复习题（3）：VLIW、Vector and Multi-thread

Problem 1: VLIW

In this problem, you will port code to a simple VLIW machine, and modify it to improve performance. Details about the VLIW machine:

- Three fully pipelined functional units (Integer ALU, Memory, and Floating Point)
- Integer ALU has a 1 cycle latency
- Memory Unit has a 2 cycle latency
- FPU has a 3 cycle latency and can complete one add or one multiply (but not both) per clock cycle
- No interlocks

C Code:

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

Assembly Code:

```
loop:  ld   f1, 0(r1)
       ld   f2, 0(r2)
       fmul f1, f1, f1
       fadd f1, f1, f2
       st   f1, 0(r3)
       addi r1, r1, 4
       addi r2, r2, 4
       addi r3, r3, 4
       bne  r3, r4, loop
```

Problem 1.A

Schedule operations into the VLIW instructions in the following table. Show only one iteration of the loop. Make the code efficient, but do not use software pipelining or loop unrolling. You do not need to write in NOPs (can leave blank).

ALU	Memory Unit	FPU
addi r1, r1, 4	ld f1, 0(r1)	
addi r2, r2, 4	ld f2, 0(r2)	
		fmul f1, f1, f1
		fadd f1, f1, f2
addi r3, r3, 4		
bne r3, r4, loop	st f1, -4(r3)	

What performance did you achieve? (in FLOPS per cycle): **2/9**_____

Problem 1.B

Unroll the loop by one iteration (so two iterations of the original loop are performed for every branch in the new assembly code). You only need to worry about the steady-state code in the core of the loop (no epilogue or prologue). Make the code efficient, but do not use software pipelining. You do not need to write in NOPs (can leave blank).

ALU	Memory Unit	FPU
	ld f1, 0(r1)	
addi r1, r1, 8	ld f3, 4(r1)	
	ld f2, 0(r2)	fmul f1, f1, f1
addi r2, r2, 8	ld f4, 4(r2)	fmul f3, f3, f3
		fadd f1, f1, f2
		fadd f3, f3, f4
addi r3, r3, 8	st f1, 0(r3)	
bne r3, r4, loop	st f3, -4(r3)	

What performance is achieved now? (in FLOPS per cycle): **4/10**_____

Problem 1.C

With unlimited registers, if the loop was fully optimized (loop unrolling and software pipelining), how many FLOPS per cycle could it achieve? What is the bottleneck?
(Hint: You should not have to write out the assembly code.)

It could achieve 2/3 flops per cycle. It will be bottlenecked by memory accesses, since each iteration has 3 memory ops (2 loads and 1 store) and only 2 floating point ops, and there is only one functional unit for each.

Many people picked the FPU because it has the longest latency. In steady state, it is a matter of throughput rather than latency.

Problem 2: Vector

In this problem we will examine how vector architecture implementations could affect the performance of various codes. As a baseline implementation, assume:

- 64 elements per vector register
- 8 lanes
- One ALU per lane; 2 cycle latency
- One load/store unit per lane; 8 cycle latency
- No dead time
- No support for chaining
- Scalar instructions execute on a separate five-stage pipeline

Between two given alternatives, pick the modification that will yield the greatest performance improvement and explain why (assuming everything else is held constant). Be sure to explain why the other choice will not help as much.

Problem 2.A

```
# Vector Assembly

   LV   V0, R1
   ADDV V1, V1, V2
   MULV V2, V2, V2
   ADDV V3, V3, V4
   ADDV V5, V5, V6
   ADDV V1, V1, V0
```

Circle one:

- Double number of lanes
- Add support for chaining

There will be no gain from chaining because there aren't any stalls caused by dependencies. The instructions aren't all entirely independent since the last two instructions use previously computed vectors. If you work out the latencies, you will see that V0 and V1 will be ready before they are needed so there will be no stalls.

Doubling the number of lanes will improve performance because 16 lanes is still less than the vector length.

Problem 2.B

```
# C Code
#  Vector reduction from lecture
#  VL is vector length (power of 2)

   do {
      VL = VL/2;
      sum[0:VL-1] += sum[VL:2*VL-1];
   } while (VL>1);
```

Circle one:

- Double number of lanes
- Double vector unit clock frequency

In this vector reduction code, the vector length keeps getting halved. This means for a significant portion of its execution it will be using short vectors. Additional lanes can't help with short vectors.

Doubling the clock rate will still offer the theoretical doubling of throughput, but it will be able to achieve that even with short vectors.

Problem 2.C

```
# Vector Assembly

   LV   V0, R1
   MULV V0, V0, V0
   LV   V1, R2
   ADDV V1, V1, V0
   SV   V1, R2
```

Circle one:

- Double number of lanes
- Add support for chaining

Many of these instructions are dependent, so even with more lanes, the system will need to stall for dependencies. Chaining will allow for the biggest performance improvement.

Problem 3: Multithreading

Consider the following code on a multithreaded architecture. You can assume each thread is running the same program. You can assume:

- Single-issue and in-order machine pipeline
- ALU is fully pipelined with a latency of 2
- Branches (conditional and unconditional) take 2 cycles to complete (if branch is started on cycle 1, the following instruction can't start until cycle 3)
- Memory Unit is fully pipelined with a latency of 16

Code:

```
loop:   beq r1, r0, end      1     36
        lw  r2, 4(r1)        3
        add r3, r3, r2       19
        lw  r1, 0(r1)        20
        j   loop             21
end:
```

Problem 3.A

How many cycles does it take for one iteration of the loop to complete if the system is single threaded?

36 – 1 = 35 cycles. The jump executes while the last lw is in progress, but the next iteration can't start until the load is done.

Problem 3.B

If the system is multithreaded with fixed round-robin scheduling, how many threads are needed to fully utilize the pipeline?

It needs to cover 15 cycles of latency (between lw and add), so 15 + 1 = 16 threads will be needed.

Problem 3.C

What is the minimum number of threads needed for peak performance if the system is changed to data-dependent thread scheduling (i.e., can pick next instruction from any thread that is ready to execute) and you are allowed to re-schedule instructions in the assembly code? Explain.

If the second load is moved up to below the first, latency of iteration becomes 21 cycles (5 issues + 16 stall cycles). The ceiling of (21/5) = 5 so at least five threads are needed. Four threads is almost just enough. If you assume no thread switches on branches, then 3 threads should be sufficient.

Problem 4: Memory Latency

In this problem, we will compare how the three architectures studied (VLIW, vector, and multithreaded) tolerate memory latency, both in their hardware design and in how they are programmed. To make the comparison fair, each system has only one floating-point unit (FPU). For the rest of the problem, please consider the following program (vector add):

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

Problem 4.A

Assume the latency to memory is always 100 cycles (no cache). How do each of the architectures tolerate the memory latency? Be sure to describe what programming techniques and what hardware mechanisms help for each machine type.

i) VLIW

The impact of memory latency is reduced by getting more iterations of the loop in flight at the same time. With a VLIW system, this must be explicitly managed by the programmer (or compiler). Software pipelining allows iterations to be overlapped, while loop unrolling allows iterations to be combined. A rotating register file can be used to simplify and shorten the code.

ii) Vector

A vector machine hides the latency of instructions by overlapping their execution. With sufficiently long vectors, the memory latency can be amortized over many elements. With chaining, the multiple vector instructions can be overlapped, further hiding the latency. Multiple lanes doesn't help to hide the latency, and for this problem it was stated there was only one FPU. To use a vector machine, the programmer must stripmine the code.

iii) Multithreaded

This system hides the latency by running other threads while waiting. The programmer must partition the program into multiple threads.

Problem 4.B

What if the memory latency increased from 100 (Problem Q5.4.A) to 1000 cycles? For each architecture, what changes are needed to hardware and software to continue to hide the latency?

i) VLIW

To hide a longer latency, more iterations will need to be in flight at the same time which means the code will need to have its loops unrolled more times and be software pipelined to a greater degree. This will require explicit and significant changes to the code by the programmer. It will also require many more physical registers to support this.

ii) Vector

To best hide the latency, longer vectors will be needed. To utilize longer vectors will require negligible changes to the software.

iii) Multithreaded

To hide more latency will require more threads, and these will need to be created by the programmer. Each additional thread will require more registers to holds its architectural state.

Problem 5: Multithreading

This problem evaluates the effectiveness of multithreading using a simple database benchmark. The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

```
struct node {
  int key;
  struct node *next;
  struct data *ptr;
}
```

The following MIPS code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key. Assume MIPS has no delay slots.

```
      ;
      ; R1: a pointer to the linked list
      ; R2: the key to find

loop: LW R3, 0(R1)       ; load a key
      LW R4, 4(R1)       ; load the next pointer
      SEQ R3, R3, R2     ; set R3 if R3 == R2
      BNEZ R3, End       ; found the entry
      ADD R1, R0, R4
      BNEZ R1, Loop      ; check the next node
End:
      ; R1 contains a pointer to the matching entry or zero if not found
```

We run this benchmark on a single-issue in-order processor. The processor can fetch and issue (dispatch) one instruction per cycle. If an instruction cannot be issued due to a data dependency, the processor stalls. Integer instructions take one cycle to execute and the result can be used in the next cycle. For example, if SEQ is executed in cycle 1, BNEZ can be executed in cycle 2. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

5.A Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and nonblocking. After the processor issues a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation. How many cycles does it take to execute one iteration of the loop in steady state?

Since there is no penalty for conditional branches, instructions take one cycle to execute unless there is a dependency problem. The following table summarizes the execution time for each instruction. From the table, the loop takes 104 cycles to execute.

Instruction	Start Cycle	End Cycle
LW R3, 0(R1)	1	100
LW R4, 4(R1)	2	101
SEQ R3, R3, R2	101	101
BNEZ R3, End	102	102
ADD R1, R0, R4	103	103
BNEZ R1, Loop	104	104

5.B Now we add zero-overhead multithreading to our pipeline. A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle. In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling. Each of the N threads executes one instruction every N cycles. What is the minimum number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?

If we have N threads and the first load executes at cycle 1, SEQ, which depends on the load, executes at cycle 2N + 1. To fully utilize the processor, we need to hide 100-cycle memory latency, $2N + 1 \geq 101$. The minimum number of thread needed is 50.

5.C How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time processor takes to find an entry with a specific key)? Assume the processor switches to a different thread every cycle and is fully utilized.

Better throughput, worse latency

5.D We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency. What is the minimum number of threads to fully utilize the processor now? Note that the processor issues instructions in-order in each thread.

In steady state, each thread can execute 6 instructions (SEQ, BNEZ, ADD, BNEZ, LW, LW). Therefore, to hide 98 (104-6) cycles between the second LW and SEQ, a processor needs 98/ 6+1 = 18 threads.

Problem 6: **Simultaneous Multithreading**

Consider a **Simultaneous Multithreading (SMT)** machine with limited hardware resources. **Circle** the following hardware constraints that can limit the total number of threads that the machine can support. For the item(s) that you circle, **briefly describe** the minimum requirement to support **N** threads.

(A) Number of Functional Unit:

Since not all the treads are executed each cycle, the number of functional unit is not a constraint that limits the total number of threads that the machine can support.

(B) Number of Physical Registers:

We need at least [N × (number of architecture registers)] physical registers for an in-order system. Since it is SMT, it is actually least [N × (number of architecture registers) + 1] physical registers, because you can't free a physical register until the next instruction commits to that same architectural register.

(C) Data Cache Size:

This is for performance reasons.

(D) Data Cache Associatively:

This is for performance reasons.

Acknowledgement
This problem set is from CS 152, UC Berkeley:
http://www-inst.eecs.berkeley.edu/~cs152/sp12/