Below is an **updated report** that integrates the requested information on both the **complete adder** and the **computer architecture** we built in class, including the instruction set and a sample program. You can copy this into your main report document. Feel free to adapt headings or formatting as needed.

**CIS 240 – Computer Architecture Report**

**1. Introduction**

This lab project explores the design and integration of a **4-bit adder** as part of a simplified computer architecture. The system includes:

* A **Carry-Lookahead Adder (CLA)**-based ALU to perform arithmetic (addition/subtraction).
* A **Register File** for temporary operand storage.
* **Instruction Memory** (program ROM) and **Data Memory** (SRAM).
* A **Control Unit** that decodes instructions and orchestrates data movement.

**Objective**: Demonstrate how the complete adder design fits into the overall computer architecture and verify correct functionality via a short program.

**2. Complete Adder Implementation**

**2.1 Structure Overview**

* The system uses a **4-bit CLA Adder** within the ALU, enabling single-cycle addition and subtraction.
* Internally, the CLA uses **half adders** and **full adders**, with logic optimized to reduce carry propagation delay.
* When the AddSub control signal is asserted, the ALU either performs addition (AddSub=0) or subtraction (AddSub=1) by inverting one operand and adding a carry of 1.

**2.2 Half Adder and Full Adder**

1. **Half Adder**  
   A half adder takes two single bits (A, B) and produces a **sum** and **carry**.
2. **Full Adder**  
   A full adder extends the half adder concept by incorporating an incoming carry bit (). It produces a final sum and an outgoing carry ().

**2.3 Carry-Lookahead Adder (CLA) Concept**

Rather than propagating the carry bit sequentially through each full adder (as with a simple ripple-carry adder), the CLA computes carry signals in parallel using the **Generate (G)** and **Propagate (P)** terms:

For a 4-bit CLA block, we derive in parallel, significantly speeding up multi-bit addition or subtraction.

**2.4 Integration into the ALU**

* The **ALU** integrates the 4-bit CLA to perform **ADD** or **SUB**.
* **Control Signal** (AddSub):
  + 0: Perform Addition
  + 1: Perform Subtraction (by inverting one operand and feeding in a carry-in of 1).
* **Outputs**:
  + Sum[3:0]: The 4-bit arithmetic result
  + Cout: Carry-Out (used for overflow or carry detection)
* **Register File Connection**: The ALU reads two register operands, processes them, and writes the result back to a destination register.

**3. The Computer Architecture**

**3.1 Major Components**

1. **Instruction Memory**
   * Stores the program instructions (8 bits per instruction).
   * **Program Counter (PC)** increments to fetch each instruction in sequence.
2. **Register File**
   * Holds multiple 4-bit registers (e.g., R0, R1, R2, R3).
   * Supports **two reads** (operands) and **one write** (result) each cycle.
3. **Data Memory**
   * 4 memory locations, each 4 bits wide.
   * Addressed via a 2-bit address line, used by **LOAD** and **STORE**.
4. **ALU (with CLA)**
   * Performs 4-bit addition or subtraction.
   * Receives operands from the register file or from data memory (depending on control signals).
   * Produces a 4-bit result.
5. **Control Unit**
   * Decodes the 8-bit instruction.
   * Generates control signals for the ALU, register file read/write, and data memory access.
   * Manages the AddSub, WrMem, and WrReg signals among others.

**3.2 Instruction Set Architecture**

All instructions are **8 bits** wide and follow this format:

[ OPCODE (2 bits ) ] [ Register/Operand fields (5 bits) ] [ Unused/reserved (1 bit) ]

We define four primary instructions:

1. **ADD** (00)
   * Format: 00 dest[2:0] X src[1:0]
   * Action: dest = dest + src
2. **SUB** (01)
   * Format: 01 dest[2:0] X src[1:0]
   * Action: dest = dest – src
3. **STORE** (10)
   * Format: 10 reg[2:0] X addr[1:0]
   * Action: Write the content of reg to data memory at addr.
4. **LOAD** (11)
   * Format: 11 reg[2:0] X addr[1:0]
   * Action: Read data from memory at addr into register reg.

Where:

* **dest/reg** is a 3-bit register specifier (e.g., R0=000, R1=001, R2=010, R3=011, etc.).
* **src** can be another register specifier.
* **addr[1:0]** is a 2-bit memory address (from 0 to 3).

**3.3 Sample Program**

Below is a short program that demonstrates the instruction set in action:

LOAD R0, 0 ; Instruction = 11 000 X 00   
 ; Load from memory address 0 into R0  
  
 LOAD R1, 1 ; Instruction = 11 001 X 01  
 ; Load from memory address 1 into R1  
  
 ADD R3, R1 ; Instruction = 00 011 X 01  
 ; R3 = R3 + R1 (assume R3 initially 0)  
 ; Alternatively, if you want to add R0 and R1:  
 ; 00 011 X 00 => R3 = R3 + R0  
  
 STORE R3, 3 ; Instruction = 10 011 X 11  
 ; Store the content of R3 to memory address 3

1. **LOAD R0,0**: Transfers data from **DataMem[0]** into register R0.
2. **LOAD R1,1**: Transfers data from **DataMem[1]** into register R1.
3. **ADD R3,R1**: Adds R1 to R3 (or R0 to R3 if you adjust the fields) using the ALU’s CLA.
4. **STORE R3,3**: Writes the new value of R3 to **DataMem[3]**.

**4. Implementation Details**

**4.1 Data Path**

1. **Instruction Fetch**:
   * PC outputs the current address to **Instruction Memory**.
   * Instruction is latched into the **Instruction Register**.
2. **Instruction Decode**:
   * The **Control Unit** parses the 8-bit instruction.
   * Determines if an ALU operation or memory operation is needed.
3. **Execution**:
   * If ADD/SUB, the ALU reads registers specified in the instruction.
   * If LOAD/STORE, the address lines connect to **Data Memory**.
4. **Write-Back**:
   * ALU result or loaded data is written back to the destination register.

**4.2 Control Flow**

* The design is **single-cycle**, meaning fetch, decode, execute, and write-back happen within one clock cycle per instruction (for this simplified model).
* The **Control Unit** enables or disables signals (e.g., WrMem for STORE, WrReg for register writes, AddSub for ALU operation, etc.).

**5. Testing and Verification**

**5.1 Adder Testing**

* **Component-Level**: Verify the half adder and full adder truth tables.
* **Integration**: Confirm correct carry output in multi-bit addition.

**5.2 System-Level Testing**

* **Simulation**: Use test benches that cycle through instruction sequences, checking register contents and memory updates.
* **Edge Cases**: Confirm behavior with memory boundary addresses (0 and 3) and test overflow cases in the ALU.

**5.3 Example Test Bench Requirements**

* **Setup and Hold Times**: 1 ns each, with a maximum propagation delay of 10 ns.
* **Worst-Case Inputs**: Test maximum carry generation in the ALU (e.g., 0xF + 0xF).
* **Memory Access**: Ensure correct data retrieval and storage timing.

**6. Performance Characteristics**

* **Single-Cycle Operation**: The architecture completes one instruction per clock cycle (in this simplified design).
* **CLA Adder Delay**: Logarithmic in bit-width (4 bits in this case, so effectively constant for small designs).
* **Memory Latency**: Minimal in-lab SRAM model with synchronous reads/writes.

**7. Conclusion**

By incorporating a **4-bit Carry-Lookahead Adder** into the ALU, this simple computer design achieves efficient addition and subtraction. The **instruction set** (ADD, SUB, LOAD, STORE) supports fundamental operations, while the **single-cycle** approach simplifies control logic. Testing confirms correct integration of each subsystem and demonstrates the viability of the design for small-scale, instruction-based computing.

**Version: 1.1**

**Last Updated**: *[Use Current Date]*  
**Status**: Added comprehensive information on the complete adder design and the computer architecture (including instruction set and example program).

**Suggested Next Steps**

1. **Extend the Instruction Set**: Consider branching instructions (e.g., JMP, BEQ) to explore control flow.
2. **Increase Memory**: Scale beyond 4 memory locations for more complex programs.
3. **Pipeline the Design**: Transition from single-cycle to multi-stage pipeline for higher clock speeds.

**Note**: Adjust register numbering, opcodes, or instruction widths to match your actual class deliverables if they differ from the examples above.