IL2234

Digital Systems Design and Verification using Hardware Description Languages

Project Milestone 4

Processor Design & Implementation

Introduction

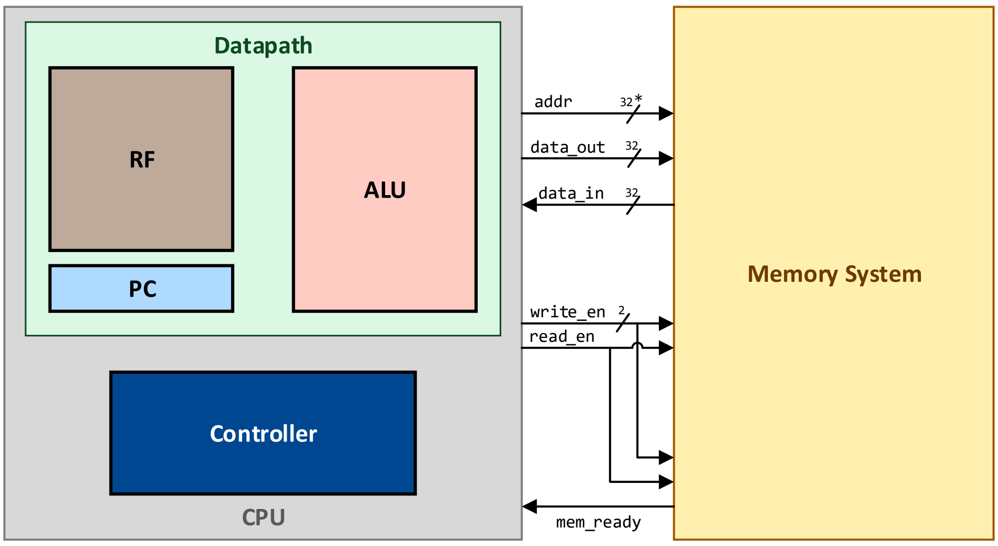
In this milestone, you will design and model both the datapath and the controller for a RISC-V-based processor. For the datapath, you will utilise the ALU and Register File that you previously designed and modelled in SystemVerilog during Milestones 1 and 2.

Overview

The controller FSM is responsible for interpreting and executing the instructions provided in the instruction memory. The controller **fetches** an instruction, **decodes** its contents, and **executes** the appropriate control actions according to the content and type of instruction.

In its simplest form, the controller is responsible for providing the correct control signals to the datapath components: register file, ALU, multiplexer, etc.

A simplified diagram of the CPU can be seen below:



In the next section, you will find the information about the Instruction Set Architecture (ISA).

Instruction Set Architecture

Instruction types

In the base ISA, there are four core instruction formats (R/I/S/U). There are two further variants of the instruction formats (SB/UJ) based on the handling of Immediates. All instructions are fixed 32 bits in length and must be aligned on a four-byte boundary in memory. Each type defines the contents and position of each element in the instruction.

Grouping instructions into types simplifies the decoding hardware, since the decoding and muxing logic needed for each instruction type can be shared across all the instructions of the same type.

R-type instructions

Register-to-register instructions. Used for **arithmetic and logical operations between two registers**; takes two source registers rs1, rs2, applies an ALU operation, and writes the result into a destination register rd.

The type is described as follows:

31 25	24 20	19 15	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode

I-type instructions

Immediate instructions. Used for **operations with immediate values, loads from memory, and jumps**; takes one source register rs1 and a 12-bit signed immediate value imm, performs the operation, and stores the result in a destination register rd.

The type is described as follows:

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	func3	rd	opcode

S-type instructions

Used for **store instructions**, where data from a source register rs2 is written into memory at an address computed by adding a 12-bit signed immediate to a base register rs1; no destination register is used.

The type is described as follows:

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

SB-type instructions

Similar to S-type instructions, but with a different encoding of the immediate.

The type is described as follows:

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	func3	imm[4:1]	imm[11]	opcode

U-type instructions

Used for **building large constants or PC-relative addressing**; loads a 20-bit immediate into the upper 20 bits of a destination register rd, with the lower 12 bits set to zero.

The type is described as follows:

31 12	11 7	6 0
imm[31:12]	rd	opcode

UJ-type instructions

Similar to U-type instructions, but with a different encoding of the immediate.

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

Instruction definitions

Instructions can be grouped by functionalities, depending on what resources they use and what function they perform.

ALU instructions

ALU instructions perform arithmetic and logical operations on processor registers or immediate values.

R-type, register to register

The table below lists all the register-to-register ALU instructions. Note that the `opcode` is the same for all of them (0110011), and the operation is decided by the contents of `func3` and `func7`.

Instruction	opcode	func3	func7	*Description*
ADD	0110011	000	0000000	`x[rd] = x[rs1] + x[rs2]`
SUB	0110011	000	0100000	`x[rd] = x[rs1] - x[rs2]`
SLT	0110011	010	0000000	`x[rd] = (x[rs1] < x[rs2]) ? 1 : 0` (signed)
SLTU	0110011	011	0000000	`x[rd] = (x[rs1] < x[rs2]) ? 1 : 0` (unsigned)
XOR	0110011	100	0000000	`x[rd] = x[rs1] ^ x[rs2]`
SLL	0110011	001	0000000	`x[rd] = x[rs1] << x[rs2]` (logical left shift)
SRL	0110011	101	0000000	`x[rd] = x[rs1] >> x[rs2]` (logical right shift, zero-fill)
SRA	0110011	101	0100000	`x[rd] = x[rs1] >> x[rs2]` (arithmetic right shift, sign-extend)
OR	0110011	110	0000000	`x[rd] = x[rs1] | x[rs2]`
AND	0110011	111	0000000	`x[rd] = x[rs1] & x[rs2]`

I-type, immediate

The table below lists all the immediate ALU instructions. Note that the `opcode` is the same for all of them (0010011), and the operation is decided by the contents of `func3` and part of the immediate bits. Additionally, the lower 5 bits of the immediate are used to determine the shift amount `shamt` in the shift operations:

shamt = imm[4:0]

Note that this resembles an R-type instruction, where `shamt` is equivalent to `rs2`, and `imm[11:5]` to `func7`.

Instruction	opcode	func3	imm[11:5]	*Description*
ADDI	0010011	000	xxxxxxx	`x[rd] = x[rs1] + imm`
SLTI	0010011	010	xxxxxxx	`x[rd] = (x[rs1] < imm) ? 1 : 0` (signed)
SLTIU	0010011	011	xxxxxxx	`x[rd] = (x[rs1] < imm) ? 1 : 0` (unsigned)
XORI	0010011	100	xxxxxxx	`x[rd] = x[rs1] ^ imm`
ORI	0010011	110	xxxxxxx	`x[rd] = x[rs1] | imm`
ANDI	0010011	111	xxxxxxx	`x[rd] = x[rs1] & imm`

SLLI	0010011	001	0000000	`x[rd] = x[rs1] << shamt` (logical left shift)
SRLI	0010011	101	0000000	`x[rd] = x[rs1] >> shamt` (logical right shift, zero-fill)
SRAI	0010011	101	0100000	`x[rd] = x[rs1] >> shamt` (arithmetic right shift, sign-extend)

U-type, constant building

The table below lists the instructions used to build constants. They are not necessarily related to the ALU, but they are often used in combination with other ALU instructions to build constants.

Instruction	opcode	*Description*
LUI	0110111	`x[rd] = imm << 12`
AUIPC	0010111	`x[rd] = PC + (imm << 12)`

Note that these instructions do not have any `func3` or `func7` fields.

Memory access

Interactions for memory accesses can be divided into loads (I-type) and stores (S-type).

I-type, loads

The table below lists the load memory instructions. Note that the `opcode` is the same for all instructions, and the type of load is decided based on the value of `func3`

Instruction	*opcode*	*func3*	*Description*
LB	0000011	000	Load byte (sign-extend to 32 bits): `x[rd] = signext(MEM[rs1+imm][7:0])`
LH	0000011	001	Load halfword (16 bits, sign-extend): `x[rd] = signext (MEM[rs1+imm][15:0])`
LW	0000011	010	Load word (32 bits): `x[rd] = MEM[rs1+imm][31:0]`
LBU	0000011	100	Load byte unsigned (zero-extend to 32 bits): `x[rd] = MEM[rs1+imm][7:0]`
LHU	0000011	101	Load halfword unsigned (zero-extend): `x[rd] = MEM[rs1+imm][15:0]`

S-type, stores

The table below lists the store memory instructions.

Instruction	*opcode*	*funct3*	*Description*
SB	0100011	000	Store **byte**: `MEM[rs1+imm][7:0] = x[rs2][7:0]`
SH	0100011	001	Store **halfword** (16 bits): `MEM[rs1+imm][15:0] = x[rs2][15:0]`
SW	0100011	010	Store **word** (32 bits): `MEM[rs1+imm][31:0] = x[rs2][31:0]`

Control

Instructions are used to control the execution flow of the program by manipulating the program counter (`PC`). They can be further divided into three types.

U-type, conditional branches

The table below lists the conditional branch instructions. These instructions modify the program counter based on the result of a comparison.

Instruction	*opcode*	*func3*	*Description*
BEQ	1100011	000	Branch if equal: `if (x[rs1] == x[rs2]) PC += imm`
BNE	1100011	001	Branch if not equal: `if (x[rs1] != x[rs2]) PC += imm`
BLT	1100011	100	Branch if less than (signed): `if (x[rs1] < x[rs2]) PC += imm`
BGE	1100011	101	Branch if greater/equal (signed): `if (x[rs1] >= x[rs2]) PC += imm`
BLTU	1100011	110	Branch if less than (unsigned): `if (x[rs1] < x[rs2]) PC += imm`
BGEU	1100011	111	Branch if greater/equal (unsigned): `if (x[rs1] >= x[rs2]) PC += imm`

UJ-type, unconditional jump

There is only one unconditional jump instruction: **JAL** (jump and link). This instruction updates the program counter based on an immediate value and stores the current program counter value in a specified register. This can be used for function calls, where the program has to resume execution after the function call ends.

Instruction	*opcode*	*Description*

JAL	1101111	`x[rd] = PC + 4`
		`PC = PC + imm`

Note that the program counter is stored with the increment already applied.

I-type, indirect jump

There is only one indirect jump instruction: **JALR** (jump and link register). This instruction updates the program counter based on the content of a register (and an optional immediate offset).

Instruction	*opcode*	*func3*	*Description*

JALR	1100111	000	`x[rd] = PC + 4`
			`PC = (x[rs1] + imm) & ~0b11`

Note that RISC-V instructions are always 4-byte aligned, meaning valid instruction addresses must be multiples of 4. Therefore, the bitwise NOT of `0b11` (3 in decimal) ensures the lower 2 bits of the PC are forced to 0.

Others

There are other instructions in the RV32I base instruction set that are used for system calls and memory fences; these are opcodes 1110011 and 0001111. For this project, these instructions will be ignored. So, if detected, the CPU should continue with its normal execution, i.e., increment the program counter, and ignore the instruction.

For more information about the RISCV ISA, please refer to the following link: RISCV Specifications

Memory Organization

The RISC-V ISA defines 32 32-bit general-purpose registers. Additionally, it constrains the main memory by the size of the address bus and the byte-addressable requirements.

Address Bus Size

The address bus is 32 bits wide. This means the CPU can generate addresses in the range `0x00000000` – `0xFFFFFFFF`.

Addressable Unit

RISC-V is a byte-addressable architecture. Each address corresponds to a single byte (8 bits).

- **1 byte (8 bits):** directly accessible at any address (0,1,2,3…)
- **2 bytes (16 bits, half-word):** must be aligned to 2-byte boundaries (0,2,4,6…)
- **4 bytes (32 bits, word):** must be aligned to 4-byte boundaries (0,4,8,12…)

Addressing Modes

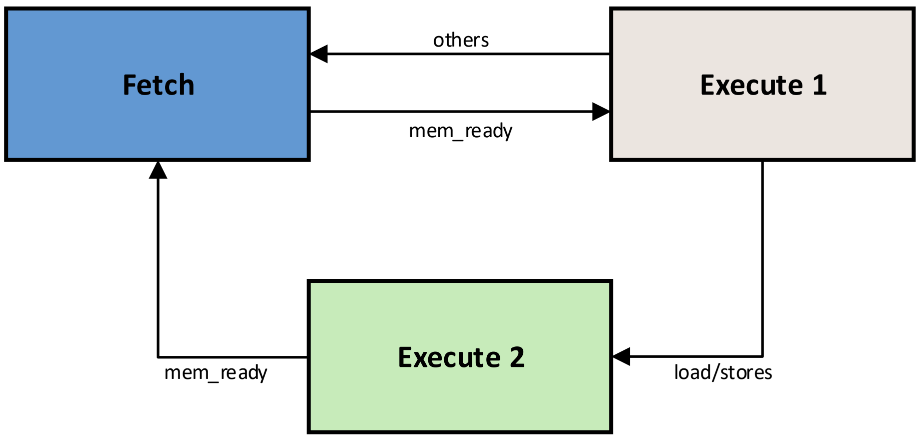
The addressing modes describe how memory locations are accessed. The RV32I memory access instructions define only *Register-indirect* memory accesses. This means that the accessed address is determined by the contents of one of the registers, `rs1`, plus an optional offset from the immediate value. There is an implicit direct memory access mode by reading from register 0, which is hardwired to zero, resulting in the immediate being the address.

Microarchitecture Design

Once the instruction set is defined, we can proceed with the microarchitectural design of the datapath and controller for the CPU. The CPU has a multi-cycle controller and features a Von Neumann architecture, where the same memory is used for both data and instructions.

Control State Machine

The controller will consist of 3 states: fetch-decode, execute-1, and execute-2.



During the fetch-decode state, the instruction is read from the memory. The value of PC determines the memory address. During this state, the controller should issue the necessary control signals to enable the reading of the SRAM. The instruction is stored in the instruction register and is decoded according to the instruction types on the next rising edge of the clock.

The execution state of an instruction is determined by its type. Memory load and store operations require two execution states, while other instructions typically require just one.

During execute-1, the operations dictated by the instruction are carried out. For memory load and store instructions, this state computes the address (usually calculated through `rs1+imm`). For other instruction types, the complete operation is performed within this single state.

During execute-2 for loads and stores, the data is loaded/stored in the memory based on the address computed in the previous instruction.

The update of the program counter should happen at the end of the execution state. For control instructions, the `PC` is updated based on the results of the conditions; for the other instructions, the `PC` is incremented by 4. Note that an increment of 4 is needed since the instructions are 32 bits (4 bytes).

Memory System

The memory system developed in Milestone 3 will be used to connect it to your CPU.

Tasks

1. Draw a schematic of the CPU datapath. To do this,
   - Place all the required combinational and sequential components in the datapath, including ALU, registers (PC, instruction register, register file, etc.), and all the necessary muxes.
   - Decide on the bussing structure interconnecting sequential and combinational components
   - Decide on all control signals to/from the controller for components and buses of the datapath
2. Draw a schematic of the CPU controller, with all control signals.
3. Draw a schematic of the top-level CPU, showing the connections between the datapath and the controller.
4. Write the SystemVerilog model of the datapath, strictly following the schematic diagram of the datapath.
5. Write the SystemVerilog model of the controller, strictly following the controller's FSM. Follow the Huffman coding style, considering two combination parts: one for issuing the next state, and the other for issuing the control signals.

   Caution: As consistently discussed in lectures, the controller merely coordinates and triggers actions by issuing appropriate control signals to the various parts of the datapath, but does not perform any computations or data operations itself.
6. Write a top-level module in SystemVerilog to connect the datapath to the controller.
7. Write a comprehensive testbench to test all instructions individually. For each instruction, you need to check that all the correct control signals are issued, and the instruction execution follows the expected behaviour based on the ISA definitions.

Deliverables

Use GitHub Classroom to submit your RTL and testbench files.

The GitHub Classroom repositories contain a skeleton of the RTL and testbench files for the CPU that you may use for your development.

Upload the following schematic diagrams to Canvas:

- Datapath
- Controller
- Top level

These schematics should include **all** signals, with the exception of clock and reset.