Computer System Architecture
6.5900 Quiz #3
December 11th, 2024

Name: _____

This is a closed book, closed notes exam.
80 Minutes
15 Pages (+1 Scratch)

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Page 16 is a scratch page. Use it if you need more space to answer one of the questions, or for rough work.

Part A	_____	39 Points
Part B	_____	30 Points
Part C	_____	31 Points
TOTAL	**_____**	**100 Points**

Name _____

Part A: VLIW + Vector Processors (39 Points)

Consider the following piece of code:

```
// Assume: prefilled w/ uniform random numbers in [0, SIZE)
// Values of SIZE will be provided in later questions.
int32_t array[SIZE];

// Values of WIDTH will be provided in later questions
int32_t indices[WIDTH] = { 0, 1, … WIDTH – 1 };

void func(int* result) {
     for (int32_t round = 0; round < (1 << 20); round++) {
          for (int32_t i = 0; i < WIDTH; i++) {
               int32_t x = array[indices[i]];
               result[i] += (x & 1);
               indices[i] = x;
          }
     }
}
```

In RISC-V assembly, this compiles to:

```
     ...
.outer:
     mv a5, sp            // indices is stack allocated
     mv a4, a0            // a4 = `result`
.inner:
     lw a1, 0(a5)         // a1 = indices[i]
     slli a1, a1, 2       // a1 = a1 * 4
     add a1, a1, t1       // compute index into array
     lw a1, 0(a1)         // a1 = array[indices[i]]
     lw a2, 0(a4)         // a2 = results[i]
     andi a3, a1, 1       // a3 = array[indices[i] & 1
     sw a1, 0(a5)         // indices[i] = x
     add a2, a2, a3       // a2 += array[indices[i]] & 1
     sw a2, 0(a4)         // store out results[i]
     addi a4, a4, 4       // increment ptr into `results`
     addi a5, a5, 4       // increment ptr into `indices`
     bne a4, t0, .inner   // inner loop
     addi a7, a7, 1       // increment `rounds`
     bne a7, a6, .outer   // outer loop
     ...
```

Name _____

Question 1 (10 points)

We now define the following in-order, single-threaded VLIW processor. Every cycle, it can issue a single **instruction**. A instruction contains several **independent** "operations"--data (RAW, WAW, WAR) or control dependences between operations in the same bundle is undefined behavior.

A bundle as 4 **operation slots**. Each slot can contain 1 ordinary RISC-V instruction. Precisely, each instruction has the following slots:

- 1 load/store operation slot: can contain lw, sw
- 2 ALU operation slots: can contain slli, add, andi, addi
- 1 control operation slot: can contain bne

All ALU instructions take 1 cycle to execute. The processor has full bypassing, so later instructions can use the results of earlier instructions on the next cycle. Memory operations take 1 cycle on a cache hit and 100 cycles on a cache miss. The processor has stalling logic to stall the entire pipeline on a cache miss. So, the results of memory operations **can** be used in the next cycle.

Compile the assembly above into VLIW instructions for this architecture. Use the table below. Do not modify the assembly from the scalar code! Note: not all rows must be used.

Instruction Number	Load/Store Slot	ALU 1 Slot	ALU 2 Slot	Control Slot
1		mv a5, sp	mv a5, sp	
2	lw a1, 0(a5)			
3		slli a1, a1, 2		
4	lw a2, 0(a4)	add a1, a1, t1		
5	lw a1, 0(a1)			
6	sw a1, 0(a5)	andi a3, a1, 1		
7		add a2, a2, a3		
8	sw a2, 0(a4)			
9		addi a4, a4, 4	addi a5, a5, 4	
10				bne a4, t0, .inner
11			addi a7, a7, 1	
12				bne a7, a6, .outer
13				

Name _____

Question 2 (5 points)

This VLIW processor has the following memory system:

- 32 KB 16-way set associative cache with **32 byte** cache lines
- Cache hits take 1 cycle, cache misses take 100 cycles
- Assume that the results and indices arrays are both cache-line aligned and *are always cached, never evicted*

Assuming no branch mispredictions (branches induce *no* stall cycles), how many loop iterations per cycle would you expect your VLIW code to achieve in the following scenarios? (your answer will be a fraction, it is less than 1!) You should assume that time spent evaluating the outer loop is negligible.

- Scenario 1: `SIZE = (1 << 7), WIDTH = 128`

  Everything is a cache hit. 9 instructions take 9 cycles, so 1/9 iterations per cycle

- Scenario 2: `SIZE = (1 << 30), WIDTH = 128`

  Loading from array is nearly always a cache miss that takes 100 cycles, so 1/108 iterations per cycle

Name _____

Question 3 (8 points)

Suppose that we compile a version of this code with WIDTH = 64 and decide to unroll the *entire inner loop* (unroll WIDTH-many times) and software-pipeline it optimally. Assuming no cache misses, infinite registers, and perfect branch prediction. In steady state, how many of the original inner loop iterations are completed every cycle?

You do not need to write out the instructions.
Be careful: loop unrolling happens in the source code-- the code will be recompiled to take advantage of the infinite available registers! It also will **remove** the following 3 instructions that are no longer necessary:

```
addi a4, a4, 4       // increment ptr into `results`
addi a5, a5, 4       // increment ptr into `indices`
bne a4, t0, .inner   // inner loop
```

6 ALU ops per iteration, 1 control op per iteration, 5 memory ops per iteration. So, given 2 ALU slots, 1 load/store slots, and 1 control slot, the load store slot is the bottleneck.

This means we'll get 1/5 iterations per cycle.

Question 4 (8 points)

Instead of VLIW, we'll try to execute this code on a *vector processor* instead. We take a pipelined in-order RISC-V processor and add the following ISA extensions:

Vector Registers: v0, v1, v2, v3, v4, v5, v6, v7 (all hold VLEN-many 32-bit values)

Note: vd, vrs1, vrs2 are vector registers. rd, rs1, rs2 are normal scalar registers.

Instruction	Semantics
vlw vd, imm(rs1)	vd[i] ← mem[imm + rs1 + 4 * i]
vsw vrs2, imm(rs1)	mem[imm + rs1 + 4 * i] ← vrs2[i]
vgather vd, vrs1	vd[i] ← mem[vrs1[i]]
vssli vd, vrs1, imm	vd[i] ← vrs1[i] << imm
vadd vd, vrs1, rs2	vd[i] ← vrs1[i] + rs2
vaddi vd, vrs1, imm	vd[i] ← vrs1[i] + imm
vandi vd, vrs1, imm	vd[i] ← vrs1[i] & imm

Assume that the architecture has VLEN many lanes, all compute operations take one cycle, and we have full bypassing.

Suppose VLEN = 8 and VLEN divides WIDTH. We can now recompile our code to take advantage of the vector ISA:

```
      ...
.outer:
      mv a5, sp            // indices is stack allocated
      mv a4, a0            // a4 = `result`
.inner:
      vlw v1, 0(a5)        //
      vslli v1, v1, 2      //
      vadd v1, v1, t1      //
      vgather v1, 0(v1)    //
      vlw v2, 0(a4)        //
      vandi v3, v1, 1      //
      vsw v1, 0(a5)        //
      vadd v2, v2, v3      //
      vsw v2, 0(a4)        //
      addi a4, a4, 32      //
      addi a5, a5, 32      //
      bne a4, t0, .inner   //
      addi a7, a7, 1       //
      bne a7, a6, .outer   //
     ...
```

Note: the memory subsystem only supports loading 1 cache line per cycle. So, if a vgather instruction loads values that span across multiple cache lines, it must start those loads on *consecutive cycles*. The memory system can support VLEN many in-flight memory operations (one memory operation is defined as a load or a store to a unique cache line). Assume the same memory hierarchy as the previous questions.

Question 4 continued

Assuming no branch mispredictions, what IPC would you expect this new vectorized code to achieve in the following scenarios? Assume time spent executing outer-loop related instructions is negligible.

- Scenario 1: `SIZE = (1 << 7), WIDTH = 128`

  Every instruction except the vgather takes 1 cycle. The vgather is just cache hits and takes 8 cycles. So, 12 instructions / (11 + 8 cycles) = 12 / 19

- Scenario 2: `SIZE = (1 << 30), WIDTH = 128`

  Same as previous part except vgather takes 107 cycles because all the accesses are cache misses on different lines, so 12 / 118

Name _____

Question 5 (8 points)

Now, we will examine the same code running on a simple GPU. Suppose we have SIZE = (1 << 30). We spawn WIDTH-many *threads* (Nvidia terminology) each executing the computation for a single i (inner loop iteration):

```
    ...
// assume a6 contains a thread's index
// sp contains the base address of `indices`
// a0 contains the base address of `results`
     slli a7, a6, 2
     add a5, sp, a6
     add a4, a0, a6
.outer:
     lw a1, 0(a5)          // a1 = indices[i]
     slli a1, a1, 2        // a1 = a1 * 4
     add a1, a1, t1        // compute index into array
     lw a1, 0(a1)          // a1 = array[indices[i]]
     lw a2, 0(a4)          // a2 = results[i]
     andi a3, a1, 1        // a3 = array[indices[i] & 1
     sw a1, 0(a5)          // indices[i] = x
     add a2, a2, a3        // a2 += array[indices[i]] & 1
     sw a2, 0(a4)          // store out results[i]
     addi a7, a7, 1        // increment `rounds`
     bne a7, t2, .outer    // t2 holds the outer loop bound
     ...
```

However, since this is a GPU and not a multicore, the threads are grouped together in *warps* (Nvidia terminology). Each warp consists of *8 consecutive threads*. So, for example threads with index 0, 1, … 7 all execute together as part of warp 0. Each warp executes instructions *in lockstep* using all 8 available *lanes*.

The memory system can still only load or store one cache line per cycle! If the threads in a warp issue loads (or stores) to different cache lines, those loads (or stores) will be serialized and issue on consecutive cycles. During those cycles, the GPU is *not* ready to issue a new instruction. Example: if a warp lw needs to load 8 cache lines, that instruction will **issue over 8 cycles**. These 8 cycles *do not* count as system stalls—the GPU is doing useful work! There is no limit on the number of inflight memory operations.

The GPU keeps issuing instructions from the *same warp* every cycle until that warp stalls. Upon detecting a stall, it switches to the next non-stalled warp in round-robin order.

Name _____

Question 5 continued

What is the minimum value of WIDTH for which this GPU will not experience any stalls *in steady state*?

We have 10 “uninteresting” instructions in the loop (every instruction except for the gather load to `array`). These each take 1 cycle of activity. The gather load takes 8 cycles of activity, as it needs to load 8 separate cache lines.

Assuming a single warp, this means that each warp does 18 cycles of activity then stalls for 100. We need enough warps to fill up the 100 cycle stall. This means we need ceil(100 / 18) = 6 more warps for a total of 7 warps = 56 threads = WIDTH = 56

Name _____

Part B: Security + Reliability (30 Points)

Sidney (Sid for short) Channel is currently executing computations on sensitive private data on a shared uniprocessor system.

Little does he know, but an evil mad scientist, Dr. Fission Chips, is trying to steal his data. Dr. Chips has the following two tools at his disposal:

- He too can execute code on this system
- He can point a neutron beam *very precisely* at some of the chip's hardware structures. This neutron beam, when activated, will randomly flip bits in the target hardware structure.

You should assume that the operating system is time-sharing the processor between Sid and Dr. Chips' code. The operating system is using all standard memory protection mechanisms and has no bugs.

Question 1 (5 points)

Suppose Dr. Chips points his neutron beam at the processor's TLB. Explain how flipping some TLB bits could potentially lead to security vulnerabilities.

The TLB caches translations from VPNs to PPNs. Bits from one of Dr. Chips' PPNs could flip to point to a PPN in Sid's address space, giving Dr. Chips access to Sid's data.

Question 2 (10 points)

Assume that Dr. Chips is pointing his neutron beam at the TLB. Describe in detail *what code* Dr. Chips could run to take advantage of TLB bit flips to extract Sid's data. How would this code potentially leak Sid's data?

```
char* page = malloc(PAGE_SIZE);
memset(page, 0, PAGE_SIZE);

while (true) {
  for (int i = 0; i < PAGE_SIZE; i++) {
     if (page[i] != 0) printf("I see some data that might be private!");
  }
}
```

Name _____

Question 3 (10 points)

Now, Dr. Chips points his neutron beam at the virtually-indexed physically tagged (VIPT) L1 cache *tag array*. The cache is *not* flushed on context switches. Explain how flipping some bits in the cache could potentially lead to security vulnerabilities. For the purposes of this question, the cache tag array *only* stores physical tags and no additional metadata.

Suppose that Sid has some data X in the cache with physical tag A. Some bits flip, and this tag now becomes A'.

If A' is a PPN that is mapped within Dr. Chip's address space, then Dr. Chips can read X by loading the address in his address space that maps to PPN A' with the same virtual page offset.

Question 4 (5 points)

Which of these two attacks (neutron beam on cache or neutron beam on TLB) is more likely to leak Sid's sensitive data? You do not need to calculate exact probabilities. Simply explain why one attack will leak data at a higher rate than the other.

The TLB. A very small number of bit-flips can grant relatively long-lived access to an entire page's worth of data.

However, we accepted both answers given good assumptions and sufficient explanation.

Name _____

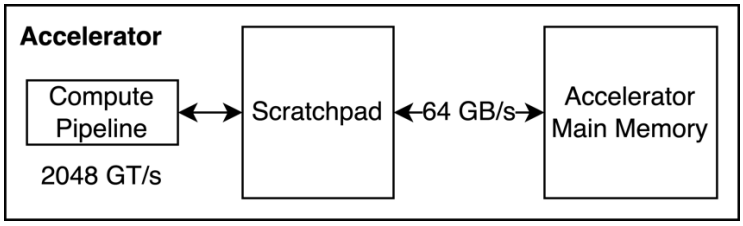
Part C: Accelerators (31 Points)

In this part, you will answer questions about a simple accelerator designed to speed up the following naïve shading algorithm:

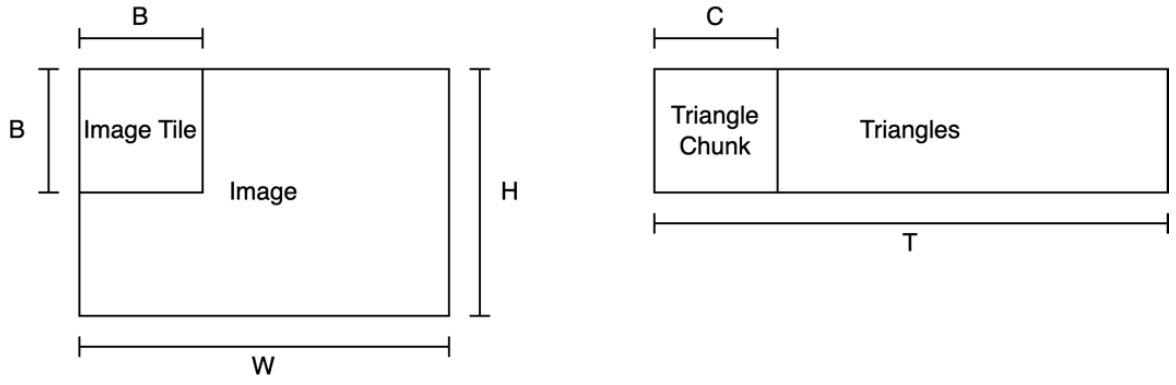
```
const triangle_t triangles[T];
color_t img[W][H];

for (int x = 0; x < W; x++) {
     for (int y = 0; y < H; y++) {
          for (int t = 0; t < T; t++) {
               img[x][y] += shade(triangles[t], x, y);
          }
     }
}
```

color_t is an opaque 4 byte structure representing the color of a pixel and triangle_t is an opaque 64 byte structure representing a triangle in the scene.

Here is an overview of our accelerator's architecture:



To execute computation, the accelerator loads blocks of pixels and chunks of triangles from accelerator main memory. Once both the pixel block and the triangle chunk are fully present in the scratchpad, they are fed into the "compute pipeline" that computes the cumulative color contributions of the triangle chunk onto the pixel block. The compute pipeline can execute 2048 GT/s (1 GT/s = 2^{30} calls to shade/second).
The exact order in which it does this will be specified in future questions. Assume all operands start in accelerator main memory.



Name _____

Question 1 (21 points)

We are rendering T = 256 triangles onto a W = 4096 x H = 2048 image. Suppose B = 64 and C = 32.

You are presented with the following two computation schedules:

Schedule A	**Schedule B**

`for w in range(0, W, B):`
`  for h in range(0, H, B):`
`  pb = load(img[w:w+B, h:h+B])`
`  for t in range(0, T, C):`
`  tc = load(triangles[t:t+C])`
`  pb += pipeline(pb, tc)`
`  store(pb, img[w:w+B, h:h+B])`
	`for t in range(0, T, C):`
	`  tc = load(triangles[t:t+C])`
	`  for w in range(0, W, B):`
	`  for h in range(0, H, B):`
	`  pb = load(img[w:w+B, h:h+B])`
	`  pb += pipeline(pb, tc)`
	`  store(pb, img[w:w+B, h:h+B])`

Loads and stores move data *between accelerator main memory and the scratchpad.* Assume the scratchpad is large enough to hold one triangle chunk and one pixel block.

Just for Q1, assume that the accelerator is bottlenecked on either *memory bandwidth* or *compute throughput*.

(Q1 Part A: 4 points): compute the total data movement between accelerator main memory and the scratchpad for schedule A. How many seconds would this data movement take?

We load the entire image once and store the entire image once, so:

W * H * 4 bytes * 2 (load and then store) = 2^12 * 2^11 * 2^2 * 2^1 = 2^26

We load the entire triangle array (W / B) * (H / B) times. The triangle array is 256 * 64 bytes.
(W / B) * (H / B) = (2^12 / 2^6) * (2^11 / 2^6) = 2^11. Size of triangle array is 2^6 * 2^8 = 2^14.

Total data movement is (2^25 + 2^26 = 3 * 2^25) bytes. This takes (3 * 2^25) / (2^6 * 2^30) = 3 / 2^11 seconds

(Q1 Part B: 4 points): compute the total data movement between accelerator main memory and the scratchpad for schedule B. How many seconds would this data movement take?

We load the entire triangle array once. This is 2^14 bytes (from previous question). We load and store the entire image (T / C) = 8 = 2^3 times.

From the previous question, we know loading and storing the image once is 2^26 bytes. Doing this 2^3 times will be 2^29 bytes.

Our total data movement is therefore 2^14 + 2^29 bytes.

This will take roughly (2^29 / (2^6 * 2^30)) = 1 / 2^7 seconds (we can ignore the 2^14, it's insignificant).

(Q1 Part C: 4 points): *if the accelerator was compute bound*, how many seconds would it take for it to perform this computation? (schedule A and schedule B perform the same amount of computation)

We compute W * H * T triangles = 2^12 * 2^11 * 2^8 = 2^31.

Our available compute throughput is 2048 GT/s = 2^41 T/s.

So, this will take 2^31 / 2^41 = 2^{-10} seconds.

(Q1 Part D: 3 points): Is schedule A compute bound or memory bound? How many seconds would it take the accelerator to execute this computation using schedule A?

If this was memory bound, it'd take 3 * 2^{-11} seconds (from part A)
If this was compute bound, it'd take 2^{-10} seconds (from part C)

3 * 2^{-11} > 2^{-10} so this is memory bound and takes 3 * 2^{-11} seconds.

(Q1 Part E: 3 points): Is schedule B compute bound or memory bound? How many seconds would it take the accelerator to execute this computation using schedule B?

If this was memory bound, it'd take 2^{-7} seconds (from part B)
If this was compute bound, it'd take 2^{-10} seconds (from part C)

2^{-7} > 2^{-10} so this is memory bound and takes 2^{-7} seconds.

(Q1 Part F: 3 points): Suppose we pick **schedule B**. What is the utilization of our compute pipeline? Utilization = (GT/s achieved / maximum achievable GT/s)

We execute 2^{31} triangles in 2^{-7} seconds. This means we do:

2^{31} / 2^{-7} = 2^38 triangles / second. Maximum achievable throughput is 2^41 triangles per second.

2^38 / 2^41 = 1/8

*The answer to Question 2 may depend on values that you computed in Question 1. To avoid double penalizing wrong answers to Question 1, please **clearly mark** which values in your computation come from Question 1.*

Question 2 (10 points)

(Q2 Part A: 5 points): Suppose that we are executing **schedule B** from Q1. We still have W = 4096, H = 2048, T = 256, B = 64, C = 32. We allocate sufficient scratchpad space to store tc, but only have 16K of scratchpad space left (enough to store a single tile of pb). This means that each iteration of the inner loop is necessarily serialized.

Each iteration must load, compute, then store back sequentially before beginning the next iteration. Roughly what fractions of *both compute throughput and memory bandwidth* are actually utilized? Please justify your answer.

We know from Q1.B that the data transfer time is 2^{-7} seconds.

We know from Q1.C that the compute time is 2^{-10} seconds.

Our total time is 2^{-7} + 2^{-10}

This means that we use 2^{-7} / (2^{-7} + 2^{-10}) = 8/9 of memory bandwidth
and 2^{-10} / (2^{-7} + 2^{-10}) = 1/9 of compute throughput

(Q2 Part B: 5 points): To address this under-utilization, we can run several innermost-loop iterations in parallel. However, doing so would require more scratchpad space. Roughly how much more scratchpad space would be required to fully utilize *at least one of the two resources*? Assume that we only load/store whole pixel blocks/triangle chunks (not fractional ones).

Getting from 8/9 to full memory bandwidth doesn't require much more— we simply need to overlap one more tile to achieve full memory bandwidth utilization.

This requires B * B * 4 = 16384 more bytes of scratchpad space.

Name _____

Scratch Space