CS2022 Computer Architecture I

Computer Architecture

- Lecturer: Dr. Michael Manzke
  - Office: Lloyd Institude, Room 047
  - Ext: 2400
  - Email: michael.manzke@cs.tcd.ie
  - HTML: http://www.cs.tcd.ie/Michael.Manzke/index.php/mm-teaching/undergraduate/cs2022

CS2022 CS2022 Exam

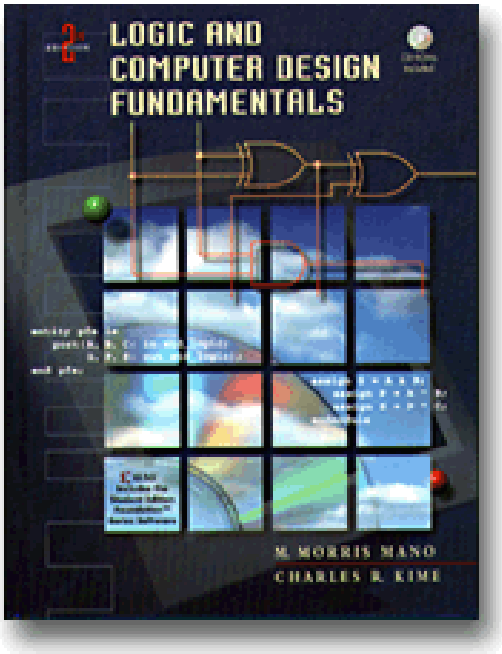
- 4 Questions
- You must answer 3 questions

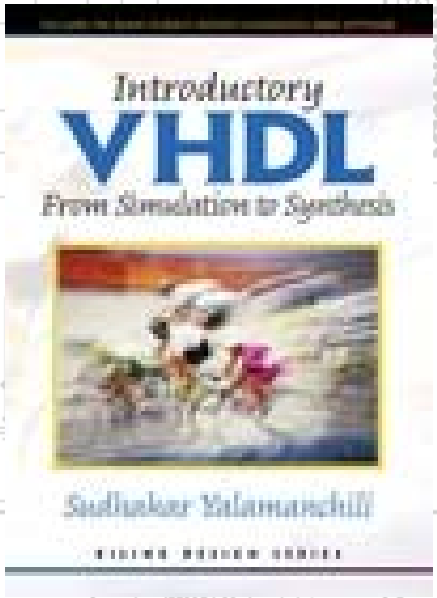
CS2022 CS2022- Exam

- Questions will seek to establish that you have a good grasp of the following concepts:
  - Microoperations
  - Datapath
  - Busses
  - Alu/shift design
  - Status bit generation and use
  - Control unit design and operation

CS2022 Course Text:

- "Introductory VHDL: From Simulation to Synthesis"
- "Logic and Computer Design Fundamentals" 2nd Edition updated, Mano (includes Xilinx Student Edition 4.2i software)





CS2022 Organisation of the Course

- Lectures:
  - Relate to chapter 7 and 8 of Mano & Kime textbook
  - Prerequisite:
    - Chapter 1-6 of Mano & Kime textbook
- Tutorials:
  - Will largely be problems from Mano & Kime textbook and the two projects.
- Projects:
  - There will be two in which you will be asked to design and simulate key elements of a processor.

CS2022 Assessment

- By examination and coursework
- Ratio 80% to 20% respectively

CS2022 Course Objective

- The aim is to give you a good understanding of the design and operation of an instruction processing unit and the functional subsystems which execute these instructions.

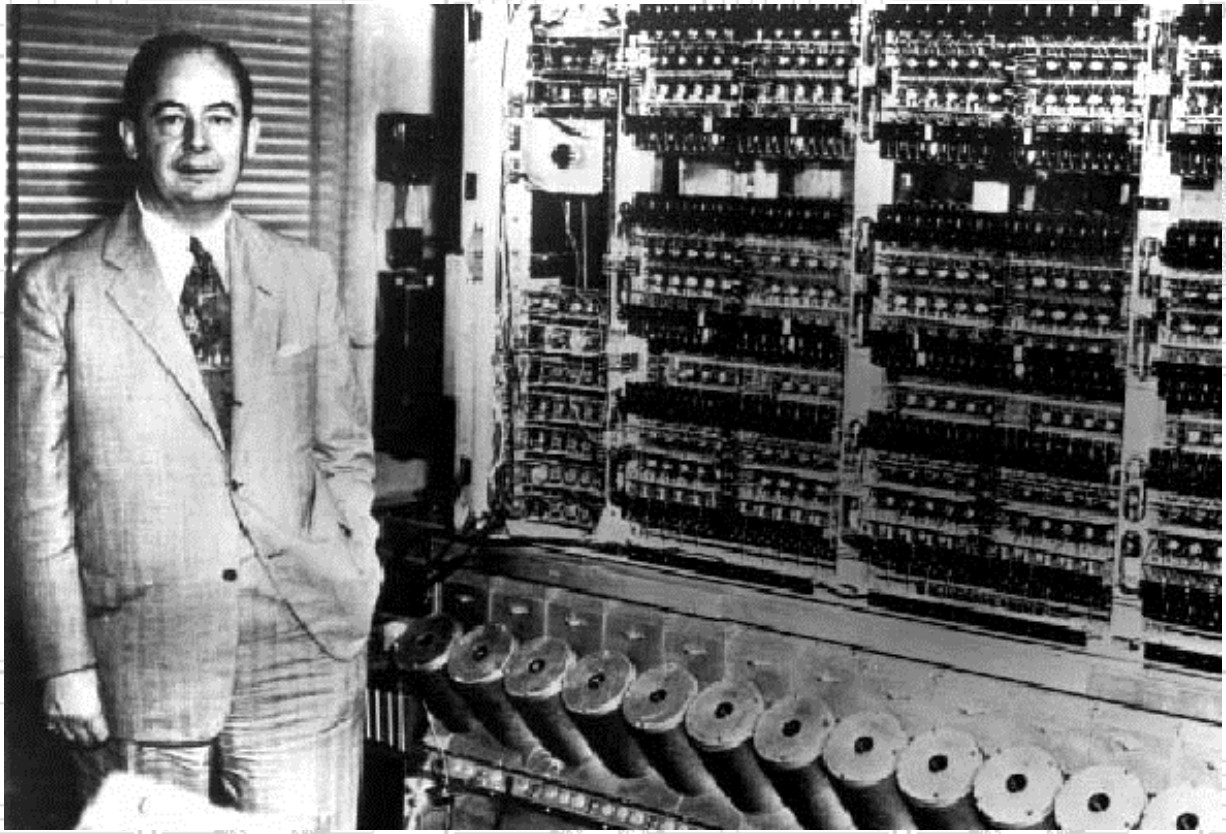
CS2022 Von Neumann Architecture

- This design was first specified by John von Newmann in 1940 and so the resulting architecture is often called the ‘von Neumann’ architecture.
- Please see the following web page:
  - **http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Von_Neumann.html**

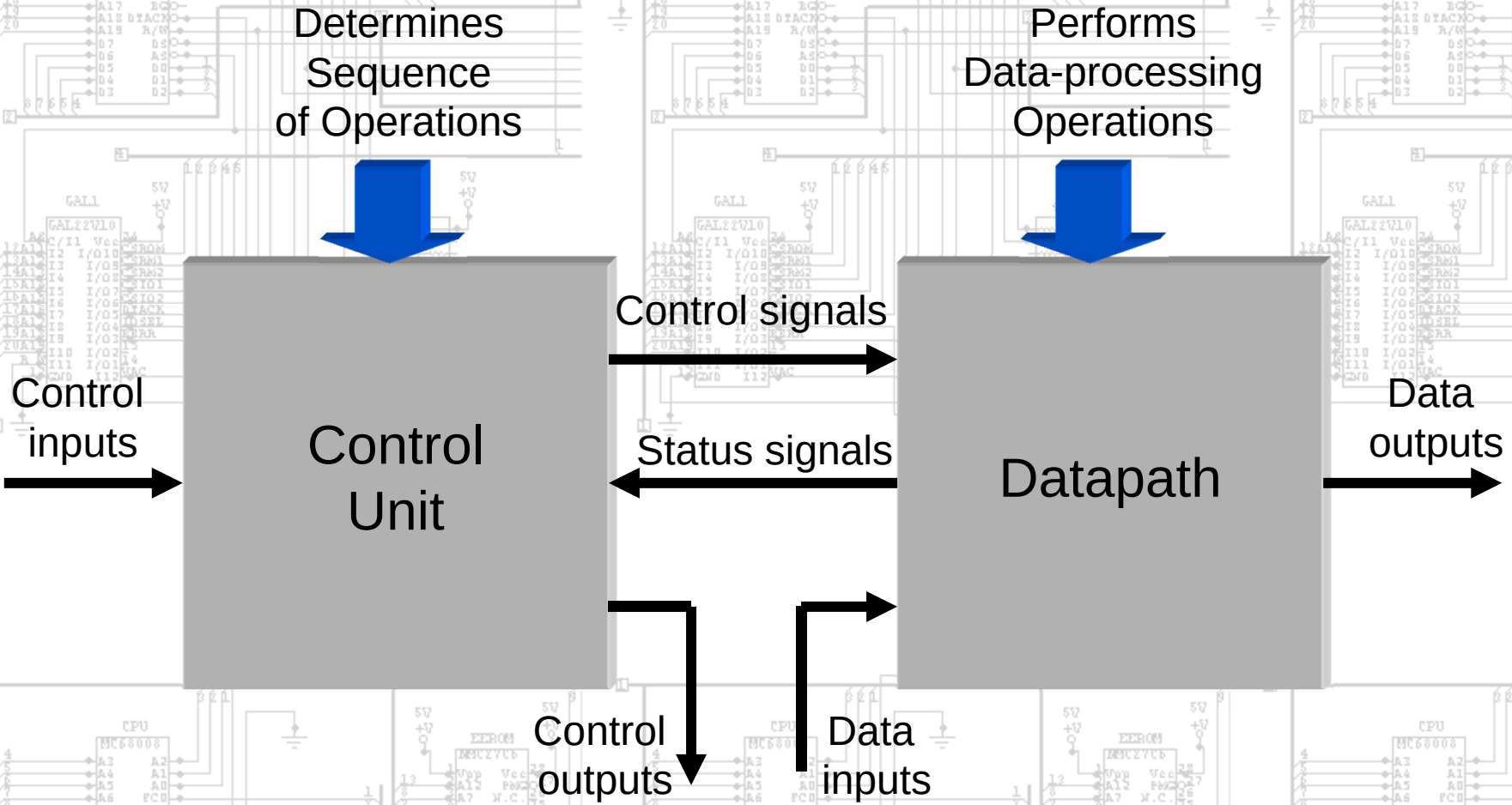
John von Neumann

Born: 28 Dec 1903 in Budapest, Hungary

Died: 8 Feb 1957 in Washington D.C., USA



CS2022 Datapath and Control Unit



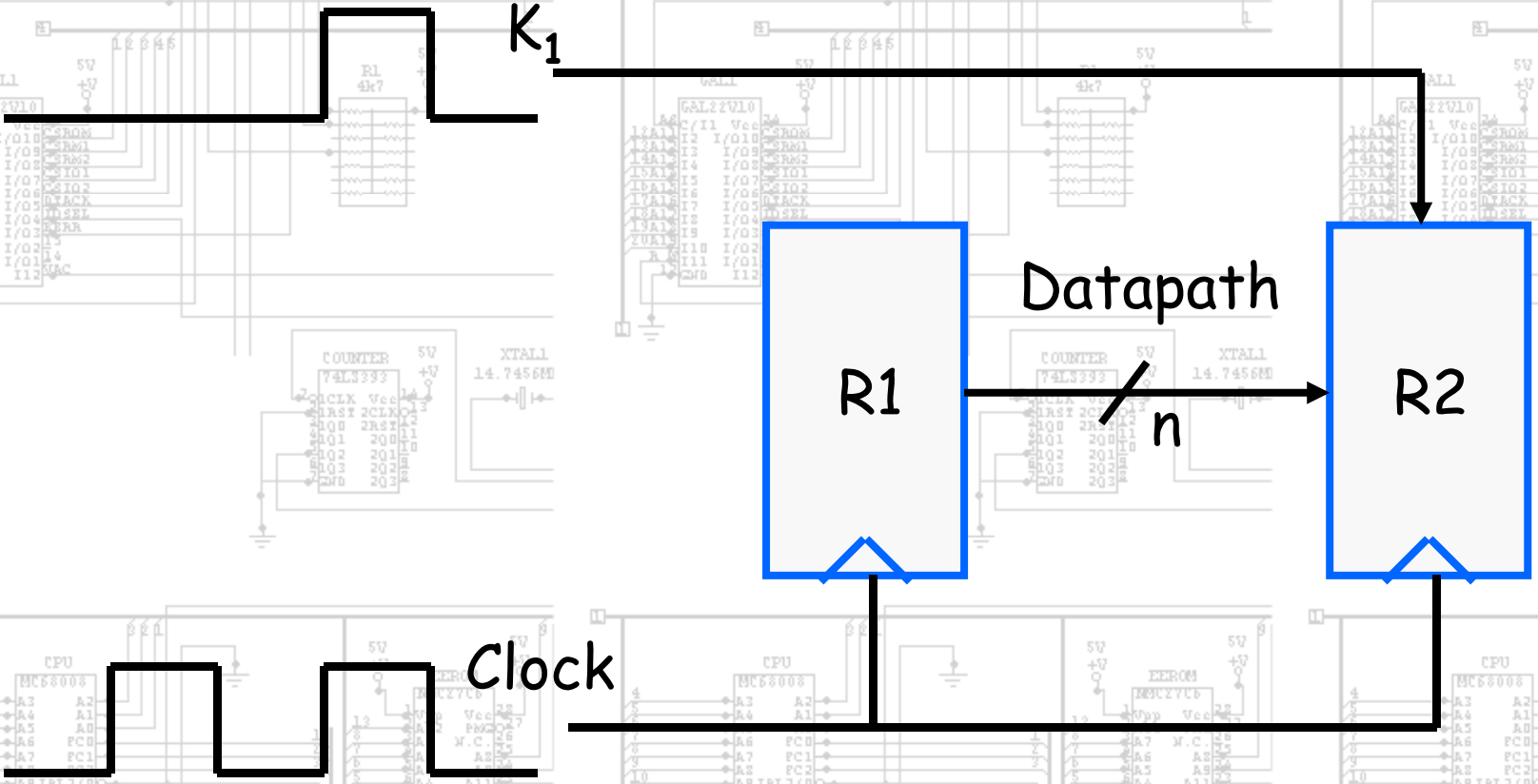
CS2022 Register Transfer

- Describing large-scale processor activity.
- To discuss digital systems of this scale and level of complexity we need a number of descriptive tools.
- For example:

  a) Circuit schematics highlight the circuit components and their connectivity.

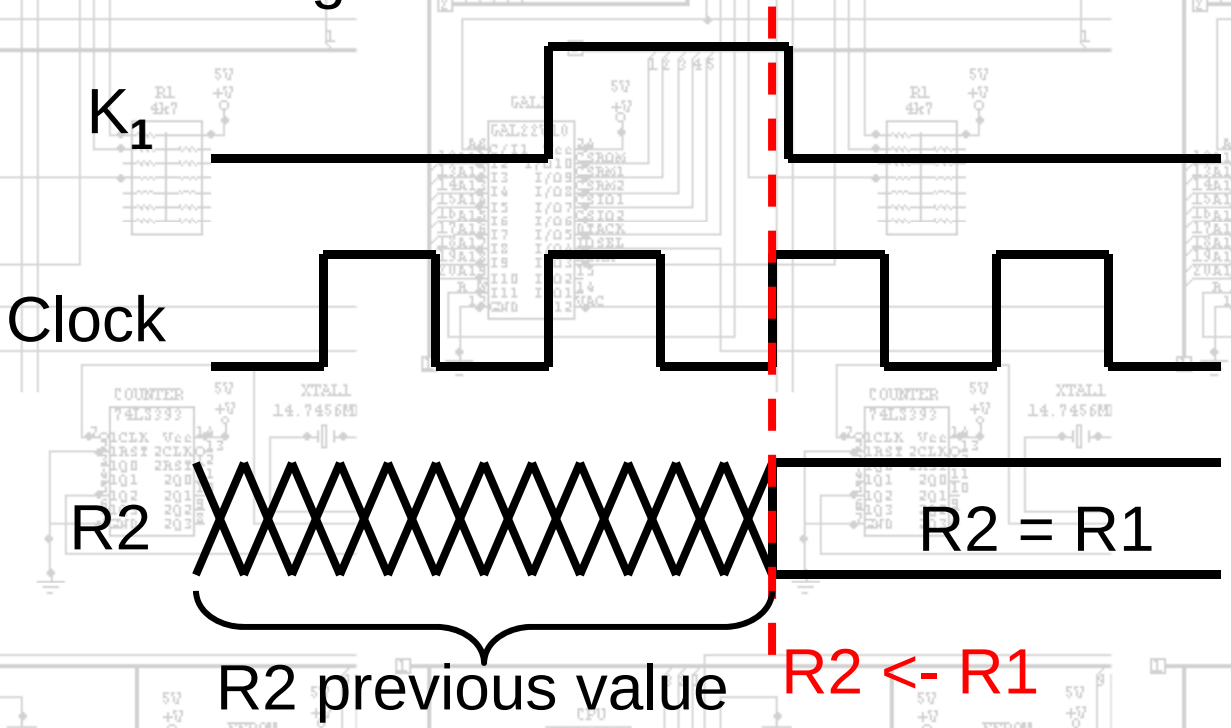
Transfer from R1 to R2 when K_1=1

Circuit Schematic



CS2022 Timing Diagram

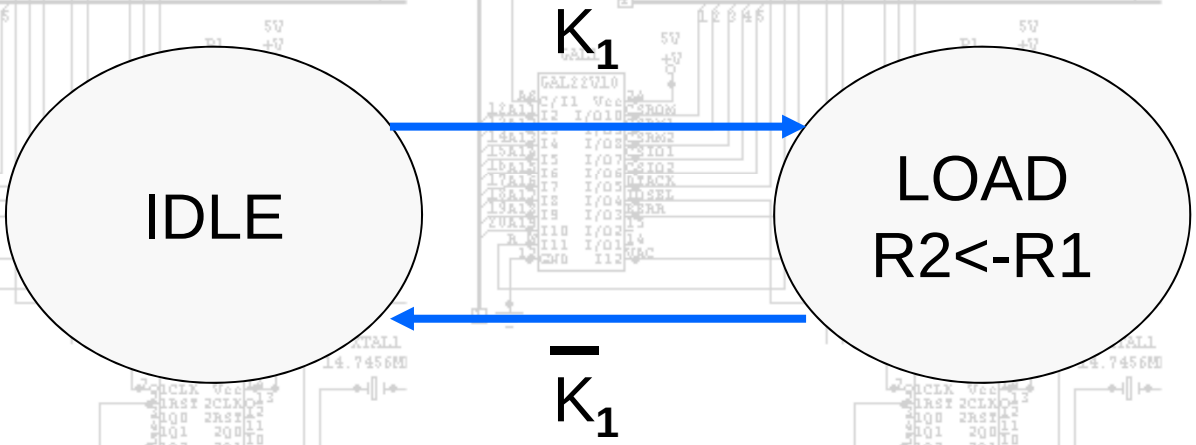
b) Timing diagrams highlight the detailed time sequence of transfer between registers.



The transfer R2 <-r1 occurs at the end of K_1

CS2022 State Diagram

c) State diagrams highlight the modes of operation and their control

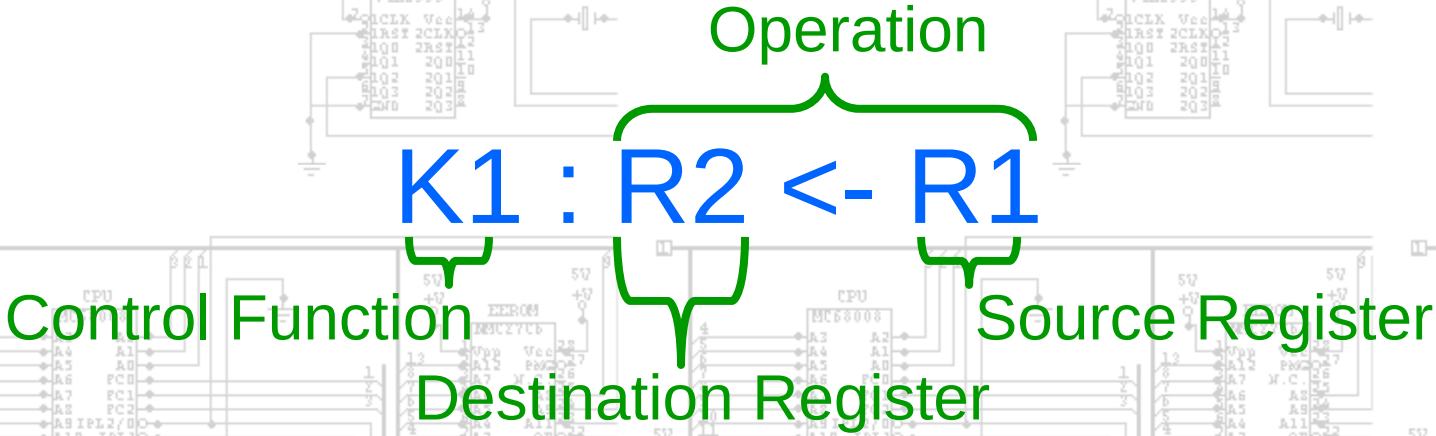


When the system is synchronous we normally omit the clock specification.

i.e. $K_1 \equiv K_1.CLOCK\uparrow$

Register Transfer Specification

- Source Register
- Destination Register
- Operation to be applied
- Condition or control function under which the transfer will occur.
  - We assume synchronous operation and omit the clock



Building Register-Transfer Statements

Symbol(s)	Description	Examples
Letters and Numerals	Denote Registers	AR, DR, R2, IR
Parentheses	Denote sections of Registers	R2(9), AR(2),R1(7:0)
Arrow	Denotes data transfer	R1<-R2 IR<-DR
Comma	Separates simultaneous transfers	R1<-R2, R3<-AR
Square brackets	Denote memory addressing	DR<-M[AR] /* a read M[AR]<-DR /* a write

CS2022 VHDL and RTL

Operation	RTL	VHDL
Combinational Assignment	=	<=(concurrent)
Register Transfer	←	<=(concurrent)
Addition	+	+
Subtraction	-	-
Bitwise AND	^	and
Bitwise OR	∨	or
Bitwise XOR	⊕	xor
Bitwise NOR	$\overline{\oplus}$	not
Shift left (logical)	sl	sll
Shift right (logical)	sr	srl
Vector/Register	A(3:0)	A(3 downto 0)
Concatenation	||	&

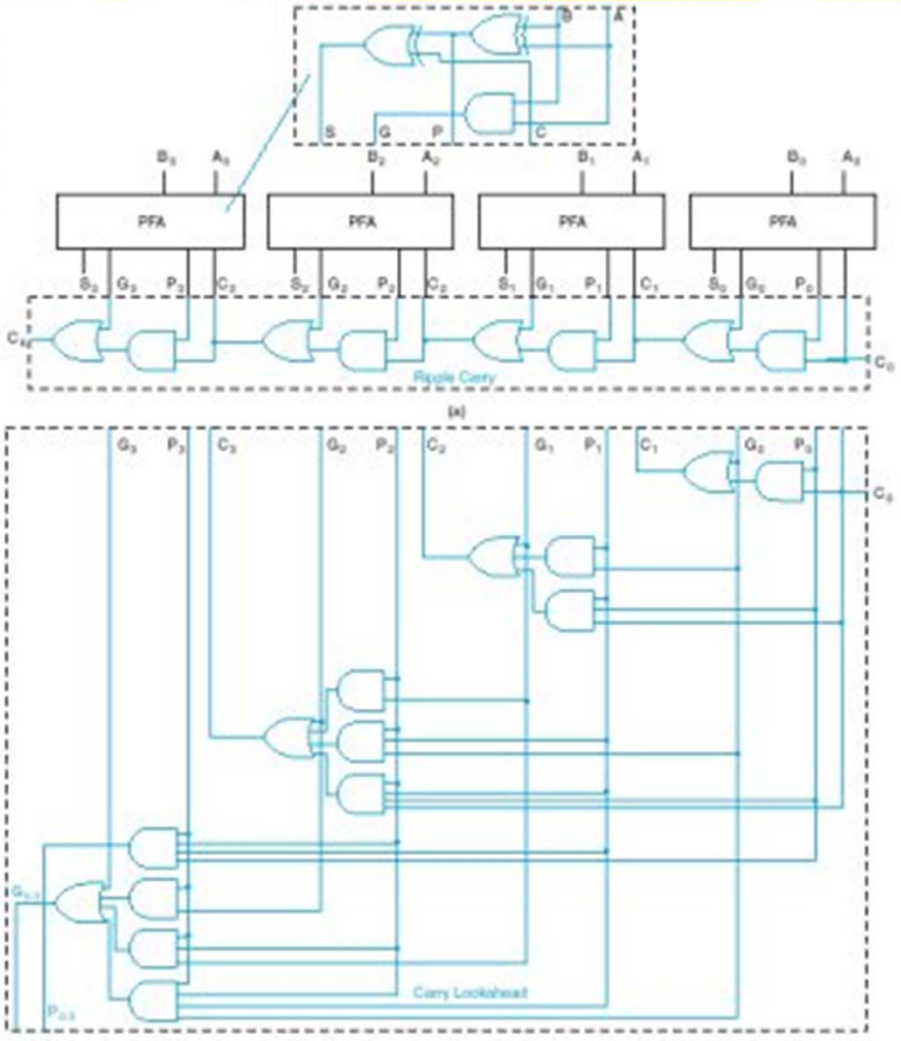
CS2022 Micro-Operation

- A micro-operation is an operation which can be accomplished within a small number of gate propagation delays upon data stored in adjacent registers and memory.
- Those commonly encountered in digital systems divide naturally into four groups
  - Transfer or identity micro-ops copy data, e.g. R1<-R2, DR<-M[AR]
  - Arithmetic micro-ops provide the elements of arithmetic, e.g. R0<-R1+R2
  - Logic micro-0ps provide per bit opearation, e.g. R1<-R2 or R2
  - Shift micro-ops provide bit rotations, e.g. R1<-sr R2, R0<- rol R1

CS2022 Arithmetic Micro-ops

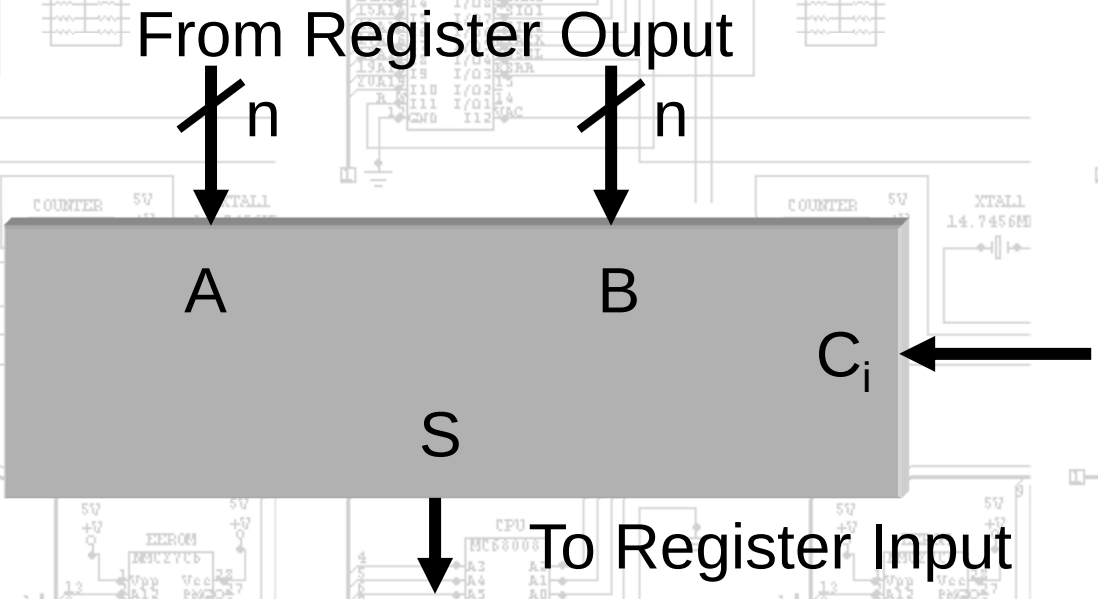
- These are operations which can be accomplished with a full-adder, which, with carry lookahead logic, can be made to deliver a substantial result, e.g. 64-bit in just a few gate delays.

CS2022 Carry Lookahead Adder



CS2022 CLA

- Let R0, R1,R3 be n-bit Register and consider what can be done with an n-bit CLA (carry lookahead adder)



Conditioned use of CLA

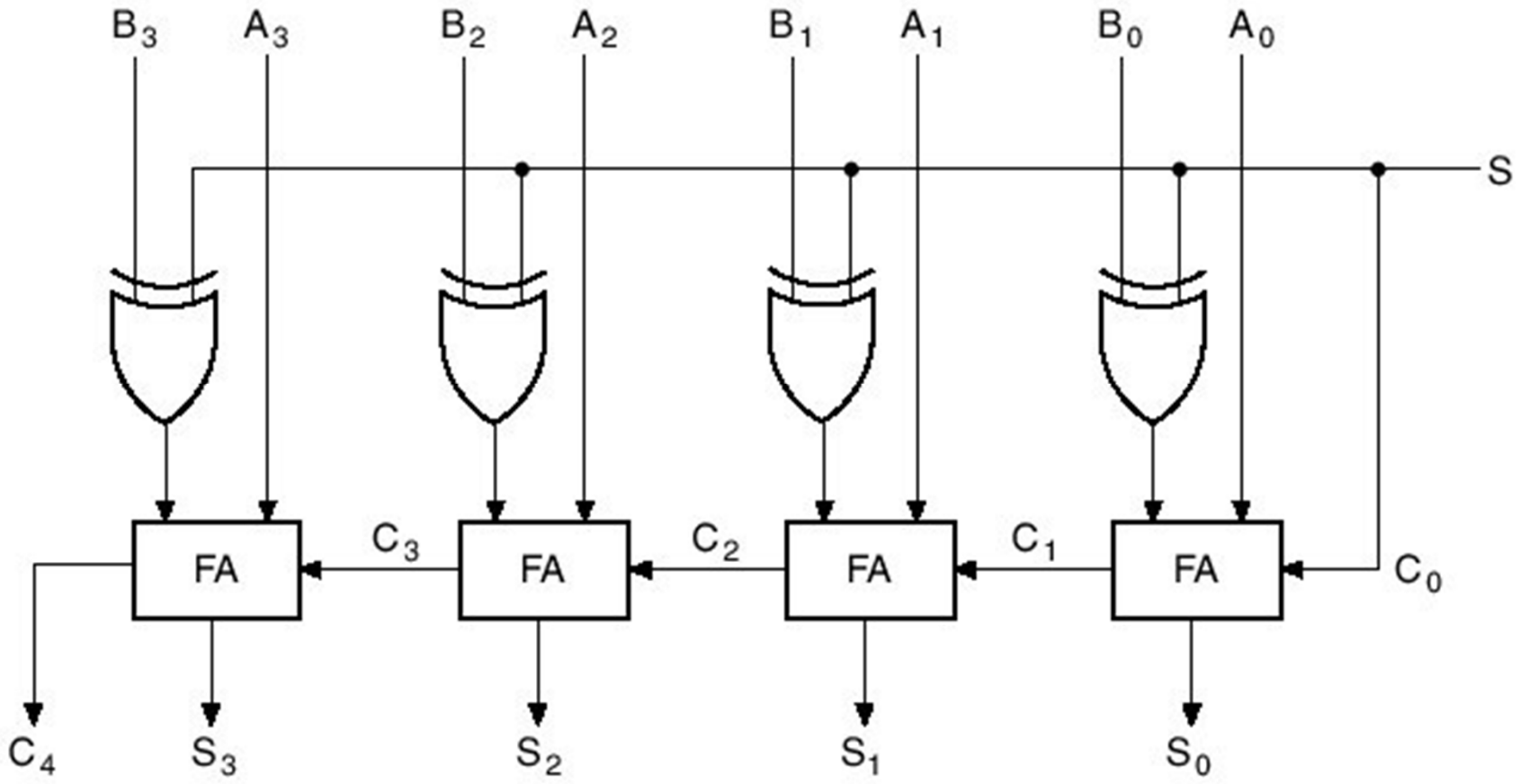
- By conditioning what arrives at A,B,C_i we can achieve:

Symbolic micro-op	CLA Inputs A		B		C	Function S
R0<-R1+R2	R1	+	R2	+	0	Addition
R0<-R1-R2	R1	+	$\overline{R2}$	+	1	Subtraction
R0<-R1+1	R1	+	0…0	+	1	Increment
R0<-R1-1	R1	+	1…1	+	0	Decrement
R0<-$\overline{R2}$	0…0	+	$\overline{R2}$	+	0	1's Complement
R0<-R2	0…0	+	$\overline{R2}$	+	1	2's Complement

CS2022 Add &Sub Implementation

- The first two of these operations may be accomplished by the addition of an XOR gate to the B-input of each full-adder, as show on the next page.

CS2022 Adder-Subtractor Circuit



CS2022 Overflow

- When we use fixed size Register for arithmetic operands there is the hazard at each micro-ops of overflow.
- If Register R stores an n-bit 2's compliment number, then:

$$-2^{n-1} \leq R \leq 2^{n-1}-1$$

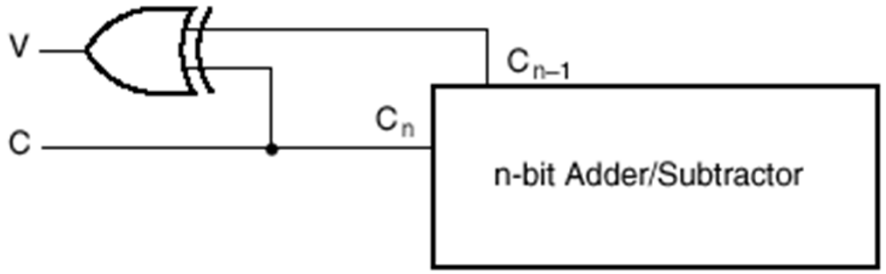
- Note the asymmetry, so even negation can cause overflow, e.g.

$$R \leftarrow -2^{n-1} \textbf{ then } R \leftarrow -R \Rightarrow \text{OVERFLOW}$$

CS2022 Carry and Overflow

- We can detect overflow for all the previous operations simply by recording the status bits C = Carry and V = Overflow (see section 3.10 I Mano and Kime).

$$K_1: C \leftarrow C_n,\ V \leftarrow C_n \oplus C_{n-1}$$

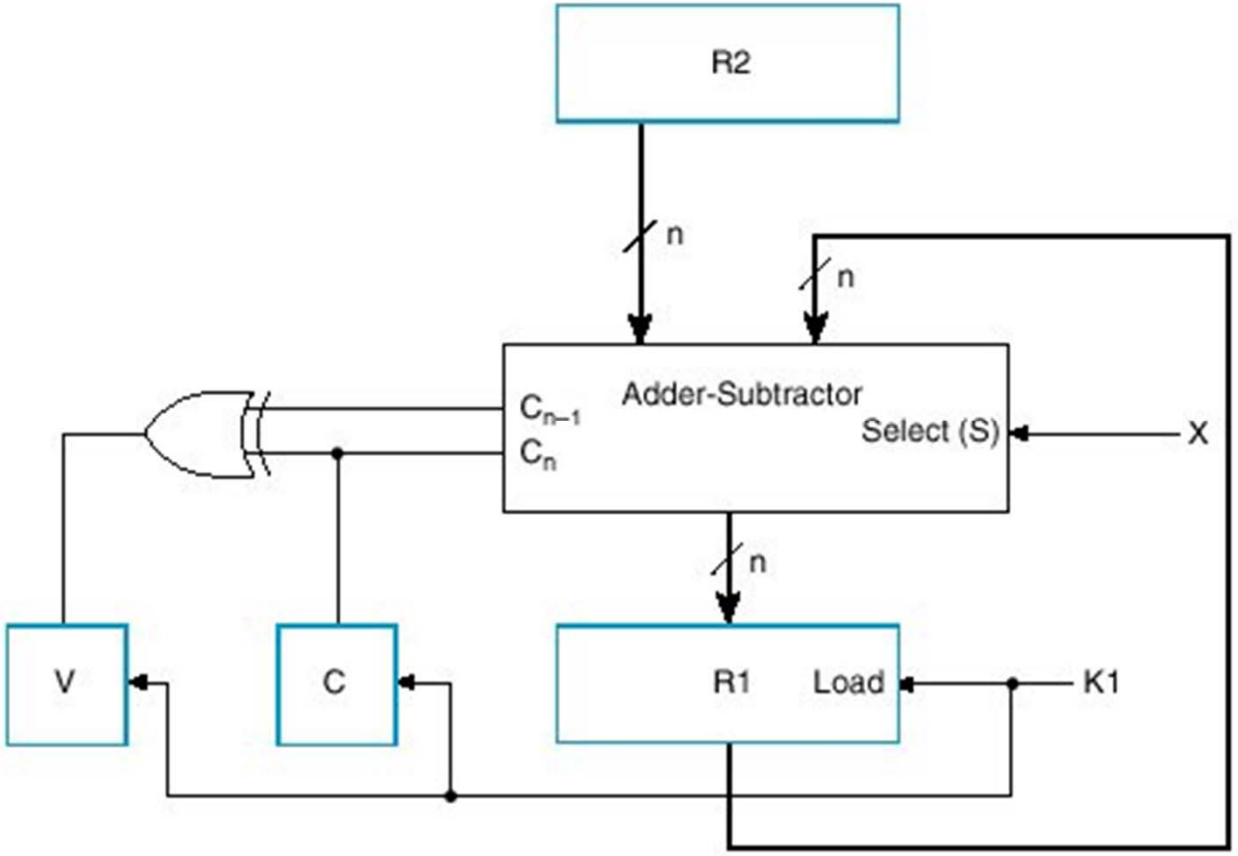


CS2022 Adder-Subtractor

- This then is the basis for the adder-subtractor on the next page which uses control input X to select addition and $\overline{X}$ for subtraction.

$$\overline{X}.K_1: R1 \leftarrow R1 + R2$$
$$X.K_1: R1 \leftarrow R1 + \overline{R2} + 1$$
$$K_1: C \leftarrow C_n,\ V \leftarrow C_n \oplus C_{n-1}$$

CS2022 Add & Subtract Micro operation



CS2022 Logic Micro-operations

- The aim here is to provide an effective set of bit-wise functions. A typical basic set follows:

Symbolic micro-op	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1's compliment)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

CS2022 ∧ And ∨

- George Bode Prof. Of mathematics in UCC introduced the notation '∧' and '∨' in 1854, and they are used in Register Transfer (RT) notation if it is necessary to distinguish addition from logical OR. For example:

$$K_1 + K_2 : R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$$

Logical OR

addition

Logical OR

- The OR micro-ops will always use ∨.

CS2022 Shift Micro-operations

- These provide lateral bitwise shift which are essential for many basic arithmetic algorithms e.g. multiplication division , square root…
- The minimal set is:

$$R \leftarrow sr\ R \equiv R_i \leftarrow R_{i+1}\ i=0, n-2, R_{n-1} \leftarrow 0$$
$$R \leftarrow sl\ R \equiv R_i \leftarrow R_{i-1}\ i=1, n-1, R_0 \leftarrow 0$$

CS2022 Logical Shifts

- These are logical shifts and from them you can develop variants which handle the end bits differently, e.g. arithmetic shift, rotates…
- Please see table 9-5 in Mano and Kime.

CS2022 Transfer Micro-operations

- Providing choice of path
- Two approaches are used:
  - Multiplexer-based transfer for speed
  - Bus-based transfers for flexibility and economy

CS2022 Transfer with Multiplexer

- The if-then else control structure, when applied to identity micro-ops, results in the destination register requiring selective access to two different source registers.

If (K_1=1) then R0←R1
else if (K_2=1) then R0←R2

K_1: R0←R1, $\overline{K_1}K_2$:R0←R2

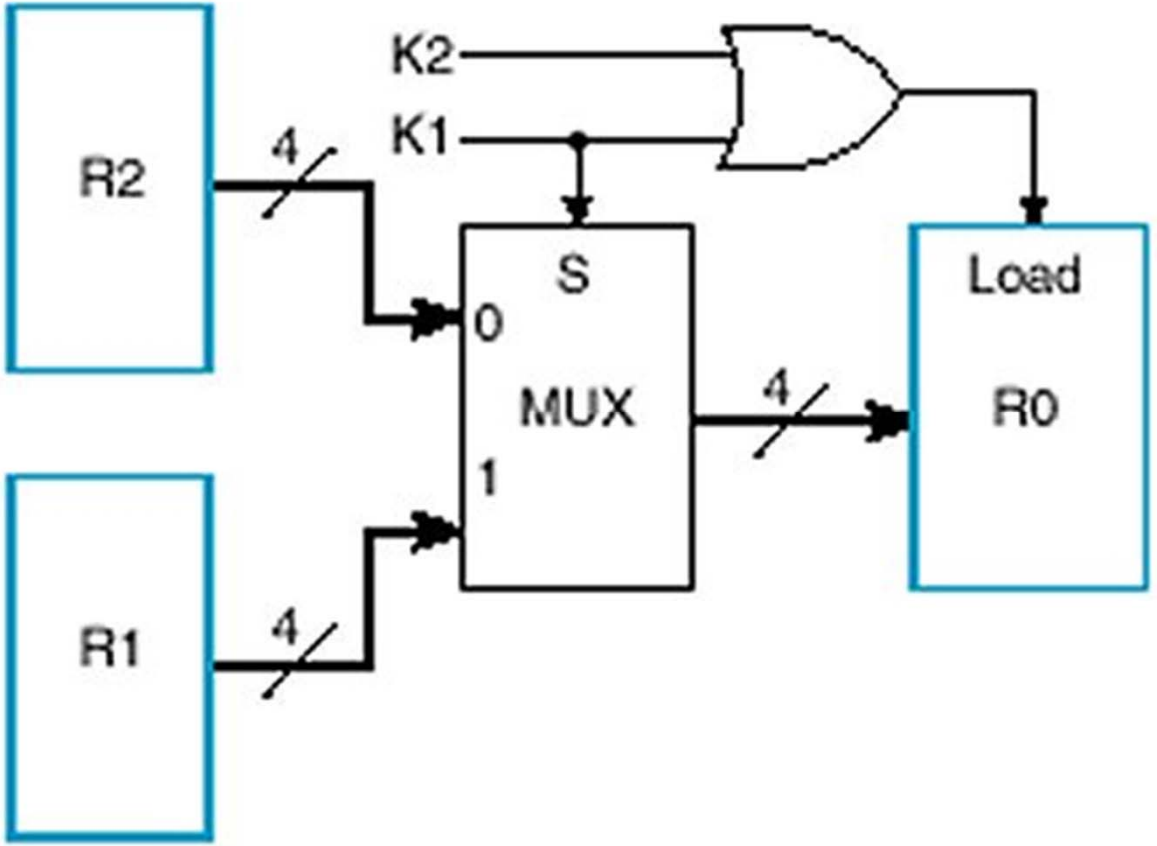
Two Sources - One Destination

- To route two sources to one destination we can use a 2:1 MUX with control input S, data input D0 $\wedge$ D1, and R0 must have a load control $LOAD_{R0}$.
- From the RT description we deduce the following functions for these control inputs.

$$LOAD_{R0} = K_1 + \overline{K}_1 K_2 = K_1 + K_2$$
$$S = K_1 \Rightarrow D1 = R1 \wedge D0 = R2$$

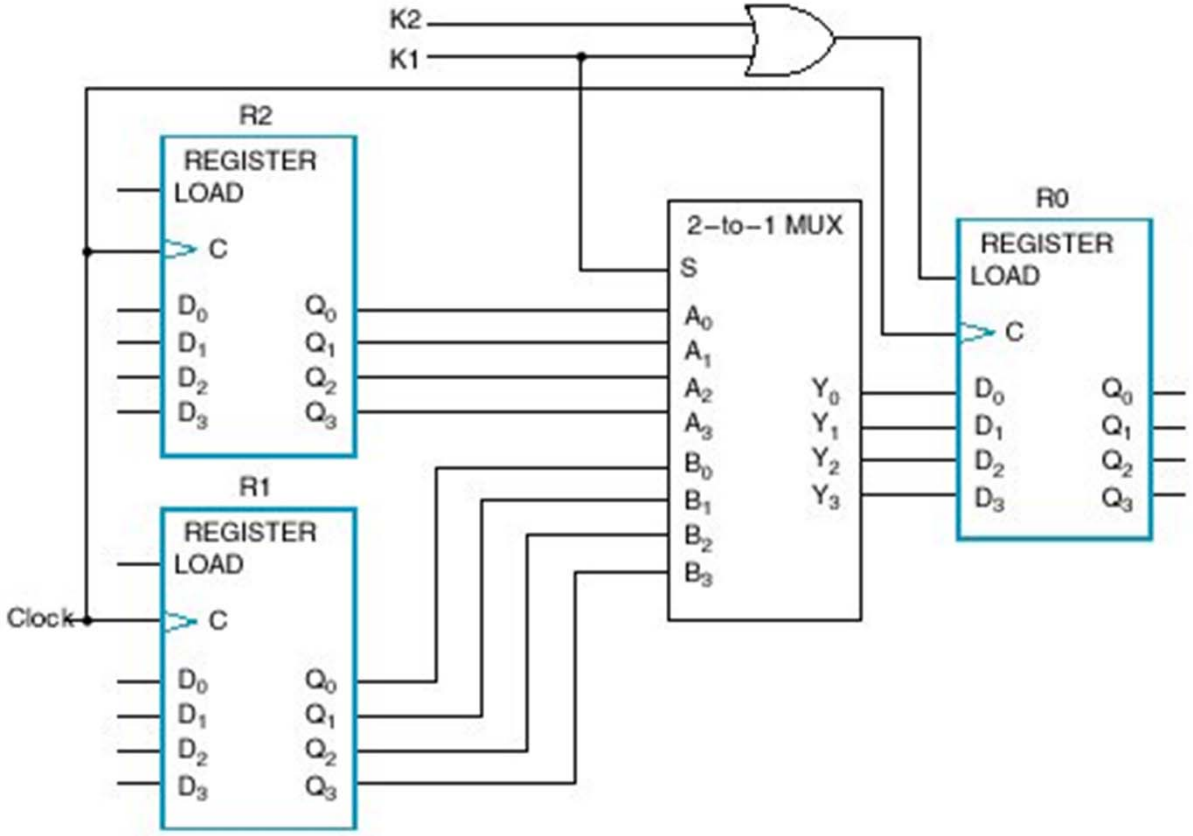
Multiplexer

- From this we derive the circuit schematic, assuming 4-bit Registers.



CS2022 Multiplexer Detail

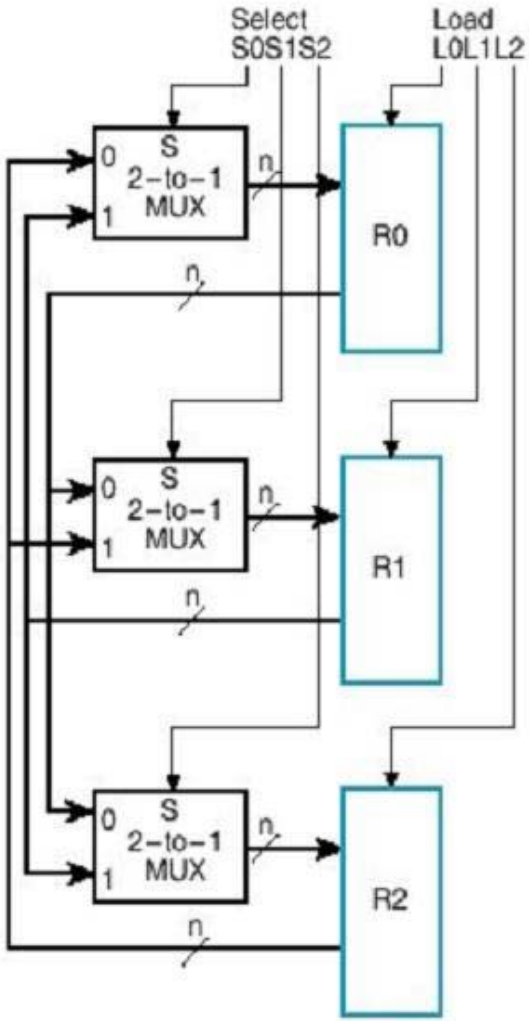
- Note changes in K1 only affect the MUX – R0 path, so transients are short, allowing fast operation.



CS2022 Bus-based Transfers

- Digital systems typically have a considerable number of registers N.
  - Typically $8 \leq N \leq 256$
  - Programmer need to be able to make transfers between any pair of them.
  - Let us consider the N=3 and use 2:1 MUXs to interconnect R0, R1, R2.
    - The result is on the next slide.

CS2022 Register Transfer via MUX Diagram



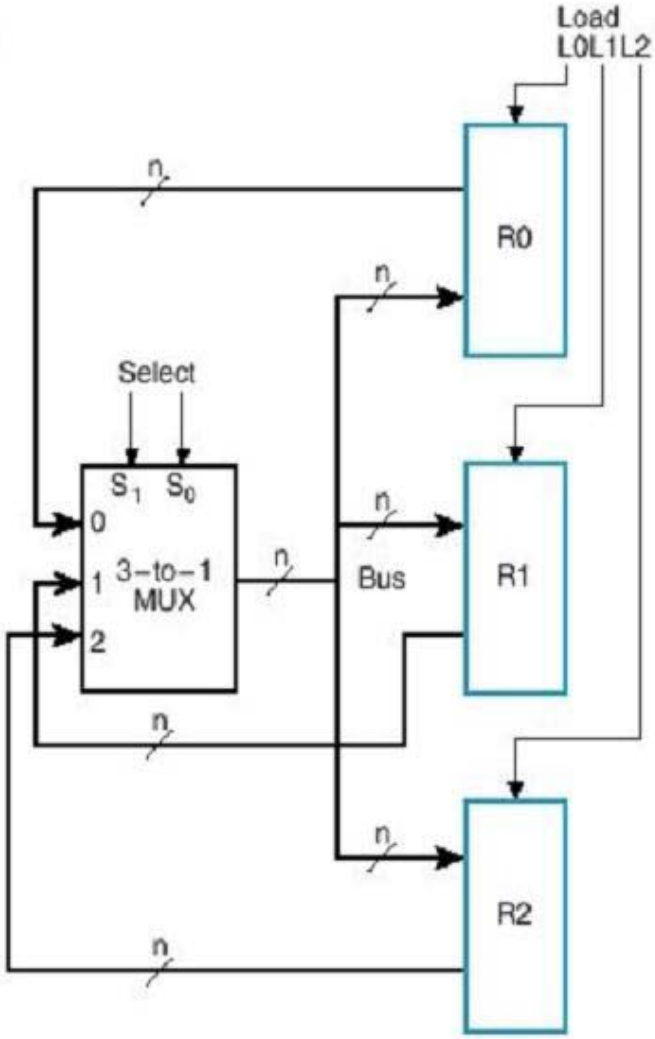
CS2022 Register Transfer via MUXs

- This is a very flexible system for it can make up to three independent transfers in one clock period.

RT	S2 S1 S0	L2 L1 L0	Description
R2 ← R1	1 X X	1 0 0	Point-to-Point
R2 ← R1, R1 ← R2	1 1 X	1 1 0	Reg. Exchange
R2 ← R1, R1 ← R0 R0 ← R2	1 0 0	1 1 1	Reg. Rotate
R2 ← R0, , R1 ← R0	0 0 X	1 1 0	Reg. broadcast

- But this is very costly in terms of interconnect, requiring 6*n MUX input connections.
- To connect N*n-bit registers will require (N-1)*N*n wires.

CS2022 Register Transfer via MUX & Bus



- To reduce the amount of interconnects we can use 3-1 MUXs with a single MUX-to-Register bus connection.

CS2022 Register Transfer via 3-1 MUXs and MUX–to-Register Bus

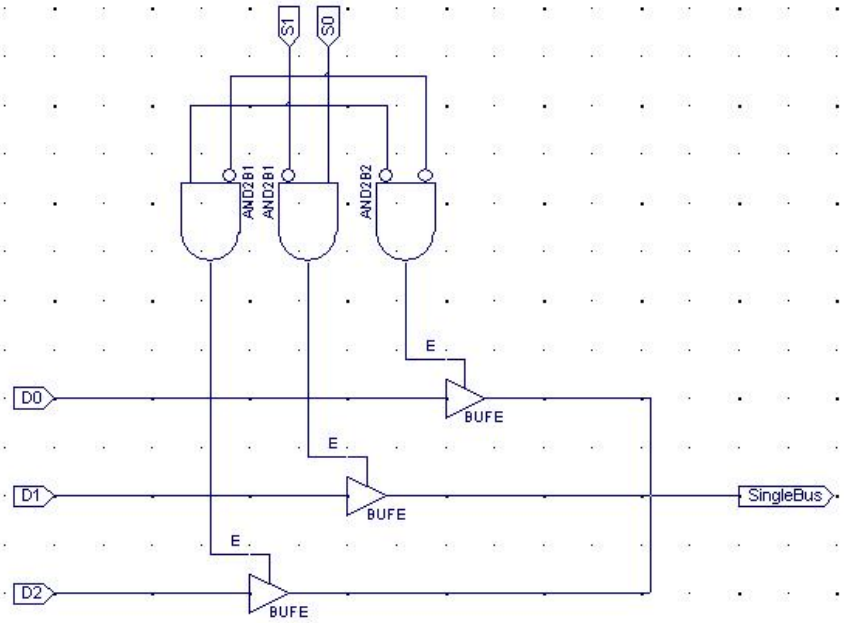
- This results in the loss of some flexibility:

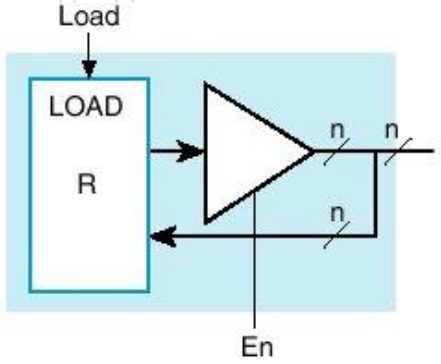
RT	S1 S0	L2 L1 L0	Description
R0 ← R2	1 0	0 0 1	Point-to-Point
R0 ← R1, R2 ← R1	0 1	1 0 1	Reg. Broadcast
R0 ← R1, R1 ← R0	IMPOSSIBLE		Single source only

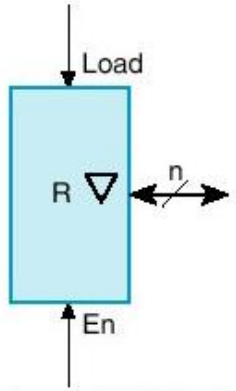
- But the MUX input connections have reduced from 6*n to 3*n.
- For N Register we need only N*n wires.

CS2022 Tri-state Bus

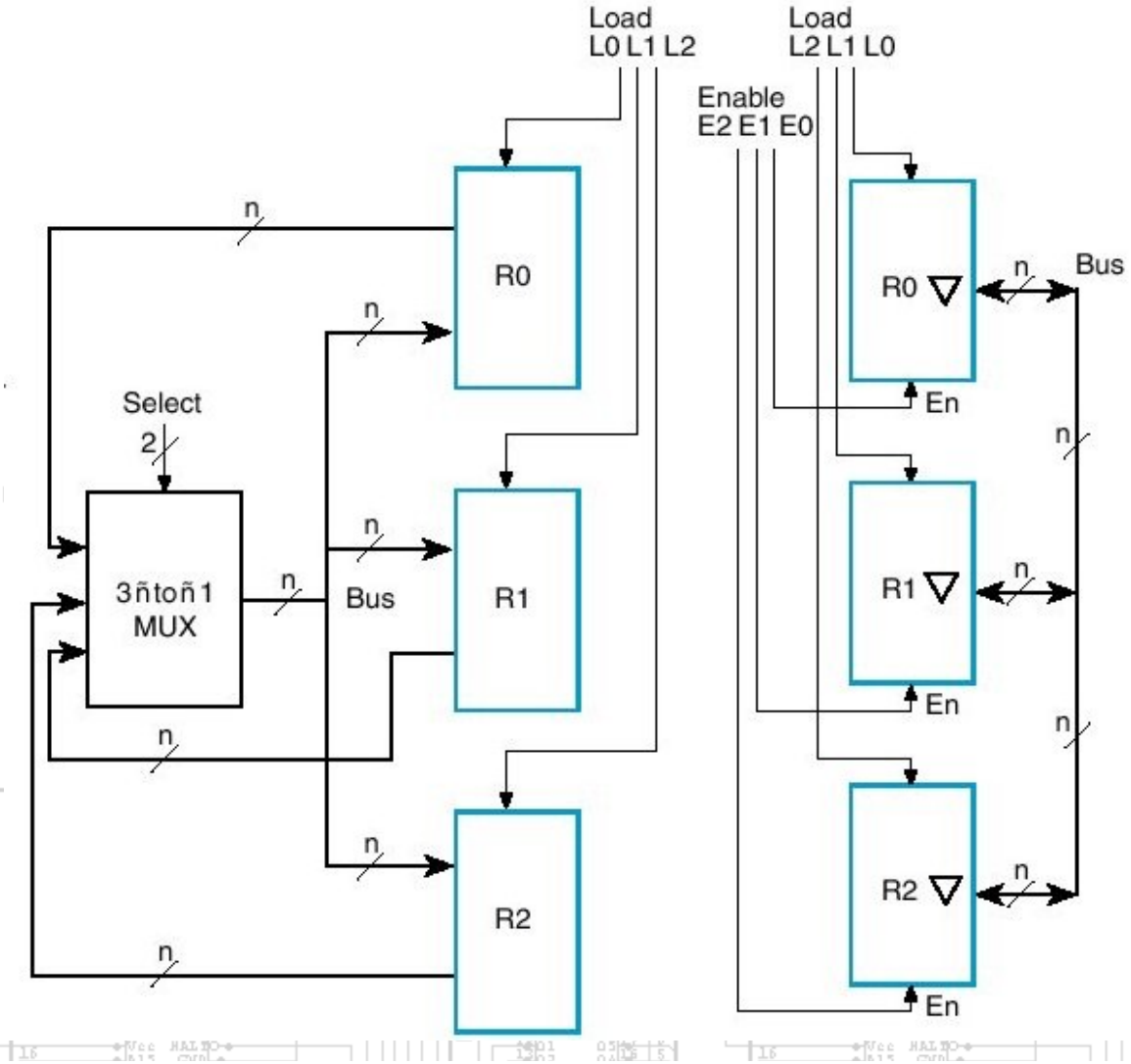
- Tri-state buffers provide the means to construct a wired-or of arbitrary fan-in, with which we can effectively disperse the MUX on the previous slide right back to the register latches. That is we build our 3:1 MUX as:







CS2022 Tri-state-bus vs. MUX-bus



- This results in a solution seen on the right (tri-state), which has the same functionality as the MUX based solution on the left, using just a single bi-directional bus.

CS2022 Bi-directional Tri-state Bus

RT	E2 E1 E0	L2 L1 L0	Description
R0 ← R2	1 0 0	0 0 1	Point-to-Point
R0 ← R1, R2 ← R1	0 1 0	1 0 1	Reg. Broadcast
R0 ← R1, R1 ← R0	IMPOSSIBLE		Single source only

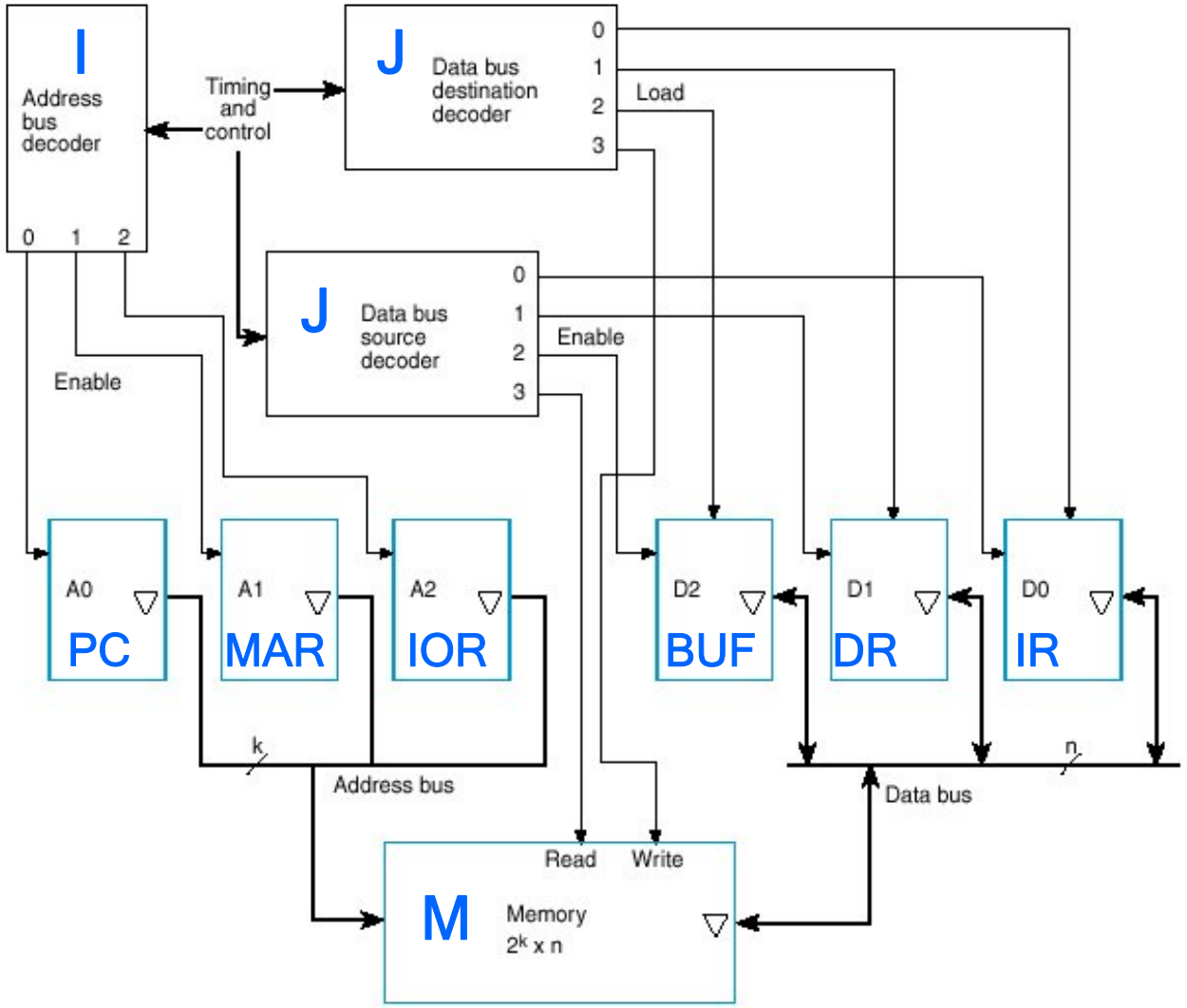
- With this arrangement it is possible to connect N*n-bit register with N-1 paths of width n.

CS2022 Memory Transfers

- Typically processor memories are addressed by a number of address registers:
  - **PC**, **MAR**, **IOR**
- Address register information is transmitted over an address bus to memory **M**.
- Similarly data is transferred to/from a number of data registers over a data bus.
  - **Data register: IR**, **DR**, **BUF**
- The schematic on the next slide illustrates how we may organise three address register **A**, selected by I, and data register D, selected by **J**, to implement read/write on memory **M**.

CS2022

READ: D[J] ← M[A[I]]
WRITE: M[A[I]] ← D[J]

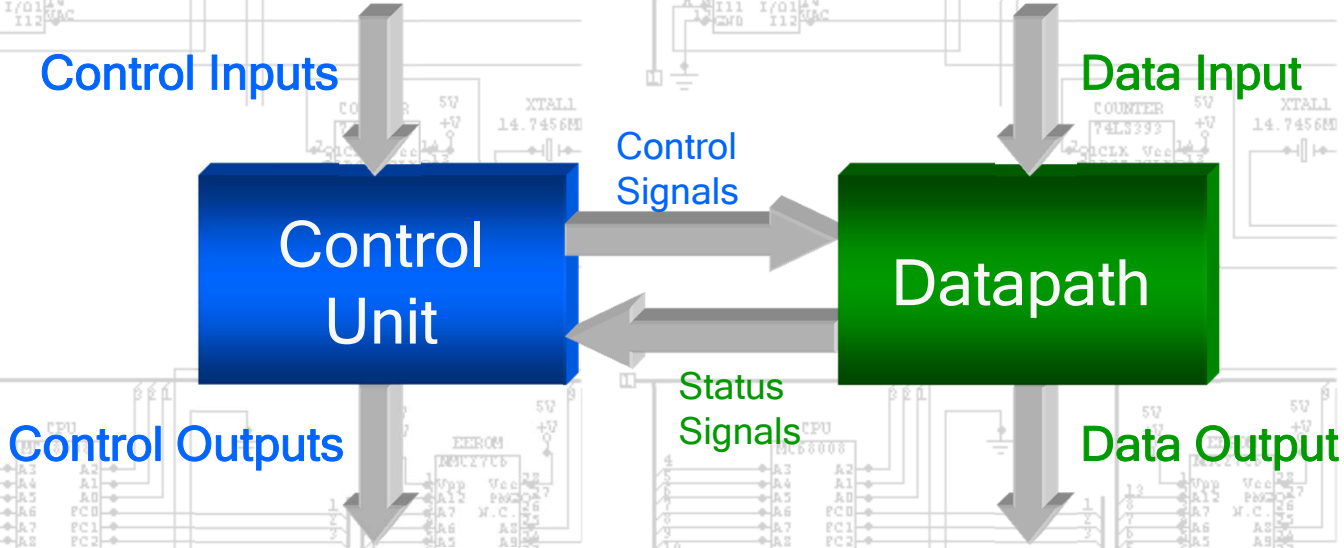


CS2022 Datapath

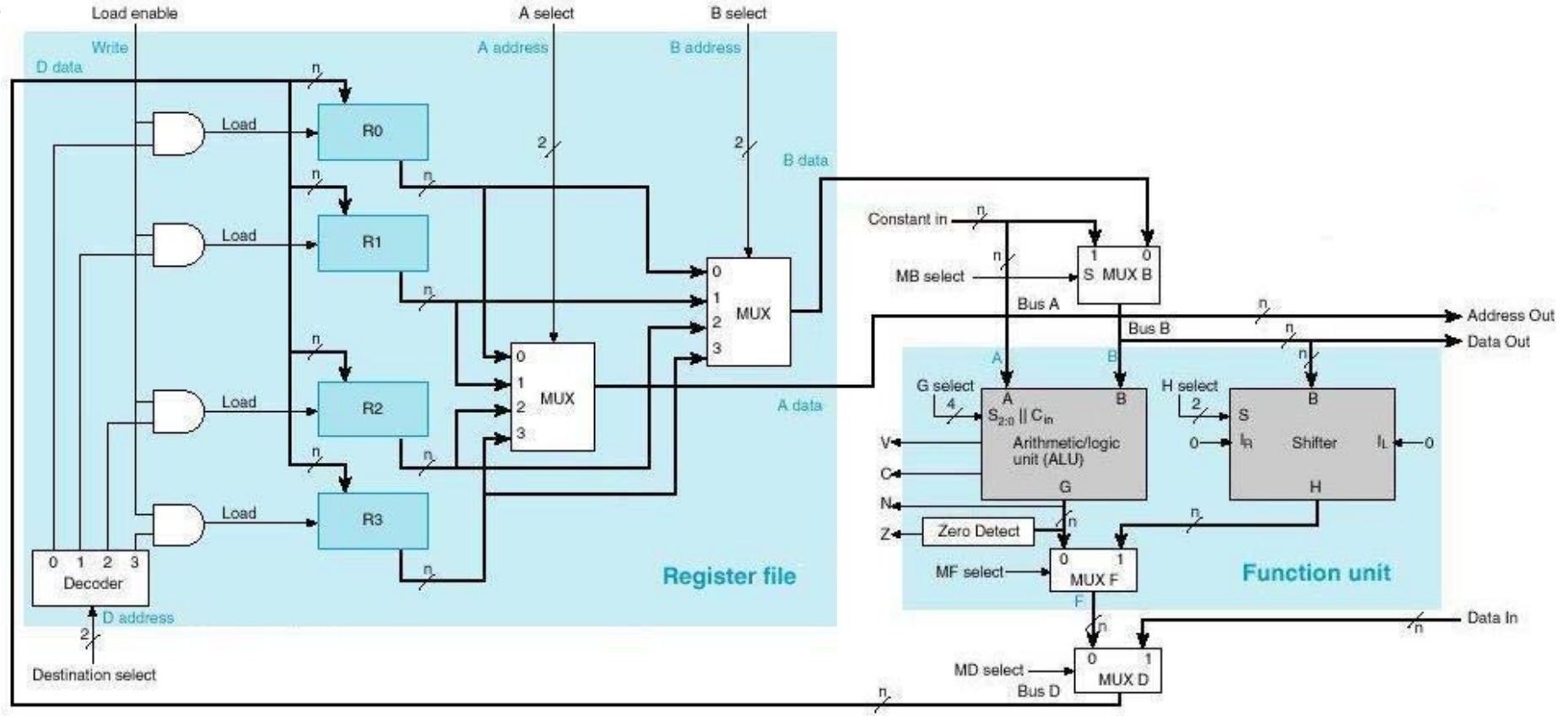
- In all digital systems we can partition the structure into two sections:

a) The data path which performs data-processing operations on the data stream.

b) The control unit which determines the schedule for these data processing operations.



CS2022 Datapath Schematic



CS2022 Register File & Functional Unit

- In practice the three functional micro-ops are implemented in one compact circuit, the Function Unit, composed of the ALU and Shifter.
- Data for this Functional Unit comes from a physical adjacent Register file, with dual MUX-busses able to supply two operands per clock cycle.
- Since the register file is itself modest in size typically 8-32 register, there must be provision to send and receive data to/from the main memory system via DATA IN and DATA OUT.

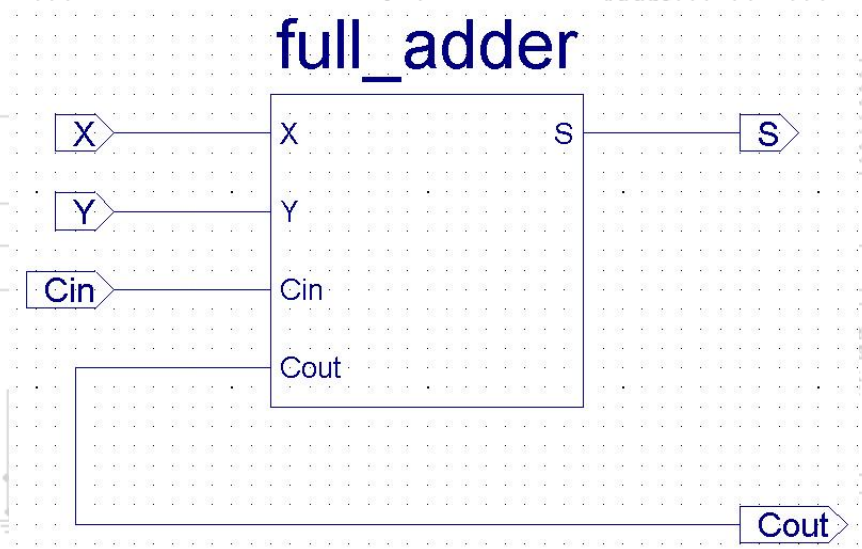
CS2022 ALU Design

- The fundamental operation of the arithmetic is addition.
- All others:
  - Subtraction
  - Multiplication
  - Division
- are implemented in terms of it.
  - We need therefore an efficient implementation.

N-bit Ripple-Carry-Adder (RCA)

n Full Adders

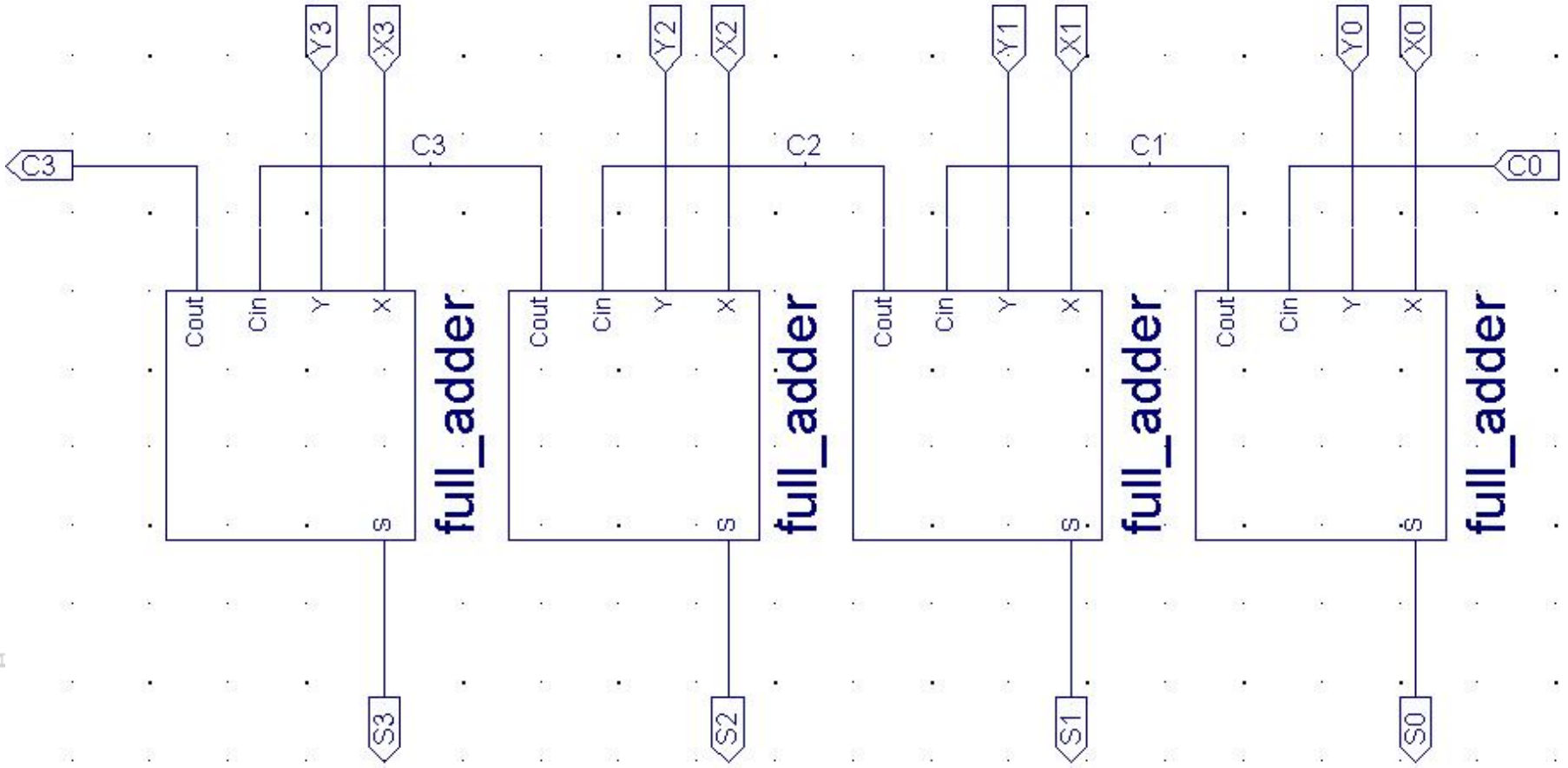
- An n-bit ripple-carry-adder is constructed from n full-adders



$$S = x \oplus y \oplus C_{in}$$

$$C_{out} = XY + xC_{in} + yC_{in}$$

CS2022 4-bit Ripple Carry Adder



RCA Equations

$$S_i = x_i \oplus y_i \oplus C_i$$
$$C_{i+1} = X_i Y_i + xC_i + yC_i$$

- Hence, using an AND+wired-OR and n-bit RCA intruduces n gate delays.
- For 64 bit calculations this is too slow
  - (64 gate delays)

Carry Lookahead Boolean Expression

$$C_{i+1} = x_i y_i + C_i(x_i + y_i)$$

$$C_1 = x_0 y_0 + C_0(x_0 + y_0)$$

$$C_2 = x_1 y_1 + C_1(x_1 + y_1)$$

$$= x_1 y_1 + [x_0 y_0 + C_0(x_0 + y_0)](x_1 + y_1)$$

with $g_i = x_i y_i$ **Generate Carry**

$p_i = x_i + y_i$ **Carry Propagate**

$$C_{i+1} = g_i + p_i C_i$$

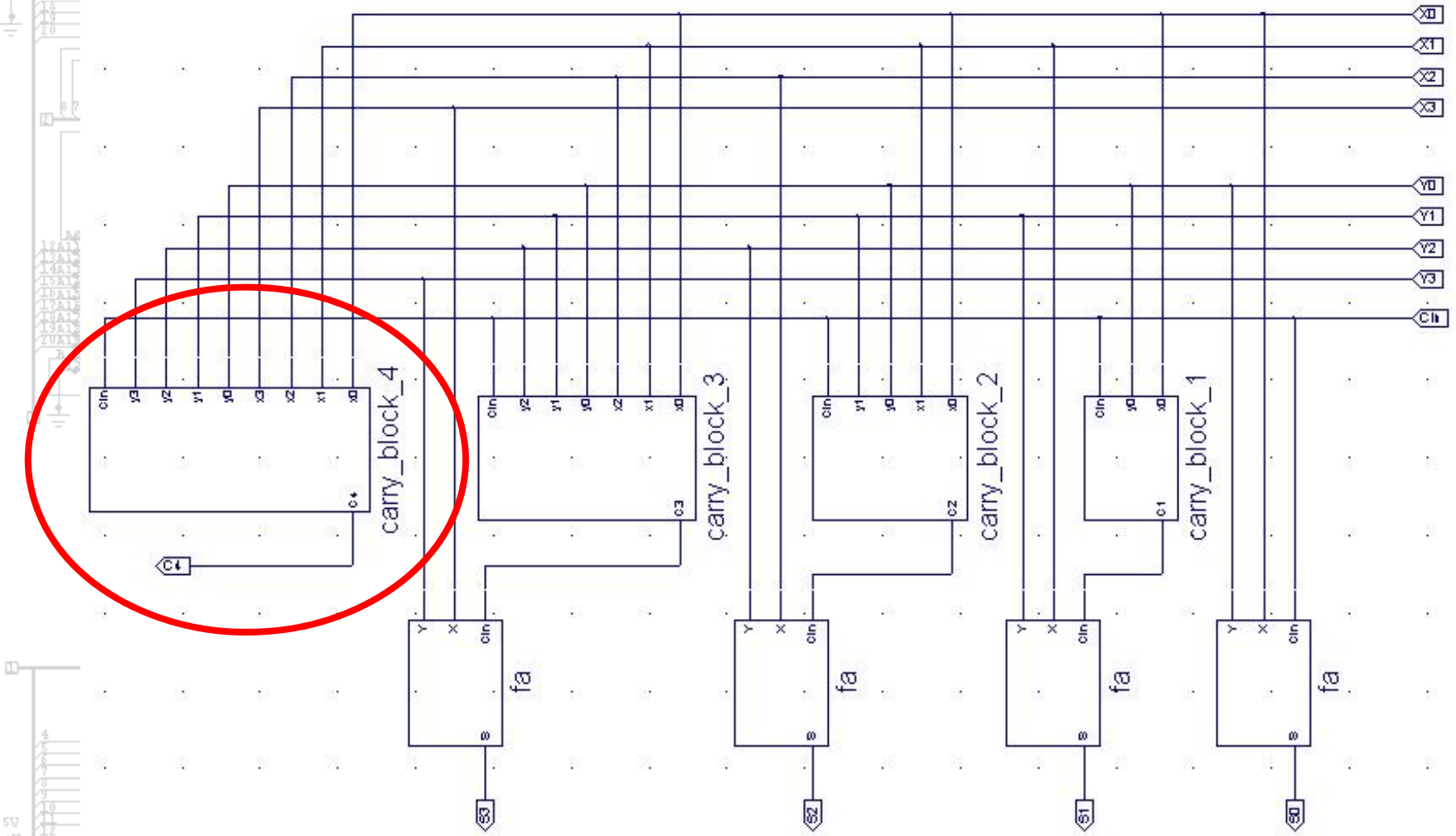
Carry Lookahead

Boolean Expression

$$C_{i+1} = g_i + p_i C_i$$

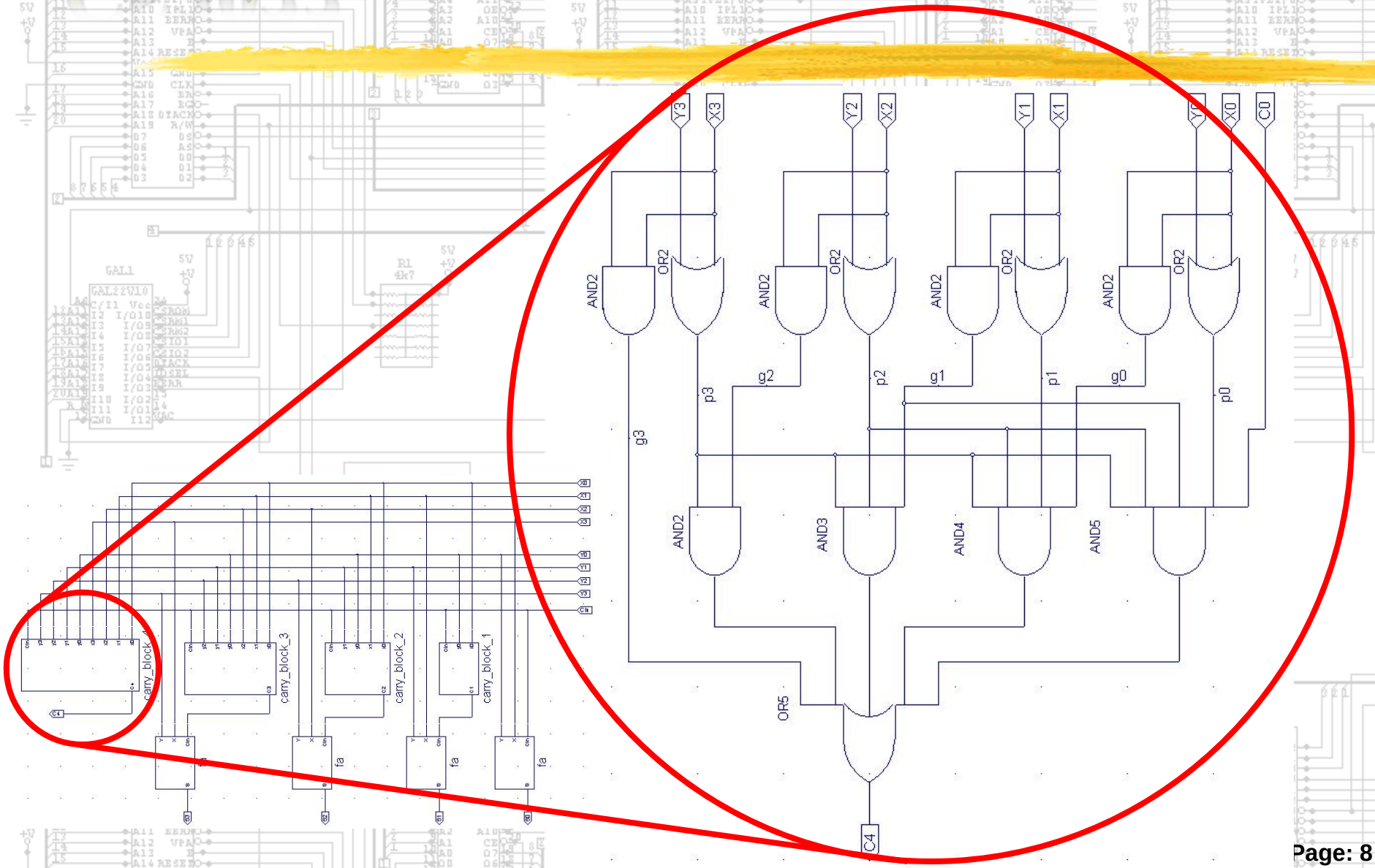
$$\begin{aligned}
C_1 &= x_0 y_0 + C_0(x_0 + y_0) \\
&= g_0 + C_0 p_0 \\
C_2 &= x_1 y_1 + C_1(x_1 + y_1) \\
&= x_1 y_1 + [x_0 y_0 + C_0(x_0 + y_0)](x_1 + y_1) \\
&= g_1 + p_1 g_0 + p_0 p_1 C_0 \\
C_3 &= g_2 + p_2 g_1 + p_1 p_2 g_0 + p_0 p_1 p_2 C_0 \\
C_4 &= g_3 + p_3 g_2 + p_2 p_3 g_1 + p_1 p_2 p_3 g_0 + p_0 p_1 p_2 p_3 C_0
\end{aligned}$$

CS2022 4-bit Carry Lookahead Adder



C_4 Carry Block

$$C_4 = g_3 + p_3g_2 + p_2p_3g_1 + p_1p_2p_3g_0 + p_0p_1p_2p_3C_0$$



CS2022 Carry Lookahead Adder

$$C_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_0 C_0$$

- This requires just two gate delays:
  - One to generate g_i and p_i
  - Another to AND them
- Again we can use wired OR
- But, it requires AND gates with a fan in of n
- In practice we can only efficiently build single gates with a limited fan-in
- we build the lookahead circuit as a multi-level circuit

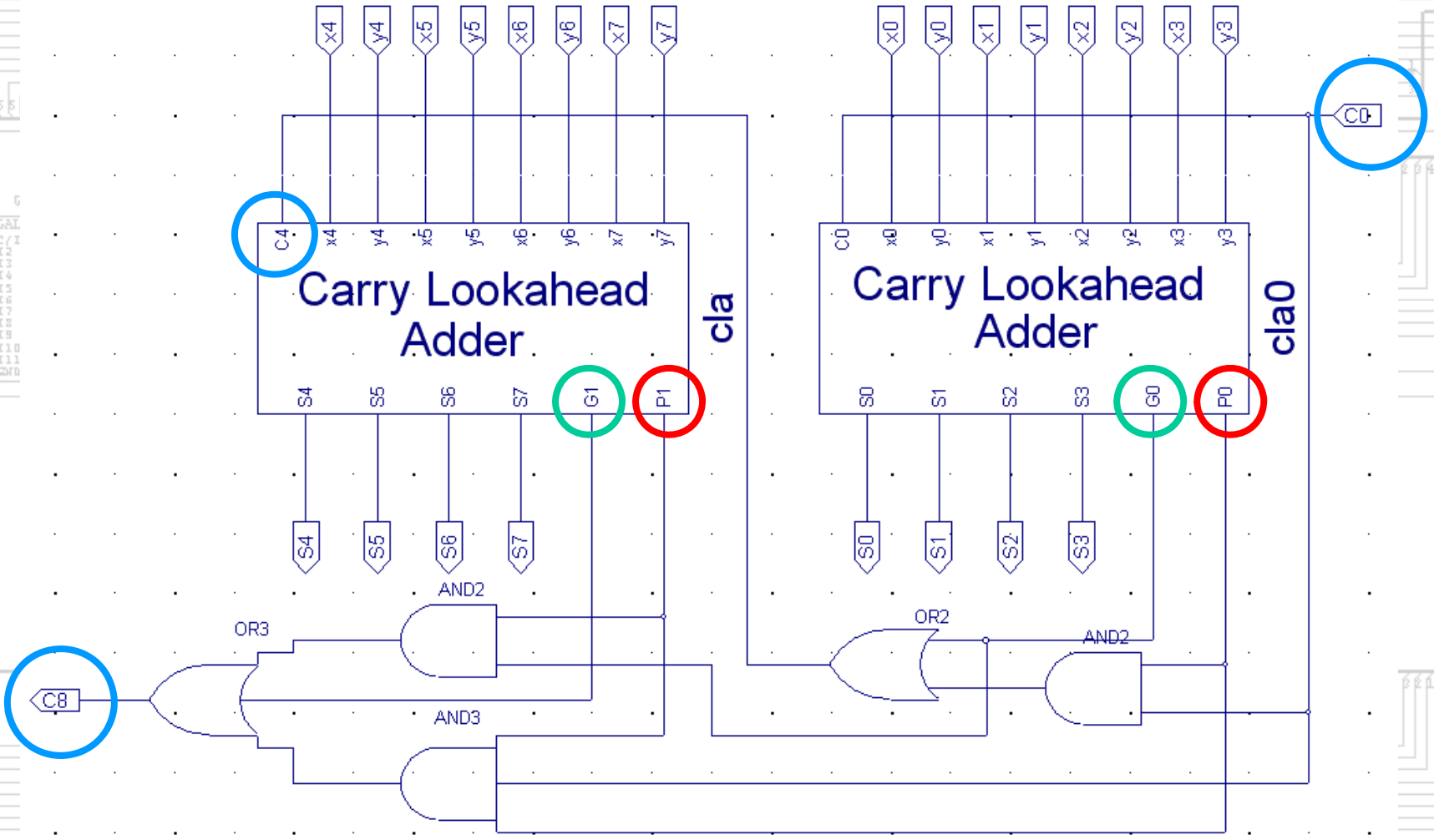
CS2022 Groups of Input Bits

- For example, let fan-in = 4 and define:
  - G_i' A carry out is generated in the i^{th} group of four input bits
  - P_i' A carry out is propagated by the i^{th} group of four input bits

$$G_0' = g_3 + p_3g_2 + p_2p_3g_1 + p_1p_2p_3g_0$$
$$P_0' = p_0p_1p_2p_3$$
$$C_4 = G_0' + C_0P_0'$$
$$C_8 = G_1' + P_1'G_0' + P_0'P_1'C_0$$
$$C_{12} = G_2' + P_2'G_1' + P_1'P_2'G_0' + P_0'P_1'P_2'C_0$$

$C_4 = G_0' + C_0P_0'$

$C_8 = G_1' + P_1'G_0' + P_0'P_1'C_0$



Generate G'' and Propagate P''

- The next level of generate G'' and propagate P'' terms will cover 16 bits

$$G'' = G_3' + P_3'G_2' + P_3'P_2'G_1' + P_3'P_2'P_1'G_0$$
$$P'' = P_3'P_2'P_1'P_0$$

CS2022 64-bit Adder Propagation Delay

- We can implement a 64-bit adder using AND-or logic with a fan-in = 4 and a maximum propergation delay of:

$$t_{pmax} = 3(G_1') + 2(G_1'') + 2(C_{48}) + 2(C_{60}) + 3(S_{63})$$
$$= 12 \text{ gate delays}$$

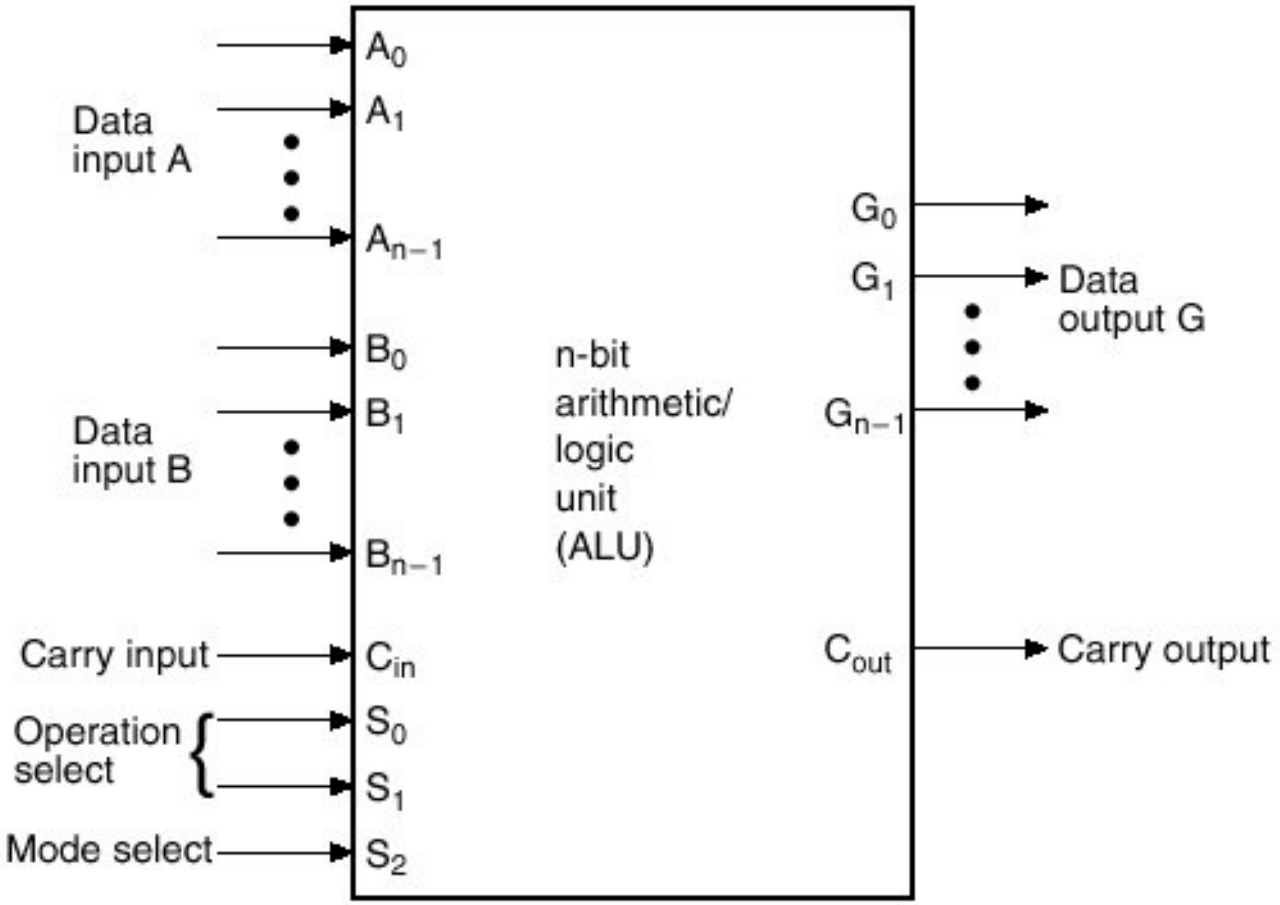
- Compare this with RCA using AND-wiredOR which requires 64 gate delays.
- If we add a third layer (**G'''**, **P'''**) we can construct a 4x64 = 256 bit adder with maximum delay:

$$t_{pmax} = 3 + 2 + 2 + 2 + 2 + 2 + 3$$
$$= 16 \text{ gate delays}$$

CS2022 Arithmetic Circuit

- The arithmetic circuit may implemented with the following components:
  - Parallel Adder
    - Build from a cascade of full-adder circuits
  - The data input to the parallel adder is manipulated in order to achieve a number of arithmetic operations

CS2022 N-bit Arithmetic/Logic Unit(ALU)

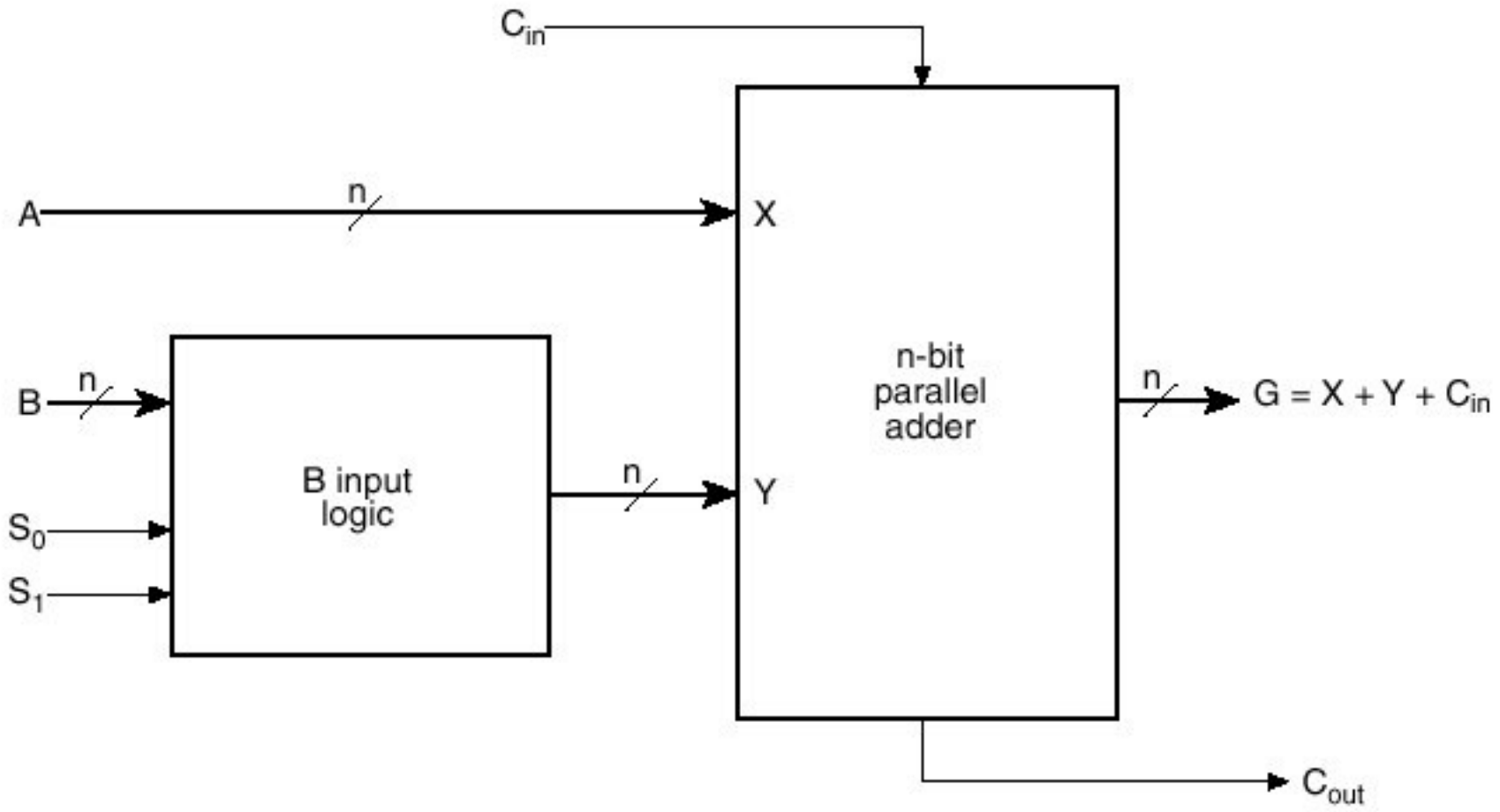


$G = A + Y + C_{in}$

- The arithmetic micro-ops can be implemented using the carry-in C_{in} and two select inputs S_1 & S_0, which condition the B input to deliver **Y** to the full-adder computing: $G = A + Y + C_{in}$.

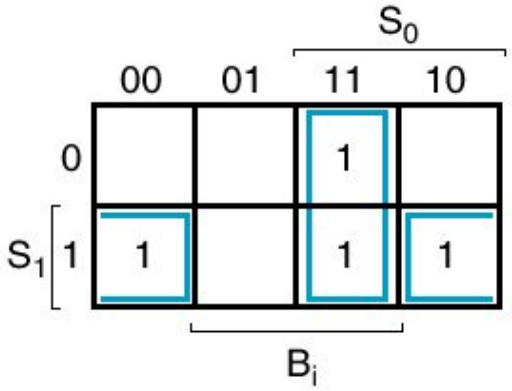
Select		Input		
S_1	S_0	Y	$C_{in}=0$	$C_{in}=1$
0	0	all 0's	G=A	G=A+1
0	1	$\overline{B}$	$G=A+\overline{B}$	$G=A+\overline{B}+1$
1	0	B	G=A+B	G=A+B+1
1	1	all 1's	G=A-1	G=A

CS2022 Arithmetic Circuit



CS2022 Y(S,B)

- The logic function **Y(S,B)** is derived as:

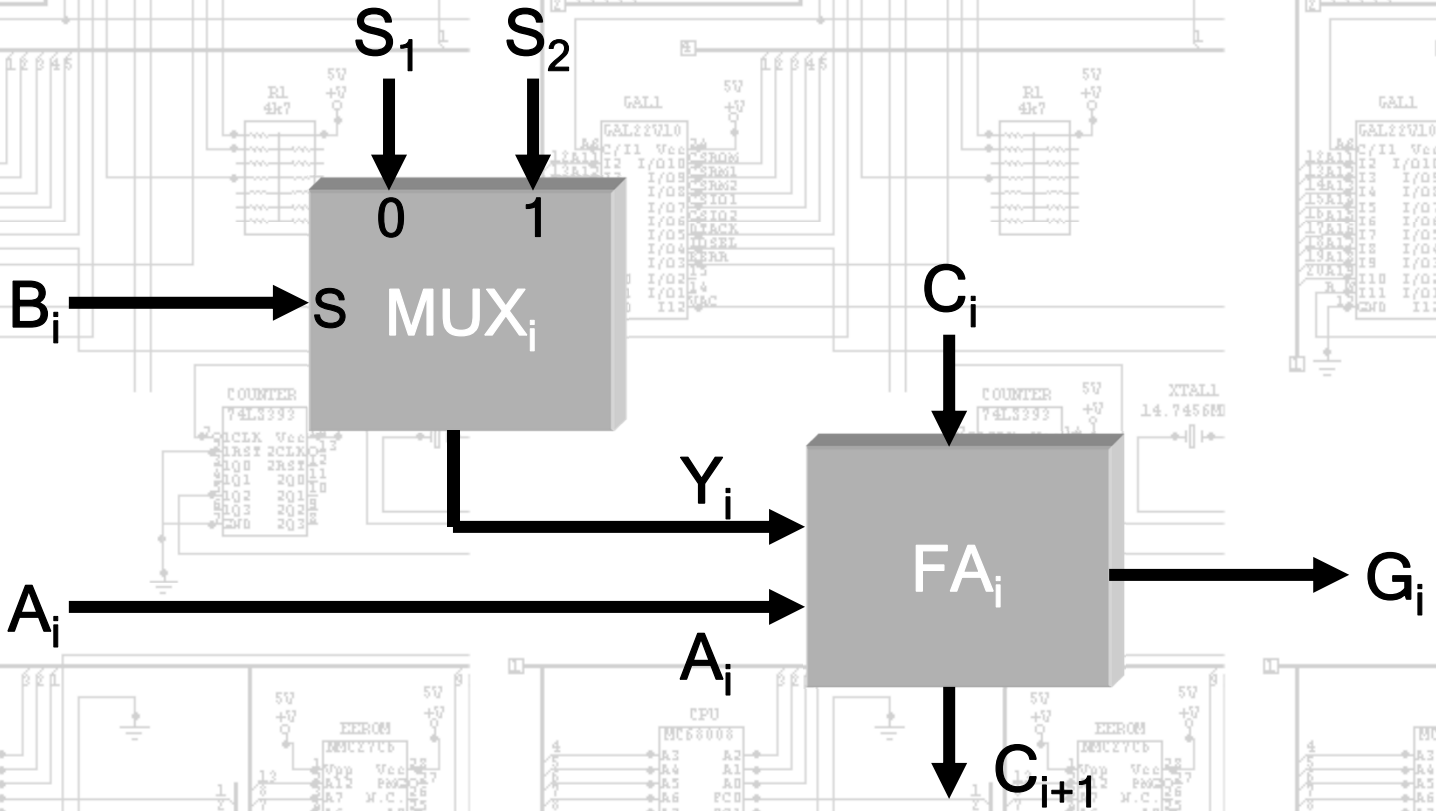


$$Y_i = S_0 B_i + S_1 \overline{B_i}$$

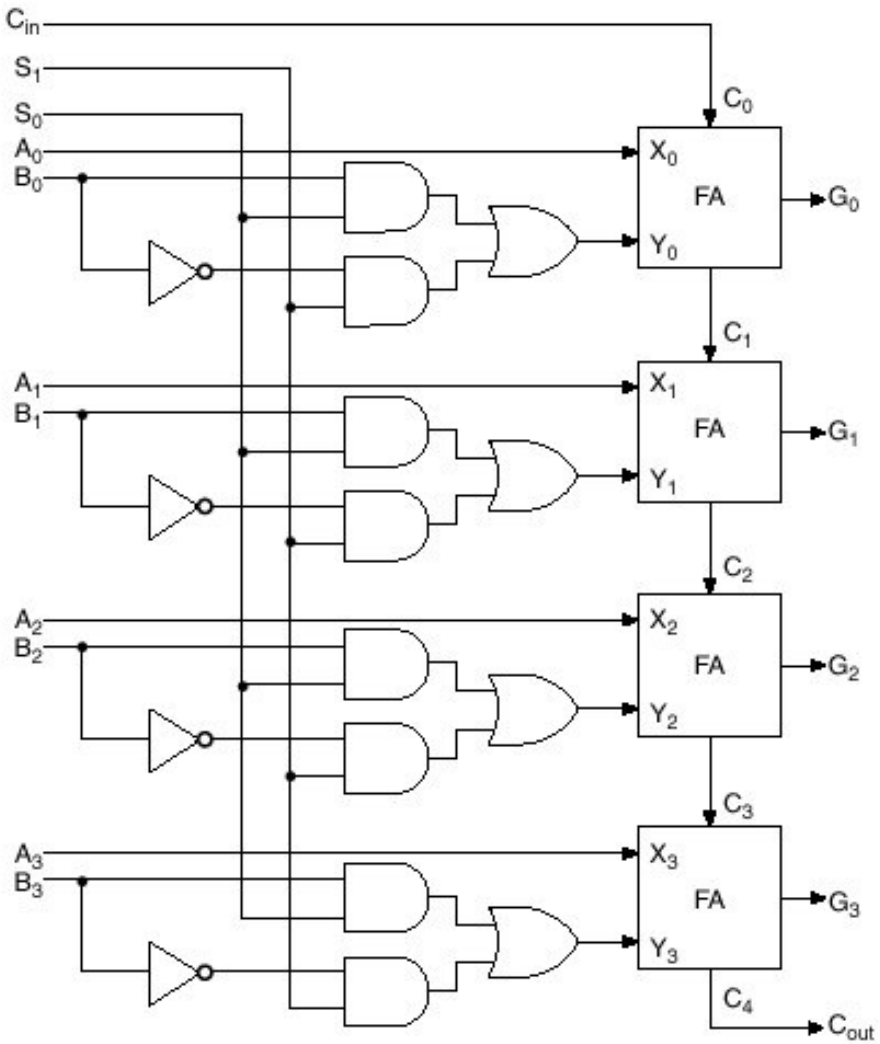
- Thus a 2:1 MUX controlled by B_i can efficiently generates Y_i

CS2022 One Bit Slice

- One bit slice of the Arithmetic unit on the next slide.



CS2022 4-Bit Arithmetic Circuit

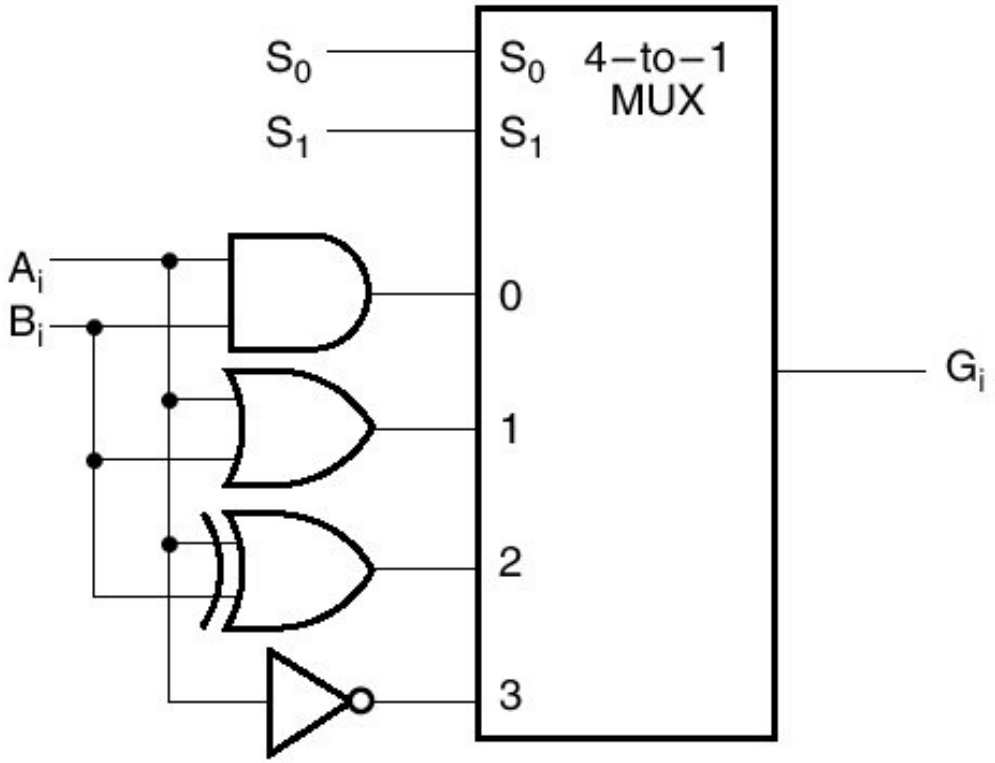


CS2022 Logic Circuit

- The logic function are similarly selected by input S_1 and S_0 :

Select		Output	
S_1	S_0		
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

CS2022 Logic Circuit Implemented with a 4:1 MUX

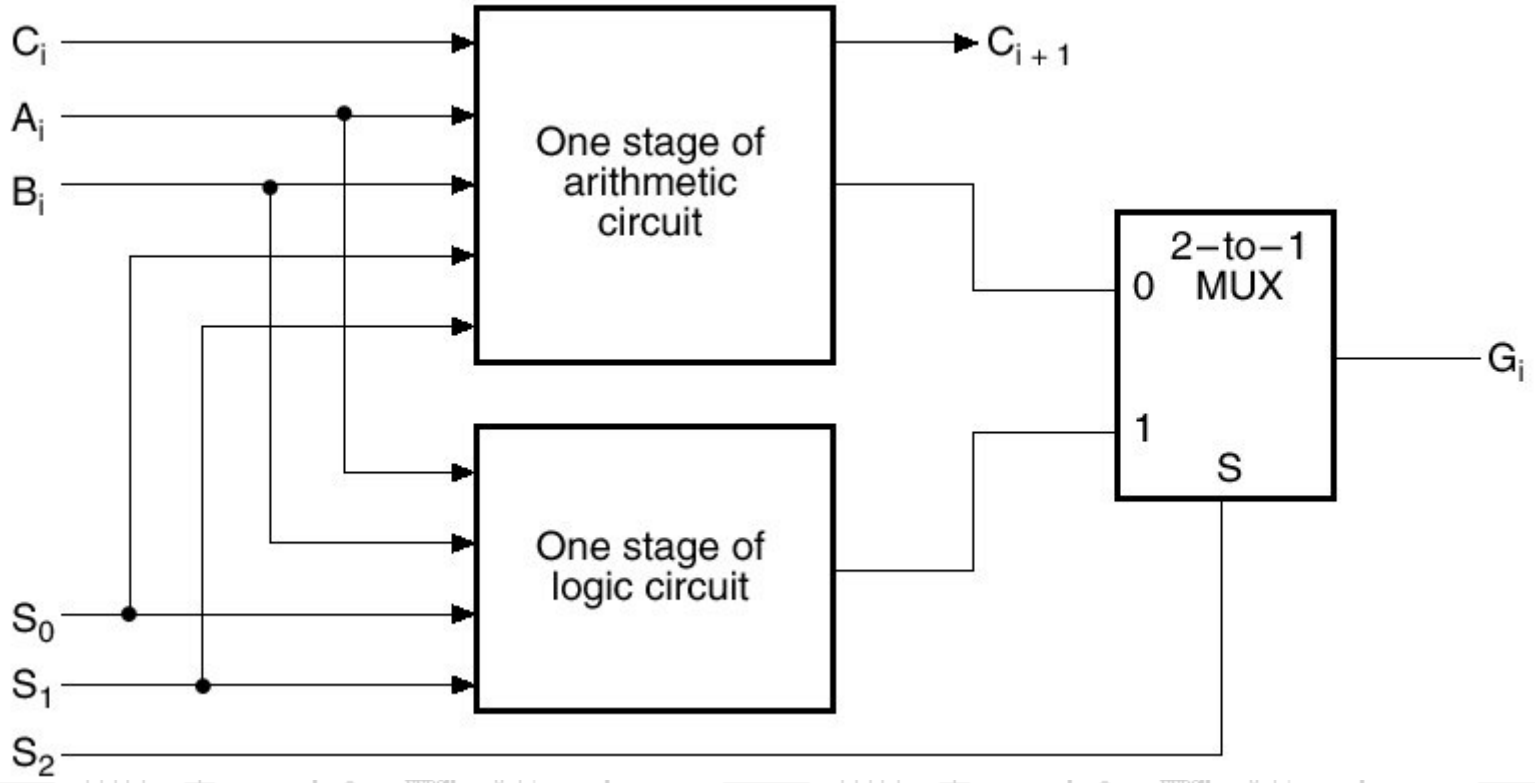


- One bit slice of the logic unit.

CS2022 ALU (Arithmetic/Logic)

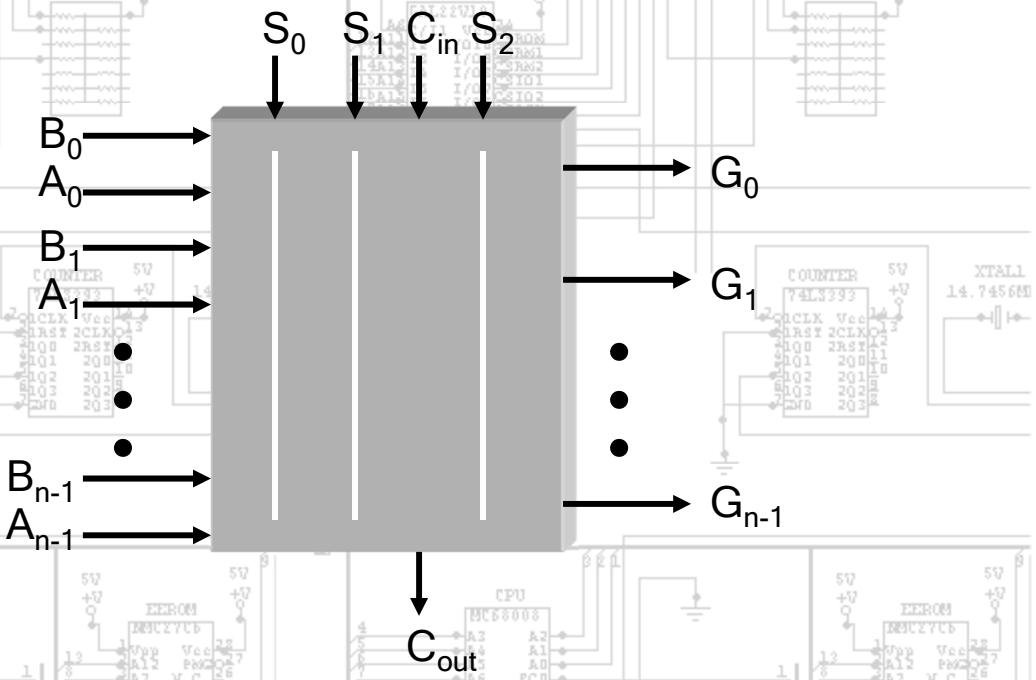
- We next use an additional 2:1 MUX controlled by S_2 to select either the arithmetic output bit or the logic output bit as shown on the next slide.

CS2022 One bit slice ALU



CS2022 N-bit ALU

- To construct an n-bit ALU we concatenate n-bit slices together:



CS2022 Physical Implementation

- Physical schematic of an n-bit ALU assembled from a bit slices as shown on the previous slide.

1. Note the control signals, because they apply to the whole word, tend to cross the datapath.
2. This geometry results in very efficient VLSI chip implementation.

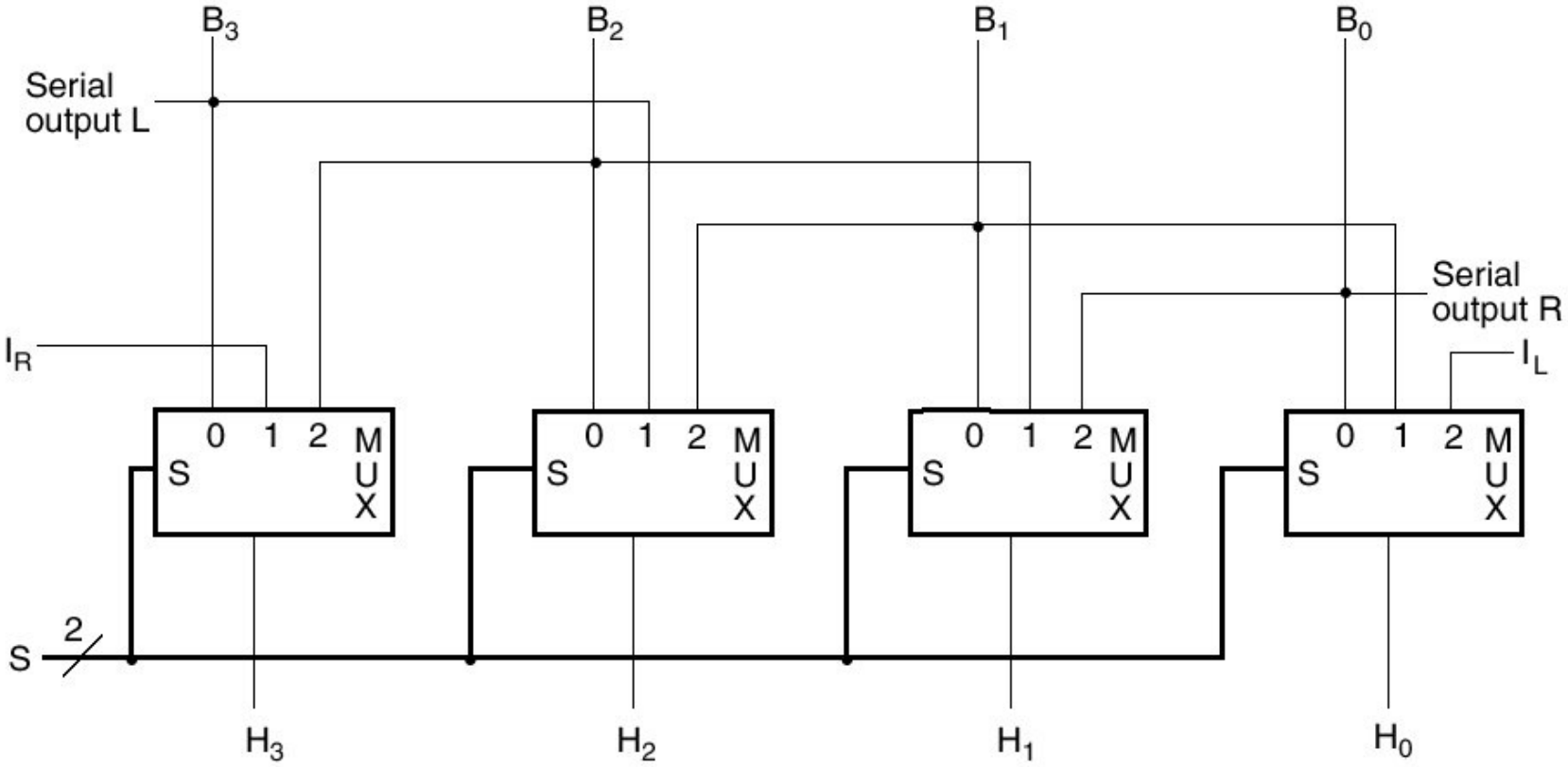
CS2022 Carry-Lookahead Adder

- With a carry-lookahead adder this gives us a fast combinational ALU with the following functionality:

Select				Output	
S_2	S_1	S_0	C_{in}		
0	0	0	0	G = A	TRANSFER
0	0	0	1	G = A + 1	INCREMENT
0	0	1	0	G = A + B	ADD
0	0	1	1	G = A + B +1	ADD WITH C
0	1	0	0	G = A + B	A plus 1's C.B
0	1	0	1	G = A + B +1	SUBTRACT
0	1	1	0	G = A - 1	DECREMENT
0	1	1	1	G = A	TRANSFER
1	0	0	X	G = A $\wedge$ B	AND
1	0	1	X	G = A $\vee$ B	OR
1	1	0	X	G = A $\oplus$ B	XOR
1	1	1	X	G = A	NOT

CS2022 4-Bit SR/SL Shifter Unit

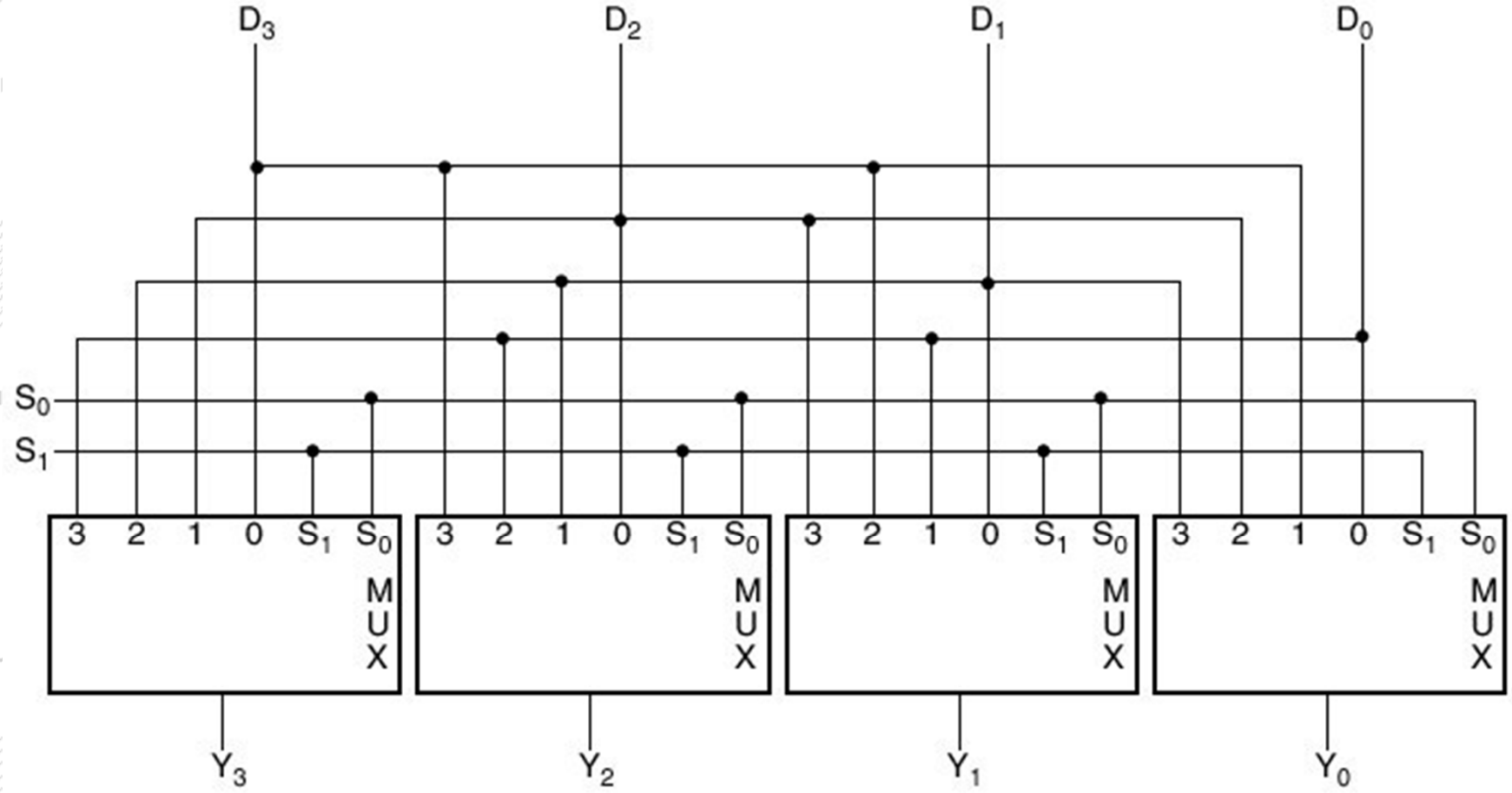
- For speed of execution the shifter unit is always implemented as a combinational circuit based on a MUX:



CS2022 Shift Operations

- By controlling IR and IL with multiplexers it is straightforward to adapt this circuit to perform arithmetic shifts, constructive and destructive shifts, Rotates and rotate-then-carry operation

CS2022 4-Bit Barrel Shifter



CS2022 Multiple Shifts

- If multiple shifts are required we wire them into multiplexers that have an input for every bit on the bus to obtain a Barrel shifter (Previous slide).

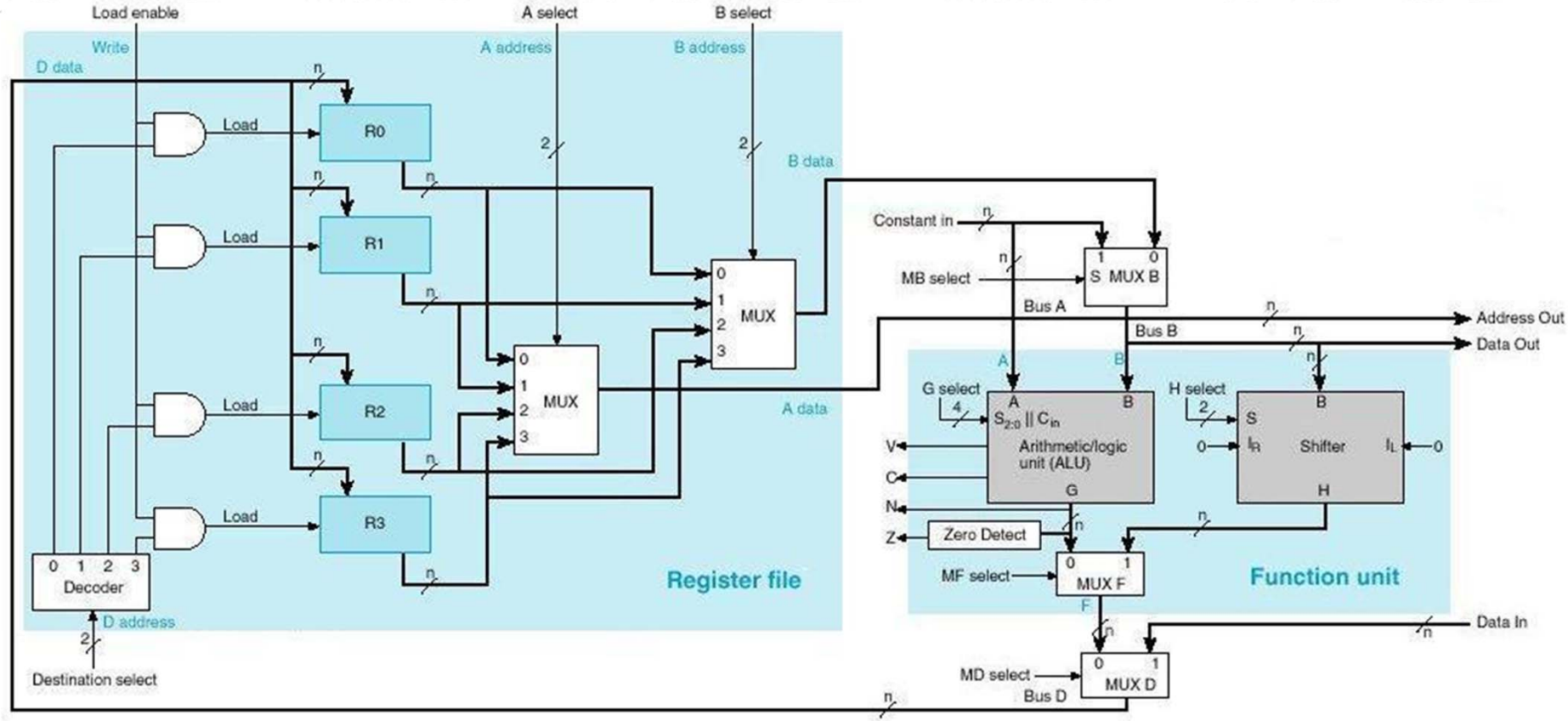
S_1 S_2	Y_3 Y_2 Y_1 Y_0	Micro-ops
0 0	D_3 D_2 D_1 D_0	No Rotate
0 1	D_2 D_1 D_0 D_3	Rotate One
1 0	D_1 D_0 D_3 D_2	Rotate Two
1 1	D_0 D_3 D_2 D_1	Rotate Three

CS2022 Controlling a Datapath

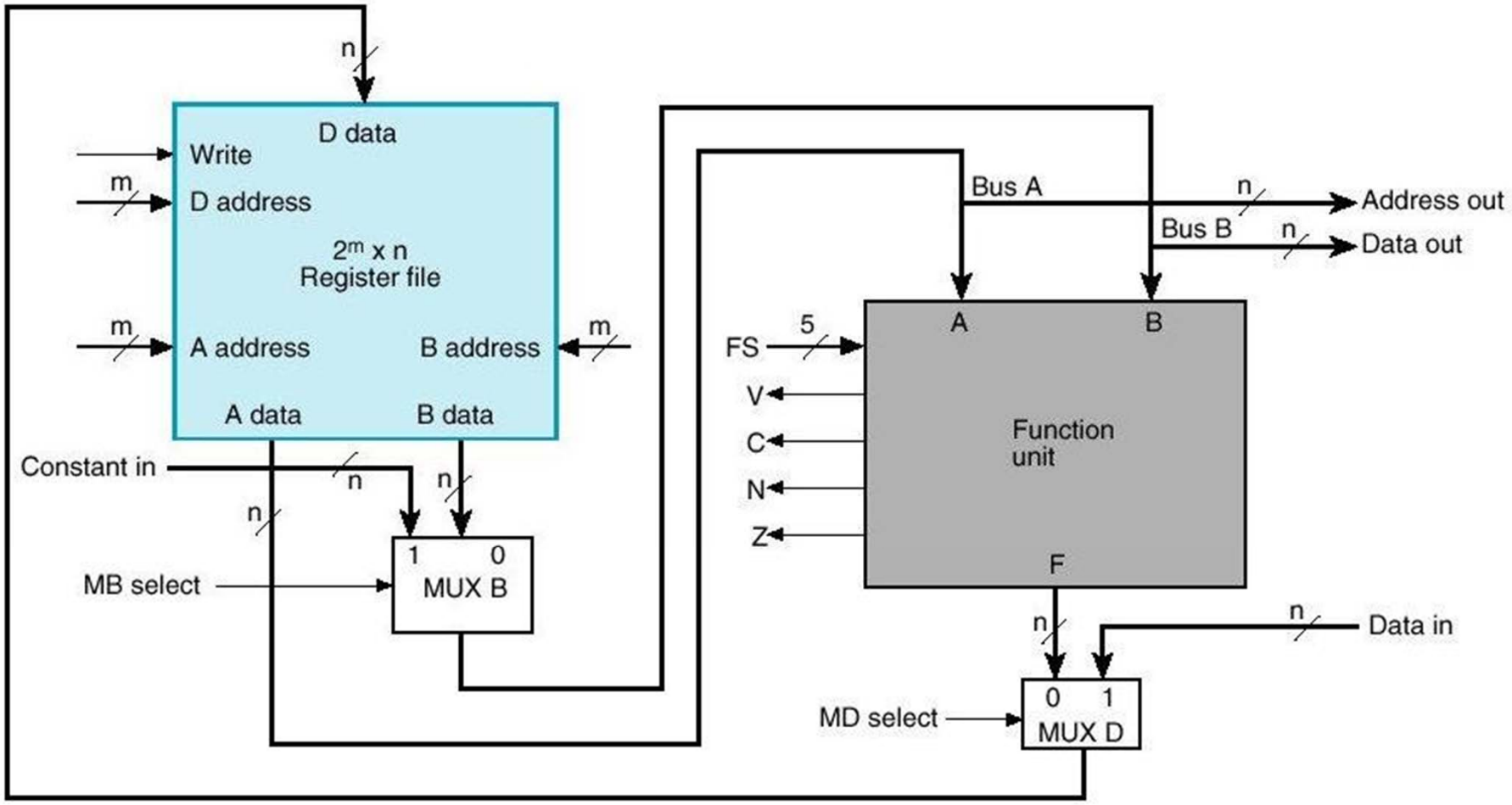
The Control Word

- The figure on the next slide is an updated version of our introductory datapath (4th Lecture, Page 12) where the register file has been expanded to a more realistic eight n-bit registers.
- Consequently the destination decoder and A and B bus MUX require three-bit select input.
- The Function Unit still requires five bits to select ALU/Shift micro-ops.
- Three more bits are required to control:
  - Writing to the registers (RW)
  - MUX B (MB)
  - MUX D (MD)

CS2022 4th Lecture - Page 12



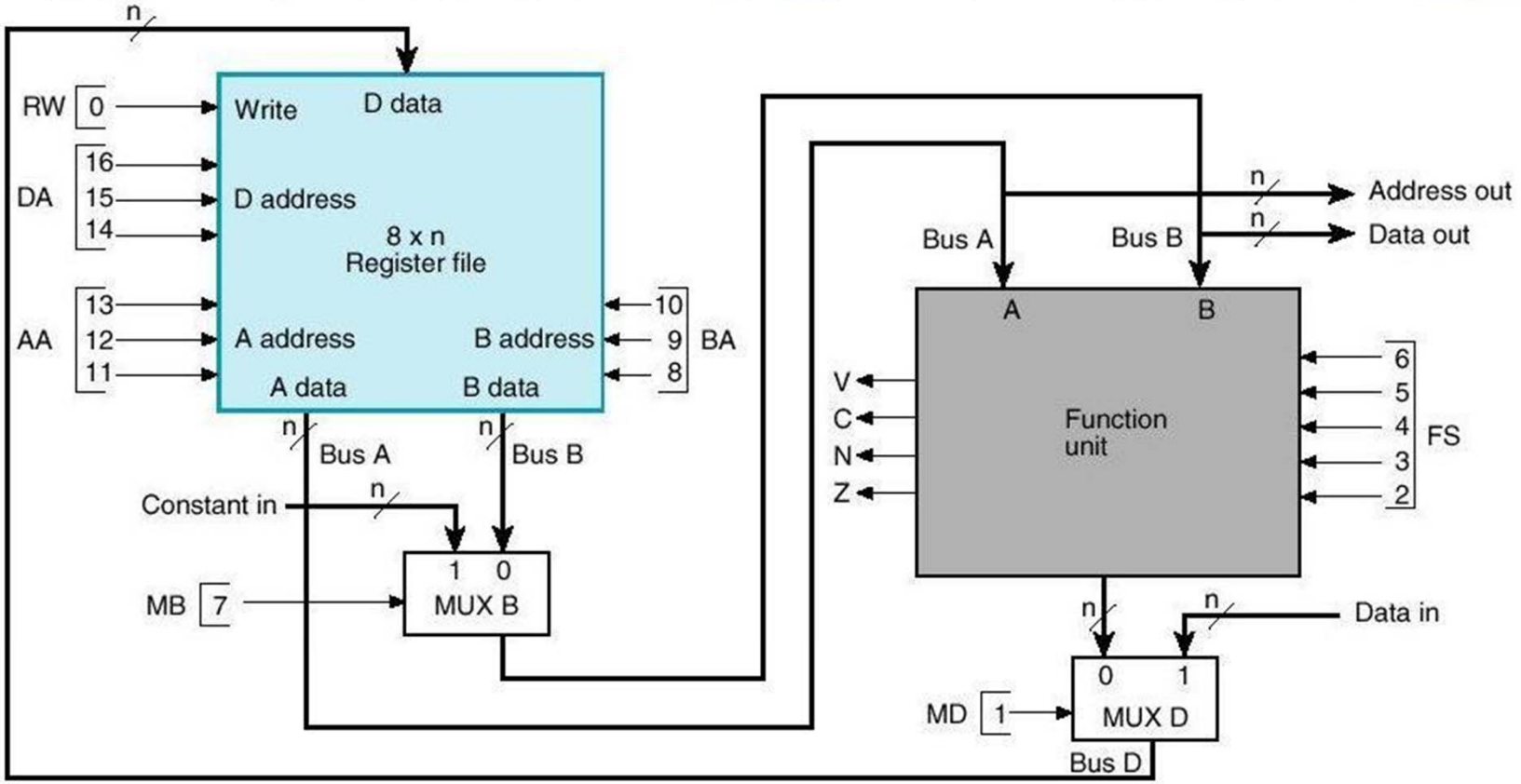
CS2022 Updated Datapath

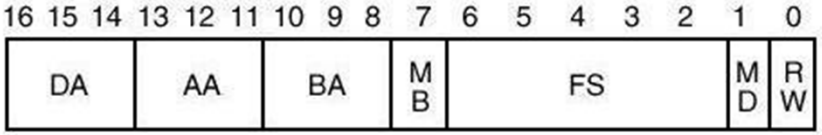


CS2022 Control Word

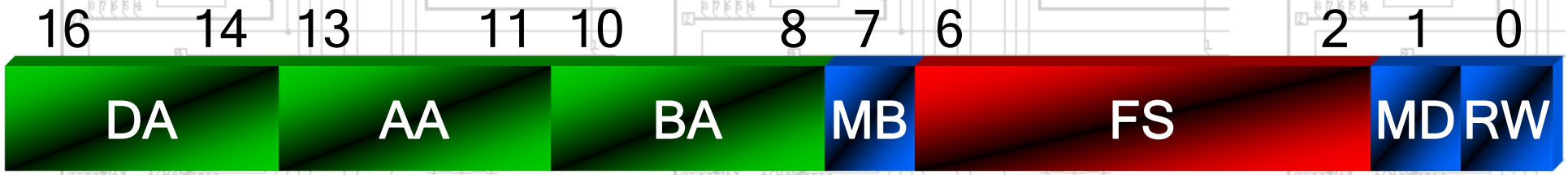
- The schematic on the next slide identifies all these control inputs and arranges them in a 17-bit vector called the Control Word.

CS2022 Datapath and the Control Word





The Control Word Specifies One Micro-ops

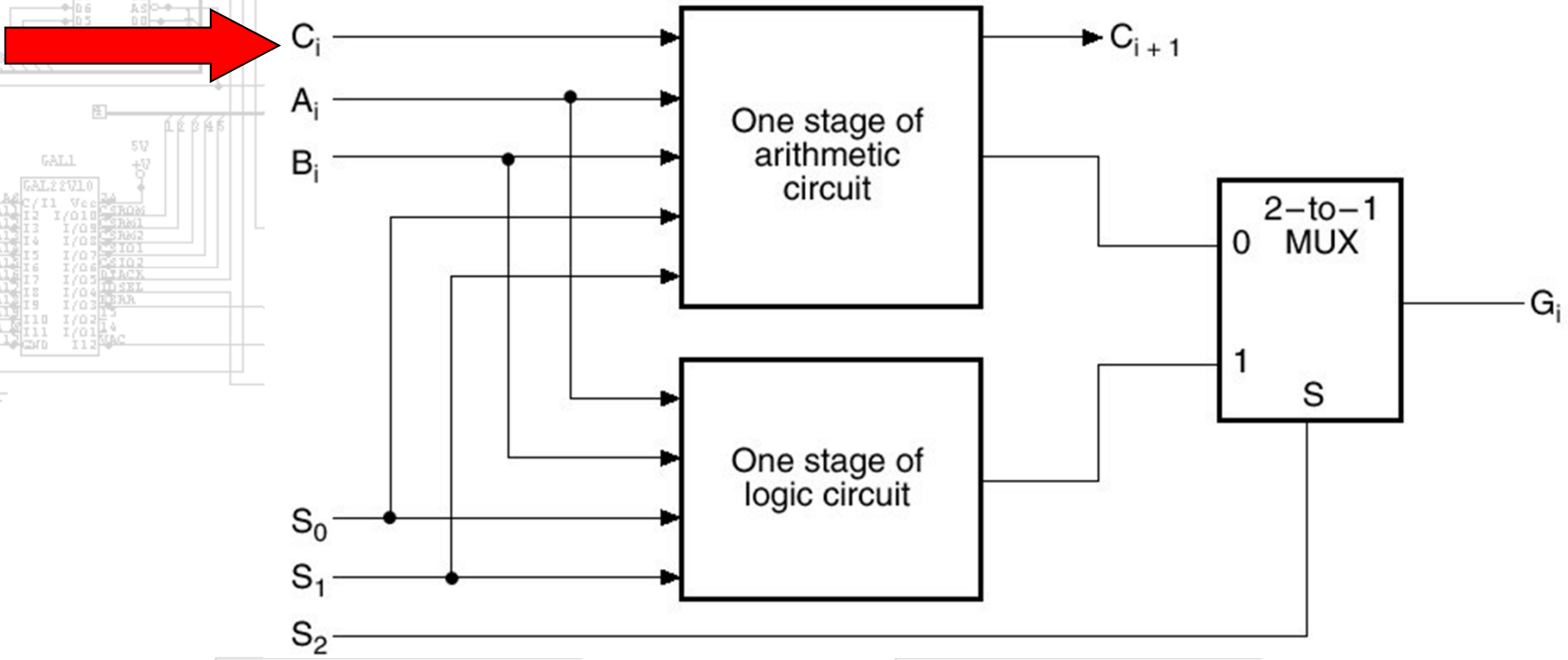


- Note that for the function select FS we need to make explicit the relationship between its value and the micro-ops.
  - See next slide for details.

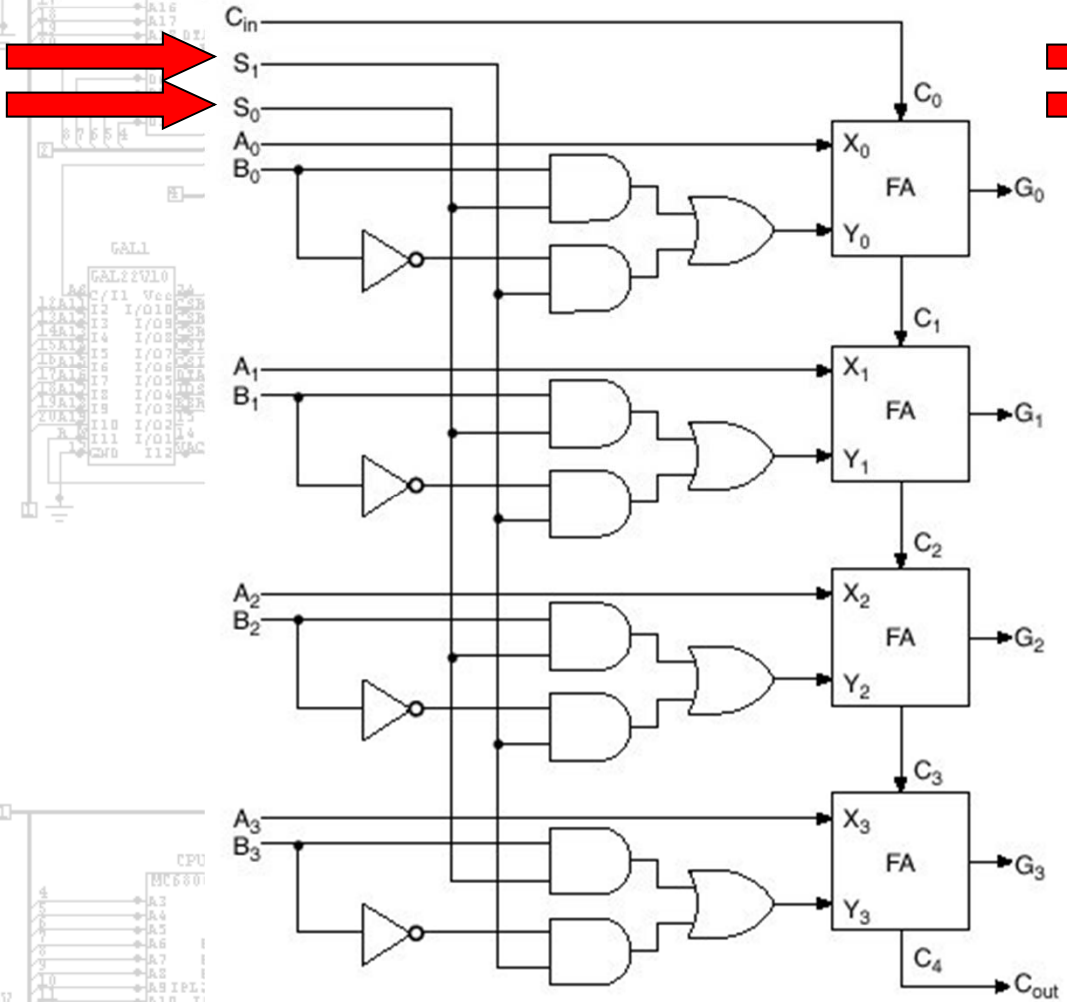
G Select, H Select and MF Select determine the FS code

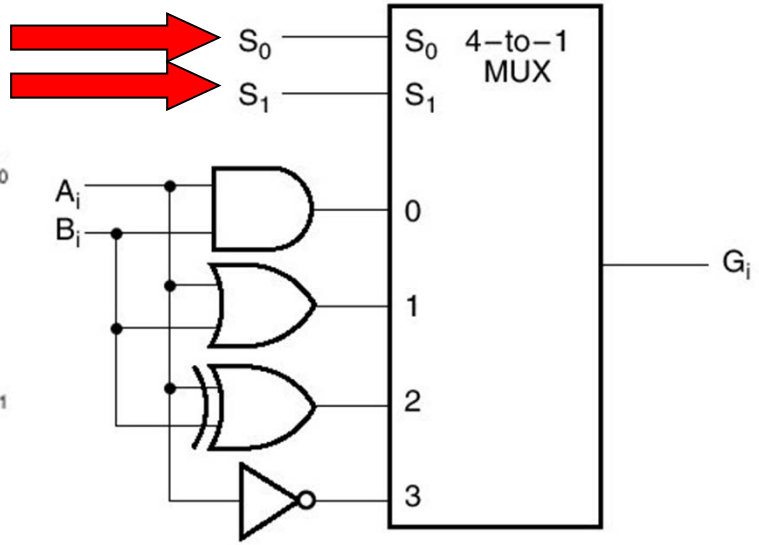
FS	MF Select	G Select	H Select	Output	
00000	0	0000	XX	G = A	TRANSFER
00001	0	0001	XX	G = A + 1	INCREMENT
00010	0	0010	XX	G = A + B	ADD
00011	0	0011	XX	G = A + B +1	ADD WITH C
00100	0	0100	XX	G = A + $\overline{B}$	A plus 1's C.B
00101	0	0101	XX	G = A + $\overline{B}$ +1	SUBTRACT
00110	0	0110	XX	G = A - 1	DECREMENT
00111	0	0111	XX	G = A	TRANSFER
01000	0	1000	XX	G = A $\wedge$ B	AND
01010	0	1010	XX	G = A $\vee$ B	OR
01100	0	1100	XX	G = $\overline{A} \oplus$ B	XOR
01110	0	1110	XX	G = $\overline{A}$	NOT
10000	1	XXXX	00	G = B	TRANSFER
10100	1	XXXX	01	G = sr B	SHIFT RIGTH
11000	1	XXXX	10	G = sl B	SHIFT LEFT

CS2022 FS[2]= C_{in}

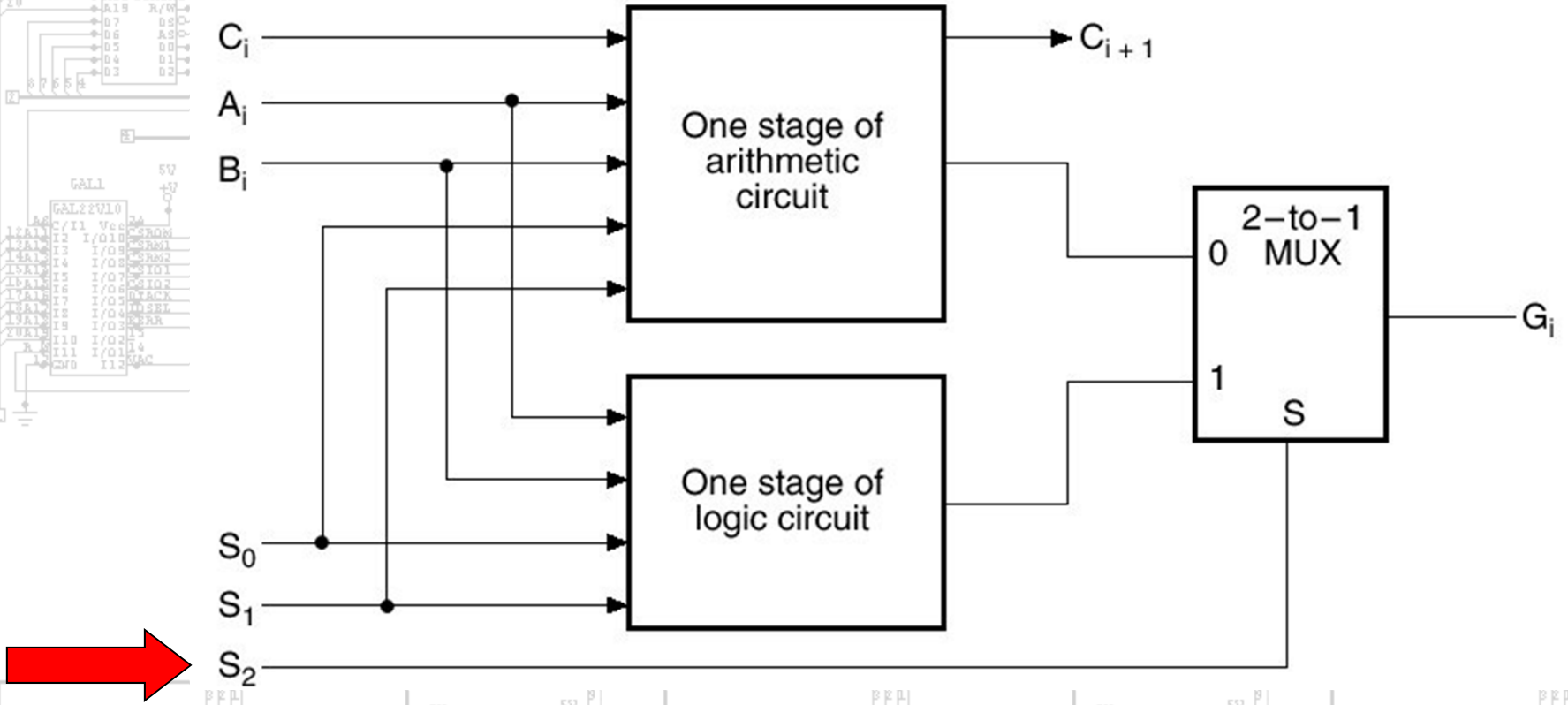


CS2022 $FS[3 \wedge 4] = S_0 S_1$

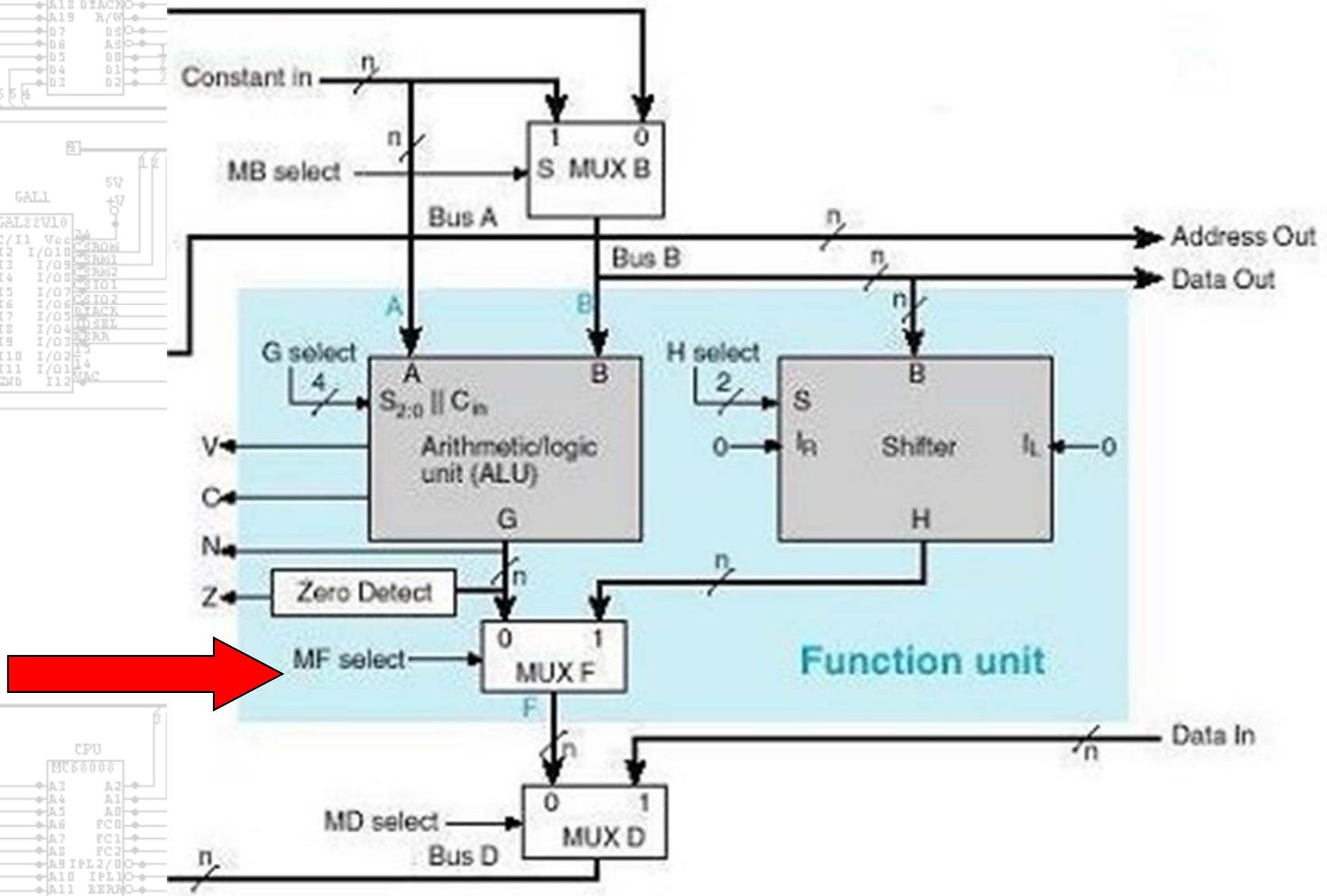




CS2022 FS[5]=S_2



CS2022 FS[6]=MF select



CS2022 RT Description of the Datapath

RW: R[DA] ← If MD then
FS(R[AA], if MB then R[BA]
else Constant in)
else DATA in,

$V \leftarrow C_n \oplus C_{n-1}$,
$C \leftarrow C_n$,
$Z \leftarrow R_{n-1}, R_{n-2}, \ldots, R_0$,
$N \leftarrow R_{n-1}$,

CS2022 Symbolic Notation for Micro-ops

- Because human beings working in binary code tend to be highly error-prone, we usually employ intuitive symbols to specify datapath micro-ops.
- Typical symbol/code assignments are:

CS2022 Symbol-binary Map of Control Word Fields

DA, AA, BA	
Function	Code
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

MB	
Function	Code
Register	0
Constant	1

MD	
Function	Code
Function	0
Data In	1

RW	
Function	Code
No Write	0
Write	1

FS	
Function	Code
G = A	00000
G = A + 1	00001
G = A + B	00010
G = A + B +1	00011
G = A + $\overline{B}$	00100
G = A + $\overline{B}$ +1	00101
G = A - 1	00110
G = A	00111
G = A $\wedge$ B	01000
G = A $\vee$ B	01010
G = A $\oplus$ B	01100
G = $\overline{A}$	01110
G = B	10000
G = sr B	10100
G = sl B	11000

CS2022 Symbol Conversion

- With the symbolic notation it is easy to accurately specify control words which may then be automatically converted to binary.
- For example: R1←R2 + $\overline{R3}$ +1

Field :	DA	AA	BA	MB	FS	MD	RW
Symbol:	R1	R2	R3	Register	F=A+$\overline{B}$+1	Function	Write
Binary:	001	010	011	0	00101	0	1

CS2022 Microoperations Example

- DIY - Convert these to binary and check your results against the table on the next slide.

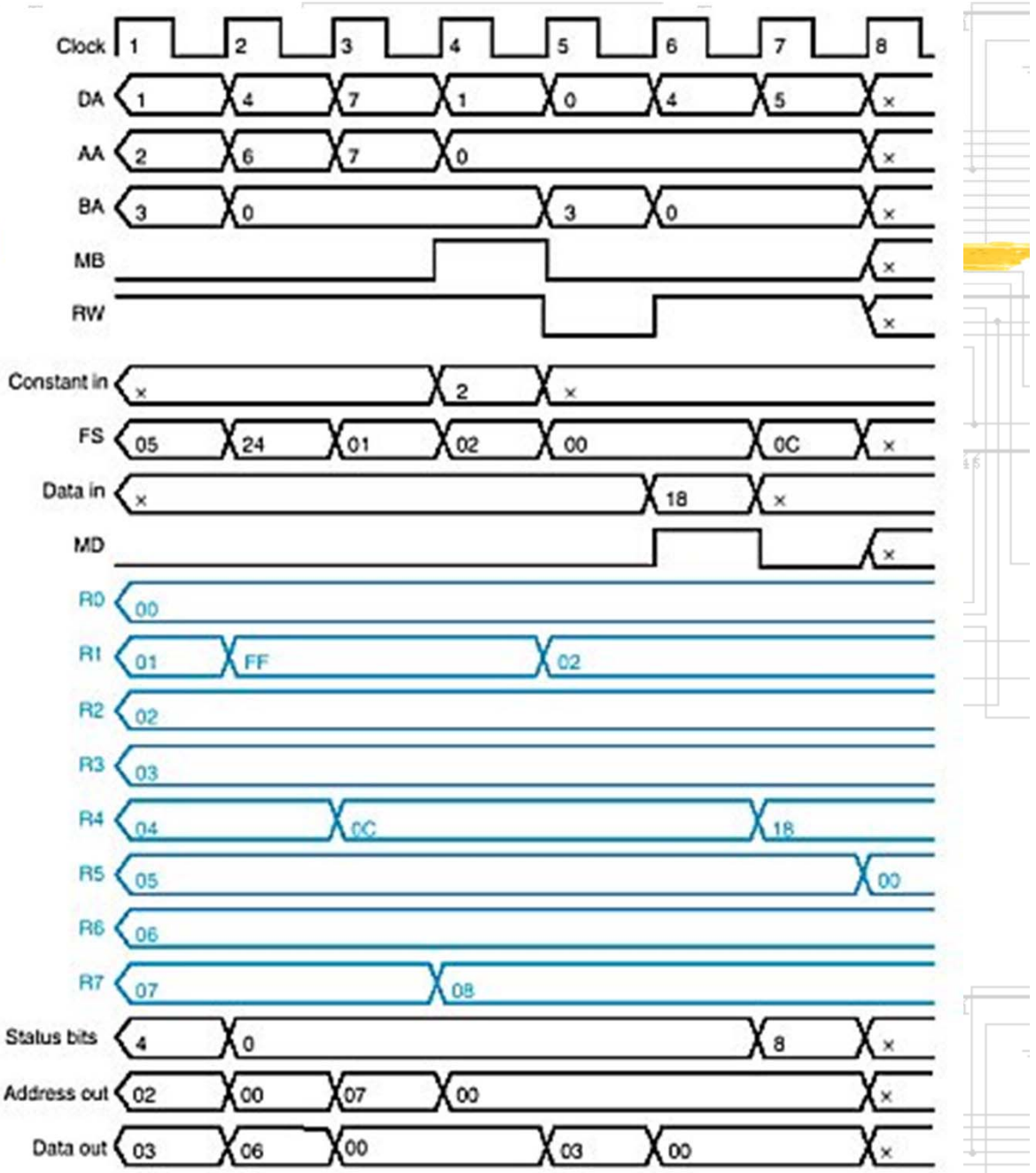
Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 + \overline{R3} + 1$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
$R4 \leftarrow$ sl R6	$R4$	—	$R6$	Register	$F =$ sl B	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	Register	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	$R0$	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	—	—	$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	—	—	—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

CS2022 Binary Control Words from Example

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	00101	0	1
$R4 \leftarrow$ sl R6	100	000	110	0	11000	0	1
$R7 \leftarrow R7 + 1$	111	111	000	0	00001	0	1
$R1 \leftarrow R0 + 2$	001	000	000	1	00010	0	1
Data out $\leftarrow R3$	000	000	011	0	00000	0	0
$R4 \leftarrow$ Data in	100	000	000	0	00000	1	1
$R5 \leftarrow 0$	101	000	000	0	01100	0	1

CS2022

- Examine the figure:
  - Register transfer on Clock ↑.
  - R0 – R7 are initialised to $R_i \leftarrow i$.

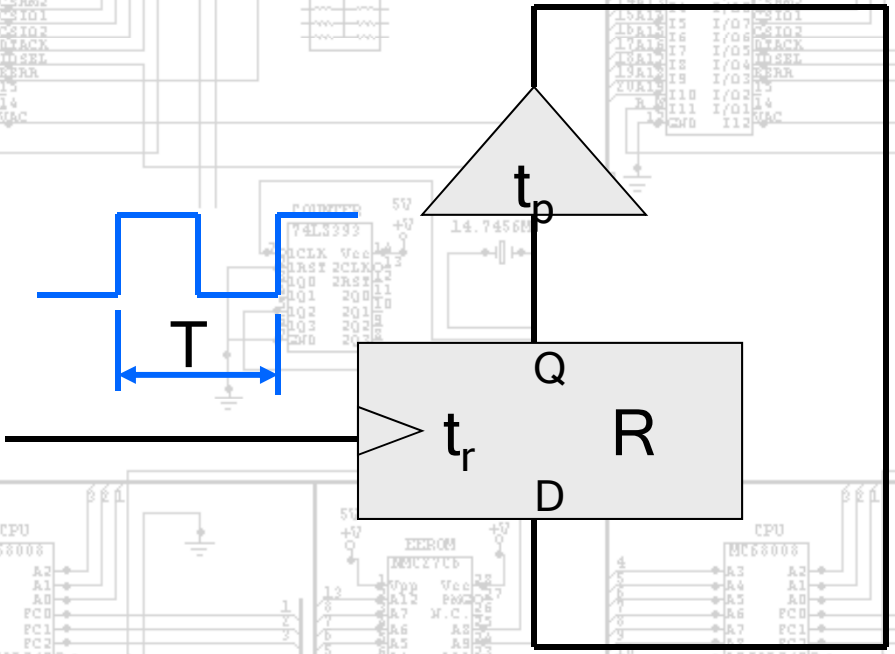


CS2022 Status Bits

- Status bits shows the input for the Zero, Negative, Carry-out and Overflow bits respectively.
- Hence the change following a control word change.

CS2022 Datapath Timing

- The total, worst case propagation delay determines the maximum rate at which we may clock a system.

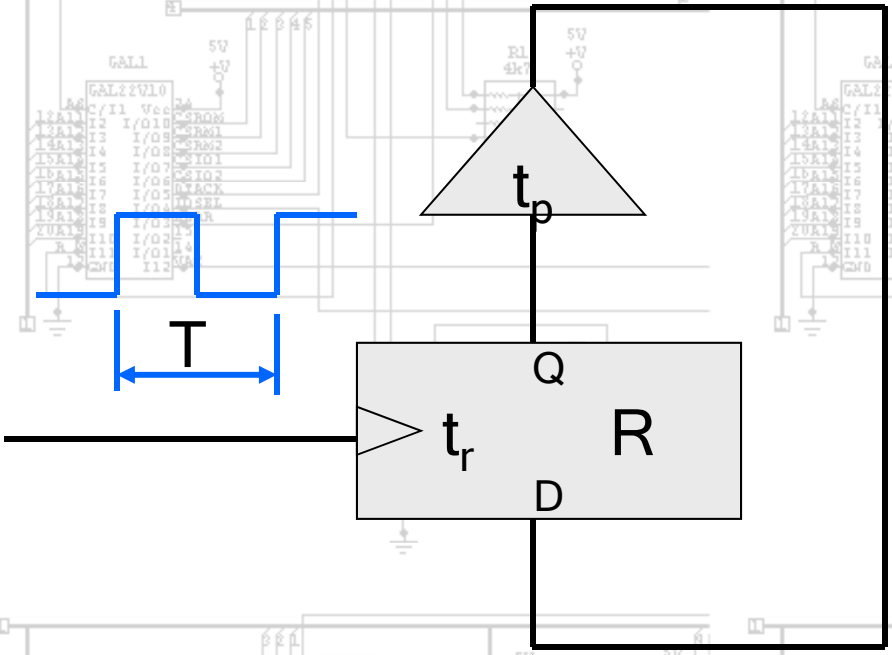


Combinational logic with delay t_p

Register with delay t_r

CS2022 Timing

- For successful operations we must have:



$$T \geq t_p + t_r$$

$$T_{min} = t_p + t_r$$

$$f_{max} = 1 / T_{min}$$

CS2022 Conventional and Pipelined Datapath Timing

t_r = 1ns

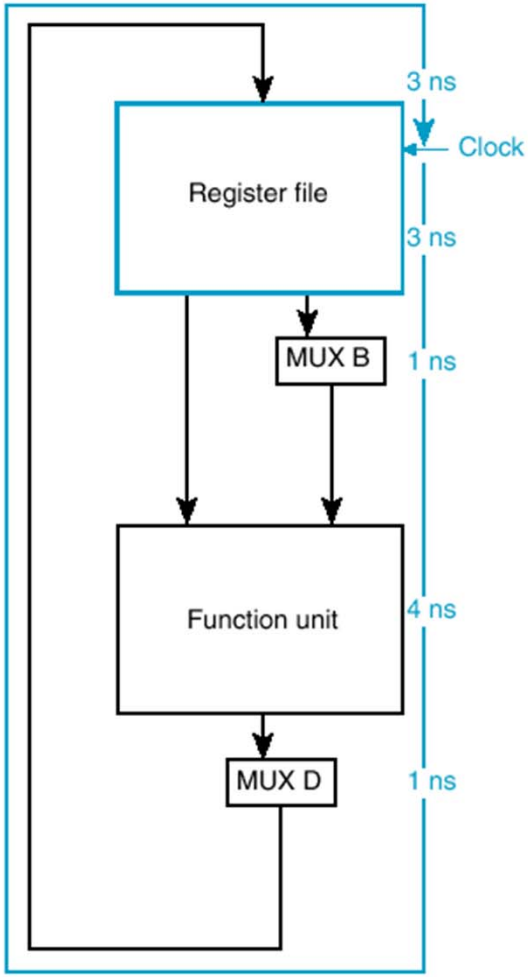
t_p = 3+1+4+1+2ns

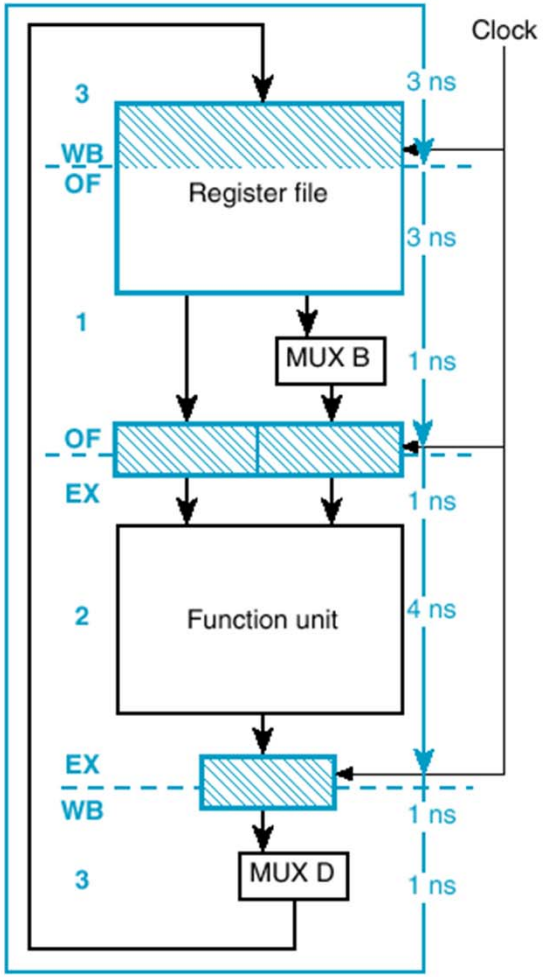
= 11ns

T_{min} = 12ns

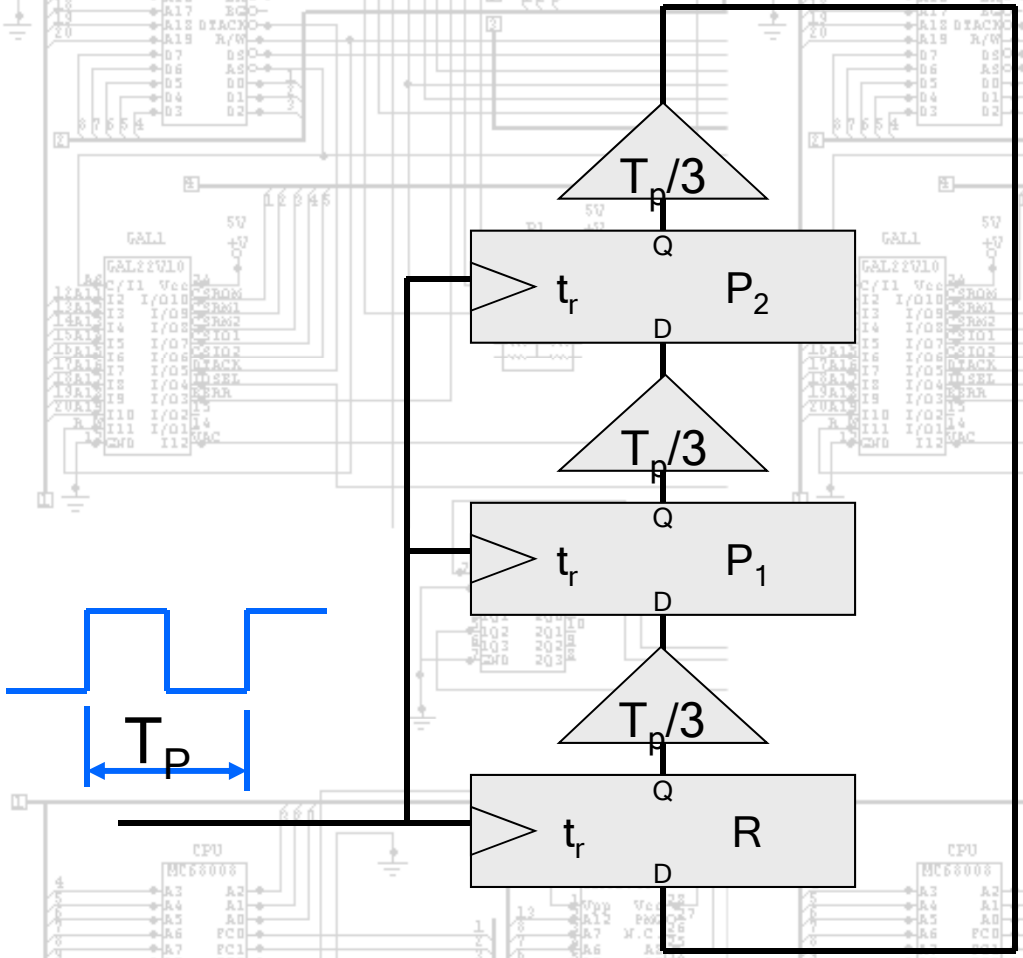
f_{max} = 1 / T_{min}

= 83.3MHz





CS2022 Three-Stage Pipeline



- To speed things up we can introduce register into the combinational logic.

$$T_P \geq t_p/3 + t_r$$

$$T_{max} = 1+4ns$$

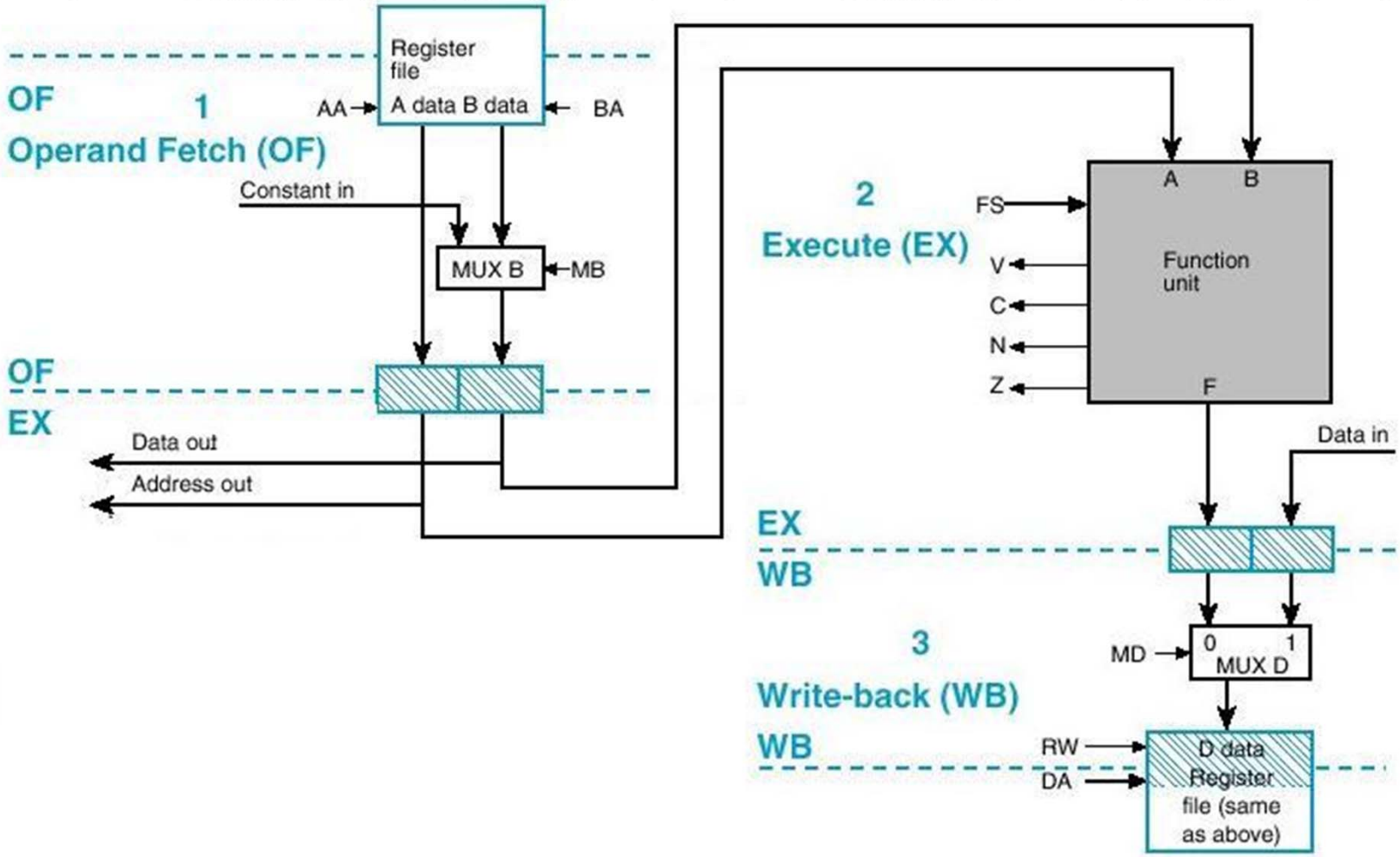
$$f_{max} = 1 / T_{max}$$

$$=200MHz$$

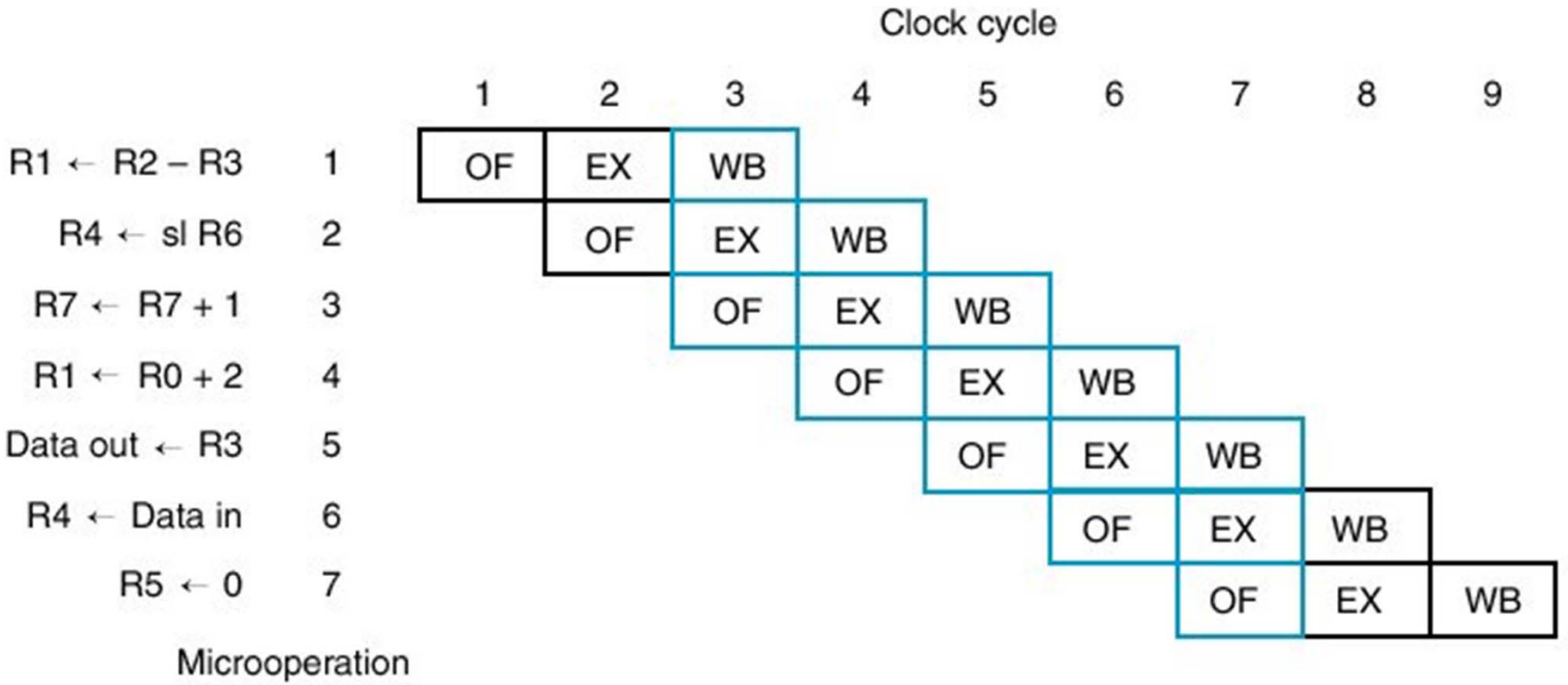
CS2022 Pipelined Datapath

- When a datapath is pipelined it divides naturally into stages seen on the next slide.
  - OF = Operand Fetch
  - EX = Execute
  - WB = Write Back

CS2022 Pipelined Datapath Schematic

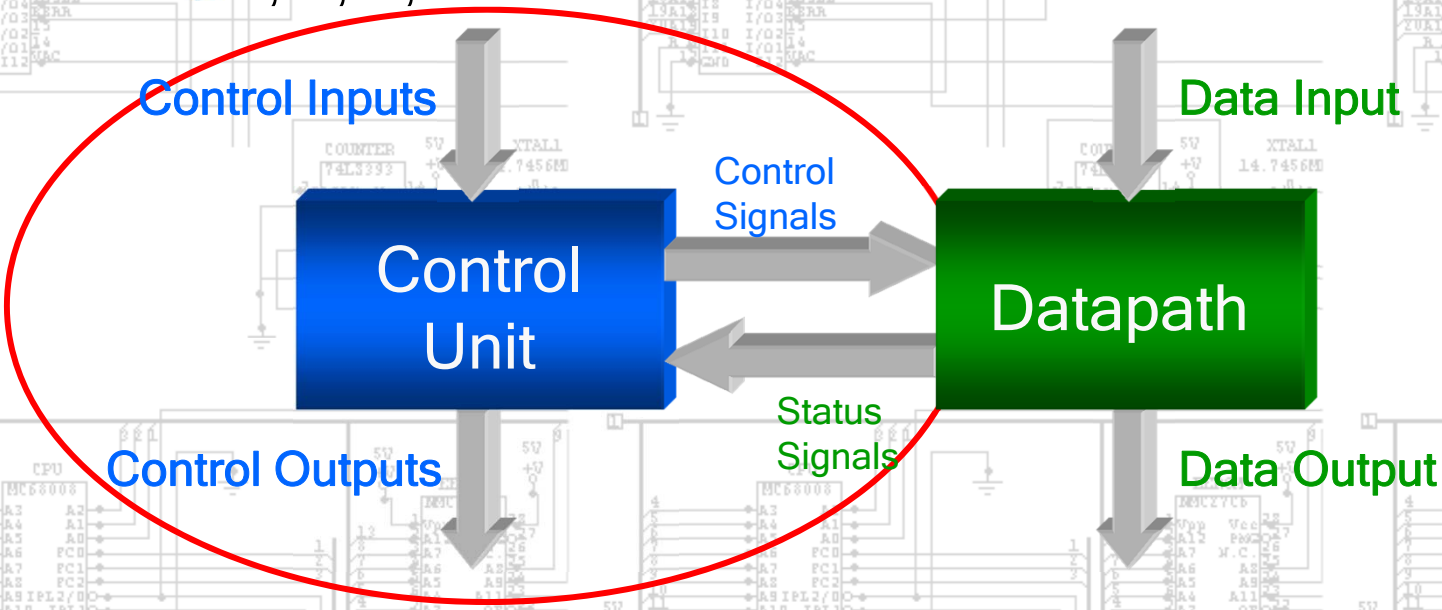


CS2022 Pipe Execution Pattern



CS2022 The Control Unit

- Control unit's job:
  - Supply all the control signals to the datapath
  - Respond appropriately to its status signals:
    - Z, N, C, V



CS2022 Von Neumann Architecture

- Input to the control unit:
  - A stream of instructions coming from memory **M**
  - This steam must be converted to a sequence of micro-operations for the datapath
- Control Unit uses:
  - Program counter **PC** to index in M the next executable instruction

CS2022 Algorithmic State Machine

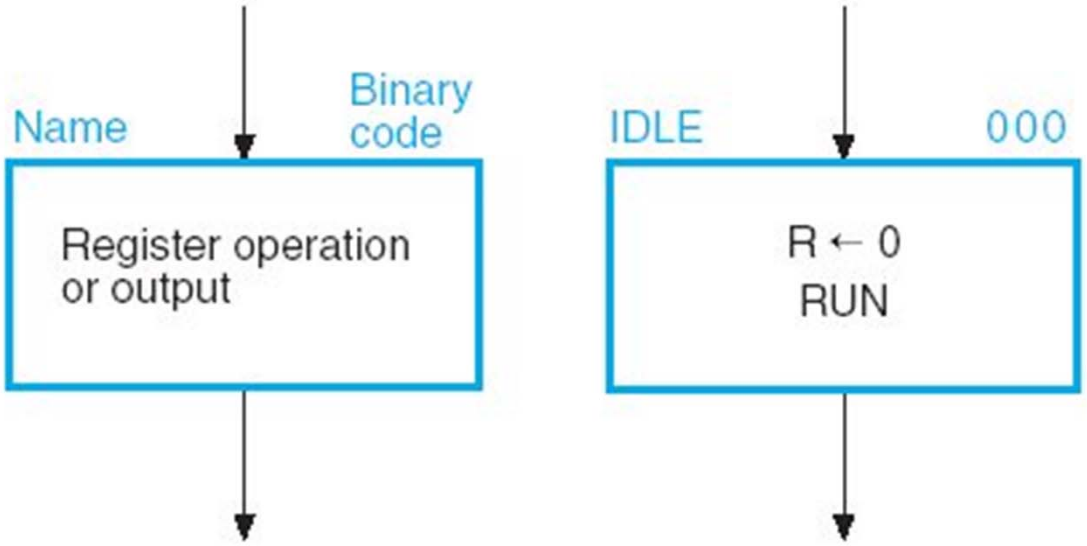
- Data processing may be achieved through:
  - Sequencing Register transfer operations
  - May be specified as hardware algorithm
    - Consists of a finite number of procedural steps
- ASM are used:
  - Control Unit
  - Datapath

CS2022 ASM Chart

- Algorithmic State Machine (ASM) Chart
  - Defines the hardware algorithm
  - Defines relation ship to time
    - Clock
- Three basic elements:
  - State Box
  - Decision Box
  - Conditional Output Box

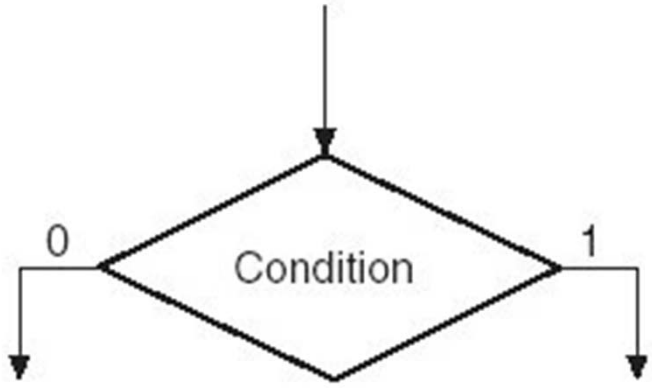
CS2022 State Box

- State Box contains:
  - Register transfer operation or output signals that are activated while the control unit is in this state.
  - RUN is 1 for any box it appears and 0 for any box it does not appear.



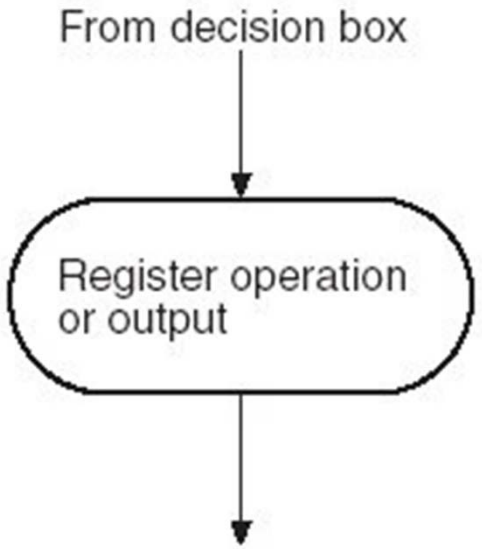
CS2022 Decision Box

- Exit path is taken if input condition is:
  - True (1)
  - False (0)

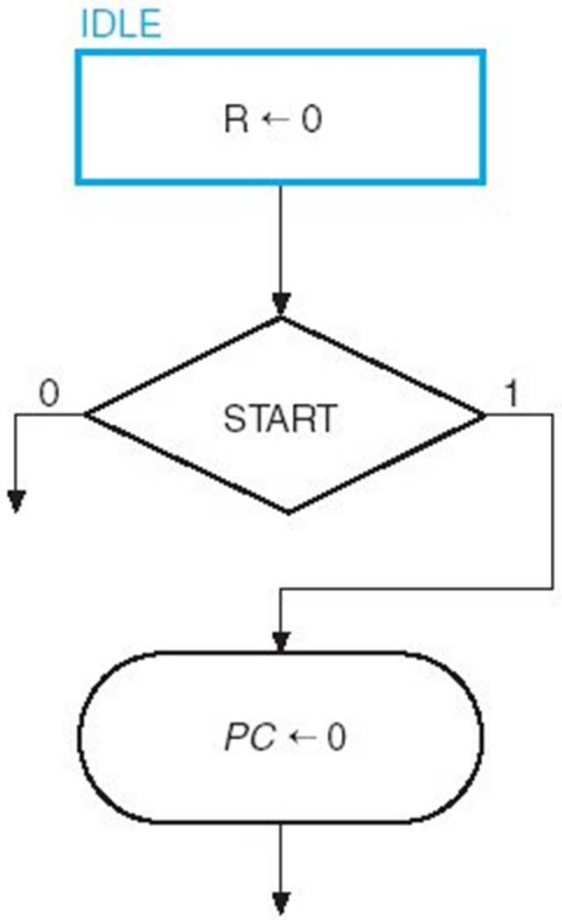


CS2022 Conditional Output Box

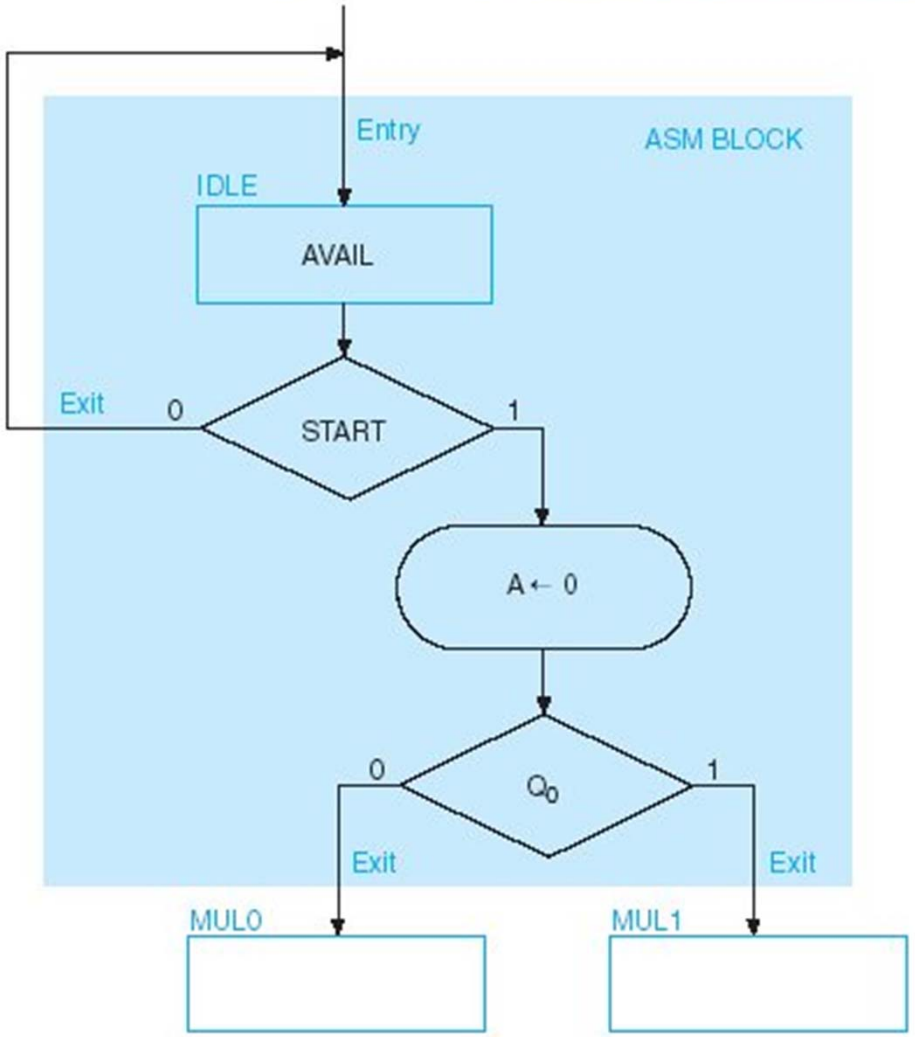
- Conditional Output Box entry path must pass through one or more decision boxes.



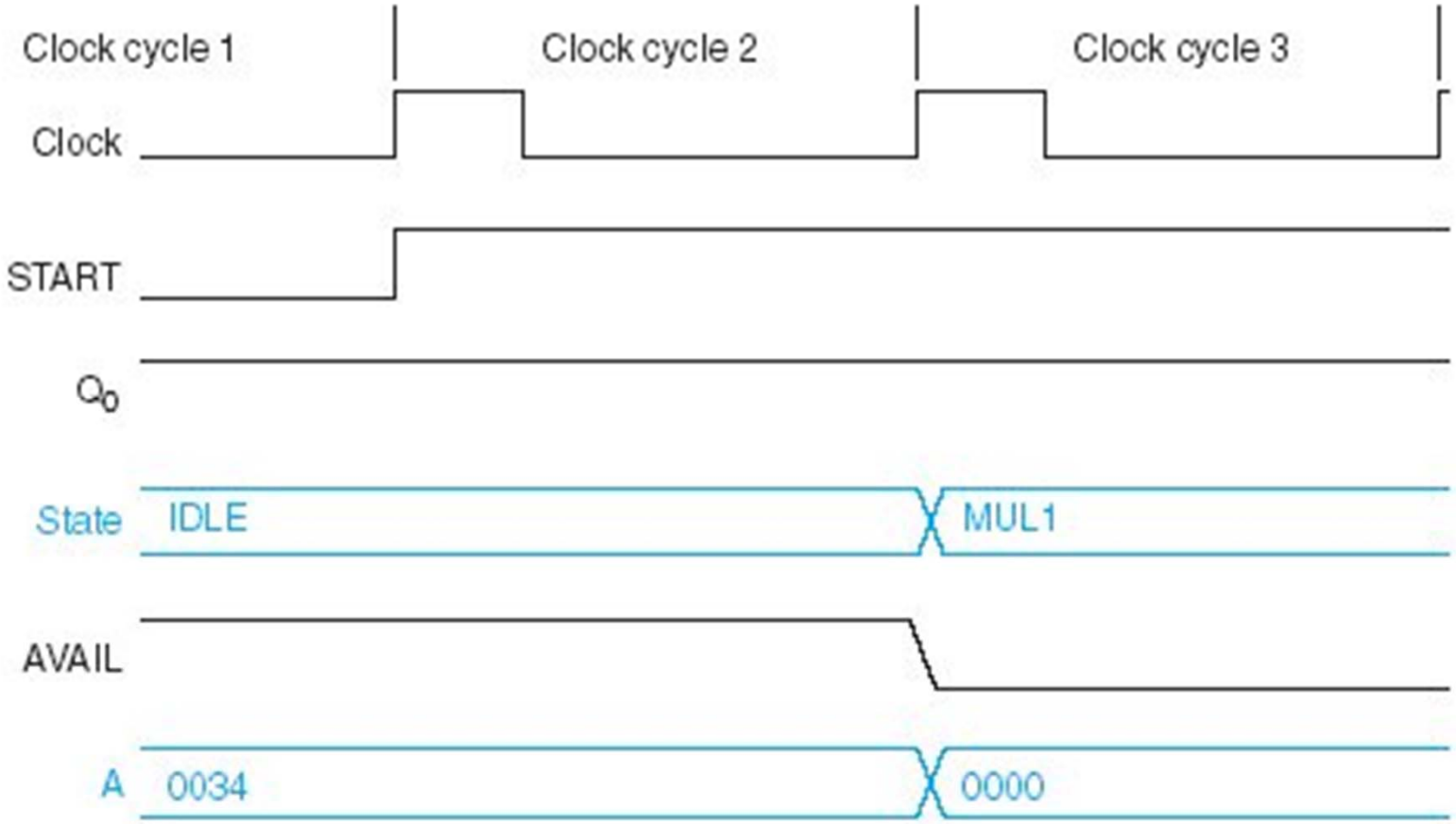
CS2022 ASM Box Example



CS2022 ASM Block



CS2022 Timing



CS2022 Control Unit Design

- Two contrasting approaches to control unit design have evolved:
  - Hard-wired
  - Micro-coded

CS2022 Example

- We will consider a shift-and-add multiply circuit as an example of each design approach.
- If **A** and **B** are n-bit unsigned integers.
  - To compute their product **P**:

$$P = A \times B$$

Product = Multiplier × Multiplicand

CS2022 Bit Products

- We can generate the bit products:

$$P_i \quad i=0,\ n-1$$

$$P_i = a_i \times B = \left\{ \begin{array}{l} 0 \text{ if } a_i=0 \\ B \text{ if } a_i=1 \end{array} \right\} = a_i \wedge B$$

$$PP_j = \sum_{i=0}^{j} P_i \times 2^i = P_j \times 2^j + PP_{j-1}$$

$$P = PP_{n-1}$$

CS2022 Hand Multiplication

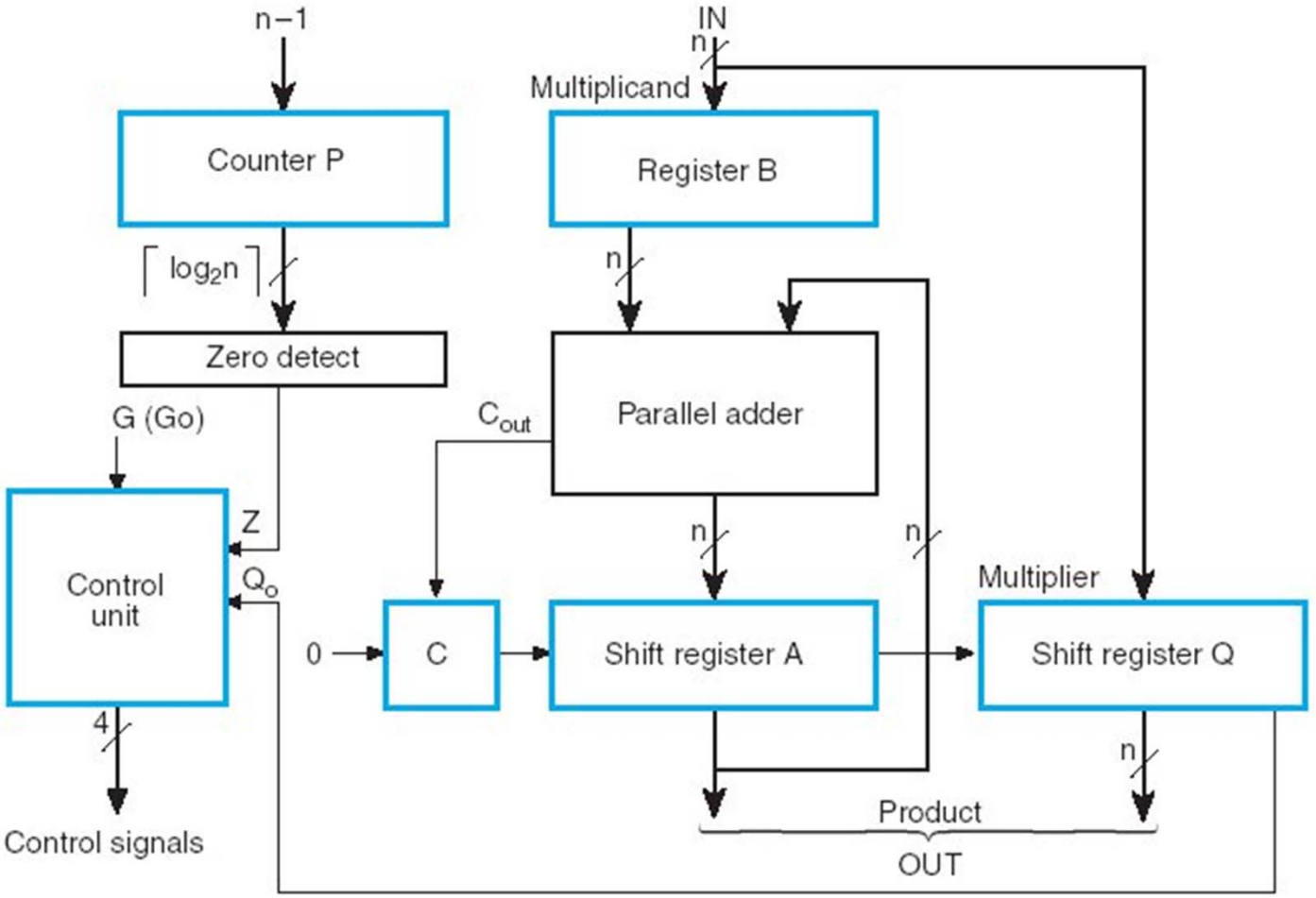
23	10111	Multiplicand	
19	10011	Multiplier	
	10111	$\leftarrow P_0$	
	10111	$\leftarrow P_1 \times 2^1$	
	00000	$\leftarrow P_2 \times 2^2$	
	00000	$\leftarrow P_3 \times 2^3$	
	10111	$\leftarrow P_4 \times 2^4$	
437	110110101	Product	

CS2022 Hardware Multiplication

23	10111	Multiplicand
19	10011	Multiplier
	00000	Initial partial product
	10111	Add multiplicand, since multiplier bit is 1
	10111	Partial product after add and before shift
	010111	Partial product after shift
	10111	Add multiplicand, since multiplier bit is 1
	1000101	Partial product after add and before shift[a]
	1000101	Partial product after shift
	01000101	Partial product after shift
	001000101	Partial product after shift
	10111	Add multiplicand, since multiplier bit is 1
	110110101	Partial product after add and before shift
437	0110110101	Product after final shift

- a. Overflow temporarily occurred

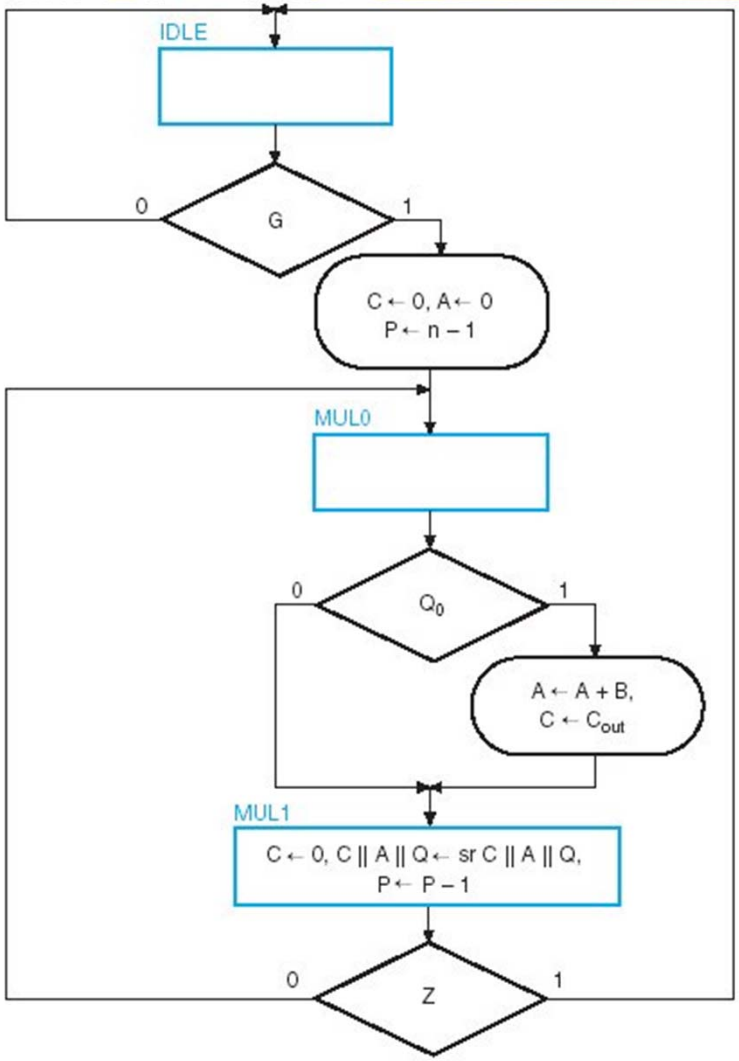
CS2022 Binary Multiplier Diagram



CS2022 Binary Multiplier

- The figure on the previous slide shows:
  - Datapath
  - Status signals **Z** and Q_0
  - External Input
  - **G** = Go
  - Output

CS2022 Binary Multiplier ASM



CS2022

Shift-and-Add Multiplication ASM

- Note the concatenation notation
- From the ASM we can write out the RT description of the system in terms of:
  - System state
  - Input signals
- The table on the following slide allows us to deduce the design of each register:

CS2022 Control and Sequencing

- Two distinct aspects in control unit design
  - Control of micro-operations
  - Sequencing
- We separate the two aspects by providing:
  - A state table
    - Defines signals in terms of states and inputs
  - A simplified ASM chart
    - Represents only state transitions

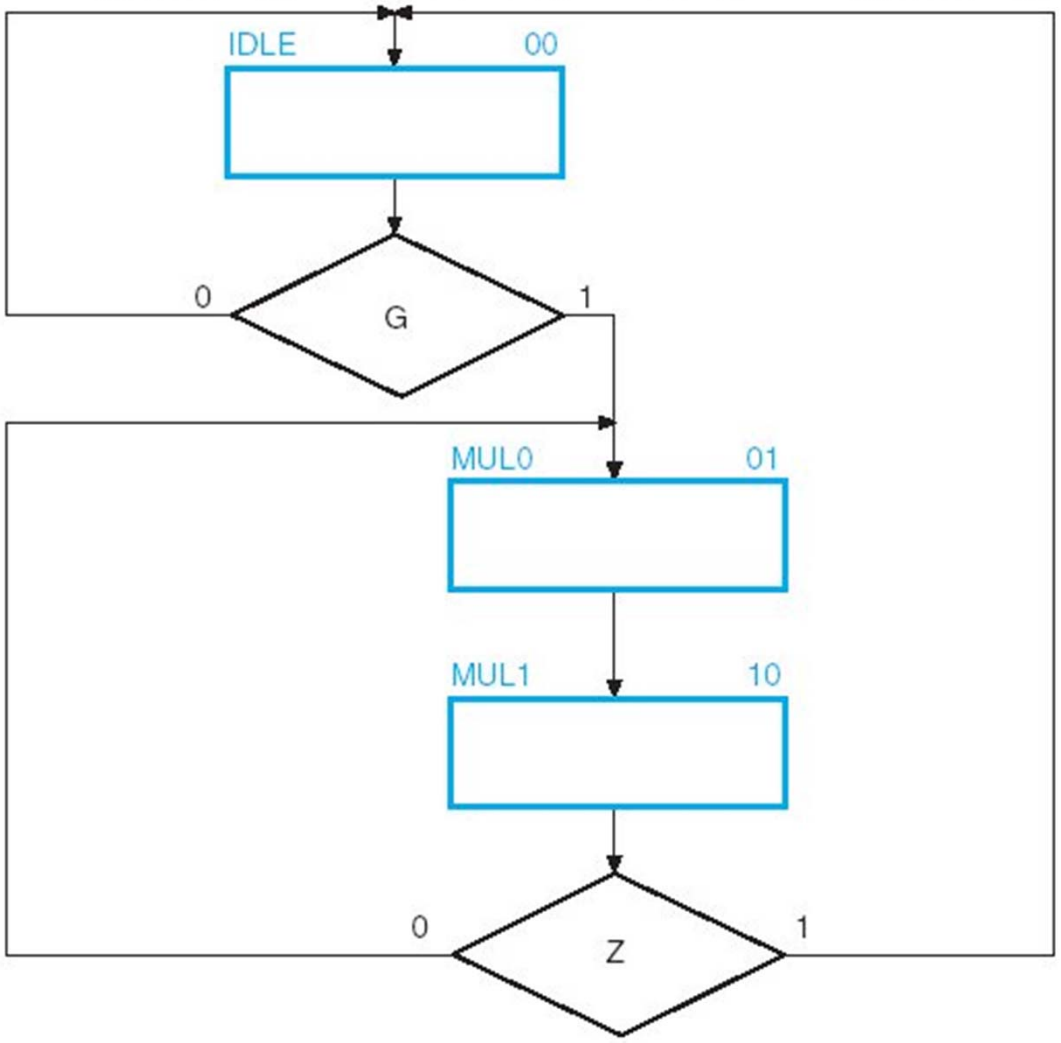
CS2022 Register Transfers

- From the ASM we can write out the RT description of the system in terms of:
  - System state
  - Input signals
- By gathering together the RTs loading each register we may easily deduce the design of each register.

CS2022 Control Signals for Binary Multiplier

Block Diagram Module	Microoperation	Control Signal Name	Control Expression
Register A:	$A \leftarrow 0$	Initialize	$\text{IDLE} \cdot G$
	$A \leftarrow A + B$	Load	$\text{MUL0} \cdot Q_0$
	$C \| A \| Q \leftarrow \text{sr}\ C \| A \| Q$	Shift_dec	MUL1
Register B:	$B \leftarrow IN$	Load_B	LOADB
Flip-Flop C:	$C \leftarrow 0$	Clear_C	$\text{IDLE} \cdot G + \text{MUL1}$
	$C \leftarrow C_{\text{out}}$	Load	—
Register Q:	$Q \leftarrow IN$	Load_Q	LOADQ
	$C \| A \| Q \leftarrow \text{sr}\ C \| A \| Q$	Shift_dec	—
Counter P:	$P \leftarrow n - 1$	Initialize	—
	$P \leftarrow P - 1$	Shift_dec	—

CS2022 Sequencing Part of ASM Chart



CS2022 Sequence Register and Decoder

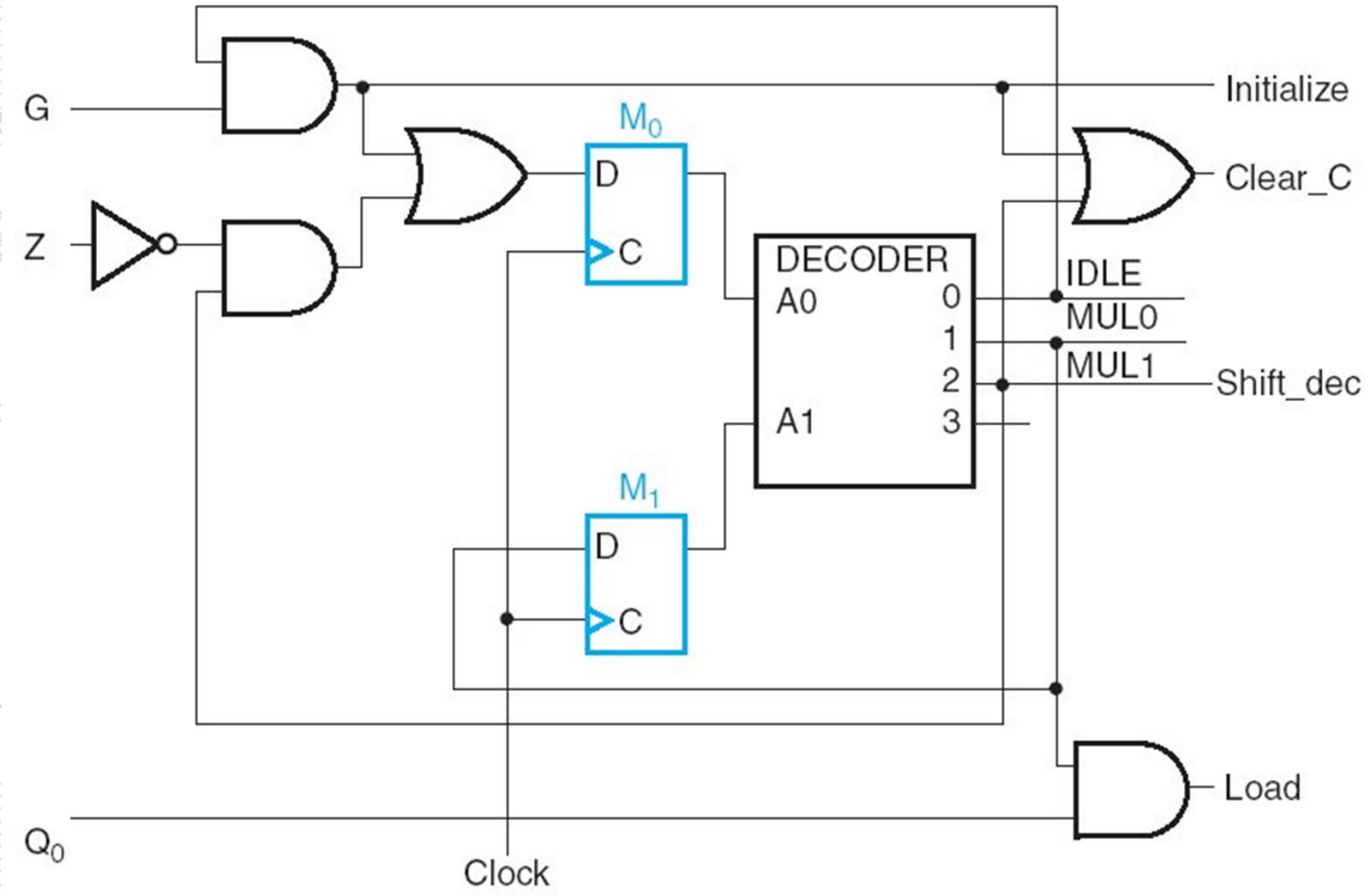
- This method uses:
  - Sequence Register:
    - That holds control states
    - Register with n flop-flops has 2^n states
  - Decoder
    - Provides output signal for each state.
    - An n-to-2^n decoder has 2^n outputs

CS2022 State Table

- Derived from the Sequencing Part of ASM Chart
  - $D_{M0} = IDLE \cdot G + MUL1 \cdot \overline{Z}$
  - $D_{M1} = MUL0$

Present state			Inputs		Next state		Decoder Outputs		
Name	M_1	M_0	G	Z	M_1	M_0	IDLE	MUL0	MUL1
IDLE	0	0	0	×	0	0	1	0	0
	0	0	1	×	0	1	1	0	0
MUL0	0	1	×	×	1	0	0	1	0
MUL1	1	0	×	0	0	1	0	0	1
	1	0	×	1	0	0	0	0	1
—	1	1	×	×	×	×	×	×	×

CS2022 Control Unit for Binary Multiplier

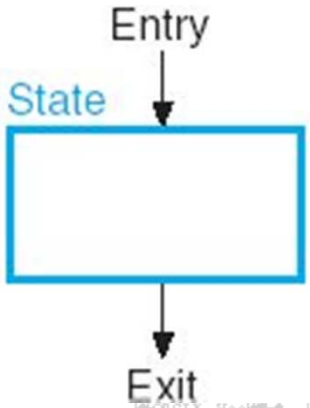


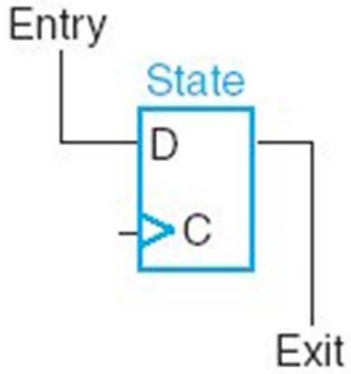
CS2022 One Flip-Flop per State

- Alternative Design
  - A flip-flop is assigned to each state
  - Only one flip-flop may be true
  - Each flip-flop represents a state
- The next four slides give:
  - Symbol substitution rules that:
    - Change an ASM chart into:
      - A sequential circuit with one flip-flop per state.

CS2022 State Box Transformation

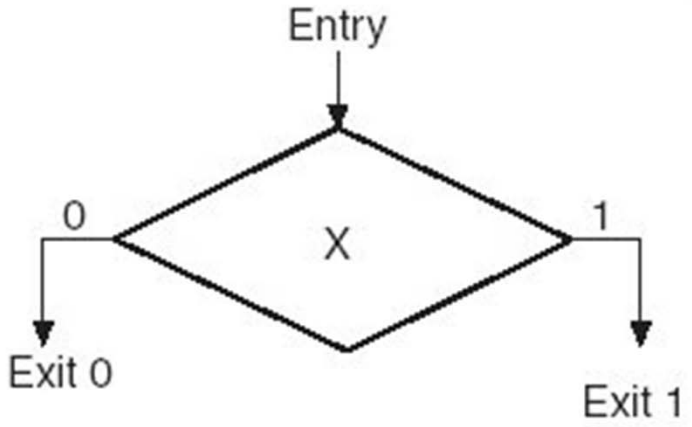
-> D flip-flop

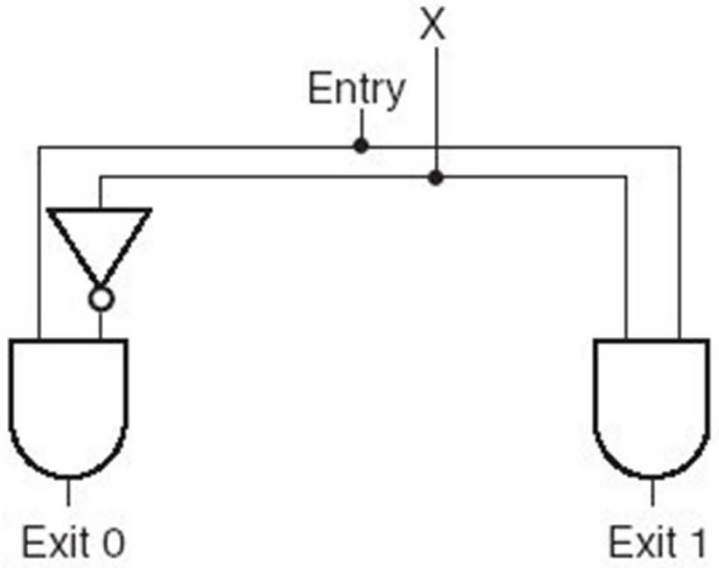




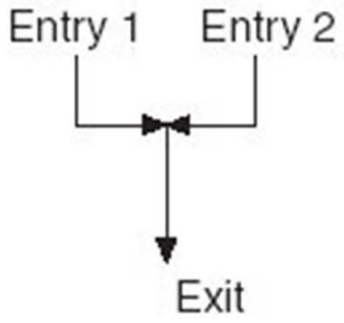
CS2022 Decision Box Transformation

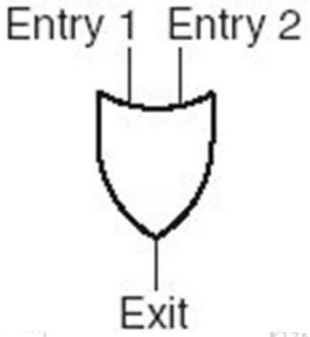
-> Demultiplexer





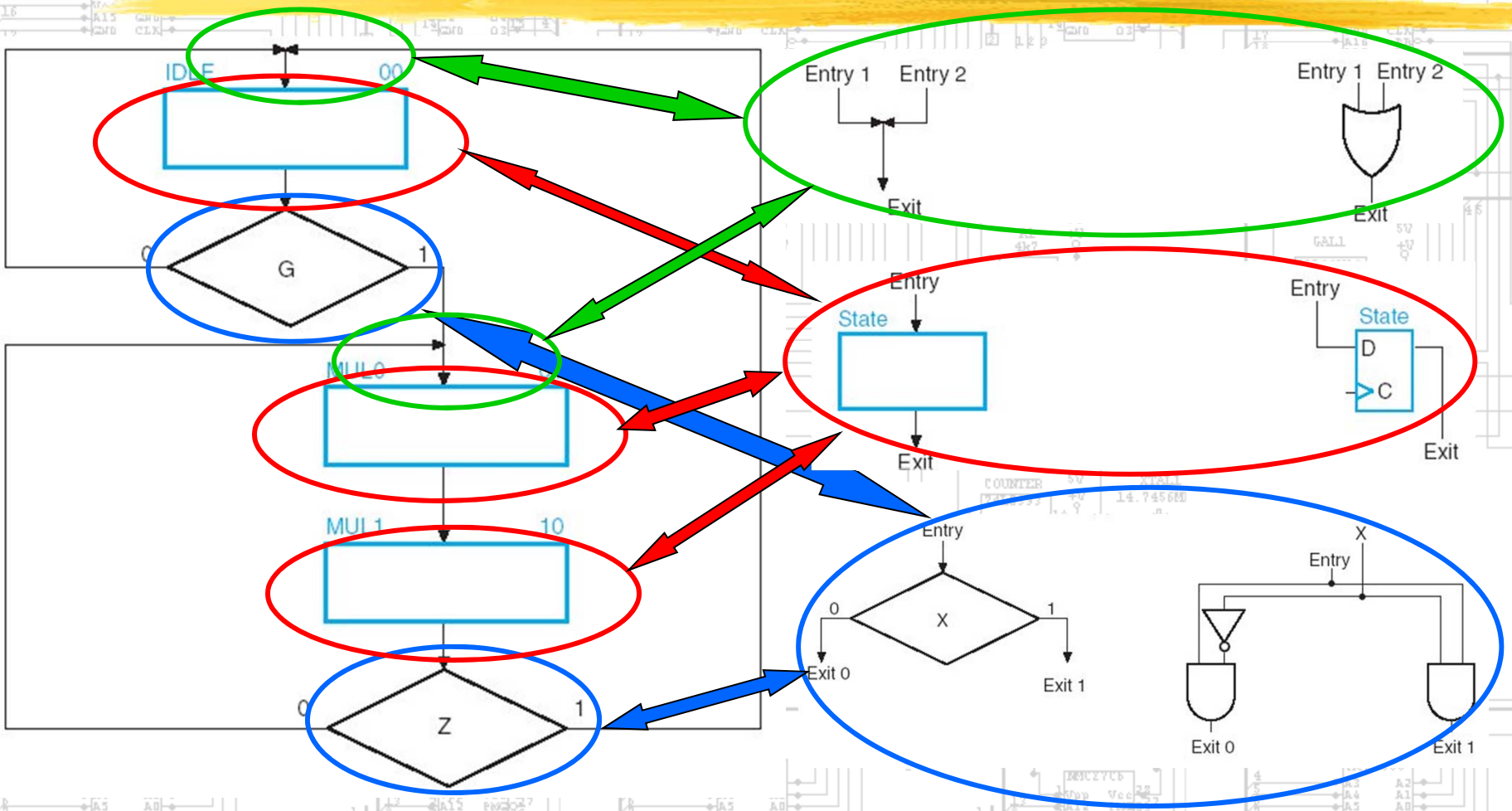
Junction Box Transformation
-> OR gate





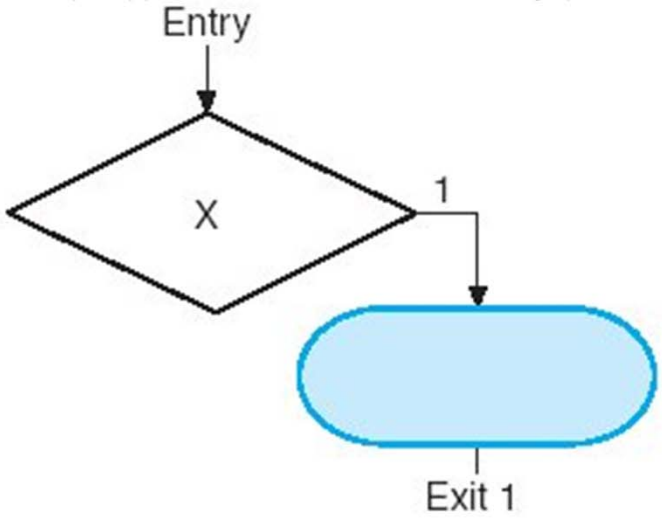
- The previous three transformations may be used to transform the sequencing part of a ASM chart into a circuit with one flip-flop per state

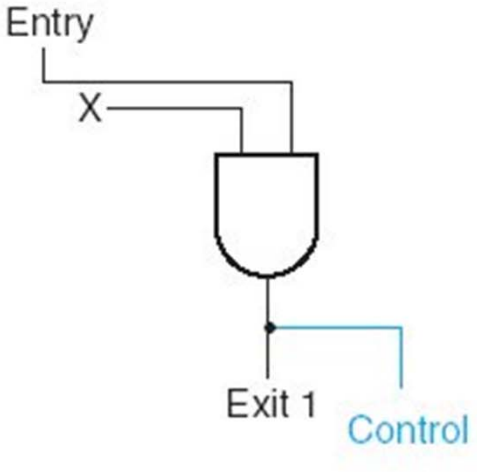
CS2022 Sequencing Part of ASM Chart



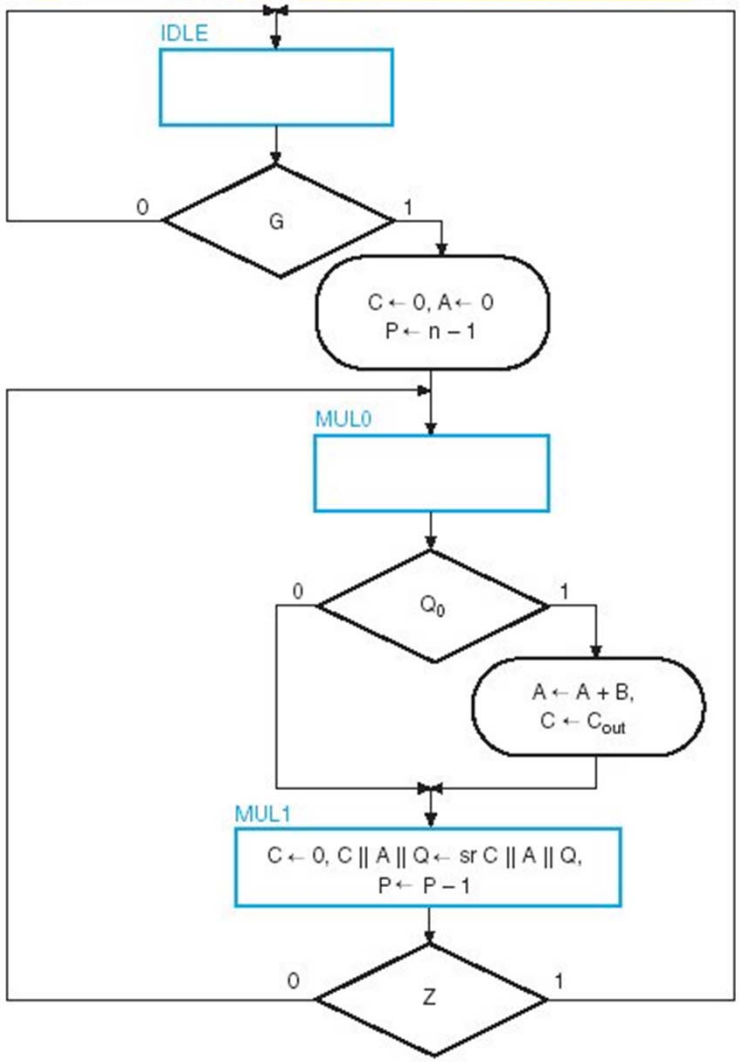
Conditional Output Box Transformation

- Control output is generated by:
  - Attaching Control line in the right location
  - Adding output logic
- The Original ASM is used for the control

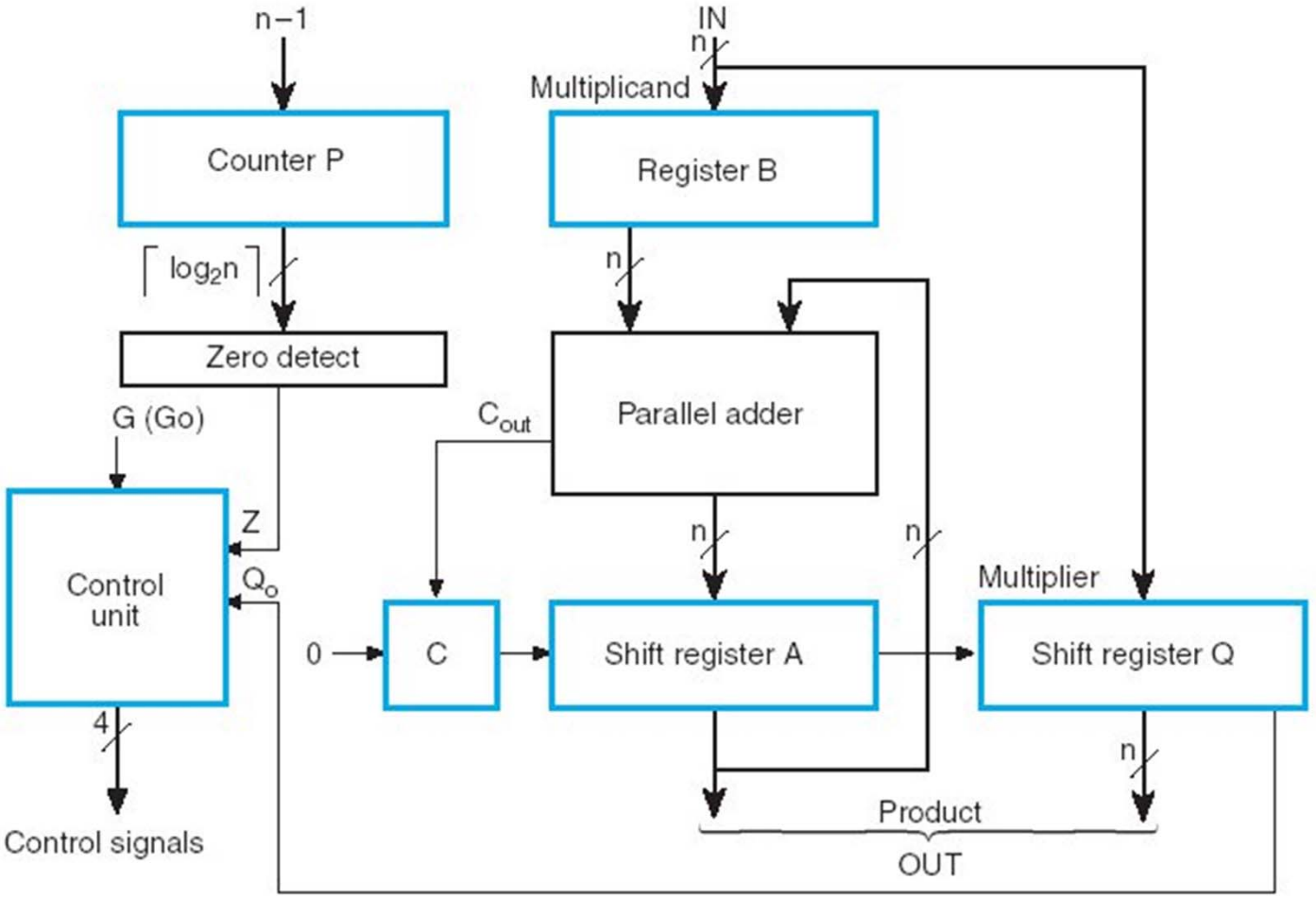




CS2022 Binary Multiplier ASM



CS2022 Binary Multiplier Diagram



CS2022 Transformation

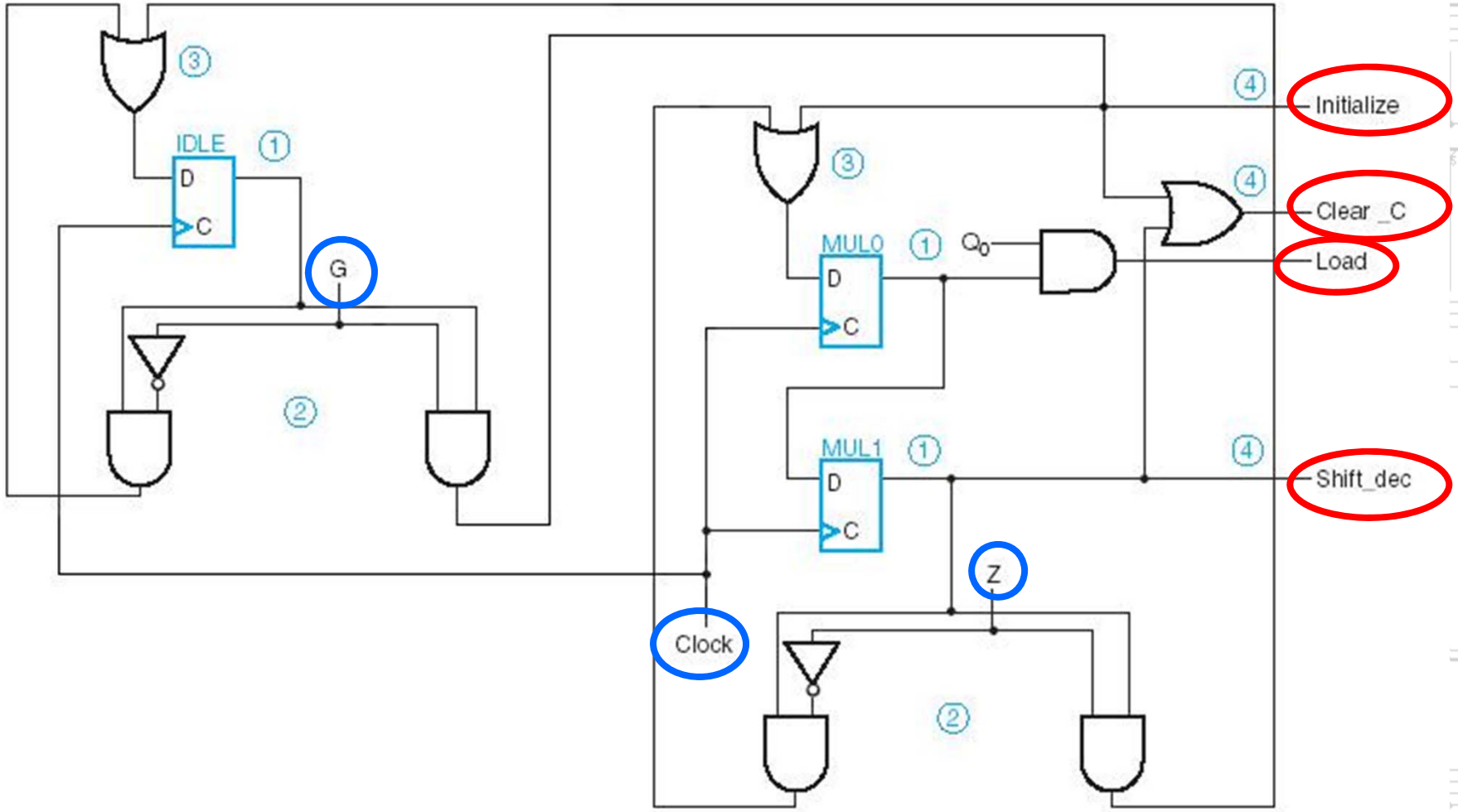
- Replace:
  1. State boxes with D flip-flops
  2. Decision boxes with Demultiplexers
  3. Junctions with OR gates
  4. Add output signals
     - Use table on the following slide

CS2022 Control Signals for Binary Multiplier

Block Diagram Module	Microoperation	Control Signal Name	Control Expression
Register A:	$A \leftarrow 0$	Initialize	$\text{IDLE} \cdot G$
	$A \leftarrow A + B$	Load	$\text{MUL0} \cdot Q_0$
	$C \| A \| Q \leftarrow \text{sr}\ C \| A \| Q$	Shift_dec	MUL1
Register B:	$B \leftarrow IN$	Load_B	LOADB
Flip-Flop C:	$C \leftarrow 0$	Clear_C	$\text{IDLE} \cdot G + \text{MUL1}$
	$C \leftarrow C_{\text{out}}$	Load	—
Register Q:	$Q \leftarrow IN$	Load_Q	LOADQ
	$C \| A \| Q \leftarrow \text{sr}\ C \| A \| Q$	Shift_dec	—
Counter P:	$P \leftarrow n - 1$	Initialize	—
	$P \leftarrow P - 1$	Shift_dec	—

Binary Multiplier Control Unit

One Flip-Flop per State



Binary Multiplier (VHDL)

Entity

```
-- Binary Multiplier with n = 4: VHDL Description
-- See Figures 8-6 and 8-7 for block diagram and ASM Chart
-- in Mano and Kime
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity binary_multiplier is
        port(CLK, RESET, G, LOADB, LOADQ: in std_logic;
                MULT_IN: in std_logic_vector(3 downto 0);
                MULT_OUT: out std_logic_vector(7 downto 0));
end binary_multiplier;
```

Binary Multiplier (VHDL)

architecture

```
architecture behavior_4 of binary_multiplier is
        type state_type is (IDLE, MUL0, MUL1);
        signal state, next_state : state_type;
        signal A, B, Q: std_logic_vector(3 downto 0);
        signal P: std_logic_vector(1 downto 0);
        signal C, Z: std_logic;
begin

        Z <= P(1) NOR P(0);
        MULT_OUT <= A & Q;
```

Binary Multiplier (VHDL)

state_register: process (CLK, RESET)

```
state_register: process (CLK, RESET)
begin
        if (RESET = '1') then
                state <= IDLE;
        elsif (CLK'event and CLK = '1') then
                state <= next_state;
        end if;
end process;
```

Binary Multiplier (VHDL)

next_state_func: process (G, Z, state)

```
next_state_func: process (G, Z, state)
begin
        case state is
                when IDLE =>
                        if G = '1' then
                                next_state <= MUL0;
                        else
                                next_state <= IDLE;
                        end if;
                when MUL0 =>
                        next_state <= MUL1;
                when MUL1 =>
                        if Z = '1' then
                                next_state <= IDLE;
                        else
                                next_state <= MUL0;
                        end if;
        end case;
end process;
```

CS2022 Binary Multiplier (VHDL)

datapath_func: process (CLK) Part 1

```
datapath_func: process (CLK)
variable CA: std_logic_vector(4 downto 0);
begin
        if (CLK'event and CLK = '1') then
                if LOADB = '1' then
                        B <= MULT_IN;
                end if;
                if LOADQ = '1' then
                        Q <= MULT_IN;
                end if;
```

CS2022 Binary Multiplier (VHDL)

datapath_func: process (CLK) Part 2

```
                case state is
                when IDLE =>
                        if G = '1' then
                                C <= '0';
                                A <= "0000";
                                P <= "11";
                        end if;
                when MUL0 =>
                        if Q(0) = '1' then
                                CA := ('0' & A) + ('0' & B);
                        else
                                CA := C & A;
                        end if;
                        C <= CA(4);
                        A <= CA(3 downto 0);
                when MUL1 =>
                        C <= '0';
                        A <= C & A(3 downto 1);
                        Q <= A(0) & Q(3 downto 1);
                        P <= P - "01";
                end case;
                end if;
        end process;
end behavior_4;
```

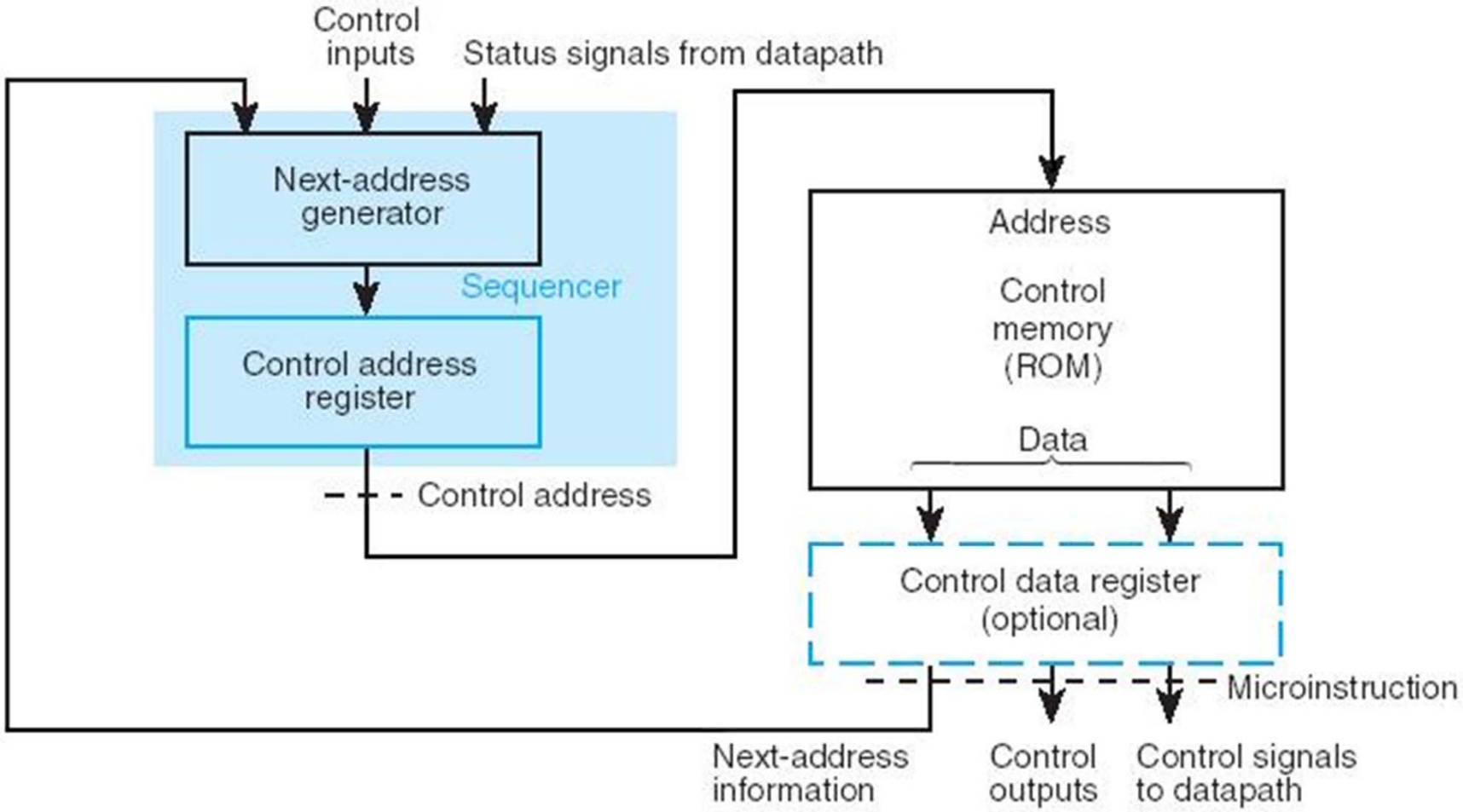
CS2022 Microprogrammed Control

- Hardwired control units have the advantage of great compactness and low propagation delay on account of the shorter loop path.
- However as the number of states and control signals increases they become increasingly costly to:
  - Design
  - Debug
  - Upgrade
- A more flexible approach is used.

CS2022 The Flexible Approach

- We store the control words, together with next-state information , in a control memory which is usually:
  - ROM
  - EPROM
- Then use the control inputs and status signals to select the appropriate address of the next state.
  - See figure on the next slide.

CS2022 Microprogrammed Control Unit Organization

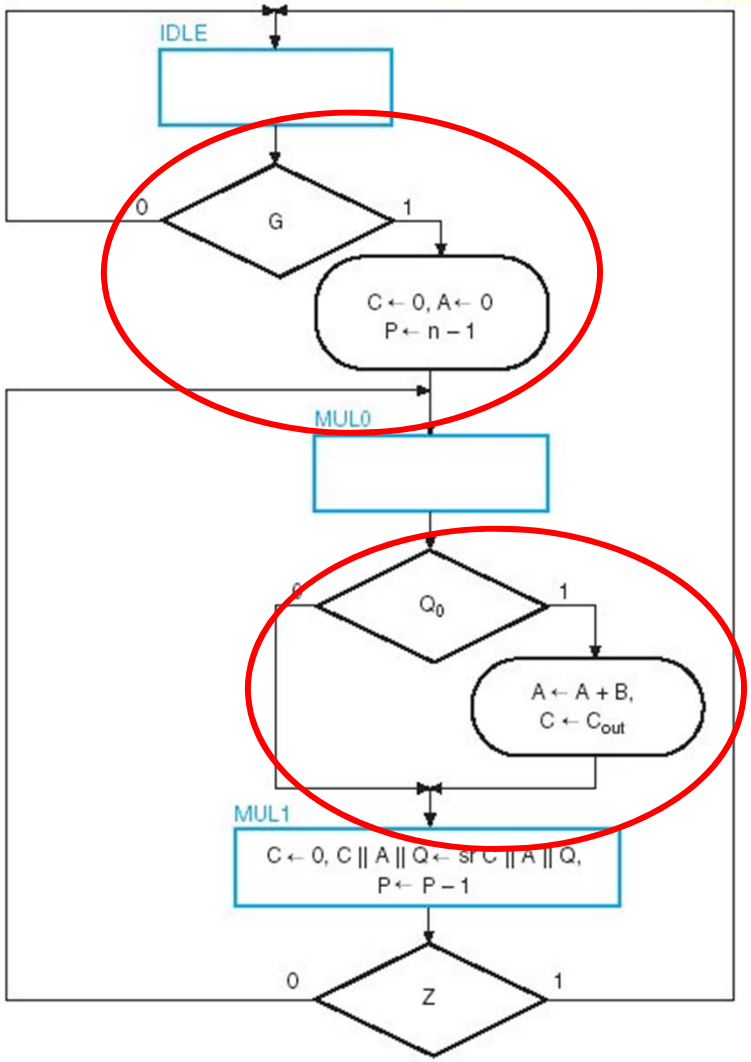


CS2022 Control Input

- One difference which emerges is that since control words are now stored, they cannot be made to depend dynamically on the on the value of the control input

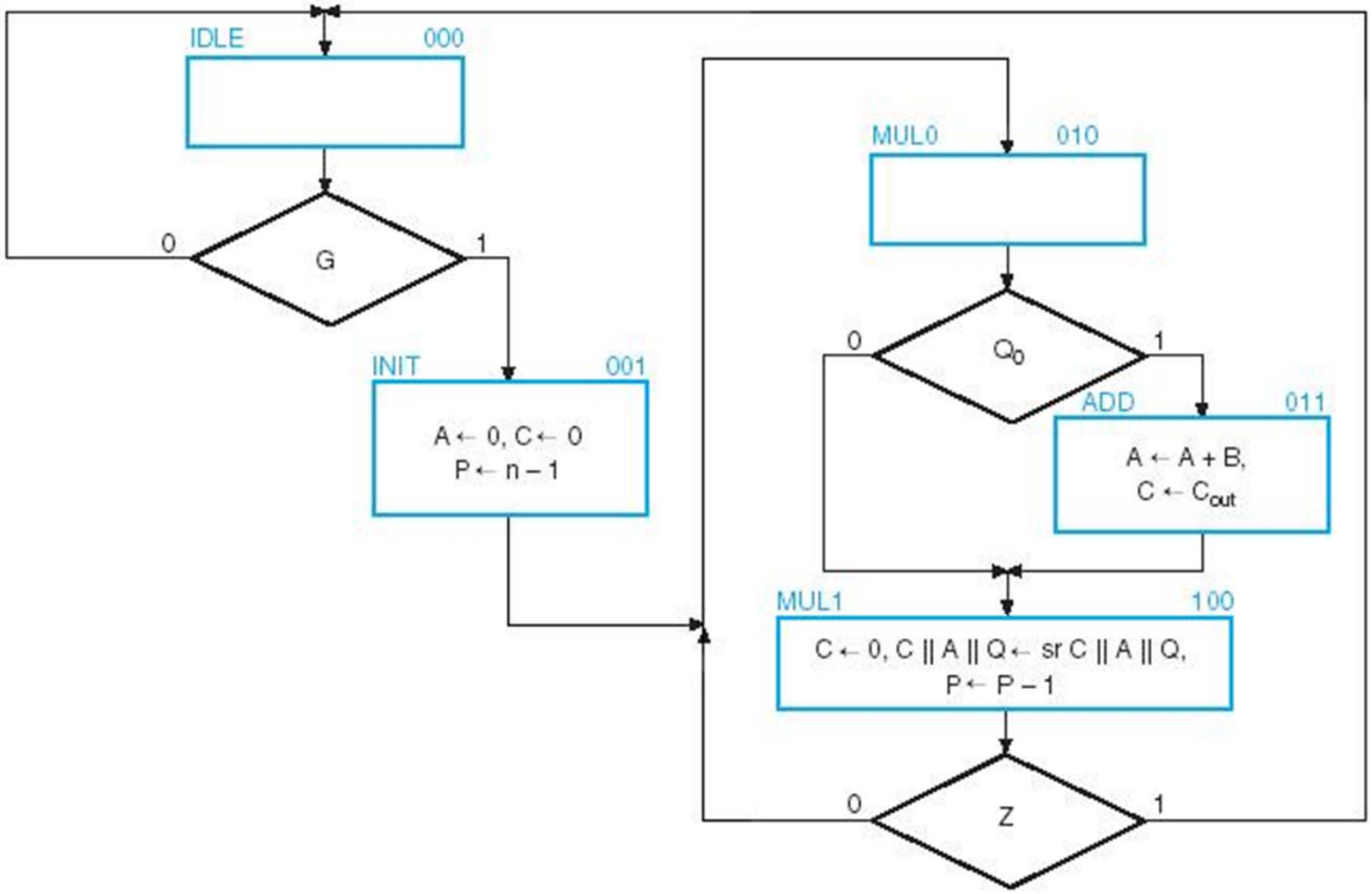
STATE	Control Input	RT	Control Word
IDLE •	G=0	none	CW_1
IDEL •	G=1	C, A ← 0	CW_2
MUL0 •	Q_0=0	none	CW_3
MUL0 •	Q_0=1	A←A+B, C←C_{out}	CW_4
more		more	more

CS2022 Binary Multiplier ASM



- Hence we must introduce additional states to supply these alternative control words.
- See next slide

CS2022 Microprogrammed Control Unit Binary Multiplier



CS2022 Control Address Register (CAR)

- This gives five states so that the control memory must store five control words
  - We will need a 3-bit address register
    - Control Address Register (CAR)

CS2022 SEL bits

- The sequence must be able to respond to two status bits:
  - Z
  - Q_0
- One control input:
  - G
- Hence two SEL bits must be included in the control word.

Next-Address Fields

- Finally we add two next-address fields:
  - NXTADD0
  - NXTADD1
- This design makes no assumption about the sequence control word accesses.
- The functional design of the sequence is as follows on the next slide:

CS2022 SEL Field Definition

SEL		
Symbolic notation	**Binary Code**	**Sequencing Microoperations**
NXT	00	$CAR \leftarrow NXTADD0$
DG	01	$\overline{G}$: $CAR \leftarrow NXTADD0$ G: $CAR \leftarrow NXTADD1$
DQ	10	$\overline{Q_0}$: $CAR \leftarrow NXTADD0$ Q_0: $CAR \leftarrow NXTADD1$
DZ	11	$\overline{Z}$: $CAR \leftarrow NXTADD0$ Z: $CAR \leftarrow NXTADD1$

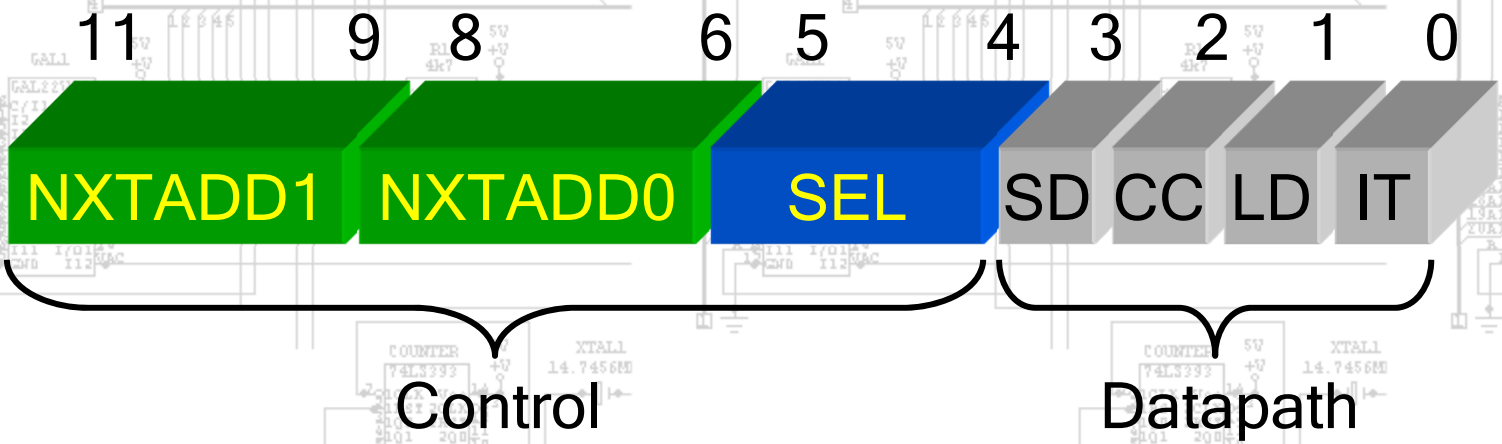
Control Signals for Multiplier control

- The control word must supply four control signals:

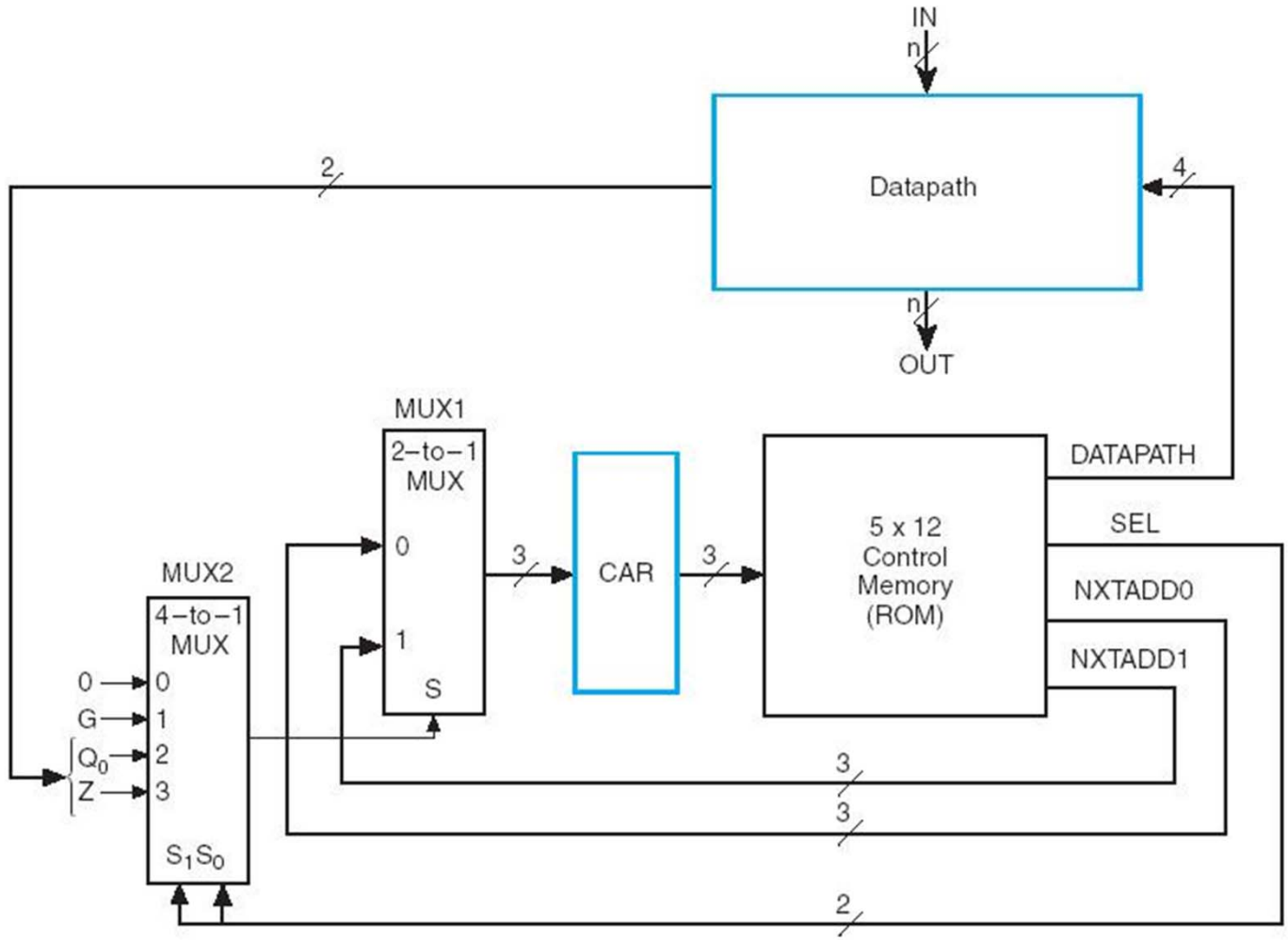
Control Signal	Register Transfers	States in Which Signal is Active	Micro-instruction Bit Position	Symbolic Notation
Initialize	$A \leftarrow 0, P \leftarrow n-1$	INIT	0	IT
Load	$A \leftarrow A+B, C \leftarrow C_{out}$	ADD	1	LD
Clear_C	$C \leftarrow 0$	INIT, MUL1	2	CC
Shift_dec	$C \| A \| Q \leftarrow \text{sr } C \| A \| Q, P \leftarrow P-1$	MUL1	3	SD

CS2022 Control Word

- This results in a 12-bit control word:



CS2022 Control Unit



CS2022 Register Transfer Description

- Next we design the microprogram in symbolic RT form:

Address	Symbolic transfer statement
IDLE	$G: CAR \leftarrow \text{INIT}, \overline{G}: CAR \leftarrow \text{IDLE}$
INIT	$C \leftarrow 0, A \leftarrow 0, P \leftarrow n-1, CAR \leftarrow \text{MUL0}$
MUL0	$Q_0: CAR \leftarrow \text{ADD}, \overline{Q_0}: CAR \leftarrow \text{MUL1}$
ADD	$A \leftarrow A+B, C \leftarrow C_{\text{out}}, CAR \leftarrow \text{MUL1}$
MUL1	$C \leftarrow 0, C \Vert A \Vert Q \leftarrow \text{sr } C \Vert A \Vert Q, Z: CAR \leftarrow \text{IDLE}, \overline{Z}: CAR \leftarrow \text{MUL0},$ $P \leftarrow P-1$

CS2022

Symbolic Microprogram & Binary Microprogram

Address	NXTADD1	NXTADD0	SEL	DATAPATH	Address	NXTADD1	NXTADD0	SEL	DATAPATH
IDLE	INIT	IDLE	DG	None	000	001	000	01	0000
INIT	—	MUL0	NXT	IT, CC	001	000	010	00	0101
MUL0	ADD	MUL1	DQ	None	010	011	100	10	0000
ADD	—	MUL1	NXT	LD	011	000	100	00	0010
MUL1	IDLE	MUL0	DZ	CC, SD	100	000	010	11	1100

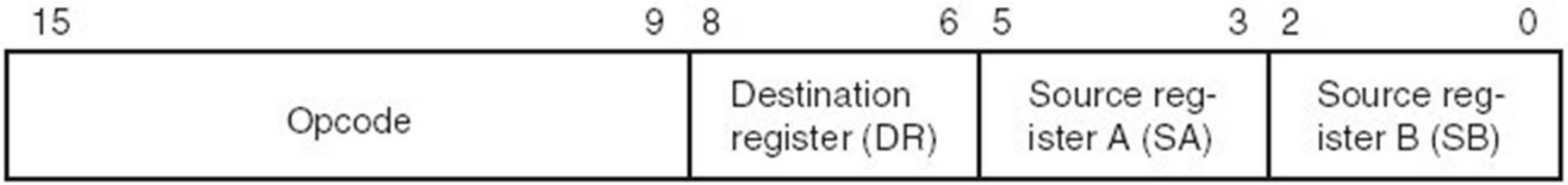
CS2022 Basic Computer Architecture

- Computers consist of:
  - Datapath
  - Control unit
- It is designed to implement a particular instruction set.
- The individual instructions are the engineering equivalent of the mathematician's
  - **z=f(x,y)**

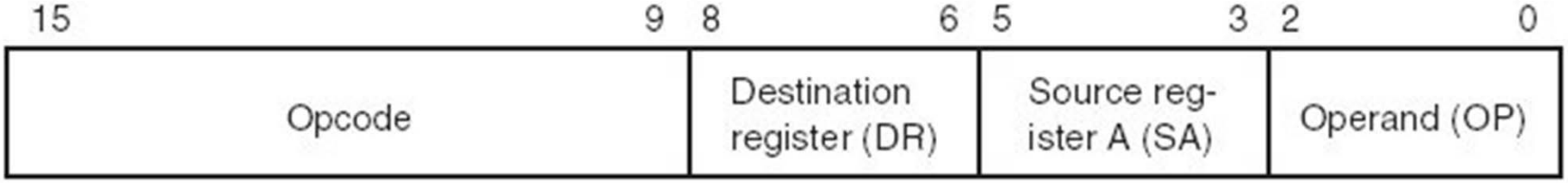
Opcode – Destination -Operands

- OPCODE
  - Selects the function
- DESTINATION
  - Is nearly always a datapath register
- OPERANDS
  - Usually come from datapath register

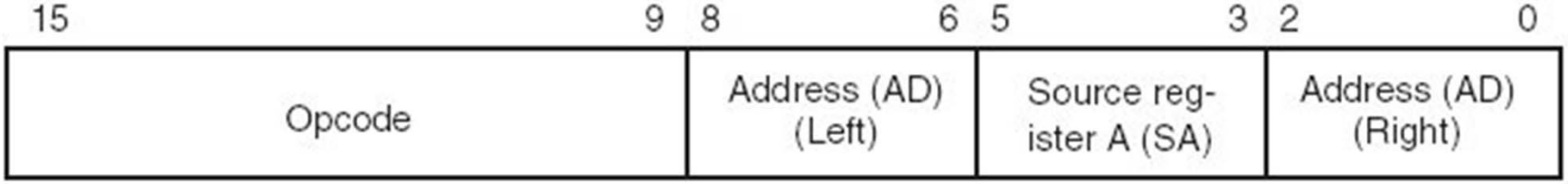
CS2022 Instruction Format Examples



(a) Register



(b) Immediate



(c) Jump and Branch

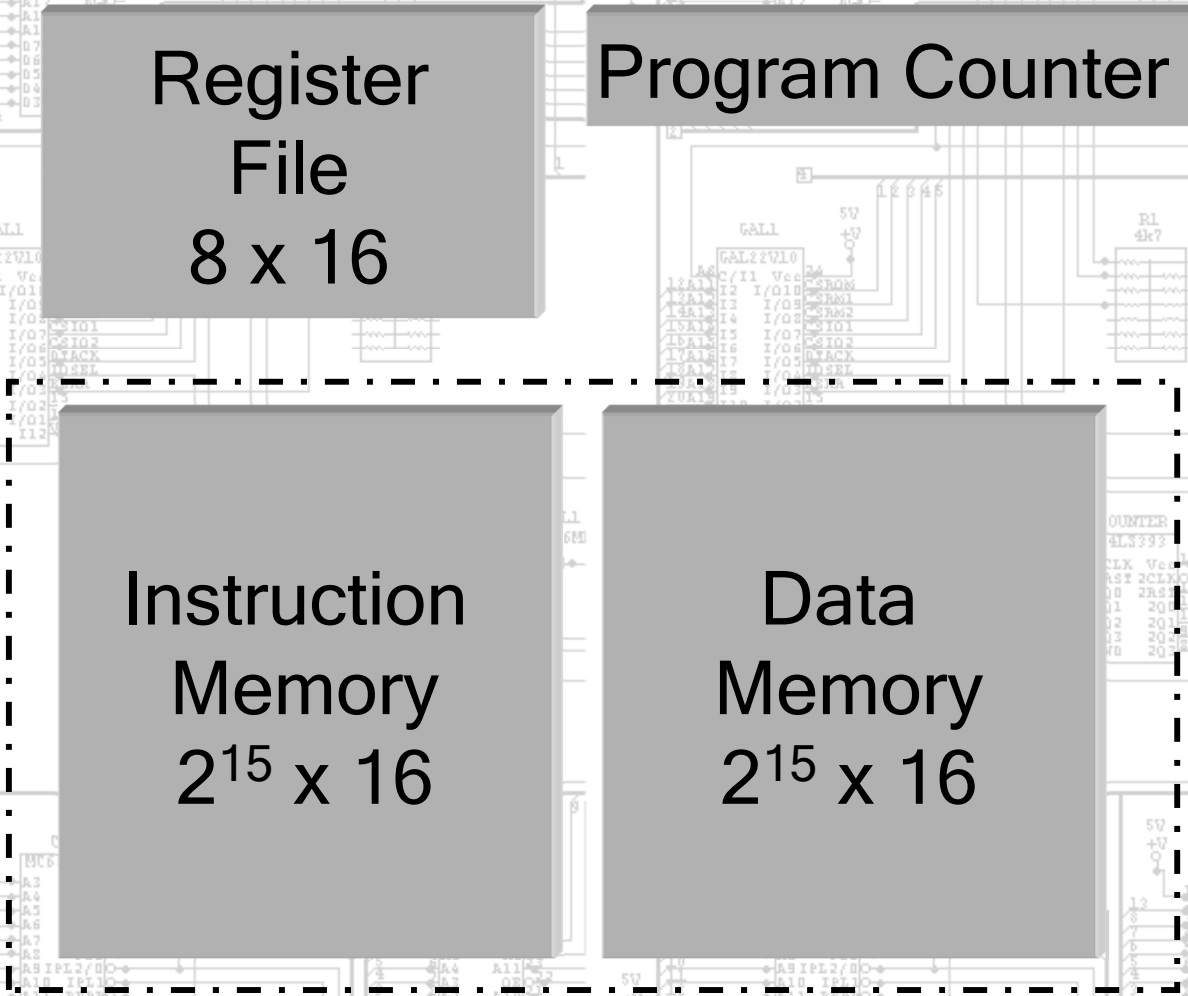
CS2022 Instruction Formats

- Where DR, SA $\wedge$ SB point to processor registers in the datapath
- But Operand is itself an immediate operand

Data and Instructions in Memory

Decimal address	Memory contents	Decimal opcode	Other specified fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2 SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4 SB:5	M [R4] ← R5
45	1000010 010 111 011	66 (Add Immedi-ate)	DR:2 SA:7 OP:3	R2 ← R7 + 3
55	1100011 101 110 100	96 (Branch on zero)	AD: 44 SA:6	If R6 = 0, PC ← PC – 20
70	0000000 011 000 000	Data = 192. After execution of instruction in 35, Data = 80.		

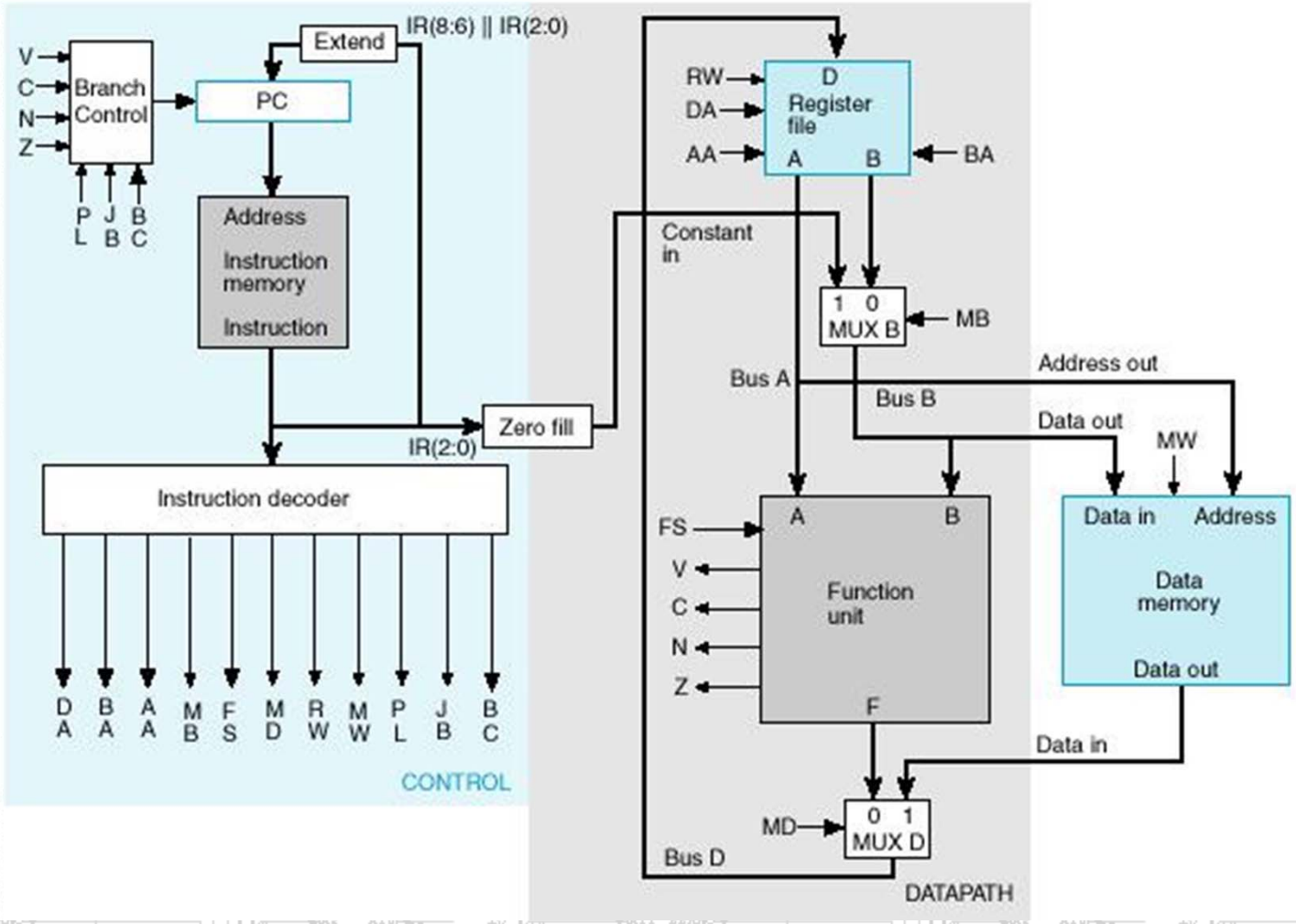
CS2022 User View of Storage



CS2022 A Single-cycle Hardwired Control Unit

- We briefly consider a system with the simplest possible control unit.
- The control unit:
  - Maps each OPCODE to a single datapath operation.
- Instructions are fetched from an instruction memory
- This is what all present systems with separate instruction and data code do.

CS2022 Single-Cycle Computer



CS2022 Memory Module [entity]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity memory is -- use unsigned for memory address
    Port ( address : in unsigned std_logic_vector(31 downto 0);
           write_data : in std_logic_vector(31 downto 0);
           MemWrite, MemRead : in std_logic;
           read_data : out std_logic_vector(31 downto 0));
end memory;
```

CS2022 Memory Module [architecture]

```
architecture Behavioral of memory is
type mem_array is array(0 to 7) of std_logic_vector(31 downto 0);
-- define type, for memory arrays
begin
mem_process: process (address, write_data)
-- initialize data memory, X denotes hexadecimal number
variable data_mem : mem_array := (
X"00000000", X"00000000", X"00000000",X"00000000",
X"00000000", X"00000000", X"00000000",X"00000000");
variable addr:integer
begin  -- the following type conversion function is in std_logic_arith
addr:=conv_integer(address(2 downto 0));
if MemWrite ='1' then
data_mem(addr):= write_data;
elsif MemRead='1' then
read_data <= data_mem(addr) after 10 ns;
end if;
end process;
end Behavioral;
```

CS2022 Branch

- Instruction Bit13=1
  - Jump occurs
- Instruction Bit13=0
  - Conditional Branch occurs
  - Bit11, Bit10 and Bit9
    - Select the status bit

BC	Status Bit
000	C
001	N
010	V
011	Z
100	$\overline{C}$
101	$\overline{N}$
110	$\overline{V}$
111	$\overline{Z}$

CS2022 Jump and Branch

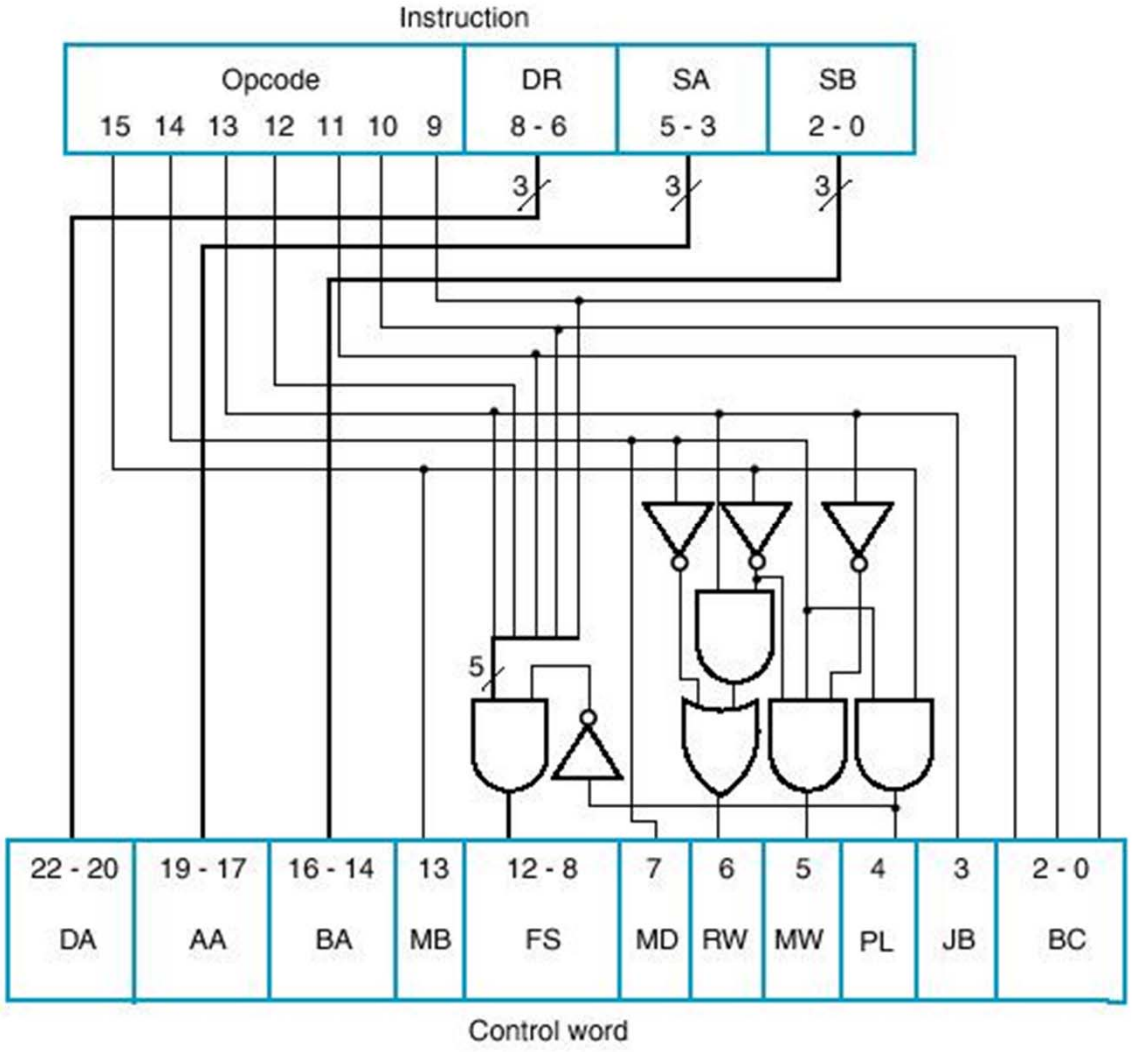
- PL=1
  - Jump or Branch, loading the PC
- PL=0
  - PC is incremented
- PL=1 $\wedge$ JB=0
  - Jump
- PL=1 $\wedge$ JB=1
  - Conditional branch

CS2022 Truth Table BIT15 – BIT13

- The following operations classification helps with the implementation of the instruction decoder

	Instruction Bits			Control Word Bits					
Instruction Function Type	**Bit 15**	**Bit 14**	**Bit 13**	**MB**	**MD**	**RW**	**MW**	**PL**	**JB**
ALU function using registers	0	0	0	0	0	1	0	0	X
Shifter function using registers	0	0	1	0	0	1	0	0	X
Memory write using register data	0	1	0	0	X	0	1	0	X
Memory read using register data	0	1	1	0	1	1	0	0	X
ALU operation using a constant	1	0	0	1	0	1	0	0	X
Shifter function using a constant	1	0	1	1	0	1	0	0	X
Conditional Branch	1	1	0	X	X	0	0	1	0
Unconditional Jump	1	1	1	X	X	0	0	1	1

CS2022 Instruction Decoder



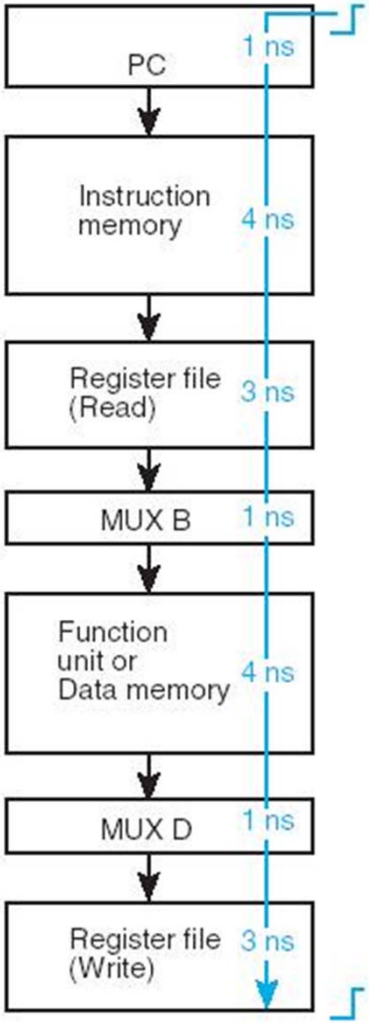
CS2022 Single-Cycle Computer Instruction Example

Operation code	Symbolic name	Format	Description	Function	MB	MD	RW	MW	PL	JB
1000010	ADI	Immediate	Add immediate operand	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + \mathrm{zf}\ I(2{:}0)$	1	0	1	0	0	0
0110000	LD	Register	Load memory content into register	$R[\mathrm{DR}] \leftarrow M[R[\mathrm{SA}]]$	0	1	1	0	0	1
0100000	ST	Register	Store register content in memory	$M[R[\mathrm{SA}]] \leftarrow R[\mathrm{SB}]$	0	1	0	1	0	0
0011000	SL	Register	Shift left	$R[\mathrm{DR}] \leftarrow \mathrm{sl}R[\mathrm{SB}]$	0	0	1	0	0	1
0001110	NOT	Register	Complement register	$R[\mathrm{DR}] \leftarrow \overline{R[\mathrm{SA}]}$	0	0	1	0	0	0
1100000	BRZ	Jump/Branch	If $R[\mathrm{SA}] = 0$, branch to PC + se AD	If $R[\mathrm{SA}] = 0, PC \leftarrow PC + \mathrm{se}AD$, If $R[\mathrm{SA}] \neq 0, PC \leftarrow PC + 1$	1	0	0	0	1	0

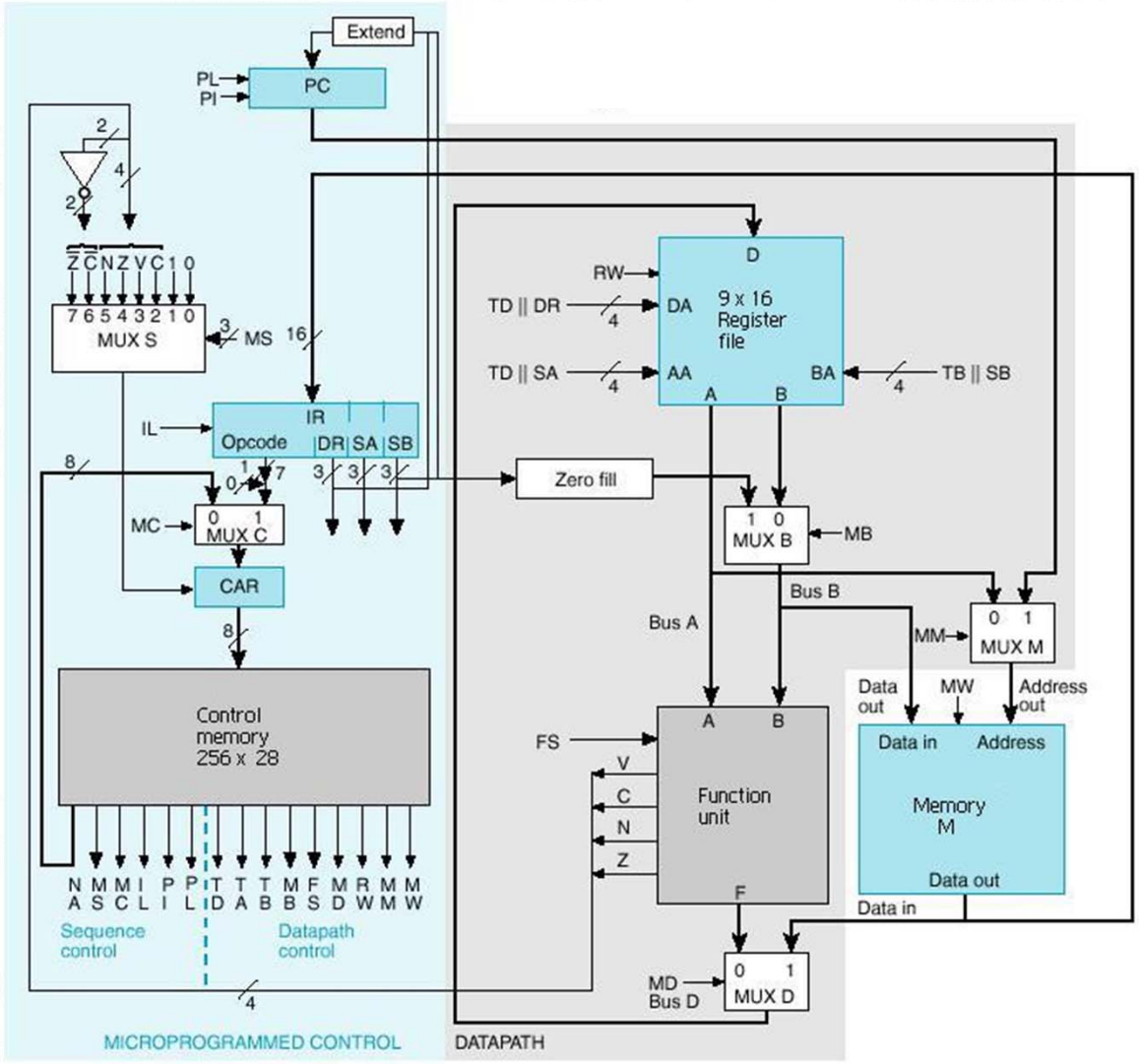
CS2022 Single-cycle Problem

- A single-cycle control unit cannot implement:
  - more complex addressing modes
  - Composite functions
    - E.g. Multiplication
- A single-cycle control unit has long worst case delay path.
  - Slow clock.

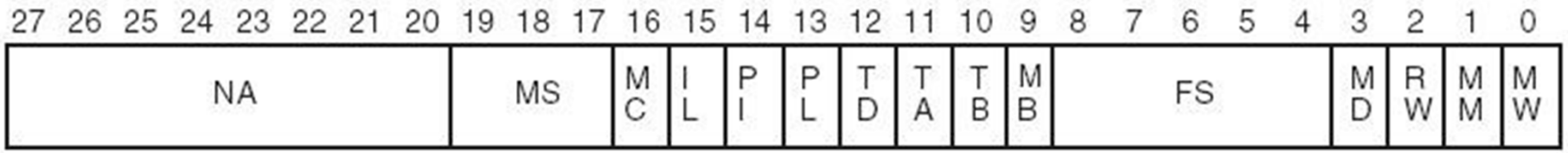
CS2022 Worst Case Delay



CS2022 Multiple-Cycle Microprogrammed Computer



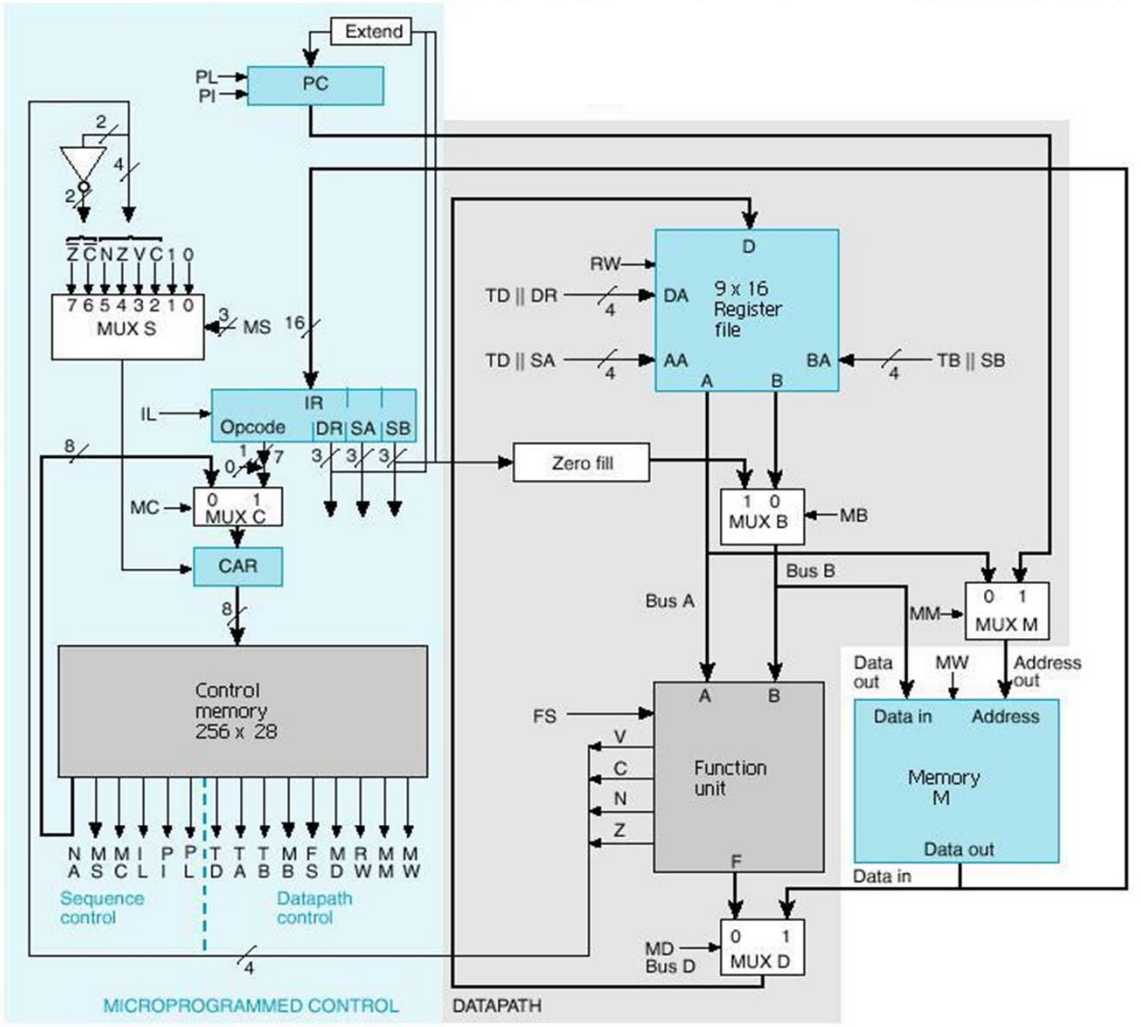
CS2022 Microinstruction Format



Control Word Information for Datapath

TD	TA	TB	MB		FS		MD	RW	MM	MW	
Select	Select	Select	Select	Code	Function	Code	Select	Function	Select	Function	Code
R[DR]	R[SA]	R[SB]	Register	0	$F = A$	00000	FnUt	No write (NW)	Address	No write (NW)	0
$R8$	$R8$	$R8$	Constant	1	$F = A + 1$	00001	Data In	Write (WR)	PC	Write (WR)	1
					$F = A + B$	00010					
					$F = A + B + 1$	00011					
					$F = A + \overline{B}$	00100					
					$F = A + \overline{B} + 1$	00101					
					$F = A - 1$	00110					
					$F = A$	00111					
					$F = A \wedge B$	01000					
					$F = A \vee B$	01010					
					$F = A \oplus B$	01100					
					$F = \overline{A}$	01110					
					$F = B$	10000					
					$F = \text{sr } B$	10100					
					$F = \text{sl } B$	11000					

CS2022 Multiple-Cycle Microprogrammed Computer



CS2022 Project 2
Microcoded Instruction Set Processor

- Project 2 in incremental steps
- modifications are required for tomorrow:
  - Increase the number of registers in the register-file from 8 to 9
    - This requires an additional select bit for the two multiplexers (Bus A and Bus B) and the destination decoder. These are separate signals (TD, TA, TB) that are provided by the Control Memory
    - The size of the registers in the register-file has to be increased to 16bit (size of instructions)

CS2022 Datapath Modifications

- Consequently all components of the Datapath:
  - MUXs in the Register file
  - Decoder in the Register file
  - Arithmetic/logic Unit
  - Shifter and MUXs …
- have to be adjusted to 16bit operations.

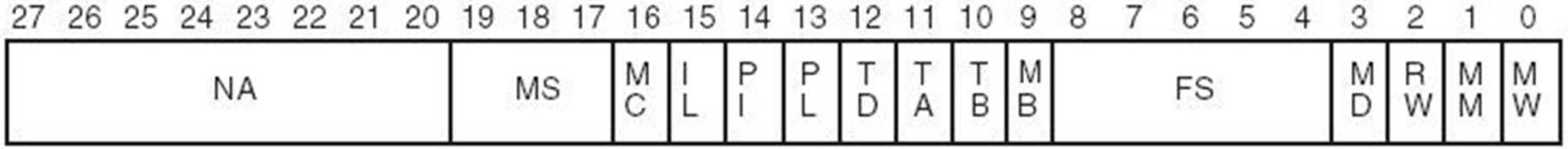
CS2022 Datapath Modifications

- Add and test:
  - Memory M (512 x 16)
  - Control Memory (256 x 28)
- to your project.
  - MUX M will feed 16 bit addresses from ether the Bus A or the PC into the Memory M entity but only the 9 least significant address bits will be used to index into the array. This restricts the memory size to 512.

CS2022

Control Memory 256 x 28

library IEEE




```
-- Michael Manzke
-- michael.manzke@cs.tcd.ie
-- 25 April 2003

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

CS2022 entity control_memory

```
entity control_memory is
    Port ( MW : out std_logic;
           MM : out std_logic;
           RW : out std_logic;
           MD : out std_logic;
           FS : out std_logic_vector(4 downto 0);
           MB : out std_logic;
           TB : out std_logic;
           TA : out std_logic;
           TD : out std_logic;
           PL : out std_logic;
           PI : out std_logic;
           IL : out std_logic;
           MC : out std_logic;
           MS : out std_logic_vector(2 downto 0);
           NA : out std_logic_vector(7 downto 0);
           IN_CAR : in std_logic_vector(7 downto 0));
end control_memory;
```

architecture Behavioral of control_memory is

```
architecture Behavioral of control_memory is
type mem_array is array(0 to 255) of std_logic_vector(27 downto 0);

begin

memory_m: process(IN_CAR)
        variable control_mem : mem_array:=(
                                        -- 0
                                        X"FFFFFFF", -- 0
                                        X"0000000", -- 1
                                        X"AAAAAAA", -- 2
                                        X"0000000", -- 3
                                        X"BBBBBBB", -- 4
                                        X"0000000", -- 5
                                        X"CCCCCCC", -- 6
                                        X"0000000", -- 7
```

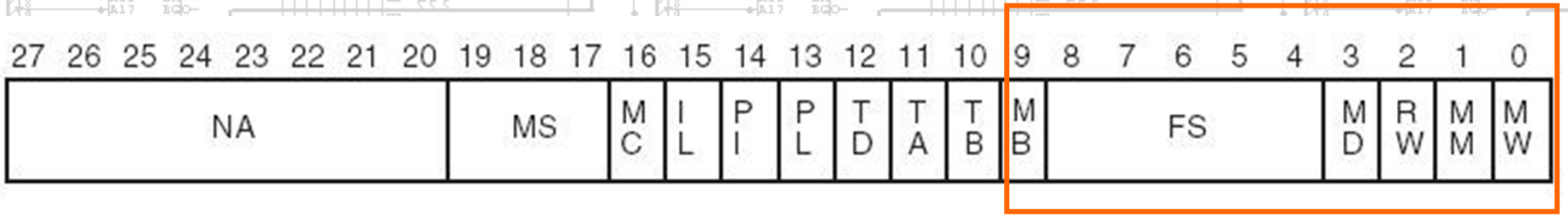
CS2022 -- Address $08 to $17

```
X"DDDDDDD", -- 8
X"0000000", -- 9
X"1111111", -- A
X"0000000", -- B
X"2222222", -- C
X"0000000", -- D
X"3333333", -- E
X"0000000", -- F
-- 1
X"0000000", -- 0
X"0000000", -- 1
X"0000000", -- 2
X"0000000", -- 3
X"0000000", -- 4
X"0000000", -- 5
X"0000000", -- 6
X"0000000", -- 7
```

CS2022 -- Address $F8 to $FF

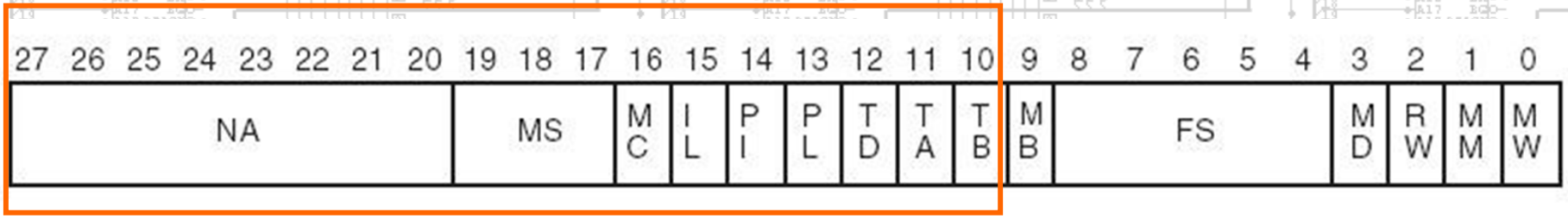
```
                •
                •
                •
        X"0000000", -- 8
        X"0000000", -- 9
        X"0000000", -- A
        X"0000000", -- B
        X"0000000", -- C
        X"0000000", -- D
        X"0000000", -- E
        X"0000000"  -- F
        );
variable addr : integer;
variable control_out : std_logic_vector(27 downto 0);
```

CS2022 Begin (process) LSB




```
begin
        addr := conv_integer(IN_CAR);
        control_out := control_mem(addr);
        MW <= control_out(0);
        MM <= control_out(1);
        RW <= control_out(2);
        MD <= control_out(3);
        FS <= control_out(8 downto 4);
        MB <= control_out(9);
```

Begin (process) MSB




```
                TB <= control_out(10);
                TA <= control_out(11);
                TD <= control_out(12);
                PL <= control_out(13);
                PI <= control_out(14);
                IL <= control_out(15);
                MC <= control_out(16);
                MS <= control_out(19 downto 17);
                NA <= control_out(27 downto 20);
                end process;
end Behavioral;
```

CS2022 Control Word for Datapath

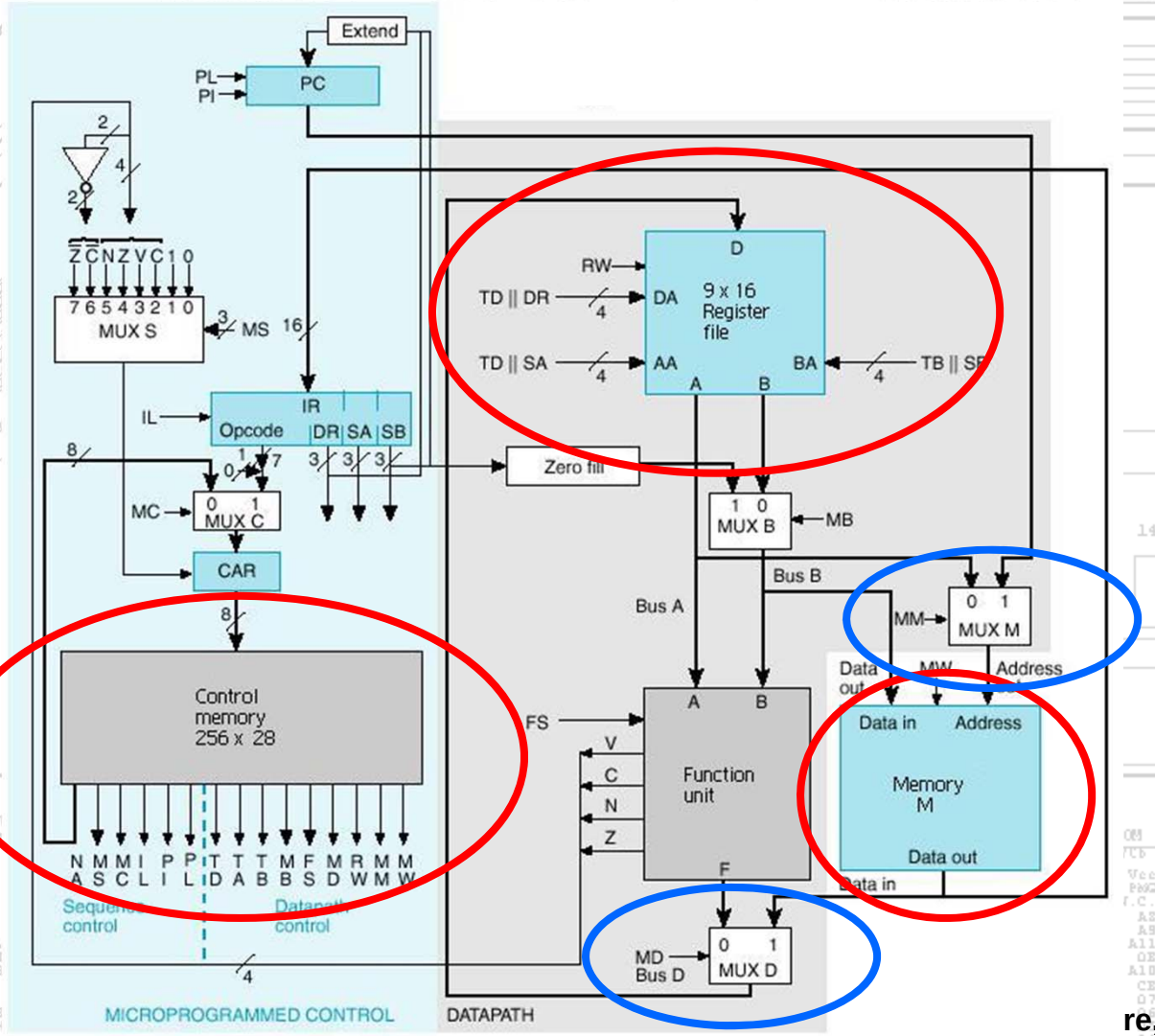
- The Datapath should have the following functionality:

TD	TA	TB	MB		FS		MD	RW	MM	MW	
Select	Select	Select	Select	Code	Function	Code	Select	Function	Select	Function	Code
R[DR]	R[SA]	R[SB]	Register	0	$F = A$	00000	FnUt	No write (NW)	Address	No write (NW)	0
R8	R8	R8	Constant	1	$F = A + 1$	00001	Data In	Write (WR)	PC	Write (WR)	1
					$F = A + B$	00010					
					$F = A + B + 1$	00011					
					$F = A + \overline{B}$	00100					
					$F = A + \overline{B} + 1$	00101					
					$F = A - 1$	00110					
					$F = A$	00111					
					$F = A \wedge B$	01000					
					$F = A \vee B$	01010					
					$F = A \oplus B$	01100					
					$F = \overline{A}$	01110					
					$F = B$	10000					
					$F = \text{sr } B$	10100					
					$F = \text{sl } B$	11000					

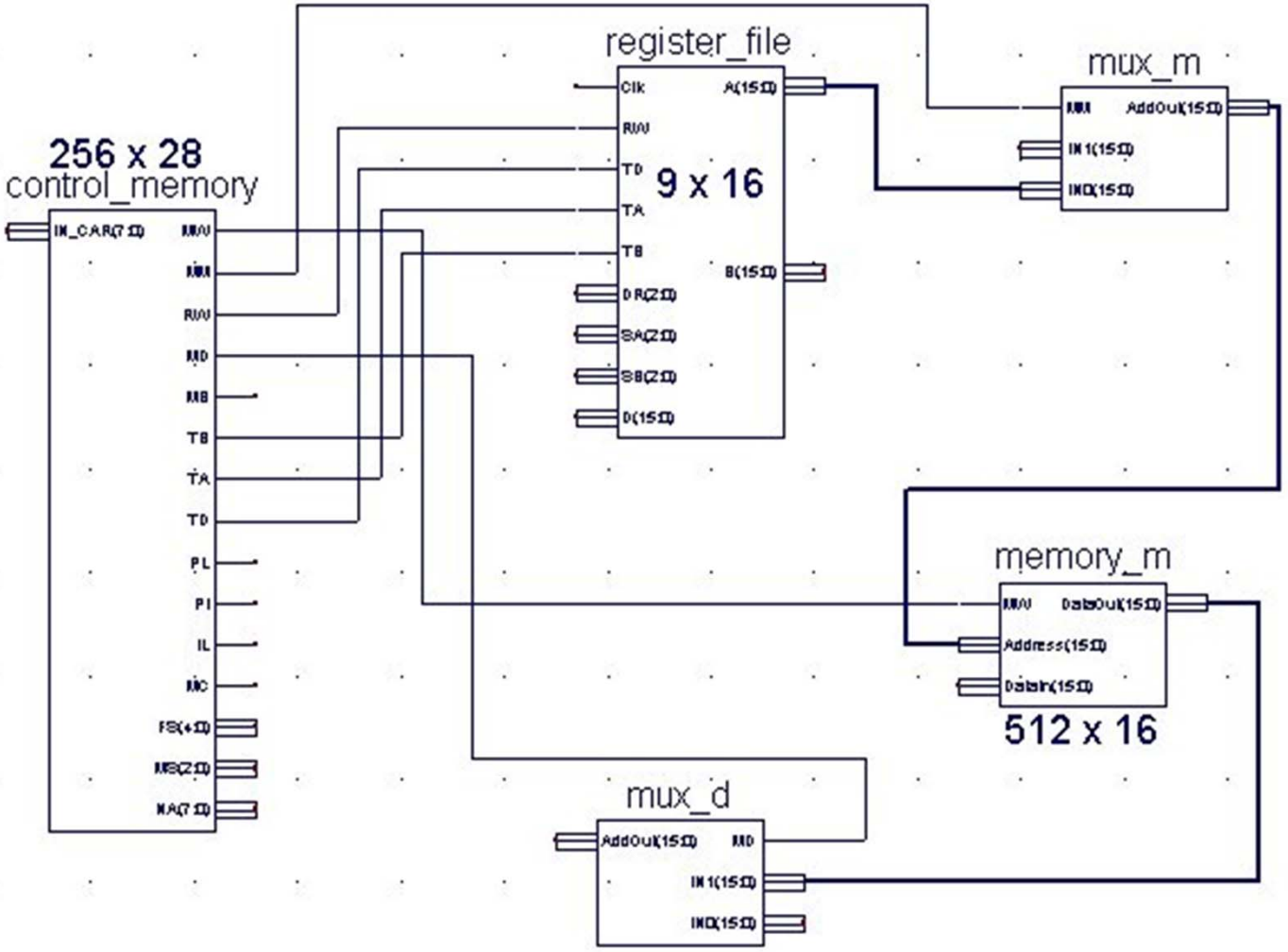
CS2022 VHDL top-level models

- The Modified register-file
- The Functional Unit
- The two memories
  - must be implemented as VHDL top-level models and symbols must be created for these components. A schematic should then interconnect these symbols.

CS2022 Block Diagram



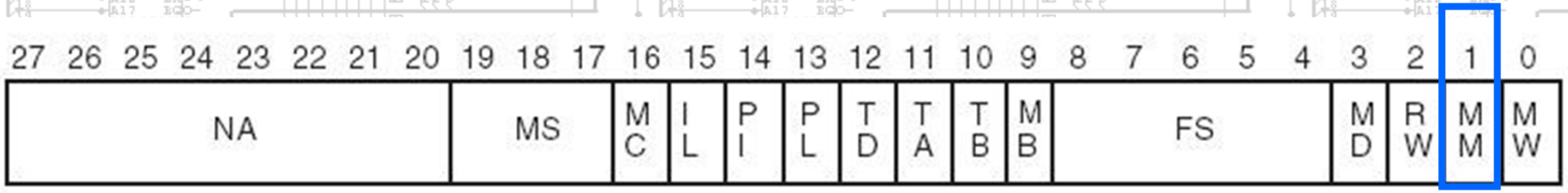
CS2022 Incomplete Schematic



CS2022 Multiple-Cycle Design

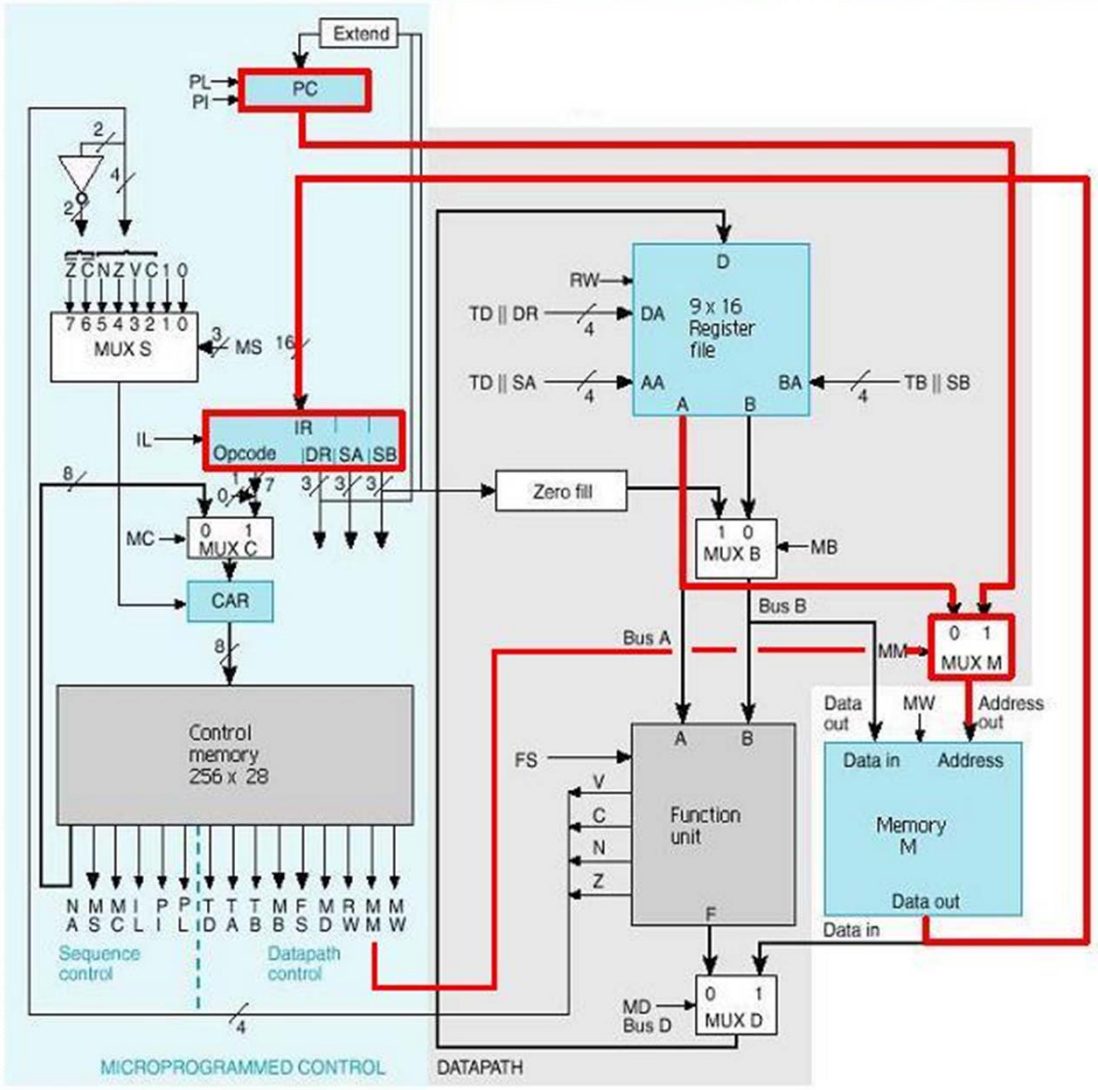
- The Multiple-Cycle Implementation demonstrates the use of a single memory for:
  - Data
  - Instruction
- This design is also used to show the implementation of more complex instructions

CS2022 Memory M Address

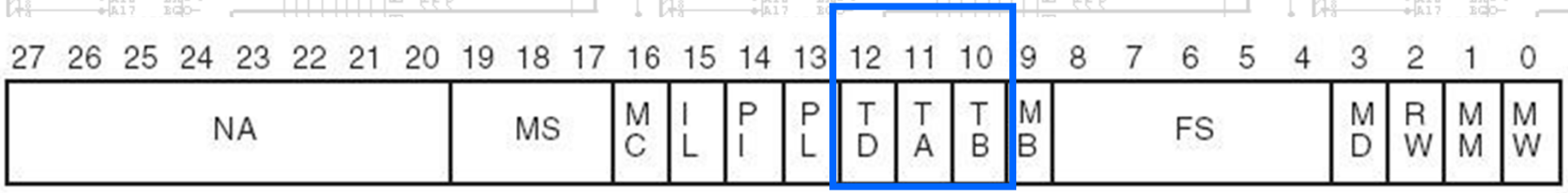


- The following address sources are used to fetch:
  - Instructions -> PC Program Counter Register (16bit)
  - Data -> Bus A (16bit)
- MUX M selects between the two address sources through the MM control signal

CS2022 PC - Bus A - MM

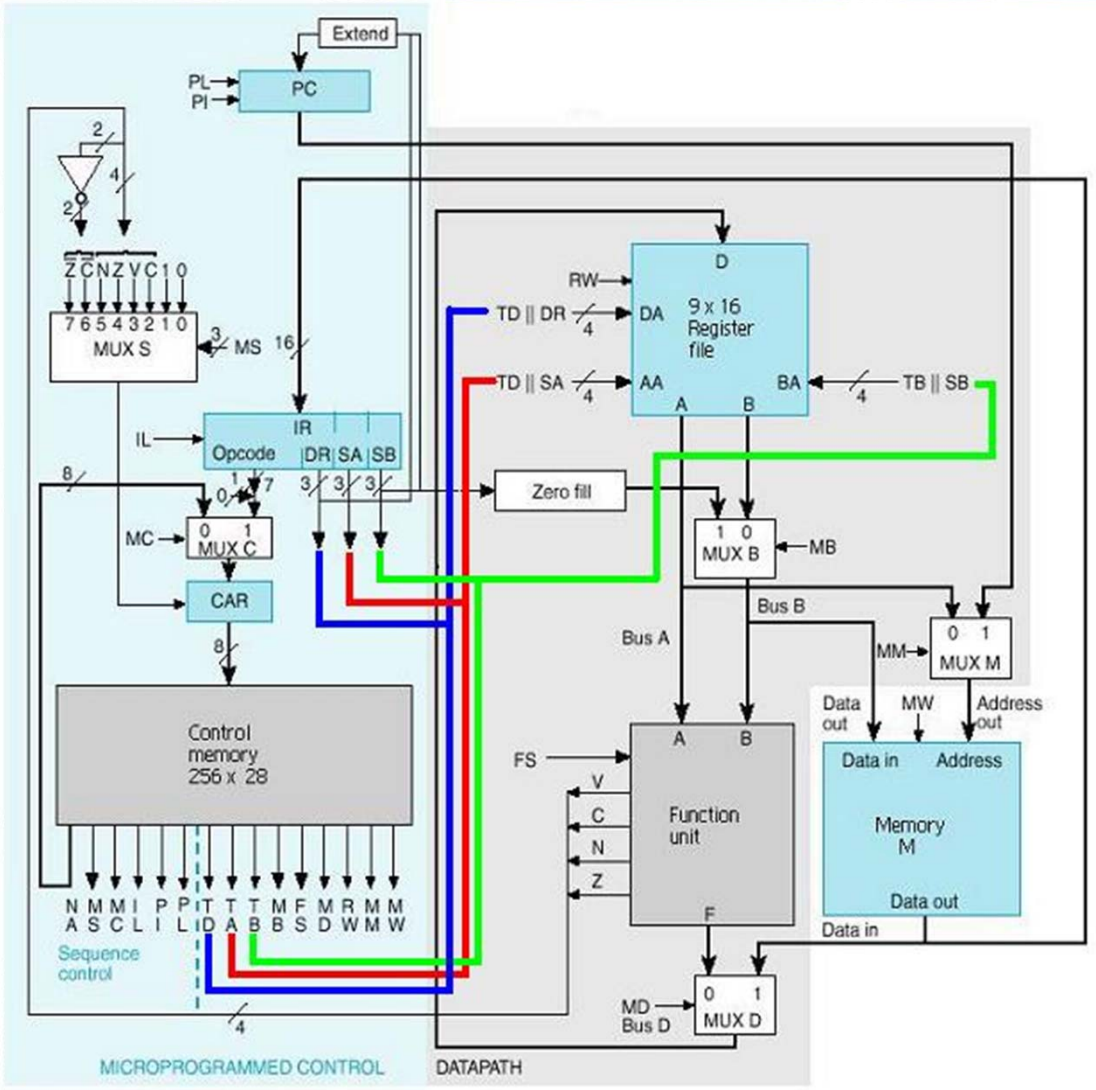


Temp Register

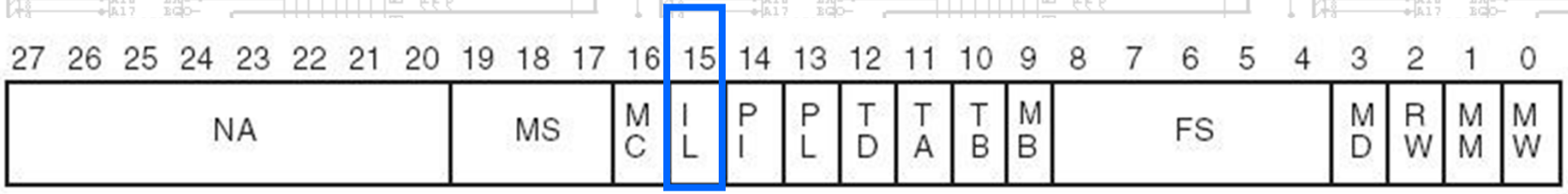


- Instructions are executed over multiple clock cycles
- This requires an additional register
  - R8 for temporary storage
- This register should be selected through an additional bit control signals:
  - TD, TA, TB
- These control signal are to the left of:
  - SA, SB, DR (from IR register)

CS2022 TD||DR -TA||SA - TB||SB

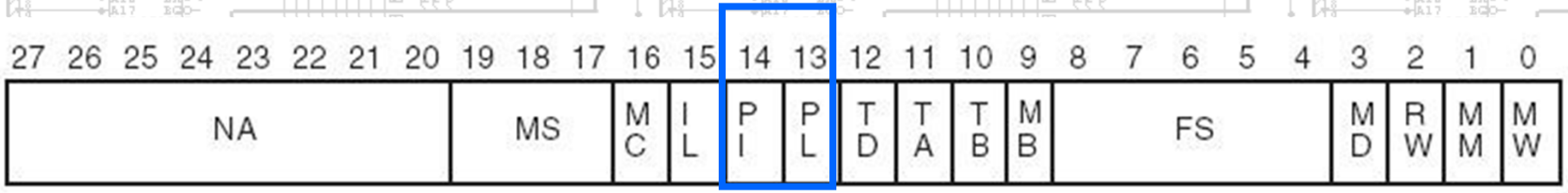


CS2022 IR Instruction Register



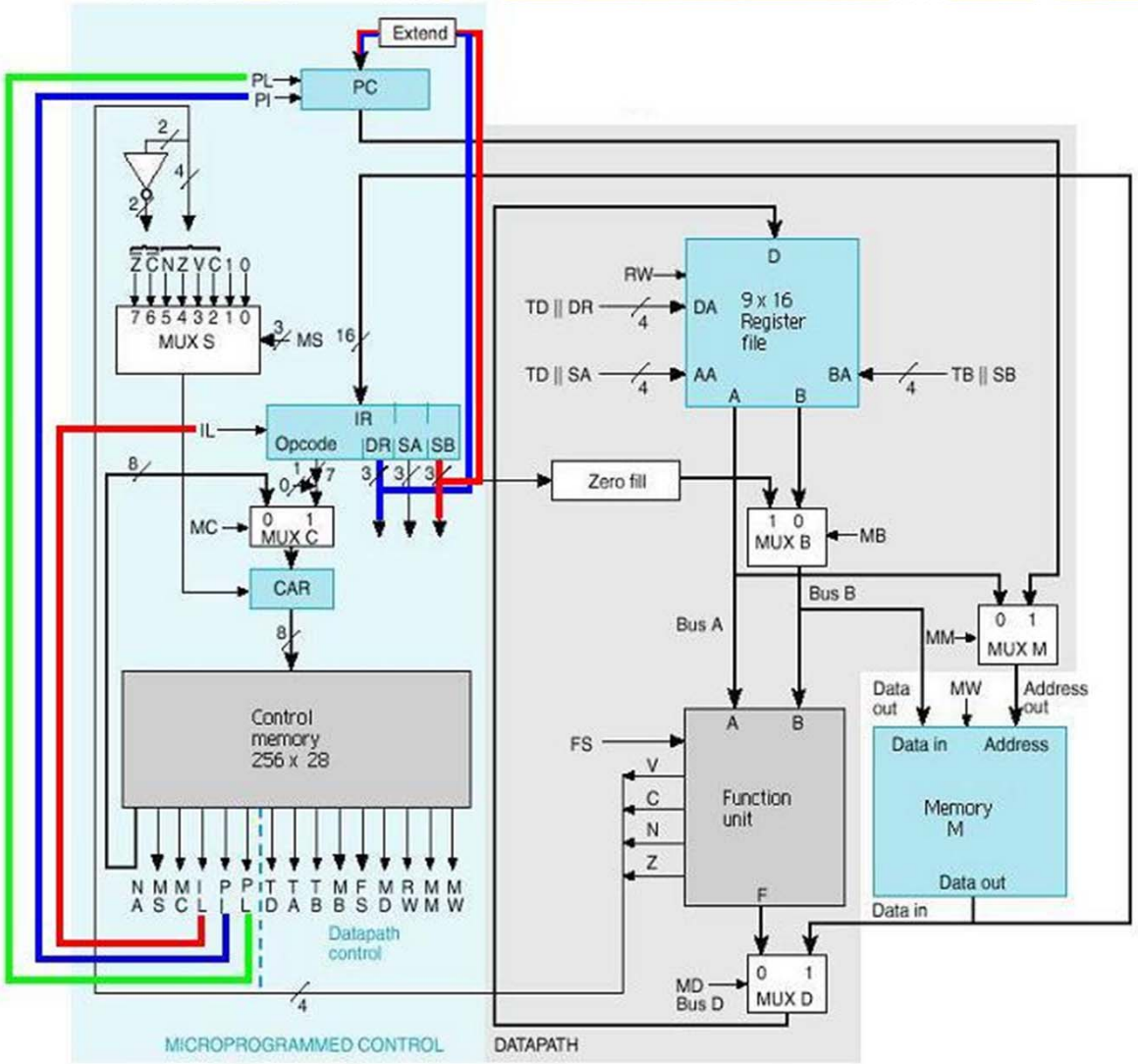
- Instructions must be held in an register during the execution of multiple micro-ops
- The IR is only loaded if an instruction is fetched from memory M
  - The IR has an load enable control signal IL
  - This signal is part of the control word

PC Program Counter Register

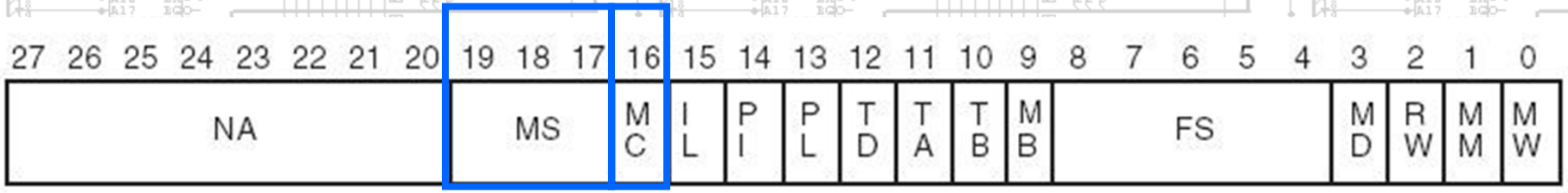


- The PC only increments if an instruction is fetched from memory M
- The control word has two bits that determine the PC modifications:
  - PI - increment enable signal
    - PC ← PC + 1
  - PL – PC load signal
    - PC ← PC + se AD

CS2022 IR - IL - PC - PI - PL

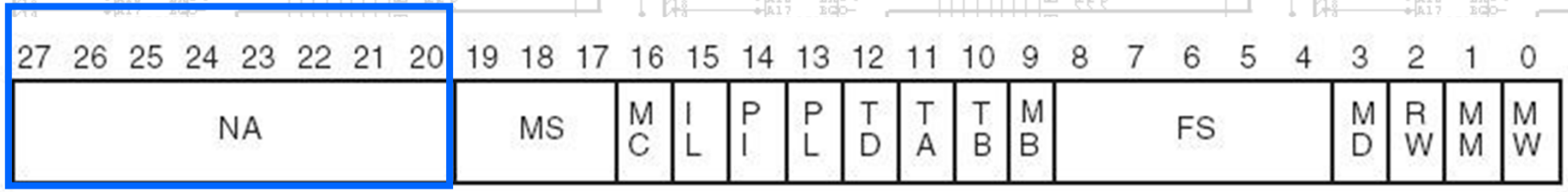


CS2022 Next Address Logic



- The CAR Control Address Register selects the control word in the 256x 28 control memory
- The next logic (MUX S) determines whether CAR is incremented on loaded.
  - Controlled with MS
- The source of the loaded address is determined by MUX C
  - Selected by MC

Next Address Field

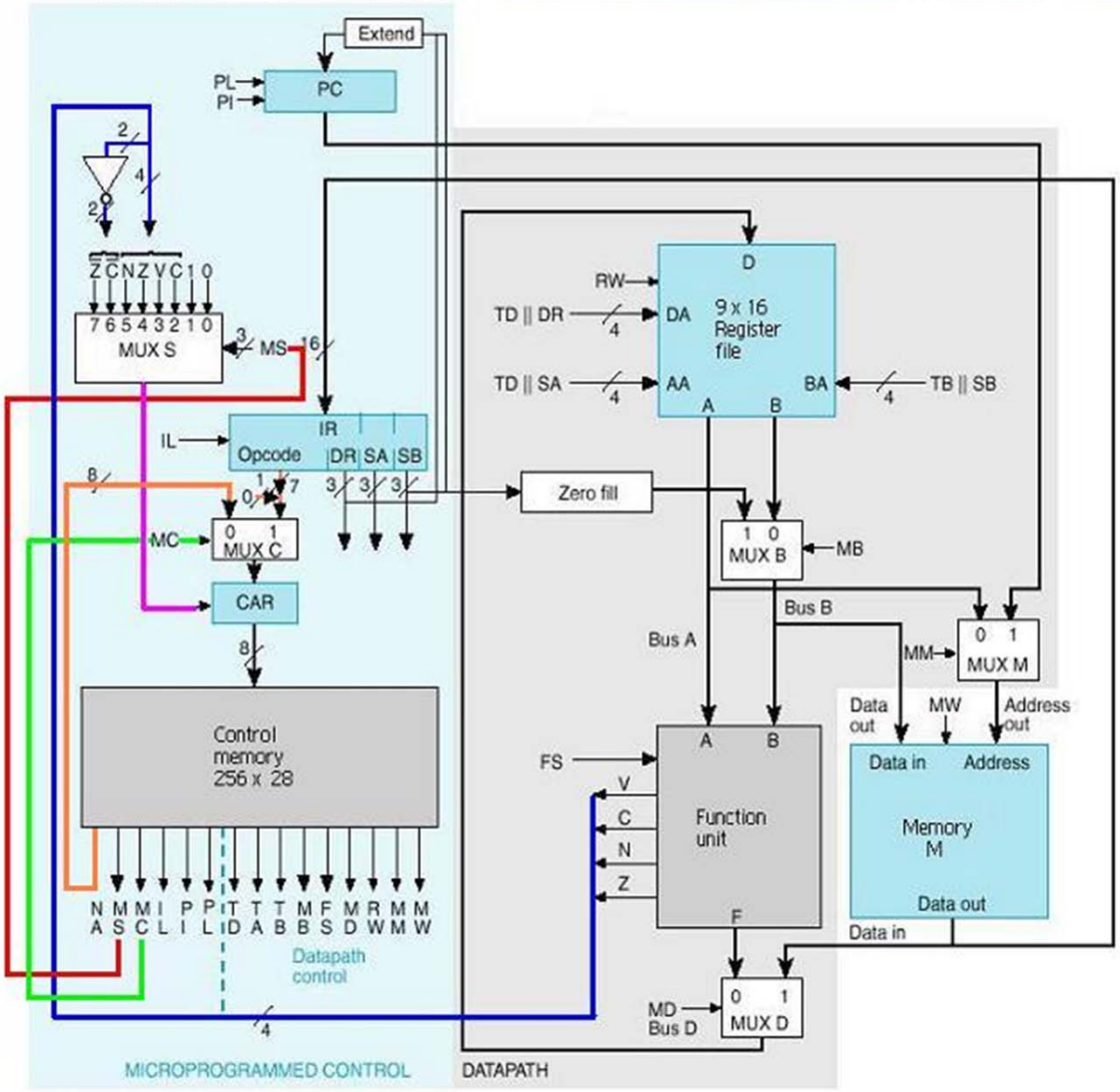


- The sources for the multiplexer can be:
  - Contents of the 8 bit NA Next Address field
  - 7 bit from the opcode field in the IR
- An opcode loaded into the CAR points to:
  - Microprogram in Control Memory
  - This program implements the instruction through the execution of micro operations
- MUX S determines whether the CAR is:
  - Incremented
  - Loaded

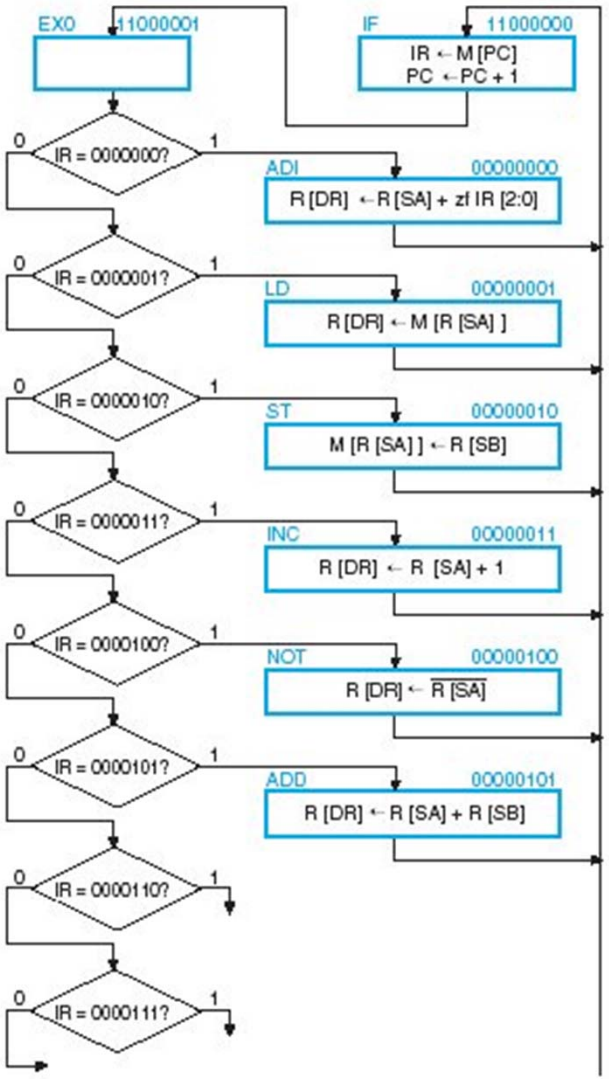
CS2022 Sequencer Control Fields

	MS		MC		IL		PI		PL	
Action	Symbolic Notation	Code	Select	Symbolic Notation	Action	Symbolic Notation	Action	Symbolic Notation	Action	Symbolic Notation	Code
Increment *CAR*	CNT	000	NA	NXA	No load	NLI	No load	NLP	No load	NLP	0
Load *CAR*	NXT	001	Opcode	OPC	Load instr.	LDI	Increment PC	INP	Load PC	LDP	1
If $C = 1$, load *CAR*; else increment *CAR*	BC	010									
If $V = 1$, load *CAR*; else increment CAR	BV	011									
If $Z = 1$, load *CAR*; else increment *CAR*	BZ	100									
If $N = 1$, load *CAR*; else increment *CAR*	BN	101									
If $C = 0$, load *CAR*; else increment *CAR*	BNC	110									
If $Z = 0$, load *CAR*, else increment *CAR*	BNZ	111									

CS2022 NA - MS - MC



CS2022 Microprogram ASM



CS2022 Microprogram Design

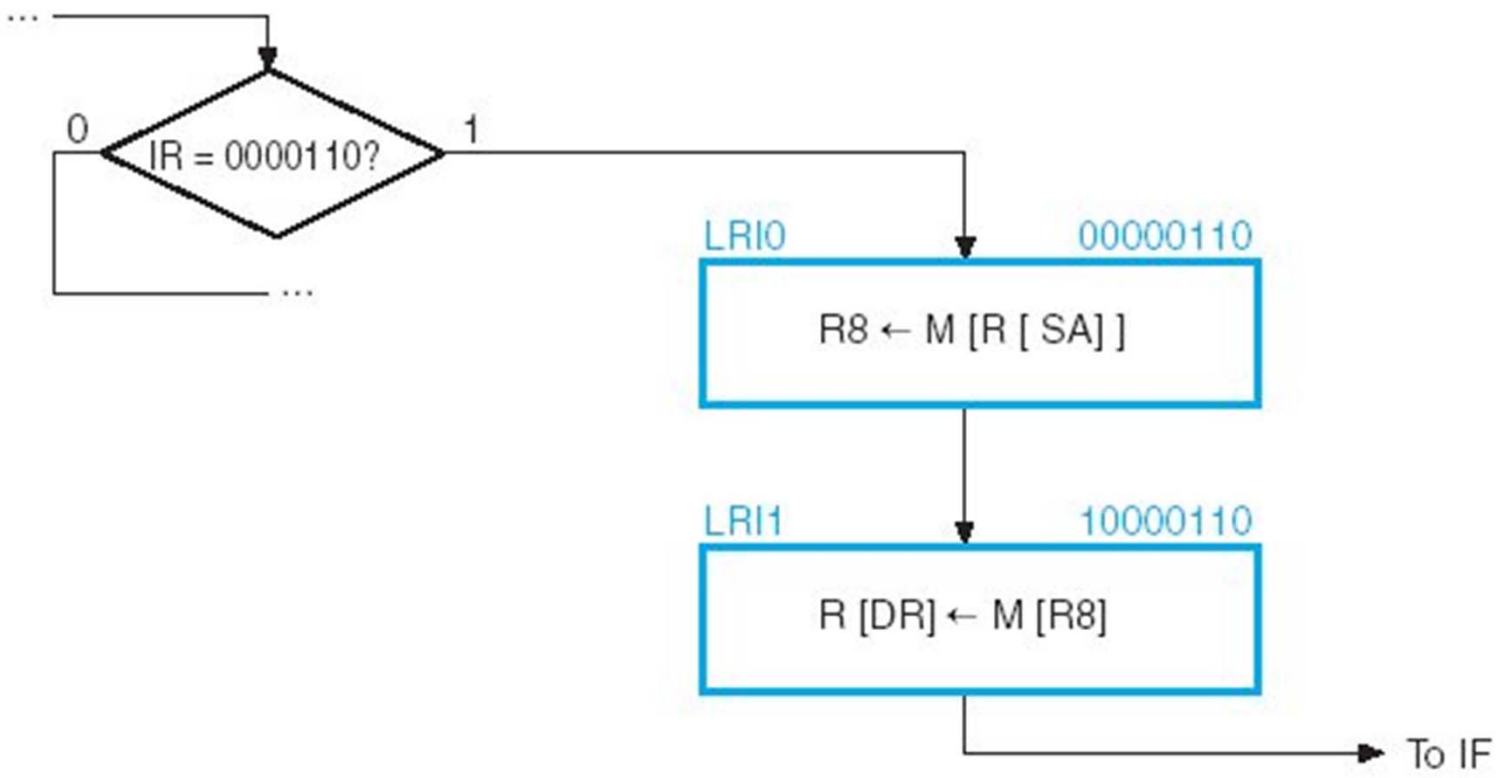
- Use ASM from Lecture 16 to design the Microprograms
- See Symbolic/Binary Microprogram on next slide

CS2022 Symbolic/Binary Microprogram

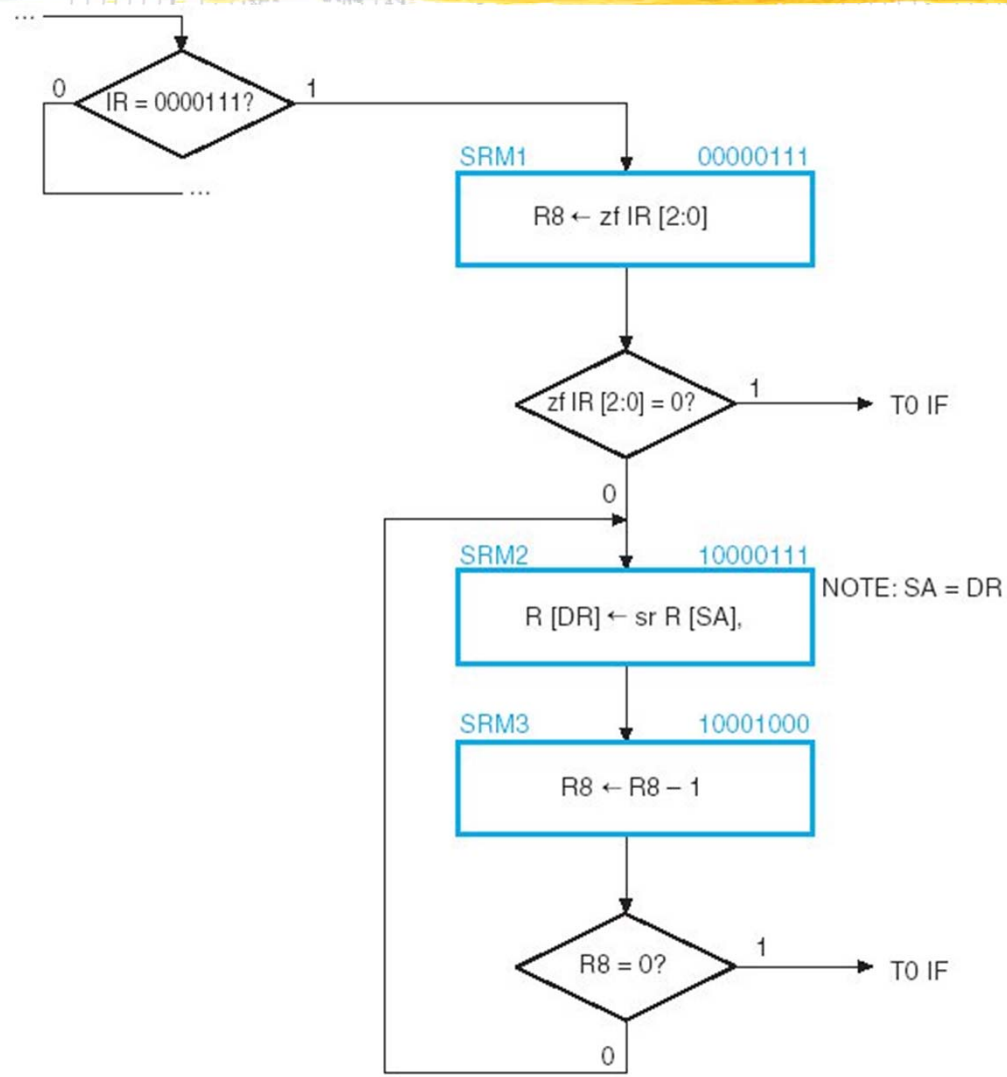
Address	NXT ADD	MS	MC	IL	PI	PL	TD	TA	TB	MB	FS	MD	RW	MM	MW
IF	EX0	CNT	—	LDI	INP	NLP	—	—	—	—	—	—	NW	PC	NW
EXO	—	NXT	OPC	NLI	NLP	NLP	—	—	—	—	—	—	NW	—	NW
ADI	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	Constant	$F = A + B$	FnUt	WR	—	NW
LD	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	—	Data	WR	MA	NW
ST	IF	NXT	NXA	NLI	NLP	NLP	—	SA	SB	Register	—	—	NW	MA	WR
INC	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	$F = A + 1$	FnUt	WR	—	NW
NOT	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	—	—	$F = \overline{A}$	FnUt	WR	—	NW
ADD	IF	NXT	NXA	NLI	NLP	NLP	DR	SA	SB	Register	$F = A + B$	FnUt	WR	—	NW

Address	NXT ADD	MS	MC	IL	PI	PL	TD	TA	TB	MB	FS	MD	RW	MM	MW
192	193	000	0	1	1	0	0	0	0	0	00000	0	0	1	0
193	000	001	1	0	0	0	0	0	0	0	00000	0	0	0	0
000	192	001	0	0	0	0	0	0	0	1	00010	0	1	0	0
001	192	001	0	0	0	0	0	0	0	0	00000	1	1	0	0
002	192	001	0	0	0	0	0	0	0	0	00000	0	0	0	1
003	192	001	0	0	0	0	0	0	0	0	00001	0	1	0	0
004	192	001	0	0	0	0	0	0	0	0	01110	0	1	0	0
005	192	001	0	0	0	0	0	0	0	0	00010	0	1	0	0

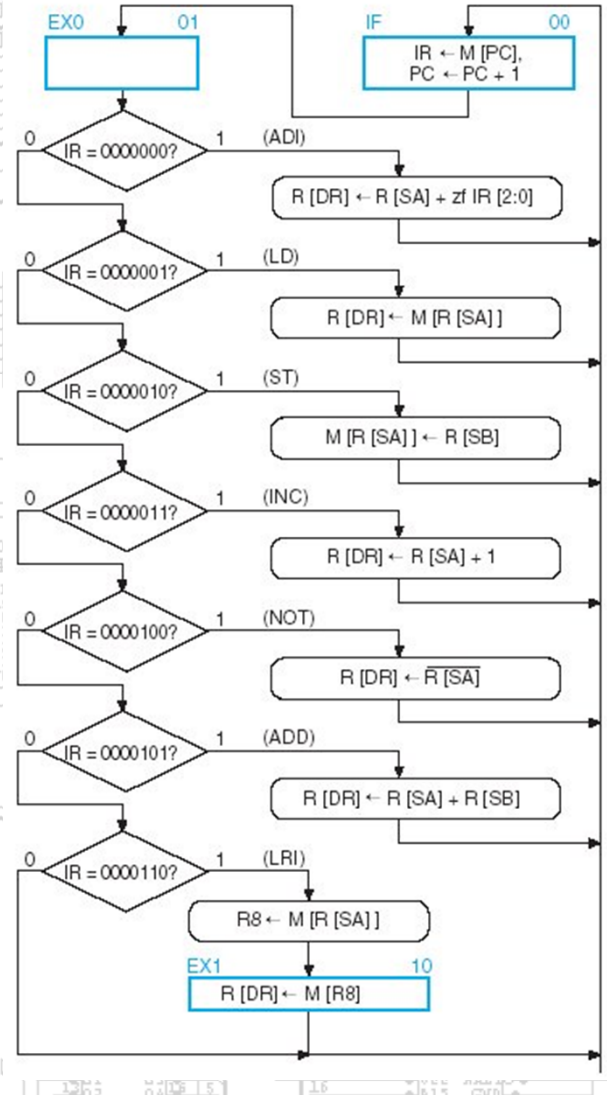
CS2022 Indirect Instruction ASM



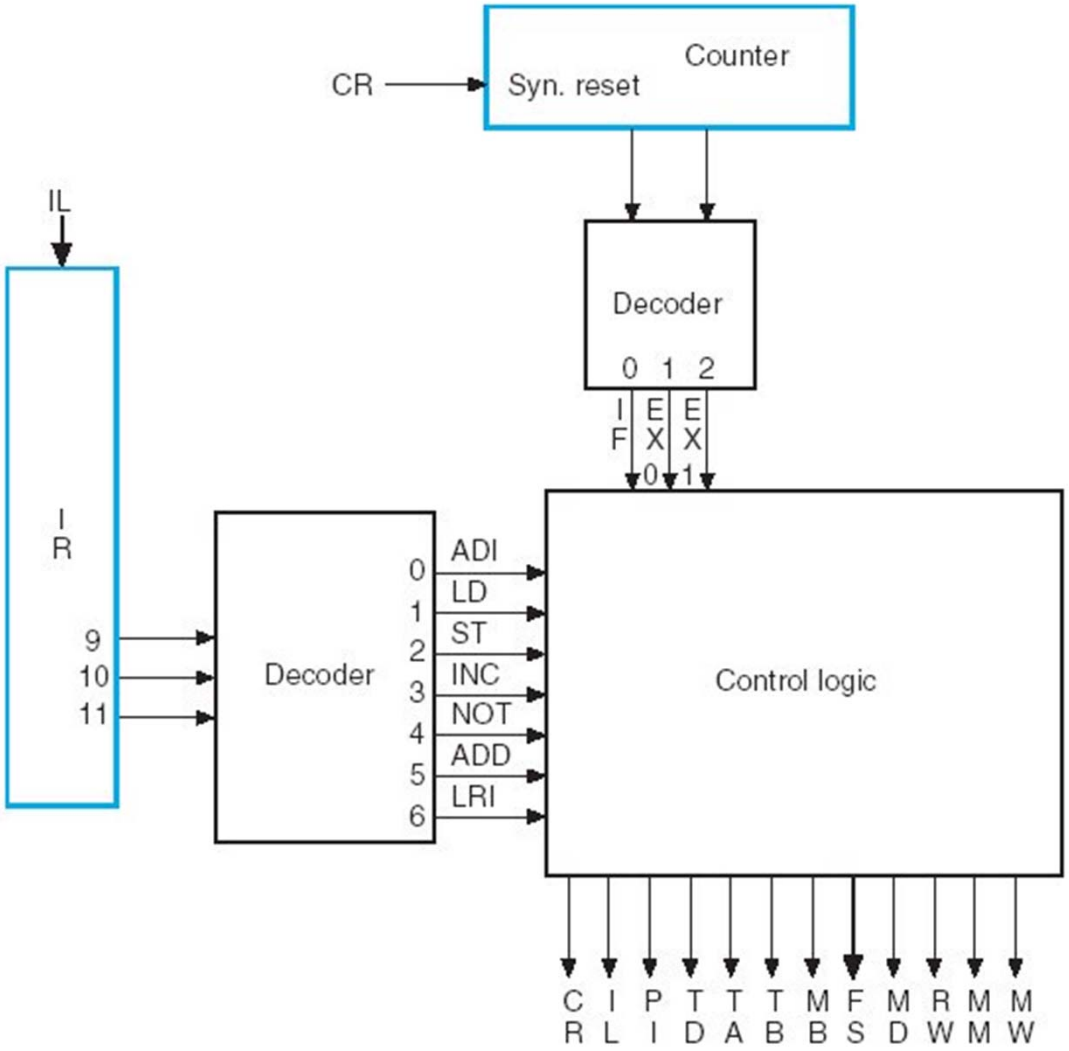
CS2022 Right-Shift Instruction ASM



CS2022 Hardwired Multiple-Cycle Control



CS2022 Hardwired Control Unit



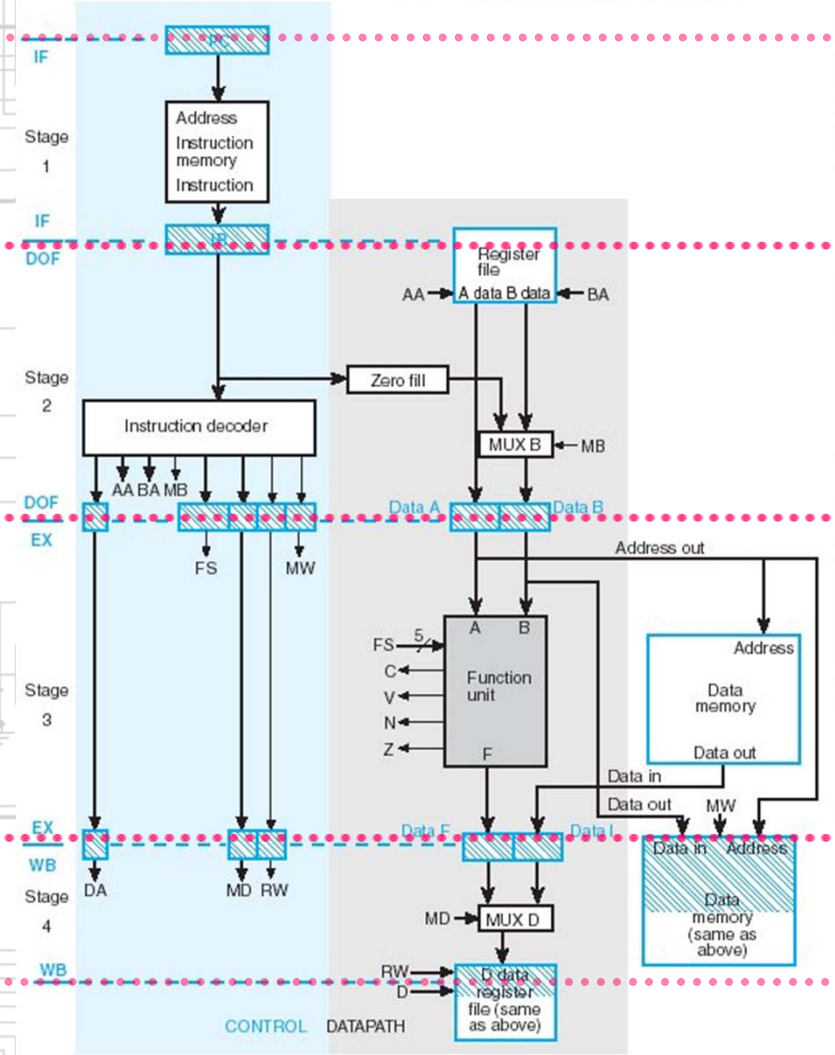
Pipelined (based on single-cycle)

Instruction Fetch

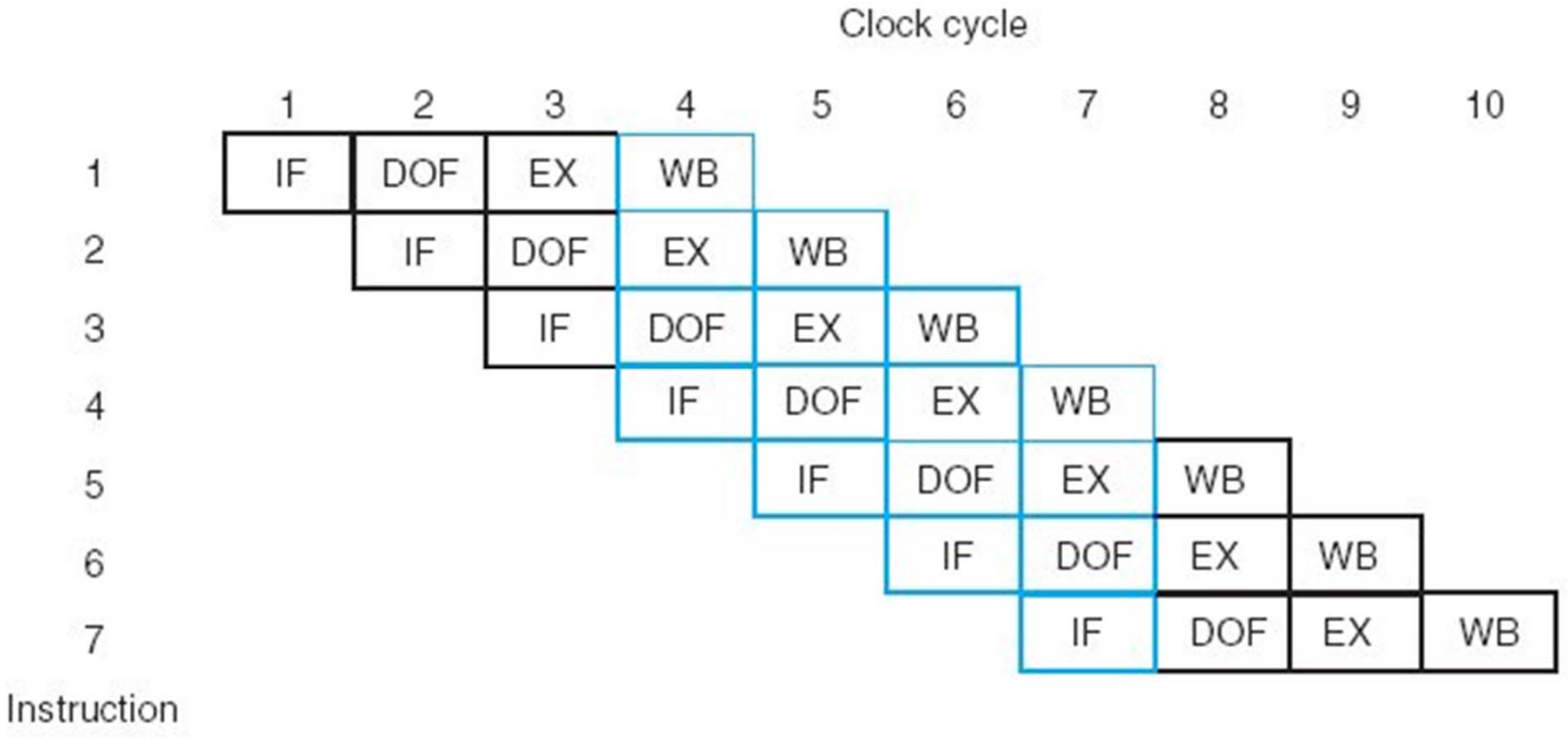
Decode and Operand Fetch

Execution

Write-back

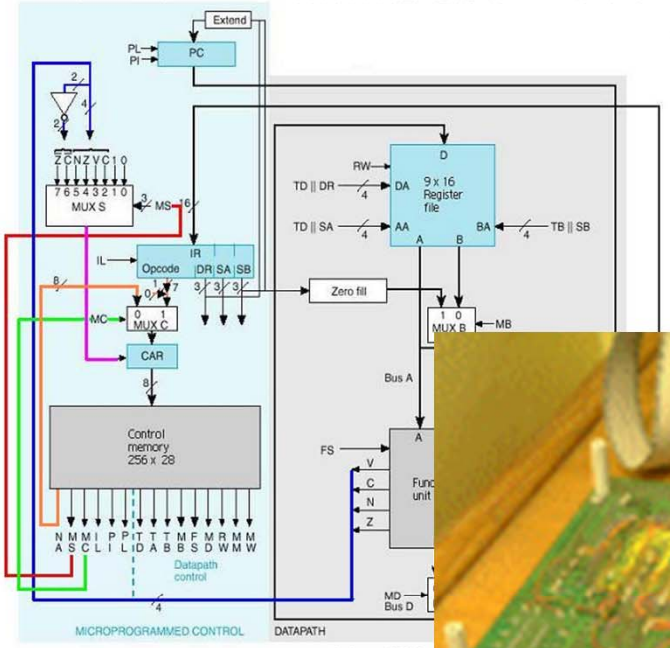


CS2022 Pipelined Execution Pattern

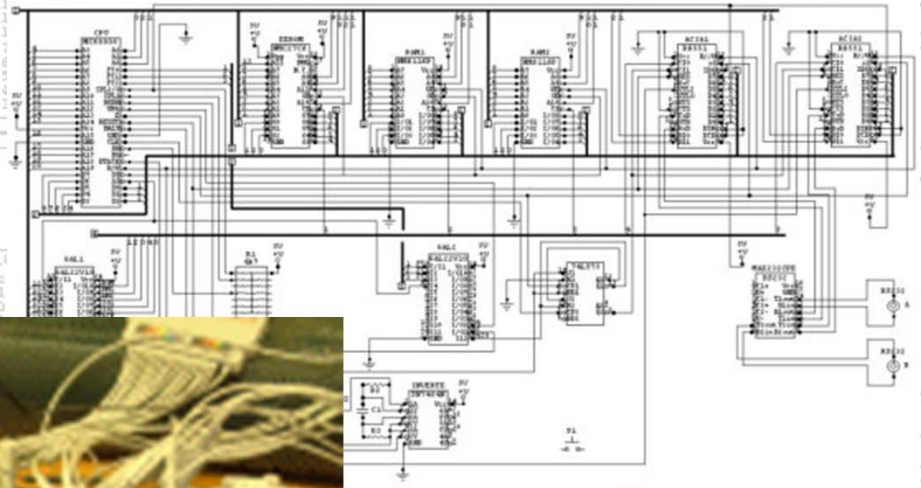


CS2022

Computer Architecture and Microprocessor Systems



CPU



Board Level

Assembly Language, Digital Logic, Electrotechnology