

Problem M15.1: Exploiting Parallelism (Spring 2014 Quiz 3, Part B)

Consider the following C code sequence:

```
const int size = 64 * 1024;
int a[SIZE], b[SIZE], c[SIZE];
for (int i = 0; i < SIZE; i++) {
    if (a[i] > b[i]) {
        c[i] = a[i] + b[i];
    }
}
```

This is a repetitive computation with a simple dependency graph. If we look at the MIPS assembly code, we see that a large percentage of the instructions are doing bookkeeping. We'd like to reduce this overhead.

```
        // R1 points to a, R2 points to b, R3 points to c
        // R6 is i
        ADD R6, R0, SIZE
Loop:   LD R4, 0(R1)
        LD R5, 0(R2)
        SUB R8, R4, R5
        BGEZ R8, Skip
        ADD R4, R5, R4
        ST R4, 0(R3)
Skip:   ADD R1, R1, 4
        ADD R2, R2, 4
        ADD R3, R3, 4
        SUB R6, R6, 1
        BNEZ R6, Loop
```

Problem M15.1.A

Circle the MIPS instructions in the assembly above that perform “useful work” rather than bookkeeping.

Shown in red above.

Problem M15.1.B

If the loads in the preceding code take four cycles, then this code sequence will stall and performance will suffer. Explain how an in-order, fine-grain multithreaded processor with two threads could mitigate this effect?

Fine-grain multithreaded processors use round robin to schedule threads. So with two threads, each thread executes one instruction (or tries to, at least) every two cycles. This effectively halves the load latency, and therefore leads to fewer stalls.

How would the program need to change for multithreading? (You do *not* need to write the code.)

You need to split the iterations evenly between the threads. This can be done in many ways; one simple way is to have the first do even iterations and the second to do odd iterations.

Problem M15.1.C

An alternative approach is to hide the load latency within a single thread by using loop unrolling. Loads take four cycles and adds take one cycle. Write a loop unrolled VLIW version of the preceding code using the same VLIW instruction format as in Part A:

Memory operation	ALU operation	ALU operation / Branch
------------------	---------------	------------------------

Unroll the fewest number of loop iterations necessary to cover the load's latency. Whatever degree of unrolling you choose, assume it divides the array size. Also assume that predication is allowed:

(p1) instruction executes the instruction if predicate register p1 is set.
cmp.gt p1, r1, r2 sets predicate register p1 if r1 is greater than r2.

Finally, R1 points to a, R2 points to b, R3 points to c, and R6 is i.

LD R4, 0(R1)		
LD R5, 0(R2)		
LD R4, 4(R1)		
LD R5, 4(R2)		
LD R4, 8(R1)		
LD R5, 8(R2)	ADD R6, R4, R5	CMP.GT P1, R4, R5
	ADD R1, R1, 12	ADD R2, R2, 12
(P1) ST R6, 0(R3)	ADD R6, R4, R5	CMP.GT P1, R4, R5
	ADD R3, R3, 12	SUB R6, R6, 3
(P1) ST R6, 4(R3)	ADD R6, R4, R5	CMP.GT P1, R4, R5
(P1) ST R6, -4(R3)	BNEZ R6, Loop	

Problem M15.1.D

Write a vector version using vector instructions and the vector mask register. Assume that the vector machine can do up to 64 operations per instruction, and note that `SIZE` is a multiple of 64.

VLR register stores the vector length.

`LV v1, r1, Imm` loads vector register `v1` with memory starting at address `r1` and stride `Imm`.

`SV v1, r1, Imm` behaves similarly for stores.

`ADDV v1, v2, v3` adds `v2` and `v3` and puts the result in `v1`.

`SGTVV v1, v2` sets the vector mask register for each vector element in `v1` greater than the corresponding element in `v2` (mask set means the operation is enabled).

`CVM` resets the vector mask register (turns on all elements).

```
        // R1 points to a, R2 points to b, R3 points to c
        // R6 is i
        ADD R6, R0, SIZE
        LI VLR, 64
Loop:
```

```
        CVM
        LV V1, R1, 4
        LV V2, R2, 4
        SGTVV V1, V2
        ADDV V1, V1, V2
        SV V1, R3, 4
```

```
Skip:   ADD R1, R1, 64*4
        ADD R2, R2, 64*4
        ADD R3, R3, 64*4
        SUB R6, R6, 64
        BNEZ R6, Loop
```

Problem M15.1.E

Is this program easy to map to GPUs? What inefficiencies may arise? Explain your answer in one or two sentences.

This program is easy to write for GPUs because each iteration is completely independent. It may be inefficient, however, due to branch divergence, depending on the distribution of $A[i] > B[i]$ within the array.

Problem M15.2: VLIW, Vector Machines, and GPUs (Spring 2015 Quiz 4, Part C)

Consider the following C code fragment:

```
for(int i = 0; i < 301; i++)
{
    if(A[i] != B[i])
        C[i] = A[i] + 1;
    else
        C[i] = A[i] - 1;
}
```

A, B and C are arrays of 301 integers each. (Note: sizeof(int) = 4 bytes). Assume that A, B and C are stored in non-overlapping regions of memory.

The MIPS assembly for this code is shown below.

```
# R1 points to A[0]
# R2 points to B[0]
# R3 points to C[0]
# R4 contains a value of 301

loop:   LW      R5, 0(R1)
        LW      R6, 0(R2)
        BEQ     R5, R6, else
        ADDI    R5, R5, #1
        J       next
else:   ADDI    R5, R5, #-1
next:   SW      R5, 0(R3)
        ADDI    R1, R1, #4
        ADDI    R2, R2, #4
        ADDI    R3, R3, #4
        ADDI    R4, R4, #-1
        BNEZ    R4, loop
```

In the rest of the problem, assume that load instructions that hit in the cache take 4 cycles (i.e., if load instruction I1 starts execution at cycle N, then instructions that depend on the result of I1 can only start execution at or after cycle N+4) while all other instructions take 1 cycle. Assume the data cache has two read ports, two write ports, and is pipelined (i.e., it can accept a new request every cycle). Also assume perfect branch prediction and 100% hit rate in the instruction and data caches.

[illegible]

Problem M15.2.B

Now consider a vector machine. In addition to scalar registers, the machine has 32 vector registers, each 32-elements long. Vector instructions are described in the following table.

Instruction		Meaning
MTC1	VLR, Ri	Set VLR (vector length register) to the value of register Ri.
CVM		Set all elements in vector-mask (VM) register to 1.
LV	Vi, Rj	Load vector register Vi from memory starting at address Rj (under mask vector).
SV	Vi, Rj	Store Vi to memory starting at address Rj (under mask vector).
ADDVV	Vi, Vj, Vk	Add elements of Vj and Vk and then put each result in Vi (under mask vector).
ADDVS	Vi, Vj, Rk	Add Rk to each element of Vj and then put each result in Vi (under mask vector).
SUBVV	Vi, Vj, Vk	Subtract elements of Vk from Vj and then put each result in Vi (under mask vector).
SUBVS	Vi, Vj, Rk	Subtract Rk from elements of Vj and then put each result in Vi (under mask vector).
S--VV	Vi, Rj	Compare the elements (EQ, NE, GT, LT, GE, LE) in Vi and Vj. If the condition is true, put a 1 in the mask vector (VM), otherwise put 0.

Rewrite the code fragment for this vector machine by filling in the table on the next page. For your convenience, part of the assembly code is already written for you. You may not need all the rows.

```
# R1 points to A[0]
# R2 points to B[0]
# R3 points to C[0]
# R4 contains a value of 301
```

Label	Instruction	Comment (Optional)
-------	-------------	--------------------

	ADDI R7, R0, #1	Set R7 to 1
	ANDI R5, R4, #31	Set R5 to R4%32
	MTC1 VLR, R5	Set VLR to R5
	SLL R6, R5, #2	Set R6 to R5*4
loop:	CVM	Set all elements in mask to 1
	LV V1, R1	
	LV V2, R2	
	SNEVV V1, V2	
	ADDVS V3, V1, R7	
	SEQVV V1, V2	
	SUBVS V3, V1, R7	
	CVM	
	SV V3, R3	
	ADD R1, R1, R6	
	ADD R2, R2, R6	
	ADD R3, R3, R6	
	SUB R4, R4, R5	
	ADDI R5, R0, #32	Set R5 to 32
	MTC1 VLR, R5	Set VLR to R5
	SLL R6, R5, #2	Set R6 to R5*4
	BGTZ R4, loop	

Problem M15.2.C

Suppose this vector machine has four lanes. Each lane has one ALU for adds, one ALU for comparisons, and a load-store unit with one read port and one write port. Both ALUs take a single cycle, and memory takes 4 cycles. Assume we use vector chaining to reduce stalls due to data dependences. The machine can chain a load to an ALU instruction, or an add ALU instruction to a

compare ALU instruction. Also assume that the mask register is updated at the end of the cycle when an entire S—VV instruction is finished.

In this question, assume each vector register has at least N elements. If we run the same program but with N iterations (instead of 301) on this vector machine, what is the average number of cycles per element for this loop in steady state for a very large value of N ?

The answer to this question is based on the answer of Question2-1. We give you full grades if your calculation is correct based on the program you wrote.

Since the program has N iterations and each vector register has N elements, there is only one iteration.

(1) If we assume that the machine cannot chain a compare ALU instruction to an add ALU instruction:

LV	V1, R1	-> $N/4$
LV	V2, R2	-> $+ N/4$
SNEVV	V1, V2	-> $+ 4$ (chaining: start after first 4 elements in V2 finish loading)
ADDVS	V3, V1, R7	-> $+ N/4$ (no chaining: start after SNEVV is done)
SEQVV	V1, V2	-> $+ 1$ (start a cycle after ADDVS to avoid overwriting mask)
SUBVS	V3, V1, R7	-> $+ N/4$ (no chaining: start after SEQVV is done)
CVM		-> $+ 1$
SV	V3, R3	-> $+ N/4$

Since N is very large, the average number of cycles per element is $(N*5/4)/N = 5/4$

(2) If we assume that the machine can chain a compare ALU instruction to an add ALU instruction:

LV	V1, R1	-> $N/4$
LV	V2, R2	-> $+ N/4$
SNEVV	V1, V2	-> $+ 4$ (chaining: start after first 4 elements in V2 finish loading)
ADDVS	V3, V1, R7	-> $+ 1$ (chaining with SNEVV)
SEQVV	V1, V2	-> $+ (N/4 - 1)$ (start after SNEVV is done)
SUBVS	V3, V1, R7	-> $+ 1$ (chaining with SEQVV)
CVM		-> $+ 1$
SV	V3, R3	-> $+ N/4$

Since N is very large, the average number of cycles per element is $(N*4/4)/N = 1$

Problem M15.2.D

Suppose we code this program to run on a GPU with N warps. Each warp has 32 threads sharing the same PC and thus executing the same instruction. Assume each operation takes 16 cycles to execute. At most one instruction can be issued per cycle. In this GPU, each lane has one ALU and one load-store unit.

- (1) If the machine has 32 lanes, what is the minimum value of N to achieve the highest pipeline utilization?

With 32 lanes, issuing 32 threads in a warp takes 1 cycle ($1=32/32$). To achieve the highest pipeline utilization, we need at least 16 warps ($16 \text{ warps} = 16 \text{ cycle} / 1 \text{ cycle per warp}$).

- (2) If the machine has 16 lanes, what is the minimum value of N to achieve the highest pipeline utilization?

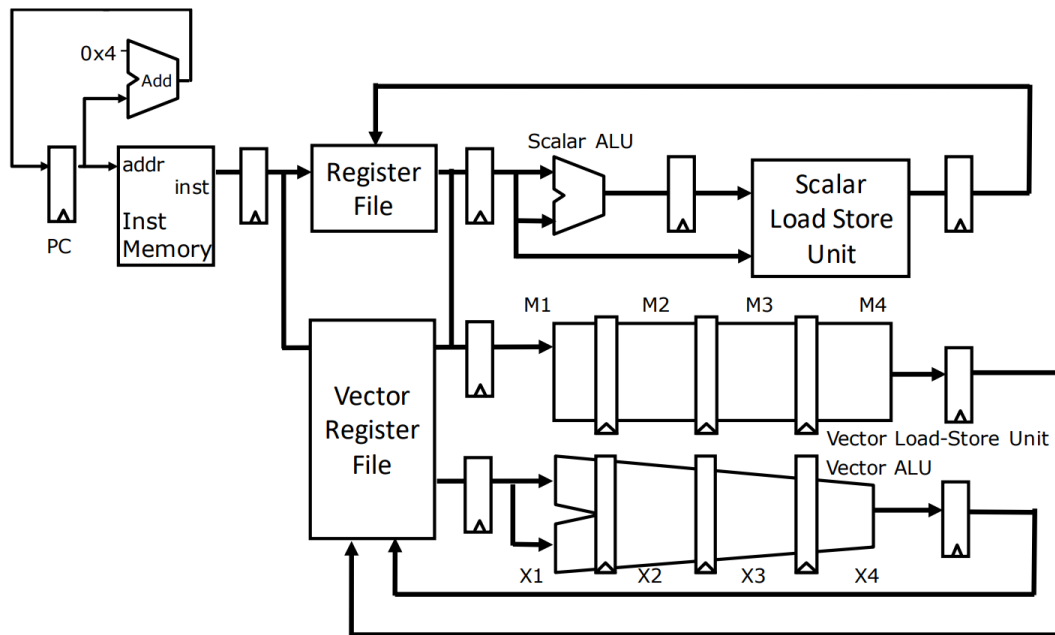
With 16 lanes, issuing 32 threads in a warp takes 2 cycles ($2=32/16$). To achieve the highest pipeline utilization, we need at least 8 warps ($8 \text{ warps} = 16 \text{ cycle} / 2 \text{ cycle per warp}$).

Problem M15.3: Vector Processors and GPUs (Spring 2020 Quiz 4, Part B)

Ben's vector processor has these features:

- Single-issue, in-order execution.
- Scalar instructions execute on a 5-stage, fully-bypassed pipeline.
- 32 vector registers named V0 through V31. Each vector register holds **32 floating-point elements**. The register files have enough ports to keep all lanes busy.
- Four vector lanes, each with one floating-point ALU and one load-store unit. Vector loads and arithmetic take **four** cycles to produce results and **one** cycle for writeback.
- No support for vector chaining.

This schematic shows a simplified view of the processor:



The processor can issue a single (scalar or vector) instruction per cycle. Once it issues, a vector instruction uses either all lanes' ALUs or all lanes' load-store units for as many cycles as needed to produce all of its results. Vector units are pipelined, so independent operations can be issued in sequence such that each stage in each vector unit operates on different values every cycle. A vector load or store can execute in parallel with independent operations that use the vector ALUs, and vector operations can execute in parallel with scalar operations. If a vector instruction depends on the result of a prior instruction, it stalls until the prior instruction finishes writing back **all** of its results. The processor implements MIPS plus the following vector instructions:

Instruction	Meaning
setv _l r rs	Set vector length register (VLR) to the value in rs
lv V _t , rs	Load vector register V _t starting at address in rs
sv V _t , rs	Store vector register V _t starting at address in rs
fadd.vv V _d , V _s , V _t	Add elements in V _s , V _t , and store result in V _d
fmul.vv V _d , V _s , V _t	Multiply elements in V _s , V _t , and store result in V _d
fadd.vs V _d , V _s , ft	Add floating-point scalar ft to each element in V _s , store result in V _d
fmul.vs V _d , V _s , ft	Multiply each element in V _s by scalar ft, and store result in V _d

Problem M15.3.A

(a)

```
loop:  lv v0, r2
       lv v1, r1
       fadd.vs v2, v0, f1
       fmul.vv v3, v1, v2
       sv v3, r1
       addi r1, r1, 128
       addi r2, r2, 128
       bne r1, r3 loop
```

(If you assume that the sv instruction allows an immediate offset/displacement, then the two addi instructions could also be placed earlier. In this case, they can be placed almost anywhere without affecting performance, since they will be issued while vector operations are being executed in parallel on the vector lanes.)

(b) 47 cycles

Each vector instruction takes 8 cycles (32 elements / 4 lanes) to get all its operands into the lanes. There's a 13-cycle latency between dependent vector instructions (8 cycles to get all elements into lanes + 4 cycles for FU latency + 1 cycle for writeback). The following table shows how the latencies add up to 47 cycles per iteration. The scalar instructions can be issued while the vector lanes are busy and are not shown.

Instruction	Iteration i	Iteration (i+1)
lv	0	47 (must wait for sv to finish using load/store units)
lv	8	
fadd.vs	13 (depends on first lv)	
fmul.vv	26 (depends on fadd.vs)	
sv	39 (depends on fmul.vv)	

Problem M15.3.B

(a) Put the `fadd.vs` between the two `lv` instructions (because the `fadd.vs` can use the vector ALUs in parallel with the use of the vector load/store units by the loads).

(b)

Dependent vector instruction issue 4 cycles after the previous one, and now the vector load/store units are maximally utilized, with a vector load or store issuing every 8 cycles.

So, each iteration of the vectorized loop takes 24 cycles and performs 64 floating-point operations, so the throughput is $64/24 = 8/3 = 2.67 \text{ FLOPs/cycle}$

This table shows how the latencies add up for one loop iteration:

Instruction	Iteration i	Iteration (i+1)
lv	0	24 (waits for sv to finish)
fadd.vs	4 (dependant on 1st lv)	
lv	8 (waits for 1st lv to finish)	
fmul.vv	12 (dependant on 2nd lv)	
sv	16 (dependant on fmul.vv)	

Problem M15.3.C

A GPU executes many threads concurrently, overlapping many long-latency memory accesses from independent threads (by switching warps).

Problem M15.3.D

a) The code can be vectorized with additional support for vector masking.

b) Yes, the code can be vectorized (without any additional hardware support).

(Depending on the implementation, you could make a copy of some parts of array A, and that is easily done without requiring new hardware features.)

c) No, the code cannot be vectorized. Each iteration depends on the value $A[i-1]$ computed in the previous iteration, forcing serialized execution.

d) The code can be vectorized with additional support for vector gather.