Department of Electronic & Telecommunication Engineering

University of Moratuwa

In19-S5-EN3030

Circuits and Systems Design

Single Cycle CPU Design with Direct Mapped Cache having Victim Cache functionality

Hermes - Group – 8

Table of Contents

Introduction

This document provides a comprehensive introduction to the RISC-V 32-bit CPU with a direct mapping cache and victim cache that has been designed and built. The CPU is based on the RV32I RISC-V ISA specification and supports the instruction classes R-type, I-type, S-type, and SB-type. The successful implementation of U and UJ-type instructions was a bonus and demonstrate the versatility of the design.

The CPU has been designed with performance in mind and the single-cycle architecture ensures that the overall delay is kept to a minimum. The integration of the direct mapping cache with a victim cache significantly enhances the performance of the CPU, allowing it to run faster and more efficiently. The design of the CPU has been optimized for both functionality and resource usage, ensuring that it is both capable and efficient.

To verify the design of the CPU, a set of assembly programs have been run and the results were carefully analyzed. The results demonstrate that the CPU is capable of executing all the instructions in the specified instruction classes and that it performs as expected.

Instruction Memory

Supported Instruction Classes

Our RISCV Processor supports the following Instruction Formats,

- R – Format
- I – Format
- S – format
- Sb – Format
- J – Format

To implement the above instructions, we created the following instruction formats.

op	rs	rt	rd	shamt	funct
R - Type					

op	rs	rt	Immediate
I - Type			

op	Target Address
J - Type	

op	rs	rd	offset
S - Type			

op	rs	rt	offset
Sb - Type			

R – Type – All the arithmetic operations

I – Type – Load instructions from memory(LW), Add immediate(Addi), Set less than immediate(SLTI)

S – Type – Store data in memory(SW)

Sb – Branch equal(BEQ), Branch not equal instructions(BNE)

J – Type – jump instructions(J), Jump and link instructions(Jal)

Op – basic operations for the instructions(opcode)

Rs – first source operand register

Rt – second source operand register

Rd – destination operand register

Shamt – shift amount

Funct – function opcode

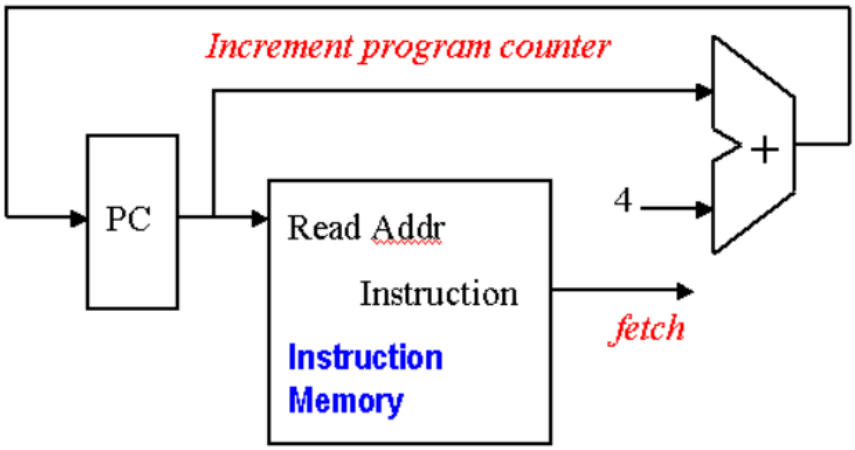
Address – offset for load/store or the jump.

Immediate – constant for immediate instructions

32 bits are divided as follows;

- **Bits 31-26:** *opcode* - always at this location
- **Bits 25-21 and 20-16:** *input register indices* - always at this location.
- **Bits 25-21:** *base register* for load/store instruction - always at this location.
- **Bits 15-0:** *16-bit offset* for branch instruction - always at this location.
- **Bits 15-11:** a *destination register* for R-format instruction - always at this location.
- **Bits 20-16:** a *destination register* for load/store instruction - always at this location.

Be aware that the two destination registers' varied positions necessitate the use of a selector (also known as a mux) to identify the correct field for each type of instruction.



Instruction Memory

00000000010000010000000000000000 – load R1 <- Mem{R2 +0}

00000000010000010000000000000001 – load R2 <- Mem{R2 +1}

00001000001000100001100000100000 – Add R3<- R1 + R2

00000100001000110000000000000000 – Store Mem{R1 + 0} <- R2

00001000010000010001100000100010 – Sub R3 <- R1-R2

00001000001000000011100000100110 – invert R3 <- R1!

00001000001000100001100000101000 – Logical shift left R3<-R1<<R2

00001000001000100001100000101001 – Logical shift right R3<-R1>>R2

00001000001000100001100000100011 – And R3<- R1 and R2

00001000001000100001100000100100 – Or R3<- R1 or R2

00001000001000100001100000100111 – R3<- 1 if R1<R2

00001000001000100001100000100101 – R3 <- R1 ^ R2

00001000001000010000100000100000 – R1 <- R1 + R1

00001100001000100000000000000001 – Brach to jump if R1=R2, PC = PC +4 + OFFSET <<2; offset = 1

00010000001000100000000000000000– Brach to jump if R1=!R2, PC = PC +4 + OFFSET <<2; offset = 0

00011000000000000000000000000000 – Jump to the beginning address

ALU Design

When implementing the ALU, we used 3 inputs. Out of these three inputs, one is from a read register in the general-purpose registers. Another input is given through a multiplexer. If the ALUSrc value is equal to zero, we get the input from the read registers otherwise we get the sign extended instruction[15:0] part. There is another input for the ALU to find out what the instruction ALU is supposed to execute. This input is generated through the ALU control unit.

Using the ALU control unit, we generate the ALU control signal which is the commander for the ALU to do a specific type of instruction. We have chosen a 6-bit ALU control unit input and a 2-bit ALU Op to generate a mapping of the instruction[5:0] to a 5-bit ALU control signal.

By considering the ALU control signal, the ALU will operate on the two input values and give an output. Also, there is a zero flag is present in our design to check whether the ALU output is equal to zero. Moreover, we have considered the overflow situation and the negative output situation also. If an overflow or a negative output generates, we set our ALU output to an all 1, 32-bit value.

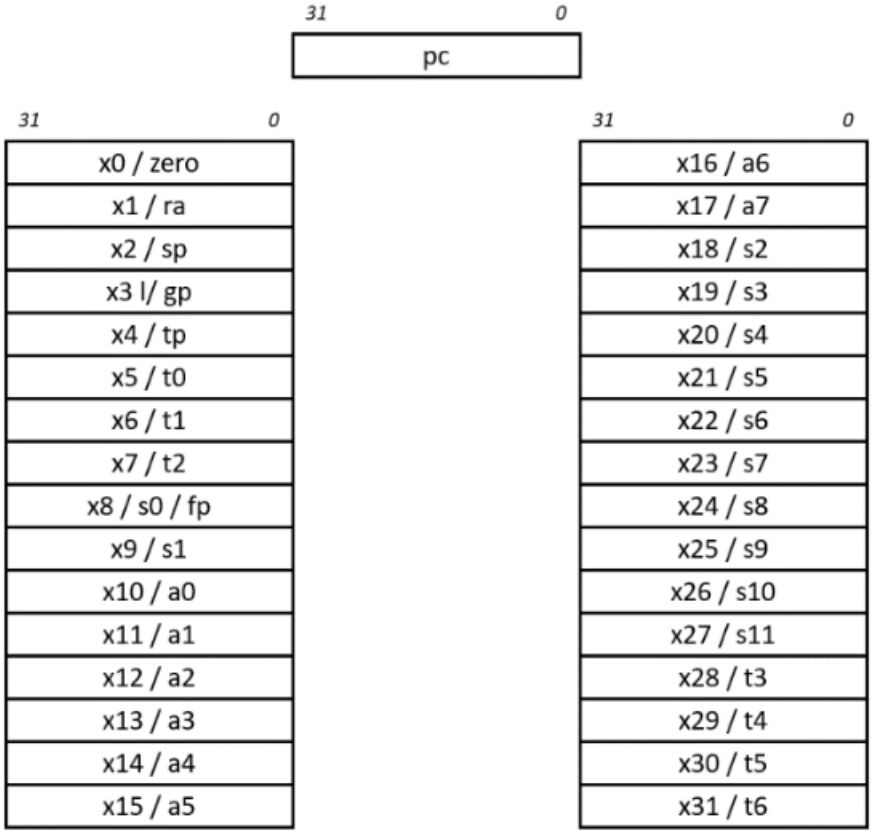
ALU Control Unit

ALU OP	OP Code	ALU Cnt	OP	Instruction
010	xxxxxx	00000	Add	Lw, Sw, Addi
001	xxxxxx	00001	Sub	Beq, Bne
000	100000	00000	Add	R-add
000	100010	00001	Sub	R-sub
000	100011	00010	AND	R-AND
000	100100	00011	OR	R-OR
000	100101	00100	XOR	R-XOR
000	100110	00101	NOT	R-NOT
000	100111	00110	SLT	R-SLT
000	101000	00111	LSL	R-LSL
000	101001	01000	LSR	R-LSR
011	xxxxxx	00110	SLTI	I-SLTI

General Purpose Registers

In this design, we used 31 registers with 31 bits each. We gave clock, reset, Write enable, register write destination, and register writes data as the inputs. We got the register read data and its address, register 2 data, and address as the outputs.

The general-purpose register array diagram is as follows.



RISC-V base ISA register set

X0 – Hardwired to zero.

Two read ports and one write port are located on the register file's hardware (corresponding to the two inputs and one output of the ALU). The RF and ALU comprise the components needed to calculate ALU instructions. A set of registers in the RF can be read or written by providing the appropriate register number and, in the case of write operations, a write authorization bit.

Control Unit

The control unit handles the control micro instructions on how to control each module in the processor. We give instruction[31:26] as the input to the control unit.

The outputs of the control signals are as follows.

- RegDst
    - 0: Register destination number for the Write register is taken from bits 20-16 (rt field) of the instruction
    - 1: Register destination number for the Write register is taken from bits 15-11 (rd. field) of the instruction
- ALUSRC
    - 0: The second ALU operand is taken from the second register file output (Read Data 2)
    - 1: the second Alu operand is the sign-extended, lower 16 bits of the instruction
- Mem to Reg
    - 0: Alu result to the register
    - 1: Read the address from the memory to the register
- RegWrite
    - 0: The value present at the writeData input is output from ALU
    - 1: The value present at the register WriteData input is taken from data memory.
- MemRead
    - 0: No action
    - 1: Data Memory contents designated by address input are presented at the ReadData.
- MemWrite
    - 0: No action
    - 1: Data memory contents designated by address input are presented at the WriteData input.
- Branch
    - 0: PC is overwritten by the output of the adder(PC + 4)
    - 1: PC is overwritten by the branch target address
- ALUOP
    - Given the instruction type selection to the ALU control unit.
        - 00 – R type
        - 10 – I type,
        - 10 – STORE(S)
        - BRANCH(BEQ, BNE)
        - 11 – SLTI
- Jump
    - Enable jump to identify J and JAL instructions.

Instruction	RegDST	ALUSRC	Mem_to_Reg	Reg Write	Mem Read	Mem Write	Branch	ALSOP	Jump
R – Type	1	0	0	1	0	0	0	00	0
LW	0	1	1	1	1	0	0	10	0
SW	X	1	X	0	0	1	0	10	0
BEQ	X	0	X	0	0	0	1	01	0
BNE	X	0	X	0	0	0	1	01	0
ADDI	0	1	0	1	0	0	0	10	0
J	X	0	X	0	0	0	0	00	1
JAL	2	0	2	1	0	0	0	00	1
SLTI	0	1	0	1	0	0	0	11	0
SLI	0	1	0	1	0	0	0	11	0

Data Memory

We give clock, memory access address memory writes data memory write enable, and memory read enable as the inputs to the memory. Then we output the memory read data as the output. Memory is 512-size arrays with 32 bits each. For the demonstration, we use the memory of 7 arrays to show that data memory works accurately. Memory writing and memory reading has happened at every positive clock edge.

Cached and Victim Memory

The code is a Verilog implementation of a victim cache which is an add-on to a traditional cache and is used to store evicted cache blocks temporarily before they are moved to the main memory. The main aim of a victim cache is to reduce the number of cache misses.

The code defines two arrays, cache and VCache that act as the cache and victim cache, respectively. The size of these arrays is determined by the parameters CACH_AD and DM_ADDRESS. Both arrays store DATA_W bits of data. The code also defines two arrays, tags and Vtags which are used to store the tags of the cache and victim cache, respectively. These arrays are used to determine whether the required data is present in the cache or not.

The code has two blocks of logic, the first block is for reading and the second block is for writing. The first block is triggered by a rising edge of the clock CLK and the read signal MemRead must be high. The block takes the input, which is the address of the data being read, and splits it into two parts, tagbit, and cache_col. The cache_col part is used to index the cache and tag arrays and the tagbit part is used to compare with the tags stored in the tags and Vtags arrays.

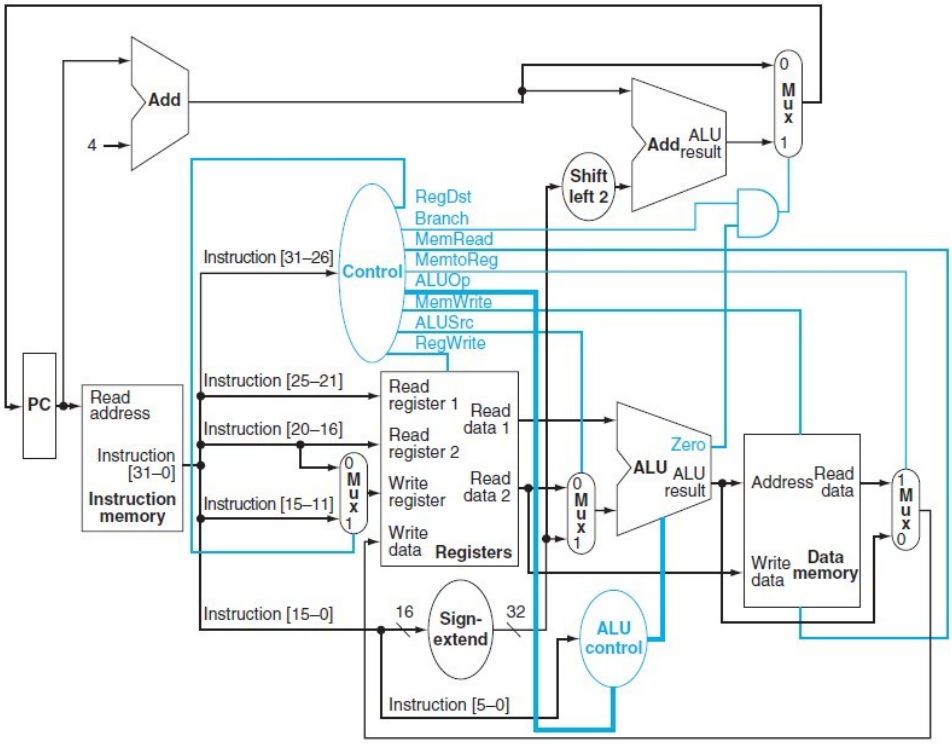
If the tagbit matches the tag stored at the indexed location of the tags array, the data stored at that location of the cache array is returned as the rd output. If there is a miss in the cache, the code then checks if the tagbit matches the tag stored at the indexed location of the Vtags array. If there is a match, the data stored at that location of the Vcache array is returned as the rd output. Finally, if there is still a miss, the data is read directly from the main memory array mem.

The second block of logic is triggered by a rising edge of the clock CLK and the write signal MemWrite must be high. This block writes the wd input data to the main memory array mem. The input address is split into tagbit and cache_col as in the read block. If the tagbit matches the tag stored at the indexed location of the tags array, the data is written to both the cache and main memory arrays. If there is a miss in the cache, the code checks if the tagbit matches the tag stored at the indexed location of the Vtags array. If there is a match, the data is written to the victim cache and main memory arrays. If there is still a miss, the data is written only to the main memory array.

Datapath

Datapath gives how to connect each module and ensure data flow is happening correctly. It also uses to handle the program counter and change instruction selection according to the control unit. Also, jump and branching instruction handling is carryout by the Datapath module. Sign extend, zero extend handling for branch, jump, and sign extend pc increment are also carried out by the Datapath module.

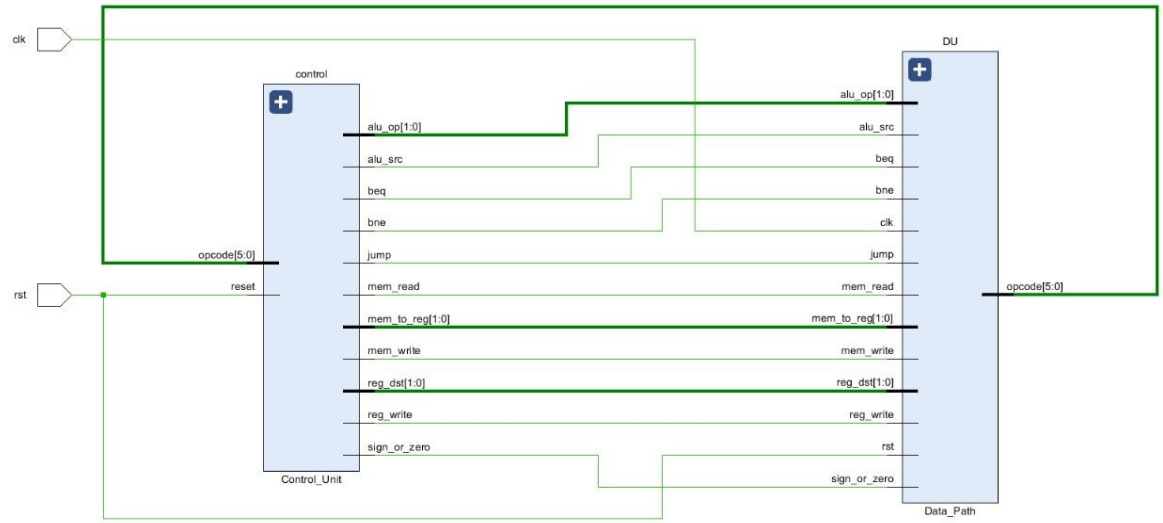
We use the following Datapath diagram for our design.

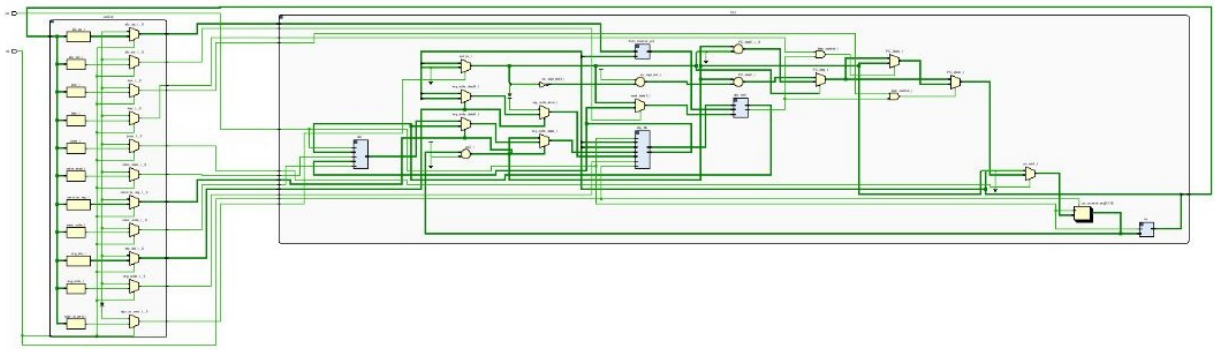


Performance

The performance of the processor is as follows.

- Our processor can handle R, I, S, Sb, and J-type instructions.
- We verified the processor design by giving 16 instructions with 7 data from data memory.
- Created a test bench to run the program.
- R – type instruction had the longest delay path.
- Processor includes cache and victim memory. Victim memory worked as an L2 cache.





Vivado Block Diagram