RISC-V CPU Design on FPGA

Yuan Li
University of California, Davis
Davis, USA
oyli@ucdavis.edu

Yibin Xu
University of California, Davis
Davis, USA
yibxu@ucdavis.edu

Bo Li
University of California, Davis
Davis, USA
obli@ucdavis.edu

***Abstract*—This paper is about a RISC-V CPU custom design that uses 4 different configurations (multi-cycle, 5 stage pipelined with simple stalling, 5 stage pipelined with forwarding, and 5-stage pipelined with forwarding and branch prediction) and 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.**

***Index Terms*—RISCV, accelerator, 5 stage pipeline, branch prediction**

I. Introduction and Background

Instruction set architecture, also called computer architecture, is an abstract model of a computer. The device that executes the instructions, such as a central processing unit, or a CPU, is an implementation of the instruction set architecture. Formerly, instruction set architectures were controlled in the hands of only a few companies. RISC-V was created to disrupt that status quo [1].

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. It was originally created by Professor Krste Asanović, and two of the graduate students he was leading: Yunsup Lee and Andrew Waterman for the Parallel Computing Laboratory at University of California, Berkeley, directed by Professor Davis Patterson. They received funding from Intel, Microsoft, other companies and the State of California. It was originally intended to advance parallel computing, not specifically a new instruction set architecture, but the team decided to create the entirely open source ISA. The free, simple, fast and the most importantly, the open source nature of RISC-V allows users world-wide to develop their own hardware that is tailored to their own needs and use cases. It serves as an international standard in the computing industry that allows it to be deployed in a variety of industries around the world [1].

II. Problem Definition

A. Breadth First Search Using Matrix-Vector Multiplication

Unlike traditional breadth-first search algorithms that uses queues to store and process graph nodes, the algorithm implemented in Algorithm 1 uses matrix-vector multiplication to compute the nodes that are reachable in each step.

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
```

A breadth-first search step can be performed by multiplying a sparse matrix $\mathbf{G}$ with a vector $\mathbf{x}$. The matrix $\mathbf{G}$ is the transposed adjacency matrix of the graph to be searched. The first step of BFS is to multiply $\mathbf{G}$ by the source vector. To search a node i, make $\mathbf{x}(i) = 1$ and every other entry 0. The result $\mathbf{y} = \mathbf{G}\mathbf{x}$ will pick the nodes that are reachable from the source node i. Multiplying the result by the input matrix again $\mathbf{y}' = \mathbf{G}\mathbf{y}$ will pick out the nodes that are reachable two steps away from the source node. Repeat this operation for $N - 1$ times, with N being the number of nodes in the input graph, will build a series of resultant vectors that contain nodes that are reachable in certain amount of steps from the source node [6]. The reason for the total number of steps being $N-1$ is that the maximum number of steps a node can be away from the source node is $N-1$. This is true when the input graph is a single line.

B. Matrix Multiplication

Matrix multiplication is one of the most fundamental mathematical operations in modern computing, and is often a performance critical component to many modern algorithms. Matrix multiplication can be implemented using various different approaches, such as using a basic loop-based algorithm or using SIMD (Single Instruction Multiple Data) architectures on hardwares that support it (e.g. GPUs), or using vectorized operations to speed up the algorithm.

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

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

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

III. Related Works

RISC-V is not a brand-new ISA but it is not as widely used in most processors today as Complex-Instruction-Set-Computer (CISC) s such as x86 from Intel or other RISC processors like ARM. However, the RISC-V architecture is gaining more and more popularity in machine learning and artificial intelligence applications because it's simple in design and low in power and area consumption, making RISC-V processors easy to deploy in embedded systems or IoT device fashion. There are many open source RISC-V processors on the internet, but all of them are geared towards a full-functionality processor that are used in the same situations as CISC processors.

Here are some examples of other RISC-V processors made for similar applications.

SiFive E20 Standard Core

The SiFive E20 Standard Core created by the company SiFive is a 2-stage, area and power optimized 32-bit RISC-V processor that supports the RISC-V RV32-IMC instruction set. It also supports hardware breakpoints accessible via JTAG ports, interrupts and system ports used for external memory access. It is typically used in MEMS sensors, digital motor control systems, low power MCUs and health care wearable monitors [8].

Codasip L11

The L11 is the smallest RISC-V core developed by Codasip. It is a power and area efficient, 3-stage pipelined RISC-V core that supports the RV32EMC instruction set architecture. It features a hardware multiplier and divider, and an interrupt controller for debugging purposes. This product is typically used for IoT devices, wearable devices, and in some cases running neural network algorithms using custom instructions [9].

OpenRISC mor1kx

The OpenRISC mor1kx IP core is one of the open-source implementations of the RISC-V instruction set architecture developed by the OpenRISC project. It is a configurable core implementation that could be either a 2-stage or 6-stage pipeline. It is developed to be high-performance and be as resource light as possible. It is also designed to be easily scalable and versatile. This product is typically used in embedded systems, high-performance networking and automotive applications [10].

The processor discussed in this paper focuses on the lightweight and easy-deploying nature of RISC-V architecture to make a processor that takes up very little hardware and consumes very low power.

IV. Our Approach

A. Multi-cycle

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

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. Furthermore, the RAM modules had to operate at a faster clock rate than the CPU itself, which is contrary to how memory is typically implemented in real systems. 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.

The multicycle CPU design comprises six states: `IDLE`, `IF`, `ID`, `EX`, `MEM`, and `WB`.

When the CPU is reset, all registers are set to 0, and the state is set to `IDLE`. The `IDLE` state is specifically designed to be entered only once after a reset, allowing the state variable to progress to the `IF` stage and the access to the instruction memory to complete for the first instruction for the program to be run.

During the `IF` stage, the program counter (PC) is incremented by 4, and the state transitions to the `ID` stage. The PC is obtained through combinational logic, and the 32-bit instruction is fetched from the instruction memory based on the PC value.

In the `ID` stage, the fetched instruction is first checked to see if it is a dummy "end of file" (EOF) instruction. The EOF instruction is designed to be 32 1s, or `32'hFFFF`. If it is, the "`done`" output signal is set to 1, and the state is held to halt the CPU from further execution. If it is not the EOF instruction, the values of rs1 and rs2 are extracted from the fetched instruction, and the value of `PC + PCOffset` is calculated in case the instruction is a branch instruction. The state is then transitioned to the EX stage.

The `EX` stage is where most of the calculations are performed, and the result is stored in `ALUOut` based on the instruction `opcode`, `funct3`, and `funct7`. Depending on the instruction type determined from the three parameters, the data is either read from or stored in the data memory during the `MEM` stage before transitioning to the `WB` stage.

The `MEM` stage is used exclusively for load and store instructions, where the data memory is being written to or read. With load instructions, `ALUOut` is used as the read address. In store instructions, `ALUOut` is used as the write address.

In the `WB` stage, the `ALUOut` result is written back to the register file. Most instructions require four clock cycles to complete, skipping the `MEM` stage and going to this stage immediately after the `EX` stage. With store instructions, this stage is skipped since there is no need to write back to the register file.

2) 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
```

3) Performance: 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. In the multicycle CPU design, we successfully achieved a CPI (Cycles Per Instruction) of 4.11. This improvement was possible because the majority of instructions do not require the `MEM` stage except for load instructions where the data memory is being accessed. As a result, the CPI has been reduced from 5 to 4.11, as shown in Section V.

4) Hardware Resources: 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.

B. 5-Stage Pipelined with Simple Stalling

In Figure 1, we present a standardized 5-stage pipelined RISC-V processor. 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 smooth data flow and maintains the temporal order of

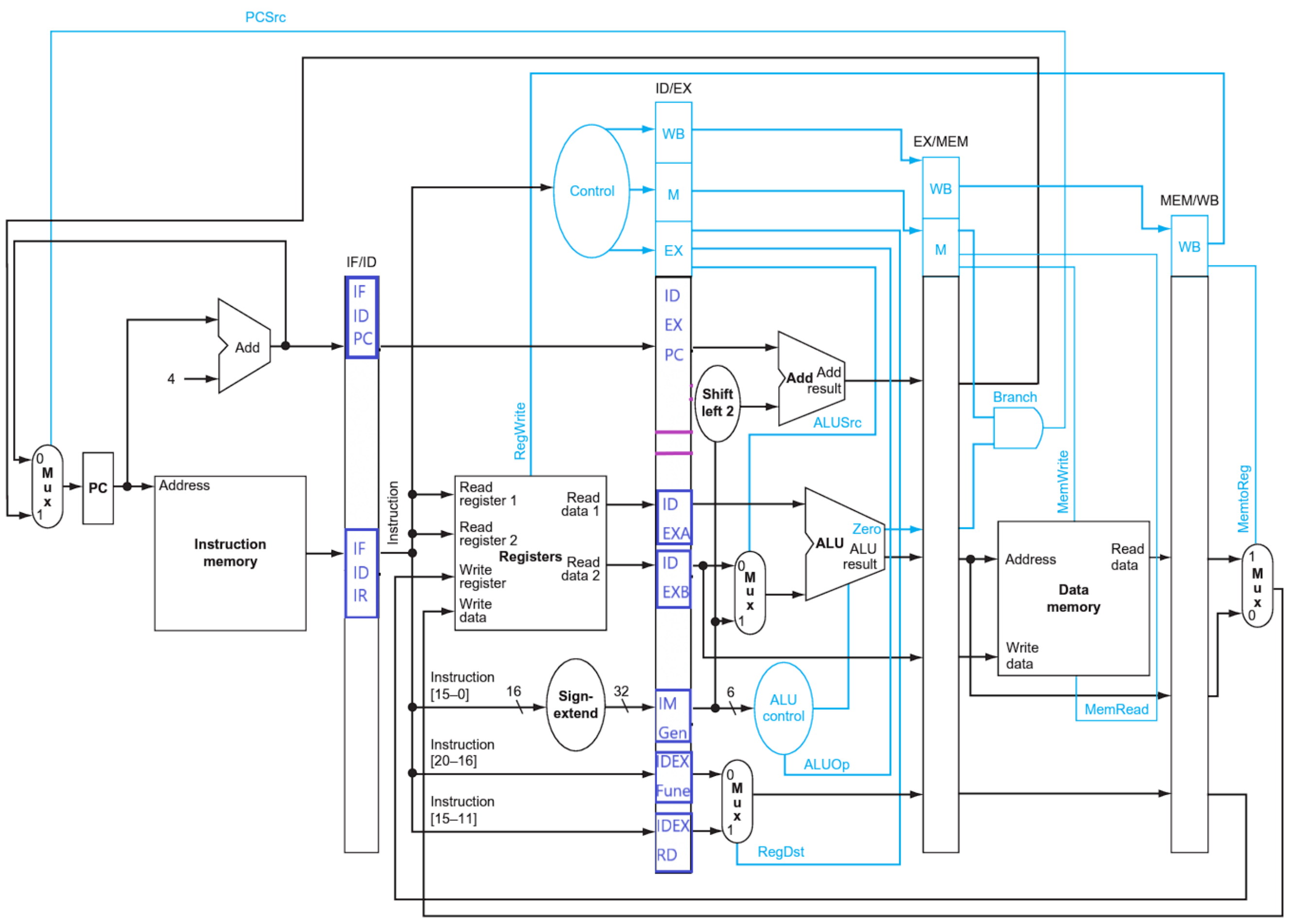

Fig. 1. 5-Stage Pipelined with Stalling [2]

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 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), and the ALU itself. Detailed explanations of each module and the functioning of the pipeline will be provided in subsequent paragraphs.

1) `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 truth table I:

The special NOP type instruction is added which was not mentioned in the Henessey textbook since we need to inject a “bubble” into this module to stop the operation of parts of the processor that are controlled by this module. Note that the ‘ALUop’ signal for NOP is specifically chosen to be `ZZ` since any other value would cause the ALU to go into a state that does some other types of instruction since there are only two bits to work with in this control line.

2) `ALU` Module: 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 Section IV-B1. There is a special output register called “`IsLessThan`” and it is used for branch instructions address calculation. Since the only branch instruction used in our assembly program is ‘`blt`’ or “branch less than”, the “`IsLessThan`” register is asserted when the first input operand is less than the second. The ALU does the following logic with input operands `Ain` and `Bin` in Table II:

3) `ALUControl` Module: This module is another combinational module that takes in two inputs, the

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

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	$A \times B$

"ALUop" signal from the control module in Section IV-B1 and the funct7 and funct3 portions of the instruction that is being executed. Together, they determine the specific kind of operation the ALU should be doing. Some kinds of instructions require the ALU to do the same kind of operation, so a "casex" statement is used. It outputs the ALUCtl signal which determines the operation of the ALU itself. It follows the truth table described in Table III:

4) Pipeline Registers: Since the control signals from both the control and the ALU control modules are purely combinational, they need to be registered each clock cycle for each instruction since with pipelining, each cycle the instruction requires different parts of the processor to do different operations. The flow of the control signals is essential since they ensure that the processor's components are serving the correct instructions and with the correct data. The "ALUop" and "ALUSrc" signals are used during the EX stage, so they need to be registered once. The "MemtoReg", "MemRead", "MemWrite", and "Branch" signals are used in the memory stage, so they need to be registered twice. Finally, the "RegWrite" and "MemtoReg" signals are used in the WB stage, so they need to be registered three times before it is used.These registers are depicted as light blue rectangles in Figure 1.

5) Incorporation with RAM: With a synchronous RAM module, after the data lines are provided with an address, the output is ready on the next positive edge of the clock, which needs to be addressed so as to not introduce a mismatch in timing from when the output from the RAM is expected to be ready and used. The two occasions that deal with this situation are when fetching instructions and retrieving the output from data memory when doing load word instructions. Thus, the output register of the instruction memory and data memory are directly connected to the components that use them instead of being registered again, which would introduce a 1 cycle delay. This is depicted as combining the output of instruction memory and data memory with the registers proceeding them in Figure 1.

C. Forwarding

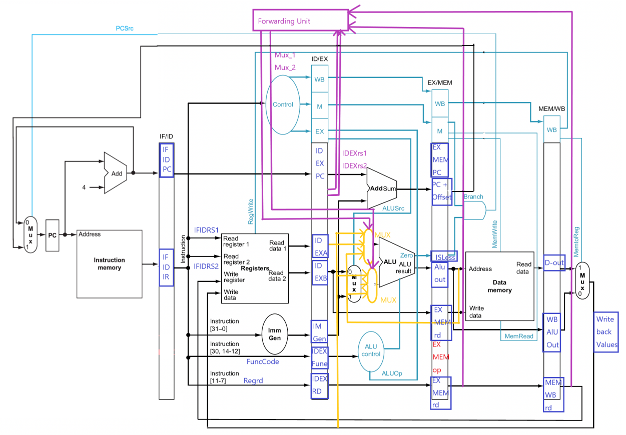

Fig. 2. Forwarding Module [2]

The Forwarding module depected in Figure 2 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

TABLE III
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

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. There are two types of situations when forwarding can help avoid stalling: the data-hazard, and the load-use hazard. In each of the two types, the instructions at play could be up to 2 instructions apart (the first instruction has PC n, the second instruction having PC n + 12), when the first instruction has not completed its write-back stage and its write back register is the same as one of the source registers of the second.

Depending on the spacing between the two instructions that cause the hazard, forwarding can route data from different parts of the processor, either from the ALU or from the data memory.

This module specifically focuses on reading the current `rs1` and `rs2` values, which are the source registers from the instruction during the execution and memory stages and comparing them with `EXMEMrd` and `MEMWBrd`, the destination registers in the memory and write-back pipeline stages. It checks for the following conditions:

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

to determine whether data from `EXMEMrd` or `MEMWBrd` needs to be forwarded to IDEXrs1 or IDEXrs2 based on which of the above conditions is met. Once the condition is true, the module identifies the correct return value between `IDEXrd` and `EXMEMrd`. It then communicates this information to the multiplexer (mux) located in front of the ALU modules, instructing it to switch its input source among `ALUout`, `write_back_val`, `IDEXA`, or `IDEXB` based on the selected return value.

D. 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 speculatively continuing the execution based on that prediction.

1) `BranchPredictor` module: 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. The state diagram of a 2-bit branch predictor is shown in Figure 3. When a branch instruction is encountered, the branch predictor consults the counter associated with that branch to make a prediction. If the counter is in the strongly taken or weakly taken state, the predictor predicts that the branch will be taken and speculatively continues execution down that path. Conversely, if the counter is in the strongly not taken or weakly not taken state, the predictor predicts that the branch will not be taken. When the latest branch instruction is computed by the `ALU` module, the branch predictor will adjust its state based on whether the latest branch was taken. If the latest branch is taken, state register (which is 2 bits, hence the name 2-bit branch predictor) inside the branch predictor will increment by 1 (except when already in strong taken state), and if the latest branch is not taken, the state register will decrement (except when already in strong not taken state). In our programs, we know that most branch instructions will be taken so we decided to start our branch predictor in the strong taken state to maximize branch prediction accuracy; this of course can easily be changed.

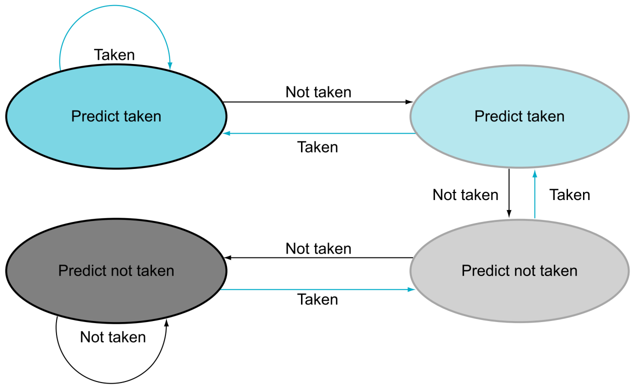

Fig. 3. State Diagram of the 2-bit Branch Predictor [2]

2) `PClogic` *module:* The internal layout of this module is shown in Figure 5. This module takes several input variables:

- `current_PC`
- `prev_taken`
- `instruction`

It produces three output signals:

- `flush`
- `predict_PC`
- `correct_PC`

It is also responsible to produce the number of times the branch predictor was wrong in its prediction and the total number of times when a branch instruction is encountered. Therefore, it additionally has the following two output signals:

- `TotalBranches`: specifies the total number of times a branch instruction is executed.
- `timeswrong`: specifies the number of times the branch predictor made a wrong prediction

The behavior of the program counter (PC) depends on the `flush` signal. When the `flush` signal is high, it indicates that our prediction was incorrect. In such cases, we need to consider the `correct_pc` as the input value for the PC. Additionally, before entering the `EX` stage, all data in the `IF` and `ID` stages must be flushed. This behavior is illustrated in Figure 4.

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.

In the case of a branch instruction, the `PClogic` module utilizes the `BranchPredictor` module in

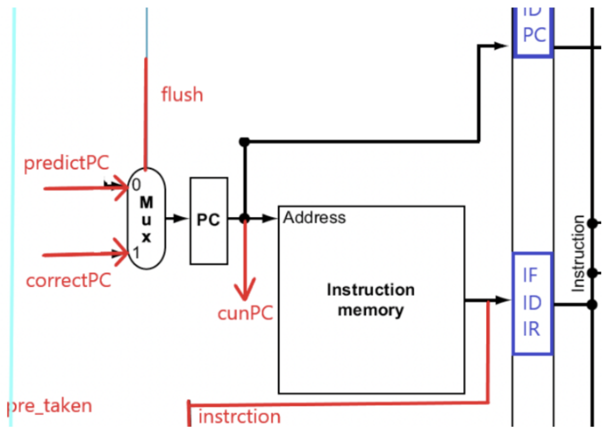

Fig. 4. Picking the Selected Input Address [2]

Section IV-D1 to make a prediction. Depending on the chosen branch prediction algorithm, two scenarios are considered. If the prediction is "not taken," the module computes the predicted PC as $\texttt{current\_pc} + 4$. If the prediction is "taken," the predicted PC is calculated as $\texttt{current\_pc} + \texttt{PCOffset} - 4$.

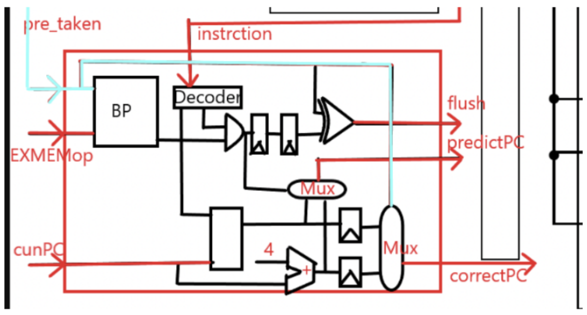

Fig. 5. Internal Layout of the `PClogic` module

However, after the execution stage of the branch instruction, the CPU gains knowledge of the correct PC. At this point, the CPU compares the correct PC generated by the EX stage with the previously predicted PC. If the prediction was incorrect, the `PClogic` module triggers a process known as "flushing," which clears all data in the instruction fetch (`IF`) and instruction decode (`ID`) stages. The correct PC is then adopted as the new input for the PC, ensuring the pipeline proceeds with the correct sequence of instructions.

E. Complete Design

With the forwarding and branch prediction module added, the complete diagram of the processor is depicted

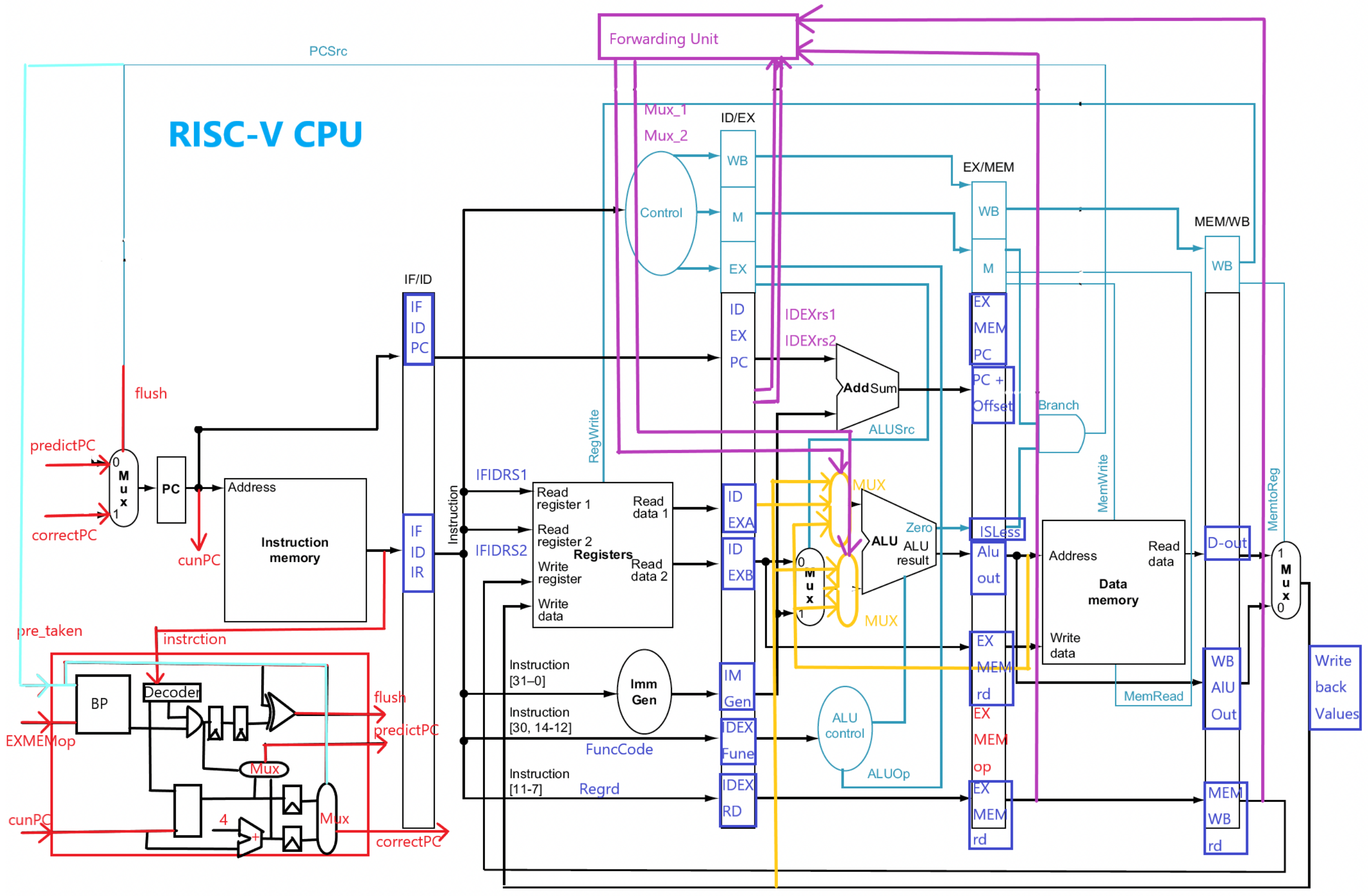

Fig. 6. Complete Layout of RISC-V Processor [2]

in Figure 6. The forwarding unit will handle data hazards and the `PClogic` module will handle the progression through the entire program.

V. Results

The results of the four proposed processor designs are analyzed in the following parameters: performance and hardware requirements. The performance of the processors is determined by the clock frequency, the number of instructions executed, and the total number of clock cycles it takes to complete the benchmark programs of BFS and matrix multiplication.

A. Benchmark Programs

The BFS program uses a randomly generated directed graph with 24 nodes and unweighted edges. The edges are unweighted because the current implementation of the BFS algorithm uses matrix-vector algorithm to solve the BFS problem. This approach does not produce any meaningful information in the resulting vectors of the BFS program. The minimum number of edges from a node is 3, and the maximum number of edges from a node is 10. The maximum number of edges is chosen relatively low since if there are too many edges that starts from one node, the BFS program would take very few rounds to complete the traversal through the whole graph.

The input matrices of the matrix multiplication program are both 20×20 and has both positive and negative integers. Note that with a 16 bit processor, the range of values that can be computed is limited compared to the standard 32 bit register processors. Specifically, a 16 bit processor can work with values in the range $[-2^{15}, 2^{15}-1] = [-32768, 32767]$, whereas a 32 bit processor can work with values in the range $[-2^{31}, 2^{31}-1] = [-2\,146\,483\,648, 2\,146\,483\,647]$.

B. Performance Results

1) Matrix Multiplication: The performance results for matrix multiplication are shown in Table IV.

2) Breadth First Search: The performance results for breadth first search are shown in Table V.

TABLE IV
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%**

TABLE V
PERFORMANCE RESULTS: BREADTH FIRST SEARCH

	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%**

C. Hardware Cost Results

The hardware resources costs are tabulated in Table VI using the matrix multiplication program as the input. The BFS program would yield similar results except for the number of memory bits used.

The DSP block usage represents the multiplier used on the FPGA to achieve multiplication operations; they are much more efficient compared to using ALM to make a multiplier.

Using timing analysis, the critical path (the path that takes the longest time to complete) of all processor designs are determined. In the multicycle design, the critical path is the path where the data memory is accessed, corresponding to load instructions. In pipelined designs, the critical path is the path through branch address calculations, corresponding to branch instructions. This is because in a pipelined design, the result of the effective address calculations goes through the longest delay of combinational elements, whereas in the multicycle design, the branch target address is calculated in the execution stage and the next instruction immediately starts in the next clock cycle, resulting in branch instructions taking only 3 clock cycles.

VI. DISCUSSION OF RESULTS

It is clear that as we progress, the performance of the processor is increasing, but the performance improvements are not without costs, and the primary cost is physical resources clock speed slow downs.

A. Multicycle

The simplest implementation, the multicycle processor, requires very little logic units and registers, and can run on almost double the clock frequency as the pipelined versions. However, it takes 4 cycles to complete most instructions, and with load instructions it takes 5 cycles. Although the clock frequency in a multicycle design is high compared to pipelined versions, a typical matrix multiplication program would take about 3 times the clock cycles as a 5-stage pipelined processor. A multi-cycle processor therefore is not preferred when performance is of any concern. The advantage of a multicycle processor is that it consumes very little area and power, making it potentially useful in situations where space and energy is highly limited and performance is of no priority. Since every instruction is completed before the next is fetched, there is no data hazard in a multicycle implementation.

B. 5-Stage Pipeline with Simple Stalling

In the simple pipelined version with stalling to handle all data hazards, the clock frequency reduces to about half the frequency of a multicycle processor, however, the CPI improves greatly with a 93.66% improvement. The cost is much more hardware consumption, but in our FPGA, still, a small portion of the hardware available is needed. Since data hazards can be resolved using little more hardware, a processor with only stalling is not ideal in any situation. The power consumption, number of ALMs, and number of registers of 5-stage pipelined with simple stalling are the highest among all the implementations discussed in this paper. This could be the result of the high amount of hardware used to detect hazards that warrant stalling the CPU during the programs.

TABLE VI
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

C. 5-Stage Pipeline with Forwarding

With forwarding, we can resolve most data hazards with only a little more hardware, in this case (percent increase in ALM). Forwarding again improves CPI by 15.71% but without sacrificing clock speed, area or power.

D. 5-stage Pipeline with Branch Prediction

Finally, the 2-bit branch predictor is able to bring CPI down to nearly 1. In a single-core processor, the theoretical minimum CPI is 1, which would correspond to every instruction taking 1 clock cycle to complete. The `NewPClogic` module is a wrapper around the 2-bit branch predictor, which means that we can swap out and replace with another branch predictor if needed. Again, branch prediction does not require much more hardware resources and does not impact clock frequency (some percentage in our design).

E. Changing the Register Size

To even further reduce hardware and energy consumption, the register size in the processor is reduced to 16 bits from the standard 32 bits. The number of registers remains at 32. This reduces registers used on the FPGA and reduces memory bits required to store the data since every entry is half the size. With less hardware, it is possible to increase clock frequency by (some percentage) compared to the 32 bit version, and it consumes (some percentage) less power and (some percentage) less area.

The reduction in register size is a reasonable design choice since in a lightweight or embedded systems situation, the precision of data is often not the biggest concern. Reducing power consumption and cost is more beneficial compared to the loss of some data accuracy.

Assumptions

The assumptions in our design decisions and progression include favoring cost and power consumption over throughput and performance. We are also assuming that less logic and memory utilization and area decidedly means less cost when making such a processor. The actual production of the processor is not within the scope of this paper.

VII. FUTURE WORK

A. More coverage in instruction support

Currently, our processor only supports a subset of the R32I instructions based on time constraint and resource concerns. In the future, support for more instructions can be added to make more programs able to run on the processor. This would inherently cost more hardware so any new instructions could be determined by the program itself. More complex instructions like jump and link, return calls are not supported in this processor either.

B. Cache

A cache can be used to further accelerate the execution of a program since frequently used data can be stored in such a way that would not incur a one clock-cycle delay between the time when the read address of memory becomes valid and when the output data is valid.

C. External access for FPGA memory contents

Currently, all of results from programs are simulated using ModelSim. Using system ports on the FPGA to access the memory contents will allow us to verify the correct functioning of our design on a hardware level, instead of purely relying on software simulations. This can also be done through the memory content viewer in the Quartus Prime software that we are using to compile and synthesize the processor designs.

D. Changing of Pipeline Stages

One possible optimization is to change the number of pipeline stages, perhaps even parameterize it.

To further reduce the hardware costs of pipeline, 2 or 3 pipeline stages should reduce the amount of hardware required. Increasing to 6 stages will further increase the performance since more pipeline stages allows the processor to operate on more instructions at the same time, increasing performance. Some open source designs referenced in Section III have configurable pipeline stages to tailor to the specific needs of the user to balance out performance and hardware resources requirements, and can be a useful addition to our design as well.

Acknowledgment

We would like to express our utmost gratitude for Prof. Venkatesh Akella and teaching assistant Thomas Abbott for their invaluable assistance and feedback throughout the entirety of our project.

References

[1] "History," RISC-V International. https://riscv.org/about/history (accessed Feb. 16, 2023).

[2] D. A. Hennessy, COMPUTER ORGANIZATION AND DESIGN RISC-V EDITION : the hardware software interface. S.L.: Morgan Kaufmann Publisher, 2021.

[3] Riley, K. F.; Hobson, M. P.; Bence, S. J.. Mathematical methods for physics and engineering (2010).

[4] Thomas H. "22.2 Breadth-first search". Introduction to algorithms (accessed Feb. 16, 2023).

[5] M. Verleysen, Proceedings: ESANN 2015. Presses universitaires de Louvain, 2015. Accessed: Feb. 16, 2023. [Online].

[6] V. B. Shah, J. Gilbert, and S. Reinhardt, "4. Some Graph Algorithms in an Array-Based Language," Graph Algorithms in the Language of Linear Algebra, pp. 29–43, Jan. 2011, doi: https://doi.org/10.1137/1.9780898719918.ch4.

[7] Y. Zhang, P.-A. Tsai, and H.-W. Tseng, "SIMD2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM," arXiv:2205.01252 [cs], Aug. 2022, Accessed: Feb. 16, 2023. [Online]. Available: https://arxiv.org/abs/2205.01252

[8] "E20," SiFive. https://www.sifive.com/cores/e20 (accessed Jun. 06, 2023).

[9] "L11," Codasip. https://codasip.com/products/l11/# (accessed Jun. 06, 2023).

[10] "mor1kx - an OpenRISC processor IP core," GitHub, May 30, 2023. https://github.com/openrisc/mor1kx/blob/master/doc/mor1kx.asciidoc (accessed Jun. 06, 2023).