# ECS201A - ASSIGNMENT2

**Hezhi Xie, Mo Lyu**
**Email**: hezxie@ucdavis.edu, molyu@ucdavis.edu
**Github UserName**: helenxie-bit, MOLl02
**Github TeamName**: ababa

## 1 ANALYSIS AND SIMULATION: STEP 0

1. For the DAXPY's assembly code, identify the ROI. In your report, copy the assembly code segment corresponding to `m5_work_begin` and `m5_work_end`?

```
# daxpy.cpp:21: m5_work_begin(0,0);
        li      a1,0            #,
        li      a0,0            #,
        call    m5_work_begin           #
        lui     a5,%hi(.LC4)    # tmp195,
        li      s1,1048576              # tmp179,
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fld     fa3,%lo(.LC4)(a5)       # tmp181,
        add     s1,s0,s1        # tmp179, _14, ivtmp.125
# daxpy.cpp:21: m5_work_begin(0,0);
        mv      a5,s0   # ivtmp.133, ivtmp.125
.L35:
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fld     fa4,0(a5)       # MEM[(double *)_56],
                                  MEM[(double *)_56]
        fld     fa5,0(s2)       # MEM[(double *)_49],
                                  MEM[(double *)_49]
# daxpy.cpp:25:   for (int i = 0; i < N; ++i)
        addi    a5,a5,8 #, ivtmp.133, ivtmp.133
        addi    s2,s2,8 #, ivtmp.132, ivtmp.132
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fmadd.d fa5,fa5,fa3,fa4 # _5, MEM[(double *)_49],
                                  tmp181, MEM[(double *)_56]
# daxpy.cpp:27:     Y[i] = alpha * X[i] + Y[i];
        fsd     fa5,-8(a5)      # _5, MEM[(double *)_56]
# daxpy.cpp:25:   for (int i = 0; i < N; ++i)
        bne     s1,a5,.L35      #, _14, ivtmp.133,
# daxpy.cpp:32:     m5_work_end(0,0);
        li      a1,0            #,
        li      a0,0            #,
        call    m5_work_end             #
```

## 2 ANALYSIS AND SIMULATION: STEP I

1. If you were to divide instructions in a program into the three categories of **integer**, **floating point**, and **memory** instructions, do you think each category would constitute equal parts of a program?

**Before simulation:**
We think each category would not constitute equal parts of a program. The distribution of instructions among the three categories depends on the computational requirements and

characteristics of the application, and it's typically not evenly distributed.

For instance, programs involving a substantial amount of logical and integer-based operations (such as control flow, integer arithmetic, and bitwise operations) tend to have a higher proportion of integer instructions. On the other hand, scientific simulations, numerical computations, and applications dealing with real numbers often contain a considerable number of floating-point instructions. Additionally, programs with frequent data manipulations, data access, and memory transfers may exhibit a significant proportion of memory instructions.

**After simulation:**
When simulating the execution of the DAXPYWorkload on HW2TimingSimpleCPU, as shown in Figure 1 and Table 1, the breakdown of instruction counts reveals distinct percentages of the three categories. **Integer** instructions (including IntAlu) total **393,224**, constituting **42.86%** of the total commited instructions. **Floating-point** instructions (including FloatMultAcc) amount to **131,072**, representing **14.29%**. **Memory** instructions (including FloatMemRead and FloatMemWrite) total **393,217**, comprising another **42.86%**.

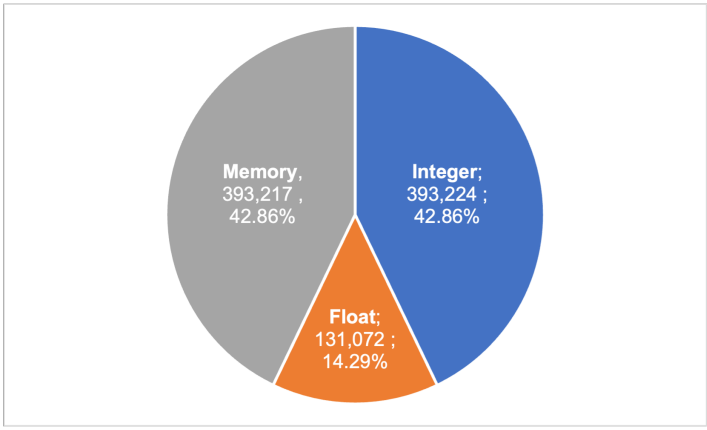


Figure 1: Distribution of Instructions when Simulating DAXPYWorkload

| Category | commitedInstType | Count | Percentage |
|---|---|---|---|
| Integer | IntAlu | 393,224 | 42.86% |
| Float | FloatMultAcc | 131,072 | 14.29% |
| Memory | FloatMemRead | 262,145 | 28.57% |
| | FloatMemWrite | 131,072 | 14.29% |
| | Subtotal | 393,217 | 42.86% |
| **Total** | | **917,513** | **100.00%** |

Table 1: Distribution of Instructions when Simulating DAXPYWorkload

Similarly, when executing the HelloWorldWorkload on HW2TimingSimpleCPU, the results shown in Figure 2 and Table 2 reveal an uneven breakdown of instruction counts. **Integer** instructions (including IntAlu, IntMult, and IntDiv) total **4,179**, accounting for **62.43%**. There are **no floating-point** instructions recorded. **Memory** instructions (including MemRead, MemWrite, and FloatMemWrite) amount to **2,488**, constituting **37.17%**. Additionally, instructions classified as **No_OpClass** total **27**, making up **0.40%** of the workload.

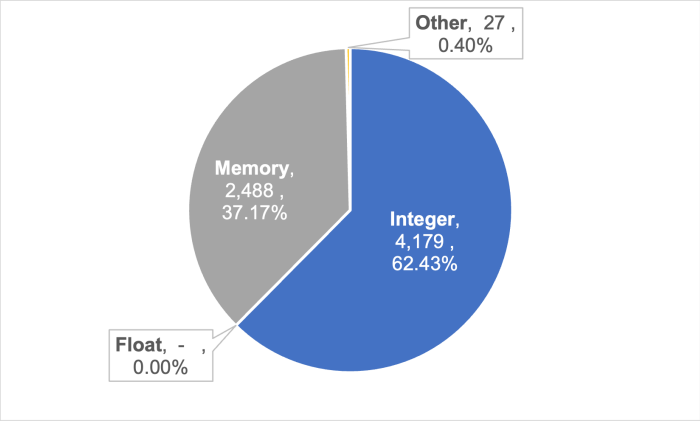


Figure 2: Distribution of Instructions when Simulating HelloWorldWorkload

| Category | committedInstType | Count | Percentage |
|---|---|---|---|
| Integer | IntAlu | 4,173 | 62.34% |
| | IntMult | 2 | 0.03% |
| | IntDiv | 4 | 0.06% |
| | Subtotal | 4,179 | 62.43% |
| Memory | MemRead | 1,257 | 18.78% |
| | MemWrite | 1,219 | 18.21% |
| | FloatMemWrite | 12 | 0.18% |
| | Subtotal | 2,488 | 37.17% |
| Other | No-OpClass | 27 | 0.40% |
| **Total** | | **6,694** | **100.00%** |

Table 2: Distribution of Instructions when Simulating HelloWorldWorkload

These findings indicate that instructions are **not evenly distributed** among the three categories of integer, floating point, and memory, aligning with our analysis in the ”Before simulation” section.

2. Do you think different programs will have a different mix of these categories? Why?

**Before simulation:**
We think different programs will have a different mix of these categories. Because the mix of integer, floating-point, and memory instructions in a program is highly dependent on the application, algorithms, data structures, programming language, compiler optimizations, and hardware architecture involved.

For example, scientific simulations and numerical computations often involve heavy use of floating-point instructions to perform complex mathematical calculations; systems programming or low-level software may involve a higher proportion of integer instructions for tasks such as control flow, bitwise operations, and data manipulation; database applications or programs dealing with large datasets may have a significant number of memory instructions for data access and manipulation. Therefore, different programs will exhibit different mixes of these instruction types.

**After simulation:**

When simulating the execution of the DAXPYWorkload and HelloWorldWorkload on HW2TimingSimpleCPU, the results depicted in Figure 3 highlight variations in the percentage distribution of instruction categories between the two scenarios. Specifically, when simulating the DAXPYWorkload, **integer**, **floating-point**, and **memory** instructions constitute **42.86%**, **14.29%**, and **42.86%** respectively. In contrast, when simulating the HelloWorldWorkload, these categories represent **62.43%**, **0.00%**, and **37.17%** respectively. This example underscores that the mix of instructions differs across programs due to their distinct applications and computational requirements.

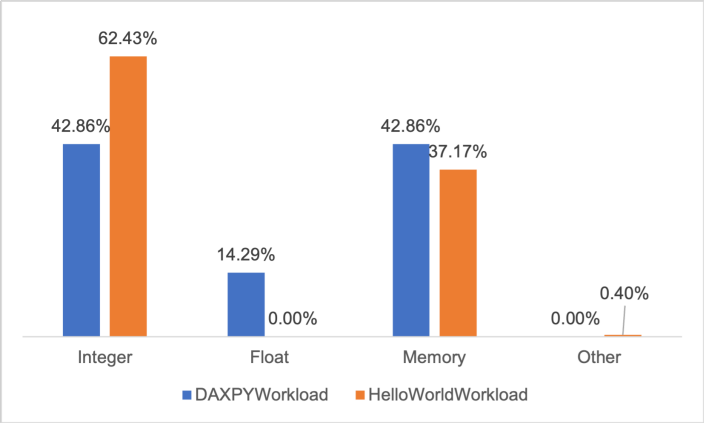


Figure 3: Comparison of Distribution of Instructions

## 3 Analysis and Simulation: Step II

1. Between the 3 designs, which one did you find to be the best design?

| Configurations | Issue Latency | Floating Point Operation Latency | SimTicks | Change |
|---|---|---|---|---|
| #1 | 4 | 2 | 2294374000 | 0 |
| #2 | 2 | 4 | 2295038250 | +0.0290% |
| #3 | 8 | 1 | 2435775500 | +6.6163% |

Table 3: Results of Three Configurations

The simulation results are shown in Table 3, where

$$\text{Change} = \frac{\text{SimTicks(Configurations)} - \text{SimTicks(\#1)}}{\text{SimTicks(\#1)}} \times 100\%$$

Configuration #1 exhibits the best performance with the lowest SimTicks. Both Configurations #2 and #3 show an increase in SimTicks compared to Configuration #1, indicating a decrease in performance. Notably, Configuration #2 has only a marginal increase (0.0290%) in SimTicks, suggesting a relatively small difference in performance.

Therefore, our results show **4 cycles for issue latency and 2 cycles for floating point operation latency** is the best design among the 3 designs.

2. Why do you think your chosen design in question 1 results in the best performance? Can you reason about why you would prefer optimizing one of the latencies over the other?

In order to analyze the performance with different design, we examined the instruction pipeline within the DAXPY loop based on the assembly code, as shown in Figure 4, Figure 5, and Figure 6 (**Note: only the loop part of the assembly code was considered, excluding some initialization operations**).

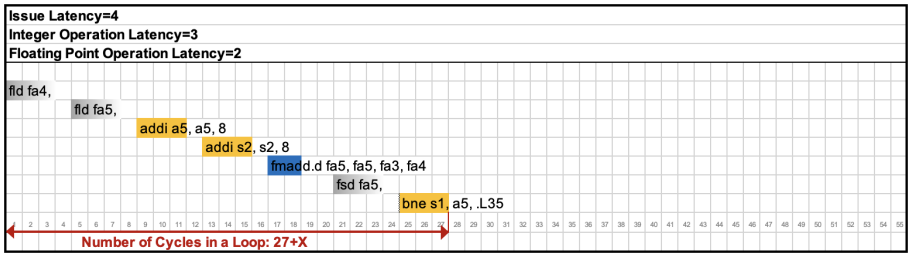


Figure 4: Instruction Pipeline in Configuration#1

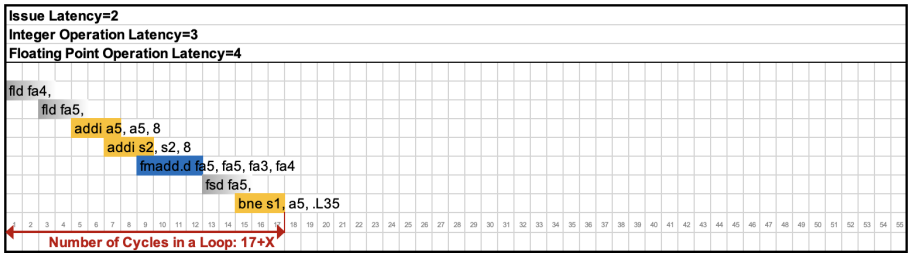


Figure 5: Instruction Pipeline in Configuration#2

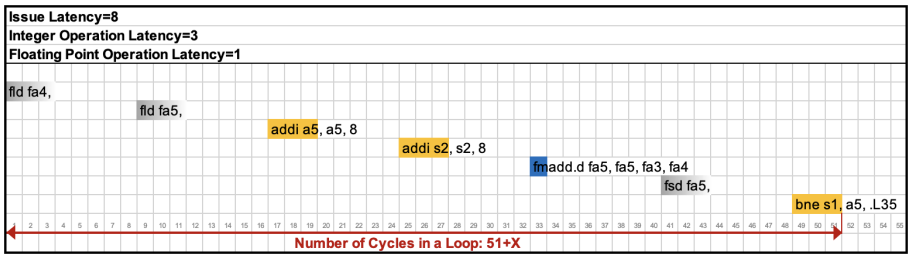


Figure 6: Instruction Pipeline in Configuration#3

Within the loop, there are three integer instructions (`addi`, `bne`), one floating point instruction (`fmadd`), and three memory instructions (`fld`, `fsd`). Since the program runs on an in-order pipelined CPU, instructions are scheduled every issue latency, except in cases where dependencies between instructions exist (e.g., the execution of `fsd` depends on the completion of `fmadd`). Additionally, with two integer units available, two integer instructions can be executed simultaneously.

Since we only know the issue latency, integer operation latency, and floating-point operation latency, and lack information about the latency of memory instructions, we can only infer the minimum number of cycles required for each case. For the three scenarios, the minimum cycle counts in the loop are determined to be **27**, **17**, and **51**, respectively. The variable "**X**" in the figures represents the number of cycles corresponding to memory instructions (which is unknown) and the delay caused by dependencies among different instructions (e.g., fmadd must wait for the completion of the two fld instructions). As "**X**" increases, the distinctions among different configurations tend to decrease.

Based on the analysis above, the best design appears to be configuration #2, which does not align with our simulation results in question 1. We speculate that this discrepancy may stem from configuration #1 achieving a balance between issue and floating-point operation latencies, while configuration #2 may have been affected by **pipeline stalls and resource contention due to an imbalance in these latencies**.

Specifically, despite 'FloatMultAcc' operations constituting only 14.29% of the total instructions(Table 1), their computational intensity within the DAXPY computation, characterized by scalar-vector multiplications and additions, disproportionately impacts overall performance. The significant portion (42.86%) of memory-related operations, closely linked to these floating-point calculations, further emphasizes the workload's reliance on efficient handling of floating-point data. Therefore, achieving a balance between issue latency and floating-point operation latency is crucial for maintaining a steady flow of instructions, minimizing stalls, and expediting computationally intensive floating-point operations. This balance not only aligns with the mixed nature of the DAXPY workload—encompassing integer, floating-point, and memory operations—but also leverages the in-order execution model of the HW2MinorCPU, where efficient pipeline utilization is paramount.

To validate our speculation, we conducted additional simulations. The extended simulation results in Table 4 present an intriguing insight into the performance dynamics of the HW2MinorCPU when executing the DAXPY workload, particularly focusing on the interplay between issue latency and floating-point operation latency. Surprisingly, when we simultaneously reduced the issue latency and floating-point latency to 2 cycles, the execution time was the longest, exceeding that of configuration #1 by more than **22%**.

| Configurations | Issue Latency | Floating Point Operation Latency | SimTicks | Change |
|---|---|---|---|---|
| #1 | 4 | 2 | 2294374000 | 0 |
| #2 | 2 | 4 | 2295038250 | +0.0290% |
| Additional#1 | 4 | 4 | 2321879250 | +1.1988% |
| Addiitonal#2 | 2 | 2 | 2803460500 | +22.1885% |

Table 4: Results of Additional Simulations

These results confirm the importance of balancing issue latency and floating-point operation latency. While a lower floating-point operation latency theoretically allows for quicker execution of floating-point instructions, it could inadvertently lead to increased pipeline contention and hazards. This is particularly relevant in an in-order execution environment like the HW2MinorCPU, where rapid completion of floating-point operations could result in dependent instructions being issued prematurely, causing pipeline stalls. Moreover, when coupled with lower issue latencies, this rapid execution might exacerbate resource contention, as multiple instructions vie for execution units.

Additionally, issue latencies play a crucial role in pipeline efficiency. Lower issue latencies can lead to a higher rate of instruction issuance, which, while seemingly beneficial, may increase the likelihood of pipeline stalls and resource contention due to unresolved dependencies and data hazards. Conversely, higher issue latencies, though seemingly suboptimal, can inadvertently allow for better synchronization of instruction flow, reducing pipeline stalls and improving overall throughput.

In summary, these findings indicate that the optimal configuration for the DAXPY workload on the HW2MinorCPU is not solely reliant on minimizing issue latency or floating-point operation latency. Instead, it requires a holistic approach that considers **the intricate balance between issue latency and floating-point operation latency**. This balanced configuration approach leads to reduced pipeline stalls and enhanced overall performance, ensures effective pipeline utilization, aligning with the processor's architectural characteristics and the specific demands of the workload.

# 4 ANALYSIS AND SIMULATION: STEP III

1. Use Amdahl's Law and the information you gathered from **Step I** to predict the speed up of each **improved** case over the **baseline**. Which design would you choose? **NOTE**: The only simulation result you can use to answer this question is the data you gathered from **Step I**.

Amdahl's law is a formula that gives the theoretical speedup of the execution time of a task at a fixed workload that can be expected of a system whose resources are improved. It states that the overall performance improvement gained by optimizing a single part of a system is limited by the proportion of the part that cannot be improved. Amdahl's law can be formulated in the following way:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

Where:

- $S$ is the theoretical speedup of the execution of the whole task;
- $p$ is the proportion of the part benefiting from improved resources originally occupied;
- $s$ is the speedup of the part of the task that benefits from improved system resources.

For the following three cases in this step:

| Case | Issue Latency | Integer Operation Latency | Floating Point Operation Latency |
|---|---|---|---|
| Baseline | 1 | 6 | 12 |
| Improvement 1 | 1 | 3 | 12 |
| Improvement 2 | 1 | 6 | 6 |

Table 5: Parameters of Three Cases

In Improvement 1, since we aim to improve the execution time of integer operations, based on the data gathered from Step I, the application of Amdahl's law formula suggests the anticipated speedup to be:

$$S_1 = \frac{1}{(1 - p_{integer}) + \frac{p_{integer}}{s_{integer}}} = \frac{1}{(1 - 42.86\%) + \frac{42.86\%}{2}} \approx 1.2728$$

In Improvement 2, since we aim to improve the execution time of floating operations, based on the data gathered from Step I, the application of Amdahl's law formula suggests the anticipated speedup to be:

$$S_2 = \frac{1}{(1 - p_{float}) + \frac{p_{float}}{s_{float}}} = \frac{1}{(1 - 14.29\%) + \frac{14.29\%}{2}} \approx 1.0769$$

Given that the predicted speedup in Improvement 1 surpasses that of Improvement 2, we will choose the design of Improvement 1.

2. Using simulation results, what is the speed up of each improved case over the baseline design?

The simulation results are shown in Table 6, where Speedup $= \frac{\text{SimTicks(Baseline)}}{\text{SimTicks(Improvement)}}$. Improvement 1 exhibits a speedup of approximately 1.0011 over the baseline design, while Improvement 2 demonstrates a speedup of around 1.0779 compared to the baseline design.

| Case | SimTicks | Speedup |
|---|---|---|
| Baseline | 2,535,109,000 | |
| Improvement 1 | 2,532,277,750 | 1.0011 |
| Improvement 2 | 2,351,911,750 | 1.0779 |

Table 6: Performance of Three Cases

3. If there are any differences between your answer to questions 1 and 2, what do you think could be the reason?

Based on the calculations using Amdahl's law in question 1, Improvement 1 and Improvement 2 are expected to achieve speedups of 1.2728 and 1.0769, respectively. However, the simulation results in question 2 reveal a significant difference in the actual speedup of Improvement 1, which only achieves approximately 1.0011. On the other hand, Improvement 2 achieves a speedup of 1.0779, aligning closely with the anticipated value.

We think the significant difference in the speedup of Improvement 1 can be attributed to the bottleneck in the workload, which primarily lies in float point operations. To further analyze the performance, we examined the instruction pipeline within the DAXPY loop based on the assembly code, as shown in Figure 7, Figure 8, and Figure 9 (Note: only the loop part of the assembly code was considered, excluding some initialization operations).

Given that the program runs on an in-order pipelined CPU with an issue latency of 1 cycle for all cases, we can issue a new instruction every cycle if there are no dependencies between instructions. Additionally, with two integer units available, two integer instructions can be executed simultaneously.

Since we only know the issue latency, integer operation latency, and floating-point operation latency, and lack information about the latency of memory instructions, we can only infer the minimum number of cycles required for each case. For the three scenarios, the minimum cycle counts in the loop are determined to be 23, 20, and 17, respectively. The variable "X" in the figures represents the number of cycles corresponding to memory instructions (which is unknown) and the delay caused by dependencies among different instructions (e.g., fmadd must wait for the completion of the two fld instructions).

Therefore, despite optimizing the integer operation latency by 2, the performance improvement remains limited. This observation highlights a constraint of Amdahl's law, which fails to account for the variability in execution times and the intricate dependencies among instructions within a program.

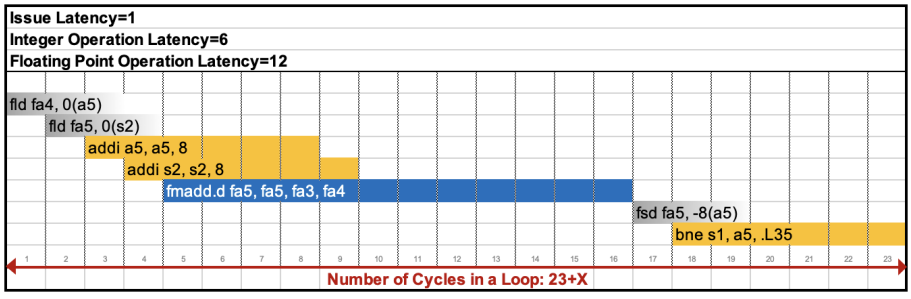


Figure 7: Instruction Pipeline in Baseline Case

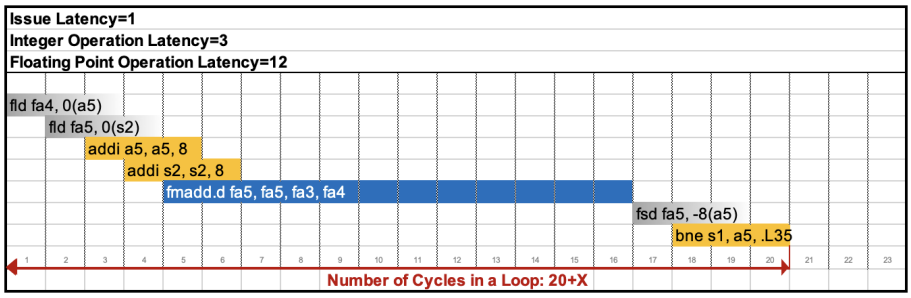


Figure 8: Instruction Pipeline in Improvement 1

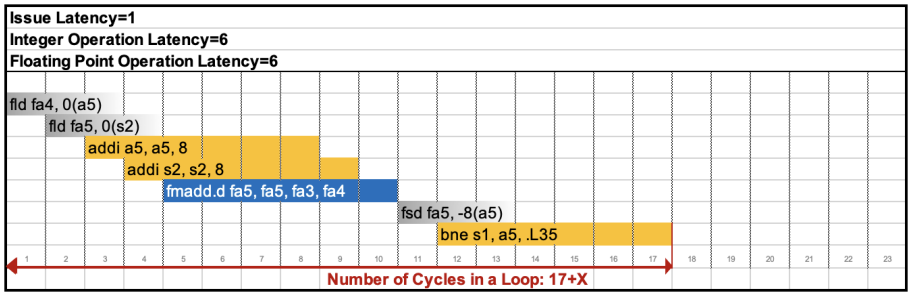


Figure 9: Instruction Pipeline in Improvement 2