Computer Programming

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

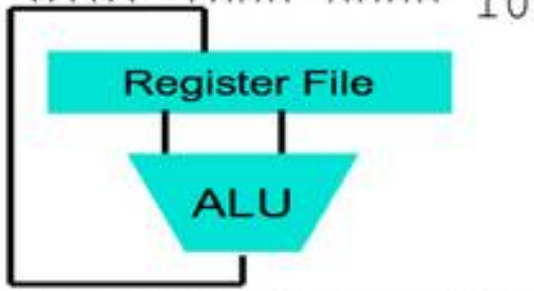
Logic Circuit Description (Circuit Schematic Diagrams)

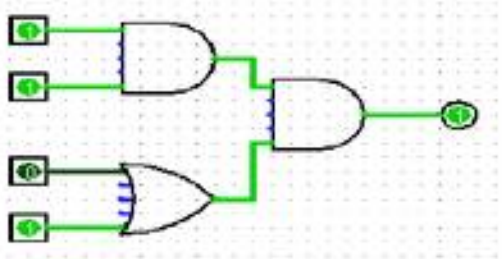
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw      $t0, 0($2)
lw      $t1, 4($2)
sw      $t1, 0($2)
sw      $t0, 4($2)
```

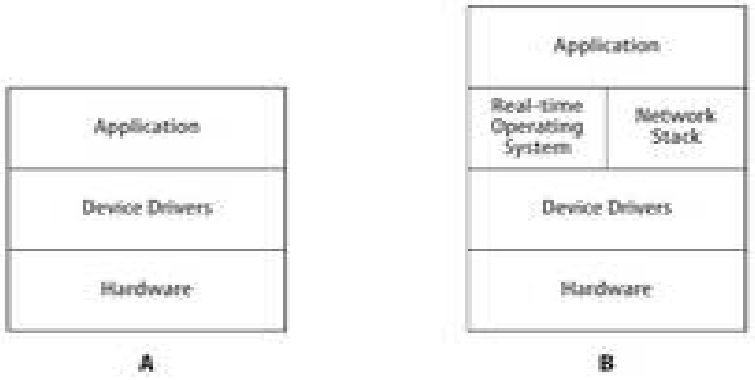
Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





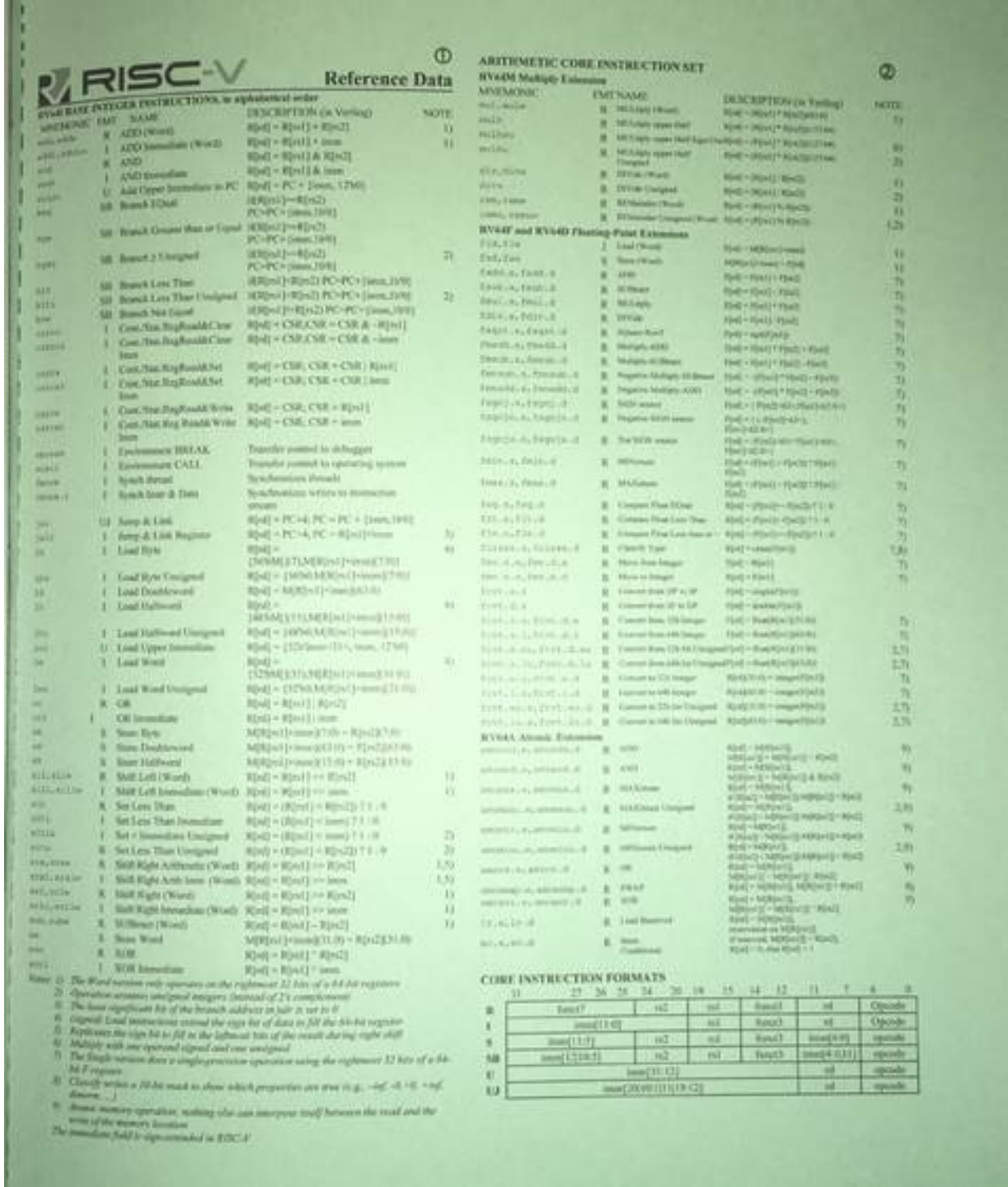
Requirements: Basic and Complex



Criterion	Low	Medium	High
Processor	4- or 8-bit	16-bit	32- or 64-bit
Memory	< 64 KB	64 KB to 1 MB	> 1 MB
Development cost	< $100,000	$100,000 to $1,000,000	> $1,000,000
Production cost	< $10	$10 to $1,000	> $1,000
Number of units	< 100	100 to 10,000	> 10,000
Power consumption	> 10 mW/MIPS	1 to 10 mW/MIPS	< 1 mW/MIPS
Lifetime	Days, weeks, or months	Years	Decades
Reliability	May occasionally fail	Must work reliably	Must be fail-proof

Instruction Set Architecture

Instruction Set Architecture (ISA) serves as the interface between software (e.g., high-level programming languages) and hardware (digital systems like CPUs or processors).
It defines the set of instructions that a processor can execute, along with the corresponding machine-level operations.



Processor Memory Map: Hardware and Software Placement

- **The processor's memory map is critical for defining how different memory regions and peripherals are addressed. It resides in both hardware and software, serving distinct roles in each domain.**

Hardware Placement of Memory Map

In hardware, the memory map is implemented as part of the address decoding logic, typically located within the memory controller module or the interconnect/bus fabric.

Address Decoding Logic:

Encodes address ranges in comparators or lookup tables.

Routes requests based on address ranges.

Placement in HDL:

Found in the memory controller module or bus arbiter module.

Implements logic for decoding incoming address lines and routing requests to the correct memory region.

Address Range	Memory Region
0x0000_0000 – 0x0000_FFFF	Local Scratchpad Memory
0x0001_0000 – 0x0001_FFFF	Cache
0x1000_0000 – 0x1FFF_FFFF	Main Memory

Hardware Implementation (Verilog):

```
always @(*) begin
    if (address >= 32'h0000_0000 && address <= 32'h0000_FFFF) begin
        memory_select <= SPM; // Scratchpad Memory
    end else if (address >= 32'h1000_0000 && address <= 32'h1FFF_FFFF) begin
        memory_select <= MAIN_MEM; // Main Memory
    end else begin
        memory_select <= NONE; // Invalid Address
    end
end
```

Software: Memory Map

In software, the memory map is defined in **header files**, configuration files, or linker scripts for firmware and OS use. This ensures that software developers can access hardware components and memory regions using predefined constants.

Header Files:

Define base addresses and address ranges as macros or constants.

Example (C Header File):

```
#define SPM_BASE_ADDR   0x00000000  // Scratchpad Memory Base
#define MAIN_MEM_ADDR   0x10000000  // Main Memory Base
#define CACHE_ADDR      0x00010000  // Cache Bas
```

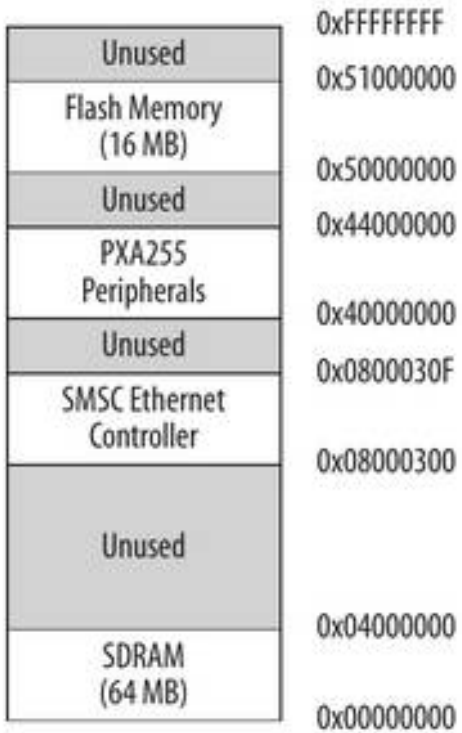
Linker Script:

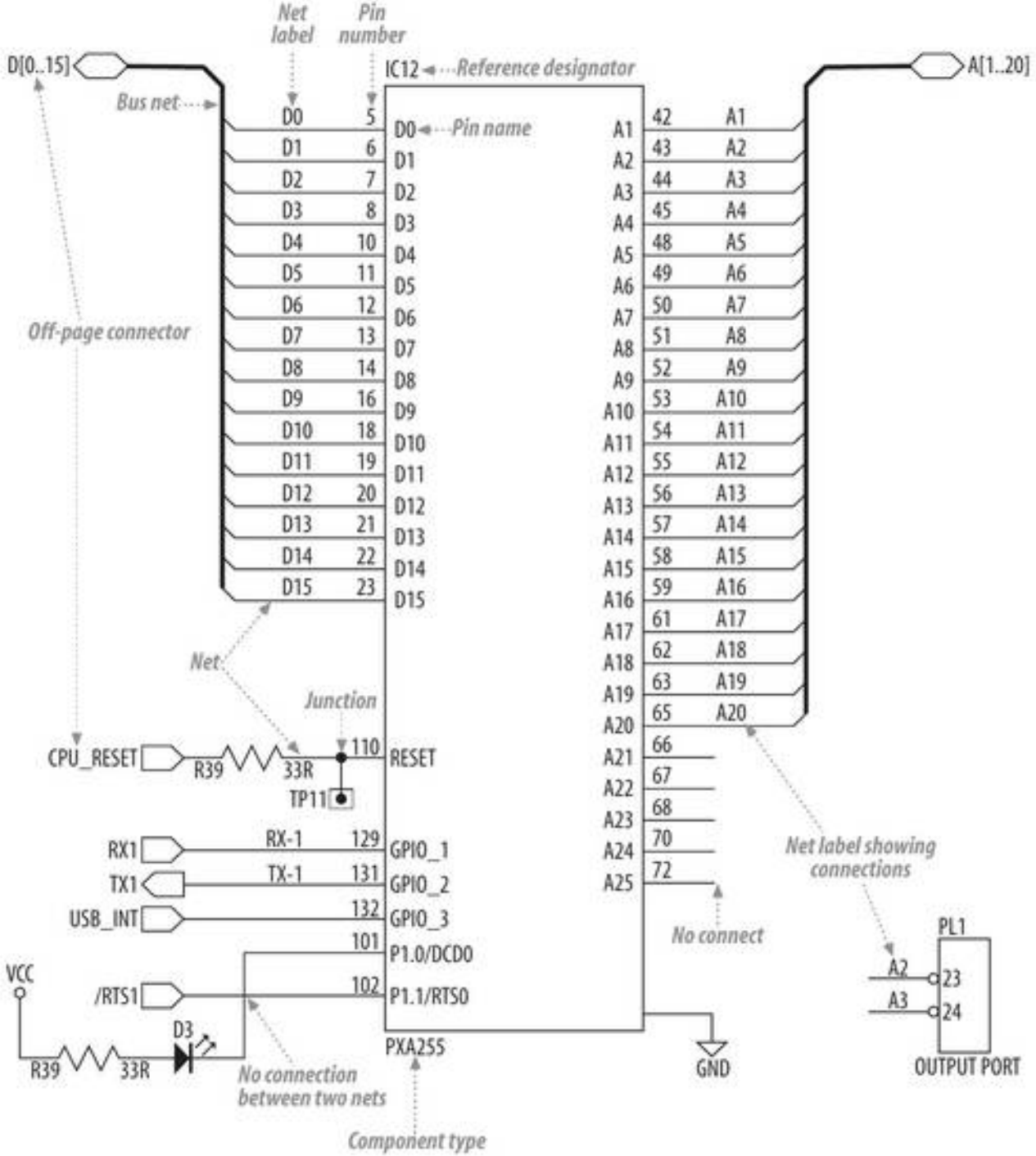
Specifies physical memory layout and assigns sections for code and data.

Example (Linker Script):

```
MEMORY
{
   SPM (rwx) : ORIGIN = 0x00000000, LENGTH = 64K
   MAIN_MEM (rwx) : ORIGIN = 0x10000000, LENGTH = 512M
}
SECTIONS
{
   .text : { *(.text) } > SPM
   .data : { *(.data) } > MAIN_MEM
}
```

System Schematic and Memory Mapping





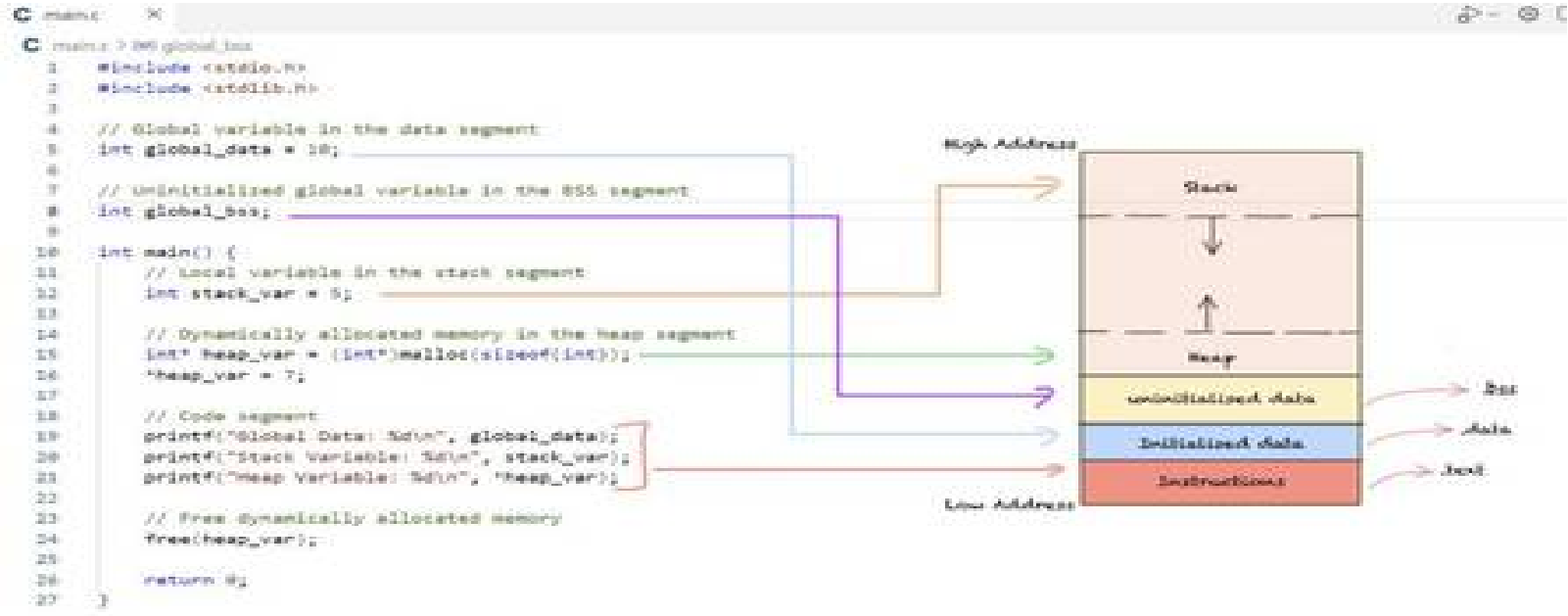
Define Memory Address

```
/* Timer Registers */
#define TIMER_0_MATCH_REG          (*((uint32_t volatile *)0x40A00000))
#define TIMER_1_MATCH_REG          (*((uint32_t volatile *)0x40A00004))
#define TIMER_2_MATCH_REG          (*((uint32_t volatile *)0x40A00008))
#define TIMER_3_MATCH_REG          (*((uint32_t volatile *)0x40A0000C))
#define TIMER_COUNT_REG            (*((uint32_t volatile *)0x40A00010))
#define TIMER_STATUS_REG           (*((uint32_t volatile *)0x40A00014))
#define TIMER_INT_ENABLE_REG       (*((uint32_t volatile *)0x40A0001C))

/* Timer Interrupt Enable Register Bit Descriptions */
#define TIMER_0_INTEN              (0x01)
#define TIMER_1_INTEN              (0x02)
#define TIMER_2_INTEN              (0x04)
#define TIMER_3_INTEN              (0x08)

/* Timer Status Register Bit Descriptions */
#define TIMER_0_MATCH              (0x01)
#define TIMER_1_MATCH              (0x02)
#define TIMER_2_MATCH              (0x04)
#define TIMER_3_MATCH              (0x08)

/* Interrupt Controller Registers */
#define INTERRUPT_PENDING_REG      (*((uint32_t volatile *)0x40D00000))
#define INTERRUPT_ENABLE_REG       (*((uint32_t volatile *)0x40D00004))
#define INTERRUPT_TYPE_REG         (*((uint32_t volatile *)0x40D00008))

/* Interrupt Enable Register Bit Descriptions */
#define GPIO_0_ENABLE              (0x00000100)
#define UART_ENABLE                (0x00400000)
#define TIMER_0_ENABLE             (0x04000000)
#define TIMER_1_ENABLE             (0x08000000)
#define TIMER_2_ENABLE             (0x10000000)
#define TIMER_3_ENABLE             (0x20000000)
```

```
/* General Purpose I/O (GPIO) Registers */
#define GPIO_0_LEVEL_REG           (*((uint32_t volatile *)0x40E00000))
#define GPIO_1_LEVEL_REG           (*((uint32_t volatile *)0x40E00004))
#define GPIO_2_LEVEL_REG           (*((uint32_t volatile *)0x40E00008))
#define GPIO_0_DIRECTION_REG       (*((uint32_t volatile *)0x40E0000C))
#define GPIO_1_DIRECTION_REG       (*((uint32_t volatile *)0x40E00010))
#define GPIO_2_DIRECTION_REG       (*((uint32_t volatile *)0x40E00014))
#define GPIO_0_SET_REG             (*((uint32_t volatile *)0x40E00018))
#define GPIO_1_SET_REG             (*((uint32_t volatile *)0x40E0001C))
#define GPIO_2_SET_REG             (*((uint32_t volatile *)0x40E00020))
#define GPIO_0_CLEAR_REG           (*((uint32_t volatile *)0x40E00024))
#define GPIO_1_CLEAR_REG           (*((uint32_t volatile *)0x40E00028))
#define GPIO_2_CLEAR_REG           (*((uint32_t volatile *)0x40E0002C))
#define GPIO_0_FUNC_LO_REG         (*((uint32_t volatile *)0x40E00054))
#define GPIO_0_FUNC_HI_REG         (*((uint32_t volatile *)0x40E00058))
```

Computer Programming and Memory Layout

- Understanding C memory layout is crucial for debugging, optimizing performance, security and interfacing with low-level systems.
- **Text (Code) Segment**:
- **Data Segment**:
- **BSS Segment**:
- **Heap Segment**:
- **Stack Segment**:



• Text (Code), Data and BSS Segment:

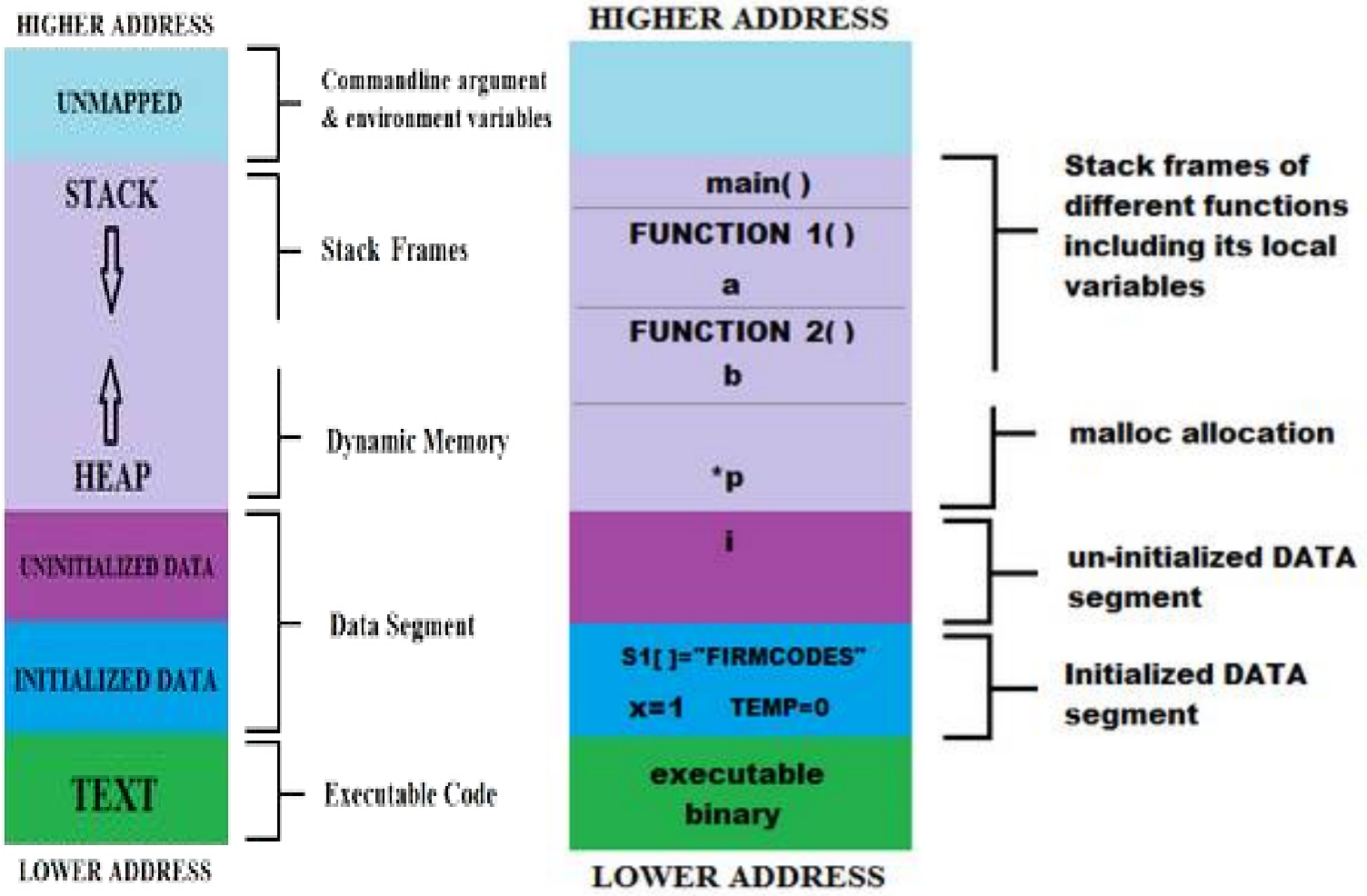
- The text segment contains the executable code of the program. It is read-only and holds the instructions for the program.
- The data segment contains initialized global and static variables. In the example code, global_data is an initialized global variable with value 10.
- The BSS (Block Started by Symbol) segment contains uninitialized global and static variables. The BSS segment is set to zero during program startup. In the example code, global_bss variable will be added to the bss section by linker.
- The Text, Data, and BSS segments collectively form the static part of the program that contains fixed-sized instructions and data that persists throughout its execution. These should be kept in a non-volatile memory to ensure successful execution of code following a power cycle.
- You can use the size utility that comes with the compiler to get the size of the executable. Below is the output for the example code:

- text data bss dec hex filename
- 1585 600 8 2193 891 main.out

Heap and Stack Segments

- **Heap Segment**:
- The heap segment is used for dynamic memory allocation during the program's runtime. In the example, we allocate memory for an integer using malloc(), and heap_var points to the newly allocated memory location.
- It's important to free the allocated memory after it is no longer needed.
- Over time, repeated memory allocation without freeing memory can cause the program's memory usage to grow unnecessarily leading to poor performance and runtime allocation failures.

- **Stack Segment**:
- The stack segment is used for managing function calls, local variables, and function call frames. In the example, stack_var is a local variable that will be allotted on the stack during the execution of the main() function.
- The stack and heap memory share the dynamic memory area of the program. The stack typically starts from the end address of the memory and grows downward, while the heap starts from the end of the BSS segment.




```c
#include<stdio.h>
#include<malloc.h>

void FUNCTION_1();
void FUNCTION_2();

char S1[]="FIRMCODES";  //initialized read-write area of DATA segment
int i;                  //uninitialized DATA segment
const int x=1;          //initialized read-only area of DATA segment

int main()
{
    static int TEMP=0;  //uninitialized DATA segment

    char *p=(char*)malloc(sizeof(char)); //Heap segment

    FUNCTION_1();       //FUNCTION_1 stack frame

    return 0;
}

void FUNCTION_1()
{
    int a;          //initialized in stack frame of FUNCTION_1

    FUNCTION_2();   //FUNCTION_2 stack frame
}

void FUNCTION_2()
{
    int b;          //initialized in stack frame of FUNCTION_2
}
```

- Instruction Section: Contains the compiled machine code instructions (text section).
- Data Section: Contains initialized data (data section).
- The linker combines the code and data sections, resolves symbols, and sets up memory addresses.
- The linker script defines how different sections are mapped into the memory of the microcontroller.
- It specifies memory regions and assigns addresses to different sections of the code and data.

```
MEMORY
{
    ROM (rx) : ORIGIN = 0x08000000, LENGTH = 512K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
    .text : {
        *(.text)
    } > ROM

    .data : {
        *(.data)
    } > RAM
}
```

Linker Script: Program and Data Memory Allocation

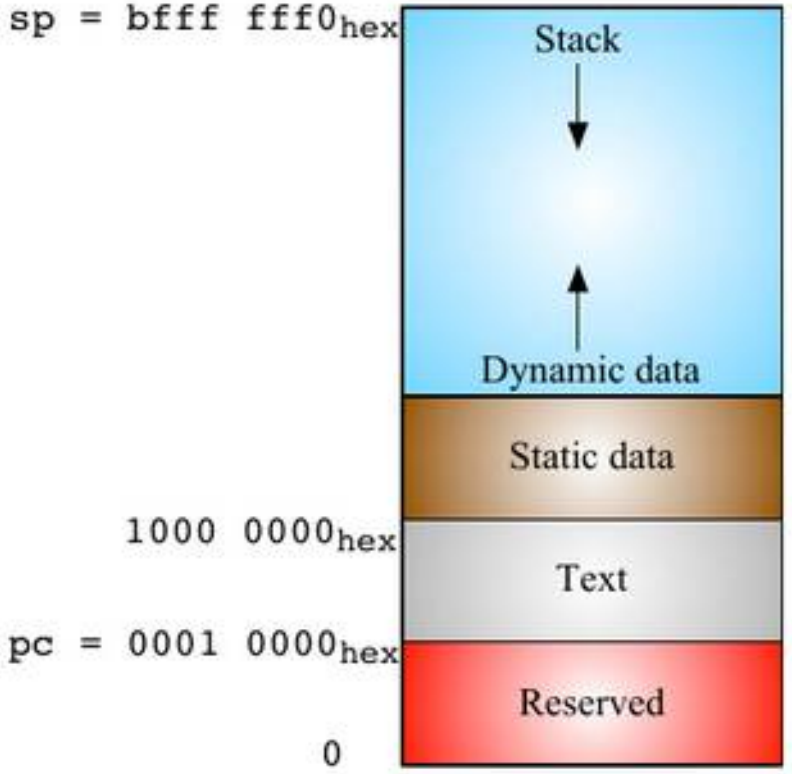
The high addresses are the top of the figure and the low addresses are the bottom.

The stack pointer (sp) starts at BFFF FFF0 hex and grows down toward the Static data.

The text (program code) starts at 0001 0000hex and includes the statically-linked libraries.

The Static data starts immediately above the text region; in this example, we assume that address is 1000 0000hex .

Dynamic data, allocated in C by malloc(), is just above the Static data. Called the heap, it grows upward toward the stack. It includes the dynamically-linked libraries.

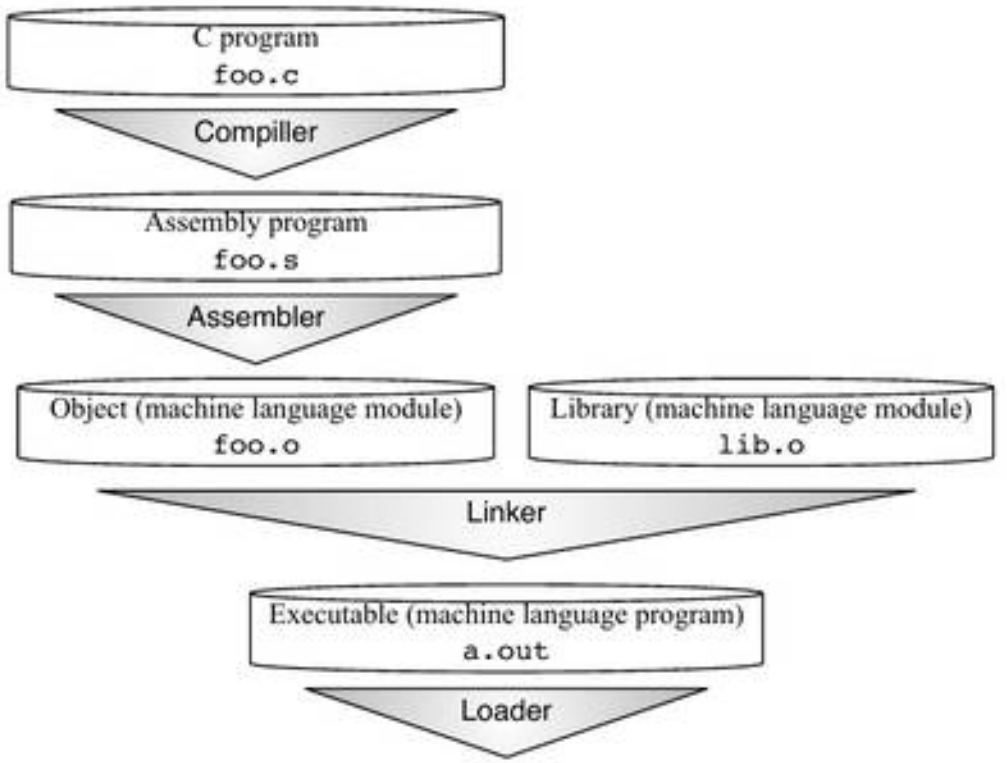


Assembly or C/C++

- Write Efficient Code
- Secure Application
- Multi-Threaded and Complex Program to run multiple devices (OS)
- Real-Time Applications for Real world Problems

Assembly Programming

- Unlike HLL like C or Java, assembly does not have *variables* as you know and love them
  - More primitive, closer what simple hardware can directly support
- Assembly operands are objects called registers
  - Limited number of special places to hold values, built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 ns - light travels 1 foot in 1 ns!!!)



RISCV GCC Assembler

```
    .section .data          # Data section (if needed)
    .section .text          # Code section
    .globl _start           # Define global start label

_start:
    li s3, 0x1121           # Load immediate value 0x1121 into s3
    li rs2, 0x22233         # Load immediate value 0x22233 into rs2
    add rd, rs1, rs2        # Add rs1 and rs2, store the result in rd
    sw rd, 0x1(zero)        # Store the value in rd to memory address 0x00000001
    j _exit                 # Jump to exit (optional, depending on context)

_exit:
    nop                 # No operation (placeholder for termination)
```

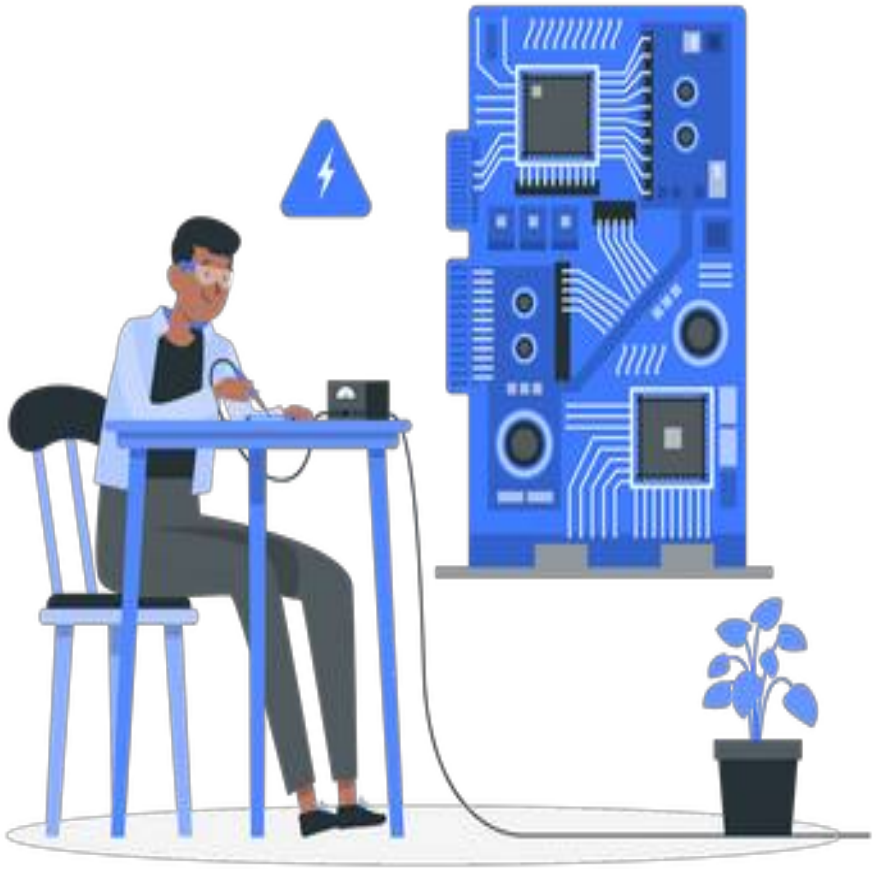
Introduction to C Programming

Standard C (often just called "C") is a programming languages used to write software, but they differ in their target environments, constraints, and some aspects of functionality.

C is one of the most widely used languages for programming systems based on the RISC-V architecture. It provides a high-level abstraction while still allowing for low-level hardware manipulation. Here's a detailed introduction to programming in C for RISC-V.

Standard C is sufficient for most use cases, especially when working with a RISC-V system that supports an operating system.

Specialized C (hardware-specific code, inline assembly, and custom startup code) is required for bare-metal programming or hardware accelerators.



Introduction to Embedded C Programming

VS

Target Environment

A structural and programming language used by developers to create desktop-based applications

An extension of C primarily used to develop microcontroller based applications.

Memory Constraint

Typically used on systems with more resources.

Often used in environments with limited resources (memory, processing power).

Hardware Interaction

Hardware interactions are managed by operating system or libraries, unless used in system-level programming.

Interacts directly with hardware components, such as registers, I/O ports, and peripheral devices.

Libraries and Extensions

Uses standard libraries provided by the C standard library (e.g., stdio.h, stdlib.h) and other platform-specific or third-party libraries.

Uses specialized libraries and extensions for embedded systems (e.g., specific APIs for handling hardware interrupts, timers, and serial communication).

Introduction to Embedded C Programming

VS

Development Tools

Typically uses general-purpose IDEs (e.g., Visual Studio, Eclipse) and compilers (e.g., GCC, Clang).

Specific Integrated Development Environments (IDEs), compilers, and debuggers designed for embedded system development (e.g., Keil, IAR, MPLAB).

Real-Time Constraint

It can be used in real-time applications, but it is not inherently designed for real-time constraints and may rely on external real-time extensions or operating systems.

Often used in real-time systems where meeting timing constraints is crucial. It may include real-time operating systems (RTOS) or bare-metal programming.

Code Portability

Code is generally more portable across different platforms, adhering to the C standard.

Code is often less portable due to hardware-specific dependencies and optimizations. Porting code between different embedded platforms can be challenging.

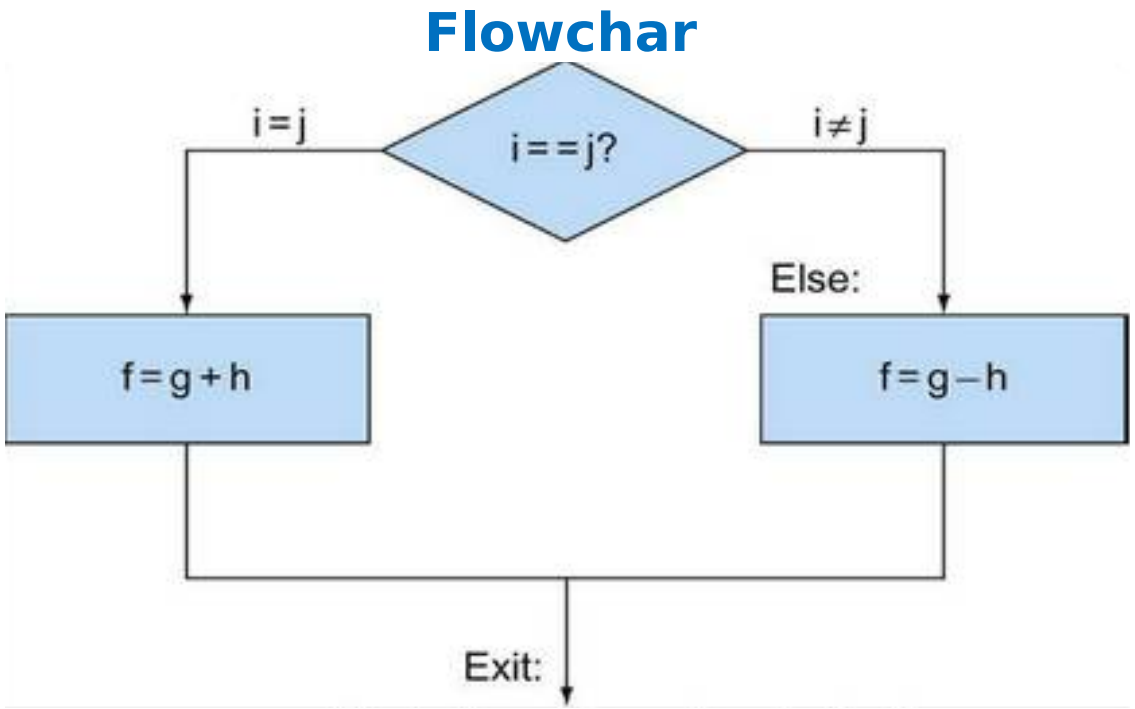
```
// example.c
int global_var = 10;
int main() {
    int local_var = 5;
    int result = global_var +
local_var;
    return result;
}
```

riscv32-unknown-elf-gcc example.o -o example

- The compiler generates an object file in ELF format. This object file contains machine code, data, and metadata, organized into different sections like .text (code), .data (initialized data), and .bss (uninitialized data).

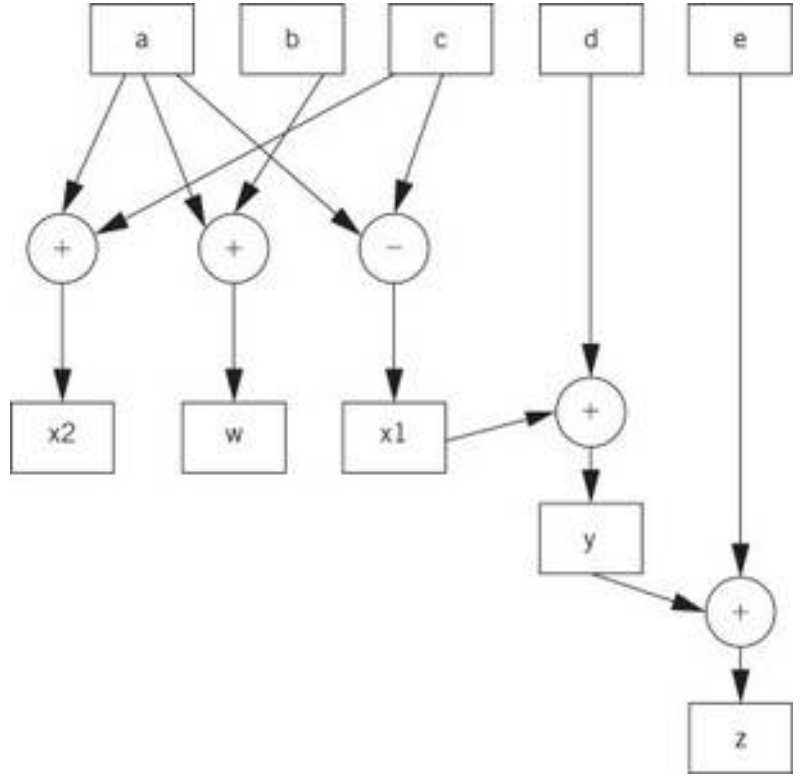
Programming RISC-V

- Problem
- Write it in your own words
- Make Pseudo Code
- Create Control and Data-flow Graph
- Program (C/C++, ASM)
- Debug
- Profile
- Optimize/Fine Tune
- Execute
- Test



Hazards

- Data Hazards: Instructions are waiting for data from other instructions.
- Control Hazards: Changes in instruction flow cause delays.
- Structural Hazards: Limited hardware resources cause delays.



Testing and Executing the Code

RIPES

https://ripes.me/

https://github.com/mortbopet/Ripes/releases/download/v2.2.6/Ripes-v2.2.6-win-x86_64.zip

Next:

RISCV Micro Controller

RISCV Simulator and Emulators

RISCV Single Board Computer

Programmer and Debugger

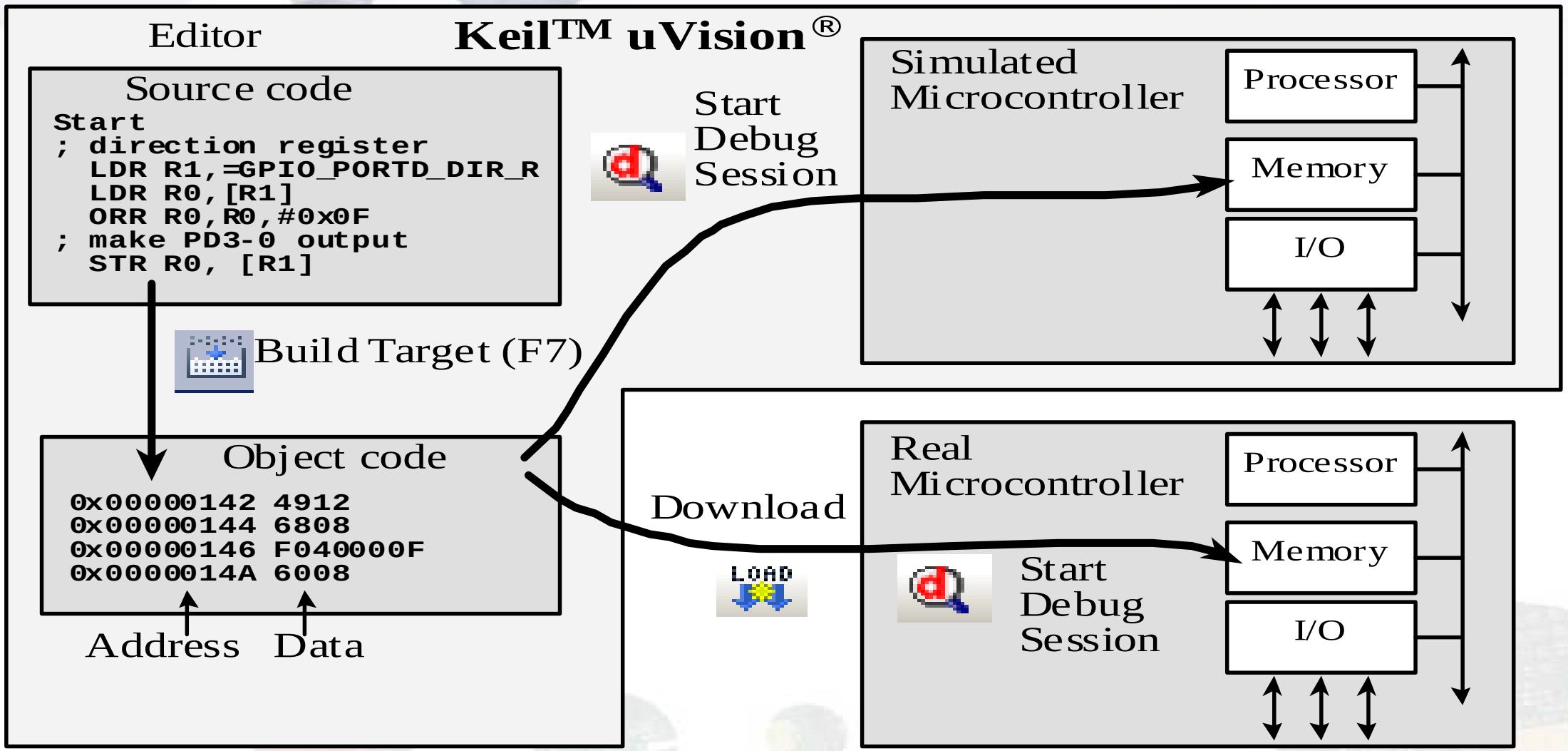
- Programmer or debugger tool to flash the firmware into the RISCV System.
  - Instruction Memory: The code from the .text section is loaded into the system instruction memory.
  - Data Memory: The initialized data from the .data section is loaded into the system data memory.

- Target Hardware Architecture:
  - Processor and Specifications:
  - Program Memory and Data Memory Size:
  - Peripherals and Components
- Memory Mapping
- Software Development
  - GCC Compiler: Compiler: riscv32-unknown-elf-gcc or riscv64-unknown-elf-gcc.
  - Debugger: GDB with RISC-V support.
  - ELF Loader: OpenOCD or RISC-V Proxy Kernel.

Stress Checking and Profiling Tools for RISC-V:

- RISC-V Performance Monitor or Perf.

SW Development Environment



Compiler Options

- riscv32-unknown-elf-gcc //
  -march=rv32imac // Architecture and ISA Extensions:
  -mabi=ilp32 // ABI (Application Binary Interface: Int, long, pointer):
  -O2 // Optimization Levels:
  -mtune=sifive-e31 // Code Genartion for specific RISCV core
  -g // Debugging and Profiling -pg
  mhard-float // Floating Point Options: Hard/Soft Floting point:
- -T linker_script.ld // -T: Specify a linker script.
  -I/path/to/include // Include Paths and Libraries
  -L/path/to/li //
  -o output.elf // Output file
  source.c // source file
  -lm // -lm (math library)
- -funroll-loops // Loop Unrolling option

C code for testing

```
void main(void) {
    int a = 5, b = 10;
    int result = a + b;
  }
```

Steps: Code Compilation to Execution

- riscv32-unknown-elf-gcc -march=rv32i -S -o riscv.s ./code.c
- riscv32-unknown-elf-as -march=rv32i -o riscv.o ./riscv.s
- riscv32-unknown-elf-ld -o riscv ./riscv.o
- riscv32-unknown-elf-objcopy -O binary --only-section=.text riscv instr.mem
- riscv32-unknown-elf-objcopy -O binary --only-section=.data riscv data.mem
- riscv32-unknown-elf-objdump -D -b binary -m riscv:rv32i instr.mem

```
with open("instr.mem", "rb") as file:
    content = file.read()
    hex_data = content.hex()  # Convert the binary content to hexadecimal

    # Split the hexadecimal data into 4-byte (32-bit) chunks
    instructions = [hex_data[i:i+8] for i in range(0, len(hex_data), 8)]

    # Reverse the byte order for each instruction (little-endian to big-endian)
    for instruction in instructions:
        # Reverse the byte order by grouping the hex string in chunks of 2 (1 byte) and reversing the
order
        big_endian_instruction = ''.join([instruction[j:j+2] for j in range(0, len(instruction), 2)][::-1])
        print(big_endian_instruction)
```

- 130101fe
- 232e8100
- 13040102
- 93075000
- 2326f4fe
- 9307a000
- 2324f4fe
- 0327c4fe
- 832784fe
- b307f700
- 2322f4fe
- 6f000000
- 130101ff
- 23261100
- 23248100
- 13040101
- 17110000
- 13014101
- eff09ffb
- 6f000000

Debugging

- # Compile with debugging information
- riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -g -o my_program ./for_loop.c

Start GDB and load program

- riscv64-unknown-elf-gdb my_program
- # Run program in GDB
- (gdb) target sim
  - (gdb) break linenumber
  - (gdb) print variable_name

Profiling

- **# Compile for performance analysis with perf**
- riscv32-unknown-elf-gcc -march=rv32i -o my_program ./code.c
- # Run program with QEMU and collect profiling data
- qemu-riscv32 -cpu rv32, my_program -perf my_program
- # Analyze profiling data with perf
- // Not yet configured in cluster

Stress Testing

- riscv32-unknown-elf-gcc -march=rv32i -o stress-ng stress-ng.c
- **# Run stress tests with stress-ng**
- qemu-riscv32 -L /path/to/riscv/rootfs ./stress-ng --cpu 4 --io 2 --vm 2 --vm-bytes 128M --timeout 60s
- **Custom Stress Checking**
- riscv32-unknown-elf-gcc -march=rv32i -o stress_test ./stress_test.c
- **# Run custom stress test program**
- qemu-riscv32 ./stress_test

Performance Analysis

- riscv32-unknown-elf-gcc -march=rv32i -o my_program ./code.c
- qemu-riscv32 -L /path/to/riscv/rootfs valgrind --tool=cachegrind ./my_program
- # Run program with QEMU for performance analysis
- qemu-riscv32 -d in_asm,cpu ./my_program > qemu_log.txt
- # Analyze QEMU log
- grep -E 'IN:|CPU:|Cycle:' qemu_log.txt

Testing Spike

/opt/riscv-gnu32/bin/spike --isa=RV32IMAC -d /opt/riscv/riscv32-unknown-elf/bin/pk ./heap32
until reg 0 pc 0x1000 # Stop execution when program counter of core 0 reaches 0x1000
mem 0 0x80000000 # View memory content at address 0x80000000 for core 0
freg 0 f0 # Display floating-point register f0 for core 0
run 1000 # Resume execution for 1000 instructions
reg 0 # View all registers for core 0
pc 0   # View the program counter of core 0
until pc 0 0x1000 # Stop execution when PC of core 0 reaches address 0x1000
while reg 0 sp 0x80000000 # Continue running while stack pointer (sp) of core 0 is 0x80000000
dump 0x80000000 0x80001000 # Dump memory from address 0x80000000 to 0x80001000
quit
mtime
mtimecmp 0

QEMU Debuging

- qemu-system-riscv32 -gdb tcp::1234 -S -kernel ./hello32.o
- riscv32-unknown-elf-gdb ./hello32.o #Sperate window open
- Debug Commands
- (gdb) target remote :1234 # Connect to the QEMU GDB server

(gdb) load # Load the binary into QEMU
(gdb) b main # Set a breakpoint at the main function
(gdb) c # Continue execution until the breakpoint is hit
(gdb) info reg # Display registers
(gdb) step # Step through code line by line
(gdb) next # Step over functions
(gdb) continue # Continue execution until the next breakpoint
(gdb) quit # Exit GDB

Profiling QEMU

- qemu-system-riscv32 -d exec,int -kernel ./hello32.o
- perf record -e cycles -a -- qemu-system-riscv32 -kernel ./hello32.o
- perf report

RISC-V Computer Architecture Course Tasks: By completing these tasks, you'll gain hands-on experience in programming, designing, and testing a RISC-V processor, bridging the gap between theoretical knowledge and practical application.

Task 1: Programming RISC-V Using Assembly Language and Ripes Simulator

Write RISC-V programs in Assembly language.

Simulate the Assembly code using the Ripes Simulator to understand the execution flow.

Task 2: Developing a RISC-V Processor with Custom ISA in Verilog

Design the Processor

Implement a RISC-V processor in Verilog that supports at least 20 instructions from the RISC-V ISA.

Test the Processor

Write manual machine code corresponding to your implemented instructions.

Compare the results of executing machine code against Assembly-level output to verify functionality.

Task 3: Programming Your RISC-V Processor Using a C Compiler

Write a Simple C Program

Develop a basic C program for the RISC-V architecture (e.g., arithmetic operations or loops).

Convert C Code to Assembly

Compile the C code to generate Assembly code using a RISC-V toolchain (e.g., GCC or LLVM).

Generate an ELF File

Produce an ELF (Executable and Linkable Format) file as part of the compilation process.

Extract Program and Data Code

Extract the program (instruction code) and data sections from the ELF file.

Integrate Code with Verilog Design

Load the extracted program and data into the memory system of your Verilog-based RISC-V processor.

Perform Simulation

Simulate the processor with the integrated program and data.

Observe the processor’s behavior and verify correctness.

Task 4: Documentation and Understanding

Write a Report

Document the entire process, including implementation, testing, and results.

Highlight challenges, solutions, and observations.

Prepare for Viva

Gain a deep understanding of all tasks for oral examination.

Be prepared to explain your processor design, testing methodology, and results in detail