RISC-V Single-Cycle Processor

Milestone 2

Adham Ali - 900223243
Saif Abd Elfattah - 900225535

Submitted to Dr. Cherif Salama
This Project Milestone is prepared for CSCE3301, Section 1
Computer Science and Engineering Department
The American University in Cairo

October 28, 2024

Implementation

In our design of the RISC-V single-cycle processor, we focused on enhancing the control unit to accommodate the comprehensive RISC-V 42 instruction set. This expansion involved the integration of a variety of control signals, including `Branch`, `MemRead`, `MemtoReg`, `ALUOp`, `MemWrite`, `ALUsrc`, `RegWrite`, `auipc`, `jump`, `srcPC`, and `pcload`. Each signal plays a critical role in guiding the processor's operations, such as memory access and branching decisions. To effectively manage various branching instructions, we designed a dedicated branching module that processes these signals and incorporates new flags in the ALU to precisely identify the specific branch instruction to execute.

Top Module: Big

The `big` module represents the top-level architecture of a RISC-V single-cycle processor. Below are the key function calls and their roles:

```
PC (32) programCounter (clk, rst, PC1, adrn, adrnow);
```

This instantiates the program counter, which manages the address of the current instruction based on the clock and reset signals.

```
InstMem instructionMemory(adrnow[31:0], Ins);
```

This function call fetches the instruction from memory at the address specified by `adrnow` and stores it in `Ins`.

```
control Control(Ins[6:2], branch, memRead, memToReg, aluop, memWrite, aluSrc,
regWrite, auipc, Jumps, pcSelect, pcLoad, immm);
```

The control unit is instantiated here, generating control signals based on the opcode extracted from the instruction.

```
ALUControl aluControlUnit(aluop, Ins[14:12], Ins[30], aluSel);
```

This call determines the specific ALU operation to perform based on the ALU operation code and function fields from the instruction.

```
ImmGen immediateGenerator(Ins, immediate);
```

This function extracts the immediate value from the instruction for use in calculations.

```
DataMemory memoryUnit(clk, memRead, memWrite, aluResult[7:0], readData2, readData,
Ins[14:12]);
```

This function handles reading from and writing to the data memory based on the control signals.

```
registerFile regFile(clk, rst, rs1, rs2, rd, regWrite, readDataDest, readData1,
readData2);
```

This call manages the register file, allowing for read and write operations based on the instruction.

```
ALU ALU(aluSel, aluInput2[4:0], readData1, aluInput2, aluResult, zeroF, carryF,
overflowF, signF);
```

This instantiates the ALU to perform arithmetic and logic operations based on the selected ALU function.

```
branching branches(zeroF, overflowF, signF, carryF, Ins[14:12], branch, PCsel);
```

This function determines whether a branch instruction should be taken based on the flags produced by the ALU.

`MUXF branchPCSelector(resAddress, pcSelectAddress, aluResult, 32'd0, pcSelect, adrn);`

This multiplexer selects the next address for the program counter based on the control signals and the results from the ALU.

R-type Instructions

For R-type instructions, The control unit evaluates each instruction's opcode to determine the appropriate control signals, including `ALUOp`, which is then relayed to the `ALUControl` unit. Based on the values of `ALUOp`, `funct3`, and bit 30 of the instruction, `ALUControl` can accurately identify which ALU operation is necessary. Additionally, we introduced a specialized shift module to manage shift instructions effectively. This module utilizes the `alu_function` to differentiate among various shift operations while considering the shift amount and the target value.

I-type Instructions

I-type instructions make use of the same `ALUOp` signals to facilitate the selection of ALU operations. However, a key distinction for I-type instructions lies in the second ALU input, which is derived from the output of the immediate generator. A subset of I-type instructions includes load instructions, which utilize the `MemRead` signal alongside `fc3` to determine the specific load operation. After executing this operation, the result is stored in the designated target register.

For control flow modifications, the `jalr` instruction is particularly important as it alters the program counter (PC). It accomplishes this by employing the `jump`, `pcl`, and `PCs` signals, ultimately saving the modified PC value to a specified register. Additionally, instructions like `ECALL` and `FENCE` direct the PC to zero by adjusting `PCs`, while the `EBREAK` instruction maintains the current PC value by setting `pcl` to zero.

S-type Instructions

The S-type instructions, which are responsible for storing data, utilize the control signals `MemWrite` and `fc3` to define the specific storage operation. When an S-type instruction is executed, it stores the specified value in memory using little-endian format, ensuring that the data is correctly formatted for retrieval. This precise handling of memory operations is crucial for maintaining the integrity of data within the processor.

U-type Instructions

In the case of `lui` instructions, the immediate value is processed with a 12-bit shift before being stored in the designated register. This operation is crucial for loading upper immediate values into registers. For `auipc` operations, the `auipc` signal facilitates the addition of the shifted immediate to the current PC, writing the result into the selected register. Given that multiple sources may write to a register—either from memory data or U-type instructions—a multiplexer is employed to select the final value that will be stored in the register. This selection process is driven by a control line that combines the `jump` and `auipc` signals.

B-type Instructions

Branch instructions are essential for controlling the flow of execution within programs. They rely on several important flags, including the `zerof`, `overflowf`, `signf`, and `carryf`. These flags are computed within the ALU by comparing values based on their equality or relative magnitudes. The branching module processes the `branch` control signal along with `fc3`, combining this information with the `jump` signal using an OR gate. A multiplexer (mux) is then utilized to select the next value for the PC. If the instruction specifies a branch or jump, the PC is updated to reflect the current address plus the shifted immediate value. Conversely, if no branch is taken, the PC simply increments by 4. A final mux determines the ultimate value of the PC by selecting from the previous mux's output for `jal` instructions, the ALU's output for `jalr`, or the current address plus 4 (or zero for `EBREAK`).

J-type Instructions

When `jal` is executed, it activates the `jump` signal and stores the address of `PC+4` in the destination register, ensuring that the return address is saved for subsequent operations. The PC is then adjusted by adding the immediate value associated with the `jal` instruction. Finally, this new PC value is processed through a multiplexer to finalize the next address for the program counter, ensuring seamless execution of control flow changes.

Conclusion

In summary, the design and implementation of the RISC-V single-cycle processor demonstrate a comprehensive approach to integrating various instruction types and control signals. By meticulously expanding the control unit and implementing dedicated modules for R-type, I-type, S-type, B-type, U-type, and J-type instructions, we have created a processor capable of efficiently executing a wide array of operations. The systematic management of control signals ensures that each instruction is processed accurately.

Datapath Diagram

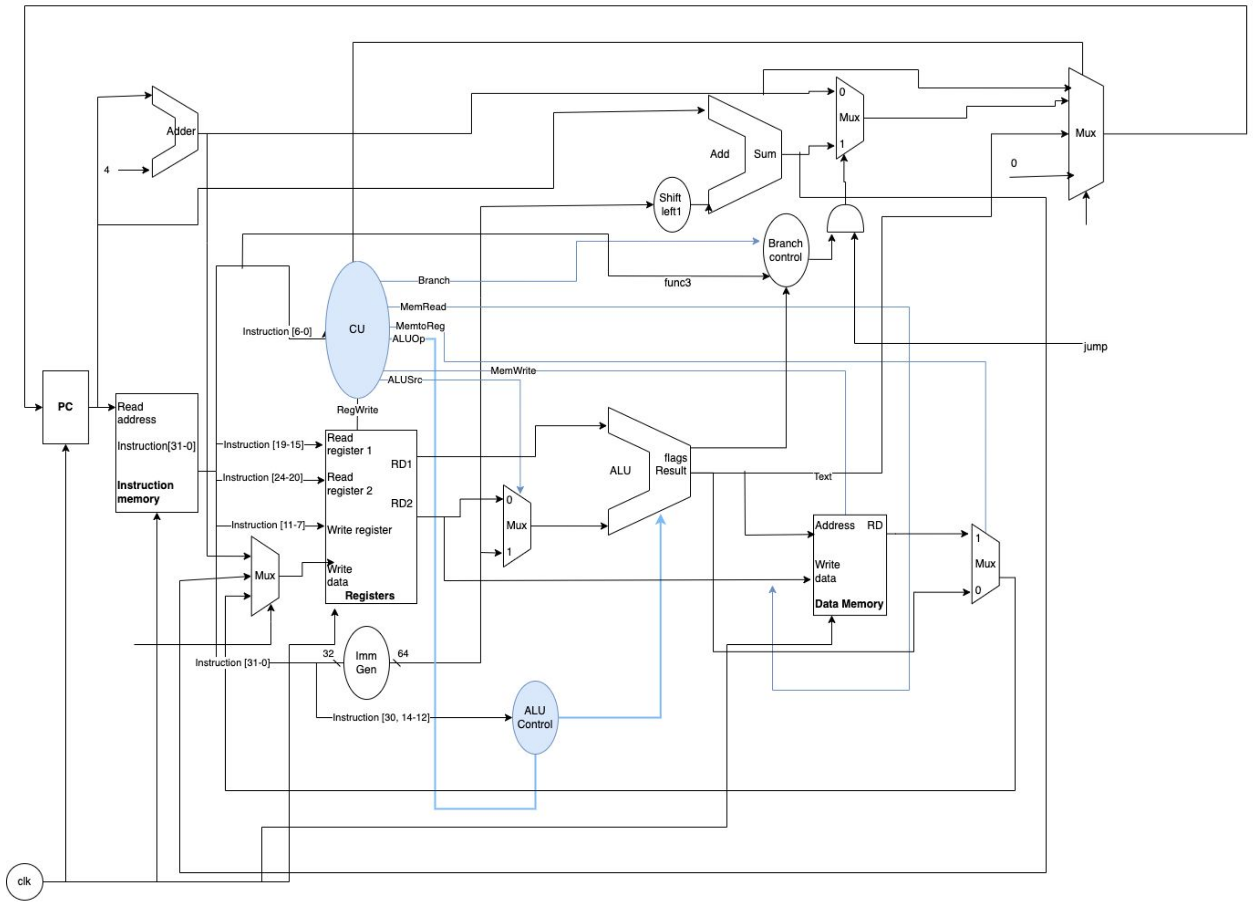

Figure 1: Datapath Diagram of the RISC-V Single-Cycle Processor