

# Product Manual

LearnRisc.com

library/risc/Manual 8/11/25  
version 1.0

# Table of Contents

LearnRisc.com.....	4
About Us.....	4
Our Mission.....	6
RISC-5 Assembly.....	8
About RISC-5.....	8
Basic Instructions.....	9
Web Tool.....	11
Source Editor.....	11
Flowchart Window.....	13
Code Listing Window.....	16
Importance of an Assembly Program Listing.....	16
Example of a Simple Assembly Program Listing:.....	17
Load File Window.....	18
Example Load File Content:.....	18
IO (input /output).....	19
Output Instruction.....	19
Input Instruction.....	19
IO File Example:.....	19
Run Window.....	20
Functionality of the Run Window.....	20
Purpose and Use.....	20
HTML LED Bar for Simulated I/O Instruction Output.....	21
Purpose of the LED Bar Simulation.....	21
I/O Instruction Example.....	21
Special LED Instruction Mapping.....	22
LED Features.....	22
Pattern Change Delay.....	22
Educational Impact.....	22
Program Examples:.....	23
Fibonacci Series.....	23
RESULTS – PROGRAM OUTPUT for Fibonacci code:.....	23
FIBONACCI FLOW CHART.....	24
Fibonacci Code Listing.....	25
Fibonacci Source.....	26
Multiply Program.....	28
MULTIPLY RESULTS – PROGRAM OUTPUT.....	29
MULTIPLY FLOW CHART.....	30
Multiply Code Listing:.....	31
Multiply Code Source.....	35
AdividesB Program.....	38
AdividesB RESULTS – PROGRAM OUTPUT.....	39
AdividesB FLOW CHART.....	40
AdividesB Code Listing:.....	41
AdividesB Code Source.....	44
Primes.....	46
Programmer approach.....	46
PRIME FLOW CHART.....	47
PRIME Code Listing:.....	51

RISC-v Assembler by RWescott - Version 1.05.....	51
PRIME Code Sourcevoid main().....	59
Introduction to Global Variables and Structures.....	66
What is a Structure?.....	66
Why Use Structures?.....	66
Current Limitation.....	66
Example.....	66
Load a memory value.....	67
Store a memory value.....	67
Multi-core processors.....	69
Assembly and Multi-core Programming.....	69
Advanced Risc Coding.....	70
Introduction.....	70
Advantages of a Unix-like OS in IoT.....	70
Coding Unix Utilities in RISC-V.....	70
Learning Outcomes and Real-World Application.....	70
Conclusion.....	71
Register Convention use.....	72
1. t registers (temporary).....	72
2. s registers (saved).....	72
3. a registers (arguments / return values).....	73
4. Special registers.....	73
📌 Convention for address registers in stores.....	73
Raspberry Pi and Ubuntu.....	74
Raspberry Pi Micro SDXC Memory Card Offer.....	75

## About Us

We are about writing modern programs using AI, C++, and web technologies.

In today's evolving software development landscape, C++ stands out as the language of choice for providing the core functionality of our applications. It is the language our assembler and simulators are written in. Its versatility, performance, and deep access to system-level operations make it an invaluable tool for creating robust and efficient code. C++ remains one of the most powerful languages due to its balance of high- and low-level capabilities, allowing both object-oriented design and precise memory control.

To complement the functionality written in C++, we use HTML and Node.js to deliver the graphical user interfaces of our programs. HTML provides the structure and layout, while Node.js enables back-end capabilities and integration with web services, allowing real-time interactivity and fast, scalable responses. This separation of concerns ensures a clean, modular design where user interface and core logic evolve independently.

Our applications are flexible in deployment — they can be run locally on a Wi-Fi network or hosted as publicly accessible websites. This architecture provides users with the freedom to use the application in varied environments, whether for internal use in a closed network or for broad public access.

By design, this model naturally supports multiple users. Node.js's asynchronous, non-blocking architecture handles concurrent sessions efficiently, ensuring that every user has a smooth and isolated experience without impacting others.

Users always interact with the system through their web browser. This universal client requires no installation and works across devices and platforms, greatly simplifying access and reducing technical overhead for end users.

A key feature of our approach is that the user interface is generated by describing the required functionality to an AI system. Instead of manually coding HTML or front-end logic, developers specify what the interface should do and how it should behave. The AI then creates the required interface components rapidly.

The AI system provides nearly all of the interface code, generating modern, responsive layouts and forms that meet the user's specifications. This results in a dramatic reduction in frontend development time and helps ensure consistency across projects.

Similarly, much of the C++ back-end code is written by the AI. Developers define the procedures they need, and the AI generates implementations that often include integration with the existing support library. These routines are generally ready to use with little to no modification, accelerating development cycles.

This AI-assisted methodology enables developers to focus on system integration, architectural direction, and debugging. While AI excels at generating the functional components and modules, it is the human developer who ensures everything fits together cohesively and meets the desired goals.

This collaboration between human insight and machine speed defines the future of efficient software development.

## Our Mission

At LearnRisc.com, our mission is to make RISC-V assembly language accessible, practical, and engaging for learners of all levels. We believe the best way to understand RISC-V is through direct interaction with the language—writing real programs, experimenting with code, and seeing results instantly. Our platform is designed to turn the challenge of learning assembly into an intuitive and rewarding experience.

### 1. Learning by Doing

LearnRisc.com focuses on experiential learning. Users are encouraged to dive into the world of RISC-V by creating and modifying actual programs. Through hands-on practice, students develop a deeper understanding of how RISC-V instructions control hardware behavior.

### 2. Rich Library of Examples

We provide a vast collection of example routines that cover everything from basic arithmetic to control flow, string manipulation, memory operations, and more. These examples are fully executable and can be modified in real time, helping users immediately see the impact of their changes.

### 3. Integrated Coding Environment

Our all-in-one environment combines a code editor, assembler, and simulator. This means learners can write, compile, and run their programs in a single unified interface without the need for complex toolchains or external tools.

### 4. Simulation Support

The platform includes a built-in simulator that executes RISC-V code, displays register contents, memory access, and supports real-time debugging. It allows users to step through programs instruction-by-instruction, making the learning process clear and traceable.

### 5. C-Like Assembly Syntax

To reduce the learning curve, our assembler is designed to resemble simplified C programming. This familiar structure helps beginners transition smoothly into assembly while retaining powerful low-level control.

### 6. Console Output Utility

A specialized ``cout`` utility mimics C++ style output, enabling intuitive I/O operations. This feature allows users to print variables and messages to the console, making it easier to understand program flow and data manipulation.

### 7. Support for C-Style Comments

We support common C-style comments using ``//``, ``/*``, and ``*/``. This allows learners to annotate

and document their code as they would in a high-level language, promoting good programming habits.

## 8. Memory Access Through C Structures

Users can define and access memory layouts using C-style structures. This feature bridges the gap between assembly and higher-level programming, offering a more meaningful way to manage complex data.

## 9. Embedded Flow Charting

By embedding special control characters into comments, our platform can auto-generate flowcharts from the assembly code. This visual representation of control logic greatly enhances comprehension and is ideal for teaching algorithmic thinking.

## 10. No Linker Required

Programs on LearnRisc.com run without requiring a linker. Our simplified execution model makes it easy to focus on learning concepts without getting bogged down by toolchain complexities.

## 11. Broad Range of Code Samples

From basic math operations to intricate control systems, we provide a wide spectrum of code samples to cater to both beginners and experienced developers. Users can study, modify, and adapt these examples to fit their learning path.

## 12. Readable Register Naming

We support human-friendly register names for improved readability. This helps learners associate register functions with their purpose and reduces the confusion common in traditional assembly environments.

## 13. OS Development and Multitasking

LearnRisc.com goes beyond tutorials—our platform enables learners to explore operating system development. Topics such as task switching, interrupt handling, and memory management are supported with real code examples.

## 14. Advanced Topics: IoT Applications

For advanced users, we introduce real-world applications such as IoT (Internet of Things) systems. From sensor integration to low-power multitasking, LearnRisc.com helps users apply RISC-V to solve modern engineering challenges.

LearnRisc.com is more than a tutorial—it's a fully featured RISC-V learning ecosystem. Whether you're a student, hobbyist, or professional, our goal is to equip you with the tools and knowledge to master assembly language and unlock the full potential of the RISC-V architecture.

## RISC-5 Assembly

### About RISC-5

RISC-V is an open standard instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). Originally developed at the University of California, Berkeley, RISC-V offers a streamlined set of instructions that are easy to implement in both hardware and software. Unlike proprietary ISAs such as ARM or x86, RISC-V is freely available under open-source licenses, allowing anyone to design, modify, and produce compatible processors without licensing fees or restrictions. This has made it a popular choice for academic research, embedded systems, and even commercial products that require custom processing capabilities.

The importance of RISC-V lies in its openness, flexibility, and scalability. It empowers startups, researchers, and governments to innovate at the silicon level without being tied to costly vendor lock-ins. This enables broader participation in processor development and fosters a more competitive hardware ecosystem. As computing requirements grow across industries—from edge computing and artificial intelligence to super-computing—RISC-V's modular architecture allows implementers to add or remove features as needed, making it well-suited to a wide range of performance and power efficiency targets. As such, RISC-V is poised to become a cornerstone of the next generation of computing platforms.



## Basic Instructions

This section explains the purpose and behavior of core RISC-V instructions in the base RV32I set.

**ADD** Performs integer addition between two registers. The result is stored in the destination register. This is a basic arithmetic operation used frequently in programs.

**SUB** Subtracts the value in one register from another and stores the result in the destination register. It is essential for arithmetic operations requiring differences.

**ADDI** Adds an immediate (constant) value to a register and stores the result in the destination register. Useful for quick arithmetic without needing another register.

**LUI** Loads a 20-bit immediate value into the upper 20 bits of a register. Often used to construct large constants in combination with other instructions.

**AUIPC** Adds a 20-bit immediate value to the current program counter (PC) and stores the result in a register. Helpful for generating PC-relative addresses.

**LW** Loads a 32-bit word from memory into a register. The memory address is calculated from a base register plus an offset. Used for retrieving stored data from memory.

**SW** Stores a 32-bit word from a register into memory. The target address is computed using a base register plus an offset. Essential for saving data from registers to memory.

**LB** Loads an 8-bit byte from memory into a register, sign-extending the value. Useful for handling character or byte-level data.

**SB** Stores an 8-bit byte from a register into memory. The address is computed from a base register plus an offset. Typically used for storing characters or small data values.

**BEQ** Branches to a target address if two registers contain equal values. Commonly used for conditional execution and loops.

**BNE** Branches to a target address if two registers contain different values. Often used for skipping instructions based on comparisons.

**BLT** Branches to a target address if one register is less than another (signed comparison). Helps control program flow based on ordered values.

**BGE** Branches to a target address if one register is greater than or equal to another (signed comparison). Used for control flow in loops and conditional statements.

**JAL** Jumps to a target address and stores the return address in a register. Enables calling subroutines while keeping track of where to return.

**JALR** Jumps to an address contained in a register plus an immediate offset, storing the return address in another register. Provides flexible function calls and indirect jumps.

**AND** Performs a bitwise AND between two registers, storing the result in the destination register. Used for masking bits and logical operations.

**OR** Performs a bitwise OR between two registers, storing the result in the destination register. Used for combining bit patterns.

**XOR** Performs a bitwise XOR between two registers, storing the result in the destination register. Useful for toggling specific bits.

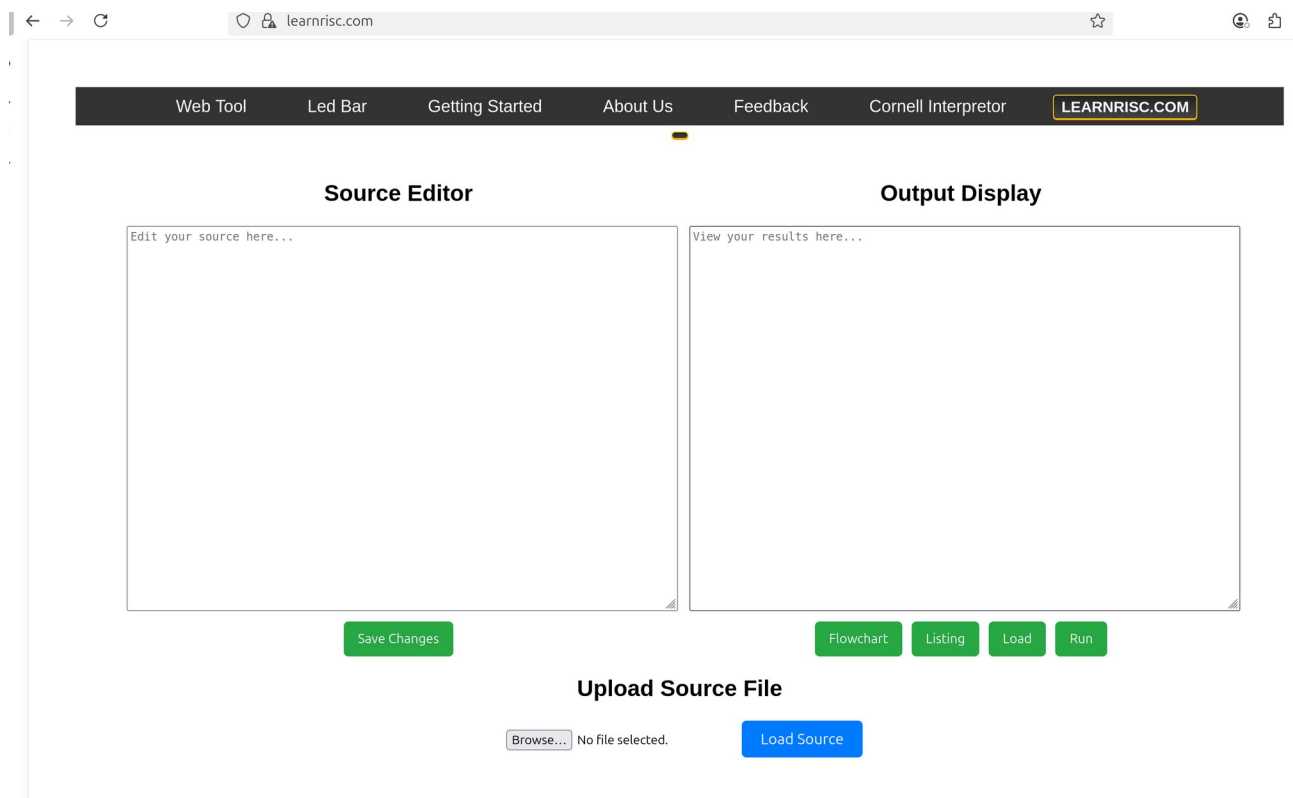
**SLL** Shifts the bits in a register to the left by a specified number of positions (from another register). Often used for multiplication by powers of two.

**SRL** Shifts the bits in a register to the right logically, filling with zeros. Used for unsigned division by powers of two or bit manipulation.

**SRA** Shifts the bits in a register to the right arithmetically, preserving the sign bit. Used for signed division by powers of two.

## Web Tool

The web tool page contains a source window and an multiple output windows.



## Source Editor

The **Source Editor Window** can be used to write your program directly or the editor application Geany can be used.

Sometimes for ease, it may be useful to create a program in the Source window Editor and use the output display features as part of a debug as you go process. The “Save Changes” button can be used to for saving and “compile”, “flow chart”, etc can be used as part of an quick design method.

If your code is already in a folder the “Browse button” is used to select the file and the load source button to load the selected file to the source window.

## Flowchart Window

The **flowchart** button calls a process that uses control characters you embed in your source code comments to generate a flow of your code. The chart is drawn with ASCII characters.

### EXAMPLE FLOW CHART

#### FIBONACCI FLOW CHART

```
void _Fibonacci(FirstNumber, NumberToCalculate)
|
|   last_value = 0
|   |
|   Memory_Address = 0x2000
|   |
|   display "Fibonacci Series"
|   |
|   [Start]->|
|   |
|   [Finish]<-Y-< > NumberToCalculate == 0 ?
|   |N
|   |   last_value_temp = Number
|   |   |
|   |   Number = Number + last_value
|   |   |
|   |   log Number to memory
|   |   |
|   |   increment mem ptr
|   |   |
|   |   Display Number
|   |   |
|   |   last_value = last_value_temp
|   |   |
|   |   NumberToCalculate--
|   |   |
|   |   [Start ]<-o
|   |   |
|   |   [Finish]->|
|   |   |
|   |   Add 2 zeroes to mem for readability
|   |   |
|   |   Display "Complete"
|   |   |
|   |   o exit
```

## Flowchart Quick sheet

Overview:

Comments in C programs and many others support two type of comments. The first is the `//` , these double slash characters designate all character that follow in the same line as comment characters that should not be compiled or processed.

The second type is the `/* */` pair implies all lines between as comments.

```
/*
```

This is a multi-line comment!

```
*/
```

The program flow program generates a flowchart from characters embedded in your program files and defined by control characters such as `//c-x` that the flow program interpreters, but the assembler ignores.

For multi-line flow comments `/*c-g` at the start of a comment implies the text between it and the `*/` to follow should be included in the flowchart usually as a summary of the code.

Embedded into the comments of a program. The FlowChart output is generated in a txt file named FlowChart.txt.

### Here are the options:

```
//s- void Computer::PrintCout(unsigned offset)
```

```
-----
```

```
void Computer::PrintCout(unsigned offset)
```

```
|
```

```
//s-x void Computer::PrintCout(unsigned offset)
```

```
-----
```

```
void Computer::PrintCout(unsigned offset)
```

```
//c- This is a comment!
```

```
-----
```

Output:

```
// This is a comment!  
//c-3 This is a two comment,blank lines added above
```

-----

```
//c- Second comment//e-b entry point b
```

-----

Output:

```
[b]->|
```

|

```
//g-d goto point d
```

-----

```
[d] ← o
```

```
//Y-b Is the sub red?
```

```
7/8-----
```

```
[b]<-N-< > Is the sun red?
```

```
|Y
```

```
//N-c Is the sub red?
```

-----

```
[c]<-Y-< > Is the sun red?
```

```
|N
```

```
//x- exit
```

-----

```
o exit
```

```
/*c-g to start multiline comments, */ to end it
```

-----

```
this is
```

```
a multi
```

```
line comment
```

## Code Listing Window

An assembly program listing is a comprehensive, human-readable output generated by an assembler that shows both the original assembly language instructions and the corresponding machine code. It serves as a bridge between the high-level assembly source code and the low-level binary instructions executed by a computer's processor.

Components of an Assembly Program Listing:

1. **Memory Addresses:** Indicate where each instruction or data segment resides in memory.
2. **Machine Code (Opcodes):** The binary or hexadecimal representation of instructions.
3. **Source Code:** The original assembly language instructions written by the programmer.
4. **Labels and Symbols:** Identifiers representing memory addresses or constants.
5. **Comments:** Explanations added by the programmer to clarify code functionality.
6. **Assembler Directives:** Instructions to the assembler, such as defining data sections.

### Importance of an Assembly Program Listing

1. **Debugging:** Listings allow programmers to verify that the assembler correctly translated the source code, aiding in locating errors such as incorrect opcodes or misaligned data.
2. **Optimization:** By analyzing the machine code and instruction flow, developers can fine-tune code for performance and efficiency.
3. **Educational Insight:** Listings provide a clear view of how high-level programming constructs are implemented at the hardware level, which is crucial for learning system architecture.
4. **Verification:** Ensures that labels, addresses, and instructions are correctly resolved and placed.
5. **Reverse Engineering and Analysis:** Useful for understanding and analyzing compiled binaries.
6. **Documentation:** Acts as a technical reference detailing how symbolic names map to actual machine instructions and memory locations.



## Example of a Simple Assembly Program Listing:

```
// Title: Fibonacci SeriesRISC-v Assembler by RWescott - Version 1.05
Flag = -n
```

```
_Setup();
1    jal t0,148      // Call to _Setup()
```

```
// Title: Fibonacci Series
```

```
void main()
{
2    sw t0,-4(sp)     // Save ret, caller left space
```

```
// Calculate 18 numbers starting with seed
```

```
int Seed ;
3    addi t1,zero,33
4    sw t1,0(sp)
5    addi sp,sp,4
```

```
int Number ;
6    addi t1,zero,33
7    sw t1,0(sp)
8    addi sp,sp,4
```

```
Seed = 6;
9    addi zero,zero,0 // nop
10   addi t1,zero,6   // macro- t1 = 6
11   sw t1,-8(sp)
Number = 18;
12   addi zero,zero,0 // nop
13   addi t1,zero,18  // macro- t1 = 18
14   sw t1,-4(sp)
```

```
_Fibonacci(Seed,Number);
15   lw t1,-8(sp)
16   lw t2,-4(sp)
17   jal t0,20        // Call to _Fibonacci()
```

```
_Exit();
18   jal t0,92        // Call to _Exit()
19   addi sp,sp,-12    // Exiting sub, Clear Stack
20   lw t0,0(sp)       // Load return address
21   jalr zero,t0,0     // return
}
```

## Load File Window

Each instruction in this load file represents a real machine-level operation intended to be loaded at specific memory addresses. Simulators like Cornell's map these instructions into a virtual memory layout to execute them step-by-step.

The combination of stack pointer adjustments (`sp`), immediate values (`addi`, `andi`), jump instructions (`jal`, `jalr`), and data loads/stores (`lw`, `sw`) illustrates typical operations found in low-level program execution.

This format aids in visualizing how instructions and variables are laid out in memory, ensuring correctness during program execution. Load files are vital tools for debugging, testing, and educational exploration of RISC-V architecture.

A RISC-V load file consists of a sequence of actual machine instructions and data variables. This file is structured to map instructions directly to memory addresses, which a processor or simulator reads and executes. It is commonly used in simulation environments like the Cornell RISC-V Simulator to emulate program execution realistically.

---

### Example Load File Content:

```
jal t0,148
sw t0,-4(sp)
addi t1,zero,33
sw t1,0(sp)
addi sp,sp,4
addi t1,zero,33
sw t1,0(sp)
addi sp,sp,4
addi zero,zero,0
addi t1,zero,6
sw t1,-8(sp)
addi zero,zero,0
addi t1,zero,18
sw t1,-4(sp)
lw t1,-8(sp)
lw t2,-4(sp)
jal t0,20
```

## IO (input /output)

### Output Instruction

To output text and variables to a display, a cout utility is included. This is similar to the C language instruction. This instruction is supported by the “addi x0, offset,x31” instruction that is normally a nop (no operation) instruction. The compiler handles the message in an IO block that is part of the load file.

Example Instruction:

```
cout << “The number is “ << 0x22 << endl;
```

### Input Instruction

Input text is requested by the cin utility. It is a request for input from the program user. An option key phrase and a buffer for the response append to the request. This instruction is supported by the “addi x0, offset,x30” instruction that is normally a nop (no operation) instruction.

Example instruction:

```
cin >> “message” >> buffer length >> buffer_address ;
```

### IO File Example:

```
00636F7574203C3C20225072
6F6772616D3A204C65644F6E
20656E646C00636F7574203C
3C202022494F3A4C45445322
20203C3C20656E646C3B0063
6F7574203C3C202253657474
696E67207833312026202078
333020746F20342220203C3C
20656E646C00636F7574203C
3C202022494F3A4C45445322
20203C3C20656E646C3B0063
6F7574203C3C2022436F6D70
6C6574652220203C3C20656E
646C3B00636F7574203C3C20
2250726F6772616D20436F6D
706C657465223B00
```

## Run Window

A Run Window is an interface that displays the textual output generated by a running RISC-V program. Unlike traditional RISC-V environments where output is minimal and often limited to register or memory displays, this Run Window provides the functionality of high-level language console. Running the program gives you instant response indicating the program is successful.

These i/o instructions are handled at runtime by the simulator, translating into appropriate formatted text in the Run Window.

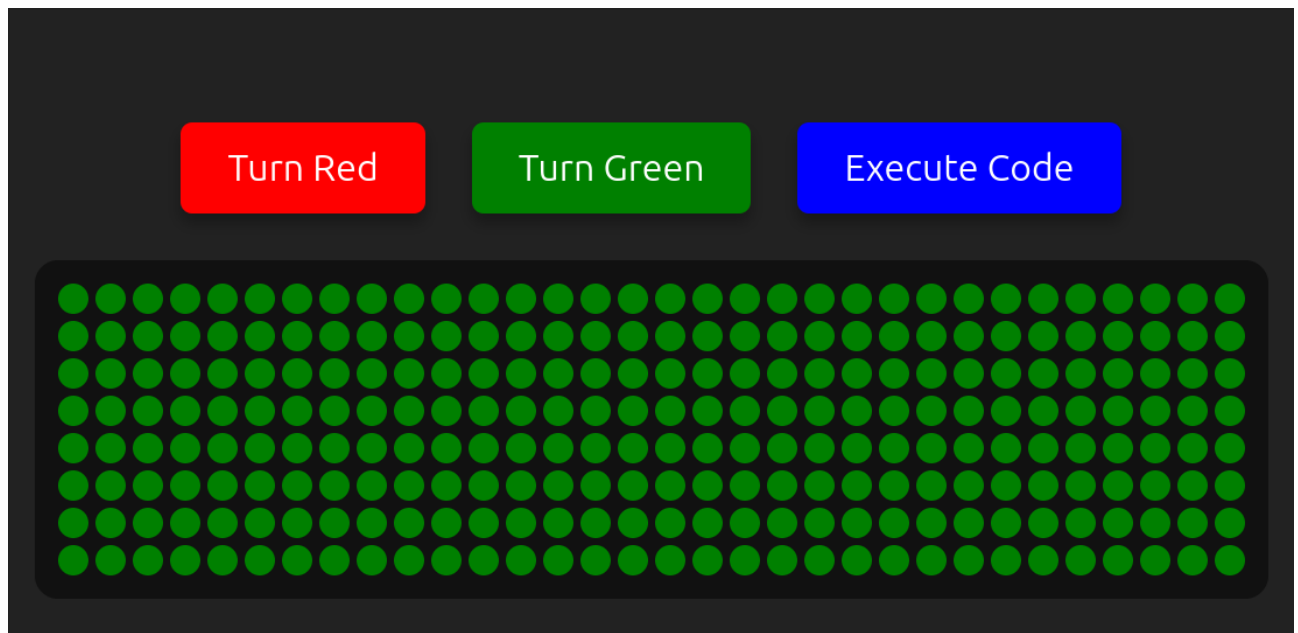
### Functionality of the Run Window

- Displays sequential console-like output from a running RISC-V program.
- Supports formatted text output, such as strings and register values.
- Updates dynamically as the program executes, similar to how a terminal window reflects program progress.
- Enhances debugging and program interaction by providing human-readable output directly from assembly routines.
- Allows C++ style output behavior while still writing in RISC-V assembly.

### Purpose and Use

This Run Window bridges the gap between low-level RISC-V assembly programming and high-level program interaction. It makes teaching and visualizing program flow more intuitive for learners accustomed to high-level language outputs, and facilitates debugging by allowing easy insertion of print statements for debugging purposes.

## Led Bar



### HTML LED Bar for Simulated I/O Instruction Output

An HTML LED Bar Page is an interactive web interface designed to visually represent the output of I/O instructions from a RISC-V (or other embedded system) program. The page features an 8 x 32 grid of simulated LEDs—a matrix layout commonly used in real hardware displays. This simulation allows users to see how I/O instructions can manipulate hardware outputs in real-time, making it a powerful educational and debugging tool.

#### Purpose of the LED Bar Simulation

- **Demonstrate I/O Instruction Behavior:** The LED grid visually reflects the effect of assembly-level I/O instructions that write to output ports, simulating the toggling of LEDs in patterns.
- **Bridge Software and Hardware Concepts:** Shows how low-level code (like writing a byte to an output register) translates to real-world hardware responses, without needing physical LEDs.
- **Educational Visualization:** Helps students and developers understand concepts like bit masking, shifting, port output, and pattern generation in a highly visual way.
- **Debugging Aid:** Makes it easier to observe and verify complex I/O patterns generated by code loops or algorithmic sequences.

#### I/O Instruction Example

For instance, a RISC-V extended instruction like:

```
cout <<
```

where `t0` holds a byte pattern like `0b10101010`, would light up alternating LEDs in a row. Similarly, sequences can be written to output scrolling patterns, flashing sequences, or image-like arrangements across the 8 x 32 grid.

## Special LED Instruction Mapping

To simplify pattern output, a special pseudo-instruction named LED\_OUT is introduced. This instruction maps the contents of registers x16 through x23 directly to the LED matrix rows. Each register (x16-x23) represents one row of 32 bits, where each bit corresponds to an LED in that row.

## LED Features

- 8 Rows x 32 Columns of Simulated LEDs: Each cell can be individually turned on (lit) or off (dimmed), controlled via code.
- Live I/O Update Interface: The page updates the LED grid in real-time based on data sent by the program.
- 

## Pattern Change Delay

To enhance readability and allow users to clearly see each pattern transition, a 1-second delay is inserted after each LED\_OUT execution. This ensures that LED patterns do not change too rapidly and allows observers to follow the output sequence easily.

## Educational Impact

This interactive LED Bar bridges the gap between low-level assembly programming and visual hardware interaction. It provides an engaging platform to teach bitwise operations, port I/O, and pattern generation techniques in a highly visual context.

## Program Examples:

### Fibonacci Series

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, starting from 0 and 1. That is, the series begins 0, 1, 1, 2, 3, 5, 8, 13, 21, and continues indefinitely. Named after the Italian mathematician Leonardo of Pisa—also known as Fibonacci—this sequence was introduced to Western mathematics in his 1202 book *Liber Abaci*. The Fibonacci series appears frequently in nature, such as in the arrangement of leaves, the branching of trees, and the spirals of shells. It also has applications in computer algorithms, financial modeling, and art due to its connection with the golden ratio, which emerges as the ratio of successive Fibonacci numbers approaches approximately 1.618.

### RESULTS – PROGRAM OUTPUT for Fibonacci code:

Fibonacci Series

```
6
12
18
30
48
78
126
204
330
534
864
1398
2262
3660
5922
9582
15504
25086
Complete
```

Program Complete

## FIBONACCI FLOW CHART

```

void _Fibonacci (FirstNumber, NumberToCalculate)
|
|   last_value = 0
|   |
|   Memory_Address = 0x2000
|   |
|   display "Fibonacci Series"
|   |
|   [Start]->|
|   |
|   [Finish]<-Y-< > NumberToCalculate == 0 ?
|   |N
|   |   last_value_temp = Number
|   |   |
|   |   Number = Number + last_value
|   |   |
|   |   log Number to memory
|   |   |
|   |   increment mem ptr
|   |   |
|   |   Display Number
|   |   |
|   |   last_value = last_value_temp
|   |   |
|   |   NumberToCalculate--
|   |   |
|   |   [Start ]<-o
|   |   |
|   |   [Finish]->|
|   |   |
|   |   Add 2 zeroes to mem for readability
|   |   |
|   |   Display "Complete"
|   |   |
|   |   o exit

```



## Fibonacci Code Listing:

// Title: Fibonacci SeriesRISC-v Assembler by RWescott - Version 1.05  
Flag = -n

\_Setup();

1 jal t0,148 // Call to \_Setup()

// Title: Fibonacci Series

void main()

{

2 sw t0,-4(sp) // Save ret, caller left space

// Calculate 18 numbers starting with seed

int Seed ;

3 addi t1,zero,33

4 sw t1,0(sp)

5 addi sp,sp,4

int Number ;

6 addi t1,zero,33

7 sw t1,0(sp)

8 addi sp,sp,4

Seed = 6;

9 addi zero,zero,0 // nop

10 addi t1,zero,6 // macro- t1 = 6

11 sw t1,-8(sp)

Number = 18;

12 addi zero,zero,0 // nop

13 addi t1,zero,18 // macro- t1 = 18

14 sw t1,-4(sp)

\_Fibonacci(Seed,Number);

15 lw t1,-8(sp)

16 lw t2,-4(sp)

17 jal t0,20 // Call to \_Fibonacci()

\_Exit();

18 jal t0,92 // Call to \_Exit()

19 addi sp,sp,-12 // Exiting sub, Clear Stack

20 lw t0,0(sp) // Load return address

21 jalr zero,t0,0 // return

}

## Fibonacci Source:

```
// Title: Fibonacci Series
```

```
void main()
{
```

```
// Calculate 18 numbers starting with seed
int Seed ;
int Number ;
```

```
Seed = 6;
Number = 18;
```

```
_Fibonacci(Seed,Number);
```

```
_Exit();
}
```

```
void _Fibonacci(FirstNumber,NumberToCalculate) //a-3
{
assign t1 FirstNumber
assign t1 Number
assign t2 NumberToCalculate
assign x27 last_value_temp
assign x28 last_value
assign x29 NumberToCalculate
assign x30 mem_add
assign x31 last_value
```

```
//s- last_value = 0
add last_value,zero,zero
```

```
//s- Memory_Address = 0x2000
// lower 12 bits to 0
lui mem_add,0x2
```

```
//s- display "Fibonacci Series"
cout << "Fibonacci Series" << endl;
```

```
Start:
```

```
//N-Finish NumberToCalculate == 0 ?
```

```
beq NumberToCalculate,zero, Finish
```

```
//s- last_value_temp = Number  
addi last_value_temp, Number, 0 // previou = number
```

```
//s- Number = Number + last_value  
add Number,Number,last_value
```

```
//s- log Number to memory  
sw Number,0(mem_add)
```

```
//s- increment mem ptr  
addi mem_add,mem_add,4
```

```
//s- Display Number  
cout << x6 << endl;
```

```
//s- last_value = last_value_temp  
add last_value,last_value_temp,zero
```

```
//s- NumberToCalculate--  
addi NumberToCalculate,NumberToCalculate,-1
```

```
//g-Start  
beq zero,zero, Start
```

Finish:

```
//s- Add 2 zeroes to mem for readability  
sw zero,0(mem_add)
```

```
sw zero,4(mem_add)
```

```
//s- Display "Complete"  
cout << "Complete" << endl;
```

```
//x-  
}
```

## Multiply Program

### RISC-V Base Multiply Example using Hex-Digit Loop

The Base RISC-V (RV32I) instruction set does not include a hardware multiply instruction. Multiplication must be performed using basic operations such as additions and shifts. This is analogous to the method learned in grade school, but executed in binary or hexadecimal (base 16), which is more suitable for digital systems.

#### Hex-Digit Multiplication Approach:

Instead of processing one binary digit at a time, this approach processes 4-bit chunks (hex digits). Each hex digit ranges from 0 to 15. For each digit, the multiplicand is multiplied by the digit value, shifted left according to its position, and accumulated into the result.

#### Explanation:

1. The multiplier is processed 4 bits (hex digit) at a time.
2. For each digit, a loop adds the multiplicand the number of times equal to the digit value (0-15).
3. The partial product is then shifted to align with its correct position in the overall product.
4. This simulates grade-school multiplication in base 16.
5. The loop continues until all hex digits are processed.

#### Conclusion:

This example demonstrates how multiplication can be implemented in the RISC-V base instruction set using shifts and additions, processing one hex digit at a time. While not as fast as a hardware multiply, it is functionally complete.

## **MULTIPLY RESULTS – PROGRAM OUTPUT**

Multiply 456 times 92

Multipicand = 456

Multiplier = 92

Multipling by hex 12

Multipling by hex 5

Result = 41952

Program Complete

## MULTIPLY FLOW CHART

```

    int _Multiply()
    |
    // Multiplier x Multiplicand = Product
    |
    Product = 0
    |
    [Next]<-N-< >Multiplier > Multiplicand ?
    |Y
Switch positions, lowest to Multiplicand
    |
    [Next]->|
    |
    Print Values
    |
    [Digits]->|
    |
    digit = lowest 16 of Multiplier
    |
    Multiplier = Multiplier / 16
    |
    [Add2Total]->|
    |
    Product += digit * Multiplicand
    |
    multiply by addition
    |
    [Add2Total]<-N-< >Done ?
    |Y
    [NextDigit]->|
    |
    [Digits]<-Y-< >Multiplier has more digits ?
    |N
    Print results
    |
    o exit

```

## Multiply Code Listing:

RISC-v Assembler by RWescott - Version 1.05

Flag = -n

```
_Setup();
1    jal t0,184      // Call to _Setup()

// Multiply.txt

void main()
{
2    sw t0,-4(sp)     // Save ret, caller left space
// Calculate product of two numbers

3    addi t1,zero,33
4    sw t1,0(sp)
5    addi sp,sp,4
6    addi t1,zero,33
7    sw t1,0(sp)
8    addi sp,sp,4
9    addi t1,zero,33
10   sw t1,0(sp)
11   addi sp,sp,4

NumberA = 456;
12   addi zero,zero,0 // nop
13   addi t1,zero,456 // macro- t1 = 456
14   sw t1,-12(sp)
NumberB = 92;
15   addi zero,zero,0 // nop
16   addi t1,zero,92  // macro- t1 = 92
17   sw t1,-8(sp)

Results = _Multiply(NumberA,NumberB); // 456 * 92 = 41,952
18   lw t1,-12(sp)
19   lw t2,-8(sp)
20   jal t0,24        // Call to _Multiply()
Results = t1
21   sw t1,0(sp)

_Exit();
22   jal t0,112       // Call to _Exit()
23   addi sp,sp,-16    // Exiting sub, Clear Stack
24   lw t0,0(sp)       // Load return address
25   jalr zero,t0,0    // return
}
```

```

// multiplication by addition and shifting
//      abc x ef
//  abc
//  ef
//  ^^
//  ||
//  |-----multiply abc times f
//  -----multiply abc times e times 16 , total = product

// Multiple 2 positive integers
// t2 = input_Multiplier
// t1 = input_Multiplicand

// output
// t1 = Product
int _Multiply() //a-
{
    assign t1 InputMultiplicand
    assign t2 Multiplier
    assign t1 Product
    assign t3 Multiplicand;
    assign t4 Temp
    assign t4 Digit

//s- // Multiplier x Multiplicand = Product

    26    addi zero,x31,0

    27    addi Multiplicand,InputMultiplicand,0// set Multiplicand

//s- Product = 0
    28    addi Product,zero,0    /// zero Product register

//Y-Next Multiplier > Multiplicand ?
    29    blt Multiplier,Multiplicand,16

//s- Switch positions, lowest to Multiplicand
    30    addi Temp,Multiplicand,0
    31    addi Multiplicand,Multiplier,0
    32    addi Multiplier,Temp,0

    Next:

//s- Print Values
    33    addi zero,x31,116
    34    addi zero,x31,200

```



```

    Digits:
//s- digit = lowest 16 of Multiplier
35    addi Digit,Multiplier,0
36    andi Digit,Digit,15

//s- Multiplier = Multiplier / 16
37    srli Multiplier,Multiplier,4// for next digit

38    addi zero,x31,280

    Add2Total:
//s- Product += digit * Multiplicand
//s- multiply by addition

39    beq Digit,zero,16
40    add Product,Product,Multiplicand
41    addi Digit,Digit,-1 // decrement t1

//Y-Add2Total Done ?
42    bne Digit,zero,-12 // = 0 ?

    NextDigit:
// shift for next digit
43    slli Multiplicand,Multiplicand,4// Multiplicand * 16

//N-Digits Multiplier has more digits ?
44    bne Multiplier,zero,-36// bne Digits

//s- Print results
45    addi zero,x31,372
//x-
46    jalr zero,t0,0    // return
}
//library

// setuup memory, globals, sp, tp, gp, string storage...
void _Setup()
{
// code setup section
sp = 0x1000
47    lui sp, 1        //    macro- sp = 0x1000
48    addi sp,sp,0

49    jalr zero,t0,0    // return
}

```

```
void _Exit()
{
50    addi zero,x31,446
51    jal x1,8        // Call to 8()
52    jalr zero,t0,0    // return
}
```

## Multiply Code Source

```
// Multiply.txt

void main()
{
// Calculate product of two numbers

int NumberA;
int NumberB;
int Results;

NumberA = 456;
NumberB = 92;

Results = _Multiply(NumberA,NumberB); // 456 * 92 = 41,952

_Exit();
}


// multiplication by addition and shifting
//      abc x ef
//  abc
//   ef
//  ^^
//  ||
//  |----multiply abc times f
//  -----multiply abc times e times 16 , total = product


// Multiple 2 positive integers
// t2 = input_Multiplier
// t1 = input_Multiplicand


// output
// t1 = Product
int _Multiply() //a-
{
assign t1 InputMultiplicand
assign t2 Multiplier
assign t1 Product
assign t3 Multiplicand;
assign t4 Temp
assign t4 Digit
```

```

//s- // Multiplier x Multiplicand = Product

cout << "Multiply " << x6 << " times " << x7 << endl;

addi Multiplicand,InputMultiplicand,0 // set Multiplicand

//s- Product = 0
addi Product,zero,0 // zero Product register

//Y-Next Multiplier > Multiplicand ?
blt Multiplier,Multiplicand,Next

//s- Switch positions, lowest to Multiplicand
addi Temp,Multiplicand,0
addi Multiplicand,Multiplier,0
addi Multiplier,Temp,0

Next:

//s- Print Values
cout << "Multipicand = " << x28 << endl;
cout << "Multiplier = " << x7 << endl;

Digits:
//s- digit = lowest 16 of Multiplier
addi Digit,Multiplier,0
andi Digit,Digit,15

//s- Multiplier = Multiplier / 16
srli Multiplier,Multiplier,4 // for next digit

cout << "Multipling by hex " << x29 << endl;

Add2Total:
//s- Product += digit * Multiplicand
//s- multiply by addition

beq Digit,zero,NextDigit
add Product,Product,Multiplicand
addi Digit,Digit,-1 // decrement t1

//Y-Add2Total Done ?
bne Digit,zero,Add2Total // = 0 ?

NextDigit:
// shift for next digit
slli Multiplicand,Multiplicand,4 // Multiplicand * 16

```

```
//N-Digits Multiplier has more digits ?  
bne Multiplier,zero,Digits  
  
//s- Print results  
cout << "Result = " << x6 << endl;  
//x-  
}
```

## AdividesB Program

RISC-V Program: AdividesB (A,B)

Concept:

**Given variables A (divisor) and B (dividend), we want to determine if B is evenly divisible by A (i.e.,  $B \% A == 0$ ). Since the RISC-V base instruction set (RV32I) does not include a hardware division instruction, this functionality must be implemented using subtraction and bit shifting.**

---

**Optimized Approach Overview:**

- 1. Start with B (dividend) and subtract large multiples of A (divisor).**
- 2. Determine the largest multiple of A (by left-shifting) that can be subtracted from B.**
- 3. Continue subtracting until B is less than A.**
- 4. If the remainder is zero, A divides B evenly; otherwise, it does not.**

**Explanation:**

- 1. The program starts by checking for division by zero.**
  - 2. It then finds the largest multiple of A (via left shifting) that is less than or equal to B.**
  - 3. A loop subtracts this multiple from the current remainder (starting as B).**
  - 4. If a multiple becomes too large, it is reduced by shifting right.**
  - 5. The loop continues until no larger multiples can be subtracted.**
  - 6. If the final remainder is zero, A divides B evenly.**
  - 7. The function returns 1 if divisible, 0 otherwise.**
-

**Bit Shifting for Speed:**

**Using shifts accelerates the division process by subtracting large multiples of the divisor at each step. This approach reduces the number of subtraction iterations significantly compared to a basic subtraction loop, similar to how binary long division operates.**

## **AdividesB RESULTS – PROGRAM OUTPUT**

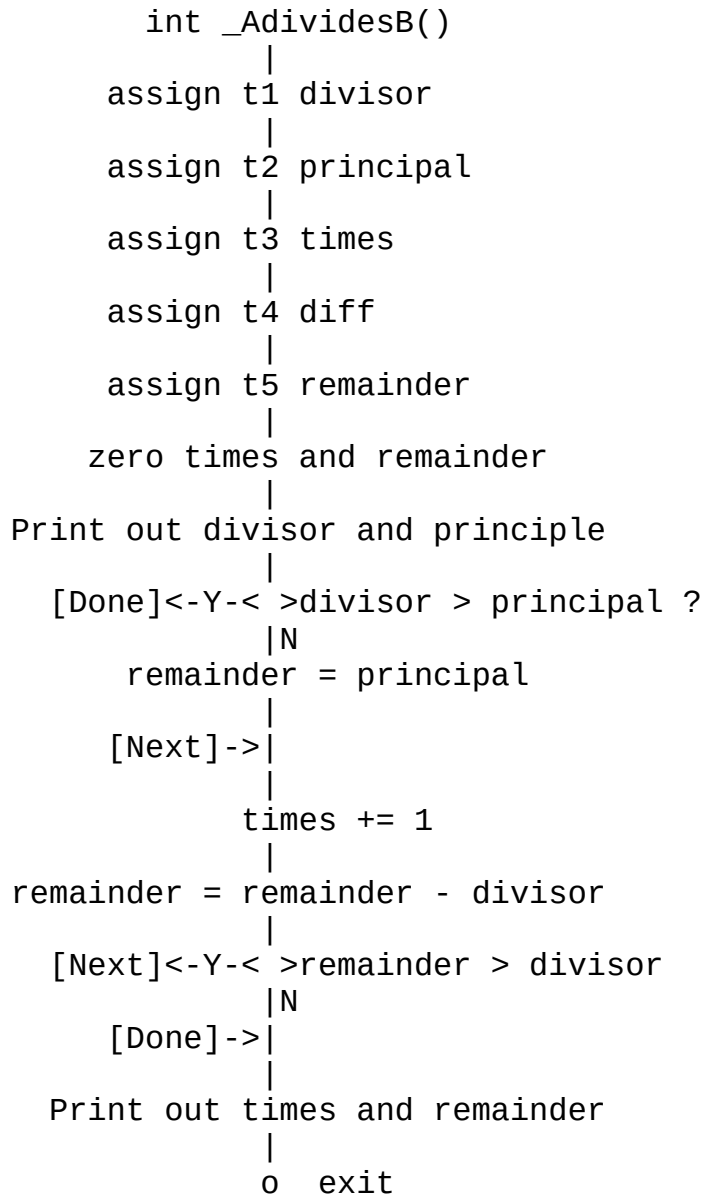
divisor = 6  
principal = 125

Result:  
Times = 20  
Remainder = 5

Program Complete

## AdividesB FLOW CHART

Program AdividesB does not include a divide instruction  
This routine divides by subtraction  
Divide A into B and return times and remainder





## AdividesB Code Listing:

RISC-v Assembler by RWescott - Version 1.05

Flag = -n

```
_Setup();
1    jal t0,144      // Call to _Setup()

void main()
{
2    sw t0,-4(sp)     // Save ret, caller left space
3    addi t1,zero,33
4    sw t1,0(sp)
5    addi sp,sp,4
6    addi t1,zero,33
7    sw t1,0(sp)
8    addi sp,sp,4

divisor = 6;
9    addi zero,zero,0 // nop
10   addi t1,zero,6   // macro- t1 = 6
11   sw t1,-8(sp)
principal = 125;
12   addi zero,zero,0 // nop
13   addi t1,zero,125 // macro- t1 = 125
14   sw t1,-4(sp)

_AdividesB(divisor,principal);
15   lw t1,-8(sp)
16   lw t2,-4(sp)
17   jal t0,20        // Call to _AdividesB()

_Exit();
18   jal t0,88        // Call to _Exit()
19   addi sp,sp,-12    // Exiting sub, Clear Stack
20   lw t0,0(sp)       // Load return address
21   jalr zero,t0,0     // return
}

int _AdividesB() //a-5
{
// input
assign t1 divisor //a-
assign t2 principal //a-
```

```

//uses
assign t3 times    //a-
assign t4 diff     //a-
assign t5 remainder //a-

//returns

//s- zero times and remainder
22    addi times,zero,0
23    addi remainder,zero,0

//s- Print out divisor and principle
24    addi zero,x31,0
25    addi zero,x31,78

//N-Done divisor > principal ?
26    blt t2,t1,20

//s- remainder = principal
27    addi remainder,principal,0

    Next:

//s- times += 1
28    addi times,times,1

//s- remainder = remainder - divisor
29    sub remainder,remainder,divisor

30    bge remainder,divisor,-8
//N-Next remainder > divisor

    Done:
31    addi zero,x31,176
//s- Print out times and remainder
32    addi zero,x31,232
33    addi zero,x31,304

34    addi t2,remainder,0
35    addi t1,times,0

//x-
36    jalr zero,t0,0    // return
}
//library

// setep memory, globals, sp, tp, gp, string storage...
void _Setup()
{

```

```

// code setup section
sp = 0x1000
37    lui sp, 1        //    macro- sp = 0x1000
38    addi sp,sp,0

39    jalr zero,t0,0    // return
}

```

```

void _Exit()
{
40    addi zero,x31,386
41    jal x1,8          // Call to 8()
42    jalr zero,t0,0    // return
}

```

## AdividesB Code Source

```
void main()
{
int divisor;
int principal;

divisor = 6;
principal = 125;

_AdividesB(divisor,principal);

_Exit();
}

/*c-g
Program AdividesB does not include a divide instruction
This routine divides by subtraction
Divide A into B and return times and remainder
*/

int _AdividesB() //a-5
{
// input
assign t1 divisor //a-
assign t2 principal //a-

//uses
assign t3 times //a-
assign t4 diff //a-
assign t5 remainder //a-

//returns
x1 = times
x2 remainder

//s- zero times and remainder
addi times,zero,0
addi remainder,zero,0

//s- Print out divisor and principle
cout << "divisor = " << x6 << endl;
cout << "principal = " << x7 << endl << endl;
```

```
//N-Done divisor > principal ?  
blt t2,t1,Done
```

```
//s- remainder = principal  
addi remainder,principal,0
```

Next:

```
//s- times += 1  
addi times,times,1
```

```
//s- remainder = remainder - divisor  
sub remainder,remainder,divisor
```

```
bge remainder,divisor,Next  
//N-Next remainder > divisor
```

```
Done:  
cout << "Result:" << endl;  
//s- Print out times and remainder  
cout << "Times = " << x28 << endl;  
cout << "Remainder = " << x30 << endl;
```

```
addi t2,remainder,0  
addi t1,times,0
```

```
//x-  
}
```

## Primes

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. In other words, a prime number can only be divided evenly by 1 and by itself without leaving a remainder. Examples of prime numbers include 2, 3, 5, 7, 11, and 13. The number 2 is the smallest and the only even prime number; all other even numbers are divisible by 2 and therefore not prime.

Prime numbers are important because they serve as the building blocks of all natural numbers. This concept is known as the Fundamental Theorem of Arithmetic, which states that every integer greater than 1 can be represented uniquely as a product of prime numbers. Prime numbers are also crucial in modern cryptography, particularly in public-key encryption algorithms such as RSA, where large prime numbers are used to secure digital communications. Their unpredictable distribution makes them useful for computer security, random number generation, and various fields of mathematics.

### Programmer approach

#### Approach to Creating a List of the First n Prime Numbers

To create a list of the first n prime numbers, the goal is to identify numbers that meet the definition of a prime: having no positive divisors other than 1 and itself. This process involves checking each candidate number for primality and collecting the results until n prime numbers have been found. One common approach is as follows: 1. Start with an empty list to store prime numbers. 2. Begin testing numbers starting from 2, the smallest prime. 3. For each candidate number, check whether it is divisible by any previously found prime (or by any number from 2 up to its square root). 4. If the candidate has no divisors other than 1 and itself, it is prime; add it to the list. 5. Repeat the process until the list contains n prime numbers.

For efficiency, the divisibility check can be limited to numbers up to the square root of the candidate, since any factor larger than the square root will have a corresponding factor smaller than the square root. Additionally, eliminate even number since they can be divided by 2.

Of note is the code used allows a group of programs to share common registers. In assembly there is a freedom to do so, without using common memory locations.

## PRIME FLOW CHART

```
    _GeneratePrimes(20);  
    |  
Set number of primes to calculate  
    |  
    Set Prime Memory start  
    |  
    first prime = 2  
    |  
    numberFound = 1  
    |  
    number = 1  
    |  
[Test]->|  
    |  
    number = number + 2  
    |  
[Test]<-N-< >Number is Prime ?  
    |Y  
    add as prime  
    |  
    numberFound++  
    |  
[Test]<-N-< >Max reached ?  
    |Y  
    o  exit
```

```

    int _AdividesB()
        |
    assign t1 divisor
        |
    assign t2 principal
        |
    assign t3 times
        |
    assign t4 diff
        |
    assign t5 remainder
        |
    zero times and remainder
        |
Print out divisor and principle
        |
    if divisor > principal Done
        |
    [Done]<-Y-< >divisor > principal ?
        |N
        remainder = principal
        |
    [Next]->|
        |
        Increment times
        |
    remainder = remainder - divisor
        |
    [Next]<-Y-< >remainder > divisor

```



```

        |N
    [Done]->|
        |
    Print out times and remainder
        |
        o  exit

```

```

    int _CheckIfPrime()
        |
    Test number against all primes before
        |
        number in x13
        |
        load prime array ptr
        |
    [test]->|
        |
        load prime
        |
    [NotPrime]<-Y-< >Prime a factor of number?
        |N
    [test]<-Y-< >More primes ?
        |N
    [Prime]->|
        |
        Set results to 1
        |
        goto end
        |

```

```
[NotPrime]->|
    |
    set results to 0
    |
[end]->|
    |
    return results
    |
    o  exit
```

## PRIME Code Listing:

### RISC-v Assembler by RWescott - Version 1.05

Flag = -n

\_Setup();

```
1    jal t0,240      // Call to _Setup()
```

void main()

{

```
2    sw t0,-4(sp)    // Save ret, caller left space
```

\_GeneratePrimes(20); //a-3

```
3    addi zero,zero,0 // nop
```

```
4    addi t1,zero,20  // macro- t1 = 20
```

```
5    jal t0,20        // Call to _GeneratePrimes()
```

\_Exit();

```
6    jal t0,232      // Call to _Exit()
```

```
7    addi sp,sp,-4    // Exiting sub, Clear Stack
```

```
8    lw t0,0(sp)      // Load return address
```

```
9    jalr zero,t0,0    // return
```

}

void \_GeneratePrimes()

{

```
10   sw t0,0(sp)      // Save Return
```

```
11   addi sp,sp,4
```

assign x10 Max // passed number

assign x11 arrayStart

assign x12 memoryPtr

assign x13 number

assign x14 true

assign x15 numberFound

// calls:

// CheckIfPrimes() shares needed registers

// returns t0 = 1 if prime, t1 = 0 otherwise

//s- Set number of primes to calculate

12     addi Max,t1,0

//s- Set Prime Memory start

13     addi zero,zero,0     // nop

14     addi x11,zero,0x100   //     macro- x11 = 0x100

15     addi x12,x11,0

16     addi zero,x31,0

//s- first prime = 2

17     addi number,zero,2

18     sw number,0(memoryPtr)

19     addi memoryPtr,memoryPtr,4// increment ptr

20     addi zero,x31,78

//s- numberFound = 1

21     addi numberFound,zero,1

//s- number = 1

22     addi number,zero,1

Test:

```

//s- number = number + 2
23    addi number,number,2

//Y-Test Number is Prime ?
_CheckIfPrime();
24    jal t0,88      // Call to _CheckIfPrime()

25    addi true,zero,1  // true = 1
26    bne true,t1,-12   // bne Test

//s- add as prime
27    sw number,0(memoryPtr)
28    addi memoryPtr,memoryPtr,4// increment ptr

29    addi zero,x31,122

//s- numberFound++
30    addi numberFound,numberFound,1

//Y-Test Max reached ?
31    blt numberFound,Max,-32

32    addi zero,x31,168

//x-
33    addi sp,sp,-4
34    lw t0,0(sp)
35    jalr zero,t0,0    // return
}

```

```

int _AdividesB() //a-5
{
// input
assign t1 divisor //a-
assign t2 principal //a-

//uses
assign t3 times //a-
assign t4 diff //a-
assign t5 remainder //a-

//returns

//s- zero times and remainder
36    addi times,zero,0
37    addi remainder,zero,0

//s- Print out divisor and principle
//cout << "divisor = " << x6 << endl;
//cout << "principal = " << x7 << endl << endl;

//s- if divisor > principal Done
//N-Done divisor > principal ?
38    blt t2,t1,20

//s- remainder = principal
39    addi remainder,principal,0

```

Next:

```
//s- Increment times
```

```
40    addi times,times,1
```

```
//s- remainder = remainder - divisor
```

```
41    sub remainder,remainder,divisor
```

```
42    bge remainder,divisor,-8
```

```
//N-Next remainder > divisor
```

Done:

```
//cout << "Result:" << endl;
```

```
//s- Print out times and remainder
```

```
//cout << "Times = " << x28 << endl;
```

```
//cout << "Remainder = " << x30 << endl;
```

```
43    addi t2,remainder,0
```

```
44    addi t1,times,0
```

```
//x-
```

```
45    jalr zero,t0,0    // return
```

```
}
```

```
// number is prime if no lower prime divides it evenly
```

```
int _CheckIfPrime() //a-3
```

```
{
```

```
46    sw t0,0(sp)      // Save Return
```

```
47    addi sp,sp,4
```

```
// ro implies read only
```

```
// inherit from GeneratePrimes()
```

```
assign x11 ro.firstPrimePtr
```

```

assign x12 ro.lastPrimePtr
assign x13 ro.number

//return to GeneratePrimes(), 1 = true = prime, 0 = not prime
assign t1 results

// AdividesB subroutine interface
//input
assign t1 in.divisor
assign t2 in.principal

//results
assign t1 op.times
assign t2 op.remainder

//working registers used
assign x16 PrimePtr

//s- Test number against all primes before
//s- number in x13

//s- load prime array ptr
48    addi PrimePtr,ro.firstPrimePtr,0

test:
//s- load prime
49    lw in.divisor,0(t0tr)
//cout << "PrimePtr = " << x16 << endl;
//cout << "Dividing by " << x6 << endl;

```



```

50    addi in.principal,ro.number,0// t2 = ro.number to check if prime

//N-NotPrime Prime a factor of number?
_AdividesB() ; // returns 1 if divides evenly
51    jal t0,-60      // Call to _AdividesB()

52    beq t2,zero,20

//N-test More primes ?
53    addi PrimePtr,PrimePtr,4// next to try

// test while PrimePtr != ro.lastPrimePtr
54    bne PrimePtr,ro.lastPrimePtr,-20// bne test

    Prime:
//s- Set results to 1
55    addi results,zero,1 // return prime number
//s- goto end
56    beq zero,zero,8

    NotPrime:
//s- set results to 0
57    addi results,zero,0 // false
    end:
//s- return results
//x-
58    addi sp,sp,-4
59    lw t0,0(sp)
60    jalr zero,t0,0    // return
}

```

```
//library
```

```
// seteup memory, globals, sp, tp, gp, string storage...
```

```
void _Setup()
```

```
{
```

```
// code setup section
```

```
sp = 0x1000
```

```
61    lui sp, 1          //    macro- sp = 0x1000
```

```
62    addi sp,sp,0
```

```
63    jalr zero,t0,0     // return
```

```
}
```

```
void _Exit()
```

```
{
```

```
64    addi zero,x31,224
```

```
65    jal x1,8           // Call to 8()
```

```
66    jalr zero,t0,0     // return
```

```
}
```

## **PRIME Code Source**

```
void main()
{
    _GeneratePrimes(20); //a-3

    _Exit();
}

void _GeneratePrimes()
{
    assign x10 Max    // passed number
    assign x11 arrayStart
    assign x12 memoryPtr
    assign x13 number
    assign x14 true
    assign x15 numberFound

    // calls:
    // CheckIfPrimes() shares needed registers
    // returns t0 = 1 if prime, t1 = 0 otherwise

    //s- Set number of primes to calculate
    addi Max,t1,0

    //s- Set Prime Memory start
    x11 = 0x100
    addi x12,x11,0
    cout << "Calculating Primes" << endl;

    //s- first prime = 2
    addi number,zero,2
```

```
sw number,0(memoryPtr)
addi memoryPtr,memoryPtr,4 // increment ptr
```

```
cout << x13 << endl;
```

```
//s- numberFound = 1
addi numberFound,zero,1
```

```
//s- number = 1
addi number,zero,1
```

```
Test:
```

```
//s- number = number + 2
addi number,number,2
```

```
//Y-Test Number is Prime ?
_CheckIfPrime();
```

```
addi true,zero,1 // true = 1
bne true,t1,Test
```

```
//s- add as prime
sw number,0(memoryPtr)
addi memoryPtr,memoryPtr,4 // increment ptr
```

```
cout << x13 << endl;
//s- numberFound++
```

```
addi numberFound,numberFound,1
```

```
//Y-Test Max reached ?
```

```
blt numberFound, Max, Test
```

```
cout << "Complete" << endl;
```

```
//x-
```

```
}
```

```
int _AdividesB() //a-5
```

```
{
```

```
// input
```

```
assign t1 divisor //a-
```

```
assign t2 principal //a-
```

```
//uses
```

```
assign t3 times //a-
```

```
assign t4 diff //a-
```

```
assign t5 remainder //a-
```

```
//returns
```

```
x1 = times
```

```
x2 remainder
```

```
//s- zero times and remainder
```

```
addi times,zero,0
```

```
addi remainder,zero,0
```

```
//s- Print out divisor and principle
```

```
//cout << "divisor = " << x6 << endl;
//cout << "principal = " << x7 << endl << endl;
```

```
//s- if divisor > principal Done
//N-Done divisor > principal ?
blt t2,t1,Done
```

```
//s- remainder = principal
addi remainder,principal,0
```

Next:

```
//s- Increment times
addi times,times,1
```

```
//s- remainder = remainder - divisor
sub remainder,remainder,divisor
```

```
bge remainder,divisor,Next
//N-Next remainder > divisor
```

Done:

```
//cout << "Result:" << endl;
//s- Print out times and remainder
//cout << "Times = " << x28 << endl;
//cout << "Remainder = " << x30 << endl;
```

```
addi t2,remainder,0
addi t1,times,0
```

```
//x-
```

```

}

// number is prime if no lower prime divides it evenly
int _CheckIfPrime() //a-3
{
// ro implies read only
// inherit from GeneratePrimes()
assign x11 ro.firstPrimePtr
assign x12 ro.lastPrimePtr
assign x13 ro.number

//return to GeneratePrimes(), 1 = true = prime, 0 = not prime
assign t1 results

// AdividesB subroutine interface
//input
assign t1 in.divisor
assign t2 in.principal

//results
assign t1 op.times
assign t2 op.remainder

//working registers used
assign x16 PrimePtr

//s- Test number against all primes before
//s- number in x13

//s- load prime array ptr

```

```

addi PrimePtr,ro.firstPrimePtr,0

test:
//s- load prime
lw in.divisor,0(PrimePtr)
//cout << "PrimePtr = " << x16 << endl;
//cout << "Dividing by " << x6 << endl;

addi in.principal,ro.number,0    // t2 = ro.number to check if prime

//N-NotPrime Prime a factor of number?
_AdividesB() ; // returns 1 if divides evenly

beq t2,zero, NotPrime

//N-test More primes ?
addi PrimePtr,PrimePtr,4        // next to try

// test while PrimePtr != ro.lastPrimePtr
bne PrimePtr,ro.lastPrimePtr, test

Prime:
//s- Set results to 1
addi results,zero,1 // return prime number
//s- goto end
beq zero,zero, end

NotPrime:
//s- set results to 0
addi results,zero,0 // false
end:

```



```
//s- return results
```

```
//x-
```

```
}
```

## Introduction to Global Variables and Structures

In programming, variables and structures are stored in memory and can be accessed by multiple procedures without needing to be declared locally in each one. This allows for shared data across different parts of a program.

### What is a Structure?

A structure (or struct) is a user-defined data type that groups related variables together under one name. Each variable inside a structure is called a member or field.

Structures are useful for organizing data logically and representing complex entities in a program.

### Why Use Structures?

- Group related variables into a single type for better organization.
- Store mixed data types (e.g., int, bool, char[]) in one container.
- Pass complex data more easily to functions.
- Model real-world entities in a modular, maintainable way.

### Current Limitation

At present, globally defined variables in this system must be of type int.

### Example

```
#set_address 0x1000

int a;
int b;
int lengths[8];

struct MyStruct {
    int x;
    int y;
    int values[8]; ;
};

struct TaskControlBlock[10] {
    // CPU Context
    int stack_pointer;    // Saved Stack Pointer (SP)
    int ra;               // Return Address (PC to resume execution)

    // Callee-saved registers (s0-s11)
    int s[12];           // Saved registers during context switch

    int stack_base_pointer; // Pointer to base of task's stack
    int task_id;           // Unique Task Identifier
    int state;             // Task State (e.g., READY, RUNNING, BLOCKED)

    // Scheduling Links
    int nextControlBlock_pointer; // Pointer to next TCB in scheduler list
};

#end
```

## Load a memory value

In RISC-V, to load a value from memory into a register, you use a load instruction. These follow the general format:

**lw rd, offset(rs1)**

Where:

- `lw` = Load Word (32 bits)
- `rd` = destination register
- `offset(rs1)` = memory address formed by adding the offset to the contents of register rs1

Common load instructions:

Instruction	Description	Size Loaded
----- ----- -----		
`lb`	Load Byte	8 bits
`lw`	Load Word	32 bits

## Store a memory value

**sw rd, offset(rs1)**

Where:

- `sw` = Store Word (32 bits)
- `rd` = destination register
- `offset(rs1)` = memory address formed by adding the offset to the contents of register rs1

STORE instructions:

Instruction	Description	Size Loaded
----- ----- -----		
`sb`	Load Byte	8 bits
`sw`	Load Word	32 bits

## Macros for accessing memory values

### macro for load global contents at address a

LOAD t0, a

If the address were, say, 0x12345678, the expansion would be:

```
lui    t0, 0x12345    # Upper 20 bits
addi   t0, t0, 0x678  # Add low 12 bits
lw     t0, 0(t0)
```

### macro for setting global contents at address a

Store t0, a

If the address were, say, 0x12345678, the expansion would be:

```
lui    x2, 0x12345    # Upper 20 bits
addi   x2, t0, 0x678  # Add low 12 bits
sw     t0, 0(x2)
```

## Multi-core processors

RISC-V chips with 8 or more CPU cores represent a major leap forward in the open hardware movement. These multicore processors are no longer limited to research environments or embedded systems but are now pushing into high-performance applications such as servers, edge computing, and AI workloads. Designs like the SiFive Performance P670 and Alibaba's Xuantie series showcase how RISC-V can scale to meet the demands of modern parallel processing while maintaining the instruction set architecture's hallmark simplicity and extensibility.

The importance of these high-core-count RISC-V chips lies in their ability to offer an open, customizable alternative to proprietary processors from dominant players like Intel and ARM. This openness fosters innovation by allowing companies, researchers, and governments to develop and optimize chips for specific use cases without being locked into expensive licensing schemes. Furthermore, as computing needs continue to grow—particularly in data centers, autonomous systems, and IoT backbones—RISC-V's scalable, open architecture positions it as a key enabler of future technologies across a wide range of industries.

## Assembly and Multi-core Programming

Writing software in assembly language allows developers to orchestrate precise interactions across multiple CPUs because it exposes the hardware's lowest-level capabilities—such as explicit control of cache coherency, inter-processor interrupts, memory fences, and atomic operations. By manipulating these primitives directly, an engineer can build finely tuned synchronization routines, tailor thread-to-core affinity, and optimize shared-memory access patterns, ensuring that work is efficiently partitioned and coordinated among all cores. This granular control is crucial for squeezing maximum parallel performance out of multicore systems where higher-level languages may hide or abstract away the deterministic behavior required for ultra-low-latency or real-time workloads.

# Advanced Risc Coding

## Introduction

The rise of IoT (Internet of Things) has led to a growing demand for lightweight, efficient, and customizable operating systems tailored for embedded and resource-constrained devices. One compelling approach to building such systems is to design a Unix-like OS specifically for IoT applications, leveraging the RISC-V architecture. By integrating common Unix utilities into the development process, learners can deepen their understanding of low-level systems programming while also creating practical, functional tools for embedded environments.

## Advantages of a Unix-like OS in IoT

Unix-like operating systems are known for their modularity, simplicity, and robustness. For IoT applications, these characteristics are beneficial in several ways:

- **Modularity**: Developers can include only the components they need, reducing memory and processing overhead.
- **Simplicity**: A minimalistic design eases debugging and enhances security.
- **POSIX Compliance**: Ensures code portability and reuse of a vast array of existing tools and libraries.
- **Educational Value**: Understanding Unix internals provides a solid foundation for advanced system-level programming.

## Coding Unix Utilities in RISC-V

Re-implementing standard Unix utilities (such as `ls`, `cat`, `grep`, `echo`, etc.) in RISC-V assembly or C is an effective way to learn advanced coding concepts. These utilities provide real-world use cases for practicing system calls, file I/O, memory management, and text processing at a low level. By coding these utilities from scratch, students can:

- Gain hands-on experience with the RISC-V ISA (Instruction Set Architecture).
- Understand how operating system abstractions are built upon hardware.
- Learn optimization techniques for constrained IoT environments.
- Develop a deeper appreciation for the simplicity and power of Unix philosophy ('Do one thing well').

## Learning Outcomes and Real-World Application

The project of building a Unix-like OS and utilities for IoT devices serves both as an educational journey and a pathway to real-world development skills. Participants will acquire:

- Proficiency in low-level systems programming.
- Insight into OS kernel design and device drivers.
- Practical knowledge of cross-compilation and toolchains.
- Experience in optimizing software for limited-resource devices.

Moreover, such a system could evolve into a practical microcontroller or edge computing platform, offering a customizable and open-source alternative to proprietary IoT ecosystems.

## Conclusion

Creating a Unix-like OS tailored for IoT applications, while developing core Unix utilities using RISC-V, is a challenging yet rewarding project. It bridges the gap between academic learning and practical system-level programming. This approach not only deepens one's coding skills but also contributes to the growing open-source movement in embedded systems.

For the multitasking example an interrupt service will be call every n cycles

```
INTERRUPT _switchTask 5000
```

## Register Convention use

Most assemblers (and GCC's inline assembly) pick **t1** (x6) as the default scratch register for things like computing an address in a **STORE** macro, because:

- **t0** is often used for intermediate values in instruction sequences.
- **t1** is rarely reserved for anything special.
- It's still in the "temporary" range so no save/restore is needed.

We are not following this

by convention **t0** is temporary register. Using **x2** for now.

In RISC-V, the **general-purpose integer registers** are named according to their *usage convention* rather than just numbers. There are 32 registers (x0-x31), and they fall into functional groups that the RISC-V calling convention defines.

---

### 1. t registers (temporary)

- **Names:** t0-t6
  - **Numbers:** x5-x7, x28-x31
  - **Purpose:**
    - Used for temporary values.
    - **Caller-saved** – meaning a function *does not* need to preserve them before using them; if you care about their value, you must save them yourself before making a function call.
  - **Good for:**
    - Intermediate results
    - Short-lived values
    - Holding scratch addresses for memory access
- 

### 2. s registers (saved)

- **Names:** s0-s11
- **Numbers:** x8-x9, x18-x27
- **Purpose:**



- **Callee-saved** – a function *must* save and restore these if it changes them, so the caller can rely on them being unchanged after a call.
  - **Good for:**
    - Variables that must persist across function calls
    - Keeping important state like base pointers, loop indices across calls
- 

### 3. a registers (arguments / return values)

- **Names:** a0–a7
  - **Numbers:** x10–x17
  - **Purpose:**
    - Pass arguments to functions (a0–a7 for up to 8 integer arguments).
    - a0 and a1 also hold return values.
  - **Caller-saved** like t registers.
- 

### 4. Special registers

- **x0** – Always zero (writes are ignored, reads always give 0).
  - **x1 / ra** – Return address.
  - **x2 / sp** – Stack pointer.
  - **x3 / gp** – Global pointer.
  - **x4 / tp** – Thread pointer.
- 

#### **Convention for address registers in stores**

There's **no hardware-enforced rule** for which register to use when holding a memory address – it's up to your code and ABI conventions.

However:

- For **short-lived scratch addressing**, it's common to use a t register (e.g., t0) so you don't have to save/restore.
- If the address must survive across function calls, you'd typically use an s register (e.g., s1).
- gp (x3) is sometimes used for accessing global/static data with small offsets.

## Raspberry Pi and Ubuntu

The Raspberry Pi is a small, affordable single-board computer developed by the Raspberry Pi Foundation to promote computer science education and low-cost computing. Over the years, it has evolved from a basic educational tool into a powerful platform for hobbyists, developers, and even industrial applications. The Raspberry Pi 5 (Pi 5), released in 2023, is a significant upgrade over its predecessors, featuring a 2.4GHz quad-core ARM Cortex-A76 CPU, faster RAM, PCIe expansion support, and better I/O capabilities. It delivers performance comparable to entry-level desktop PCs. The newer Raspberry Pi 500, a compact all-in-one keyboard computer, builds on the Pi 5's capabilities and packages them into a convenient form factor, making it ideal for use as a lightweight desktop or workstation.

Ubuntu, a popular Linux-based operating system, is fully compatible with both the Raspberry Pi 5 and Pi 500. Canonical, the company behind Ubuntu, offers official support through Ubuntu Desktop and Ubuntu Server builds optimized for ARM architecture. Ubuntu provides a familiar, user-friendly environment with access to a vast software repository, making it an excellent choice for development, education, and daily computing on Raspberry Pi. With its powerful command-line tools, built-in security features, and wide support for open-source applications, Ubuntu allows Raspberry Pi users to run web servers, write code, manage media centers, or even experiment with AI and robotics. This compatibility extends the Pi's usefulness well beyond its modest size and price.


## Raspberry Pi Micro SDXC Memory Card Offer

 Special Offer for Raspberry Pi Owners! 

Unlock the full potential of your Raspberry Pi 5 or Pi 500 with our pre-configured microSDXC cards, ready to boot and packed with performance! Each card comes fully loaded with Ubuntu Desktop optimized for Raspberry Pi, along with essential tools, updates, and drivers—so you can get started immediately, no setup required.

- ◆ High-speed 32G Class A2 microSDXC
- ◆ Plug-and-play: pre-installed Ubuntu + performance tweaks
- ◆ Web server with our website LearnRisc preinstalled, Access by browser at localhost:8000
- ◆ Perfect for coding
- ◆ High speed execution, no shared access
- ◆ Bonus Terminal based simulator and Assembler
- ◆ Howto video included

Order now and level up your Pi experience—hassle-free!

 Limited-time price starting at just \$29.95!