VietNam National University
University of Engineering and Technology

THIẾT KẾ MẠCH TÍCH HỢP SỐ (DIGITAL IC DESIGN)

TS. Nguyễn Kiêm Hùng
Email: kiemhung@vnu.edu.vn

Laboratory for Smart Integrated Systems

10/12/2022

Lecture 3.1: RTL Design Techniques

Objectives

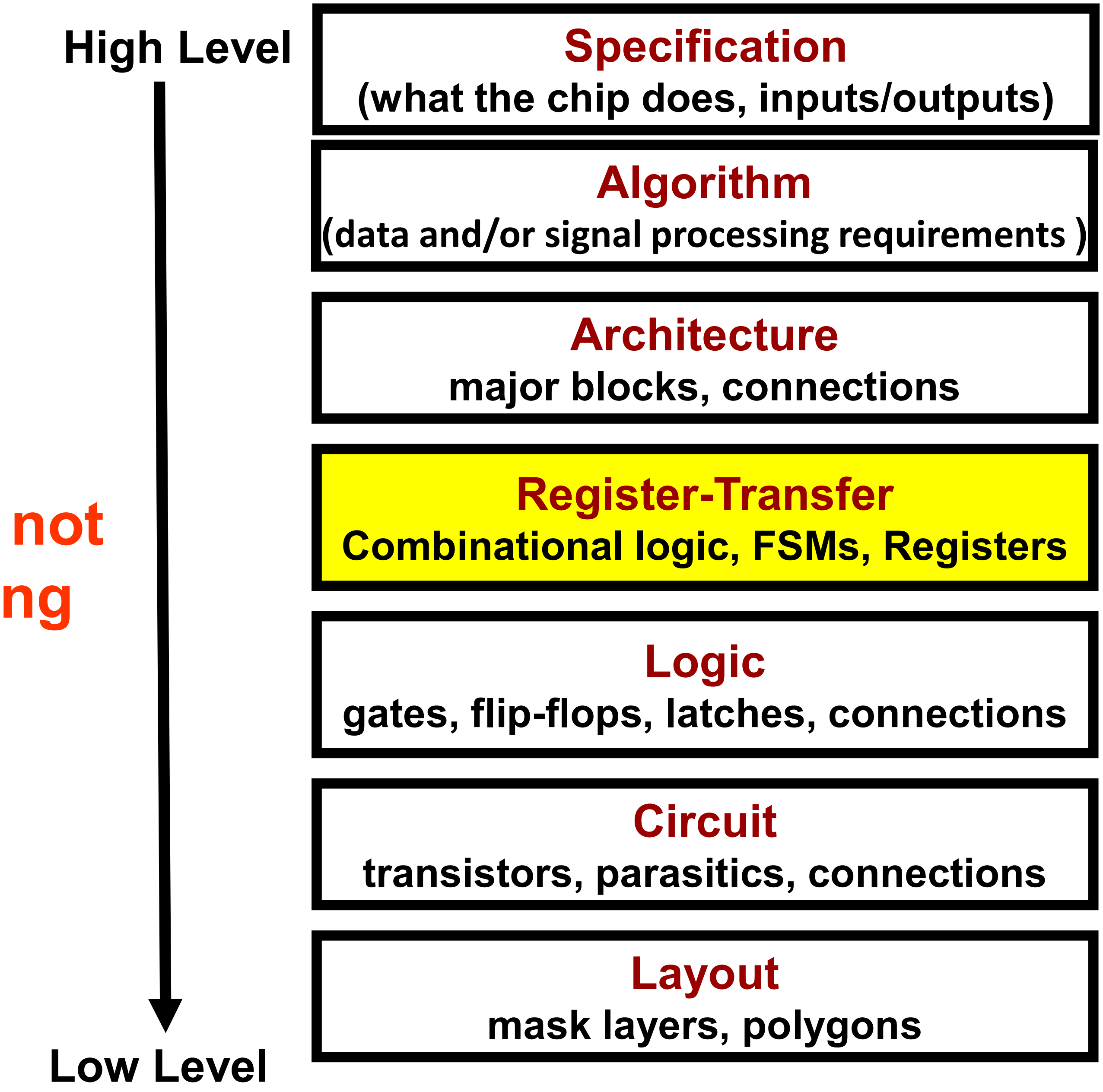
In this lecture you will be introduced to:

- **General-purpose IC vs. Special-purpose IC**
- **Method and techniques for designing General-purpose IC vs. Special-purpose IC at RTL**
  - How to convert an algorithm into a single-purpose architecture;
  - How to reach a optimized design

Outline

- **RTL design methodology**
- Review of Combinational and Sequential logic design
- Single-purpose Vs. general-purpose architectures
- RT-level custom single-purpose circuit design
- RT-level custom general-purpose processor design

Remind: Levels of Abstraction



High Level

Specification
(what the chip does, inputs/outputs)

Algorithm
(data and/or signal processing requirements)

Architecture
major blocks, connections

Register-Transfer
Combinational logic, FSMs, Registers

Logic
gates, flip-flops, latches, connections

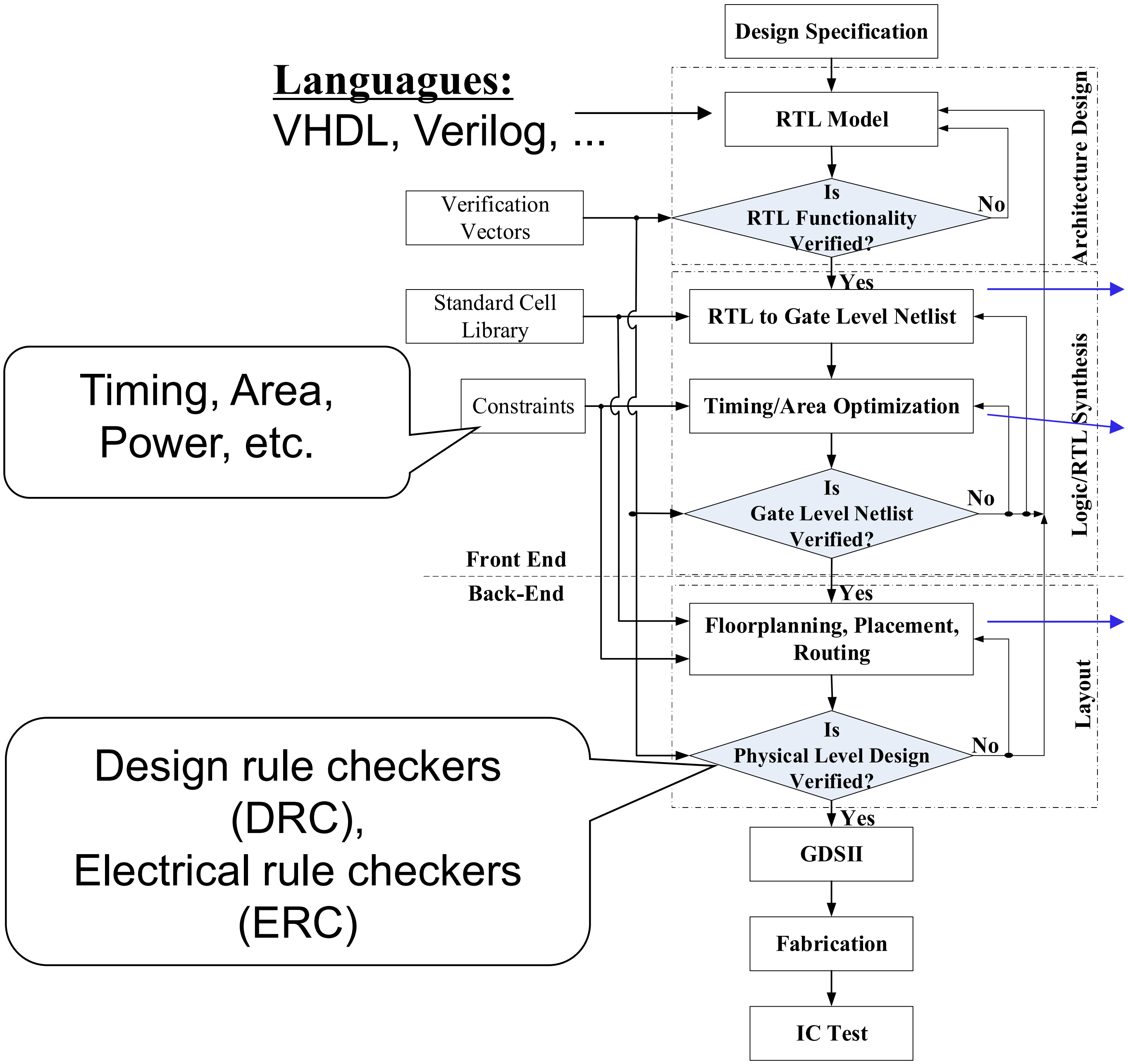
Circuit
transistors, parasitics, connections

Layout
mask layers, polygons

Low Level

Idea is not changing

Remind: IC Design Flow



Languages:
VHDL, Verilog, ...

Design Specification

RTL Model

Is RTL Functionality Verified?

No

Yes

Verification Vectors

Architecture Design

Standard Cell Library

RTL to Gate Level Netlist

Constraints

Timing/Area Optimization

Is Gate Level Netlist Verified?

No

Yes

Logic/RTL Synthesis

Front End

Back-End

Floorplanning, Placement, Routing

Is Physical Level Design Verified?

No

Yes

Layout

GDSII

Fabrication

IC Test

Timing, Area, Power, etc.

Design rule checkers (DRC),
Electrical rule checkers (ERC)

Tools:
Design Compiler (Synopsys),
RTL Compiler (Cadence)

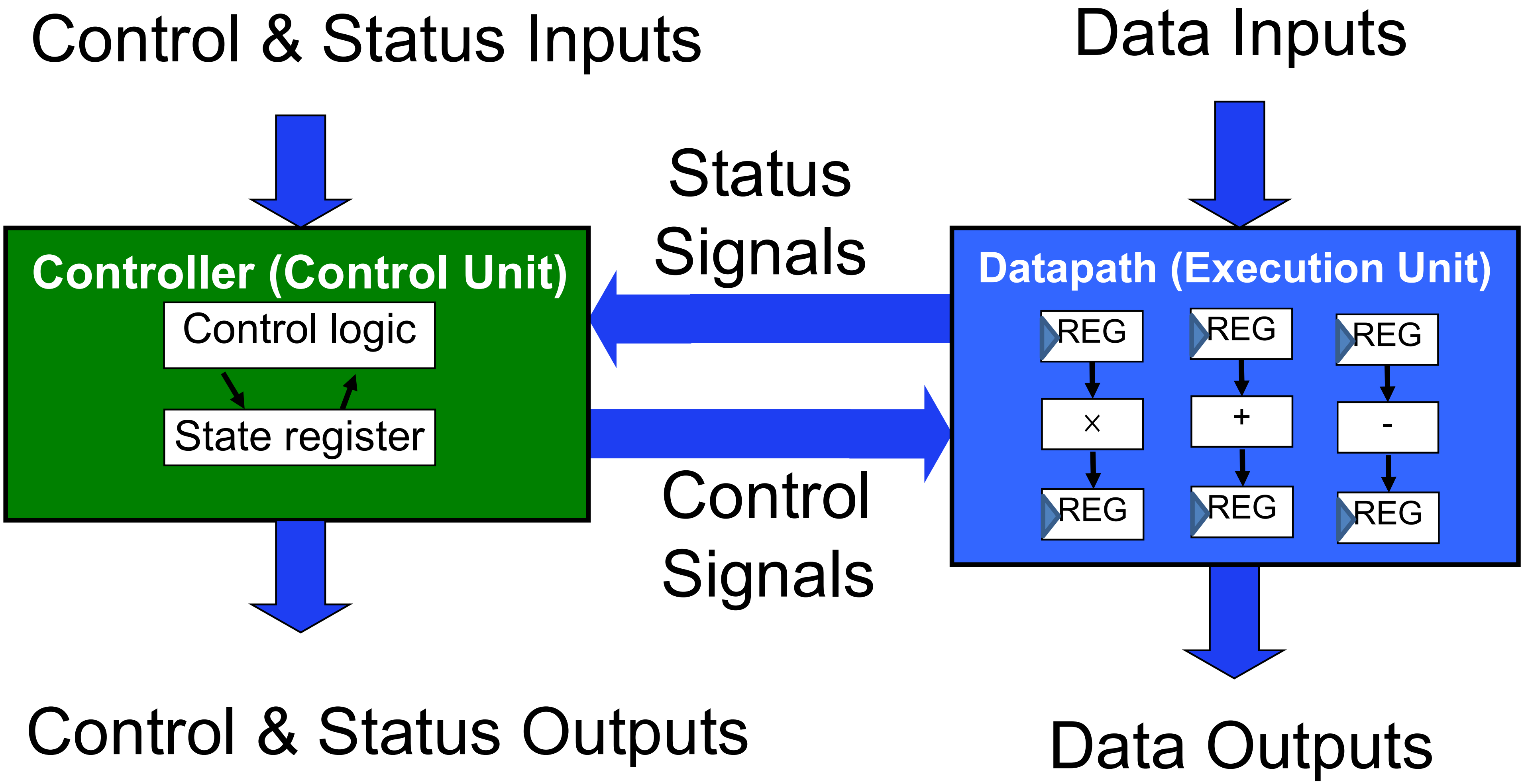
Tools:
Primetime (Synopsys)

Tools:
IC Compiler (Synopsys),
Encouter (Cadence),

RTL Design

- **How the data are manipulated and transferred between registers**
- **RT Level architecture are constructed from the *basic building blocks:***
  - Use registers to store the intermediate data and to imitate the variables used in an algorithm.
  - Use a custom *data path* to realize all the required operations, including:
    - data manipulation circuit: **adders, multiplier and comparators, …**
    - routing components: (de)multiplexer
    - storage components**:** registers
  - Use a custom control path to specify the order of the register operations.
    - realized by an FSM (Finite State Machine)

RTL Structure of a Typical Digital Circuit



Control & Status Inputs

Data Inputs

Status Signals

Controller (Control Unit)

Control logic

State register

Datapath (Execution Unit)

REG

REG

REG

×

+

-

REG

REG

REG

Control Signals

Control & Status Outputs

Data Outputs

The goals of RTL design

- Decide on the necessary ***basic building blocks*** for carrying out computations on data and/or signal processing.
- Organize their interaction to meet target specifications.
- Concerns:
  - Functional correctness
  - Performance targets (throughput, operation rate, etc.)
  - Circuit size
  - Energy efficiency
  - Flexibility (adaptive to evolving needs, changing specs, future standards)
  - Engineering effort and time-to-market

RTL Design Method (1)

- First create algorithm for the solution to the problem
- Write an algorithmic description

Write an algorithmic description

- The programming language used to formalize an algorithmic solution to design problem is referred to as mini-C (a derivative of the C-programming language)
  - **IF**

    ```
    if (condition) then BODY_1 else BODY_2
    ```
  - **FOR**

    ```
    for (i=A; i<B; i += 1)
    ```
  - **WHILE**

    ```
    while(condition) BODY
    ```
  - **ASSIGNMENT**

    ```
    X = value
    ```

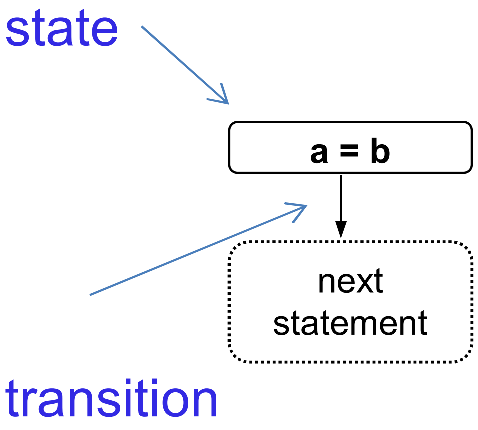
RTL Design Method (2)

- First create algorithm for the solution to the problem
- Write an algorithmic description
- Convert algorithm to "complex" state machine
  - Known as FSMD: finite-state machine with datapath (or CDFG: control/Data flow Graph)
  - Can use **templates** to perform such conversion based on the classification of statements:
    - Assignment statements
    - Loop statements (*For* or *while*)
    - Branch statements (*if-then-else* or *case*)

State diagram templates

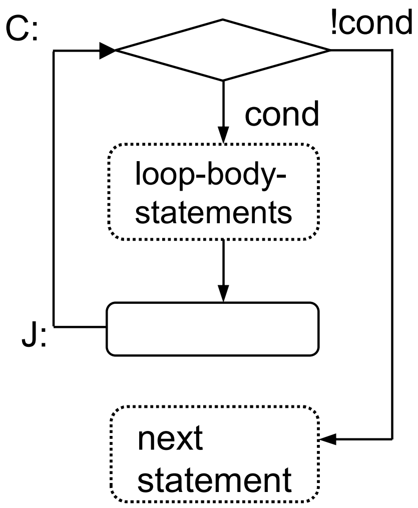
Assignment statement

```
a = b
next tatement
```



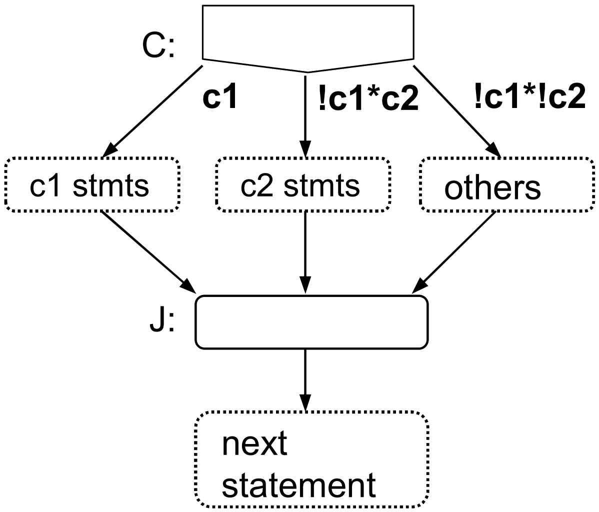
Loop statement

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



Branch statement

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



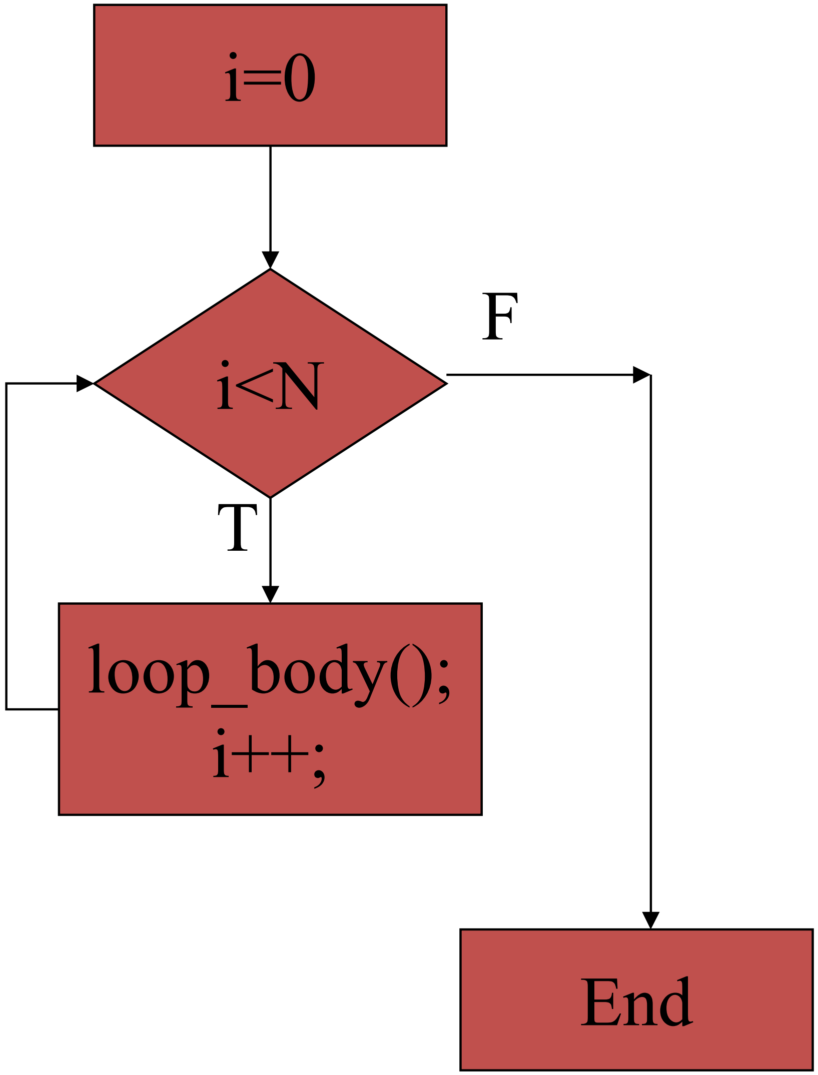
Loop statements

```
for (i=0; i<N; i++)
    loop_body();
```

for loop

```
i=0;
while (i<N) {
    loop_body(); i++; }
```

equivalent



RTL Design Method (3)

- First create algorithm for the solution to the problem
- Write an algorithmic description
- Convert algorithm to “complex” state machine
  - Known as FSMD: finite-state machine with datapath (or CDFG)
  - Can use **templates** to perform such conversion based on the classification of statements:
    - Assignment statements
    - Loop statements (*For* or *while*)
    - Branch statements (*if-then-else* or *case*)
- Partition FSMD into a datapath part and controller part
  - The datapath contains a netlist of functional units like multiplexors, registers, subtractors and a comparator, …
    - This design is structural
  - The controller is an FSM which issues control signals to the datapath based on the current state and the external inputs.
    - This can be a behavioral description.

Partition FSMD – If/Then/Else

```
if boolean_expr_1 then
        sequential_statements;
elsif boolean_expr_2 then
        sequential_statements;
elsif boolean_expr_3 then
        sequential_statements;
…
else
        sequential_statements;
end if;
```

Partition FSMD – If/Then/Else

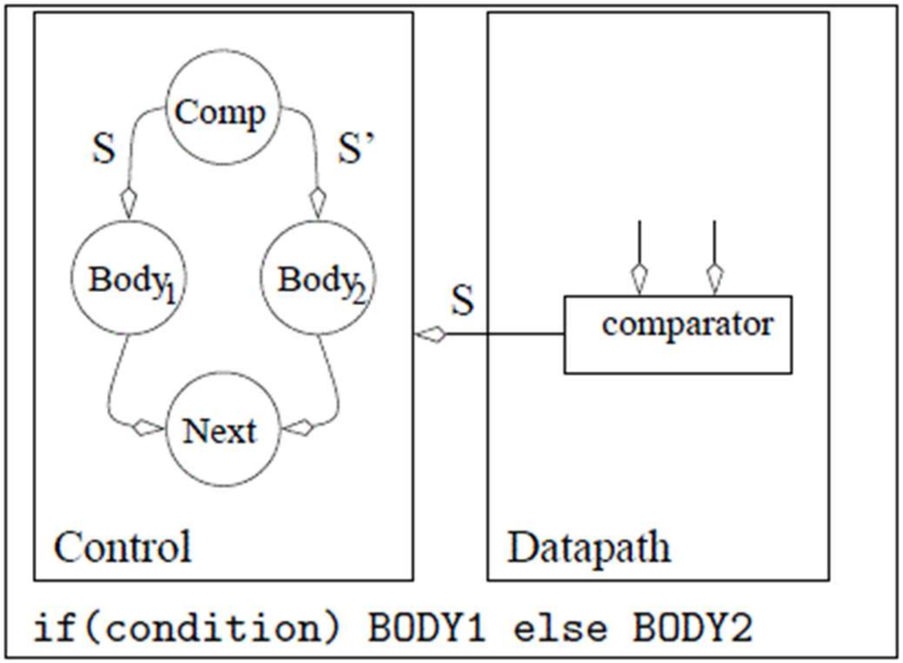


Fig 1 - The datapath and control components required to realize an if/then/else structure.

Partition FSMD – For Loop

```
for index in loop_range loop
   sequential statements;
end loop;
```

Partition FSMD – For Loop

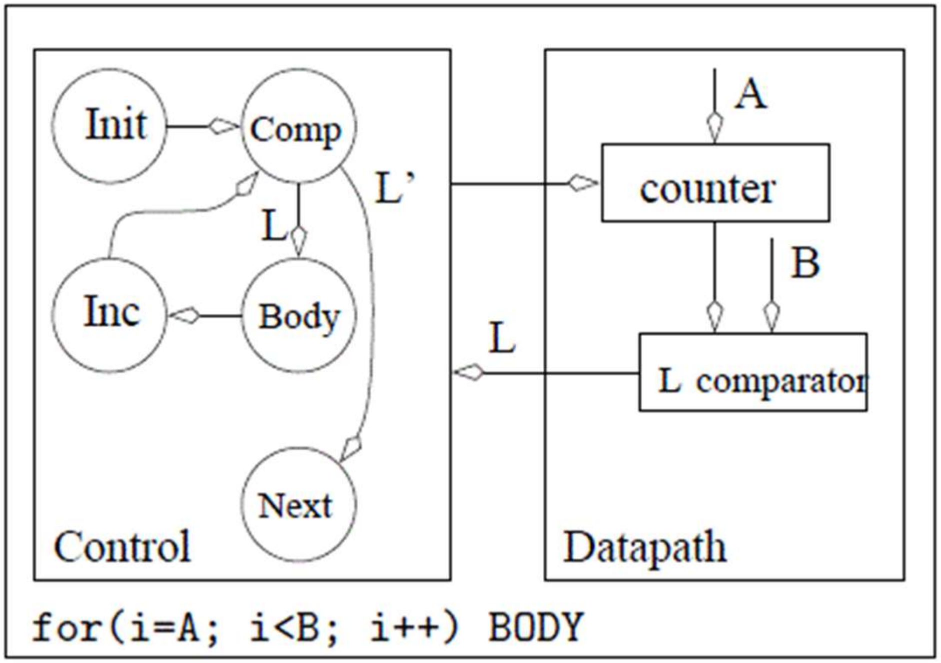


Fig 2 - The datapath and control components required to realize a for loop.

Partition FSMD – While Loop

```
while condition loop
  sequential statements;
end loop;
```

Partition FSMD – While Loop

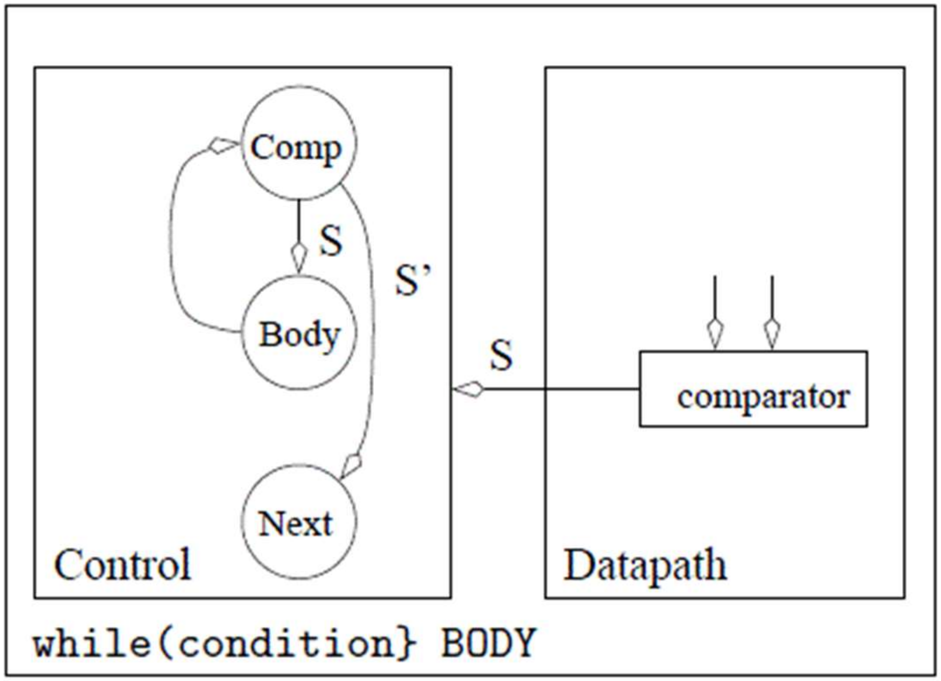


Fig 3 - The datapath and control components required to realize a while statement.

Partition FSMD – Assignment

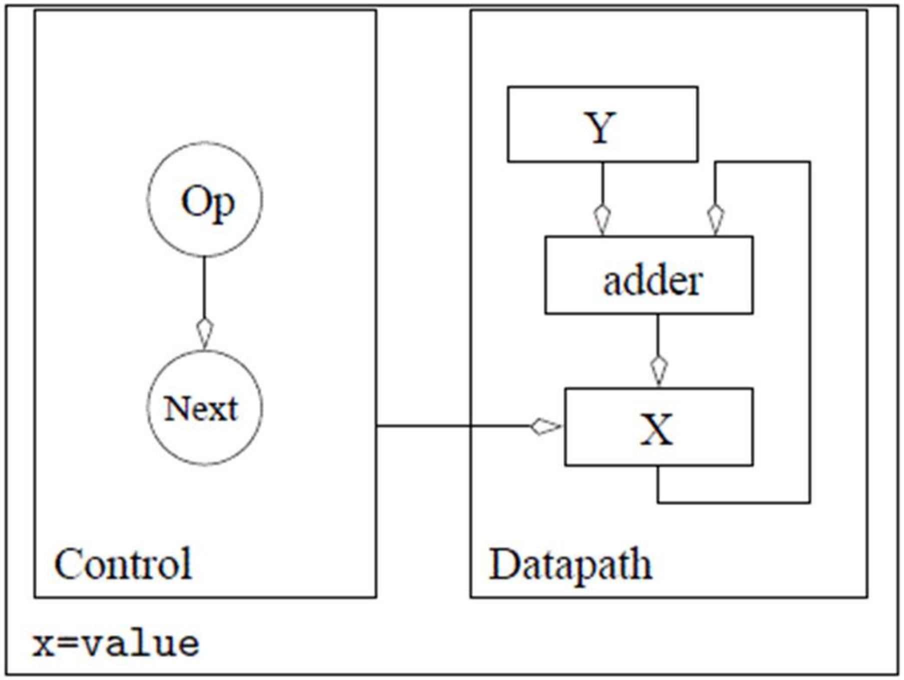


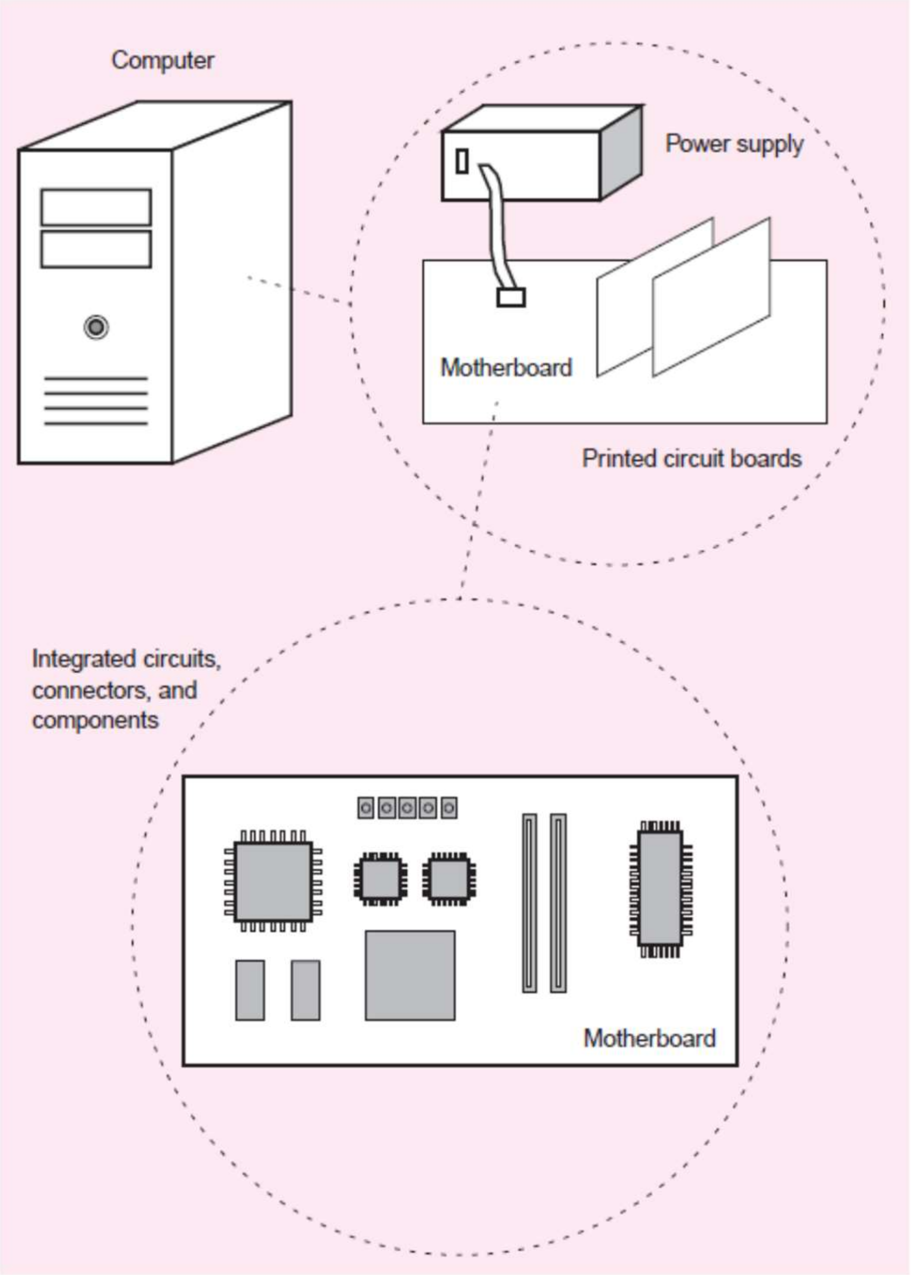
Fig 4 - The datapath and control components required to realize an assignment statement of the form X=X+Y.

Outline

- RTL design methodology
- **Review of Combinational and Sequential logic design**
- Single-purpose Vs. general-purpose architectures
- RT-level custom single-purpose circuit design
- RT-level custom general-purpose processor design

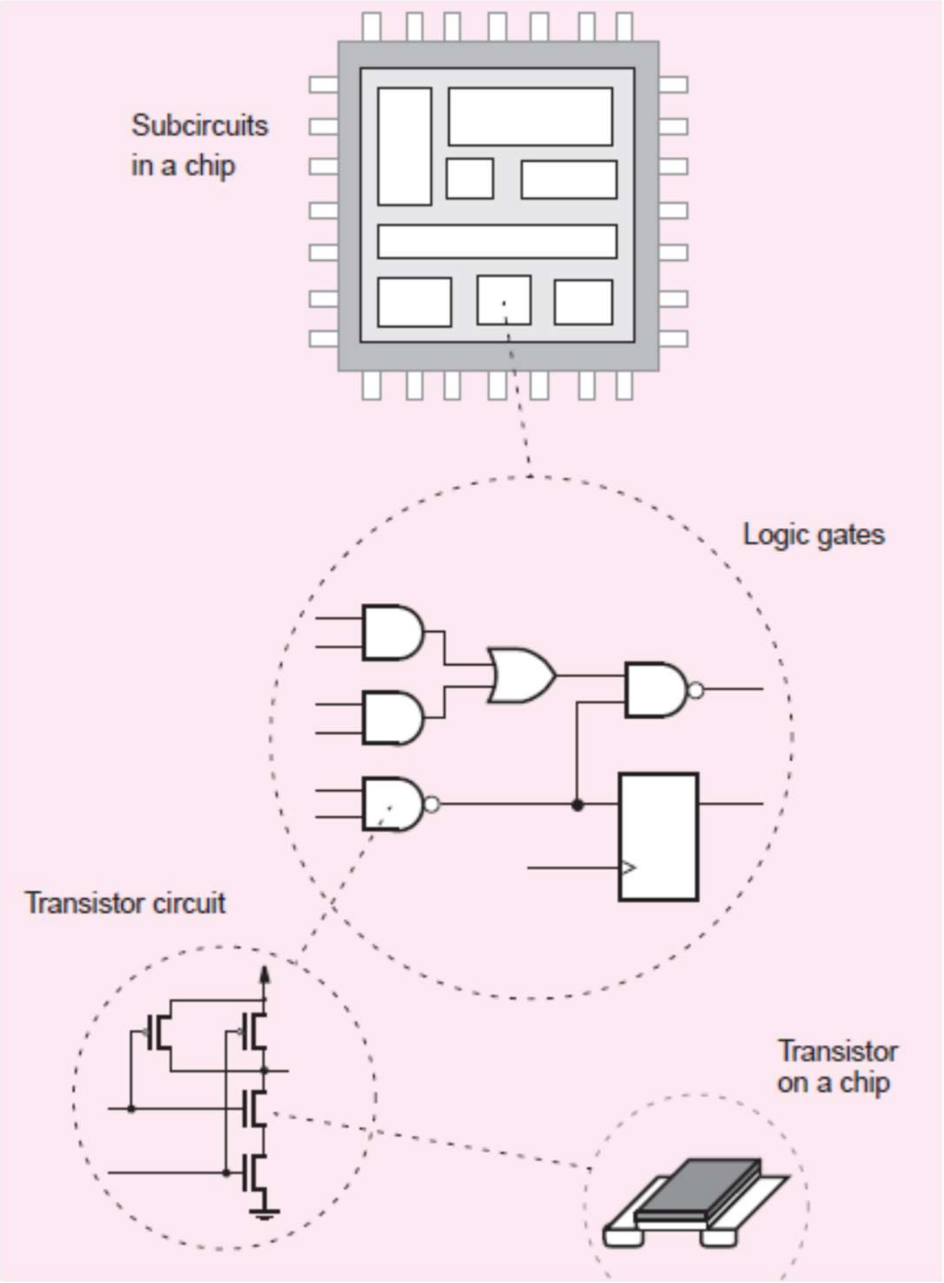
Review

- Structure of Computer
  - Power Supply
  - Motherboard
    - ICs (Integrated Circuits)
  - daughter boards



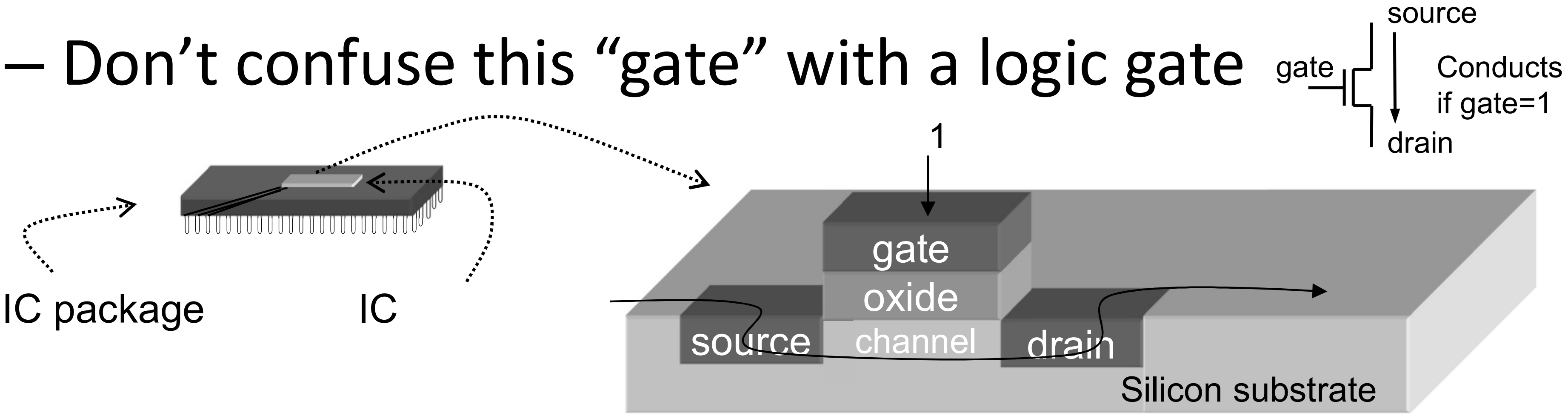
Review

- Structure of a IC
  - Include several Sub-circuits
    - arithmetic operations, store data, or control the flow of data
  - Sub-circuit comprises a network of connected *logic gates*
    - logic gate performs a very simple function
    - Logic gates are built with transistors



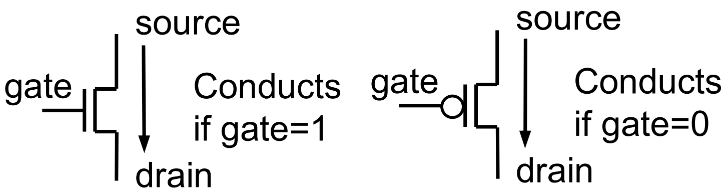
CMOS transistor on silicon

- Complementary Metal Oxide Semiconductor
- Transistor
  - The basic electrical component in digital systems
  - Acts as an on/off switch in digital circuit
  - Voltage at “gate” controls whether current flows from source to drain
  - Don’t confuse this “gate” with a logic gate

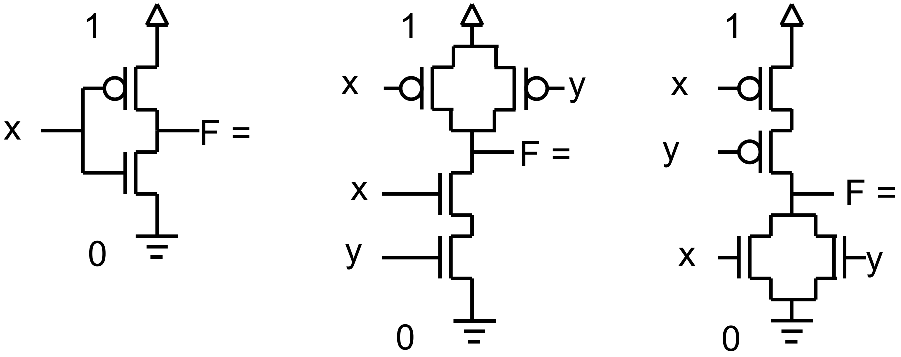


CMOS transistor implementations

- We refer to logic levels
  - Typically 0 is 0V, 1 is 5V
- Two basic CMOS types
  - nMOS conducts if gate=1; good for conducting “0”
  - pMOS conducts if gate=0; good for conducting “1”
  - Hence “complementary”
- Basic gates:
  - NAND, Inverter, NOR



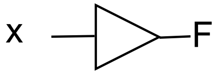
nMOS pMOS



inverter NAND gate NOR gate

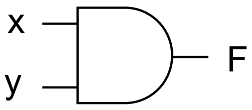
Basic logic gates

- Digital system designers usually work at abstraction level of logic gates:



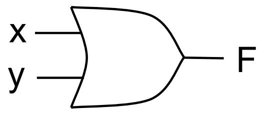
F = x
Driver

x	F
0	0
1	1



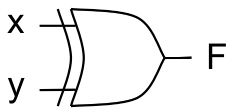
F = x y
AND

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



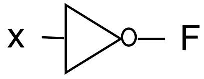
F = x + y
OR

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



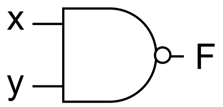
F = x $\oplus$ y
XOR

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



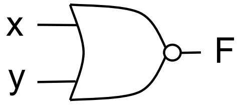
F = x’
Inverter

x	F
0	1
1	0



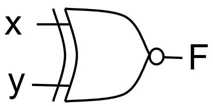
F = (x y)’
NAND

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



F = (x+y)’
NOR

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



F = x $\odot$ y
XNOR

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

Combinational circuit design

- Digital circuits whose output is purely a function of its present inputs

A) Problem description

y is 1 if a is 1, or b and c are 1. z is 1 if b or c is 1, but not both (or, a, b, and c are all 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	1	0
1	1	1	1	1

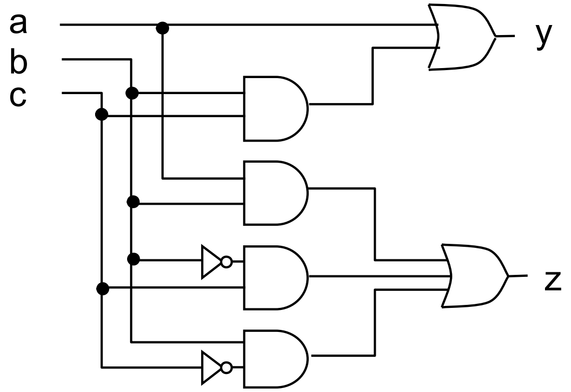
y = a + bc

z

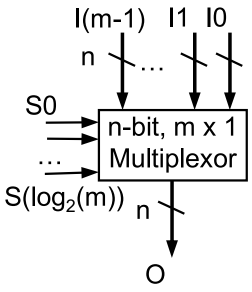
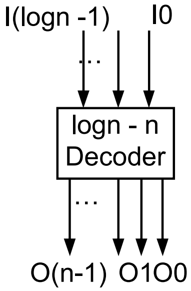
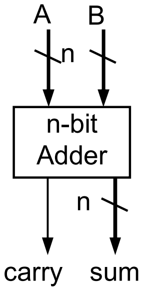
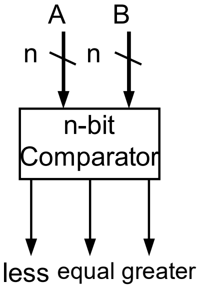
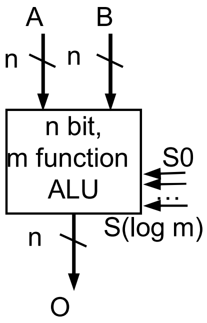
a \ bc	00	01	11	10
0	0	1	0	1
1	0	1	1	1

z = ab + b’c + bc’

E) Logic Gates



RT-Level Combinational components

 I(m-1) I1 I0 n … S0 n-bit, m x 1 Multiplexor … S(log$_2$(m)) n O	 I(logn -1) I0 … logn - n Decoder … O(n-1) O1O0		 A B n n n-bit Comparator less equal greater	 A B n n n bit, m function ALU S0 … S(log m) n O

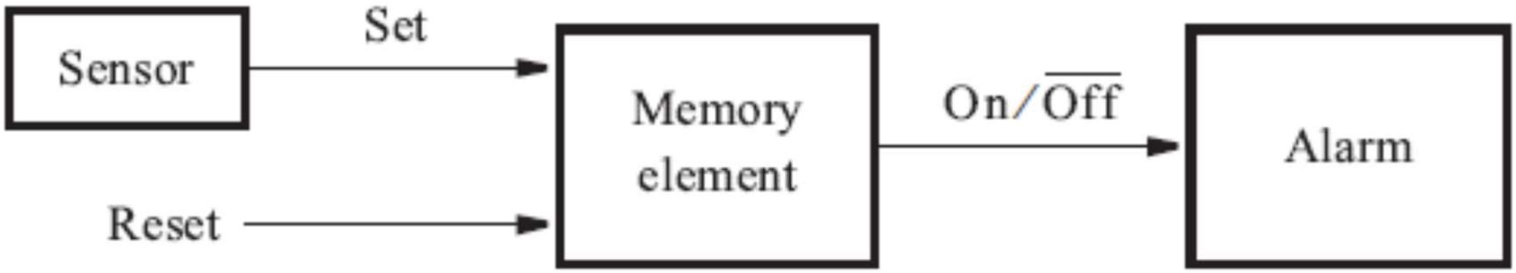
O =
I0 if S=0..00
I1 if S=0..01
…
I(m-1) if S=1..11
	O0 =1 if I=0..00
	O1 =1 if I=0..01
	…
	O(n-1) =1 if I=1..11
		sum = A+B
		(first n bits)
		carry = (n+1)'th
		bit of A+B
			less = 1 if A<B
			equal =1 if A=B
			greater=1 if A>B
				O = A *op* B
				op determined by S.

	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.

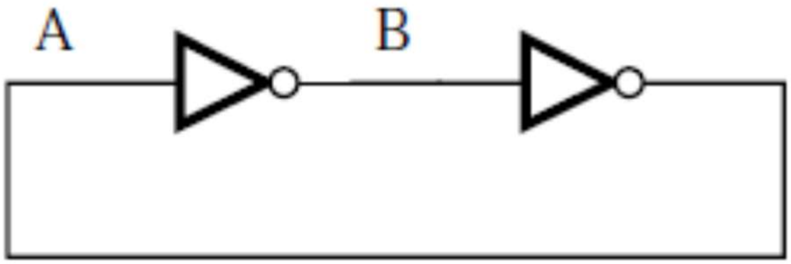
Others: Barrel Shifter

Sequential circuit

- **Sequential Circuits:** Digital circuits whose outputs is a function of the present as well as previous inputs; have memory.
    - Examples: latches and flip-flops



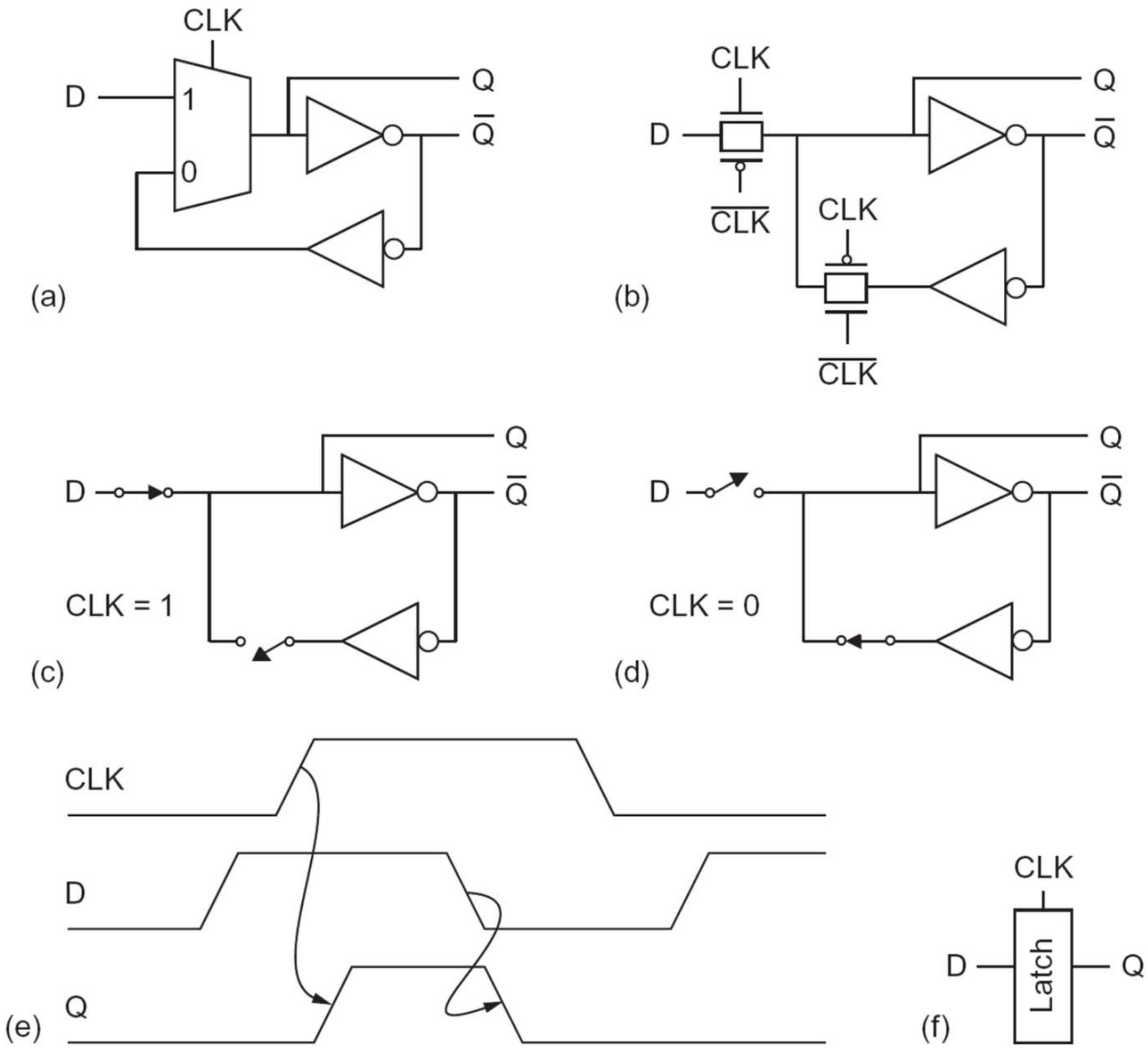
Control of an alarm system



A simple memory element

Sequential circuit

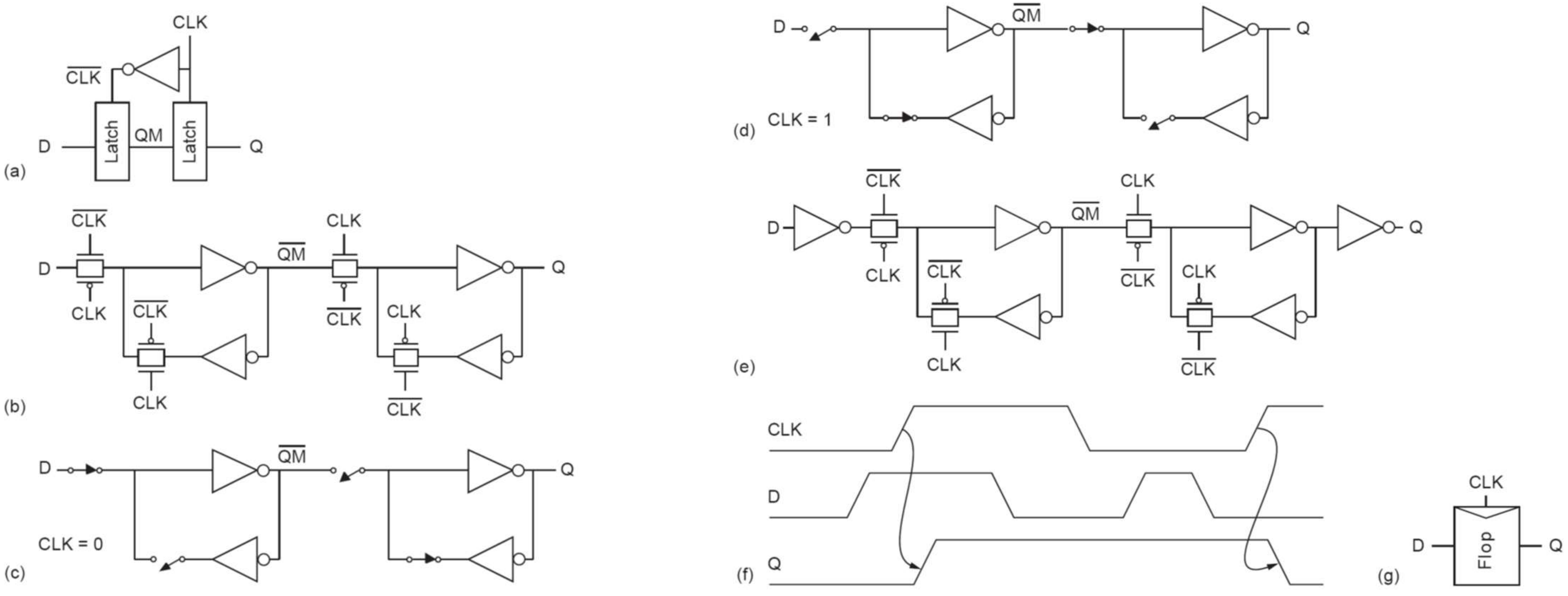
- **D Latch:**



CMOS positive-level-sensitive *D* latch

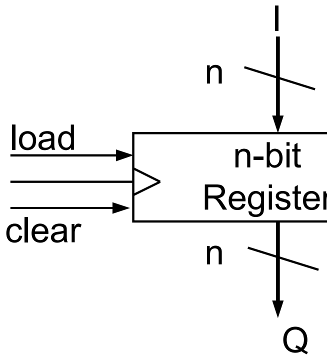
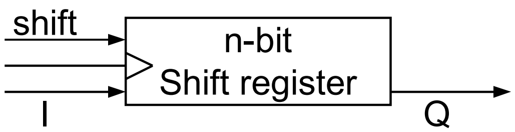
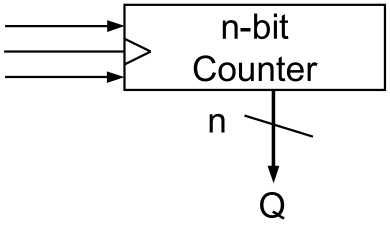
Sequential circuit

- **Flip-Flops:**
  - combining between a negative-level-sensitive latch and one positive-level-sensitive latch
  - Can be triggered by either positive-edge or negative-edge triggered



CMOS positive-edge-triggered D flip-flop

RT-Level Sequential components

		
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.

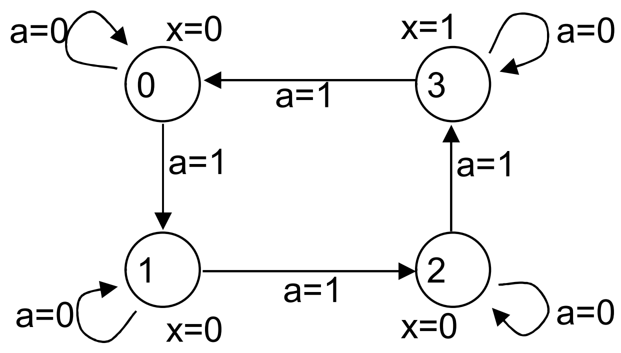
Sequential logic design: FSM

A) Problem Description

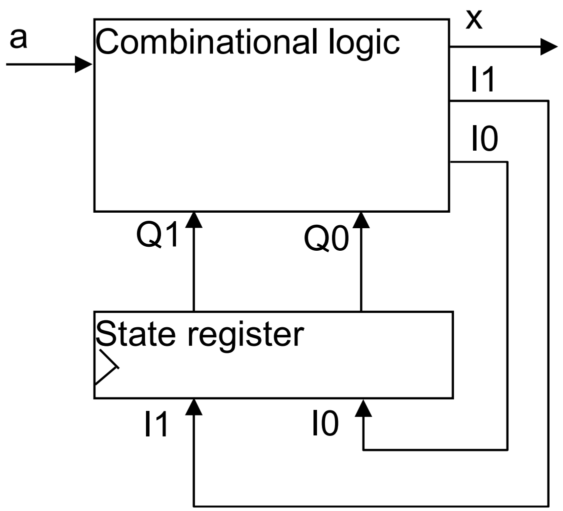
You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles

a: input; x: output

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: FSM (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

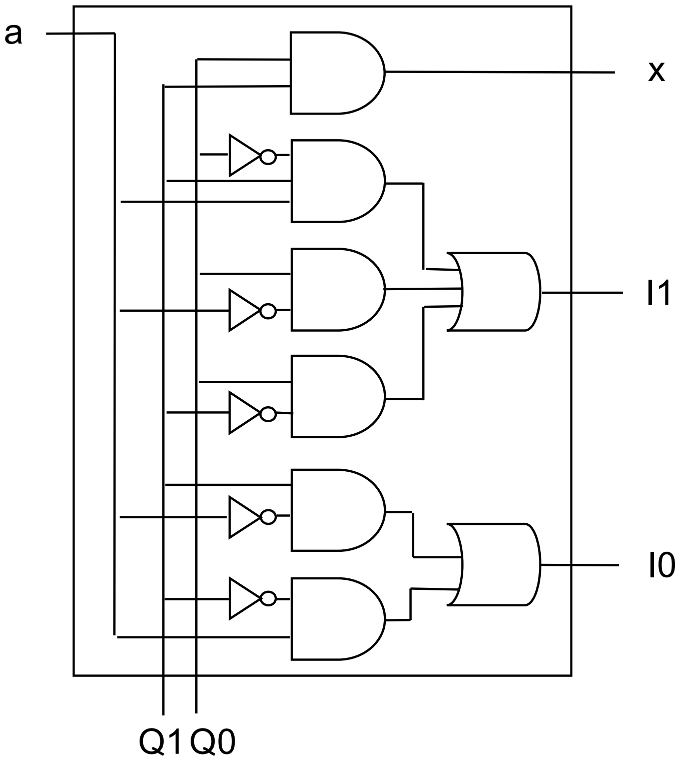
I0 = Q0a' + Q0'a

x

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

x = Q1Q0

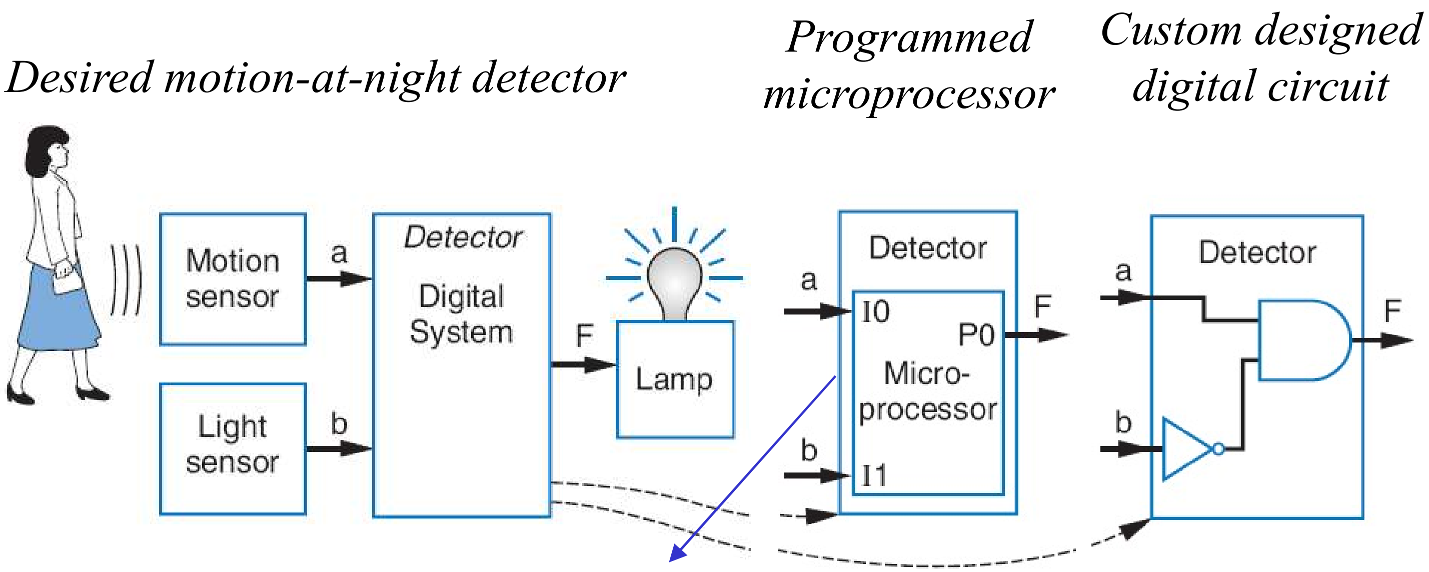
F) Combinational Logic



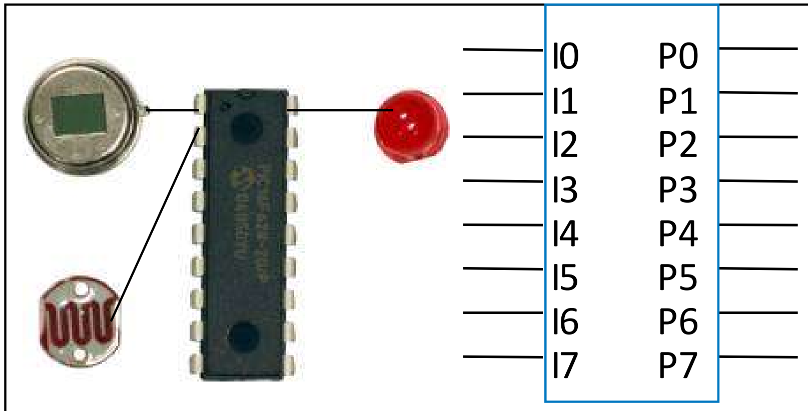
Outline

- RTL design methodology
- Review of Combinational and Sequential logic design
- **Single-purpose vs. general-purpose architectures**
- RT-level custom single-purpose circuit design
- RT-level custom general-purpose processor design

Implementing Digital Systems:
Programming Microprocessors Vs. Designing Digital Circuits

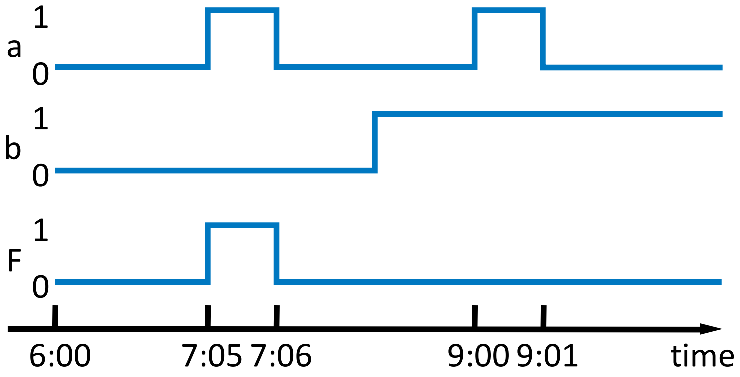


- Microprocessors is a common choice to implement a digital system
  - Easy to program
  - Cheap (as low as $1)
  - Available now



Microprocessor

```
void main()
{
  while (1) {
    P0 = I0 && !I1;
    // F = a and !b,
  }
}
```



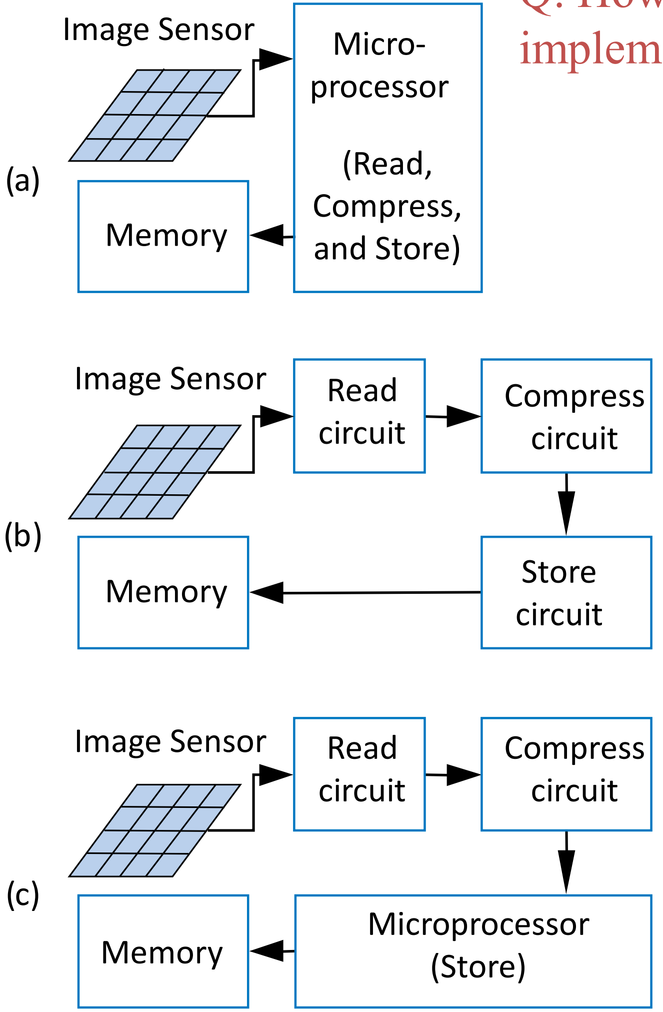
Timing diagram

Digital Design: When Microprocessors Aren’t Good Enough

- With microprocessors so easy, cheap, and available, why design a digital circuit?
  - Microprocessor may be too slow
  - Or too big, power hungry, or costly

Sample digital camera task execution times (in seconds) on a microprocessor versus a digital circuit:

Task	Microprocessor	Custom Digital Circuit
Read	5	0.1
Compress	8	0.5
Store	1	0.8



Q: How long for each implementation option?

5+8+1 =14 sec

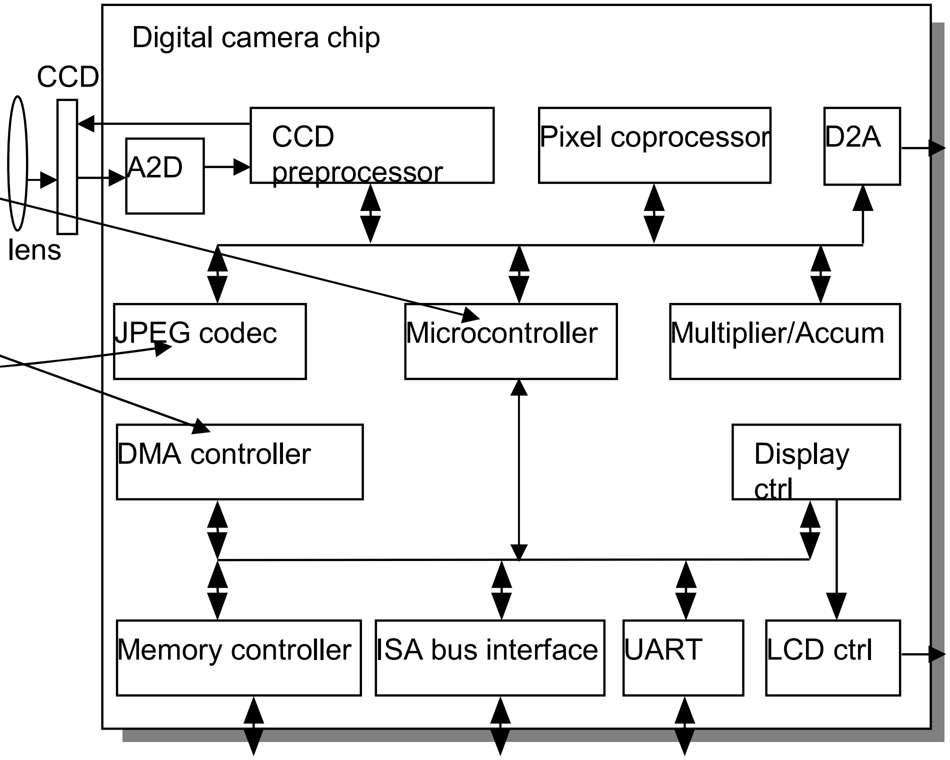
.1+.5+.8 =1.4 sec

.1+.5+1 =1.6 sec

Good compromise

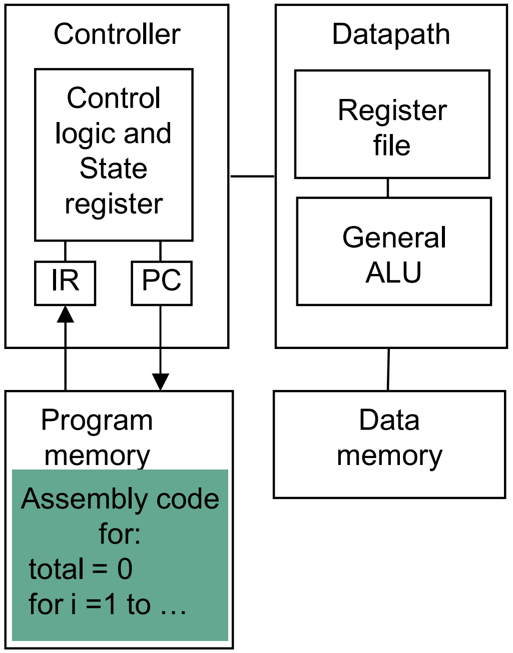
Architectures of VLSI Circuits

- Digital IC
  - Digital circuit that performs a computation tasks
  - General-purpose: variety of computation tasks
  - Standard Single-purpose: one particular computation task
  - Custom single-purpose: non-standard task
- A single-purpose architecture may be
  - Fast, small, low power
  - But, high NRE, longer time-to-market, less flexible

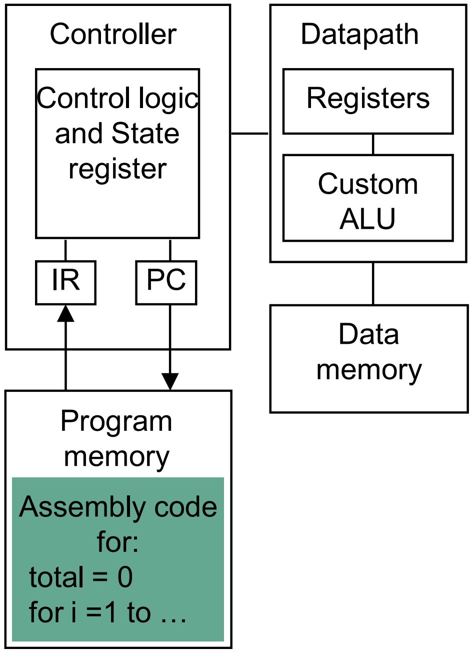


Architectures of VLSI Circuits

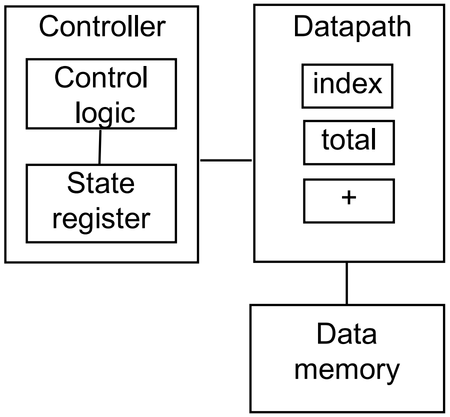
- The architecture of the VLSI circuits used to implement a system’s desired functionality:
  - **Processor**: program-controlled machine
    - General-purpose processor
    - Application-Specific Instruction Processor
  - **Single-purpose architectures**: hardwired electronic circuit does not have programmability



General-purpose processor



Application-specific instruction processor



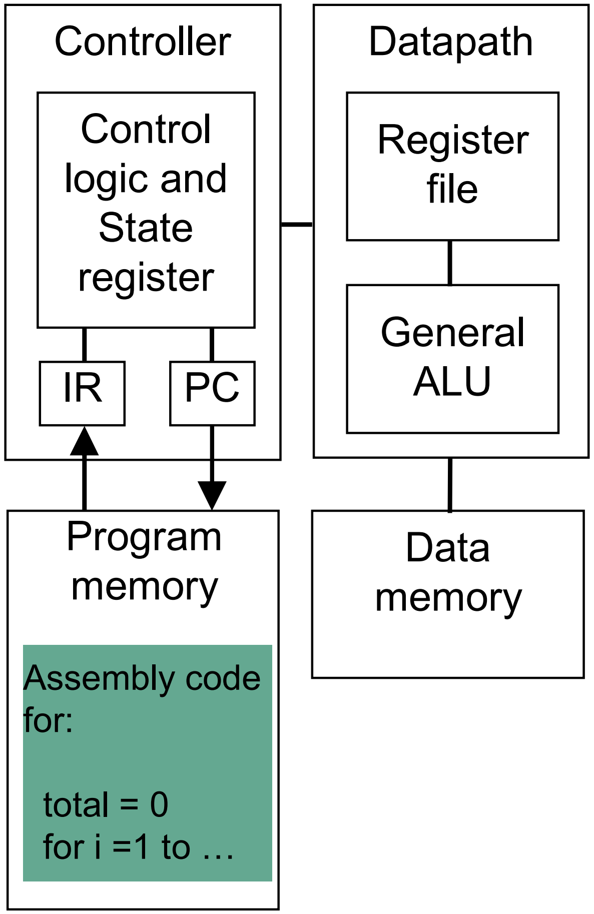
Single-purpose hardware

Architectures of VLSI Circuits

	Hardware architecture	
	General Purpose	Single purpose
Algorithm	Any, not know a priori	Fixed, must be known
Architecture	Instruction set processor	Dedicated
Execution model	Fetch-decode-execute-store “instruction-driven”	Process data item “Dataflow-driven”
Datapath	ALU(s) + Memory	Customized design
Controller	With program microcode	Typically hardwired
Performance indicator	Instructions per second Run time of benchmarks	Data throughtput, can be anticipated analytically
Strengths	Highly flexible, immediately available, routine design flow, low up-front costs	Max. performance, highly energy-efficient,
Development effort	Mostly software design	Mostly hardware design

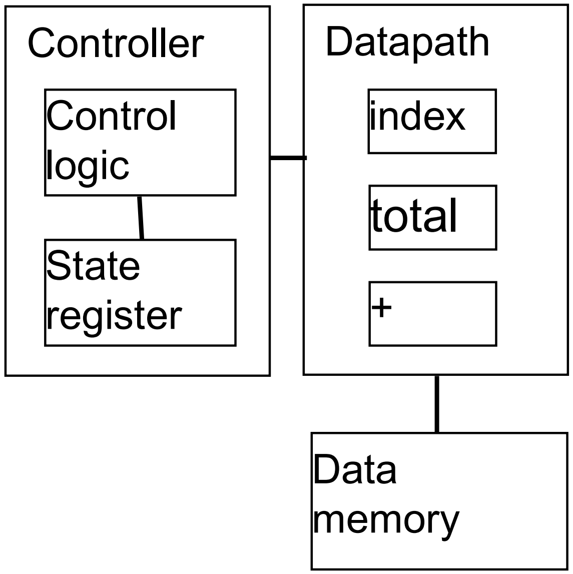
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as “microprocessor”
- Features
  - Program memory
  - General datapath with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility
- Intel, AMD processors are the most well-known, but there are hundreds of others



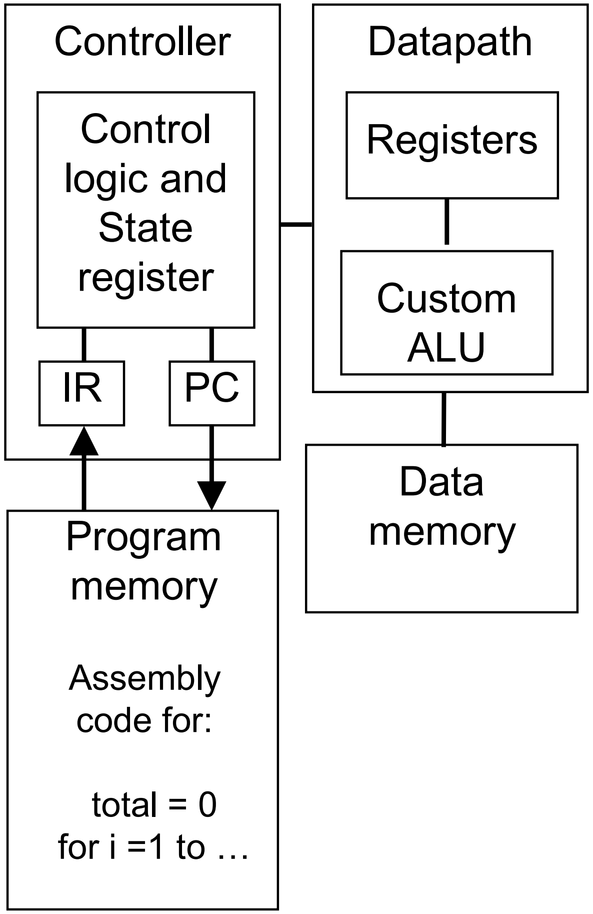
Single-purpose hardware

- Digital circuit designed to execute exactly one function/algorithm
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single function
  - No program memory
- Benefits
  - Fast
  - Low power
  - Small size



Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
  - Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power

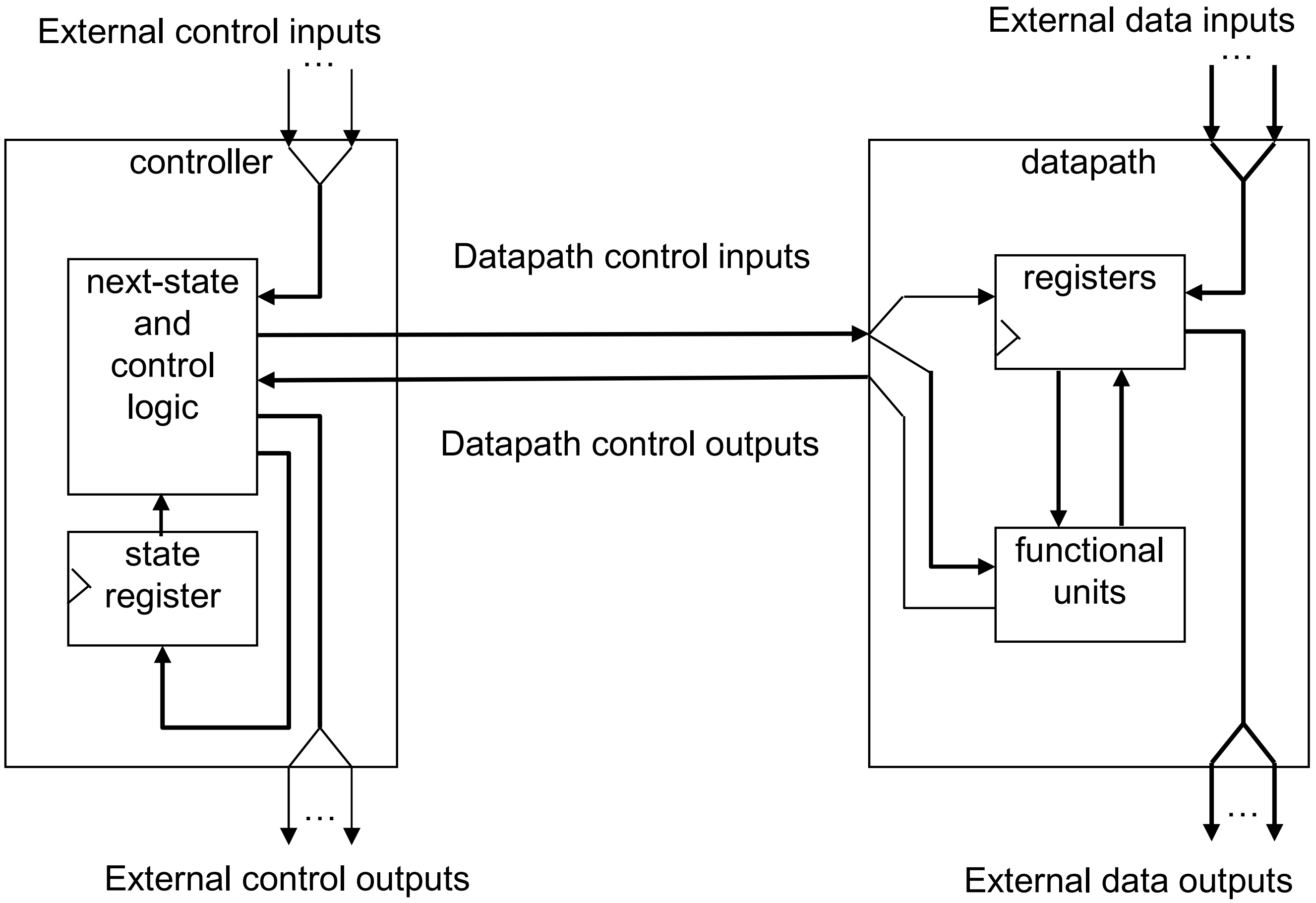


Outline

- RTL design methodology
- Review of Combinational and Sequential logic design
- Single-purpose vs. general-purpose architectures
- **RT-level custom single-purpose circuit design**
  - How to convert an algorithm into a single-purpose architecture;
  - How to reach a optimized design
- RT-level custom general-purpose processor design

Basic Model of Single-Purpose Hardware

FSMD Model: Finite State Machine with Datapath



controller and datapath

Case study: **Greatest Common Divisor (GCD)**

Problem: **Design a architecture for determine the Greatest Common Divisor (GCD) of two integer variables x and y.**

GCD(12,8) = 4

GCD(13,5) = 1

…

GCD(x,y) = ?

GCD: *Mathematical model*

Assume that x > y then

x = a*y + r

- If r = 0 then GCD(x, y) = y

- If r ≠ 0 then GCD(x, y) = GCD(y, r)

GCD: *Creating algorithm*

Algorithm 1

```
 0: int  x, y, r;
 1: while (1) {
 2:    while (!Start_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:    GCD_o = x; Done_o = '1';
    }
```

Reduce the computation complexity

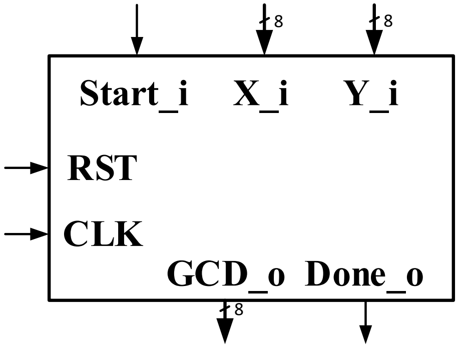
Algorithm 2

```
0: int  x, y;
1: while (1) {
2:   while (!Start_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:   GCD_o = x; Done_o = '1';
  }
```

GCD: *Creating FSMD*

- Convert algorithm to Finite-state machine with Datapath

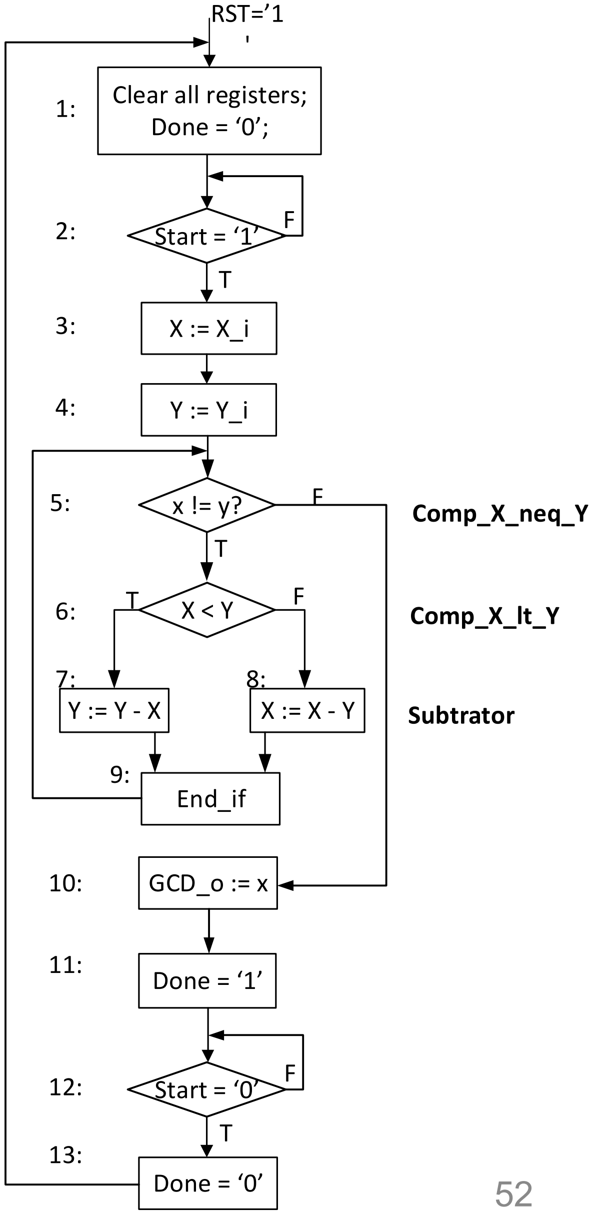
(a) black-box view



(b) desired functionality

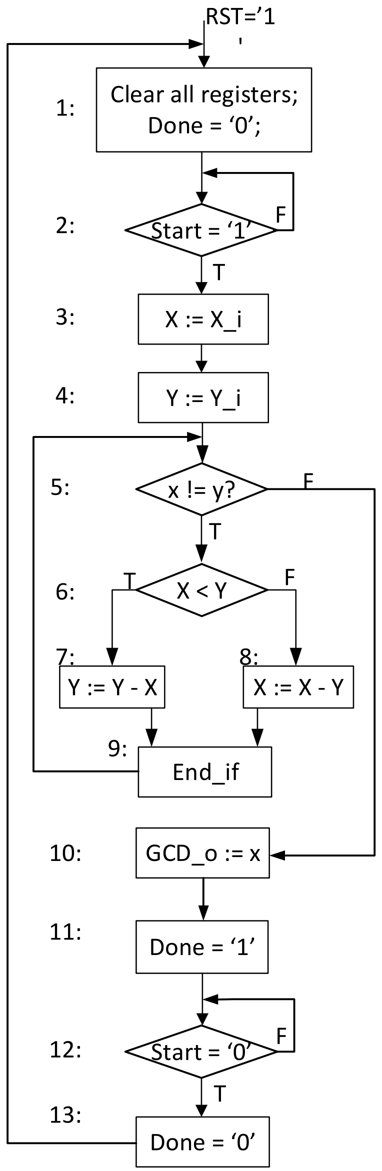
```
0: int  x, y;
1: If (RST = ‘1’) { Done_0 := ‘0’}
   Else {
2:  while (Start_i = ‘1’);
3:   x := x_i;
4:   y := y_i;
5:   while  (x != y)  {
6:       if  (x < y)
7:           y := y - x;
         else
8:           x := x - y;
       }
9:    GCD_o := x; Done_o := ‘1’
   }
```

(c) state diagram FSMD



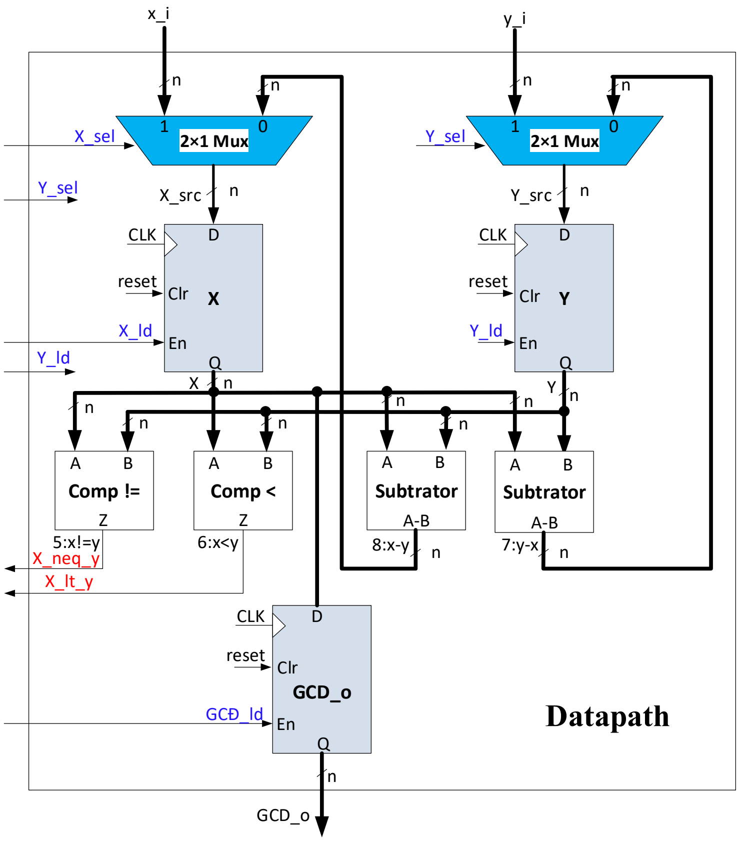
GCD: *Creating the datapath*

(c) state diagram FSMD



(d) Datapath

?????

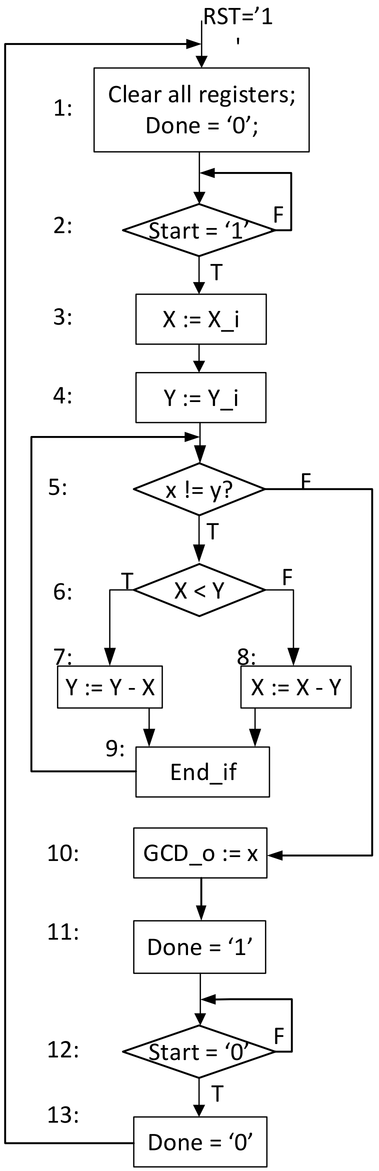


Datapath consists of **interconnection** of **combinational and sequential components**!

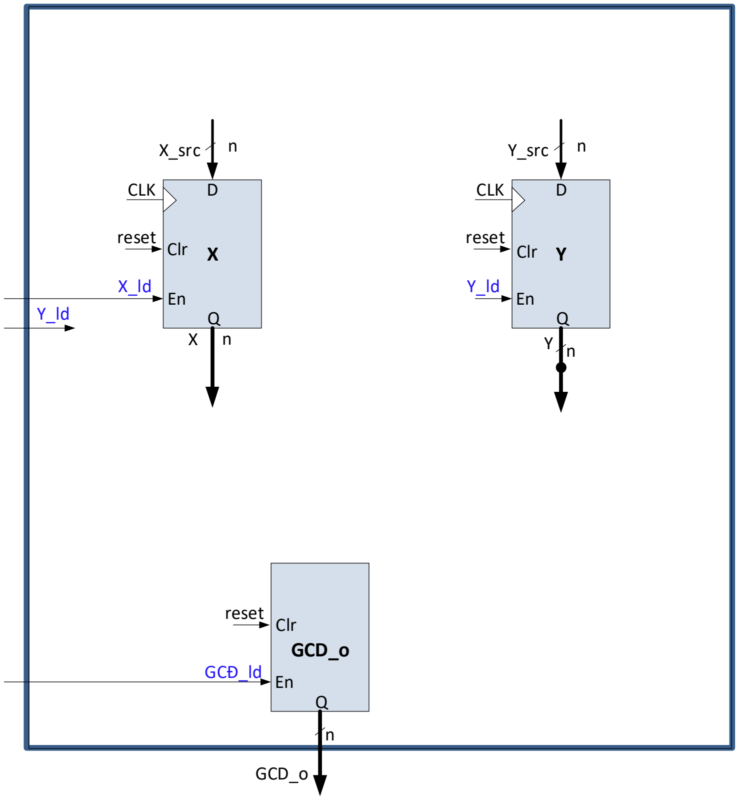
GCD: *Creating the datapath*

1. Create a register for any declared variable and output port

(c) state diagram FSMD



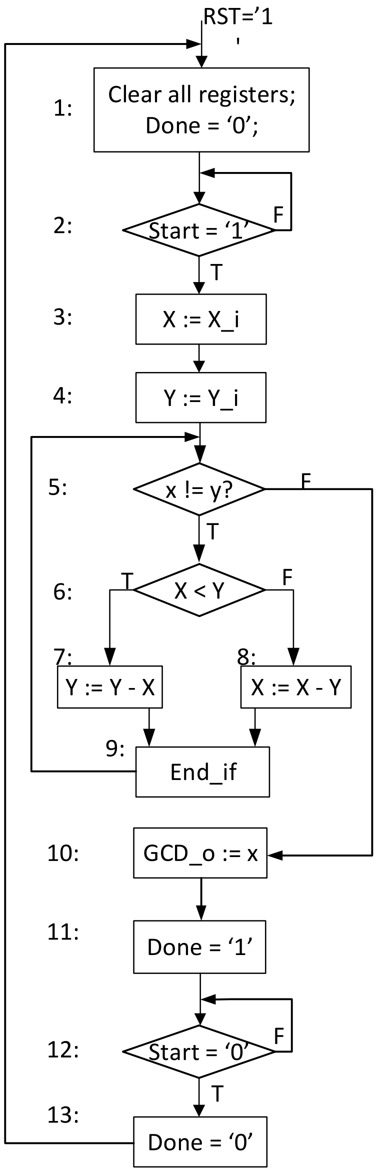
(d) Datapath



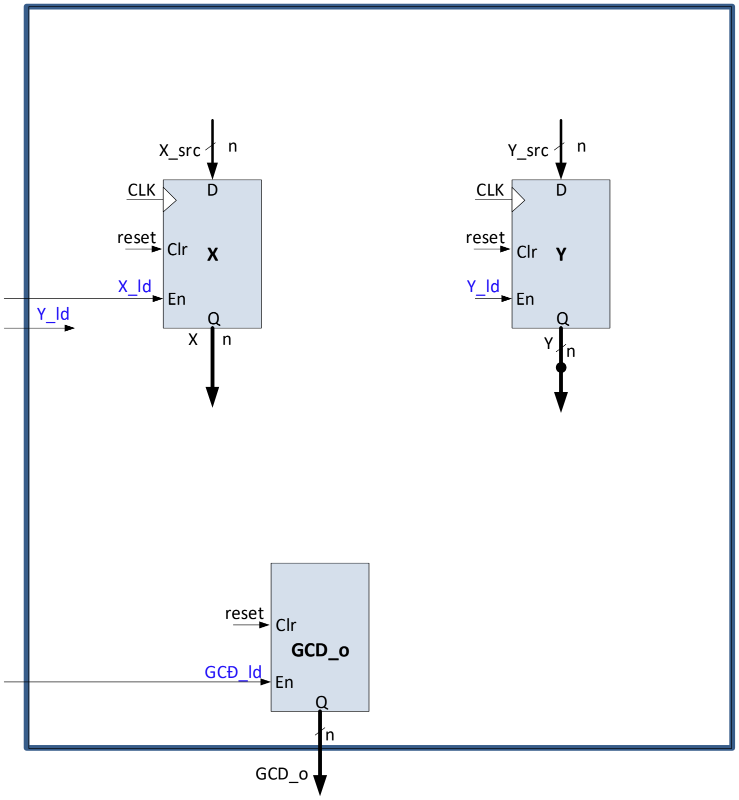
GCD: *Creating the datapath*

1. Create a register for any declared variable and output port
2. Create a functional unit for each arithmetic operation

(c) state diagram FSMD



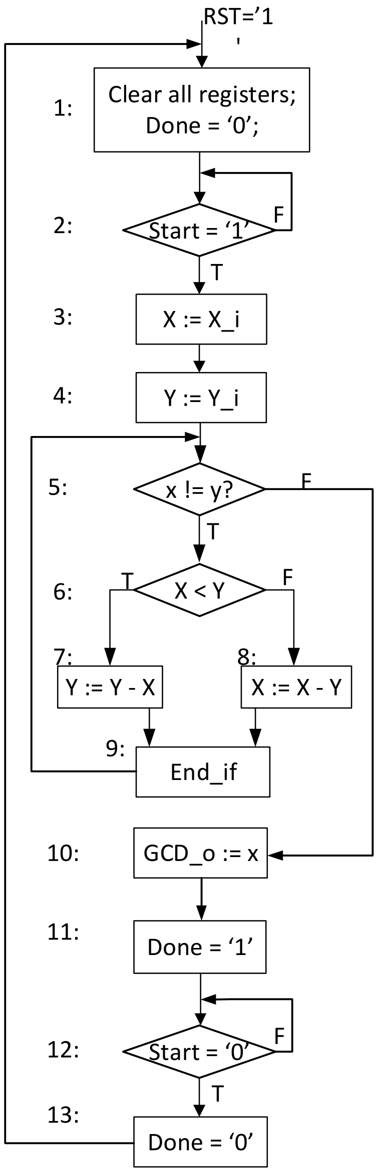
(d) Datapath



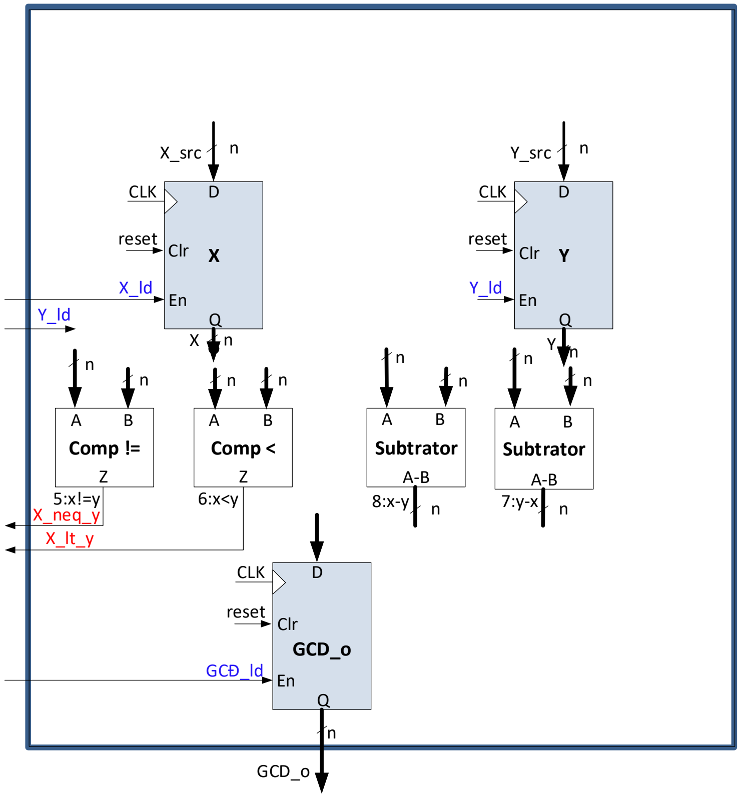
GCD: *Creating the datapath*

1. Create a register for any declared variable and output port
2. Create a functional unit for each arithmetic operation

(c) state diagram FSMD



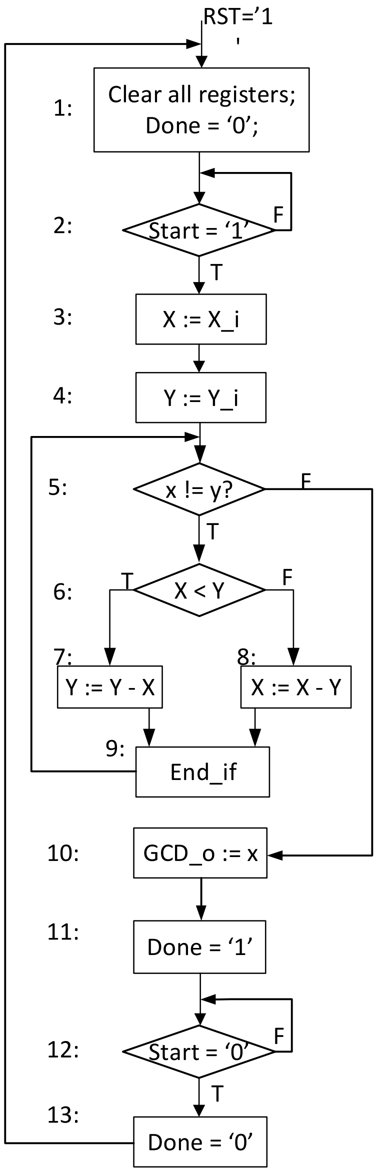
(d) Datapath



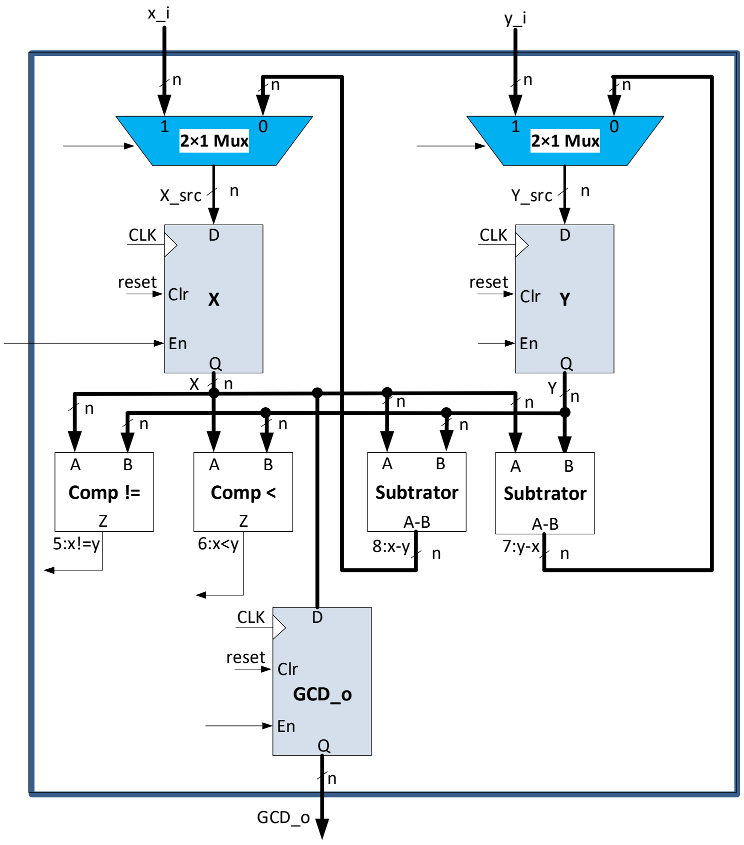
GCD: *Creating the datapath*

1. Create a register for any declared variable and output port
2. Create a functional unit for each arithmetic operation
3. Connect the ports, registers and functional units
   - Based on reads and writes
   - Use multiplexors for multiple sources

(c) state diagram FSMD

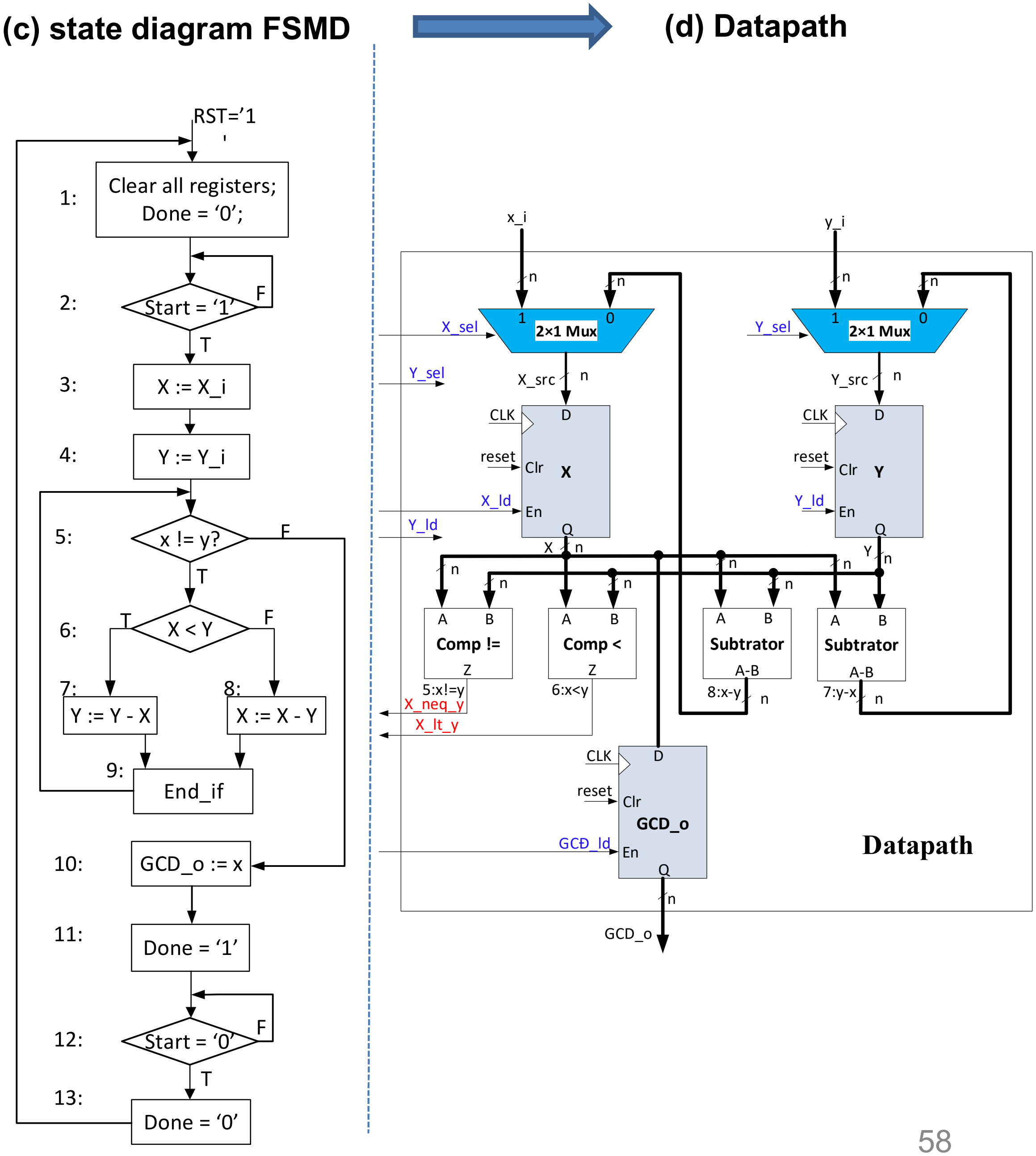


(d) Datapath



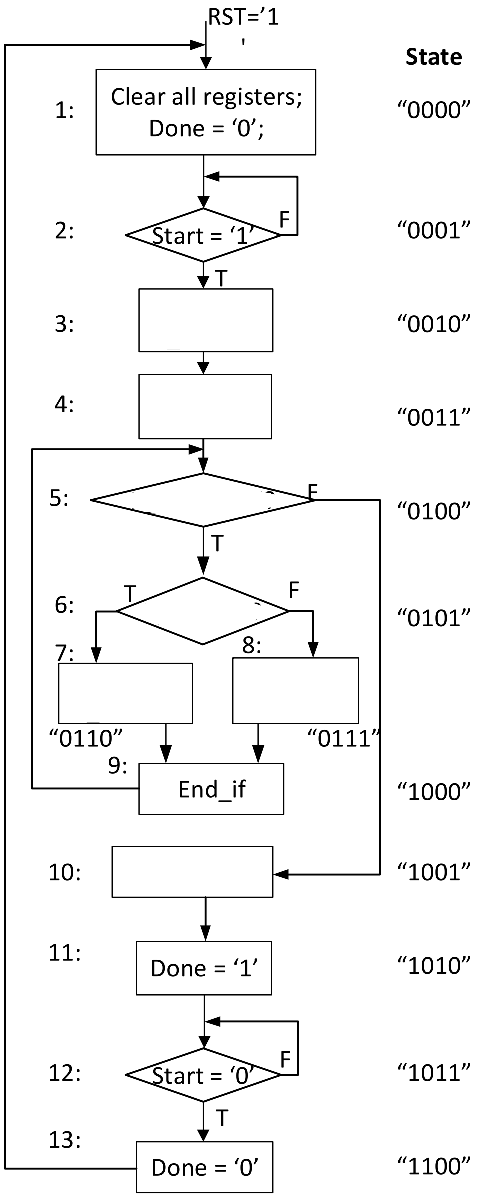
GCD: *Creating the datapath*

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

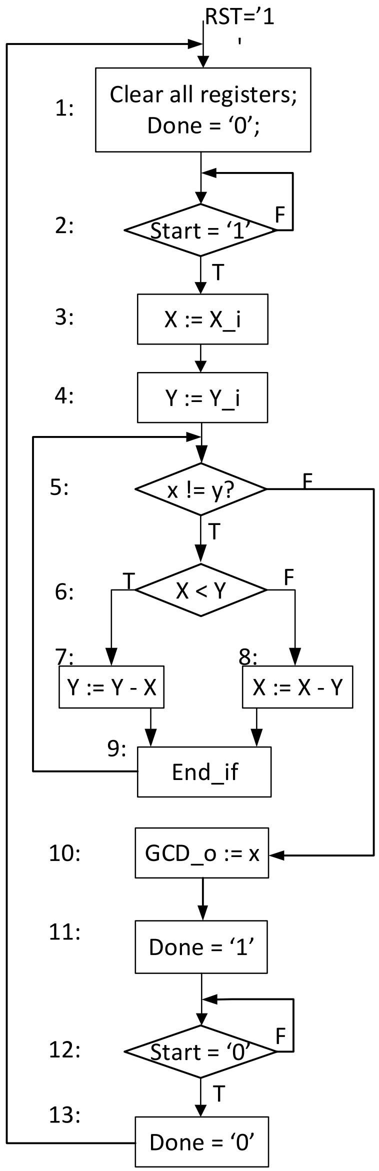


GCD: *Creating the controller's FSM*

(c) state diagram FSM

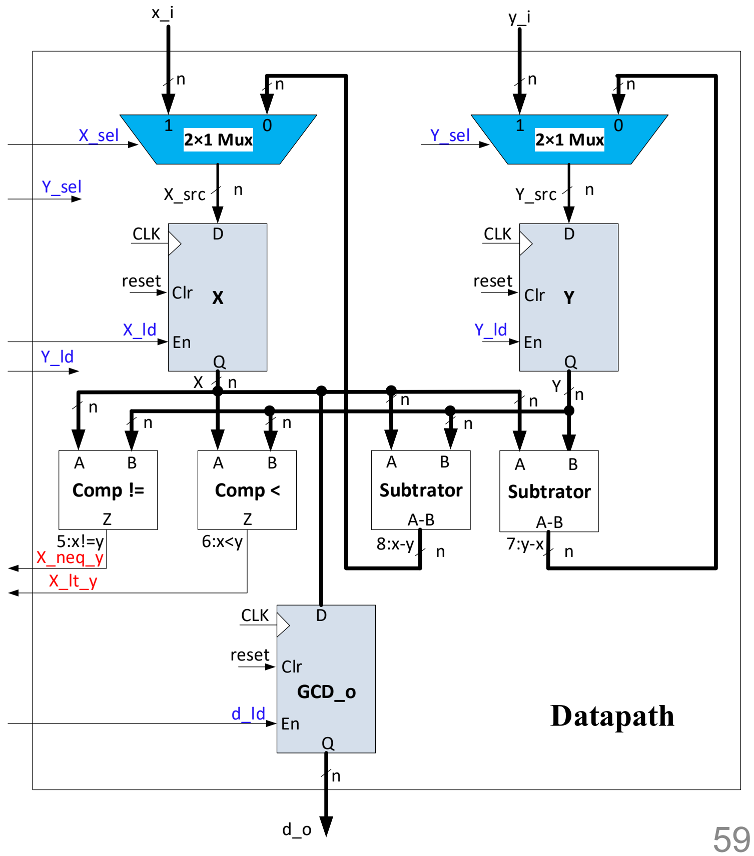


(c) state diagram FSMD



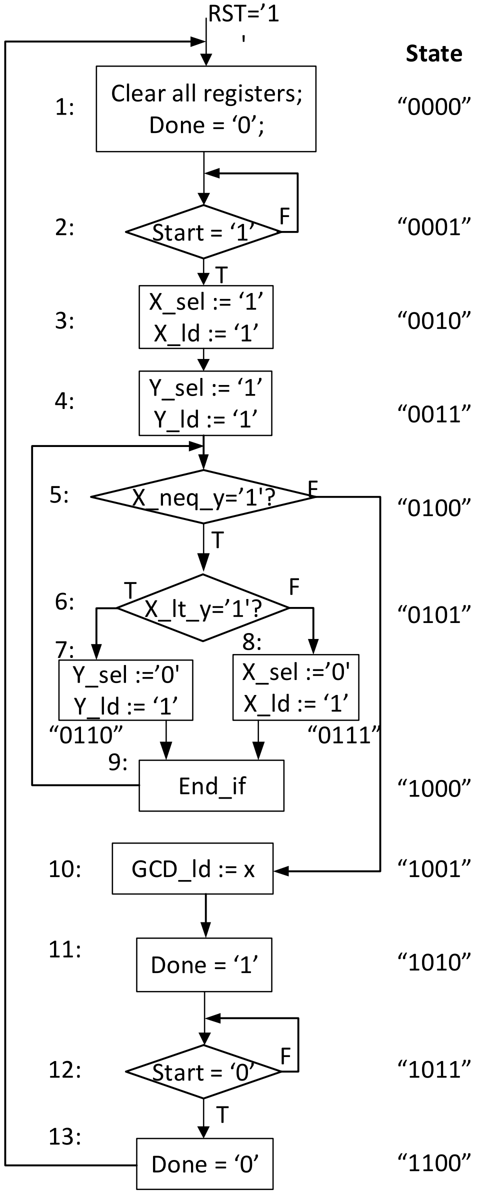
(d) Datapath

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

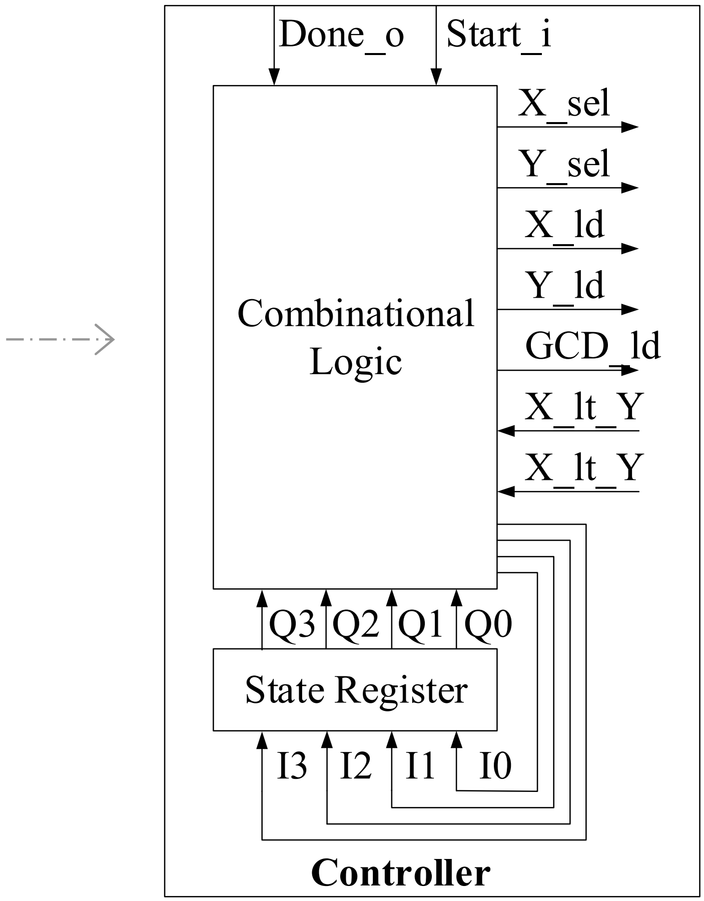


GCD: Splitting into a controller and datapath

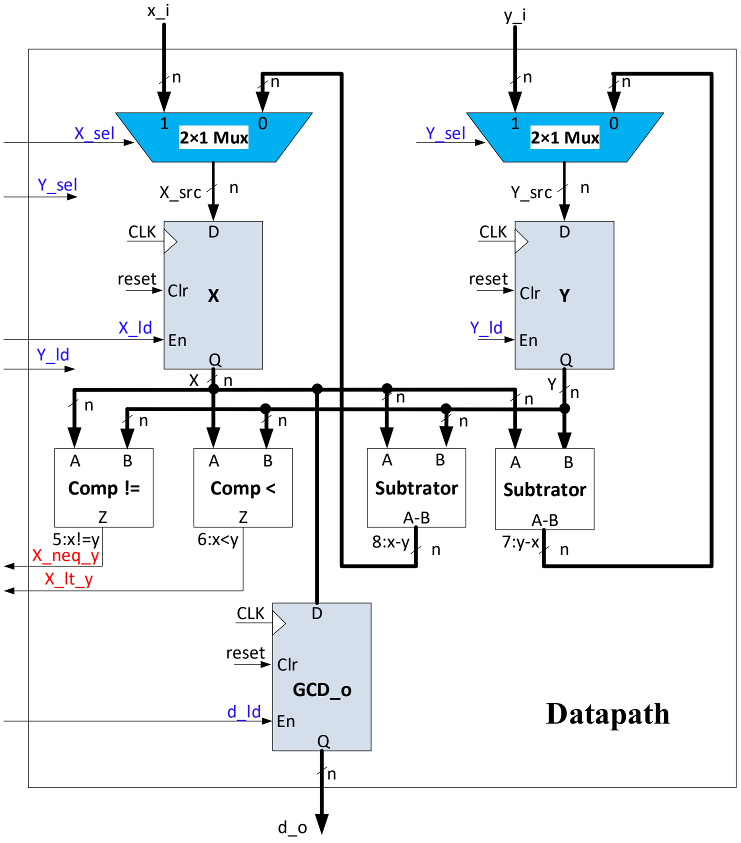
(c) state diagram FSM



(c) Controller



(d) Datapath

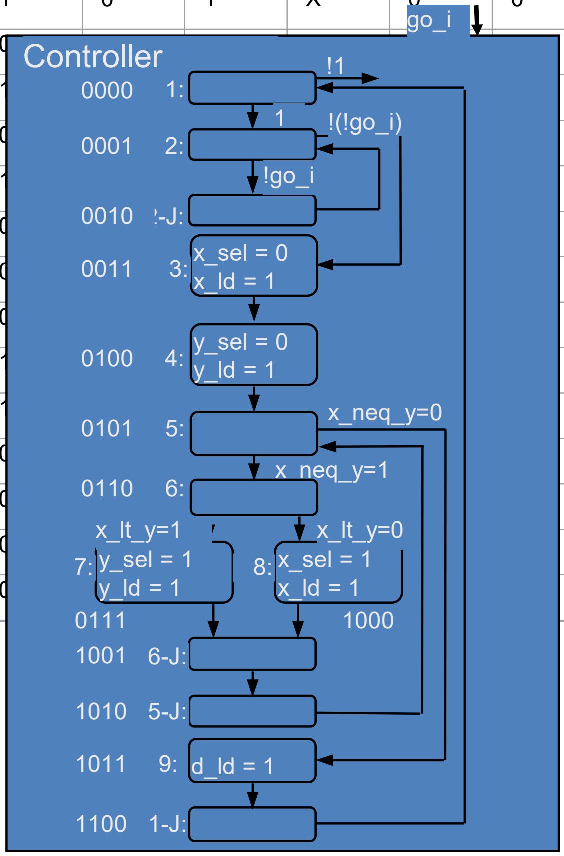


Controller state table for the GCD example

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	Start_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	GCD_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

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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
0	1	0	1	1	*	*	0	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
0	1	1	0	*	0	*	1	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
0	1	1	0	*	1	*	0	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
0	1	1	1	*	*	*	1	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	0
1	0	0	0	*	*	*	1	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	0	0	1	*	*	*	1	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	0	1	0	*	*	*	0	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	0	1	1	*	*	*	1	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	1
1	1	0	0	*	*	*	0	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	1	0	1	*	*	*	0	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	1	1	0	*	*	*	0	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
1	1	1	1	*	*	*	0	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0



x_ld default value = 0

y_ld default value = 0

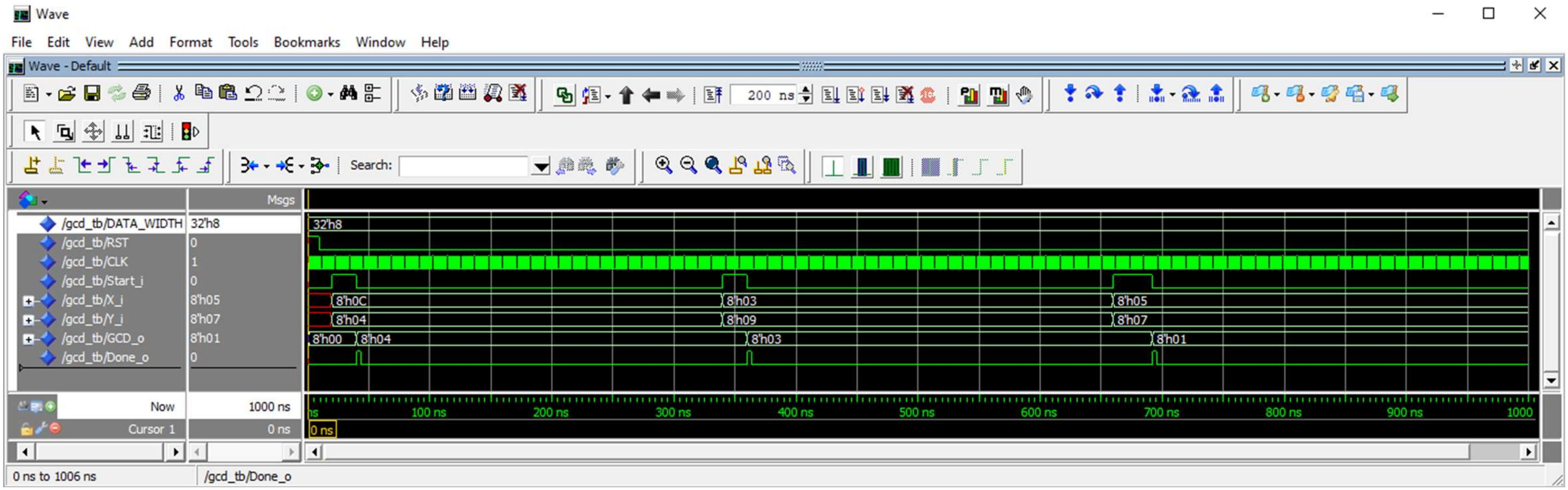
gcd_ld default value = 0

RTL Modeling GCD design in VHDL

- **Controller**: is behavior description of the FSM
- **Datapath:** structural description, including:
  - **Mux:** takes 2 4-bit inputs and one select line. Based on the select line, it outputs either the 1st 4-bit number or the 2nd 4-bit number.
  - **Register:** Takes a 4-bit input, a load signal, reset, and a clock signal. If the load signal is high and the clock is pulsed, it outputs the 4-bit number.
  - **Comparator:** Takes 2 4-bit numbers, and assets one of 3 signals depending on whether the 1st number is less than, greater than or equal to the 2nd number.
  - **Subtractor:** Takes 2 4-bit numbers, subtracts the smaller number from the larger.
  - **Output Register:** Holds the GCD value. When x = y the GCD has been found and can be outputted. Because it is a register entity it should also take a clock and reset signal.

Write a test bench and verify the correctness of GCD design

Simulation with ModelSIM

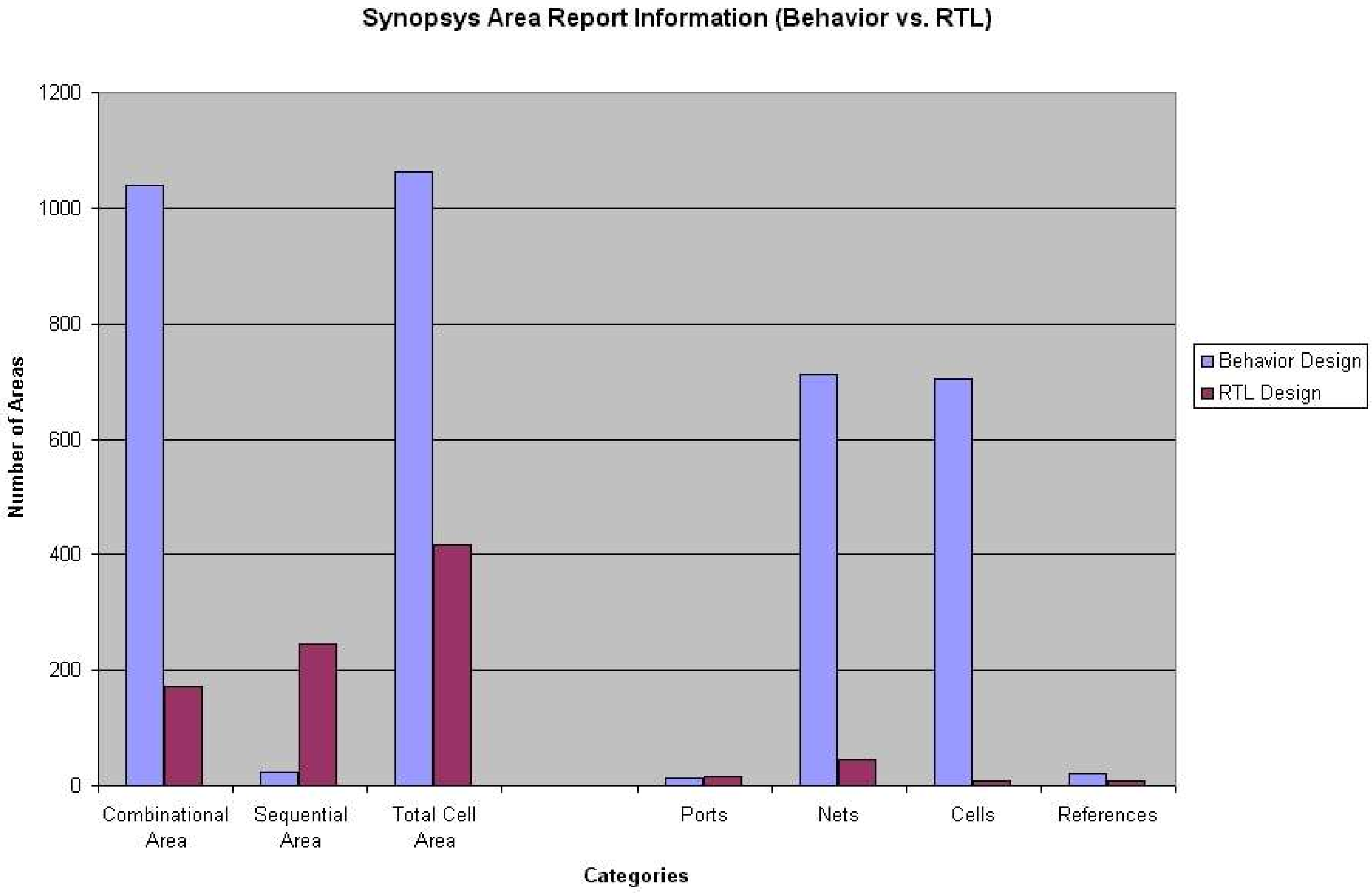


	Msgs
/gcd_tb/DATA_WIDTH	32'h8
/gcd_tb/RST	0
/gcd_tb/CLK	1
/gcd_tb/Start_i	0
/gcd_tb/X_i	8'h05
/gcd_tb/Y_i	8'h07
/gcd_tb/GCD_o	8'h01
/gcd_tb/Done_o	0
Now	1000 ns
Cursor 1	0 ns

0 ns to 1006 ns /gcd_tb/Done_o

Write a test bench and verify the correctness of GCD design

Synthesis with Synopsys DC tool

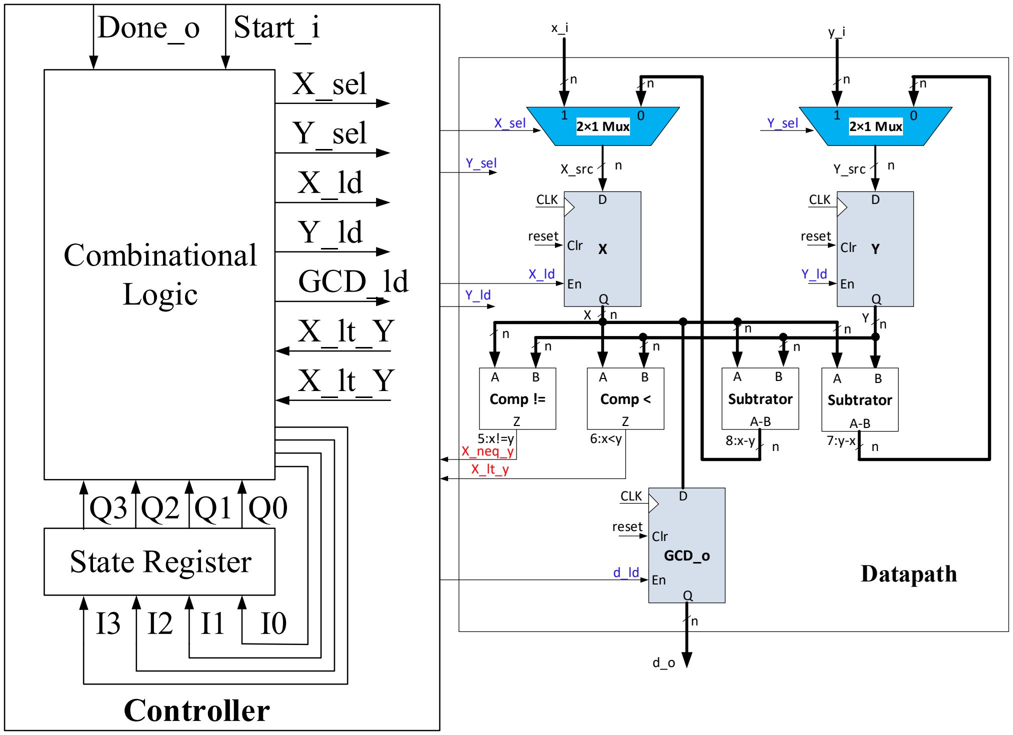


Outline

- RTL design methodology
- Review of Combinational and Sequential logic design
- Single-purpose vs. general-purpose architectures
- **RT-level custom single-purpose circuit design**
    - How to convert an algorithm into a single-purpose architecture;
    - How to reach a optimized design
- RT-level custom general-purpose processor design

Completing the GCD single-purpose hardware 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



Optimizing single-purpose hardware

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - Original program (algorithm)
  - 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')

Algorithm 1

```
int  x, y, r;
while (1) {
   while (!Start_i);
   // x must be the larger number
   if (x_i >= y_i)  {
      x=x_i;
      y=y_i;
   }
   else {
      x=y_i;
      y=x_i;
   }
   while  (y != 0)  {
      r = x % y;
      x = y;
      y = r;
   }
   GCD_o = x; Done_o = ‘1’;
}
```

Algorithm 2

```
int  x, y;
while (1) {
   while (!Start_i);
   x = x_i;
   y = y_i;
   while  (x != y)  {
      if  (x < y)
         y = y - x;
      else
         x = x - y;
   }
   GCD_o = x; Done_o = ‘1’;
}
```

Reduce the computation complexity

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

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

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

GCD(42, 8) - 9 iterations to complete the loop

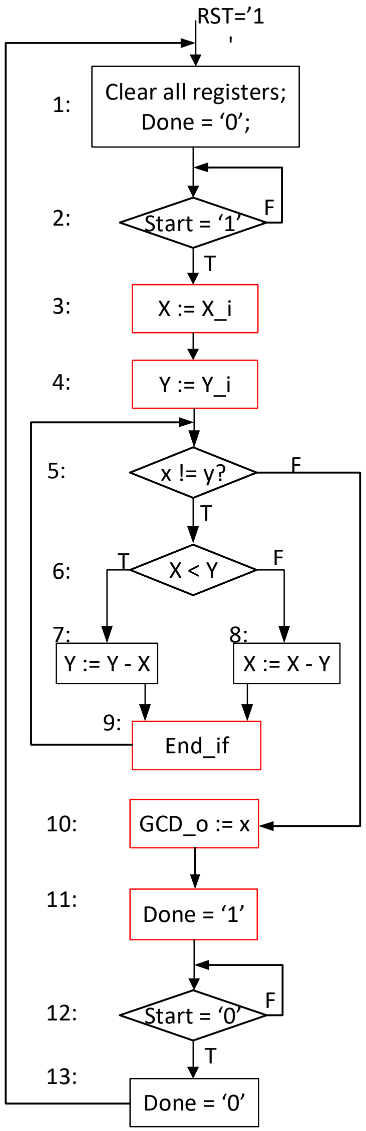
x and y values evaluated as follows : (42, 8), (34, 8), (26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

Optimizing the FSMD

- Areas of possible improvements
  - merge states
    - states with constants on transitions can be eliminated,
    - 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.)

original FSMD

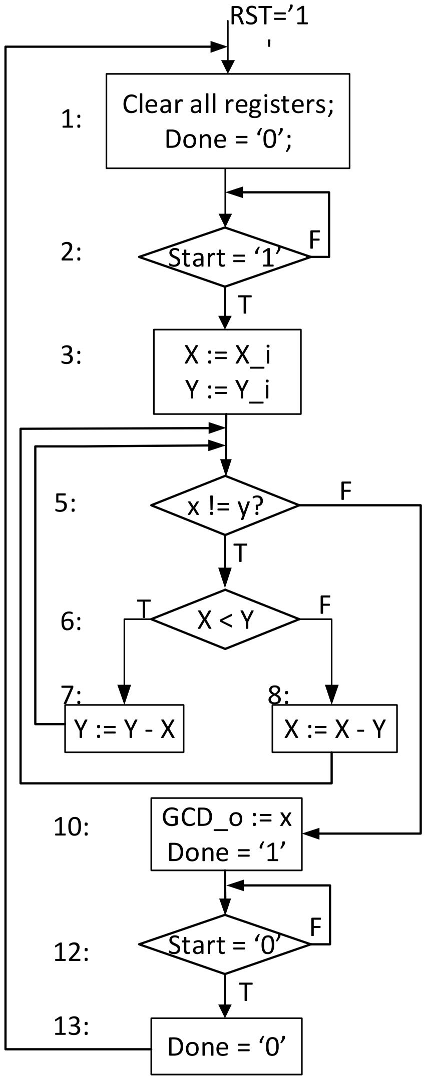


merge state 3 and state 4 – assignment operations are independent of one another

eliminate state 9 – transitions from state 9 can be done directly from state 7 and 8

merge state 10 and state 11 – assignment operations are independent of one another

optimized FSMD



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

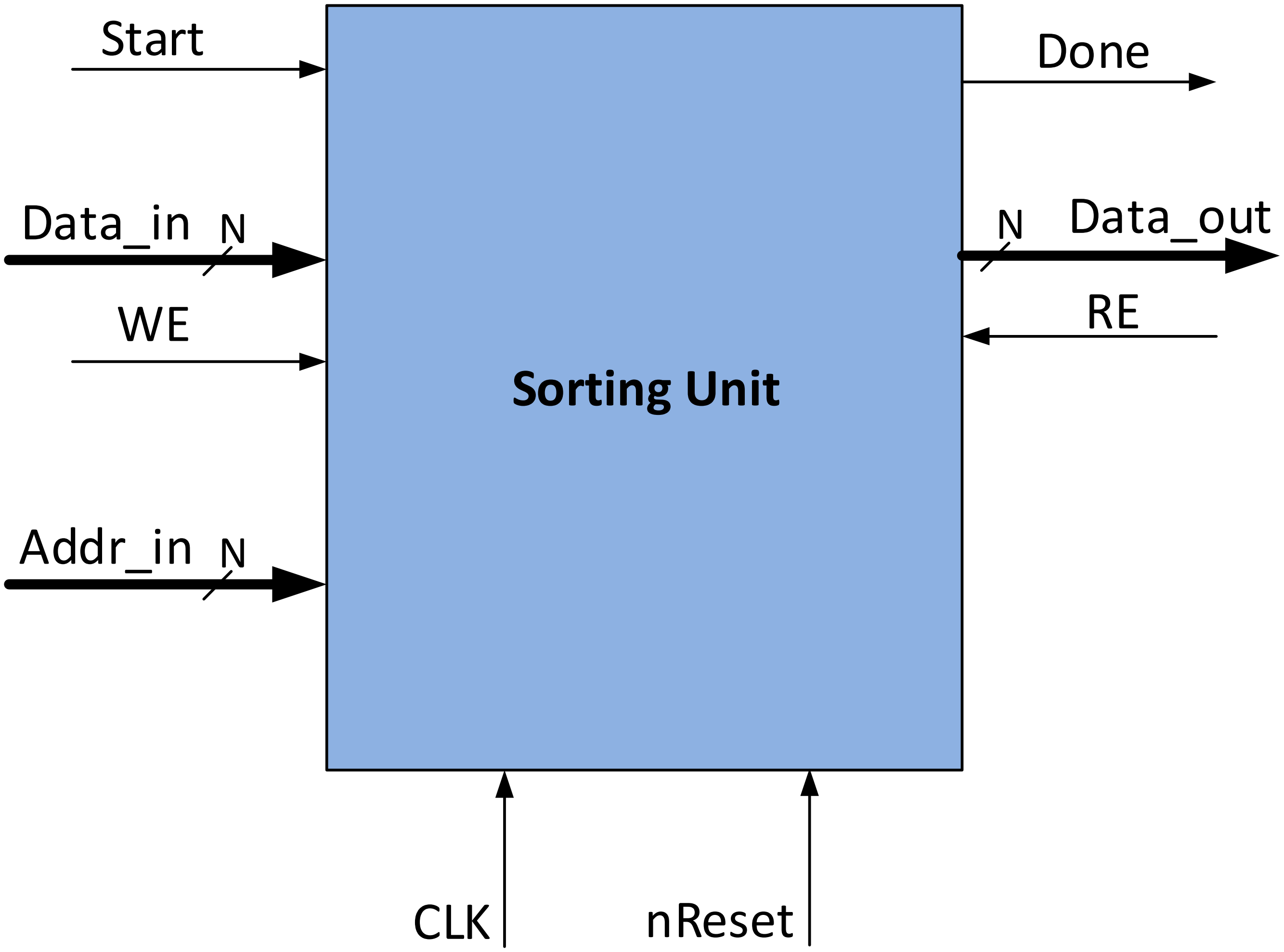
Optimizing the FSM

- 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

Mid-semester Project

Designing a DMA with sorting feature

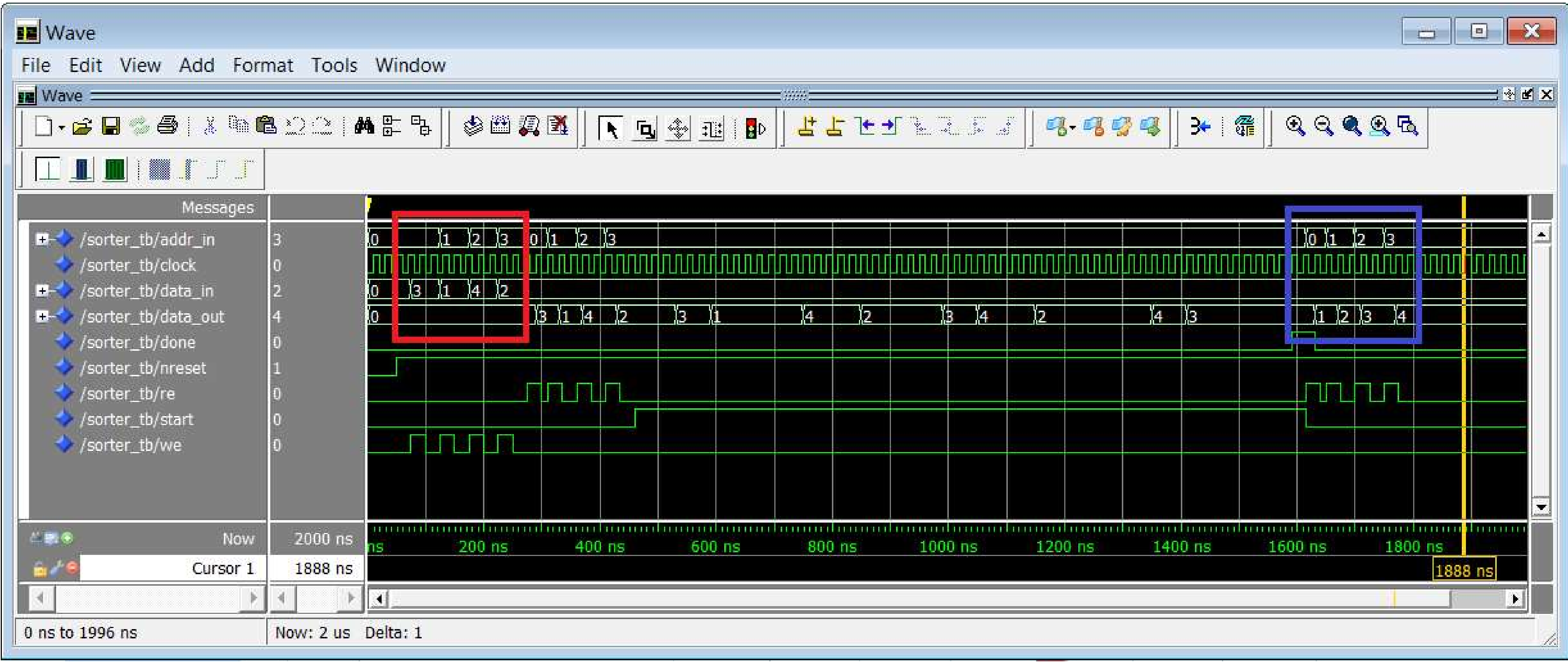
Sorting - Required Interface



Sorting - Required Interface

TT	Port	Direction	Width	Meaning
1	Clk	In	1	System clock
2	nReset	In	1	System Reset is used to clear all internal registers. Active low
3	Data_in	In	N	Input data bus
4	WE	In	1	Synchronous write enable signal
5	Data_out	Out	N	Output data bus
6	RE	In	1	Read Enable. - 0: high impedance on the Data_out bus - 1: valid output on the Data_out bus
7	Addr	In	$L=\log_2 K$	Address of the internal memory array
8	Start	In	1	Selecting operation mode: - 0: Idle or initialization - 1: Sorting
9	Done	Out	1	Asserted when sorting has been finished

Simulation results for the sort operation



Wave
File Edit View Add Format Tools Window
Wave

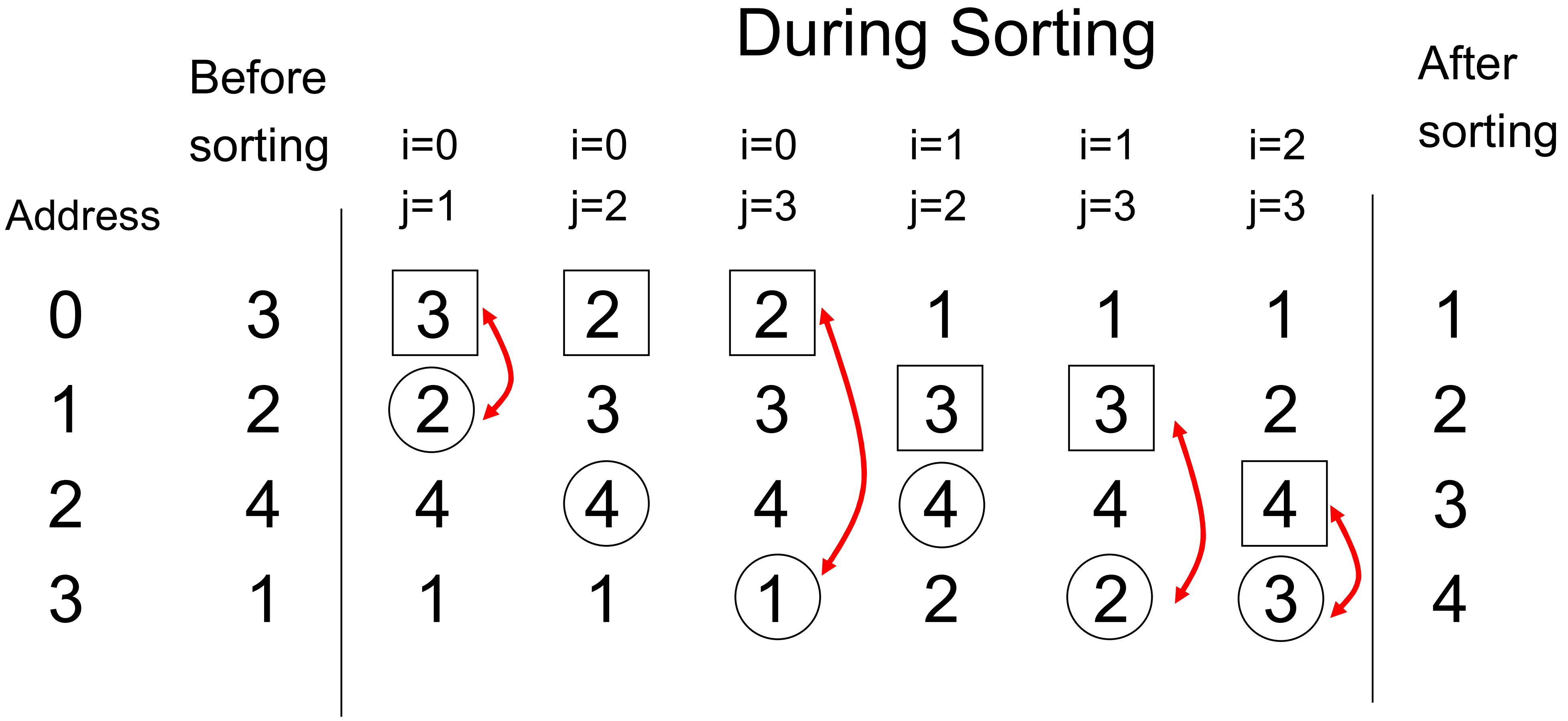
Messages	
/sorter_tb/addr_in	3
/sorter_tb/clock	0
/sorter_tb/data_in	2
/sorter_tb/data_out	4
/sorter_tb/done	0
/sorter_tb/nreset	1
/sorter_tb/re	0
/sorter_tb/start	0
/sorter_tb/we	0
Now	2000 ns
Cursor 1	1888 ns

0 1 2 3 0 1 2 3 0 1 2 3
0 3 1 4 2
0 3 1 4 2 3 1 4 2 3 4 2 4 3 1 2 3 4
ns 200 ns 400 ns 600 ns 800 ns 1000 ns 1200 ns 1400 ns 1600 ns 1800 ns
1888 ns

0 ns to 1996 ns

Now: 2 us Delta: 1

Sorting - Example



During Sorting

Address	Before sorting	i=0 j=1	i=0 j=2	i=0 j=3	i=1 j=2	i=1 j=3	i=2 j=3	After sorting
0	3	3	2	2	1	1	1	1
1	2	2	3	3	3	3	2	2
2	4	4	4	4	4	4	4	3
3	1	1	1	1	2	2	3	4

Legend:

position of memory indexed by i $\boxed{M_i}$

position of memory indexed by j M_j

Algorithm

FOR k = 4

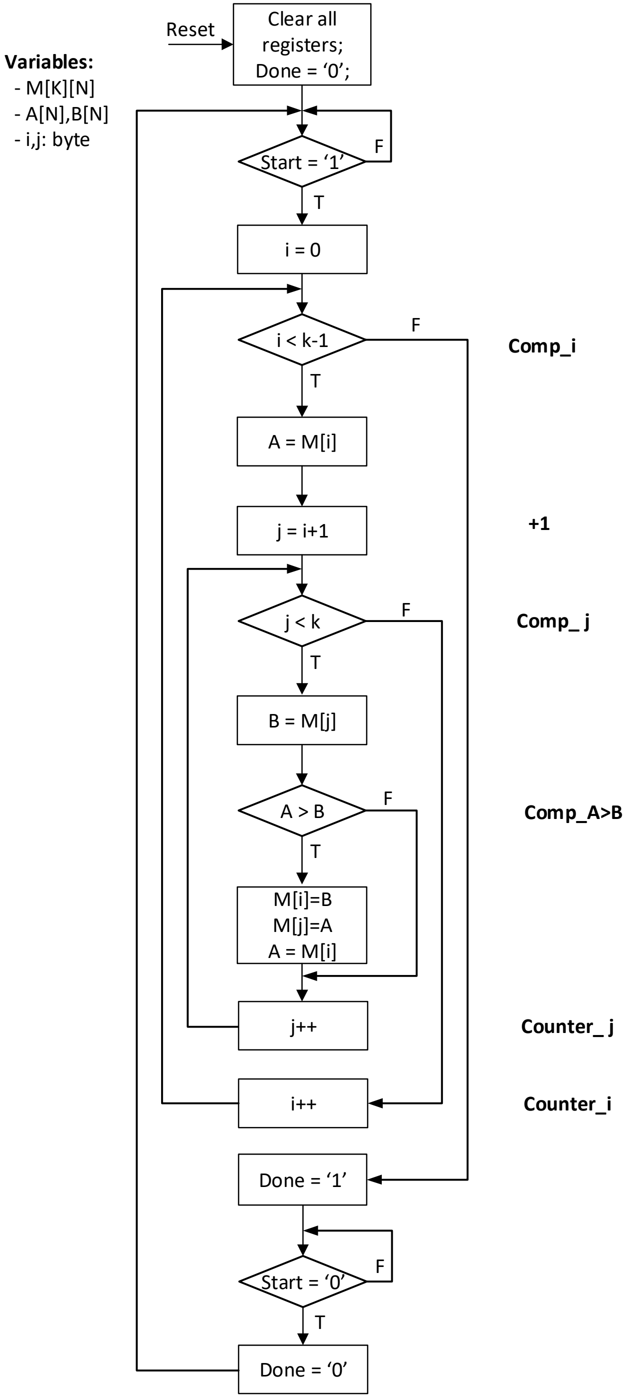
```
[load input data]
wait for s=1
for i = 0 to 2 do
    $A = M_i$ ;
    for j = i + 1 to 3 do
        $B = M_j$ ;
        if B < A then
            $M_i = B$ ;
            $M_j = A$ ;
            $A = M_i$ ;
        endif ;
    endfor;
endfor;
Done
wait for s=0
[read output data]
go to the beginning
```

FOR any k ≥ 2

```
[load input data]
wait for s=1
for i = 0 to k-2 do
    $A = M_i$ ;
    for j = i + 1 to k – 1 do
        $B = M_j$ ;
        if B < A then
            $M_i = B$ ;
            $M_j = A$ ;
            $A = M_i$ ;
        endif ;
    endfor;
endfor;
Done
wait for s=0
[read output data]
go to the beginning
```

Algorithm

```
wait for s=1
for i=0 to k-2 do
    $A = M_i$
    for j=i+1 to k-1 do
        $B = M_j$
        if A > B then
            $M_i = B$
            $M_j = A$
            $A = M_i$
        end if
    end for
end for
Done
wait for s=0
go to the beginning
```



Reset

Clear all registers; Done = ‘0’;

Variables:

- M[K][N]
- A[N],B[N]
- i,j: byte

Start = ‘1’

F

T

i = 0

i < k-1

F

T

Comp_i

A = M[i]

j = i+1

+1

j < k

F

T

Comp_ j

B = M[j]

A > B

F

T

Comp_A>B

M[i]=B
M[j]=A
A = M[i]

j++

Counter_ j

i++

Counter_i

Done = ‘1’

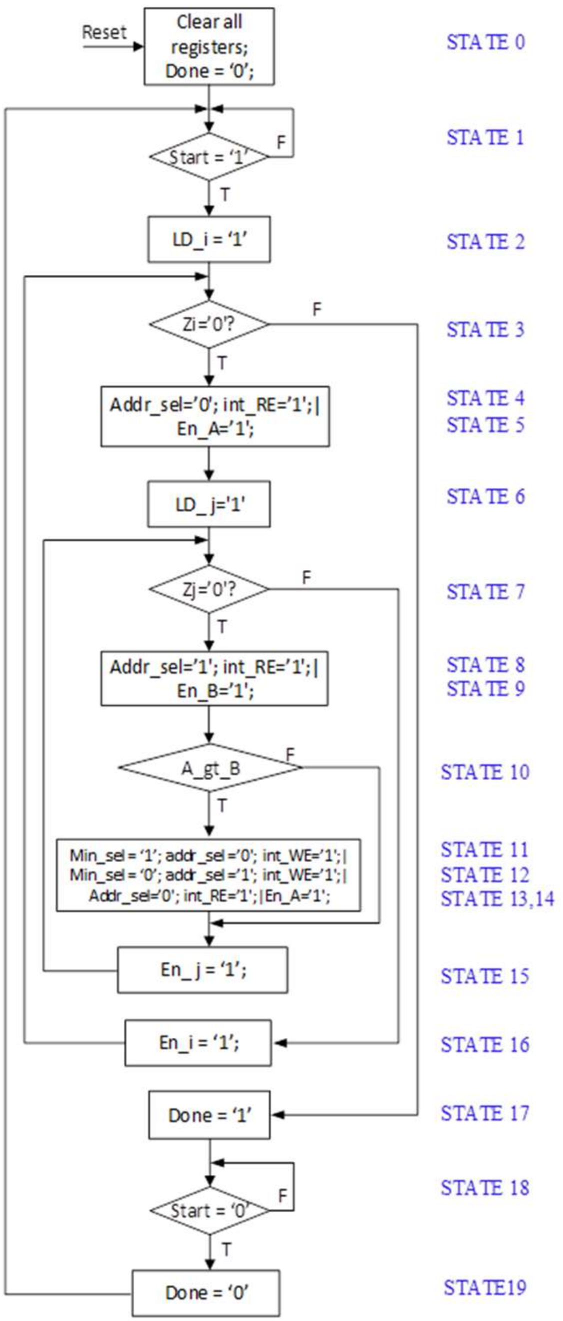
Start = ‘0’

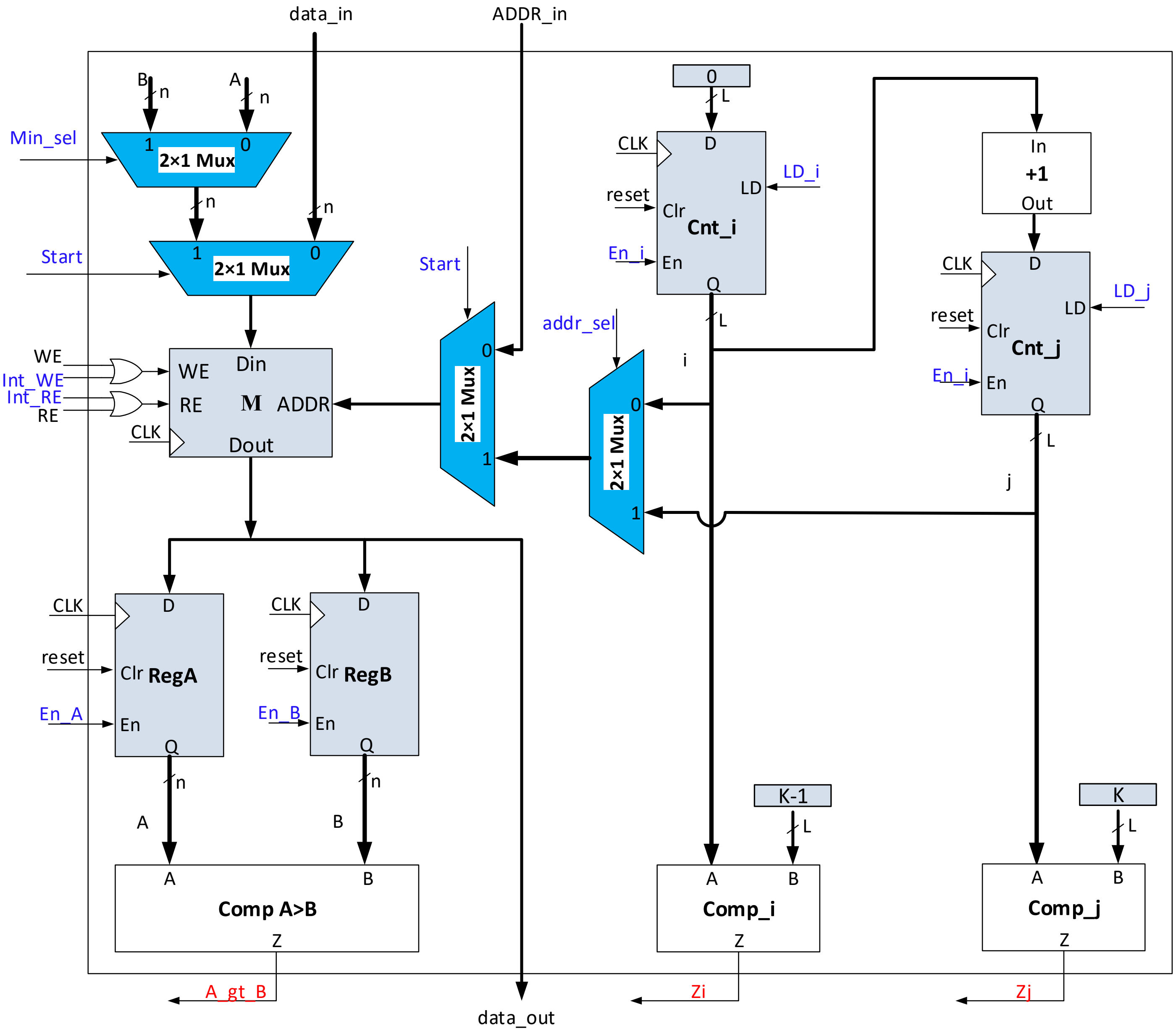
F

T

Done = ‘0’

FSM and Datapath





Summary

- ❖ Basic concepts on RTL design
- ❖ RTL design method
  - ✓ Basic technique for converting an algorithm into a single-purpose architecture
  - ✓ Optimization techniques
- ❖ Homework:
  - ✓ writing the VHDL code for the GCD design and the report of simulation and synthesis