# CS2160: Computer Organization Laboratory
## *ToyRISC* Specification

# 1 Specification

## 1.1 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 $2^{16}$ words may be stored. These include the program instructions, the static data, and the stack.

## 1.2 Register

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

Table 1: Registers in the custom ISA

| Register | Purpose |
|:---:|:---:|
| x0 | Zero Register |
| x1 | Stack Pointer |
| x2 | Frame Pointer |
| x3 to 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.

## 1.3 Instruction Formats

Table 2 lists the 3 instruction formats in our custom ISA.

### 1.3.1 Arithmetic Instructions

Table 3 lists the different arithmetic instructions.

Table 2: 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 | rd | immediate |
|--------|--------|--------|-----------|
| 5 bits | 5 bits | 5 bits | 17 bits |

**RI-Type**

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

Table 3: Arithmetic instructions in the custom ISA

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

Note: imm values are placed in sourceOperand2 in ParsedProgram

Table 4: Memory instructions in the custom 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

Note: imm values are placed in sourceOperand2 in ParsedProgram

### 1.3.2 Memory Instructions

Table 4 lists the different memory instructions in our custom ISA.

### 1.3.3 Control Flow Instructions

Table 5 lists the different control instructions in our custom ISA.

Control flow instructions are slightly more involved. The assembly notation, and the corresponding binary code have a subtle but important difference.

Table 5: Control Flow instructions in the custom 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 |
| Note: for jmp, while writing the assembly program, we follow the convention that either rd or imm is used. In machine code, the unused one is set to zero. In ParsedProgram, the used one is placed in the destinationOperand field of the Instruction class. | | | |
| Note: in ParsedProgram, for conditional branches, the two registers that are compared are placed in sourceOperand1 and sourceOperand2. The imm value is placed in destinationOperand. | | | |

### 1.3.4 Special Instruction: end

The end instruction is used to indicate the end of the program.
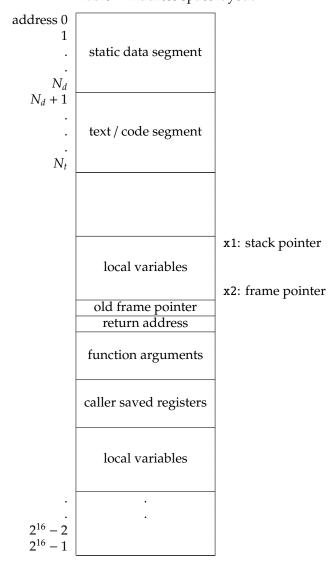
Table 6: End instruction

| Assembly Notation | |
|---|---|
| **Operation** | **Description** |
| end | terminate execution |

| Binary Code | | | |
|---|---|---|---|
| **Operation** | **Opcode** | **Format** | **Description** |
| end | 11101 | RI-Type | rd and imm are unused |

## 1.4 Address Space Layout

Addresses 0 to $N_d$ correspond to the static data. Addresses $N_d$ to $N_t$ correspond to the text segment or the code segment. These lines contain the instructions of the program – $N_t - N_d$ instructions, one instruction per line. The stack grows in the reverse direction – the top of the stack has a lower address than the bottom. The stack begin growing from address $2^{16} - 1$ onwards.

Table 7: Address space layout

| | |
|---|---|
| address 0 | |
| 1 | static data segment |
| . | |
| . | |
| $N_d$ | |
| $N_d + 1$ | |
| . | text / code segment |
| . | |
| . | |
| $N_t$ | |
| | |
| | |
| | local variables |
| | old frame pointer |
| | return address |
| | function arguments |
| | caller saved registers |
| | local variables |
| . | . |
| . | . |
| $2^{16} - 2$ | |
| $2^{16} - 1$ | |

**x1**: stack pointer

**x2**: frame pointer

## 1.5 Function Calling Convention

All function arguments are passed through the stack. Return values are also passed through the stack.

**Caller Behavior**

- The caller function first pushes onto the stack all registers whose values it wishes to preserve for use *after* the function call.

  Pushing a value means decrementing the stack pointer by one, and then performing a store to the address pointed to by the stack pointer. Similarly, popping a value means performing a load from the address pointed to by the stack pointer, and then incrementing the stack pointer by one. Note that the typical behavior is explained – you may optimize the number of additions and subtractions.

- It then pushes all the arguments onto the stack.

- It then pushes the return address (address of the instruction following the jump to the function).

- It sets the stack pointer x1 to point to the top of the stack.

- It then performs the jump.

- Once the called function returns, it finds the return values in the addresses starting from the stack pointer x1 (address smaller than x1).

- It then pops out all the register values it had earlier preserved.

**Callee Behavior**

- The callee first pushes x2 onto the stack.

- It then updates the value of the frame pointer: x2 takes the value of x1 subtracted by 1.

- It then performs its work. To access the arguments, it does so relatively based on the value of the frame pointer x2. As part of its work, it may perform further memory operations in the stack space, but only in addresses strictly lesser than the frame pointer x2.

- Once it is done with its work, it copies x2 to x1.

- It pops out the earlier stored value of x2 into x2.

- It then pushes all the values to be returned onto the stack.

- It then jumps to the return address, which is accessed using the stack pointer x1.

**Note**

Be very meticulous in updating the value of the frame pointer and the stack pointer.

# 2 Example Assembly Programs

## 2.1 Adding Two Numbers

The syntax will be described using the following example program, written in our custom ISA, to add two numbers '123' and '234' and place the result in a certain register location:

```
        .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. They 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.

## 2.2  Linear Search

Consider the following program to search for number in an array a of size n. If found, '1' is written to x10. Else, '-1' is written.

```
    .data
a:
    5
    6
    30
    24
    10
    7
n:
    6
number:
    88
    .text
main:
```

```
    add %x0, %x0, %x3
    load %x0, $n, %x6
    load %x0, $number, %x5
loop:
    load %x3, $a, %x4
    beq %x4, %x5, success
    addi %x3, 1, %x3
    bgt %x3, %x6, endl
    jmp loop
success:
    addi %x0, 1, %x10
    end
endl:
    subi %x0, 1, %x10
    end
```