Lecture Summary – Module 4

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

- **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 numbers identical

Radix has no "negative zero"

All negative numbers different

Radix has extra negative number

Observations:

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**

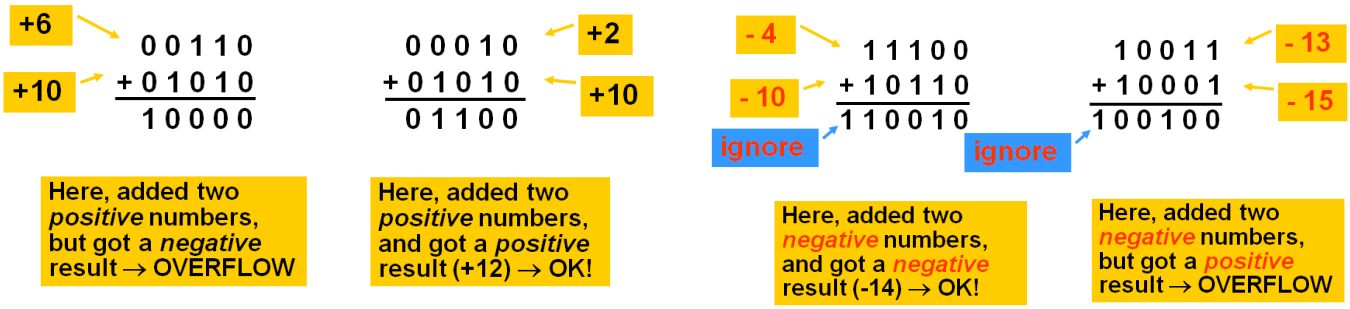
- **sign extension: SM – pad magnitude with leading zeroes; R and DR – replicate the sign digit**

Lecture Summary – Module 4-B

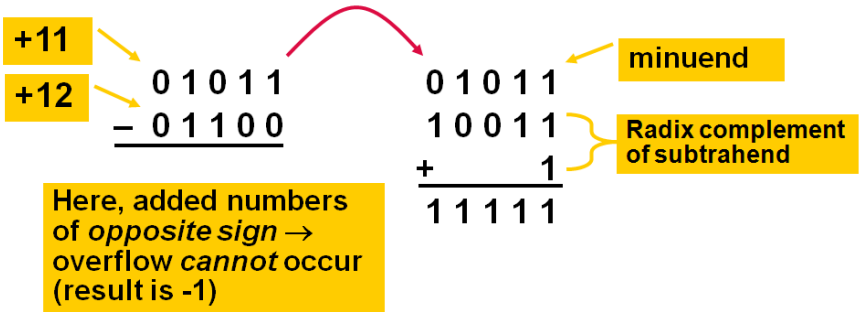
Radix Addition and Subtraction

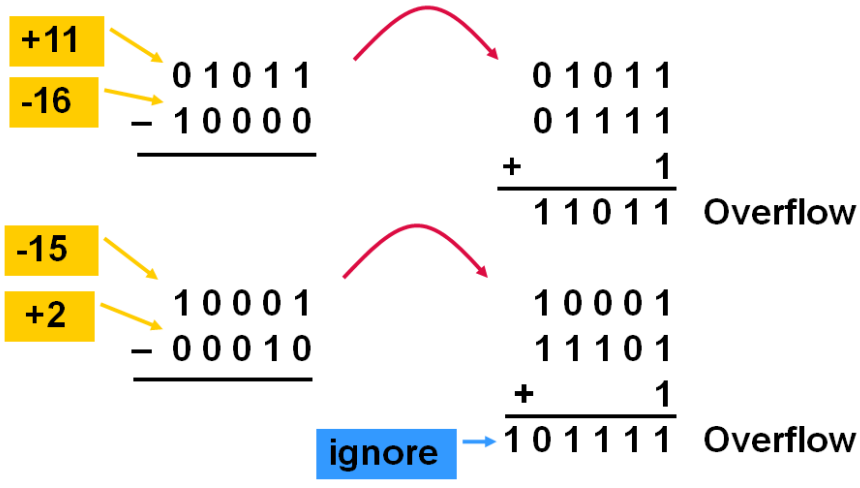
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 39-43

- **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***



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





Lecture Summary – Module 4-C

Adder, Subtractor, and Comparator Circuits

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 458-466, 474-478

- **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**

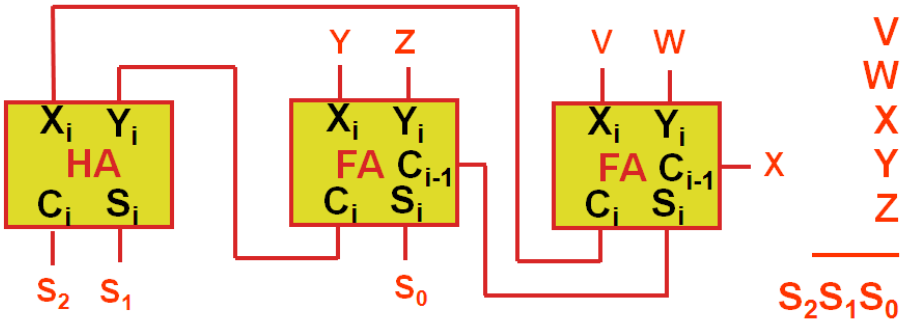
- **building blocks**
  - **half adder**

Xi	Yi	Ci	Si
0	0		
0	1		
1	0		
1	1		

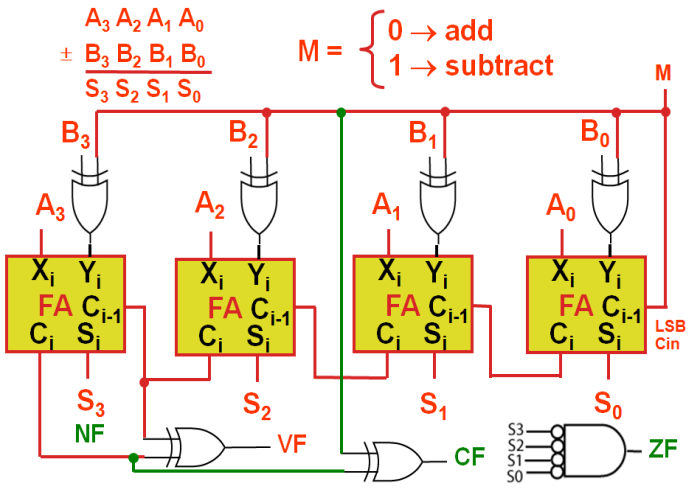
  - **full adder**

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**

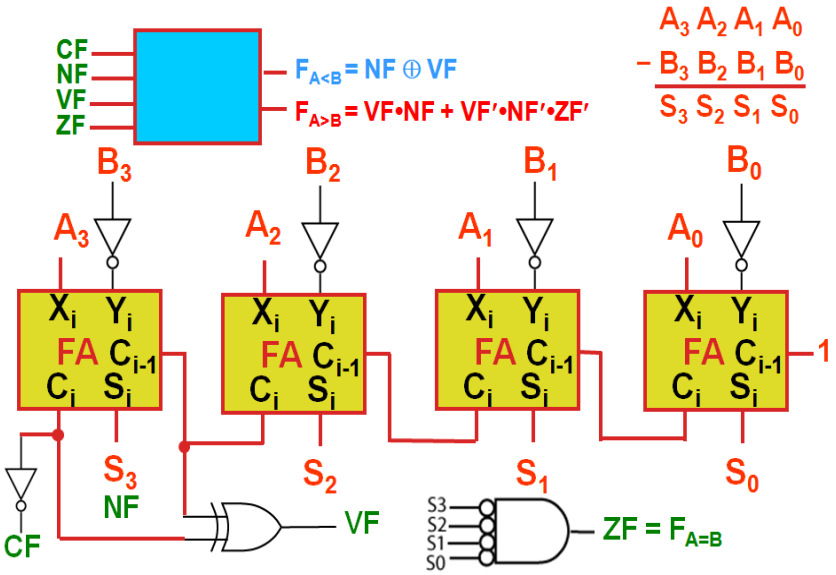


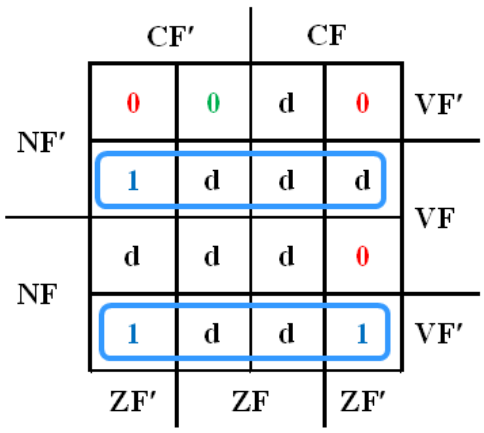
- **multi-digit adder/subtractor**
  - **ripple = iterative**
  - **to subtract, take DR radix of subtrahend and add 1**
  - **conditions of interest ("condition codes")**
    - **overflow (VF)**
    - **negative (NF)**
    - **zero (ZF)**
    - **carry/borrow (CF)**



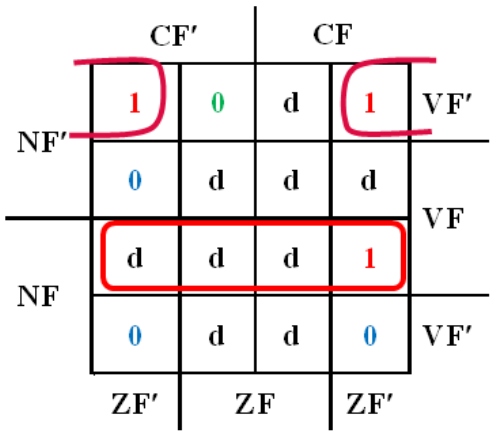
- **magnitude comparator**
  - **calculate A–B and condition codes**
  - **results (A=B, A<B, A>B) are *functions of the condition codes***

A1	A0	B1	B0	?	CF	ZF	NF	VF
0	0	0	0	A=B	0	1	0	0
0	0	0	1	A<B	1	0	1	0
0	0	1	0	A>B	1	0	1	1
0	0	1	1	A>B	1	0	0	0
0	1	0	0	A>B	0	0	0	0
0	1	0	1	A=B	0	1	0	0
0	1	1	0	A>B	1	0	1	1
0	1	1	1	A>B	1	0	1	1
1	0	0	0	A<B	0	0	1	0
1	0	0	1	A<B	0	0	0	1
1	0	1	0	A=B	0	1	0	0
1	0	1	1	A<B	1	0	1	0
1	1	0	0	A<B	0	0	1	0
1	1	0	1	A<B	0	0	1	0
1	1	1	0	A>B	0	0	0	0
1	1	1	1	A=B	0	1	0	0





$F_{A<B} = NF \oplus VF$



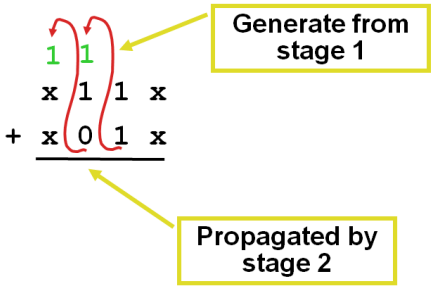
$F_{A>B} = VF \cdot NF + VF' \cdot NF' \cdot ZF'$

Lecture Summary – Module 4-D

Carry Look-Ahead (CLA) Adder Circuits

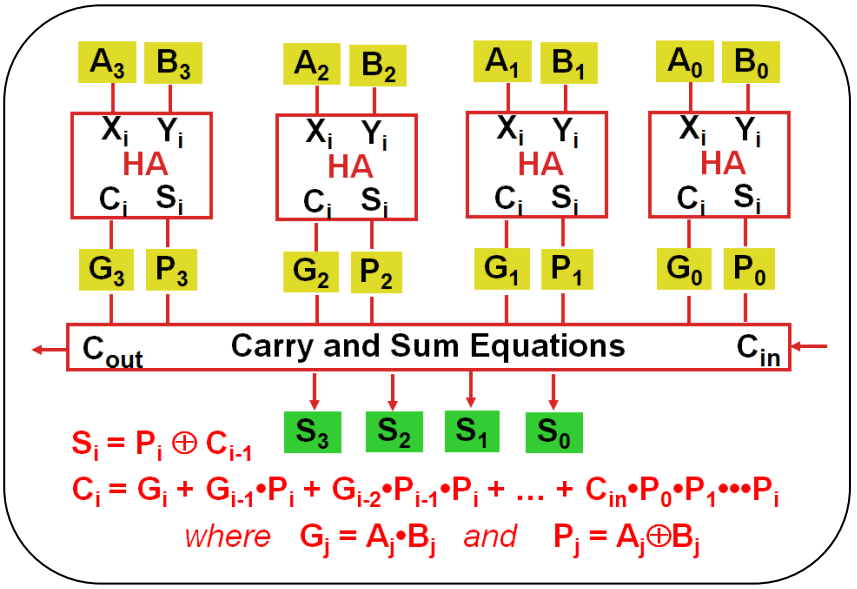
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 478-482, 484-488

- **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*)** $G_i = X_i \cdot Y_i$
  - **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** $C_i = Gi + C_{i-1} \cdot P_i$

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



- **rewriting carry equations for 4-bit adder in terms of *available inputs* (successive expansion)**
  - $C_{-1} = C_{in}$
  - $C_0 = G_0 + C_{in} \cdot P_0$
  - $C_1 = G_1 + C_0 \cdot P_1 = G_1 + (G_0 + C_{in} \cdot P_0) \cdot P_1 = G_1 + G_0 \cdot P_1 + C_{in} \cdot P_0 \cdot P_1$
  - ***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 ABEL**

```
MODULE cla4
TITLE '4-bit Carry Look-Ahead Adder'

DECLARATIONS
X0..X3, Y0..Y3 pin;  " operands
CIN pin;  " carry in
S0..S3 pin istype 'com';  " sum outputs
G0 = X0&Y0;  " generate function definitions
G1 = X1&Y1;
G2 = X2&Y2;
G3 = X3&Y3;
P0 = X0$Y0;  " propagate function definitions
P1 = X1$Y1;
P2 = X2$Y2;
P3 = X3$Y3;
C0 = G0 # CIN&P0; " carry function definitions
C1 = G1 # G0&P1 # CIN&P0&P1;
C2 = G2 # G1&P2 # G0&P1&P2 # CIN&P0&P1&P2;
C3 = G3 # G2&P3 # G1&P2&P3 # G0&P1&P2&P3 # CIN&P0&P1&P2&P3;

EQUATIONS
S0 = CIN$P0;
S1 = C0$P1;
S2 = C1$P2;
S3 = C2$P3;

END
```

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

```
MODULE cla4p
TITLE '4-bit Carry Look-Ahead Adder Using + Operator'

DECLARATIONS
X0..X3, Y0..Y3 pin;  " operands
CIN pin;  " carry in
S0..S3 pin istype 'com';  " sum outputs
X = [X3..X0];     " input operand set definition
Y = [Y3..Y0];
C = [0,0,0,CIN];  " carry in set definition
S = [S3..S0];     " output sum set definition

EQUATIONS
S = X + Y + C;

END
```

- **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
```

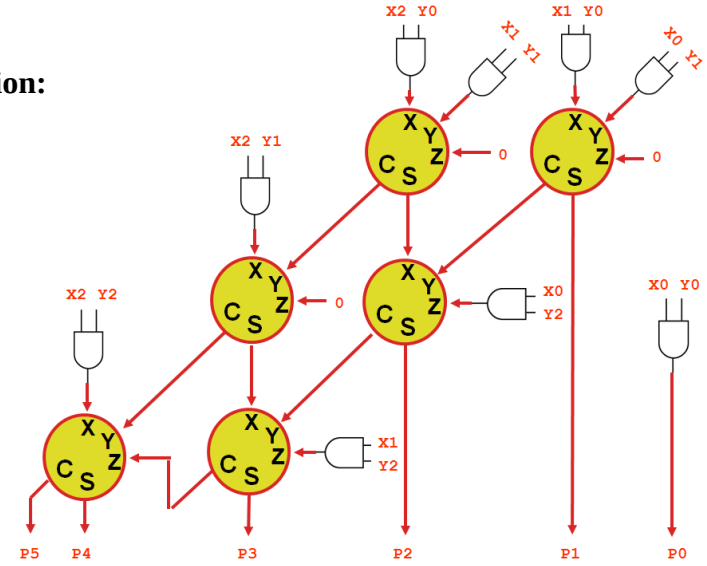
Lecture Summary – Module 4-E

Multiplier Circuits

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 45-47, 494-497

- **overview**
  - **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
```

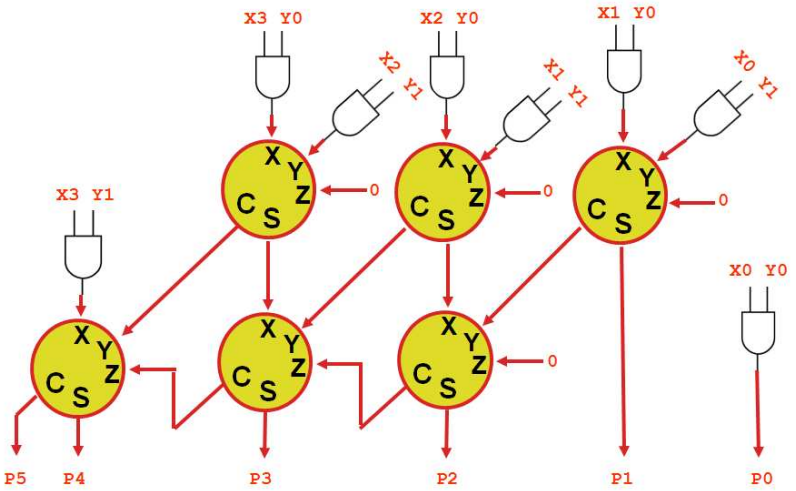


  - **based on "shift and add" algorithm**
  - **each row is called a *product component***
  - **each $x_i \cdot y_j$ term represents a *product component bit* (logical AND)**
  - **the *product* P is obtained by adding together the product components**

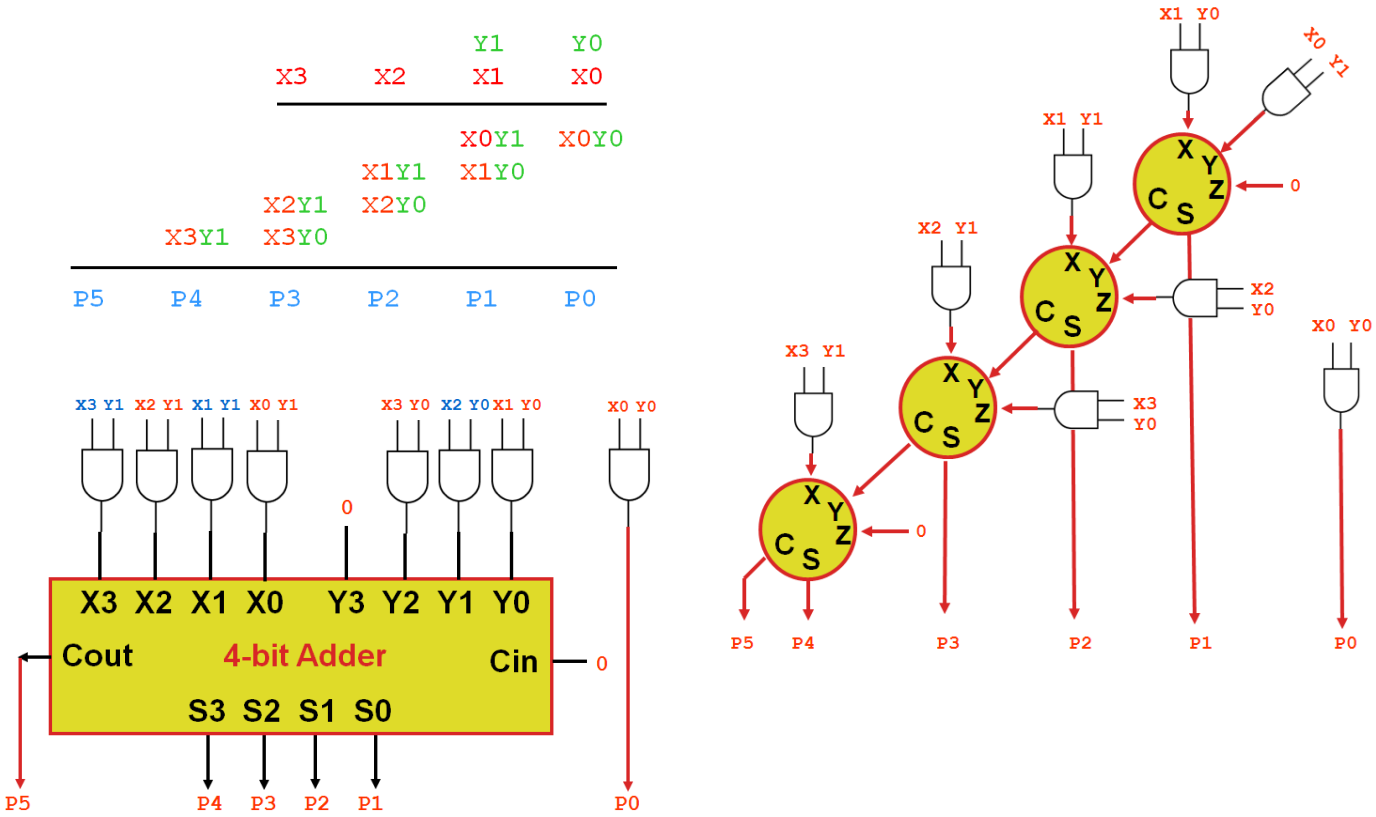
- **generalizations for an NxM multiplier array circuit**
  - **N = number of bits in multiplicand**
  - **M = number of bits in multiplier**
  - **produces an N+M digit result**
  - **requires NxM AND gates to generate the product components**
  - **requires N-1 "diagonals" of full adders**
  - **requires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

- **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**



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

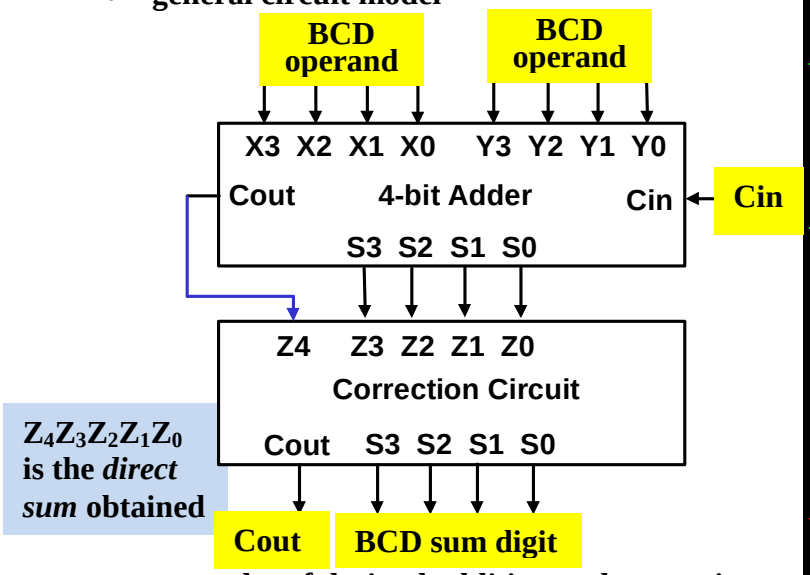
```
MODULE mul4x4
TITLE '4x4 Combinational Multiplier'

DECLARATIONS
X3..X0, Y3..Y0 pin;  " multiplicand, multiplier bits
P7..P0 pin istype 'com';  " product bits

P = [P7..P0]; " set of product bits
" Definition of product components
PC1 = Y0 & [ 0, 0, 0, 0,X3,X2,X1,X0];
PC2 = Y1 & [ 0, 0, 0,X3,X2,X1,X0, 0];
PC3 = Y2 & [ 0, 0,X3,X2,X1,X0, 0, 0];
PC4 = Y3 & [ 0,X3,X2,X1,X0, 0, 0, 0];

EQUATIONS
" Form unsigned sum of product components
P = PC1 + PC2 + PC3 + PC4;
END
```

Lecture Summary – Module 4-F

BCD Adder Circuits

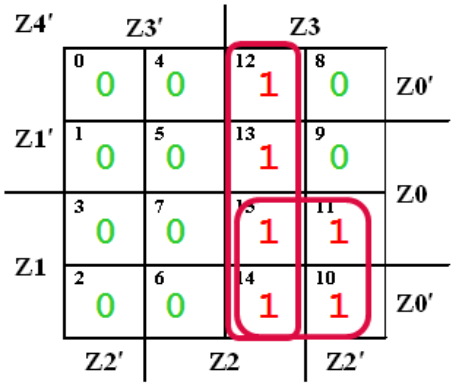
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 48-51

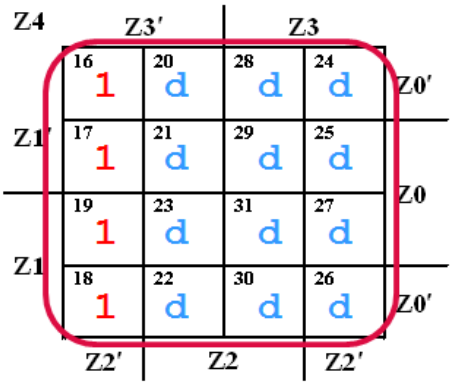
- **overview**
  - **external computer interfaces may need to read or display decimal digits (examples)**
  - **need to perform arithmetic operations on decimal numbers directly**
  - **most commonly used code in *binary-coded decimal* (BCD)**
  - **object is to design circuit that adds two BCD digit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



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>

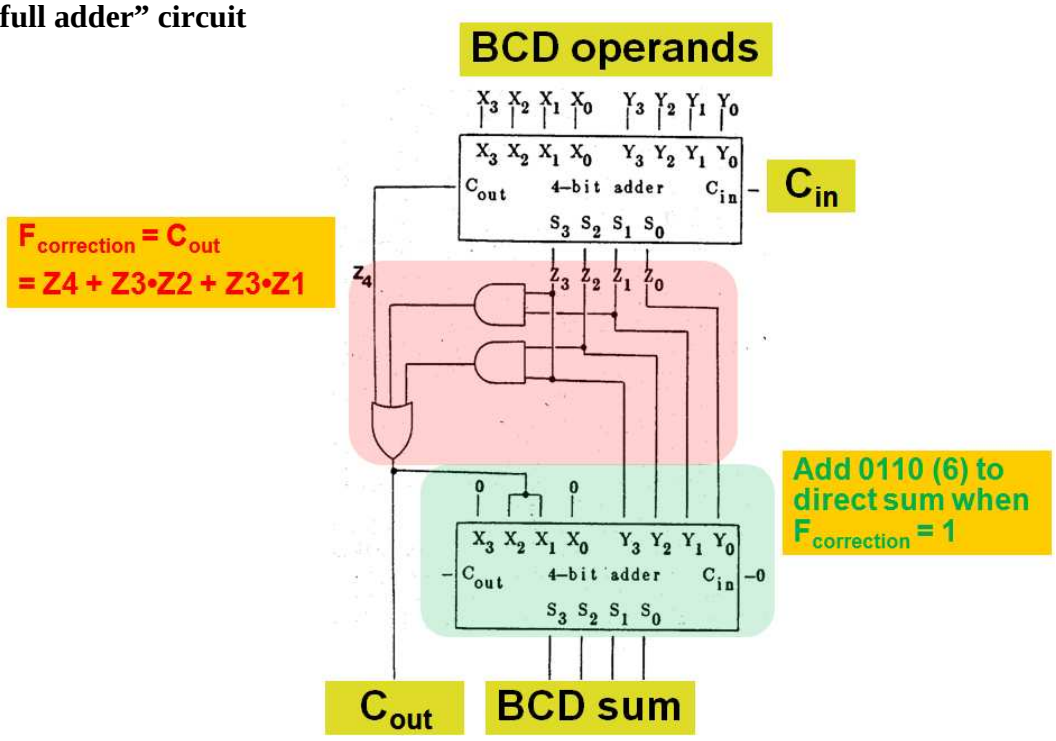
- **examples of decimal addition and correction**
- **summary of rules**
  - **if the sum of two BCD digits is ≤ 9 (i.e. 1001), no correction is needed**
  - **if the sum of two BCD digits is > 9, the result must be corrected by adding six (0110)**
- **"correction function" derivation**



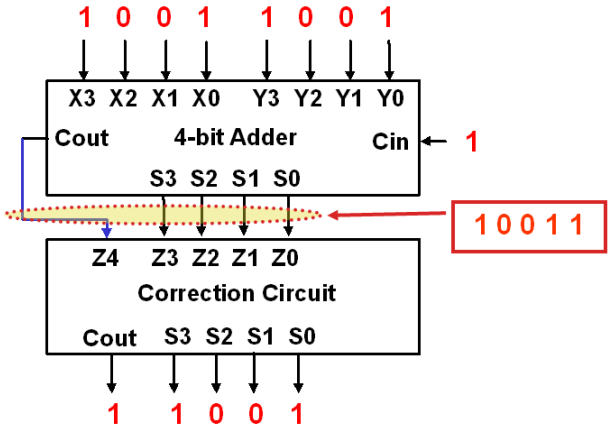


$F_{correction} = C_{out} = Z4 + Z3 \cdot Z2 + Z3 \cdot Z1$

- **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
TITLE  'Nines Complement Box'
DECLARATIONS
X3..X0 pin;  " Input code
Y3..Y0 pin istype 'com'; " Output code

TRUTH_TABLE ([X3, X2, X1, X0]->[Y3, Y2, Y1, Y0])
             [ 0,  0,  0,  0]->[ 1,  0,  0,  1];
             [ 0,  0,  0,  1]->[ 1,  0,  0,  0];
             [ 0,  0,  1,  0]->[ 0,  1,  1,  1];
             [ 0,  0,  1,  1]->[ 0,  1,  1,  0];
             [ 0,  1,  0,  0]->[ 0,  1,  0,  1];
             [ 0,  1,  0,  1]->[ 0,  1,  0,  0];
             [ 0,  1,  1,  0]->[ 0,  0,  1,  1];
             [ 0,  1,  1,  1]->[ 0,  0,  1,  0];
             [ 1,  0,  0,  0]->[ 0,  0,  0,  1];
             [ 1,  0,  0,  1]->[ 0,  0,  0,  0];
END
```

Lecture Summary – Module 4-G

Simple Computer – Top-Down Specification

Reference: Meyer Supplemental Text, pp. 1-18

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

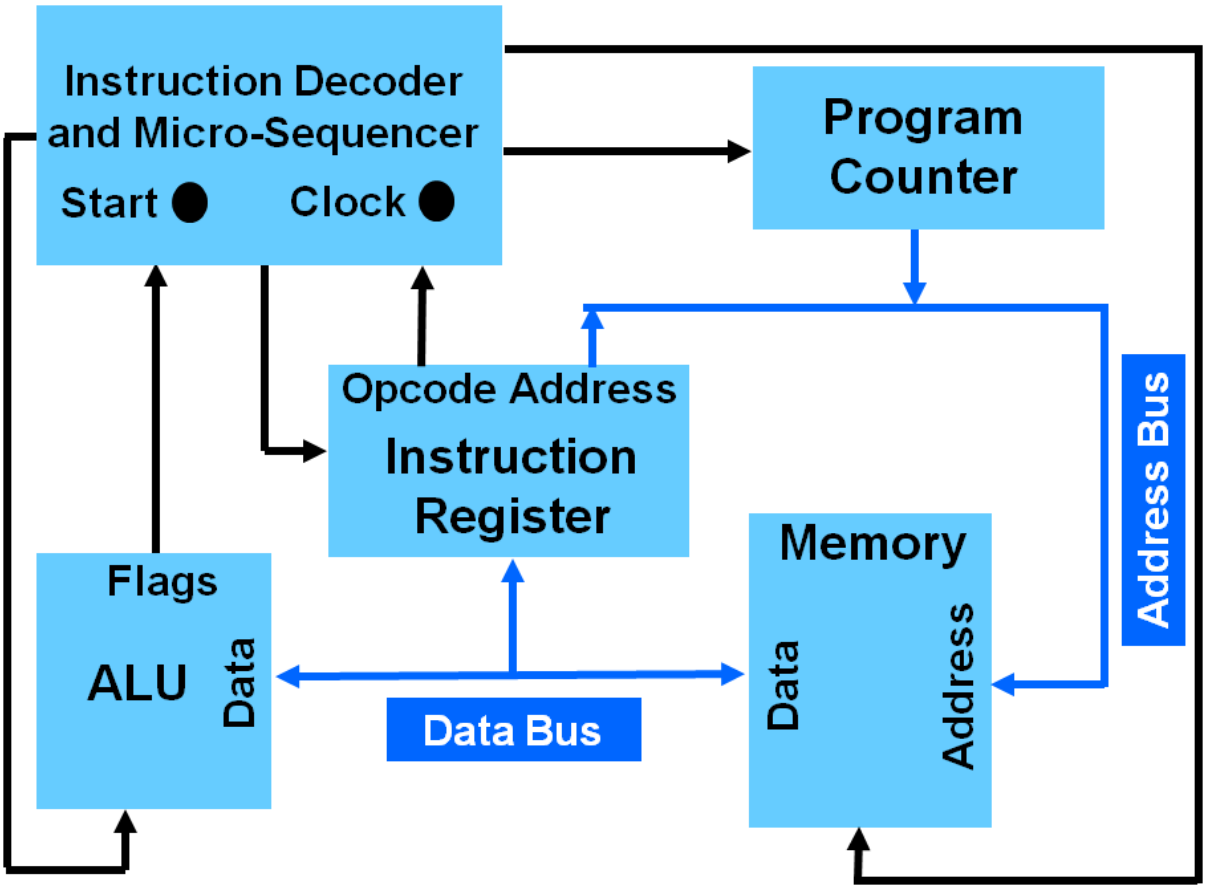
Opcode	Mnemonic	Function Performed
0 0 0	LDA *addr*	Load A with contents of location ***addr***
0 0 1	STA *addr*	Store contents of A at location ***addr***
0 1 0	ADD *addr*	Add contents of ***addr*** to contents of A
0 1 1	SUB *addr*	Subtract contents of ***addr*** from contents of A
1 0 0	AND *addr*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

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	00001011	Program
00001	01001100	
00010	00101101	
00011	00001011	
00100	10001100	
00101	00101110	
00110	00001011	
00111	01101100	
01000	00101111	
01001	10100000	
01010		
01011	10101010	Operands
01100	01010101	
01101		Results
01110		
01111		

Calculation of ADD, AND, and SUB results:

- **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**

Lecture Summary – Module 4-H

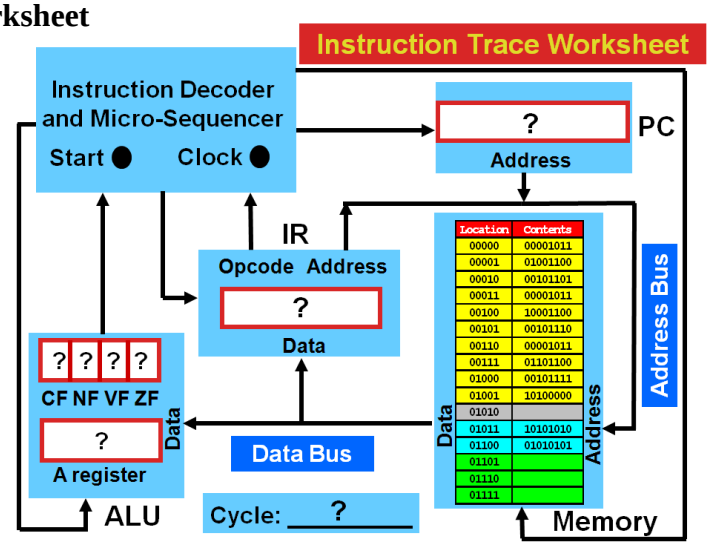
Simple Computer – Instruction Tracing

Reference: Meyer Supplemental Text, pp. 18-24

- **overview**
  - **two basic steps in "processing" an instruction**
    - **fetch**
    - **execute**
  - **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

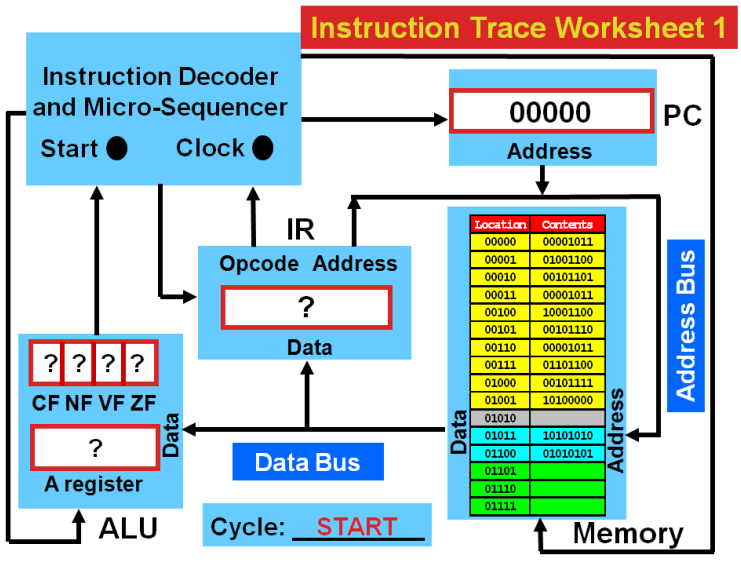
- **worksheet**

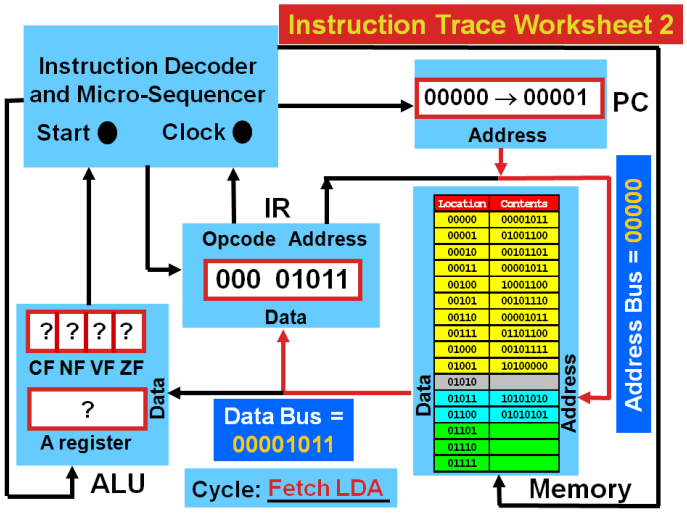


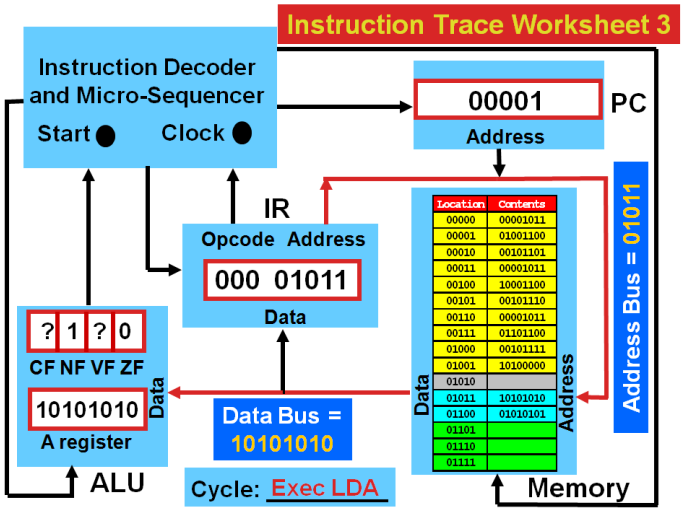
Notes:

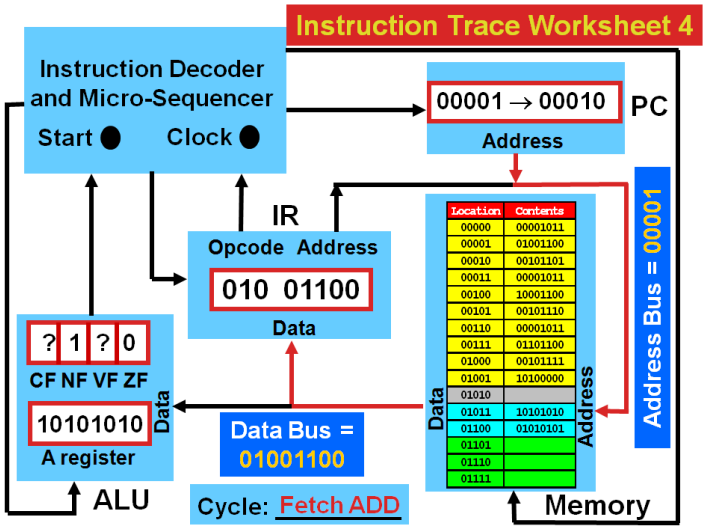
1. **The clock edges drive the <u>synchronous</u> functions of the computer (e.g., increment program counter)**
2. **The decoded states (here, fetch and execute) enable the <u>combinational</u> functions of the computer (e.g., turn on tri-state buffers)**

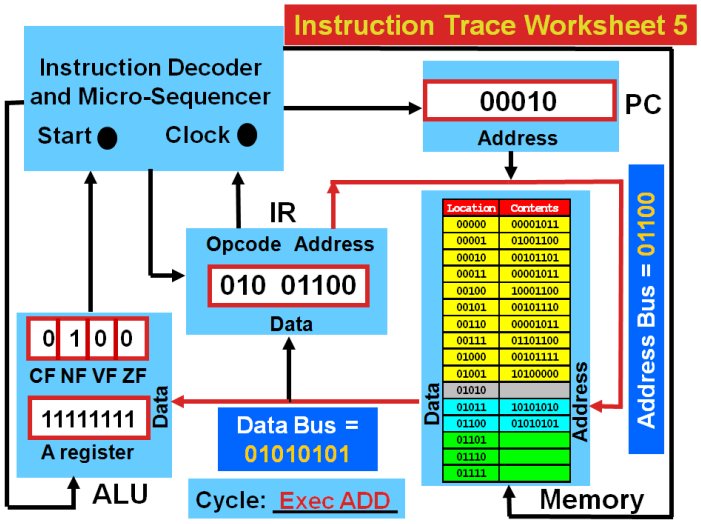
- **step 1 (after START pushbutton pressed)**

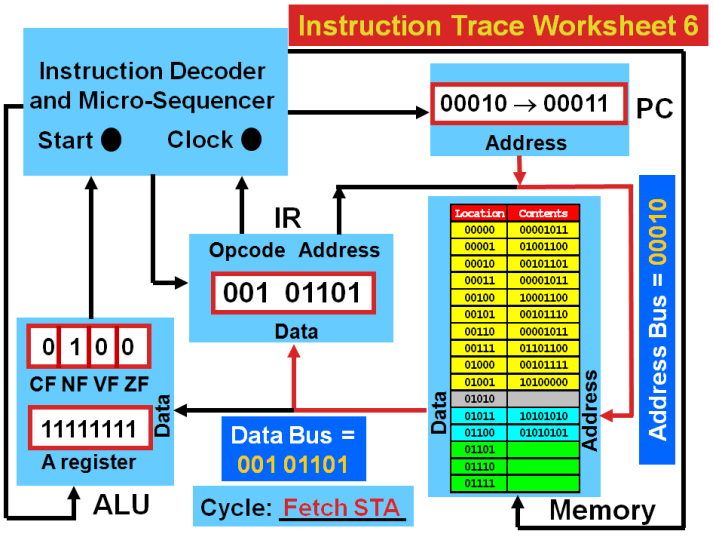


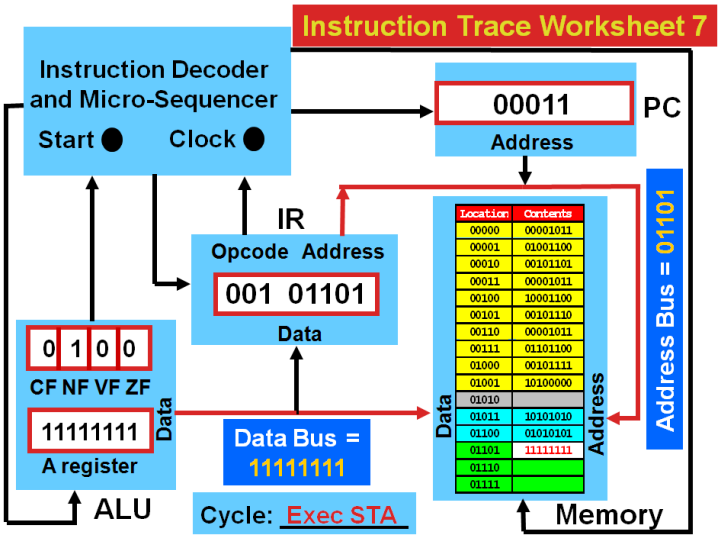

Instruction Trace Worksheet 2
Instruction Decoder and Micro-Sequencer
Start
Clock
00000 → 00001
PC
Address
IR
Opcode Address
000 01011
Data
? ? ? ?
CF NF VF ZF
?
A register
ALU
Data Bus = 00001011
Cycle: Fetch LDA
Memory
Address Bus = 00000


Instruction Trace Worksheet 3
Instruction Decoder and Micro-Sequencer
Start
Clock
00001
PC
Address
IR
Opcode Address
000 01011
Data
? 1 ? 0
CF NF VF ZF
10101010
A register
ALU
Data Bus = 10101010
Cycle: Exec LDA
Memory
Address Bus = 01011


Instruction Trace Worksheet 4
Instruction Decoder and Micro-Sequencer
Start
Clock
00001 → 00010
PC
Address
IR
Opcode Address
010 01100
Data
? 1 ? 0
CF NF VF ZF
10101010
A register
ALU
Data Bus = 01001100
Cycle: Fetch ADD
Memory
Address Bus = 00001


Instruction Trace Worksheet 5
Instruction Decoder and Micro-Sequencer
Start
Clock
00010
PC
Address
IR
Opcode Address
010 01100
Data
0 1 0 0
CF NF VF ZF
11111111
A register
ALU
Data Bus = 01010101
Cycle: Exec ADD
Memory
Address Bus = 01100

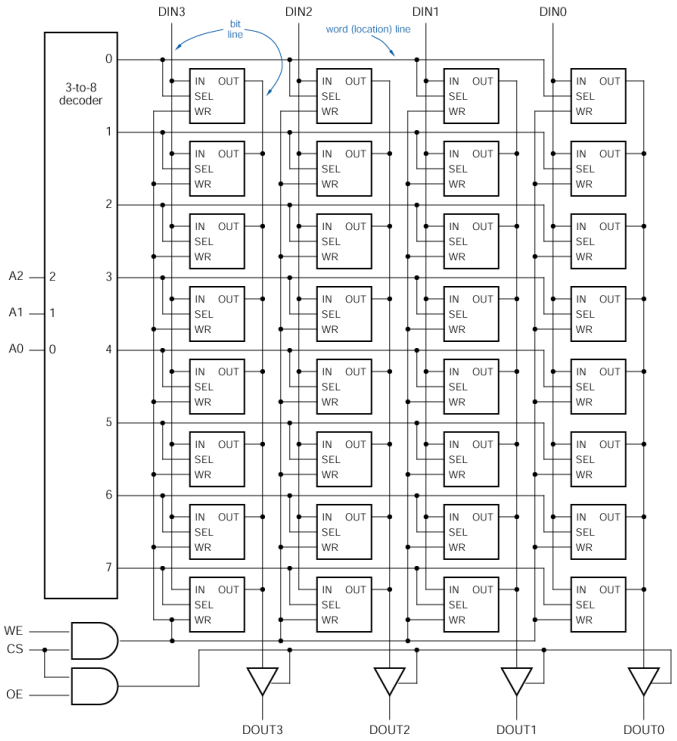

Instruction Trace Worksheet 6
Instruction Decoder and Micro-Sequencer
Start
Clock
00010 → 00011
PC
Address
IR
Opcode Address
001 01101
Data
0 1 0 0
CF NF VF ZF
11111111
A register
ALU
Data Bus = 001 01101
Cycle: Fetch STA
Memory
Address Bus = 00010


Instruction Trace Worksheet 7
Instruction Decoder and Micro-Sequencer
Start
Clock
00011
PC
Address
IR
Opcode Address
001 01101
Data
0 1 0 0
CF NF VF ZF
11111111
A register
ALU
Data Bus = 11111111
Cycle: Exec STA
Memory
Address Bus = 01101

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 combination**
    - **write operation involves open/closing latch → setup and hold timing matters**
- **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**

```
MODULE pc
TITLE    'Program Counter Module'
DECLARATIONS
CLOCK pin;
PC0..PC4 pin istype 'reg_D,buffer';
PCC pin; " PC count enable
POA pin; " PC output on address bus tri-state enable
ARS pin; " asynchronous reset (connected to START)

EQUATIONS
"       retain state   count up by 1
PC0.d = !PCC&PC0.q # PCC&!PC0.q;
PC1.d = !PCC&PC1.q # PCC&(PC1.q $ PC0.q);
PC2.d = !PCC&PC2.q # PCC&(PC2.q $ (PC1.q&PC0.q));
PC3.d = !PCC&PC3.q # PCC&(PC3.q $ (PC2.q&PC1.q&PC0.q));
PC4.d = !PCC&PC4.q # PCC&(PC4.q $ (PC3.q&PC2.q&PC1.q&PC0.q));
[PC0..PC4].oe = POA;
[PC0..PC4].ar = ARS;
[PC0..PC4].clk = CLOCK;
END
```

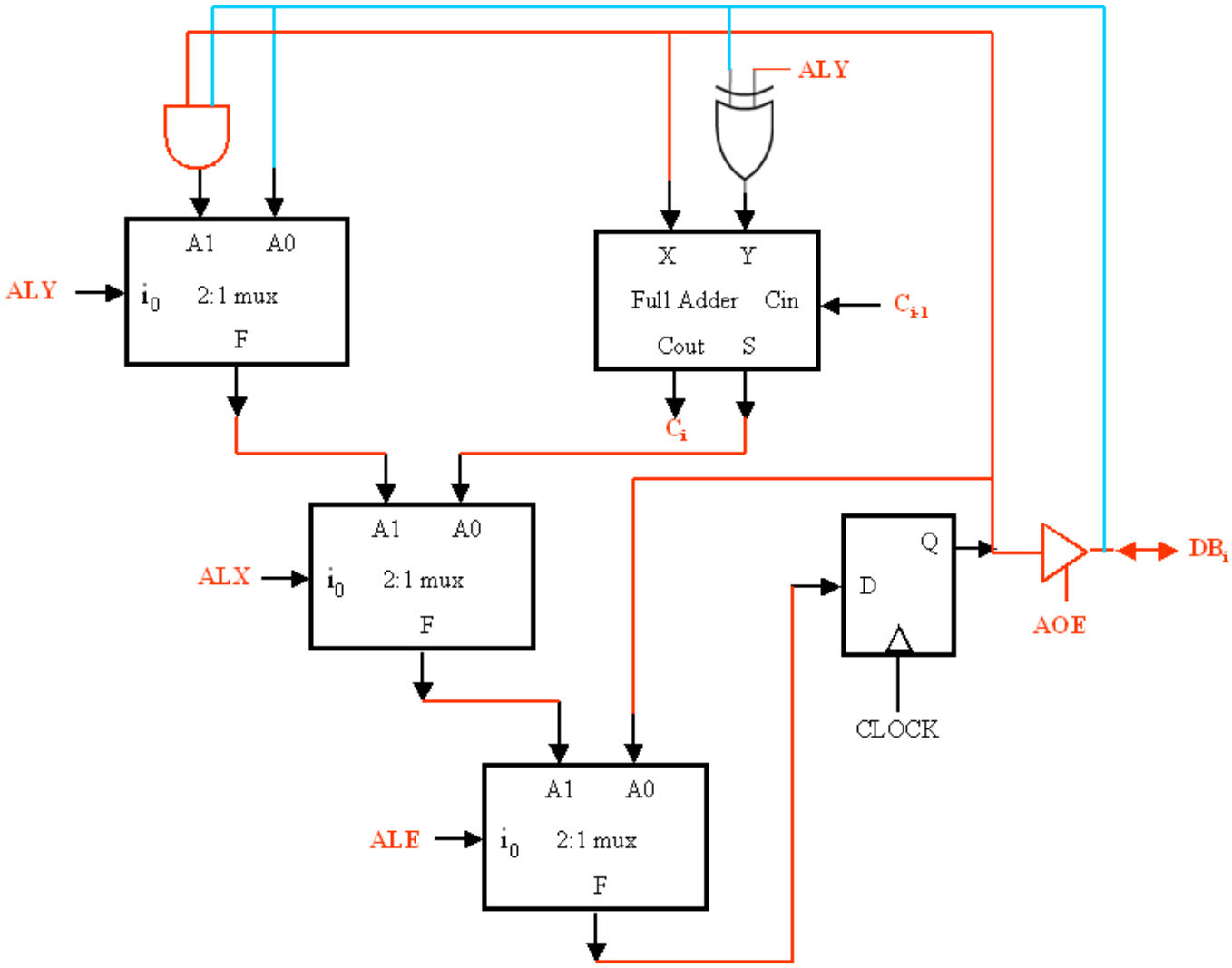
- **instruction register**
  - **basically an 8-bit data register, with tri-state outputs on the lower 5 (address) bits**
  - **upper 3 bits (opcode) output directly to instruction decoder and micro-sequencer**
  - **2 control signals**
    - **IRL – instruction register load enable**
    - **IRA – instruction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

```
MODULE ir
TITLE    'Instruction Register Module'
DECLARATIONS
CLOCK pin;
" IR4..IR0 connected to address bus
" IR7..IR5 supply opcode to IDMS
IR0..IR7 pin istype 'reg_D,buffer';
DB0..DB7 pin; " data bus
IRL pin; " IR load enable
IRA pin; " IR output on address bus enable

EQUATIONS
"                  retain state          load
[IR0..IR7].d = !IRL&[IR0..IR7].q # IRL&[DB0..DB7];
[IR0..IR7].clk = CLOCK;
[IR0..IR4].oe = IRA;
[IR5..IR7].oe = [1,1,1];
END
```

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

```
MODULE alu
TITLE 'ALU Module'
"   8-bit, 4-function ALU with bi-directional data bus
"
"   ADD:  (Q7..Q0) <- (Q7..Q0) + DB7..DB0
"   SUB:  (Q7..Q0) <- (Q7..Q0) - DB7..DB0
"   LDA:  (Q7..Q0) <-  DB7..DB0
"   AND:  (Q7..Q0) <- (Q7..Q0) & DB7..DB0
"   OUT:  Value in Q7..Q0 output on data bus DB7..DB0
"
"   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
"
```

- **ALU, continued…**
  - **block diagram of one bit**

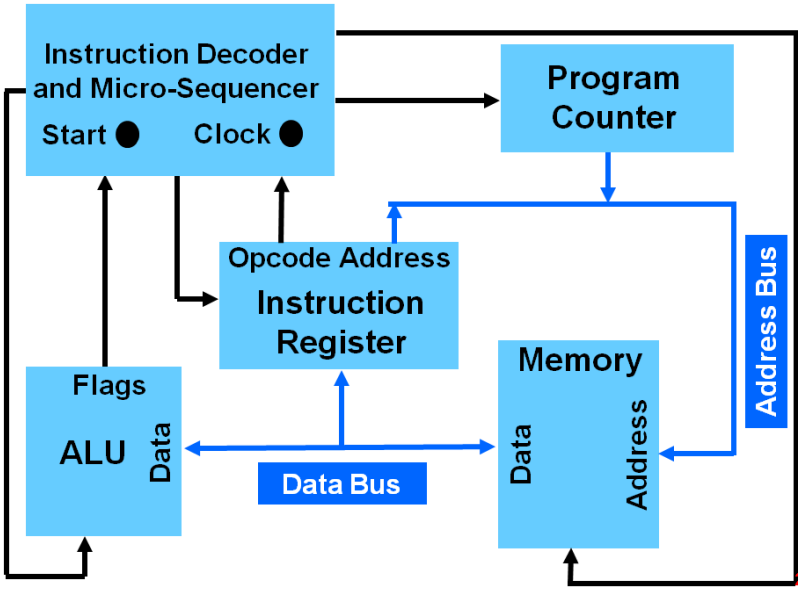


  - **flag (condition code) register**

```
" Flag register state equations

CF.d = !ALE&CF.q # ALE&(!ALX&(CY7 $ ALY) # ALX&CF.q);
CF.clk = CLOCK;
ZF.d = !ALE&ZF.q # ALE&(!ALU7&!ALU6&!ALU5&!ALU4&!ALU3&!ALU2&!ALU1&!ALU0);
ZF.clk = CLOCK;
NF.d = !ALE&NF.q # ALE&ALU7;
NF.clk = CLOCK;
VF.d = !ALE&VF.q # ALE&(!ALX&(CY7 $ CY6) # ALX&VF.q);
VF.clk = CLOCK;
END

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

- **instruction decoder and microsequencer**
  - **state machine that tells all the other state machines what to do ("whole enchilada")**
  - **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)**
  - **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**

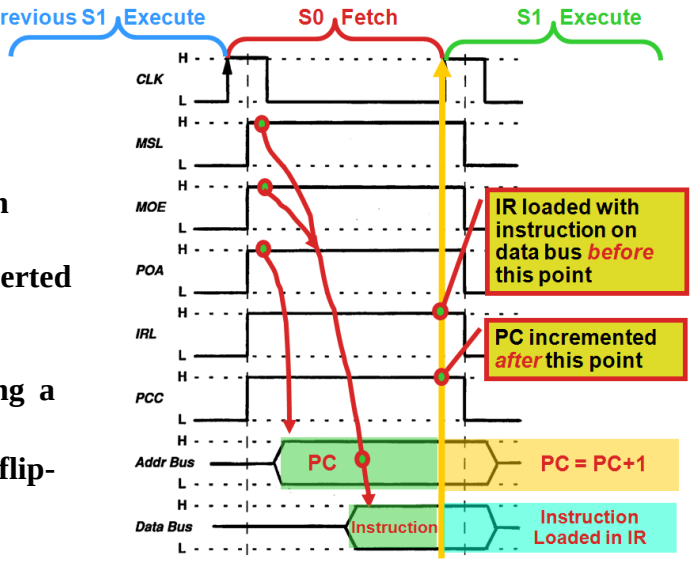
The synchronous fetch functions (IRL and PCC) will take place on the clock edge that causes the state counter to transition from the fetch state to the execute state



Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY
S0	—	H	H		H	H	H					
S1	LDA	H	H					H		H	H	
S1	STA	H		H				H	H			
S1	ADD	H	H					H		H		
S1	SUB	H	H					H		H		H
S1	AND	H	H					H		H	H	H
S1	HLT	L			L		L			L		

```
MODULE idms
TITLE    'Instruction Decoder and Microsequencer'
DECLARATIONS
CLOCK pin;
START pin; " asynchronous START pushbutton
OP0..OP2 pin; " opcode bits (input from IR5..IR7)
" State counter
SQ node istype 'reg_D,buffer';
" RUN/HLT state
RUN node istype 'reg_D,buffer';
" RUN/HLT state
RUN node istype 'reg_D,buffer';
" Memory control signals
MSL,MOE,MWE pin istype 'com';
" PC control signals
PCC,POA,ARS pin istype 'com';
" IR control signals
IRL,IRA pin istype 'com';
" ALU control signals (not using flags yet)
ALE,ALX,ALY,AOE pin istype 'com';

" Decoded opcode definitions
LDA = !OP2&!OP1&!OP0;  " LDA opcode = 000
STA = !OP2&!OP1& OP0;  " STA opcode = 001
ADD = !OP2& OP1&!OP0;  " ADD opcode = 010
SUB = !OP2& OP1& OP0;  " SUB opcode = 011
AND =  OP2&!OP1&!OP0;  " AND opcode = 100
HLT =  OP2&!OP1& OP0;  " HLT opcode = 101
" Decoded state definitions
S0 = !SQ.q; " fetch
S1 =  SQ.q; " execute
```

```
EQUATIONS

" State counter
SQ.d = RUN.q&!SQ.q; " if RUN negated, resets SQ
SQ.clk = CLOCK;
SQ.ar = START;      " start in fetch state

" Run/stop (equivalent of SR latch)
RUN.ap = START;  " start with RUN set to 1
RUN.clk = CLOCK;
RUN.d = RUN.q;
RUN.ar = S1&HLT;  " RUN is cleared when HLT executed

" System control equations
MSL = RUN.q&(S0 # S1&(LDA # STA # ADD # SUB # AND));
MOE = S0 # S1&(LDA # ADD # SUB # AND);
MWE = S1&STA;
ARS = START;
PCC = RUN.q&S0;
POA = S0;
IRL = RUN.q&S0;
IRA = S1&(LDA # STA # ADD # SUB # AND);
AOE = S1&STA;
ALE = RUN.q&S1&(LDA # ADD # SUB # AND);
ALX = S1&(LDA # AND);
ALY = S1&(SUB # AND);
END
```



- 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

Lecture Summary – Module 4-J

Simple Computer – Basic Extensions

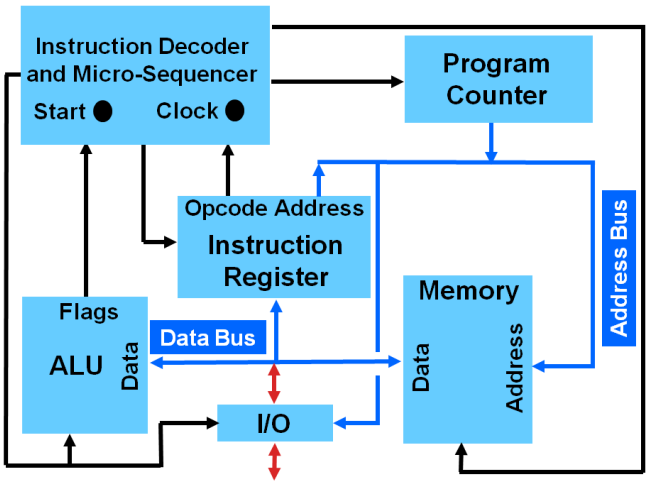
Reference: Meyer Supplemental Text, pp. 42-50

- **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**

```
MODULE alum
TITLE 'ALU Module - Modified for Shift Instructions'
"   8-bit, 4-function ALU with bi-directional data bus
"
"   LDA:  (Q7..Q0) <- DB7..DB0
"   LSR:  (Q7..Q0) <- 0 Q7 Q6 Q5 Q4 Q3 Q2 Q1, CF <- Q0
"   ASL:  (Q7..Q0) <- Q6 Q5 Q4 Q3 Q2 Q1 Q0 0, CF <- Q7
"   ASR:  (Q7..Q0) <- Q7 Q7 Q6 Q5 Q4 Q3 Q2 Q1, CF <- Q0
"   OUT:  Value in Q7..Q0 output on data bus DB7..DB0
"
"   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
```

```
DECLARATIONS
CLOCK pin;

ALE pin;
AOE pin;
ALX pin;
ALY pin;

ALU0..ALU7 node istype 'com';

DB0..DB7 pin istype 'reg_d,buffer';

CF pin istype 'reg_d,buffer';  " carry flag
VF pin istype 'reg_d,buffer';  " overflow flag
NF pin istype 'reg_d,buffer';  " negative flag
ZF pin istype 'reg_d,buffer';  " zero flag
EQUATIONS
"           LDA                  LSR              ASL              ASR
ALU0 = !ALX&!ALY&DB0.pin # !ALX&ALY&DB1.q # ALX&!ALY&  0   # ALX&ALY&DB1.q;
ALU1 = !ALX&!ALY&DB1.pin # !ALX&ALY&DB2.q # ALX&!ALY&DB0.q # ALX&ALY&DB2.q;
ALU2 = !ALX&!ALY&DB2.pin # !ALX&ALY&DB3.q # ALX&!ALY&DB1.q # ALX&ALY&DB3.q;
ALU3 = !ALX&!ALY&DB3.pin # !ALX&ALY&DB4.q # ALX&!ALY&DB2.q # ALX&ALY&DB4.q;
ALU4 = !ALX&!ALY&DB4.pin # !ALX&ALY&DB5.q # ALX&!ALY&DB3.q # ALX&ALY&DB5.q;
ALU5 = !ALX&!ALY&DB5.pin # !ALX&ALY&DB6.q # ALX&!ALY&DB4.q # ALX&ALY&DB6.q;
ALU6 = !ALX&!ALY&DB6.pin # !ALX&ALY&DB7.q # ALX&!ALY&DB5.q # ALX&ALY&DB7.q;
ALU7 = !ALX&!ALY&DB7.pin # !ALX&ALY&  0   # ALX&!ALY&DB6.q # ALX&ALY&DB7.q;
" Register bit and data bus control equations
[DB0..DB7].d = !ALE&[DB0..DB7].q # ALE&[ALU0..ALU7];
[DB0..DB7].clk = CLOCK;
[DB0..DB7].oe = AOE;
" Flag register state equations
CF.d = !ALE&CF.q #
  ALE&(!ALX&!ALY&CF.q # !ALX&ALY&DB0.q # ALX&!ALY&DB7.q # ALX&ALY&DB0.q);
"         LDA                 LSR              ASL              ASR
CF.clk = CLOCK;
ZF.d = !ALE&ZF.q # ALE&(!ALU7&!ALU6&!ALU5&!ALU4&!ALU3&!ALU2&!ALU1&!ALU0);
ZF.clk = CLOCK;
NF.d = !ALE&NF.q # ALE&ALU7;
NF.clk = CLOCK;
VF.d = !ALE&VF.q # ALE&VF.q;  " NOTE: NOT AFFECTED
VF.clk = CLOCK;
END
```

- **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, ABEL code for I/O module, modified system control table**

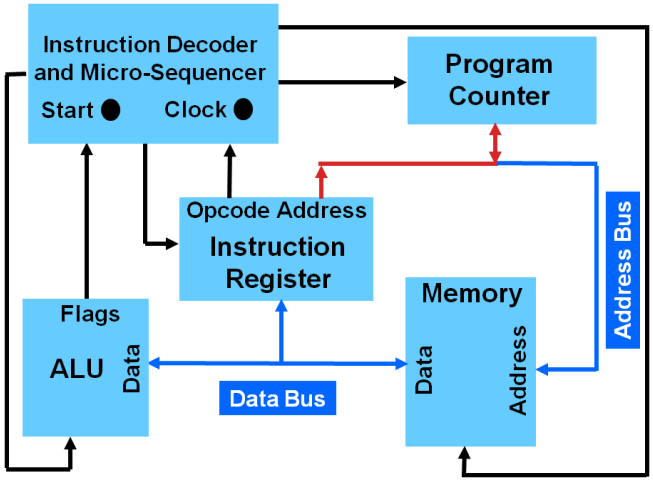



```
MODULE iol
TITLE    'Input/Output Port 00000 - With Latch'
DECLARATIONS

DB0..DB7 pin istype 'com';    " data bus
AD0..AD4 pin;                 " address bus
IN0..IN7 pin;                 " input port
OUT0..OUT7 pin istype 'com'; " output port
IOR pin; " Input port read
IOW pin; " Output port write
" Port select equation for port address 00000
PS = !AD4&!AD3&!AD2&!AD1&!AD0;
EQUATIONS
[DB0..DB7] = [IN0..IN7];
[DB0..DB7].oe = IOR&PS;
" Transparent latch for output port
[OUT0..OUT7] = !(IOW&PS)&[OUT0..OUT7] #
IOW&PS&[DB0..DB7];
END
```

Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	IOR	IOW
S0	—	H	H		H	H	H							
S1	LDA	H	H					H		H	H			
S1	STA	H		H				H	H					
S1	ADD	H	H					H		H				
S1	SUB	H	H					H		H		H		
S1	AND	H	H					H		H	H	H		
S1	HLT	L			L		L			L				
S1	IN							H		H	H		H	
S1	OUT							H	H					H

- **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**



- **ABEL code for modified PC (with "load from address bus" capability)**

```
MODULE pc
TITLE    'Program Counter with Load Capability'
DECLARATIONS
CLOCK pin;
PC0..PC4 pin istype 'reg_D,buffer';
PCC pin; " PC count enable
PLA pin; " PC load from address bus enable
POA pin; " PC output on address bus tri-state enable
ARS pin; " asynchronous reset (connected to START)
" Note: Assume PCC and PLA are mutually exclusive
EQUATIONS
"          retain state       load          count up by 1
PC0.d = !PCC&!PLA&PC0.q # PLA&PC0.pin # PCC&!PC0.q;
PC1.d = !PCC&!PLA&PC1.q # PLA&PC1.pin # PCC&(PC1.q $ PC0.q);
PC2.d = !PCC&!PLA&PC2.q # PLA&PC2.pin # PCC&(PC2.q $
(PC1.q&PC0.q));
PC3.d = !PCC&!PLA&PC3.q # PLA&PC3.pin # PCC&(PC3.q $
(PC2.q&PC1.q&PC0.q));
PC4.d = !PCC&!PLA&PC4.q # PLA&PC4.pin # PCC&(PC4.q $
(PC3.q&PC2.q&PC1.q&PC0.q));
[PC0..PC4].oe = POA;
[PC0..PC4].ar = ARS;
[PC0..PC4].clk = CLOCK;
END
```

- **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	LDA	H	H					H		H	H		
S1	STA	H		H				H	H				
S1	ADD	H	H					H		H			
S1	SUB	H	H					H		H		H	
S1	AND	H	H					H		H	H	H	
S1	HLT	L			L		L			L			
S1	JMP							H					H
S1	JZF							ZF					ZF

Lecture Summary – Module 4-K

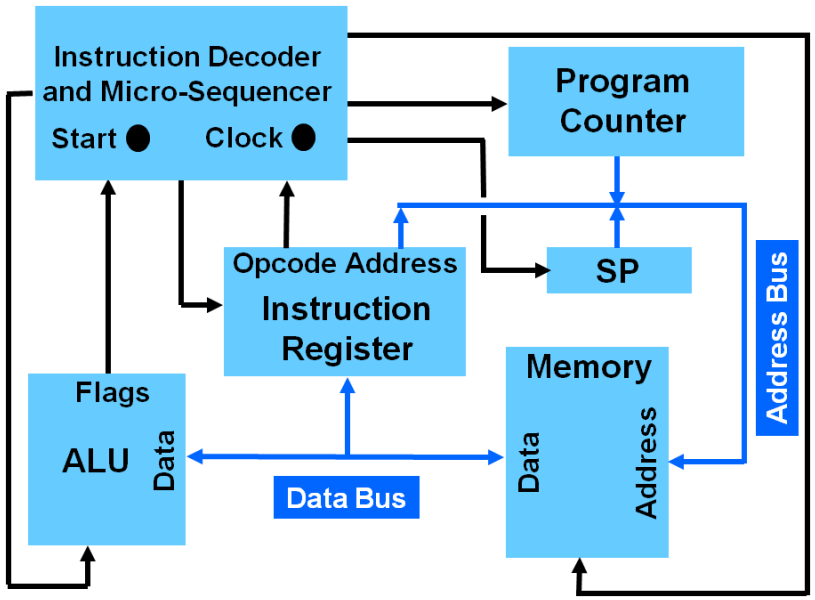
Simple Computer – Advanced Extensions

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)**

```
MODULE idmsr
TITLE 'Instruction Decoder and Microsequencer with Multi-Execution States'
DECLARATIONS
CLOCK pin;
START pin;    " asynchronous START pushbutton
OP0..OP2 pin; " opcode bits (input from IR5..IR7)
" State counter
SQA node istype 'reg_D,buffer'; " low bit of state counter
SQB node istype 'reg_D,buffer'; " high bit of state counter
" Synchronous state counter reset
RST node istype 'com';
" RUN/HLT state
RUN node istype 'reg_D,buffer';
" Memory control signals
MSL,MOE,MWE pin istype 'com';
" PC control signals
PCC,POA,ARS pin istype 'com';
" IR control signals
IRL,IRA pin istype 'com';
" ALU control signals
ALE,ALX,ALY,AOE pin istype 'com';
" Decoded opcode definitions
LDA = !OP2&!OP1&!OP0;  " opcode 000
STA = !OP2&!OP1& OP0;  " opcode 001
ADD = !OP2& OP1&!OP0;  " opcode 010
SUB = !OP2& OP1& OP0;  " opcode 011
AND =  OP2&!OP1&!OP0;  " opcode 100
HLT =  OP2&!OP1& OP0;  " opcode 101
" Decoded state definitions
S0 = !SQB&!SQA; " fetch state
S1 = !SQB& SQA; " first execute state
S2 =  SQB&!SQA; " second execute state
S3 =  SQB& SQA; " third execute state
EQUATIONS
" State counter
SQA.d = !RST & RUN.q & !SQA.q;          " if RUN negated or RST asserted,
SQB.d = !RST & RUN.q & (SQB.q $ SQA.q); " state counter is reset
SQA.clk = CLOCK;
SQB.clk = CLOCK;
SQA.ar = START;       " start in fetch state
SQB.ar = START;
" Run/stop (equivalent of SR latch)
RUN.ap = START;      " start with RUN set to 1
RUN.clk = CLOCK;
RUN.d = RUN.q;
RUN.ar = S1&HLT;     " RUN is cleared when HLT executed
" System control equations (for base machine)
; (others same as before)
RST = S1&(LDA # STA # ADD # SUB # AND)  " assert on final execute state of each instruction
END
```

- **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)**



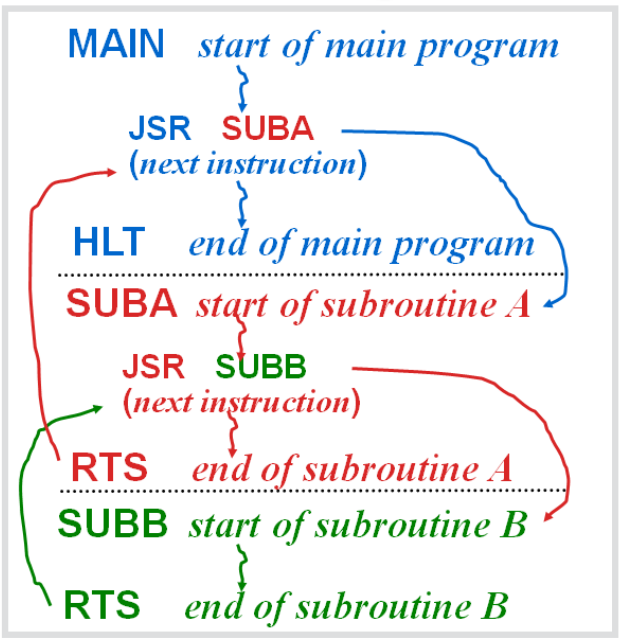

```
MODULE sp
TITLE    'Stack Pointer'
DECLARATIONS
CLOCK pin;
SP0..SP4 pin istype 'reg_D,buffer';
SPI pin; " SP increment enable
SPD pin; " SP decrement enable
SPA pin; " SP output on address bus tri-state enable
ARS pin; " asynchronous reset (connected to START)
" Note: Assume SPI and SPD are mutually exclusive
EQUATIONS
"          retain state      increment/decrement
SP0.d = !SPI&!SPD&SP0.q # SPI&!SP0.q
                        # SPD&!SP0.q;
SP1.d = !SPI&!SPD&SP1.q # SPI&(SP1.q$SP0.q)
                        # SPD&(SP1.q$!SP0.q);
SP2.d = !SPI&!SPD&SP2.q # SPI&(SP2.q$(SP1.q&SP0.q))
                        # SPD&(SP2.q$(!SP1.q&!SP0.q));
SP3.d = !SPI&!SPD&SP3.q # SPI&(SP3.q$(SP2.q&SP1.q&SP0.q))
                        # SPD&(SP3.q$(!SP2.q&!SP1.q&!SP0.q));
SP4.d = !SPI&!SPD&SP4.q # SPI&(SP4.q$(SP3.q&SP2.q&SP1.q&SP0.q))
                        # SPD&(SP4.q$(!SP3.q&!SP2.q&!SP1.q&!SP0.q));
[SP0..SP4].oe = SPA;
[SP0..SP4].ar = ARS;
[SP0..SP4].clk = CLOCK;
END
```

- **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	LDA	H	H					H		H	H					H
S1	STA	H		H				H	H							H
S1	ADD	H	H					H		H						H
S1	SUB	H	H					H		H		H				H
S1	AND	H	H					H		H	H	H				H
S1	HLT	L			L		L			L						
S1	PSH													H		
S1	POP	H	H							H	H		H		H	H
S2	PSH	H		H					H						H	H

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



- **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**

```
MODULE pcr
TITLE    'Program Counter with Data Bus Interface'
DECLARATIONS
CLOCK pin;
PC0..PC4 node istype 'reg_D,buffer'; " PC register bits
AB0..AB4 pin; " address bus (5-bits wide)
DB0..DB7 pin; " data bus (8-bits wide)
PCC pin; " PC count enable
PLA pin; " PC load from address bus enable
PLD pin; " PC load from data bus enable
POA pin; " PC output on address bus tri-state enable
POD pin; " PC output on data bus tri-state enable
ARS pin; " asynchronous reset (connected to START)
" Note: Assume PCC, PLA, and PLD are mutually exclusive
EQUATIONS
"          retain state       load from AB  load from DB   increment
PC0.d = !PCC&!PLA&!PLD&PC0.q # PLA&AB0.pin # PLD&DB0.pin # PCC&!PC0.q;
PC1.d = !PCC&!PLA&!PLD&PC1.q # PLA&AB1.pin # PLD&DB1.pin # PCC&(PC1.q$PC0.q);
PC2.d = !PCC&!PLA&!PLD&PC2.q # PLA&AB2.pin # PLD&DB2.pin # PCC&(PC2.q$(PC1.q&PC0.q));
PC3.d = !PCC&!PLA&!PLD&PC3.q # PLA&AB3.pin # PLD&DB3.pin # PCC&(PC3.q$(PC2.q&PC1.q&PC0.q));
PC4.d = !PCC&!PLA&!PLD&PC4.q # PLA&AB4.pin # PLD&DB4.pin #
PCC&(PC4.q$(PC3.q&PC2.q&PC1.q&PC0.q));
[AB0..AB4] = [PC0..PC4].q;
[DB0..DB4] = [PC0..PC4].q;
" Output logic zero on upper 3-bits of data bus
[DB5..DB7] = 0;
[AB0..AB4].oe = POA;
[DB0..DB7].oe = POD;
[PC0..PC4].ar = ARS;
[PC0..PC4].clk = CLOCK;
END
```

Dec. State	Instr. Mnem.	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	PLA	POD	PLD	SPI	SPD	SPA	RST
S0	—	H	H		H	H	H												
S1	LDA	H	H					H		H	H								H
S1	STA	H		H				H	H										H
S1	ADD	H	H					H		H									H
S1	SUB	H	H					H		H		H							H
S1	AND	H	H					H		H	H	H							H
S1	HLT	L			L		L			L									
S1	JSR																H		
S1	RTS	H	H												H	H		H	H
S2	JSR	H		H										H				H	
S3	JSR							H					H						H