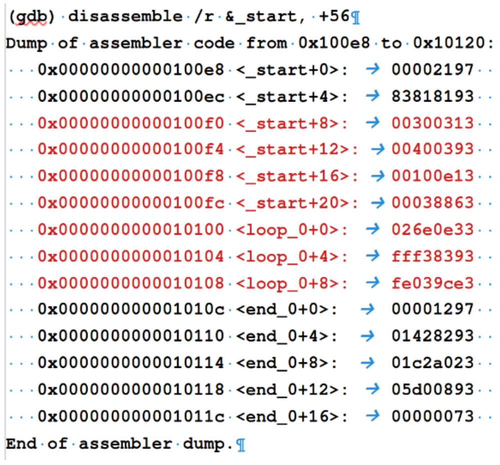
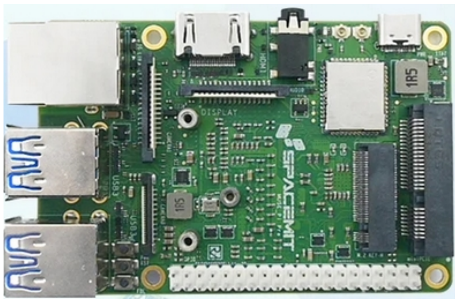
Programming and Modeling RISC-V on RISC-V architecture

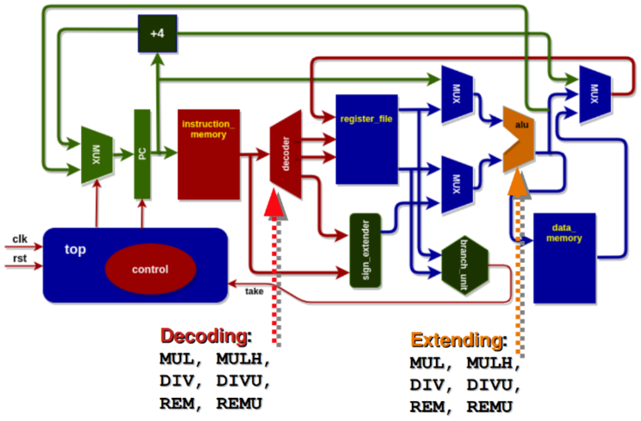
Przemyslaw **Bakowski**

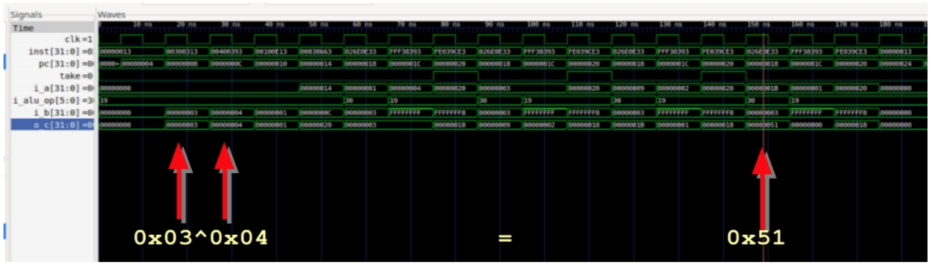


```
(gdb) disassemble /r &_start, +56
Dump of assembler code from 0x100e8 to 0x10120:
   0x00000000000100e8 <_start+0>:   →  00002197
   0x00000000000100ec <_start+4>:   →  83818193
   0x00000000000100f0 <_start+8>:   →  00300313
   0x00000000000100f4 <_start+12>:  →  00400393
   0x00000000000100f8 <_start+16>:  →  00100e13
   0x00000000000100fc <_start+20>:  →  00038863
   0x0000000000010100 <loop_0+0>:   →  026e0e33
   0x0000000000010104 <loop_0+4>:   →  fff38393
   0x0000000000010108 <loop_0+8>:   →  fe039ce3
   0x000000000001010c <end_0+0>:    →  00001297
   0x0000000000010110 <end_0+4>:    →  01428293
   0x0000000000010114 <end_0+8>:    →  01c2a023
   0x0000000000010118 <end_0+12>:   →  05d00893
   0x000000000001011c <end_0+16>:   →  00000073
End of assembler dump.
```

```
    // M Extension Operations
    `OP_ALU_MUL:  o_c=i_a * i_b;
    `OP_ALU_MULH: o_c=(($signed(i_a)*$signed(i_b))>>32);
    // Multiplication High (signed)
    `OP_ALU_DIV:  o_c=(i_b!=0)?$signed(i_a)/$signed(i_b):0;
   // Division (signed)
   `OP_ALU_DIVU: o_c=(i_b!=0)?i_a/i_b:0;
   // Division Unsigned
   `OP_ALU_REM:  o_c=(i_b!=0)?$signed(i_a)%$signed(i_b):i_a;
   // Remainder (signed)
  `OP_ALU_REMU: o_c=(i_b!=0)?i_a%i_b:i_a;
  // Remainder Unsigned
  default: o_c = 0;
endcase
```







Lab 0

Introduction

RISC-V is gaining popularity in digital systems due to its **simplicity**, **efficiency**, and **open-source** nature. Understanding RISC-V architecture prepares students for working with these systems. Many companies seek professionals who understand this architecture for its roles in processor design, embedded systems development, IoT systems development, and related fields.
Teaching RISC-V architecture via **assembly-level programming** and **RTL modeling** allows students to learn **core concepts** effectively and apply them in practical projects.

This book presents the **didactic and development platform** to teach and model RISC-V ISA.
Our method is **two-fold** (**software/hardware**) and **self-contained** (modeling **RISC-V on RISC-V**).
The platform itself is largely affordable and running exclusively on open source software, modeling tools included.

The initial didactical content is built from several **Programming Labs** (**Plabs**), and **Modeling Labs** (**MLabs**).
PLabs start with simple examples involving arithmetical instructions and input/output operations. We also delve, with the help of the debugger, into the binary representations to understand the instruction formats and build binary code snippets.
MLabs start with a short introduction to Verilog HDL. With the following MLabs we study simple RISC-V architecture, first to model **RV32I**-subset with R-type instructions then to model full RV32I plus M subsets. Then, running on the RISC-V platform, we inject the generated binaries into the Verilog model. As such the platform is open for further experimentation with RISC-V ISA based programming and modeling.

Along with the programming and the modeling processes we specify **ChatGPT** prompts to generate **assembly code** snippets and the **Verilog modules** providing test bench codes.

Our two-sided, **software-hardware**, approach to teach the RISC-V ISA requires the use of an **actual hardware platform** with RISC-V processor SoCs.
There are several RISC-V implementations commercially available. The most affordable and complete are the boards integrating **SpacemiT X60 Intelligence Core**.
X60 complies with **RVA22 profile** and implements **256-bit RVV1.0** standard. **MUSE Pi Pro** is the reference board integrating X60 SoC.

The board operates under Debian like **Bianbu OS**.
The following picture shows architectural block scheme of X60 and the MUSE Pi Pro board.
Below are essential features of **X60**:
Compliance with RISC-V **64GCVB** and RVA22 standards

- Each core has 32KB L1-I cache and 32KB L1-D cache
- Each cluster contains 512KB L2 cache
- Cluster 0 integrates 512KB TCM (Tight-Coupled Memory) for AI extension
- L1 cache supports MESI consistency protocol, instead L2 cache supports MOESI consistency protocol
- Vector extension: RVV1.0 with VLEN 256/128-bit and x2 execution width
- AI customized instructions explored and implemented in Cluster 0
- Support for CLINT and PLIC with a total of 256 interrupts
- Support for RISC-V performance PMU
- Support for SV39 virtual memory
- Support for 32 PMP entries adhering to RISC-V security framework
- Support for RISC-V debug framework
- Support for the following extensions:

- RV64I M A F D C V and many Binary extension plus AI customized instructions

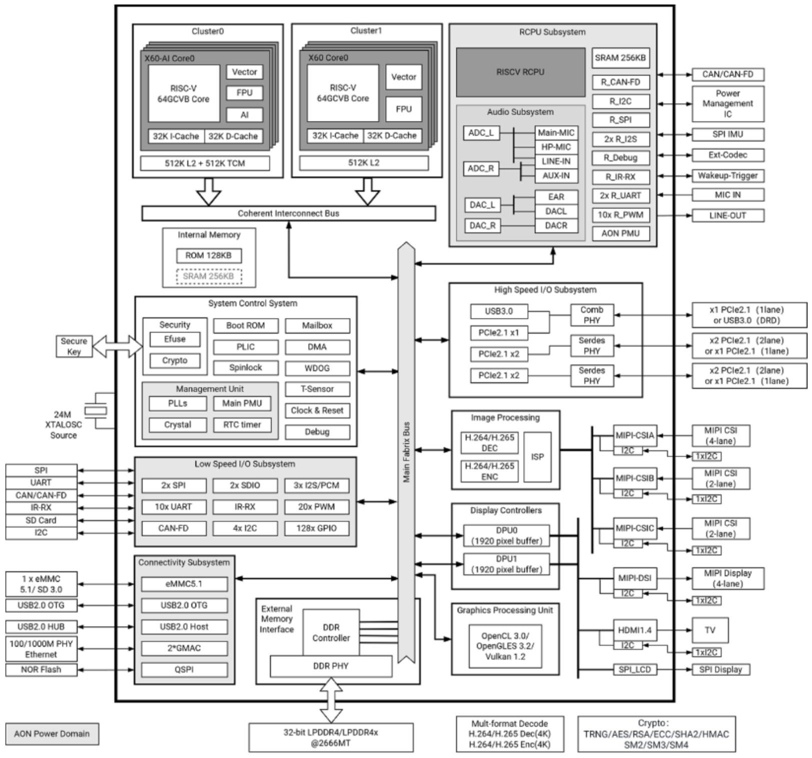


Fig 0.1 Architectural block scheme of **X60** SoC including two **RV64 clusters**.

Muse Pi Pro shown in the following picture is a feature-packed, credit card-sized SBC powered by the SpacemIT M1 octa-core 64-bit RISC-V AI SoC with a 2 TOPS NPU and equipped with up to 16GB LPDDR4x and 128GB eMMC flash.
The single board computer features gigabit Ethernet and a WiFi 6 + Bluetooth 5.3 module for connectivity, HDMI and MIPI DSI display interfaces, two MIPI CSI interfaces, a 3.5mm audio jack, four USB 3.0 ports, an M.2 socket for an NVMe SSD or wireless module, a mini PCIE socket for WiFi and 4G LTE/5G cellular connector, and a 40-pin GPIO header for expansion. That's quite a lot of features for such a compact board.

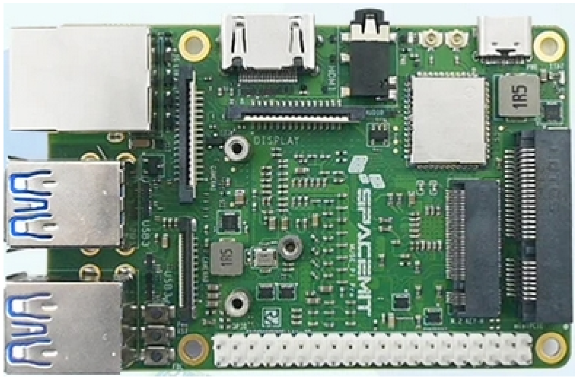


Fig 0.2 MUSE Pi Pro board from **SpacemiT** integrating **X60 SoC**

Other boards with SpacemiT X60 SoC

For our laboratory exercises and examples, we utilized the MUSE Pi Pro development board. However, these experiments can also be conducted on other compatible hardware platforms including:

- **Banana Pi BPI-F3**
- **OrangePi RV2**

Note: The OrangePi RV2 entry-level pricing begins at $30–40 (or €30–40), varying according to RAM configuration.

Book organization

The book is organized in two principal parts:

1. Programming Labs - **PLabs**
2. Modeling Labs - **MLabs**

Both parts are built from **series of Labs**. The last **PLab** provides the output – binary codes to be used in the following **MLabs** .

Resources

You will find the RISC-V assembly codes and Verilog models in
`github.com/smartcomputerlab/Programming.and.Modeling.RISC-V.on.RISC-V`
repository

Programming Labs

Programming RISC-V with assembly language offers several key advantages, especially in scenarios where control, optimization, and hardware awareness are critical. Below are some of the primary benefits:

1. **Fine-Grained Control of Hardware**
   - **Direct Access to CPU Features**: RISC-V assembly allows developers to directly interact with processor instructions, registers, memory, and I/O devices. This level of control is essential for hardware-level tasks such as interrupt handling, device drivers, or manipulating specific hardware peripherals.
   - **Custom Instruction Set Extensions**: RISC-V allows for user-defined custom instructions, so writing in assembly helps exploit these extensions effectively when needed for specialized tasks.

2. **Performance Optimization**
   - **Manual Optimization**: Assembly language gives developers the ability to optimize their code for speed, size, or power efficiency by manually tuning instructions, avoiding unnecessary overhead, and making decisions about which operations are faster for a given processor.
   - **Instruction-Level Parallelism**: Developers can control how instructions are scheduled, potentially reducing instruction stalls, pipeline hazards, and maximizing the use of the CPU's pipelines.
   - **Efficient Use of Memory**: Assembly allows developers to minimize memory usage, a critical factor for embedded systems or resource-constrained environments like micro-controllers.

3 **Small Code Size**
   - **Minimal Overhead**: Writing in assembly produces minimal overhead since high-level language constructs like loops, conditionals, and function calls are replaced with direct machine instructions. This is particularly useful in systems with limited memory (e.g., embedded systems).
   - **Precise Control of Memory Layout**: In assembly, the programmer has direct control over how data and code are laid out in memory, allowing for optimized and compact memory usage.

4 **Embedded Systems and Real-Time Applications**
   - **Low-Level Access**: Assembly language is often used in embedded and real-time systems where low-level control is essential, such as controlling specific peripherals, real-time performance tuning, and interrupt handling.
   - **Deterministic Execution**: In real-time systems, knowing the exact execution time of instructions is important. Assembly provides a clear understanding of how long each instruction will take, ensuring real-time constraints are met.

Writing in assembly helps **developers gain a deep understanding of the underlying RISC-V architecture**, including how **memory** is accessed, how **instructions are executed**, and how **control flow** is managed.
By learning to write in assembly, programmers also develop insights into what compilers do behind the scenes, allowing for better high-level code optimization and debugging.

RISC-V: base assembly instruction set

The RISC-V **base integer instruction set**, often referred to as the **"I" (Integer)** instruction set, includes a small but complete set of instructions necessary for general-purpose computing. This set is part of the **RV32I** and **RV64I** instruction sets, with RV32I being a 32-bit variant and RV64I being a 64-bit variant. These instructions include basic arithmetic, logical, control, memory access, and system instructions.
In our labs we use X60 based development board (SBC). X60 integrates two cluster of **RV64** processors.
Here's an **overview** of the basic instructions in the RISC-V "**I**" instruction set, categorized by their purpose.

Note the typical **RISC type architecture** separation between the:

- **arithmetical**/**logical** instructions
- **memory load**/**save** instructions

and

- **control jump**/**branch** instructions

1 Arithmetic Instructions
These instructions perform integer arithmetic operations.

- **`add rd, rs1, rs2`** — Add two registers (**`rd = rs1 + rs2`**).
- **`addi rd, rs1, imm`** — Add immediate (**`rd = rs1 + imm`**).
- **`sub rd, rs1, rs2`** — Subtract (**`rd = rs1 - rs2`**).
- **`lui rd, imm`** — Load upper immediate (**`rd = imm << 12`**).
- **`auipc rd, imm`** — Add upper immediate to PC (**`rd = PC + (imm << 12)`**).
-

2 Logical Instructions
These instructions perform **bitwise logical operations**.

- **`and rd, rs1, rs2`** — Bitwise AND (**`rd = rs1 & rs2`**).
- **`andi rd, rs1, imm`** — Bitwise AND with immediate (**`rd = rs1 & imm`**).
- **`or rd, rs1, rs2`** — Bitwise OR (**`rd = rs1 | rs2`**).
- **`ori rd, rs1, imm`** — Bitwise OR with immediate (**`rd = rs1 | imm`**).
- **`xor rd, rs1, rs2`** — Bitwise XOR (**`rd = rs1 ^ rs2`**).
- **`xori rd, rs1, imm`** — Bitwise XOR with immediate (**`rd = rs1 ^ imm`**).
-

3. Shift Instructions
These instructions perform **left or right shifts**.

- **`sll rd, rs1, rs2`** — Shift left logical (**`rd = rs1 << rs2`**).
- **`slli rd, rs1, imm`** — Shift left logical immediate (**`rd = rs1 << imm`**).
- **`srl rd, rs1, rs2`** — Shift right logical (**`rd = rs1 >> rs2`**).
- **`srli rd, rs1, imm`** — Shift right logical immediate (**`rd = rs1 >> imm`**).
- **`sra rd, rs1, rs2`** — Shift right arithmetic (**`rd = rs1 >> rs2`**).
- **`srai rd, rs1, imm`** — Shift right arithmetic immediate (**`rd = rs1 >> imm`**).

4 Comparison Instructions
These **instructions compare values** in registers and set the **destination register to 1** if the comparison is `true`, **otherwise set it to 0**.

- **`slt rd, rs1, rs2`** — Set if less than (**`rd = (rs1 < rs2))`**.
- **`slti rd, rs1, imm`** — Set if less than immediate (**`rd = (rs1 < imm)`**).
- **`sltu rd, rs1, rs2`** — Set if less than (unsigned) (**`rd = (rs1 < rs2) unsigned`**).
- **`sltiu rd, rs1, imm`** — Set if less than immediate (unsigned) (**`rd = (rs1 < imm`**) unsigned).

5 Memory Access Instructions
These instructions **load** data from **memory into registers** or **store** data from registers into memory.

- **`lw rd, imm(rs1)`** — Load word (**`rd = Mem[rs1 + imm]`**).
- **`lh rd, imm(rs1)`** — Load halfword.

- **`lb rd, imm(rs1)`** — Load byte.
- **`lbu rd, imm(rs1)`** — Load byte unsigned.
- **`lhu rd, imm(rs1)`** — Load halfword unsigned.
- **`sw rs2, imm(rs1)`** — Store word (**`Mem[rs1 + imm] = rs2`**).
- **`sh rs2, imm(rs1)`** — Store halfword.
- **`sb rs2, imm(rs1)`** — Store byte.

6 Control Transfer Instructions

These instructions control the flow of execution, including conditional branches and unconditional jumps.

- **`beq rs1, rs2, offset`** — Branch if equal.
- **`bne rs1, rs2, offset`** — Branch if not equal.
- **`blt rs1, rs2, offset`** — Branch if less than (signed).
- **`bge rs1, rs2, offset`** — Branch if greater than or equal (signed).
- **`bltu rs1, rs2, offset`** — Branch if less than (unsigned).
- **`bgeu rs1, rs2, offset`** — Branch if greater than or equal (unsigned).
- **`jal rd, offset`** — Jump and link (used for function calls).
- **`jalr rd, offset(rs1)`** — Jump and link register.

7 System Instructions

These instructions provide system-level control, including **traps and environment calls** (for example, for operating system services).

- **`ecall`** — Environment call (used to invoke system services, e.g., syscalls).
- **`ebreak`** — Environment break (used for debugging or breakpoints).

8 No-Operation Instruction

This instruction does nothing and is often used for padding.

- **`nop`** — No operation (**`addi x0, x0, 0`** is commonly used as **`nop`**).

9 Example Program: Sum of Two Numbers

Here is a simple RISC-V assembly program that **adds two numbers and stores the result in a register**.

```
    .text
    .globl _start

_start:
    # Load two numbers into registers
    li a0, 10          # Load immediate value 10 into register a0
    li a1, 20          # Load immediate value 20 into register a1
    # Perform addition
    add a2, a0, a1     # a2 = a0 + a1 (10 + 20 = 30)
    # Exit the program using ecall
    li a7, 93          # Syscall number for exit
    ecall              # Make system call
```

Basic instruction formats

Below we provide a simplified visual representation of the basic RISC-V instruction formats for the **I** (Integer) instruction set. These formats are **R-type**, **I-type**, **S-type**, **B-type**, **U-type**, and **J-type**.

Each format specifies how to structure the 32-bit instruction word in RISC-V assembly. These formats are not explicitly "visible" in assembly code but there they provide some insight into the capacities of the register block and the encoding schemes.

The implementation of these elements are essential for **Verilog models** studied in the second part of this book.

1. R-Type Format (Used for register-register operations)

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode

- **`opcode`**: Operation code (e.g., `0110011` for R-type).
- **`rd`**: Destination register.
- **`funct3`**: Specifies the operation within the opcode.
- **`rs1`**: Source register 1.
- **`rs2`**: Source register 2.
- **`funct7`**: Additional bits to distinguish operations.

2. I-Type Format (Used for immediate instructions, loads)

31 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[11:0]	rs1	funct3	rd	opcode

- **`opcode`**: Operation code (e.g., `0000011` for loads).
- **`rd`**: Destination register.
- **`funct3`**: Specifies the operation.
- **`rs1`**: Source register.
- **`imm[11:0]`**: 12-bit immediate value.

3. S-Type Format (Used for stores)

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

- **`opcode`**: Operation code (e.g., `0100011` for stores).
- **`funct3`**: Specifies the operation.
- **`rs1`**: Base register for address calculation.
- **`rs2`**: Source register to store.
- **`imm[11:5]`** & **`imm[4:0]`**: Split 12-bit immediate value.

4. B-Type Format (Used for branches)

31	30 - 25	24 - 20	19 - 15	14 - 12	11	10 - 7	6 - 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

- **`opcode`**: Operation code (e.g., `1100011` for branches).
- **`funct3`**: Specifies the branch type.
- **`rs1`** & **`rs2`**: Registers to compare.
- **`imm[12], imm[10:5], imm[4:1], imm[11]`**: Split immediate value for target address.

5. U-Type Format (Used for upper immediate)

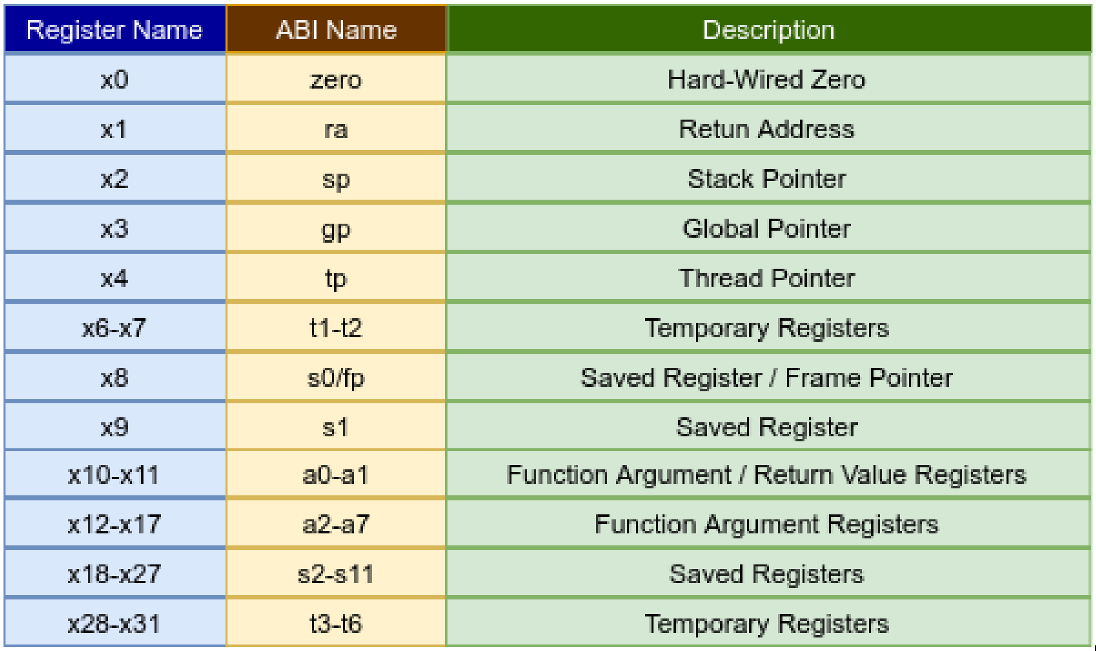
`31 – 12`	`11 – 7`	`6 – 0`
`imm[31:12]`	`rd`	`opcode`

- **`opcode`**: Operation code (e.g., `0010111` for **`AUIPC`**).
- **`rd`**: Destination register.
- **`imm[31:12]`**: 20-bit immediate value (upper 20 bits).

6. J-Type Format (Used for jumps)

`31`	`30 – 21`	`20`	`19 – 12`	`11 – 7`	`6 – 0`
`imm[20]`	`imm[10:1]`	`imm[11]`	`imm[19:12]`	`rd`	`opcode`

- **`opcode`**: Operation code (e.g., `1101111` for jumps).
- **`rd`**: Destination register.
- **`imm[20], imm[10:1], imm[11], imm[19:12]`**: Split immediate for target address.



Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Retun Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x6-x7	t1-t2	Temporary Registers
x8	s0/fp	Saved Register / Frame Pointer
x9	s1	Saved Register
x10-x11	a0-a1	Function Argument / Return Value Registers
x12-x17	a2-a7	Function Argument Registers
x18-x27	s2-s11	Saved Registers
x28-x31	t3-t6	Temporary Registers

Fig 0.5 RISC-V 32/64 register file: **`t1,t2,..,t3-t6`** and **`a0-a7`** - user data registers; **`a0-a7`** – function argument registers, **`a0,a1`** - return value registers

Binary codes for basic R-type format - example

Let us take an R-Type format instruction and look at the corresponding binary code. This type of instruction is used for register-register operations like addition, subtraction, and logical operations. **R-type instructions have six fields in their 32-bit format**.

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
funct7	rs2	rs1	funct3	rd	opcode

Here's a breakdown of each field in the **R-type format**:

1. **`opcode`** (7 bits): Specifies the operation type (e.g., `0110011` for all R-type instructions).
2. **`rd`** (5 bits): The destination register.
3. **`funct3`** (3 bits): A sub-field of the operation code that helps specify the exact operation.
4. **`rs1`** (5 bits): The first source register.
5. **`rs2`** (5 bits): The second source register.
6. **`funct7`** (7 bits): Another sub-field of the operation code for further operation specification.

Example: Binary Code for `ADD` and `SUB` Instructions

For specific operations like `ADD` and `SUB`, here's how the binary fields would look in this format:

- **ADD** (Addition):
  - **`opcode`**: `0110011`
  - **`funct3`**: `000`
  - **`funct7`**: `0000000`
- **SUB** (Subtraction):
  - **`opcode`**: `0110011`
  - **`funct3`**: `000`
  - **`funct7`**: `0100000`

Example of an `ADD` Instruction (**`add x1, x2, x3`**)

funct7	rs2	rs1	funct3	rd	opcode
0000000	00011	00010	000	00001	0110011

- **`funct7`**: `0000000` for ADD
- **`rs2`**: Register x3 (binary `00011`)
- **`rs1`**: Register x2 (binary `00010`)
- **`funct3`**: `000` for ADD
- **`rd`**: Register x1 (binary `00001`)
- **`opcode: 0110011`**

Here's how we can write a simple **`"HelloWorld"`** program in RISC-V assembly and compile it to run on a RISC-V system.
Then we will explain the differences between compiling with **`gcc`** and **`as`**.

RISC-V Assembly Code for `"HelloWorld"`

RISC-V assembly doesn't have built-in support for printing to standard output, so this example relies on making a **system call to handle printin**g. Here's the code:

```
    .section .data
message:
    .string "HelloWorld\n"
    .section .text
    .globl _start
_start:
    # Load the address of our message into a1
    la a1, message
    # Load the file descriptor for stdout (1) into a0
    li a0, 0
    # Load the length of the message (11 characters) into a2
    li a2, 11
    # Load the syscall number for write (64) into a7
    li a7, 64
    # Make the syscall to write
    ecall
    # Exit the program
    li a7, 93       # Syscall number for exit
    li a0, 0        # Exit code 0
     ecall
```

This program:

1. Loads the address of the `message` string into register **`a0`**.
2. Sets up the **syscall** for writing by placing the appropriate syscall number (64 for `write`) in **`a7`**, the **file descriptor** (**1** for **`stdout`**) in **`a1`**, and the **length of the string** in **`a2`**.
3. Uses **`ecall`** to invoke the system call to write to the console.
4. Exits the program by invoking the **`exit`** syscall.

Compiling with `as` vs. gcc

1. **Compiling with `as` (Assembler)**

- The `as` command is used as a stand-alone assembler. It takes your RISC-V assembly file (e.g., **`helloworld.s`**) and converts it directly to machine code (an object file, **`helloworld.o`**).
- To produce an executable with **`as`**, you need to use the linker **`ld`** after assembly:

```
as -o helloworld.o helloworld.s
ld -o helloworld helloworld.o
```

- This process requires you to handle linking manually. If your assembly code depends on external libraries or needs startup routines (like **`_start)`**, you must provide them explicitly.

2. **Compiling with `gcc`**

- **`gcc`** is a higher-level compiler that can handle both C and assembly source files. When you pass an assembly file to **`gcc`**, it first **assembles** it (using `as` internally) and then **links** it **automatically**.
- The advantage of using **`gcc`** is that it automates linking and includes the standard startup code and libraries that set up the **environment** for your program.

You can compile the above assembly code using **`gcc`** like this:

```
gcc -nostartfiles -o helloworld helloworld.s
```

Here, **`-nostartfiles`** is used to prevent `gcc` from adding the **default C runtime startup files** (like **`crt0.o`**), as we already defined the **`_start`** entry point.

Key Differences between `as` and `gcc`:

- **Manual Linking (as)**: `as` **only assembles**; it doesn't automatically link. You must use `ld` to produce an executable, making it a more manual process.
- **Automatic Linking (gcc)**: `gcc` automates linking and may add standard C runtime initialization code unless explicitly suppressed. It **handles dependencies on standard libraries** and routines, which can be helpful if you're mixing assembly with C code.
- **Error Checking and Libraries**: `gcc` provides more extensive error checking and can link to the standard C library by default, which is useful if your program depends on functions like `printf`.

Important:

Add to protect the data sections:

```
      _start:
          .option norelax
          la gp, __global_pointer$
```

`.option norelax` Tells assembler not to **allow linker relaxations**. Keeps address loading full/explicit.

`la gp, __global_pointer$`
`Loads the address of the` **`special global pointer`** `into the` **`gp`** `register.`

Note:

the `stdout` (1) must be initialized in `a0`, and the string address in `a1`.

Analyze the above code and compile it with `gcc` (without and with `-nostartfiles` option) and with `as` and `ld`.

Note the difference of binary code (`elf`) size, can you explain it ?

```
-rwxrwxr-x 1 bako bako 1352 11月11 10:48 HelloWorldass
-rwxrwxr-x 1 bako bako 8576 11月11 10:54 HelloWorldass_gcc
-rwxrwxr-x 1 bako bako 6232 11月11 10:55 HelloWorldass_gcc_no
```

Attention:

In case of use `-nostartfiles` option we may nave problem with the initialization of **global pointer** (`gp`) address. If we use this flag or simple `as` and `ld` commands we may initialize the global pointer at the beginning of .text section:

Example:

```
    .section .text
    .globl _start
_start:
    .option norelax
    la gp, __global_pointer$
..  # here starts the program ..
```