

# CS210 Cheatsheet

## Logic Gates

AND  $\Rightarrow A \cdot B$

NAND  $\Rightarrow \bar{A} \cdot \bar{B}$

OR  $\Rightarrow A + B$

NOR  $\Rightarrow \bar{A} + \bar{B}$

NOT  $\Rightarrow \bar{A}$

XOR  $\Rightarrow A \oplus B$

## XOR TRUTH TABLE

A	B	RESULT
0	0	0
0	1	1
1	0	1
1	1	0

## Definitions

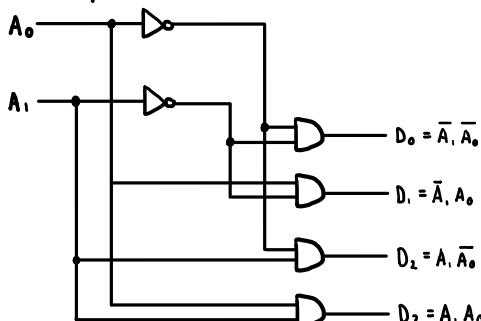
Decoders

Recognizes specific bit patterns

Multiplexors

chooses among various inputs

## Example of a decoder

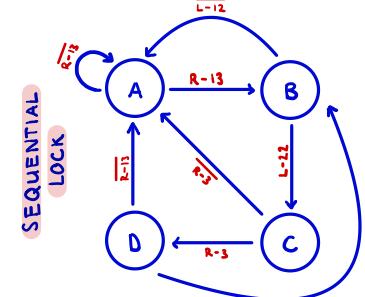


TRUTH TABLE

A <sub>1</sub>	A <sub>0</sub>	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

## State Diagrams

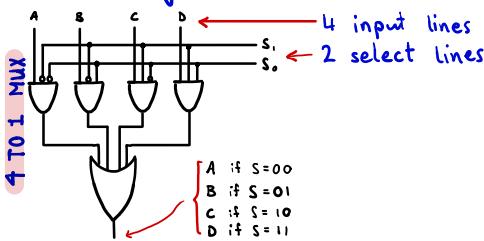
→ Show states and actions that cause transition between states



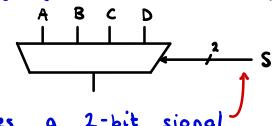
## Multiplexer (Mux)

→ A mux has  $2^n$  data inputs, n select lines, and 1 output

→ The select bits are to "choose" one of the data inputs to flow through to the output



## STANDARD SYMBOL FOR MUX



Indicates a 2-bit signal

## Instruction Processing

### Fetch

- 1) loads next instruction from memory store at address in PC and places into Instruction Register (IR)
- 2) PC is incremented to point to next instruction

### Decode

- 1) Identifies the opcode
- 2) Depending on opcode, identifies other operands from remaining bits

Evaluate Address - For operations that require memory access, compute address for access

Fetch Operands - Obtain source operands needed to perform operation

Execute - Perform the operation, using source operands

Store - Write results to destination (register or memory)

## LC3 Written as Von Neumann Model

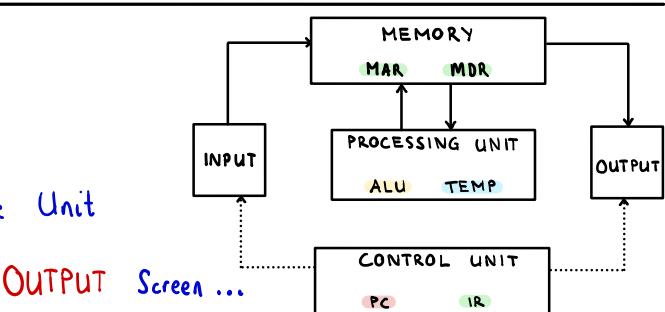
MAR Memory Address Register PC Program Counter

MDR Memory Data Register ALU Arithmetic Logic Unit

IR Instruction Registers

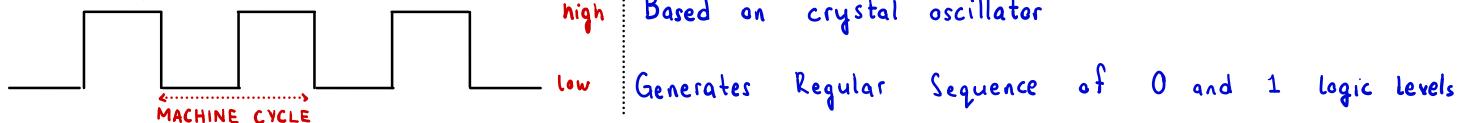
INPUT Keyboard/Mouse...

OUTPUT Screen...

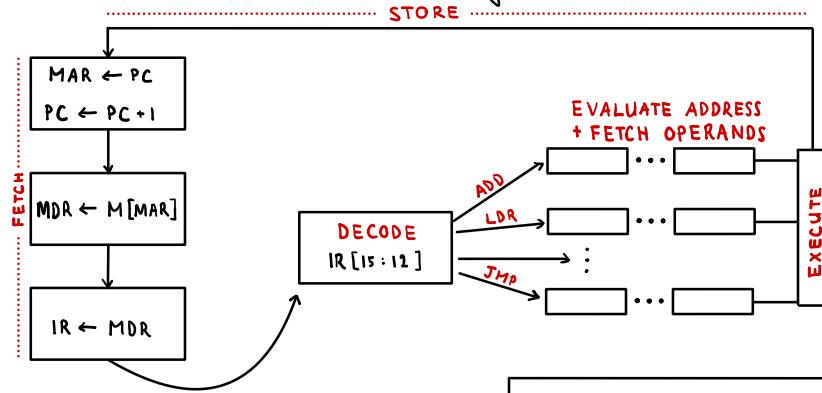


The Clock is a signal which keeps the Control Unit moving. Each clock tick moves the CU to the next step.

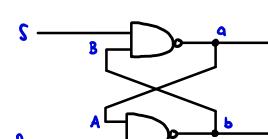
### CLOCK GENERATOR CIRCUIT



### Control Unit State Diagram



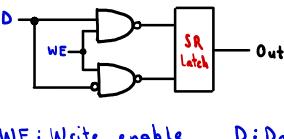
### SR - Latch



Used to store 1 bit

- Stands for Set Reset Latch
- Note: Latch is unstable if S=R=0
- Needs Active High Input

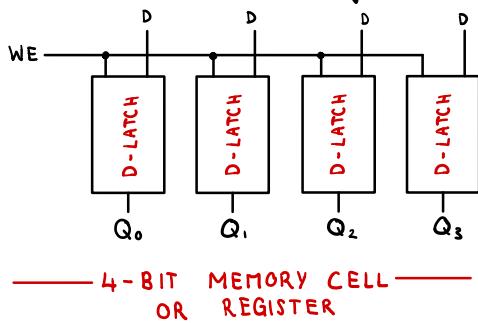
### Gated D - Latch



When WE=0, Latch holds data.

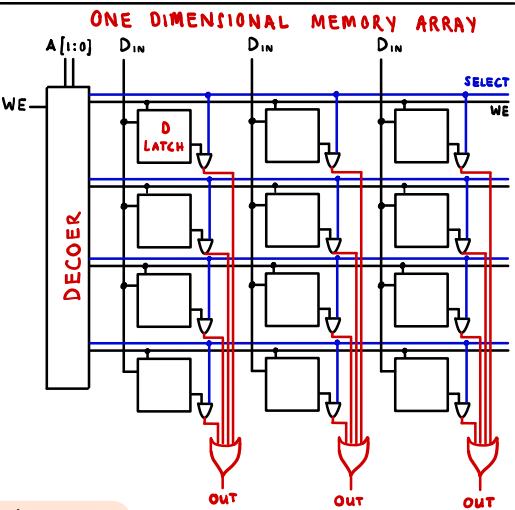
When WE=1, latch is set to value of D.

### Registers and Memory Cells

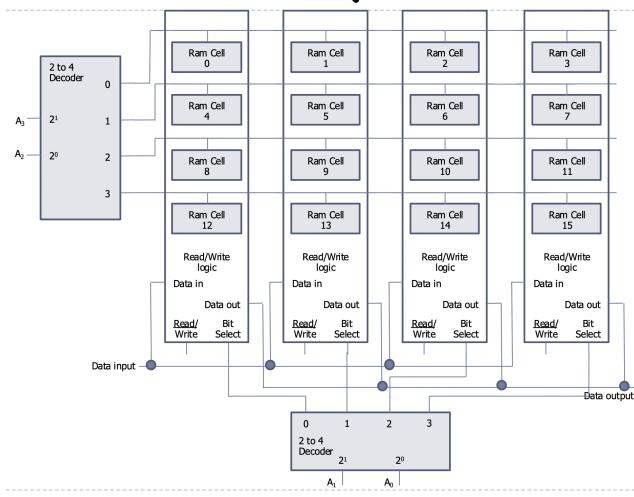


### Memory Circuits

- Bits are stored in array of D Latches
- An address is decoded to select a row
- WE specifies if we want to retrieve or store a value
- To write data, data inputs at top.
- To read data, data outputs at bottom



### 2 dimensional Memory



### Interface to Memory

- LOAD from memory
  - Write address of memory into MAR
  - Send a "read" signal to memory
  - Store contents into MDR

### STORE to memory

- Write data into MDR
- Write the address to write to in MAR
- Send "write" signal to memory, copy contents from MDR into location at MAR

### Common Trap Routines

vector symbol method

x20 GETC reads a character (no echo) into R0

x21 OUT output a character (R0) to the console

x22 PUTS write a string to console.

R0 contains pointer to string

x23 IN same as GETC but echos character

x25 HALT halts the program

### System Calls

- User program invokes system call
- Operating System code performs operation
- Returns control to user program

RTI Encoding 1000 000000000000

Operations PC  $\leftarrow$  copied back from OS stack

### TRAP Encoding Operations

1111 0000 trapvector8

- 1) PC copied to OS stack
- 2) New PC = MEM[trapvector8]

## Subroutines

- Lives in user space
- Performs well defined task
- Invoked by another program
- Returns control to main program

## Assembly Syntax

LABEL OPCODE OPERANDS ; COMMENT

optional  
mandatory  
signifies start of comment

## Assembly Keywords

.ORIG	address	specify program start address
.END		
.BLKW	n	allocate n addresses
.FILL	n	fill address with n
.STRINGZ	"string"	allocate n+1 addresses inc null

## # STACKS

R1 ← Pointer to Stack Pointer  
R7 ← Value to Push / Value from Pop

We Push and Pop to the stack using ...

JSR Pop  
JSR PUSH

## \*\*PUSH AND POP\*\*

PUSH MEM[R1] ← R6  
R6 ← R6 - 1

POP R6 ← R6 + 1  
R1 ← MEM[R6]

EMPTY Specifies initial stack pointer

## Memory Mapped Controller Registers

Location	I/O Register	
x FEO0	Keyboard Status Register	KBSR
x FEO2	Keyboard Data Register	KBDR
...	...	

Addressed from x FEO0 → x FFFF (is a privileged zone)

## Statuses

Controller 0 means controller can change value and toggle status

1 means controller can't change value

CPU 0 means CPU cannot read value

1 means CPU can read value and change status to 0

JSR

Encoding Operations

0100 1 PC offset 11

1) R7 ← PC 2) PC ← PC + PC offset

JSRR

Encoding Operations

0100 0 00 Base 000000

1) R7 ← PC 2) PC ← Base R

RET

Encoding Operations

1100 000 111 000000

PC ← R7

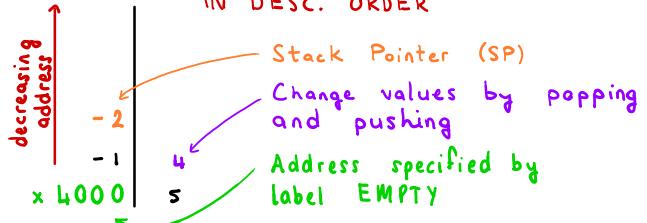
## De Morgan's Law

$$\text{OR FORM} \quad \overline{AB} = \overline{A} + \overline{B}$$

$$\text{AND FORM} \quad A + B = \overline{\overline{A} \cdot \overline{B}}$$

## Visualising Stacks

THIS COURSE USES EMPTY STACK CONVENTION IN DESC. ORDER



note: popping from stack doesn't delete value, it just changes stack pointer

## Overflow and Underflow

Stack Range: x 4000 → x 3 FFC

Underflow SP > x 4000

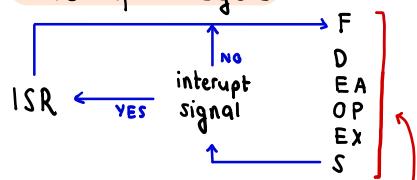
Overflow SP < x 3 FFC

## I/O Controller

- Provides necessary interface for I/O devices
- Takes care of low level, device dependant details
- It buffer data sent to processor so devices remain in sync



## Interrupt Cycle



Fetch Decode Execute Cycle

ISR Interrupt Service Routine

## Polling

```
# HOW TO USE CONTROLLER REGISTERS IN CODE
POLL    LDI R1, KBSRPtr
        BRzp POLL
        LDI R0, KBDRPtr
        ...
KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

### \*\*To Note:\*\*

- Polling is a waste of cycles

## Interrupt Service Routine

When an external device needs handing ...

- 1) The current program is stopped by the OS and the state is saved.
  - 2) The Interrupt Service Routine is run, satisfying the I/O devices needs.
  - 3) The programs state is restored and gains control over CPU.
- Interrupts are assigned a priority, P0 → P6, with 6 being the highest
- Check page before for Interrupt Cycle

## Finite State Machine (control unit)

On each machine cycle FSM changes control signals for next phase of instruction processing. Like ...

- What component drives the bus
- Which registers are write enabled
- Which operation should the ALU perform

## Global Bus

Set of wires that allow various components to transfer 16 bit data to other components.

One or more components may read data from the bus on any cycle

## LC3 Data Path

### # C Programming Notes

#### \*\*Pointers\*\*

- `\*`
  - Used to declare a variable is a pointer to another variable
    - e.g. `int *p = &a` | meaning p stores the memory address of a
  - Used to deobfuscate a pointer
    - e.g. `*p = 12` | meaning the store 12 in the memory address stored in p
- `&`
  - Returns the memory address of a variable
    - e.g. `printf("%p\n", &a);` | prints something like 0x3005

#### \*\*Formatting Strings\*\*

- `%d` or `%i` : Signed decimal integer
- `%u` : Unsigned decimal integer
- `%f` : Floating point number
- `%c` : Single character
- `%s` : String (null-terminated character array)
- `%p` : Pointer address
- `%x` or `%X` : Hexadecimal integer (lowercase or uppercase)

## Bitwise Operations

Operation	Operator	Examples
AND	&	a & b
OR		a   b
NOT	~	a ~ b
XOR	^	a ^ b
LEFT SHIFT	<<	a << b
RIGHT SHIFT	>>	a >> b

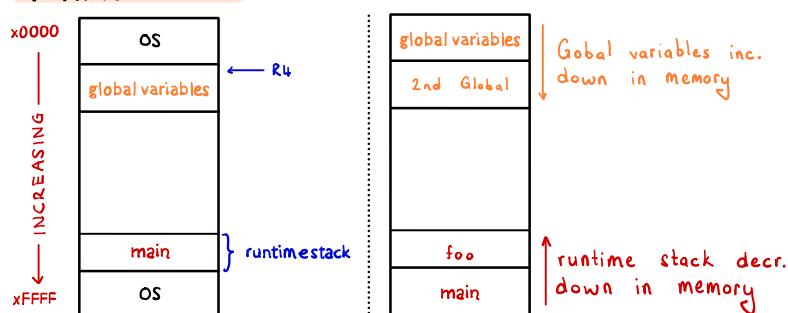
## Scope and Symbol Table

Global can be accessed anywhere in the program.

Local only accessible in a particular region.

SYMBOL TABLE CONDENSED			
Name	Type	Offset	Scope
x	int	0	global
y	int	1	global
z	int	2	global
a	int	0	local
b	int	1	local
c	int	2	local

## Runtime Stack



## LC3 Register Conventions

Register	Conventional Role
R4	Pointer to global data
R5	Frame Pointer (base of stack frame)
R6	Stack Pointer (top of stack)
R7	Link Register (return address from JSR)
PC	Program Counter

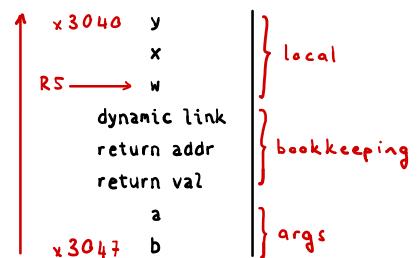
## Runtime Stack

- Runtime Stack is comprised of stack frames.
- Local variables are stored in stack frames
- A stack frame is an activation record on runtime stack
- When a new function is called, its activation record is pushed on the stack.

## Activation Record

```
int function(int a, int b) {  
    int w, x, y;  
    ...  
    return y;  
}
```

Name	Type	Offset	Scope
a	int	4	function
b	int	5	function
w	int	0	function
x	int	-1	function
y	int	-2	function



Return Value Space for value returned by function

Return Addr Save pointer to next instruction in calling function

Dynamic Link Callers frame pointer. Used to pop this activation record from stack.

## # Caching

### 1) Directly mapped vs Fully Associative

- Directly Mapped
  - \*Mapping:\* Address `index` maps to 1 specific cache slot using a hashing algo
  - \*Pros:\* Simple, and fast
  - \*Cons:\* High conflict misses (two active blocks mapping to same slot causing frequent evictions)
- Fully Associative Cache:
  - \*Mapping:\* Address can go into ANY cache slot. Usually placed consecutively
  - \*Pros:\* Very low conflict misses, high hit rate.
  - \*Cons:\* Complex/expensive, impractical for large caches
- Set-Associative Cache
  - \*Mapping:\* Address `set index` maps to two possible slots
  - \*Balance:\* Combines aspects of both
  - \*Pros:\* Good balance of performance. Reduced conflict misses and reduced expensive operations

### 2) Types of cache misses

- \*Compulsory:\* is when first loading data
- \*Capacity:\* in that cache is smaller than main memory
- \*Conflict:\* when two or more addresses map to the same slot

### 3) Cache Eviction (LRU)

- When we need to push data to cache and slot is full, we evict data that is least recently used (LRU)

### 4) Writing Policies

- \*Write-Through\*: Data is written to **both** the cache and main memory **simultaneously**.
- \*Write-Back\*: Data is written **only to the cache initially**. The cache line is marked as "dirty". Main memory is updated **only when the dirty cache line is evicted** (replaced) or explicitly flushed.