Computer Architecture I Mid-Term II

Chinese Name: _____

Pinyin Name: _____

Student ID: _____

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

Question	Points	Score
1	1	
2	10	
3	16	
4	18	
5	16	
6	24	
7	15	
Total:	100	

- This test contains 27 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 exam time is 120 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 (worth one point): Fill in you name
Fill in your name and email on the front page and your ShanghaiTech email on top of every page (without @shanghaitech.edu.cn) (so write your email in total 27 times).

2. **MISC [10 points]**

[2] (a) (_____) **[Multi-choice]** Select all that apply about cache.

A. A computer must have cache to work.

B. A cache at higher-level (closer to CPU, e.g. L1 cache) usually has a smaller capacity than lower-level cache.

C. A direct-mapped cache allows a single main memory block to be stored in exactly one possible cache block.

D. If a processor has separate L1 instruction and data caches (Harvard architecture), instruction fetches and data accesses can happen simultaneously.

E. Reducing the cache associativity from 8-way to 4-way can decrease the hit time but increase the (conflict) miss rate if the other conditions unchange.

F. None of the above

Solution: B, C, D, E

[1] (b) (_____) **[Single-choice]** A cache uses 2-way set-associative mapping, with a total capacity of 32 bytes, a block size of 8 bytes, and a 12-bit address. What is the bit width of the tag?

A. 6

B. 7

C. 8

D. 9

Solution: C. 8
Offset = $\log_2 8 = 3$; Sets = $(32/8)/2 = 2 \Rightarrow$ Index = 1;
Tag = $12 - 1 - 3 = 8$.

[1] (c) (_____) **[True or False]** A single-core CPU cannot realize thread-level parallelism.

Solution: F

[1] (d) (_____) **[True or False]** Under ideal weak scaling (no communication overhead and load balanced, etc.), a program's runtime remains constant and speedup is linear to the number of processors.

Solution: T

[1] (e) (_____) **[True or False]** In a classic 5-stage IF/ID/EX/MEM/WB pipeline with full forwarding, forwarding can eliminate all data-hazard stalls.

Solution: F

1 (f) (_____) **[True or False]** If two VLIW machines have the same clock frequency, the one with the larger numbers of issuing slot always has the higher realized IPC.

Solution: F

1 (g) (_____) **[True or False]** In a direct-mapped cache, no replacement policy (such as LRU) is needed.

Solution: True. Each block maps to exactly one cache line. On a miss, the new block must overwrite the existing one—no choice exists, so no replacement policy is needed.

1 (h) (_____) **[True or False]** A multi-issue processor is the same as a multi-core processor. Both improve performance by replicating complete CPU cores.

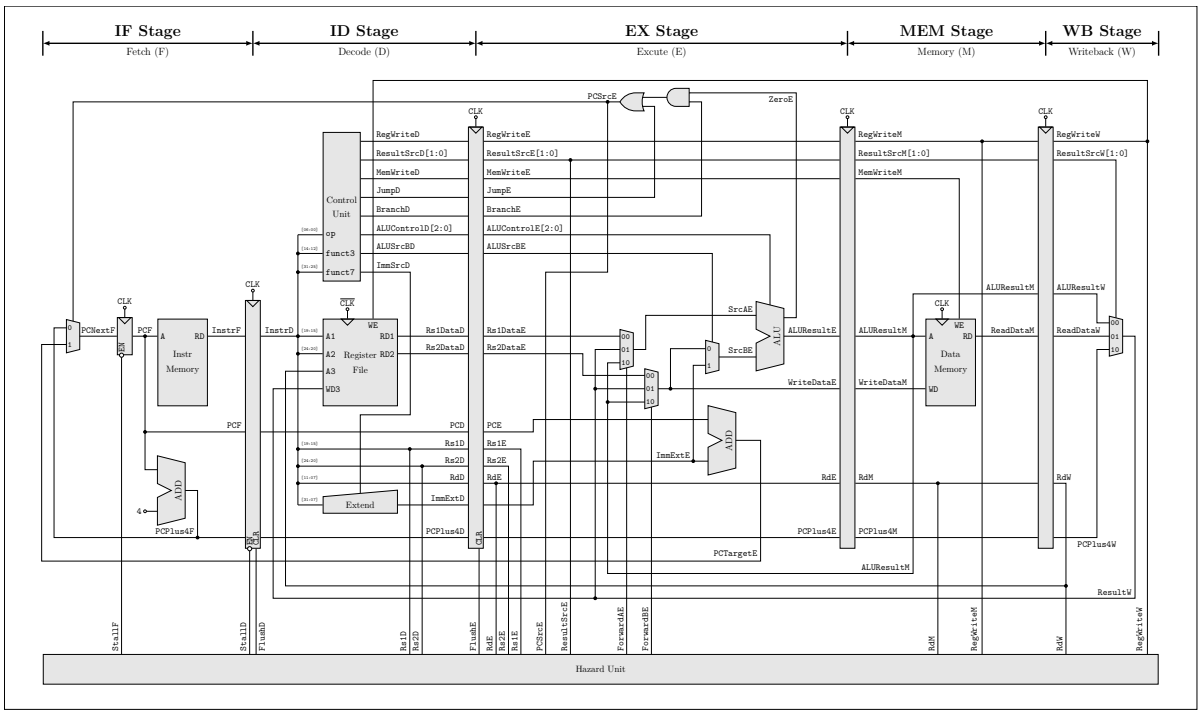
Solution: False. Multi-issue: one core issues multiple instructions/cycle (ILP), duplicates execution units only. Multi-core: multiple complete cores on one chip (TLP). Different parallelism levels and hardware replication.

1 (i) (_____) **[True or False]** LRU cache replacement always produces equal or fewer misses than FIFO replacement for any arbitrary reference string.

Solution: False.

3. **Pipelining: single-cycle comparison and hazards [16 points]**

Consider the five-stage pipelined RISC-V processor shown below. The pipeline stages are IF, ID, EX, MEM, and WB. Assume separate instruction and data memories, full forwarding to the EX stage, and branch/jump decisions made in the EX stage.



4 (a) **Single-cycle CPU vs. pipelined CPU**

The same datapath can be implemented either as a single-cycle CPU or as the five-stage pipelined CPU above. The combinational delay of each stage is:

Stage	IF	ID	EX	MEM	WB
Delay	250 ps	150 ps	200 ps	300 ps	100 ps

Each pipeline register adds 20 ps of overhead (clk-to-q delay and setup time). Ignore hazards for this part and assume a very long program.

(i) What is the clock period of the single-cycle CPU?

(ii) What is the clock period of the pipelined CPU?

(iii) What is the ideal instruction throughput speedup of the pipelined CPU over the single-cycle CPU considering the previous two questions?

(iv) Does pipelining make the latency of one single instruction smaller? Briefly explain.

Solution:

(i) Single-cycle clock period:

$$250 + 150 + 200 + 300 + 100 = 1000 \text{ ps}.$$

(ii) Pipelined clock period:

$$\max(250, 150, 200, 300, 100) + 20 = 320 \text{ ps}.$$

(iii) With ideal CPI $= 1$ for both designs on a very long program, throughput speedup is

$$\frac{1000}{320} = 3.125 \times .$$

(iv) No. A single instruction now passes through five pipeline cycles, so its latency is roughly $5 \times 320 = 1600$ ps. Pipelining improves throughput by overlapping different instructions; it does not necessarily reduce the latency of one instruction.

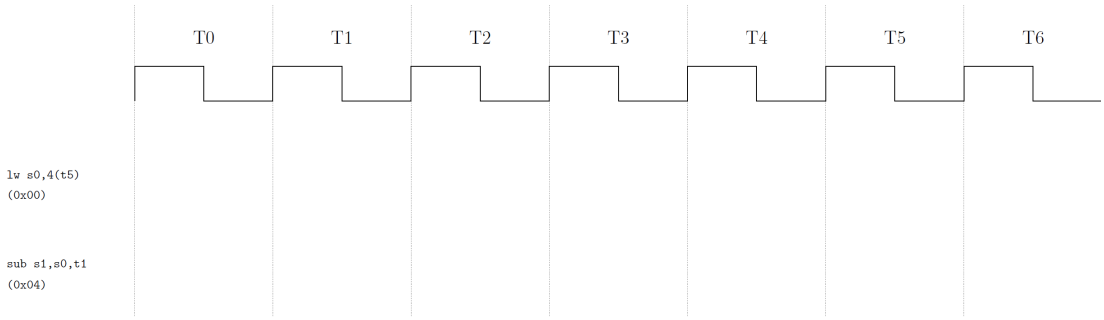
8 (b) **Data hazards: forwarding or stalling?**

For each instruction sequence below, assume the instructions in each case are adjacent unless an extra independent instruction is explicitly shown. In the last column, state whether the RAW data hazard can be solved by forwarding with no stall, or whether a stall is required.

Case	Code	Forwarding/stall decision
A	`add s0,t0,t1` `sub s1,s0,t2`	
B	`lw s0,4(t5)` `sub s1,s0,t1`	

C	`lw s0,0(t0)`
	`add t3,t4,t5`
	`sub s1,s0,t2`	

For Case B only, fill in the timing diagram below with the pipeline stages of the two instructions. Assume IF stage of instruction `lw s0,4(t5)` occurs in T0 period. Justify your forwarding/stall decision according to the diagram.



Solution: The forwarding/stall decisions are:

Case	Hazard	Decision
A	RAW on `s0` from an ALU instruction	Forward ALU result to EX; no stall.
B	RAW load-use hazard on `s0`	One stall is required, then forward the loaded value.
C	RAW on `s0`, but there is one independent instruction between producer and consumer	No stall; forward from MEM/WB to EX, or read the value after it is written back depending on the register-file timing.

For Case B, the timing should be:

Instruction	T0	T1	T2	T3	T4	T5	T6
`lw s0,4(t5)`	IF	ID	EX	MEM	WB		
`sub s1,s0,t1`		IF	ID	stall/ID	EX	MEM	WB

Case B is a RAW load-use hazard on register `s0`. Without a stall, `sub` would need `s0` at the beginning of its EX stage while `lw` is still accessing data memory. The loaded value is not available early enough in that same cycle without lengthening the clock critical path. The hazard unit holds the PC and IF/ID pipeline register for one cycle and inserts a bubble/NOP into ID/EX; in the next cycle, the loaded value can be forwarded to the EX stage of `sub`.

4 (c) **Taken branch control hazard**

Consider the instruction sequence below. The processor predicts the next PC as sequential (branch not taken), and the branch outcome is known in the EX stage.

```
I1: beq s0, s1, L
I2: add t0, t1, t2
I3: sub t3, t4, t5
L: or t6, t7, t8
```

Suppose `I1` is fetched in cycle T0 and the branch is taken.

For each item below, choose exactly one answer.

(i) (_____) **[Single-choice]** In which cycle and in which stage is the branch decision made?

A. T1, ID
B. T2, EX
C. T3, MEM
D. T4, WB

(ii) (_____) **[Single-choice]** At that moment, which wrong-path instructions are in the pipeline?

A. No wrong-path instruction is in the pipeline.
B. `I2` only, in IF.
C. `I2` is in ID and `I3` is in IF.
D. `I2` is in EX and `I3` is in ID.

(iii) (_____) **[Single-choice]** How many cycles of control-hazard penalty are caused by this taken branch, assuming flushing the pipeline takes no extra cycle?

A. 0 cycles
B. 1 cycle
C. 2 cycles
D. 3 cycles

(iv) (_____) **[Single-choice]** What does the hardware do to prevent the wrong-path instructions from changing architectural state?

A. Let the wrong-path instructions finish, but ignore their PC values.
B. Redirect the PC to the branch target and flush the wrong-path instructions by clearing their valid/control bits.
C. Stall the entire pipeline until the branch target instruction reaches WB.
D. Forward the branch comparison result to the PC while leaving younger instructions unchanged.

Solution:

(i) `I1` is in EX in cycle T2, so the branch decision is made in EX during T2.

(ii) `I2` is in ID and `I3` is in IF when the taken branch is resolved.

(iii) The taken branch causes a two-cycle penalty because two younger wrong-path instructions must be discarded.

(iv) The PC is redirected to the branch target, and the F/D and D/E pipeline registers are flushed by turning the wrong-path instructions into NOPs, usually by clearing their control signals/valid bits. This prevents them from writing registers or memory.

4. **Multi-Issue Scheduling in RISC-V [18 points]**

Consider the following RISC-V loop body:

```
1. addi s0,x0,s          # initialize s0
2. L: lw t3,0(t1)        # load array element
3. add t3,t3,s0          # add s to t3
4. sw t3,0(t1)           # store result
5. addi t1,t1,-4         # t1 = t1 - 4
6. bne t1,t2,L           # repeat if t1 ≠ t2
```

Assume a **2-issue pipelined processor** with two issue paths:

- **Slot X**: one **`ALU`** or **`branch`** instruction
- **Slot M**: one **`load`** or **`store`** instruction

The pipeline stages are:

- **ALU/branch path**: **`IF ID EX WB`**
- **load/store path**: **`IF ID EX MEM WB`**

Assume throughout this problem:

- at most one instruction may be issued into each slot per cycle;
- branch prediction is perfect, so the next iteration may begin as early as data dependences allow;
- cache misses and exceptions are ignored;
- in part (c), the forwarding relations available are

`lw(MEM)` $\rightarrow$ **`add(EX)`**, **`add(EX)`** $\rightarrow$ **`sw(MEM)`**, **`addi(EX)`** $\rightarrow$ **`bne(EX)`**.

3 (a) **Instruction classification**

For instructions 1–6, state whether each instruction uses **Slot X** or **Slot M**.

Instr.	Instruction	Slot
1	`addi s0,x0,s`	
2	`lw t3,0(t1)`	
3	`add t3,t3,s0`	
4	`sw t3,0(t1)`	
5	`addi t1,t1,-4`	
6	`bne t1,t2,L`	

Solution:

Instr.	Instruction	Slot
1	`addi s0,x0,s`	X
2	`lw t3,0(t1)`	M
3	`add t3,t3,s0`	X
4	`sw t3,0(t1)`	M
5	`addi t1,t1,-4`	X
6	`bne t1,t2,L`	X

6 (b) **Ideal multi-issue schedule without hazard resolution**

Li Hua proposes to schedule the instructions as they are using a two-issue pipelined processor, without considering the correctness of the program execution or hazard resolution (no forwarding). The scheduled instructions are shown below.

Instruction	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9
1. **`addi s0,x0,s`**	IF	ID	EX	WB					
2. **`lw t3,0(t1)`**	IF	ID	EX	MEM	WB				
3. **`add t3,t3,s0`**		IF	ID	EX	WB				
4. **`sw t3,0(t1)`**		IF	ID	EX	MEM	WB			
5. **`addi t1,t1,-4`**			IF	ID	EX	WB			
`nop`			nop	nop	nop	nop	nop		
6. **`bne t1,t2,L`**				IF	ID	EX	WB		
2. next **`lw t3,0(t1)`**				IF	ID	EX	MEM	WB	
3. next **`add t3,t3,s0`**					IF	ID	EX	WB	
4. next **`sw t3,0(t1)`**					IF	ID	EX	MEM	WB
5. next **`addi t1,t1,-4`**						IF	ID	EX	WB
`nop`						nop	nop	nop	nop

Please identify the hazards that appear in this schedule. You need to clearly state the type of each hazard (RAW, WAW, or WAR) and the instructions involved.

Solution: Hazards:

- **RAW** hazard: instruction 1 → instruction 3 on `s0`;
- **RAW** hazard: instruction 2 → instruction 3 on `t3`;
- **RAW** hazard: instruction 3 → instruction 4 on `t3`;
- **RAW** hazard: instruction 5 → instruction 6 on `t1`;
- **cross-iteration RAW** hazard: instruction 5 → next iteration's instruction 2 on `t1`.
- **WAW** hazard or structural hazard: instruction 2 → instruction 3 on `t1` at cc5;

6 (c) **Schedule with forwarding**

Now resolve the hazards and ensure the correctness of the program using **ONLY** the forwarding relations stated above. Again, construct the **earliest valid schedule** that preserves the original program order.

Instruction	**cc1**	**cc2**	**cc3**	**cc4**	**cc5**	**cc6**	**cc7**	**cc8**	**cc9**
1. **`addi s0,x0,s`**									
2. **`lw t3,0(t1)`**									
`nop`									
`nop`									
3. **`add t3,t3,s0`**									
4. **`sw t3,0(t1)`**									
5. **`addi t1,t1,-4`**									
`nop`									
6. **`bne t1,t2,L`**									
2. next **`lw t3,0(t1)`**									

Then state which forwarding relations are used, using the stage-level notation above.

Solution: The unique earliest schedule with the stated forwarding relations is:

Instruction	**cc1**	**cc2**	**cc3**	**cc4**	**cc5**	**cc6**	**cc7**	**cc8**	**cc9**
1. `addi s0,x0,s`	IF	ID	EX	WB					
2. `lw t3,0(t1)`	IF	ID	EX	MEM	WB				
`nop`		nop	nop	nop	nop				
`nop`		nop	nop	nop	nop	nop			
3. `add t3,t3,s0`			IF	ID	EX	WB			
4. `sw t3,0(t1)`			IF	ID	EX	MEM	WB		
5. `addi t1,t1,-4`				IF	ID	EX	WB		
`nop`				nop	nop	nop	nop	nop	
6. `bne t1,t2,L`					IF	ID	EX	WB	
2. next `lw t3,0(t1)`					IF	ID	EX	MEM	WB

The forwarding relations used are:

- `lw(MEM)` → `add(EX)` from instruction 2 to instruction 3;
- `add(EX)` → `sw(MEM)` from instruction 3 to instruction 4;
- `addi(EX)` → `bne(EX)` from instruction 5 to instruction 6;
- `addi(MEM)` → `lw(EX)` from instruction 5 to next-iteration intstruction 2; otherwise, delay `bne` and next-iteration `lw` by 1 clock cycle.

3 (d) **Loop unrolling and multi-issue scheduling**

Now consider the following unrolled code, which processes four array elements in one loop iteration. Assume the scheduler may re-order independent instructions from this unrolled body as long as all data dependences and the meaning of the loop are preserved.

```
1. addi s0,x0,s
2. L: lw t3,0(t1)
3. lw t4,-4(t1)
4. lw t5,-8(t1)
5. lw t6,-12(t1)
6. add t3,t3,s0
7. add t4,t4,s0
8. add t5,t5,s0
9. add t6,t6,s0
10. sw t3,0(t1)
11. sw t4,-4(t1)
12. sw t5,-8(t1)
13. sw t6,-12(t1)
14. addi t1,t1,-16
15. bne t1,t2,L
```

For each item below, choose exactly one answer.

(i) (_____) **[Single-choice]** What is the main reason loop unrolling can improve the schedule on this 2-issue processor?

A. It changes every memory instruction into an ALU instruction.

B. It exposes independent work from several original iterations, giving the scheduler more instructions that can be paired across the two slots.

C. It removes all true data dependences in the loop.

D. It makes the branch instruction execute in **Slot M**.

(ii) (_____) **[Single-choice]** If one `lw` is waiting before its dependent `add` can use the loaded value, what can the scheduler do more easily after loop unrolling?

A. Issue independent instructions from other unrolled elements while waiting for the dependent `add`.

B. Issue the dependent `add` before the `lw` completes.

C. Convert the RAW data hazard into a control hazard.

D. Issue two `add` instructions in **Slot X** during the same cycle.

(iii) (_____) **[Single-choice]** After loop unrolling, which same-cycle pairing becomes easier for the scheduler to find?

A. A `lw` and the dependent `add` for the same array element.

B. An `add` for one array element in Slot X together with a `lw` or `sw` for a different array element in Slot M.

C. Two `lw` instructions, one in each issue slot.

D. A `bne` in **Slot M** together with an `add` in **Slot X**.

Solution:

(i) Loop unrolling exposes independent work from several original iterations, giving the scheduler more instructions that can be paired across the two slots.

(ii) The scheduler can use independent instructions from other unrolled elements while waiting for a load-dependent `add`.

(iii) The scheduler can more easily pair an `add` for one element in **Slot X** with a `lw` or `sw` for another element in **Slot M**.

5. **Scalar, Loop Unrolling, and SIMD [16 points]**
Xiao Ming is optimizing the following dot-product loop:

```
float sum = 0.0f;
for (int i = 0; i < N; i++) {
   sum += A[i] * B[i];
}
```

The two arrays `A` and `B` both consist of 32-bit floating-point numbers. Assume that **`N`** is very large and is a multiple of 8, so no cleanup/remainder loop is needed in any part of this question. The compiler is allowed to reassociate floating-point additions, so small differences due to floating-point rounding can be ignored. At loop entry, the relevant registers contain:

```
# x1 = &A[0]
# x2 = &B[0]
# x3 = &A[N]
# f8 = 0.0    # sum
```

The processor is an in-order, pipelined, single-issue processor with **perfect branch prediction**. All functional units are fully pipelined, so when there is no stall the processor can issue one new instruction every cycle. Integer ALU operations have a 1-cycle latency, loads have a 2-cycle latency, and floating-point arithmetic operations have a 3-cycle latency.

Use the following timing convention throughout this question: if an instruction issued in cycle t has latency L, then a dependent instruction may issue in cycle $t + L$. If a dependent instruction would issue earlier, the in-order pipeline stalls until the value is available. Independent instructions may issue in the intervening cycles.

4 (a) Xiao Ming first writes the following scalar assembly code directly from the C loop:

```
L1:
   flw   f0, 0(x1)     # load A[i] to register f0
   flw   f2, 0(x2)     # load B[i] to register f2
   fmul.s f4, f0, f2   # A[i] * B[i], result to register f4
   fadd.s f8, f8, f4   # sum += A[i] * B[i]
   addi  x1, x1, 4     # advance A pointer
   addi  x2, x2, 4     # advance B pointer
   bne   x1, x3, L1
```

What is the steady-state cycles per iteration? How many stall cycles are inserted per loop iteration? What is the steady-state throughput in FLOPs/cycle (FLOP=Floating-point Operations, each floating-point addition or multiplication is considered as 1 FLOP)?

Solution: There are two sources of stalls.

First, there is one load-use stall between:

```
flw   f2, 0(x2)
fmul.s f4, f0, f2
```

The value loaded into `f2` is used immediately by `fmul.s`. Since loads have a 2-cycle latency, one stall cycle is needed.

Second, there are two stall cycles between:

```
fmul.s f4, f0, f2
fadd.s f8, f8, f4
```

The result of `fmul.s` is used immediately by `fadd.s`. Since floating-point arithmetic operations have a 3-cycle latency, two stall cycles are needed when there are no independent instructions between the producer and the consumer.

The 3-cycle floating-point latency is therefore reflected in the required spacing before the dependent `fadd.s`; because the functional unit is fully pipelined, it does not consume three separate issue slots for the `fmul.s` instruction itself.

Thus, each loop iteration has:

$$1 + 2 = 3 \text{ stall cycles.}$$

With single issue and fully pipelined functional units, the steady-state initiation interval contains one issue slot for each of the 7 loop-body instructions plus the 3 stall slots:

$$7 \text{ issued instructions} + 3 \text{ stall cycles} = 10 \text{ cycles/iteration.}$$

Each iteration performs two floating-point arithmetic operations:

$$1 \text{ multiply} + 1 \text{ add} = 2 \text{ FLOPs.}$$

Therefore, Xiao Ming's scalar code has throughput:

$$\frac{2}{10} = 0.2 \text{ FLOPs/cycle.}$$

7 (b) Xiao Ming tries to improve the scalar loop by unrolling it by a factor of 2. He first writes the following code:

```
L1:
   flw   f0, 0(x1)    # load A[i]
   flw   f2, 0(x2)    # load B[i]
   fmul.s f4, f0, f2  # A[i] * B[i]
   fadd.s f8, f8, f4  # sum += A[i] * B[i]
```

```
flw   f10, 4(x1)   # load A[i+1]
flw   f12, 4(x2)   # load B[i+1]
fmul.s f14, f10, f12 # A[i+1] * B[i+1]
fadd.s f8, f8, f14 # sum += A[i+1] * B[i+1]

addi  x1, x1, 8    # advance A pointer by 2 floats
addi  x2, x2, 8    # advance B pointer by 2 floats
bne   x1, x3, L1
```

Xiao Ming claims that this loop unrolling removes the stalls because the loop now does twice as much work per iteration, but the above code does not actually achieve that goal. Rewrite the loop body so that the steady-state loop has as few stalls as possible while still using an unrolling factor of 2. **Hint:** You may reorder instructions but do not induce additional instructions. You can use additional floating-point registers and combine two partial sums after the loop.

```
L1:
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   ____________________
   bne   x1, x3, L1
# code after the loop:
   ____________________
```

Solution: Xiao Ming's first unrolled code does not effectively reduce stalls. A better version should do two things:

- schedule independent loads and multiplies to separate dependent instructions;
- use two independent accumulators instead of updating `f8` twice.

One valid scheduled version is:

```
# f8 = partial sum 0, initialized to 0 before the loop
# f16 = partial sum 1, initialized to 0 before the loop

L1:
   flw   f0, 0(x1)    # load A[i]
```

```
    flw   f2, 0(x2)    # load B[i]
    flw   f10, 4(x1)   # load A[i+1]
    flw   f12, 4(x2)   # load B[i+1]

    fmul.s f4, f0, f2  # A[i] * B[i]
    fmul.s f14, f10, f12 # A[i+1] * B[i+1]

    addi  x1, x1, 8    # advance A pointer by 2 floats
    addi  x2, x2, 8    # advance B pointer by 2 floats

    fadd.s f8, f8, f4  # partial sum 0
    fadd.s f16, f16, f14 # partial sum 1

    bne   x1, x3, L1

# after the loop:
    fadd.s f8, f8, f16
```

This schedule avoids the load-use stalls by placing the loads for both iterations before the multiplies.

It also avoids the back-to-back dependency on the same accumulator. Xiao Ming's original unrolled code updates `f8` twice:

```
    fadd.s f8, f8, f4
    ...
    fadd.s f8, f8, f14
```

In the improved version, the two additions update different accumulators:

```
    fadd.s f8, f8, f4
    fadd.s f16, f16, f14
```

Thus, the two additions are independent.

The scheduled loop body has 11 issued instructions and 0 stall cycles under the assumptions in the problem, so its steady-state initiation interval is 11 cycles. It processes 2 elements per iteration and performs:

$$2 \times (1 \text{ multiply} + 1 \text{ add}) = 4 \text{ FLOPs}.$$

Therefore, the throughput is:

$$\frac{4}{11} \approx 0.364 \text{ FLOPs/cycle}.$$

Other instruction schedules are also acceptable if they correctly preserve the computation, use independent accumulators, and avoid unnecessary stalls.

5 (c) Now assume that the same single-issue processor also has a simple 128-bit SIMD extension. Each vector register **vx** holds four 32-bit floating-point numbers. A vector instruction still occupies one issue slot, but a vector arithmetic instruction performs one arithmetic operation in each of the four lanes. The following SIMD instructions are available:

```
vflw     v0, 0(x1)    # load 4 floats
vfmul.vv v2, v0, v1   # 4 parallel FP multiplies
vfadd.vv v4, v4, v2   # 4 parallel FP adds
vredsum  f8, v4       # horizontal reduction of vector elements
                      # into scalar register f8
```

Vector loads have the same latency as scalar loads, and vector arithmetic instructions have the same latency as scalar floating-point arithmetic instructions. As above, ignore the one-time cost of the final horizontal reduction when computing steady-state loop throughput.

The compiler generates the following SIMD loop:

```
# v8 = [0.0, 0.0, 0.0, 0.0] # vector accumulator

VL:
   vflw     v0, 0(x1)    # load A[i : i+3]
   vflw     v2, 0(x2)    # load B[i : i+3]

   vfmul.vv v4, v0, v2   # 4 multiplications

   addi     x1, x1, 16   # advance A pointer by 4 floats
   addi     x2, x2, 16   # advance B pointer by 4 floats

   vfadd.vv v8, v8, v4   # 4 additions into vector accumulator

   bne      x1, x3, VL

# after the loop:
# vredsum f8, v8
```

(i) How many scalar array elements does each SIMD loop iteration process?

(ii) How many FLOPs does each SIMD loop iteration perform?

(iii) What is the steady-state cycles per SIMD loop iteration (initiation interval)?

(iv) What is the speedup over the scalar code in part (a) if considering the throughput of FLOPs/cycle?

Solution: Each SIMD iteration processes 4 scalar elements.
For each element, the computation performs:

$$1 \text{ multiply} + 1 \text{ add} = 2 \text{ FLOPs}.$$

Therefore, each SIMD iteration performs:

$$4 \times 2 = 8 \text{ FLOPs}.$$

There is one load-use stall between:

```
vflw     v2, 0(x2)
vfmul.vv v4, v0, v2
```

The dependency from `vfmul.vv` to `vfadd.vv` does not stall because the two `addi` instructions provide enough separation.
Thus, the SIMD loop has steady-state initiation interval:

$$7 \text{ issued instructions} + 1 \text{ stall cycle} = 8 \text{ cycles/iteration}.$$

The SIMD throughput is:

$$\frac{8 \text{ FLOPs}}{8 \text{ cycles}} = 1 \text{ FLOP/cycle}.$$

From part (a), Xiao Ming's scalar code had throughput:

$$0.2 \text{ FLOPs/cycle}.$$

Therefore, the speedup over Xiao Ming's scalar code from part (a) is:

$$\frac{1}{0.2} = 5 \times .$$

This speedup is larger than the SIMD width of 4 because the scalar baseline in part (a) was poorly scheduled and contained unnecessary stalls.

6. **Cache [24 points]**

Li Hua is reverse-engineering the L1 Data Cache of a newly released 32-bit processor. He reads from the manual that the cache has a **Total Data Capacity of 2 KiB (2048 Bytes)** and strictly uses a **Write-Back** and **Write-Allocate** policy with an **LRU (Least Recently Used)** replacement strategy. However, the manual is torn, and the Block Size and Associativity are entirely unknown.

Part A: Reverse Engineering (8 points)

To uncover the missing cache parameters, Li Hua writes a test script that accesses specific memory addresses and records whether each access is a Hit or a Miss. He ensures the cache is completely empty (Cold Start) before running the trace.

Assume all addresses below are strictly **32-bit physical addresses**, and all accesses are **4-byte (`int`) reads**.

Op #	Memory Address (Hex)	Result
1	`0x00010000`	Miss
2	`0x0001000C`	Hit
3	`0x00010010`	Miss
4	`0x00010000`	Hit
5	`0x00010800`	Miss
6	`0x00010000`	Hit
7	`0x00011000`	Miss
8	`0x00010800`	Miss

Based **ONLY** on the deterministic logic of the trace above and the known 2 KiB capacity, answer the following questions.

3 (a) What is the exact **Block Size** (or Cacheline Size) of this cache (in Bytes)? Briefly explain which operations prove this.

Solution: 16 Bytes.
Proof: Op 1 Miss loads the entire block. Op 2 reading `0x0001000C` is a Hit, indicating the block covers offsets 0-15. Op 3 reading `0x00010010` is a Miss, meaning it exceeded the block boundary. Thus, the block size is strictly 16 Bytes.

3 (b) What is the exact **Associativity** (e.g., Direct-Mapped, 2-Way, 4-Way, etc.)? Provide a rigorous logical proof using the operations and address bits.

Solution: 2-Way Set-Associative.
Proof: Given 2KB capacity and 16B blocks, the Index is at most 7 bits. The test addresses `0x00010000`, `0x00010800`, and `0x00011000` are all 0 in bits 4 to 10, meaning they strictly map to the same Cache Index (Set 0). Op 6 is a Hit, proving the first two blocks can co-exist (not 1-Way), which also makes `0x00010800` the

LRU. Op 8 reading **0x00010800** is a Miss, proving the arrival of the 3rd block (Op 7) evicted the oldest block via LRU. Thus, the set can hold exactly 2 blocks.

2 (c) Specify the exact number of bits allocated for the **Tag**, **Index**, and **Offset**.

Solution: Offset = **4** bits (0-3), Index = **6** bits (4-9), Tag = **22** bits (10-31).

Part B: Macro-Scale Conflict Analysis (8 points)

Li Hua now executes the following C code snippet. Assume Arrays **A** and **B** are contiguous in memory, respectively. Array **A** starts exactly at physical address **0x00020000**, and Array **B** starts exactly at **0x00120000**. **sizeof(int) == 4**.

```
int A[512][512];
int B[512][512];

for (int i = 0; i < 512; i++) {
   for (int j = 0; j < 512; j++) {
      // Interleaved memory access
      A[i][j] = B[j][i];
   }
}
```

2 (d) In the inner loop (as **j** increments), what is the memory address stride (the difference in bytes) between accessing **B[0][0]** and **B[1][0]**? Do **B[0][0]** and **B[1][0]** map to the same Cache Index?

Solution: Stride = 512×4 = **2048 Bytes** (**0x0800**).
Yes, they map to the same Index. Adding 2048 only changes bit 11 (Tag), leaving the Index bits (4-9) entirely unchanged (all zeros).

2 (e) According to the 3C (Compulsory/Conflict/Capacity) model, what specific type of Cache Miss heavily dominates the accesses to Array **B**? Briefly explain why.

Solution: Conflict Miss.
Reason: The 2048-byte stride forces all elements accessed in the inner loop to map to the exact same Cache Index. Since the set can only hold 2 blocks, they continuously evict each other (thrashing) long before the total cache capacity is exhausted.

3 (f) Consider the interleaved access pattern of Array **A** and Array **B**. Briefly trace the LRU state of the relevant Cache Set for the first 3 iterations of the inner loop (**j = 0, 1, 2**). Is the A/B read/write access a cache hit or miss? Which cacheline (Way 0, Way 1, ...) is being filled? Which cacheline is LRU? Then Determine the actual Cache Miss Rate for Array **A** during the execution of this code.

Solution: Actual Miss Rate: **25%**.
Proof (LRU Shielding):
`j=0`: Read B (Miss, Way 0); Write A (Miss, Way 1). Way 0(B) becomes LRU.
`j=1`: Read B (Miss). New B evicts old B in Way 0. Way 1(A) becomes LRU. Write A (**Hit** in Way 1). A restores to MRU, Way 0(B) becomes LRU again.
`j=2`: Read B evicts old B in Way 0 again. Write A **Hits** in Way 1 again.
Conclusion: B continuously thrashes in Way 0, leaving Array A perfectly protected in Way 1.

Part C: Micro-State & Memory Traffic (6 points)

Li Hua resets the cache (Cold Start) and runs a new micro-benchmark focusing strictly on **Set 1** (Index = 1). The cache still uses Write-Back, Write-Allocate, and LRU.

He executes the following **6 operations** sequentially.

1. `Read 0x00100010`
2. `Write 0x00200014`
3. `Read 0x00100018`
4. `Write 0x0030001C`
5. `Write 0x00100010`
6. `Read 0x00400010`

6 (g) Fill in the final state of **Set 1** after all 6 operations are completed. (Leave unused ways blank. Assume Valid and Dirty bits are initially 0).

Way	Tag (Hex)	Valid (1/0)	Dirty (1/0)
0	_____	_____	_____
1	_____	_____	_____

Solution:

Way	Tag (Hex)	Valid (1/0)	Dirty (1/0)
0	`0400`	1	1
1	`1000`	1	0

(Note: Tags like `0x0400` or `400` are completely acceptable)

3 (h) During the execution of these 6 operations, how many total Bytes were **read from** Main Memory, and how many total Bytes were **written back to** Main Memory? Show brief calculation steps.

Solution: Read from Main Memory: 64 Bytes.
Op 1, 2, 4, 6 caused 4 Cache Misses. Under Write-Allocate, both Read and Write misses load a 16B block. $4 \times 16 = 64$ Bytes.
Written back to Main Memory: 32 Bytes.

Op 4 evicted dirty block **`0x0020...`** (16B). Op 6 evicted dirty block **`0x0030...`** (16B). 2 Write-backs $\times 16$ = 32 Bytes.

7. **Multithreading and OpenMP [15 points]**

In this problem, you may use the following OpenMP directives and clauses.

(A) `#pragma omp task shared(var)`
Creates an OpenMP task that may execute asynchronously on an available thread. The clause `shared(var)` means that the variable `var` is shared between the parent task and the child task. Therefore, the task accesses the same memory location instead of a private copy.

(B) `#pragma omp task depend(out: var)`
Declares that the task writes to `var`. Any later task with `depend(in: var)` must wait until this task finishes.

(C) `#pragma omp task depend(in: var)`
Declares that the task reads `var`. The OpenMP runtime guarantees that all prior tasks with `depend(out: var)` are completed before this task begins.

(D) `#pragma omp task depend(in: a,b)depend(out: c)`
A task may contain multiple dependency clauses. This example means that the task waits for both `a` and `b` to become available, then produces `c`. The runtime system automatically constructs a task dependency graph (DAG).

(E) `#pragma omp taskwait`
Makes the current thread wait here until all tasks it has created so far are finished.

(F) `#pragma omp parallel`
Creates a team of threads that execute the following block of code in parallel.

(G) `#pragma omp single`
Ensures that only one thread in the team executes the following block, while the others wait at the end.

(a) **Basic OpenMP Task Usage [8 points]**

8

i. Complete the missing OpenMP directives in the code below. Use each directive in the box exactly once. The task for `x` is completed as an example.

A. `#pragma omp task shared(y)`	**B.** `#pragma omp taskwait`
C. `#pragma omp parallel`	**D.** `#pragma omp single`

```
int fib(int n) {
   if (n < 2) return n;
   int x, y;

   /* example */
   #pragma omp task shared(x)
   x = fib(n - 1);

   _____________________________________

   y = fib(n - 2);
```

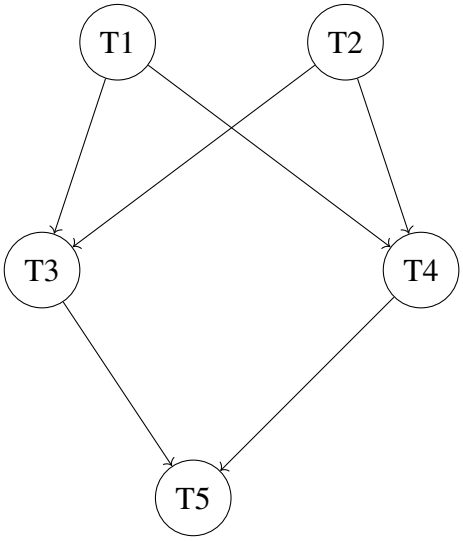
```
    _____________________________________
    return x + y;
}

int main() {
    int n = 40, result;

    _____________________________________
    {
        _____________________________________
        result = fib(n);
    }
}
```

Solution:

```
#pragma omp task shared(y)

#pragma omp taskwait

#pragma omp parallel

#pragma omp single
```

(b) **Task DAG and the `depend` Clause [7 points]**

Li Hua now wants to parallelize the following computation DAG:



The dependency relationships are:

- T1 and T2 have no dependencies and may execute in parallel.
- T3 depends on both T1 and T2.
- T4 depends on both T1 and T2.

- T5 depends on both T3 and T4.

Li Hua uses OpenMP tasks together with the `depend` clause. **Variables `a`, `b`, `c`, and `d` store the outputs of T1–T4 respectively, while `result` stores the output of T5.**

4 i. Complete the remaining blanks with the appropriate `depend` clauses. The first task is completed as an example. Use **`a, b, c, d, result`** according to your need.

```
#pragma omp parallel
{
   #pragma omp single
   {
      int a, b, c, d, result;

      /* T1: foo(1), example */
      #pragma omp task _____________________________________
      a = foo(1);

      /* T2: foo(2) */
      #pragma omp task _____________________________________
      b = foo(2);

      /* T3: bar(a, b) */
      #pragma omp task depend(in: a,b) depend(out: c)
      c = bar(a, b);

      /* T4: baz(a, b) */
      #pragma omp task _____________________________________
      d = baz(a, b);

      /* T5: qux(c, d) */
      #pragma omp task _____________________________________
      result = qux(c, d);
   }
}
```

Solution:

```
#pragma omp task depend(out: a)

#pragma omp task depend(out: b)

#pragma omp task depend(in: a,b) depend(out: c)

#pragma omp task depend(in: a,b) depend(out: d)

#pragma omp task depend(in: c,d) depend(out: result)
```

3

ii. What is the relationship between T3 and T4 (serial, parallel, or mutually exclusive)? Does the OpenMP runtime guarantee that T3 must execute before T4? Explain your answer.

Solution: T3 and T4 may execute in parallel. Both depend on T1 and T2, but there is no dependency edge between T3 and T4 themselves. Therefore, once T1 and T2 have finished, the OpenMP runtime may schedule T3 and T4 in either order or at the same time; it does not guarantee that T3 executes before T4.