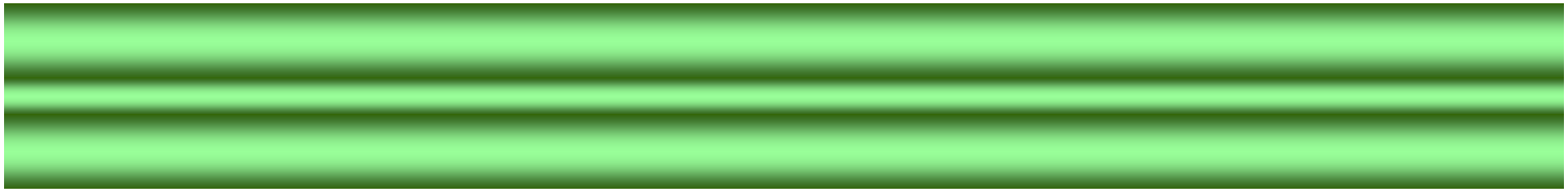
EE414 Embedded Systems

Ch 2. Custom Single-Purpose Processors: Hardware



Byung Kook Kim
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Outline

- 2.1 Introduction

Review Digital Logic

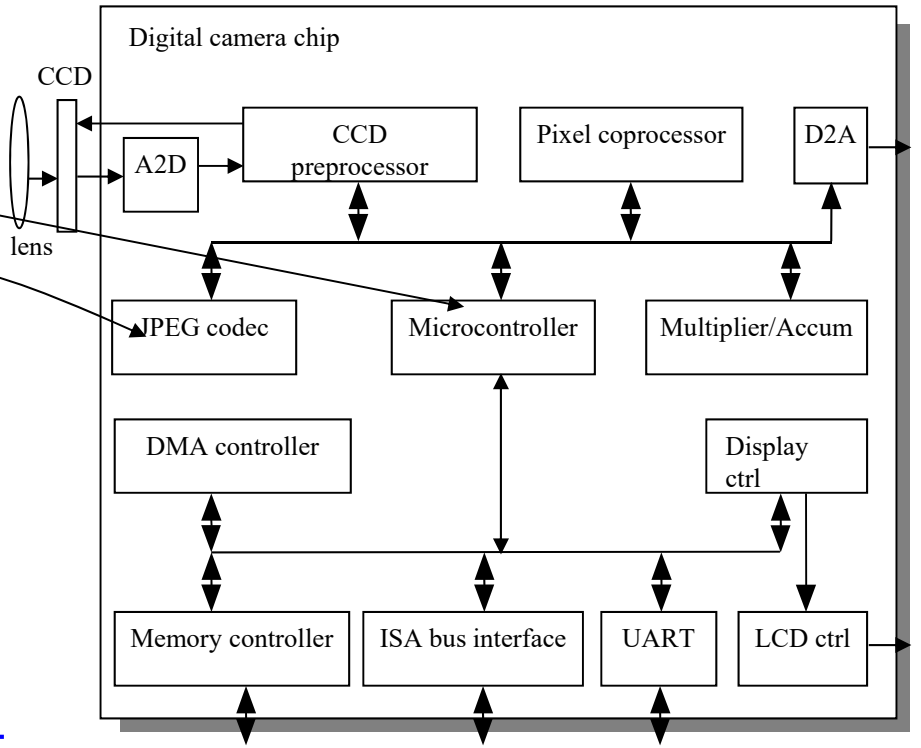
- 2.2 Combinational logic
- 2.3 Sequential logic

Custom Single-Purpose Processor

- 2.4 Custom single-purpose processor design
- 2.5 RT-level custom single-purpose processor design
- 2.6 Optimizing custom single-purpose processors

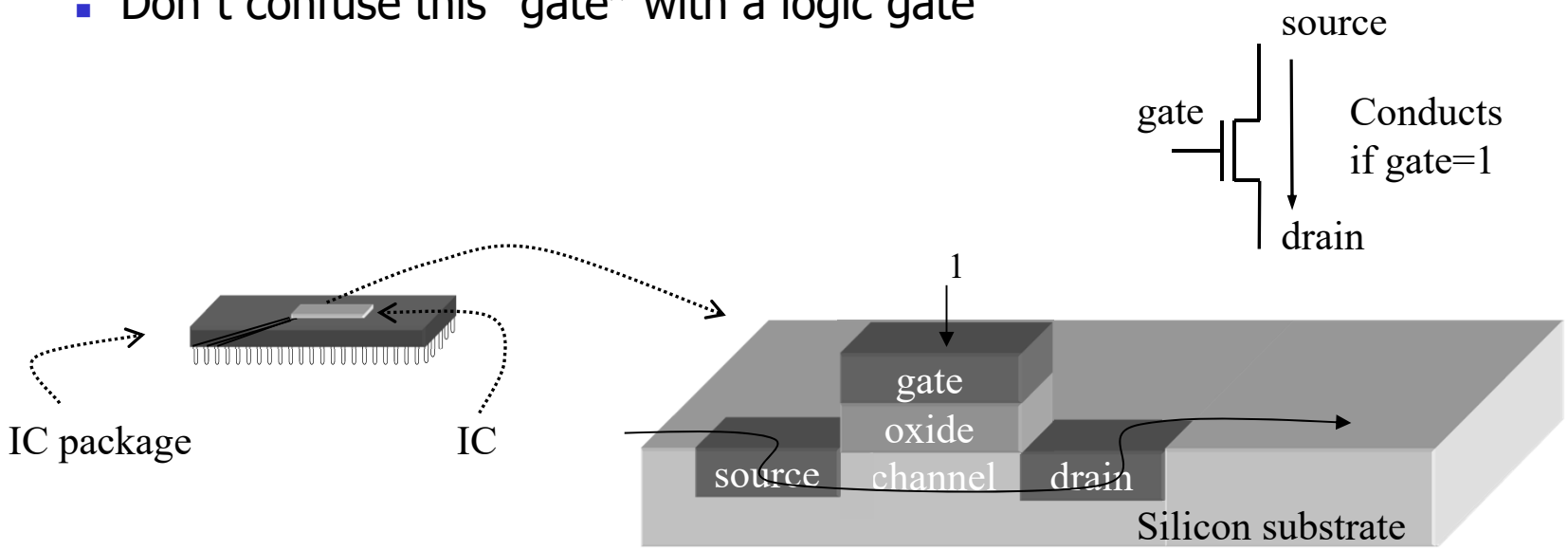
2.1 Introduction

- **Processor**
  - Digital circuit that performs computation tasks
  - Controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Custom single-purpose processor: non-standard task
- **Custom single-purpose processor**
  - Faster, smaller, low power
  - But, high NRE cost, longer time-to-market, less flexible.



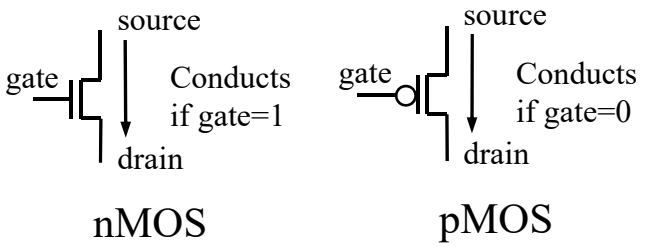
2.2 Combinational Logic
Transistors and Logic Gates

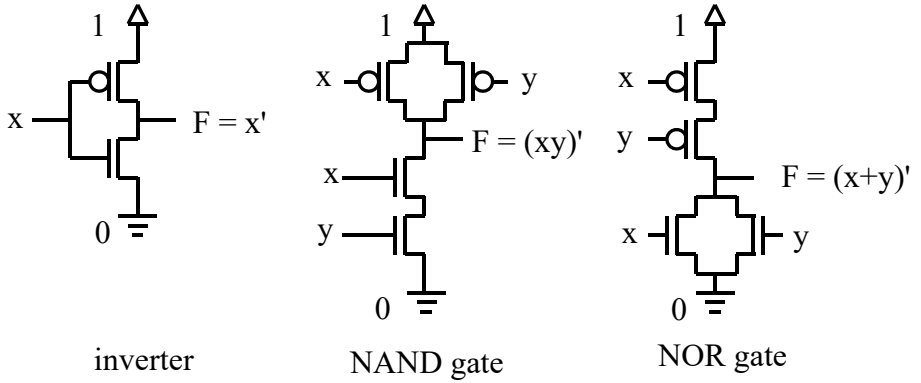
- **Transistor**
  - The basic electrical component in digital systems.
  - Acts as an on/off switch (not as an amplifier).
  - Voltage at "gate" controls whether current flows from source to drain
    - Don't confuse this "gate" with a logic gate



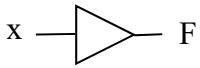
CMOS transistor – Logic gate implementations

- **CMOS: Complementary Metal Oxide Semiconductor**
  - Most widely used. Transistor < Logic gates < Digital systems.
- We refer to logic levels
  - Typically 0 is 0V, 1 is 5V
- Two basic CMOS TR types
  - nMOS conducts if gate=1 →
  - pMOS conducts if gate=0 →
  - Hence "complementary"
- **Basic logic gates**
  - Inverter, NAND, NOR →



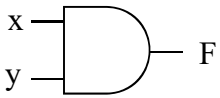


Basic Logic Gates



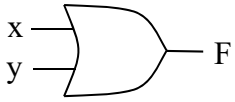
x	F
0	0
1	1

F = x
Driver



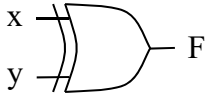
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

F = x y
AND



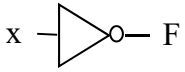
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

F = x + y
OR



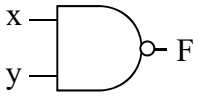
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

F = x ⊕ y
XOR



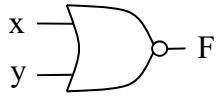
x	F
0	1
1	0

F = x'
Inverter



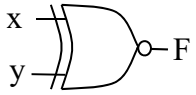
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

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 ⊙ y
XNOR

- Digital system designers usually work at the abstraction level of logic gates rather than transistors.
- Transistor < Logic gates < Digital systems.

Basic Combinational Logic Design

Combinational circuit: Output purely function of present input (No memory)

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

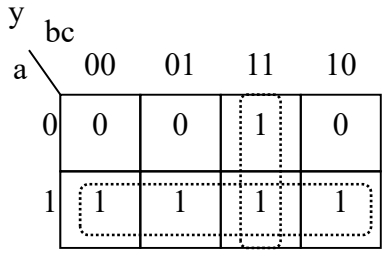
Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

C) Output equations

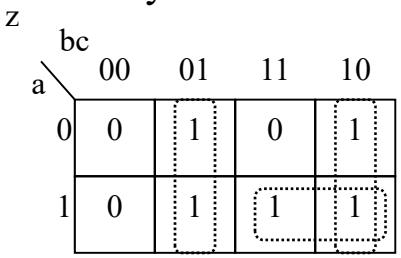
y = a'bc + ab'c' + ab'c + abc' + abc

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

D) Minimized output equations



y = a + bc

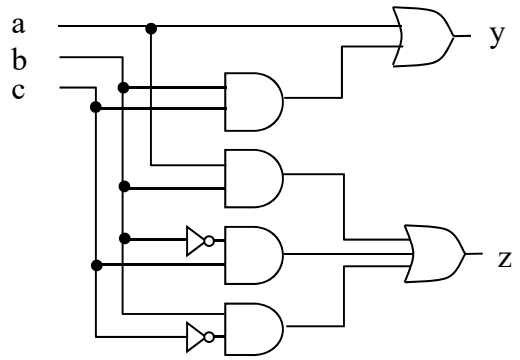


z = ab + b'c + bc'

Karnaugh maps:

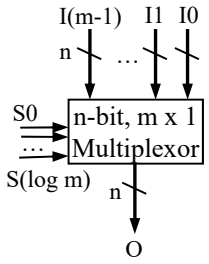
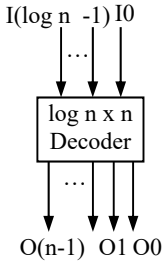
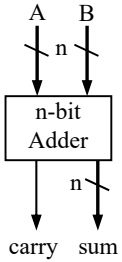
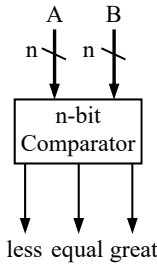
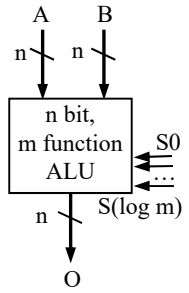
Maximize squares enclosing 1's.

E) Logic Gates



RT-level combinational components

- Multiplexer, Decoder, Adder, Comparator, and ALU

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

2.3 Sequential Logic

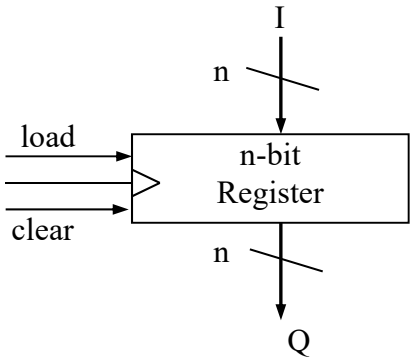
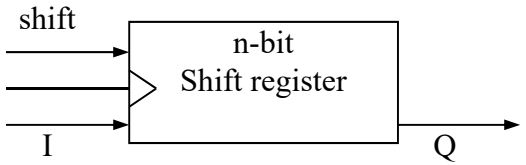
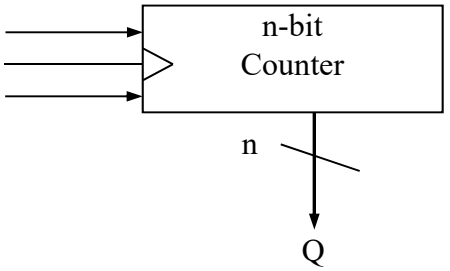
Flip-Flops

- **Sequential circuit**
  - A digital circuit whose outputs are function of the present as well as previous input values
  - Sequential logic possesses memory: Combinational logic and memory.
  - Ex. Coffee vending machine

- **Flip-flop**
  - Basic memory in sequential circuit
  - D flip-flop
  - SR flip-flop
  - JK flip-flop
  - Level triggered
  - Edge triggered

RT-level sequential components

- Register, shift register, and counter

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

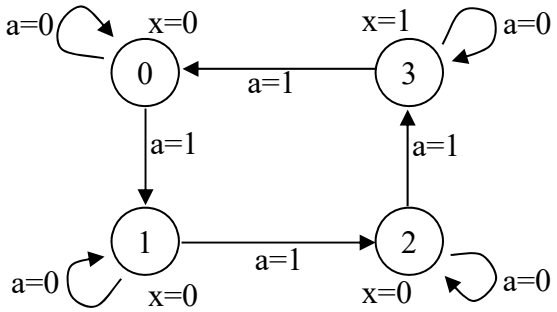
- Synchronous: Input value has an affect during clock edge.
- Asynchronous: Input value affects the circuit independent of clock.

Sequential Logic Design

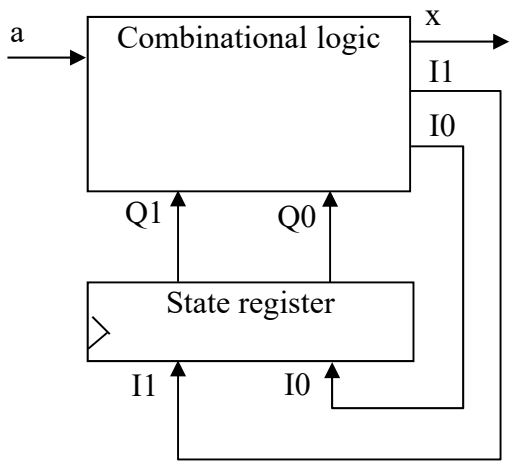
A) Problem Description

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	

- State diagram
  - Finite State Machine (FSM)
- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design.

Sequential logic design (cont.)

E) Minimized Output Equations

I1

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

I1 = Q1'Q0a + Q1a' + Q1Q0'

I0

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

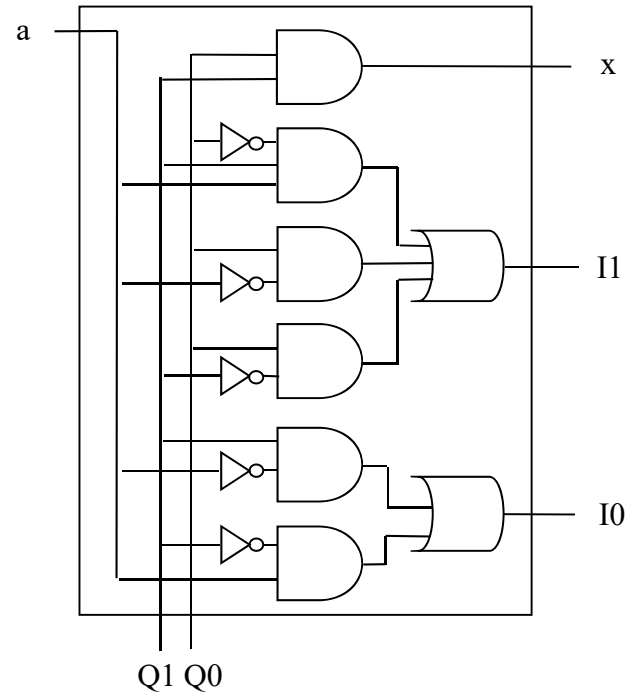
I0 = Q0a' + Q0'a

x

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

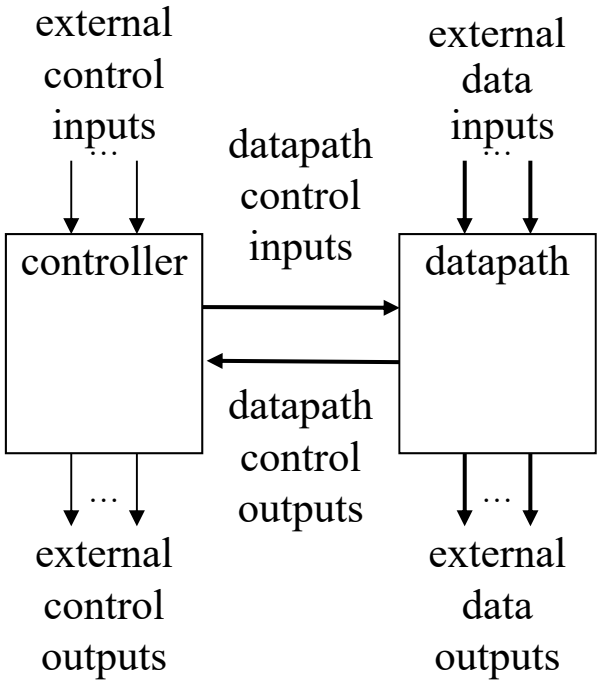
x = Q1Q0

F) Combinational Logic



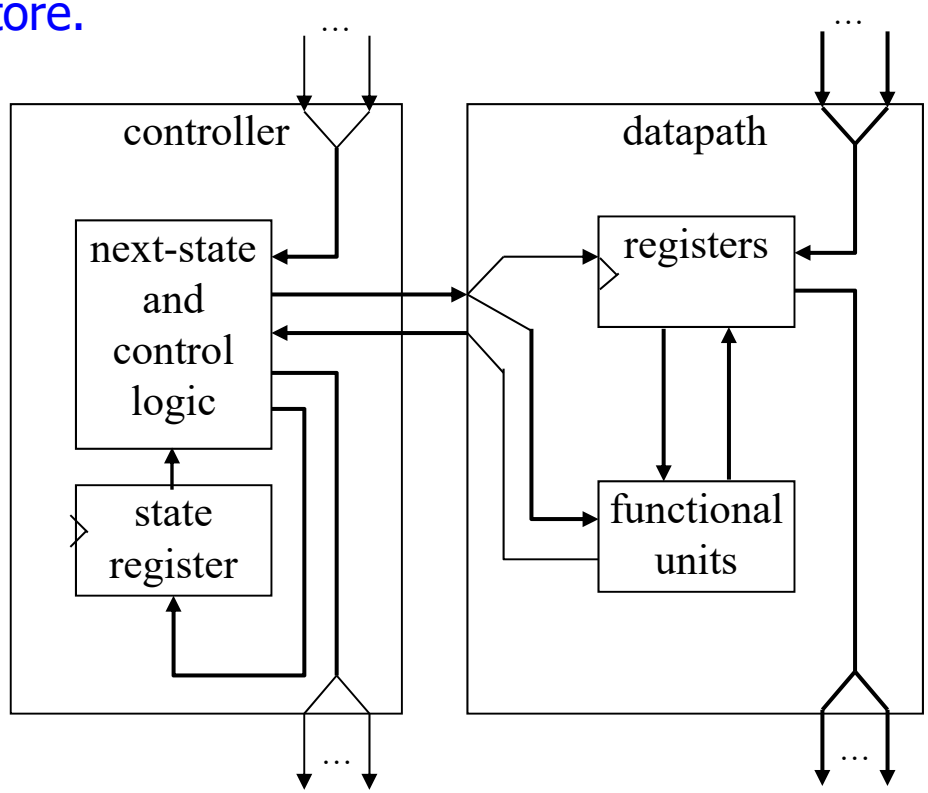
2.4 Custom Single-Purpose Processor Design

- A basic processor model
  - Controller: Control datapath.
  - Datapath: Manipulate and store.



Controller and datapath

- Inside view

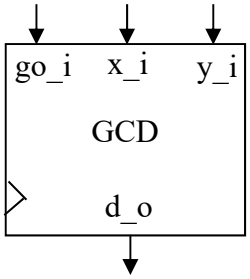


A view inside the controller and datapath

Design example: Greatest Common Divisor

- 1. First create algorithm
  - Subtraction (b)
- 2. Convert algorithm to "complex" state machine (c)
  - Known as FSMD: finite-state machine with data
  - Can use templates to perform such conversion

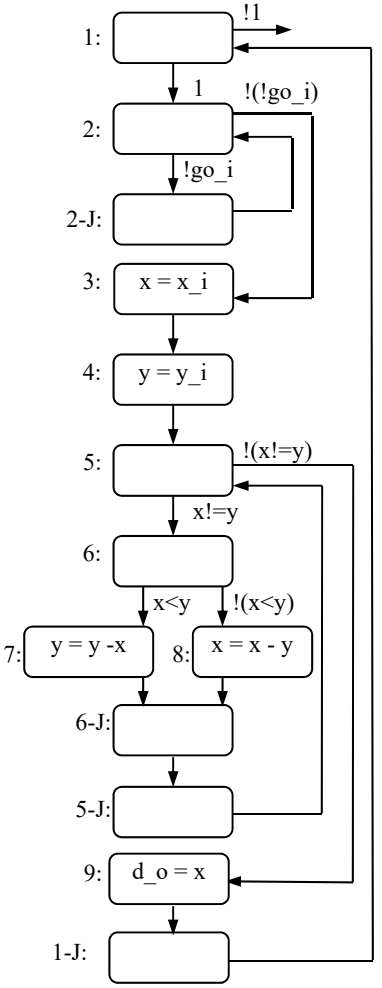
(a) black-box view



(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;
       else
8:         x = x - y;
     }
9:   d_o = x;
   }
```

(c) state diagram



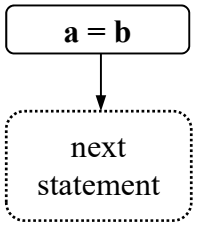
2. State diagram templates

Templates

- (a) Assignment

Assignment statement

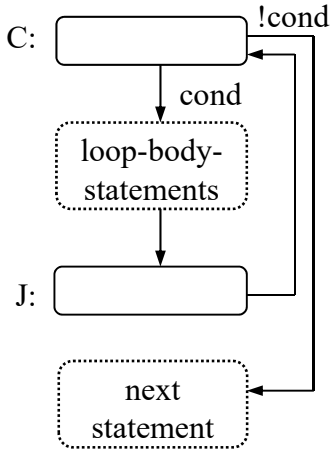
a = b
next statement



(b) Loop

Loop statement

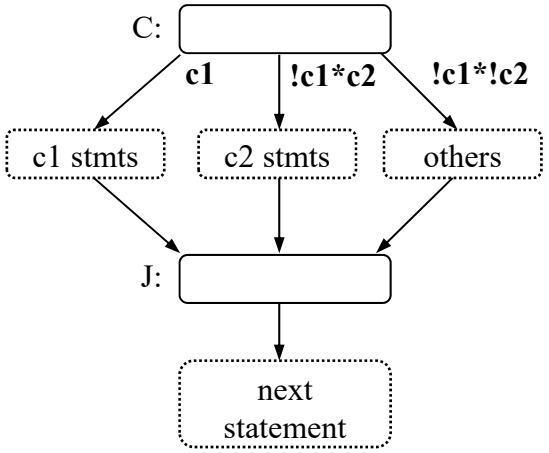
while (cond) {
loop-body-statements
}
next statement



(c) Branch

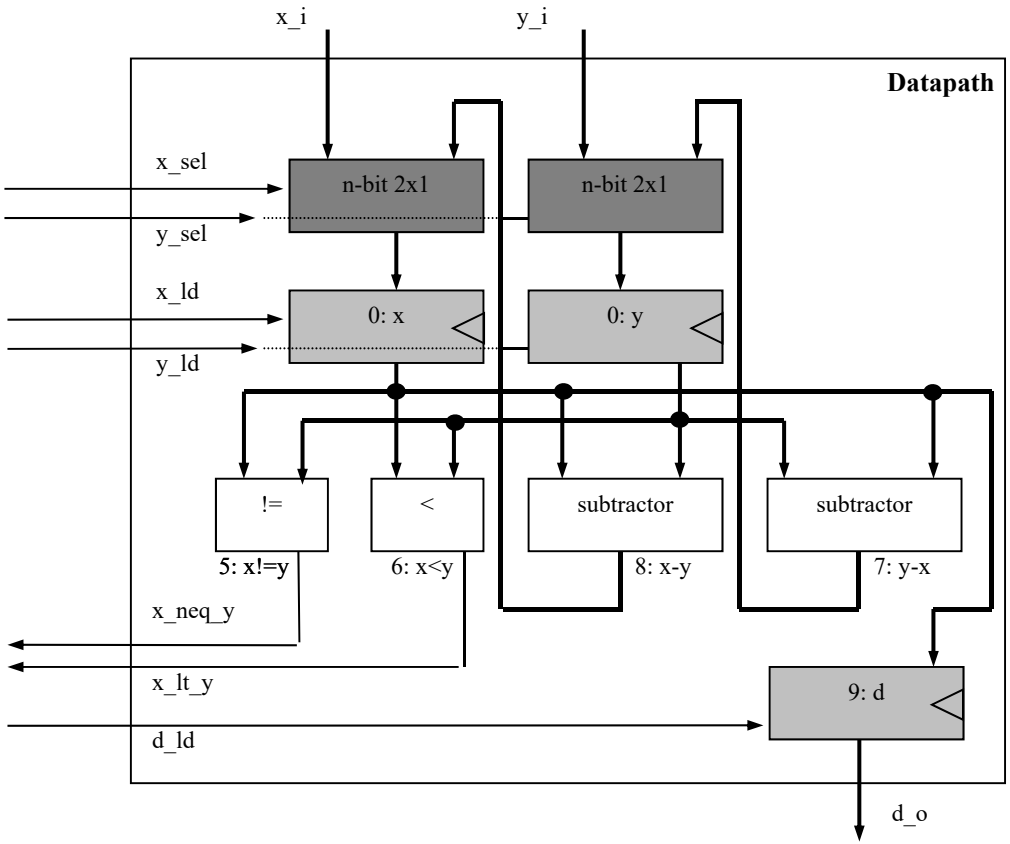
Branch statement

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

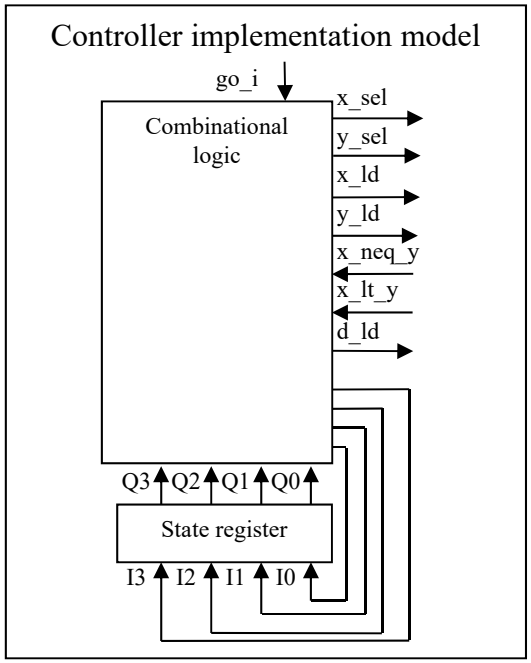


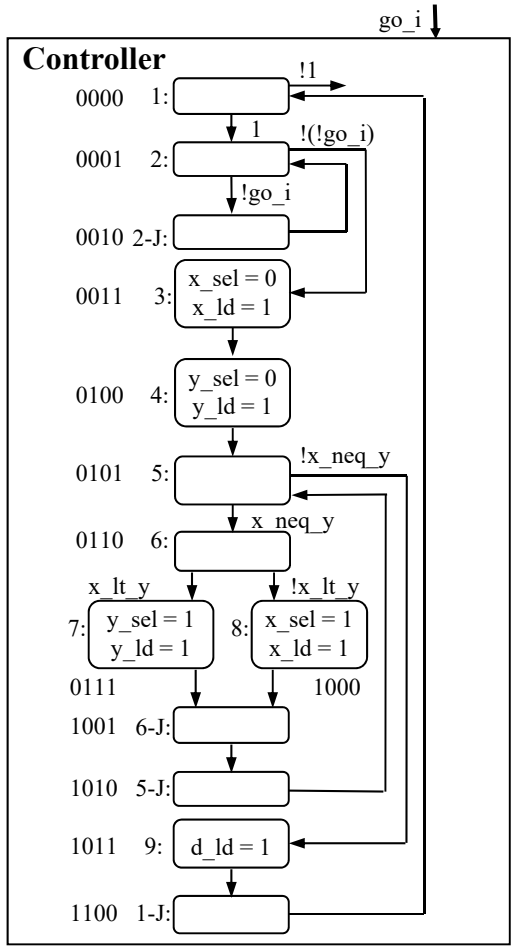
3. Creating the datapath

- 1. Create a register & ports for any declared variable
- 2. Create a functional unit for each arithmetic operation
- 3. Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexers for multiple sources
- 4. Create unique identifier
  - for each datapath component control input and output

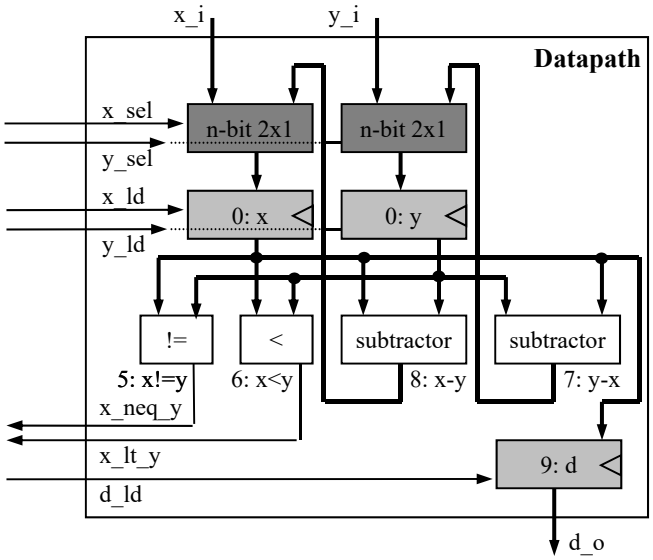


4. Creating the controller's FSM





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

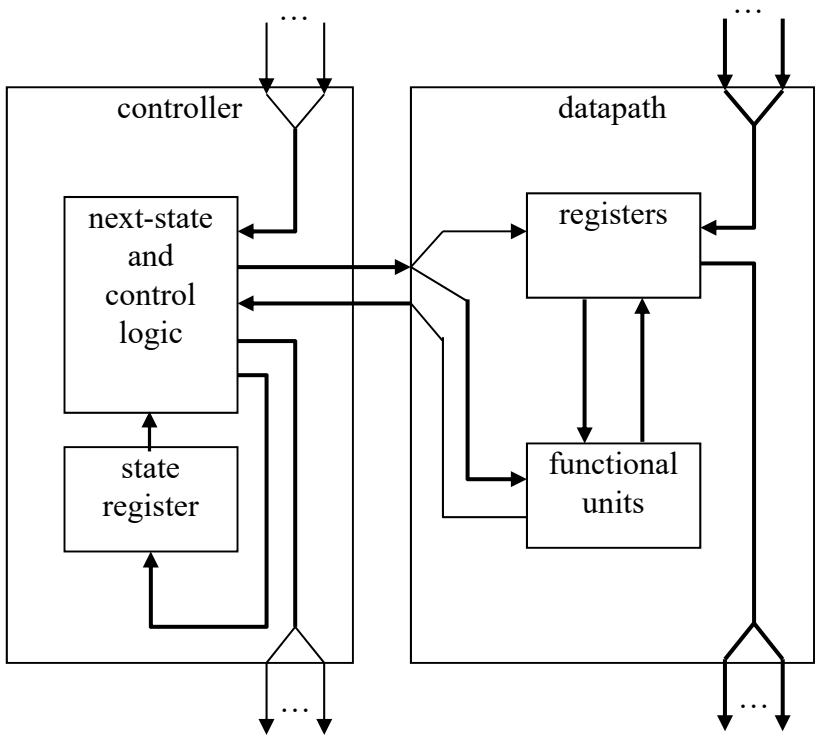


5. Controller state table for the GCD example

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

6. Completing the GCD custom single-purpose processor design

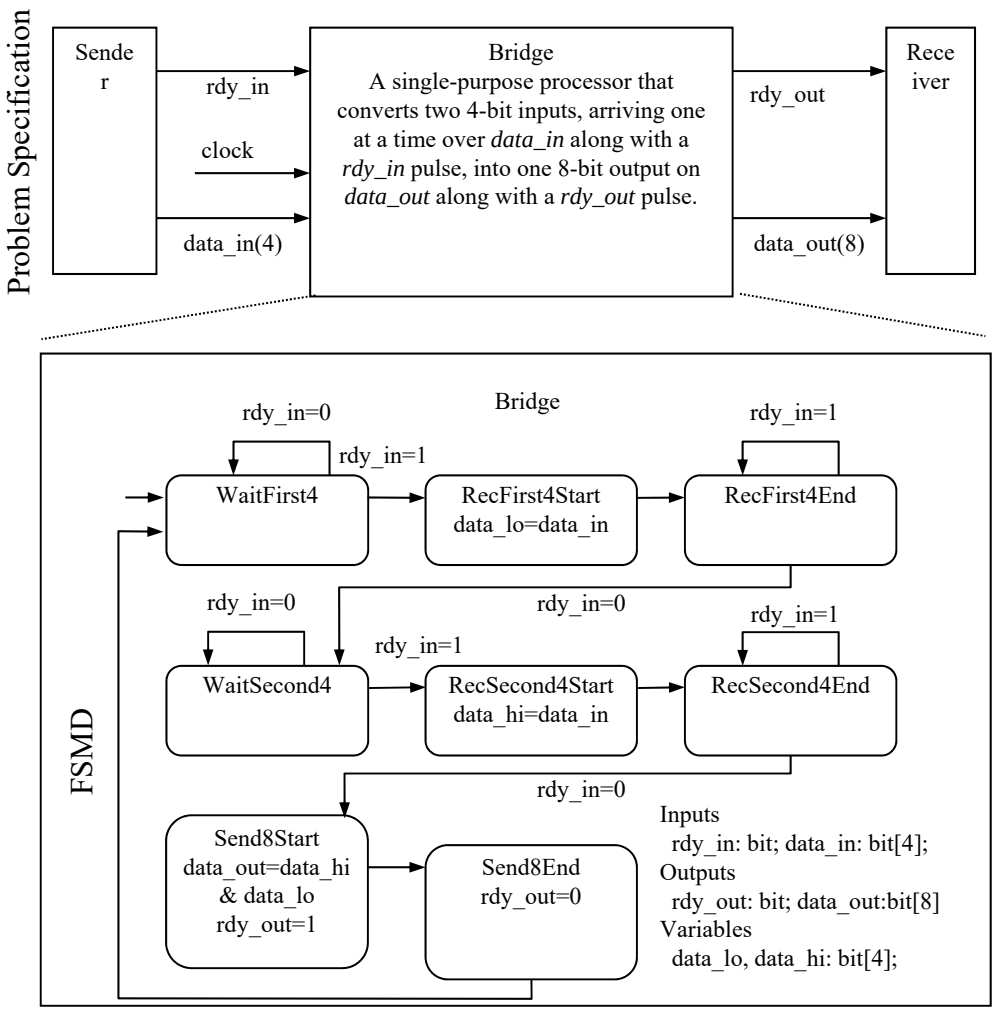
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps



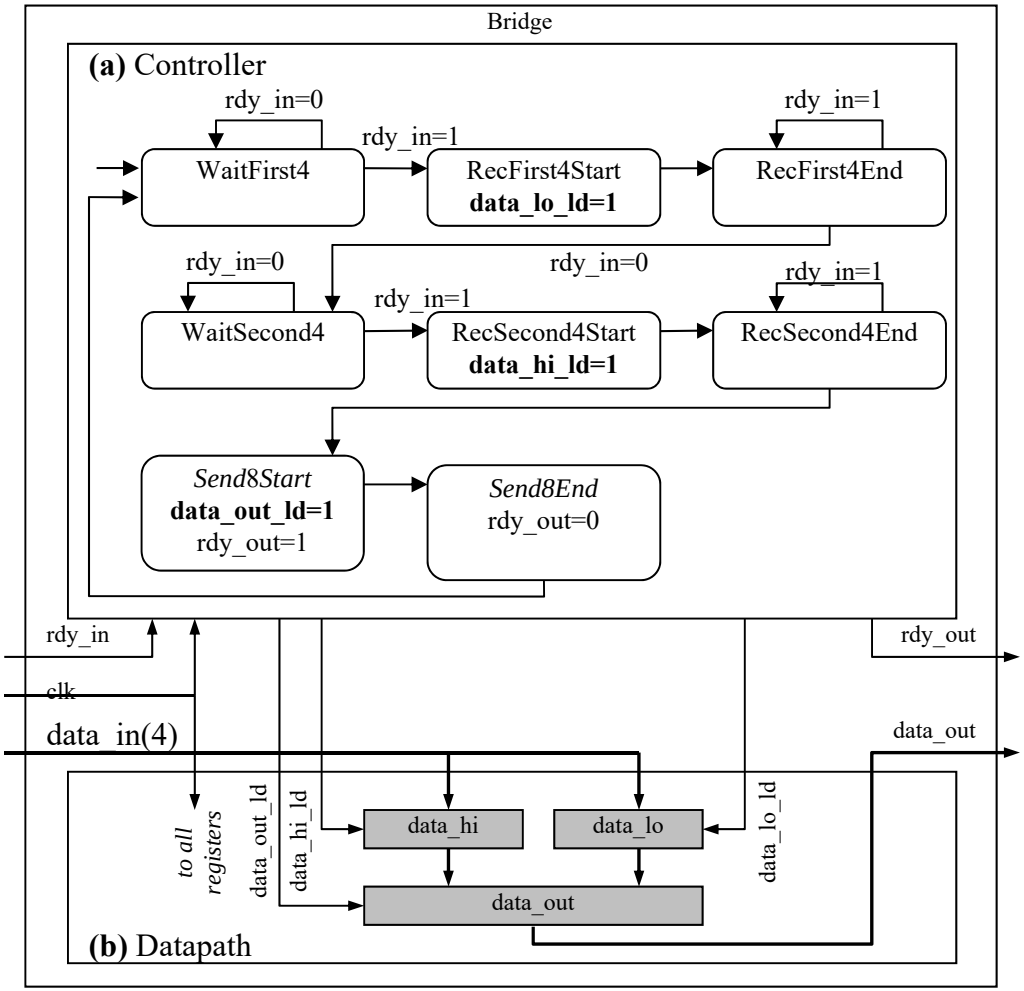
a view inside the controller and datapath

2.5 RT-level Custom Single-Purpose Processor Design

- We often start with a state machine, rather than program.
  - Especially when explicit cycle timing is important.
- 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')



2.6 Optimizing Custom Single-Purpose Processors

- **Optimization** is the task of making design metric values the best possible.

- Optimization opportunities
  - A. Original program / algorithm
  - B. FSMD
  - C. Datapath
  - D. FSM

A. Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - Time and space complexity
    - Number of computations
    - Size of variable
  - Operations used
    - Multiplication and division are very expensive.

Optimizing the original program: Using Modulo operation

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;
       else
8:         x = x - y;
     }
9:   d_o = x;
   }
```

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);
      // x must be the larger
number
 3:   if (x_i >= y_i)  {
 4:     x=x_i;
 5:     y=y_i;
      }
 6:   else {
 7:     x=y_i;
 8:     y=x_i;
      }
 9:   while  (y != 0)  {
10:     r = x % y;
11:     x = y;
12:     y = r;
      }
13:   d_o = x;
    }
```

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).

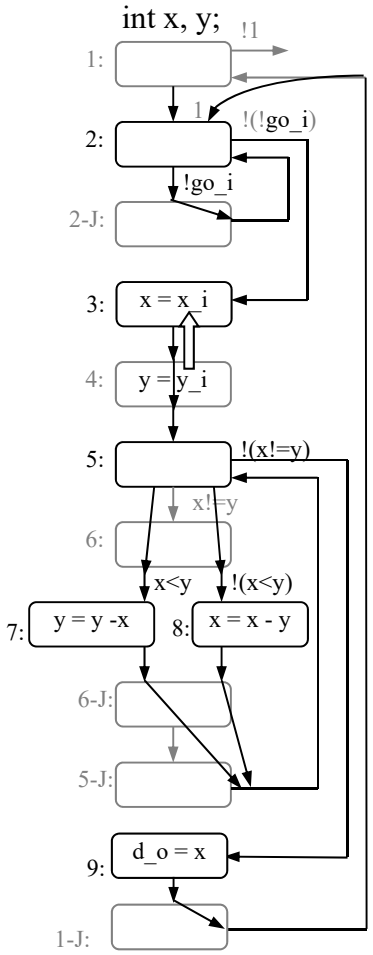
GCD(42,8) - 3 iterations to complete the loop

x and y values evaluated as follows: (42, 8), (8,2), (2,0)

B. Optimizing the FSMD

- Areas of possible improvements
  - Scheduling
    - Task of assigning operations from the original program to states in an FSMD.
  - 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

Optimizing the FSMD (cont.)



Original FSMD

eliminate state 1 – transitions have constant values

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

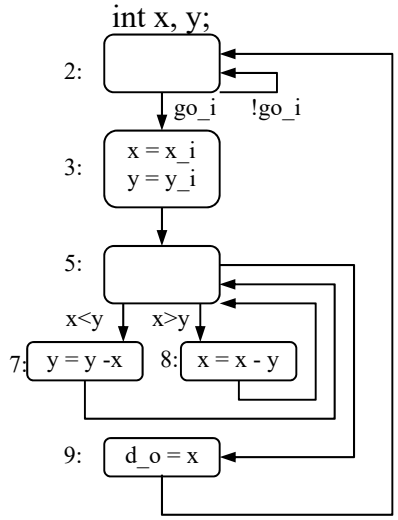
merge state 3 and state 4 – assignment operations are independent of one another

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 FSMD



C. 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
- Allocation: Choose RT components to use
- Bind: Map operations from FSMD to components.

D. 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

- For non-standard tasks

Optimized Design

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

References

- [1] Frank Vahid, “Embedded system design: A unified hardware/software introduction”, John Wiley & Sons, 2002.