LC-3 Assembly Language
VSCode Extension Guide

A Comprehensive Reference for

ECE 306: Introduction to Computing

The University of Texas at Austin

Based on the Patt & Patel

Introduction to Computing Systems ISA

Version 1.0.0

March 2026

Contents

1 Introduction

This guide documents the **LC-3 Assembly Language VSCode Extension**, a development environment designed specifically for the LC-3 instruction set architecture as defined in the Patt & Patel textbook and taught in ECE 306 at UT Austin.

The extension provides:

- **Syntax highlighting** for all 15 LC-3 opcodes, pseudo-ops, TRAP aliases, and more
- **Hover documentation** with encoding diagrams and examples from the textbook
- **Autocomplete** with smart tab-stop snippets
- **Real-time diagnostics** that catch common assembly mistakes
- **Code navigation** — go to definition, find references, symbol outline
- **Snippet library** with 16 common LC-3 programming patterns
- **Signature help** showing expected operands as you type

Prerequisites

You need **Visual Studio Code** (version 1.75 or later) and **Node.js** (version 18 or later) installed on your system to build and run this extension from source.

2 Installation

2.1 Method 1: Development Mode (Quick Start)

This is the fastest way to try the extension. It runs inside a VSCode debug window.

1. Unzip `lc3-vscode-extension.zip` to any directory.
2. Open the `lc3-vscode` folder in VSCode.
3. Open the integrated terminal (`Ctrl+‘`) and run:

```
npm install
npm run compile
```

4. Press `F5` to launch the Extension Development Host.
5. In the new window, open any `.asm` file.

Tip

The extension activates automatically for files ending in `.asm`, `.lc3`, or `.s`. If VSCode doesn't recognize the language, click the language mode in the bottom-right status bar and select **LC-3 Assembly**.

2.2 Method 2: Permanent Install via .vsix

This installs the extension system-wide so it persists across VSCode sessions.

1. Install the packaging tool:

```
npm install -g @vscode/vsce
```

2. Navigate into the extension directory, install dependencies, and package:

```
cd lc3-vscode
npm install
vsce package
```

3. Install the generated `.vsix` file:

```
code --install-extension lc3-assembly-1.0.0.vsix
```

4. Restart VSCode. The extension is now permanently active.

2.3 Method 3: Publish to the VS Marketplace

Publishing to the Marketplace enables seamless syncing across all devices via VSCode Settings Sync.

1. Create a free account at https://dev.azure.com.
2. Generate a **Personal Access Token** with Marketplace publish scope.
3. Log in and publish:

```
npx vsce login <your-publisher-name>
npx vsce publish
```

After publishing, anyone can install it from the Extensions panel in VSCode.

2.4 File Associations

By default, the extension activates for `.asm`, `.lc3`, and `.s` files. To force `.asm` files to always use the LC-3 language mode, add this to your VSCode `settings.json`:

```
"files.associations": {
    "*.asm": "lc3"
}
```

Open settings with `Ctrl+Shift+P` → `Open User Settings (JSON)`.

3 Syntax Highlighting

The extension provides rich, semantic colorization via a TextMate grammar (`lc3.tmLanguage.json`). Every token category receives a distinct scope so that any VSCode color theme will render LC-3

code clearly.

3.1 Token Categories

Category	Scope	Tokens
Operate Instructions	`keyword.operator.arithmetic`	`ADD`, `AND`, `NOT`
Data Movement	`keyword.operator.memory`	`LD`, `LDI`, `LDR`, `LEA`, `ST`, `STI`, `STR`
Control Flow	`keyword.control.flow`	`JMP`, `JSR`, `JSRR`, `RET`, `RTI`, `TRAP`
Branches	`keyword.control.branch`	`BR`, `BRn`, `BRz`, `BRp`, `BRnz`, `BRnp`, `BRzp`, `BRnzp`
TRAP Aliases	`support.function.trap`	`HALT`, `GETC`, `OUT`, `PUTS`, `IN`, `PUTSP`
Pseudo-ops	`keyword.control.directive`	`.ORIG`, `.END`, `.FILL`, `.BLKW`, `.STRINGZ`, `.EXTERNAL`
Registers	`variable.language.register`	`R0` through `R7`
Hex Numbers	`constant.numeric.hex`	`x3000`, `xFE00`, etc.
Decimal Numbers	`constant.numeric.decimal`	`#0`, `#-1`, `#15`, etc.
Binary Numbers	`constant.numeric.binary`	`b0101`, `b11110000`, etc.
Strings	`string.quoted.double`	`"Hello, world!"`
Labels (definition)	`entity.name.function.label`	Labels at the start of a line
Labels (reference)	`variable.other.label`	Labels used as operands
Comments	`comment.line.semicolon`	Everything after `;`

3.2 Case Insensitivity

All opcodes, pseudo-ops, TRAP aliases, and register names are case-insensitive. The grammar handles `ADD`, `add`, `Add`, and any other casing identically. Labels are matched case-insensitively for navigation but are preserved as written for display.

3.3 Example

```
; Multiply NUMBER by 6
.ORIG x3050

        LD   R1, SIX
        LD   R2, NUMBER
        AND  R3, R3, #0      ; Clear accumulator

AGAIN   ADD  R3, R3, R2      ; product += number
        ADD  R1, R1, #-1     ; decrement counter
        BRp  AGAIN

        HALT

NUMBER  .BLKW 1
SIX     .FILL x0006
.END
```

4 Hover Documentation

Hovering over any recognized token displays a rich tooltip with information pulled from the Patt & Patel ISA specification.

4.1 Instruction Hovers

When you hover over an instruction mnemonic (e.g., `ADD`, `LD`, `BRnz`), the tooltip shows:

1. **Mnemonic and category** (Operate, Data Movement, or Control)
2. **Whether condition codes are set** (N, Z, P)
3. **Syntax format(s)** — e.g., `ADD DR, SR1, SR2` and `ADD DR, SR1, imm5`
4. **Binary encoding** — the bit-field layout from bits [15:0]
5. **Full description** — how the instruction works, including which PC is used (incremented PC)
6. **Example usage** — a working code snippet

4.2 Register Hovers

Hovering over `R0`–`R7` displays the register number and any conventions:

Register	Convention Note
`R0`	Used for character I/O by TRAP routines (`GETC`, `OUT`, `IN`, `PUTS`).
`R1`–`R5`	General purpose — no special convention.
`R6`	Conventionally used as the **stack pointer** (SP).
`R7`	Stores the **return address** for `JSR`/`JSRR`/`TRAP`. Overwriting R7 before `RET` will break subroutine return.

4.3 TRAP Alias and Pseudo-op Hovers

Hovering over `HALT`, `PUTS`, `.ORIG`, `.FILL`, etc., displays the same rich tooltip format with the assembler directive's purpose, syntax, and an example.

4.4 Configuring Hover Behavior

The binary encoding section can be toggled in settings:

```
"lc3.hover.showEncoding": true    // or false to hide
```

5 Autocomplete

The extension provides context-aware IntelliSense completions triggered by typing. Pressing `Ctrl+Space` at any point on a line will show all available completions.

5.1 Instruction Completions

Every LC-3 instruction is offered with a **tab-stop snippet** so you can quickly fill in operands:

Type	Inserts
ADD	`ADD R0, R1, R2` (tab between R0, R1, R2)
NOT	`NOT R0, R1`
LD	`LD R0, LABEL`
LDR	`LDR R0, R1, #0`
ST	`ST R0, LABEL`
STR	`STR R0, R1, #0`
BR	`BR⟨n|z|p|nz|np|zp|nzp⟩ LABEL` (dropdown)
JSR	`JSR LABEL`
TRAP	`TRAP x25`

5.2 Pseudo-op Completions

Typing a dot (`.`) triggers pseudo-op suggestions:

Type	Inserts
.ORIG	`.ORIG x3000`
.FILL	`.FILL x0000`
.BLKW	`.BLKW 1`
.STRINGZ	`.STRINGZ ""` (cursor between quotes)
.END	`.END`

5.3 Label Completions

The extension scans your file and offers all defined labels as completion items. Each label shows its line number for easy identification.

5.4 Register Completions

All eight registers (`R0`–`R7`) are offered as completions with their convention notes.

6 Snippet Library

The extension includes 16 snippets for common LC-3 programming patterns. Type the prefix and press `Tab` to expand.

6.1 Program Structure

6.1.1 `program` — Full Program Template

```
; Program description
;
```

```
.ORIG x3000

; Your code here

HALT

; Data section

.END
```

6.1.2 subroutine — Subroutine with R7 Save/Restore

```
; Subroutine: NAME
; Input:  R0
; Output: R0
; Modifies: none
NAME   ST R7, NAME_R7     ; Save return address
       ; subroutine body
       LD R7, NAME_R7     ; Restore return address
       RET
NAME_R7 .BLKW 1
```

R7 Clobbering

If your subroutine calls another subroutine via `JSR`, the nested call will overwrite R7. You **must** save R7 before the inner `JSR` and restore it before `RET`. The `subroutine` and `callee_save` snippets handle this automatically.

6.2 Loop Patterns

6.2.1 loop_counter — Counter-Controlled Loop

```
AND R1, R1, #0      ; Clear counter
ADD R1, R1, #10     ; Set loop count
LOOP ; loop body
     ADD R1, R1, #-1   ; Decrement counter
     BRp LOOP          ; Loop while positive
```

6.2.2 loop_sentinel — Sentinel-Controlled Loop

```
LOOP   LDR R1, R0, #0      ; Load current value
       BRz DONE             ; Exit if sentinel (zero)
       ; process value
       ADD R0, R0, #1       ; Advance pointer
       BRnzp LOOP           ; Continue loop
DONE
```

6.3 Arithmetic and Logic Idioms

6.3.1 `negate` — Two's Complement Negation

```
NOT R1, R0      ; Negate: R1 = -R0
ADD R1, R1, #1  ; Complete two's complement
```

Why NOT + ADD #1?

In two's complement, $-x = \sim x + 1$. The `NOT` instruction computes the bitwise complement ($\sim x = -x - 1$), and adding 1 corrects the off-by-one to give the true negation.

6.3.2 `clear` — Clear a Register

```
AND R0, R0, #0     ; Clear R0
```

6.3.3 `multiply` — Multiply by Repeated Addition

```
; R2 = R0 * R1  (R1 must be >= 0)
AND R2, R2, #0        ; Clear accumulator
MUL_LOOP ADD R1, R1, #0   ; Test multiplier
      BRz MUL_DONE
      ADD R2, R2, R0    ; accumulator += multiplicand
      ADD R1, R1, #-1   ; multiplier--
      BRnzp MUL_LOOP
MUL_DONE
```

6.3.4 `or_demorgan` — Bitwise OR via De Morgan's Law

```
; R2 = R0 OR R1 using NOT(NOT(a) AND NOT(b))
NOT R0, R0
NOT R1, R1
AND R2, R0, R1
NOT R2, R2
; Restore R0 and R1
NOT R0, R0
NOT R1, R1
```

Tip

The LC-3 has no native OR or XOR instructions. Use De Morgan's law: $A \vee B = \neg(\neg A \wedge \neg B)$. For XOR: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.

6.4 Stack Operations

6.4.1 `push` — Push to Stack

```
ADD R6, R6, #-1       ; Decrement stack pointer
STR R0, R6, #0        ; Push R0 onto stack
```

6.4.2 pop — Pop from Stack

```
LDR R0, R6, #0        ; Pop top of stack into R0
ADD R6, R6, #1        ; Increment stack pointer
```

6.4.3 callee_save — Callee-Save Register Pattern

```
; Save registers
ST R7, SAVE_R7
ST R1, SAVE_R1
ST R2, SAVE_R2

; subroutine body

; Restore registers
LD R2, SAVE_R2
LD R1, SAVE_R1
LD R7, SAVE_R7
RET

SAVE_R7 .BLKW 1
SAVE_R1 .BLKW 1
SAVE_R2 .BLKW 1
```

6.5 I/O Patterns

6.5.1 print_string — Print a String

```
LEA R0, MSG           ; Load address of string
PUTS                  ; Print string to console

MSG .STRINGZ "Hello, world!"
```

6.5.2 getchar — Read and Echo a Character

```
GETC     ; Read character into R0 (no echo)
OUT      ; Echo character to console
```

6.5.3 poll_input — Polling Keyboard Input

```
POLL  LDI R0, KBSR
      BRzp POLL           ; Wait until key pressed
      LDI R0, KBDR        ; Read character

KBSR .FILL xFE00
KBDR .FILL xFE02
```

6.5.4 poll_output — Polling Display Output

```
POLL_OUT LDI R1, DSR
       BRzp POLL_OUT       ; Wait until display ready
       STI R0, DDR         ; Write character

DSR .FILL xFE04
DDR .FILL xFE06
```

6.6 Control Flow

6.6.1 `if_else` — Conditional Branch Pattern

```
; if (R0 > 0)
ADD R0, R0, #0       ; Set condition codes
BRnz ELSE            ; Branch to else
  ; then-block
  BRnzp ENDIF
ELSE
  ; else-block
ENDIF
```

7 Real-Time Diagnostics

The extension continuously analyzes your code and reports errors, warnings, and hints in the Problems panel (`Ctrl+Shift+M`).

7.1 Error-Level Diagnostics

These indicate code that will fail to assemble or execute correctly.

7.1.1 Invalid Register

```
ADD R8, R1, R2     ; Error: R8 does not exist
ADD R0, LABEL, R2 ; Error: Expected register for SR1
```

7.1.2 Immediate Out of Range

The LC-3 has strict bit-width limits for immediate values:

Field	Bits	Range (signed)
imm5 (ADD/AND)	5	−16 to +15
offset6 (LDR/STR)	6	−32 to +31
PCoffset9 (LD/ST/BR/etc.)	9	−256 to +255
PCoffset11 (JSR)	11	−1024 to +1023
trapvect8 (TRAP)	8	0 to 255

```
ADD R0, R0, #16    ; Error: imm5 out of range (-16 to 15)
LDR R1, R2, #40    ; Error: offset6 out of range (-32 to 31)
```

7.1.3 Wrong Operand Count

```
ADD R0, R1          ; Error: ADD requires 3 operands
NOT R0, R1, R2      ; Error: NOT requires exactly 2 operands
HALT R0             ; Error: HALT takes no operands
```

7.1.4 Label as ALU Operand

A very common student mistake: using a label where `ADD` or `AND` expects a register or immediate.

```
AND R0, R0, MASK    ; Error: third operand must be
                    ;   register or immediate, not label
; Fix: load the mask into a register first
LD R1, MASK
AND R0, R0, R1      ; Correct
```

7.1.5 Undefined Labels

```
LD R0, CONUT        ; Error: Label "CONUT" is never defined
                    ;   (typo -- should be COUNT)
```

7.1.6 Unknown Opcodes

```
SUB R0, R1, R2      ; Error: Unknown opcode "SUB"
                    ;   (LC-3 has no subtract instruction)
```

7.2 Warning-Level Diagnostics

7.2.1 Missing .ORIG or .END

Every LC-3 program requires both a `.ORIG` directive at the top and a `.END` directive at the bottom. A warning is shown if either is absent.

7.3 Hint-Level Diagnostics

7.3.1 Unused Labels

If a label is defined but never referenced, a hint is displayed. This helps catch dead code or typos in label names.

```
RESULT .BLKW 1      ; Hint: "RESULT" defined but never referenced
```

7.4 Configuration

Setting	Default	Description
`lc3.diagnostics.enable`	`true`	Enable/disable all diagnostics.
`lc3.diagnostics.warnUnusedLabels`	`true`	Show hints for defined but unreferenced labels.

8 Code Navigation

8.1 Go to Definition

Place your cursor on any label reference and press `F12` (or `Ctrl+Click`) to jump to the line where that label is defined.

This works for labels used in instructions like `LD R0, COUNT`, branch targets like `BRp LOOP`, and subroutine calls like `JSR MULTIPLY`.

8.2 Find All References

Press `Shift+F12` on a label to see every location in the file where that label is used. This is invaluable for understanding control flow in larger programs.

8.3 Symbol Outline

Press `Ctrl+Shift+O` to open the symbol outline, which lists every label in your file. Labels are categorized:

- **Function symbols** — labels on code lines (subroutine entry points, branch targets)
- **Variable symbols** — labels on data directives (`.FILL`, `.BLKW`, `.STRINGZ`)

You can type to filter, and clicking a symbol jumps to that line.

8.4 Breadcrumb Navigation

VSCode's breadcrumb bar (top of the editor) shows which label region your cursor is in, making it easy to orient yourself in long programs.

9 Signature Help

As you type operands for an instruction, the extension displays a tooltip showing the expected syntax with the current parameter highlighted. Signature help is triggered by typing a comma (`,`) or a space.

For example, while typing `ADD R0,`:

- The tooltip shows: `ADD DR, SR1, SR2`

- `SR1` is highlighted, indicating it's the next expected operand

For instructions with multiple syntax forms (like `ADD` with register mode vs. immediate mode), both signatures are displayed.

10 Architecture Overview

This section describes how the extension is structured internally, which is useful if you want to modify or extend it.

10.1 File Structure

```
lc3-vscode/
  package.json                  Extension manifest
  language-configuration.json   Bracket/comment settings
  tsconfig.json                 TypeScript configuration
  syntaxes/
    lc3.tmLanguage.json         TextMate grammar
  snippets/
    lc3.json                    Snippet definitions
  src/
    extension.ts                Main entry point
    instructionReference.ts     ISA database
    diagnostics.ts              Error checking
  out/                          Compiled JavaScript
  examples/
    multiply.asm                Example: multiply by 6
    charcount.asm               Example: character counter
```

10.2 Module Descriptions

10.2.1 `instructionReference.ts` — The ISA Database

This is the single source of truth for the entire LC-3 ISA. It exports:

- `LC3_INSTRUCTIONS` — A record mapping each of the 15 opcodes (plus `RET` and `RTI`) to an `InstructionInfo` object containing the mnemonic, brief description, full description, syntax formats, binary encoding, condition code behavior, category, and example.
- `TRAP_ALIASES` — Maps `HALT`, `GETC`, `OUT`, `PUTS`, `IN`, `PUTSP` to their `InstructionInfo` objects with trap vector numbers.
- `PSEUDO_OPS` — Maps `.ORIG`, `.FILL`, `.BLKW`, `.STRINGZ`, `.END`, `.EXTERNAL`.
- `BRANCH_VARIANTS` — Maps `BRn`, `BRz`, `BRp`, `BRnz`, `BRnp`, `BRzp`, `BRnzp` to human-readable descriptions.
- `lookupMnemonic(name)` — Unified lookup that checks all of the above.

10.2.2 `diagnostics.ts` — The Error Checker

Implements a full line-by-line parser that:

1. Strips comments and tokenizes each line into label, opcode, and operands.
2. Collects all defined labels and all referenced labels.
3. Validates each instruction against its expected operand format (register, immediate, label).
4. Checks immediate values against their bit-width constraints.
5. Detects missing `.ORIG`/`.END`, undefined labels, unused labels, and unknown opcodes.

Diagnostics run on every document open, save, and text change, providing real-time feedback.

10.2.3 `extension.ts` — The Main Entry Point

Registers all VSCode language feature providers:

- **HoverProvider** — Looks up the word under cursor in the ISA database and formats a Markdown tooltip.
- **CompletionItemProvider** — Generates completion items for instructions, pseudo-ops, TRAP aliases, registers, and file-local labels.
- **SignatureHelpProvider** — Parses the current line to determine the opcode and active parameter index.
- **DocumentSymbolProvider** — Scans for labels and categorizes them as code or data.
- **DefinitionProvider** — Searches the file for a label definition matching the word under cursor.
- **ReferenceProvider** — Finds all occurrences of a label using regex matching.

11 LC-3 ISA Quick Reference

This section provides a condensed reference of the LC-3 ISA as documented in the extension's hover tooltips.

11.1 The 15 Opcodes

Opcode	Binary	Category	Sets CC?
ADD	0001	Operate	Yes
AND	0101	Operate	Yes
NOT	1001	Operate	Yes
BR	0000	Control	No
JMP	1100	Control	No
JSR	0100	Control	No
JSRR	0100	Control	No
LD	0010	Data Movement	Yes
LDI	1010	Data Movement	Yes

Opcode	Binary	Category	Sets CC?
LDR	0110	Data Movement	Yes
LEA	1110	Data Movement	No
ST	0011	Data Movement	No
STI	1011	Data Movement	No
STR	0111	Data Movement	No
TRAP	1111	Control	No
RET	1100	Alias for `JMP R7`	
RTI	1000	Return from interrupt	
reserved	1101	Unused (illegal opcode exception)	

11.2 TRAP Service Routines

Vector	Alias	Description
x20	`GETC`	Read a character from keyboard (no echo). ASCII code placed in R0[7:0].
x21	`OUT`	Write the character in R0[7:0] to the console.
x22	`PUTS`	Write null-terminated string starting at address in R0.
x23	`IN`	Print prompt, read character (with echo), place in R0[7:0].
x24	`PUTSP`	Write packed string (two chars per word) starting at R0.
x25	`HALT`	Halt the machine.

11.3 Assembler Directives (Pseudo-ops)

Directive	Description
`.ORIG addr`	Set the starting address for the program.
`.FILL val`	Allocate one word, initialized to `val`.
`.BLKW n`	Reserve `n` uninitialized words.
`.STRINGZ "str"`	Store string with null terminator (one char per word).
`.END`	Mark the end of the source file.
`.EXTERNAL sym`	Declare a symbol defined in another module.

11.4 Addressing Modes

Mode	Used By	Effective Address
Immediate	ADD, AND	Value encoded in bits [4:0] (sign-extended)
Register	ADD, AND, NOT	Value in the named register
PC-relative	LD, ST, LEA, LDI, STI, BR, JSR	$PC^{\dagger} + SEXT(offset)$
Base+Offset	LDR, STR	$BaseR + SEXT(offset6)$

Mode	Used By	Effective Address
Indirect	LDI, STI	$mem[PC^{\dagger} + SEXT(offset9)]$

†PC refers to the *incremented* PC (address of the next instruction).

11.5 Condition Codes

The LC-3 has three single-bit condition code registers: **N** (negative), **Z** (zero), and **P** (positive). Exactly one is set at any time. They are updated by instructions marked "Sets CC" in the opcode table above.

The `BR` instruction tests these codes:

- `BRn` branches if the last result was negative
- `BRz` branches if the last result was zero
- `BRp` branches if the last result was positive
- Combinations like `BRnz`, `BRnp`, `BRzp` test multiple conditions (OR logic)
- `BRnzp` (or just `BR`) is an unconditional branch

Common Pitfall

`LEA` does **not** set condition codes, even though it loads a value into a register. If you need to branch based on the result of `LEA`, add `ADD Rx, Rx, #0` afterward to set the condition codes without changing the value.

12 Configuration Reference

All settings are under the `lc3` namespace in VSCode settings.

Setting	Default	Description
`lc3.diagnostics.enable`	`true`	Master toggle for all diagnostic checks.
`lc3.diagnostics .warnUnusedLabels`	`true`	Show hint-level warnings for labels that are defined but never referenced.
`lc3.hover.showEncoding`	`true`	Include the binary encoding bit-field diagram in hover tooltips.

13 Keyboard Shortcuts

These are standard VSCode shortcuts that work with the extension's language features.

Shortcut	Action
`Ctrl+Space`	Trigger autocomplete
`Ctrl+Shift+Space`	Trigger signature help
`F12`	Go to Definition
`Ctrl+Click`	Go to Definition (mouse)
`Shift+F12`	Find All References
`Ctrl+Shift+O`	Open Symbol Outline
`Ctrl+Shift+M`	Open Problems Panel (diagnostics)
`Ctrl+.`	Quick Fix suggestions
`Ctrl+/`	Toggle line comment (`;`)
`F5`	Launch Extension Development Host

On macOS, replace `Ctrl` with `Cmd`.

14 Troubleshooting

14.1 Extension Not Activating

- Verify the file has a recognized extension: `.asm`, `.lc3`, or `.s`.
- Check the language mode in the bottom-right status bar. Click it and select "LC-3 Assembly" if needed.
- If installed via `.vsix`, restart VSCode completely.

14.2 Diagnostics Not Appearing

- Confirm `lc3.diagnostics.enable` is `true` in settings.
- Open the Problems panel (`Ctrl+Shift+M`).
- Check the Output panel (`Ctrl+Shift+U`) for extension errors.

14.3 Syntax Highlighting Looks Wrong

- Some color themes may render certain scopes with similar colors. Try switching to a different theme.
- Ensure the language mode is set to "LC-3 Assembly" and not generic "Assembly".

14.4 Lab Machine Issues

- If you lack permission to install globally, use the `.vsix` method with `code -install-extension`.
- If `npm` is unavailable, carry a pre-packaged `.vsix` on a USB drive or download from Google Drive.
- The compiled `out/` directory is included in the zip, so you can skip `npm run compile` if the TypeScript source hasn't changed.

15 Extending the Extension

The extension is designed to be easily modified. Here are some ideas:

15.1 Adding a New Snippet

Open `snippets/lc3.json` and add a new entry following the existing pattern:

```
"My Pattern": {
    "prefix": "my_trigger",
    "body": [
        "AND ${1:R0}, ${1}, #0",
        "ADD ${1}, ${1}, #${2:5}"
    ],
    "description": "Clear and load a small value"
}
```

Recompile is not needed — snippet changes take effect on reload.

15.2 Adding a New Diagnostic Rule

In `src/diagnostics.ts`, add your check inside the main validation loop. Use the `addDiag()` helper:

```
addDiag(diagnostics, document, line.lineIndex,
    "Your error message here",
    vscode.DiagnosticSeverity.Warning);
```

Then recompile with `npm run compile`.

15.3 Updating Instruction Documentation

All instruction data lives in `src/instructionReference.ts`. To update a description or add a new mnemonic, modify the appropriate object and recompile.

Acknowledgments

This extension and its ISA reference are based on *Introduction to Computing Systems: From Bits & Gates to C/C++ & Beyond* by Yale N. Patt and Sanjay J. Patel, as used in ECE 306 at The University of Texas at Austin under Dr. Nina Telang.