Introduction to Digital Design

Week 15: Programmable Processors and HDL

Yao Zheng
Assistant Professor
University of Hawai'i at Mānoa
Department of Electrical Engineering

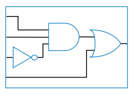
Overview

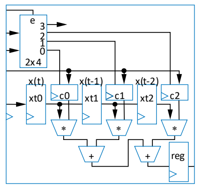
- Programmable processors are widely used
  - Easy availability, short design time
- Basic architecture
  - Datapath with register file and ALU
  - Control unit with PC, IR, and controller
  - Memories for instructions and data
  - Control unit fetches, decodes, and executes
- Three-instruction processor with machine-level programs
  - Extended to six instructions
  - Real processors have dozens or hundreds of instructions
  - Extended to access external pins
  - Modern processors are far more sophisticated
- Hardware Description Languages (HDLs)
  - VHDL, Verilog, and SystemC are popular
  - Introduced languages mainly through examples

8.1

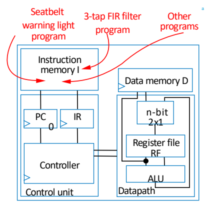
Programmable Processor

- Programmable (general-purpose) processor
  - Mass-produced, then programmed to implement different processing tasks
    - Well-known common programmable processors: Pentium, Sparc, PowerPC
    - Lesser-known but still common: ARM, MIPS, 8051, PIC, AVR
      - Low-cost embedded processors found in cell phones, blinking shoes, etc.
  - Instructive to design a very simple programmable processor
    - Real processors can be much more complex


Seatbelt warning light single-purpose processor



3-tap FIR filter single-purpose processor



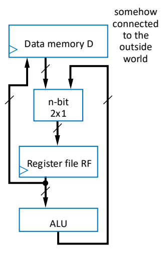
General-purpose processor

Note: Slides with animation are denoted with a small red "a" near the animated items

8.2

Basic Architecture

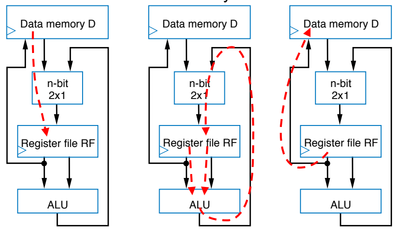
- Processing generally consists of:
  - Loading some data
  - Transforming that data
  - Storing that data
- *Basic datapath*: Useful circuit in a programmable processor
  - Can read/write data memory, where main data exists
  - Has register file to hold data locally
  - Has ALU to transform local data



Datapath

Basic Datapath Operations

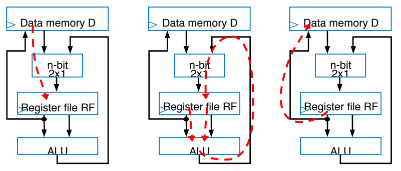
- Load operation: Load data from data memory to RF
- ALU operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- Store operation: Stores RF register value back into data memory
- Each operation can be done in one clock cycle



Load operation ALU operation Store operation

Basic Datapath Operations

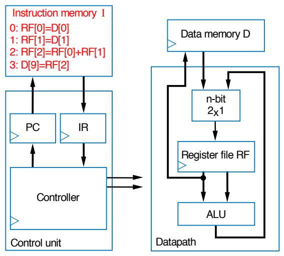
- Q: Which are valid *single-cycle operations* for given datapath?
  - Move D[1] to RF[1] (i.e., RF[1] = D[1])
    - A: YES – That's a load operation
  - Store RF[1] to D[9] and store RF[2] to D[10]
    - A: NO – Requires two separate store operations
  - Add D[0] plus D[1], store result in D[9]
    - A: NO – ALU operation (ADD) only works with RF. Requires two load operations (e.g., RF[0]=D[0]; RF[1]=D[1], an ALU operation (e.g., RF[2]=RF[0]+RF[1]), and a store operation (e.g., D[9]=RF[2])



Load operation ALU operation Store operation

Basic Architecture – Control Unit

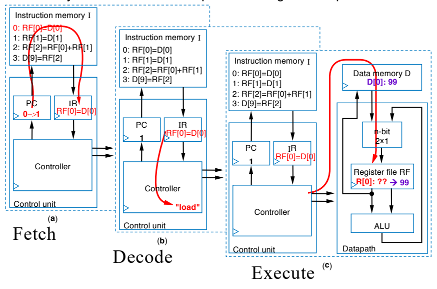
- D[9] = D[0] + D[1] – requires a sequence of four datapath operations:
  - 0: RF[0] = D[0]
  - 1: RF[1] = D[1]
  - 2: RF[2] = RF[0] + RF[1]
  - 3: D[9] = RF[2]
- Each operation is an *instruction*
  - Sequence of instructions – *program*
  - Looks cumbersome, but that's the world of programmable processors – Decomposing desired computations into processor-supported operations
  - Store program in *Instruction memory*
  - *Control unit* reads each instruction and executes it on the datapath
    - PC: Program counter – address of current instruction
    - IR: Instruction register – current instruction



7

Basic Architecture – Control Unit

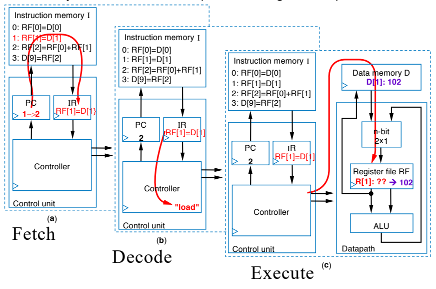
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



8

Basic Architecture – Control Unit

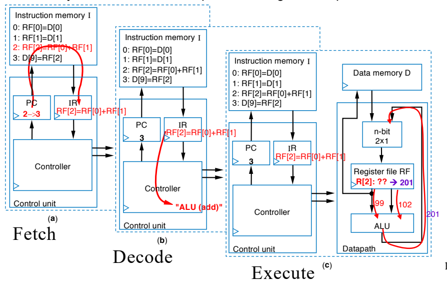
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



9

Basic Architecture – Control Unit

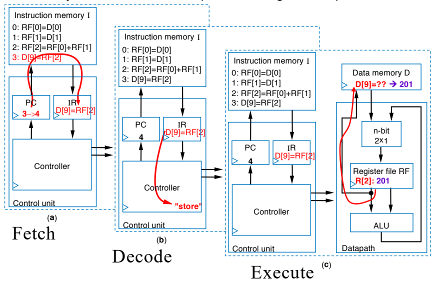
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



10

Basic Architecture – Control Unit

- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

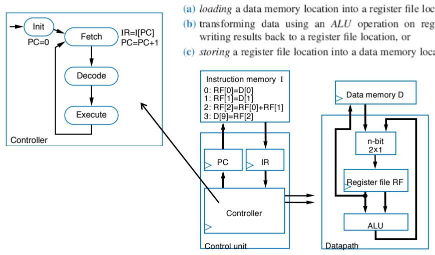


11

Basic Architecture – Control Unit

To summarize, the control unit processes each instruction in three stages:

1. first *fetching* the instruction by loading the current instruction into *IR* and incrementing the *PC* for the next fetch,
2. next *decoding* the instruction to determine its operation, and
3. finally *executing* the operation by setting the appropriate control lines for the datapath, if applicable. If the operation is a datapath operation, the operation may be one of three possible types:
   (a) *loading* a data memory location into a register file location,
   (b) transforming data using an *ALU* operation on register file locations and writing results back to a register file location, or
   (c) *storing* a register file location into a data memory location.



12

Creating a Sequence of Instructions

- Q: Create sequence of instructions to compute D[3] = D[0]+D[1]+D[2] on earlier-introduced processor
- A1: One possible sequence
  - First load data memory locations into register file
    - R[3] = D[0]
    - R[4] = D[1]
    - R[2] = D[2]

    (Note arbitrary register locations)
  - Next, perform the additions
    - R[1] = R[3] + R[4]
    - R[1] = R[1] + R[2]
  - Finally, store result
    - D[3] = R[1]
- A2: Alternative sequence
  - First load D[0] and D[1] and add them
    - R[1] = D[0]
    - R[2] = D[1]
    - R[1] = R[1] + R[2]
  - Next, load D[2] and add
    - R[2] = D[2]
    - R[1] = R[1] + R[2]
  - Finally, store result
    - D[3] = R[1]

13

Number of Cycles

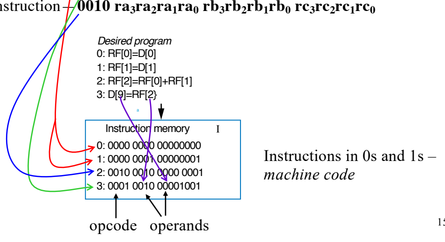
- Q: How many cycles are needed to execute six instructions using the earlier-described processor?
- A: Each instruction requires 3 cycles – 1 to fetch, 1 to decode, and 1 to execute
  - Thus, 6 instr * 3 cycles/instr = 18 cycles

14

8.3

Three-Instruction Programmable Processor

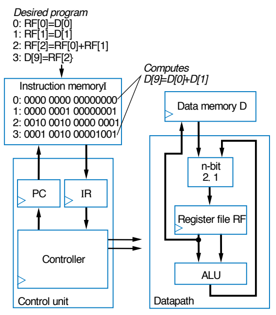
- Instruction Set – List of allowable instructions and their representation in memory, e.g.,
  - ***Load*** instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
  - ***Store*** instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
  - ***Add*** instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**



Instructions in 0s and 1s – *machine code*

15

Program for Three-Instruction Processor



16

Program for Three-Instruction Processor

- Another example program in machine code
  - Compute D[5] = D[5] + D[6] + D[7]

```
0: 0000 0000 00000101  // RF[0] = D[5]
1: 0000 0001 00000110  // RF[1] = D[6]
2: 0000 0010 00000111  // RF[2] = D[7]
3: 0010 0000 0000 0001  // RF[0] = RF[0] + RF[1]
                        // which is D[5]+D[6]
4: 0010 0000 0000 0010  // RF[0] = RF[0] + RF[2]
                        // now D[5]+D[6]+D[7]
5: 0001 0000 00000101  // D[5] = RF[0]
```

–***Load*** instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
–***Store*** instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
–***Add*** instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

17

Assembly Code

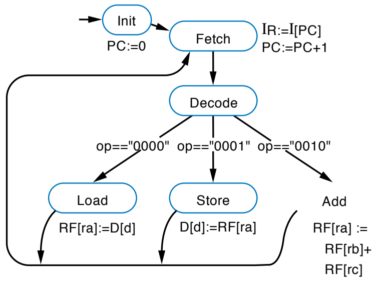
- Machine code (0s and 1s) hard to work with
- Assembly code – Uses mnemonics
  - ***Load*** instruction—**MOV Ra, d**
    - specifies the operation *RF[a]=D[d]*. *a* must be 0,1, ..., or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255
  - • ***Store*** instruction—**MOV d, Ra**
    - specifies the operation *D[d]=RF[a]*
  - • ***Add*** instruction—**ADD Ra, Rb, Rc**
    - specifies the operation *RF[a]=RF[b]+RF[c]*

Desired program		
0: RF[0]=D[0]	0: 0000 0000 00000000	0: MOV R0, 0
1: RF[1]=D[1]	1: 0000 0001 00000001	1: MOV R1, 1
2: RF[2]=RF[0]+RF[1]	2: 0010 0010 0000 0001	2: ADD R2, R0, R1
3: D[9]=RF[2]	3: 0001 0010 00001001	3: MOV 9, R2
	machine code	assembly code

18

Control-Unit and Datapath for Three-Instruction Processor

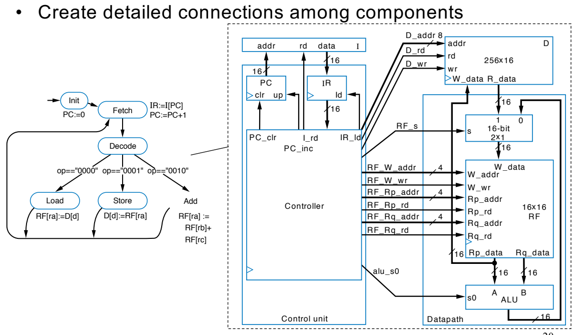
- To design the processor, we can begin with a high-level state machine description of the processor's behavior



19

Control-Unit and Datapath for Three-Instruction Processor

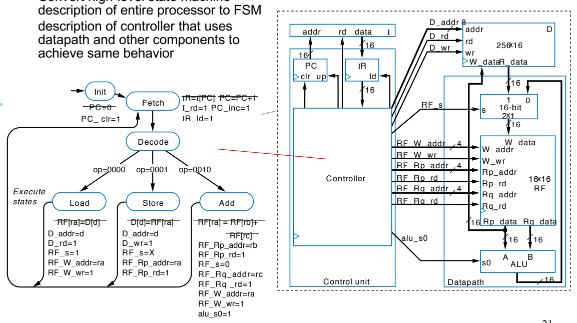
- Create detailed connections among components



20

Control-Unit and Datapath for Three-Instruction Processor

- Convert high-level state machine description of entire processor to FSM description of controller that uses datapath and other components to achieve same behavior



21

8.4

A Six-Instruction Programmable Processor

- Let's add three more instructions:
  - *Load-constant* instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**
    - **MOV Ra, #c**—specifies the operation *RF[a]=c*
  - *Subtract* instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**
    - **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*
  - *Jump-if-zero* instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**
    - **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

TABLE 8.1 Six-instruction instruction set..

Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

22

Extending the Control-Unit and Datapath

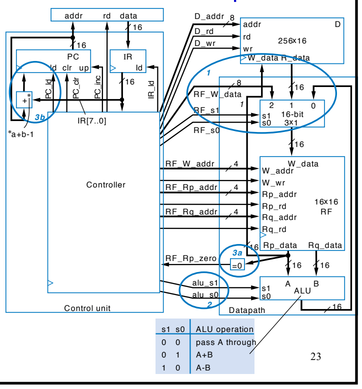
1: The *load constant* instruction requires that the register file be able to load data from *IR[7..0]*, in addition to data from data memory or the ALU output. Thus, we widen the register file's multiplexer from 2x1 to 3x1, add another mux control signal, and also create a new signal coming from the controller labeled *RF_W_data*, which will connect with *IR[7..0]*.

2: The subtract instruction requires that we use an ALU capable of subtraction, so we add another ALU control signal.

3: The jump-if-zero instruction requires that we be able to detect if a register is zero, and that we be able to add *IR[7..0]* to the *PC*.

3a: We insert a datapath component to detect if the register file's *Rp* read port is all zeros (that component would just be a NOR gate).

3b: We also upgrade the *PC* register so it can be loaded with *PC* plus *IR[7..0]*. The adder used for this also subtracts 1 from the sum, to compensate for the fact that the *Fetch* state already added 1 to the *PC*.



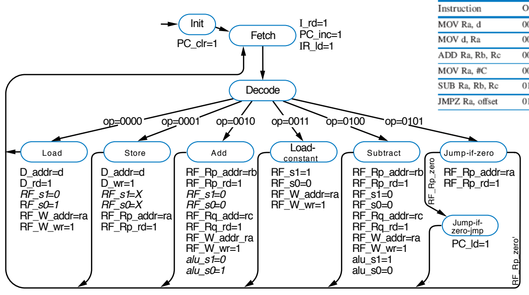
s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

23

Controller FSM for the Six-Instruction Processor

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101



24

Program for the Six-Instruction Processor

- Example program – Count number of non-zero words in D[4] and D[5]
  - Result will be either 0, 1, or 2
  - Put result in D[9]

```
     MOV R0, #0; // initialize result to 0                  0011 0000 00000000
     MOV R1, #1; // constant 1 for incrementing result      0011 0001 00000001
     MOV R2, 4; // get data memory location 4               0000 0010 00000100
     JMPZ R2, lab1; // if zero, skip next instruction       0101 0010 00000010
     ADD R0, R0, R1; // not zero, so increment result       0010 0000 0000 0001
lab1:MOV R2, 5; // get data memory location 5               0000 0010 00000101
     JMPZ R2, lab2; // if zero, skip next instruction       0101 0010 00000010
     ADD R0, R0, R1; //not zero, so increment result        0010 0000 0000 0001
lab2:MOV 9, R0; // store result in data memory location 9   0001 0000 00001001
```

(a) (b)

TABLE 8.2 Instruction opcodes.

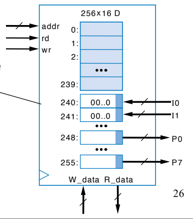
Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

25

8.5

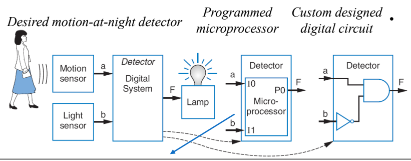
Further Extensions to the Programmable Processor

- Typical processor instruction set will contain dozens of data movement (e.g., loads, stores), ALU (e.g., add, sub), and flow-of-control (e.g., jump) instructions
  - Extending the control-unit/datapath follows similarly to previously-shown extensions
- Input/output extensions
  - Certain memory locations may actually be external pins
    - e.g, D[240] may represent 8-bit input I0, D[255] may represent 8-bit output P7

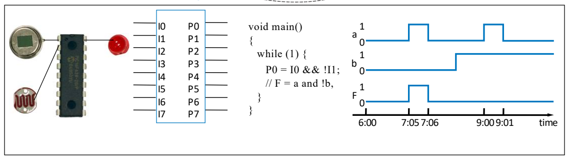


26

Program using I/O Extensions – Recall Chpt 1 C-Program Example


Desired motion-at-night detector
Programmed microprocessor
Custom designed digital circuit
Motion sensor
Light sensor
Detector
Digital System
Lamp
I0
P0
Micro-processor
I1
a
b
F

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



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

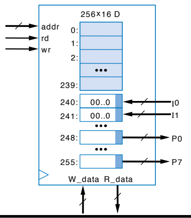
27

Program Using Input/Output Extensions

Underlying assembly code for C expression I0 && !I1.

0: MOV R0, 240 // move *D[240]*, which is the value at pin *I0*, into *R0*
1: MOV R1, 241 // move *D[241]*, which is that value at pin *I1*, into *R1*
2: NOT R1, R1 // compute *!I1*, assuming existence of a complement instruction
3: AND R0, R0, R1 // compute *I0* && *!I1*, assuming an AND instruction
4: MOV 248, R0 // move result to *D[248]*, which is pin *P0*

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

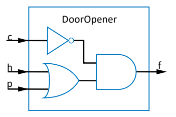


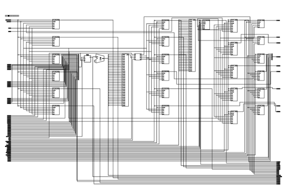
28

9.1

Hardware description language

- A drawing of a circuit, or *schematic*, contains graphical information about a design
  - Inverter is above the OR gate, AND gate is to the right, etc.
- Such graphical information may not be useful for large designs
- Can use textual language instead





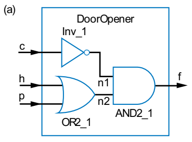
29

Note: Slides with animation are denoted with a small red "a" near the animated items

Textual Language – English

- Can describe circuit using English text rather than using a drawing
  - Of course, English isn't a good language for a computer to read
  - Need a more precise, computer-oriented language

(a)



(b)

We'll now describe a circuit whose name is DoorOpener.
  The external inputs are c, h and p, which are bits.
  The external output is f, which is a bit.

We assume you know the behavior of these components:
  An inverter, which has a bit input x, and bit output F.
  A 2-input ORgate, which has inputs x and y, and bit output F.
  A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.
The DoorOpener circuit internally consists of:
  An inverter named Inv_1, whose input x connects to
    external input c, and whose output connects to n1.
  A 2-input ORgate named OR2_1, whose inputs connect to external
    inputs h and p, and whose output connects to n2.
  A 2-input AND gate named AND2_1, whose inputs connect to n1
    and n2, and whose output connects to external output f.
That's all.

30

Computer-Readable Textual Language for Describing Hardware Circuits: HDLs

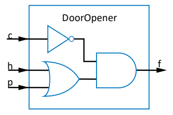
- Hardware description language (HDL)
  - Intended to describe circuits textually, for a computer to read
  - Evolved starting in the 1970s and 1980s
- Popular languages today include:
  - VHDL –Defined in 1980s by U.S. military; Ada-like language
  - Verilog –Defined in 1980s by a company; C-like language
  - SystemC –Defined in 2000s by several companies; consists of libraries in C++

31

Combinational Logic Description using Hardware Description Languages

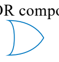
9.2

- **Structure**
  - Another word for "circuit"
  - An interconnection of components
  - Key use of HDLs is to describe structure

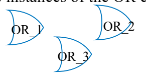


Note: The term "instantiate" will be used to indicate adding a new copy of a component to a circuit

The OR component



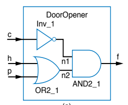
Three instances of the OR component



32

Describing Structure in VHDL

- Entity – Defines new item's name & ports (inputs/outputs)
  - std_logic means bit type, defined in ieee library
- Architecture – Describes internals, which we named "Circuit"
  - Declares 3 previously-defined components
  - Declares internal signals
    - Note "--" comment
  - Instantiates and connects those components



(a)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

(b)

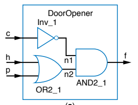
```
library ieee;
use ieee.std_logic_1164.all;
entity DoorOpener is
  port ( c, h, p: in std_logic;
         f: out std_logic
  );
end DoorOpener;

architecture Circuit of DoorOpener is
  component Inv
    port (x: in std_logic;
          F: out std_logic);
  end component;
  component OR2
    port (x,y: in std_logic;
          F: out std_logic);
  end component;
  component AND2
    port (x,y: in std_logic;
          F: out std_logic);
  end component;
  signal n1,n2: std_logic; --internal wires

begin
  Inv_1: Inv port map (x=>c, F=>n1);
  OR2_1: OR2 port map (x=>h,y=>p,F=>n2);
  AND2_1: AND2 port map (x=>n1,y=>n2,F=>f);
end Circuit;
```

(c)

33

Describing Structure in Verilog

- Modules defined for Inv, OR2, and AND2 (details omitted)
  - Note "//" comment
- Module defined for DoorOpener
  - Lists inputs and outputs
  - Declares internal wires
  - Instantiates and connects three components



(a)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

(b)

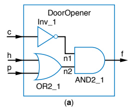
```
module Inv(x, F);
   input x;
   output F;
   // details not shown
endmodule
module OR2(x, y, F);
   input x, y;
   output F;
   // details not shown
endmodule
module AND2(x, y, F);
   input x, y;
   output F;
   // details not shown
endmodule

module DoorOpener(c, h, p, f);
   input c, h, p;
   output f;
   wire n1, n2;
   Inv Inv_1(c, n1);
   OR2 OR2_1(h, p, n2);
   AND2 AND2_1(n1, n2, f);
endmodule
```

(c)

34

Describing Structure in SystemC

- Module defined
  - Declares inputs and outputs
  - Declares internal wires
    - Note "//" comment
  - Declares three previously-defined components
  - Constructor function "CTOR"
    - Instantiates components
    - Connects components



(a)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

(b)

```
#include "systemc.h"
#include "inv.h"
#include "or2.h"
#include "and2.h"

SC_MODULE (DoorOpener)
{
   sc_in<sc_logic> c, h, p;
   sc_out<sc_logic> f;
   // internal wires
   sc_signal<sc_logic> n1, n2;
   // component declarations
   Inv Inv1;
   OR2 OR2_1;
   AND AND2_1;
   // component instantiations
   SC_CTOR(DoorOpener):Inv_1("Inv_1"),
      OR2_1("OR2_1"), AND2_1("AND2_1")
   {
      Inv_1.x(c);
      Inv_1.F(n1);
      OR2_1.x(h);
      OR2_1.y(p);
      OR2_1.F(n2);
      AND2_1.x(n1);
      AND2_1.y(n2);
      AND2_1.F(f);
   }
};
```

(c)

35

Combinational Behavior

- **Combinational behavior**
  - Description of desired behavior of combinational circuit without creating circuit itself
  - e.g., F = c' * (h + p) can be described as equation rather than circuit
  - HDLs support description of combinational behavior

36

Describing Combinational Behavior in VHDL

- Describing an OR gate's behavior
  - Entity defines input/output ports
  - Architecture
    - Process – Describes behavior
      - Process "sensitive" to x and y
        - » Means behavior only executes when x changes or y changes
      - Behavior assigns a new value to output port F, computed using built-in operator "or"

```
library ieee;
use ieee.std_logic_1164.all;

entity OR2 is
  port (x, y: in std_logic;
        F: out std_logic
  );
end OR2;

architecture behavior of OR2 is
begin
  process (x, y)
  begin
    F <= x or y;
  end process;
end behavior;
```

37

Describing Combinational Behavior in VHDL

- Describing a custom function's behavior
  - Desired function: f = c'*(h+p)
  - Entity defines input/output ports (not shown)
  - Architecture
    - Process
      - Sensitive to c, h, and p
      - Assigns a new value to output port f, computed using built-in operators "not", "and", and "or"

```
architecture beh of DoorOpener is
begin
  process(c, h, p)
  begin
    f <= not(c) and (h or p);
  end process;
end beh;
```

38

Describing Combinational Behavior in Verilog

- Describing an OR gate's behavior
  - Module declares input/output ports
    - Also indicates that F is "reg"
    - Means F stores value
      - By default, ports are wires, having no storage
  - "always" procedure executes statement block when change occurs on x or on y
    - "Sensitive" to x and y
    - Assigns value to F, computed using built-in OR operator "|"

```
module OR2(x,y,F);
  input x, y;
  output F;
  reg F;

  always @(x or y)
  begin
    F <= x | y;
  end
endmodule
```

39

Describing Combinational Behavior in Verilog

- Describing a custom function's behavior
  - Desired function: f = c'*(h+p)
  - Module defines input/output ports
    - Output f defined as "reg"
  - "always" procedure sensitive to inputs
    - Assigns value to f, computed using built-in operators for NOT (~), AND (&), and OR (|)

```
module DoorOpener(c,h,p,f);
  input c, h, p;
  output f;
  reg f;

  always @(c or h or p)
  begin
    f <= (~c) & (h | p);
  end
endmodule
```

40

Describing Combinational Behavior in SystemC

- Describing an OR gate's behavior
  - Module declares input/output ports
  - Constructor (CTOR)
    - Indicates module described by a method (procedure) "comblogic"
    - Sensitive to x and y
  - Method "comblogic" assigns F a new value using built-in OR operator "|"
    - Reading input port done using .read() function defined for input port type; likewise, writing done using .write() function

```
#include "systemc.h"

SC_MODULE(OR2)
{
  sc_in<sc_logic> x, y;
  sc_out<sc_logic> F;

  SC_CTOR(OR2)
  {
    SC_METHOD(comblogic);
    sensitive << x << y;
  }

  void comblogic()
  {
    F.write(x.read() | y.read());
  }
};
```

41

Describing Combinational Behavior in SystemC

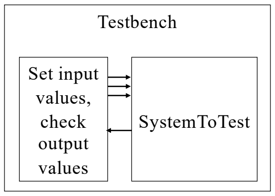
- Describing a custom function's behavior
  - Desired function: f = c'*(h+p)
  - Module defines input/output ports
  - Constructor
    - Indicates module described by a method (procedure) "comblogic"
    - Sensitive to c, h, and p
  - "comblogic" method
    - Assigns value to f, computed using built-in operators for NOT (~), AND (&), and OR (|)

```
#include "systemc.h"

SC_MODULE(DoorOpener)
{
  sc_in<sc_logic> c, h, p;
  sc_out<sc_logic> f;

  SC_CTOR(DoorOpener)
  {
    SC_METHOD(comblogic);
    sensitive << c << h << p;
  }

  void comblogic()
  {
    f.write((~c.read()) & (h.read() |
            p.read()));
  }
};
```

42

Testbenches

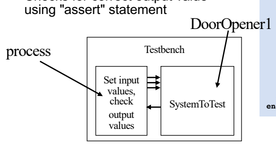
- **Testbench**
  - Assigns values to a system's inputs, check that system outputs correct values
  - A key use of HDLs is to simulate system to ensure design is correct



43

Testbench in VHDL

- Entity
  - No inputs or outputs
- Architecture
  - Declares component to test, declares signals
  - Instantiates component, connects to signals
  - Process writes input signals, checks output signal
    - Waits a small amount of time after writing input signals
    - Checks for correct output value using "assert" statement

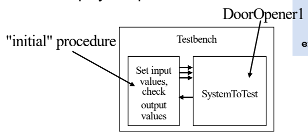


```
library ieee;
use ieee.std_logic_1164.all;

entity Testbench is
end Testbench;

architecture behavior of Testbench is
  component DoorOpener
    port ( c, h, p: in std_logic;
           f: out std_logic
    );
  end component;
  signal c, h, p, f: std_logic;
begin
  DoorOpener1: DoorOpener port map (c,h,p,f);

  process
  begin
    -- case 0
    c <= '0'; h <= '0'; p <= '0';
    wait for 1 ns;
    assert (f='0') report "Case 0 failed";

    -- case 1
    c <= '0'; h <= '0'; p <= '1';
    wait for 1 ns;
    assert (f='1') report "Case 1 failed";
    -- (cases 2-6 omitted from figure)
    -- case 7
    c <= '1'; h <= '1'; p <= '1';
    wait for 1 ns;
    assert (f='0') report "Case 7 failed";

    wait; -- process does not wake up again
  end process;
end behavior;
```

44

Testbench in Verilog

- Module
  - Three register signals for inputs (values must be written and stored)
  - One wire signal for output (value is only read, need not be stored)
  - Instantiates component, connects to signals
  - "initial" procedure executed only once at beginning
    - Sets input values
    - Displays output value

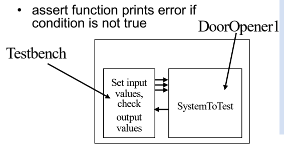


```
module Testbench;
  reg c, h, p;
  wire f;

  DoorOpener DoorOpener1(c, h, p, f);

  initial
  begin
    // case 0
    c <= 0; h <= 0; p <= 0;
    #1 $display("f = %b", f);
    // case 1
    c <= 0; h <= 0; p <= 1;
    #1 $display("f = %b", f);
    // (cases 2-6 omitted from figure)
    // case 7
    c <= 1; h <= 1; p <= 1;
    #1 $display("f = %b", f);
  end
endmodule
```

45

Testbench in SystemC

- Module
  - Testbench is its own module
  - Three outputs, one for each input of system to test
  - One input, for the one output of the system to test
  - Constructor defined as THREAD
    - Like MODULE, but allows use of "wait" to control timing
  - testbench procedure writes outputs, waits for small amount of time, checks for correct output
    - assert function prints error if condition is not true



```
#include "systemc.h"

SC_MODULE(Testbench)
{
  sc_out<sc_logic> c_t, h_t, p_t;
  sc_in<sc_logic> f_t;

  SC_CTOR(Testbench)
  {
    SC_THREAD(testbench_proc);
  }

  void testbench_proc()
  {
    // case 0
    c_t.write(SC_LOGIC_0);
    h_t.write(SC_LOGIC_0);
    p_t.write(SC_LOGIC_0);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_0 );

    // case 1
    c_t.write(SC_LOGIC_0);
    h_t.write(SC_LOGIC_0);
    p_t.write(SC_LOGIC_1);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_1 );

    // (cases 2-6 omitted from figure)
    // case 7
    c_t.write(SC_LOGIC_1);
    h_t.write(SC_LOGIC_1);
    p_t.write(SC_LOGIC_1);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_0 );

    sc_stop();
  }
};
```

46

Sequential Logic Description using Hardware Description Languages

9.3

- Will consider description of three sequential components
  - Registers
  - Oscillators
  - Controllers

47

Describing a 4-bit Register in VHDL

- Entity
  - 4 data inputs, 4 data outputs, and a clock input
  - Use std_logic_vector for 4-bit data
    - I: in std_logic_vector(3 downto 0)
    - I <= "1000" would assign I(3)=1, I(2)=0, I(1)=0, I(0)=0
- Architecture
  - Process sensitive to clock input
    - First statement detects if change on clock was a rising edge
    - If clock change was rising edge, sets output Q to input I
    - Ports are signals, and signals store values – thus, output retains new value until set to another value

```
library ieee;
use ieee.std_logic_1164.all;

entity Reg4 is
    port ( I: in std_logic_vector(3 downto 0);
           Q: out std_logic_vector(3 downto 0);
           clk: in std_logic
    );
end Reg4;

architecture behavior of Reg4 is
begin
    process(clk)
    begin
       if (clk='1' and clk'event) then
        Q <= I;
       end if;
    end process;
end behavior;
```

48

Describing a 4-bit Register in Verilog

- Module
  - 4 data inputs, 4 data outputs, and a clock input
  - Define data inputs/outputs as vectors
    - input [3:0] I
    - I<=4'b1000 assigns I[3]=1, I[2]=0, I[1]=0, I[0]=0
    - Output defined as register to store value
  - "always" procedure sensitive to positive (rising) edge of clock
    - Sets output Q to input I

```
module Reg4(I, Q, clk);
  input [3:0] I;
  input clk;
  output [3:0] Q;
  reg [3:0] Q;

  always @(posedge clk)
  begin
    Q <= I;
  end
endmodule
```

49

Describing a 4-bit Register in SystemC

- Module
  - 4 data inputs, 4 data outputs, and a clock input
  - Define data inputs/outputs as vectors
    - sc_in<sc_lv<4> > I;
    - I<="1000" assigns I[3]=1, I[2]=0, I[1]=0, I[0]=0
  - Constructor calls seq_logic method, sensitive to positive (rising) edge of clock
    - seq_logic writes output Q with input I
    - Output port is signal, and signal has storage, thus output retains value

```
#include "systemc.h"

SC_MODULE(Reg4)
{
  sc_in<sc_lv<4> > I;
  sc_out<sc_lv<4> > Q;
  sc_in<sc_logic> clk;

  SC_CTOR(Reg4)
  {
    SC_METHOD(seq_logic);
    sensitive_pos << clk;
  }

  void seq_logic()
  {
    Q.write(I.read());
  }
};
```

50

Describing an Oscillator in VHDL

- Entity
  - Defines clock output
- Architecture
  - Process
    - Has no sensitivity list, so executes non-stop as infinite loop
    - Sets clock to 0, waits 10 ns, sets clock to 1, waits 10 ns, repeats

```
library ieee;
use ieee.std_logic_1164.all;

entity Osc is
  port ( clk: out std_logic );
end Osc;

architecture behavior of Osc is
begin
  process
  begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;
end behavior;
```

51

Describing an Oscillator in Verilog

- Module
  - Has one output, clk
    - Declare as "reg" to hold value
  - "always" procedure
    - Has no sensitivity list, so executes non-stop as infinite loop
    - Sets clock to 0, waits for 10 ns, sets clock to 1, waits for 10 ns, repeats

```
module Osc(clk);
  output clk;
  reg clk;

  always
  begin
    clk <= 0;
    #10;
    clk <= 1;
    #10;
  end
endmodule
```

52

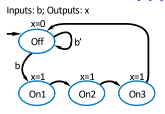
Describing an Oscillator in SystemC

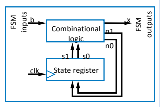
- Module
  - Has one output, clk
  - Constructor creates single thread
    - Thread consists of infinite loop – while (true) {
    - Sets clock to 0, waits 10 ns, sets clock to 1, waits 10 ns, repeats

```
#include "systemc.h"

SC_MODULE(Osc)
{
  sc_out<sc_logic> clk;

  SC_CTOR(Osc)
  {
    SC_THREAD(seq_logic);
  }

  void seq_logic()
  {
    while(true) {
      clk.write(SC_LOGIC_0);
      wait(10, SC_NS);
      clk.write(SC_LOGIC_1);
      wait(10, SC_NS);
    }
  }
};
```

53

Describing a Controller in VHDL





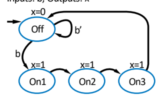
- FSM behavior captured using architecture with 2 processes
  - First process models state register
    - Asynchronous reset sets state to "S_Off"
    - Rising clock edge sets currentstate to nextstate
  - Second process models combinational logic
    - Sensitive to currentstate and FSM inputs
    - Sets FSM outputs based on currentstate
    - Sets nextstate based on currentstate and present FSM input values
  - Note declaration of new type, statetype

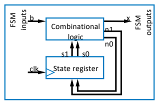
```
library ieee;
use ieee.std_logic_1164.all

entity LaserTimer is
   port (b: in std_logic;
         x: out std_logic;
         clk, rst: in std logic
   );
end LaserTimer;

architecture behavior of LaserTimer is
   type statetype is
      (S_Off, S_On1, S_On2, S_On3);
   signal currentstate, nextstate:
      statetype;
begin
   statereg: process(clk, rst)
   begin
      if (rst='1') then -- intial state
         currentstate <= S_Off;
      elsif (clk='1' and clk'event) then
         currentstate <= nextstate;
      end if;
   end process;

   comblogic: process (currentstate, b)
   begin
      case currentstate is
         when S_Off =>
            x <= '0'; -- laser off
            if (b='0') then
               nextstate <= S_Off;
            else
               nextstate <= S_On1;
            end if;
         when S_On1 =>
            x <= '1'; -- laser on
            nextstate <= S_On2;
         when S_On2 =>
            x <= '1'; -- laser still on
            nextstate <= S_On3;
         when S_On3 =>
            x <= '1'; -- laser still on
            nextstate <= S_Off;
      end case;
   end process;
end behavior;
```

54

Describing a Controller in Verilog

Inputs: b; Outputs: x





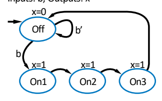
- FSM behavior captured using 2 "always" procedures
  - First procedure models state register
    - Asynchronous reset sets state to "S_Off"
    - Rising clock edge sets currentstate to nextstate
  - Second process models combinational logic
    - Sensitive to currentstate and FSM inputs
    - Sets FSM outputs based on currentstate
    - Sets nextstate based on currentstate and present FSM input values
  - Note state register size must be explicit – 2 bits, reg [1:0] currentstate

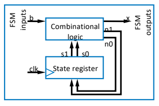
```
module LaserTimer(b, x, clk, rst);
   input b, clk, rst;
   output x;
   reg x;

   parameter S_Off = 2'b00,
             S_On1 = 2'b01,
             S_On2 = 2'b10,
             S_On3 = 2'b11;

   reg [1:0] currentstate;
   reg [1:0] nextstate;
   // state register procedure
   always @(posedge rst or posedge clk)
   begin
     if (rst==1) // initial state
       currentstate <= S_Off;
     else
       currentstate <= nextstate;
   end
   // combinational logic procedure
   always @(currentstate or b)
   begin
     case (currentstate)
       S_Off: begin
         x <= 0; // laser off
         if (b==0)
           nextstate <= S_Off;
         else
           nextstate <= S_On1;
       end
       S_On1: begin
         x <= 1; // laser on
         nextstate <= S_On2;
       end
       S_On2: begin
         x <= 1; // laser still on
         nextstate <= S_On3;
       end
       S_On3: begin
         x <= 1; // laser still on
         nextstate <= S_Off;
       end
     endcase
   end
endmodule
```

55

Describing a Controller in SystemC

Inputs: b; Outputs: x





- FSM behavior captured using 2 methods
  - First method models state register
    - Asynchronous reset sets state to "S_Off"
    - Rising clock edge sets currentstate to nextstate
  - Second process models combinational logic
    - Sensitive to currentstate and FSM inputs
    - Sets FSM outputs based on currentstate
    - Sets nextstate based on currentstate and present FSM input values
  - Note use of new type, statetype

```
#include "systemc.h"

enum statetype { S_Off, S_On1, S_On2, S_On3 };

SC_MODULE(LaserTimer)
{
  sc_in<sc_logic> b, clk, rst;
  sc_out<sc_logic> x;
  sc_signal<statetype> currentstate, nextstate;

  SC_CTOR(LaserTimer) {
    SC_METHOD(statereg);
    sensitive_pos << rst << clk;
    SC_METHOD(comblogic);
    sensitive << currentstate << b;
  }

  void statereg() {
    if( rst.read() == SC_LOGIC_1 )
      currentstate = S_Off; // initial state
    else
      currentstate = nextstate;
  }
  void comblogic() {
    switch (currentstate) {
      case S_Off:
        x.write(SC_LOGIC_0); // laser off
        if( b.read() == SC_LOGIC_0 )
          nextstate = S_Off;
        else
          nextstate = S_On1;
        break;
      case S_On1:
        x.write(SC_LOGIC_1); // laser on
        nextstate = S_On2;
        break;
      case S_On2:
        x.write(SC_LOGIC_1); // laser still on
        nextstate = S_On3;
        break;
      case S_On3:
        x.write(SC_LOGIC_1); // laser still on
        nextstate = S_Off;
        break;
    }
  }
};
```

Datapath Component Description using Hardware Description Languages

9.4

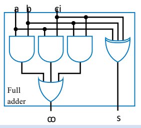
- Will consider description of three datapath components
  - Full-adders
  - Carry-ripple adders
  - Up-counter

57

Describing a Full-Adder in VHDL

- Entity
  - Declares inputs/outputs
- Architecture
  - Described behaviorally (could have been described structurally)
  - Process sensitive to inputs
  - Computes expressions, sets outputs

s = a xor b xor ci
co = bc + ac + ab



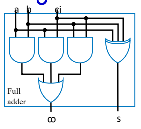
```
library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
   port ( a, b, ci: in std_logic;
          s, co: out std_logic
   );
end FullAdder;

architecture behavior of FullAdder is
begin
   process (a, b, ci)
   begin
      s <= a xor b xor ci;
      co <= (b and ci) or (a and ci) or (a and b);
   end process;
end behavior;
```

58

Describing a Full-Adder in Verilog

- Module
  - Declares inputs/outputs
  - Described behaviorally (could have been described structurally)
  - "always" procedure
    - Sensitive to inputs
  - Computes expressions, sets outputs

s = a xor b xor ci
co = bc + ac + ab



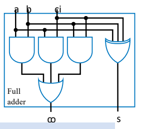
```
module FullAdder(a, b, ci, s, co);
   input a, b, ci;
   output s, co;
   reg s, co;

   always @(a or b or ci)
   begin
      s <= a ^ b ^ ci;
      co <= (b & ci) | (a & ci) | (a & b);
   end
endmodule
```

59

Describing a Full-Adder in SystemC

- Module
  - Declares inputs/outputs
  - Described behaviorally (could have been described structurally)
  - comblogic method
    - Computes expressions, sets outputs

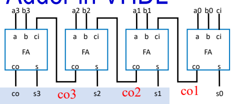
s = a xor b xor ci
co = bc + ac + ab



```
#include "systemc.h"

SC_MODULE(FullAdder)
{
  sc_in<sc_logic> a, b, ci;
  sc_out<sc_logic> s, co;

  SC_CTOR(FullAdder)
  {
    SC_METHOD(comblogic);
    sensitive << a << b << ci;
  }

  void comblogic()
  {
    s.write(a.read() ^ b.read() ^ ci.read());
    co.write((b.read() & ci.read()) |
             (a.read() & ci.read()) |
             (a.read() & b.read()));
  }
};
```

60

Describing a Carry-Ripple Adder in VHDL



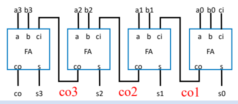
- Entity
  - Declares inputs/outputs
  - Uses std_logic_vector for 4-bit inputs/outputs
- Architecture
  - Described structurally by composing four full-adders (could have been described behaviorally instead)
  - Declares full-adder component, instantiates four full-adders, connects
    - Note use of three internal signals for connecting carry-out of one stage to carry-in of next stage

```
library ieee;
use ieee.std_logic_1164.all;

entity CarryRippleAdder4 is
   port ( a:  in std_logic_vector(3 downto 0);
          b:  in std_logic_vector(3 downto 0);
          ci: in std_logic;
          s:  out std_logic_vector(3 downto 0);
          co: out std_logic
   );
end CarryRippleAdder4;

architecture structure of CarryRippleAdder4 is
   component FullAdder
      port ( a, b, ci: in std_logic;
             s, co: out std_logic
      );
   end component;
   signal co1, co2, co3: std_logic;
begin
   FullAdder1: FullAdder
      port map (a(0), b(0), ci, s(0), co1);
   FullAdder2: FullAdder
      port map (a(1), b(1), co1, s(1), co2);
   FullAdder3: FullAdder
      port map (a(2), b(2), co2, s(2), co3);
   FullAdder4: FullAdder
      port map (a(3), b(3), co3, s(3), co);
end structure;
```

61

Describing a Carry-Ripple Adder in Verilog

- Module
  - Declares inputs/outputs
  - Uses vectors for 4-bit inputs/outputs
  - Described structurally by composing four full-adders (could have been described behaviorally instead)
  - Instantiates four full-adders, connects
    - Note use of three internal wires for connecting carry-out of one stage to carry-in of next stage

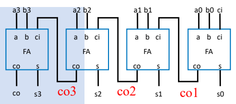


```
module CarryRippleAdder4(a, b, ci, s, co);
  input [3:0] a;
  input [3:0] b;
  input ci;
  output [3:0] s;
  output co;

  wire co1, co2, co3;

  FullAdder FullAdder1(a[0], b[0], ci,
                       s[0], co1);
  FullAdder FullAdder2(a[1], b[1], co1,
                       s[1], co2);
  FullAdder FullAdder3(a[2], b[2], co2,
                       s[2], co3);
  FullAdder FullAdder4(a[3], b[3], co3,
                       s[3], co);
endmodule
```

62

Describing a Carry-Ripple Adder in SystemC

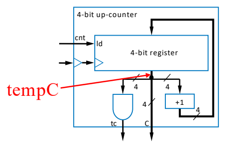
- Module
  - Declares inputs/outputs
  - Uses vectors for 4-bit inputs/outputs
  - Described structurally by composing four full-adders (could have been described behaviorally instead)
  - Instantiates four full-adders, connects
    - Note use of three internal wires for connecting carry-out of one stage to carry-in of next stage



```
#include "systemc.h"
#include "fulladder.h"

SC_MODULE(CarryRippleAdder4)
{
  sc_in<sc_logic> a[4];
  sc_in<sc_logic> b[4];
  sc_in<sc_logic> ci;
  sc_out<sc_logic> s[4];
  sc_out<sc_logic> co;

  sc_signal<sc_logic> co1, co2, co3;

  FullAdder FullAdder_1;
  FullAdder FullAdder_2;
  FullAdder FullAdder_3;
  FullAdder FullAdder_4;

  SC_CTOR(CarryRipple4):
    FullAdder_1("FullAdder_1"),
    FullAdder_2("FullAdder_2"),
    FullAdder_3("FullAdder_3"),
    FullAdder_4("FullAdder_4")
  {
    FullAdder_1.a(a[0]); FullAdder_1.b(b[0]);
    FullAdder_1.ci(ci); FullAdder_1.s(s[0]);
    FullAdder_1.co(co1);

    FullAdder_2.a(a[1]); FullAdder_2.b(b[1]);
    FullAdder_2.ci(co1); FullAdder_2.s(s[1]);
    FullAdder_2.co(co2);

    FullAdder_3.a(a[2]); FullAdder_3.b(b[2]);
    FullAdder_3.ci(co2); FullAdder_3.s(s[2]);
    FullAdder_3.co(co3);

    FullAdder_4.a(a[3]); FullAdder_4.b(b[3]);
    FullAdder_4.ci(co3); FullAdder_4.s(s[3]);
    FullAdder_4.co(co);
  }
};
```

63

Describing an Up-Counter in VHDL

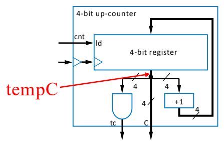


- Described structurally (could have been described behaviorally)
- Includes process that updates output port C whenever internal signal tempC changes
  - Need tempC signal because can't read C due to C being an output port

```
library ieee;
use ieee.std_logic_1164.all;

entity UpCounter is
   port ( clk: in std_logic;
          cnt: in std_logic;
          C: out std_logic_vector(3 downto 0);
          tc: out std_logic
   );
end UpCounter;

architecture structure of UpCounter is
   component Reg4
      port ( I: in std_logic_vector(3 downto 0);
             Q: out std_logic_vector(3 downto 0);
             clk, ld: in std_logic
      );
   end component;
   component Inc4
      port ( a: in std_logic_vector(3 downto 0);
             s: out std_logic_vector(3 downto 0)
      );
   end component;
   component AND4
      port ( w,x,y,z: in std_logic;
             F: out std_logic
      );
   end component;
   signal tempC: std_logic_vector(3 downto 0);
   signal incC: std_logic_vector(3 downto 0);
begin
   Reg4_1: Reg4 port map(incC, tempC, clk, cnt);
   Inc4_1: Inc4 port map(tempC, incC);
   AND4_1: AND4 port map(tempC(3), tempC(2),
                         tempC(1), tempC(0), tc);

   outputC: process(tempC)
   begin
      C <= tempC;
   end process;
end structure;
```

64

Describing an Up-Counter in Verilog

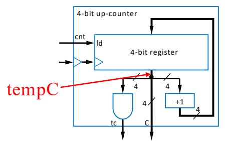


- Described structurally (could have been described behaviorally)
- Includes always procedure that updates output C whenever internal wire tempC changes
  - Need tempC wire because can't use C in the connection statements

```
module Reg4(I, Q, clk, ld);
  input [3:0] I;
  input clk, ld;
  output [3:0] Q;
  // details not shown
endmodule

module Inc4(a, s);
  input [3:0] a;
  output [3:0] s;
  // details not shown
endmodule

module AND4(w,x,y,z,F);
  input w, x, y, z;
  output F;
  // details not shown
endmodule

module UpCounter(clk, cnt, C, tc);
  input clk, cnt;
  output [3:0] C;
  reg [3:0] C;
  output tc;

  wire [3:0] tempC;
  wire [3:0] incC;

  Reg4 Reg4_1(incC, tempC, clk, cnt);
  Inc4 Inc4_1(tempC, incC);
  AND4 AND4_1(tempC[3], tempC[2],
              tempC[1], tempC[0], tc);

  always @(tempC)
  begin
    C <= tempC;
  end
endmodule
```

65

Describing an Up-Counter in SystemC



- Described structurally (could have been described behaviorally)
- Includes method that updates output C whenever internal signal tempC changes
  - Need tempC signal because can't use C in the connection statements
- Can't use logic vector bits individually for connections, so needed tempC_b array too

```
#include "systemc.h"
#include "reg4.h"
#include "inc4.h"
#include "and4.h"

SC_MODULE(UpCounter)
{
  sc_in<sc_logic> clk, cnt;
  sc_out<sc_lv<4> > C;
  sc_out<sc_logic> tc;

  sc_signal<sc_lv<4> > tempC, incC;
  sc_signal<sc_logic> tempC_b[4];

  Reg4 Reg4_1;
  Inc4 Inc4_1;
  AND4 AND4_1;

  SC_CTOR(UpCounter) : Reg4_1("Reg4_1"),
                       Inc4_1("Inc4_1"),
                       AND4_1("AND4_1")
  {
   Reg4_1.I(incC); Reg4_1.Q(tempC);
   Reg4_1.clk(clk); Reg4_1.ld(cnt);

   Inc4_1.a(tempC); Inc4_1.s(incC);

   AND4_1.w(tempC_b[0]); AND4_1.x(tempC_b[1]);
   AND4_1.y(tempC_b[2]); AND4_1.z(tempC_b[3]);
   AND4_1.F(tc);

   SC_METHOD(comblogic);
   sensitive << tempC;
 }

 void comblogic()
 {
   tempC_b[0] = tempC.read()[0];
   tempC_b[1] = tempC.read()[1];
   tempC_b[2] = tempC.read()[2];
   tempC_b[3] = tempC.read()[3];
   C.write(tempC);
 }
};
```

66

RTL Design using Hardware Description Languages

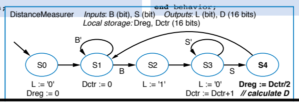
- Will consider two forms of RTL descriptions
  - High-level state machine
  - Controller and datapath

HLSM of Laser-Based Dist. Measurer: VHDL

- Architecture similar to FSM
  - Curr/next sigs for all regs
- Asynch reset forces to state S0
- Rising clock
  - Perform state's computation
  - Prepare to go to next state based on state and inputs
- Concurrent sig assignment sets D to Dreg always

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LaserDistMeasurer is
   port (
      clk, rst : in  std_logic;
      B, S     : in  std_logic;
      L        : out std_logic;
      D        : out std_logic_vector
                       (15 downto 0)
);
end LaserDistMeasurer;

architecture behavior of
      LaserDistMeasurer is

   type statetype is (S0,S1,S2,S3,S4);
   signal State, StateNext : statetype;

   signal Dctr, DctrNext:
       std_logic_vector(15 downto 0);
   signal Dreg, DregNext:
       std_logic_vector(15 downto 0);

   constant U_ZERO :
       std_logic_vector(15 downto 0)
              := "0000000000000000";
   constant U_ONE  :
       std_logic_vector(15 downto 0)
              := "0000000000000001";

begin

Regs: process(clk, rst)
begin
   if(rst = '1') then
      State <= S0;
      Dctr <= U_ZERO;
      Dreg <= U_ZERO;
   elsif(clk'event and clk='1') then
      State <= StateNext;
      Dctr  <= DctrNext;
      Dreg  <= DregNext;
   end if;
end process;
```

```
CombLogic: process(State, Dctr, B, S)
   begin
      case State is
         when S0 =>
            L <= '0'; -- laser off
            DregNext <= U_ZERO; --clr D
            DctrNext <= U_ZERO; --clr Dctr
            StateNext <= S1;
         when S1 =>
            DctrNext <= U_ZERO; --clr Dctr
            L <= '0'; -- laser off

            if(B = '1') then
               StateNext <= S2;
            else
               StateNext <= S1;
            end if;
         when S2 =>
            L <= '1';  --laser on
            DctrNext <= U_ZERO;
            StateNext <= S3;
         when S3 =>
            L <= '0';  -- laser off
            DctrNext <= Dctr + 1;

            if( S = '1') then
               StateNext <= S4;
            else
               StateNext <= S3;
            end if;
         when S4 =>
            DctrNext <= Dctr;
            DregNext <= SHR(Dctr, U_ONE);
            L <= '0';
            StateNext <= S1;
         when others =>
            DregNext <= U_ZERO;
            DctrNext <= U_ZERO;
            L <= '0';
            StateNext <= S0;
      end case;
   end process;

--assign Dreg output to D output
D <= Dreg;

end behavior;
```

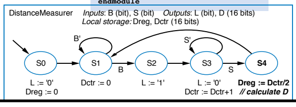


HLSM of Laser-Based Distance Measurer: Verilog

```
module LaserDistMeasurer(clk,rst,B,S,L,D);
   input clk, rst, B, S;
   output L;
   output [15:0] D;
   reg L;
   reg [15:0] D;

   parameter    S0 = 3'b000,
                S1 = 3'b001,
                S2 = 3'b010,
                S3 = 3'b011,
                S4 = 3'b100;

   reg [2:0] State, StateNext;
   reg [15:0] Dctr, DctrNext;
   reg [15:0] Dreg, DregNext;

   //Registers
   always@(posedge clk, posedge rst) begin
      if(rst == 1) begin //asynchr. reset
         State <= S0;
         Dctr  <= 0;
         Dreg  <= 0;
      end
      else begin
         State <= StateNext;
         Dctr  <= DctrNext;
         Dreg  <= DregNext;
      end
   end

   always @(Dreg) begin
      D <= Dreg;
   end
```

```
   //Combinational logic
   always@(State, Dctr, B, S) begin
      case (State)
         S0: begin
            L <= 0; //Laser off
            DregNext <= 0; //clr D
            StateNext <= S1;
            DctrNext  <= 0;
         end
         S1: begin
            DctrNext <= 0;
            L <= 0;

            if(B == 1)
               StateNext <= S2;
            else
               StateNext <= S1;
        end
        S2: begin
            L <= 1;      //Laser on
            DctrNext <= 0;
            StateNext <= S3;
        end
        S3: begin
           L <= 0;       //Laser off
           DctrNext <= Dctr + 1;

           if(S == 1)
              StateNext <= S4;
           else
              StateNext <= S3;
        end
        S4: begin
           DregNext <= Dctr >> 1;
           StateNext <= S1;
        end
      endcase
    end
 endmodule
```

- Architecture similar to FSM
  - Curr/next sigs for all regs
- Asynch reset forces to state S0
- Rising clock
  - Perform state's computation
  - Prepare to go to next state based on state and inputs
- Another procedure sets D to Dreg always

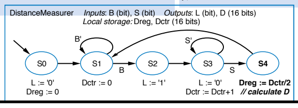


HLSM of Laser-Based Distance Measurer: SystemC

- Architecture similar to FSM
  - Curr/next sigs for all regs
- Asynch reset forces to state S0
- Rising clock
  - Perform state's computation
  - Prepare to go to next state based on state and inputs
- Another process sets D to Dreg always

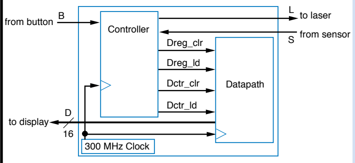
```
#include "systemc.h"

enum statetype { S0, S1, S2, S3, S4 };

SC_MODULE(LaserDistMeasurer)
{
   sc_in<sc_logic> clk, rst;
   sc_in<sc_logic> B, S;
   sc_out<sc_logic> L;
   sc_out<sc_uint<16> > D;

   sc_signal<statetype> State, StateNext;
   sc_signal<sc_uint<16> > Dctr, DctrNext;
   sc_signal<sc_uint<16> > Dreg, DregNext;

   SC_CTOR(LaserDistMeasurer)
   {
      SC_METHOD(Regs);
      sensitive << clk.pos();

      SC_METHOD(CombLogic);
      sensitive << State << B << S << Dctr;

      SC_METHOD(Output);
      sensitive << Dreg;
   }

   void Regs(){
      if (rst.read() == SC_LOGIC_1){
         State.write(S0);
         Dctr.write(0);
         Dreg.write(0);
      }
      else{
         State.write(StateNext.read());
         Dctr.write(DctrNext.read());
         Dreg.write(DregNext.read());
      }
   }
```

```
 void CombLogic(){
    switch (State) {
       case S0:
          L.write(SC_LOGIC_0);//laser off
          StateNext.write(S1);
          break;
       case S1:
          DctrNext.write(0); // clr count
          if (B.read() == SC_LOGIC_1){
             StateNext.write(S2);
          }
          break;
       case S2:
          L.write(SC_LOGIC_1);//laser on
          StateNext.write(S3);
          break;
       case S3:
          L.write(SC_LOGIC_0);//laser off
          DctrNext.write(Dctr.read()+1);
          if (S.read() == SC_LOGIC_1)
             StateNext.write(S4);
          else
             StateNext.write(S3);
          break;
       case S4:
          DregNext.write
             ((Dctr.read()>>1));
          StateNext.write(S1);
          break;
    }
 }

 void Output(){
    D.write(Dreg.read());
 }

};
```



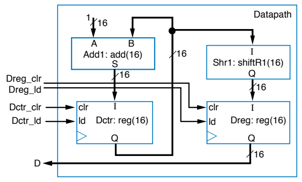
Controller and DP of Laser-Based Distance Measurer in VHDL

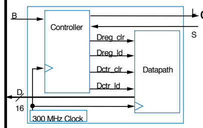
- At highest level, just connection of controller and datapath components



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LaserDistMeasurer is
    port (
       clk, rst : in  std_logic;
       B, S    : in  std_logic;
       L       : out  std_logic;
       D       : out
          std_logic_vector(15 downto 0)
       );
end LaserDistMeasurer;

architecture structure of LaserDistMeasurer is
   component LDM_Controller
   port ( clk, rst : in std_logic;
          B, S     : in std_logic;
          L        : out std_logic;
          Dreg_clr, Dreg_ld : out std_logic;
          Dctr_clr, Dctr_ld : out std_logic
       );
   end component;

   component LDM_Datapath
   port ( clk : in std_logic;
          Dreg_clr, Dreg_ld : in std_logic;
          Dctr_clr, Dctr_ld : in std_logic;
          D : out std_logic_vector(15 downto 0)
       );
   end component;

   signal Dreg_clr, Dreg_ld : std_logic;
   signal Dctr_clr, Dctr_ld : std_logic;

  begin
    LDM_Controller_1 : LDM_Controller
      port map ( clk, rst, B, S, L,
                 Dreg_clr, Dreg_ld, Dctr_clr,
                    Dctr_ld);

    LDM_Datapath_1 : LDM_Datapath
      port map ( clk, Dreg_clr, Dreg_ld,
                 Dctr_clr, Dctr_ld, D);
end structure;
```

DP of Laser-Based Distance Measurer in VHDL

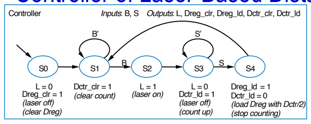




- Datapath is just another connection of components
  - Assume adder, register, and shift-right components are already designed (similar to earlier-designed items)

```
library ieee;
use ieee.std_logic_1164.all;

entity LDM_Datapath is
   port ( clk: in std_logic;
          Dreg_clr, Dreg_ld: in std_logic;
          Dctr_clr, Dctr_cnt: in std_logic;
          D: out std_logic_vector(15 downto 0)
   );
end LDM_Datapath;

architecture structure of LDM_Datapath is
   component UpCounter16
      port ( clk: in stdlogic;
             clr, cnt: in std_logic;
             C: out std_logic_vector(15 downto 0)
      );
   end component;
   component Reg16
      port ( I: in std_logic_vector(15 downto 0);
             Q: out std_logic_vector(15 downto 0);
             clk, clr, ld: in std_logic
      );
   end component;
   component ShiftRightOne16
      port ( I: in std_logic_vector(15 downto 0);
             S: out std_logic_vector(15 downto 0)
      );
   end component;
   signal tempC : std_logic_vector(15 downto 0);
   signal shiftC : std_logic_vector(15 d ownto 0);
begin
   Dctr: UpCounter16
      port map (clk, Dctr_clr, Dctr_cnt, tempC);
   ShiftRight: ShiftRightOne16
      port map (tempC, shiftC);
   Dreg: Reg16
      port map (shiftC, D, clk, Dreg_clr, Dreg_ld);
end structure;
```

Controller of Laser-Based Distance Measurer: VHDL



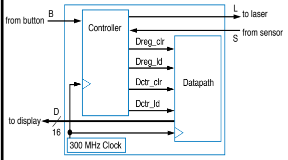
- FSM similar to high-level state machine
  - But high-level operations replaced by low-level datapath signals
  - Use two-process FSM description approach

```
library ieee;
use ieee.std_logic_1164.all;

entity LDM_Controller is
   port ( clk, rst: in std_logic;
          B, S: in std_logic;
          L: out std_logic;
          Dreg_clr, Dreg_ld: out std_logic;
          Dctr_clr, Dctr_ld: out std_logic
         );
end LDM_Controller;

architecture behavior of LDM_Controller is

type statetype is (S0, S1, S2, S3, S4);
signal currentstate, nextstate: statetype;

begin
statereg: process(clk, rst)
begin
   if (rst='1') then
      currentstate <= S0; -- initial state
   elsif (clk='1' and clk'event) then
      currentstate <= nextstate;
    end if;
end process;
```

```
comblogic: process(currentstate, B, S)
begin
   L <= '0';
   Dreg_clr <= '0';
   Dreg_ld <= '0';
   Dctr_clr <= '0';
   Dctr_ld <= '0';
   case currentstate is
      when S0 =>
         L <= '0'; -- laser off
         Dreg_clr <= '1'; -- clr Dreg
         nextstate <= S1;
      when S1 =>
         Dctr_clr <= '1'; -- clr count
         if (B='1') then
            nextstate <= S2;
         else
            nextstate <= S1;
         end if;
      when S2 =>
         L <= '1'; -- laser on
         nextstate <= S3;
      when S3 =>
         L <= '0'; -- laser off
         Dctr_ld <= '1'; -- count up
         if (S='1') then
            nextstate <= S4;
         else
            nextstate <= S3;
         end if;
      when S4 =>
         Dreg_ld <= '1'; -- load Dreg
         Dctr_ld <= '0'; -- stop count
         nextstate <= S1;
   end case;
end process;

end behavior;
```

Controller and DP of Laser-Based Distance Measurer: Verilog

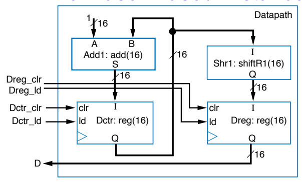
- At highest level, just connection of controller and datapath components



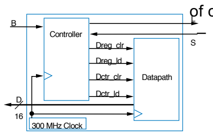
```
module LaserDistMeasurer(clk,rst,B,S,L,D);
   input clk, rst, B, S;
   output L;
   output [15:0] D;

   wire Dreg_clr, Dreg_ld;
   wire Dctr_clr, Dctr_ld;

   LDM_Controller
      LDM_Controller_1(clk, rst, B, S, L,
                       Dreg_clr, Dreg_ld,
                       Dctr_clr, Dctr_ld);
   LDM_Datapath
      LDM_Datapath_1(clk, Dreg_clr, Dreg_ld,
                     Dctr_clr, Dctr_ld, D);
endmodule
```

74

DP of Laser-Based Distance Measurer:Verilog



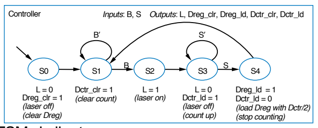
- Datapath just another connection of components
  - Assume adder, register, and shift-right components are already designed (similar to earlier-designed items)



```
module Add16(A, B, S);
   input [15:0] A, B;
   output [15:0] S;
   //details not shown
endmodule

module Reg16(I, Q, clk, clr, ld);
   input [15:0] I;
   input clk, clr, ld;
   output [15:0] Q;
   // details not shown
endmodule

module ShiftR1_16(I, S);
   input [15:0] I;
   output [15:0] S;
   // details not shown
endmodule

module LDM_Datapath(clk, Dreg_clr, Dreg_ld,
                    Dctr_clr, Dctr_ld, D);
   input clk;
   input Dreg_clr, Dreg_ld;
   input Dctr_clr, Dctr_ld;
   output [15:0] D;

   wire [15:0] addC, tempC, shiftC;

   Reg16 Dctr(addC,tempC,clk,Dctr_clr,Dctr_ld);
   Add16 Add1(1, tempC, addC);
   ShiftR1_16 ShiftRight(tempC, shiftC);
   Reg16 Dreg(shiftC,D,clk,Dreg_clr,Dreg_ld);
endmodule
```

75

Controller of Laser-Based Distance Measurer: Verilog



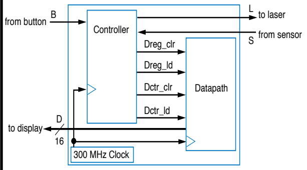
- FSM similar to high-level state machine
  - But high-level operations replaced by low-level datapath signals
  - Use two-procedure FSM description approach

```
module LDM_Controller
   (clk, rst, B, S, L, Dreg_clk,
    Dreg_ld, Dctr_clr, Dctr_ld);
   input clk, rst, B, S;
   output L;
   output Dreg_clk, Dreg_ld;
   output Dctr_clr, Dctr_ld;
   reg L;
   reg Dreg_clr, Dreg_ld;
   reg Dctr_clr, Dctr_ld;

   parameter S0 = 3'b000,
              S1 = 3'b001,
             S2 = 3'b010,
             S3 = 3'b011,
              S4 = 3'b100;

   reg [2:0] State;
   reg [2:0] StateNext;

   always @(posedge rst or posedge clk)
   begin
      if (rst==1)
         State <= S0; // initial state
      else
          State <= StateNext;
   end
```

```
  always @(State or B or S)
    begin
       L <= 0;
      Dreg_clr <= 0;
      Dreg_ld <= 0;
       Dctr_clr <= 0;
       Dctr_ld <= 0;
      case (State)
         S0:  begin
            L <= 0; // laser off
              Dreg_clr <= 1; // clr Dreg
              StateNext <= S1;
          end
         S1:  begin
            Dctr_clr <= 1; // clr count
             if (B==1)
                 StateNext <= S2;
             else
               StateNext <= S1;
          end
          S2:begin
              L <= 1; // laser on
            StateNext <= S3;
          end
          S3:begin
              L <= 0; // laser off
              Dctr_ld <= 1; // count up
             if (S==1)
               StateNext <= S4;
             else
                 StateNext <= S3;
          end
         S4:  begin
              Dreg_ld <= 1; // load Dreg
              Dctr_ld <= 0; // stop count
              StateNext <= S1;
          end
      endcase
  end
endmodule
```

Controller and DP of Laser-Based Distance Measurer: SystemC

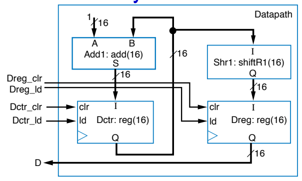
- At highest level, just connection of controller and datapath components



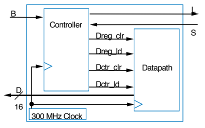
```
#include "systemc.h"
#include "LDM_Controller.h"
#include "LDM_Datapath.h"

SC_MODULE(LaserDistMeasurer)
{
   sc_in<sc_logic> clk, rst;
   sc_in<sc_logic> B, S;
   sc_out<sc_logic> L;
   sc_out<sc_uint<16> > D;

   sc_signal<sc_logic> Dreg_clr, Dreg_ld;
   sc_signal<sc_logic> Dctr_clr, Dctr_ld;

   LDM_Controller LDM_Controller_1;
   LDM_Datapath LDM_Datapath_1;

   SC_CTOR(LaserDistMeasurer) :
      LDM_Controller_1("LDM_Controller_1"),
      LDM_Datapath_1("LDM_Datapath_1")
   {
      LDM_Controller_1.clk(clk);
      LDM_Controller_1.rst(rst);
      LDM_Controller_1.B(B);
      LDM_Controller_1.S(S);
      LDM_Controller_1.Dreg_clr(Dreg_clr);
      LDM_Controller_1.Dreg_ld(Dreg_ld);
      LDM_Controller_1.Dctr_clr(Dctr_clr);
      LDM_Controller_1.Dctr_ld(Dctr_ld);
      LDM_Datapath_1.clk(clk);
      LDM_Datapath_1.Dreg_clr(Dreg_clr);
      LDM_Datapath_1.Dreg_ld(Dreg_ld);
      LDM_Datapath_1.Dctr_clr(Dctr_clr);
      LDM_Datapath_1.Dctr_ld(Dctr_ld);
      LDM_Datapath_1.D(D);
   }
};
```

77

DP of Laser-Based Distance Measurer: SystemC

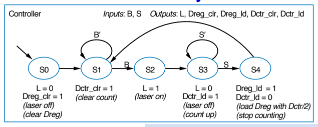


- Datapath just another connection of components
  - Assume adder, register, and shift-right components are already designed (similar to earlier-designed items)



```
#include "systemc.h"
#include "add16.h"
#include "reg16.h"
#include "shiftr1_16.h"

SC_MODULE(LDM_Datapath)
{
   sc_in<sc_logic> clk;
   sc_in<sc_logic> Dreg_clr, Dreg_ld;
   sc_in<sc_logic> Dctr_clr, Dctr_ld;
   sc_out<sc_uint<16> > D;

   sc_signal<sc_uint<16> > tempC;
   sc_signal<sc_uint<16> > addC;
   sc_signal<sc_uint<16> > shiftC;

   Add16 Add1;
   Reg16 Dctr;
   Reg16 Dreg;
   ShiftR1_16 ShiftRight;

   SC_CTOR(LDM_Datapath) :
      Dctr("Dctr"),
      Dreg("Dreg"),
      Add1("Add1"),
      ShiftRight("ShiftRight")
   {
      Add1.A(1);
      Add1.B(tempC);
      Add1.B(addC);

      Dctr.I(addC);
      Dctr.Q(tempC);
      Dctr.clk(clk);
      Dctr.clr(Dctr_clr);
      Dctr.ld(Dctr_ld);

      ShiftRight.I(tempC);
      ShiftRight.S(shiftC);

      Dreg.I(shiftC);
      Dreg.Q(D);
      Dreg.clk(clk);
      Dreg.clr(Dreg_clr);
      Dreg.ld(Dreg_ld);
   }
};
```

Controller of Laser-Based Distance Measurer: SystemC



- FSM similar to high-level state machine
  - But high-level operations replaced by low-level datapath signals
  - Use two-procedure FSM description approach

```
#include "systemc.h"

enum statetype { S0, S1, S2, S3, S4 };
SC_MODULE (LDM_Controller)
{
   sc_in<sc_logic > clk, rst, B, S;
   sc_out <sc_logic > L;
   sc_out <sc_logic > Dreg_clr, Dreg_ld;
   sc_out<sc_logic> Dctr_clr, Dctr_ld;

   sc_signal<statetype> State, StateNext;

   SC_CTOR(LDM_Controller)
   {
      SC_METHOD(statereg);
      sensitive << clk.pos();

      SC_METHOD(comblogic);
      sensitive << State << B << S;
   }

   void statereg() {
      if ( rst.read() == SC_LOGIC_1 )
         State = S0; // initial state
      else
         State = StateNext;
   }
```

```
void comblogic() {
   L.write(SC_LOGIC_0);
   Dreg_clr. write(SC_LOGIC_0);
    Dreg_ldwrite(SC_LOGIC_0);
    Dctr_clrwrite(SC_LOGIC_0);
    Dctr_ldwrite(SC_LOGIC_0);

   switch (State) {
      case S0:
          Lwrite(SC_LOGIC_0); // laser off
          Dreg_clrwrite(SC_LOGIC_0);
          StateNextwrite(S1);
         break;
      case S1:
          Dctr_clrwrite(SC_LOGIC_1);
         if (B.read() == SC_LOGIC_1)
             StateNextwrite(S2);
         else
           StateNext. write(S1);
         break ;
      case S2:
        L. write(SC_LOGIC_1); // laser on
        StateNext. write(S3);
         break ;
      case S3:
        L. write(SC_LOGIC_0); // laser off
        Dctr_ld. write(SC_LOGIC_1);
         if (S.read() == SC_LOGIC_1)
           StateNext. write(S4);
         else
             StateNextwrite(S3);
         break;
      case S4:
          Dreg_ldwrite(SC_LOGIC_1);
          Dctr_ldwrite(SC_LOGIC_0);
          StateNextwrite(S1);
      break;
   }
 }
};
```

Summary

- Programmable processors are widely used
  - Easy availability, short design time
- Basic architecture
  - Datapath with register file and ALU
  - Control unit with PC, IR, and controller
  - Memories for instructions and data
  - Control unit fetches, decodes, and executes
- Three-instruction processor with machine-level programs
  - Extended to six instructions
  - Real processors have dozens or hundreds of instructions
  - Extended to access external pins
  - Modern processors are far more sophisticated
- Hardware Description Languages (HDLs)
  - VHDL, Verilog, and SystemC are popular
  - Introduced languages mainly through examples