Coen 210: Computer Architecture Design Project

Performance via Parallelism

by

Gousia Sultana Syeda & Alex Whelan

University of Santa Clara

Project Advisor: Professor Peter Nghiem

1. **Table of contents.**
2. **Assumptions**
3. **Instruction Set Architecture**
    3.1) Instructions
    3.2) Instruction Format
    3.3) Addressing Modes
    3.4) Discussion
4. **Application**
    4.1) Assembly program(s) of the benchmark
    4.2) Emulation of RISC-V instructions
5. **CPU organization**
    5.1) Control Logic
    5.2) Datapath(s)
    5.3) List of all hardware blocks
6. **Performance Analysis and Discussion**

2. Assumptions

Benchmark

1. The size of the array is provided as a parameter.
2. The register x5 is initialized to zero prior to running the benchmark.

Datapath

1. Forwarding and Data Hazard unit for existing RISC-V instructions are present but not shown in the schematic.

Analysis

1. Assume the array length is a multiple of 32 to simplify math calculations for instruction count versushaving to do ceil(n/32) when arrays are not multiple of 32.
2. We perform a worst case analysis, i.e. the element is not found in the array.

3. **Instruction Set Architecture (ISA)**

3.1 Instructions

a) Supported Instructions: add, sub, addi, ld, sd, and, or, nor, andi, beq, jal, jalr

b) New Instructions: beqv, lv

Instruction Type	Instruction(s)
R	add, sub, and, or, nor
I	addi, andi, l[d|h|w|...], jalr, lv
S	s[d|h|w|...]
SB	b[eq|ge|ne|...], beqv
UJ	jal

3.2 Instruction Format

Instruction Type	31-25	24-20	19-15	14-12	11-7	6-0
R	func7	rs2	rs1	func3	rd	Opcode
I	imm[11:0]		rs1	func3	rd	Opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	Opcode
SB	imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	Opcode
UJ	imm[20|10:1|11|19:12]				rd	Opcode

3.3 Addressing Modes

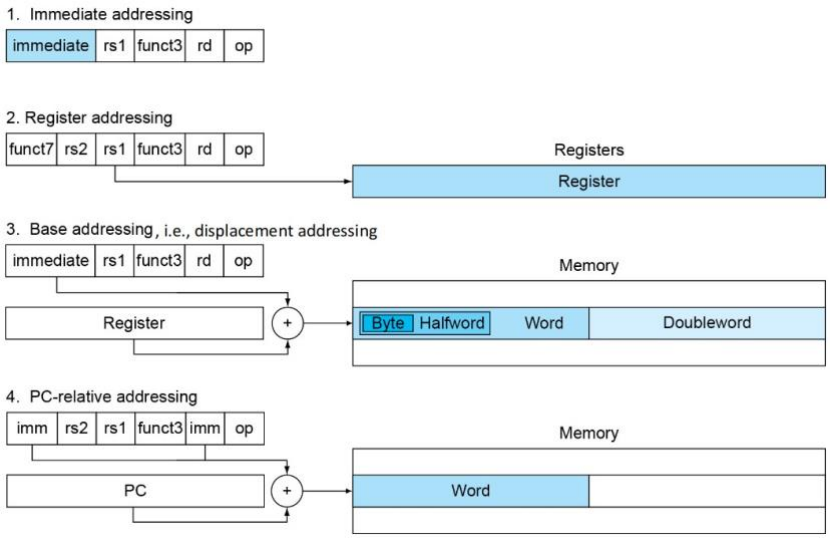
a) Immediate

b) Register

c) Base/Displacement

d) PC-Relative

e) Non-unit (constant) stride for load vector (lv)



https://passlab.github.io/CSCE513/notes/lecture04_RISCV_ISA.pdf

3.4 Discussion

The ISA presented aims to improve on the existing RISC-V architecture by adding new instructions without redefining an already excellent architecture. We want the programmer or designer that is acquainted with RISC-V to be comfortable and familiar with the new improved version. The design challenge in modern processors is the ability to overcome the power-wall. We are able to accomplish this task by increasing performance using parallelism. Specifically, we are attempting to increase throughput by decreasing the instruction count, clock cycles, and therefore decreasing the total execution time of the benchmark program.

Pros: The improvements are accomplished by adding two new hardware blocks, the vector load unit and the branch vector functional unit. The vector load unit is a substitute for the load instruction as it is able to retrieve 32 values from the cache in a single clock cycle. This hardware block can also be scaled to retrieve more than 32 values and it can also be modified to store values in parallel as well. The second improvement is with the branch vector functional unit. This unit currently supports a membership test that checks if a given key value is present in a single vector of 32 values. It returns the index in the vector or zero if the value is not present. Once again, this unit can be scaled to support other operations such as vector addition/subtraction, scalar multiplication, etc.

Cons: The added hardware blocks in their current state are sparse and not generalized. In other words, they serve the purpose of making the benchmark fast but are not yet applicable to other benchmark programs such as matrix multiplication which can benefit greatly. For example, the load vector command lv vd, base-address does not store the vector in the register file or a vector file. The format of the instruction is identical to the ld instruction to make it inline with the RISC standard but there is only one vector available and it is nested inside the vector load unit. This design decision was made to illustrate the power of parallel processing at the expense of making is more general. Similarly, for the beqv instruction beqv rs1, rs2, jump-address the vector is forwarded from the load vector unit. It is implicitly used in this situation with only the designer

knowing the correct use of this instruction. rs2 is a general purpose register to hold the result of the membership test when comparing rs2 with vector elements. In reality, the vector functional unit should be able to perform ample operations on vector-vector or vector-register. Lastly, parallelism comes at a cost of additional hardware. In the vector load unit we have to have 32 data caches which must be consistent with the main data cache in case a value in the array is changed with a store instruction in the middle of the benchmark. However, this type of hazard is resolved. Similarly, the branch vector functional unit adds 32 comparators to be able to perform the operations in parallel. A true cost-benefit analysis must be performed if we wish to justify adding this much hardware. Luckily, we can scale the parallel operations down to 4 or 8 if the performance metrics are not inline with the cost.

4. Application

4.1) Assembly program of the benchmark

Baseline

```
.section .text
.align 2
.globl asm_find
.type asm_find, @function
# int A[] = {1,2,5,9};
# x10 -> base array, x11 -> size, x12 -> key
# x5 : index # x6 : value asm_find:     lw
x6, 0(x10)         # v = A[i]     addi x10,
x10, 4      # increment A[i+1]      addi x5,
x5, 1        # i++
    beq x6, x12, found    # if(A[i] == key) then return
beq x5, x11, Exit     # if i == size then Exit      beq
x0, x0, asm_find # goto top of loop found:      add
x10, x5, 1        # return index + 1 if found      jal
x1                Exit:      add x10, x0, x0       # if
Not found return 0      jal x1
```

Optimized

```
.section .text
.align 2
.globl asm_find
.type asm_find, @function
# int A[] = {1,2,5,9};
# x10 -> base array, x11 -> size, x12 -> key
# x5 : index # x6 : value asm_find:     lv v1,
0(x10)            # v1 = A[i...i+31]
    addi x10, x10, 128       # incr. base addr. A+128 (4 bytes * 32 elements)
    beqv x127, x6, found     # CMP(membership) key == v1 (v1 is implicit), store match index or 0 in x127
addi x5, x5, 32          # i += 32
    bge x5, x11, Exit        # if i >= size then Exit      beq x0, x0, asm_find     #
goto top of loop found:      add x10, x5, x127        # move to result
iteration(0,32,64,...n) + found index      addi x10, x10, 1         # return index +
1 if found      jal x0, x1 Exit:      add x10, x0, x0          # if Not found return
0      jal x0, x1
```

4.2) Emulation of RISC-V instructions

All instructions are a subset of the RISC-V architecture except for **nor** which can be implemented by negating the result from the **or** instruction. In other words, R[rd] = ~(R[rs1] | R[rs2]) this is considered a pseudo instruction. The two new instruction, beqv and lv, are emulation of the beq and lw, respectively. 1 beqv or lv instruction is the equivalent to 32 beq or lw instructions.

5. CPU organization

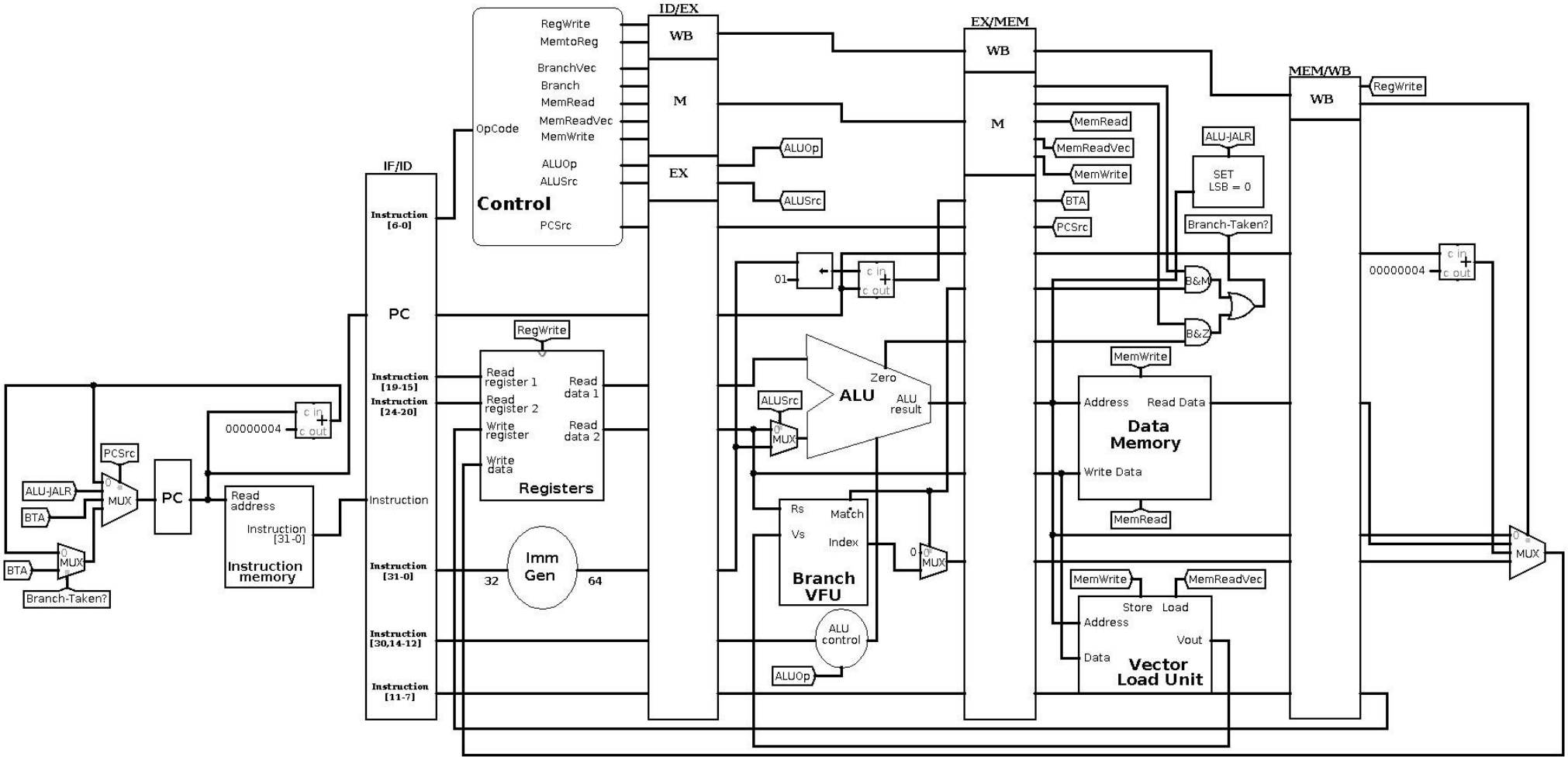
5.1 Control Unit Logic

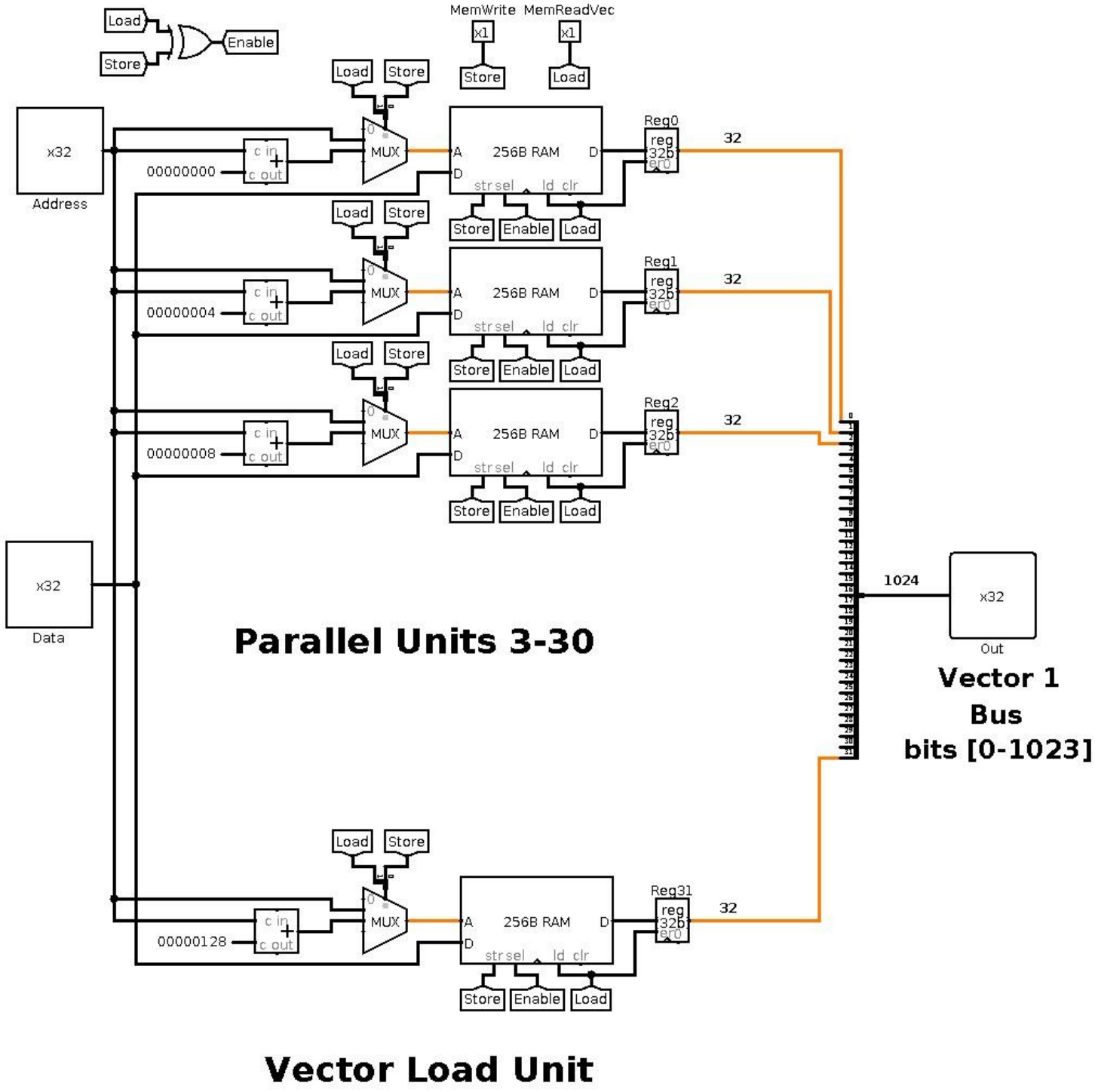
	EX		MEM						WB	
Instruction	ALUOp	ALUSrc	BranchVec	Branch	MemRead	MemReadVec	MemWrite	PCSrc	RegWrite	MemtoReg
R-format	10	0	0	0	0	0	0	00	1	00
addi, andi	00	1	0	0	0	0	0	00	1	00
ld	00	1	0	0	1	0	0	00	1	01
lv	00	1	0	0	0	1	0	00	0	XX
jalr	00	1	0	0	0	0	0	01	1	10
jal	XX	X	0	0	0	0	0	10	1	10
sd	00	1	0	0	0	0	1	00	0	XX
beq	01	0	0	1	0	0	0	11	0	XX
beqv	XX	X	1	0	0	0	0	11	1	11

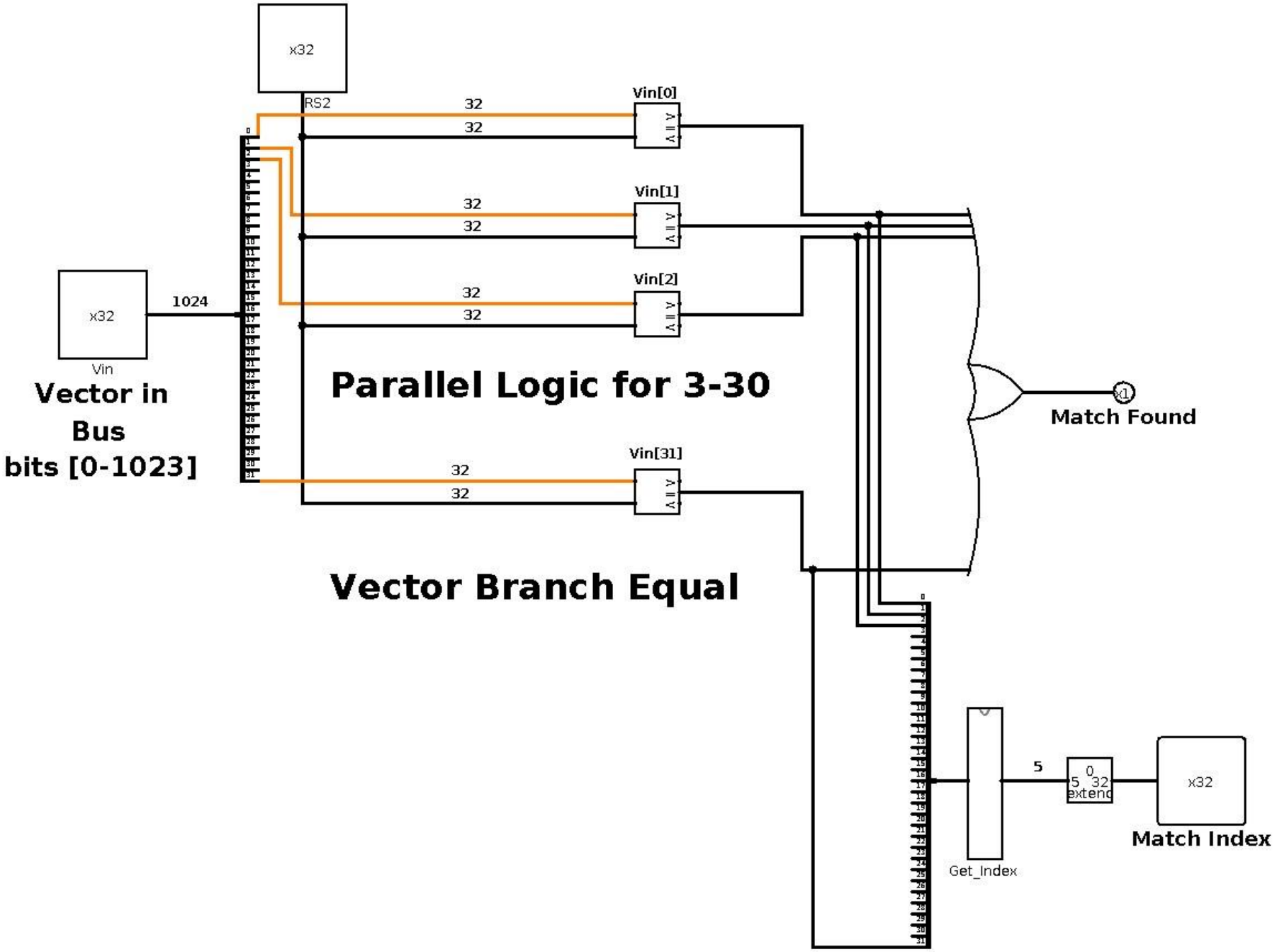
Instruction	OPCODE	FUNC3	FUNC7 or IMM
add	0110011	000	0000000
sub	0110011	000	0100000
addi	0010011	000	imm
ld	0000011	011	imm
sd	0100011	011	imm
and	0110011	111	0000000
or	0110011	110	0000000
nor	0110011	110	1111111
andi	0010011	111	imm
beq	1100011	000	imm
jal	1101111	imm	imm
jalr	1100111	000	imm

beqv	0000000	imm	imm
lv	1111111	111	imm

Datapath


ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
RegWrite
MemtoReg
BranchVec
Branch
MemRead
MemReadVec
MemWrite
ALUOp
ALUSrc
PCSrc
OpCode
Control
Instruction [6-0]
PC
Instruction [19-15]
Instruction [24-20]
Instruction
Instruction [31-0]
Instruction [30,14-12]
Instruction [11-7]
00000004
c_in
c_out
PCSrc
ALU-JALR
BTA
Branch-Taken?
MUX
Read address
Instruction [31-0]
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Imm Gen
32
64
01
ALUSrc
Zero
ALU
ALU result
Rs
Vs
Match
Index
Branch VFU
ALU control
ALUOp
MemRead
MemReadVec
MemWrite
BTA
PCSrc
ALU-JALR
SET LSB = 0
Branch-Taken?
B&M
B&Z
Address
Read Data
Data Memory
Write Data
Store
Load
Address
Vout
Data
Vector Load Unit
RegWrite
00000004


Load
Store
Enable
MemWrite
MemReadVec
x1
x1
Store
Load
x32
Address
00000000
00000004
00000008
00000128
c in
c out
MUX
0
A
D
256B RAM
str sel
ld clr
Store
Enable
Load
Reg0
Reg1
Reg2
Reg31
reg
32b
32
x32
Data
1024
x32
Out
Parallel Units 3-30
Vector 1
Bus
bits [0-1023]
Vector Load Unit


x32
RS2
32
32
Vin[0]
Vin[1]
Vin[2]
Vin[31]
x32
1024
Vin
Vector in
Bus
bits [0-1023]
Parallel Logic for 3-30
Vector Branch Equal
x1
Match Found
5
0
5 32
extend
x32
Match Index
Get_Index

5.3 Hardware Blocks

1. PC: A 32 bit counter that provides the address to instruction memory.
2. Instruction Memory: A ROM containing the instructions of the program.
3. Control Unit: Fully logical unit implemented with logic gates that takes as input the lowest 7 bits of an instruction and outputs the control signals to the processor.
4. Register File: Contains 96 32 bit registers numbered 0-31 and 63-127. The registers 0-31 are used exactly as in the RISC-V architecture with registers 63-127 are used as temporary or general purpose registers.
5. Immediate Generator: Takes as input the 32 bit instruction and splices the immediate values into correct positions, sign extending the 32 bit value to a 64 bit value in the process.
6. ALU: The arithmetic logic unit performs various operations on the two input values. Some operations are addition, subtraction, multiplication, division, comparison (equal, less than, greater than, etc), and logical operations (or, and, xor, nor, etc)
7. ALU Control: A unit that generates the control signals for the ALU corresponding to the appropriate operation the ALU must perform on the input. It takes as input the func3 field and ALUOP and outputs a 4 bit control signal.
8. BTA: The bit shifter and adder form the branch target address used in jump and branch instructions.
9. Branch VFU: The vector functional unit performs a membership test on a vector with a value provided in Rs. The vector input is a bus of 1024 bits representing the values from the 32 registers that make up a single vector. Rs is a source register in fact it is the second operand in the instruction format R[rs2]. The bit lines of the vector bus are segmented and using a comparator are evaluated against the register value of Rs. These operations are done in parallel, in other words only taking a single clock cycle. The output of the comparators generate 2 signals a 1 bit match signal and a 32 bit match index value corresponding to the register file of the vector which matches Rs.

9b. Get_index: This block is implemented using logic gates or a ROM that takes the 32 1-bit signals from the comparators and outputs a 5 bit signal corresponding the the register index where the match occurred it is then extended to 32 bits.

10. Data Memory: A Data cache containing values used by the program. It takes as input the address of the value and data and performs a load or store according the the MemRead or MemWrite signals respectively.

11. Vector Load Unit: This unit acts as a Data Memory but is used specifically in vector operators to improve performance in the benchmark. The inputs are the address, data, MemReadVec and MemWrite signals. In a load it uses non-unit constant stride of length 32 to get 32 consecutive word values from the memories and stores them each in registers 32-63 corresponding to vector 1. If a store occurs consistency must be maintained between the main data memory and the vector memory that is why all 32 caches must be updated. These operators are done in parallel so they also only take one clock cycle to complete. The registers are enabled only on a vector load instruction so they maintain their values between clock cycles. The registers form a bus line of 1024 bits that is forwarded to the Branch VFU to check for membership when required.
12. IF/ID, ID/EX, EX/MEM, MEM/WB: These blocks are pipes. They contain registers of varying length that store control signals or intermediate data between clock cycles.

6. Performance Analysis and Discussion

Clock Cycle Times

	Baseline	**Optimized via Parallelism**
IF STAGE	305ps	330ps
ID STAGE	150ps	150ps
EX STAGE	275ps	225ps
MEM STAGE	250ps	450ps
WB STAGE	25ps	195ps
Clock Cycle TIme	**305ps**	**450ps**

Instruction Count

Baseline	5n + (n-1) + 2 = 6n + 1
Optimized	5(n/32) + ((n/32) - 1) + 2 = (6n/32) + 1

Total Cycles

Minimum Cycles without stalls: n + k – 1 n
= number of instructions
k = number of stages in the pipeline = 5

Baseline	6n + 1 + 5 - 1 = 6n + 5 cycles

Optimized	(6n/32) + 1 + 5 - 1 = (6n/32) + 5 cycles

Instruction Mix Baseline

load	add[i]	branch	jal
10 %	40 %	30 %	20 %

Instruction M

lv	add[i]	branch	beqv	jal
~9 %	~45 %	~18 %	~18 %	~9 %

Optimizations to the load and branch instruction significantly reduce the number of instructions required to perform the benchmark. We calculate that as n, the number of elements in the array approach infinity the speedup converges to a value of 21.7 which is equivalent to 21.7 times the speedup versus the baseline implementation using only avaliable risc-v instructions and iterating through the array sequentially.

$$\lim_{n->\infty} \frac{(6n+5)*305}{(\frac{6n}{32}+5)*450} \approx 21.6889$$

Performance Optimized / Performance Baseline = Execution Time Baseline / Execution Time Optimized

Execution Time = total clock cycles * cycle time