

## Problem M3.16: Vector Machines [?? Hours]

In this problem, we analyze the performance of vector machines. We start with a baseline vector processor with the following features.

- 32 elements per vector register
- 8 lanes
- One ALU per lane: 1 cycle latency
- One MULT per lane: 2 cycle latency, fully pipelined
- One LOAD/STORE unit per lane: 4 cycle latency, fully pipelined
- No dead time
- No support for chaining
- Scalar instructions execute on a separate 5-stage fully-bypassed pipeline

To simplify the analysis, we assume a **magic memory system** with no bank conflicts and no cache misses. Also, scalar operands of vector instructions are read in the Decode stage.

The program we will use for this problem is listed below. (In all questions, you should assume that arrays **A**, **B** and **C** do not overlap in memory.)

### C code

```
for (i=0; i<328; i++) {  
    A[i] = A[i] * B[i];  
    C[i] = C[i] + A[i];  
}
```

### Problem M3.16.A

---

Consider the implementation of the C-code on the vector machine that executes it in the least number of cycles. Assuming the following initial values, insert vector instructions to complete the implementation.

- R1 points to A[0]
- R2 points to B[0]
- R3 points to C[0]
- R4 contains the value 328

```
        ANDI R5, R4, 31      # 328 mod 32
        MTC1 VLR, R5         # set VLR to remainder
loop:
    LV     V1, R1             # load A
    LV     V2, R2             # load B
    SLL    R7, R5, 2          #
    ADD    R1, R1, R7         # increment A ptr
    ADD    R2, R2, R7         # increment B ptr
    ADD    R3, R3, R7         # increment C ptr
    SUB    R4, R4, R5         # update loop counter
    LI     R5, 32             # reset VLR to max
    MTC1   VLR, R5
    BGTZ   R4, loop
```

Complete the pipeline diagram below with the loop code from Question M3.16.A on the baseline vector processor for one loop iteration. Do not fill in scalar instructions. Assume the scalar registers are available immediately, whenever needed. You may not require the entire length of the table.

Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**). A vector instruction is also fetched (**F**) and decoded (**D**). Then, it stalls (—) until its required vector functional unit is available. With no chaining, a dependent vector instruction stalls until the previous instruction finishes writing back ALL of its elements. A vector instruction is pipelined across all the lanes in parallel. For each element, the operands are read (**R**) from the vector register file, the operation executes on the load/store unit (**M**) or the ALU (**X**) or the MUL (**Y**), and the result is written back (**W**) to the vector register file. Assume that there is no structural conflict on the writeback port. A stalled vector instruction does not block a scalar instruction from executing.

[illegible]

[illegible]

**Problem M3.16.D**

---

What is the performance (flops/cycle) of the program with chaining?

**Problem M3.16.E**

---

Would loop unrolling of the assembly code improve performance without chaining? Explain. (You may rearrange the instructions when performing loop unrolling.)

## Problem M3.17: Vector Machines [?? Hours]

In this problem, we analyze the performance of vector machines. We start with a baseline vector processor with the following features.

- 32 elements per vector register
- 8 lanes
- One ALU per lane: 1 cycle latency
- One load/store unit per lane: 4 cycle latency, fully pipelined
- No dead time
- No support for chaining
- Scalar instructions execute on a separate 5-stage pipeline

To simplify the analysis, we assume a magic memory system with no bank conflicts and no cache misses.

We consider the execution of the following loop.

<u>C code</u>	<u>assembly code</u>
<pre>for (i=0; i&lt;320; i++) {     C[i] = A[i] + B[i] - 1; }</pre>	<pre># initial conditions: # R1 points to A[0] # R2 points to B[0] # R3 points to C[0] # R4 = 1 # R5 = 320  loop:     LV    V1, R1      # load A     LV    V2, R2      # load B     ADDV  V3, V1, V2   # add A+B     SUBVS V4, V3, R4   # subtract 1     SV    R3, V4      # store C     ADDI  R1, R1, 128  # incr. A pointer     ADDI  R2, R2, 128  # incr. B pointer     ADDI  R3, R3, 128  # incr. C pointer     SUBI  R5, R5, 32   # decr. count     BNEZ  R5, loop     # loop until done</pre>

Complete the pipeline diagram of the baseline vector processor running the given code.

Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**).

A stalled vector instruction does not block a scalar instruction from executing.

LV<sub>1</sub> and LV<sub>2</sub> refer to the first and second LV instructions in the loop.

instr.	cycle																																							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
LV <sub>1</sub>	F	D	R	M1	M2	M3	M4	W																																
LV <sub>1</sub>				R	M1	M2	M3	M4	W																															
LV <sub>1</sub>					R	M1	M2	M3	M4	W																														
LV <sub>1</sub>						R	M1	M2	M3	M4	W																													
LV <sub>2</sub>	F	D	—	—	—	R	M1	M2	M3	M4	W																													
LV <sub>2</sub>							R	M1	M2	M3	M4	W																												
LV <sub>2</sub>								R	M1	M2	M3	M4	W																											
LV <sub>2</sub>									R	M1	M2	M3	M4	W																										
ADDV		F	D	—	—	—	—	—	—	—	—	—	—	—	—	R	X1	W																						
ADDV																	R	X1	W																					
ADDV																		R	X1	W																				
ADDV																			R	X1	W																			
SUBVS		F	D	—																																				
SUBVS																																								
SUBVS																																								
SUBVS																																								
SV		F	D	—																																				
SV																																								
SV																																								
SV																																								
ADDI				F	D	X	M	W																																
ADDI					F	D	X	M	W																															
ADDI						F	D	X	M	W																														
SUBI							F	D	X	M	W																													
BNEZ								F	D	X	M	W																												
LV <sub>1</sub>									F	D	—																													
LV <sub>1</sub>																																								
LV <sub>1</sub>																																								
LV <sub>1</sub>																																								

### Problem M3.17.B

---

In this question, we analyze the performance benefits of chaining and additional lanes. Vector chaining is done through the register file and an element can be read (**R**) on the same cycle in which it is written back (**W**), or it can be read on any later cycle (chaining is *flexible*). For this question, we always assume 32 elements per vector register, so there are 2 elements per lane with 16 lanes, and 1 element per lane with 32 lanes.

To analyze performance, we calculate the total number of cycles per vector loop iteration by summing the number of cycles between the issuing of successive vector instructions. For example, in Question M3.17.A,  $LV_1$  begins execution in cycle 3,  $LV_2$  in cycle 7 and ADDV in cycle 16. Therefore, there are 4 cycles between  $LV_1$  and  $LV_2$ , and 9 cycles between  $LV_2$  and ADDV.

Complete the following table. The first row corresponds to the baseline 8-lane vector processor with no chaining. The second row adds flexible chaining to the baseline processor, and the last two rows increase the number of lanes to 16 and 32.

(Hint: You should consider each pair of vector instructions independently, and you can ignore the scalar instructions.)

Vector processor configuration	Number of cycles between successive vector instructions					Total cycles per vector loop iter.
	$LV_1$ , $LV_2$	$LV_2$ , ADDV	ADDV, SUBVS	SUBVS, SV	SV, $LV_1$	
8 lanes, no chaining	4	9				
8 lanes, chaining						
16 lanes, chaining						
32 lanes, chaining						



### Problem M3.17.C

Even with the baseline 8-lane vector processor with no chaining (used in Question M3.17.A), we can improve performance using software loop-unrolling and instruction scheduling. As a first step, we unroll two iterations of the loop and rename the vector registers in the second iteration.

```
loop:
I1:   LV    V1, R1      # load A
I2:   LV    V2, R2      # load B
I3:   ADDV  V3, V1, V2   # add A+B
I4:   SUBVS V4, V3, R4   # subtract 1
I5:   SV    R3, V4      # store C
I6:   ADDI  R1, R1, 128  # incr. A pointer
I7:   ADDI  R2, R2, 128  # incr. B pointer
I8:   ADDI  R3, R3, 128  # incr. C pointer
I9:   SUBI  R5, R5, 32   # decr. count
I10:  LV    V5, R1      # load A
I11:  LV    V6, R2      # load B
I12:  ADDV  V7, V5, V6   # add A+B
I13:  SUBVS V8, V7, R4   # subtract 1
I14:  SV    R3, V8      # store C
I15:  ADDI  R1, R1, 128  # incr. A pointer
I16:  ADDI  R2, R2, 128  # incr. B pointer
I17:  ADDI  R3, R3, 128  # incr. C pointer
I18:  SUBI  R5, R5, 32   # decr. count
I19:  BNEZ  R5, loop     # loop until done
```

Reorder the instructions in the unrolled loop to improve performance on the baseline vector processor (your solution does not need to be optimal).

Provide a valid ordering by listing the instructions below (a few have already been filled in for you). You may assume that the A, B and C arrays do not overlap.

Instr. Number	Instruction
I1	LV    V1, R1
I2	LV    V2, R2
I15	ADDI  R1, R1, 128
I16	ADDI  R2, R2, 128
I17	ADDI  R3, R3, 128
I9	SUBI  R5, R5, 32
I18	SUBI  R5, R5, 32
I19	<b>BNEZ  R5, loop</b>

### Problem M3.18: Vectorizing memcpy and strcpy [?? Hours]

Ben Bitdiddle has bought a state-of-the-art vector machine, the Zirconium™, which has vector registers holding up to 32 elements, and has decided to vectorize his C library functions. As a starting point, he vectorizes the C function memcpy. The specification for memcpy is given as

```
/* copy n words from ct to s, and return s.      */
/* The actual C code copies one byte at a time.  */
/* Our version copies one word at a time.        */
void *memcpy(void *s, void *ct, size_t n)
```

Ben implements memcpy in the following fashion, assuming s, ct, and n are in registers R1, R2, and R3 respectively. Assume that there are no delay slots.

```
      ADD    R5,R1,R0      ; store destination address in R5
      ADD    R4,R2,R0      ; store source address in R4
      ANDI   R6,R3,#31     ; N % 32
      MTC1   VLR,R6       ; put length in vector length register
loop:
      LV     V1,R4
      SV     R5,V1
      SUB    R3,R3,R6      ; subtract elements
      SLLI   R6,R6,#2
      ADD    R4,R4,R6      ; bump source pointer
      ADD    R5,R5,R6      ; bump destination pointer
      ADDI   R6,R0,#32
      MTC1   VLR,R6       ; reset to full length
      BNEZ   R3,loop      ; any more to do?
```

---

#### Problem M3.18.A

The Zirconium processor has one load/store unit with a single lane that is fully pipelined with a latency of 10 cycles and a dead time of 10 cycles. Instructions do not need to spend an extra cycle writing back values. All scalar instructions are executed on a separate 5-stage pipelined fully-bypassed datapath. Therefore, the execution of scalar instructions and vector instructions maybe overlapped. How many cycles are required to copy each element when a very long memory vector is copied, i.e., in steady state?

### Problem M3.18.B

---

Ben's next target is `strcpy`, defined as follows:

```
/* copy string ct to string s, including '\0' and return s */
/* The actual C code copies one byte at time.                */
/* Our version copies one word at a time.                    */
void *strcpy(void *s, void *ct)
```

The difference between `strcpy` and `memcpy` is that `strcpy` terminates when it sees the string terminating character `'\0'` while `memcpy` copies a given length.

Ben makes several attempts to vectorize the code, but gives up deciding that it is not vectorizable. Alyssa, however, informs Ben that this function can be vectorized using some additional vector instructions listed below.

CLZM	R1, VM	Counts the number of leading 0s in the vector-mask register VM and puts the result in R1. For example, if the contents of VM are 0001010...000, <code>clzm R1, VM</code> puts 3 into R1.
S--V	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If the condition is true, put a 1 in the corresponding bit vector; otherwise put 0.
S--SV	F0, V1	Put the resulting bit vector in the vector-mask register (VM). The instruction S--SV performs the same compare but using a scalar value as one operand.

Given the additional instructions, help Ben write vectorized code for the Zirconium processor. Assume `s` and `ct` are in register R1 and R2, respectively. The Zirconium processor does not have virtual memory and does not trap on memory protection violations on vector memory loads. Also, assume that a string must be word-aligned. The terminating character must start at a word boundary and the remaining 3 bytes after the terminating character must be 0x0. (Hint: The ASCII value of `'\0'` is 0.)

### Problem M3.18.C

---

Compare the performance of vectorized `memcpy` and vectorized `strcpy` with and without vector chaining. Specifically, how many cycles are required to transfer one element in steady state? Assume that there is one vector compare unit with one lane and one cycle latency that compares whether two values are equal.

### Problem M3.19: Performance of Vector Machines [?? Hours]

The vector processor Germanium™ has a vector addition and a vector multiply unit with the following attributes.

- 1) Vector registers have 32 elements. The vector register file supports 2 read ports and 1 write port for each addition unit and multiplication unit.
- 2) The vector addition unit has a 2-cycle latency and is fully pipelined.
- 3) The vector multiplication unit has a 3-cycle latency and is fully pipelined.

You are now given the following code.

```
I1: ADDV  V3, V2, V1
I2: ADDV  V4, V2, V1
I3: MULTV V5, V4, V3
```

Note: All vectors are 32 elements in length.

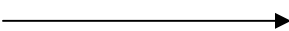
#### Problem M3.19.A

---

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, a 2-cycle dead time, and no vector chaining. Instruction fetch takes one cycle, so does instruction decode (unless the instruction is stalled). Reading data from the register file also takes one cycle. Use F for fetch, D for Decode, R for Vector register read and W for write back.

How many cycles does the given code take to execute? Count execution time as the number of cycles from when the first result is written to when the last result is written (inclusive).

Pipeline diagram for ADDV V3, V2, V1 and vector lengths of 24 elements, is shown below. Because we need to do 24 operations using 8 lanes, the vector register file should be read three times. X1 is the first stage of the addition unit and X2 is the second. In cycle 6, the results of the first 8 operations are written back. This instruction takes 3 cycles to execute.

**Time** 

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
F	D	R	X1	X2	W		
			R	X1	X2	W	
				R	X1	X2	W

**Problem M3.19.B**

---

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, no dead time, and vector chaining. Vectoring chaining is done through the register file. A vector unit can read an element from the register file in the same cycle it is being written back. How many cycles does the given code take to execute?

**Problem M3.19.C**

---

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 16 lanes, no dead time, and vector chaining. How many cycles does the given code take to execute?