Computer Architecture I Mid-Term II

Chinese Name: _____

Pinyin Name: _____

Student ID: _____

E-Mail ... @shanghaitech.edu.cn: _____

Question	Points	Score
1	1	
2	26	
3	11	
4	10	
5	6	
6	29	
7	10	
8	7	
Total:	100	

- This test contains 24 numbered pages, including the cover page, printed on both sides of the sheet.
- We will use gradescope for grading, so only answers filled in at the obvious places will be used.
- Use the provided blank paper for calculations and then copy your answer here.
- Please turn **off** all cell phones, smartwatches, and other mobile devices. Remove all hats and headphones. Put everything in your backpack. Place your backpacks, laptops and jackets out of reach.
- The total estimated time is 105 minutes.

- You have 120 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use one A4 page (front and back) of handwritten notes in addition to the provided green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided.
- Do **NOT** start reading the questions/ open the exam until we tell you so!

[1] 1. **First Task (1 point): Fill in you name**
Fill in your name and email on top of every page (without @shanghaitech.edu.cn) (so write your name in total 24 times).

2. **MISC (26 Points)**

[2] (a) Which of the following scenarios best exemplifies data-level parallelism? _____

A. A multi-core CPU distributes different tasks across different cores.
B. A vector processor executes a single instruction to add four pairs of numbers in two arrays simultaneously.
C. A pipelined CPU overlaps the fetch, decode, and execute stages of consecutive instructions.
D. A web server handles multiple client requests concurrently through thread pooling.

Solution: B

[2] (b) In a classic 5-stage instruction pipeline (IF → ID → EX → MEM → WB), assuming there is no forwarding mechanism, which of the following instruction pairs does **NOT** cause a data hazard? _____

A. `ADD R1, R2, R3; SUB R4, R1, R5`
B. `LW R1, 0(R2); ADD R3, R1, R4`
C. `ADD R1, R2, R3; SW R1, 0(R4)`
D. `LW R1, 0(R2); SW R2, 0(R3)`

Solution: D

[2] (c) Choose statement(s) that is/are True. _____

A. A computer with higher IPC will always run faster.
B. Synchronous circuits are usually modeled by FSM.
C. Mealy and Moore machines are not interchangeable.
D. You can obtain the value of PC register using RV32I instructions.

Solution: B, D

[2] (d) (**True or False**) In the classic five-stage pipeline for a RISC-V processor, at most one cache miss may happen to an arithmetic instruction. _____

Solution: T

[2] (e) (**True or False**) The output of a combinational circuit solely depends on the inputs to it.
_____

Solution: T

2 (f) (**True or False**) Given a C program running on two computers with the same instruction set architecture (ISA), say, RISC-V, one computer that yields a higher CPI (clock cycles per instruction) is impossible to be faster than the other one. _____

Solution: F

2 (g) (**True or False**) In the classic five-stage pipeline, read-after-write (RAW) is the only one structural hazard that we need to handle with stall or bypassing. _____

Solution: F

2 (h) (**True or False**) A direct-mapped cache does not need any replacement policy, such as the least-recently used (LRU) or random, to make space. _____

Solution: T

2 (i) (**True or False**) Given two computers with the same instruction set architecture (ISA), say, RISC-V, they may have completely different memory hierarchy systems but the same number of CPU cores. _____

Solution: T

2 (j) (**True or False**) Given two caches with the same size on two computers (the other parts of the computers are the same), the set-associative four-way cache has a lower miss rate than the two-way cache when the computers run the same program. _____

Solution: F

2 (k) (**True or False**) The reason why loading an instruction into the CPU cache hierarchy for one thread causes an eviction of data that the other thread has put through a store instruction, is that the two threads have a data race condition. _____

Solution: F

2 (l) (**True or False**) In a computer with a typical CISC ISA, e.g., x86, an arithemetic instruction is allowed to access memory locations to load or store data. _____

Solution: T

2 (m) (**True or False**) A computer with L1 cache only must be slower than a computer with L1 and L2 caches. _____

Solution: F

3. **Calling Convention [11 Points]**

Your friend Li Hua defines a `struct node` type as follows:

```
struct node {
  int val;       // 4 bytes
  struct node *next; // 4 bytes
};
```

The following function recursively reverses a single-linked list:

```
struct node *reverse(struct node *head, struct node *prev) {
  if (head == NULL) return prev;
  struct node *next_node = head->next;
  head->next = prev;
  return reverse(next_node, head);
}
```

The function is initially called as follows to reverse an entire linked list **head**:

```
struct node *new_head = reverse(head, NULL);
```

7 (a) Please help Li Hua complete the RV32I assembly implementation of the **reverse** function below according to RISC-V calling conventions. Assume:

- Argument **head** is passed through register **a0**.
- Argument **prev** is passed through register **a1**.

```
reverse:
  addi sp, sp, __ // (1) Allocate the minimum stack space
     required regardless of stack alignment requirements.
  sw ______ // (2) Select all must-save registers below
     (multiple choices):
```

(A) **a0**. (B) **a1**. (C) **s0**. (D) **s1**.
(E) **t0**. (F) **ra**. (G) **sp**.

```
  beq a0, x0, reverse_end // if (head == NULL) return prev
  mv s0, a0          // s0 = head
  mv s1, a1          // s1 = prev
  lw t0, _______     // (3) t0 = head->next
  sw s1, _______     // (4) head->next = prev.
  _____________ // (5) Set the 1st argument for recursive call
  _____________ // (6) Set the 2nd argument for recursive call
  jal reverse
```

```
reverse_end:
  mv a0, a1           // Return value: prev
  lw ...              // Restore registers (not shown)
  addi sp sp ...      // Restore stack pointer (not shown)
  ret
```

2 (b) Below is an optimized version of list reversion. Complete the implementation:

```
reverse_opt:
  beq a0, x0, reverse_end_opt // if head == NULL return prev
  lw t0, 4(a0)        // next = head->next
  sw a1, 4(a0)        // head->next = prev
  mv a1, a0           // prev = head
  mv a0, t0           // head = next
  _________     // (7) Write the recursive call instruction
reverse_end_opt:
  mv a0, a1           // Return value: prev
  ret
```

2 (c) Next Li Hua asks you to answer the following question:

If the original non-optimized implementation (**reverse**) processes a linked list of length 6, how many bytes of stack space does the optimized version (**reverse_opt**) save at the deepest recursion point compared to the non-optimized version? Please write your answer directly.

Solution:

- (1) (1 point) **-12** - Allocate 12 bytes of stack space
- (2) (2 points) Correct answers: C, D and F (**ra**, **s0**, **s1**)
  - **ra** must be saved because the recursive call will overwrite it.
  - **s0** and **s1** are callee-saved registers that we use, so they must be preserved.
- (3) (1 point) **4(s0)** (or **4(a0)**) - The **next** field is at offset 4 bytes from the start of the struct.
- (4) (1 point) **4(s0)** (or **4(a0)**) – Same offset for storing to the **next** field.
- (5) (1 point) **mv a0, t0** – First argument should be **next_node** which is in **t0**.
- (6) (1 point) **mv a1, s0** – Second argument should be **head** which is in **s0**.
- (7) (1 point) **j reverse_opt** – Jumping back to the beginning of the function without modifying **ra**. The following answers are also right.

  - `jal x0, reverse_opt`
  - `jal zero, reverse_opt`
  - `tail reverse_opt`

- (8) (1 point) 84 bytes. For a linked list of length 6, the recursion depth is 7 (6 for list nodes, 1 for the last **`NULL`** check). **`reverse`** creates a new stack frame of 12 bytes for each recursive call while **`reverse_opt`** does not reserve any stack space. Thus $12 \times 7 = 84$ bytes are saved.

4. **Scalar and Loop Unrolling (10 Points)**

Next, please help Li Hua examine the execution of the following C loop on a scalar processor. This code operates on two arrays, containing 32-bit floating-point numbers.

```
for (i = 0; i < N; ++i) // N is the the length of two arrays.
   A[i] = A[i] + B[i] * 3.1415926;
```

Let us start by compiling and running the loop on Li Hua's scalar processor. The compiler generates the following instructions.

```
# Intially, f1 = 3.1415926, and N is a large constant number.
# x1 = &A[0] and x2 = &B[0]
# x3 = &A[N] (the 1st address beyond vector A)
I1: flw f0, 0(x2) # load B[i]
I2: flw f2, 0(x1) # load A[i]
I3: fmul f3, f0, f1 # f0 * f1 --> f3
I4: addi x1, x1, 4 # x1 + 4 --> x1
I5: fadd f4, f2, f3 # f2 + f3 --> f4
I6: addi x2, x2, 4 # x2 + 4 --> x2
I7: fsw f4, -4(x1) # store A[i]
I8: bne x1, x3, I1 # if x1 != x3, go to I1
```

5

(a) The code above runs on an in-order, pipelined, single-issue scalar processor with *perfect* branch prediction and *full* bypassing. ALU (integer) operations have a 1-cycle latency (so, thanks to bypassing, consecutive dependent ALU operations execute without stalling), loads have a 2-cycle latency, and floating-point operations have a 3-cycle latency.

(i) How many cycles will the processor stall per loop iteration? Please briefly explain.

Solution: No stalls for the I1→I3 or I2→I5 load-use dependencies, as the compiler has scheduled these instructions far enough apart. I3→I5 and I5→I7 each require one stall to wait for a floating-point operation. Total: 2 stalls

(ii) How many floating-point arithmetic operations per cycle will the processor perform on average? You can assume that N is a very large number, e.g., N $\geq$ 1,000,000,000.

Solution: 8 instructions + 2 stalls = 10 cycles per iteration, so with 2 floating point operations this gives: 2/10 = 1/5 = 0.2 FLOPs/cycle.

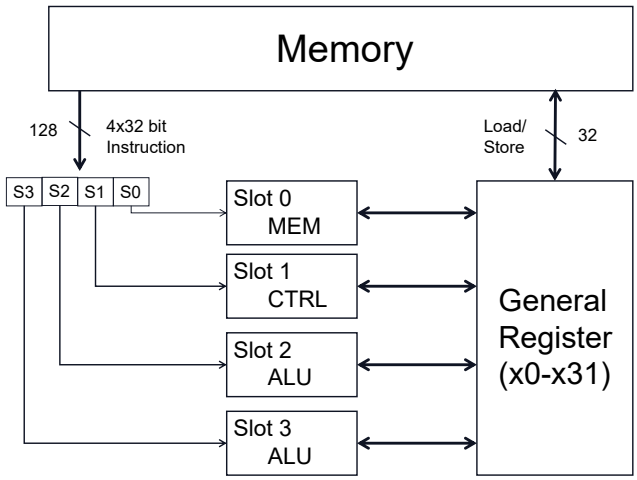
5 (b) Next, Li Hua asks you to apply unrolling to the loop. What is the minimum unrolling factor needed to remove all stalls in a long run of steady-state execution (i.e., N is a very large number)? The unrolling factor is the total number of copies of code that you end up with for the computation in the loop. Explain your answer.

Solution: Unrolling by 2 is sufficient. We could simply interleave instructions from two iterations, which works since the original loop never required two stalls for a single instruction. Although not required, for this particular loop, we can also reduce the bookkeeping instructions and an unrolling factor of 2 is still sufficient. The following code with only two additions (which is the minimum possible) still has zero stalls:

```
1
2 I1: lw f0, 0(x2) ; load B[i]
3 I2: lw f5, 4(x2) ; load B[i+1]
4 I3: lw f2, 0(x1) ; load A[i]
5 I4: lw f6, 0(x1) ; load A[i+1]
6 I5: fmul f3, f0, f1 ; f0 * f1 --> f3
7 I6: fmul f7, f5, f1 ; f5 * f1 --> f7
8 I7: addi x1, x1, 8 ; x1 + 8 --> x1
9 I8: fadd f4, f2, f3 ; f2 + f3 --> f4
```

```
I9: fadd f8, f6, f7 ; f6 + 67 -- > f8
I10: addi x2, x2, 8 ; x2 + 8 --> x2
I11: sw f4, -8(x1) ; store A[i]
I12: sw f8, -4(x1) ; store A[i + 1]
I13: bne x1, x3, I1; if x1 != x3, go to I1
```

5. **Very Long Instruction Word (6 Points)**



6 (a) Li Hua has got a new processor that supports up to four parallel operations. The operations are executed in four parallel pipelined datapath which are referred to as slots. The four slots are named slot 0, slot 1, slot 2, and slot 3. Slot 0 is for the execution of memory access operations. Slot 1 is for the execution of branch operations. Slots 2 and 3 are for the execution of integer arithmetic operations. Operations can be grouped in a *very long instruction word* (VLIW) instruction to be executed in parallel. A VLIW instruction may contain 1, 2, 3, or up to 4 operations. Grouping operations in a VLIW instruction must be explicitly specified in software. For each VLIW instruction: (1) First, all operations in it read their source registers in parallel; (2) After that, all operations in it execute in parallel; (3) Then, all operations in it write their destination registers in parallel.

Each VLIW instruction is atomic from the program's perspective. An instruction has a single PC address that is the address of the start of the instruction. Operations in a VLIW instruction cannot write to the same destination register. Also, there must be no data dependency inside a VLIW instruction. Branches cannot be performed in the middle of an instruction. Architecturally, an instruction executes to completion – including updating all registers and memory where necessary – before the next instruction begins.

```
1  I1:  addi x5, x0, 4
2  I2:  addi x6, x0, 0
3  I3:  lw x10, 0(x11)
4  I4:  lw x28, 0(x12)
5  I5:  lw x29, 0(x13)
6  I6:  mul x28, x28, x29
7  I7:  add x10, x10, x28
8  I8:  addi x12, x12, 4
9  I9:  addi x13, x13, 4
10 I10: addi x6, x6, 1
11 I11: beq x5, x6, I4
12 end: # End of program
```

Suppose Li Hua gives you a RISC-V program shown above. Assume that each operation, i.e., each RISC-V instruction shown in this program, takes 1 cycle to run. If you run it on this VLIW processor, what is the **minimum** number of cycles it will take? _____

Fill your VLIW instructions in the following table with their labels, e.g., I1, to ensure the minimum number of cycles. You're not allowed to change the content of given operations. If no operation can be put in a slot for a VLIW instruction, please put '`NOP`'. You may not use all rows or add more if you think it is necessary. The first row has been filled for your reference.

Instruction No.	Slot 0 (Memory)	Slot 1 (Branch)	Slot 2 (Arithmetic)	Slot 3 (Arithmetic)
1	I3	NOP	I1	I2
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				

Solution:

4 cycles

Instruction No.	Slot 0 (Memory)	Slot 1 (Branch)	Slot 2 (Arithmetic)	Slot 3 (Arithmetic)
1	I4	NOP	I1	I2
2	I5	NOP	I8	I10
3	I3	NOP	I6	I9
4	NOP	I11	I7	NOP

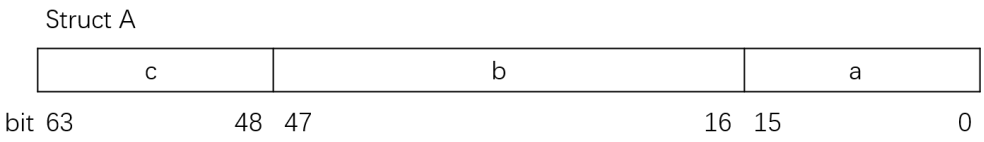
or

5 cycles(See table on next page.)

Instruction No.	Slot 0 (Memory)	Slot 1 (Branch)	Slot 2 (Arithmetic)	Slot 3 (Arithmetic)
1	I3	NOP	I1	I2
2	I4	NOP	I10	NOP
3	I5	NOP	I8	NOP
4	NOP	NOP	I6	I9
5	NOP	I11	I7	NOP
6				
7				
8				

6. **Cache (29 Points)**

Recently Li Hua is very interested in CPU cache. In this question, assume a 32-bit address space. For data types, assume `sizeof(short)== 2` and `sizeof(int)== 4`. Standard C alignment rules apply. The array `arr` starts at memory address `0x10000`. The memory layout of `struct A` is shown in the figure below.



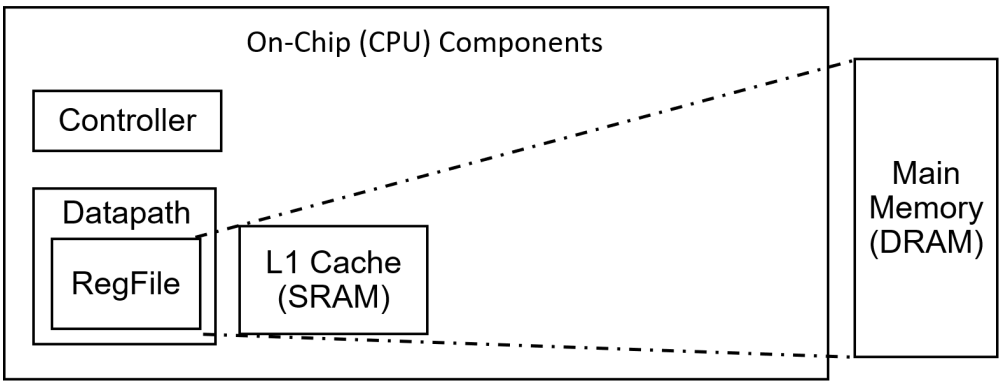
Consider the following C code that Li Hua has written:

```
// Starts at address 0x10000
struct A {short a; int b; short c;} arr[1024];

void f() {
   // i and j are stored in registers and cause no memory access
   register int i, j;

   // Loop X
   for (i = 0; i < 1024; ++i) {
      arr[i].a = i;
      arr[i].b = i;
      arr[i].c = i;
   }

   // Loop Y
   for (i = 0; i < 32; ++i) {
      for (j = 0; j < 60; ++j) {
         // Intermediate calculations for address and values are
            register-based
         // Right-hand side operands are read first
         arr[i + 32 + j * 16].a += arr[i + j * 16].a;
         arr[i + 32 + j * 16].b += arr[i + j * 16].b;
      }
   }
}
```

17 (a) Single-Level Cache Analysis

Li Hua considers a system with a single-level unified cache with the following properties:

- Total Size: 8 KiB (2^{13} bytes)
- Associativity: 4-way Set-Associative
- Block Size: 8 Bytes (2^3 bytes)
- Write Policy: Write-Back



- Write Allocation: Write-Allocate
- Replacement Policy: Least Recently Used (LRU)
- Hit Time (HT): 1 cycle
- Miss Penalty (MP): 80 cycles (time to fetch from main memory)

(i) Cache Anatomy (3 points)

Determine the number of bits for Tag, Index, and Offset for this cache configuration.

Tag: 21 bits, Index: 8 bits, Offset: 3 bits

(ii) Loop X Access Trace (3 points)

Li Hua finds that the cache is initially empty (cold start). Trace all memory accesses (address and type) generated by the **first two iterations (i=0 and i=1)** of Loop X. Please determine if each access causes a Hit or Miss.

Hint: Remember addresses correspond to the start of the accessed data (`short` or `int`). Consider the structure layout, data types, and alignment. `sizeof(struct A)` must be deducted.

Memory Address (Hex)	Tag (Hex)	R/W	Hit/Miss?
10000	20	W	M
10004	20	W	H
10008	20	W	M
1000C	20	W	H
10010	20	W	M
10014	20	W	H

The struct is 4-byte aligned (due to `int b`) and the two `short` members `a` and `c` are padded to 4 bytes. Therefore, the offset of `a`, `b` and `c` are `0x0`, `0x4` and `0x8`, respectively.

(iii) Loop Y Access Trace (6 points)

Continue tracing from the state the cache was left in after the accesses in (a.ii), where Loop X only executed for two iterations before executing Loop Y. Trace all memory accesses generated by the **first iteration of the outer loop (i=0)** of Loop Y, and for **only**

the first iteration of the inner loop (j=0) within each outer loop iteration. Determine if each access is a Hit or Miss.

Memory Address (Hex)	Tag (Hex)	R/W	Hit/Miss?
10000	20	R	H
10180	20	R	M
10180	20	W	H
10004	20	R	H
10184	20	R	H
10184	20	W	H

Reading `0x10000` is a hit because it has been fetched in (a.ii). The access sequence for each in-place addition is reading the RHS operand, then reading the LHS operand, followed by writing the result into the LHS operand.

(iv) Miss Rate Calculation (3 points)

Based *only* on the accesses traced in (a.ii) and (a.iii), calculate the miss rates and round to the nearest integer percentage.

Loop X Miss Rate (Traced): 50 %

Loop Y Miss Rate (Traced): 17 %

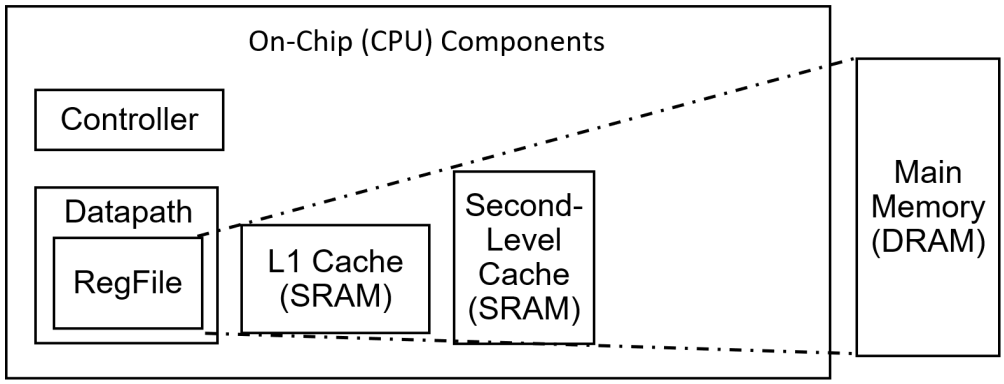
Overall Miss Rate (Traced): 33 %

(v) AMAT Calculation (2 points)

Calculate the Average Memory Access Time (AMAT) for this cache based on the *Overall Miss Rate (Traced)* calculated in (a.iv). Round to one decimal place.

AMAT (based on traced accesses): 27.4 cycles

12 (b) Two-Level Cache Analysis



Now Li Hua considers a system with a two-level unified cache hierarchy with the following properties:

- **L1 Cache:**
  - Size: 1 KiB (2^{10} bytes)

  - Associativity: Direct-Mapped
  - Block Size: 8 Bytes (2^3 bytes)
  - Write Policy: Write-Through
  - Write Allocation: No-Write-Allocate
  - Hit Time (HT1): 1 cycle
- **L2 Cache:**
  - Size: 16 KiB (2^{14} bytes)
  - Associativity: Direct-Mapped
  - Block Size: 16 Bytes (2^4 bytes)
  - Write Policy: Write-Through
  - Write Allocation: No-Write-Allocate
  - Hit Time (HT2): 10 cycles (time to access L2 after L1 miss)
- Main Memory Access Time (MMAT): 120 cycles (time to access main memory after L2 miss)
- Assume L2 is not inclusive of L1 (L1 hit does not guarantee L2 hit). Use local miss rates for calculations.

(i) Cache Anatomy (2 points)

Determine the number of bits for Tag, Index, and Offset for the L1 and L2 caches.

L1: Tag: 22 bits, Index: 7 bits, Offset: 3 bits

L2: Tag: 18 bits, Index: 10 bits, Offset: 4 bits

(ii) Loop X Access Trace (3 points)

Assume both caches are initially empty (cold start). Trace all memory accesses (address and type) generated by the **first two iterations (i=0 and i=1)** of Loop X. Determine if each access is an L1 Hit/Miss and an L2 Hit/Miss. Note that the caches are write-through. Leave L2 status blank if L2 is not accessed.

Memory Address (Hex)	L1 Tag (Hex)	L2 Tag (Hex)	R/W	L1 Hit/Miss?	L2 Hit/Miss?
10000	40	4	W	M	M
10004	40	4	W	M	M
10008	40	4	W	M	M
1000C	40	4	W	M	M
10010	40	4	W	M	M
10014	40	4	W	M	M

Write misses will not cause the destination be cached with no-write-allocate policy. Writes will access both levels of cache with write-through policy.

(iii) Loop Y Access Trace (3 points)

Continue tracing from the state the caches were left in after the accesses in (b.ii), where Loop X only executed for two iterations before executing Loop Y. Trace all memory accesses generated by the **first iteration of the outer loop (i=0)** of Loop Y, and for **only** the **first one iteration of the inner loop (j=0)** within each outer loop iteration. Note that the caches are write-through. Determine L1 Hit/Miss and L2 Hit/Miss status. Leave L2 status blank if L2 is not accessed.

`Memory Address (Hex)`	L1 Tag (Hex)	L2 Tag (Hex)	R/W	L1 Hit/Miss?	L2 Hit/Miss?
`10000`	40	4	R	M	M
`10180`	40	4	R	M	M
`10180`	40	4	W	H	H
`10004`	40		R	H	
`10184`	40		R	H	
`10184`	40	4	W	H	H

Writing `0x10180` and `0x10184` access both levels of cache with write-through policy.

(iv) Miss Rate Calculation (2 points)

Based *only* on the accesses traced in (b.ii) and (b.iii), calculate the L1 miss rates and the L2 *local* miss rate and round to the nearest integer percentage.

L1 Loop X Miss Rate (Traced): 100 %

L1 Loop Y Miss Rate (Traced): 33 %

L1 Overall Miss Rate (Traced): 67 %

L2 Local Miss Rate (Traced): 100 % (L2 Misses / L1 Misses)

(v) AMAT Calculation (2 points)

Calculate the Average Memory Access Time (AMAT) for this two-level cache hierarchy based on the *L1 Overall Miss Rate (Traced)* and *L2 Local Miss Rate (Traced)* calculated in 2.4. Round to one decimal place.

AMAT (based on traced accesses): 88.1 or 131 cycles

Read AMAT based on calculated rates: $1 + 67\% \times (10 + 100\% \times 120) = 88.1$.

Write AMAT is always $1 + 10 + 120 = 131$.

Because the question does not specify which AMAT clearly, both are acceptable.

7. **Performance and Amdahl's Law (10 Points)**

3 (a) Assume that one program written in C runs 10 seconds on machine A. Li Hua has an optimized C compiler that compiles that program into 50% as much instructions as the old compiler. However, half of the instructions require 140% average CPI compared to original program, while the rest requires the same average CPI as the original program. How much time would the program compiled by the newer compiler cost if run on machine A now? Give your calculation.

Solution: $10 \times 50\ \% \times (50\% + 50\% \times 140\%) = 6.0$

4 (b) Li Hua considers an ISA in which instructions can be divided into four different classes (A, B, C, D) according to their CPI. Processor 1 (P1) is with a clock rate of 2.5 GHz and CPIs are 1, 2, 3 and 3 for four classes, respectively. Processor 2 (P2) works at a clock rate of 3 GHz and CPIs are 3, 2, 2 and 2 for four classes, respectively. Given a program that contains 1.0×10^6 instructions with 10% A, 20% B, 50% C, and 20% D.

(i) What is the average CPI of that program for P1 and P2? Give your calculation steps.

Solution: P1: 10% * 1 + 20% * 2 + 50% ×3 + 20% * 3 = 2.6; P2: 10% *3 + 20% * 2 + 50% * 2 + 20% * 2= 2.1.

(ii) Which processor runs faster for that program? Briefly explains your answer.

Solution: 2.6 / 2.5 ¿ 2.1 / 3, so P2 is faster.

3 (c) Li Hua has one more problem. She has a function `f()` that accounts for 80% ($f_E = 0.80$) of the execution time of a large program when run on a processor. The remaining 20% of the program is sequential. Suppose the computation performed by `f()` can be perfectly parallelized, meaning its execution time can be divided by N when run on N parallel

execution units. Please calculate the number of parallel execution units (N) required to achieve an overall program speedup of **4**$\times$. Brielfly explains your answer.

7 8. **Multithreading and OpenMP (7 Points)**

Li Hua has received one more task of counting negative and non-negative numbers in an array **A** that only holds integers. Using a single thread is too slow. She asks you to help her parallelize it with the following program. Assume that each integer takes four bytes (32 bits) and has a memory address starting with the last two bits being '00'.

```
#include <stdio.h>
#include <omp.h>
// size holds the total number of elements in A.
// threads is the number of threads that Li Hua expects.
// Both of them are valid numbers.
void count_num (int *A, int size, int threads) {
      int negatives = 0, non_negatives = 0;
      int i;

      omp_set_num_threads(threads);
      #pragma omp parallel for {
            for (i = 0; i < size; ++i)
                  if (A[i] < 0) negatives += 1;
                  else non_negatives += 1;
      }
      printf("negatives: %d\n", negatives);
      printf("non-negatives: %d\n", non_negatives);
}
```

(a) As Li Hua increases the number of threads running this code, will it certainly print the correct results for negative and non-negative integers? Explain your answer to her.

Solution: No. There may be a data race.

(b) Eventually, Li Hua asks you to correctly count zeros in the array **A**. Please complete the following program by filling in the blank lines. Note that you may not use all blanks or you may need to add blanks if you think it is necessary.

```
#include <stdio.h>
#include <omp.h>
void count_zeros (int *A, int size, int threads)
{
      int zeros = 0;
      int i;

      omp_set_num_threads(threads);

      #pragma omp ____________________________________ {
            for (i = 0; i < size; ++i) {

              ____________________________________

              ____________________________________

              ____________________________________

              ____________________________________









            }
   }
      printf("Zeros: %d\n", zeros);
}
```

Solution:

```
void count_zeros (int *A, int size, int threads)
{
      int zeros = 0;
      int i;

```

```
6         omp_set_num_threads(threads);
7
8     #pragma omp parallel for reduction(+:zeros)
9     for (i = 0; i < size; i++) {
10        if (A[i] == 0) zeros++;
11    }
12        printf("Zeros: %d\n", zeros);
13  }
```

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

REGISTER NAME, USE, CALLING CONVENTION ④

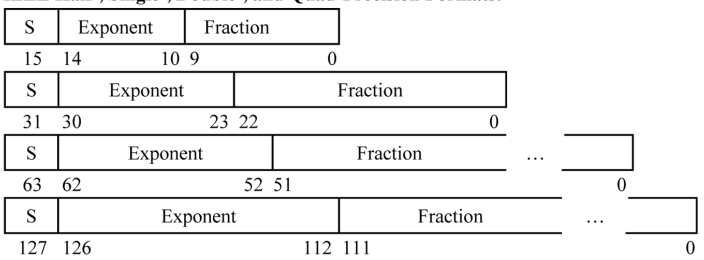
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

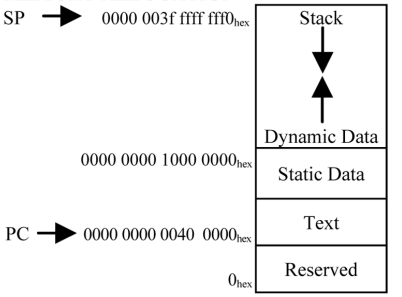
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127,
Double-Precision Bias = 1023, Quad-Precision Bias = 16383

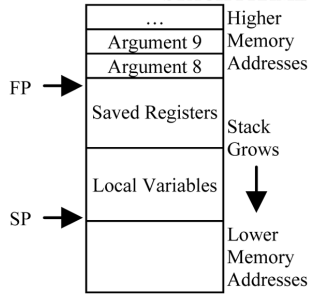
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y