Lab 1: Introduction to the RISC-V RV32I Instruction set Architecture (ISA)

1 Introduction (mainly for reference)

The RV32I[1] ISA is an open-source specification for the behavior of a processor core. It describes which binary instructions it can read and how it should behave. Any architecture and implementation that adheres to the specification is valid and can call itself RISC-V.

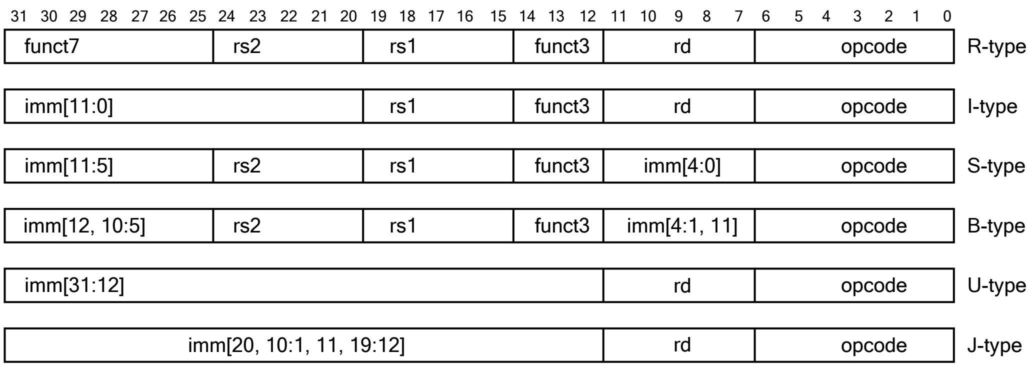
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12, 10:5] | rs2 | rs1 | funct3 | imm[4:1, 11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20, 10:1, 11, 19:12] | rd | opcode | J-type

Figure 1: RV32I Format

Figure 1 details the format of the instructions available in the RV32I instruction set. They are split into categories R(register), I(immediate), S(store), B(branch), U(upper immediate), and J(jump). This lab aims to familiarize you with the 37 basic RV32I instructions [RV32I].

Core to the RISC-V 32-bit architecture are 32 registers of 32 bits *Table 1*, which can be quickly accessed by the core to mutate, move, store, and load data to and from RAM. These registers have specific names and uses for modern compilers, although most are functionally identical.

Reg.	Name	Description
x0	zero	**Description**
x1	ra	return address
x2	sp	Stack pointer
x3	gp	Global Pointer
x4	tp	Thread pointer
x5	t0	Temporaries
x6	t1	~
x7	t2	~
x8	s0	Saved registers
x9	s1	~
x10	a0	Return Value 1
x11	a1	Return Value 2
x12	a2	Function args.
x13	a3	~
x14	a4	~
x15	a5	~
x16	a6	~
x17	a7	~
x18	s2	Saved registers
x19	s3	~
x20	s4	~
x21	s5	~
x22	s6	~
x23	s7	~
x24	s8	~
x25	s9	~
x26	s10	~
x27	s11	~
x28	t3	Temporaries
x29	t4	~
x30	t5	~
x31	t6	~

Table 1: 32 core registers of a RISC-V processor

[1]The name of the ISA (RV32I) is a code which means: RV (RISC-V), 32 (32-bit CPU), I (Integer). There exist multiple ISAs for different functionalities e.g. RV32M (multiplication), RV32A (atomics - bit manipulation). All can be implemented in the same processor, however only RV32I is in the scope of this labs.

1.1 RISC-V assembly

Assembly	Format[2]	Description
ADDI	addi rd, rs1, imm	Adds imm to rs1 and stores result in rd
ANDI	andi rd, rs1, imm	Bitwise AND between rs1 and imm; result in rd
ORI	ori rd, rs1, imm	Bitwise OR between rs1 and imm; result in rd
XORI	xori rd, rs1, imm	Bitwise XOR between rs1 and imm; result in rd
SLTI	slti rd, rs1, imm	Sets rd to 1 if rs1 < sign-extended imm (signed), else 0
SLTIU	sltiu rd, rs1, imm	Sets rd to 1 if rs1 < imm (unsigned), else 0
SRAI	srai rd, rs1, imm	Arithmetic right shift of rs1 by imm; result in rd
SLLI	slli rd, rs1, shamt	Shifts rs1 left logically by shamt bits; stores result in rd
ADD	add rd, rs1, rs2	Adds rs1 and rs2; stores result in rd
SUB	sub rd, rs1, rs2	Subtracts rs2 from rs1; stores result in rd
AND	and rd, rs1, rs2	Performs bitwise AND between rs1 and rs2; stores result in rd
OR	or rd, rs1, rs2	Performs bitwise OR between rs1 and rs2; stores result in rd
XOR	xor rd, rs1, rs2	Performs bitwise XOR between rs1 and rs2; stores result in rd
SLT	slt rd, rs1, rs2	Sets rd to 1 if rs1 < rs2 (signed), else 0
SLTU	sltu rd, rs1, rs2	Sets rd to 1 if rs1 < rs2 (unsigned), else 0
SRA	sra rd, rs1, rs2	Shifts rs1 right arithmetically by rs2; stores result in rd
SRL	srl rd, rs1, rs2	Shifts rs1 right logically by rs2; stores result in rd
SLL	sll rd, rs1, rs2	Shifts rs1 left logically by rs2; stores result in rd
SB	sb rs2, offset(rs1)	Stores least significant byte of rs2 at memory[rs1 + offset]
SW	sw rs2, offset(rs1)	Stores 32-bit word from rs2 at memory[rs1 + offset]
LB	lb rd, offset(rs1)	Loads byte from memory[rs1 + offset] into rd with sign-extension
LW	lw rd, offset(rs1)	Loads 32-bit word from memory[rs1 + offset] into rd
LUI	lui rd, imm	Loads upper 20 bits of imm into rd, lower 12 bits are zeros
AUIPC	auipc rd, imm	Adds imm (shifted left 12 bits) to PC and stores result in rd
JAL	jal rd, offset	Stores PC+4 in rd and jumps to PC + offset
JALR	jalr rd, rs1, offset	Stores PC+4 in rd and jumps to (rs1 + offset) with LSB cleared
BEQ	beq rs1, rs2, offset	Branches to PC + offset if rs1 == rs2
BNE	bne rs1, rs2, offset	Branches to PC + offset if rs1 != rs2
BLT	blt rs1, rs2, offset	Branches to PC + offset if rs1 < rs2 (signed)
BGE	bge rs1, rs2, offset	Branches to PC + offset if rs1 >= rs2 (signed)
BLTU	bltu rs1, rs2, offset	Branches to PC + offset if rs1 < rs2 (unsigned)
BGEU	bgeu rs1, rs2, offset	Branches to PC + offset if rs1 >= rs2 (unsigned)

Table 2: RISC-V basic assembly code format and usage

Each 32-bit binary instruction has a corollary in assembly language[3] in addition to some user functionality for jumping like labels[4] *Table 2*.

[2]**rd** is the destination register, **rs** is the source register, and **shamt** is the shift amount.

[3]A user friendly shorthand for pure binary instructions, designed to make the process of hand rolling binaries easier.

[4]Syntax for labels: "<name>:" – They are used for marking places in the code to jump to. Although not directly converted into binaries, their position is used to calculate the offset for relative jumping.

The memory for a program is split into 2 sections: TEXT and STACK. A hidden register (not part of the 32 core) stores the **Program Counter** (PC). This is used to tell the processor core which instruction to execute.

Memory is made up of many 8-bit registers. The **text** is consists of a sequence of binary instructions loaded into memory. Note that 4 registers will be used to store a 32-bit instruction so to move to the next instruction, the PC must be incremented by 4.

Similarly, the **stack** is stored in 8-bit data registers and can store bytes, half-words (16-bit), and words (32-bit).

1.2 Example Programs

Simple for-loop multiplier example

```
1 | START:
2 |   addi x1, x0, 8        // x1 = 8; pc+=4;
3 |   addi x2, x0, 7        // x2 = 7; pc+=4;
4 |   addi x3, x0, 0        // x3 = 0; pc+=4;
5 |   addi x10, x0, 0       // x10 = 0; pc+=4;
6 | LOOP:
7 |   bge x3, x1, COMPLETE  // if (x3 >= x1) ? pc -> COMPLETE : pc+=4;
8 |   addi x3, x3, 1        // x3 += 1; pc+=4;
9 |   add x10, x10, x2      // x10 += x2; pc+=4;
10|   jal x0, LOOP          // pc -> LOOP
11| COMPLETE:
12|   jal x0, 0             // pc += 0;
```

Note that for lines 7 and 10, COMPLETE will translate to `pc += 16` and LOOP into `pc -= 12`. This is done by the compiler.

Loading and storing into RAM example

```
START:
  addi x1, x0, 0    // x1 = 0;
  addi x2, x0, 111  // x2 = 111;
  sw x2, 0(x1)      // MEM(x1+0) = x2;
  addi x2, x0, 222  // x2 = 222;
  sw x2, 4(x1)      // MEM(x1+4) = x2;
  lw x10, 0(x1)     // x10 = MEM(x1+0);
  lw x11, 4(x1)     // x11 = mem(x1+4);
```

In RISC-V, a 32-bit word spans four 8-bit memory registers, and the hardware enforces that word operations use offsets that are multiples of 4. Similarly, for 16-bit half-words, offsets must be multiples of 2.

2 Exercise

Learning outcomes:

- Understand and implement RV32I assembly
- Ability to convert between assembly and machine code
- Understanding of the MULSI3 algorithm

It is recommended to build familiarity with the example programs before starting this labs. The following external resources are useful for testing assembly and gaining familiarity with the binaries.

- **RISC-V interpreter:** https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/
- **RISC-V encoder:** https://luplab.gitlab.io/rvcodecjs/

2.1 Implementation of MULSI3 **[20 Marks]**

This will lab requires an implementation of MULSI3[5] in RV32I binaries.

Mulsi3 is an algorithm used by compilers to perform multiplication of two numbers without a dedicated hardware multiplier.

MULSI3 algorithm

```
Load multiplicand from MEM[0];
Load multiplier from MEM[4];
result = 0;           // initialize result to 0
While (multiplier > 0) {
  if (multiplier LSB == 1) then result += multiplicand;
  multiplicand << 1;  // left shift
  multiplier >>> 1;   // logical right shift
   } // while
store result in MEM[8]
```

Only the final answer inside of MEM[8] will be checked at the end of each test, and implementation decisions like which registers to use are left open.

****Extension**

The current algorithm can only multiply two unsigned integers. Change, or add onto, this algorithm to handle signed integers inside MEM[0] and MEM[4] and output their multiplication to MEM[8].

2.2 Submission instructions

The final submission should be in the form of a submission.hex file, which can be found inside lab1/work/submission.hex. In addition, a readme.txt file stating whether the design is compatible with unsigned or signed integers.

[5]MULSI3 is code for mul(Multiply), s(signed), i(integer), 3(32-bit).

Example submission format

```
// These are comments and will be ignored
AA AA AA AA // Instruction 1
BB BB BB BB // all 32 bit instructions consist of 8 hexadecimal characters
CC CC CC CC // Each instruction should have its own line
00 00 00 6F // It is good practice to end with JAL x0, 0
```

To test your design, run the command `<./simulate -unsigned>` or `<./simulate -signed>` inside the `lab1/work` directory.