ECE 327: Digital Systems Engineering
Lecture Slides

2020t1 (Winter)

Mark Aagaard

University of Waterloo
Department of Electrical and Computer Engineering
# Contents

## 1 Fundamentals of VHDL

1.1 Introduction to VHDL ........................................ 22
  1.1.1 Levels of Abstraction .................................. 22
  1.1.2 VHDL Origins and History ............................... 23
  1.1.3 Semantics .................................................... 24
  1.1.4 Synthesis of a Simulation-Based Language .............. 26
  1.1.5 Solution to Synthesis Sanity ............................ 27
  1.1.6 Standard Logic 1164 ...................................... 28
1.2 Comparison of VHDL to Other Hardware Description Languages 29
1.3 Overview of Syntax ........................................... 29
  1.3.1 Syntactic Categories ..................................... 29

1.3.2 Library Units ............................................... 29
1.3.3 Entities and Architecture ................................. 30
1.3.4 Concurrent Statements .................................... 33
1.3.5 Component Declaration and Instantiations ............... 36
1.3.6 Processes ..................................................... 36
1.3.7 Generate Statements ...................................... 41
1.3.8 Sequential Statements .................................... 42
1.3.9 A Few More Miscellaneous VHDL Features ............... 43

1.4 Concurrent vs Sequential Statements ........................ 43
  1.4.1 Concurrent Assignment vs Process ....................... 44
  1.4.2 Conditional Assignment vs If Statements ............... 45
  1.4.3 Selected Assignment vs Case Statement .................. 46
  1.4.4 Coding Style ............................................... 47

1.5 Overview of Processes ....................................... 48
  1.5.1 Combinational Process vs Clocked Process .............. 52
  1.5.2 Latch Inference ............................................ 59

1.6 VHDL Execution: Delta-Cycle Simulation .................. 64
  1.6.1 Simple Simulation ......................................... 64
  1.6.2 Temporal Granularities of Simulation .................... 65
  1.6.3 Zero-Delay Simulation ..................................... 66
  1.6.4 Intuition Behind Delta-Cycle Simulation ................ 67
    1.6.4.1 Introduction to Delta-Cycle Simulation ............. 67
    1.6.4.2 Intuitive Rules for Delta-Cycle Simulation ........ 68
    1.6.4.3 Example of Delta: Buffers ............................ 69
    1.6.4.4 Example of Delta and Proj: Buffers .................. 69
    1.6.4.5 Example of Proj Asn: Flip-Flops ..................... 70
    1.6.4.6 Example of Delta and Proj: Comb Loop ............... 71
  1.6.5 VHDL Delta-Cycle Simulation ............................. 78
    1.6.5.1 Informal Description of Algorithm .................... 79
    1.6.5.2 Example: VHDL Sim for Buffers ...................... 80
    1.6.5.3 Definitions and Algorithm ........................... 81
    1.6.5.4 Example: Delta-Cycle for Flip-Flops ................. 83
    1.6.5.5 Ex: VHDL Sim of Comb Loop ........................... 84
    1.6.5.6 Rules and Observations for Drawing Delta-Cycle Simulations .................................................. 86
  1.6.6 External Inputs and Flip-Flops .......................... 88

1.7 Register-Transfer-Level Simulation ......................... 90
  1.7.1 Overview ................................................... 90
  1.7.2 Technique for Register-Transfer Level Simulation ..... 92
  1.7.3 Examples of RTL Simulation ................................ 93
4 State Machines

4.1 Notations ........................................... 192
4.2 Finite State Machines in VHDL ....................... 193
  4.2.1 HDL Coding Styles for State Machines .......... 193
  4.2.2 State Encodings ................................ 194
  4.2.3 Traditional State-Machine Notation .......... 195
  4.2.4 Our State-Machine Notation ................. 196
  4.2.5 Bounce Example ................................ 197
  4.2.6 Registered Assignments ..................... 202
  4.2.7 More Notation ................................ 205
    4.2.7.1 Extension: Transient States .......... 205
    4.2.7.2 Assignments within States ........ 207
    4.2.7.3 Conditional Expressions ............ 210
    4.2.7.4 Default Values ...................... 211
  4.2.8 Semantic and Syntax Rules .................. 218
  4.2.9 Reset ........................................ 224
4.3 LeBlanc FSM Design Example ......................... 228
  4.3.1 State Machine and VHDL .................... 229
  4.3.2 State Encodings ............................. 232
4.4 Parcels ......................................... 239
  4.4.1 Bubbles and Throughput ..................... 240
  4.4.2 Parcel Schedule .............................. 245
  4.4.3 Valid Bits .................................. 247
4.5 LeBlanc with Bubbles ................................ 251
4.6 Pseudocode ....................................... 253
4.7 Interparcel Variables and Loops .................... 255
  4.7.1 Introduction to Looping Le Blanc ........... 255
  4.7.2 Pseudo-Code ................................ 256
  4.7.3 State Machine ................................ 258
  4.7.4 VHDL Code for Loop and Bubbles ............ 260
4.8 Memory Arrays and RTL Design ....................... 262
  4.8.1 Memory Operations .......................... 262
  4.8.2 Memory Arrays in VHDL ..................... 265
  4.8.3 Using Memory ................................ 266
    4.8.3.1 Writing from Multiple Vars ........ 267
    4.8.3.2 Reading from Memory to Multiple Variables 268
    4.8.3.3 Example: Maximum Value Seen so Far .... 270
  4.8.4 Build Larger Memory from Slices ........... 273
  4.8.5 Memory Arrays in High-Level Models .......... 274

5 Dataflow Diagrams .................................. 275
5.1 Dataflow Diagrams ................................ 276
  5.1.1 Dataflow Diagrams Overview ................. 276
  5.1.2 Dataflow Diagram Execution .................. 284
  5.1.3 Dataflow Diagrams, Hardware, and Behaviour .. 286
  5.1.4 Performance Estimation ..................... 290
  5.1.5 Area Estimation ................................ 291
  5.1.6 Design Analysis ................................ 293
5.2 Design Example: Hnatysynch DFD ..................... 298
  5.2.1 Requirements .................................. 298
  5.2.2 Data-Dependency Graph ....................... 299
  5.2.3 Initial Dataflow Diagram ..................... 300
  5.2.4 Area Optimization ............................ 301
  5.2.5 Assign Names to Registered Signals .......... 302
  5.2.6 Allocation .................................... 305
  5.2.7 State Machine ................................ 312
  5.2.8 VHDL Implementation .......................... 316
5.3 Design Example: Hnatysynch with Bubbles ............ 321
  5.3.1 Adding Support for Bubbles .................. 322
  5.3.2 Control Table with Valid Bits ................ 326
  5.3.3 VHDL .......................................... 328
5.4 Inter-Parcel Variables: Hnatysynch with Internal State ....... 331
  5.4.1 Requirements and Goals ...................... 332
  5.4.2 Dataflow Diagrams and Waveforms ............ 333
  5.4.3 Control Tables ................................ 337
  5.4.4 VHDL Implementation .......................... 339
  5.4.5 Summary of Bubbles and Inter-Parcel Variables .... 341
5.5 Design Example: Vanier ............................ 342
  5.5.1 Requirements .................................. 343
  5.5.2 Algorithm ...................................... 344
  5.5.3 Initial Dataflow Diagram .................... 345
  5.5.4 Reschedule to Meet Requirements ............. 346
  5.5.5 Optimization: Reduce Inputs ................. 348
  5.5.6 Allocation ...................................... 350
  5.5.7 Explicit State Machine ...................... 352
  5.5.8 VHDL #1: Explicit ......................... 353
  5.5.9 VHDL #2 ....................................... 357
  5.5.10 Notes and Observations ..................... 359
5.6 Memory Operations in Dataflow Diagrams ............... 361
5.7 Data Dependencies ............................... 366
5.8 Example of DFD and Memory ........................ 371
Chapter 1

Fundamentals of VHDL
1.1 Introduction to VHDL

1.1.1 Levels of Abstraction

Transistor  Signal values and time are continuous (analog). Each transistor is modeled by a resistor-capacitor network.

Switch  Time is continuous, but voltage may be either continuous or discrete. Linear equations are used.

Gate  Transistors are grouped together into gates. Voltages are discrete values such as 0 and 1.

Register transfer level  Hardware is modeled as assignments to registers and combinational signals. Basic unit of time is one clock cycle.

Transaction level  A transaction is an operation such as transferring data across a bus. Building blocks are processors, controllers, etc. VHDL, SystemC, or SystemVerilog.

Electronic-system level  Looks at an entire electronic system, with both hardware and software.

1.1.2 VHDL Origins and History

VHDL = VHSIC Hardware Description Language
VHSIC = Very High Speed Integrated Circuit

The VHSIC Hardware Description Language (VHDL) is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis and testing of hardware designs, the communication of hardware design data, and the maintenance, modification, and procurement of hardware.


VHDL is a lot more than synthesis of digital hardware

1.1.3 Semantics

The original goal of VHDL was to simulate circuits. The semantics of the language define circuit behaviour.

But now, VHDL is used in simulation and synthesis. Synthesis is concerned with the structure of the circuit.

Synthesis: converts one type of description (behavioural) into another, lower level, description (usually a netlist).

Synthesis vs Simulation

For synthesis, we want the code we write to define the structure of the hardware that is generated.

The VHDL semantics define the behaviour of the hardware that is generated, not the structure of the hardware.
1.1.4 Synthesis of a Simulation-Based Language

This section reserved for your reading pleasure

1.1.5 Solution to Synthesis Sanity

- Pick a high-quality synthesis tool and study its documentation thoroughly
- Learn the idioms of the tool
- Different VHDL code with same behaviour can result in very different circuits
- Be careful if you have to port VHDL code from one tool to another
- KISS: Keep It Simple Stupid
  - VHDL examples will illustrate reliable coding techniques for the synthesis tools from Synopsys, Mentor Graphics, Altera, Xilinx, and most other companies as well.
  - Follow the coding guidelines and examples from lecture
  - As you write VHDL, think about the hardware you expect to get.

**Note:** If you can’t predict the hardware, then the hardware probably won’t be very good (small, fast, correct, etc)

1.1.6 Standard Logic 1164

std_logic_1164: IEEE standard for signal values in VHDL.

- `'U'` uninitialized
- `'X'` strong unknown
- `'0'` strong 0
- `'1'` strong 1
- `'Z'` high impedance
- `'W'` weak unknown
- `'L'` weak 0
- `'H'` weak 1
- `'-'` don’t care

The most common values are: `'U'`, `'X'`, `'0'`, `'1'`.

If you see `'X'` in a simulation, it usually means that there is a mistake in your code.

1.2 Comparison of VHDL to Other Hardware Description Languages

1.3 Overview of Syntax

1.3.1 Syntactic Categories

1.3.2 Library Units

This section reserved for your reading pleasure
1.3.3 Entities and Architecture

Each hardware module is described with an Entity/Architecture pair

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

entity and_or is
  port (
    a, b, c : in std_logic ;
    z      : out std_logic
  );
end entity;
```

Example of an entity

```
architecture main of and_or is
  signal x : std_logic;
begin
  x <= a and b;
  z <= x or (a and c);
end architecture;
```

Example of architecture

1.3.4 Concurrent Statements

- An architecture contains concurrent statements
- Concurrent statements execute in parallel
  - Concurrent statements make VHDL fundamentally different from most software languages.
  - Hardware (gates) naturally execute in parallel — VHDL mimics the behaviour of real hardware.
  - At each infinitesimally small moment of time, in parallel, every gate:
    1. samples its inputs
    2. computes the value of its output
    3. drives the output
Concurrent Statements

The order of concurrent statements doesn’t matter

Types of Concurrent Statements

Example Process with Sensitivity List

process (a, b, c)
begin
  y <= a and b;
  if (a = '1') then
    z1 <= b and c;
    z2 <= not c;
  else
    z1 <= b or c;
    z2 <= c;
  end if;
end process;
Example Process with Wait Statements

process
begin
wait until rising_edge(clk);
if (a = '1') then
  z <= '1';
  y <= '0';
else
  y <= a or b;
end if;
end process;

Sensitivity Lists and Wait Statements

- Processes must have either a sensitivity list or at least one wait statement on each execution path through the process.
- Processes cannot have both a sensitivity list and a wait statement.

The sensitivity list contains the signals that are read in the process.

A process is executed when a signal in its sensitivity list changes value.

An important coding guideline to ensure consistent synthesis and simulation results is to include all signals that are read in the sensitivity list.

There is one exception to this rule: for a process that implements a flip-flop with an if rising_edge statement, it is acceptable to include only the clock signal in the sensitivity list — other signals may be included, but are not needed.

1.3.6 Processes

Sensitivity List

1.3.7 Generate Statements

- Two categories of generate statements:
  - if-generate: conditionally generate some hardware
  - for-generate: generate multiple copies of some hardware

- Generate statements are executed during elaboration (at compile time)
- The conditions and loop ranges must be static
  - Must be able to be evaluated at elaboration
  - Must not depend upon the value of any signal
- A generate statement must be preceded by a label
1.3.8 Sequential Statements

Used inside processes, functions, and procedures.

<table>
<thead>
<tr>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait until ...;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>signal assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>... &lt;= ...;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>if-then-else</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ... then ... elsif ... end if;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>case ... is</td>
</tr>
<tr>
<td>when ...</td>
</tr>
<tr>
<td>when ... =&gt; ...;</td>
</tr>
<tr>
<td>end case;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop ... end loop;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>while loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ... loop ... end loop;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ... in ... loop ... end loop;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>next ...;</td>
</tr>
</tbody>
</table>

The most commonly used sequential statements

1.3.9 A Few More Miscellaneous VHDL Features

This section reserved for your reading pleasure

1.4 Concurrent vs Sequential Statements

All concurrent assignments can be translated into sequential statements. But, not all sequential statements can be translated into concurrent statements.

1.4.1 Concurrent Assignment vs Process

The two code fragments below have identical behaviour:

```
architecture main of tiny is
begin
  b <= a;
end main;
```

```
architecture main of tiny is
begin
  process (a) begin
    b <= a;
  end process;
end main;
```

1.4.2 Conditional Assignment vs If Statements

The two code fragments below have identical behaviour:

**Concurrent Statements**

```
t <= <val1> when <cond>
t <= <val2> when <cond>
else <val2>;
```

**Sequential Statements**

```
if <cond> then
  t <= <val1>;
else
  t <= <val2>;
end if
```
1.4.3 Selected Assignment vs Case Statement

The two code fragments below have identical behaviour:

**Concurrent Statements**

```vhdl
with <expr> select
  t <= <val1> when <choices1>,
       <val2> when <choices2>,
       <val3> when <choices3>;
```

**Sequential Statements**

```vhdl
case <expr> is
  when <choices1> =>
    t <= <val1>;
  when <choices2> =>
    t <= <val2>;
  when <choices3> =>
    t <= <val3>;
end case;
```

1.4.4 Coding Style

Code that's easy to write with **sequential** statements, but difficult with **concurrent**:

```vhdl
case <expr> is
  when <choice1> =>
    if <cond> then
      o <= <expr1>;
    else
      o <= <expr2>;
    end if;
  when <choice2> =>
    ...
end case;
```

1.5 Overview of Processes

Processes are the most difficult VHDL construct to understand. This section gives an overview of processes. **Section 1.6** gives the details of the semantics of processes.

- Within a process, statements are executed *almost* sequentially
- Among processes, execution is done in parallel
- Remember: a process is a concurrent statement!

**Process Semantics**

- VHDL mimics hardware
- Hardware (gates) execute in parallel
- Processes execute in parallel with each other
- All possible orders of executing processes must produce the same simulation results (waveforms)
- If a signal is not assigned a value, then it holds its previous value

**All orders of executing concurrent statements must produce the same waveforms**
1.5.1 Combinational Process vs Clocked Process

Each well-written synthesizable process is either combinational or clocked.

**Combinational process:**
- Executing the process takes part of one clock cycle
- Target signals are outputs of combinational circuitry
- A combinational process must have a sensitivity list
- A combinational process must not have any wait statements
- A combinational process must not have any rising_edges, or falling_edges
- The hardware for a combinational process is just combinational circuitry

**Clocked process:**
- Executing the process takes one (or more) clock cycles
- Target signals are outputs of flops
- Process contains one or more wait or if rising_edge statements
- Hardware contains combinational circuitry and flip flops

**Note:** Clocked processes are sometimes called “sequential processes”, but this can be easily confused with “sequential statements”, so in ECE-327 we’ll refer to synthesizable processes as either “combinational” or “clocked”.
Combinational or Clocked Process? (1)

```vhdl
process (a,b,c)
  variable p1: bit;
begin
  if (b = c) then
    p1 <= a;
  else
    p1 <= a;
  end if;
end process;
```

Combinational or Clocked Process? (2)

```vhdl
process
begin
  wait until rising_edge(clk);
  b <= a;
end process;
```

Combinational or Clocked Process? (3)

```vhdl
process (clk)
begin
  if rising_edge(clk) then
    b <= a;
  end if;
end process;
```

Combinational or Clocked Process? (4)

```vhdl
process (clk)
begin
  a <= clk;
end process;
```
1.5.2 Latch Inference

The semantics of VHDL require that if a signal is assigned a value on some passes through a process and not on other passes, then on a pass through the process when the signal is not assigned a value, it must maintain its value from the previous pass.

Example of latch inference

Question: Write VHDL code for each of the above circuits.
Review: Introduction to VHDL

1. The goal of ece327 is help you think ____________.

2. Hardware runs ____________

   Software runs ____________

3. In VHDL, the interface of a circuit is called a(n) ____________.

   In VHDL, the body of a circuit is called a(n) ____________.

   The body of a circuit contains ____________ statements, which execute

   ____________

   A process contains ____________ statements, which execute

   ____________

4. To simulate hardware:

   At each ____________, every gate in the circuit:

   1 ________________

   2 ________________

   3 ________________
1.6 VHDL Execution: Delta-Cycle Simulation

1.6.1 Simple Simulation

**Hardware runs in parallel:** At each infinitesimally small moment of time, in parallel, each gate:

1. samples its inputs
2. computes the value of its output
3. drives the output

![Diagram of logic gates and timing chart]
1.6.2 Temporal Granularities of Simulation

**register-transfer-level**
- smallest unit of time is a clock cycle
- combinational logic has zero delay
- flip-flops have a delay of one clock cycle

**timing simulation**
- smallest unit of time is a nano, pico, or fempto second
- combinational logic and wires have delay as computed by timing analysis tools
- flip-flops have setup, hold, and clock-to-Q timing parameters

**delta cycles**
- units of time are artifacts of VHDL semantics and simulation software
- simulation cycles, delta cycles, and simulation steps are infinitesimally small amounts of time
- VHDL semantics are defined in terms of these concepts

1.6.3 Zero-Delay Simulation

Register-transfer-level and delta-cycle simulation are both examples of zero-delay simulation.

There are two fundamental rules for zero-delay simulation:
1. Events appear to propagate through combinational circuitry instantaneously.
2. All of the gates appear to operate in parallel

1.6.4 Intuition Behind Delta-Cycle Simulation

1.6.4.1 Introduction to Delta-Cycle Simulation
- To make it appear that events propagate instantaneously through combinational circuitry: VHDL introduces the delta cycle
  - Infinitesimally small artificial unit of time
  - In each delta cycle, in parallel, every gate in the circuit
    1. samples its input signals
    2. computes its result value
    3. drives the result value on its output signal
- To make it appear that gates operate in parallel: VHDL introduces the projected assignment
  - the effect of simulating a gate remains invisible until the beginning of the next delta cycle

1.6.4.2 Intuitive Rules for Delta-Cycle Simulation
1. Simulate a gate if any of its inputs changed.
   If no input changed, then the current value of the output is correct and the output can stay at the same value.
2. Each gate is simulated at most once per delta cycle.
3. When a gate is executed, the projected (i.e., new) value of the output remains invisible until the beginning of the next delta cycle.
4. Increment time when there is no need for another delta cycle.
   No gate had an input change value in the current delta cycle.
1.6.4.3 Example of Delta: Buffers

This section reserved for your reading pleasure

1.6.4.4 Example of Delta and Proj: Buffers

Abbreviated Code

```
proc (a)
  b <= a;
end;
proc (b)
  c <= b;
end;
```

Hardware

```
a  b  c
S: __________________
C: _________________
D: _________________
```

1.6.4.5 Example of Proj Asn: Flip-Flops

This section reserved for your reading pleasure
1.6.4.6 Example of Delta and Proj: Comb Loop

Truly parallel simulation:
- multiple gates/processes execute at the same time
- therefore no need for projected assignment.
Correct Simulation

- Processes execute one at a time.
- Projected assignments become visible at the beginning of the next delta cycle.
Different Execution Orders

The order in which we execute the processes does not affect the behaviour.

Execution order: b, c, d

Execution order: b, d, c
Buggy Simulation

- Processes execute one at a time.
- Projected assignments become visible immediately.

**Execution order: b, c, d**

**Execution order: b, d, c**
Analysis

Which values does \( c \) see?

<table>
<thead>
<tr>
<th>Correct</th>
<th>Buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td></td>
</tr>
<tr>
<td>( d )</td>
<td></td>
</tr>
</tbody>
</table>
Re-do with Waveforms

Execution order: b, c, d

Execution order: b, d, c
Buggy Simulation

Execution order: b, c, d

Execution order: b, d, c
The algorithm presented here is a simplification of the actual algorithm in the VHDL Standard.

This algorithm does not support:

- delayed assignments; for example: \( a \leq b \) after 2 ns;
- resolution, which is where multiple processes write to the same signal (usually a mistake, but useful for tri-state busses)

### Informal Description of Algorithm

- Processes have three modes:
  - **Resumed**: The process has work to do and is waiting its turn to execute.
  - **Executing**: The process is running.
  - **Suspended**: The process is idle and has no work to do.

- A simulation run is initialization followed by a sequence of simulation rounds
  - **Initialization**:
    - Each process starts off **resumed**.
    - Each signal starts off with its **default value**. (‘U’ for std_logic)

- In each **simulation round**:
  - Increment time
  - Resume all processes that are waiting for the current time
  - A simulation round is a sequence of simulation cycles.

- In each **simulation cycle**:
  - Copy projected value of signals to current value.
  - Resume processes based on sensitivity lists and wait conditions.
  - Execute each resumed process.
  - If no projected assignment changed the value of a signal, then increment time and start next simulation round.

### Definitions and Algorithm

#### Notes on Simulation Algorithm

- At a wait statement, the process will suspend even if the condition is true in the current simulation cycle. The process will resume the next time that a signal in the condition changes and the condition is true.
- If we execute multiple assignments to the same signal in the same process in the same simulation cycle, only the last assignment actually takes effect — all but the last assignment are ignored.
- In a simulation round, the first simulation cycle is not a delta cycle.
- The mode of a process is determined implicitly by keeping track of the set of processes that are resumed (the **resume set**) and the process(es) that is(are) executing. All other processes are suspended.
### VHDL Simulation Definitions

**Definition** simulation step: Executing one sequential assignment or process mode change.

**Definition** simulation cycle: The operations that occur in one iteration of the simulation algorithm.

**Definition** delta cycle: A simulation cycle where time did not advance at the beginning of the cycle.

**Definition** simulation round: A sequence of simulation cycles that all have the same simulation time.

---

**More Formal Description of Algorithm**

This section reserved for your reading pleasure

---

#### 1.6.5.4 Example: Delta-Cycle for Flip-Flops

This section reserved for your reading pleasure

---

```vhdl
proc_a : process begin
    a <= '0';
    wait for 1 ns;
    a <= '1';
    wait;
end process;

proc_b : process (a) begin
    b <= not(a);
end process;

proc_c : process (a,b,d) begin
    c <= not(a) or b or d;
end process;

proc_d : process (a,c) begin
    d <= a and c;
end process;
```
1.6.5 VHDL Delta-Cycle Simulation
1.6.5.6 Rules and Observations for Drawing Delta-Cycle Simulations

The VHDL Language Reference Manual gives only a textual description of the VHDL semantics. The conventions for drawing the waveforms are just our own.

- Each column is a simulation step.
- In a simulation step, either exactly one process changes mode or exactly one signal changes value, except in the first two simulation steps of each simulation cycle, when multiple current values may be updated and multiple processes may resume.
- If a projected assignment assigns the same value as the signal’s current projected value, the projected assignment must still be shown, because this assignment will force another simulation cycle in the current simulation round.
- If a signal’s visible value is updated with the same value as it currently has, this assignment is not shown, because it will not trigger any sensitivity lists.
- Assignments to signals may be denoted by either the number/letter of the new value or one of the edge symbols:

<table>
<thead>
<tr>
<th>Old Value</th>
<th>U</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Value</td>
<td>U</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Some observations about delta-cycle simulation waveforms that can be helpful in checking that a simulation is correct:

- In the first simulation step of the first simulation cycle of a simulation round (i.e., the first simulation step of a simulation round), at least one process will resume. This is contrast to the first simulation step of all other simulation cycle, where current values of signals are updated with projected values.
- At the end of a simulation cycle all processes are suspended.
- In the last simulation cycle of a simulation round either no signals change value, or any signal that changes value is not in the sensitivity list of any process.

1.6.6 External Inputs and Flip-Flops

Question: Do the signals b1 and b2 have the same behaviour from 10–20 ns?

```
architecture mathilde of sauvé is
  signal clk, a, b : std_logic;
begin
  process
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;
  process
    wait until rising_edge(clk);
    a1 <= '1';
  end process;
  process
    wait until rising_edge(clk);
    a2 <= '1';
  end process;
  process
    b1 <= a1;
    b2 <= a2;
  end process;
end architecture;
```

Review: Delta-Cycle Simulation

A delta-cycle is a ____________ at the beginning of which ____________.

The two illusions of zero-delay simulation:

1. ____________propagate ______________
2. ____________operate ____________

VHDL achieves the illusions by:

1. ______________
2. ______________
1.7 Register-Transfer-Level Simulation

1.7.1 Overview
- Much simpler than delta cycle
- Columns are real time: clock cycles, nanoseconds, etc.
- Can simulate both synthesizable and unsynthesizable code
- Cannot simulate combinational loops
- Same values as delta-cycle at end of simulation round

```vhdl
process begin
  a <= '0';
  wait for 10 ns;
  a <= '1';
  ...
end process;

process begin
  b <= '0';
  wait for 10 ns;
  b <= a;
  ...
end process;
```

**Question:** In this code, what value should `b` have at 10 ns — does it read the new value of `a` or the old value?

1.7.2 Technique for Register-Transfer Level Simulation

1. Pre-processing
   - (a) Separate processes into timed, clocked, and combinational
   - (b) Decompose each combinational process into separate processes with one target signal per process
   - (c) Sort combinational processes into topological order based on dependencies

2. For each moment of real time:
   - (a) Run timed processes in any order, reading old values of signals.
   - (b) Run clocked processes in any order, reading new values of timed signals and old values of registered signals.
   - (c) Run combinational processes in topological order, reading new values of signals.

1.7.3 Examples of RTL Simulation

1.7.3.1 RTL Simulation Example 1

We revisit an earlier example from delta-cycle simulation, but change the code slightly and do register-transfer-level simulation.

```vhdl
proc1: process (a, b, c) begin
  d <= NOT c;
  c <= a AND b;
end process;

proc2: process (b, d) begin
  e <= b AND d;
end process;
```

```vhdl
proc3: process begin
  a <= '1';
  b <= '0';
  wait for 3 ns;
  b <= '1';
  wait for 99 ns;
end process;
```
Decompose and sort comb procs

```vhdl
proc1d: process (c) begin
  d <= NOT c;
end process;

proc1c: process (a, b) begin
  c <= a AND b;
end process;

proc2: process (b, d) begin
  e <= b AND d;
end process;
proc3: process begin
  a <= '1';
  b <= '0';
  wait for 3 ns;
  b <= '1';
  wait for 99 ns;
end process;
```

Decomposed

```vhdl
proc1d: process (c) begin
  d <= not c;
end process;

proc1c: process (a, b) begin
  c <= a and b;
end process;

proc2: process (b, d) begin
  e <= b and d;
end process;

proc3: process begin
  a <= '1';
  b <= '0';
  wait for 3 ns;
  b <= '1';
  wait for 99 ns;
end process;
```

Sorted

Why is RTL-simulation unable to support combinational loops?

```vhdl
process (a, c) begin
  b <= a xor c;
end process;

process (b) begin
  c <= not b;
end process;
```

Waveforms

```
<table>
<thead>
<tr>
<th></th>
<th>0ns</th>
<th>1ns</th>
<th>2ns</th>
<th>3ns</th>
<th>102ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Decomposing if-then-else Clauses

This example illustrates how to decompose a combinational process that contains assignments to multiple variables and if-then-else clauses.

```
process (a, b, c) begin
  if a = '1' then
    y <= b;
    z <= c;
  else
    y <= not b;
    z <= not c;
  end if;
end process;
```

Original

Decomposed

```
proc1c: process (a, b) begin
  c <= a and b;
end process;

proc1d: process (c) begin
  d <= not c;
end process;

proc2: process (b, d) begin
  e <= b and d;
end process;
```
Review: RTL Simulation

1. Algorithm for RTL simulation:

Preprocessing

(a) Separate processes into two groups: _______ and _______

(b) _______ the _______ processes so that each process _______

(c) Sort the _______ processes into _______ order

Running For each moment in time or clock cycle:

(a) Run the _______ processes in _______ order. Processes read the _______ value of signals.

(b) Run the _______ processes in _______ order. Processes read the _______ value of signals.

2. What are the defining characteristics of zero-delay simulation?

(a) _______ operate _______

(b) _______ propagate _______

3. Comparing delta-cycle and RTL simulation:

<table>
<thead>
<tr>
<th>Illusion #1</th>
<th>Illusion #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delta cycle</td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td></td>
</tr>
</tbody>
</table>
1.11.3 Hardware and Code for Flops

1.11.3.1 Flops with Waits and Ifs

This section reserved for your reading pleasure

1.11.3.2 Flops with Synchronous Reset

process (clk)
begin
if rising_edge(clk) then
  if (reset = '1') then
    q <= '0';
  else
    q <= d;
  end if;
end if;
end process;

Flop with Synchronous Reset: Wait-Style

process
begin
  wait until rising_edge(clk);
  if reset = '1' then
    q <= '0';
  else
    q <= d;
  end if;
end process;

Variation on a Floppy Theme

Question: What is this?

process (clk, reset)
begin
  if reset = '1' then
    q <= '0';
  else
    if rising_edge(clk) then
      q <= d;
    end if;
  end if;
end process;
Flop with Chip-Enable

process (clk)
begin
  if rising_edge(clk) then
    if ce = '1' then
      q <= d;
    end if;
  end if;
end process;

Wait-style flop with chip-enable included in course notes

Q: Flop with a Mux on the Input?

Behavioural Comparison

Question: For the two circuits above, does q have the same behaviour in both circuits?

Mux on input

Mux on output
1.11.3.3 Flop with Chip-Enable and Mux on Input

Hint: Chip Enable

process (clk)
begin
  if rising_edge(clk) then
    if ce = '1' then
      q <= d;
    end if;
  end if;
end process;

1.11.3.4 Flops with Chip-Enable, Muxes, and Reset

This section reserved for your reading pleasure

1.11.4 Example Coding Styles

This section reserved for your reading pleasure

1.12 Synthesizable vs Non-Synthesizable Code

For us to consider a VHDL program synthesizable, all of the conditions below must be satisfied:

- the program must be theoretically implementable in hardware
- the hardware that is produced must be consistent with the structure of the source code
- the source code must be portable across a wide range of synthesis tools, in that the synthesis tools all produce correct hardware

Synthesis is done by matching VHDL code against templates or patterns.

It’s important to use idioms that your synthesis tools recognize.

Think like hardware: when you write VHDL, you should know what hardware you expect to be produced by the synthesizer.

1.12.1 Wait For

Wait for length of time (UNSYNTHESIZABLE)

wait for 10 ns;

Reason: Delays through circuits are dependent upon both the circuit and its operating environment, particularly supply voltage and temperature. For example, imagine trying to build an AND gate that will have exactly a 2ns delay in all environments.

1.12.2 Initial Values

Initial values on signals (UNSYNTHESIZABLE)

signal bad_signal : std_logic := '0';

Reason: At powerup, the values on signals are random (except for some FPGAs).
1.12.3 Assignments before Wait Statement

If a synthesizable clocked process has a `wait` statement, then the process must begin with a `wait` statement.

```
process
  c <= a;
  d <= b;
  wait until rising_edge(clk);
end process;
```

Unsynthesizable

```
process
  wait until rising_edge(clk);
  c <= a;
  d <= b;
end process;
```

Synthesizable

Reason: Cannot synthesize reasonable hardware that has the correct behavior.

In simulation, any assignments before the first wait statement will be executed in the first delta-cycle. In the synthesized circuit, the signals will be outputs of flip-flops and will first be assigned values after the first rising-edge.

1.12.4 “if rising_edge” and “wait” in Same Process

An `if rising_edge` statement and a `wait` statement in the same process (UNSYNTHESIZABLE)

```
process
  begin
    if rising_edge(clk) then
      q0 <= d0;
    end if;
    wait until rising_edge(clk);
    q0 <= d1;
  end process;
```

Reason: The idioms for synthesis tools generally expect just a single type of flop-generating statement in each process.

1.12.5 “if rising_edge” with “else” Clause

The `if` statement has a `rising_edge` condition and an `else` clause (UNSYNTHESIZABLE).

```
process (clk)
  begin
    if rising_edge(clk) then
      q0 <= d0;
    else
      q0 <= d1;
    end if;
  end process;
```

Reason: The idioms for the synthesis tools expect a signal to be either registered or combinational, not both.

1.12.6 While Loop with Dynamic Condition and Combinational Body

A while loop where the condition is dynamic (depends upon a signal value) and the body is combinational is unsynthesizable. The loop below is unsynthesizable:

```
process (a,b,c)
  begin
    while a = '1' loop
      z <= b and c;
    end loop;
  end process;
```

This loop is designed to be very small, but illustrate the problem. The loop itself is non-sensical.
For Loop with Combinational Body

A for-loop with a combinational body is synthesizable, because the loop condition can be evaluated statically (at compile/elaboration time). The loop below is synthesizable:

```
process ( b, c ) begin
  for i in 0 to 3 loop
    z(i) <= b(i) and c(i);
  end loop;
end process;
```

An equivalent while loop would require variables, which are an advanced topic (section 1.9).

While loops with dynamic conditions and clocked bodies are synthesizable, but are an example of an implicit state machine and are an advanced topic.

1.13 Guidelines for Desirable Hardware

Code that is synthesizable, but undesirable (i.e., bad coding practices):
- latches
- combinational loops
- multiple drivers for a signal
- asynchronous resets
- using a data signal as a clock
- using a clock signal as data

To prevent undesirable hardware, some synthesis tools will flag some of these problems as "unsynthesizable".

1.13.1 Latches

Difference between a flip-flop and a latch:
- **flip-flop**  Edge sensitive: output only changes on rising (or falling) edge of clock
- **latch**    Level sensitive: output changes whenever clock is high (or low)

A common implementation of a flip-flop is a pair of latches (Master/Slave flop).

Latches are sometimes called “transparent latches”, because they are transparent (input directly connected to output) when the clock is high.

The clock to a latch is sometimes called the “enable” line.

There is more information in the course notes on timing analysis for storage devices (section 8.3).
Latch: Combinational if-then without else

process (a, b)
begin
    if (a = '1') then
        c <= b;
    end if;
end process;

- For a combinational process, every signal that is assigned to, must be assigned to in every branch of if-then and case statements.

  reason If a signal is not assigned a value in a path through a combinational process, then that signal will be a latch.

  note For a clocked process, if a signal is not assigned a value in a clock cycle, then the flip-flop for that signal will have a chip-enable pin. Chip-enable pins are fine; they are available on flip-flops in essentially every cell library.

Signals Missing from Sensitivity List

process (a)
begin
    c <= a and b;
end process;

- For a combinational process, the sensitivity list should contain all of the signals that are read in the process.

  reason Gives consistent results across different tools. Many synthesis tools will implicitly include all signals that a process reads in its sensitivity list. This differs from the VHDL Standard. A synthesis tool that adheres to the standard will either generate an error or will create hardware with latches or flops clocked by data signals if not all signals that are read from are included in the sensitivity list.

  exception In a clocked process using an if rising_edge, it is acceptable to have only the clock in the sensitivity list.

1.13.2 Combinational Loops

A combinational loop is a cyclic path of dependencies through one or more combinational processes.

process (a, b, c) begin
    if a = '0' then
        d <= b;
    else
        d <= c;
    end if;
end process;

process (d, e) begin
    b <= d and e;
end process;

- If you need a signal to be dependent on itself, you must include a register somewhere in the cyclic path.
- Some FPGA synthesis tools consider a combinational loop to be unsynthesizable. We consider it to be synthesizable and bad-hardware, because the hardware is obvious and is obviously bad.

1.13.3 Multiple Drivers

z <= a and b;
z <= c;

- Each signal should be assigned to in only one process. This is often called the “single assignment rule”.

  reason Multiple processes driving the same signal is the same as having multiple gates driving the same wire. This can cause contention, tri-state values, and other bad things.
Multiple Drivers Example

The example below shows how a “software style” structure that puts the reset code in one process will cause multiple drivers for the signals y and z.

process begin
  wait until rising_edge(clk);
  if reset = '1' then
    y <= '0';
    z <= '0';
  end if;
end process;

process begin
  wait until rising_edge(clk);
  if reset = '0' then
    if a = '1' then
      z <= b and c;
    else
      z <= d;
    end if;
  end if;
end process;

process begin
  wait until rising_edge(clk);
  if reset = '0' then
    if b = '1' then
      y <= c;
    end if;
  end if;
end process;

1.13.4 Asynchronous Reset

In an asynchronous reset, the test for reset occurs outside of the test for the clock edge.

process (reset, clk)
begin
  if (reset = '1') then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;

• All reset signals should be synchronous.
  
  reason If a reset occurs very close to a clock edge, some parts of the circuit might be reset in one clock cycle and some in the subsequent clock cycle. This can lead the circuit to be out of sync as it goes through the reset sequence, potentially causing erroneous internal state and output values.

1.13.5 Using a Data Signal as a Clock

process begin
  wait until rising_edge(clk);
  count <= count + 1;
end process;

process begin
  wait until rising_edge(count(5));
  b <= a;
end process;

• Data signals should be used only as data.
  
  reason All data assignments should be synchronized to a clock. This ensures that the timing analysis tool can determine the maximum clock speed accurately. Using a data signal as a clock clock signals can lead to unpredictable delays between different assignments, which makes it infeasible to do an accurate timing analysis.

1.13.6 Using a Clock Signal as Data

process begin
  wait until rising_edge(clk);
  count <= count + 1;
end process;

b <= a and clk;

• Clock signals should be used only as clocks.
  
  reason Clock signals have two defined values in a clock cycle and transition in the middle of the clock cycle. At the register-transfer level, each signal has exactly one value in a clock cycle and signals transition between values only at the boundary between clock cycles.
1.14 Bad VHDL Coding

This section lists some coding practices to avoid in VHDL unless you have a very good reason.

1.14.1 Tri-State Buffers and Signals

‘Z’ as a Signal Value

process (sel, a0)
  b <= a0 when sel = '0'
  else 'Z';
end process;

process (sel, a1)
  b <= a1 when sel = '1'
  else 'Z';
end process;

- Use multiplexers, not tri-state buffers.
  reason Multiplexers are more robust than tri-state buffers, because tri-state buffers rely on analog effects such as drive-strength and voltages that are between ‘0’ and ‘1’. Multiplexers require more area than tri-state buffers, but for the size of most busses, the advantage in a more robust design is worth the cost in extra area.

Inout and Buffer Port Modes

```vhdl
entity bad is
  port (
    io_bad : inout std_logic;
    buf_bad : buffer std_logic
  );
end entity;
```

- Use in or out, do not use inout or buffer
  reason inout and buffer signals are tri-state.
  note If you have an output signal that you also want to read from, you might be tempted to declare the mode of the signal to be inout. A better solution is to create a new, internal, signal that you both read from and write to. Then, your output signal can just read from the internal signal.

1.14.2 Variables in Processes

```vhdl
process
  variable bad : std_logic;
begin
  wait until rising_edge(clk);
  bad := not a;
  d <= bad and b;
  e <= bad or c;
end process;
```

- In a process, use signals; do not use variables
  reason The intention of the creators of VHDL was for signals to be wires and variables to be just for simulation. Some synthesis tools allow some uses of variables, but when using variables, it is easy to create a design that works in simulation but not in real hardware. (section 1.9)
1.14.3 Bits and Booleans as Signals

signal bad1 : bit;
signal bad2 : boolean;

- Use std_logic signals, do not use bit or Boolean signals.

  reason std_logic is the most commonly used signal type across synthesis tools and simulation tools.

Review: Synthesizable, Good, and Bad VHDL

For each code fragment below, answer whether it is synthesizable. If the code is synthesizable, answer whether it follows good coding practices for synthesizable hardware.

1. process (clk) begin
   if rising_edge(clk) then
      q <= a;
   else
      q <= b;
   end if;
end process;

2. process (clk) begin
   if rising_edge(clk) then
      q1 <= d1;
   end if;
   if rising_edge(clk) then
      q2 <= d2;
   end if;
end process;

3. process (a, b) begin
   if a = '1' then
      q <= b;
   else
      q <= not q;
   end if;
end process;

4. process (a, b) begin
   if a = '1' then
      q <= b;
   else
      q <= not q;
   end if;
end process;

Chapter 2

Additional Features of VHDL
2.1 Literals

2.1.1 Numeric Literals

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Integer</td>
<td>17</td>
<td>1023</td>
</tr>
<tr>
<td>Decimal</td>
<td>Real</td>
<td>17.0</td>
<td>1023.1</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>Integer</td>
<td>16#FF#</td>
<td>16#2F190#</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>Real</td>
<td>16#FF.F#</td>
<td>16#2F1.90#</td>
</tr>
<tr>
<td>Binary</td>
<td>Integer</td>
<td>2#1101#</td>
<td>2#011101#</td>
</tr>
<tr>
<td>Binary</td>
<td>Real</td>
<td>2#1101.111#</td>
<td>2#0111.01#</td>
</tr>
<tr>
<td>Exponent</td>
<td>Integer</td>
<td>17E+3</td>
<td>2#111#E3</td>
</tr>
<tr>
<td>Exponent</td>
<td>Real</td>
<td>17.1E+3</td>
<td>2#11.1#E3</td>
</tr>
<tr>
<td>Underscore</td>
<td>Integer</td>
<td>123_45_67</td>
<td>16#FF_3A#</td>
</tr>
</tbody>
</table>

2.1.2 Bit-String Literals

| Binary   | B"1101010" | B"1101_1010" |
| Octal    | O"3470100" | O"45_23"     |
| Hexadecimal | X"FF2300" | X"Ff3dbF_23" |

Note: Array literals are called “aggregates” and are described in section 2.2.2.

2.2 Arrays and Vectors

2.2.1 Declarations

VHDL arrays have:
- direction (to or downto)
- upper bound
- lower bound

```vhdl
signal a : std_logic_vector( 3 downto 0 );
signal b : std_logic_vector( 0 to 3 );
signal c : std_logic_vector( 1 to 4 );
```

Constant Arrays

To define a constant array:

```vhdl
constant a : array( 0 to 3 ) of integer := ( 10, 17, -31, 23 );
constant b : array( 0 to 3 ) of integer := ( 0 => 10, 1 => 17, 2 => -31, 3 => 23 );
constant c : array( 0 to 3 ) of integer := ( 0 => 10, 1 => 17, others => 23 );
```
2.2.2 Indexing, Slicing, Concatenation, Aggregates

Operations

Indexing an array to reference a single element
\[ a(0) \]

A slice or “discrete subrange” of an array
\[ a(3 \text{ downto } 2) \]

Concatenating an element onto an array, or concatenation two arrays
\[ '1' \& a \]
\[ b \& a \]

Array literals or “aggregates”
\[ ('0', '0', '1') \]
\[ (a(0), b(2), a(3)) \]

Aggregate with positional indices
\[ (0=>'0', 2=>'X', 1=>'U') \]

Aggregate with "others" keyword
\[ (0=>'0', 3=>'1', others=>'X') \]

Assignments

1. The ranges on both sides of the assignment must be the same.
2. The direction (\textit{downto} or \textit{to}) of each slice must match the direction of the signal declaration.
3. The direction of the target and expression may be different.

Legal code
\[ b(3 \text{ downto } 0) \leq a(15 \text{ downto } 12); \]
\[ bx(0 \text{ to } 3) \leq a(15 \text{ downto } 12); \]
\[ (b(3), b(4)) \leq a(13 \text{ downto } 12); \]
\[ (bx(4), b(4)) \leq a(13 \text{ downto } 12); \]

Illegal code
\[ bx(0 \text{ to } 3) \leq a(12 \text{ to } 15); \]
\[ -- \text{ slice dirs must be same as decl, fails for } a \]
\[ c(3 \text{ downto } 0) \leq (a \& b)(3 \text{ downto } 0); \]
\[ -- \text{ may not index an expression} \]
\[ b(3) \& b(2) \leq a(12 \text{ to } 13); \]
\[ -- \& \text{ may not be used on lhs} \]

2.3 Arithmetic

VHDL includes all of the common arithmetic operators and relations.

Use the VHDL arithmetic operators and let the synthesis tool choose the best implementation for you.

2.3.1 Arithmetic Packages

To do arithmetic with signals, use the \texttt{numeric_std} package.

\texttt{numeric_std} supersedes earlier arithmetic packages, such as \texttt{std_logic_arith}.

Use only one arithmetic package, otherwise the different definitions will clash and you can get strange error messages.

We will describe arithmetic with the \texttt{numeric_std} package.
2.3.2 Arithmetic Types

Arithmetic may be done on three types of expressions:

- **integers** Numeric values, such as 17
- **unsigned** Unsigned vectors, such as signals defined as type
  `unsigned(7 downto 0)`.
- **signed** Signed vectors, such as signals defined as type
  `signed(7 downto 0)`.

The types `signed` and `unsigned` are `std_logic` vectors on which you can do signed or unsigned arithmetic and all of the operations that are supported by `std_logic_vectors`.

2.3.3 Overloading of Arithmetic

The arithmetic operators `+`, `−`, and `∗` are overloaded on `signed` vectors, `unsigned` vectors, and integers.

<table>
<thead>
<tr>
<th>Target</th>
<th>Src1/2</th>
<th>Src2/1</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>u3 &lt;= u1 + u2; OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>unsigned</td>
<td>u3 &lt;= u1 + 17; OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>signed</td>
<td>s3 &lt;= s1 + s2; OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>signed</td>
<td>s3 &lt;= s1 + -17; OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>u3 &lt;= u1 + s2; Fail</td>
<td>Fail</td>
<td></td>
</tr>
</tbody>
</table>

2.3.4 Widths for Addition and Subtraction

- Sources may have different widths
- The target must be the same width as the widest source

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>w1, w2, w3 : unsigned(7 downto 0)</code> – wide</td>
</tr>
<tr>
<td><code>n1, n2, n3 : unsigned(3 downto 0)</code> – narrow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
<th>Src1/2</th>
<th>Src2/1</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>narrow</td>
<td>narrow</td>
<td>narrow</td>
<td>n3 &lt;= n1 + n2; OK</td>
</tr>
<tr>
<td>narrow</td>
<td>int</td>
<td>narrow</td>
<td>n3 &lt;= n1 + 17; OK</td>
</tr>
<tr>
<td>narrow</td>
<td>wide</td>
<td>—</td>
<td>n3 &lt;= w1 + n2; Fail</td>
</tr>
</tbody>
</table>

These failures are caught at *elaboration*, which happens after typechecking.

2.3.4 Widths for Multiplication

- The sources may be different widths
- The width of the result must be the sum of the widths of the sources

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v4a, v4b, v4c : unsigned(3 downto 0)</code></td>
</tr>
<tr>
<td><code>v8 : unsigned(7 downto 0)</code></td>
</tr>
<tr>
<td><code>v12 : unsigned(11 downto 0)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
<th>Src1/2</th>
<th>Src2/1</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bits</td>
<td>4-bits</td>
<td>4-bits</td>
<td>v8 &lt;= v4a * v4b; OK</td>
</tr>
<tr>
<td>12-bits</td>
<td>4-bits</td>
<td>8-bits</td>
<td>v12 &lt;= v4a * v8; OK</td>
</tr>
<tr>
<td>4-bits</td>
<td>4-bits</td>
<td>4-bits</td>
<td>v4c &lt;= v4a * v; Fail</td>
</tr>
</tbody>
</table>
### 2.3.5 Overloading of Comparisons

- Comparisons are overloaded on arrays and integers.
- If both operands are arrays, both must be of the same type.

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1, u2 : unsigned( 7 downto 0);</td>
</tr>
<tr>
<td>s1, s2 : signed( 7 downto 0);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1/2</th>
<th>Src2/1</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned unsigned</td>
<td>u1 &gt;= u2</td>
<td>OK</td>
</tr>
<tr>
<td>unsigned integer</td>
<td>u1 &gt;= 17</td>
<td>OK</td>
</tr>
<tr>
<td>signed signed</td>
<td>s1 &gt;= s2</td>
<td>OK</td>
</tr>
<tr>
<td>signed integer</td>
<td>s1 &gt;= 17</td>
<td>OK</td>
</tr>
<tr>
<td>unsigned signed</td>
<td>u1 &gt;= s1</td>
<td>Fail</td>
</tr>
</tbody>
</table>

### 2.3.6 Widths for Comparisons

- Sources may have different widths

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1, w2 : unsigned(7 downto 0) – wide</td>
</tr>
<tr>
<td>n1, n2 : unsigned(3 downto 0) – narrow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1/2</th>
<th>Src2/1</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>wide</td>
<td>w1 &gt;= n1</td>
<td>OK</td>
</tr>
<tr>
<td>narrow</td>
<td>n1 &gt;= w2</td>
<td>OK</td>
</tr>
</tbody>
</table>

### 2.3.7 Type Conversion

If you convert between two types of the same width, then no additional hardware will be generated.

**Use: typecast a signal**

```vhdl
unsigned ( val : std_logic_vector ) return unsigned;
signed ( val : std_logic_vector ) return signed;
```

**Use: assign an integer literal to a signal**

```vhdl
to_unsigned( val : integer; width : natural) return unsigned;
to_signed ( val : integer; width : natural) return signed;
```

**Use: use a signal as an index into an array**

```vhdl
to_integer ( val : signed ) return integer;
to_integer ( val : unsigned ) return integer;
```

### Examples of Conversions

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1, u2, u3 : unsigned( 7 downto 0);</td>
</tr>
<tr>
<td>sn1, sn2, sn3 : signed( 7 downto 0);</td>
</tr>
<tr>
<td>sw1, sw2, sw3 : signed( 8 downto 0);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>u3 &lt;= to_unsigned( 17, 8 );</td>
</tr>
<tr>
<td>sn3 &lt;= to_signed( 17, 8 );</td>
</tr>
<tr>
<td>sw3 &lt;= signed( &quot;0&quot; &amp; u1 );</td>
</tr>
<tr>
<td>sn3 &lt;= signed( u1 );</td>
</tr>
<tr>
<td>sw3 &lt;= signed( &quot;0&quot; &amp; u1 ) – signed( &quot;0&quot; &amp; u2);</td>
</tr>
<tr>
<td>sw3 &lt;= signed( &quot;0&quot; &amp; (u1 + u2));</td>
</tr>
<tr>
<td>sw3 &lt;= signed( &quot;0&quot; &amp; (u1 - u2));</td>
</tr>
</tbody>
</table>

The **Bad** examples above will typecheck and elaborate without any errors, but they potentially will produce incorrect results.
Resizing and Sign Extension

The function resize resizes vectors, performing sign extension if necessary, based upon the type of the argument. It is overloaded for different types of arguments.

\[
\begin{align*}
\text{resize( } v : \text{ std\_logic\_vector} ; \text{ width : natural } ) & \text{ return std\_logic\_vector;} \\
\text{resize( } u : \text{ unsigned } ; \text{ width : natural } ) & \text{ return unsigned;} \\
\text{resize( } s : \text{ signed } ; \text{ width : natural } ) & \text{ return signed;} 
\end{align*}
\]

Declarations

\[
\begin{align*}
\text{un1, un2 : unsigned( 4 downto 0);} \\
\text{uw1, uw2 : unsigned( 7 downto 0);} \\
\text{sn1, sn2 : signed( 4 downto 0);} \\
\text{sw1, sw2 : signed( 7 downto 0);} 
\end{align*}
\]

Examples

\[
\begin{align*}
\text{uw1 <= resize( un1, 8 ); OK} \\
\text{un1 <= resize( uw1, 4 ); OK} \\
\text{sw1 <= resize( sn1, 8 ); OK} \\
\text{sn1 <= resize( sw1, 4 ); OK} \\
\text{sw1 <= resize( un1, 8 ); Fail} \\
\text{uw1 <= resize( sn1, 8 ); Fail} 
\end{align*}
\]

Type Conversion and Array Indices

To use a signal as an index into an array, you must convert the signal into an integer using the function to\_integer.

Declarations

\[
\begin{align*}
\text{signal u : unsigned( 3 downto 0);} \\
\text{signal v : std\_logic\_vector( 3 downto 0);} \\
\text{signal a : std\_logic\_vector(15 downto 0);} 
\end{align*}
\]

Examples

\[
\begin{align*}
\text{a( to\_integer( u ) ) Ok} \\
\text{a( to\_integer( unsigned( v ) ) ) Ok} \\
\text{v( u ) Fail} \\
\text{a( unsigned( v ) ) Fail} 
\end{align*}
\]

2.3.8 Shift and Rotate Operations

Shift and rotate operations are described with three character acronyms:

\[
\{ \text{shift} \} \{ \text{left/right} \} \{ \text{arithmetic/logical} \}
\]

The shift right arithmetic (sra) operation preserves the sign of the operand, by copying the most significant bit into lower bit positions.

The shift left arithmetic (sla) does the analogous operation, except that the least significant bit is copied.

\[
a \text{sra 2 -- arithmetic shift of a by 2 bits}
\]

2.3.9 Arithmetic Optimizations

Multiply by a constant power of two wired shift logical left
Multiply by a power of two shift logical left
Divide by a constant power of two wired shift logical right
Divide by a power of two shift logical right

Question: How would you implement: \( z \Leftarrow a + 3 \)?
2.4 Types

2.4.1 Enumerated Types

VHDL supports enumerated types:

```vhdl
type color is (red, green, blue);
```

2.4.2 Defining New Array Types

When defining a new array type, the range may be left unconstrained:

```vhdl
type color is (red, green, blue);
type color_vector is array (natural range <> ) of color;
```

We may then use the unconstrained array type as the basis for defining a constrained array `subtype`:

```vhdl
subtype few_colors is color_vector(0 to 3);
subtype many_colors is color_vector(0 to 1023);
```

Note the use of `subtype` above. It is illegal to use `type` to define a constrained array in terms of an unconstrained array.

We can use `type` to define a constrained array directly:

```vhdl
type few_colors is array (0 to 3) of color;
```

Chapter 3

Overview of FPGAs

3.1 Generic FPGA Hardware

- This section: generic FPGA with 4 inputs per lookup table.
- Many real FPGAs have more (e.g., 6) inputs per lookup table.
- Principles described here are applicable in general, even as details differ.
### Generic FPGA Cell

FPGA “Cell” = “Logic Element” (LE) in Altera
= “Configurable Logic Block” (CLB) in Xilinx

“LUT” = “lookup table”
= PLA (programmable logic array)

**Separate Comb and Flop**

- **Connect Comb and Flop**

- **Flopped and Unflopped Outputs**
3.1.2 Lookup Table

A 4:1 lookup table is usually implemented as a memory array with 16 1-bit elements.

\[
\begin{align*}
z &= (a \text{ AND } b) \quad \text{OR} \\
    &\quad (b \text{ AND NOT } c) \quad \text{OR} \\
    &\quad (c \text{ AND NOT } d)
\end{align*}
\]

\[
z = \text{NOT } a
\]

<table>
<thead>
<tr>
<th>4-bit address</th>
<th>1-bit data</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3.1.3 Interconnect for Generic FPGA

Local Connections

Note: In these pictures, the space between tightly grouped wires sometimes disappears, making a group of wires appear to be a single large wire.

General-Purpose Wires and Carry Chains

General purpose interconnect configurable, slow

Carry chains and cascade chains vertically adjacent cells, fast
### 3.1.4 Blocks of Cells for Generic FPGA

![Diagram of blocks of cells for FPGA]

- Column of cells in blocks
- Two rows of blocks
- Path to connect cells in different rows

**Connecting Through Cells**

Cells that are not used for computation can be used as "wires" to shorten length of path between cells.

### 3.1.5 Special Circuitry in FPGAs

**Memory**

Since the mid 1990s, almost all FPGAs have had special circuits for RAM and ROM. These special circuits are possible because many FPGAs are fabricated on the same processes as SRAM chips. So, the FPGAs simply contain small chunks of SRAM.

**Microprocessors**

In 2001, some high-end FPGAs had one or more hardwired microprocessors on the same chip as programmable hardware. In 2005, the Xilinx-II Pro had 4 Power PCs and enough programmable hardware to implement the first-generation Intel Pentium microprocessor.

**Arithmetic Circuitry**

In 2001, FPGAs began to have hardwired circuits for multipliers and adders. Using these resources can improve significantly both the area and performance of a design.

**Input / Output**

Some FPGAs include special circuits to increase the bandwidth of communication with the outside world.

### 3.2 Area Estimation for FPGAs

This section describes three methods to estimate the number of FPGA cells required to implement a circuit:

- **section 3.2.1** Rough estimate based simply upon the number of flip-flops and primary inputs that are in the fanin of each flip-flop or output.
- **section 3.2.2** A more accurate, and more complex, technique that uses a greedy algorithm to allocate as many gates as possible into the lookup table of each FPGA cell.
- **section 3.2.3** A technique to estimate the area for arithmetic circuits with registers.

Each cell:

- LUT for any combinational function with up to four inputs and one output
- Carry-in and carry-out signals used only for arithmetic carries
- Flip-flop can be driven by LUT or separate input
3.2.1 Area for Circuit with one Target

This section gives a technique to estimate the number of FPGA cells required for a purely combinational circuit with one output.

**Question:** What is the maximum number of inputs for a function that can be implemented with one LUT?

**Question:** Number of inputs for two LUTs?

**Question:** Three LUTs?

**Question:** Four LUTs?

**Single Target vs Multiple Targets**

For a single target signal, this technique gives a lower bound on the number of LUTs needed.

For multiple target signals, this technique might be an overestimate, because a single LUT can be used in the logic for multiple target cells.

---

4:1 Mux in Two FPGA Cells

A 4:1 mux has 6 inputs, so it should fit into two FPGA cells.

But, with some clever tricks, a 4:1 mux can be implemented in two FPGA cells:

---

3.2.2 Algorithm to Allocate Gates to Cells

This section presents an algorithm to allocate gates to FPGA cells for circuits with:
- multiple outputs
- combinational gates
- flip-flops

The algorithm mimics what a synthesis tool does in transforming a netlist of generic gates into an FPGA:

**Technology map** Map groups of generic combinational gates into LUTs

**Placement** Assign each LUT and flip-flop to an FPGA cell

In addition to above, synthesis tools do the step of routing: connecting the signals between FPGA cells.

Because we are working with general-purpose combinational gates, we cannot use the carry-in and carry-out signals with the LUTs.
Overview of Algorithm

For each flip-flop and output: traverse backward through the fanin gathering as much combinational circuitry as possible into the FPGA cell.

Stopping conditions:
• flip-flop
• more than four inputs — However, have more than four signals as input, then further back in the fanin, the circuit will collapse back to four or fewer signals.

Question: Map the circuit below onto generic FPGA cells.
Do not perform any algebraic optimizations. Use NC (no connect) for any unused pins on the cells.

Number of FPGA Cells (1)

Number of FPGA Cells (2)

Number of FPGA Cells (3)
In this question, the signal becomes a new output.

Question: Map the circuit below onto generic FPGA cells.
3.2.3 Area for Arithmetic Circuits

For arithmetic circuits, we take into account inputs, outputs, carry-in, and carry-out signals.

1 lookup table can implement one 1-bit full-adder

\[ a \rightarrow d0 \rightarrow d1 \rightarrow d2 \rightarrow d3 \rightarrow sum \rightarrow co \]

\[ a0, b0, c0 \rightarrow d0 \rightarrow d1 \rightarrow d2 \rightarrow d3 \rightarrow sum0, sum1 \rightarrow sum2, sum3 \rightarrow co \]

Two-Bit Adder

Question: How many lookup tables are needed for a two-bit adder?

Adder with a Multiplexer

Question: How many lookup tables for an adder with a 2:1 mux on one input?

Arithmetic VHDL Code

Question: How many cells are needed for each of the code fragments below?

All signals are 8 bits.

\[ z \leftarrow a + b; \]
\[ z \leftarrow a + b + c; \]
\[
\begin{align*}
& \text{process begin} \\
& \quad \text{wait until rising_edge(clk);} \\
& \quad z \leftarrow a + b + c; \\
& \text{end process;}
\end{align*}
\]
Arithmetic VHDL Code (Cont’d)

```vhdl
process begin
  wait until rising_edge(clk);
  a <= i_a;
  b <= i_b;
  c <= i_c;
  z <= a + b + c;
end process;

a <= i_a;
b <= i_b;
c <= i_c;
process begin
  wait until rising_edge(clk);
  z <= a + b + c;
end process;

m <= a when sel='0'
  else b;
process begin
  wait until rising_edge(clk);
  z <= m + c;
end process;
```

Other Arithmetic Operations

<table>
<thead>
<tr>
<th>Code</th>
<th>Number of LUTs per bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>z &lt;= a + 1;</td>
<td></td>
</tr>
<tr>
<td>z &lt;= a = b;</td>
<td></td>
</tr>
<tr>
<td>z &lt;= a = 0;</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4

State Machines

4.1 Notations

We will use a variety of notations to model our hardware:

- **Pseudocode** For algorithms. Used early in the design process for sequential behaviour and high-level optimizations.
- **Dataflow diagrams** Models the structure and behaviour of datapath-intensive circuits.
- **State machines** A variation on the conventional bubble-and-arrow style state machines.
- **VHDL code** For the real implementation.

4.2 Finite State Machines in VHDL

4.2.1 HDL Coding Styles for State Machines

- **Explicit** VHDL code contains a state signal. At most one `wait` statement per process.
  - **Explicit-Current** The state signal represents the current state of the machine and the signal is assigned its next value in a clocked process.
  - **Explicit-Current+Next** there is a signal for the current state and another signal for the next state. The next-state signal is assigned its value in a combinational process or concurrent statement and is dependent upon the current state and the inputs. The current-state signal is assigned its value in a clocked process and is just a flopped copy of the next-state signal. (“three-process” style)
- **Implicit** There is no explicit state signal. At least one process has multiple `wait` statements. Each `wait` statement corresponds to a single state (Advanced topic not covered in this course).
4.2.2 State Encodings

Explicit state machines require a state signal. Before we can define a state signal, we must define values for the names of the states. For example, we might define $S0$ to be "000" and $S1$ to be "001". The value for the name of state is called the "encoding" of the state. In hardware, each value is a bit-vector. There are a variety common encodings for states: binary, one-hot, Gray, and thermometer.

We can either define the encoding ourselves, or let the synthesis tool choose the encoding for us. If we define the encoding, then the type for the states is `std_logic_vector`. To let the synthesis choose the encoding, we create an enumerated type to the states, where each state is an element of the type. The synthesis tool then chooses a specific binary value for each state. Usually, the synthesis tool has heuristics to choose either a binary or one-hot encoding.

4.2.3 Traditional State-Machine Notation

This section reserved for your reading pleasure

4.2.4 Our State-Machine Notation

A simple extension to Mealy machines, allow both:
- combinational assignments $z = 0$
- registered assignments $z' = 0$

4.2.5 Bounce Example

This section reserved for your reading pleasure

Explicit-Current Coding Style
**Combinational Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    case state is
      when S0 =>
        if a = '1' then
          state <= S1;
        else
          state <= S2;
        end if;
      when others =>
        state <= S0;
    end case;
  end if;
end process;
```

**Registered Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    case state is
      when S0 =>
        if a = '1' then
          state <= S1;
          z <= '1';
        else
          state <= S2;
          z <= '0';
        end if;
      when others =>
        state <= S0;
        z <= '1';
    end case;
  end if;
end process;
```

**Additional Coding Options**

**Combinational Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    case state is
      when S0 =>
        if a = '1' then
          state <= S1;
          z <= '1';
        else
          state <= S2;
          z <= '0';
        end if;
      when others =>
        state <= S0;
        z <= '1';
    end case;
  end if;
end process;
```

**Registered Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    if state = S0 and a = '1' then
      z <= '1';
    else
      z <= '0';
    end if;
  end if;
end process;
```

**Explicit-Current+Next**

**Combinational Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    next_st <= S1 when st = S0 and a = '1' else S2 when st = S0 else S0;
    state <= S0;
    z <= '1' when (st = S0 and a = '1') or (st = S2) else '0';
  end if;
end process;
```

**Registered Assignments**

```
process (clk) begin
  if rising_edge(clk) then
    if (st = S0 and a = '1') or (st = S2) then
      z <= '1';
    else
      z <= '0';
    end if;
  end if;
end process;
```

**Implicit**

**Combinational Assignments**

```
process (clk) begin
  wait until rising_edge(clk); -- S0
  if a = '1' then
    z <= '1';
  else
    z <= '0';
  end if;
end process;
```

**Registered Assignments**

```
process begin
  wait until rising_edge(clk); -- S0
  if a = '1' then
    z <= '1';
  end if;
end process;
```

**Note:** Implicit state machines do not support combinational assignments, because an implicit state machine is a clocked process and in a clocked process, all assignments are registered.

**Note:** Implicit state machines are an advanced topic and are not covered in ECE-327.
4.2.6 Registered Assignments

Combinational assignments Appear to happen instantaneously.

Registered assignments Clock-cycle boundary between when inputs are sampled and when target signal is driven.

VHDL and FSMs use different techniques to achieve the same behaviour.

Use a registered assignment based on the state to illustrate.

```
process begin
  wait until re(clk);
  if state = S0 then
    z <= 1;
  else
    z <= 0;
  end if;
end process;
```

FSM Assignment is executed before the clock edge.

Delay driving the output until after clock edge.

VHDL Assignment is executed after the clock edge.

Sample the old (visible) value of registered inputs from before the clock edge.


4.2.7 More Notation

4.2.7.1 Extension: Transient States

With transient-state, write \( y = 1 \) just once.
Transient States with Registered Assignments

- Syntactically, registered assignments may appear before combinational assignments.
- Semantically, the effect of the registered assignments occurs after the combinational assignments.

Assignments within States

As another example to illustrate moving assignments between edges and states, the three machines below have the same behaviour:

Assignments within States (Cont’d)

2. If all incoming edges have the same registered assignment, then the assignment may be transformed into a combinational assignment and moved into the state.

The three state machines below all have the same behaviour.

As another example to illustrate moving assignments between edges and states, the three machines below have the same behaviour:
4.2.7.3 Conditional Expressions

The FSMs below have the same behaviour:

\[
\text{S0} \quad z = \overline{a}z = c \quad \text{S1}
\]

The FSMs below have the same behaviour:

\[
\text{S0} \quad \overline{a}z = b \quad \text{S1}
\]

4.2.7.4 Default Values

**Combinational**

Intuition: a signal is defined its default value in any state-to-state transition where it is not explicitly assigned a value.

**More Examples**

With default values

Equivalent FSM without default values
The semantics define that if a registered variable is not assigned a value in a clock cycle, then it holds its previous value.

<table>
<thead>
<tr>
<th>Default expression</th>
<th>Behaviour when not assigned a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>z holds its previous value.</td>
</tr>
<tr>
<td>( z' = a )</td>
<td>z is assigned a.</td>
</tr>
<tr>
<td>( z' = ' - ' )</td>
<td>z is unconstrained.</td>
</tr>
</tbody>
</table>

**Default Value: Registered Assignment**

<table>
<thead>
<tr>
<th>( z' = 99 )</th>
<th>Equivalent FSM without default values</th>
</tr>
</thead>
</table>

**Questions to Answer and Ponder**

**Question:** Why do combinational variables not need a “don’t care” default statement?

**Question:** Why do register variables need a “don’t care” default statement?
4.2.8 Semantic and Syntax Rules

**Inputs, Combinational, Registered**

There are three categories of variables in FSMs. Each category has its own rules for how and when the variables are updated.

- **Inputs** Values are updated every clock cycle.
- **Combinational** If a variable is not assigned a value in a clock cycle, then its value is unconstrained.
- **Registered** If a variable is not assigned a value in a clock cycle, then it holds its previous value.

If there is any ambiguity about whether a signal is an input, then it should be declared as an input.

**Multiple Assignments to Same Signal**

For a sequence of transitions within the same clock cycle, only the last assignment to each signal is visible.

**Summary of Semantic Rules**

1. Signals take on the value of the last assignment that is executed in a clock cycle.
2. Combinational assignments become visible immediately.
3. Registered assignments become visible in the next clock cycle.
4. If a combinational signal is not assigned to in a given clock cycle, then the value of that signal is unconstrained (in other words, arbitrary, non-deterministic, or don't-care).

**Syntax Rules**

Our state machines are designed to match closely with VHDL code and hardware.

The state machine notation is equivalent to synthesizable hardware that satisfies our rules for good coding practices, with the addition that we also support non-determinism. Non-determinism is not synthesizable, but is often useful in specifications for state machines.

1. For a given signal, it must be that either all assignments are combinational or all assignments are registered.
   - It is illegal to have both combinational and registered assignments to the same signal. The reason is that this will lead to unsynthesizable code, because a signal cannot be both combinational and registered.
2. Within a clock cycle, a combinational signal must not be written to after it has been read.
   - Violating this rule will lead to combinational loops.
3. Completeness of transitions: The conditions on the outgoing edges from a state must cover all possibilities. That is, from a given state, it must always be possible to make a transition. This includes a self-looping transition back to the state itself.

Additional guidelines:

1. Within a clock cycle, a combinational signal should be assigned to \textit{before} it is read.

Violating this guideline will lead to non-deterministic behaviour, because the value of a combinational signal is unconstrained in a clock cycle until it has been written to.

### Deterministic vs Non-Deterministic

**Deterministic**  Exactly one outgoing transition is enabled (condition is true)

**Non-deterministic**  Multiple outgoing transitions are enabled; machine randomly chooses which transition to take

- Our state machines may be non-deterministic.
- Non-determinism happens when multiple outgoing transitions are enabled at the same time.
- Non-determinism is sometimes useful in specifications and high-level models.
- Real hardware is deterministic
  (unless you are building a quantum computer)
- For real hardware, your transitions must be \textit{mutually exclusive}.

### Reset

All circuits should have a reset signal that puts the circuit back into a good initial state. However, not all flip flops within the circuit need to be reset. In a circuit that has a datapath and a state machine, the state machine will probably need to be reset, but datapath may not need to be reset.

**This section reserved for your reading pleasure**

```vhdl
process (clk) begin
  if rising_edge(clk) then
    case state is
    when S0 =>
      if a = '1' then
        z <= '1';
        state <= S1;
      else
        z <= '0';
        state <= S2;
      end if;
    when S1 =>
      z <= '0';
      state <= S0;
    when others =>
      z <= '1';
      state <= S0;
    end case;
  end if;
end process;
```

```vhdl
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      state <= S0;
    else
      case state is
        when S0 =>
          if a = '1' then
            z <= '1';
            state <= S1;
          else
            z <= '0';
            state <= S2;
          end if;
        when S1 =>
          z <= '0';
          state <= S0;
        when others =>
          z <= '1';
          state <= S0;
      end case;
    end if;
  end if;
end process;
```
Do the state-machine fragments below have the same behaviour?

**Without Reset**

```vhdl
process (clk) begin
  if rising_edge(clk) then
    st <= next_st;
  end if;
end process;
next_st <= S1 when st = S0 and a = '1'
  else S2 when st = S0
  else S0;
z <= '1' when (st = S0 and a = '1')
  or (st = S2)
  else '0';
```

**With Reset**

```vhdl
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      st <= S0;
    else
      st <= next_st;
    end if;
  end if;
end process;
next_st <= S1 when st = S0 and a = '1'
  else S2 when st = S0
  else S0;
z <= '1' when (st = S0 and a = '1')
  or (st = S2)
  else '0';
```

**Review: Introduction to State Machines**

Do the state-machine fragments below have the same behaviour?

**4.3 LeBlanc FSM Design Example**

**4.3.1 State Machine and VHDL**

```vhdl
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= S0;
  else
    case state is
      when S0 =>
        if a = '0' then
          st <= S1;
        else
          st <= S2;
        end if;
      when S3 =>
        st <= S0;
    end case;
  end if;
end process;
```

```vhdl
process begin
  wait until rising_edge(clk);
  if a = '0' then
    z <= b - c;
  else
    z <= b + c;
  end if;
end process;
```

```vhdl
type state_ty is (S0, S1, S2, S3);
signal state : state_ty;
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= S0;
  else
    case state is
      when S0 =>
        if a = '0' then
          st <= S1;
        else
          st <= S2;
        end if;
      when S3 =>
        st <= S0;
    end case;
  end if;
end process;
```

```vhdl
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= S0;
  else
    case state is
      when S0 =>
        if a = '0' then
          st <= S1;
        else
          st <= S2;
        end if;
      when S3 =>
        st <= S0;
    end case;
  end if;
end process;
```

```vhdl
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= S0;
  else
    case state is
      when S0 =>
        if a = '0' then
          st <= S1;
        else
          st <= S2;
        end if;
      when S3 =>
        st <= S0;
    end case;
  end if;
end process;
```
4.3.2 State Encodings

With 7 states

<table>
<thead>
<tr>
<th>Binary</th>
<th>One-Hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
</tbody>
</table>

Le Blanc in Binary

```vhdl
---type state_ty is
  signal state : state_ty;
  S0 :
  S1 :
  S2 :
  S3 :

  process begin
    wait until rising_edge(clk);
    if reset = '1' then
      state <= __________
    else
      process begin
        wait until rising_edge(clk);
        if
          z <= b - c;
        else
          z <= b + c;
        end if;
      end process;
    end if;
  end process;
```
State encodings affect the amount of circuitry needed to:

- test conditions that drive the control signals for the datapath.
- choose the next state

Define a custom encoding to simplify the circuitry needed to recognize the condition that the system is in either S1 or S2.

One-Hot LeBlanc

```plaintext
signal state : S0 : S1 : S2 : S3 :
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= __________
  else
    signal state :
      S0 :
      S1 :
      S2 :
      S3 :
  process begin
    wait until rising_edge(clk);
    if z <= b - c;
      z <= b + c;
    end if;
  end process;
end process;
```
4.4 Parcels

- "Parcel" = basic unit of data in a system
- Examples

<table>
<thead>
<tr>
<th>System</th>
<th>Parcel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>Instruction</td>
</tr>
<tr>
<td>Car factory</td>
<td>Car</td>
</tr>
</tbody>
</table>

- A parcel flows through a system
- A parcel may be composed of multiple components

<table>
<thead>
<tr>
<th>Parcel</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Opcode, operands, result</td>
</tr>
<tr>
<td>Car</td>
<td>Doors, windows, engine, etc.</td>
</tr>
</tbody>
</table>

4.4.1 Bubbles and Throughput

- Between each pair of parcels is a sequence of zero or more bubbles

  bubble : invalid or garbage data that must be ignored
- Each system has a requirement for minimum number of bubbles between parcels
- Throughput: number of parcels per clock cycle

  throughput = 1 parcel / 3 clock cycles
  = 1/3 parcels per clock cycle

  throughput = 3 parcels / 12 clock cycles
  = 1/4 parcels per clock cycle

Maximum and Actual Throughput

- Maximum Throughput: The maximum rate of parcels per cycle (minimum number of bubbles) at which the system will work correctly.
  usually: max throughput = 1/(minimum number of bubbles + 1)

- Actual Throughput: The actual rate at which the environment sends parcels to the system.
  Actual throughput must be less-than-or-equal-to maximum throughput.
  Actual number of bubbles must be greater-than-or-equal-to minimum number of bubbles.
Max Tput: Pipelining and Superscalar

Question: Label each of the arrows and dots below with one of: Unpipelined, Pipelined, Fully-pipelined, or Superscalar

As an advanced topic, some systems with both combinational inputs and outputs use an area optimization that reduces the maximum throughput of an unpipelined system to be 1/(latency+1).

FSMs, Latency, and Tput

Question: What are the latency and maximum throughput of the FSM below?

Answer:

Actual Throughput: Constant and Variable

Two categories of actual throughput:

Constant Throughput  Always the same number of bubbles between parcels. Often actual number of bubble is the minimum number of bubbles. Choose actual throughput = maximum throughput.

Variable Throughput  The number of bubbles changes over time. Usually the number of bubbles is unpredictable. Actual number of bubbles must be at least as great as minimum required.

4.4.2 Parcel Schedule

Actual Throughput and Parcel Schedule

To reduce confusion about the meaning of “throughput”, we will use:

- “throughput” means “maximum possible throughput”
- “parcel schedule” means “actual throughput”
- “as soon as possible (ASAP) parcel schedule” means actual throughput is constant and is the maximum possible
- “unpredictable number of bubbles” means actual throughput is variable
4.4.3 Valid Bits

When the parcel schedule is unpredictable number of bubbles, we need a mechanism to distinguish between a parcel and a bubble.

Most common solution: valid bit protocol.

<table>
<thead>
<tr>
<th>i_valid</th>
<th>i_data</th>
<th>o_valid</th>
<th>o_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td></td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>δ</td>
</tr>
</tbody>
</table>

α β γ δ

i_data

o_valid

o_data

Waveform for one-hot:

```
i_data
state(0)
state(1)
state(2)
o_data
```

Hardware implementation of one-hot:

```
reset
```

ASAP Parcels One-hot, binary, or custom.

Bubbles Valid bits

One-Hot State Encoding
Waveform for valid bits:

<table>
<thead>
<tr>
<th></th>
<th>i_valid</th>
<th>i_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>v(0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o_valid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o_data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hardware implementation of valid bits:

```
process begin
  wait until rising_edge(clk);
  if reset = '1' then
    state <= ______;
  else
    z <= b - c;
    z <= b + c;
  end if;
end process;
```

4.5 LeBlanc with Bubbles

Le Blanc with a parcel schedule of unpredictable number of bubbles.

```
\[ z' = b - c \]
\[ z' = b + c \]
```

4.6 Pseudocode

We use pseudocode to describe multi-step computation (e.g., algorithms).

**Declarations** We must declare “special” variables.

- **Inputs** Value might change in each clock cycle.
- **Interproc** section 4.7 Used to communicate between parcels
- **Outputs**
- **If-then-else**
- **While loop**
- **For loop**
- **Repeat-until loop**
- **Assignments**
- **Expressions** Arithmetic, logical, arrays, etc.

```
Example
input: a, b;
output: z;
p = a + b;
for i in 0 to 3 {
  p = p + b;
}
z = p;
```
Pseudocode Semantics

• Identical to conventional software semantics
• Executed sequentially: target is updated when the assignment is executed.
• All assignments are instantaneous: no reg vs comb.
• Variables hold value until assigned a new value.
• No notion of time or clock cycles.

Core vs System

• Pseudocode describes the core of a computation. It does not show the parcel schedule.
• But, with a finite sequence of parcels, the pseudocode may show more than 1 parcel.
• FSM for core does not show i_valid and o_valid.
• FSM for system (including parcel schedule) does show i_valid and o_valid if needed (i.e., if the parcel schedule is unpredictable number of bubbles).

4.7 Interparcel Variables and Loops

4.7.1 Introduction to Looping Le Blanc

Two new concepts:
• Inter-parcel variables
• Outer loop around “core”

Inter-parcel variables are used to communicate data between parcels.

Until now, all of our variables have been intra-parcel: used within a single parcel:

<table>
<thead>
<tr>
<th>Simple</th>
<th>Inter-parcel var T</th>
<th>Loop and inter-parcel var</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs a, b, c; outputs z; if a then { z = b + c; } else { z = b - c; }</td>
<td>inputs b, c; outputs z; interpcl T; if a then { T = T + b + c; } else { T = T + b - c; }</td>
<td>inputs b, c; outputs z; interpcl T; T = 0; for i in 0 to 127 { if a then { T = T + b + c; } else { T = T + b - c; } } z = T;</td>
</tr>
</tbody>
</table>

All intra-parcel vars

z = a + b + c

“Total is an inter-parcel variable

Total = Total + a + b

4.7.2 Pseudo-Code

We add variable declarations to distinguish inputs, outputs, and interparcel variables.

Below, “T” stands for “total”.

```plaintext
Simple | Inter-parcel var T | Loop and inter-parcel var
inputs a, b, c; outputs z; if a then { z = b + c; } else { z = b - c; } | inputs b, c; outputs z; interpcl T; if a then { T = T + b + c; } else { T = T + b - c; } | inputs b, c; outputs z; interpcl T; T = 0; for i in 0 to 127 { if a then { T = T + b + c; } else { T = T + b - c; } } z = T; ```
### State Machine Design Patterns

ASAP Parcels

Unpredictable number of bubbles

<table>
<thead>
<tr>
<th>State Machine Design Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASAP Parcels</td>
</tr>
<tr>
<td>Unpredictable number of bubbles</td>
</tr>
</tbody>
</table>

#### VHDL Code for Loop and Bubbles

```vhdl
v(0) <= i_v;
process begin
  wait until re(clk);
  if reset = '1' then
    v(1 to 4) = (others => '0');
  elsif v(4) and i >= 128 then
    v(1 to 4) = (others => '0');
  elsif v(1)='1' then
    total = total + b - c;
  elsif v(2) then
    total = total + b + c;
  end if;
end process;
```

```vhdl
process begin
  wait until re(clk);
  if reset = '1' then
    i = (others => '0');
  elsif v(3)='1' then
    i = i + 1;
  end if;
end process;
```

```vhdl
o_valid <= v(4) and i >= 128;
z <= total;
```
4.8 Memory Arrays and RTL Design

4.8.1 Memory Operations

Read of Memory

Hardware

<table>
<thead>
<tr>
<th>clk</th>
<th>WE</th>
<th>A</th>
<th>DI</th>
<th>DO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Behaviour

<table>
<thead>
<tr>
<th>clk</th>
<th>α</th>
<th>a</th>
<th>M(αa)</th>
<th>do</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>M(αa)</td>
<td></td>
</tr>
</tbody>
</table>

FSM

Write to Memory

Hardware

<table>
<thead>
<tr>
<th>clk</th>
<th>WE</th>
<th>A</th>
<th>DI</th>
<th>DO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Behaviour

<table>
<thead>
<tr>
<th>clk</th>
<th>α</th>
<th>a</th>
<th>M(αa)</th>
<th>do</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>M(αa)</td>
<td></td>
</tr>
</tbody>
</table>

FSM

4.8.2 Memory Arrays in VHDL

entity mem is
  generic (data_width : natural := 8; addr_width : natural := 7);
port (clk : in std_logic; wr_en : in std_logic; addr : in unsigned(addr_width - 1 downto 0); i_data : in data; o_data : out data);
end mem;

architecture main of mem is
  type mem_type is array(2**addr_width-1 downto 0) of std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if wr_en = '1' then
        mem(to_integer(addr)) <= i_data;
      end if;
      o_data <= mem(to_integer(addr));
    end if;
  end process;
end main;
4.8.3 Using Memory

Pseudocode

\[
\begin{align*}
M[i] &= a; \\
p &= M[i+1]; \\
M'[i] &= a; \\
M'[i+1] &= b;
\end{align*}
\]

Hardware:

```
u_mem : entity work.mem
  port map ( 
    clk => clk, 
    wr_en => mem_wr_en, 
    addr => mem_addr, 
    i_data => mem_i_data, 
    o_data => p;
  );
mem_wr_en <= '1' when state = S1 or state = S2;
mem_addr <= i when state = S1 else i + 1;
```

Question: How should we connect memory to \( p \) and \( q \)?

Multivar Reading (cont’d)

```
u_mem : entity work.mem
  port map ( 
    clk => clk, 
    wr_en => mem_wr_en, 
    addr => mem_addr, 
    i_data => mem_i_data, 
    o_data => mem_o_data;
  );
mem_wr_en <= '0';
mem_addr <= i when state = S1 else i + 1;
p <= mem_o_data when state = S2 else b;
q <= a when state = S2 else mem_o_data;
```
4.8.3.3 Example: Maximum Value Seen so Far

Design an FSM that iterates through a memory array, replacing each value with the maximum value seen so far.

Example execution:

<table>
<thead>
<tr>
<th>Initial value of M</th>
<th>Final value of M</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3 2 6 7 3 5</td>
<td></td>
</tr>
</tbody>
</table>

Pseudocode #1

```plaintext
i = 0
max = M[i]
while i < 128 {
    i = i + 1
    b = M[i]
    if max < b {
        max = b
    } else {
        M[i] = max
    }
}
```

Pseudocode #2

```plaintext
i = 0
while i < 128 {
    if max < b {
        max = b
    } else {
        M[i] = max
    }
}
```

4.8.4 Build Larger Memory from Slices

This section reserved for your reading pleasure
Chapter 5

Dataflow Diagrams

5.1 Dataflow Diagrams

5.1.1 Dataflow Diagrams Overview

- Dataflow diagrams are data-dependency graphs where the computation is divided into clock cycles.
- Purpose:
  - Provide a disciplined approach for designing datapath-centric circuits
  - Guide the design from algorithm, through high-level models, and finally to register transfer level code for the datapath and control circuitry.
  - Estimate area and performance
  - Make tradeoffs between different design options
- Background
  - Based on techniques from high-level synthesis tools
  - Some similarity between high-level synthesis and software compilation
  - Each dataflow diagram corresponds to a basic block in software compiler terminology.

Data-Dependency Graphs and Dataflow Diagrams

Models for \( z = a + b + c + d + e + f \)
Unconnected signal tails are inputs
Horizontal lines mark clock cycle boundaries
Signals crossing clock boundaries are flip-flops
Blocks in clock cycles are datapath components
Unconnected signal heads are outputs

5.1.2 Dataflow Diagram Execution

Definition Latency: Number of clock cycles from inputs to outputs.
- A combinational circuit has latency of zero.
- A single register has a latency of one.
- A chain of $n$ registers has a latency of $n$.

Latency =

5.1.3 Dataflow Diagrams, Hardware, and Behaviour

Primary Input

Dataflow Diagram

Behaviour
5.1.3 Dataflow Diagrams, Hardware, and Behaviour

Reuse a Component

5.1.4 Performance Estimation

Performance Equations

\[ \text{Performance} \propto \frac{1}{\text{TimeExec}} \]

\[ \text{TimeExec} = \text{Latency} \times \text{ClockPeriod} \]

Performance of Dataflow Diagrams

- Latency: count horizontal lines in diagram
- Min clock period (Max clock speed) limited by longest path in a clock cycle
5.1.5 Area Estimation

- Maximum number of **blocks in a clock cycle** is total number of that component that are needed
- Maximum number of **signals that cross a cycle boundary** is total number of registers that are needed
- Maximum number of **unconnected signal tails in a clock cycle** is total number of inputs that are needed
- Maximum number of **unconnected signal heads in a clock cycle** is total number of outputs that are needed
- These estimates are just approximations. Does not take into account:
  - Area and delay of control circuitry
  - Multiplexers on registers and datapath components
  - Relative area and delay of different components
  - Technology-specific features, constraints, and costs
- These estimates give lower bounds.
- Other constraints or design goals might force you to use more components. Examples:
  - Decreasing latency \( \Rightarrow \) larger area
  - Constraint on max number of registers \( \Rightarrow \) more datapath components

### Area Estimation

Implementation-technology factors, such as the relative size of registers, multiplexers, and datapath components, might force you to make tradeoffs that increase the number of datapath components to decrease the overall area of the circuit.

- With some FPGA chips, a 2:1 multiplexer has the same area as an adder.
- With some FPGA chips, a 2:1 multiplexer can be combined with an adder into one FPGA cell per bit.
- In FPGAs, registers are usually “free”, in that the area consumed by a circuit is limited by the amount of combinational logic, not the number of flip-flops.
Design Analysis 2 (Cont’d)

For each of the diagrams below, calculate the latency, minimum clock period, and minimum number of adders required.

<table>
<thead>
<tr>
<th>Latency</th>
<th>Clock period</th>
<th>Adders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design Analysis 3

5.2 Design Example: Hnatyshyn DFD

5.2.1 Requirements

- Functional requirement:
  - Compute the following formula: \( z = a + b + c \)

- Performance requirements:
  - Max clock period: flop plus (1 add)
  - Max latency: 2

- Cost requirements
  - Maximum of two adders
  - Unlimited registers
  - Maximum of three inputs and one output
  - Maximum of 5000 student-minutes of design effort

- Combinational inputs, registered outputs
- Parcels arrive as-soon-as-possible (ASAP)
5.2.2 Data-Dependency Graph

Requirements and algorithm:
\[ z = a + b + c \]
Create a data-dependency graph for the algorithm.

5.2.3 Initial Dataflow Diagram

Schedule operations into clock cycles

Area and performance analysis
- latency
- clock period
- inputs
- outputs
- registers
- adders

- Best-case analysis for a theoretical design
- No guarantee that we will achieve best-case (optimal) design
- Design process: systematic method to try to come close close to optimal design
- Start with sub-optimal, but obviously correct, design
- Series of optimizations to improve area and speed while avoiding bugs

5.2.4 Area Optimization

5.2.5 Assign Names to Registered Signals

We start our initial (sub-optimal) design.

Before we can write VHDL code for our dataflow diagram, we must assign a name to each internal registered value.

Optionally, we may assign names to combinational values.
### 5.2.6 Allocation

**Allocation** is the area optimization of mapping a large number of objects in current design to smaller number of objects.

- Example: allocate both \( x_i \) registers to the same register
- Similar to register allocation in software
- This design is so simple that allocation is trivial. For real designs, finding the best allocation is very difficult. Many different heuristics for how to do allocation.
- We will allocate inputs, outputs, registers, and datapath components.
- We will work clock-cycle by clock-cycle.
- Annotate dataflow diagram and fill in cells in I/O schedule and control table.

#### Use ASAP Parcel Schedule

**Question:** When to start parcel \( \beta \)?

#### Allocate Clock Cycle 0: Inputs and Datapath

**Question:** What is the maximum throughput that this system supports?
Allocate Clock Cycle 0: Regs

Allocate Clock-Cycle 1: Regs

Allocate Clock-Cycle 1: Inputs and Datapath

Allocate Output

With registered outputs, each output port must be connected directly to a register.
5.2.7 State Machine

- Done with datapath design and optimization
- Now build the control circuitry

Control-circuit optimizations:
- Choose state encoding
- Design state machine
- Design control circuitry that drives datapath
  - Multiplexer select lines
  - Chip enables
  - Operation selection

Transform control table:
- Label rows by state
- Add next-state column
- Identify "don't-care" values

Find Constants

If all of the cells in a column have the same value, then that column can be reduced to a constant.
Control Table, State Machine, Hardware

<table>
<thead>
<tr>
<th>state</th>
<th>src1</th>
<th>src2</th>
<th>rl</th>
<th>o1</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>i1</td>
<td>i2</td>
<td>i</td>
<td>d</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>r1</td>
<td>i2</td>
<td>i</td>
<td>d</td>
<td>S0</td>
</tr>
</tbody>
</table>

Control table for entire system

State machine for entire system

Hardware for entire system

5.2.8 VHDL Implementation

architecture main of hnatyshyn is
signal r1, a1, a1_src1 : unsigned(7 downto 0);
type state_ty is (S0, S1);
signal state : state_ty;
begin
---------------------------------------------
-- control
process (clk) begin
if rising_edge(clk) then
if reset = '1' then
state <= S0;
else
case state is
when S0 => state <= S1;
when S1 => state <= S0;
end case;
end if;
end if;
end process;
---------------------------------------------
-- registers
process (clk) begin
if rising_edge(clk) then
r1 <= a1;
end if;
end process;
---------------------------------------------
-- datapath
a1 <= a1_src1 + i2;
o1 <= r1;
---------------------------------------------
end architecture;

VHDL Implementation #2

- One-hot encoding for state
- Define constants for S0, S1
- Replace state = S0 with state(0) = '1'.

<table>
<thead>
<tr>
<th>state</th>
<th>src1</th>
<th>src2</th>
<th>rl</th>
<th>d</th>
<th>o1</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>i1</td>
<td>i2</td>
<td>i</td>
<td>d</td>
<td></td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>r1</td>
<td>i2</td>
<td>i</td>
<td>d</td>
<td></td>
<td>S0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const</th>
<th>i2</th>
<th>i</th>
<th>d</th>
<th>o1</th>
<th>rl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i1</td>
<td></td>
<td></td>
<td>d</td>
<td>S0</td>
</tr>
<tr>
<td></td>
<td>i2</td>
<td>i</td>
<td>d</td>
<td>o1</td>
<td>r1</td>
</tr>
</tbody>
</table>
architecture main of hnatyshyn is

signal r1, a1 : unsigned(7 downto 0);
subtype state_ty is std_logic_vector(1 downto 0);
constant S0 : state_ty := "01";
constant S1 : state_ty := "10";
signal state : state_ty;
begin
  --------------------------------------------
  -- control
  process (clk) begin
    if rising_edge(clk) then
      if reset = '1' then
        state = S0;
      else
        state <= state rol 1;
      end if;
    end if;
  end process;
  --------------------------------------------
  -- registers
  process (clk) begin
    if rising_edge(clk) then
      r1 <= a1;
    end if;
  end process;
  --------------------------------------------
  -- datapath
  a1 <= a1_src1 + i2;
o1 <= r1;
end architecture;

---

5.3 Design Example: Hnatyshyn with Bubbles

- section 5.2: Hnatyshyn with ASAP parcels
- This section: Hnatyshyn with unpredictable number of bubbles
- Key feature: valid bits for control circuitry

5.3.1 Adding Support for Bubbles

- No change to dataflow diagram (dataflow diagrams are independent of parcel schedule)
- Add i_valid and o_valid to denote whether input or output is parcel or bubble
- Add idle state to state machine for when there is not a parcel in the system
Add Valid Bits

Use Valid Bits as Control

5.3.2 Control Table with Valid Bits

**Initial Table**

- Label the rows of the control table by valid bits, instead of by states.
- Do not include a row for the last valid bit.
- We have registered outputs
- Therefore, no control decisions are made in the last clock cycle
- Therefore, the last valid bit does not affect the datapath

![Control Table with Valid Bits Diagram]
## 5.3.3 VHDL

The only difference between the VHDL code for Hnatyshyn with bubbles and Hnatyshyn with ASAP parcels is the control circuitry. The datapath is exactly the same for both designs.

```vhdl
-- control
v(0) <= i_valid;
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      v(1 to 2) <= (others => '0');
    else
      v(1 to 2) <= v(0 to 1);
    end if;
  end if;
end process;
a1_src1 <= i1 when v(0) = '1' else r1;
```

```vhdl
-- registers
process (clk) begin
  if rising_edge(clk) then
    r1 <= a1;
  end if;
end process;
```

```vhdl
-- datapath
a1 <= a1_src1 + i2;
o_valid <= v(2);
o1 <= r1;
end architecture;
```

```vhdl
entity hnatyshyn_bubble is
  port (clk : in std_logic;
        i_valid : in std_logic;
        i1, i2 : in unsigned(7 downto 0);
        o_valid : out std_logic;
        o1 : out unsigned(7 downto 0));
end entity;
```

```vhdl
architecture main of hnatyshyn_bubble is
  signal r1, a1, a1_src1 : unsigned(7 downto 0);
  signal v : std_logic_vector(0 to 2);
begin
```

```vhdl
-- control
v(0) <= i_valid;
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      v(1 to 2) <= (others => '0');
    else
      v(1 to 2) <= v(0 to 1);
    end if;
  end if;
end process;
a1_src1 <= i1 when v(0) = '1' else r1;
```

```vhdl
-- registers
process (clk) begin
  if rising_edge(clk) then
    r1 <= a1;
  end if;
end process;
```

```vhdl
-- datapath
a1 <= a1_src1 + i2;
o_valid <= v(2);
o1 <= r1;
end architecture;
```
Inter-parcel variables are used to communicate data between parcels.

Previous systems: \[ z = a + b + c \]

“Sum” is an inter-parcel variable: \[ \text{Sum} = \text{Sum} + a + b \]

**intra-parcel variables** The type of variables and signals that we have used until now

- Also called “temporary values”
- Stores intermediate data from clock-cycle to clock-clock cycle
- Each value is read only by the same parcel that wrote the value

**inter-parcel variables** The new type of variables and signals

- Also called “programmer-visible”, “internal-state”, or “visible-state” variables
- Stores data that is used to communicate between parcels
- Each value is written by one parcel and then read by other parcels

### 5.4.1 Requirements and Goals

- **Functional requirements**: compute the following formula: \[ \text{Sum} = \text{Sum} + a + b \]
- **Performance requirement**:
  - Max clock period: flop plus (1 add)
  - Max latency: 3
- **Cost requirements**
  - Maximum of two adders
  - Unlimited registers
  - Maximum of three inputs and one output
  - Maximum of 5000 student-minutes of design effort
- **Combinational inputs**
- **Registered outputs**
- **Parcel schedule is “Unpredictable number of bubbles”**

### 5.4.2 Dataflow Diagrams and Waveforms

**Bad DFD**

**Question**: What is wrong with the dataflow diagram below?

---

**Diagram**: Dataflow diagrams and waveforms.
States and Bubbles

Question: Label the states on the DFD and execution. Complete the FSM.

DFD

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>i1</td>
</tr>
<tr>
<td>i2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSM

<table>
<thead>
<tr>
<th>state</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>i2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>a</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
</tbody>
</table>

Execution

<table>
<thead>
<tr>
<th>state</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>i2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>a</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
</tbody>
</table>

Initial Control Table

<table>
<thead>
<tr>
<th>state</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>i2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>a</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
</tbody>
</table>

Reset

<table>
<thead>
<tr>
<th>state</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>i2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>a</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r1</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>r2</td>
<td>α</td>
<td>β</td>
<td>γ</td>
</tr>
</tbody>
</table>

VHDL Code for Control Circuitry

The VHDL code for just the control circuitry is below. In section 5.4.4, we show the complete code.

```vhdl
a1_src1 <= i1 when v(0) = '1' else r1;
a1_src2 <= i2 when v(0) = '1' else r2;
r1_ce <= v(0) or v(1);
```
5.4.4 VHDL Implementation

-- valid bits
v(0) <= i_valid;
process begin
    wait until rising_edge(clk);
    if reset = '1' then
        v(1 to 2) <= (others => '0');
    else
        v(1 to 2) <= v(0 to 1);
    end if;
end process;

-- a1
a1_src1 <= i1 when v(0) = '1' else r1;
a1_src2 <= i2 when v(0) = '1' else r2;
a1 <= a1_src1 + a1_src2;

-- r1
process begin
    wait until rising_edge(clk);
    if reset = '1' then
        r1 <= (others => '0');
    elsif v(0) = '1' or v(1) = '1' then
        r1 <= a1;
    end if;
end process;

-- r2
process begin
    wait until rising_edge(clk);
r2 <= r1;
end process;

-- outputs
o_valid <= v(2);
o1 <= r1;

5.4.5 Summary of Bubbles and Inter-Parcel Variables

Options for state encoding:

<table>
<thead>
<tr>
<th></th>
<th>System has Inter-pcl vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>State encoding</td>
<td>No</td>
</tr>
<tr>
<td>ASAP</td>
<td>State encoding</td>
</tr>
<tr>
<td>FSM has idle state</td>
<td>Ctrl table hhas idle row</td>
</tr>
<tr>
<td>Bubbles</td>
<td>State encoding</td>
</tr>
<tr>
<td>FSM has idle state</td>
<td>Ctrl table hhas idle row</td>
</tr>
</tbody>
</table>

5.5 Design Example: Vanier

Design Process

1. Requirements
2. Algorithm
3. Data-dependency graph
4. Schedule
5. Allocate I/O ports, datapath components, registers
6. Separate datapath and control
7. Connect datapath, add muxes
8. Block-diagram of datapath
9. Control-table for state machine
10. Don’t-care assignments
11. VHDL code #1 (core)
12. Parcel schedule
13. State encoding
14. VHDL code #2 (system)
5.5.1 Requirements

- Functional requirements: compute the following formula:
  \[ z = (a \times d) + c + (d \times b) + b \] for sixteen-bit unsigned data.
- Performance requirement:
  - Max clock period: flop plus (2 adds or 1 multiply)
  - Max latency: 4
- Cost requirements
  - Maximum of two adders
  - Maximum of two multipliers
  - Unlimited registers
  - Maximum of three inputs and one output
  - Maximum of 5000 student-minutes of design effort
- Combinational inputs
- Registered outputs
- ASAP parcel schedule

5.5.2 Algorithm

\[ z = (a \times d) + c + (d \times b) + b \]

Create a data-dependency graph for the algorithm.

5.5.3 Initial Dataflow Diagram

Schedule operations into clock cycles.

Requirement for max clock period: \( \text{max}(2 \text{ add, mul}) + \text{flop} \).

Area and performance analysis

<table>
<thead>
<tr>
<th>latency</th>
<th>clock period</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

inputs
outputs
registers
adders
multipliers

5.5.4 Reschedule to Meet Requirements

Requirement: no more than 3 inputs.
5.5.5 Optimization: Reduce Inputs

Assume that inputs are much more expensive than other resources.

**Question:** Should we move the second addition from clock-cycle 2 up to 1?
5.5.6 Allocation

I/O

<table>
<thead>
<tr>
<th>m1</th>
<th>a1</th>
<th>a2</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>o1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc1</td>
<td>sc1</td>
<td>sc2</td>
<td>ce</td>
<td>d</td>
<td>ce</td>
<td>d</td>
</tr>
</tbody>
</table>

0 0
1 1
2
3

const
needs mux
needs ce
Alternative Allocation

![Diagram of Alternative Allocation]

<table>
<thead>
<tr>
<th>I/O</th>
<th>m1</th>
<th>a1</th>
<th>a2</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>o1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i1</td>
<td>i2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>d</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>const</td>
<td>i1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Needs mux
- Needs ce

5/9
0/3
5.5.7 Explicit State Machine

From Clock Cycles to States

- ASAP parcel schedule
- Latency is 3, therefore 3 states (S0, S1, S2)
- State machine iterates through states, with S2 looping back to S0.

5.5.8 VHDL #1: Explicit

```vhdl
architecture main of vanier is
signal r1, r2, r3,a1, a1_src1, a1_src2, a2, m1, m1_src2: unsigned(15 downto 0);
type state_ty is (S0, S1, S2);
signal state : state_ty;
begina------------------------
    control
        process (clk) begin
            if rising_edge(clk) then
                if reset = '1' then
                    state <= S0;
                else
                    case state is
                        when S0 => state <= S1;
                        when S1 => state <= S2;
                        when S2 => state <= S0;
                    end case;
                end if;
            end if;
        end process;
    end control
------------------------
    datapath
        process (clk) begin
            if rising_edge(clk) then
                r1 <= i1;
            end if;
        end process;
        process (clk) begin
            if rising_edge(clk) then
                r2 <= m1;
            end if;
        end process;
        process (clk) begin
            if rising_edge(clk) then
                r3 <= i2;
            end if;
        end process;
        o1 <= r3;
e nd architecture;

------------------------
    registers
        process (clk) begin
            if rising_edge(clk) then
                if state = S0 then
                    r1 <= i1;
                else
                    r1 <= r2;
                end if;
            end if;
        end process;
        process (clk) begin
            if rising_edge(clk) then
                r2 <= m1;
            end if;
        end process;
        process (clk) begin
            if rising_edge(clk) then
                r3 <= i2;
            end if;
        end process;
------------------------
    datapath
        m1_src2 <= i2 when state = S0
            else r1;
        m1 <= i1(7 downto 0) * m1_src2(7 downto 0);
        a1_src1 <= r3 when state = S1
            else a2;
        a1_src2 <= i1 when state = S1
            else a2;
        a1 <= a1_src1 + a1_src2;
        a2 <= r1 + r3;
------------------------
    State Encoding

Use a one-hot state encoding.
```
**Don’t Care: Encoding-Based Instantiations**

For this simple example, the encoding-based instantiations are trivial.

---

### 5.5.10 Notes and Observations

Our functional requirement was written as:

\[ z = (a \times d) + (d \times b) + b + c \]

If we had been given the functional requirement:

\[ z = (a \times d) + b + (d \times b) + c \]

we could have used the same design, because the two equations are equivalent.
Data Dependency Graphs: Clean vs Ugly

The naive data dependency graph for the second formulation is much messier than the data dependency graph for the original formulation:

**Original**

\[(a \times d) + (d \times b) + b + c\]

**Alternative**

\[(a \times d) + b + (d \times b) + c\]

5.6 Memory Operations in Dataflow Diagrams

<table>
<thead>
<tr>
<th>Memory Read</th>
<th>Memory Write</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read</strong></td>
<td><strong>Write</strong></td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td></td>
</tr>
</tbody>
</table>

Memory Read

**Hardware**

Dataflow diagram

**Behaviour**

FSM

Memory Write

**Hardware**

Dataflow diagram

**Behaviour**

FSM
5.7 Data Dependencies

Definition of Three Types of Dependencies

- **Read after Write**
  - (True dependency)

- **Write after Write**
  - (Load dependency)

- **Write after Read**
  - (Anti dependency)

Instructions in a program can be reordered, so long as the data dependencies are preserved.

Sequence of Memory Operations

Purpose of Dependencies

- **WAW ordering** prevents \( W_0 \) from happening after \( W_1 \).
- **RAW ordering** prevents \( R_1 \) from happening before \( W_1 \).
- **WAR ordering** prevents \( W_i \) from happening before \( R_j \).

\[ M[i] := \]

\[ \ldots \Rightarrow M[i] \ldots \]

\[ M[i] := \]

\[ \ldots \Rightarrow M[i] \ldots \]

\[ M[i] := \]

\[ \ldots \Rightarrow M[i] \ldots \]
Ordering of Memory Operations

Data Dependencies

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

M[2] := 21
M[3] := 31
A := M[2]
B := M[0]
M[0] := 01
C := M[3]

Initial Program

Data Dependencies (Cont’d)

5.7. DATA DEPENDENCIES

This section examines the implementation of the pseudocode specification:

M[a+1] = b;
M[a] = M[a+1];
M[c] = M[c] - M[a];
z = M[c]

5.8 Example of DFD and Memory

Valid Modification
NOTES:
1. Inputs shall be *combinational*
2. Outputs shall be *registered*
3. The system shall support an *unpredictable number of bubbles*
4. Memory has combinational inputs and registered outputs (same as in class)
5. The memory may be *either dual-ported or single-ported.*
6. Optimization goals in order of decreasing importance:
   (a) minimize *latency* to \( z \)
   (b) minimize *clock period*
   (c) minimize *area*
      i. input ports
      ii. adders and subtracters
      iii. registers
      iv. output ports
      v. use single-ported memory instead of dual-ported memory
7. Input values may be read in any clock cycle, but each input value shall be read exactly once.
8. Optimizations to the pseudocode are allowed, as long as the final values of \( z \) and \( M \) are correct.
9. You do *not* need to do allocation.

### Pseudocode Optimization

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M[a+1] = b; )</td>
<td></td>
</tr>
<tr>
<td>( M[a] = M[a+1]; )</td>
<td></td>
</tr>
<tr>
<td>( M[c] = M[c] - M[a]; )</td>
<td></td>
</tr>
<tr>
<td>( z = M[c] )</td>
<td></td>
</tr>
</tbody>
</table>

### Memory Ports

How many ports does your memory have? [ ]

Briefly justify that your choice of number of memory ports produced the most optimal design.
Chapter 6

Optimizations

6.1 Pipelining

- Exploit “hardware runs in parallel”
- Performance optimization at cost of increased area
- Overlap the execution of multiple parcels
- Divide design into stages
- Maximum of one parcel executing per stage
- No sharing of hardware between stages

6.1.1 Introduction to Pipelining
### Sequential (Unpipelined) Hardware

**Question:** Parcel schedule?

**Question:** State encoding?

**Question:** Control circuitry?

### Pipelined Hardware and VHDL Code

**6.1.2 Partially Pipelined**
- Fully pipelined: throughput is one parcel per clock cycle
- Partially pipelined: throughput is less than one parcel per clock cycle.
- Superscalar: throughput is more than one parcel per clock cycle.

**Question:** How do we execute α followed by β?
6.1.3 Terminology

Definition Depth: The depth of a pipeline is the number of stages on the longest path through the pipeline.

Definition Latency: The latency of a pipeline is measured the same as for an unpipelined circuit: the number of clock cycles from inputs to outputs.

Definition Throughput: The number of parcels consumed or produced per clock cycle.

Definition Upstream/downstream: Because parcels flow through the pipeline analogously to water in a stream, the terms upstream and downstream are used respectively to refer to earlier and later stages in the pipeline. For example, stage 1 is upstream from stage 2.

Definition Bubble: When a pipe stage is empty (contains invalid data), it is said to contain a “bubble”.

6.1.4 Overlapping Pipeline Stages

- A single parcel may be in multiple stages at the same time
  
  Example: Store instruction in a microprocessor uses separate stages for address and data

- Transferring a parcel between stages may require multiple clock cycles
  
  Example: 16 × 16 macroblock of pixels in video processing

Illustrate overlapping pipe stages with a simple example.
Design Space Exploration (Cont’d)

Goal: Maximum throughput using just 1 F and 1 G

Implementation of Overlapping Stages

```vhdl
-- valid bits
v(0) <= i_valid;
process begin
  wait until rising_edge(clk);
  v(6 downto 1) <= v(5 downto 0);
end process;

-- stage 1
f1_src2 <= i2 whenelse r1;
process begin
  wait until rising_edge(clk);
  r1 <= f( i1, f1_src2 );
  r2 <= r1;
end process;
```
Implementation of Overlapping (Cont’d)

Review: Pipelining

Analyze the dataflow diagram below.

<table>
<thead>
<tr>
<th>Stages</th>
<th>Latency</th>
<th>Clock period</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.2 Staggering

This is an advanced section. It is not covered in the course and will not be tested.

6.3 Retiming

Goal: decrease clock period without changing input-to-output behaviour of the system.

Technique: move registers “through” gates to balance delay between registers.

Retime to balance delays

Push flop through AND

Push flop through wire fork
**Question:** Do the two circuits below have the same behaviour?

Extra copy for scratch work:

---

**Example with State Machine**

**Original behaviour**

```vhdl
process (state) begin
  if state = \texttt{S1} then
    sel = \texttt{'1'};
  else
    sel = \texttt{'0'};
  end if;
end process;

process begin
  wait until rising_edge(clk);
  if sel = \texttt{'1'} then
    \textcolor{red}{\textit{... code for } z}\textcolor{black}{;}
  end if;
end process;
```

**Retimed**

```vhdl
process begin
  wait until rising_edge(clk);
  if state = \texttt{S1} then
    sel = \texttt{'1'};
  else
    sel = \texttt{'0'};
  end if;
end process;

process begin
  wait until rising_edge(clk);
  if sel = \texttt{'1'} then
    \textcolor{red}{\textit{... code for } z}\textcolor{black}{;}
  end if;
end process;
```
6.4 General Optimizations

6.4.1 Strength Reduction

Strength reduction replaces one operation with another that is simpler.

6.4.1.1 Arithmetic Strength Reduction

Multiply by a constant power of two  
Multiply by a power of two  
Divide by a constant power of two  
Divide by a power of two  
Multiply by 3

wired shift logical left  
shift logical left  
wired shift logical right  
shift logical right  
wired shift and addition

6.4.1.2 Boolean Strength Reduction

Boolean tests that can be implemented as wires

- is_odd, is_even
- is_neg, is_pos

By choosing your encodings carefully, you can sometimes reduce a vector comparison to a wire.

For example if your state uses a one-hot encoding, then the comparison \( \text{state} = S3 \) reduces to \( \text{state}(3) = '1' \). You might expect a reasonable logic-synthesis tool to do this reduction automatically, but most tools do not do this reduction.

When using encodings other than one-hot, Karnaugh maps can be useful tools for optimizing vector comparisons. By carefully choosing our state assignments, when we use a full binary encoding for 8 states, the comparison:

\[
(\text{state} = S0 \text{ or } \text{state} = S3 \text{ or } \text{state} = S4) = '1'
\]

can be reduced from looking at 3 bits, to looking at just 2 bits. If we have a condition that is true for four states, then we can find an encoding that looks at just 1 bit.
6.4.2 Replication and Sharing

6.4.2.1 Mux-Pushing

Pushing multiplexors into the fanin of a signal can reduce area.

Before
\[
z <= a + b \text{ when } (w = '1') \\
\text{else } a + c;
\]

After
\[
tmp <= b \text{ when } (w = '1') \\
\text{else } c; \\
z <= a + tmp;
\]
The first circuit will have two adders, while the second will have one adder. Some synthesis tools will perform this optimization automatically, particularly if all of the signals are combinational.

6.4.2.2 Common Subexpression Elimination

Introduce new signals to capture subexpressions that occur multiple places in the code.

Before
\[
y <= a + b + c \text{ when } (w = '1') \\
\text{else } d; \\
z <= a + c + d \text{ when } (w = '1') \\
\text{else } e;
\]

After
\[
tmp <= a + c; \\
y <= b + tmp \text{ when } (w = '1') \\
\text{else } d; \\
z <= d + tmp \text{ when } (w = '1') \\
\text{else } e;
\]

6.4.2.3 Computation Replication

- To improve performance
  - If same result is needed at two very distant locations and wire delays are significant, it might improve performance (increase clock speed) to replicate the hardware
- To reduce area
  - If same result is needed at two different times that are widely separated, it might be cheaper to reuse the hardware component to repeat the computation than to store the result in a register

Note: Muxes are not free. Each time a component is reused, multiplexors are added to inputs and/or outputs. Too much sharing of a component can cost more area in additional multiplexors than would be spent in replicating the component.

Subexpression Elimination

Note: Clocked subexpressions. Care must be taken when doing common subexpression elimination in a clocked process. Putting the “temporary” signal in the clocked process will add a clock cycle to the latency of the computation, because the tmp signal will be flip-flop. The tmp signal must be combinational to preserve the behaviour of the circuit.
6.4.3 Arithmetic

Perform arithmetic on the minimum number of bits needed. If you only need the lower 12 bits of a result, but your input signals are 16 bits wide, trim your inputs to 12 bits. This results in a smaller and faster design than computing all 16 bits of the result and trimming the result to 12 bits.

6.5 Customized State Encodings

This is an advanced section. It is not covered in the course and will not be tested.

Chapter 7

Performance Analysis
### 7.1 Introduction

Hennessey and Patterson’s *Quantitative Computer Architecture* (textbook for E&CE 429) has good information on performance. We will use some of the same definitions and formulas as Hennessey and Patterson, but we will move away from generic definitions of performance for computer systems and focus on performance for digital circuits.

### 7.2 Defining Performance

#### Performance

\[ \text{Performance} = \frac{\text{Work}}{\text{Time}} \]

You can double your performance by:

- doing twice the work in the same amount of time
- OR doing the same amount of work in half the time

#### Benchmarking

Measuring time is easy, but how do we accurately measure work?

The game of benchmarketing is finding a definition of work that makes your system appear to get the most work done in the least amount of time.

<table>
<thead>
<tr>
<th>Measure of Work</th>
<th>Measure of Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock cycle</td>
<td>MHz</td>
</tr>
<tr>
<td>instruction</td>
<td>MIPs</td>
</tr>
<tr>
<td>synthetic program</td>
<td>Whetstone, Dhrystone, D-MIPs (Dhrystone MIPs)</td>
</tr>
<tr>
<td>real program</td>
<td>SPEC (PCs), EEMBC (Embedded)</td>
</tr>
<tr>
<td>travel 1/4 mile</td>
<td>drag race</td>
</tr>
</tbody>
</table>

#### Throughput vs Latency

Two common measures of performance:

- **Latency**  Response time
- **Throughput** Bandwidth

Often there is a tradeoff between latency and bandwidth

- For *general-purpose systems*, *throughput* is usually most important.
- For *real-time systems*, *latency* is often most important.
7.3 Benchmarks

Historical Benchmarks

MIPS  Millions of instructions per second (My NOP instruction is faster than yours)

Whetstone
- First general-purpose benchmark for computer performance
- Synthetic: an artificial program designed to be quick and easy to run, but reflects the performance of a real program.
- Based on the Algol-60 compiler developed by Atomic Power Division of the English Electric Company, Whetstone, Leicester, England, for the KDF9 Computer.

Dhrystone  pun on Whetstone

D-MIPS  MIPS using Dhrystone mix of instructions
- Synthetic benchmarks worked well for computers in 1970s and 1980s.
- As caches became larger, entire synthetic program could fit in first-level cache, which resulted in unrealistic performance.

SPEC Benchmarks

The Spec Benchmarks are among the most respected and accurate predictions of real-world performance for desktop PCs and servers.

Definition SPEC:  Standard Performance Evaluation Corporation MISSION:
“To establish, maintain, and endorse a standardized set of relevant benchmarks and metrics for performance evaluation of modern computer systems http://www.spec.org.”

The Spec organization has different benchmarks for integer software, floating-point software, web-serving software, etc.

7.4 Comparing Performance

7.4.1 General Equations
**Speedup**

Example sentences:
- A new system has \( n \)-times the performance of the old system.
- This optimization provides a \( n \times \) speedup.

\[
\text{Speedup}(\text{New}, \text{Old}) = \frac{\text{Perf}_{\text{New}}}{\text{Perf}_{\text{Old}}}
\]

Using speedup to calculate performance:

\[
\text{Perf}_{\text{New}} = \frac{\text{Perf}_{\text{Old}}}{\text{Speedup}}
\]

Using \( \text{Perf}_{\text{High}} \) and \( \text{Perf}_{\text{Low}} \):

\[
\text{Speedup} = \frac{\text{Perf}_{\text{High}}}{\text{Perf}_{\text{Low}}}
\]

**Performance vs Time**

Performance is inversely proportional to time:

\[
\text{Perf} = \frac{1}{\text{Time}}
\]

Using time to measure performance, the equation for speedup is:

\[
\text{Speedup}(\text{New}, \text{Old}) = \frac{\text{Perf}_{\text{New}}}{\text{Perf}_{\text{Old}}} = \frac{1/\text{Time}_{\text{New}}}{1/\text{Time}_{\text{Old}}} = \frac{\text{Time}_{\text{Old}}}{\text{Time}_{\text{New}}}
\]

Using \( \text{Time}_{\text{Slow}} \) and \( \text{Time}_{\text{Fast}} \):

\[
\text{Speedup} = \frac{\text{Time}_{\text{Slow}}}{\text{Time}_{\text{Fast}}}
\]

**Bigger Than and Smaller Than**

Equation for “New is \( n \)% bigger than Old”:

\[
PctBigger = \frac{\text{New} - \text{Old}}{\text{Old}}
\]

New is \( n \)% smaller than Old:

\[
PctSmaller = \frac{\text{Old} - \text{New}}{\text{Old}}
\]

Derive \( n \)% bigger from speedup:

\[
PctBigger = \text{Speedup} - 1 = \frac{\text{New}}{\text{Old}} - 1 = \frac{\text{New}}{\text{Old}} - \frac{\text{Old}}{\text{Old}} = \frac{\text{New} - \text{Old}}{\text{Old}}
\]

**Bigger Than (Cont’d)**

The performance of \( \text{New} \) is \( n \)% bigger than the performance of \( \text{Old} \):

\[
PctBigger = \frac{\text{Perf}_{\text{New}} - \text{Perf}_{\text{Old}}}{\text{Perf}_{\text{Old}}}
\]

Use percentage-bigger to write equation for \( \text{Perf}_{\text{New}} \) in terms of \( \text{Perf}_{\text{Old}} \):

\[
\text{Perf}_{\text{New}} = (\text{PctBigger} + 100\%) \text{Perf}_{\text{Old}}
\]

OR, Equivalently:

\[
\text{Perf}_{\text{New}} = \text{PctBigger} \times \text{Perf}_{\text{Old}} + \text{Perf}_{\text{Old}}
\]
Converting between Bigger and Smaller Than

Question: If A is $n\%$ bigger than B, how smaller is B than A?

Average Performance of Multiple Tasks

Another useful formula is the average time to do one of $k$ different tasks, each of which happens $\%i$ of the time and takes an amount of time $T_i$ to do each time it is done.

$$T_{\text{Avg}} = \sum_{i=1}^{k} (\%i)(T_i)$$

7.4.2 Example: Performance of Printers

This section reserved for your reading pleasure

7.5 Clock Speed, CPI, Program Length, and Performance

7.5.1 Mathematics

<table>
<thead>
<tr>
<th>CPI</th>
<th>Cycles per instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumInsts</td>
<td>Number of instructions</td>
</tr>
<tr>
<td>ClockSpeed</td>
<td>Clock speed</td>
</tr>
<tr>
<td>ClockPeriod</td>
<td>Clock period</td>
</tr>
</tbody>
</table>

$$\text{Time} = \text{NumInsts} \times \text{CPI} \times \text{ClockPeriod}$$

$$\text{Time} = \frac{\text{NumInsts} \times \text{CPI}}{\text{ClockSpeed}}$$
7.5.2 Example: CISC vs RISC and CPI

Clock Speed  SPECint
AMD Athlon 1.1GHz  409
Fujitsu SPARC64 675MHz  443

The AMD Athlon is a CISC microprocessor (it uses the IA-32 instruction set). The Fujitsu SPARC64 is a RISC microprocessor (it uses Sun’s Sparc instruction set). Assume that it requires 20% more instructions to write a program in the Sparc instruction set than the same program requires in IA-32.

### SPECint and Performance

<table>
<thead>
<tr>
<th>Clock Speed</th>
<th>SPECint</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Athlon 1.1GHz</td>
<td>409</td>
</tr>
<tr>
<td>Fujitsu SPARC64 675MHz</td>
<td>443</td>
</tr>
</tbody>
</table>

**Question:** Which of the two processors has higher performance?

### Absolute CPI

**Question:** Can you determine the absolute (actual) CPI of either microprocessor?

### Relative CPI

**Question:** What is the ratio between the CPIs of the two microprocessors?
7.5.3 Effect of Instruction Set on Performance

In this section we examine how changing the instructions that a processor performs can effect its performance.

Your group designs a microprocessor and you are considering adding a fused multiply-accumulate to the instruction set.

- Fused multiply-accumulate instruction does a multiply and an add
  
  \[ \text{op src1 src2 dst} \]
  \[ \text{MAC R1, R2, R4 = MUL R1, R2, R3 ADD R3, R4, R4} \]

- Often used in digital signal processing. See the multiply-accumulate pattern in the finite-impulse-response filter below:

- First added to RISC instruction sets by IBM with its POWER processor family: “Performance With Enhanced Risc”.

### Using MAC Instruction

#### Original program
- MUL R1, R2, R3
- ADD R4, R3, R4
- SUB R5, R7, R9
- MUL R1, R2, R3
- ADD R5, R3, R5
- SUB R1, R2, R3
- MUL R2, R3, R5
- ADD R5, R2, R5

#### Using MAC
- MAC R1, R2, R4 = MUL R1, R2, R3 ADD R3, R4, R4

### Options

You have three options:

- **option 1**: no change
- **option 2**: add the MAC instruction, increase the clock period by 20%, and MAC has the same CPI as MUL.
- **option 3**: add the MAC instruction, keep the clock period the same, and the CPI of a MAC is 50% greater than that of a multiply.

**Question**: Which option will result in the highest overall performance?
7.6 Effect of Time to Market on Relative Performance

The performance of digital-hardware based system has grown historically at an exponential rate.

To illustrate this concept, imagine companies A and B release competing products ($A_1, B_1, A_2, B_2, A_3, B_3, ...$) over a series of years where the performance of the average product in this category doubles every year.

Equation for exponential growth, where $P$ increases by a factor of $n$ every $k$ units of time:

$$P(t_1) = P(t_0) \times n^{(t_1 - t_0)/k}$$

---

Review: Performance of Programs

Which option is better:

1. 90% performance improvement to 10% of instructions
2. 10% performance improvement to 90% of instructions
Example Problem

Assume that performance of the average product in your market segment doubles every 18 months.

You are considering an optimization that will improve the performance of your product by 7%.

**Question:** If you add the optimization, how much can you allow your schedule to slip before the delay hurts your relative performance compared to not doing the optimization and launching the product according to your current schedule?
8.1 Delays and Definitions

In this section we will look at the different timing parameters of circuits. Our focus will be on those parameters that limit the maximum clock speed at which a circuit will work correctly.

8.1.1 Background Definitions

This section reserved for your reading pleasure

8.1.2 Clock-Related Timing Definitions

At the register transfer level, we think of the system as having a single, global clock.

On the chip, the single clk signal in our source code is implemented as a clock tree containing many buffers and individual wires. On the physical chip, each flip-flop has its own clock signal.

```
begin process
    wait until rising_edge(clk);
    c <= a + b;
    e <= c - d;
end process;
```
8.1.2.1 Clock Latency

**Definition** Clock Latency: The delay from the source (oscillator) to a point in the clock tree.

**Note:** Clock latency does not affect the limit on the minimum clock period.

8.1.2.2 Clock Skew

**Definition** Clock Skew: The difference in arrival times for the same clock edge at different flip-flops.

Clock skew is caused by the difference in interconnect delays to different points on the chip.

\[
\text{Skew}(\text{clk}_{1.0.1}, \text{clk}_{2.1.1}) = |\text{Latency}(\text{clk}_{1.0.1}) - \text{Latency}(\text{clk}_{2.1.1})|
\]

8.1.2.3 Clock Jitter

**Definition** Clock Jitter: Difference between actual clock period and ideal clock period.

Clock jitter is caused by:
- temperature and voltage variations over time
- temperature and voltage variations across different locations on a chip
- manufacturing variations between different parts
Clock Tree Design

Clock tree design is critical in high-performance designs to minimize clock skew. Sophisticated synthesis tools put lots of effort into clock tree design, and the techniques for clock tree design still generate PhD theses.

8.1.3 Storage-Related Timing Definitions

8.1.3.1 Flops and Latches

Storage devices have two modes: load mode and store mode.

Flops are edge sensitive; they are in load mode just before the clock edge.

Latches are level sensitive; they are in load mode while their enable signal is asserted high (low for active low latches).

8.1.3.2 Timing Parameters

In the pictures below, the goal is to load the data $\alpha$ into the flip-flop or latch.

Setup and hold define the window in which input data are required to be constant in order to guarantee that the storage device will store data correctly.

Clock-to-Q defines the delay from the clock edge to when the output is guaranteed to be stable.

8.1.3.3 Timing Parameters for a Flop

Good timing

Too slow = setup violation

Too fast = hold violation
8.1.4 Propagation Delays

**Propagation delay** time it takes a signal to travel from the source (driving) flop to the destination flop

**Load delay** combinational gates between the flops

**Interconnect delay** wires between gates and flops

**Propagation delay** = load delay + interconnect delay

8.1.5 Timing Constraints

**Minimum Clock Period**

- **Hold Constraint**
  - **Simple clock-to-q**
  - **Realistic clock-to-q**

8.1.5 Timing Constraints

**Review: Timing Parameters**

1. **Setup**: Time ___________ ___________ transition that input data is ___________ to ___________ being stable
2. **Hold**: time ___________ ___________ transition that input data is ___________ to ___________ stable
3. **Clock-to-Q-Min**: Time ___________ ___________ when output data is ___________ to ___________ stable with old data
4. **Clock-to-Q-Max**: Time ___________ ___________ when output data is ___________ to ___________ stable with new data

ClockPeriod > ________________________________
5. How do you fix a setup violation?

6. How do you fix a hold violation?

7. Draw a timing diagram for a flip-flop with the following timing parameters:
   - setup: 2 ns
   - hold: 1 ns
   - clock-to-Q: 3 ns

8. Draw a timing diagram for an active-high latch with the following timing parameters:
   - setup: 2 ns
   - hold: 1 ns
   - clock-to-Q: 3 ns

8.2 Timing Analysis of Simple Latches

In this section, each gate has a delay of 1 time unit.

8.2.1 Review: Active-High Latch Behaviour

The functionality of storage devices depends on timing.
- For functionality at the register transfer level, we ignore timing (combinational logic has zero delay).
- For storage devices, functionality depends on timing (delays through combinational logic).
- Ignoring delays, the circuits above are equivalent, but the circuit on the right is actually incorrect.
- The pair of inverters on the clk signal are needed. Together, they prevent a glitch on the OR gate when clk is deasserted. If there was only one inverter, a glitch would occur. For more on this, see section 8.2.8.
8.2.3 Strategy for Timing Analysis of Storage Devices

The key to calculating setup and hold times of a latch, flop, etc is to identify:

1. how the data is stored when in storage mode (often a combinational loop with a pair of inverters)
2. the gate(s) that the clock uses to turn on the load path (allow the input to affect the internals of the storage element)
3. the gate(s) that the clock uses to turn on the storage loop (allow the stored data continue to circulate through the storage loop)
4. the gate where the load path and storage loop join
8.2.4 Clock-to-Q Time of a Latch
8.2.5 From Load Mode to Store Mode

Circuit is stable in load mode

$t=0$: Clk transitions from load to store

$t=1$: Clk edge propagates through inverter

$t=2$: $s_1$ propagates to $s_2$, because $c_n$ turns on AND gate

$t=3$: $l_2$ is set to 0, because $c_2$ turns off AND gate

$t=4$: $\alpha$ from store path propagates to $q$

$t=5$: $\alpha$ from store path completes cycle
8.2.6 Setup Time Analysis

1. When the latch is in store mode, there must be consistent values in the storage loop. Otherwise, the loop will be metastable.

2. As the circuit transitions from load mode to store mode, there must be consistent values at the point where the load and store paths join.

3. We must saturate the storage loop with the current value ($\alpha$) before we turn on the storage loop, otherwise some instances of the old value ($\omega$) will remain in the loop.

4. Setup time is the time before the clock edge that the $d$-input must be stable with the current value ($\alpha$).

5. The setup time must be sufficient to saturate the storage loop with the current value ($\alpha$) and flush out all of the old values ($\omega$) before the storage loop is turned on.

6. Paths for this specific circuit:
   - Path to saturate storage loop = $d \rightarrow s2$
   - Path to turn on storage loop = $clk \rightarrow s2$

7. Equation for this specific circuit:
   \[ T_{SU} = delay(d \rightarrow s2) - delay(clk \rightarrow s2) = 6 - 2 = 4 \]

Setup is the ___________ that ____________ needs so that it can ____________ ____________ before ____________ ____________ ____________.
8.2.6 Setup Time Analysis

Setup Violation

Circuit is stable in load mode with \( \omega \)

t=1: \( \alpha \) propagates through AND gate for load path
\( \omega \) is on input to AND gate for storage loop
Clk propagates through inverter
Trouble: inconsistent values on load path and store path.
Old value (\( \omega \)) still in store path when store path is enabled.

t=0: \( \alpha \) propagates through inverter
Clk transitions from load to store

\( t=-1: D \) transitions from \( \omega \) to \( \alpha \)

\( t=2: \) old \( \omega \) propagates through AND

\( t=3: l_2 \) is set to 0,
because \( c_2 \) turns off AND gate
t=4: $\omega/\alpha$ from store path propagates to $q$

$t=5$: $\omega/\alpha$ from store path completes cycle

$t=5$: Illustrate instability with $\omega=0$, $\alpha=1$

setup with negative margin

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\omega$</th>
<th>$\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8.2.6 Setup Time Analysis
We now repeat the analysis of setup violation, but illustrate the minimum violation (input transitions from $\omega$ to $\alpha$ 3 time-units before the clock edge).

Circuit is stable in load mode with $\omega$

$t=-3$: D transitions from $\omega$ to $\alpha$

$t=-2$: $\alpha$ propagates through inverter

$t=-1$: $\alpha$ propagates through AND

$t=0$: Clk transitions from load to store

$t=1$: Clk propagates through inverter
Trouble: inconsistent values on load path and store path. Old value ($\omega$) still in store path when store path is enabled.

$t=2$: old $\omega$ propagates through AND

$t=3$: $l_2$ is set to 0, because $c_2$ turns off AND gate

$t=4$: $\omega/\alpha$ from store path propagates to $q$

$t=5$: $\omega/\alpha$ from store path completes cycle

$t=5$: Illustrate instability with $\omega=0$, $\alpha=1$

$t=5$: Illustrate instability with $\omega=0$, $\alpha=1$

<table>
<thead>
<tr>
<th>t</th>
<th>$d$</th>
<th>$\omega$</th>
<th>$\alpha$</th>
<th>$l_1$</th>
<th>$l_2$</th>
<th>$q_n$</th>
<th>$q$</th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>clk</th>
<th>$c_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

-3 -2 -1 0 1 2 3 4 5 6

setup with negative margin
8.2.7 Hold Time of a Multiplexer Latch

Hold Time Behaviour
1. When the latch is in store mode, there must be consistent values in the storage loop. Otherwise, the loop will be metastable.

2. As the circuit transitions from load mode to store mode, there must be consistent values at the point where the load and store paths join.

3. We must turn off the load path before the next value (β) affects the storage loop.

4. Hold time is the time after the clock edge that the d-input must be stable with the current value (α).

5. The hold time must be sufficient to turn off the load path before the next data value (β) is able to affect the internal circuitry such that it will affect storage loop.

6. Paths for this specific circuit:
   - Path to turn off load path = clk → 12
   - Path to affect internals = d → 12

7. Equation for this specific circuit:
   \[ T_{HO} = delay(clk \rightarrow 12) - delay(d \rightarrow 12) \]
   \[ = 2 - 1 \]
   \[ = 1 \]
Behaviour of a Bad Latch

Circuit is stable in load mode

t=0: Clk transitions from load to store
CHAPTER 8. TIMING ANALYSIS

Analysis of Bad Latch

1. When the latch is in store mode, there must be consistent values in the storage loop. Otherwise, the loop will be metastable.

2. As the circuit transitions from load mode to store mode, there must be consistent values at the point where the load and store paths join.

3. The current value ($\alpha$) must arrive at the join gate for the storage loop and load path before the constant “off” value from the load path arrives at the join gate.

4. Paths for this specific circuit:
   - Path from clk to store-enable to join =
   - Path from clk to load-enable to join =

5. Equation for this specific circuit:

8.2.9 Summary

1. Test if latch is correct
   (a) Find the storage loop
   (b) Find the load path
   (c) Check that load-mode and storage-mode are mutually exclusive
   (d) Find gates for load-enable, store-enable, and paths-join
   (e) Check that have even number of inversions on load path when in load mode
   (f) Check that have an even number of inversions in storage loop when in store mode
   (g) Check that path for clk to store-enable to join is faster than path for clk to load-enable to join

2. Determine if latch is active high or active low

3. Find clock-to-Q time: delay along path clk to load-enable to output

4. Find setup time:
   - delay(path for input to saturate storage loop) – delay(path to turn on storage loop)
   - delay(d to store-enable) – delay(clk to store-enable)

5. Find hold time:
   - delay(path for input to affect internals) – delay(path to turn off load path)
   - delay(clk to load-enable) – delay(d to load-enable)

8.3 Advanced Timing Analysis of Storage Elements

This is an advanced section. It is not covered in the course and will not be tested.

8.4 Critical Path

The critical path of a circuit is used to determine the maximum propagation delay of the circuit, which in turn constrains the minimum clock period.

Scenario:
- a purely combinational circuit
- at $t = 0$, one or more inputs change value
- record the time of the last value change (edge) on an output

The maximum of the times of the last edge is the maximum delay (sometimes, just “delay”) through the circuit.
Example of Max Delay

Each input may be 0, 1, $\top$, or $\bot$. For $n$ inputs, there are $4^n - 2^n$ possible input vectors.

8.4.1 Introduction to Critical and False Paths

**Definition** critical path: The slowest path on the chip between flops or flops and pins. The critical path limits the maximum clock speed.

**Definition** false path: A path along which an edge cannot travel from beginning to end.

Throughout our discussion of critical paths, we will use the delay values for gates shown in the table below.

<table>
<thead>
<tr>
<th>gate</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>2</td>
</tr>
<tr>
<td>AND</td>
<td>4</td>
</tr>
<tr>
<td>OR</td>
<td>4</td>
</tr>
<tr>
<td>XOR</td>
<td>6</td>
</tr>
</tbody>
</table>

8.4.1.1 Example of Critical Path in Full Adder

**Question:** Find the longest path through the full-adder circuit shown below.

**Question:** Does the excitation $c_i=1$, $a=1$, $b=0$ exercise the longest path?

**Alternative Excitation**

**Question:** Does the excitation $c_i=0$, $a=1$, $b=1$ exercise the critical path?
Exercising the Critical Path

Not all all transitions on the inputs will exercise the critical path. Using timing simulation to find the maximum delay of a circuit might underestimate the delay, because the inputs values that you simulate might not exercise the critical path.

8.4.1.2 Longest Path and Critical Path

The longest path through the circuit might not be the critical path, because the behaviour of the gates might prevent an edge from travelling along the path. Using the longest path to find the maximum delay of a circuit might overestimate the delay, because the longest path might be a false path.

Example False Path

Question: Determine whether the longest path in the circuit below is a false path

Question: How can we determine analytically that this is a false path?

Analytic Approach

Note: False paths are an advanced topic and are covered in section 8.5.

8.4.1.3 Criteria for Critical Path Algorithms

Let $T_r$ be the real (as measured by stimulating the circuit with all possible input vectors) maximum time of the last edge on an output.

Given an algorithm to calculate the critical path, let $T_a$ be the time of the last edge as calculated by the algorithm.

Three criteria for evaluating merits of a critical path algorithm:

1. Correctness: $T_a \geq T_r$: The delay given by the algorithm must be at least as long as the real maximum delay.

2. Optimality: The goal is to minimize $T_a - T_r$. The closer algorithm is to the real maximum delay, the more optimal the algorithm is.

3. Complexity: The goal is to minimize the computational complexity (i.e., runtime) of the algorithm.

Question: Is “longest path” a correct critical path algorithm?
8.4.2 Longest Path

8.4.2.1 Algorithm to Find Longest Path

The basic idea is to annotate each signal with the maximum delay from it to an output.

- Start at destination signals and traverse through fanin to source signals.
  - Destination signals have a delay of 0
  - At each gate, annotate the inputs by the delay through the gate plus the delay of the output.
  - When a signal fans out to multiple gates, annotate the output of the source (driving) gate with maximum delay of the destination signals.
- The primary input signal with the maximum delay is the start of the longest path. The delay annotation of this signal is the delay of the longest path.
- The longest path is found by working from the source signal to the destination signals, picking the fanout signal with the maximum delay at each step.

8.4.2.2 Longest Path Example

**Question:** Find the longest path through the circuit below.

![Circuit Diagram](image)

Variability:
- The delay through a gate can change over time
  - temperature
  - supply voltage
  - other effects
- The delay through two “identical” gates can be different
  - manufacturing variability
  - load

Example: measure the delay 1000 times for each of 1000 “identical” AND gates.

Timing Models

When you design a circuit, you do not know the precise delays through the physical gates that will be on the chips.

The manufacturer will discard all chips whose gates are too fast or too slow.

Manufacturers give min/max bounds on the delay for each gate in the cell library.

Critical path analysis with min/max delays is very complex.

Goal: do critical path analysis with just the max delay through a gate.

Problem: using the maximum delay through each gate might not cause the maximum delay in the circuit.
8.3 Monotone Speedup

Review: Critical Path Analysis

1. If we say that the delay along the longest path is the delay of the circuit, will this algorithm be correct with respect to monotone speedup?

8.5 False Paths

This is an advanced section. It is not covered in the course and will not be tested.
8.6 Analog Timing Model

Goal: define how to compute the delay of a gate or FPGA cell.

- section 8.6: precise differential equations; complex because no closed-form solutions for realistic circuits
- section 8.7: Elmore’s approximation to precise differential equations

Objectives
- How to define the delay through a gate or circuit. (section 8.6.1)
- How to model a circuit as an RC-network. (section 8.6.2)
- How to calculate delay of an RC-network. (section 8.6.3)

8.6.1 Defining Delay

Goal: define “the delay through a gate”.

Easy:

Reality:

1. The slope of the output \( V_b \) is dependent upon the slope of the input \( V_a \):

Defining Delay (cont’d)

2. The more gates that a gate drives (the larger the load) the slower the output voltage will rise.

Defining Delay (cont’d)

3. Because the output waveform is sloped, we must choose the voltage level for \( V_b \) at which we will measure the delay.

Definition Trip Points: A high or ‘1’ trip point is the voltage level where an upwards transition means the signal represents a ‘1’.

A low or ‘0’ trip point is the voltage level where a downwards transition means the signal represents a ‘0’.
Summary: Analog Delay Model

The standard approach to define the delay through a gate is:

- **Input waveform**: Step function
- **Load circuit**: 4 copies of the gate
- **Output trip points**: 0.65 Vdd for a 0 to 1 transition
  - 0.35 Vdd for a 1 to 0 transition.

The delay through a circuit is identical, except that the load is either a standard load such as 4 NAND gates, specified by the user, or the module in which the circuit is used.

8.6.2 Modeling Circuits for Timing

- Model the delay of wires, gates, and connections between wires (via, switch-box, or antifuse).
- Dominant factors in delay are resistance and capacitance
- Resistance and capacitance affected by different parameters.

### Resistance and Capacitance of Wires

**Wires**

- **Material**
  - Typically aluminum or copper
  - Copper has less resistance than aluminum
  - Aluminum is simpler to work with
  - Aluminum is used for long, fat wires, copper is used everywhere else.
  - For capacitance, material of di-electric (material surrounding wire) is also important
- **Cross section**
  - Larger cross section has lower resistance
  - Tall narrow wires require less area, but have higher capacitance
- **Length**: longer length has greater resistance and capacitance

### Components and Models

- **Physical chip**
- **Physical model**
- **RC network**
- **Gate**
- **Switchbox**
- **Wire**
8.6.2.1 Example: Two Buffers with Complex Wiring

Schematic

Physical chip (one of many possible layouts)

Physical model

RC network

8.6.2.2 Example: Two Buffers with Simple Wiring

Schematic

Physical chip

Physical model

RC network

8.6.3 Calculate Delay

Trim for timing analysis

- Even this simple example is too complex for our first simple example
- Simplify the simple example by simply assuming that the capacitance of the wire is much less than the capacitance of the gate ($C_{W1} \ll C_{G2}$).

Simplify

- Another simplification: collapse the line of resistors into a single resistor (this just makes the algebra simpler, it does not affect the precision of the analysis).

Two Bufs: Derivation of Delay Equation

Goal: calculate delay from $V_{G1}$ to $V_{G2}$

Strategy:
1. Derive equation for $\ldots$ in terms of $\ldots$.
2. Solve for time for $\ldots$ to reach $\ldots$, assuming that $\ldots$.
Two Bufs Derivation (Cont’d)

Goal: calculate delay from $V_{G1}$ to $V_{G2}$

Derivation of equation for $V_{G2}$:

$$V_{G2}(t) = V_{G1}(t) - \text{voltage drop from } V_{G1} \text{ to } V_{G2}$$

$$= V_{G1}(t) - I(t)R$$

Equation for the current through the capacitor

$$I(t) = C\frac{dV_{G2}(t)}{dt}$$

$$V_{G2}(t) = V_{G1}(t) - \left(C\frac{dV_{G2}(t)}{dt}\right)R$$

$$= V_{G1}(t) - RC\frac{dV_{G2}(t)}{dt}$$

Delay Analysis

With initial condition $V_{G2}(0) = 0$ and forcing function $V_{G1}(t) = \text{VDD-step-function}$, the closed form solution is:

$$V_{G2}(t) = \text{VDD} - \text{VDD}e^{-t/RC}$$

Find delay for $V_{G2}$ to reach 0.65VDD

$$0.65 \text{VDD} = \text{VDD} - \text{VDD}e^{-t/RC}$$

$$0.35 = e^{-t/RC}$$

$$\ln 0.35 = \ln(e^{-t/RC})$$

$$-1.05 = -t/RC$$

$$-1.0 \approx -t/RC$$

$$t = RC$$

With a step function input, the delay for $V_{G2}$ to reach 0.65VDD is:

$$(R_{S1} + R_{W1} + R_{S2})C_{G2}$$

which is commonly known as the $RC$ time constant.
8.6.4 Ex: Two Bufs with Both Caps

To calculate voltages and currents correctly, we number the nodes of the circuit consistently.

- The voltage source and the top of each capacitor is a node.
- We number the nodes, capacitors, and resistors.
- Resistors are numbered according to the capacitor to their right.
- Multiple resistors in series without an intervening capacitor are lumped into a single resistor.
- \( V_i \) is the voltage at node \( i \).
- \( I_{ri} \) is the current flowing through \( R_i \)
- \( I_{ci} \) is the current flowing into/out-of \( C_i \)

![Original RC network](image1)

![With node numbers](image2)

![RC network](image3)

With node numbers

![Waveforms](image4)

**Summary: Precise Equations**

Measure delay from source node (node 0) to a particular destination (node \( i \))

- Initial condition: node \( i \) is at GND: \( V_i(0) = 0 \)
- Voltage at source node is step function to VDD
- Measure time for \( V_i \) to reach 0.65VDD
- \( V_i(t) = V_0(t) - \) voltage drop across resistors on path from 0 to \( i \)
- Voltage drop across resistor: \( V = I_R R \)
- Current through resistor is sum of currents through downstream capacitors

Result is a partial differential equation without a closed-form solution.

To calculate \( V_i(t) \) precisely:

- Write one equation for each node
- Have set of \( n \) partial differential equations for \( n \) variables
- Use numerical methods to calculate \( V_i(t) \)
- Note: to calculate the voltage at one node, need to calculate the voltage at every node.
8.7 Elmore Delay Model

8.7.1 Elmore Delay as an Approximation

To avoid solving a set of partial differential equations, Elmore proposed a simple, but effective approximation.

\[
V_2(t) = V_0(t) - R_1C_1 \frac{dV_1(t)}{dt} - (R_1 + R_2)C_2 \frac{dV_2(t)}{dt}
\]

Template with a closed-form solution:

\[
= V_0(t) - k \frac{dV(t)}{dt}
\]

Elmore’s approximation:

\[
\frac{dV_1(t)}{dt} = \frac{dV_2(t)}{dt}
\]

\[
= V_0(t) - R_1C_1 \frac{dV_2(t)}{dt} - (R_1 + R_2)C_2 \frac{dV_2(t)}{dt}
\]

\[
= V_0(t) - (R_1C_1 + (R_1 + R_2)C_2) \frac{dV_2(t)}{dt}
\]

With initial condition \( V_2(0) = 0 \)

and forcing function \( V_0(t) = VDD \cdot \text{step-function} \):

\[
= VDD - VDDe^{-t/(R_1C_1 + (R_1 + R_2)C_2)}
\]

Time for \( V_2 \) to go from GND to 0.65VDD

\[
t = R_1C_1 + (R_1 + R_2)C_2
\]

8.7.2 A More Complicated Example

**Definition** path: The path from the source node to a node \( i \) is the set of all resistors between the source and \( i \).

**Example:** \( \text{path}(2) = \)

**Definition** down: The set of capacitors downstream from a resistor is the set of all capacitors where current would flow through the resistor to charge the capacitor. You can think of this as the set of capacitors that are between the node and ground.

**Example:** \( \text{down}(R_2) = \)  

**Example:** \( \text{down}(R_4) = \)
Definition: Elmore time constant

Simple formula:

\[ T_{Di} = \sum_{r \in \text{path}(i)} R_r \sum_{c \in \text{down}(r)} C_c \]

The conventional formula is more complex syntactically, but equivalent mathematically:

\[ T_{Di} = \sum_{k \in \text{Nodes}} C_k \sum_{r \in (\text{path}(i) \cap \text{path}(k))} R_r \]

The equivalence of the two formulations can be shown by observing that if \( c \) is downstream from \( r \), then \( r \) is on the path to \( c \):

\[ c \in \text{down}(r) \iff r \in \text{path}(c) \]

### Summary of Analog and Elmore Delay

1. The voltage drop from the source to a node is:

2. The voltage drop across a resistor is:

3. The current through a capacitor is:

4. The Elmore approximation is:

### Calculate Elmore Delay

**Question:** Calculate the Elmore delay to node 2.

![Diagram of resistors and capacitors](image)

**Question:** You must increase the size of one resistor while minimizing the increase of the delay to node 2. Which resistor should you increase?

**Question:** Which resistor should you decrease to maximally reduce the delay to node 2?
This is an advanced section. It is not covered in the course and will not be tested.

Chapter 9

Power Analysis and Power-Aware Design

9.1 Overview

9.1.1 Importance of Power and Energy

- Laptops, PDA, cell-phones, etc — obvious!
- For microprocessors in personal computers, every watt above 40W adds $1 to manufacturing cost
- Approx 25% of operating expense of server farm goes to energy bills
- (Dis)Comfort of Unix labs in E2
- Sandia Labs had to build a special sub-station when they took delivery of Teraflops massively parallel supercomputer (over 9000 Pentium Pros)
- High-speed microprocessors today can run so hot that they will damage themselves — Athlon reliability problems, Pentium 4 processor thermal throttling
- In 2000, information technology consumed 8% of total power in US.
- Future power viruses: cell phone viruses cause cell phone to run in full power mode and consume battery very quickly; PC viruses that cause CPU to meltdown batteries
9.1.2 Power vs. Energy

Most people talk about “power” reduction, but sometimes they mean “power” and sometimes “energy.”

- Energy is stored and transmitted.
  - Stored (battery or capacitor), or transferred.
  - Consumed to perform an operation
- Goals of minimization:
  - Reduce energy costs
  - Increase battery life

- Power is the rate of energy consumption or transmission (Energy/time)
  - Goals of minimization:
    - Reduce cost of heat removal equipment
    - Passive devices receive their energy continuously over the air.
    - Reduce size and cost.
    - Increase operational distance from energy source.

### Power vs. Energy

<table>
<thead>
<tr>
<th>Type</th>
<th>Units</th>
<th>Equivalent Types</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>Joules</td>
<td>Work</td>
<td>( = \text{Volts} \times \text{Coulombs} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( = \frac{1}{2} \times C \times \text{Volts}^2 )</td>
</tr>
<tr>
<td>Power</td>
<td>Watts</td>
<td>Energy / Time</td>
<td>( = \frac{\text{Joules}}{\text{sec}} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( = \frac{\text{Volts} \times \text{Coulombs}}{\text{sec}} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( = \text{Volts} \times \text{Current} )</td>
</tr>
</tbody>
</table>

Memory refresh:

Capacitors store energy.

- Capacitance = \( \frac{\text{Coulombs}}{\text{Volts}} \)
- Current = \( \frac{\text{Coulombs}}{\text{sec}} \)

9.1.3 Batteries, Power and Energy

9.1.3.1 Do Batteries Store Energy or Power?

\[
\text{Energy} = \text{Volts} \times \text{Coulombs}
\]

\[
\text{Power} = \frac{\text{Energy}}{\text{Time}}
\]

Batteries rated in Amp-hours at a voltage.

\[
\text{battery} = \text{Amps} \times \text{Seconds} \times \text{Volts}
\]

\[
= \frac{\text{Coulombs}}{\text{sec}} \times \text{Seconds} \times \text{Volts}
\]

\[
= \text{Coulombs} \times \text{Volts}
\]

\[
= \text{Energy}
\]

Batteries store energy.

9.1.3.2 Battery Life and Efficiency

To extend battery life, we want to increase the amount of work done and/or decrease energy consumed.

Work and energy are same units, therefore to extend battery life, we truly want to improve efficiency.

“Power efficiency” of microprocessors normally measured in MIPS/Watt. Is this a real measure of efficiency?

\[
\text{MIPS/Watts} = \frac{\text{millions of instructions}}{\text{Seconds}} \times \frac{\text{Seconds}}{\text{Energy}}
\]

\[
= \frac{\text{millions of instructions}}{\text{Energy}}
\]

Both instructions executed and energy are measures of work, so MIPS/Watt is a measure of efficiency.

**Question:** What is the weakness of this analysis?
Question: Running a VHDL simulation requires executing an average of 1 million instructions per simulation step. My computer runs at 1.5GHz, has a CPI of 0.67, and burns 18W of power. My battery is rated at 11V and 5.6Ah. Assuming all of my computer’s clock cycles go towards running VHDL simulations, how many simulation steps can I run on one battery charge?

Question: In low-power mode, the clock speed is reduced to 1.0GHz and the power consumption is 10W. In low-power mode, how much longer can I run the computer on one battery charge?

9.2 Power Equations

\[ \text{Power} = \frac{\text{SwitchPower} + \text{ShortPower}}{\text{DynamicPower}} + \frac{\text{LeakagePower}}{\text{StaticPower}} \]

- **Dynamic Power**: dependent upon clock speed
- **Switching Power**: useful — charges up transistors
- **Short Circuit Power**: not useful — both N and P transistors are on
- **Static Power**: independent of clock speed
- **Leakage Power**: not useful — leaks around transistor
Dynamic Power

Dynamic power is proportional to how often signals change their value (switch).

- Roughly 20% of signals switch during a clock cycle.
- Need to take glitches into account when calculating activity factor. Glitches increase the activity factor.
- Equations for dynamic power contain clock speed and activity factor.

Activity factor = \[\frac{\text{number of value changes}}{\text{number of signal} \times \text{number of clock cycles}}\]

9.2.1 Switching Power

- Charging a capacitor
- Discharging a capacitor

\[\text{energy to (dis)charge capacitor} = \frac{1}{2} \times \text{CapLoad} \times \text{VoltSup}^2\]

9.2.2 Short-Circuit Power

\[\text{PwrShort} = \text{ActFact} \times \text{ClockSpeed} \times \text{TimeShort} \times I_{\text{Short}} \times \text{VoltSup}\]

9.2.3 Leakage Power

\[\text{PwrLk} = I_{\text{Short}} \times \text{VoltSup}
\]

\[I_{\text{Leak}} \propto e^{-\frac{q \times \text{VoltThresh}}{k \times T}}\]
9.3 Overview of Power Reduction Techniques

We can divide power reduction techniques into two classes: analog and digital.

### Analog Parameters

- **Capacitance** for example: Silicon on Insulator (SOI) and high-K dielectrics
- **Resistance** for example: copper wires rather than aluminum
- **Voltage** low-voltage circuits

### Digital Parameters

- **Capacitance** (number of gates)
- **Activity Factor**
- **Clock Frequency**

### Analog Techniques

Power reduction techniques at the analog level.

- **Dual-VDD** Two different supply voltages: high voltage for performance-critical portions of design, low voltage for remainder of circuit. Alternatively, can vary voltage over time: high voltage when running performance-critical software and low voltage when running software that is less sensitive to performance.
- **Dual-Vt** Two different threshold voltages: transistors with low threshold voltage for performance-critical portions of design (can switch more quickly, but more leakage power), transistors with high threshold voltage for remainder of circuit (switches more slowly, but reduces leakage power).
- **Exotic Circuits** Special flops, latches, and combinational circuitry that run at a high frequency while minimizing power
- **Adiabatic Circuits** Special circuitry that consumes power on \(0 \rightarrow 1\) transitions, but not \(1 \rightarrow 0\) transitions. These sacrifice performance for reduced power.
- **Clock Trees** Up to 30% of total power can be consumed in clock generation and clock tree
Digital Techniques

For power-reduction techniques at the digital level:

- **Multiple clocks** Put a high speed clock in performance-critical parts of design and a low speed clock for remainder of circuit.
- **Clock gating** Turn off clock to portions of a chip when it's not being used.
- **Data encoding** Gray coding vs one-hot vs fully encoded vs ...
- **Glitch reduction** Adjust circuit delays or add redundant circuitry to reduce or eliminate glitches.
- **Asynchronous circuits** Get rid of clocks altogether....

### 9.4 Voltage, Power, and Delay

If our goal is to reduce power, the most promising approach is to reduce the supply voltage, because, from:

\[
\text{Power} = (\text{ActFact} \times \text{ClockSpeed} \times \frac{1}{2} \text{CapLoad} \times \text{VoltSup}^2) + (\text{ActFact} \times \text{ClockSpeed} \times \text{TimeShort} \times \text{IShort} \times \text{VoltSup}) + (\text{ILeak} \times \text{VoltSup})
\]

we observe:

\[
\text{Power} \propto \text{VoltSup}^2
\]

### Reducing Difference Between Supply and Threshold Voltage

When supply voltage decreases:

- supply current decreases
- time to charge capacitance load increases
- load delay of circuit increases

More precisely, the load delay depends on both the supply voltage and the difference between the supply and threshold voltages.

\[
\text{LoadDelay} \propto \frac{\text{VoltSup}}{(\text{VoltSup} - \text{VoltThresh})^2}
\]

### Effect of Decreasing Supply Voltage on Delay

Timing with original VDD

Timing with reduced VDD
Effect of Decreasing Supply Voltage on Delay

**Question:** If the delay through a circuit is 20 ns, the supply voltage is 2.8 V, and the threshold voltage is 0.7 V, calculate the delay if the supply voltage is dropped to 2.2 V.

Sacrifice or Optimization?

- Decreasing the supply voltage increases the delay through the circuit
- Increasing the clock period allows us to:

Reducing Threshold Voltage Increases Leakage Current

If we reduce the supply voltage, we want to also reduce the threshold voltage, so that we do not increase the delay through the circuit. However, as threshold voltage drops, leakage current increases:

$$I_{\text{leak}} \propto e^{\frac{-q \times \text{VoltThresh}}{k \times T}}$$

And increasing the leakage current increases the power:

$$\text{Power} \propto I_{\text{leak}}$$

So, need to strike a balance between reducing VoltSup (which has a quadratic affect on reducing power), and increasing Ileak, which has a linear affect on increasing power.

Clock Speed and Power Consumption

**Question:** In the question on high-performance / low-power for VHDL simulation earlier in the chapter, the laptop was able to execute 27.7 million simulation steps in high power mode and 33.2 million simulation steps in low-power mode. What percentage of the additional simulation steps was due to reducing the clock speed and what percentage was due to reducing the supply voltage?
Summary of Supply Voltage and Clock Speed

<table>
<thead>
<tr>
<th>Pixel (example of a unit of work)</th>
<th>Performance</th>
<th>Power</th>
<th>Uptime</th>
<th>Battery Energy</th>
</tr>
</thead>
</table>

Clock Circuit

- **VDD** speed delay Slack Performance Power Energy Uptime Work
- **V** cyc/sec sec sec pix/sec J/sec sec/bat pix/bat
- = ↑
- ↑ =
- ↑ ↑
- ↓ ↓

9.5 Data Encoding for Power Reduction

9.5.1 How Data Encoding Can Reduce Power

Data encoding is a technique that chooses data values so that normal execution will have a low activity factor.

The most common example is “Gray coding” where exactly one bit changes value each clock cycle when counting.

Question: For an eight-bit counter, how much more power will a binary counter consume than a Gray-code counter?

8-bit Counter

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Gray</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 9. POWER

Random Data

Question: For completely random eight-bit data, how much more power will a binary circuit consume than a Gray-code circuit?

9.5.2 Example Problem: Sixteen Pulser

9.5.2.1 Problem Statement

Your task is to do the power analysis for a circuit that should send out a one-clock-cycle pulse on the done signal once every 16 clock cycles. (That is, done is ‘0’ for 15 clock cycles, then ‘1’ for one cycle, then repeat with 15 cycles of ‘0’ followed by a ‘1’, etc.)

![Circuit Diagram]

Required behaviour

You have been asked to consider three different types of counters: a binary counter, a Gray-code counter, and a one-hot counter. (The table below shows the values from 0 to 15 for the different encodings.)

Question: What is the relative amount of power consumption for the different options?

9.5.2.2 Additional Information

Your implementation technology is an FPGA where each cell has a programable combinational circuit and a flip-flop. The combinational circuit has 4 inputs and 1 output. The capacitive load of the combinational circuit is twice that of the flip-flop.

1. You may neglect power associated with clocks.
2. You may assume that all counters:
   (a) are implemented on the same fabrication process
   (b) run at the same clock speed
   (c) have negligible leakage and short-circuit currents

Data Encoding

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Gray</th>
<th>One-Hot</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000000000000001</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0000000000000010</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
<td>0000000000000100</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>0000000000001000</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
<td>0000000000100000</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0111</td>
<td>0000000010000000</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0101</td>
<td>0000000010000001</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0100</td>
<td>0000000100000000</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1100</td>
<td>0000001000000000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1101</td>
<td>0000001000000001</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1111</td>
<td>0000010000000000</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td>0000100000000000</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>1010</td>
<td>0010000000000000</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>1011</td>
<td>0100000000000000</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>1001</td>
<td>0100000000000000</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
<td>1000000000000000</td>
<td>1111</td>
</tr>
</tbody>
</table>
### Sketch the Circuitry

Name the output “done” and the count digits “d( )”.

#### Capacitance

<table>
<thead>
<tr>
<th></th>
<th>cap</th>
<th>number</th>
<th>subtotal cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray</td>
<td>d ( )</td>
<td>PLAs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-Hot</td>
<td>d ( )</td>
<td>PLAs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>d ( )</td>
<td>PLAs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Activity Factors

**Gray Coding Activity Factor**

![Gray Coding Activity Factor](attachment:image1.png)

**One-Hot Activity Factor**

![One-Hot Activity Factor](attachment:image2.png)
9.6 Clock Gating

The basic idea of clock gating is to reduce power by turning off the clock when a circuit isn’t needed. This reduces the activity factor.

Related to clock gating:

**Chip enable** Use the chip-enable on a flip flop to hold the output constant when not needed.

**Operand gating** Use AND gates and an enable signal to set data (operand) values to zero when a datapath circuit is not needed.

**Power gating** Turn off supply voltage to part of the chip.

### Examples of Clock Gating

<table>
<thead>
<tr>
<th>Condition</th>
<th>Circuitry turned off</th>
</tr>
</thead>
<tbody>
<tr>
<td>O/S in standby mode</td>
<td>Everything except “core” state (PC, registers, caches, etc)</td>
</tr>
<tr>
<td>No floating point instructions for $k$ clock cycles</td>
<td>floating point circuitry</td>
</tr>
<tr>
<td>Instruction cache miss</td>
<td>Instruction decode circuitry</td>
</tr>
<tr>
<td>No instruction in pipe stage $i$</td>
<td>Pipe stage $i$</td>
</tr>
</tbody>
</table>

---

## Binary Coding Activity Factor

<table>
<thead>
<tr>
<th>clk</th>
<th>d(0)</th>
<th>d(1)</th>
<th>d(2)</th>
<th>d(3)</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1/16</td>
<td>8/16</td>
<td>4/16</td>
<td>2/16</td>
<td>2/16</td>
</tr>
</tbody>
</table>

Binary coding

---

## Putting it all Together

<table>
<thead>
<tr>
<th>Subtotal cap</th>
<th>act fact</th>
<th>power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray</td>
<td>d()</td>
<td>PLAs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td>Flops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>1-Hot</td>
<td>d()</td>
<td>PLAs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td>Flops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Binary</td>
<td>d()</td>
<td>PLAs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flops</td>
</tr>
<tr>
<td>done</td>
<td>PLAs</td>
<td>Flops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
</tbody>
</table>
9.6.2 Implementing Clock Gating

Clock gating is implemented by adding a component that disables the clock when the circuit isn’t needed.

Without clock gating

With clock gating

9.6.3 Design Process

This section reserved for your reading pleasure

9.6.4 Effectiveness of Clock Gating

- **PctClk** = Percentage of clock cycles that clock toggles.
- **PctBusy** = Percentage of clock cycles when busy doing useful work.
- **A** = Activity factor without clock gating
- **A’** = Activity factor with clock gating
- **A’_{\text{min}}** = Activity factor if clock is on only when busy
- **Eff** = Effectiveness of clock gating

\[
\text{Eff} = 0 \% \quad \Rightarrow \quad A' = A'_{\text{min}}
\]

\[
\text{Eff} = 100 \% \quad \Rightarrow \quad A' = A
\]

Effectiveness measures the percentage of clock cycles when the circuit is idle (contains only bubbles) that the clock is turned off.

Effectiveness (Cont’d)

\[
\text{Eff} = \frac{\text{PctBusy}}{\text{PctClk}}
\]

\[
PctClk =
\]

\[
A' =
\]
Clock Gating Effectiveness Questions

Question: What is the effectiveness if the clock toggles only when the circuit contains a parcel?

Question: What is the effectiveness of a clock that always toggles?

Question: What does it mean for a clock gating scheme to be 75% effective?

Question: What happens if PctClk < PctBusy?

9.6.5 Example: Reduced Activity Factor with Clock Gating

Question: How much power will be saved in the following clock-gating scheme?

- 70% of the time the main circuit contains at least one parcel
- clock gating circuit is 90% effective
- clock gating circuit has 10% of the area of the main circuit
- clock gating circuit has same activity factor as the main circuit
- neglect short-circuiting and leakage power
9.6.6 Calculating PctBusy

9.6.6.1 Valid Bits and Busy

Use valid bits to determine when a circuit is busy.

Microscopic Analysis

Which clock edges are needed?

LenBusy when Tput < 1
9.6.6.3 From LenBusy to PctBusy

Find the core of a repeating pattern of parcels and bubbles.

\[ \text{LenCore} = \]
\[ \text{LenBusy} = \]
\[ \text{PctBusy} = \]

**Question:** What happens if Lat > NumBubbles?

9.6.7 Example: Pipelined Circuit with Clock-Gating

Design a “clock enable state machine” for the pipelined component described below.

- area of pipelined component = 100
- latency varies from 5 to 10 clock cycles, uniform distribution of latencies
- contains a maximum of 6 parcels
- 60% of clock cycles have a parcel on the inputs
- average length of continuous sequence of valid parcels is 80
- area of clock-enable state machine = 13
- use input and output valid bits for wakeup
- leakage current is negligible
- short-circuit current is negligible

**Behavioural Analysis**

**Question:** Without further detailed analysis, can we determine which design is the better option?
Question: Which design option has lower power and how much lower is it?

9.6.8 Clock Gating in ASICs

A register with a chip-enable can be synthesized into a register with a gated clock.

process (clk) begin
  if rising_edge ( clk ) and en = '1' then
    a <= i_a;
    b <= i_b;
  end if;
end process;

Synthesis tools have commands and flags that cause this type of code to be synthesized into a circuit with a gated clock.

FPGAs do not support clock gating — use alternatives in section 9.6.9.
9.6.9 Alternatives to Clock Gating

9.6.9.1 Use Chip Enables

Same coding as with clock gating, just do not enable clock-gating in the synthesis tool.

This technique is used with both ASICs and FPGAs.

Advantages

Disadvantages

Chapter 10

Review

This chapter lists the major topics of the term. The “Topics List” section for each major area is meant to be relatively complete.

10.1 Overview of the Term

- The purely digital world
  - VHDL
  - design and optimization methods
  - performance analysis

- Analog effects in the digital world
  - timing analysis
  - power
10.2 VHDL

10.2.1 VHDL Topics
- simple syntax and semantics — things that you should know simply by having done the labs and project
- behavioural semantics of VHDL
- synthesis semantics of VHDL
- VHDL code as legal, synthesizable, and good-practice

10.2.2 VHDL Example Problems
- identify whether a particular signal will be the output of combinational circuitry or a flop
- identify whether a particular process is combinational or clocked
- legal, synthesizable, and good code
- perform delta-cycle simulation of VHDL
- perform RTL simulation of VHDL
- identify whether two VHDL fragments have same behaviour
- analyze area, approximate clock period, latency, throughput, etc of VHDL code

10.3 RTL Design Techniques

10.3.1 Design Topics
- coding guidelines
- generic FPGA hardware
- area estimation
- finite state machines
  - implicit
  - explicit-current
  - explicit-current+next
- from algorithm to hardware
  - dependency graph
  - dataflow diagram
- scheduling
- allocation
- hardware block diagram
- state machine
- memory dependencies
- memory arrays and dataflow diagrams
- Pipelining
- Retiming
- Area and performance optimizations

10.3.2 Design Example Problems
- estimate area to implement a circuit in an FPGA
- calculate resource usage for a dataflow diagram
- calculate performance data for a dataflow diagram
- given an algorithm, design a dataflow diagram
- given a dataflow diagram, draw a control table and do resource allocation
- optimize a dataflow diagram to improve performance or reduce area
- analyze and compare the functionality, area, and performance of
  - VHDL code
  - pseudocode
  - state machine
  - dataflow diagram
  - waveforms
  - schematics
- use retiming to improve clock speed or area
- use retiming to determine if two circuits have the same behaviour
10.4 Performance Analysis and Optimization

10.4.1 Performance Topics
- time to execute a program
- definition of performance
- speedup
- n% bigger, smaller
- calculating performance of different tasks and of average task
- changing frequency of task and overall performance
- choosing which task to optimize to best improve overall performance
- performance increase over time
- design tradeoffs (CPI vs NumInsts vs ClockSpeed vs time-to-market)
- Clock speed vs. performance
- Optimality — performance / area tradeoffs

10.4.2 Performance Example Problems
- calculate tradeoffs between performance, area, schedule, and power
- evaluate performance criteria

10.5 Timing Analysis

10.5.1 Timing Topics
- circuit parameters that affect delay
  - clock period
  - clock skew
  - clock jitter
  - propagation delay
  - load delay
- setup time
- hold time
- clock-to-Q time
- timing analysis of latch
- concepts of critical path vs false path, timing models, monotonic speedup
- elmore timing model

10.5.2 Timing Example Problems
- timing parameters for minimum clock period
- timing parameters for hold constraint
- determine if a latch will work correctly
- compute timing parameters of a latch
- identify timing violation, suggest remedy
- find the longest path
- test if an excitation excites a particular path
- compute the Elmore delay constant
- use concepts of Elmore delay to compare delay of two circuits
- compare accuracy of different timing models
- suggest design change to increase clock speed
10.6 Power

10.6.1 Power Topics

- power vs energy
- equations for power
  - dynamic power
  - static power
  - switching power
  - short circuit power
  - leakage power
- activity factor
- leakage current
- threshold voltage
- supply voltage
- analog power reduction techniques
- rtl power reduction techniques
- clock gating

10.6.2 Power Example Problems

- predict effect of new fabrication process (supply voltage, threshold voltage, capacitance, circuit delay) on power
- predict effect of environment change (temp, supply voltage, etc) on power consumption
- predict effect of design change on power consumption (capacitance, activity factor, clock speed)
- design clock gating scheme for a circuit, predict effect on power consumption
- assess validity of various power- or energy-consumption metrics

\[
\begin{align*}
P &= \frac{1}{2}(A \times C \times V^2 \times F) + (\tau \times A \times V \times I_{Sh} \times F) + (V \times I_L) \\
T &= \frac{I_{s} \times C}{F} \\
F &\propto \frac{(V - V_t)^2}{V} \\
P &= V \times I \\
P &= \frac{W}{T} \\
I_L &\propto \frac{-q \times V_t}{e^{k \times T}}
\end{align*}
\]
\[ S = \frac{T_1}{T_2} \]

\[ M = \frac{F/10^6}{\sum_{i=0}^{n} P_i \times C_i} \]

\[ A' = (1 - E(1 - P_v))A \]

\[ q = 1.60218 \times 10^{-19} \text{C} \]

\[ k = 1.38066 \times 10^{-23} \text{J/K} \]

\[ \log_b y = \frac{\log y}{\log x} \]

\[ (x^y)^z = x^{yz} \]

\[ (x^y)(x^z) = x^{y+z} \]

\[ a = b^c \quad \text{is equivalent to}: \]

\[ a^{1/c} = b \]