RISC-V Assembly Language Programming on C906 and Ubuntu 22.10

1.1 Introduction

C906, Allwinner D1-H, Lichee RV SBC KIT

LINUX GPIO DAEMON (GPIOD) : https://github.com/maquefel/gpiod

RISC-V pronounced as “RISC-five”, is an open-source standard **Instruction Set Architecture** (ISA), designed based on Reduced Instruction Set Computer (**RISC**) principles. With a flexible architecture to build systems ranging from a simple microprocessor to complex multi-core systems, RISC-V caters to any market.

The RISC-V ISA provides two specifications, one, the User Level Instructions which guides in developing simple embedded systems and connectivity applications and two, the Privilege Level Instructions which guides in building secure systems, kernel, and protected software stacks.
RISC-V currently supports **three privilege levels**, viz.. **Machine**/**Supervisor**/**User**, with each level having **dedicated Control Status Registers** (CSRs) for system state observation and manipulation.

In addition, RISC-V provides 31 read/write registers. While all can be used as general-purpose registers, they have dedicated functions as well. RISC-V is divided into different categories based on the maximum width of registers the architecture can support, for example, RV32 (RISC-V 32) provides registers whose maximum width is 32-bits and RV64 (RISC-V 64) provides registers whose maximum width is 64-bits. Processors with larger register widths can support instructions and data of smaller widths. So an RV64 platform supports both RV32 and RV64.

Note:
This text uses the term **XLEN** to refer to the platform register width, in bits.
PART-I of the RISC-V programmer’s manual, details RISC-V assembly instructions, registers in use and the machine privilege level.
Advanced concepts on Privilege levels, Memory Management unit and Trap delegation will be dealt with in PART-II of the manual.
The objective of the RISC-V ASM (assembly language) programmer manual is to aid users in writing extensive assembly programs and provide necessary information to write simple **embedded applications**.

1.2 Registers

RISC-V architecture provides **31 user modifiable general-purpose** (base) **registers**, namely, **x1 to x31**, and with an additional read-only register x0, hard-wired to zero. One common use of **x0** register is to aid in initializing other registers to **zero**.
In comparison to other ISAs, RISC-V uses a larger number of integer registers which helps in performance, where extensive use of loop unrolling and software pipelining is required.
In RISC-V systems, the following are the available base registers:
There are 31 general purpose registers.

- there are 7 **temporary registers** (**t0 – t6**).
- **a0 – a7** are used for **function arguments**.
- **s0 – s11** are used as s**aved registers** or within function definitions.
- There is one **stack pointer**, one **global pointer** and one **thread pointer register**.
- A **return address register** (**x1**) to store the **return address in a function call**.
- One **program counter** (**pc**). **pc** holds the **address of the current instruction**.
-
- All the registers can be used as a **general purpose register**.

The **Base registers** can hold either data or a valid address and are usually identified with the letter ’**x**’ prefixing the register number.

A brief description of the registers and their additional functions are as follows.

1.2.1 Stack Pointer Register

In RISC-V architecture, the **x2** register is used as **Stack Pointer** (**sp**) and holds the **base address of the stack**.
When **programming explicitly in RISC-V assembly language**, it is mandatory to load **x2** with the **stack base address** while the C/C++ compilers for RISC-V, are always designed to use x2 as the stack pointer. In addition, **stack base address must aligned to 4 bytes**.
Failing which, a load/store alignment fault may arise.
The **x2** register can hold an operand in the following ways:

- As a **base register for load and store instruction**. In this case, the load/store address must be **4 byte aligned**.

- As a source or destination register for **arithmetic/logical/csr** instructions.

1.2.2 Global Pointer Register

Data is allocated to the memory when it is globally declared in an application. Using **pc-relative** or absolute addressing mode leads to utilization of extra instructions, thus increasing the code size.
In order to decrease the code size, RISC-V places all the global variables in a particular area which is **pointed** to, using the **x3** (**gp**) **register**. The **x3** register will hold the base **address of the location where the global variables reside**.

1.2.3 Thread Pointer Register

In **multi-threaded** applications, each thread may have its **own private set of variables** which are called "thread specific variables". This **set of variables** will be pointed to by the register **x4** (**tp)**.
Hence, each thread will have a different value in its **x4** register.

1.2.4 Return Address Register

The **x1** (**ra**) register is used to save the subroutine **return addresses**. Before a subroutine call is performed, **x1** is explicitly set to the **subroutine return address** which is usually '**pc + 4**'.
The standard software calling convention uses **x1** (**ra**) register to hold the **return address on a function call**.

1.2.5 Argument Register

In RISC-V, **8 argument registers**, namely, **x10 to x17** are used to pass **arguments in a subroutine**.
Before a subroutine call is made, the arguments to the subroutine are copied to the argument registers.
The **stack is used** in case the number of arguments exceeds **8**.

1.2.6 Temporary Register

As the name suggests, the temporary registers are used to hold **intermediate values** during instruction execution. There are **seven temporary registers** (**t0 – t6**) in RISC-V.

Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary/Alternate Link Register
x6-7	t1-t2	Temporary Register
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-11	a0-a1	Function Argument/Return Value Registers
x12-17	a2-a7	Function Argument Registers
x18-27	s2-s11	Saved Registers
x28-31	t3-t6	Temporary Registers

1.3 Privilege mode

Inter-process security for a system necessitates the extent to which each process can use the system resources, to maintain the system and data integrity. These processes are grouped into **different modes/levels**, from low to high, and possess varying levels of privilege.
Higher privilege modes have a greater system leveraging capacity in addition to their own. A mode trying to access a region it has no permission for, **causes exceptions/traps**.

The three privilege levels are listed below,

Privilege	Value	Encoding	Abbreviation
User mode	0	00	U
Machine mode	3	11	M
Supervisor mode	1	01	S

With reference to the above, the value field states the value of a privilege level. Encoding is used to encode the privilege level in a CSR registers. Machine level has the highest privilege and is also mandatory. Machine mode is inherently trusted, as it has low level access to the machine implementation.

All software by default start in Machine Mode.

This preparation deals with the **Machine Mode**. The other two modes are used for developing conventional applications and system software.

1.4 Control and Status Registers (CSRs)

The **Control and Status Register** (**CSR**) are **system registers** provided by RISC-V to control and monitor system states 1 . **CSR's** can be **read**, **written** and bits can be **set/cleared**. RISC-V provides **distinct CSRs** for **every privilege level**. Each CSR has a special name and is assigned a unique function. In addition to the machine level CSRs described in this section, **M-mode** code can access the CSRs at lower privilege levels.

Reading and/or writing to a CSR will affect processor operation. CSR's are used in operations, where a normal register cannot be used. For example, knowing the system configuration, handling exceptions, switching to different privilege modes and handling interrupts are some tasks for which a CSR is needed. The CSR cannot be read/written the way a general register can.
A special set of instructions called **csr instructions** are used to facilitate this process. CSR instructions require an intermediate base register to perform any operation on CSR registers. Further, it is possible to write immediate values to CSR registers. table1.3 lists the CSRs present in machine mode.

1.4.1 CSR Field Specifications

An attempt to access a CSR that is not visible in the current mode of operation results in privilege violation. Similarly, in the current mode of operation, a privilege violation occurs when an attempt is made to write to a "read-only" labeled CSR. This attempt results in an illegal instruction exception.
In addition to restrictions on how a CSR register is accessed, fields within some registers come with their own restrictions which are as listed as follows.

Register	Description
misa	Machine ISA
mvendorid	Machine Vendor ID
marchid	Machine Architecture ID
mimpid	Machine Implementation ID
mstatus	Machine Status
mcause	Machine trap cause
mtvec	Trap vector base address

Register	Description
mhartid	Machine Hardware thread ID
mepc	Machine exception program counter
mie	Machine interrupt enable
mip	Machine interrupt pending
mtval	Machine trap value
mscratch	Scratch register

1.4.1.1 Reserved Writes Ignored, Reads Ignore Values (WIRI)

Read-only fields within some read-only and read/write registers, have been reserved for future use. Such fields have been named as **Reserved Writes Ignored**, **Reads Ignore Values** (**WIRI**). A read or write to these fields must be ignored. In case the entire CSR is a read-only register, an attempt to write to the WIRI field will raise an **illegal instruction exception**.

1.4.1.2 Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Although, there are fields labeled “read/write” in some registers, they are reserved for future use and are not available for software modifications. Such fields are called as **Reserved Writes Preserve Values**, **Reads Ignore Values** (WPRI). Values returned on a reading such fields must be ignored, while an attempt to write to the whole register containing such fields must preserve the original value.

1.4.1.3 Write/Read Only Legal Values (WLRL)

Some fields restrict the values that can be read/written to a field. Such values are called “legal” values and are specified by the processor. Fields with this restriction are labeled as **Write/Read Only Legal Values** (WLRL). A read on such a field returns a legal value if legal values are written to it. Caution should be exercised to write only legal values as illegal writes may not return legal values.

1.4.1.4 Write Any Values, Reads Legal Values (WARL)

Some read/write fields offer the freedom of writing any value to it while reading them, will only return values which are legal. Such fields are labeled as **Write Any Values**, **Reads Legal Values** (WARL). Implementations will not raise an exception on writes of unsupported values to an WARL field. Implementations must always deterministically return the same legal value after a given illegal value is written.

2. Load and Store Instructions

This section of manual covers the memory access instructions available in RISC-V Architecture. There are different instructions available for 8 bit, 16 bit, 32 bit and 64 bit access.

2.1 RV 32I

RV32I deals with the 32 bit instruction that are used for load and store operations. The instructions are broadly classified as register-register and immediate instructions

2.1.1 Load-Store Instructions

Load-store instructions transfer **data between memory and processor registers**. The **LW** instruction **loads** a **32-bit value** from memory into the **destination register** (**rd**). **LH** loads a **16-bit value** from memory, then **sign-extends to 32-bits** before storing in **rd**. **LHU** loads a 16-bit value from memory but then **zero extends** to 32-bits before storing in rd. **LB** and **LBU** are for 8-bit values.

The **SW**, **SH**, and **SB** instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register to memory. The **load or store address** should always aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The processor will generate a misaligned access, if the addresses are not aligned properly. If the load or store instruction tries to access an invalid memory, a load/store access fault is generated. An invalid memory can arise because of PMP access controls or unavailable memory address.

2.1.1.1 LB

The **Load Byte** (**LB**) instruction, moves a byte from memory to register. The instruction is used for signed integers.

Syntax

```
lb rd, imm(rs 1 )
```

where,

rd - destination register
imm - immediate data
rs1 - source register 1

Description

The **LB** is a **data transfer instruction**, defined for 8-bit values. It works with signed integers and places the result in the **LSB** of **rd** and fills the upper bits of **rd** with copies of the sign bit.

Usage

```
lb x5, 40(x6)           # x5 ← valueAt[x6+40]
```

2.1.1.2 LBU

The Load Byte, Unsigned (LBU) instruction, moves a byte from memory to register. The instruction is used for unsigned integers:

Syntax

```
lbu rd, imm(rs 1 )
```

where,

rd - destination register
imm - immediate data
rs1 - source register 1

Description

The **LBU** instruction, is defined for 8-bit values. It works with unsigned integers and places the result in the **LSB** of **rd** and zero-fills the upper bits of **rd**.

Usage

```
lbu x5, 40(x6)          # x5 ← valueAt[x6+40]
```

2.1.1.5 LW

The Load Word (LW) instruction, moves a word, 32-bit value, from memory to register. The instruction is used for signed values.

Syntax

```
lw rd, imm(rs1)
```

Description

The LW instruction, is defined for 32-bit values. It works with signed integers and places the result in the LSB of rd and fills the upper bits of rd with copies of the sign bit.

Usage

```
lw x5, 40(x6)      # x5 ← valueAt[x6 + 40]
```

2.1.1.6 SB

Store Byte (SB) instruction, stores 8-bit values from a register to memory.

Syntax

```
sb rs2,offset(rs1)
```

where,

rs1 - base register
rs2 - source register
offset - 12-bit integer value

Description

The SB is a store type instruction which **stores 8-bit values** from the l**ow bits of a register rs2** to memory. The **low-order byte** of the register **rs2** is copied to memory while the rest of the register is ignored and is unchanged.
The **address** to which the byte will be stored to in the memory, is calculated at run time by **adding** an **offset** to a **rs1** .

Usage

```
sb x1,0(x5)        # x1 ← valueAt[x5 + 0]
```

2.1.1.8 SW

Store Word (SW) instruction, stores 32-bit values from a register to memory.

Syntax

```
sw rs2 , offset(rs1 )
```

where,

rs1 - base register
rs2 - source register
offset - 12-bit integer value

Description

The SW is a store type instruction which stores **32-bit values from the low bits of register rs2** to memory. The word from the register **rs2** is copied to memory. The address to which the word will be stored to in the memory, is calculated at run time by adding an **offset** to a **base register**.

Usage

Store the 32-bit value in **x1** register to location pointed to by **x5**.

```
sw x1, 0(x5)       # mem[x5 + offset] ← x1
```

2.1.2 Immediate instructions

Immediate i**nstructions** are those which **contain the actual data** to be operated upon, rather than the addresses of the data. It is directly encoded as part of an instruction.

2.1.2.1 LUI

The **Load Upper Immediate** (LUI) instruction, copies the **20-bit immediate value** to the upper 20 bits of the destination register (**rd**) and **resets the lower 12 bits to zero**.

Syntax

```
lui rd, imm
```

where,

`rd` - **destination register**
`imm` - **immediate Data**

Description

The LUI instruction, copies the immediate value to the upper 20 bits of the **destination register** (**rd**).
The lower 12 bits of the destination register is **reset to zero**. This instruction is usually used, when a register needs to be populated with a **large value**. The immediate value can be represented in hexadecimal or decimal format. In a RV64 systems, the most significant bit is **sign extended to fill the most significant 32 bits** (bits **63 – 32**).
The destination registers can be any of the 31 base registers. The **x0** register can be used as a source register only, but not as a destination register.

2.2 RV 64I

RV 64I deals with the 64 bit instructions that are used for load and store operations. The instructions are broadly classified as register-register and immediate instructions

2.2.1 Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. The **LD** instruction **loads a 64-bit** value from memory into the **destination register** (**rd**). The **SD** instructions **store 64-bit value** in the register **to memory**.
The load or store address should always aligned for 64 bits. The processor will generate a misaligned access, if the addresses are not aligned properly.

2.3 Pseudo Instructions

RISC-V provides several pseudo-instructions which are simple to understand, easy to use and translate or expand to their base instructions. Pseudo instructions supported by RISC-V have the format shown as follows.

```
OpCode destination register, source register
```

Where content of the source register is copied into the destination register, and is read as, destination register ←– source register

2.3.1 Load pseudo instructions

2.3.1.1 MV

Move (MV) instruction to copy contents of one register to another.

Syntax

```
mv rd, rs1
```

Translation

```
addi rd,rs1,0
```

where,

`rs1` - source register 1
`rd` - destination register

Usage

```
mv x6, x5
```

Description

Move (**MV**) instruction is a simple “Copy Register”, assembler pseudo-instruction which copies the contents of one register to another register.
This assembler pseudo-instruction translates to add immediate **ADDI** instruction.

This instruction translates to: `addi x6,x5,0`
Assuming `x5` has a value `3` and `x6` is initialized to `0,` after move instruction, `x6` will have the value `3`.

2.3.1.2 LI

The **Load Immediate** (**LI**) loads a register (rd) with an immediate value given int the instruction.

Syntax

```
li rd, CONSTANT
```

Description
The **LI** instruction loads a register (**rd**) with an **integer value**. With this instruction both positive and negative values can be loaded into the register.

Usage

```
li x5,100               # x5 ←–100
li x5,–170              # x5 ←––170
```

2.3.1.3 LA

The **Load Address** (**LA**) loads the location address of the specified SYMBOL.

Syntax

```
la rd, SYMBOL
```

Description
The **LA** directive is an **assembler pseudo-instruction** which computes a **pointer-sized effective address** of the **SYMBOL**, but **does not perform any memory access**. The effective address itself is then stored in register **rd**. Depending on the addressing mode, the instruction expands to

```
lui rd, SYMBOL[31:12]
addi rd, t0, SYMBOL[11:0]
```

where **SYMBOL[31:12]** is the **upper 20 bits of SYMBOL**, and **SYMBOL[11:0]** is the lower 12 bits of **SYMBOL**.

Usage

```
.data
NumElements: .byte 6
.text
la x5, NumElements  # x5 ←– addr[NumElements]
```

As an example, '**NumElements**' **SYMBOL** has a location address **'10010074'**. When **LA** is given, this address, **'10010074'** is loaded into register **x5**

2.3.1.4 SEXT.W

Sign Extend Word (SEXT.W) instruction sign extends a 32-bit value to 64-bits or 128-bits.

Syntax

```
sext.w rd, rs1
```

where,

rs1 - source register 1
rd - destination register

Translation

```
addiw rd,rs1,x0
```

Description
SEXT.W is an assembler pseudo-instruction which is available only for **64-bit** and **128-bit** machines. This instruction **sign extends the lower 32 bits** of value in **rs1** to **64** or **128** bits with the result being placed in the register **rd. SEXT.W** is useful when a 32-bit signed value must be extended to a larger value on 64-bit or 128-bit machine.

Usage

```
sext.w x6, x5                  # x6 ←– x5
```

Assuming register **x5** is loaded with value **0xfda961a6e88e974d, SEXT.W** sign extends this value to **0xfffffffe88e974d**, and is stored in **x6**. As this instruction translates to **ADDIW**, the sign extension translates to, **x6 = x5+0**

2.3.1.5 NEG

Negate (**NEG**) instruction computes **two's complement of a value**.

Syntax

```
neg rd, rs 1
```

Translation

```
sub rd, x0, rs1
```

where,

`rs1` - source register 1
`rd` - destination register

Description

NEG instruction arithmetically negates the contents of rs 1 and places the result in register **`rd`**. This instruction translates to instruction **Subtraction** (**SUB**) where the contents of **`rs1`** is subtracted from zero.

Usage

```
neg x6, x5              # x6 ←– x5
```

Assuming **`x5`** is initialized to **`1`**, negating **`x5`** results in **`–1`** which is stored in **`x6`**. As this instruction translates to instruction **SUB**, the negation is computed as, **`x6 = 0–x5`**.

Exception

Overflow can only occur when the most negative value is negated. Overflow is ignored.

2.3.1.6 NEGW

Negate Word (**NEGW**) instruction computes the two's complement of a 32-bit value.

Syntax

```
negw rd, rs1
```

Translation

```
subw rd, x0, rs1
```

where,

`rs1` - source register 1
`rd` - destination register

Description

Similar to instruction **NEG**, the **NEGW** is used to **negate a 32-bit number** stored in **`rs1`** with the result being stored in register **`rd`**. **NEGW** translates to **SUBW** where the 32-bit number in **`rs1`** is subtracted from zero.

Usage

```
negw x6, x5             # x6 ←– x5
```

Assuming register **`x5`** is initialized to the value **`168496141`**, negating **`x5`** results in **`–168496141`** which is stored in **`x6`**. As this instruction translates to **SUBW**, the negation is computed as, **`x6 = 0–x5.`**

2.3.1.7 SEQZ

Set If Equal to Zero (**SEQZ**) instruction provides an indication if a register's content is zero.

Syntax

```
seqz rd, rs1
```

Translation

```
sltiu rd,rs1,1
```

Description

RISC-V provides a simple pseudo-assembler instruction, **SEQZ**, to check if the contents of the register **`rs1`** , **is zero or not**. Indication is provided by a single bit value 0 if the register content is not 0 or **value `1`**, i**f the**

register content is zero. **SEQZ** performs an unsigned comparison against 1. Since the comparison is unsigned, the only value less than 1 is 0. Hence if the comparison holds true, register **rs1** must contain **0**.

Usage

```
seqz x6, x5                    # x6 ←– (x5 = 0) ? 1:0
                               # x6 = 1
```

Assuming register **x5** contains **0**, **SEQZ** instruction writes value **1** into register **x6**.

2.3.1.8 SNEZ

Set If Not Equal to Zero (**SNEZ**) instruction provides an indication if a register contains non-zero value.

Syntax

```
snez rd, rs1
```

Translation

```
sltu rd, x0, rs1
```

Description

SNEZ is a pseudo-assembler instruction that is used to check if the contents of a rs1 , is a non-zero value. This instruction sets value of register rd to 1 if the rs 1 is a non-zero value or sets rd to 0 otherwise. This instruction is implemented with an unsigned comparison against 0 using its base instruction **SLTU**. Since it is an unsigned comparison, the only value less than 0 is 0 itself. Therefore, if the less-than condition holds, the value in **rs1** must not be 0.

Usage

```
snez x6, x5                    # x6←– (x5 != 0) ? 1:0
                               # x5 = 9
                               # x6= x0<x5 = 0<9 = 1
                               #x6 = 1
```

Assuming **rs1** (**x5**) is initialized to value 5, since this is greater than 0 value 1 is written into **rd**

2.3.1.9 SLTZ

Set If Less Than Zero (**SLTZ**) is a signed instruction which examines if a register's content is less then zero and indicates accordingly.

Syntax

```
sltz rd, rs 1
```

Translation

```
slt rd, rs 1 , x0
```

Description

SLTZ is **a signed pseudo-assembler instruction** which translates to **SLT**, examines if the value in register **rs1** is less than zero. If register value found to be less than zero, a value 1 is stored in register **rd**. Otherwise the value 0 is stored.

Usage

```
sltz x6, x5                # x6 ←– (x5 < 0) ? 1:0
                           # x5 = –2
                           # x6 = x5<0 = –2<0 = 1
                           # x6 = 1
```

2.3.1.10 SGTZ

Set If Greater Than Zero (**SGTZ**) instruction examines if a register contains a value is greater than zero and indicates it accordingly.

Syntax

```
sgtz rd, rs1
```

Description

SGTZ is a signed pseudo-assembler instruction which examines if the value in register rs 1 is greater than zero. If found true, value 1 is stored to register (**rd**) or value 0 is stored otherwise.

Usage

```
sgtz x6, x5              # x6←– (x5 > 0) ? 1:0
                         # x5= 9
                         # x6= x0<x5 = 0<9 = 1
                         # x6= 1
```

Assume **rs1** (**x5**) is initialized to **9**, since this is greater than 0. Value 1 will be stored in **rd** (**x6**).

3. Bitwise Instructions

3.1 RV 32I

RV 32I deals with the 32 bit instruction that are used for **bit manipulation**. The instructions are broadly classified as **register-register** and **immediate instructions**

3.1.1 Register to Register Instructions

Register operations involve **both the operands as registers**. The operation is performed on the value in the register and result is stored in **destination register** (**rd**). The **source and destination registers** can be **any of the 31 base registers**. The **x0** register can be used as a source register only, but not as a destination register. **32 bits of result** is written to the destination register.

3.1.1.1 SLL

Shift Logical Left (**SLL**) performs logical left on the value in register (rs 1) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
sll rd, rs1 , rs2
```

where,

rd - destination register
rs1 - source register 1
rs2 - source register 2

Description

A **SLL** of one position moves **each bit to the left by one**. The **low-order bit** (**the right-most bit**) is **replaced by a zero bit** and the **high-order bit** (the left-most bit) is **discarded**.

Usage

```
li x5,4                  # x5 ←– 2
li x3,2                  # x3 ←– 2
sll x1,x5,x3             # x1 ←– x5 << x3
```

x1 will have a value **16**.

3.1.1.2 SRL

Shift Logically Right (**SRL**) performs logical Right on the value in register (**rs1**) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
srl rd, rs1,rs2
```

Description

A **SRL** of one position moves each bit to the Right by one. The **high-order bit** (**the left-most bit**) is **replaced by a zero bit** and **the low-order bit** (the Right-most bit) is **discarded**.

Usage

```
li x5, 4                # x5 ← 4
li x3, 2                # x3 ← 2
srl x1, x5, x3          # x1 ← x5 >> x3
```

x1 will have a value **1**.

3.1.1.3 SRA

Shift Right Arithmetic (**SRA**) performs right shift on the value in register (**rs1**) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
sra rd, rs1 , rs2
```

Description
SRA directive performs an **arithmetic shift right by 0 to 32 places**. **The vacated bits at the most significant end are filled with zeros if the original value (the source operand) was positive**.
The vacated bits are filled with ones if the original value was negative.
This is known as "**sign extending**" because the most significant bit of the original value is the sign bit for 2's complement numbers, i.e. 0 for positive and 1 for negative numbers. **Arithmetic shifting therefore preserves the sign of numbers**.

Usage

```
li x5, 4                # x5 ← 4
li x3, 2                # x3 ← 2
sra x1, x5, x3          # x1 ← x5 >> x3
```

3.1.1.4 OR

OR directive performs **bit-wise logical OR** operation between contents of register (**rs1**) and contents of register (**rs2**) and stores in (**rd**) register.

Syntax

```
or rd, rs1,rs2
```

Description
A **bit-wise OR** is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits.

Usage

```
li x5, 0x0100               # x5 ← 0x0100
li x3, 0x0010               # x3 ← 0x0010
or x1, x5, x3               # x1 ← x5|x3
```

x1 will have a value **0x0110**.

3.1.1.5 XOR

XOR performs bit-wise binary **Exclusive-OR** operation on the source register operands.

Syntax

```
xor rd, rs1 , rs2
```

Description
A **bit-wise XOR** is a binary operation that takes two bit patterns of equal length and performs the **logical inclusive XOR operation on each pair of bits**.

Usage

```
li x5, 0x0100               # x5 ← 0x0100
li x3, 0x0010               # x3 ← 0x0010
xor x1, x5, x3              # x1 ← x5|x3 (x1 ← 0x0110)
```

3.1.1.6 NOT

NOT is a **bit-wise invert operation**, which performs a one's complement arithmetic.

Syntax

```
not rd, rs1
```

Translation

```
xori rd, rs 1 , -1            # [-1 = 0xFFFFFFFF]
```

Description

NOT instruction **flips each bit of a register**. This instruction translates to an exclusive OR operation XORI and implements the negation. The result is loaded into the destination register (**rd**).

Usage

```
not x6, x5                # x6 ←~ x5
```

Assuming register **x5** (**rs1**) is initialized to value 1, on applying the NOT instruction on **x5**, 1 will be **xored** (since **XORI** is the base instruction for **XORI**) with **-1**, resulting to **-2** (stored in **x6**). Now let's assume **x5** is initialized to value **-1**, on applying **NOT** to it results in a value **0**.

3.1.1.7 SLT

Set Less Than (**SLT**) perform the **signed and unsigned comparison between** (**rs1**) and (**rs2**) and stores the result in (**rd**).

Syntax

```
slt rd, rs1, rs2
```

Description

SLT performs signed and unsigned compares respectively, writing **1** to **rd** if **rs1< rs2** , **0** otherwise.

Usage

```
li x5, 3                  # x5 ←- 3
li x3, 5                  # x3 ←- 5
slt x1, x5, x3            # x1 ←- x5 < x3
```

x1 will have a value **1**.

3.1.1.8 SLTU

Set Less Than Unsigned (**SLTU**) perform the signed and unsigned comparison between (**rs1**) and (**rs2**) and stores the result in (**rd**).

Syntax

```
sltu rd, rs1,rs2
```

Description

SLTU sets **rd** to **1** if **rs2** **is not equal to zero**, otherwise sets **rd** to zero. **SLTU** performs signed and unsigned compares respectively, writing **1** to **rd** if **rs1** != **rs2** , **0** otherwise.

Usage x1 will have a value 1.

```
li x5, 3                  # x5 ←- 3
li x3, 5                  # x3 ←- 5
slt x1, x5, x3            # x1 ←- x5 < x3
```

3.1.2 Immediate instructions

Any instruction which **contains an operand** that is directly encoded as part of an instruction is called an **immediate instruction** and the operand as immediate operand. This section covers shift and logical operations with immediate operands as part of the instruction.

3.1.2.1 SLLI

Shift Logically Left Immediate (**SLLI**) performs logical left on the value in register (**rs1**) by the shift amount held in the register (**imm**) and stores in (**rd**) register.

Syntax

```
slli rd, rs 1 ,imm
```

Description

A **SLLI** of one position **moves each bit to the left by one**. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
slli x1, x1, 1                # x1 ← x1<<1
```

3.1.2.2 SRLI

Shift Logically Right Immediate (SRLI) performs logical Right on the value in register (**rs1**) by the shift amount held in the register (**imm**) and stores in (**rd**) register.

Syntax

```
srli rd, rs1 ,imm
```

Description

A Shift Right Logical Immediate (**SRLI**) of one position **moves each bit to the Right by one**. The most significant bit is replaced by a zero bit and the least significant bit is discarded.

Usage

```
srli x1, x1, 1                # x1 ← x1>>1
```

3.1.2.3 SRAI

Shift Right Arithmetic Immediate (**SRAI**) performs right shift on the value in register (**rs1**) by the shift amount held in the (**imm**) and stores in (**rd**) register.

Syntax

```
srai rd, rs1 ,imm
```

Description

SRAI is arithmetic shift right of a number by 'N' places. The vacated bits at the most significant end are filled with value of sign bit (**0** for **+ve** sign and **1** for **-ve** sign). This is known as "**sign extending**".
The most significant bit of the original value is the sign bit for **2's complement** numbers.

Usage

```
srai x1,x1,1                  # x1 ← x1>>1
```

3.1.2.4 ANDI

AND Immediate (**ANDI**) performs binary operation between contents of register (**rs1**) and immediate data (**imm**) and stores in (**rd**) register.

Syntax

```
andi rd, rs1 ,imm
```

Description

A **Bitwise ANDI** is a binary operation that takes two bit patterns of equal length and performs the logical inclusive AND Immediate operation over each bits. The source and destination registers can be any of the 31 base registers. The **x0** register can be used as a source register only, but not as a destination register. 32 bits of result is written to the destination register.

Usage

```
andi x5, x5, 4                # x5 ← x5&4
```

3.1.2.5 ORI

OR Immediate (**ORI**) performs binary operation between register (**rs1**) and Immediate data (**imm**) and stores in (**rd**) register.

Syntax

```
ori rd, rs1 ,imm
```

Description

A **bitwise ORI** is a binary operation that takes two bit patterns of equal length and performs the **logical inclusive OR** operation on each pair of corresponding bits.

Usage

```
li x5, 0x0100                  # x5 ← 0x0100
ori x1, x5, 0x0010             # x1 ← x5|2
```

x1 will have a value **0x0110.**

3.1.2.6 XORI

Exclusive-OR Immediate (**XORI**) performs bit-wise binary operation between register contents (**rs1**) and Immediate data (**imm**) and stores in (**rd**) register.

Syntax

```
xori rd, rs1 ,imm
```

Description

A **bitwise XORI** is a binary operation that takes two bit patterns of equal length and performs logical inclusive **XOR** operation on each pair of corresponding bits.

Usage

```
xori x5, x5, 0b100000              # x5 ← x5|0x0b100000
```

3.1.2.7 SLTI

Set Less than Immediate (**SLTI**) compares contents of register (**rs1**) and Immediate data (**imm**) and sets value in (**rd**) register.

Syntax

```
slti rd, rs1 ,imm
```

Description

A **SLTI** is a signed comparison between contents of the specified registers. If the value in register is less than the immediate value, value **1** is stored in destination register, otherwise, value **0** is stored in the destination register.

Usage

```
slti x5,x1,2                   # x5 ← x1 < 2
```

3.1.2.8 SLTIU

Set Less Than Immediate Unsigned (**SLTIU**) does comparison between register contents (**rs1**) and Immediate data (**imm**) and sets value in (**rd**) register.

Syntax

```
sltiu rd, rs1 ,imm
```

Description

A **SLTIU** is a comparison to the contents of register using **unsigned comparison**. If the value in register is less than the immediate value, the value **1** is stored in destination Register, otherwise, the value **0** is stored in destination register.

Usage

```
slti x5, x1, 2                # x5 ← x1 < 2
```

3.2 RV 64I

RV 64I deals with the 64 bit instruction that are used for bit manipulation arithmetic operations. The instructions are broadly classified as register-register and immediate instructions.

3.2.1 Register to Register Instructions

The **RV64I** register-register operations involve both the operands as 64 bit registers. The operation is performed on the value in the register and result is stored in a destination register (**rd**). The source and destination registers can be any of the **31** base registers. **x0** is read only.

3.2.1.1 SLLW

Shift Left Logical Word (**SLLW**) performs logical left on the value in register (**rs1**) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
sllw rd, rs 1 , rs 2
```

Description

A **SLLW** of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
li x3,5                 # x3 ← 5
li x1,3                 # x1 ← 3
sllw x1, x1, x3         # x1 ← x1<<x3
```

3.2.1.2 SRLW

Shift Right Logically Word (**SRLW**) performs logical right on the value in register (**rs1**) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
srlw rd, rs1 , rs2
```

Description

A **SRLW** of one position moves each bit to the Right by one. The High-order bit (the left-most bit) is replaced by a zero bit and the low-order bit (the Right-most bit) is discarded.

Usage

```
li x1, 3                # x1 ← 3
li x3, 5                # x1 ← 5
srlw x1, x1, x3         # x1 ← x1>>x3
```

3.2.1.3 SRAW

Shift Right Arithmetic Word (**SRAW**) performs Arithmetic right on the value in register (**rs1**) by the shift amount held in the register (**rs2**) and stores in (**rd**) register.

Syntax

```
sraw rd, rs1 , rs2
```

Description

SRAW is an arithmetic shift right of a word by 'N' places. The vacated bits at the most significant end are **filled with value of sign bit** (0 for **+ve** sign and 1 for **-ve** sign). This is known as "**sign extending**".
The most significant bit of the original value is the sign bit for 2's complement numbers.

Usage

```
li x1, 3                # x1 ← 3
li x3, 5                # x1 ← 5
sraw x1, x1, x3         # x1 ← x1>>x3
```

3.2.2 Immediate instructions

A **64-bit system** involves **64-bit constant** operands as part of their instructions.

3.2.2.1 SRLIW

Shift Right Logical Immediate Word (**SRLIW**) performs Logical right on the value in register (**rs1**) by the shift amount held in the immediate data (**imm**) and stores in (**rd**) register.

Syntax

```
srliw rd, rs 1 , imm
```

Description

A **SRLIW** does one position move of each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
li x3,5                 # x3 ←– 5
li x1,3                 # x1 ←– 3
srliw x1, x1, x3        # x1 ←– x1>>x3
```

3.2.2.2 SRAIW

Shift Right Arithmetic Immediate Word (**SRAIW**) performs Arithmetic right on the value in register (**rs1**) by the shift amount held in the Immediate (**imm**) and is stored in (**rd**) register.

Syntax

```
sraiw rd, rs1 ,imm
```

Description

SRAIW is an **arithmetic shift right immediate by 0 to 64 places**. The vacated bits at the most significant end are filled with zeros if the original value (the source operand) was positive. The vacated bits are filled with ones if the original value was negative. This is known as "**sign extending**" because the most significant bit of the original value is the sign bit for 2's complement numbers, i.e. 0 for positive and 1 for negative numbers. Arithmetic shifting therefore preserves the sign of numbers.

Usage

```
li x1, 3                # x1 ←– 3
sraiw x1, x1, x3        # x1 ←– x1>>x3
```

4. Arithmetic Instructions

4.1 RV 32I

RV 32I deals with the 32 bit instruction that are used for arithmetic operations. The source and destination registers can be any of the 31 base registers. The **x0** register can be used as a source register only, but not as

a destination register. The instructions are broadly classified as **register-register and immediate instructions**

4.1.1 Register to Register instructions

Register to register instruction involves, both the operands as a register. The contents of the register holds the content of the operands

4.1.1.1 ADD

Addition (**ADD**) adds the contents of two registers and stores the result in another register.

Syntax

```
add rd, rs1 , rs2
```

Description

The **ADD** instruction adds content of the two registers **rs1** and **rs2** and stores the resulting value in **rd** register. The source and destination registers can be **any of the 31 base registers**. The **x0** register can be used as a source register only, but not as a destination register. **Overflows are ignored** and the lower 32 bits of result is written to the destination register.

Usage

```
li x2, 3                # x2 ← 3
li x3, 4                # x3 ← 4
add x1, x2, x3          # x1 ← x2 + x3
```

Assuming **rs1** (**x2**) and **rs2** (**x3**) contain values 3 and 4 respectively, an addition operation on them will result in value 7 which will be stored in **rd** (**x1**). **x1** will have a value 7.

4.1.1.2 SUB

Subtraction (**SUB**) subtracts contents of one register from another and stores the result in another register.

Syntax

```
sub rd, rs1 , rs2
```

Description

The **SUB** instruction subtracts content of the source register **rs2** from **rs1** and stores the value in the register **rd**. **Overflows are ignored** and the lower **XLEN** bits of the result is written to **rd**.
The source and destination registers can be any of the 31 base registers. The **x0** register can be used as a source register only, but not as a destination register. The overflows as well as borrow are ignored and the lower 32 bits of result is written to the destination register.

Usage

```
li x2, 4                # x2 ← 4
li x3, 3                # x3 ← 3
sub x1, x2, x3          # x1 ← x2 - x3
```

x1 will have a value **1**.

4.1.1.3 MUL

Multiplication (**MUL**) calculates the product of the **multiplier** in source register 1 (**rs1**) and **multiplicand** in source register 2 (rs2), with the resulting **product** being stored in destination register (**rd**).

Syntax

```
mul rd, rs1,rs2
```

Description

MUL calculates the product of two **XLEN**-bit operands in the source registers 1 and 2 (**rs1** , **rs2**). This instruction stores the less significant part of the result in the destination register and **any overflow is ignored**.

Usage

```
mul x4, x9, x13             # x4 ← Low Bits [x9 * x13]
```

4.1.1.4 MULH

Multiply signed and return upper bits (**MULH**)) calculates the product of signed values in source registers (**rs1**) and (**rs2**) and stores result in the specified destination register (**rd**).

Syntax

```
mulh rd, rs1 , rs2
```

Description

MULH calculates the **product of signed multiplier and signed multiplicand** (present in the two source registers specified respectively), and places the upper **XLEN** bits of the full **2*XLEN** product, into the destination register. **MULH** has to be used with **MUL** to get the complete **2*XLEN** bits result.

Usage

```
li x1,-80               # x1 ← -80
li x5,20                # x5 ← 20
mulh x5, x5, x1         # x5 ← High Bits[x5*x1]
```

4.1.1.5 MULHU

Multiply Unsigned and return upper bits (**MULHU**)) calculates the product of two unsigned values in source registers **rs1** and **rs2** . The resulting value is placed in the specified destination register (**rd**).

Syntax

```
mulhu rd, rs1 , rs2
```

Description

MULHU multiplies two unsigned operands in the source registers and the most significant part of result is stored in the destination register.

Usage

```
li x1,-80               # x1 ← -80
li x5,20                # x5 ← 20
mulhu x5, x5, x1        # x5 ← High Bits [x5*x1]
```

4.1.1.6 MULHSU

Multiply Signed-Unsigned and return upper bits (**MULHSU**)) calculates the product of a signed value in source register **rs1** with an unsigned value in source register **rs2** and the resulting product is stored in destination register, **rd**.

Syntax

```
mulhsu rd, rs1 , rs2
```

Description

MULHSU computes the product of the signed, most significant word of the multiplier and the unsigned, least significant word of the multiplicand. The most significant part of the resulting product is stored in the specified destination register. The **resulting value is a signed value**.

Usage

```
li x1,-80               # x1 ← -80
li x5,20                # x5 ← 20
mulhsu x5, x5, x1       # x5 ← High Bits[x5*x1]
```

4.1.1.7 DIV

Division (**DIV**) performs division on the value in source register (**rs1**) with the value in the source register (**rs2**) and stores quotient in (**rd**) register.

Syntax

```
div rd, rs1 , rs2
```

Description
DIV does the division of operands in source registers and stores quotient in the destination register. Both **operands and the result are signed values**.

Usage
```
li x9, -400             # x9 ← -400
li x13, 200             # x13 ← 200
div x4, x9, x13         # x4 ← x9/x13
```

4.1.1.8 DIVU

Division Unsigned (**DIVU**) performs unsigned Division on the value in source register (**rs1**) by the value in the source register (**rs2**) and stores quotient in the destination register (**rd**).

Syntax
```
divu rd, rs 1 , rs 2
```

Description
DIVU does the division of unsigned operands in source registers and stores quotient in the destination register. Both operands and the result are unsigned values.

Usage
```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
divu x4, x9, x13        # x4 ← x9/x13
```

4.1.1.9 REM

Reminder (**REM**) performs division on the value in source register (**rs1**) with the value in the source register (**rs2**) and stores remainder in (**rd**) register

Syntax
```
rem rd, rs1 , rs2
```

Description
REM does the **signed division of operands** in source registers and stores the **remainder** in the destination register. Both operands and the result are signed values.

Usage
```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
rem x4, x9, x13         # x4 ← x9%x13
```

NOTE:
Sometimes a programmer needs both quotient and remainder. In such cases it is recommended to perform DIV first and REM later.

4.1.2 Immediate Instructions

Instructions involving a constant operand are immediate instructions. Here we are going to load and store immediate instructions.

4.1.2.1 LI

Load Immediate (**LI**) load register rd with a value that is immediately available

Syntax

```
li rd, imm
```

Description

The LI instruction loads a positive or negative value that is immediately available, without going into memory. The value maybe a 16-bit or a 32-bit integer.

Usage

```
li x5, 24               # x5 ← 24
```

4.1.2.2 ADDI

Add Immediate (**ADDI**) adds content of the source registers **`rs1`** , immediate data (**`imm`**) and store the result in the destination register (**`rd`**).

Syntax

```
addi rd, rs1 , imm
```

Description

The ADDI instruction adds content of a source register with an absolute value and stores the result in the destination register. **Overflows are ignored** and the lower 32 bits of result is written to the destination register.

Usage

```
li x2,24                # x2 ← 24
addi x1, x2,64          # x1 ← x2 + 64
```

x1 will have a value 88.

4.2 RV 64I

RV 64I deals with the 64 bit integer instructions that are used for arithmetic operations. The instructions are broadly classified as **register-register** and **immediate instructions**.

4.2.1 Register to Register instructions

The register operations involve both the operands as registers. The operation is performed on the value in the register and result is stored in destination register (**rd**).

4.2.1.1 ADDW

Add Word (**ADDW**) adds content of the source registers (**rs1,rs2**) and stores the result in the destination register (**rd**).

Syntax

```
addw rd, rs1 , rs2
```

Description

The ADDW instruction adds content of the two source registers and stores the value in the destination register. The overflows are ignored and the lower 64 bits of result is stored in destination register.

Usage

```
addw x4, x9, x13              # x4 ← x9 + x13
```

4.2.1.2 SUBW

Subtract Word (**SUBW**) subtracts content of the source registers (**rs1,rs2**) and store the result in the destination register (**rd**).

Syntax

```
subw rd, rs1, rs2
```

Description

The SUBW instruction **subtracts** content of the source register **rs2 from rs1** and stores the value in the destination register (**rd**). The **overflows as well as borrow are ignored** and the lower 64 bits of result is written to the destination register.

Usage

```
li x2, 456              # x2 ← 456
li x3, 123              # x3 ← 123
subw x1, x2, x3         # x1 ← x2 – x3
```

x1 will have a value 333.

4.2.1.3 REMU

Reminder Unsigned (REMU) performs division on the value in source register (**rs1**) with the value in the source register (**rs2**) and stores remainder in (**rd**) register.

Syntax

```
remu rd, rs 1 , rs 2
```

Description

REMU does the division of operands in source registers and stores remainder in the destination register. Both **operands and the result are unsigned values**.

Usage

```
li x9, 400              # x4 ← 400
li x13,200              # x4 ← 200
remu x4, x9, x13        # x4 ← x9%x13
```

Note:

Sometime's a programmer needs both quotient and remainder. In such cases it is recommended to perform DIV first and REM later.

4.2.1.4 MULW

Multiplication Word (MULW) directive multiplies contents of register **rs1** with that of register **rs2** and stores result in register **rd**. Only the lower order 32-bits of the result are used, which is sign extended to the full length of the register.

Syntax

```
mulw rd, rs1 , rs2
```

Description

MULW does the multiplication of operands in source registers and stores result in the destination register. Only the lower order 32-bits of the result are used the lower 32 bits are signed extended to the full length of the register. This instruction is used to properly emulate 32-bit multiplication on a 64-bit or 128-bit machine. Only the least-significant 32 bits of Reg1 and Reg2 can possibly affect the result. If you want the upper 32-bits of the full 64-bit result use the **MUL** instruction on a 64-bit machine.

Usage

```
mulw x4, x9, x13              # x4 ← x9*x13
```

4.2.1.5 DIVW

Divide Word (**DIVW**) performs Division on the value in source register (**rs1**) with the value in the source register (**rs2**) and stores **quotient** in (**rd**) register.

Syntax

```
divw rd, rs1 , rs2
```

Description

DIVW does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are signed values, only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
divw x4, x9, x13        # x4 ← x9/x13
```

4.2.1.6 DIVUW

Divide Unsigned Word (**DIVUW**) performs division on the value in source register (**rs1**) with the value in the source register (**rs2**) and stores quotient in (**rd**) register.

Syntax

```
divuw rd, rs1 , rs2
```

Description

DIVUW does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are unsigned values, only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
divuw x4, x9, x13       # x4 ← x9/x13
```

4.2.1.7 REMW

Reminder Word (**REMW**) performs Division on the value in source register (**rs1**) with the value in the source register (**rs2**) and **stores remainder** in (**rd**) register.

Syntax

```
remw rd, rs1 , rs2
```

Description

REMW does the division of operands in source registers and stores remainder in the destination register. Both operands and the result are signed values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
remw x4, x9, x13        # x4 ← x9%x13
```

NOTE:
Sometime, a programmer might need both quotient and remainder. In such cases it is recommended to perform DIV first and REM later.

4.2.1.8 REMUW

Reminder Unsigned Word (**REMUW**) performs Division on the value in source register (**rs1**) with the value in the source register (rs2) and **stores remainder** in (rd) register.

Syntax

```
remuw rd, rs1 , rs2
```

Description
REMUW does the division of operands in source registers and stores remainder in the destination register. Both operands and the result are unsigned values. The least significant 32 bits of the operands are used and the 32-bit result is signed-extended.

Usage

```
li x9, 400              # x9 ← 400
li x13,200              # x13 ← 200
remuw x4, x9, x13       # x4 ← x9%x13
```

NOTE:
Sometime, a programmer might need both quotient and remainder. In such cases it is recommended to perform DIV first and REM later.

4.2.2 Immediate Word Instructions

Instructions which involve a 32-bit constant operand have the ”W” to specify 32-bit operations to be performed on them.

4.2.2.1 ADDIW

Add Immediate Word (**ADDIW**) adds content of the source registers **rs1** , **imm** and store the result in the destination register (**rd**).

Syntax

```
addiw rd, rs1 , imm
```

Description
The **ADDIW** instruction adds content of the **two source registers** and stores the value in the destination register. This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic. The result is then truncated to 32-bits, signed-extended to 64 or 128-bits and placed in destination register. The overflows are ignored and the lower 64 bits of result is written to the destination register.

Usage

```
li x9,456               # x9 ← 456
addiw x4, x9,123        # x4 ← x9 + 123
```

5. Control Transfer Instructions

5.1 Branch Instructions

A branch instruction in a program causes the system to execute a different instruction sequence, making the system deviate from its normal course of action of executing instructions in sequence.
Branches are useful for implementing logical constructs since the architecture allows compares and dependent branches to be scheduled in the same cycle.

5.1.0.1 BEQ

Branch If Equal (**BEQ**) the contents of source register **`rs1`** is compared with source register **`rs2`** , if found equal, the control is transferred to the specified **label**.

Syntax

```
beq rs 1 , rs 2 , label
```

Description
The **BEQ** instruction compares contents of (**`rs1`**) is compared to the contents of (**`rs2`**). If equal, control jumps.
The **target address** is given as a **PC-relative offset**.
More precisely, the offset is **sign-extended**, **multiplied by 2**, and **added to the value of the PC**. The value of the PC used is the address of the instruction following the branch, not the branch itself.
The offset is multiplied by 2, since **all instructions must be half word aligned**.

Usage

```
loop: addi x5, x1, 1                # x5 <– x1 + 1
beq x0, x0, loop                    # x0 = x0 jump to loop
```

5.1.0.2 BNE

Branch If Not Equal (**BNE**) the contents of source register **`rs1`** , is compared with source register **`rs2`** if they are not equal control is transferred to the label as mentioned.

Syntax

```
bne rs1 , rs2 , label
```

Description
The BNE instruction compares contents of (**`rs1`**) is compared to the contents of (**`rs2`**). If not equal, control jumps. The target address is given as a PC-relative offset.

Usage

```
label: addi x4, x9,123              # x4 <– x9 + 123
bne x4, x9, label                   # x4 = x9 jump to label
```

5.1.0.3 BLT

Branch If Less Than (**BLT**) the contents of source register **`rs1`** , is compared with contents of source register **`rs2`** . If (**`rs1`**) is less than (**`rs2`**) control is transferred to the **`label`** as mentioned.

Syntax

```
blt rs1 , rs2 , label
```

Description
The **BLT** instruction compares contents of (**`rs1`**) is compared to the contents of (**`rs2`**). If (**`rs1`**) contents is less than (**`rs2`**)(**signed comparison**), control jumps. The target address is given as a PC-relative offset.

Usage

```
label: addi x4, x9, 123             # x4 <– x9 + 123
blt x4, x9, label                   # x4 < x9 jump to label
```

5.1.0.4 BLTU

Branch If Less Than Unsigned (**BLTU**) the contents of source register **`rs1`** , is compared with contents of source register **`rs2`** if (**`rs1`**) is less than (**`rs2`**) control is transferred to the **`label`** as mentioned.

Syntax

```
bltu rs 1 , rs 2 , label
```

Description

The **BLTU** instruction compares contents of (**rs1**) is compared with the contents of (**rs2**). If (**rs1**) contents is less than (**rs2**), (**unsigned comparison**) control jumps. The target address is given as a PC-relative offset.

Usage

```
loop: addi x1, x0, 1                # x1 ←- x0 + 1
addi x5, x0, 3                      # x5 ←- x0 + 3
bltu x1, x5, loop                   # x1 < x5 jump to loop
```

5.1.0.5 BGE

Branch If Greater Than or Equal, **signed** (**BGE**) the contents of source register **rs1** , is compared with contents of source register **rs2** if (**rs1**) is greater than (rs2) control is transferred to the **label** as mentioned.

Syntax

```
bge rs1 , rs2 , label
```

Description

The **BGE** instruction compares contents of (**rs1**) with the contents of (**rs2**). If (**rs1**) contents is greater than or equal to contents of (**rs2**), (**signed comparison**) control jumps to the specified location. The target address is given as a PC-relative offset.

Usage

```
label: addi x4, x9, 123             # x4 ←- x9 + 123
bge x4, x9, label                   # if x4 ≥ x9 jump to label
```

5.1.0.6 BGEU

Branch If Greater Than or Equal, Unsigned (**BGEU**) the contents of source register **rs1** , is compared with contents of source register **rs2** . If **rs1** is **greater than or equal to rs2** , control is transferred to the **label** as mentioned.

Syntax

```
bgeu rs1 , rs2 , label
```

Description

The **BGEU** instruction compares contents of (**rs1**) is compared with the contents of (**rs2**). If (**rs1**) contents is greater than (**rs2**), (**unsigned comparison**) control jumps. The target address is given as a PC-relative offset.

Usage

```
label: addi x4, x9,123              # x4 ←- x9 + 123
bgeu x4, x9, label                  # x4 ≥ x9 jump to label
```

5.1.1 Pseudo Instructions

Branching instructions in this section are pseudo or convenient instructions to be used in place of the base instructions.

5.1.1.1 BEQZ

Branch if Equal to Zero (**BEQZ**) instruction jumps to a specified location in the program if the condition, equal to zero is met.

Syntax

```
beqz rs1 , label
```

Translation

```
beq rs1 , x0, label
```

Description
The **BEQZ** translates to **`beq rs1,x0,label,`** as the expansion reveals, the (**`rs1`**) contents is compared with the zero register (**`x0`**) and the program counter branches to the specified label if the condition equal to zero is met.

Usage

```
li x6, 0                    # x6 = 0
loop: li x5, x5, 100        # Example operation
beqz x6, loop               # x6 = 0 branch to loop
```

Assume **`rs1`** (**`x6`**) is initialized to 0 and there is an example operation within the specified **label** (**`loop`**).
BEQZ on register **`rs1`** (**`x6`**) will shift the program counter to the specified label since the contents of **`rs1`** (**`x6`**) is indeed **0**.

Part 2:

Assembly, object, and executable files

In this part we are going to study complete code example exploiting the presented instructions in part 1.
We present the main concepts and elements of assembly, object, and executable files.

2.0 Generating native programs

In this section, we discuss how a program written in a high-level language, such as C, is translated into a native program. A native program is a program encoded using instructions that can be directly executed by the computer hardware, without help from an emulator or a virtual machine . These programs are usually automatically translated from programs written in high-level languages, such as C, by tools like compilers, assemblers, and linkers.
A program written in a high-level language, such as C, is encoded as a plain text file. High-level languages are designed to be agnostic of ISA and they are composed of several abstract elements, such as variables, repetition, or loop statements, conditional statements, routines, etc..
The following code shows an example of a program written using the C language, which is a high-level language. The program is composed with two source files : the first file contains the main() function , the second contains func() function.
Note that in the first file with main() function we denote func() as external : extern int func().

The principal file – **main.c**

```
#include <stdio.h>
extern int func();

int main()
{
int v,r;
        scanf("%d",&v);
        r = func(v);
        return r;
}
```

The second file – **func.c**

```
int func(int a)
{
        return a*10;
}
```

A compiler is a tool that translates a program from one language to another . Usually, programming language compilers are employed to translate programs written in high-level languages into assembly language.
The following are essential compiler commands:

Compiler command to produce executable code in **test_func** file. This operation include compilation and link edition for two functions.

```
ubuntu@ubuntu:~/assembly$ gcc –o test_func main.c func.c
```

Compiler command to produce assembly file - **main.s**.

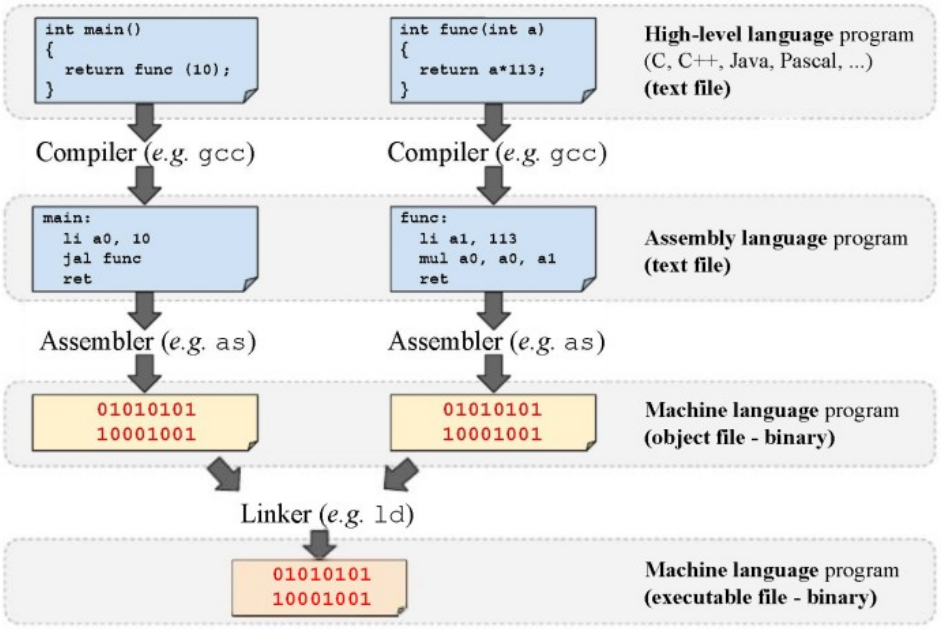
```
ubuntu@ubuntu:~/assembly$ gcc –S main.c
```

Compiler command to produce assembly file - **func.s**.

```
ubuntu@ubuntu:~/assembly$ gcc –S func.c
```

Compiler command to produce executable file from assembly codes.

```
ubuntu@ubuntu:~/assembly$ gcc –o test_func main.s func.s
```



2.1 RISC-V simple assembly example: `Hello World`

The following is Risc-V assembler program to print "Hello World!" to **`stdout`**. **`a0-a2`** are registered parameters to linux function services, **`a7`** – contains linux function number.
The first directive **`.globl`** indicates the entry to the program.

```
.globl main      # Provide program starting address to linker

main:
        addi  a0, x0, 1      # 1 = StdOut – stdout 1
        la    a1, helloworld # load address of helloworld
        addi  a2, x0, 13     # length of our string
        addi  a7, x0, 64     # linux write system call
        ecall                # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.

        addi    a0, x0, 0   # Use 0 return code
        addi    a7, x0, 93  # Service command code 93 terminates
        ecall               # Call linux to terminate the program
.data
helloworld:      .ascii "Hello World!\n"
```

```
ubuntu@ubuntu:~/assembly$ gcc –o helloworld helloworld.s
ubuntu@ubuntu:~/assembly$ ./helloworld
Hello World!
```

We can **disassembly** code into **binary dump**:

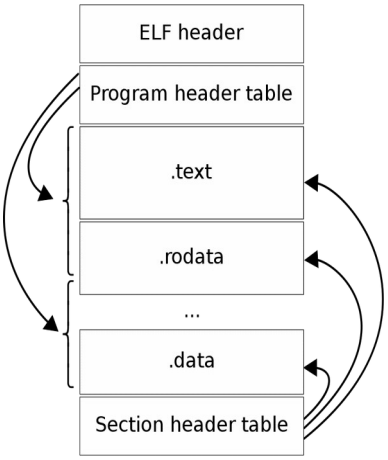
```
ubuntu@ubuntu:~/assembly$ riscv64-linux-gnu-objdump -d helloworld
```

```
0000000000000638 <main>:
 638:   4505                    li      a0,1
 63a:   00002597                auipc   a1,0x2
 63e:   9fe5b583                ld      a1,-1538(a1) # 2038 <_GLOBAL_OFFSET_TABLE_+0x8>
 642:   4635                    li      a2,13
 644:   04000893                li      a7,64          # load to a7 linux service code: 64
 648:   00000073                ecall
 64c:   4501                    li      a0,0
 64e:   05d00893                li      a7,93          # load to a7 linux service code: 93
 652:   00000073                ecall
```

2.1.2 ELF file

The GNU **`readelf`** tool can be used to display information about **ELF files**. The following command shows how the **`riscv64-unknown-elf-readelf`** tool can be used to inspect the header of the **`test_func`** executable file.

Note:
In computing, the **Executable and Linkable Format** (**ELF**), is a common standard file format for executable files, object code, shared libraries, and core dumps. By design, the ELF format is flexible, extensible, and cross-platform . For instance, it supports different endiannesses and address sizes so it does not exclude any particular processor – CPU or instruction set architecture – ISA. It is adopted by many different operating systems including Linux on different hardware platforms including RISC-V.



The following is the result of the use **`readelf`** command for RISC-V and our executable code. It shows the essential parameters of the program layout.

```
ubuntu@ubuntu:~/assembly$ riscv64-unknown-elf-readelf -h test_func
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x690
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6904 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
ubuntu@ubuntu:~/assembly$
```

2.1.3 Program sections

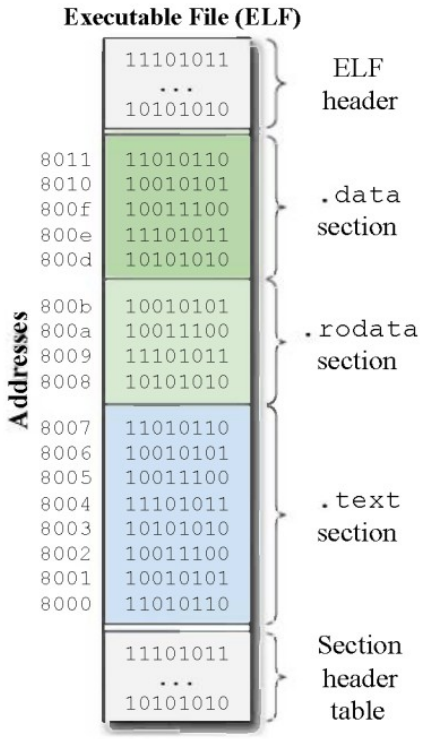
Executable and object files, and assembly programs are usually organized in **sections**. A section may contain **data** or **instructions**, and the contents of each section are mapped to a **set of consecutive main memory addresses**.
The following sections are often present on executable files generated for Linux-based systems:

- **`.text`** : a section dedicated to store the **program instructions**;
- **`.data`** : a section dedicated to store **initialized global variables**, i.e., the variables that need their value to be initialized before the program starts executing;
- **`.bss`** : a section dedicated to store **uninitialized global variables**;
- **`.rodata`** : a section dedicated to **store constants**, i.e., values that are read by the program but not modified during execution.

When linking multiple object files, the linker groups information from sections with the same name and places them together into a single section on the executable file. For example, when linking multiple object files, the contents of the .text sections from all object files are grouped together and placed sequentially on the executable file on a single section that is also called .text .

The following figure shows the layout of an executable file that was generated by the **`riscv64-unknown-elf-ld`** tool, and is encoded using the Executable and Linking Format.
This file contains three sections: the **`.data`** , the **`.rodata`** , and the **`.text`** sections. The contents of section **`.text`** are mapped to addresses **`8000`** to **`8007`** , while the contents of section .data are mapped to addresses **`800d`** to **`8011`** .

By default, the GNU assembler tool adds all the information to the **`.text`** section.



2.1.4 Executable vs object files

The Executable and Linking Format, or ELF, is used by several Linux-based operating systems to encode both object and executable files. Even though object and executable files may contain machine code and both can be encoded using the ELF, they differ in the following aspects:

- Addresses on object files are not final and elements from different sections may be assigned the same addresses. As a consequence, the elements of different sections may not reside in the main memory at the same time;
- Object files usually contain several references to undefined symbols, which are expected to be resolved by the linker;
- Object files contain a relocation table so that instructions and data on object files can be relocated on linking. Addresses on executable files are usually final;
- Object files do not have an entry point;

2.2 Assembly language (formal aspects)

Assembly programs are encoded as plain text files and contain four main elements:

- **Comments** : comments are textual notes that are often used to document information on the code, however, they have no effect on the code generation and the assembler discards them;
- **Labels** : labels are “markers” that represent program locations. They are usually defined by a name ended with the suffix “**:**” and can be inserted into an assembly program to “mark” a program position so that it can be referred to by assembly instructions or other assembly commands, such as assembly directives;
- **Assembly instructions** : Assembly instructions are instructions that are converted by the assembler into machine instructions. They are usually encoded as a string that contains a mnemonic and a sequence of parameters, known as operands. For example, the “ **`addi a0,a1,1`** ” string contains the **`addi`** mnemonic and three operands: **`a0,a1,`** and **`1 ;`**
- **Assembly directives** : Assembly directives are commands used to coordinate the assembling process. They are interpreted by the assembler. For example, the **`.word 10`** directive instructs the assembler to assemble a 32-bit value (**`10`**) into the program. Assembly directives are usually encoded as strings that contains the directive name, which have a **dot** (‘**.**’) **prefix**, and its arguments.

As discussed before, comments have no effect on the assembling process and are discarded by the assembler. This is usually performed by a pre-processor, which removes all comments and extra white spaces.
Once comments and extra white spaces are discarded, the assembly program contains only three kinds of elements: labels, assembly instructions and assembly directives. Assuming <label> , <instruction> , and the <directive> represent valid labels, assembly instructions, and assembly directives, respectively, the following regular expression can be used to summarize the syntax of the assembly language once its comments and extra white spaces are removed.

```
PROGRAM     -> LINES
LINES       -> LINE ['\n' LINES]
LINE        -> [<label>] [<instruction>] |
               [<label>] [<directive>]
```

The first **two rules** of the previous regular expression indicate that an assembly program is composed by one or more lines, which are delimited by the end of line character, i.e., ‘ **`\ n`** ’.
The last rule implies that:

- a line may be empty. Notice that the <label> , the <instruction> , and the <directive> elements are optional 1
- a line may contain a single label;
- a line may contain a label followed by an assembly instruction;
- a line may contain a single assembly instruction;
- a line may contain a label followed by an assembly directive;
- a line may contain a single assembly directive;

2.3 System Calls

User programs usually need to perform **input and output operations**. For example, after performing some computation, the program may need to print the result on the screen, or write it to a file.
Input and output operations are usually performed by input and output devices, such as keyboards, pointing devices, printers, display, network devices, hard drives, etc..

These devices are **managed by the operating system**, hence, input and output operations are also performed by the operating system. To increase portability and facilitate software development, operating systems usually implement a set of abstractions (such and folders, files, etc.) and offer a set of service routines to allow user programs to perform input and output operations. In this context, user programs perform input and output operations by invoking the operating system service routines.
A system call, or **syscall**, is a special call operation used to invoke operating system service routines.

A **system call**, or **syscall**, is a special call operation used to invoke operating system service routines .
In **RISC-V**, this operation is performed by a special instruction called **`ecall`** .

As an example, let us assume the user wants to invoke the Linux operating system service routine "**`write`**" (also known as **write syscall**) to display some information on the screen.
The **`write`** syscall takes three parameters: the **file descriptor**, the **address of a buffer** that contains the information that must be written, and the **number of bytes** that must be written.
The file descriptor is an integer value that **identifies a file or a device**. In this case, it indicates the file or device where the information must be written to.
In many Linux distributions, the **file descriptor** '**`1`**' (one) is used to represent the **standard output**, or **`stdout`** , which is usually the **terminal screen**.

The following code shows an example in which the contents of the msg buffer is written to the file descriptor '1', hence, the screen.
First, the code sets the **system call parameters** (lines 6 to 8), then it **sets register `a7`** with a **number that indicates the service routine** that must be invoked, which, in this case, is the **`write`** **syscall** (line 9).
Finally, it invokes the operating system by executing the **`ecall`** instruction.

```
.data
msg: .asciz "SmartComputerLab - 10 years ahead\n"       # Allocates a string on memory

.text
.globl  main

main:
        li      a0,1            # a0: File descriptor = 1 (stdout)
        la      a1,msg          # a1: Message buffer address
        li      a2,34           # a2: Message buffer size (14 bytes)
        li      a7,64           # Syscall code (write = 64)
        ecall                   # Invoke the syscall
        li      a0,0
        li      a7,93           # Syscall code (exit = 93)
        ecall
```

2.4 Detecting overflow

The **RISC-V does not provide special hardware support to detect overflow** when performing arithmetic operations. However, regular conditional branch and conditional set instructions may be used to find out whether or not an overflow occurred.
The following code shows how to detect whether or not an overflow occurred when adding two unsigned integers. In this case, the **`bltu`** instruction jumps to label **`handle_ov`** in case there is an overflow.

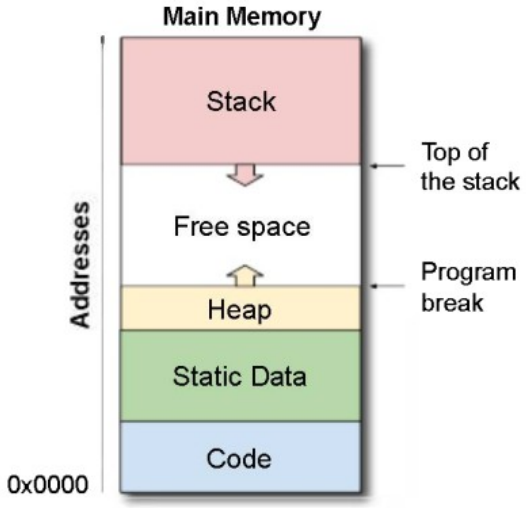
```
add   a0, a1, a2
# Add two values
bltu  a0, a1, handle_ov       # Jumps to handle_ov in case
# an overflow happened
...
handle_ov:
# Code to handle the overflow
```

2.5 Implementing routines

2.5.1 The program memory layout

A **stored-program computer architecture** is a computer architecture that stores both data and code on memory. A Von Neumann architecture is a computer architecture that stores both the data and the code at the same address space. Most modern computer architectures are Von Neumann architectures.

The following figure shows a common way of organizing programs on the memory of Von Neumann architectures.
The **code space** (**`text`**) is a memory space that stores the **program code and is usually placed first**, in the **lowest addresses**. The **static data space** is a memory space that stores the **program static data** (e.g., **global variables**) and **is placed after the code**.
The **heap space** is a memory space **managed** by the **memory allocation library** and is **allocated** after the static data. The heap starts small and, whenever the memory allocation library needs more space, it **invokes the operating system** to grow the heap area, which does it by increasing the "**program break**" **address.**
Finally, the **stack space** is a memory space that stores the program stack and is usually **placed at the end (high addresses) of the memory**.



2.5.2 The program stack

An **active routine** is a routine that was invoked but **did not return yet** . Notice that there may be multiple active routines at a given point of the execution.
For example, in the following code, **routine `fun`** is invoked by **routine `bar`** , which is invoked by **routine `main`**.
Initially, the main routine is active. Then, it invokes the bar routine, which also becomes active. Finally, the fun routine is invoked and it also becomes active. At this point, **there are three active routines** in the system.

```
int a = 10;

int main()
{
        return bar() + 2;
}

int bar()
{
        return fun() + 4;
}

int fun()
{
        return a;
}
```

The **set of active routines increases** whenever a routine is invoked and decreases whenever a routine returns. Routines are activated and deactivated in a **last-in-first-out fashion**, i.e., the last one to be activated must be the first one to be deactivated.

Consequently, the most natural data structure to keep track of active routines is a stack .
Routines usually need memory space to store important information, such as **local variables**, **parameters**, and the **return address**s. Hence, whenever a routine is invoked (and becomes active), the system needs to allocate memory space to store information related to the routine. Once it returns (is deactivated), all the information associated with the routine invocation is not needed anymore and this memory space must be freed.

The **program stack** is a stack **data structure that stores information about active routines** , such as local variables, parameters, and the return address.
The program stack is stored in the main memory and, whenever a routine is invoked, the information about the routine is **pushed on top of the stack**, which causes it to grow. Also, whenever a routine returns, the information about the routine is discarded by dropping the contents at the top of the stack, which causes it to shrink.

The program stack is allocated at the stack space, which is usually placed at the end (high addresses) of the memory. As a consequence, the **program stack must grow towards low addresses**.

The **stack pointer** is a pointer to the **top of the stack** , i.e., it stores the address of the top of the stack. Growing or shrinking the stack is performed by adjusting the stack pointer.

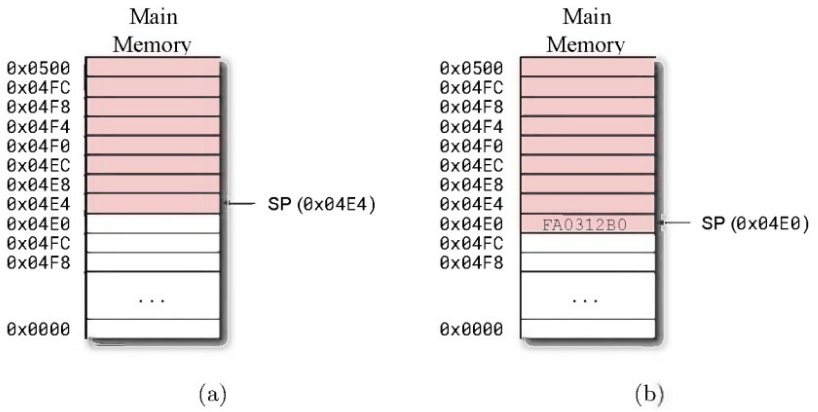
In **RISC-V**, the stack pointer is stored by register **`sp`** . Also, in RISC-V, the stack grows towards low addresses, hence, growing (or allocating space on) the stack can be performed by decreasing the value of register **`sp`** (the stack pointer).
The following code shows how to push the contents of register **`a0`** into the stack. **First, the stack pointer is decreased to allocate space** (4 bytes), then, the contents of register **`a0`** (4 bytes) are **stored on the top** of the program stack using the **`sw`** instruction.

```
addi  sp, sp, -4        # allocate stack space
sw    a0, 0(sp)         # store data into the stack
```

Alternatively, shrinking the stack can be performed by increasing the value of register **`sp`** . The following code shows how to pop a value from the top of the stack into register **`a0`** . First, the value on the top of the program stack is loaded into register **`a0`** (4 bytes) using the **`lw`** instruction.
Then, the **stack pointer is increased to deallocate the space** (4 bytes).

```
lw    a0, 0(sp)         # retrieve data from stack
addi  sp, sp, 4         # deallocate space
```

The following figure (**a**) illustrates a program stack that starts at address **`0x0500`** and grows down to address **`0x04E4`** . Since the stack pointer points to the top of the stack, the contents of the register **`sp`** is equal to **`0x04E4`** .
Figure (b) shows how the program stack is modified after executing the following code, i.e., after pushing the contents of register **`a0`** into the stack. Notice that the value of the stack pointer (**`sp`**) was **decremented by 4** units and the contents of register **`a0`** (**`0xFA0312B0`**) was stored at the memory starting at address **`0x04E0`** .



```
li    a0, 0xFA0312B0
addi sp, sp, -4          # allocate stack space
sw    a0, 0(sp)          # store data into the stack
```

The previous examples discussed how to push and pop a single word (4-byte value) to and from the stack. In many situations, a program may need to push or pop multiple values to or from the stack. For example, the program may need to save a set of register values on the stack. In these cases, the code may be optimized by adjusting (increasing or decreasing) the stack pointer only once.
The following code shows how to push four values from registers a0 , a1 , a2 , and a3 into the program stack. Notice that the stack pointer was adjusted only once and the immediate field of the store word instruction (sw) was used to select the proper position to store each one of the values. In this example, the last value pushed into the stack was the value stored in the register a3 .

```
addi  sp, sp, -16        # allocate stack space (4 words)
sw    a0, 12(sp)         # store the first value (SP+12)
sw    a1, 8(sp)          # store the second value (SP+8)
sw    a2, 4(sp)          # store the third value (SP+4)
sw    a3, 0(sp)          # store the fourth value (SP+0)
```

Attention

Note that for an RV64 processor the 4-byte data values should replaced by 8-byte values (64-bit)

The following code shows how to pop four values from the program stack into registers **a3,a2,a1**, and **a0** . Notice that the stack pointer was adjusted only once and the immediate field of the load word instruction (**lw**) was used to select the proper position to load each one of the values. In this example, the first value popped from the stack was stored into register **a3** .

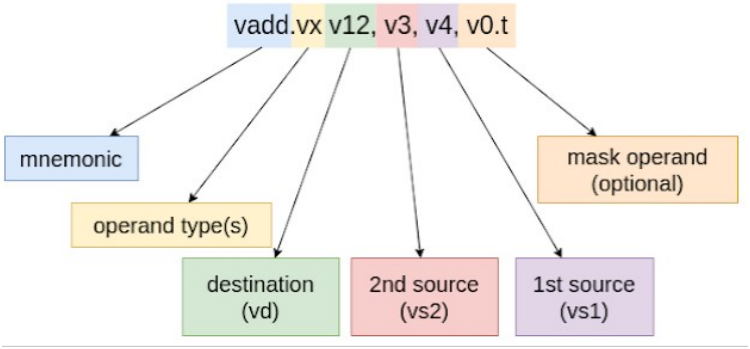
```
Lw    a3, 0(sp)          #retrieve the first value (SP+0)
lw    a2, 4(sp)
lw    a1, 8(sp)
lw    a0, 12(sp)
addi  sp, sp, 16         #deallocate stack space (4 words)
```

2.6 Vector extensions

It is difficult to interpret a **RVV assembly instruction**. Let's review the main component of such instructions and study various examples.
We will draw some generalities from a practical example: a masked integer vector-scalar addition:

```
vadd.vx v12, v3, v4, v0.t
```

The following diagram illustrates the **6 main components** of any RVV assembly instruction. Most of those components have numerous variants and some of them are optional.



2.6.1 Mnemonic

The first component is the **mnemonic** which describes **which operation should be performed** by the instruction (e.g. in our case **`vadd`** will perform a **vector add** while **`vcompress`** will compress a **bitmask**).
The mnemonic often describes the **destination type**: for example **`vadd`** is a vector add with a **single-width destination**, while **`vwadd`** is a **widening vector add** and **`vmadc`** is a vector addition with carry returning a mask of output carries.

2.6.2 Operand type(s)

The second component is the **operand type(s)**. It describes on what type of operand(s) the operation is performed. The most common is **`.vv`** (**vector-vector**) which often means that the instruction admits at least two inputs and **both are single-width vectors**.

The list of various possibilities includes:

- **`.vv`** operation between two (or three) **single-width vectors**, e.g. **`vmul.vv`**
- **`.wv`** operation between two (or three) vectors, **`vs1`** is single-width while **`vs2`** and **`vd`** are wide operands/destinations (EEW=2*SEW), e.g. **`vwmacc.wv`**
- **`.vx/.vf`** operation between one or **multiple vector and a scalar** (general purpose register: **`x`** or floating-point register **`f`**), e.g. **`vfadd.vf`**, the scalar operand is splat to build a vector operand
- **`.wx/.wf`**: operation between a scalar and a wide second vector operand, e.g. **`vfwadd.wf`**
- **`.vi`** operation between one or multiple vectors and an immediate. The immediate is often 5-bit wide encoded in the **`rs1/vs1`** field, **e.g. `vsub.vi`**
- **`.vs`**: operation between a **vector and a single element** (scalar) contained in a vector register, e.g. **`vredsum.vs`** (the single scalar is used to carry the **reduction accumulator**)
- **`.vm`**: operation between a vector and a mask, e.g. **`vcompress.vm`**
- **`.v`**: operation with a single vector input, e.g. **`vmv2r.v`**, may also be used for vector loads (which have scalar operands for the address on top of a single data vector operand: **`vle16.v`**.
- **`.vvm/.vxm/.vim`** operation **vector-vector** / **vector-scalar** / **vector-immediate** with a mask operand (e.g. **`vadc.vvm`**, addition between two vectors with an extra mask operand constituting an input carry vector)

The conversions constitutes a category of its own for the operand types, because the mnemonic suffix describes: the destination format, the source format, and the type of operand.
For example **`vfcvt.x.f.v`** is a vector (**`.v`**) conversion from floating-point element (**`.f`**) to signed integer (**`.x`**) result elements. **`.xu`** is used to indicate unsigned integers, **`.rtz`** is used to indicate a static round-towards-zero rounding mode.

2.6.3 Destination and source(s)

In the assembly instruction, destination and sources follows the mnemonic. The **destination is the first register to appears**, followed by **one or multiple sources**.

Each of those element encodes a **register group**. The destination and source operands register groups a**re represented by the first register in the group.**
For example if **`LMUL=4`**, then **`v12` represents the 4-wide register group `v12v13v14v15`)**. Thus the actual register group depends on the assembly opcode but also on the value of **`vtype`:** it is **context sensitive**.
Most **RVV** operations have a **vector destination**, denoted by **`vd`**, some may have a **scalar destination** (e.g. **`vmv.x.s`** with a **`x`** register destination or **`vfmv.f.s`** with a **`f`** register destination) and others have a **memory destination** such as the **vector stores**, e.g. **`vse32.v`**.

There can be **one or two sources**: **`vs2`** and **`vs1`** for vector-vector instructions. If the operations admits a **scalar operand**, or an **immediate operand** then **`vs1`** is replaced by **`rs1`** (respectively ***`imm`***),
e.g. **`vfadd.vf v4, v2, ft3`**.

Vector loads have a memory source, e.g. **`vloxei8.v vd,(rs1),vs2[,vm]`** which has a **scalar register as address source** and a **vector register as destination source**.

RVV defines 3-operand instruction, e.g. **`vmacc.vv`**. For those operations the destination register **`vd`** is both a source and a destination: the operation is destructive: one of the source operand is going to be overwritten by the result.

2.6.4 Mask operand

Most **RVV operation can be masked**: in such case the **`v0` register is used as a mask** to determine **which elements are active** and whose elements of the result will be copied from the old destination value or filled with a pre-determined pattern. **RVV 1.0** only supports **true bit as active masks**: the element is considered active if the bit at the **corresponding index is set to `1`** in **`v0`**, and **inactive** if it is **`0`**.
This is what is encoded by the last operand of our example: **`v0.t`** (for **`v0`** "**true**").
If this last operand i**s missing**, then the operation is unmasked (**all body elements are considered active**).

Compiling the assembly code with vector instructions on our board (128-bit vector register: 8*8 or 16*8 – 8 operations in parallel max.):

```
gcc –o memchr memchr.s –march=rv64gcv
```

Two modules: test_func.s and func.s

```
        .file   "test_func.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        li      a0,10
        call    func@plt
        mv      a5,a0
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        addiw   a5,a5,1
        sext.w  a5,a5
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
~

        .file   "func.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  func
        .type   func, @function
func:
        addi    sp,sp,-32
        sd      s0,24(sp)
        addi    s0,sp,32
        mv      a5,a0
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a4,a5
        mv      a5,a4
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        sext.w  a5,a5
        mv      a0,a5
        ld      s0,24(sp)
        addi    sp,sp,32
        jr      ra
        .size   func, .-func
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
~
```

Simplified version (no stack):

```
.LC1:
        .string  "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
        li a0,10
        call funcs@plt
        lla  a0,.LC1
        call printf@plt
        li   a0,0
        addi  a7,x0,93
        ecall



.globl  funcs
.type   funcs,@function

.LC0:
        .string  "a=%d\n"

funcs:
        li a4,100
        mv a5,a0
        mulw a5,a5,a4
        mv  a1,a5
        ret
```

To compile and execute:

```
ubuntu@ubuntu:~/assembly$ gcc -o test_funcs test_funcs.s funcs.s
ubuntu@ubuntu:~/assembly$ ./test_funcs
1000
```

To do:

Complete the above code with **`scanf`** – assembly function:

```
.LC0:
        .string "%d"

        mv      a1,a5                   # argument-address for scanf
        lla     a0,.LC0                 # format string for scanf
        call    scanf@plt
```

Solution:

```
.LC0:
        .string  "%d\n"
        .text
        .globl  main
        .type   main, @function
.LC1:
        .string  "%d"
        .align  3
main:
        addi    a5,s0,-8            # preparing pointer address
        mv      a1,a5               # moving address to a1 argument
        lla     a0,.LC1             # loading scanf argument string
        call    scanf@plt           # calling scanf linux function
        lw      a5,-8(s0)           # loading the result from the indicated address
        mv      a0,a5               # moving the value to a0 register
        call    funcs@plt           # calling the function to calculate the result
        lla     a0,.LC0             # the result is passed in register a1
        call    printf@plt          # calling printf linux function
        li      a0,0                # loading exit value
        addi    a7,x0,93            # preparing the linux function number
        ecall                       # calling the linux function
```

2.2 RISC-V assembly - `Hello counter`

```
        .file   "helloc.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "counter=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        li      a5,5
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt           # linux call with function name
        li      a5,0
        mv      a0,a5
        addi    a7,x0,93             # linux call with function number
        ecall
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Reading and printing values

```
        .file   "myio.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "%d"
        .align  3
.LC1:
        .string "a=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        addi    s0,sp,32
        li      a4,0
        li      a5,4
        sw      a5,-28(s0)
        addi    a5,s0,-32               # here scanf puts the value: s0-32
        mv      a1,a5                   # argument-address for scanf
        lla     a0,.LC0                 # format string for scanf
        call    scanf@plt
        lw      a5,-32(s0)              # loading the received value
        lw      a4,-28(s0)              # loading the constant-program value - 4
        addw    a5,a4,a5
        sext.w  a5,a5
        mv      a1,a5                   # here is the value to be printed
        lla     a0,.LC1                 # format for printf
        call    printf@plt
        li      a0,0
        addi    a7,x0,93                # linux exit call number
        ecall
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Multiplication by 10 and 100

```
        .file   "mult10.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "a=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sw      zero,-24(s0)
        li      a5,4
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a4,a5
        mv      a5,a4
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        sw      a5,-24(s0)
        lw      a5,-24(s0)
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a5,0
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits


        .file   "mult100.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "a=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sw      zero,-24(s0)
        li      a5,4
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a4,a5
        li      a5,100
        mulw    a5,a4,a5
        sw      a5,-24(s0)
        lw      a5,-24(s0)
        mv      a1,a5
```

```
        lla     a0,.LC0
        call    printf@plt
        li      a5,0
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
~
```

Floating point division

```
        .file   "div100.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC2:
        .string "a=%f\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        lla     a5,.LC0
        flw     fa5,0(a5)
        fsw     fa5,-24(s0)
        sw      zero,-20(s0)
        flw     fa4,-24(s0)
        lla     a5,.LC1
        flw     fa5,0(a5)
        fdiv.s  fa5,fa4,fa5
        fsw     fa5,-20(s0)
        flw     fa5,-20(s0)
        fcvt.d.s        fa5,fa5
        fmv.x.d a1,fa5
        lla     a0,.LC2
        call    printf@plt
        li      a5,0
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .section        .rodata
        .align  2
.LC0:
        .word   1137180672
        .align  2
.LC1:
        .word   1133903872
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
~
```

2.3 RISC-V assembly - `stack layout`

Let us take the following example:

```
int calculate(int x, int y) {  // a0, a1
```

```
   int tmp = x * 10;
   return tmp + y;
}

int main() {
   int alpha = 4;
   int beta = calculate(2, 3);
   return alpha + beta;
}

        .file   "calc1.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  calculate
        .type   calculate, @function
calculate:
        addi    sp,sp,-48
        sd      s0,40(sp)
        addi    s0,sp,48
        mv      a5,a0
        mv      a4,a1
        sw      a5,-36(s0)
        mv      a5,a4
        sw      a5,-40(s0)
        lw      a5,-36(s0)
        mv      a4,a5
        mv      a5,a4
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a4,a5
        lw      a5,-40(s0)
        addw    a5,a4,a5
        sext.w  a5,a5
        mv      a0,a5
        ld      s0,40(sp)
        addi    sp,sp,48
        jr      ra
        .size   calculate, .-calculate
        .align  1
        .globl  main
        .type   main, @function


main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        li      a5,4
        sw      a5,-24(s0)
        li      a1,3
        li      a0,2
        call    calculate
        mv      a5,a0
        sw      a5,-20(s0)
        lw      a5,-24(s0)
        mv      a4,a5
        lw      a5,-20(s0)
        addw    a5,a4,a5
        sext.w  a5,a5
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

The first few parameters, type permitting, are passed in registers so they don't even appear on the stack. Other than that, it's unclear what you really want to know. If you do get some arguments that are on the stack, they stay there even after you adjust the stack pointer so you can still address them relative to the **adjusted stack pointer** or a **frame pointer** (here **$s0** apparently).

The important steps, that need clarification are:

```
addi    sp,sp,-32       # allocate space on stack
sd      ra,24(sp)       # save $ra (value of ra)
sd      s0,16(sp)       # save $s0 (vaue of s0)
addi    s0,sp,32        # set up s0 as frame pointer
li      a5,4            # load imm 4 to a5
sw      a5,-24(s0)      # save a5 on stack at frame address 32-24=8
li      a1,3            # load imm 3 to a1
li      a0,2            # load imm 2 to a0
call    calculate
```

Calculating power

In this example we study loop construct for power calculation function.
This is C code for this function. Note the use of separate segment code for printing the result.

```
#include <stdio.h>
/* Iterative Function to calculate (x^y) in O(logy) */
int power(int x, unsigned int y)
{
    int res = 1; // Initialize result
    while (y > 0) {
        // If y is odd, multiply x with result
        if (y & 1)
            res = res * x;
        // n must be even now
        y = y >> 1; // y = y/2
        x = x * x; // Change x to x^2
    }
    return res;
}

int main()
{
        int pow=0;
        pow=power(2,8);
        printf("2^8=%d\n",pow);
}
```

The initial values to calculate **3^5** are registered in **.data** section as **x: .word** and **y: .word.** Additionally we prepare to character strings **str1** and **str2** for the result **block**.

The **.text** section (code) starts with **main:**
After the preparation of the arguments in **s0,s1** and **a0,a1** we call the **power:** block for calculation phase.

In the **power:** block we prepare 3 places on the stack (3*4) for the return address (**ra**) and 2 arguments (**s0,s1**).

The initial result – **1**, is prepared in register **s2**.

In the loop: block we start with the compare to zero instruction, that is branch instruction to Ret: block if the tested argument is equal

```
.data
x: .word 3
y: .word 5                              # unsigned
str1: .string " ^ "
str2: .string " = "

.text
main:
        lw s0, x                        # Set s0 equal to the parameter x
        lw s1, y                        # Set s1 equal to the parameter y
        mv a0, s0                       # Set arguments
        mv a1, s1
        jal power                       # return the result a0
        mv s2, a0                       # Set s2 equal to the result
        j print
power:
        addi sp, sp, -12
        sw ra, 0(sp)
        sw s0, 4(sp)
        sw s1, 8(sp)
        mv s0, a0                       # s0 -> parameter x
        mv s1, a1                       # s1 -> parameter y
        li s2, 1                        # s2 -> res, init = 1
loop:
        bge x0, s1, Ret         # if (0 >= y) break
        andi t0, s1, 0x1        # t0 -> y & 0x1
        li t1, 0x1                      # t1 -> 0x1
        li ra, 0x54                     # Set ra equal to srli s1, s1, 1
        beq t0, t1, odd         # y is odd
        srli s1, s1, 1          # y_u >> 1, y /= 2
        mul s0, s0, s0          # x = x * x
        j loop                          # goto loop
odd:
        mul s2, s2, s0          # res = res * x
        jr ra                           # goto srli s1, s1, 1
Ret:
        mv a0, s2                       # Set return reg a0
        lw ra, 0(sp)            # Restore ra
        lw s0, 4(sp)            # Restore s0
        lw s1, 8(sp)            # Restore s1
        addi sp, sp, 12         # Free space on the stack for the 3 words
        jr ra                           # Return to the caller
```

Prepared print function:

```
print:
        mv t0, a0                       # Set tmp equal to a0
        mv a0, s0                       # prepare to print x
        li a7, 1                        # print int
        ecall
```

```
        la a0, str1                     # prepare to print string ^
        li a7, 4                        # print string
        ecall
        mv a0, s1                       # prepare to print y
        li a7, 1                        # print int
        ecall
        la a0, str2                     # prepare to print string =
        li a7, 4                        # print string
        ecall
        mv a0, s2                       # prepare to print result
        li a7, 1                        # print int
```

Second example of power calculation

```
        .file   "pow.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  power
        .type   power, @function
power:
        addi    sp,sp,-48           # prepare 48 bytes on stack for calculation
        sd      s0,40(sp)           # store double s0 on sp+40
        addi    s0,sp,48            # prepare frame s0 with initial sp value
        mv      a5,a0               # move a0 value to a5 - first argument : x
        mv      a4,a1               # move a1 value to a4 - second argument: y
        sw      a5,-36(s0)          # store word a5 on stack at s0-36
        mv      a5,a4               # move a4 to a5
        sw      a5,-40(s0)          # store word a5 on stack at s0-40
        li      a5,1                # move 1 to a5 : initial calculation
        sw      a5,-20(s0)          # store word a5 on stack at s0-20
        j       .L2                 # jump to .L2 block
.L4:
        lw      a5,-40(s0)          # load word - second argument : y
        andi    a5,a5,1             # and imm 1 with a5 (demask the lowest bit)
        sext.w  a5,a5               # add sign extension to a5
        beq     a5,zero,.L3         # if the lowest bit is 0 branch to .L3 block (no multi)
        lw      a5,-20(s0)          # load word - intermediate result: x to a5
        mv      a4,a5               # move a5 to a4
        lw      a5,-36(s0)          # load first argument: x from stack
        mulw    a5,a4,a5            # multiply word by x by intermediate result
        sw      a5,-20(s0)          # save new intermediate result
.L3:
        lw      a5,-40(s0)          # load second argument to a5 from stack
        srliw   a5,a5,1             # shift right imm word 1 bit (new second argument)
        sw      a5,-40(s0)          # save new second argument
        lw      a5,-36(s0)          # load first argument : x
        mulw    a5,a5,a5            # multiply a5*a5 and register in a5
        sw      a5,-36(s0)          # store new product on stack
.L2:
        lw      a5,-40(s0)          # load word from stack at s0-40 to a5: first argument
        sext.w  a5,a5               # add sign extension to a5
        bne     a5,zero,.L4         # if a5 not equal to zero branch to .L4 block
        lw      a5,-20(s0)          # load word to a5 from s0-20 (1)
```

```
        mv      a0,a5                   # move a5 to a0
        ld      s0,40(sp)               # load s0 with sp+40 (address)
        addi    sp,sp,48                # return address from .power block
        jr      ra                      # jump to return address
        .size   power, .-power
        .section        .rodata
        .align  3
.LC0:
        .string "2^8=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32               # preparing stack pointer for 8 words (-32)
        sd      ra,24(sp)               # save double ra on stack at -8
        sd      s0,16(sp)               # save double s0 on stack at -16
        addi    s0,sp,32                # add integer imm to s0=sp+32 - initial value
        sw      zero,-20(s0)            # put zero to stack at -20 , rel initial value
        li      a1,8                    # load to a1 imm 8
        li      a0,2                    # load to a0 imm 2
        call    power                   # call power block
        mv      a5,a0                   # return from power block : move a0 to a5
        sw      a5,-20(s0)              # save a5 at stack at -20
        lw      a5,-20(s0)              # load a5 from stack at -20
        mv      a1,a5                   # move a5 to a1
        lla     a0,.LC0                 # load address of .LC0 block to a0
        call    printf@plt              # call linux print function with .LC0 string
        li      a5,0                    # load zerotoa5
        mv      a0,a5                   # move a5 value to a0
        ld      ra,24(sp)                       # restore (load) return address from stack
        ld      s0,16(sp)                       # load s0 with sp+16
        addi    sp,sp,32                        # add imm 32 to sp to return to initial sp value
        jr      ra                              # return to call address from linux
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Simplified power function without stack

```
        .file   "pow.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  power
        .type   power, @function
power:
        li      a4,2                    # first argument - base
        li      a5,8                    # second argument - exponent
        li      a6,1                    # initial result base^0
        j       .L2                     # jump to starting block - L4
.L4:
        mv      a7,a5                   # move a5 to a7 for mask test
        andi    a7,a7,1                 # mask test 0 or 1
        beq     a7,zero,.L3             # if zero go to L3
        mulw    a6,a6,a4                # else multiply the base with partial result
.L3:
        mulw    a4,a4,a4                # multiply the base
        srli    a5,a5,1                 # shift right the exponent
.L2:
        bne     a5,zero,.L4             # test the value of shift exponent, ifnon zero go to L4
        jr      ra                      # else return to main block
        .size   power, .-power
        .section        .rodata
        .align  3
.LC0:
        .string "2^8=%d\n"
        .text
        .align  1
        .globl  main
```

```
        .type   main, @function
main:
        li      a4,2                # prepare base value
        li      a5,8                # prepare exponent value
        call    power               # call power block
        mv      a1,a6               # move result to a1
        lla     a0,.LC0             # load string address to a0
        call    printf@plt          # call linux printf
        li      a0,0                # load 0 to a0
        addi    a7,x0,93            # load linux function number - exit
        ecall                       # call linux
```

To do:

Extend the above code by providing the base and exponents values from terminal.

Processing interruptions (signals)

```
#include  <signal.h>
#include  <stdio.h>
#include  <unistd.h>

void mysig(int sig)
{
printf("SIGINT\n");
}

int main()
{
        signal(SIGINT,mysig);
        while(1)
        {
                sleep(3);
        }
}
```

```
        .file   "int_sig2.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "SIGINT"
        .text
        .align  1
        .globl  mysig
        .type   mysig, @function
mysig:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        mv      a5,a0
        sw      a5,-20(s0)
        lla     a0,.LC0
        call    puts@plt
        nop
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   mysig, .-mysig
        .align  1
        .globl  main
```

```
        .type   main, @function
main:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        addi    s0,sp,16
        lla     a1,mysig
        li      a0,2
        call    signal@plt
.L3:
        li      a0,3
        call    sleep@plt
        j       .L3
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

RV – 5 stages architecture:

https://hackmd.io/@shauming1020/riscv-home1