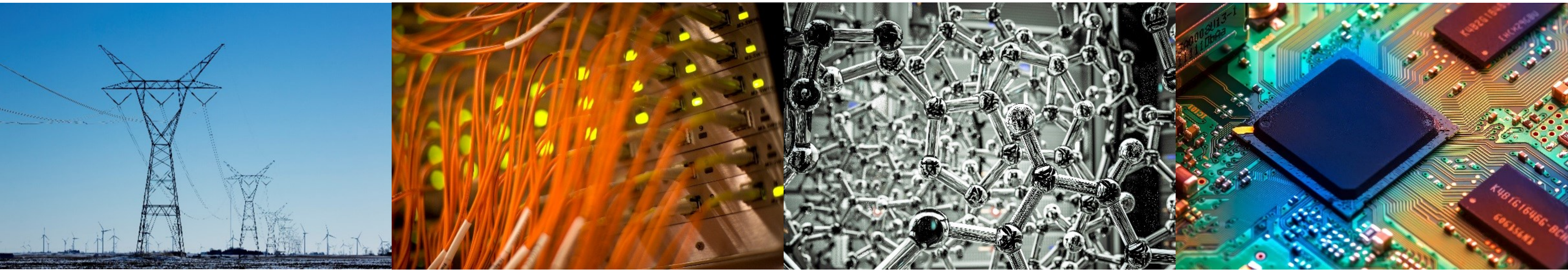
ECE 120 Final Exam Review

HKN Review Session



ILLINOIS

Electrical & Computer Engineering

GRAINGER COLLEGE OF ENGINEERING

Representations of Bits

- Representations map sequences of bits into meaningful information
  - There must be no ambiguity for any bit pattern
- Many possible representations, but some are better than others
- Without knowing the representation, bits are meaningless

Unsigned Representation

- Sequentially maps base 2 numbers to base 10 numbers
- **Range:** 0 to 2^k-1 for ***k*** bits
- **Overflow Conditions:** Most Significant Bit (MSB) has a carry out
- **Extending:** Pad the left with zeros

Sign-Magnitude Representation

- MSB is the sign bit, followed by the magnitude
  - MSB is 1 for negative numbers, or 0 for positive numbers
- **Range:** $-2^{k-1} - 1$ to $2^{k-1} - 1$ for ***k*** bits
- Modern computers don’t use this anymore …
  - Two possible ways to represent zero (0 or 1 followed by a zero magnitude) – wastes bits
  - Requires separate logic to perform arithmetic

Two’s Complement Representation

- **Range:** -2^{k-1} to $2^{k-1} - 1$ for ***k*** bits
- **Overflow Conditions:** Adding two numbers with the same MSB yields a result with a different MSB
- **Extending:** Pad the left with value of the MSB
- Uses the same logic as the unsigned representation does for arithmetic
- Negative when the MSB is 1

Two’s Complement Representation (cont.)

Decimal to 2’s Comp.

Positive Numbers:

1. Treat as an unsigned number and convert to binary

Negative Numbers:

1. Convert the magnitude as an unsigned number
2. Negate and add one

2’s Comp. to Decimal

MSB is 0 (Positive):

1. Treat as an unsigned binary number and convert to decimal

MSB is 1 (Negative):

1. Negate and add one to obtain the magnitude
2. Convert the magnitude as an unsigned number
3. Add a negative sign to the magnitude

Fun Fact: First negating a number then adding one produces the same result as first subtracting one then negating!

Unsigned & Two’s Complement Arithmetic

Unsigned Addition:

$$\begin{array}{ccccc} & 0 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline \boxed{1} & \boxed{0\quad 0\quad 0\quad 1} \\ & \text{Result} \end{array}$$

Overflow when there is a carry out

Two’s Complement Addition:

Even though there is a carry out, there is no overflow here (3 + -2 = 1)

$$\begin{array}{ccccc} & 0 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline \boxed{1} & \boxed{0\quad 0\quad 0\quad 1} \\ & \text{Result} \end{array}$$

There is only overflow when the MSB’s of the addends is different from the MSB of the sum

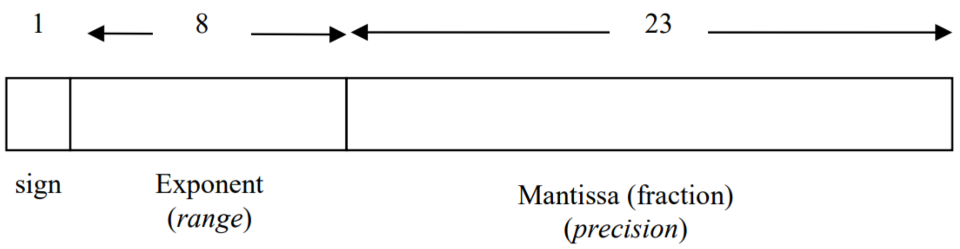
$$\begin{array}{ccccc} & 1 & 0 & 0 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 \\ & \text{Result} \end{array}$$

Overflow!

ECE ILLINOIS

IEEE 754 Floating Point Representation

- Approximates decimals and very large or small numbers
- **Conversion:** $(-1)^{\textit{sign}} \times 1.\textit{mantissa} \times 2^{(\textit{exponent} - 127)}$
- Special Meanings
  - Exponent is 255: +/- infinity (depending on sign bit) when mantissa is 0, otherwise NaN (not a number) when mantissa is non-zero
  - Exponent is 0: Denormalized number (leading 1 before mantissa is replaced with a 0)



IEEE 754 Floating Point to Decimal Conversion

Convert: $\boxed{1}$ $\boxed{1000\ 0101}$ $\boxed{1110\ 0011\ 0000\ 0000\ 0000\ 000}$

Sign Exponent Mantissa

$$(-1)^{1} \times 1.11100011000000000000000 \times 2^{(133-127)}$$

$$-1 \times 1111000.11000000000000000$$

$$-1 \times (2^{7} + 2^{6} + 2^{5} + 2^{4} + 2^{-1} + 2^{-2})$$

-120.75

Decimal to IEEE 754 Floating Point Conversion

Convert: 3481.375

$3481_{10} = 110110011001_2.$

Exponent is 2^{11}

$11 + 127 = 138$

$138_{10} = 1000\ 1010_2$

$0.375 \times 2 = 0.75$ (< 1, so the bit at 2^{-1} is 0)

$(0.75 - 0) \times 2 = 1.5$ (≥ 1, so the bit at 2^{-2} is 1)

$(1.5 - 1) \times 2 = 1$ (≥ 1, so the bit at 2^{-3} is 1)

$1 - 1 = 0$ (Done!)

Keep going until you get a remainder of zero or you run out of bits …

Decimal to IEEE 754 Floating Point Conversion

Convert: 3481.375

Putting it together …

From 3481_{10} with the left-most 1 removed (since there is an implicit 1)

From our repeated division, right-padded with zeroes since we reached a zero remainder.

Sign	Exponent	Mantissa
0	10001010	10110011001 011000000000

Hexadecimal Representation

- Each hexadecimal character is four bits
- Remember to prefix your hexadecimal answers with “x”
- Having the hexadecimal to binary chart on your cheat sheet (or memorizing it) can save time and prevent mistakes on the exam

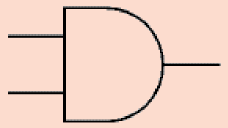
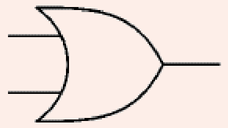
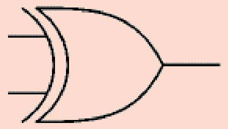
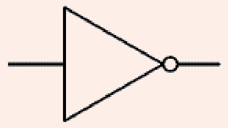
Hex:	x0	x1	x2	x3	x4	x5	x6	x7
Binary:	0000	0001	0010	0011	0100	0101	0110	0111

Hex:	x8	x9	xA	xB	xC	xD	xE	xF
Binary:	1000	1001	1010	1011	1100	1101	1110	1111

ASCII Representation

- American Standard Code for Information Interchange
- Each ASCII character is 7 bits, but usually stored in a byte (leading bit is zero)
- You will be given an ASCII chart on the exam if needed – no need to memorize anything
- Common mistake: numbers in ASCII are not the same as in hex or decimal (e.g. “3” in ASCII is NOT x03 in hex)!

Boolean Logic

Operation	Expression	Symbol	Truth Table
AND	AB		
OR	A + B		
XOR	A ⊕ B		
NOT	A’		

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

A	A’
0	1
1	0

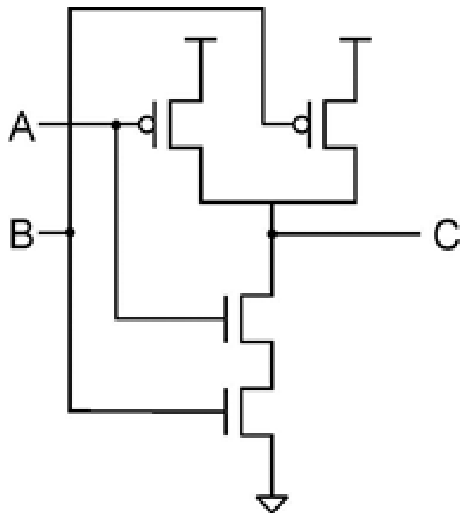
Bitmasks

- Quickly isolate or modify only certain bits in a group

	Isolating a group of bits	Set bit to 0	Set bits to 1	Flip bits
Data:	1 0 1 0 0 1 1 0	1 0 1 0 0 1 1 0	1 0 1 0 0 1 1 0	1 0 1 0 0 1 1 0
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Operation:	AND	AND	OR	XO R
Bitmask:	0 0 0 0 0 1 1 1	1 1 1 1 1 0 1 1	0 0 0 0 1 1 0 0	1 0 0 0 0 0 0 1
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Result:	0 0 0 0 0 1 1 0	1 0 1 0 0 0 1 0	1 0 1 0 1 1 1 0	0 0 1 0 0 1 1 1
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0

CMOS logic

- Metal Oxide Semiconductor Field Effect Transistors (MOSFET)
  - N-type: pull-down, voltage applied (1) means it conducts, voltage absent (0) means it is off
  - P-type: pull-up, voltage applied (1) means it is off, voltage absent (0) means it is on
- Complementary layout
  - Prevents floating / short-circuited outputs
  - Requires p-type on top (connected to V_{dd}), requires that the top is the dual of the bottom



NAND Gate

Boolean Expressions and Algebra

	* indicates AND	+ indicates OR
identity	1 * A = A	0 + A = A
null	0 * A = 0	1 + A = 1
idempotence	A * A = A	A + A = A
complementarity	A * A’ = 0	A + A’ = 1
DEMORGAN	(A + B)’ = A’B’	(AB)’ = A’ + B’
involution	(x’)’ = x	
Distribution	(A + B)C = AC + BC	AB + C = (A + C)(B + C)
Consensus	(A + B)(A’ + C)(B + C) = (A + B)(A’ + C)	AB + A’C + BC = AB + A’C

Duality Principle, Ex. Demorgan's Law

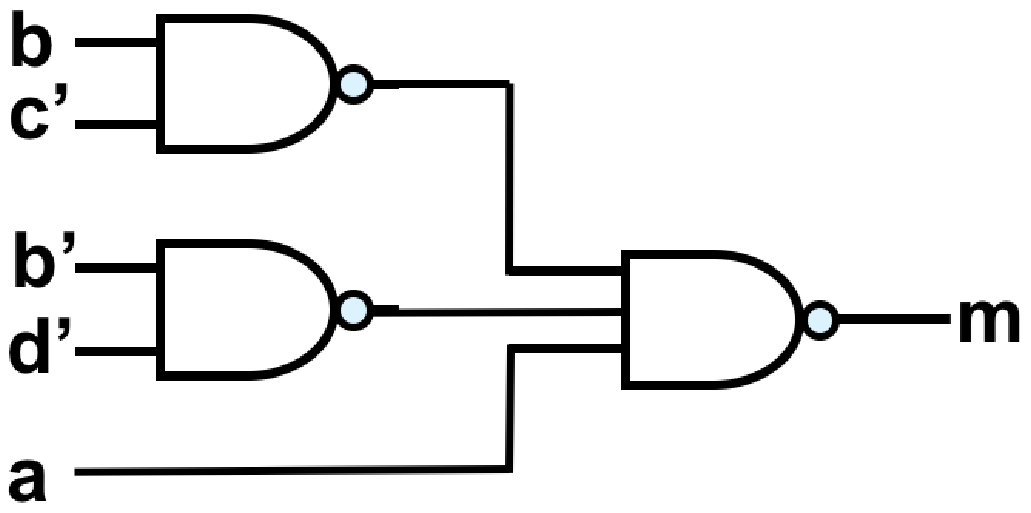
Duality: swap 1's and 0's, swap AND's and OR's, maintain precedence using parentheses

Complement: find dual, complement each variable

m(a,b,c,d) = a' + bc' + b'd'

Dual: n(a,b,c,d) = a'(b+c')(b'+d')

1) m'(a,b,c,d) = (a' + bc' + b'd')'
2) m'(a,b,c,d) = a(bc')'(b'd')'



Logical Equivalence

2 ways to prove that two boolean functions are equivalent

1. Use truth tables: if they have the same output over all input combinations then they are logically equivalent
2. Use boolean algebra properties to show equality.

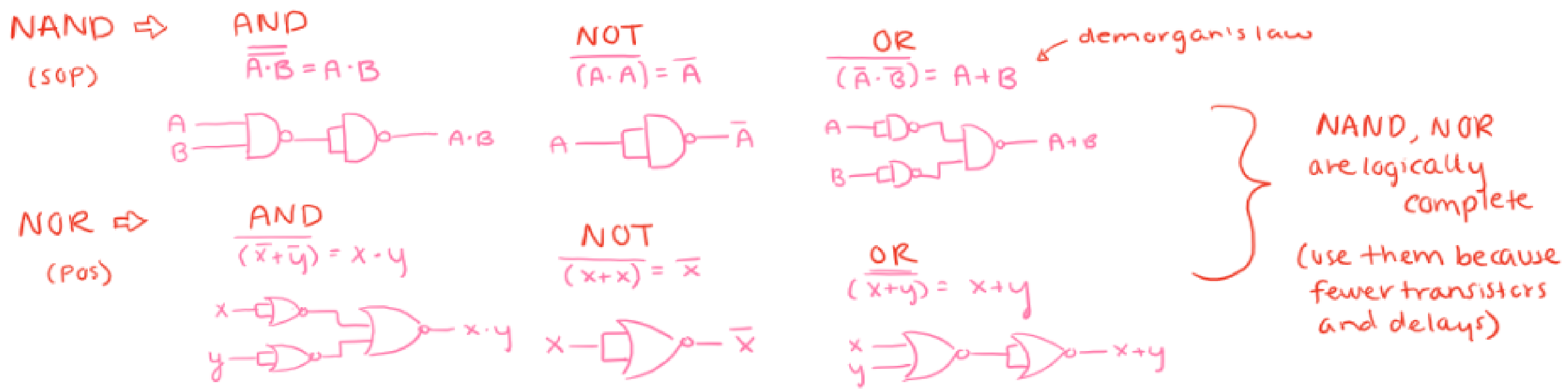
Let’s prove deMorgan’s law:

(A + B)’: So, we can see in this truth table that (A+B)’ is logically equivalent to A’B’. QED.

A	B	(A+B)’	A’B’
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Logical Completeness

- A set of gates is said to be logically complete if it can implement any boolean expression with exclusively those set of gates.
- For example, NAND and NOR are individually logically complete as well as AND, OR, and NOT together.
  - How?
  - What about NOT? Is NOT logically complete?


NAND ⇨ (SOP)
AND
$\overline{\overline{A \cdot B}} = A \cdot B$
NOT
$\overline{(A \cdot A)} = \overline{A}$
OR
$\overline{(\overline{A} \cdot \overline{B})} = A + B$
demorgan's law
NOR ⇨ (POS)
AND
$\overline{(\overline{x} + \overline{y})} = x \cdot y$
NOT
$\overline{(x + x)} = \overline{x}$
OR
$\overline{\overline{(x + y)}} = x + y$
NAND, NOR are logically complete
(use them because fewer transistors and delays)

K(arnaugh)-maps

- A way to represent the outputs of a logical design with respect to its inputs
- Should arrange inputs along row and columns in gray code order.
    - 00   01   11   10
- Should be able to form whether asked to from a circuit, problem statement, or a design goal.
- Can accommodate up to 4 inputs, more requires more than a 2d representation.
- Simple example for a 3-input AND, f(A,B,C):

A, BC->	00	01	11	10
0	0	0	0	0
1	0	0	1	0

SOP, POS

- SOP- Sum of Products
  - A group of products (ANDs) which are being summed (OR’d).
  - Example: f(A,B) = AB’+A’B   (XOR)
  - AB’ and A’B are the products, which are summed.

- POS- Product of Sums
  - A group of sums (ORs) which are being multiplied (AND’d).
  - Example: f(A,B) = (A+B)(A’+B’)   (Still XOR)
  - (A+B) and (A’+B’) are the sums, which are AND’d.

Minimal vs. Canonical

- Minimal - Listing terms as efficiently as possible (no redundancy)
  - Can find through the bubbling method on kmaps.
  - Example: f(A,B,C) = A+BC

- Canonical - Listing every term of the function with every input.
  - Can find through writing out a truth table.

  - Canonical SOP: A listing of every set that makes the function return true (OR'd)
    - Example for above f: f(A,B,C) = ABC+ABC'+AB'C+AB'C'+A'BC

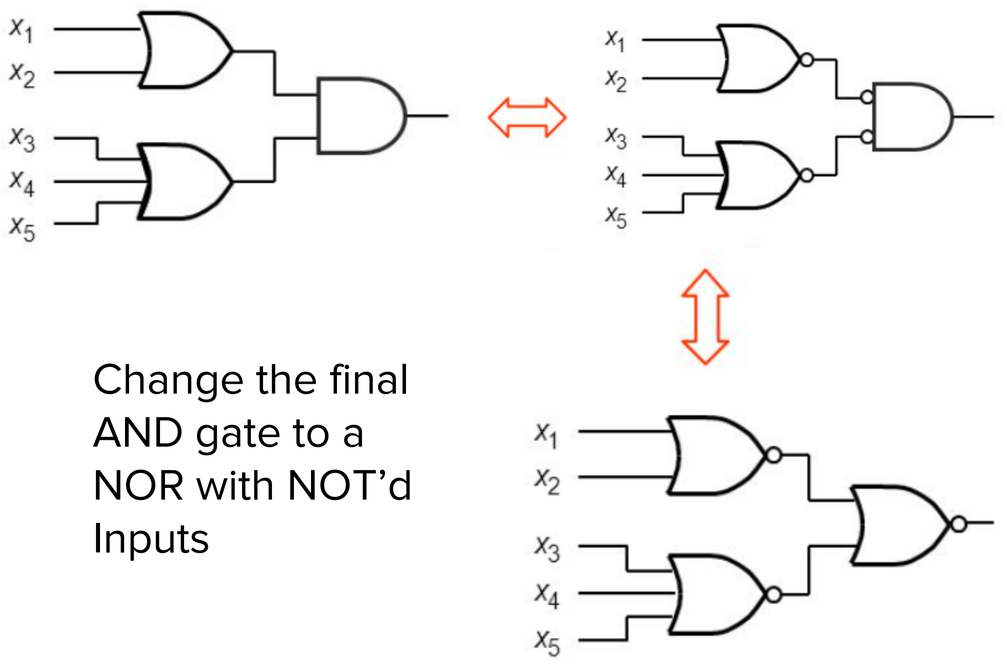
  - Canonical POS: A listing of every set that makes the function return false
    - Example for above: f(A,B,C) = (A+B+C)(A+B+C')(A+B'+C)

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

Converting SOP/POS to 2-level NAND or NOR

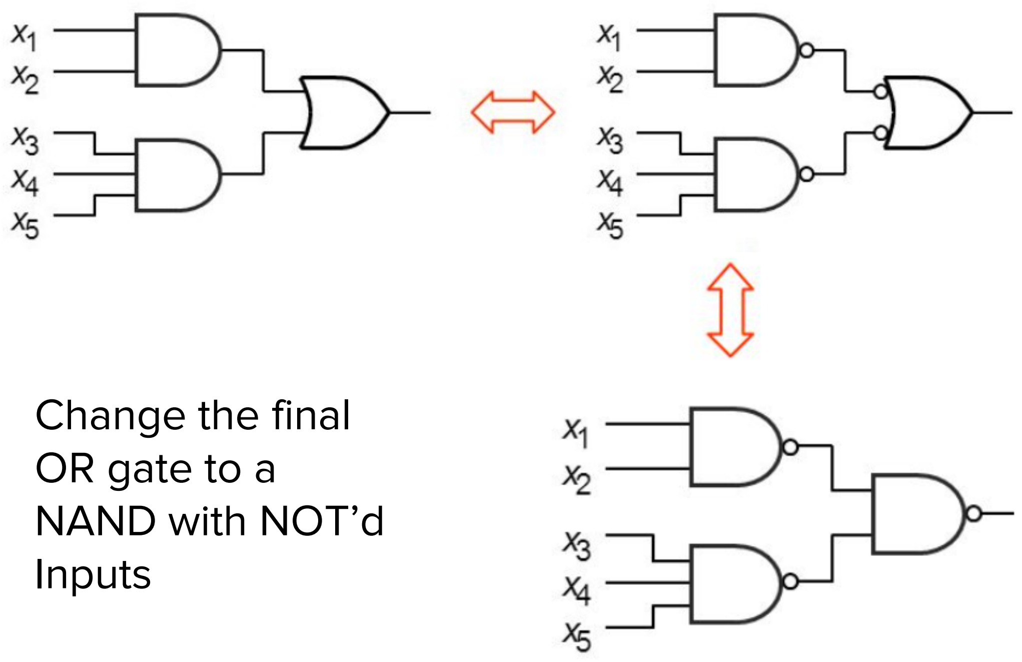
Any 2 level POS or SOP circuit can be converted quickly to a NOR-NOR or NAND-NAND implementation.

POS



Change the final AND gate to a NOR with NOT'd Inputs

SOP

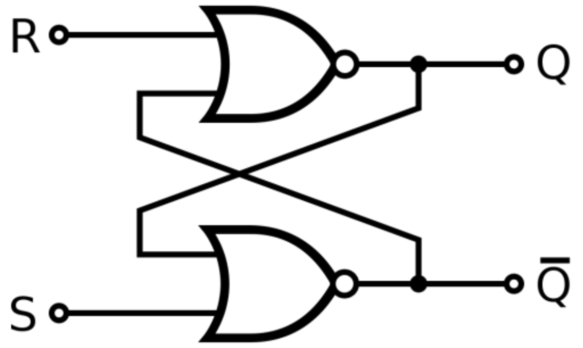


Change the final OR gate to a NAND with NOT'd Inputs

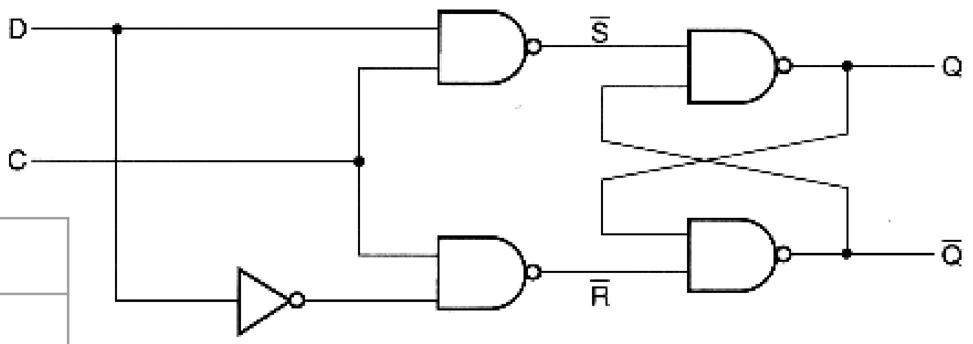
Optimized Boolean Expressions

- When trying to find the Boolean expression for the outputs of a function
  - consider both the POS and SOP forms
  - Check which of these minimizes the total number of gates
- Prime implicants
  - The largest grouping of 1’s not fully contained by another grouping of 1’s (with power of 2 dimensions, ie. 2,4,8)
- Uniqueness
  - Look for other prime implicants of equal size that are not currently being used in your minimal expression
  - If you can write another function with the same number of terms, then it is not unique
- Don’t Cares
  - Symbol: X, used if the output can be a 0 or a 1
  - Can be included in any kmap circling
  - Important for making simplified SOP/POS (allows larger implicants)

Latches (S-R & D)



R	S	Q^+
0	0	Q
0	1	1
1	0	0
1	1	Invalid



(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

This SR latch above is active High for S (set) and R (reset)

functionality. This SR latch is made from 8 transistors

C is the enable bit for

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Nor Truth Table

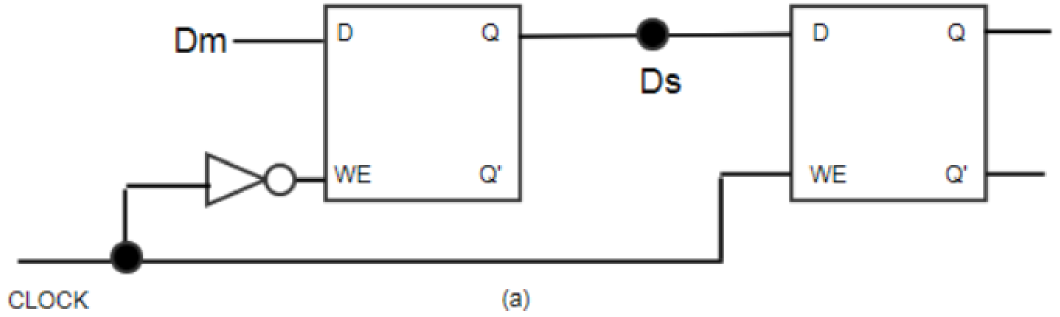
This D - latch is made from 18 transistors

How to build Flip-Flops from latches

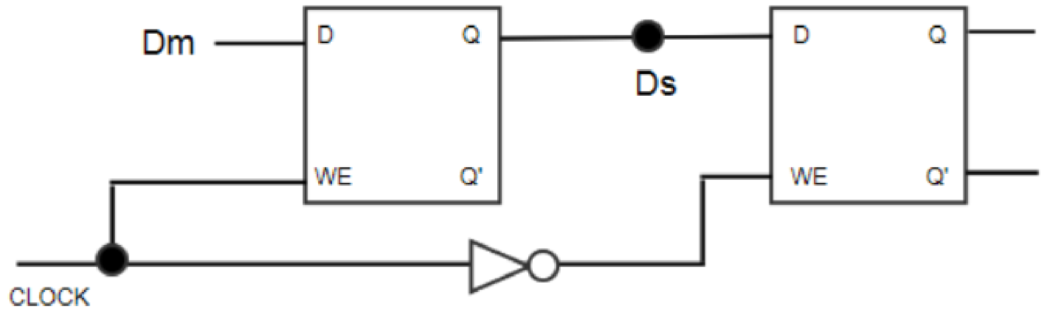
- Rising edge triggered flip flop on top, Falling edge triggered flip flop on the bottom.

- Looking at the top picture we have a rising edge triggered flip flop. How?

- When the clock is 0, master latch is write enabled, slave latch is write disabled. This way once the edge rises, the master latch becomes write disabled storing/holding the value of the input Dm from right before the rising edge. The WE of the slave (second) latch goes high allowing the value stored in the master to be written to and stored into the slave latch.



(a)



(b)

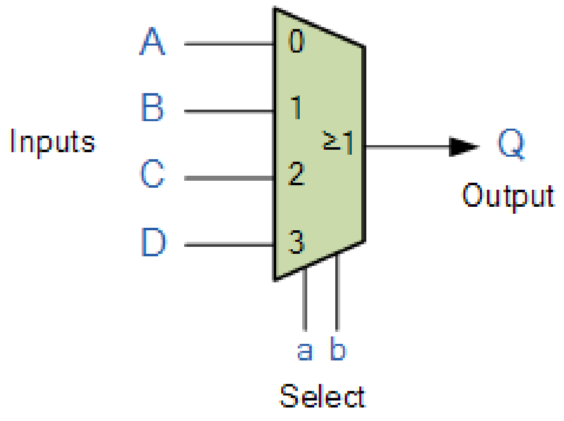
Multiplexers and Decoders

Multiplexers:

n select bits

2^n input options

1 output



Decoders:

1 enable bit

n select bits

(input lines specify which output to raise)

2^n possible outputs (one high at a time)

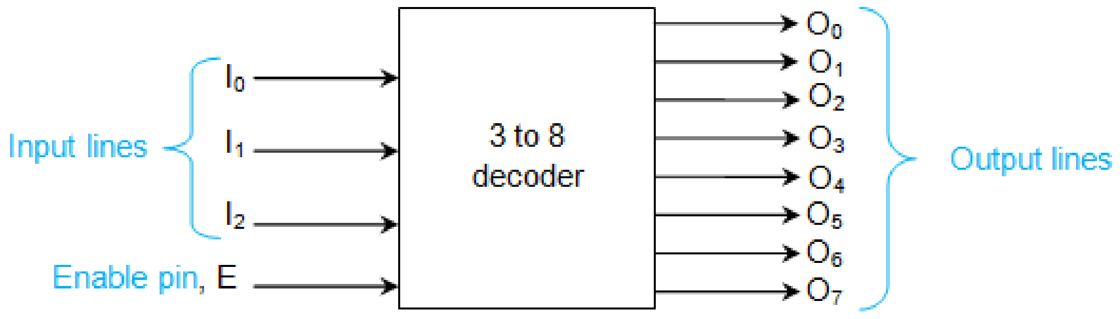
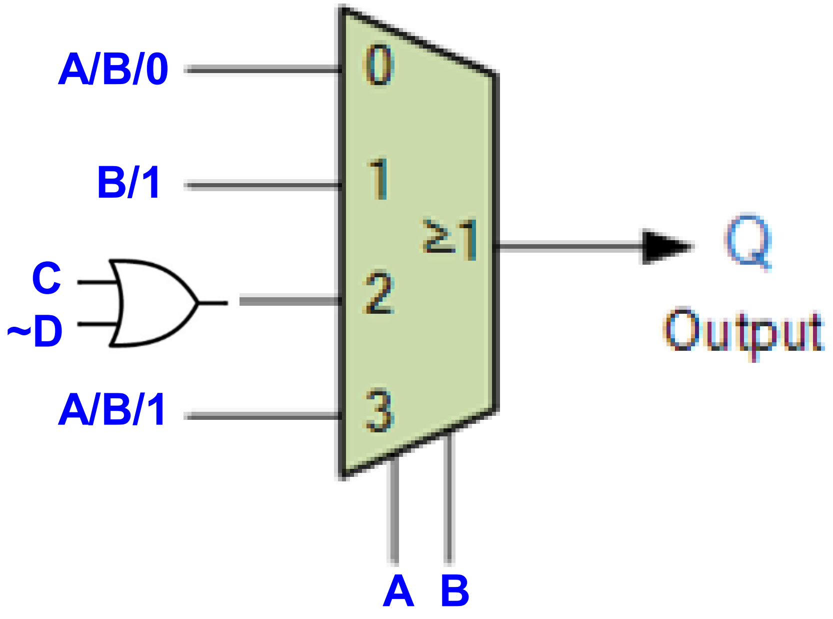


Figure 1 3 to 8 binary decoder

Multiplexers and Decoders

Implement g(a,b,c,d)=ab+a'b+ab'c+ab'c'd' using a **4:1 MUX** and no more than **one extra**

a	b	c	d	g
T	T	T	T	T
T	T	T	F	T
T	T	F	T	T
T	T	F	F	T
T	F	T	T	T
T	F	T	F	T
T	F	F	T	F
T	F	F	F	T
F	T	T	T	T
F	T	T	F	T
F	T	F	T	T
F	T	F	F	T
F	F	T	T	F
F	F	T	F	F
F	F	F	T	F
F	F	F	F	F



Adders and Bit Slice Design

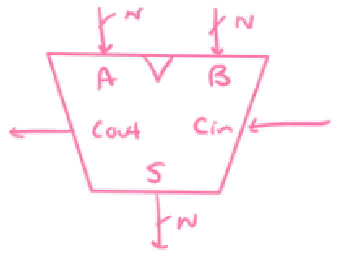
Bit Slice Adder Circuit: broken down into repeated operations on individual bits, this design extends to any # of bits bc slices pass information between them

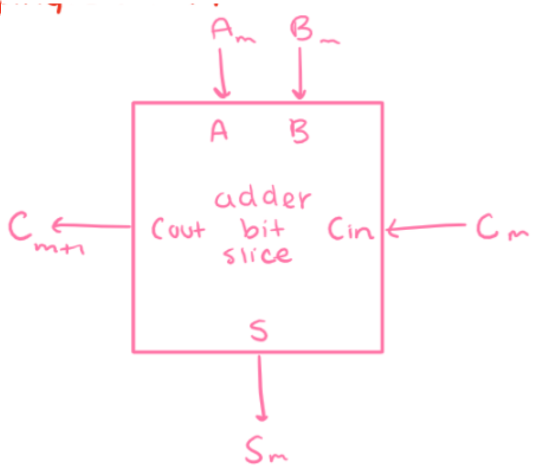
$$\begin{array}{rl} C & c_3 c_2 c_1 c_0 \\ A & a_3 a_2 a_1 a_0 \\ + \; B & b_3 b_2 b_1 b_0 \\ \hline S & S_3 S_2 S_1 S_0 \end{array}$$

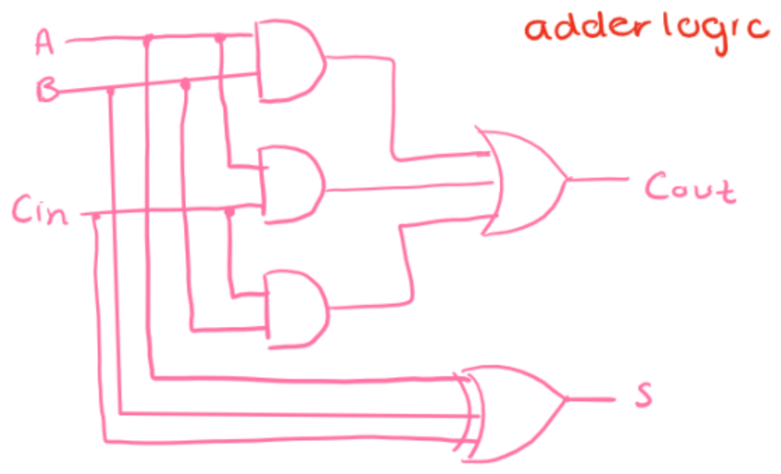
← 3inputs b/c carryout

must perform column addition

symbol

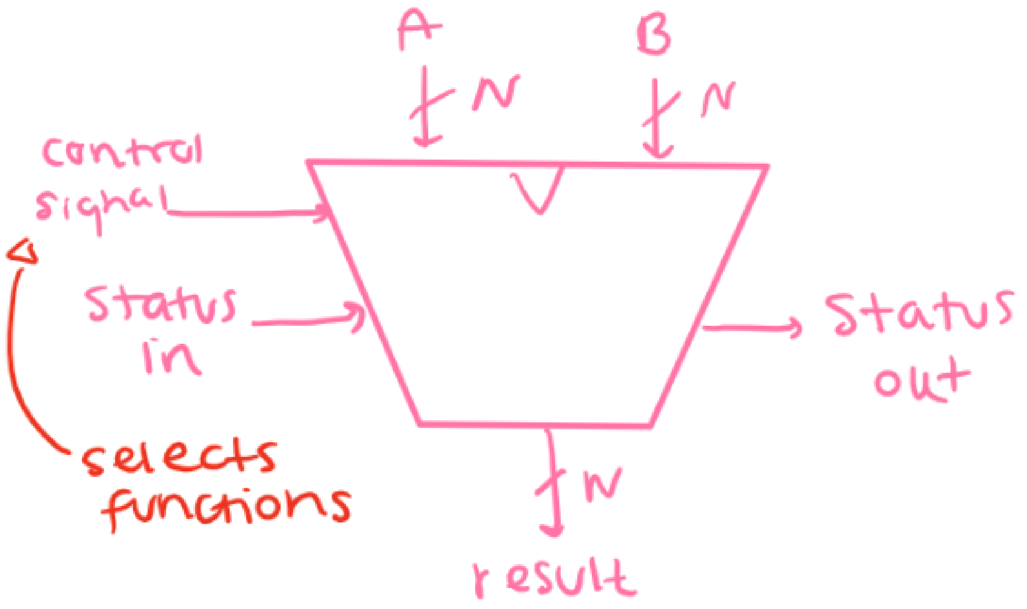






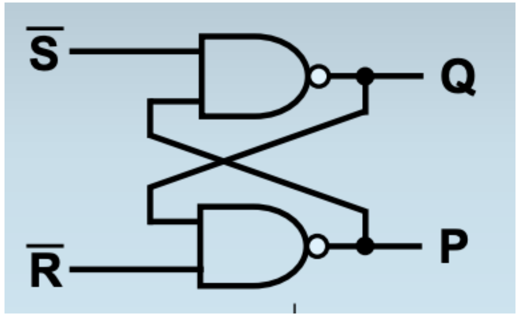
ALUs

Combinational logic circuit that performs arithmetic & logic operations on integers



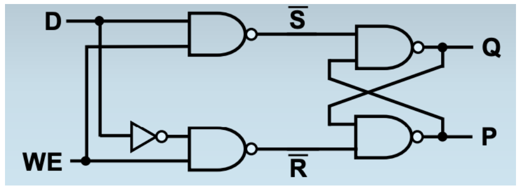
S'-R' Latch

- If S = R, then Q and P can't store a bit.
- When S' is 0 and R' is 1, stores 1 into Q.
- When S' is 1 and R' is 0, stores 0 into Q.

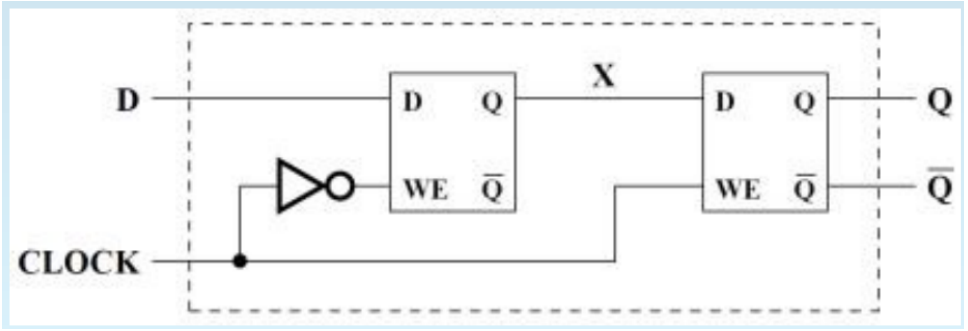


D latch

- D stands for the DATA, WE stands for Write Enable
- Stores the value of D when WE is high.



D Flip Flops



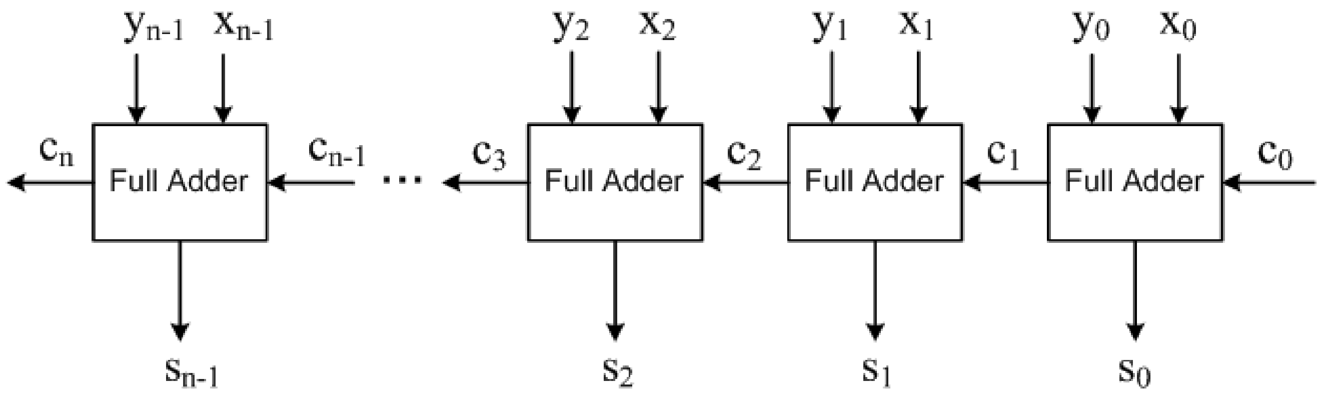
- Made from two D latches
- D latch behavior:

C	D	R	S	Q+	Definition
1	0	1	0	0	Reset
1	1	0	1	1	Set
0	0	0	0	Q	No change
0	1	0	1	1	No change

- D flip flop stores the input bit D when clock goes low to high

Bit Slice Design

- Design 1 component for a single bit
- Repeat it k times for a k-bit design

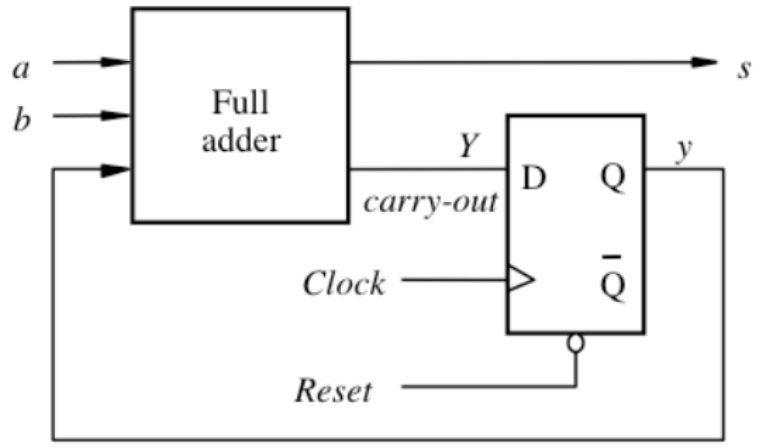


Example of bit slice design: Ripple Carry Adder

Serialized Design

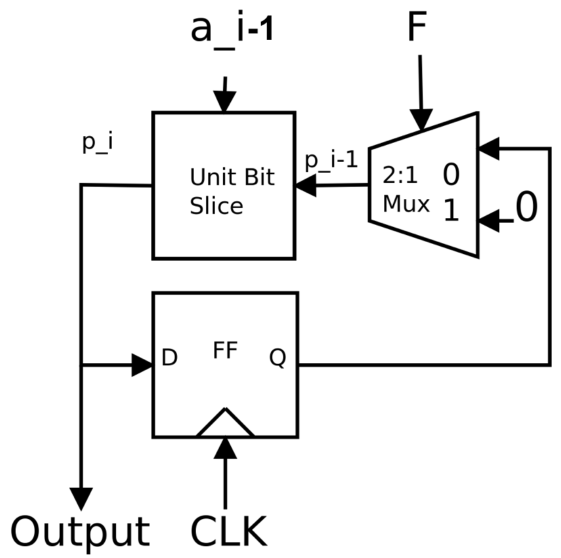
- Same as a bit slice design, except instead of many bit slices only use 1
- Use flip-flops to store intermediate results
- Compute each bit slice in sequence, over time

Serial Adder



Serialized Design

- The ‘F’ Signal is 1 when the first bit of data enters, 0 otherwise
- Use the ‘F’ signal to initialize your registers to the correct value
  - In the full adder example, this means load the register with carry in of 0

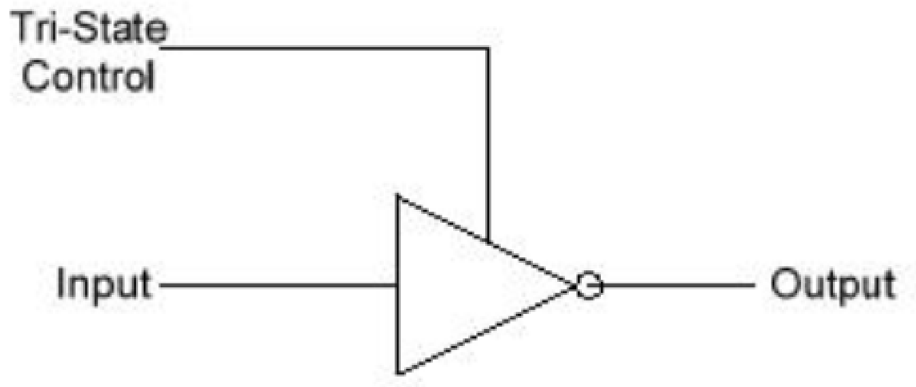


Bit Slice VS Serial

- Area
  - For large number of bits, serial uses less area
  - For smaller number of bits, bit slice uses less area
- Speed
  - Bit slice is usually faster
  - Serial is usually slower

Tri-State Buffers

- When enable is 1, they act as a wire
- When enable is 0, they act as high impedance (disconnected)



Finite State Machine

3 Main Components of FSMs:

- Next state logic
- Current state
- Output logic

How many bits are needed to represent N states in a standard FSM?

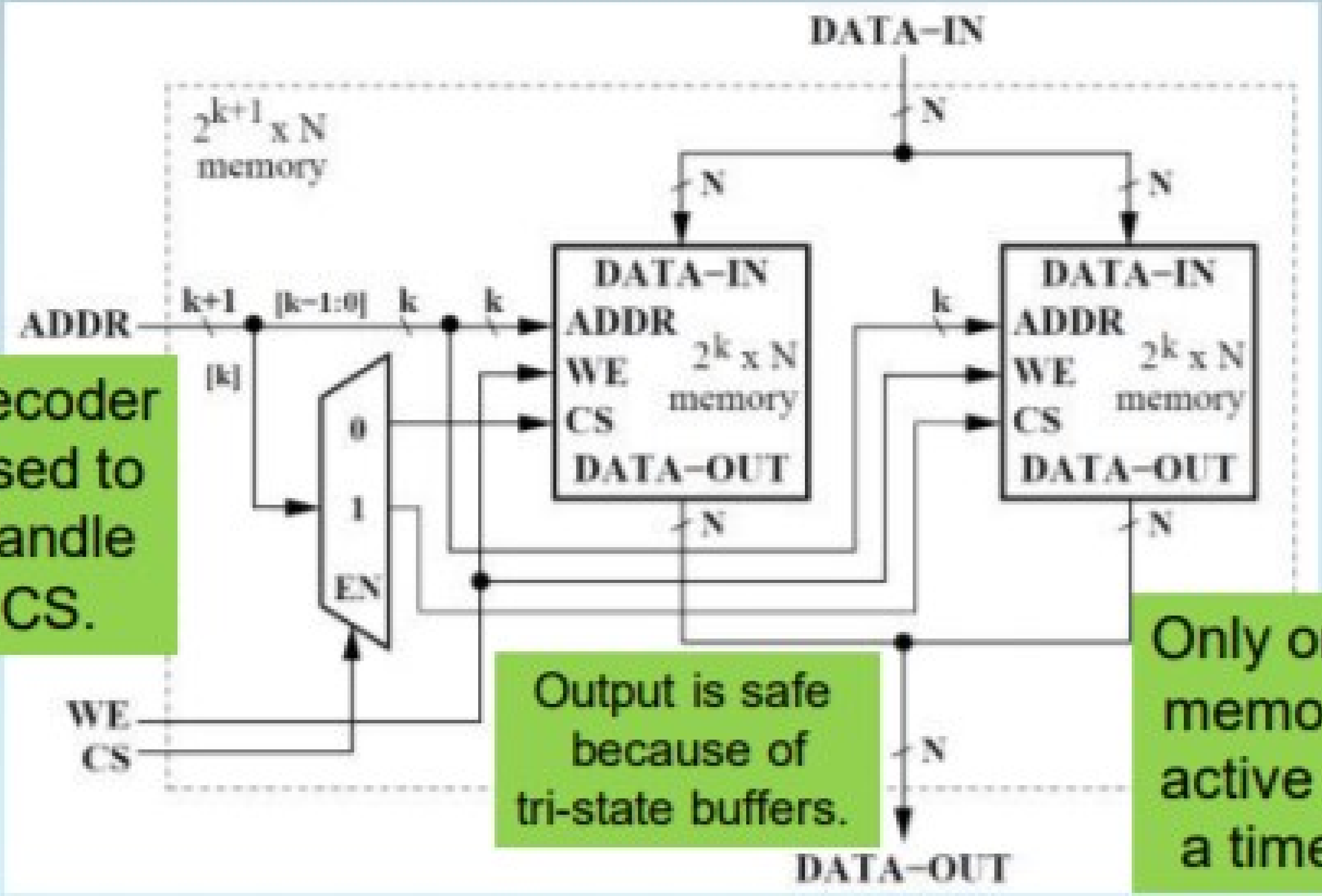
FSMs In Detail

- 2 types: Mealy and Moore
  - Moore: Outputs are a function of current state **only**
  - Mealy: Outputs are a function of current state and inputs
  - Mealy machines use less states
- 5 key elements:
  - Finite number of states
  - Finite number of inputs
  - Finite number of outputs
  - Explicit number of state transitions
  - Explicit definition of outputs as a function of state (and as function of inputs for Mealy machines)

FSM Design

1. List all inputs to your FSM
2. List all outputs from the FSM
3. Create a truth table for the FSM
    a) State and inputs on the left
    b) Next state and outputs on the right
    c) Outputs should be the same for the same current state (unless Mealy machine)
4. Create a state diagram
5. K-maps, etc.

Building a Memory with More Addresses



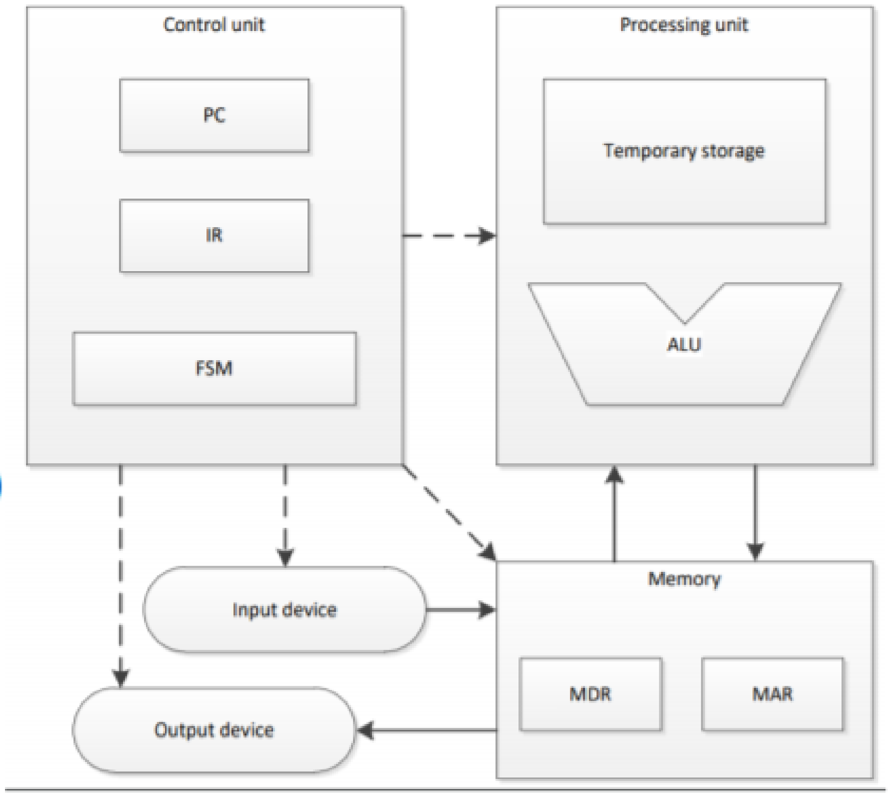
Decoder used to handle CS.

Output is safe because of tri-state buffers.

Only one memory active at a time.

LC3: A Brief Overview

- 16 Bit Data
- 16 Bit Address (coincidence)
- 8 Registers (R0-R7)
- Memory and Mem. Interface
- MAR (Accessing addresses)
- MDR (Accessing actual data)
- Input (KBSR, KBDR)
- Output (DSR, DDR)
- PC and IR



Operations in LC3

Operations:
ADD, **AND**, **NOT**

Control:
BRnzp, **JSR** (and JSRR), JMP, **RET**, **TRAP**
(Also RTI for interrupts)

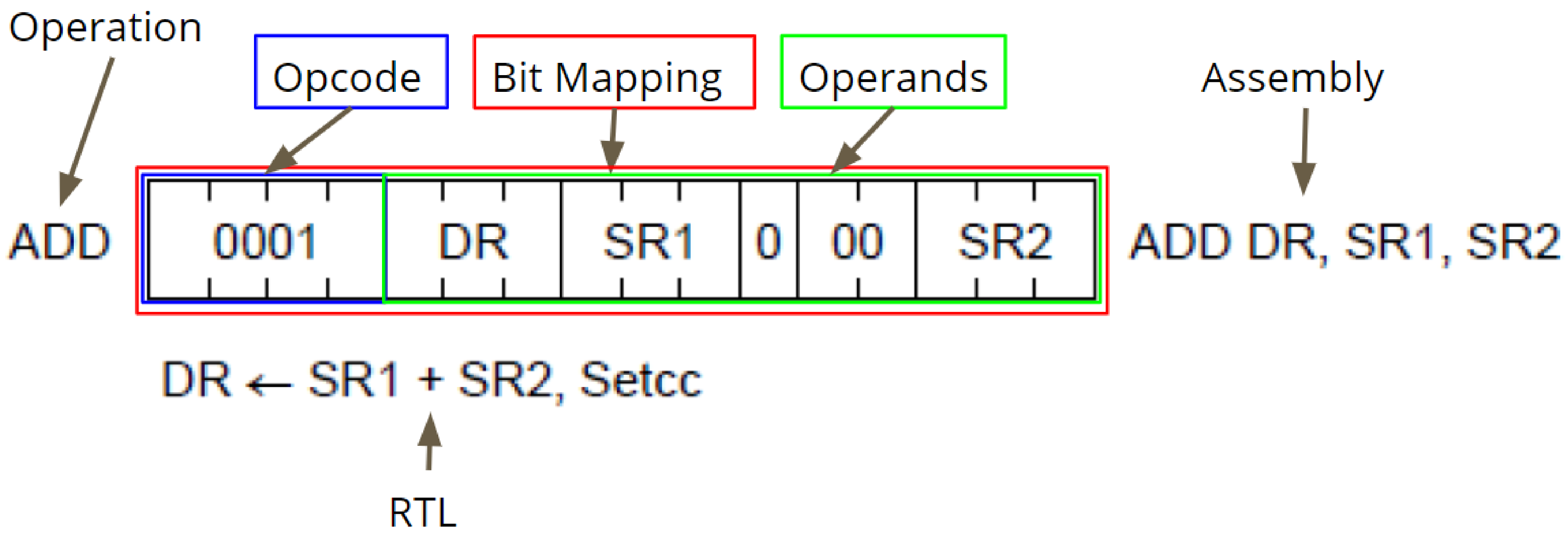
Memory Interface:
LD (LDR, LDI), **ST** (STR, STI), **LEA**

EVERY 16 BIT INSTRUCTION CORRESPONDS TO ONE OF THESE!

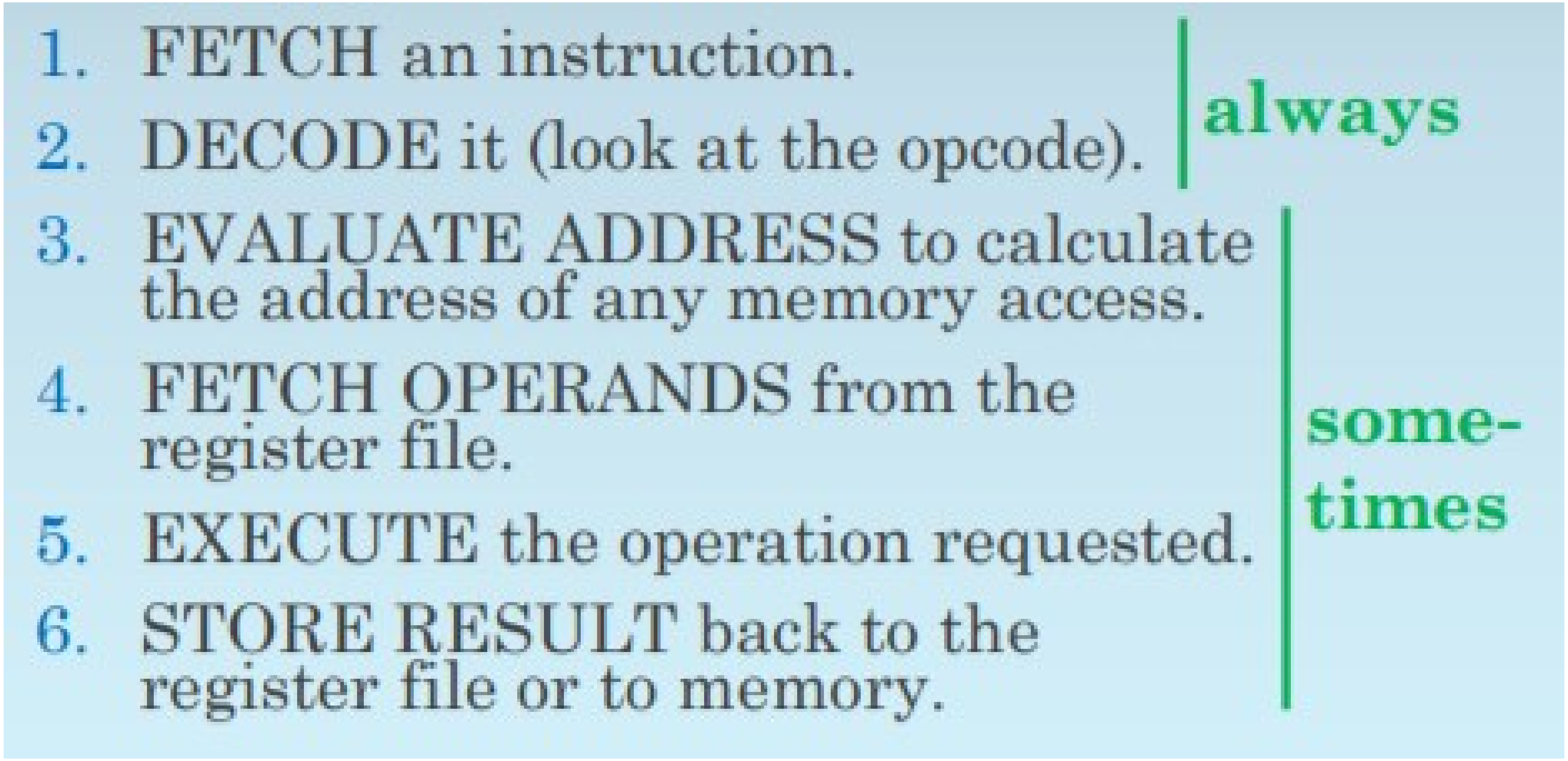
CHEAT SHEET IS YOUR SAVIOR HERE!

Side: Don't worry about JSR, JSRR, TRAP, or RTI yet, just know they exist

How to read your Cheat Sheet (for ISA)



Instruction Cycle!



1. FETCH an instruction.
2. DECODE it (look at the opcode).

always

3. EVALUATE ADDRESS to calculate the address of any memory access.
4. FETCH OPERANDS from the register file.
5. EXECUTE the operation requested.
6. STORE RESULT back to the register file or to memory.

some-times

Details

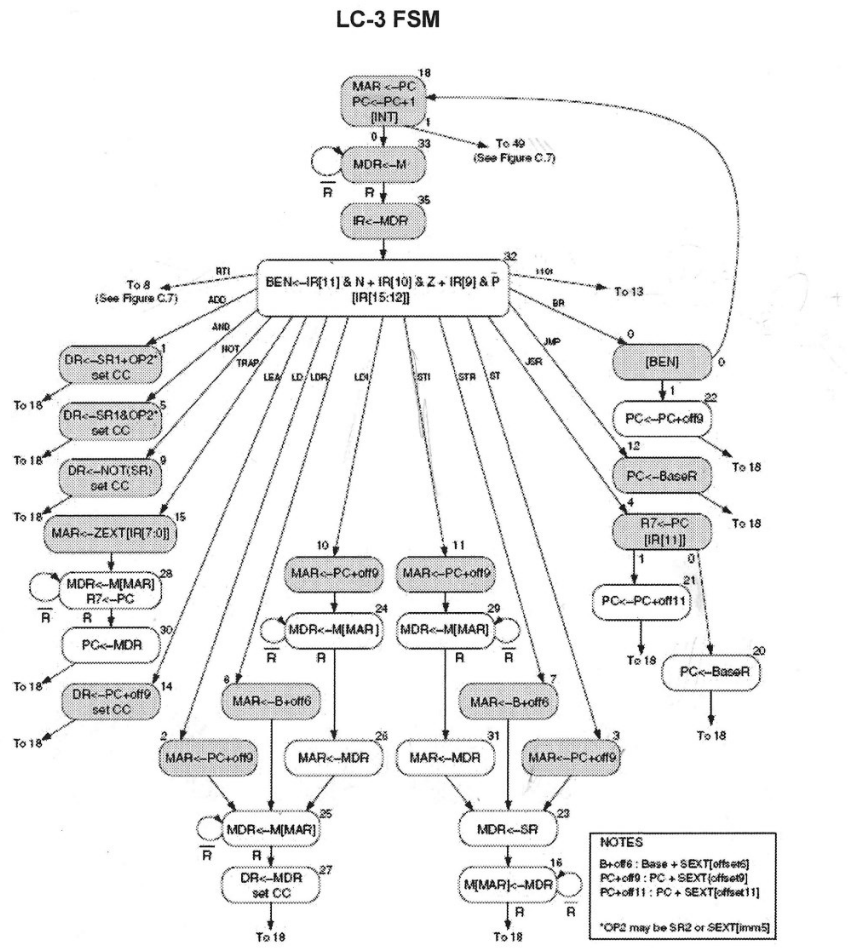
FETCH

- **MAR ← PC**
- **PC ← PC+1**
- **MDR ← M[MAR]**
- **IR ← MDR**
- Used to obtain the next instruction
- IMPORTANT! PC is incremented to next spot BEFORE current instruction put in IR

DECODE

- Opcode (IR[15:12]) examined (first 4 bits)

Learn to read the LC-3 FSM


LC-3 FSM
18
MAR <–PC
PC<–PC+1
[INT]
1
To 49
(See Figure C.7)
0
33
MDR<–M
$\overline{R}$
R
35
IR<–MDR
32
BEN<–IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
(See Figure C.7)
1101
To 13
ADD
AND
NOT
TRAP
LEA
LD
LDR
LDI
STI
STR
ST
JSR
JMP
BR
0
[BEN]
0
1
22
PC<–PC+off9
To 18
12
PC<–BaseR
To 18
4
R7<–PC
[IR[11]]
To 18
1
0
21
PC<–PC+off11
To 18
20
PC<–BaseR
To 18
1
DR<–SR1+OP2*
set CC
To 18
5
DR<–SR1&OP2*
set CC
To 18
9
DR<–NOT(SR)
set CC
To 18
15
MAR<–ZEXT[IR[7:0]]
28
MDR<–M[MAR]
R7<–PC
$\overline{R}$
R
30
PC<–MDR
To 18
14
DR<–PC+off9
set CC
To 18
10
MAR<–PC+off9
24
MDR<–M[MAR]
$\overline{R}$
R
11
MAR<–PC+off9
29
MDR<–M[MAR]
R
$\overline{R}$
6
MAR<–B+off6
7
MAR<–B+off6
2
MAR<–PC+off9
26
MAR<–MDR
31
MAR<–MDR
3
MAR<–PC+off9
25
MDR<–M[MAR]
$\overline{R}$
R
27
DR<–MDR
set CC
To 18
23
MDR<–SR
16
M[MAR]<–MDR
R
$\overline{R}$
To 18
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
PC+off11 : PC + SEXT[offset11]
*OP2 may be SR2 or SEXT[imm5]