Design of a 16 bit multi-cycle RISC processor

Ashwin Rajesh (Sr. No 21519)
M.Tech Microelectronics and VLSI Design
Indian Institute of Science, Bangalore

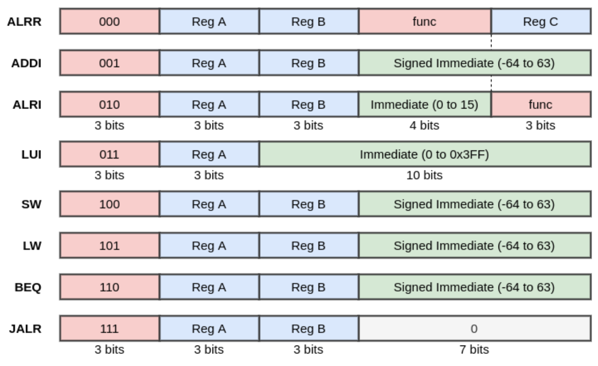
***Abstract*— This report summarises the design of a 16-bit RISC processor as part of the Processor System Design course taken by Prof. Kuruvilla Varghese at the Indian Institute of Science, Bengaluru.**

I. Problem statement

Design a 16-bit RISC CPUwith 16-bit Address spacewith RISCtype instructions. The processoris a multi-cycleprocessorwhich does fetch, decode, execution, and write stages sequentiallyin multiple cycles.It should support minimal arithmetic, logic,and branch instructions with a program counter. It must support a branchon condition to implement loops. Assume 3 operand instructions. Test the processor with an algorithm to find the maximum number from a set of ten random numbers.

II. Instruction set architecture

The architecture was designed with a 16-bit datapath and address width and an address space of 8 registers, r0-r7 of which r0 is always 0. The 3 most significant bits of the instruction is dedicated to the opcode. Arithmetic and logical instructions use additional function bits to decide the operation to be performed.



ALRR	000	Reg A	Reg B	func	Reg C
ADDI	001	Reg A	Reg B	Signed Immediate (-64 to 63)	
ALRI	010	Reg A	Reg B	Immediate (0 to 15)	func
	3 bits	3 bits	3 bits	4 bits	3 bits
LUI	011	Reg A	Immediate (0 to 0x3FF)		
	3 bits	3 bits	10 bits		
SW	100	Reg A	Reg B	Signed Immediate (-64 to 63)	
LW	101	Reg A	Reg B	Signed Immediate (-64 to 63)	
BEQ	110	Reg A	Reg B	Signed Immediate (-64 to 63)	
JALR	111	Reg A	Reg B	0	
	3 bits	3 bits	3 bits	7 bits	

Fig. 1: Instruction set

It is inspired by the RISC-V ISA and more closely from the RiSC-16 teaching ISA[1] used by Prof. Bruce Jacob at UMD. Slight changes were made to implement a wider range of useful arithmetic and logical instructions and use some free space in the instructions. All signed immediate values are sign extended.

The ALRR instruction type implements arithmetic and logical instructions requiring two variable inputs. The ALRI instruction type is added which is used for constant shifts and bit manipulation instructions which need immediate values from 0 to 15. Shift instructions are also added but are not implemented in the design because of high hardware costs of a barrel shifter. Nevertheless, it is also considered in the design and support can be easily added.

ALRR (Arighmetic/Logic RR type) functions		
0000	: ADD	(rd <- rs1 + rs2)
0001	: SUB	(rd <- rs1 - rs2)
0010	: SLT	(rd <- rs1 < rs2 ? 1 : 0)
0011	: SLE	(rd <- rs1 <= rs2 ? 1 : 0)
0100	: SLL	(rd <- rs1 << rs2)
0101	: SRL	(rd <- rs1 >> rs2)
0110	: SRA	(rd <- rs1 >>> rs2)
0111	: RR	(rotate right)
1000	: AND	(rd <- rs1 & rs2)
1001	: OR	(rd <- rs1 | rs2)
1010	: XOR	(rd <- rs1 ^ rs2)
1011	: ANDN	(rd <- rs1 & ~rs2)
1100	: NAND	(rd <- !(rs1 & rs2))
1101	: NOR	(rd <- !(rs1 | rs2))
1110	: XNOR	(rd <- !(rs1 ^ rs2))
1111	: ORN	(rd <- rs1 | ~rs2)

ALRI (Arithmetic/Logic RI type) functions		
000	: SLL	(shift left logical)
001	: SRL	(shift right logical)
010	: SRA	(shift right arithmetic)
011	: RR	(rotate right)
100	: NOT	(bitwise NOT)
101	: SETB	(set bit)
110	: SELB	(select bit with AND mask)
111	: RESETB	(reset or zero out the bit)

Fig. 2: Arithmetic and logical instructions

The LUI instruction instruction loads the top 10 bits (with other bits 0) and is used with the ADDI instruction to load a 16-bit immediate value into memory.

III. Datapath

The processor uses a bus-based microarchitecture. Two bus units transfer the data between the instruction fetch unit, data memory, instruction decoder, register file and the ALU.

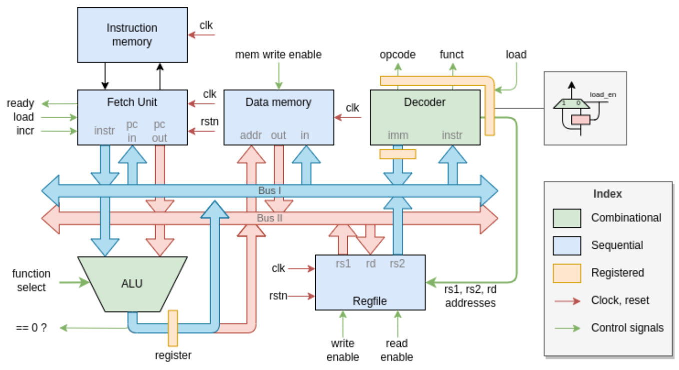


Fig. 3: Architecture of the processor datapath

The fetch unit works independent to the rest of the system. It has an internal program counter and instruction register. The program counter is incremented in the decode cycle, starting an instruction fetch. The ready output signal tells the controller when the next instruction has been latched into the prefetch register. This means variable latency instruction fetches can be easily supported with minimal overhead because the instruction fetch starts immediately after the previous instruction has been decoded. For jump and branch instructions, the fetch starts when the new PC is loaded into the fetch unit. Fig. 4 illustrates the waveform for the fetch stage for normal control flow and jump using the jalr instruction (jump address is obtained directly from register).

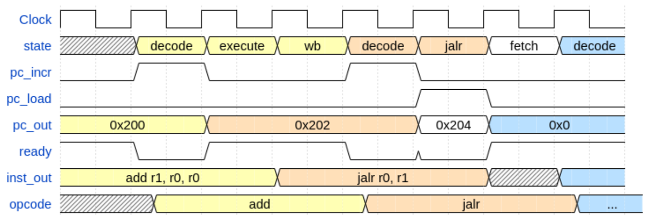


Fig. 4: Instruction fetch waveform

Fig. 5 shows instruction fetch with an instruction fetch taking multiple cycles.

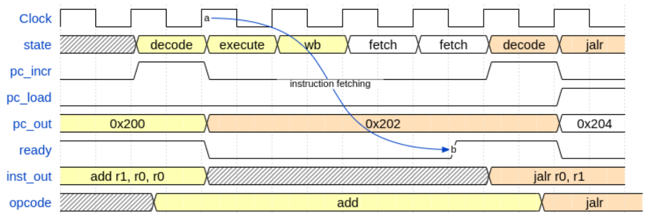


Fig. 5: Multiple cycle instruction fetch waveform

The data memory is simply a BRAM with synchronous read and synchronous write. The address is not registered, so we get the output in the next clock edge after an address is latched onto the input.

The register file is a distributed RAM with synchronous read and write with enable signals for both reads and writes. Register reads also happen in the decode stage.

The decoder generates the rs1, rs2 and rd register addresses, the immediate value and the opcode and function values from the instruction. The opcode and function values are fed to the controller and rs1, rs2 and rd addresses are fed directly to the register file. The decoder is a completely combinational circuit and its outputs are latched. However, since the controller needs the opcode and funct and the regfile needs the addresses in the decode state itself, a forwarding path is made using a multiplexer which activates in the decode state. This structure is called a skid buffer in our design.

The ALU is designed to share as many resources as possible. The ALU has 3 stages. In the first stage, the 2nd input can be inverted. This is useful for subtraction or inverting a mask as used in the RESETB instruction or in the ANDN and ORN instructions. The 2nd stage performs the operations like addition, or one of three logical operations (AND, OR, XOR) or shift (not implemented). A compare unit takes the output of the adder and checks if the output is negative or zero. This is used by the SLT and SLE instructions. A multiplexer selects the output from these 4 units and finally, the output can be inverted if required. This is used primarily for the nand, nor and xnor instructions.

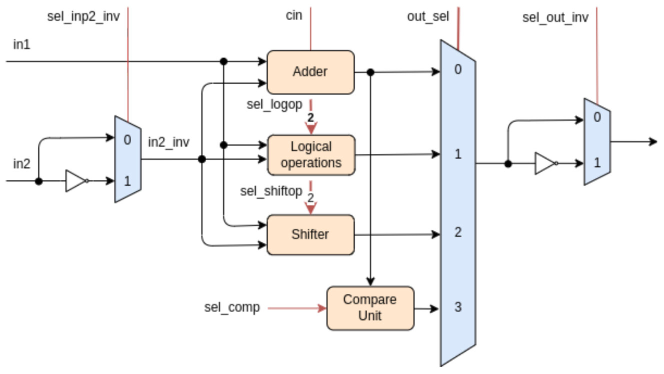


Fig. 4: ALU block diagram

The interconnect network consists of two buses. Each can take input from one of 4 sources and has 4 enable inputs corresponding to each. The interconnect module was written with such that it could use tristate logic if available, else uses multiplexers. The enable signals to the bus is given from the controller.

IV. Controller

The controller starts from a fetch state where is stays till the fetch unit gives the ready signal. Thereafter, it goes to the decode stage and then to one of multiple state sequences depending on the decoded opcode. There are 5 main branches, for arithmetic and logical (including ADDI, LUI), for load, for store and for jump and link and branch instructions. After finishing the sequence, the next state logic checks if the fetch unit has finished fetching the next instruction. If so, it goes to the decode state. Else, it goes to the fetch state and waits for the next instruction to be ready.

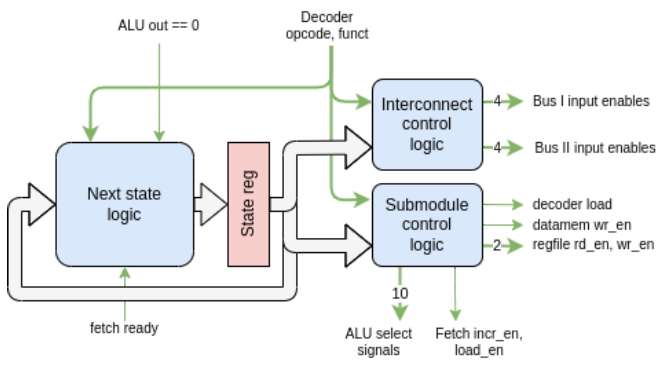


Fig. 6: Controller block diagram

The controller also controls the interconnect and decides which component gives the output on each of the two buses.

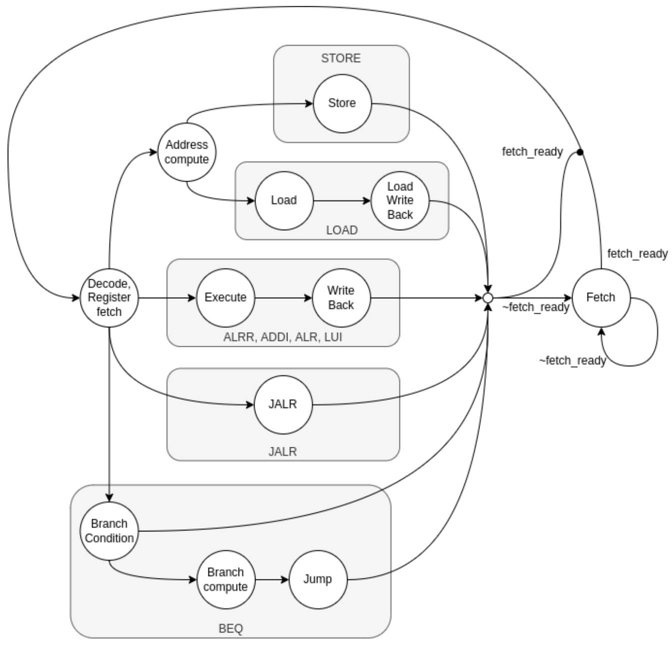


Fig. 7: States of the processor controller

Table I shows the number of cycles required for each instruction. Typically, instructions take 3 cycles except branch which can take 2 cycles if not taken and 5 if taken and load word which takes 4 cycles. The additional delay in load word is because the data memory is synchronous. The additional delay in the case of branch instruction is because we need to do execute in two cycles, one for comparison and one for computing the branch target. Both branch and JALR take an additional cycle in the fetch state to fetch the next instruction.

Instruction type	**Number of cycles**
ALRR	3
ADDI	3
ALRI	3
LUI	3
LW	4
SW	3
JALR	3 (including fetch next)
BEQ	2 (not taken), 5 (taken)

Table I: Number of cycles taken to execute instructions

V. Test program

The test program finds the (signed) maximum value among 10 numbers and stores it in the R2 register. The following is the assembly code for the test program.

```
add r2 r0, r0       # r2 = 0
addi r1, r0, 10     # r1 = 10
addi r1, r1, -1     # r1 = r1 - 1
lw r3, r1, 0        # r3 = mem[r1]
slt r4, r2, r3      # r4 = r2 < r3 ? 1 : 0
beq r4, r0, 2       # if r4 == 0 (r3 < r2),
                    # skip next instruction
add r2, r3, r0      # r2 = r3
beq r1, r0, 2       # If r1 == 0, stop
beq r0, r0, -14     # else, repeat
beq r0, r0, -2      # inf loop / halt
nop
…
```

Similarly, a test program for a blink program with a two-level nested loop for delay was also written and tested

VI. Results

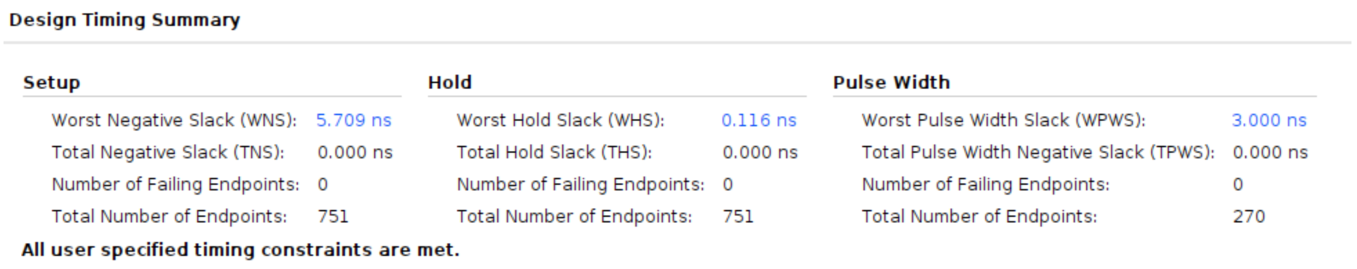
The design was implemented and tested on the basys3 board for blink and maximum programs for 50MHz. The worst negative slack was 5.7ns giving a maximum frequency of **69.93MHz**. About 2% of LUT and BRAM resources were used for implementing the design.

VII. References

[1] The Risc-16 Instruction set architecture, *Prof. Bruce Jacob, University of Maryland*

[2] The RISC-V Instruction Set Manual, *RISC-V foundation*

[3] Computer Organization and Design "The Hardware/Software Interface: RISC-V Edition" *David A. Patterson, John L. Hennessy*



Design Timing Summary

Setup

Worst Negative Slack (WNS):	5.709 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	751

Hold

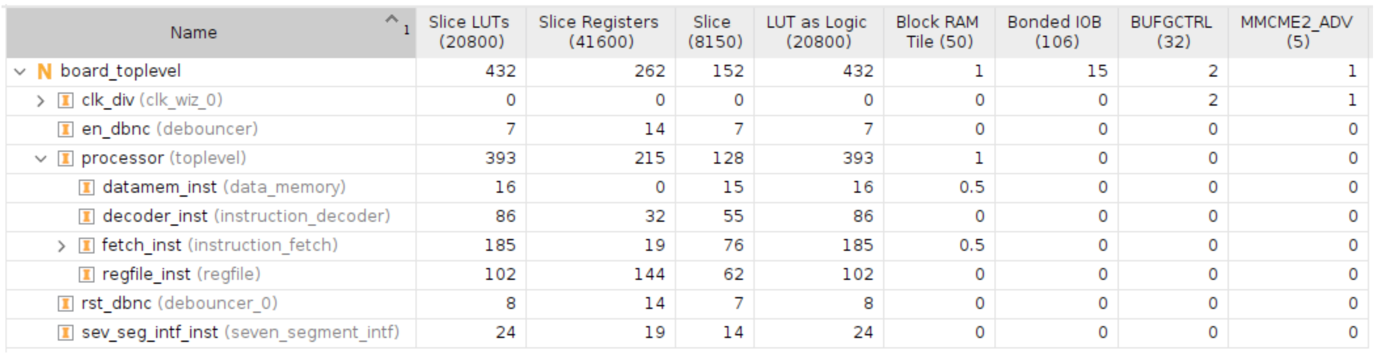
Worst Hold Slack (WHS):	0.116 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	751

Pulse Width

Worst Pulse Width Slack (WPWS):	3.000 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	270

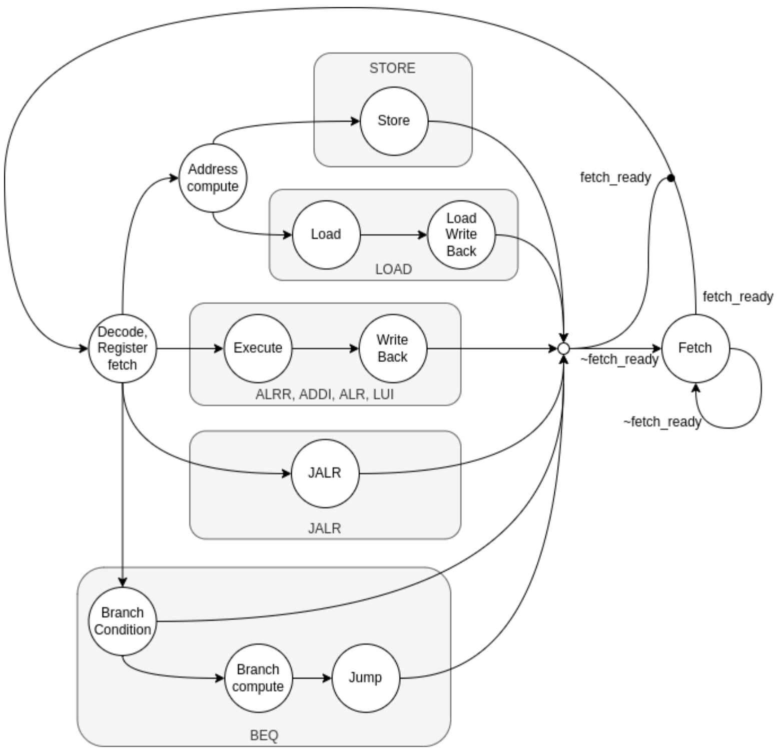
All user specified timing constraints are met.

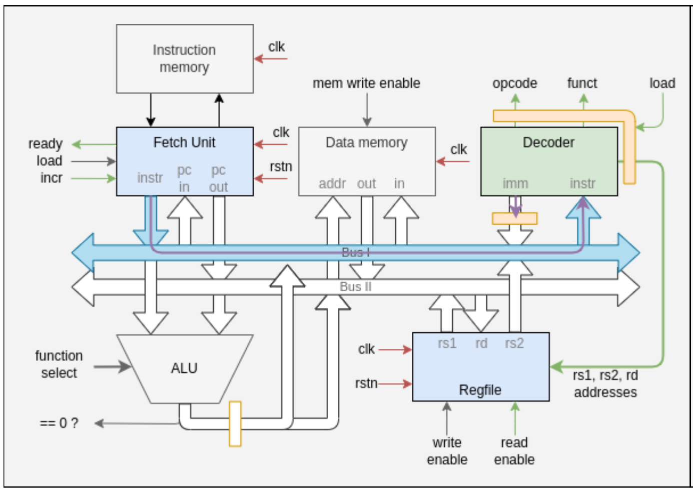
Timing report summary



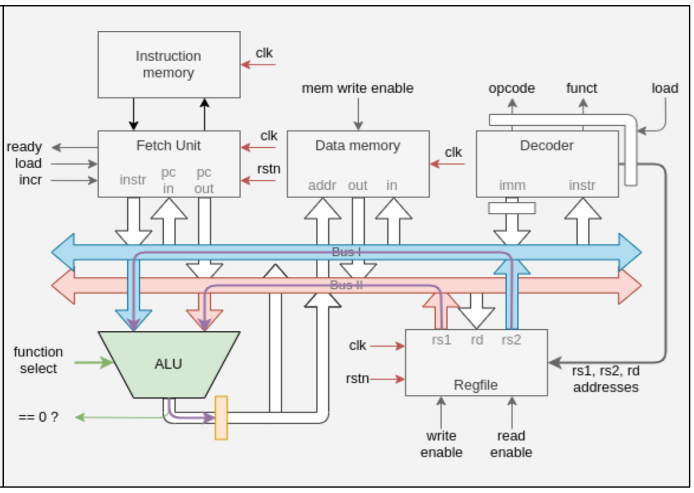
Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)	MMCME2_ADV (5)
board_toplevel	432	262	152	432	1	15	2	1
clk_div (clk_wiz_0)	0	0	0	0	0	0	2	1
en_dbnc (debouncer)	7	14	7	7	0	0	0	0
processor (toplevel)	393	215	128	393	1	0	0	0
datamem_inst (data_memory)	16	0	15	16	0.5	0	0	0
decoder_inst (instruction_decoder)	86	32	55	86	0	0	0	0
fetch_inst (instruction_fetch)	185	19	76	185	0.5	0	0	0
regfile_inst (regfile)	102	144	62	102	0	0	0	0
rst_dbnc (debouncer_0)	8	14	7	8	0	0	0	0
sev_seg_intf_inst (seven_segment_intf)	24	19	14	24	0	0	0	0

Utilization report summary

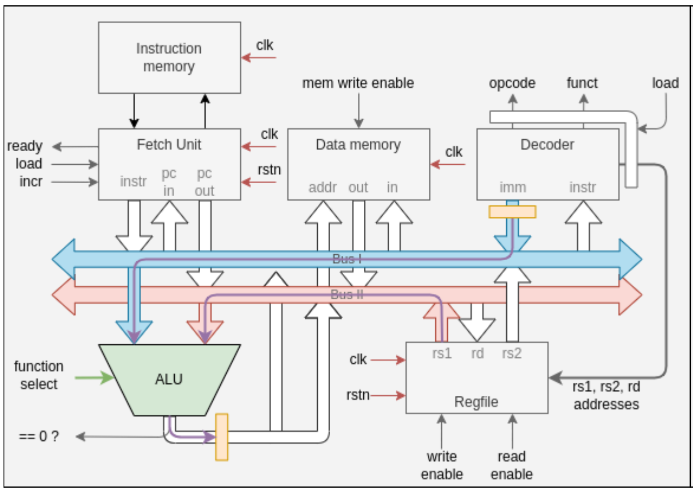




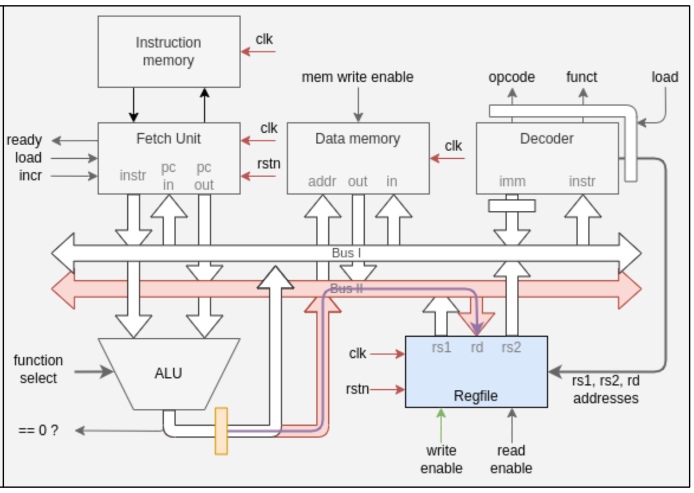
Decode state



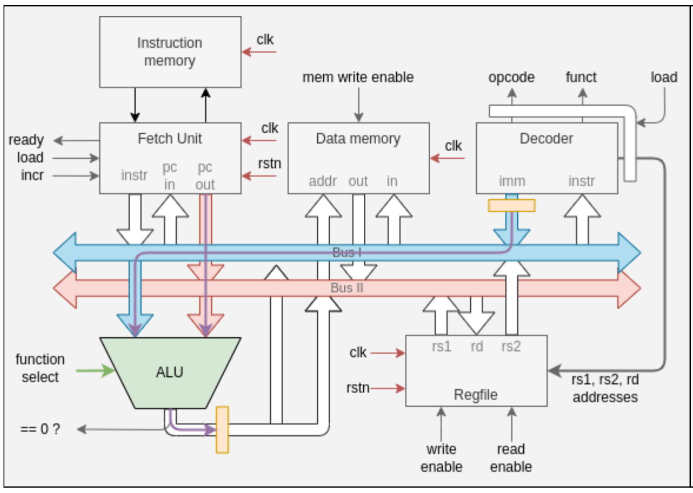
RR type execute (ALRR) & BEQ condition states



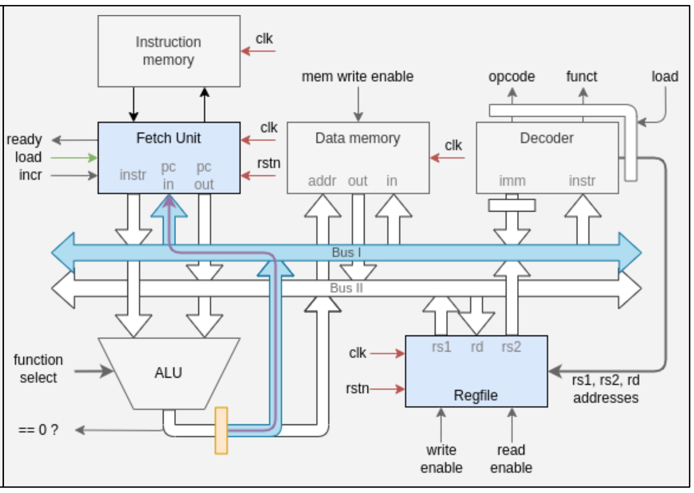
RI type execute (ALRI, ADDI, LUI), Address compute states



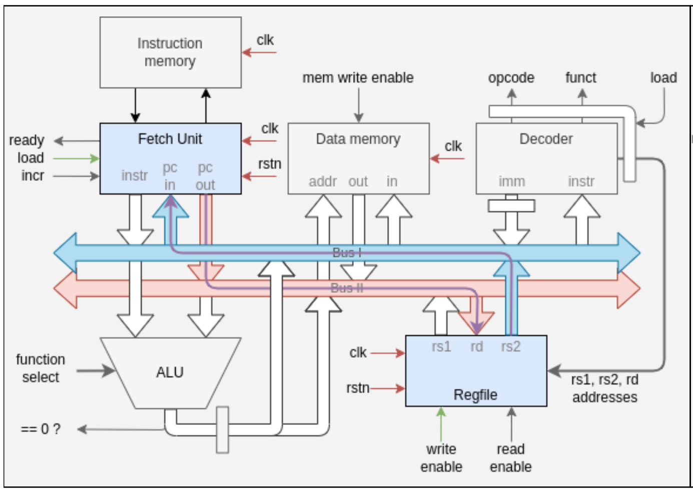
Writeback state



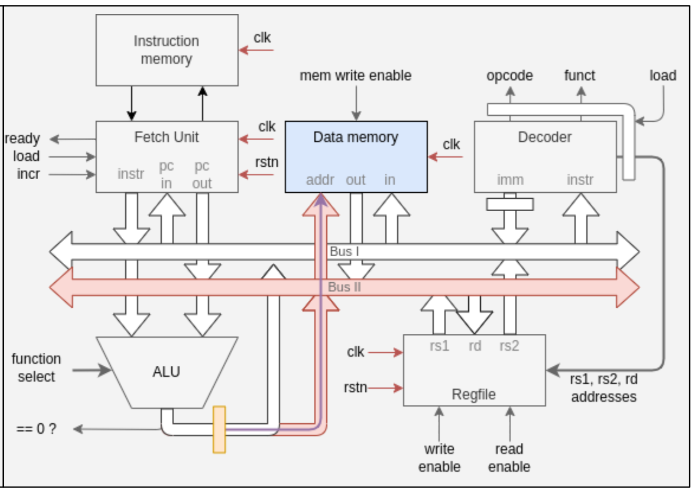
BEQ target compute state



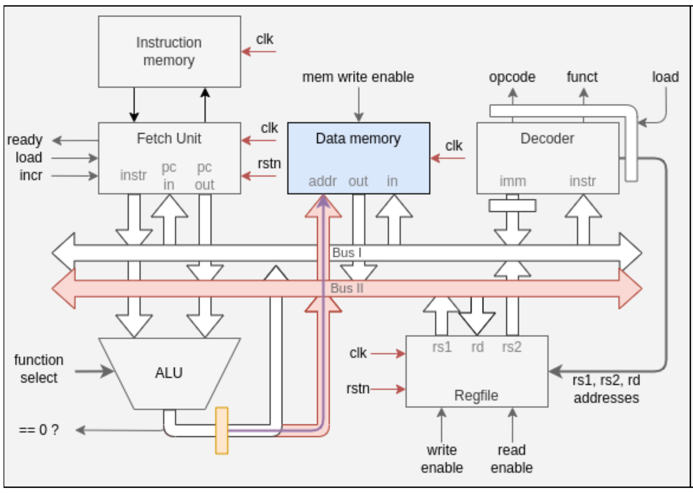
BEQ branch state



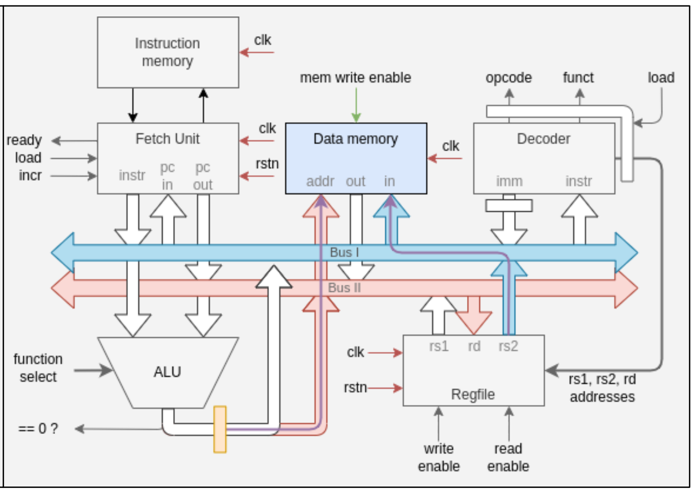
JALR state



Load state



Load writeback state



Store state