

# Embedded Systems Design: A Unified Hardware/Software Introduction

---

## Chapter 2: Custom single-purpose processors

---

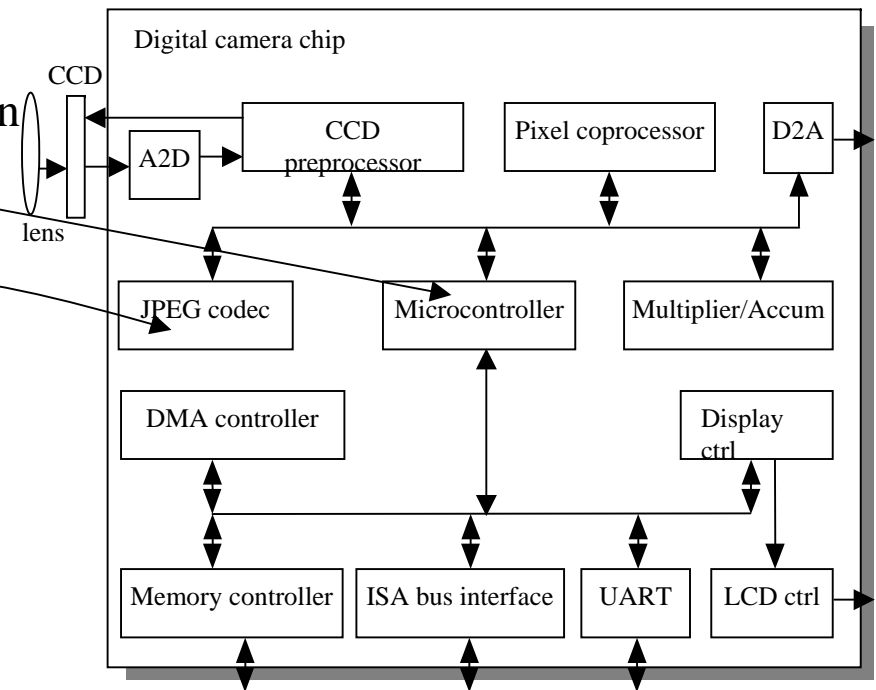
# Outline

---

- Introduction
- Combinational logic
- Sequential logic
- Custom single-purpose processor design
- RT-level custom single-purpose processor design

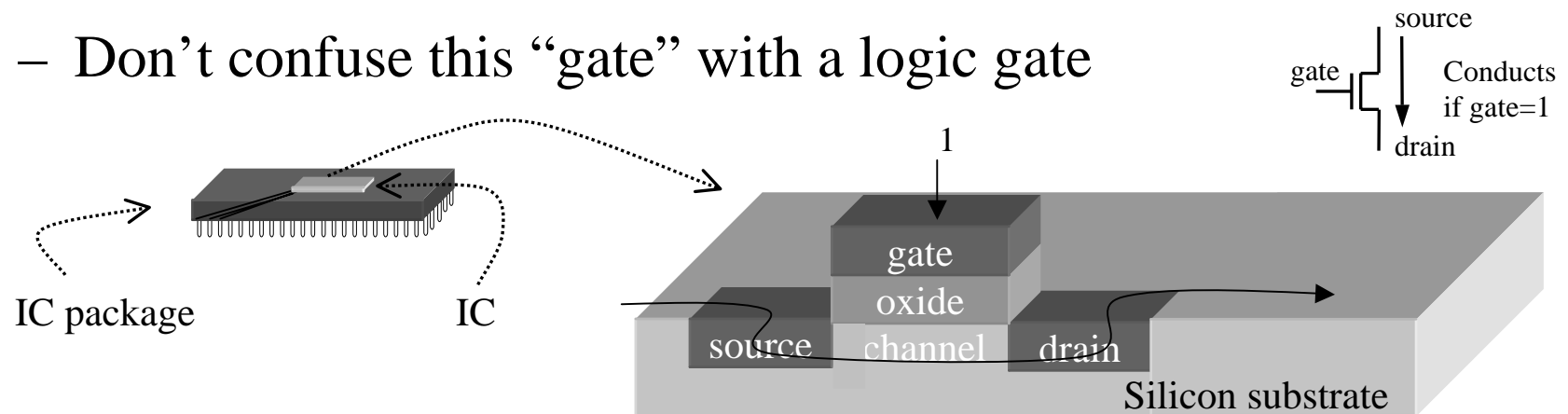
# Introduction

- 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



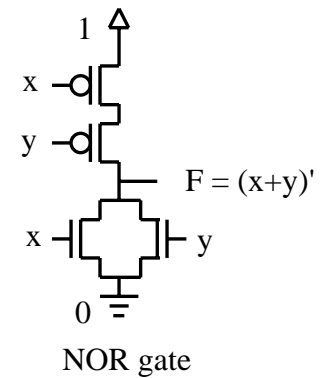
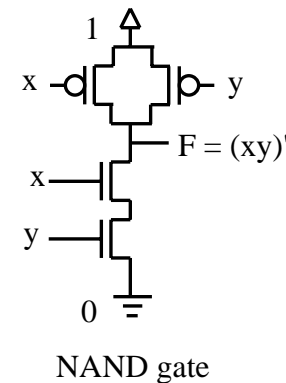
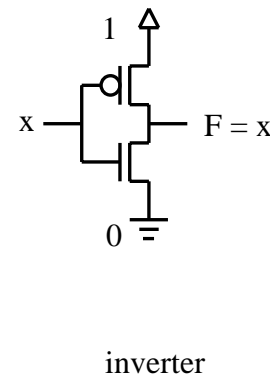
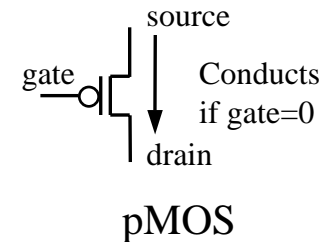
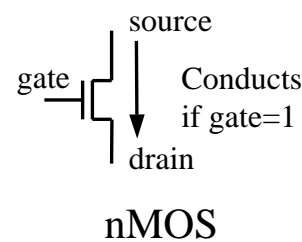
# CMOS transistor on silicon

- 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
  - Don’t confuse this “gate” with a logic gate

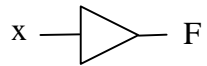


# CMOS transistor implementations

- Complementary Metal Oxide Semiconductor
- We 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

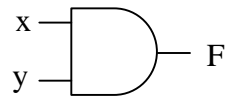


# Basic logic gates



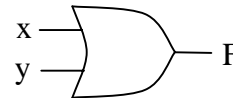
x	F
0	0
1	1

$F = x$   
Driver



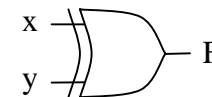
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = x y$   
AND



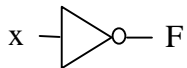
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = x + y$   
OR



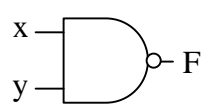
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

$F = x \oplus y$   
XOR



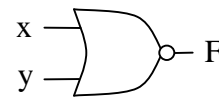
x	F
0	1
1	0

$F = x'$   
Inverter



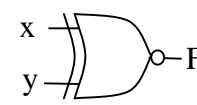
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

$F = (x y)'$   
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

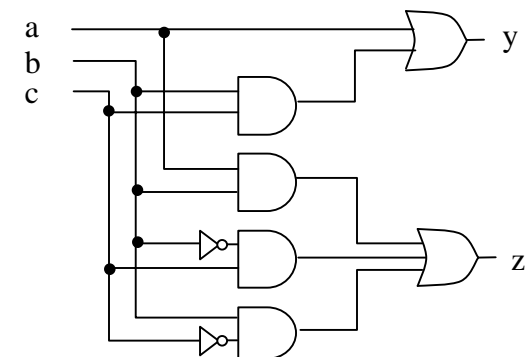
y	a	bc			
		00	01	11	10
0	0	0	0	1	0
1	1	1	1	1	1

$$y = a + bc$$

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

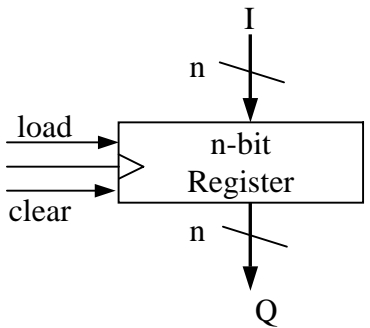
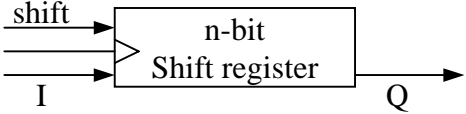
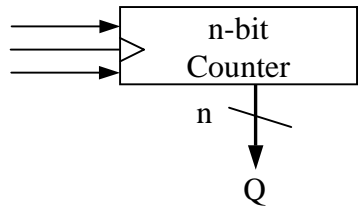


# Combinational components

<p> <math>O =</math>  <math>I_0</math> if <math>S=0..00</math>  <math>I_1</math> if <math>S=0..01</math>  <math>\dots</math>  <math>I_{(m-1)}</math> if <math>S=1..11</math> </p>	<p> <math>O_0 = 1</math> if <math>I=0..00</math>  <math>O_1 = 1</math> if <math>I=0..01</math>  <math>\dots</math>  <math>O_{(n-1)} = 1</math> if <math>I=1..11</math> </p>	<p> <math>sum = A + B</math>                      (first <math>n</math> bits)  <math>carry = (n+1)</math>'th                      bit of <math>A+B</math> </p>	<p> <math>less = 1</math> if <math>A &lt; B</math>  <math>equal = 1</math> if <math>A = B</math>  <math>greater = 1</math> if <math>A &gt; B</math> </p>	<p> <math>O = A \ op \ B</math>  <math>op</math> determined                      by <math>S</math>.                 </p>
	<p>With enable input <math>e \rightarrow</math> all <math>O</math>'s are 0 if <math>e=0</math></p>	<p>With carry-in input <math>C_i \rightarrow</math> <math>sum = A + B + C_i</math></p>		<p>May have status outputs carry, zero, etc.</p>



# Sequential components

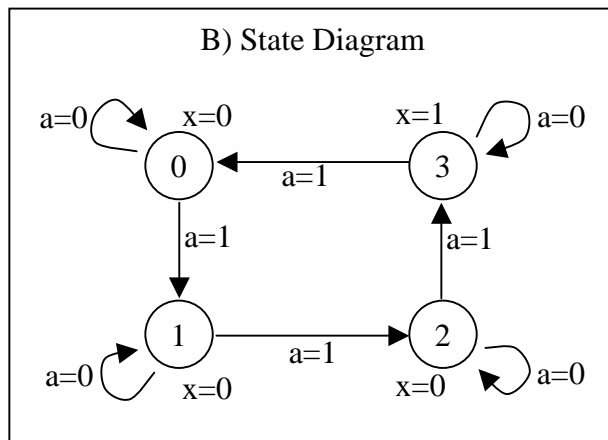
		
<p>Q =            0 if clear=1,            I if load=1 and clock=1,            Q(previous) otherwise.</p>	<p>Q = lsb            - Content shifted            - I stored in msb</p>	<p>Q =            0 if clear=1,            Q(prev)+1 if count=1 and clock=1.</p>

# Sequential logic design

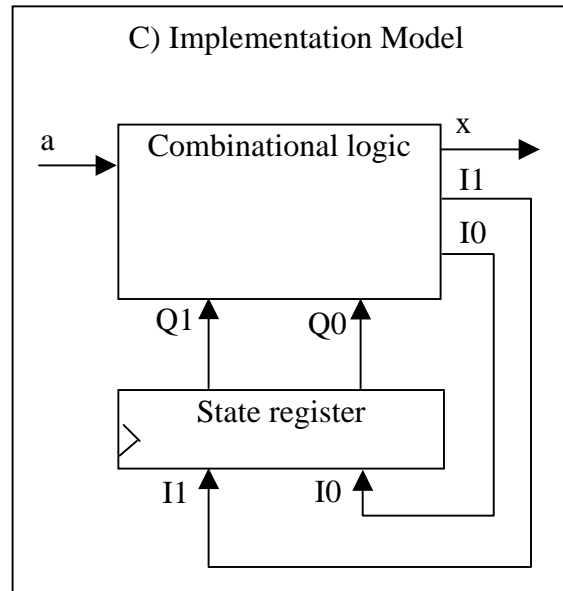
## A) Problem Description

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



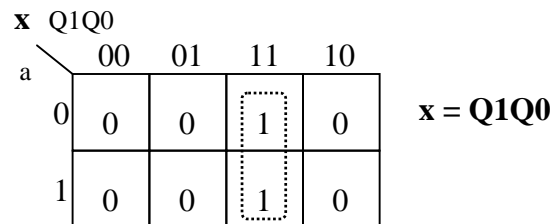
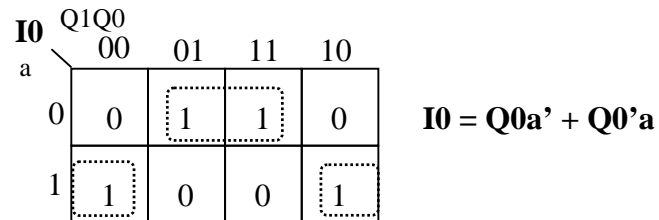
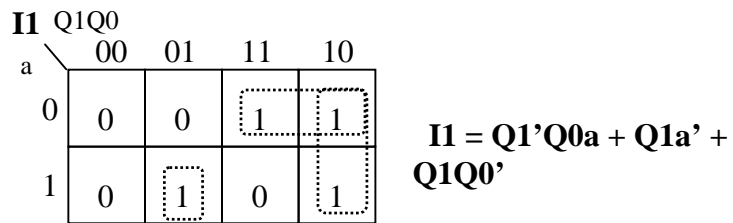
## D) State Table (Moore-type)

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

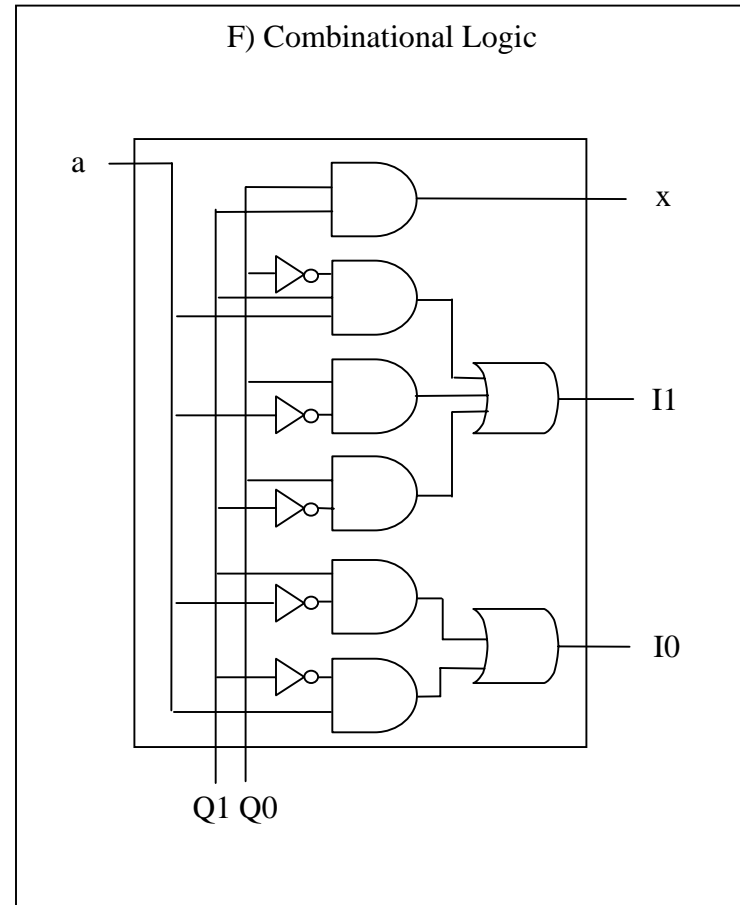
- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design

# Sequential logic design (cont.)

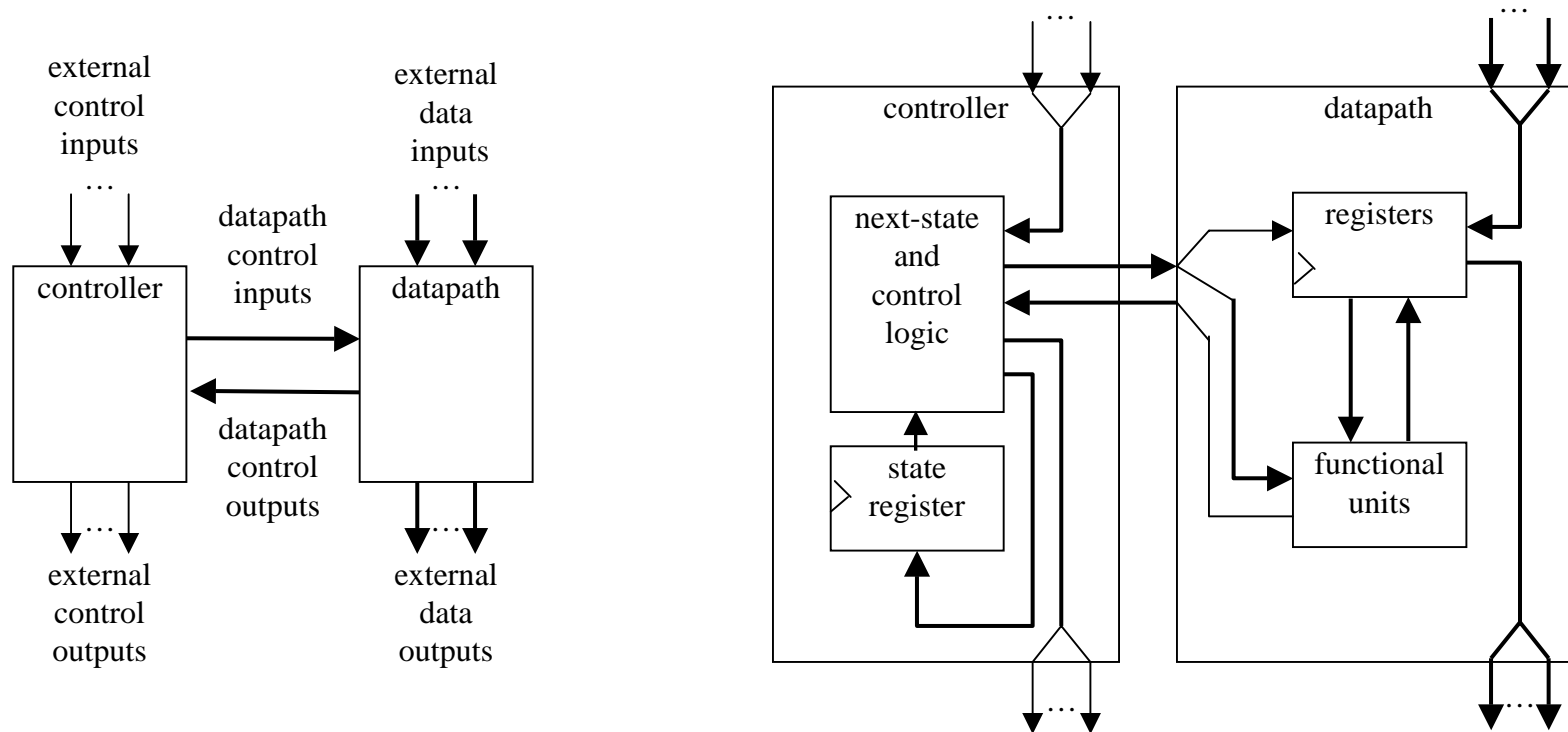
E) Minimized Output Equations



F) Combinational Logic



# Custom single-purpose processor basic model

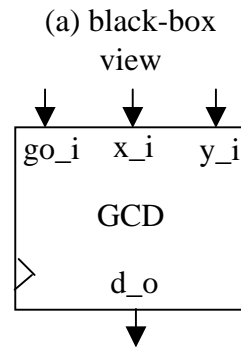


controller and datapath

a view inside the controller and datapath

# Example: greatest common divisor

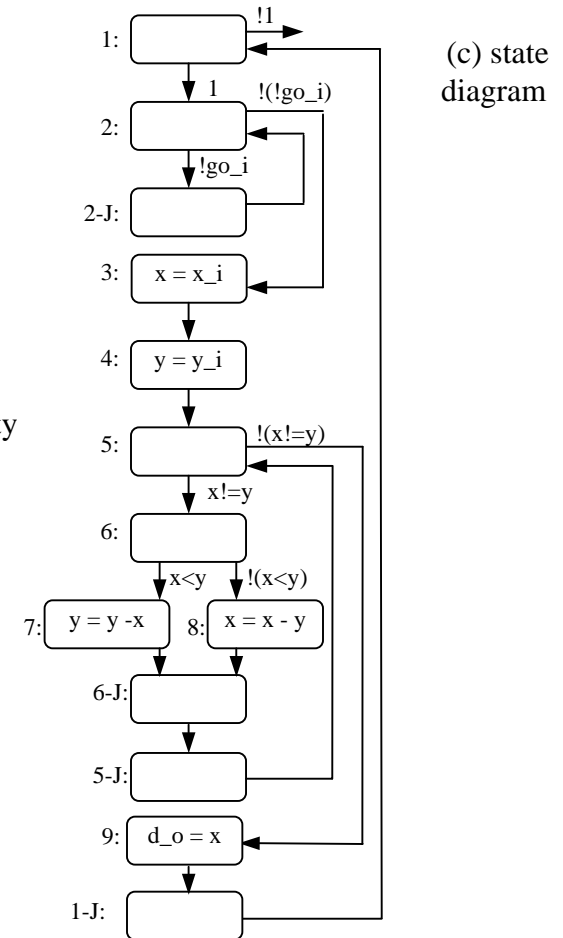
- First create algorithm
- Convert algorithm to “complex” state machine
  - Known as FSM: finite-state machine with datapath
  - Can use templates to perform such conversion



(b) desired functionality

```

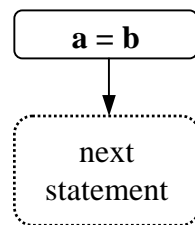
0: int x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
8:     else
9:       x = x - y;
9:   d_o = x;
}
    
```



# State diagram templates

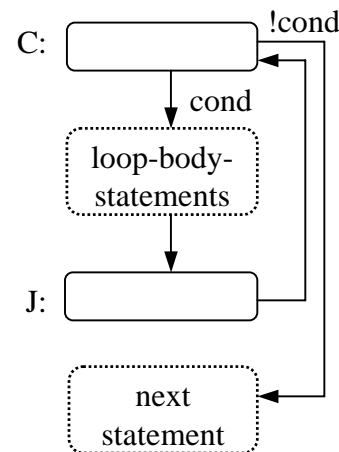
Assignment statement

```
a = b
next statement
```



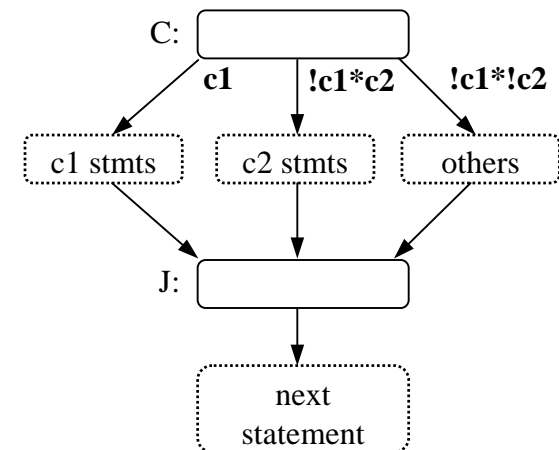
Loop statement

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



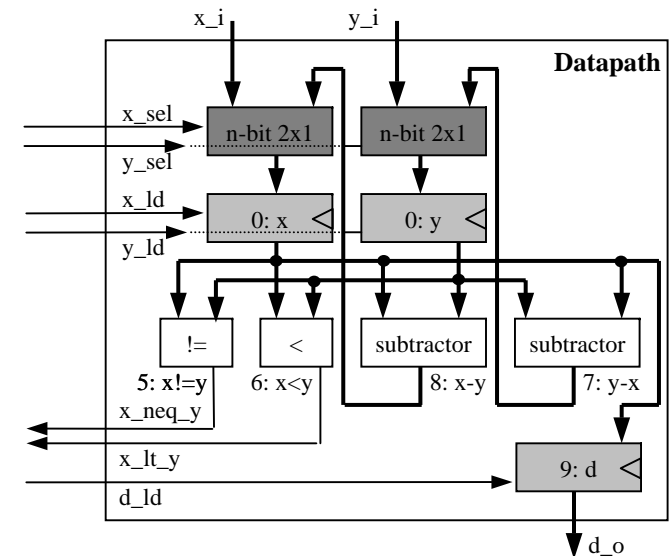
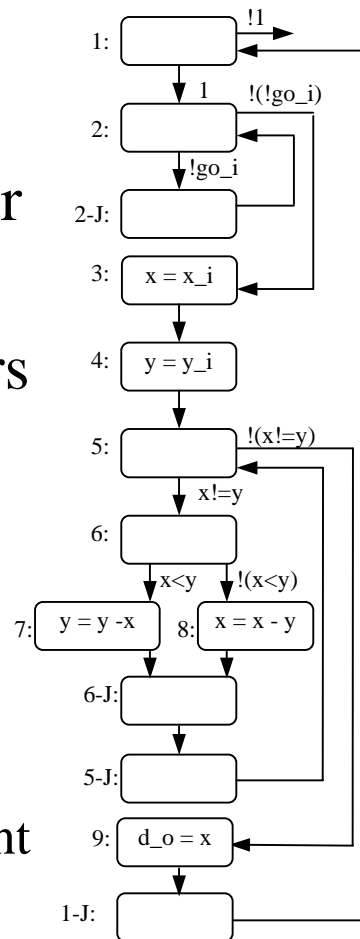
Branch statement

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

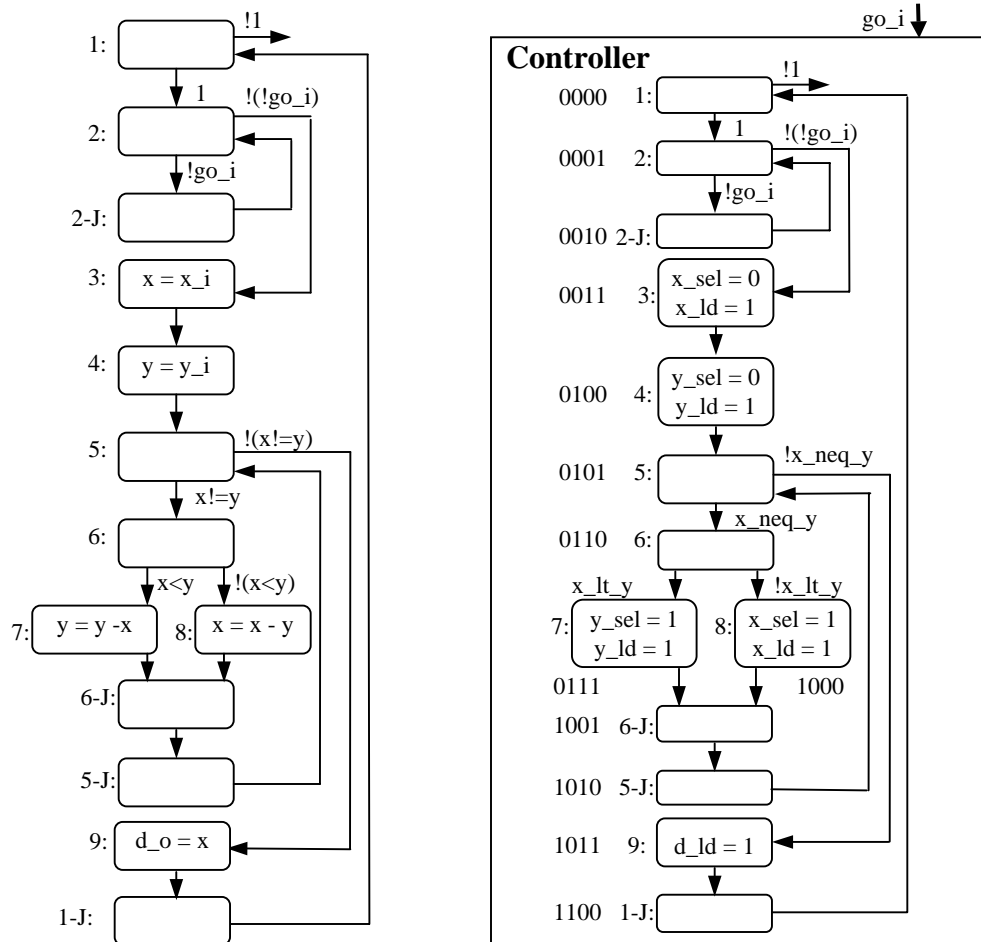


# Creating the datapath

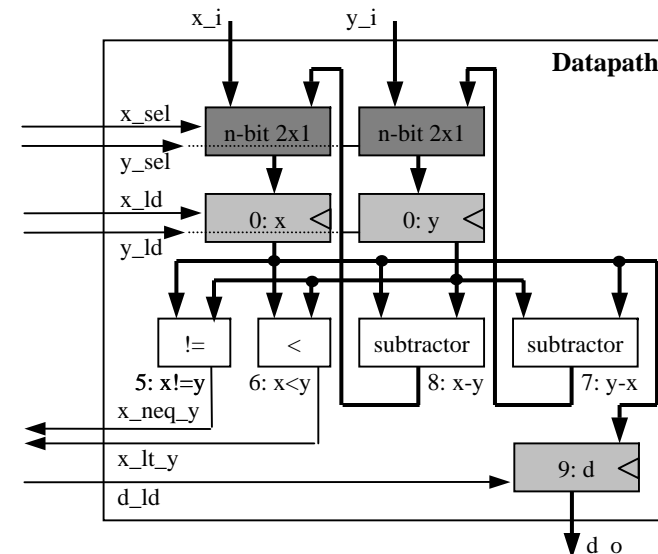
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output



# Creating the controller's FSM

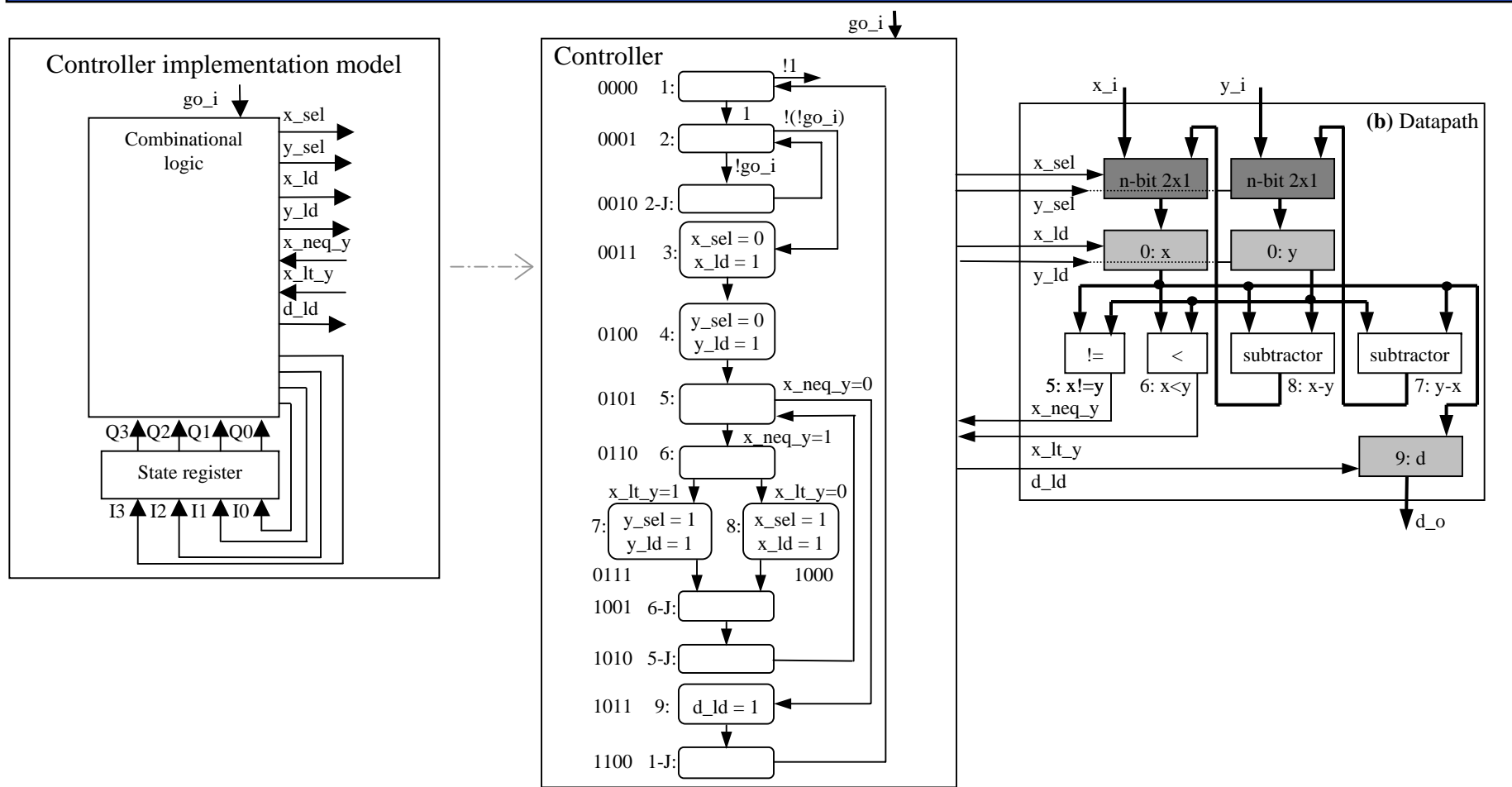


- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations





# Splitting into a controller and datapath

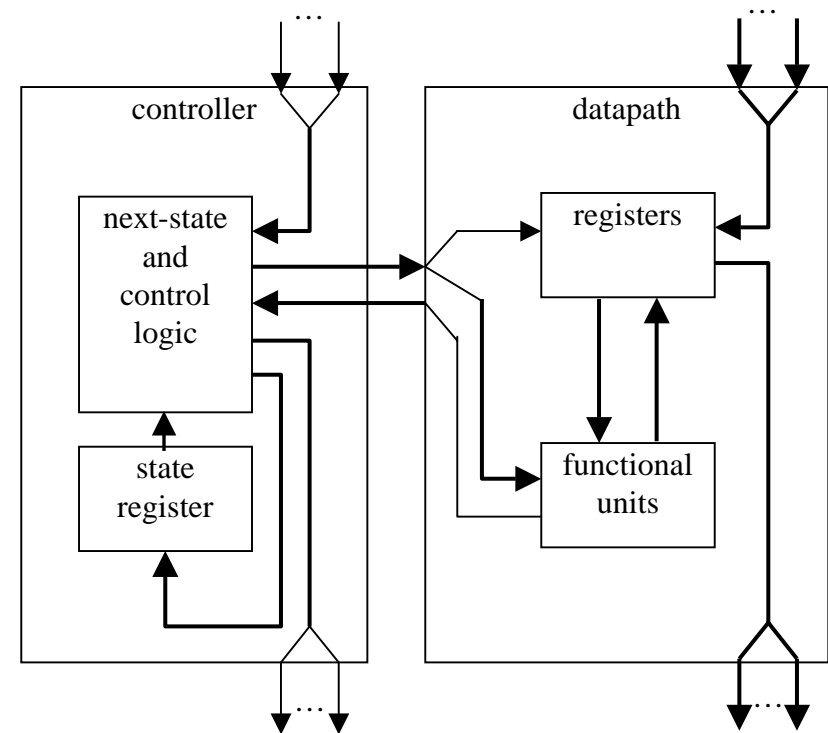


# Controller state table for the GCD example

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

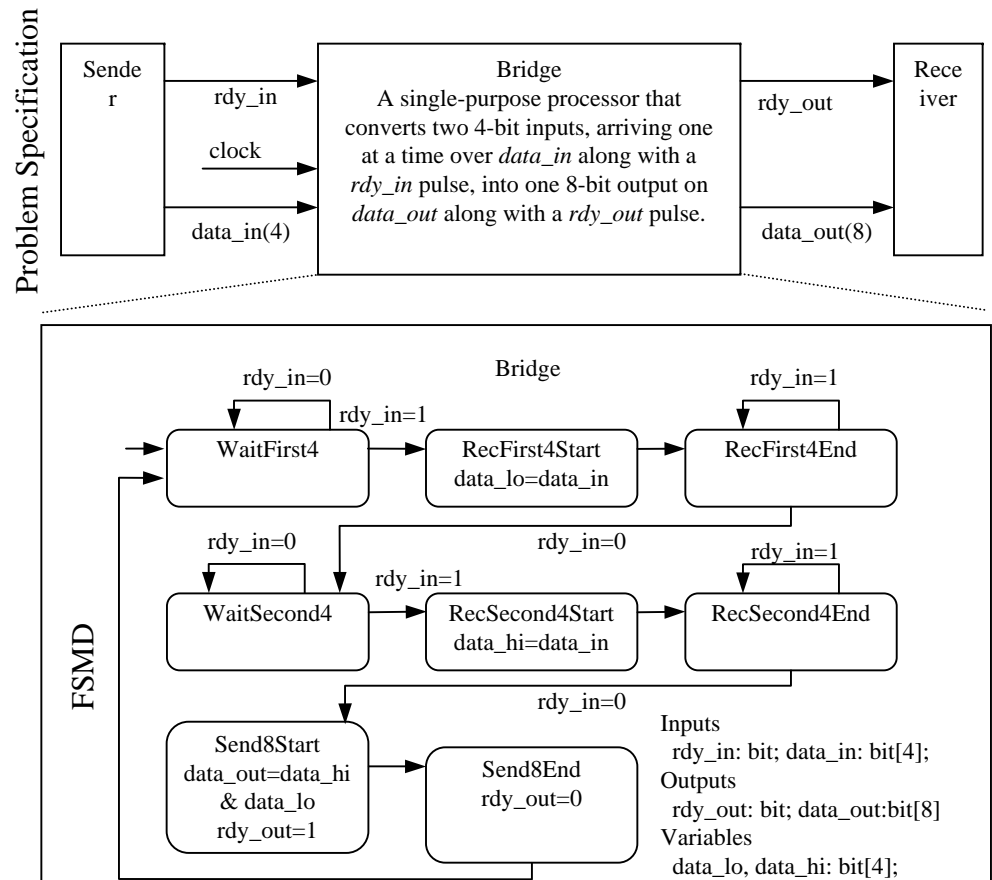
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps



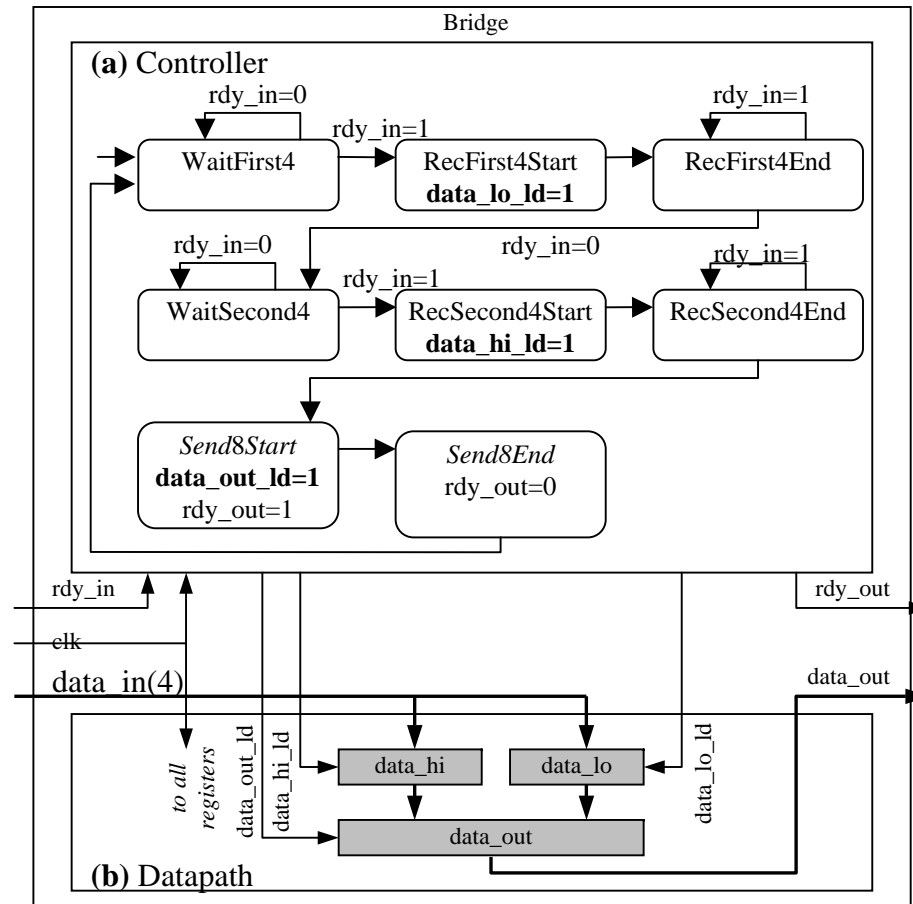
a view inside the controller and datapath

# 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



# 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), (43, 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)

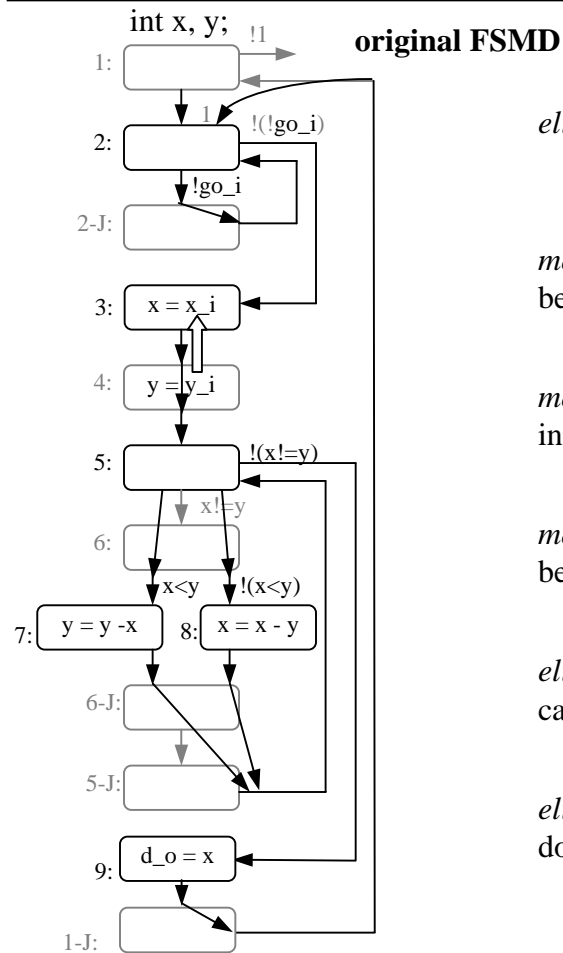


# Optimizing the FSMD

---

- 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 FSM (cont.)



*eliminate state 1* – transitions have constant values

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

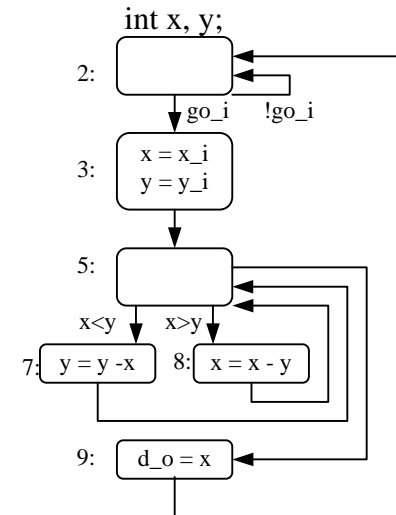
*merge state 3 and state 4* – assignment operations are independent of one another

*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

**optimized FSM**



# 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

# Summary

---

- Custom single-purpose processors
  - Straightforward design techniques
  - Can be built to execute algorithms
  - Typically start with FSMD
  - CAD tools can be of great assistance