Great! You’ve got a well-defined project specification for **SimpleLang**, a minimalist high-level language targeting an **8-bit CPU**, along with a step-by-step intern task list that covers everything from setting up the environment to compiler implementation and documentation.

To help interns (or yourself) get started and stay focused, here’s a **breakdown, enhancement, and practical suggestions** for each task, along with **next steps** and **potential gotchas** to avoid.

Task Overview + Enhancements

---

1. Setup the 8-bit CPU Simulator

✅ **Action Items**:

- Clone the repo:
- `git clone https://github.com/lightcode/8bit-computer`
- `cd 8bit-computer`
- Read `README.md` and study sample programs.

🧠 **Pro Tip**:

- Run sample programs first. Understand how machine instructions are structured and where they’re loaded.

- If a simulator GUI is provided (like GTKWave or a CLI interface), get comfortable using it.

---

2. Understand the 8-bit CPU Architecture

✅ **Focus Areas**:

- Understand **registers**, **ALU**, **memory layout**, and the **instruction set**.
- Key files:
  - `rtl/machine.v`
  - `rtl/alu.v`
  - `rtl/control_unit.v`

Pro Tip:

- Create a **cheat sheet** summarizing instructions like:
- `LDA addr   ; Load accumulator from memory`
- `ADD addr   ; Add memory content to accumulator`
- `STA addr   ; Store accumulator to memory`
- `JMP addr   ; Unconditional jump`
- `BRZ addr   ; Branch if zero`

---

3. Design SimpleLang (Refinement)

You already defined:

- Variable declarations: `int a;`
- Assignments: `a = b + c;`
- Arithmetic: `+`, `-`
- Conditional: `if (a == b) { ... }`

📘 **Suggestion: Extend the grammar spec**
Provide a more formal EBNF-style definition for better parsing:

```
program       ::= { declaration | statement }
declaration   ::= "int" identifier ";"
statement     ::= assignment | conditional
assignment    ::= identifier "=" expression ";"
expression    ::= term { ("+" | "-") term }
term          ::= identifier | number
conditional   ::= "if" "(" condition ")" block
condition     ::= expression "==" expression
block         ::= "{" { statement } "}"
```

---

4. Create a Lexer in C

You've already provided a basic lexer. To improve:

✅ **Enhance your lexer**:

- Add support for all tokens: `+`, `-`, `==`, `{`, `}`, `;`, `=`
- Fix edge cases: double `==` vs `=`, invalid identifiers, long tokens.

🧠 **Pro Tip**:
Store token types as readable strings during debug:

```
const char* tokenTypeToString(TokenType type);
```

---

5. Develop a Parser

✅ **Goals**:

- Generate an **Abstract Syntax Tree (AST)**
- Catch syntax errors and give line/column numbers

📘 **AST Node Types**:

```
typedef enum {
  NODE_VAR_DECL, NODE_ASSIGN, NODE_IF,
  NODE_BINOP, NODE_LITERAL, NODE_IDENTIFIER
} NodeType;
```

```
typedef struct ASTNode {
  NodeType type;
  // Union of node types (e.g., assignment = left, op, right)
  struct ASTNode* left;
  struct ASTNode* right;
  char value[32]; // for variable names or numbers
} ASTNode;
```

🧠 **Debug Tip**:
Write a function to **pretty-print your AST**.

---

6. Generate Assembly Code

✅ **Approach**:

- Map each AST node to assembly
- Use **symbol table** to store variable-to-address mappings

📘 **Examples**:

SimpleLang:

```
a = b + c;
```

Assembly (assuming vars at known addresses):

```
LDA b
ADD c
STA a
```

Conditional:

```
if (a == b) {
  c = c + 1;
}
```

Assembly:

```
LDA a
SUB b
BRZ label_true
JMP label_end
label_true:
LDA c
ADD one     ; one = memory address holding value 1
STA c
label_end:
```

🧠 **Key Tasks**:

- Generate unique labels (e.g., `label1`, `label2`)

- Handle literal numbers (need to store them in memory if the CPU doesn’t support immediate values)

---

7. Integrate and Test

✅ **Steps**:

- Combine lexer + parser + codegen into a single CLI tool.
- Input: `.slang` file → Output: `.asm` file

🧪 **Testing Plan**:

- Write 3–5 test programs in SimpleLang
- Compare expected vs actual assembly output
- Run `.asm` on 8-bit CPU simulator

---

8. Documentation and Presentation

✅ **Deliverables**:

- Markdown or PDF report:
  - Language Design
  - Compiler Architecture
  - Code Examples
  - Challenges & Solutions
- Slide deck (PowerPoint or Google Slides)
- Demo video (optional but great)

🧠 **Presentation Tips**:

- Show a simple program → compiler output → CPU running it
- Talk about future features (e.g., loops, functions)

🛠 Suggested Directory Structure

```
simplelang-compiler/
├── lexer/
│   └── lexer.c
├── parser/
│   └── parser.c
├── codegen/
│   └── codegen.c
├── ast/
│   └── ast.h / ast.c
├── tests/
│   ├── test1.slang
│   └── expected1.asm
```

```
├── main.c
├── Makefile
└── README.md
```

✅ Suggested Next Step

If you're ready to move forward, we can:

- Expand the **lexer** to support the missing tokens.
- Build an **AST node structure** in C.
- Start writing the **parser** for statements and expressions.

Would you like to proceed with a detailed parser walkthrough next?