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.

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?

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

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`.

NOTE: The back of this page has additional space.

Additional space:

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:




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.

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.

Last updated:
11/28/2022

Problem M15.2.A

Consider a VLIW processor. Each instruction can contain up to two integer ALU operations (including branches) and two memory operations. In addition, in this machine, any operation can be predicated with any general-purpose register. For example:

`[R3] SW R1, 0(R2)` executes the store instruction only if `R3` is not zero; similarly, `[!R3] SW R1, 0(R2)` executes the store only if `R3` is zero.

Fill in the following table by unrolling enough loop iterations to eliminate the stall cycles in the main loop. Do not use software pipelining.

Label	Mem	Mem	ALU/Branch	ALU/Branch

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, Vj	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)

loop:	CVM	Set all elements in mask to 1
	LV V1, R1	
	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?

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?

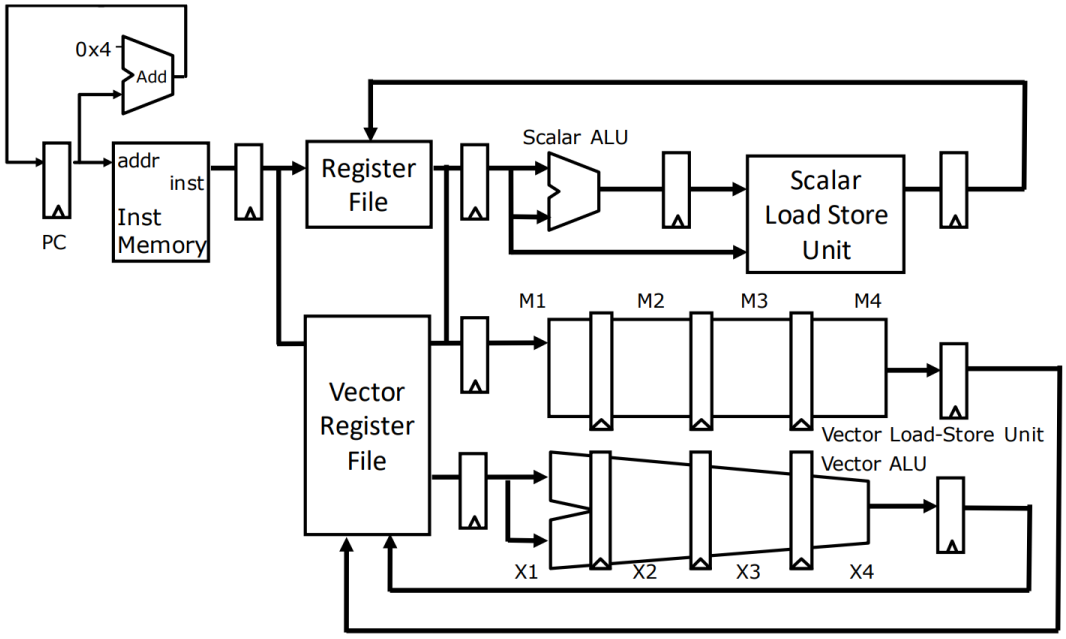
(2) If the machine has 16 lanes, what is the minimum value of N to achieve the highest pipeline utilization?

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**
`setvlr rs`	Set vector length register (VLR) to the value in rs
`lv Vt, rs`	Load vector register Vt starting at address in rs
`sv Vt, rs`	Store vector register Vt starting at address in rs
`fadd.vv Vd, Vs, Vt`	Add elements in Vs, Vt, and store result in Vd
`fmul.vv Vd, Vs, Vt`	Multiply elements in Vs, Vt, and store result in Vd
`fadd.vs Vd, Vs, ft`	Add floating-point scalar ft to each element in Vs, store result in Vd
`fmul.vs Vd, Vs, ft`	Multiply each element in Vs by scalar ft, and store result in Vd

Problem M15.3.A

Ben wants to analyze the performance of this vector processor the following loop:

```
for (i = 0; i < N; i++)
     A[i] = A[i] * (B[i] + 1.0);
```

For this part, assume that N is a multiple of 32. For your convenience, we've reproduced the original scalar assembly code for this loop:

```
     ;; Initial values:
     ;; f1 := 1.0
     ;; r1 := &A[0] and r2 = &B[0]
     ;; r3 := &A[N] (first address after vector A)
I1:  loop: ld f0, 0(r2)         ;; Load B[i]
I2:        ld f2, 0(r1)         ;; Load A[i]
I3:        fadd f3, f0, f1
I4:        addi r1, r1, 4
I5:        fmul f4, f2, f3
I6:        addi r2, r2, 4
I7:        st f4, -4(r1)        ;; Store A[i]
I8:        bne r1, r3, loop
```

(a) Provide equivalent vector code. For full credit, your code should execute as quickly as possible while using no more than two vector ALU instructions.

```
          addi r20, r0, 32 ;; set r20 to 32
          setvlr r20 ;; use all 32 vector elements
     loop:
```

(b) How many cycles are required in steady state for each vectorized loop iteration (which corresponds to 32 iterations of the original loop)?

Problem M15.3.B

Suppose we add chaining support to the processor. With chaining, a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is either already written to the vector register file or is available in the writeback stage (we add the required bypass paths).

(a) How would you reorder the vector instructions in your vectorized loop from the previous question to improve performance with chaining?

(b) What is the resulting throughput in floating-point arithmetic operations per cycle in steady state?

Problem M15.3.C

So far, we have assumed that loads can execute within a few cycles, which is reasonable if the data can be served by an L1 cache. Now consider if we are accessing arrays whose size far exceeds the capacity of any of our caches. Explain how a GPU would enable us to obtain high performance in this case.

Problem M15.3.D

State whether each of the loops below can be vectorized on our vector processor. If the code would require the vector processor to have additional features to be vectorizable, specify those features. If the code cannot be vectorized regardless of what additional features the processor were to implement, state your reasoning. The loops operate on floating-point arrays `A[N]`, `B[N]`, and `C[N]`. These arrays do not overlap.

a)
```
for (int i = 0; i < N; i++)
    if (A[i] < 0)
        A[i] = C[i] + B[i];
```

b)
```
for (int i = 0; i < N; i++)
    A[i] = A[i+1] * B[i] – 1.0;
```

c)
```
for (int i = 0; i < N; i++)
    A[i] = 1.0 + A[i-1] * B[i];
```

d)
```
for (int i = 0; i < N; i++)
    A[i] = A[i] + B[C[i]];
```