1 Flynn’s Taxonomy

1.1 Explain SISD and give an example if available.

Single instruction/single data stream. Each instructions are executed in order, operating single streams of data.
For example, Intel Pentium 4.

1.2 Explain SIMD and give an example if available.

Single instruction multiple data stream.
It uses one controller to control multiple processors, simultaneously operate each data in a data vector.
For example, x86 SSE instruction.

1.3 Explain MISD and give an example if available.

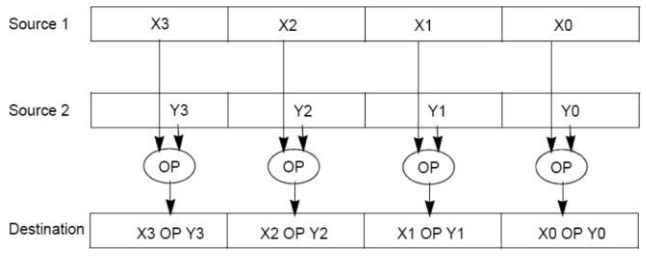
Multiple instruction/single data stream Multiple instructions are executed simultaneously, operating single streams of data.
It doesn’t have any instance.

1.4 Explain MIMD and give an example if available.

Multiple instruction/multiple data stream. Multiple instructions are executed simultaneously, operating multiple streams of data.
For example, Intel Core i7.

2 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type_m128i is used when these registers hold 4 ints, 8 shorts or 16 chars;_m128d is used for 2 double precision floats, and_m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- __m128i _mm_set1_epi32(**int** i):
  Set the four signed 32-bit integers within the vector to i.
- __m128i _mm_loadu_si128(m128i *p):
  Load the 4 successive ints pointed to by p into a 128-bit vector.
- __m128i _mm_mullo_epi32(m128i a, m128i b): Return vector ($a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3$).
- __m128i _mm_add_epi32(m128i a, m128i b): Return vector ($a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3$)
- **void** _mm_storeu_si128(m128i *p, m128i a): Store 128-bit vector a at pointer p.
- __m128i _mm_and_si128(m128i a, m128i b):
  Perform a bitwise按位的 AND of 128 bits in a and b, and return the result.
- __m128i _mm_cmpeq_epi32(m128i a, m128i b):
  The ith element of the return vector will be set to 0xFFFFFFFF(这里好像描述和八个f不同?描述是元素被置为1八个f好像是vector都为1) if the ith第i个 elements of a and b are equal, otherwise it'll be set to 0.

Notice: On this worksheet, we are using the *unaligned* versions of the commands that interface with memory (i.e. storeu/loadu vs. store/load). This is because the store/load commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. 不一定对齐 For instance, _mm_store_si128 needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

2.1 You have an array of 32-bit integers and a 128-bit vector as follows:

```
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
m128i vector = _mm_loadu_si128((_m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply vector by itself, and set vector to the result.

```
vector = _mm_mullo_epi32    ( vector    ,vector     );
```

(b) Add 1 to each of the first 4 elements of the arr, resulting in arr = {2, 3,4, 5, 5, 6, 7, 8}

```
m128i vector_ones = _mm_set1_epi32( 1   );
m128i result = _mm_add_epi32(vector , vector_ones   );
_mm_storeu_si128(( m128i *) arr , result    );
```

(c) Add the second half of the array to the first half of the array, resulting in arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}

```
 m128i result = _mm_add_epi32(_mm_loadu_si128(( m128i *) (arr+4)    ), vector
);//_就是把 arr 的后半部分读出来
_mm_storeu_si128(( m128i *) arr , result    );
```

(d) Set every element of the array that is not equal to 5 to 0, resulting in arr = {0, 0, 0, 0, 5, 0, 0, 0}. Remember that the first half of the array(1 2 3 4) has already been loaded into vector.

```
m128i fives =   _mm_set1_epi32( 5   ); //set [5 5 5 5 ]
m128i mask =    _mm_cmpeq_epi32( vecotr , fives );
m128i result =  _mm_and_si128(vector , mask  );
_mm_storeu_si128( ( m128i *) arr     , result    );
vector = _mm_loadu_si128(  ( m128i *) (arr+4)   );
mask = _mm_cmpeq_epi32( vecotr  , fives );
result =         _mm_and_si128(vector , mask  );
_mm_storeu_si128( ( m128i *) (arr+4)    , result    );
```

2.2 SIMD-ize the following function, which returns the product of all of the elements in an array. Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration?另外开一个然后转移过去 What if our array has a length that isn't a multiple of 4? Can we always SIMD-ize an entire array? No . What can we do to handle this tail case?

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) { product *= a[i];
    }
    return product;
}
static int product_vectorized(int n, int *a) {
    int result[4];
    m128i prod_v =  _mm_set1_epi32( 1   );
    for (int i = 0; i < n/4    ; i += 1 ) { // Vectorized loop
```

```
        prod_v = _mm_mullo_epi32( _mm_loadu_si128(( m128i *) (a+4*i)    ), prod_v  )
    }// 怎么并发让每个数都乘起来?
    mm_storeu_si128( (__m128i *)result , prod_V    );
    for (int i = n/4*4  ; i < n     ; i++) {
        result[0] *= a[i]
        // Handle tail case result[0] *=  ;
    }
    return  result[0]* result[1]*result[2]* result[3];
}
```

3 Thread-Level Parallelism

As powerful as data level parallelization is, it can be quite inflexible不可弯曲的呆板的坚定的, as not all applications have data that can be vectorized不是都可以向量化的. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile通用的.

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

a. The parallel directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

#pragma omp parallel
{
...
}

NOTE: The opening curly brace needs to be on a newline or **else** there will be a compile-time error!

b. The parallel **for** directive指令 will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The following two code snippets are equivalent.

#pragma omp parallel **for**
for (**int** i = 0; i < n; i++) {
...
}

#pragma omp parallel
{
#pragma omp **for**
for (**int** i =0; i < n; i++) { ... }
}

There are two functions you can call that may be useful to you:

c. **int** omp_get_thread_num() will return the number of the thread executing the code

d. **int**omp_get_num_threads()will return the number of total hardware threads executing the code

3.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an **int**[] of length n.

```
(a) // Set element i of arr to i
#pragma omp parallel
{
for (int i = 0; i < n; i++) arr[i] = i;
}
```

slower than serial, The parallel **for** directive指令 will split up iterations of a for loop over various threads. But there is no **for** directive so that each thread would begin at I = 0, the traversal time is as same as serial execution. But the various threads require more time overhead.

```
(b) // Set arr to be an array of Fibonacci numbers. arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
arr[i] = arr[i-1] + arr[i - 2];
```

always incorrect,
arr[3] = arr[2] +arr[1] will fetch arr[2] before (arr[2] = arr[1] +arr[0]) is completed. Because we assume no thread will complete before another thread starts executing. It would always fetch the wrong data.

```
(c) // Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
arr[i] = 0;
```

Faster than serial, each arr[i] is different, it would not occur mistake. The parallel **for** directive指令 will split up iterations of a for loop over various threads.

3.2 What potential issue can arise from this code?

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
    int threadCount = omp_get_num_threads();
    int myThread = omp_get_thread_num();
    for (int i = 0; i < n; i++) {
        if (i % threadCount == myThread) arr[i] -= 1;
    }
}
```

Although different threads change independent variables, if these variables share the same cache line, the threads may affect each other.
For example, thread A in core A, thread B in core B, A change arr[1], B change arr[2] , arr[1] and

arr[2] in the same cache line. A will send a message to invalidate the cache line in B, while B wants to send a message to invalidate the cache line in A. For this reason, they may have conflicts.

3.3
```
// Assume n holds the length of arr
    double fast_product(double *arr, int n) {
        double product = 1;
        #pragma omp parallel for
        for (int i = 0; i < n; i++) {
            product *= arr[i];
        }
        return product;
    }
```
(a) What is wrong with this code?

The variable **product** is shared by different threads in different cores. It may lead to a repeated calculation.

(b) Fix the code using #pragma omp critical
```
double fast_product(double *arr, int n) {
double product = 1;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    #pragma omp critical
    product *= arr[i];
}
return product;
}
```

(c) Fix the code using #pragma omp reduction(operation: var).
```
double fast_product(double *arr, int n) {
double product = 1;
#pragma omp for reduction(* : product )
for (int i = 0; i < n; i++) {
product *= arr[i]; // product them after add
}
return product;
}
```

4 Instruction Level Parallelism - Trace Scheduling

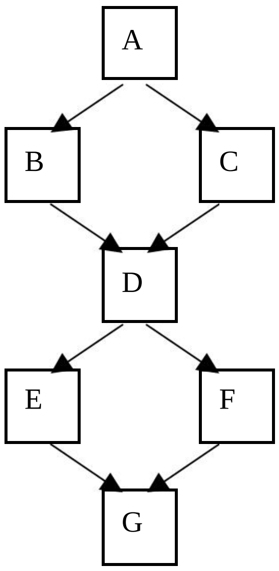
Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively推测的, 推理的 executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a general technique that can be used in different types of machines and in this question, we apply it to a single-issue RISC-V processor.

Consider the following piece of C code (% is modulus) with basic blocks labeled:

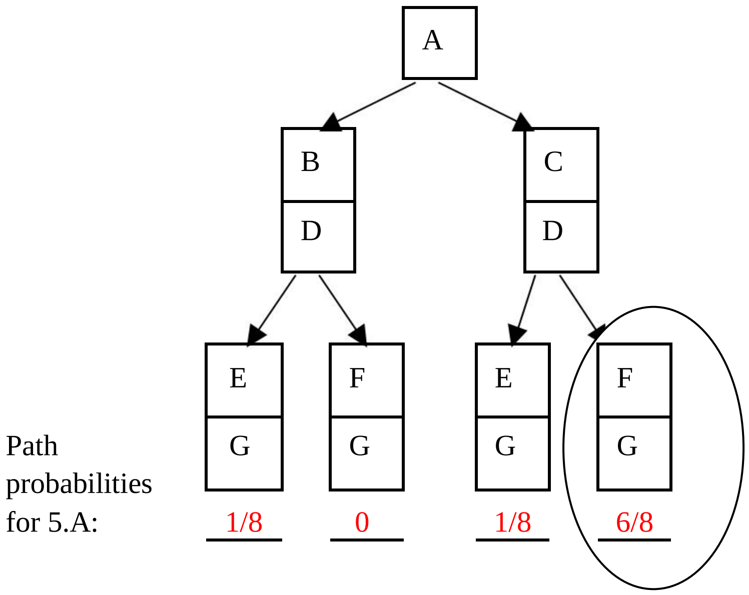
```
A       if (data % 8 == 0)
B       X = V0 / V1; else
C           X = V2 / V3;// 1 2 3 4 5 6 7 8
D       if (data % 4 == 0) // 能被 8 整除一定能被 4 整除
E       Y = V0 * V1; else
F           Y = V2 * V3;
G
```

Assume that **data** is a uniformly distributed integer random variable that is set sometime before executing this code.

The program’s control flow graph is



The decision tree is



A control flow graph and the decision tree both show the possible flow of execution through basic blocks. However, the control flow graph captures the static structure of the program, while the decision tree captures the dynamic execution (history) of the program.

4.1 On the decision tree, label each path with the probability of traversing that path. For example, the leftmost block will be labeled with the total probability of executing the path ABDEG. (Hint: you might want to write out the cases). Circle the path that is most likely to be executed.

4.2 This is the RISC-V code:

```
A:      lw   x1, data            #  x cycles
        andi x2, x1, 7           # x2 <- x1 % 8
        beqz  x2, C
B:      div  x3, x4, x5          # X  <- V0 / V1 ycycles
        j    D
C:      div  x3, x6, x7          # X  <- V2 / V3 ycycles
D:      andi x2, x1, 3           # x2 <- x1 % 4
        bnez x2, F
E:      mul  x8, x4, x5          # Y  <- V0 * V1 z
        j    G
F:      mul  x8, x6, x7          # Y  <- V2 * V3 z
G:
```

This code is to be executed on a single-issue processor **with perfect branch prediction**. Assume that the memory, divider, and the multiplier are all separate, long latency, **unpipelined** units that can be run in parallel.
Assume that the load takes x cycles, the divider takes y cycles, and the multiplier takes z cycles. Approximately how many cycles does this code take ***in the best case, in the worst case***, and ***on average***? (ignore the latency of ALU)
This code has constant execution time.
Best case: x+ max(y,z)
Worst case: x+ max(y,z)
On average: x+ max(y,z)

4.3 With trace scheduling, we can obtain the following code:

```
ACF:    ld   x1, data
        div  x3, x6, x7          # X  <- V2 / V3
        mul  x8, x6, x7          # Y  <- V2 * V3
D:      andi x2, x1, 3           # x2 <- x1 % 4
        bnez x2, G
A:      andi x2, x1, 7           # x2 <- x1 % 8
        bnez x2, E
B:      div  x3, x4, x5          # X  <- V0 / V1
E:      mul  x8, x4, x5          # Y  <- V0 * V1
G:
```

We optimize only for the most common path, but the other paths are still correct. Approximately how many cycles does the new code take ***in the best case, in the worst case*** and ***on average***? Is it faster ***in the best case, in the worst case*** and ***on average*** than the code in Problem 4.2?

Best case: max(x,y,z) Because ld, div,mul could execute in parallel.
Worst case: max(x,y,z) + max(y,z) the second div,mul could execute in parallel.
On average: max(x,y,z) *6/8 + (max(x,y,z) + max(y,z))/8 + (max(x,y,z) + z)/8 = max(x,y,z) + max(y,z)/8 + z/8 <= max(x,y,z) + max(y,z)/4

If max (x, y,z) =x. The best case and average case are faster than the code in Problem 4.2. The worst case has as much time as Problem 4.2.
If max (x, y,z) =y , The best case are faster than the code in Problem 4.2. If y/4 < x The average case is faster than the code in Problem 4.2. The worst case max(y, z) + max(y,z) > x+ max(y,z)the code in Problem 4.2
If max (x, y, z) =z, the best case are faster than the code in Problem 4.2. If z/4 < x The average case is faster than the code in Problem 4.2 but slower otherwise. The worst case max(y,z) + max(y,z) > x+ max(y,z)the code in Problem 4.2.

We could optimize the most possible situation to accelerate average execution time.

5 Amdahl's Law

In the programs we write, there are sections of code that are naturally able to be sped up. However, there are likely sections that just can't be optimized any further to maintain correctness. In the end, the overall program speedup is the number that matters, and we can determine this using Amdahl's Law:

True Speedup = 1/(S+(1-S)/F)

where *S* is the non-sped-up part and *P* is the speedup factor (determined by the number of cores, threads, etc.).

5.1 You are going to run a convolutional network to classify a set of 100,000 images using a computer with 32 threads. You notice that 99% of the execution of your project code can be parallelized on these threads. What is the speedup?

1/(0.01+(1-0.01)/32) = 24.427

5.2 You run a profiling program on a different program to find out what percent of this program each function takes. You get the following results:

Function	% Time
f	30%
g	10%
h	60%

(a) We don't know if these functions can actually be parallelized. However, assuming all of them can be, which one would benefit the most from parallelism?
H

(b) Let’s assume that we verified that your chosen function can actually be parallelized. What speedup would you get if you parallelized just this function with 8 threads?

1/(0.4+(1-0.6)/8) = 2.1

6 Multithreading

This problem evaluates the effectiveness of multithreading using a simple database benchmark. The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

```
struct node { int key;
        struct node *next; struct data *ptr;
}
```

The following RISC-V code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key.

```
loop:   LW    x3, 0(x1)          # load a key
        LW    x4, 4(x1)          # load the next pointer
        SEQ   x3, x3, x2         # set x3 if x3 == x2
        BNEZ  x3, end            # found the entry
        ADD   x1, x0, x4
        BNEZ  x4, loop           # check the next node
end:
```

We run this benchmark on a single-issue in-order processor. The processor can fetch and issue one instruction per cycle. If an instruction cannot be issued发出 due to a data dependency, the processor stalls. Integer instructions **take one cycle to execute** and the result can be used in the next cycle. For example, if SEQ is executed in cycle 1, BNEZ can be executed in cycle 2. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

6.1 Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and non-blocking. After the processor issues发出 a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation. How many cycles does it take to execute one iteration of the loop in steady state?

104 cycles

Instruction	Start Cycle	End Cycle
LW x3, 0(x1)	0	100
LW x4, 4(x1)	1	101
SEQ x3, x3, x2	101	101
BNEZ x3, End	102	102
ADD x1, x0, x4	103	103

BNEZ x1, Loop	104	104

6.2 Now we add zero-overhead multithreading to our pipeline. A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle.
In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling轮叫调度 (similar to CDC 6600 PPUs这是啥我也不知道). Each of the N threads executes one instruction every N cycles. What is the **minimum** number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?
N个 threads, first thread begin frist lw at cycles 1, The nth thread begin first lw at cycles n, the first thread begin second lw at cycles n+1, The nth thread begin first lw at cycles 2n
first thread begin seq at cycles 2n+1 , 2n+1 >= 101, n= 50
6.3 How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time processor takes to find an entry with a specific key)? Assume the processor switches to a different thread every cycle and is fully utilized. Check the correct boxes.

	Throughput	Latency
Better	It could Fully utilize waiting time	
Same		
Worse		One thread get a time slice instead of full processor

6.4 We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency. What is the minimum number of threads to fully utilize the processor now? Note that the processor issues instructions in-order in each thread

Each thread can execute 6 instructions (SEQ, BNEZ, ADD, BNEZ, LW, LW). Therefore, to hide 98 cycles (100-2 (the number of instructions between LW and SEQ)) cycles between the second LW and SEQ, the processor needs 18threads.
At the beginning, thread 1 lw, lw, switch to thread2, lw,lw , switch to thread 3….
At thread 18, wait 64 cycles.
Then, thread 1seq ,(thread2 prepared) BNEZ, ADD, BNEZ, LW(to next seq 2 cycles, and 17thread*6 cycles=102), LW, switch to thread 2 , SEQ, BNEZ, ADD, BNEZ, LW, LW, switch to thread .