University of Connecticut
ECE 3401 Digital System Design

Introduction to *riscv-uconn*

The programming assignments (PAs) will make use of *riscv-uconn*, a RISC-V simulator developed by UConn's *Computer Architecture Group (CAG)*. This guide provides the instruction set architecture.

1 Instruction Set Architecture

1.1 Memory

riscv-uconn memory is partitioned into instructions and data, and its total size is limited to 16,384 addresses. A word (4 bytes, or 32-bits) is stored at each memory address, leading to a total memory capacity of 65,538 bytes. The machine only supports word addressable `memory[]`.

Instructions reside in the first 256 locations of memory, starting from address 0. Each instruction is one word. A total of 256 instructions (1,024 bytes) can be stored in memory. Each instruction word is read from right to left.

Data resides in addresses 256 through 16,383. Each address contains a single word of data.

1.2 Program Counter

The machine's program counter register (`pc`) initially points at address 0, and addresses the first instruction word (4 bytes). The next instruction word is at address 4, and so on and so forth. The index into the `memory[]` array is always computed by dividing `pc` by 4. For example, if the `pc` is 32, the index containing the corresponding instruction word is calculated as $32/4 = 8$. Instruction execution increments the program counter. However, control flow instructions, may modify the next program counter to a non-sequential instruction address. Make sure to pay special attention to where control-flow instructions resolve.

1.3 Registers

The machine implements a RISC-V ISA with 32 registers, where each register is 32-bits (or one word). These ISA registers are shown in Table 1.

Register Number	ABI Name	Description
`x0`	`zero`	hardwired 0x00000000
`x1-4`	`ra, sp, gp, tp`	return address, stack pointer, global pointer, thread pointer
`x5-7`	`t0-2`	temporary registers
`x8-9`	`s0-1`	saved registers
`x10-11`	`a0-1`	function arguments / return values
`x12-17`	`a2-7`	function arguments
`x18-27`	`s2-11`	saved registers
`x28-31`	`t3-6`	temporary registers

Table 1: riscv-uconn registers and their purposes.

The `zero` register is expected to contain a value of 0, but it will be set to 1 to trigger program termination. The mapping from register indices (0–31) to register names can be found in `Register_file.vhd`.

1.4 Instructions

The *riscv-uconn* instruction format is the same as the standard RISC-V 32-bit integer instruction set. A 32-bit instruction is broken down into six formats: R-Type (figure 1.1), I-Type (figure 1.2), S-Type (figure1.3), (figure B-Type) 1.4, U-Type (figure 1.5), and J-Type (figure 1.6).

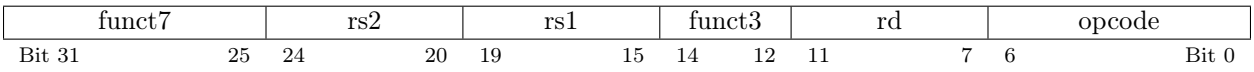

Figure 1.1: R-Type instruction format

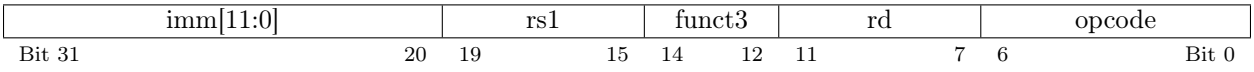

Figure 1.2: I-Type instruction format

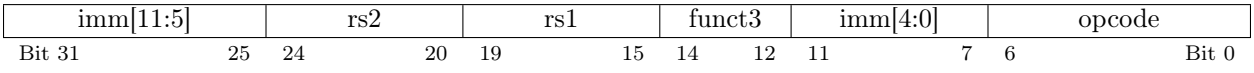

Figure 1.3: S-Type instruction format

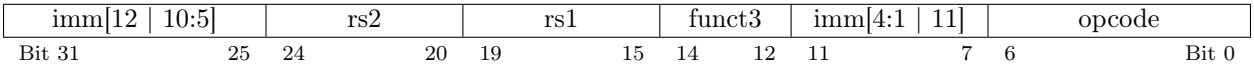

Figure 1.4: B-Type instruction format

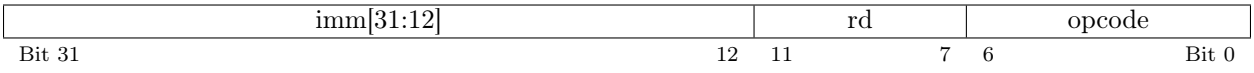

Figure 1.5: U-Type instruction format

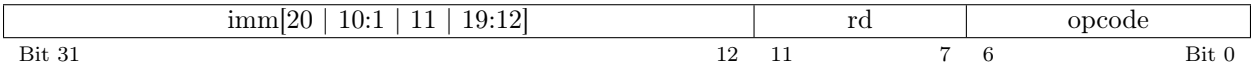

Figure 1.6: J-Type instruction format

The 7-bit `opcode` field, 3-bit `funct3`, and 7-bit `funct7` fields are used to differentiate between instruction types. The 5-bit `rd`, `rs1`, and `rs2` fields encode the indices of the destination, source 1, and source 2 registers, respectively. The `imm` field encodes the immediate/offset value used by various instruction types. The size and encoding of the immediate varies depending on the instruction type. Fields for each instruction type used in this course are already extracted for you in the `Decode.vhd`.

2 Assembler

The *riscv-uconn* assembler is provided to you and will not be modified. However, you will need to compile it by following the instructions in the programming assignment handout. The assembler converts instructions to machine code. The assembler directives `.text` and `.data` direct the assembler to the start of instruction and data memory respectively. For example, instructions following `.text` are converted into 32-bit machine code starting at address 0. The `.data` assembler directive identifies the start of data memory. Each data word (defined with the `.word`) following the directive will be loaded into memory starting at address 256. For example, the third word after `.data` will have a memory address of 258.

You are provided with the VHDL code to initialize the machine's registers and memory, and load the assembled `*.out` file into the machine's memory beginning with the `.text` (code) section. Each row (instruction)

in the `*.out` file is read one by one and loaded into memory starting at address 0. The row containing `11111111111111111111111111111111` indicates the end of the code section, and is not loaded into memory. The following rows contain the data section and are loaded into memory starting at address 256.

The program counter is initalized to zero and after deasserting reset, the design starts fetching instructions from address 0 and execute them in one cycle per instruction. A terminate flag detects the termination instruction, i.e., when an instruction has written a 1 to the `x0` register, such as in `addi x0, x0, 1`, the program terminates by sending a single cycle `terminate` to the testbench.

3 Implementation Details for Instructions

The implementation details of each instruction in the subset of the RISC-V ISA for this course are described next.

3.1 R-Type Instructions: ADD, SUB, SLL, SRL

ADD

Full Name:	Addition
Description:	Add the contents of two registers and store the result in a register.
Assembler Syntax:	`add rd, rs1, rs2`
Operation:	rd = rs1 + rs2
Decode:	`registers[rs1]` and `registers[rs2]` are read as the two ALU operands.
Execute:	The two ALU operands are added using the + operator to compute the output value.
Writeback:	`registers[rd]` is updated with the output value.

SUB

Full Name:	Subtraction
Description:	Subtract the contents of two registers and store the result in a register.
Assembler Syntax:	`sub rd, rs1, rs2`
Operation:	rd = rs1 − rs2

Implementation is the same as ADD except that the − operator is used to compute the output value in execute.

SLL

Full Name:	Shift Left Logical
Description:	Shift the contents (values) of rs1 register left by the specified number of positions identified by the lower 5-bits of the second rs2 register. It stores the result in rd (Note: whenever the value of rs2 $>$ 32, shifting has the same effect as when rs2 = 32).
Assembler Syntax:	`sll rd, rs1, rs2`
Operation:	rd = rs1 << rs2

Implementation is the same as ADD except that the << operator is used to compute the output value in execute.

SRL

Full Name:	Shift Right Logical
Description:	Shift the contents (values) of rs1 register left by the specified number of positions identified by the lower 5-bits of the second rs2 register. It stores the result in rd (Note: whenever the value of rs2 > 32, shifting has the same effect as when rs2 $= 32$).
Assembler Syntax:	`srl rd, rs1, rs2`
Operation:	rd = rs1 >> rs2

Implementation is the same as ADD except that the >> operator is used to compute the output value in execute.

3.2 I-Type Instructions: LW, ADDI, SLLI, SRLI

LW

Full Name:	Load Word
Description:	A word is loaded into a register from the specified memory address.
Assembler Syntax:	`lw rd, offset(rs1)`
Operation:	alu_out = rs1 + offset rd = memory[alu_out]
Decode:	`registers[rs1]` is read to determine the base for the address calculation and is set as the first ALU operand. The second ALU operand, the address offset, is set to the immediate field.
Execute:	The memory address is calculated by adding the two ALU operands with the + operator and is stored in `mem_addr`. `mem_buffer` is set to `memory[mem_addr]`.
Writeback:	`mem_buffer` is stored in `registers[rd]`.

ADDI

Full Name:	Add Immediate
Description:	Add the contents of a register to a sign-extended immediate value and store the result in a register.
Assembler Syntax:	`addi rd, rs1, immediate`
Operation:	rd = rs1 + immediate
Decode:	`registers[rs1]` is read as the first ALU operand. The second ALU operand is set to the immediate field.
Execute:	The output value is computed as the addition of the two operands using the + operator.
Writeback:	`registers[rd]` is updated with the output value.

SLLI

Full Name:	Shift Left Logical Immediate
Description:	Shift the contents of rs1 left by the least 5-bit immediate positions and store the result in rd.
Assembler Syntax:	`slli rd, rs1, immediate`
Operation:	rd = rs1 << immediate

Implementation is the same as ADDI except that the << operator is used to compute the output value in the execute stage.

SRLI

Full Name:	Shift Right Logical Immediate
Description:	Shift the contents of rs1 right by the least 5-bit immediate positions and store the result in rd.
Assembler Syntax:	`srli rd, rs1, immediate`
Operation:	rd = rs1 >>immediate

Implementation is the same as ADDI except that the >> operator is used to compute the output value in the execute stage.

3.3 S-Type Instructions: SW

SW

Full Name:	Store Word
Description:	The contents of a register is stored at the specified memory address.
Assembler Syntax:	`sw rs2, offset(rs1)`
Operation:	alu_out = rs1 + offset memory[alu_out] = rs2
Decode:	`registers[rs1]` is read to determine the base for the address calculation and is set as the first ALU operand. The second ALU operand, the address offset, is set to the immediate field. `mem_buffer` is set to `registers[rs2]` to propogate the value to be stored to memory.
Execute:	The memory address is calculated by adding the two ALU operands with the + operator and is stored in `mem_addr`. `mem_buffer` is written to `memory[mem_addr]`.
Writeback:	Nothing is done for this instruction.

3.4 B-Type Instructions: BNE

BNE

Full Name:	Branch on Not Equal
Description:	Branches if the contents of two registers are not equal. Resolves in the execute stage.
Assembler Syntax:	`bne rs1, rs2, offset`
Operation:	if rs1 $\neq$ rs2, then pc_n = inst_addr + offset; else do nothing (normal pc_n = pc + 4 is carried out).

Implementation is the same as BEQ except that the branch condition tests for non-equality in the execute stage.

Note: The offset in B-Type instructions is calculated by the assembler using the difference between the 32-bit address of the instruction and the address of the label. For example, if a program wants to loop back four instructions, then the offset will be stored as `0xFFFFFFF0` or `-16`. The branch address will then be calculated as `pc+(-16)`, which will allow the program to loop back four instructions. Similarly, if the program wants to loop forward 4 instructions, then the offset will be stored as `0x10` or `16`. When the condition is checked, the branch address will be calculated as `pc+16`.