Problem 1: Out-of-order Execution (20 Points)

We execute the following function on an out-of-order core with a unified physical register file.

```
void scale_6(int* a0, int a1) {
  for (int i = 0; i < 6; i++) {
    a0[i] = a0[i] * a1;
  }
}
```

```
      addi  t1, a0, 0x18
loop: lw    t0, 0(a0)
      mul   t0, t0, a1
      sw    t0, 0(a0)
      addi  a0, a0, 0x4
      bnez  a0, t1, loop
      ret
```

1.A (10 points) OOO Structures

Fill out the contents of the reorder-buffer, freelist, and map table as the decode unit would until the ROB is full. Additionally, indicate the final values for the ROB head and tail pointers. The first instruction has been filled out for you. Assume no instruction will complete execution.

Note: The instruction column of the ROB is optional and will not be graded. Space is provided to assist you in bookkeeping.

FIFO Free List															
p11	p12	p13	p14	p15	p16	p17	p18	p19							

Map Table								
a0	p0							
a1	p1							
t0	p2							
t1	p3							

ROB			
IDX	**Instruction**	**PRd**	**LPRd**
0			
1			
2			
3	`addi t1, a0, 0x18`		
4			
5			
6			
7			

Final ROB Head:		**Final ROB Tail:**	

1.B (4 points) Precise exceptions
Note: For this problem, you should not update the structures in 1A.
After the ROB becomes full, there is a protection fault on the first load in the instruction sequence. The ROB recovers architectural state using the iterative exception recovery procedure discussed in class.

How many cycles does it take to recover architectural state?

Which registers are returned to the free list during this process?

1.C (3 points) Checkpointing
To provide fast pipeline restarts after branch mispredictions, some structures and values must be checkpointed when branches are dispatched. Circle which of the following structures must be checkpointed for fast pipeline restarts. No explanation is necessary. Assume all buffers are implemented as circular buffers, with head and tail pointers.

- ROB head pointer
- ROB tail pointer
- Physical register contents
- Physical register present bits
- Issue queue contents
- Physical register free list
- Architectural register map table
- Speculative store buffer head pointer
- Speculative store buffer tail pointer

1.D (3 points) Branch prediction
You build a PC-indexed BHT to improve prediction accuracy of the branch in this loop. What is the prediction accuracy for the conditional branch after this function is called many times when using 1-bit counters?

Prediction accuracy with 1-bit counters = _____

What is the prediction accuracy for the conditional branch after this function is called many times when using 2-bit saturating counters?

Prediction accuracy with 2-bit counters = _____

Problem 2: Multithreading (20 points)

2.A (12 points) True/False

For each statement below, indicate whether that statement is true or false and briefly explain your reasoning.

Adding multithreading to a single-issue out-of-order core will in general yield a greater relative performance improvement than adding multithreading to a similar single-issue in-order core with an identical memory hierarchy.

A multithreaded processor does not benefit from branch prediction with a sufficiently large number of threads.

The ISA must contain special multithreading instructions to support multithreading.

2.A (8 points) Performance

Consider execution of the following code on a single-issue in-order processor. Loads have 50-cycle latency and are fully pipelined. Adds have 4-cycle latency and are fully pipelined. Assume perfect branch prediction and that branches execute in one cycle.

```
// pointer chasing
loop: addi a1, a1, -0x1
      lw a0, 0(a0)
      bnez a1, loop
```

We add multithreading to this processor.

Fixed Switching: How many threads are needed to avoid stalls if threads are switched every cycle in a fixed round-robin schedule? Show your work.

Data-dependent Switching: How many threads are needed to avoid stalls if threads are switched only when an instruction cannot issue due to a data dependency?

Problem 3: VLIW (16 points)

In this problem, we will optimize the following code sequence, which performs a scatter operation with scaling, for a VLIW architecture.

```
loop:
     flw  ft0, 0(a0)         # x = src[i]
     lw   t0, 0(a1)          # p = idx[i]
     addi a0, a0, 0x4        # bump idx
     addi a1, a1, 0x4        # bump src
     addi a2, a2, -0x1       # decrement n
     fmul ft1, ft0, fa0      # scale x
     fsw  ft1, (t0)          # *p = x
     bnez a2, loop
```

- The source, index, and destination arrays do not overlap in memory. (Assume there are no memory-memory dependencies.)
- The number of iterations (initial value of `a2`) is greater than 0.
- Assume that no exceptions arise during execution.

The code is executed on an in-order VLIW machine with four execution units. All execution units are fully pipelined and latch their operands at issue.

- Two integer ALUs, 1-cycle latency, also used for branches
- One load/store unit, 3-cycle latency for loads
- One floating-point unit, 2-cycle latency

Instructions are statically scheduled with no interlocks; all latencies are exposed in the ISA. All register operands are read before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

Execution units write to the register file at the end of their last pipeline stage, and the results become visible at the beginning of the following cycle. There is no bypassing. **Old values can be read from registers until they have been overwritten.** (You may leverage this to more efficiently schedule VLIW code.)

The unoptimized scheduling of the above assembly code is shown in the following table.

Label	ALU0	ALU1	MEM	FPU
`loop:`	`addi a0, a0, 0x4`		`flw ft0, 0(a0)`	
	`addi a1, a1, 0x4`		`lw t0, 0(a0)`	
	`addi a2, a2, -0x1`			
				`fmul ft1,ft0,fa0`
			`fsw ft1, 0(t0)`	

3.A (12 points) Software Pipelining

Schedule operations with VLIW instructions using only software pipelining (no loop unrolling). Include the prologue and epilogue to initiate and drain the software pipeline. Try to optimize for efficiency and minimize the number of cycles, but prioritize correctness. Your code should behave correctly for any `a2` > 0.

You do not have to write out the full instructions unless they differ from the original – just the opcode and the destination register are sufficient. Entries for NOPs can be left blank.

Label	ALU0	ALU1	MEM	FPU

3.B (4 points) Loop Unrolling

Could this code achieve higher throughput by combining loop unrolling with software pipelining? If so, briefly explain the approach of applying loop unrolling to the software-pipelined code. If not, explain your reasoning.

Problem 4: Vectors (18 points)

4.A (12 points) Vectorization

Vectorize the following C function. Write your code in the table. You may not need to use all the spaces. Refer to Appendix A for an abbreviated RISC-V vector instruction listing.

- Fill in the blanks for the `vsetvli` instruction.
- The vector unit is configured with SEW=32 for 32-bit elements.
- Note that LMUL=1, so you may use any of the 32 standard vector registers. You may freely use any temporary registers.
- Assume there is no aliasing between the `a`, `b`, and `idx` arrays.

```
// Argument registers:             a0          a1         a2           a3
void leaky_activation(unsigned int n, int32_t *a, int32_t b, uint32_t *idx,
//                            a4              a5
                      int32_t scale, uint32_t dilation) {
    for (unsigned int i = 0; i < n; i++) {
        if (a[idx[i]] < 0) {
            b[dilation*i] = scale * a[idx[i]];
        }
    }
}
```

Label	**Instruction**	**Comment** (optional)
	`// Prologue (if any)`	
`loop:`	`vsetvli    ,    , e32, m1, ta, mu`	
	`// Do vector operations`	

	`// Update pointers`	
	`// Branch to loop`	
`end:`	`ret`	

4.B (6 points) Iron Law

For each of the following modifications to a baseline vector machine, indicate whether it has an **increasing**, **decreasing**, **negligible**, or **ambiguous** effect on CPI and on overall runtime (time per program) for a trivially vectorizable program, such as vector-vector add.

Briefly explain your reasoning in one or two sentences.

	Cycles / Instruction	**Time / Program**
Increase maximum hardware vector length (MAXVL)		
Increase number of lanes		