RISC-V CPU Design on FPGA

Yuan Li, Yibin Xu, Bo Li
University of California,
Davis Davis, USA

INTRODUCTION

Our project is a RISC-V CPU custom design that uses 4 different configurations

- multi-cycle
- 5 stage pipelined with simple stalling
- 5 stage pipelined with forwarding
- 5-stage pipelined with forwarding and branch prediction)

Those can run RISCV32I instruction set programs with a subset of instructions in the instruction set. A collection of 2 programs that are relevant to nowadays's increasingly common machine learning and AI operations: matrix multiplication and breadth first search.

BACKGROUND

Many different instruction set architectures exist, but RISC-V is a relatively new instruction set architecture that is open source, high performance, and highly scalable, making it particularly promising in this data- driven era. RISC-V is a free and open instruction set architecture that is driven through open collaboration while enabling freedom of design across all domains and industries and cementing the strategic foundations of semiconductors.

PROBLEM DEFINITION

Breadth First Search

Algorithm 1 A Breadth First Search Algorithm

```
function BFS(G, Root Vector)
    let N be the number of nodes in G
    let rv be N−1 vectors with N 0's
    rv[0] := G · Root Vector
    for i from 1 to N−1 do
        rv[i] := G · rv[i−1]
    end for
return Result Vectors
end function
```

Matrix Multiplication

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$= \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Multi-cycle

In a multicycle CPU design, instructions are divided into a series of smaller stages, each dedicated to a specific phase of operation. These stages include instruction fetch, instruction decode, instruction execution, memory access, and register write back. The execution of each cycle is determined based on the type of instruction fetched from the instruction memory (IMemory) within the CPU.

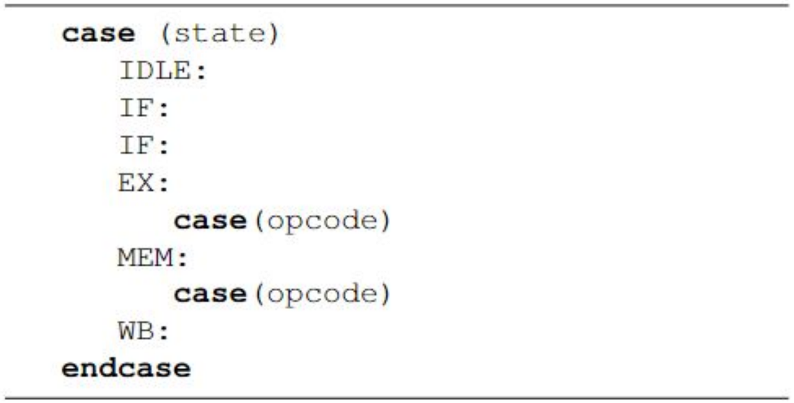
Problem

Initially, the implementation consisted of a single always @(posedge clk) block, without clear separation between combinational and sequential logic. As a result, numerous unintended registers and latches were created along the datapath, causing issues with memory access.

The design is later separated into 1 combinational and 1 sequential parts, with the sequential part having all necessary registers getting their corresponding values, and the combinational part fetching and decoding instructions, executing calculations, reading from to writing to data memory or writing back to registers

Complexity

Multicycle CPUs tend to have a simpler design compared to pipeline CPUs. The CPU architecture can be implemented in a more straightforward manner. Our design primarily relies on a “case” statement, which determines the appropriate actions based on the current “state” and the opcode of the input instructions. Each instruction will finish all necessary stages before the next instruction is fetched from the instruction memory and executed



```
case  (state)
    IDLE:
    IF:
    IF:
    EX:
        case(opcode)
    MEM:
        case(opcode)
    WB:
endcase
```

Performance and Hardware Resources

Multicycle CPUs typically have longer instruction execution times since each instruction requires multiple cycles to complete. However, they can achieve higher clock frequencies due to simpler logic and can be better optimized in certain tasks.

Multicycle CPUs requires fewer hardware and control logic units compared with the pipelined implementations.

For example, we only utilized a total of 49 registers for our multicycle CPUs compare with a total of 1317 registers on the pipelined CPU with branch prediction

5-Stage Pipelined with Simple Stalling

This processor not only executes instructions in a pipelined manner but also incorporates hazard detection mechanisms within the assembly program. By introducing stalls, it ensures correct data flow and maintains the temporal order of the program execution.

The architecture comprises four distinct modules, each separate from the top module, which houses the combinational logic responsible for generating accurate control signals for data flow and other components such as the ALU.

These modules include the:

- The main control module (depicted as a light blue oval near the top of the diagram)
- The ALUControl module (represented as a light blue oval near the bottom of Figure 1)
- The ALU itself.

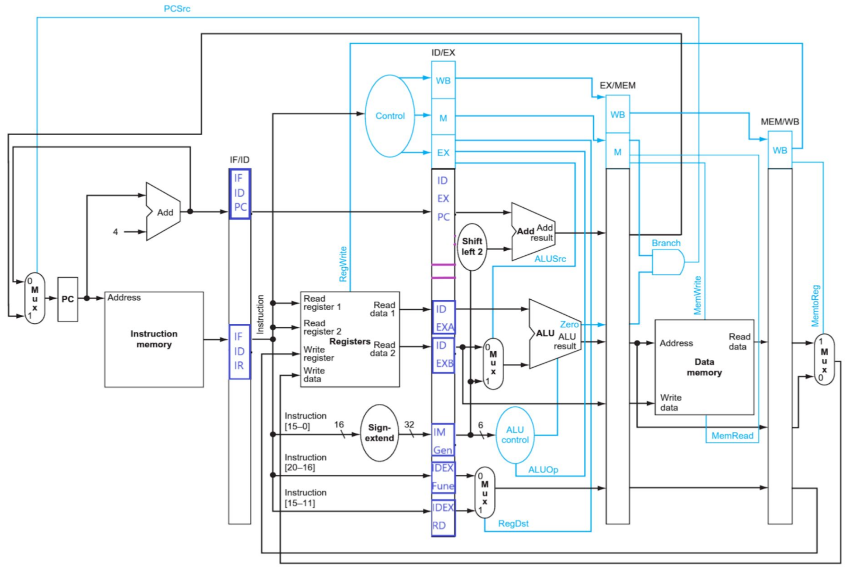
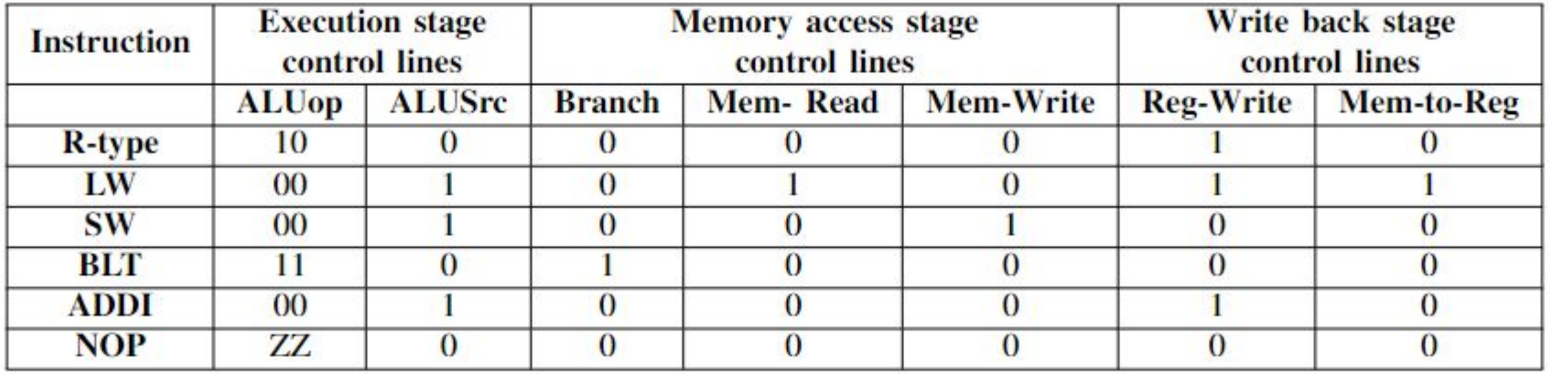


Fig. 1. 5-Stage Pipelined with Stalling

Control Module

This module is responsible for setting all of the control that governs the operations of the processor. It takes in the opcode of the instruction as input and sets several control signals using the following truth table:

TABLE I
CONTROL MODULE SIGNALS

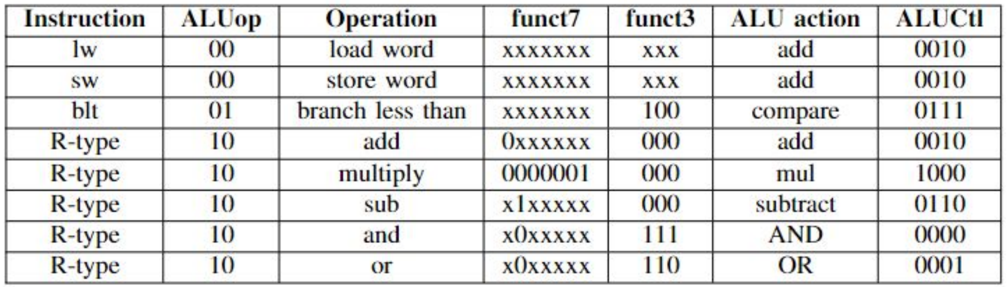


Instruction	Execution stage control lines		Memory access stage control lines			Write back stage control lines	
	ALUop	ALUSrc	Branch	Mem- Read	Mem-Write	Reg-Write	Mem-to-Reg
R-type	10	0	0	0	0	1	0
LW	00	1	0	1	0	1	1
SW	00	1	0	0	1	0	0
BLT	11	0	1	0	0	0	0
ADDI	00	1	0	0	0	1	0
NOP	ZZ	0	0	0	0	0	0

ALU Control Module

The output of the ALUControl module is the 4-bit "ALUCtl" signal, which determines the operation that the ALU will perform. The module follows a truth table to determine the appropriate value of the "ALUCtl" signal based on the input signals.

ALUCONTROL MODULE TRUTH TABLE



Instruction	ALUop	Operation	funct7	funct3	ALU action	ALUCtl
lw	00	load word	xxxxxxx	xxx	add	0010
sw	00	store word	xxxxxxx	xxx	add	0010
blt	01	branch less than	xxxxxxx	100	compare	0111
R-type	10	add	0xxxxxx	000	add	0010
R-type	10	multiply	0000001	000	mul	1000
R-type	10	sub	x1xxxxx	000	subtract	0110
R-type	10	and	x0xxxxx	111	AND	0000
R-type	10	or	x0xxxxx	110	OR	0001

ALU Module

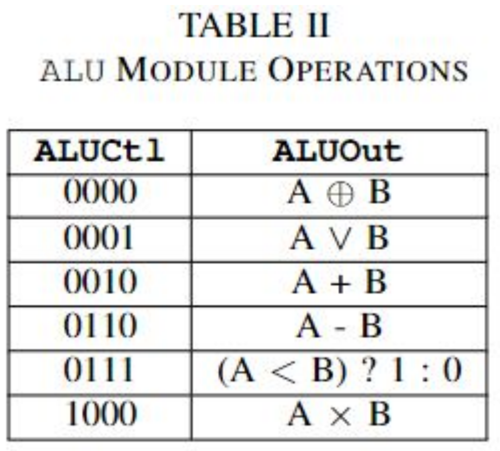
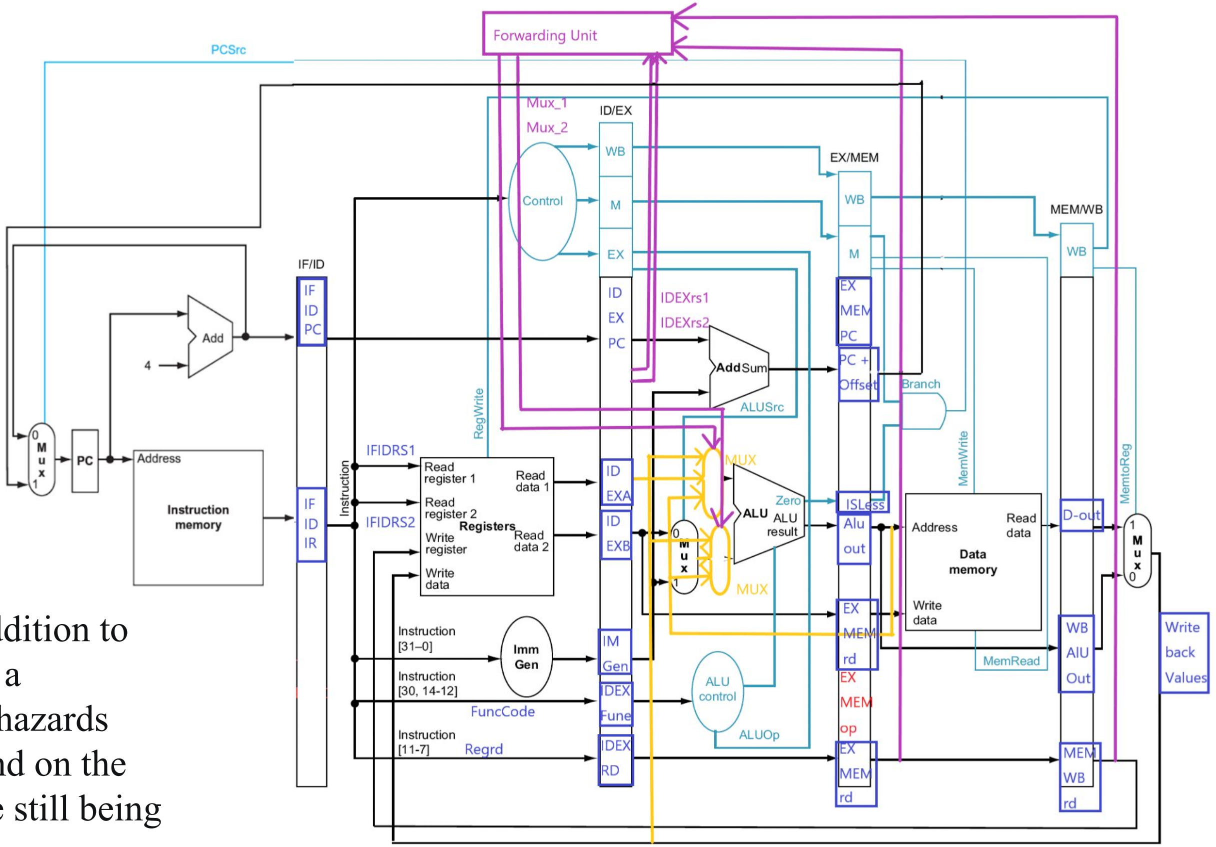


TABLE II
ALU MODULE OPERATIONS

ALUCtl	ALUOut
0000	$A \oplus B$
0001	$A \vee B$
0010	$A + B$
0110	$A - B$
0111	$(A < B)\ ?\ 1 : 0$
1000	$A \times B$

This module is the ALU that does some basic operations with its two input operands and based on the “ALUCtl” signal that was produced in the ALU Control module described in the section above.

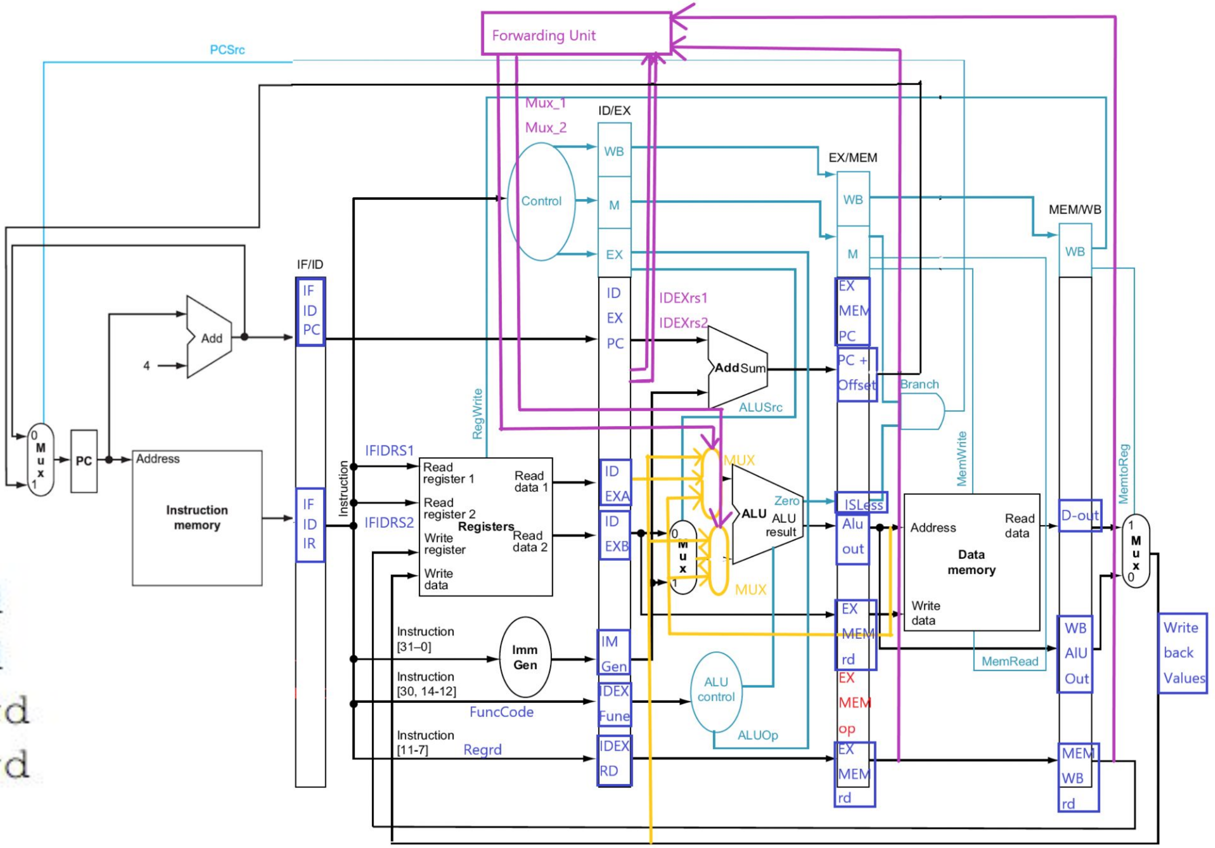
Forwarding



The Forwarding module is a crucial addition to our pipelined version of the CPU. It is a technique used to minimize stalls and hazards that can occur when instructions depend on the results of previous instructions that are still being processed in the pipeline.

When an instruction requires the result of a previous instruction that has not yet completed its execution, a hazard occurs. Without forwarding, the pipeline would have to stall until the required data is available, which would reduce the CPU's performance

- `IDEXrs1 == EXMEMrd`
- `IDEXrs2 == EXMEMrd`
- `EXMEMrs1 == MEMWBrd`
- `EXMEMrs2 == MEMWBrd`



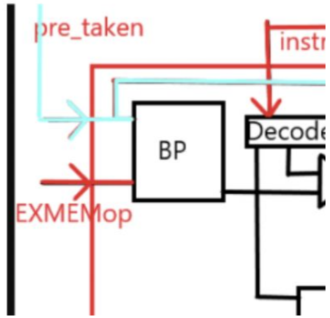
Branch Prediction

Branch prediction is a technique used in Our pipelined RISC-V CPUs (and other processors) to improve performance by predicting the outcome of branch instructions before they are actually executed.

The goal of branch prediction is to minimize the impact of these control hazards by predicting whether a branch will be taken or not taken and preemptively continuing the execution based on that prediction.

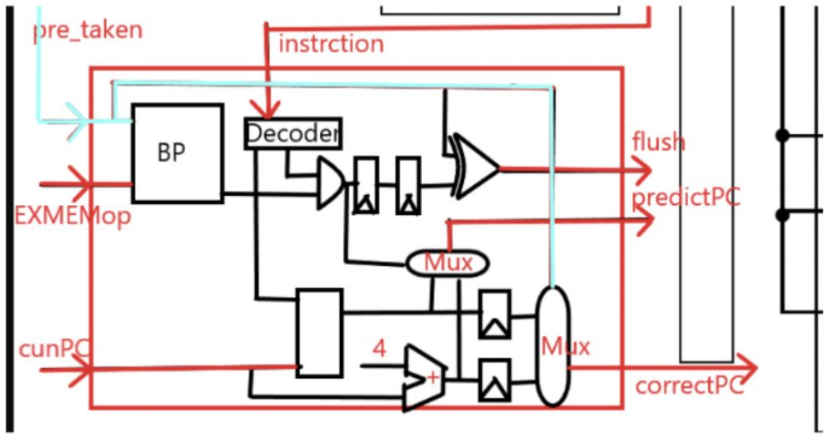
Branch Predictor

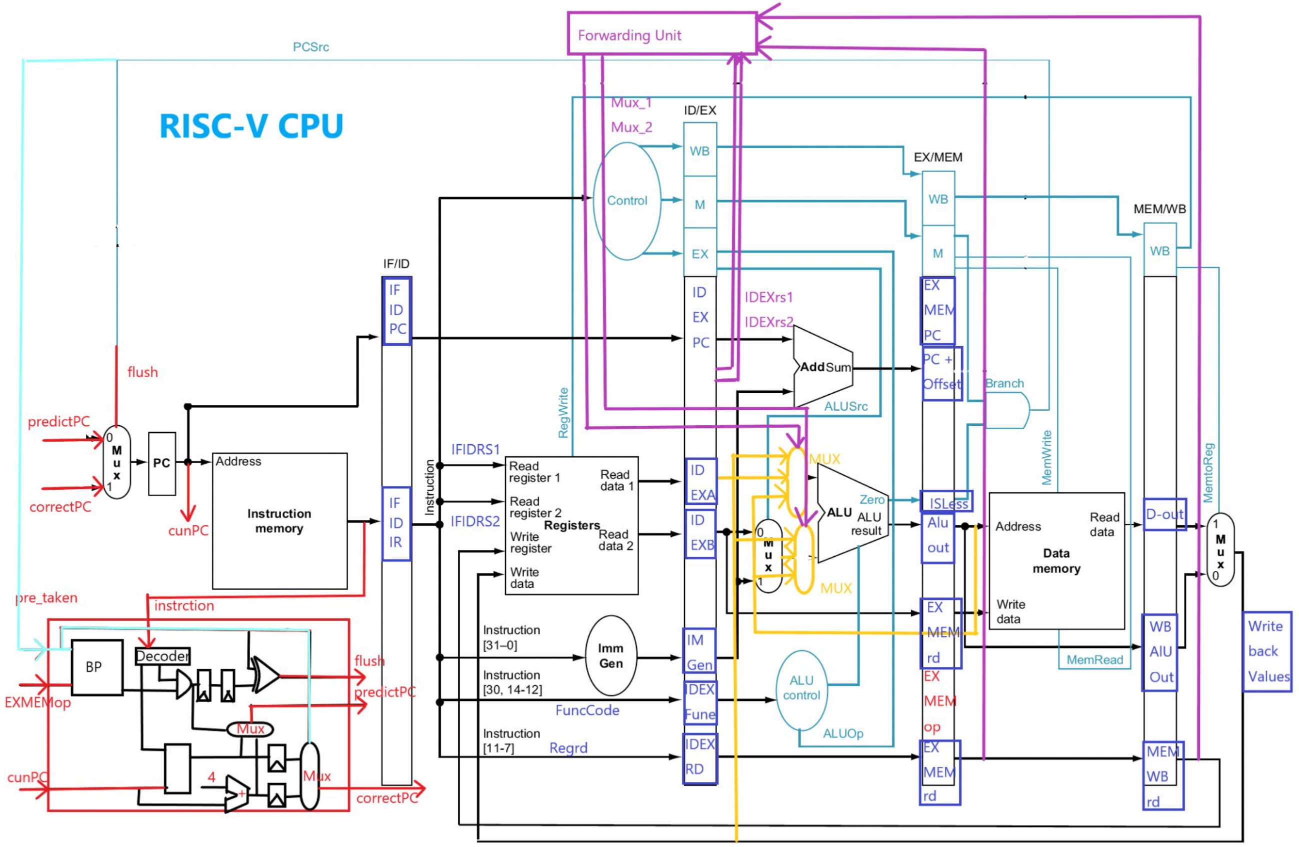
A 2-bit branch predictor is a type of dynamic branch prediction mechanism used in pipelined CPUs to predict the outcome of branch instructions. It takes as an input the result of the last branch instruction to change its prediction state for the next time a branch instruction is encountered. If later down the pipeline during the ALU stage the prediction is determined to be wrong, the pipeline is flushed and the correct instruction is fetched and executed.



PClogic

The PClogic module plays a critical role in determining the next instruction address (PC) in our CPU pipeline. Upon receiving and decoding the current instruction, the module first checks whether it is a branch instruction. If it is not a branch instruction, a simple addition of 4 is performed to produce the predicted PC, as the pipeline increments by 4 instructions by default.



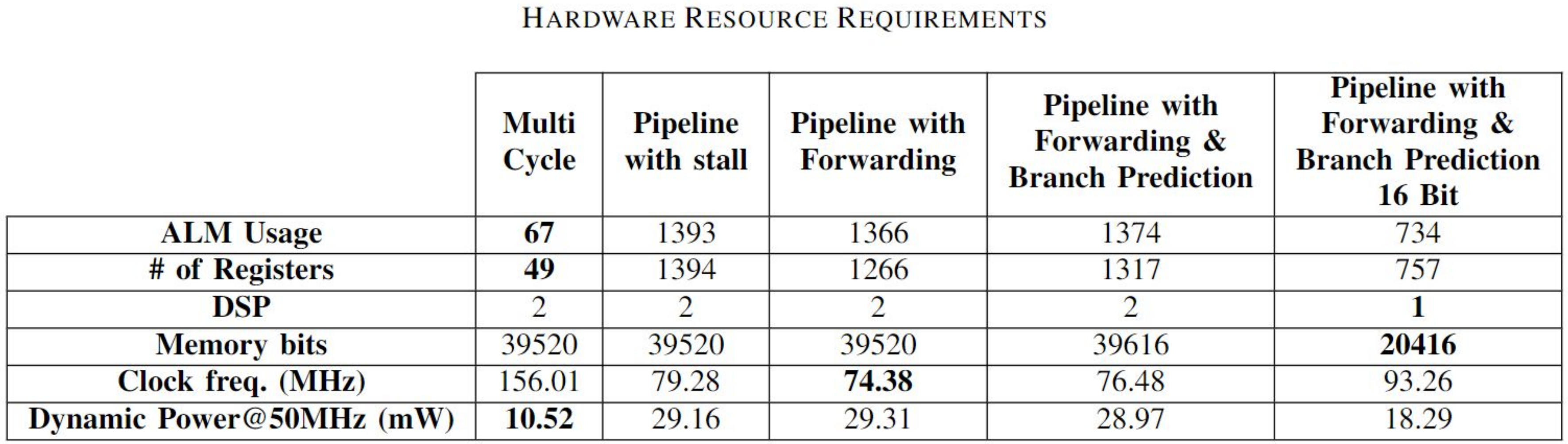


RISC-V CPU
Forwarding Unit
PCSrc
Mux_1
Mux_2
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
ID EX PC
IDEXrs1
IDEXrs2
AddSum
ALUSrc
Branch
RegWrite
MemWrite
MemtoReg
MemRead
flush
predictPC
correctPC
cunPC
pre_taken
instrction
EXMEMop
BP
Decoder
Mux
4
PC
Address
Instruction memory
IF ID PC
IF ID IR
Instruction
IFIDRS1
IFIDRS2
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
ID EXA
ID EXB
MUX
ALU
Zero
ALU result
ISLess
Alu out
EX MEM PC
PC + Offset
Instruction [31–0]
Instruction [30, 14-12]
Instruction [11-7]
Imm Gen
IM Gen
IDEX Func
IDEX RD
FuncCode
Regrd
ALU control
ALUOp
EX MEM rd
EX MEM op
Data memory
Read data
D-out
WB AIU Out
MEM WB rd
Write back Values

Performance Results: Matrix Multiplication

	Multicycle	Pipelined (Stalling)	Pipelined (Forwarding)	Pipelined (BP)
Clock Cycle Count	278431	124606	108552	**85014**
Instruction Count	**67713**	83456	83711	77013
Cycles Per Instruction (CPI)	4.111	1.493	1.297	**1.104**
Branch Prediction Accuracy	N/A	N/A	N/A	**94.785%**

	Multicycle	Pipelined (Stalling)	Pipelined (Forwarding)	Pipelined (BP)
Clock Cycle Count	452560	217116	190609	**164736**
Instruction Count	**109972**	149714	149714	151487
Cycles Per Instruction (CPI)	4.115	1.490	1.273	**1.087**
Branch Prediction Accuracy	N/A	N/A	N/A	**94.515%**



HARDWARE RESOURCE REQUIREMENTS

	Multi Cycle	**Pipeline with stall**	**Pipeline with Forwarding**	**Pipeline with Forwarding & Branch Prediction**	**Pipeline with Forwarding & Branch Prediction 16 Bit**
ALM Usage	**67**	1393	1366	1374	734
# of Registers	**49**	1394	1266	1317	757
DSP	2	2	2	2	**1**
Memory bits	39520	39520	39520	39616	**20416**
Clock freq. (MHz)	156.01	79.28	**74.38**	76.48	93.26
Dynamic Power@50MHz (mW)	**10.52**	29.16	29.31	28.97	18.29

Future Works

- More instruction coverage
  - Expand to more RV32 or even 64 bit instructions
- Cache
  - Keep frequently used instruction and data ready
  - Faster RAM access, increase performance
- Reduction of pipeline stages
- On-chip memory content verification
  - Verify Memory Content after program completion using more robust methods

THANK YOU

We really appreciate the support and guidance provided by Prof Akella and Thomas throughout the completion of our senior design project!