# Lecture Summary – Module 4

## *Arithmetic and Computer Logic Circuits*

**Learning Outcome:** ***an ability to analyze and design computer logic circuits***

**Learning Objectives:**

4-1. compare and contrast three different signed number notations: sign and magnitude, diminished radix, and radix

4-2. convert a number from one signed notation to another

4-3. describe how to perform sign extension of a number represented using any of the three notation schemes

4-4. perform radix addition and subtraction

4-5. describe the various conditions of interest following an arithmetic operation: overflow, carry/borrow, negative, zero

4-6. describe the operation of a half-adder and write equations for its sum (S) and carry (C) outputs

4-7. describe the operation of a full adder and write equations for its sum (S) and carry (C) outputs

4-8. design a "population counting" or "vote counting" circuit using an array of half-adders and/or full-adders

4-9. design an N-digit radix adder/subtractor circuit with condition codes

4-10. design a (signed or unsigned) magnitude comparator circuit that determines if A=B, A<B, or A>B

4-11. describe the operation of a carry look-ahead (CLA) adder circuit, and compare its performance to that of a ripple adder circuit

4-12. define the CLA propagate (P) and generate (G) functions, and show how they can be realized using a half-adder

4-13. write the equation for the carry out function of an arbitrary CLA bit position

4-14. draw a diagram depicting the overall organization of a CLA

4-15. determine the worst case propagation delay incurred by a practical (PLD-based) realization of a CLA

4-16. describe how a "group ripple" adder can be constructed using N-bit CLA blocks

4-17. describe the operation of an unsigned multiplier array constructed using full adders

4-18. determine the full adder arrangement and organization (rows/diagonals) needed to construct an NxM-bit unsigned multiplier array

4-19. determine the worst case propagation delay incurred by a practical (PLD-based) realization of an NxM-bit unsigned multiplier array

4-20. describe the operation of a binary coded decimal (BCD) "correction circuit"

4-21. design a BCD full adder circuit

4-22. design a BCD N-digit radix (base 10) adder/subtractor circuit

4-23. define computer architecture, programming model, and instruction set

4-24. describe the top-down specification, bottom-up implementation strategy as it pertains to the design of a computer

4-25. describe the characteristics of a "two address machine"

4-26. describe the contents of memory: program, operands, results of calculations

4-27. describe the format and fields of a basic machine instruction (opcode and address)

4-28. describe the purpose/function of each basic machine instruction (LDA, STA, ADD, SUB, AND, HLT)

4-29. define what is meant by "assembly-level" instruction mnemonics

4-30. draw a diagram of a simple computer, showing the arrangement and interconnection of each functional block

4-31. trace the execution of a computer program, identifying each step of an instruction’s microsequence (fetch and execute cycles)

4-32. distinguish between synchronous and combinational system control signals

4-33. describe the operation of memory and the function of its control signals: MSL, MOE, and MWE

4-34. describe the operation of the program counter (PC) and the function of its control signals: ARS, PCC, and POA

4-35. describe the operation of the instruction register (IR) and the function of its control signals: IRL and IRA

4-36. describe the operation of the ALU and the function of its control signals: ALE, ALX, ALY, and AOE

4-37. describe the operation of the instruction decoder/microsequencer and derive the system control table

4-38. describe the basic hardware-imposed system timing constraints: only one device can drive a bus during a given machine cycle, and data cannot pass through more than one flip-flop (register) per cycle

4-39. discuss how the instruction register can be loaded with the contents of the memory location pointed to be the program counter *and* the program counter can be incremented on the same clock edge

4-40. modify a reference ALU design to perform different functions (e.g., shift and rotate)

4-41. describe how input/output instructions can be added to the base machine architecture

4-42. describe the operation of the I/O block and the function of its control signals: IOR and IOW

4-43. compare and contrast the operation of OUT instructions with and without a transparent latch as an integral part of the I/O block

4-44. compare and contrast “jump” and “branch” transfer-of-control instructions along with the architectural features needed to support them

4-45. distinguish conditional and unconditional branches

4-46. describe the basis for which a conditional branch is “taken” or “not taken”

4-47. describe the changes needed to the instruction decoder/microsequencer in order to dynamically change the number of instruction execute cycles based on the opcode

4-48. compare and contrast the machine’s asynchronous reset (“START”) with the synchronous state counter reset (“RST”)

4-49. describe the operation of a stack mechanism (LIFO queue)

4-50. describe the operation of the stack pointer (SP) register and the function of its control signals: ARS, SPI, SPD, SPA

4-51. compare and contrast the two possible stack conventions: SP pointing to the top stack item vs. SP pointing to the top stack item

4-52. describe how stack manipulation instructions (PSH/POP) can be added to the base machine architecture

4-53. discuss the consequences of having an unbalanced set of PSH and POP instructions in a given program

4-54. discuss the reasons for using a stack as a subroutine linkage mechanism: arbitrary nesting of subroutine calls, passing parameters to subroutines, recursion, and reentrancy

4-55. describe how subroutine linkage instructions (JSR/RTS) can be added to the base machine architecture

4-56. analyze the effect of changing the stack convention utilized (SP points to top stack item vs. next available location) on instruction cycle counts

# Lecture Summary – Module 4-A

*Signed Number Notation*

**Reference:** *Digital Design Principles and Practices* (4th Ed.) pp. 39-43, (5th Ed.) pp. 44-48

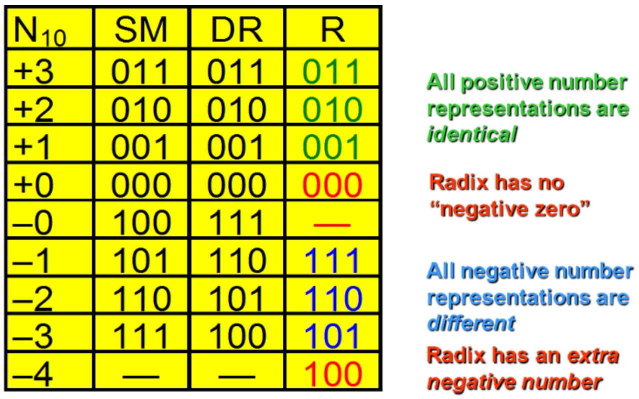
- **overview – signed number notations**
  - **sign and magnitude (SM)**
  - **diminished radix (DR)**
  - **radix (R)**
  - **only negative numbers are different – positive numbers are the same in all 3 notations**

- **sign and magnitude**
  - **vacuum tube vintage**
  - **left-most ("most significant") digit is sign bit**
    - **0 → positive**
    - **R-1 → negative (where R is *radix* or *base* of number)**
  - **positive-negative pairs are called *sign and magnitude complements* of each other**
  - **negation method: replace sign digit ($n_s$) with R-1-$n_s$**

- **diminished radix**
  - **most significant digit is still sign bit**
  - **positive-negative pairs are called *diminished radix complements* of each other**
  - **negation method: subtract each digit (including $n_s$) from R-1, i.e. $-(N)_R = (R^n-1)_R - (N)_R$**

- **radix**
  - **most significant digit is still sign bit**
  - **positive-negative pairs are called *radix complements* of each other**
  - **negation method: add one to the DR complement of $(N)_R$, i.e. $-(N)_R = (R^n)_R - (N)_R$**

- **comparison (3-bit signed numbers, each notation):**



| $N_{10}$ | SM | DR | R |
|---|---|---|---|
| +3 | 011 | 011 | 011 |
| +2 | 010 | 010 | 010 |
| +1 | 001 | 001 | 001 |
| +0 | 000 | 000 | 000 |
| –0 | 100 | 111 | — |
| –1 | 101 | 110 | 111 |
| –2 | 110 | 101 | 110 |
| –3 | 111 | 100 | 101 |
| –4 | — | — | 100 |

All positive number representations are *identical*

Radix has no "negative zero"

All negative number representations are *different*

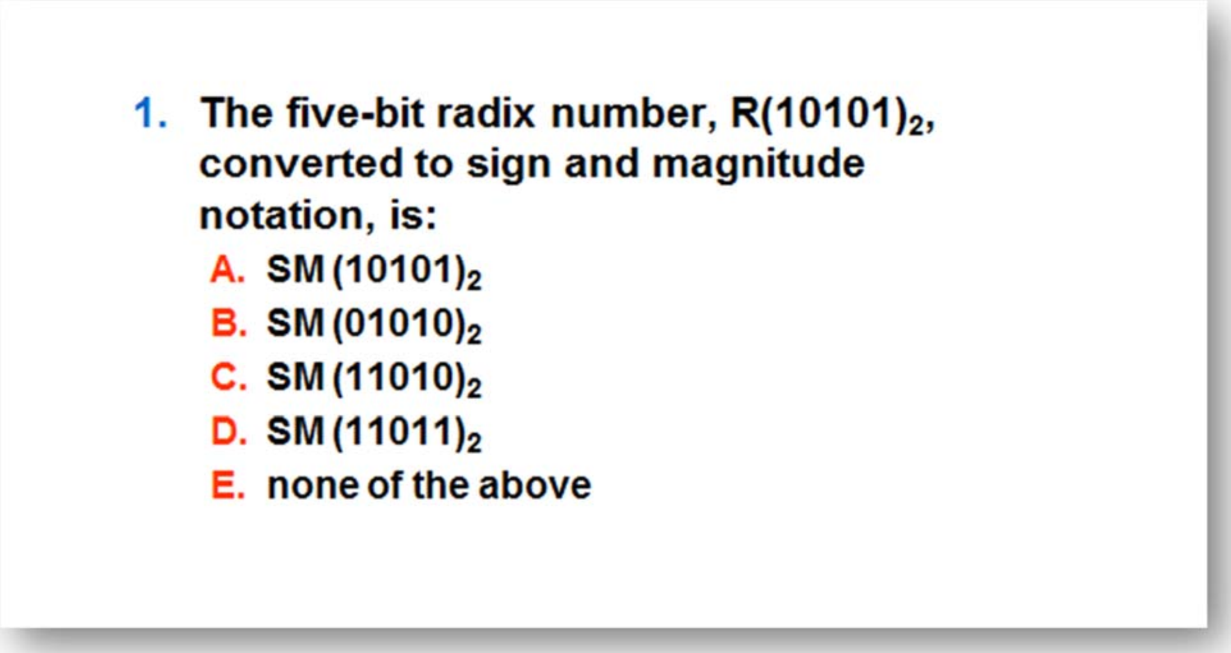
Radix has an *extra negative number*

**<u>Observations</u>:**

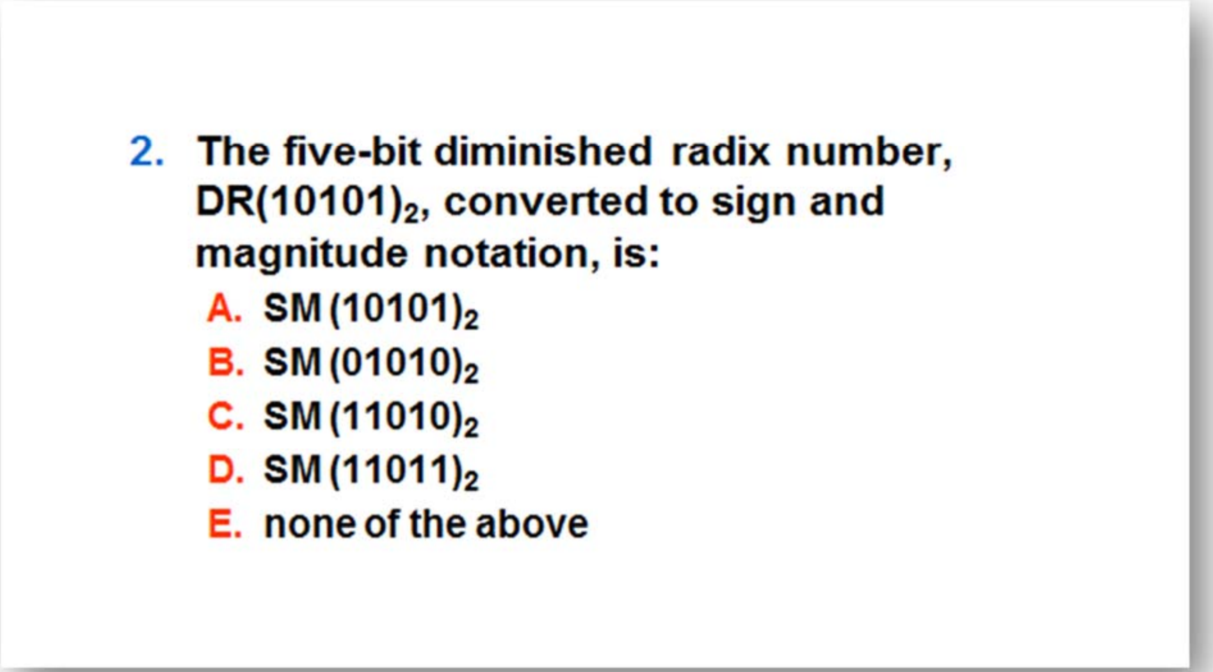
1. **SM and DR have a balanced set of positive and negative numbers (as well as +0 and -0)**
2. **R notation has a single representation for zero, which results in an "extra negative number" – this unbalanced set of positive and negative numbers can lead to round-off errors in numeric computations**
3. **Virtually all computers in service today use R notation**

- **simplifications for binary (base 2)**
  - **SM: complement sign position (0 ↔ 1)**
  - **DR (also called 1's complement): complement each bit**
  - **R (also called 2's complement):**
    - **add 1 to DR complement -or-**
    - **scan number from right to left and complement each bit to the left of the first "1" encountered**

- **sign extension: SM – pad magnitude with leading zeroes; R and DR – replicate the sign digit**



1. The five-bit radix number, R$(10101)_2$, converted to sign and magnitude notation, is:
   - A. SM $(10101)_2$
   - B. SM $(01010)_2$
   - C. SM $(11010)_2$
   - D. SM $(11011)_2$
   - E. none of the above



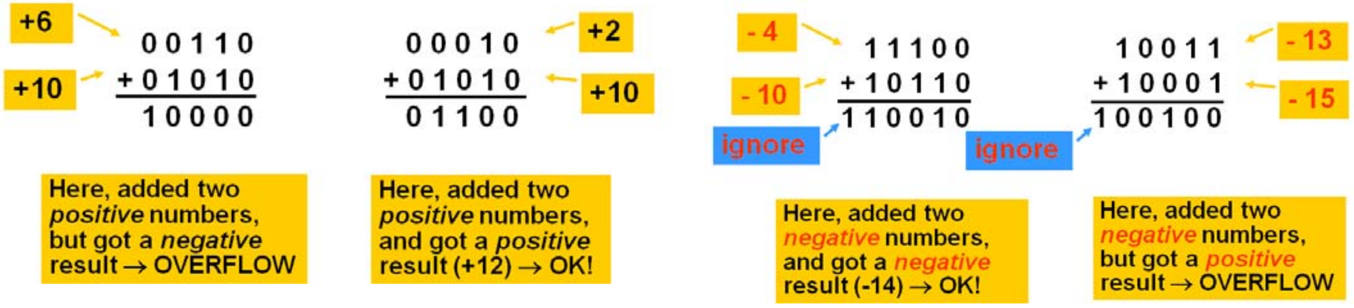
2. The five-bit diminished radix number, DR$(10101)_2$, converted to sign and magnitude notation, is:
   - A. SM $(10101)_2$
   - B. SM $(01010)_2$
   - C. SM $(11010)_2$
   - D. SM $(11011)_2$
   - E. none of the above

# Lecture Summary – Module 4-B

## *Radix Addition and Subtraction*

**Reference:** *Digital Design Principles and Practices* (4th Ed.) pp. 39-43, (5th Ed.) pp. 48-52

- **radix addition**
  - **method: add all digits, including the sign digits; ignore any carry out of the sign position**
  - **note that overflow can occur, since we are working with numbers of *fixed length***
    - **overflow occurs if two numbers of *like sign* are added and a result with the *opposite sign* is obtained**
    - **overflow cannot occur when adding numbers of opposite sign**
    - **another way to detect overflow: if the carry *in* to the sign position is *different* than the carry *out* of the sign position, then overflow has occurred**
    - **when overflow occurs, there is *no valid numeric result***



```
  +6     0 0 1 1 0        0 0 0 1 0   +2        - 4     1 1 1 0 0          1 0 0 1 1   - 13
 +10   + 0 1 0 1 0      + 0 1 0 1 0   +10       - 10  + 1 0 1 1 0        + 1 0 0 0 1   - 15
         1 0 0 0 0        0 1 1 0 0          ignore   1 1 0 0 1 0  ignore  1 0 0 1 0 0
```

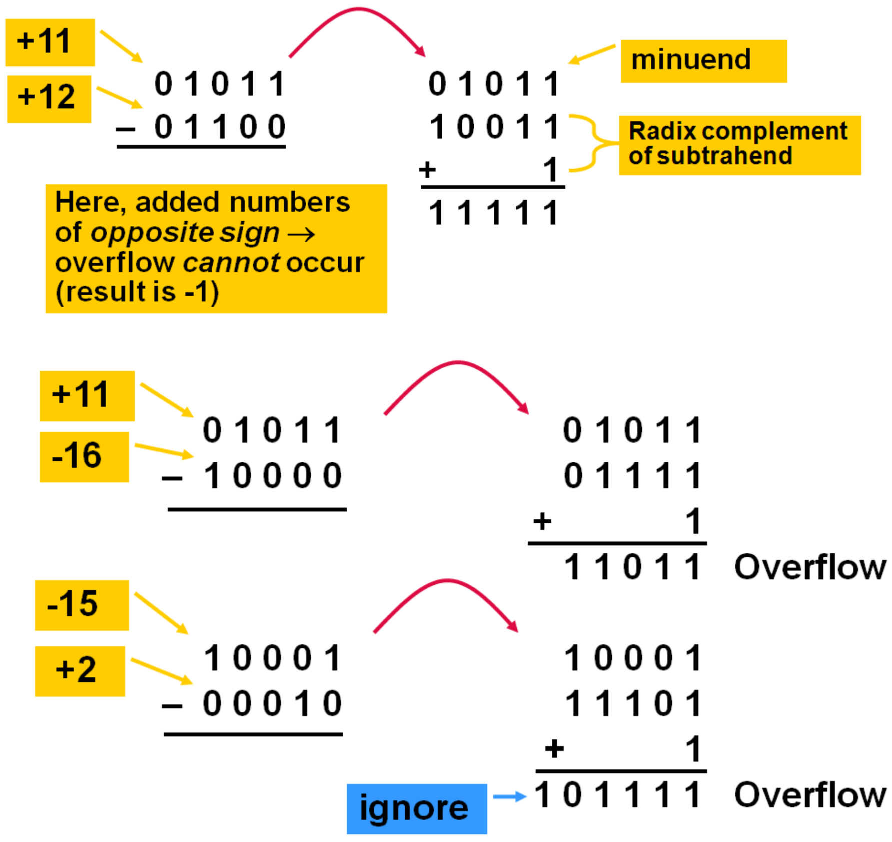
Here, added two *positive* numbers, but got a *negative* result → OVERFLOW

Here, added two *positive* numbers, and got a *positive* result (+12) → OK!

Here, added two *negative* numbers, and got a *negative* result (-14) → OK!

Here, added two *negative* numbers, but got a *positive* result → OVERFLOW

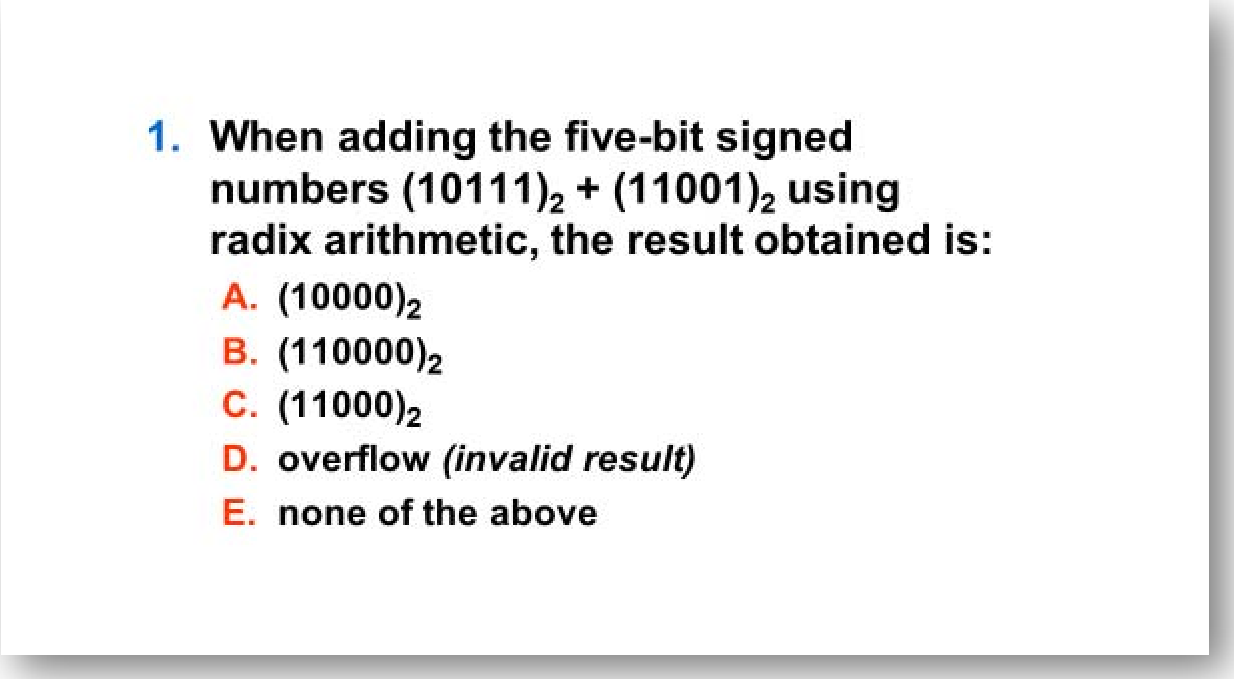
- **radix subtraction**
  - **method: form the radix complement of the subtrahend and ADD (the same rules for overflow detection apply)**



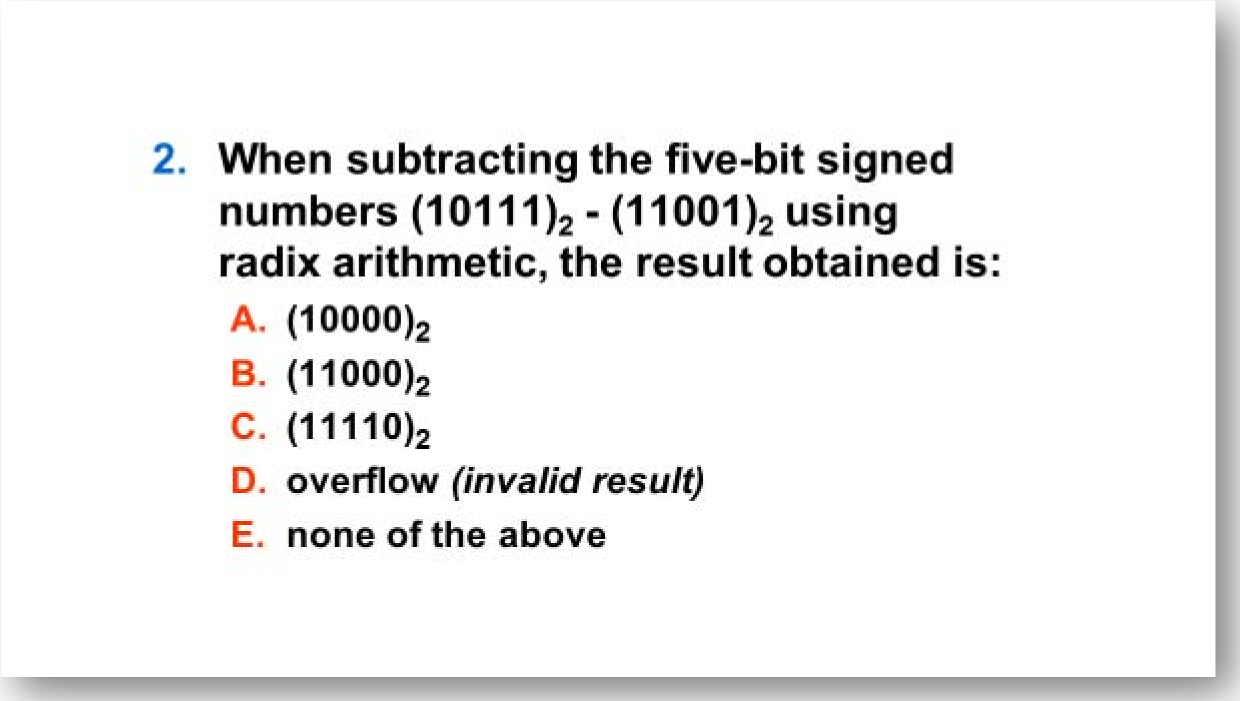
```
 +11     0 1 0 1 1          0 1 0 1 1    minuend
 +12   – 0 1 1 0 0          1 0 0 1 1  } Radix complement
                          +         1  } of subtrahend
                            1 1 1 1 1
```

Here, added numbers of *opposite sign* → overflow *cannot* occur (result is -1)

```
 +11     0 1 0 1 1          0 1 0 1 1
 -16   – 1 0 0 0 0          0 1 1 1 1
                          +         1
                            1 1 0 1 1   Overflow

 -15     1 0 0 0 1          1 0 0 0 1
 +2    – 0 0 0 1 0          1 1 1 0 1
                          +         1
              ignore →    1 0 1 1 1 1   Overflow
```



1. When adding the five-bit signed numbers $(10111)_2 + (11001)_2$ using radix arithmetic, the result obtained is:
   - A. $(10000)_2$
   - B. $(110000)_2$
   - C. $(11000)_2$
   - D. overflow *(invalid result)*
   - E. none of the above



2. When subtracting the five-bit signed numbers $(10111)_2 - (11001)_2$ using radix arithmetic, the result obtained is:
   - A. $(10000)_2$
   - B. $(11000)_2$
   - C. $(11110)_2$
   - D. overflow *(invalid result)*
   - E. none of the above

# Lecture Summary – Module 4-C

## *Adder, Subtractor, and Comparator Circuits*

**Reference:** *DDPP* (4th Ed.) pp. 458-466, 474-478; (5th Ed.) pp. 331-339, 341-345, 372-375

- **overview**
  - **an adder circuit combines two operands based on rules described in 5-C**
  - **same addition rules apply for both signed (2's complement) and unsigned numbers**
  - **subtraction performed by taking complement of subtrahend and performing add**

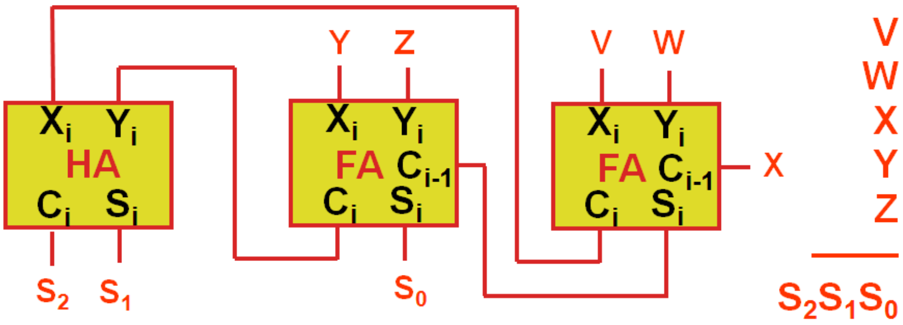
- **building blocks**
  - **half adder**

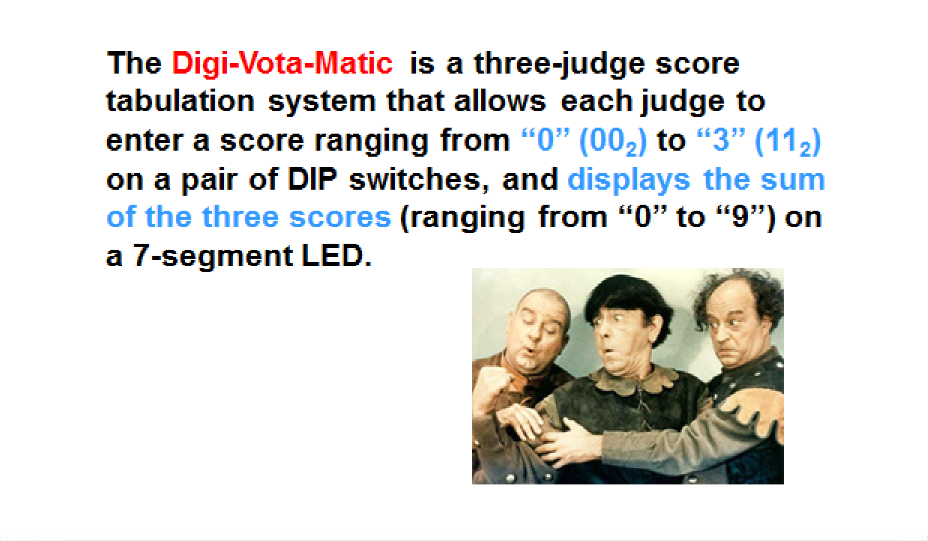
| Xi | Yi | Ci | Si |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

  - **full adder**

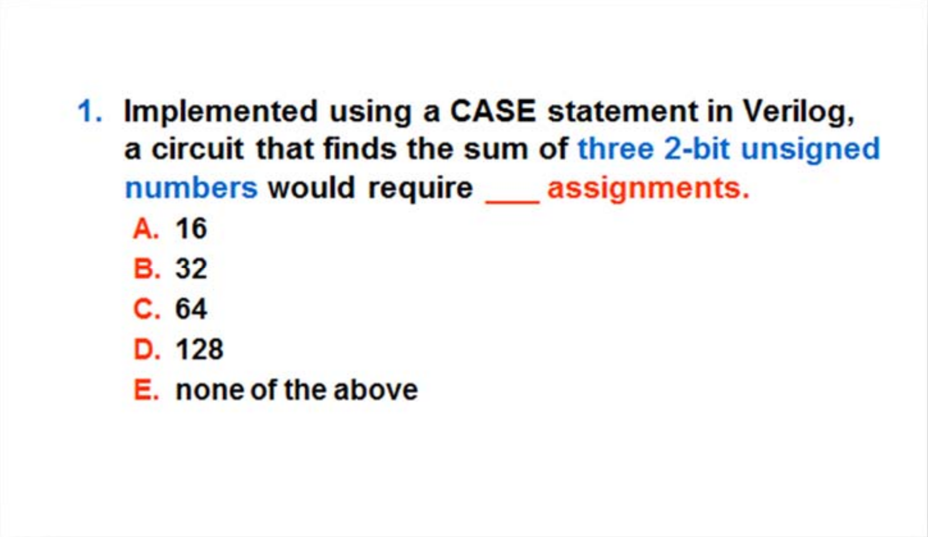
| Xi | Yi | Ci-1 | Ci | Si |
|---|---|---|---|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

  - **"vote counting" application**

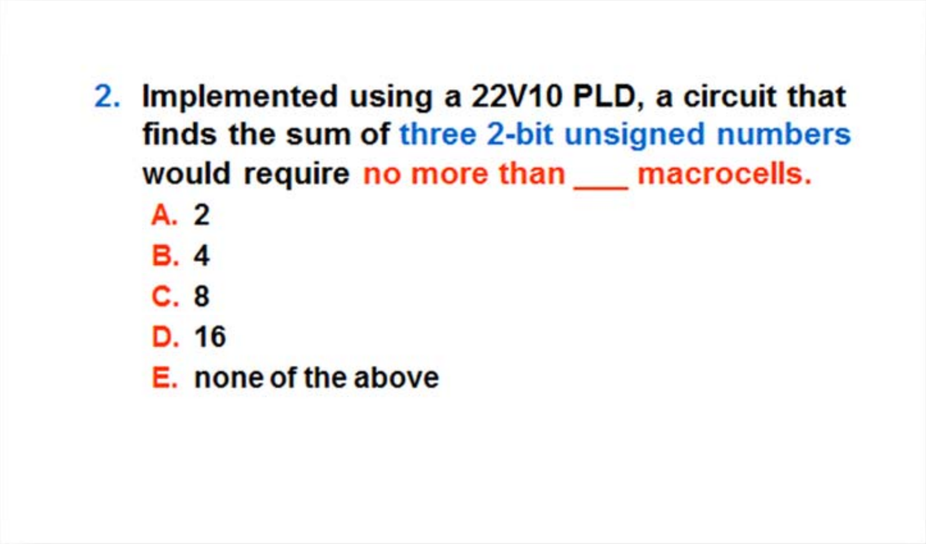




The Digi-Vota-Matic is a three-judge score tabulation system that allows each judge to enter a score ranging from “0” ($00_2$) to “3” ($11_2$) on a pair of DIP switches, and displays the sum of the three scores (ranging from “0” to “9”) on a 7-segment LED.

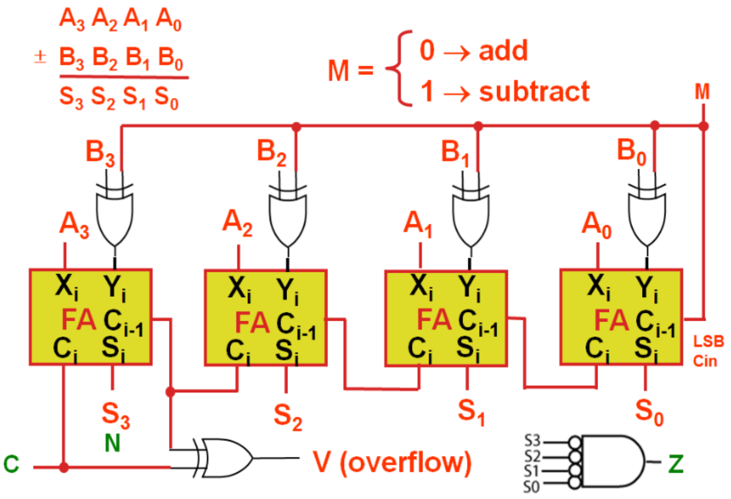


1. Implemented using a CASE statement in Verilog, a circuit that finds the sum of three 2-bit unsigned numbers would require ___ assignments.
   A. 16
   B. 32
   C. 64
   D. 128
   E. none of the above



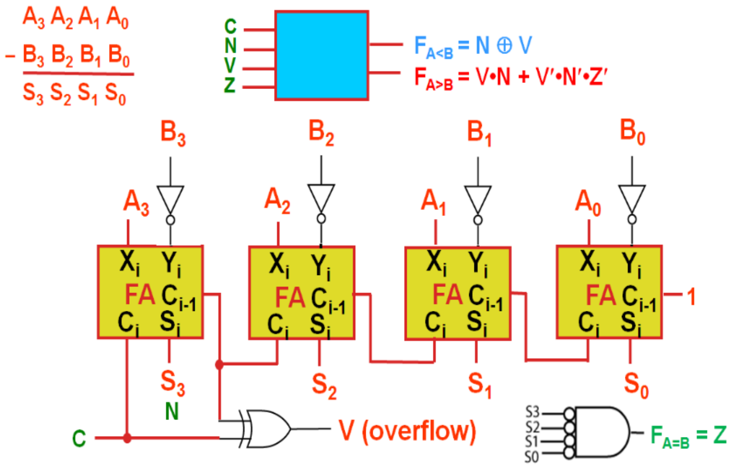
2. Implemented using a 22V10 PLD, a circuit that finds the sum of three 2-bit unsigned numbers would require no more than ___ macrocells.
   A. 2
   B. 4
   C. 8
   D. 16
   E. none of the above

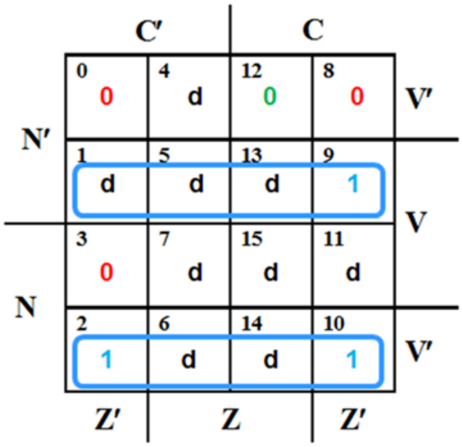
- **multi-digit adder/subtractor**
  - **ripple = iterative**
  - **to subtract, take DR radix complement of subtrahend and add 1**
  - **conditions of interest ("condition codes")**
    - **overflow (V)**
    - **negative (N)**
    - **zero (Z)**
    - **carry/borrow (C)**

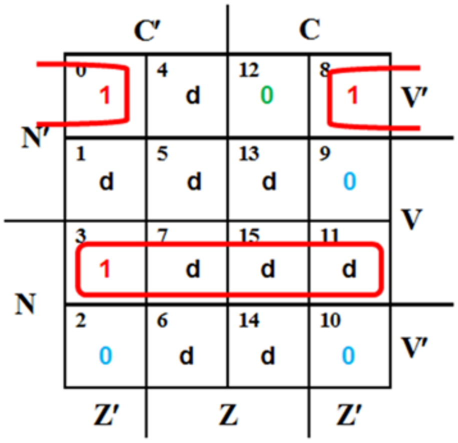


- **magnitude comparator**
  - **calculate A–B and condition codes**
  - **results (A=B, A<B, A>B) are *functions of the condition codes***

| A1 | A0 | B1 | B0 | ? | C | Z | N | V |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | A=B | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | A<B | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | A>B | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | A>B | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | A>B | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | A=B | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | A>B | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | A>B | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | A<B | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | A<B | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | A=B | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | A<B | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | A<B | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | A<B | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | A>B | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | A=B | 1 | 1 | 0 | 0 |

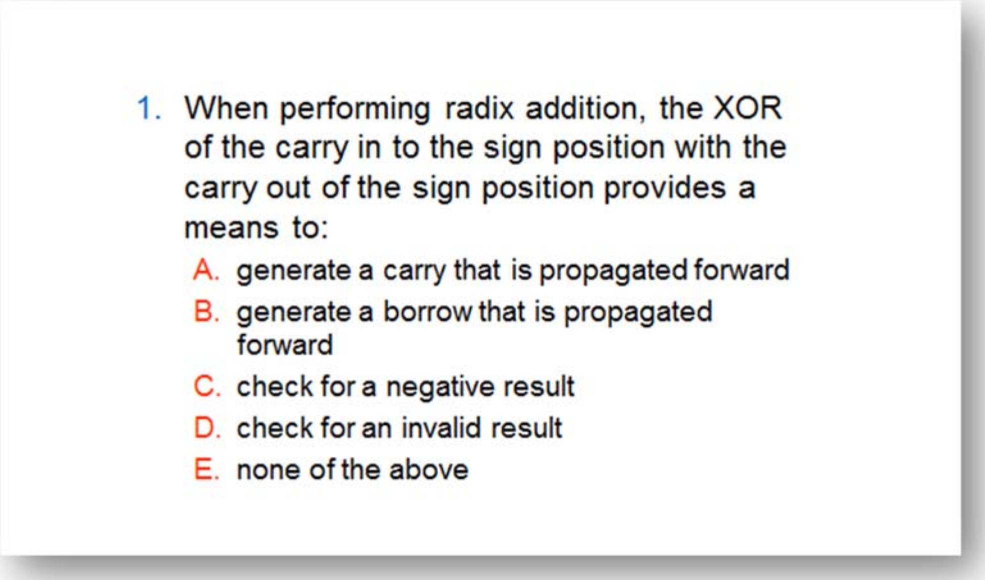






$F_{A<B} = N \oplus V$

$F_{A>B} = V \bullet N + V' \bullet N' \bullet Z'$



1. When performing radix addition, the XOR of the carry in to the sign position with the carry out of the sign position provides a means to:

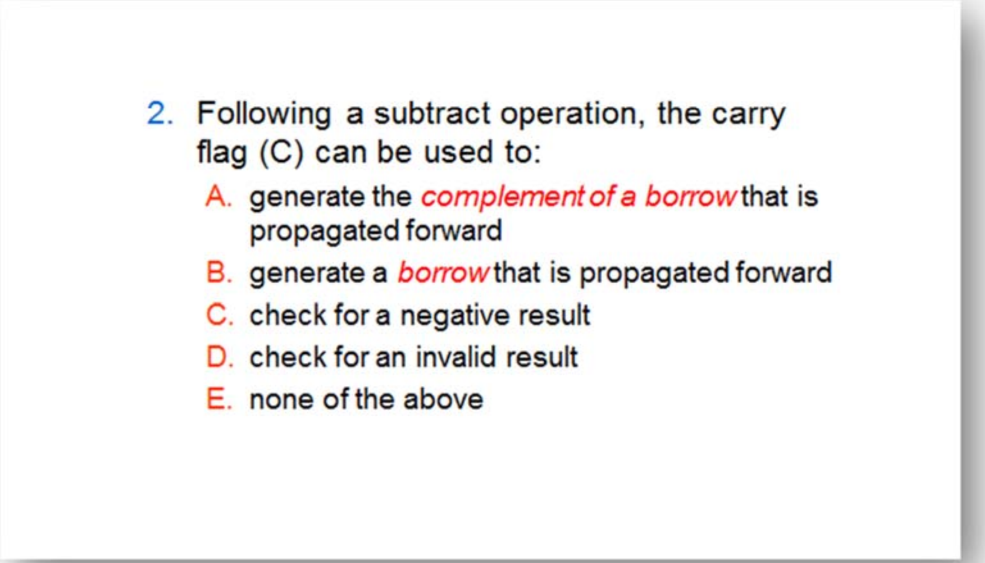
A. generate a carry that is propagated forward

B. generate a borrow that is propagated forward

C. check for a negative result

D. check for an invalid result

E. none of the above



2. Following a subtract operation, the carry flag (C) can be used to:

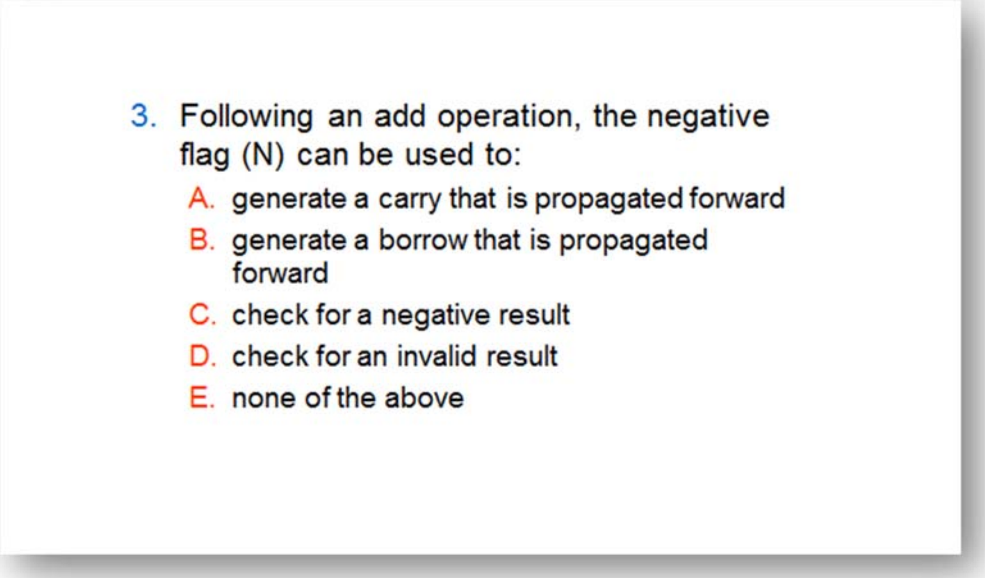
A. generate the *complement of a borrow* that is propagated forward

B. generate a *borrow* that is propagated forward

C. check for a negative result

D. check for an invalid result

E. none of the above



3. Following an add operation, the negative flag (N) can be used to:

A. generate a carry that is propagated forward

B. generate a borrow that is propagated forward

C. check for a negative result

D. check for an invalid result

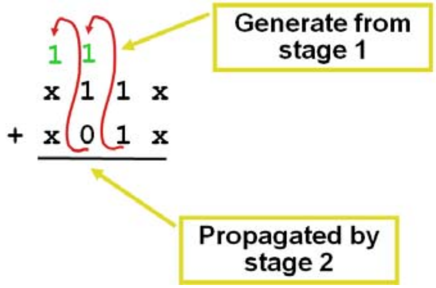
E. none of the above

# Lecture Summary – Module 4-D

## *Carry Look-Ahead (CLA) Adder Circuits*

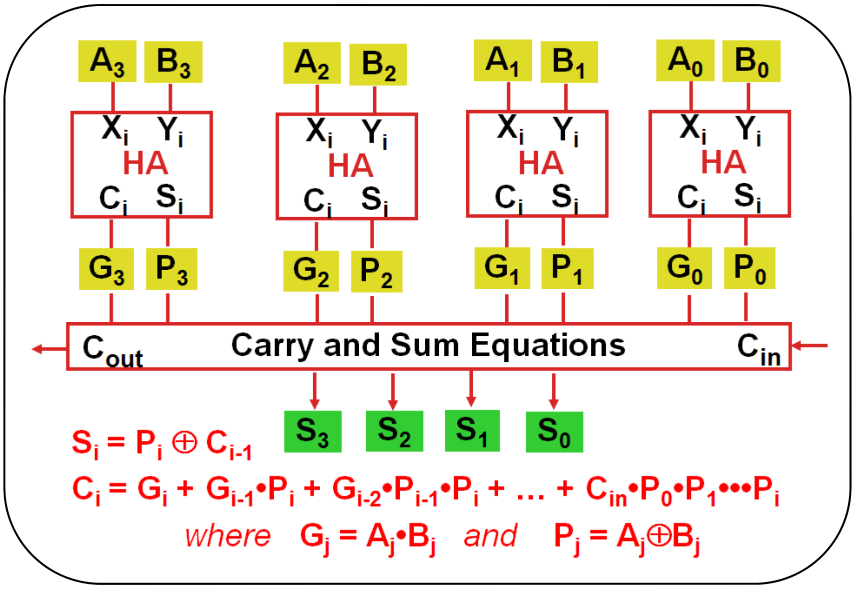
**Reference:** *DDPP* (4th Ed.) pp. 478-482, 484-488; (5th Ed.) pp. 376-383, 384-386

- **introduction**
  - **previously considered iterative ("ripple") adder circuit**
  - **problem: propagation delay increases with number of bits**
  - **solution: determine carries in parallel rather than iteratively → significant speedup**
  - **"look-ahead" → "anticipated"**



- **definitions and derivations**
  - **generate function (carry *guaranteed*) $G_i = X_i \bullet Y_i$**
  - **propagate function (carry *in* propagated *out*) $P_i = X_i \oplus Y_i$**
  - **note that a "PG box" is just a half-adder (HA)**
  - **can rewrite sum bit equation as $S_i = P_i \oplus C_{i-1}$ ($C_{-1}$ is $C_{in}$)**
  - **can rewrite carry out equation as $C_i = Gi + C_{i-1} \bullet P_i$**

- **rewriting carry equations for 4-bit adder in terms of P's and G's**
  - $C_{-1} = C_{in}$
  - $C_0 = G_0 + C_{in} \bullet P_0$
  - $C_1 = G_1 + C_0 \bullet P_1$
  - $C_2 = G_2 + C_1 \bullet P_2$
  - $C_3 = C_{out} = G_3 + C_2 \bullet P_3$



$S_i = P_i \oplus C_{i-1}$

$C_i = G_i + G_{i-1} \bullet P_i + G_{i-2} \bullet P_{i-1} \bullet P_i + \ldots + C_{in} \bullet P_0 \bullet P_1 \bullet\bullet\bullet P_i$

*where* $G_j = A_j \bullet B_j$ *and* $P_j = A_j \oplus B_j$

- **rewriting carry equations for 4-bit adder in terms of *available inputs* (successive expansion)**
  - $C_{-1} = C_{in}$
  - $C_0 = G_0 + C_{in} \bullet P_0$
  - $C_1 = G_1 + C_0 \bullet P_1 = G_1 + (G_0 + C_{in} \bullet P_0) \bullet P_1 = G_1 + G_0 \bullet P_1 + C_{in} \bullet P_0 \bullet P_1$
  - ***know what these equations are "saying"***
  - $C_2$ = ______________________________________________
  - $C_3$ = ______________________________________________

- **observations**
  - **regardless of adder length (number of operand bits), the time required to produce any sum digit is the same (i.e. they are all produced *in parallel*)**
  - **large CLA adders are difficult to build in practice because of "product term explosion"**
  - **reasonable compromise is to make a group ripple adder (cascading m-bit CLA blocks together to get desired operand length)**

- **4-bit CLA realized in Verilog**

```
module cla4(X, Y, CIN, S);

 input wire [3:0] X, Y;     // Operands
 input wire CIN;            // Carry in
 output wire [3:0] S;       // Sum outputs

 wire [3:0] C;              // Carry equations (C[3] is Cout)
 wire [3:0] P, G;

 assign G = X & Y;          // Generate functions G[0] = X[0]&Y[0];
                                                  G[1] = .. so on
 assign P = X ^ Y;          // Propagate functions P[0] = X[0]^Y[0];
                                                  P[1] = .. so on

 // Carry function definitions
 assign C[0] = G[0] | CIN & P[0];
 assign C[1] = G[1] | G[0] & P[1] | CIN & P[0] & P[1];
 assign C[2] = G[2] | G[1] & P[2] | G[0] & P[1] & P[2]
                    | CIN & P[0] & P[1] & P[2];
 assign C[3] = G[3] | G[2] & P[3] | G[1] & P[2] & P[3]
                    | G[0] & P[1] & P[2] & P[3]
                    | CIN & P[0] & P[1] & P[2] & P[3];

 assign S[0] = CIN ^ P[0];
 assign S[3:1] = C[2:0] ^ P[3:1];

endmodule
```

- **alternate version using "+" (addition) operator**

```
module cla4p(X, Y, CIN, S);

  input wire [3:0] X, Y;    // Operands
  input wire CIN;           // Carry in
  output wire [3:0] S;      // Sum outputs

  assign S = X + Y + {3'b000,CIN};

endmodule
```

- **identical timing analysis for both versions → "+" operator synthesizes CLA equations**

```
Timing Analysis for ispMACH 4256ZE 5.8 ns CPLD

  Delay      Level     Source      Destination
  =====      =====     ======      ===========
   6.40        1        CIN            S3
   6.40        1         X0            S3
   6.40        1         Y0            S3
   6.35        1         X1            S3
   6.35        1         Y1            S3
   6.30        1         X2            S3
   6.30        1         Y2            S3
   6.25        1         Y3            S3
```

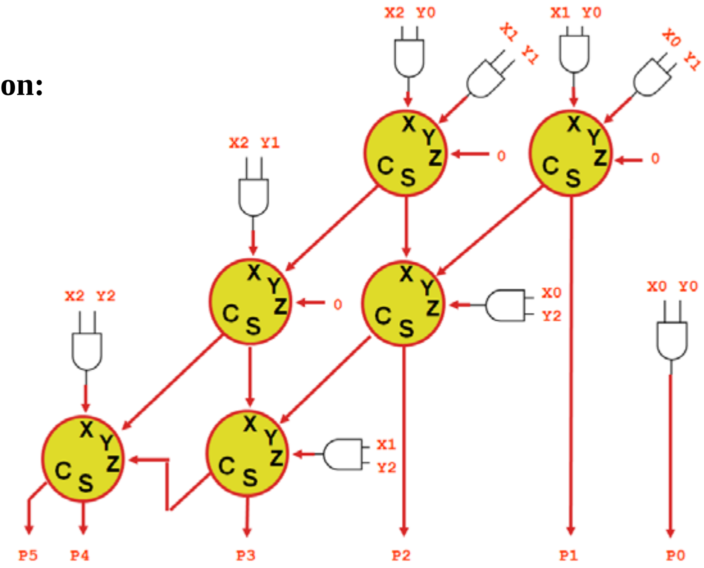
# Lecture Summary – Module 4-E

## *Multiplier Circuits*

**Reference:** *DDPP* (4th Ed.) pp. 45-47, 494-497; (5th Ed.) pp. 54-56, 416-419

- **overview**
  - o **consider 3x3 unsigned binary multiplication:**

```
Multiplicand:                X2    X1    X0
Multiplier:                × Y2    Y1    Y0
                    ------------------------
                           X2Y0  X1Y0  X0Y0
                     X2Y1  X1Y1  X0Y1
               X2Y2  X1Y2  X0Y2
                    ------------------------
     P5    P4    P3    P2    P1    P0
     \_____________________________/
                 Product
```

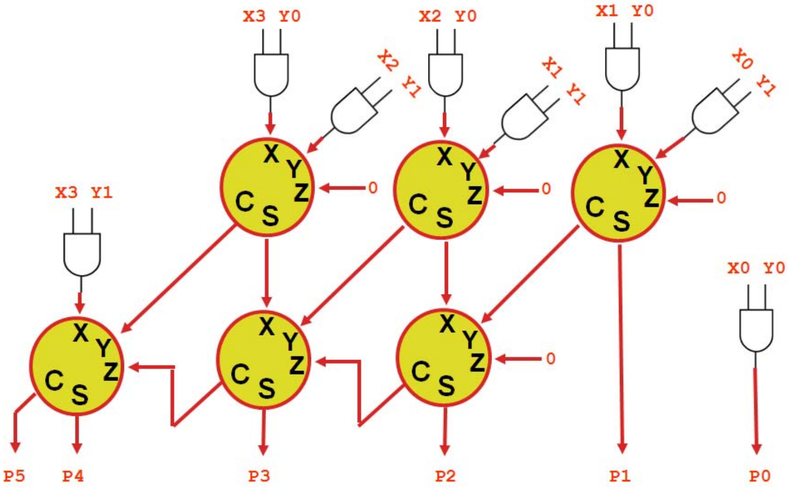


  - o **based on “shift and add” algorithm**
  - o **each row is called a *product component***
  - o **each $x_i \bullet y_j$ term represents a *product component bit* (logical AND)**
  - o **the *product* P is obtained by adding together the product components**

- **generalizations for an NxM multiplier array circuit**
  - o **N = number of bits in multiplicand**
  - o **M = number of bits in multiplier**
  - o **produces an N+M digit result**
  - o **requires NxM AND gates to generate the product components**
  - o **requires N-1 “diagonals” of full adders**
  - o **requires M rows of full adders**

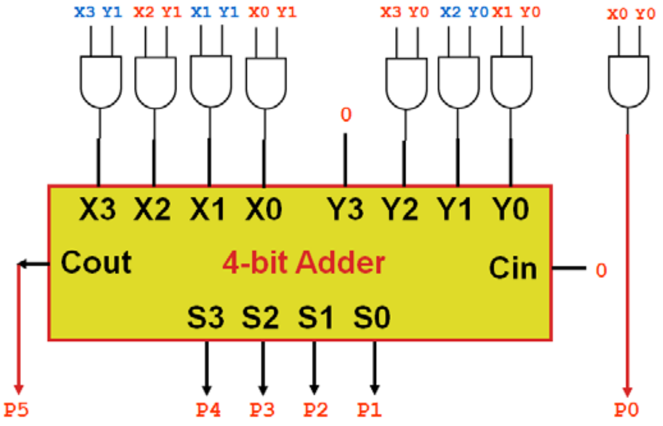
- **exercise: 4x2 multiplier array circuit**

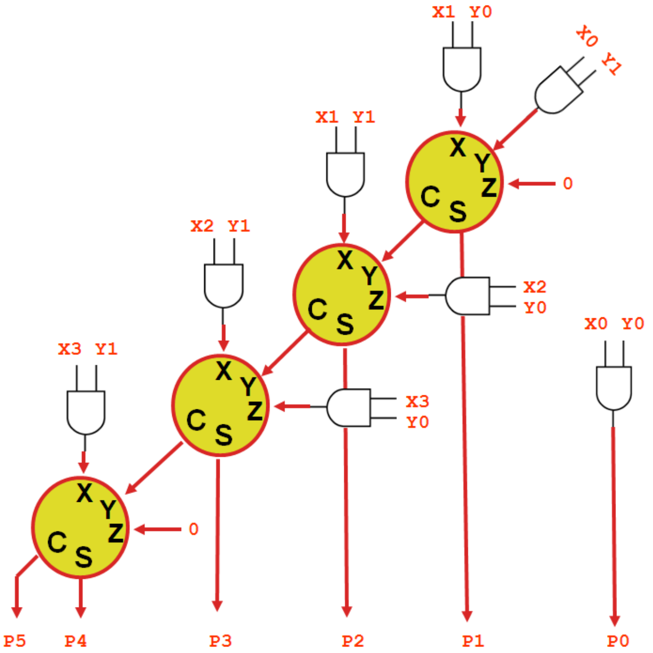
```
                X3    X2    X1    X0
                            Y1    Y0
                ----------------------
                X3Y0  X2Y0  X1Y0  X0Y0
          X3Y1  X2Y1  X1Y1  X0Y1
----------------------------------------
P5    P4    P3    P2    P1    P0
```



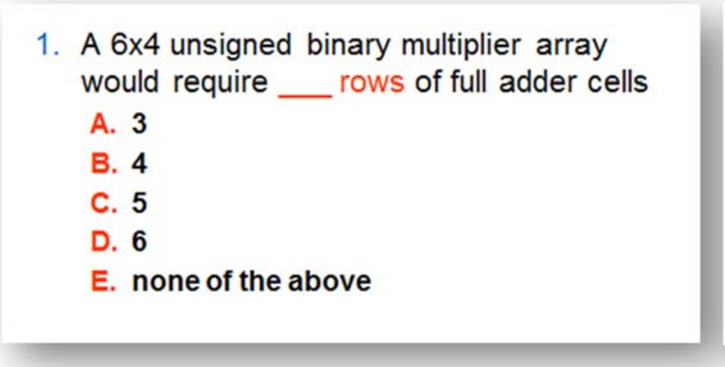
- **exercise: 2x4 multiplier array circuit**

| | | | | | |
|---|---|---|---|---|---|
| | | | | Y1 | Y0 |
| | | X3 | X2 | X1 | X0 |
| | | | | X0Y1 | X0Y0 |
| | | | X1Y1 | X1Y0 | |
| | | X2Y1 | X2Y0 | | |
| | X3Y1 | X3Y0 | | | |
| P5 | P4 | P3 | P2 | P1 | P0 |



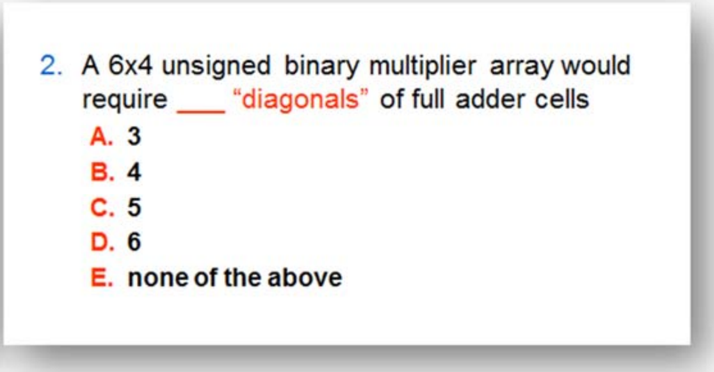


- **generalizations for an NxM multiplier**
  - **N = number of bits in multiplicand (top)**
  - **M = number of bits in multiplier (bottom)**
  - **produces an N+M digit result**
  - **requires NxM AND gates to generate the product components**
  - **requires N–1 *diagonals* of full adders**
  - **requires M rows of full adders**



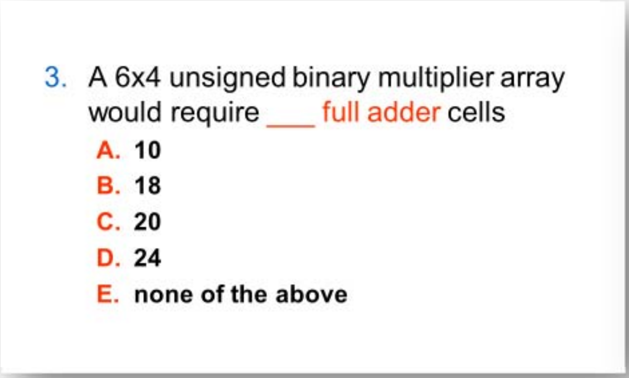
1. A 6x4 unsigned binary multiplier array would require ___ rows of full adder cells

   A. 3
   B. 4
   C. 5
   D. 6
   E. none of the above

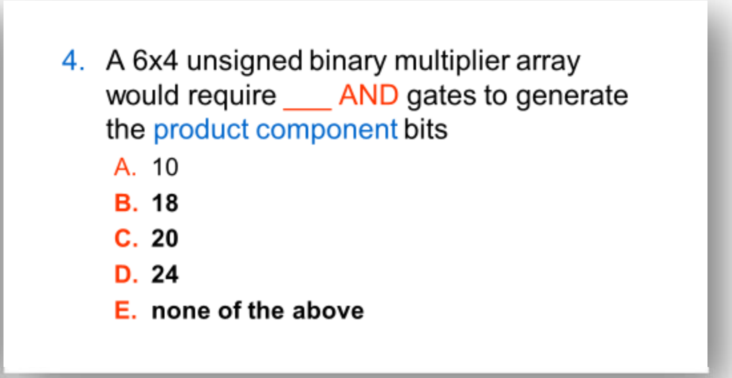


2. A 6x4 unsigned binary multiplier array would require ___ “diagonals” of full adder cells

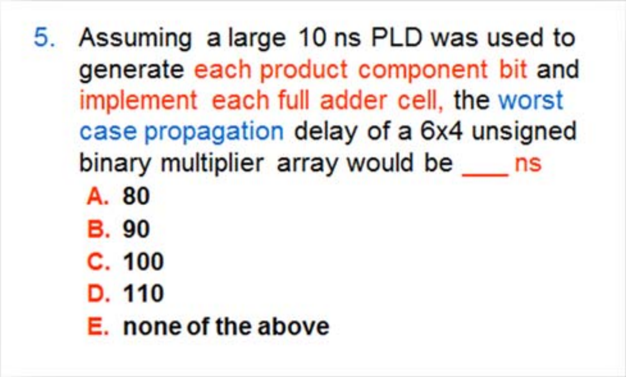
   A. 3
   B. 4
   C. 5
   D. 6
   E. none of the above



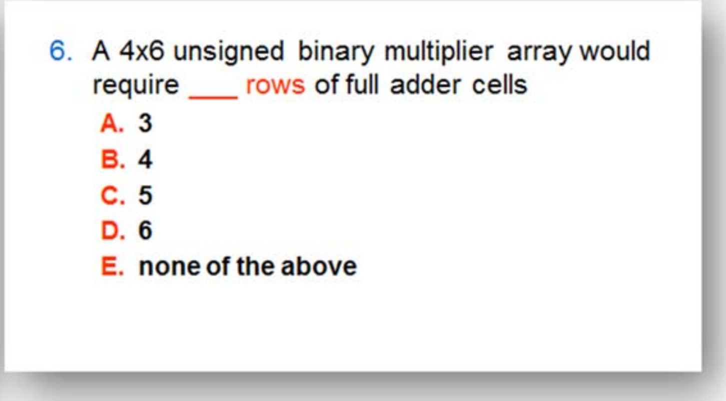
3. A 6x4 unsigned binary multiplier array would require ___ full adder cells
   - A. 10
   - B. 18
   - C. 20
   - D. 24
   - E. none of the above



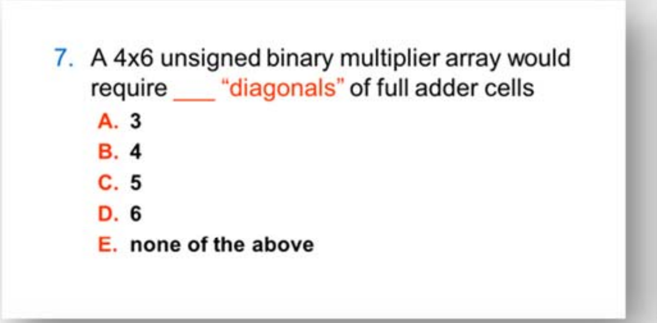
4. A 6x4 unsigned binary multiplier array would require ___ AND gates to generate the product component bits
   - A. 10
   - B. 18
   - C. 20
   - D. 24
   - E. none of the above



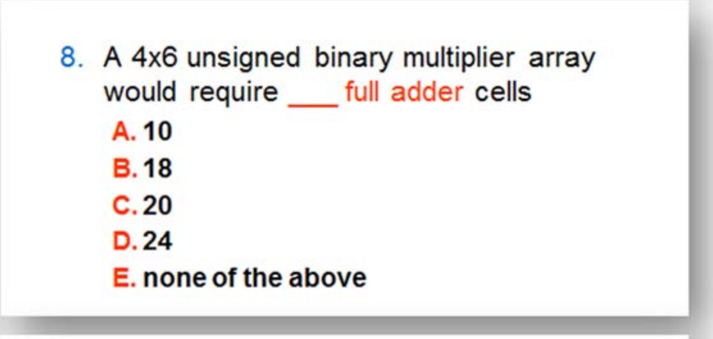
5. Assuming a large 10 ns PLD was used to generate each product component bit and implement each full adder cell, the worst case propagation delay of a 6x4 unsigned binary multiplier array would be ___ ns
   - A. 80
   - B. 90
   - C. 100
   - D. 110
   - E. none of the above



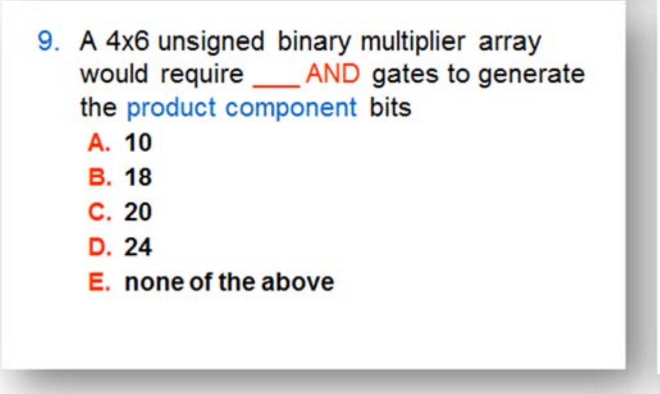
6. A 4x6 unsigned binary multiplier array would require ___ rows of full adder cells
   - A. 3
   - B. 4
   - C. 5
   - D. 6
   - E. none of the above



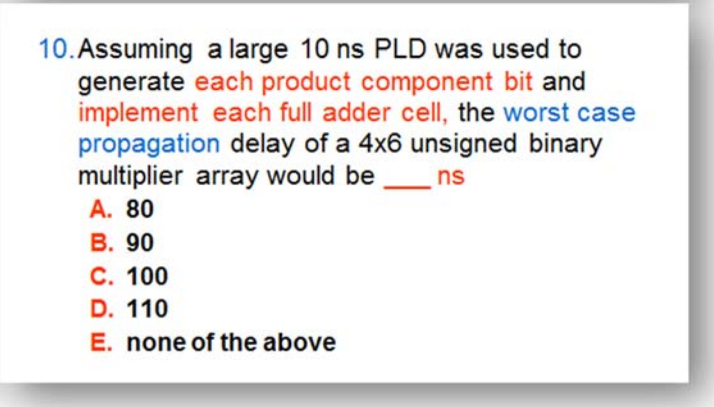
7. A 4x6 unsigned binary multiplier array would require ___ "diagonals" of full adder cells
   - A. 3
   - B. 4
   - C. 5
   - D. 6
   - E. none of the above



8. A 4x6 unsigned binary multiplier array would require ___ full adder cells
   - A. 10
   - B. 18
   - C. 20
   - D. 24
   - E. none of the above



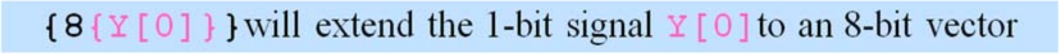
9. A 4x6 unsigned binary multiplier array would require ___ AND gates to generate the product component bits
   - A. 10
   - B. 18
   - C. 20
   - D. 24
   - E. none of the above



10. Assuming a large 10 ns PLD was used to generate each product component bit and implement each full adder cell, the worst case propagation delay of a 4x6 unsigned binary multiplier array would be ___ ns
   - A. 80
   - B. 90
   - C. 100
   - D. 110
   - E. none of the above

- **realizations in Verilog**
  - **use *expressions* to define product components**
  - **use *addition operator* (+) to form unsigned sum of product components**
  - **example: 4x4 multiplier array circuit**

```
/* 4x4 Combinational Multiplier */
module mul4x4(X, Y, P);

  input wire [3:0] X, Y;     // Multiplicand, multiplier
  output wire [7:0] P;       // Product bits

  wire [7:0] PC[3:0];        // Four 8-bit variables

  assign PC[0] = {8{Y[0]}} & {4'b0, X};        // 0000X3X2X1X0
  assign PC[1] = {8{Y[1]}} & {3'b0, X, 1'b0}; // 000X3X2X1X00
  assign PC[2] = {8{Y[2]}} & {2'b0, X, 2'b0}; // 00X3X2X1X000
  assign PC[3] = {8{Y[3]}} & {1'b0, X, 3'b0}; // 0X3X2X1X0000

  assign P = PC[0] + PC[1] + PC[2] + PC[3];

endmodule
```



{8{Y[0]}}will extend the 1-bit signal Y[0] to an 8-bit vector

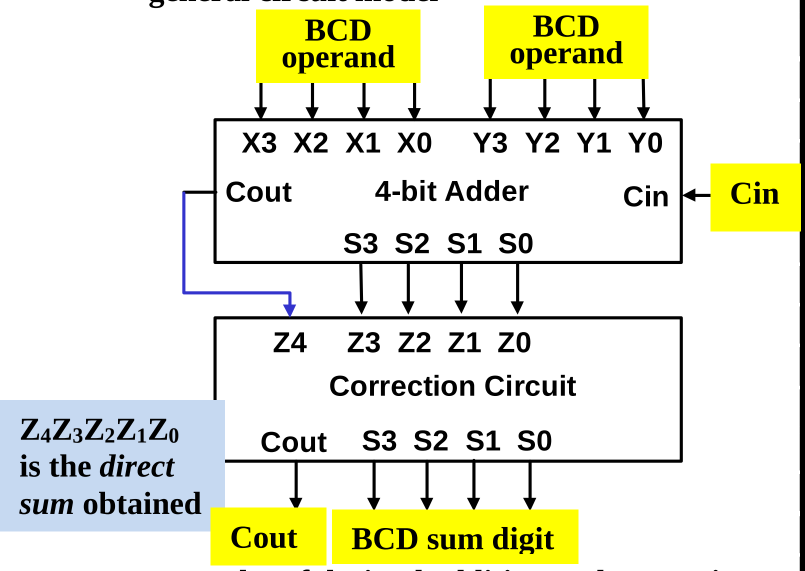
```
Timing Analysis for ispMACH 4256ZE 5.8 ns CPLD

  Delay    Level    Source     Destination
  =====    =====    ======     ===========
   6.50      1        X0            P4
   6.50      1        X0            P5
   6.50      1        X1            P4
   6.50      1        X1            P5
   6.50      1        X2            P4
   6.50      1        X2            P5
   6.50      1        Y0            P4
   6.50      1        Y0            P5
   6.50      1        Y1            P4
   6.50      1        Y1            P5
   6.50      1        Y2            P4
   6.50      1        Y2            P5
   6.45      1        X3            P4
   6.45      1        X3            P5
   6.45      1        Y3            P4
   6.45      1        Y3            P5
   6.05      1        X0            P0
   6.05      1        X0            P1
   6.05      1        X0            P2
   6.05      1        X0            P3
```

# Lecture Summary – Module 4-F

## *BCD Adder Circuits*

**Reference:** *Digital Design Principles and Practices* (4th Ed.) pp. 48-51; (5th Ed.) pp. 58-60

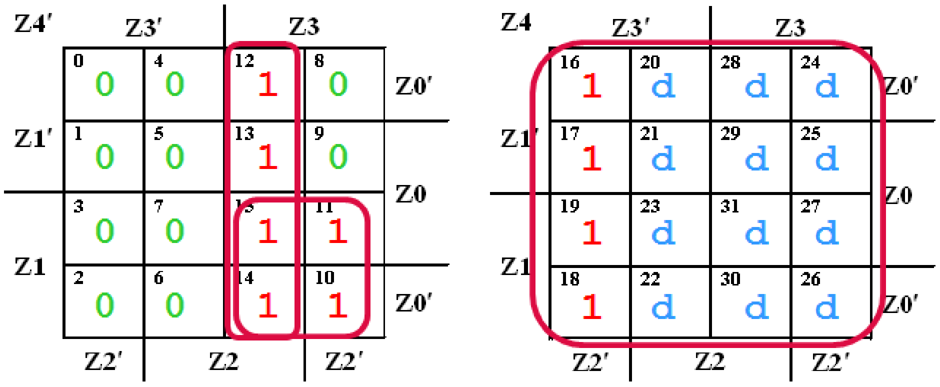
- **overview**
  - o **external computer interfaces may need to read or display decimal digits (examples)**
  - o **need to perform arithmetic operations on decimal numbers directly**
  - o **most commonly used code in *binary-coded decimal* (BCD)**
  - o **object is to design circuit that adds two BCD digit codes plus carry in, to produce a sum digit plus a carry out**
  - o **want to use standard 4-bit binary adder modules as "building blocks"**
  - o **note that there are six "unused combinations" in BCD, so potential exists for needed to perform a "correction"**
- **general circuit model**



**$Z_4Z_3Z_2Z_1Z_0$ is the *direct sum* obtained**

| $N_{10}$ | $Z_4\ Z_3\ Z_2\ Z_1\ Z_0$ | $C_{out}\ S_3\ S_2\ S_1\ S_0$ | Correction |
|---|---|---|---|
| 0 | 0 0 0 0 0 | 0 0 0 0 0 | <none> |
| 1 | 0 0 0 0 1 | 0 0 0 0 1 | <none> |
| 2 | 0 0 0 1 0 | 0 0 0 1 0 | <none> |
| 3 | 0 0 0 1 1 | 0 0 0 1 1 | <none> |
| 4 | 0 0 1 0 0 | 0 0 1 0 0 | <none> |
| 5 | 0 0 1 0 1 | 0 0 1 0 1 | <none> |
| 6 | 0 0 1 1 0 | 0 0 1 1 0 | <none> |
| 7 | 0 0 1 1 1 | 0 0 1 1 1 | <none> |
| 8 | 0 1 0 0 0 | 0 1 0 0 0 | <none> |
| 9 | 0 1 0 0 1 | 0 1 0 0 1 | <none> |
| 10 | 0 1 0 1 0 | 1 0 0 0 0 | <add 6> |
| 11 | 0 1 0 1 1 | 1 0 0 0 1 | <add 6> |
| 12 | 0 1 1 0 0 | 1 0 0 1 0 | <add 6> |
| 13 | 0 1 1 0 1 | 1 0 0 1 1 | <add 6> |
| 14 | 0 1 1 1 0 | 1 0 1 0 0 | <add 6> |
| 15 | 0 1 1 1 1 | 1 0 1 0 1 | <add 6> |
| 16 | 1 0 0 0 0 | 1 0 1 1 0 | <add 6> |
| 17 | 1 0 0 0 1 | 1 0 1 1 1 | <add 6> |
| 18 | 1 0 0 1 0 | 1 1 0 0 0 | <add 6> |
| 19 | 1 0 0 1 1 | 1 1 0 0 1 | <add 6> |

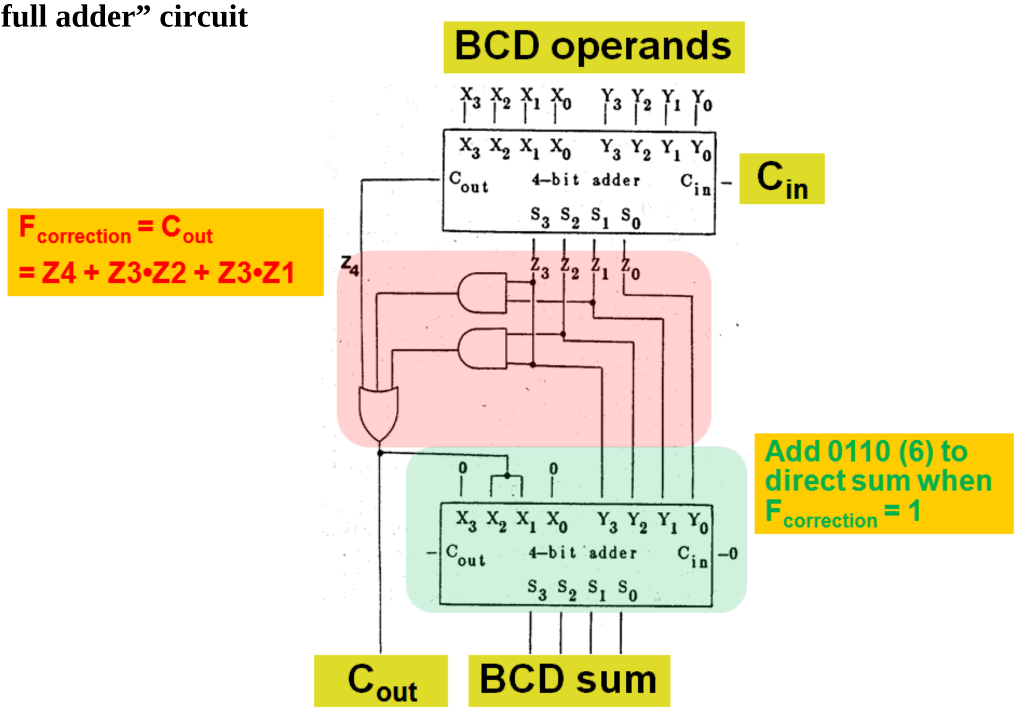
- **examples of decimal addition and correction**
- **summary of rules**
  - o **if the sum of two BCD digits is ≤ 9 (i.e. 1001), no correction is needed**
  - o **if the sum of two BCD digits is > 9, the result must be corrected by adding six (0110)**
- **"correction function" derivation**



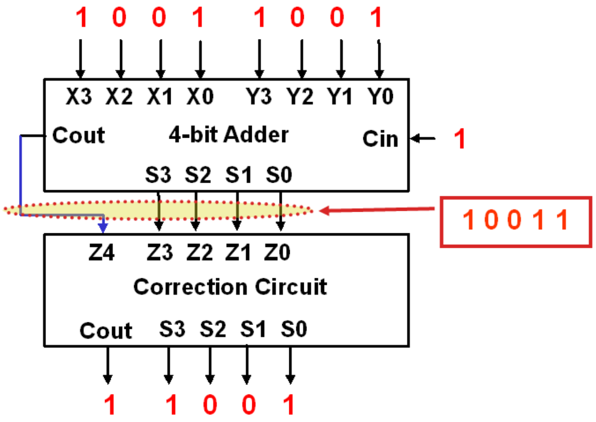
**$F_{correction} = C_{out} =$**

**Z4 + Z3•Z2 + Z3•Z1**

- **BCD “full adder” circuit**

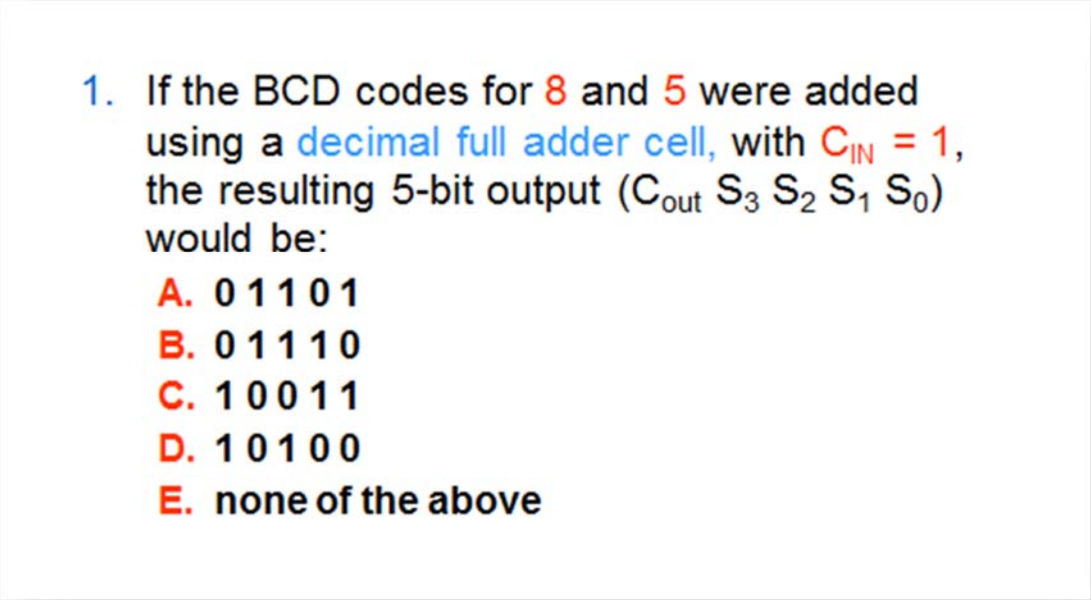


- **example: maximum value that can be generated by a BCD full adder cell (9+9+Cin)**

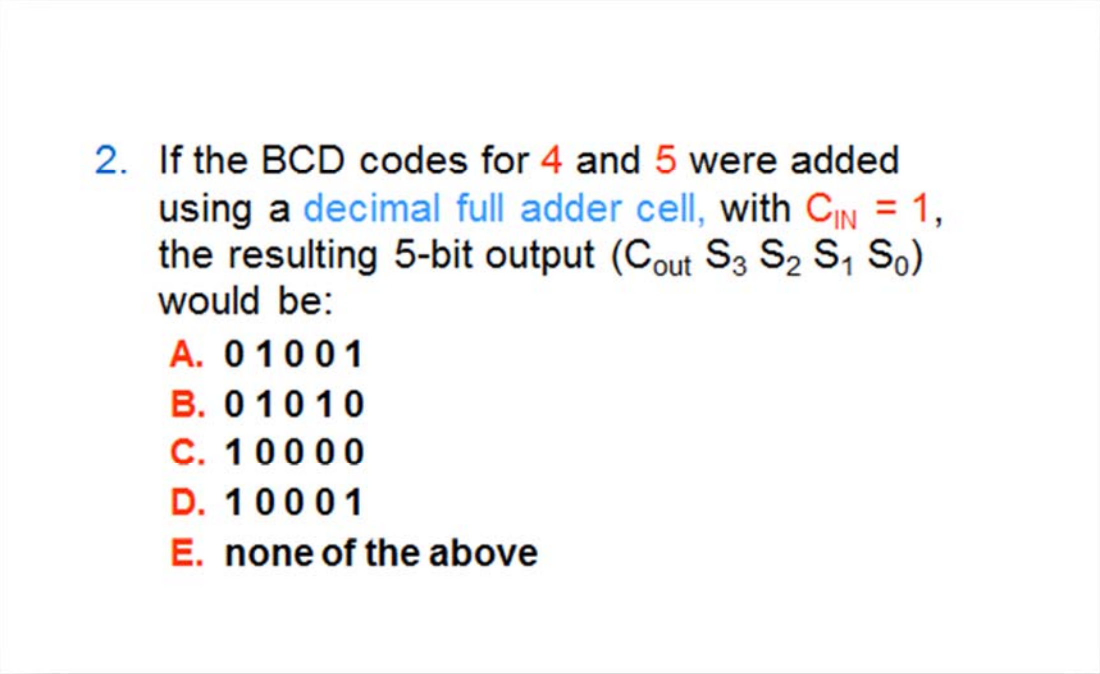


- **example: circuit that produces the diminished radix complement of a BCD digit**

```
module ninescmp(X, Y);
  input wire [3:0] X;      // Input code
  output reg [3:0] Y;      // Output code
  always @ (X) begin
    case (X)
      4'b0000:      Y = 4'b1001;
      4'b0001:      Y = 4'b1000;
      4'b0010:      Y = 4'b0111;
      4'b0011:      Y = 4'b0110;
      4'b0100:      Y = 4'b0101;
      4'b0101:      Y = 4'b0100;
      4'b0110:      Y = 4'b0011;
      4'b0111:      Y = 4'b0010;
      4'b1000:      Y = 4'b0001;
      4'b1001:      Y = 4'b0000;
      default:      Y = 4'b0000; // used for inputs > 9
    endcase
  end
endmodule
```



1. If the BCD codes for 8 and 5 were added using a decimal full adder cell, with $C_{IN}$ = 1, the resulting 5-bit output ($C_{out}$ $S_3$ $S_2$ $S_1$ $S_0$) would be:
   A. 0 1 1 0 1
   B. 0 1 1 1 0
   C. 1 0 0 1 1
   D. 1 0 1 0 0
   E. none of the above



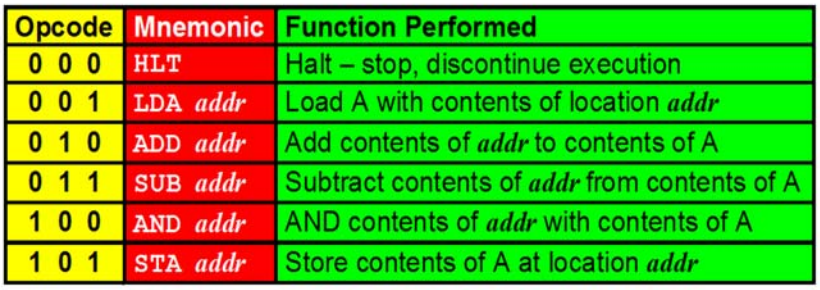
2. If the BCD codes for 4 and 5 were added using a decimal full adder cell, with $C_{IN}$ = 1, the resulting 5-bit output ($C_{out}$ $S_3$ $S_2$ $S_1$ $S_0$) would be:
   A. 0 1 0 0 1
   B. 0 1 0 1 0
   C. 1 0 0 0 0
   D. 1 0 0 0 1
   E. none of the above

# Lecture Summary – Module 4-G

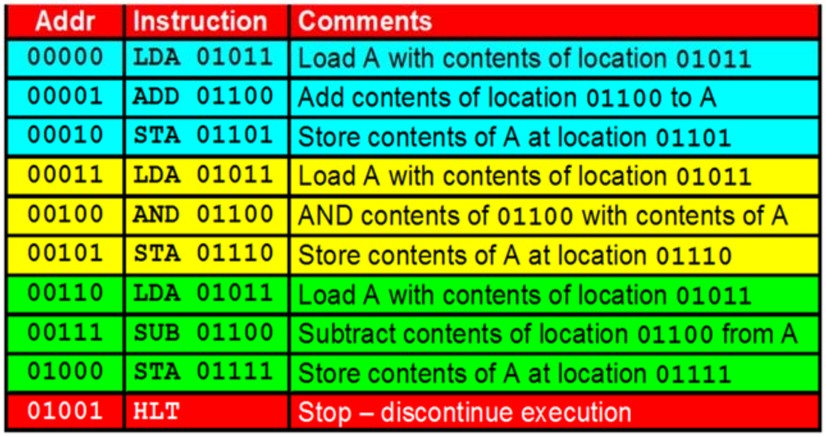
## *Simple Computer – Top-Down Specification*

**Reference:** Meyer Supplemental Text, pp. 1-18

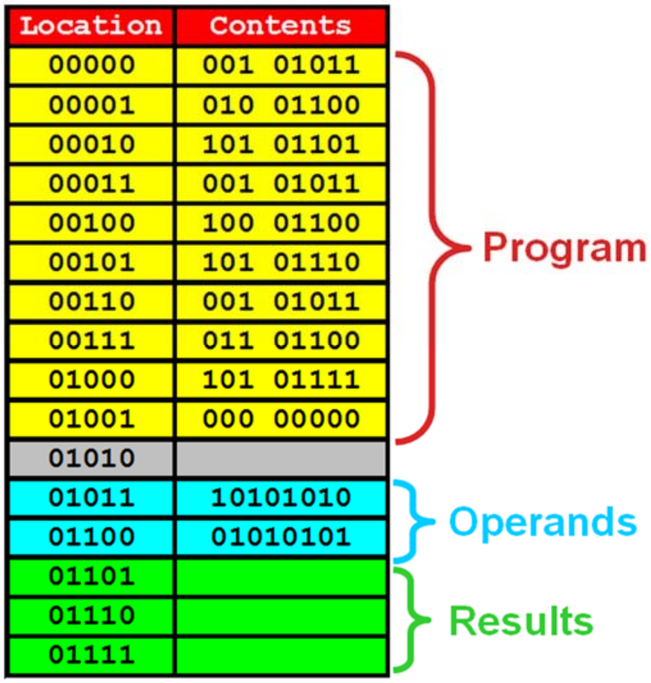
- **overview**
  - o **the "ultimate application" of what we have learned**
  - o **computer defn – sequential execution of stored program**
  - o **architecture defn – arrangement and interconnection of functional blocks**
  - o **house analogy**
- **big picture**
  - o **input/output**
  - o **start (reset)**
  - o **clock**
- **floor plan**
  - o **programming model**
  - o **instruction set**
  - o **registers**
  - o **instruction format**
    - ▪ **opcode**
    - ▪ **address**
  - o **two-address machine**
- **programming example**
- **memory snapshot**



| Opcode | Mnemonic | Function Performed |
|---|---|---|
| 0 0 0 | HLT | Halt – stop, discontinue execution |
| 0 0 1 | LDA *addr* | Load A with contents of location *addr* |
| 0 1 0 | ADD *addr* | Add contents of *addr* to contents of A |
| 0 1 1 | SUB *addr* | Subtract contents of *addr* from contents of A |
| 1 0 0 | AND *addr* | AND contents of *addr* with contents of A |
| 1 0 1 | STA *addr* | Store contents of A at location *addr* |

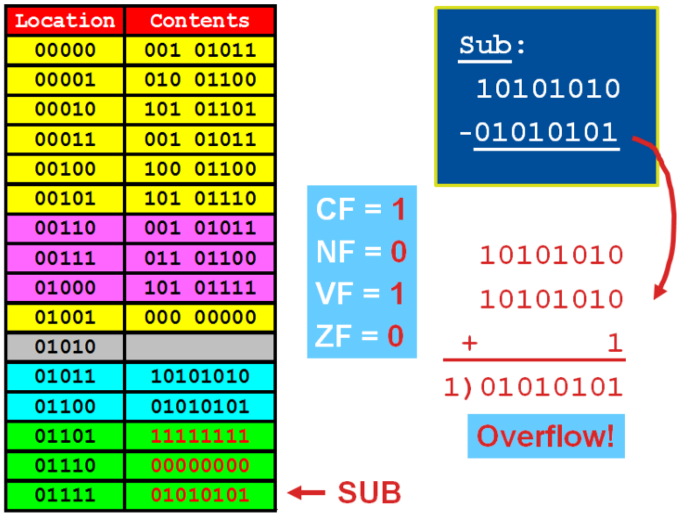


| Addr | Instruction | Comments |
|---|---|---|
| 00000 | LDA 01011 | Load A with contents of location 01011 |
| 00001 | ADD 01100 | Add contents of location 01100 to A |
| 00010 | STA 01101 | Store contents of A at location 01101 |
| 00011 | LDA 01011 | Load A with contents of location 01011 |
| 00100 | AND 01100 | AND contents of 01100 with contents of A |
| 00101 | STA 01110 | Store contents of A at location 01110 |
| 00110 | LDA 01011 | Load A with contents of location 01011 |
| 00111 | SUB 01100 | Subtract contents of location 01100 from A |
| 01000 | STA 01111 | Store contents of A at location 01111 |
| 01001 | HLT | Stop – discontinue execution |



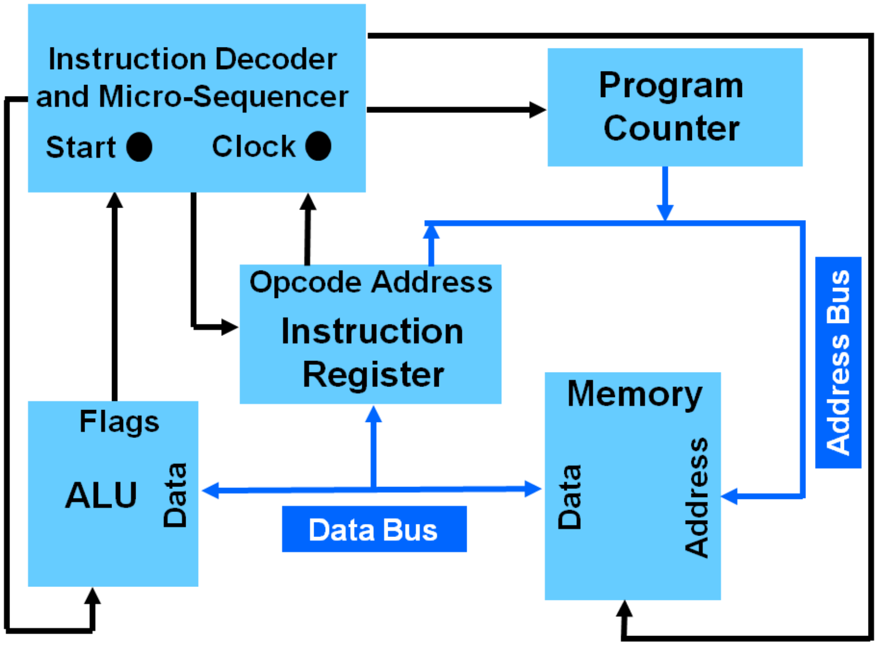
| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |

**Calculation of ADD, AND, and SUB results:**

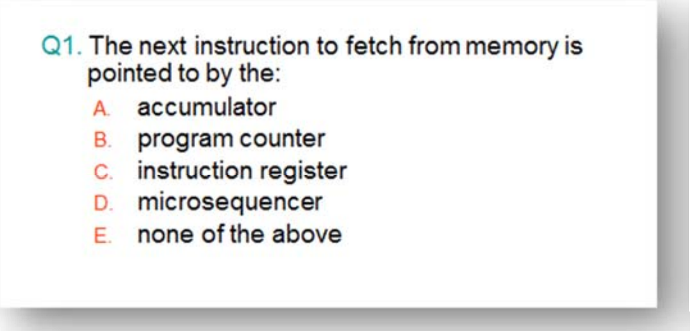


| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | 11111111 |
| 01110 | 00000000 |
| 01111 | 01010101 |

- **block diagram**
  - **memory**
  - **program counter**
  - **instruction register**
  - **arithmetic logic unit**
  - **instruction decoder and micro-sequencer**

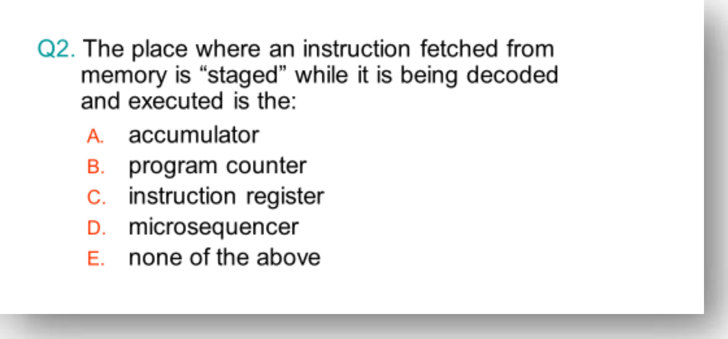


- **notes**
  - **each functional block is "self-contained" (can be *independently tested*)**
  - **can add more instructions by increasing number of opcode bits**
  - **can add more memory by increasing the number of address bits**
  - **can increase numeric range by increasing the number of data bits**



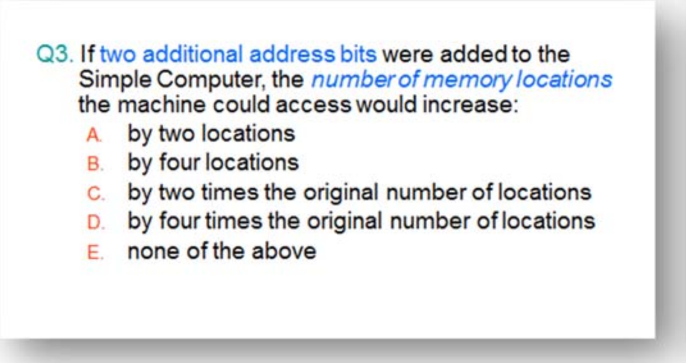
Q1. The next instruction to fetch from memory is pointed to by the:

A. accumulator
B. program counter
C. instruction register
D. microsequencer
E. none of the above



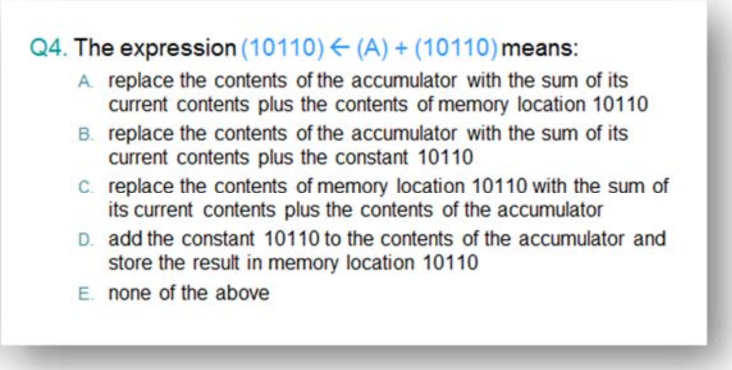
Q2. The place where an instruction fetched from memory is "staged" while it is being decoded and executed is the:

A. accumulator
B. program counter
C. instruction register
D. microsequencer
E. none of the above



Q3. If two additional address bits were added to the Simple Computer, the *number of memory locations* the machine could access would increase:

A. by two locations
B. by four locations
C. by two times the original number of locations
D. by four times the original number of locations
E. none of the above



Q4. The expression (10110) ← (A) + (10110) means:

A. replace the contents of the accumulator with the sum of its current contents plus the contents of memory location 10110
B. replace the contents of the accumulator with the sum of its current contents plus the constant 10110
C. replace the contents of memory location 10110 with the sum of its current contents plus the contents of the accumulator
D. add the constant 10110 to the contents of the accumulator and store the result in memory location 10110
E. none of the above

# Lecture Summary – Module 4-H

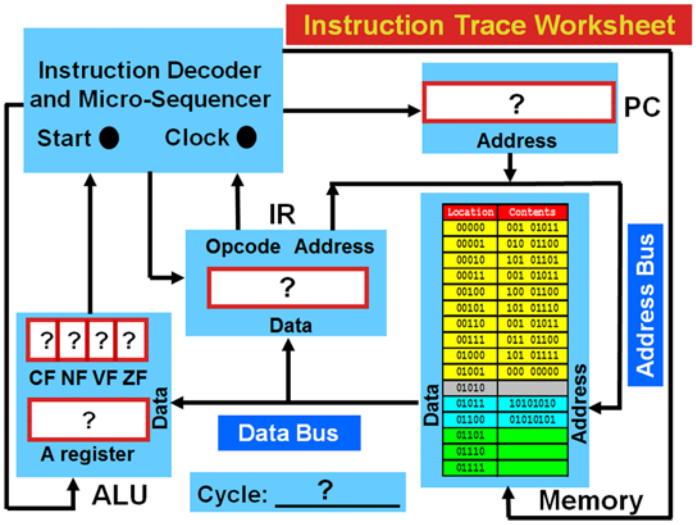
## *Simple Computer – Instruction Tracing*

**Reference:** Meyer Supplemental Text, pp. 18-24

- **overview**
  - o **two basic steps in "processing" an instruction**
    - ▪ **fetch**
    - ▪ **execute**
  - o **will trace the processing of several instructions to better understand this**
- **program segment to trace**

| Addr | Instruction | Comments |
|---|---|---|
| 00000 | LDA 01011 | Load A with contents of location 01011 |
| 00001 | ADD 01100 | Add contents of location 01100 to A |
| 00010 | STA 01101 | Store contents of A at location 01101 |

- **worksheet**

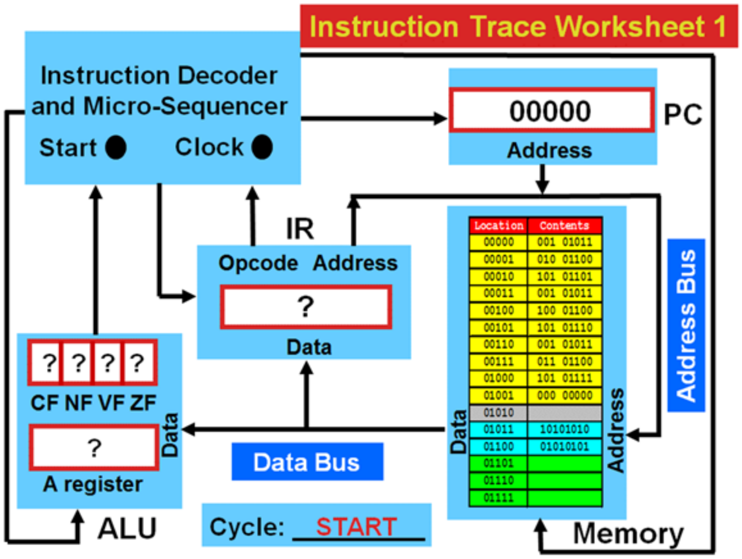


| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |

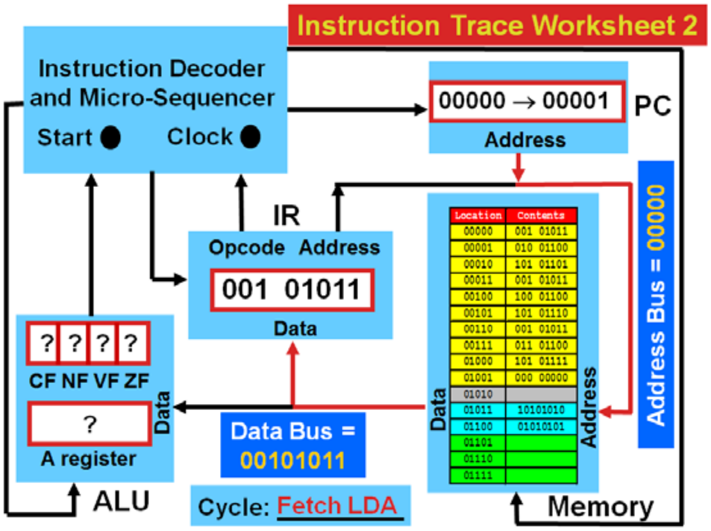
**Notes:**

1. **The clock edges drive the <u>synchronous</u> functions of the computer (e.g., increment program counter)**
2. **The decoded states (here, fetch and execute) enable the <u>combinational</u> functions of the computer (e.g., turn on tri-state buffers)**

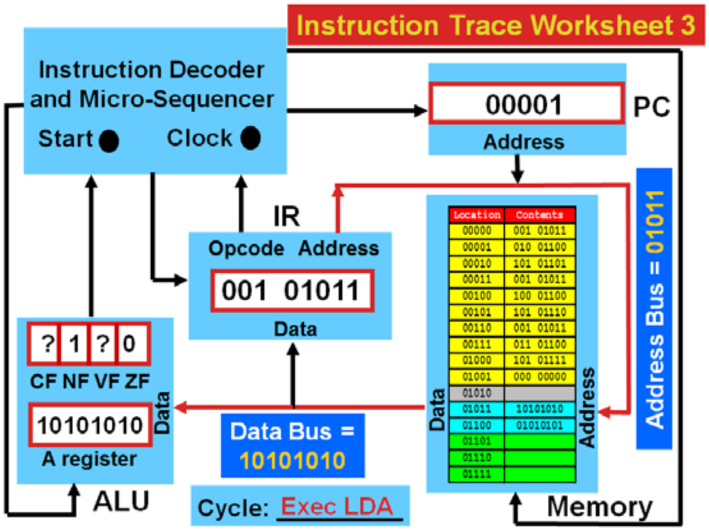
- **step 1 (after START pushbutton pressed)**



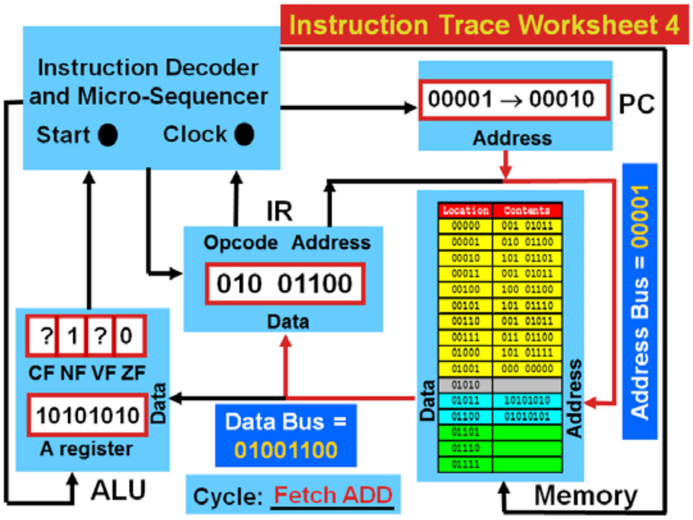
| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |



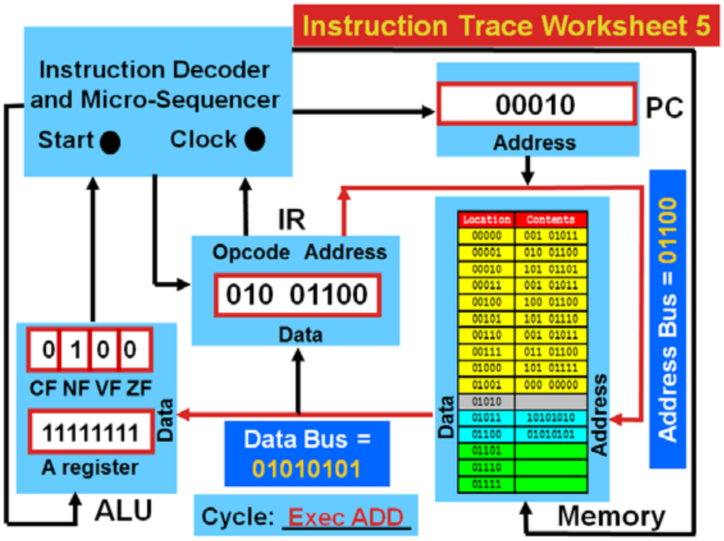
| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |



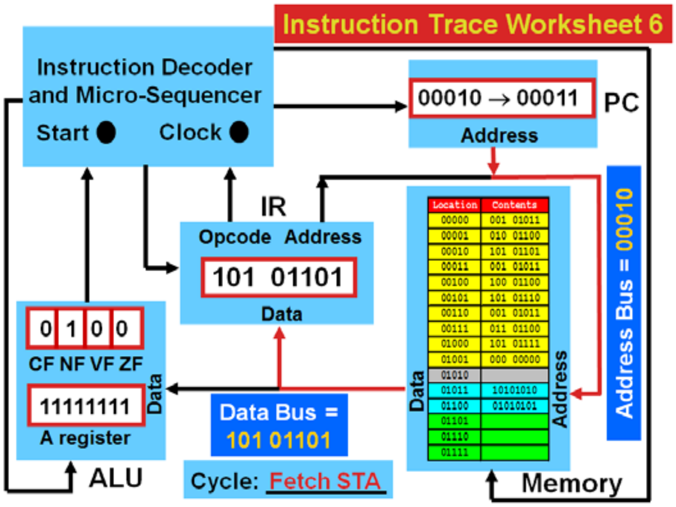
| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |



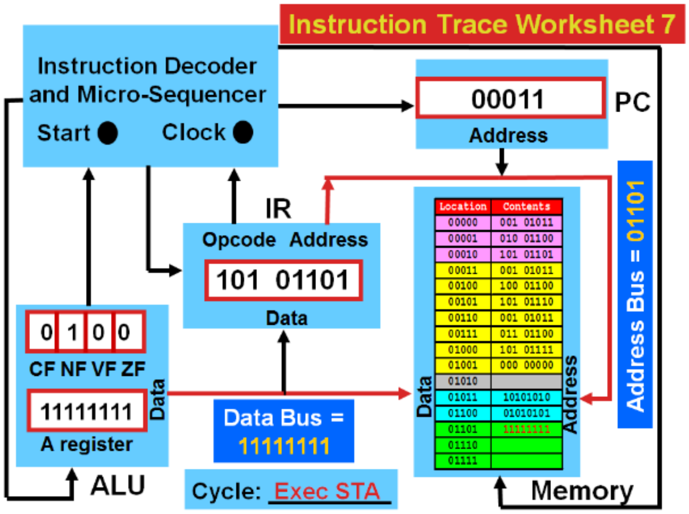
| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |



| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |



| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | |
| 01110 | |
| 01111 | |

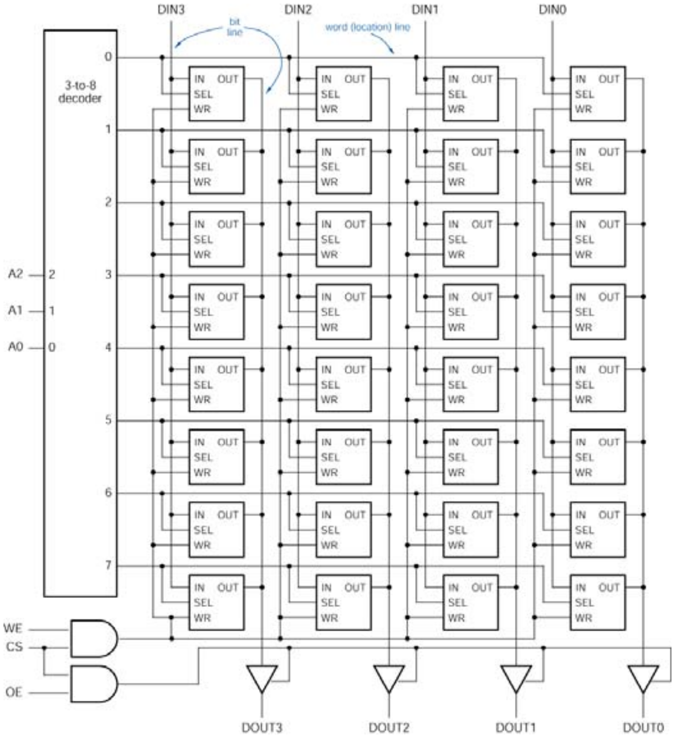


| Location | Contents |
|---|---|
| 00000 | 001 01011 |
| 00001 | 010 01100 |
| 00010 | 101 01101 |
| 00011 | 001 01011 |
| 00100 | 100 01100 |
| 00101 | 101 01110 |
| 00110 | 001 01011 |
| 00111 | 011 01100 |
| 01000 | 101 01111 |
| 01001 | 000 00000 |
| 01010 | |
| 01011 | 10101010 |
| 01100 | 01010101 |
| 01101 | 11111111 |
| 01110 | |
| 01111 | |

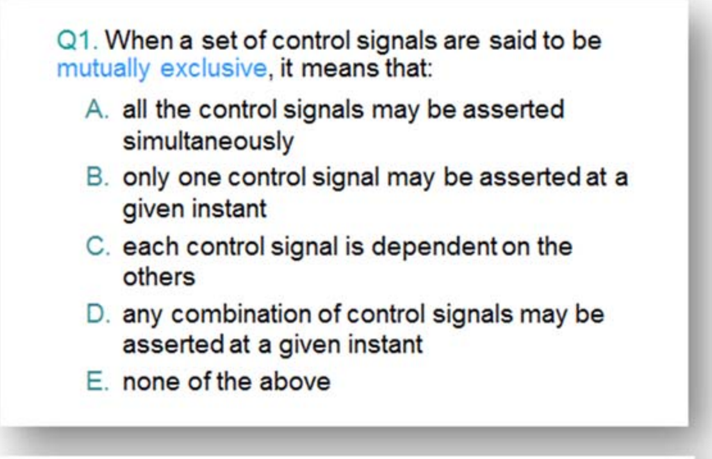
# Lecture Summary – Module 4-I

## *Simple Computer – Bottom-Up Implementation*

**Reference:** Meyer Supplemental Text, pp. 24-42



- **overview**
  - **finished top-down specification of design**
  - **ready for bottom-up implementation**
  - **all system control signals *active high***
  - **some control signals *mutually exclusive***
  - **all blocks use the *same clock signal***
- **memory**
  - **key definitions/terms**
    - **read/write**
    - **"random access" (wrt prop delay)**
    - **static (does not need "refresh")**
    - **volatile (loses data when "off")**
    - **size NxM (here 32x8)**
  - **3 control signals**
    - **MSL – memory select**
    - **MOE – memory output enable**
    - **MWE – memory write enable**
  - **notes**
    - **read operation is combinational**
    - **write operation involves open/closing latch → setup and hold timing matters**



Q1. When a set of control signals are said to be mutually exclusive, it means that:

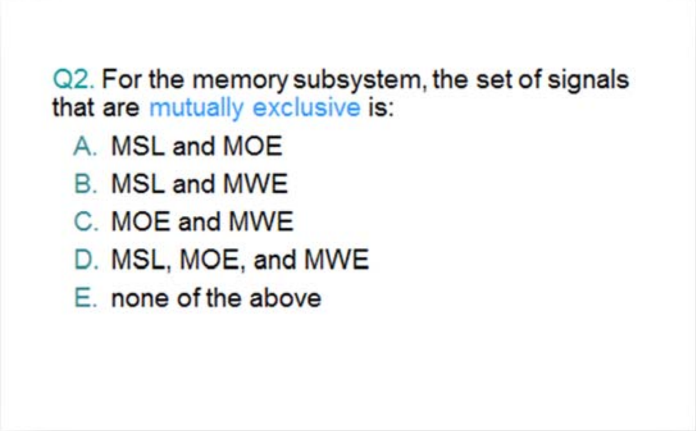
A. all the control signals may be asserted simultaneously

B. only one control signal may be asserted at a given instant

C. each control signal is dependent on the others

D. any combination of control signals may be asserted at a given instant

E. none of the above



Q2. For the memory subsystem, the set of signals that are mutually exclusive is:

A. MSL and MOE

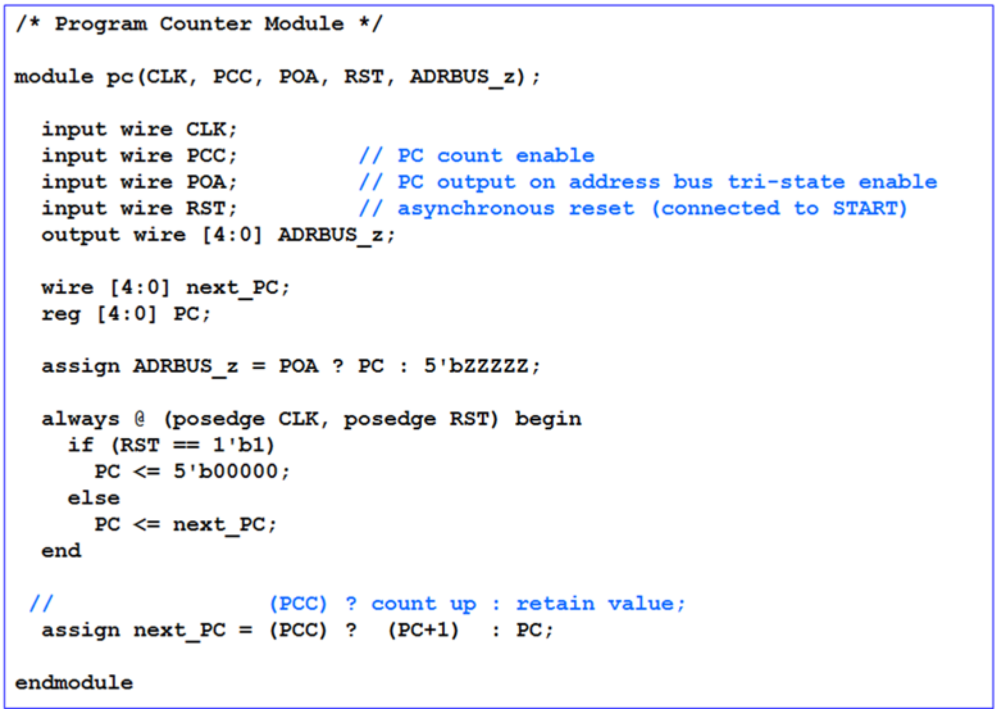
B. MSL and MWE

C. MOE and MWE

D. MSL, MOE, and MWE

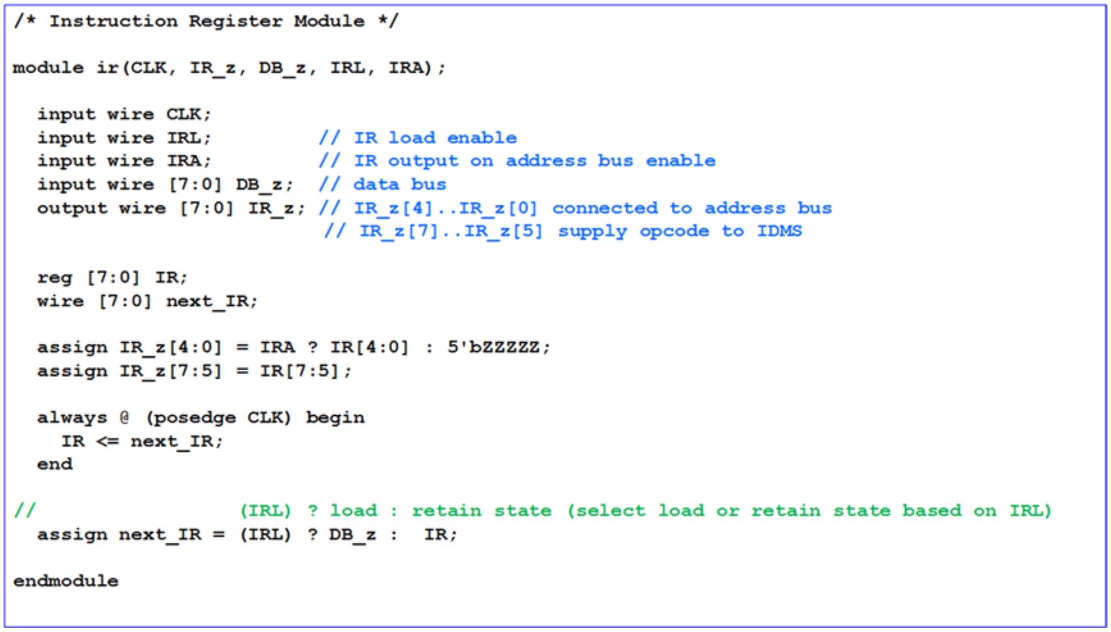
E. none of the above

- **program counter**
    - **basically a binary "up" counter with tri-state outputs and an asynchronous reset**
    - **3 control signals**
        - **ARS – asynchronous reset**
        - **PCC – program counter count enable**
        - **POA – program counter output on address bus tri-state buffer enable**



```
/* Program Counter Module */

module pc(CLK, PCC, POA, RST, ADRBUS_z);

  input wire CLK;
  input wire PCC;         // PC count enable
  input wire POA;         // PC output on address bus tri-state enable
  input wire RST;         // asynchronous reset (connected to START)
  output wire [4:0] ADRBUS_z;

  wire [4:0] next_PC;
  reg [4:0] PC;

  assign ADRBUS_z = POA ? PC : 5'bZZZZZ;

  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      PC <= 5'b00000;
    else
      PC <= next_PC;
  end

 //                (PCC) ? count up : retain value;
  assign next_PC = (PCC) ?  (PC+1)  : PC;

endmodule
```

- **instruction register**
    - **basically an 8-bit data register, with tri-state outputs on the lower 5 (address) bits**
    - **upper 3 bits (opcode) output directly to instruction decoder and micro-sequencer**
    - **two control signals**
        - **IRL – instruction register load enable**
        - **IRA – instruction register address field tri-state output enable**

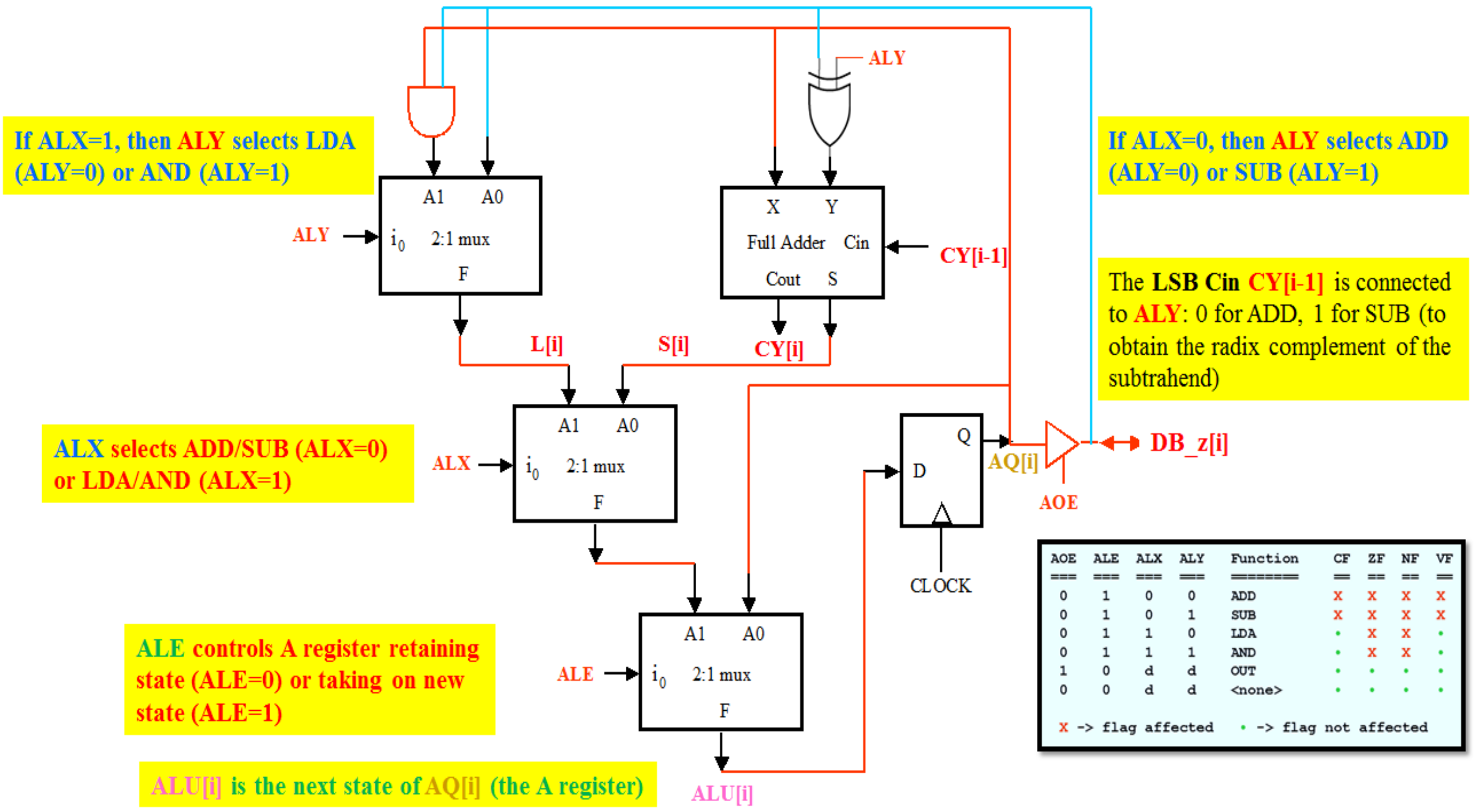


```
/* Instruction Register Module */

module ir(CLK, IR_z, DB_z, IRL, IRA);

  input wire CLK;
  input wire IRL;         // IR load enable
  input wire IRA;         // IR output on address bus enable
  input wire [7:0] DB_z;  // data bus
  output wire [7:0] IR_z; // IR_z[4]..IR_z[0] connected to address bus
                          // IR_z[7]..IR_z[5] supply opcode to IDMS

  reg [7:0] IR;
  wire [7:0] next_IR;

  assign IR_z[4:0] = IRA ? IR[4:0] : 5'bZZZZZ;
  assign IR_z[7:5] = IR[7:5];

  always @ (posedge CLK) begin
    IR <= next_IR;
  end

//                 (IRL) ? load : retain state (select load or retain state based on IRL)
  assign next_IR = (IRL) ? DB_z :  IR;

endmodule
```

- **ALU**
  - o **a multi-function register that performs arithmetic and logical operations**
  - o **four control signals**
    - ▪ **ALE – overall ALU enable**
    - ▪ **ALX – function select**
    - ▪ **ALY – function select**
    - ▪ **AOE – A register tri-state output enable**

```
/* ALU Module */

module alu(CLK, ALE, AOE, ALX, ALY, DB_z, CF, VF, NF, ZF);

/*  8-bit, 4-function ALU with bi-directional data bus
    Accumulator register is AQ, tri-state data bus is DB_z

    ADD:  (AQ[7:0]) <- (AQ[7:0]) + DB_z[7:0]
    SUB:  (AQ[7:0]) <- (AQ[7:0]) - DB_z[7:0]
    LDA:  (AQ[7:0]) <- DB_z[7:0]
    AND:  (AQ[7:0]) <- (AQ[7:0]) & DB_z[7:0]
    OUT:  Value in AQ[7:0] output on data bus DB_z[7:0]

    AOE  ALE  ALX  ALY   Function     CF  ZF  NF  VF
    ===  ===  ===  ===   ========     ==  ==  ==  ==
     0    1    0    0    ADD          x   x   x   x
     0    1    0    1    SUB          x   x   x   x
     0    1    1    0    LDA          •   x   x   •
     0    1    1    1    AND          •   x   x   •
     1    0    d    d    OUT          •   •   •   •
     0    0    d    d    <none>       •   •   •   •

     x -> flag affected    • -> flag not affected

     Note: If ALE = 0, the state of all register bits should be retained */
```

  - o **block diagram of one bit**



If ALX=1, then ALY selects LDA (ALY=0) or AND (ALY=1)

If ALX=0, then ALY selects ADD (ALY=0) or SUB (ALY=1)

The LSB Cin CY[i-1] is connected to ALY: 0 for ADD, 1 for SUB (to obtain the radix complement of the subtrahend)

ALX selects ADD/SUB (ALX=0) or LDA/AND (ALX=1)

ALE controls A register retaining state (ALE=0) or taking on new state (ALE=1)

ALU[i] is the next state of AQ[i] (the A register)

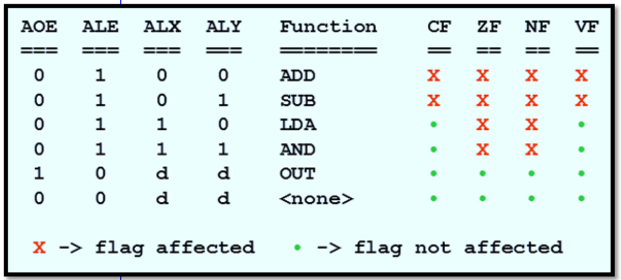
| AOE | ALE | ALX | ALY | Function | CF | ZF | NF | VF |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | ADD | x | x | x | x |
| 0 | 1 | 0 | 1 | SUB | x | x | x | x |
| 0 | 1 | 1 | 0 | LDA | • | x | x | • |
| 0 | 1 | 1 | 1 | AND | • | x | x | • |
| 1 | 0 | d | d | OUT | • | • | • | • |
| 0 | 0 | d | d | <none> | • | • | • | • |

x -> flag affected • -> flag not affected

```
 input wire CLK;

 // ALU control lines
 input wire ALE;              // overall ALU enable
 input wire AOE;              // data bus tri-state output enable
 input wire ALX, ALY;         // function select

 inout wire [7:0] DB_z;       // bidirectional 8-bit tri-state data bus

 output reg CF, VF, NF, ZF;   // condition code register bits (flags)
                              // Carry, Overflow, Negative, Zero
 // Carry equations
 wire [7:0] CY;

 // Combinational ALU outputs
 wire [7:0] ALU;

 wire [7:0] S;                 // Adder/subtractor sum
 wire [7:0] L;                 // LDA/AND multiplexer output
 reg [7:0] AQ;                 // A register flip-flops

 // Next state variables
 reg next_CF, next_VF, next_NF, next_ZF;
 reg [7:0] next_AQ;
```

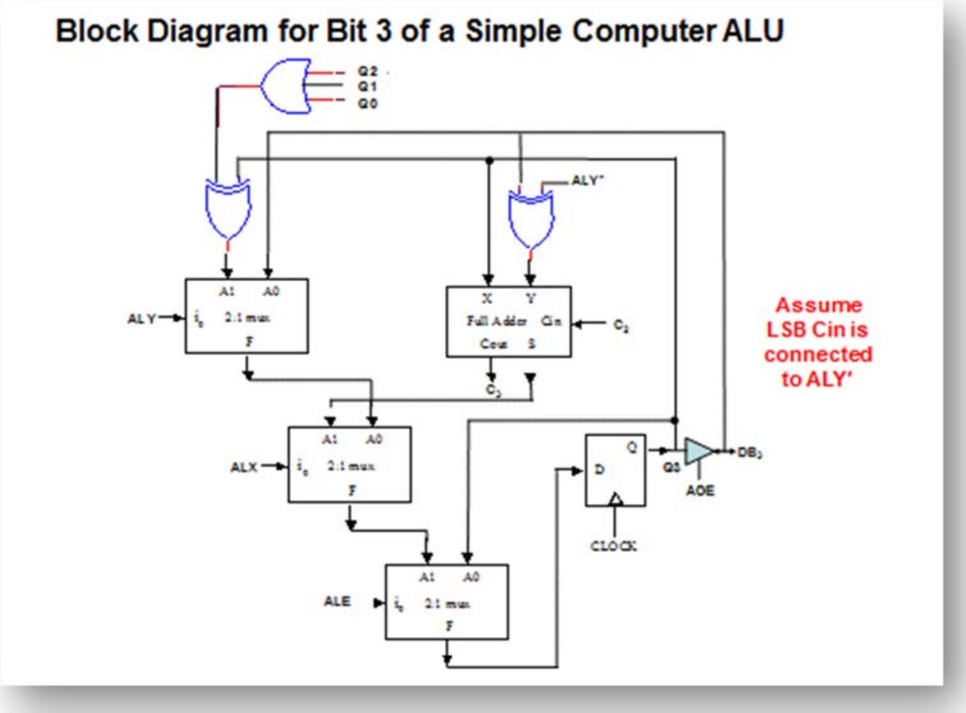
```
// Declaration of intermediate equations
// Least significant bit carry in (0 for ADD, 1 for SUB => ALY)
assign CIN = ALY;

// Intermediate equations for adder/subtractor SUM (S) selected when ALX = 0
assign S = AQ ^ (DB_z ^ ALY) ^ {CY[6:0],CIN};

// Ripple carry equations (CY[7] is COUT, DB_z is data from data bus)
assign CY = AQ & (ALY ^ DB_z) | AQ & {CY[6:0],CIN} | ALY & DB_z & {CY[6:0],CIN};

// Intermediate equations for LOAD and AND, selected when ALX = 1
//        (ALY)?    AND    :  LDA    (select LDA or AND based on ALY)
assign L = ALY ? AQ & DB_z : DB_z ;

// Combinational ALU outputs
//          (ALX)? L : S     (select LDA/AND or ADD/SUB based on ALX)
assign ALU = ALX ? L : S;
```

```
 // Register bit and data bus control equations
 always @(posedge CLK) begin
   AQ <= next_AQ;
 end

 always @ (AQ, ALE, ALU) begin
   next_AQ = ALE ? ALU : AQ;
 end

 assign DB_z = AOE ? AQ : 8'bZZZZZZZZ;

 // Condition code register state equations
 always @ (posedge CLK) begin
   CF <= next_CF;
   ZF <= next_ZF;
   NF <= next_NF;
   VF <= next_VF;
 end

 always @ (CF, NF, ZF, VF, ALE, ALX, ALY, CY) begin
   next_CF = ALE ? (ALX ? CF : (CY[7])          : CF;
   next_ZF = ALE ? (ALU == 0)                   : ZF;
   next_NF = ALE ? ALU[7]                       : NF;
   next_VF = ALE ? (ALX ? VF : (CY[7] ^ CY[6])) : VF;
 end
endmodule
```



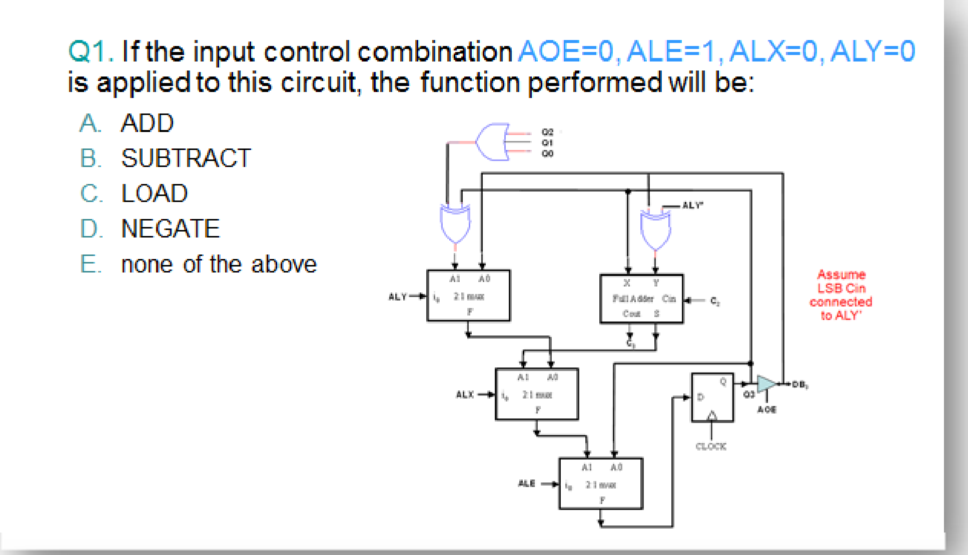
| AOE | ALE | ALX | ALY | Function | CF | ZF | NF | VF |
|---|---|---|---|---|---|---|---|---|
| === | === | === | === | ======= | == | == | == | == |
| 0 | 1 | 0 | 0 | ADD | x | x | x | x |
| 0 | 1 | 0 | 1 | SUB | x | x | x | x |
| 0 | 1 | 1 | 0 | LDA | • | x | x | • |
| 0 | 1 | 1 | 1 | AND | • | x | x | • |
| 1 | 0 | d | d | OUT | • | • | • | • |
| 0 | 0 | d | d | <none> | • | • | • | • |

x -> flag affected • -> flag not affected



## Block Diagram for Bit 3 of a Simple Computer ALU

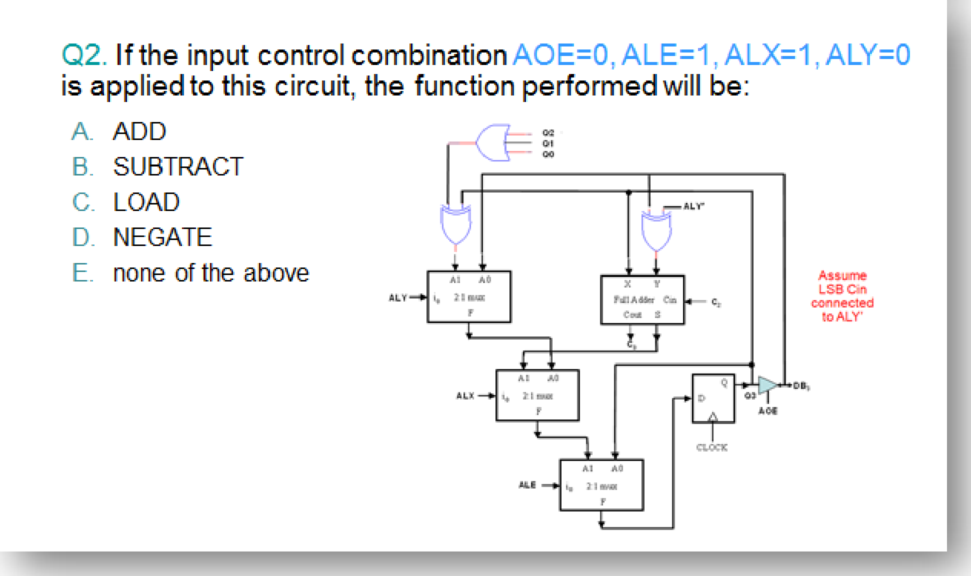
Assume LSB Cin is connected to ALY'



Q1. If the input control combination AOE=0, ALE=1, ALX=0, ALY=0 is applied to this circuit, the function performed will be:

A. ADD
B. SUBTRACT
C. LOAD
D. NEGATE
E. none of the above

Assume LSB Cin connected to ALY'



Q2. If the input control combination AOE=0, ALE=1, ALX=1, ALY=0 is applied to this circuit, the function performed will be:

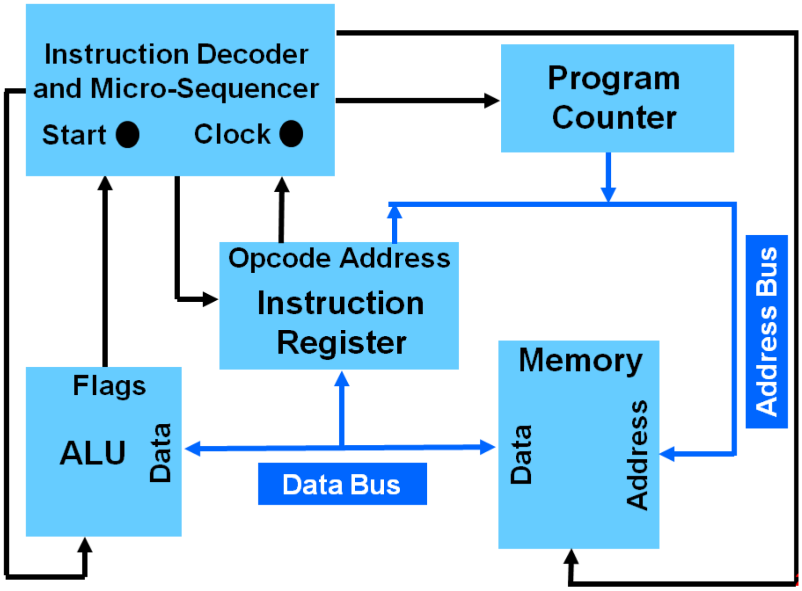
A. ADD
B. SUBTRACT
C. LOAD
D. NEGATE
E. none of the above

Assume LSB Cin connected to ALY'

- **instruction decoder and microsequencer**
  - **state machine that tells all the other state machines what to do ("orchestra director")**
  - **micro-sequence consists of two steps (states)**
    - **fetching instruction from memory**
    - **executing instruction**
    - **fetch/execute state represented by single flip-flop (SQ)**
  - **fetch cycle**
    - **POA (output location of instruction on address bus)**
    - **MSL (select memory, i.e., enable memory to participate)**
    - **MOE (turn on memory tri-state buffers, so that selected location can be read)**
    - **IRL (enable IR to load instruction fetched from memory)**
    - **PCC (enable PC to increment)**
  - **execute cycle – ALU functions (ADD, SUB, LDA, AND)**
    - **IRA (output operand location on address bus)**
    - **MSL (select memory)**
    - **MOE (enable memory to be read)**
    - **ALE (enable ALU to perform the selected function)**
  - **execute cycle – STA instruction**
    - **IRA (output location at which to store result)**
    - **MSL (select memory)**
    - **MWE (enable write to memory)**
    - **AOE (output data in A register via data bus to memory)**

**The synchronous fetch functions (IRL and PCC) will take place on the clock edge that causes the state counter to transition from the fetch state to the execute state**

  - **to stop execution ("halt"), need a "run/stop" flip-flop**
    - **when START pressed, asynchronously set RUN flip-flip**
    - **when HLT instruction executed, asynchronously clear RUN flip-flop**
    - **AND the RUN signal with each synchronous enable signal → effectively disables all functional blocks**



| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H |  | H | H | H |  |  |  |  |  |
| S1 | HLT (000) | L |  |  | L |  | L |  |  | L |  |  |
| S1 | LDA (001) | H | H |  |  |  |  | H |  | H | H |  |
| S1 | ADD (010) | H | H |  |  |  |  | H |  | H |  |  |
| S1 | SUB (011) | H | H |  |  |  |  | H |  | H |  | H |
| S1 | AND (100) | H | H |  |  |  |  | H |  | H | H | H |
| S1 | STA (101) | H |  | H |  |  |  | H | H |  |  |  |

```
/* Instruction Decoder and Microsequencer */

module idms(CLK, START, OP, MSL, MOE, MWE, PCC, POA, ARS, IRL, IRA, ALE, ALX, ALY, AOE);

  input wire CLK;
  input wire START;                // Asynchronous START pushbutton
  input wire [2:0] OP;             // opcode bits (input from IR5..IR7)
  output wire MSL, MOE, MWE;       // Memory control signals
  output wire PCC, POA, ARS;       // PC control signals
  output wire IRL, IRA;            // IR control signals
  output wire ALE, ALX, ALY, AOE;  // ALU control signals (without flags)

  reg SQ, next_SQ;                 // State counter
  reg RUN, next_RUN;               // RUN/HLT state

  wire LDA, STA, ADD, SUB, AND, HLT; // Opcode names
  wire [1:0] S;                    // State variables

  wire RUN_ar;                     // Asynchronous reset for RUN
```

```
  assign HLT = ~OP[2] & ~OP[1] & ~OP[0];  // HLT opcode = 000
  assign LDA = ~OP[2] & ~OP[1] &  OP[0];  // LDA opcode = 001
  assign ADD = ~OP[2] &  OP[1] & ~OP[0];  // ADD opcode = 010
  assign SUB = ~OP[2] &  OP[1] &  OP[0];  // SUB opcode = 011
  assign AND =  OP[2] & ~OP[1] & ~OP[0];  // AND opcode = 100
  assign STA =  OP[2] & ~OP[1] &  OP[0];  // STA opcode = 101

// Decoded state definitions
  assign S[0] = ~SQ;      // fetch
  assign S[1] =  SQ;      // execute

  // State counter
  always @ (posedge CLK, posedge START) begin
    if(START == 1'b1)     // start in fetch state
      SQ <= 1'b0;
    else                  // if RUN negated, resets SQ
      SQ <= next_SQ;
    end

  always @ (SQ, RUN) begin
    next_SQ = RUN & ~SQ;
  end

  // Run/stop
  assign RUN_ar = S[1] & HLT;
  always @ (posedge CLK, posedge RUN_ar, posedge START) begin
    if(START == 1'b1)              // RUN set to 1 when START asserted
      RUN <= 1'b1;
    else if(RUN_ar == 1'b1)        // RUN is cleared when HLT is executed
      RUN <= 1'b0;
  end
```

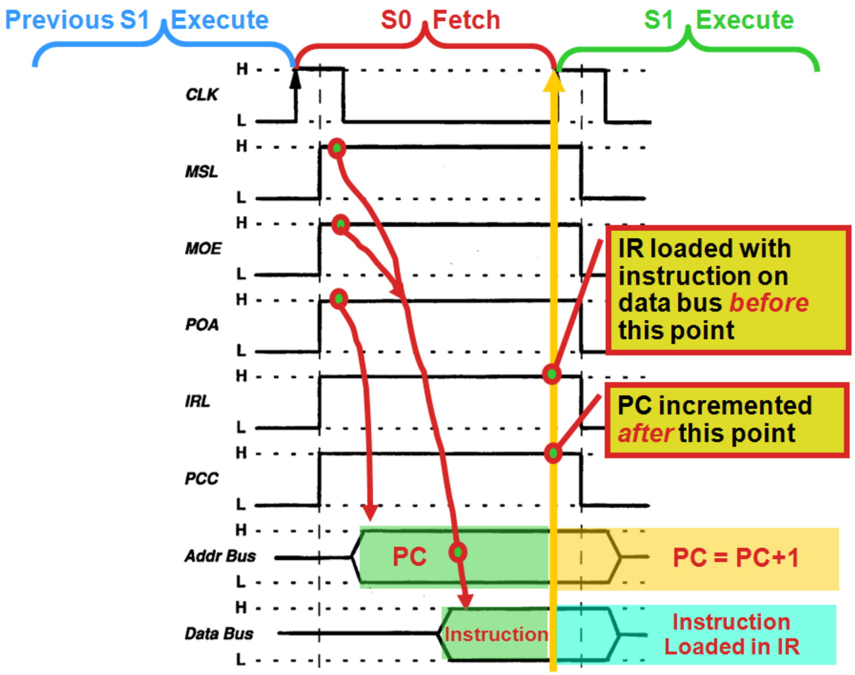
| Opcode | Mnemonic | Function Performed |
|---|---|---|
| 0 0 0 | HLT | Halt – stop, discontinue execution |
| 0 0 1 | LDA *addr* | Load A with contents of location *addr* |
| 0 1 0 | ADD *addr* | Add contents of *addr* to contents of A |
| 0 1 1 | SUB *addr* | Subtract contents of *addr* from contents of A |
| 1 0 0 | AND *addr* | AND contents of *addr* with contents of A |
| 1 0 1 | STA *addr* | Store contents of A at location *addr* |

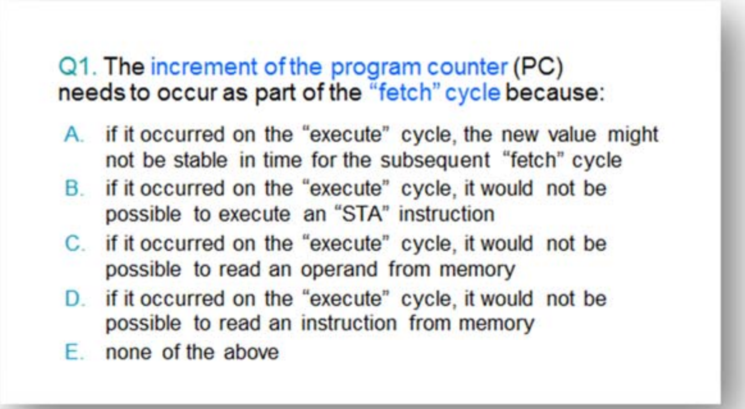
*A D flip-flop synthesized by an always block will retain its value by default unless otherwise specified*

```
  // System control equations
  assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
  assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
  assign MWE = S[1] & STA;
  assign ARS = START;
  assign PCC = RUN & S[0];
  assign POA = S[0];
  assign IRL = RUN & S[0];
  assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
  assign AOE = S[1] & STA;
  assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
  assign ALX = S[1] & (LDA | AND);
  assign ALY = S[1] & (SUB | AND);

endmodule
```

| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H |  | H | H | H |  |  |  |  |  |
| S1 | HLT (000) | L |  |  | L |  | L |  |  | L |  |  |
| S1 | LDA (001) | H | H |  |  |  |  | H |  | H | H |  |
| S1 | ADD (010) | H | H |  |  |  |  | H |  | H |  |  |
| S1 | SUB (011) | H | H |  |  |  |  | H |  | H |  | H |
| S1 | AND (100) | H | H |  |  |  |  | H |  | H | H | H |
| S1 | STA (101) | H |  | H |  |  |  | H | H |  |  |  |

- **system data flow analysis – procedure**
  - **understand operation of functional units**
  - **understand what each instruction does**
  - **identify address & data source/destination**
  - **identify micro-operations required**
  - **identify control signals that need to be asserted**
  - **examine timing relationship**
- **system data flow analysis - constraints**
  - **only one device can drive the bus during a machine cycle**
  - **data cannot pass through more than one flip-flop or latch per cycle**





Q1. The increment of the program counter (PC) needs to occur as part of the "fetch" cycle because:

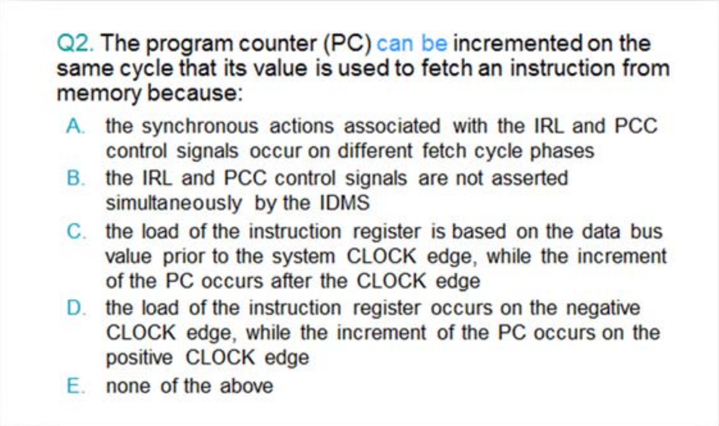
A. if it occurred on the "execute" cycle, the new value might not be stable in time for the subsequent "fetch" cycle

B. if it occurred on the "execute" cycle, it would not be possible to execute an "STA" instruction

C. if it occurred on the "execute" cycle, it would not be possible to read an operand from memory

D. if it occurred on the "execute" cycle, it would not be possible to read an instruction from memory

E. none of the above



Q2. The program counter (PC) can be incremented on the same cycle that its value is used to fetch an instruction from memory because:

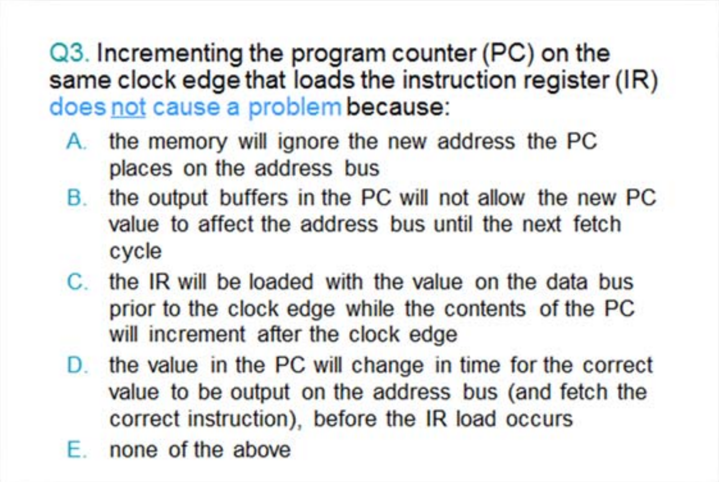
A. the synchronous actions associated with the IRL and PCC control signals occur on different fetch cycle phases

B. the IRL and PCC control signals are not asserted simultaneously by the IDMS

C. the load of the instruction register is based on the data bus value prior to the system CLOCK edge, while the increment of the PC occurs after the CLOCK edge

D. the load of the instruction register occurs on the negative CLOCK edge, while the increment of the PC occurs on the positive CLOCK edge

E. none of the above



Q3. Incrementing the program counter (PC) on the same clock edge that loads the instruction register (IR) does not cause a problem because:

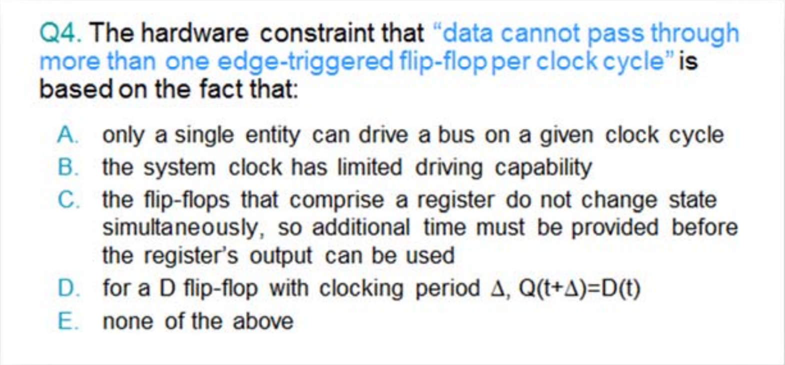
A. the memory will ignore the new address the PC places on the address bus

B. the output buffers in the PC will not allow the new PC value to affect the address bus until the next fetch cycle

C. the IR will be loaded with the value on the data bus prior to the clock edge while the contents of the PC will increment after the clock edge

D. the value in the PC will change in time for the correct value to be output on the address bus (and fetch the correct instruction), before the IR load occurs

E. none of the above



Q4. The hardware constraint that "data cannot pass through more than one edge-triggered flip-flop per clock cycle" is based on the fact that:

A. only a single entity can drive a bus on a given clock cycle

B. the system clock has limited driving capability

C. the flip-flops that comprise a register do not change state simultaneously, so additional time must be provided before the register's output can be used

D. for a D flip-flop with clocking period Δ, Q(t+Δ)=D(t)

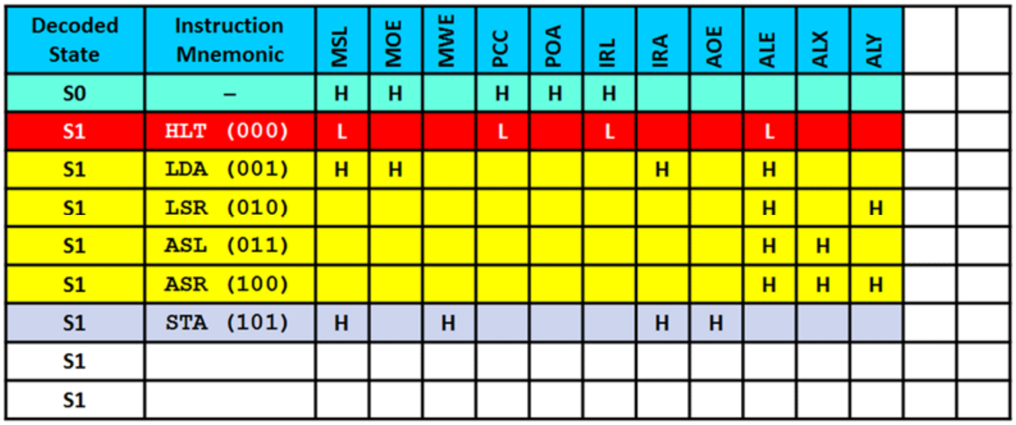
E. none of the above

# Lecture Summary – Module 4-J

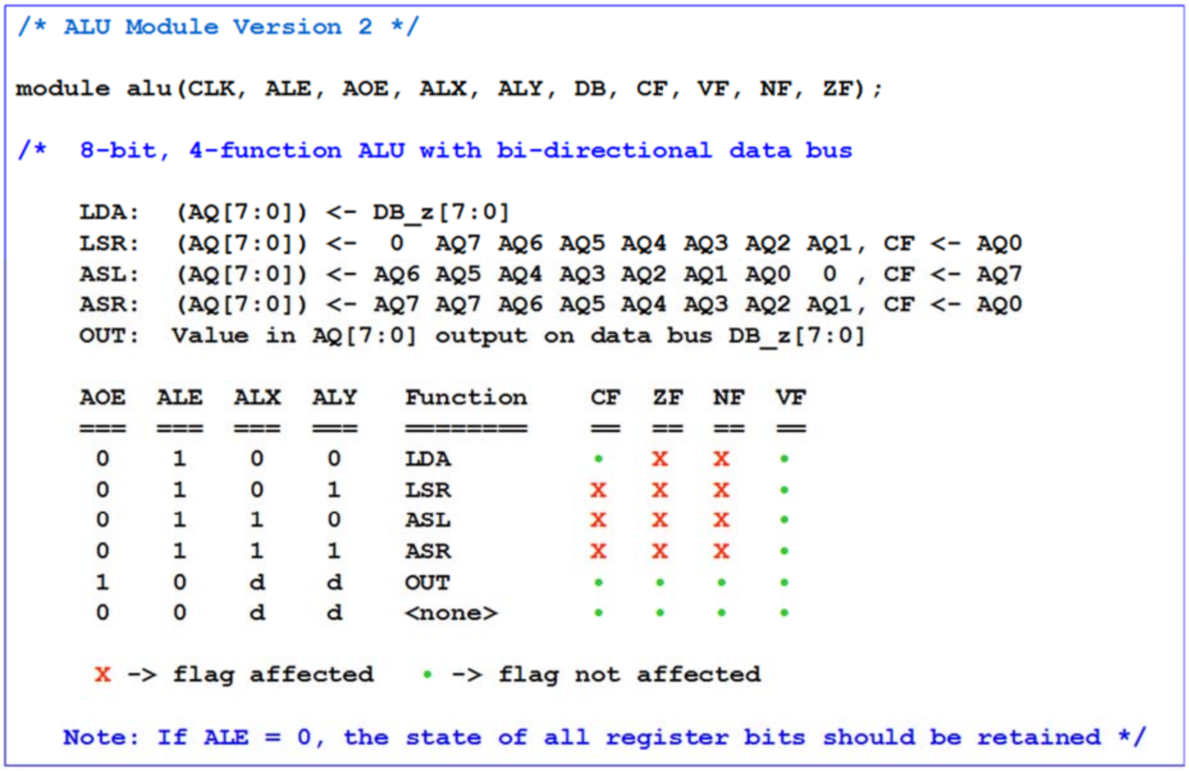
## *Simple Computer – Basic Extensions*

**Reference:** Meyer Supplemental Text, pp. 42-50

- **overview**
  - **will use "spare" opcodes (110 and 111) to add new instructions**
  - **will add rows and columns to original system control table as needed**
- **shift instructions (extension to ALU)**
  - **translation of bits to the left or right**
  - **end off: discard bit shifted out**
  - **preserving: retain bit shifted out**
  - **logical: zero fill (zero shifted in)**
  - **arithmetic: sign preserving**



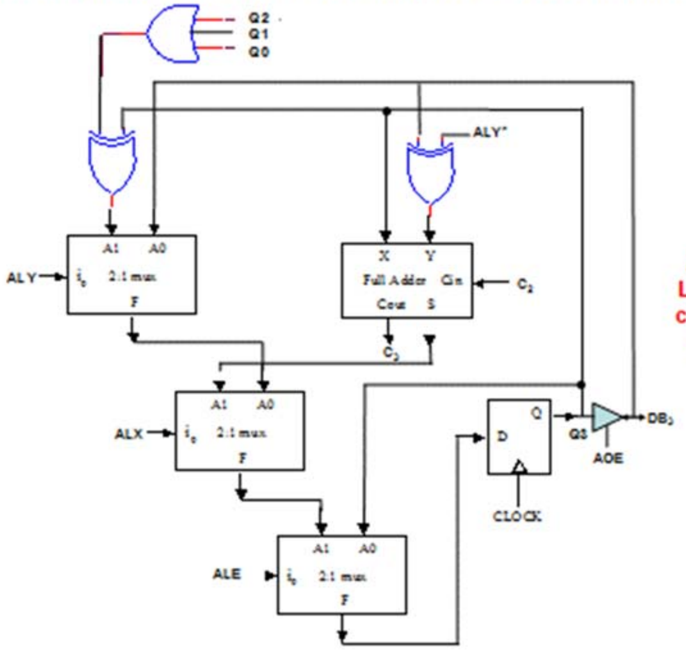
| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H | | H | H | H | | | | | | | |
| S1 | HLT (000) | L | | | L | | L | | | L | | | | |
| S1 | LDA (001) | H | H | | | | | H | | H | | | | |
| S1 | LSR (010) | | | | | | | | | H | | H | | |
| S1 | ASL (011) | | | | | | | | | H | H | | | |
| S1 | ASR (100) | | | | | | | | | H | H | H | | |
| S1 | STA (101) | H | | H | | | | H | H | | | | | |
| S1 | | | | | | | | | | | | | | |
| S1 | | | | | | | | | | | | | | |



```
/* ALU Module Version 2 */

module alu(CLK, ALE, AOE, ALX, ALY, DB, CF, VF, NF, ZF);

/*  8-bit, 4-function ALU with bi-directional data bus

    LDA:  (AQ[7:0]) <- DB_z[7:0]
    LSR:  (AQ[7:0]) <-  0  AQ7 AQ6 AQ5 AQ4 AQ3 AQ2 AQ1, CF <- AQ0
    ASL:  (AQ[7:0]) <- AQ6 AQ5 AQ4 AQ3 AQ2 AQ1 AQ0  0 , CF <- AQ7
    ASR:  (AQ[7:0]) <- AQ7 AQ7 AQ6 AQ5 AQ4 AQ3 AQ2 AQ1, CF <- AQ0
    OUT:  Value in AQ[7:0] output on data bus DB_z[7:0]

    AOE  ALE  ALX  ALY   Function     CF  ZF  NF  VF
    ===  ===  ===  ===   ========     ==  ==  ==  ==
     0    1    0    0    LDA          •   X   X   •
     0    1    0    1    LSR          X   X   X   •
     0    1    1    0    ASL          X   X   X   •
     0    1    1    1    ASR          X   X   X   •
     1    0    d    d    OUT          •   •   •   •
     0    0    d    d    <none>       •   •   •   •

     X -> flag affected    • -> flag not affected

   Note: If ALE = 0, the state of all register bits should be retained */
```

```
 input wire CLK;
  // ALU control lines
  input wire ALE;                // Overall ALU enable
  input wire AOE;                // Data bus tri-state output enable
  input wire ALX, ALY;           // Function select
  inout wire [7:0] DB_z;         // Bidirectional 8-bit data bus
  output reg CF, VF, NF, ZF;     // Condition code bits (flags)
                                 // Carry, Overflow, Negative, Zero
  // Combinational ALU outputs
  wire [7:0] ALU;
  // Accumulator (A) register
  reg [7:0] AQ;
  // Next state variables
  reg next_CF, next_VF, next_NF, next_ZF;
  reg [7:0] next_AQ;

  // Combinational ALU outputs
  always @ (ALX, ALY, DB_z) begin
    case ({ALX,ALY})
      2'b00: ALU = DB_z;              // LDA
      2'b01: ALU = {1'b0,AQ[7:1]};    // LSR
      2'b10: ALU = {AQ[6:0],1'b0};    // ASL
      2'b11: ALU = {AQ[7],AQ[7:1]};   // ASR
    endcase
  end
  // Register bit and data bus control equations
  always @(posedge CLK) begin
    AQ <= next_AQ;
  end
  always @ (ALE, ALU, AQ) begin
    next_AQ = ALE ? ALU : AQ;
  end
  assign DB_z = AOE ? AQ : 8'bZZZZZZZZ;
  // Flag register state equations
  always @ (posedge CLK) begin
    CF <= next_CF;
    ZF <= next_ZF;
    NF <= next_NF;
    VF <= next_VF;
  end
  always @ (ALE, ALX, ALY, CF, ZF, NF, VF, ALU, AQ) begin
    casez ({ALE,ALX,ALY})
      3'b0??: next_CF = 1'b0;
      3'b100: next_CF = CF;// LDA (not affected)
      3'b101: next_CF = AQ[0];   // LSR
      3'b110: next_CF = AQ[7];   // ASL
      3'b111: next_CF = AQ[0];   // ASR
    endcase
    next_ZF = ALE ? (ALU == 0) : ZF;
    next_NF = ALE ? ALU[7] : NF;
    next_VF = VF;                // NOTE: NOT AFFECTED
  end
endmodule
```

## Block Diagram for Bit 3 of a Simple Computer ALU



Assume LSB Cin is connected to ALY'

Q1. If the input control combination AOE=1, ALE=1, ALX=1, ALY=1 is applied to this circuit, the function (inadvertently) performed on (A) will be equivalent to:

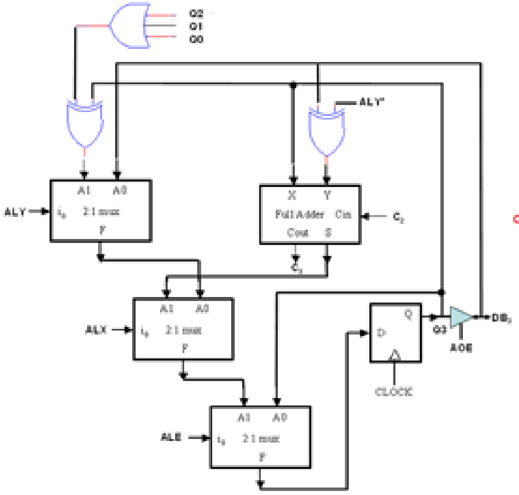
A. logical left shift

B. logical right shift

C. rotate left

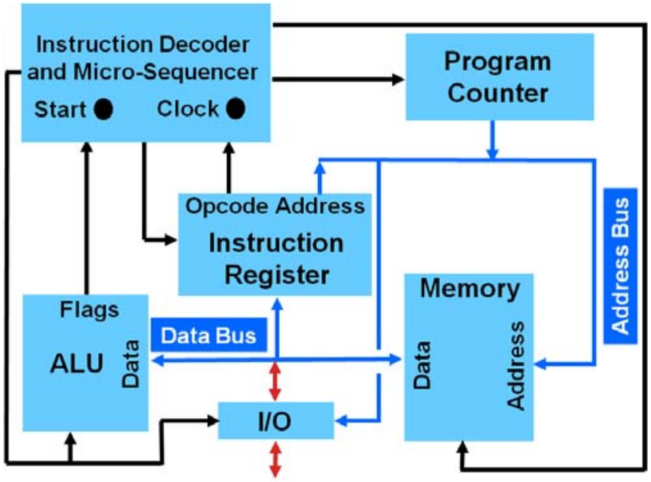
D. rotate right

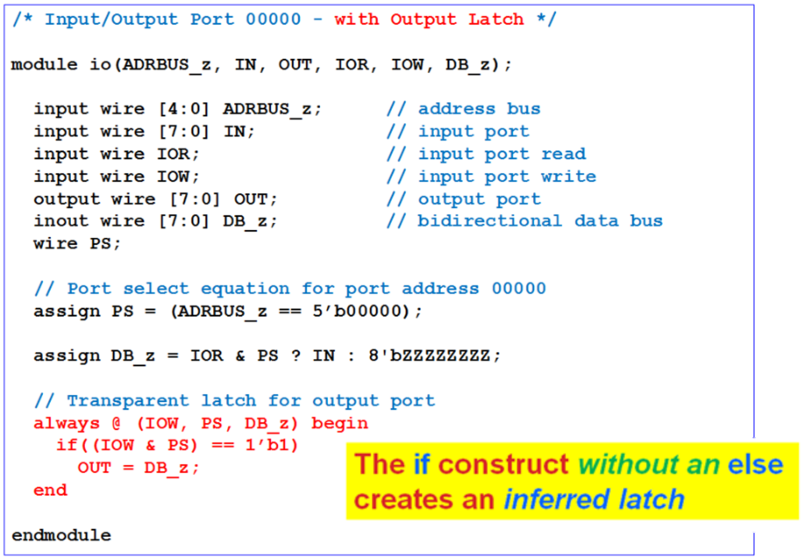
E. none of the above



Assume LSB Cin connected to ALY'

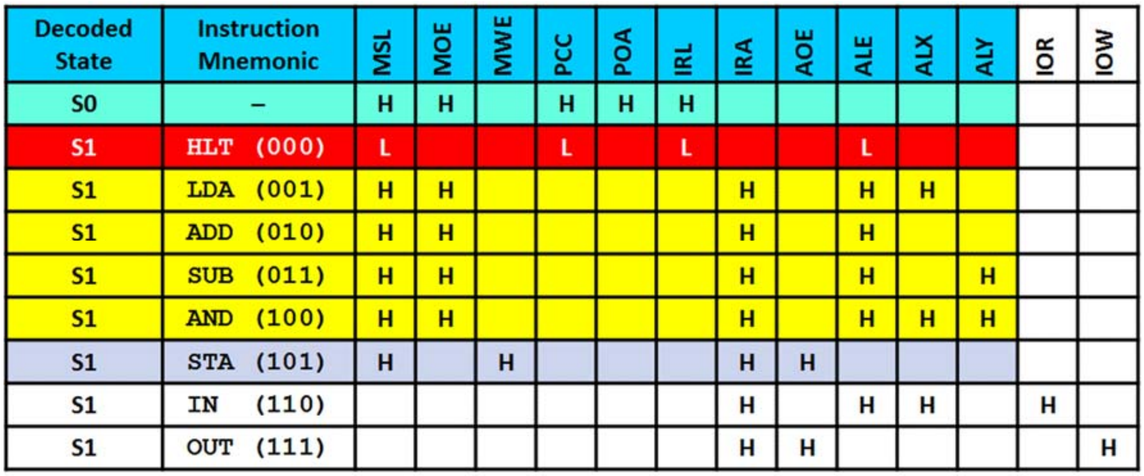
- **input/output (I/O) instructions**
  - **new instructions**
    - **IN *addr* – input data from port *addr* and load into A register**
    - **OUT *addr* – output data in A register to port *addr***
  - **new control signals**
    - **IOR – asserted when IN executed**
    - **IOW – asserted when OUT executed**
  - **modified block diagram, Verilog code for I/O module, modified system control table**



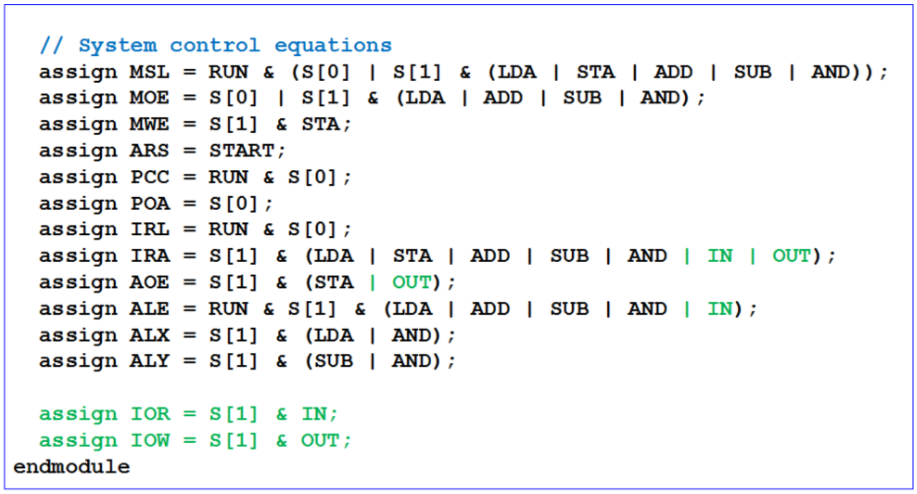


```
/* Input/Output Port 00000 - with Output Latch */

module io(ADRBUS_z, IN, OUT, IOR, IOW, DB_z);

  input wire [4:0] ADRBUS_z;      // address bus
  input wire [7:0] IN;            // input port
  input wire IOR;                 // input port read
  input wire IOW;                 // input port write
  output wire [7:0] OUT;          // output port
  inout wire [7:0] DB_z;          // bidirectional data bus
  wire PS;

  // Port select equation for port address 00000
  assign PS = (ADRBUS_z == 5’b00000);

  assign DB_z = IOR & PS ? IN : 8'bZZZZZZZZ;

  // Transparent latch for output port
  always @ (IOW, PS, DB_z) begin
    if((IOW & PS) == 1’b1)
      OUT = DB_z;
  end

endmodule
```

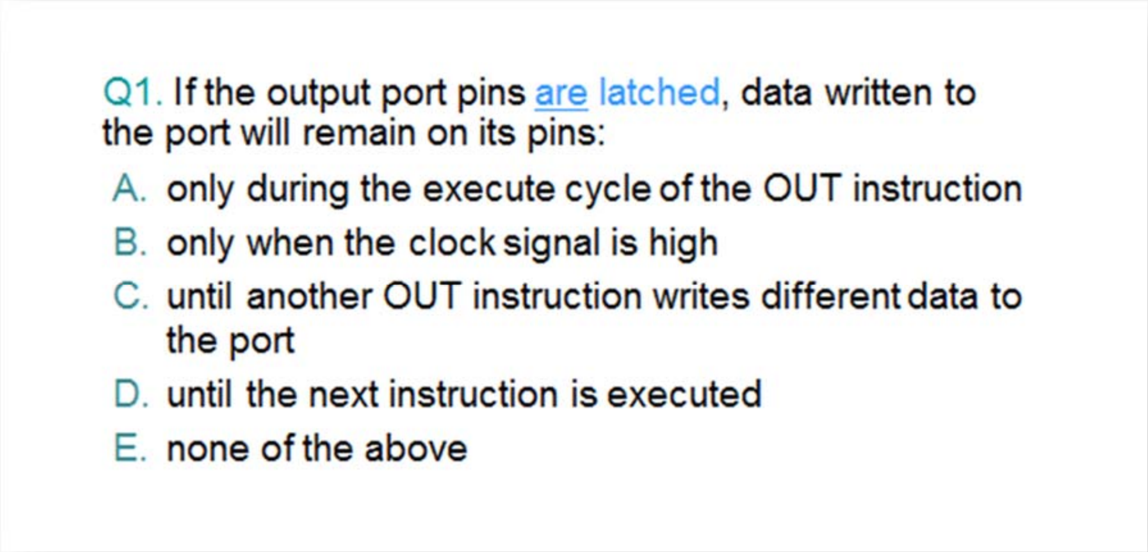
The if construct *without an* else creates an *inferred latch*



| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY | IOR | IOW |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H | | H | H | H | | | | | | | |
| S1 | HLT (000) | L | | | L | | L | | | L | | | | |
| S1 | LDA (001) | H | H | | | | | H | | H | H | | | |
| S1 | ADD (010) | H | H | | | | | H | | H | | | | |
| S1 | SUB (011) | H | H | | | | | H | | H | | H | | |
| S1 | AND (100) | H | H | | | | | H | | H | H | H | | |
| S1 | STA (101) | H | | H | | | | H | H | | | | | |
| S1 | IN (110) | | | | | | | H | | H | H | | H | |
| S1 | OUT (111) | | | | | | | H | H | | | | | H |



```
  // System control equations
  assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
  assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
  assign MWE = S[1] & STA;
  assign ARS = START;
  assign PCC = RUN & S[0];
  assign POA = S[0];
  assign IRL = RUN & S[0];
  assign IRA = S[1] & (LDA | STA | ADD | SUB | AND | IN | OUT);
  assign AOE = S[1] & (STA | OUT);
  assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND | IN);
  assign ALX = S[1] & (LDA | AND);
  assign ALY = S[1] & (SUB | AND);

  assign IOR = S[1] & IN;
  assign IOW = S[1] & OUT;
endmodule
```



Q1. If the output port pins are latched, data written to the port will remain on its pins:

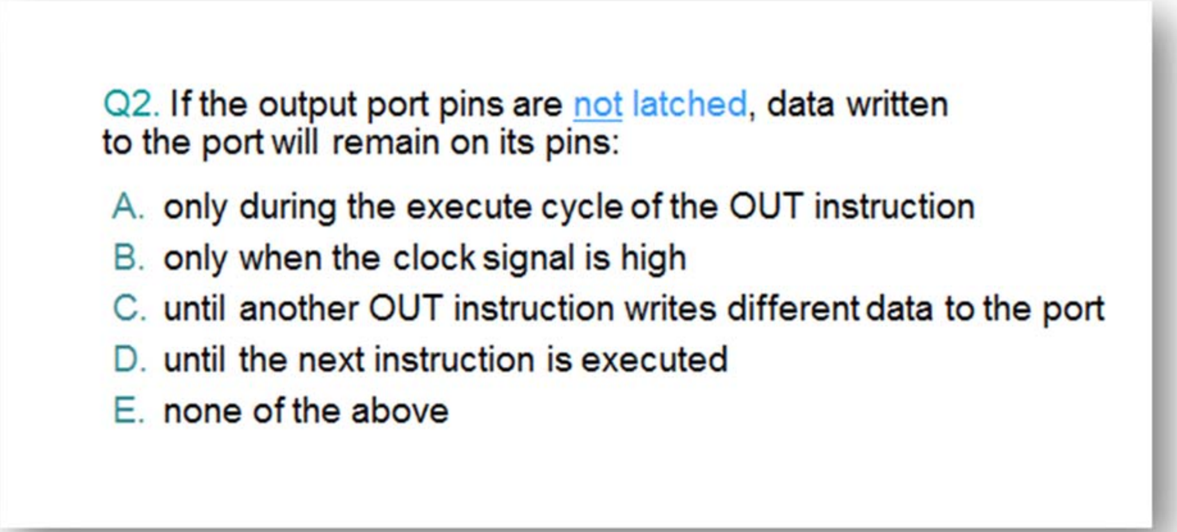
A. only during the execute cycle of the OUT instruction

B. only when the clock signal is high

C. until another OUT instruction writes different data to the port

D. until the next instruction is executed

E. none of the above



Q2. If the output port pins are not latched, data written to the port will remain on its pins:

A. only during the execute cycle of the OUT instruction

B. only when the clock signal is high

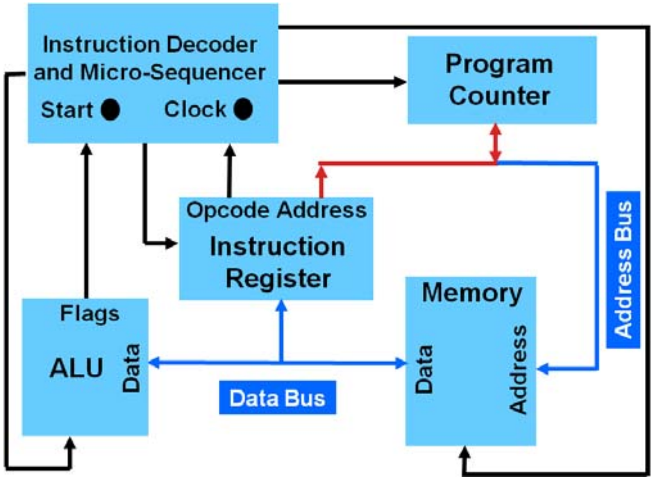
C. until another OUT instruction writes different data to the port

D. until the next instruction is executed

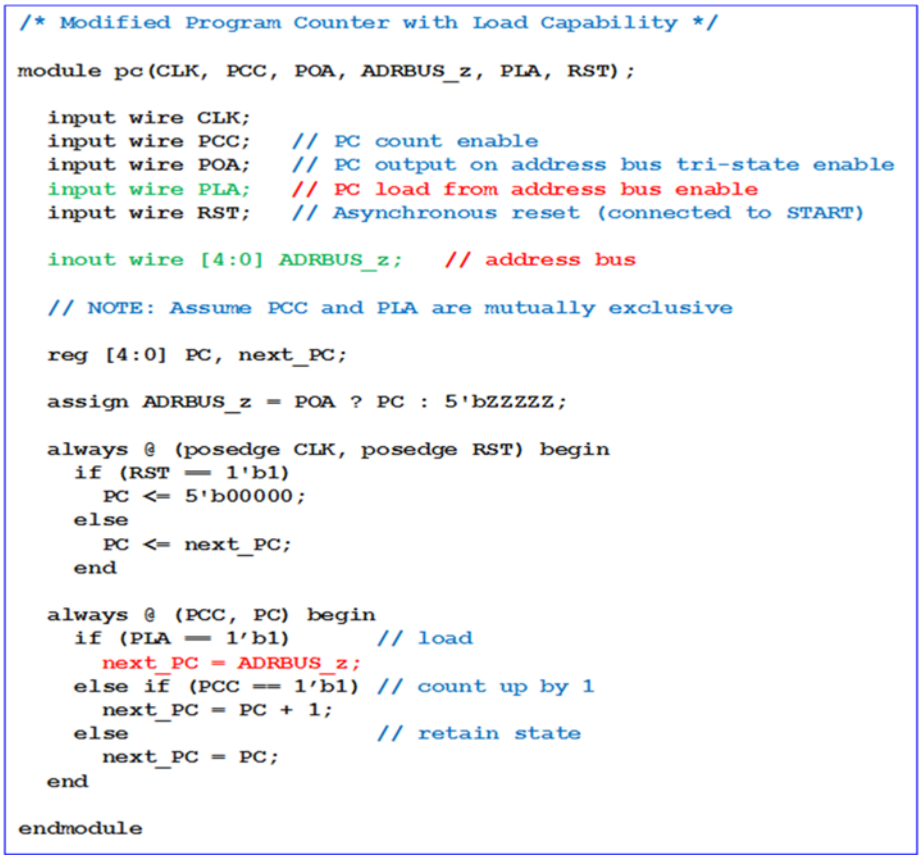
E. none of the above

- **transfer of control instructions**
  - **addressing mode**
    - **absolute – operand field of instruction contains *absolute address* in memory**
    - **relative - operand field contains *signed offset* that should be added to PC**
  - **condition**
    - **unconditional – always happen**
    - **conditional – happen only if specific condition is true (else no-operation)**
  - **illustrative examples**
    - **JMP *addr* – unconditional jump (to absolute address)**
    - **JZF *addr* – jump (to absolute address) *iff* ZF=1 (else no-op)**

  - **modified block diagram**

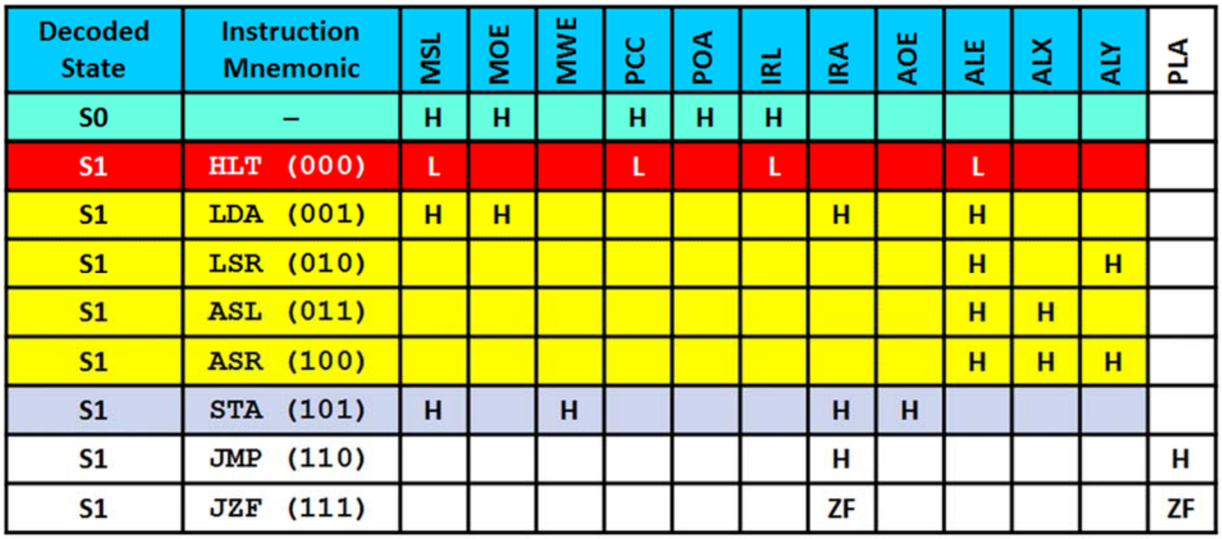


  - **Verilog code for modified PC (with “load from address bus” capability)**

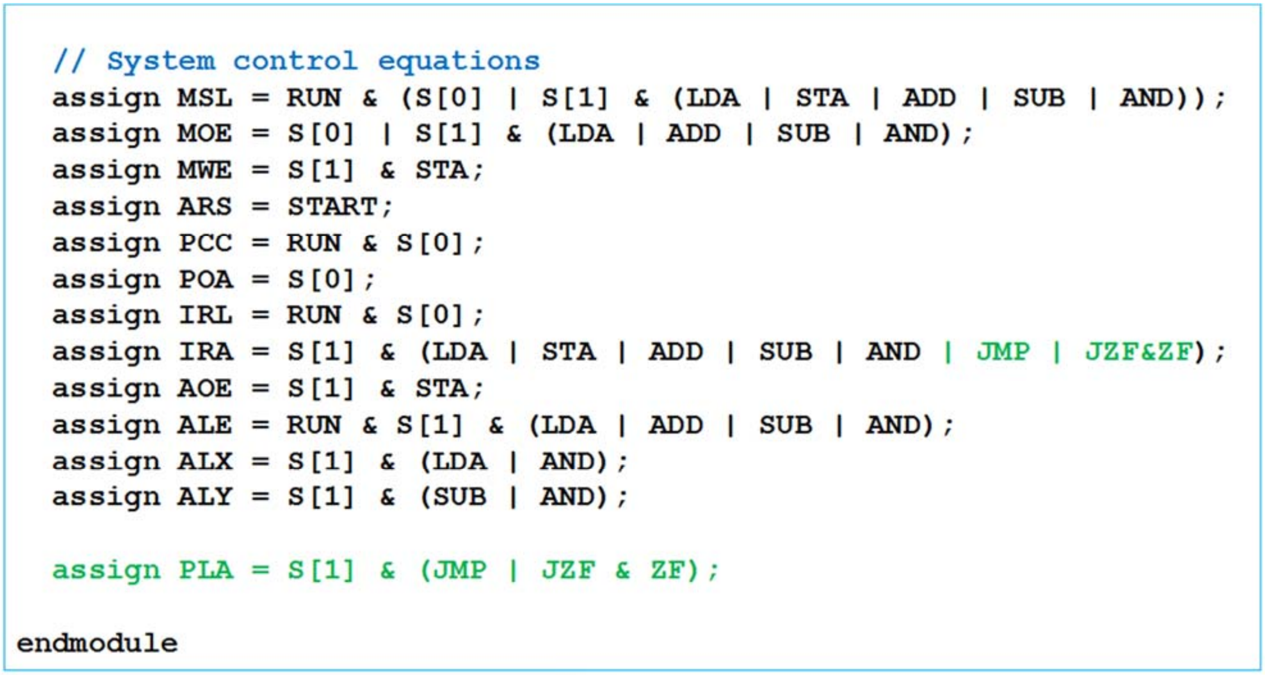


```
/* Modified Program Counter with Load Capability */

module pc(CLK, PCC, POA, ADRBUS_z, PLA, RST);

  input wire CLK;
  input wire PCC;    // PC count enable
  input wire POA;    // PC output on address bus tri-state enable
  input wire PLA;    // PC load from address bus enable
  input wire RST;    // Asynchronous reset (connected to START)

  inout wire [4:0] ADRBUS_z;   // address bus

  // NOTE: Assume PCC and PLA are mutually exclusive

  reg [4:0] PC, next_PC;

  assign ADRBUS_z = POA ? PC : 5'bZZZZZ;

  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      PC <= 5'b00000;
    else
      PC <= next_PC;
    end

  always @ (PCC, PC) begin
    if (PLA == 1’b1)        // load
      next_PC = ADRBUS_z;
    else if (PCC == 1’b1) // count up by 1
      next_PC = PC + 1;
    else                  // retain state
      next_PC = PC;
  end

endmodule
```

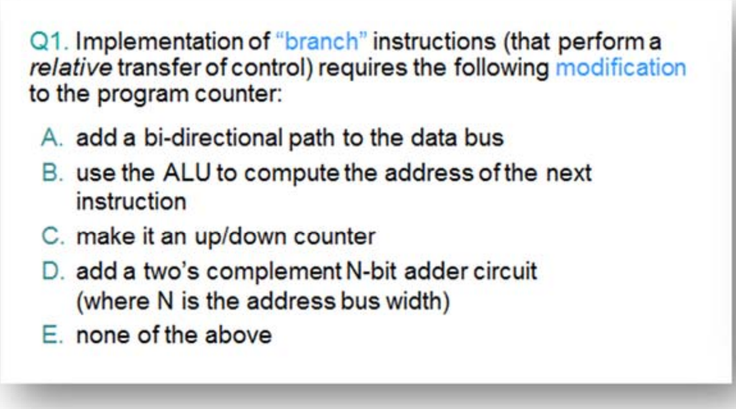
- **modified system control table**



| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY | PLA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H |  | H | H | H |  |  |  |  |  |  |
| S1 | HLT (000) | L |  |  | L |  | L |  |  | L |  |  |  |
| S1 | LDA (001) | H | H |  |  |  |  | H |  | H |  |  |  |
| S1 | LSR (010) |  |  |  |  |  |  |  |  | H |  | H |  |
| S1 | ASL (011) |  |  |  |  |  |  |  |  | H | H |  |  |
| S1 | ASR (100) |  |  |  |  |  |  |  |  | H | H | H |  |
| S1 | STA (101) | H |  | H |  |  |  | H | H |  |  |  |  |
| S1 | JMP (110) |  |  |  |  |  |  | H |  |  |  |  | H |
| S1 | JZF (111) |  |  |  |  |  |  | ZF |  |  |  |  | ZF |



```
   // System control equations
   assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
   assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
   assign MWE = S[1] & STA;
   assign ARS = START;
   assign PCC = RUN & S[0];
   assign POA = S[0];
   assign IRL = RUN & S[0];
   assign IRA = S[1] & (LDA | STA | ADD | SUB | AND | JMP | JZF&ZF);
   assign AOE = S[1] & STA;
   assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
   assign ALX = S[1] & (LDA | AND);
   assign ALY = S[1] & (SUB | AND);

   assign PLA = S[1] & (JMP | JZF & ZF);

endmodule
```



Q1. Implementation of "branch" instructions (that perform a *relative* transfer of control) requires the following modification to the program counter:

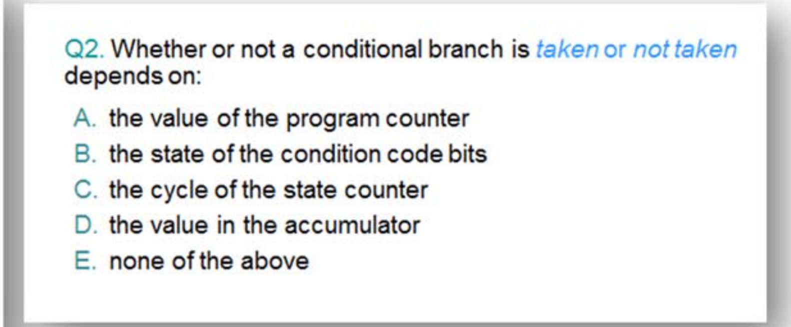
A. add a bi-directional path to the data bus

B. use the ALU to compute the address of the next instruction

C. make it an up/down counter

D. add a two's complement N-bit adder circuit (where N is the address bus width)

E. none of the above



Q2. Whether or not a conditional branch is *taken* or *not taken* depends on:

A. the value of the program counter

B. the state of the condition code bits

C. the cycle of the state counter

D. the value in the accumulator

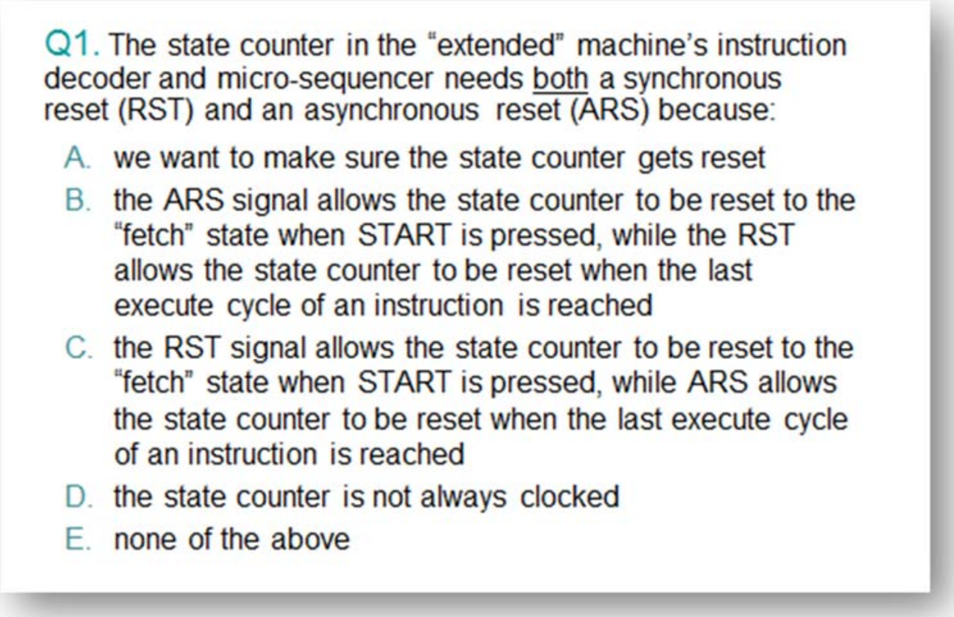
E. none of the above

# Lecture Summary – Module 4-K

*Simple Computer – Advanced Extensions*

**Reference:** Meyer Supplemental Text, pp. 50-64

- **overview**
  - **advanced extensions include**
    - **multi-cycle execution**
    - **stack mechanism**
- **state counter modifications**
  - **provide multiple execute cycles (here, up to 3)**
  - **determine number of execute cycles based on opcode**
  - **realize using 2-bit synchronously resettable state counter [SQB SQA]**
  - **new state names**
    - **S0 – fetch**
    - **S1..S3 – execute (first, second, third)**
  - **new control signal: RST (asserted on final execute state of each instruction)**



Q1. The state counter in the "extended" machine's instruction decoder and micro-sequencer needs both a synchronous reset (RST) and an asynchronous reset (ARS) because:

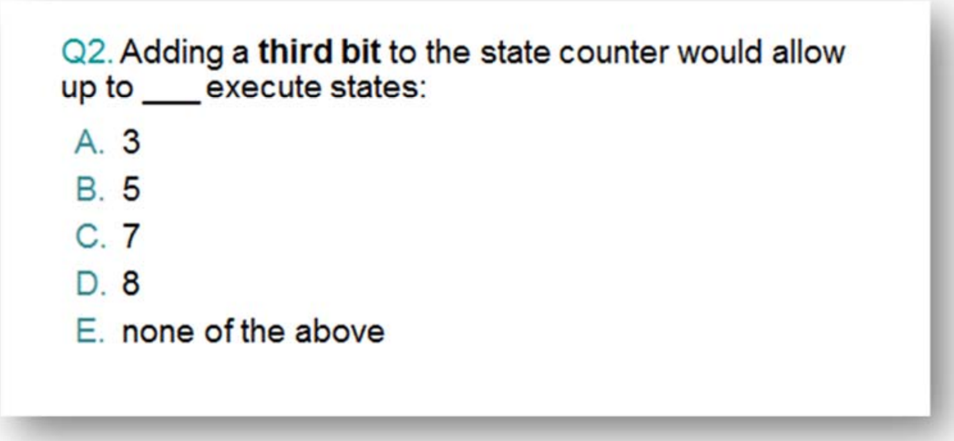
A. we want to make sure the state counter gets reset

B. the ARS signal allows the state counter to be reset to the "fetch" state when START is pressed, while the RST allows the state counter to be reset when the last execute cycle of an instruction is reached

C. the RST signal allows the state counter to be reset to the "fetch" state when START is pressed, while ARS allows the state counter to be reset when the last execute cycle of an instruction is reached

D. the state counter is not always clocked

E. none of the above



Q2. Adding a **third bit** to the state counter would allow up to ___ execute states:

A. 3

B. 5

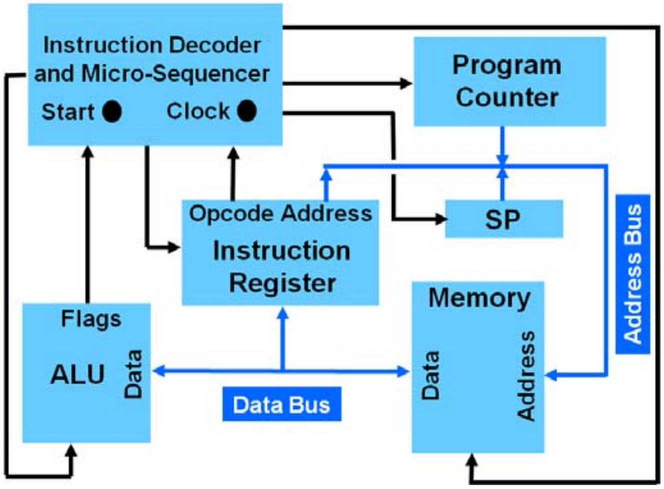
C. 7

D. 8

E. none of the above

```
/* Instruction Decoder and Microsequencer with Multi-Execution States */
module idmsr(CLK, START, OP, MSL, MOE, MWE, PCC, POA, ARS, IRL, IRA, ALE, ALX, ALY, AOE);
  input wire CLK;
  input wire START;                     // Asynchronous START pushbutton
  input wire [2:0] OP;                  // opcode bits (input from IR5..IR7)
  output wire MSL, MOE, MWE;            // Memory control signals
  output wire PCC, POA, ARS;            // PC control signals
  output wire IRL, IRA;                 // IR control signals
  output wire ALE, ALX, ALY, AOE;       // ALU control signals
  reg SQA, SQB;                         // State counter low bit, high bit
  reg RUN;                              // RUN/HLT state
  wire RST;                             // Synchronous state counter reset
  wire LDA, STA, ADD, SUB, AND, HLT;
  wire [3:0] S;
  reg next_SQA, next_SQB;
  wire RUN_ar;                          // Asynchronous reset for RUN
  // Decoded opcode definitions
  assign HLT = ~OP[2] & ~OP[1] & ~OP[0];       // HLT opcode = 000
  assign LDA = ~OP[2] & ~OP[1] &  OP[0];       // LDA opcode = 001
  assign ADD = ~OP[2] &  OP[1] & ~OP[0];       // ADD opcode = 010
  assign SUB = ~OP[2] &  OP[1] &  OP[0];       // SUB opcode = 011
  assign AND =  OP[2] & ~OP[1] & ~OP[0];       // AND opcode = 100
  assign STA =  OP[2] & ~OP[1] &  OP[0];       // STA opcode = 101

  // Decoded state definitions
  assign S[0] = ~SQB & ~SQA;            // fetch state
  assign S[1] = ~SQB &  SQA;            // first execute state
  assign S[2] =  SQB & ~SQA;            // second execute state
  assign S[3] =  SQB &  SQA;            // third execute state
  // State counter
  always @ (posedge CLK, posedge START) begin
    if(START == 1'b1) begin             // start in fetch state
      SQA <= 1'b0;
      SQB <= 1'b0;
    end else begin
      SQA <= next_SQA;
      SQB <= next_SQB;
    end
  end
  always @ (RST, RUN, SQA, SQB) begin
    next_SQA = ~RST & RUN & ~SQA;              // if RUN negated or RST asserted,
    next_SQB = ~RST & RUN & (SQA ^ SQB);       //    state counter is reset
  end
  assign RUN_ar = S[1] & HLT;
  // Run/stop
  always @ (posedge CLK, posedge RUN_ar, posedge START) begin
    if(START == 1'b1)                   // start with RUN set to 1
      RUN <= 1'b1;
    else if(RUN_ar == 1'b1)             // RUN is cleared when HLT is executed
      RUN <= 1'b0;
  end
  // System control equations
  assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
  assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
  assign MWE = S[1] & STA;
  assign ARS = START;
  assign PCC = RUN & S[0];
  assign POA = S[0];
  assign IRL = RUN & S[0];
  assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
  assign AOE = S[1] & STA;
  assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
  assign ALX = S[1] & (LDA | AND);
  assign ALY = S[1] & (SUB | AND);
  assign RST = S[1] & (LDA | STA | ADD | SUB | AND);
endmodule
```

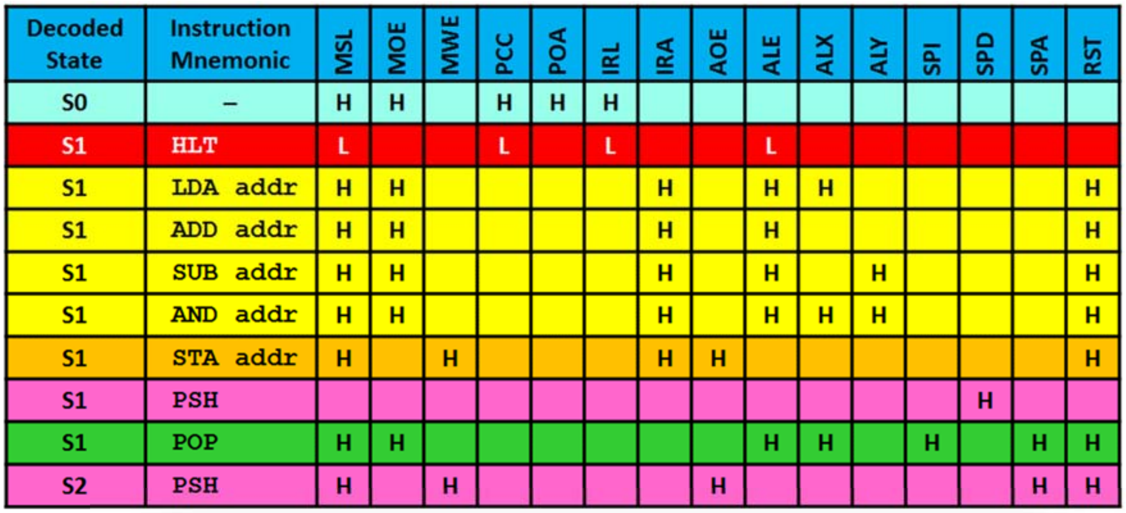
- **stack mechanism**
  - **defn: last-in, first-out (LIFO) data structure**
  - **primary uses of stacks in computers**
    - **subroutine linkage**
    - **saving/restoring machine context**
    - **expression evaluation**
  - **conventions**
    - **stack area usually placed at “top” of memory (highest address range)**
    - **stack pointer (SP) register used to indicate address of *top stack item***
    - **stack *growth* is toward *decreasing addresses***
  - **SP register control signals**
    - **SPI – stack pointer increment**
    - **SPD – stack pointer decrement**
    - **SPA – stack pointer output on address bus**
    - **ARS – asynchronous reset (“stack empty” → (SP) = 00000)**



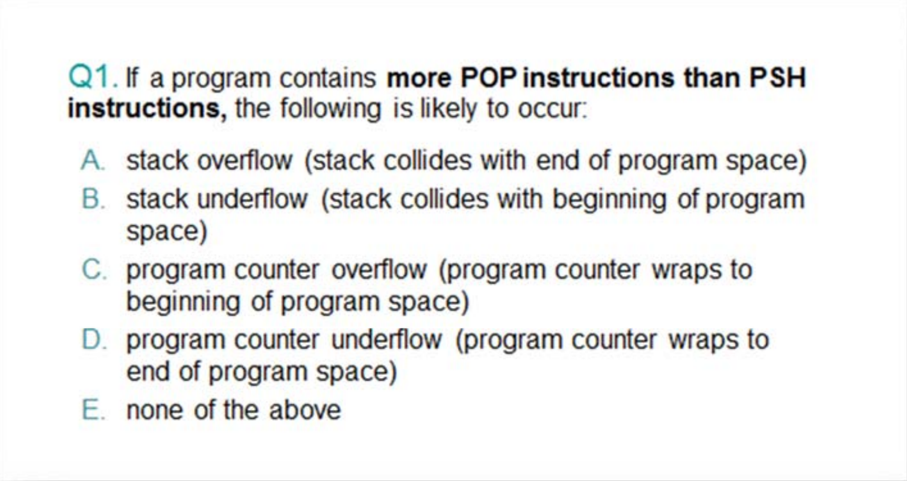
```
/* Stack Pointer */

module sp(CLK, SPI, SPD, SPA, ARS, ADRBUS_z);
  // NOTE: Assume SPI and SPD are mutually exclusive
  input wire CLK;
  input wire SPI, SPD;            // SP increment, decrement
  input wire SPA;                 // SP output on address but tri-state enable
  input wire ARS;                 // asynchronous reset (connected to START)
  output wire [4:0] ADRBUS_z;     // address bus
  reg [4:0] SP, next_SP;
  assign ADRBUS_z = SPA ? SP : 5'bZZZZZ;
  always @ (posedge CLK, posedge ARS) begin
    if (ARS == 1'b1)
      SP <= 5'b00000;
    else
      SP <= next_SP;
  end
  always @ (SPI, SPD, SP) begin
    if (SPI == 1’b1)              // increment
      next_SP = SP + 1;
    else if (SPD == 1’b1)         // decrement
      next_SP = SP - 1;
    else                          // retain state
      next_SP = SP;
  end
endmodule
```

- **stack mechanism, continued…**
  - **new instructions** ***understand this notation***
    - **PSH – save (A) on stack**
      - **(SP) ← (SP) – 1** **SPD**
      - **((SP)) ← (A)** **SPA, MSL, MWE, AOE**
    - **POP – load A with value of top stack item**
      - **(A) ← ((SP))**
      - **(SP) ← (SP) + 1**

      } **SPA, MSL, MOE, ALE, ALX, SPI**
  - **note the *overlap* of operations (single execute state) possible with "POP"**



| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | IRL | IRA | AOE | ALE | ALX | ALY | SPI | SPD | SPA | RST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H | | H | H | H | | | | | | | | | |
| S1 | HLT | L | | | L | | L | | | L | | | | | | |
| S1 | LDA addr | H | H | | | | | H | | H | H | | | | | H |
| S1 | ADD addr | H | H | | | | | H | | H | | | | | | H |
| S1 | SUB addr | H | H | | | | | H | | H | | H | | | | H |
| S1 | AND addr | H | H | | | | | H | | H | H | H | | | | H |
| S1 | STA addr | H | | H | | | | H | H | | | | | | | H |
| S1 | PSH | | | | | | | | | | | | | H | | |
| S1 | POP | H | H | | | | | | | H | H | | H | | H | H |
| S2 | PSH | H | | H | | | | | H | | | | | | H | H |



Q1. If a program contains **more POP instructions than PSH instructions,** the following is likely to occur:

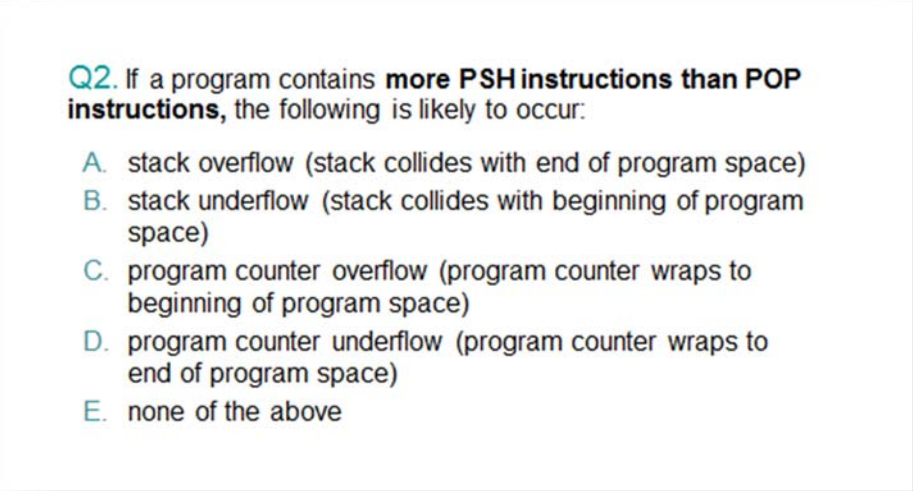
A. stack overflow (stack collides with end of program space)

B. stack underflow (stack collides with beginning of program space)

C. program counter overflow (program counter wraps to beginning of program space)

D. program counter underflow (program counter wraps to end of program space)

E. none of the above



Q2. If a program contains **more PSH instructions than POP instructions,** the following is likely to occur:

A. stack overflow (stack collides with end of program space)

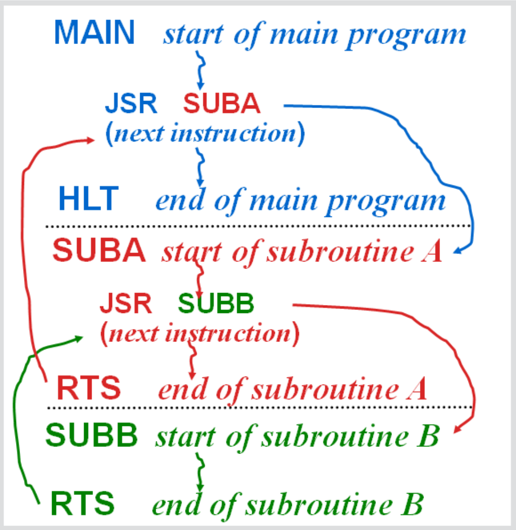
B. stack underflow (stack collides with beginning of program space)

C. program counter overflow (program counter wraps to beginning of program space)

D. program counter underflow (program counter wraps to end of program space)

E. none of the above

- **subroutine linkage**
  - **capabilities provided**
    - **arbitrary nesting of subroutine calls**
    - **passing parameters to subroutine**
    - **recursion**
    - **reentrancy**


MAIN *start of main program*
JSR SUBA
*(next instruction)*
HLT *end of main program*
SUBA *start of subroutine A*
JSR SUBB
*(next instruction)*
RTS *end of subroutine A*
SUBB *start of subroutine B*
RTS *end of subroutine B*

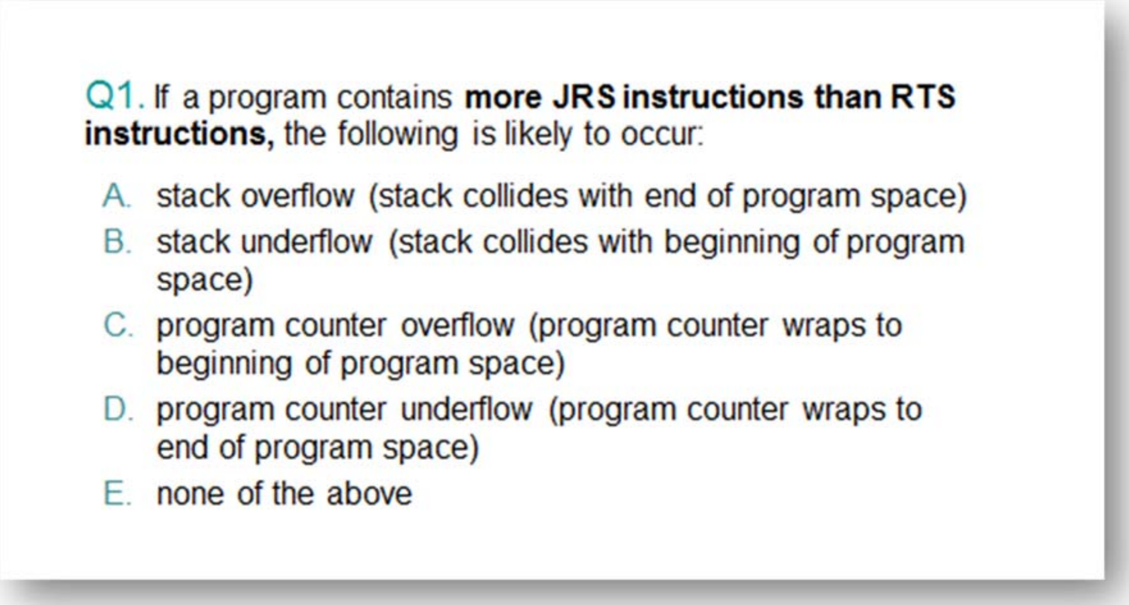
  - **new instructions** ***understand this notation***
    - **JSR *addr* – jump to subroutine at location *addr***
      - **(SP) ← (SP) – 1** **SPD**
      - **((SP)) ← (PC)** **SPA, MSL, MWE, POD**
      - **(PC) ← ($IR_{5..0}$)** **IRA,PLA**
    - **RTS – return from subroutine**
      - **(PC) ← ((SP))**
      - **(SP) ← (SP) + 1**

      } **SPA, MSL, MOE, PLD, SPI**
      ***note: value loaded into PC truncated to 5 bits***
    - **note the *overlap* of operations (single execute state) possible with "RTS"**
  - **need PC with bi-directional data bus interface**

| Decoded State | Instruction Mnemonic | MSL | MOE | MWE | PCC | POA | PLA | POD | PLD | IRL | IRA | AOE | ALE | ALX | ALY | SPI | SPD | SPA | RST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S0 | – | H | H |  | H | H |  |  |  | H |  |  |  |  |  |  |  |  |  |
| S1 | HLT | L |  |  | L |  |  |  |  | L |  |  | L |  |  |  |  |  |  |
| S1 | LDA addr | H | H |  |  |  |  |  |  |  | H |  | H | H |  |  |  |  | H |
| S1 | ADD addr | H | H |  |  |  |  |  |  |  | H |  | H |  |  |  |  |  | H |
| S1 | SUB addr | H | H |  |  |  |  |  |  |  | H |  | H |  | H |  |  |  | H |
| S1 | AND addr | H | H |  |  |  |  |  |  |  | H |  | H | H | H |  |  |  | H |
| S1 | STA addr | H |  | H |  |  |  |  |  |  | H | H |  |  |  |  |  |  | H |
| S1 | JSR addr |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | H |  |  |
| S1 | RTS | H | H |  |  |  |  |  | H |  |  |  |  |  |  | H |  | H | H |
| S2 | JSR addr | H |  | H |  |  |  | H |  |  |  |  |  |  |  |  |  | H |  |
| S3 | JSR addr |  |  |  |  |  | H |  |  |  | H |  |  |  |  |  |  |  | H |

```
/* Program Counter with Data Bus interface */

module pc(CLK, PCC, PLA, POA, RST, ADRBUS_z, DB_z, PLD, POD, PC);
  input wire CLK;
  input wire PCC;                // PC count enable
  input wire PLA;                // PC load from address bus enable
  input wire POA;                // PC output on address bus tri-state enable
  input wire RST;                // Asynchronous reset (connected to START)
  input wire PLD;                // PC load from data bus enable
  input wire POD;                // PC output on data bus tri-state enable
  inout wire [4:0] ADRBUS_z;     // address bus (5-bits wide)
  inout wire [7:0] DB_z;         // data bus (8-bits wide)
  output reg [4:0] PC;           // PC register
  reg [4:0] next_PC;

  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      PC <= 5'b00000;
    else
      PC <= next_PC;
  end

  always @ (PLA, PLD, PCC, ADRBUS_z, DB_z, PC) begin
  // synchronous control signals PLA, PLD, and PCC are mutually exclusive
    if (PLA == 1’b1)             // load PC from address bus
      next_PC = ADRBUS_z;
    else if (PLD == 1’b1)  // load PC from data bus
      next_PC = DB_z;
    else if (PCC == 1’b1)  // increment PC
      next_PC = PC + 1;
    else                   // retain state
      next_PC = PC;
  end

  assign ADRBUS_z = POA ? PC[4:0] : 5'bZZZZZ;
  assign DB_z = POD ? {3'b000, PC[4:0]} : 8'bZZZZZZZZ;  // pad upper 3 bits of DB w/ 0

endmodule
```

```
// System control equations

  assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND | RTS) | S[2] & JSR);
  assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND | RTS);
  assign MWE = S[1] & STA | S[2] & JSR;
  assign ARS = START;
  assign PCC = RUN & S[0];
  assign POA = S[0];
  assign PLA = S[3] & JSR;
  assign POD = S[2] & JSR;
  assign PLD = S[1] & RTS;
  assign IRL = RUN & S[0];
  assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
  assign AOE = S[1] & STA;
  assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
  assign ALX = S[1] & (LDA | AND);
  assign ALY = S[1] & (SUB | AND);
  assign SPI = S[1] & RTS;
  assign SPD = S[1] & JSR;
  assign SPA = S[1] & RTS | S[2] & JSR;
  assign RST = S[1] & (LDA | STA | ADD | SUB | AND | RTS) | S[3] & JSR;
endmodule
```



Q1. If a program contains **more JRS instructions than RTS instructions,** the following is likely to occur:

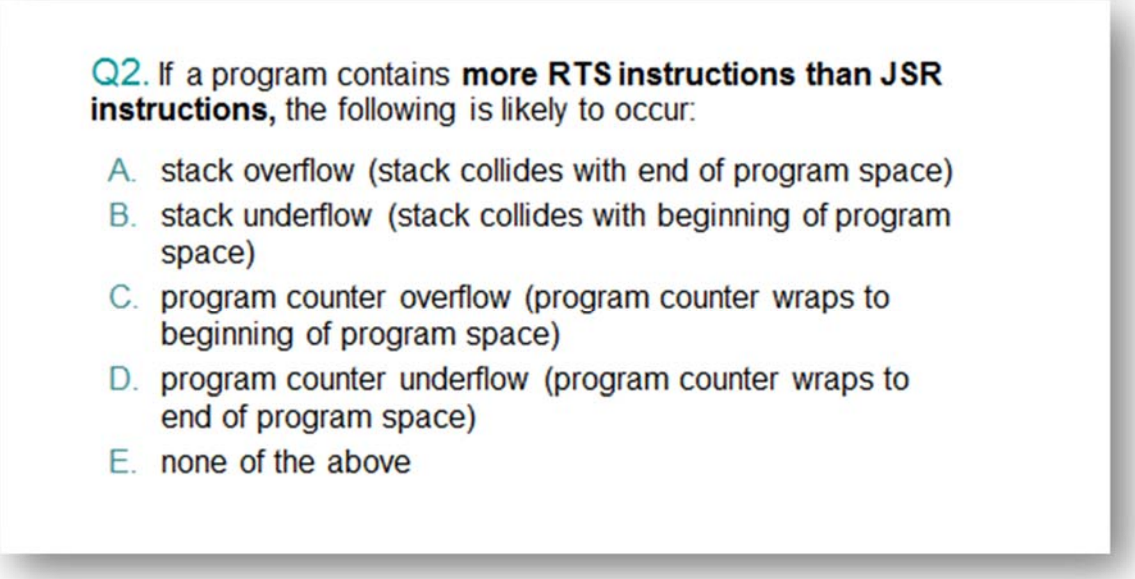
A. stack overflow (stack collides with end of program space)

B. stack underflow (stack collides with beginning of program space)

C. program counter overflow (program counter wraps to beginning of program space)

D. program counter underflow (program counter wraps to end of program space)

E. none of the above



Q2. If a program contains **more RTS instructions than JSR instructions,** the following is likely to occur:

A. stack overflow (stack collides with end of program space)

B. stack underflow (stack collides with beginning of program space)

C. program counter overflow (program counter wraps to beginning of program space)

D. program counter underflow (program counter wraps to end of program space)

E. none of the above

**Fun things to think about…**

- **what kinds of new instructions would be useful in writing "real" programs?**
- **what new kinds of registers would be good to add to the machine?**
- **what new kinds of addressing modes would be nice to have?**
- **what would we have to change if we wanted "branch" transfer-of-control instructions instead of "jump" instructions?**

**These are all good reasons to "continue your 'digital life' beyond this course"!**