Anirban Khara
G40191570

An Efficient Algorithm for Exploiting Multiple Arithmetic Unit

Summary

The research paper focuses on optimizing arithmetic operations, particularly floating-point operations in computer systems. Even after enhancing storage access time and using advanced pipelining, there remains a challenge in ensuring swift arithmetic operations. The paper identifies two primary issues: 1) single operations are not speedy enough for serial execution, and 2) a universal execution unit cannot achieve the fastest times for multiple tasks. A proposed solution divides the execution into two parts: fixed-point and floating-point areas. This division facilitates simultaneous execution but requires a mix of both instruction types for maximum efficiency. The paper's main theme revolves around the concurrent execution of floating-point instructions in the IBM System/360 Model 91. A significant challenge arises due to the need to manage dependencies between instructions, given that there's a single set of accumulators. The Model 91 resolves this through the "common data bus" (CDB) system, which allows maximum task concurrency with minimal programmer or compiler intervention. The CDB offers a simplified hardware algorithm that automatically utilizes multiple execution units, ensuring efficient operation.

Figure 1 specifically offers a detailed view of the data registers and transfer paths in the absence of the CDB implementation. This diagram demonstrates the inherent complexity and interconnectedness of the components that partake in data transfers and arithmetic operations. Key elements depicted include the Storage Bus, Floating-Point Buffers (FLB), Floating-Point Operand Stack (FLOS), and control units. The arrangement showcases how data is channeled through various buffers and registers, ultimately leading to arithmetic units like the adder. The Decoder plays a pivotal role in determining the operations, and the instruction unit governs the overall flow. What becomes apparent is that while this structure is functional, it lacks the streamlined efficiency that could be achieved with the introduction of the CDB. Through the illustration, Tomasulo effectively highlights the challenges in data flow and arithmetic operations, setting the stage for the proposed enhancements with the CDB in the subsequent sections of the paper.

In the context of the Model 91 organization relating to the System/360 architecture, the instruction unit converts both storage-to-register and register-to-register instructions into a pseudo-register-to-register format. Here, R1 consistently represents one of four floating-point registers (FLR) and typically receives the outcome of the operation. The sole exception is store operations where R1 designates the operand's source for storage. The R2 field, as interpreted by the floating-point operation stack (FLOS), has three possible meanings. It can represent an FLR, a floating-point buffer (FLB), or a store data buffer (SDB), depending on the instruction. The instruction unit simplifies storage by mapping it exclusively to 6 floating-point buffers and 3 store data buffers, making all operations appear as pseudo-register-to-register to the FLOS.

The study examines the interpretation of the ADO, 2 instruction, which denotes the double-precision sum of registers 0 and 2. In this configuration, R0 acts as both the source and destination, labeled as the sink, while R2 is termed the source. This suggests a data register system connected by transfer paths. Previous discussions have highlighted primary registers like FLR’s, FLB’s, FLOS, and SDB’s. Two additional registers, connected to each execution circuit, were initially considered essential working registers but later expanded in their roles due to the reservation station concept. The design of machine data paths reveals four primary instruction types: register loading from storage, storage-to-register arithmetic, register-to-register arithmetic, and store operations. Fig. 2 illustrates the timing relationship between an instruction's processing and the FLOS decode. Notably, in storage-to-register arithmetic and register-to-register operations, the FLOS decode allows for a dynamic transfer of data. However, challenges arise in concurrent instruction execution, as demonstrated by a sequence where a load operation can occur, but a subsequent multiply operation faces issues due to concurrency. The pivotal principle here is that a floating-point register shouldn't be engaged if it's the designated sink for an incomplete instruction. The design necessitates

mechanisms to detect dependencies, ensure proper sequencing, and distinguish sequences to maintain integrity and performance. Future sections aim to provide solutions to these concurrency challenges.

The presented content discusses the efficient execution of arithmetic operations in computer systems, particularly emphasizing how to optimize concurrent operations. When a result is stored in the source register, control bits manage its delivery to a waiting unit via the FLR bus. The system's design focuses on meeting requirements that allow simultaneous execution of operations to speed up computation. For instance, an arithmetic expression like A+B+C+D∗E is broken down into a sequence of load and store operations to different registers.

The "busy bit scheme" ensures that multiple operations, such as addition and multiplication, can be performed almost simultaneously, maximizing the efficiency of execution units. However, challenges arise when one operation's result is required for a subsequent operation, causing the latter to wait.

The introduction of "reservation stations" offers a solution. These stations hold operations until the necessary operands are available, ensuring seamless execution. In the Model 91, there are both addition and multiplication/division reservation stations to cater to different arithmetic operations. Furthermore, code optimization by a programmer plays a vital role. While overlapping operations can save cycles, it's also crucial to design loops and sequences thoughtfully. Two independent instruction sequences can be achieved by having them use different sink registers or by initiating the second sequence with a load operation.

Example 3a:
In this sequence, a straightforward approach to evaluating the expression is shown. The operations are linear and have a clear sequence:
1. Load E into F0
2. Multiply F0 by D
3. Add C to F0
4. Add B to F0
5. Add A to F0

However, due to this linear structure, there's no opportunity for overlapping or parallelizing operations. As a result, despite having fewer instructions than Example 2, this program will take six more cycles to execute.

Example 3b:
This example shows the operations inside a loop structure. Within each loop iteration, similar arithmetic operations are carried out as in Example 3a but with different data sets (specified by the index i).

1. In the first loop (LOOP 1), the operations are:
- Load a value from an array element E_i into F0
- Multiply F0 with D_i
- Add C_i , B_i , and A_i in successive steps to F0
- Store the result in F1
- Decrement the loop counter i and repeat if i is still positive

2. In the second loop (LOOP 2), operations are almost identical to LOOP 1 but involve both F0 and F2 registers and incorporate subsequent array elements.

The significance of Example 3b is that it highlights potential for parallelism. Though each iteration of LOOP 1 seems dependent on the previous, the instructions in each iteration are, in fact, independent, allowing potential overlapping of operations across iterations. This demonstrates a method to achieve concurrency in execution and optimize run-time.

The article focuses on the challenges and solutions associated with achieving concurrency in floating-point operations. A critical issue is how to effectively sequence the execution of instructions while also setting results into the floating-point registers. The primary mechanism introduced for handling this challenge is the Common Data Bus (CDB). The CDB serves as a central channel through which data from various units, such as adders and multipliers, is communicated. Significantly, another output port is added to the floating-point buffers, which, when combined with existing units like adders and multiplier/dividers, forms the CDB. This configuration allows the CDB to interact not only with the registers but also with source and sink registers of all reservation stations.

In the proposed system, each unit that contributes to the CDB is uniquely identified by a tag. These tags are essential for maintaining the order and precedence of operations. For instance, when a particular floating-point instruction is decoded, the system checks for a 'busy bit' in the specified registers. If it is not set, the content of the register(s)

is forwarded to the intended unit. Simultaneously, the system updates the 'busy bit' and assigns a tag to the destination register. This tagging system ensures that subsequent instructions can effectively use the results produced by prior operations, without any contention or misalignment.

Furthermore, the introduction of the CDB, combined with the tag system, preserves the order of operations. The priority control within the CDB ensures that results from different units are correctly sequenced and that no two units conflict over the same data. This architecture, represented in Figure 4, effectively demonstrates the complex interplay between various registers, buffers, and arithmetic units, all coordinated by the CDB. By embracing this structure, the system achieves efficient concurrency in floating-point operations, overcoming traditional limitations.

The Common Data Bus (CDB):
The CDB serves as a centralized data communication channel, allowing different computational units to communicate their results. Let's denote the CDB by C. At any given cycle, C can contain the result from any one of its connected units.

Tagging System:
Each computational unit or result holder is assigned a unique identifier or tag. If there are N units contributing to the CDB, a tag can be represented as a binary number of ceil(log_2(N)) \rceil bits. For instance, if there are 11 contributors (as mentioned), then 4 bits would be sufficient to uniquely tag each contributor. Let's represent these tags as T_1, T_2,...., T_N.

Operation Sequencing:
Given an instruction I that involves a source register S and a destination register D, before execution, the system checks:
$B(D)$
If $B(D) = 0$, it means the destination register is not busy and can accept the result. Upon issuing instruction I, the system sets $B(D) = 1$ and assigns a tag to D, say T_k, where k is the identity of the unit executing I.

Precedence:
For preserving the order of operations, the CDB uses its priority control mechanism. When a unit wants to send its result to the CDB, the priority control ensures the CDB is free, then broadcasts the tag of the requesting unit. All active reservation stations compare their tags with the CDB's tag. If there's a match, they take the data from the CDB.

In essence, the entire mechanism operates on a series of checks and updates using the busy bits and tags to ensure correct sequencing and precedence of operations. The mathematical representations and operations above provide a formalized view of the system's workings.

Summary:
The study focuses on the advancement of two concepts vital for high-performance computer design, namely the reservation stations and the Common Data Bus (CDB). The former acts as an efficient buffering method, optimizing execution in scenarios where significant time disparities exist between unit transmissions. With multiple execution units, each often waits for operands, and reservation stations hold these pending operands, letting circuits engage with whichever reservation station becomes available first.

More crucially, the CDB works in tandem with reservation stations. Using a simple tagging system, the CDB maintains operation precedence while simultaneously promoting concurrency. This plays a pivotal role in reducing the vulnerability of the Model 91 to programming exercises. By facilitating the flow of buffering within the CPU, the CDB enhances the overall performance of the system. To illustrate, consider the issue of a program continuously executing without modifying the contents of F0. The CDB's mechanism ensures that only the final iteration will update F0's result. For initiating an independent instruction sequence, two pathways exist – either specify a different sink register or load a new register. The CDB efficiently manages both scenarios, as depicted in Example 6 and Figure 5. Furthermore, the tag of the source register can be transferred to the sink floating-point register, ensuring smooth operation without wasting time or resources.

While it might seem that the CDB introduces an additional execution cycle, practical observations suggest otherwise. For instance, if an operation requires 120 nsec, incorporating the CDB would need 150 nsec. This extra time is analogous to the time required to send results to floating-point registers and receive them as input, even without the CDB's intervention. The utility of the CDB becomes evident when examining a typical partial differential equation

loop, showcasing a potential performance increase. In the absence of the CDB, one iteration might take up to 17 cycles. With the CDB's assistance, the same iteration can be completed in 11 cycles, signifying a performance improvement by approximately one-third.

In conclusion, the research underscores the significance of the reservation stations and the CDB in streamlining high-performance computing. By ensuring efficient buffering and data flow, these innovations play a pivotal role in enhancing execution speeds and improving overall computational efficiency.