Tai Lung RV32I-MAF Project

MAS v1.3

(Micro Architecture Specification)

Authors:

Christian Shakkour, chrisshakkour@gmail.com

Shahar Dror, shdrth1@gmail.com

Git repository:

https://github.com/ChrisShakkour/RV32I-MAF-project

Release Information

DATE	AUTHOR	DESCRIPTION
02-FEB-2022	Chris	Created v1.0
05-MAR-2022	Shahar	Add pipe stages
17-MAR-2022	Shahar	Add data path
28-APR-2022	Shahar	Add block level

Acknowledgements

Special thanks to our supervisor Amihai Ben David for contributing knowledge and experience throughout the development of this project. Many thanks to the VLSI Lab at the Technion - Israel Institute of technology for supporting and providing tools and resources that were an integral part of the development process.

About this document

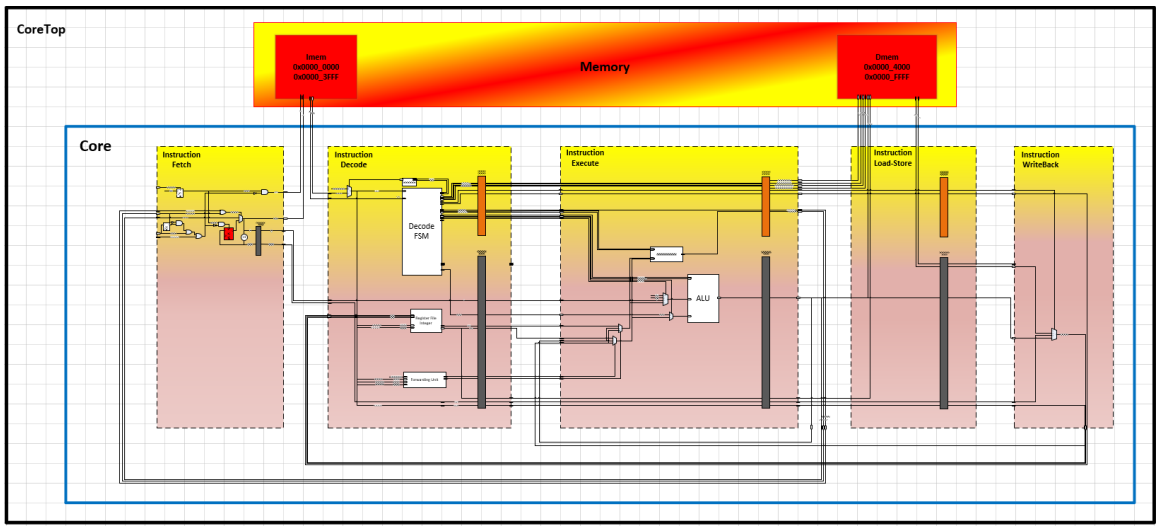
In this document you will find detailed information of Tai Lung Core architectural and RTL implementation.

Table of Contents

Introduction

This project features a basic RISC-V CPU called Tai-Lung. Tai-Lung project is a single Core,5 stage pipelined, RV32I CPU based on the RISC-V ISA. This document will describe the architecture of the Tai-Lung Core, The design and implementation of each of the pipe stages, the data, and the control paths.

Block diagram



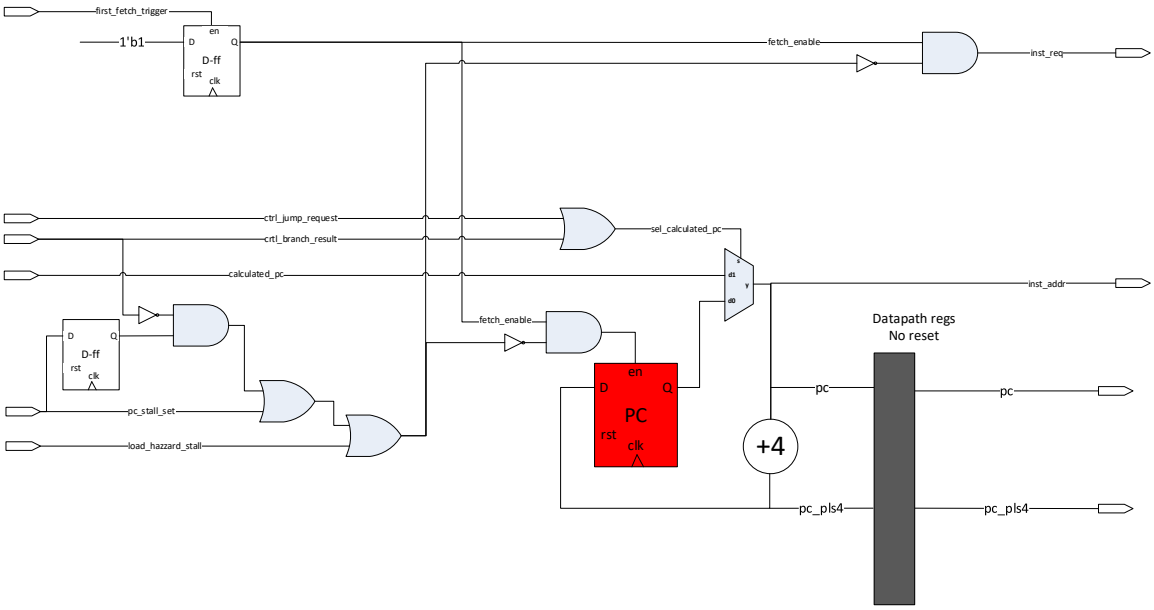
Pipeline stages

The Design is divided into 5 pipe stages, (1) Fetch where the desired command is brought from the instruction memory to the pipe. (2) Decode stage where the brought instruction is decoded. (3) execution stage where an arithmetic operation is done on 2 elements. (4) memory stage where data memory is brought into/out-of the execution pipe. (5) write-back stage where data is written back into the register file. Following deeper dive into the pipe stages.

Fetch

The "Fetch" stage responsibility is to keep track on the commands that should be pushed to the next stage of the pipe, The next command address is stored in a PC (Program Counter) register.

In every cycle the Instruction memory receives the next command address from the PC reg, while the PC of the next cycle is chosen between the following instruction or a different address due to a jump/branch instruction.



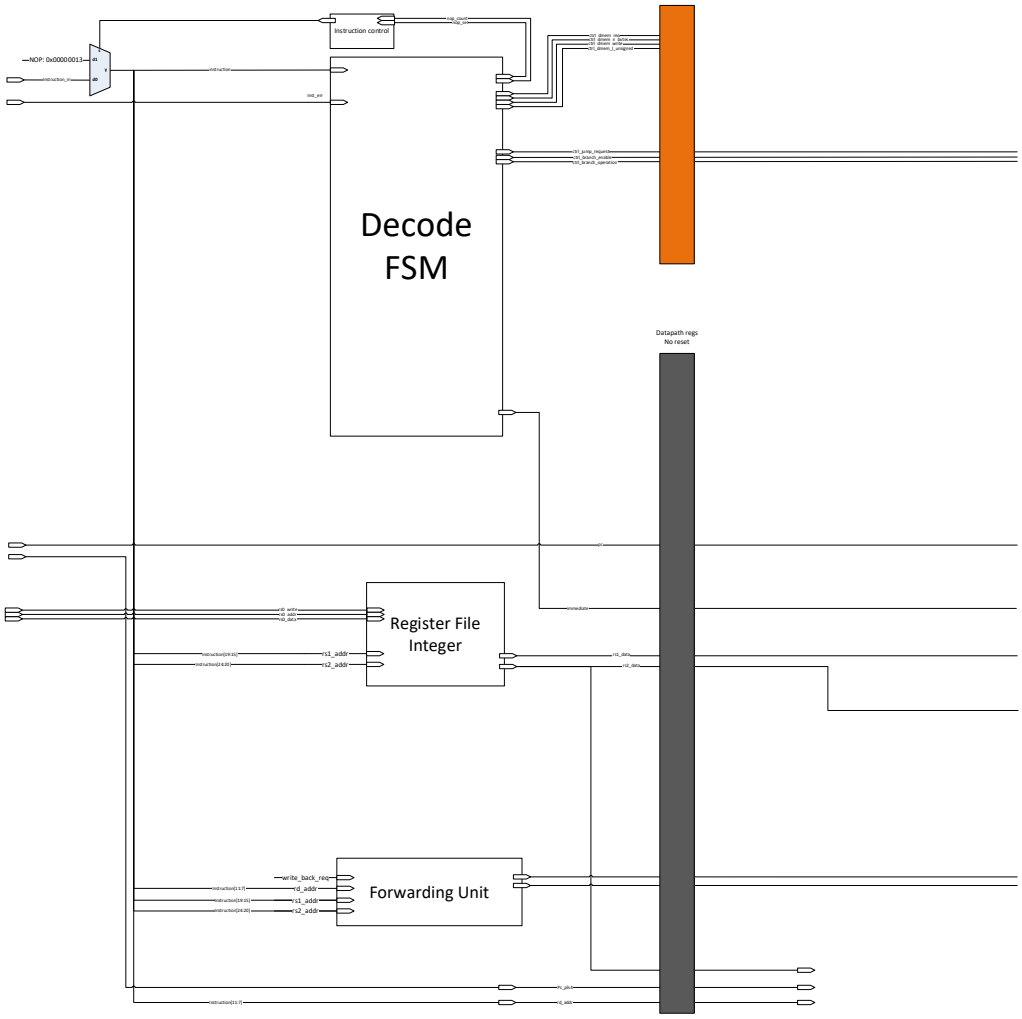
The updating of the PC can also be stalled in this stage to overcome hazard that caused by load instruction followed by instruction that used the loaded value.

"wake up" sequence:
For this stage start to fetch instruction first the reset need to be de-asserted,
and then "*first_fetch_trigger*" input have to be asserted for at least one pose-edge for the clock.

Decode

The Decode Stage decodes the instruction he received from the I_MEM to control signals that control the data flow of the core.
The register file is instantiated in The Decode stage, and at the same clock cycle he also bring the data from the registers that encoded in the instruction.

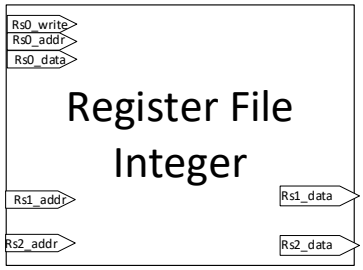
First the instruction type is determent by the instruction op_code (using the 7 LSB), then the rest of the instruction can be decoded by the instruction type format.



The decode stage includes a few building blocks, that are presented below.

Register-file

The Register file unit store 32 registers with size of 32bit. While register number 0 is not writable and have a constant value of 0. The Register file can write 1 new value to register at each clock cycle and read 2 registers values.



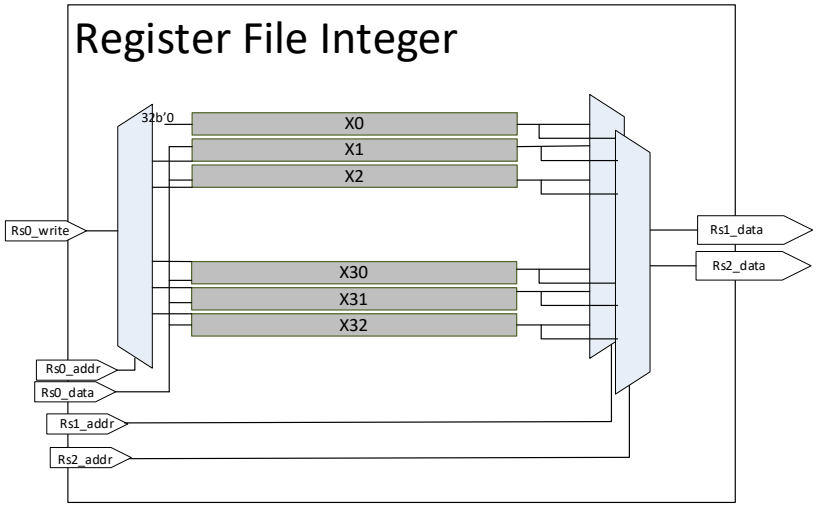
I/O:

Pin	Width	direction	description
Rs0_write	1	input	Write enable
Rs0_addr	5	input	Address of the register that will be written
Rs0_data	32	input	Data that will be written
Rs1_addr	5	Input	Address of rs1 register
Rs2_addr	5	Input	Address of rs1 register
Rs1_data	32	Output	Data stored in register "rs1_addr"
Rs2_data	32	Output	Data stored in register "rs2_addr"

Table 1: input/output pins

Implementation:

The Register file contains 32 registers with size of 32bit. A 32 inputs mux for each of the data output with rs_addr[1/2] as the selector. A 32 outputs decoder with rs0_addr as the selector, and rs0_write as the input, controlling the enable of the of the registers choosing the desired register to write, or not write at all.



Forwarding unit

The forwarding unit detects a data hazard in the pipe:
A read request from the register-file of a register that his updated value is not been wrote back to the register-file yet. After the detection of the pipe stage with the updated value the unit forwarding the data to the execution stage.

I/O:

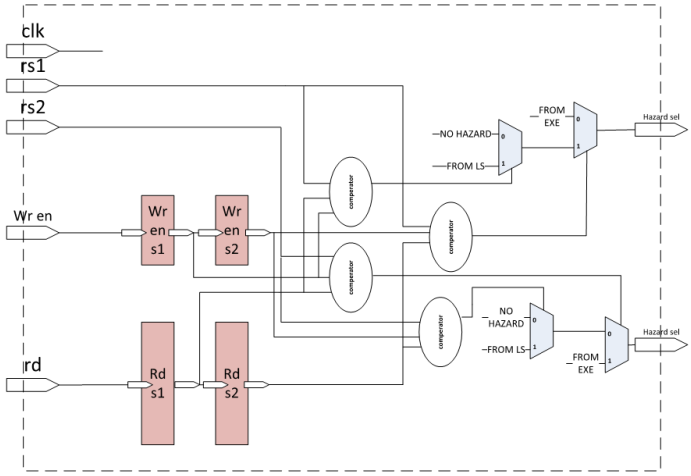
Pin	Width	direction	description
Rs1	5	input	Register address of read operation
Rs2	5	input	Register address of read operation
Wr_en	1	input	Write enable of rd register
rd	5	input	Distention register address
Hazard_sel1	2	output	Selector of the pipe stage with rs1 updated value
Hazard_sel2	2	output	Selector of the pipe stage with rs12 updated value

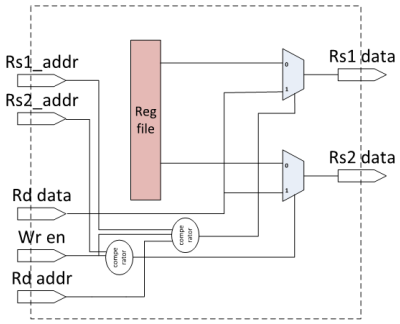
Table 2: input/output pins

Implementation:

The Forwarding unit implantation relies on comparation unit and muxes, and registers.

The destination registers and 1 bit that indicates if was there a write command, is stored in the unit until the data was wrote back to the register file.
The unit compare the RS1 and RS2 input to the rd address that stored inside the unit,
Those comparation unit output are a selector to a mux with a hard-coded value that indicates the pipe stage which we should forward the data from.





Load Hazard Unit

A load hazard unit, detects when a load hazard is expected, meaning a load instruction is followed by another instruction that needs the data loaded from the memory. This data is present only in the Writeback stage so 2 NOP's must be pushed, or the relevant pipe stages must be stalled to create a backpressure on the fetching unit. In our design one can use the forwarding unit from Whitbeck stage to ALU, hence only suffer one NOP instruction penalty.

I/O:

Pin	Width	direction	Description
load_req	1	input	Indicates a load command is in the decode stage
load_enable	1	input	module functional enable
rd	5	input	Destination register address
rsx_active	2	input	Source 0/1 active, which source needs attention
rs1	5	input	Source 1 register address
rs2	5	input	Source 2 register address
load_hazard_stall	1	output	initiate stall signal (load hazard expected)
nop_req	1	output	NOP request signal (load hazard expected)

Table 2: input/output pins

Implementation:

The implementation is simple. at every command the module takes the previous control, and data signals that relate to a load command, if found that there was a load command the module compares the destination register address with both the sources registers address and injects a NOP command instead of the actual command, and a signal is sent to the Fetch unit to notify it of stalling the PC from progressing.

*In the verification. a Race Bug was found in this mechanism, will be fixed in the next project.

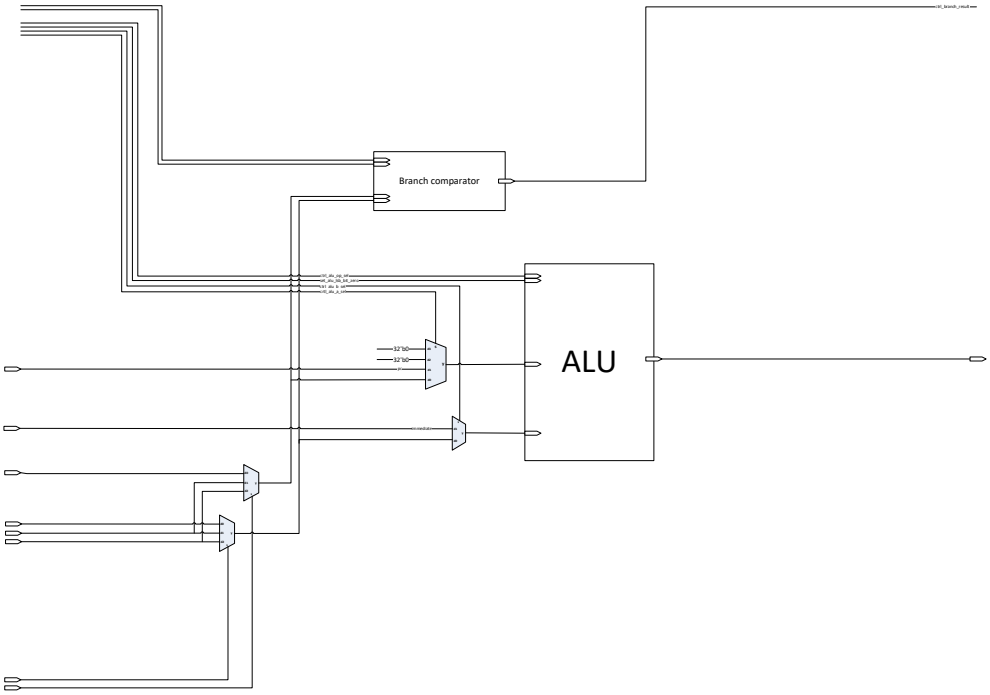
Execute

The execute main component is the ALU (Arithmetic Logic Unit) which support all the basic arithmetic operation between 2 integers with size of 32 bit.
Addition, subtraction, set lest then (unsigned), bitwise or/and/xor, shift right logic/arithmetic, shift left logic.

The 2 operands are chosen in respect to the instruction from the register file, the instruction PC or an immediate.

To handle any data hazard if the operand is from the register file a mux is instantiated before the ALU that can chose a value that not yet been writing to the register file back.

A branch comparator is instantiated in the execute stage to support branch instructions and determine if a branch should be taken or not.



Following brief explanation of the internal blocks, ALU and Branch comparator

ALU

ALU performs arithmetic operation between 2 operands. The supported operations are: addition, subtraction, signed and unsigned comparation, bit wise xor, bit wise and, bit wise or, logic shift left, logic shift right, arithmetic shift right. A mux selects between the outputs of all operations to one output of the ALU.

I/O:

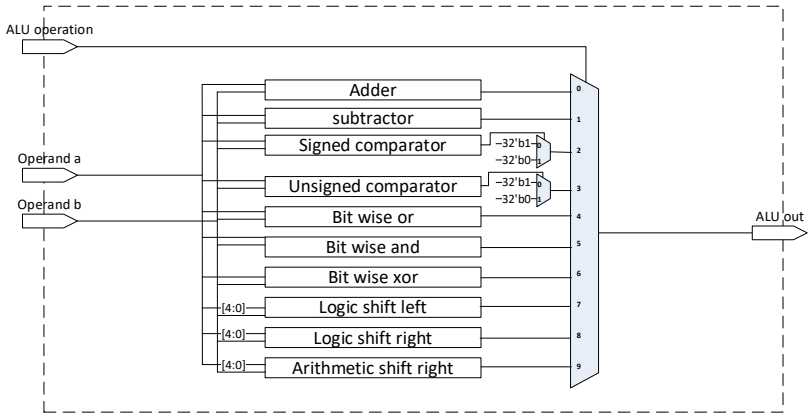
Pin	Width	direction	description
ALU operation	4	input	Selects the arithmetic operation
Operand a	32	input	Operand A
Operand b	32	input	Operand B
ALU out	32	output	Result of the selected operation

Table 3: input/output pins

Implementation:

The ALU contains 10 different arithmetic units the preforms single operation:
Addition while disregarding overflow, subtraction.
signed and unsigned comparation.
bit wise xor, bit wise and, bit wise or.
logic shift left, logic shift right, arithmetic shift right – while using only 5 ls-bists of operand b.

The outputs of all the arithmetic operation go through 10 inputs mux, and chose the ALU output based on the “ALU operation” input.



Branch Comparator

Figure 1: branch comparator symbol

Description:

A branch comparator unit compares between Two operands A and B and asserts if the Arithmetic comparison is true. This branch predictor supports BEQ, BNE, BLT, BLTU, BGE, BGEU commands. following explanation of these branch operations:

Compare operation	Description	Arithmetic operation
BEQ	branch equal	A == B
BNE	branch not equal	A != B
BLT	branch less than	A < B
BLTU	branch less than unsigned	unsigned(A) < unsigned(B)
BGE	branch greater equal	A >= B
BGEU	branch greater equal unsigned	unsigned(A) >= unsigned(B)

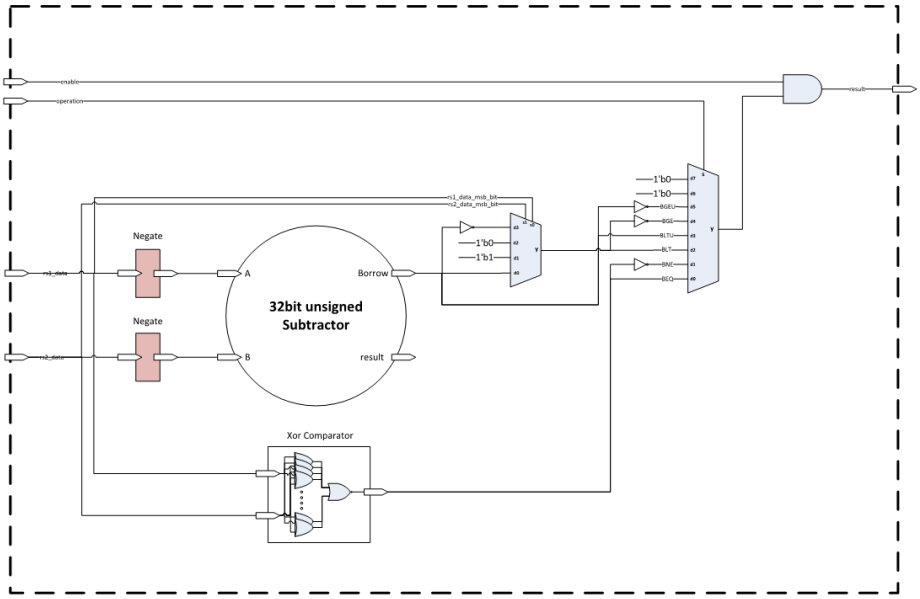
I/O:

Pin	Width	direction	description
enable	1	input	The result is propagated only when enable is asserted
operation	3	input	Selects the comparison operation (BEQ, BLT…)
rs1_data	32	input	Operand A
rs2_data	32	input	Operand B
result	1	output	branch compare result(TAKEN 1, or NOT TAKEN 0)

Implementation:

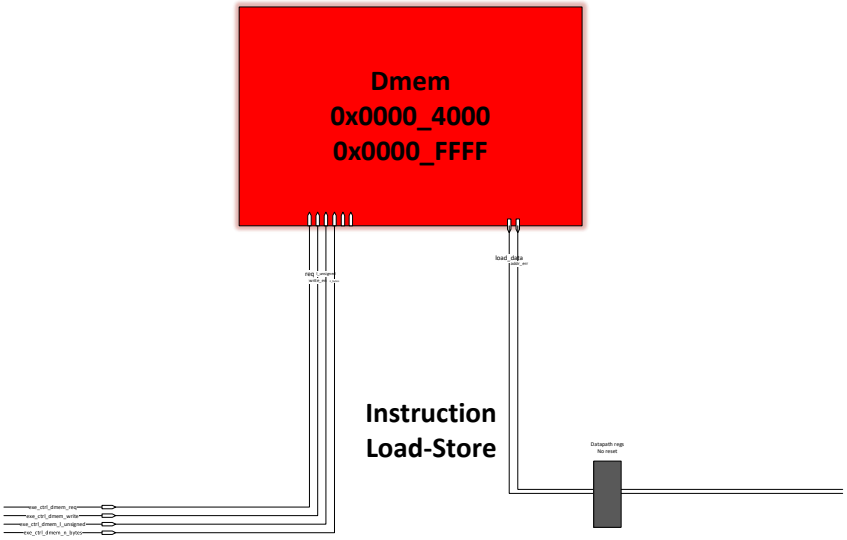
The implementation relies only on two main components, an Equal “==” comparison unit, and an unsigned subtraction unit, the Equal comparison unit design is simply a bitwise Xor between A and B. the outputs are then driven into a NOR gates to produce the result, if A and B are Equal the bitwise XOR result in Zero and after the NOR gate the result is asserted true. If A is not equal to B then the bitwise XOR result is non zero and the output is not asserted.

The unsigned subtractor is used only for it’s borrow signal, whenever the borrow is asserted it means the unsigned(A) is less than unsigned(B), with the same logic it can be concluded that unsigned(A) is greater equal to unsigned(B) only if the borrow signal is inverted. However, to also compare A and B in the same manner when one or both operands are signed one shall negate the signed operand and compare accordingly, the comparison result then must pass thru simple logic to compensate on the negation as follows in the design.


enable
operation
result
Negate
Negate
A
B
Borrow
result
32bit unsigned
Subtractor
1'b0
1'b1
1'b0
1'b0
BGEU
BGE
BLTU
BLT
BNZ
BEQ
Xor Comparator

Load/store

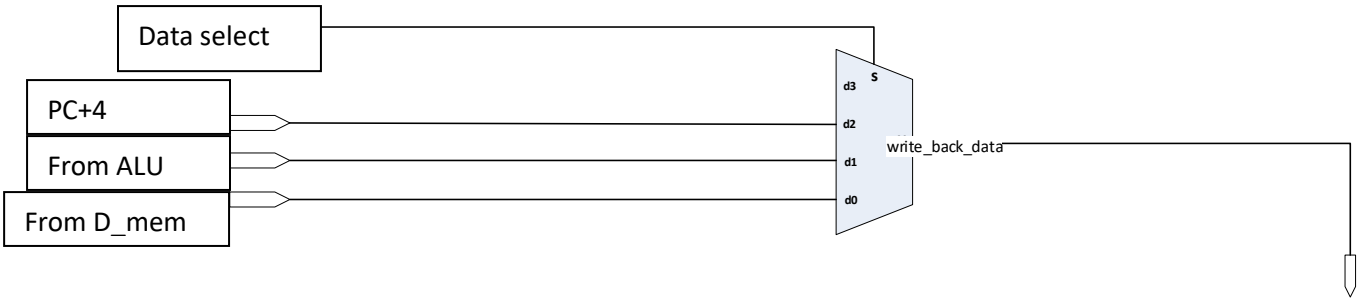
This stage does not contain combinatorial logic.
This stage sampled the values from the execute stage and transfer them to the memory for store instructions with the control signals. For load instructions this stage transfer the desired address, with the control signals, and the data will go from the memory at the next cycle to the write-back stage:



Write back

This stage responsible on managing the writes to the register file. For all instructions that require a value to be saved in a register. The desired data is selected and transferred to the register file, and the write enable signal is asserted.

For any instruction that does not require save a new value in a register the write enable signal is de-asserted.



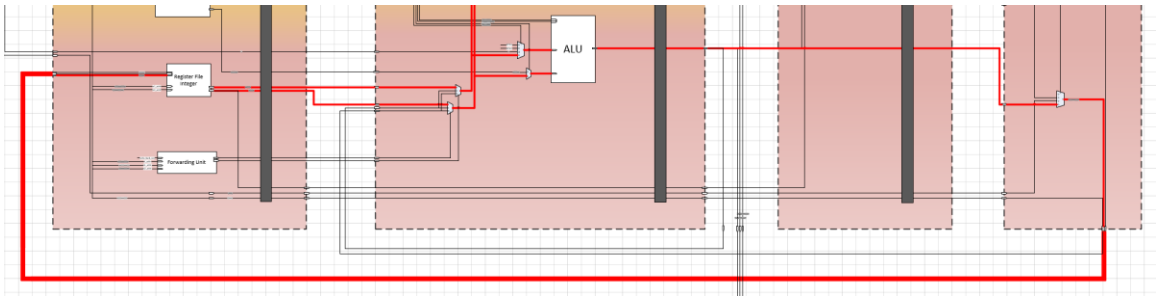
Datapath

RV32I instruction set supports 6 main opcode structures, R, I, S, J, U, B for different types of commands each with its unique execution Datapath, following the RISC-V official opcodes structure:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

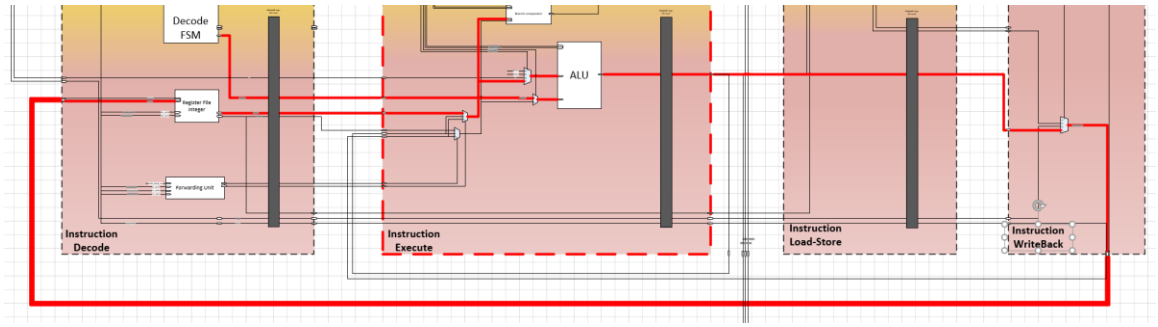
R-Type

R-Type instruction performs an arithmetic operation on 2 registers from the register file, and save the result to a register in the register file

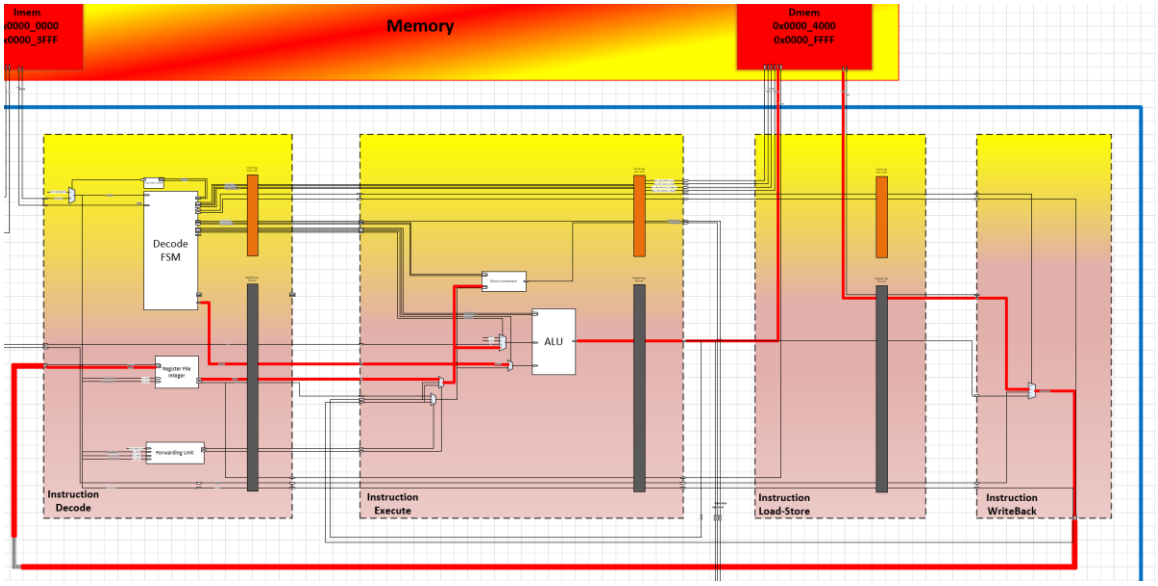


I-Type

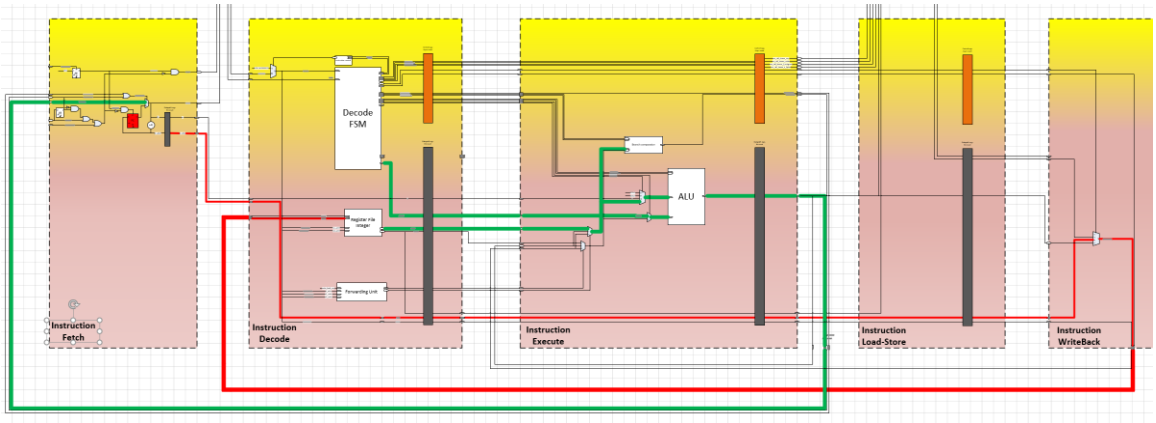
I-Type instruction can either, (1) perform an arithmetic operation on a register from the register file and an immediate value from the encoded instruction and save the result to a register in the register file as follows.



(2) Load a value from the Data memory to a register in the register file, the data size can be either a word (32 bit), half word (16 bit) or 8 bits. The effective address is obtained by adding the base address value from the register file to an offset that encoded as immediate in the instruction as follows.

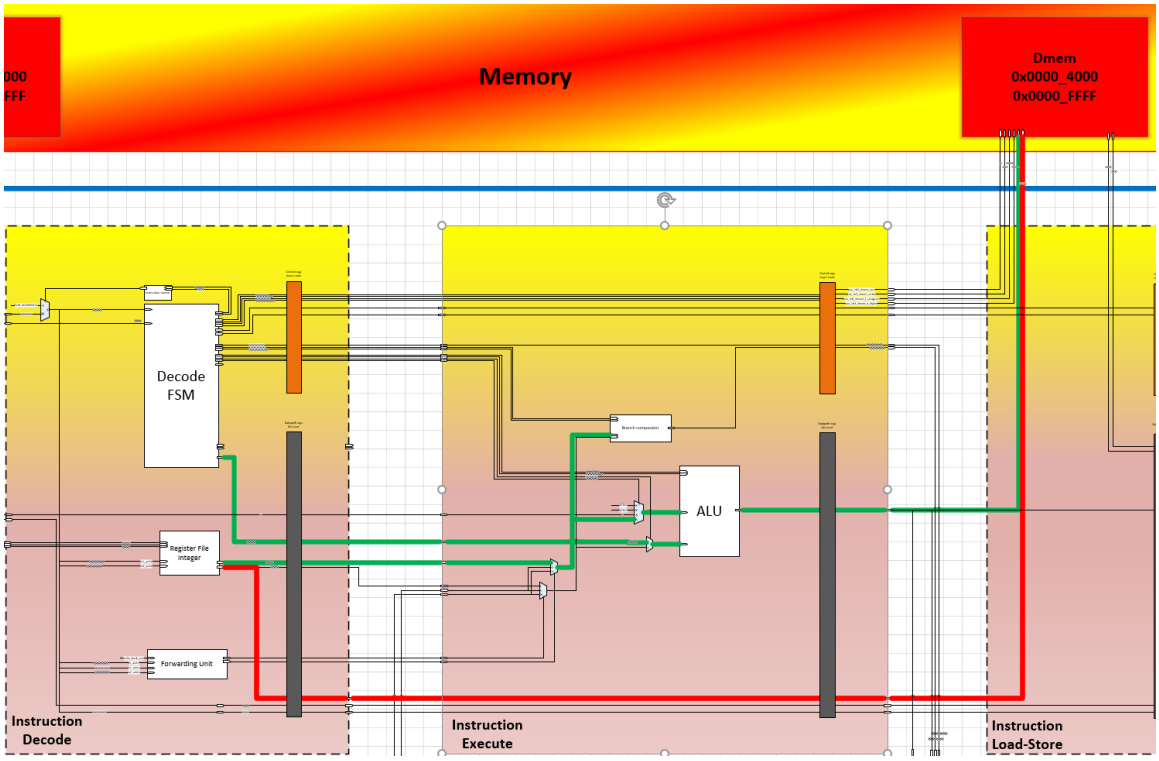


Perform JALR instruction: calculate destination address by adding offset value that encoded in the instruction to a value from the register file, and save PC+4 to the register file:



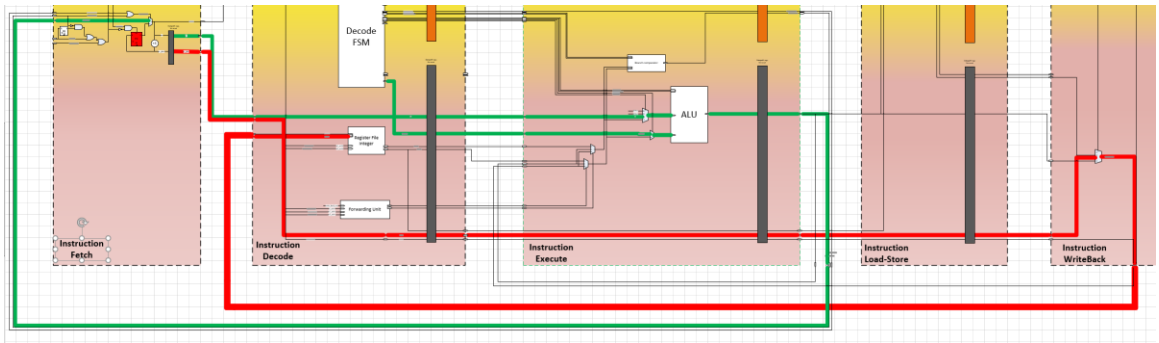
S-Type

S-Type instruction save a value to the Data memory from the register file. The saved value can be either a word (32 bit), half word (16 bit) or 8 bits.
The effective address is obtained by adding the base address value from the register file to an offset that encoded as immediate in the instruction:



J-Type

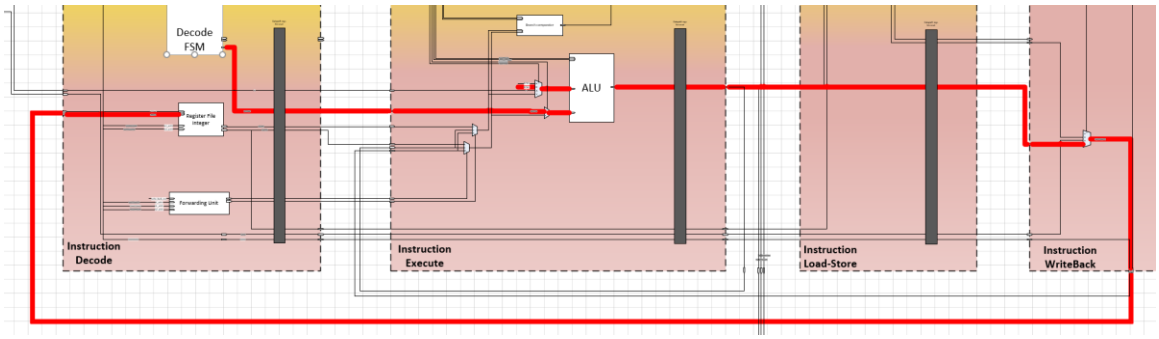
perform addition between immediate encoded in the instruction to the current PC, and save PC+4 to a register in the register file:



U-Type

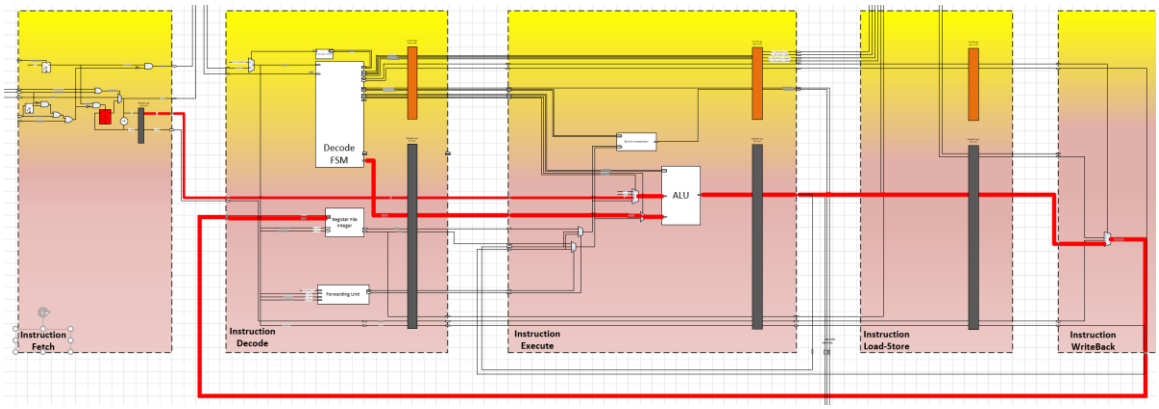
LUI instruction:

this instruction has no source register, only destination register. A 20-bit immediate is save to the 20 upper bit of the destination register:



AUIPC instruction:

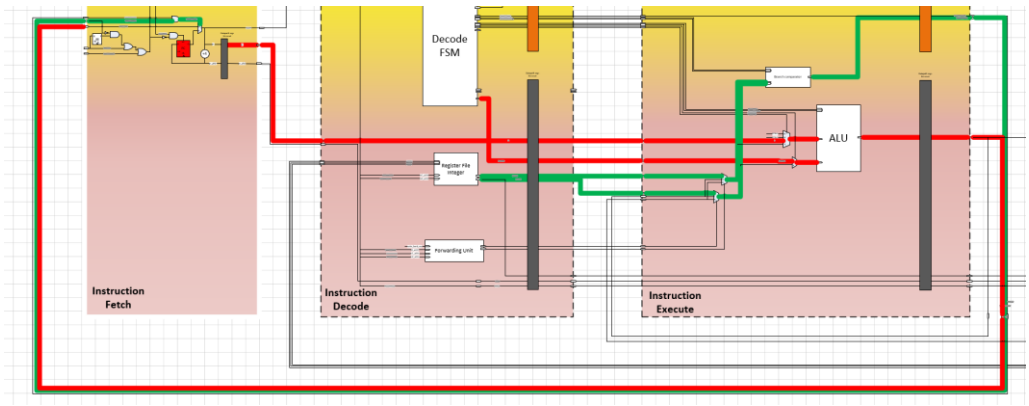
performs addition between the current PC to a 20-bit immediate encoded in the instruction:



B-Type

Branch instructions:

This instruction preforms comparation between 2 registers from the register file and addition between an immediate to the current PC to calculate the destination address:



References

There are no sources in the current document.