Pipelined **RISCV** processor

Giuliano Verrando

July 7, 2024

Abstract

The goal of the project was to simulate the functional behavior of a **RISCV** pipelined processor. In my implementation every atomic instruction takes 5 different clock cycles and the processor support a subset of the **RISCV** `ISA`. The subset is taken from the integer instructions in such a way to permit an implementation of a `bubble sort` algorithm.

Contents

1 Introduction **4**

2 Fetch **4**
- 2.1 Program counter . . . 4
- 2.2 Program memory . . . 5

3 Decode **6**
- 3.1 Register file . . . 7
- 3.2 Control unit . . . 7
- 3.3 Immediate generator . . . 7

4 Execute **8**
- 4.1 ALU . . . 8
- 4.2 ALU control unit . . . 8

5 Memory Access **9**
- 5.1 Data memory . . . 9

6 Write Back and top hierarchy processor **9**
- 6.1 Hazard detection unit . . . 9
  - 6.1.1 Data hazard . . . 10
  - 6.1.2 Control hazard . . . 10

7 Conclusion **12**

8 Appendix **12**
- 8.1 Prompt and reply to describe the PC . . . 12
- 8.2 Prompt and reply to describe the program memory . . . 13
- 8.3 Prompt and reply to describe the register file . . . 13
- 8.4 Prompt and reply to describe the control unit . . . 14

Acronyms

ALU	arithmetic logic unit. 6–10
CPU	central processing unit. 9, 16
FPGA	field programmable gate array. 4
HDU	hazard detection unit. 9, 10
ISA	instruction set architecture. 1, 4, 7, 8, 16
MUX	multiplexer. 8, 9, 16
PC	program counter. 4, 5, 8
RAM	random-access memory. 4, 5
ROM	read-only memory. 5, 9, 12

1 Introduction

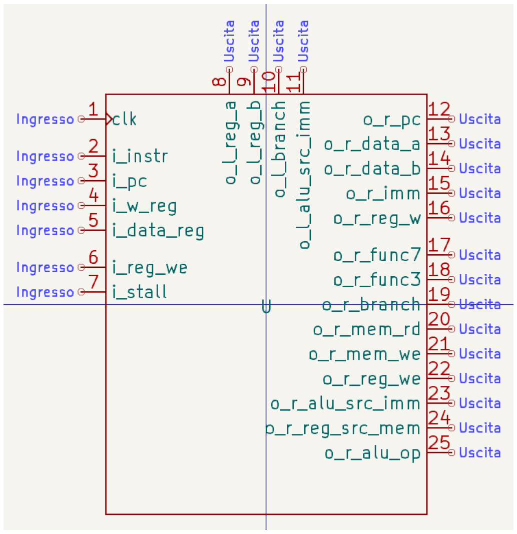
To accomplish the task of this project I decided to implement a subset of the **RISCV** `ISA` that could permit me to sort an array using the `bubble sort` algorithm. I designed the circuits using VHDL, a hardware description language used in the domain of the FPGA programming. I didn't synthesize the circuit, but I performed a functional simulation in which I used a clock signal of `ISA`10ns.

The subset that I decided to implement is summarized in this table:

Instruction	Type	ALU Operation	Name
`ld`	I-Type	sum	load
`sd`	S-Type	sum	store
`add`	R-Type	sum	add
`addi`	I-Type	sum	add immediate
`beq`	B-Type	subtract	branch if $=$
`bge`	B-Type	subtract	branch if $\geq$
`nop`	-	-	No Operation

2 Fetch

This stage is responsible for incrementing the **PC** to point the right instruction in the program memory. This instruction will be decoded and executed in the appropriate way during the following clock cycles.
The aim of this step is to 'choose' the right source for the **PC** register that has to increment during execution of instructions that does not modify the program flow, but must load the right value if instructions like `beq` or `bge` are in the program and if it is present some conditional execution. In case of stall must hold the same value and fetch the same instruction.



2.1 Program counter

The program counter (PC) is a fundamental component in the Fetch stage of a pipelined **RISCV** processor. Its primary aim is to hold the address of the next instruction to be executed. At the beginning of each instruction cycle, the Program Counter's value is used to fetch the instruction from memory. Once the instruction is fetched, the **PC** is incremented to point to the subsequent instruction in memory.

The update might simply be an increment by a constant value, or it might involve more complex operations, such as adding an offset for branch instructions. The ability to change the **PC** value dynamically is crucial for handling control flow changes, like jumps, branches, and subroutine calls, enabling the processor to execute instructions out of sequential order when required. This increment is by a fixed amount, corresponding to the size of each instruction in the instruction set architecture [Ope].

In my case my program **RAM** provided 32-bit data, so the length of the instructions defined by the the `ISA` themselves. This is a difference with the pure **RISCV** `ISA`, so **my implementation is not completely compliant with RISCV specifications**, I choose to not conform totally because I had to adapt the code even if I had used **RISCV** specifications because the linking process that concern with address calculations is related to the memory layout that in our case is different to the standard one that usually is present in true computer. So in my case it every instruction is far away from the nearest by 1 memory location.

In my implementation it takes in input three signals:

- `stall`, if asserted it should stop incrementing the **PC**
- `branch taken`, if asserted it means that the normal flow is interrupted and the program counter must take a specific value, not the last number incremented (as the default behavior should be)
- `next instruction`, this is the custom value that the **PC** must take if the normal flow is interrupted

and in output it determines two signals. They both represent the current value of the **PC**, but one it is stored in a register the other is in input to the program memory (2.2), this is useful because every step takes a clock cycle, and with the value stored it can calculate the new address during normal execution and transmit its value to the Decode phase (3) in sync with the instruction to execute.

My **PC** have the register initialized to `0xFFFFFFFFFFFFFFFF` so that this number is summed to the increment and (in my case is `0x0000000000000001`) the first instruction fetched will be the one placed at memory address `0x0000000000000000`.

2.2 Program memory

The program memory in a 5-stage pipelined **RISCV** processor primary aim is to store the instructions that the processor will execute. This memory is distinct from the data memory (5.1), which holds the data operands and results of instructions. The program memory is typically implemented as a read-only memory (**ROM**) or a non-volatile memory that retains the stored instructions even when the power is off, though it can also be implemented using **RAM** in certain designs.

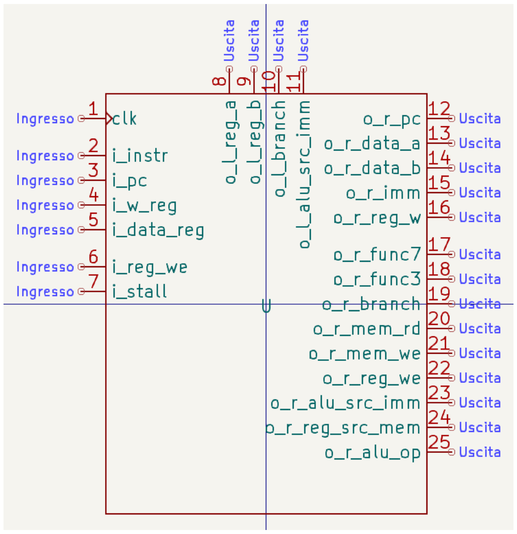
In the Fetch stage, the program memory works closely with the program counter (**PC**), which keeps track of the address of the next instruction to be executed. The Fetch stage retrieves or "fetches" the instruction stored at the address specified by the program counter. This involves the program counter sending the address to the program memory, which then outputs the instruction located at that address. This fetched instruction is then passed on to the next stage in the pipeline for decoding and execution [Ope].

In my case my memory took in input only the address to read and is read-only (so a **ROM**). It takes in input 64-bit input and provide 32-bit data (the instructions). It is a 1 KiB memory, so the address can assume values $\in$ [`0x00` : `0xFF`], if it is more than the maximum, it saturates the value to `0xFF` and return the data located at this very last valid address.

Composition

The fetch stage is composed by these two components and so takes in input the signals that determines the value of the **PC** as described in 2.1. After a clock cycle it provides the instruction and his associated address. The memory is linked with the **PC** before its value is placed in the associated register, so that the **ROM** can provide the instruction in sync with the **PC**.

3 Decode



This stage is responsible for interpreting the instruction and assert the right control signals to choose the operation to perform on the data and his flow in and out the registers and the data memory. Another important thing that happens during this stage is the reading of the register of the **register file** (3.1). The data is then forwarded to the following stage.

The control lines that are asserted are:

- `branch`, this line is asserted if the instruction is a *branch instruction* (that can modify the program flow) for example `beq`
- `memory write`, this line is asserted if the data memory has to be written (`sd`)
- `memory read`, this line is asserted if the data memory has to be read (`ld`)
- `register write`, this line is asserted if the register has to be written, for example `addi`
- `ALU source`, this line is asserted if the source of the second operand to the **ALU** has to be the immediate calculated by the immediate generator (3.3) (like in the case of `addi`) or a selected register (like in the case of `add`)
- `register source`, this line is asserted if the register has to be overwritten and the source of his new value has to be the data memory (`ld`). If it is not asserted the source of the value has to be the result of the **ALU** calculation (for example like in `add`)
- `ALU operation`, `func7` and `func3` are forwarded to the **ALU control unit** (4.2) that determine precisely the operation that the **ALU** must perform on the data.

This step takes as input also the `stall` signal. If this signal is asserted the control signals are forced to be inactive (in my case ='`0`') in this way the instruction that is decoded act like as a `nop`.

The data that is forwarded by this stage is

- `pc`, this is the address of the instruction decoded. With this number in the following stage is calculated the new address if there is a branch operation
- `data a`, this is the data that was stored in the first register decoded. The index of the register is encoded in the instruction.
- `data b`, this is the data that was stored in the second register decoded. The index of the register is encoded in some instructions
- `immediate`, this is a number extended by the immediate generator (3.3) that is encoded in some instructions
- `register to write`, this is the index of the register that has to be overwritten
- `register a`, this is the register from which the first operand of the **ALU** is taken. This index is immediately forwarded to the **hazard detection unit** (6.1) to check for any data hazard
- `register b`, this is the same of the last but for the second register

3.1 Register file

The register file in a **RISCV** processor primary aim is to store and provide access to the processor's general-purpose registers, which are used to hold intermediate data and addresses during instruction execution.

The register file consists of an array of registers, typically 32 in the case of a standard **RISCV** implementation, each capable of holding a fixed amount of data, such as 32 or 64 bits. When an instruction is fetched and decoded, the register file is accessed to retrieve the values stored in the source registers specified by the instruction. These values are needed for the execution of various operations, such as arithmetic computations, logical operations, and address calculations. In addition to reading data, the register file also supports writing data back to a register. This write operation occurs in the write-back stage (6) of the pipeline, where the result of an instruction is written back into a designated destination register [Ope].

My implementation deals only with 64-bit data, so every register of the 32 that are present, is 8-byte wide. As specified in the `ISA`, the register with index 0 (`x0`) is hardwired to the value `0x0000000000000000`. This register file firstly read the data and then if is asserted the right pin, a register is overwritten.

3.2 Control unit

The control unit in a **RISCV** processor plays a critical role in coordinating the operations of the processor by generating the necessary control signals based on the instruction being executed. Its primary aim is to interpret the fetched instruction and generate the appropriate signals to direct the activities of other components within the processor, ensuring that each instruction is executed correctly.

In the decode stage, the control unit reads the instruction, which contains the operation code (opcode). By analyzing this code, the control unit determines the nature of the operation to be performed, such as arithmetic, logic, memory access, or branch operations. Based on this determination, it generates control signals that guide the subsequent stages of the pipeline.

These control signals dictate the behavior of various parts of the processor, including the arithmetic logic unit (ALU), memory, register file, and multiplexers. For instance, the control unit will specify whether the **ALU** should perform an addition or subtraction, whether data should be read from or written to memory, and which registers should be accessed for reading or writing data [Ope].

This is how I asserted the signals depending on the instructions:

	`add`	`sd`	`ld`	`beq`, `bge`	`addi`	`nop` and unknown
`branch`	✗	✗	✗	✓	✗	✗
`memory read`	✗	✗	✓	✗	✗	✗
`memory write`	✗	✓	✗	✗	✗	✗
`register write`	✓	✗	✓	✗	✓	✗
`ALU source`	✗	✓	✓	✗	✓	✗
`register source`	✗	✗	✓	✗	✗	✗

3.3 Immediate generator

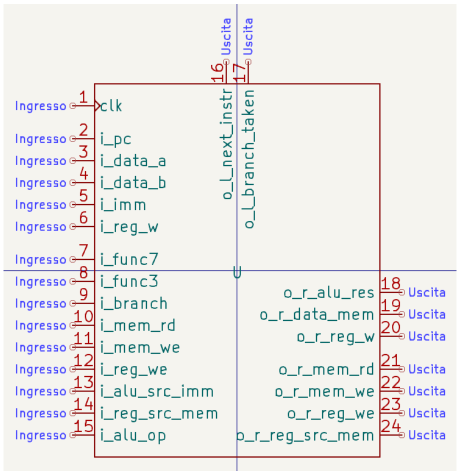
The aim of this component is to sign extend the immediate that is encoded in some instructions. This encoding for all the types that requires it and that I have implemented, specify the value with a 12-bit number, this immediate is than combined with the data in the register (for example to calculate an address for `ld` and `sd`) and must be a 64-bit number. For this reason this component enlarges the representation of the number according to his sign. To perform this operation it takes in input the whole instruction, it extracts the `opcode` and read the number accordingly.

Composition

The Decode input takes the whole instruction and split his different fields. In particular, it takes the `opcode` and send it to the Immediate Generator Unit and to the Control Unit. The Control Unit set the right control signals if the `stall` signal is not asserted. The Immediate Generator takes the same `opcode` and extracts the immediate of 12-bit, then sign-extend it.

4 Execute

This stage is responsible for implementing the right operation on the right operands and so perform the required calculation. It takes in input from the previous step the data from the two selected registers (the second one will not be used for I-Type instructions). It takes the immediate that will be used if the operation is not an R-Type instruction. In input there is also two other signals that for some instructions complete the `opcode`; these are `func3` and `func7` that are forwarded to the ALU Control Unit (4.2). This unit will set the right value to a signal that determine precisely the operation that the **ALU** will perform. In my case the **ALU** can perform only additions and subtractions.



In this phase if the instruction is a branch instruction, the immediate is used for calculate a new address from the current **PC** using an independent full-adder. According to the `ISA`, the value of the immediate must be shifted by 1 bit to permit only jump to address with the same parity of the current address. In my implementation there is an $1 : 1$ relation between the addresses and the instructions; for this reason I choose to not conform to the `ISA` and do not perform this shift.

The outputs are the result of the **ALU** calculation, the data to store in the memory if the instruction is `sd` (that in this case is taken from the second register specified), and the control signals that determine if the data memory has to be written or read and if a specific register has to be overwritten, which it will be and if it will be overwritten with a data that was in memory or with the **ALU** result.

4.1 ALU

This is the part of the processor that implement the calculation between the operands determined. Usually this unit can support a wide variety of basic arithmetic operation like sums, subtractions, shifts and logic operations like bitwise AND and bitwise OR of the operands.

My implementation support only sum and subtraction. For performing this operation it uses a full-adder implemented as described by the section **Full Adder** of the Wikipedia page Adder (eletronics). This adder takes in input also a flag that determine if the operation is a true addition or if you want to add the second operand in his negative representation (so perform a subtraction). To get the negative representation the operand is negated bitwise and is added summed a +1. This is easy to implement because in the standard implementation the full-adder is a concatenation of 1 bit full adders that takes in input the previous carry and the two bits and add them. They provide the result of the sum and the carry for the subsequent adders. With this setup the carry of the first adder should be hardwired to ‘`0`’, but it is set to ‘`1`’ if the operation is a subsection and to have the negative representation of the second operand it is required to add the +1.

4.2 ALU control unit

This component takes in input the `func3` and `func7` fields and set the right `ALU operation` code. In my implementation it uses also the `func3` field to distinguish the `beq` and `bge` instructions (that have the same `opcode`).

I choose to delegate this unit to distinguish between the two because I wanted the Control Unit to only takes in input the `opcode`. With this idea in mind maybe the name ‘**ALU control unit**’ it is not accurate, because it determines other characteristics other than the operation to perform.

Composition

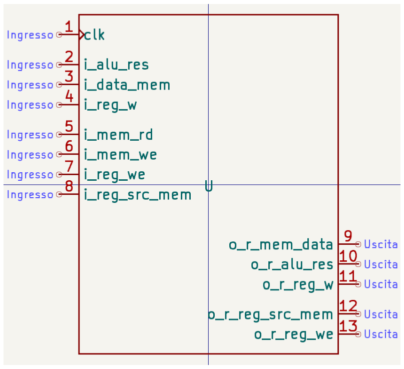
The data of the two register and the immediate are linked to the **ALU**. A 1-bit MUX determines if the immediate or the data from the second register must be used in the operation (this MUX is selected by the `ALU source` signal in input).

The program counter is summed to the immediate by an independent adder. The result will be the instruction address from which continue the execution if the instruction changes the normal program flow and a branch is taken. If this last is taken a signal is asserted. This signal will notify the HDU.

All the results are stored in a register with the exception of the `branch taken` signal and the signal with the value of the next instruction to execute. This is because a branch is taken at this stage of the pipeline, so there is no need to store the information and to forward it to the next stages.

5 Memory Access

This stage is responsible for reading the data memory if a `ld` instruction is executed, or to overwrite it with a `sd`. This stage takes in input the data to write to the memory, the **ALU** result (that if the instruction is a `ld` or a `sd` determine the location in memory to deal with) and all the control signals related to the register to overwrite if the instruction imply it.



5.1 Data memory

Every memory entry is initialized to `0x0000000000000000` (in my implementation the first 12 data entry are initialized to an array that I want to sort with the `bubble sort` algorithm that I have hard coded on the program **ROM**).

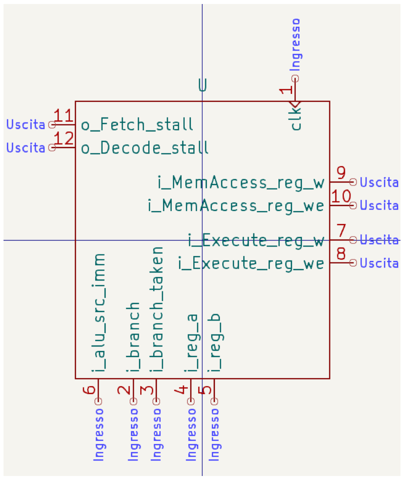
The memory takes a clock cycle to be written or read and saturates the address as explained for the program memory (2.2) and support 1 KiB.

6 Write Back and top hierarchy processor

Te Write Back stage is responsible for overwrite the register file that is placed inside the Decode entity. I didn't implement a whole entity, because the logic that is needed is only a 1-bit multiplexer (MUX) selected by the `register source` control signal that if asserted overwrite the register file with data read from the memory, otherwise with the **ALU** result. I implemented this logic into the top hierarchy component **CPU**.

6.1 Hazard detection unit

The last component that is present in the top hierarchy entity is the hazard detection unit (HDU) that is a component responsible for the detection of control hazards and hazards. If it detects these hazards, it asserts the **stall** signals for the Fetch and the Decode steps. Taking into account that these two different types of occurs requires a different management of the stalls, the HDU manage the two independently.
Let's analyze separately the two types of hazard:



6.1.1 Data hazard

This type of hazard take place if one of the registers used in the calculation by the current instruction is the same that the register that have to be written by an earlier command that didn't finish his pipeline drive-through. In this case we want that the processor stop the execution until the register is not overwrite and only then read that register and perform the following instruction.

It is possible to check which register an instruction requires in the Decode phase, before reading from the register file. If an earlier instruction require to write a register means that the `register write enable` in the Execute phase or in the Memory Access phase, will be asserted. If this is the case we want to stall the Fetch to continue fetching the same instruction but also neutralize his effect until the register is written. To neutralize his effect we stall the Decode phase. We need a register in the HDU because we have to wait a clock cycle after the hazard is present to have the time to complete the Write Back stage.

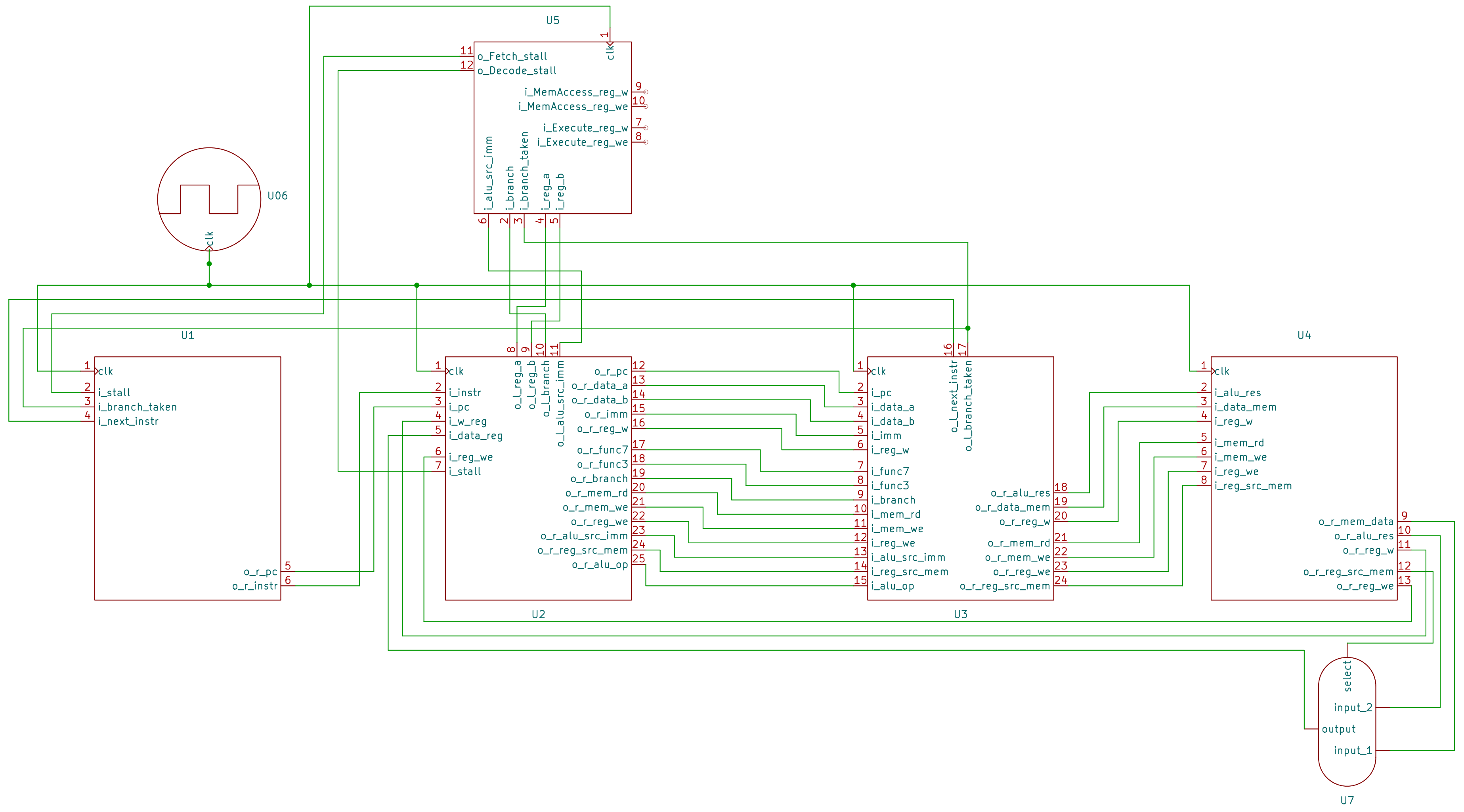
6.1.2 Control hazard

This type of hazard take place if a branch instruction occurs. In this case we want to wait until we don't know the **ALU** result that determine if the branch is taken. If it isn't, we want to continue the execution, if it is, we want to neutralize the effect of the next instruction until we did the jump and the correct instruction is fetched.

This means that to check if a control hazard occurs we have to check only the type of the instruction right after the instruction is decoded. A difference with the previous hazard is that we don't have to wait that an earlier instruction complete his pipeline, so we stall for a fixed amount of clock cycles (in my implementation 1 is sufficient), so we use a register to determine when the stalloccurred. During the stall we have to neutralize the effect of the branch instruction itself to not stall forever, and lastly we have to stall for one more clock cycle the Decode phase if the branch is taken because we don't want to execute the instruction that follows the branch, but we fetched it during the Execution step in which we calculate also the value of the next instruction.

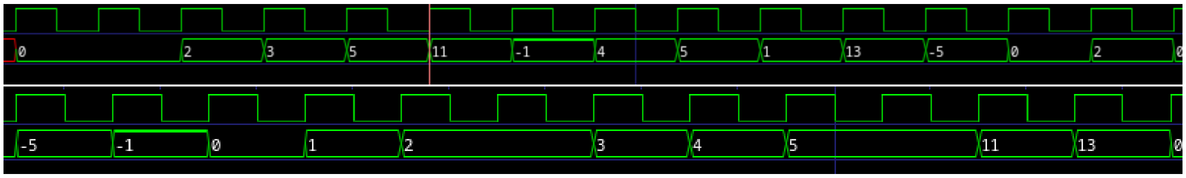
Composition

Finally, I connected the subsequent steps. The Decode and Execute steps and also connected to the HDU as explained in 6.1. The Write Back step consists in the connection between the Memory Access step and the register file in the Decode. Lastly the Execute after calculating the new instruction in case of a taken branch forward it to the Fetch phase. The HDU is connected to Fetch and Decode to stall them.


U5
1 clk
11 o_Fetch_stall
12 o_Decode_stall
9 i_MemAccess_reg_w
10 i_MemAccess_reg_we
7 i_Execute_reg_w
8 i_Execute_reg_we
6 i_alu_src_imm
2 i_branch
3 i_branch_taken
4 i_reg_a
5 i_reg_b
U06
clk
U1
1 clk
2 i_stall
3 i_branch_taken
4 i_next_instr
5 o_r_pc
6 o_r_instr
U2
1 clk
2 i_instr
3 i_pc
4 i_w_reg
5 i_data_reg
6 i_reg_we
7 i_stall
8 o_l_reg_a
9 o_l_reg_b
10 o_l_branch
11 o_l_alu_src_imm
12 o_r_pc
13 o_r_data_a
14 o_r_data_b
15 o_r_imm
16 o_r_reg_w
17 o_r_func7
18 o_r_func3
19 o_r_branch
20 o_r_mem_rd
21 o_r_mem_we
22 o_r_reg_we
23 o_r_alu_src_imm
24 o_r_reg_src_mem
25 o_r_alu_op
U3
1 clk
2 i_pc
3 i_data_a
4 i_data_b
5 i_imm
6 i_reg_w
7 i_func7
8 i_func3
9 i_branch
10 i_mem_rd
11 i_mem_we
12 i_reg_we
13 i_alu_src_imm
14 i_reg_src_mem
15 i_alu_op
16 o_l_next_instr
17 o_l_branch_taken
18 o_r_alu_res
19 o_r_data_mem
20 o_r_reg_w
21 o_r_mem_rd
22 o_r_mem_we
23 o_r_reg_we
24 o_r_reg_src_mem
U4
1 clk
2 i_alu_res
3 i_data_mem
4 i_reg_w
5 i_mem_rd
6 i_mem_we
7 i_reg_we
8 i_reg_src_mem
9 o_r_mem_data
10 o_r_alu_res
11 o_r_reg_w
12 o_r_reg_src_mem
13 o_r_reg_we
U7
select
input_2
output
input_1

7 Conclusion

I firstly programmed in `C` the `bubble sort`, then I cross-compiled it for **RISCV**. After that looked at the assembly code, I tried to simplify the instructions used and hard coded the new program translated in machine code into the program **ROM**. I initialized the data ram with an array and I simulated the functional behavior of the processor. I also added some instruction to load the data of the array in the data ram into the `x0` register at the beginning and end of the `bubble sort`, so that analyzing the data of the `data to write on register` signal I would see the unsorted array and [hopefully] the sorted one at the end. The simulation of the circuit was possible thanks to the ghdl software, and the wave visualization with GTKWave. I include the screenshot of the signal before and after the bubble sort:



8 Appendix

To retrieve this answers from `Chat GPT` I didn't provide more context than the one present in the prompts. I used clean and independent chat stateless sessions.

8.1 Prompt and reply to describe the PC

I am doing a project for which I am implementing a 5 stage pipelined RISCV processor in VHDL, my first step is the Fetch step. In this step there is the Program Counter. Can you describe the aim and the general functioning of a Program Counter just using text, no code and no lists? I have to add it to the report that I am writing.

```
The Program Counter (PC) is a fundamental component in the Fetch
stage of a pipelined RISCV processor.
Its primary aim is to hold the address of the next instruction to be executed.
At the beginning of each instruction cycle, the Program Counter's value is used
to fetch the instruction from memory.
Once the instruction is fetched, the PC is typically incremented to point
to the subsequent instruction in memory.
This increment is usually by a fixed amount,
corresponding to the size of each instruction
in the instruction set architecture.

In more detail, during the Fetch stage, the PC's current value is sent to memory
to retrieve the instruction located at that address.
After this, the PC is updated to reflect the address of the next instruction.
This update might simply be an increment by a constant value,
or it might involve more complex operations,
such as adding an offset for branch instructions.
The ability to change the PC value dynamically
is crucial for handling control flow changes,
like jumps, branches, and subroutine calls,
enabling the processor to execute instructions
out of sequential order when required.

The PC operates in a cycle, ensuring a continuous flow of instruction addresses,
which drives the overall execution process of the processor.
Its design ensures that the processor can systematically and efficiently
fetch and execute a sequence of instructions,
```

while also providing flexibility for complex control flows in the program execution.

8.2 Prompt and reply to describe the program memory

I am doing a project for which I am implementing a 5 stage pipelined RISCV processor in VHDL, my first step is the Fetch step. In this step there is the Program Memory (different from the data memory). Can you describe the aim and the general functioning of a program memory just using text, no code and no lists? I have to add it to the report that I am writing.

The program memory in a 5-stage pipelined RISC-V processor serves a critical role in the Fetch stage, where its primary aim is to store the instructions that the processor will execute. This memory is distinct from the data memory, which holds the data operands and results of instructions. The program memory is typically implemented as a read-only memory (ROM) or a non-volatile memory that retains the stored instructions even when the power is off, though it can also be implemented using RAM in certain designs for flexibility during development.

In the Fetch stage, the program memory works closely with the program counter (PC), which keeps track of the address of the next instruction to be executed. The Fetch stage retrieves or "fetches" the instruction stored at the address specified by the program counter. This involves the program counter sending the address to the program memory, which then outputs the instruction located at that address. This fetched instruction is then passed on to the next stage in the pipeline for decoding and execution.

The efficiency and speed of the program memory are crucial, as the Fetch stage sets the pace for the entire instruction execution pipeline. If the program memory access is slow, it can create bottlenecks that degrade the overall performance of the processor. To mitigate this, program memories are often designed with fast access times and may utilize caching mechanisms to further enhance performance.

Overall, the program memory's functionality is foundational to the operation of a pipelined processor, ensuring that instructions are promptly and accurately fetched, allowing for smooth and efficient instruction execution throughout the pipeline stages.

8.3 Prompt and reply to describe the register file

I am doing a project for which I am implementing a 5 stage pipelined RISCV processor in VHDL, my second step is the Decode step. In this step there is the Register File. Can you describe the aim and the general functioning of a register file just using text, no code and no lists? I have to add it to the report that I am writing.

The register file in a RISC-V processor is a crucial component within the decode stage of the pipeline. Its primary aim is to store and provide rapid access to the processor's general-purpose registers, which are used to hold intermediate data and addresses during instruction execution.

The register file consists of an array of registers, typically 32 in the case of a standard RISC-V implementation, each capable of holding a fixed amount of data, such as 32 or 64 bits. When an instruction is fetched and decoded, the register file is accessed to retrieve the values stored in the source registers specified by the instruction. This step is essential because these values are needed for the execution of various operations, such as arithmetic computations, logical operations, and address calculations.

During the decode stage, the instruction is examined to determine which registers are being referenced. The register file then reads the values from the specified source registers and makes them available for the subsequent execution stage. In addition to reading data, the register file also supports writing data back to a register. This writes operation typically occurs in the write-back stage of the pipeline, where the result of an instruction is written back into a designated destination register.

The register file is designed for fast access to support the high-speed requirements of the pipeline. It achieves this by using dual read ports and one write port, allowing two registers to be read simultaneously and one register to be written at the same time, which is a common requirement for most instructions. This parallelism is essential for maintaining the efficiency and throughput of the pipelined processor.

In summary, the register file serves as a high-speed storage and retrieval unit for the processor's registers, playing a vital role in ensuring that the necessary data is readily available for each instruction as it progresses through the pipeline. Its efficient design is fundamental to achieving the overall performance goals of the processor.

8.4 Prompt and reply to describe the control unit

I am doing a project for which I am implementing a 5 stage pipelined RISCV processor in VHDL, my second step is the Decode step. In this step there is the Control Unit. Can you describe the aim and the general functioning of a control unit just using text, no code and no lists? I have to add it to the report that I am writing.

The control unit in a RISC-V processor plays a critical role in coordinating the operations of the processor by generating the necessary control signals based on the instruction being executed. Its primary aim is to interpret the fetched instruction and generate the appropriate signals to direct the activities of other components within the processor, ensuring that each instruction is executed correctly and efficiently.

In the decode stage, the control unit reads the instruction, which contains the operation code (opcode) and, in some cases, additional function codes. By analyzing these codes, the control unit determines the nature of the operation to be performed, such as arithmetic, logic, memory access, or branch operations.

Based on this determination, it generates control signals that guide the subsequent stages of the pipeline.

These control signals dictate the behavior of various parts of the processor, including the arithmetic logic unit (ALU), memory, register file, and multiplexers. For instance, the control unit will specify whether the ALU should perform an addition or subtraction, whether data should be read from or written to memory, and which registers should be accessed for reading or writing data.

The control unit ensures that the correct data paths are selected, enabling the processor to handle different types of instructions in a streamlined manner. It orchestrates the timing of operations, managing the flow of data through the pipeline stages to prevent conflicts and ensure that each instruction progresses smoothly from fetch to execution to write-back.

In summary, the control unit is essential for decoding instructions and generating the necessary control signals to manage the processor's operations, enabling the processor to execute a wide variety of instructions efficiently and accurately.

Glossary

bubble sort Sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed [Wik24a]. 1, 4, 9, 12

clock cycle Also known as a machine cycle or a clock tick, is the basic unit of time in a computer's CPU and synchronous electronic. 1, 4, 5, 9, 10, 16

control signal In a CPU is a signal that select the path of the data in the different stages of execution of the instruction. They determine for example the operations performed by the different components and the selection of the MUXs. They can be viewed as the signals that hold and transport the internal data of the CPU . 6, 8, 9

hazard In the domain of central processing unit (CPU) design, hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle, and can potentially lead to incorrect computation results. Three common types of hazards are data hazards, structural hazards, and control hazards (branching hazards) [Wik24b]. 6, 9, 10, 16

linker In computing, a linker or link editor is a computer system program that takes one or more object files (generated by a compiler or an assembler) and combines them into a single executable file, library file, or another "object" file [Wik24c]. 16

linking The act of 'linking' intended as the action on the code performed by the linker. 4

multiplexer In electronics, a multiplexer (or mux; spelled sometimes as multiplexor), also known as a data selector, is a device that selects between several analog or digital input signals and forwards the selected input to a single output line. The selection is directed by a separate set of digital inputs known as select lines. A multiplexer of 2^n inputs has n n select lines, which are used to select which input line to send to the output [Wik24d] . 3, 9

RISCV RISC-V is an open standard `ISA`. 1, 4, 5, 7, 12

stall In the design of pipelined computer processors, a pipeline stall is a delay in execution of an instruction in order to resolve a hazard [Wik23]. 4, 10

References

[Ope] OpenAI. Chatgpt. [Large language model; accessed July, 2024; prompt list in 8].

[PH17] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.

[Wik23] Wikipedia. Pipeline stall, 2023. [Online; accessed July 3, 2024].

[Wik24a] Wikipedia. Bubble sort, 2024. [Online; accessed July 3, 2024].

[Wik24b] Wikipedia. Hazard (computer architecture), 2024. [Online; accessed July 3, 2024].

[Wik24c] Wikipedia. Linker (computing), 2024. [Online; accessed July 3, 2024].

[Wik24d] Wikipedia. Multiplexer, 2024. [Online; accessed July 5, 2024].