

# ReTI-Tools: A VS Code Extension for ReTI Assembly Programming

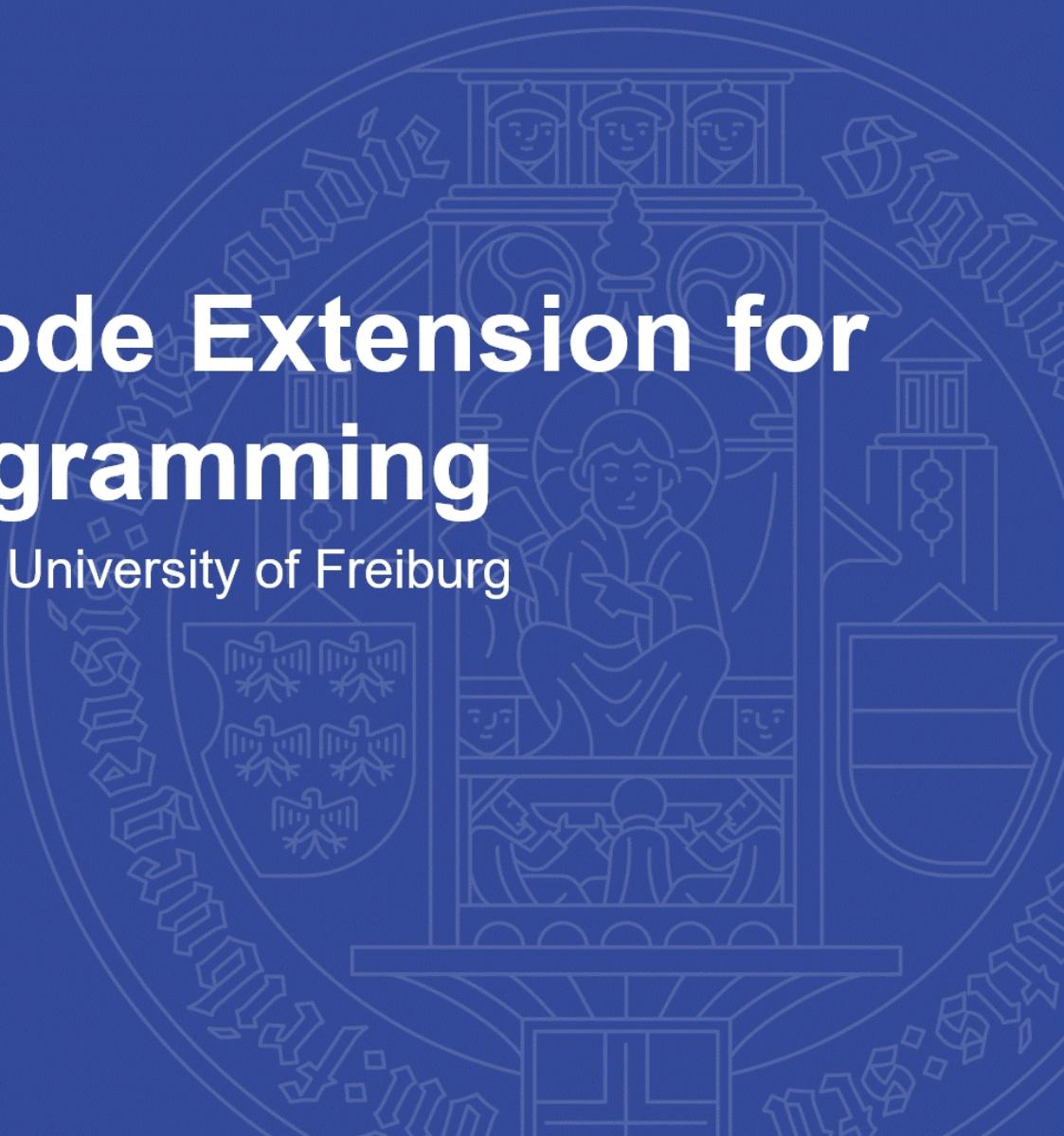
Bachelorthesis for the B. Sc. Informatik at University of Freiburg

Presenter: Malte Pullich

Chair of Computer Architecture

Examiner: Prof. Dr. Armin Biere

Adviser: Dr. Mathias Fleury, Tobias Faller  
Freiburg, 12.11.2025



# Agenda

---

- 1. Background:** ReTI Architecture and Differences between both Versions
- 2. Motivation:** State of the Art and Usage
- 3. Approach:** Improving on Interactivity and Adding Features
- 4. Results:** Improvement on existing Features, new Features
- 5. Conclusion:** Goals Reached & Future Work

# ReTi-Architecture

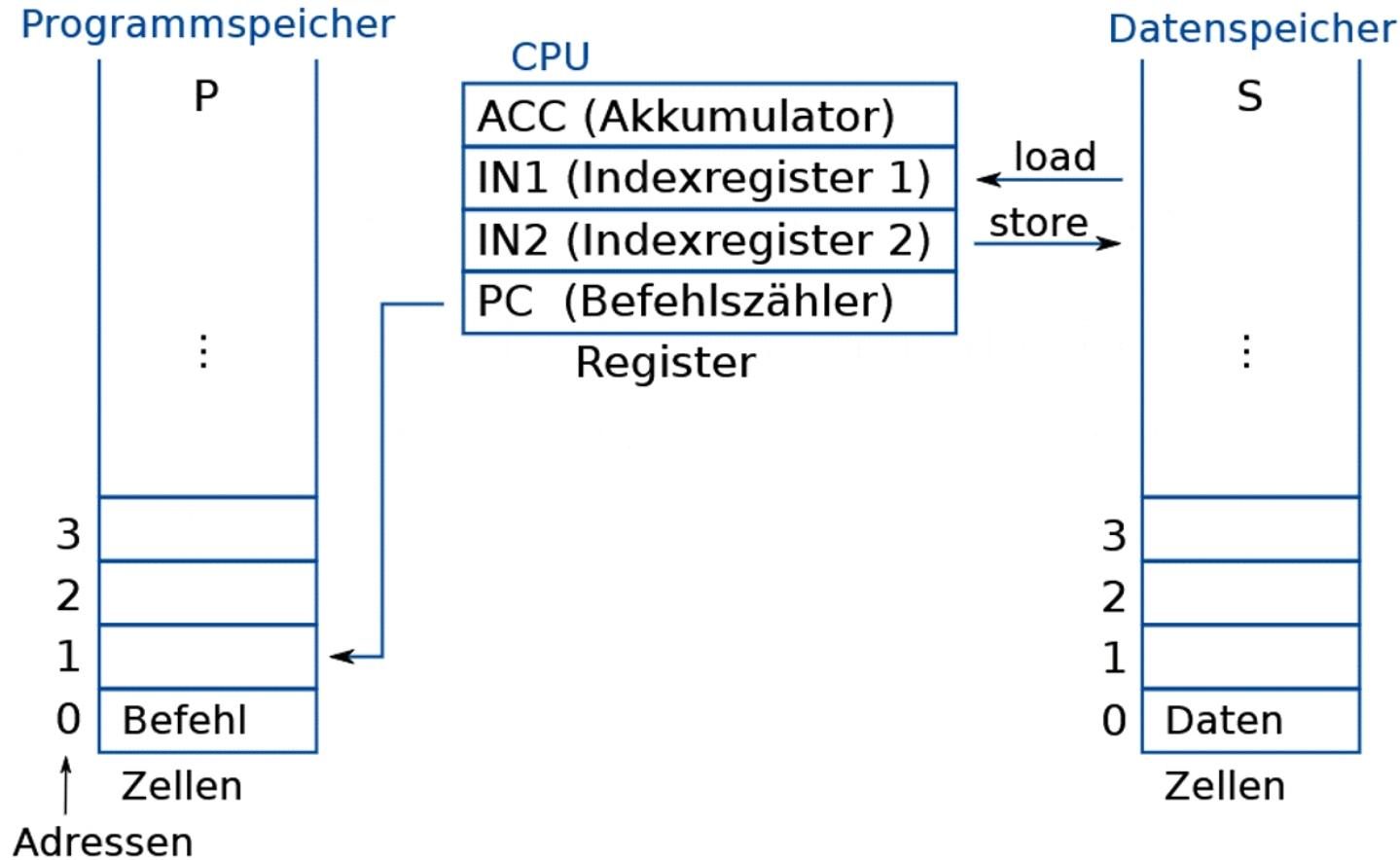


Fig 1: Abstract ReTI architecture [2]

# Differences Between the ReTI Variants

# Differences Between the ReTI Variants

**ReTI-I (Technical Informatics):**

**ReTI-II (Operating Systems):**

# Differences Between the ReTI Variants

## ReTI-I (Technical Informatics):

- Memory is single SRAM

## ReTI-II (Operating Systems):

- Memory split into SRAM, EPROM, UART

# Differences Between the ReTI Variants

## ReTI-I (Technical Informatics):

- Memory is single SRAM
- 1 new internal registers:
  - **I**, Instruction Register

## ReTI-II (Operating Systems):

- Memory split into SRAM, EPROM, UART
- 4 new user-visible registers:
  - **SP**, Stack Pointer
  - **BAF**, Begin Active Frame
  - **CS**, (Begin of) Code Segment
  - **DS**, (Begin of) Data Segment

# Differences Between the ReTI Variants

## ReTI-I (Technical Informatics):

- Memory is single SRAM
- 1 new internal registers:
  - **I**, Instruction Register
- No Interrupts

## ReTI-II (Operating Systems):

- Memory split into SRAM, EPROM, UART
- 4 new user-visible registers:
  - **SP**, Stack Pointer
  - **BAF**, Begin Active Frame
  - **CS**, (Begin of) Code Segment
  - **DS**, (Begin of) Data Segment
- **Interrupts:**
  - Interrupt controller
  - New register **IVN**

# Differences Between the ReTI Variants

## ReTI-I (Technical Informatics):

- Memory is single SRAM
- 1 new internal registers:
  - **I**, Instruction Register
- No Interrupts

## ReTI-II (Operating Systems):

- Memory split into SRAM, EPROM, UART
- 4 new user-visible registers:
  - **SP**, Stack Pointer
  - **BAF**, Begin Active Frame
  - **CS**, (Begin of) Code Segment
  - **DS**, (Begin of) Data Segment
- **Interrupts:**
  - Interrupt controller
  - New register **IVN**
- **New instruction encoding (3 bits for registers)**

# Differences Between the ReTI Variants

## ReTI-I (Technical Informatics):

- Memory is single SRAM
- 1 new internal registers:
  - **I**, Instruction Register
- No Interrupts

## ReTI-II (Operating Systems):

- Memory split into SRAM, EPROM, UART
- 4 new user-visible registers:
  - **SP**, Stack Pointer
  - **BAF**, Begin Active Frame
  - **CS**, (Begin of) Code Segment
  - **DS**, (Begin of) Data Segment
- **Interrupts:**
  - Interrupt controller
  - New register **IVN**
- **New instruction encoding (3 bits for registers)**
- **New instructions (MUL, DIV, MOD)**

# Motivation

The screenshot shows the ReTI-Tools emulator interface. At the top, there are buttons for RUN and UART, and dropdown menus for light, Examples, Pico-C, PLAY, PREV, NEXT, SHOW GRAPH (disabled), SPEED: 1 Hz, and NUMBER STYLE. A green message box indicates "Compilation successful. Took 1357ms".

**Instruction 0 | FETCH P0**  
LOADI DS -2097152

**Registers:**

PC	IN1	IN2	ACC	SP	BAF	CS	DS
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

**Memory:**

SRAM		EPROM		UART	
ADDRESS	DATA	A	DATA	REGISTER	DATA
0	JUMP 0	0	LOADI DS -2097152	R0	00000000
1	2147483648	1	MULI DS 1024	R1	00000000
2	0	2	MOVE DS SP	R2	00000001
3	0	3	3208642560	R3	00000000

**UART Output:**

```
1 void main() {
2     int x;
3     int y;
4     int z;
5     while (1) {
6         x = 0;
7         y = 1;
8         while (x < 255) {
9             z = x + y;
10            x = y;
11            y = z;
12            print(x);
13        }
14    }
15 }
```

Fig. 2 : Screenshot of the Emulator by Michel Giehl [3]

# Motivation

The screenshot shows the ReTI-Tools Emulator interface. At the top, there are buttons for RUN and UART, and dropdown menus for light, Examples, Pico-C, PLAY, PREV, NEXT, SHOW GRAPH (disabled), SPEED: 1 Hz, and NUMBER STYLE. A green message box indicates "Compilation successful. Took 1357ms".

**Instruction 0 | FETCH P0**  
LOADI DS -2097152

**Registers:**

PC	IN1	IN2	ACC	SP	BAF	CS	DS
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

**Memory:**

SRAM		EPROM		UART	
ADDRESS	DATA	A	DATA	REGISTER	DATA
0	JUMP 0	0	LOADI DS -2097152	R0	00000000
1	2147483648	1	MULI DS 1024	R1	00000000
2	0	2	MOVE DS SP	R2	00000001
3	0	3	3208642560	R3	00000000

**UART Output:**

```
1 void main() {
2     int x;
3     int y;
4     int z;
5     while (1) {
6         x = 0;
7         y = 1;
8         while (x < 255) {
9             z = x + y;
10            x = y;
11            y = z;
12            print(x);
13        }
14    }
15 }
```

Emulator by Michel Giehl:

Fig. 2 : Screenshot of the Emulator by Michel Giehl [3]

# Motivation

The screenshot shows the ReTI-Tools emulator interface. At the top, there are buttons for RUN and UART, and dropdown menus for light, Examples, Pico-C, PLAY, PREV, NEXT, SHOW GRAPH (disabled), SPEED: 1 Hz, and NUMBER STYLE. A green message box indicates "Compilation successful. Took 1357ms".

**Instruction 0 | FETCH P0**  
LOADI DS -2097152

**Registers:**

PC	IN1	IN2	ACC	SP	BAF	CS	DS
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

**Memory:**

SRAM		EPROM		UART	
ADDRESS	DATA	A	DATA	REGISTER	DATA
0	JUMP 0	0	LOADI DS -2097152	R0	00000000
1	2147483648	1	MULI DS 1024	R1	00000000
2	0	2	MOVE DS SP	R2	00000001
3	0	3	3208642560	R3	00000000

**UART Output:**

```
1 void main() {
2     int x;
3     int y;
4     int z;
5     while (1) {
6         x = 0;
7         y = 1;
8         while (x < 255) {
9             z = x + y;
10            x = y;
11            y = z;
12            print(x);
13        }
14    }
15 }
```

Fig. 2 : Screenshot of the Emulator by Michel Giehl [3]

## Emulator by Michel Giehl:

- Web app, as of Monday 10<sup>th</sup> November, only accessible through Uni network

# Motivation

The screenshot shows the ReTI-Tools emulator interface. At the top, there are buttons for RUN and UART, and dropdown menus for light, Examples, Pico-C, PLAY, PREV, NEXT, SHOW GRAPH (disabled), SPEED: 1 Hz, and NUMBER STYLE. A green message box indicates "Compilation successful. Took 1357ms".

**Instruction 0 | FETCH P0**  
LOADI DS -2097152

**Registers:**

PC	IN1	IN2	ACC	SP	BAF	CS	DS
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

**Memory:**

SRAM		EPROM		UART	
ADDRESS	DATA	A	DATA	REGISTER	DATA
0	JUMP 0	0	LOADI DS -2097152	R0	00000000
1	2147483648	1	MULI DS 1024	R1	00000000
2	0	2	MOVE DS SP	R2	00000001
3	0	3	3208642560	R3	00000000

**Serial Port:**

1908408320
834666496

Fig. 2 : Screenshot of the Emulator by Michel Giehl [3]

## Emulator by Michel Giehl:

- Web app, as of Monday 10<sup>th</sup> November, only accessible through Uni network
- Doesn't support interrupts

# Motivation

Registers	SRAM Codesegment: PC (2147483652)	SRAM Datasegment: DS (2147483670)	SRAM Stack: SP (2147549183)
PC: 2147483652 (-2147483644)	00000: ADDI PC 5<- CS	00005: STORE ACC 3	65501: -631242752
IN1: 0 (0)	00001: ADDI PC 2	00006: LOADI IN2 3	65502: -631242752
IN2: 0 (0)	00002: LOADI ACC 2	00007: JUMP<= 12	65503: -631242752
ACC: 2 (2)	00003: STORE ACC 1	00008: LOAD IN1 2	65504: -631242752
SP: 2147549183 (-2147418113)	00004: LOADI ACC 1<- PC	00009: LOAD ACC 3	65505: -631242752
BAF: 2147549183 (-2147418113)	00005: STORE ACC 3	00010: STORE ACC 2	65506: -631242752
CS: 2147483648 (-2147483648)	00006: LOADI IN2 3	00011: ADD IN1 3	65507: -631242752
DS: 2147483670 (-2147483626)	00007: JUMP<= 12	00012: MOVE IN1 ACC	65508: -631242752
	00008: LOAD IN1 2	00013: STORE ACC 3	65509: -631242752
	00009: LOAD ACC 3	00014: ADDI ACC 1	65510: -631242752
EPROM: PC (2147483652)	00010: STORE ACC 2	00015: STORE ACC 1	65511: -631242752
	00011: ADD IN1 3	00016: LOAD ACC 0	65512: -631242752
	00012: MOVE IN1 ACC	00017: SUB ACC 1	65513: -631242752
	00013: STORE ACC 3	00018: JUMP -11	65514: -631242752
	00014: ADDI ACC 1	00019: NOP	65515: -631242752
	00015: STORE ACC 1	00020: JUMP 0	65516: -631242752
	00016: LOAD ACC 0	00021: ADD ACC IN1	65517: -631242752
	00017: SUB ACC 1	00022: 281542656<- DS	65518: -631242752
	00018: JUMP -11	00023: 281542656	65519: -631242752
	00019: NOP	00024: 281542656	65520: -631242752
	00020: JUMP 0	00025: 281542656	65521: -631242752
	00021: ADD ACC IN1	00026: 281542656	65522: -631242752
	00022: 281542656<- DS	00027: 281542656	65523: -631242752
UART	00023: 281542656	00028: 281542656	65524: -631242752
0: 0	00024: 281542656	00029: 281542656	65525: -631242752
1: 0	00025: 281542656	00030: 281542656	65526: -631242752
2: 3	00026: 281542656	00031: 281542656	65527: -631242752
Current send data:	00027: 281542656	00032: 281542656	65528: -631242752
All send data:	00028: 281542656	00033: 281542656	65529: -631242752
Waiting time sending: 0	00029: 281542656	00034: 281542656	65530: -631242752
Waiting time receiving: 0	00030: 281542656	00035: 281542656	65531: -631242752
Current input:	00031: 281542656	00036: 281542656	65532: -631242752
Remaining input:	00032: 281542656	00037: 281542656	65533: -631242752
	00033: 281542656	00038: 281542656	65534: -631242752
	00034: 281542656	00039: 281542656	65535: -631242752<- SP BAF

(n)ext instruction, (c)ontinue to breakpoint, (r)estart, (s)tep into isr, (f)inalize isr, (t)rigger isr, (a)ssign watchobject reg or addr,

Fig. 3 : Screenshot of the Emulator by Jürgen Mattheis [1]

# Motivation

Registers	SRAM Codesegment: PC (2147483652 00000: ADDI PC 5<- CS 00001: ADDI PC 2 00002: LOAD ACC 2 00003: STORE ACC 1 00004: LOADI ACC 1<- PC 00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656	SRAM Datasegment: DS (2147483670 00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656 00035: 281542656 00036: 281542656 00037: 281542656 00038: 281542656 00039: 281542656	SRAM Stack: SP (2147549183) 65501: -631242752 65502: -631242752 65503: -631242752 65504: -631242752 65505: -631242752 65506: -631242752 65507: -631242752 65508: -631242752 65509: -631242752 65510: -631242752 65511: -631242752 65512: -631242752 65513: -631242752 65514: -631242752 65515: -631242752 65516: -631242752 65517: -631242752 65518: -631242752 65519: -631242752 65520: -631242752 65521: -631242752 65522: -631242752 65523: -631242752 65524: -631242752 65525: -631242752 65526: -631242752 65527: -631242752 65528: -631242752 65529: -631242752 65530: -631242752 65531: -631242752 65532: -631242752 65533: -631242752 65534: -631242752 65535: -631242752<- SP BAF
EPROM: PC (2147483652)			
UART			
0: 0 1: 0 2: 3 Current send data: All send data: Waiting time sending: 0 Waiting time receiving: 0 Current input: Remaining input:			

(n)ext instruction, (c)ontinue to breakpoint, (r)estart, (s)tep into isr, (f)inalize isr, (t)rigger isr, (a)ssign watchobject reg or addr,

Fig. 3 : Screenshot of the Emulator by Jürgen Mattheis [1]

Emulator by Jürgen Mattheis:

# Motivation

Registers	SRAM Codesegment: PC (2147483652 00000: ADDI PC 5<- CS 00001: ADDI PC 2 00002: LOAD ACC 2 00003: STORE ACC 1 00004: LOADI ACC 1<- PC 00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656	SRAM Datasegment: DS (2147483670 00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656 00035: 281542656 00036: 281542656 00037: 281542656 00038: 281542656 00039: 281542656	SRAM Stack: SP (2147549183) 65501: -631242752 65502: -631242752 65503: -631242752 65504: -631242752 65505: -631242752 65506: -631242752 65507: -631242752 65508: -631242752 65509: -631242752 65510: -631242752 65511: -631242752 65512: -631242752 65513: -631242752 65514: -631242752 65515: -631242752 65516: -631242752 65517: -631242752 65518: -631242752 65519: -631242752 65520: -631242752 65521: -631242752 65522: -631242752 65523: -631242752 65524: -631242752 65525: -631242752 65526: -631242752 65527: -631242752 65528: -631242752 65529: -631242752 65530: -631242752 65531: -631242752 65532: -631242752 65533: -631242752 65534: -631242752 65535: -631242752<- SP BAF
EPROM: PC (2147483652)			
UART			
0: 0 1: 0 2: 3 Current send data: All send data: Waiting time sending: 0 Waiting time receiving: 0 Current input: Remaining input:			

(n)ext instruction, (c)ontinue to breakpoint, (r)estart, (s)tep into isr, (f)inalize isr, (t)rigger isr, (a)ssign watchobject reg or addr,

Fig. 3 : Screenshot of the Emulator by Jürgen Mattheis [1]

## Emulator by Jürgen Mattheis:

- OS-dependent (only Linux support)

# Motivation

Registers	SRAM Codesegment: PC (2147483652)	SRAM Datasegment: DS (2147483670)	SRAM Stack: SP (2147549183)	
PC: 2147483652 (-2147483644) IN1: 0 (0) IN2: 0 (0) ACC: 2 (2) SP: 2147549183 (-2147418113) BAF: 2147549183 (-2147418113) CS: 2147483648 (-2147483648) DS: 2147483670 (-2147483626)	00000: ADDI PC 5<- CS 00001: ADDI PC 2 00002: LOAD ACC 2 00003: STORE ACC 1 00004: LOADI ACC 1<- PC 00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656	00005: STORE ACC 3 00006: LOADI IN2 3 00007: JUMP<= 12 00008: LOAD IN1 2 00009: LOAD ACC 3 00010: STORE ACC 2 00011: ADD IN1 3 00012: MOVE IN1 ACC 00013: STORE ACC 3 00014: ADDI ACC 1 00015: STORE ACC 1 00016: LOAD ACC 0 00017: SUB ACC 1 00018: JUMP -11 00019: NOP 00020: JUMP 0 00021: ADD ACC IN1 00022: 281542656<- DS 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656 00035: 281542656 00036: 281542656 00037: 281542656 00038: 281542656 00039: 281542656	65501: -631242752 65502: -631242752 65503: -631242752 65504: -631242752 65505: -631242752 65506: -631242752 65507: -631242752 65508: -631242752 65509: -631242752 65510: -631242752 65511: -631242752 65512: -631242752 65513: -631242752 65514: -631242752 65515: -631242752 65516: -631242752 65517: -631242752 65518: -631242752 65519: -631242752 65520: -631242752 65521: -631242752 65522: -631242752 65523: -631242752 65524: -631242752 65525: -631242752 65526: -631242752 65527: -631242752 65528: -631242752 65529: -631242752 65530: -631242752 65531: -631242752 65532: -631242752 65533: -631242752 65534: -631242752 65535: -631242752<- SP BAF	
EPROM: PC (2147483652)				
UART	0: 0 1: 0 2: 3 Current send data: All send data: Waiting time sending: 0 Waiting time receiving: 0 Current input: Remaining input:	00000: 281542656 00001: 281542656 00002: 281542656 00003: 281542656 00004: 281542656 00005: 281542656 00006: 281542656 00007: 281542656 00008: 281542656 00009: 281542656 00010: 281542656 00011: 281542656 00012: 281542656 00013: 281542656 00014: 281542656 00015: 281542656 00016: 281542656 00017: 281542656 00018: 281542656 00019: 281542656 00020: 281542656 00021: 281542656 00022: 281542656 00023: 281542656 00024: 281542656 00025: 281542656 00026: 281542656 00027: 281542656 00028: 281542656 00029: 281542656 00030: 281542656 00031: 281542656 00032: 281542656 00033: 281542656 00034: 281542656		

(n)ext instruction, (c)ontinue to breakpoint, (r)estart, (s)tep into isr, (f)inalize isr, (t)rigger isr, (a)ssign watchobject reg or addr,

Fig. 3 : Screenshot of the Emulator by Jürgen Mattheis [1]

## Emulator by Jürgen Mattheis:

- OS-dependent (only Linux support)
- Workflow requires switching between editor and debugger

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

Extend existing VS Code extension [6]:

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

- Emulator (implement ReTI-II alongside ReTI-I)

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

- Emulator (implement ReTI-II alongside ReTI-I)
- Debugger (implement ReTI-II alongside ReTI-I)

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

- Emulator (implement ReTI-II alongside ReTI-I)
- Debugger (implement ReTI-II alongside ReTI-I)
- Language Server (implement ReTI-II alongside ReTI-I)

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

- Emulator (implement ReTI-II alongside ReTI-I)
- Debugger (implement ReTI-II alongside ReTI-I)
- Language Server (implement ReTI-II alongside ReTI-I)

**Identified additional requirement:**

# Approach

**Goal:** Improve **accessibility** and **responsiveness** to provide a better educational experience.

**Extend existing VS Code extension [6]:**

- Most popular editor/IDE
- OS-independent (available for Linux, Mac, Windows)
- Make it a single tool supporting both lectures

**Identified key features to extend:**

- Emulator (implement ReTI-II alongside ReTI-I)
- Debugger (implement ReTI-II alongside ReTI-I)
- Language Server (implement ReTI-II alongside ReTI-I)

**Identified additional requirement:**

- Provide a memory view

# Approach

## Extension's Architecture

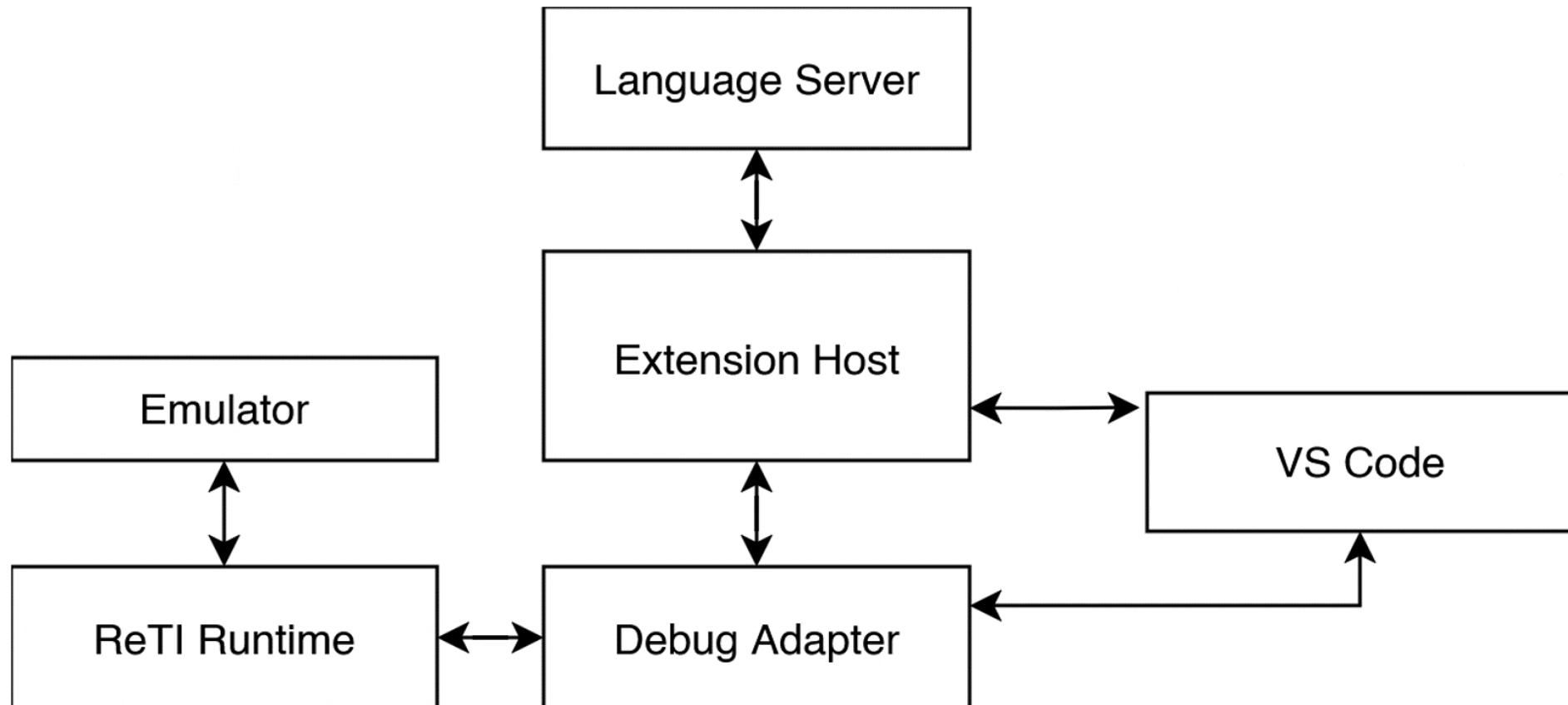


Fig. 4: Diagram illustrating the interaction between the different components of the extension.

# Approach

## Extension's Architecture

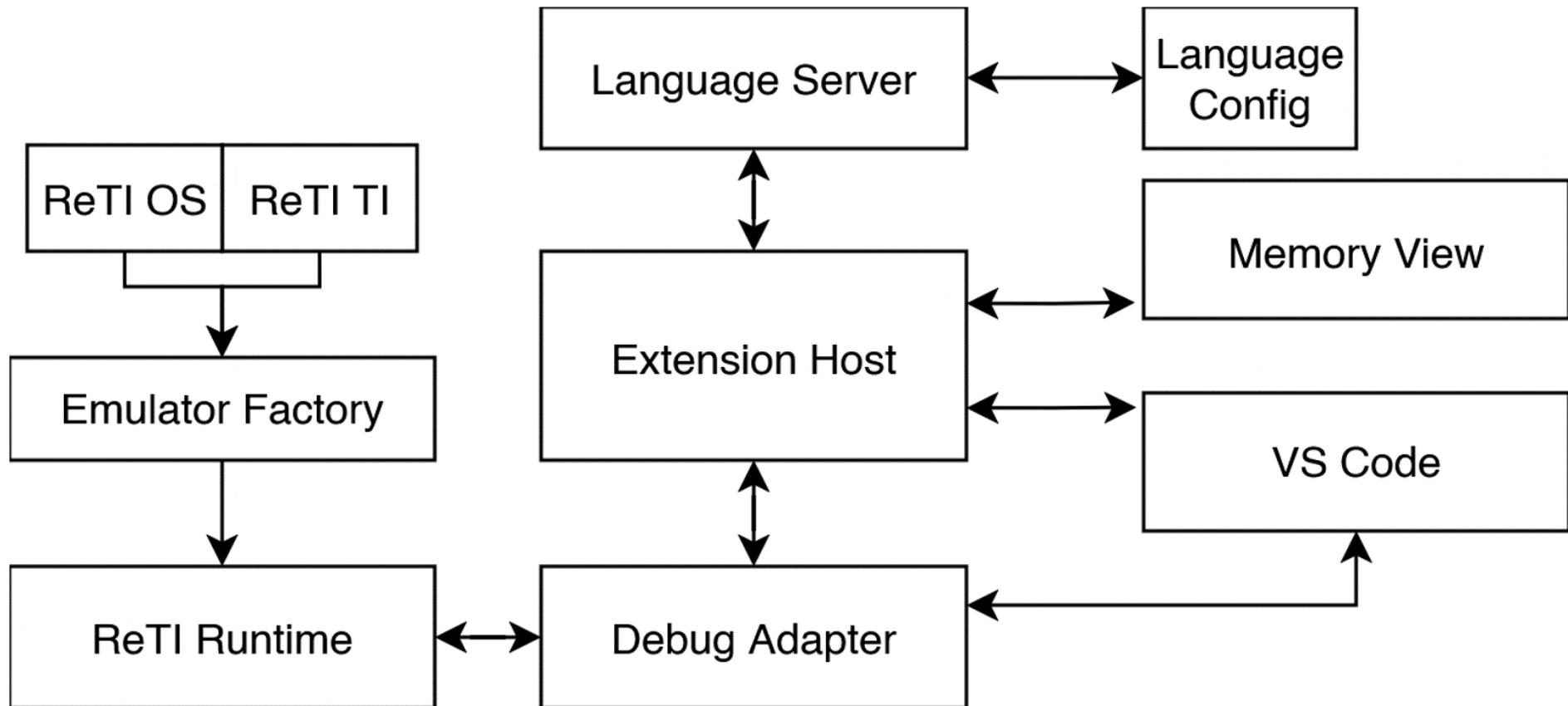


Fig. 4: Diagram illustrating the interaction between the different components of the extension.

# Results

## Support for Both Architectures

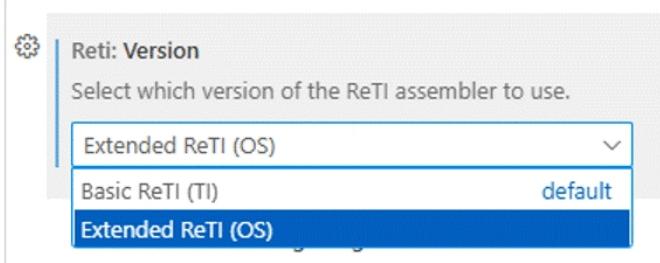


Fig 5: Screenshot of the new setting in VS Code

The screenshot shows the VS Code editor with an open file named "testisrs.reti". The code contains the following assembly instructions:

```
1 ; Initialization
2 LOADI ACC 5           ; Initiate n = 5,
3 STORE ACC 0           ; Store n in M<0>
4 LOADI ACC 2           ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1           ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
```

A context menu is open over the code, with the "Emulate" option highlighted. The menu also includes "Debug ReTI" and the keyboard shortcut "Ctrl+Alt+E".

Fig 6: Example Usage of the ReTI Emulation in VS Code

# Results

## Support for Both Architectures

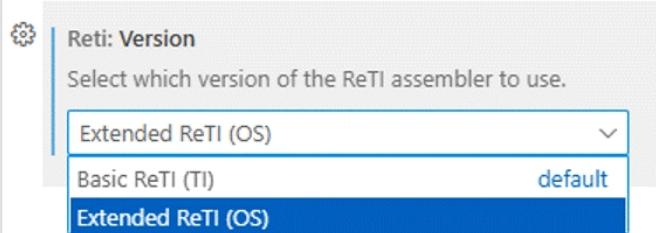


Fig 5: Screenshot of the new setting in VS Code

The screenshot shows the VS Code editor with an assembly file named "testisrs.reti". The code consists of six lines of assembly instructions:

```
1 ; Initialization
2 LOADI ACC 5           ; Initiate n = 5,
3 STORE ACC 0           ; Store n in M<0>
4 LOADI ACC 2           ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1           ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
```

An "Emulate" context menu is open over the code, showing the command "Debug ReTI" with a keyboard shortcut "Ctrl+Alt+D".

Fig 6: Example Usage of the ReTI Emulation in VS Code

- Added setting to specify desired version

# Results

## Support for Both Architectures

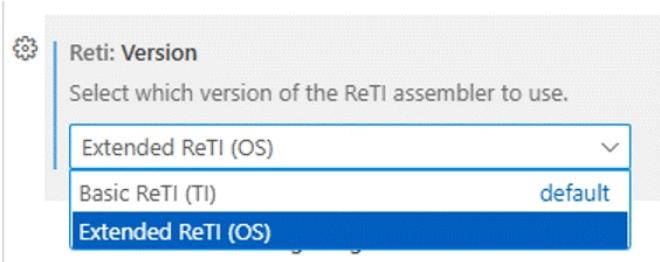


Fig 5: Screenshot of the new setting in VS Code

The screenshot shows the VS Code editor with an assembly file named "testisrs.reti". The code consists of six lines of assembly instructions:

```
1 ; Initialization
2 LOADI ACC 5           ; Initiate n = 5,
3 STORE ACC 0           ; Store n in M<0>
4 LOADI ACC 2           ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1           ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
```

A context menu is open over the code, with the "Emulate" option highlighted. The menu also includes "Debug ReTI" and the keyboard shortcut "Ctrl+Alt+E".

Fig 6: Example Usage of the ReTI Emulation in VS Code

- Added setting to specify desired version
- Affects all features (emulator, debugger, language server) except quiz

# Results

## Support for Both Architectures

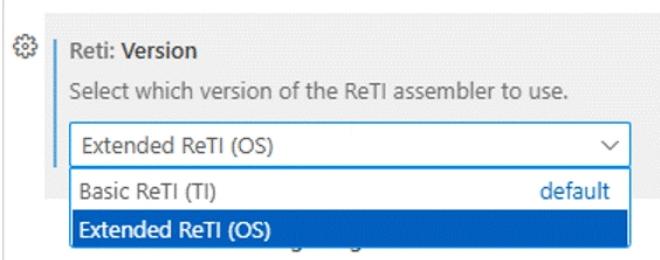


Fig 5: Screenshot of the new setting in VS Code

The screenshot shows the VS Code editor with an assembly file named 'testisrs.reti'. The code contains the following instructions:

```
1 ; Initialization
2 LOADI ACC 5           ; Initiate n = 5,
3 STORE ACC 0           ; Store n in M<0>
4 LOADI ACC 2           ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1           ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
```

A context menu is open over the code, with the 'Emulate' option highlighted. The menu also includes 'Debug ReTI' and a keyboard shortcut 'Ctrl+Alt+E'.

Fig 6: Example Usage of the ReTI Emulation in VS Code

- Added setting to specify desired version
- Affects all features (emulator, debugger, language server) except quiz
- Emulator now callable in .reti files for ReTI-II (OS)

# Results

## Language Server

A screenshot of a VS Code editor window displaying assembly code for the ReTI architecture. The code is contained in a file named 'testisrs.reti'. Several syntax errors are highlighted with red squiggly lines:

- Line 2: 'IVTE 2' is highlighted with an error message: 'IVTE command is only intended for use in isr files.'
- Line 3: 'STD' is highlighted with an error message: 'Unknown instruction "STD".'
- Line 4: 'LOA' is highlighted with a tooltip: 'STORE S i'.
- Line 5: 'STO' is highlighted with a tooltip: 'STOREIN'.
- Line 6: 'INT' is highlighted with a tooltip: 'F(n-1) will be stored in M<2>, since its 0 no intializati'.
- Line 7: 'INT' is highlighted with a tooltip: 'ored in M<3>'.
- Line 14: 'JUMP<= ACC' is highlighted with an error message: 'Invalid operand.'
- Line 15: 'LOAD IN1 2' is highlighted with a tooltip: 'LOAD F(n-1) into IN1'.
- Line 16: 'LOAD ACC 3' is highlighted with a tooltip: 'LOAD F(n) into IN1'.

Fig 7: Screenshot from VS Code highlighting LSP Features

# Results

## Language Server

A screenshot of the VS Code editor showing ReTI assembly code in a file named testisrs.reti. The code includes comments and various instructions like IVTE, STO, LOA, and INT. Several syntax errors are highlighted with red squiggly lines: 'IVTE 2' (error), 'STO' (error), 'LOA' (error), 'STOIN' (error), 'INT 0' (error), and 'JUMP<= ACC' (error). A tooltip for the 'STORE' instruction is open, providing documentation: 'Stores the value of the specified register into the i-th memory cell.' The code also contains comments explaining variable storage and initialization.

```
testisrs.reti
1 ; Initialization
2 IVTE 2    IVTE command is only intended for use in isr files.
3 STO        ; Store n in M<0>  Unknown instruction "STO".
4 LOA ↗ STORE
5 STO ↗ STOREIN
6 · F(n-1) will be stored in M<2>, since its 0 no initializati
7          ; orred in M<3>
8 INT
9 Usage: INT i
10 Result: PC := IVT[i]      ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= ACC    Invalid operand.
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
```

Fig 7: Screenshot from VS Code highlighting LSP Features

## Features:

# Results

## Language Server

A screenshot of a VS Code editor window displaying assembly code for the ReTI architecture. The file is named 'testisrs.reti'. The code includes comments and various instructions like IVTE, STO, LOA, and JUMP. Several syntax errors are highlighted with red squiggly lines: 'IVTE 2' (error), 'STO' (error), 'LOA' (warning), 'STO' (warning), 'JUMP<= ACC' (error), and 'LOAD IN1 2' (warning). A tooltip for the 'STORE' instruction is open, providing documentation: 'Stores the value of the specified register into the i-th memory cell.' The code also shows usage examples for INT and IN1 registers.

```
testisrs.reti
1 ; Initialization
2 IVTE 2    IVTE command is only intended for use in isr files.
3 STO        ; Store n in M<0>  Unknown instruction "STO".
4 LOA ↗ STORE
5 STO ↗ STOREIN
6 · F(n-1) will be stored in M<2>, since its 0 no intializati
7 INT          ; F(n) will be stored in M<3>
8 Usage: INT i
9
10 Result: PC := IVT[i] ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= ACC  Invalid operand.
15 LOAD IN1 2    ; LOAD F(n-1) into IN1
16 LOAD ACC 3    ; LOAD F(n) into IN1
```

Fig 7: Screenshot from VS Code highlighting LSP Features

### Features:

- Syntax Highlighting

# Results

## Language Server

A screenshot of a VS Code editor window displaying assembly code for the ReTI architecture. The file is named 'testisrs.reti'. The code includes comments and several instructions. A tooltip is open over the 'STORE' instruction at line 4, providing documentation: 'Stores the value of the specified register into the i-th memory cell.' Other parts of the code show syntax highlighting for labels like 'INT' and 'ACC' and errors for invalid instructions like 'IVTE' and 'JUMP<='.

```
1 ; Initialization
2 IVTE 2    IVTE command is only intended for use in isr files.
3 STO      ; Store n in M<0>  Unknown instruction "STO".
4 LOA ↗ STORE
5 STO ↗ STOREIN
6 · F(n-1) will be stored in M<2>, since its 0 no intializati
7          ; orred in M<3>
8 INT
9 Usage: INT i
10 Result: PC := IVT[i]      ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= ACC    Invalid operand.
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
```

Fig 7: Screenshot from VS Code highlighting LSP Features

### Features:

- Syntax Highlighting
- Autocomplete suggestions

# Results

## Language Server

A screenshot of a VS Code editor window displaying assembly code for the ReTI architecture. The file is named 'testisrs.reti'. The code includes various instructions like IVTE, STO, LOA, STORE, INT, JUMP<=, LOAD, and ADD. Several syntax errors are highlighted in red, such as 'IVTE 2' (warning: IVTE command is only intended for use in isr files), 'STO' (Unknown instruction "STO"), 'JUMP<= ACC' (Invalid operand), and 'LOAD IN1 2' (warning: LOAD F(n-1) into IN1). A tooltip for the 'STORE' instruction is open, providing documentation: 'Stores the value of the specified register into the i-th memory cell.' Another tooltip for 'INT' is partially visible.

```
testisrs.reti
1 ; Initialization
2 IVTE 2    IVTE command is only intended for use in isr files.
3 STO        ; Store n in M<0>  Unknown instruction "STO".
4 LOA ↗ STORE
5 STO ↗ STOREIN
6 · F(n-1) will be stored in M<2>, since its 0 no initializati
7 INT          ; F(n) will be stored in M<3>
8 Usage: INT i
9
10 Result: PC := IVT[i]      ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= ACC    Invalid operand.
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
```

Fig 7: Screenshot from VS Code highlighting LSP Features

### Features:

- Syntax Highlighting
- Autocomplete suggestions
- Tooltips (syntax and documentation)

# Results

## Language Server

A screenshot of a VS Code editor window displaying assembly code for a ReTI processor. The code is contained in a file named 'testisrs.reti'. Several syntax errors are highlighted with red squiggly lines:

- Line 2: 'IVTE 2' is highlighted with an error message: 'IVTE command is only intended for use in isr files.'
- Line 3: 'STO' is highlighted with an error message: 'Unknown instruction "STO".'
- Line 4: 'LOA' is highlighted with an error message: 'STORE S i'.
- Line 5: 'STO' is highlighted with an error message: 'STOREIN'.
- Line 14: 'JUMP<= ACC' is highlighted with an error message: 'Invalid operand.'

The code itself includes comments and instructions like 'INT', 'LOAD', and 'STORE'. A tooltip for the 'STORE' instruction is open, providing documentation: 'Stores the value of the specified register into the i-th memory cell.' The tooltip also shows a preview of the instruction's assembly representation: 'INT 0'.

```
1 ; Initialization
2 IVTE 2    IVTE command is only intended for use in isr files.
3 STO      ; Store n in M<0>  Unknown instruction "STO".
4 LOA ↗ STORE
5 STO ↗ STOREIN
6 · F(n-1) will be stored in M<2>, since its 0 no intializati
7 INT          ; F(n-1) will be stored in M<3>
8 Usage: INT i
9
10 Result: PC := IVT[i] ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= ACC  Invalid operand.
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
```

Fig 7: Screenshot from VS Code highlighting LSP Features

### Features:

- Syntax Highlighting
- Autocomplete suggestions
- Tooltips (syntax and documentation)
- Realtime compilation and checking

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for a ReTI assembly program. The assembly code is as follows:

```
os > fibonacci_os.reti
1  ; Initialization
2  LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3  STORE ACC 0     ; Store n in M<0>
4  LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5  STORE ACC 1     ; Store i in M<1>
; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
; F(n) will be stored in M<3>
6  LOADI ACC 1
7  STORE ACC 3
8  LOADI IN2 3      ; IN2 will save the offset for the used variables
9
10 ; LOOP
11
12 ; JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
13  LOAD IN1 2      ; LOAD F(n-1) into IN1
14  LOAD ACC 3      ; LOAD F(n) into IN1
15  STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
16  ADD IN1 3        ; calculate F(n+1)
17  MOVE IN1 ACC
18
19  STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20  ADDI ACC 1        ; increase i
21  STORE ACC 1      ; STORE new value of i in M<1>
22 ; Check if i < n for JUMP condition
23  LOAD ACC 0
24  SUB ACC 1        ; ACC = n - M<1> = n - i, will be positive as long as n > i
25  JUMP -11         ; JUMP back to the start of the loop
26  NOP
27  JUMP 0
```

The memory view shows a table with two rows:

Address	Value	Write
1	10	Write

The variables pane shows the following register values:

- PC = 2
- IN1 = 0
- IN2 = 0
- ACC = 5
- SP = 2147483703
- BAF = 2147483650
- CS = 2147483651
- DS = 2147483672
- I = 2248146944

Fig 8: Screenshot of running ReTI-Debug session

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for a ReTI assembly program. On the left, there's a 'MEMORY VIEW' panel with two tables: 'Read' and 'Variables'. The 'Read' table shows address 1 with value 10 and write permission 'Write'. The 'Variables' table shows registers PC=2, IN1=0, IN2=0, ACC=5, SP=2147483703, BAF=2147483650, CS=2147483651, DS=2147483672, and I=2248146944. The main area displays assembly code:

```
os > fibonacci_os.reti
1 v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0     ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1     ; Store i in M<1>
; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
; F(n) will be stored in M<3>
6    LOADI ACC 1
7    STORE ACC 3
8    LOADI IN2 3      ; IN2 will save the offset for the used variables
9
10   ; LOOP
11
12   ; JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
13   v JUMP<= 12
14   LOAD IN1 2      ; LOAD F(n-1) into IN1
15   LOAD ACC 3      ; LOAD F(n) into IN1
16   STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17   ADD IN1 3      ; calculate F(n+1)
18   MOVE IN1 ACC
19   STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20   ADDI ACC 1      ; increase i
21   STORE ACC 1      ; STORE new value of i in M<1>
22   ; Check if i < n for JUMP condition
23   LOAD ACC 0
24   SUB ACC 1      ; ACC = n - M<1> = n - i, will be positive as long as n > i
25   JUMP -11        ; JUMP back to the start of the loop
26   NOP
27   JUMP 0
```

Features both architectures:

Fig 8: Screenshot of running ReTI-Debug session

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for a ReTI assembly program. On the left, there's a 'MEMORY VIEW' panel with two tables: one for 'Read' operations and one for 'Write' operations. The 'Read' table has a single entry at address 1 with value 10. The 'Write' table has entries at addresses 00, 01, and 02 with values 5, 10, and 0 respectively. Below these is a 'VARIABLES' panel showing register values: PC = 2, IN1 = 0, IN2 = 0, ACC = 5, SP = 2147483703, BAF = 2147483650, CS = 2147483651, DS = 2147483672, and I = 2248146944.

The main area displays assembly code:

```
os > fibonacci_os.reti
1  v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0      ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1      ; Store i in M<1>
6    ; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
7    ; F(n) will be stored in M<3>
8    LOADI ACC 1
9    STORE ACC 3
10   LOADI IN2 3      ; IN2 will save the offset for the used variables
11
12  ; LOOP
13  v JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
14    LOAD IN1 2      ; LOAD F(n-1) into IN1
15    LOAD ACC 3      ; LOAD F(n) into IN1
16    STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17    ADD IN1 3        ; calculate F(n+1)
18    MOVE IN1 ACC      ;
19    STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20    ADDI ACC 1        ; increase i
21    STORE ACC 1      ; STORE new value of i in M<1>
22    ; Check if i < n for JUMP condition
23    LOAD ACC 0
24    SUB ACC 1        ; ACC = n - M<1> = n - i, will be positive as long as n > i
25    JUMP -11          ; JUMP back to the start of the loop
26    NOP
27    JUMP 0
```

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

- Reading and writing register values

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for a ReTI assembly program. On the left, the 'MEMORY VIEW' panel displays memory at address 0 and 3. At address 0, value 10 is shown with a 'Write' button. At address 3, value 0 is shown. Below this is a table for reading and writing:

Address	Value	Write
1	10	Write

On the right, the assembly code for the 'fibonacci\_os.ret' file is shown:

```
os > fibonacci_os.ret
1 v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0     ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1     ; Store i in M<1>
; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
; F(n) will be stored in M<3>
6    LOADI ACC 1
7    STORE ACC 3
8    LOADI IN2 3      ; IN2 will save the offset for the used variables
9
10   ; LOOP
11
12   ; JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
13   v JUMP<= 12
14   LOAD IN1 2      ; LOAD F(n-1) into IN1
15   LOAD ACC 3      ; LOAD F(n) into IN1
16   STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17   ADD IN1 3      ; calculate F(n+1)
18   MOVE IN1 ACC
19   STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20   ADDI ACC 1      ; increase i
21   STORE ACC 1      ; STORE new value of i in M<1>
22   ; Check if i < n for JUMP condition
23   LOAD ACC 0
24   SUB ACC 1      ; ACC = n - M<1> = n - i, will be positive as long as n > i
25   JUMP -11        ; JUMP back to the start of the loop
26   NOP
27   JUMP 0
```

The 'VARIABLES' panel shows register values:

- Registers:
  - PC = 2
  - IN1 = 0
  - IN2 = 0
  - ACC = 5
  - SP = 2147483703
  - BAF = 2147483650
  - CS = 2147483651
  - DS = 2147483672
  - I = 2248146944

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

- Reading and writing register values
- Reading and writing memory

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for a ReTI assembly program. On the left, the 'MEMORY VIEW' panel displays memory at address 0, showing a value of 10 at address 1, which is being written. Below it, the 'VARIABLES' panel shows register values: PC = 2, IN1 = 0, IN2 = 0, ACC = 5, SP = 2147483703, BAF = 2147483650, CS = 2147483651, DS = 2147483672, and I = 2248146944. The main area shows the assembly code for calculating Fibonacci numbers:

```
os > fibonacci_os.reti
1 v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0      ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1      ; Store i in M<1>
; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
; F(n) will be stored in M<3>
6    LOADI ACC 1
7    STORE ACC 3
8    LOADI IN2 3      ; IN2 will save the offset for the used variables
9
10   ; LOOP
11
12   ; JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
13   v JUMP<= 12
14     LOAD IN1 2      ; LOAD F(n-1) into IN1
15     LOAD ACC 3      ; LOAD F(n) into IN1
16     STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17     ADD IN1 3      ; calculate F(n+1)
18     MOVE IN1 ACC
19     STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20     ADDI ACC 1      ; increase i
21     STORE ACC 1      ; STORE new value of i in M<1>
22     ; Check if i < n for JUMP condition
23     LOAD ACC 0
24     SUB ACC 1      ; ACC = n - M<1> = n - i, will be positive as long as n > i
25     JUMP -11        ; JUMP back to the start of the loop
26     NOP
27     JUMP 0
```

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

- Reading and writing register values
- Reading and writing memory
- Breakpoints

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for the ReTI-II OS. On the left, the 'MEMORY VIEW' panel displays memory at address 0 and 3. At address 1, there is a 'Read' operation with value 10 and a 'Write' button. Below it, a table shows memory at addresses 00, 01, and 02 with values 5, 10, and 0 respectively. The 'VARIABLES' panel lists registers and memory locations with their current values.

On the right, the assembly code for the fibonacci\_os.reti program is shown:

```
os > fibonacci_os.reti
1  v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0      ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1      ; Store i in M<1>
6    ; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
7    ; F(n) will be stored in M<3>
8    LOADI ACC 1
9    STORE ACC 3
10   LOADI IN2 3      ; IN2 will save the offset for the used variables
11
12  ; LOOP
13  v JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
14    LOAD IN1 2      ; LOAD F(n-1) into IN1
15    LOAD ACC 3      ; LOAD F(n) into IN1
16    STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17    ADD IN1 3        ; calculate F(n+1)
18    MOVE IN1 ACC
19    STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20    ADDI ACC 1        ; increase i
21    STORE ACC 1      ; STORE new value of i in M<1>
22    ; Check if i < n for JUMP condition
23    LOAD ACC 0
24    SUB ACC 1        ; ACC = n - M<1> = n - i, will be positive as long as n > i
25    JUMP -11          ; JUMP back to the start of the loop
26    NOP
27    JUMP 0
```

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

- Reading and writing register values
- Reading and writing memory
- Breakpoints

### Features for the ReTI-II (OS):

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for the ReTI-II OS. On the left, the 'MEMORY VIEW' panel displays memory at address 0 and 3. At address 1, there is a 'Read' operation with value 10 and a 'Write' button. The 'VARIABLES' panel shows register values: PC = 2, IN1 = 0, IN2 = 0, ACC = 5, SP = 2147483703, BAF = 2147483650, CS = 2147483651, DS = 2147483672, and I = 2248146944. The main area shows assembly code for the fibonacci\_os.ret file:

```
os > fibonacci_os.ret
1 v ; Initialization
2    LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3    STORE ACC 0      ; Store n in M<0>
4    LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5    STORE ACC 1      ; Store i in M<1>
6    ; F(n-1) will be stored in M<2>, since its 0 no initialization is needed
7    ; F(n) will be stored in M<3>
8    LOADI ACC 1
9    STORE ACC 3
10   LOADI IN2 3      ; IN2 will save the offset for the used variables
11
12  ; LOOP
13  v JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
14    LOAD IN1 2      ; LOAD F(n-1) into IN1
15    LOAD ACC 3      ; LOAD F(n) into IN1
16    STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-
17    ADD IN1 3        ; calculate F(n+1)
18    MOVE IN1 ACC
19    STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be
20    ADDI ACC 1       ; increase i
21    STORE ACC 1      ; STORE new value of i in M<1>
22    ; Check if i < n for JUMP condition
23    LOAD ACC 0
24    SUB ACC 1        ; ACC = n - M<1> = n - i, will be positive as long as n > i
25    JUMP -11         ; JUMP back to the start of the loop
26    NOP
27    JUMP 0
```

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

- Reading and writing register values
- Reading and writing memory
- Breakpoints

### Features for the ReTI-II (OS):

- Switching between main program and interrupt service routine file

# Results

## Debugger and Memory View

The screenshot shows a debugger interface for the ReTI-II OS. On the left, there's a 'MEMORY VIEW' panel with two tables: 'Read' and 'Variables'. The 'Read' table has one entry at address 1 with value 10 and 'Write' status. The 'Variables' table lists registers and memory locations with their current values: PC = 2, IN1 = 0, IN2 = 0, ACC = 5, SP = 2147483703, BAF = 2147483650, CS = 2147483651, DS = 2147483672, and I = 2248146944. The main area displays assembly code for a Fibonacci calculation. The code includes initialization steps, a loop structure with a JUMP condition, and various arithmetic operations like LOADI, STORE, ADDI, and SUB. A specific line of code, 'LOADI ACC 2 ; Initiate i = 2 since F(0) = 0, F(1) = 1', is highlighted in yellow. The assembly code is as follows:

```
os > fibonacci_os.reti
1 v ; Initialization
2 LOADI ACC 5 ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0 ; Store n in M<0>
4 LOADI ACC 2 ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1 ; Store i in M<1>
; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
; F(n) will be stored in M<3>
6 LOADI ACC 1
7 STORE ACC 3
8 LOADI IN2 3 ; IN2 will save the offset for the used variables
9
10 ; LOOP
11
12 ; JUMP<= 12 ; Skip over the loop if n - i <= 0, meaning i > 0
13 v JUMP<= 12
14 LOAD IN1 2 ; LOAD F(n-1) into IN1
15 LOAD ACC 3 ; LOAD F(n) into IN1
16 STORE ACC 2 ; store F(n) in M<2> since in the next iteration it is F(n)
17 ADD IN1 3 ; calculate F(n+1)
18 MOVE IN1 ACC ;
19 STORE ACC 3 ; M<3> now holds F(n+1) which in the next iteration will be
20 ADDI ACC 1 ; increase i
21 STORE ACC 1 ; STORE new value of i in M<1>
22 ; Check if i < n for JUMP condition
23 LOAD ACC 0
24 SUB ACC 1 ; ACC = n - M<1> = n - i, will be positive as long as n > i
25 JUMP -11 ; JUMP back to the start of the loop
26 NOP
27 JUMP 0
```

Fig 8: Screenshot of running ReTI-Debug session

### Features both architectures:

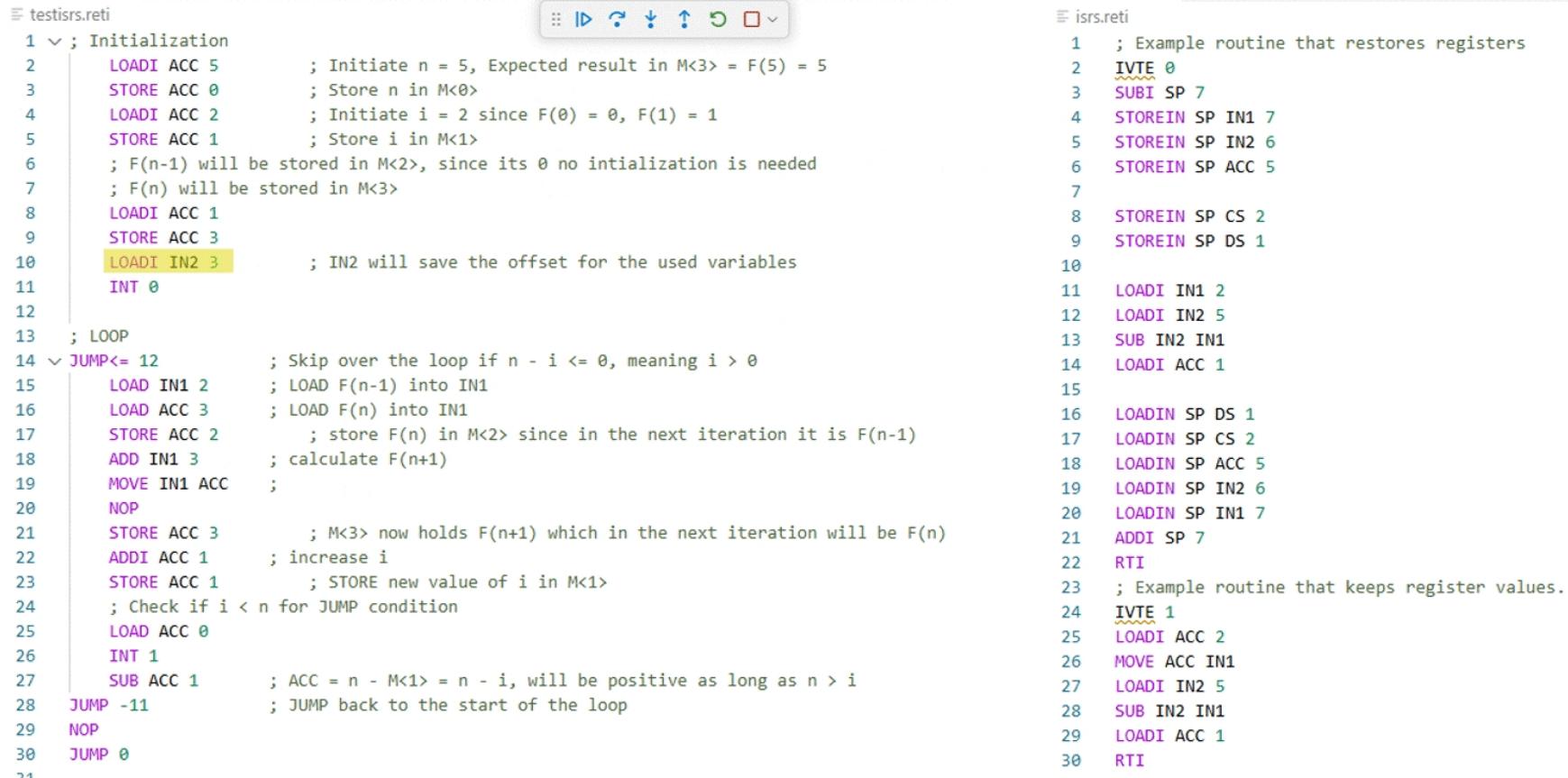
- Reading and writing register values
- Reading and writing memory
- Breakpoints

### Features for the ReTI-II (OS):

- Switching between main program and interrupt service routine file
- Updated Stepping Logic to support interrupts

# Results

## Debugger, Stepping



The image shows two side-by-side windows of a debugger interface, likely ReTI-Debug, displaying assembly code. The left window is titled 'testisrs.ret' and the right window is titled 'isrs.ret'. Both windows show a list of assembly instructions with line numbers and comments.

**testisrs.ret:**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

**isrs.ret:**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping

testisrs.reti

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3      ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3       ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1      ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1      ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11        ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```



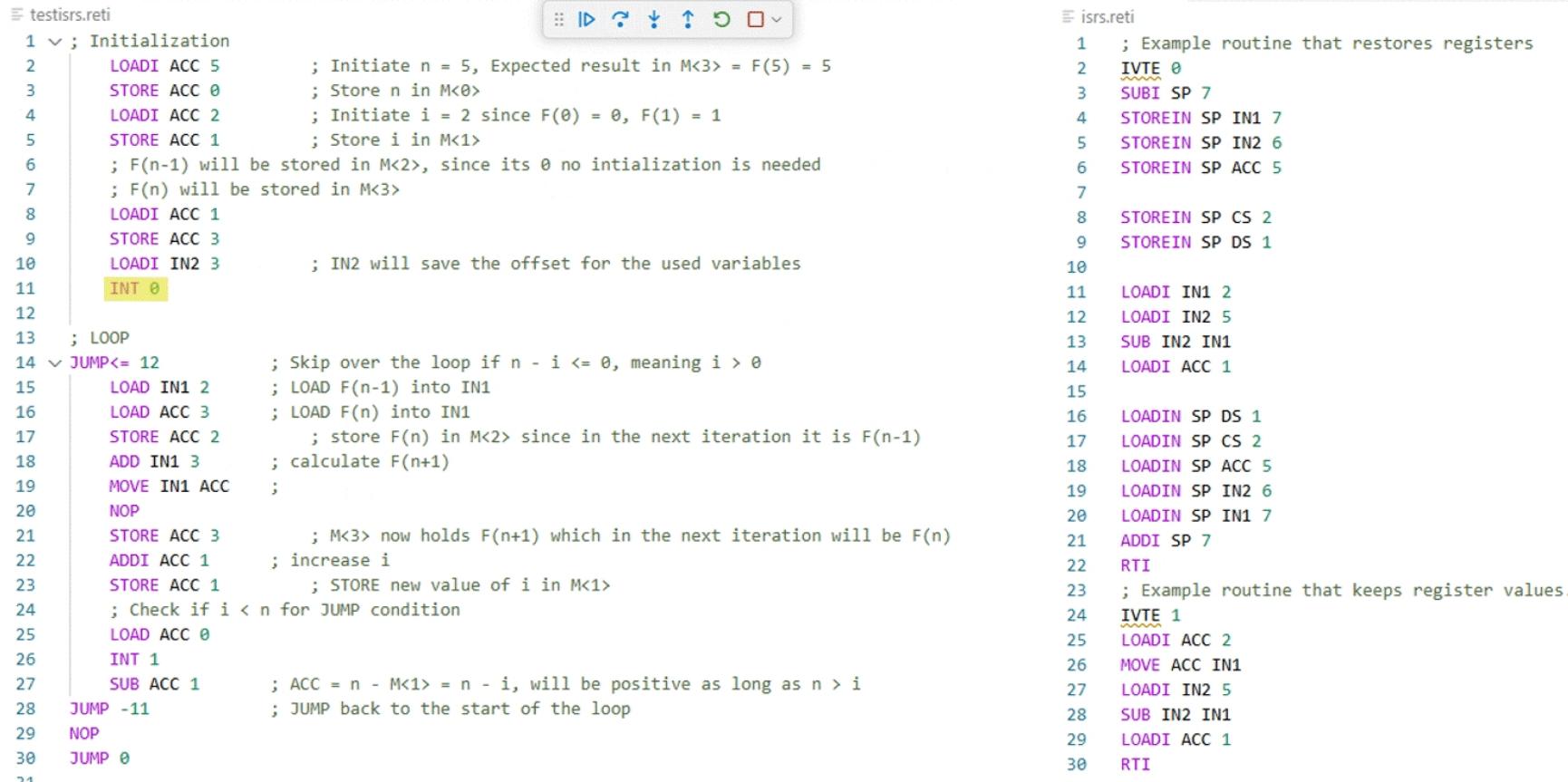
isrs.reti

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side windows of a debugger interface, likely ReTI-Debug, displaying assembly code. Both windows have a toolbar at the top with icons for file operations, search, and navigation.

**testisrs.ret:**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

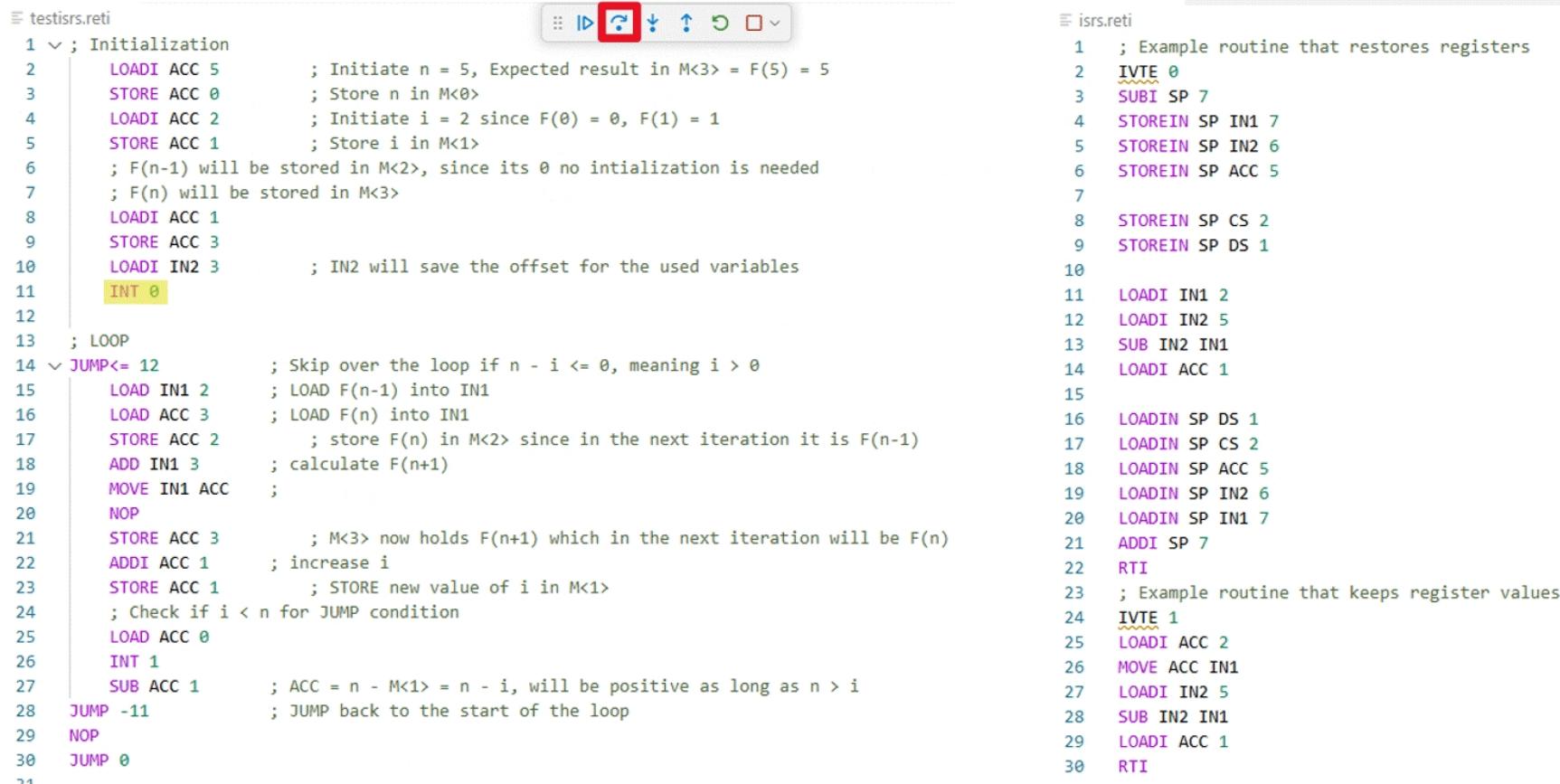
**isrs.ret:**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side screenshots of a ReTI-Debug session. On the left, the assembly code for `testisrs.ret` is displayed. On the right, the assembly code for `isrs.ret` is displayed. Both screens show the assembly code with various instructions highlighted in different colors (e.g., purple, green, blue) and some lines are underlined with a wavy line. A toolbar with several icons is visible at the top of each screen. The `testisrs.ret` code implements a loop to calculate F(n) using a temporary variable IN2. The `isrs.ret` code contains routines for saving and restoring registers.

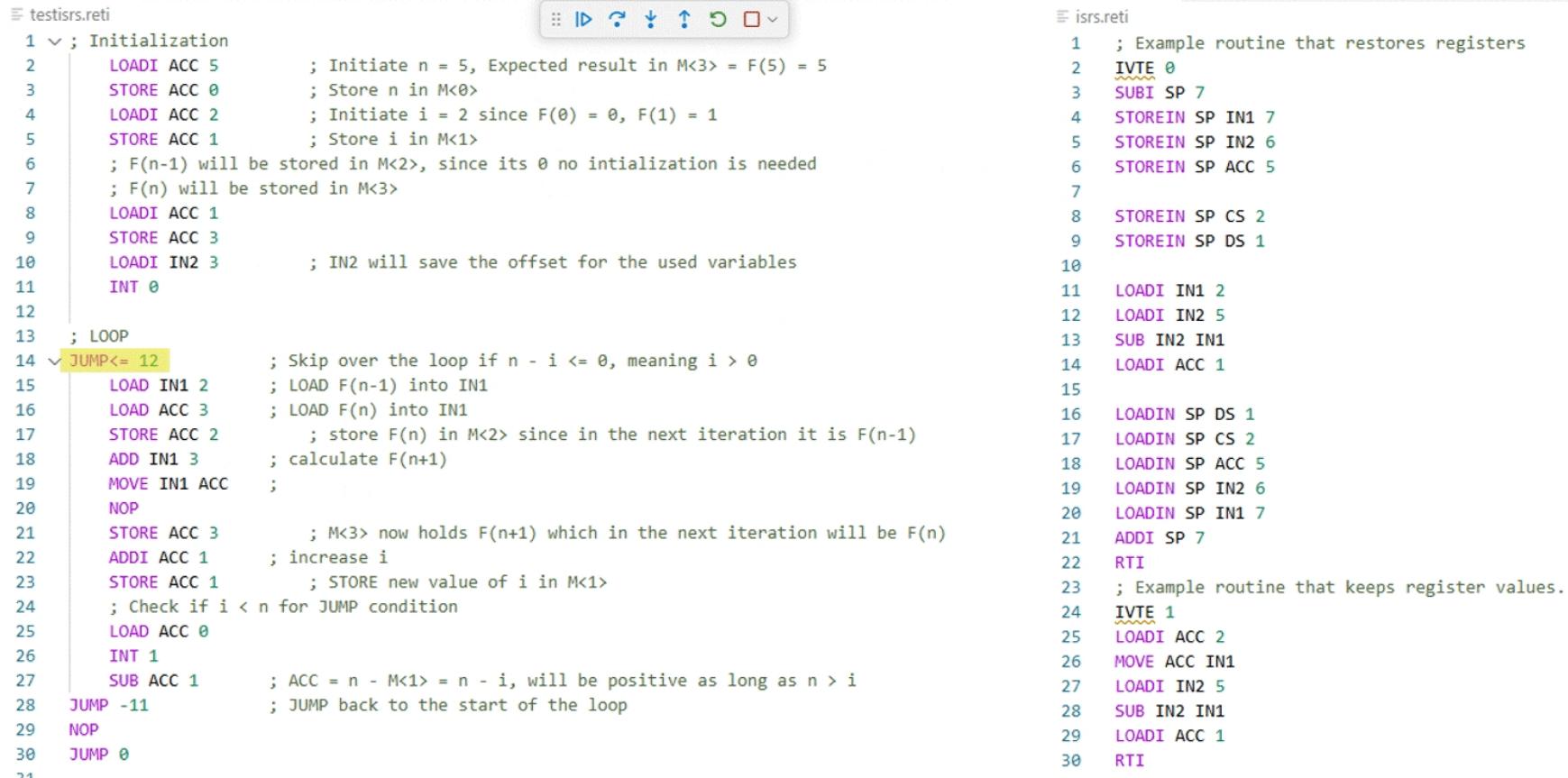
```
testisrs.ret
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0      ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1      ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1        ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0

isrs.ret
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side code editors within a debugger interface. The left editor is titled 'testisrs.ret' and the right is 'isrs.ret'. Both editors have a toolbar at the top with icons for file operations, search, and navigation.

**testisrs.ret:**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

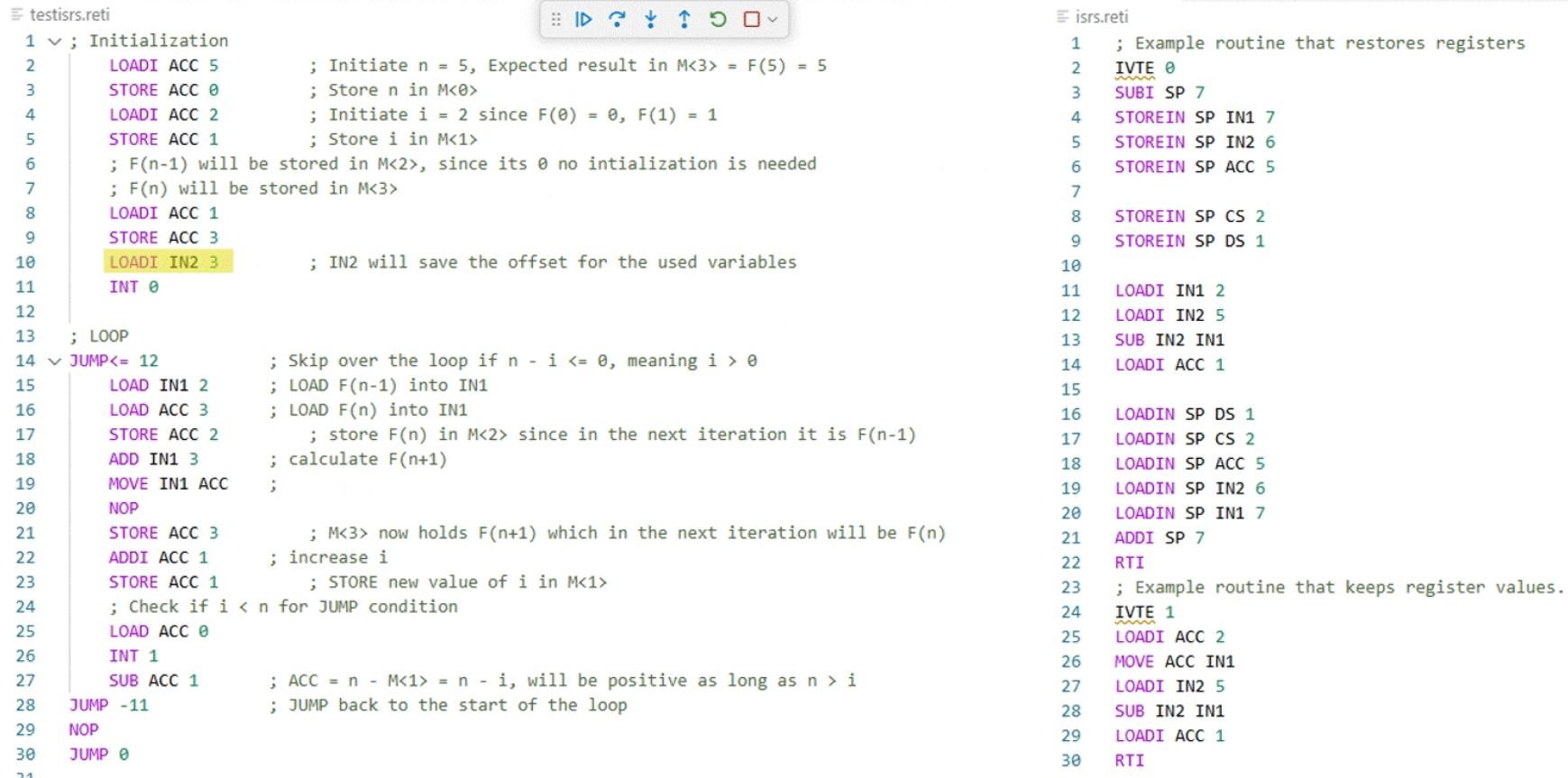
**isrs.ret:**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side code editors within a debugger interface. The left editor is titled 'testisrs.ret' and the right is 'isrs.ret'. Both editors have a toolbar at the top with icons for file operations, search, and navigation.

**testisrs.ret:**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

**isrs.ret:**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping

testisrs.reti

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3      ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12      ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2      ; LOAD F(n-1) into IN1
16 LOAD ACC 3      ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3      ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1      ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1      ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11        ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```



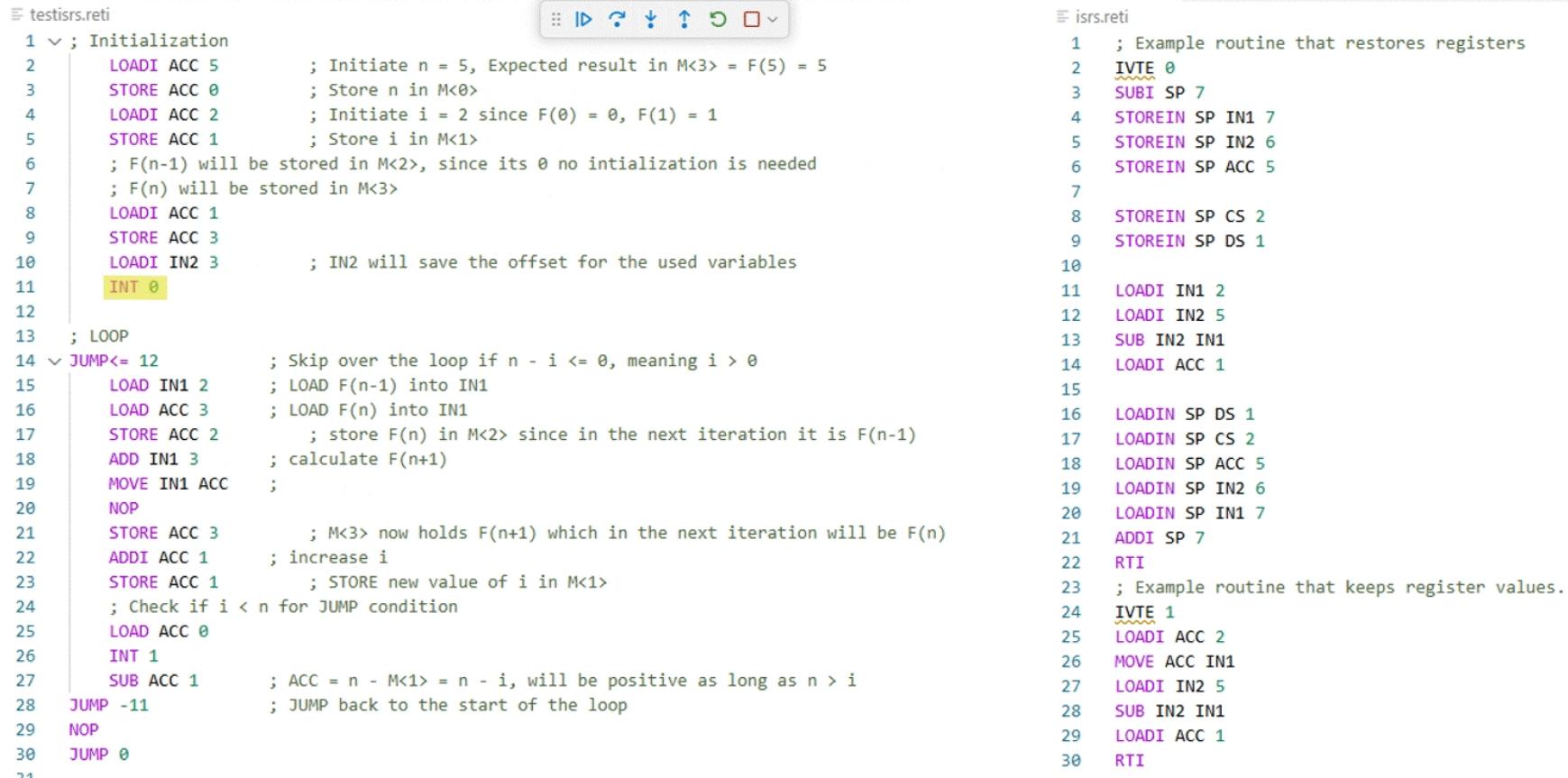
isrs.reti

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side windows of a debugger interface, likely ReTI-Debug, displaying assembly code. Both windows have a toolbar at the top with icons for file operations, search, and navigation.

**testisrs.ret**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

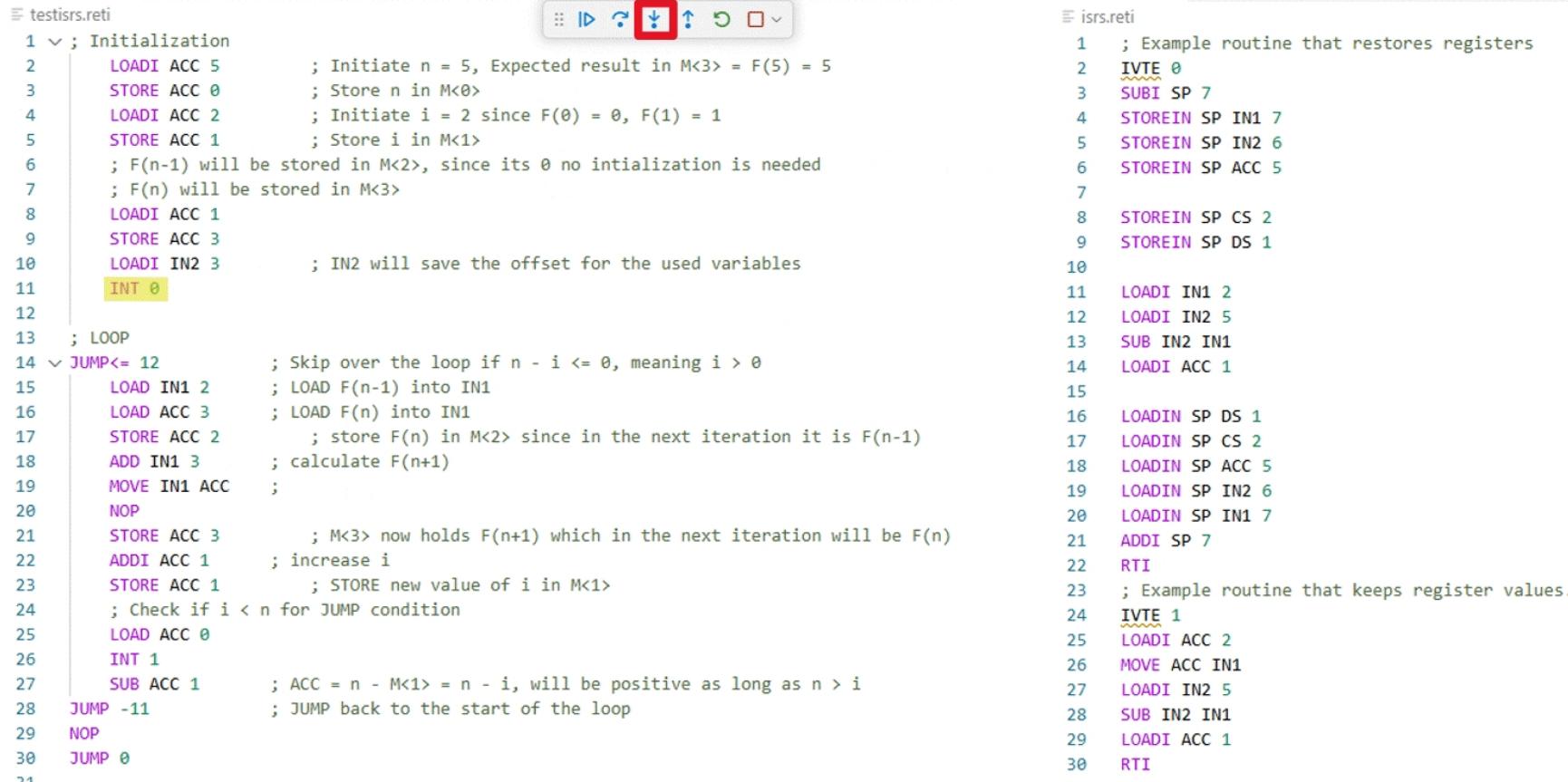
**isrs.ret**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



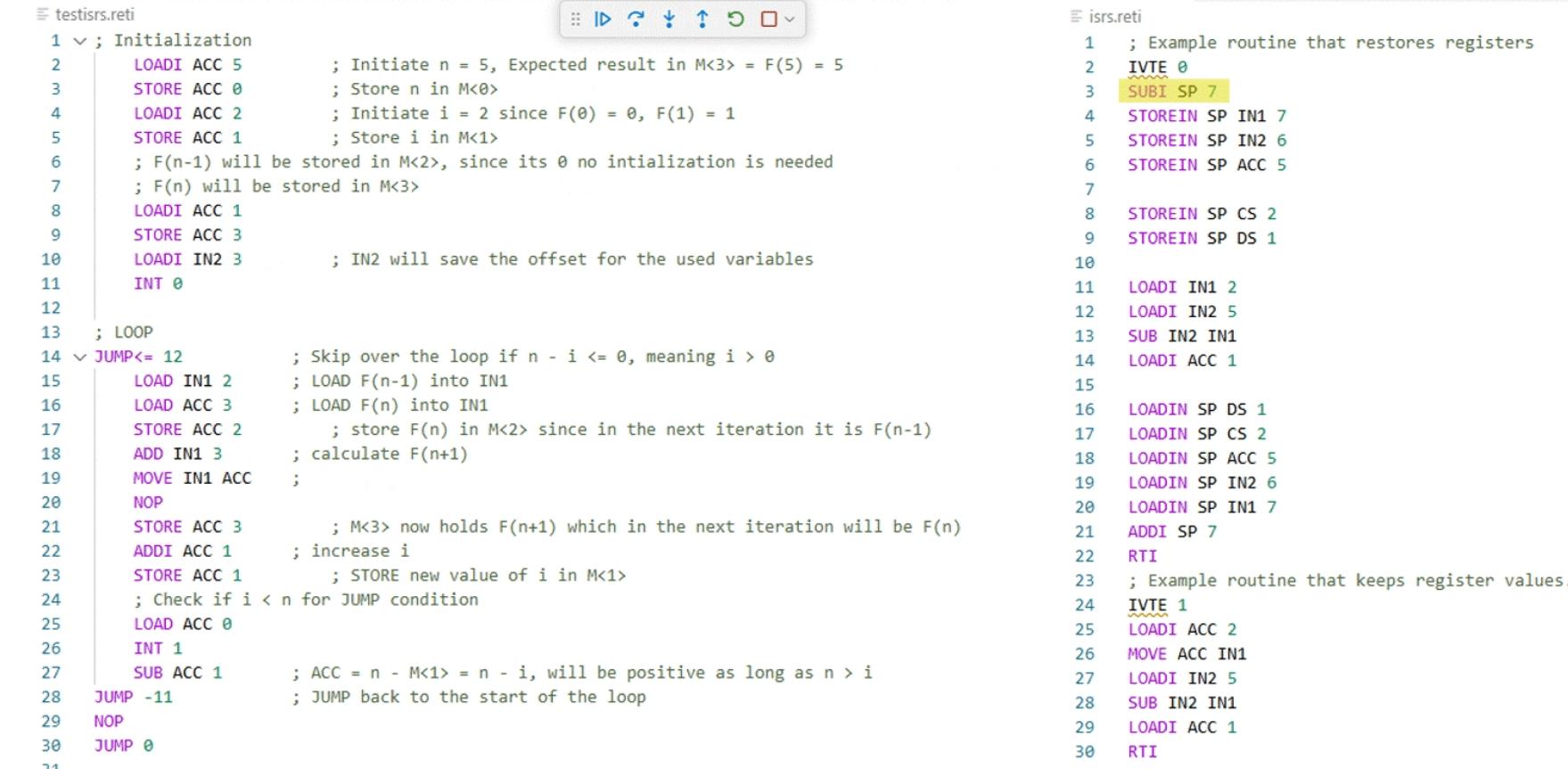
```
testisrs.reti
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0

isrs.reti
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two assembly code files in a ReTI-Debug session:

**testisrs.ret**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

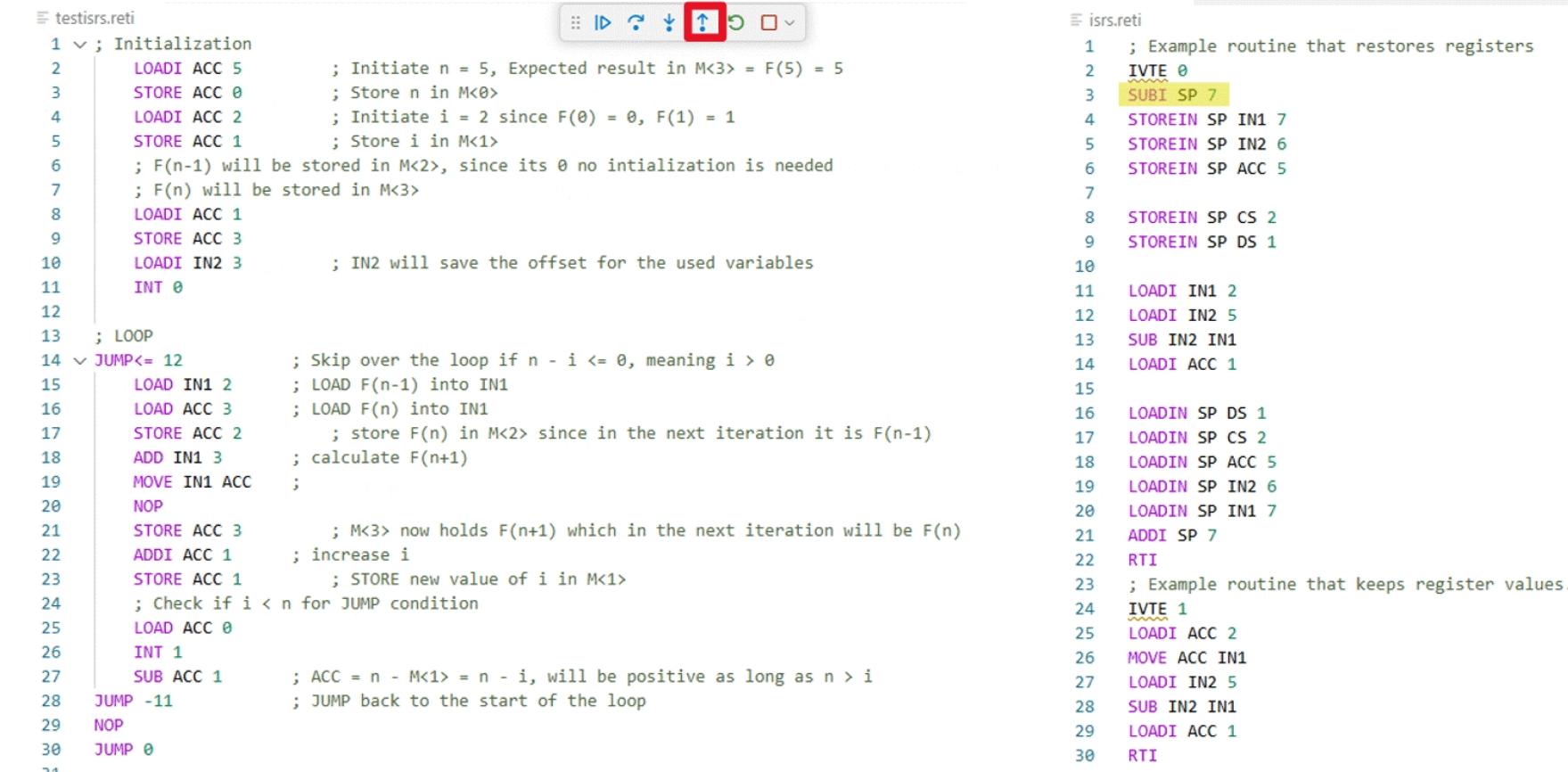
**isrs.ret**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The screenshot shows two windows of the ReTI-Debug extension in VS Code. The left window displays assembly code for a routine named 'testisrs.ret'. The right window displays assembly code for a routine named 'isrs.ret'. Both windows show the assembly code with syntax highlighting and line numbers. A toolbar at the top of the interface has several icons, with the step-up arrow icon highlighted by a red box.

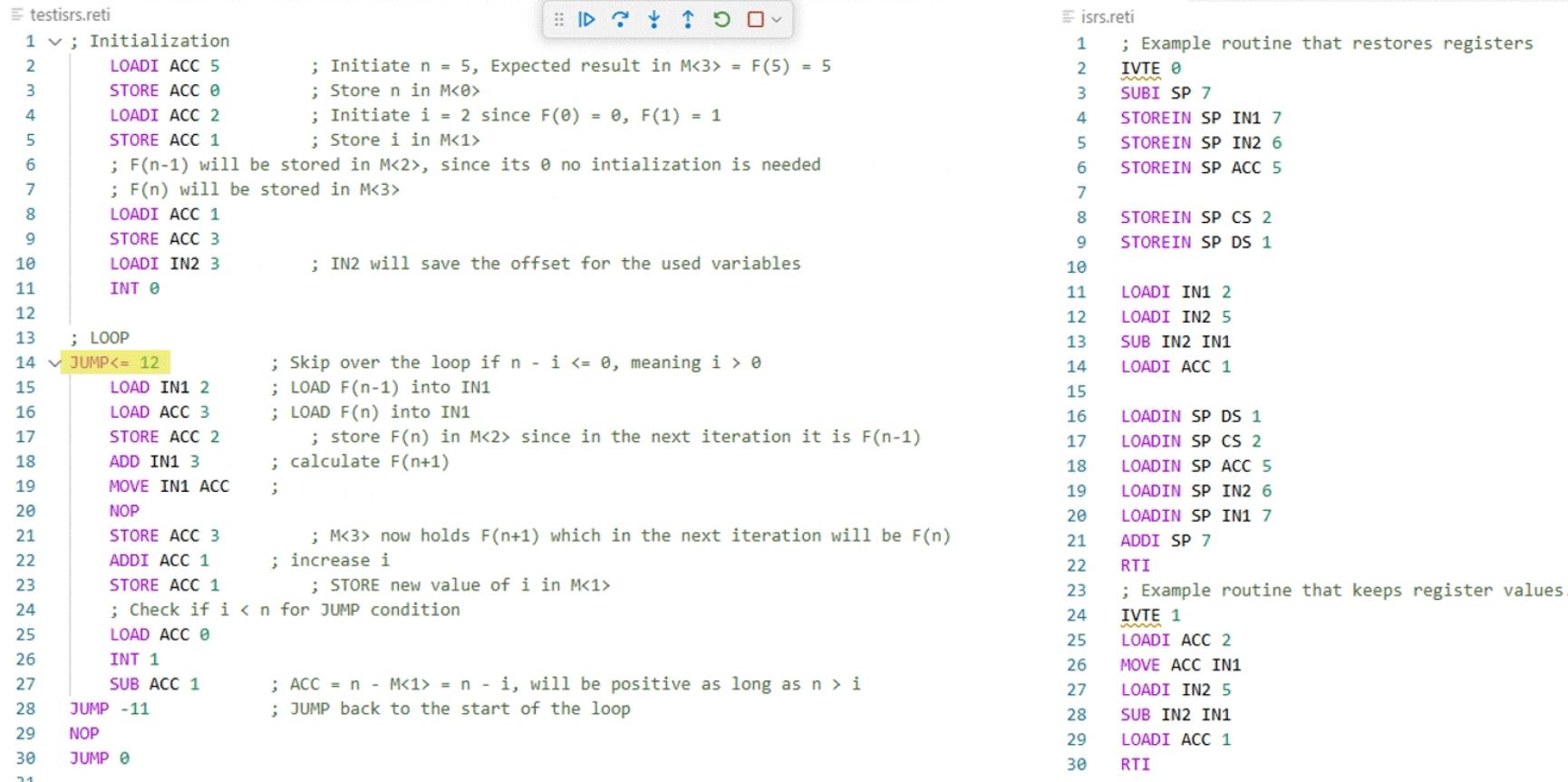
```
testisrs.ret
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0

isrs.ret
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The image shows two side-by-side screenshots of a ReTI-Debug session. Both windows have a header bar with icons for file operations, search, and navigation.

**Left Window (testisrs.ret):**

```
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

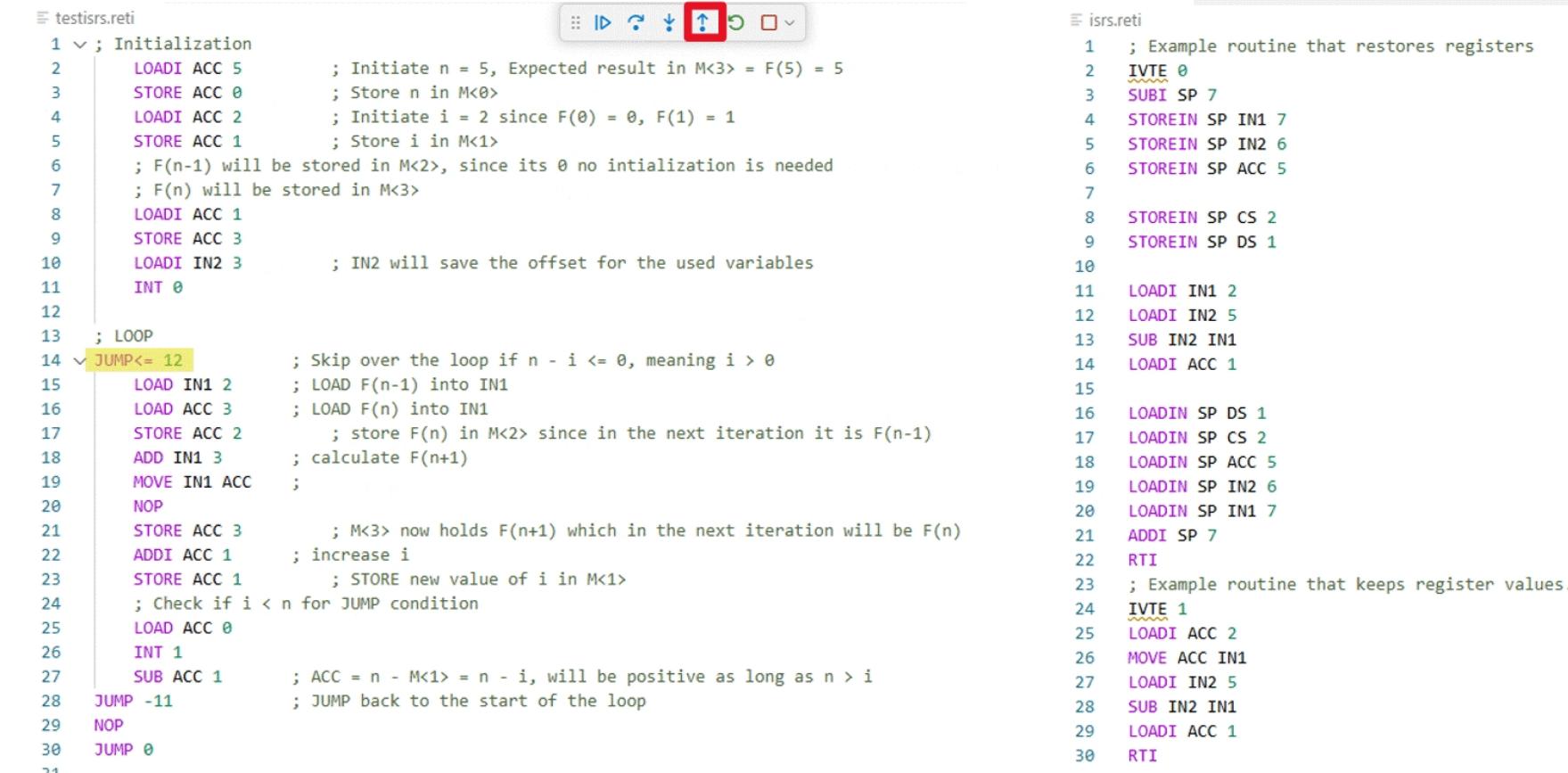
**Right Window (isrs.ret):**

```
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



The screenshot shows two assembly files in a debugger interface:

- testisrs.reti:** This file contains the main logic for calculating the nth Fibonacci number. It includes initialization code, a loop that calculates  $F(n+1)$ , and a jump condition to skip iterations where  $i > n$ . The assembly code is annotated with comments explaining its purpose.
- isrs.reti:** This file contains two routines: one for restoring registers (IVTE) and another for keeping register values (LOADIN).

The toolbar at the top features several icons, with the step-over icon (represented by an upward arrow) highlighted with a red box.

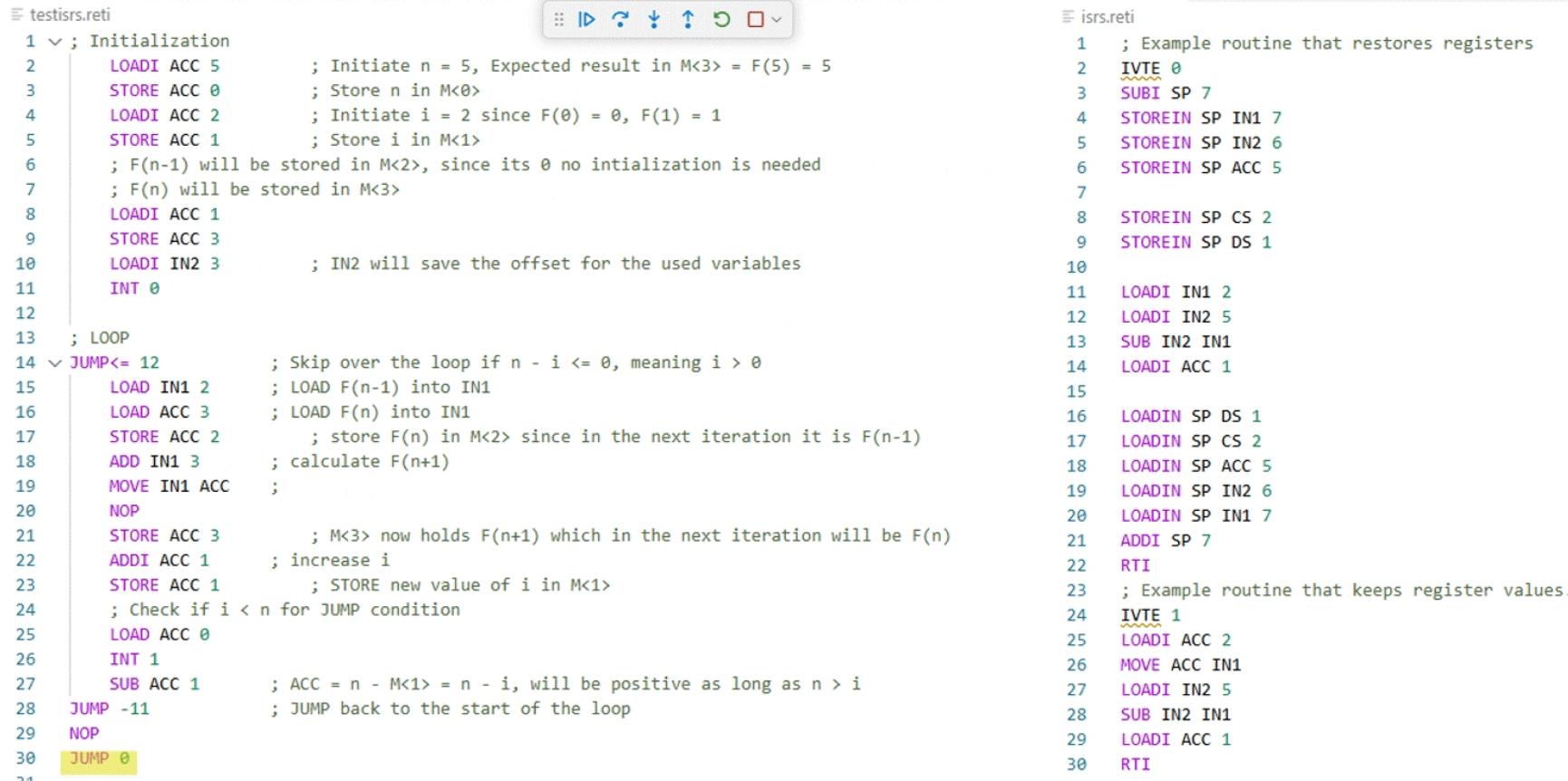
```
testisrs.reti
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0      ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1      ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > n
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0

isrs.reti
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Results

## Debugger, Stepping



```
testisrs.ret
1 ; Initialization
2 LOADI ACC 5      ; Initiate n = 5, Expected result in M<3> = F(5) = 5
3 STORE ACC 0     ; Store n in M<0>
4 LOADI ACC 2      ; Initiate i = 2 since F(0) = 0, F(1) = 1
5 STORE ACC 1     ; Store i in M<1>
6 ; F(n-1) will be stored in M<2>, since its 0 no intialization is needed
7 ; F(n) will be stored in M<3>
8 LOADI ACC 1
9 STORE ACC 3
10 LOADI IN2 3     ; IN2 will save the offset for the used variables
11 INT 0
12
13 ; LOOP
14 JUMP<= 12       ; Skip over the loop if n - i <= 0, meaning i > 0
15 LOAD IN1 2       ; LOAD F(n-1) into IN1
16 LOAD ACC 3       ; LOAD F(n) into IN1
17 STORE ACC 2      ; store F(n) in M<2> since in the next iteration it is F(n-1)
18 ADD IN1 3        ; calculate F(n+1)
19 MOVE IN1 ACC
20 NOP
21 STORE ACC 3      ; M<3> now holds F(n+1) which in the next iteration will be F(n)
22 ADDI ACC 1       ; increase i
23 STORE ACC 1      ; STORE new value of i in M<1>
24 ; Check if i < n for JUMP condition
25 LOAD ACC 0
26 INT 1
27 SUB ACC 1       ; ACC = n - M<1> = n - i, will be positive as long as n > i
28 JUMP -11         ; JUMP back to the start of the loop
29 NOP
30 JUMP 0
```

```
isrs.ret
1 ; Example routine that restores registers
2 IVTE 0
3 SUBI SP 7
4 STOREIN SP IN1 7
5 STOREIN SP IN2 6
6 STOREIN SP ACC 5
7
8 STOREIN SP CS 2
9 STOREIN SP DS 1
10
11 LOADI IN1 2
12 LOADI IN2 5
13 SUB IN2 IN1
14 LOADI ACC 1
15
16 LOADIN SP DS 1
17 LOADIN SP CS 2
18 LOADIN SP ACC 5
19 LOADIN SP IN2 6
20 LOADIN SP IN1 7
21 ADDI SP 7
22 RTI
23 ; Example routine that keeps register values.
24 IVTE 1
25 LOADI ACC 2
26 MOVE ACC IN1
27 LOADI IN2 5
28 SUB IN2 IN1
29 LOADI ACC 1
30 RTI
```

Fig. 9: Screenshots of running ReTI-Debug session

# Conclusion

## Goals Reached

# Conclusion

## Goals Reached

- All features (emulator, debugger, language server) extended to cover both versions ✓

# Conclusion

## Goals Reached

- All features (emulator, debugger, language server) extended to cover both versions ✓
- Memory view allows easier monitoring and manipulation of memory ✓

# Conclusion

## Goals Reached

- All features (emulator, debugger, language server) extended to cover both versions ✓
- Memory view allows easier monitoring and manipulation of memory ✓
- Debugger and Language Server offer
  - Syntax Highlighting ✓
  - Realtime Compilation ✓
  - Code Completion Suggestions ✓
  - Executing Code and Inspecting/Manipulating ReTI State ✓

# Conclusion

## Goals Reached

- All features (emulator, debugger, language server) extended to cover both versions ✓
- Memory view allows easier monitoring and manipulation of memory ✓
- Debugger and Language Server offer
  - Syntax Highlighting ✓
  - Realtime Compilation ✓
  - Code Completion Suggestions ✓
  - Executing Code and Inspecting/Manipulating ReTI State ✓
- Extension provides familiar programming workflow and OS-independent use ✓

# Conclusion

## Limitations & Future Work

# Conclusion

## Limitations & Future Work

- Emulate UART interface

# Conclusion

## Limitations & Future Work

- Emulate UART interface
- Make debug adapter independent from VS Code (embedding in other editors/tools)

# Conclusion

## Limitations & Future Work

- Emulate UART interface
- Make debug adapter independent from VS Code (embedding in other editors/tools)
- Integrate Pico-C compiler

# Conclusion

## Limitations & Future Work

- Emulate UART interface
- Make debug adapter independent from VS Code (embedding in other editors/tools)
- Integrate Pico-C compiler
- Add datapaths visualization [3]:

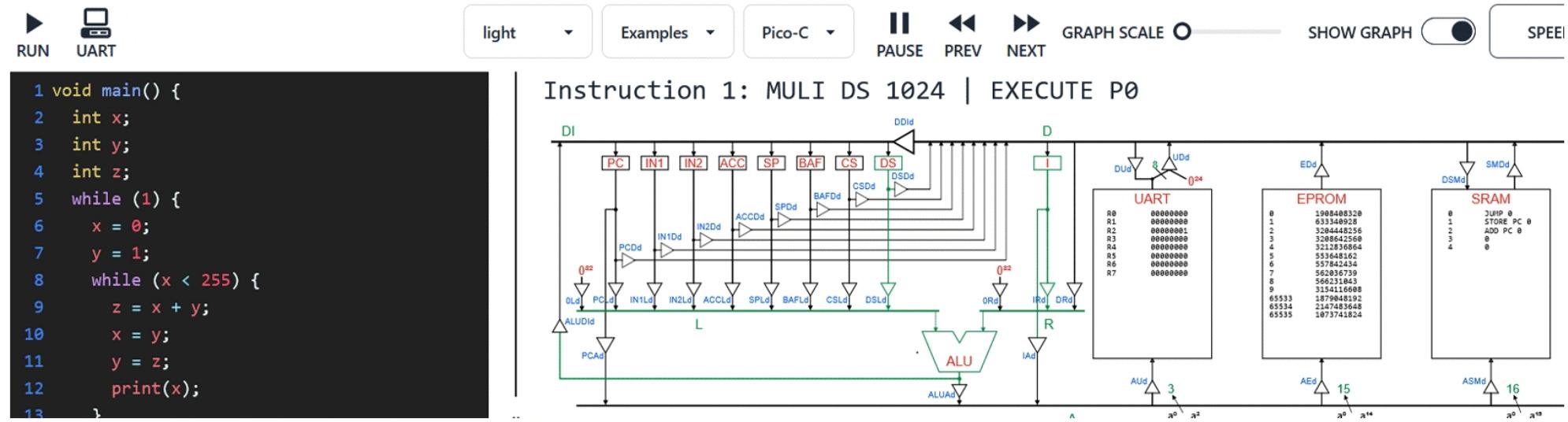


Fig. 10: Screenshot of ReTI-Emulator with Datapath-Visualization [3]

# Extra: Language-Server-Protocol

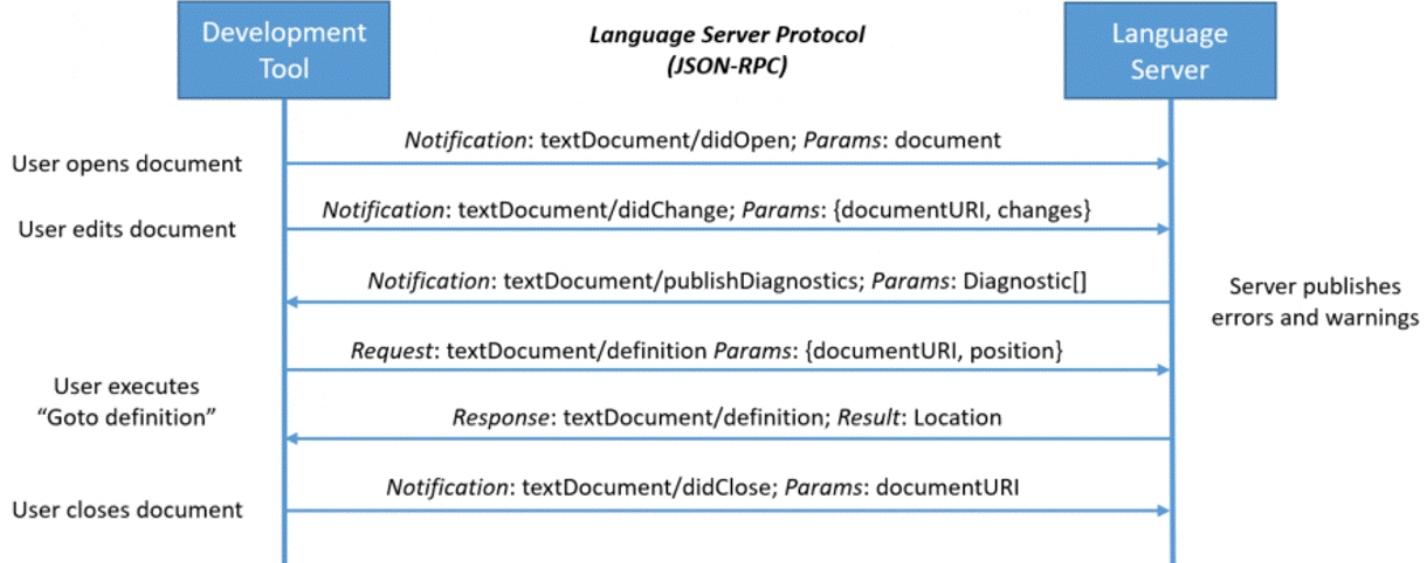


Fig. 11: Example for Communication between Tool and Server in the Language-Server-Protocol [4]

# Extra: Debug-Adapter-Protocol

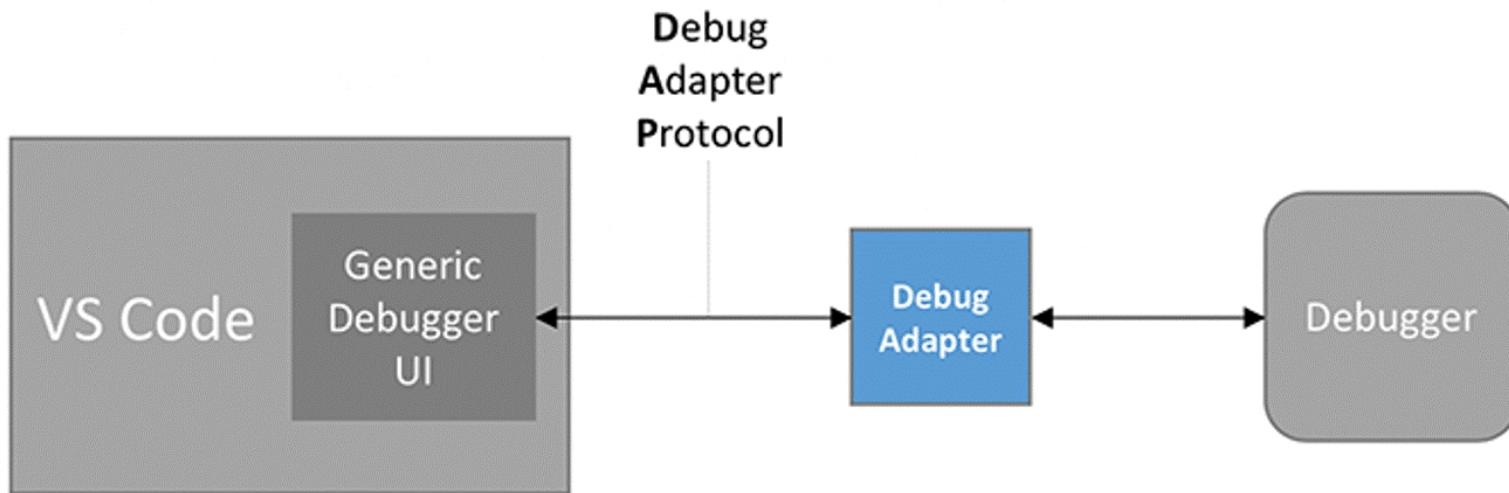


Fig. 12: VS Code Debug Architecture [5]

# References

- [1]: RETI-Emulator by Jürgen Mattheis, <https://github.com/matthejue/RETI-Emulator>
- [2]: Technische Informatik – Kapitel 2 – Kodierung, Prof. Dr. Armin Biere, University of Freiburg, SS 2024
- [3]: RETI-Emulator by Michel Giehl, [github.com/michel-giehl/Reti-Emulator](https://github.com/michel-giehl/Reti-Emulator)
- [4]: Microsoft *Language Server Protocol - Sequence Diagram*. Retrieved 13. July 2025, from <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>
- [5]: Microsoft. VS Code Debug Architecture. Retrieved 13. July 2025, from <https://code.visualstudio.com/api/extension-guides/debugger-extension>
- [6]: ReTI-Tools by Malte Pullich, <https://github.com/mlt279/vscode-reti>