Arithmetic circuits, parametric design and finite state machines

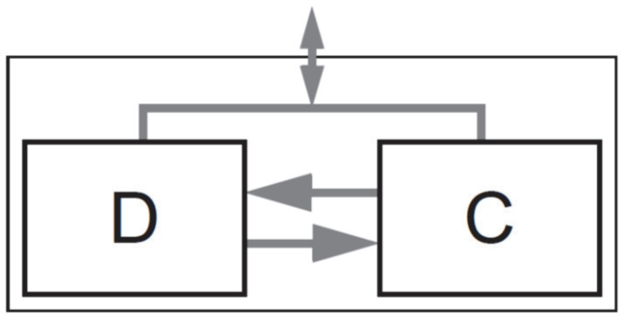
IAS0600 Digital Systems Design with VHDL

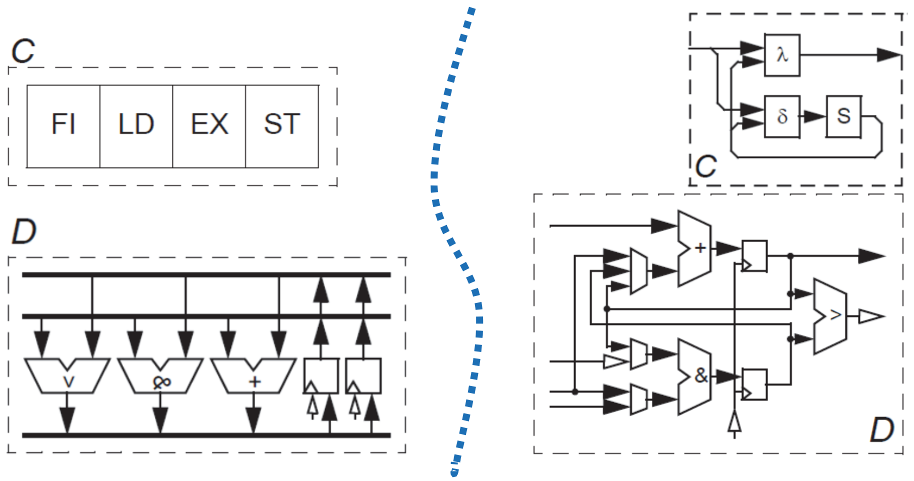
Natalia Cherezova, Peeter Ellervee

Arithmetic circuits

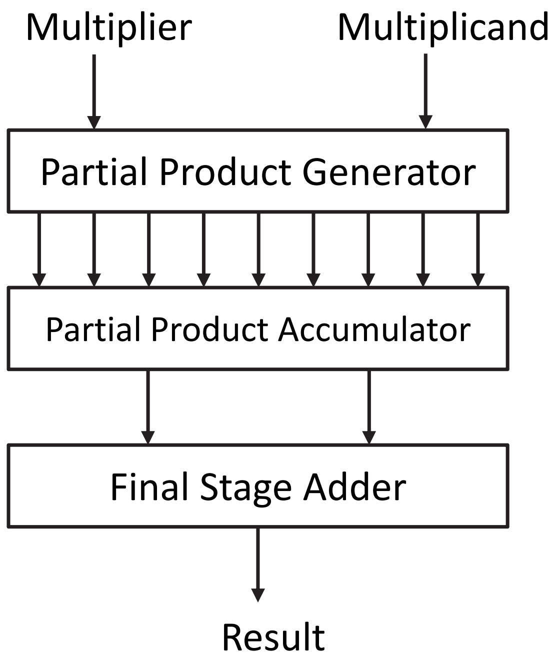
- Digital system == data-path + control-path
  - Data-path ~~ arithmetic units & registers
  - Control-path ~~ finite state machine (FSM)
    - Often multiple FSM-s

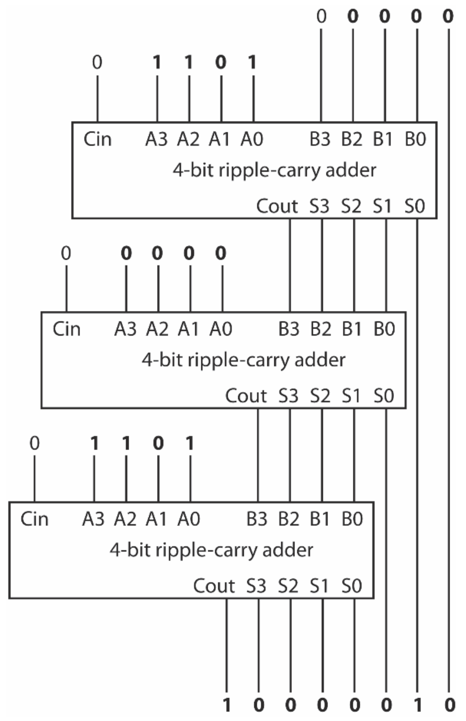
- Central building blocks of digital systems
- Various implementations exist
- Trade off between complexity and speed



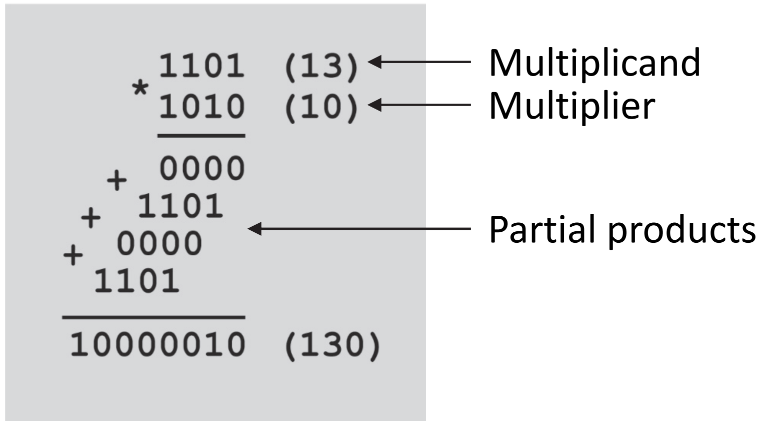


Digital multiplier





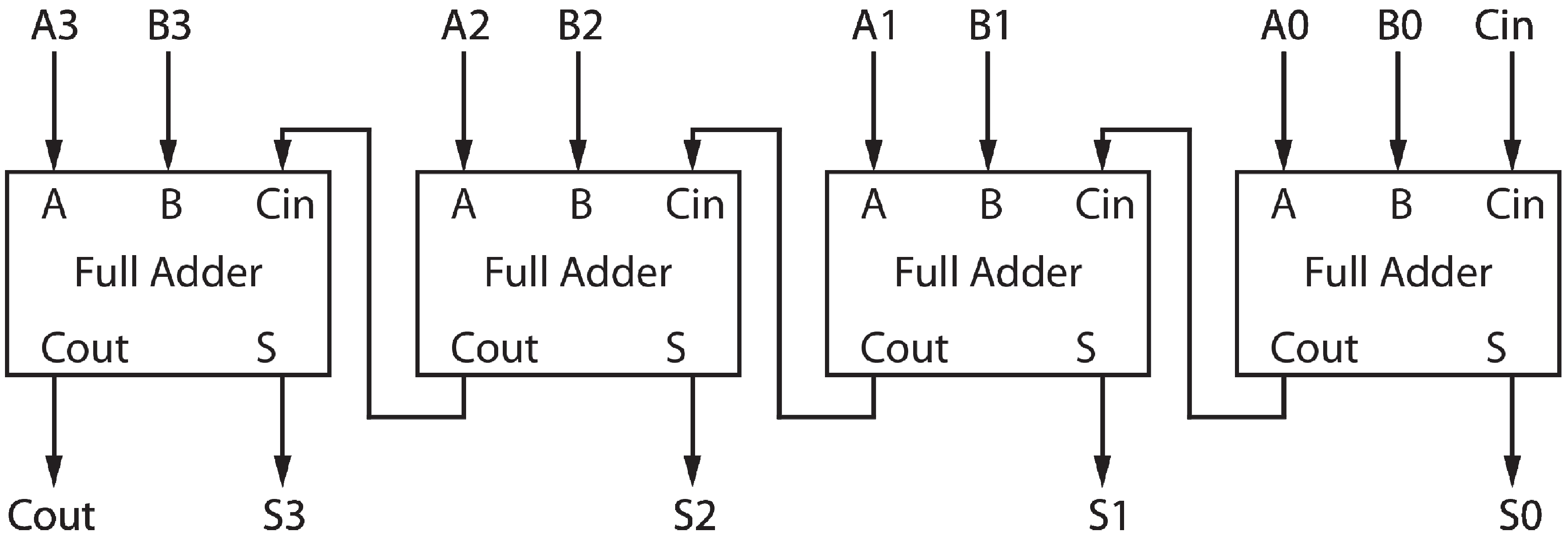
Partial products accumulator



```
     1101   (13)  ← Multiplicand
   * 1010   (10)  ← Multiplier
     ----
   + 0000
  + 1101          ← Partial products
 + 0000
  1101
 --------
 10000010   (130)
```

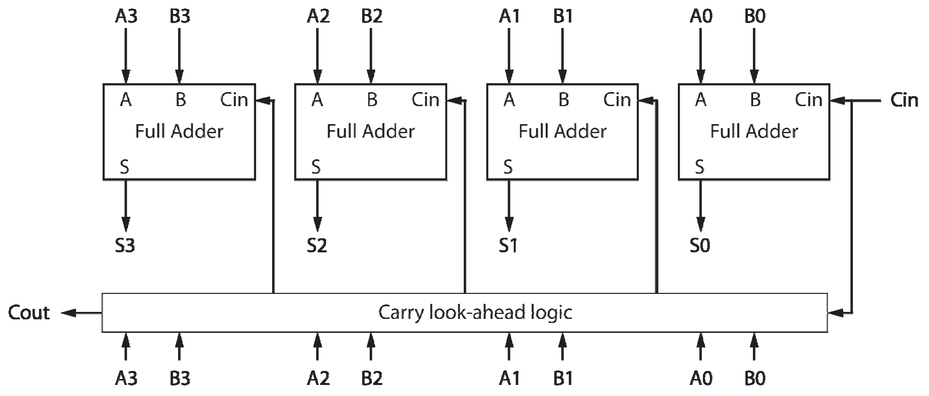
Ripple carry adder

- Full adders connected in series

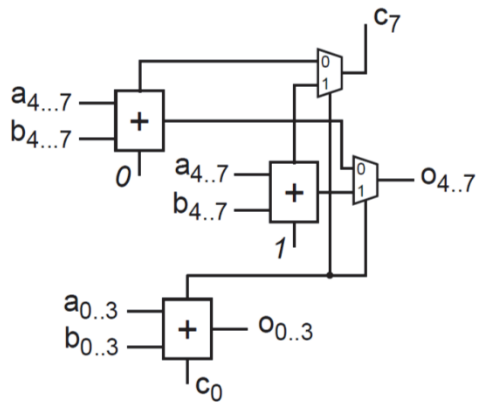


Fast carry propagation

- Carry look-ahead adder
  - Full adders work in parallel, carry values are calculated separately
  - Generated carry G_i = A_i and B_i
  - Propagated carry P_i = A_i or B_i (or P_i = A_i xor B_i)
  - C_{i+1} = G_i or (P_i and C_i)
  - C_0 = C_{input}

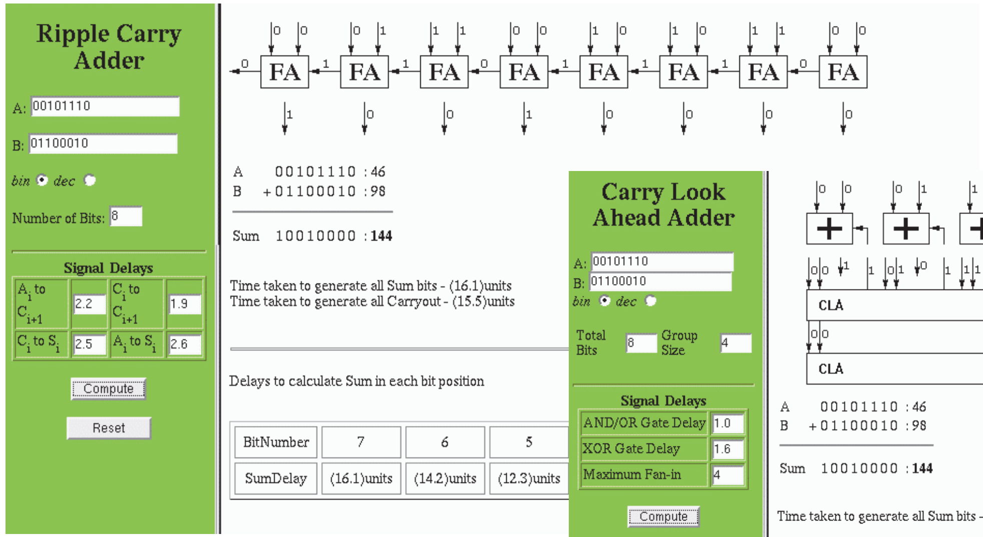


- Carry select adder
  - Result is calculated for both carry-in values
  - Result is selected when the carry-in arrives



Arithmetic operations simulator

- http://www.ecs.umass.edu/ece/koren/arith/simulator/



Ripple Carry Adder

A: 00101110

B: 01100010

bin [✓ ●] dec ○

Number of Bits: 8

Signal Delays

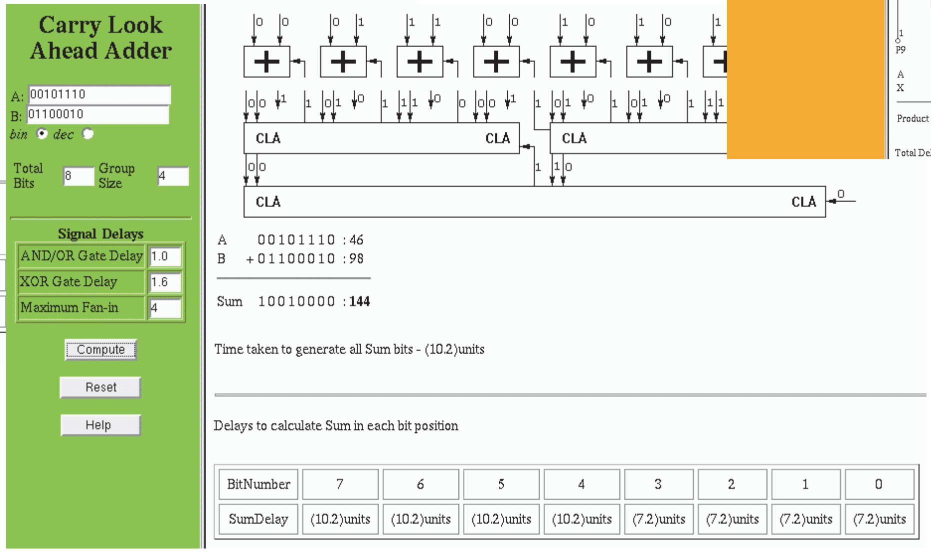
A_i to C_{i+1}	2.2	C_i to C_{i+1}	1.9
C_i to S_i	2.5	A_i to S_i	2.6

```
A      00101110 :46
B     +01100010 :98

Sum    10010000 :144
```

Time taken to generate all Sum bits - (16.1)units
Time taken to generate all Carryout - (15.5)units

Delays to calculate Sum in each bit position

BitNumber	7	6	5
SumDelay	(16.1)units	(14.2)units	(12.3)units



Carry Look Ahead Adder

A: 00101110

B: 01100010

bin [✓ ●] dec ○

Total Bits 8 Group Size 4

Signal Delays

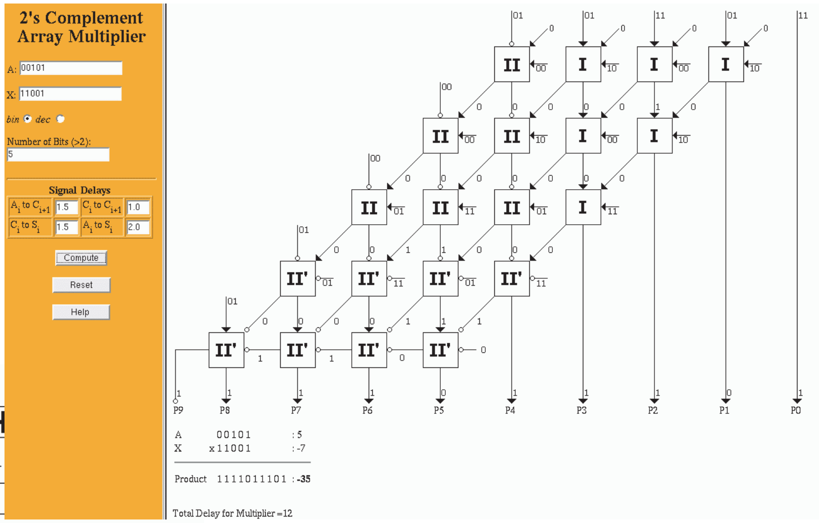
AND/OR Gate Delay	1.0
XOR Gate Delay	1.6
Maximum Fan-in	4

```
A      00101110 :46
B     +01100010 :98

Sum    10010000 :144
```

Time taken to generate all Sum bits - (10.2)units

Delays to calculate Sum in each bit position

BitNumber	7	6	5	4	3	2	1	0
SumDelay	(10.2)units	(10.2)units	(10.2)units	(10.2)units	(7.2)units	(7.2)units	(7.2)units	(7.2)units



2's Complement Array Multiplier

A: 00101

X: 11001

bin [✓ ●] dec ○

Number of Bits (>2): 5

Signal Delays

A_i to C_{i+1}	1.5	C_i to C_{i+1}	1.0
C_i to S_i	1.5	A_i to S_i	2.0

```
A          00101       : 5
X        x 11001       : -7

Product  1111011101 : -35
```

Total Delay for Multiplier =12

Parametric design

- Generic value
- Generate statement
- Attributes

Generic

- Generic parameters allow parameterizing the design
- Increase flexibility and reusability of the code
- Default value can be overwritten during component instantiation

```
entity <entity_name>
  generic (<name> : <type> := <default_value>;
           <name> : <type> := <default_value>;
           ... );
  port ( ... );
end <entity_name>;
```

Generate statement

- Concurrent statement
- Unconditional generate

Both range limits should be static

```
<label> : for <identifier> in <range> generate
  <concurrent statements>
end generate;
```

- Conditional generate

```
<label> : if <condition> generate
  <concurrent statements>
end generate;
```

Attributes

- Four categories of predefined attributes
  - Predefined attributes of scalar types
    - E.g., integer'image(x)
  - Predefined attributes of array types
  - Predefined attributes of signals
    - E.g., clock'event
  - Predefined attributes of named entities

Attributes of array types

Name	Result
`x'left`	The left bound of the index range of x
`x'right`	The right bound of the index range of x
`x'low`	Lower bound of the index range of x
`x'high`	Higher bound of the index range of x
`x'range`	Range of the index range of x
`x'reverse_range`	Reverse range of the index range of x
`x'length`	Number of values in x
`x'ascending`	TRUE if the index range of x is ascending, FALSE otherwise
`x'element`	Type of elements of x

Arrays

- Considered as a type in VHDL
- Constrained
  - Range is defined in the type definition

```
type constr_array is array (0 to n-1) of std_logic_vector(n-1 downto 0);
signal constr_array_signal : constr_array;
```

- Unconstrained
  - Range is defined in the signal declaration

```
type unconstr_arr is array (natural range <>) of std_logic_vector(n-1 downto 0);
signal unconstr_array_signal : unconstr_arr(0 to n-1);
```

NATURAL is the subtype of the INTEGER restricted to the values greater than or equal to 0

Using attributes – flexible loops

- Inflexible code

```
signal s: bit_vector (7 downto 0);
...
for i in 0 to 7 loop ...
```

- Flexible code

```
constant sz: integer := 16;
signal s: bit_vector (sz-1 downto 0);
...
for i in 0 to sz-1 loop ...
```

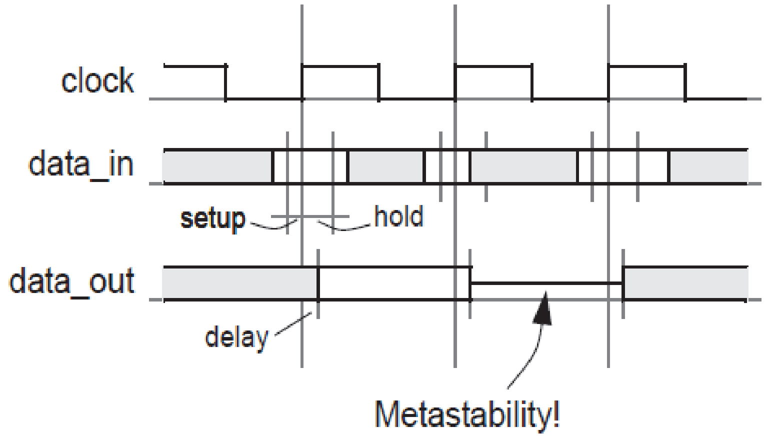
- More flexible code
  - Attributes & generic parameter

```
entity ... is
generic (sz: positive);
port ...
...
signal s: bit_vector (sz-1 downto 0);
...
for i in s'low to s'high loop ...
for i in s'reverse_range loop ...
```

Using attributes – set-up & hold times

- Set-up – the input data must be fixed for certain time before the active clock flank
- Hold – the input data must be stable for certain time after the active clock flank

```
process (clock,data_in) begin
  if clock’event and clock=’1’ then
    assert data_in’last_event >= 3 ns
      report ”setup time violation” severity warning;
    data_out <= data_in after 3 ns;
  end if;
  if data_in’event and clock=’1’ then
    assert clock’last_event >= 5 ns
      report ”hold time violation” severity warning;
  end if;
end process;
```

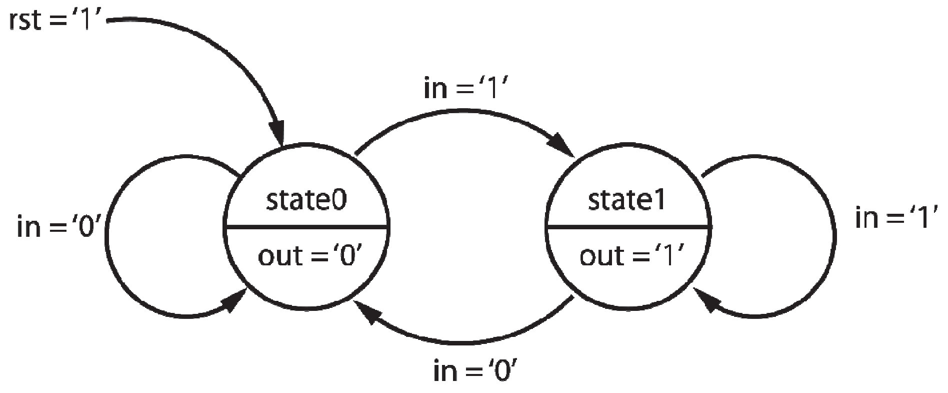


Finite State Machine

- FSM is a modelling technique that provides systematic approach for designing sequential circuits
- A process is divided into finite number of states
- Each state performs its task
- Process can be in one of those states at a time
- Workflow of the process is represented by the transition between states
- Transition between states is based on control signals (inputs)

Representation

Transition graph



Transition table

Current state	Next state			Output
	rst = '1'	in = '0'	in = '1'	
state0	state0	state0	state1	out = '0'
state1	state0	state0	state1	out = '1'

FSM types

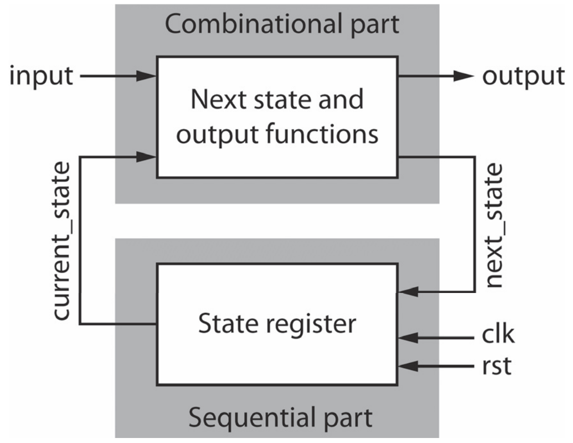
- Moore FSM
  - Named after Edward F. Moore (1925–2003)
  - The output of the state does not depend on the input
- Mealy FSM
  - Named after George H. Mealy (1927–2010)
  - The output of the state depends on the input
  - Usually require less states than the Moore machine

FSM in VHDL

- Create a state type for your FSM (enumerated type)

```
type State is (State_0, State_1, State_2, State_3, State_4);
signal current_state, next_state : State;
```

- Divide the FSM description into 2 parts (processes)
  - Combinational for output and next state computations
  - Sequential for current state storage
- Keep in mind
  - Registers change on the next clock edge
  - Outputs of the combinational logic are computed immediately

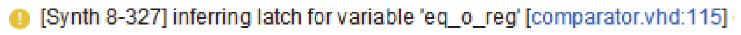


Unintended latch

- To avoid unintended latch inference from a combinational process
  - Add all read signals to the sensitivity list
  - Assign values to the output signals in every case

Tip: assign default value at the beginning of the process

- During functional simulation everything will be correct
- But the design will not work as intended on the board
- Keep an eye for warning messages from the synthesis tool

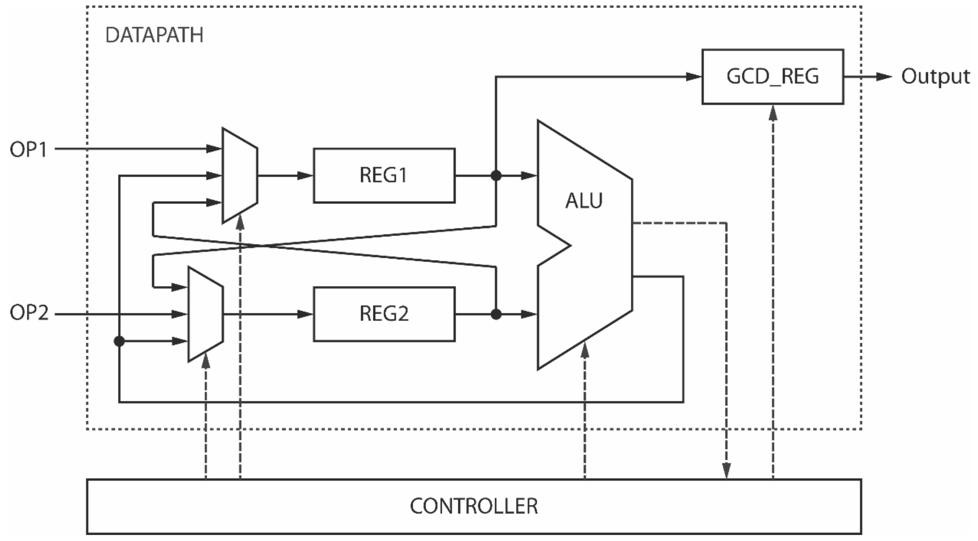


```
[Synth 8-327] inferring latch for variable 'eq_o_reg' [comparator.vhd:115]
```

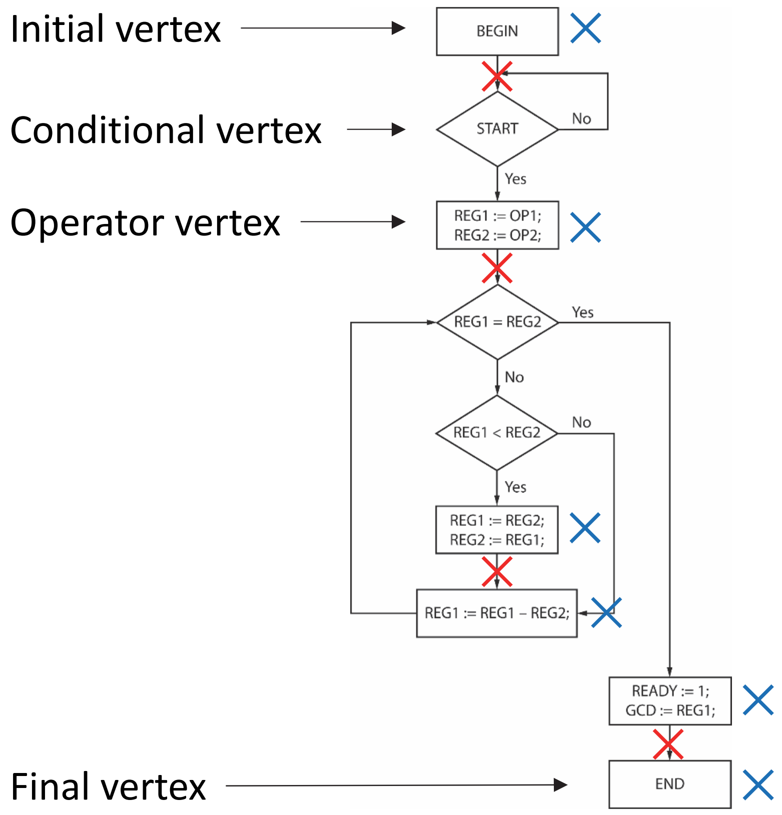
- Try post-implementation timing simulation

Datapath

- Datapath is a collection of units that hold the data and operate on the data
- Functional units that form the path between the inputs and the outputs
- Datapath is controlled by the FSM



Algorithm to FSM



- Draw an algorithmic graph
- Identify inputs of the FSM
  - x0 = start
  - x1 = (reg1 = reg2)
  - x2 = (reg1 < reg2)

Inputs are defined by the conditional vertices

- Identify outputs of the FSM
  - reg1, reg2, gcd_reg inputs
  - ready value, ALU opcode

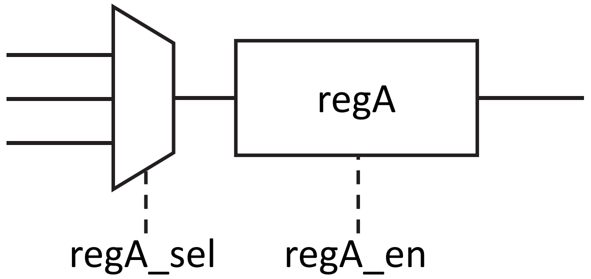
Outputs will control the operations

- Identify states

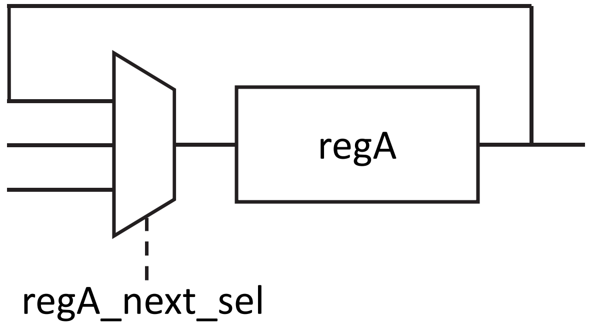
  Mealy machine: by marking the inputs of vertices following operator vertices

  Moore machine: by marking operator vertices

Controlling register values



```
-- regA_next value is defined by FSM
process (clk, rst) begin
  if rst = '0' then
    regA <= (others => '0');
  elsif rising_edge(clk) then
    regA <= regA_next;
  end if;
end process;
```



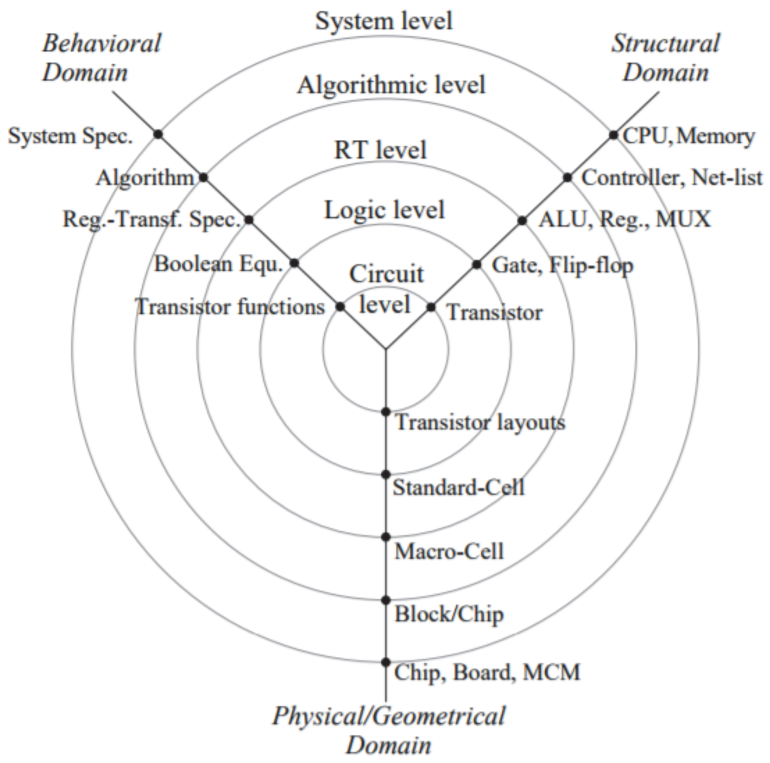
```
-- regA_en and regA_sel values
-- are defined by FSM
process (clk, rst) begin
  if rst = '0' then
    regA <= (others => '0');
  elsif rising_edge(clk) then
    if regA_en = '1' then
      regA <= regA_next;
    end if;
  end if;
end process;

with regA_sel select
  regA_next <= <...>
```

Register Transfer Level

- Register Transfer Level (RTL) is a level of abstraction for digital systems modeling
- Design is formed from a collection of registers linked by combinational logic
- System function is performed as a sequence of register transfers
- Register transfer is the movement and transformation of data between registers

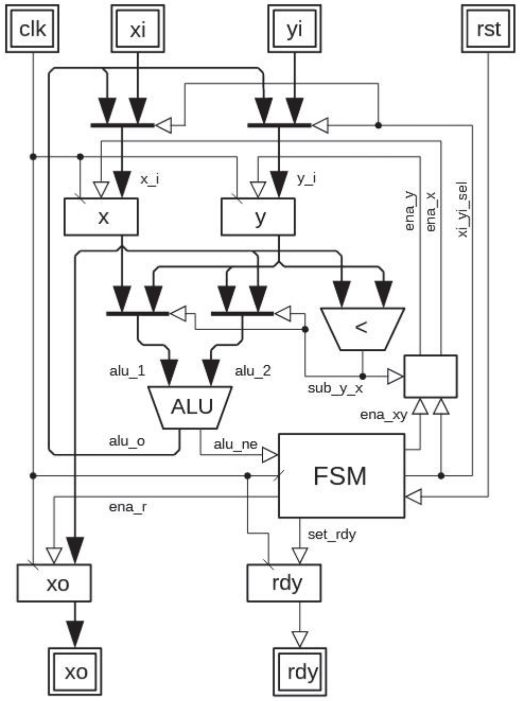
A datapath

Controlled by FSM



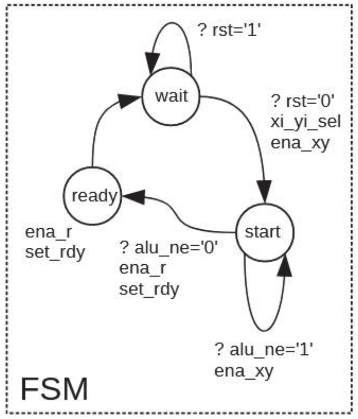
Gajski–Kuhn diagram

RTL examples – GCD architectures



Register-transfer level description with comparator controlling subtractions.

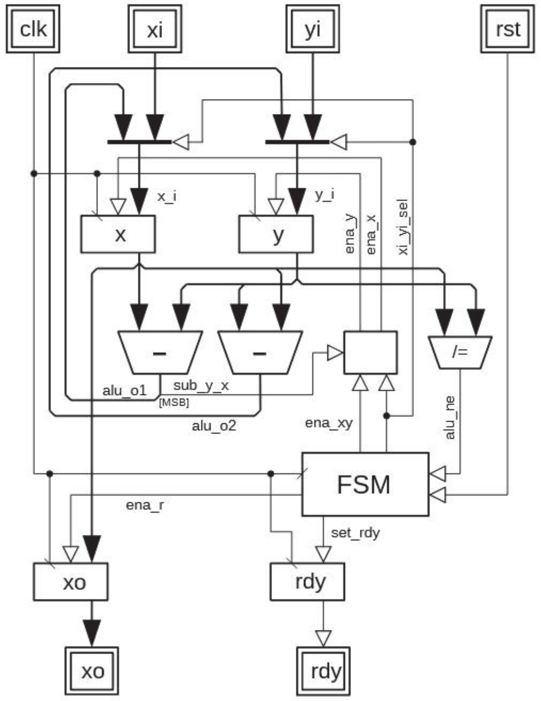
Default value for the control signals is '0'.



Code: gcd-rtl3.vhdl

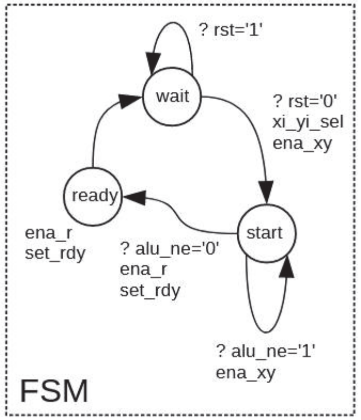
1 ALU (“–”, “/=”), 1 comparator
[1 clock step per iteration]

ASIC: 1134 e.g. / 20.0 ns
FPGA: 58 SLC / 17.0 ns



Register-transfer level description with out-of-order subtractions. Only data-path differs from RTL #3 & #4.

Default value for the control signals is '0'.



Code: gcd-rtl5.vhdl

2 subtracters, 1 comparator
[1 clock step per iteration]

ASIC: 915 e.g. / 20.0 ns
FPGA: 58 SLC / 8.0 ns