Computer Organization and Architecture

Lecture Contents

- Multi cycle implementation of MIPS processor
- Designing the control unit of a Multi cycle MIPS processor

Issues with single cycle implementation

- Clock cycle must have same length for every instruction
- Clock cycle length is determined by longest instruction in ISA
- Time is wasted for instructions that could complete in shorter time
- Although CPI is one, but overall performance will be bad
- For very simple ISA: single cycle implementation is valid
- For complex ISA: solution is multi cycle implementation

Example

Operation times for

Memory unit = 200 ps

ALU and adders = 100 ps

Register file (read or write) = 50ps

Assume other control units have no delay

Two implementations under consideration

1. Every instruction operates in one clock cycle of fixed length
2. Every instruction executes in one clock cycle of variable length

There are 25% loads, 10% stores, 45% ALU instructions, 15% branches, 5% jumps

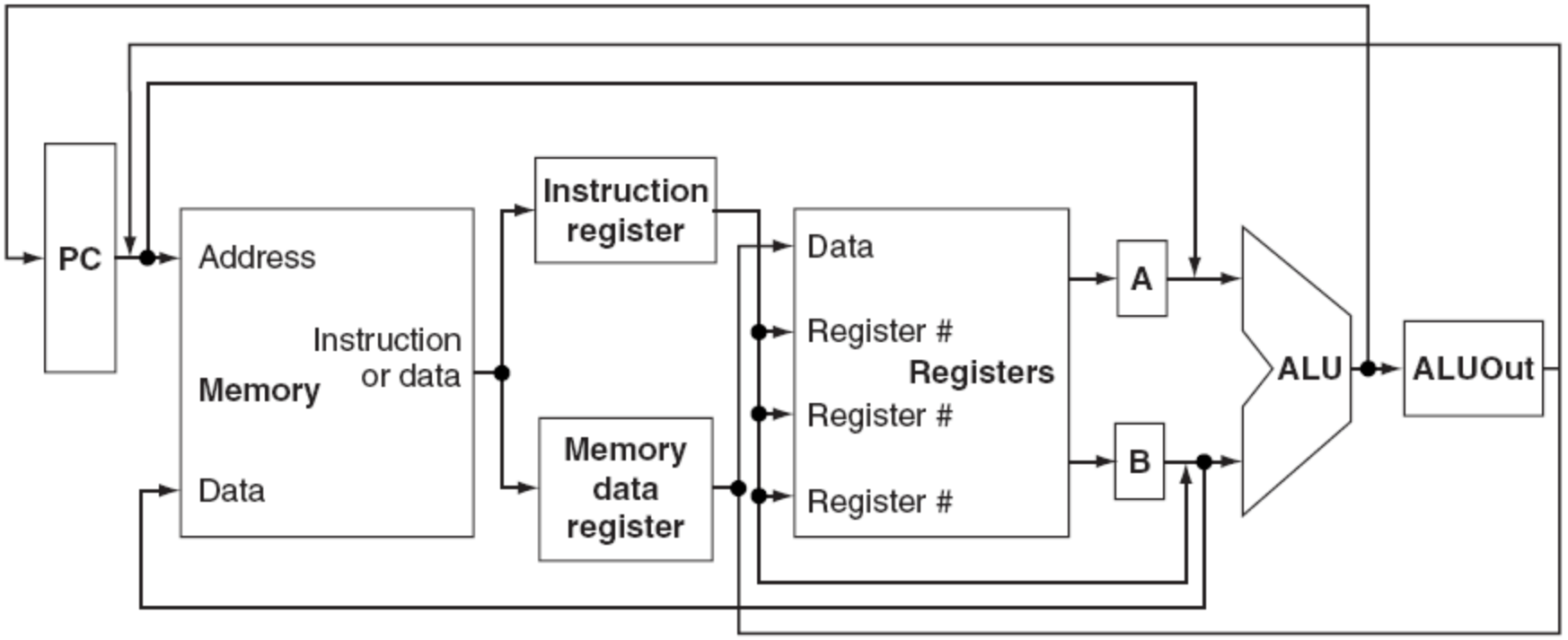
Example solution

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Multi cycle implementation

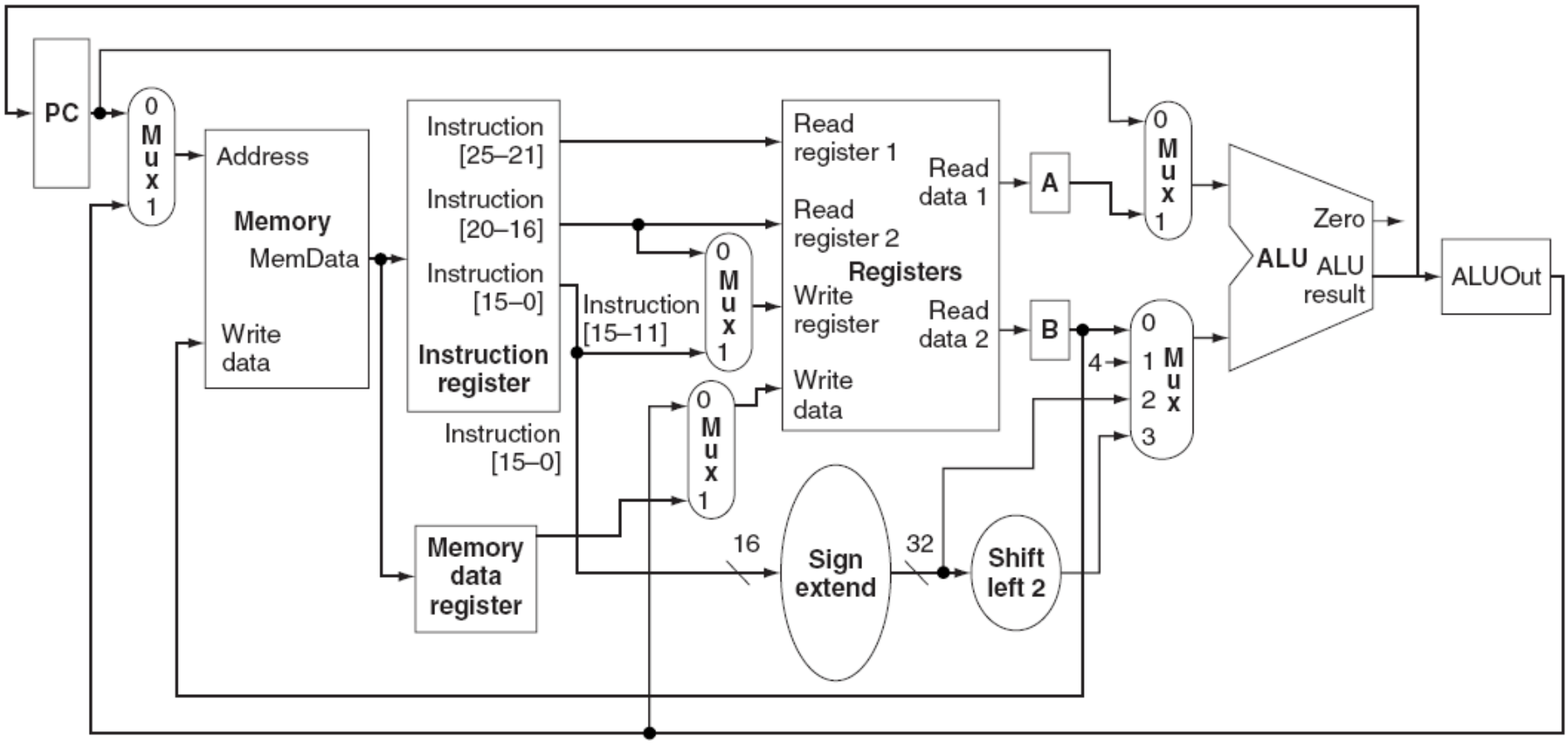
- Multi cycle design allows
  - Different instructions to take different number of clock cycles
  - To share different functional units within the execution of single instruction
  - An abstract view of multi cycle MIPS implementation



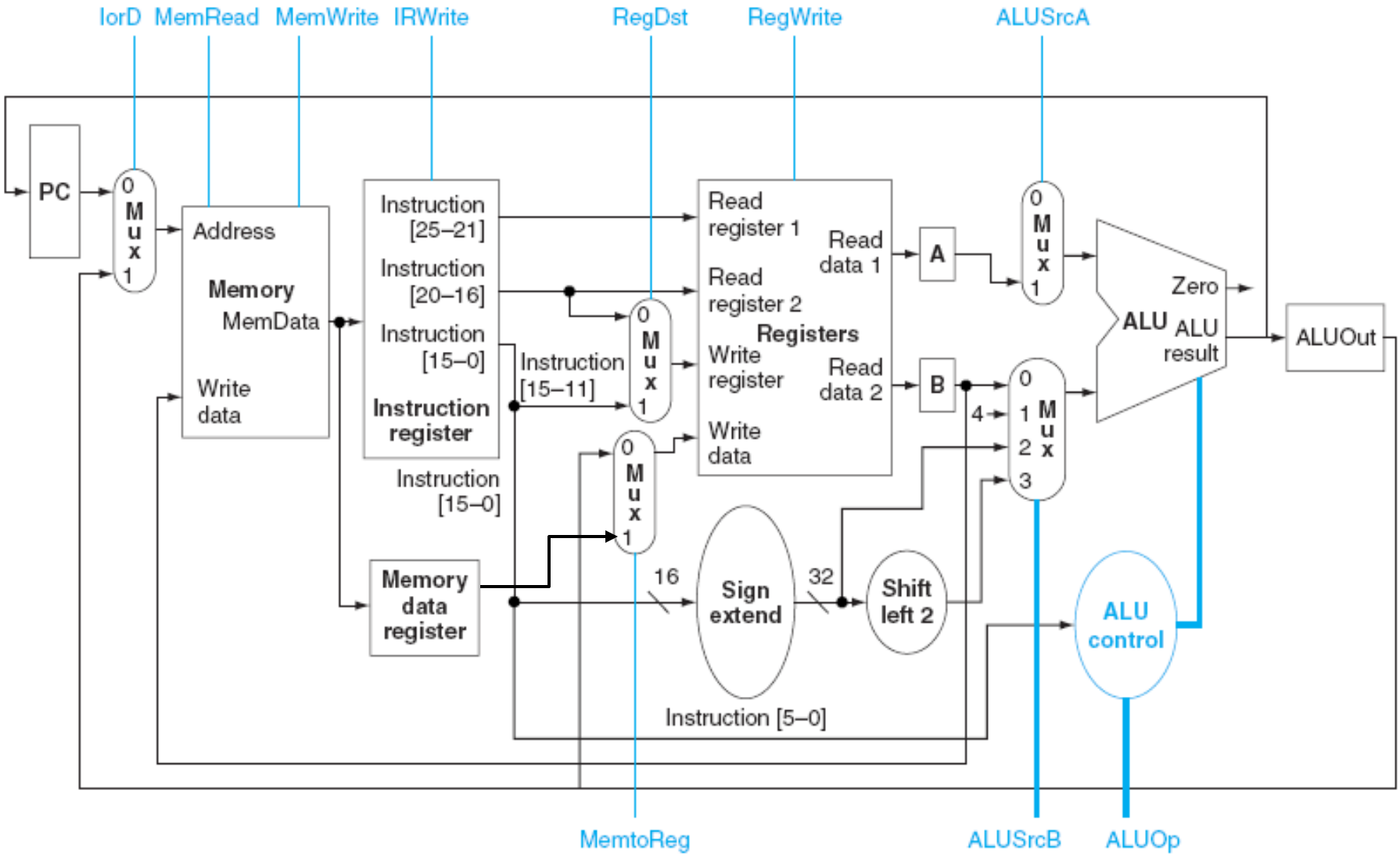
Differences with single cycle implementation

- Single memory unit used for both instruction and data
- Single ALU rather than three ALUs
- One or more registers are added after every major functional unit
- Registers are used to hold data that might be needed in later clock cycles
- Position of registers determined by
  - What combinational units fit in one clock cycle (memory access, register access, ALU)
  - What data is needed in later clock cycles

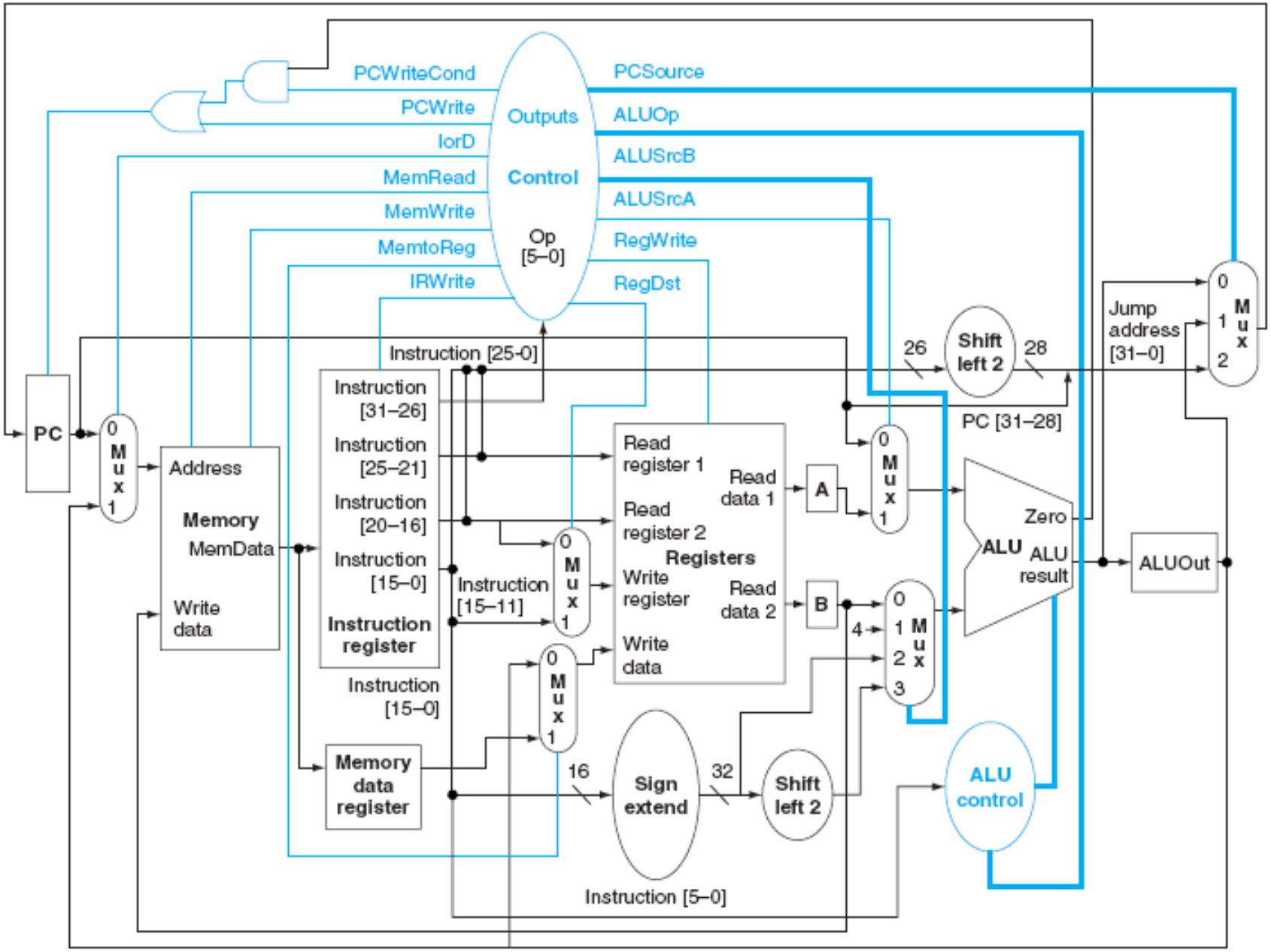
Multi cycle MIPS implementation with MUXs



MIPS multi cycle implementation with control lines



Complete multi cycle MIPS implementation



Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

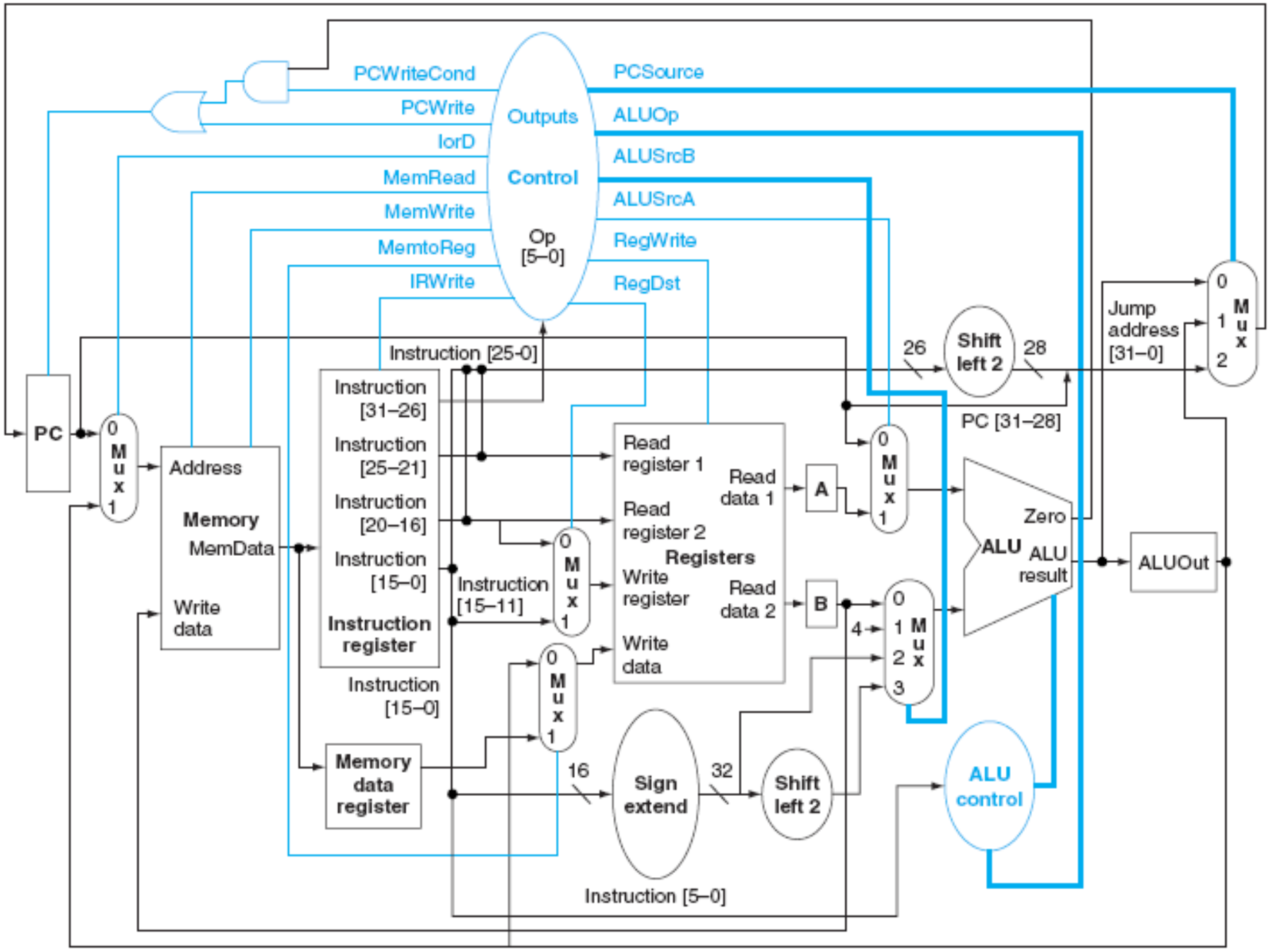
Breaking instruction execution into clock cycles

Mainly five steps

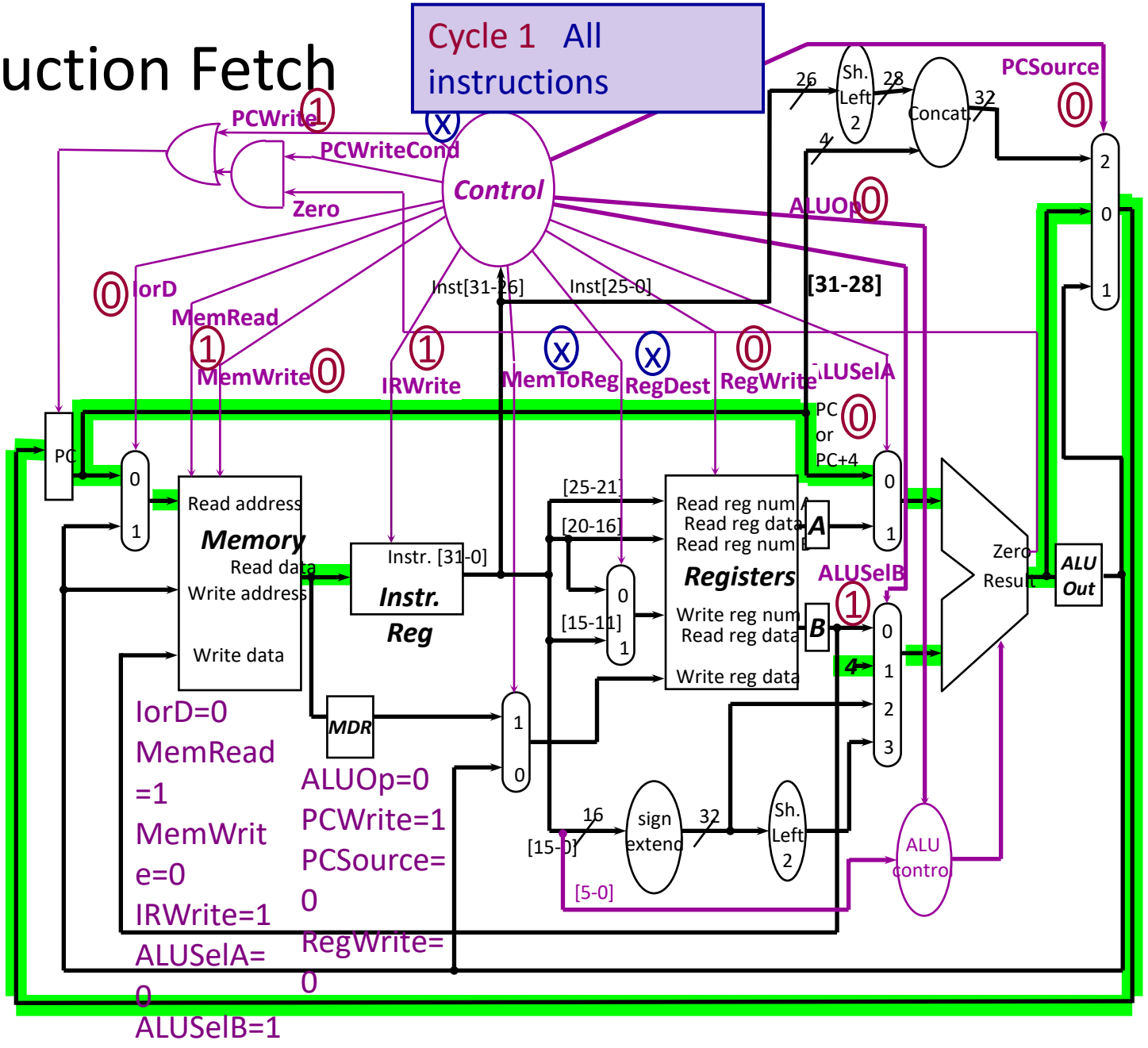
- Instruction fetch step
- Instruction decode and register fetch step
- Execution, memory address computation, or branch completion
- Memory access, or R-type completion
- Memory read completion step

Instruction fetch

- Send the PC to memory as the address and perform a write to IR
- Also increment PC by 4 and store new value in PC register
- The new PC should not look up instruction memory until we are done with instruction execution
- Assert or de-assert relevant control signals

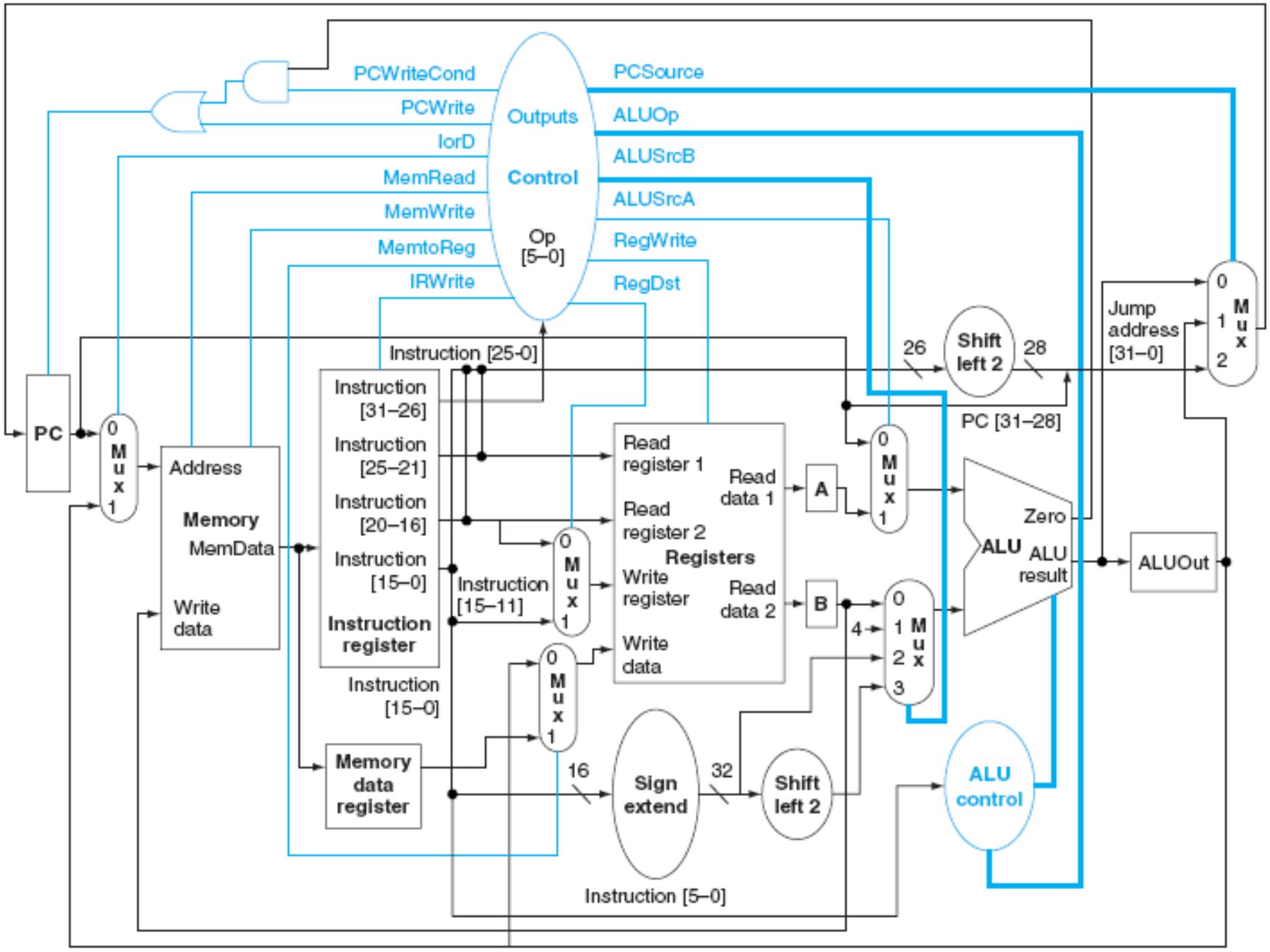

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction [25-0]
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]

Instruction Fetch



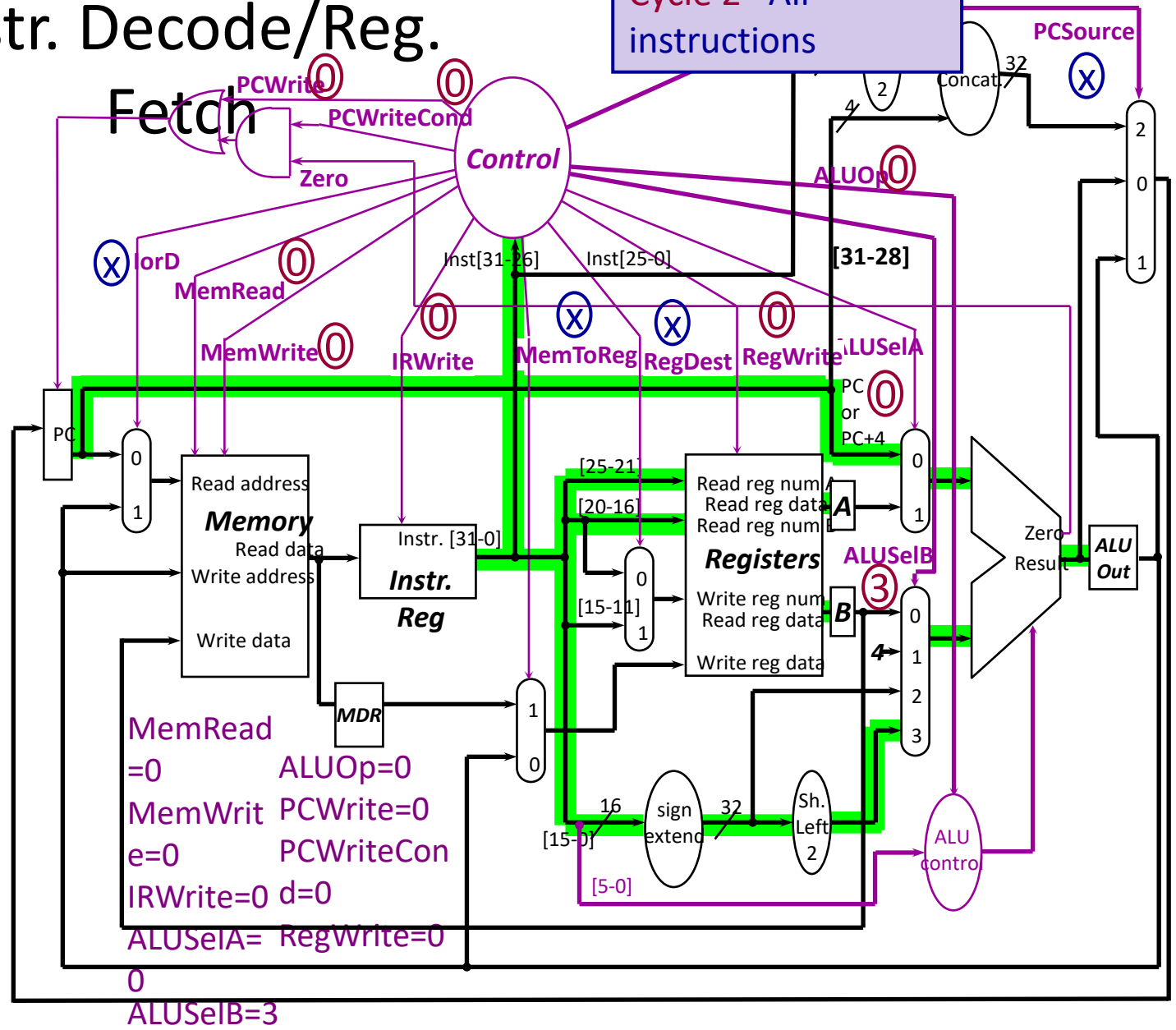
Instruction decode

- We don’t know yet instruction type (we’ll know in execution phase)
- Perform actions that are applicable on all types of instructions
- Two registers are read and values stored in A and B. Reading is not harmful even if it is unnecessary.
- Compute the branch address using PC+4+16 bit offset and store result in ALUOut


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction [25-0]
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]

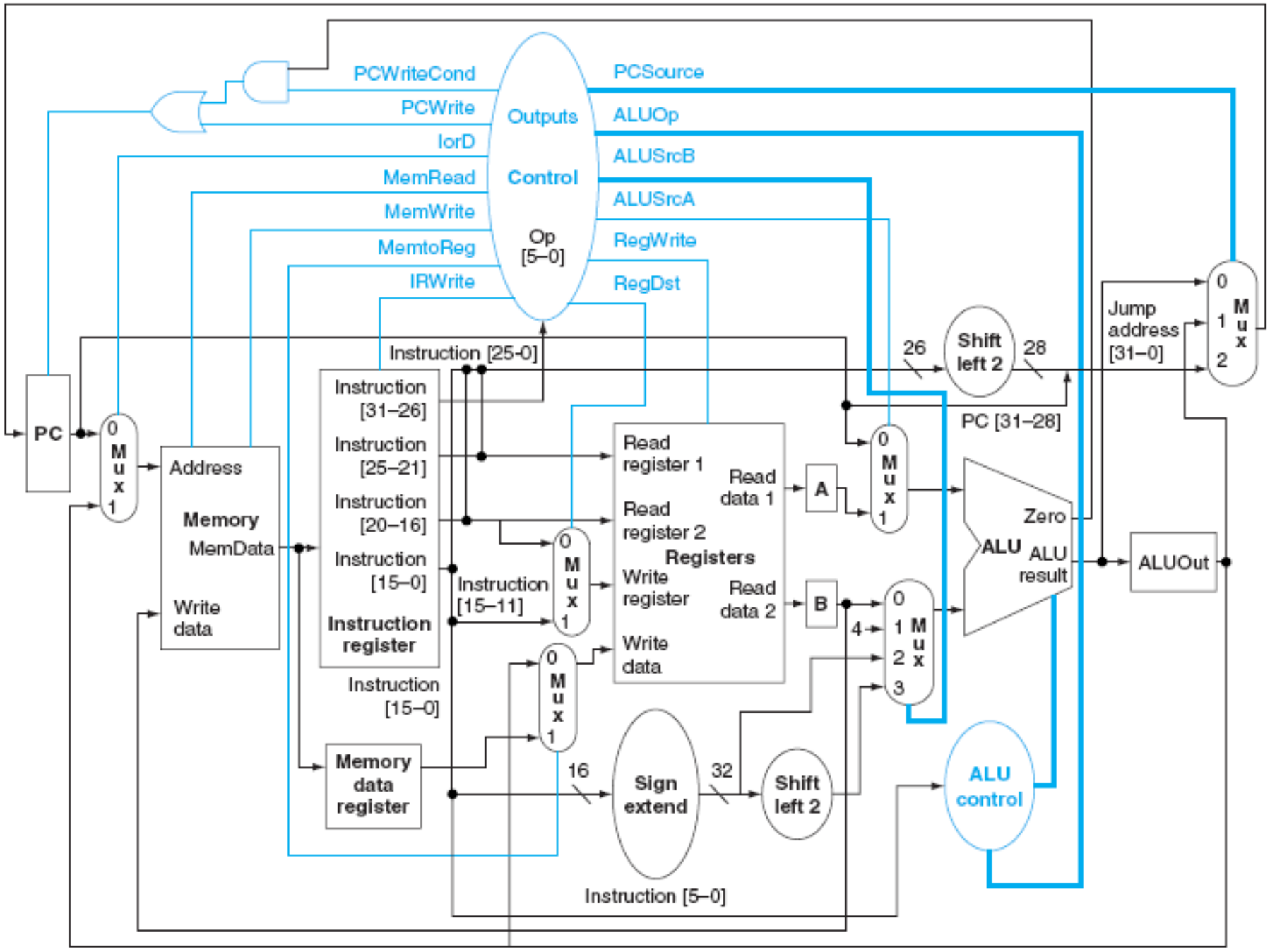
Instr. Decode/Reg. Fetch

Cycle 2 All instructions



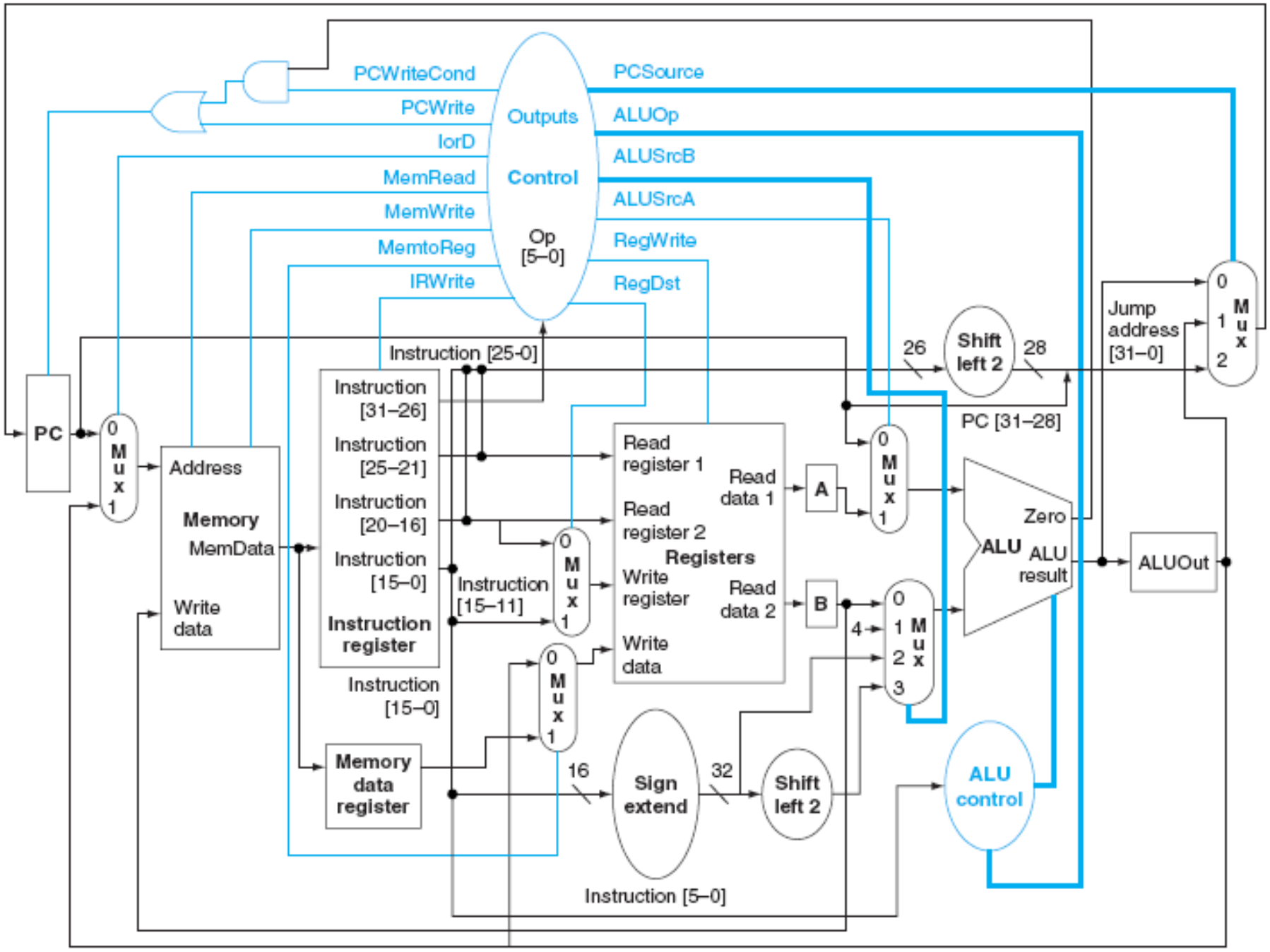
Execution, memory address computation or branch completion

- In this cycle operation to be performed is determined
- ALU performs one of four operations
  - Memory reference
  - Arithmetic logic instruction (R-type)
  - Branch
  - Jump
- Note that branch operation is completed by the end of cycle 3
- R-type, memory reference instructions are still pending


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction [25-0]
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]

Memory access or R-type instruction completion

- Load or store instructions access memory
- Arithmetic-logic instruction writes data to register
- Value retrieved from memory is stored in MDR to be used in next clock cycle
- At the end of this cycle, the R-type and store instruction are complete

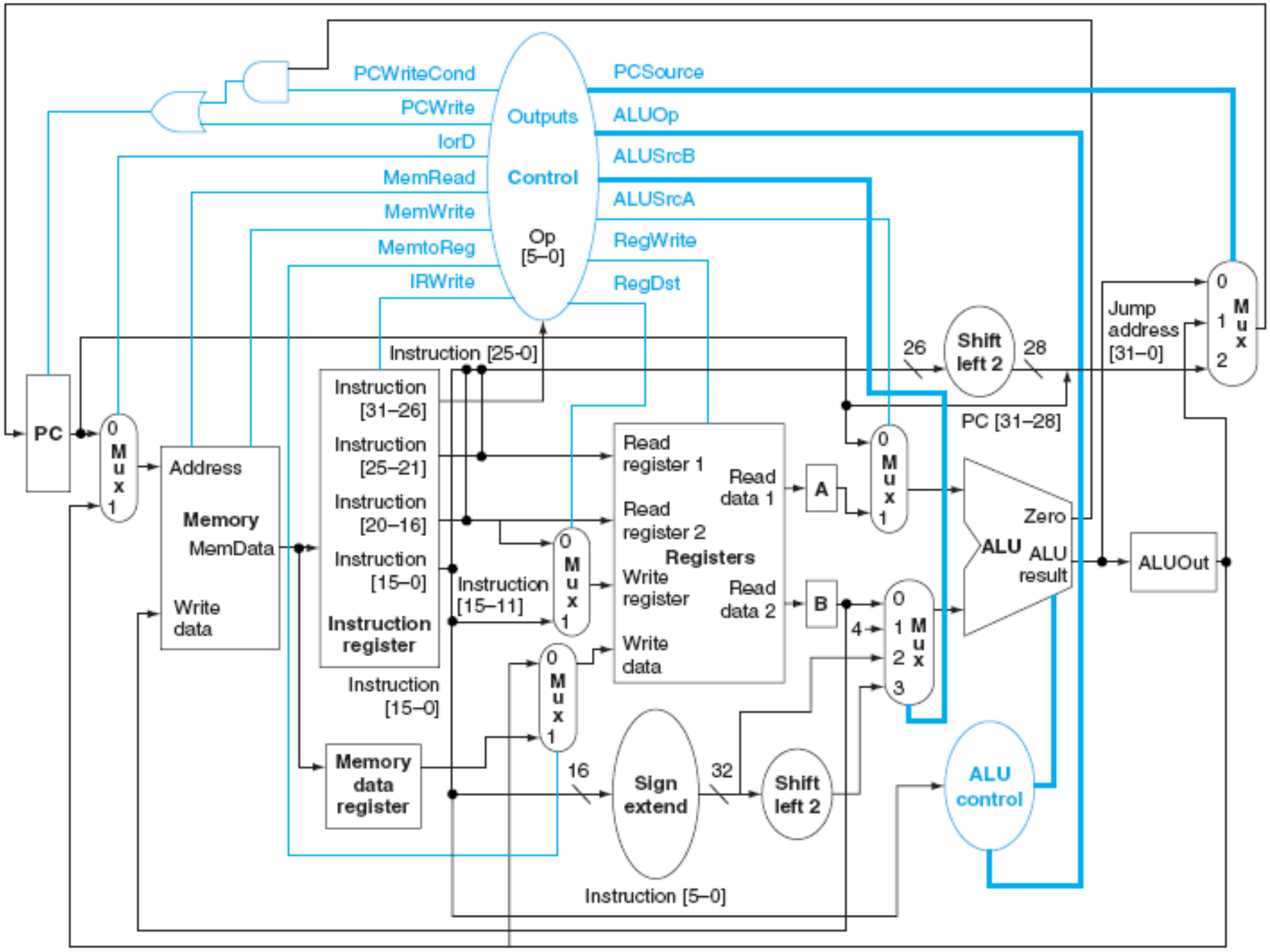

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction [25-0]
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]

Memory read completion

Memory read: the value read from memory (and latched in the memory data register) is now written into the register file

Summary:

- Branches and jumps: 3 cycles
- ALU, stores: 4 cycles
- Memory access: 5 cycles


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction [25-0]
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5–0]

Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Average CPI

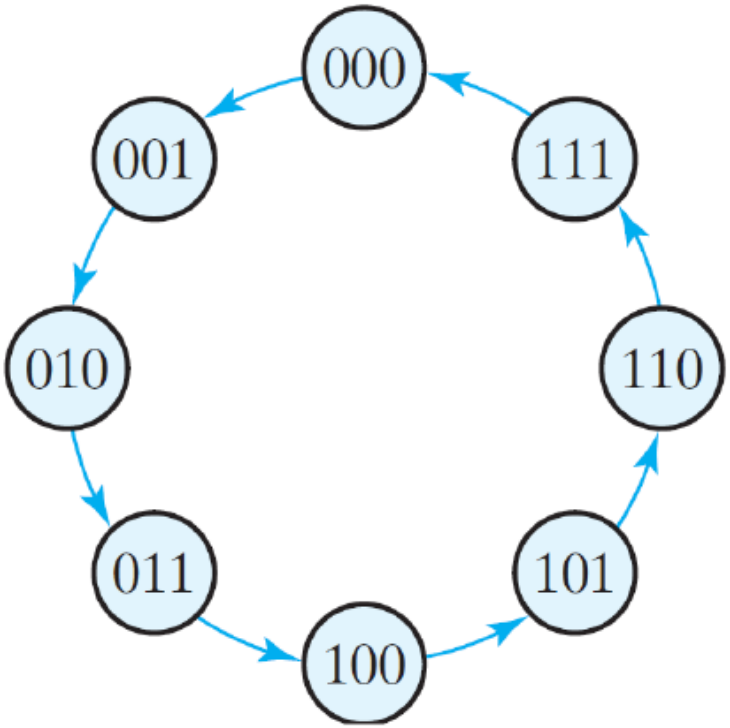
Now we can compute average CPI for a program: if the given program is composed of loads (25%), stores (10%), branches (13%), and ALU ops (52%), the average CPI is

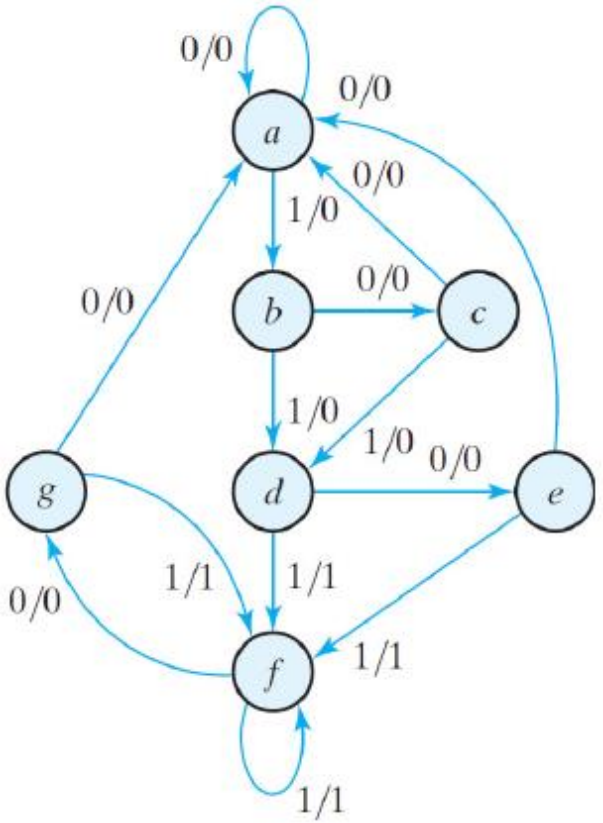
$0.25 \times 5 + 0.1 \times 4 + 0.13 \times 3 + 0.52 \times 4 = 4.12$

You can break this CPU design into shorter cycles, for example, a load would then take 10 cycles, stores 8, ALU 8, branch 6 ➔ average CPI would double, but so would the clock speed, the net performance would remain roughly the same

Defining the control of multi cycle implementation

- Multi cycle control is specified through finite state machine (FSM)
- Remember FSMs??

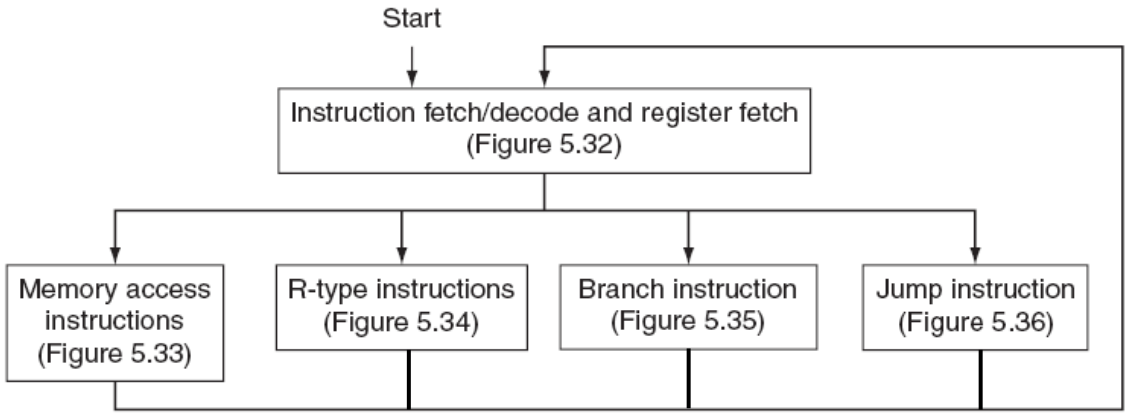




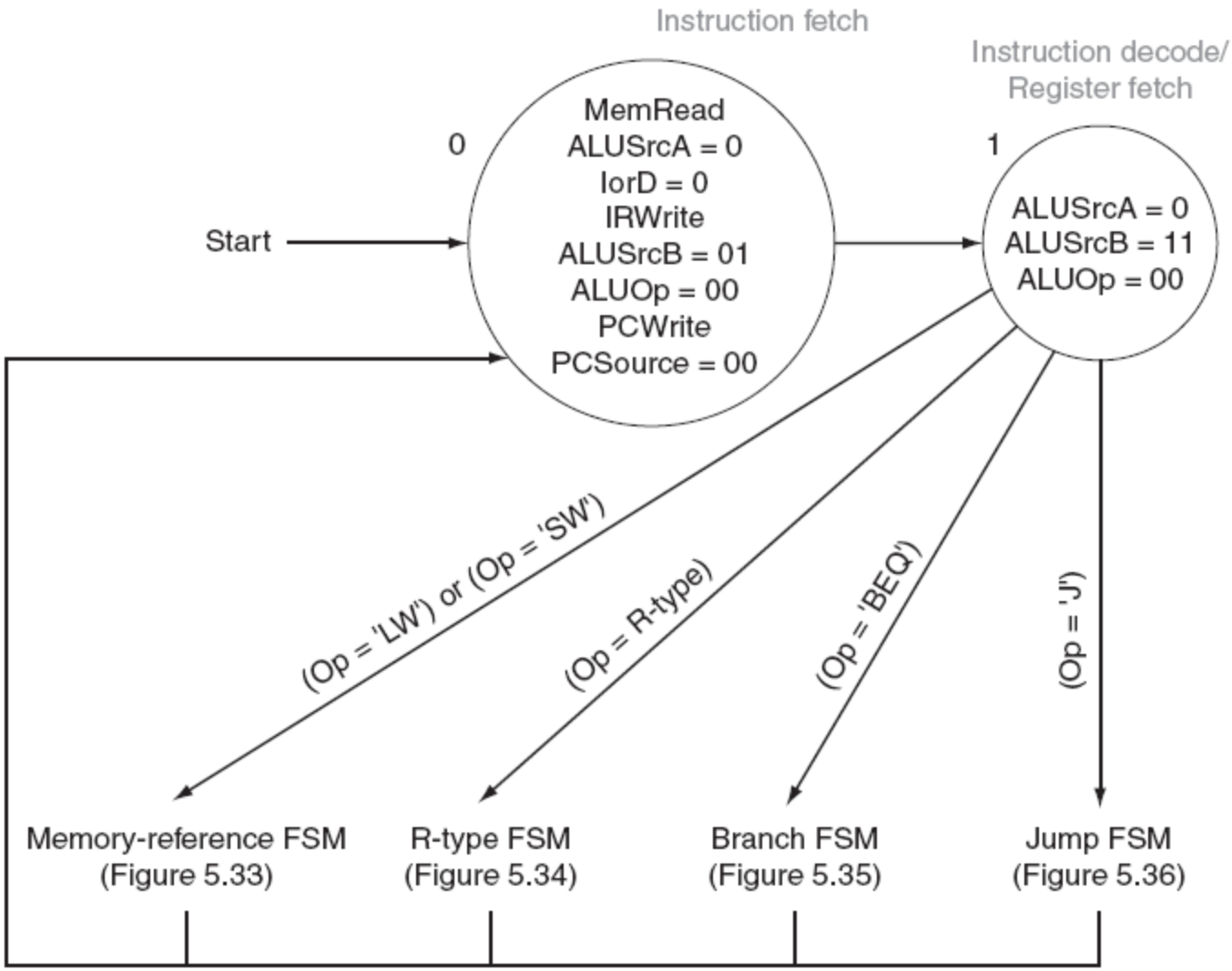
FSM of MIPS multi cycle control

- FSM of MIPS control will be divided into 5 states
- Initial two states will be same for all the instructions
- Step 3 through 5 will depend on the opcode
- After execution of last state, FSM will return to initial state

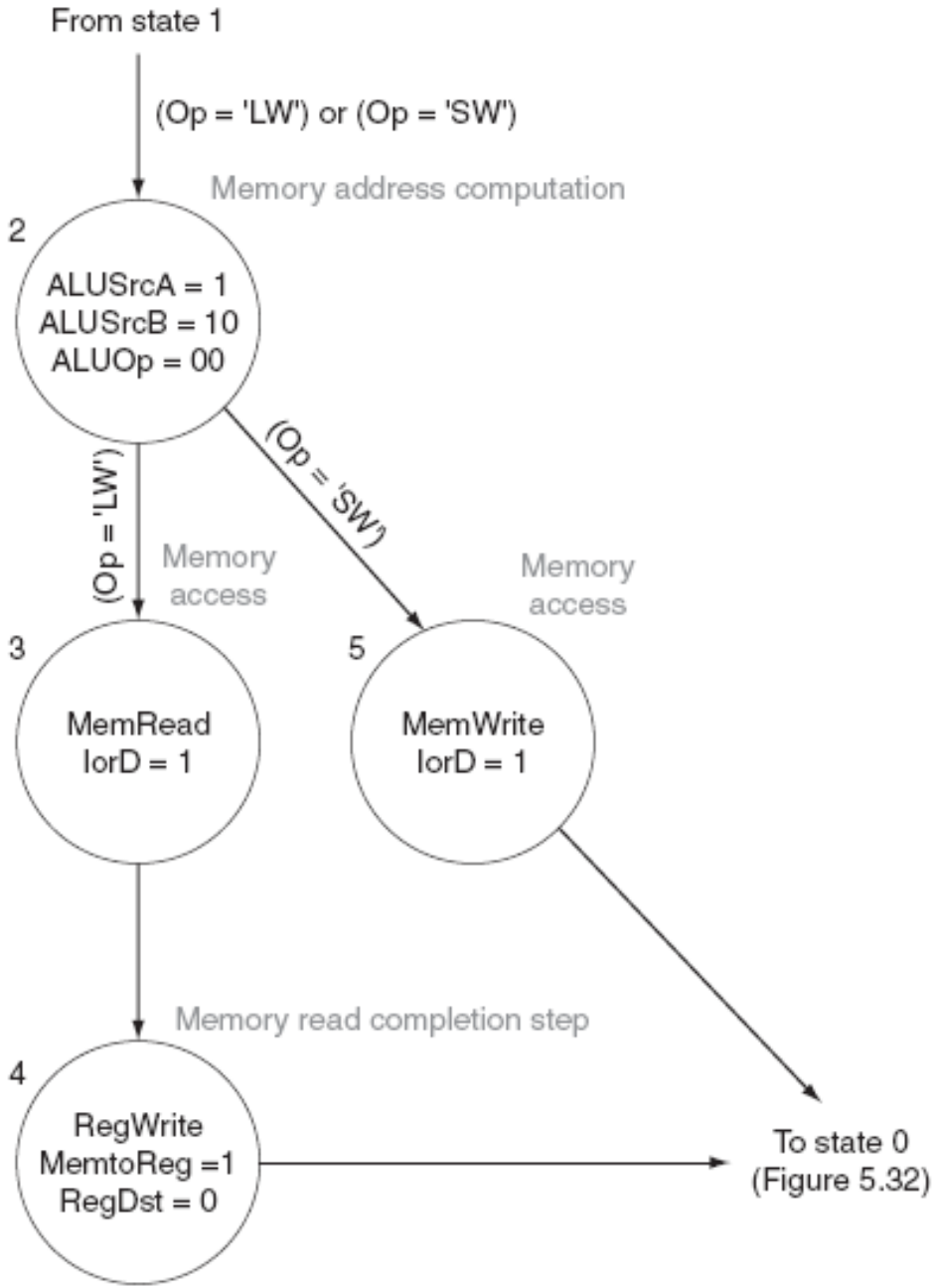
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		



FSM for IF/ID portions

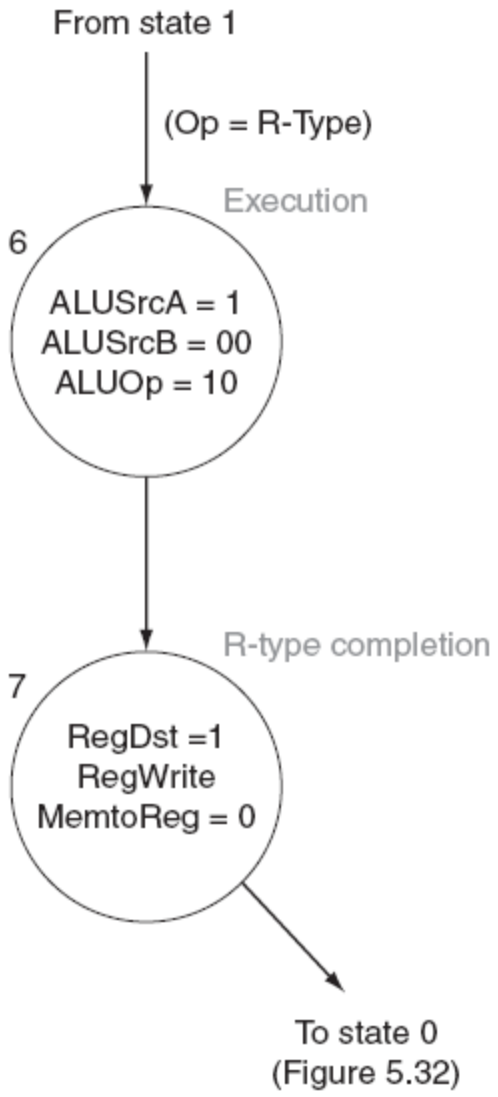


FSM for controlling memory reference instructions

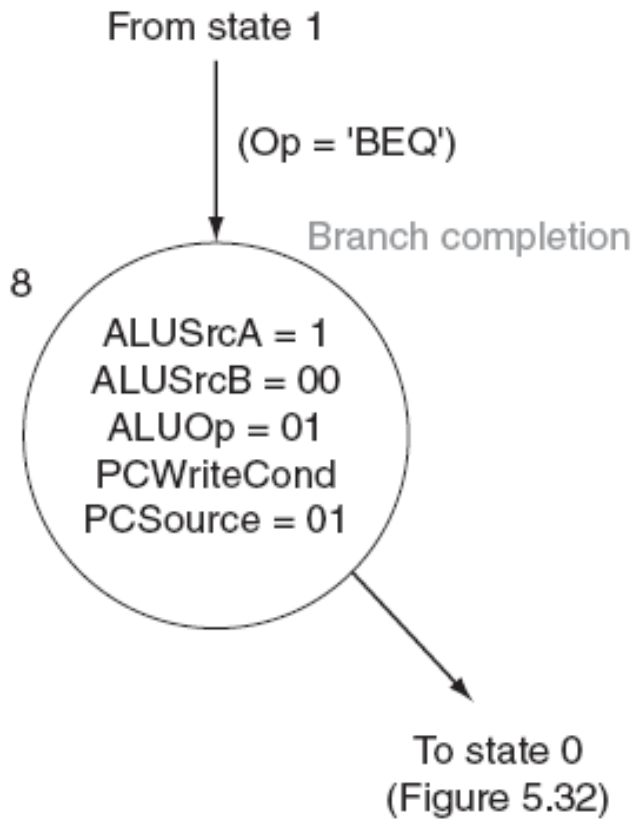


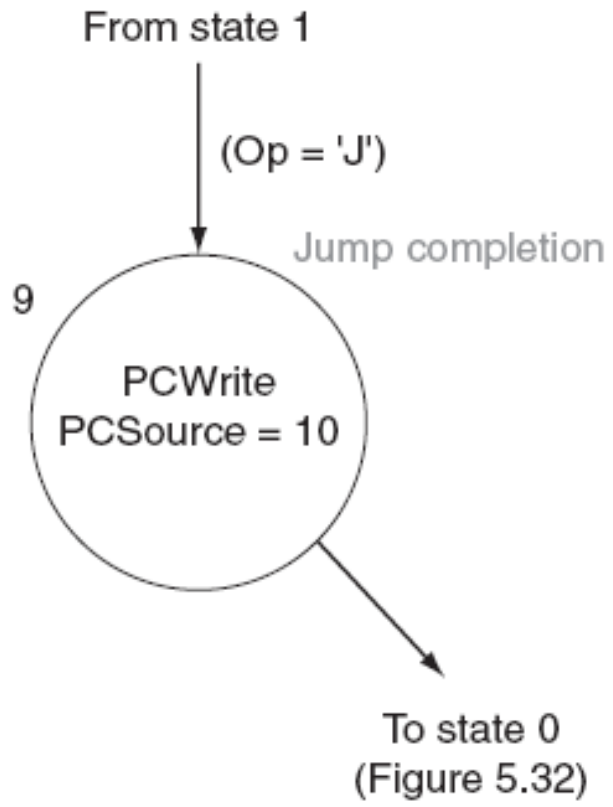
FSM for R-type instructions

- Registers are read from instruction register file
- According to Op code, operation (add, sub etc..) is performed
- Result taken from ALUOut register and written into register file

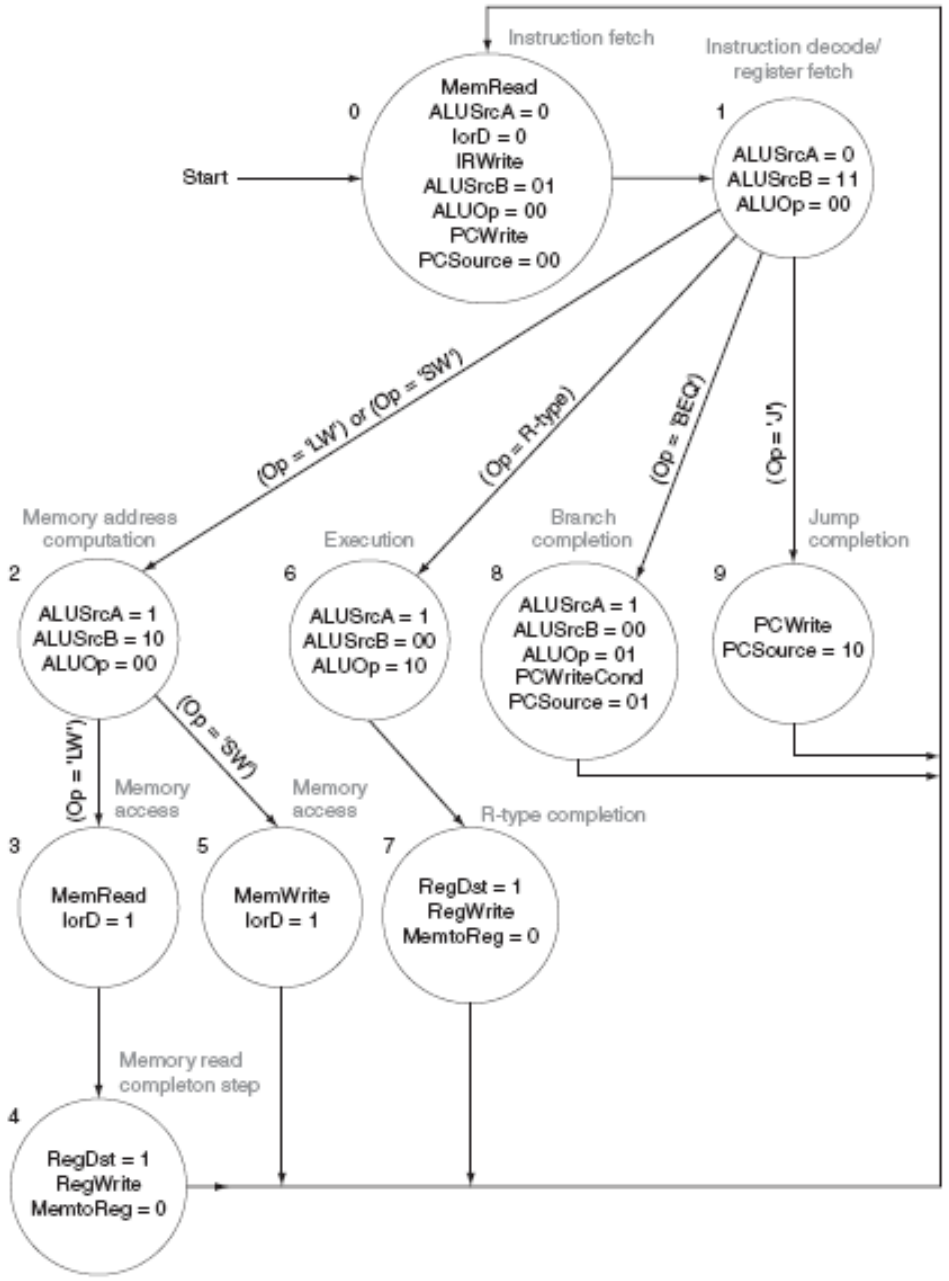


FSM for branch/jump instruction





Complete FSM

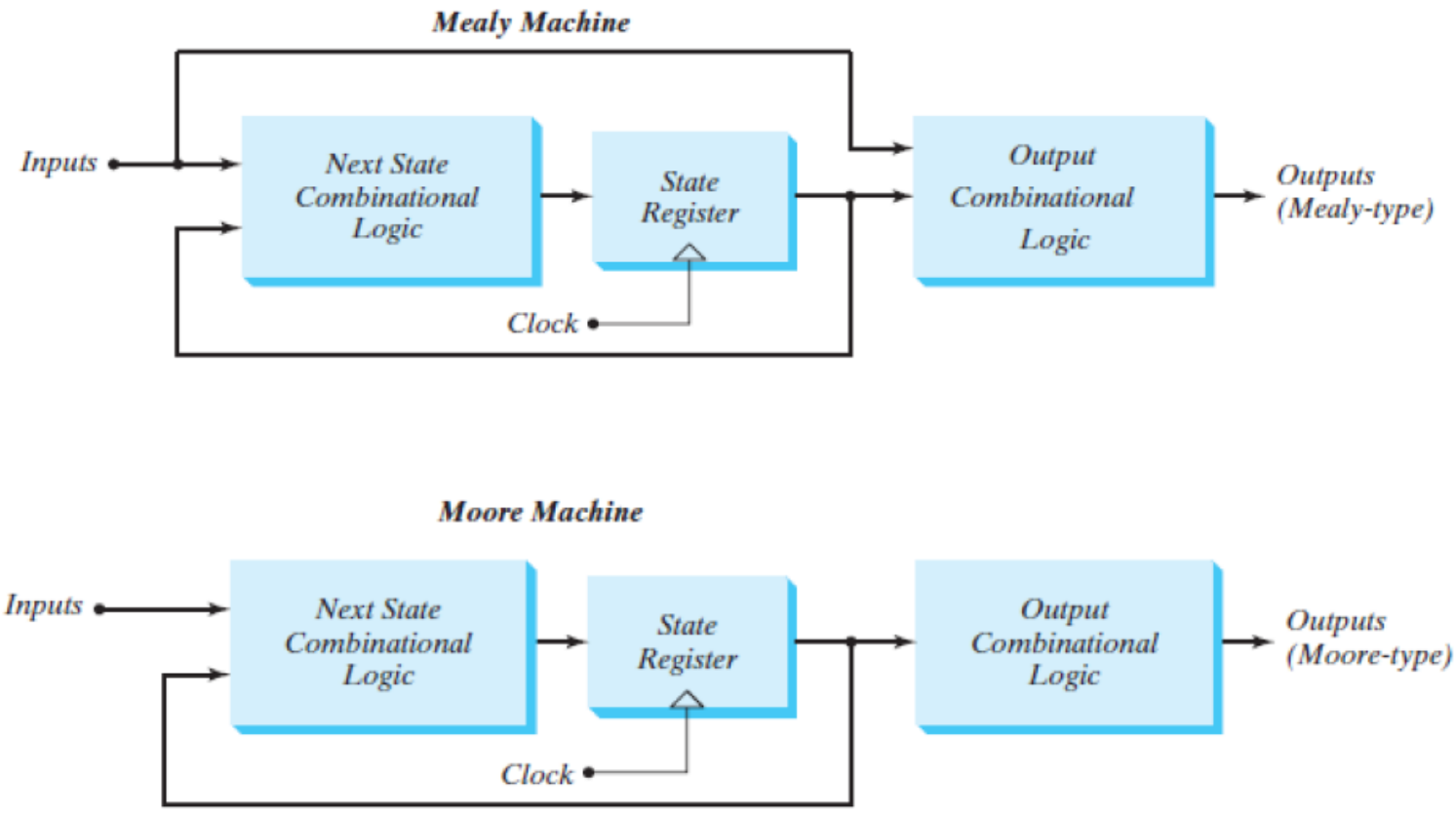


Finite State Machines (FSM)

- State diagrams are representations of *FSM*
  - Two types
    - Mealy FSM
    - Moore FSM
  - Difference
    - How output is determined

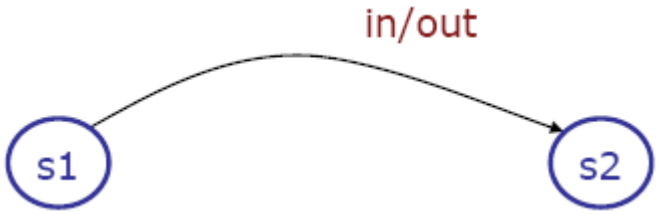
- Mealy FSM
  - Output depends on input and state
  - Output is not synchronized with clock
    - Can have temporarily unstable output
- Moore FSM
  - Output depends only on state

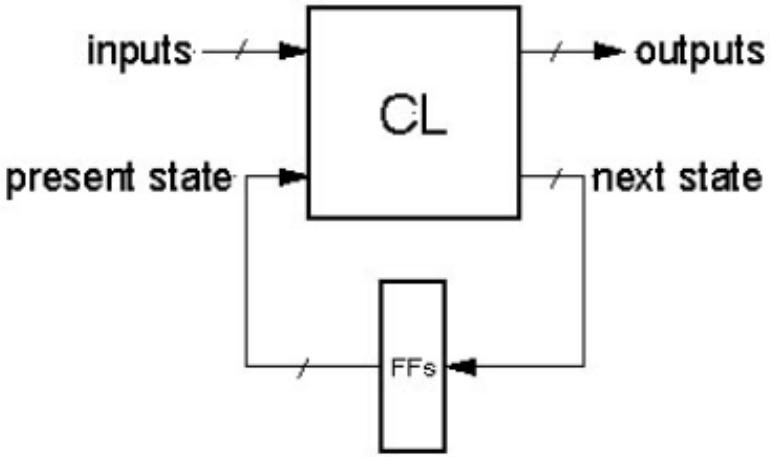
Mealy vs. Moore FSM



Mealy FSM

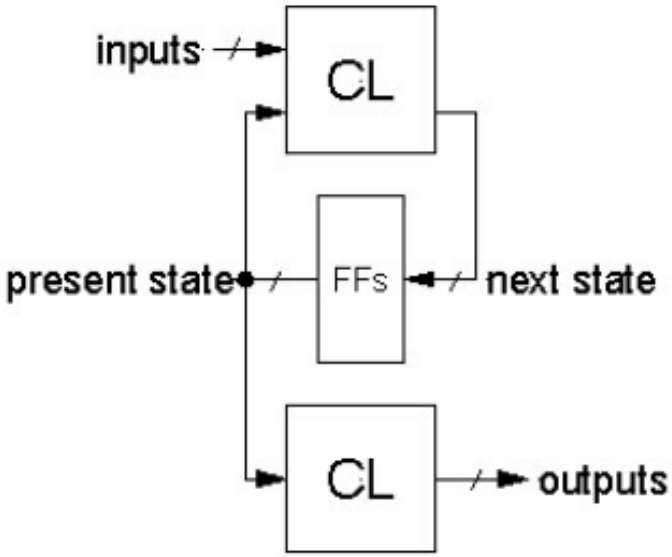
- Output based on state and present input
  - Output changes during transition



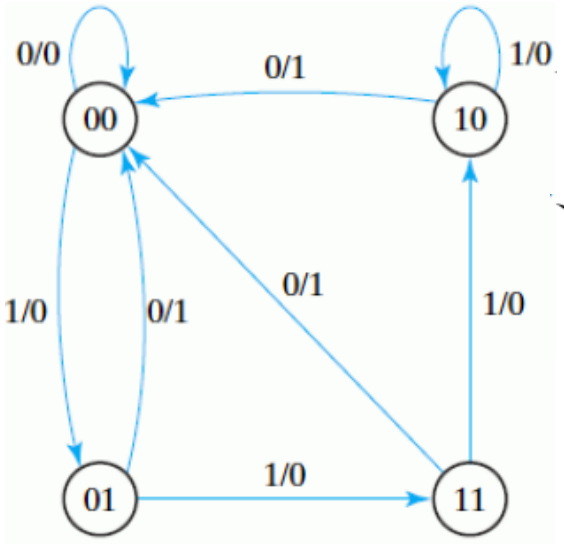


Moore FSM

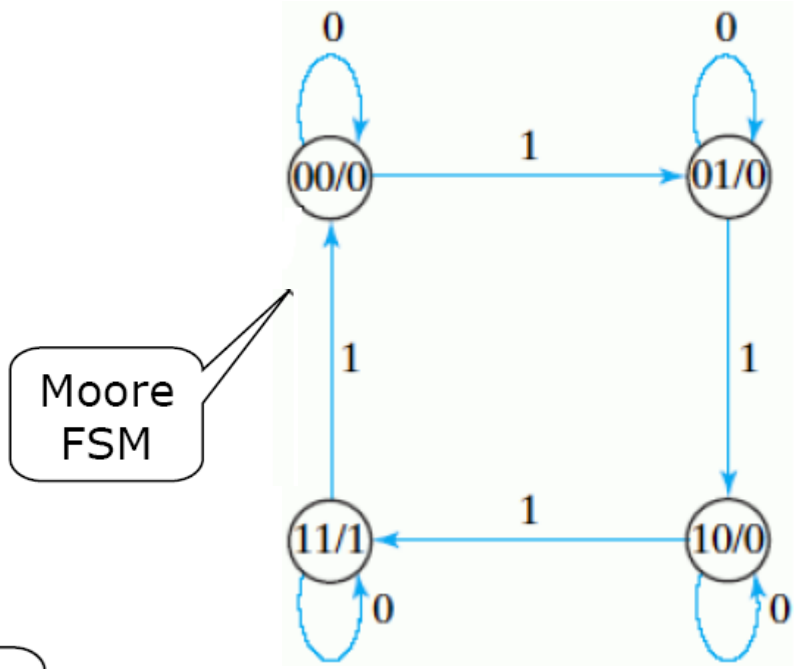
- Output based on state only
  - Output is associated with state



Mealy vs. Moore FSM



Mealy FSM



Moore FSM

Design of Sequential Circuits

- How can we design a sequential circuit?
  - For instance, a circuit that detects 3 or more consecutive 1's in input

- Design procedure
  1. Derive state diagram from description
  2. Reduce number of states if necessary
  3. Assign binary values to states
  4. Obtain binary coded state table (transition table)
  5. Choose type of flip-flops
  6. Derive flip-flop input equations and output equations
  7. Draw logic diagram

State Assignment

- States are represented by flip-flop values in circuit
  - Need to encode state in binary
- What is the minimum number of flip-flops that are necessary to encode m states?
  - Need at least $log_2(m)$ bits
- Many possible encodings
- Terminology:
  - "State table" uses uncoded states
  - "Transition table" uses coded states

State	Binary	Gray	One-hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Example 1: Sequence Detector

- Circuit specification:
  - Design a circuit that outputs a 1 when three consecutive 1s have been applied to input, and 0 otherwise

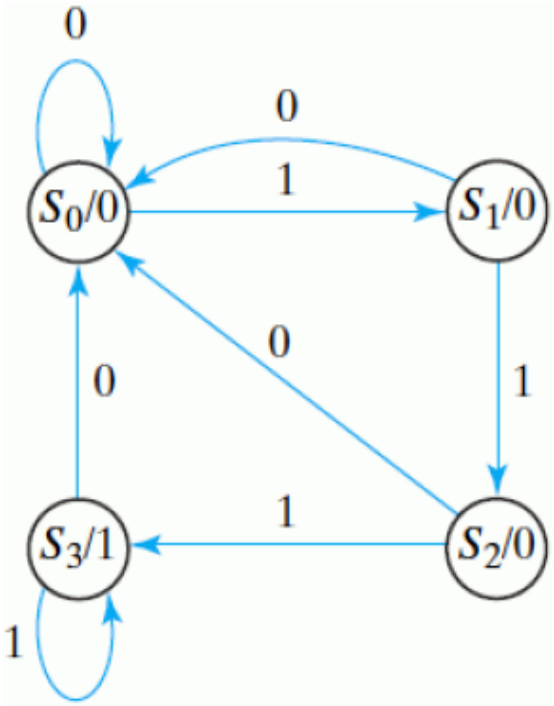
- Step 1: derive state diagram
  - What should a state represent?
    - "number of 1's seen so far"
  - Moore or Mealy FSM?
    - Both possible
    - Chose Moore to simplify diagram
  - State diagram:
    - State S_0: zero 1s detected
    - State S_1: one 1 detected
    - State S_2: two 1s detected
    - State S_3: three 1s detected

Example 1: Sequence Detector

- Step 2: reduce number of states
  - State table

current state	next state x=0	next state x=1	output
S_0	S_0	S_1	0
S_1	S_0	S_2	0
S_2	S_0	S_3	0
S_3	S_0	S_3	1

  - Which states are equivalent?
    - None – no state reduction possible



- Step 3: state assignment
  - How many flip-flops?
    - Two

Example 1: Sequence Detector

- Step 4: Binary coded state table
  - Name flip-flops A and B

current state		next state				output
		x=0		x=1		
A	B	A	B	A	B	
0	0	0	0	0	1	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0
1	1	0	0	1	1	1

- Step 5: Choose type of flip-flops
  - D flip-flop
  - Characteristic equation: $Q(t+1)=D_Q$

Example 1: Sequence Detector

- Step 6: derive flip-flop input equations and output equation
  - Use state table

  - $A(t+1) = D_A(A,B,x) = \Sigma(3,5,7)$

  - $B(t+1) = D_B(A,B,x) = \Sigma(1,5,7)$

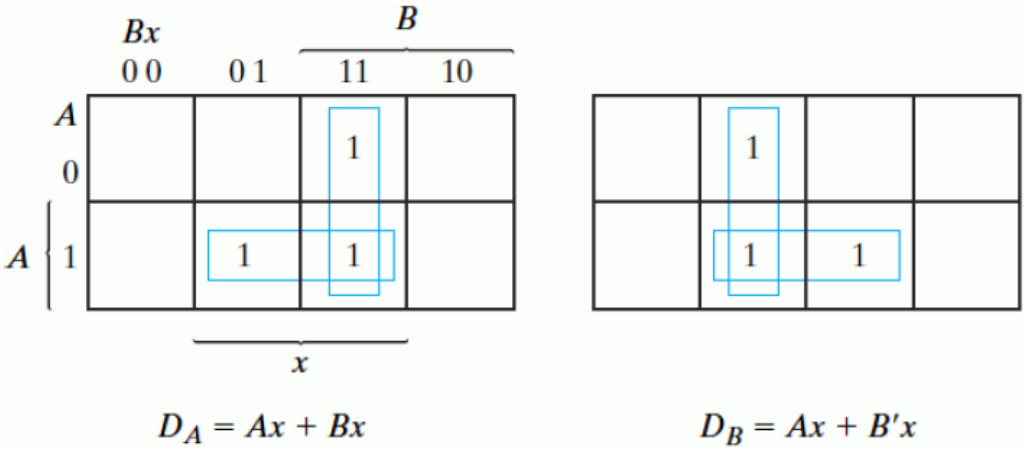
  - $y(A,B,x) = \Sigma(6,7)$ or $y(A,B) = \Sigma(3)$

current state		input	next state		output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

Example 1: Sequence Detector

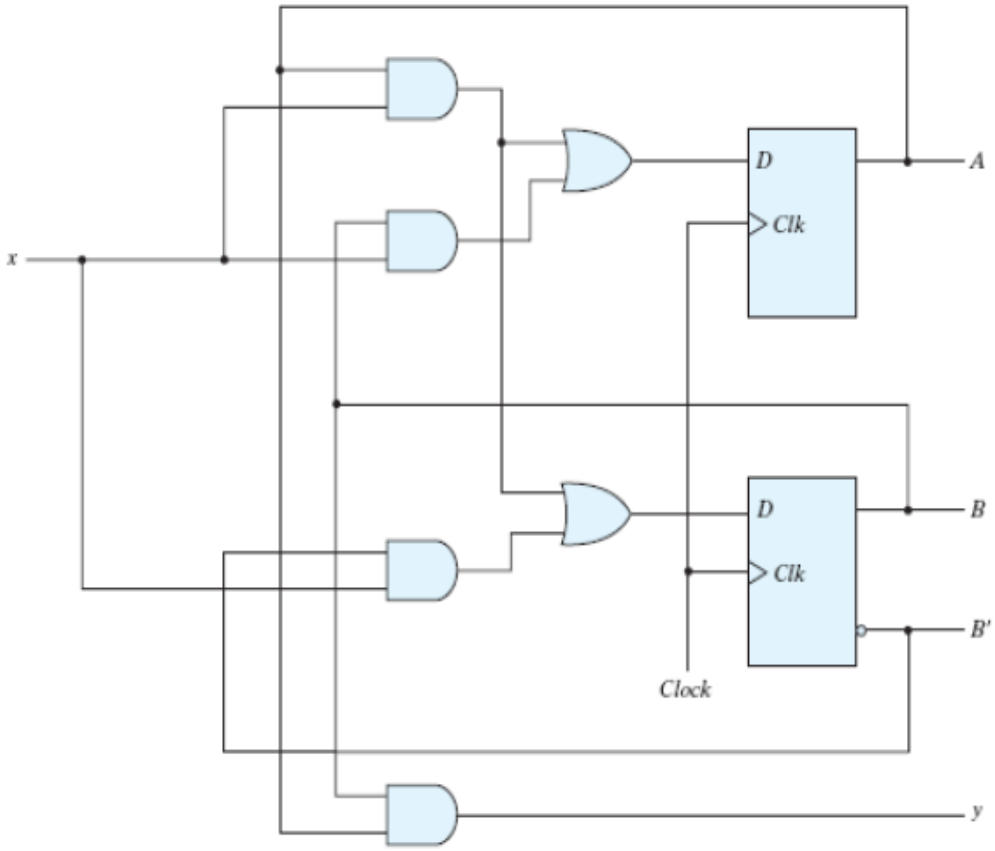
- Step 6b: minimize equations
  - $A(t+1) = \Sigma(3,5,7)$
  - $B(t+1) = \Sigma(1,5,7)$
  - $y(A,B) = \Sigma(3)$ – easy: $y = AB$

  - Karnaugh maps for A and B

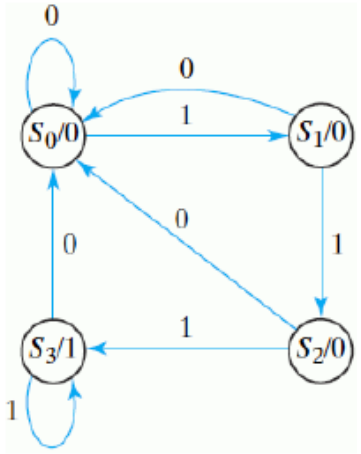


Example 1: Sequence Detector

- Step 7: Circuit diagram
  - $D_A = Ax + Bx$
  - $D_B = Ax + B'x$
  - $Y = AB$



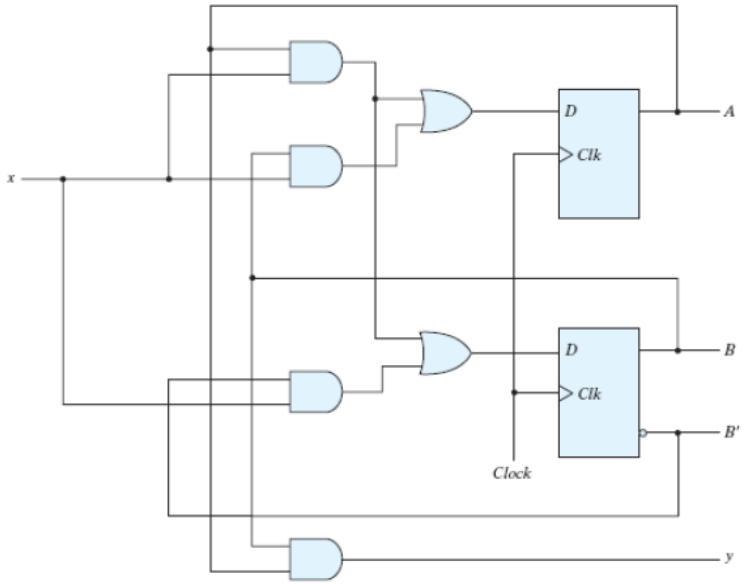
Sequential Circuit Design (FSM Approach)



- Example 1 (review) – sequence detector
  - Design a circuit that outputs a 1 when a sequence "111" has been applied to input, and 0 otherwise

- **Steps**

  1: derive state diagram
  2: reduce number of states
  3: encode the states
  4: create state table
  5: choose flip-flop type
  6: derive and minimize FF
  and output equations (use D FFs)
  7: construct circuit diagram



State Reduction

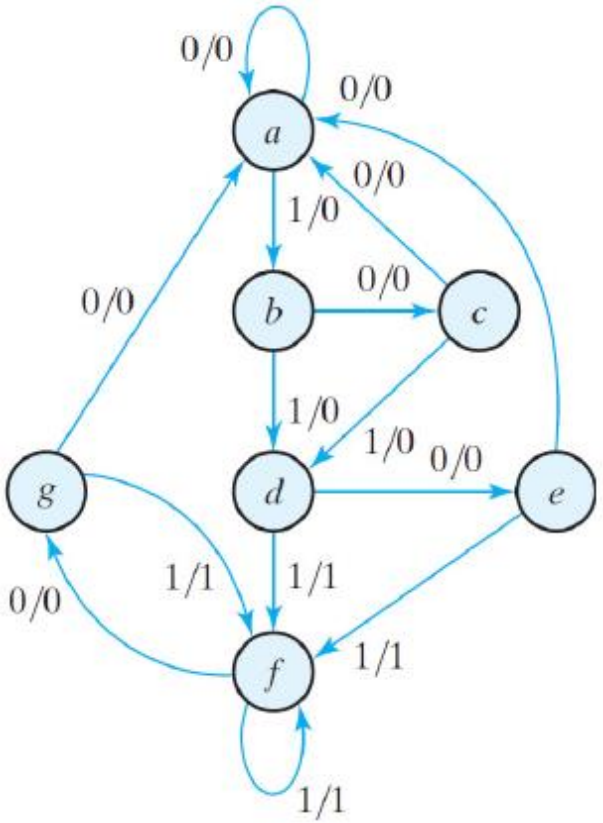
- Step 2 in the FSM design process: state reduction
  - Some states are equivalent
    - Have same next state and produce same output under same input
  - Can be merged together to minimize the number of states

State Reduction

- What do equivalent states "look like"?
  - State diagram: arrows from two states go to same next state and output is the same
  - State table: next state and output entry is same
  - State equations:
    - $Q_1(t+1)…Q_n(t+1)$ and output is the same for two different $Q_1(t)…Q_n(t)$ for all inputs
- Which representation is easier to handle?
  - State table

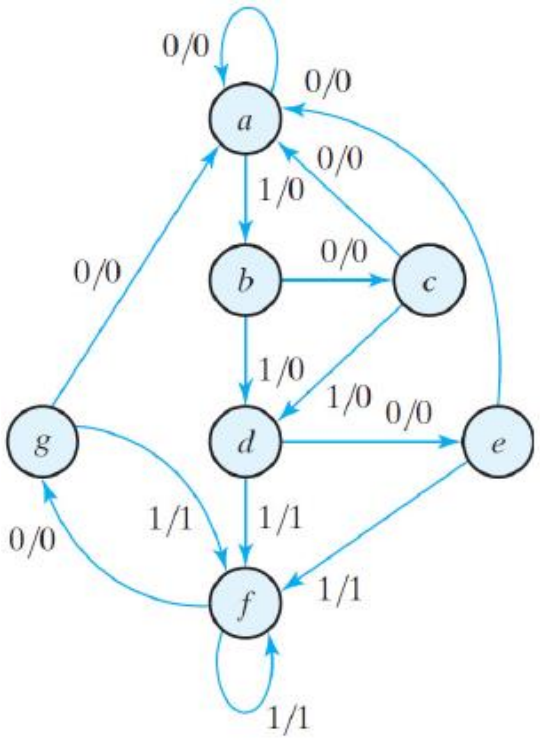
State Reduction: Example

- Reduce, if possible, the following state diagram



State Reduction: Example

- State diagram → State table



→

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

State Reduction: Example

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

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

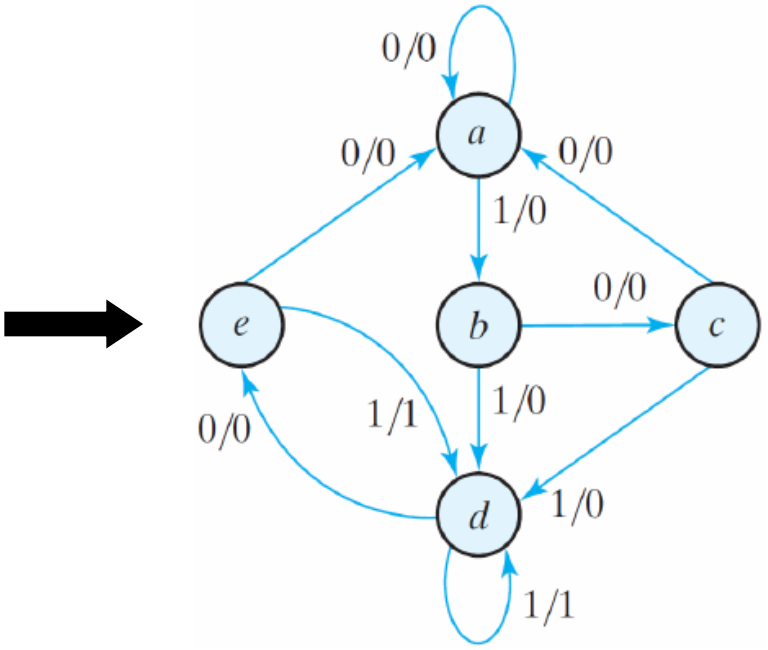
State Reduction: Example

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

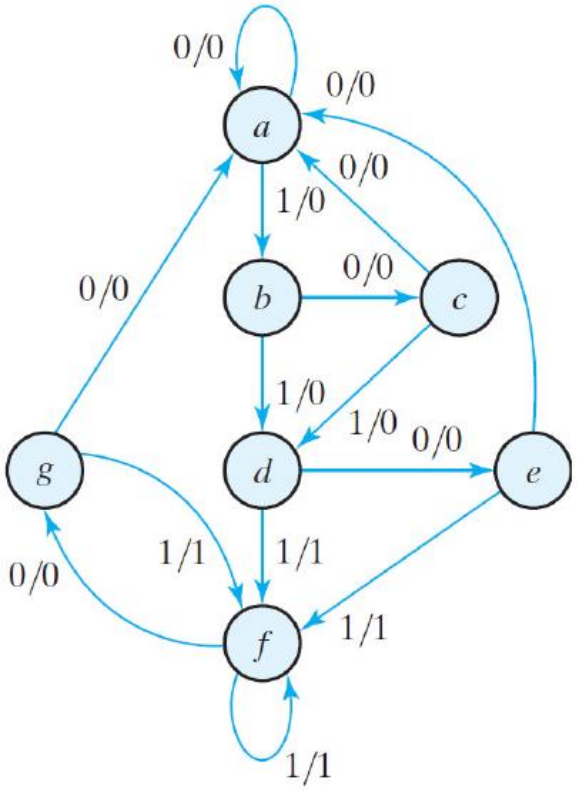
Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

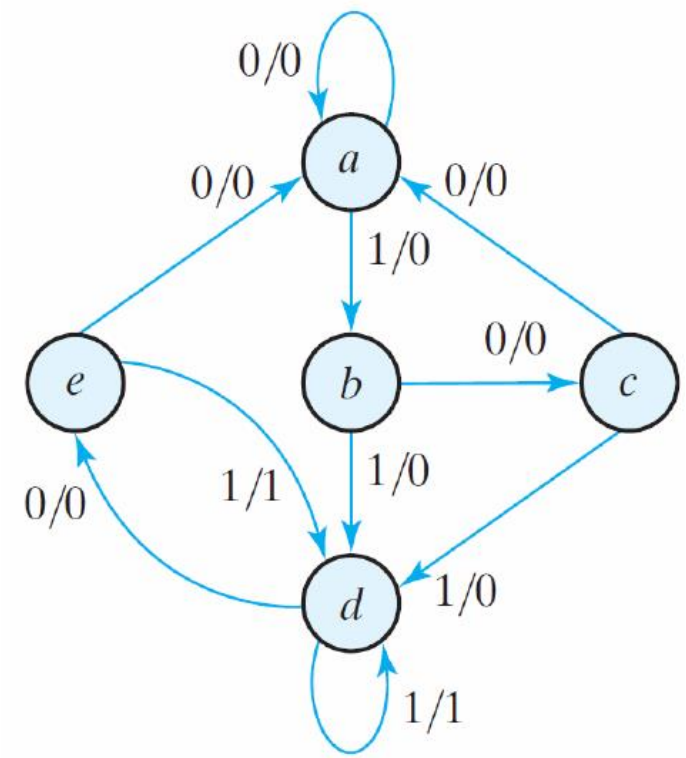
State Reduction: Example

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



State Diagram Comparison





Excitation Tables

- D flip-flops are the easiest to handle
  - State equations are same as input equations
- What about JK and T flip-flops
  - Necessary to derive a relationship between state table and input equations

- JK flip-flop excitation table

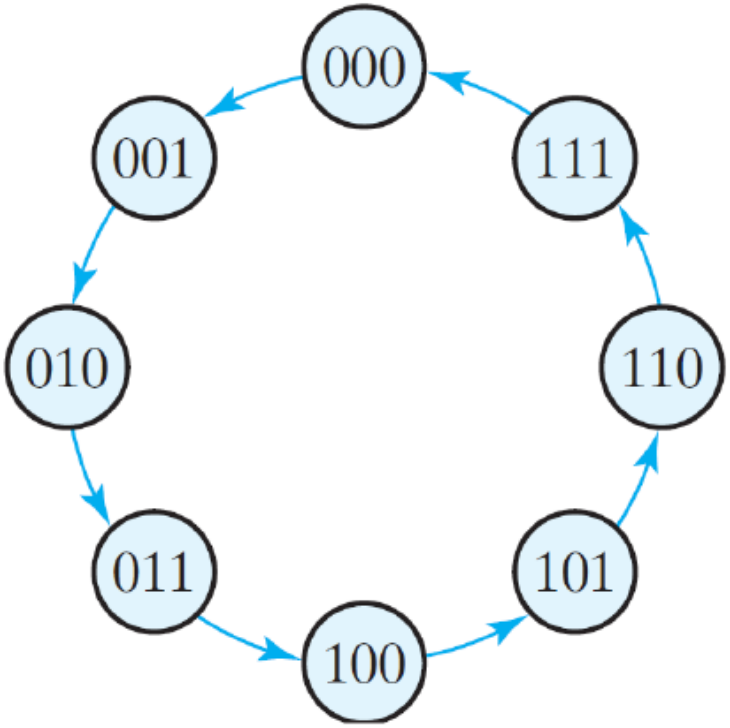
Q(t)	Q(t = 1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

- T flip-flop excitation table

Q(t)	Q(t = 1)	T
0	0	0
0	1	1
1	0	1
1	1	0

Counters: Example with T Flip-flops

- Design a 3-bit binary counter



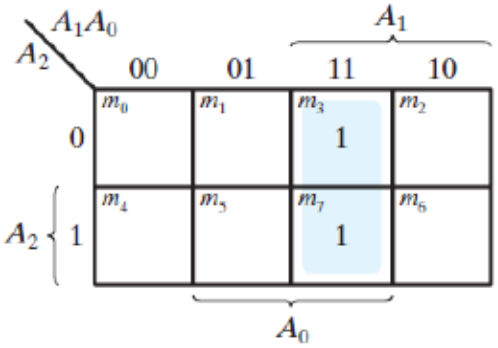
- How many states?
  - Eight
- How many flip-flops?
  - Three
- Which flip-flop?
  - T flip-flop

Counters: Example with T Flip-flops

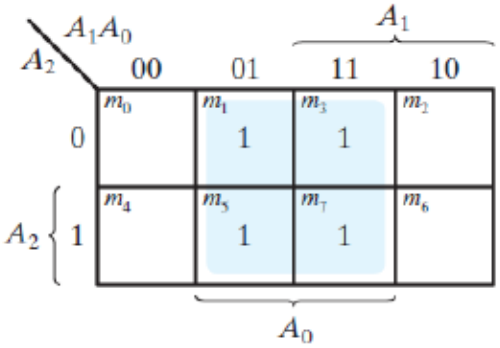
Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

Counters: Example with T Flip-flops

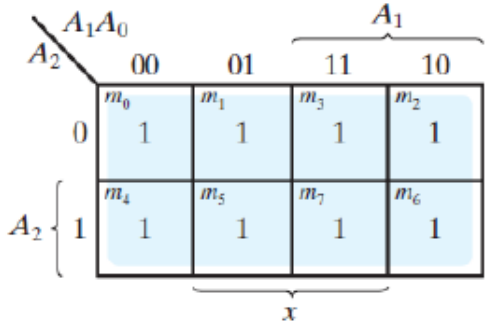
- Minimization using K-maps



$T_{A2} = A_1A_0$



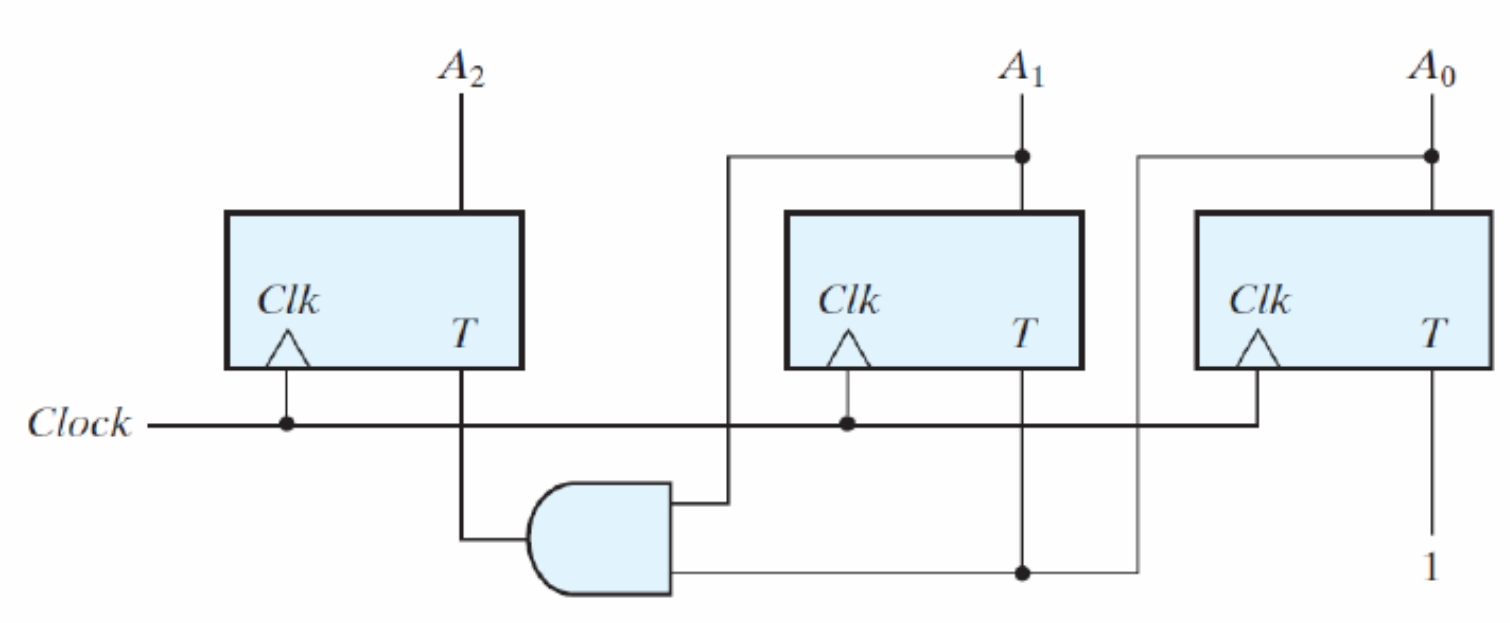
$T_{A1} = A_0$



$T_{A0} = 1$

Counters: Example with T Flip-flops

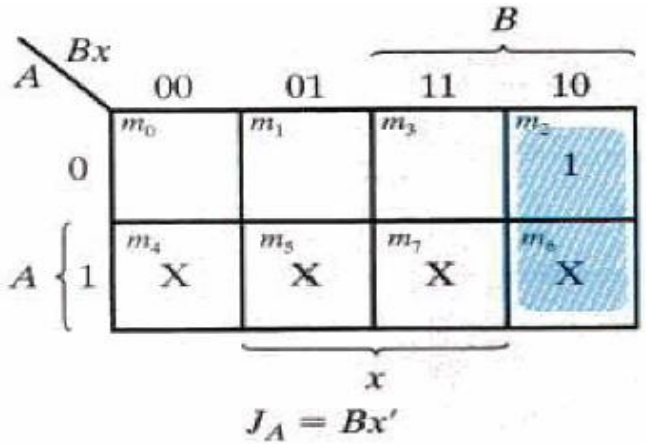
- Circuit diagram



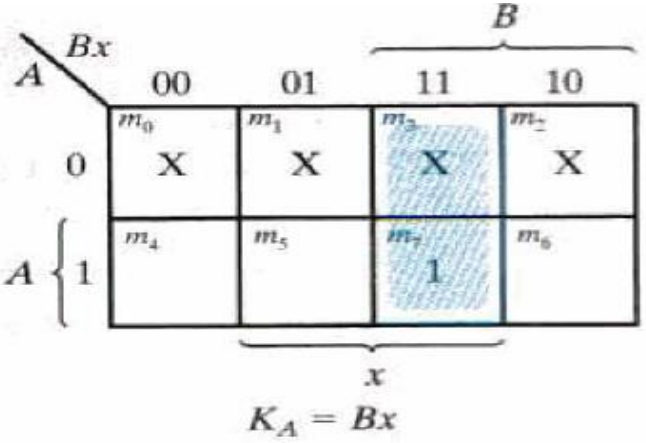
Counters: Example with JK Flip-flops

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

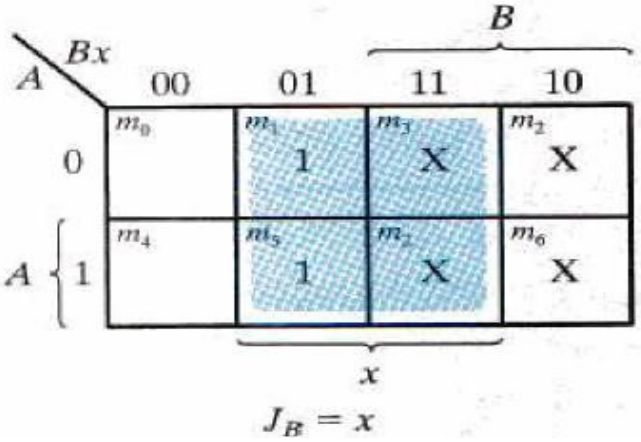
Counters: Example with JK Flip-flops



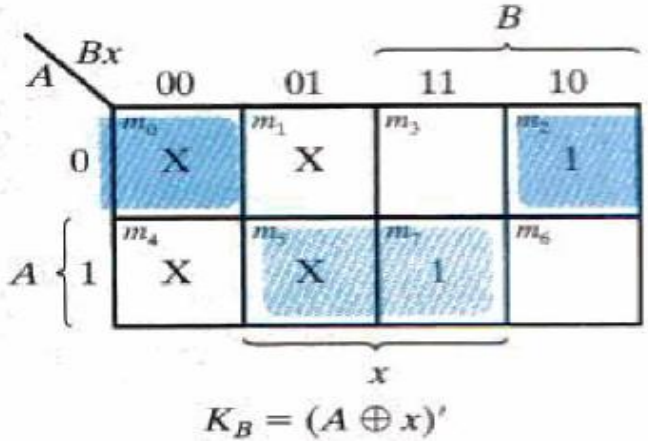
$J_A = Bx'$



$K_A = Bx$



$J_B = x$



$K_B = (A \oplus x)'$

Counters: Example with JK Flip-flops