Danmarks
Tekniske
Universitet

DTU

Assignment 1 Greatest Common Divisor Report

02203: Design of Digital Systems
Fall 2025

Group 14

Theodoros Pontzouktzidis - s250239
William Martin SĂ¸rensen - s243895
Aske Mop Rasmussen - s249370
Vasileios Tsiapas - s252859

We confirm that this report contains our own independent and original work. Unless otherwise explained in the preface of the report, all group members have contributed equally to all parts of the work.

Contents

1 Introduction

This assignment focuses on applying a systematic, top-down design flow for digital systems using SystemVerilog and the Xilinx Vivado toolchain, targeting an Artix-7 FPGA board (Nexys 4 DDR / Nexys A7). As a case study, we implement Euclid's algorithm for computing the greatest common divisor (GCD) of two positive integers.

2 Task 0

a)

Before starting, we carefully read through the entire assignment document to get an overview of the workflow, tasks and downloaded and skimmed the provided SystemVerilog files.

b)

Assuming the datapath implementation shown in Figure 2, and that the available technology primitives on the FPGA are D flip-flops and 6-input LUTs, with 16-bit unsigned registers A and B as well as an output register C, we can estimate the resources as follows:

- **Registers:**
  Each register (A, B) is 16 bits, requiring 16 D flip-flops each. Thus, we need 32 flip-flops in total. Additionally, the FSM state register will require 2 more flip-flops (4 states) **Total 34 flip-flops**.
- **ALU:**
  The ALU performs subtraction ($A - B$ and $B - A$) and pass-through (A, B). It can be analyzed on a per-bit basis:
  - A **1-bit full adder (FA)** is used for subtraction, since subtraction can be expressed as addition with two's complement. A FA computes:

    $$\text{Sum} = A \oplus B \oplus C_{in}, \qquad C_{out} = \text{majority}(A, B, C_{in}).$$

    Each output bit is a Boolean function that maps naturally to LUTs. This typically requires about 2 LUTs per bit, giving 32 LUTs for 16 bits are needed.
  - A **1-bit 2-to-1 multiplexer** is needed to select between a pass-through value (A or B) and the ALU subtraction result. For a 16-bit datapath, this corresponds to 16 LUTs.
  - **Control and flag logic:** The zero flag (Z) is computed as the NOR of all 16 output bits, requiring a small reduction tree (about 3–5 LUTs). The negative flag (N) is simply the MSB of the result, so no additional LUTs are required.

Total 51–53 LUTs

- **Multiplexers / Control Logic:**
  To select between operands and ALU outputs, we need 16-bit 2-to-1 multiplexers. This adds approximately 16 LUTs. Additional logic for the handshake protocol and flag generation (Z and N) contributes around 5–10 LUTs. **Total 21–26 LUTs**

Estimated totals:

- D-Flip-flops: 34
- LUTs: 72–79

This high-level estimate provides a reasonable expectation of the hardware cost, though the exact numbers will depend on synthesis optimizations performed by the toolchain.

3 Task 1

a)

After compiling the files and simulating the complete system (gcd_tb) we can see the waveforms. As we can see in figure 1 the simulation completed successfully with all the C results being computed correctly.

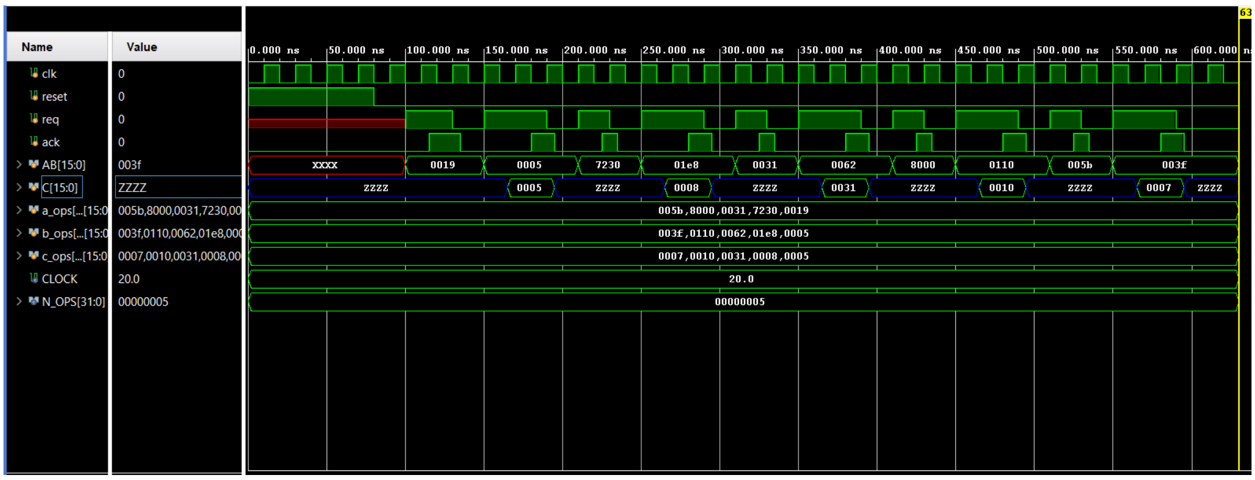


Figure 1: Simulation result of task 1.

4 Task 2

a)

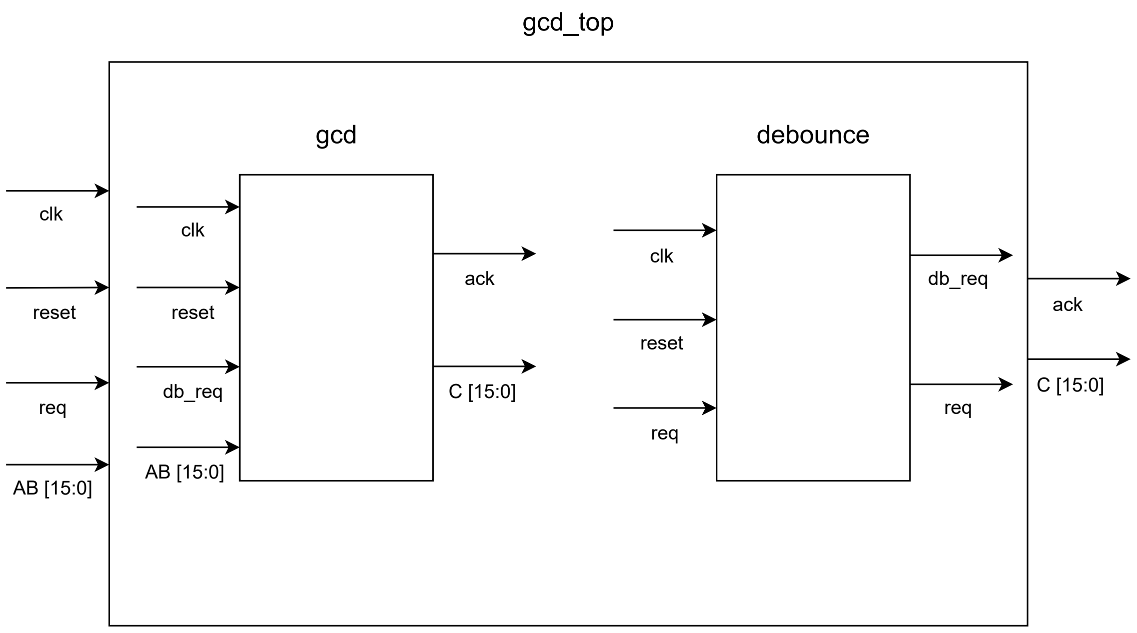


Figure 2: block diagram of the GCD system.

b)

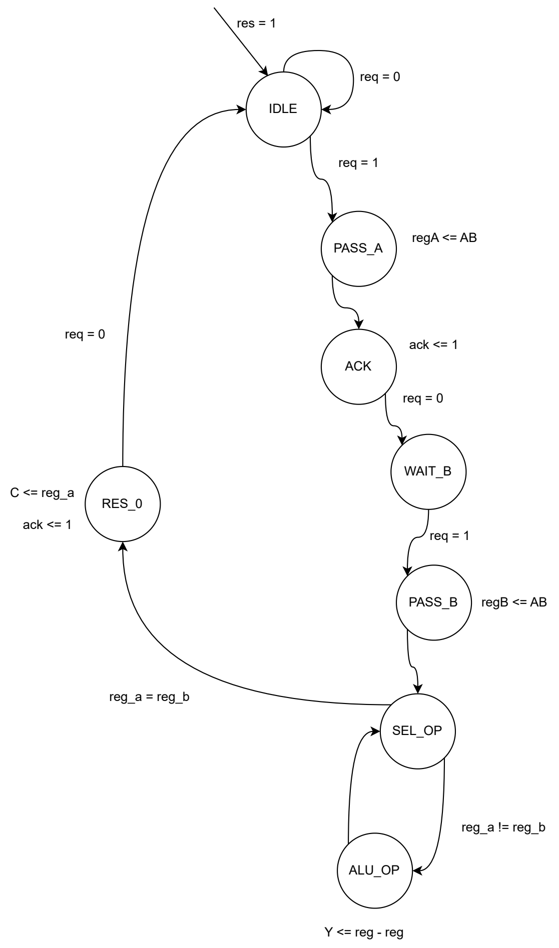


Figure 3: state diagram for the controller.

The finite state machine (FSM) [1] shown in Figure 3 controls the calculation of the greatest common divisor (GCD) of two numbers. It begins in the `IDLE` state, where it waits for a request signal (`req=1`). Once a request is received, the machine moves to the `PASS_A` and then to `PASS_B` state to load the input operands into registers(`regA` and `regB`).

If the two registers are equal, the FSM transitions directly to the `RES_0` state, where the result ready. Otherwise, the machine enters the `ALU_OP` state, where subtraction operations are performed (`Y` $\leftarrow$ `reg - reg`) to reduce the larger operand until both registers match.

When the GCD is found, the FSM transitions to the `RES_0` state where the acknowledge signal (`ack=1`) is asserted to indicate completion. Once the request signal is deasserted (`req=0`), the controller returns to the `IDLE` state, ready for the next computation.

c)

The new architecture body was completed, and it follows the requirements as described in the task description. The result can be found in the "task2/gcd.sv" file.

d)

In this task, we created a new Vivado project and simulated our GCD design using the provided gcd_tb testbench from Task 1. The operation begins in the IDLE state, where the circuit waits for the start signal and valid operands. Once triggered, the operands are loaded into internal registers during the LOAD state. The design then transitions into the comparison and subtraction loop, where the two operands are repeatedly compared, and the larger operand is reduced by subtraction until both operands converge to the same value. This iterative process continues until the GCD is obtained. When the result is ready, the FSM enters the RES_0 state and the final GCD is available at both of the registers. The simulation waveforms confirm that the FSM successfully transitioned through all states and the output matched the expected GCD values for the operand pairs tested.

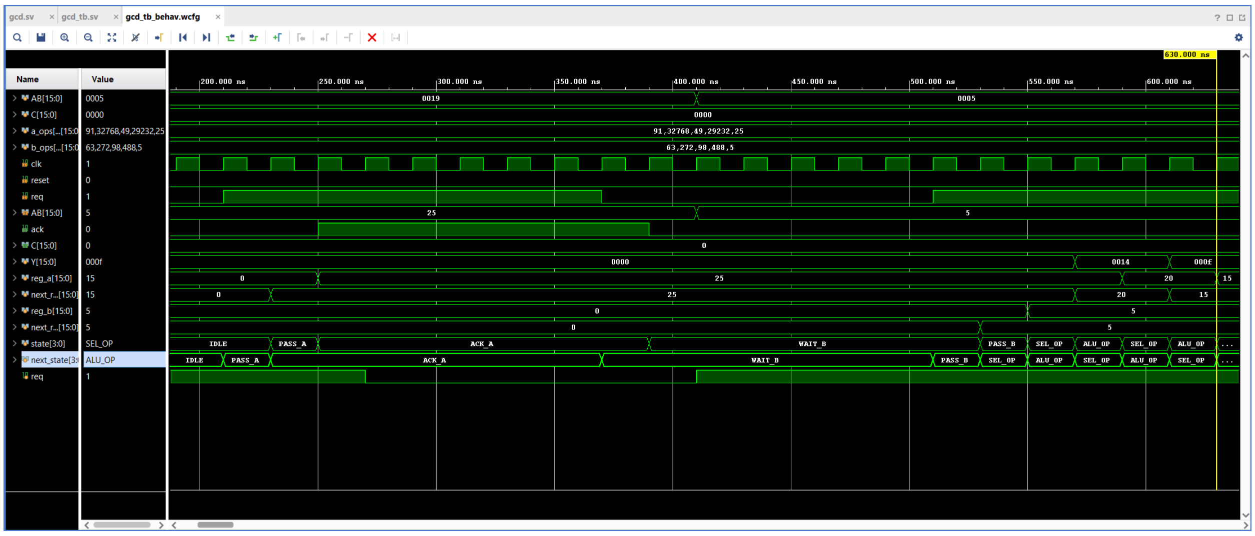


Figure 4: Simulation screenshot 1.

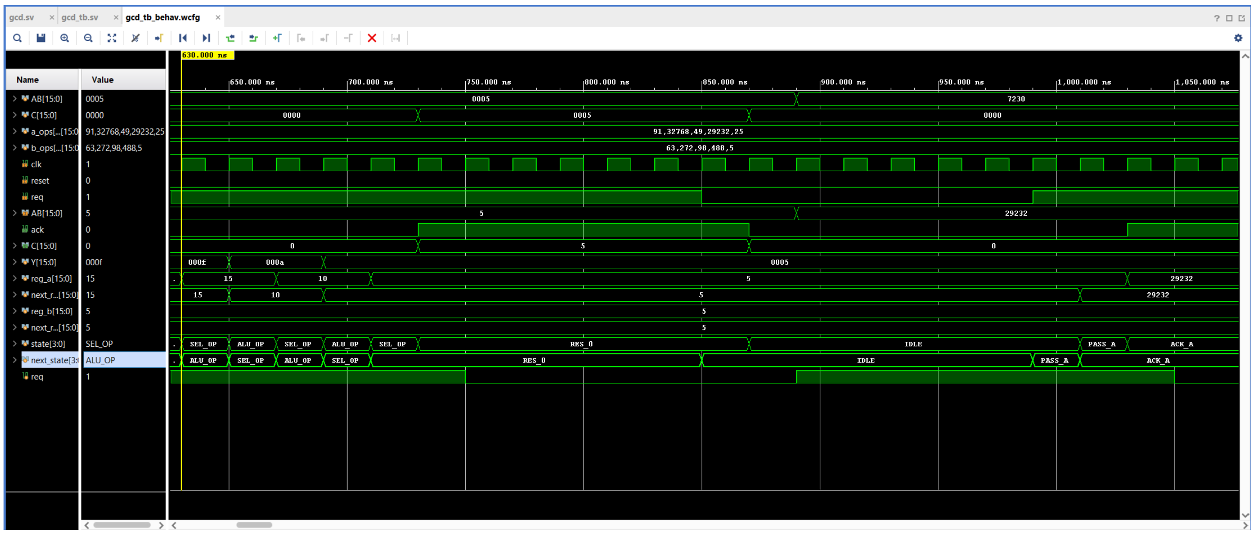
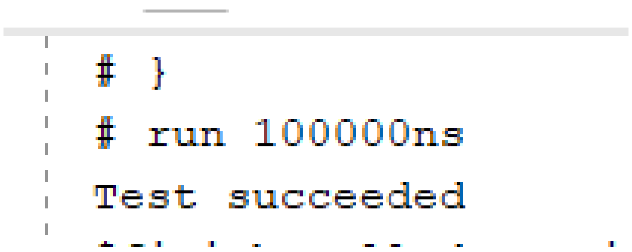


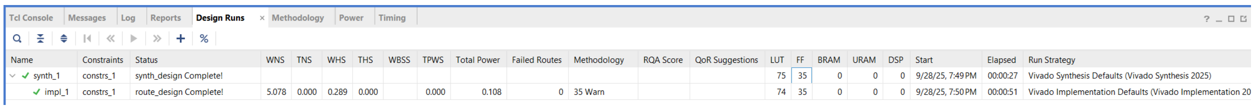
Figure 5: Simulation screenshot 2.



```
# }
# run 100000ns
Test succeeded
```

Figure 6: Simulation succeeded screenshot.

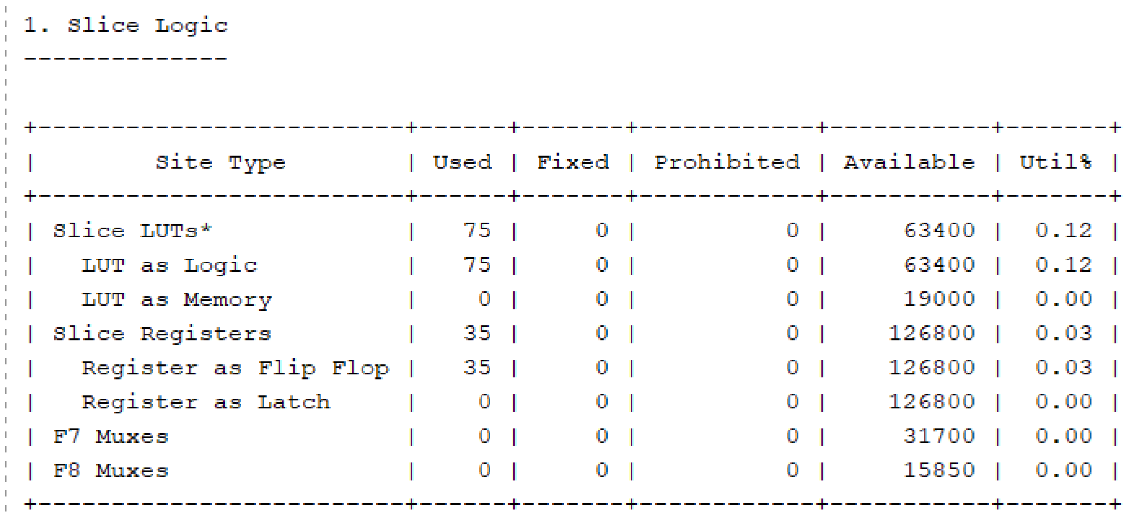
e)



Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	synth_design Complete!												75	35	0	0	0	9/28/25, 7:49 PM	00:00:27	Vivado Synthesis Defaults (Vivado Synthesis 2025)
impl_1	constrs_1	route_design Complete!	5.078	0.000	0.289	0.000		0.000	0.108	0	35 Warn			74	35	0	0	0	9/28/25, 7:50 PM	00:00:51	Vivado Implementation Defaults (Vivado Implementation 20

Figure 7: Synthesize screenshot.

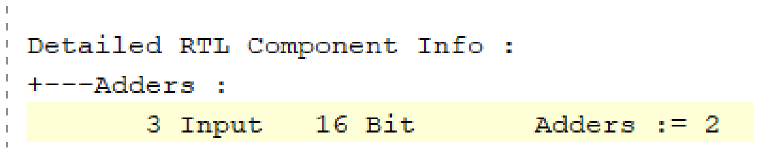
f)



```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   75 |     0 |          0 |     63400 |  0.12 |
|   LUT as Logic          |   75 |     0 |          0 |     63400 |  0.12 |
|   LUT as Memory         |    0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         |   35 |     0 |          0 |    126800 |  0.03 |
|   Register as Flip Flop |   35 |     0 |          0 |    126800 |  0.03 |
|   Register as Latch     |    0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     15850 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

Figure 8: Synthesis report LUTs and registers.



```
Detailed RTL Component Info :
+---Adders :
	   3 Input   16 Bit       Adders := 2
```

Figure 9: Synthesis report adders.

From the synthesis report (Figures 8 and 9), we observe that the design requires **75 LUTs** and **35 registers**. This result is very close to our estimation in Task 0b, where we expected approximately 34 D-Flip-Flops and 72-79 LUTs. Furthermore, the report shows that the design uses **2 adders**. This can be explained by the use of an `if`-statement in our code: since both branches perform a `+` operation, Vivado synthesizes two separate adders (ALUs), one for each condition, rather than sharing a single unit.

5 Task 3

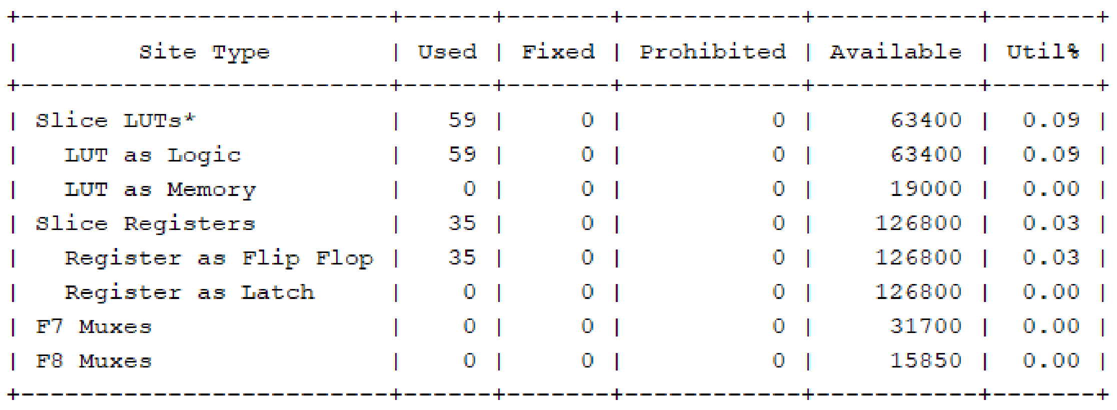
a)

For our optimization, we selected **operator sharing** [1]. In the original design, Vivado created two separate adders because the `if`-statement contained two branches that each used the addition operator. By applying operator sharing, we expect both operations to reuse a single adder. This should **reduce the number of ALUs from 2 to 1**, and as a result, we also anticipate a reduction in the overall number of LUTs required for the implementation.

b)

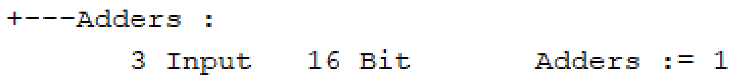
We wrote a new **SystemVerilog** file "task3/gcd.sv" describing the optimized GCD circuit.

c)



```
+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   59 |     0 |          0 |     63400 |  0.09 |
|   LUT as Logic          |   59 |     0 |          0 |     63400 |  0.09 |
|   LUT as Memory         |    0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         |   35 |     0 |          0 |    126800 |  0.03 |
|   Register as Flip Flop |   35 |     0 |          0 |    126800 |  0.03 |
|   Register as Latch     |    0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |     15850 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

Figure 10: Synthesis report LUTs and registers.



```
+---Adders :
           3 Input   16 Bit        Adders := 1
```

Figure 11: Synthesis report adders.

The synthesis report confirms that our optimization was successful. The number of adders was reduced from **2 to 1**, as shown in Figure 11. Moreover, the number of LUTs decreased significantly from 75 to **59 LUTs** (Figure 10), which matches our expectations from part (a). The number of registers remained essentially the same, as this optimization mainly targets arithmetic resource sharing.

The reduction in LUTs can be explained by the fact that operator sharing avoids logic duplication: instead of synthesizing two separate adders and the associated multiplexing logic for the conditional branches, the design reuses one adder, resulting in simpler logic and thus fewer LUTs.

References

[1] DTU, Department of Computer Science and Engineering, "Lecture 4 – Design of Digital Systems (02203), Fall 2025." https://learn.inside.dtu.dk/content/enforced/270903-DTU_e25_02203/02203%20-%20Lecture%204%20-%202025.pdf, 2025. Accessed: 2025-10-02.