ELEC-H-409
Th04: State machines & complex design management

Dragomir Milojevic
Université libre de Bruxelles

Today

1. State machine: basic concepts
2. State machine: practical example
3. Typical circuits – adders
4. Resource sharing
5. Sub-programs, packages & libraries

1. State machine: basic concepts

Overview

Finite → finite # states

- State machines – mathematical model of sequential circuits; known in literature as Finite State Machines (FSM) – the system can be found in exactly one of a finite number of states at any given time
- Represented using state diagrams – easy way of representing sequential behavior as transition of states are shown graphically
- Or using state tables – indicate future state in tabular form, as a function of inputs and current state
- State tables are more convenient to use since they force us to think of **all possible evolutions** of the system given the present state; even more care should be taken when writing VHDL – as much as possible evolutions must be analyzed to validate the RTL model
- Assuming number of states N, the number of state variables is:

$$n = log_2 N$$

- Therefore more states mean more state variables, the less optimal system is (area, power, performance); hence we need optimization to reduce the number of states and thus simplify the logic circuit

State table

- We read current state in the column to the left in the state table – this is the present for the machine
- Next (future) states are given in the table depending on the current state & inputs change
- Assuming a present state and an input change, the future state is first decided (transition) and then reached (stable state here in bold); state tables "predicts" the future of the machine
- There could be an arbitrary number of transitions between two stable states
- Here the value of the output is set according to the current state – Moore state machine – the output is calculated using some combinatorial logic that simply decodes the current state

present state

2 inputs variable

4 states

	00	01	11	10	Q
1	**1**	2		3	0
2	1	**2**	4	3	0
3		5	**3**		0
4		**4**			1

Why and how you can reduce FSMs complexity?

Lo see ELEC-H310

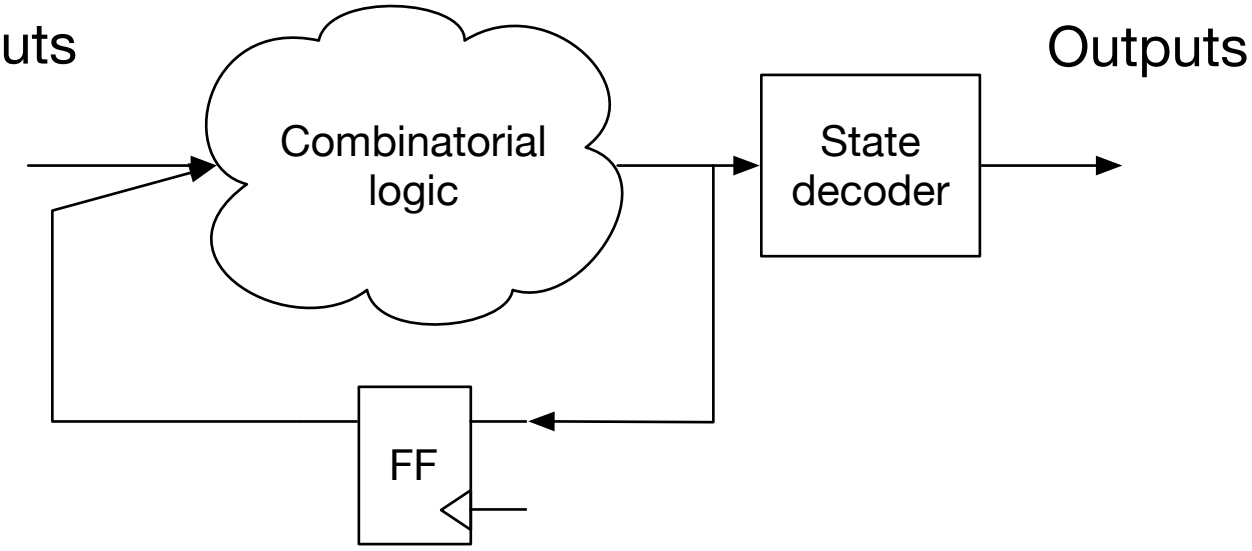
- State machine design begins with manual (verbal) description of the system that is first translated into initial state table; the purpose is to capture the exact behavior of the system
- This state table can have identical/redundant states, and hence may not be optimal for physical implementation
- Some states could possibly be interpreted as one; in another words they could be merged and this is very **hard to see** in advance
- This is because states could, but don't necessarily, have **verbal meaning** that humans "see" easily
- If there are two or more states that lead to the same evolution of the automaton, then they could be **merged**
- Initial, man made state tables could have a lot of such redundant states offering the room for state optimization
- Manual synthesis implies manual optimization of states

State machine classes

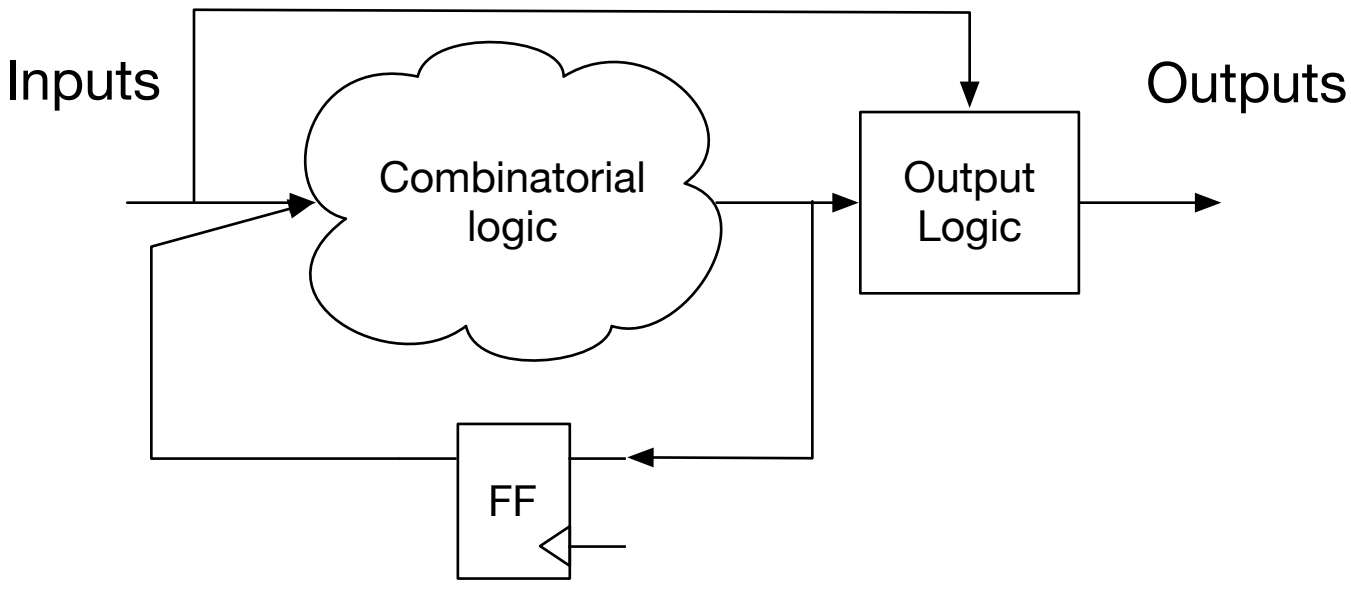
- Depending how the **system output** is calculated we distinguish between two types of sequential logic machines: Moore and Mealy
  - Moore – the output is function of system state only
    - Output needs only a simple decoding (combinatorial) circuitry
    - We can merge **only** states with the same output
    - Possibly less states could be merged, hence a priori more logic
  - Mealy – the output is function of system state & input
    - Output logic might be more complicated than in Moore
    - But we can now **merge states with different outputs**
    - Possibly more states could be merged requiring less logic
- Note that Mealy machines are not always better; final circuit complexity depends on the content of the state table itself

Schematic view of the difference

Moore



Mealy



State simplification with Moore & Mealy

Initial state table:

	00	01	11	10	Z
1	**1**	2	-	3	0
2	1	**2**	4	-	1
3	1	-	4	**3**	0
4	-	5	**4**	-	1
5	6	**5**	4	-	0
6	**6**	5	-	3	1

Moore:

	00	01	11	10	Z
1	**1**	2	-	1	0
2	1	**2**	3	-	1
3	**3**	4	**3**	1	1
4	3	**4**	3	-	0

Mealy:

	00	01	11	10
1	**1/0**	**1/1**	2	**1/0**
2	**2/1**	**2/0**	**2/1**	1

1 and 2 could be merged for Mealy, but not for Moore
States 1 and 6 can not be merged not even in Mealy
Can you explain?

Mealy vs. Moore – summarized differences

- Mealy machine use fewer states, because we can possibly merge more states, due to input dependency that allows several output values to be specified for the same state
- Note that it is not because there are fewer states that the corresponding circuit is necessarily more simple; we need to take output circuitry into account too
- Mealy machine respond faster – whenever the input changes for the same state, the output will follow (we don't have to wait for the state to update)
- But, Mealy machine may be transparent to glitches on inputs, i.e., they will pass them to the outputs directly
  - ▷ How could you prevent this?
- Moore machines are safer because they used registered values of the state; they are also easier to design

State encoding

- Optimized state table then undergoes the process of state encoding to enable implementation with Boolean logic
- Here we pick an arbitrary encoding: 1-00, 2-01, 3-11, 4-10

State table:

	00	01	11	10	Z
1	-	2	**1**	-	0
2	3	**2**	1	-	0
3	**3**	4	-	-	1
4	3	**4**	1	-	1

Encoded state table:

	00	01	11	10	Z
00	-	01	00	-	0
01	11	01	00	-	0
11	11	10	-	-	1
10	11	10	00	-	1

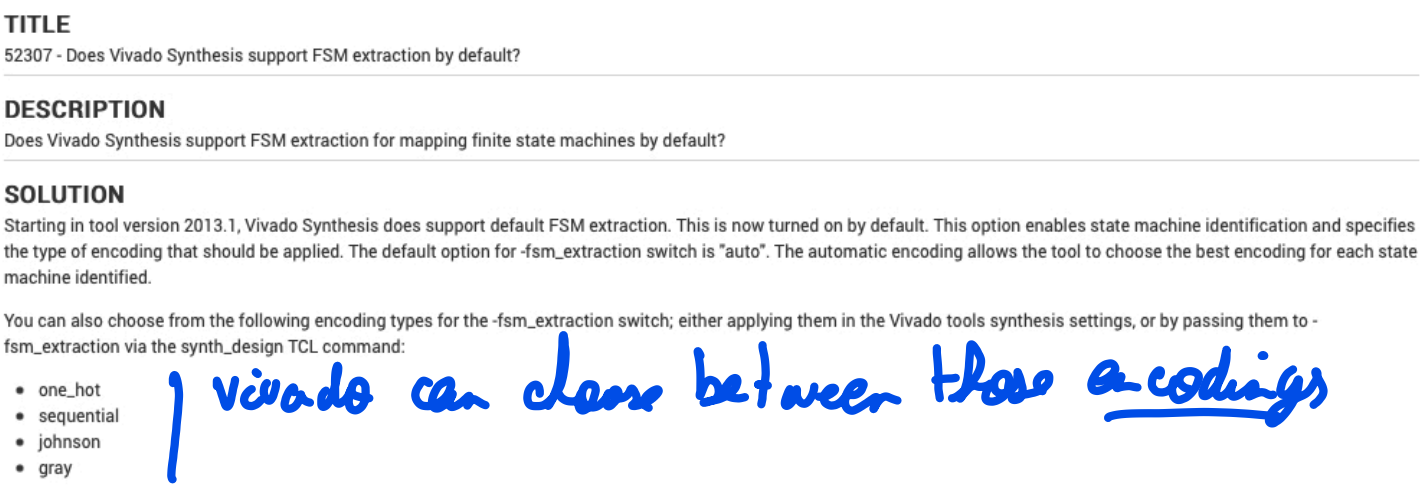
- In basic logic circuit courses we typically make a hypothesis that all state encodings are the same; reason why we picked an arbitrary one above

State encoding – do we have a choice?

- In theory FSM "speed" is **independent** of the number of states & number of transitions
- Most of the time transitions are assumed to be instantaneous, so arbitrary state encoding is ok
- In real circuits this is not the case!
- More states and transitions mean more logic, so slower logic circuits; this is why state optimization is so important
- But in real circuits state encoding choice could impact the performance of the logic circuit too
- Poorly encoded sequential machine may slow the circuit, and this will get worse with FSMs that have more states
- State encoding does influence HW complexity and hence impact final performance, area & power of the synthesized circuit

Options for state encoding

- If you are implementing state machines in some HDLs, and intend to build a circuit you will have to decide on the encoding scheme
- You can decide on the encoding by explicitly specifying the codes in your VHDL (see next chapter, last slide), or you can leave synthesis tool to perform state encoding for you in an automated fashion
  - ▹ Vivado synthesis tool will pick the best encoding for you if set leave the option to `auto` (default)



TITLE

52307 - Does Vivado Synthesis support FSM extraction by default?

DESCRIPTION

Does Vivado Synthesis support FSM extraction for mapping finite state machines by default?

SOLUTION

Starting in tool version 2013.1, Vivado Synthesis does support default FSM extraction. This is now turned on by default. This option enables state machine identification and specifies the type of encoding that should be applied. The default option for -fsm_extraction switch is "auto". The automatic encoding allows the tool to choose the best encoding for each state machine identified.

You can also choose from the following encoding types for the -fsm_extraction switch; either applying them in the Vivado tools synthesis settings, or by passing them to -fsm_extraction via the synth_design TCL command:

- one_hot
- sequential
- johnson
- gray

vivado can choose between those encodings

If FSM extraction needs to be disabled, it can be achieved by passing "off" to the -fsm_extraction switch.

FSM extraction was not turned on by default until 2012.4. This was turned off by default and the finite state machine HDL was synthesized as logic.

- Tool can pick between: sequential, random, optimized, gray, one-hot

Typical state encoding schemes

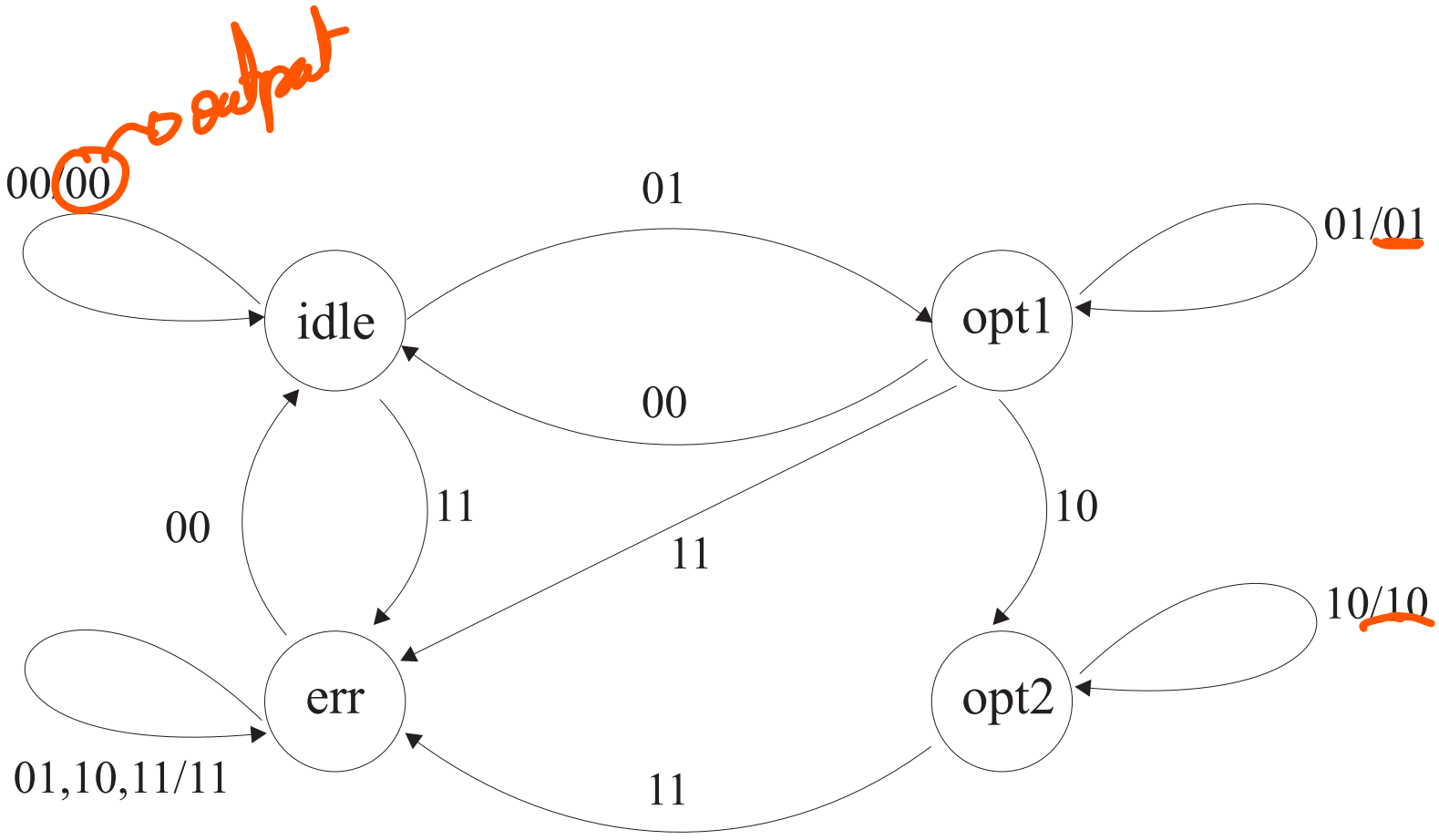
- Sequential – same order as given in the state table
- Random – as the name suggests
- Optimized – find encoding such that one of the following parameters is optimized: area, logic depth & fanout (performance), bit flips (power) etc.
- Gray – one bit changes at a time, safer for glitches, good for asynchronous implementation (counters), which we don't do here (not always possible; example: when transitions from all to all states)
- One-hot – one bit of the state vector per state
  - All other state bits are zero; n states result in n state FF example: 1,2,3,4 into 0001, 0010, 0100, 1000
  - Any transition involves two bit flip
  - One-hot encoding generates loads of bits for the state vector and we have a lot of unused combinations of the state variables (these will be filled with don't cares); but logic functions may be very simple

One-hot encoding advantages/disadvantages

- Faster than other encoding techniques
- State decoding is simplified, need only to check state bits to find the state (this can be done by simple priority encoder/decoder)
- Hence additional logic is not required for decoding, this is extremely advantageous when implementing big FSMs
- Low switching activity, hence lower dynamic power & less prone to glitches
- Adding/deleting states and changing state transitions is easy and does not affect the rest of the design
- Finding the critical path of the design is easier
- Particularly advantageous for FPGA implementations as FPGA have many FFs (one per LUT)
- Other encoding schemes might require additional logic, which in case of FPGA could be more expensive then more FFs
- Disadvantages: they consume area and power !!! (not really problem for FPGAs, FFS are there and FPGAs are **NOT** low-power devices)

2. State machine: practical example

FSM example: state graph

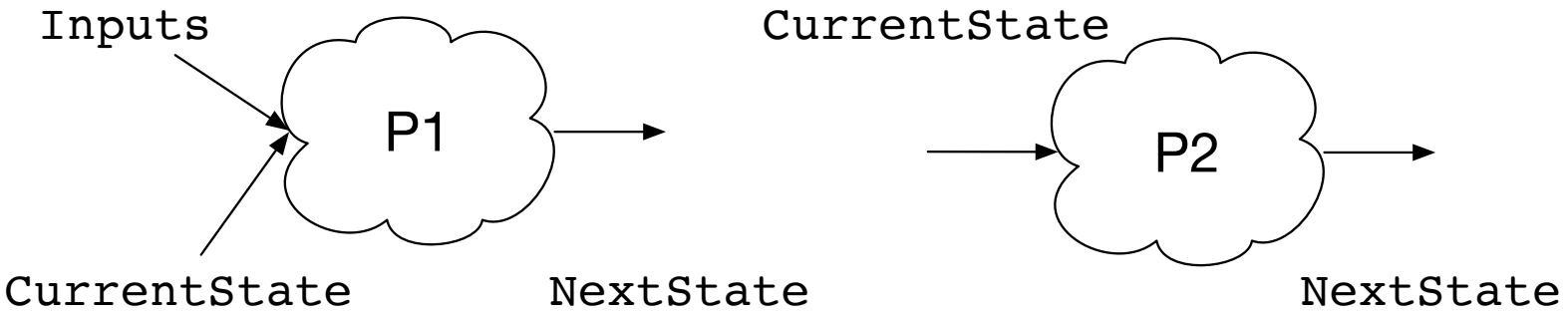


You should be able to translate this state graph into a state table

FSM: process view

In VHDL, FSMs are typically modeled using two distinct processes:

- First process (`P1`) controls the computation of the next state as a function of inputs and the present state; note that this one is a pure combinatorial process
- Second process (`P2`) updates the state variables that are stored in FFs to avoid race conditions; this process is and must be sequential, since we do want to infer FFs as synchronous storage elements to store the system state (these are needed to avoid race conditions)

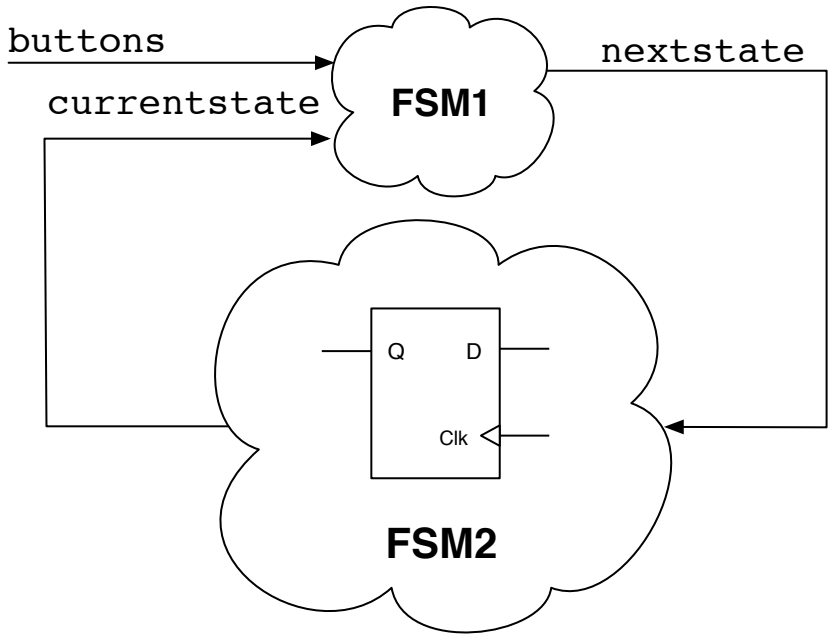


- Two processes communicate using shared signals: `CurrentState`

FSM template

- Both processes `P1` & `P2`, let's call them `FSM1` & `FSM2` are typically implemented in a single VHDL module:

similar to the drawing of a general sequential circuit.



- Note that there is a feedback loop on state variables! This is allowed since there is one FF for every state variable (`CurrentState,NextState` is a bus)
- How many memory elements you would expect in our example?

Complete model – entity & states

- States are typically defined using enumerated types
- During logic synthesis every state will get a unique binary code when state encoding occurs; this will be automated in Vivado

```
1  entity vending is port(
2    reset,clock: in std_logic;
3    buttons    : in std_logic_vector(1 downto 0);
4    lights     : out std_logic_vector(1 downto 0)
5  ); end vending;
6  architecture synthesis1 of vending is
7   type statetype is (idle, opt1, opt2, error);
8   signal currentstate, nextstate : statetype;
9   begin
10  -- Continues on next slide
```

-- enumerated data type

here : NOT encoded states

- Follow two processes:
  - `fsm1`: "sensitive" to inputs and the current system state
  - `fsm2`: "sensitive" to clock and reset; enable FSM to "move" in time by updating current with next state (state variables update)

Complete model – combinatorial process

- Moore doesn't depend on the input

```
1  -- Continued from the previous slide
2  begin
3   fsm1: process( buttons, currentstate )
4    begin
5     case currentstate is
6      when idle => lights <= "00";
7        case buttons is
8         when "00" => nextstate <= idle;
9         when "01" => nextstate <= opt1;
10        when "10" => nextstate <= opt2;
11        when others => nextstate <= error;
12        end case;
13     when opt1 => lights <= "01";
14        if buttons /= "01" then nextstate <= idle;
15        end if;
16     when opt2 => lights <= "10";
17        if buttons /= "10" then nextstate <= idle;
18        end if;
19     when error => lights <= "11";
20        if buttons = "00" then nextstate <= idle;
21        end if;
22  end case;
23  end process;
24 -- Continues on next slide
```

- output (see graph)
- "00" = input

⇒ results in combinatorial circuit after synthesis

Complete model – sequential process

```
1  -- Continued from previous slide
2
3  -- Second sequential process
4  -- Sensitive to reset and clock
5  -- We do want FFs to be inferred
6
7  fsm2: process(reset, clock)
8   begin
9    -- Note the reset and the initial state
10   -- whenever the FSM powers-up
11   -- it will find itself in the idle state
12   if (reset = '0') then
13       currentstate <= idle;
14          -- We rising edge: this is where synthesis infers FFs
15          elsif (clock'event) and (clock = '1') then
16             currentstate <= nextstate;
17   end if;
18 end process;
19 end synthesis1;
```

Deals of the state update makes the circuit sequential

1 FF

FSM implementation using one-hot encoding

```
1  type states is (idle, state1, state2, state3, state4, state5);
2  attribute enum_encoding : string;
3  attribute enum_encoding of states: type is "000001 000010 000100 001000 010000 100000";
4  -- Look at the codes above
5  signal present_state, next_state : states;
6  begin
7    comb_proc : process (present_state,start,branch,hold)
8    begin
9      case present_state is                                          -- state transitions
10       when idle =>                                                 -- when in idle
11         if (start = '0') then next_state <= state1;                -- go to state1
12          else next_state <= idle;                                  -- or stay in idle
13         end if ;
14       when state1 =>                                               -- in state 1
15         if (branch = '1') then next_state <= state4;               -- go to state4
16           else next_state <= state2;                               -- or to state2
17         end if ;
18       when state2 => next_state <= state3;                         -- go to state3
19       when state3 =>
20         if (hold = '0') then next_state <= state4;                 -- then go to state4
21            else next_state <= state3;                              -- or stay in state3
22         end if ;
23       when state4 => next_state <= state5;                         -- go to state5
24       when state5 => next_state <= idle;                           -- go to idle
25     end case;
26   end process comb_proc;
27   sync_proc : process (clk,reset)
28   begin
29   if (reset = '0') then present_state <= idle;                     -- async reset & idle
30     elsif (clk'event and clk = '0') then present_state <= next_state;   -- change state
31   end if ;
32 end process sync_proc;
```

explicit encoding

3. Typical circuits – adders

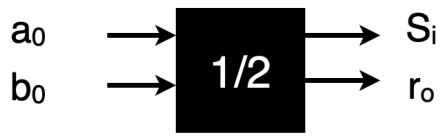
Adder implementations

- Any computational problem could be solved in different ways, in SW we speak of algorithms
  - Effective computational procedure, it defines in **detail** how computations are performed
  - Assumes certain computer architecture model
  - In general algorithm development assumes Von Neumann architecture: data & instructions are stored in the same memory, and machine executes one instruction at a time
- In HW we speak of micro-architecture
  - How a given algorithm is translated into HW using logic gates & FFs
  - And/or how it is decomposed in time (e.g. multiplication as a successive additions in time)
  - While design automation allows automated circuit synthesis, micro-architecture design can not be automatized
- Even simple operations could be implemented in different ways: fixed point integer addition with ripple carry orcarry look ahead

1. Ripple carry adder (1/2)

- The simplest of all to understand, but worse performance!
- Based on two elementary circuits: half-adder and full-adder
- It is easy to derive logic equations & implement them in VHDL

Half-adder



s_0	0	1	b_0
0	0	1	
1	1	0	

a_0

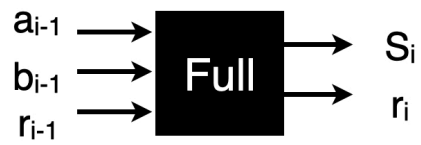
$S_0 = a_0'b_0 + a_0b_0' = a_0 \oplus b_0$

r_o	0	1	b_0
0	0	0	
1	0	1	

a_0

$r_0 = a_0\ b_0$

Full-adder



S	00	01	11	10	$a_{i-1}b_{i-1}$
0	0	1	0	1	
1	1	0	1	0	

r_{i-1}

$S_i = a_{i-1} \oplus b_{i-1} \oplus r_{i-1}$

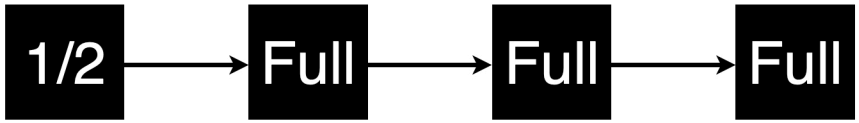
r_i	00	01	11	10	$a_{i-1}b_{i-1}$
0	0	0	1	0	
1	0	1	1	1	

r_{i-1}

$r_i = a_{i-1}b_{i-1} + r_{i-1}b_{i-1} + r_{i-1}a_{i-1}$

1. Ripple carry adder (2/2)

- To implement circuit with generic operands of n bits we could use:
  - 1 half-adder and $n-1$ full adders connected in series



- This is going to work just fine from the computational & logic perspective (perfect for simulation), but there is a BIG problem with this approach if it targets physical circuit implementation!
  - Ask yourself a question: what will happen if the operands are using words of say 256 bits?
  - Critical path will be VERY deep, thus the **propagation delay** is going to be HUGE and the circuit is going to be SLOW
  - Above approach is ok for simulation, but not good for implementation (no practical circuit will use this method)
- We should be able to do (much) better than this ...

2. Full combinatorial (1/3)

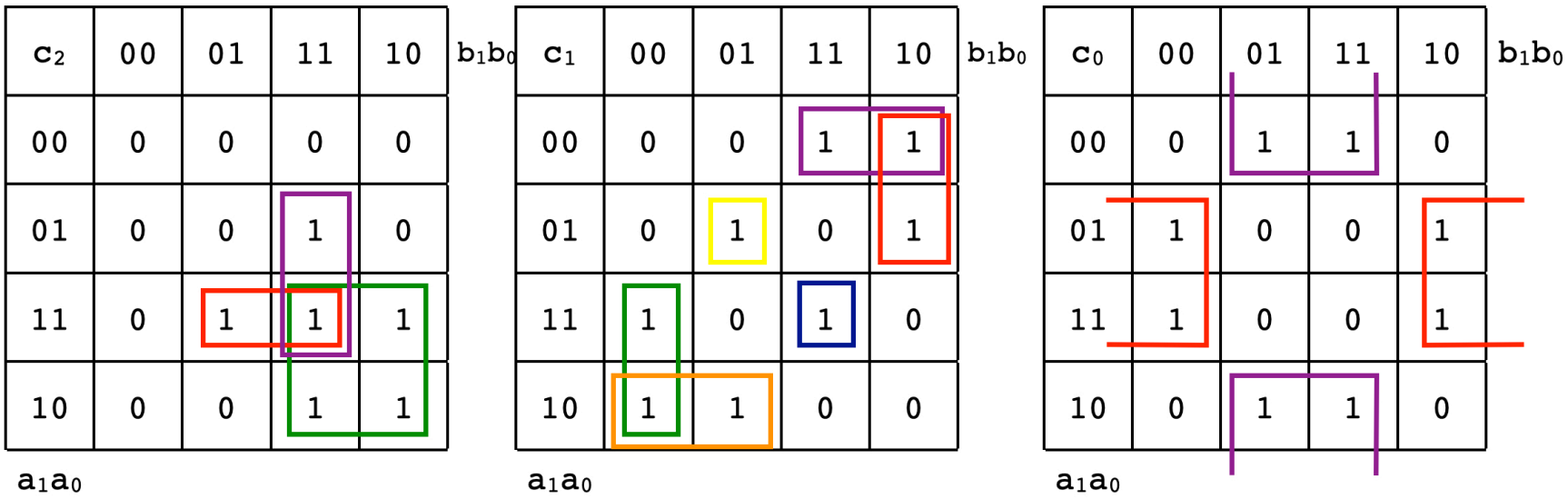
- You could consider addition as a fully **combinatorial** problem
- Input words are concatenated and considered to be one unique input to the system
- Example for 2 operands of 2 bits each; we have 3 bits of output, each being implemented using independent combinatorial circuit
- Computation of the carry bit is "embedded" into the combinatorial problem
- Note that in the table the line $(10)_2 + (11)_2 = (101)_2$ corresponds to: $(2)_{10} + (3)_{10} = (5)_{10}$

a1	a0	b1	b0	c2	c1	c0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

3. $2^4 = 48$ bits → way to much!!

2. Full combinatorial (2/3)

You can then extract the logic equations:

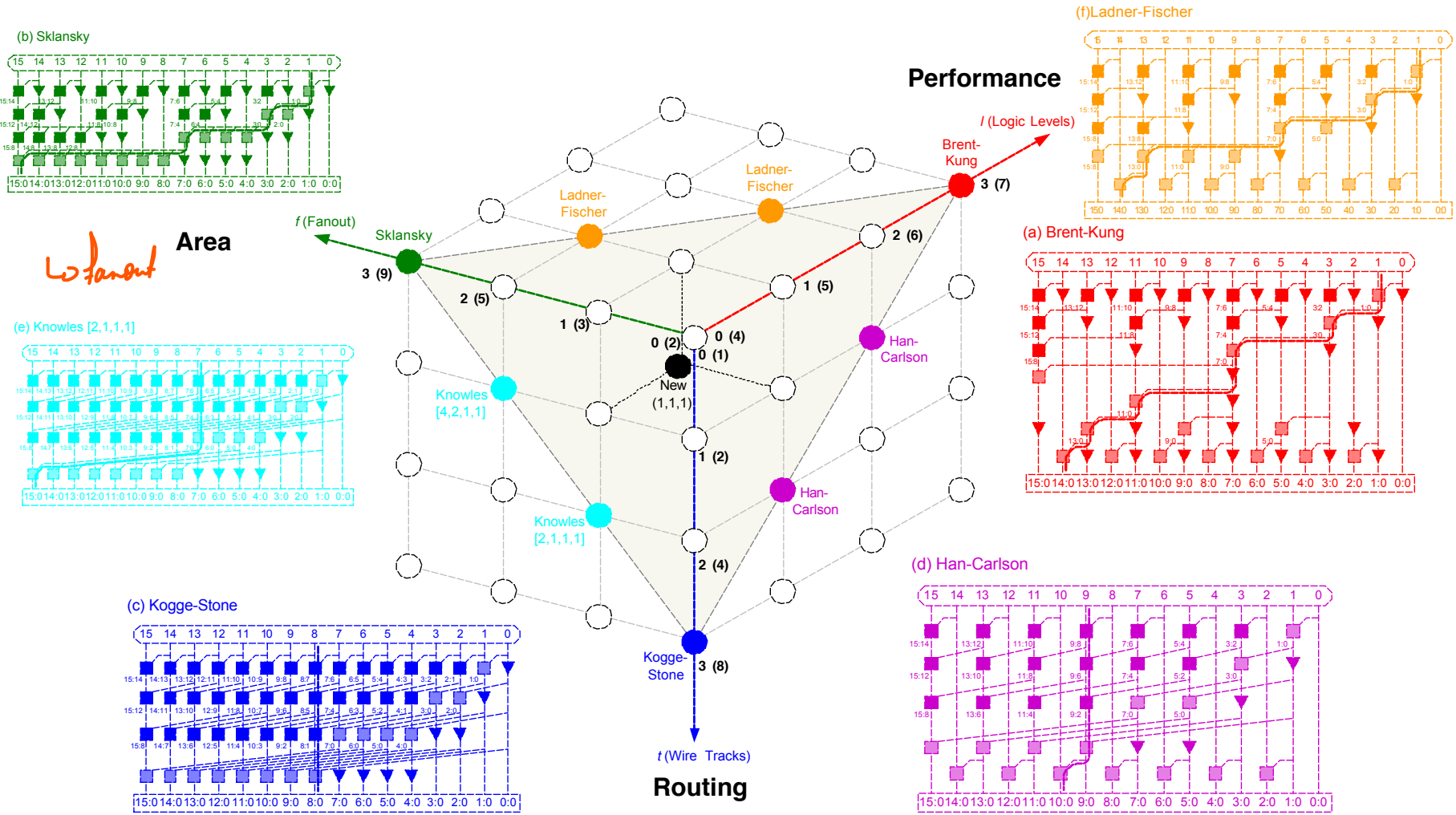


$$
\begin{aligned}
c_2 &= a_1 b_1 + a_1 a_0 b_0 + a_0 b_1 b_0 \\
c_1 &= a_1 b_1' b_0' + a_1 a_0' b_1' + a_1' a_0' b_1 + a_1' b_1 b_0' + a_1' a_0 b_1' b_0 + a_1 a_0 b_1 b_0 \\
c_0 &= a_0 b_0' + a_0' b_0
\end{aligned}
$$

2. Full combinatorial (3/3)

- Use the equations to implement the VHDL model
- What is THE consequence of this approach?
  Complexity of the circuit is a function of the word size
- More bits in the word, more complex the functions will be
- More complex functions mean more area, deeper the critical path, worse performance due to combinatorial explosion
- To circumvent the above real-world adder circuits use some kind of cary prediction, the computation is split into 3 phases:
  - Pre-processing – intermediate signals, function of operands only
    $G_i = A_i \cdot B_i$ and $P_i = A_i + B_i$
  - Carry look ahead network – uses previous results
    $C_{i+1} = G_i + (P_i \cdot C_i)$
  - Post processing – final result
    $S_i = P_i \oplus C_{i-1}$
- Depending on how the above is implemented – many adder flavors

Different adder variants : every adder has a certain cost for those 3 big characteristics/aspects

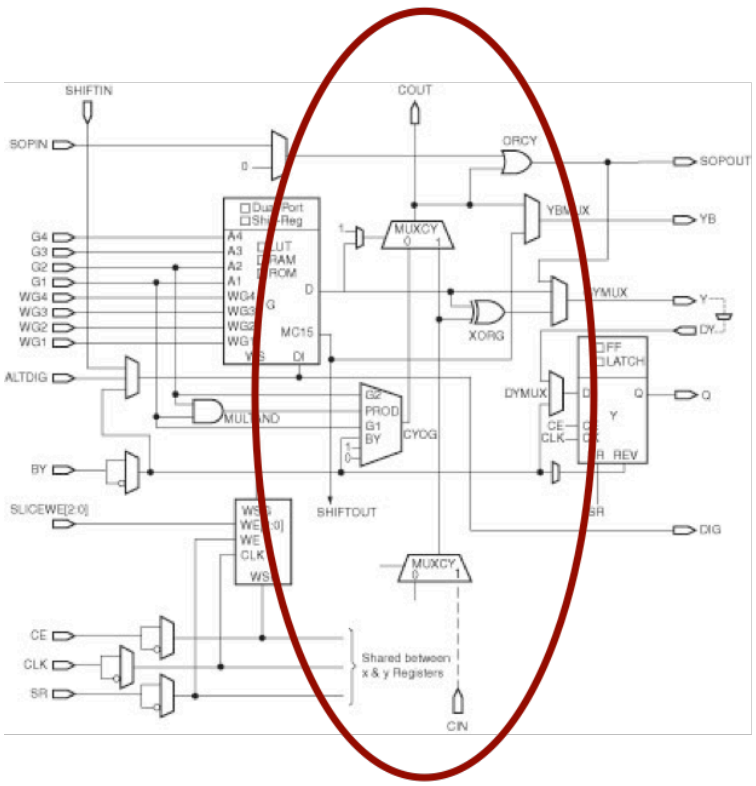


Digital circuits and their logic

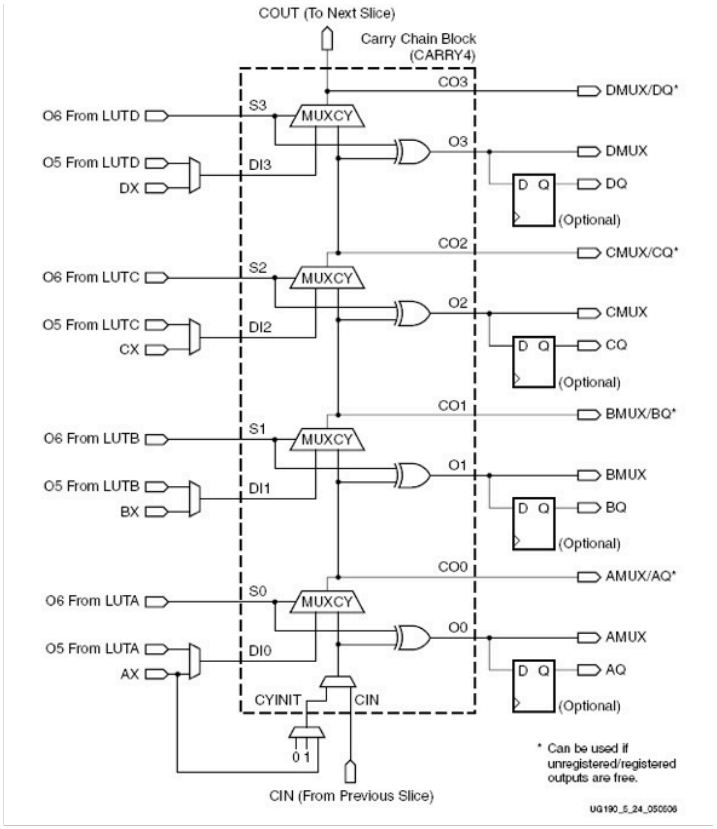
- Most of the time more complex designs are built upon:
  - Assembly of more or less complex arithmetical operators: add, subtract, multiply or divide & perform logical operations – this is why we speak of Arithmetical Logical Units (ALUs) in microprocessors, micro-controllers, DSPs GPUs, etc.
  - Some control logic that will drive the computation
  - Glue logic that will "connect" whatever needs to be logically connected
- When designing digital circuits you want to concentrate on the last two, and do not spend your time on the first!
- Most of the synthesis tools (both FPGA and ASIC) will have a certain "knowledge" on how to assemble these operators for you in most efficient way and depending on what you are asking for (as usual trade-off between area, power and performance)

Adders in Xilinx FPGAs: HW support at LUT level

Each LUT has dedicated logic to implement carry processing and connectivity to build adder chains:



At LUT level



At slice level

Adders in Xilinx FPGAs

beyond adders

Using Softcore(s): components that are already implemented in HW and available; Xilinx synthesis engine is aware of these component and may use them when you infer + operator on integer data types

Introduction

The Xilinx® LogiCORE™ IP Adder/Subtracter core provides LUT and single XtremeDSP™ slice add/sub implementations. The Adder/Subtracter module can create adders (A+B), subtracters (A–B), and dynamically configurable adder/subtracters that operate on signed or unsigned data. The function can be implemented in a single XtremeDSP slice or LUTs (but currently not a hybrid of both). The module can be pipelined.

Features

- Drop-in module for Virtex®-7 and Kintex™-7, Virtex-6, Virtex-5, Virtex-4, Spartan®-6, Spartan-3/XA, Spartan-3E/XA, Spartan-3A/3AN/3A DSP/XA FPGAs
- Backwards compatible with version 9.1
- Generates adder, subtracter and adder/subtracter functions
- Supports two's complement-signed and unsigned operations
- Supports fabric implementation inputs ranging from 1 to 256 bits wide
- Supports XtremeDSP slice implementations with inputs ranging from 1 to 36 or 48 bits wide (varies with device family)
- Optional carry input and output
- Optional clock enable and synchronous clear
- Optional bypass (load) capability
- Option to set the B Value to a constant
- Optional pipelined operation

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family(1)	Virtex-7 and Kintex-7, Virtex-6, Virtex-5, Virtex-4, Spartan-6, Spartan-3/XA, Spartan-3E/XA, Spartan-3A/3AN/3A DSP/XA				
Supported User Interfaces	Not Applicable				
	Resources(2)				**Frequency**
Configuration	**LUTs**	**FFs**	**DSP Slices**	**Block RAMs**	**Max. Freq.**
Virtex-5, 32-bit input width, latency = 3	70	91	0	0	410 MHz
Provided with Core					
Documentation	Product Specification				
Design Files	Netlist				
Example Design	Not Provided				
Test Bench	Not Provided				
Constraints File	Not Applicable				
Simulation Model	VHDL behavioral model in the xilinxcorelib library VHDL UniSim structural model Verilog UniSim structural model				
Tested Design Tools					
Design Entry Tools	CORE Generator tool 13.1 System Generator for DSP 13.1				
Simulation	Mentor Graphics ModelSim 6.6d Cadence Incisive Enterprise Simulator (IES) 10.2 Synopsys VCS and VCS MX 2010.06 ISIM 13.1				
Synthesis Tools	N/A				
Support					
Provided by Xilinx, Inc.					

4. Resource sharing

Circuits and their complexity 1/2

- These days typical cores could contain hundreds of adder circuits!
- Adders are not only found in ALUs, they are almost everywhere, think of address manipulation in any computer for example
- Hence we want them to be as **efficient** as possible, so best possible performance, for smallest area & minimal power dissipation
- But as always you can not gain on all parameters: depending on the target application, designer will choose the micro-architecture that suit best the needs and hence will have to trade-off performance, area, power; just like different adders we saw in the previous chapter
- Golden rule: high-performance components take area and burn power (read they are costly), while low-power ones are designed with low-power in mind and will sacrifice performance to make savings in area & power

Circuits and their complexity 2/2

- In general we want to keep the number of computational resources **to bare minimum** to save area & thus the system cost, which is always the most important parameter no matter if you are in high-performance or low-power computing
- To minimize the number of the instantiated components we can use resource sharing
- The idea is to use the same physical component (module instance) for computations that could potentially come from 2 different places; the computations should of course occur in two different moments in time
- In general ALUs/adders are fast, and they are not necessarily the bottleneck in current computing systems; this motivate the re-use of these resources
- To enable resource sharing we need to implement control logic that manages the usage of these resources trough HW controlled muxes; this control will occur at run-time

Concrete example

- You want to make an ALU module with couple of basic arithmetic operations (addition & subtraction)
- Model below will work in simulation, and would result in a synthesizable and implementable logic circuit

```
1  alu: process (operand0, operand1, operand2, opcode)
2  begin
3    case opcode is
4      when add_operand1 =>
5        result <= operand0 + operand1;    -- 1 Add operands a and b
6      when sub_operand1 =>
7        result <= operand0 - operand1;    -- 2
8      when add_operand2 =>
9        result <= operand0 + operand2;    -- 3 Add operands a and c
10     when sub_operand2 =>
11       result <= operand0 - operand2;    -- 4
12   end case;
13 end process;
```

- Synthesis could produce 4 instances of ADD/SUB circuits
- Problem – it is difficult for synthesis tool to figure out that a resource of the adder used for additions in steps 1 and 3 could be potentially the same instance

Why not ... but only if needed

- Previous approach may not be efficient!
- Unless you provide means to keep the above ALU busy with all 4 arithmetic operations at the same time
  - In some cases you might need 4 operations in parallel, but that is far from a general case (look at VLIW computer architectures)
  - In the model above parallel execution is not possible since `case opcode` has only one possible outcome at a time
  - Can you explain what needs to be done to enable the above?
- If we can not use all 4 arithmetic circuits at the same time, we could instantiate **only one ADD/SUB circuitry** and share it
- But in order this to work, you need to do some manual "micro-architectural" tuning
- Some synthesis tools are very smart, but still do not expect miracles: designers are those that decide on micro-architecture, not the tools!

ALU with a shared resource

- **Focus on control logic** and prepare correct data for the input of a single ADD/SUB circuitry
- We make operand **preselection** – pure combinatorial circuit, just adds a bit more delay to the data-path, latency remains the same

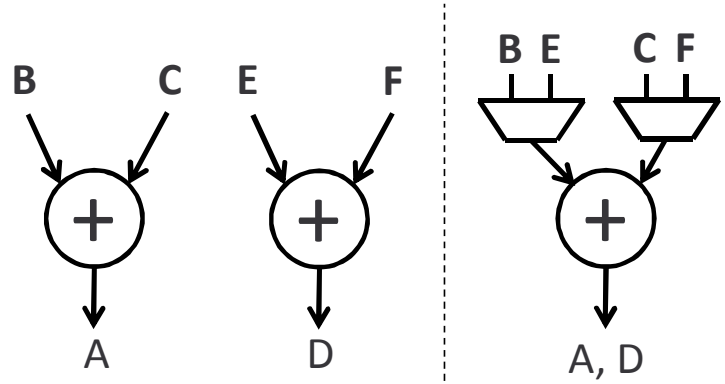
```
1  p1: process (opcode)
2    begin
3    case opcode is
4      when add_operand1 =>          -- 1
5        add_sub <=    1   ;
6        b <= operand1;
7      when sub_operand1 =>          -- 2
8        add_sub <=    0   ;
9        b <= operand1;
10     when add_operand2 =>          -- 3
11       add_sub <=    1   ;
12       b <= operand2;
13     when sub_operand2 =>          -- 4
14       add_sub <=    0   ;
15       b <= operand2;
16     end case;
17 end process;
```

- Draw conceptual schematic of this circuit and compare it with the previous implementation

Resource sharing

- Difference in two approaches is illustrated below: 2 additions, 2 different implementations

```
A = B + C;
D = E + F;
```

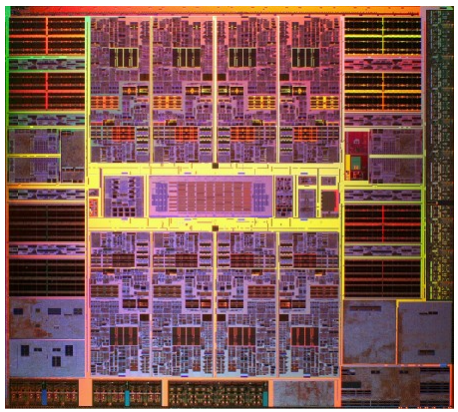


- If no sharing, we have more resources (2 adders), but these **could and should be used concurrently** and will have higher data throughput, so better performance
- If system specification explicitly states that no sharing could occur (computation is never concurrent), then resource sharing reduces the implementation area (cost) & power

5. Sub-programs, packages & libraries

Purpose

- We know that current CMOS technologies (5nm) used for ASIC/FPGA manufacturing offer very high transistor density
  - ▷ Apple M1 Pro chip (2021) counts 60 billions transistors that translate into millions of logic gates in a single IC package
- Question is how to manage such designs @ RTL-level? And enable VHDL re-usability, readability, de-bug, work distribution between teams of people etc.



  - ▷ Concrete example: OpenSPARC T2 System-on-Chip with 8 cores for an equivalent of 2M gates is implemented with **2283 VHDL files** that account for 1.5M lines of VHDL!

- To manage complex designs VHDL introduces sub-programs, packages & libraries

Sub-programs

- Sub-programs are just like the normal processes, but instead of being instantiated only once & within a current module, these are stored in a library, so that they can be shared & re-used by different modules in a design
- Statements in sub-programs are "executed" sequentially just like in any VHDL process (we know why I put quotes on "executed")
- There are two "sub-program" flavors:
  1) procedures – **without** return value &
  2) functions – **with** return value
- These names are the same as in the SW world, but in VHDL & HW they have a very different meaning!
- In SW these are resolved at run-time, while in VHDL procedures & functions are resolved at synthesis time
  - In VHDL a **function is not called**, rather it is **instantiated** within a module during logic synthesis of the circuit (this is close to concept of a macro or in-line functions in C language)

Sub-programs: implementation

- Procedures & functions are implemented in VHDL using:
  - declaration – used for interface definition and
  - body – implementation (sequence of statements)
- Example:

```
1 procedure procedure_name (parameter_list) is -- no return value
2    declarations
3 begin
4    -- sequential statements go here
5   ...
6 end procedure_name;
```

```
1 function function_name (parameter_list) return type is -- with return value
2    declarations
3 begin
4    -- sequential statements go here
5   ...
6 end function_name;
```

- This is the same as when implementing any VHDL module

Procedure & function; example

```
1  procedure display_mux ( alarm_time, current_time : in digit;
2                          show_a   : in std_ulogic;
3                          signal display_time : out digit) is
4    begin
5      if (show_a = '1') then
6        display_time <= alarm_time;
7          else
8            display_time <= current_time;
9      end if;
10 end display_mux;
11
12   ...
13
14   -- somewhere else in your VHDL you use the function procedure
15   display_mux(t1, t2, b, t_out);
16   -- explain the data types for t1, t2, b, t_out
```

```
1  function max (L, R:integer) return integer is
2    begin
3      if L > R then
4        return L; -- here we "return" something
5      else
6        return R;
7        end if;   -- here we "return" something
8  end max;
9
10   ...
11
12   -- somewhere else in your VHDL you assign return value to something
13   val <= max(a,b);
```

Packages

- Packages are used to **encapsulate** things for structured designs; they are **collections** of personal data types, constants, sub-programs (so functions & routines)
- Defined in two parts (once again): package declaration (i.e. various declarations) & package body (i.e. implementation)

```
1  -- Package declaration
2  package package_name is
3    -- Declaration of types and subtypes, subprograms, constants, signals etc.
4    ...
5  end package_name;
6  -- Package body
7  package body package_name is
8    -- Definition of previously declared constants subprograms
9    -- Declaration/definition of additional:
10     -- types, subtypes
11     -- sub-programs
12     -- constants, signals and shared variables ...
13 end package_name;
```

- Multiple packages can be stored in the same library; context clause `use` allows to select the appropriate package within the VHDL module

Example of a package

```
1  package simple is                                  -- Package definition
2   constant cdiff: integer;
3   subtype word is bit_vector(15 downto 0);
4   subtype add is bit_vector (24 downto 0);
5   type mem is array(0 to 31) of word;
6   function add2int(val : add) return integer;
7   function incr_word(val : word) return word;
8  end simple;
9
10 package body simple is                              -- Package body
11  constant cdiff: integer:= 500;
12
13  function add2int(value : add) return integer is
14     ...  -- the definition of the function
15  end address2int;
16
17  function incr_word( val : word) return word is
18     .... -- the definition of the function
19  end increment_word;
20 end simple;
```

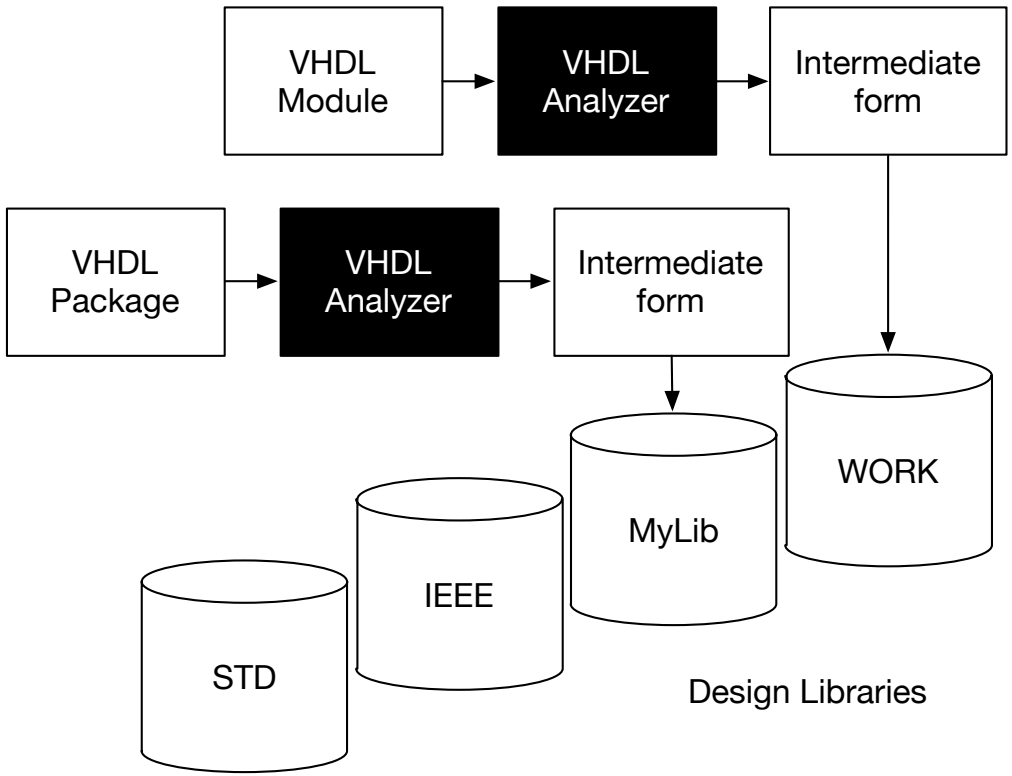
```
1  use work.simple.all ;                              -- We point out the package we use
2
3  -- Typical module definition & implementation statements
4  ...
5  -- Below types are defined in the package
6  variable pc: add;
7  variable a: word;
```

Design synthesis with packages

- If a module uses something that is defined in some package, the package needs to be explicitly synthesized before
- Typically the synthesis engine will store this information at some "known place"
- When a module to be synthesized references the package, the tool will expect to find this information at this "known place"
- Library is nothing else but this known place (so, a path from OS perspective) where synthesis engine stores & looks for information about a design: everything goes there!
- For design modules this path is often called `work` (e.g. MentorGraphics simulation tools such as ModelSim or QuestaSim)
- Note that synthesis tools can figure the inter-module dependency, so the file order in which modules are synthesized is not important (i.e. dependences are resolved automatically)
- For packages this is not the case, they need to be synthesized first

Design synthesis with packages in practice

- Each VHDL package **HAVE to be first converted** into **library**; synthesis engine performs this "conversion"
- Each design module is analysed and placed in a design library
- Libraries can be and are synthesised separately (for sharing)



Example of pre-defined & user defined libraries

- Once a library is declared, all of the sub-programs, user defined data types etc. from this library become visible to the module that issued the `use` clause
- You already used this approach with IEEE library to access basic data types & operators
- In the module below we add our stuff (`my_lib` from slide 48)

```
1  library ieee;                          -- This one we know and use all the time
2  use ieee.std_logic_1164.all;           -- This is a context clause for this module
3  -- use ieee.std_logic_signed.all;      -- You could use this one alternatively
4
5  library my_lib;                        -- This is your stuff pre-synthesized
6  use my_lib.my_pack_a.all;
7
8  entity arith is end entity;
9
10 architecture arch of arith
11   signal a, b, c: word;                -- Where does word comes from?
12 end architecture arch;
```

- Context clauses only apply to the following design module; if multiple entities are defined in a single VHDL file, they need to be re-defined before `entity` declaration