CS 152 Computer Architecture and Engineering
CS 252 Graduate Computer Architecture

Midterm #2

SOLUTIONS

April 15, 2020

Professor Krste Asanović

Name:_______________________

SID:_______________________

I am taking CS152 / CS252

(circle one)

80 Minutes, 17 pages.

Notes:

- Not all questions are of equal difficulty, so look over the entire exam!
- Please carefully state any assumptions you make.
- Please write your name on every page in the exam.
- Do not discuss the exam with other students who haven't taken the exam.
- If you have inadvertently been exposed to an exam prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Question	CS152 Point Value	CS252 Point Value
1	20	20
2	20	20
3	20	20
4	20	20
TOTAL	80	80

Problem 1: Vectors (20 points)

Problem 1.A (12 points)

State whether each of the following loops can be successfully vectorized and explain your reasoning. If it can only be vectorized under certain circumstances, give requirements on the inputs and any other necessary conditions.

In all cases, assume that A, B, and C are non-overlapping arrays in memory.

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

Yes, the dot product is vectorizable. In the absence of a special vector reduction instruction, the loop-carried dependence can be vectorized by reordering the summation into a binary tree, such that the vector of partial sums can be repeatedly split into two halves which are then added together with vector addition.

```
for (i = 0; i < N; i++) {
      if (C[i] == 0) {
            if (A[i] > B[i])
                  C[i] = A[i];
            else
                  C[i] = B[i];
      }
}
```

Yes, this conditional loop is vectorizable using predication. Two masks are required: `vm0[i] = (C[i] == 0)` and `vm1[i] = (A[i] > B[i])`. The innermost `if` clause would be predicated under the mask `(vm0 && vm1)` and the `else` clause under `(vm0 && !vm1)`.

Alternatively, the nested `if/else` statement could be handled with a `vmax.vv` instruction predicated under `vm0`, if available.

For simplicity, assume N is evenly divisible by 2.

```
for (i = 0; i < N; i++) {
      A[i] = B[i % (N/2)];
}
```

Yes, this is vectorizable by decomposing the loop to maintain unit-stride accesses:

```
for (i = 0; i < N/2; i++) {
      A[i] = B[i];
      A[i + (N/2)] = B[i];
}
```

A less efficient alternative is to use a gather operation for B.

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

This generally cannot be vectorized except in special cases. Although the indirect accesses appear vectorizable with vector indexed memory operations, there is an ordering constraint: Either the architecture must support an ordered scatter variant or `A[i] != A[j]` for `i`, `j` within a vector. Furthermore, to prevent a forward-carried dependence, it must also be known that `A[i] != B[j]` for `i < j` within a vector.

Problem 1.B (8 points)

Write vector assembly code for the following loop. Use either the RISC-V vector ISA described in lecture and Lab 4 or comparable vector pseudocode. Refer to Appendix A for an abbreviated RISC-V vector instruction listing.

- Arrays A and B contain 64-bit integers and do not overlap.
- N is passed in register `a0`.
- The base addresses of arrays A and B are passed in registers `a1` and `a2`, respectively.

```
for (i = 0; i < N; i++) {
     A[3*i+1] = A[3*i+2] + (A[3*i] * B[i]);
}
```

Label	**Instruction**	**Comment** (optional)
	addi t1, x0, 24	set stride in bytes
loop	vsetvli t0, a0, e64, m8	configure VL, SEW=64, LMUL=8
	addi a3, a1, 16	compute &A[3*i+2]
	addi a4, a1, 8	compute &A[3*i+1]
	vlse.v v0, (a1), t1	load A[3*i]
	vlse.v v8, (a3), t1	load A[3*i+2]
	vle.v v16, (a2)	load B[i]
	vmadd.vv v0, v16, v8	v0 = (v0 * v16) + v8
	vlse.v v0, (a4), t1	store A[3*i+1]
	sub a0, a0, t0	decrement VL
	slli t2, t0, 4	scale 2*VL to bytes
	slli t0, t0, 3	scale VL to bytes
	add t2, t2, t0	compute 3*VL in bytes
	add a2, a2, t0	bump pointer B
	add a1, a1, t2	bump pointer A
	bnez a0, loop	

Specifying the stride in terms of elements instead of bytes was also acceptable. However, pointers and offsets *must* be treated in units of bytes since memory is byte-addressed.

Problem 1: Vectors (20 points)

Problem 1.A (12 points)

State whether each of the following loops can be successfully vectorized and explain your reasoning. If it can only be vectorized under certain circumstances, give requirements on the inputs and any other necessary conditions.

In all cases, assume that A, B, and C are non-overlapping arrays in memory.

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

Yes, the dot product is vectorizable. In the absence of a special vector reduction instruction, the loop-carried dependence can be vectorized by reordering the summation into a binary tree, such that the vector of partial sums can be repeatedly split into two halves which are then added together with vector addition.

For simplicity, assume N is evenly divisible by 3.

```
for (i = 0; i < N; i++) {
      A[i] = B[i % (N/3)];
}
```

Yes, this is vectorizable by decomposing the loop to maintain unit-stride accesses:

```
for (i = 0; i < N/3; i++) {
      A[i] = B[i];
      A[i + (N/3)] = B[i];
      A[i + (N/3)*2] = B[i];
}
```

A less efficient alternative is to use a gather operation for B.

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

Yes, this conditional loop is vectorizable using predication. Two masks are required: `vm0[i] = (B[i] == 0)` and `vm1[i] = (A[i] != B[i])`. The innermost `if` clause would be predicated under the mask `(vm0 && vm1)` and the `else` clause under `!vm0`.

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

This generally cannot be vectorized except in special cases. Although the indirect accesses appear vectorizable with vector indexed memory operations, there is an ordering constraint: Either the architecture must support an ordered scatter variant or `B[i] != B[j]` for `i`, `j` within a vector. Furthermore, to prevent a forward-carried dependence, it must also be known that `B[i] != A[j]` for `i < j` within a vector.

Problem 1.B (8 points)

Write vector assembly code for the following loop. Use either the RISC-V vector ISA described in lecture and Lab 4 or comparable vector pseudocode. Refer to Appendix A for an abbreviated RISC-V vector instruction listing.

- Arrays A and B contain 64-bit integers and do not overlap.
- N is passed in register `a0`.
- The base addresses of arrays A and B are passed in registers `a1` and `a2`, respectively.

```
for (i = 0; i < N; i++) {
     A[3*i+2] = A[3*i] + (A[3*i+1] * B[i]);
}
```

Label	**Instruction**	**Comment** (optional)
	addi t1, x0, 24	set stride in bytes
loop	vsetvli t0, a0, e64, m8	configure VL, SEW=64, LMUL=8
	addi a3, a1, 8	compute &A[3*i+1]
	addi a4, a1, 16	compute &A[3*i+2]
	vlse.v v0, (a1), t1	load A[3*i]
	vlse.v v8, (a3), t1	load A[3*i+1]
	vle.v v16, (a2)	load B[i]
	vmadd.vv v8, v16, v0	v8 = (v8 * v16) + v0
	vlse.v v8, (a4), t1	store A[3*i+2]
	sub a0, a0, t0	decrement VL
	slli t2, t0, 4	scale 2*VL to bytes
	slli t0, t0, 3	scale VL to bytes
	add t2, t2, t0	compute 3*VL in bytes
	add a2, a2, t0	bump pointer B
	add a1, a1, t2	bump pointer A
	bnez a0, loop	

Specifying the stride in terms of elements instead of bytes was also acceptable. However, pointers and offsets *must* be treated in units of bytes since memory is byte-addressed.

Problem 2: VLIW (20 points)

In this problem, we will optimize the following code sequence, which implements a prefix sum interleaved with a vector multiply, for a VLIW architecture.

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

```
     # f1 contains initial value of x
     addi x1, x0, N*8         # initialize loop boundary
     addi x2, x0, 0           # initialize array index
loop:
     fld f2, A(x2)            # load A[i]
     fld f3, B(x2)            # load B[i]
     addi x2, x2, 8           # bump index
     fadd.d f1, f1, f2        # compute x
     fmul.d f4, f1, f3        # compute C[i]
     fsd f4, (C-8)(x2)        # store C[i]
     bltu x2, x1, loop
```

- "`A`", "`B`", and "`C`" are immediates generated by the compiler that encode the base addresses of arrays A, B, and C, respectively.
- Arrays A, B, and C do not overlap in memory.
- N is a large number that is statically known.
- N is evenly divisible as needed for loop unrolling and software pipelining.
- Register `f1` holds the initial value of x.
- Assume that no exceptions arise during execution.

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

- One integer ALU, 1-cycle latency, also used for branches
- One load unit, 2-cycle latency
- One store unit (ignore the latency of memory-memory dependencies for this problem)
- One floating-point adder, 3-cycle latency
- One floating-point multiplier, 3-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	**ALU**	**LOAD**	**STORE**	**FADD**	**FMUL**
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`				
`loop:`		`fld f2, A(x2)`			
	`addi x2, x2, 8`	`fld f3, B(x2)`			
				`fadd.d f1, f1, f2`	
					`fmul.d f4, f1, f3`
	`bltu x2, x1, loop`		`fsd f4,(C-8)(x2)`		

Problem 2.A: Loop Unrolling (8 points)

Write the assembly code for unrolling the loop once, so that two iterations of the original code are processed in one unrolled iteration.

Label	Instruction	Comment (optional)
	`addi x1, x0, N*8`	`# initialize loop boundary`
	`addi x2, x0, 0`	`# initialize array index`
`loop:`	`fld f2, A(x2)`	`# load A[i]`
	`fld f3, B(x2)`	`# load B[i]`
	`fld f4, (A+8)(x2)`	`# load A[i+1]`
	`fld f5, (B+8)(x2)`	`# load B[i+1]`
	`addi x2, x2, 16`	`# bump index`
	`fadd.d f1, f1, f2`	`# compute x`
	`fmul.d f6, f1, f3`	`# compute C[i]`
	`fadd.d f1, f1, f4`	`# compute x’`
	`fmul.d f7, f1, f5`	`# compute C[i+1]`
	`fsd f6, (C-16)(x2)`	`# store C[i]`
	`fsd f7, (C-8)(x2)`	`# store C[i+1]`
	`bltu x2, x1, loop`	

Schedule operations with VLIW instructions using only loop unrolling (no software pipelining). Try to optimize for efficiency and minimize the number of cycles, but prioritize correctness. Entries for NOPs can be left blank.

Label	ALU	LOAD	STORE	FADD	FMUL
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`				
`loop:`		`fld f2, A(x2)`			
		`fld f3, B(x2)`			
		`fld f4, (A+8)(x2)`		`fadd.d f1, f1, f2`	
	`addi x2, x2, 16`	`fld f5, (B+8)(x2)`			
				`fadd.d f1, f1, f4`	`fmul.d f6, f1, f3`
			`fsd f6,(C-16)(x2)`		`fmul.d f7, f1, f5`
	`bltu x2, x1, loop`		`fsd f7, (C-8)(x2)`		

Problem 2.B: Software Pipelining (10 points)

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.

If possible, use different colors to distinguish between instructions from different iterations. Entries for NOPs can be left blank.

Label	ALU	LOAD	STORE	FADD	FMUL
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`	`fld f2, A(x2)`			
	`addi x2, x2, 8`	`fld f3, B(x2)`			
				`fadd.d f1, f1, f2`	
		`fld f2, A(x2)`			
	`addi x2, x2, 8`	`fld f3, B(x2)`			
				`fadd.d f1, f1, f2`	`fmul.d f4, f1, f3`
`loop:`		`fld f2, A(x2)`			
	`addi x2, x2, 8`	`fld f3, B(x2)`			
	`bltu x2, x1, loop`		`fsd f4,(C-24)(x2)`	`fadd.d f1, f1, f2`	`fmul.d f4, f1, f3`
			`fsd f4,(C-16)(x2)`		`fmul.d f4, f1, f3`
			`fsd f4,(C-8)(x2)`		

Problem 2.C: Loop Unrolling Performance (1 point)

What is the throughput of the unrolled loop (Part 2.A) in floating-point operations per cycle (FLOPS/cycle)? Only consider the steady-state behavior of the loop. Do not count memory operations.

4/12 = 1/3 FLOPS/cycle

Problem 2.D: Software Pipelining Performance (1 point)

What is the throughput of the software-pipelined loop (Part 2.B) in floating-point operations per cycle (FLOPS/cycle)? Only consider the steady-state behavior of the loop. Do not count memory operations.

2/3 FLOPS/cycle

Problem 2: VLIW (20 points)

In this problem, we will optimize the following code sequence, which implements a scan operation, for a VLIW architecture.

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

```
     # f1 contains initial value of x
     addi x1, x0, N*8         # initialize loop boundary
     addi x2, x0, 0           # initialize array pointer
loop:
     fld f2, A(x2)            # load A[i]
     fld f3, B(x2)            # load B[i]
     addi x2, x2, 8           # bump pointer
     fmul.d f4, f2, f3        # compute product
     fadd.d f1, f1, f4        # compute x
     fsd f1, (C-8)(x2)        # store C[i]
     bltu x2, x1, loop
```

- "`A`", "`B`", and "`C`" are immediates generated by the compiler that encode the base addresses of arrays A, B, and C, respectively.
- Arrays A, B, and C do not overlap in memory.
- N is a large number that is statically known.
- N is evenly divisible as needed for loop unrolling and software pipelining.
- Register `f1` holds the initial value of x.
- Assume that no exceptions arise during execution.

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

- One integer ALU, 1-cycle latency, also used for branches
- Two memory units, 2-cycle latency, both of which can perform either a load or store (ignore the latency of memory-memory dependencies for this problem)
- One floating-point adder, 3-cycle latency
- One floating-point multiplier, 3-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	ALU	MEM0	MEM1	FADD	FMUL
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`				
`loop:`	`addi x2, x2, 8`	`fld f2, A(x2)`	`fld f3, B(x2)`		
					`fmul.d f4, f2, f3`
				`fadd.d f1, f1, f4`	
	`bltu x2, x1, loop`		`fsd f4,(C-8)(x2)`		

Problem 2.A: Loop Unrolling (8 points)

Write the assembly code for unrolling the loop once, so that two iterations of the original code are processed in one unrolled iteration.

Label	Instruction	Comment (optional)
	`addi x1, x0, N*8`	`# initialize loop boundary`
	`addi x2, x0, 0`	`# initialize array index`
`loop:`	`fld f2, A(x2)`	`# load A[i]`
	`fld f3, B(x2)`	`# load B[i]`
	`fld f4, (A+8)(x2)`	`# load A[i+1]`
	`fld f5, (B+8)(x2)`	`# load B[i+1]`
	`addi x2, x2, 16`	`# bump index`
	`fadd.d f6, f2, f3`	`# compute product [i]`
	`fmul.d f6, f1, f6`	`# compute x`
	`fadd.d f7, f4, f5`	`# compute product [i+1]`
	`fmul.d f1, f6, f7`	`# compute x’`
	`fsd f6, (C-16)(x2)`	`# store C[i]`
	`fsd f1, (C-8)(x2)`	`# store C[i+1]`
	`bltu x2, x1, loop`	

Schedule operations with VLIW instructions using only loop unrolling (no software pipelining). Try to optimize for efficiency and minimize the number of cycles, but prioritize correctness. Entries for NOPs can be left blank.

Label	ALU	MEM0	MEM1	FADD	FMUL
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`				
`loop:`		`fld f2, A(x2)`	`fld f3, B(x2)`		
	`addi x2, x2, 16`	`fld f4, (A+8)(x2)`	`fld f5, (B+8)(x2)`		
					`fmul.d f6, f2, f3`
					`fmul.d f7, f4, f5`
				`fadd.d f6, f1, f6`	
			`fsd f6,(C-16)(x2)`	`fadd.d f1, f1, f4`	
	`bltu x2, x1, loop`		`fsd f7, (C-8)(x2)`		

Problem 2.B: Software Pipelining (10 points)

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.

If possible, use different colors to distinguish between instructions from different iterations. Entries for NOPs can be left blank.

Label	ALU	MEM0	MEM1	FADD	FMUL
	`addi x1, x0, N*8`				
	`addi x2, x0, 0`				
	`addi x2, x2, 8`	`fld f2, A(x2)`	`fld f3, B(x2)`		
					`fmul.d f4, f2, f3`
	`addi x2, x2, 8`	`fld f2, A(x2)`	`fld f3, B(x2)`		
				`fadd.d f1, f1, f4`	`fmul.d f4, f2, f3`
`loop:`	`addi x2, x2, 8`	`fld f2, A(x2)`	`fld f3, B(x2)`		
	`bltu x2, x1, loop`		`fsd f1,(C-32)(x2)`	`fadd.d f1, f1, f4`	`fmul.d f4, f2, f3`
			`fsd f1,(C-16)(x2)`	`fadd.d f1, f1, f4`	
			`fsd f1, (C-8)(x2)`		

Problem 2.C: Loop Unrolling Performance (1 point)

What is the throughput of the unrolled loop (Part 2.A) in floating-point operations per cycle (FLOPS/cycle)? Only consider the steady-state behavior of the loop. Do not count memory operations.

4/12 = 1/3 FLOPS/cycle

Problem 2.D: Software Pipelining Performance (1 point)

What is the throughput of the software-pipelined loop (Part 2.B) in floating-point operations per cycle (FLOPS/cycle)? Only consider the steady-state behavior of the loop. Do not count memory operations.

2/3 FLOPS/cycle

Problem 3: Multithreading (20 points)

Consider the following code, which performs an in-place slide operation that moves non-zero elements forward in array A by a displacement M. To parallelize the loop on a multithreaded processor, suppose that we split the loop so that each thread executes every iteration for which `(i % T) == TID`, where T is the total number of threads and TID is a thread identifier from 0 to T-1 inclusive that is uniquely assigned to each thread.

```
for (i = 0; i < N; i++) {
    if (A[i+M] != 0)
        A[i] = A[i+M];
}
```

N and M are arbitrary integers, and N > M and N > T.

The code is executed on a multithreaded in-order core with no data cache, perfect branch prediction with no penalty for both taken and not-taken branches, and no threading overhead.

- Main memory latency is 60 cycles.
- After the processor issues a load, it can continue executing instructions until it reaches an instruction that is dependent on the load value.
- Integer arithmetic operations have a 1-cycle latency.

Problem 3.A (5 points)

For the loop to be safely parallelized this way, what constraint(s) must there be on T, the number of threads? Explain your reasoning.

Assume there is no synchronization among threads while executing the loop.

T must evenly divide M, and T <= M (i.e., gcd(T, M) = T). Due to the loop-carried dependence of A[i] on A[i+M], the ordering of iterations must be preserved to maintain correctness of the code. Since threads can be scheduled arbitrarily, A[i+M] cannot be written by a different thread than the one that writes to A[i].

Problem 3.B (5 points)

Write the assembly code that is executed by each thread. Treat the elements of A as 32-bit integers. You may use any available register, such as `t1`-`t6`.

```
     # A is passed in a0
     # M is passed in a1
     # T is passed in a2
     # TID is passed in a3
     addi t0, a0, N*4    # Initialize loop boundary (t0 = A + N)
     slli a1, a1, 2      # Scale M to bytes
     slli a2, a2, 2      # Scale T to bytes
     slli a3, a3, 2      # Scale TID to bytes
     add a0, a0, a3      # Offset pointer by thread ID
loop:
     add t1, a0, a1
     lw t1, 0(t1)
     beqz t1, skip
     sw t1, 0(a0)
skip:
     add a0, a0, a2
     bltu a0, t0, loop
```

Problem 3.C (5 points)

Suppose that threads are switched every cycle using a fixed round-robin schedule. If the thread is not ready to run on its turn, a bubble is inserted into the pipeline.

What is the minimum number of threads needed to always fully utilize the pipeline while maintaining correct execution? Show your work.

Assume that M=90 and that N is arbitrarily large.

At least 60 threads are required to hide a 60-cycle latency. However, since T must be a factor of M, T should be rounded up to 90.

T=60 was also accepted as a correct answer if it was noted that no memory dependencies would be violated if threads were always scheduled in ascending order by TID.

Problem 3.D (5 points)

Now consider a dynamic scheduling policy that switches threads only when the next instruction would stall due to a data dependency.

What is the minimum number of threads needed to always fully utilize the pipeline while maintaining correct execution? Show your work.

Assume that M=90 and that N is arbitrarily large.

In steady state, each thread can execute 5 instructions before stalling, considering the worst case in which `beqz` is always taken. At least $\lceil (60 - 1)/5 \rceil + 1 = 13$ threads are needed. However, since T must be a factor of M, T should be rounded up to 15.

Problem 3: Multithreading (20 points)

Consider the following code, which performs an in-place slide operation that moves non-zero elements forward in array A by a displacement M. To parallelize the loop on a multithreaded processor, suppose that we split the loop so that each thread executes every iteration for which `(i % T) == TID`, where T is the total number of threads and TID is a thread identifer from 0 to T-1 inclusive that is uniquely assigned to each thread.

```
for (i = 0; i < N; i++) {
    if (A[i+M] != 0)
        A[i] = A[i+M];
}
```

N and M are arbitrary integers, and N > M and N > T.

The code is executed on a multithreaded in-order core with no data cache, perfect branch prediction with no penalty for both taken and not-taken branches, and no threading overhead.

- Main memory latency is 40 cycles.
- After the processor issues a load, it can continue executing instructions until it reaches an instruction that is dependent on the load value.
- Integer arithmetic operations have a 1-cycle latency.

Problem 3.A (5 points)

For the loop to be safely parallelized this way, what constraint(s) must there be on T, the number of threads? Explain your reasoning.

Assume there is no synchronization among threads while executing the loop.

T must evenly divide M, and T <= M (i.e., gcd(T, M) = T). Due to the loop-carried dependence of A[i] on A[i+M], the ordering of iterations must be preserved to maintain correctness of the code. Since threads can be scheduled arbitrarily, A[i+M] cannot be written by a different thread than the one that writes to A[i].

Problem 3.B (5 points)

Write the assembly code that is executed by each thread. Treat the elements of A as 32-bit integers. You may use any available register, such as `t1`-`t6`.

```
     # A is passed in a0
     # M is passed in a1
     # T is passed in a2
     # TID is passed in a3
     addi t0, a0, N*4    # Initialize loop boundary (t0 = A + N)
     slli a1, a1, 2      # Scale M to bytes
     slli a2, a2, 2      # Scale T to bytes
     slli a3, a3, 2      # Scale TID to bytes
     add a0, a0, a3      # Offset pointer by thread ID
loop:
     add t1, a0, a1
     lw t1, 0(t1)
     beqz t1, skip
     sw t1, 0(a0)
skip:
     add a0, a0, a2
     bltu a0, t0, loop
```

Problem 3.C (5 points)

Suppose that threads are switched every cycle using a fixed round-robin schedule. If the thread is not ready to run on its turn, a bubble is inserted into the pipeline.

What is the minimum number of threads needed to always fully utilize the pipeline while maintaining correct execution? Show your work.

Assume that M=70 and that N is arbitrarily large.

At least 40 threads are required to hide a 40-cycle latency. However, since T must be a factor of M, T should be rounded up to 70.

T=40 was also accepted as a correct answer if it was explicitly noted that no memory dependencies would be violated if threads are always scheduled in ascending order by TID.

Problem 3.D (5 points)

Now consider a dynamic scheduling policy that switches threads only when the next instruction would stall due to a data dependency.

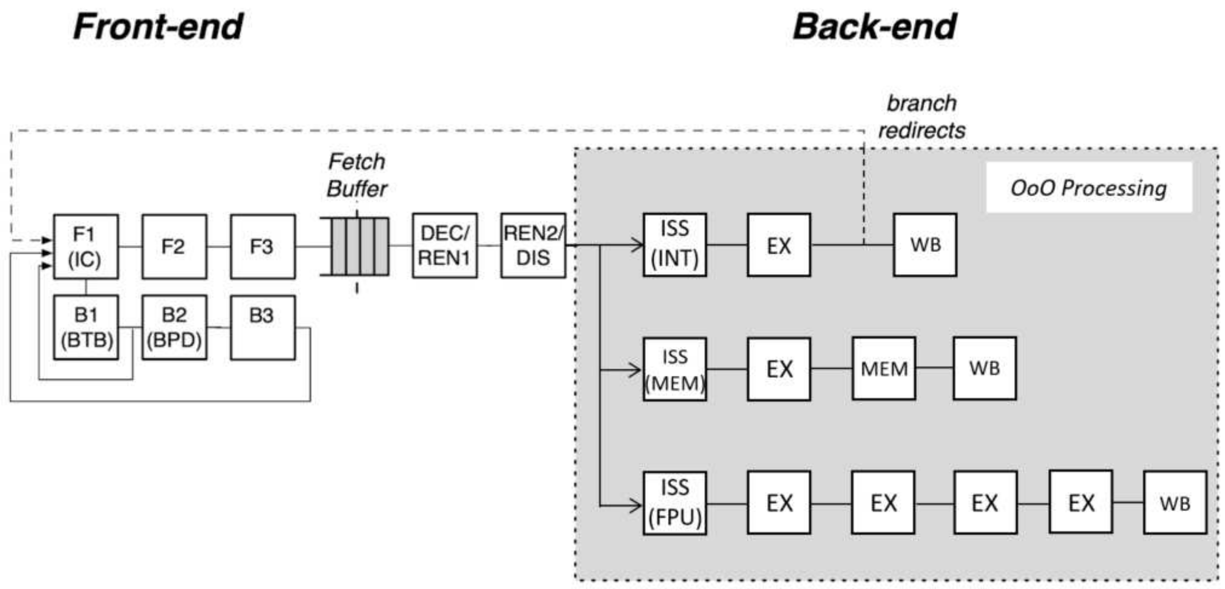
What is the minimum number of threads needed to always fully utilize the pipeline while maintaining correct execution? Show your work.

Assume that M=70 and that N is arbitrarily large.

In steady state, each thread can execute 5 instructions before stalling, considering the worst case in which `beqz` is always taken. At least $\lceil(40-1)/5\rceil + 1 = 9$ threads are needed. However, since T must be a factor of M, T should be rounded up to 10.

Problem 4: Out-of-Order Execution (20 points)

For this problem, we consider the following dual-issue out-of-order superscalar processor with a unified physical register file.



Dispatch

- **Up to two instructions** can be renamed and dispatched per cycle.
- Register renaming follows the **unified physical register file** scheme.
- Instructions are written into the ROB at the end of the REN2/DIS stage.

Issue

- **There are three issue windows separate from the ROB:**
    - o ALU operations and branches (INT)
    - o Memory operations (MEM)
    - o Float-point operations (FP)
- Instructions are written into the issue window at the end of the REN2/DIS stage.
- **Up to one instruction** can be issued to execution per cycle from each issue window.
- Assume that an age-based scheduler always selects the oldest instruction to issue.
- An instruction may issue in the same cycle when the last operand that it depends on is in the writeback stage.
- Register operands are read during the ISS stage.
- The physical register file has 6 read and 3 write ports. (Assume no structural hazards.)

Execution

- All functional units are fully pipelined with the following latencies:
    - ALU operations: 1 cycle
    - Loads and stores: 2 cycles latency (Assume that all accesses hit in the data cache.)
    - Floating-point operations: 4 cycles
- Writeback occurs in a separate WB stage.
- **Mispredicted branches redirect the frontend and trigger a pipeline flush in the cycle after they are resolved in the INT EX stage.**

Commit

- **Up to two instructions** can be committed per cycle.
- Commit is handled by a decoupled unit that looks at the ROB entries.
- The earliest that an instruction can commit is in the cycle following writeback.

Problem 4.A (10 points)

The following instruction sequence is executed on the out-of-order core described above.

`fmul f1, f3, f2`	
`add x1, x1, x2`	
`fsw f1, 4(x1)`	
`lw x2, 0(x3)`	
`bnez x2, done`	Misprediction
`flw f1, 0(x1)`	Page fault
`fadd f3, f1, f2`	

Fill out the table with the cycles at which instructions enter the ROB, issue to the functional units, complete and write back to the physical register file, and commit. If an instruction is killed before issuing, completing, or committing, mark the corresponding entries with "–".

- The ROB is initially empty and contains enough entries for the instructions shown.
- All instructions are present already in the fetch buffer.
- `bnez` is initially predicted to be not taken but later resolves as taken. The "`done`" branch target points to unrelated code elsewhere.
- A page fault is detected for `flw`.

The first instruction has been done for you.

	Dispatch	Issue	Completion	Commit
`fmul`	0	1	6	7
`add`	0	1	3	7
`fsw`	1	6	9	10
`lw`	1	2	5	10
`bnez`	2	5	7	11
`flw`	2	3	6	-
`fadd`	3	-	-	-

Problem 4.B (10 points)

For the same code as Part 4.A (reproduced below), show the state of the ROB, issue windows, rename table, and free list **in the cycle after recovering from all mispredicts and exceptions** – i.e., immediately after precise architectural state has been restored and the processor has been redirected to the correct branch target. Assume that mispredictions and exceptions use the same rollback procedure, which happens instantaneously.

The first instruction has been done for you.

`fmul f1, f3, f2`	
`add x1, x1, x2`	
`fsw f1, 4(x1)`	
`lw x2, 0(x3)`	
`bnez x2, done`	Misprediction
`flw f1, 0(x1)`	Page fault
`fadd f3, f1, f2`	

The same assumptions as Part 4.A apply:

- The ROB and issue windows are initially empty.
- `bnez` is initially predicted to be not taken but later resolves as taken. The "`done`" branch target points to unrelated code elsewhere.
- A page fault is detected for `flw`.

For each entry in the rename table, show all changes in sequence. Unused architectural registers are omitted from the rename table for clarity.

The free list is treated as a FIFO, and entries are dequeued from the top and appended to the bottom. Cross out (or mark with "x") the entries from the free list that have been dequeued.

Rename Table	
x1	P10 → P4
x2	P7 → P6
x3	P2
f1	P5 → P9 → P3 → P9
f2	P8
f3	P11 → P1 → P11

Free List	
~~P9~~	x
~~P4~~	x
~~P6~~	x
~~P3~~	x
~~P1~~	x
P12	
P1 (or P5)	
P3 (or P10)	
P5 (or P1)	
P10 (or P3)	

Show the previous contents of all deallocated entries in the ROB and issue windows, and assume they are not immediately reused.

Reorder Buffer				
#	Done?	Rd	LPRd	Exception?
0	1	`f1`	P5	
1	1	`x1`	P10	
2	0	-	-	
3	1	`x2`	P7	
4	1	-	-	
5	1	`f1`	P9	Page fault
6	0	`f3`	P11	
7				

INT Issue Window								
Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`add`	1	P10	1	P7	P4	1
0	1	`bnez`	1	P6	-	-	-	4

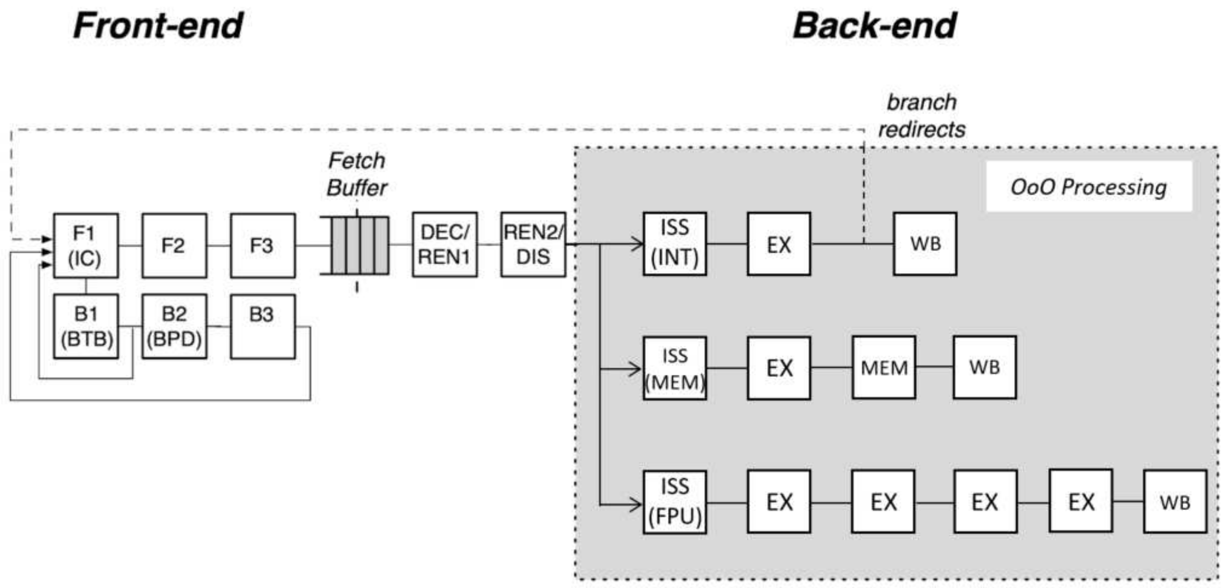
MEM Issue Window								
Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`fsw`	1	P9	1	P4	-	2
0	1	`lw`	-	-	1	P2	P6	3
0	1	`flw`	-	-	1	P4	P3	5

FP Issue Window								
Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`fmul`	1	P11	1	P8	P9	0
0	0	`fadd`	0	P3	1	P8	P1	6

The rollback from the branch misprediction happens in cycle 7 (when `bnez` "completes"), and therefore the state of these structures should reflect cycle 8 in Part 4.A. At that point, `fmul` and `add` have committed but *not* any subsequent instructions. Since the rollback and commit occur in the same cycle, there is some ambiguity about whether the freed registers or restored registers are enqueued first onto the free list, so either order was accepted.

Problem 4: Out-of-Order Execution (20 points)

For this problem, we consider the following dual-issue out-of-order superscalar processor with a unified physical register file.



Dispatch

- **Up to two instructions** can be renamed and dispatched per cycle.
- Register renaming follows the **unified physical register file** scheme.
- Instructions are written into the ROB at the end of the REN2/DIS stage.

Issue

- **There are three issue windows separate from the ROB:**
  - ALU operations and branches (INT)
  - Memory operations (MEM)
  - Float-point operations (FP)
- Instructions are written into the issue window at the end of the REN2/DIS stage.
- **Up to one instruction** can be issued to execution per cycle from each issue window.
- Assume that an age-based scheduler always selects the oldest instruction to issue.
- An instruction may issue in the same cycle when the last operand that it depends on is in the writeback stage.
- Register operands are read during the ISS stage.
- The physical register file has 6 read and 3 write ports. (Assume no structural hazards.)

Execution

- All functional units are fully pipelined with the following latencies:
    - ALU operations: 1 cycle
    - Loads and stores: 2 cycles latency (Assume that all accesses hit in the data cache.)
    - Floating-point operations: 4 cycles
- Writeback occurs in a separate WB stage.
- **Mispredicted branches redirect the frontend and trigger a pipeline flush in the cycle after they are resolved in the INT EX stage.**

Commit

- **Up to two instructions** can be committed per cycle.
- Commit is handled by a decoupled unit that looks at the ROB entries.
- The earliest that an instruction can commit is in the cycle following writeback.

Problem 4.A (10 points)

The following instruction sequence is executed on the out-of-order core described above.

`fmul f1, f3, f2`	
`add x1, x1, x2`	
`fsw f1, 4(x1)`	
`lw x2, 0(x3)`	
`bnez x2, done`	Misprediction
`flw f1, 0(x1)`	Page fault
`fadd f3, f1, f2`	

Fill out the table with the cycles at which instructions enter the ROB, issue to the functional units, complete and write back to the physical register file, and commit. If an instruction is killed before issuing, completing, or committing, mark the corresponding entries with "–".

- The ROB is initially empty and contains enough entries for the instructions shown.
- All instructions are present already in the fetch buffer.
- `bnez` is initially predicted to be not taken but later resolves as taken. The "`done`" branch target points to unrelated code elsewhere.
- A page fault is detected for `flw`.

The first instruction has been done for you.

	Dispatch	Issue	Completion	Commit
`fmul`	0	1	6	7
`add`	0	1	3	7
`fsw`	1	6	9	10
`lw`	1	2	5	10
`bnez`	2	5	7	11
`flw`	2	3	6	-
`fadd`	3	-	-	-

Problem 4.B (10 points)

For the same code as Part 4.A (reproduced below), show the state of the ROB, issue windows, rename table, and free list **in the cycle after recovering from all mispredicts and exceptions** – i.e., immediately after precise architectural state has been restored and the processor has been redirected to the correct branch target. Assume that mispredictions and exceptions use the same rollback procedure, which happens instantaneously.

The first instruction has been done for you.

`fmul f1, f3, f2`	
`add x1, x1, x2`	
`fsw f1, 4(x1)`	
`lw x2, 0(x3)`	
`bnez x2, done`	Misprediction
`flw f1, 0(x1)`	Page fault
`fadd f3, f1, f2`	

The same assumptions as Part 4.A apply:

- The ROB and issue windows are initially empty.
- `bnez` is initially predicted to be not taken but later resolves as taken. The "`done`" branch target points to unrelated code elsewhere.
- A page fault is detected for `flw`.

For each entry in the rename table, show all changes in sequence. Unused architectural registers are omitted from the rename table for clarity.

The free list is treated as a FIFO, and entries are dequeued from the top and appended to the bottom. Cross out (or mark with "x") the entries from the free list that have been dequeued.

Rename Table	
`x1`	P3 → P11
`x2`	P8 → P16
`x3`	P5
`f1`	P12 → P2 → P14 → P2
`f2`	P6
`f3`	P9 → P7 → P9

Free List	
~~P2~~	x
~~P11~~	x
~~P16~~	x
~~P14~~	x
~~P7~~	x
P4	
P7 (or P12)	
P14 (or P3)	
P12 (or P7)	
P3 (or P14)	

Show the previous contents of all deallocated entries in the ROB and issue windows, and assume they are not immediately reused. Not all entries may be needed.

Reorder Buffer

#	Done?	Rd	LPRd	Exception?
0	1	`f1`	P12	
1	1	`x1`	P3	
2	0	-	-	
3	1	`x2`	P8	
4	1	-	-	
5	1	`f1`	P2	Page fault
6	0	`f3`	P9	
7				

INT Issue Window

Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`add`	1	P3	1	P8	P11	1
0	1	`bnez`	1	P16	-	-	-	4

MEM Issue Window

Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`fsw`	1	P2	1	P11	-	2
0	1	`lw`	-	-	1	P5	P16	3
0	1	`flw`	-	-	1	P11	P14	5

FP Issue Window

Used?	Exec?	Operation	p1	PR1	p2	PR2	PRd	ROB #
0	1	`fmul`	1	P9	1	P6	P2	0
0	0	`fadd`	0	P14	1	P6	P7	6

The rollback from the branch misprediction happens in cycle 7 (when `bnez` "completes"), and therefore the state of these structures should reflect cycle 8 in Part 4.A. At that point, `fmul` and `add` have committed but *not* any subsequent instructions. Since the rollback and commit occur in the same cycle, there is some ambiguity about whether the freed registers or restored registers are enqueued first onto the free list, so either order was accepted.