Department of Electrical and Computer Engineering
Queen's University

ELEC 374 Digital Systems Engineering
Laboratory Project

Winter 2024

Designing a Simple RISC Computer: CPU Specification

---

1. Objectives

The purpose of this project is to design, simulate, implement, and verify a Simple RISC Computer (Mini SRC), consisting of a simple RISC processor, memory, and I/O. You are to use the Intel Quartus Prime Design Software and ModelSim-Intel for this purpose, and your design is to be implemented on the Cyclone V chip (5CEBA4F23C7) of the DE0-CV development board.

The Mini SRC is similar to the SRC described in the Lab Reader, reproduced from the text by Heuring and Jordan. The Datapath, Control Unit, and Memory Interface for Mini SRC have the same relationship as shown in Figure 4.1 on page 142 of the Lab Reader for the SRC system. The processor design is similar to the information presented in Figures and Tables on pages 143 through 167 of the Lab Reader. As part of the learning process for the CPU design project, make sure you carefully read the Mini SRC CPU Specification (this document) and the descriptions of different lab phases, the CPU Design Project Tutorial, the Introduction to Intel Quartus Prime Design Software, ModelSim-Intel FPGA Starter Edition, and DE0-CV Development Board, and the Lecture Slides on Computer Arithmetic, and Verilog or VHDL.

2. Processor Specification

The Mini SRC is a 32-bit machine, having a 32-bit datapath and sixteen 32-bit registers R0 to R15, with R0 to R7 as general-purpose registers, R8 and R9 as the two return value registers, R10 to R13 as the four argument registers, R14 as the stack pointer (SP), and R15 as the return address register (RA), holding the return address for a *jal* instruction. It also has two dedicated 32-bit registers HI and LO for multiplication and division instructions. Note that Mini SRC does not have a condition code register. Rather, it allows any of the general-purpose registers to hold a value to be tested for conditional branching. The memory unit is 512 words.
The following is a formal definition of the Mini SRC.

Processor State

PC<31..0>:	32-bit Program Counter (PC)
IR<31..0>:	32-bit Instruction Register (IR)
R[0..15]<31..0>:	16 32-bit registers, named R[0] through R[15]
R[0..7]<31..0>:	Eight General-Purpose Registers
R[8..9]<31..0>:	Two Return Value Registers
R[10..13]<31..0>:	Four Argument Registers
R[14]<31..0>:	Stack Pointer (SP)
R[15]<31..0]:	Return Address Register (RA)
HI<31..0>:	32-bit HI Register dedicated to keep the high-order word of a Multiplication product, or the Remainder of a Division operation
LO<31..0>:	32-bit LO Register dedicated to keep the low-order word of a Multiplication product, or the Quotient of a Division operation

Memory State

Mem[0..511]<31..0>:	512 words (32 bits per word) of memory
MDR<31..0>:	32-bit Memory Data Register
MAR<31..0>:	32-bit Memory Address Register

I/O State

In.Port<31..0>:	32-bit Input Port
Out.Port<31..0>:	32-bit Output Port
Run.Out:	Run/halt Indicator
Stop.In:	Stop signal
Reset.In:	Reset signal

The *Arithmetic Logic Unit* (ALU) performs 13 operations: addition, subtraction, multiplication, division, shift right, shift right arithmetic, shift left, rotate right, rotate left, logical AND, logical OR, Negate (2's complement), and NOT (1's complement).

The instructions in Mini SRC are one-word (32-bit) long each. They can be categorized as Load and Store instructions, Arithmetic and Logical instructions, Conditional Branch and Jump instructions, Input/Output instructions, and miscellaneous instructions. There are no push and pop instructions (they can be implemented by other instructions). The following six addressing modes are supported: Direct, Indexed, Register, Register Indirect, Immediate, and Relative.

2.1 Instruction Formats

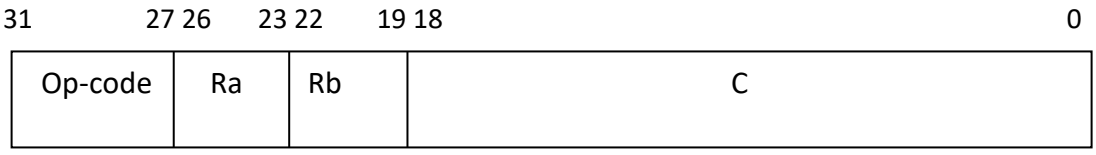
There are five instruction formats, as shown in the table below.

Name	Fields					Comments
Field size	31..27 5 bits	26..23 4 bits	22..19 4 bits	18..15 4 bits	14..0 15 bits	All instructions are 32-bit long
R-Format	OP-code	Ra	Rb	Rc	Unused	Arithmetic/Logical
I-Format	OP-code	Ra	Rb	Constant C / Unused		Arithmetic/Logical; Load/Store; Imm.
B-Format	OP-code	Ra	C2	Constant C		Branch
J-Format	OP-code	Ra	Unused			Jump; Input/Output; Special
M-Format	OP-code	Unused				Misc.

The instruction formats for different categories are detailed below:

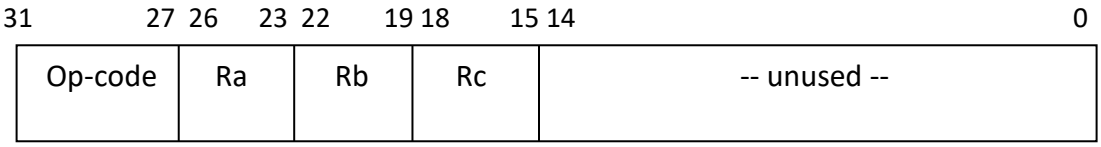
- Load and Store instructions: operands in memory can be accessed only through load/store instructions.

(a) ld, ldi, st I-Format

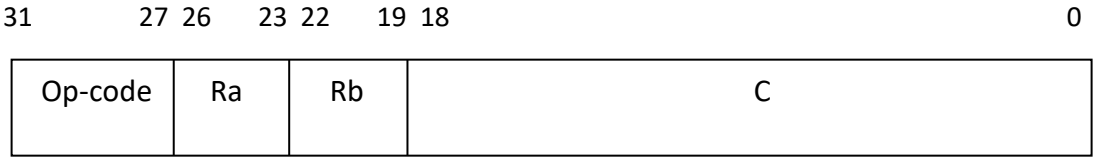


- Arithmetic and Logical instructions:

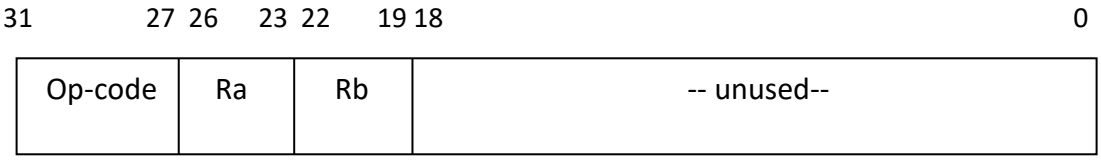
  (a) add, sub, and, or, shr, shra, shl, ror, rol R-Format



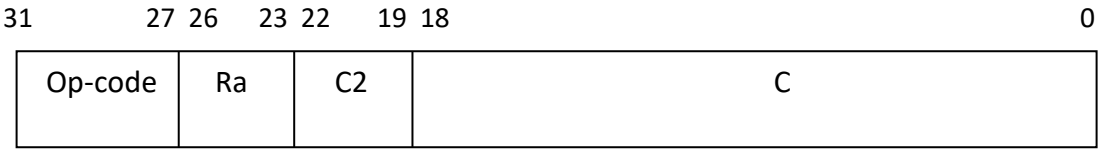
  (b) addi, andi, ori I-Format



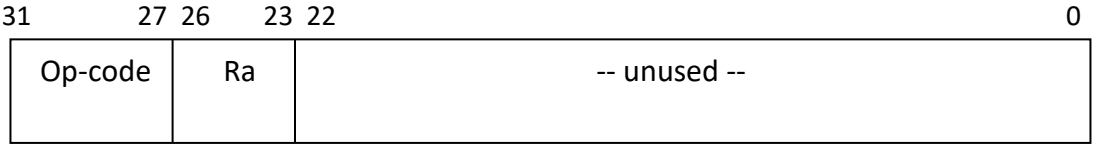
  (c) mul, div, neg, not I-Format



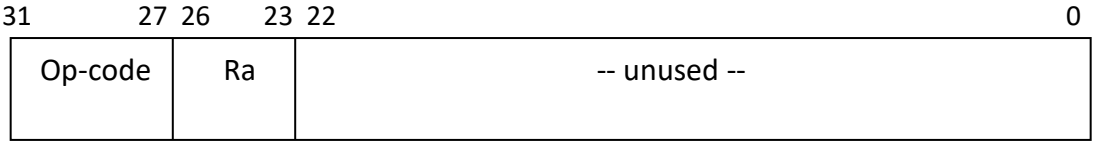
- Branch instructions: brzr, brnz, brmi, brpl B-Format



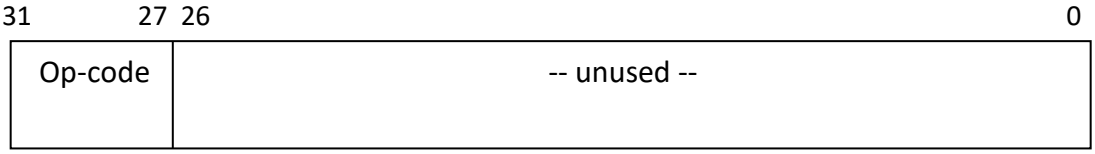
- Jump instructions: jr, jal J-Format



- Input/Output and MFHI/MFLO instructions: in, out, mfhi, mflo J-Format



- Miscellaneous instructions: nop, halt M-Format



Op-code: specifies the operation to be performed.
Ra, Rb, Rc: 0000: R0, 0001: R1, …, 1111: R15
C: constant (data or address)
C2: condition --00: branch if zero
--01: branch if nonzero
--10: branch if positive
--11: branch if negative

Notation: x: 0 or 1
- : unused

2.2 Instructions

The instructions (with their op-code patterns shown in parentheses) perform the following operations:

Load and Store Instructions

ld, ldi, st:

		Assembly language
ld: Load Direct (00000xxxx0000xxxxxxxxxxxxxxxxxxx)	R[Ra] ← M[C (sign-extended)] Direct addressing, Rb = R0	ld Ra, C
ld: Load Indexed/Register Indirect (00000xxxxxxxxxxxxxxxxxxxxxxxxxxx)	R[Ra] ← M[R[Rb] + C (sign-extended)] Indexed addressing, Rb ≠ R0 If C = 0 → Register Indirect addressing	ld Ra, C(Rb)
ldi: Load Immediate (00001xxxx0000xxxxxxxxxxxxxxxxxxx)	R[Ra] ← C (sign-extended) Immediate addressing, Rb = R0	ldi Ra, C
(00001xxxxxxxxxxxxxxxxxxxxxxxxxxx)	R[Ra] ← R[Rb] + C (sign-extended) Immediate addressing, Rb ≠ R0 If C = 0 → instruction acts like a simple register transfer If C ≠ 0 and Ra = Rb → Increment/decrement instruction	ldi Ra, C(Rb)
st: Store Direct (00010xxxx0000xxxxxxxxxxxxxxxxxxx)	M[C (sign-extended)] ← R[Ra] Direct addressing, Rb = R0	st C, Ra
st: Store Indexed/Register Indirect (00010xxxxxxxxxxxxxxxxxxxxxxxxxxx)	M[R[Rb] + C (sign-extended)] ← R[Ra] Indexed addressing, Rb ≠ R0 If C = 0 → Register Indirect addressing	st C(Rb), Ra

Arithmetic and Logical Instructions

(a): add, sub, shr, shra, shl, ror, rol, and, or

add: Add — R[Ra] ← R[Rb] + R[Rc] — add Ra, Rb, Rc
(00011xxxxxxxxxxx----------)

sub: Sub — R[Ra] ← R[Rb] - R[Rc] — sub Ra, Rb, Rc
(00100xxxxxxxxxxx----------)

shr: Shift Right — Shift right R[Rb] into R[Ra] by count in R[Rc] — shr Ra, Rb, Rc
(00101xxxxxxxxxxx----------)

shra: Shift Right Arithmetic — Shift right arithmetic R[Rb] into R[Ra] by count in R[Rc] — shra Ra, Rb, Rc
(00110xxxxxxxxxxx----------)

shl: Shift Left — Shift left R[Rb] into R[Ra] by count in R[Rc] — shl Ra, Rb, Rc
(00111xxxxxxxxxxx----------)

ror: Rotate Right — Rotate right R[Rb] into R[Ra] by count in R[Rc] — ror Ra, Rb, Rc
(01000xxxxxxxxxxx----------)

rol: Rotate Left — Rotate left R[Rb] into R[Ra] by count in R[Rc] — rol Ra, Rb, Rc
(01001xxxxxxxxxxx----------)

and: AND — R[Ra] ← R[Rb] ∧ R[Rc] — and Ra, Rb, Rc
(01010xxxxxxxxxxx----------)

or: OR — R[Ra] ← R[Rb] $_{V}$ R[Rc] — or Ra, Rb, Rc
(01011xxxxxxxxxxx----------)

(b): addi, andi, ori

addi: Add Immediate — R[Ra] ← R[Rb] + C (sign-extended) — addi Ra, Rb, C
(01100xxxxxxxxxxxxxxxxxxxxxxxxxx)
Immediate addressing
If C = 0 ➔ instruction acts like a simple register transfer
If C ≠ 0 and Ra = Rb ➔ Increment/decrement instruction
Similar to ldi, however Rb can be any register

andi: AND Immediate — R[Ra] ← R[Rb] ∧ C (sign-extended) — andi Ra, Rb, C
(01101xxxxxxxxxxxxxxxxxxxxxxxxxx)
Immediate addressing

ori: OR Immediate — R[Ra] ← R[Rb] v C (sign-extended) — ori Ra, Rb, C
(01110xxxxxxxxxxxxxxxxxxxxxxxxxx) — Immediate addressing

(c): mul, div, neg, not

mul: Multiply — HI, LO ← R[Ra] × R[Rb] — mul Ra, Rb
(01111xxxxxxxx----------)

div: Divide — HI, LO ← R[Ra] ÷ R[Rb] — div Ra, Rb
(10000xxxxxxxx----------)

neg: Negate — R[Ra] ← - R[Rb] — neg Ra, Rb
(10001xxxxxxxx----------)

not: NOT — R[Ra] ← $\overline{R[Rb]}$ — not Ra, Rb
(10010xxxxxxxx----------)

Conditional Branch Instructions

brzr, brnz, brmi, brpl

Branch — PC ← PC + 1 + C (sign-extended) — if R[Ra] meets the condition
(10011xxxx--xxxxxxxxxxxxxxxxxxxx)

Condition:		
--00: branch if zero	brzr	Ra, C
--01: branch if nonzero	brnz	Ra, C
--10: branch if positive	brpl	Ra, C
--11: branch if negative	brmi	Ra, C

Jump Instructions

jr, jal

jr: return from procedure — PC ← R[Ra] — jr Ra
(10100xxxx----------) — If Ra = R15, it is for procedure return

jal: jump and link — R[15] ← PC + 1 — jal Ra
(10101xxxx----------) — PC ← R[Ra]

Input/Output and MFHI/MFLO Instructions

in, out, mfhi, mflo

in: Input — R[Ra] ← In.Port — in Ra
(10110xxxx----------)

out: Output (10111xxxx----------)	Out.Port ← R[Ra]	out	Ra
mfhi: Move from HI (11000xxxx----------)	R[Ra] ← HI	mfhi	Ra
mflo: Move from LO (11001xxxx----------)	R[Ra] ← LO	mflo	Ra

Miscellaneous Instructions

nop, halt

nop: No-operation (11010----------)	Do nothing	nop
halt: Halt (11011----------)	Halt the control stepping process	halt

3. Design Phases and Final Report

There are **four design phases** in this team project, and you are strongly advised to start working on your project early and follow the guidelines in each phase. Labs start in the third week of classes, and **only three students** are allowed per group. You may use an all HDL (VHDL or Verilog) design approach for your CPU. However, a Schematic, mixed HDL/Schematic, or a mixed Verilog/VHDL design methodology is not recommended.
Consult the CPU Design Project Tutorial. Also, you are advised to refer to Section 6, "A Simple Datapath Design in Verilog", of the Introduction to Intel Quartus Prime Design Software, ModelSim-Intel FPGA Starter Edition, and DE0-CV Development Board tutorial document as a starting point for designing your registers, bus, data transfer over the bus, testbenches, etc. There are also many examples in the Lecture Slides on Verilog and VHDL that can help with your design, simulation, and implementation of your lab project.

Phase 1: The processor Datapath will be partially designed and tested using Functional Simulation. Phase one is worth 7% of the course mark. You will demo your design and simulation results in the lab, and then upload your lab report to onQ, consisting of HDL code, schematic screenshots (if any), testbenches, and Functional Simulation results by 11:59pm on the day of demo.

Phase 2: The Datapath will be complemented by adding the “Select and Encode logic”, “CON FF Logic”, branch and jump instructions, “Memory Subsystem”, and the “Input/Output Ports”. It will be tested using Functional Simulation. Phase two is worth 7% of the course mark. You will demo your design and simulation results in the lab, and then upload your lab report to onQ, consisting of HDL code, schematic screenshots (if any), testbenches, Functional Simulation results, and contents of the memory before and after execution of load and store instructions by 11:59pm on the day of demo.

Phase 3: The Control Unit will be designed in Verilog or VHDL, and tested using Functional Simulation. You will be provided with a test program to verify your Control Unit. Phase three is worth 5% of the course mark.
You will demo your design and simulation results in the lab, and then upload your lab report to onQ, consisting of HDL code, schematic screenshots (if any), contents of the memory before and after the program run, and Functional Simulation results by 11:59pm on the day of demo.

Phase 4: The Datapath and Control Unit will be tested together using both Functional Simulation and implementation on the DE0-CV FPGA development board. You will be provided with a test program to verify your CPU design and implementation. Phase four is worth 3% of the course mark. You will demo your design, simulation, and on-board FPGA results in the lab. There is no lab report submission for this phase.

Lab Final Report: You will upload a comprehensive lab final report to onQ, describing all aspects of your CPU design project, including performance results and analysis, conclusions, future work, and the required appendices. The appendices include your final HDL code, schematic screenshots (if any), Functional Simulation results, contents of the memory before and after the program run, and screenshots of the FPGA board showing the displays and switches when running your program. The lab final report is worth 3% of the course mark.

4. Bonus Mark

There will be up to 5% **bonus marks** if you design, simulate, and implement, for instance, a 3-bus architecture, advanced techniques that we learned in class for ALU operations, interrupt/exception handling, stack, new instructions, as well as any other advanced techniques such as pipelining, branch prediction, hazard detection, and superscalar design, etc., to improve performance. You are advised to tackle the 1-bus architecture first, and when you feel comfortable with your design then aim for any other improvements.

5. Schedule

For the lab schedule and the deadline for each phase and final lab report with/without penalty, consult the ELEC 374 Labs page and ELEC 374 Course Schedule. Lab Demos will be done during the labs, and all members of the lab group must be present during their lab demo. There are two "expected demo dates" for each phase to support all the groups. Groups can have their simulation/board demo in either of the two dates (e.g., Feb 5 or Feb 12 for Phase 1, Monday Section), whenever they are ready, and then submit their lab report by 11:59pm on that same day. Only **one lab report** per group should be submitted in onQ. As a universal accommodation, we have also considered an automatic one-week extension without penalty.