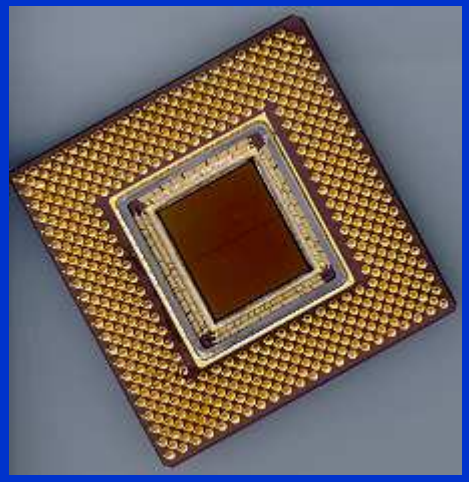


Introduction to Digital System Design

Module 4
Arithmetic and Computer Logic Circuits

Glossary of Common Terms

- **RADIX – a commonly-used signed number notation (also called 2's complement)**
- **OPERAND – a binary number involved in an arithmetic or logical operation**
- **HALF ADDER – logic circuit that adds two binary bits to produce carry and sum outputs**
- **FULL ADDER – logic circuit that adds three binary bits to produce carry and sum outputs**
- **ADDER/SUBTRACTOR – logic circuit that adds and subtracts pairs of binary operands**
- **MAGNITUDE COMPARATOR – logic circuit that determines which binary operand is greater/less than a second binary operand**

Glossary of Common Terms

- **CARRY LOOK-AHEAD – a means of generating the carry functions needed for addition in parallel**
- **MULTIPLIER – logic circuit that multiplies pairs of binary operands**
- **COMPUTER – device that sequentially executes a stored program**
- **PROGRAM – a series of instructions that direct the processing activity of a computer**
- **INSTRUCTION – a unit of processing activity ("line of code") executed by a computer**
- **OPCODE – the "operation code" field of an instruction**

Glossary of Common Terms

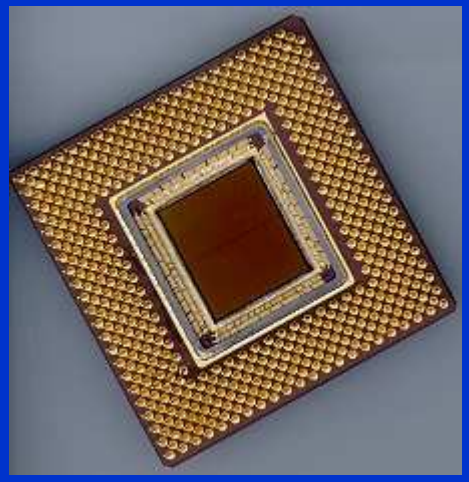
- **MEMORY – array of D latches used to store instructions, operands, and results**
- **PROGRAM COUNTER – register that points to the next instruction to execute**
- **INSTRUCTION REGISTER – register used to "stage" the instruction fetched from memory**
- **ALU – arithmetic logic unit, performs arithmetic and logic operations on binary operands**
- **INSTRUCTION DECODER & MICROSEQUENCER – state machine that orchestrates the activities of a computer's functional blocks**

Glossary of Common Terms

- **MICROSEQUENCE – the "minute" phases of instruction processing by a computer**
- **TRANSFER OF CONTROL – continue execution of program at a location different than the next consecutive instruction**
- **I/O – data input and output operations performed by a computer**
- **STACK – last in, first out data structure used to support expression evaluation and subroutine linkage**
- **STACK POINTER – register used to point to the top stack item (or next available location)**

Module 4

- **Learning Outcome: "An ability to analyze and design arithmetic and computer logic circuits"**
  - **A. Signed Number Notation**
  - **B. Radix Addition and Subtraction**
  - **C. Adder, Subtractor, and Comparator Circuits**
  - **D. Carry Look-Ahead Adder Circuits**
  - **E. Multiplier Circuits**
  - **F. BCD Adder Circuits**
  - **G. Simple Computer Top-Down Specification**
  - **H. Simple Computer Instruction Execution Tracing**
  - **I. Simple Computer Bottom-Up Realization**
  - **J. Simple Computer Basic Extensions**
  - **K. Simple Computer Advanced Extensions**



Introduction to Digital System Design

Module 4-A

Signed Number Notation

Reading Assignment:

DDPP 4th Ed., pp. 34-39

Learning Objectives:

- **Compare and contrast three different signed number notations: sign and magnitude, diminished radix, and radix**
- **Convert a number from one signed notation to another**
- **Describe how to perform sign extension of a number represented using any of the three notation schemes**

Outline

- Overview
- Signed number notation
  - Sign and magnitude
  - Diminished radix
  - Radix
  - Comparison chart
- Simplifications for binary numbers
- Sign extension

Overview

- **In order to represent positive and negative numbers as a series of digits without "+" and "–" signs, various *signed number notations* have been devised**
- **We will discuss the three most commonly used signed number notations:**
  - ***sign and magnitude*** **(SM)**
  - ***diminished radix*** **(DR)**
  - ***radix*** **(R)**

Sign and Magnitude

- **The original signed number convention employed by "vacuum tube vintage digijocks" was *sign and magnitude* notation**
- **Here the *left-most digit* (or "sign bit") indicates whether the number is positive or negative:**
  - **"0" → positive**
  - **"R–1" → negative (where R is the *radix* or *base* of the number)**

Sign and Magnitude

- **Examples:**

  $(+123)_{10} = SM(0123)_{10}$

  $(-123)_{10} = SM(9123)_{10}$

  $(+144)_5 = SM(0144)_5$

  $(-144)_5 = SM(4144)_5$

$SM(0123)_{10}$ and $SM(9123)_{10}$ are referred to as the *sign and magnitude complements* of one another

Sign and Magnitude

- **Negation Method: If N is a number in base R with sign digit n_S, such that**

  $(N)_R = n_S n_3 n_2 n_1 n_0$

  then

  $-(N)_R = (R-1-n_S) n_3 n_2 n_1 n_0$

- **Examples:**

  $(+1101)_2 = SM(01101)_2$

  $(-1101)_2 = SM(11101)_2$

Diminished Radix

- **The negation (or complement) of a number represented in *diminished radix* (DR) notation can be found by subtracting each digit (including the sign digit) from (R–1), i.e., the "radix minus one" or the "radix diminished by one"**
- **<u>Examples</u>:**

  $(+123)_{10} = DR(0123)_{10}$

  $(-123)_{10} = (9999 - 0123)_{10} = DR(9876)_{10}$

$DR(0123)_{10}$ and $DR(9876)_{10}$ are referred to as the *diminished radix complements* of one another.

Diminished Radix

- **Negation Method: If N is a number in base R,**

$$-(N)_R = (R^n - 1)_R - (N)_R$$

- **Examples:**

$$(+1101)_2 = DR(01101)_2$$

$$(-1101)_2 = DR(10010)_2$$

Note that *positive* DR numbers have the same representation as *positive* SM numbers; *negative* DR and SM numbers, however, have different representations

Radix

- **The negation (or complement) of a number represented in *radix* (R) notation can be found by *adding one* to the least significant position of the *diminished radix negation* of that number**
- **<u>Examples</u>:**

  $(+123)_{10} = R(0123)_{10}$

  $(-123)_{10} = (9999 - 0123 + 1)_{10} = R(9877)_{10}$

$R(0123)_{10}$ and $R(9877)_{10}$ are referred to as the *radix complements* of one another

Radix

- **Negation Method: If N is a number in base R,**

$$-(N)_R = (R^n)_R - (N)_R$$

- **Examples:**

$$(+1101)_2 = R(01101)_2$$

$$(-1101)_2 = R(10011)_2$$

Note that *positive* R, DR, and SM numbers all have the same representation; *negative* R, DR, and SM numbers, however, all have different representations

Comparison of Signed Number Notations

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

Comparison of Signed Number Notations

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

Radix has no "negative zero"

Comparison of Signed Number Notations

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	
–2	110	101	
–3	111	100	
–4	—	—	

All positive numbers identical

Radix has no "negative zero"

Comparison of Signed Number Notations

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

Comparison of Signed Number Notations

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

Simplifications for Binary

- **When finding the negations (complements) of binary (base 2) numbers, the methods simplify as follows:**
  - **SM: complement the sign position**
  - **DR (also called *1's complement*): complement each position**
  - **R (also called *2's complement*):**
    - **add 1 to the DR complement *-or-***
    - ***scan number* from *right* to *left; complement* each position to the *left* of the first "1" encountered**

Practice

- If $(N)_2$ = SM$(01100)_2$, find $-(N)_2$

  SM$(11100)_2$

- If $(N)_2$ = DR$(01100)_2$, find $-(N)_2$

  DR$(10011)_2$

- If $(N)_2$ = R$(01100)_2$, find $-(N)_2$

  R$(10100)_2$

Sign Extension

- **Sometimes signed numbers of different length (number of bits) need to be added together – here, the "shorter" number needs to be "padded" with *leading digits* to make it the two numbers the same length**
- **The rules for padding signed numbers with leading digits are as follows:**
  - **SM: insert as many zeroes as needed to the *right* of the sign position**
  - **DR & R: *replicate the sign digit* as many times as needed**

Practice

- Extend $SM(09345)_{10}$ to 8 digits

  $SM(00009345)_{10}$

- Extend $DR(76500)_{8}$ to 8 digits

  $DR(77776500)_{8}$

- Extend $R(01100)_{2}$ to 8 digits

  $R(00001100)_{2}$

- Extend $R(11100)_{2}$ to 8 digits

  $R(11111100)_{2}$

Comparison/Observations

- **SM and DR notation have a *balanced set* of positive and negative numbers, and have *two representations* for zero**
- **R notation has an *unbalanced* set of positive and negative numbers (there is an "extra negative number"), and has a *single* representation for zero**
- **99% of all computers use *radix* notation; our discussion on addition and subtraction will therefore focus on *radix arithmetic* (we will assume a prefix of "R" on all numbers subsequently used)**

Clicker Quiz

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:**

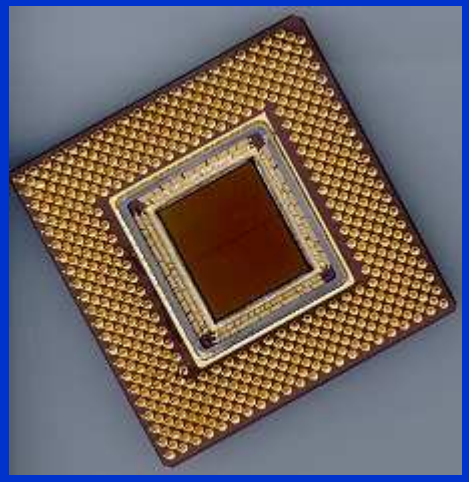
   A. **SM $(10101)_2$**

   B. **SM $(01010)_2$**

   C. **SM $(11010)_2$**

   D. **SM $(11011)_2$**

   E. **none of the above**



Introduction to Digital System Design

Module 4-B
Radix Addition and Subtraction

Reading Assignment:

DDPP 4th Ed., pp. 39-43

Learning Objectives:

- **perform radix addition and subtraction**
- **describe the various conditions of interest following an arithmetic operation: overflow, carry/borrow, negative, zero**

Outline

- **Radix Addition**
- **Overflow Detection**
- **Radix Subtraction**

Radix Arithmetic – Addition

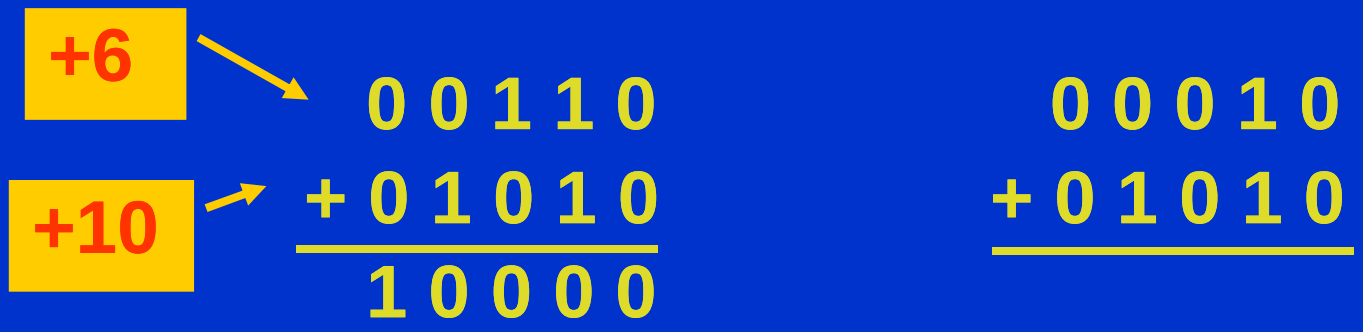
- **Method: Add all digits, including the sign digits; ignore any carry out of the sign position**
- **Problem: Since we are working with numbers of fixed length, the result of an addition can yield a number which is *too large* to represent in the same number of digits – this error condition is called *overflow***
- **Important: When overflow occurs, there is *no valid numeric result***

Overflow Detection

- **Summarization: Overflow occurs if two positive numbers are added and a negative result is obtained, or if two negative numbers are added and a positive result is obtained (or, if 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**

Radix Arithmetic – Addition

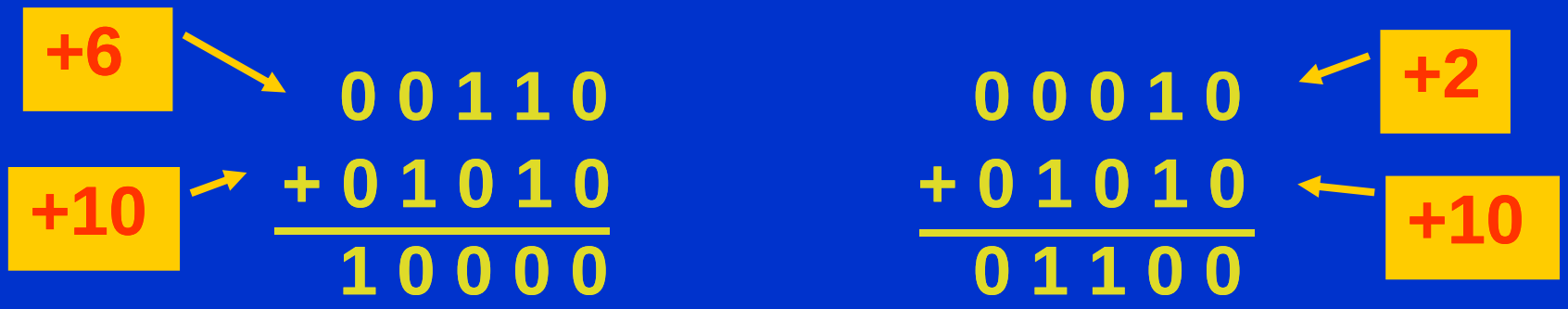
- **Examples: (all numbers are binary)**



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

Radix Arithmetic – Addition

- **Examples: (all numbers are binary)**

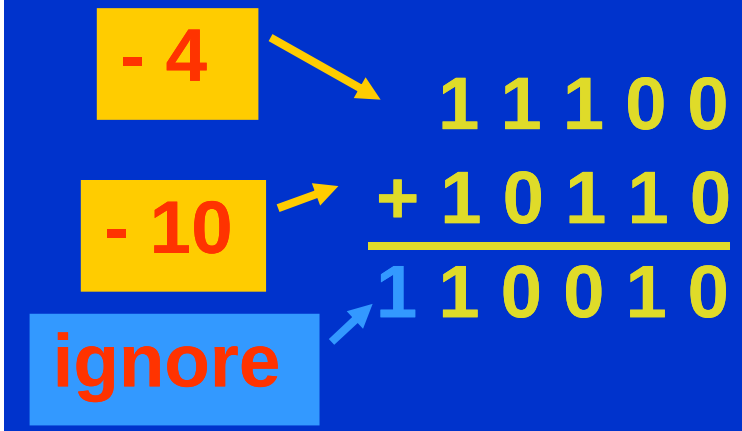


Here, added two *positive* numbers, but got a *negative* result → OVERFLOW

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

Radix Arithmetic – Addition

- **Examples: (all numbers are binary)**

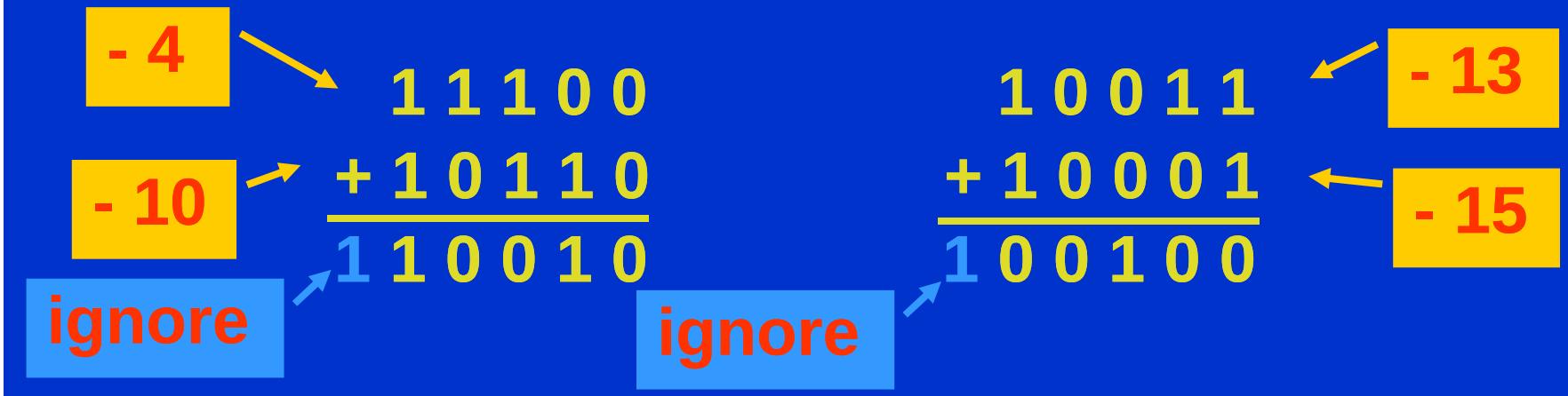


1 0 0 1 1
+ 1 0 0 0 1

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

Radix Arithmetic – Addition

- **Examples: (all numbers are binary)**

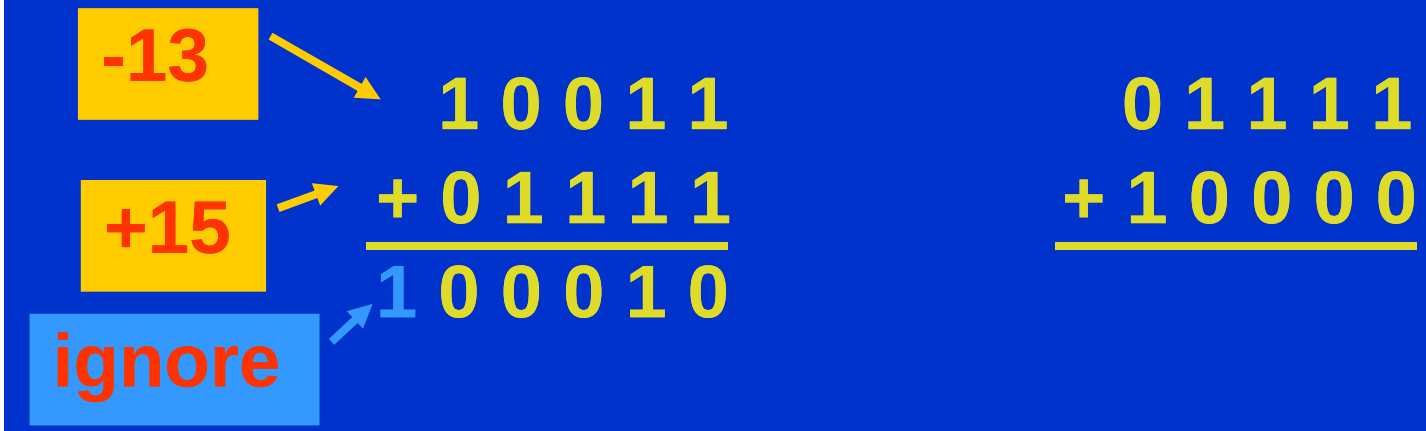


Here, added two *negative* numbers, and got a *negative* result (-14) → OK!

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

Radix Arithmetic – Addition

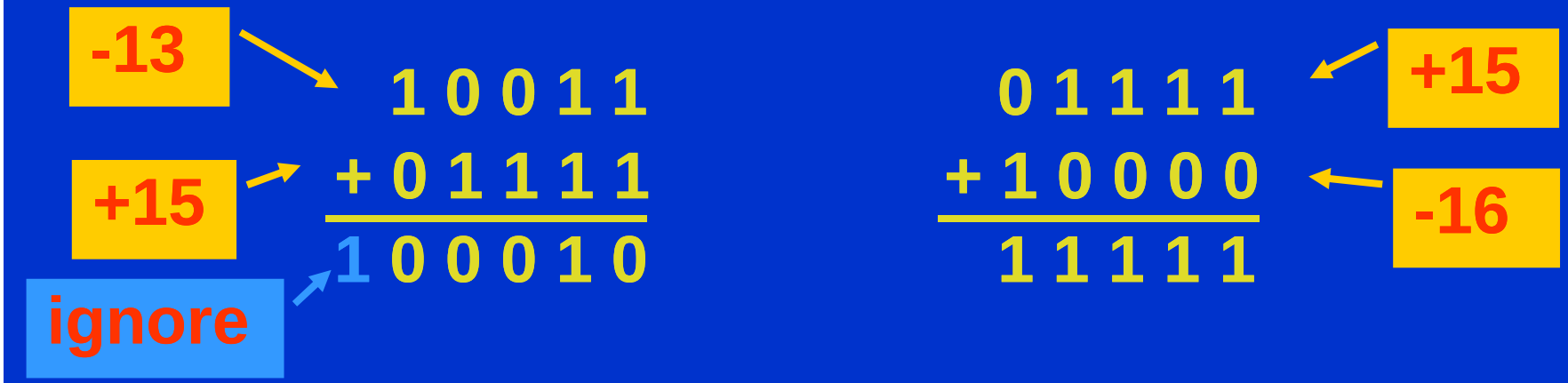
- **Examples: (all numbers are binary)**



Here, added numbers of *opposite sign* → overflow *cannot* occur (result is +2)

Radix Arithmetic – Addition

- **Examples: (all numbers are binary)**



Here, added numbers of *opposite sign* → overflow *cannot* occur (result is +2)

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

Radix Arithmetic – Subtraction

- **Method: Take the radix complement of the subtrahend and ADD; the same rules for overflow apply**
- **Examples:**

$$\begin{array}{r} 0\,1\,0\,1\,1 \\ -\;0\,1\,1\,0\,0 \\ \hline \end{array}$$

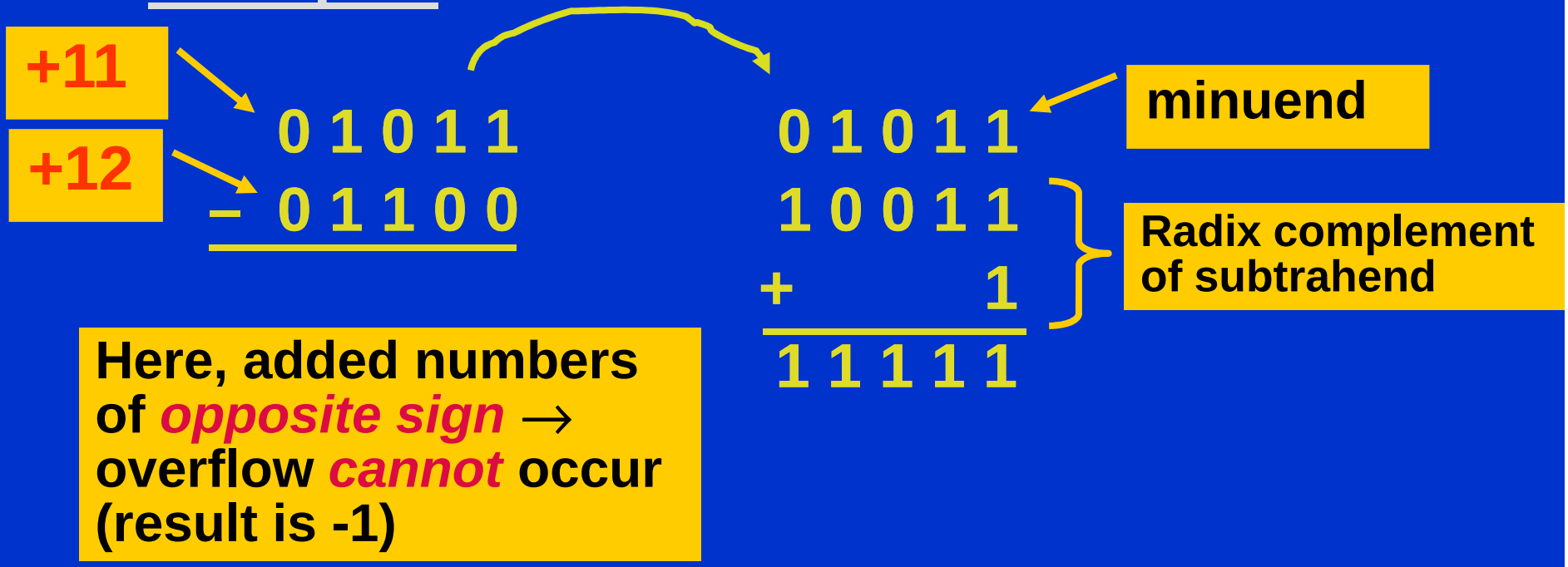
Why does this work?

Examples: 5 – (+3) = 5 + (–3) = 2

9 – (–13) = 9 + (+13) = 22

Radix Arithmetic – Subtraction

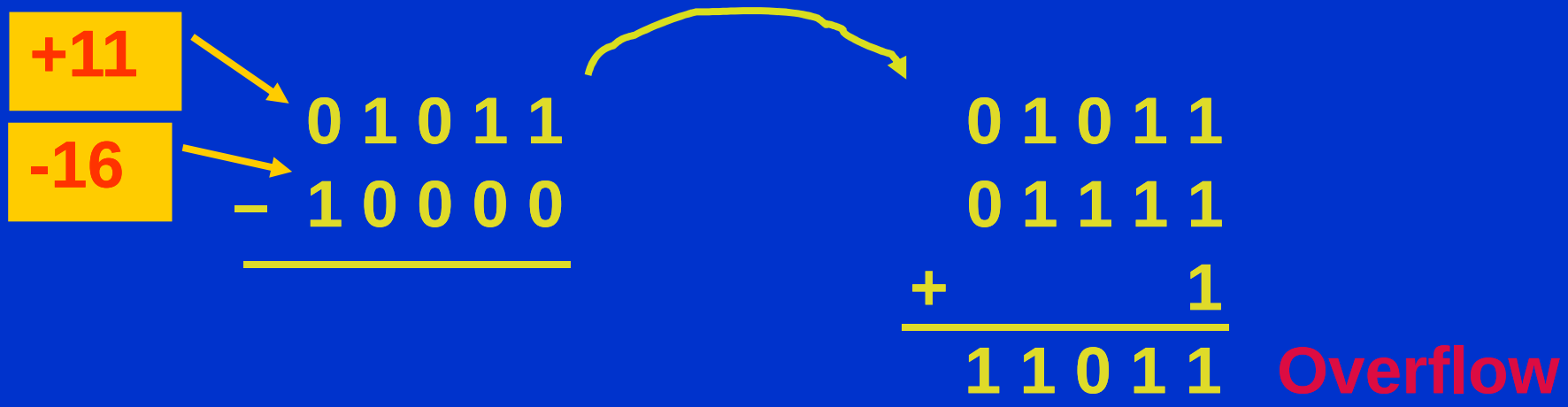
- **Method: Take the radix complement of the subtrahend and ADD; the same rules for overflow apply**
- **Examples:**



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

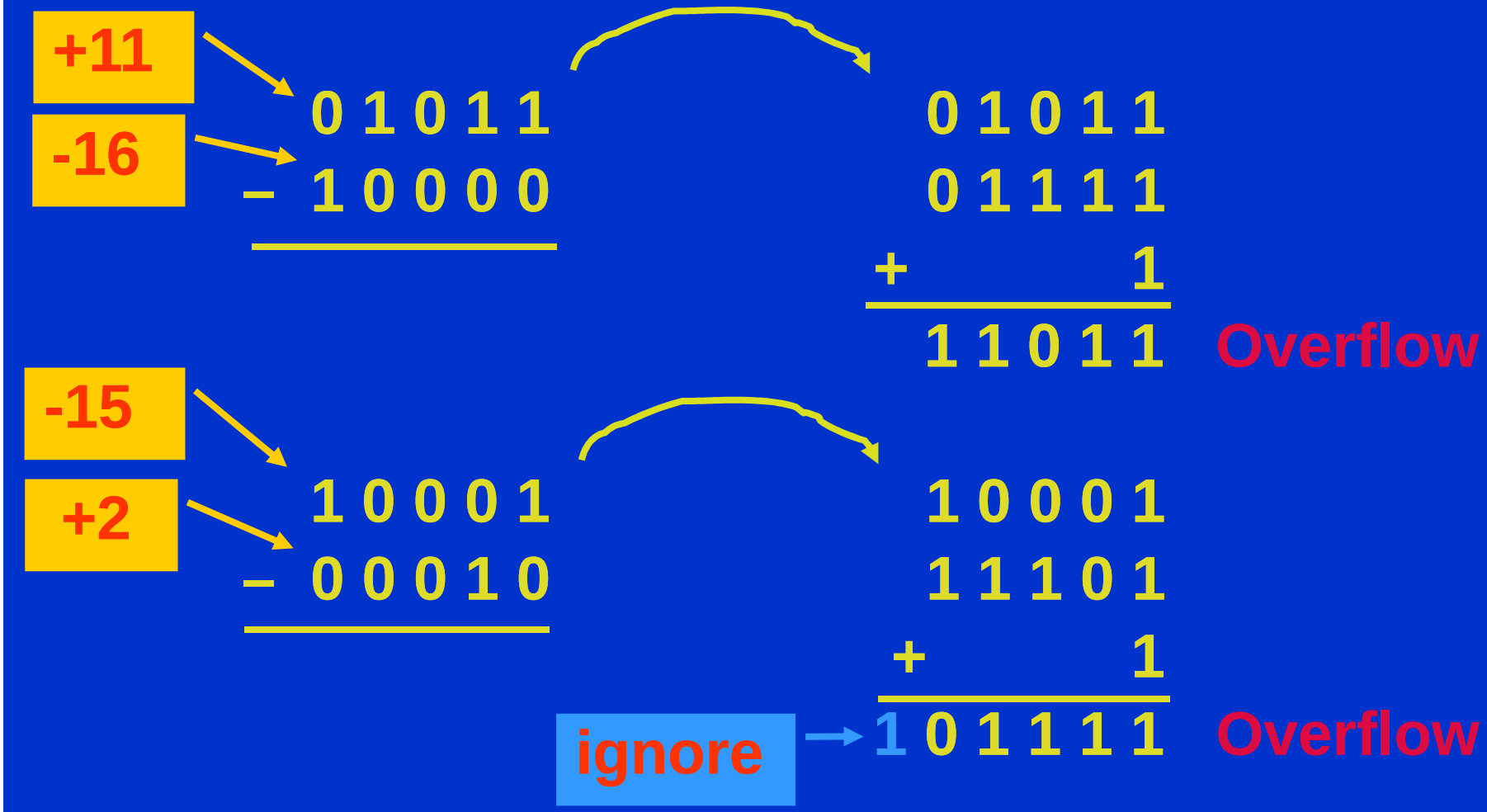
Radix Arithmetic – Subtraction

- **Examples:**



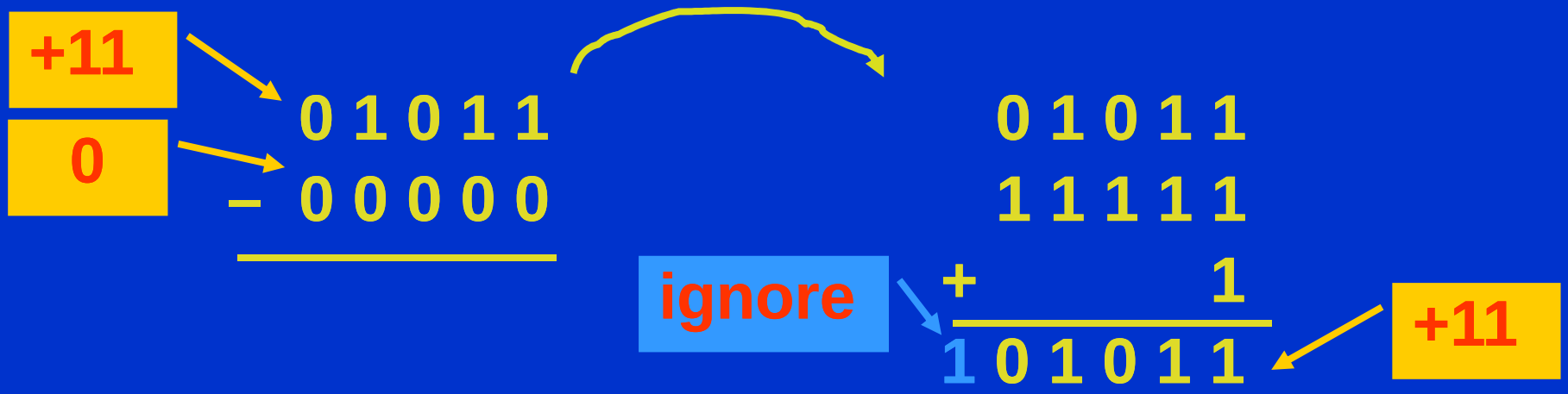
Radix Arithmetic – Subtraction

- **Examples:**



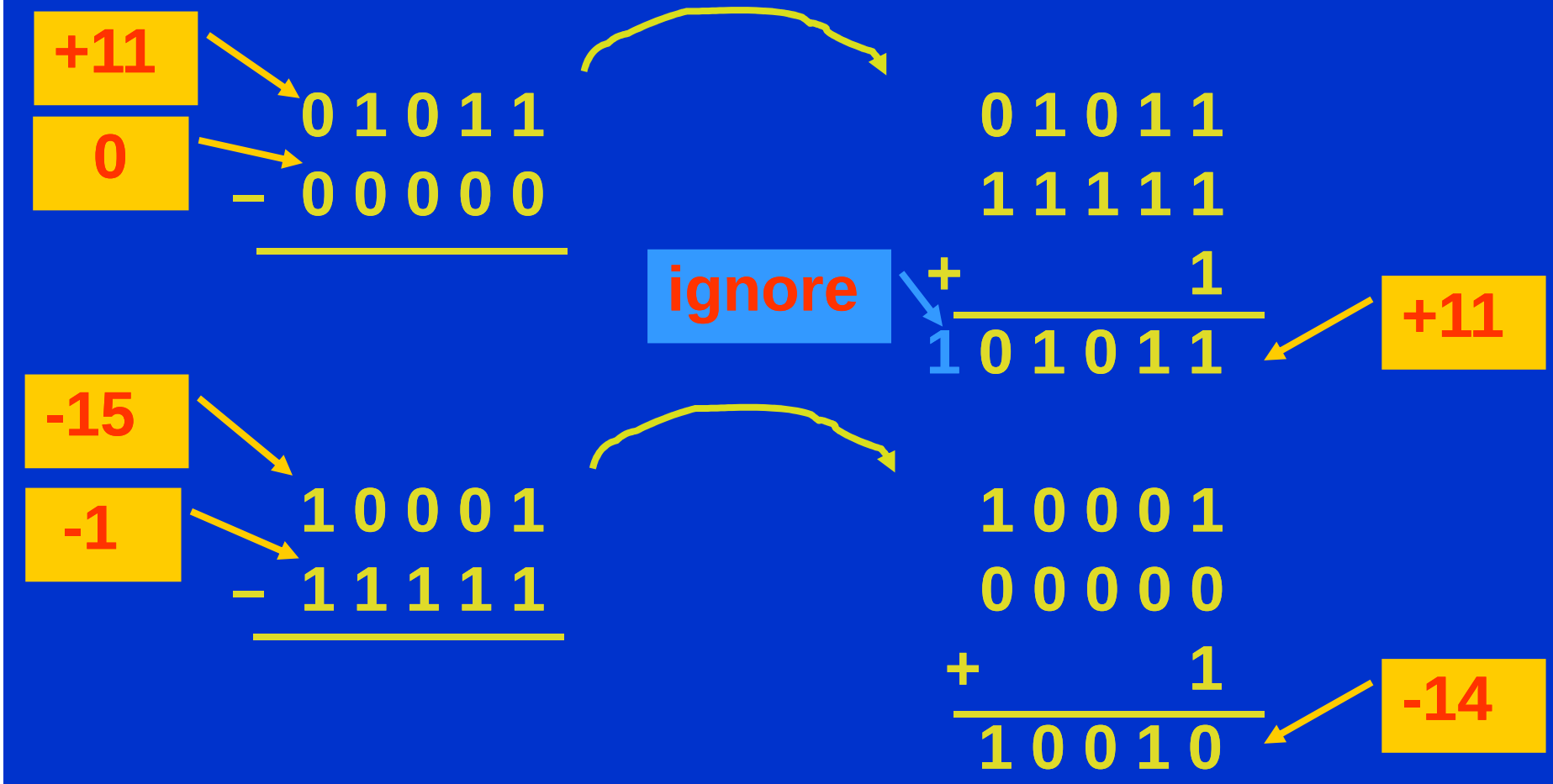
Radix Arithmetic – Subtraction

- **Examples:**



Radix Arithmetic – Subtraction

- **Examples:**

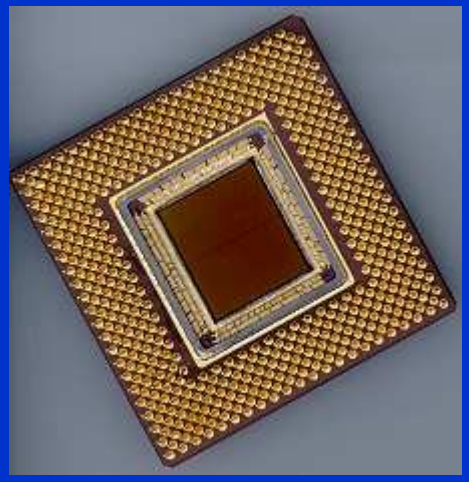


Clicker Quiz

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



Introduction to Digital System Design

Module 4-C

Adder, Subtractor, and Comparator Circuits

Reading Assignment:

DDPP 4th Ed., pp. 458-466, 474-478

Learning Objectives:

- **Describe the operation of a half-adder and write equations for its sum (S) and carry (C) outputs**
- **Describe the operation of a full adder and write equations for its sum (S) and carry (C) outputs**
- **Design a "population counting" or "vote counting" circuit using an array of half-adders and/or full-adders**
- **Design an N-digit radix adder/subtractor circuit with condition codes**
- **Design a (signed or unsigned) magnitude comparator circuit that determines if A=B, A<B, or A>B**

Outline

- Overview
- Half Adders
- Full Adders
- Radix Adder/Subtractors
- Comparators

Overview

- **Addition is the most commonly performed arithmetic operation in digital systems**
- **An *adder* combines two arithmetic operands using the addition rules described previously**
- **The same addition rules (and circuits) are used for both *signed* (two's complement) and *unsigned* numbers**
- **Subtraction can be performed by taking the *complement* of the subtrahend and adding it to the minuend**

Half Adders

- A half adder is used to add two binary digits, X_i and Y_i, to form a sum digit, S_i, and a carry digit, C_i

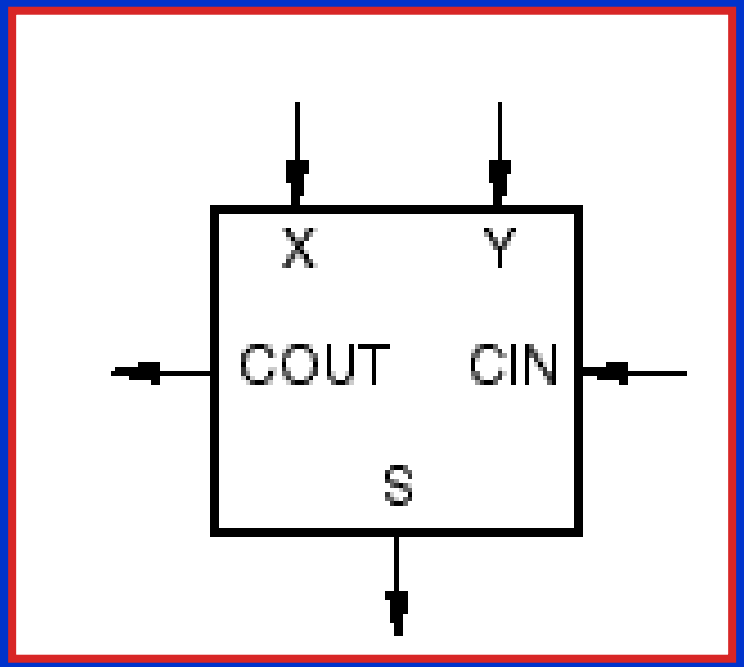
X_i	Y_i	C_i	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S_i = X_i \oplus Y_i$$

$$C_i = X_i \bullet Y_i$$

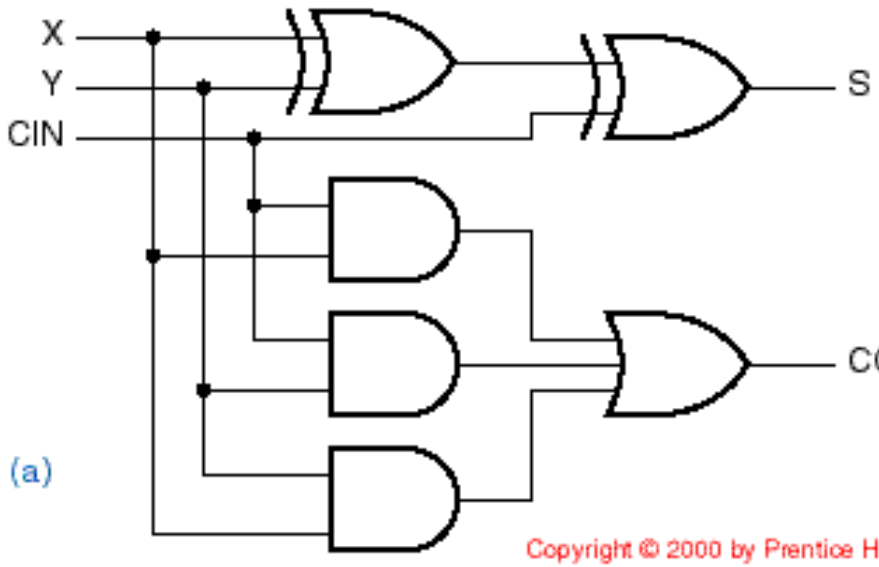
Full Adders

- **A full adder is used to add three binary digits, X_i, Y_i, C_{i-1} (where C_{i-1} is usually the carry *in* from a previous stage), to form a sum digit, S_i, and a carry *out* digit, C_i**



Full Adders

X_i	Y_i	C_{i-1}	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

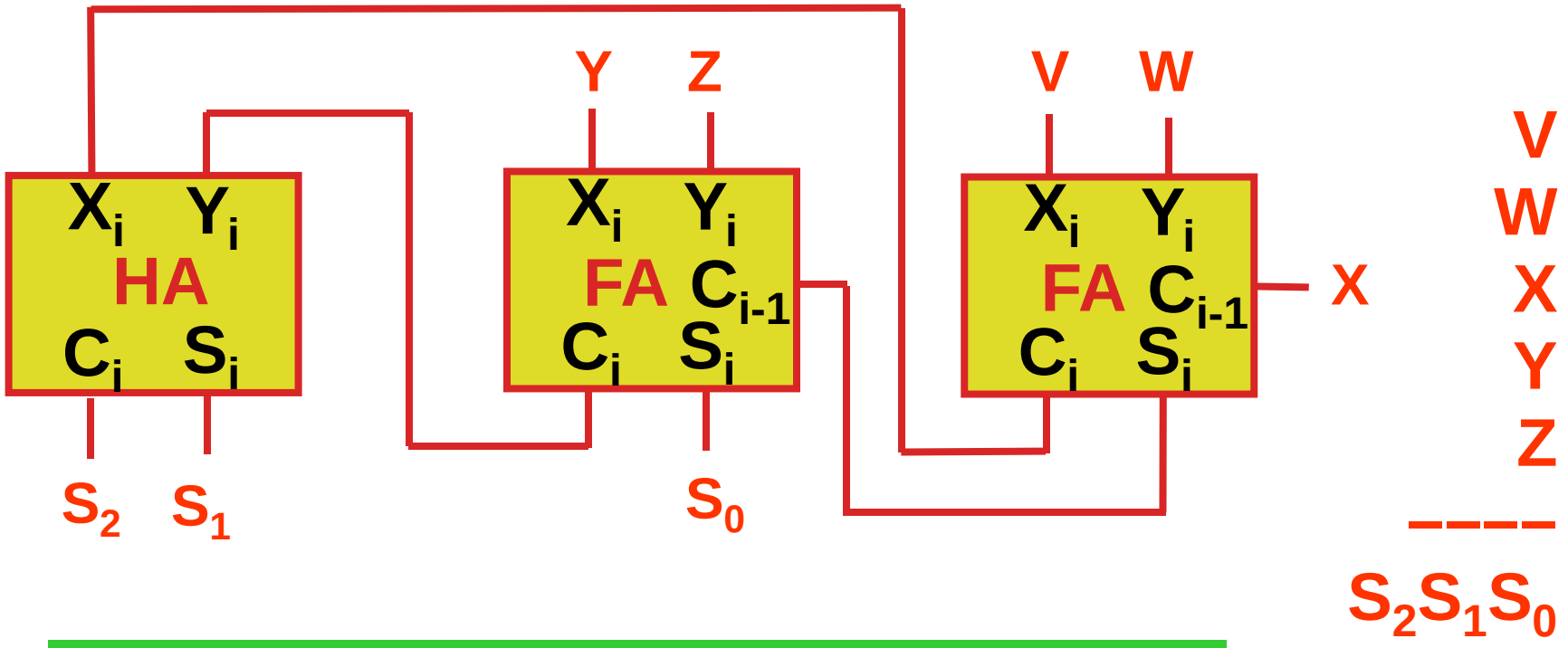


$$S_i = X_i \oplus Y_i \oplus C_{i-1}$$

$$C_i = X_i \bullet Y_i + X_i \bullet C_{i-1} + Y_i \bullet C_{i-1}$$

Example – Vote Counting Circuit

- **Using only half adders and full adders, design a circuit that finds the (unsigned) sum of five binary digits**



Also called a "population" counter

Clicker Quiz

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 TRUTH_TABLE directive, a circuit that finds the sum of three 2-bit unsigned numbers would require ___ rows in the table (following the declaration).**
   - **A. 16**
   - **B. 32**
   - **C. 64**
   - **D. 128**
   - **E. none of the above**

2. **Implemented using a TRUTH_TABLE directive, a circuit that finds the sum of three 2-bit unsigned numbers would require ___ discrete entries in the table (following the declaration).**

   A. **64**
   B. **128**
   C. **640**
   D. **1280**
   E. **none of the above**

3. **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 Circuits

- **Two binary words, each with *n* bits, can be added using a *ripple* adder – a cascade of *n* full-adder stages, each of which handles one bit (also called an *iterative* circuit)**
- **The word *ripple* describes the flow of the carries from one full adder cell to the next**
- **Subtraction is performed by taking the *diminished radix* complement of the subtrahend (using XOR gates) and setting the *least significant bit carry-in* (LSB C_{in}) to "1" (effectively forming the *radix* complement of the subtrahend)**

Review of Radix Addition

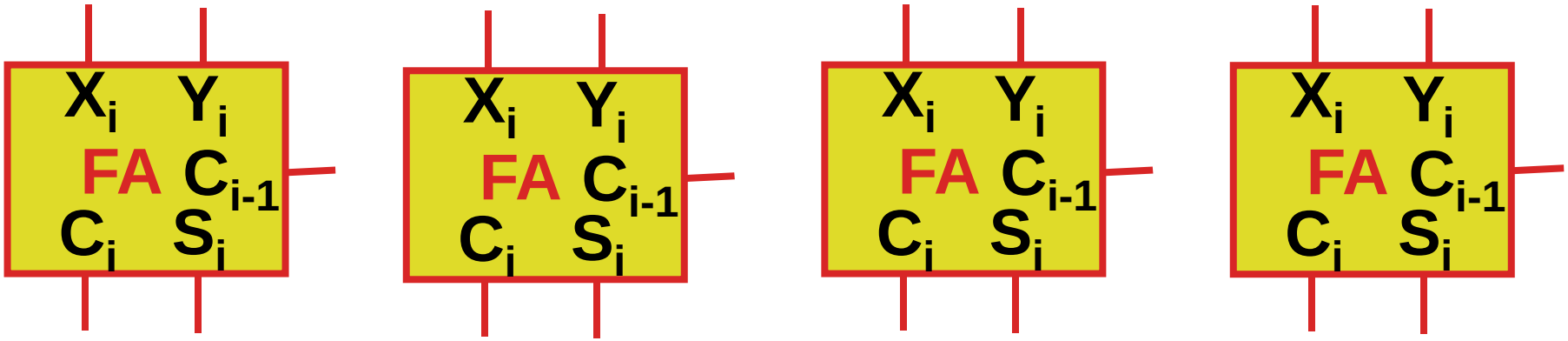
- **Method: Add all digits, including the sign digits; ignore any carry out of the sign position**
- **Problem: Since we are working with numbers of fixed length, the result of an addition can yield a number which is *too large* to represent in the same number of digits – this error condition is called *overflow***
- **Important: When overflow occurs, there is *no valid numeric result***

Overflow Detection

- **Summarization: Overflow occurs if two positive numbers are added and a negative result is obtained, or if two negative numbers are added and a positive result is obtained (or, if 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**

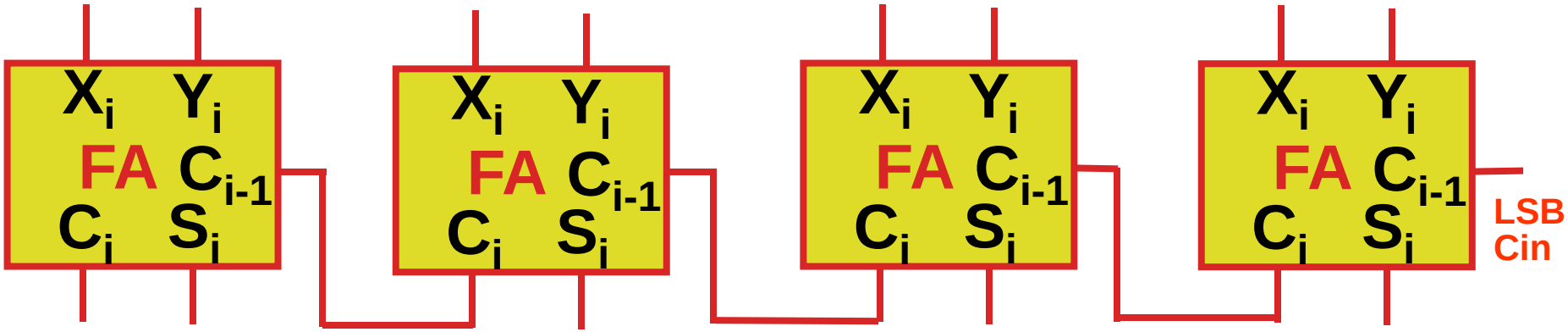
Example – 4-bit Ripple Adder/Subtractor

$$\begin{array}{rl} & A_3\ A_2\ A_1\ A_0 \\ \pm & B_3\ B_2\ B_1\ B_0 \\ \hline & S_3\ S_2\ S_1\ S_0 \end{array}$$



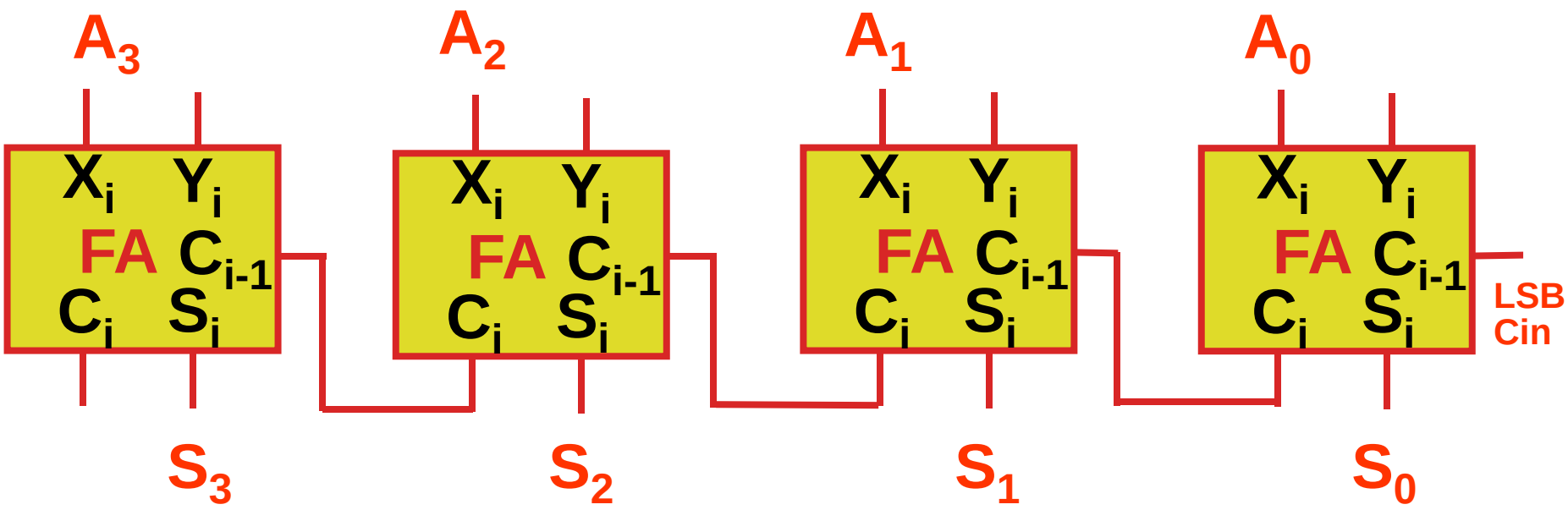
Example – 4-bit Ripple Adder/Subtractor

$$\begin{array}{rl} & A_3\ A_2\ A_1\ A_0 \\ \pm & B_3\ B_2\ B_1\ B_0 \\ \hline & S_3\ S_2\ S_1\ S_0 \end{array}$$

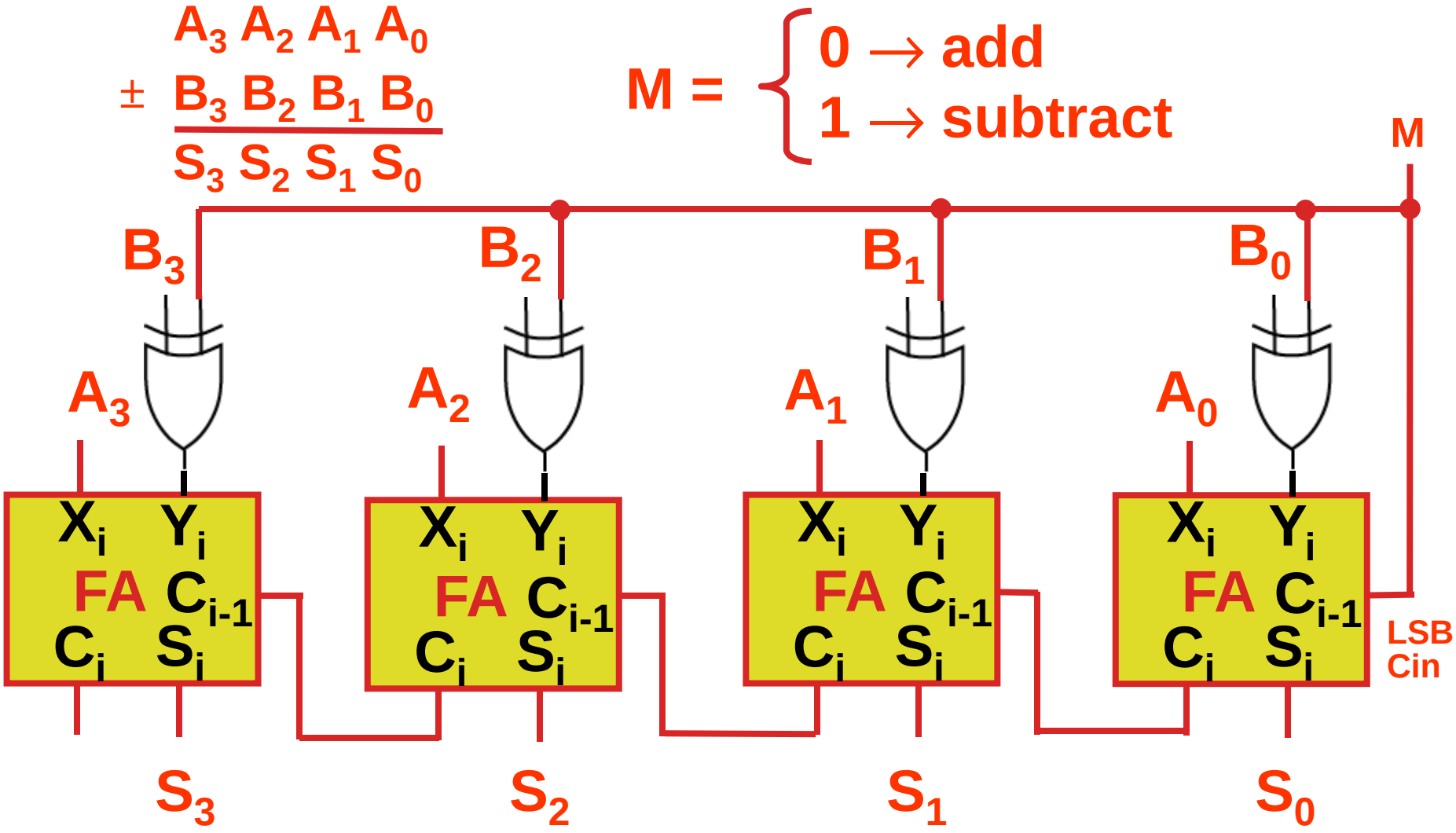


Example – 4-bit Ripple Adder/Subtractor

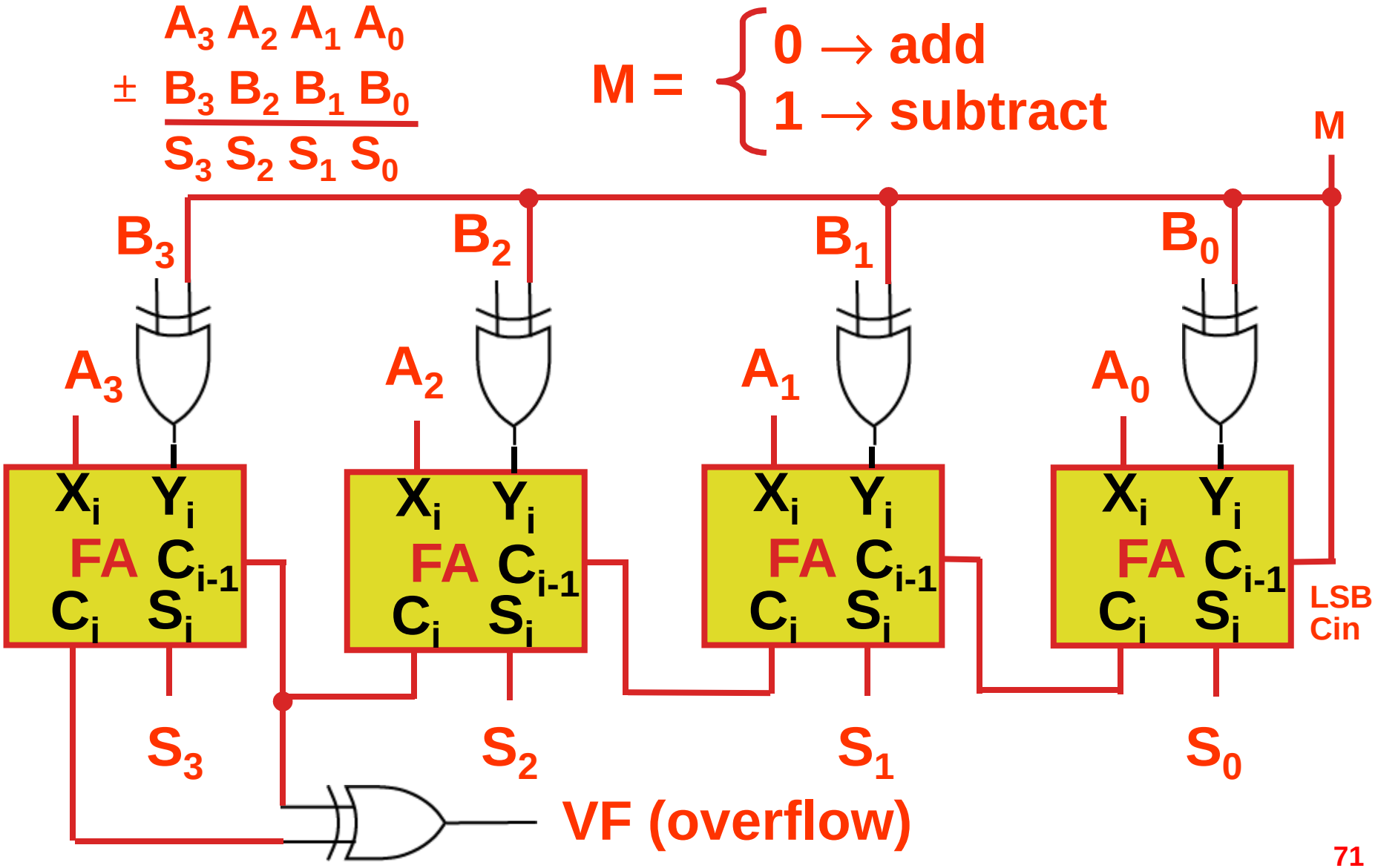
$$\begin{array}{rl} & A_3\ A_2\ A_1\ A_0 \\ \pm & B_3\ B_2\ B_1\ B_0 \\ \hline & S_3\ S_2\ S_1\ S_0 \end{array}$$



Example – 4-bit Ripple Adder/Subtractor



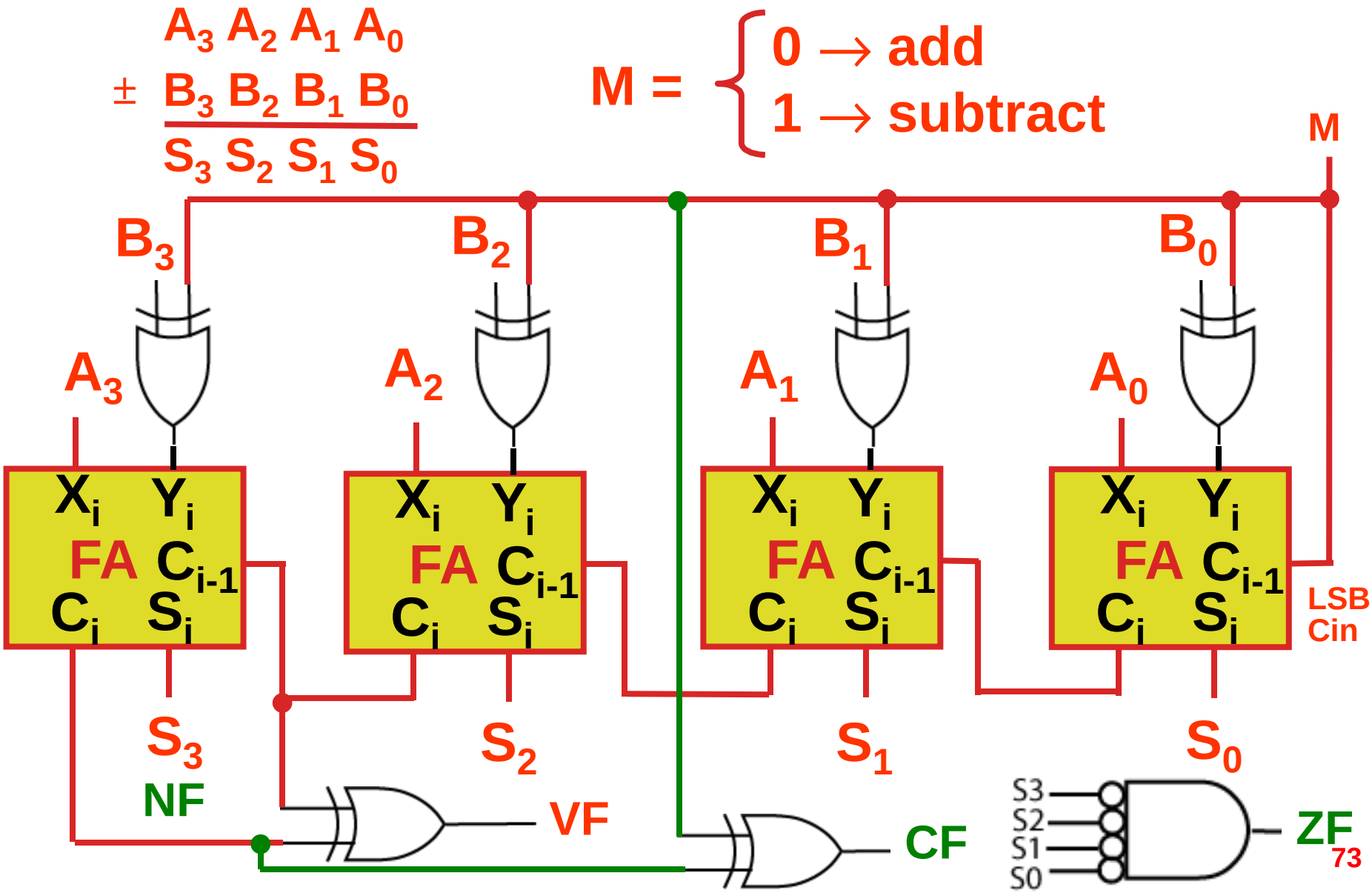
Example – 4-bit Ripple Adder/Subtractor



Other Conditions of Interest

- **In addition to overflow, other conditions of interest following an arithmetic operation include the following:**
  - **ZERO –** ***the result of the computation was 00…0***
  - **NEGATIVE –** ***the result of the computation was a negative number***
  - **CARRY/BORROW –** ***the computation produced a carry out of the sign position after an addition, or produced a borrow out of the sign position after a subtraction (the complement of the carry out)***
- **These conditions are sometimes referred to as "*condition codes*" or "*flags*"**

Example – 4-bit Ripple Adder/Subtractor



Clicker Quiz

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 (CF) 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

3. Following an add operation, the negative flag (NF) 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

Comparators

- **Comparing two binary words for equality is a commonly used operation in computer systems – a circuit that does this is called a *comparator***
- **XOR and XNOR gates may be viewed as *one-bit comparators* (from which larger comparators can be built)**
- **Circuits that determine an arithmetic relationship between two operands (greater or less than) are called *magnitude comparators***

Example – 4-bit Magnitude Comparator

- **Design a 4-bit (signed) magnitude comparator that determines if A=B, A<B, or A>B**
- **Solution: Calculate A–B and examine the condition codes produced for each case**
  - **ZERO (“ZF” for zero flag)**
  - **NEGATIVE (“NF” for negative flag)**
  - **CARRY (“CF” for carry/borrow flag)**
  - **OVERFLOW (“VF” for overflow flag)**
- **Note: Because a SUBTRACTION is being performed, need to *complement* the carry out of the sign to make it a BORROW**

Need to know how the condition codes are affected for all possible results generated

Example – 4-bit Magnitude Comparator

Step 1: Determine condition codes produced for all possible (2-bit) cases of A–B

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

Example – 4-bit Magnitude Comparator

Step 2: Make note of the condition code combinations corresponding to the functions A=B, A<B, and A>B

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

Example – 4-bit Magnitude Comparator

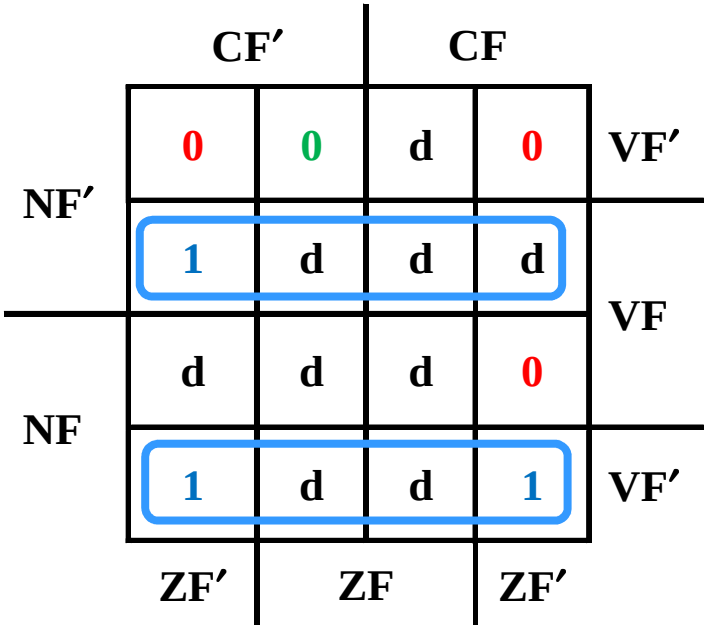
Step 3: Observe that $F_{A=B} = ZF$

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

Example – 4-bit Magnitude Comparator

Step 4: Map and minimize the function for $F_{A<B}$

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

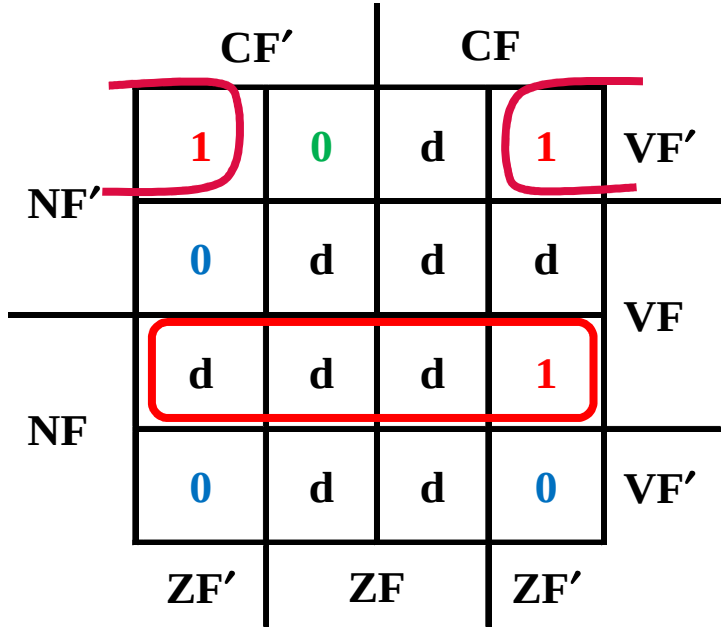


$F_{A<B} = NF \oplus VF$

Example – 4-bit Magnitude Comparator

Step 5: Map and minimize the function for $F_{A>B}$

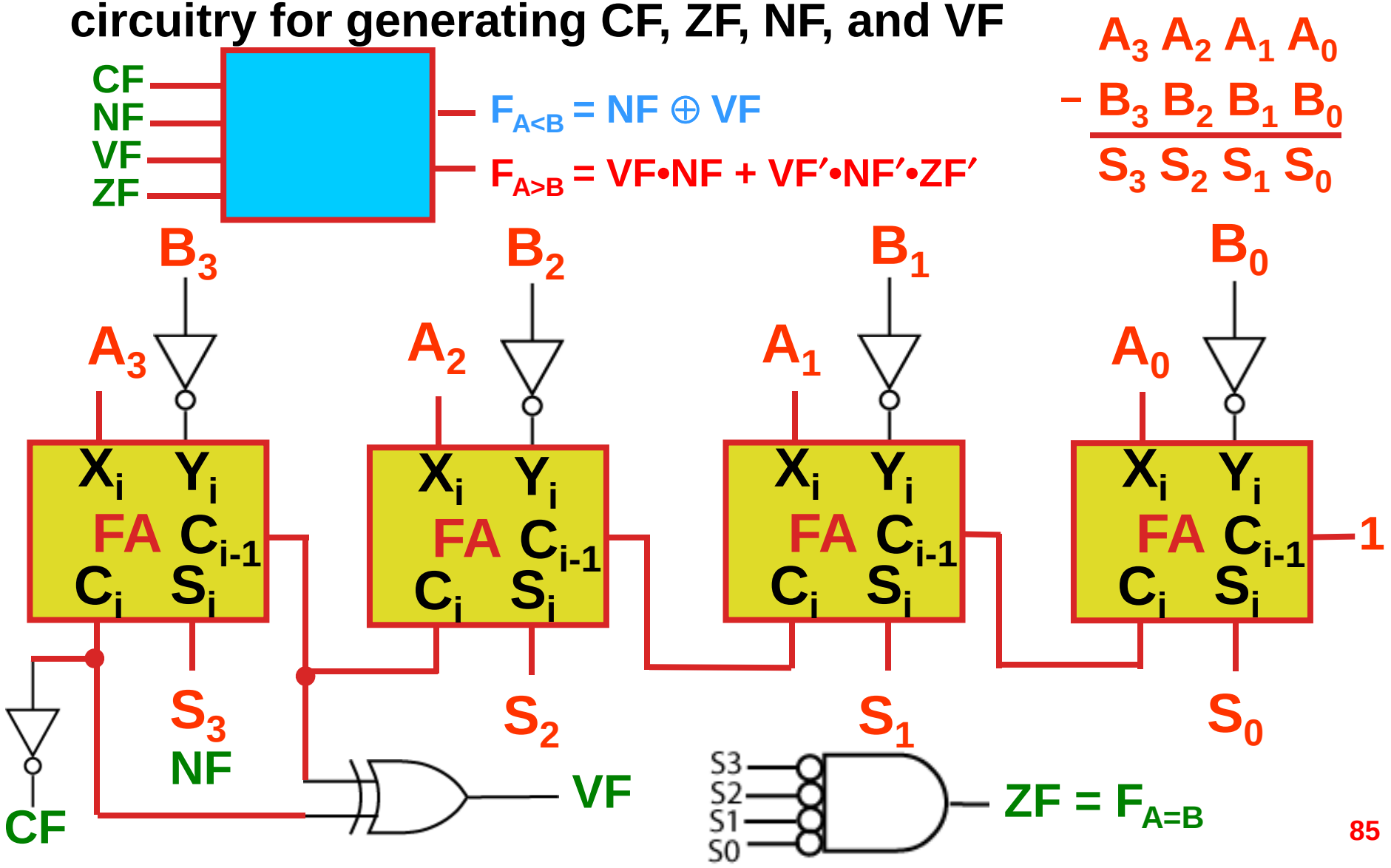
A1	A0	B1	B0	?	CF	ZF	NF	VF
0	0	0	0	A=B	0	1	0	0
0	0	0	1	A<B	1	0	1	0
0	0	1	0	A>B	1	0	1	1
0	0	1	1	A>B	1	0	0	0
0	1	0	0	A>B	0	0	0	0
0	1	0	1	A=B	0	1	0	0
0	1	1	0	A>B	1	0	1	1
0	1	1	1	A>B	1	0	1	1
1	0	0	0	A<B	0	0	1	0
1	0	0	1	A<B	0	0	0	1
1	0	1	0	A=B	0	1	0	0
1	0	1	1	A<B	1	0	1	0
1	1	0	0	A<B	0	0	1	0
1	1	0	1	A<B	0	0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

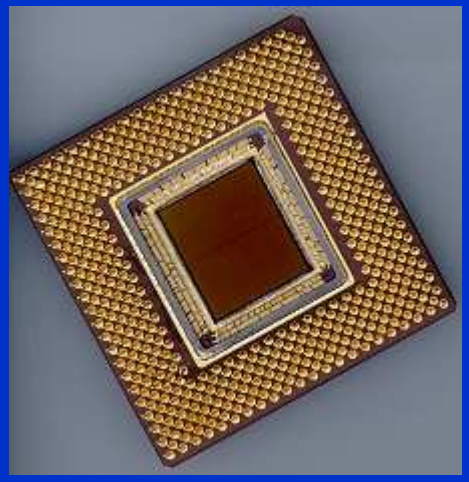


$F_{A>B} = VF \cdot NF + VF' \cdot NF' \cdot ZF'$

Example – 4-bit Magnitude Comparator

Step 6: Design a 4-bit subtractor with condition code circuitry for generating CF, ZF, NF, and VF





Introduction to Digital System Design

Module 4-D

Carry Look-Ahead Adder Circuits

Reading Assignment:

DDPP 4th Ed., pp. 478-482, 484-488

Instructional Objectives:

- **Describe the operation of a carry look-ahead (CLA) adder circuit, and compare its performance to that of a ripple adder circuit**
- **Define the CLA propagate (P) and generate (G) functions, and show how they can be realized using a half-adder**
- **Write the equation for the carry out function of an arbitrary CLA bit position**
- **Draw a diagram depicting the overall organization of a CLA**
- **Determine the worst case propagation delay incurred by a practical (PLD-based) realization of a CLA**
- **Describe how a "group ripple" adder can be constructed using N-bit CLA blocks**

Outline

- Overview
- CLA derivation
- CLA organization
- Sample CLA realization in ABEL
- Observations

Overview

- In a previous module, we looked at one method of constructing an n-digit binary adder from *n* full adders: connecting the ***carry out*** from one stage to the ***carry in*** of the next in a ***ripple*** fashion
- For large values of *n*, the propagation delay of a ripple adder can be excessive
- A significant speed-up could be obtained by calculating the carries ***in parallel***, rather than ***iteratively*** – a design that accomplishes this goal is the ***carry look-ahead*** (CLA) adder circuit (***look-ahead*** → ***anticipated***)

CLA Derivation

Consider the 4-bit binary adder:

Stage:	3	2	1	0
Augend:	X3	X2	X1	X0
Addend:	Y3	Y2	Y1	Y0
Sum:	S3	S2	S1	S0

CLA Derivation

- **Definition**: The ***generate function*** $G_i = 1$ if there is a ***carry out*** of stage i regardless of whether or not there is a ***carry in*** to stage i (i.e., both X_i and Y_i are 1)

$$G_i = X_i \bullet Y_i$$

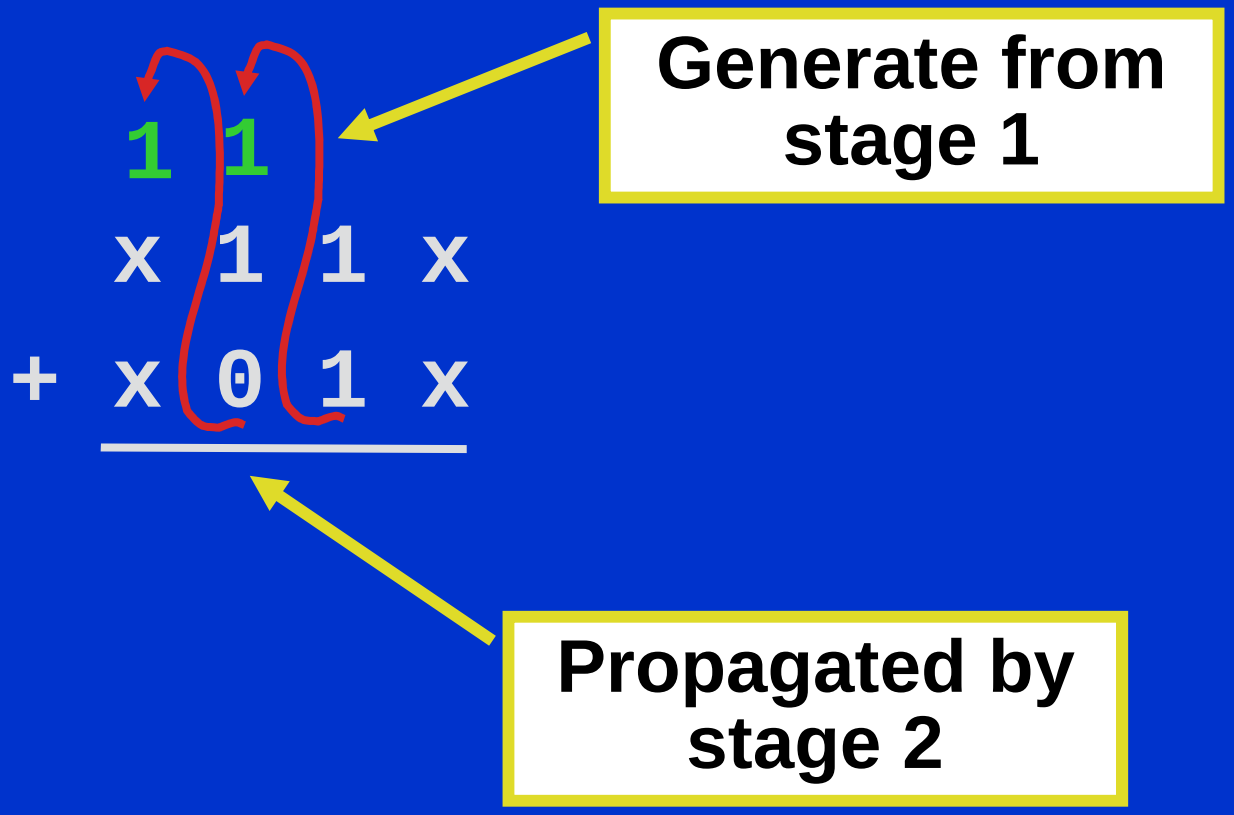
- **Definition**: The ***propagate function*** $P_i = 1$ if a carry in to stage i will cause a carry out of stage i (i.e., ***either*** $X_i = 0$ and $Y_i = 1$ ***or*** $X_i = 1$ and $Y_i = 0$)

$$P_i = X_i \oplus Y_i$$

NOTE: Another valid definition of P_i is X_i+Y_i – the "XOR" definition leads to some CLA circuit simplifications, however

CLA Derivation

- **Illustration of propagate and generate**



CLA Derivation

- **Note that the P_i and G_i functions can be generated using a *half adder***

X_i	Y_i	C_i	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$P_i = S_i = X_i \oplus Y_i$$

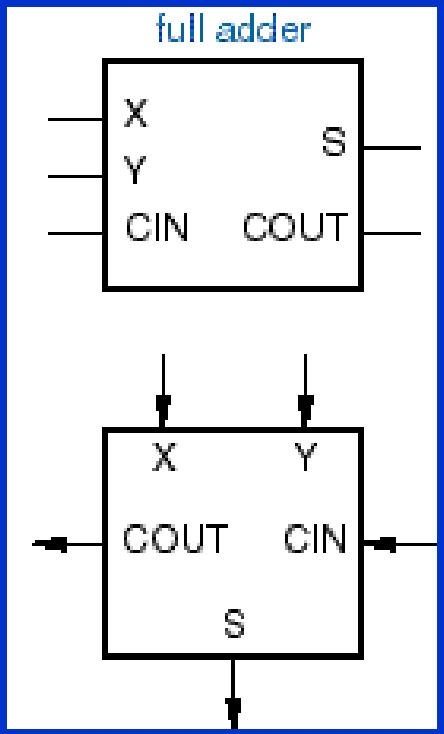
$$G_i = C_i = X_i \bullet Y_i$$

CLA Derivation

- **Next we would like to write our basic full adder equations in terms of propagate and generate functions**
- **To do this, we will need to reexamine the K-maps for the sum and carry equations of the full adder**

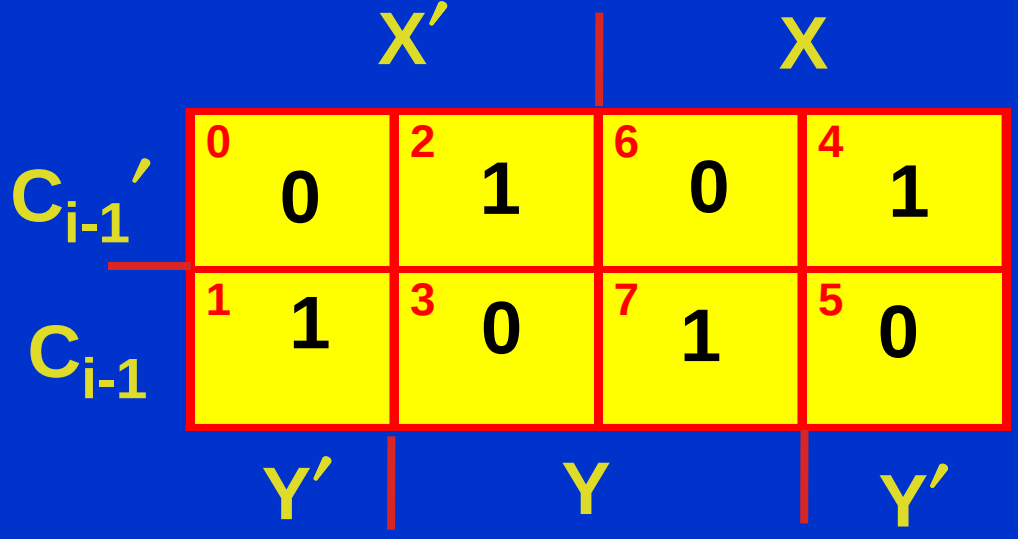
Full Adder – Review

X_i	Y_i	C_{i-1}	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Full Adder – Review

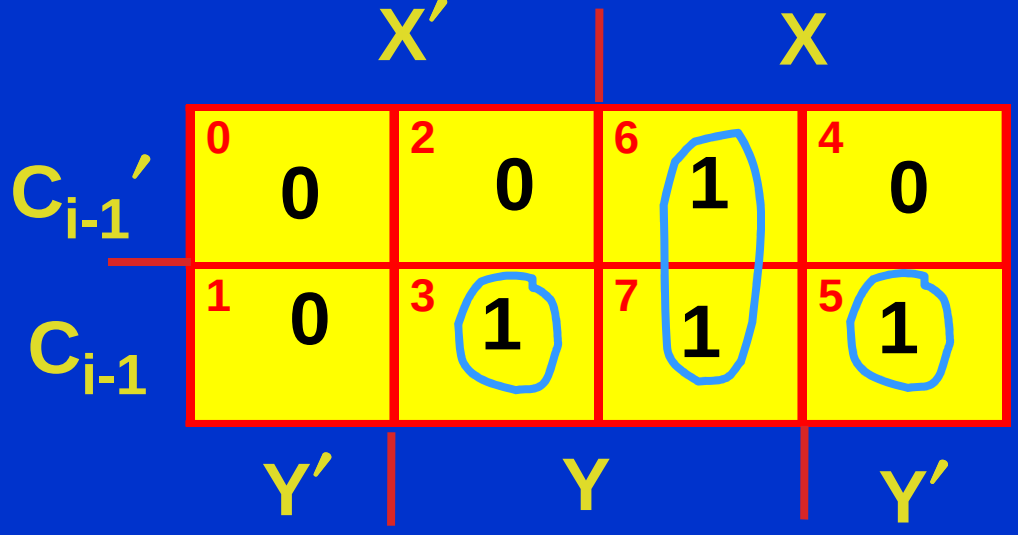
- **Map of sum function:**



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

Full Adder – Review

- **Map of carry function:**



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

CLA Derivation

- **Rewriting the equations for our 4-bit binary adder, we obtain the following:**

$$C_{-1} = C_{in}$$

$$C_0 = G_0 + C_{in} \cdot P_0$$

$$C_1 = G_1 + C_0 \cdot P_1$$

$$C_2 = G_2 + C_1 \cdot P_2$$

$$C_3 = C_{out} = G_3 + C_2 \cdot P_3$$

CLA Derivation

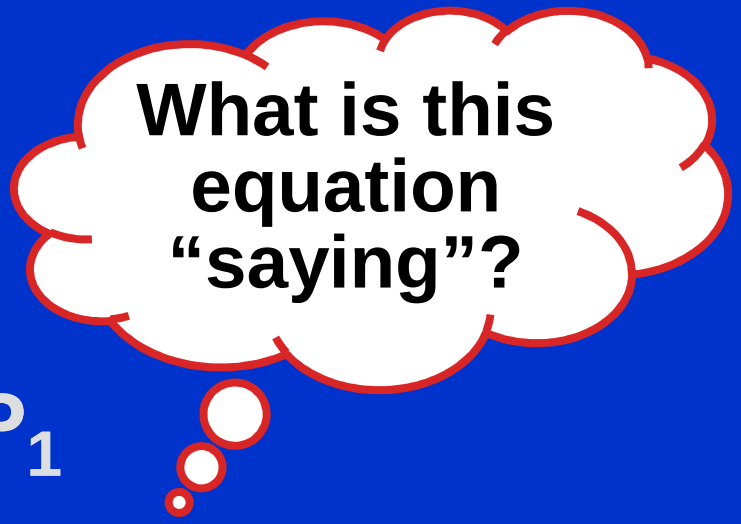
- We would like to write these equations in terms of ***available inputs*** (P's, G's, and C_{in}) rather than the ***intermediate carries*** (C_0, C_1, etc.) – the key to doing this is ***successive expansion*** of the previous equations in terms of the equation for C_0:

$$C_1 = G_1 + C_0 \cdot P_1$$

$$= G_1 + (G_0 + C_{in} \cdot P_0) \cdot P_1$$

CLA Derivation

$$C_1 = G_1 + C_0 \cdot P_1$$
$$= G_1 + (G_0 + C_{in} \cdot P_0) \cdot P_1$$
$$= G_1 + G_0 \cdot P_1 + C_{in} \cdot P_0 \cdot P_1$$



- Each term represents one possibility for obtaining a carry out of stage 1:
  - there is a ***generate*** in stage 1 ($G_1 = 1$)
  - there is a ***generate*** in stage 0 ($G_0 = 1$) which is ***propagated*** by stage 1 ($P_1 = 1$)
  - there is a ***carry in*** ($C_{in} = 1$) which is ***propagated*** by stages 0 ($P_0=1$) and 1 ($P_1=1$)

CLA Derivation – Exercise

- **Write the remaining 4-bit CLA adder carry equations:**

  $C_2 =$

  $C_3 =$

CLA Derivation – Exercise

- **Write the remaining 4-bit CLA adder carry equations:**

$$C_2 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_{IN} \cdot P_0 \cdot P_1 \cdot P_2$$

$$C_3 =$$

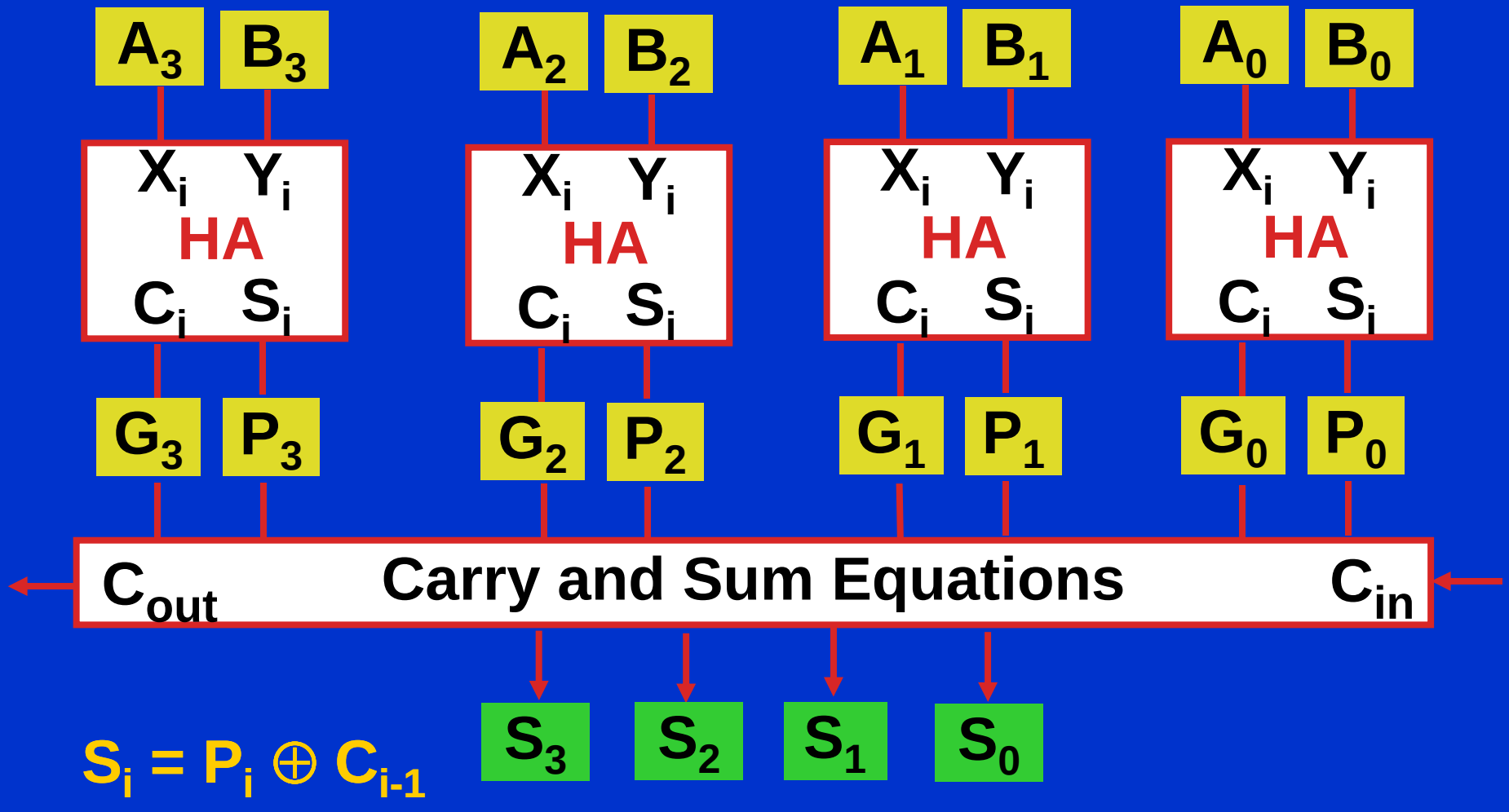
CLA Derivation – Exercise

- **Write the remaining 4-bit CLA adder carry equations:**

$$C_2 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_{IN} \cdot P_0 \cdot P_1 \cdot P_2$$

$$C_3 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_{IN} \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

CLA Organization



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

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

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

Example – Sample 4-bit CLA Realization

```
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
```

Example – Sample 4-bit CLA Realization

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
5.95	1	CIN	S0
5.95	1	CIN	S1
5.95	1	CIN	S2
5.95	1	X0	S0
5.95	1	X0	S1
5.95	1	X0	S2
5.95	1	Y0	S0
5.95	1	Y0	S1
5.95	1	Y0	S2
5.90	1	X1	S1
5.90	1	X1	S2
5.90	1	Y1	S1

Example – CLA Realization Using + 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
```

Note: The "+" operator in ABEL synthesizes CLA equations

Example – CLA Realization Using + Operator

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
5.95	1	CIN	S0
5.95	1	CIN	S1
5.95	1	CIN	S2
5.95	1	X0	S0
5.95	1	X0	S1
5.95	1	X0	S2
5.95	1	Y0	S0
5.95	1	Y0	S1
5.95	1	Y0	S2
5.90	1	X1	S1
5.90	1	X1	S2
5.90	1	Y1	S1

Obtain same timing results as "hand coded" CLA equations!

Observations

- Note that ***regardless*** of the adder length (***n***), the time required to produce ***any*** sum digit is the ***same*** – i.e., all sum digits are produced ***in parallel***
- Large CLA adders are difficult to build in practice because of the "product term explosion" that occurs as the carry equations are expanded
- A reasonable compromise is to make a ***group ripple adder*** by cascading **m**-bit CLA blocks together to make a **k**x**m**-bit adder (where **k** is the number of CLA blocks)
- The "**+**" operator in ABEL can be used to ***automatically synthesize*** CLA blocks

Clicker Quiz

```
MODULE cla4_v2
TITLE '4-bit Carry Look-Ahead Adder Block – Version 2'

DECLARATIONS

X0..X3, Y0..Y3 pin;  " operands
CIN pin;  " carry in (Cin)
C0..C3 pin istype 'com';  " carry equations (C3 is Cout)
S0..S3 pin istype 'com';  " sum outputs

G0 = X0&Y0;   " generate function declarations
G1 = X1&Y1;
G2 = X2&Y2;
G3 = X3&Y3;

P0 = X0$Y0;   " propagate function declarations
P1 = X1$Y1;
P2 = X2$Y2;
P3 = X3$Y3;

EQUATIONS

" carry equations
C0 = G0 # CIN&P0;
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;

" sum equations
S0 = CIN$P0;
S1 = C0$P1;
S2 = C1$P2;
S3 = C2$P3;

END
```

1. **If coded as shown, the number of product terms required to implement the equation for C1 would be:**

   A. 3

   B. 4

   C. 7

   D. 10

   E. none of the above

2. **If coded as shown, the number of product terms required to implement the equation for S1 would be:**

   A. 3

   B. 4

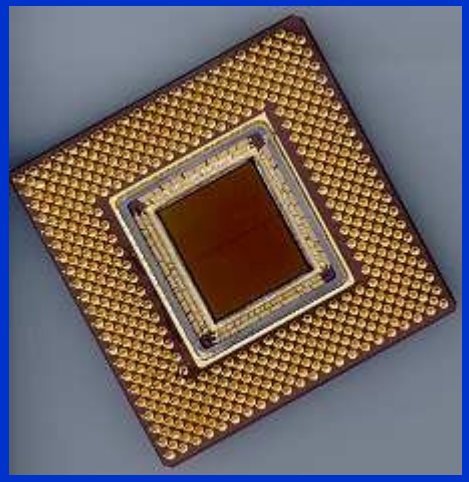
   C. 7

   D. 10

   E. none of the above

3. **If coded as shown and realized in an N-nanosecond PLD, the worst case propagation delay (in nanoseconds) would be:**

   A. **N**
   B. **2N**
   C. **3N**
   D. **4N**
   E. **none of the above**



Introduction to Digital System Design

Module 4-E
Multiplier Circuits

Reading Assignment:

DDPP 4th Ed., pp. 45-47, 494-497

Learning Objectives:

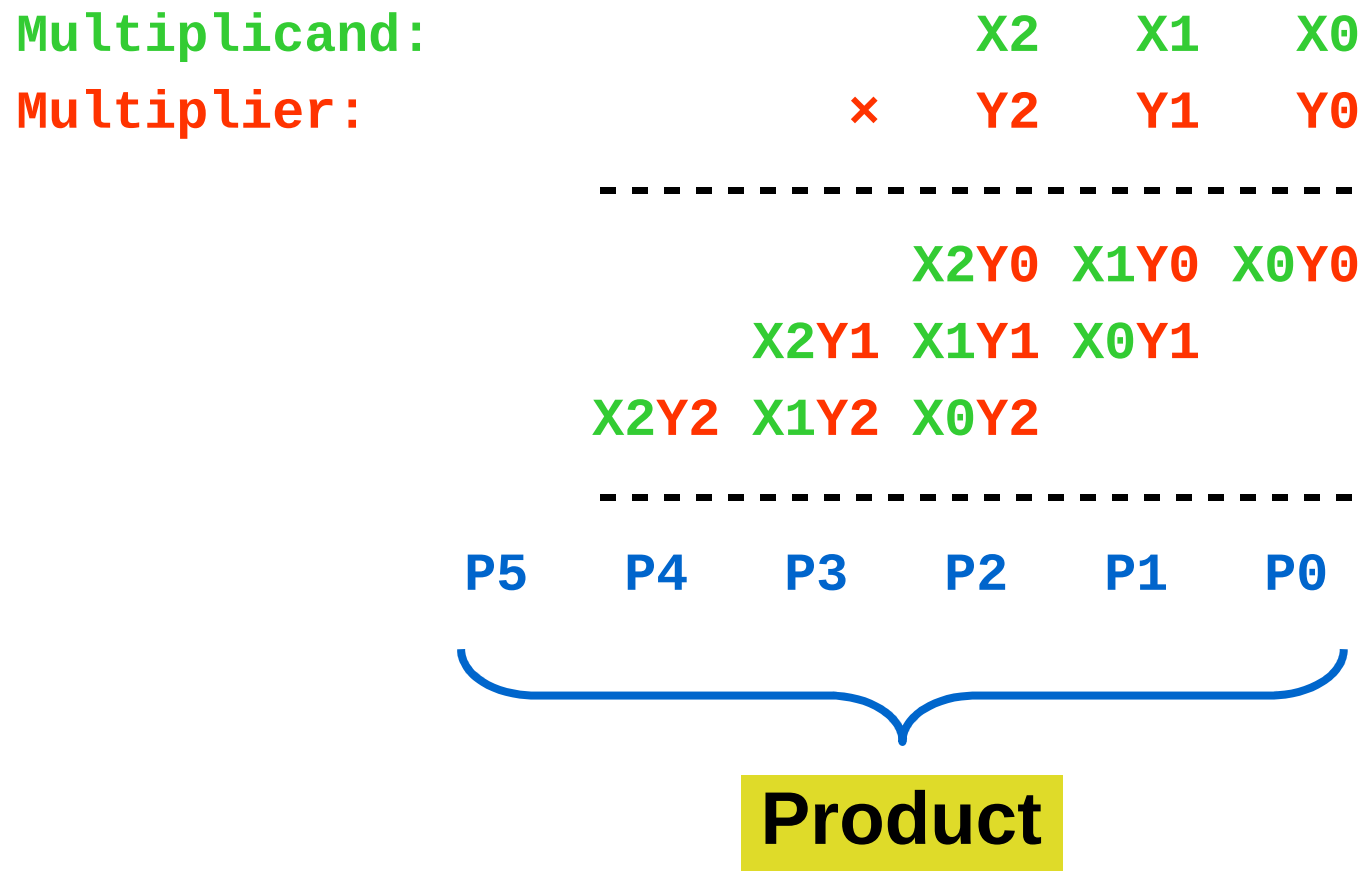
- **Describe the operation of an unsigned multiplier array constructed using full adders**
- **Determine the full adder arrangement and organization (rows/diagonals) needed to construct an NxM-bit unsigned multiplier array**
- **Determine the worst case propagation delay incurred by a practical (PLD-based) realization of an NxM-bit unsigned multiplier array**

Outline

- Overview
- Product components
- Example circuit
- Critical path analysis
- Generalizations
- Realizations in ABEL

Overview

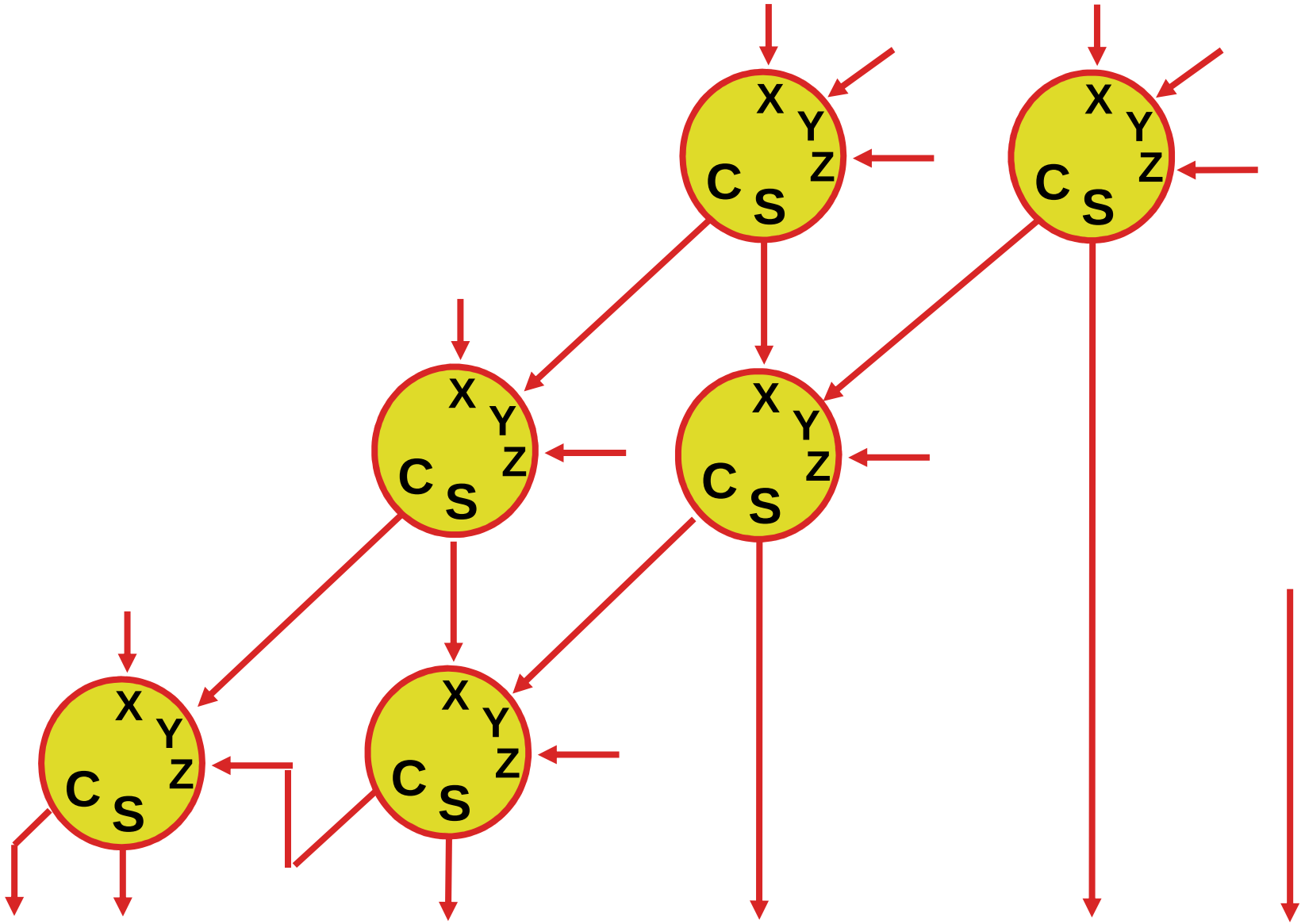
Consider a 3x3 unsigned binary multiplication:



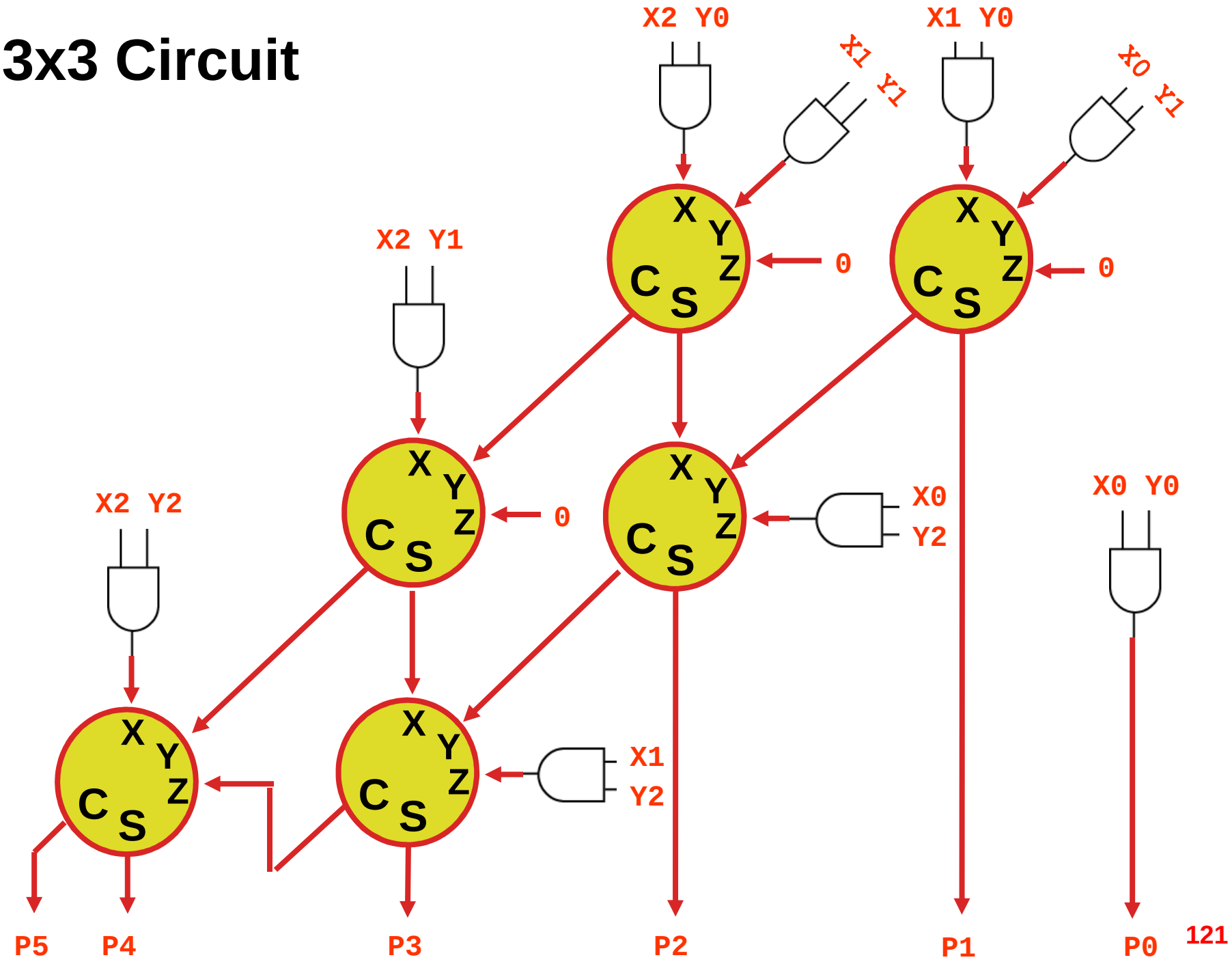
Product Components

- **Most approaches to (unsigned) combinational binary multiplication are based on the "paper-and-pencil" *shift and add* algorithm**
- **Each row is called a *product component* – a shifted multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit**
- **Each $x_i y_j$ term represents a *product component bit* (the logical AND of multiplicand bit x_i with multiplier bit y_j)**
- **The *product* P = $p_5 p_4 p_3 p_2 p_1 p_0$ is obtained by adding together all the product components**

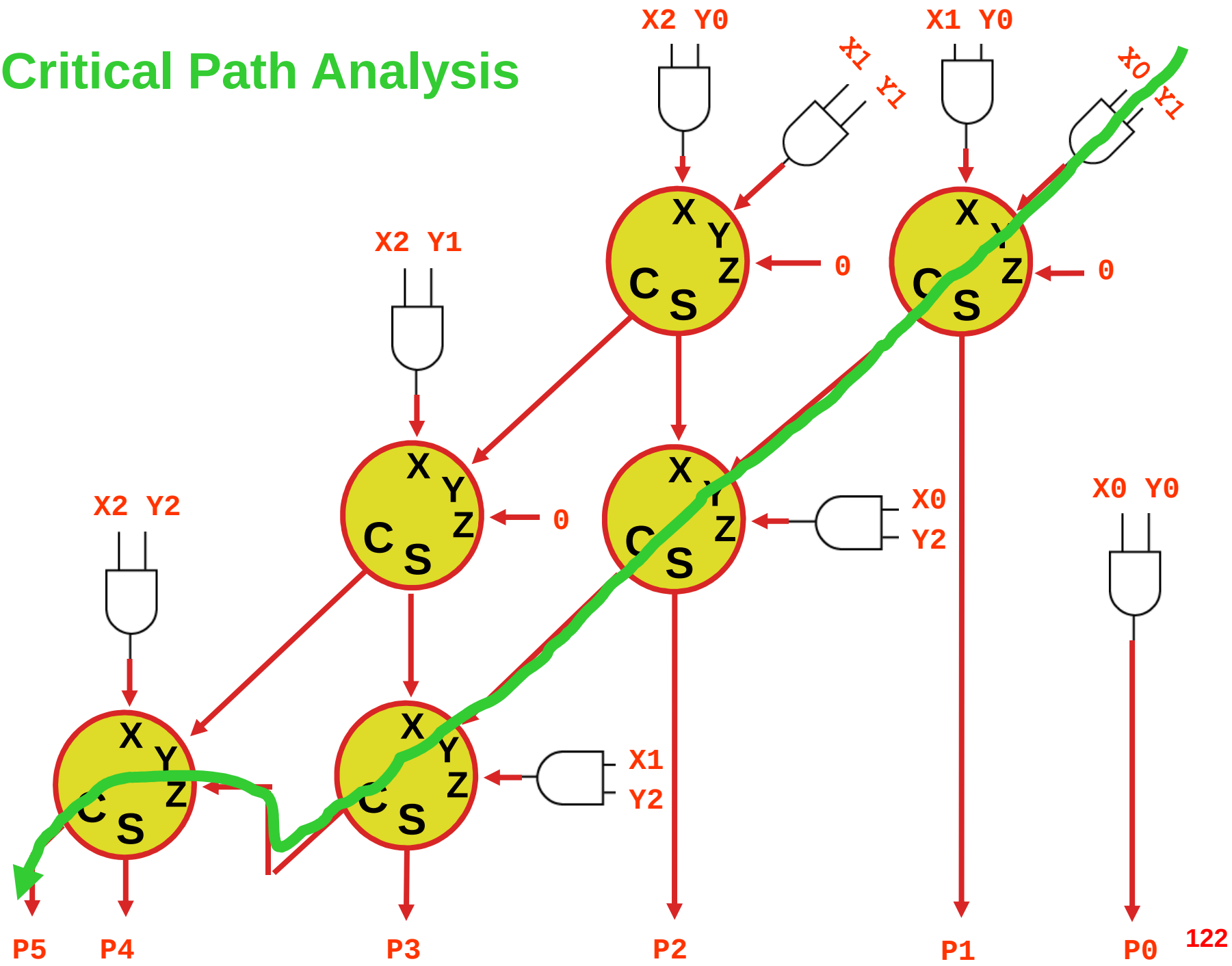
3x3 Circuit (2 diagonals, 3 rows)



3x3 Circuit


X2 Y0
X1 Y1
X1 Y0
X0 Y1
X2 Y1
X2 Y2
X0 Y2
X0 Y0
X1 Y2
0
X
Y
Z
C
S
P5
P4
P3
P2
P1
P0
121

Critical Path Analysis

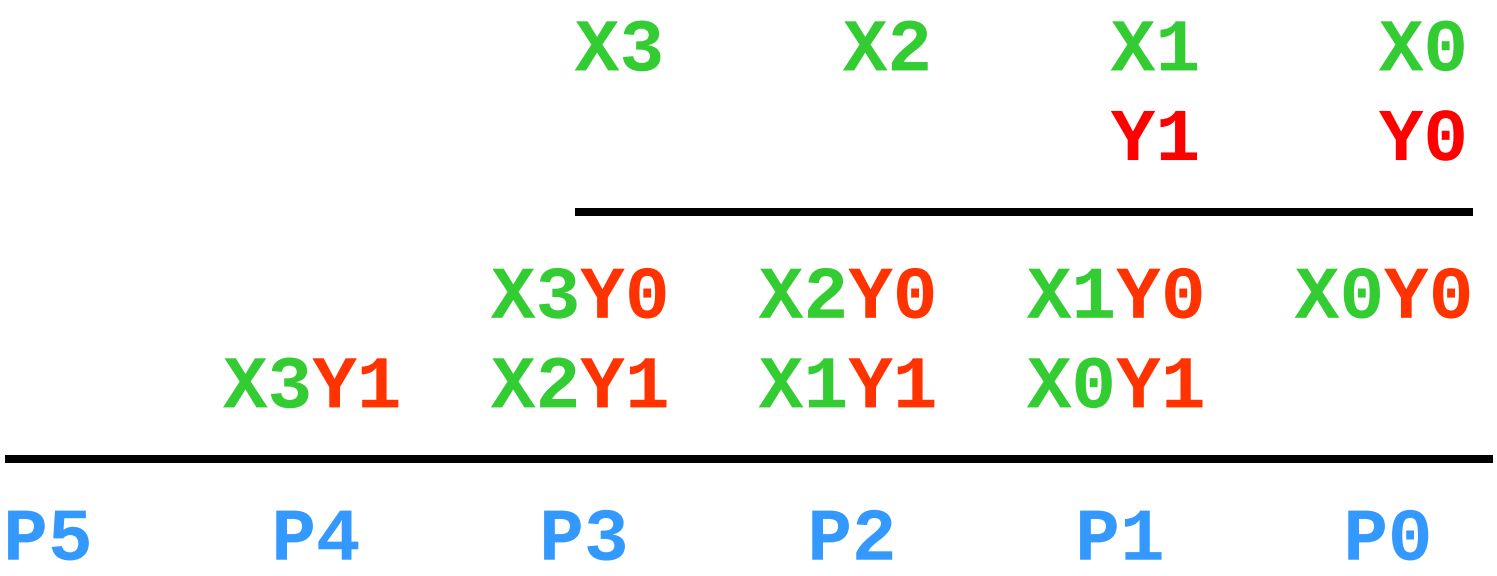


Generalizations

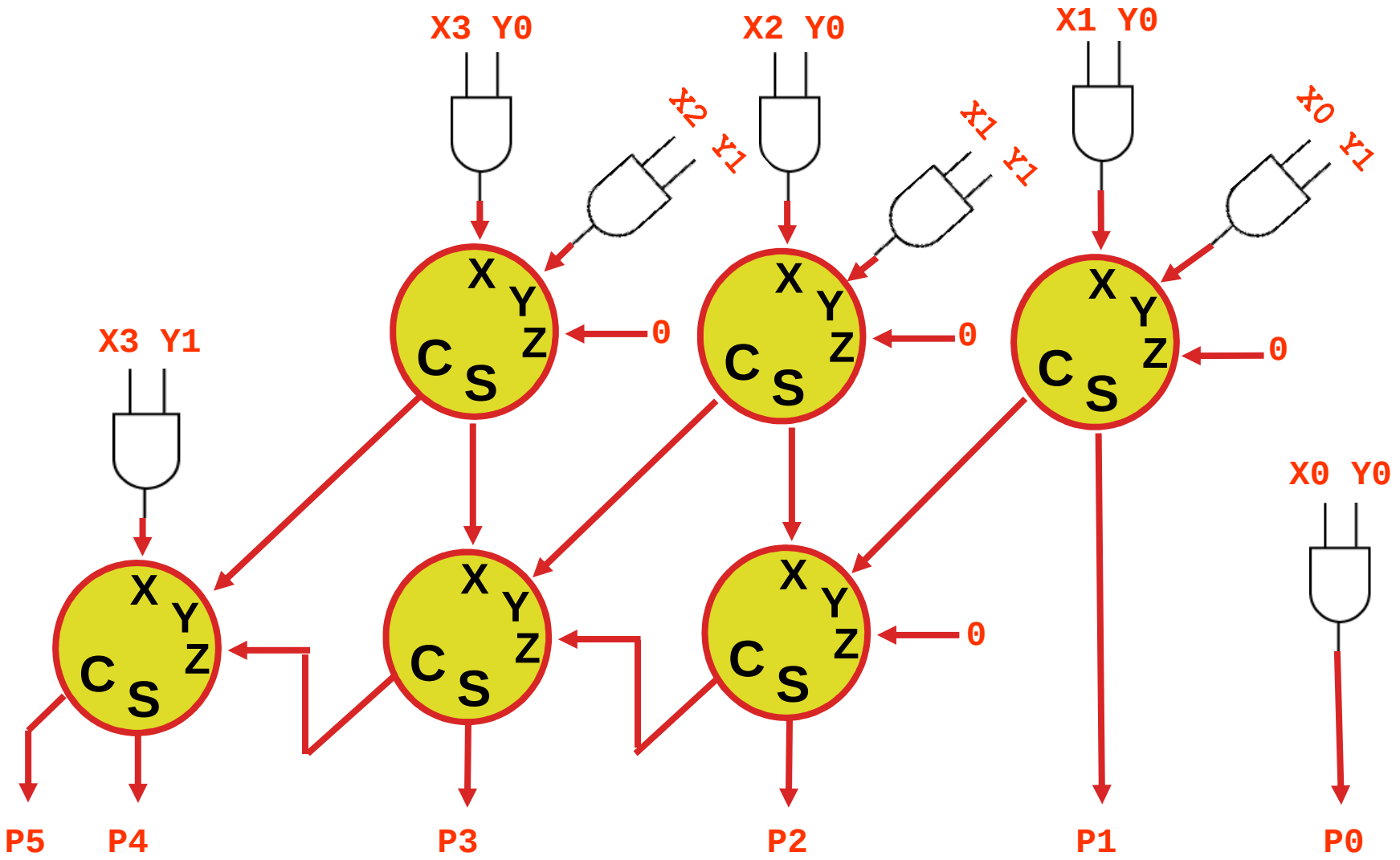
- Generalizations for an NxM multiplier
  - N = number of bits in multiplicand (top)
  - M = number of bits in multiplier (bottom)
  - produces an N+M digit result
  - requires NxM AND gates to generate the product components
  - requires N–1 *diagonals* of full adders
  - requires M rows of full adders

Exercise

- **Design a 4x2 multiplier array**

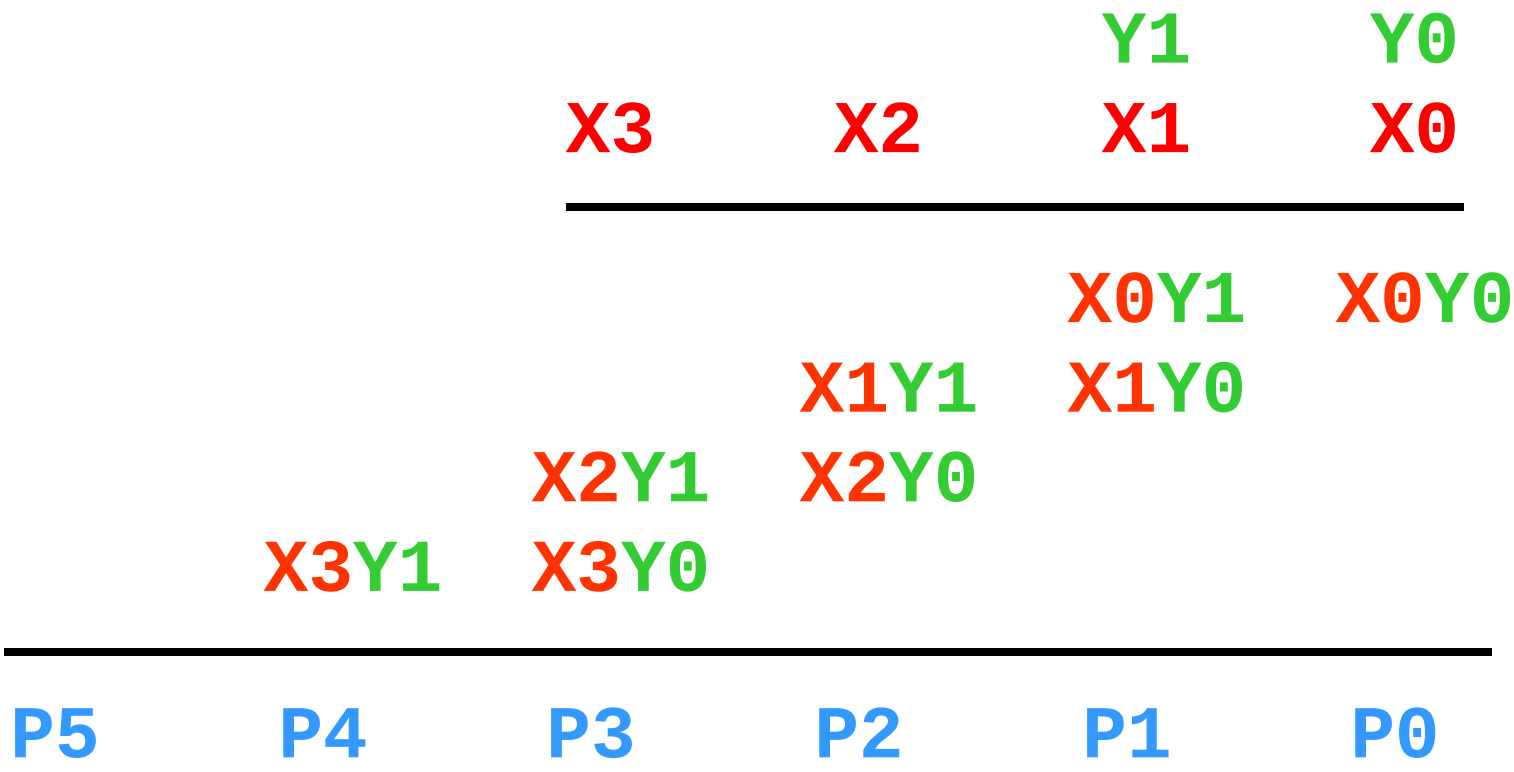


4x2 Circuit (3 diagonals, 2 rows)

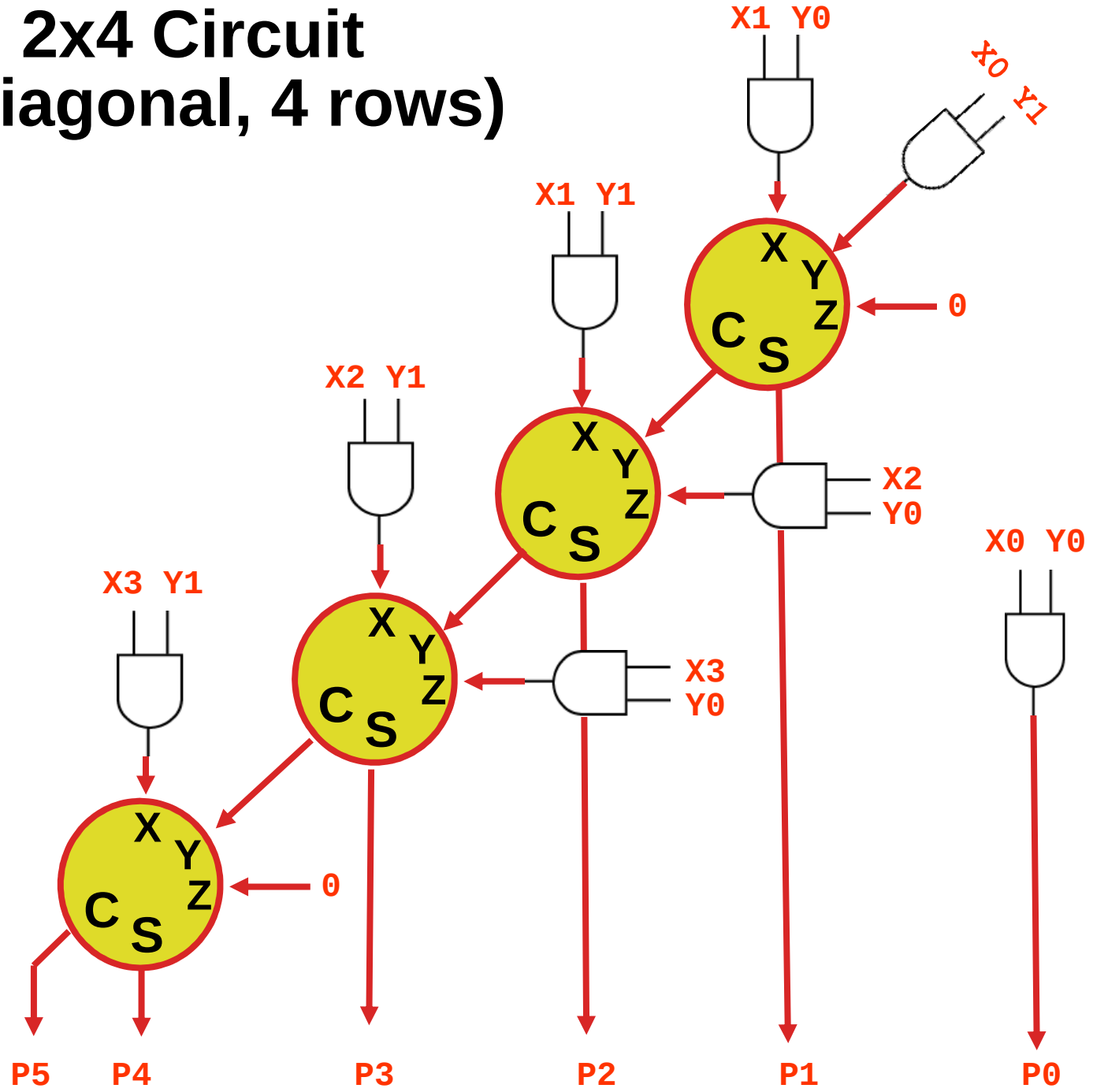


Exercise

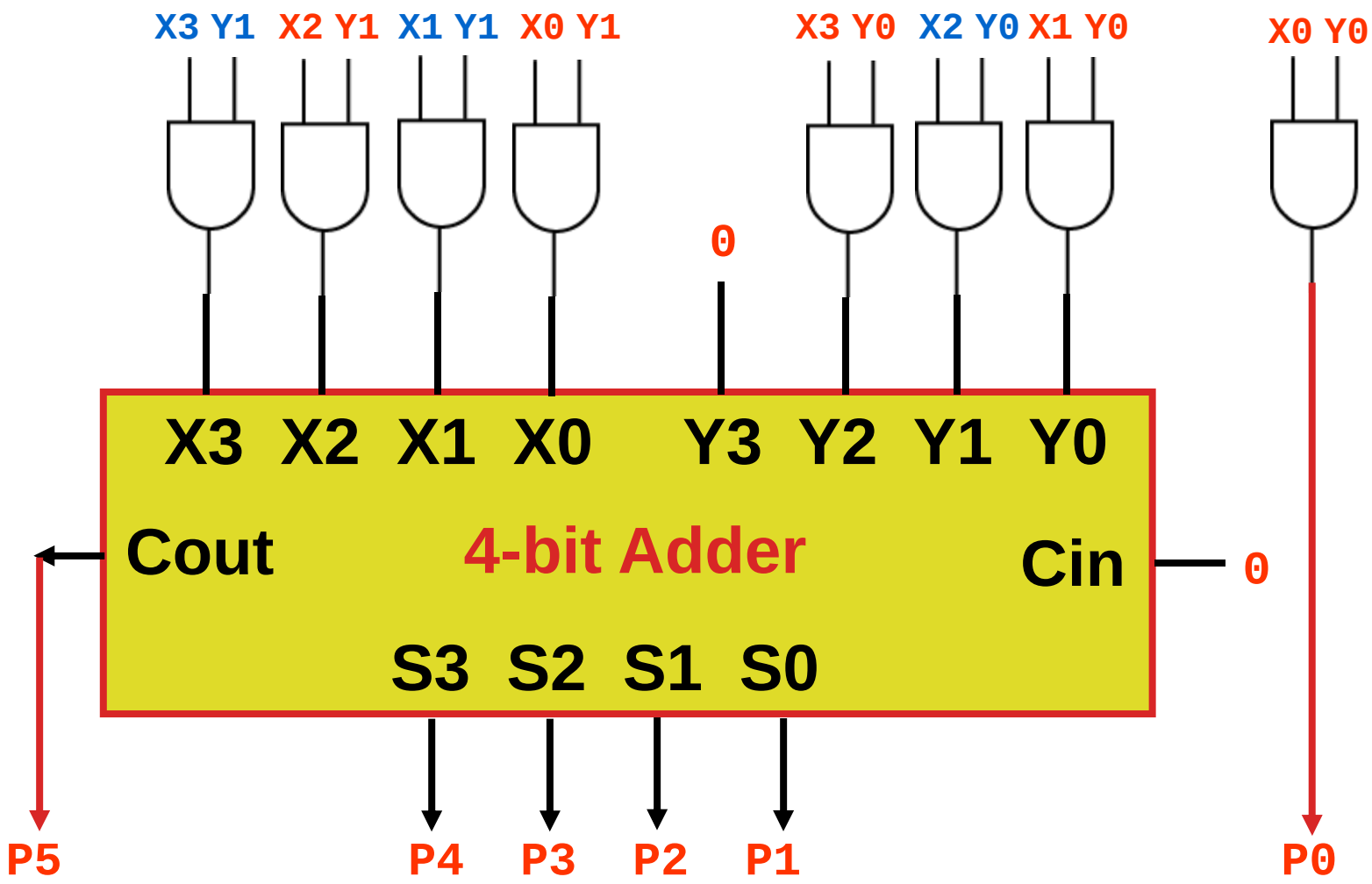
- **Design a 2x4 multiplier array**



2x4 Circuit
(1 diagonal, 4 rows)



2x4 Circuit – implemented using 4-bit adder



Clicker Quiz

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 ABEL

- **Keys**
  - **use *expressions* to define product components**
  - **use *addition operator* (+) to form unsigned sum of product components *(generates CLA adder equations)***
- **Example: 4x4 multiplier realization**

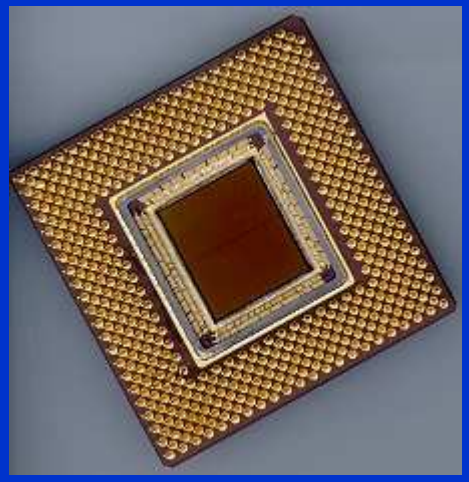
```
MODULE mul4x4

TITLE '4x4 Combinational Multiplier'

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

P = [P7..P0];
" 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
```

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

Device Resource Summary for ispMACH 4256ZE 5.8 ns CPLD

	Total	Used	Not Used		Utilization
Dedicated Pins					
Clock/Input Pins	4	4	0	-->	100
Input-Only Pins	6	4	2	-->	66
I/O / Enable Pins	2	0	2	-->	0
I/O Pins	62	8	54	-->	12
Logic Functions	256	8	248	-->	3
Input Registers	64	0	64	-->	0
GLB Inputs	576	46	530	-->	7
Logical Product Terms	1280	124	1156	-->	9
Occupied GLBs	16	6	10	-->	37
Macrocells	256	8	248	-->	3
Control Product Terms:					
GLB Clock/Clock Enables	16	0	16	-->	0
GLB Reset/Presets	16	0	16	-->	0
Macrocell Clocks	256	0	256	-->	0
Macrocell Clock Enables	256	0	256	-->	0
Macrocell Enables	256	0	256	-->	0
Macrocell Resets	256	0	256	-->	0
Macrocell Presets	256	0	256	-->	0
Global Routing Pool	324	8	316	-->	2
GRP from IFB	..	4	..	-->	..
(from input signals)	..	4	..	-->	..
(from output signals)	..	0	..	-->	..
(from bidir signals)	..	0	..	-->	..
GRP from MFB	..	4	..	-->	..



Introduction to Digital System Design

Module 4-F
BCD Adder Circuits

Reading Assignment:

DDPP 4th Ed., pp. 48-51

Learning Objectives:

- **Describe the operation of a binary coded decimal (BCD) "correction circuit"**
- **Design a BCD full adder circuit**
- **Design a BCD N-digit radix (base 10) adder/subtractor circuit**

Outline

- Overview
- General BCD adder circuit model
- Decimal addition and correction
- Decimal adder circuits
- Nine’s complement circuit

Overview

- **Even though binary numbers are the most appropriate for the internal computations of a digital system, most people still prefer to deal with decimal numbers**
- **External interfaces of a digital system may need to read or display decimal numbers, and therefore need to perform arithmetic on decimal numbers directly**
- **The most commonly used decimal code is *binary-coded decimal* (BCD)**
- **Some computers place two BCD digits in an 8-bit byte ("packed-BCD format")**

Overview

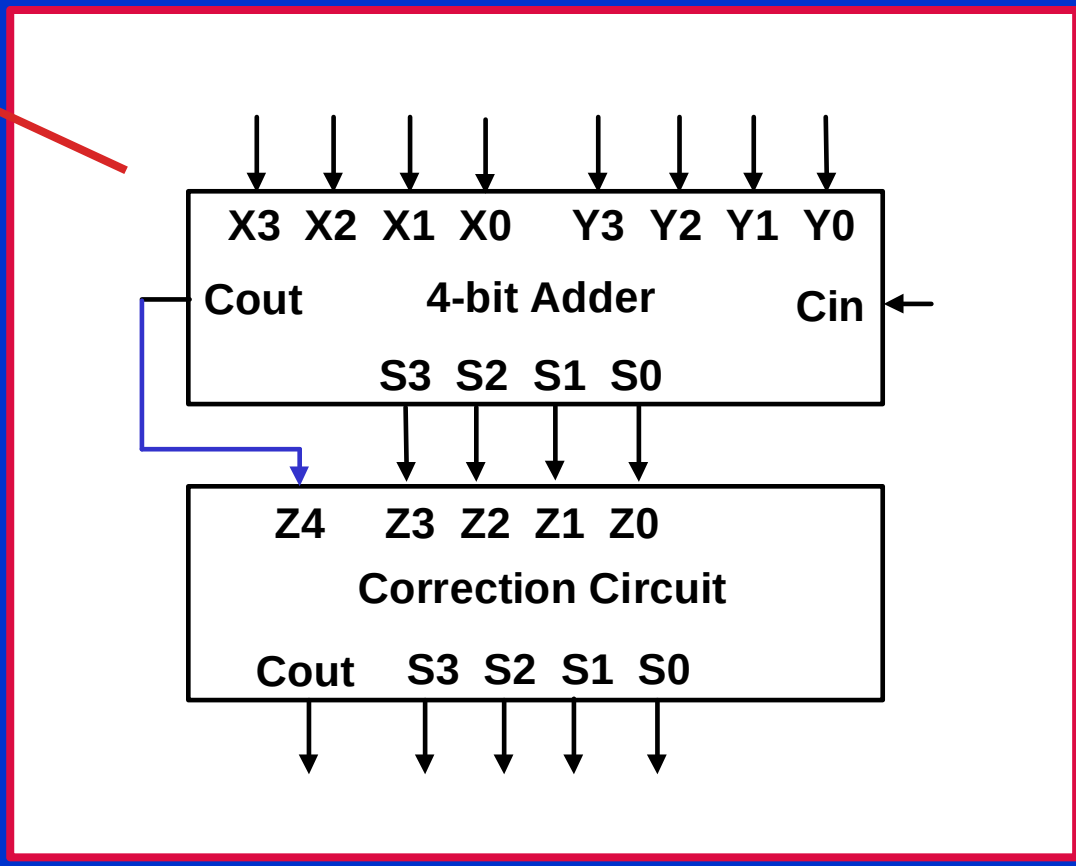
- **Consider the problem of adding a pair of BCD digits – the objective is to design a circuit that adds the two 4-bit codes along with a carry in to produce a 4-bit coded sum digit plus a carry out**
- **We would like to use standard 4-bit binary adder modules (with which we are already familiar) as basic building blocks**
- **Note that because there are six "unused combinations" in BCD, a *correction* must be performed if the direct sum of the two 4-bit codes exceeds 1001**

General BCD Adder Circuit Model

- **General circuit model:**

Conventional 4-bit binary adder

$Z_4Z_3Z_2Z_1Z_0$ is the *direct sum* obtained from the 4-bit adder



Decimal Addition and Correction

```
   3          0011
+  4       +  0100
  --         ------
   7          0111
```

Result of ADD

Here, direct addition of the 4-bit BCD codes yields the correct 4-bit BCD code for the sum digit

Decimal Addition and Correction

```
   7          0111
+  8       +  1000
  --       -------
  15          1111
           +  0110
           -------
           1  0101
```

Result of ADD

Since result > 9, add 6 to adjust

Carry out = ten's position

Decimal Addition and Correction

```
  7          0111
+ 9        + 1001
 --        ------
 16         10000
          + 0110
          ------
          1 0110
```

Result of ADD

Since result > 9, add 6 to adjust

Carry out = ten's position

Decimal Adder Circuits

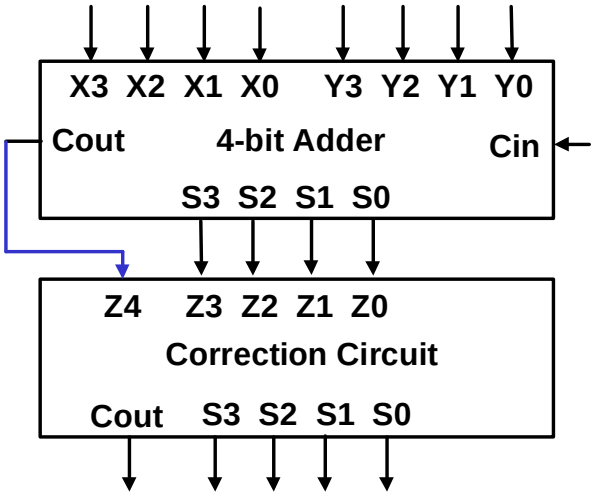
- **Summary of rules**
  - **If the sum of the two BCD digits is less than or equal to nine (1001), *no correction* is needed**
  - **If the sum is greater than nine, the result obtained directly from the 4-bit adder must be *corrected* in order to represent the proper BCD digit**
- **Some microprocessors include a "decimal adjust" (DAA) instruction for performing this correction following the addition of BCD operands**

Decimal Adder "Correction Function"

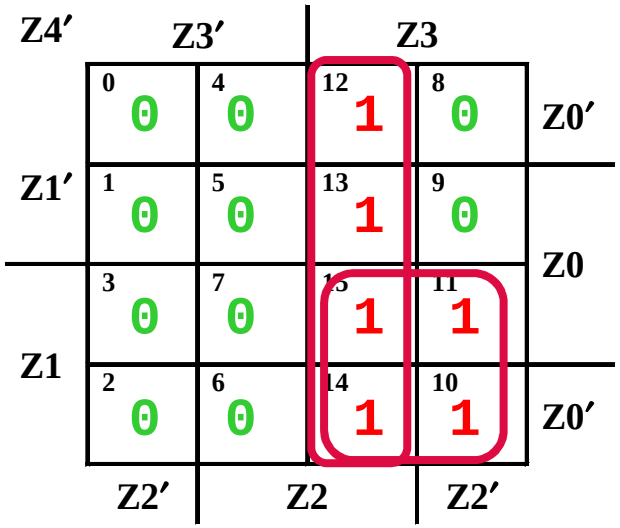
N_{10}	$Z_4\ Z_3\ Z_2\ Z_1\ Z_0$	$C_{out}\ S_3\ S_2\ S_1\ S_0$	Correction
0	0 0 0 0 0	0 0 0 0 0	<none>
1	0 0 0 0 1	0 0 0 0 1	<none>
2	0 0 0 1 0	0 0 0 1 0	<none>
3	0 0 0 1 1	0 0 0 1 1	<none>
4	0 0 1 0 0	0 0 1 0 0	<none>
5	0 0 1 0 1	0 0 1 0 1	<none>
6	0 0 1 1 0	0 0 1 1 0	<none>
7	0 0 1 1 1	0 0 1 1 1	<none>
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>

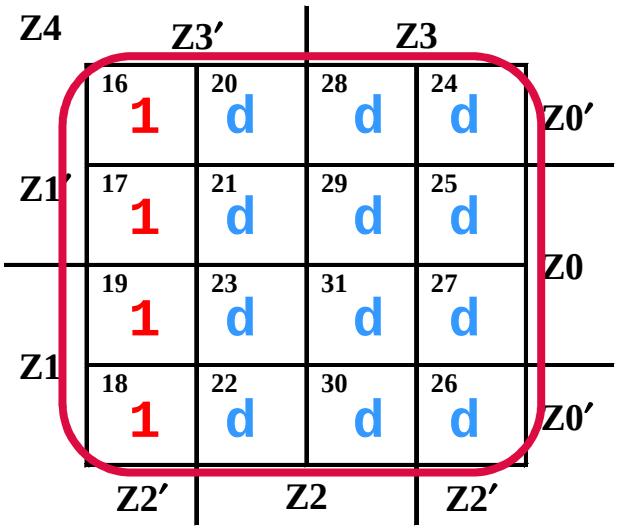
Decimal Adder "Correction Function"

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>



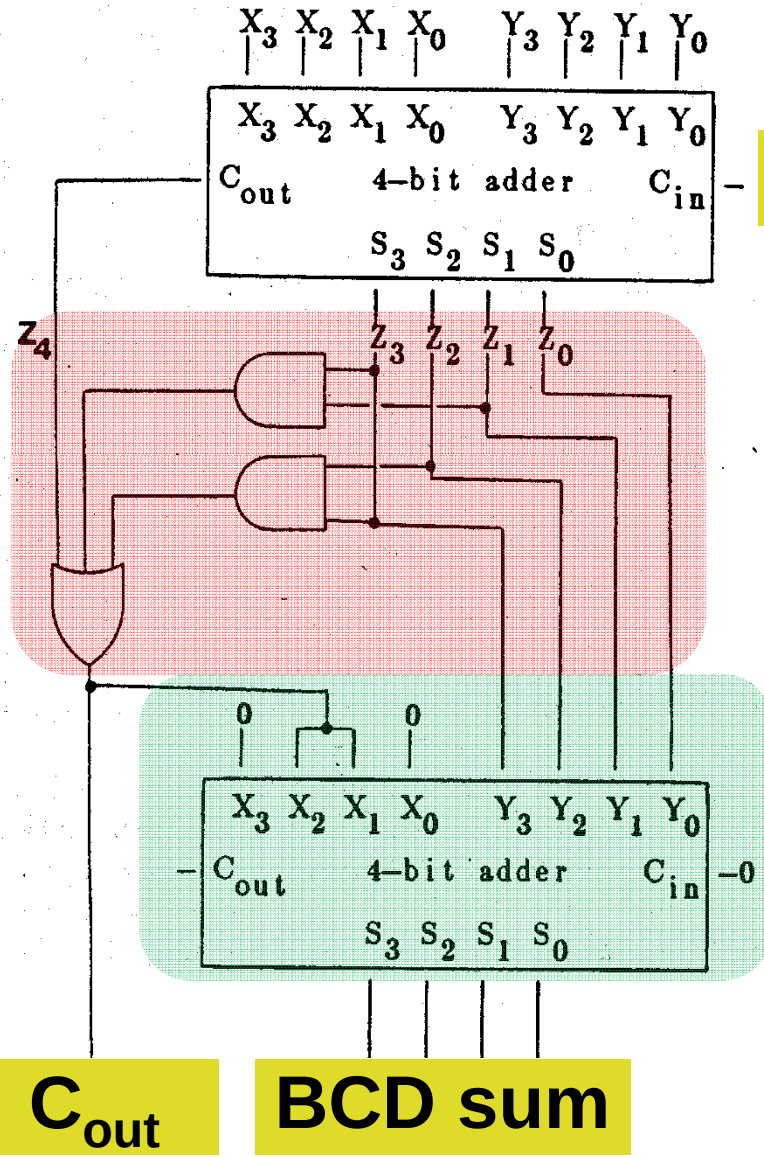
$$F_{correction} = C_{out} = Z4 + Z3 \cdot Z2 + Z3 \cdot Z1$$





BCD "Full Adder" Circuit

BCD operands

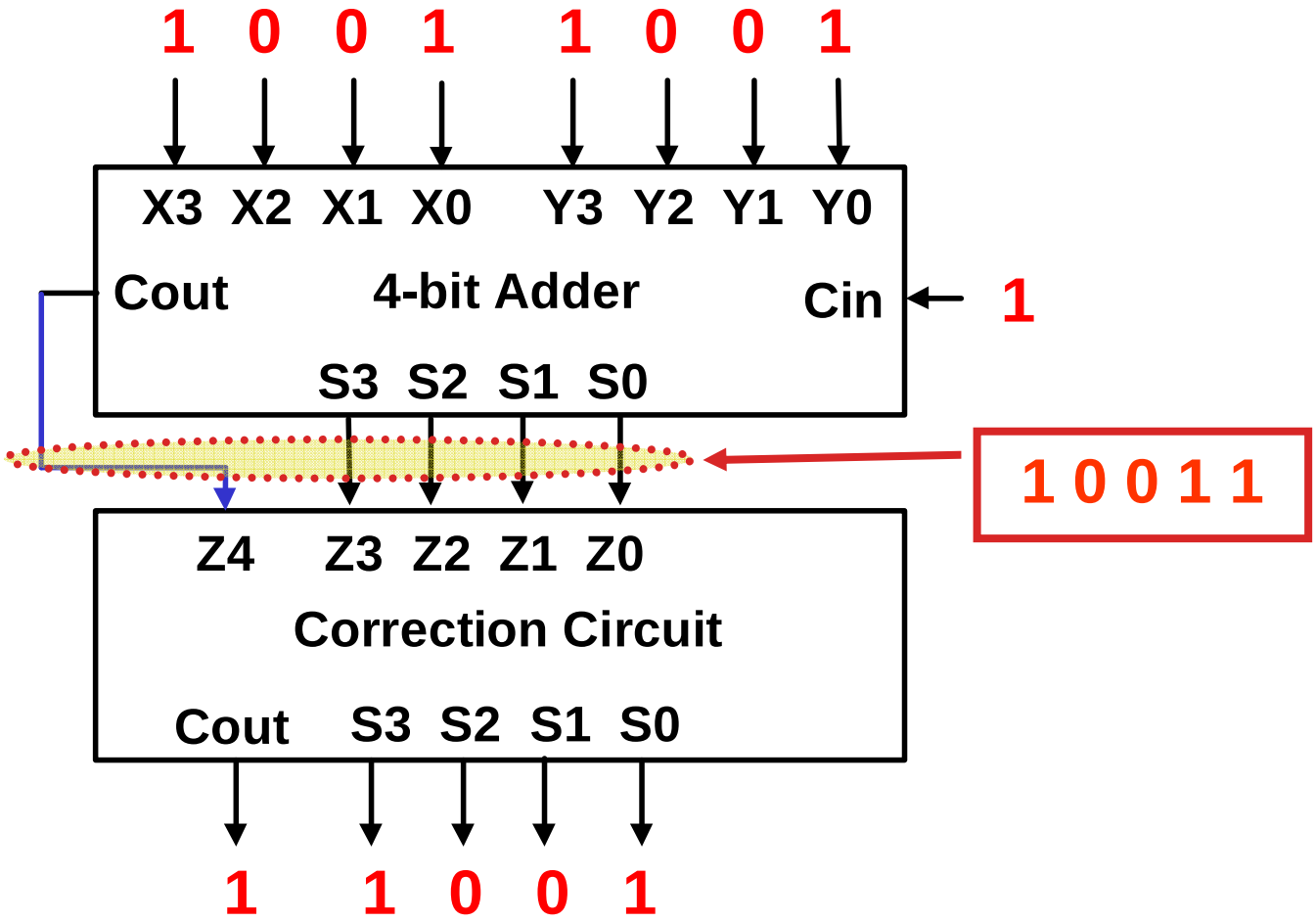


C_{in}

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

Add 0110 (6) to direct sum when $F_{correction} = 1$

Example: **The maximum value that can be generated by a BCD full adder cell is 19_{10}. Write the binary values (bit patterns) that depict this case.**



Clicker Quiz

Clicker Quiz

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**

Clicker Quiz

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

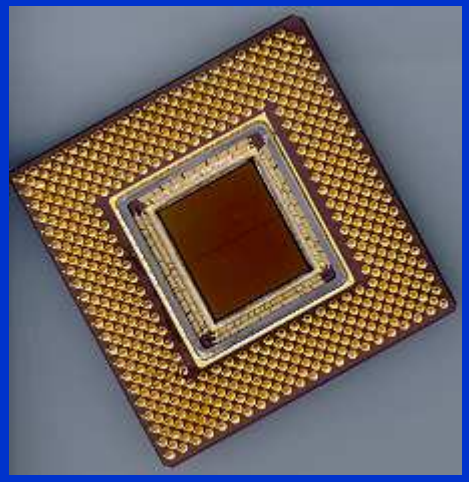
Decimal Adder Circuits

- **Thought questions:**
  - **How could an *n-digit* BCD adder be constructed using the "decimal full adder" circuit just designed?**
  - **How could this n-digit BCD adder be made into an n-digit BCD *adder/subtractor*?**

Hint: How could the *radix complement* of a BCD digit (where the radix or base is 10) be generated?

Example – ABEL program that generates the *diminished radix* (or 9's) 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
```



Introduction to Digital System Design

Module 4-G

Simple Computer Top-Down Specification

Reading Assignment:

Meyer Supplemental Text, pp. 1-18

Learning Objectives:

- **Define computer architecture, programming model, and instruction set**
- **Describe the top-down specification, bottom-up implementation strategy as it pertains to the design of a computer**
- **Describe the characteristics of a "two address machine"**
- **Describe the contents of memory: program, operands, results of calculations**
- **Describe the format and fields of a basic machine instruction (opcode and address)**
- **Describe the purpose/function of each basic machine instruction (LDA, STA, ADD, SUB, AND, HLT)**
- **Define what is meant by "assembly-level" instruction mnemonics**
- **Draw a diagram of a simple computer, showing the arrangement and interconnection of each functional block**

Outline

- Introduction
- Top-Down, Bottom-Up Design Methodology
- Simple Computer “Big Picture”
- A Simple Instruction Set
- A Simple Programming Example
- System Block Diagram

Introduction

- **The focus thus far has been on a number of digital system "building blocks"**
  - **state machines**
  - **latches and flip-flops**
  - **arithmetic logic circuits**
  - **decoders, encoders, multiplexers**
  - **basic gates (NAND, NOR, XOR, etc.)**
- **Question: What is the primary utility of these building blocks?**

Building a computer or interfacing to an existing one

Introduction

- **Question: What is a computer?**

A device that sequentially executes a stored program

- **Question: What is a microprocessor?**

A single-chip embodiment of the major functional blocks of a computer

- **Question: What is a microcontroller?**

A microprocessor with a number of integrated peripherals typically used in control-oriented applications

Introduction

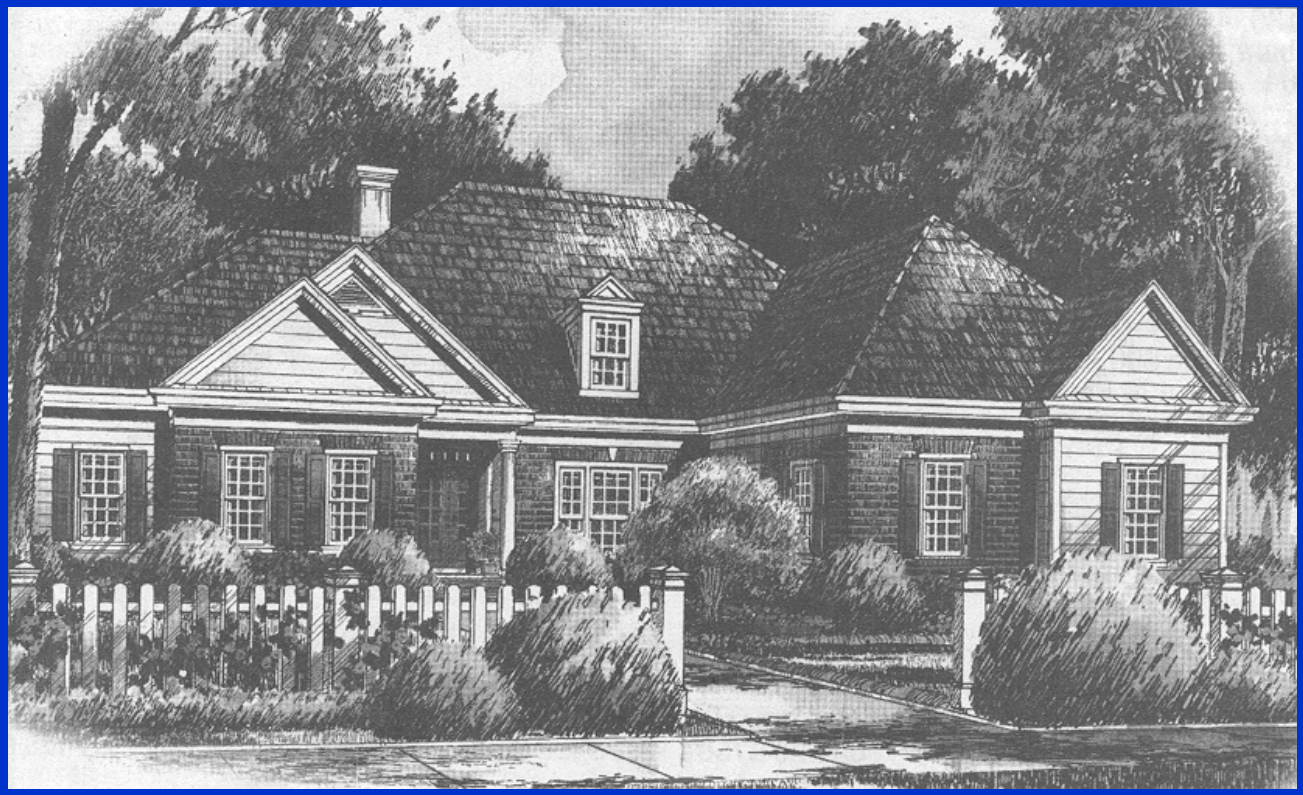
- **Question: How can we apply what we know about various digital system building blocks to design a simple computer?**

We need a *structured approach* that enables us to transform a "word" description of what a computer does into a block diagram

- **Definition: The *architecture* of a computer is the *arrangement* and *interconnection* of its functional blocks**

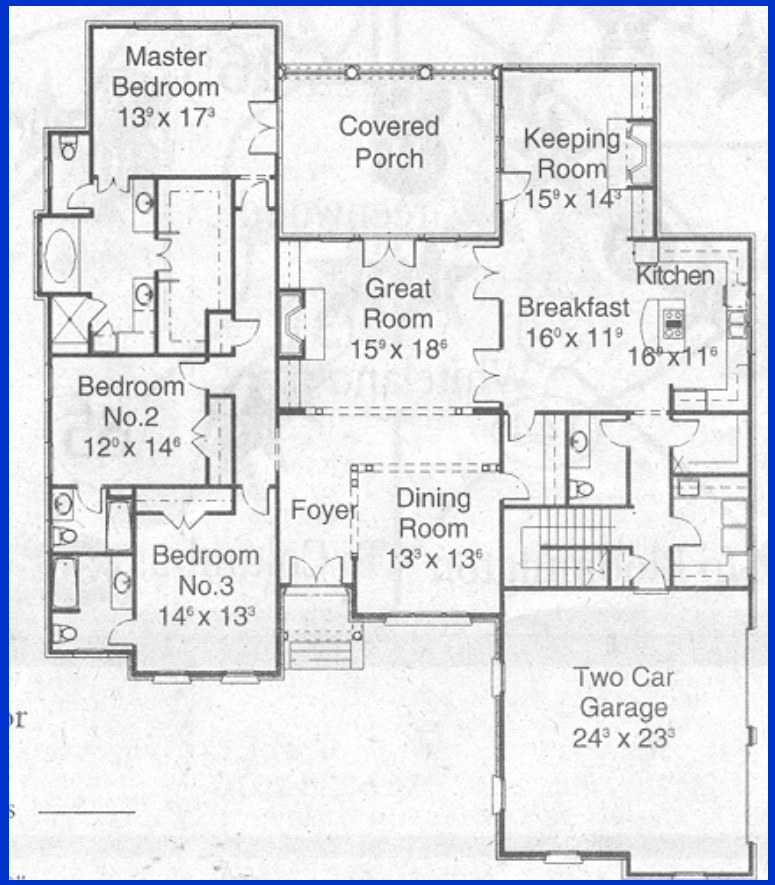
Introduction

- **Analogy: Designing and building a house - 1**
  - **Start with the “big picture”...**



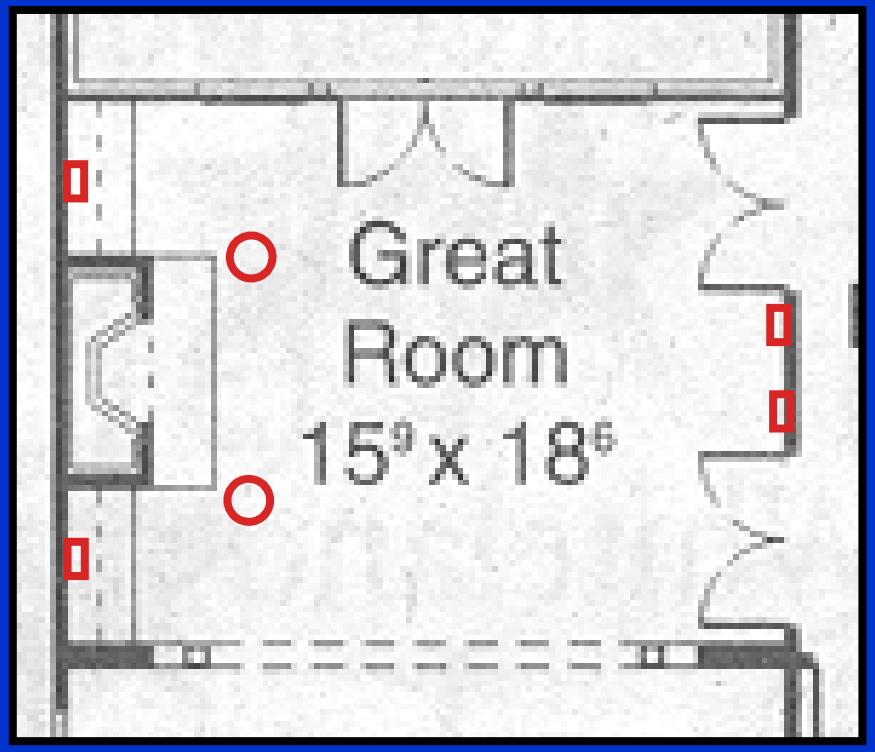
Introduction

- **Analogy: Designing and building a house - 2**
  - **Then develop a "floor plan"...**



Introduction

- **Analogy: Designing and building a house - 3**
  - **Then embellish the floor plan with details (outlets, lights, plumbing, HVAC, etc.)**



Introduction

- **Analogy: Designing and building a house - 4**
  - **When you're ready to build, how do you proceed?**

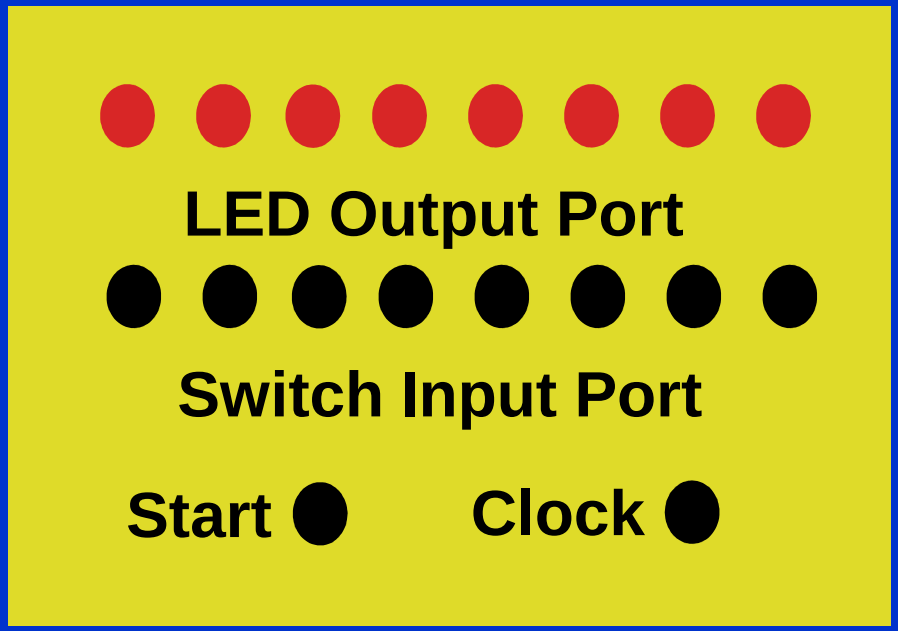
From the *ground-up* – start with the foundation, create basic structure, embellish with finishing details

- **Question: What would you call the basic procedure we have just described?**

***Top-down specification* of functionality, *bottom-up implementation* of basic blocks**

Simple Computer “Big Picture”

- **We wish to apply this “top-down, bottom-up” methodology to the design of a simple 8-bit computer (or, “microprocessor”)**
- **<u>First step</u>: Draw the *big picture***



Simple Computer "Floor Plan"

- **Question: In the design of a computer, what is analogous to the *floor plan* of a house?**

  The computer's *programming model* and *instruction set*

- **Definition: The *programming model* of a computer is the set of "user" registers available to the programmer**
- **Definition: A collection of two or more flip-flops with a common clock (and generally a common purpose) is called a *register***

In the simple computer designed here, the programming model will contain a single *data register* "where the result accumulates" (the *accumulator*, or "A register" for short), plus *condition codes* (CF, VF, NF, ZF)

Simple Computer "Floor Plan"

- **Definition: The *instruction set* of a computer is the set of operations the computer can be programmed to perform on data**
- **Instructions typically consist of several fields that indicate the *operation* to be performed ("operation code", or *opcode*) and the *data* on which the operation is to be performed (specified using an *addressing mode*)**
- **Our 8-bit computer will utilize a 3-bit opcode field (thus allowing 8 different kinds of instructions to be implemented) and a 5-bit address field (thus allowing 32 locations)**

Simple Computer "Floor Plan"

- **Instruction format:**

X X X Y Y Y Y Y

XXX – indicates operation to perform

YYYYY – indicates location of operand

Called a "two address machine" since one operand will be the accumulator ("A") register and the other operand will be obtained from the specified location in memory

Simple Computer "Floor Plan"

- **Instruction set:**

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

Note: We will use *parentheses* to denote the *contents* of a register or memory location, e.g., "(A)" is read as "the contents of A"

Simple Programming Example

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

Memory “Snapshot”

Location	Contents
00000	00001011
00001	01001100
00010	00101101
00011	00001011
00100	10001100
00101	00101110
00110	00001011
00111	01101100
01000	00101111
01001	10100000
01010	
01011	10101010
01100	01010101
01101	
01110	
01111	

Memory “Snapshot”

Program

Operands

Results

Location	Contents
00000	00001011
00001	01001100
00010	00101101
00011	00001011
00100	10001100
00101	00101110
00110	00001011
00111	01101100
01000	00101111
01001	10100000
01010	
01011	10101010
01100	01010101
01101	11111111
01110	
01111	

← Add

Add:

```
  10101010
+ 01010101
  11111111
```

CF = 0

NF = 1

VF = 0

ZF = 0

Location	Contents
00000	00001011
00001	01001100
00010	00101101
00011	00001011
00100	10001100
00101	00101110
00110	00001011
00111	01101100
01000	00101111
01001	10100000
01010	
01011	10101010
01100	01010101
01101	11111111
01110	00000000
01111	

← AND

AND:

10101010

∩01010101

00000000

CF = <unaffected>

NF = 0

VF = <unaffected>

ZF = 1

Location	Contents
00000	00001011
00001	01001100
00010	00101101
00011	00001011
00100	10001100
00101	00101110
00110	00001011
00111	01101100
01000	00101111
01001	10100000
01010	
01011	10101010
01100	01010101
01101	11111111
01110	00000000
01111	01010101

← Sub

CF = 0
NF = 0
VF = 1
ZF = 0

Sub:

```
  10101010
-01010101
```

```
  10101010
  10101010
+        1
-----------
1)01010101
```

Overflow!

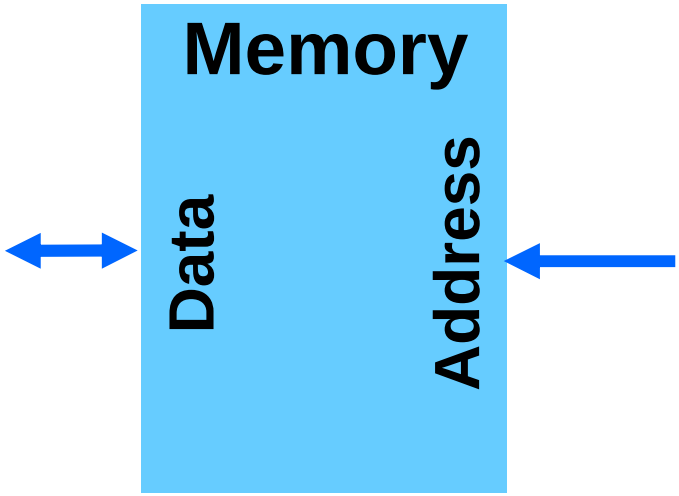
Simple Computer Block Diagram

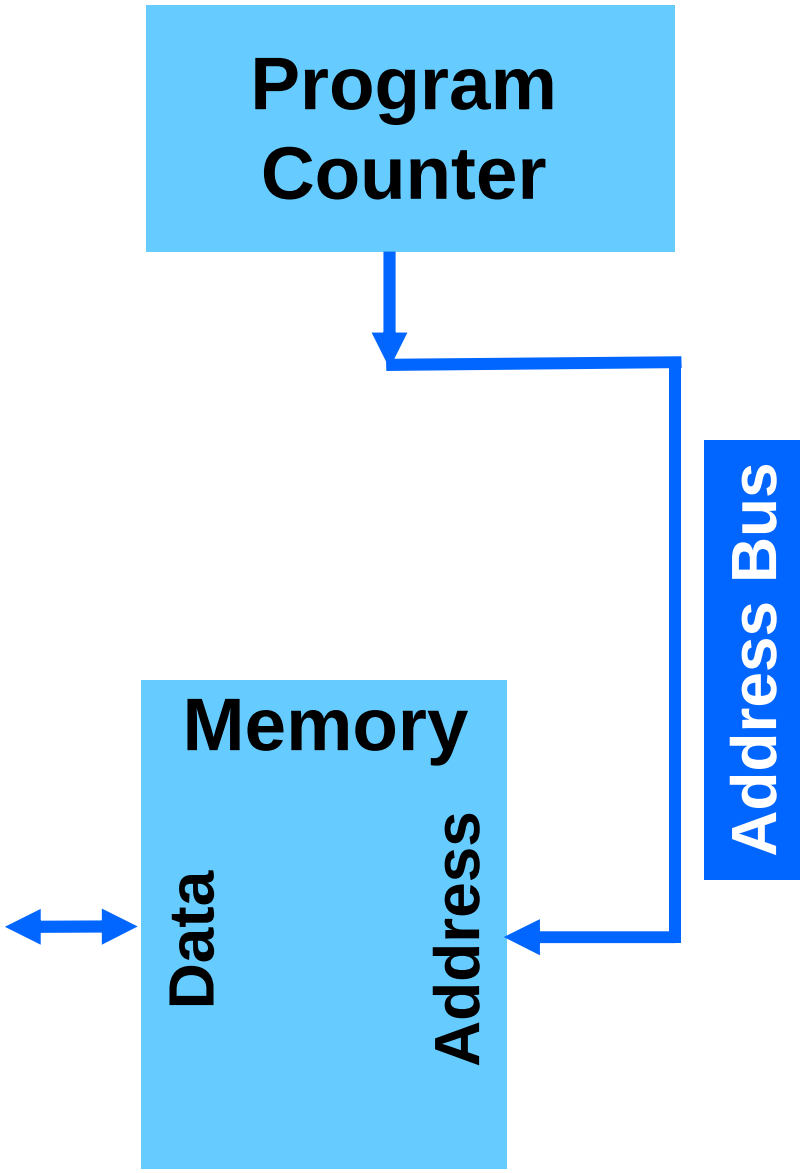
- **Question: What functional blocks are necessary to implement a computer that executes a stored program consisting of the instructions we have just defined?**

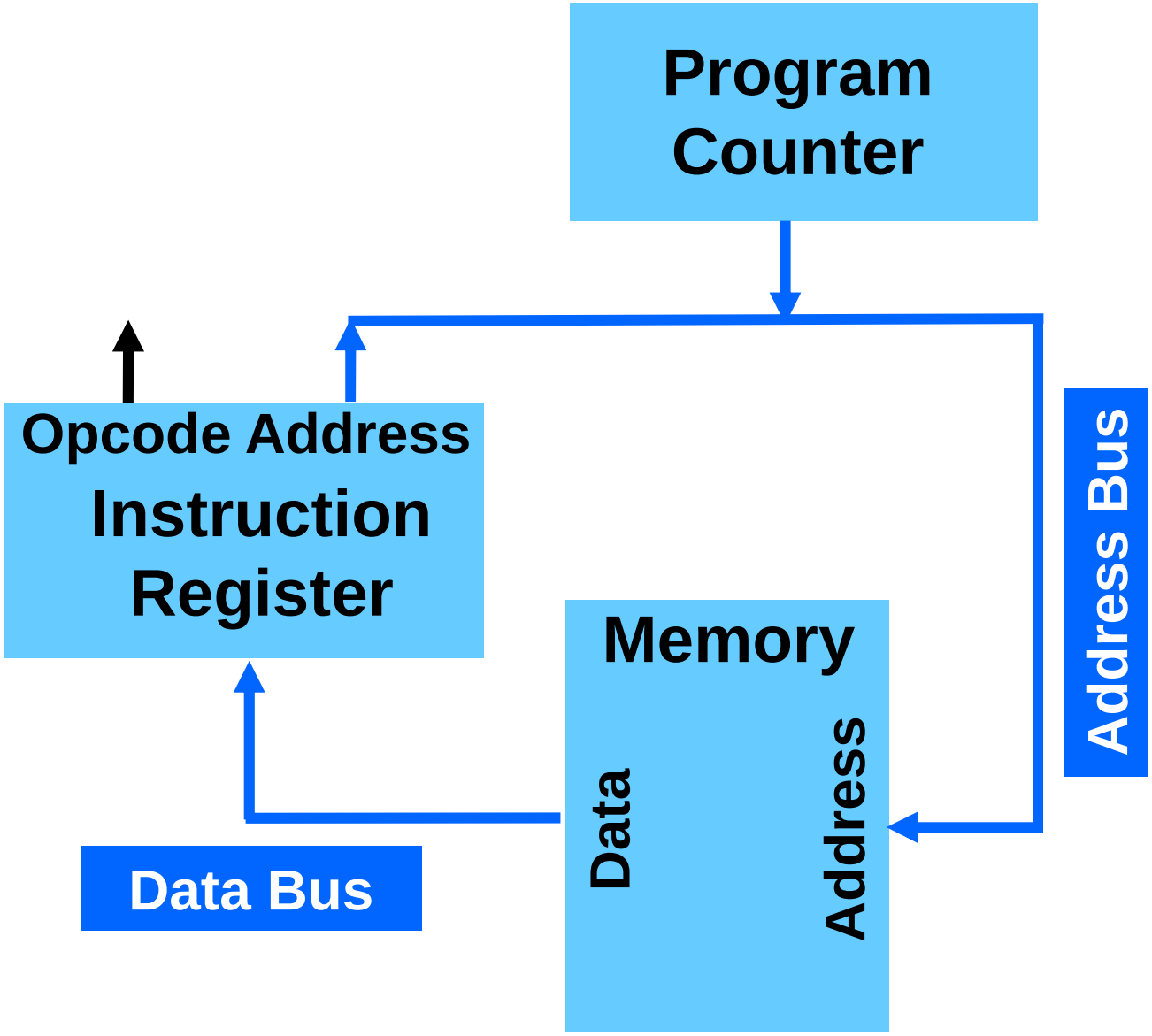
Two basic steps are required to perform an instruction: (1) it must be *fetched* from memory, and (2) it must be *decoded and executed*

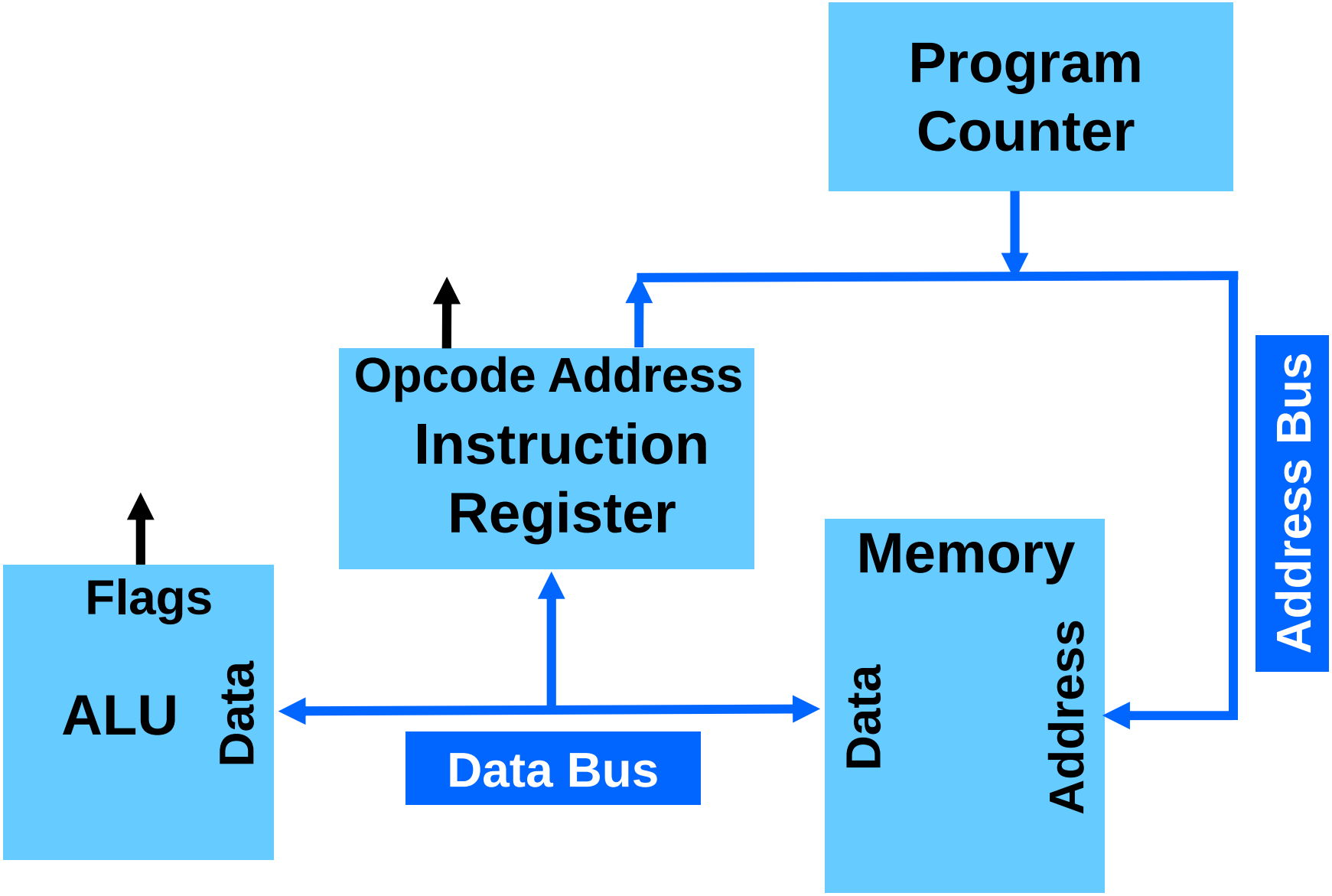
Simple Computer Block Diagram

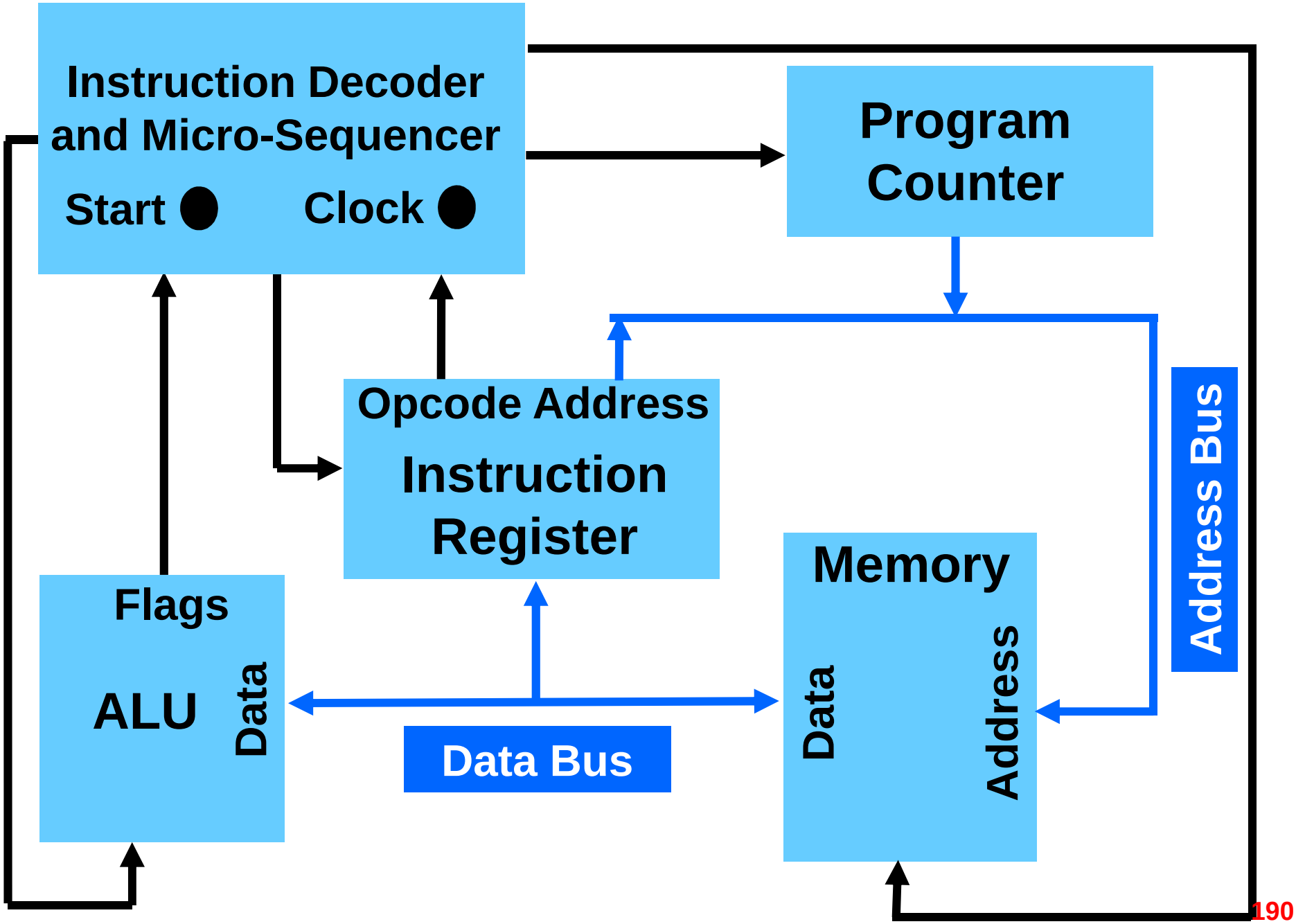
- **Functional blocks required:**
  - **a place to store the program, operands, and computation results – *memory***
  - **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Memory
Data
Address


Program Counter
Address Bus
Memory
Data
Address


Program Counter
Opcode Address
Instruction Register
Memory
Data
Address
Address Bus
Data Bus


Program Counter
Opcode Address
Instruction Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

Notes About Block Diagram

- **Each functional block is “self-contained” (which means each block can be designed and tested *independently*)**
- **Additional instructions can be added by increasing the number of *opcode* bits**
- **Additional memory can be added by increasing the number of *address* bits**
- **The numeric range can be expanded by increasing the number of *data* bits**

Clicker Quiz

Q1. 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

Q2. The next instruction to fetch from memory is pointed to by the:

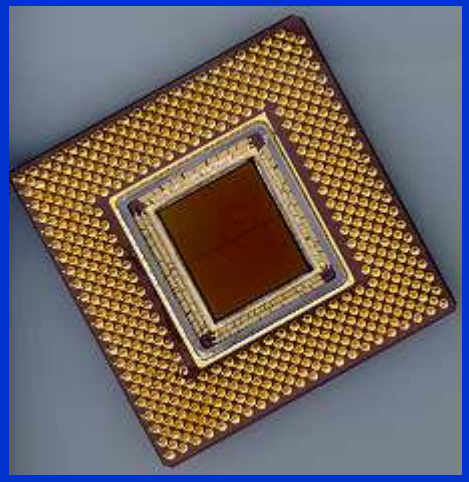
A. accumulator
B. program counter
C. instruction register
D. microsequencer
E. none of the above

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



Introduction to Digital System Design

Module 4-H

Simple Computer Instruction Execution Tracing

Reading Assignment:

Meyer Supplemental Text, pp. 18-24

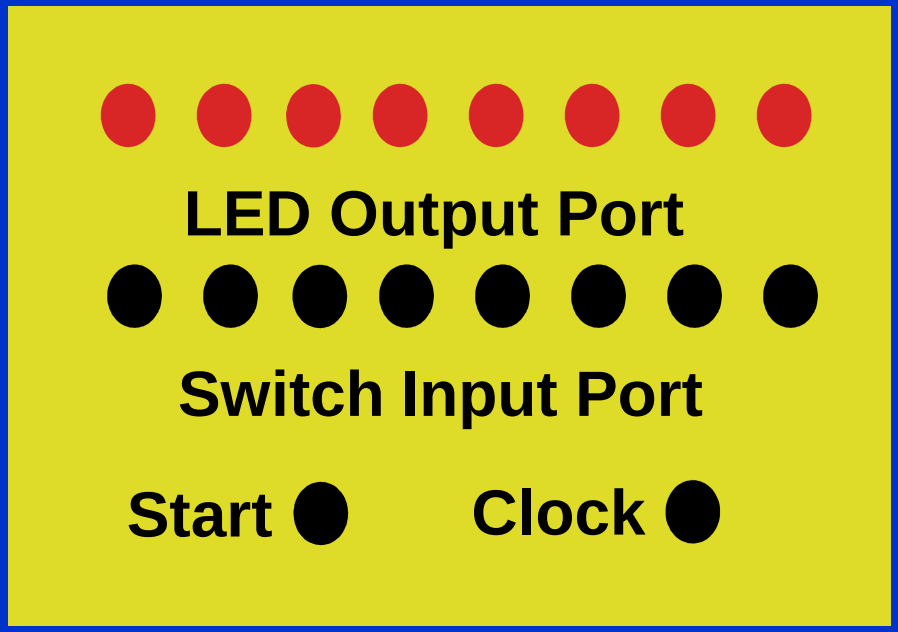
Learning Objectives:

- **Trace the execution of a computer program, identifying each step of an instruction's microsequence (fetch and execute cycles)**
- **Distinguish between synchronous and combinational system control signals**

Outline

- Review of top-down specification phase of design process
  - Big picture
  - Floor plan (instruction set)
  - Block diagram
- Instruction execution tracing

Simple Computer “Big Picture”



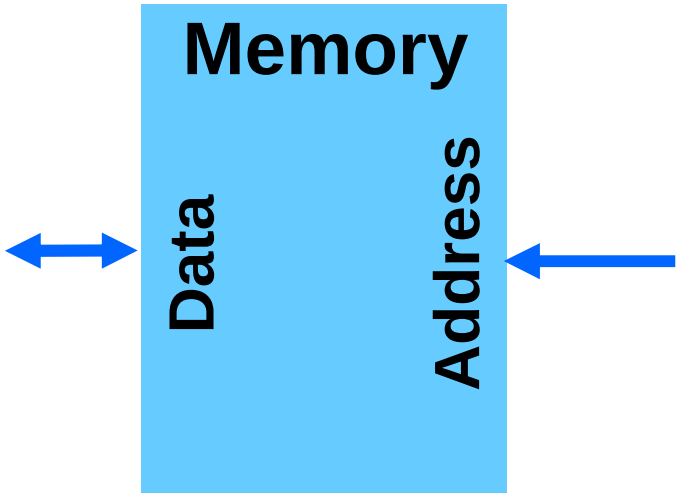
Simple Computer "Floor Plan"

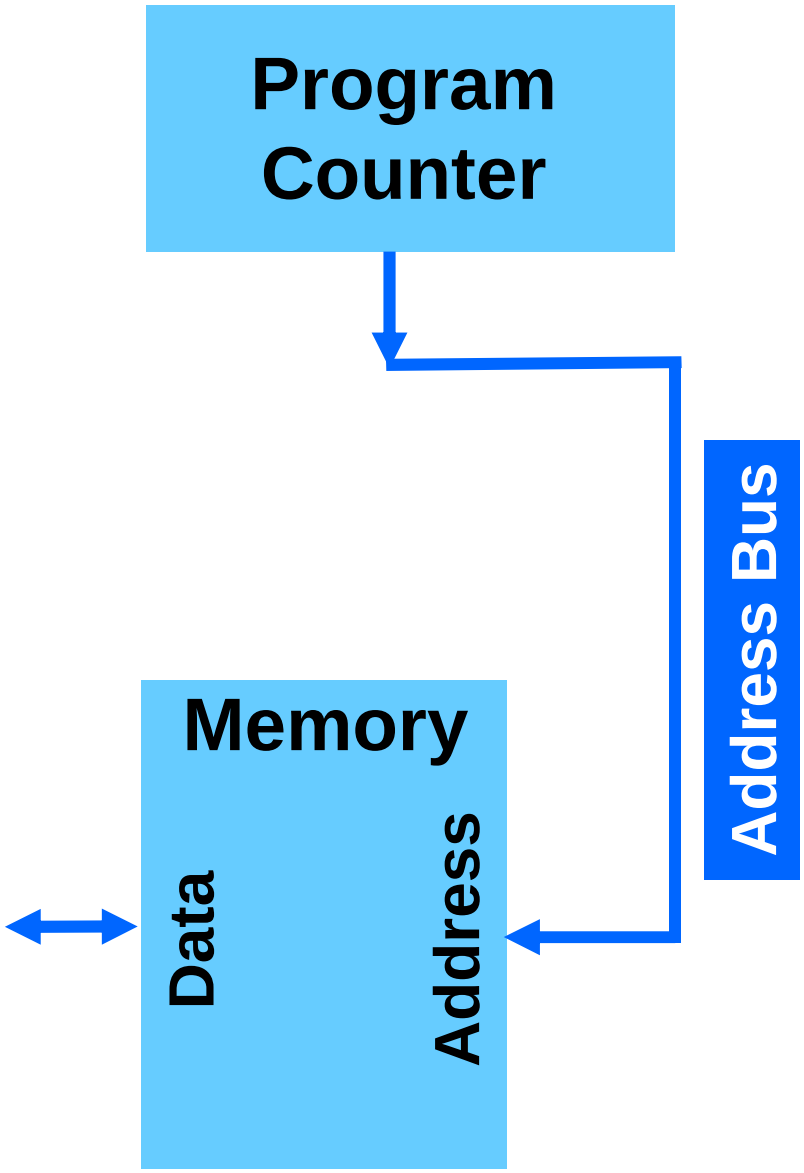
- **Instruction set:**

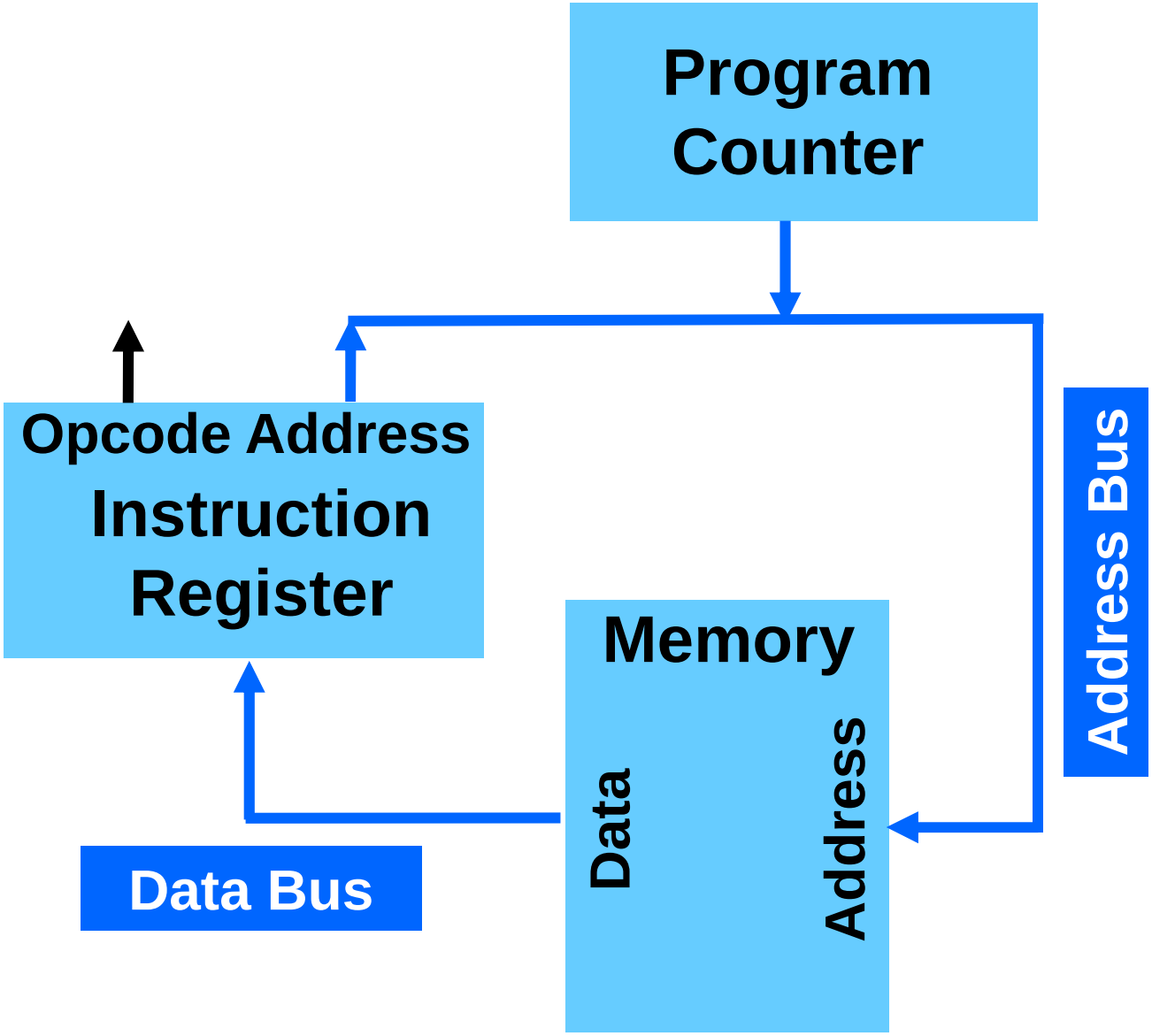
Opcode	Mnemonic	Function Performed
0 0 0	LDA *addr*	Load A with contents of location *addr*
0 0 1	STA *addr*	Store contents of A at location *addr*
0 1 0	ADD *addr*	Add contents of *addr* to contents of A
0 1 1	SUB *addr*	Subtract contents of *addr* from contents of A
1 0 0	AND *addr*	AND contents of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

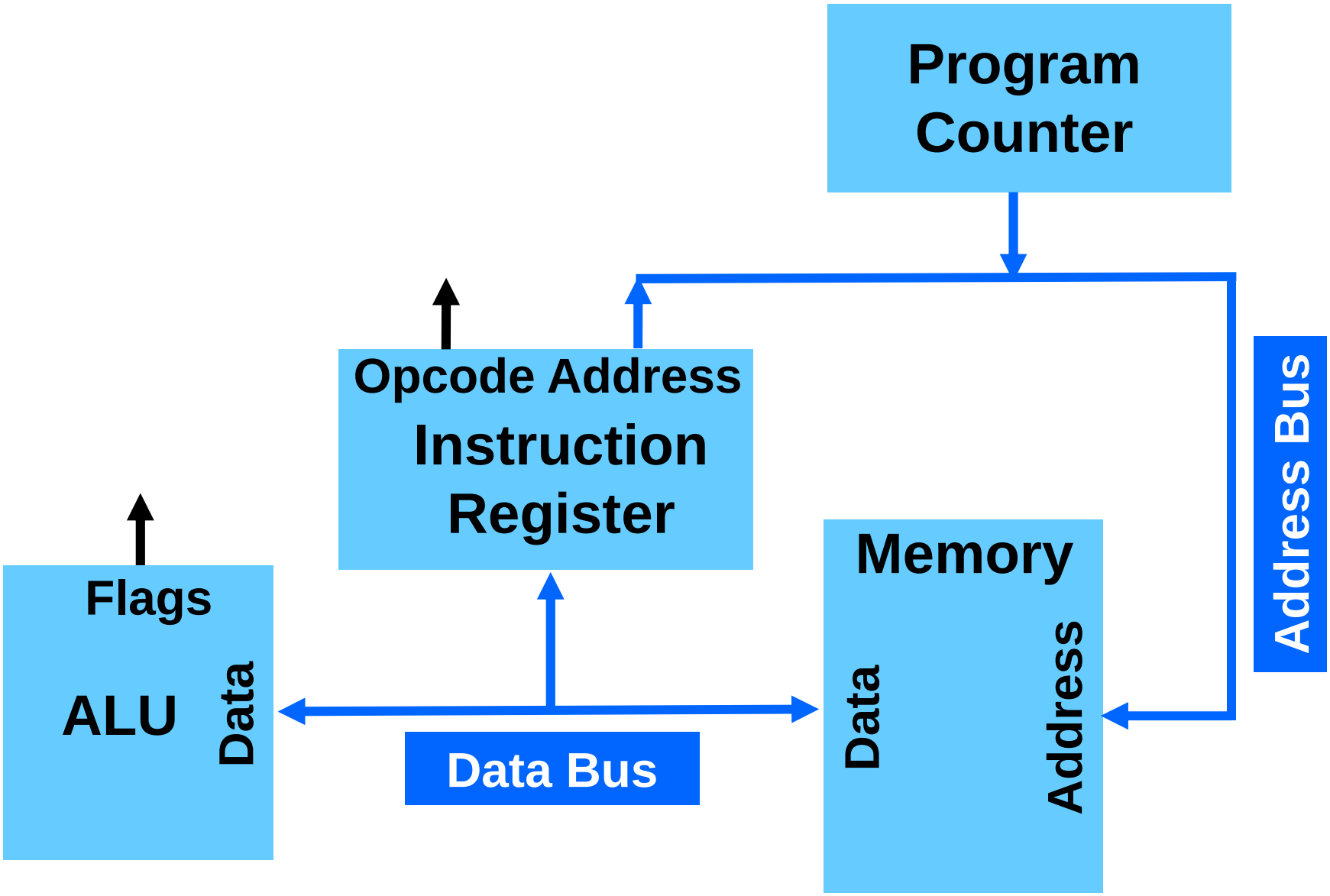
Simple Computer Block Diagram

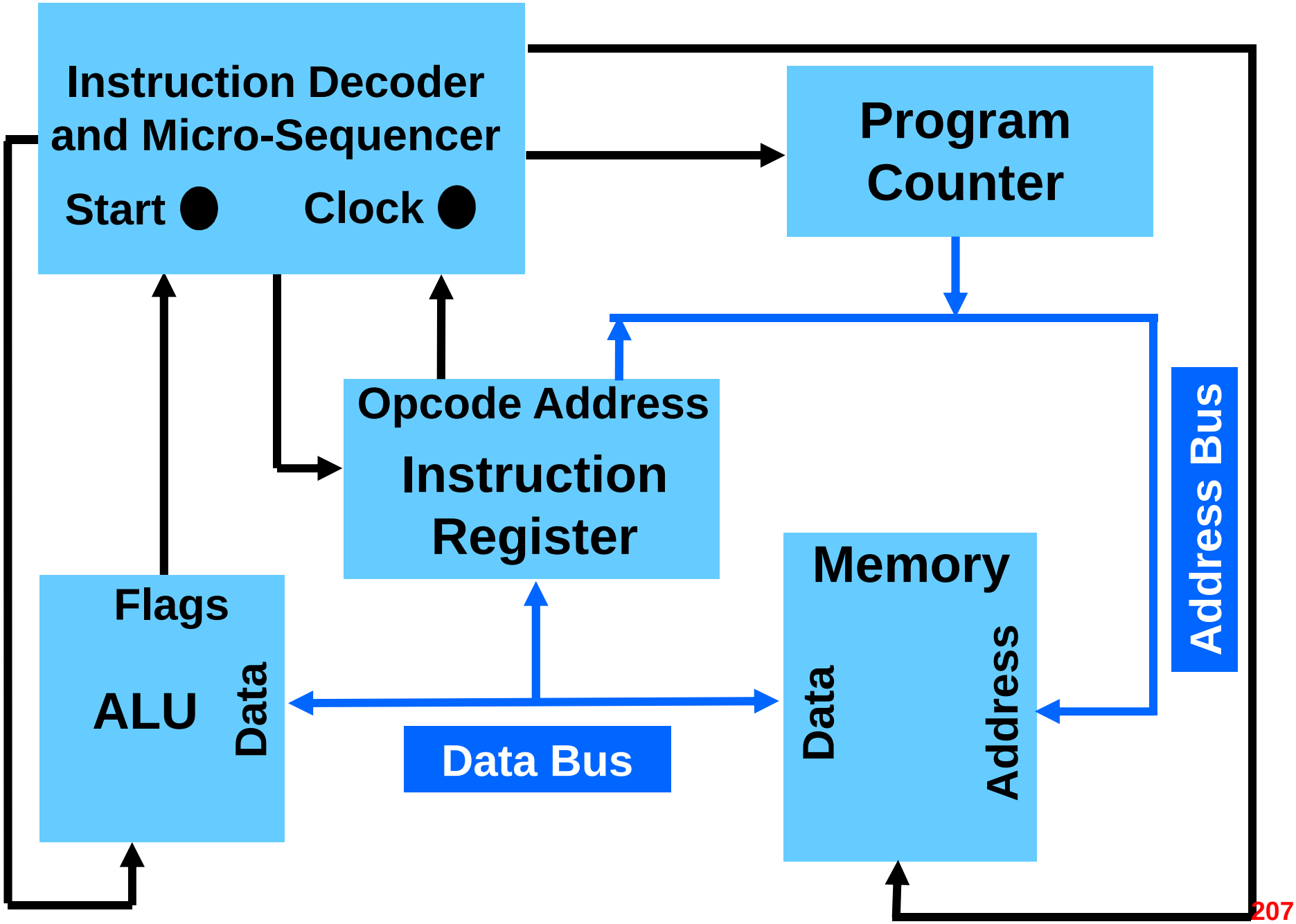
- **Functional blocks required:**
  - **a place to store the program, operands, and computation results – *memory***
  - **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Memory
Data
Address


Program Counter
Address Bus
Memory
Data
Address


Program Counter
Opcode Address
Instruction Register
Memory
Data
Address
Address Bus
Data Bus


Program Counter
Opcode Address
Instruction Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

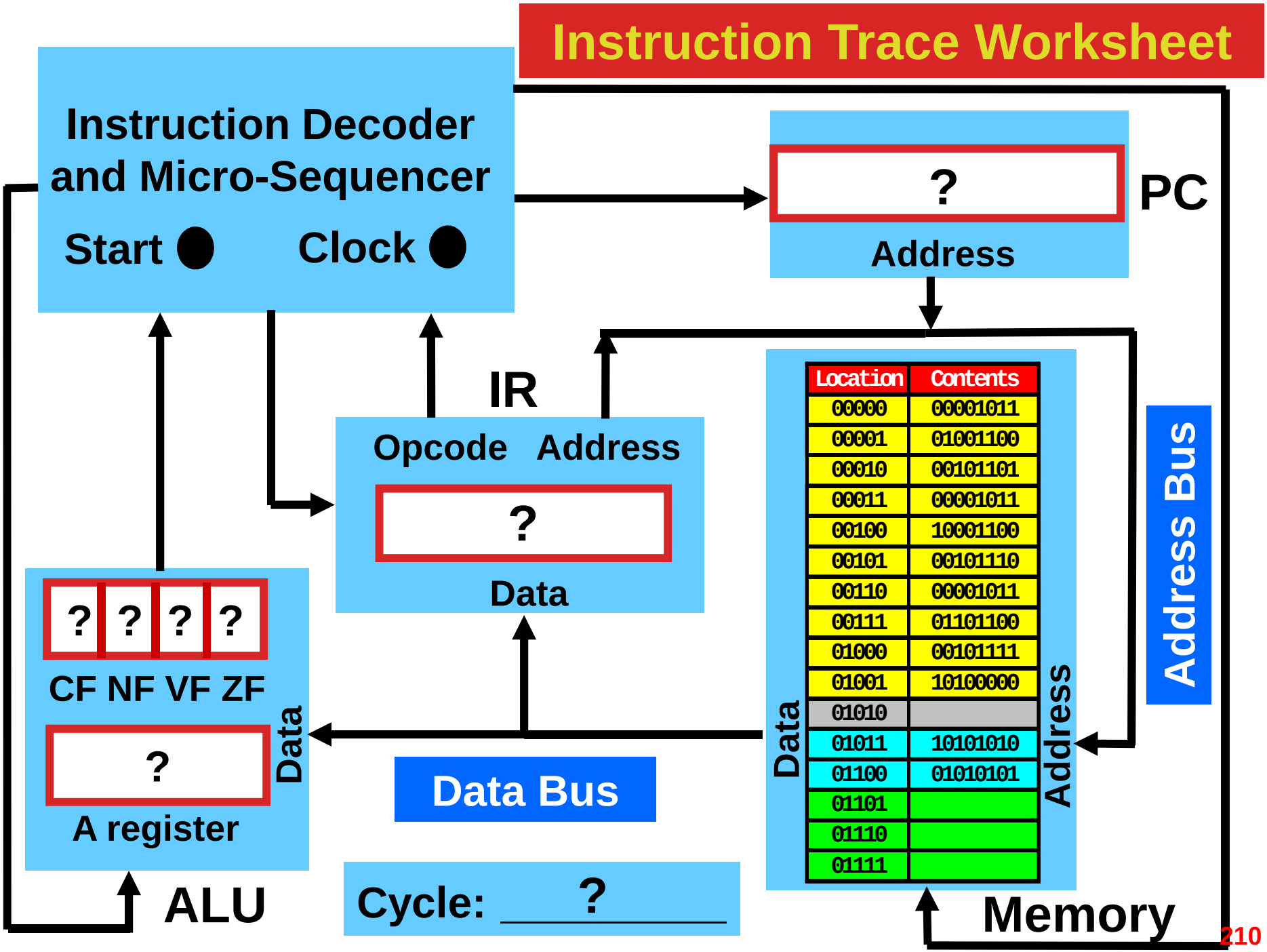

Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

Instruction Tracing

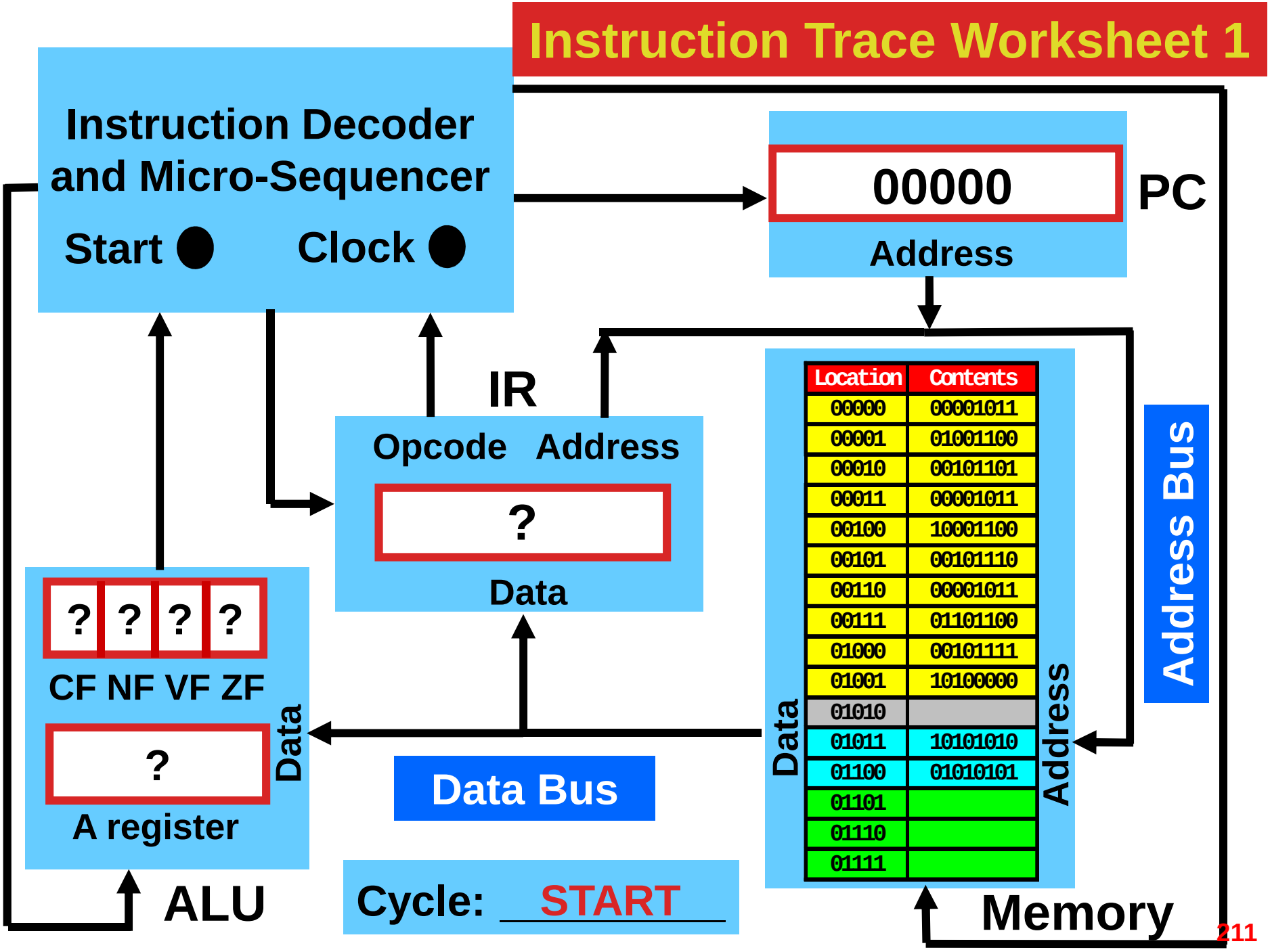
- **In our simple computer, there are two basic steps (or phases) in "processing" an instruction:**
  - **Step 1: The instruction pointed to by the PC is *fetched* from memory and loaded into the IR**
  - **Step 2: Based on the opcode field of the instruction in the IR, the specified operation is *executed* by the ALU on the data pointed by address field of the instruction in the IR**
- **To better understand fetch/execute cycles, we wish to *trace* through the processing of several instructions**

Simple Programming Example

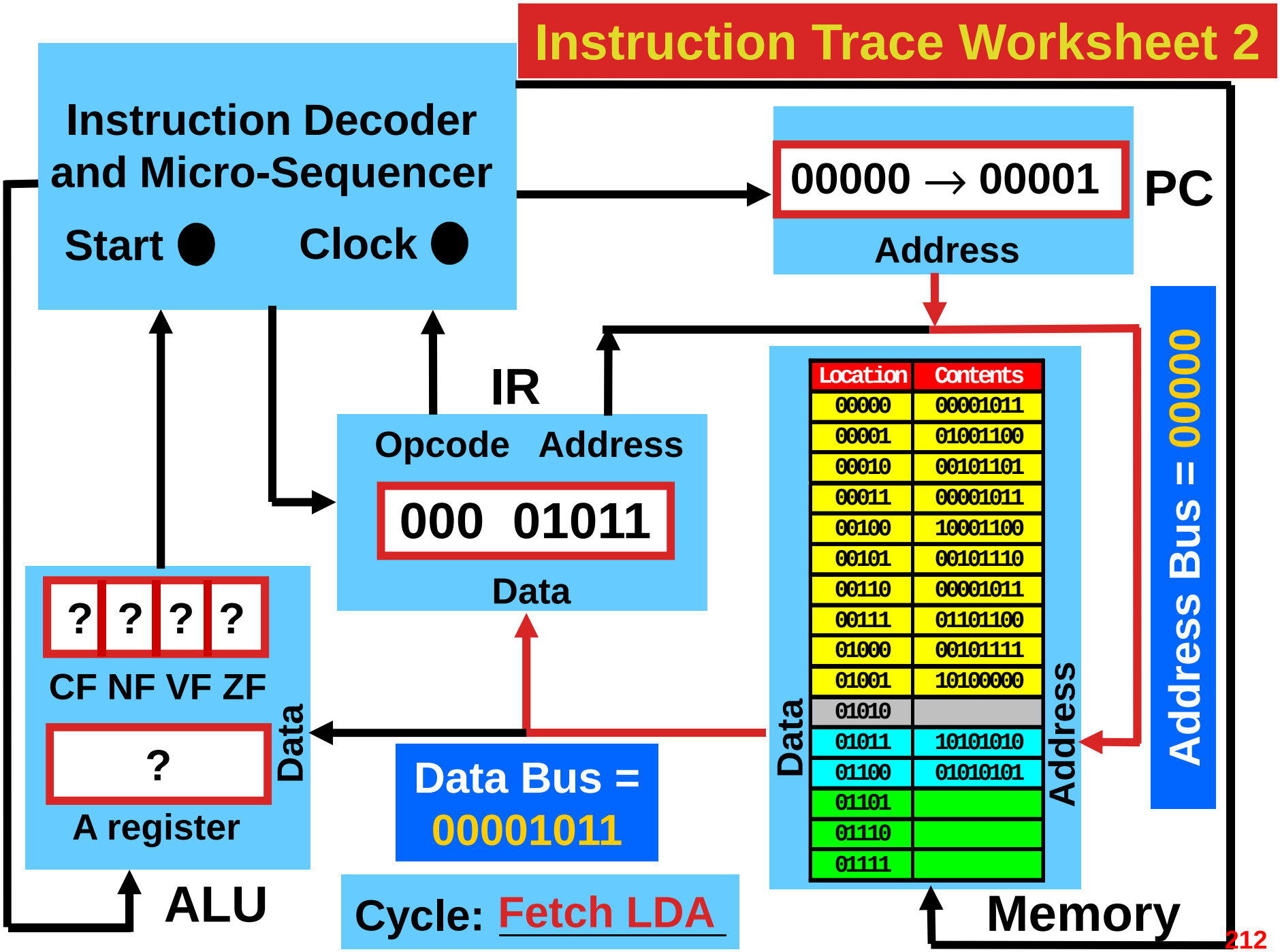
Addr	Instruction	Comments
00000	**LDA 01011**	Load A with contents of location **01011**
00001	**ADD 01100**	Add contents of location **01100** to A
00010	**STA 01101**	Store contents of A at location **01101**
00011	**LDA 01011**	Load A with contents of location **01011**
00100	**AND 01100**	AND contents 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


Instruction Trace Worksheet
Instruction Decoder and Micro-Sequencer
Start
Clock
?
PC
Address
IR
Opcode Address
?
Data
? ? ? ?
CF NF VF ZF
?
A register
Data
ALU
Data Bus
Cycle: ?
Location Contents
00000 00001011
00001 01001100
00010 00101101
00011 00001011
00100 10001100
00101 00101110
00110 00001011
00111 01101100
01000 00101111
01001 10100000
01010
01011 10101010
01100 01010101
01101
01110
01111
Data
Address
Address Bus
Memory
210

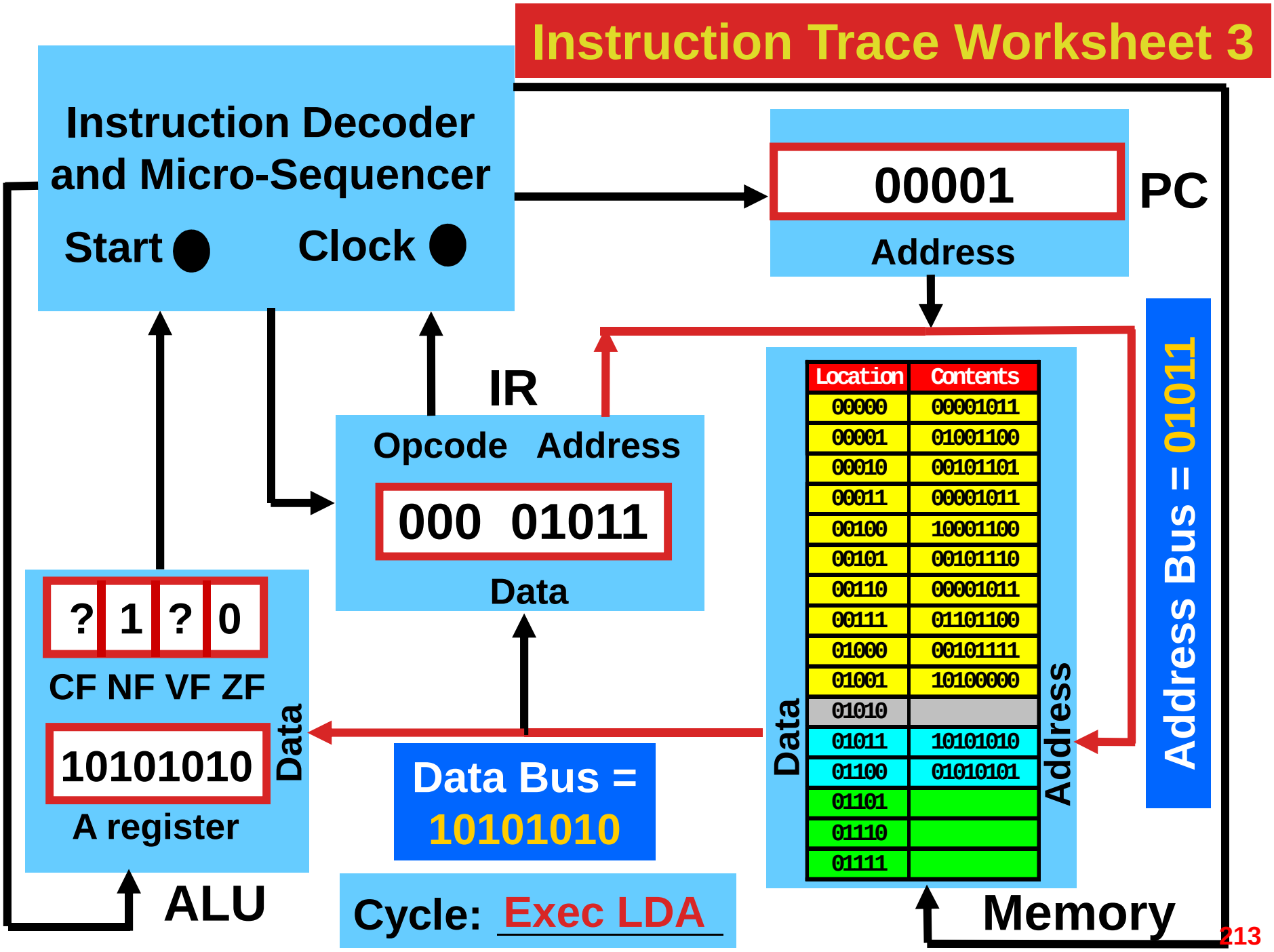
Instruction Trace Worksheet 1



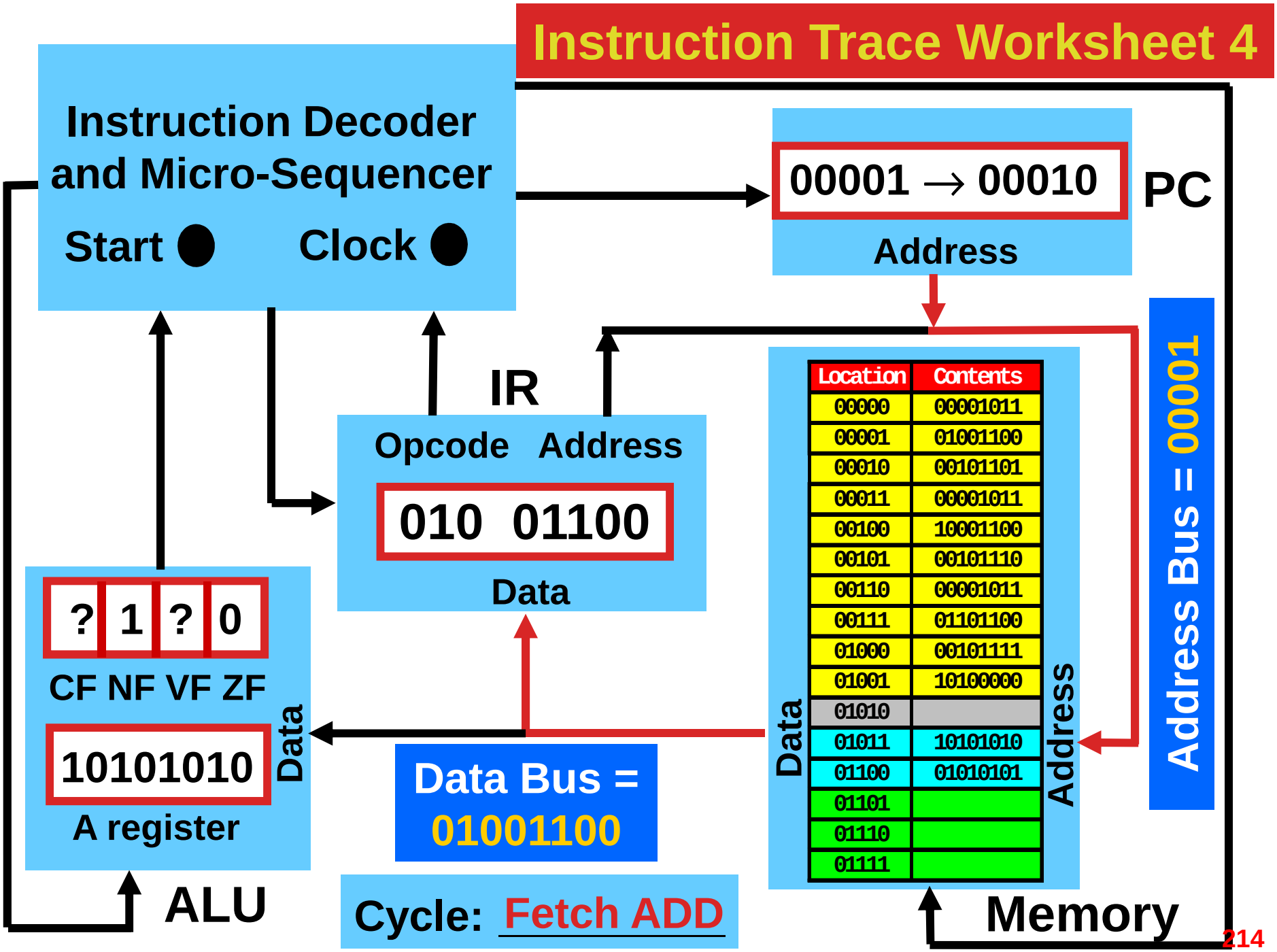
Instruction Trace Worksheet 2

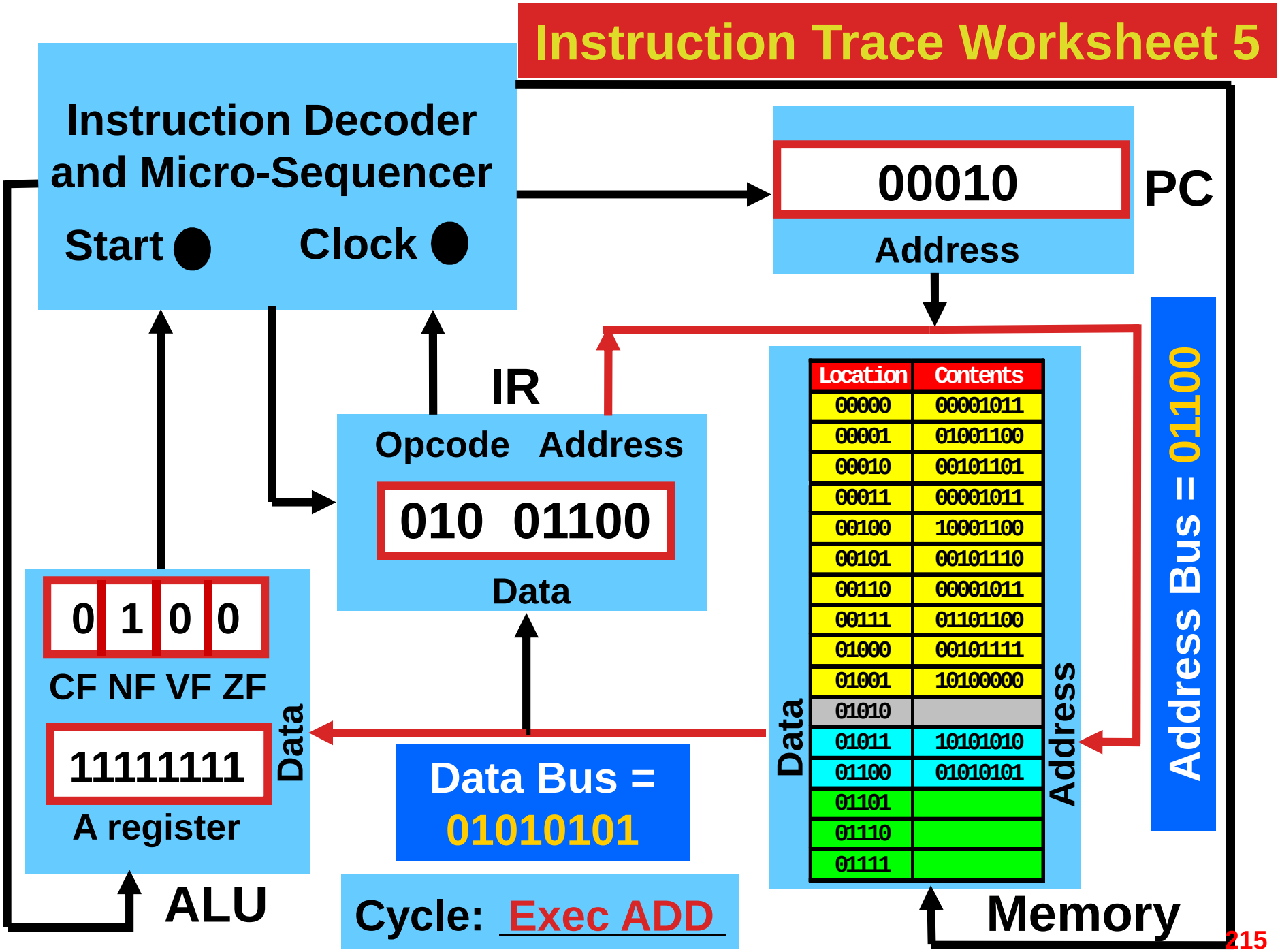


Instruction Trace Worksheet 3

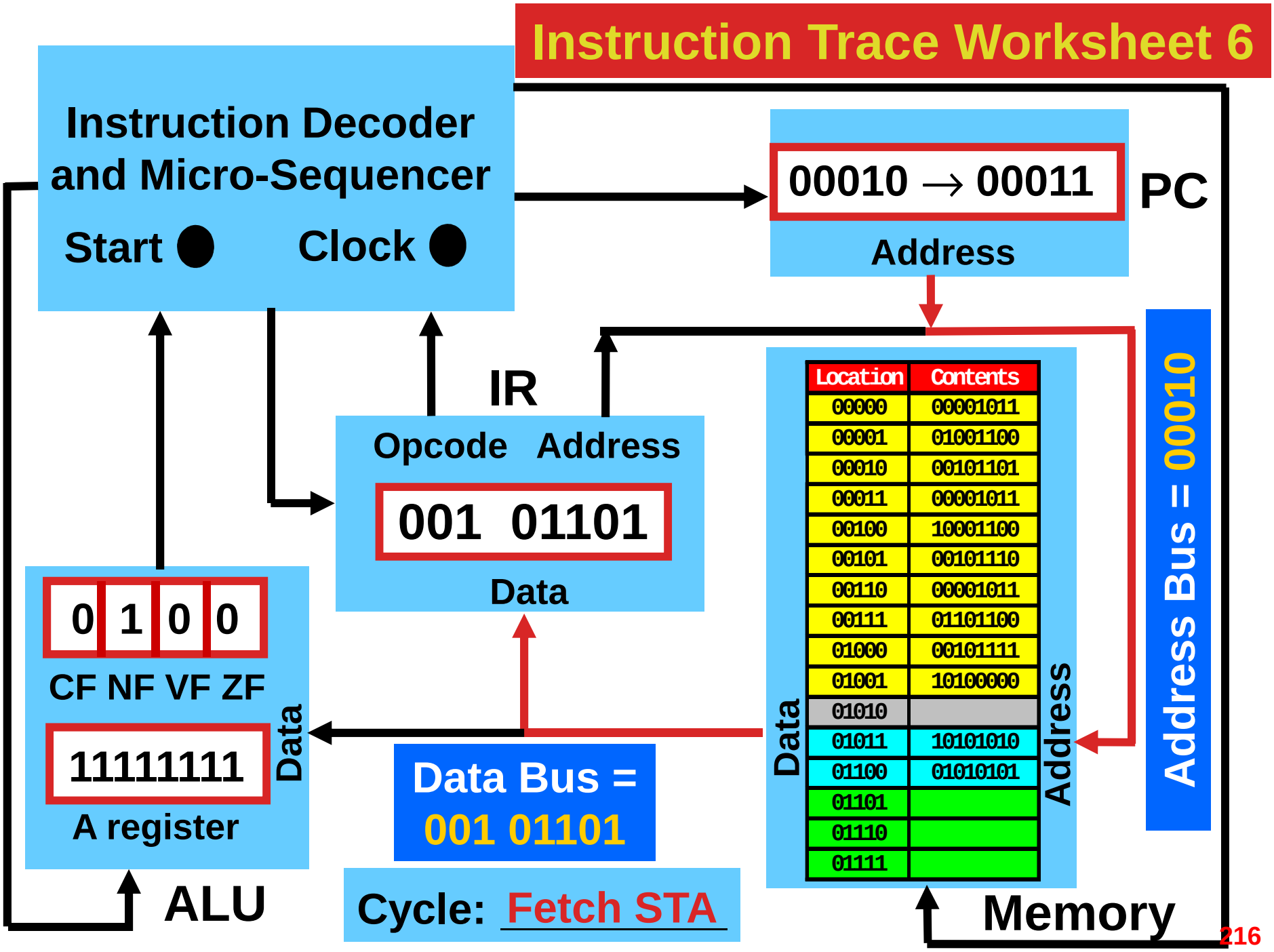


Instruction Trace Worksheet 4

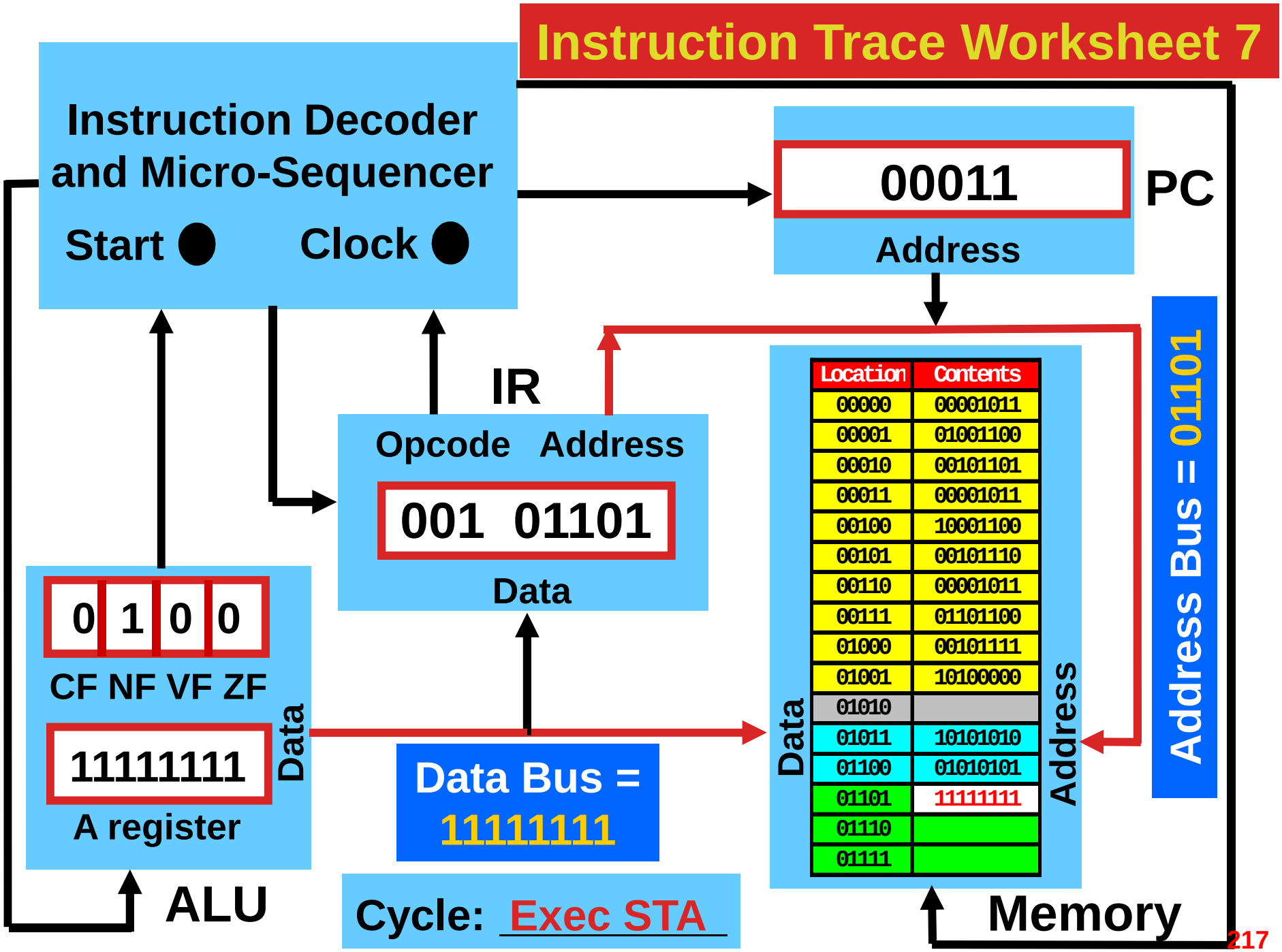



Instruction Trace Worksheet 5
Instruction Decoder and Micro-Sequencer
Start
Clock
00010
PC
Address
IR
Opcode Address
010 01100
Data
0 1 0 0
CF NF VF ZF
11111111
A register
Data
ALU
Data Bus = 01010101
Cycle: Exec ADD
Location Contents
00000 00001011
00001 01001100
00010 00101101
00011 00001011
00100 10001100
00101 00101110
00110 00001011
00111 01101100
01000 00101111
01001 10100000
01010
01011 10101010
01100 01010101
01101
01110
01111
Data
Address
Address Bus = 01100
Memory
215

Instruction Trace Worksheet 6

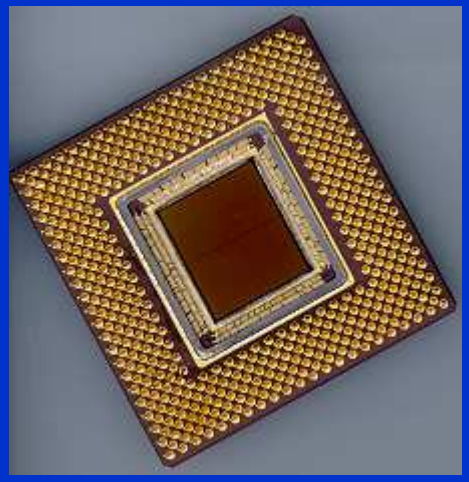


Instruction Trace Worksheet 7



Notes About Instruction Tracing

- **The clock edges drive the *synchronous* functions of the computer (e.g., increment program counter, load instruction register)**
- **The decoded states (here, fetch and execute) enable the *combinational* functions of the computer (e.g., turn on tri-state buffers)**



Introduction to Digital System Design

Module 4-I

Simple Computer Bottom-Up Realization

Reading Assignment:

Meyer Supplemental Text, pp. 24-42

Learning Objectives:

- **Describe the operation of memory and the function of its control signals: MSL, MOE, and MWE**
- **Describe the operation of the program counter (PC) and the function of its control signals: ARS, PCC, and POA**
- **Describe the operation of the instruction register (IR) and the function of its control signals: IRL and IRA**
- **Describe the operation of the ALU and the function of its control signals: ALE, ALX, ALY, and AOE**
- **Describe the operation of the instruction decoder/microsequencer and derive the system control table**
- **Describe the basic hardware-imposed system timing constraints**
- **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**

Outline

- Bottom-up Realization Phase of Design Process
  - Memory
  - Program Counter
  - Instruction Register
  - Arithmetic Logic Unit
  - Instruction Decoder and Microsequencer
- System Data Flow and Timing Analysis

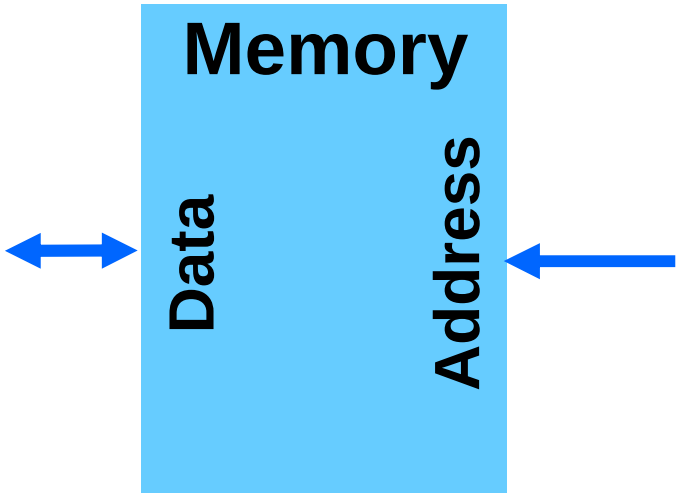
Bottom-Up Implementation

- **Having finished the "top-down" *specification* phase of the design process, we are now ready to *implement* each block identified from the "bottom-up"**
- **Note that, in practice, an important aspect of this process is to *independently test* (and debug) each block (or module) of the system *as it is implemented***
- **If each module is independently tested and verified as it is implemented, then – when the modules are assembled together into a system – there is a much higher probability that it will "work the first time"!**

Simple Computer Block Diagram

- **Functional blocks required (review):**
  - **a place to store the program, operands, and computation results – *memory***
  - **– a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **– a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **– a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **– a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***

Read/Write Memory (RWM)



Read/Write Memory (RWM)

- **The name *read/write memory* (RWM) is given to memory arrays in which we can store and retrieve information at any time**
- **Most of the RWMs used in digital systems are *random-access memories* (RAMs), which means that the time it takes to read or write a bit of memory is independent of the bit's location in the RAM**
- **In a static RAM (SRAM), once data is written to a given location, it remains stored as long as *power* is applied to the chip**
- **If power is removed, data is lost – this is referred to as a *volatile memory***

Read/Write Memory (RWM)

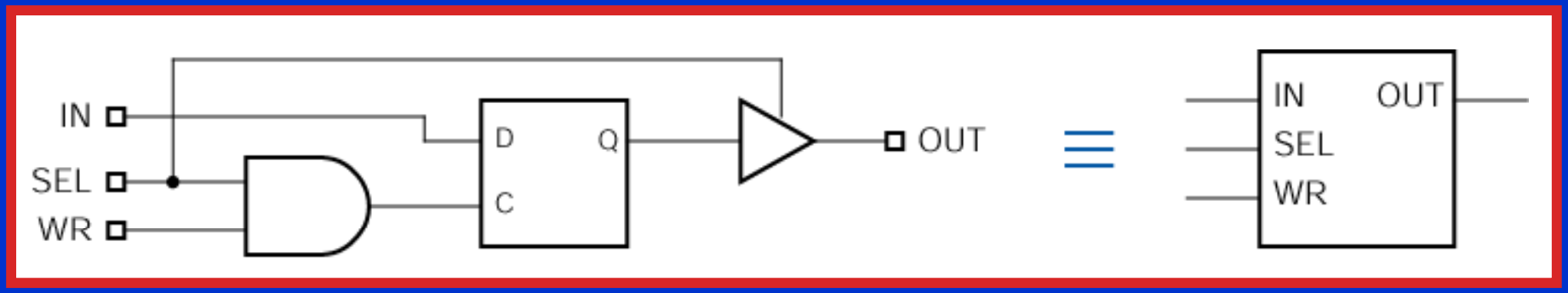
- **An SRAM has three (typically *active low*) control inputs:**
  - **a *chip select* (CS) signal that serves as the overall enable for the memory chip**
  - **an *output enable* (OE) signal that tells the memory chip to drive the data output lines with the contents of the memory location specified on its address lines**
  - **a *write enable* (WE) signal that tells the memory chip to write the data supplied on its data input lines at the memory location specified on its address lines**

Read/Write Memory (RWM)

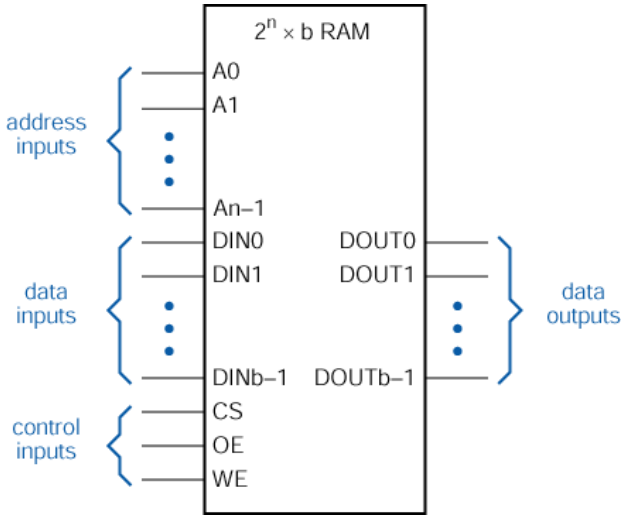
- **SRAM normally has two *access operations:***
  - **READ: An address is placed on the address lines while CS and OE are asserted; the latch outputs for the selected location are output on the data lines**
  - **WRITE: An address is placed on the address lines, data is placed on the data lines, then CS and WE are asserted; the latches of the selected location open, and the data is stored**

Read/Write Memory (RWM)

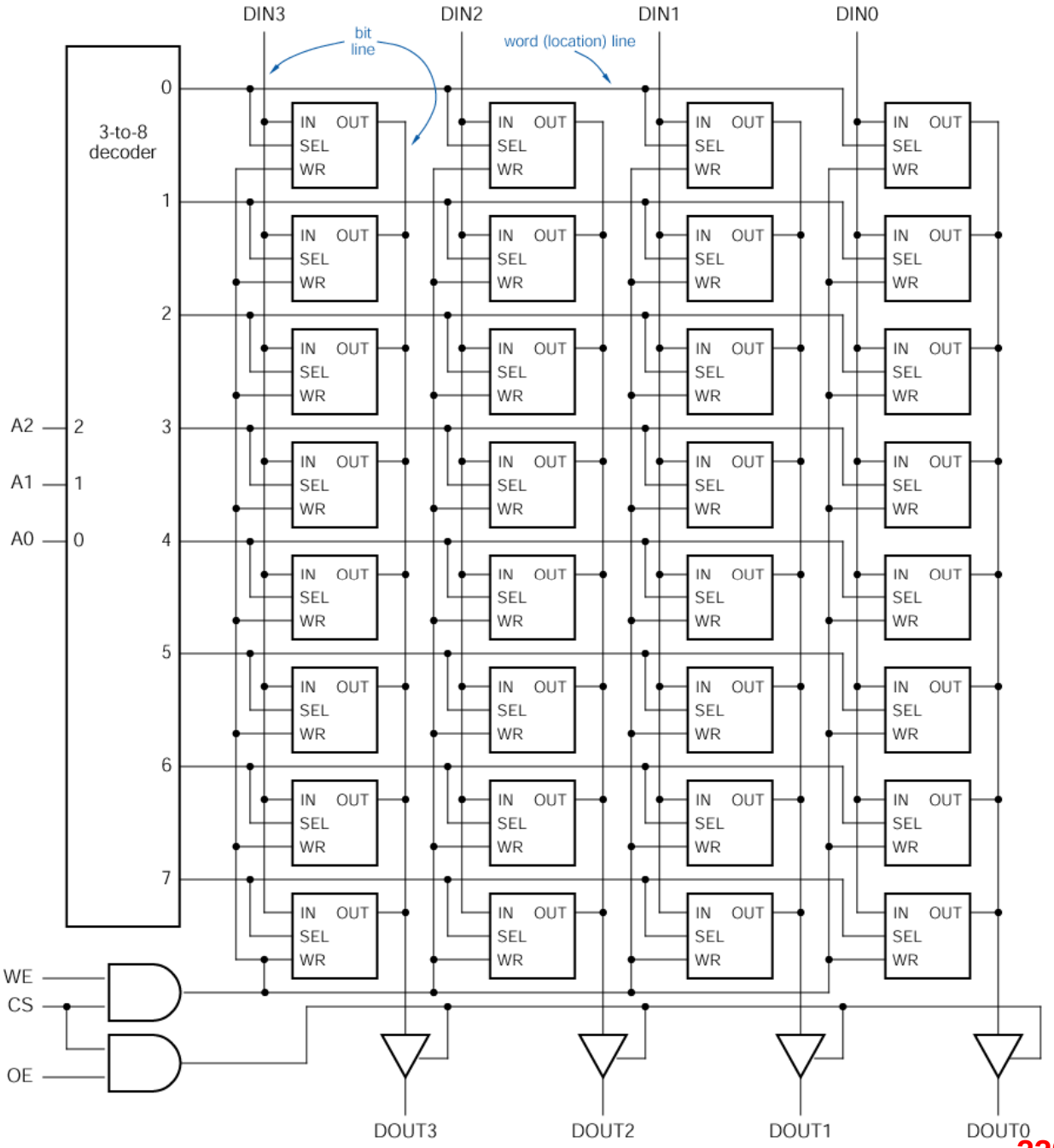
- Each bit of memory (or SRAM cell) in a static RAM behaves as the circuit depicted below



- When the SEL input is asserted, the stored data is placed on the cell's output
- When both SEL and WR are asserted, the latch is open and a new data bit is stored
- SRAM cells are combined in an array with additional control logic to form a complete static RAM



Internal Structure of 8x4 SRAM

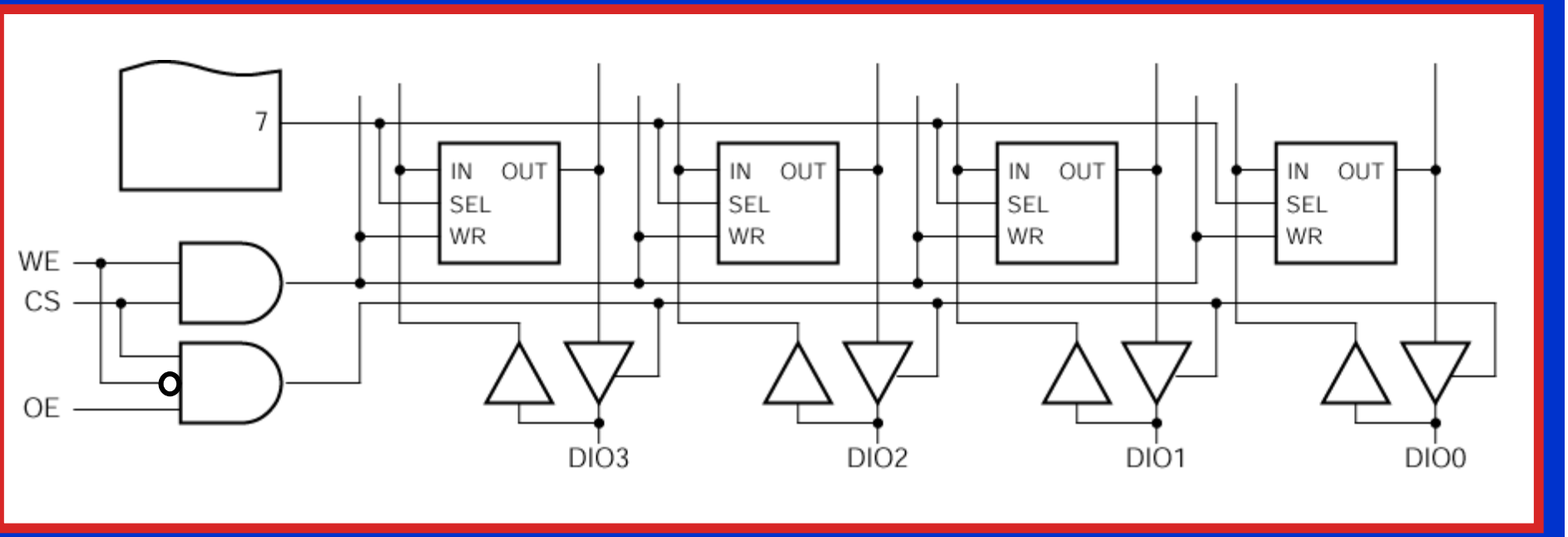


Read/Write Memory (RWM)

- **Some things to note:**
  - **during *read* operations, the output data is a combinational function of the address inputs, so no "harm" is done by changing the address while CS and OE are asserted**
  - **during *write* operations, data is stored in latches – this means that data must meet certain *setup and hold times* with respect to the *negation* of the WE signal**
  - **also during *write* operations, the address lines must be stable for a certain setup time *before* WR is asserted internally and for a hold time *after* WR is negated**

Read/Write Memory (RWM)

- **Most SRAMs utilize a *bi-directional data bus* (i.e., the same data pins are used for reading and writing data)**



Simple Computer Memory

- **The memory for our simple computer will contain 32 locations (5-bit address), each 8 bits wide (i.e., a "32x8" memory)**
- **The memory subsystem will have three control signals:**
  - **MSL: Memory SeLect**
  - **MOE: Memory Output Enable**
  - **MWE: Memory Write Enable**

NOTE: For simplicity (and clarity) all system control signals as well as address and data bus signals will be assumed to be ACTIVE HIGH

Clicker Quiz

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

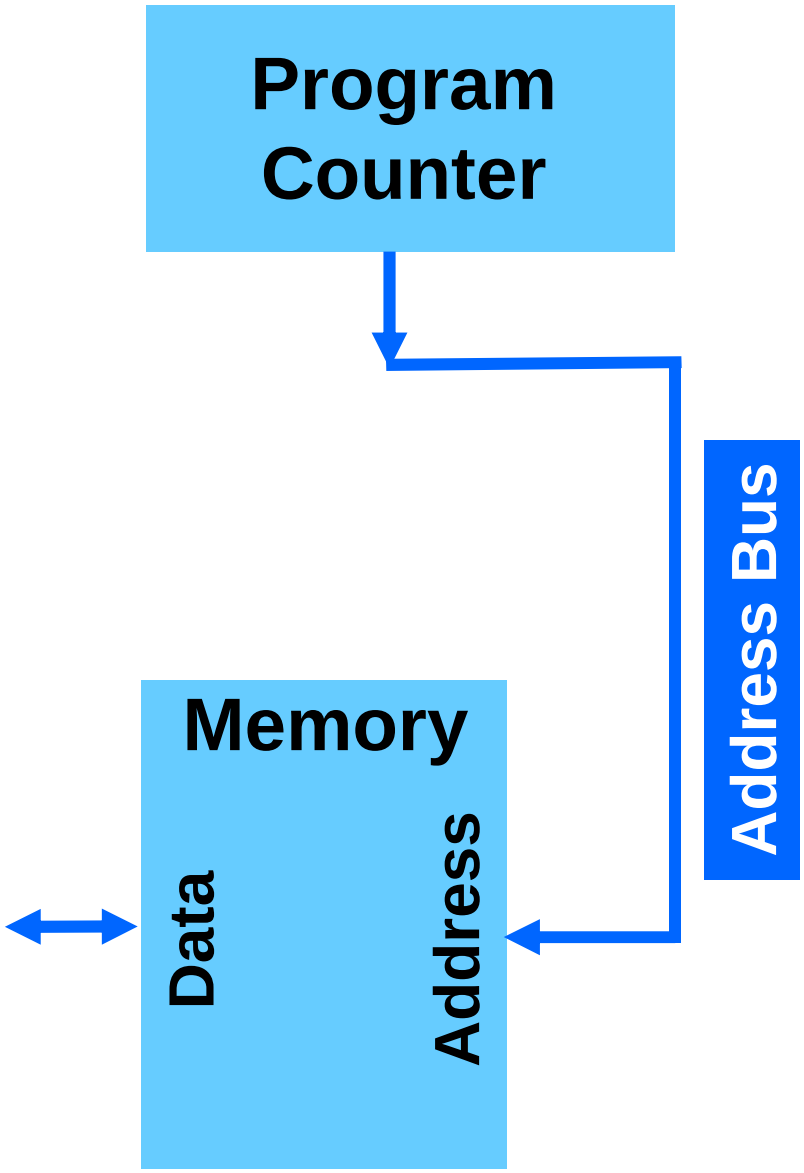
A. all the control signals may be asserted simultaneously
B. only one control signal may be asserted at a given instant
C. each control signal is dependent on the others
D. any combination of control signals may be asserted at a given instant
E. none of the above

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

Simple Computer Block Diagram

- **Functional blocks required (review):**
  - **a place to store the program, operands, and computation results – *memory***
  - ➢ **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Program Counter
Address Bus
Memory
Data
Address

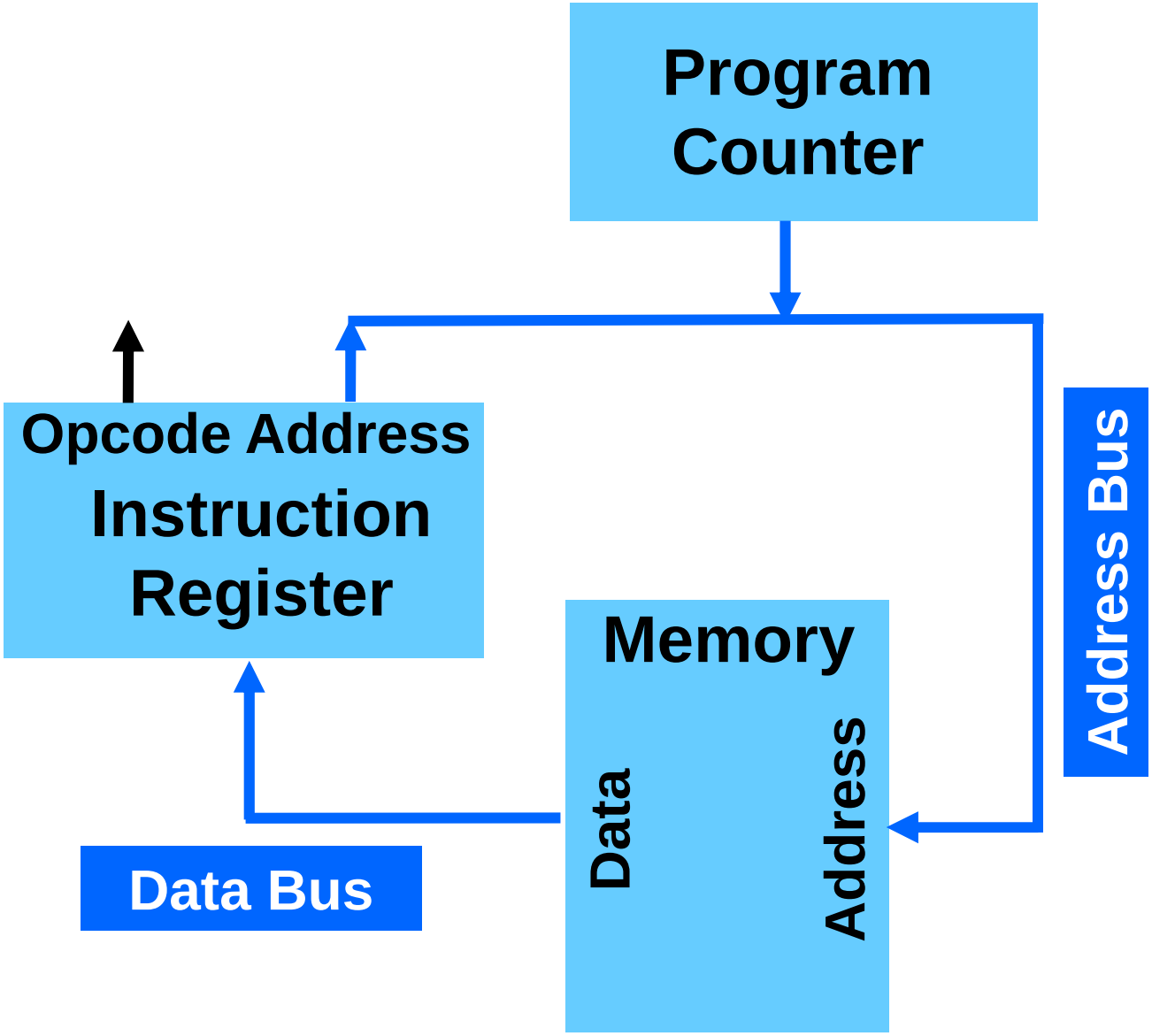
Program Counter

- The ***program counter*** (PC) is basically a binary "up" counter with tri-state outputs
- The functions and corresponding control signals required are as follows:
  - **ARS: A**synchronous **R**e**S**et
  - **PCC: P**rogram **C**ounter **C**ount enable
  - **POA: P**rogram counter **O**utput on **A**ddress 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
```

Simple Computer Block Diagram

- **Functional blocks required (review):**
  - **a place to store the program, operands, and computation results – *memory***
  - **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - ➢ **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Program Counter
Opcode Address
Instruction Register
Memory
Data
Address
Address Bus
Data Bus

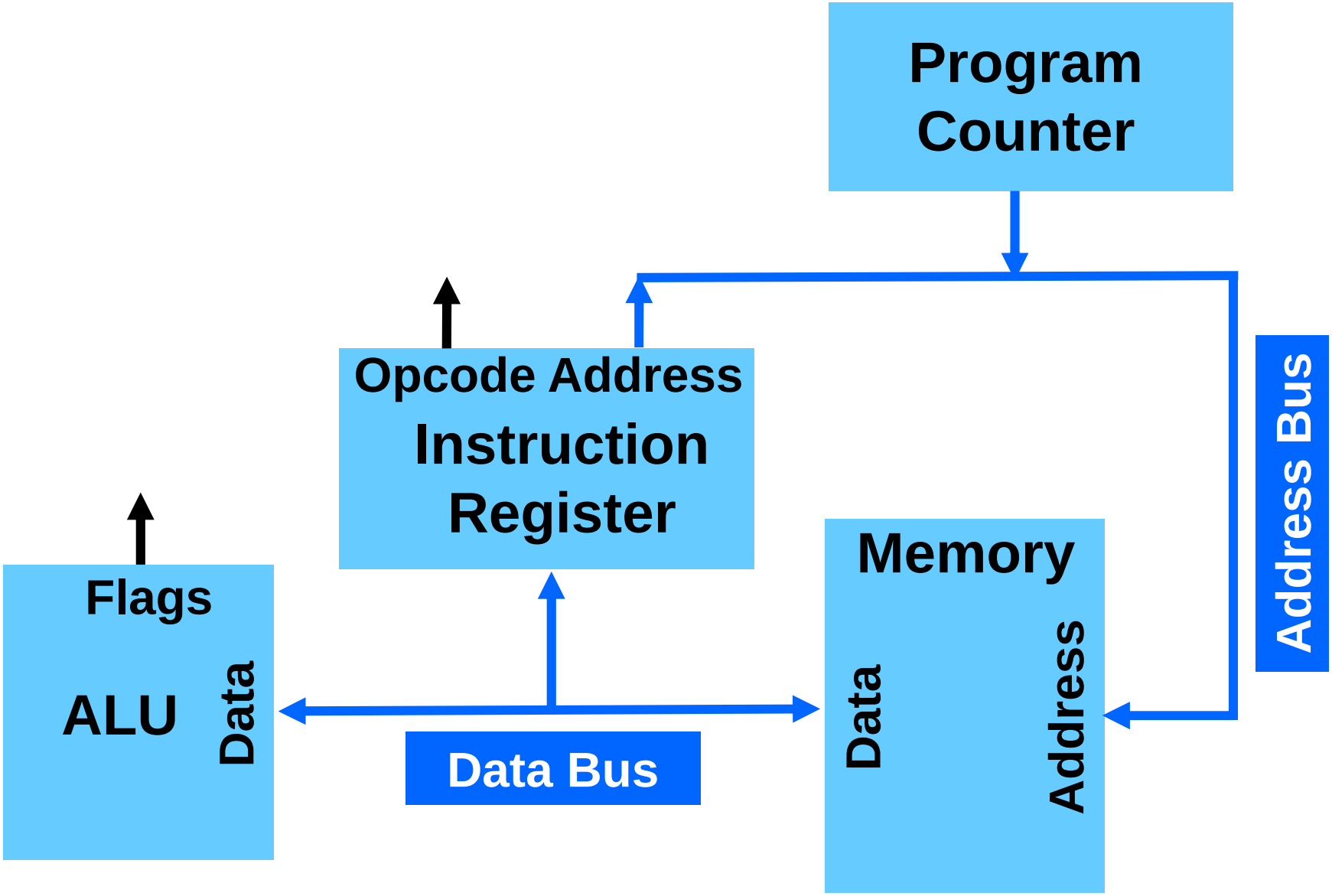
Instruction Register

- The ***instruction register*** (IR) is basically an 8-bit data register, with tri-state outputs on the lower 5 bits
- Note that the upper 3 bits (opcode field) are output directly to the instruction decoder and micro-sequencer
- The functions and corresponding control signals required are as follows:
  - **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
```

Simple Computer Block Diagram

- **Functional blocks required (review):**
  - **a place to store the program, operands, and computation results – *memory***
  - **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Program Counter
Opcode Address
Instruction Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

Arithmetic Logic Unit

- **The *arithmetic logic unit* (ALU) is a multi-function register that performs all the arithmetic and logical (Boolean) operations necessary to implement the instruction set**
- **The functions and corresponding control signals required are as follows:**
  - **ALE: ALU Enable**
  - **ALX: ALU "X" function select**
  - **ALY: ALU "Y" function select**
  - **AOE: A register tri-state Output Enable**

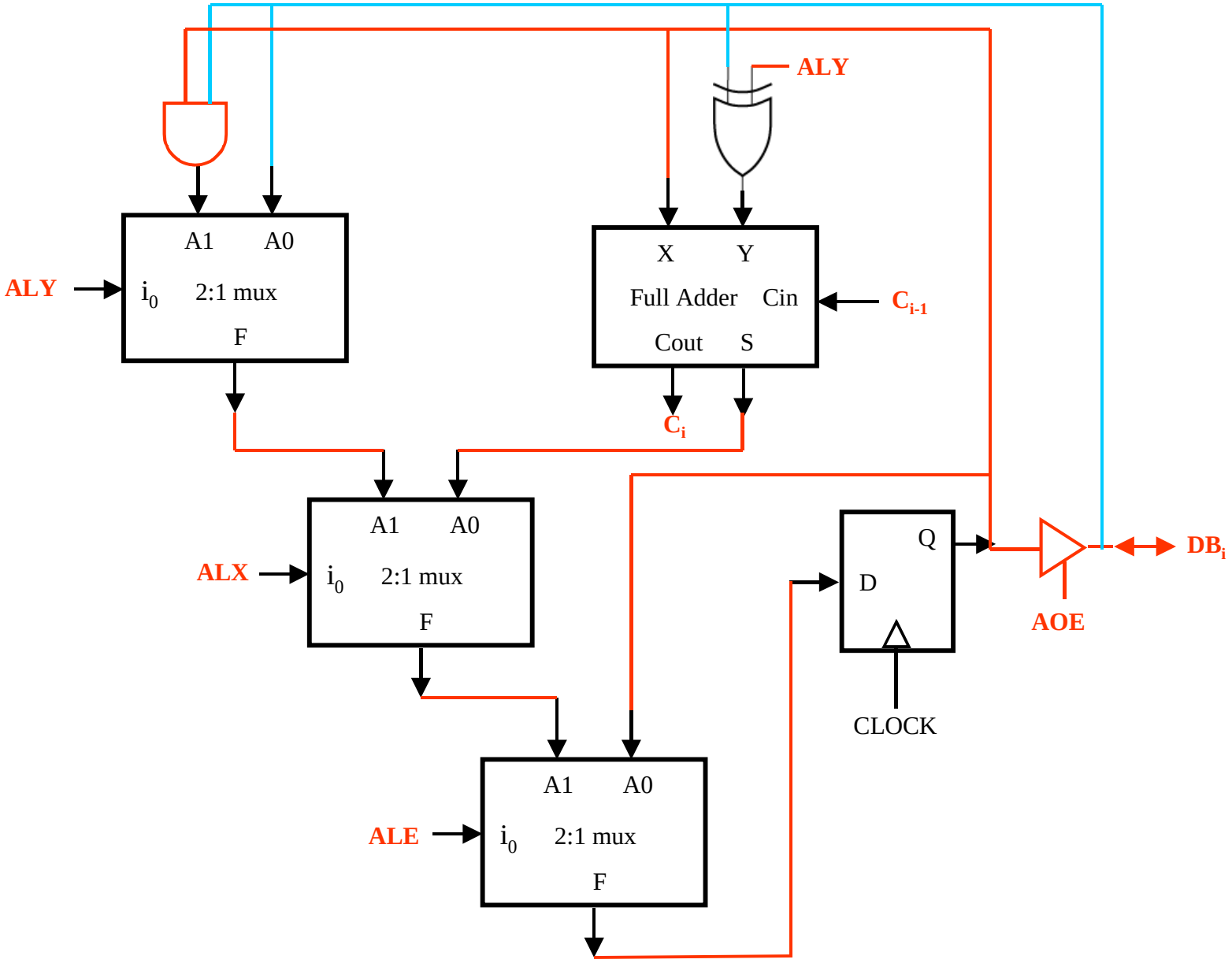
```
MODULE alu

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

```
DECLARATIONS

CLOCK pin;

" ALU control lines (enable & function select)

ALE pin; " overall ALU enable
AOE pin; " data bus tri-state output enable
ALX pin; " function select
ALY pin;

" Carry equations (declare as internal nodes)

CY0..CY7 node istype 'com';

" Combinational ALU outputs (D flip-flop inputs)
" Used for flag generation (declare as internal nodes)

ALU0..ALU7 node istype 'com';

" Bi-directional 8-bit data bus (also, accumulator register bits)

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

" Condition code register bits

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
```

```
" Declaration of intermediate equations

" Least significant bit carry-in (0 for ADD, 1 for SUB => ALY)

CIN = ALY;

" Intermediate equations for adder/subtractor SUM (S0..S7),
" selected when ALX = 0

S0 = DB0.q $ (DB0.pin $ ALY) $ CIN;
S1 = DB1.q $ (DB1.pin $ ALY) $ CY0;
S2 = DB2.q $ (DB2.pin $ ALY) $ CY1;
S3 = DB3.q $ (DB3.pin $ ALY) $ CY2;
S4 = DB4.q $ (DB4.pin $ ALY) $ CY3;
S5 = DB5.q $ (DB5.pin $ ALY) $ CY4;
S6 = DB6.q $ (DB6.pin $ ALY) $ CY5;
S7 = DB7.q $ (DB7.pin $ ALY) $ CY6;

" Intermediate equations for LOAD and AND, selected when ALX = 1

L0 = !ALY&DB0.pin # ALY&DB0.q&DB0.pin;
L1 = !ALY&DB1.pin # ALY&DB1.q&DB1.pin;
L2 = !ALY&DB2.pin # ALY&DB2.q&DB2.pin;
L3 = !ALY&DB3.pin # ALY&DB3.q&DB3.pin;
L4 = !ALY&DB4.pin # ALY&DB4.q&DB4.pin;
L5 = !ALY&DB5.pin # ALY&DB5.q&DB5.pin;
L6 = !ALY&DB6.pin # ALY&DB6.q&DB6.pin;
L7 = !ALY&DB7.pin # ALY&DB7.q&DB7.pin;
```

```
EQUATIONS

" Ripple carry equations (CY7 is COUT)
CY0 = DB0.q&(ALY$DB0.pin) # DB0.q&CIN # (ALY$DB0.pin)&CIN;
CY1 = DB1.q&(ALY$DB1.pin) # DB1.q&CY0 # (ALY$DB1.pin)&CY0;
CY2 = DB2.q&(ALY$DB2.pin) # DB2.q&CY1 # (ALY$DB2.pin)&CY1;
CY3 = DB3.q&(ALY$DB3.pin) # DB3.q&CY2 # (ALY$DB3.pin)&CY2;
CY4 = DB4.q&(ALY$DB4.pin) # DB4.q&CY3 # (ALY$DB4.pin)&CY3;
CY5 = DB5.q&(ALY$DB5.pin) # DB5.q&CY4 # (ALY$DB5.pin)&CY4;
CY6 = DB6.q&(ALY$DB6.pin) # DB6.q&CY5 # (ALY$DB6.pin)&CY5;
CY7 = DB7.q&(ALY$DB7.pin) # DB7.q&CY6 # (ALY$DB7.pin)&CY6;

" Combinational ALU equations
ALU0 = !ALX&S0 # ALX&L0;
ALU1 = !ALX&S1 # ALX&L1;
ALU2 = !ALX&S2 # ALX&L2;
ALU3 = !ALX&S3 # ALX&L3;
ALU4 = !ALX&S4 # ALX&L4;
ALU5 = !ALX&S5 # ALX&L5;
ALU6 = !ALX&S6 # ALX&L6;
ALU7 = !ALX&S7 # ALX&L7;

" 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;
```

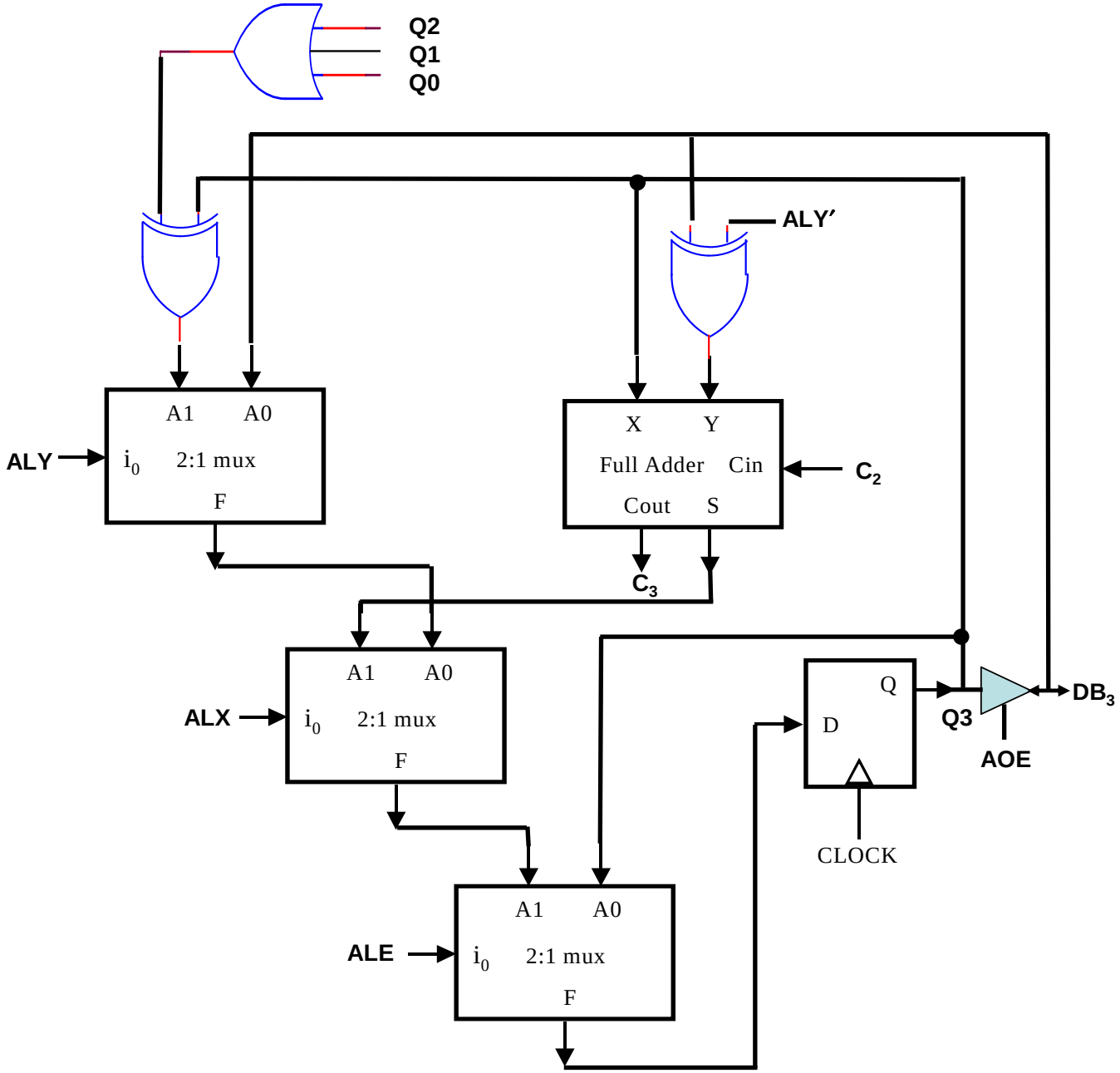
ALU Multiplexer Block Diagram



```
" 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
```

Clicker Quiz

Block Diagram for Bit 3 of a Simple Computer ALU



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

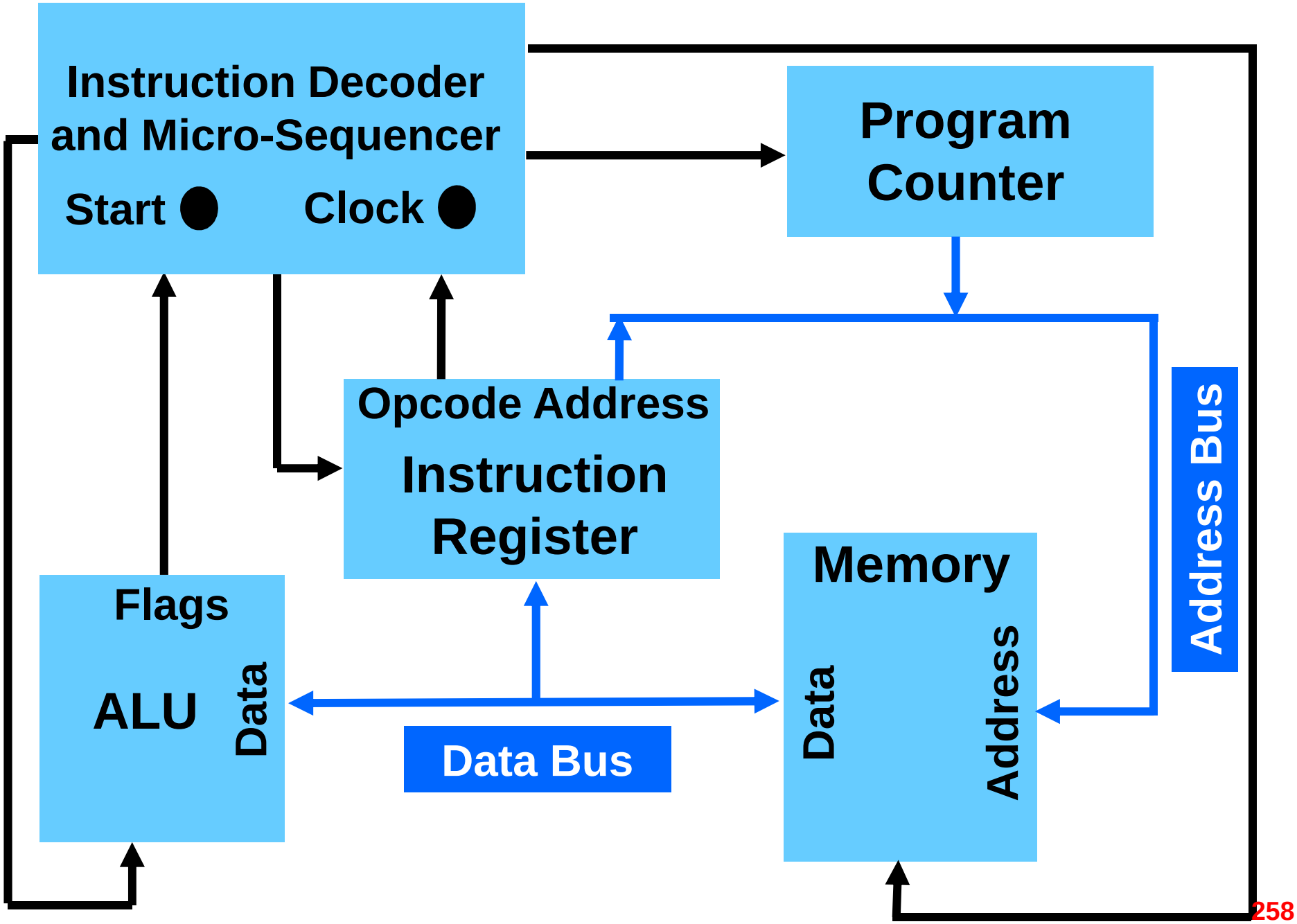
A. ADD
B. SUBTRACT
C. LOAD
D. NEGATE
E. none of the above

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

Simple Computer Block Diagram

- **Functional blocks required (review):**
  - **a place to store the program, operands, and computation results – *memory***
  - **a way to keep track of which instruction is to be executed next – *program counter (PC)***
  - **a place to temporarily "stage" an instruction while it is being executed – *instruction register (IR)***
  - **a way to perform arithmetic and logic operations – *arithmetic logic unit (ALU)***
  - ➢ **a way to coordinate and sequence the functions of the machine – *instruction decoder and micro-sequencer (IDMS)***


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

Instruction Decoder and Microsequencer

- **The *instruction decoder and microsequencer* (IDMS) is a state machine that orchestrates the activity of all the other functional blocks**
- **There are two basic steps involved in "processing" each instruction of a program (called a *micro-sequence*):**
  - ***fetching* the instruction from memory (at the location pointed to by the PC), loading it into the IR, and incrementing the PC**
  - ***executing* the instruction staged in the IR based on the opcode field and the operand address field**

Instruction Decoder and Microsequencer

- **Since there are only *two states* (fetch and execute), a single flip-flop can be used to implement the *state counter* ("SQ")**
- **The control signals that need to be asserted during the *fetch* cycle include:**
  - **POA: turn on PC output buffers**
  - **MSL: select memory**
  - **MOE: turn on memory output buffers**
  - **IRL: enable IR load**
  - **PCC: enable PC count**

NOTE: The *synchronous* 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

Instruction Decoder and Microsequencer

- **The control signals that need to be asserted during an *execute* cycle for the synchronous ALU functions (ADD, SUB, LDA, AND) are:**
  - **IRA: turn on operand address output buffers**
  - **MSL: select memory**
  - **MOE: turn on memory data output buffers**
  - **ALE: enable ALU operation**
  - **ALX, ALY: select ALU function**
- **The control signals that need to be asserted during an *execute* cycle for STA are:**
  - **IRA: turn on operand address output buffers**
  - **MSL: select memory**
  - **MWE: enable memory write**
  - **AOE: turn on A register output buffers**

System Control Signals

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		

Instruction Decoder and Microsequencer

- **In order to *stop* execution (i.e., disable all the functional blocks) when a HLT instruction is executed, an additional flip-flop will be used (called "RUN") , as follows:**
  - **when the START pushbutton is pressed, the RUN flip-flop will be *asynchronously set***
  - **when a HLT instruction is executed, the RUN flip-flop will be *asynchronously cleared***
  - **the RUN signal will be ANDed with the synchronous system enable signals, thus effectively *halting* execution when a HLT instruction is executed**

```
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';

" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

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		

```
" 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 Control Signals

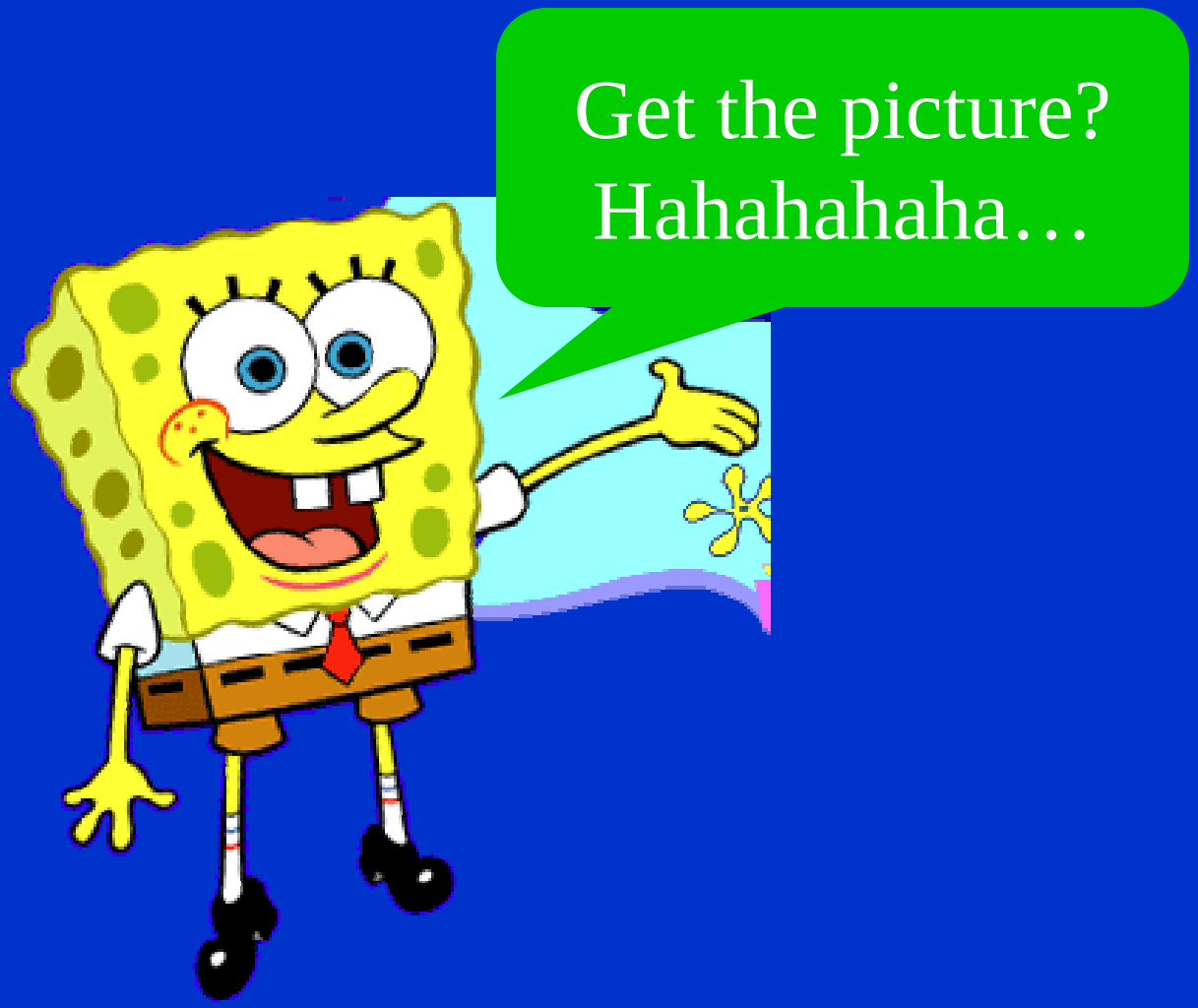
Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY
S0	—	H	H		H	H	H					
S1	LDA	H	H					H		H	H	
S1	STA	H		H				H	H			
S1	ADD	H	H					H		H		
S1	SUB	H	H					H		H		H
S1	AND	H	H					H		H	H	H
S1	HLT	L			L		L			L		

```
" 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 Control Signals

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		

```
" 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
```


Get the picture?
Hahahahaha…

System Data Flow and Timing Analysis

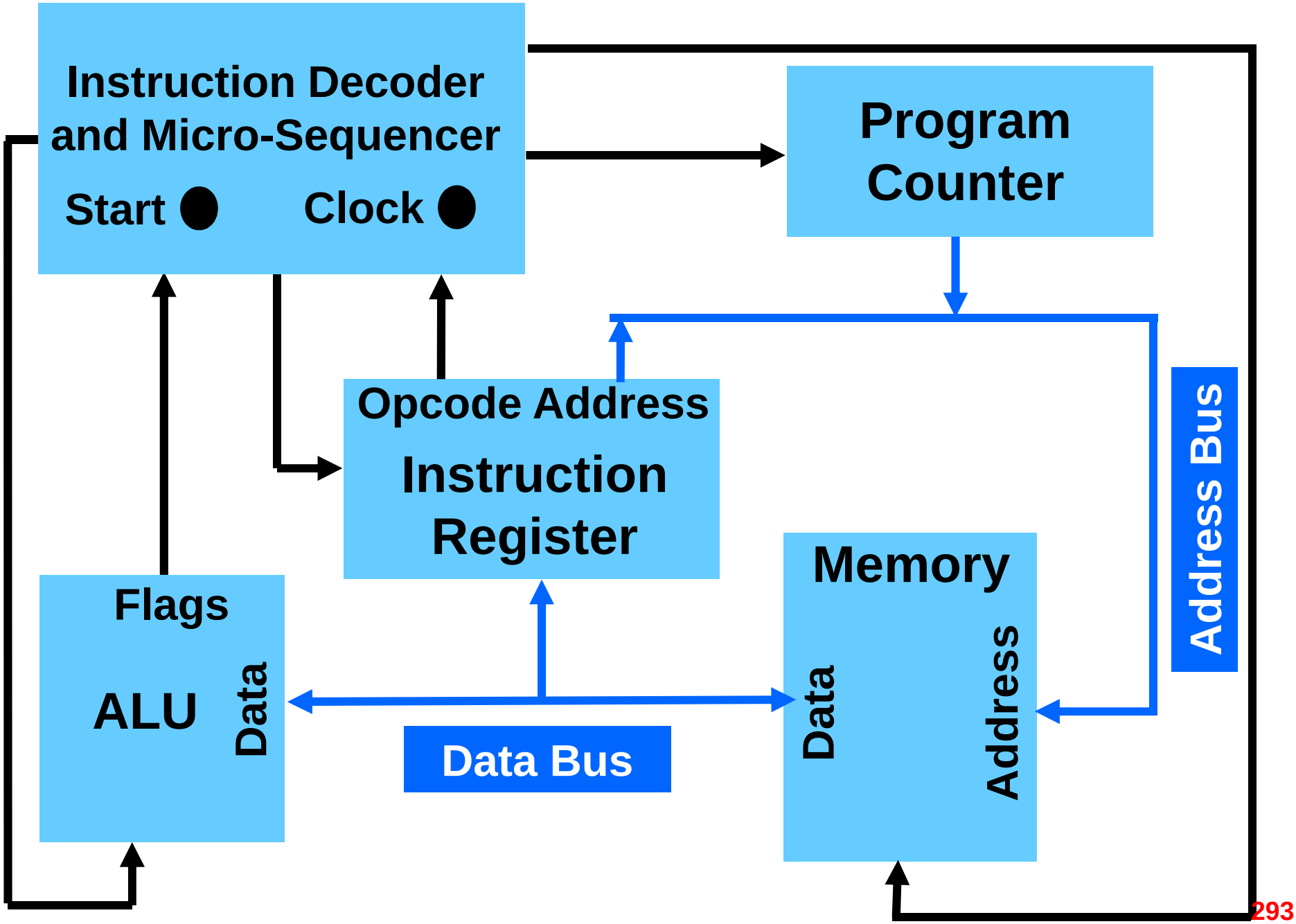
- General procedure
  - Understand operation of individual functional units
    - Memory
    - Program counter
    - Instruction register
    - Arithmetic logic unit
    - Instruction decoder and microsequencer
    - (New functional blocks to be added)

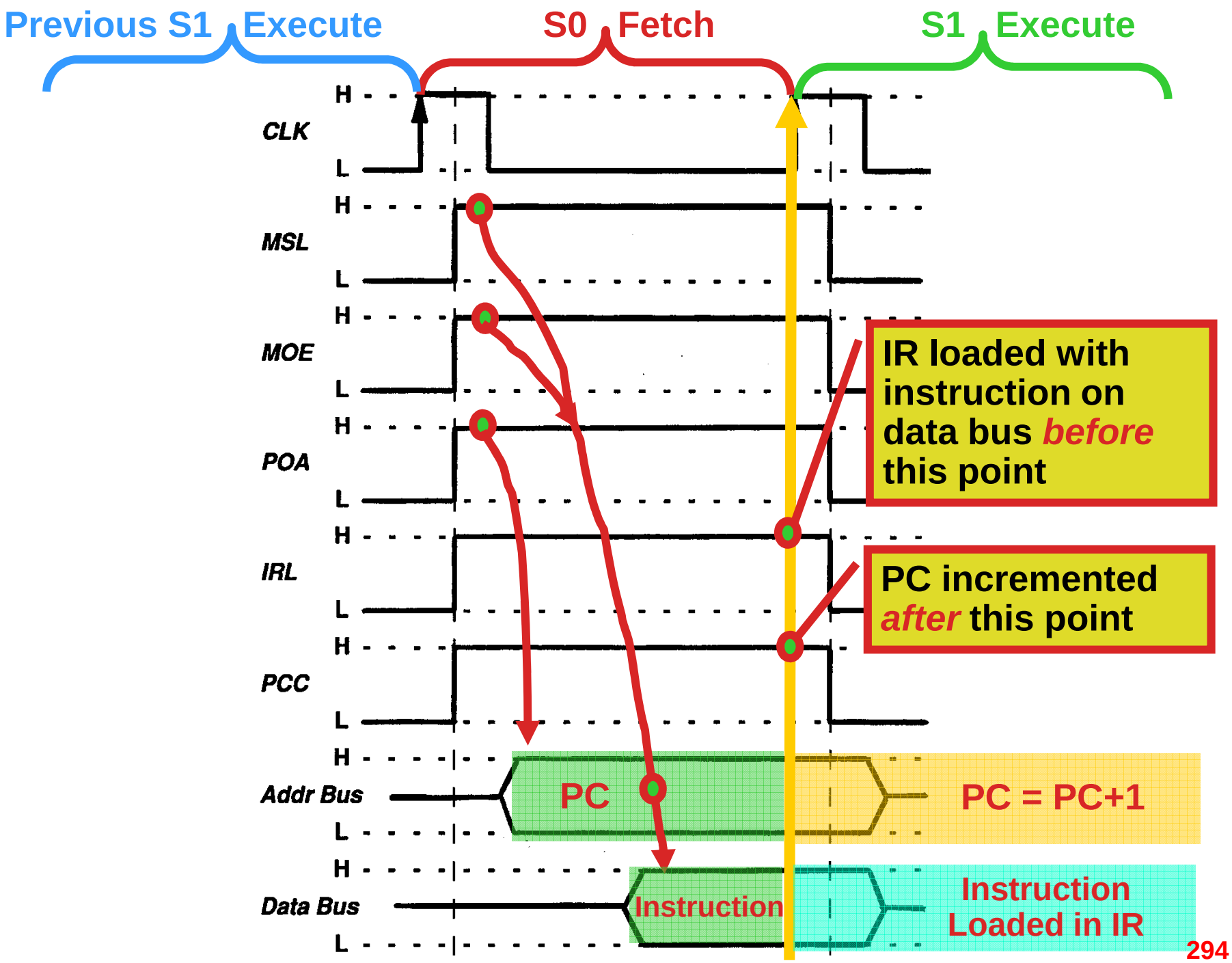
System Data Flow and Timing Analysis

- **General procedure**
  - **Understand function (“data processing”) performed by each instruction**
  - **Identify address and data flow required to execute each instruction**
  - **Identify micro-operations required to execute each instruction**
  - **Identify control signals that need to be asserted to generate the required sequence of micro-operations for each instruction**
  - **Examine timing relationship of control signals**

System Data Flow and Timing Analysis

- **Basic hardware-imposed constraints**
  - **Only *one device* is allowed to drive a bus during any machine cycle (i.e., "bus fighting" must be avoided)**
  - **Data cannot pass through *more than one* (edge-triggered) flip-flop or latch per cycle**


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address


Previous S1
Execute
S0
Fetch
S1
Execute
CLK
MSL
MOE
POA
IRL
PCC
Addr Bus
Data Bus
H
L
PC
PC = PC+1
Instruction
Instruction
Loaded in IR
IR loaded with instruction on data bus *before* this point
PC incremented *after* this point

Clicker Quiz

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 would 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:

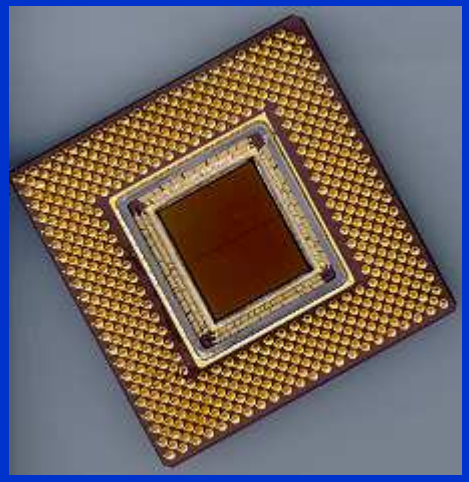
A. the synchronous actions associated with the IRL and PCC control signals occur on different fetch cycle phases
B. the IRL and PCC control signals are not asserted simultaneously by the IDMS
C. the load of the instruction register is based on the data bus value prior to the system CLOCK edge, while 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, $Q(t+\Delta)=D(t)$
E. none of the above



Introduction to Digital System Design

Module 4-J

Simple Computer Basic Extensions

Reading Assignment:

Meyer Supplemental Text, pp. 42-50

Learning Objectives:

- **Modify a reference ALU design to perform different functions (e.g., shift and rotate)**
- **Describe how input/output (IN/OUT) instructions can be added to the base machine architecture**
- **Describe the operation of the I/O block and the function of its control signals: IOR and IOW**
- **Compare and contrast the operation of OUT instructions with and without a transparent latch as an integral part of the I/O block**
- **Compare and contrast "jump" and "branch" transfer-of-control instructions along with the architectural features needed to support them**
- **Distinguish conditional and unconditional branches**
- **Describe the basis for which a conditional branch is "taken" or "not taken"**

Outline

- **Overview**
- **Adding shift instructions**
- **Adding input/output (I/O) instructions**
- **Adding transfer of control instructions**

Overview

- **We will use the two available opcodes (110 and 111) to add new instructions to the basic machine, a pair at a time**

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
1 1 0		
1 1 1		

Overview

- **We will add *rows* to the system control table to add the new instructions and add *columns* to add new control signals**

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				
S1														
S1														

Adding Shift Instructions

- **Definition: A shift instruction *translates* the bits in a register (here, the "A" register) one place to the *left* or to the *right***
- **Definition: An *end off* shift discards the bit that gets shifted out**
- **Definition: A *preserving* shift retains the bit shifted out (typically in the CF condition code bit)**
- **Definition: A *logical* shift is a "zero fill" shift**
- **Definition: An *arithmetic* shift is a "sign preserving" shift (i.e., the sign bit gets replicated as the data is shifted right)**

Shift Instruction Examples

- **Given: (A) = 10011010**
- **After *logical shift left:*** **00110100 CF=1**
- **After *logical shift right:*** **01001101 CF=0**
- **After *arithmetic shift left:*** **00110100 CF=1**
- **After *arithmetic shift right:*** **11001101 CF=0**

Note: The arithmetic left shift is identical to the logical left shift

Adding Shift Instructions

- **Modify the ALU to function as follows:**

ALX	ALY	Function Performed
0	0	Load
0	1	Logical Shift Right
1	0	Arithmetic/Logical Shift Left
1	1	Arithmetic Shift Right

Use the CF condition code bit to preserve the bit that gets shifted out

Modified Instruction Set

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	LSR	Logically shift contents of A right
0 1 1	ASL	Arithmetically/Logically shift contents of A left
1 0 0	ASR	Arithmetically shift contents of A right
1 0 1	HLT	Halt – Stop, discontinue execution
1 1 0		
1 1 1		

Modified System Control Table

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				
S1	STA	H		H				H	H					
S1	LSR									H		H		
S1	ASL									H	H			
S1	ASR									H	H	H		
S1	HLT	L			L		L			L				
S1														
S1														

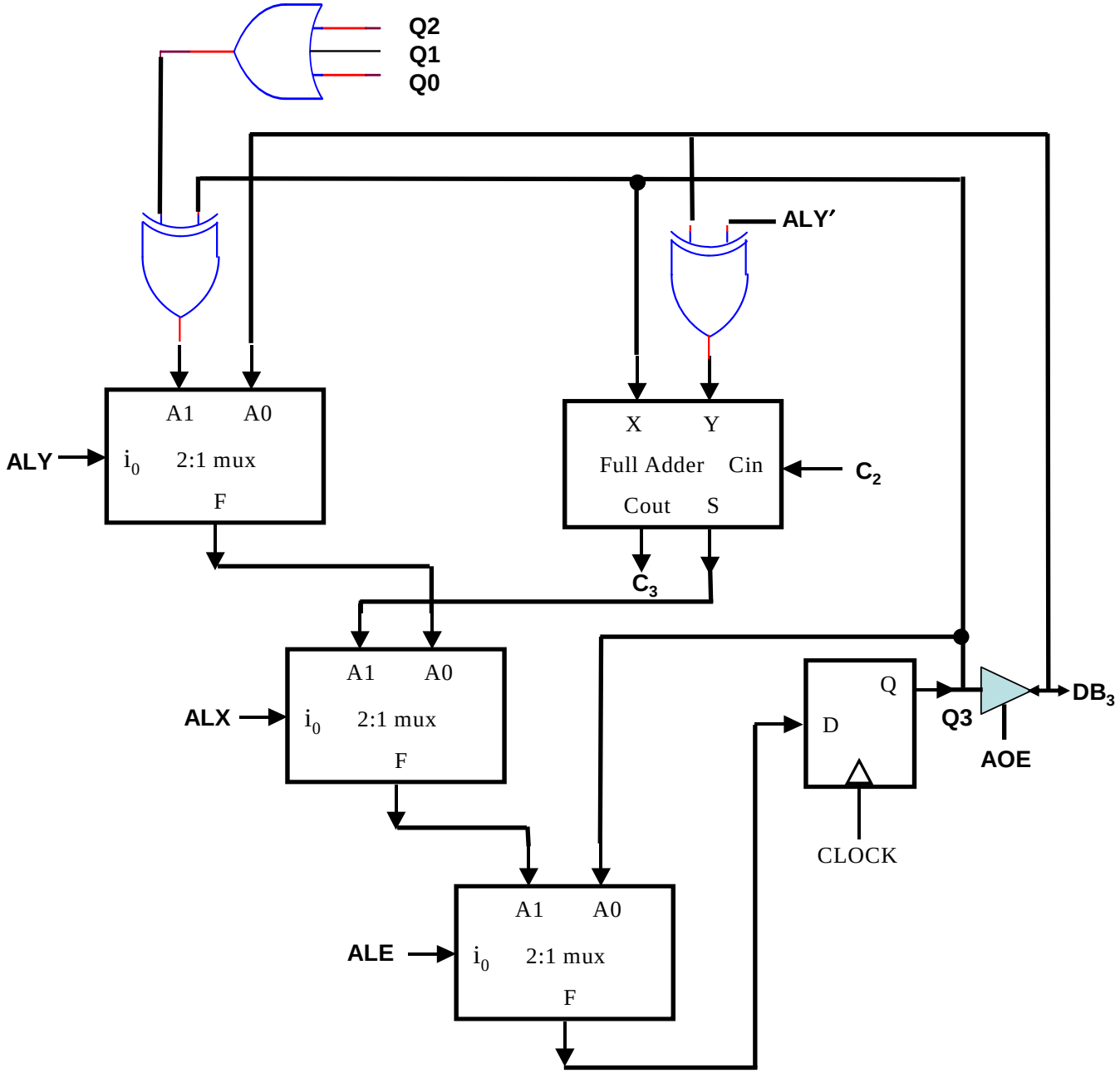
```
MODULE alum

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

```
DECLARATIONS

CLOCK pin;

" ALU control lines (enable & function select)

ALE pin; " overall ALU enable
AOE pin; " data bus tri-state output enable
ALX pin; " function select
ALY pin;

" Combinational ALU outputs (D flip-flop inputs)
" Used for flag generation (declare as internal nodes)

ALU0..ALU7 node istype 'com';

" Bi-directional 8-bit data bus (also, accumulator register bits)

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

" Condition code register bits

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
```

Clicker Quiz

Block Diagram for Bit 3 of a Simple Computer ALU



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

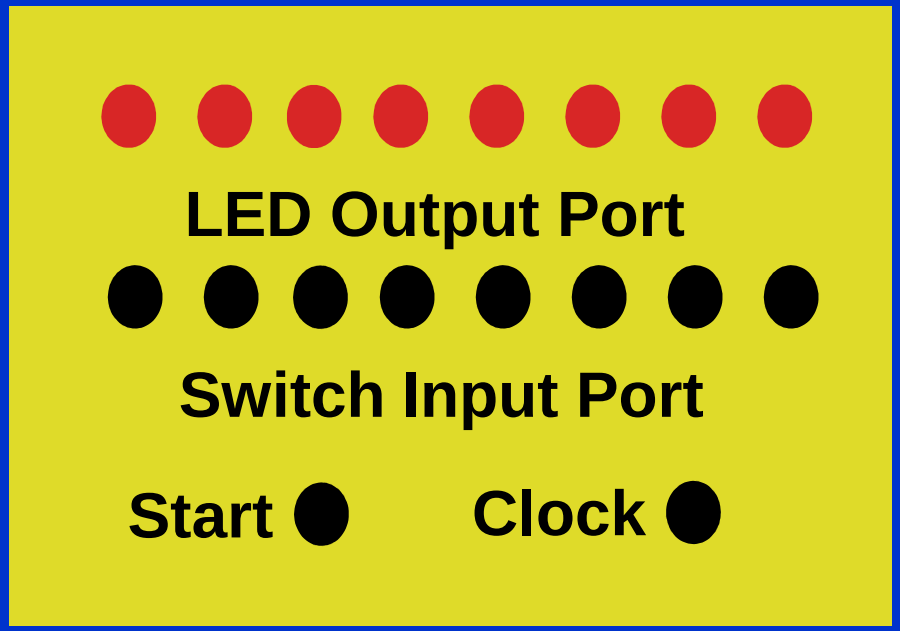
C. rotate left

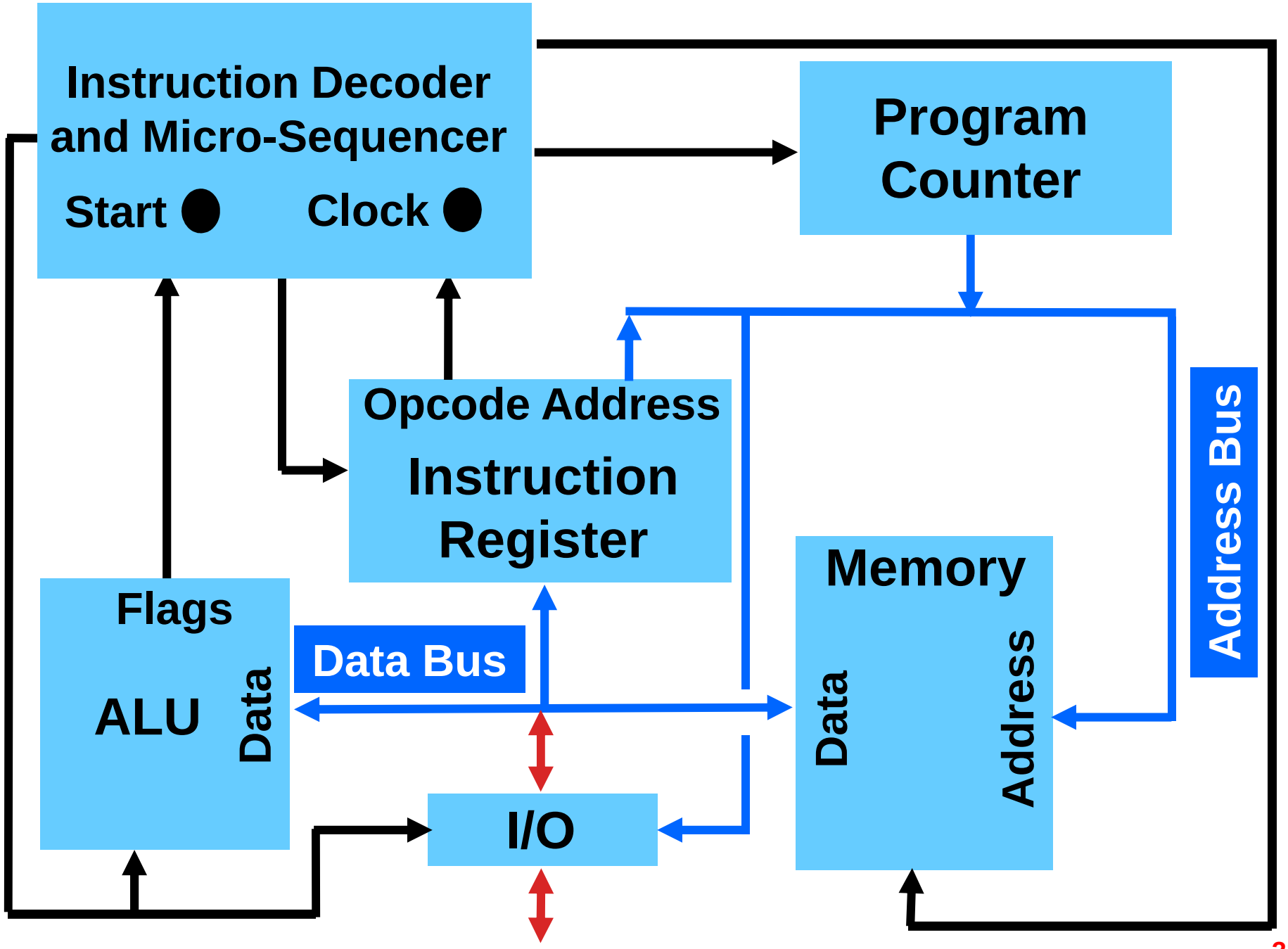
D. rotate right

E. none of the above

Adding I/O Instructions

- **When we first drew the “big picture” of our simple computer, we included a switch “input port” and an LED “output port”**




Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Flags
ALU
Data
Data Bus
I/O
Memory
Data
Address
Address Bus

Adding I/O Instructions

- **Need two new instructions:**
  - **IN *addr* – input data from port *addr* and load into the A register**
  - **OUT *addr* – output data in A register to port *addr***

Here, the *address field* of the instruction is used to specify a *port address* (or, *I/O device ID*)

- **Also need two new control signals:**
  - **IOR – asserted when IN ("I/O read") occurs**
  - **IOW – asserted when OUT ("I/O write") occurs**

```
MODULE io

TITLE    'Input/Output Port 00000'

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;

[OUT0..OUT7] = [DB0..DB7];

[OUT0..OUT7].oe = IOW&PS;

END
```

Issue: Output port bits are valid only as long as IOW&PS is asserted (Hi-Z otherwise)

```
MODULE iol

TITLE    'Input/Output Port 00000 - With Output 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
```

Adding I/O Instructions

- **Modified system control table:**

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

- **Equations that need to be updated: IRA, AOE, ALE, ALX**
- **Equations that need to be added: IOR, IOW**

```
" System control equations (IDMS)

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 # IN # OUT);
AOE = S1&(STA # OUT);
ALE = RUN.q&S1&(LDA # ADD # SUB # AND # IN);
ALX = S1&(LDA # AND # IN);
ALY = S1&(SUB # AND);

IOR = S1&IN;
IOW = S1&OUT;

END
```

Clicker Quiz

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:

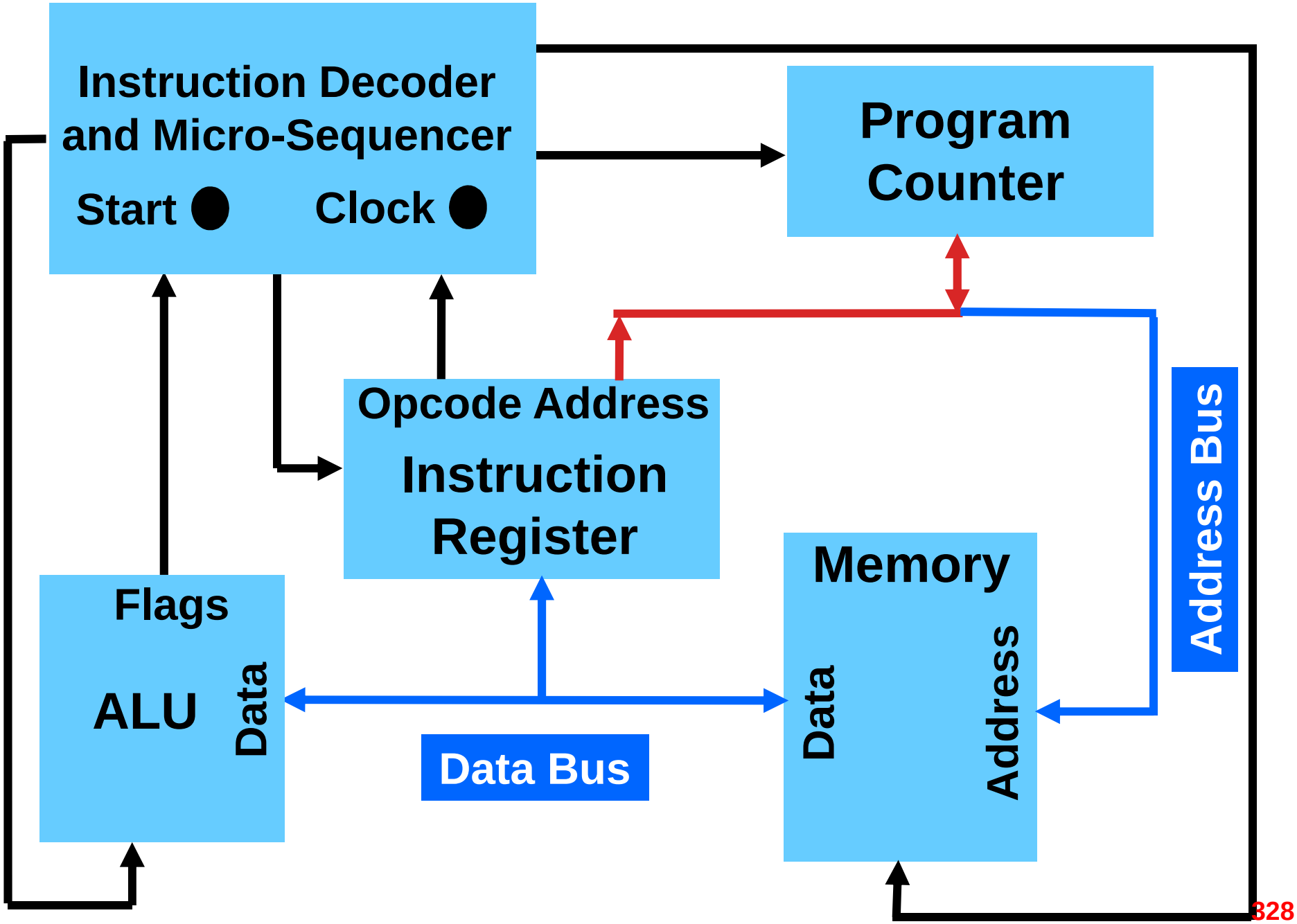
A. only during the execute cycle of the OUT instruction
B. only when the clock signal is high
C. until another OUT instruction writes different data to the port
D. until the next instruction is executed
E. none of the above

Adding Transfer of Control Instructions

- **There are two basic types of *transfer-of-control* instructions:**
  - **absolute: the operand field contains the address in memory at which execution should continue ("jump")**
  - **relative: the operand field contains the signed offset that should be added to the PC to determine the location at which execution should continue ("branch")**
- **Jumps or branches can be *unconditional* ("always happen") or *conditional* ("happen only when a specified condition is met")**

Adding Transfer of Control Instructions

- **For the purpose of illustration, we will add an *unconditional jump* ("JMP") instruction to our simple computer along with a single *conditional* jump ("J*cond*")**
- **The conditional jump illustrated will be a *jump if ZF set* ("JZF") instruction**
- **To execute a jump instruction, the operand field (from the IR) will be loaded into the PC via the *address bus* when PLA is asserted**
- **Note that, for the conditional jump instruction, that if the condition is not met, the execute cycle will be a "*no-operation*" (or, "NOP") cycle**


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
Flags
ALU
Data
Data Bus
Memory
Data
Address
Address Bus

```
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
```

Adding Transfer of Control Instructions

- **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

- **Equation that needs to be updated: IRA**
- **Equation that needs to be added: PLA**

```
" System control equations (IDMS)

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 # JMP # JZF&ZF);
AOE = S1&I1;
ALE = RUN.q&S1&(LDA # ADD # SUB # AND);
ALX = S1&(LDA # AND);
ALY = S1&(SUB # AND);

PLA = S1&(JMP # JZF&ZF);

END
```

Clicker Quiz

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

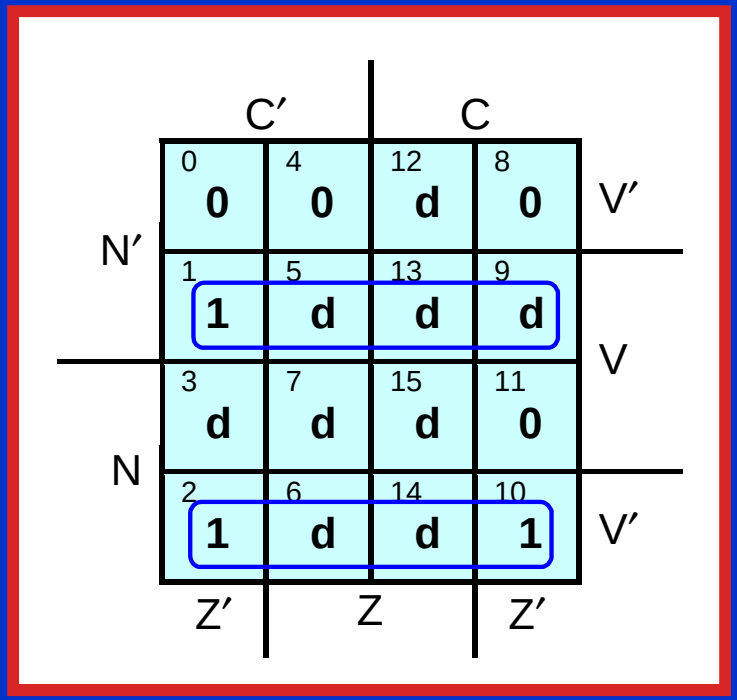
A. add a bi-directional path to the data bus
B. use the ALU to compute the address of the next instruction
C. make it an up/down counter
D. add a two's complement N-bit adder circuit (where N is the address bus width)
E. none of the above

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

Thought Questions

- **What would be required to add a *jump if less than* (“JLT”) or a *jump if greater than or equal to* (“JGE”) instruction?**



These jump conditions can be determined based on the ALU flags (N, C, Z, V)

JLT condition
= N´•V + N•V´
= N ⊕ V

JGE condition
= (N ⊕ V)´

Derivation of Signed Comparisons

R_1	R_0	(R)	M_1	M_0	(M)	?	C	Z	N	V
0	0	0	0	0	0	(R) = (M)	0	1	0	0
0	0	0	0	1	+1	(R) < (M)	1	0	1	0
0	0	0	1	0	-2	(R) > (M)	1	0	1	1
0	0	0	1	1	-1	(R) > (M)	1	0	0	0
0	1	+1	0	0	0	(R) > (M)	0	0	0	0
0	1	+1	0	1	+1	(R) = (M)	0	1	0	0
0	1	+1	1	0	-2	(R) > (M)	1	0	1	1
0	1	+1	1	1	-1	(R) > (M)	1	0	1	1
1	0	-2	0	0	0	(R) < (M)	0	0	1	0
1	0	-2	0	1	+1	(R) < (M)	0	0	0	1
1	0	-2	1	0	-2	(R) = (M)	0	1	0	0
1	0	-2	1	1	-1	(R) < (M)	1	0	1	0
1	1	-1	0	0	0	(R) < (M)	0	0	1	0
1	1	-1	0	1	+1	(R) < (M)	0	0	1	0
1	1	-1	1	0	-2	(R) > (M)	0	0	0	0
1	1	-1	1	1	-1	(R) = (M)	0	1	0	0

Z = 1
(R) = (M)

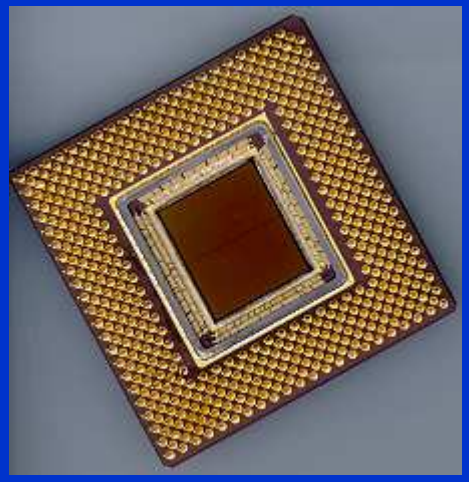
Z + (N ⊕ V) = 0
(R) > (M)

N ⊕ V = 1
(R) < (M)

Thought Questions

- **How could a *compare* ("CMP") instruction be implemented? How is this different than a *subtract* ("SUB")?**

A "CMP" works the same as "SUB" *except* that the result of (A) – (*addr*) is *not stored* in the A register (i.e., *only* the *flags* are affected)



Introduction to Digital System Design

Module 4-K

Simple Computer Advanced Extensions

Reading Assignment:
Meyer Supplemental Text, pp. 50-64

Learning Objectives:

- **describe the changes needed to the instruction decoder/microsequencer in order to dynamically change the number of instruction execute cycles based on the opcode**
- **compare and contrast the machine's asynchronous reset ("START") with the synchronous state counter reset ("RST")**
- **describe the operation of a stack mechanism (LIFO queue)**
- **describe the operation of the stack pointer (SP) register and the function of its control signals: ARS, SPI, SPD, SPA**
- **compare and contrast the two possible stack conventions: SP pointing to the top stack item vs. SP pointing to the top stack item**
- **describe how stack manipulation instructions (PSH/POP) can be added to the base machine architecture**
- **discuss the consequences of having an unbalanced set of PSH and POP instructions in a given program**

Reading Assignment:
Meyer Supplemental Text, pp. 50-64

Learning Objectives, continued:

- **discuss the reasons for using a stack as a subroutine linkage mechanism: arbitrary nesting of subroutine calls, passing parameters to subroutines, recursion, and reentrancy**
- **describe how subroutine linkage instructions (JSR/RTS) can be added to the base machine architecture**
- **analyze the effect of changing the stack convention utilized (SP points to top stack item vs. next available location) on instruction cycle counts**

Outline

- Modifications to state counter and instruction decoder to accommodate instructions with multiple execute cycles
- Introduction of a stack mechanism
- Use of a stack to implement “push” (PSH) and “pop” (POP) instructions
- Identification of micro-operations that can be overlapped
- Modifications to the system architecture necessary to implement subroutine linkage instructions
- Implementation of subroutine “call” (JSR) and “return” (RTS) instructions

State Counter Modifications

- **All of the “simple computer” instructions discussed thus far have required only two states: a fetch cycle followed by a *single* execute cycle**
- **Many “real world” instructions require more than a single execute state to achieve their desired functionality**
- **We wish to extend the state counter (and the instruction decoder) to accommodate instructions that require *multiple* execute cycles (here, up to *three*)**

State Counter Modifications

- **In the process of adding this capability, we want to make sure our original "shorter" instructions do not incur a "penalty" (i.e., execute *slower*)**
- **To accomplish this, we need to design the state counter so that the number of execute cycles can be *dynamically changed* based on the opcode of the instruction being executed**
- **To provide up to three execute cycles, we will replace the state counter flip-flop with a *two-bit binary counter* that has a *synchronous reset* input (in addition to an asynchronous clear)**

State Counter Modifications

- **The "state names" will now be:**
  - **S0 (fetch)**
  - **S1 (first execute)**
  - **S2 (second execute)**
  - **S3 (third execute)**
- **We will also add a new system control signal "RST" (connected to the synchronous reset of the binary counter) that will be asserted on the *last execute state* of each instruction, thereby *synchronously resetting* the state counter to zero (so that the next cycle will be a "fetch")**

```
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)
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);
RST = S1&(LDA # STA # ADD # SUB # AND);

END
```

Clicker Quiz

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

Stack Mechanism

- **Definition: A *stack* is a *last-in, first-out* (LIFO) data structure**
- **Primary uses of stacks in computers:**
  - **subroutine linkage**
    - ***saving return address***
    - ***parameter passing***
  - **saving machine *context* (or *state*) – especially when processing *interrupts* or *exceptions***
  - **expression evaluation**

Stack Mechanism

- **Conventions:**
  - **the stack area is generally placed at the "top" of memory (i.e., starting at the *highest address* in memory)**
  - **a stack pointer (SP) register is used to indicate the address of the *top stack item***
  - **stack *growth* is toward *decreasing* addresses (note that this is in contrast to *program growth*, which is toward *increasing addresses*)**

Note: An *alternate* convention for the stack could also be used, namely, to have the SP register point to the *next available location*

Illustration of Stack Growth

- **Initial condition (stack empty):**

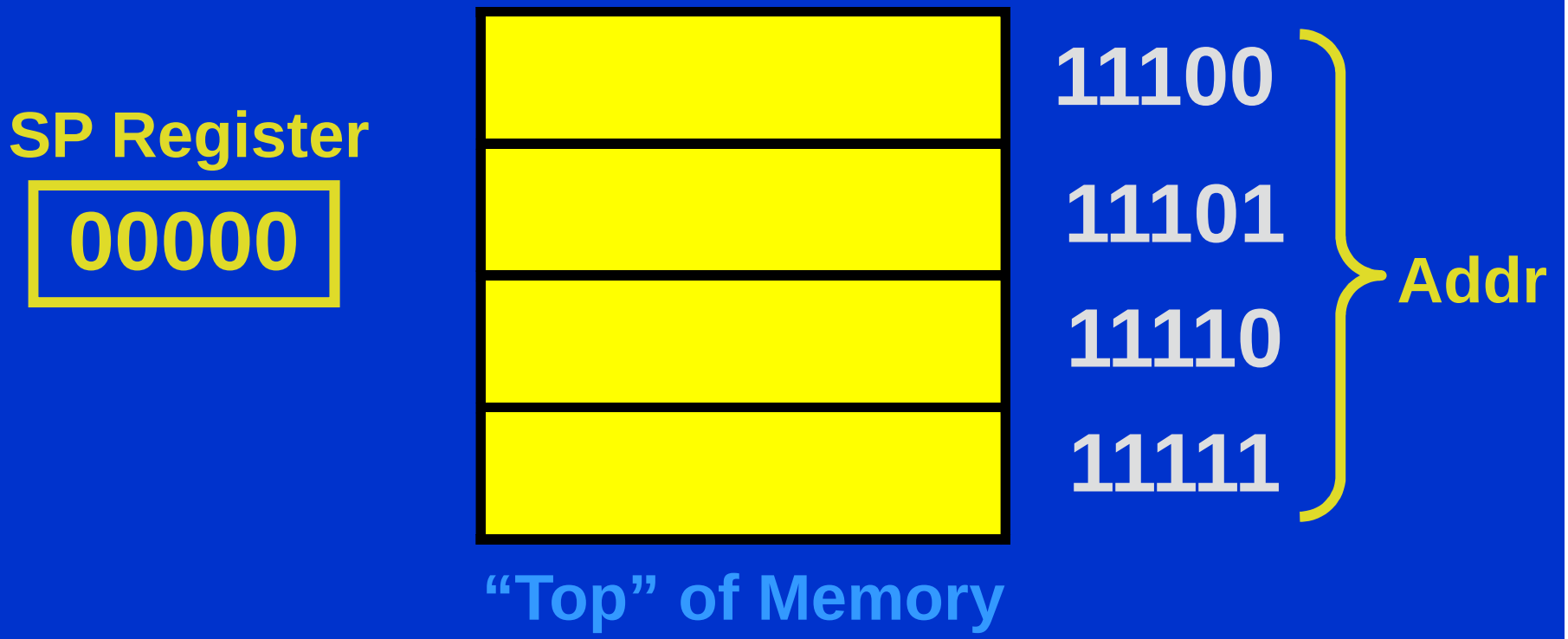


Illustration of Stack Growth

- **After first item pushed onto stack:**

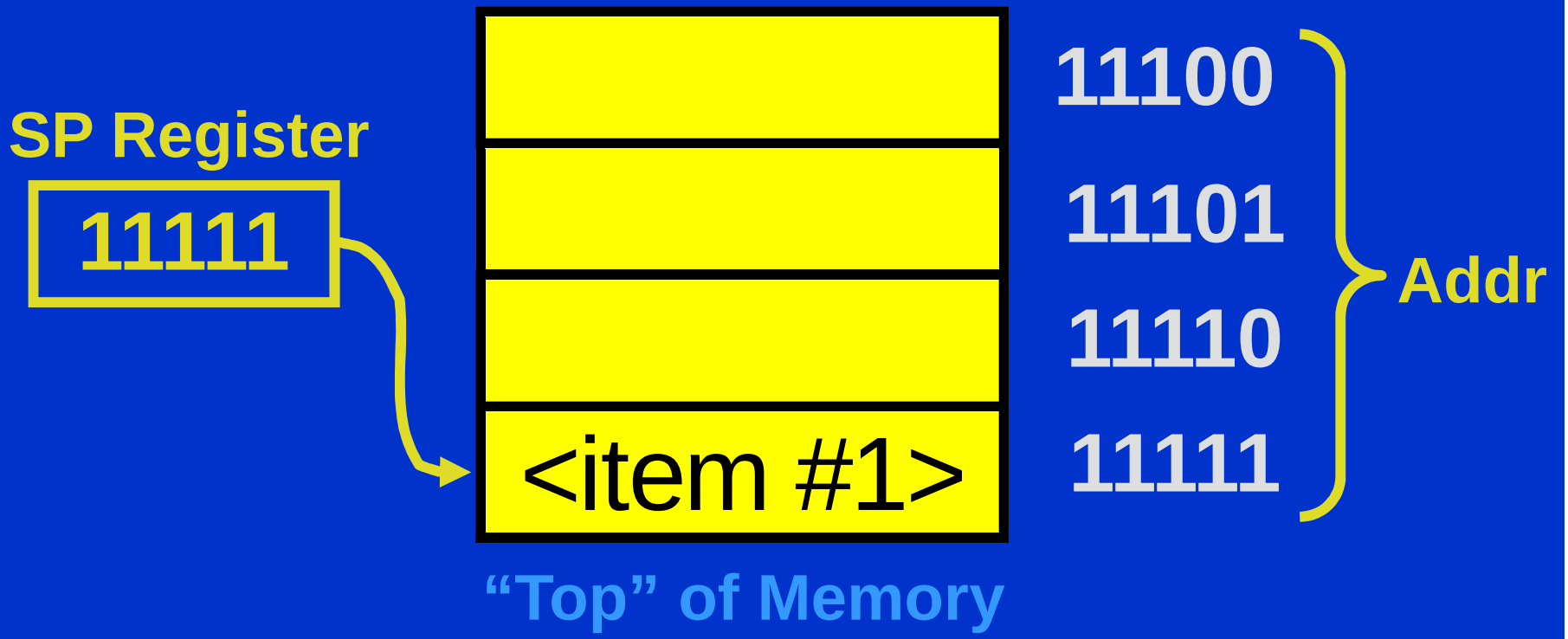


Illustration of Stack Growth

- **After second item pushed onto stack:**

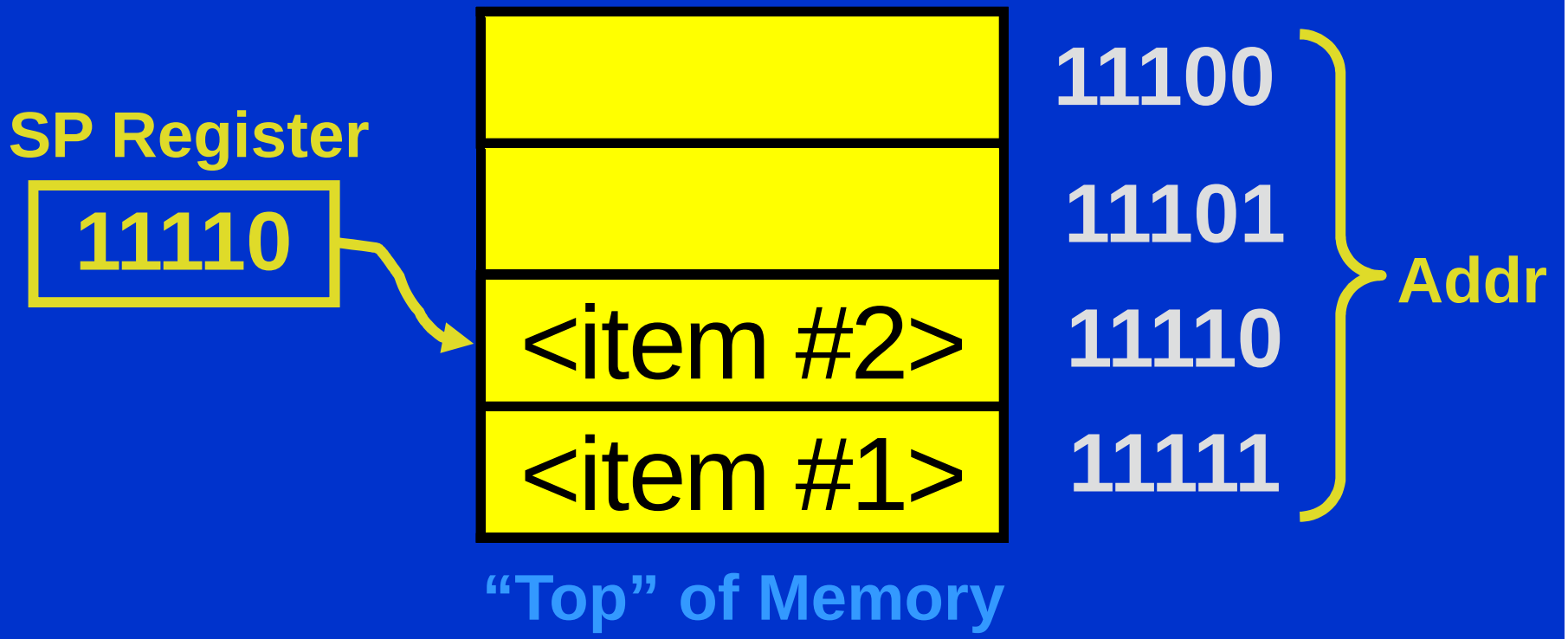


Illustration of Stack Growth

- **After third item pushed onto stack:**

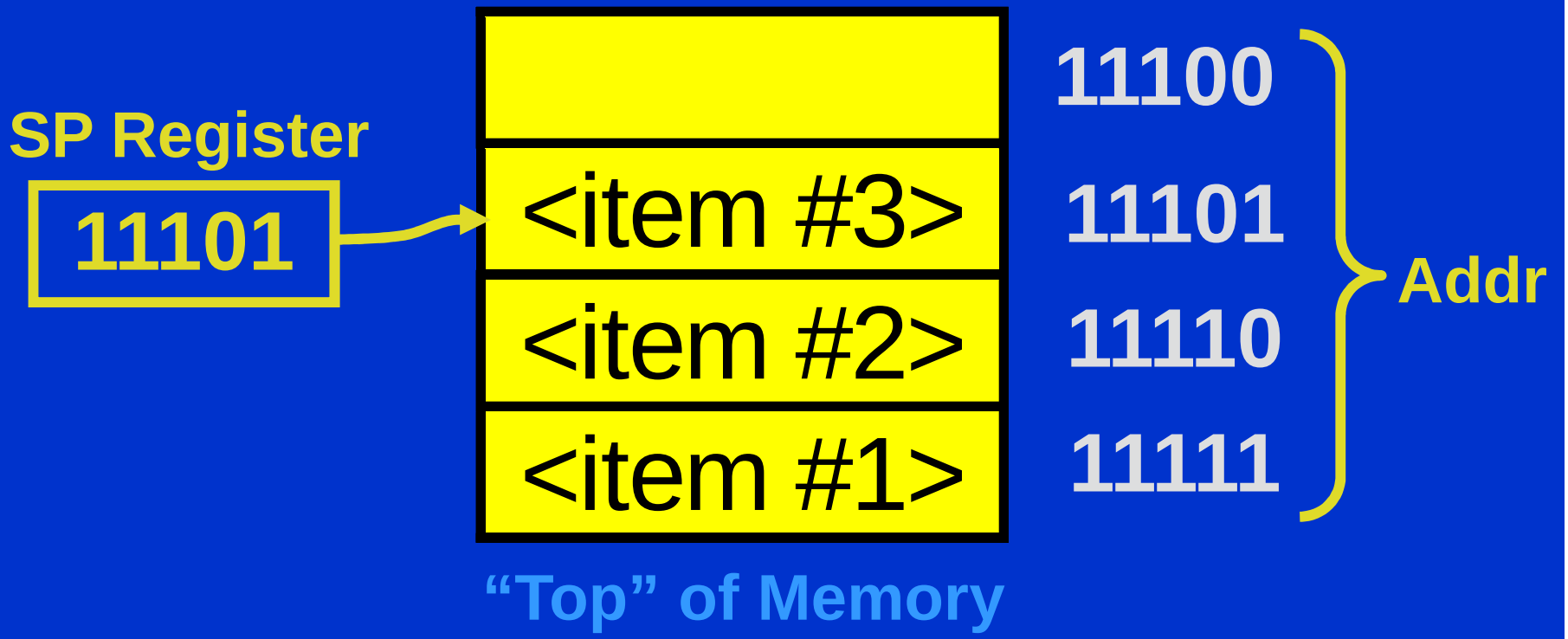


Illustration of Stack Growth

- **After fourth item pushed onto stack:**

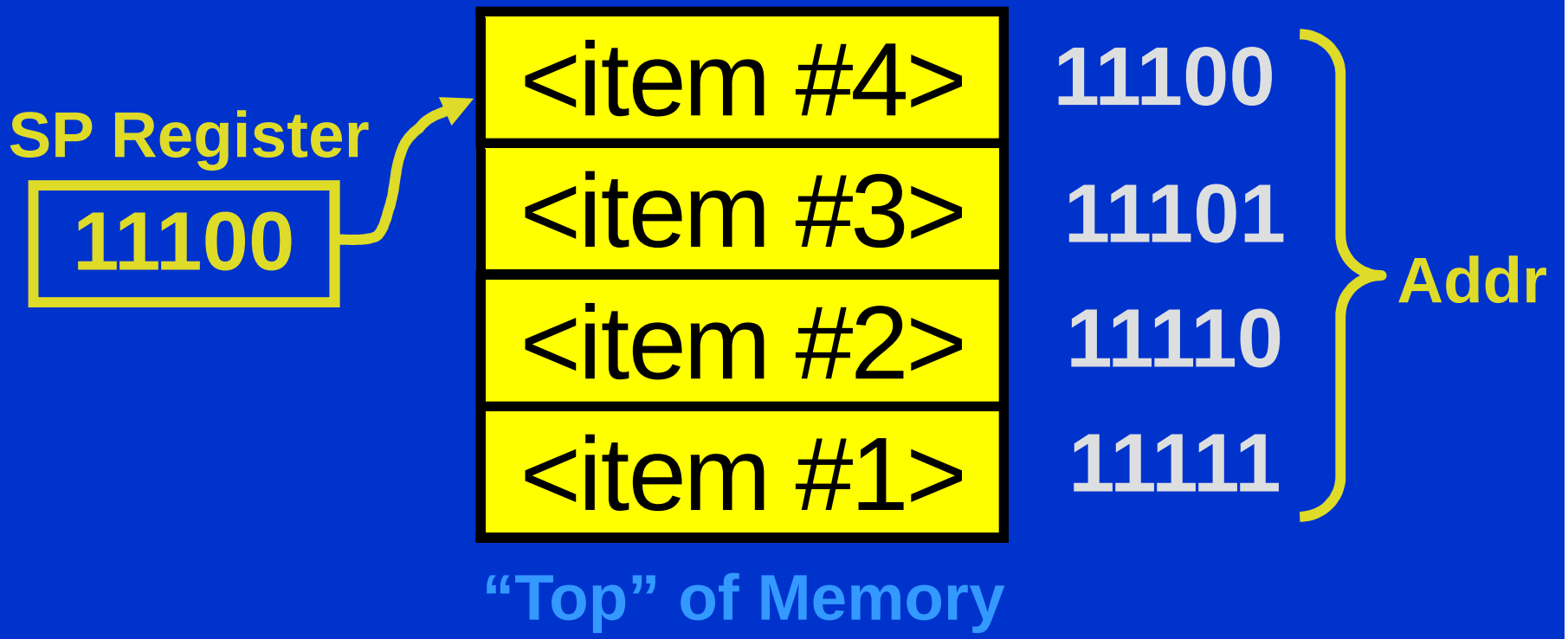


Illustration of Stack Growth

- **After top item removed:**

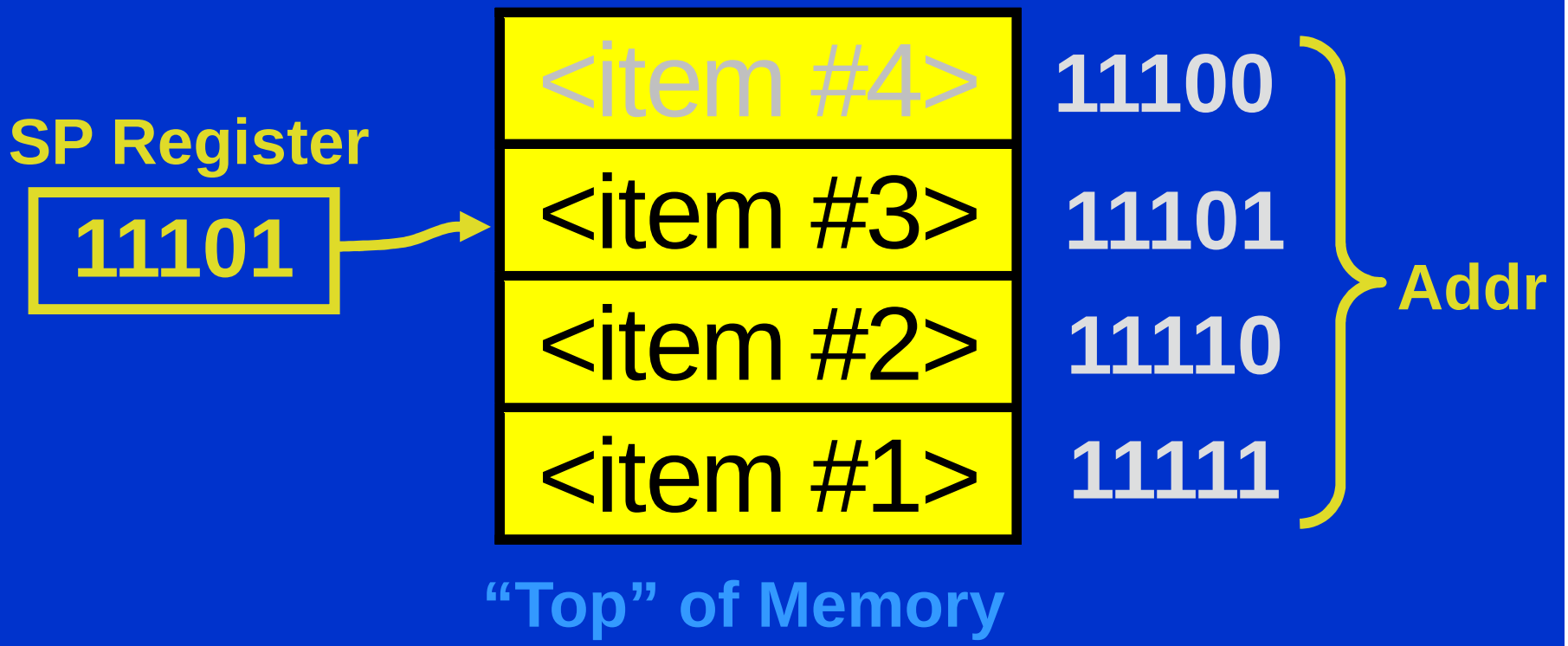


Illustration of Stack Growth

- **After top item removed:**

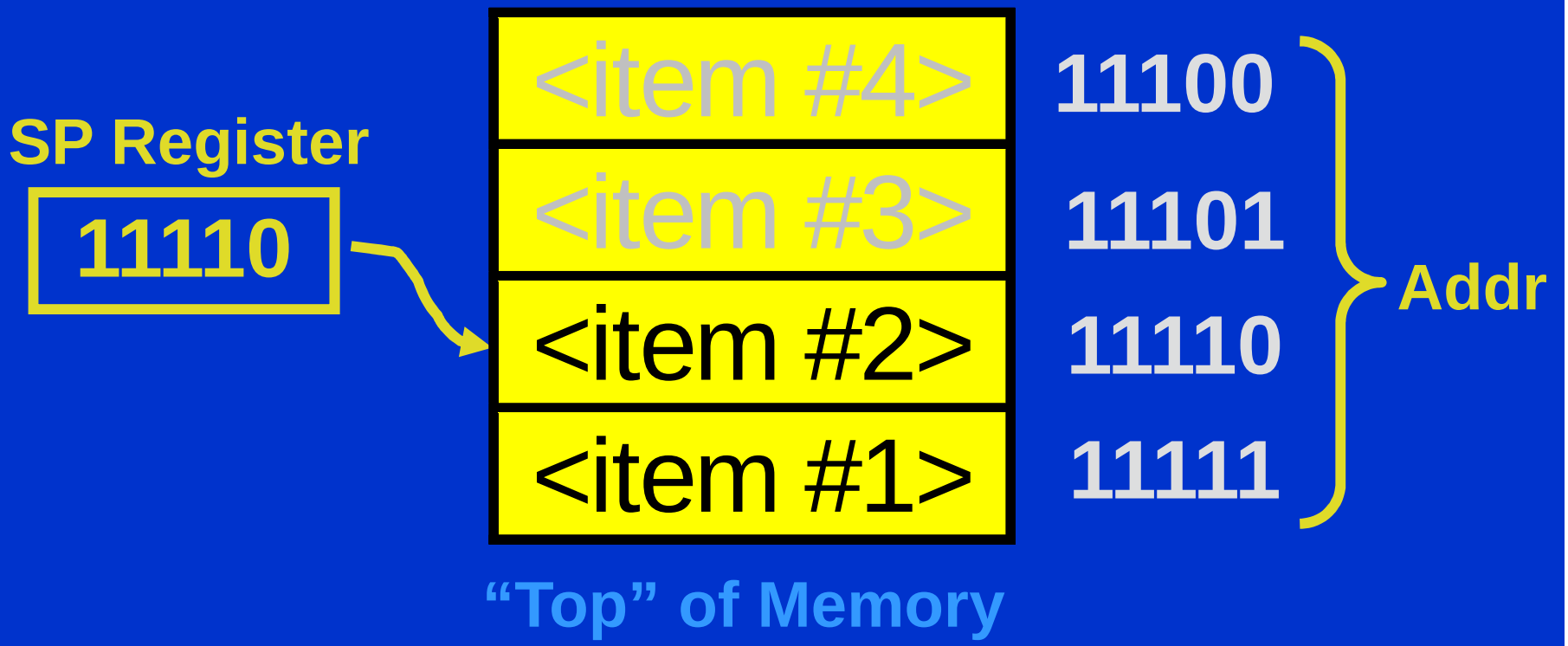


Illustration of Stack Growth

- **After top item removed:**

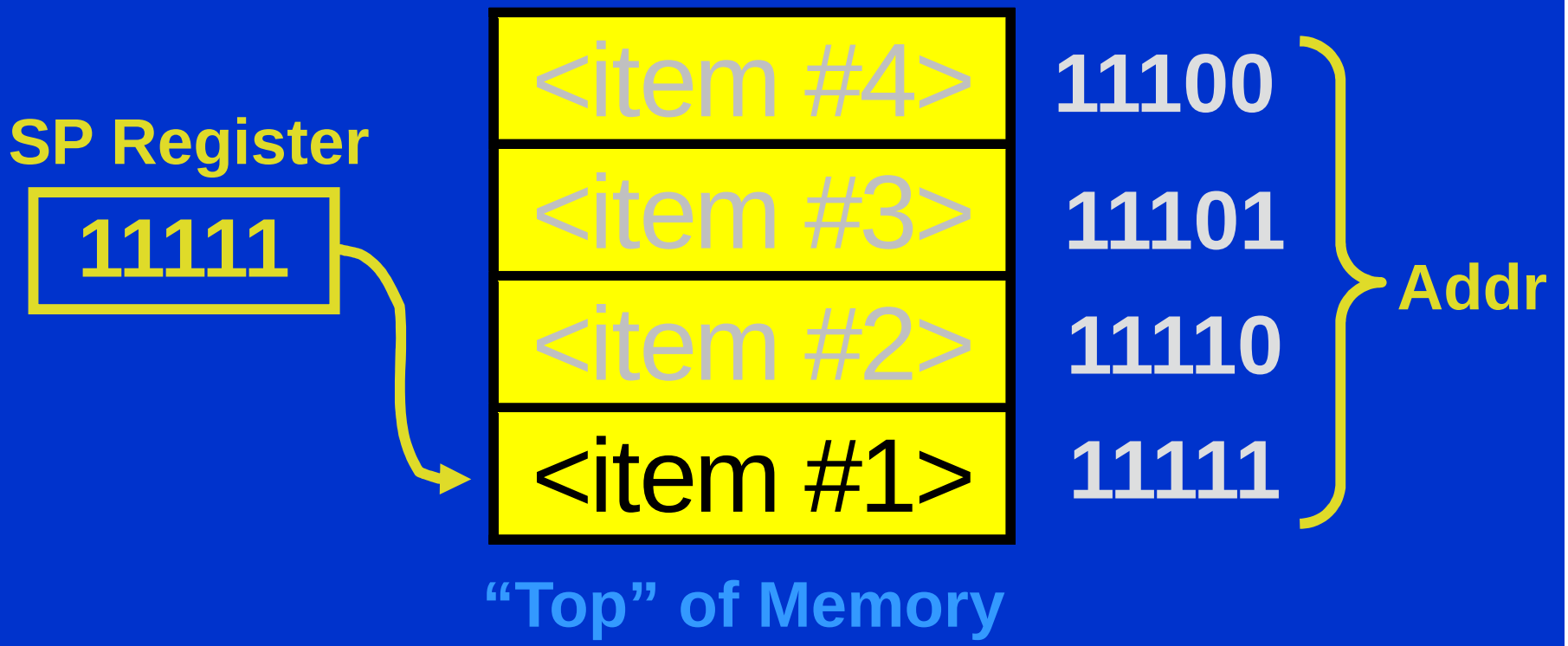


Illustration of Stack Growth

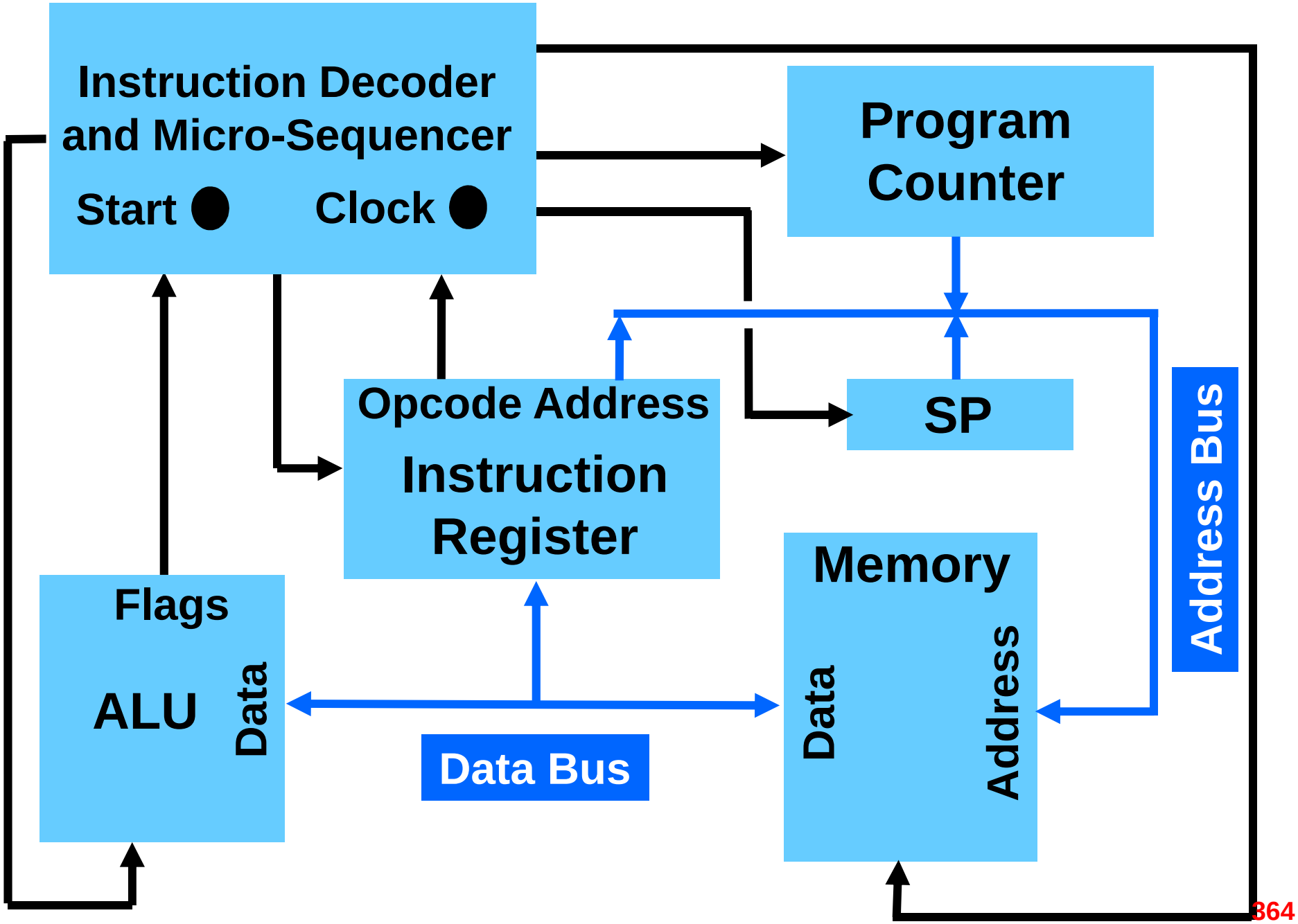
- **After top item removed (stack empty):**

Stack Mechanism

- **To add a stack mechanism to our simple computer, we need a Stack Pointer (SP) register connected to the address bus that has the following control signals:**
  - **SPI: Stack Pointer Increment**
  - **SPD: Stack Pointer Decrement**
  - **SPA: Stack Pointer output on Address bus**
  - **ARS: Asynchronous ReSet**

Note: The *stack empty* condition corresponds to the SP register being cleared to "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
```


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address

Adding Stack Manipulation Instructions

- **The most common stack operations are "push" and "pop" – here, the value that will be *pushed* and *popped* is the A register**
  - **PSH – save value in A register on stack**
    - **Step 1: *Decrement SP register***
    - **Step 2: Store value in A register at the location pointed to by SP register**
  - **POP – load A with value on stack**
    - **Step 1: Load A register from memory location pointed to by SP register**
    - **Step 2: *Increment SP register***

Note: The PSH and POP instructions can be used to implement *expression evaluation*

Adding Stack Manipulation Instructions

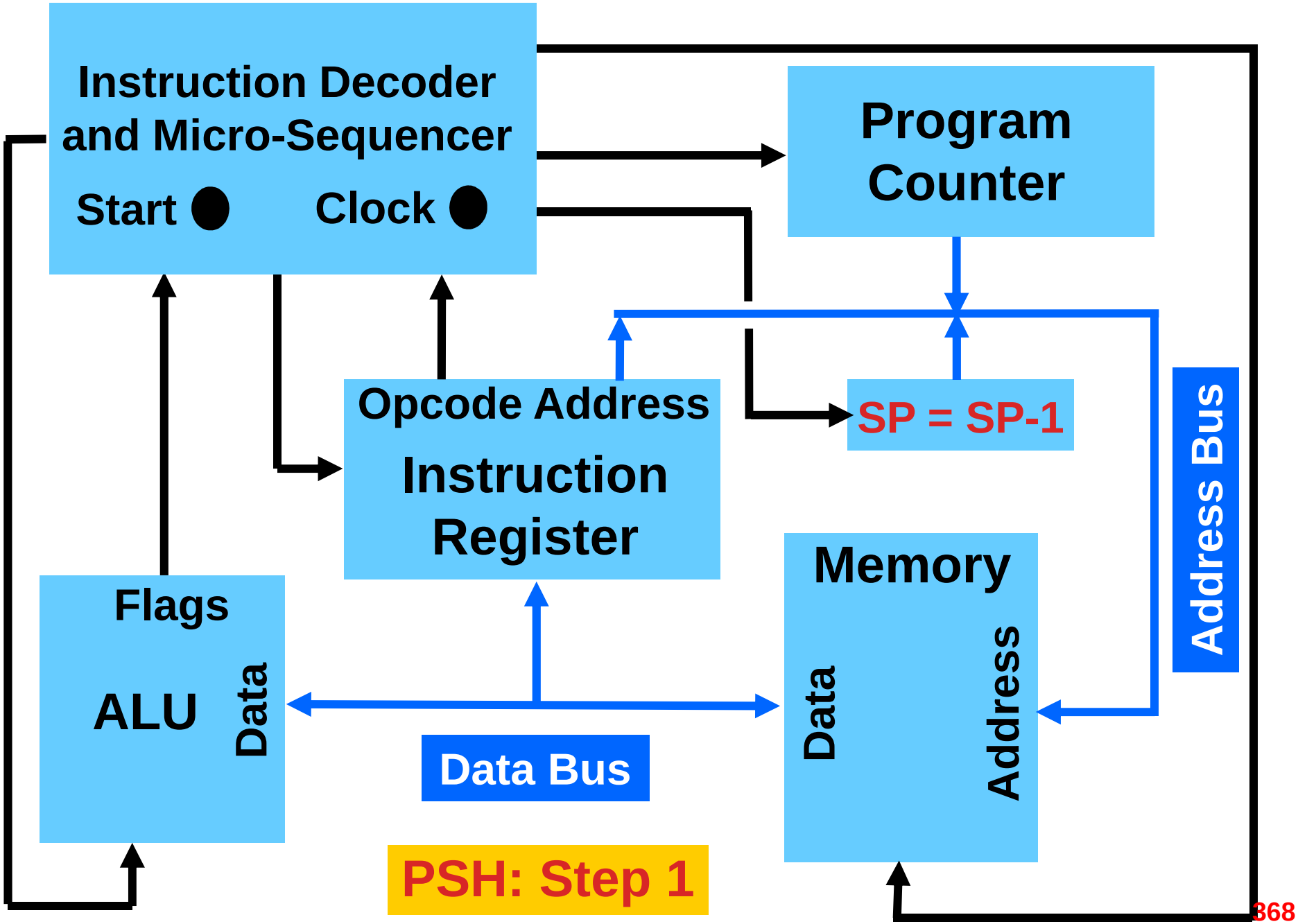
- **The opcodes that will be used for PSH and POP are as follows:**

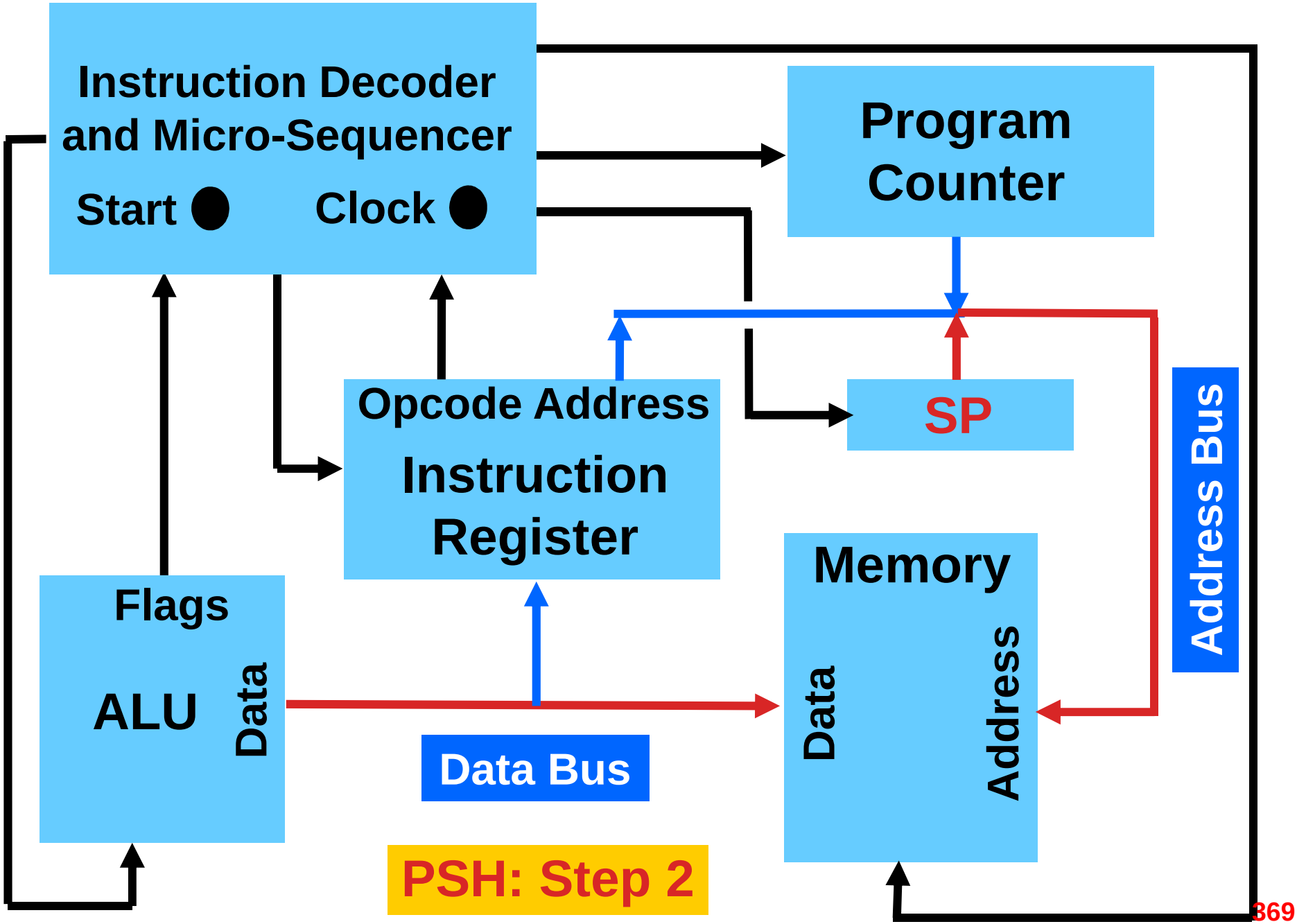
Opcode	Mnemonic	Function Performed
0 0 0	LDA *addr*	Load A with contents of location *addr*
0 0 1	STA *addr*	Store contents of A at location *addr*
0 1 0	ADD *addr*	Add contents of *addr* to contents of A
0 1 1	SUB *addr*	Subtract contents of *addr* from contents of A
1 0 0	AND *addr*	AND contents of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

Adding Stack Manipulation Instructions

- **At first glance, it would appear that *two execute cycles* are required to implement both the PSH and POP instructions**
- **As *good computer engineers**, however, we always need to be on the lookout for operations that can be *overlapped* (subject, of course, to the "rules" we learned earlier):**
  - ***Only one device is allowed to drive a bus during any machine cycle (i.e., "bus fighting" must be avoided)***
  - ***Data cannot pass through more than one (edge-triggered) flip-flop or latch per cycle***

***a.k.a.* "astute digijocks and digijockettes"**

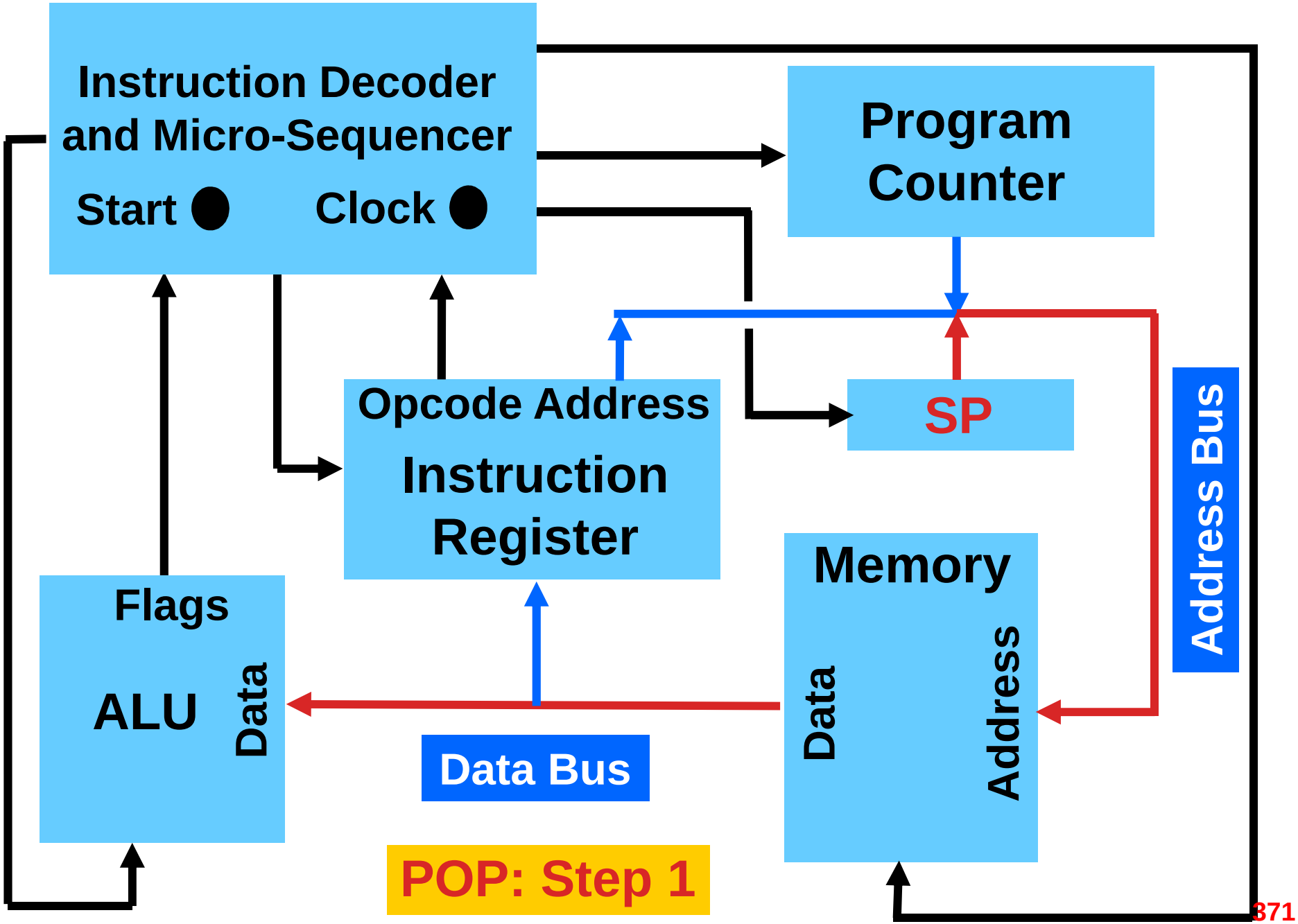

Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP = SP-1
Address Bus
Memory
Data
Address
Flags
ALU
Data
Data Bus
PSH: Step 1

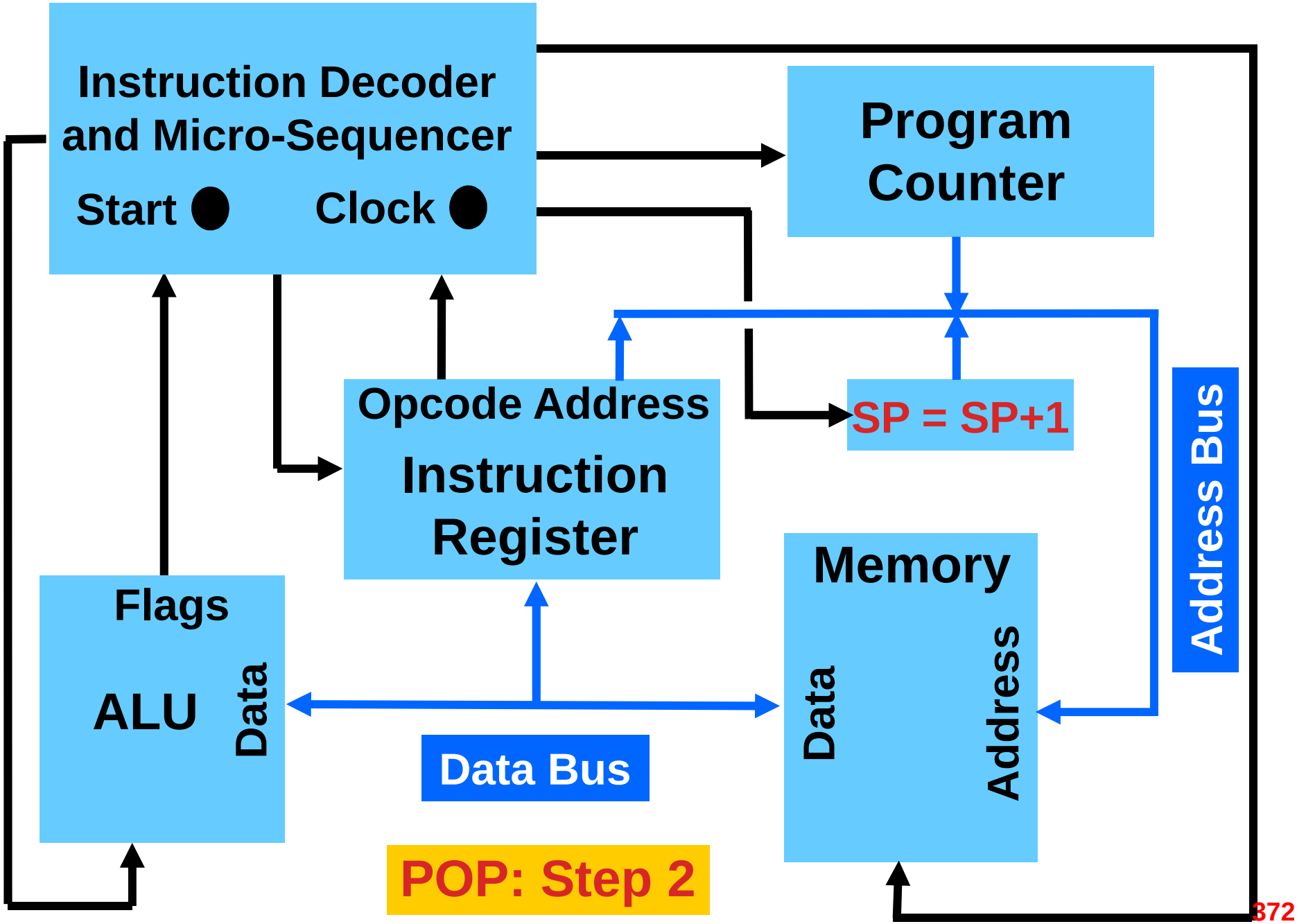

Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Memory
Data
Address
Data Bus
PSH: Step 2

Adding Stack Manipulation Instructions

- **Implementation of PSH requires *two* execute cycles:**
  - **first execute cycle: decrement SP register* (SPD)**
  - **second execute cycle: output "new" value of SP on address bus (SPA), enable a memory write operation (MSL and MWE), and tell the ALU to output value in the A register on the data bus (AOE)**

***Note: The "new" value of SP must be available (and stable) *before* it can be used to address memory**


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Memory
Data
Address
Data Bus
POP: Step 1


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP = SP+1
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address
POP: Step 2

Adding Stack Manipulation Instructions

- **Implementation of POP requires only *one* execute cycle:**
  - **first execute cycle: output value of SP on address bus (SPA), enable a memory read operation (MSL and MOE), tell the ALU to load the A register with the value on the data bus (ALE and ALX), and tell the SP register to increment* (SPI)**

***Note: The SP register will be incremented *after* the A register is loaded with the contents of the location pointed to by the SP register, i.e., the SP increment is *overlapped* with the fetch of the next instruction**

Adding Stack Manipulation Instructions

- **Modified system control table:**

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

<u>Note</u>: Recall that the RST signal is used to *synchronously reset* the state counter on the *final execute cycle* of each instruction

```
" System control equations

MSL = RUN.q&(S0 # S1&(LDA # STA # ADD # SUB # AND # POP)# S2&PSH);
MOE = S0 # S1&(LDA # ADD # SUB # AND # POP);
MWE = S1&STA # S2&PSH;
ARS = START;
PCC = RUN.q&S0;
POA = S0;
IRL = RUN.q&S0;
IRA = S1&(LDA # STA # ADD # SUB # AND);
AOE = S1&STA # S2&PSH;
ALE = RUN.q&S1&(LDA # ADD # SUB # AND # POP);
ALX = S1&(LDA # AND # POP);
ALY = S1&(SUB # AND);

SPI = S1&POP;
SPD = S1&PSH;
SPA = S1&POP # S2&PSH;

RST = S1&(LDA # STA # ADD # SUB # AND # POP) # S2&PSH;

END
```

Clicker Quiz

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

Adding Subroutine Linkage Instructions

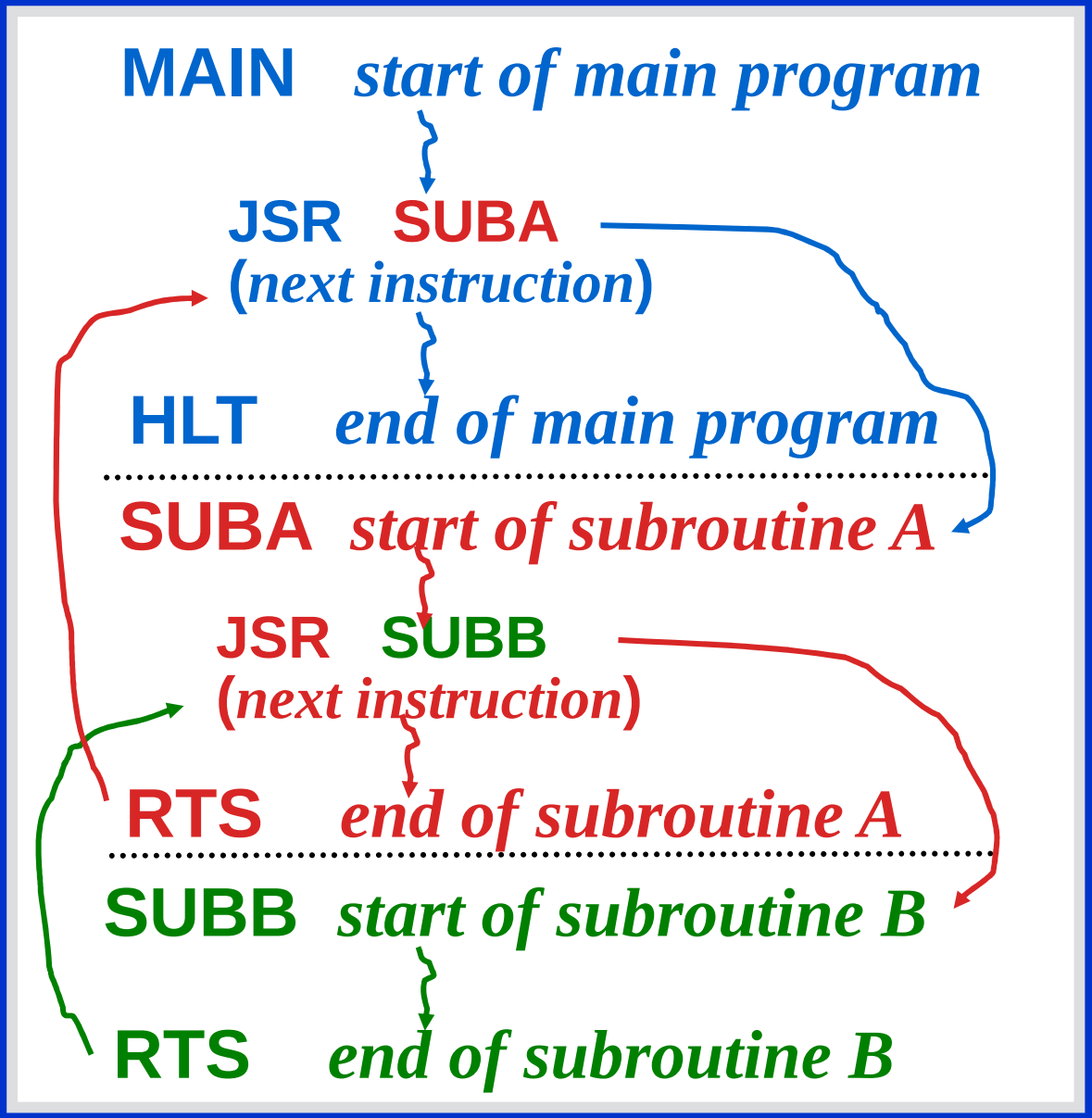
- **Why use a stack as a subroutine linkage mechanism? There are several important capabilities that a stack affords:**
  - ***arbitrary nesting*** **of subroutine calls**
  - ***passing parameters*** **to subroutines**
  - ***recursion*** **(the ability of a subroutine to call itself) – made possible by passing parameters via the stack**
  - ***reentrancy*** **(the ability of a code module to be shared among quasi-simultaneously executing tasks) – made possible by storing temporary local variables on the stack**

Adding Subroutine Linkage Instructions

- **The opcodes that will be used for JSR ("jump to subroutine") and RTS ("return from subroutine") are as follows:**

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
1 1 0	JSR *addr*	Jump to subroutine at location ***addr***
1 1 1	RTS	Return from subroutine

Subroutine Linkage in Action



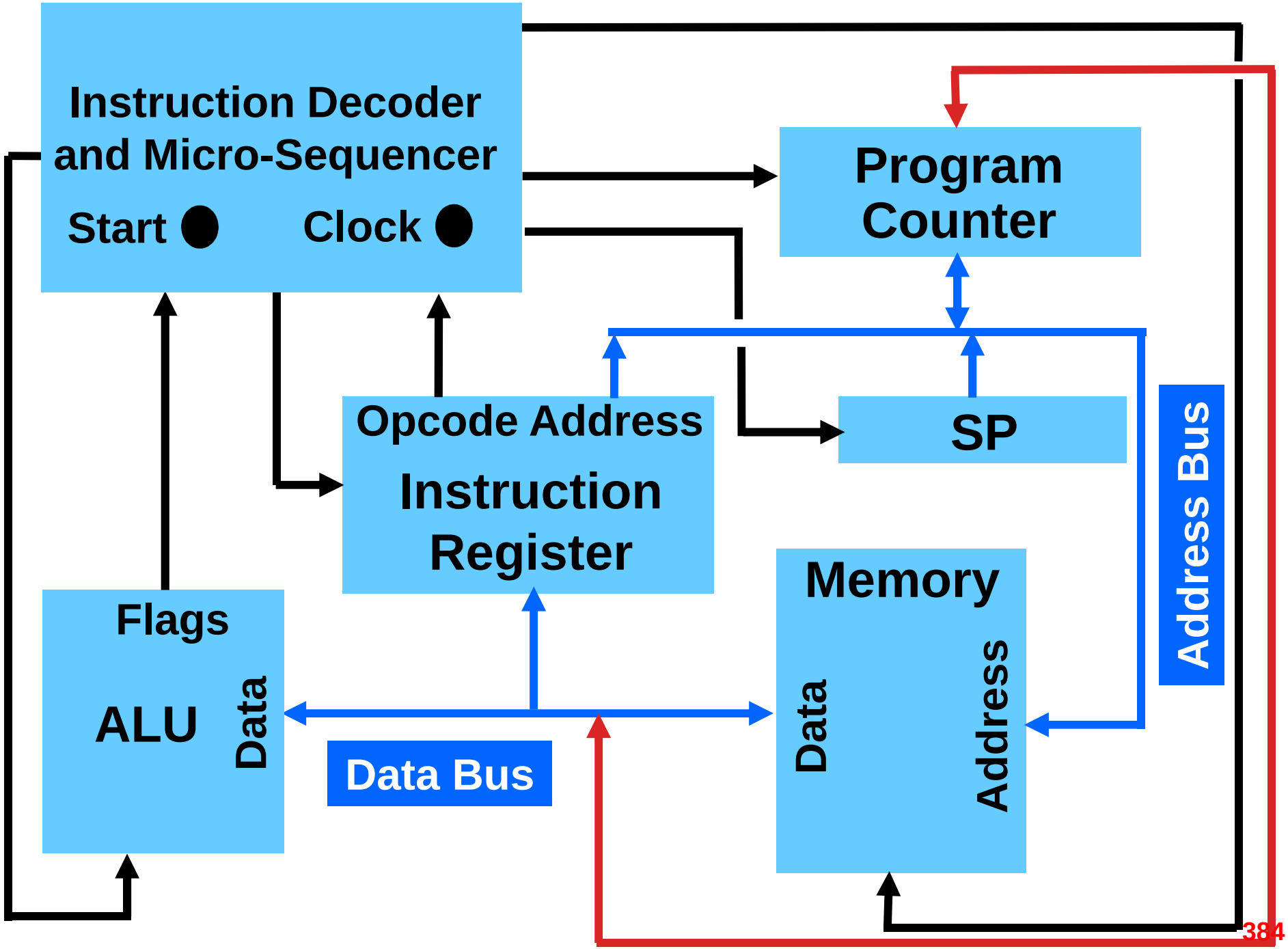
Adding Subroutine Linkage Instructions

- **Subroutine "CALL" and "RETURN"**
  - **JSR *addr* – jump to subroutine at memory location *addr***
    - **Step 1: Decrement *SP register***
    - **Step 2: Store return address* at location pointed to by SP register**
    - **Step 3: Load PC with *value in IR address field***
  - **RTS – return from subroutine**
    - **Step 1: Load PC from *memory location pointed to by SP register***
    - **Step 2: Increment SP register**

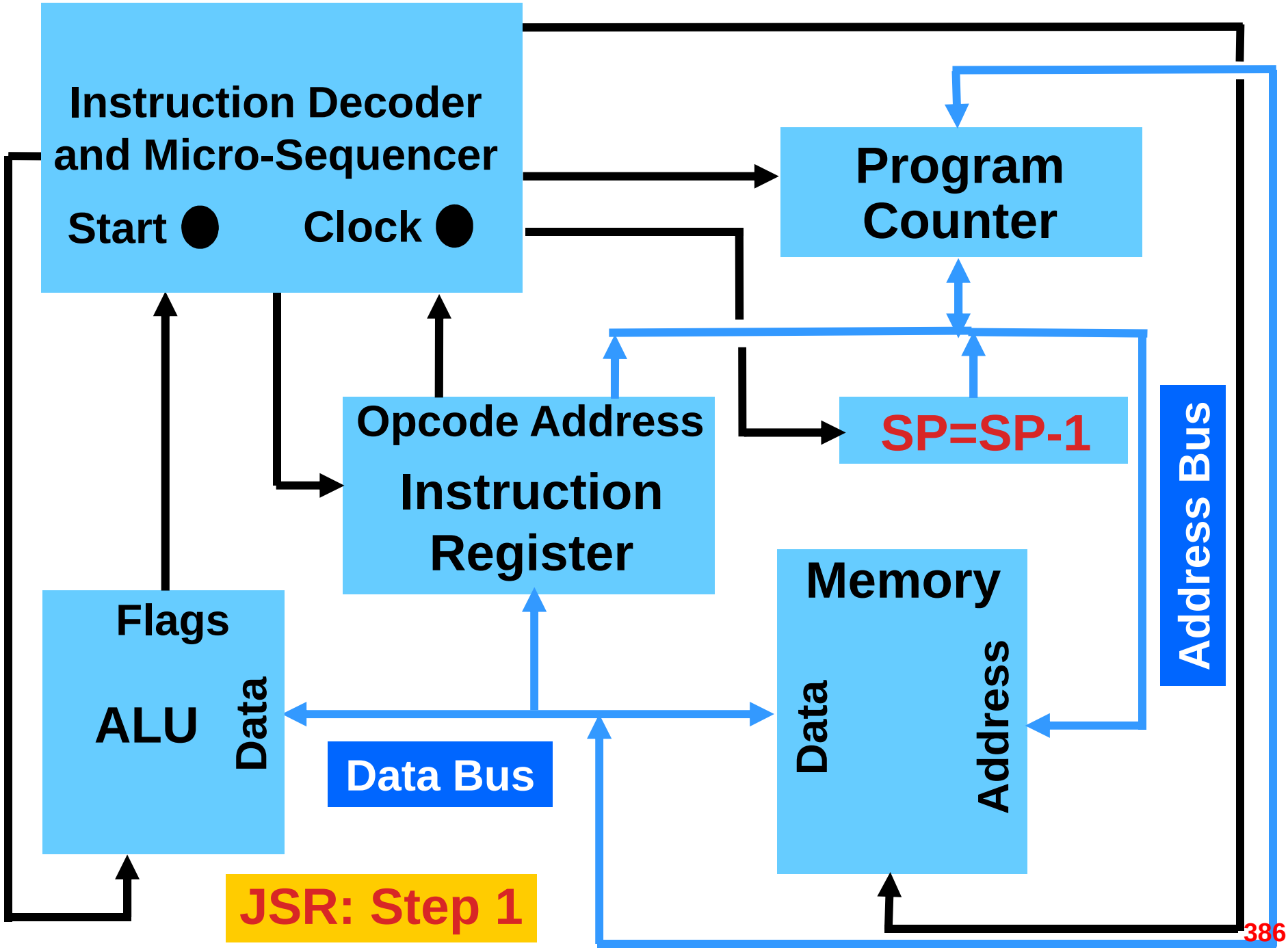
***current value in PC, which was *incremented* during the fetch cycle (points to *next instruction*)**

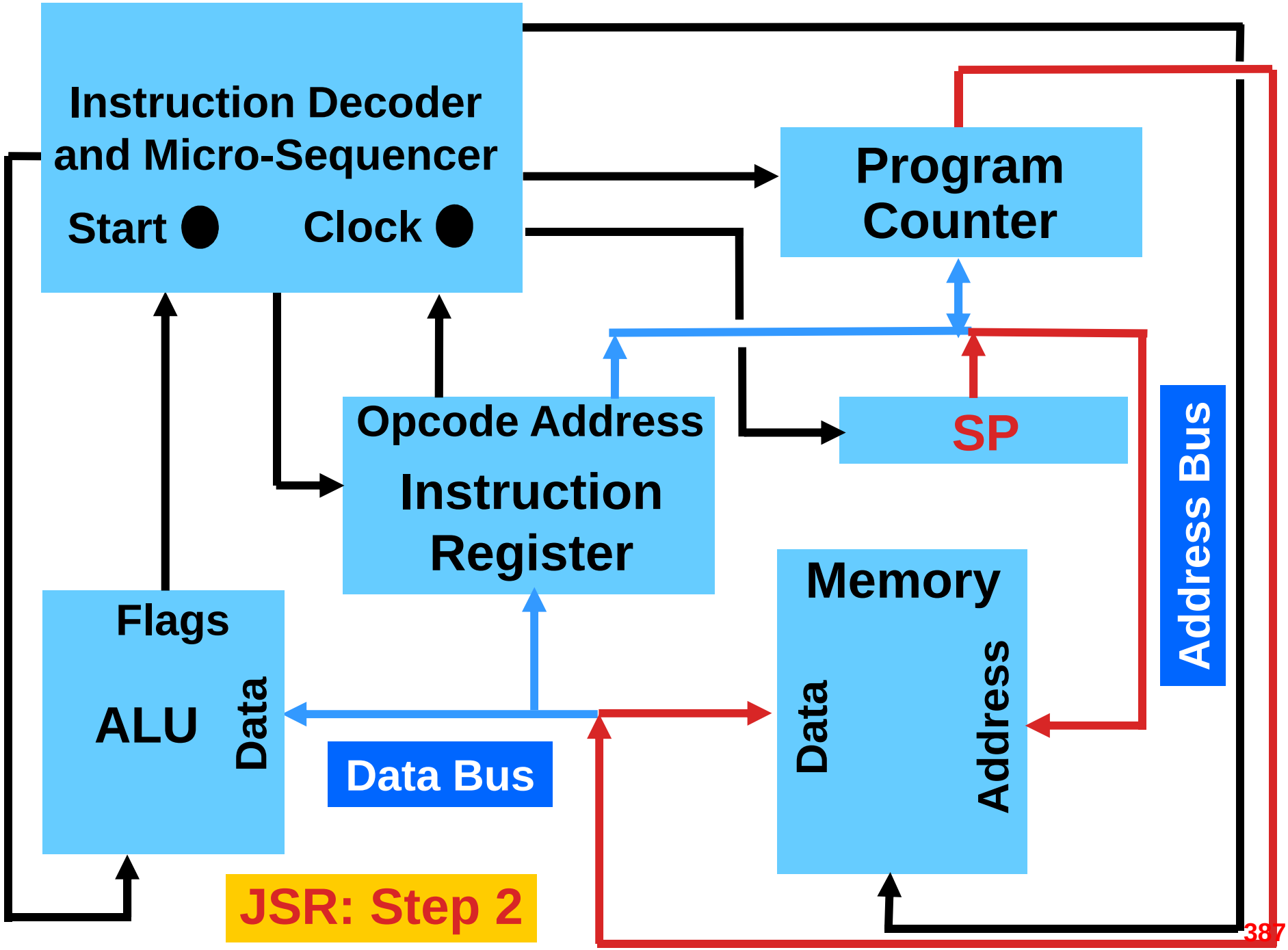
Adding Subroutine Linkage Instructions

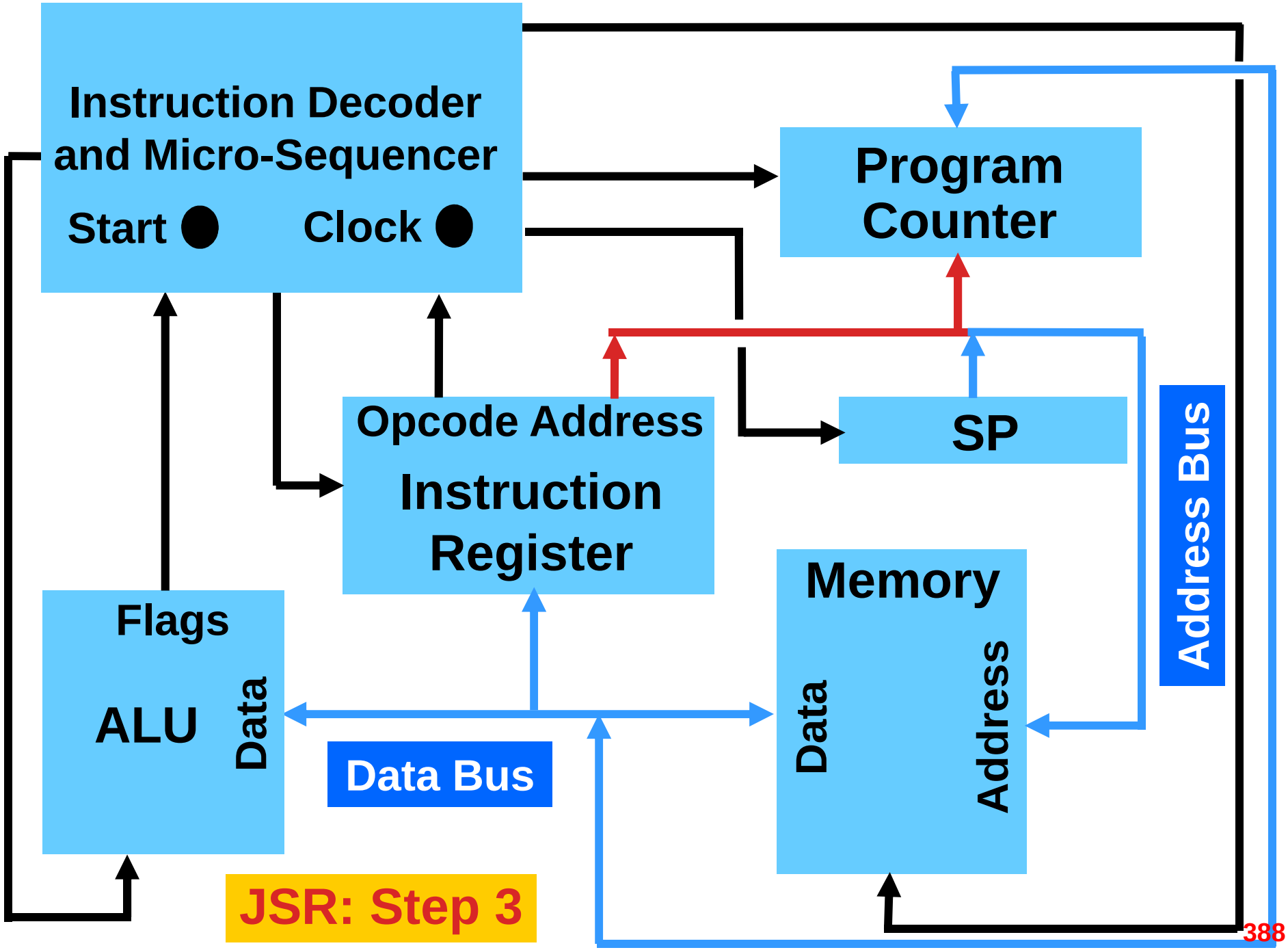
- The ***astute*** digijock(ette) will realize at this point that the program counter needs to be ***modified*** – a way to save its value on the stack, and subsequently restore it, must be provided
- Two new PC control signals need to be implemented:
  - **POD:** output value of PC on data bus
  - **PLD:** load PC with value on data bus


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address

```
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
```

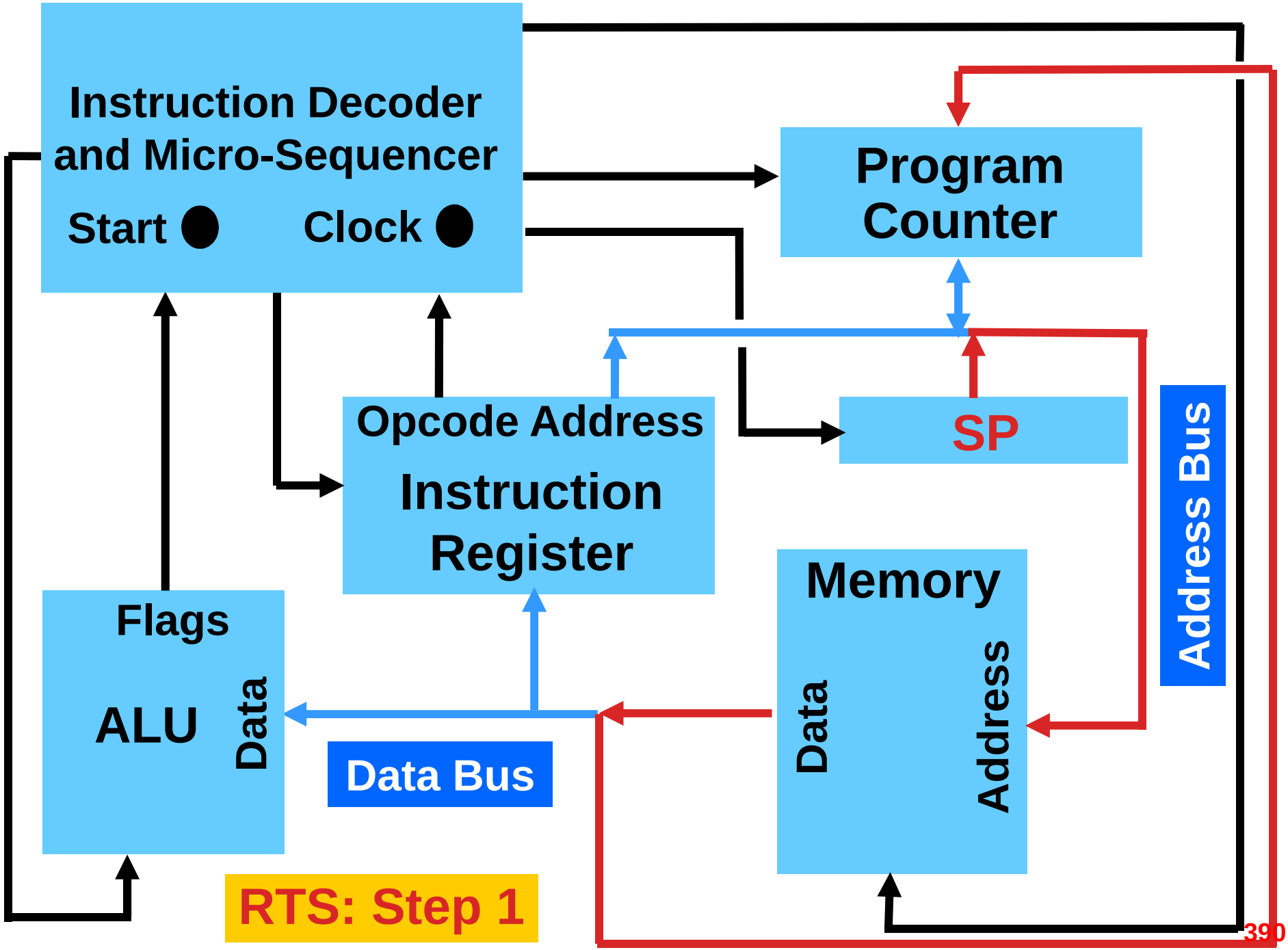

Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP=SP-1
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address
JSR: Step 1

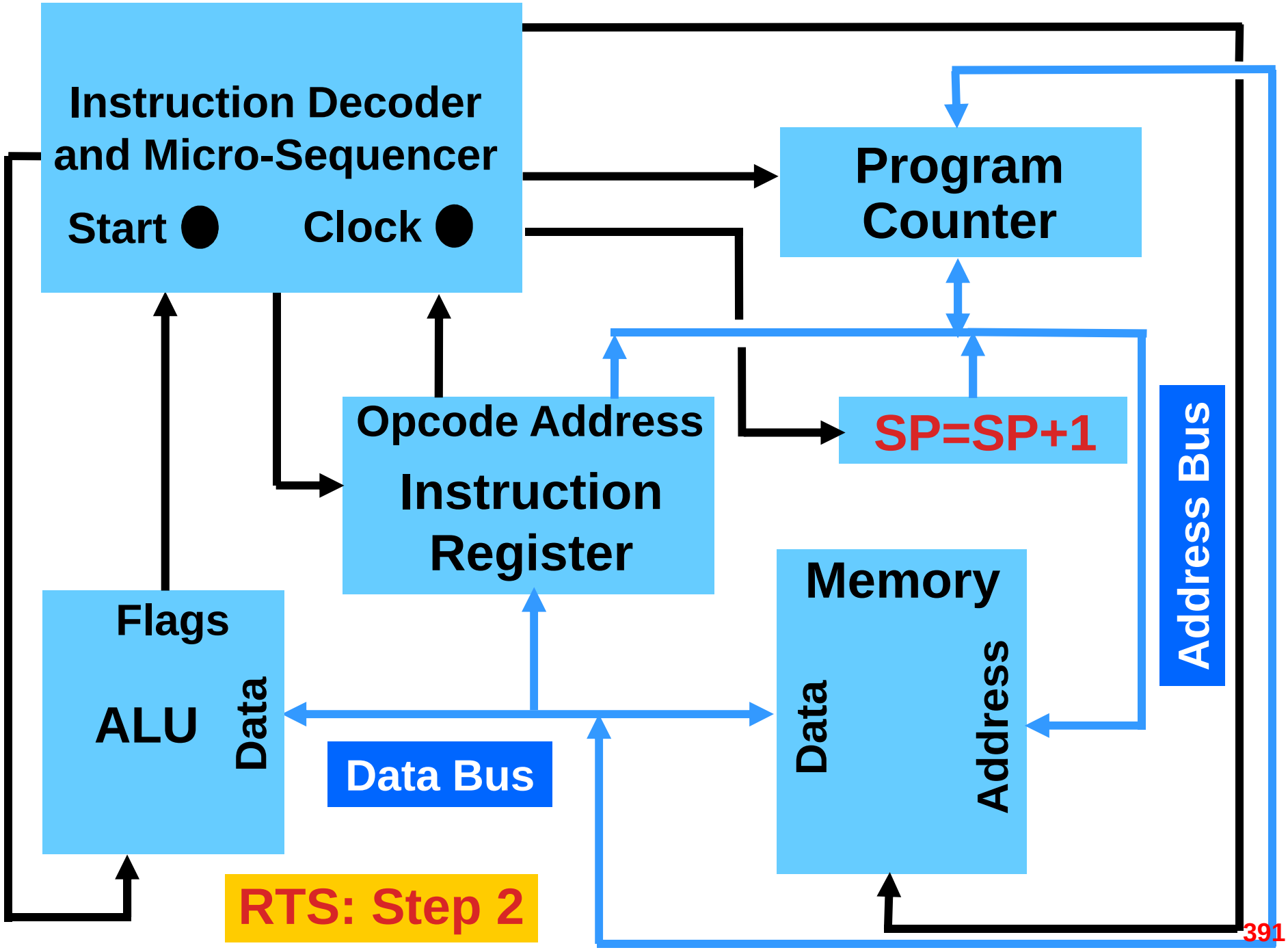

Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address
JSR: Step 2


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Flags
ALU
Data
Data Bus
Memory
Data
Address
JSR: Step 3

Adding Subroutine Linkage Instructions

- **Implementation of JSR requires *three* execute cycles:**
  - **first execute cycle: decrement SP register (SPD)**
  - **second execute cycle: output "new" value of SP on address bus (SPA), enable a memory write operation (MSL and MWE), and tell the PC register to output its value on the data bus (POD)**
  - **third execute cycle: tell IR register to output its operand field on the address bus (IRA), and tell the PC register to load the value on the address bus (PLA)**


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP
Address Bus
Memory
Data
Address
Flags
ALU
Data
Data Bus
RTS: Step 1


Instruction Decoder
and Micro-Sequencer
Start
Clock
Program
Counter
Opcode Address
Instruction
Register
SP=SP+1
Address Bus
Memory
Data
Address
Flags
ALU
Data
Data Bus
RTS: Step 2

Adding Subroutine Linkage Instructions

- **At first glance, it would appear that *two execute cycles* are required to implement the RTS instruction**
- **As *astute* digijock(ette)s, however, we always need to be on the lookout for operations that can be *overlapped* (subject, of course, to the "rules" we learned earlier):**
  - ***Only one device is allowed to drive a bus during any machine cycle (i.e., "bus fighting" must be avoided)***
  - ***Data cannot pass through more than one (edge-triggered) flip-flop or latch per cycle***

Adding Subroutine Linkage Instructions

- **Implementation of RTS requires only *one* execute cycle:**
  - **first execute cycle: output value of SP on address bus (SPA), enable a memory read operation (MSL and MOE), tell the PC to load the value on the data bus (PLD), and tell the SP register to increment* (SPI)**

***Note: The SP register will be incremented *after* the PC register is loaded with the contents of the location pointed to by the SP register, i.e., the SP increment is *overlapped* with the fetch of the next instruction**

Adding Subroutine Linkage Instructions

- **Modified system control table:**

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

<u>Note</u>: Recall that the RST signal is used to *synchronously reset* the state counter on the *final execute cycle* of each instruction

```
" System control equations

MSL = RUN.q&(S0 # S1&(LDA # STA # ADD # SUB # AND # RTS) # S2&JSR);
MOE = S0 # S1&(LDA # ADD # SUB # AND # RTS);
MWE = S1&STA # S2&JSR;
ARS = START;
PCC = RUN.q&S0;
POA = S0;

PLA = S3&JSR;
POD = S2&JSR;
PLD = S1&RTS;

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);

SPI = S1&RTS;
SPD = S1&JSR;
SPA = S1&RTS # S2&JSR;

RST = S1&(LDA # STA # ADD # SUB # AND # RTS) # S3&JSR;

END
```

Clicker Quiz

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”!