

# **EMBEDDED SYSTEMS**

## ***CHAPTER -2***

### **HARDWARE DESIGN ISSUES**

## 2. Hardware Design Issues [4 Hrs]

2.1 Combination Logic

2.2 Sequential Logic

2.3 Custom Single-Purpose  
Processor Design

2.4 Optimizing Custom Single-  
Purpose Processors

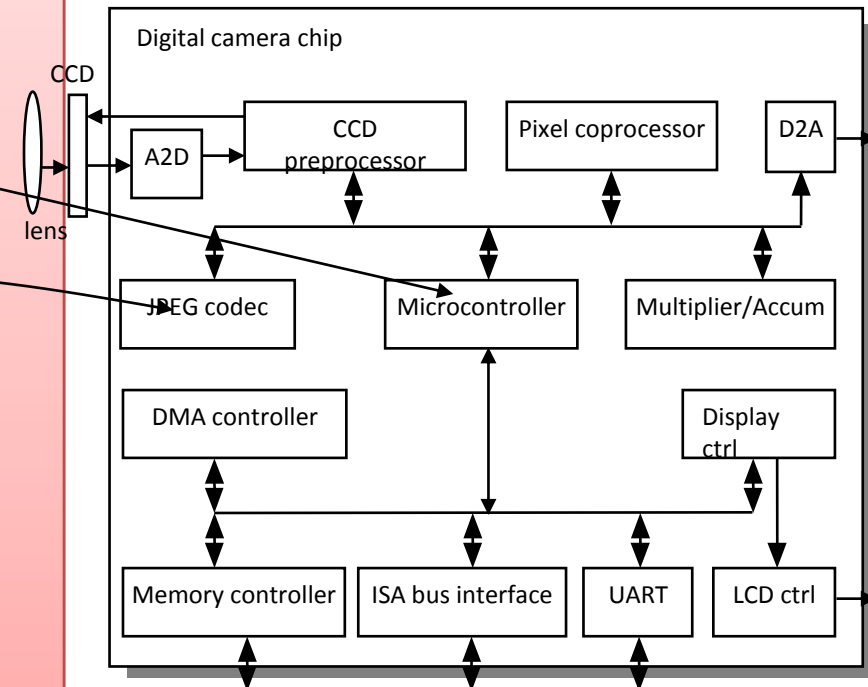
# Introduction

- **Processor**

- Digital circuit that performs a computation tasks
- Controller and datapath
- **General-purpose: variety of computation tasks**
- **Single-purpose: one particular computation task**
- **Custom single-purpose: non-standard task**

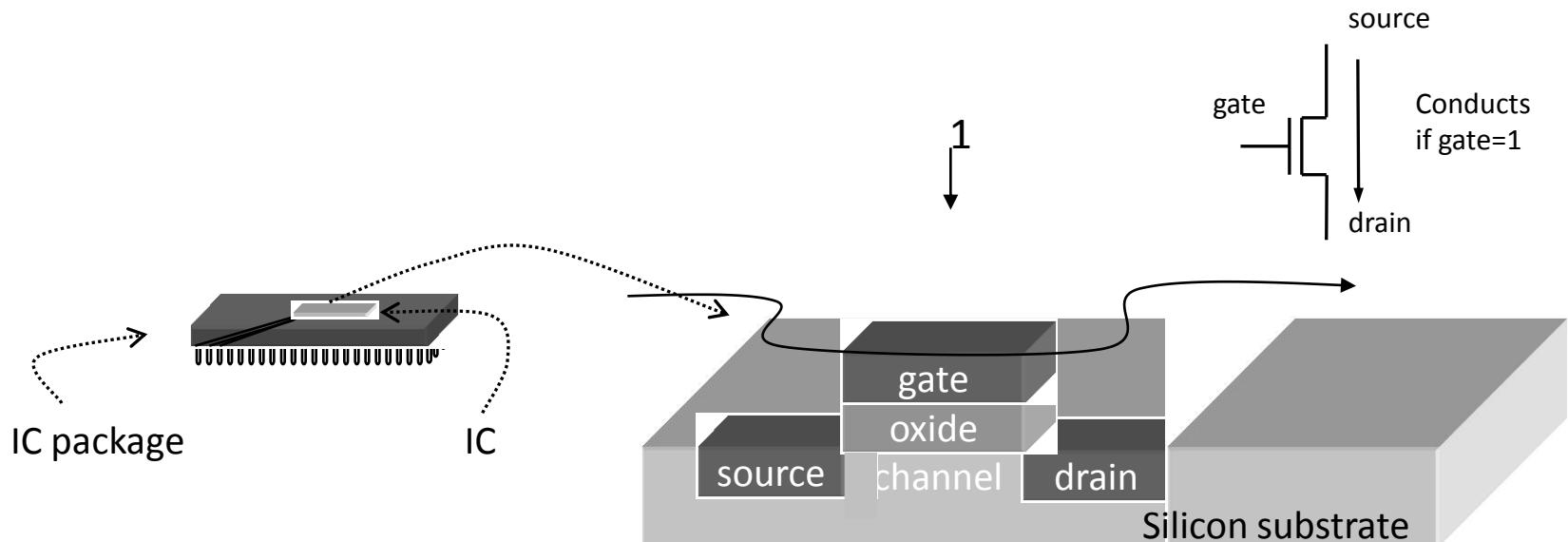
- **A custom single-purpose processor may be**

- Fast, small, low power
- But, high NRE, longer time-to-market, less flexible



# CMOS transistor on silicon

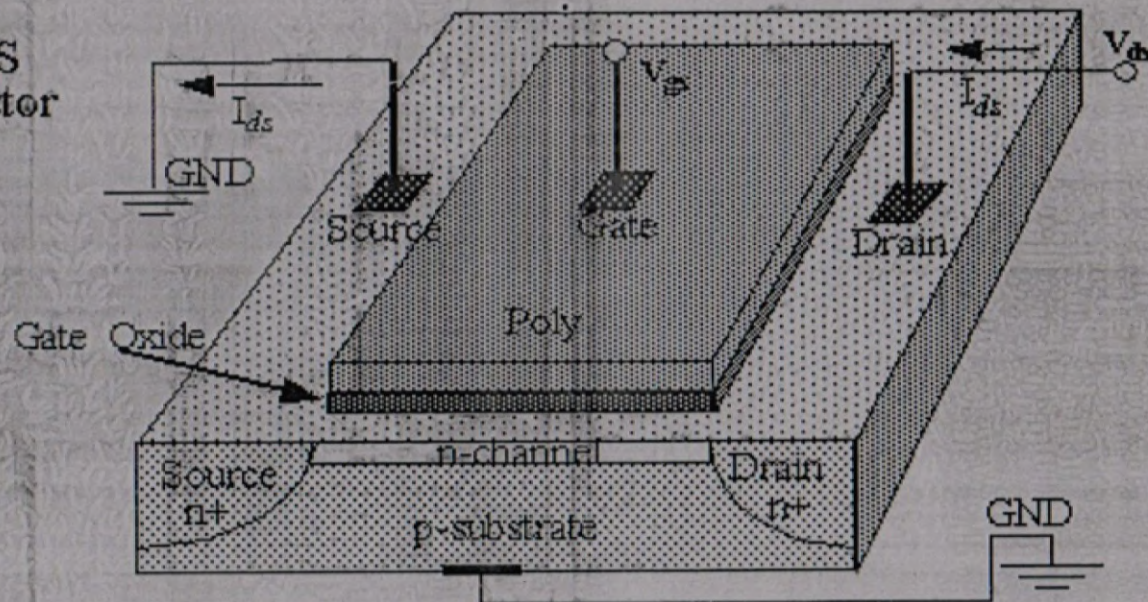
- **Transistor**
  - The basic electrical component in digital systems
  - Acts as an on/off switch
  - Voltage at “gate” controls whether current flows from source to drain
  - this “gate” with a logic gate



## MOS Transistor Definitions

- In normal operation, a positive voltage applied between source and drain ( $V_{ds}$ ).
- No current flows between source and drain ( $I_{ds} = 0$ ) with  $V_{gs} = 0$  because of back to back pn junctions.
- For n-MOS, with  $V_{gs} > V_{th}$ , electric field attracts electrons creating channel.
- Channel is p-type silicon which is inverted to n-type by the electrons attracted by the electric field.

n-MOS transistor



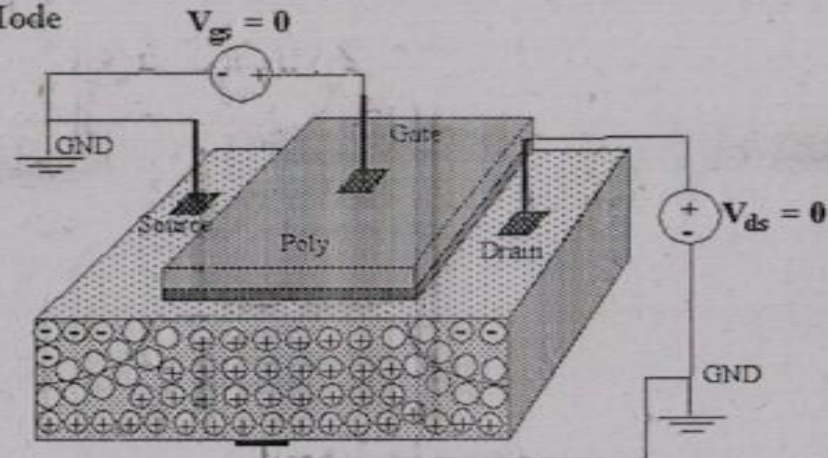


## n-MOS Enhancement Transistor Physics

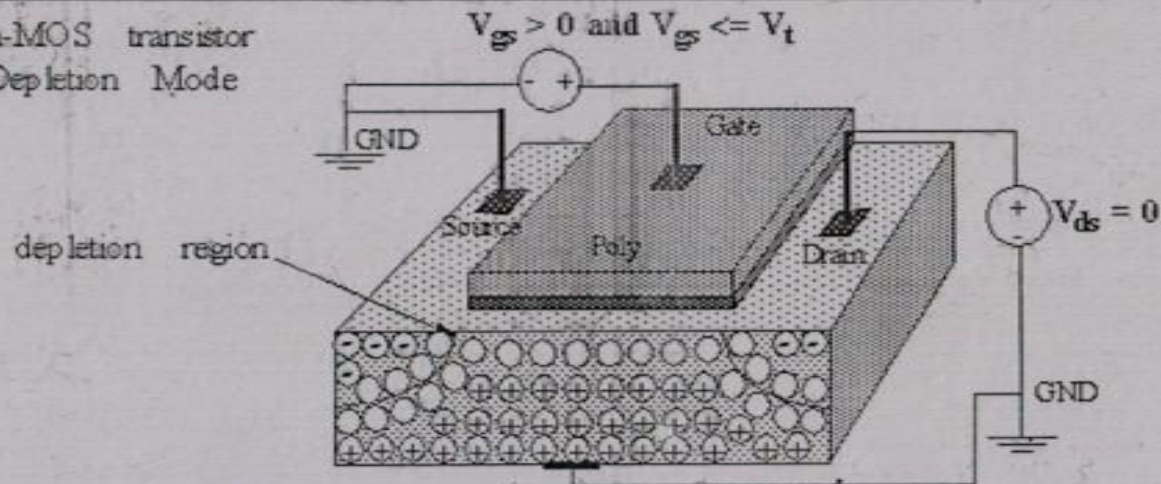
• Three modes based on the magnitude of  $V_{gs}$  : accumulation, depletion and inversion.

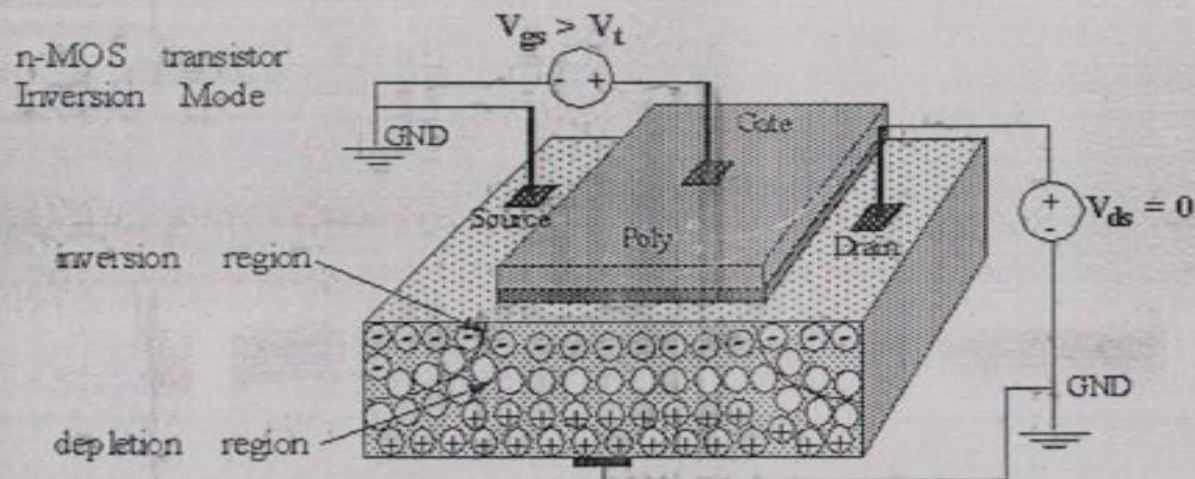
n-MOS transistor

Accumulation Mode



n-MOS transistor  
Depletion Mode



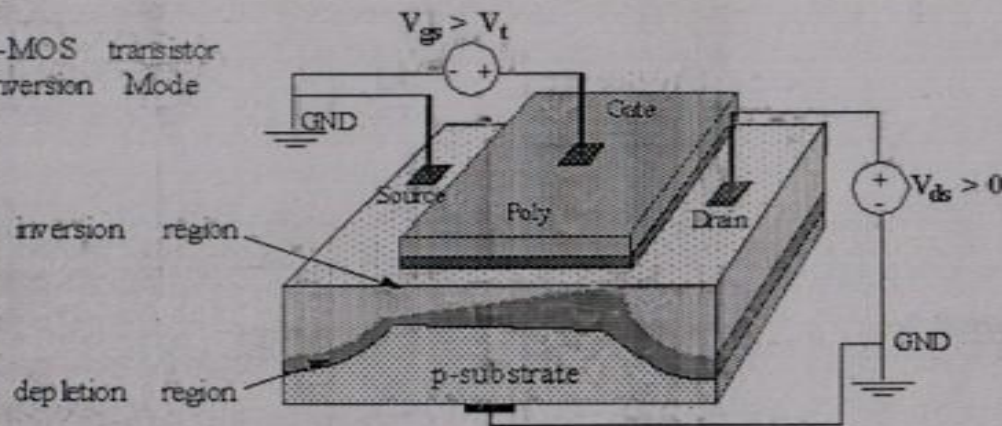


### n-MOS Enhancement Transistor

- With  $V_{ds}$  non-zero, the channel becomes smaller closer to the drain.
- When  $V_{ds} \leq V_{gs} - V_t$  (e.g.  $V_{ds} = 3V$ ,  $V_{gs} = 5V$  and  $V_t = 1V$ ), the channel reaches the drain (since  $V_{gd} > V_t$ ).
- This is termed *linear*, *resistive* or *nonsaturated* region.  $I_{ds}$  is a function of both  $V_{gs}$  and  $V_{ds}$ .

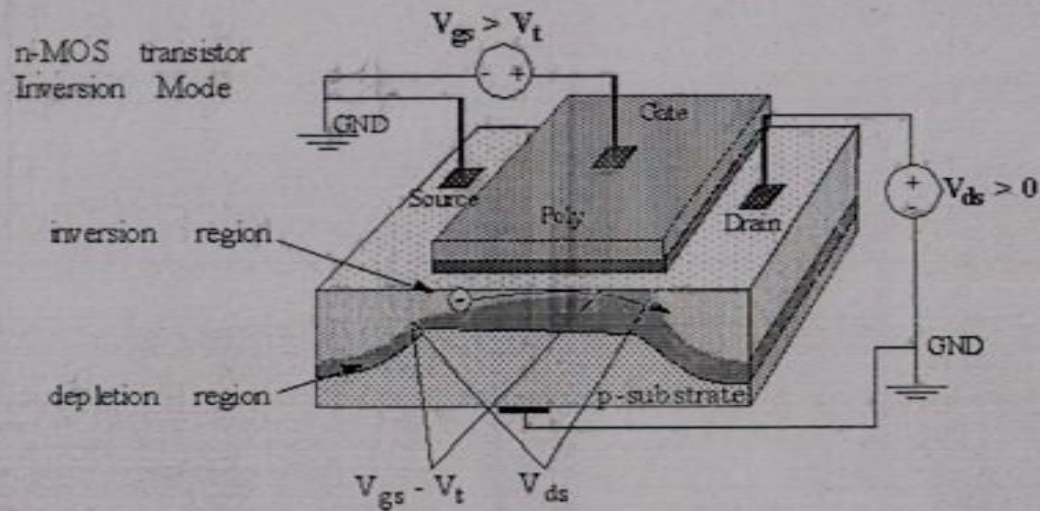


n-MOS transistor  
Inversion Mode



### n-MOS Enhancement Transistor

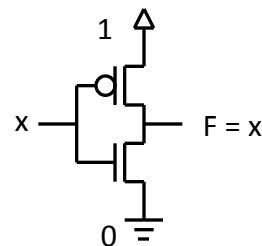
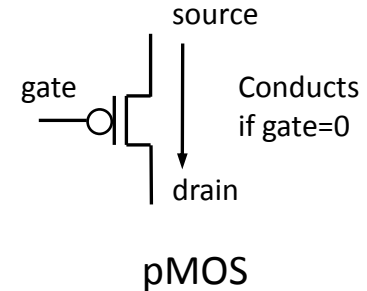
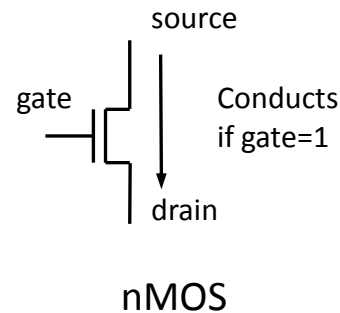
- When  $V_{ds} > V_{gs} - V_t$  (e.g.  $V_{ds} = 5V$ ,  $V_{gs} = 5V$  and  $V_t = 1V$ ), the channel is pinched off close to the drain (since  $V_{gd} < V_t$ ).
- This is termed saturated region.  $I_{ds}$  is a function of  $V_{gs}$ , almost independent of  $V_{ds}$ .



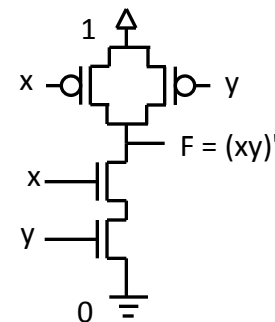


# CMOS transistor implementations

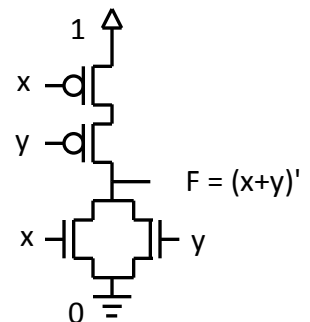
- **Complementary Metal Oxide Semiconductor**
- **refer to logic levels**
  - Typically 0 is 0V, 1 is 5V
- **Two basic CMOS types**
  - nMOS conducts if gate=1
  - pMOS conducts if gate=0
  - Hence “complementary”
- **Basic gates**
  - Inverter, NAND, NOR



inverter

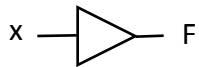


NAND gate



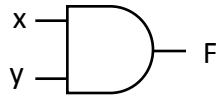
NOR gate

# Basic logic gates



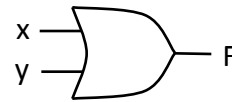
x	F
0	0
1	1

$F = x$   
Driver



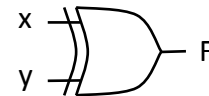
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = x y$   
AND



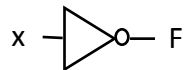
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = x + y$   
OR



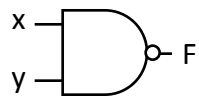
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

$F = x \oplus y$   
XOR



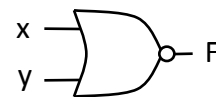
x	F
0	1
1	0

$F = x'$   
Inverter



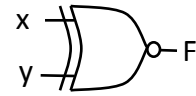
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

$F = (x y)'$   
NAND



x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

$F = (x + y)'$   
NOR



x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

$F = x \odot y$   
XNOR

# Combinational logic design

## A) Problem description

**y** is 1 if **a** is to 1, or **b** and **c** are 1. **z** is 1 if **b** or **c** is to 1, but not both, or if all are 1.

## B) Truth table

Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

## C) Output equations

$$y = a'bc + ab'c' + ab'c + abc' + abc$$

$$z = a'b'c + a'bc' + ab'c + abc' + abc$$

## D) Minimized output equations

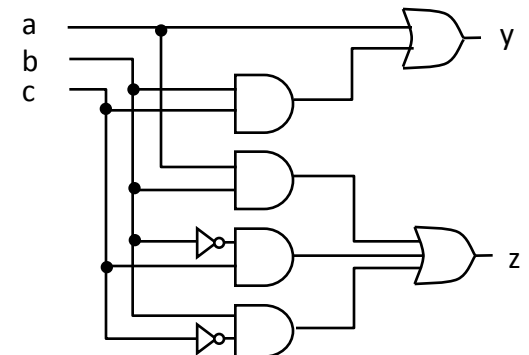
y	a	bc			
		00	01	11	10
0	0	0	0	1	0
1	1	1	1	1	1

$$y = a + bc$$

z	a	bc			
		00	01	11	10
0	0	0	1	0	1
1	1	0	1	1	1

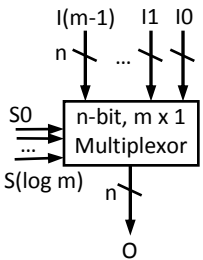
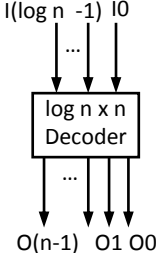
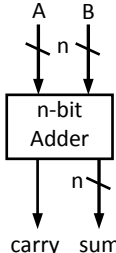
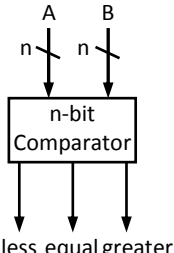
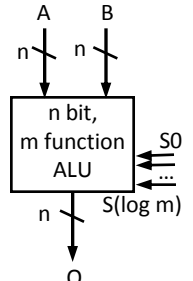
$$z = ab + b'c + bc'$$

## E) Logic Gates

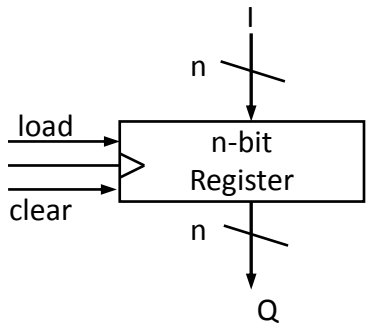
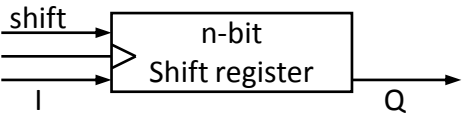
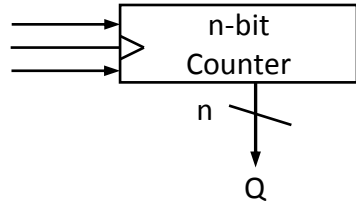




# Combinational components

				
<p><b>O =</b>  <b>I0 if S=0..00</b>  <b>I1 if S=0..01</b>  <b>...</b>  <b>I(m-1) if S=1..11</b></p>	<p><b>O0 =1 if I=0..00</b>  <b>O1 =1 if I=0..01</b>  <b>...</b>  <b>O(n-1) =1 if I=1..11</b></p>	<p><b>sum = A+B</b>  <b>(first n bits)</b>  <b>carry = (n+1)'th</b>  <b>bit of A+B</b></p>	<p><b>less = 1 if A&lt;B</b>  <b>equal =1 if A=B</b>  <b>greater=1 if A&gt;B</b></p>	<p><b>O = A op B</b>  <b>op determined</b>  <b>by S.</b></p>
	<p><b>With enable</b>  <b>input e → all O's</b>  <b>are 0 if e=0</b></p>	<p><b>With carry-in</b>  <b>input Ci→</b>  <b>sum = A + B + Ci</b></p>		<p><b>May have status</b>  <b>outputs carry,</b>  <b>zero, etc.</b></p>

# Sequential components

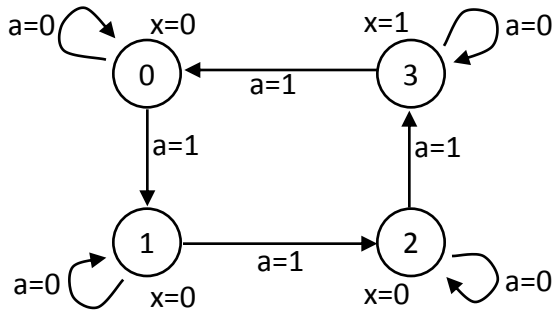
		
<p><b>Q =</b>  <b>0 if clear=1,</b>  <b>I if load=1 and clock=1,</b>  <b>Q(previous) otherwise.</b></p>	<p><b>Q = lsb</b>  <b>- Content shifted</b>  <b>- I stored in msb</b></p>	<p><b>Q =</b>  <b>0 if clear=1,</b>  <b>Q(prev)+1 if count=1</b>  <b>and clock=1.</b></p>

# Sequential logic design

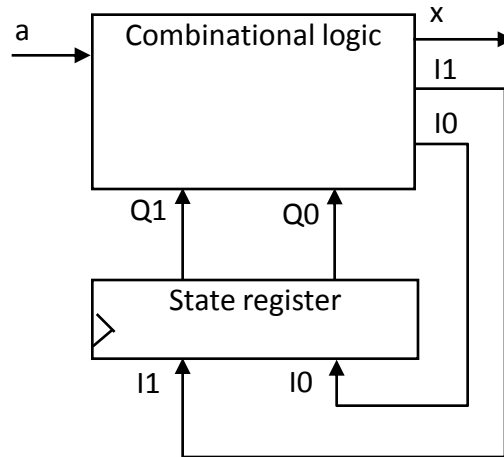
## A) Problem Description

You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles

B) State Diagram



C) Implementation Model



D) State Table (Moore-type)

Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design



# Sequential logic design (cont.)

E) Minimized Output Equations

I1	Q1Q0			
	00	01	11	10
a				
0	0	0	1	1
1	0	1	0	1

$$I1 = Q1'Q0a + Q1a' + Q1Q0'$$

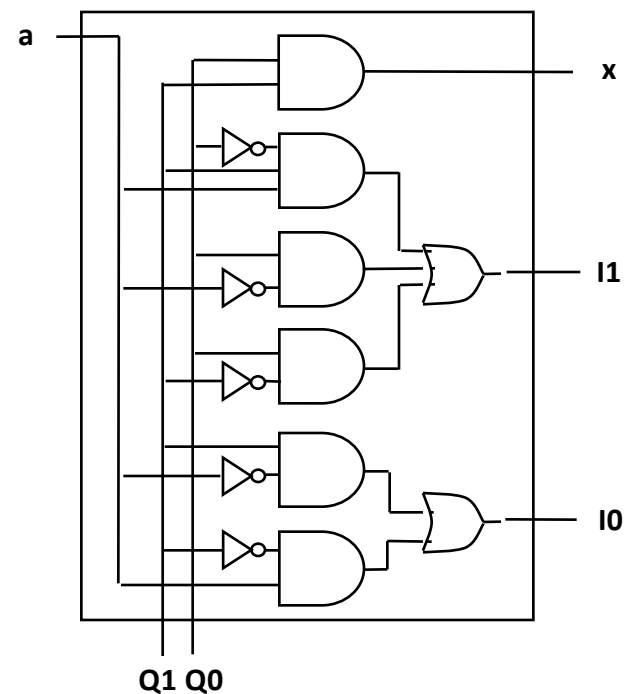
I0	Q1Q0			
	00	01	11	10
a				
0	0	1	1	0
1	1	0	0	1

$$I0 = Q0a' + Q0'a$$

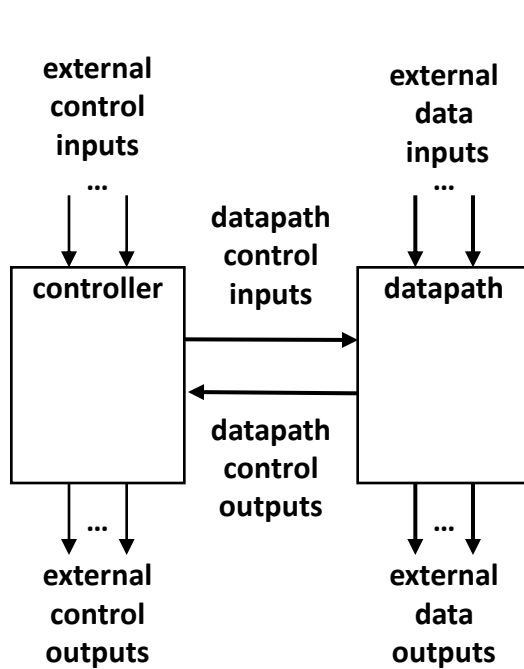
x	Q1Q0			
	00	01	11	10
a				
0	0	0	1	0
1	0	0	1	0

$$x = Q1Q0$$

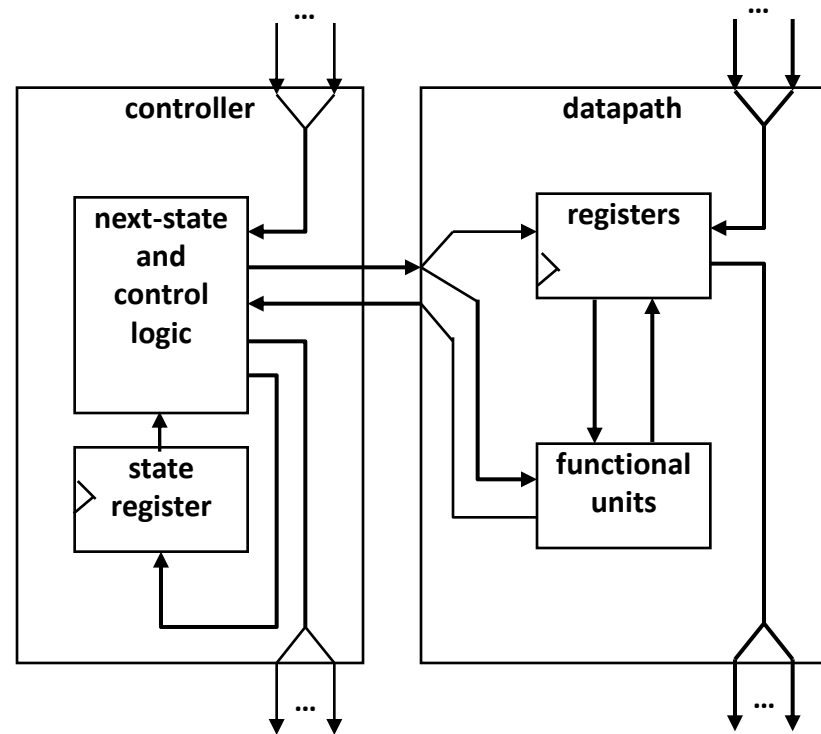
F) Combinational Logic



# Custom single-purpose processor basic model



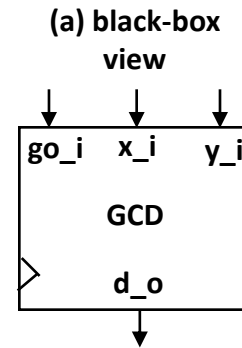
**controller and datapath**



**a view inside the controller and datapath**

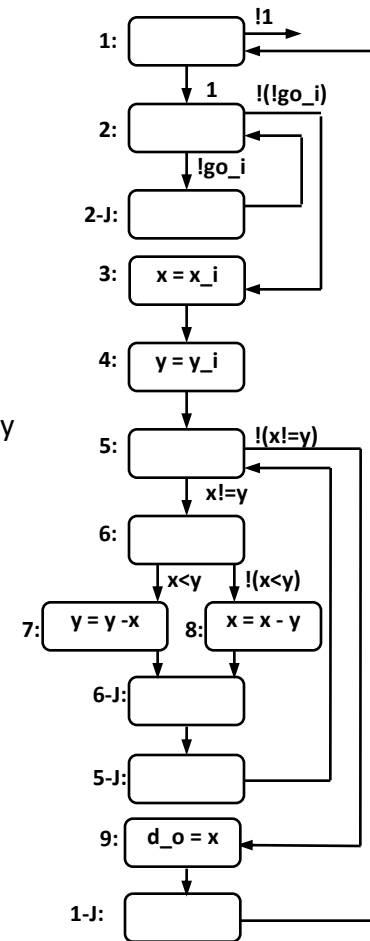
# Example: Greatest Common Divisor

- First create algorithm
- Convert algorithm to “complex” state machine
  - Known as FSMD: finite-state machine with datapath
  - Can use templates to perform such conversion



(b) desired functionality

```
0: int x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
8:     else
9:       x = x - y;
10:  }
11:  d_o = x;
12: }
```

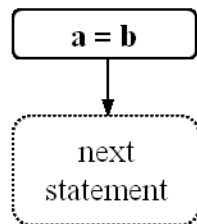




# State diagram templates

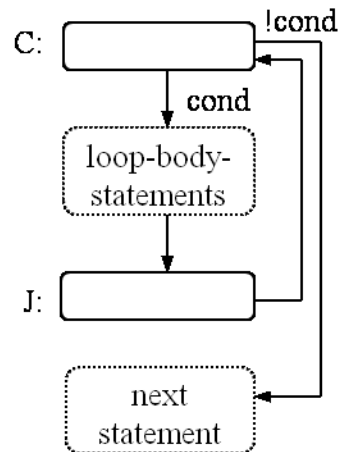
Assignment statement

**a = b**  
next statement



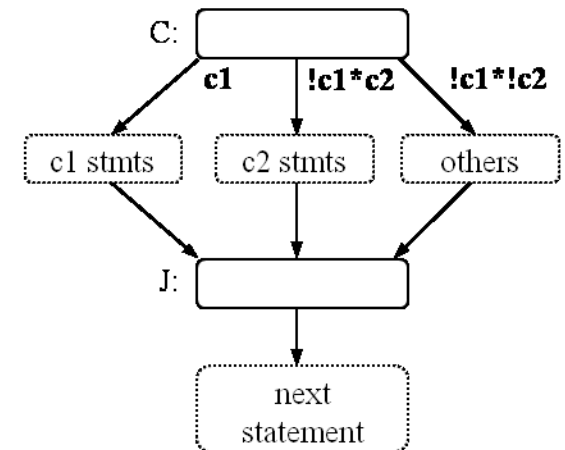
Loop statement

```
while (cond) {  
    loop-body-  
    statements  
}  
next statement
```



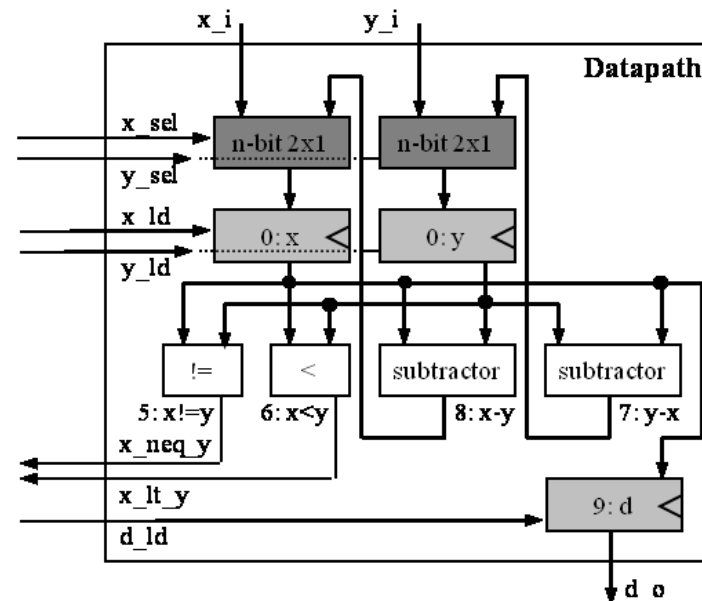
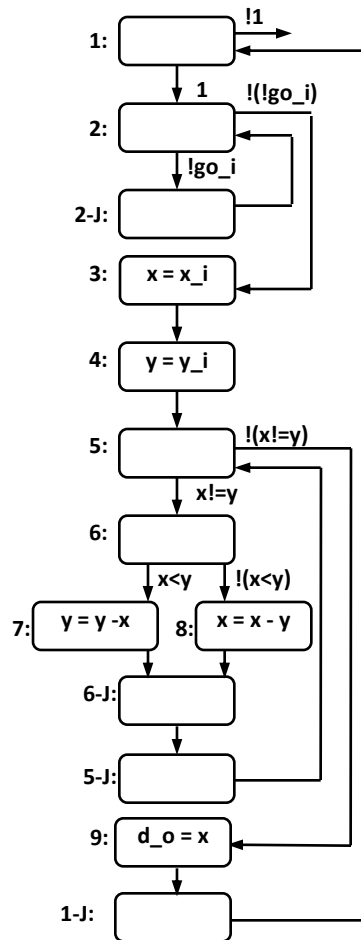
Branch statement

```
if (c1)  
    c1 stmts  
else if c2  
    c2 stmts  
else  
    other stmts  
next statement
```

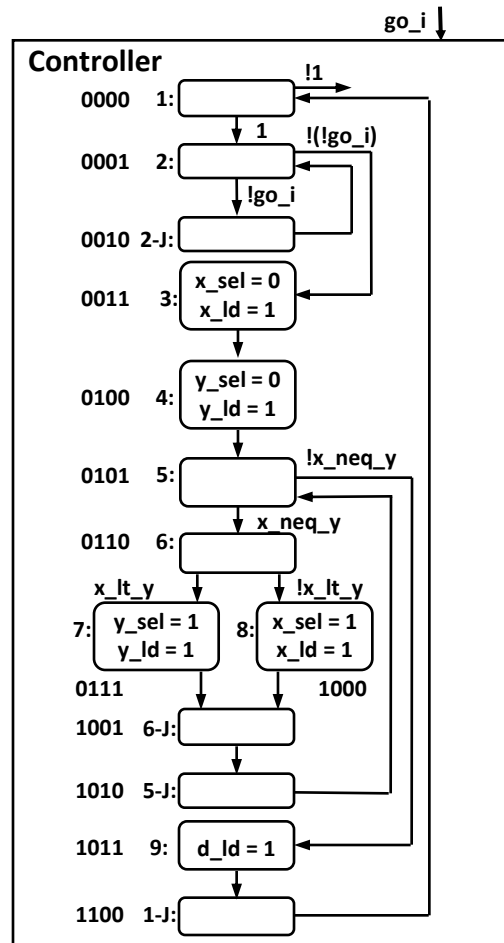
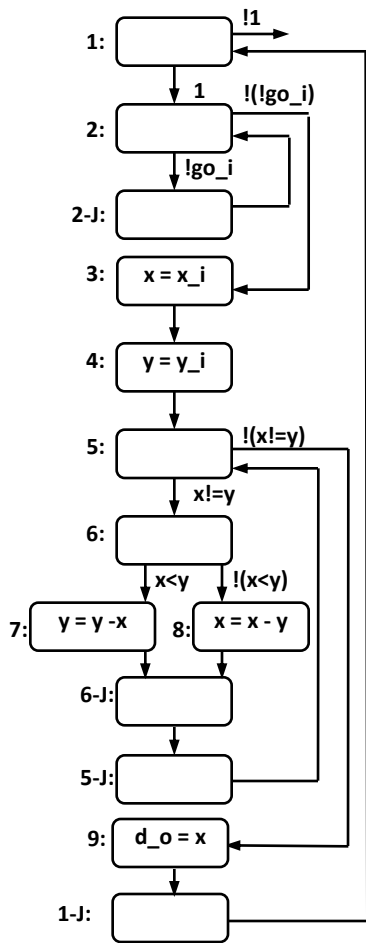


# Creating the datapath

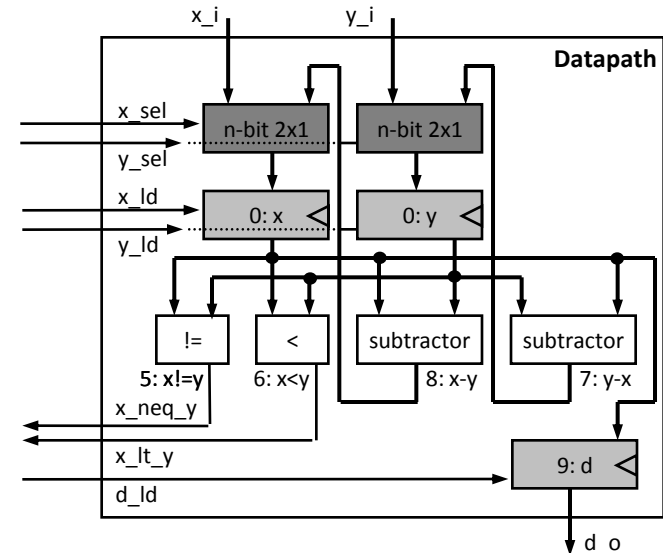
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output



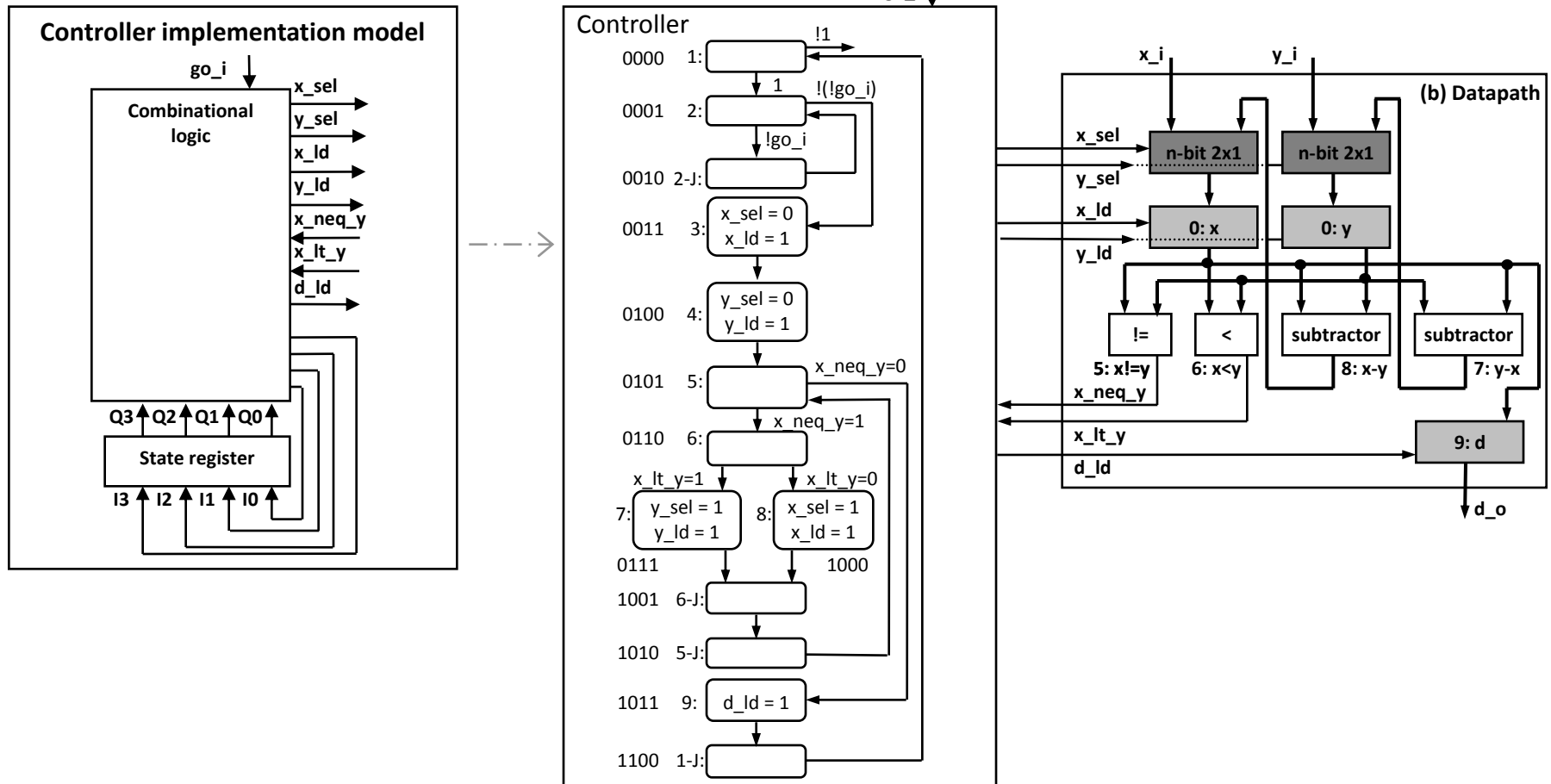
# Creating the controller's FSM



- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



# Splitting into a controller and datapath



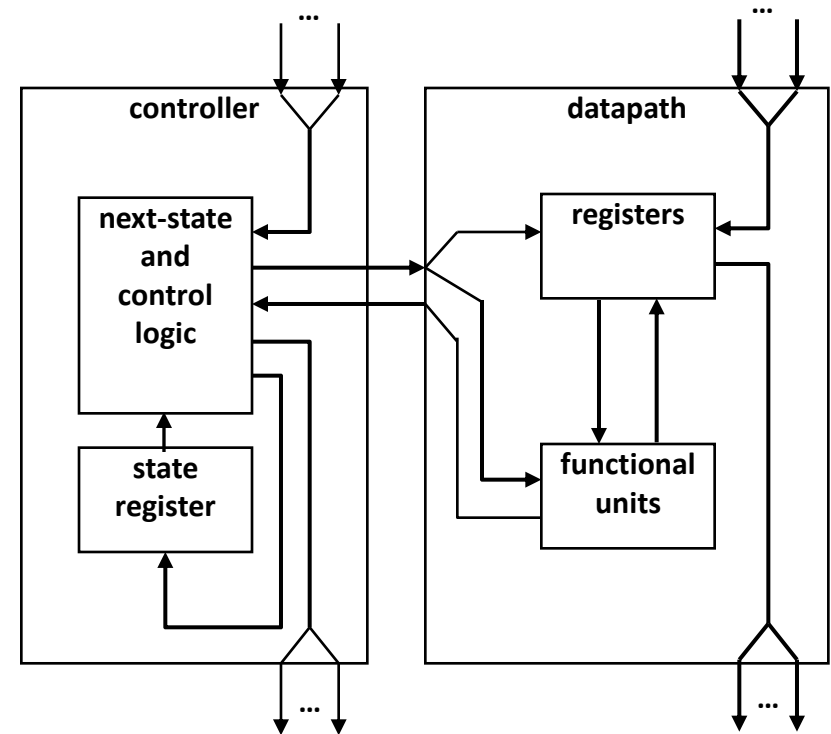
# Controller state table for the Greatest Common Divisor (GCD) example

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0



# Completing the GCD custom single-purpose processor design

- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps

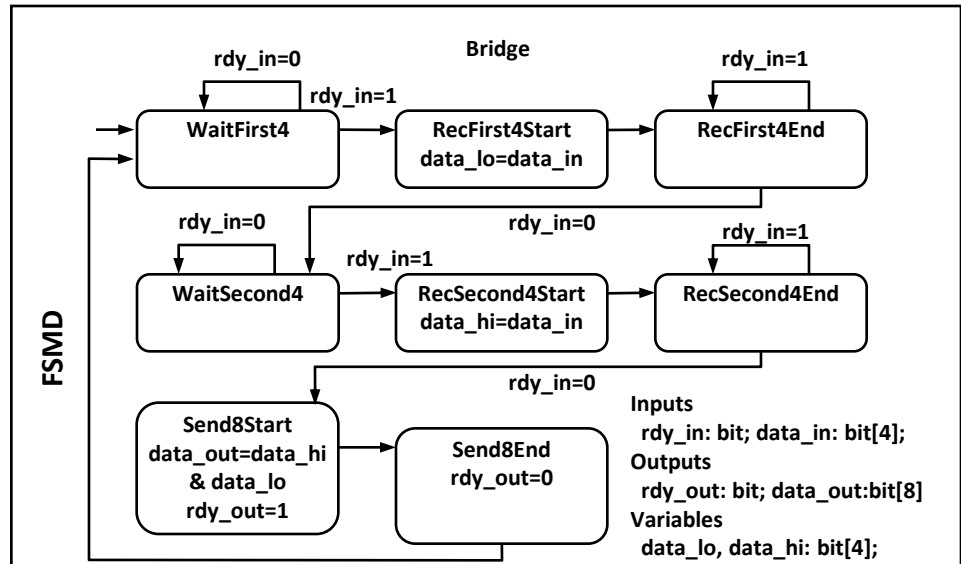
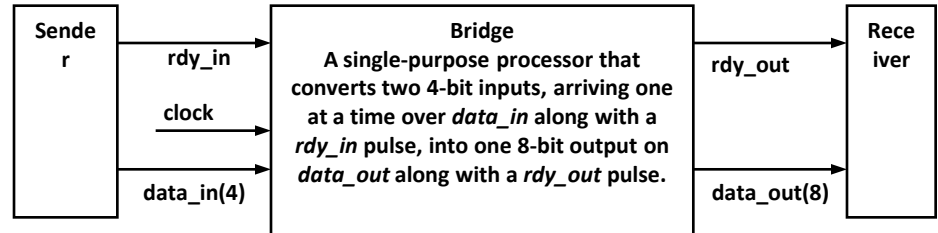


a view inside the controller and datapath

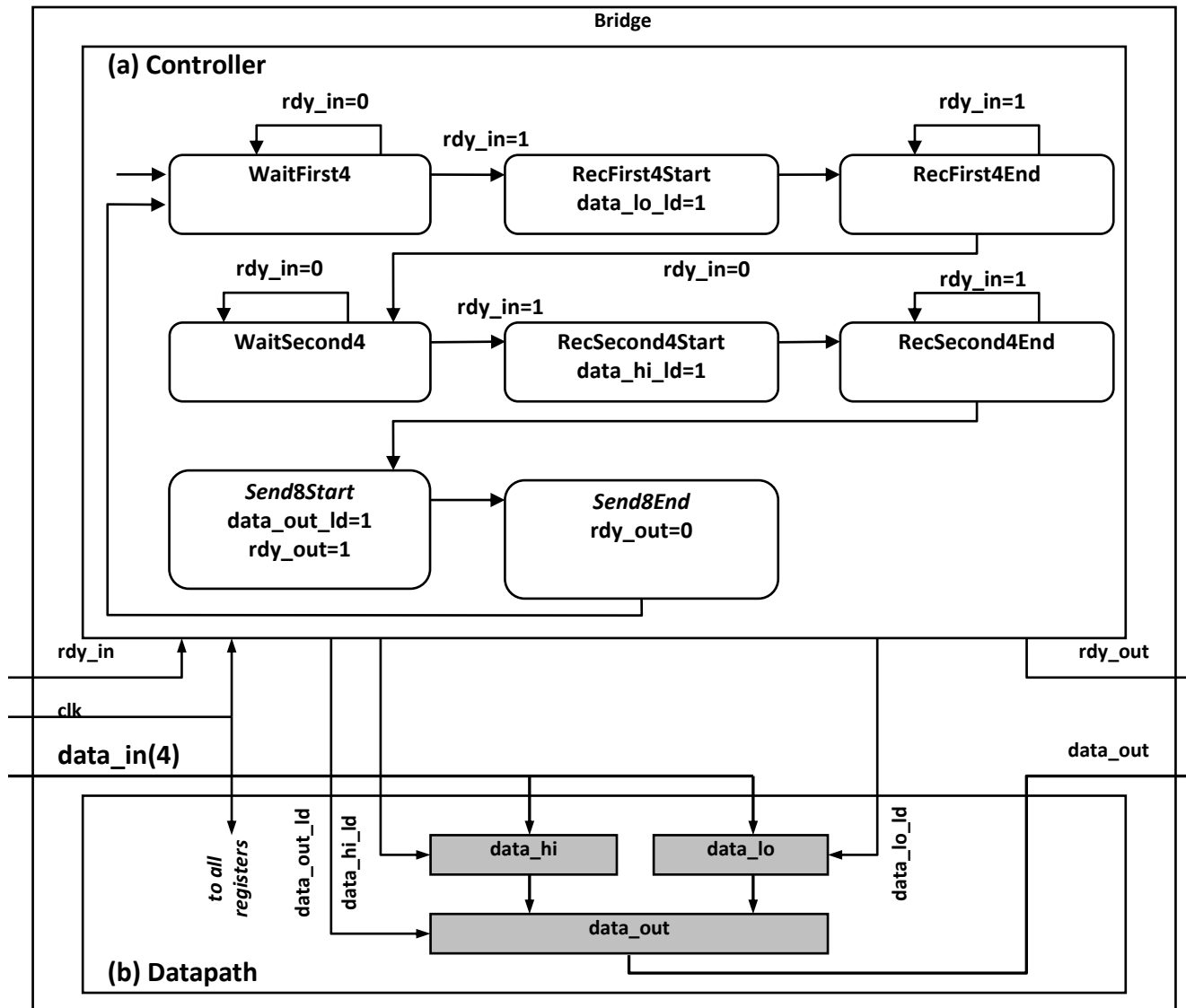
# RT-level custom single-purpose processor design

- We often start with a state machine
  - Rather than algorithm
  - Cycle timing often too central to functionality
- Example
  - Bus bridge that converts 4-bit bus to 8-bit bus
  - Start with FSMD
  - Known as register-transfer (RT) level
  - Exercise: complete the design

Problem Specification



# RT-level custom single-purpose processor design (cont')



# Optimizing single-purpose processors

- **Optimization is the task of making design metric values the best possible**
- **Optimization opportunities**
  - original program
  - FSMD
  - datapath
  - FSM

# Optimizing the original program

- **Analyze program attributes and look for areas of possible improvement**
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

# Optimizing the original program (cont')

original program

```
0: int x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
8:     else
9:       x = x - y;
10:  }
11:  d_o = x;
12: }
```

replace the subtraction  
operation(s) with modulo  
operation in order to speed  
up program

optimized program

```
0: int x, y, r;
1: while (1) {
2:   while (!go_i);
3:   // x must be the larger number
4:   if (x_i >= y_i) {
5:     x=x_i;
6:     y=y_i;
7:   }
8:   else {
9:     x=y_i;
10:    y=x_i;
11:  }
12:  while (y != 0) {
13:    r = x % y;
14:    x = y;
15:    y = r;
16:  }
17:  d_o = x;
18: }
```

GCD(42, 8) - 9 iterations to complete the loop  
x and y values evaluated as follows : (42, 8), (34,  
8), (26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

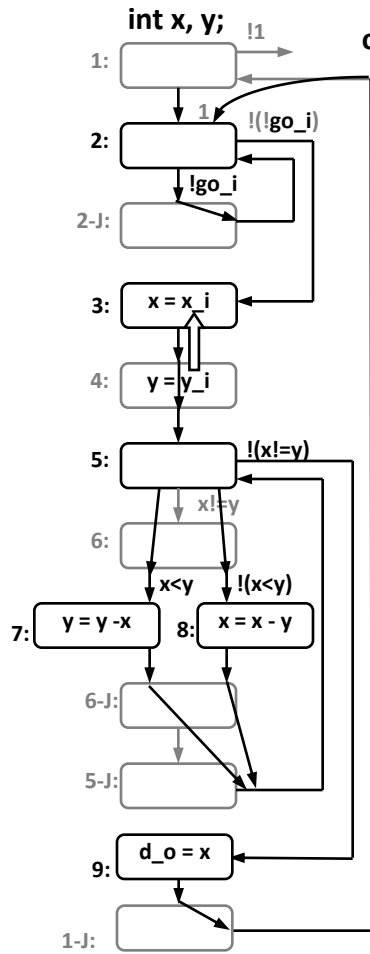
GCD(42,8) - 3 iterations to complete the loop  
x and y values evaluated as follows: (42, 8), (8,2),  
(2,0)



# Optimizing the FSMD

- **Areas of possible improvements**
  - **merge states**
    - states with constants on transitions can be eliminated, transition taken is already known
    - states with independent operations can be merged
  - **separate states**
    - states which require complex operations  
( $a*b*c*d$ ) can be broken into smaller states to reduce hardware size
  - **scheduling**

# Optimizing the FSMD (cont.)



*eliminate state 1* – transitions have constant values

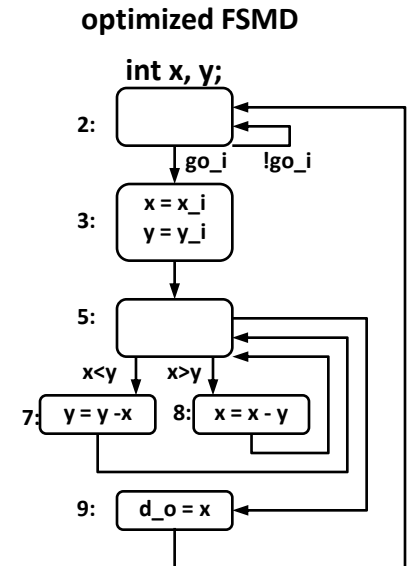
*merge state 2 and state 2J* – no loop operation in between them

*merge state 3 and state 4* – assignment operations are independent of one another

*merge state 5 and state 6* – transitions from state 6 can be done in state 5

*eliminate state 5J and 6J* – transitions from each state can be done from state 7 and state 8, respectively

*eliminate state 1-J* – transition from state 1-J can be done directly from state 9



# Optimizing the datapath

- **Sharing of functional units**
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- **Multi-functional units**
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

# Optimizing the FSM

- **State encoding**
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- **State minimization**
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state