University Of Diyala
College Of Engineering
Department of Computer Engineering

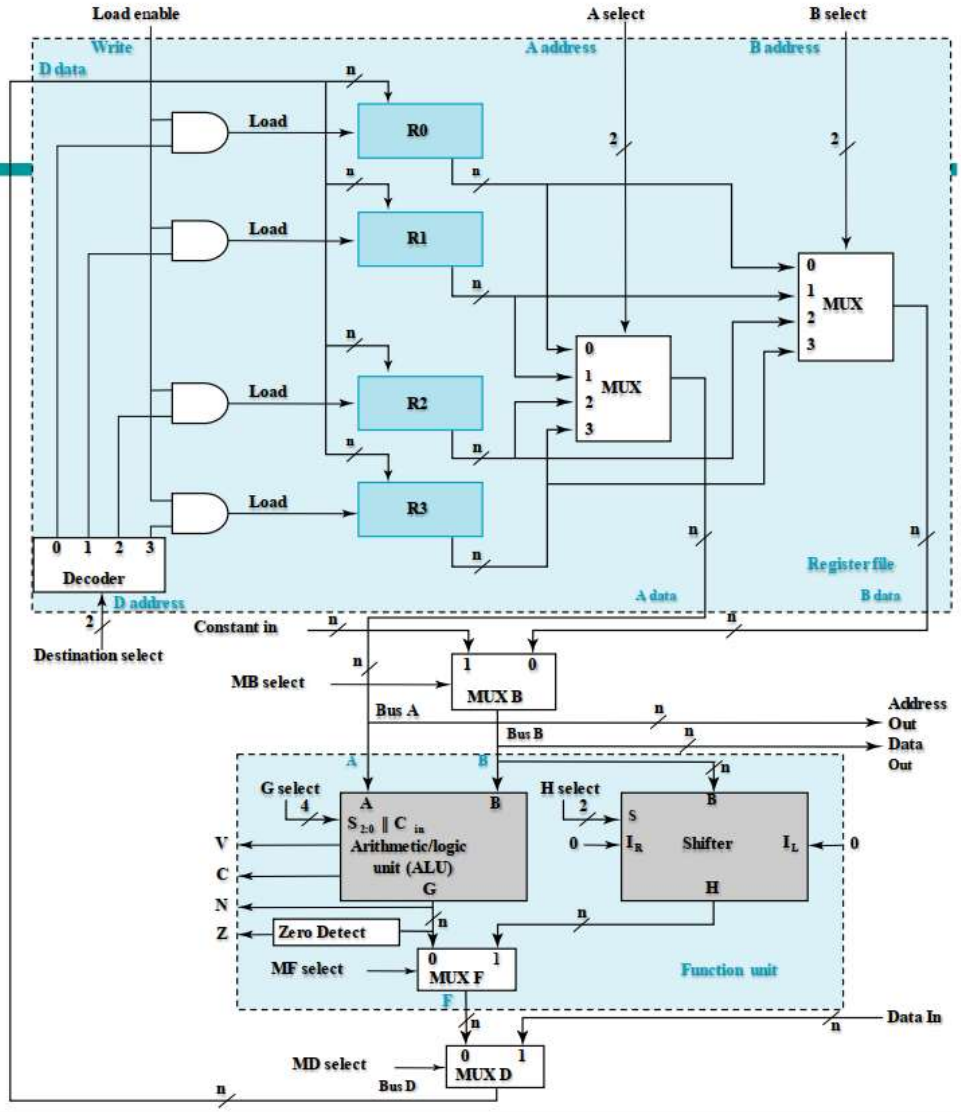
Digital System Design II
Microprogramming II-Part2

Dr. Yasir Al-Zubaidi

Third stage

2021

Datapath


Load enable
A select
B select
Write
A address
B address
D data
Load
R0
R1
R2
R3
MUX
Decoder
Register file
A data
B data
D address
Destination select
Constant in
MB select
MUX B
Bus A
Bus B
Address Out
Data Out
A
B
G select
H select
S2:0 ‖ C in
Arithmetic/logic unit (ALU)
G
Shifter
H
V
C
N
Z
Zero Detect
MF select
MUX F
Function unit
F
Data In
MD select
MUX D
Bus D

Instruction Set Architecture (ISA) for Simple Computer (SC)

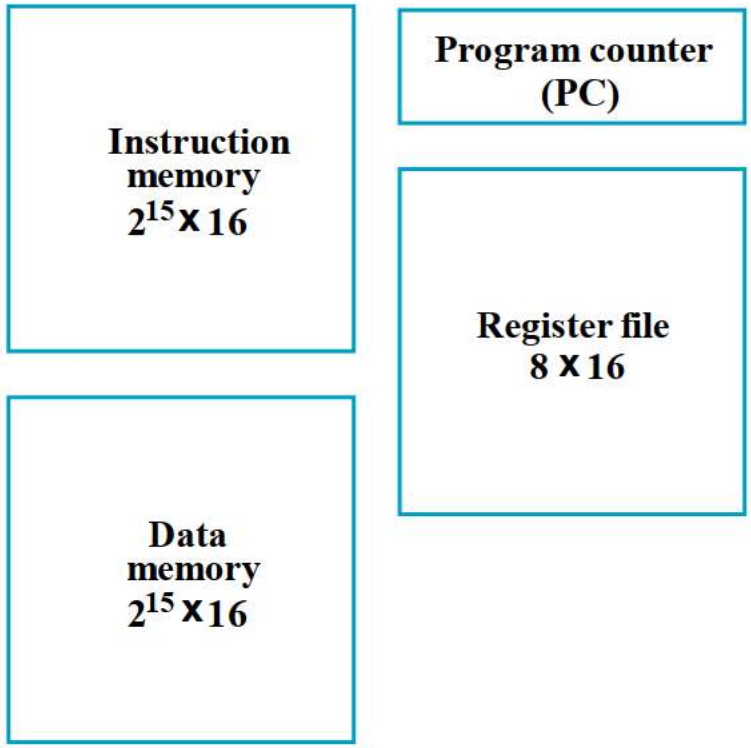
- Instructions are stored in RAM or ROM as a *program,* the addresses for instructions are provided by a *program counter (PC)*
  - Count up or load a new address
  - The PC and associated control logic are part of the Control Unit
- A typical instruction specifies:
  - Operands to use
  - Operation to be performed
  - Where to place the result, or which instruction to execute next
- Executing an instruction
  - Activate the necessary sequence of operations specified by the instruction
  - Be controlled by the control unit and performed in:
    - datapath
    - control unit
    - external hardware such as memory or input/output

Example ISAs

- RISC (Reduced Instruction Set Computer)
  - Digital Alpha
  - Sun Sparc
  - MIPS RX000
  - IBM PowerPC
  - HP PA/RISC
- CISC (Complex Instruction Set Computer)
  - Intel x86
  - Motorola 68000
  - DEC VAX
- VLIW (Very Large Instruction Word)
  - Intel Itanium

ISA: Storage Resources

- "Harvard architecture":
  Separate instruction and data memories
- Permit use of single clock cycle per instruction implementation
- Due to use of "cache" in modern computer architectures, it is a fairly realistic model

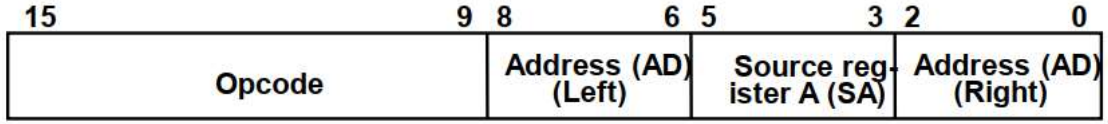


ISA: Instruction Format

- The three formats are: Register, Immediate, and Jump/Branch

(a) Register

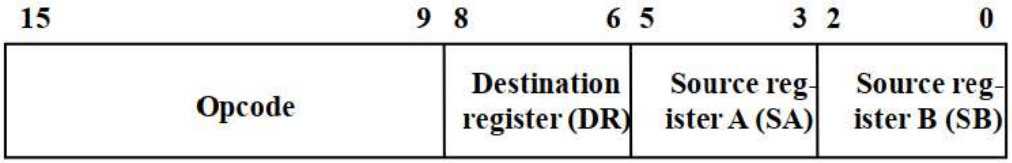
(b) Immediate



(c) Jump and Branch

- All formats contain an Opcode field in bits 9 through 15.
  - The Opcode specifies the operation to be performed

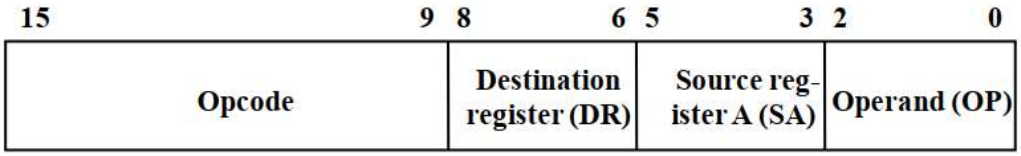
ISA: Instruction Format - Register



(a) Register

- This format supports:
  - R1 ← R2 + R3
  - R1 ← sl R2
- Three 3-bit register fields:
  - DR - destination register (R1 in the examples)
  - SA - the A source register (R2 in the first example)
  - SB - the B source register (R3 in the first example and R2 in the second example)
- Why is R2 in the second example SB instead of SA?

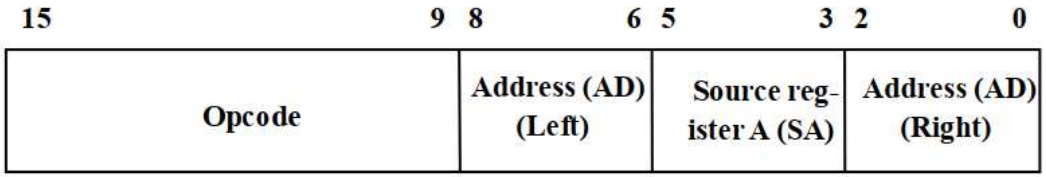
ISA: Instruction Format - Immediate



(b) Immediate

- This format supports:
  - R1 ← R2 + 3
- The B Source Register field is replaced by an Operand field OP specifying a constant. (3-bit constant, values from 0 to 7)
- The constant:
  - Zero-fill (on the left of) the operand to form 16-bit constant
  - 16-bit representation for values 0 through 7

ISA: Instruction Format – Jump & Branch



(c) Jump and Branch

- This instruction supports changes in the sequence of instruction execution by adding an extended, 6-bit, signed 2's-complement *address offset* to the PC value
- The SA field: permits jumps and branches on N or Z based on the contents of *Source register A*
- The Address (AD) field (6-bit) replaces the DR and SB fields
  - Example: Suppose that a jump for the Opcode and the PC contains 45 (0…0101101) and AD contains – 12 (110100). Then the new PC value will be:
    0…0101101 + (1…110100) = 0…0100001 (i.e., 45 + (– 12) = 33)

ISA: Instruction Specifications

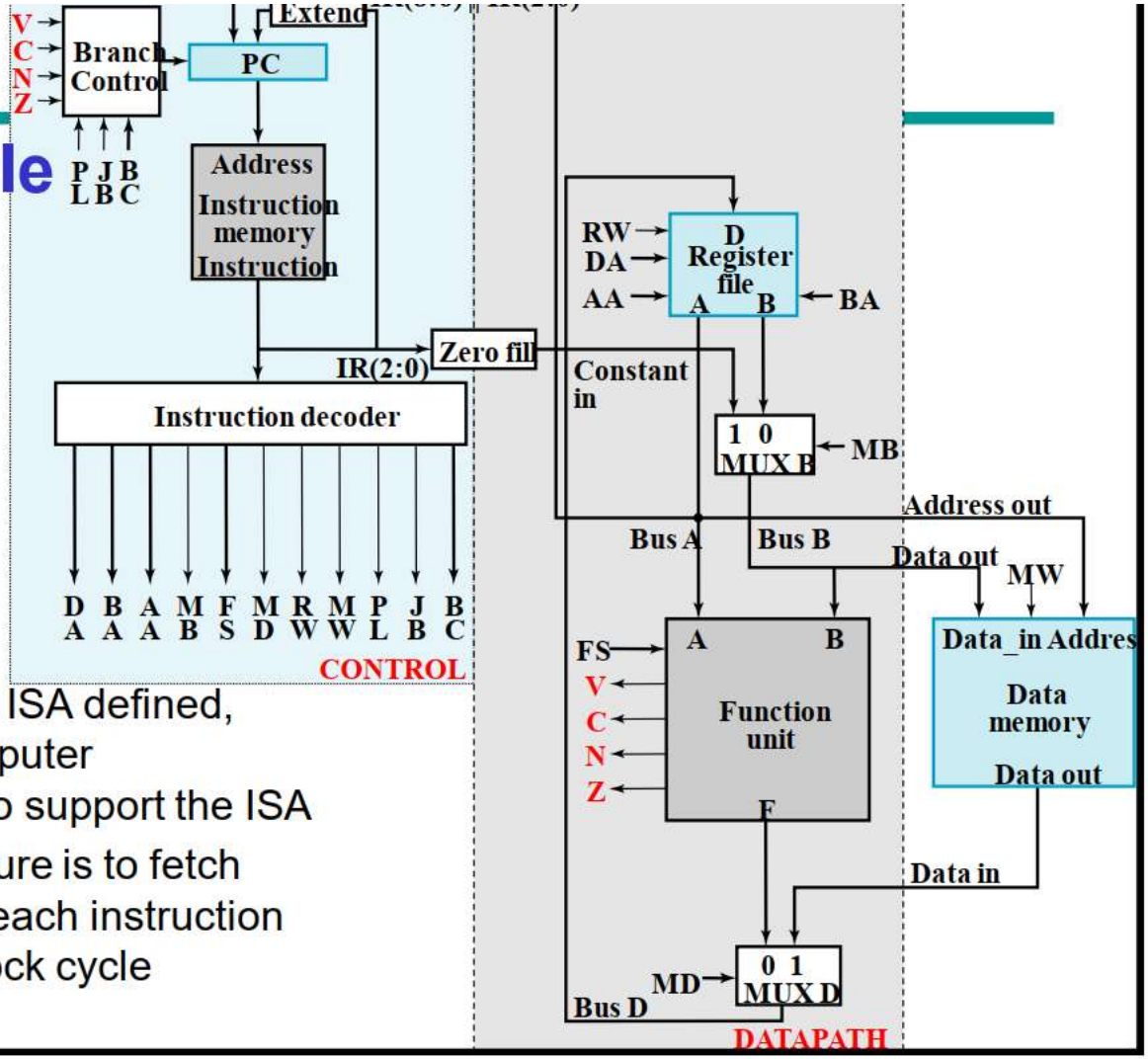
Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← $\overline{R[SA]}$	N, Z
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[R[SA]]	
Store	0100000	ST	RA,RB	M[R[SA]]← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se AD	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se AD	
Jump	1110000	JMP	RA	PC ← R[SA]	

ISA:Example Instructions and Data in Memory

Memory Representation of Instruction and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Field	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 Im mediate)	DR: 2, SA :7, OP :3	R2 ← R7 + 3
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC − 20
70	0000000 00110 0 000	Data = 192. After execution of instruction in 35, Data = 80.		

Single-Cycle Hardwired Control:



- Based on the ISA defined, design a computer architecture to support the ISA
- The architecture is to fetch and execute each instruction in a single clock cycle

The Control Unit

- Datapath: the Data Memory has been attached to the *Address Out*, *Data Out,* and *Data In* lines of the Datapath.

- Control Unit:
  - The MW input to the Data Memory is the Memory Write signal from the Control Unit.
  - The Instruction Memory address input is provided by the PC and its instruction output feeds the Instruction Decoder.
  - Zero-filled IR(2:0) becomes Constant In
  - Extended IR(8:6) || IR(2:0) and Bus A are address inputs to the PC.
  - The PC is controlled by Branch Control logic

Program Counter (PC) Function

- PC function is based on instruction specifications involving jumps and branches:

Branch on Zero	**BRZ**	**if (R[SA] = 0) PC ← PC + seA D**
Branch on Negative	**BRN**	**if (R[SA] < 0) PC ← PC + seA D**
Jump	**JMP**	**PC ← R[SA]**

  - The first two transfers require addition to the PC of:
    - Address Offset = Extended IR(8:6) || IR(2:0)
  - The third transfer requires that the PC be loaded with:
    - Jump Address = Bus A = R[SA]

- In addition to the above register transfers, the PC must implement the counting function:
  - PC ← PC + 1

PC Function (Contd.)

- Branch Control determines the PC transfers based on five inputs:
  - N,Z – negative and zero status bits
  - PL – load enable for the PC
  - JB – Jump/Branch select: If JB = 1, Jump, else Branch
  - BC – Branch Condition select: If BC = 1, branch for N = 1, else branch for Z = 1.

PL	JB	BC	PC Operation
0	X	X	Count Up
1	1	X	Jump
1	0	1	Branch on Negative (else Count Up)
1	0	0	Branch on Zero (else Count Up)

Instruction Decoder

- Converts the instruction into the signals necessary to control the computer during the single cycle execution, combinational
  - Inputs: the 16-bit Instruction
  - Outputs: control signals
    - DA, AA, and BA: Register file addresses (IR (8:0))
      - simply pass-through signals: DA = DR, AA = SA, and BA = SB
    - FS: Function Unit Select
    - MB and MD: Multiplexer Select Controls
    - RW and MW: Register file and Data Memory Write Controls
    - PL, JB, and BC: PC Controls
- Observe that for other than branches and jumps, FS = IR(12:9)
  - The other control signals should depend as much as possible on IR(15:13)

Instruction Decoder (Contd.)

Truth Table for Instruction Decoder Logic

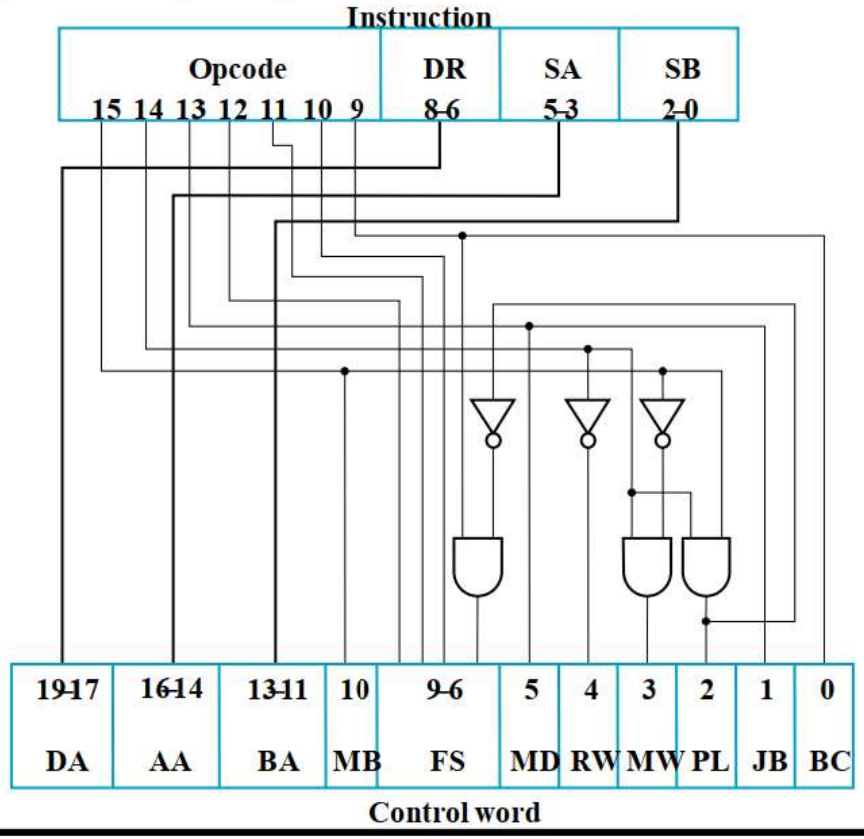
	Instruction Bits				Control Word Bits						
Instruction Function Type	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
1. Function unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
2. Memory read	0	0	1	X	0	1	1	0	0	X	X
3. Memory write	0	1	0	X	0	X	0	1	0	X	X
4. Function unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
5. Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
6. Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
7. Unconditional Jump	1	1	1	X	X	X	0	0	1	1	X

Instruction Decoder (Contd.)

- Instruction types are based on the control blocks and the seven control signals to be generated (MB, MD, RW, MW, PL, JB, BC):
  - Datapath and Memory Control (types 1-4)
    - Mux B
    - Memory and Mux D
  - PC Control (types 5-7)
    - Bit 15 = Bit 14 = 1 => PL
    - Bit 13 => JB.
    - Bit 9 was use as BC which contradicts FS = 0000 needed for branches. To force FS(0) to 0 for branches, Bit 9 into FS(0) is disabled by PL.

Instruction Decoder (Contd.)

- The end result by use of the types, careful assignment of codes, and use of don't cares, yields very simple logic:
- This completes the design of most of the essential parts of the single-cycle simple computer



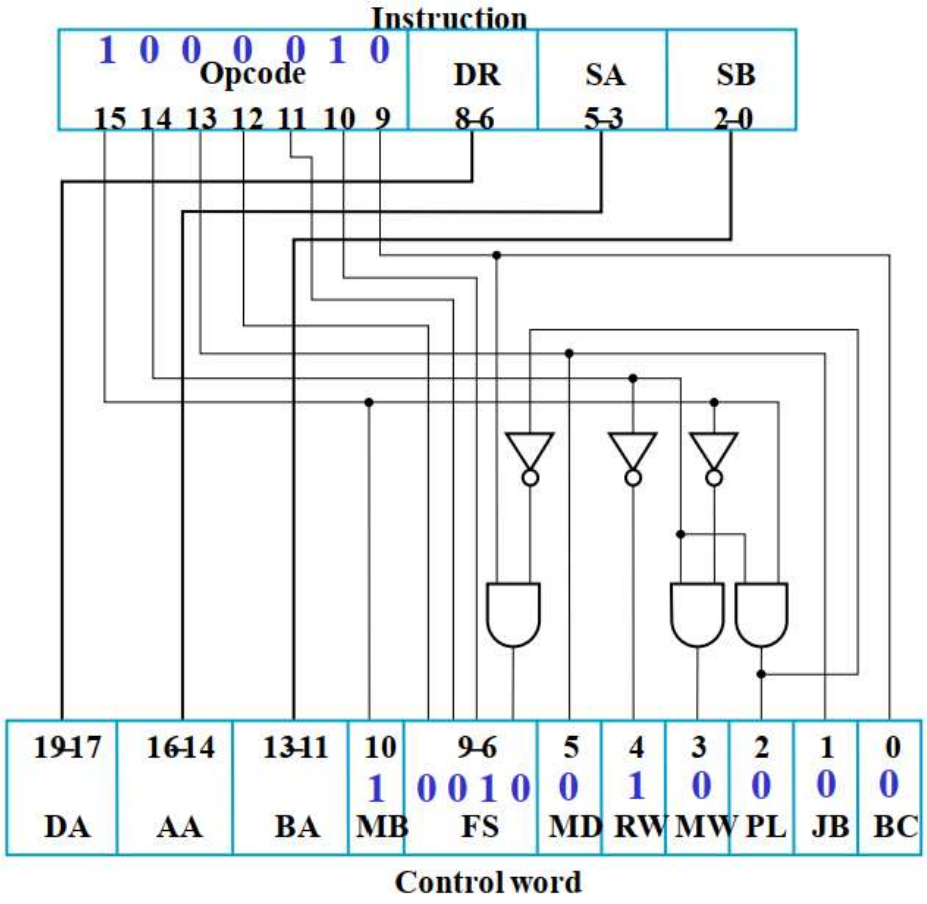
Example Instruction Execution

Six Instructions for the Single-Cycle Computer

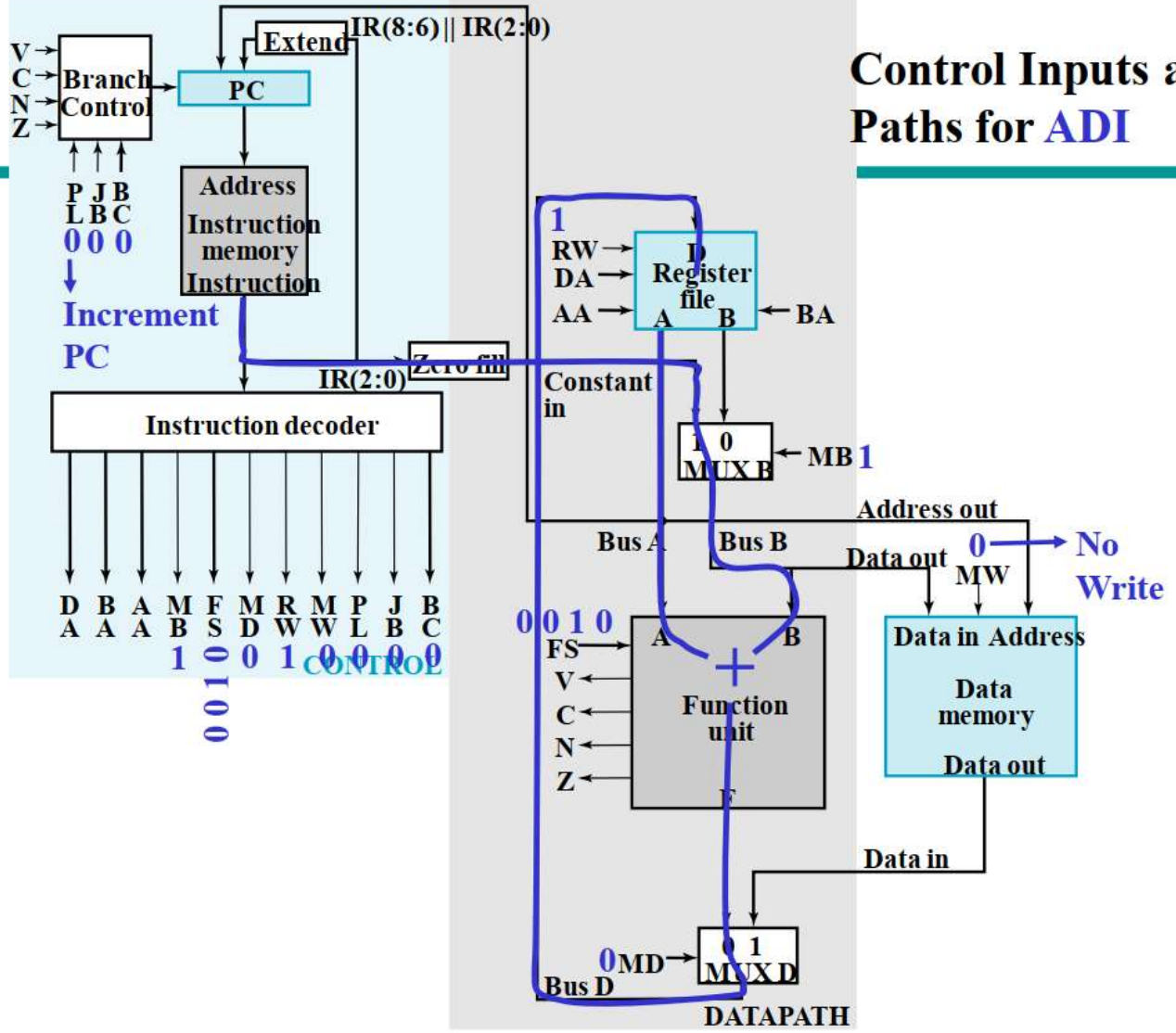
Operation code	Symbolic name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000 010	ADI	Immediate	Add immediate operand	$R[DR] \leftarrow R[SA] + zf\,I(2:0)$	1	0	1	0	0	0	0
0010 000	LD	Register	Load memory content into register	$R[DR] \leftarrow MR[SA]$	0	1	1	0	0	1	0
0100 000	ST	Register	Store register content in memory	$MR[SA] \leftarrow R[SB]$	0	1	0	1	0	0	0
0001 110	SL	Register	Shift left	$R[DR] \leftarrow slR[SB]$	0	0	1	0	0	1	0
0001 011	NOT	Register	Complement register	$R[DR] \leftarrow \overline{R[SA]}$	0	0	1	0	0	0	1
1100 000	BRZ	Jump/Branch	If $R[SA] = 0$, branch to PC + seAD	If $R[SA] = 0$, $PC \leftarrow PC + seAD$, If $R[SA] \neq 0$, $PC \leftarrow PC + 1$	1	0	0	0	1	0	0

- Decoding, control inputs and paths shown for ADI, LD and BRZ on next 6 slides

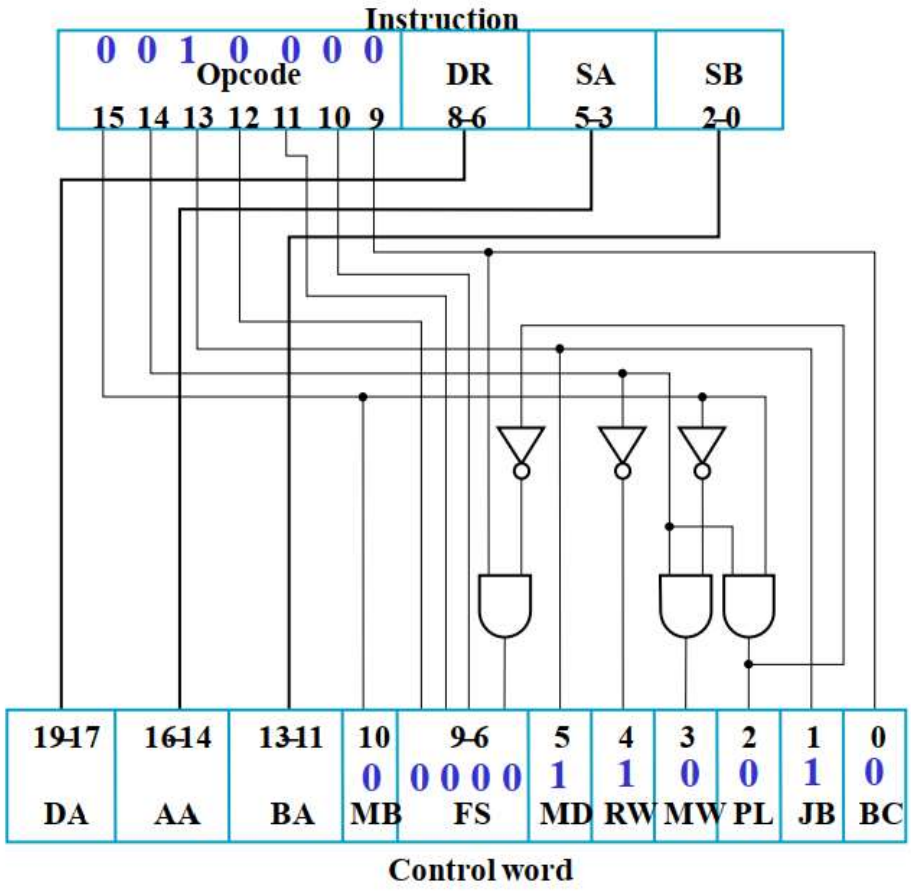
Decoding for ADI



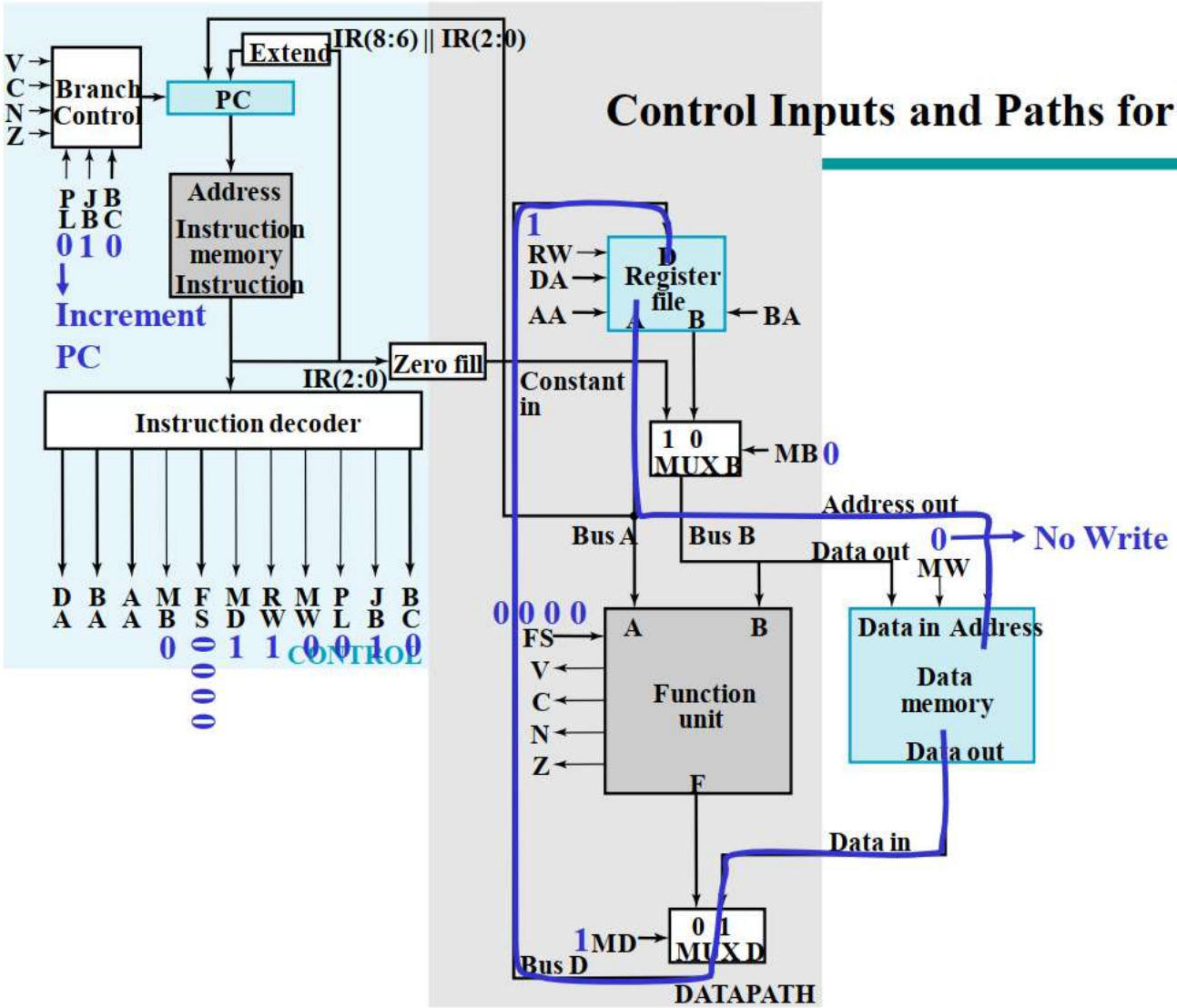
Control Inputs and Paths for ADI



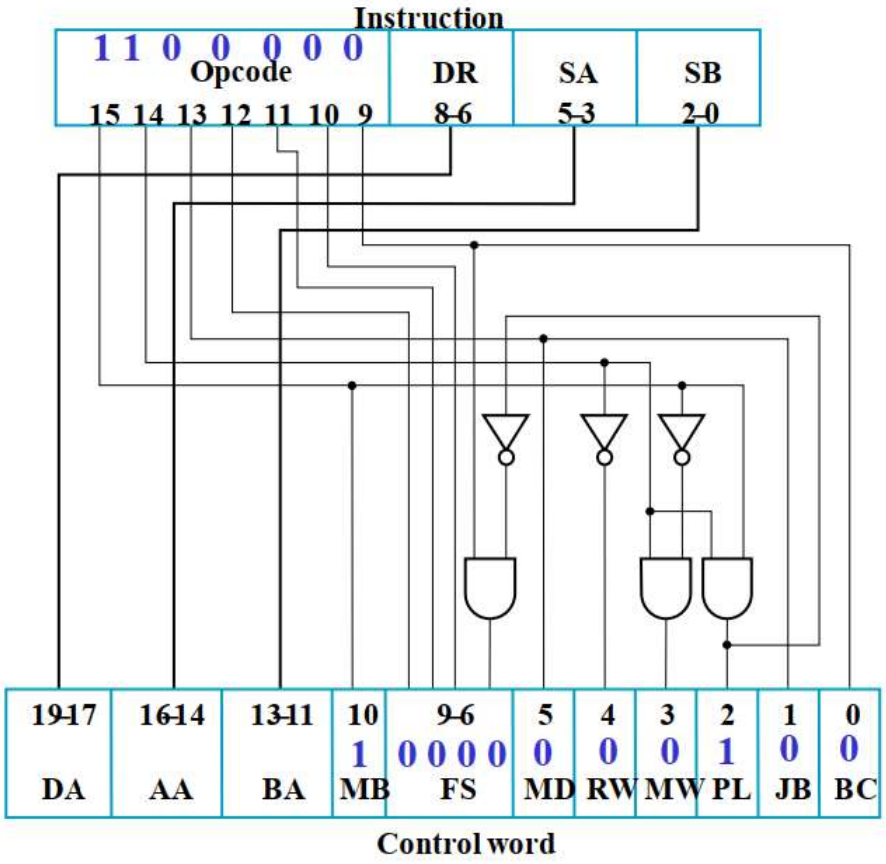
Decoding for LD



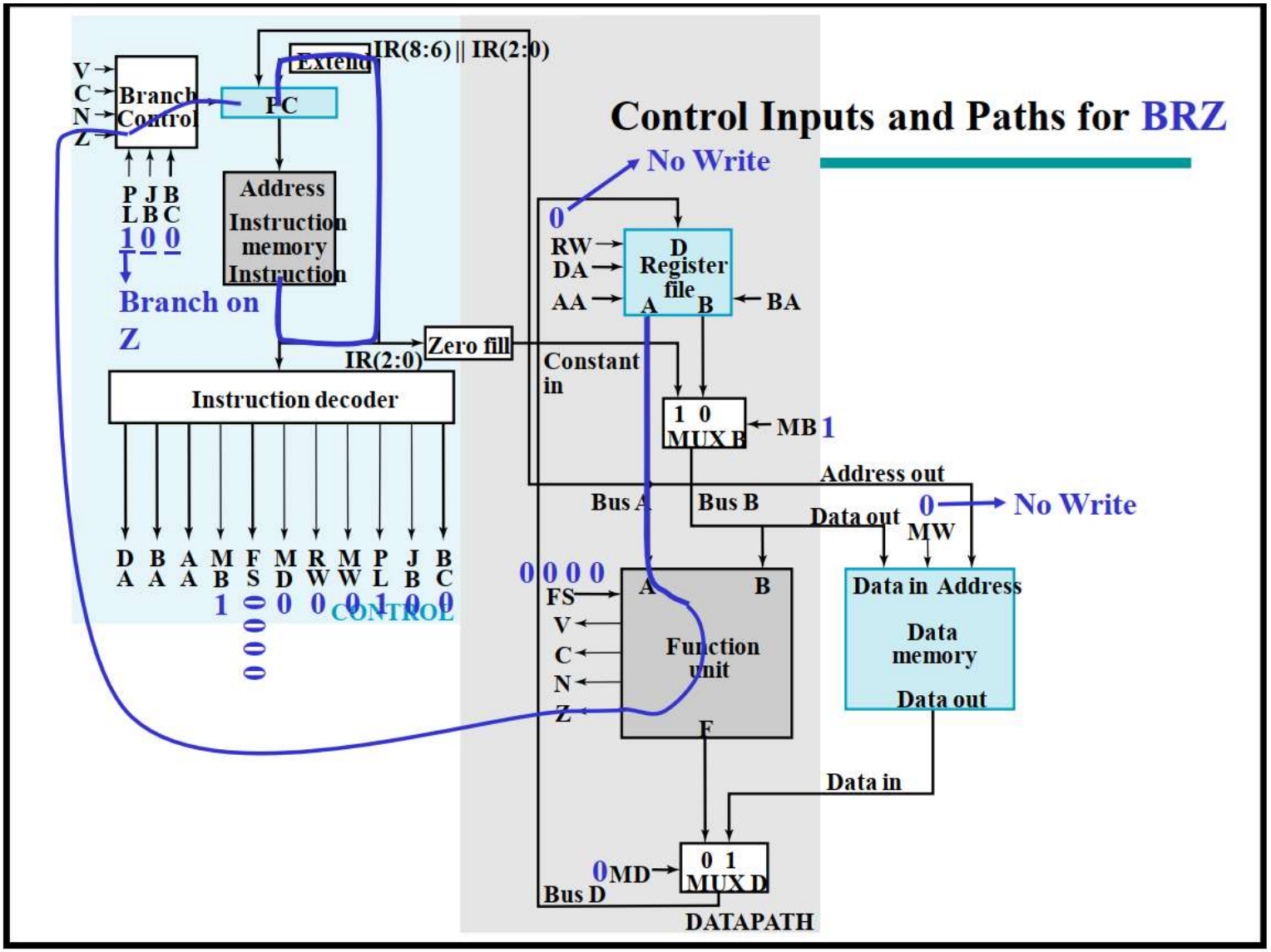
Control Inputs and Paths for LD



Decoding for BRZ

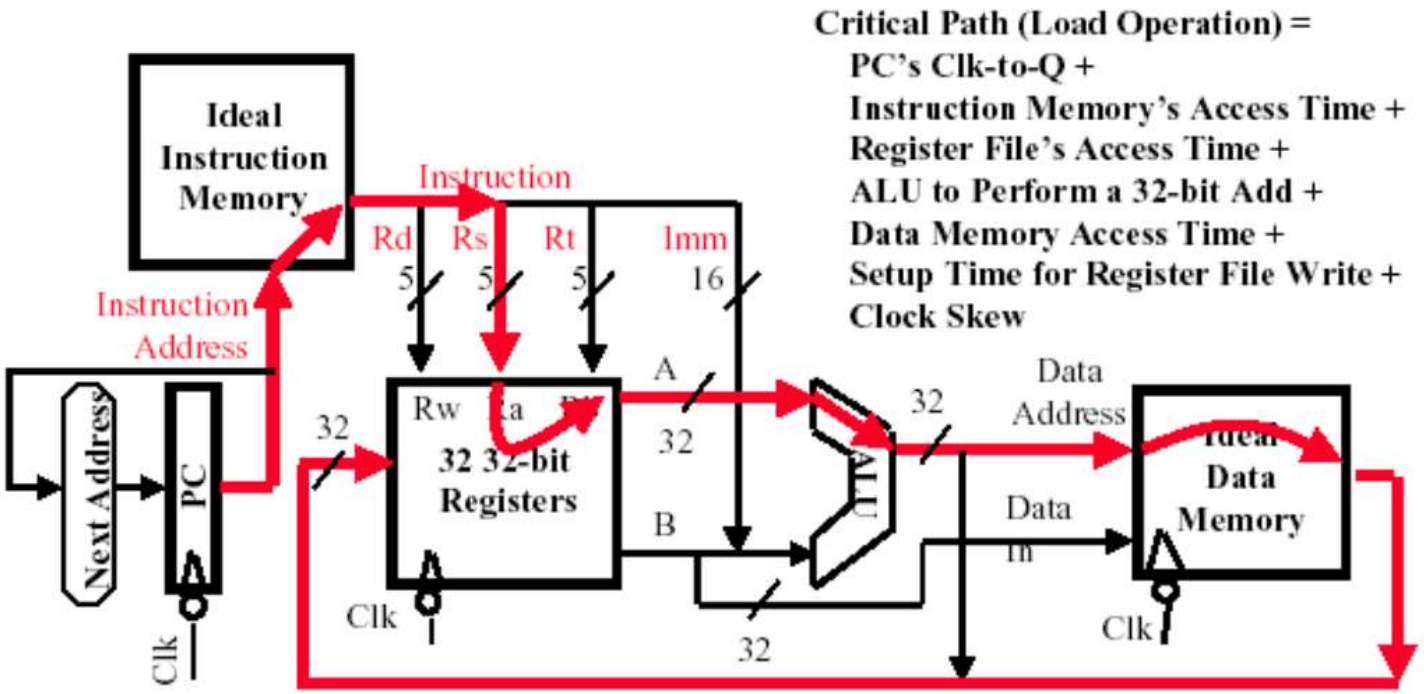


Control Inputs and Paths for BRZ



Abstract View of Critical Path

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."



Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II
Microprogramming III

Dr. Yasir Al-Zubaidi

Third stage

2019

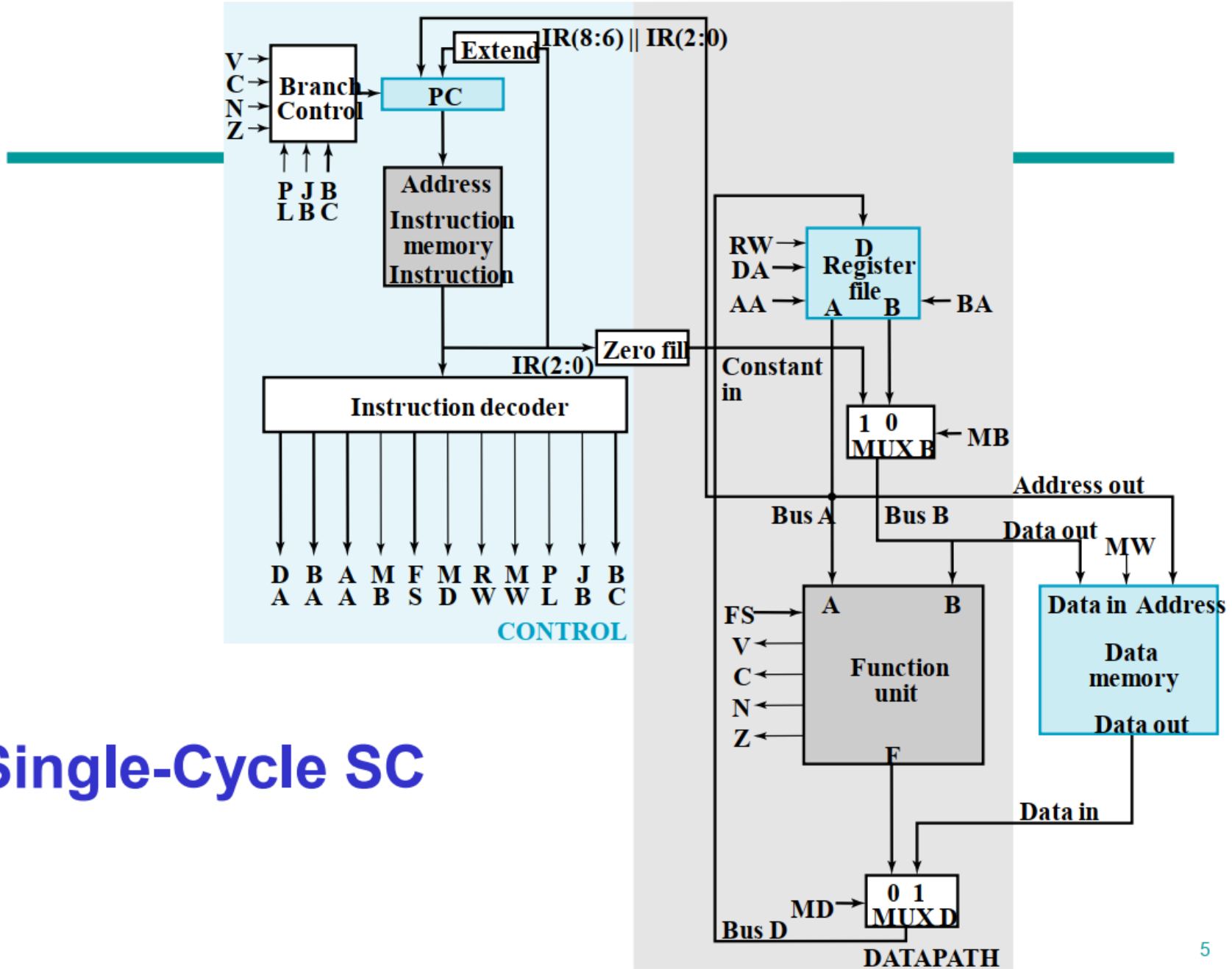
Single-Cycle Computer Issues

- Shortcoming of Single Cycle Design
  - Complexity of instructions executable in a single cycle is limited
  - Accessing both an instruction and data from a simple single memory impossible
  - A long worst case delay path limits clock frequency and the rate of performing instructions
- Handling of Shortcomings
  - The first two shortcomings can be handled by the multiple-cycle computer
  - The third shortcoming is dealt with by using a technique called pipelining described in later lectures

Multiple-Cycle Computer

- Converting the single-cycle computer into a multiple-cycle computer involves:
  - Modifications to the datapath/memory
  - Modification to the control unit
  - Design of a multiple-cycle hardwired control

Single-Cycle SC

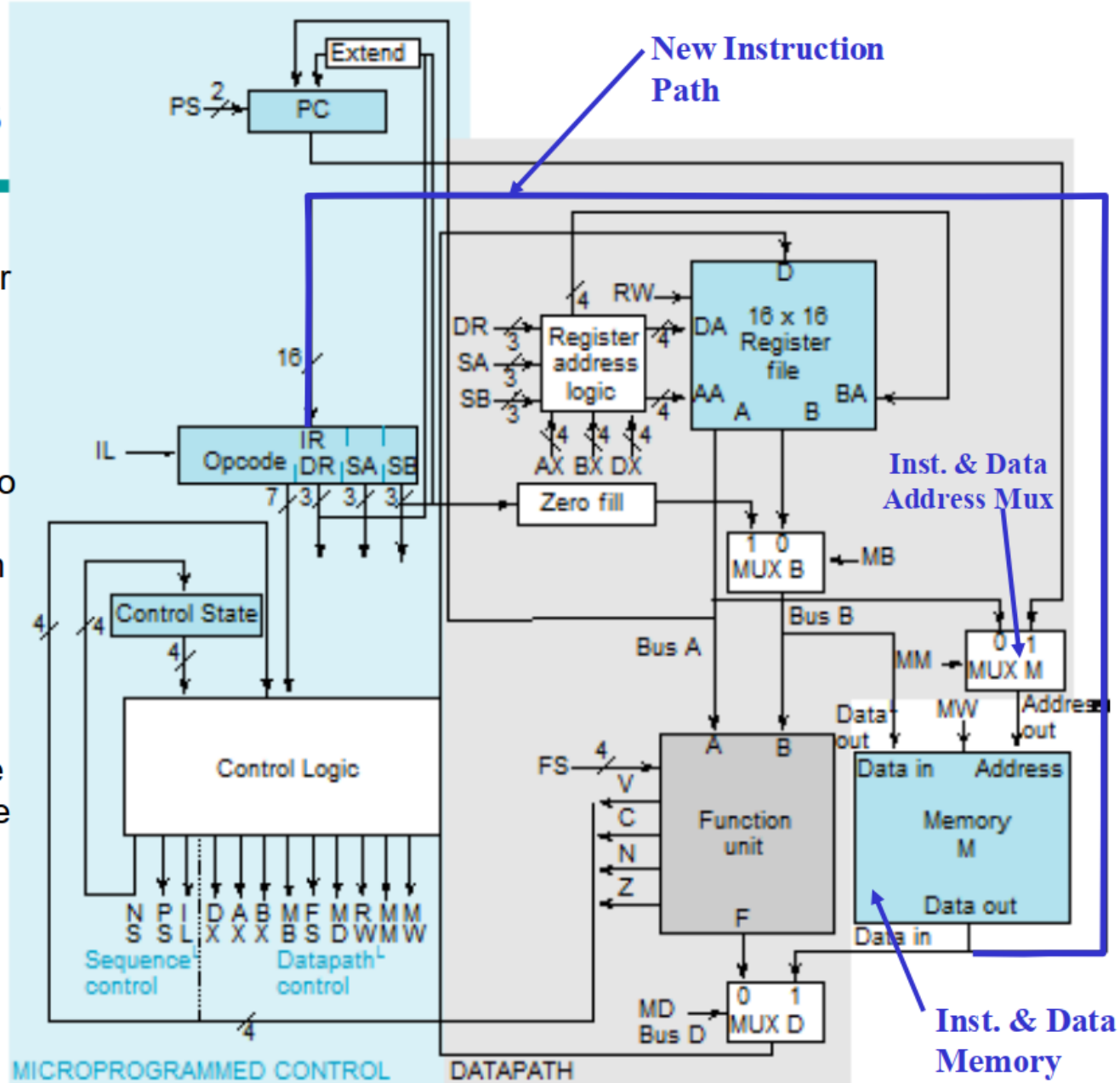


Datapath Modifications

Use a single memory for both instructions and data

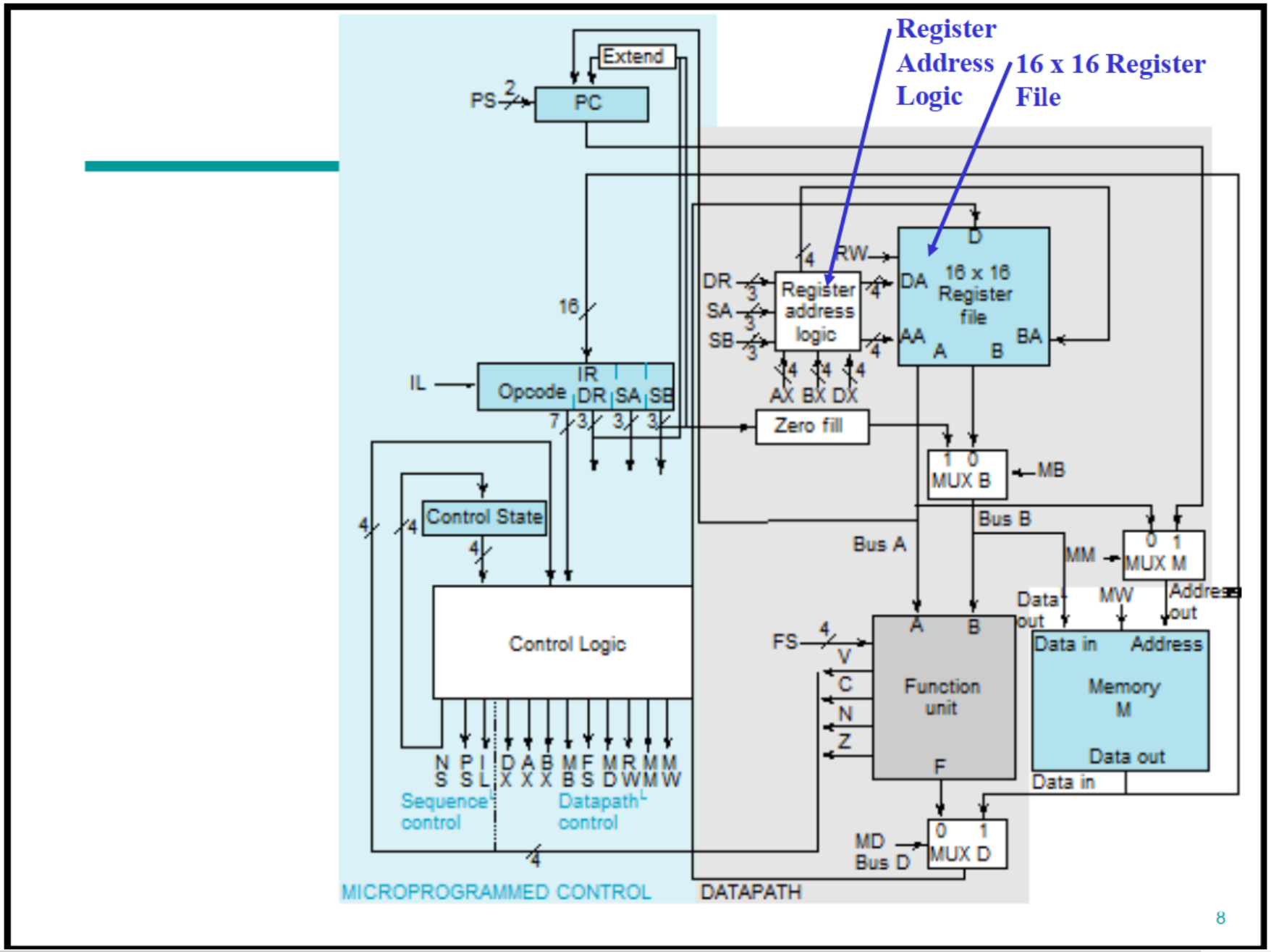
- Requires new MUX M with control signal MM to select between the instruction address from the PC and the data address

- Requires path from Memory Data Out to the instruction register in the control unit



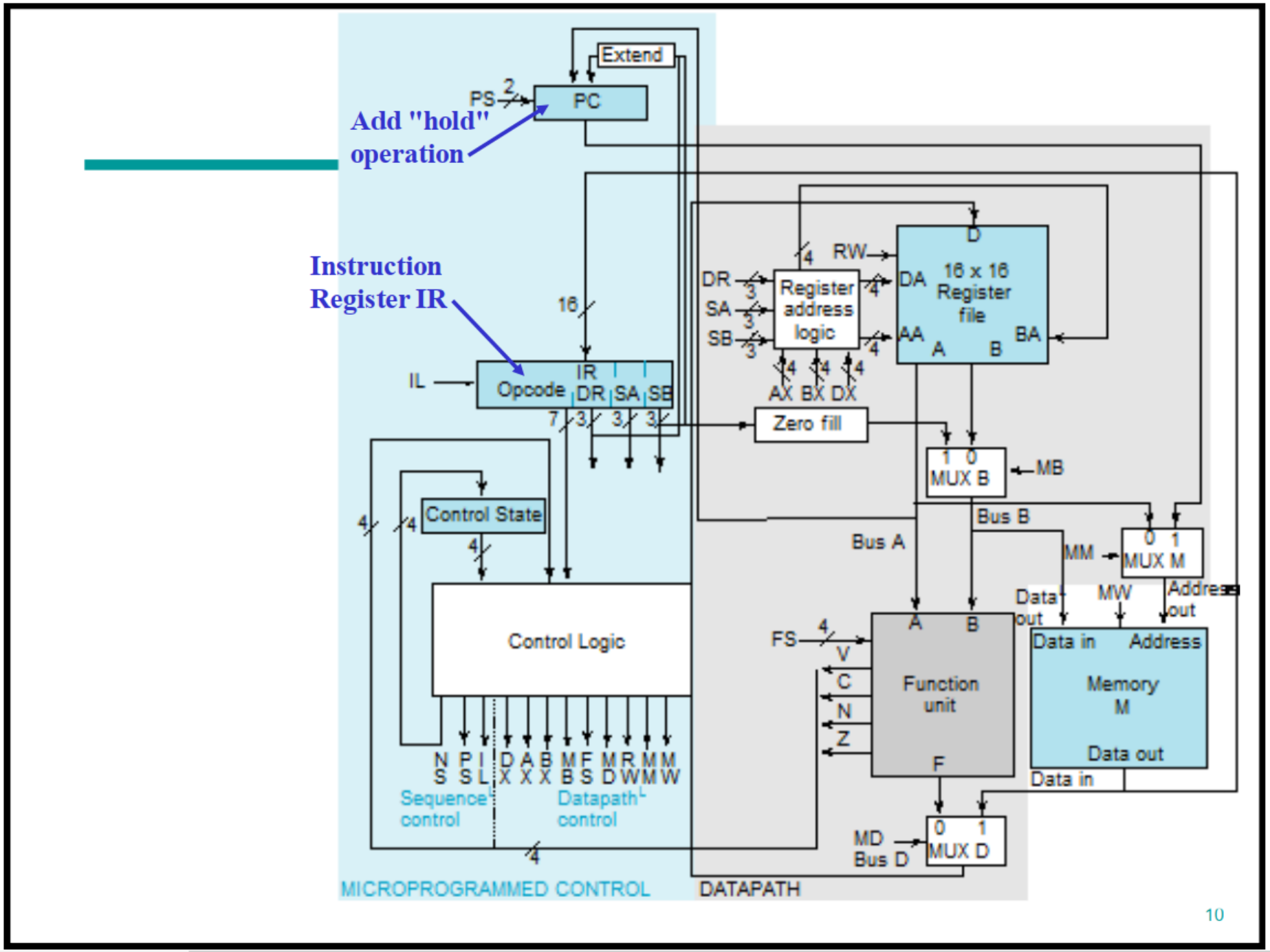
Datapath Modifications (Continued)

- Additional registers needed to hold operands between cycles
  - Add 8 temporary storage registers to the Register File
    - Register File becomes 16 x 16
    - Addresses to Register File increase from 3 to 4 bits
  - Register File addresses come from:
    - The instruction for the Storage Resource registers (0 to 7)
    - The control word for the Temporary Storage registers (8 to 15)
  - Add Register Address Logic to the Register File to select the register address sources
    - Three new control fields for register address source selection and temporary storage addressing: DX, AX, BX


Register Address Logic
16 x 16 Register File
Extend
PS
2
PC
16
IL
IR
Opcode
DR
SA
SB
7
3
3
3
Register address logic
RW
DR
SA
SB
4
3
3
3
4
AX BX DX
D
DA
16 x 16 Register file
AA
BA
A
B
Zero fill
1 0
MUX B
MB
Bus B
Bus A
MM
0 1
MUX M
Data out
MW
Address out
Control State
4
4
4
Control Logic
FS
4
V
C
N
Z
A
B
Function unit
F
Data in
Address
Memory M
Data out
Data in
0 1
MUX D
MD
Bus D
N S
P S
I L
D X
A X
B X
M B
F S
M D
R W
M M
M W
Sequence control
Datapath control
4
MICROPROGRAMMED CONTROL
DATAPATH

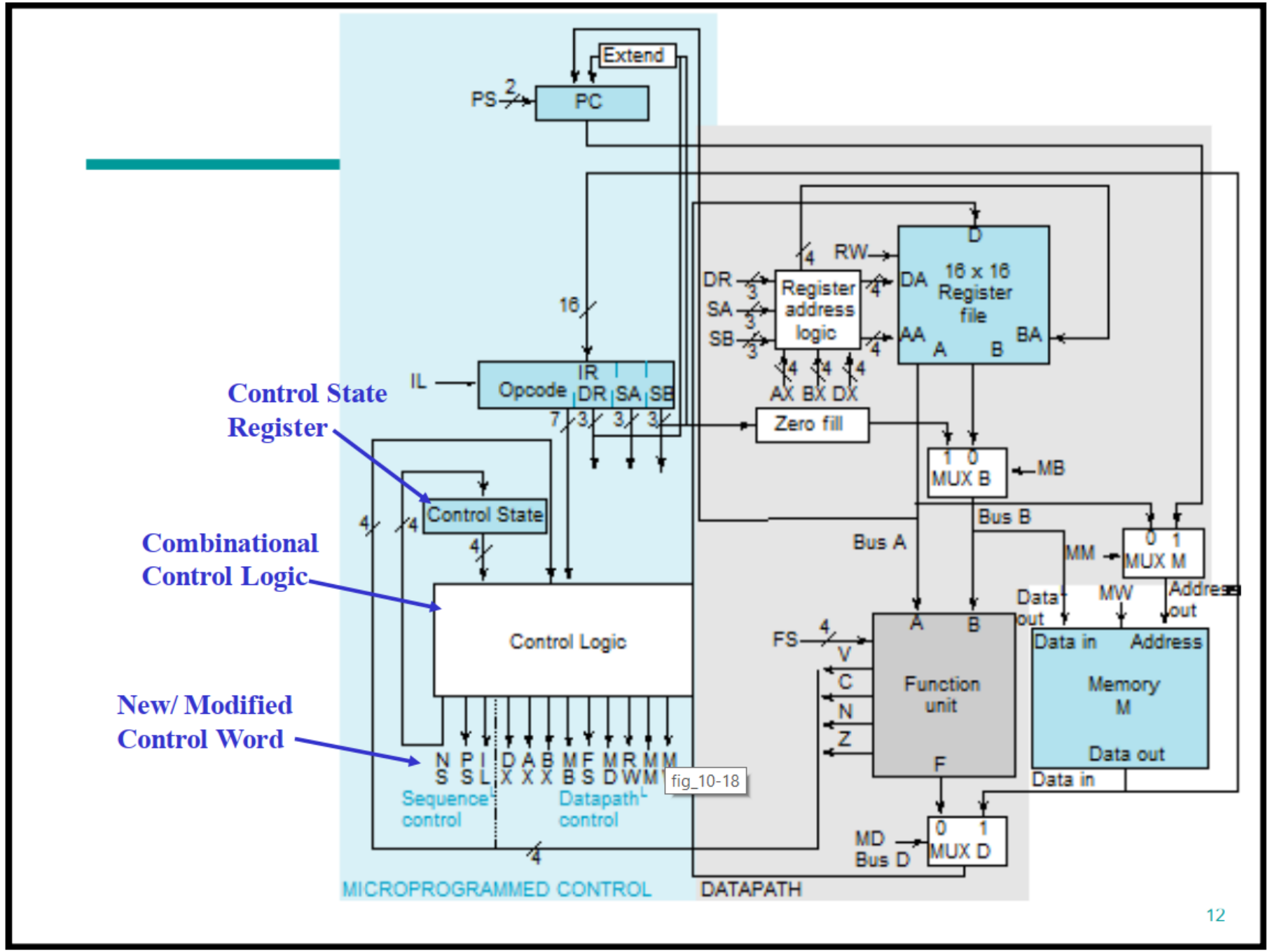
Control Unit Modifications

- Must hold instruction over the multiple cycles to draw on instruction information throughout instruction execution
  - Requires an Instruction Register (IR) to hold the instruction
    - Load control signal IL
  - Requires the addition of a "hold" operation to the PC since it only counts up to obtain a new instruction
    - New encoding for the PC operations uses 2 bits

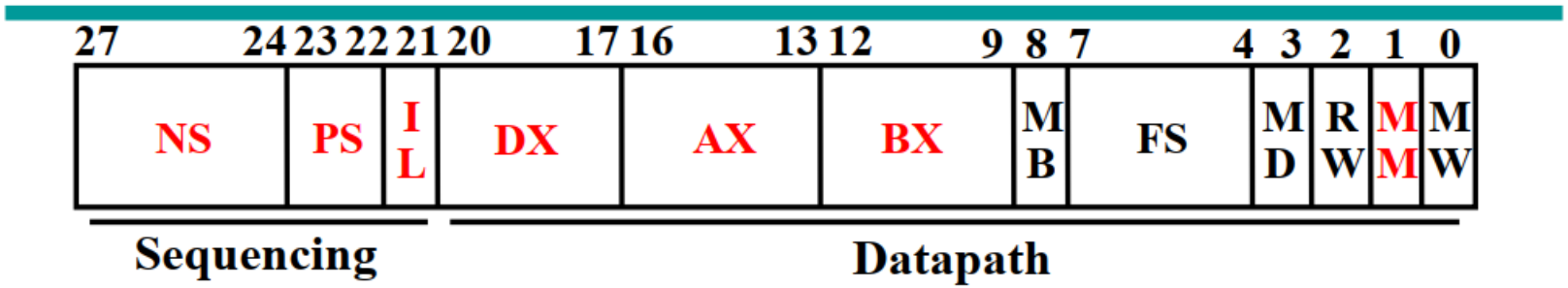

Add "hold" operation
Instruction Register IR
Extend
PS
2
PC
16
IL
IR
Opcode
DR
SA
SB
7
3
3
3
Zero fill
Control State
4
4
4
Control Logic
N S
P S
I L
D X
A X
B X
M B
F S
M D
R W
M M
W
Sequence control
Datapath control
4
MICROPROGRAMMED CONTROL
DR
SA
SB
3
3
3
Register address logic
4
RW
D
DA
16 x 16 Register file
AA
BA
A
B
AX
BX
DX
1 0
MUX B
MB
Bus B
Bus A
MM
0 1
MUX M
Data out
MW
Address out
FS
4
V
C
N
Z
A
B
Function unit
F
Data in
Address
Memory M
Data out
Data in
MD
Bus D
0 1
MUX D
DATAPATH

Sequential Control Design

- To control microoperations over multiple cycles, a Sequential Control replaces the Instruction Decoder
  - Input: Opcode, Status Bits, Control State
  - Output:
    - Control Word (Modified Datapath Control part)
    - Next State: Control Word (New Sequencing Control part)
  - Consists of:
    - Register to store the Control State
    - Combinational Logic to generate the Control Word (both sequencing and datapath control parts)
  - The Combinational Logic is quite complex so we assume that it is implemented by using a PLA or synthesized logic and focus on ASM level design


Extend
PS
2
PC
16
IL
IR
Opcode
DR
SA
SB
7
3
3
3
Control State Register
Control State
4
4
4
Combinational Control Logic
Control Logic
New/ Modified Control Word
N S
P S
I L
D X
A X
B X
M B
F S
M D
R W
M W
M M
Sequence control
Datapath control
fig_10-18
MICROPROGRAMMED CONTROL
DATAPATH
Zero fill
RW
DR
SA
SB
Register address logic
D
16 x 16 Register file
DA
AA
A
B
BA
AX BX DX
1 0
MUX B
MB
Bus A
Bus B
MM
0 1
MUX M
Data out
MW
Address out
FS
V
C
N
Z
Function unit
F
Data in
Address
Memory M
Data out
MD
Bus D
0 1
MUX D

Control Word



- Datapath part: field MM added, and fields DX, AX, and BX replace DA, AA, and BA, respectively
  - If the MSB of a field is 0, e.g., AX = 0XXX, then AA is 0 concatenated with SA (3bits) field in the IR
  - If the MSB of a field is 1, e. g. AX = 1011, then AA = 1011
- Sequencing part:
  - IL controls the loading of the IR
  - PS controls the operations of the PC
  - NS gives the next state of the Control State register
    - E.g., NS is 4 bits, the length of the Control State register - 16 states are viewed as adequate for this design

Encoding for Datapath Control

DX	AX	BX	Code	MB	Code	FS	Code	MD	RW	MM	MW	Code
R[DR]	R[SA]	R[SB]	0XXX	Register	0	$F \leftarrow A$	0000	FnUt	No write	Address Out	No write	0
R8	R8	R8	1000	Constant	1	$F \leftarrow A + 1$	0001	Data In	Write	PC	Write	1
R9	R9	R9	1001			$F \leftarrow A + B$	0010					
R10	R10	R10	1010			Unused	0011					
R11	R11	R11	1011			Unused	0100					
R12	R12	R12	1100			$F \leftarrow A + \overline{B} + 1$	0101					
R13	R13	R13	1101			$F \leftarrow A - 1$	0110					
R14	R14	R14	1110			Unused	0111					
R15	R15	R15	1111			$F \leftarrow A \wedge B$	1000					
						$F \leftarrow A \vee B$	1001					
						$F \leftarrow A \oplus B$	1010					
						$F \leftarrow \overline{A}$	1011					
						$F \leftarrow B$	1100					
						$F \leftarrow$ sr B	1101					
						$F \leftarrow$ sl B	1110					
						Unused	1111					

Encoding for Sequencing Control

NS	PS		IL	
Next State	Action	Code	Action	Code
Gives next state of Control State Register	Hold PC	00	No load	0
	Inc PC	01	Load instr.	1
	Branch	10		
	Jump	11		

ASM Charts for Sequential Control

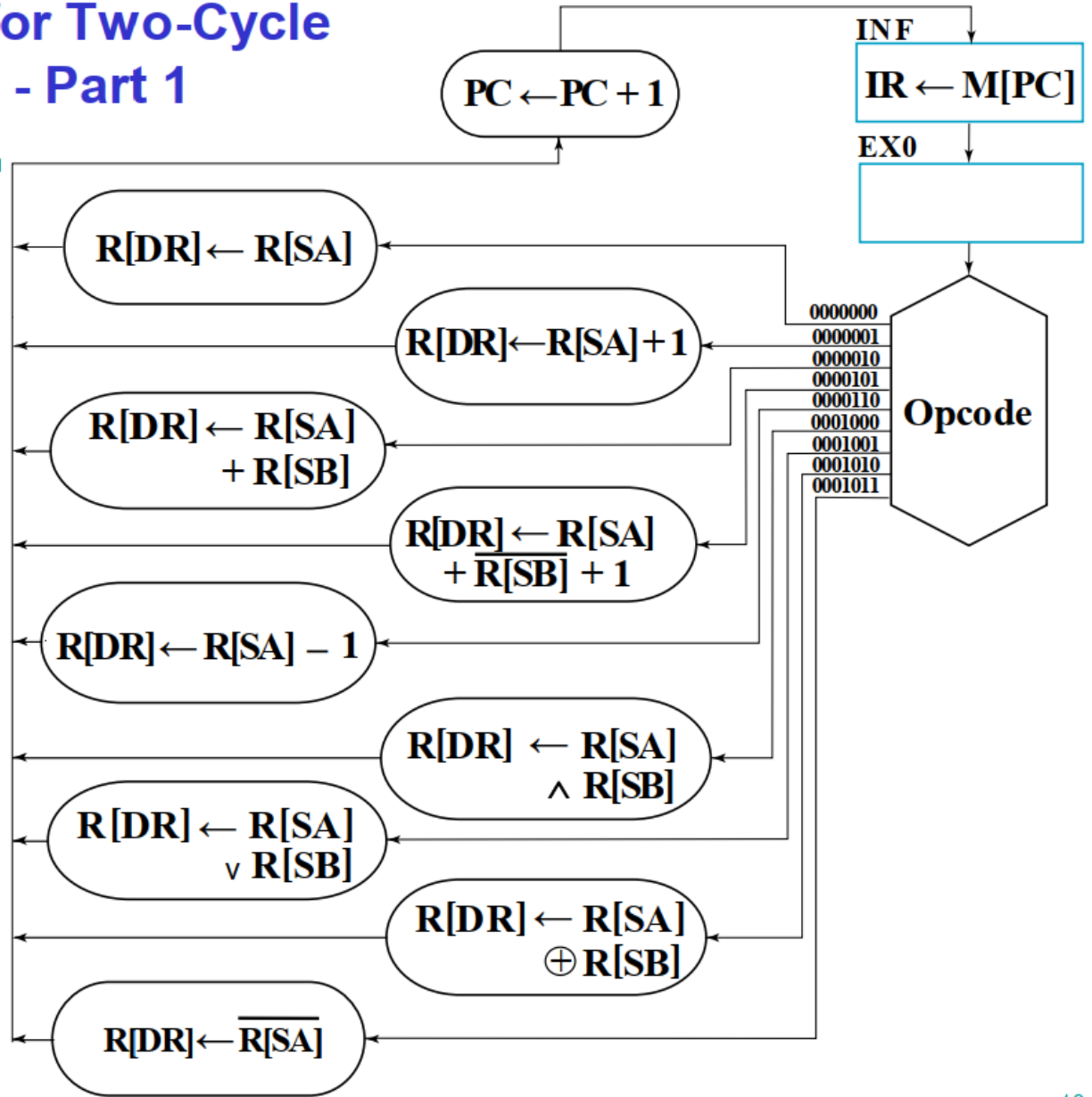
- An instruction requires two steps:
  - *Instruction fetch* – obtaining an instruction from memory
  - *Instruction execution* – the execution of a sequence of microoperations to perform instruction processing
  - Due to the use of the IR, these two steps require a minimum of two clock cycles
- ISA: Instruction Specifications and ASM charts for the instructions (that all require two clock cycles)
  - A vector decision box is used for the opcode
  - Scalar decision boxes are used for the status bits

ISA: Instruction Specifications (for reference)

Instruction Specifications for the Simple Computer - Part 1

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← $\overline{R[SA]}$	N, Z

ASM Chart for Two-Cycle Instructions - Part 1



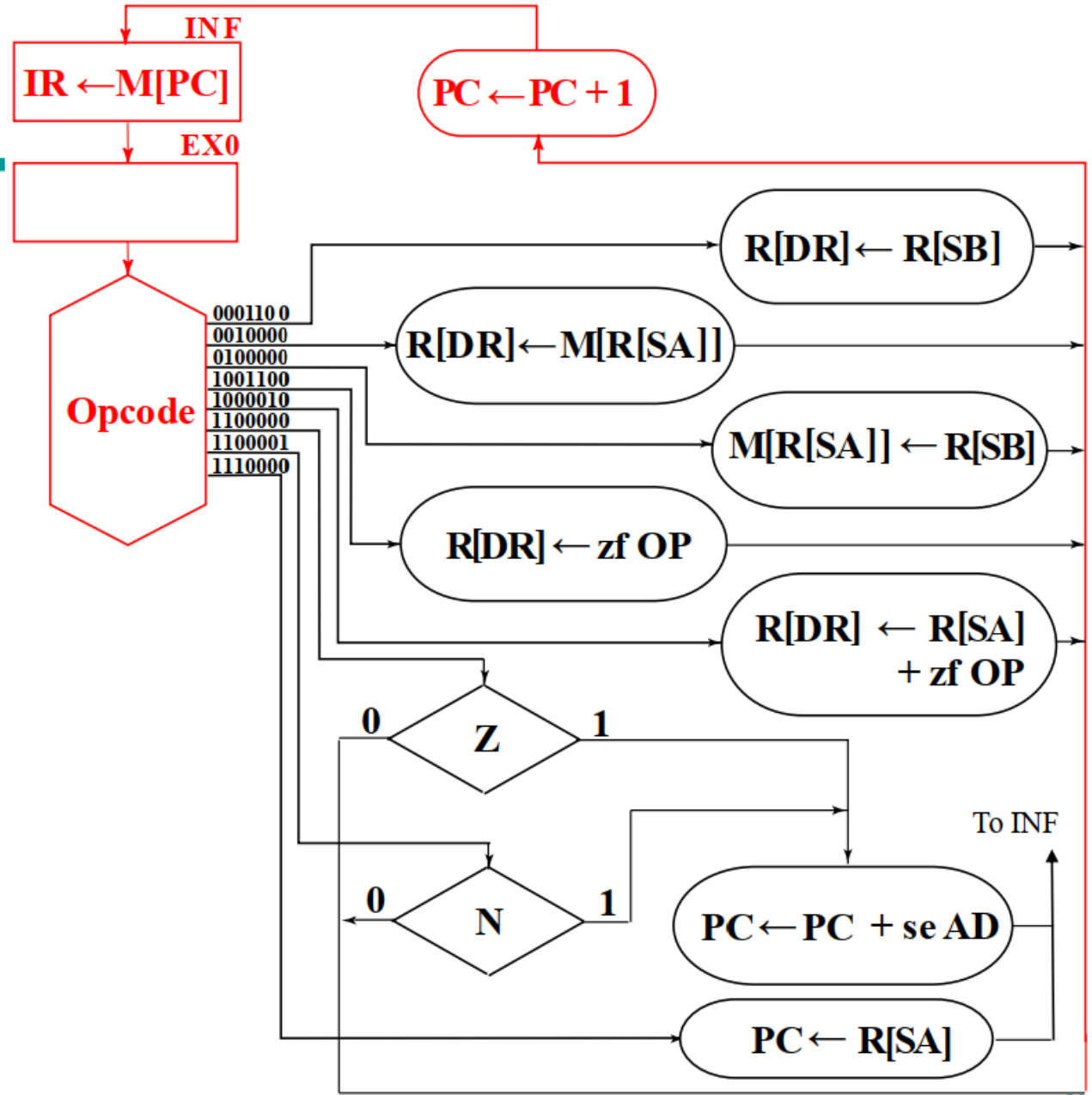
ISA: Instruction Specifications (for reference)

Instruction Specifications for the Simple Computer - Part 2

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[SA]	
Store	0100000	ST	RA,RB	M[SA] ← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se AD	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se AD	
Jump	1110000	JMP	RA	PC ← R[SA]	

ASM Chart for 2-Cycle Instruc-tions - Part 2

- Portion in Red duplicated from previous ASM chart



State Table for 2-Cycle Instructions

State	Inputs		Next state	Outputs												
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW		Comments
INF	XXXXXX	XXXX	EX0	1	00	XXX	XXX	XXX	X	XXX	X	0	1	0		IR ← M[PC]
EX0	0000000	XXXX	INF	0	01	0XX	0XX	XXX	X	0000	0	1	X	0	MOVA	R[DR] ← R[SA]*
EX0	0000001	XXXX	INF	0	01	0XX	0XX	XXX	X	0001	0	1	X	0	INC	R[DR] ← R[SA] + 1*
EX0	0000010	XXXX	INF	0	01	0XX	0XX	0XX	0	0010	0	1	X	0	ADD	R[DR] ← R[SA] + $\overline{R[SB]}$*
EX0	0000101	XXXX	INF	0	01	0XX	0XX	0XX	0	0101	0	1	X	0	SUB	R[DR} ← R[SA] + $\overline{R[SB]}$ + 1*
EX0	0000110	XXXX	INF	0	01	0XX	0XX	XXX	X	0110	0	1	X	0	DEC	R[DR] ← R[SA] + (- 1)*
EX0	0001000	XXXX	INF	0	01	0XX	0XX	0XX	0	1000	0	1	X	0	AND	R[DR] ← R[SA] ∧ R[SB]*
EX0	0001001	XXXX	INF	0	01	0XX	0XX	0XX	0	1001	0	1	X	0	OR	R[DR] ← R[SA] ∨ R[SB]*
EX0	0001010	XXXX	INF	0	01	0XX	0XX	0XX	0	1010	0	1	X	0	XOR	R[DR] ← R[SA] ⊕ R[SB]*
EX0	0001011	XXXX	INF	0	01	0XX	0XX	XXX	X	1011	0	1	X	0	NOT	R[DR] ← $\overline{R[SA]}$ *
EX0	0001100	XXXX	INF	0	01	0XX	XXX	0XX	0	1100	0	1	X	0	MOVB	R[DR] ← R[SB]*
EX0	0010000	XXXX	INF	0	01	0XX	0XX	XXX	X	XXX	1	1	0	0	LD	R[DR] ← M[R[SA]]*
EX0	0100000	XXXX	INF	0	01	XXX	0XX	0XX	0	XXX	X	0	0	1	ST	M[R[SA]] ← R[SB]*
EX0	1001100	XXXX	INF	0	01	0XX	XXX	XXX	1	1100	0	1	0	0	LDI	R[DR] ← zf OP*
EX0	1000010	XXXX	INF	0	01	0XX	0XX	XXX	1	0010	0	1	0	0	ADI	R[DR] ← R[SA] + zf OP*
EX0	1100000	XXX1	INF	0	10	XXX	0XX	XXX	X	0000	X	0	0	0	BRZ	PC ← PC + se AD
EX0	1100000	XXX0	INF	0	01	XXX	0XX	XXX	X	0000	X	0	0	0	BRZ	PC ← PC + 1
EX0	1100001	XXIX	INF	0	10	XXX	0XX	XXX	X	0000	X	0	0	0	BRN	PC ← PC + se AD
EX0	1100001	XX0X	INF	0	01	XXX	0XX	XXX	X	0000	X	0	0	0	BRN	PC ← PC + 1
EX0	1110000	XXXX	INF	0	11	XXX	0XX	XXX	X	0000	X	0	0	0	JMP	PC ← R[SA]

* For this state and input combinations, PC ← PC+1 also occurs

3-Process ASM VHDL Code

```
entity controller is
   Port ( opcode : in std_logic_vector(6 downto 0);
          reset, clk : in std_logic;
          zero, negative : in std_logic;
          IL, MB, MD, MM, RW, MW : out std_logic;
          PS : out std_logic_vector(1 downto 0);
          DX, AX, BX, FS : out std_logic_vector(3 downto 0);
);
end controller;

architecture Behavioral of controller is
  type state_type is (RES, FTH, EX);
 signal cur_state, next_state : state_type;
begin
              state_register:process(clk, reset)
              begin
                        if(reset='1') then
                                  cur_state<=RES;
                        elsif (clk'event and clk='1') then
                                  cur_state<=next_state;
                        end if;
              end process;
```

3-Process ASM VHDL Code

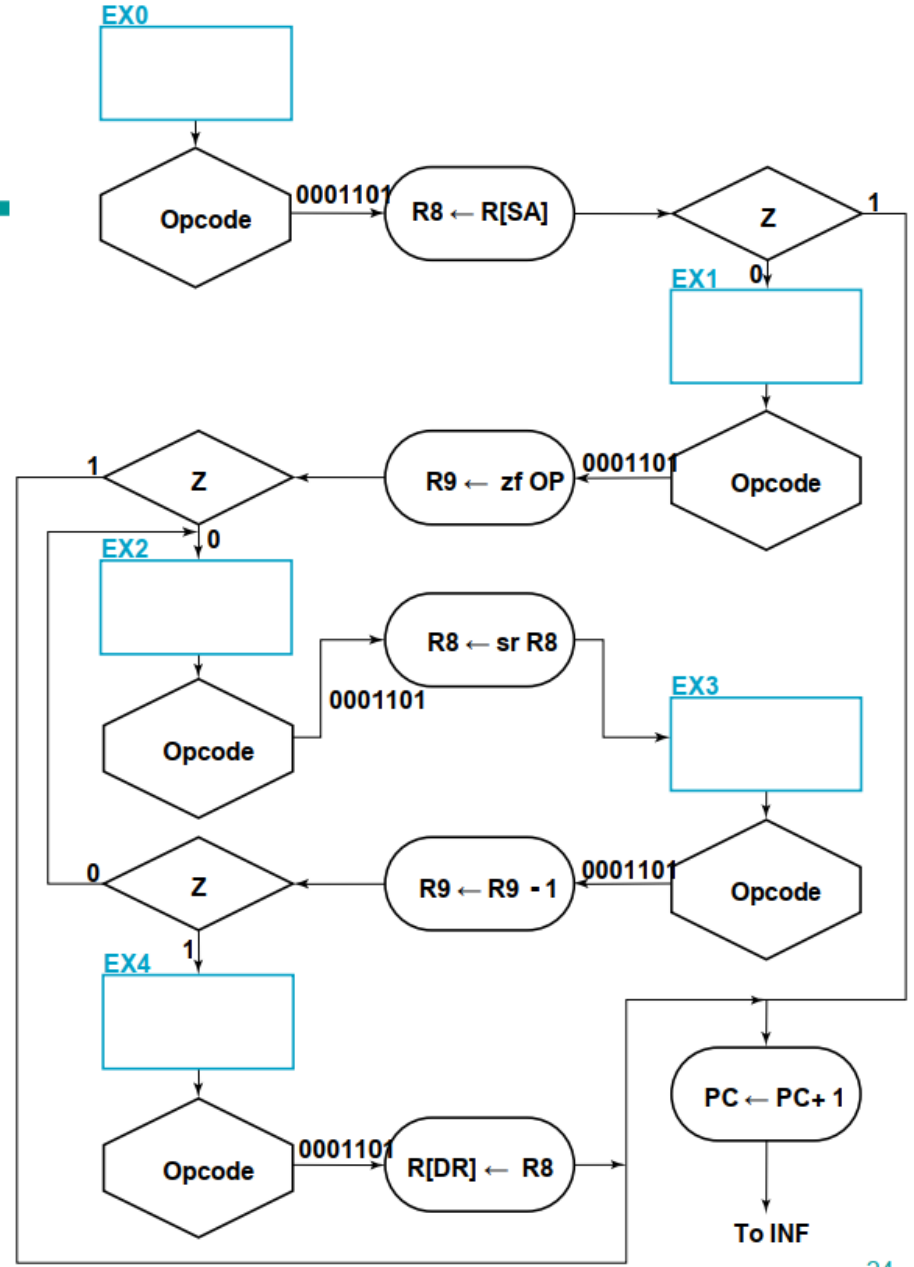
```
out_func: process (cur_state, opcode, zero, negative)
 begin
            (IL,PS, MB, FS, MD, RW, MW, MM) <= std_logic_vector'(0x"000");
            FS<="0000";
             case cur_state is
                        when RES =>
                          next_state <= FTH;
                        when FTH =>
                          -- set the control vector values
                          next_state <= EXE;
                        when EXE =>
                                    case opcde is:
                                       when “0000000” +>


end process;

End Behavioral;
```

ASM Chart for Multiple Bits Right Shift

- R8 – used to perform shifts
- R9 – used to store and decrement shift count
- Zero test in EX1 is to determine if the shift amount is 0; if so, goes to state INF



State Table For Multiple Bits Right Shift

State	Inputs		Next state	Outputs											Comments
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW	
EX0	0001101	XXX0	EX1	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	R8←R[SA], $\overline{Z}$:→EX1
EX0	0001101	XXX1	INF	0	01	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	R8←R[SA],Z:→INF*
EX1	0001101	XXX0	EX2	0	00	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	R9←zf OP, $\overline{Z}$:→EX2
EX1	0001101	XXX1	INF	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	R9←zf OP,Z:→INF*
EX2	0001101	XXXX	EX3	0	00	1000	XXXX	1000	0	1101	0	1	X	0	SRM	R8 ← sr R8,→EX3
EX3	0001101	XXX0	EX2	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	R9 ← R9 - 1, $\overline{Z}$:→EX2
EX3	0001101	XXX1	EX4	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	R9 ← R9 - 1,Z:→EX4
EX4	0001101	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	0	1	X	0	SRM	R[DR] ← R8,→ INF*

* For this state and input combinations, PC ← PC+1 also occurs

University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II
Asynchronous Sequential Logic

Dr. Yasir Al-Zubaidi

Third stage

2021

Outline

- **Asynchronous Sequential Circuits**
- Analysis Procedure
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Sequential Circuits

- Consist of a combinational circuit to which storage elements are connected to form a feedback path
- Specified by ***a time sequence of inputs, outputs,*** and ***internal states***
- Two types of sequential circuits:
  - Synchronous
  - Asynchronous

primary difference

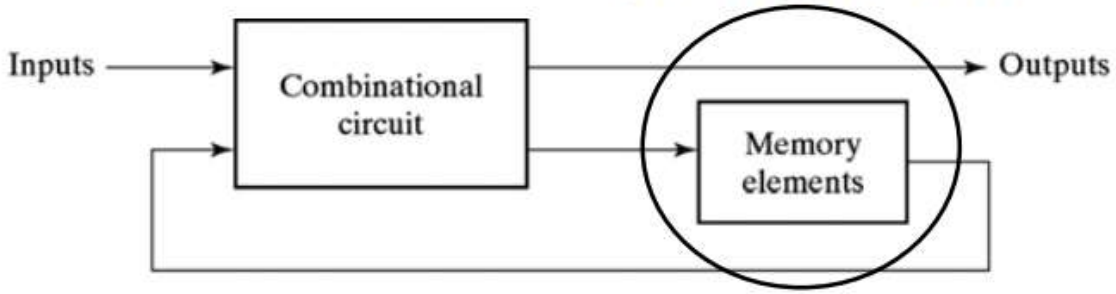
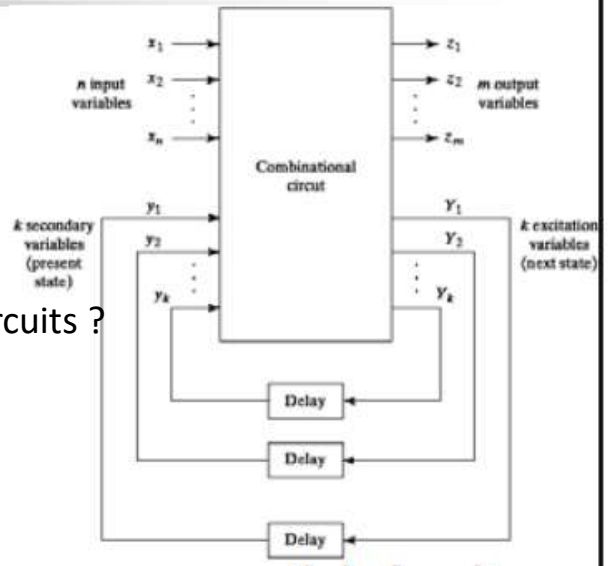


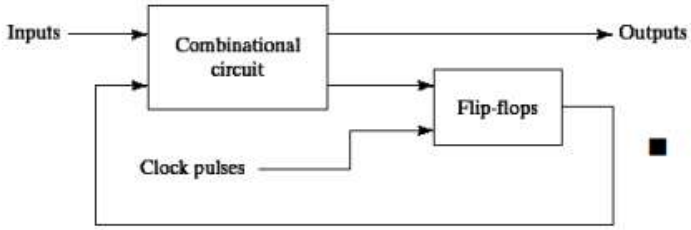
Fig. 5-1 Block Diagram of Sequential Circuit

Synchronous vs. Asynchronous

- **Asynchronous sequential circuits**
  - Internal states can change at ***any instant*** of time when there is a change in the input variables
  - ***No clock*** signal is required
  - Have better performance but hard to design due to timing problems



Why Asynchronous Circuits ?



(a) Block diagram

(b) Timing diagram of clock pulses

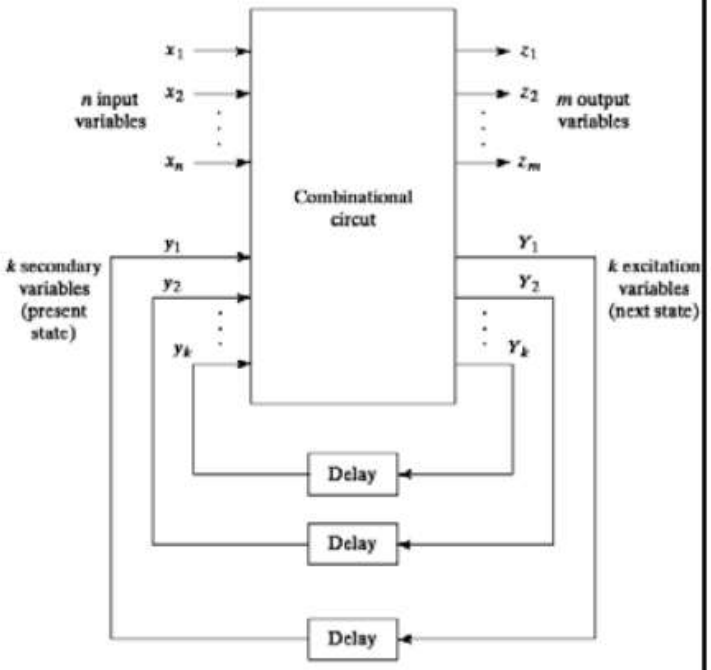
- **Synchronous sequential circuits**
  - Synchronized by a ***periodic*** train of clock pulses
  - Much easier to design (preferred design style)

Why Asynchronous Circuits ?

- Used when speed of operation is important
  - Response quickly without waiting for a clock pulse
- Used in small independent systems
  - Only a few components are required
- Used when the input signals may change independently of internal clock
  - Asynchronous in nature
- Used in the communication between two units that have their own independent clocks
  - Must be done in an asynchronous fashion

Definitions of Asyn. Circuits

- Inputs / Outputs
- Delay elements:
  - Only a short term memory
  - May not really exist due to original gate delay
- Secondary variable:
  - Current state (small y)
- Excitation variable:
  - Next state (big Y)
  - Have some delay in response to input changes



Operational Mode

- **Steady-state condition:**
  - **Current states and next states are the same**
  - **Difference between Y and y will cause a transition**
- **Fundamental mode:**
  - **No simultaneous changes of two or more variables**
  - **The time between two input changes must be longer than the time it takes the circuit to a stable state**
  - **The input signals change one at a time and only when the circuit is in a stable condition**

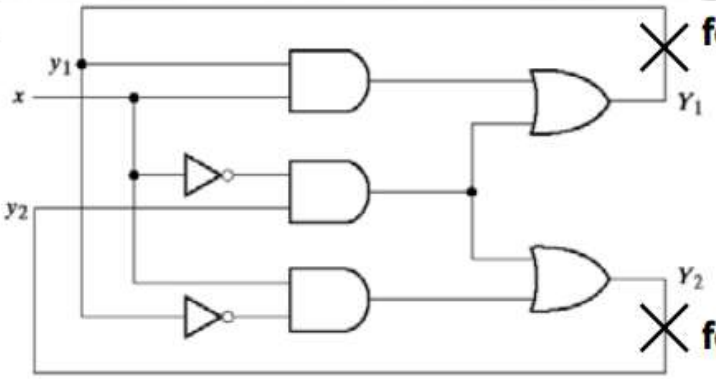
Outline

- Asynchronous Sequential Circuits
- **Analysis Procedure**
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Transition Table

- Transition table is useful to analyze an asynchronous circuit from the circuit diagram
- Procedure to obtain transition table:
  1. Determine all feedback loops in the circuits
  2. Mark the input (y_i) and output (Y_i) of each feedback loop
  3. Derive the Boolean functions of all Y's
  4. Plot each Y function in a map and combine all maps into one table
  5. Circle those values of Y in each square that are equal to the value of y in the same row

An Example of Transition Table



$$Y_1 = xy_1 + x'y_2$$

$$Y_2 = xy'_1 + x'y_2$$

$$Y = Y_1Y_2$$

inputs

current states

y_1y_2	$x=0$	$x=1$
00	0	0
01	1	0
11	1	1
10	0	1

(a) Map for $Y_1 = xy_1 + x'y_2$

y_1y_2	$x=0$	$x=1$
00	0	1
01	1	1
11	1	0
10	0	0

(b) Map for $Y_2 = xy'_1 + x'y_2$

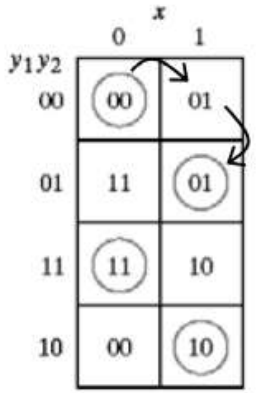
y_1y_2	$x=0$	$x=1$
00	(00)	01
01	11	(01)
11	(11)	10
(10)	00	(10)

(c) Transition table

stable !!

State Table

- When input x changes from 0 to 1 while y=00:
  - Y changes to 01 → unstable
  - y becomes 01 after a short delay → stable at the second row
  - The next state is Y=01
- Each row must have ***at least one*** stable state
- Analyze each state in this way can obtain its state table



(c) Transition table

Present State		Next State			
		X=0		X=1	
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	1	0

y_1y_2x : total state

4 stable total states: 000,011, 110,101

Flow Table

- Similar to a transition table except the states are represented by ***letter symbols***
- Can also include the output values
- Suitable to obtain the logic diagram from it
- Primitive flow table: ***only one*** stable state in each row (ex: 9-4(a))

	x = 0	x = 1
a	(a)	b
b	c	(b)
c	(c)	d
d	a	(d)

(a) Four states with one input

	x_1x_2 = 00	01	11	10
a	(a), 0	(a), 0	(a), 0	b , 0
b	a , 0	a , 0	(b), 1	(b), 0

(b) Two states with two inputs and one output

Equivalent to 9-3(c) if a=00, b=01, c=11, d=10

Flow Table to Circuits

- **Procedure to obtain circuits from flow table:**
  - **Assign to each state a distinct binary value (convert to a transition table)**
  - **Obtain circuits from the map**
- **Two difficulties:**
  - **The binary state assignment (to avoid race)**
  - **The output assigned to the unstable states**

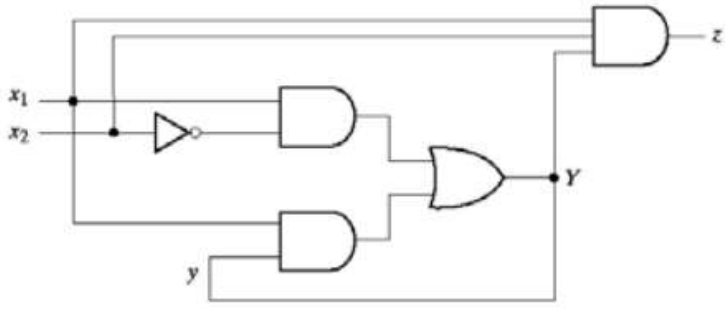
Ex: from the flow table 9-4(b)

y \ x_1x_2	00	01	11	10
0	(0)	(0)	(0)	1
1	0	0	(1)	(1)

(a) Transition table
$Y = x_1x'_2 + x_1y$

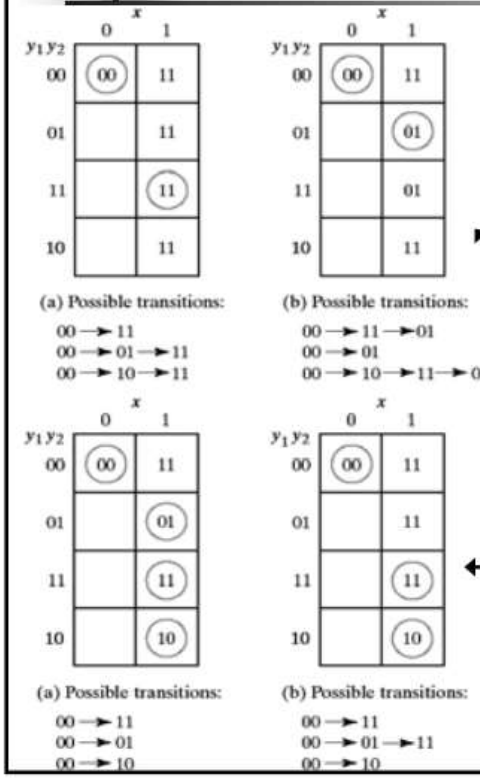
y \ x_1x_2	00	01	11	10
0	0	0	0	0
1	0	0	1	0

(b) Map for output
$z = x_1x_2y$



(c) Logic diagram

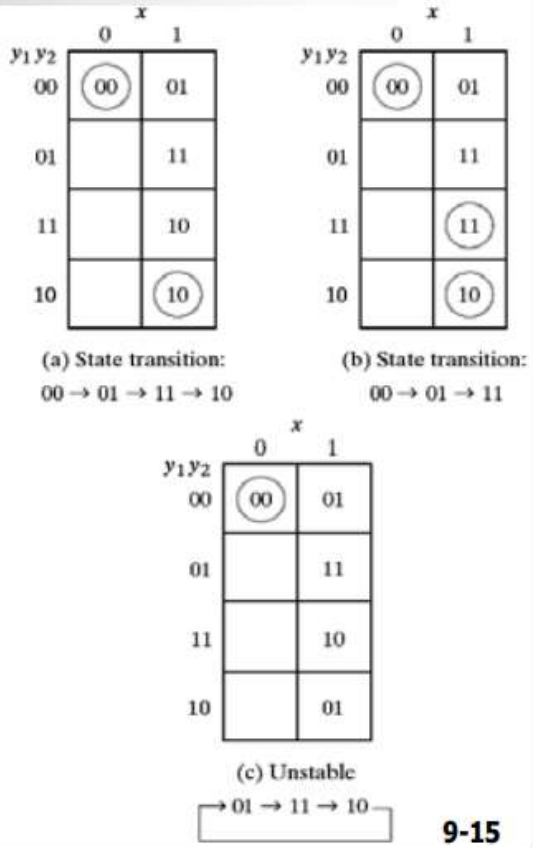
Race Conditions



- **Race condition:**
  - ***two or more*** **binary state variables will change value when one input variable changes**
  - **Cannot predict state sequence if unequal delay is encountered**
- **Non-critical race:**
  - **The final stable state *does not* depend on the change order of state variables**
- **Critical race:**
  - **The change order of state variables will result in *different* stable states**
  - **Should be avoided !!**

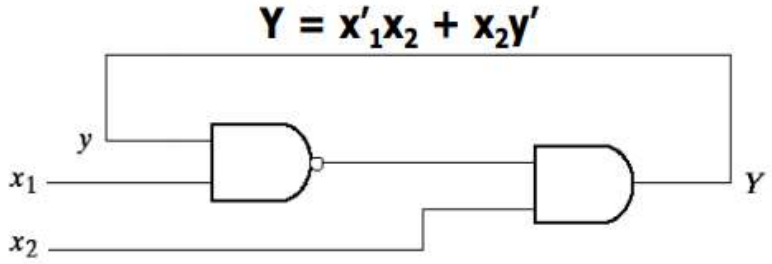
Race-Free State Assignment

- Race can be avoided by proper state assignment
  - Direct the circuit through intermediate unstable states with a unique state-variable change
  - It is said to have a ***cycle***
- Must ensure that a cycle will terminate with a stable state
  - Otherwise, the circuit will keep going in unstable states
- More details will be discussed in Section 9-6



Stability Check

- Asynchronous sequential circuits may oscillate between unstable states due to the feedback
  - Must check for stability to ensure proper operations
- Can be easily checked from the transition table
  - Any column has no stable states → unstable
  - Ex: when x_1x_2=11 in Fig. 9-9(b), Y and y are never the same



(a) Logic diagram

y \ x_1x_2	00	01	11	10
0	(0)	1	1	(0)
1	0	(1)	0	0

(b) Transition table

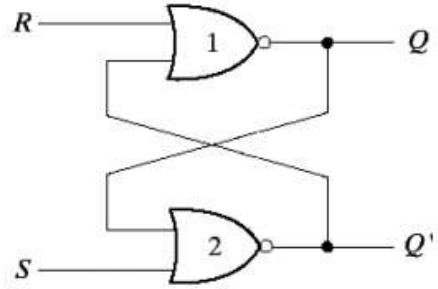
Outline

- Asynchronous Sequential Circuits
- Analysis Procedure
- **Circuits with Latches**
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Latches in Asynchronous Circuits

- The traditional configuration of asynchronous circuits is using one or more feedback loops
  - No real delay elements
- It is more convenient to employ the SR latch as a memory element in asynchronous circuits
  - Produce an orderly pattern in the logic diagram with the memory elements clearly visible
- SR latch is also an asynchronous circuit
  - Will be analyzed first using the method for asynchronous circuits

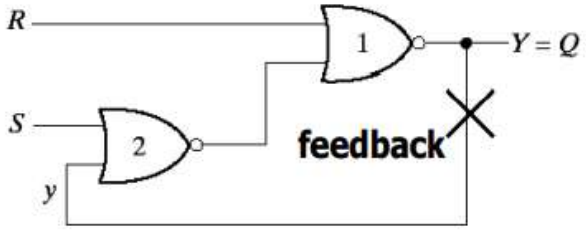
SR Latch with NOR Gates



(a) Crossed-coupled circuit

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(After $SR = 10$)
0	1	0	1	
0	0	0	1	(After $SR = 01$)
1	1	0	0	

(b) Truth table



(c) Circuit showing feedback

y \ SR	00	01	11	10
0	0	0	0	1
1	1	0	0	1

$Y = SR' + R'y$

$Y = S + R'y$ when $SR = 0$ → *

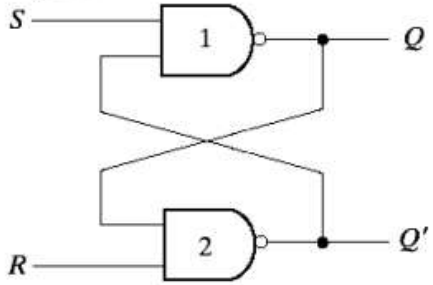
(d) Transition table

S=1, R=1 (SR = 1) should not be used

⇒ SR = 0 is normal mode

should be carefully checked first

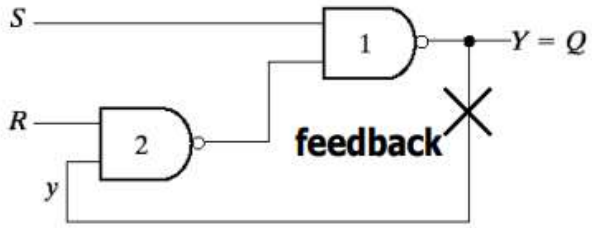
SR Latch with NAND Gates



(a) Crossed-coupled circuit

S	R	Q	Q′	
1	0	0	1	
1	1	0	1	(After $SR = 10$)
0	1	1	0	
1	1	1	0	(After $SR = 01$)
0	0	1	1	

(b) Truth table



(c) Circuit showing feedback

y \ SR	00	01	11	10
0	1	1	0	0
1	1	1	1	0

$Y = S' + Ry$ when $S'R' = 0$

(d) Transition table

S=0, R=0 (S′R′ = 1) should not be used
⇒ S′R′ = 0 is normal mode

*** should be carefully checked first**

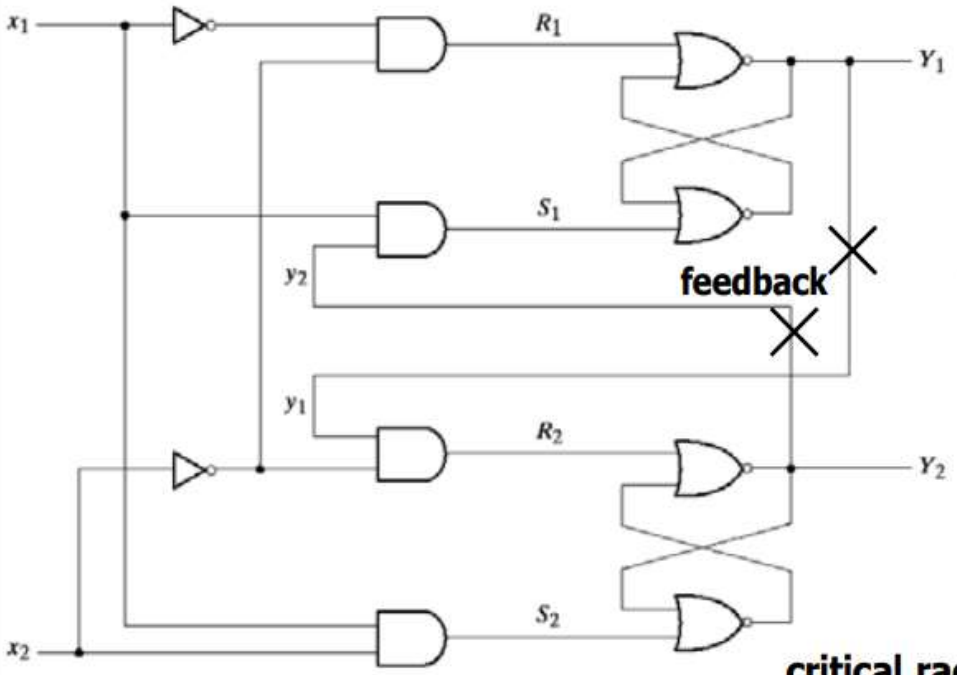
Analysis Procedure

- **Procedure to analyze an asynchronous sequential circuits with SR latches:**
  1. Label each latch output with Y_i and its external feedback path (if any) with y_i
  2. Derive the Boolean functions for each S_i and R_i
  3. Check whether $SR=0$ (NOR latch) or $S'R'=0$ (NAND latch) is satisfied
  4. Evaluate $Y=S+R'y$ (NOR latch) or $Y=S'+Ry$ (NAND latch)
  5. Construct the transition table for $Y=Y_1Y_2\ldots Y_k$
  6. Circle all stable states where $Y=y$

Analysis Example

$S_1=x_1y_2$ $R_1=x'_1x'_2 \Rightarrow S_1R_1 = x_1y_2x'_1x'_2 = 0$ (OK)
$S_2=x_1x_2$ $R_2=x'_2y_1 \Rightarrow S_2R_2 = x_1x_2x'_2y_1 = 0$ (OK)

$Y_1=S_1 + R'_1y_1$
$=x_1y_2 + (x_1+x_2)y_1$
$=x_1y_2+x_1y_1+x_2y_1$
$Y_2=S_2 + R'_2y_2$
$=x_1x_2 + (x_2+y'_1)y_2$
$=x_1x_2+x_2y_2+y'_1y_2$



y_1y_2 \ x_1x_2	00	01	11	10
00	00	00	01	00
01	01	01	11	11
11	00	11	11	10
10	00	10	11	10

critical race !!

Implementation Procedure

- **Procedure to implement an asynchronous sequential circuits with SR latches:**
  1. **Given a transition table that specifies the excitation function $Y = Y_1 Y_2 \ldots Y_k$, derive a pair of maps for each S_i and R_i using the latch excitation table**
  2. **Derive the Boolean functions for each S_i and R_i**
     (do not to make Si and Ri equal to 1 in the same minterm square)
  3. **Draw the logic diagram using *k* latches together with the gates required to generate the S and R**
     (for NAND latch, use the complemented values in step 2)

Implementation Example

Excitation table: list the required S and R for each possible transition from y to Y

x_1x_2	00	01	11	10
y = 0	(0)	(0)	(0)	1
y = 1	(0)	0	(1)	(1)

(a) Transition table

$Y = x_1x'_2 + x_1y$

y	Y	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	1

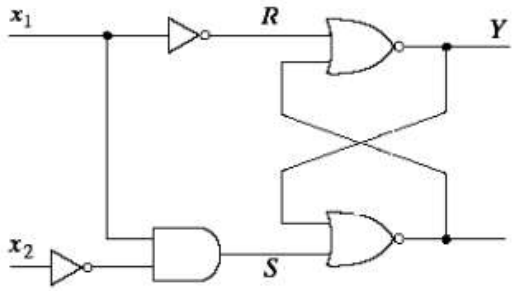
(b) Latch excitation table

x_1x_2	00	01	11	10
y = 0	0	0	0	1
y = 1	0	0	X	X

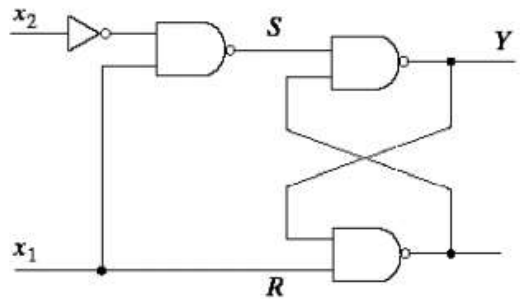
(c) Map for $S = x_1x'_2$

x_1x_2	00	01	11	10
y = 0	X	X	X	0
y = 1	1	1	0	0

(d) Map for $R = x'_1$



(e) Circuit with NOR latch



(f) Circuit with NAND latch

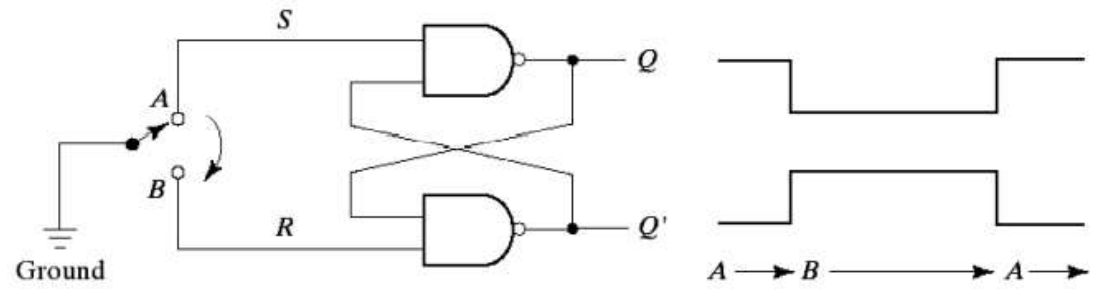
y = 1 (outside) → 0 (inside)

∴ S=0, R=1 from excitation table

Debounce Circuit

- Mechanical switches are often used to generate binary signals to a digital circuit
  - It may vibrate or bounce several times before going to a final rest
  - Cause the signal to oscillate between 1 and 0
- A debounce circuit can remove the series of pulses from a contact bounce and produce a single smooth transition
  - Position A (SR=01) → bouncing (SR=11) → Position B (SR=10)

    Q = 1 (set) → Q = 1 (no change) → Q = 0 (reset)



University Of Diyala
College Of Engineering
Department of Computer Engineering

Digital System Design II
Asynchronous Sequential Logic
Part II

Dr. Yasir Al-Zubaidi

Third stage

2021

Outline

- Asynchronous Sequential Circuits
- Analysis Procedure
- Circuits with Latches
- **Design Procedure**
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Design Procedure

1. Obtain a primitive flow table from the given design specifications
2. Reduce the flow table by merging rows in the primitive flow table
3. Assign binary state variables to each row of the reduced flow to obtain the transition table.
4. Assign output values to the dashes associated with the unstable states to obtain the output map.
5. Simplify the Boolean functions of the excitation and output variables and draw the logic diagram

Primitive Flow Table

- Design example: gated latch
- Two Input (G = gate , D = Data).
- One output is Q , the gated latch is a memory element that;
  - Accept the value of D when G=1
  - Retain this value after G goes to 0 (D has no effects now)
  - Obtain the flow table by listing all possible states.
  - Dash marks are given when both inputs change simultaneously
  - Outputs of unstable states are don't care

	Input		Output	
State	D	G	Q	Comments
a	0	1	0	D=Q because G=1
b	1	1	1	D=Q because G=1
c	0	0	0	After states a or d
d	1	0	0	After state c
e	1	0	1	After states b or f
f	0	0	1	After state e

	DG 00	01	11	10
a	c , –	(a) , 0	b , –	– , –
b	– , –	a , –	(b) , 1	e , –
c	(c) , 0	a , –	– , –	d , –
d	c , –	– , –	b , –	(d) , 0
e	f , –	– , –	b , –	(e) , 1
f	(f) , 1	a , –	– , –	e , –

Reduce the Flow Table

- Two or more rows can be merged into one row if there are ***non-conflicting*** states and outputs in ***every*** columns
- After merged into one row:
  - Don't care entries are overwritten
  - Stable states and output values are included
  - A common symbol is given to the merged row
- Formal reduction procedure is given in next section

DG

	00	01	11	10
a	*c*, –	(*a*), 0	*b*, –	–, –
c	(*c*), 0	*a*, –	–, –	*d*, –
d	*c*, –	–, –	*b*, –	(*d*), 0

DG

	00	01	11	10
b	–, –	*a*, –	(*b*), 1	*e*, –
e	*f*, –	–, –	*b*, –	(*e*), 1
f	(*f*), 1	*a*, –	–, –	*e*, –

(a) States that are candidates for merging

DG

	00	01	11	10
a, *c*, *d*	(*c*), 0	(*a*), 0	*b*, –	(*d*), 0
b, *e*, *f*	(*f*), 1	*a*, –	(*b*), 1	(*e*), 1

DG

	00	01	11	10
a	(*a*), 0	(*a*), 0	*b*, –	(*a*), 0
b	(*b*), 1	*a*, –	(*b*), 1	(*b*), 1

(b) Reduced table (two alternatives)

Transition Table and Logic Diagram

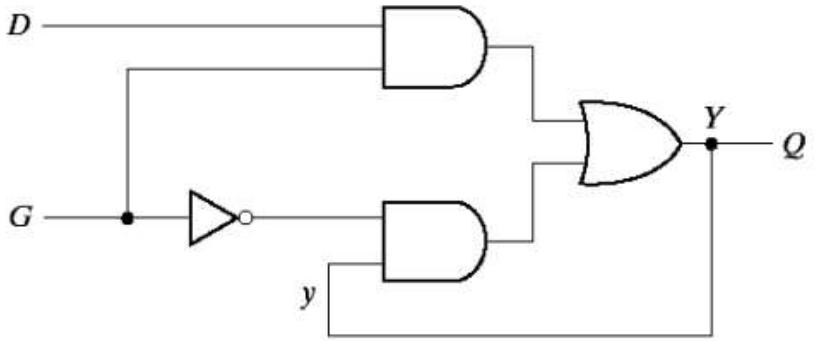
y \ DG	00	01	11	10
0	0	0	1	0
1	1	0	1	1

(a) $Y = DG + G'y$

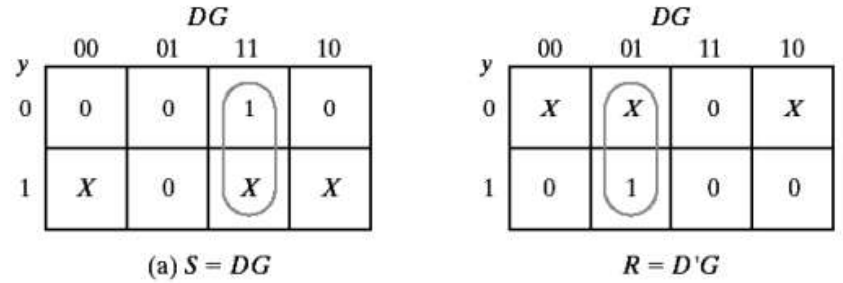
y \ DG	00	01	11	10
0	0	0	1	0
1	1	0	1	1

(b) $Q = Y$

- **Assign a binary value to each state to generate the transition table**
  - **a=0, b=1 in this example**
- **Directly use the simplified Boolean function for the excitation variable Y**
  - **An asynchronous circuit without latch is produced**



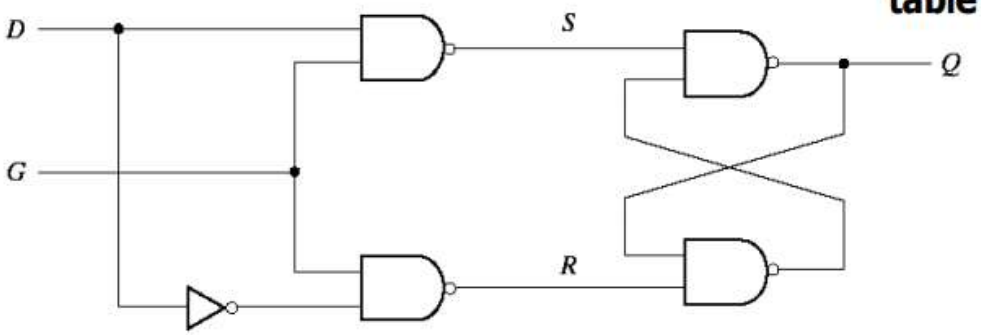
Implementation with SR Latch



(a) $S = DG$ $R = D'G$

(a) Maps for S and R

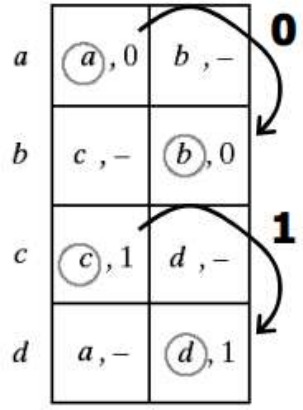
Listed according to the transition table and the excitation table of SR latch



(b) Logic diagram

Outputs for Unstable States

- **Objective: no momentary false outputs occur when the circuit switches between stable states**
- **If the output value is not changed, the intermediate unstable state must have the same output value**
  - **0 → 1 (unstable) → 0 (X)**
  - **0 → 0 (unstable) → 0 (O)**
- **If the output value changed, the intermediate outputs are don't care**
  - **It makes no difference when the output change occurs**



(a) Flow table

0	(0)
X	0
1	(1)
X	1

(b) Output assignment

Outline

- Asynchronous Sequential Circuits
- Analysis Procedure
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

State Reduction

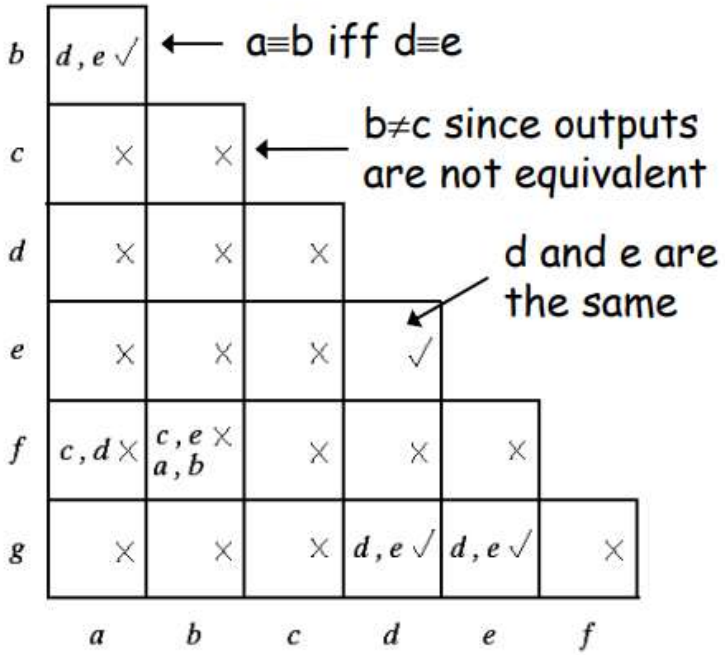
- Two states are equivalent if they have the ***same output*** and go to the ***same*** (equivalent) ***next states*** for each possible input
  - Ex: (a,b) are equivalent
    (c,d) are equivalent
- State reduction procedure is similar in both sync. & async. sequential circuits
  - For completely specified state tables:
    → use implication table
  - For incompletely specified state tables:
    → use compatible pairs

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
a	c	b	0	1
b	d	a	0	1
c	a	d	1	0
d	b	d	1	0

Implication Table Method (1/2)

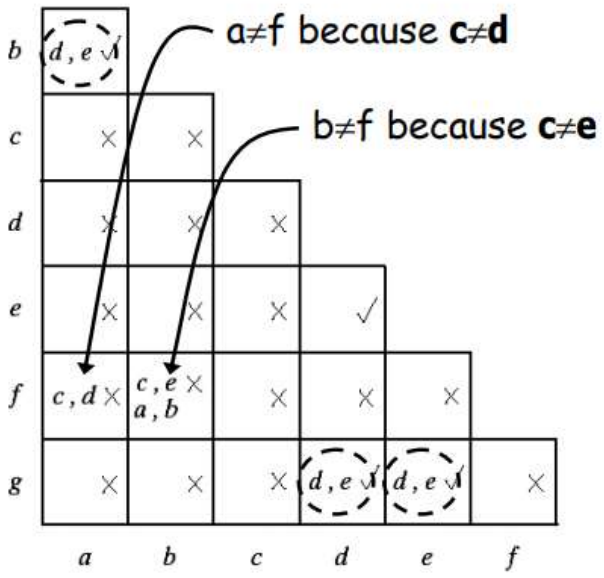
- **Step 1: build the implication chart**

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
a	d	b	0	0
b	e	a	0	0
c	g	f	0	1
d	a	d	1	0
e	a	d	1	0
f	c	b	0	0
g	a	e	1	0



Implication Table Method (2/2)

- **Step 2: delete the node with unsatisfied conditions**
- **Step 3: repeat Step 2 until equivalent states found**



equivalent states :
(a,b) (d,e) (d,g) (e,g)

d == e == g

Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
a	d	a	0	0
c	d	f	0	1
d	a	d	1	0
f	c	a	0	0

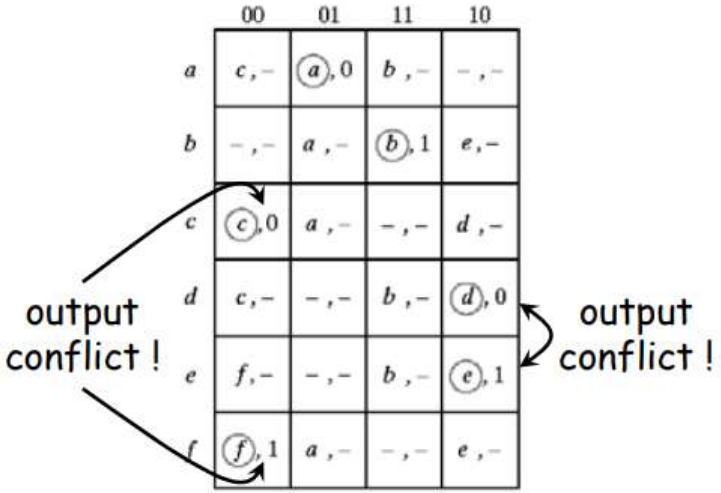
Reduced State Table

Merge the Flow Table

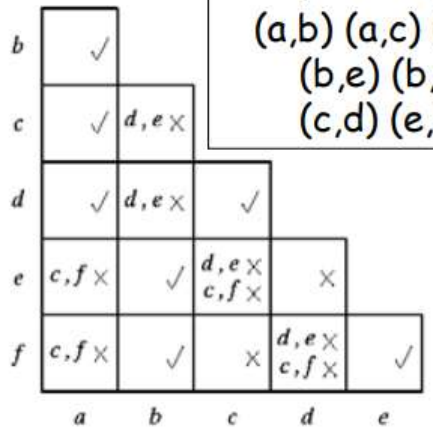
- The state table may be incompletely specified
  - Some next states and outputs are don't care
- Primitive flow tables are always incompletely specified
  - Several synchronous circuits also have this property
- Incompletely specified states are not "equivalent"
  - Instead, we are going to find "***compatible***" states
  - Two states are compatible if they have the ***same output*** and ***compatible next states*** whenever specified
- Three procedural steps:
  - Determine all compatible pairs
  - Find the maximal compatibles
  - Find a minimal closed collection of compatibles

Compatible Pairs

- **Implication tables are used to find compatible states**
  - **We can adjust the dashes to fit any desired condition**
  - **Must have *no conflict* in the output values to be merged**



(a) Primitive flow table

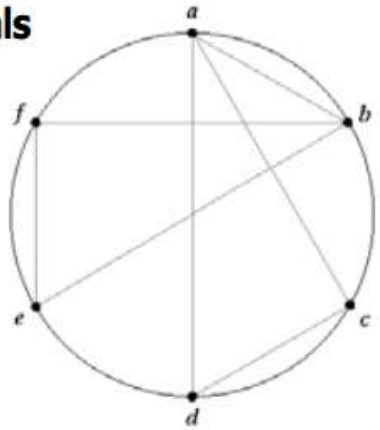


(b) Implication table

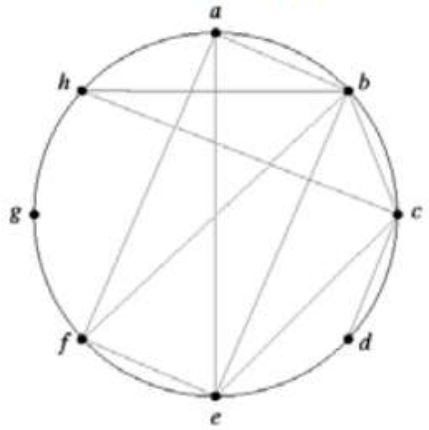
compatible pairs :
(a,b) (a,c) (a,d)
(b,e) (b,f)
(c,d) (e,f)

Maximal Compatibles

- A group of compatibles that contains all the possible combinations of compatible states
  - Obtained from a merger diagram
  - A line in the diagram represents that two states are compatible
- n-state compatible → n-sided fully connected polygon
  - All its diagonals connected
- Not all maximal compatibles are necessary



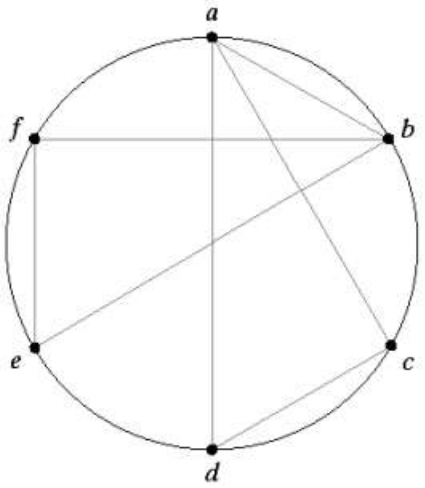
(a) Maximal compatible:
$(a, b,)\ (a, c, d)\ (b, e, f)$



(b) Maximal compatible:
$(a, b, e, f)\ (b, c, h)\ (c, d)\ (g)$

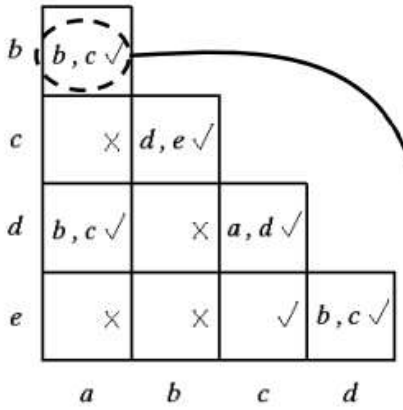
Closed Covering Condition

- The set of chosen compatibles must cover all the states and must be closed
  - Closed covering
- The closure condition is satisfied if
  - There are no implied states
  - The implied states are included within the set
- Ex: if remove (a,b) in the right
  - (a,c,d) (b,e,f) are left in the set
  - All six states are still included
  - No implied states according to its implication table 9-23(b)

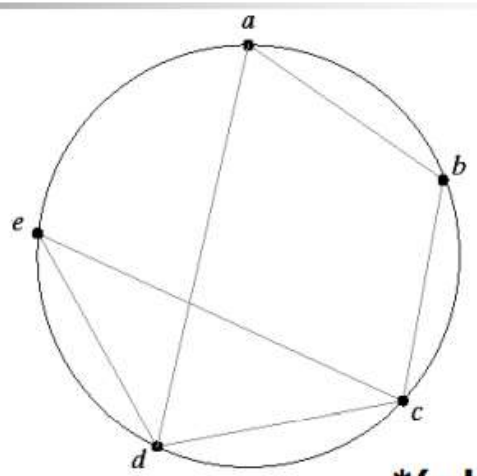


(a) Maximal compatible:
$(a, b,)$ (a, c, d) (b, e, f)

Closed Covering Example



(a) Implication table



(b) Merger diagram

Compatibles	(a, b)	(a, d)	(b, c)	(c, d, e)
Implied states	(b, c)	(b, c)	(d, e)	(a, d,) (b, c,)

(c) Closure table

***(a,b) (c,d,e) → (X) implied (b,c) is not included in the set**

*** better choice: (a,d) (b,c) (c,d,e) all implied states are included**

Outline

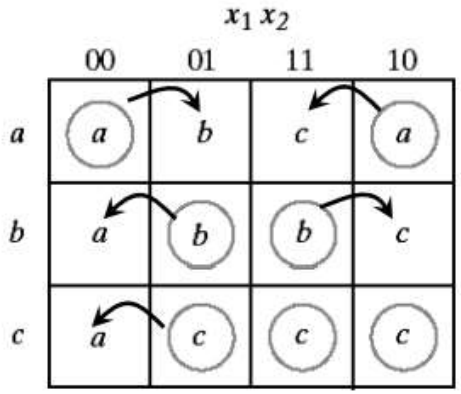
- Asynchronous Sequential Circuits
- Analysis Procedure
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Race-Free State Assignment

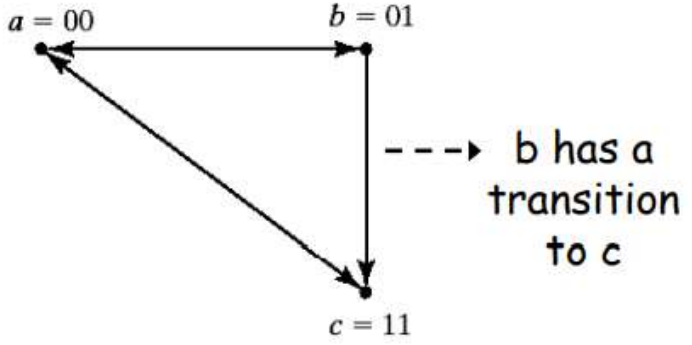
- Objective: choose a proper binary state assignment to ***prevent critical races***
- Only one variable can change at any given time when a state transition occurs
- States between which transitions occur will be given ***adjacent*** assignments
  - Two binary values are said to be adjacent if they differ in only one variable
- To ensure that a transition table has no critical races, every possible state transition should be checked
  - A tedious work when the flow table is large
  - Only 3-row and 4-row examples are demonstrated

3-Row Flow Table Example (1/2)

- Three states require two binary variables
- Outputs are omitted for simplicity
- Adjacent info. are represented by a transition diagram
- a and c are still not adjacent in such an assignment !!
  - Impossible to make all states adjacent if only 3 states are used



(a) Flow table

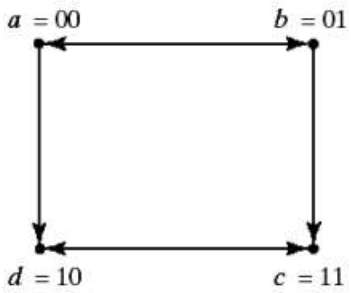


(b) Transition diagram

3-Row Flow Table Example (2/2)

- A race-free assignment can be obtained if we add an extra row to the flow table
  - Only provide a race-free transition between the stable states
- The transition from a to c must now go through d
  - 00 → 10 → 11 (no race condition)

	x_1x_2 = 00	01	11	10
a	(a)	b	d	(a)
b	a	(b)	(b)	c
c	d	(c)	(c)	(c)
d	a	–	c	–



don't care but cannot be 10 (cannot stable)

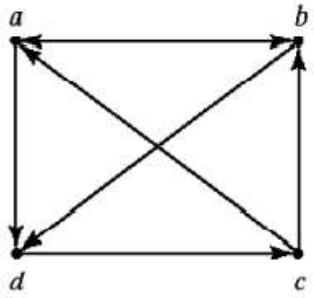
	x_1x_2 = 00	01	11	10
$a = 00$	(00)	01	10	(00)
$b = 01$	00	(01)	(01)	11
$c = 11$	10	(11)	(11)	(11)
$d = 10$	00	–	11	–

4-Row Flow Table Example (1/2)

- Sometimes, just one extra row may not be sufficient to prevent critical races
  - More binary state variables may also required
- With one or two diagonal transitions, there is no way of using two binary variables that satisfy all adjacency

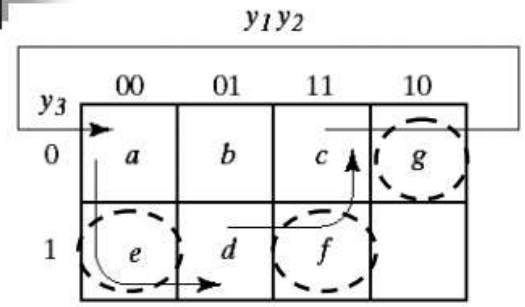
	00	01	11	10
a	*b*	(*a*)	*d*	(*a*)
b	(*b*)	*d*	(*b*)	*a*
c	(*c*)	*a*	*b*	(*c*)
d	*c*	(*d*)	(*d*)	*c*

(a) Flow table

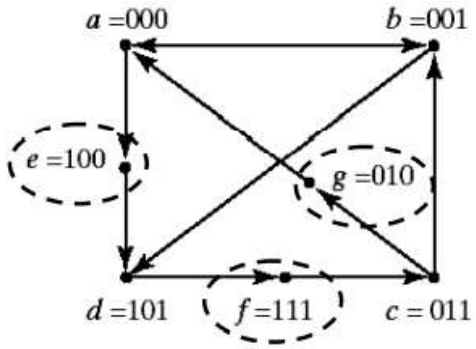


(b) Transition diagram

4-Row Flow Table Example (2/2)



(a) Binary assignment



(b) Transition diagram

	00	01	11	10
000 = *a*	*b*	(*a*)	*e*	(*a*)
001 = *b*	(*b*)	*d*	(*b*)	*a*
011 = *c*	(*c*)	*g*	*b*	(*c*)
010 = *g*	–	*a*	–	–
110 –	–	–	–	–
111 = *f*	*c*	–	–	*c*
101 = *d*	*f*	(*d*)	(*d*)	*f*
100 = *e*	–	–	*d*	–

still has only 4 stable states

Multiple-Row Method

- **Multiple-row method is easier**
  - **May not as efficient as in above *shared-row* method**
- **Each stable state is duplicated with exactly the same output**
  - **Behaviors are still the same**
- **While choosing the next states, choose the adjacent one**

can be used to any 4-row flow table

y_1 \ y_2y_3	00	01	11	10
0	a_1	b_1	c_1	d_1
1	c_2	d_2	a_2	b_2

(a) Binary assignment

	00	01	11	10
$000 = a_1$	b_1	(a_1)	d_1	(a_1)
$111 = a_2$	b_2	(a_2)	d_2	(a_2)
$001 = b_1$	(b_1)	d_2	(b_1)	a_1
$110 = b_2$	(b_2)	d_1	(b_2)	a_2
$011 = c_1$	(c_1)	a_2	b_1	(c_1)
$100 = c_2$	(c_2)	a_1	b_2	(c_2)
$010 = d_1$	c_1	(d_1)	(d_1)	c_1
$101 = d_2$	c_2	(d_2)	(d_2)	c_2

(b) Flow table

Outline

- Asynchronous Sequential Circuits
- Analysis Procedure
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

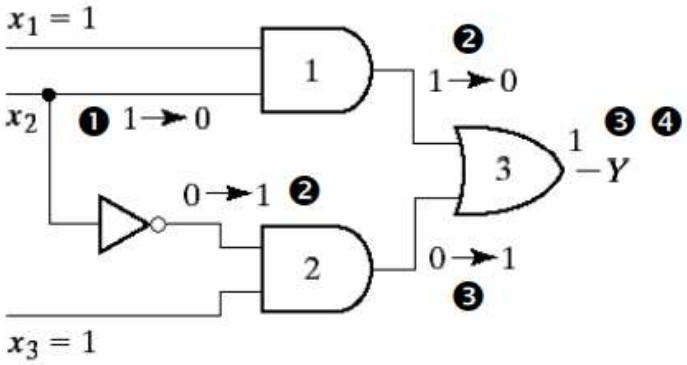
Hazards

- **Unwanted switching appears at the output of a circuit**
  - **Due to different propagation delay in different paths**
- **May cause the circuit to mal-function**
  - **Cause temporary false-output values in combinational circuits**
  - **Cause a transition to a wrong state in asynchronous circuits**
  - **Not a concern to synchronous sequential circuits**
- **Three types of hazards:**

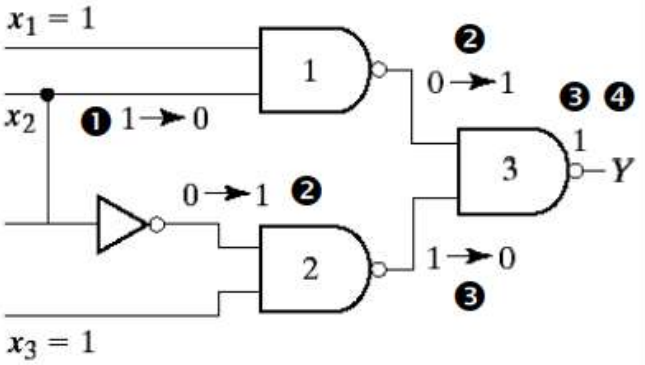
(a) Static 1-hazard (b) Static 0-hazard (c) Dynamic hazard

Circuits with Hazards

- **Static hazard: a momentary output change when no output change should occur**
- **If implemented in sum of products:**
  - **no static 1-hazard → no static 0-hazard or dynamic hazard**
- **Two examples for static 1-hazard:**



(a) AND-OR circuit

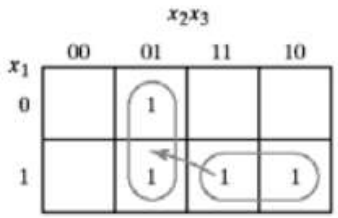


(b) NAND circuit

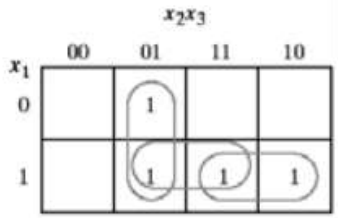
Hazard-Free Circuit

- Hazard can be detected by inspecting the map
- The change of input results in a change of covered product term

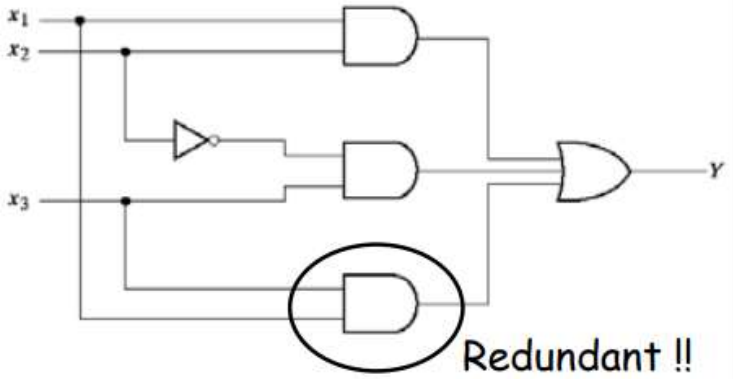
  → Hazard exists
  - Ex: 111 → 101 in (a)
- To eliminate the hazard, enclose the two minterms in another product term
  - Results in redundant gates



(a) $Y = x_1x_2 + x'_2x_3$

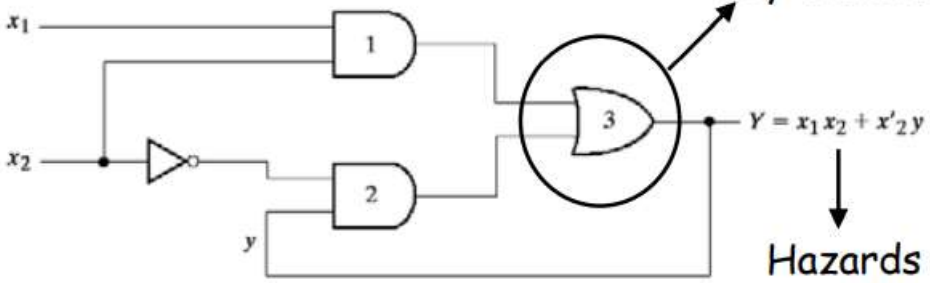


(b) $Y = x_1x_2 + x'_2x_3 + x_1x_3$



Remove Hazard with Latches

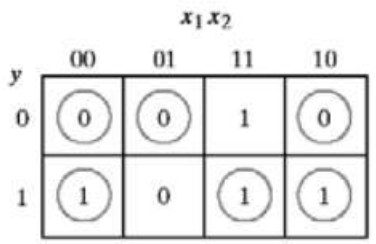
- **Implement the asynchronous circuit with SR latches can also remove static hazards**
  - **A momentary 0 has no effects to the S and R inputs of a NOR latch**
  - **A momentary 1 has no effects to the S and R inputs of a NAND latch**



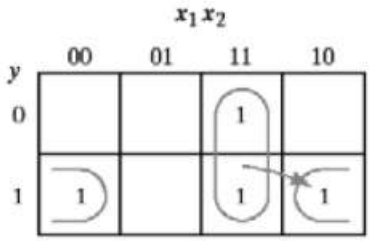
(a) Logic diagram

Replaced by a latch

Hazards exist !!



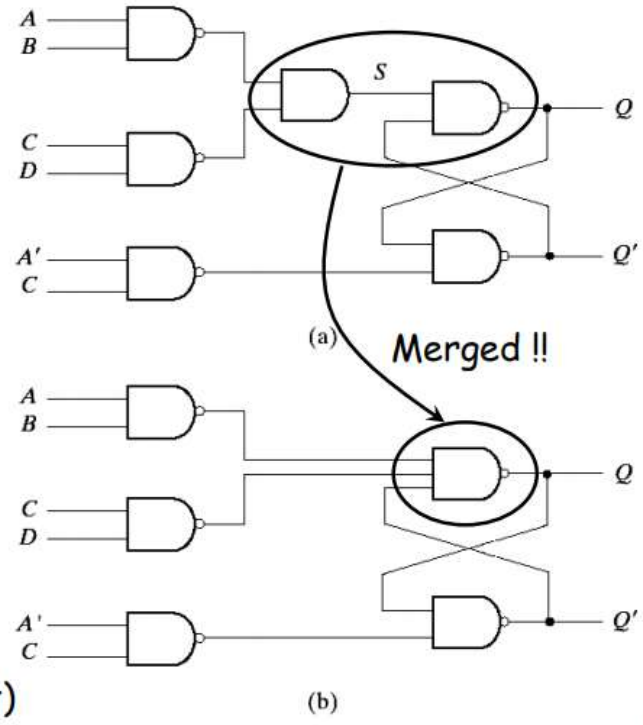
(b) Transition table



(c) Map for Y

Implementation with SR Latches

- **Given:**
  S = AB + CD
  R = A'C
- **For NAND latch, use complemented inputs**
  - **S' = (AB + CD)'**
    = (AB)'(CD)'
  - **R' = (A'C)'**
- **Q = (Q'S)'**
  = [Q'(AB)'(CD)']'
  → Two-level circuits
  (this is the output we want)



Essential Hazards

- Besides static and dynamic hazards, another type of hazard in asynchronous circuits is called ***essential hazard***
- Caused by unequal delays along two or more paths that originate from the ***same input***
- Cannot be corrected by adding redundant gates
- Can only be corrected by adjusting the amount of delay in the affected path
  - Each feedback path should be examined carefully !!

Outline

- Asynchronous Sequential Circuits
- Analysis Procedure
- Circuits with Latches
- Design Procedure
- Reduction of State and Flow Tables
- Race-Free State Assignment
- Hazards
- Design Example

Recommended Design Procedure

1. State the design specifications
2. Derive a primitive flow table
3. Reduce the flow table by merging the rows
4. Make a race-free binary state assignment
5. Obtain the transition table and output map
6. Obtain the logic diagram using SR latches

Primitive Flow Table

- Design a negative-edge-triggered T flip-flop
- Two inputs: T(toggle) and C(clock)
  - T=1: toggle, T=0: no change
- One output: Q

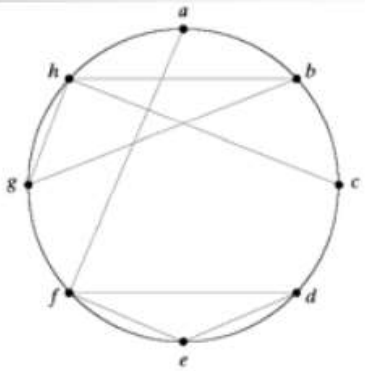
State	Input T	Input C	Output Q	Comments
a	1	1	0	Initial output is 0
b	1	0	1	After state a
c	1	1	1	Initial output is 1
d	1	0	0	After state c
e	0	0	0	After states d or f
f	0	1	0	After states e or a
g	0	0	1	After states b or h
h	0	1	1	After states g or c

	TC = 00	01	11	10
a	– , –	f , –	(a), 0	b , –
b	g , –	– , –	c , –	(b), 1
c	– , –	h , –	(c), 1	d , –
d	e , –	– , –	a , –	(d), 0
e	(e), 0	f , –	– , –	d , –
f	e , –	(f), 0	a , –	– , –
g	(g), 1	h , –	– , –	b , –
h	g , –	(h), 1	c , –	– , –

Merging the Flow Table

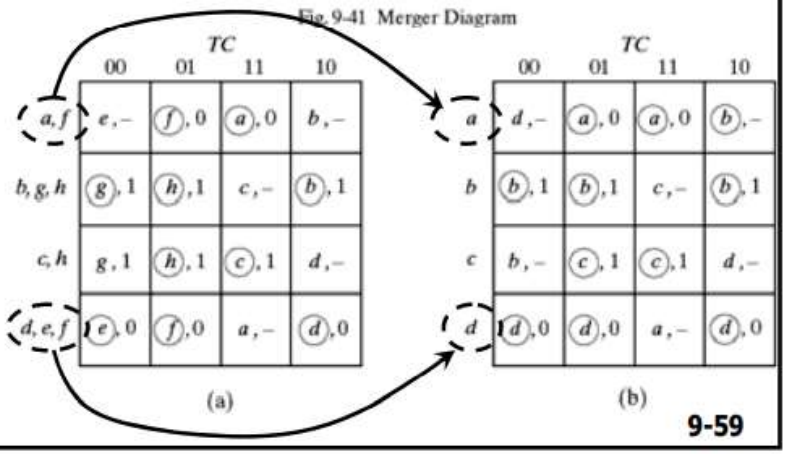
Compatible pairs:
(a,f) (b,g) (b,h) (c,h)
(d,e) (d,f) (e,f) (g,h)

Maximal compatible set:
(a,f) (b,g,h) (c,h) (d,e,f)
a b c d



	a	b	c	d	e	f	g
b	a, c ×						
c	×	b, d ×					
d	b, d ×	×	a, c ×				
e	b, d ×	e, g × b, d ×	f, h ×	✓			
f	✓	e, g × a, c ×	f, h × a, c ×	✓	✓		
g	f, h ×	✓	b, d ×	e, g × b, d ×	×	e, g × f, h ×	
h	f, h × a, c ×	✓	✓	d, e × c, f ×	e, g × f, h ×	×	✓

Fig. 9-41 Merger Diagram



State Assignment & Transition Table

- **No diagonal lines in the transition diagram**

  → **No need to add extra states**

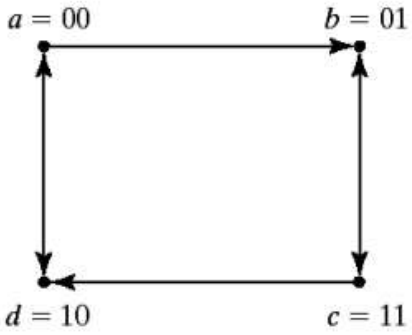


Fig. 9-43 Transition Diagram

$y_1 y_2$	TC = 00	TC = 01	TC = 11	TC = 10
$a = 00$	10	(00)	(00)	01
$b = 01$	(01)	(01)	11	(01)
$c = 11$	01	(11)	(11)	10
$d = 10$	(10)	(10)	00	(10)

(a) Transition table

$y_1 y_2$	TC = 00	TC = 01	TC = 11	TC = 10
00	0	0	0	*X*
01	1	1	1	1
11	1	1	1	*X*
10	0	0	0	0

(b) Output map $Q = y_2$

Logic Diagram

y_1y_2 \ TC	00	01	11	10
00	1	0	0	0
01	0	0	1	0
11	0	X	X	X
10	X	X	0	X

(a) $S_1 = y_2\,TC + y'_2\,T'C'$

y_1y_2 \ TC	00	01	11	10
00	0	X	X	X
01	X	X	0	X
11	1	0	0	0
10	0	0	1	0

(b) $R_1 = y_2\,T'C' + y'_2\,TC$

y_1y_2 \ TC	00	01	11	10
00	0	0	0	1
01	X	X	X	X
11	X	X	X	0
10	0	0	0	0

(c) $S_2 = y'_1\,TC'$

y_1y_2 \ TC	00	01	11	10
00	X	X	X	0
01	0	0	0	0
11	0	0	0	1
10	X	X	X	X

(d) $R_2 = y_1\,TC'$