BOOLEAN IDENTITIES

To be referenced by number in EGR/CS332, 333, and 433

"OR" Operation

Two ways to designate "AND" Operation

(1) $x + 0 = x$
(2) $x \cdot 0 = 0$
(3) $x + 1 = 1$
(4) $x \cdot 1 = x$
(5) $x + x = x$
(6) $x \cdot x = x$
(7) $x + x' = 1$
(8) $x \cdot x' = 0$
(9) $x + y = y + x$
(10) $xy = yx$
(11) $x + (y + z) = (x + y) + z$
(12) $x(yz) = (xy)z$
(13) $x(y + z) = xy + xz$
(14) $x + yz = (x + y)(x + z)$
(15) $(x + y)' = x'y'$ **DeMorgan's Law**
(16) $(xy)' = x' + y'$ **DeMorgan's Law**
(17) $(x')' = x$

"XOR" (i.e., Exclusive Or) Operation

(18) $X \oplus Y = \overline{X}Y + X$

X' is the inverse of **X** and will be referenced as $\overline{\mathbf{X}}$ in EGR/CS332,333, and 433. Please use this notation.

(The X' notation is just easier for typing)

		•	+	⊕
X	Y	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

JT Wunderlich PhD

Two's Complement

By Thomas Finley, April 2000 http://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html#fromtwo

Two's complement is the way computer represents integers.

Conversion to Two's Complement

Suppose we're working with 8 bit quantities and find how -28 is expressed in two's complement.

First write 28 in binary.

```
00011100
```

Then **invert digits**.

```
11100011
```

Then **add 1**.

```
11100100
```

Conversion from Two's Complement (DO SAME THING !)

Use number 0xFFFFFFFF (i.e., in HEX)

In binary:

```
1111 1111 1111 1111 1111 1111 1111 1111
```

It's leftmost bit is 1, which means it represents a number that is negative.
A leading 1 means number is negative, a leading 0 means positive.

Invert digits:

```
0000 0000 0000 0000 0000 0000 0000 0000
```

Then **add 1**.

```
0000 0000 0000 0000 0000 0000 0000 0001
```

So 0xFFFFFFFF in 2's compliment is -1

Arithmetic with Two's Complement

One of the nice properties of two's complement is that addition and subtraction is made very simple. With a system like two's complement, **the circuitry for addition and subtraction can be unified**, whereas otherwise they would have to be treated as separate operations.

Example 1: Add 69 and 12. If we use decimal, the sum is 81. But let's use binary.

```
  0000 0000 0000 0000 0000 0000 0100 0101 (69)
+ 0000 0000 0000 0000 0000 0000 0000 1100 (12)
-------------------------------------------
  0000 0000 0000 0000 0000 0000 0101 0001 (81)
```

Example 2: Subtract 12 from 69. Now, **69 - 12 = 69 + (-12)**.

To get the negative of 12 we take its binary representation, invert, and add one.

```
0000 0000 0000 0000 0000 0000 0000 1100
```

Invert the digits.

```
1111 1111 1111 1111 1111 1111 1111 0011
```

And add one.

```
1111 1111 1111 1111 1111 1111 1111 0100
```

The last is the binary representation for -12. As before, we'll add the two numbers together.

```
  0000 0000 0000 0000 0000 0000 0100 0101 (69)
+ 1111 1111 1111 1111 1111 1111 1111 0100 (-12)
-------------------------------------------
  0000 0000 0000 0000 0000 0000 0011 1001 (57)
```

We result in 57, which is 69-12. *Note that the leftmost sum will have a carry out which is ignored here.*

Example 3: Lastly, subtract 69 from 12.

12 - 69 = 12 + (- 69).

The two's complement representation of 69 is the following

```
1111 1111 1111 1111 1111 1111 1011 1011
```

So we add this to 12.

```
  0000 0000 0000 0000 0000 0000 0000 1100 (12)
+ 1111 1111 1111 1111 1111 1111 1011 1011 (-69)
-------------------------------------------
  1111 1111 1111 1111 1111 1111 1100 0111 (-57)
```

This results in 12 - 69 = -57, which is correct.

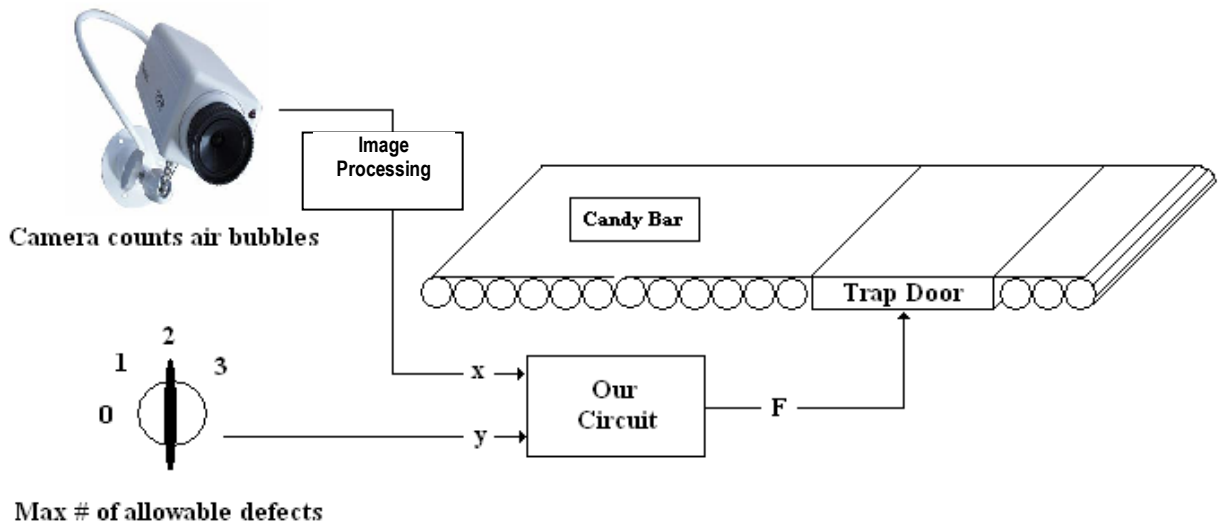
CS/EGR 332

COMBINATIONAL DIGITAL CIRCUIT DESIGN EXAMPLE

JT Wunderlich PhD

The system below is designed using J Wunderlich's **eight steps for digital combinational logic circuit design** to detect the number of bubbles in a chocolate bar, and open a trap door on a conveyer belt when the number of bubbles EXCEEDS the number on a selector switch.

1. Define Problem



- F = 1 IF x > y were x and y can be 0, 1, 2, or 3.
- F = 1 if "FAILS TEST" → Trap door opens
- **Assume** x can't be > 3, and address this assumption later

2. Encode Input and Output Variables into Binary from Analog values

Our problem has 4 analog values per variable (i.e., 0.1.2.3), so:

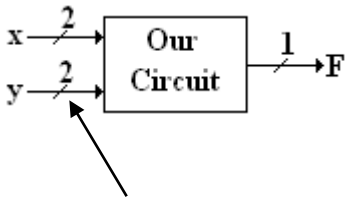
$2^n = 4$

n = 2 Binary Bits needed per Analog variable

x	x1	x0
0	0	0
1	0	1
2	1	0
3	1	1

y	y1	y0
0	0	0
1	0	1
2	1	0
3	1	1

Analog value of F is 1, so no need to encode it into Binary:



This notation means two wires

3. Create Truth-Table

$(2^2) \times (2^2) = (2^{2+2}) = 2^4 = 16$ Cases, so table needs 16 rows

For x For y 4 Input variables

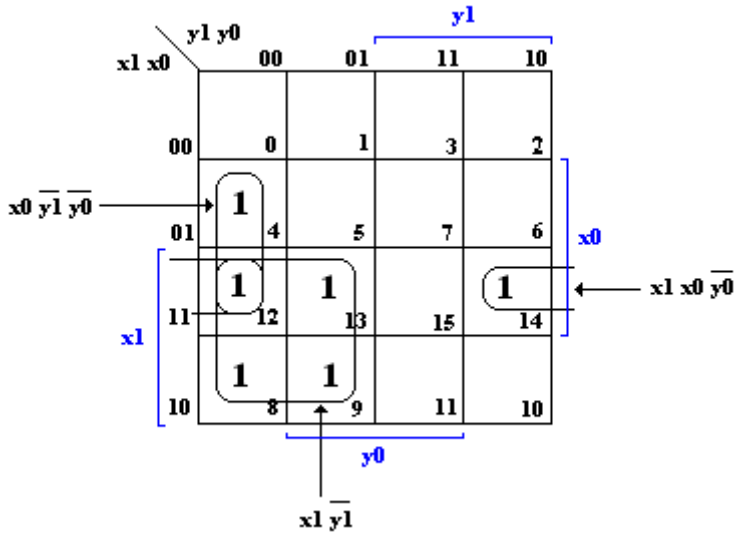
Important to start with the number 0 to represent the case 0000 when all binary input bits are zero

	X		Y		
case # m	x1	x0	y1	y0	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Output F = 1 if input x > input y

4. Find a Simplified Boolean Function

- Using **Simplification Maps** to find **Minimal Sum of Products (SOP)**
  - **i.e., an OR is a logical sum, and an AND is a logical product**
- We will spend a couple weeks learning how to derive and use all 1,2,3,4, and 5 variable versions of these maps



Minimal SOP = F = $x0\overline{y1}\overline{y0} + x1\overline{y1} + x1x0\overline{y0}$

Proof of Simplification Map Using Boolean Algebra
(THIS IS NOT A REQUIRED STEP OF THE DESIGN PROCESS)

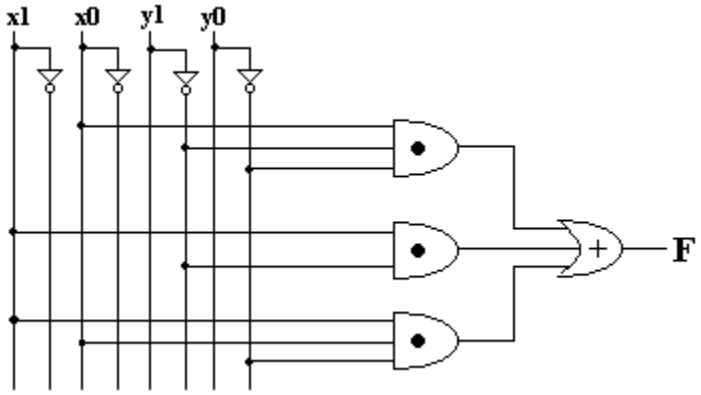
$$F = m4 + m8 + m9 + (m12 + m12 + m12) + m13 + m14$$

$$(m4 + m12)$$
$$(\overline{x1}x0\overline{y1}\overline{y0} + x1x0\overline{y1}\overline{y0})$$
$$(x1 + \overline{x1})(x0\overline{y1}\overline{y0})$$
$$(1)(x0\overline{y1}\overline{y0})$$
$$\underline{(x0\overline{y1}\overline{y0})}$$

$$(m12 + m14)$$
$$(x1x0\overline{y1}\overline{y0} + x1x0y1\overline{y0})$$
$$(y1 + \overline{y1})(x1x0\overline{y0})$$
$$(1)(x1x0\overline{y0})$$
$$\underline{(x1x0\overline{y0})}$$

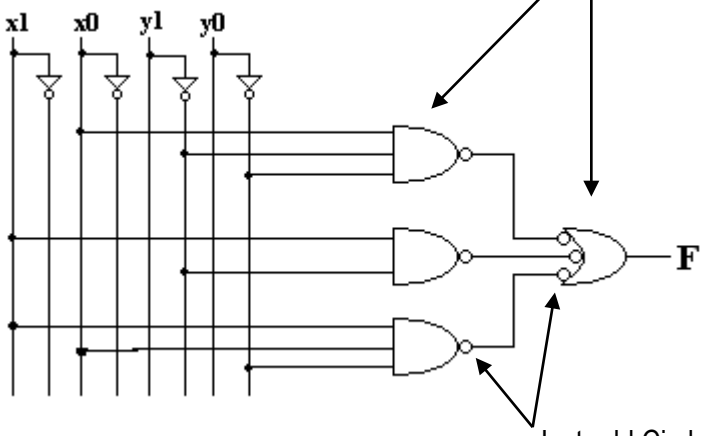
$$m8 + m9 + m12 + m13$$
$$x1\overline{x0}\,\overline{y1}\overline{y0} + x1\overline{x0}y1\overline{y0} + x1x0\overline{y1}\overline{y0} + x1x0\overline{y1}y0$$
$$(y0 + \overline{y0})(x1\overline{x0}y1) + (y0 + \overline{y0})(x1x0\overline{y1})$$
$$(1)(x1\overline{x0}y1) + (1)(x1x0\overline{y1})$$
$$(x1\overline{x0}y1) + (x1x0\overline{y1})$$
$$(x0 + \overline{x0})(x1\overline{y1})$$
$$(1)(x1\overline{y1})$$
$$\underline{(x1\overline{y1})}$$

5. <u>Logic Circuit Diagram</u> (using "Rail Logic")



These vertical lines are called "Rails"
and are an industry standard

6. <u>Possibly Convert to all NAND gates</u> (ONLY IF ASKED FOR)
(This is common for VLSI implementations because NAND gates require less area on a silicon chip)

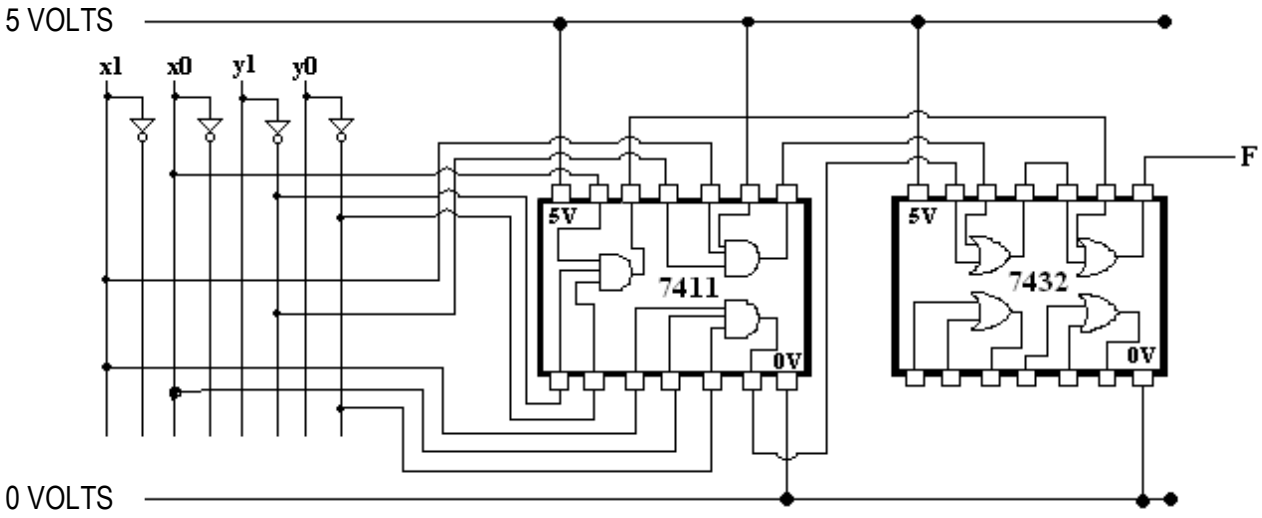


Two different ways to represent a NAND gate

Just add Circles to represent "NOT's and make sure you add one to both ends of each wire so they cancel each other, and therefore don't change the functioning of the circuit

NOTE: If one input to the OR gate does not connect to an AND gate (i.e., is coming directly from a Rail), insert an INVERTOR gate made from a two-input NAND gate with both inputs tied together

7. <u>Chip Circuit Diagram</u> *(not using NAND's)* **for implementing our Logic Circuit**



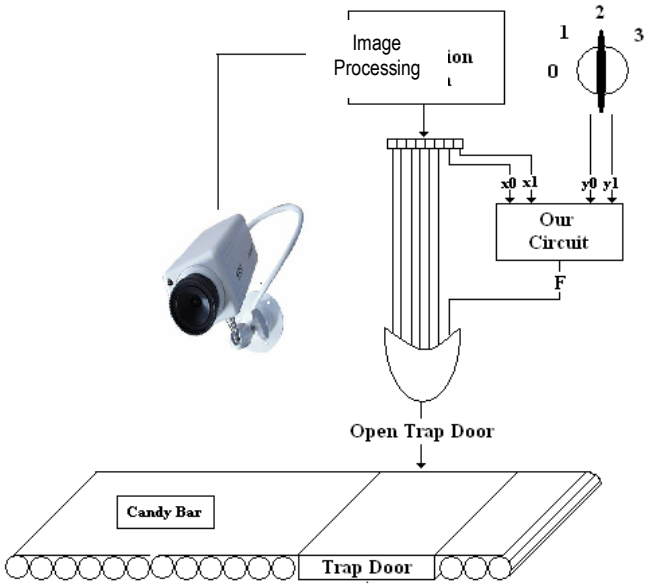
8. **Check Assumptions made in first step**

Examine our assumption that x can't be greater than 3 bubbles.
If we let x be as big as 255, x would need 8 bits to encode it into binary, and our truth table would look like this:

X7	x6	X5	x4	x3	x2	x1	x0	y1	y0	F

2^{10} = 1024 Rows !!!

HOWEVER, an AD HOC SOLUTION to this problem is:



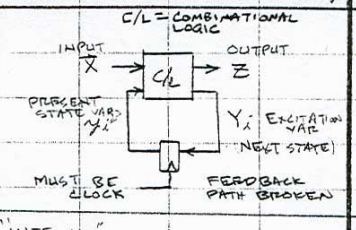
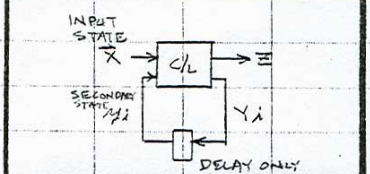
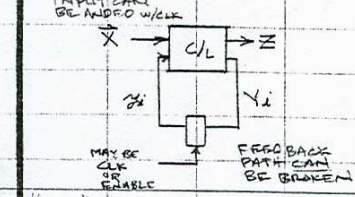
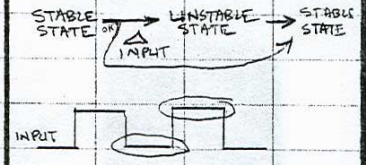
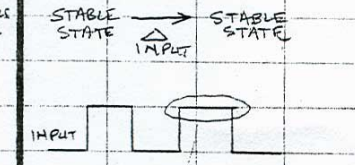
i.e., If any higher ordered bit to the left of the two bits used by our circuit is a 1, the OR gate will produce a 1,

And this will occur if the image processing system detects more than 3 (i.e., Binary 11) bubbles

SEQUENTIAL CIRCUITS

Dr. J WUNDERLICH

CS/ENG 332

	SYNCHRONOUS (CLOCKED)	ASYNCHRONOUS (LEVEL)	PULSE (CLOCK-MODE)
MODEL			
NEXT STATE ($Y_i \ldots Y_j$)	"INTERNAL" STATE $= \mathcal{F}(y_i \ldots y_j)$ (NS) CLOCKED IN	"TOTAL" STATE $= \mathcal{F}\left[(y_i \ldots y_j), \vec{X}\right]$  	"TOTAL" STATE $= \mathcal{F}\left[(y_i \ldots y_j), \vec{X}\right]$ 
	MAY Δ STATE UNDER SAME INPUTS (IE: NEXT CLOCK) $y_i = \mathcal{F}(Y_i, \text{TYPE OF FF USED})$	MUST CONSIDER UP AND DOWN SIDE OF INPUT; 2 LEVEL CHANGES WHICH ARE 2 SEPERATE EVENTS THAT MAY CAUSE TWO STATE CHANGES $(\Delta \text{STABLE STATE}) \text{ IFF } (\Delta \text{INPUT})$ STABLE STATE IFF $y_i = Y_i$	ONLY CONSIDER UP SIDE OF INPUT EACH PULSE CONSIDERED 1 EVENT AND CAN ONLY TRIGGER A SINGLE CHANGE OF STATE $(\Delta \text{STABLE STATE}) \text{ IFF } (\Delta \text{INPUT})$ $y_i = Y_i$ BUT CIRCUIT STILL STABLE IF INPUTS NOT PRESENT
RESTRICTIONS	① NO CHANGING INPUT DURING CLOCK PULSE (OR TRANSITION (EDGE))	① CAN CHANGE INPUT ANY TIME, BUT FOR FUNDAMENTAL MODE, NO INPUT Δ UNTIL IN STABLE STATE ② ONLY ONE INPUT Δ AT A TIME ③ $t_{\text{BETWEEN INPUT } \Delta\text{'s}} > t_{\text{DURATION OF INTERNAL CHANGES}}$	① INPUT PULSE MUST BE LONG ENOUGH TO Δ STATE ② " " MUST BE SHORT ENOUGH TO CAUSE ONLY ONE STATE Δ ③ ONLY ONE INPUT Δ AT A TIME
RACE IN C/L	NO, BECAUSE EVERYTHING CLOCKED	IFF MULTIPLE INPUT Δ BUT, THIS IS NOT ALLOWED	IFF MULTIPLE INPUT Δ BUT, THIS IS NOT ALLOWED
CRITICAL RACE IN FEEDBACK	NO, BECAUSE CLOCKED	YES, IF 2 OR MORE y_i's REQUIRED TO Δ SIMULTANEOUSLY	NO, BECAUSE CIRCUIT IS STABLE WHEN INPUT PULSES NOT PRESENT
STATE ASSIGNMENT	TO MINIMIZE COMPLEXITY AND STRUCTURE OF CIRCUIT EXAMPLE: USING CLOSED PARTITIONS TO REDUCE DEPENDENCY BETWEEN STATE VARIABLES	TO ELIMINATE CRITICAL RACE BY INTRODUCING CYCLES THROUGH UNSTABLE STATES	SAME AS FOR SYNCHRONOUS
DESIGN	① DERIVE STATE TABLE OR DIAGRAM ② MINIMIZE: IF COMPLETELY SPECIFIED: USE PARTITIONING IF INCOMPLETELY SPECIFIED: (a) MERGER GRAPH (b) COMPATIBILITY GRAPH ③ STATE ASSIGNMENT ④ PICK MEMORY ELEMENTS (FF's) ⑤ TRANSITION/OUTPUT TABLE ⑥ EXCITATION TABLE ⑦ EXCITATION(Y) AND OUTPUT(Z) FUNCTIONS ⑧ CIRCUIT	① PRIMITIVE FLOW TABLE FROM TRACE (SPEC STABLE STATE OUTPUTS) ② MINIMIZE (a) MERGER GRAPH (b) COMPATABILITY GRAPH ③ STATE ASSIGNMENT (ACTUALLY SECONDARY VARIABLES y_i) ④ SPEC OUTPUTS FOR UNSTABLE STATES: ⑤ EXCITATION/OUTPUT TABLE ⑥ " " FUNCTIONS ⑦ CIRCUIT	SAME AS SYNCHRONOUS EXCEPT ONLY CONSIDER WHEN INPUTS ARE ACTIVE

FOR:	STABLE STATE OUTPUT ERROR:	TO	Y
FLICKER-FREE	0	0	0
	1	1	1
FAST OUTPUT RESPONSE	0	1	1
	1	0	0
SLOW OUTPUT RESPONSE	0	1	0
	1	0	1

REFERENCES:

1. KOHAVI, Z.V. 1978, SWITCHING AND FINITE AUTOMATA THEORY, McGRAW-HILL PP(228-277, 283-285, 352-368, 385-423)
2. HILL, FREDERICK J. 1981 2ED. ED. SWITCHING THEORY AND LOGIC DESIGN, JOHN WILEY & SONS PP(266-285, 338-356)
3. LANGDON, GLEN G. 1982 COMPUTER DESIGN COMPUTECH PRESS, SAN JOSE, CA PP(504-542)

→ SAME → SAME

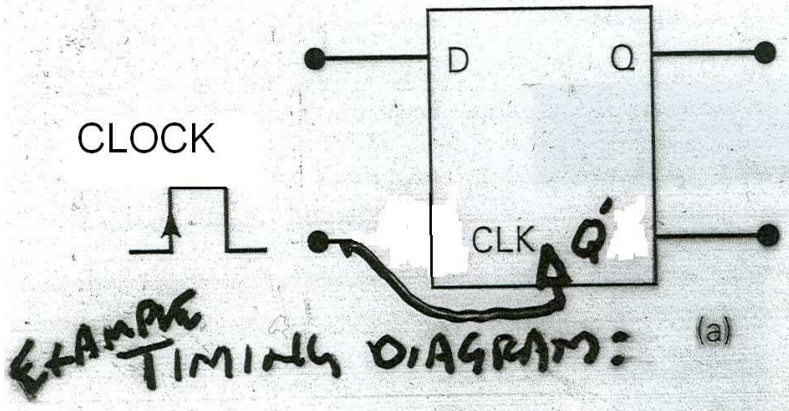
ANALYZE

D Flip-Flop
Memory Element

"CHARACTERISTIC TABLE"

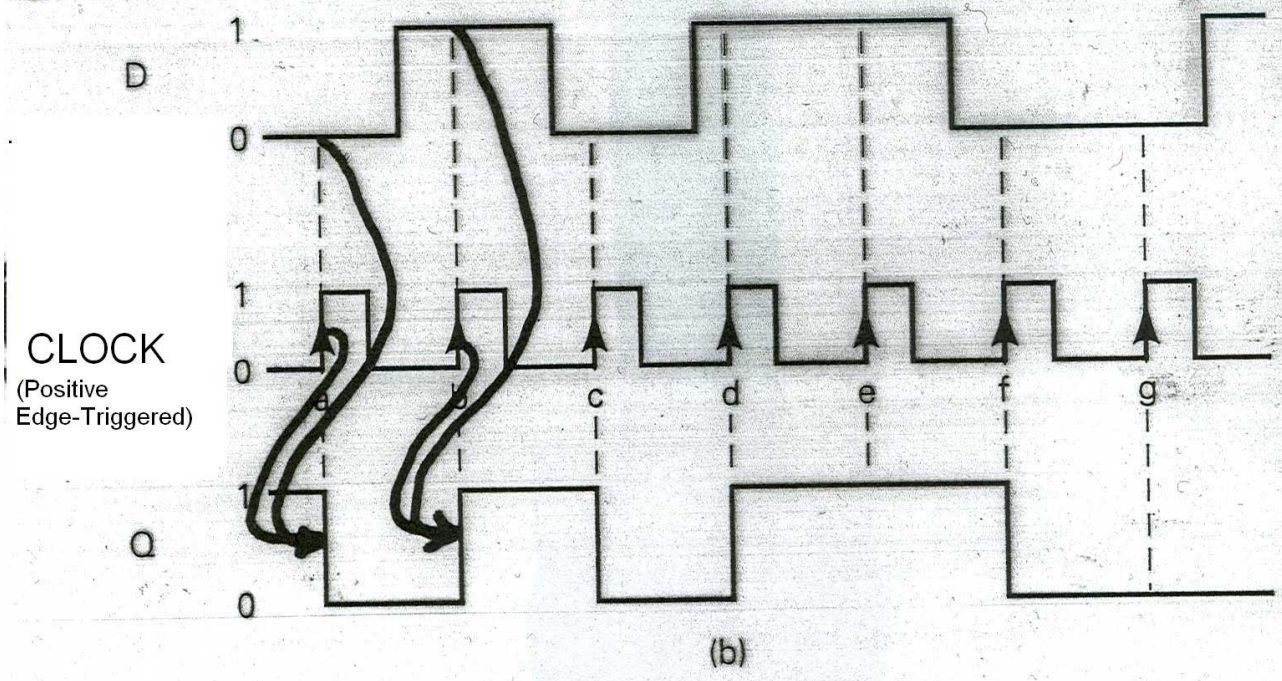
D	CLK	Q(t+1)	
0	↑	0	RESET
1	↑	1	SET

POSITIVE EDGE-TRIGGERED



(a)

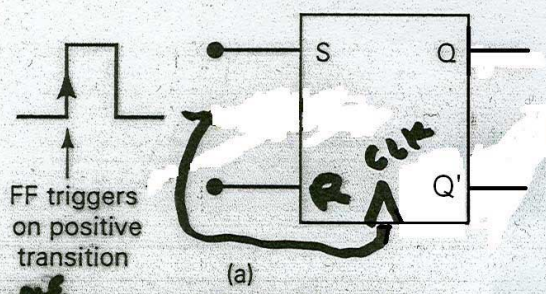
EXAMPLE TIMING DIAGRAM:



(b)

ANALYZE

RS Flip-Flop memory element



(a)

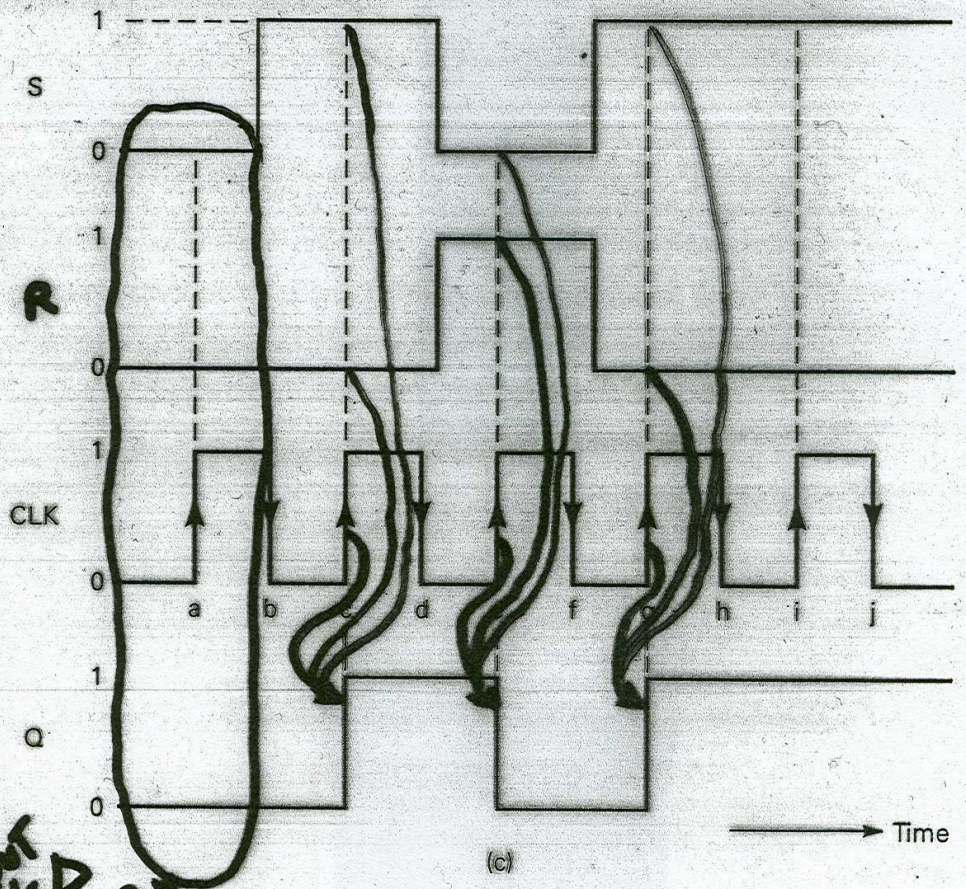
"CHARACTERISTIC" TABLE

Inputs			Output
S	R	CLK	Q(t+1)
0	0	↑	Q(t) no change
1	0	↑	1 SET
0	1	↑	0 RESET
1	1	↑	UNPREDICTABLE

Q(t) is output level prior to ↑ of CLK.
↓ of CLK produces no change in Q.

(b)

EXAMPLE TIMING DIAGRAM:



(c)

OUTPUT CAN'T CHANGE HERE →

For Sequential Synchronous
Digital Circuit Analysis

Flip-Flop Characteristic Tables

JK Flip-Flop

J	*K*	*Q*(*t* + 1)	
0	0	*Q*(*t*)	No change
0	1	0	Reset
1	0	1	Set
1	1	*Q*′(*t*)	Complement

RS Flip-Flop

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

D Flip-Flop

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

T Flip-Flop

T	*Q*(*t* + 1)	
0	*Q*(*t*)	No change
1	*Q*′(*t*)	Complement

NOTE:
CLock Edge is Implied

For Sequential Synchronous
Digital Circuit DESIGN

Flip-Flop Excitation Tables

Q(*t*)	*Q*(*t* + 1)	*S*	*R*
0	0	0	*X*
0	1	1	0
1	0	0	1
1	1	*X*	0

(a) *RS*

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

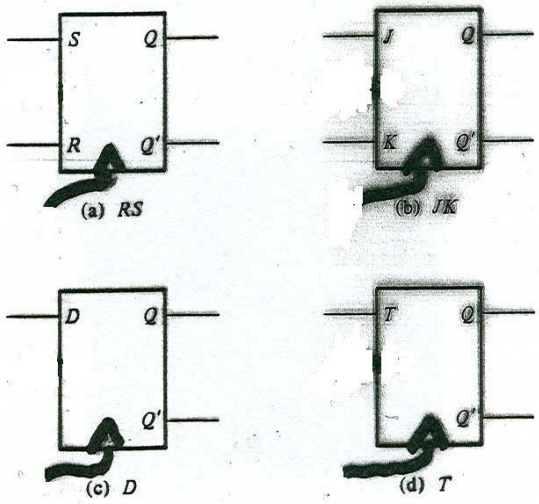
(b) *JK*

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

(c) *D*

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

(b) *T*



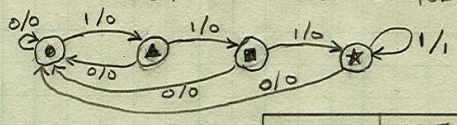
Graphic symbols for flip-flops

J. WUNDERLICH, Ph.D. | SEQUENTIAL CIRCUIT DESIGN | REVISED: 2/19/01

EXAMPLE) DESIGN A CIRCUIT TO DETECT AN INPUT SEQUENCE OF FOUR 1'S

I. STATE DIAGRAM:

★ ASSUME INPUT BIT STREAM (X) IS SYNCHRONIZED WITH CLOCK AND USE ONLY NEGATIVE EDGE-TRIGGERED FLIP-FLOPS



II. STATE TABLE REDUCTION (IF ANY):

SKIP THIS IN CS/ENGR 332

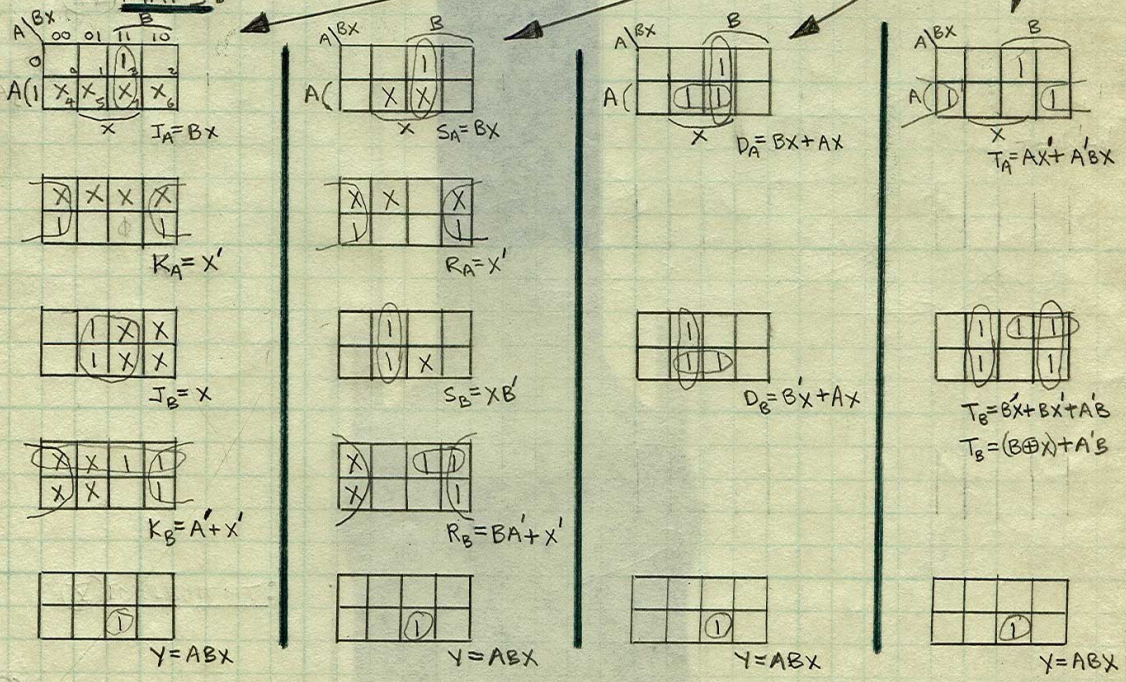
ALL ROWS OF TABLE UNIQUE, SO NO REDUCTION

PRESENT STATE	NEXT STATE X=0	NEXT STATE X=1	OUTPUT X=0	OUTPUT X=1
●	●	▲	0	0
▲	●	■	0	0
■	●	★	0	0
★	●	★	0	1

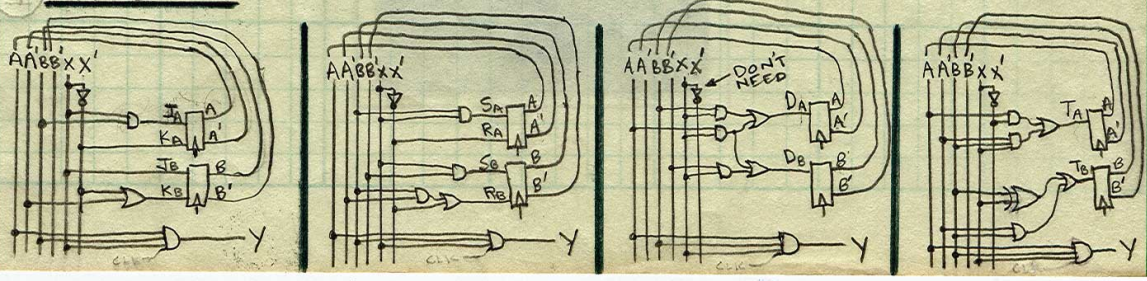
III. STATE ASSIGNMENT & STATE TABLE:

PRESENT STATE A	PRESENT STATE B	INPUT X	NEXT STATE A	NEXT STATE B	OUTPUT Y	J_A	K_A	J_B	K_B	S_A	R_A	S_B	R_B	D_A	D_B	T_A	T_B
0	0	0	0	0	0	0	X	0	X	0	X	0	X	0	0	0	0
0	0	1	0	1	0	0	X	1	X	0	X	1	0	0	1	0	1
0	1	0	0	0	0	0	X	X	1	0	X	0	1	0	0	0	1
0	1	1	1	0	0	1	X	X	1	1	0	0	1	1	0	1	1
1	0	0	0	0	0	X	1	0	X	0	1	0	X	0	1	1	0
1	0	1	1	1	0	X	0	1	X	X	0	1	0	1	1	0	1
1	1	0	0	0	0	X	1	X	1	0	1	0	1	0	0	1	1
1	1	1	1	1	1	X	0	X	0	X	0	X	0	1	1	0	0

IV. MAPS:



V. IMPLEMENT:



22-141 50 SHEETS
22-142 100 SHEETS
22-144 200 SHEETS
AMPAD

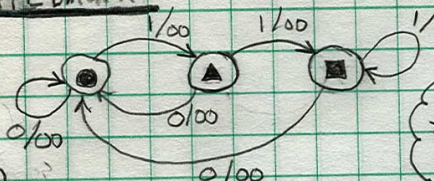
DESIGN A CIRCUIT TO DETECT AN INPUT SEQUENCE OF 3 ONES. ALSO, DETECT SELF-CORRECTION.

(ASSUME EACH UNUSED STATE TRANSITIONS DIRECTLY TO A USED STATE. THE VERIFIER THIS)

FOR CG/EGR 333 ☆ ASSUME INPUT BITSTREAM(X) IS SYNCHRONIZED WITH CLOCK AND USE ONLY NEGATIVE EDGE-TRIGGERED FLIP-FLOPS

X, CLOCK, Y, Z=1 WHEN SELF-CORRECTION OCCURS

(2) STATE DIAGRAM:



(3) (4) (5) STATE TABLE:

Q(t) AB	X	Q(t+1) AB	Y	Z
00	0	00	0	0
00	1	01	0	0
01	0	00	0	0
01	1	10	0	0
10	0	00	0	0
10	1	01	0	0
11	0	0	0	1
11	1	0	0	1

UNUSED STATE

DETECT SELF-CORRECTION

LEAVE BLANK

DON'T WANT OUTPUT=1 BY ACCIDENT!

ASSUMING SELF-CORRECTION OCCURS FOR X=0 AND X=1

(6) FLIP-FLOP INPUTS:

$J_A K_A$	$J_B K_B$	$S_A R_A$	$S_B R_B$	D_A	D_B	T_A	T_B
0 X	0 X	0 X	0 X	0	0	0	0
0 X	1 X	0 X	1 0	0	1	0	1
0 X	X 1	0 X	0 1	0	0	0	1
1 X	X 1	1 0	0 1	1	0	1	1
X 1	0 X	0 1	0 X	1	0	1	0
X 0	0 X	X 0	0 X	0	0	1	0
X X	X X	X X	X X	X	X	X	X
X X	X X	X X	X X	X	X	X	X

EXCITATION TABLES (FOR DESIGN)

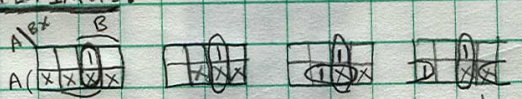
Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

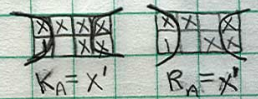
Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

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

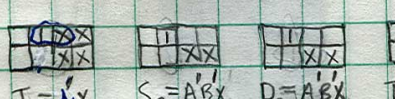
(7) SIMPLIFY OUTPUTS AND FLIP-FLOP INPUTS:



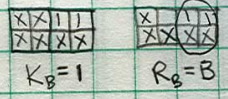
$J_A = BX$ $S_A = BX$ $D_A = BX + AX$ $T_A = AX' + BX$



$K_A = X'$ $R_A = X'$

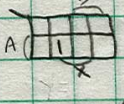


$J_B = A'X$ $S_B = A'B'X$ $D_B = A'B'X$ $T_B = B + A'X$



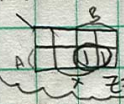
$K_B = 1$ $R_B = B$

Y OUTPUT IS SAME REGARDLESS OF FLIP-FLOP TYPE USED:



$Y = AB'X$

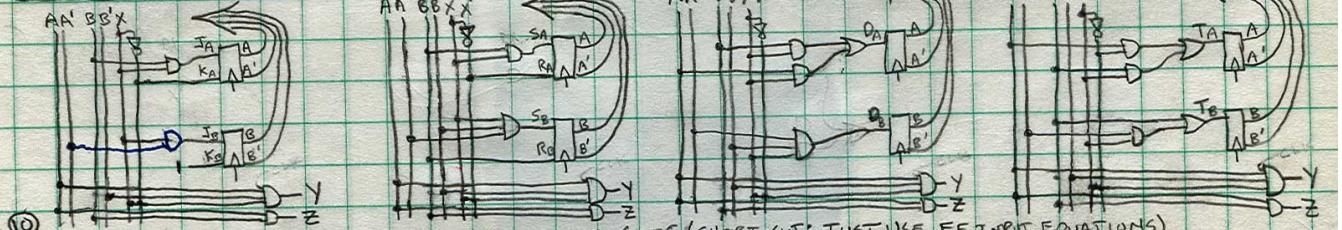
Z OUTPUT IS ALSO SAME REGARDLESS OF FF USED:



$Z = AB$

(9) SKIP

(8) (3) LOGIC CIRCUIT IMPLIMENTATION:



(10) ANALYZE CIRCUIT TO SEE WHERE UNUSED STATE GOES (SHORT-CUT: JUST USE FF INPUT EQUATIONS)

UNUSED STATE Q(t) AB	X	FLIP-FLOP INPUT AT TIME t $J_A K_A$	$J_B K_B$	WHERE IT GOES Q(t+1) AB
11	0	0 1	0 1	00
11	1	1 0	0 1	10

USING JK CHARAC. TABLE:

JK	Q(t+1)
00	Q(t) NO Δ
01	0 RESET
10	1 SET
11	Q'(t) COMPLEMENT

Q(t) AB	X	$S_A R_A$	$S_B R_B$	Q(t+1) AB
11	0	0 0	0 1	00
11	1	1 0	0 1	10

USING SR CHARAC. TABLE:

SR	Q(t+1)
00	Q(t)
01	0
10	1
11	UNPREDICTABLE!

Q(t) AB	X	D_A	D_B	Q(t+1) AB
11	0	0	0	00
11	1	1	0	10

USING D CHARAC. TABLE:

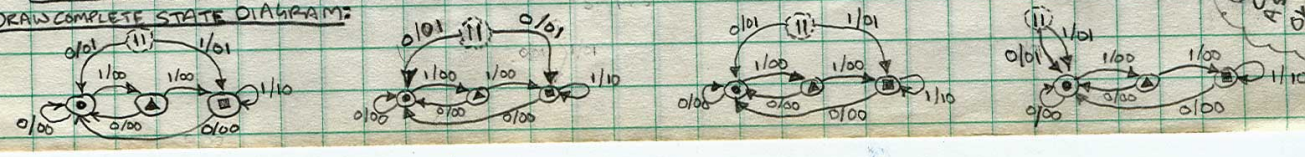
D	Q(t+1)
0	0
1	1

Q(t) AB	X	T_A	T_B	Q(t+1) AB
11	0	1	1	00
11	1	1	1	00

USING T CHARAC. TABLE:

T	Q(t+1)
0	Q(t)
1	Q'(t)

(11) DRAW COMPLETE STATE DIAGRAM:

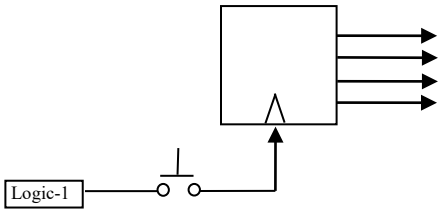


EXAMPLE #1

Design a BCD (Binary Coded Decimal) counter using T flip-flops, and the simplest circuitry possible *(i.e., this means don't have an input, and connect a push-button switch to the Flip-Flops)*. **Also, this counter resets to zero after nine. Don't force the unused states to go anywhere. And as usual, don't try to "minimize the machine," convert to NAND's, or create a Chip Circuit Diagram** *(i.e., only do these things when specifically asked for).*

STEP #1 Define The Problem (with a Block Diagram)

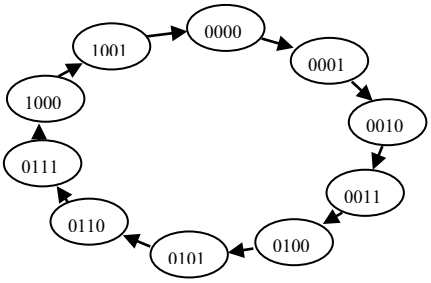
Let's not have an input to trigger each count, but instead let's act as the clock by substituting a push-button switch tied to the positive terminal of our power supply (which is equivalent to a Logic-1)



STEP #2 Draw State Diagram.

No need to label transition-arrows since there are no inputs (Thanks to our acting as the clock)

Also, we can use state variables for outputs



STEP #3 Encode Variables → Already Binary. And we need four State Variables here (A,B,C,D) for the four bits used for BCD

STEP #4 Minimize Machine → (not done in EGR/CS332)

STEP #5 Make State Table

There are no output columns since we are using the state variables for outputs

STEP #6 Append flip-flop inputs using excitation tables

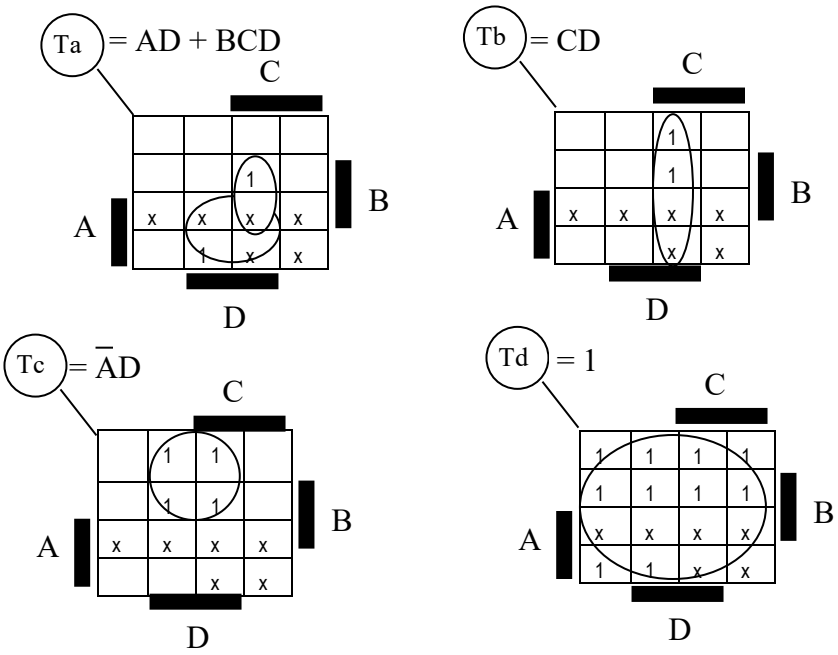
(using **EXCITATION** TABLE for T Flip-Flop for this problem):

Present State Q(t)	Next State Q(t+1)	T Flip-Flop Input to achieve Q(t+1)	
0	**0**	**0**	**No Change**
0	**1**	**1**	**Toggle**
1	**0**	**1**	**Toggle**
1	**1**	**0**	No Change

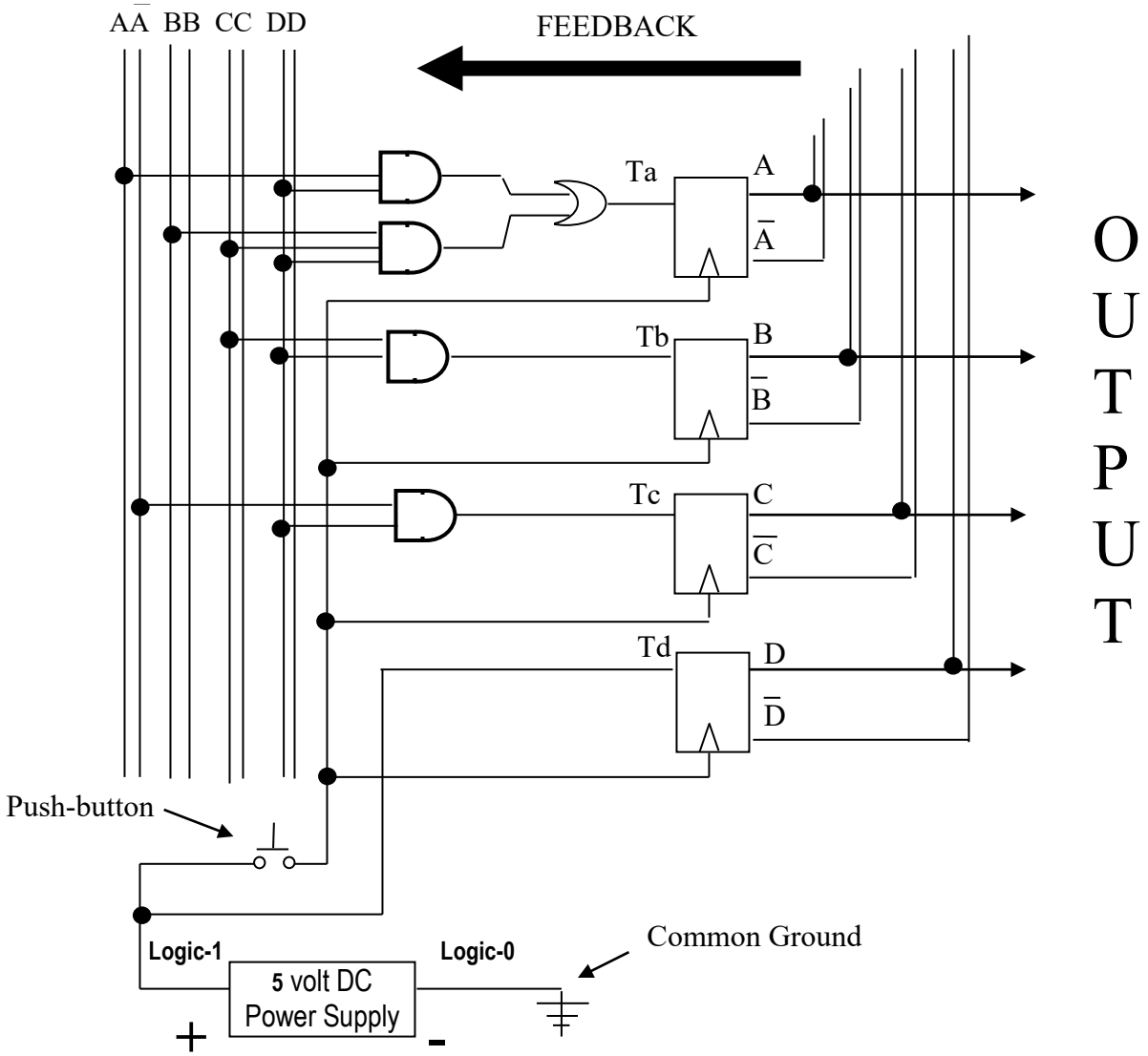
STATE TABLE

PRESENT STATE Q(t)				NEXT STATE Q(t+1)				FLIP-FLOP INPUTS				
A	B	C	D	A	B	C	D	Ta	Tb	Tc	Td	m
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	1	0	0	0	1	1	1
0	0	1	0	0	0	1	1	0	0	0	1	2
0	0	1	1	0	1	0	0	0	1	1	1	3
0	1	0	0	0	1	0	1	0	0	0	1	4
0	1	0	1	0	1	1	0	0	0	1	1	5
0	1	1	0	0	1	1	1	0	0	0	1	6
0	1	1	1	1	0	0	0	1	1	1	1	7
1	0	0	0	1	0	0	1	0	0	0	1	8
1	0	0	1	0	0	0	0	1	0	0	1	9
1	0	1	0	?	?	?	?	X	X	X	X	10
1	0	1	1	?	?	?	?	X	X	X	X	11
1	1	0	0	?	?	?	?	X	X	X	X	12
1	1	0	1	?	?	?	?	X	X	X	X	13
1	1	1	0	?	?	?	?	X	X	X	X	14
1	1	1	1	?	?	?	?	X	X	X	X	15

STEP #7 Simplify flip-flop inputs using maps



STEP #8 Draw Logic circuit



STEP #9 Convert to NANDS → Not asked for

STEP #10 Analyze unused states using flip-flop characteristic table and remembering your Flip Flo input functions:

$Ta = AD + BCD$

$Tb = CD$

$Tc = A\overline{D}$

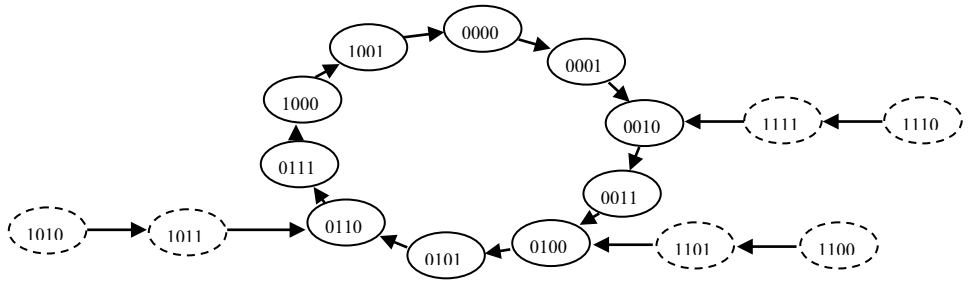
$Td = 1$

After you fill in the part of the table evaluating all Flip-Flop Inputs, use the T Flip-Flop **CHARACTERISTIC** TABLE to look at each State Variable:

Flip-Flop Input T	What happens to State Variable at next clock edge Q(t+1)	
0	Q(t)	No Change
1	$\overline{Q(t)}$	Toggle

PRESENT STATE Q(t)				CALCULATE FLIP-FLOP INPUTS				NEXT STATE Q(t+1)			
A	B	C	D	Ta	Tb	Tc	Td	A	B	C	D
1	0	1	0	0	0	0	1	1	0	1	1
1	0	1	1	1	1	0	1	0	1	1	0
1	1	0	0	0	0	0	1	1	1	0	1
1	1	0	1	1	0	0	1	0	1	0	0
1	1	1	0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	0	1	0	0	1	0

STEP #11 Re-draw state diagram to show unused states

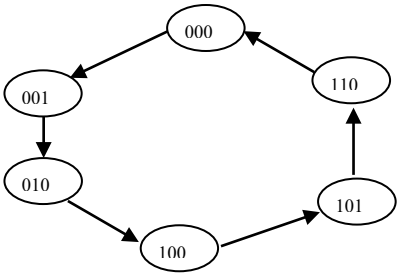


STEP #12 Chip Circuit Diagram (only when asked for in EGR/CS332)

STEP #13 Review assumptions → None made (other than we don't want to force the unused state anywhere for the given problem – however, in reality you probably want to force your machine to reset if an unused state occurs – i.e., force it back to the initial state because the unused state is most likely an error)

EXAMPLE #2

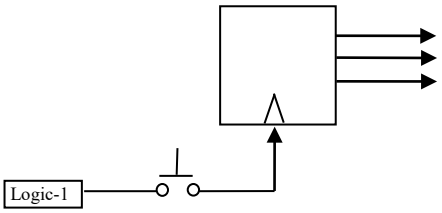
Given the following State Diagram, state in words what it is doing, then draw the block diagram, then continue all the design steps. Use JK flip-flops, and use the simplest circuitry possible *(i.e., this means don't have an input, and connect a push-button switch to the Flip-Flops)*. **And as usual, don't try to "minimize the machine," convert to NAND's, or create a Chip Circuit Diagram** ***(i.e., only do these things when specifically asked for).***



In words, what this is doing: This is a counter that counts from 000 to 001 to 010 to 100 to 101 to 110, then resets to 000

STEP #1 Define The Problem (with a Block Diagram)

Let's not have an input to trigger each count, but instead let's act as the clock by substituting
a push-button switch tied to the positive terminal of our power supply (which is equivalent to a Logic-1)



STEP #2 Draw State Diagram. GIVEN

STEP #3 Encode Variables → already Binary. And we need three State Variables (A,B,C) for the three bits used in the given count

STEP #4 Minimize Machine → (not done in this course)

STEP #5 Make State Table

There are no output columns since we are using the state variables for outputs

STEP #6 Append flip-flop inputs using excitation tables
(using **EXCITATION** TABLE for JK Flip-Flop for this problem):

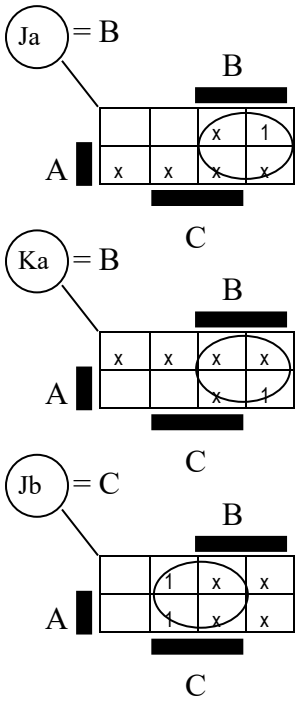
Present State Q(t)	Next State Q(t+1)	JK Flip-Flop Inputs to achieve Q(t+1)	
0	0	0 X	No Change (00) or Reset (01)
0	1	1 X	Set (10) or Toggle (11)
1	0	X 1	Reset (01) or Toggle (11)
1	1	X 0	No Change or Set

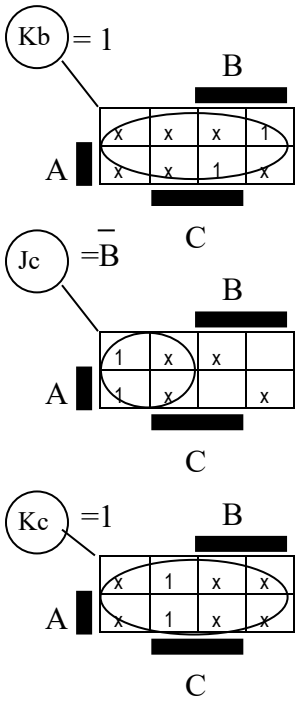
STATE TABLE

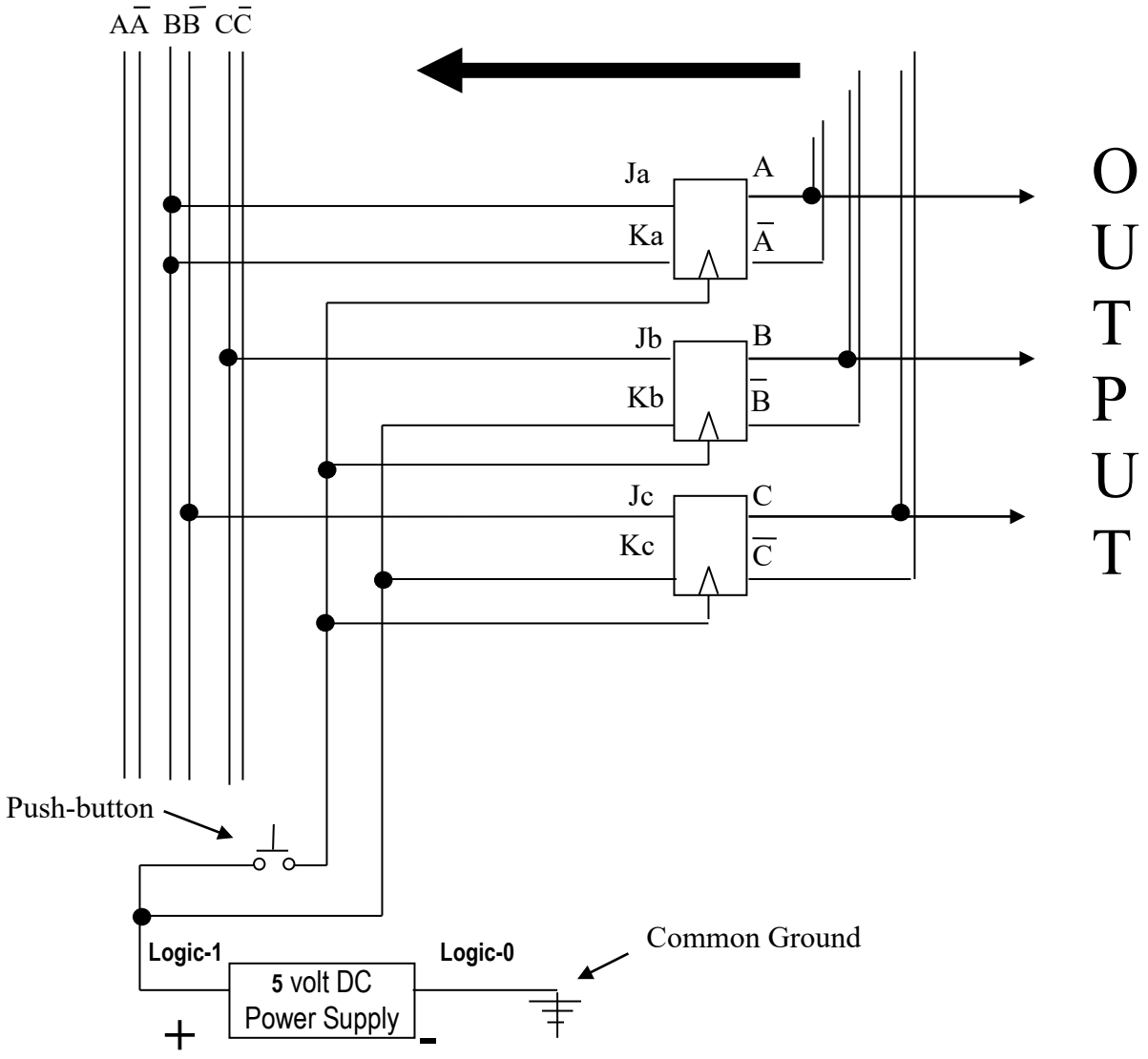
PRESENT STATE Q(t)			NEXT STATE Q(t+1)			FLIP-FLOP INPUTS						
A	B	C	A	B	C	Ja	Ka	Jb	Kb	Jc	Kc	*m*
0	0	0	0	0	1	0	X	0	X	1	X	0
0	0	1	0	1	0	0	X	1	X	X	1	1
0	1	0	1	0	0	1	X	X	1	0	X	2
0	1	1	?	?	?	X	X	X	X	X	X	3
1	0	0	1	0	1	X	0	0	X	1	X	4
1	0	1	1	1	0	X	0	1	X	X	1	5
1	1	0	0	0	0	X	1	X	1	0	X	6
1	1	1	?	?	?	X	X	X	X	X	X	7

Don't move the unused states from the row corresponding to the midterm that represents it; this is to preserve the integrity of the Simplification Maps and the order of all the cells within them

STEP #7 Simplify flip-flop inputs using maps







STEP #9 Convert to NANDS → not asked for

STEP #10 Analyze unused states using flip-flop characteristic table and remembering your Flip Flop input functions:

Ja = B
Ka = B
Jb = C
Kb = 1
Jc = $\overline{B}$
Kc = 1

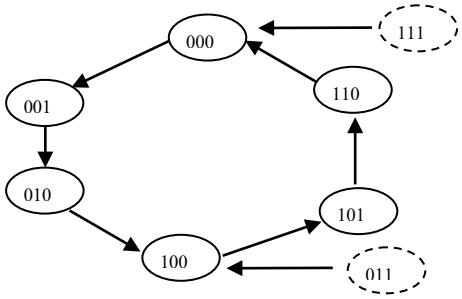
After you fill in the part of the table evaluating all Flip-Flop Inputs use the JK Flip-Flop **CHARACTERISTIC** TABLE to look at each State Variable:

Flip-Flop Inputs — What happens to State Variable at next clock edge

J	K	Q(t+1)	
0	0	Q(t)	No Change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q(t)}$	Toggle

PRESENT STATE Q(t)			CALCULATE FLIP-FLOP INPUTS						NEXT STATE Q(t+1)		
A	B	C	Ja	Ka	Jb	Kb	Jc	Kc	A	B	C
1	1	1	1	1	1	1	0	1	0	0	0
0	1	1	1	1	1	1	0	1	1	0	0

STEP #11 Re-draw state diagram to show unused states



STEP #12 Chip Circuit Diagram (only when asked for in EGR/CS332)

STEP #13 Review assumptions → None made (other than we don't want to force the unused state anywhere for the given problem – however, in reality you probably want to force your machine to reset if an unused state occurs – i.e., force it back to the initial state because the unused state is most likely an error)

Typical Machine-Instruction Cycle

Dr. Joseph Wunderlich

Phase 1: "FETCH"

- CPU puts address of the "next" machine instruction onto the Address bus
- CPU sends "READ" signal to memory (cache's first, then main memory)
- Instruction read into CPU via data bus from first memory where it is found (i.e., L1 cache, or L2 cache, or main memory)
- If it is not located in any of them, this is a "Page Fault" and a new page must be put into main memory from disk storage

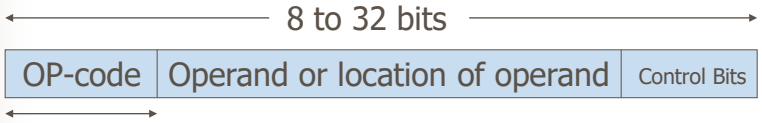
Phase 2: "DECODE"

- CPU decodes instruction put into it's Instruction Register during the "FETCH

- Machine Instructions have two main parts:
  1. **OP-Code:** Identifies which instruction to execute
  2. **Operand:** Data to be used during execution

Phase 2: "DECODE" continued

- Typical Instruction Format:



5 to 11 bits in OP-Code
Therefore 2^5=32 to 2^11=2048 different machine instructions in "**Instruction Set**"

- Some simple Microcontrollers (e.g., PIC's) have only 32 instructions
- Some large-scale machines (e.g., IBM S/390) have close to 2000 instructions

Phase 2: "DECODE" continued

- Location of operand:
  1. "IMMEDIATE:" Data encoded into machine instruction (Fastest to execute)
  2. "MEMORY-REFERENCED:" Data located in memory at a location defined by address encoded into machine instruction (Slowest to execute)
  3. "REGISTER-REFERENCED:" Data is located in an internal CPU register and it's register number is encoded into machine instruction

Phase 3: "EXECUTE"

- If necessary, read operand data from cache's or main memory:
  1. CPU puts address of operand onto address bus
  2. CPU exerts a "READ" signal
  3. Data read into CPU via data bus from first memory where it is found (i.e., L1 cache, or L2 cache, or main memory)
  4. If it is not located in any of them, this is a "Page Fault" and a new page must be put into main memory from disk storage
- Many different types of data manipulations are carried out depending on the type of instruction (e.g., ADD, SUBTACT, MULTIPLY, MOVE, JUMP, etc.)

Phase 4: "WRITE-BACK"

- This phase is only necessary for memory referenced instructions which write results back to memory:
  1. CPU puts address onto address bus of where data is to be written to
  2. CPU puts Data onto data bus
  3. CPU exerts a "WRITE" signal
  4. Data written into memory

Combinational and Sequential Logic Design of a CPU

J. Wunderlich, Ph.D.
Elizabethtown College

Review of Design Steps

Sequential Design

- Step 1: Define problem
- Step 2: Create state diagram
- Step 3: Encode variables !
- Step 4: Minimize machine ?
- Step 5: Create state table
- Step 6: Append flip-flop inputs
- Step 7: Find simplified function(s)
- Step 8: Draw logic circuit
- Step 9: Convert to NAND's ?
- Step 10: Analyze any unused states ?
- Step 11: Revise state diagram ?
- Step 12: Check Assumptions
- Step 13: Chip circuit diagram ?

Combinational Design

- Step 1: Define problem
- Step 2: Encode variables ?
- Step 3: Create truth table
- Step 4: Find simplified function(s)
- Step 5: Draw logic circuit
- Step 6: Convert to NAND's ?
- Step 7: Check assumptions
- Step 8: Chip circuit diagram ?

NOTE: ? = may not be required

CPU Design

Sequential Design

- A Finite State Machine (FSM) is the *heart* of the CONTROL UNIT (a.k.a. "*The Puppet Master*")
- Also REGISTERS made from SRAM or D flip-flops

Combinational Design

- Everything in CPU other than control unit is considered the DATA PATH (a.k.a. "*The Puppet*")

A Dr.W. CPU Design Example

"Write Specifications"

- DESIGN BIT-WIDTH : Front Side Bus (FSB) and internal CPU bus are 16-bit
- DESIGN REGISTER SET : Four 16-bit general-purpose registers (R0,R1,R2,R3)
- DESIGN INSTRUCTION SET :
  - Four register-referenced *(using user-specified registers Ri,Rj,Rk)*
    - » ADD Ri,Rj,Rk [*RTN*: R_i+R_j -->R_k]
    - » SUB Ri,Rj,Rk [*RTN*: R_i-R_j -->R_k]
    - » AND Ri,Rj,Rk [*RTN*: R_i AND R_j -->R_k]
    - » OR Ri,Rj,Rk [*RTN*: R_i OR R_j -->R_k]
  - Two memory-referenced *(using user-specified registers Ri,Rj,Rk)*
    - » ADD@ Ri,Rj,Rk [*RTN*: (R_i)+(R_j) -->(R_k)]
    - » SUB@ Ri,Rj,Rk [*RTN*: (R_i)-(R_j) -->(R_k)]
  - Two mixed register/memory-referenced *(using user-specified reg's Ri,Rj,Rk)*
    - » MOVEIN Ri,Rj [*RTN*: (R_i)--> R_j]
    - » MOVEOUT Ri,Rj [*RTN*: R_i-->(R_j)]

(R) means R contains address of data in memory

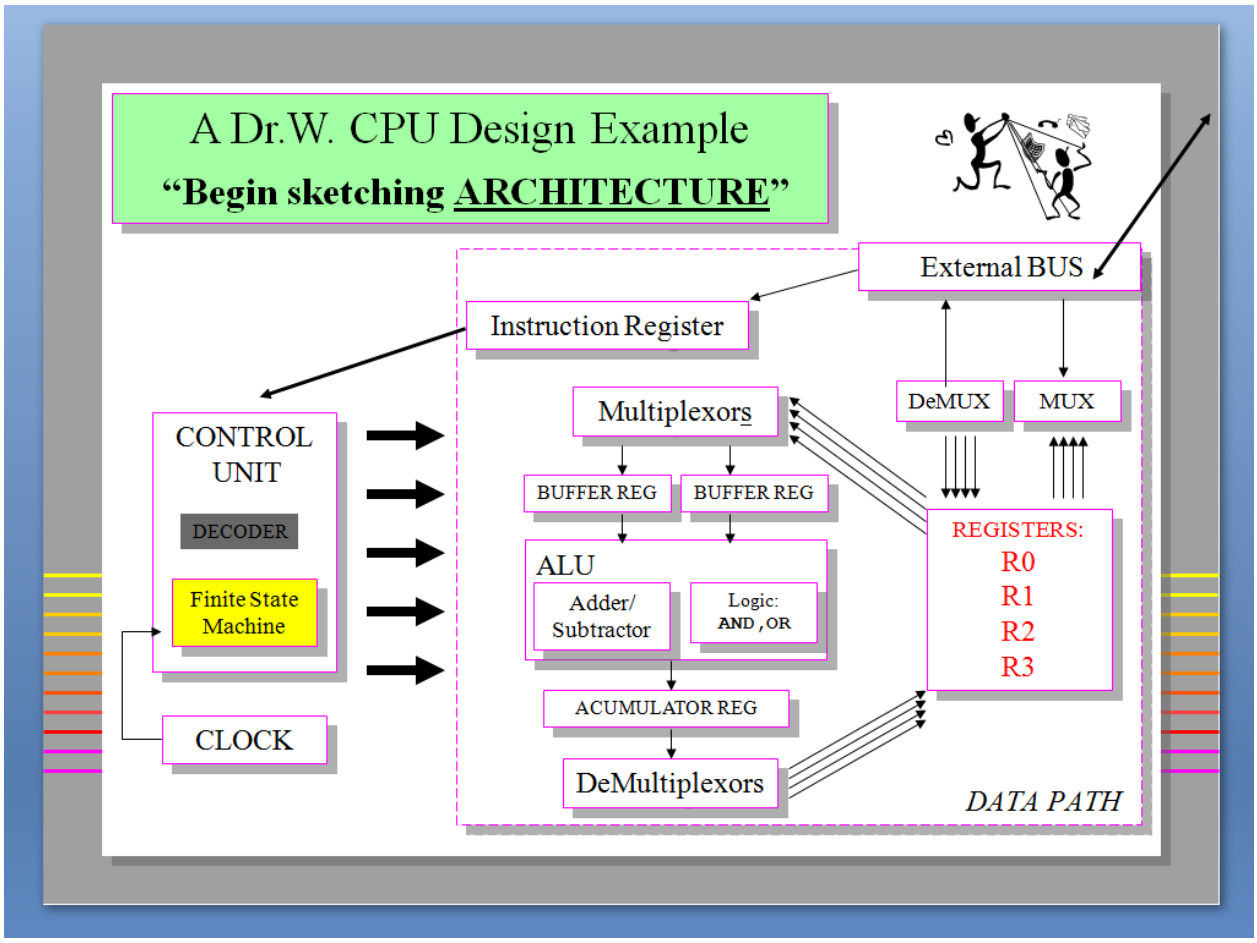
NOTE: *RTN* = Register Transfer Notation

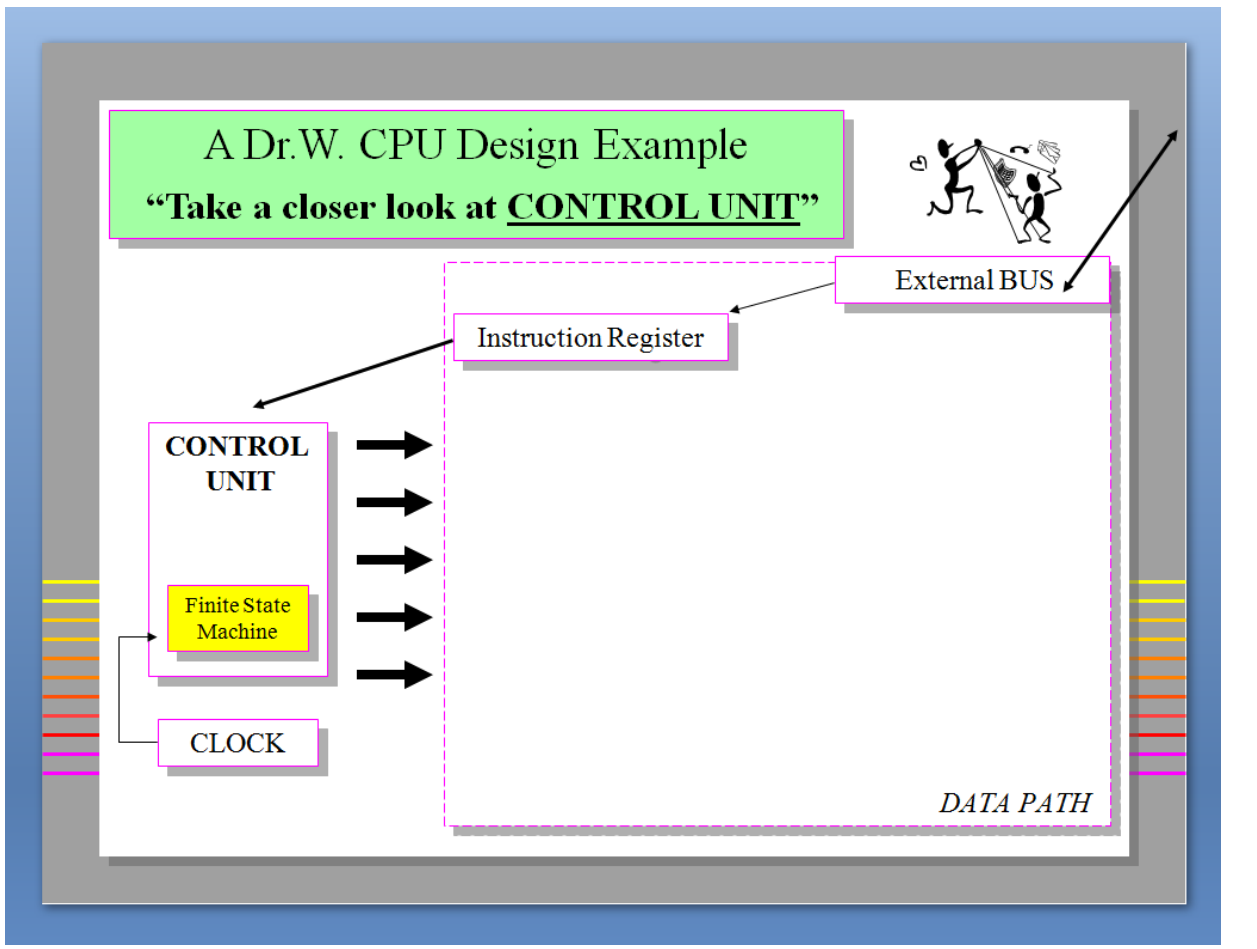
A Dr.W. CPU Design Example

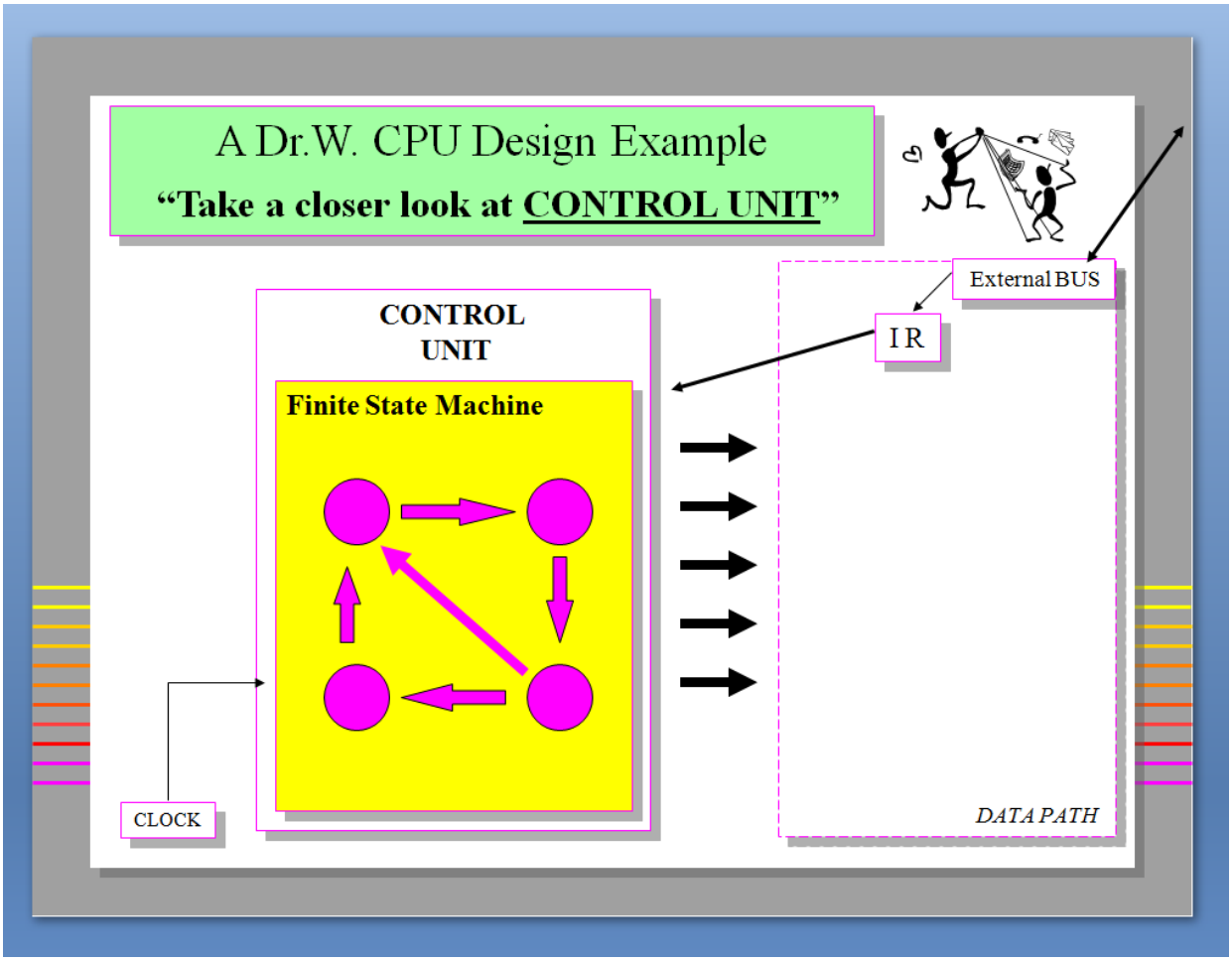
"Design Instruction FORMAT"

- Need $\log_2(8) = 3$ bits for OP-CODE (i.e., since 8 instructions)
- Need three 2-bit fields to specify OPERANDS (i.e., which of 4 reg's to use)
- EXAMPLE (including MACHINE CODE) :
  - Four register-referenced *(using user-specified registers Ri,Rj,Rk)*
    - » ADD R0,R1,R2 [*RTN*: R_0+R_1 --> R_2] [000000110]
    - » SUB R1,R2,R3 [*RTN*: R_1-R_2 --> R_3] [001011011]
    - » AND R0,R1,R2 [*RTN*: R_0 AND R_1 --> R_2] [010000110]
    - » OR R1,R2,R3 [*RTN*: R_1 OR R_2 --> R_3] [011011011]
  - Two memory-referenced *(using user-specified registers Ri,Rj,Rk)*
    - » ADD@ R0,R1,R2 [*RTN*: (R_0)+(R_1) -->(R_2)] [100000110]
    - » SUB@ R1,R2,R3 [*RTN*: (R_1)-(R_2) -->(R_3)] [101011011]
  - Two mixed register/memory-referenced *(using user-specified reg's Ri,Rj,Rk)*
    - » MOVEIN R0,R1 [*RTN*: (R_0)--> R_1] [1100001XX]
    - » MOVEOUT R1,R2 [*RTN*: R_1 -->(R_2)] [1110110XX]

NOTE:
X = "Don't Care"

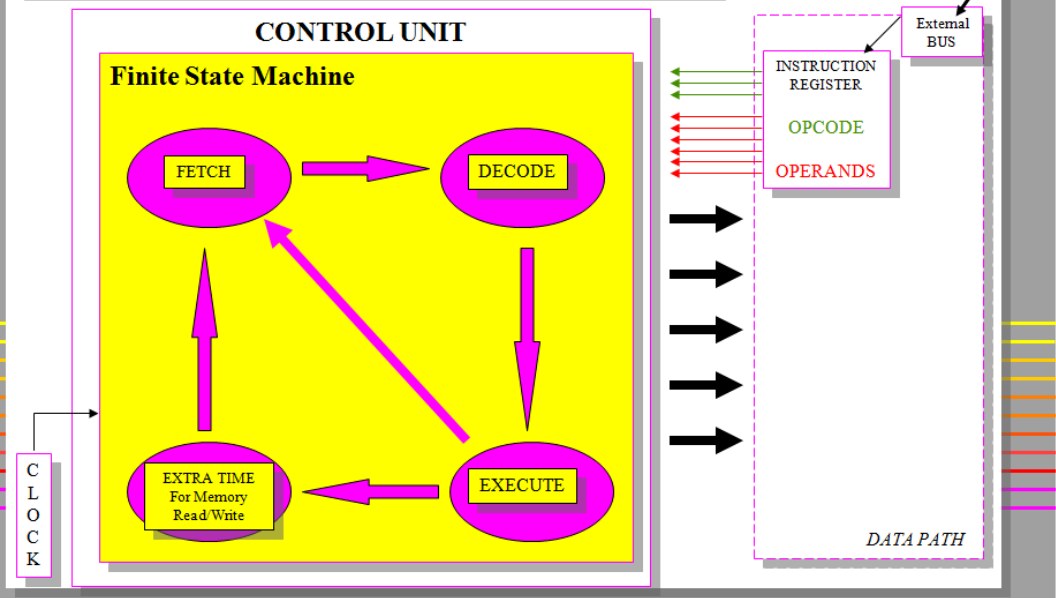

A Dr.W. CPU Design Example
"Begin sketching ARCHITECTURE"
External BUS
Instruction Register
CONTROL UNIT
DECODER
Finite State Machine
CLOCK
Multiplexors
DeMUX
MUX
BUFFER REG
BUFFER REG
ALU
Adder/ Subtractor
Logic: AND , OR
REGISTERS:
R0
R1
R2
R3
ACUMULATOR REG
DeMultiplexors
DATA PATH


A Dr.W. CPU Design Example
"Take a closer look at CONTROL UNIT"
External BUS
Instruction Register
CONTROL UNIT
Finite State Machine
CLOCK
DATA PATH


A Dr.W. CPU Design Example
"Take a closer look at CONTROL UNIT"
External BUS
IR
CONTROL UNIT
Finite State Machine
CLOCK
DATA PATH

A Dr.W. CPU Design Example

"Take a closer look at CONTROL UNIT"



Now we are ready for logic design

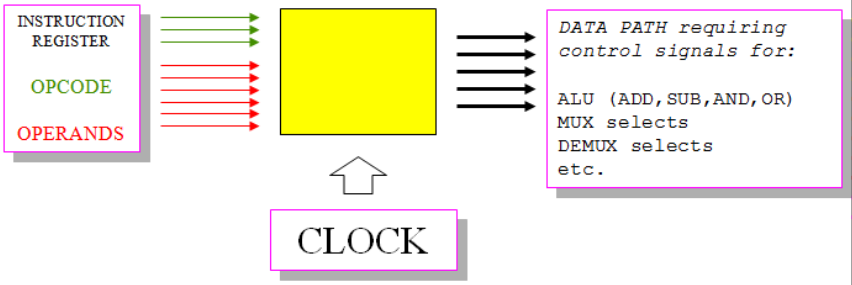
Sequential Design

- Step 1: Define problem
- Step 2: Create state diagram
- Step 3: Encode variables !
- Step 4: Minimize machine ?
- Step 5: Create state table
- Step 6: Append flip-flop inputs
- Step 7: Find simplified function(s)
- Step 8: Draw logic circuit
- Step 9: Convert to NAND's ?
- Step 10: Analyze any unused states ?
- Step 11: Revise state diagram ?
- Step 12: Chip circuit diagram ?
- Step 13: Check Assumptions

NOTE: ? = may not be required

Step 1

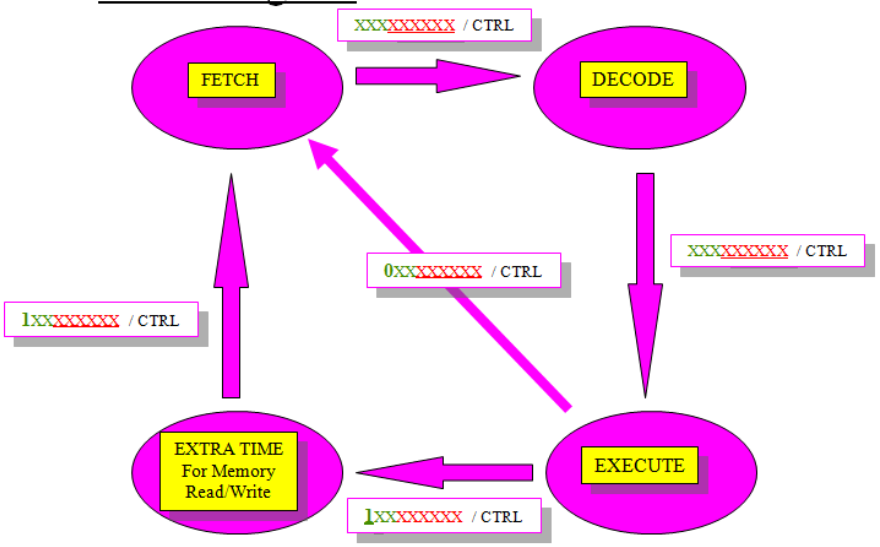
- Define Problem
  - Draw Block Diagrams
  - Make Assumptions
    - (assume synchronous for undergrad courses)



Step 2

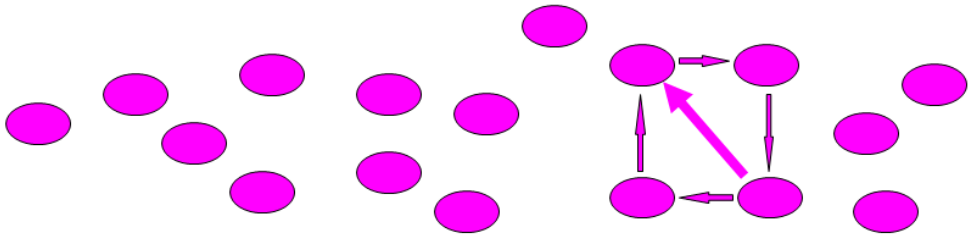
NOTE:
CNTRL = all control signals to *DATA PATH*

- Create State Diagram



Step 2 (cont.)

- Create State Diagram ***(if we did a Full Design)***
  - Much more Complexity required!!
  - Need Memory Read/Write States (with internal FSM's for communication "handshaking" with memory) for Fetching instructions, reading operands, and writing results to memory
  - Since 9 inputs, could need as many as $2^9 = 512$ transitions from each state (actually much less, but still many needed!)



We can however continue some here

NOTE:
CNTRL = all control signals to *DATA PATH*

Assign State Variables

	A	B
FETCH	0	0
DECODE	0	1
EXECUTE	1	0
WRITEBACK	1	1

State Table

Q(t) A	Q(t) B	BIT 1 of OPCODE	Q(t+1) A	Q(t+1) B	OUTPUT
0	0	0	0	1	CTRL
0	0	1	0	1	CTRL
0	1	0	1	0	CTRL
0	1	1	1	0	CTRL
1	0	**0**	**0**	**0**	CTRL
1	0	**1**	**1**	**1**	CTRL
1	1	0	0	0	CTRL
1	1	1	0	0	CTRL

Append Flip-Flop Inputs (ASSUME JK's)

J_A	K_A	J_B	K_B	m
0	x	1	x	0
0	x	1	x	1
1	x	x	1	2
1	x	x	1	3
x	1	0	x	4
x	0	1	x	5
x	1	x	1	6
x	1	x	1	7

using JK Excitation Table:

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

$J_A = B$

A \ B	0	0	1	1
x	x	x	x	

OP1

$K_A = B + OP1'$

A \ B	x	x	x	x
1	0	1	1	

OP1

$J_B = A' + OP1$

A \ B	1	1	x	x
0	1	x	x	

OP1

$K_B = 1$

A \ B	x	x	x	x
x	x	x	x	

OP1

DRAW LOGIC CIRCUIT HERE

Let's write some Assembly Language using Dr. Wunderlich's Instruction Set

RECALL:

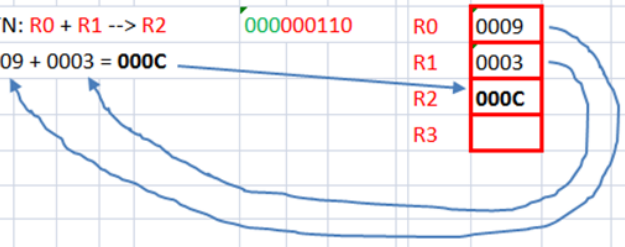
EXAMPLE (including MACHINE CODE) :

- Four register-referenced *(using user-specified registers Ri,Rj,Rk)*
  - » ADD R0,R1,R2 [*RTN*: R_0+R_1 --> R_2] [000000110]
  - » SUB R1,R2,R3 [*RTN*: R_1-R_2 --> R_3] [001011011]
  - » AND R0,R1,R2 [*RTN*: R_0 AND R_1 --> R_2] [010000110]
  - » OR R1,R2,R3 [*RTN*: R_1 OR R_2 --> R_3] [011011011]
- Two memory-referenced *(using user-specified registers Ri,Rj,Rk)*
  - » ADD@ R0,R1,R2 [*RTN*: $(R_0)+(R_1)$ --> (R_2)] [100000110]
  - » SUB@ R1,R2,R3 [*RTN*: $(R_1)-(R_2)$ --> (R_3)] [101011011]
- Two mixed register/memory-referenced *(using user-specified reg's Ri,Rj,Rk)*
  - » MOVEIN R0,R1 [*RTN*: (R_0)--> R_1] [1100001XX]
  - » MOVEOUT R1,R2 [*RTN*: R_1 --> (R_2)] [1110110XX]

NOTE:
X = "Don't Care"

Using Dr. Wunderlich's Instruction Set,
write an Assembly Language program to:
Add ("ADD") the two-byte Data (0009h) from Register 0,
to the two-byte Data (0003h) from Register 1,
and put the result in Register 2

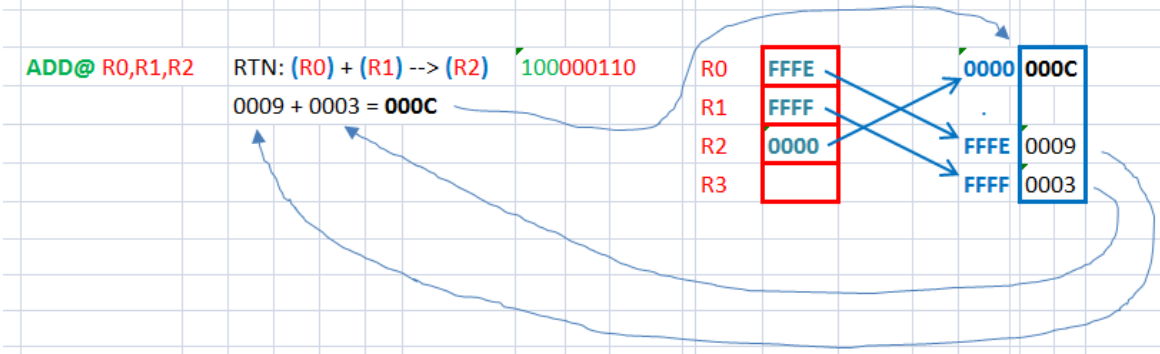
Assembly Code	Comments	Machine Code *(in Binary)*	CPU Register Contents *(in Hex)*	RAM "Memory" Contents *(everything in Hex)*
	INITIAL CONDITIONS		R0 0009 R1 0003 R2 R3	0000 . FFFF
ADD R0,R1,R2	RTN: R0 + R1 --> R2 0009 + 0003 = **000C**	000000110	R0 0009 R1 0003 R2 **000C** R3	0000 . FFFF



Using Dr. Wunderlich's Instruction Set, and assuming two-byte "Words" of Data in RAM Memory,
write an Assembly Language program to:

Add ("ADD@") the two-byte Data (0009h) from the Memory Address FFFEh stored in Register 0,
to the two-byte Data (0003h) from the Memory Address FFFFh stored in Register 1,
and put the result in Memory at the Address 0000h stored in Register 2,

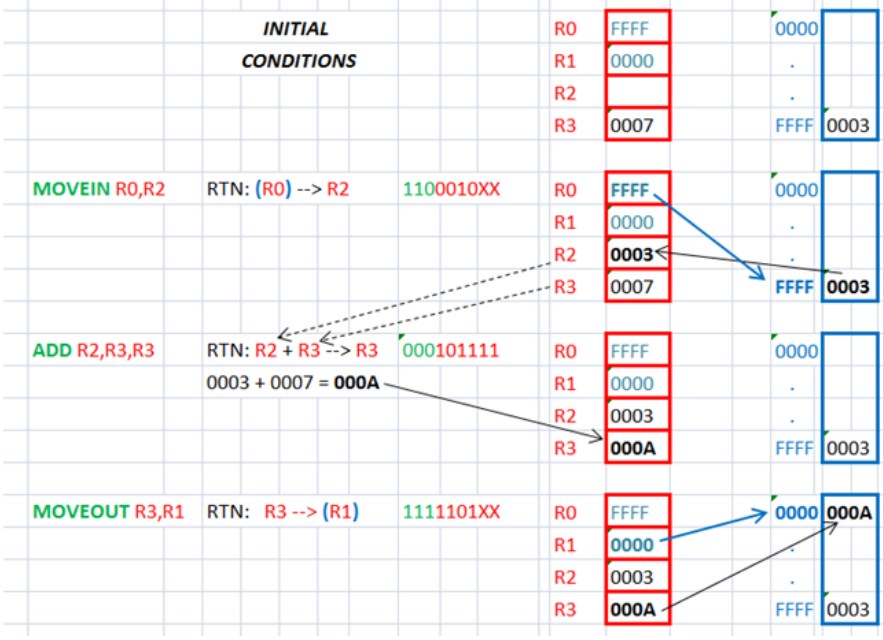
Assembly Code	Comments	Machine Code (in Binary)	CPU Register Contents (in Hex)	RAM "Memory" Contents (everything in Hex)
	INITIAL CONDITIONS		R0 FFFE R1 FFFF R2 0000 R3	0000 . FFFE 0009 FFFF 0003
ADD@ R0,R1,R2	RTN: (R0) + (R1) --> (R2) 0009 + 0003 = **000C**	100000110	R0 **FFFE** R1 **FFFF** R2 **0000** R3	**0000 000C** . **FFFE** 0009 **FFFF** 0003



Using Dr. Wunderlich's Instruction Set, and assuming two-byte "Words" of Data in RAM Memory,
write an Assembly Language program to:

1) Load (i.e., "**MOVEIN**") the two-byte Data (0003h) from the Memory Address FFFFh stored in Register 0, into Register 2
2) **ADD** contents of Register 2 to contents of Register 3 (which is initially set to 007h) and put result in Register 3
3) Store (i.e., "**MOVEOUT**") the contents of Register 3 to the Memory Address 0000h that is in Register 1

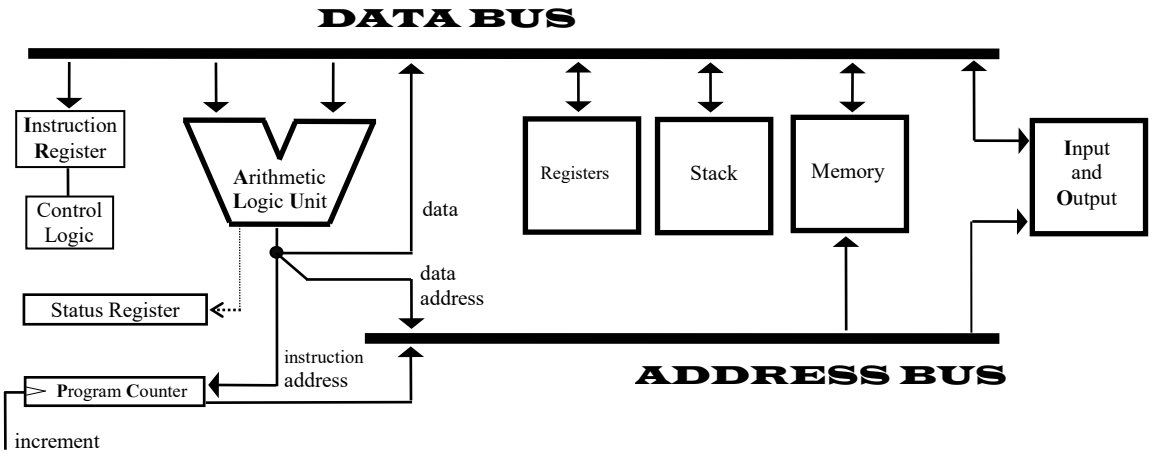
Assembly Code	Comments	Machine Code (in Binary)	CPU Register Contents (in Hex)	RAM "Memory" Contents (Addresses and Data in Hex)
	INITIAL CONDITIONS		R0 FFFF R1 0000 R2 R3 0007	0000 . . FFFF 0003
MOVEIN R0,R2	RTN: (R0) --> R2	1100010XX	R0 **FFFF** R1 0000 R2 **0003** R3 0007	0000 . . **FFFF 0003**
ADD R2,R3,R3	RTN: R2 + R3 --> R3 0003 + 0007 = **000A**	000101111	R0 FFFF R1 0000 R2 0003 R3 **000A**	0000 . . FFFF 0003
MOVEOUT R3,R1	RTN: R3 --> (R1)	1111101XX	R0 FFFF R1 **0000** R2 0003 R3 **000A**	**0000 000A** . . FFFF 0003



"Minimal Computer-Architecture"

by J Wunderlich PhD

Note: This is the one thing that you need to know for both EGR/CS230 and EGR/CS332



Program Counter addresses machine instructions to be fetched from memory

Instruction Register receives fetched machine instruction

Control Logic creates all routing signals after decoding the fetched machine instruction

Arithmetic Logic Unit (ALU) performs arithmetic and logical manipulation of data and addresses

Registers (i.e., general purpose registers) to sometimes store intermediate results of calculations

Status Register holds status flags and condition codes (Parity, Comparison Bit, and sometimes control bits for machine configuring)

"Memory" (i.e. "main memory") stores data and instructions, and sometimes intermediate results of calculations

Stack stores addresses (or processor status) for returning from program-calls (or interrupts)

Input/Output ("I/O") "Channels" often addressed as memory (i.e., memory-mapped I/O)

NOTE 1: What is actually in a CPU vs on a Motherboard varies by Microprocessor and Microcontroller system
NOTE 2: Cache Memories, Bus Controller, and an assortment of other functional parts are not shown on this simplest of diagrams