## 

## 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)*. Each PA will provide incomplete simulator code and a detailed description of the functionality that must be implemented, as well as the expected deliverables to be submitted through HuskyCT.

## 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[] array.

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. Instructions increment the program counter by calling advance\_pc() function. However, control flow instructions (BNE, BEQ, BLT, BGE, JAL, and JALR), 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 stored in the registers[] array, as shown in Table 1.

Register Number	ABI Name	Description
х0	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\_map.c in the src

directory.

#### 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 (figure 1.3), (figure B-Type) 1.4, U-Type (figure 1.5), and J-Type (figure 1.6).

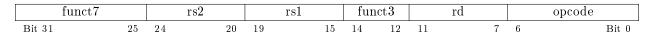


Figure 1.1: R-Type instruction format

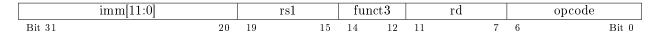


Figure 1.2: I-Type instruction format

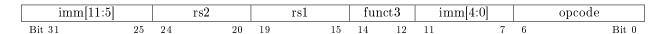


Figure 1.3: S-Type instruction format



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 are already extracted for you at the start of the decode stage in sim\_stages.c using the function decode\_fields(). The binary values for different fields are defined in instruction\_map.h. You may find #define statements helpful when working on your program assignments.

## 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 assembler's README.md. 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.

#### 3 Simulator Structure

The simulator source code is located in the src directory. sim\_core.c contains the simulator initialization functions and the main simulation loop as well as the machine's registers and memory. sim\_stages.c contains the functions for implementation of the pipeline stages.

sim\_core.c contains the simulator's entry point main(), initialization function initialize(), main simulation loop process\_instructions(), registers, and memory.

main() simply invokes the initialization function and main simulation loop, and prints state information (committed instructions, simulated cycles, register contents, memory contents, etc.) after the simulation terminates.

process\_instructions() contains the main simulation loop responsible for executing instructions. In the 4-stage pipeline implementation, the simulation loop invokes the pipeline functions (fetch(), decode(), execute(), and writeback()) and handles the passing of state information between stages. Note the order of the invocation of these functions is done to ensure the pipeline concurrency is managed correctly by the simulator. The program counter, pc is also updated with the next program counter, pc\_n. The simulation loop checks for the simulation termination condition, i.e., when an instruction has written a 1 to the x0 register, such as in addi x0, x0, 1, the simulator terminates.

The implementation of pipeline stages are in sim\_stages.c. The fetch() function fetches an instruction and stores its dynamic metadata in the State structure. Then, the simulator calls advance\_pc(). Finally, the State structure is returned and gets forwarded to the input of decode(). The output of decode() is then forwarded to execute(), and so on and so forth.

State information is passed between pipeline stages using the State structure, which is maintained for each pipeline stage and contains dynamic information about the instruction being processed. The definition of the State structure can be found in sim\_core.h, and its contents are outlined in Figure 3.1.

# 4 Implementation Details for Instructions

The implementation details of each instruction in our RISC-V ISA are described next.

Struct Member	Description
inst	fetched instruction
inst_addr	address of instruction
opcode	opcode field
funct3	3-bit function field
funct7	7-bit function field
rd	destination register specifier
rs1	source 1 register specifier
rs2	source 2 register specifier
imm	immediate value
mem_buffer	memory data for LW/SW instructions
mem_addr	memory address for LW/SW instructions
br_addr	target address for B-Type instructions
link_addr	return address for JAL/JALR instructions
alu_in1	first ALU operand
alu_in2	second ALU operand
alu_out	ALU output

Figure 3.1: Fields of the State struct

## 4.1 R-Type Instructions: ADD, SUB, AND, OR, XOR, SLT, 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 Stage:** The two ALU operands are added using the + operator to compute the output value.

Writeback Stage: 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 the execute stage.

#### AND

Full Name: Bitwise AND

**Description:** Bitwise AND the contents of two registers and store the result in a register.

Assembler Syntax: and 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 the execute stage.

#### OR

Full Name: Bitwise OR

**Description:** Bitwise OR the contents of two registers and store the result in a register.

Assembler Syntax: or 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 the execute stage.

#### XOR

Full Name: Bitwise Exclusive OR

**Description:** Bitwise XOR the contents of two registers and store the result in a register.

Assembler Syntax: xor rd, rs1, rs2 Operation:  $rd = rs1 \land rs2$ 

Implementation is the same as ADD except that the  $^{\wedge}$  operator is used to compute the output value in the execute stage.

#### SLT

Full Name: Set on Less Than

**Description:** If rs1 is less rs2, then rd is set to one. Otherwise, rd is set to zero.

Assembler Syntax: slt 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 the execute stage.

#### SLL

Full Name: Shift Left Logical

**Description:** Shift the contents of rs1 left by rs2 positions and store the result in rd (Note: whenever

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 the execute stage.

#### SRL

Full Name: Shift Right Logical

**Description:** Shift the contents of rs1 right by rs2 positions and store the result in rd (Note: whenever

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 the execute stage.

# 4.2 I-Type Instructions: LW, JALR ,ADDI, ANDI, ORI, XORI, SLTI, 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 Stage:** 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 Stage: mem\_buffer is stored in registers[rd].

#### JALR

Full Name: Jump and Link Register

**Description:** Jumps unconditionally to the calculated address. Stores the return address in rd if it

is not the zero register (x0), otherwise a jump with no link is performed. Resolves in

the decode stage.

Assembler Syntax: jalr rd, rs1, offset

Operation:  $pc_n = rs1 + offset$ 

 $rd = inst\_addr + 4$  if rd is not the zero register (x0); else perform no writeback

**Decode Stage:**  $link_addr$  is set to  $inst_addr + 4$ .  $pc_n$  is set to registers[rs1] + (sign-extended)

immediate.

**Execute Stage:** Nothing is done for this instruction.

Writeback Stage: registers[rd] is set to link\_addr if  $rd \neq 0$ . Otherwise, does nothing.

#### 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 Stage: registers[rs1] is read as the first ALU operand. The second ALU operand is set to

the immediate field.

**Execute Stage:** The output value is computed as the addition of the two operands using the + operator.

Writeback Stage: registers[rd] is updated with the output value.

**Note**: The encoding for a "nop" (no operation, or an instruction that does nothing) is represented by the instruction addi x0, x0, 0 which has no side effects on the register and memory state of the machine.

#### ANDI

Full Name: Bitwise AND Immediate

**Description:** Bitwise AND the contents of a register to a sign-extended immediate value and store

the result in a register.

Assembler Syntax: and 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.

#### ORI

Full Name: Bitwise OR Immediate

**Description:** Bitwise OR the contents of a register to a sign-extended immediate value and store the

result in a register.

Assembler Syntax: ori rd, rs1, immediate Operation:  $rd = rs1 \mid immediate$ 

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

#### XORI

Full Name: Bitwise Exclusive OR Immediate

**Description:** Bitwise XOR the contents of a register to a sign-extended immediate value and store

the result in a register.

Assembler Syntax: xori rd, rs1, immediate Operation:  $rd = rs1 \land immediate$ 

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

#### SLTI

Full Name: Set on Less Than Immediate

**Description:** If rs1 is less than the sign-extended immediate value, rd is set to one. Otherwise, rd is

set to zero.

Assembler Syntax: slti 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.

#### 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:  $rd = rs1 \ll immediate$ 

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.

## 4.3 S-Type Instructions: SW

#### $\mathbf{SW}$

Full Name: Store Word

**Description:** The contents of a register is stored at the specified memory address.

Assembler Syntax: sw rs2, offset(rs1)

Operation:  $\begin{array}{c} alu\_out = rs1 + offset \\ memory[alu\ out] = rs2 \end{array}$ 

Decode Stage: 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 propagate the value to be

stored to memory.

Execute Stage: 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 Stage: Nothing is done for this instruction.

#### 4.4 B-Type Instructions: BEQ, BNE, BLT, BGE

#### $\mathbf{BEQ}$

Full Name: Branch on Equal

**Description:** Branches if the contents of two registers are equal. Resolves in the execute stage.

Assembler Syntax: beg rs1, rs2, offset

Operation: if rs1 = rs2, then pc\_n = inst\_addr + offset; else do nothing (normal pc\_n = pc + 4)

is carried out).

Decode Stage: registers[rs1] and registers[rs2] are read as the two ALU operands. br\_addr is

set to  $inst\_addr + immediate$ .

**Execute Stage:** The two ALU operands are compared to determine if the branch will be taken or not.

If the branch is taken (i.e., the two ALU operands are equal), then pc\_n is set to br\_addr, overwritting the advance\_pc() call performed in the fetch stage. Otherwise,

nothing is done here.

Writeback Stage: Nothing is done for this instruction.

#### 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.

#### BLT

Full Name: Branch on Less Than

**Description:** Branches if the contents of one register is less than another. Resolves in the execute

stage.

Assembler Syntax: blt rs1, rs2, offset

**Operation:** if rs1 < 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 a less than condition in the execute stage.

#### BGE

Full Name: Branch on Greater Than or Equal To

**Description:** Branches if the contents of one register is greater than or equal to another. Resolves

in the execute stage.

Assembler Syntax: bge rs1, rs2, offset

**Operation:** if  $rs1 \ge rs2$ , 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 a greater than or equal to condition 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 0xffffffff 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.

## 4.5 U-Type Instructions: LUI

#### LUI

Full Name: Load Upper Immediate

**Description:** The 20-bit immediate value is extracted and stored in the upper 20 bits of a register.

The lower 12 bits are cleared to zero.

Assembler Syntax: lui rd, immediate

Operation: rd = immediate[31:12]

**Decode Stage:** Nothing is done for this instruction.

**Execute Stage:** The upper 20 bits of the ALU output is set to the immediate value. The lower 12 bits

of the ALU are set to 0.

Writeback Stage: registers[rd] is updated with the output value.

## 4.6 J-Type Instructions: JAL

#### JAL

Full Name: Jump and Link

Description: Jumps unconditionally to the calculated address. Stores the return address in rd if it

is not the zero register (x0), otherwise a jump with no link is performed. Resolves in

the decode stage.

Assembler Syntax: jal rd, offset

Operation:  $pc_n = inst_a ddr + offset$ 

operation:  $rd = inst \quad addr + 4 \text{ if } rd \text{ is not the zero register (x0); else perform no writeback.}$ 

**Decode Stage:** link\_addr is set to inst\_addr + 4. pc\_n is set to inst\_addr + immediate.

**Execute Stage:** Nothing is done for this instruction.

Writeback Stage: registers[rd] is set to link\_addr if  $rd \neq 0$ . Otherwise, do nothing.

Note: For the J-Type instruction, the offset immediate is not encoded with the bits in-order. The decode \_-fields() function re-orders the bits within the instruction according to the encoding indices to correctly calculate the target address.

# 5 Debugging

You have several options for debugging your simulator implementation. printf statements can be added anywhere in sim\_stages.c so long as they are properly gated by the debug flag variable at the top of the file. util.c provides some helpful debugging functions that output the register (rdump()) and memory (mdump()) contents. Several of these debugging functions are used in the core simulator implementation by default. You may use these functions so long as they are properly gated by the debug flag.

A facility called pipe trace is added to the simulator to support visualization of instruction processing across cycles. The file pipe\_trace.txt will be created whenever the simulator is executed. The pipe\_trace flag variable in sim\_stages.c toggles whether pipe tracing is enabled or not. You may insert debugging information into the pipe trace file so long as it is properly gated with the debug flag. Refer to sim\_core.c for examples of writing to the pipe trace. Additionally, when using the pipe\_trace output, you may configure the display mode of the trace. You may toggle between register numbers and ABI names, and between hexidecimal and decimal values for immediate. This is toggled by changing the pipe\_trace\_mode variable in sim\_stages.c according to the below table: Finally, you may use the GDB debugger (see the guide here: https://condor.depaul.edu/glancast/373class/docs/gdb.html). You can run the simulator with a unit test under GDB using the following:

Value of pipe_trace_mode	Register Display Setting	Immediate Setting
0	Register No.	Hexidecimal
1	ABI Name	Hexidecimal
2	Register No.	Decimal
3	ABI Name	Decimal

\$ gdb ./simulator unit\_test.out