# Assignment 2

**Josh Green and Akhil Sankar**

## Problem 1

How can we (if we can!) modify the following loops so that they can be maximally parallelized. As a first step identify and clearly describe what sort of dependencies exist in every loop (instruction per instruction and loop iteration per loop iteration). Clearly justify your answer whether the loop is parallelizable or not and if so how.

```
Loop 1: (3 pts)
for (i = 0; i < 100; i++) {
    A[i] = A[i] * B[i]; /* S1 */
    B[i] = A[i] + c;    /* S2 */
    A[i] = C[i] * c;    /* S3 */
    C[i] = D[i] * A[i]; /* S4 */
}

Loop 2: (3 pts)
for (i = 0; i < 100; i++) {
    A[i] = A[i] + B[i];    /* S1 */
    B[i+1] = C[i] + D[i];  /* S2 */
}
```

### Loop 1 Analysis

In loop 1: - S2 depends on the updated value of S1 - S4 depends on the updated value from S3 So S1 can be run before all other lines in the loop, S2 and S3 can be run in parallel after S1, and S4 can be run after S3. The modified loop is as follows. As for the loop iterations, they are independent and can run in parallel.

In other words, each loop has dependencies within itself, but different loop iterations are entirely independent, and each loop iteration can be run in parallel. For each loop iteration, S4 needs to wait for S3 to finish, and S2 needs to wait for S1 to finish.

In Loop 2: - S1 and S2 have no dependencies within the loop, but S2 depends on the value of B[i+1] from a different loop iteration. So S1 and S2 can be run in parallel within the same loop iteration, but different loop iterations cannot be run in parallel due to the dependency of S2 on the next iteration's value of B. In its current form, loop 2 cannot be parallelized. Furthermore, it isn't possible to modify loop 2 to be parallelizable without changing the logic of the program, as each iteration of S2 depends on the result of the previous iteration.

## Problem 2

Use the picture of the classroom showing a number of students attending the class, and project the seats as being positions in a bi-dimensional array, while the students are elements in the bi-dimensional array. The end goal here is to calculate the average age of the class collaboratively. One approach can be

completely sequential computation, while other approaches can entail one or more parallel phases, some of which are addressed in class: e.g., parallelizing work over rows or over columns, divide and conquer methods, etc. You are to select two methods for calculating the average age of the class, describe these, calculate the communication, timing, and computation costs/complexities in each, and find the optimal of the two. Our assumptions will be as follows: rows are the 8 or 9 horizontal sequences of seats, while columns are considered the vertical ones (according to the picture on CANVAS). For timing/communication costs, please consider the path/time it takes for one participant (node, student) to reach another. For example, if students sit at the same row and they are next to each other, then the communication/timing cost for student 1 to reach student 2 is 1 (clock cycle). However if the two students are in the same row but there are 3 columns empty between them, then the communication/timing cost is assumed to be 4. If communication is across rows but in the same column, then for a student in a given row to reach the student in the next row vertically, assume that the cost is 1.5 clock cycles. For communication across a diagonal, assume that the cost for one student to communicate to a neighbor student is 2 clock cycles.

The computation cost is decided as follows: addition that involves 2 + 2 digit numbers takes 2 clock cycles, addition that involves 3 + 2 digit numbers takes 3 clock cycles, addition that involves 4 + 2 digit numbers takes 4 clock cycles, etc. Division operation takes 10 clock cycles. Also, if you are going to be using algorithms such as divide and conquer that are not simply based on the proximity of nodes at each level but on pre-agreed arrangement of participating pairs at each level, this requires some type or prior scheduling with prior overhead. Assume that the computation cost or overhead for the schedule to be arranged and communicated across all participants is: 0.25 x Total Number of Nodes clock cycles. So, all these above are the basic assumptions for you to compare and decide on the optimal algorithm to use. If there are additional assumptions that should be made so that you are able to make your calculations please take the freedom to do so but describe them here. Finally, as the goal is to calculate the average age of graduate students, you can make the assumptions that the age of each student is between 22-27 years old (leave the professor .... out of this wishful thinking!). Good luck!

## General Analysis

While the problem specifies the number of columns and rows, I prefer to think of it a little more abstractly at first. Let `N` be the number of rows and `M` be the number of columns. The total number of students isn't necessarily `N * M`, as some seats may be empty, but there won't be any more than `N * M` students in the class.

There are two main considerations when it comes to time complexity. Let's define them as follows:

- **Computation Cost ($T_{Comp}$)**: The time taken for a single computation. These will be defined as:
  - **$T_{22}$**: Time taken to add two 2-digit numbers. `T_{22} = 2 clock cycles`.
  - **$T_{32}$**: Time taken to add a 3-digit number and a 2-digit number. `T_{32} = 3 clock cycles`.
  - **$T_{42}$**: Time taken to add a 4-digit number and a 2-digit number. `T_{42} = 4 clock cycles`.
  - **$T_{Div}$**: Time taken to divide a number. `T_{Div} = 10 clock cycles`.
  - The number of clock cycles for any addition operation can be calculated as `ceil((digits_in_numb1 + digits_in_numb2)/2)`.
- **Communication Cost ($T_{Comm}$)**: This is the total time taken for all communication between students (nodes) to share their ages and intermediate results.
  - **$T_{Row}$**: Time taken to communicate between two seats in the same row. `T_{Row} = 1 clock cycle`.

- **$T_{Col}$**: Time taken to communicate between two seats in the same column. `T_{Col} = 1.5 clock cycles`.
- **$T_{Diag}$**: Time taken to communicate between two seats in a diagonal. `T_{Diag} = 2 clock cycles`.

Immediately, we see that strides within a row are cheaper than strides within a column, so any scenario where `N=M` should be done row-wise if possible. Going diagonally should also be avoided: if we go diagonally, we will end up with a subgroup that is larger than any row-based or column-based subgroup, and thus will be more expensive.

### Sequential Calculation

The simplest way to calculate the average age is to do it sequentially. We snake through the rows and columns until we calculate each student's age, and then divide by the total number of students.

$T_{CompTotal} = \sum_{n=1}^{N-1} T_{Comp}[n]\ + T_{Div}$

Let's assume the average cost of adding a student's age is `T_{Comp} = 3.5 cycles`. Since it's a college class, the average age is likely to be in the mid-20s, so we only add two 2-digit numbers about 4 or 5 times before the cumulative sum is three digits. The next 15 or so will be between a 3 and 2-digit number, and the rest will be between a 4 and 2-digit number. So we can approximate the total computation cost as 3.5 cycles per addition (I will use this number for the rest of the problem as it simplifies things a great deal). This gives us:

$T_{CompTotal} = (N * M - 1) * 3.5 + 10 = 3.5NM - 3.5 + 10 = 3.5NM + 6.5$

For communication, the cheapest way will be to snake across a row, and then upwards. This can be modeled as:

$T_{CommTotal} = N(M-1) + 1.5(N-1)$

This is because there are M-1 strides in each row, and we can add on the extra 1 stride to go to the next row (1.5 cycles). Now, we can combine the two to get the total time complexity:

$T_{Total} = T_{CompTotal} + T_{CommTotal} = (NM - 1) * 3.5 + 10 + N(M-1) + 1.5(N-1)$

I am going to assume a 13x8 classroom, so N=8 and M=13. This gives us $T_{Total} = 477$ clock cycles.

### Row-Wise Parallel Calculation

With row addition, we consider one row to start, since we assume the rows are computed in parallel. Each row has M students, so the computation cost is:

$T_{CompRow} = (M-1) * 3.5$

And $T_{CommRow} = 1(M-1)$, since we only move down a row `N - 1` times.

There is also delay for consolidating the results from each row. Let's assume this is done sequentially, and has a computation cost of: $T_{CompConsolidate} = (N-1) * (3.5 + 1.5) + 10$.

So the total time complexity is: $T_{RowTotal} = T_{CompRow} + T_{CommRow} + T_{CompConsolidate} = (M-1) * 3.5 + 1(M-1) + (N-1) * (3.5 + 1.5) + 10 = 99$ cycles when using `M=13`, `N=8`.

### Conclusion

In conclusion, we have analyzed the time complexity of computing the average age of students in a classroom setting. We considered both sequential and parallel approaches, taking into account the costs of computation and communication. The row-wise parallel approach yielded a total time complexity of 99 cycles, which is more efficient than the sequential approach. Other parallel approaches can provide different trade-offs. here are some potential ones:

- If the size of the groups is sufficiently large, the computation cost can become more expensive
- If the age of people is higher than what we assumed, the computation cost can become more expensive as the cumulative sum grows larger faster
- If `M>>>N`, then column-wise addition will be more efficient (computation time exceeds communication time)
- If we don't use nodes directly adjacent to each other, we also need to account for the increased communication cost

## Problem 3

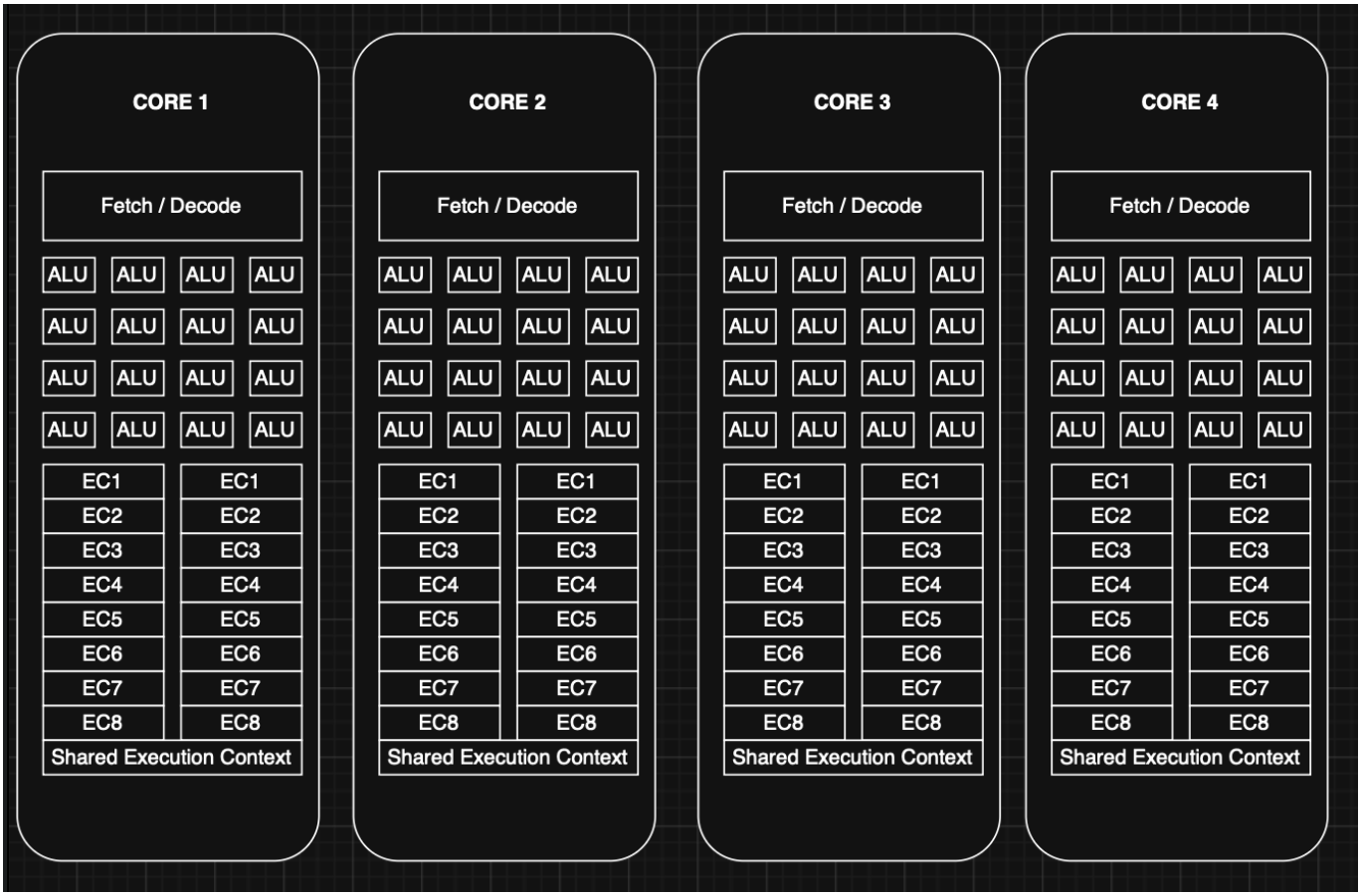
Both parts A and B (1 through 3) are combined in this part. The other sections of part 2 are answered after this part.

Some Notes:

- **Functional Unites**: Fetch/Decode Units, abbreviated as `F/D unit`
- **SIMD**: To support SIMD, the same instruction encoded in the F/D unit can be executed in parallel for multiple instances of data. Thus, each ALU associated with an F/D unit can execute the instruction for a given instance of data. An 8-way SIMD executes the same instruction for 8 different instructions, using 8 ALUs.
- **Instruction Level Parallelism (ILP)**: Can be achieved when a core has multiple F/D units. The compiler identifies independent instructions in an instruction stream and executes them in parallel using these F/D units.
- **Execution Context (EC)**:
- Individual: Each ALU gets its own data registers and local state.
- Shared: All ALUs share the instruction stream and control flow.

### Part A: 4 Core, 16 Wide SIMD

4 Core, 16 Wide SIMD




**Part 1**

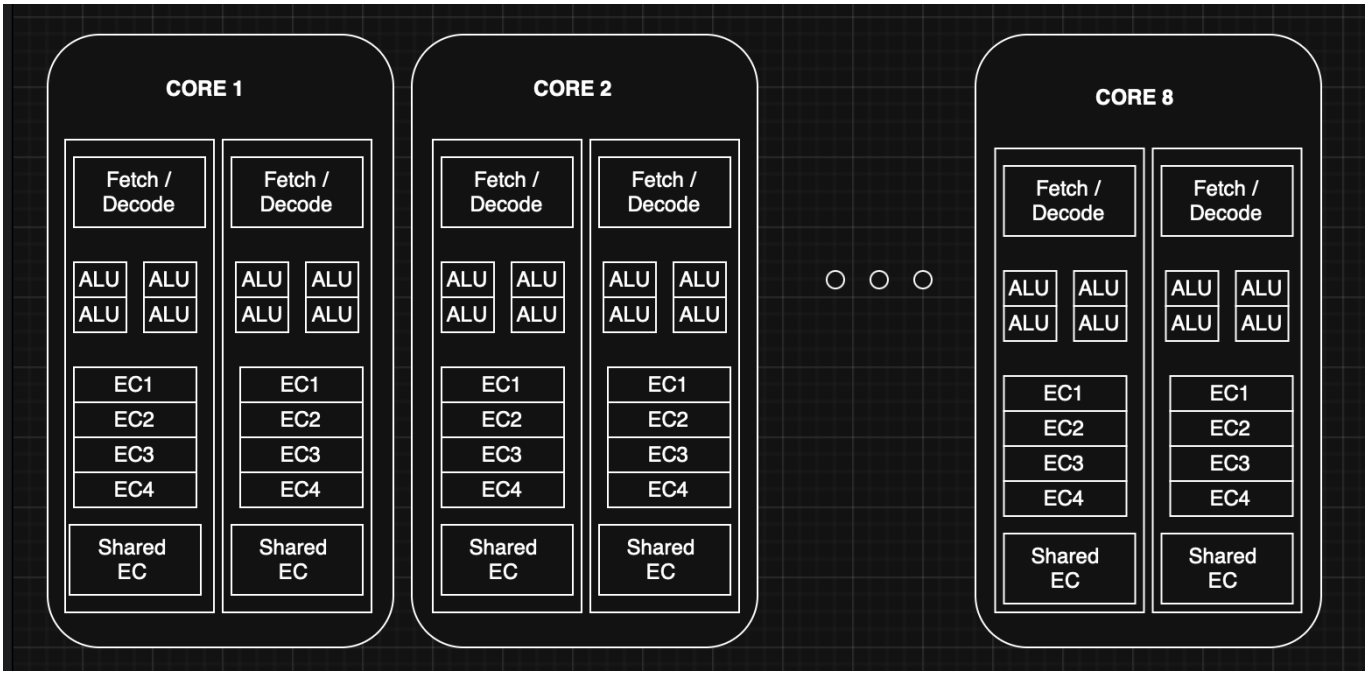
- **Functional Units**: 1 F/D unit per core, so 4 total
- **ALUs**: 16 ALUs per core, so 64 total
- **Execution Context**: 1 per ALU and shared EC as noted above

**Part 2**

- **# of Instruction Streams**: 4 instruction streams across cores (one per F/D unit)
- **# of elements executed:** 64 elements processed in parallel across all cores (64 ALUs)
- **size of registers in ECs**: 32-bit wide each

Part B:

8 Core, 4 Wide SIMD, 2 way ILP




Each core is capable of 2-way ILP, meaning that each core has 2 independent units with their own F/D unit, ALUs, and registers. Each unit is also 4-way SIMD compatibile, using all 4 ALUs to execute 1 instruction for 4 data instances in parallel.

**Part 1**

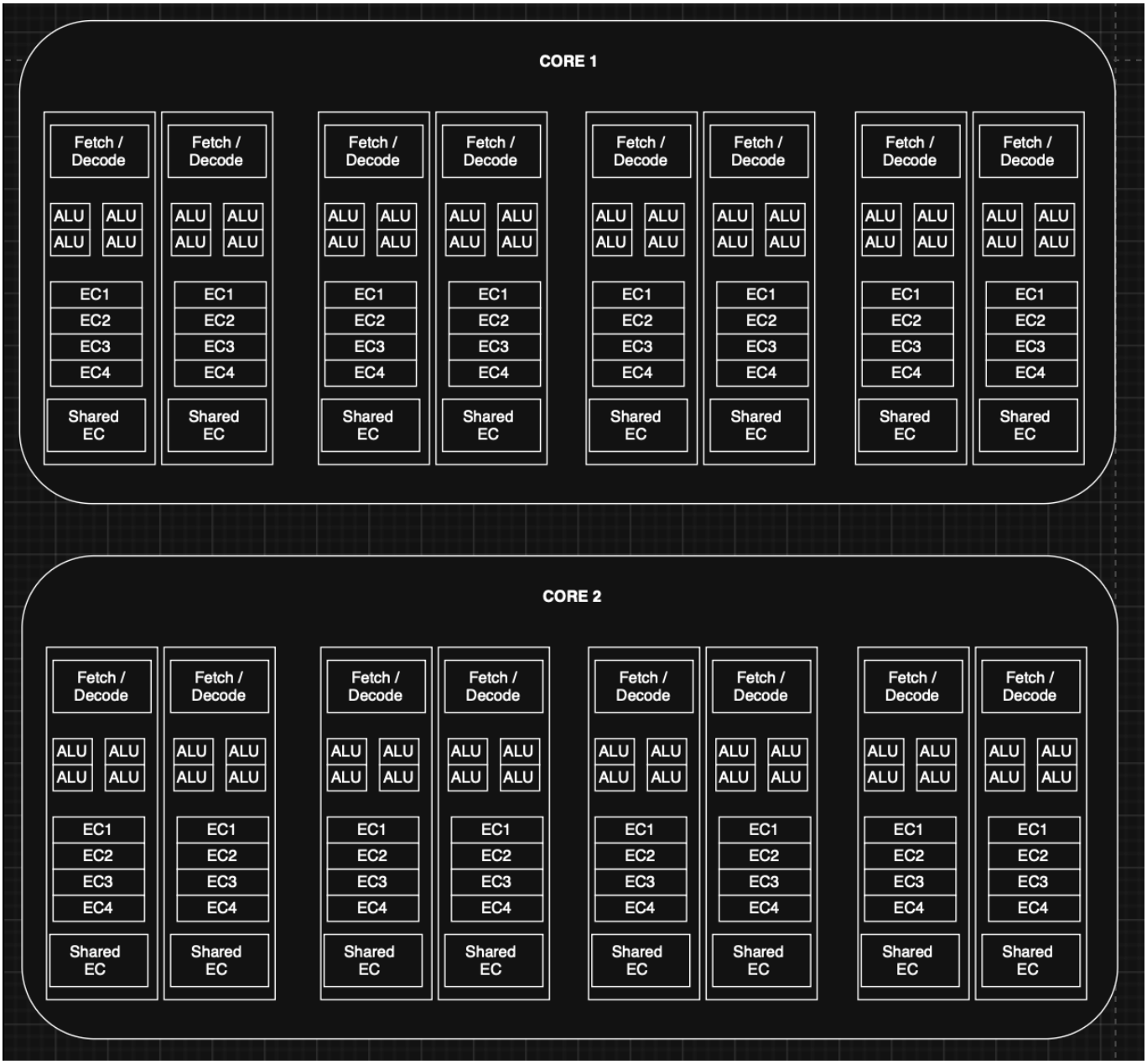
- **Functional Units**: 2 F/D units per core, so 16 total
- **ALUs**: 8 ALUs per core, so 64 total
- **Execution Context**: each super-scalar unit provides dedicated ECs and shared EC to access the common instruction.

**Part 2**

- **# of Instruction Streams**: 16 instruction streams across all cores (one per F/D unit)
- **# of elements executed:** 64 elements processed in parallel across all cores (64 ALUs)
- **size of registers in ECs**: 32-bit wide each

Part C:

2 Core, 4-Wide SIMD, 8-way ILP




This architecture is similar (at the core-level) to the previous example in Part B. However, this time, each core is capable of 8-way ILP (i.e. can execute 8 instructions in parallel). This means that each core has independent units composed of their own F/D, ALUs and registers. Further, each unit also is capable of 4-way SIMD, using 4 ALUs, to execute 1 instruction for 4 data instances in parallel.

**Part 1**

- **Functional Units**: 8 F/D units per core, so 16 total
- **ALUs**: 32 ALUs per core, so 64 total
- **Execution Context**: each independent unit provides dedicated ECs and shared EC to access the common instruction

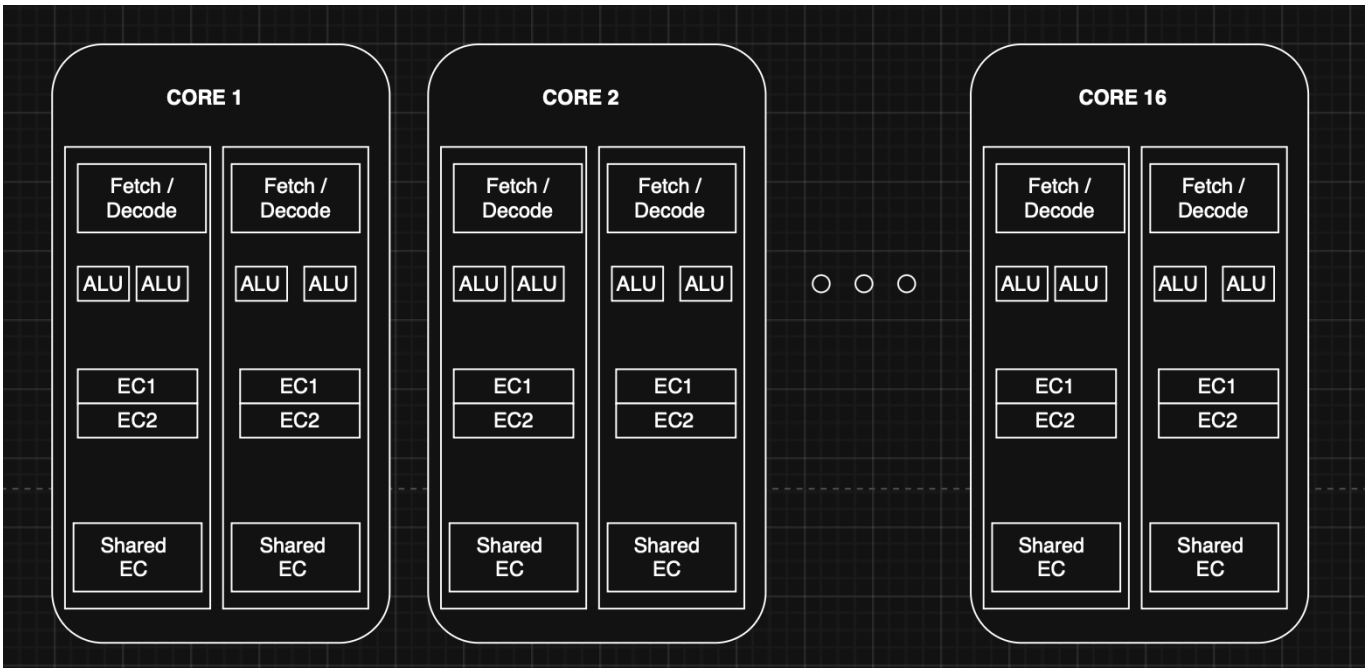
**Part 2**

- **# of Instruction Streams**: 16 instruction streams across all cores (one per F/D unit)
- **# of elements executed:** 64 elements processed in parallel across all cores (64 ALUs)

- **size of registers in ECs**: 32-bit wide each

Part D:

16 Core, 2-wide SIMD, 2-way ILP




This architecture is similar (at the core-level) to the previous example in Part B. However, this time, each core is capable of 8-way ILP (i.e. can execute 8 instructions in parallel). This means that each core has independent units composed of their own F/D, ALUs and registers. Further, each unit also is capable of 4-way SIMD, using 4 ALUs, to execute 1 instruction for 4 data instances in parallel.

**Part 1**

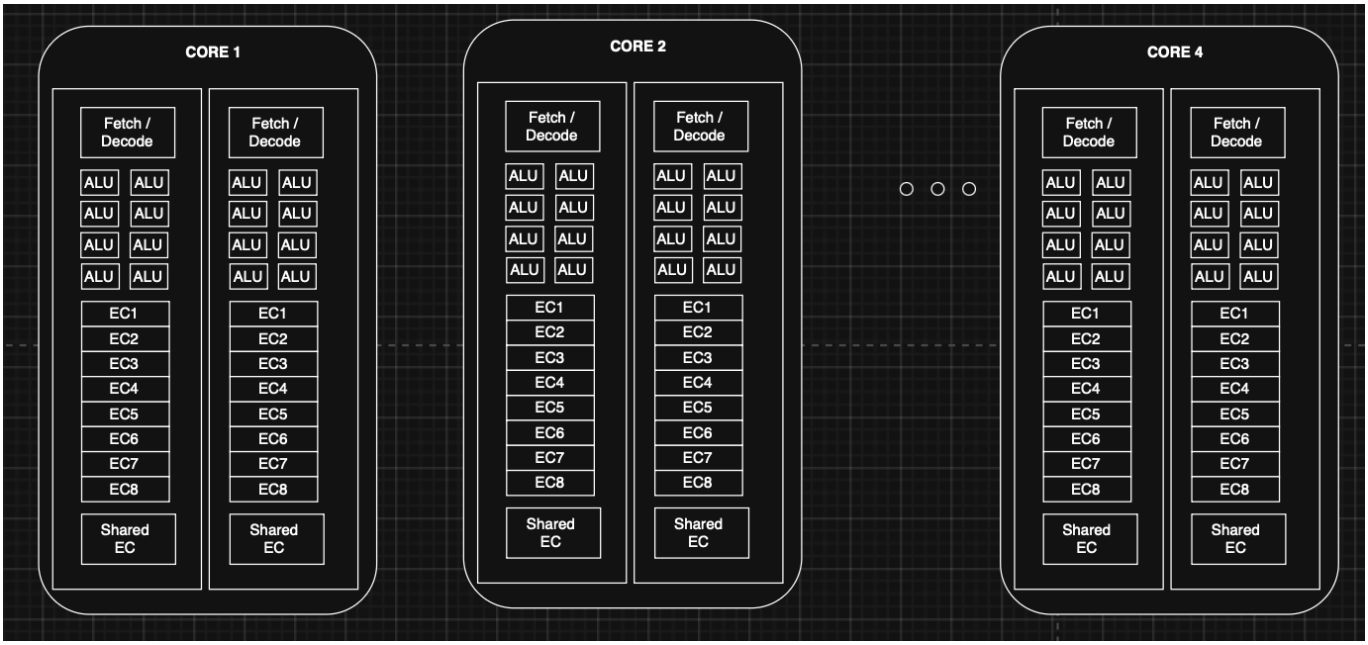
- **Functional Units**: 2 F/D units per core, so 32 total
- **ALUs**: 4 ALUs per core, so 64 total
- **Execution Context**: each independent unit provides dedicated ECs and shared EC to access the common instruction

**Part 2**

- **# of Instruction Streams**: 32 instruction streams across all cores (one per F/D unit)
- **# of elements executed:** 64 elements processed in parallel across all cores (64 ALUs)
- **size of registers in ECs**: 32-bit wide each

Part E:

4 Core, 8 Wide SIMD, 2-Way ILP




**Part 1**

- **Functional Units**: 2 F/D units per core, so 8 total
- **ALUs**: 16 ALUs per core, so 64 total
- **Execution Context**: each independent unit provides dedicated ECs and shared EC to access the common instruction

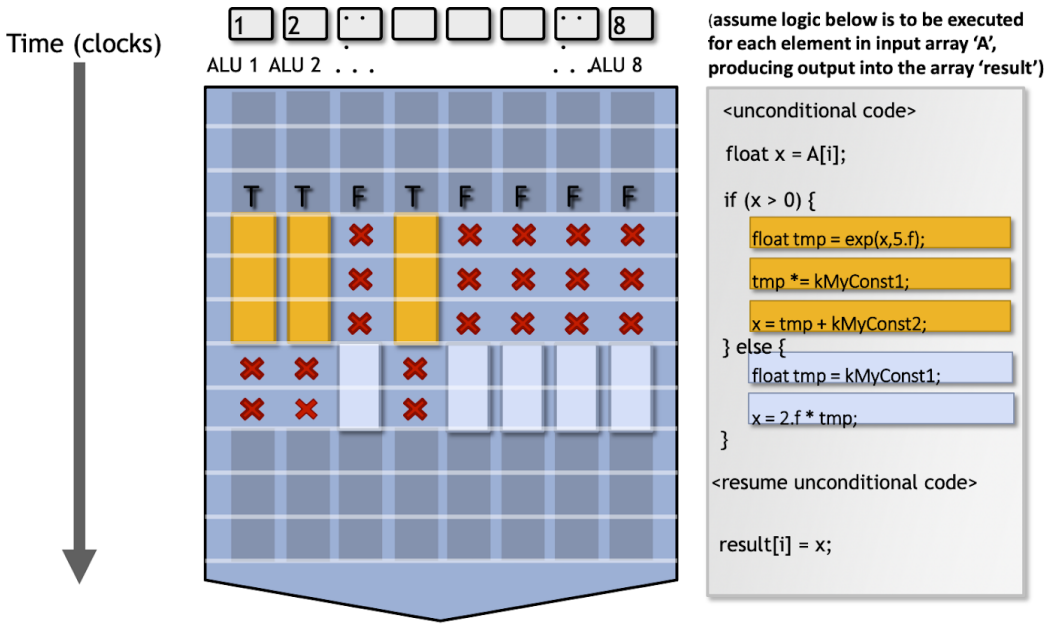
**Part 2**

- **# of Instruction Streams**: 8 instruction streams across cores (one per F/D unit)
- **# of elements executed:** 64 elements processed in parallel across all cores (64 ALUs)
- **size of registers in ECs**: 32-bit wide each

## Part 2 Continued

- Part 4: Diversion: measure of conditionality (ie. non-linear instruction flow) in software
    - The 16 core, 2-wide SIMD, 2 way ILP can fetch and decode 32 independent (different) instruction streams in parallel, making it the best option to parallelize for strongly divergent code.
    - In contrast, the 4 core, 16 wide SIMD option is likely the least efficient for divergent code. In a SIMD architecture, all the ALUs must run the same instruction. Thus, in the event of a divergent instruction stream, all the ALUs will run every instruction over all conditional blocks, discarding the invalid conditional branch post-execution. This is a waste of CPU cycles, as seen in the

diagram below:




- Parts 5 and 6: The compiler is responsible for identifying potential instruction level parallelism (ILP) within user level software and optimally leveraging the available hardware functional units to achieve parallelism.
    - The compiler is most heavily involved determining ILP for architectures with the most F/D units: 16 core, 2-wide SIMD, 2 way ILP (providing 32 F/D units)
    - The compiler would perform the least ILP compute for the 4 core, 16 wide SIMD architecture (just 4 F/D units).
- Part 7: Comparing Chips (8 core, 4 wide SIMD, 2 way ILP vs 16 core, 2-wide simd, 2 way ILP)
    - Taylor Expansion (TE) performance: TE logic can only be parallelized effectively across its outer loop (i.e. its inner loop must be processed sequentially due to upstream dependencies). However, the SIMD architecture would be more limited in its ability to process the inner loop (added degree of divergence), as discussed above. Therefore, the more superscalar (d) option would be more efficient for this algorithm.
    - Other algorithm performance: considering more "embarrassingly parallel problems" such as in Monte Carlo simulations (flipping 1 million independent coins), where there is little divergence or sequential dependence, SIMD would be equally efficient (perhaps even more so since it would not employ as many F/D operations)

## Part 3

Given:

- duration of sequential TE iteration: 150 clock cycles = 300 nsec/external iteration.
- workload: 6000 iterations
- Best case: 64 outer loops processed in parallel (common across architectures)
- Worse case: 6000 * 300 = 1.8M cycles (completely sequential)
    - We will consider a completely sequential pipeline for the worst case scenario. In these cases, SIMD will collapse to just 1 sequential pipeline.

- Architecture

| Architecture | Best Performance (nsec) | Worst Performance (nsec) |
|---|---|---|
| 4 core, 16 wide SIMD | 6000/64 * 300 = 281.2K | 6000 * 300 = 1.8M |
| 8 core, 4 wide SIMD, 2 way ILP | 6000/64 * 300 = 281.2K | 6000/16 * 300 = 112.5K |
| 2 core, 4-wide SIMD, 8-way ILP | 6000/64 * 300 = 281.2K | 6000/16 * 300 = 112.5K |
| 16 core, 2-wide SIMD, 2 way ILP | 6000/64 * 300 = 281.2K | 6000/32 * 300 = 56.25K |
| 4 core, 8-wide SIMD, 2 way ILP | 6000/64 * 300 = 281.2K | 6000/8 * 300 = 225K |