CIS 240 – Computer Architecture Report

Below is a cohesive, integrated report that merges your existing content (on the complete adder and simplified computer architecture) with additional insights from your "new-circuits" directory. You can copy this as is or adapt headings/formatting to align with your lab requirements.

---

1. Introduction

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

- **Carry-Lookahead Adder (CLA)-based ALU:** Performs addition and subtraction.
- **Register File:** Provides storage for operands and results.
- **Instruction Memory (program ROM) and Data Memory (SRAM):** Holds instructions and data.
- **Control Unit:** Decodes instructions and orchestrates data movement between components.

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, optimized to reduce carry propagation delay.
- When the `AddSub` control signal is asserted:
  - `AddSub=0`: Perform **addition**.
  - `AddSub=1`: Perform **subtraction** (by inverting one operand and adding a carry of 1).

2.2 Half Adder and Full Adder

1. **Half Adder**
   - Takes two single bits (A, B).
   - Produces a **sum** and a **carry** bit.
2. **Full Adder**
   - Extends the half-adder with an incoming carry.
   - Produces a final sum and an outgoing carry.

2.3 Carry-Lookahead Adder (CLA) Concept

A CLA pre-computes carry bits in parallel using **Generate (G)** and **Propagate (P)** signals:

- $G_i = A_i \cdot B_i$
- $P_i = A_i \oplus B_i$
- $C_{i+1} = G_i + (P_i \cdot C_i)$

For a 4-bit block, this parallel carry calculation greatly speeds up addition and subtraction compared to ripple-carry designs.

2.4 Integration into the ALU

- **ALU Operation:**
  - Reads two registers (operands) from the register file.
  - Applies CLA-based addition or subtraction.
  - Writes the result back to the destination register.

- **Outputs:**
  - `Sum[3:0]`: 4-bit arithmetic result.
  - `Cout`: Carry-out for detecting overflow or carry.

---

3. The Computer Architecture

3.1 Major Components

1. **Instruction Memory**
   - Stores 8-bit instructions.
   - 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) per cycle.
3. **Data Memory**
   - Four 4-bit memory locations (addresses 0–3).
   - Used by `LOAD` and `STORE` instructions.
4. **ALU (with CLA)**
   - Performs 4-bit addition or subtraction.
   - Receives operands from the register file.
   - Produces a 4-bit result.
5. **Control Unit**
   - Decodes the 8-bit instruction.
   - Manages control signals (`AddSub`, `WrMem`, `WrReg`, etc.).

3.2 Instruction Set Architecture

All instructions are 8 bits wide, following this format:

`[ OPCODE (2 bits) ] [ Register/Operand fields (5 bits) ] [ Unused (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 contents of `reg` to `DataMem[addr]`.
4. **LOAD (11)**
   - Format: `11 reg[2:0] X addr[1:0]`
   - Action: Read from `DataMem[addr]` into `reg`.

Note:

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

3.3 Sample Program

```
LOAD  R0, 0   ; (11 000 X 00) -> Load DataMem[0] into R0
LOAD  R1, 1   ; (11 001 X 01) -> Load DataMem[1] into R1
ADD   R3, R1  ; (00 011 X 01) -> R3 = R3 + R1 (initially R3 = 0)
STORE R3, 3   ; (10 011 X 11) -> Store R3 into DataMem[3]
```

1. `LOAD R0,0`: Transfers data from `DataMem[0]` into `R0`.
2. `LOAD R1,1`: Transfers data from `DataMem[1]` into `R1`.
3. `ADD R3,R1`: Performs addition with the ALU's CLA.
4. `STORE R3,3`: Writes the result to `DataMem[3]`.

---

4. Implementation Details

4.1 Data Path

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

4.2 Control Flow

- Single-cycle design: fetch, decode, execute, and write-back occur in one clock cycle (simplified model).
- Control signals (`WrMem`, `WrReg`, `AddSub`, etc.) are enabled/disabled by the Control Unit.

---

5. Testing and Verification

5.1 Adder Testing

- **Component-Level**: Verify half-adder and full-adder truth tables.
- **Integration**: Confirm correct carry output across multiple bits.

5.2 System-Level Testing

- **Simulation**: Use test benches that run sequences of instructions, checking register contents and memory updates.
- **Edge Cases**: Test memory boundary addresses (0 and 3) and verify overflow handling.

5.3 Example Test Bench Requirements

- **Timing**: Setup and hold times of ˜1ns each, max propagation delay ˜10ns.
- **Worst-Case Inputs**: e.g., `0xF + 0xF` to test maximum carry generation.
- **Memory**: Confirm correct data retrieval and storage timing.

---

6. Performance Characteristics

- **Single-Cycle Operation**: One instruction per clock cycle for the simplified design.
- **CLA Adder Delay**: Logarithmic in bit-width (4 bits, effectively minimal here).

- **Memory Latency**: Minimal for in-lab SRAM with synchronous R/W.

---

7. Incorporating "new-circuits" Directory Insights

In your **new-circuits** directory, you have various `.asc` (LTSpice schematic) and `.asy` (symbol) files indicating a modular approach. These circuits align with the architecture described above:

1. **Registers and Memory**
   - `RegisterBank.asc`: Implements the multi-register file.
   - `DataMem.asc`: Provides 4-bit memory locations.
   - **Integration**: The register bank and data memory directly interface with the ALU and control signals.
2. **Logic Gates and Multiplexers**
   - `MUX4.asc`: 4-to-1 multiplexer essential for routing data within the CPU.
   - `NAND.asc`, `NOR.asc`: Fundamental logic gates used in sub-circuits.
3. **Control and Processing**
   - `ALU.asc`: Your top-level ALU design, integrating the 4-bit CLA.
   - `PC.asc`: A simple program counter that increments to fetch instructions sequentially.
4. **Test Benches**
   - Files like `RegFileTest.asc` and `MUXtest.asc` contain simulation directives (`.tran`, etc.) to verify each module's functionality.

Best Practices Observed

- Clear **SYMATTR** usage: Each schematic file clearly labels components with instance names and values.
- **Subcircuit** approach: Modular design for easier debugging, reusability, and clarity.
- **Simulation** directives: `.tran` statements demonstrate timing checks and functional validation.

7.1 Updating Documentation

- **Include Descriptions**: Document each `.asc` file's purpose in your final report or lab notes (e.g., `RegisterBank.asc` → holds R0–R3 registers).
- **Explain Integration**: Show how data travels from `RegisterBank.asc` → `ALU.asc` → `DataMem.asc` under control signals.
- **Highlight Testing**: Summarize how `RegFileTest.asc` and `MUXtest.asc` confirm correct read/write operations and multiplexing logic.

---

8. Conclusion

By incorporating a **4-bit Carry-Lookahead Adder** within the ALU, this design supports efficient addition and subtraction. The instruction set (ADD, SUB, LOAD, STORE) handles fundamental operations, and a single-cycle approach keeps control logic straightforward. Testing confirms each subsystem (ALU, register file, memory, etc.) works correctly, as demonstrated by the example program and the **new-circuits** test benches.

Version: 1.2
Last Updated: [Insert Date]
Status:

- Added complete adder design notes, instruction set details, and "new-circuits" references.
- Updated with sample program, test methodology, and recommended next steps.

Suggested Next Steps

1. **Extend the Instruction Set**: Add branching (`JMP`, `BEQ`) for more complex control flow.
2. **Expand Memory**: Increase data memory locations beyond the current 4.

3. **Pipeline the Design**: Move from a single-cycle architecture to multi-stage pipelining 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.

Final Note & Further Refinements

- For more advanced simulation, consider adding **timing constraints** for each component in LTSpice or a Verilog/VHDL environment.
- If you integrate the **register bank** and **control logic** into a single schematic or an HDL project, ensure you maintain a **consistent naming convention**.

Solution Endpoint:
You now have a consolidated report reflecting both the **complete adder** and the **simplified computer architecture** details—plus the **new-circuits** directory insights. Feel free to adapt any section headings, add diagrams, or refine your test benches as needed. Let me know if you need any additional adjustments or clarification!