# Assembler-Processor Simulator

The program simulates the tasks of an assembler and a processor.

The program expects two arguments (in the mentioned order) –

1. An assembly file with '.asm' extension, which contains assembly code following TOYRISC architecture.
2. A file with '.out' extension, which is used by the program to store instructions in their binary codes.

The processor is a pipelined model, which consists of five stages – Instruction Fetch, Operand Fetch, Execute, Memory Access, Register Write, in which stages are stalled in case of a data hazard or a control hazard.

# TOYRISC Specification

## Memory Model

The memory space is of 256KB. Each word is 4 bytes long, and the memory is word-addressable. That is, a total of 65536 words may be stored. These include the program instructions, the static data, and the stack.

## Registers

There are a total of 32 registers: x0 to x31. Each register is 4 bytes wide.

## Registers in TOYRISC ISA

| Register | Purpose |
|---|---|
| x0 | Zero Register |
| x1 | Stack Pointer |
| x2 | Frame Pointer |
| x3 – x5 | Used by the assembler |
| x6-x30 | General purpose |
| x31 | Special behavior, according to particular instruction |
| PC | Program Counter |

# Encoding

32 registers require 5 bits for encoding. x0 is encoded as 00000, x1 as 00001, and so on.

# Instruction Formats in the custom ISA

## R3-type

| opcode | rs1 | rs2 | rd | unused |
|---|---|---|---|---|
| 5 bits | 5 bits | 5 bits | 5 bits | 12 bits |

## R2I-type

| opcode | rs1 | rs2 | immediate |
|---|---|---|---|
| 5 bits | 5 bits | 5 bits | 17 bits |

## RI-type

| opcode | rd | immediate |
|---|---|---|
| 5 bits | 5 bits | 22 bits |

# Arithmetic Instruction in TOYRISC ISA

<table>
<tr><th>Operation</th><th>Opcode</th><th>Format</th><th>Description</th></tr>
<tr><td>add</td><td>00000</td><td>R3-Type</td><td>rd = rs1 + rs2</td></tr>
<tr><td>addi</td><td>00001</td><td>R2I-Type</td><td>rd = rs1 + imm</td></tr>
<tr><td>sub</td><td>00010</td><td>R3-Type</td><td>rd = rs1 - rs2</td></tr>
<tr><td>subi</td><td>00011</td><td>R2I-Type</td><td>rd = rs1 - imm</td></tr>
<tr><td>mul</td><td>00100</td><td>R3-Type</td><td>rd = rs1 * rs2</td></tr>
<tr><td>muli</td><td>00101</td><td>R2I-Type</td><td>rd = rs1 * imm</td></tr>
<tr><td>div</td><td>00110</td><td>R3-Type</td><td>rd = rs1 / rs2</td></tr>
<tr><td>divi</td><td>00111</td><td>R2I-Type</td><td>rd = rs1 / imm</td></tr>
<tr><td>and</td><td>01000</td><td>R3-Type</td><td>rd = rs1 & rs2</td></tr>
<tr><td>andi</td><td>01001</td><td>R2I-Type</td><td>rd = rs1 & imm</td></tr>
<tr><td>or</td><td>01010</td><td>R3-Type</td><td>rd = rs1 | rs2</td></tr>
<tr><td>ori</td><td>01011</td><td>R2I-Type</td><td>rd = rs1 | imm</td></tr>
<tr><td>xor</td><td>01100</td><td>R3-Type</td><td>rd = rs1 (xor) rs2</td></tr>
<tr><td>xori</td><td>01101</td><td>R2I-Type</td><td>rd = rs1 (xor) imm</td></tr>
<tr><td>slt</td><td>01110</td><td>R3-Type</td><td>rd = 1 if rs1<rs2, 0 otherwise</td></tr>
<tr><td>slti</td><td>01111</td><td>R2I-Type</td><td>rd = 1 if rs1<imm, 0 otherwise</td></tr>
</table>

| | | | |
|---|---|---|---|
| sll | 10000 | R3-Type | rd = rs1 logically left shifted by rs2 bits |
| slli | 10001 | R2I-Type | rd = rs1 logically left shifted by imm bits |
| srl | 10010 | R3-Type | rd = rs1 logically right shifted by rs2 bits |
| srli | 10011 | R2I-Type | rd = rs1 logically right shifted by imm bits |
| sra | 10100 | R3-Type | rd = rs1 arithmetically right shifted by rs2 bits |
| srai | 10101 | R2I-Type | rd = rs1 arithmetically right shifted by imm bits |
| Note: If the result is greater than 32 bits, the higher bits (63 to 32) are stored in x31. In case of division operation, the remainder is stored in x31. In case of shift operations, the bits shifted out are stored in x31. | | | |

## Memory Instructions in TOYRISC ISA

| **Operation** | **Opcode** | **Format** | **Description** |
|---|---|---|---|
| load | 10110 | R2I-Type | rd = word at [rs1 + imm] |
| store | 10111 | R2I-Type | word at [rd + imm] = rs1 |
| Note: imm values can be specified as label or absolute value | | | |

## Control Flow Instructions in TOYRISC ISA

| **Operation** | **Opcode** | **Format** | **Description** |
|---|---|---|---|
| jmp | 11000 | RI-Type | PC = PC + rd + imm |
| beq | 11001 | R2I-Type | If rs1 = rd, PC = PC + imm |
| bne | 11010 | R2I-Type | If rs1,rd, PC = PC + imm |
| blt | 11011 | R2I-Type | If rs1<rd, PC = PC + imm |
| bgt | 11100 | R2I-Type | If rs1>rd, PC = PC + imm |

## End Instruction

<table>
<tr><th colspan="4">Assembly Notation</th></tr>
<tr><th>Operation</th><th colspan="3">Description</th></tr>
<tr><td>end</td><td colspan="3">terminate execution</td></tr>
<tr><th colspan="4">Binary Code</th></tr>
<tr><th>Operation</th><th>Opcode</th><th>Format</th><th>Description</th></tr>
<tr><td>end</td><td>11101</td><td>RI-Type</td><td>rd and imm are unused</td></tr>
</table>

## Push Instruction

| Assembly Notation | | | |
|---|---|---|---|
| **Operation** | **Description** | | |
| push | push an argument onto the stack | | |
| **Binary Code** | | | |
| **Operation** | **Opcode** | **Format** | **Description** |
| push | 11110 | RI-Type | rs1 and imm are unused |

## Ret Instruction

| Assembly Notation | | | |
|---|---|---|---|
| **Operation** | **Description** | | |
| ret | return the control flow back to the caller | | |
| **Binary Code** | | | |
| **Operation** | **Opcode** | **Format** | **Description** |
| ret | 11111 | RI-Type | rs1, imm and rd are unused |

## Function Calling

A function is defined by its name, followed by an empty space (‘ ’), with brackets enclosing its arguments. A comma (‘,’) and an empty space (‘ ’) need to be placed in between every two arguments of the function. The body of the function must reside between opening (‘{’) and closing (‘}’) braces, with each statement of the body other than a label properly indented with a tab (‘\t’).

An example is shown below –

Name of the function – ‘add’

Arguments of the function – ‘number1’, ‘number2’

Purpose – Adds the two numbers and stores it in x12

```
add (#number1, #number2){
	load %x0, #number1, %x10	// Retrieving the value of the first argument from the RAM and
					storing it in x10
	load %x0, #number2, %x11	// Retrieving the value of the second argument from the RAM
					storing it in x11
```

```
    add %x12, %x10, %x11          // Storing the sum of the two numbers in x12
    ret                           // Returns the control flow back to the caller
}
```

The instruction 'ret' terminates the function and returns the control flow back to the caller. Every function must have at least one 'ret' instruction.

Arguments can be passed using the 'push' instruction, which needs to be done before calling the function. For example, the instruction - 'push %x6' is interpreted as the instruction to push the value stored in 'x6' onto the stack.

A function can be called using the 'jmp' instruction. For example, the instruction - 'jmp add' is interpreted as a call to the function 'add'.

Parameters can be retrieved in the body of a function using the 'load' instruction, as shown in the first two instructions of the above example - To retrieve the value stored in the memory locations [%x2 + imm], where '%x2' is the frame pointer.

## Example 1 – A program which adds two numbers

```
.data
a:
    123
    234
.text
main:
    load %x0, $a, %x4
    addi %x0, 1, %x3
    load %x3, $a, %x5
    add %x4, %x5, %x6
end
```

- '.data' is a directive used to signify the beginning of the global data segment.
- 'a' and 'main' are descriptive names for memory addresses. Here 'a' refers to memory address 0, main refers to memory address 2. These are called 'labels', and are not essential – their only purpose is to make writing, understanding and reasoning about assembly programs easier.

- Global data are simply listed one after the other (after the .data directive). Value 123 is stored at memory address 0, value 234 at address 1.
- '.text' is a directive used to signify the beginning of the text or the code segment.
- 'main' is a special name. It indicates where the execution will commence from (program counter will be set to this value when the program is loaded).
- Destination operands are always written last. load %x0, $a, %x4 denotes a load operation that writes the read value to register x4.
- In instructions, named addresses are prefixed by a "$". load $a denotes a load operation that reads from memory address 0 (recall that a refers to address 0).
- Registers are prefixed by a '%'. load %x0, $a, %x4 denotes a load operation that writes the read value to register x4.
- Immediate values are written simply.
- 'end' is a special instruction type used to denote the end of the program.

## Example 2 – A program which computes the factorial of a given number

```
    .data
a:
    10
    .text
main:
    addi %x0, 7, %x5
    addi %x0, 1, %x20
    push %x5
    jmp factorial
    end
factorial (#a){
    load %x2, #a, %x6
    bgt %x6, %x0, continue
```

```
    ret
continue:
    mul %x20, %x6, %x20
    subi %x6, 1, %x7
    push %x7
    jmp factorial
    ret
}
```