Dynamic scheduling of Instructions in Transport Triggered Architecture based Processors

Ashrit Triambak

Department of Electrical and Information Technology

Technical University of Kaiserslautern

This Thesis is submitted for the degree of

Masters of Science

TUKL 2015

Declaration

Herewith I declare that the present thesis has been prepared according to the rules of MER "Elektrotechnik und Informationstechnik" by myself without the help of third parties except from the support of my supervisor, that all used sources and tools including the internet are completely and exactly declared, and that everything is marked which is taken unchanged, shortened or analogous from other literature.

Kaiserslautern, 28th May 2015 Ashrit Triambak

Acknowledgements

First of all, I would like to thank Prof. Dr. Klaus Schneider, Chair of Embedded Systems Group, Department of Computer Science for giving me this opportunity to carry out my Master Thesis in his research group and providing me the necessary facilities to carry out my research. I thank Prof. Dr. Wolgang Kunz, Chair of Electronic Design Automation Group, Department of Electrical and Computer Engineering for mentoring me throughout my studies and giving me the permission to carry out this research. I express my heartfelt gratitude to Mr. Anoop Bhagyanath for supervising me during the thesis. I appreciate his support, calmness and timely response to my queries that helped me to carry out my research smoothly. I also thank the members of Embedded Systems Group, Department of Computer Science who have either helped me directly or indirectly to carry out my research. Finally, I take this opportunity to express my gratitude to my beloved parents, brother and friends for their continuous support and encouragement throughout my studies.

Abstract

Usually the embedded systems have real - time performance requirements. The time predictability of hardware architectures designed for real time applications are of great importance. In modern machines, the processor architectures for high performance computing have been focusing on dynamic scheduling technique to issue and execute multiple instructions concurrently. This technique primarily uses special hardware to analyse the instructions stream during run - time and determine the dependencies of each instruction. On the other hand, static scheduling technique uses the compiler to determine dependencies of an instruction at compile time and schedules it accordingly. This method of static scheduling leads to bad performance when variable latency instructions are involved. Since it is impossible for the compiler to determine the completion time of such an instruction during compilation time, it leads to an inefficient schedule. However, one advantage of static scheduling technique is time - predictability.

Transport triggered architecture is a class of architecture, that exhibits parallelism not only at operation level, but at the data transport level too. This architecture allows for better hardware utilization due to its exposed datapath. The key feature of this architecture is the separation of data operations and data transports. This separation also allows for the machine cycle to be reduced. TTA can generate instructions by specifying the transports instead of operations. Unlike VLIWs, where the operations are packed in a single instruction, TTA packs multiple data transports in a single instruction.

So far, the research works carried out in the instruction scheduling techniques of TTA were based on static method. This means that the compiler packs a few data transports in every clock cycle and these are then executed by the hardware. In case of variable latency functional units, the exact latency is known only during run - time. Therefore the compiler has to schedule such functional units taking into consideration the maximum latency. This leads to an under-utilization of performance. Dynamic scheduling enables the performance utilization of such functional units by scheduling the instructions at run-time. In this thesis, efforts were taken to explore the dynamic scheduling possibilities in this class of processor architecture. We introduced two algorithms, Latency Stalling and Dynamic Scheduler. In the former, the instruction fetch is stalled if a structural hazard is detected. In the latter, we use buffers for

individual functional units to store instructions in FIFO order. When a structural hazard in detected in one of the functional unit, it does not affect the execution in another functional unit. This further improves the hardware utilization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's world, microprocessors exist in every other electronic gadget that we encounter in our daily life. Microprocessors are present not only in workstations and PCs, but are also found in our daily life gadgets like televisions, mobile phone, microwaves, air conditioners, automobiles, toys, automatic vending machines etc. Embedded systems are adopting more sophisticated algorithms and hardware, eventually evolving into more complex systems resulting in the coverage of a broad spectrum of applications due to its high performance computation capabilities.

General purpose processors (GPP) are the processors that are designed for general purpose computers such as PCs or workstations. Sustained high performance with adequate power consumption across a broad spectrum of general-user applications are usually the key requirements in the design of general-purpose processor cores. The computation speed of a GPP is the main concern and the cost of GPP is usually higher than that of a microcontroller or processors that are specific to certain applications only. These are called the application specific processors (ASP). Embedded systems in the present days target a very narrow or even a single application domain which would be benefited from an ASP. These processors use a much simpler microarchitecture which matches well with the characteristics of the targeted application(s) so as to gain performance and to reduce the power consumption.

A GPP which is intended for a large class of applications may contain hardware that is not efficiently used by a given applications and in addition, it may also miss a hardware that may enhance its performance for that application. Let us now consider an example of both the cases. Firstly, a GPP may contain an expensive multiplier when an application performs multiplication operation minimally; say around 0.1% of the total executed operations. Here we can observe that the use of the multiplier hardware is a waste of chip area and the power consumption by this hardware even though it is idle most of the time. Secondly, a GPP may lack some hardware; like a hashing hardware in system which performs hashing of data con-

tinuously. In such a case, we can implement a hashing hardware to accelerate the performance of the processor when such applications are executed.

In the development of a processor, the time-to-market plays a very important role. There are different markets in which various processors are used. Since these markets have different requirements of the processors, the device designed for one market may be inappropriate in the other market in most of the cases. GPPs generate a vast majority of revenue in the design of desktops, laptops and servers those are commonly used for personal use or business purpose. Since these devices are used to run a large set of applications, it is much advanced technically and hence these are relatively costlier and incurs a higher power consumption. The time to market of such processors are usually high. An ASP on the other hand is used in the design of an embedded system sold in many billion volumes every year, but at a lower cost than that of a GPP. For many companies, it is a challenge to introduce high quality ASPs at a low cost and within a short time frame. The design cost pressure in a company also calls for a much simpler design methodology in ASPs, one that uses much more automation in terms of verification, synthesis, placement and routing.

The possibility to overlap the initiation and execution of multiple instructions in parallel within a single processor is referred to as instruction level parallelism (ILP). ILP is made possible due to higher transistor densities which allows for duplication of functional units and data paths. This allows the ILP processors to simultaneously access the duplicated resources, hence improving the performance. Consider the following program

1. a = b + c;

2. d = e+ f;

3. g = a * d;

Since the result of operation 3 depends on operations 1 and 2, it cannot be calculated unless operations 1 and 2 have finished execution. However operation 1 and 2 are not dependent on each other and hence they can be executed simultaneously.

The exploitation of ILP consists of mapping the application onto the target architecture as efficiently as possible. This mapping consists of scheduling and resource allocation which can be done either in hardware or software. There exist various architectures that take advantage of this kind of parallelism. A super-pipelined machine is the one that can issue one or more instructions per clock cycle but the cycle time is much smaller than the typical instruction latency. A superscalar machine is the one which has the capabilities to issue multiple independent instructions in the same cycle.

The exploitation of large ILP requires a large amount of connectivity between the shared register file and the Functional Units (FUs). This connectivity is expensive and increases the cycle time.

A processors performance can be measured in average time per instruction (TPI). The TPI is the product of the cycle time of the processor and the cycles per instruction (CPI). To increase the performance of a processor we can either decrease the CPI or cycle time. The techniques that are known to decrease the CPI are, e.g., pipelining, out-of-order execution, branch prediction, or super-scalar designs.

In the following section, we shall look at two ways to exploit the ILP

1.1 Processor Architectures

1.1.1 Statically Scheduled Architectures

In this architecture, the instruction level parallelism is usually extracted statically. This means that the scheduling of instructions and the dependency of instructions are analysed during the compile time by the compiler. The compilers are responsible for explicitly specifying which instructions are issued and executed in parallel. The hardware is generally simple and there exists no overhead during run time to schedule the instructions.

The primary goal of static scheduling technique is to determine the set of operations that have their source operands available and have no dependencies within the determined set. These operations can be then scheduled within the same instruction subjected to the available hardware resources. Since these operations in the instructions are guaranteed by the compiler to be independent, the hardware has to just issue and execute these instructions without any dynamic analysis. The multiple operations that are packed as a single instruction leads to a very long instruction size in comparison with a traditional single operation instruction. These instructions are known as Very Long Instruction Word (VLIW) and the processor using this technique is known as VLIW processors. Here, it is important to distinguish between the operation and instruction. An operation is a simple computation such as addition, memory load, branch etc. whereas an instruction consists of multiple operations. The operations in an instruction are issued simultaneously.

VLIW processor is an example of independence architecture which contains information regarding the independence between the operations. A VLIW machine is like a superscalar machine, but the parallel instructions must be explicitly packed into very long instruction word by the compiler. To fully exploit the characteristics of a VLIW machine, there must be n instructions concurrently executing at all the times. If such parallelism is not available,

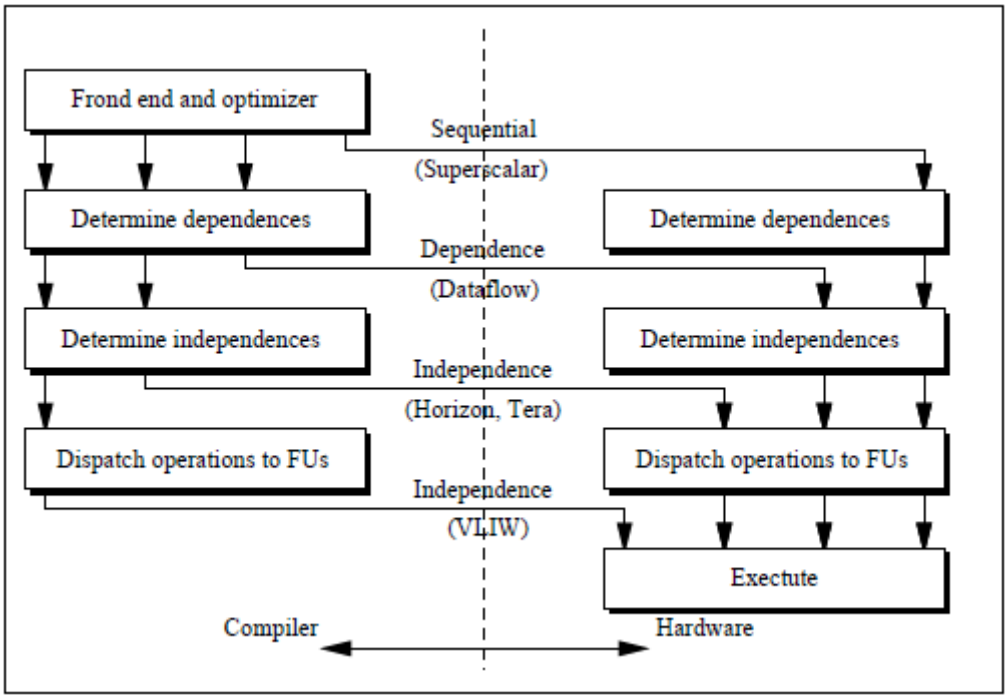


Figure 1.1: Compiler Vs Hardware Tasks

then stalls and dead time occurs which forces the instruction to wait for the results of previous instruction. For this reason, the code must be ordered carefully to utilize the performance potential to the fullest. A VLIW compiler not only releases the hardware to detect any data dependencies but also assigns the FUs to the operations. A VLIW program specifies when each operation must be issued and which FU must execute the issued operation.

The compiler plays a very important role to enhance the static scheduling performance. The compiler in fact decides the order of execution of operations by taking care of the data dependencies and resource constraints. To accomplish this constraint, a detailed description of the processor is used. This means that the compiler must exactly know about the number of instructions that can be performed in parallel. .

One of the compiler technique that was developed to increase the the compilers ability to move instruction across basic block boundaries was the trace scheduling technique. The compilation is done by selecting a likely path of execution called a trace. It creates a single long block of code from the individual basic blocks without hardware support. Instead of using the individual basic blocks, this long block is scheduled so as to achieve a greater level of concurrency. The overhead and difficulties involved in the saving and restoring the states for the instruction with its large amount of run time branches makes the trace scheduling an unattractive choice.

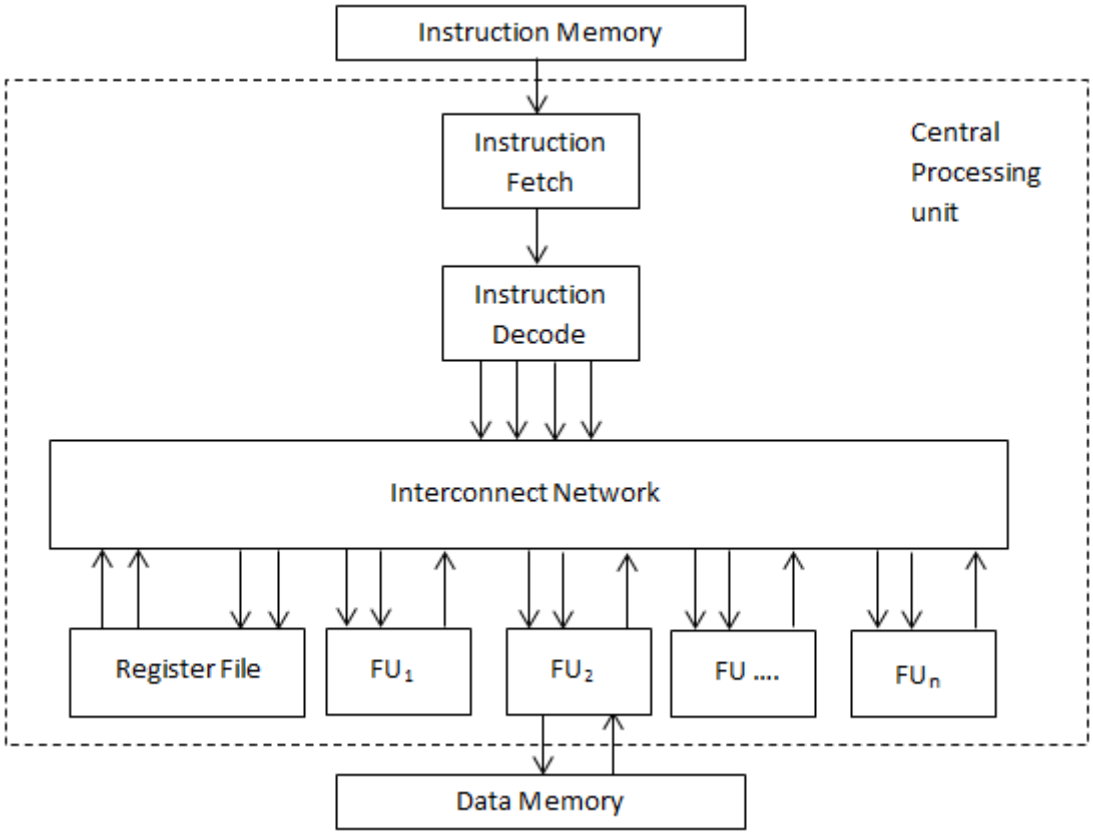


Figure 1.2: Block diagram of TTA

Figure 1.1 shows the distribution of task between various classes of processor architectures. A VLIW processor just executes an instruction that has been issued by the compiler and then produces the result. A static scheduler often has ambiguous memory references. A pair of memory references where one of the two references is a store can only be reordered when the scheduler can guarantee that the two references access different memory locations [10]. This is a problem of static scheduler when it does not have sufficient information regarding the two references. Another problem faced is the branch prediction. The dynamic branch prediction algorithms are usually more accurate than branch prediction algorithms based on static analysis.

Transport Triggered Architectures (TTAs) are similar to VLIW architectures. They form a super class of traditional VLIW architectures i.e. they not only exploit operation level parallelism but also parallelism available at data transport level. The main idea of transport triggering is to have more information about what is going on within the central processing unit [3]. This change from operation to transport triggering is similar to the changes from CISC to RISC architectures. The application of RISC principles simplifies the design and reduced the

instruction set. Similarly the transport triggering reduces the number of instructions to a single data transport and simplifies the design further. Figure 1.2 shows the basic block diagram of a TTA.

A TTA consists of a number of Functional Units (FUs) and register files (RFs). Each functional unit is connected to the interconnection network with one or more sockets. Basically the network need not be fully connected. This eventually reduces the code generation process. Each FU of TTA includes one or more operand and result registers. These registers are distinguished by their purposes and are defined as operand registers, trigger registers and result registers. TTAs gives the compiler even greater responsibilities, since now the transport resources also needs to be assigned apart from the FUs and RFs

The main advantages of TTAs are its flexibility and its simple architecture. It allows short processor cycle time and a quick processor design. The transport triggering results in a better utilization of transport network. Extra fine grain parallelism is achieved as due to the separation of ALU functionality into its individual components.

1.1.2 Dynamically Scheduled Architectures

As the name implies, dynamic scheduling is a method in which the hardware determines the execution order of instructions in contrast to the statically scheduled machines where the compiler determines the order of execution. Dynamically scheduled machines take advantage of parallelism which is not visible at compile time. They are also more versatile as the code does not necessarily be recompiled since the hardware takes care of much of the scheduling.

The simplest implementation of a processor is a single cycle implementation. This means that each instruction is executed in one clock cycle. This implementation is very slow and inefficient. This is because the instructions are generally not of the same latency. The clock cycle must be of the same length for every instruction in the single cycle implementation. The clock cycle is hence determined by the longest possible path in the processor. This longest path would certainly be a load/store instruction. Because of this assumption that the clock cycle is equal to the worst case delay of the instructions, the worst case execution time of the program does not improve. In the earlier architectures with a very simple instruction set, the use of single cycle implementation was not a problem. However, if a floating point unit or an instruction set with more complex instructions were used, then the single cycle implementation would not work well.

Another implementation technique called pipelining uses a very similar datapath compared to the single cycle datapath, but this is much more efficient by having a much higher throughput. In this technique, multiple instructions are executed in parallel. Pipelining increases the number of instructions that are executed in parallel. It has to be noted that pipelining does not

reduce the time it takes to complete the individual instruction which is also called latency, but it improves the instruction throughput.

1.2 WCET Analysis

Worst Case Execution Time (WCET) analysis is the research field that investigates the methods to access the timing behaviour of the real time systems. To understand this, let us first discuss about real time systems.

A real time system is a computer based system that reacts to the events in the environment by performing predefined actions within specific time intervals. It is usually a wrong assumption that a real time system should produce result as soon as possible. Rather, it should possess a steady and predictable behaviour. For example, consider the video and audio playback of a recording. The steady image generation with the correct synchronization of audio at a pace consistent with the recording speed is important. Being too fast is obviously not preferable, but playing the video too slow is not good either. The most important thing is to adjust to the situation. In other words, correct synchronization must be maintained.

Real time systems are of two types - Soft and Hard real time systems. In soft real time systems, an occasional failure to meet a deadline does not have a permanent damaging effect. Eg. Video playback. In hard real time system, the failure to meet a deadline can be fatal. Eg. Braking a car too late.

To guarantee the behaviour of a hard real time system, we must know the WCET of the system that has to be analysed and accounted for. If there are a number of concurrent programs running in a system, it has to be proven that each program can meet their respective deadline even in the case when all the programs are simultaneously executing at their maximum workload.

The validation of the timing of a real time system has to be made at the system level. This means that each and every module or component of a real time system must fulfil their timing requirements. Only then we can be sure that the complete system meets the requirements. When there are a number of software programs running in an embedded system, we have to analyse the timing behaviour of all the programs running in the system. Knowing the execution time properties of the code is an important part of real time system development and failing to do so can lead to system failure [6].

A program having high execution time variation can still be considered as predictable if we can model and predict the cause of variation in execution time. The WCET of a program is highly dependent on the control flow like the decision statements, function call, loop iteration etc and on the characteristics of a processor (like pipeline and caches). Hence these parameters

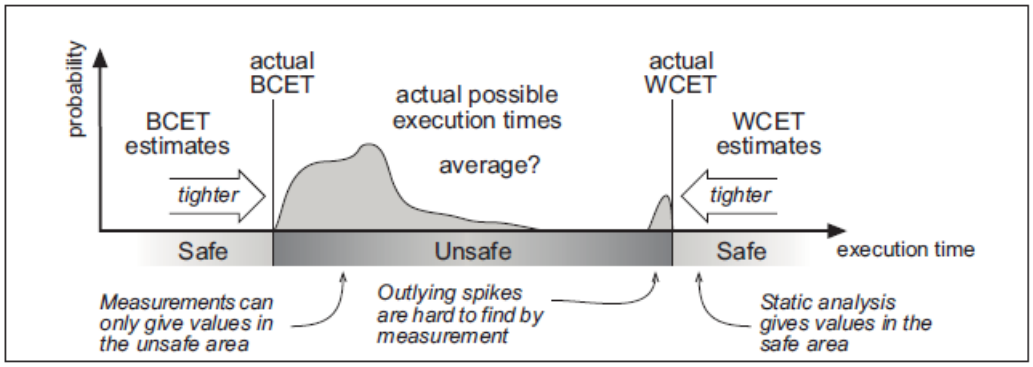


Figure 1.3: Estimation of Execution Time

are to be considered separately during the WCET analysis. Typically the control flow of a program can be modelled with reasonable precision but the hardware can pose a very big problem and give rise to actual unpredictability [6]. Real time systems involving embedded computers have to be analysed as a combination of hardware and software. To understand the predictability of the programs running on a system, both the properties of hardware and software have to analysed. This property will be discussed in the further sections.

Further we will see that the work carried out in this thesis has considered the problems that a hardware can pose to the WCET analysis and then a suitable algorithm with a possible solution was designed and implemented.

1.2.1 Understanding Execution Time

The different execution time measures that can be used to describe the timing behaviour of the program are worst case execution time (WCET) or best case execution time (BCET). WCET is the longest execution time of the program that will be observed when the program runs. The BCET is the shortest time that program will ever take to execute. The average execution time is the average which lies in between the BCET and WCET. Usually it is very hard to find the exact value of WCET and BCET since it depends upon the inputs received at run time. Further it is even more difficult to determine the average execution time since it depends on the distribution if the input data and not just the extremes of program behaviour[6].

In Figure 1.3 we can see the estimates of execution time. The curve shows the probability distribution of the program execution time. The aim of timing analysis is to produce estimates of WCET and BCET. The safe estimation of timing is very trivial. The WCET estimate must be greater than, or, in the ideal case equal to the actual WCET. Similarly, the BCET estimate has to be less than or in the ideal case equal to the actual BCET. The under-estimation of WCET is equivalent to no WCET estimate since the results produced will rest on a false

assumption which we believe to be correct, but actually it fails.

1.2.2 Influence of Processor Architectures on WCET Analysis

The architectures of tools for the determination of WCET as well as the precision of the WCET analysis strongly depends on the architecture of the employed processor [8]. The central part of analysing the behaviour of the execution time is modelling the timing behaviour of the involved hardware within the analysis process. With the use of a number of well-established WCET analysis techniques, one can carry out the analysis of simple hardware architectures. For highly dependable real time systems, the estimation of the upper bound on the execution time is trivial. WCET is often grossly overestimated because of the pessimistic timing assumptions. This results in poor utilization of resources particularly in the real time systems using high processor performance with caches and advanced pipeline techniques.

The modern microprocessor architectures uses a number of performance enhancement mechanisms such as caches, pipelining and speculative execution. Caches are mainly used to decrease the access time to main memory. Pipelining is a mechanism that enables acceleration of execution of different instructions in parallel. Speculative execution is basically used to avoid the stalling of pipeline caused due to conditional jumps. Due to these features, the execution behaviour of the instructions cannot be analysed independently since it depends on the execution history. The architectures of various processors are optimized for average case performance and not for predictable performance, as would be required for hard real time systems [8].

The WCET of a program is composed of the following tasks

- Classification of memory references as either cache hits or cache misses. This is usually called the cache analysis
- Analysis of the behaviour of the program on the processor pipeline. This is called pipeline analysis
- Prediction of the results of the speculative execution (associated with the control flow) followed by the determination of the WCET of the path. This is called path analysis.

For modern microprocessors and DSPs, the above mentioned tasks are quite complex. Since the semantics of the high level language does not refer to the architectural components such as caches, pipelines or branch prediction, the analysis must be performed on the machine code level. The data cache analysis and the pipeline analysis depends on the knowledge of the effective addresses which is in general known during run time [8].

The WCET analysis is divided into a number of steps that models the timing behaviours of the sub-systems of the processors separately. These results that are partially computed are further combined to compute the overall WCET for the code under consideration. It is important to understand that the divide and conquer strategy will only work if the partial results can be safely combined to yield a total result. The instruction schedule depends on the execution time of each individual instruction. The scheduling of further instructions can cause a counter intuitive increase or decrease in the execution time of the rest of the execution path. To find a safe estimate of the WCET in the presence of such anomalies, the effect of all possible schedules resulting from a variable latency instruction has to be analysed so that the instruction latency that leads to the longest overall execution time is obtained.

In section 1.2.4 we shall discuss the examples of the timing anomalies present in dynamically scheduled processors. The term dynamically scheduled processors is often used to describe a processor for which instructions execute out of program order. We know that the execution time of an instruction can take one of many discrete values depending on the input data. For example the execution time of load instruction depends on whether the address hits or misses in the cache. Another example can be an arithmetic instruction whose execution time may depend on the operands. In the following sections we will use the term latency as the instruction execution time. Execution time is the overall execution time of the program.

1.2.3 Effects of Timing Anomalies

1.2.3.1 General Terminologies

Timing Anomaly

The term ‘timing anomaly’ is used to describe the overall system behaviour where relaxing some constraints can lead to an increase of system timing. This is typically caused due to a greedy scheduler that cannot foresee the future impact of its local decisions. For example, with respect to the worst case execution time analysis, such a constraint may require execution of two instruction sequences to finish within a given deadline. Decreasing the execution time of the first instruction sequence relaxes the constraint for the second instruction sequence to finish within the deadline which can lead to timing anomalies [23]. An anomaly never exists alone i.e. it is necessarily embedded in some context. The context of timing anomalies is the set of WCET analysis methods.

Multiple Issue Processor

A multiple Issue processor, also known as superscalar processor is characterized by the following properties.

1. The pipeline in a superscalar processor includes all features of a classical pipeline, but furthermore, the instructions may be executed simultaneously in the same pipeline stage

2. The execution of instructions can be initiated simultaneously in one clock cycle. The instructions are dynamically scheduled i.e. the actual instruction grouping is done in runtime, in contrast to the VLIW architectures.

Dispatch

The term dispatch refers to the primary distribution of the instructions among the particular subsystems of functional units and the term issue refers to the assignment of an instruction to a particular functional unit for some immediate execution.

Resources

The resources are mainly divided into two types. The first one is the in-order resources like the registers in which the resources can be allocated in program order of execution. The second type is the out-of-order resources such as functional units, in which a new instruction can use a resource before an older instruction can use it according to some dynamic scheduling decision.

1.2.3.2 Related Work and Limitations

i. Lundqvist and Per Stenstrom were one of the first persons to discover the term timing anomaly which caused an abnormal timing behaviour when the modern processor hardwares are used. They found out that the presence of out of order resources caused timing anomaly .[15]

ii. A worst case execution time analysis method was developed by Schneider for the Motorolla PowerPC 755 architecture that handle timing anomalies occurring in that specific architecture. [15]

Limitations

For correct estimation of WCET, the effects of all variations in instruction execution times on all possible instruction schedules have to be considered. The problems that arise when performing accurate pipeline analysis for dynamically scheduled processors and how the previous methods failed are explained below. Consider the following definitions

Definition 1

The current pipeline state is the current state of the pipeline timing model. It describes which instructions are currently executing in the pipeline and the current resource allocations.

Definition 2

The current cache state is the current content of the cache timing model. It consists of the cache tag memory, i.e., the identification tags of the current blocks in the cache.

Consider that a program contains a single feasible path. The WCET is the longest execution time of the instruction sequence along this path. If a sequence contains n variable latency instructions with unknown latencies, and we know that each instruction can have k different latencies, then to be safe we have to examine k*n instruction schedules which is not feasible. Normally the timing analysis methods deal with making safe decisions at instruction or basic block level.

Consider a partial sequence of instructions i.e. a basic block containing a variable latency instruction. When the simulation of the execution of this partial sequence in the pipeline is done, we may have k different pipeline states. For safety, we must choose that pipeline state that will give us the longest overall execution time. This becomes impossible without the knowledge of the whole instruction sequence.

The previous methods presented for doing cache and pipeline analysis performs the later by looking at the instructions or the basic block first, and then combines the WCET of all these entities into a total WCET for the whole program. This was however not the best way to obtain the overall longest time of execution. Whenever it was not possible to classify a cache access as a miss or a hit, it was by default assumed to be a cache miss. This would lead to a too pessimistic estimation based on the example presented in the previous section.

Next, consider a program containing several feasible paths. The WCET is the maximum WCET found among all the paths. In order to find the maximum WCET, we would have to examine all the paths in the program.

1.2.3.3 Classification

Scheduling Timing Anomalies

In this type of timing anomalies, the length of the pivot task in two task sets is compared. Let us take an example of a cache hit vs. a cache miss. In Figure 1.4, the task set differs only in length of task A. This kind of timing anomaly has been extensively studied on different scheduling routines. The scheduling depends on the length of task A as it can be seen in the figure. A greedy scheduler is unable to prevent such anomalies in general.[17]

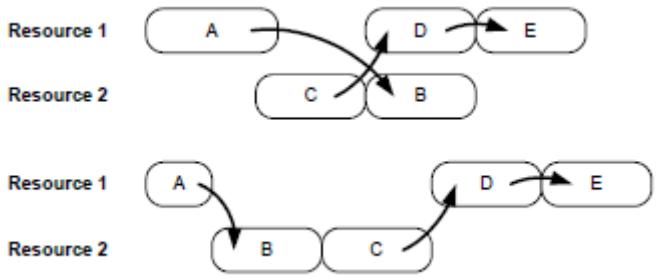


Figure 1.4: Scheduling Anomaly

Speculation Timing Anomalies

In this type of timing anomaly, the entire task set changes depending on the pivot task and not just confined to its length . The example in Figure 1.5. shows that in both cases, the processor pre fetches the instructions. A cache miss while pre fetching the first instruction, which can be the local worst case, takes so much time that the branch condition can be evaluated before more harm to the cache can be done by further pre-fetches.[17]

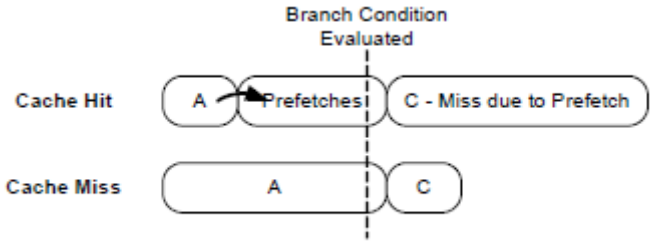


Figure 1.5: Speculation Anomaly

Cache Timing Anomalies

These timing anomalies occur due to abnormal cache behaviour. Taking an example of the ColdFire 5307 processor, where the non-local worst case cache hit results in a different future cache state than the local worst case cache miss. This difference in the cache state can be the cause of the cache hit branch to be stalled later on.[17]

1.2.4 Example of timing anomalies

To model the instruction execution in a pipelined processor, a hardware model is used most often. In this model whenever an instruction that proceeds through a pipeline gets stalled it is due to resource contention with another instruction that accesses a common resource or operand. The examples of resources are functional units and registers. The read and write ports, buses and buffers are treated as resources if they can cause an instruction to stall.

If a processor only contains in-order resources, then no timing anomalies can occur [15]. This is because if there are only in-order resources, two instructions can use a resource in program order. If the completion of an instruction is postponed by 'x' cycles, then the later instructions are also postponed as the resources cannot be allocated before the earlier instructions complete the execution.

If out of order resources are present, timing anomalies can occur. The timing anomalies presented below are studied on a simplified PowerPC architecture (Figure 1.6) containing no Floating point units. The architecture consists of a multiple issue pipeline, capable of dispatching two instructions each clock cycle and separate instruction and data caches. Each functional unit has two reservation stations to implement out-of-order execution of instructions. These can hold dispatched instructions before their operands are available. All resources in the processor are in-order resources except the Integer Unit (IU) and the multiple cycle integer unit (MCIU) which are out of order resources.

Figure 1.7 shows the example of a timing anomaly due to the presence of out of order resources. The table shows when each functional unit is busy executing an instruction. The horizontal dashed lines shows when the reservation stations are occupied. At the top, the arrows indicate when each instruction is dispatched to the reservation stations. Here we can think of two cases, one when the load address hits in the data cache and another when it misses the data cache. If the load address hits in the cache then the LD instruction executes for 2 cycles and then it can forward its result to instruction B that can start executing in cycle 3. In this, it is assumed that B gets priority over C since B is older. Likewise, if the load address misses in the cache then the LD instruction executes for 10 cycles and the execution of B is postponed. This means that C can start executing in cycle 3, one cycle earlier than in

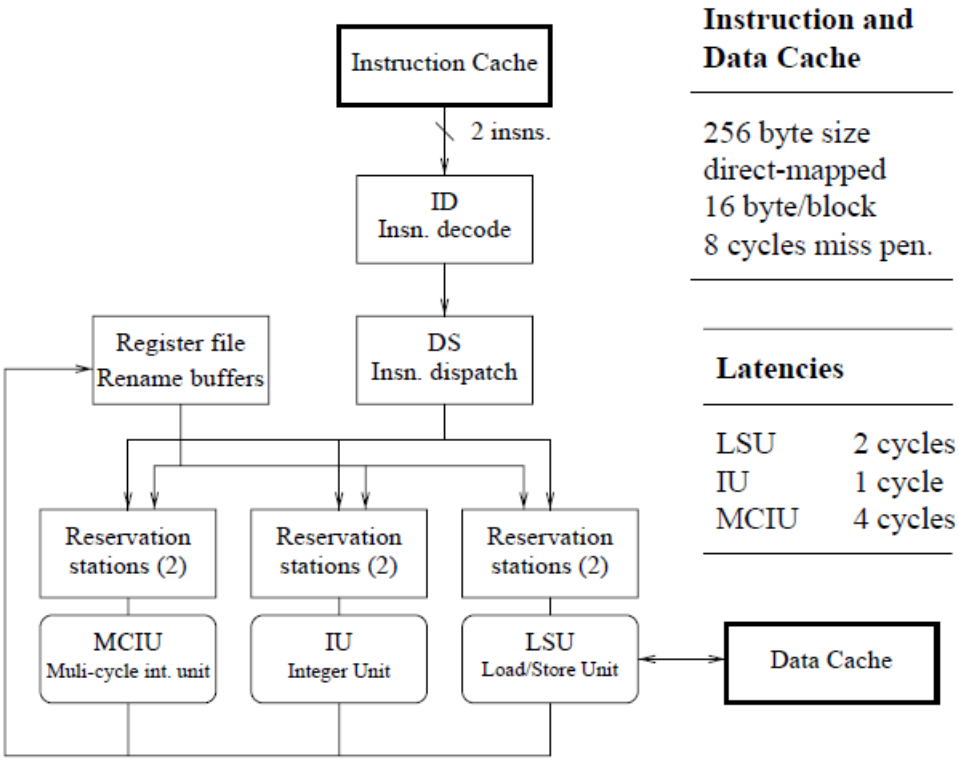
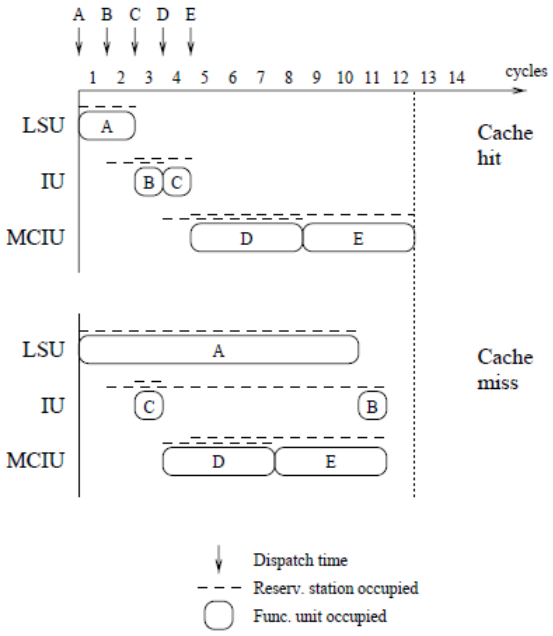


Figure 1.6: A simplified yet timing anomalous PowerPC architecture



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	ADD	r5, r4, r4
C	3	ADD	r11, r10, r10
D	4	MUL	r12, r11, r11
E	5	MUL	r13, r12, r12

Figure 1.7: Example of Cache hit causing a longer execution time than a cache miss

the cache hit case. This in turn makes D and E to execute one cycle earlier. Hence there is an overall reduction of the execution time by 1 cycle in the case when cache misses. Here the anomaly is due to the fact that the unit being an out-of-order resource permitting B and C to execute out of order.

In this thesis we design a dynamic scheduling algorithm for TTA by considering the effects of timing anomalies and presence of in - order and out - of - order resources. The design and implementation will be discussed in Chapter 3 and Chapter 4 respectively.

1.3 Motivation

In today's world the performance of microprocessors are increasing rapidly due to the demand to execute more complex and larger applications over and over again. The design of embedded systems requires high performance and energy efficiency. With increasing demand for application specific processors (ASIP), the time to market requirements and greater demand for processing speed are the challenges that designers are facing to create more efficient and reliable design in less time. Essentially an ASIP can be used for platform based designs which increases the design flexibility and programmability. Usually the embedded systems have real - time performance requirements. A deadline miss in hard real time system can be catastrophic where as in a soft real time system, a deadline miss can lead to significant losses.

Hence, the time predictability of hardware architectures designed for real time applications are of great importance. Transport Triggered Architectures (TTAs) represents a customizable processor template that can be easily customized to contain only the needed resources that can efficiently compute the given application.

Scheduling of instructions plays an important role in the performance of a processor. The instruction level parallelism can be achieved either by static or dynamic scheduling. Statically scheduled processors exploit the instruction level parallelism during the compile time. The primary goal of static scheduling technique is to determine the set of operations that have their source operands available and they have no dependencies within the determined set. These operations can be then scheduled within the same instruction subjected to the availability of hardware resources. In recent trends, high performance computer architectures have employed dynamic scheduling techniques to issue and execute multiple operations concurrently. The primary goal of dynamic scheduling is the use of special hardware to analyse the instruction stream during run time and determine the scheduling sequence of the instructions based on data dependencies during run time. The features like caches, out of order execution, branch prediction and pipelining are employed in modern processors for performance enhancements.

Statically scheduled processors however require that the latencies of all the operations to be fixed and known in advance. In static scheduling techniques, usually the maximum execution time is considered and therefore the performance is not fully utilized. If variable latency functional units are present in the hardware architecture, then it becomes difficult to predict the maximum execution time and hence it may lead to the assumption of ambiguous values. The presence of dynamic scheduling hardware further enables the code generation technique to be simple and hence the length of the program can be sufficiently decreased since the ambiguous run time assumptions can be neglected. However, one advantage of static scheduling technique is time - predictability. So far, the research works carried out in the instruction scheduling techniques of TTA were based on static method. The compiler packs a few data transports in every clock cycle which is further executed by the hardware. The advantages rendered by dynamic scheduling techniques in superscalar processors forms the base to carry out our first research in the field of dynamic scheduling techniques in TTA. We intend to utilize the run - time information for performance improvement and maintain the time predictability at the same time. We will further discuss about the performance enhancement achieved in dynamic scheduling of TTAs compared to the statically scheduled ones.

Chapter 2

Exploring TTAs

2.1 VLIW and TTA

The software approaches used in both VLIW and TTA are similar. In other words, we say that both exploit ILP at compile time [13]. The basic idea of VLIW is to determine the program execution schedule at the time of compilation. The task of the scheduler is to find out the dependencies between various instructions and then determine which operations can be executed simultaneously and by which part of the processor. An example of data path of a VLIW is given in Figure 2.1. In VLIW architecture, the functional units (FUs) are connected individually to the register file (RF). Usually to perform an operation, a FU reads two values (operands) stored in registers and then manipulates them to produce a single result and store back to a register. Consequently, the RF of a VLIW with K FUs must have 3K ports: 2K read ports and K write ports [13].

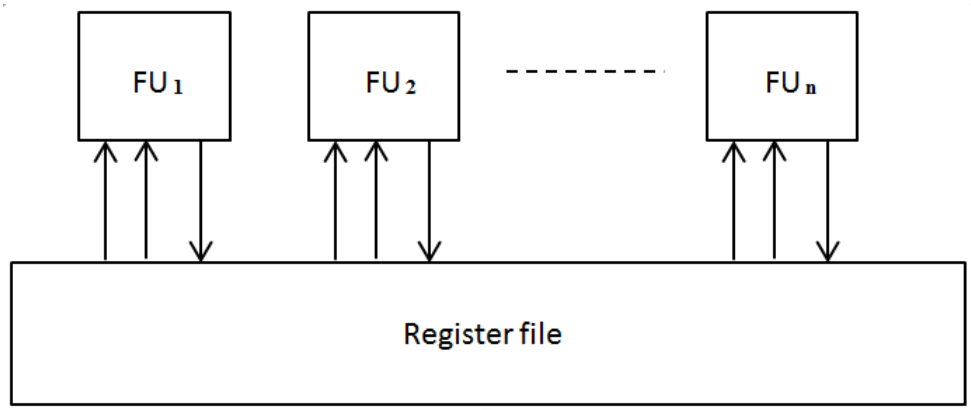


Figure 2.1: Block diagram of a VLIW processor

When the number of FUs (K) becomes larger, the 3k buses and ports become more expen-

sive in terms of power consumption and chip area. In short, even though the organization of VLIW is simple, the communication requirements are high and the communication network is under-utilized.[3]This forms the motivation to change the programming paradigm from operation triggered to transport triggered. TTAs are programmed by specifying the transport that may trigger the operations, instead of operations that trigger the transport whereas the Operation Triggered Architectures (OTA) are programmed by specifying the operations. Each specified operation results in a few data-transports on the data path. For example a subtraction results in two data transports of the operands to the ALU and one data-transport of the result from the ALU.

In traditional processor architectures like RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) the processed code is completely sequential and the processor check which instructions can be executed in parallel without changing the program semantics. Thereafter it schedules the instructions dynamically. This needs detailed analysis in the hardware and hence uses significant power. In VLIW, since the execution order of the instructions is decided by the compiler, the processor itself does not need to contain complex hardware to perform the runtime scheduling (also called dynamic scheduling). As a result, VLIW offers significant reduction in computation power with less hardware complexities. Some VLIWs have more FUs than the permitted instruction size. It is not possible for these VLIWs to keep all its FUs busy, assuming that all FUs are single cycle pipelined. The addition of a bypassing network to a VLIW may result in a better performance but decrease the utilization of the data path even further [13]. Since the data path of VLIW is rarely used for 100%, it is logical to share the transport capabilities on one FU with other FUs. This decreases the number of ports on RF and also improves the utilization of the data path. Transport Triggered Architectures (TTAs) resembles VLIW. TTAs form a super class of traditional VLIW architectures, in the sense that they not only exploit parallelism in the operation level, but also the parallelism available at data transport level. TTAs can generate instructions by specifying the transports instead of operations. TTAs must also use an extra control to ensure that no two transports use the same connection at the same time.

2.2 Transport Triggered Architecture

2.2.1 Principle

Transport triggered architecture (TTA) forms a super class of traditional VLIW architecture. This class of processor has the capability to exploit not only operation level parallelism but also the parallelism available at data transport level. The basic block diagram of a TTA is

shown in Figure 2.2. The TTA implementation primarily consists of a register file (RF) and a set of functional units (FUs) coupled via the interconnect network. The RFs provide temporary storage, the interconnect network performs the transport between the FUs and RFs and the FUs performs the operations. The FUs are usually the simplest building blocks. An ALU for example can be divided into multiple FUs that can perform addition, subtraction, multiplication, division separately. Further, to perform the memory access we have a dedicated FU called the LOAD/ STORE unit. An FU can also perform any arbitrary computation like hashing, signal processing etc. These FUs and RF are connected by means of interconnect network. Transport triggered architecture requires fewer ports on the shared register file than traditional operation triggered architecture [11]. This is achieved by programming data transports instead of operations.

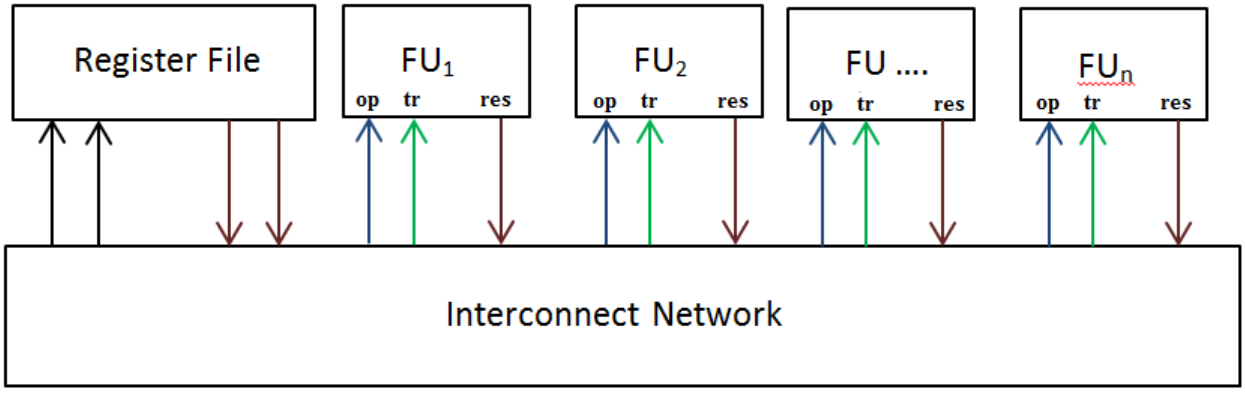


Figure 2.2: Block Diagram of TTA

2.2.2 Instruction Format

In the traditional architectures the programming is done by specifying the operations. The data transports between the different functional units and register files are implicitly triggered by the execution of the specified operations. Hence the traditional architectures are known as Operation Triggered Architectures. On the other hand, programming of TTA shows much resemblance with programming VLIWs. TTAs are programmed by specifying the data transports which in-turn performs the required operations. Unlike VLIWs, where the operations are packed in a single instruction, the TTAs pack multiple data transports in a single instruction. Figure 2.3 shows the general instruction format of TTA. The individual data move controls a bus directly. Hence for N move buses, one can specify N data moves in a single instruction. The data moves also called the data transport describes the transfer of data between the source and the destination as specified in the source ID and destination ID of a data move slot. The source and destination IDs specify the registers. The different registers can be general purpose

registers, operand registers, trigger registers and result registers. Source ID need not always be a register. It can also contain immediate values that are specified in the instruction itself. Special flags can be used to specify whether the source is referred to a register operand or an immediate value.

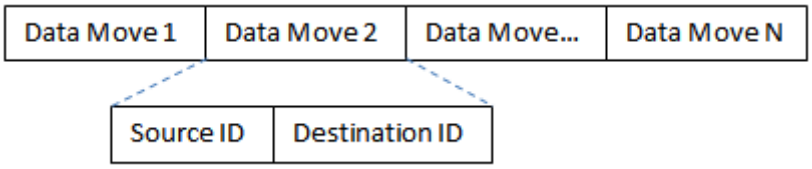


Figure 2.3: General Instruction Format of TTA with N move bus

2.2.3 Functional Units and Registers

Functional Units or FUs are the modules of TTA that performs the computations. The various tasks performed by the FUs can be addition, subtraction, multiplication, division, load and stores to memory etc. An FU can perform any arbitrary functionality and contains one or more input and output registers. These registers are called operand registers, trigger registers and result registers.

Operand Registers

These are the registers that are used to provide the operands to the FUs which can then execute operations.

Trigger Registers

These are special kind of registers where a value is moved to initiate (trigger) the computation of an operation. This value is usually one of the operands of the instruction.

Result Registers

These are the registers of the FUs where the results of the computed operations are placed.

General Purpose Registers (GPRs)

These are the fastest accessible components in memory hierarchy used to store small set of variables temporarily. These are usually used to bridge the gap between the main memory speed and the speed at which the FUs can compute the data. This leads to the speed up in the

execution time. Generally the registers are divided into register files (RF) which has a number of ports using which the registers are accessed. However, by increasing the number of ports, the chip area, the power consumption and the access time also increases [13].

Example of TTA programming

To understand the programming principle better, let us consider an example code written in assembly. Further, these instructions are translated into the respective data move format.

```
add r4, r5, r6          // r4 = r5 + r6
sub r1, r2, r3          // r1 = r2 - r3
```

Now let us translate the above operations into the respective move operations. The clock cycles in which a particular move is scheduled is depected on the left hand side.

1. r5 -> add_op ; r6 -> add_tr ;
2. add_res -> r4;
3. r2 -> sub_op ; r3 -> sub_tr ;
4. sub_res -> r1

In the data moves mentioned above, the suffixes _op, _tr, _res represents the operand, trigger, result registers respectively. In the add operation, the value of R5 and R6 are moved to the operand and trigger registers, add_op and add_tr registers respectively. After the result is computed the adder moves the result from the result register add_res to R4 in the next cycle. The delay that is present between the trigger move and the result move is equal to the latency of the FU that executes the operation. If the result move is scheduled earlier than the delay present in the FU, it will lead to incorrect results being moved.

The operations that are having longer latency than a single cycle are subjected to pipelining. The first stage of the pipeline corresponds to the operand and trigger registers and the last stage of the pipeline corresponds to the result registers. It is obvious that having both operands and the trigger registers at the input and the result register takes at least two cycles to perform the operation. If a trigger value to a FU starts the computation in time unit T, the result is available at time unit T+1 in the result register, then the result can be used in time unit T+2.

There exists several pipelining alternatives for the FUs. The two most useful alternatives are the hybrid pipelines and the virtual time latching pipelines.

Hybrid Pipelining

This method ensures that the data in the pipeline is never overwritten. This is accomplished by attaching a valid bit to the trigger register, the result register and each intermediate pipeline stage. This means that when a particular result is not read from the result register, then the other operation that starts one cycle later does not overwrite the old result. An exception will be raised when an attempt is made to overwrite a trigger register of a hybrid pipelined FU because it cannot accept the operation since the pipeline is full. Similarly the processor locks when an attempt is made to read from the result register which contains no valid data. This lock is released when the result register receives a valid data from the previous pipeline stage.

Virtual Time Latching Pipelines

This method does not have the concept of valid operations contained in a stage. The operations executed in the FU proceed from one stage to the next stage unconditionally. This means that the value of a result register can be overwritten even when the older value was not even read. The duration of availability of this result depends on how soon after the cycle T1, the FU is triggered for the next operation. If the next operation is triggered at time T2 (T2>T1), the result of the operation triggered at time T1 is available from cycle T1+N until T2+N-1.

2.2.4 Immediates

Immediates are the data values that are specified as a part of the instruction itself and does not originate from the registers. Immediates are usually divided into short and long immediates.

Short Immediates

These are provided by TTAs by storing them in the source field of the move slot. Further an immediate bit is added to this so as to identify whether the source field is an immediate or a register operand. The size of the immediate is equal to the size of the source specifier. It is therefore not more than 8 bits usually.

Long Immediates

These are the immediates that do not fit in the source id of a move and hence has to be handled differently. The compiler must have provisions to encode the long immediates in the instructions itself. This can be achieved by the addition of one or more immediate fields in the instruction format. The instructions those are fetched from the instruction memory are placed in a special registers called immediate registers. These immediates can be further

transported by specifying the id of the immediate register in the source id of the move [13]. As of now, we do not use the long immediates.

2.2.5 Interconnection Network

This is the network that is responsible for the communication between the Registers and the FUs. This interconnection network can be any arbitrary network but in this work we basically consider a set of move buses and sockets as the interconnect network. Each move bus is controlled by a data move slot of the TTA instruction. Apart from the data, the move buses also carry the source and destination register ids of the move slot and also the control signal for exceptions, guarding etc. The registers and FUs are interfaced to the move buses with the help of sockets. The Sockets primarily consists of input multiplexers, comparator and output demultiplexers. The socket checks whether the id that is supplied on the bus is accessible to the register or FU connected to it. Depending on these accessibility criteria, the data is passed in the respective direction.

The code generation may get simplified by the use of a fully connected interconnect network. This means that the register file and the FUs are connected to every move bus available. This may affect the cycle time due to the high load on the move buses. Therefore the buses may be partially connected based on the requirements so as to minimize the load on the buses as much as possible. In this thesis we consider the use of fully interconnected TTA.

2.2.6 Operand Sharing and Software Bypassing

Operand Sharing

In the previous sections we have discussed that the handling of each and every moves are done individually by the compiler. Therefore it is possible to share an operand by multiple operations. This concept can be understood by considering the following example.

1. r2 -> sub1_op ; r3 -> sub1_tr ;
2. sub1_res -> r4 ; // r4 = r2 - r3
3. r2 -> sub2_op ; r5 -> sub2_tr ;
4. sub2_res -> r6 ; // r6 = r2 - r5

The above example shows the subtraction operation being performed in the same FU and the value of the common operand value is the same. Here in this case, the second operand

move r2 -> sub2_op can be eliminated because the value of r2 is already present in the operand register of the FU. This elimination can be performed when the common operand is provided via the same operand register to the FU and when the operand register is not changed by other intervening operations. The resultant code is as shown below.

1. r2 -> sub1_op ; r3 -> sub1_tr ;
2. sub1_res -> r4;
3. r5 -> sub2_tr ;
4. sub2_res -> r6;

Software Bypassing

This method can be used to eliminate the need of unwanted register file access. This can be explained with the help of the following example

1. r2 -> sub_op ; r3 -> sub_tr ;
2. sub_res -> r4 ; // r4 = r2 - r3
3. r4 -> add_op ; r5 -> add_tr ;
4. add_res -> r6 ; // r6 = r4 + r5

Software bypassing hence allows the two moves i.e., sub_res -> r4 and r4 -> add_op to be scheduled in the same instruction provided that the result of the subtraction is directly written to the operand of addition. This shortens the execution time and saves a read operation. The scheduled code after software bypassing is as shown below.

1. r2 -> sub_op ; r3 -> sub_tr ;
2. sub_res -> add_op ;
3. r5 -> add_tr ;
4. add_res -> r6 ;

From the above schedule we can also deduce that the move defined as sub_res -> r4 has been removed because the value of r4 is not needed anymore. This is called dead result elimination. This saves a data transport and RF write access and also the register requirements.

2.2.7 Control Flow and Conditional Execution

The control flow is necessary to execute the high level language statements and change the flow of program execution. This can be achieved by changing the contents of the program counter. This can be done in TTA by directly writing a value to the program counter. An unconditional jump can be simply performed by explicitly writing the address of the target of the jump in the program counter register. Conditional jumps are performed when a particular condition is met otherwise the normal execution continues. The conditional execution is possible by means of guarded or predicated execution. A boolean execution is associated with every move. When this boolean expression is evaluated to true, only then the move takes place. This boolean expression is called guard expression and is built out of boolean variables stored in RF of boolean registers. Consider the following example

```
if (r3 > r4)
   r5 = r6;
else
   r5 = r7;
```

```
r3 -> cmp_op ; r4 -> cmp_tr ; cmp_res -> b1;
b1 : r6 -> r5;
!b1 : r7 -> r5;
```

In the above example the operation cmp performs the comparison operation by moving the values to be compared to the operand and trigger register of this comparator unit and generates a boolean value. The boolean register b1 holds the result of the performed comparison and guards the subsequent move. The move following the guarded expression !b1 : is executed only when the expression evaluates to false. Similarly, the move following the guarded expression b1 : is executed only when the expression is true. Predicated execution is the conditional execution of instructions that is based on the boolean value of a source operand, referred to as the predicate. If the value of the predicate is evaluated to true (a logic 1), the program counter value is updated to point the branched instruction, otherwise the instruction following the condition is nullified and hence preventing it from modifying the processor state.

2.3 Advantages and Disadvantages of TTA

2.3.1 Advantages

Architectural consistency

The number of building blocks used in TTA is limited. These building blocks includes the move buses, sockets, RFs, FUs and an instruction unit containing the PC and the instruction register. Sockets are used to interface the RFs and FUs to the move buses. Therefore if the specifications to interface socket is established, then all the components can be designed and implemented independent of each other. This makes it possible to use libraries and layout generators of predefined components

Interfacing flexibility

When a new FU has to be added to the architecture, it can be easily done by plugging them into the interconnect network. When more connectivity is needed to interface the new FU or when there is insufficient addressing space on the move buses to address the new FU, only then some global changes might be required.

Processor cycle time

The minimum cycle time of a TTA implementation is the time required to do a data transport over a data move bus which can be very short. The presence of result registers at FU outputs and the pipelining of all other processor components helps to limit the cycle time.

Effective hardware utilization

Programming at data transport level leads to an effective control over the transport resources and leads to a better hardware utilization. The better utilization of transport resources allows more FUs per RF. This reduces the need of clustering.

Software bypassing

Bypassing a value saves an RF read access. This means that when there are two flow dependent moves that has to be scheduled in the same cycle then bypassing leads to effective scheduling and reduction in RF port requirements in TTA. The freed RF read port can then be used for other purposes.

Dead code elimination

After performing the bypassing, the moves that write to the GPR can be eliminated. These are usually the result moves that store the values in a register. This is possible only if the intermediate values would not be required any more for further computations.

Operand sharing

FUs using common operand values for multiple operations can share the operand value over the same bus and hence save multiple transport of the same operand.

2.3.2 Disadvantages

Code Size

The code size of TTA programs is usually greater than that of the OTA. This further increases the instruction cache miss rates.

Compiler complexity

Due to the complex programming of data transport, it is necessary to compile them effectively for effective utilization of the transport resources without compromising on the program execution sequence. The correct packing of move instruction further decides the utilization of hardware units.

Chapter 3

Design

In this chapter, we will be discussing the design implemented in this thesis. Firstly, we will be discussing about the basic TTA design followed by the dynamic scheduling algorithms.

3.1 Existing TTA based processor

In the previous chapters we have defined Transport Triggered architecture (TTA) as a super class of traditional VLIW architecture that has the capability to exploit not only operation level parallelism but also the parallelism available at data transport level. The block diagram of a TTA is shown in Figure 3.1. The TTA basically consists of a register file (RF) and a number of functional units (FUs) connected via the interconnect network. The RFs provide temporary storage of data whereas the interconnect network performs the transport between the FUs and RFs and the FUs carry out the execution of operations. The FUs are usually the simplest building blocks. An ALU for example can be divided into multiple FUs that can perform addition, subtraction, multiplication, division separately. Further, to perform the memory access we have a dedicated FU called the LOAD/ STORE unit. These FUs and RF are connected by means of interconnect network. A dedicated FU may also perform any arbitrary computations like hashing, signal processing, etc.

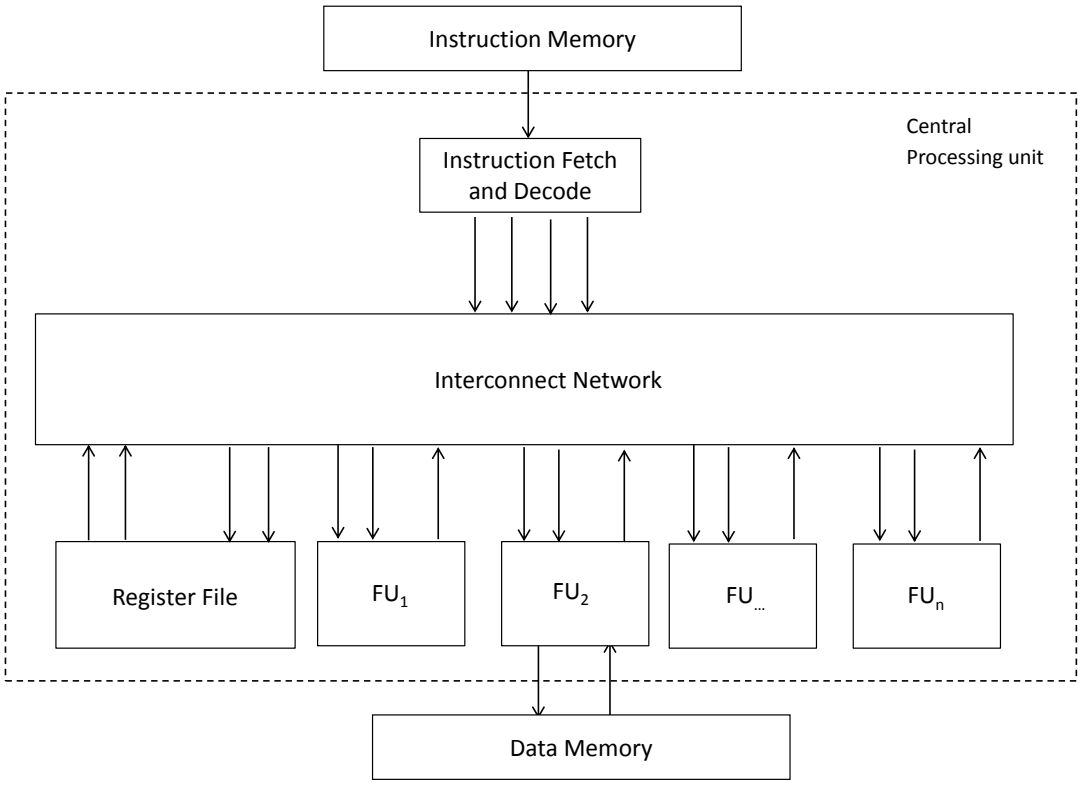


Figure 3.1: TTA Design

3.1.1 Working Principle

Instruction Fetch and Decode

An instruction in TTA is basically a set of data transports that specifies the source and target of an operation to be performed. The instructions to be executed are fetched from the instruction memory and decoded. From this decoding step, the CPU understands about the source and destination of the instruction. The decoded instructions are then passed on to the interconnect network by the control unit. The source operand can also be an immediate value that is specified in the instruction itself.

Interconnect network

The interconnect network consists of the transport buses and the sockets that connects to FUs and RFs. The interconnect network transports control signals, addresses and the data. It is responsible for transporting the data from one functional unit to another depending on the

source and target addresses of the decoded instructions. Figure 3.2 shows the block diagram of an interconnect network. An interconnection network can be any arbitrary network but in this thesis we shall consider fully connected buses and sockets as our interconnection network.

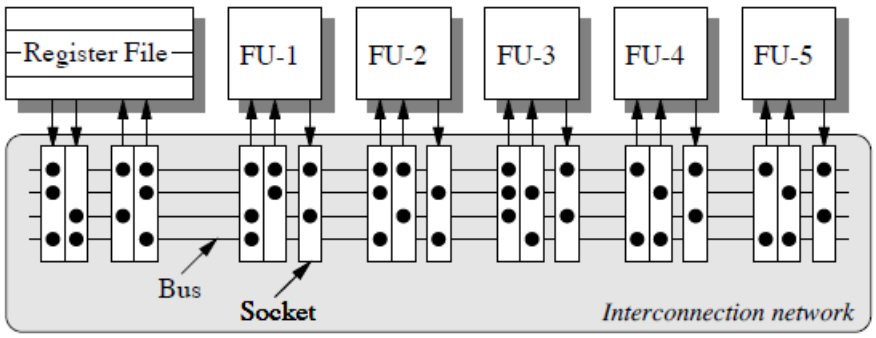


Figure 3.2: Interconnection Network

Register File

The register file acts as a temporary storage of results for a faster access by the functional units.

Functional Units

These are the units that are responsible for the actual computation of the decoded instructions. It consists of operand and trigger register that accepts the operands for computation. The result register holds the output after execution. The functional unit is triggered i.e. it starts the execution when the data is transported to the trigger register.

Data Memory

Data memory is connected to a functional unit called the load/store unit. Any requests to read from or write to memory location is handled by the load store unit.

Instruction Scheduling

In TTA the instructions are scheduled statically. This means that the compiler is responsible for scheduling the instructions at compile time. During run time, the processor just executes every instruction as scheduled by the compiler. In this design, we have considered a single cycle instruction execution. If a functional unit receives a request to perform a computation in clock cycle T, then the output is available in clock cycle T+1. It is also possible to have any arbitrary latency as long as it is known during compile time.

3.2 Latency Stalling

In the previous method, the instructions are scheduled by the compiler at compile time. This is called static scheduling of instructions. Due to this reason the compiler complexity increases. However the information regarding branch prediction, cache states etc available during compile time are limited. If variable latency functional units are present in the hardware architecture, then it becomes difficult to predict the maximum execution time and hence it may lead to the assumption of ambiguous values. This can lead to an under-utilization of performance due to static scheduling.

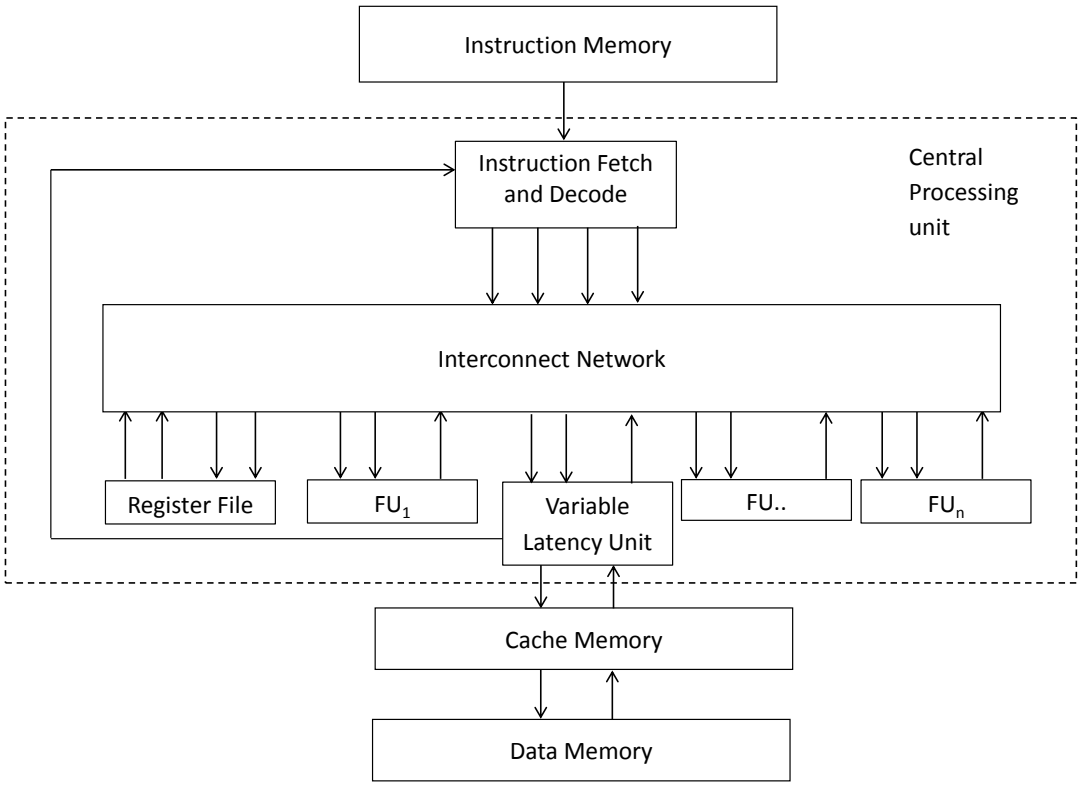


Figure 3.3: Latency Stalling

In the previous design, the computational modules were simple and we assumed a single cycle instruction execution. In this design we shall consider a situation where in the instructions will require variable clock cycles to finish the execution. This can usually be the case in terms of complex computational units like multiplier and divider depending on the range of input values. Memory access by load/store unit can also exhibit this kind of behaviour due to the presence of caches. In this design, we are basically concentrating on the load/store unit as

a variable latency unit. The block diagram of the architecture is presented in Figure 3.3. In this design we intend to utilize the maximum processing capabilities of the processor with the help of the information regarding processor states available during run-time and not compromising on the time - predictability.

The basic design employed in this architecture is similar to the one discussed in the previous section. In addition, a variable latency unit is introduced, that does the memory access. This is usually the load/store unit. This unit interacts with the data memory in association with the cache memory. When a cache memory is used, there can be two possibilities, first one is a cache hit which means that the data requested in the current clock cycle is present in the cache and can be provided back to the control unit in the next clock cycle. Another possibility is a cache miss which means that the requested data is not present in the cache and has to be fetched from the data memory. The process of fetching the data from the data memory to the cache can take several clock cycles. During this period, if the variable latency unit receives another request for memory access, it will not be able to service that request since it would be busy completing the previous memory access request. This situation is called a structural hazard where in two instructions tends to use a certain hardware in the same clock cycle. During this scenario, the further fetching of instructions must be stalled to avoid the resource conflict. This process of stalling the further instruction fetch during the on going execution of a variable latency instruction is therefore termed as latency stalling.

This design is an example of dynamic scheduling where in the scheduling is handled by the processor at run time. Here the control unit is responsible for either stalling or continuing the instructions fetch depending on the resource availability. This method is not possible to be implemented in compiler based scheduling as it would not have sufficient data regarding the latency of a particular instruction at during compile time.

Furthermore, in the presence of other variable latency units like multiplier and divider, the result is available only after a number of clock cycles depending on the algorithm used and the complexity of the input operands. The same principle of latency stalling introduced in this section can be applied to these functional units to resolve structural hazard.

3.3 Dynamic Scheduler

In this section we will be introducing a more advanced dynamic scheduler. In the previous section we considered the scenario where there is a structural hazard due to the access of a hardware resources by two instructions in the same clock cycle . In that case we introduced the concept of latency stalling. In such a situation, we would completely stall the fetching of further instructions. There would be other instructions present which could be executed in

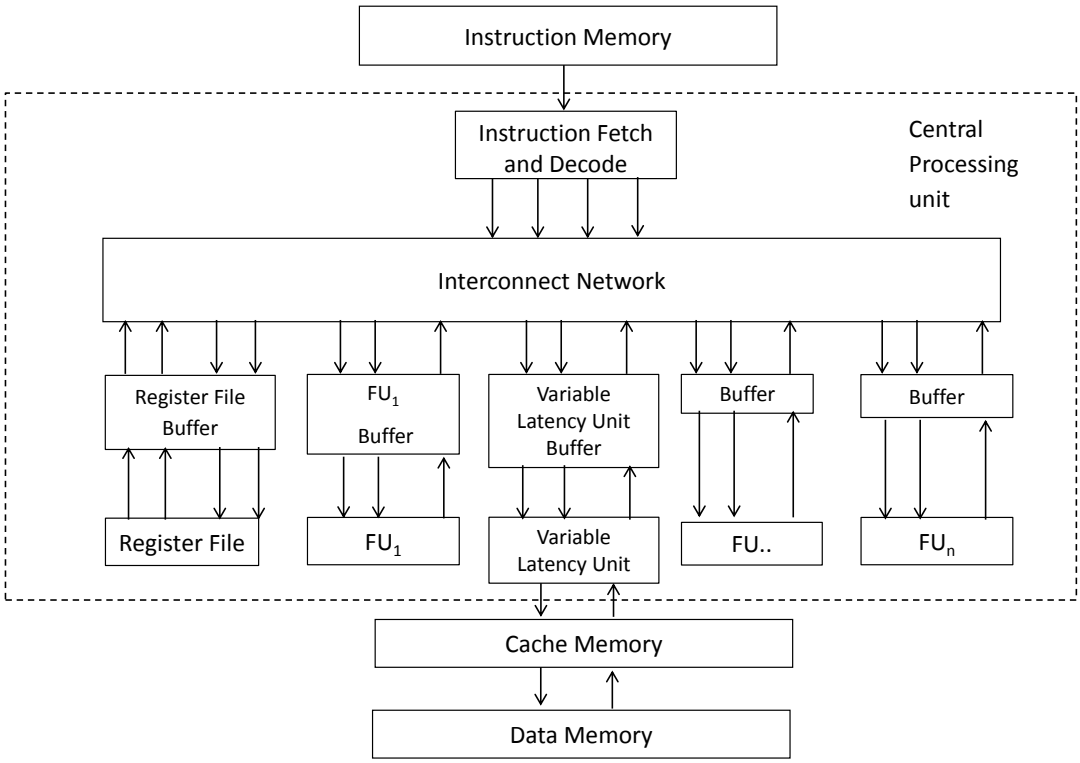


Figure 3.4: Dynamic Scheduling

different functional units rather than the one which is busy and initated the stalling operation. This would slow down the overall execution speed due to the presence of instructions that could be executed in parallel in other functional units but due to the stalling, they would not be issued to those functional units. The block diagram of the dynamic scheduler and the buffer is shown in Figure 3.4 and Figure 3.5 respectively.

In chapter 1 we discussed about the effects of timing anomalies in dynamically scheduled microprocessors. Taking into consideration that the timing anomaly does not occur in the presence of in order resources, a new dynamic scheduler is proposed in this thesis. Compared to the previous design, in this design, instead of a register for operand, trigger and result, we introduce FIFO buffers for the operands and result. Each functional unit has separate FIFO buffers for its inputs and output. The input buffers contain slots to store the source address of the operand and the data value. The result buffer contains two slots. One of the slots hold the target address where the data has to be transported and the other slot stores the computed data or the result. In this design the functional units also have the capability to access the interconnection network in addition to the instruction fetch and decode unit. After

the instruction fetch and decode, the address and corresponding data will be filled in the input buffers of the respective functional units where the instructions are intended to be executed. Similarly, in the corresponding result buffer slot, the target address is stored.

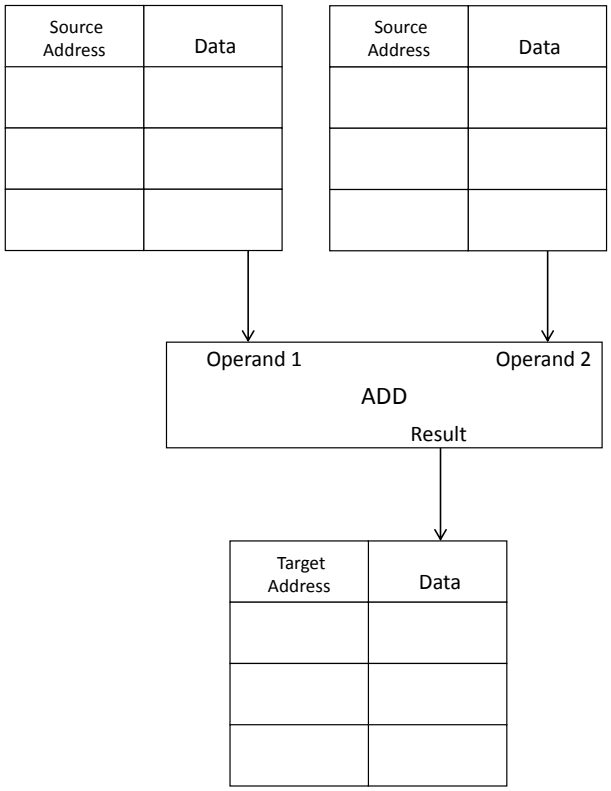


Figure 3.5: Buffer Implementation

During the issue of instructions to the buffer, we would consider two cases. In the first case we consider that the buffer has free slots and can accommodate more instructions. In the second case we consider that the buffer is full and cannot further accommodate any instructions until one of the instructions already present in the buffer completes the execution and is flushed out of the buffer to free a slot.

In the first case when the buffer slots in the functional units are free, the instructions are queued in the buffer until the buffer is full. In the second case, when the buffer is full, then the instruction fetch is stalled until a slot in the buffer becomes free to accommodate a new instruction.

Here we observe that the addition of buffers in a functional unit allows the instructions to be queued for execution. In case the buffer of a functional unit is full and the instruction fetch is stalled, the instructions already present in the buffers of the other functional units are

executed and we can intuitively say that the resource wastage is minimized. Whereas in the previous design the stalling of instructions fetch leads to resource wastage since the stalling of instruction fetch is a result of a structural hazard in a particular functional unit. Then in that case even if the other functional units are free, the instructions cannot be issued to them till the instructions fetch resumes. The functional units can be triggered based on the following cases

In-Order Triggering

The execution of the inputs present in the input FIFO buffers of the functional units takes place in first come first serve basis. The functional unit is triggered, when the corresponding tails of the input FIFO buffers holds a valid data for execution. It should be noted that the execution is strictly in-order in every functional unit. But the execution of the program as a whole can be out-of-order since the functional units are running in parallel and if there are no data dependencies between various functional units, then the instructions get executed irrespective of its order in the program. The result is then stored in the tail of the result FIFO buffer. In this thesis, this triggering mechanism was employed.

Out-of-Order Triggering

The execution of the inputs present in the input FIFO buffers of the functional units can take place out-of-order. When any slot in the input buffers have a valid data for execution, the execution can start immediately irrespective of its position in the buffer. In this case, the result is stored in the result FIFO buffer at same level compared to the input FIFO buffer slot that got executed.

3.4 Scheduling Example

In this section we shall take a programming example and show how it is scheduled in various architectures described above. Here in this example we assume that there are only 3 move buses available and the latency of the load/store operation is 5 cycles.

Example

1. 0->LSU_ADR, 1->LSU_WR, 1->LSU_DIN;

2. 1->LSU_ADR, 1->LSU_WR, 2->LSU_DIN;

3. 0->LSU_ADR, 1->LSU_WR, 3->LSU_DIN;
4. 2->REG_RD1ADR, 3->REG_RD2ADR; // read R2,R3
5. REG_R1DATA->ALU_OP_ADD, REG_R2DATA->ALU_TR_ADD; // ADDU R4,R2,R3 ; execute ADDU
6. ALU_RES1->DIV_OP;

We shall now schedule these data transports statically and dynamically.

Static scheduling

1. 0->LSU_ADR, 1->LSU_WR, 1->LSU_DIN;
2. NOP;
3. NOP;
4. NOP;
5. NOP;
6. NOP;
7. 1->LSU_ADR, 1->LSU_WR, 2->LSU_DIN;
8. NOP;
9. NOP;
10. NOP;
11. NOP;
12. NOP;
13. 0->LSU_ADR, 1->LSU_WR, 3->LSU_DIN;
14. NOP;
15. NOP;
16. NOP;

17. NOP;
18. NOP;
19. 2->REG_RD1ADR, 3->REG_RD2ADR; // read R2,R3
20. REG_R1DATA->ALU_OP_ADD, REG_R2DATA->ALU_TR_ADD; // ADDU R4,R2,R3 ; execute ADDU
21. ALU_RES1->DIV_OP;

Here in this example the data transports are scheduled statically by the compiler. We have used NOPs after each memory accesses because during compilation time we do not know whether we would encounter a cache hit or a cache miss. Hence, we assume the maximum latency possible. In this case, it would take a total of 21 clock cycles to complete the execution.

Latency Stalling

1. 0->LSU_ADR, 1->LSU_WR, 1->LSU_DIN; // Cache Miss, latency = 5 Cycles, Stall Fetch
2. NOP;
3. NOP;
4. NOP;
5. NOP;
6. NOP;
7. 1->LSU_ADR, 1->LSU_WR, 2->LSU_DIN; // Cache Miss, latency = 5 Cycles , Stall Fetch
8. NOP;
9. NOP;
10. NOP;
11. NOP;
12. NOP;

13. 0->LSU_ADR, 1->LSU_WR, 3->LSU_DIN; // Cache Hit, Data available in memory in next cycle
14. 2->REG_RD1ADR, 3->REG_RD2ADR; // read R2,R3
15. REG_R1DATA->ALU_OP_ADD, REG_R2DATA->ALU_TR_ADD; // ADDU R4,R2,R3 ; execute ADDU
16. ALU_RES1->DIV_OP;

Here in this design, the execution is completed in 16 clock cycles. Due to dynamic scheduling, the information regarding the cache state is available during runtime, hence the number of cycles are reduced compared to static scheduling where each and every memory access was assumed to be the maximum latency. In this case, the memory access in cycle 13 is completed within one clock cycle (due to cache hit) since the LSU_ADR is already present in cache due to its usage in data transport mentioned in clock cycle 1.

Dynamic Scheduler

Schedule in LSU with LSU buffer

1. 0->LSU_ADR, 1->LSU_WR, 1->LSU_DIN; //LSU_Buffer[0], cache Miss, latency = 5 Cycles
2. NOP;
3. NOP;
4. NOP;
5. NOP;
6. NOP;
7. 1->LSU_ADR, 1->LSU_WR, 2->LSU_DIN; //LSU_Buffer[1], Cache Miss, latency = 5 Cycles
8. NOP;
9. NOP;
10. NOP;

11. NOP;

12. NOP;

13. 0->LSU_ADR, 1->LSU_WR, 3->LSU_DIN; //LSU_Buffer[2], Cache Hit, Data available in memory in **next cycle**

Schedule in ALU with ALU buffer

// The REG buffer and ALU buffer are filled from cycle 4 and cycle 5 respectively and executed in parallel to the load/store unit. These data transports are available from clock cycle 4 onwards as given in the example

4. 2->REG_RD1ADR, 3->REG_RD2ADR; //Reg_Buffer[0], Cycle 4 // read R2,R3

5. REG_R1DATA->ALU_OP_ADD, REG_R2DATA->ALU_TR_ADD; //Add_IN_Buffer[0], cycle 5 // ADDU R4,R2,R3 ; execute ADDU

6. ALU_RES1->DIV_OP; //Add_IN_Buffer[1], cycle 6

Here in this schedule, a total of 14 clock cycles are needed to complete the execution. Unlike latency stalling, in this implementation the buffers are filled as soon as the data transports are available. Thereafter, the functional units execute independently.

Therefore, in this example, we can say that the dynamic scheduler is better than the latency stalling which is in turn better than static scheduling. Furthermore, examples for various other cases are not considered in this thesis.

Chapter 4

Implementation

4.1 Prerequisites

4.1.1 Quartz

Quartz is a synchronous programming language with concurrent actions. Synchronous programming languages were developed in the 1980s to aid the development of safety - critical embedded systems. The synchronous programming language has been successfully used for the design and verification of embedded systems. In this kind of language, The execution is divided into a sequence of synchronous steps or macro steps. Each macro step is a combination of a number of micro steps and these micro steps are considered to be executed in zero time in theory. For example, in a macro step a set of sensor inputs are read followed by some processing and then the output values are provided. Hence it is a reaction step. The differentiation between various time intervals (macro step) in which a particular action is performed is explicitly mentioned in a synchronous program in Quartz with the help of pause statement. Once a pause statement is encountered, it consumes one logical unit of time and hence clearly separates the other statements. The Synchronous languages Quartz is a concurrent language which is due to the concurrent execution of the micro steps in one macro step and also due to the availability of statements for the execution of parallel threads. The detailed description regarding the use of Quartz can be found in [**?**].

4.1.2 Averest Framework

Averest is a set of tools for hardware - software co-design and the verification of synchronous Quartz program. Averest allows to model, simulate and synthesize reactive systems. it consists of a compiler for translating synchronous programs to an intermediate format, a set of

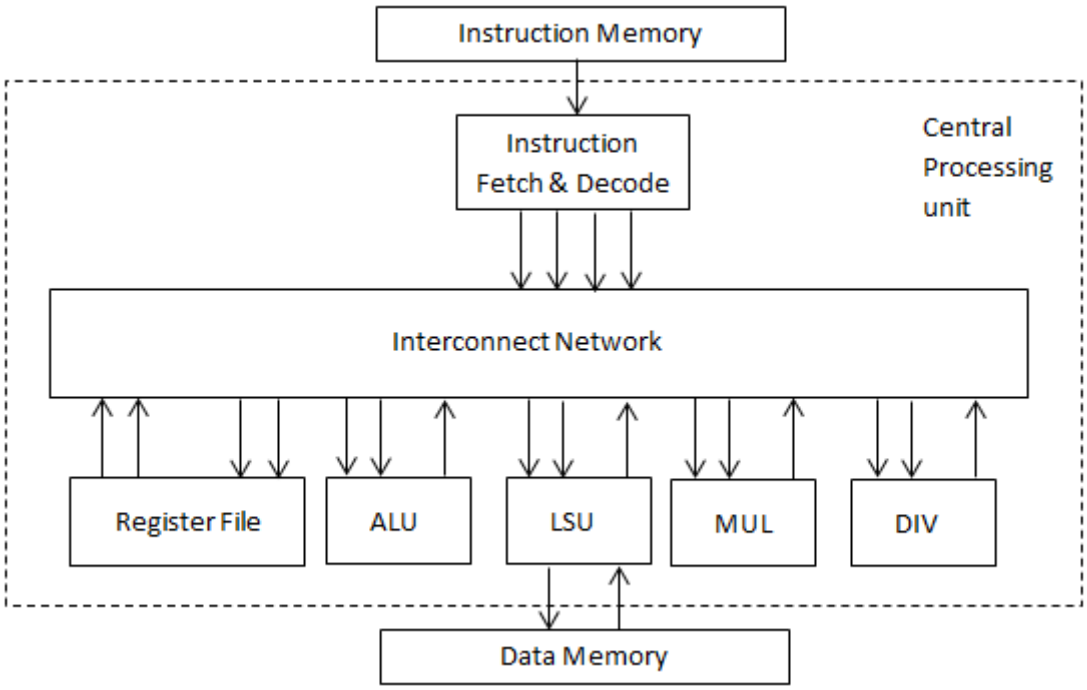


Figure 4.1: Simple TTA implementation in Quartz

transformations, a SAT solver. a simulator and tools for hardware - software synthesis. Hence Averest covers a large part of design flow of reactive systems from modelling to synthesis. The compiler computes a given Quartz proram into an Averest Interchange Format (AIF) file. To modify a generated AIF system description, a number of transformations are available. After suitable transformations, AIF systems can be converted to various target languages. There are code generators for hardware/software synthesis. A simulator and a code generator for the model checker SMV are also available in the Averest framework. As typical for a model based design, with the use of the same Quartz code, it is possible to generate software or hardware code without any changes to the Quartz module.

4.2 Existing TTA Implementation

The TTA hardware is implemented using Quartz on Averest framework. Initially a simple working model of TTA was implemented with basic functionalities consisting of the interconnect network, FUs, RF and Load Store Unit. The block diagram of this implemented is shown in Figure 4.1

4.2.1 Working Principle

Register File

These are the general purpose registers that are used to store intermediate computed values to speed up the execution of the programs. To write data into the registers, the address of the register has to be passed to this unit followed by the 8 bit data. Similarly, this unit has two read ports to perform the read operation. The address of the register whose data has to be read is passed first. The data is then available to be used by further instructions in the next clock cycle.

Arithmetic and Logic Unit (ALU)

The basic arithmetic and logical operations are performed by ALU. It represents the fundamental building block of the CPU. The ALU implementation is capable of performing arithmetic operations like addition and subtraction. The logical operations that can be performed are AND, OR, NOT, NAND, NOR and comparison. The ALU consists of one operand register, one trigger register and two result registers.

The instruction fetch and decode unit fetches the instructions from the instruction memory and then decodes them. After decoding, the function code of the instruction is available. This function code, the operand register value and the trigger register values are then passed to the ALU through the interconnect network. Then depending on the function code obtained, the necessary operations are performed. The latency of the ALU is initially one clock cycle. This means that if the function code, operand and trigger register value is available in clock cycle T, then the result is produced in clock cycle T+1.

Load Store Unit (LSU)

The Load Store Unit is responsible for managing all the load store operations. The operand register of the LSU receives the memory address on which load or store operation should be performed. The LSU trigger register receives the input which specifies whether the operation being performed is a load or a store. For store operation, the data to be stored is provided to the LSU data input register. In case of a load operation, the data present in the memory is made available at the LSU data out register.

Once the instructions are decoded by the instruction fetch and decode unit, the opcode of the instruction is available. This opcode contains the necessary details about the functional unit responsible for the particular decoded operation to be performed. The operand, trigger and the data input values are then appropriately transported to the LSU. The LSU then accesses

the data memory. If the LSU receives a read request in cycle T, it sends a read request to the main memory in the same clock cycle. The data memory module then sends the requested data to the LSU data out register in clock cycle T+2. The data is further sent back to the CPU in cycle T+2 for further usage. Similarly for data store operation, If the LSU receives a write request in cycle T, the data is stored in the data memory in cycle T+1. Further instructions can use this data from cycle T+2 onwards.

Multiplication Unit (MUL)

This unit is responsible of performing the multiplication operation. This unit waits for the operand and trigger event signals from the CPU once the decoding of an instruction is completed. Once it receives these signals, the 8 bit operand and 8 bit trigger values are passed to this unit to perform the multiplication. After the execution, this unit sends back a 16 bit result separated into 2*8 bit result registers.

Division Unit (MUL)

This unit is responsible of performing the division operation. This unit waits for the operand and trigger event signals from the CPU . Once it receives these signals, the 8 bit operand and 8 bit trigger values are passed to this unit to perform the division operation. After the execution this unit sends back the quotient and remainder stored in res1 and res2 registers respectively to the CPU.

Software / Driver Task

Since TTA exploits parallelism at transport level, the application to be executed is explicitly coded as data transports. This is simply possible by understanding the working principle of the architecture. Let us now consider a small example that shows the coding style for TTA.

```
0->LSU_ADR, 1->LSU_WR, 3->LSU_DIN; // 0 // ADDIU R1,R0,3, ST R1,R0,0 ; write mem
NOP; // 1 // NOP
1->LSU_ADR, 1->LSU_WR, 5->LSU_DIN; // 2 // ADDIU R1,R0,5, ST R1,R0,1 ; write mem
NOP; // 3 // NOP
2->LSU_ADR, 1->LSU_WR, 7->LSU_DIN; // 4 // ADDIU R1,R0,7, ST R1,R0,2 ; write mem
NOP; // 5 // NOP
```

Figure 4.2: Example of TTA Coding Style

The example in Figure 4.2 shows how to store an immediate value in the memory location. LSU_ADR is the operand register where the memory location to store the data is specified. LSU_WR is the trigger register which responds to values 0 or 1. 0 implies that the operation is a load operation and 1 implies store operation. LSU_DIN is the Data input register and holds the value that has to be stored in the memory. The use of NOP implies that the data that is written in one clock cycle is available in the memory in the next clock cycle for further usage. The usage of NOP completely depends on the understanding of the timing constraints under which the data is available for further usage by the next instructions.

4.3 Cache

Caches are smaller and faster memory that usually stores a copy of the data present in the main memory to speed up the computation process. The exchange of data from main memory to the cache is done in a block of fixed size. This is known as the cache line. A cache line consists of a valid bit , a tag and the data.

4.3.1 Working Principle

The cache module is designed in general such that it is capable of behaving either as a direct mapped cache or a set associative cache or a fully associative cache. The data width is 8 bits and the cache is capable of storing 8 bytes.

Read Hit

```
if(hit) {
        next(dataMem) = lineC[line][set][adrInL];
        pause;                  // for cache hit, give data to lsu in next cycle
        emit(doneMem);
}
```

Figure 4.3: Read Hit

In case of a read request, if it is a cache hit then the data from the cache line is given back to the LSU in the next cycle. Once the data is given back, the doneMem flag goes high in the same clock cycle which acknowledges the LSU about the completion of the memory access.

Write Hit

```
if(hit) {
        next(lineC[line][set][adrInL]) = dataMem;
        let(adrOffset = (tagC[line][set] * NumOfLines + line) * LineSize)
        next(Mem[adrOffset+adrInL]) = dataMem;
        pause;              // for cache hit, give control back to lsu unit in next cycle
        emit(doneMem);
}
```

Figure 4.4: Write Hit

In Figure 4.4, in case of a write request, if it is a cache hit then the data from the LSU is transferred to the cache line in the next clock cycle. At the same time the data is also written into the corresponding memory location. This is known as the write through method where the data is written onto the cache and the next level memory simultaneously. Once the data is written, the doneMem flag goes high in the next clock cycle which acknowledges the LSU about the completion of the memory access.

Cache Miss

In case of a cache miss, one of the blocks of the cache has to be first freed and then the data from the memory has to be fetched. In our implementation we have used the FIFO replacement policy. This is shown in Figure 4.5

```
{
   let(adrOffset = (tagC[line][0] * NumOfLines + line) * LineSize)
   {
        for(i=0..LineSize-1)
        next(Mem[adrOffset+i]) = lineC[line][0][i];
   }
         for(s=1..SetAssoc-1)
           {
                next(valid[line][s-1]) = valid[line][s];
                next(tagC[line][s-1])  = tagC[line][s];
                next(lineC[line][s-1]) = lineC[line][s];
           }
       set = SetAssoc-1;
}
```

Figure 4.5: Cache replacement

Here, the line in set 0 is first written back to the memory and the rest of the lines are shifted up, which is typically a FIFO replacement. Then once a cache line is freed, the requested

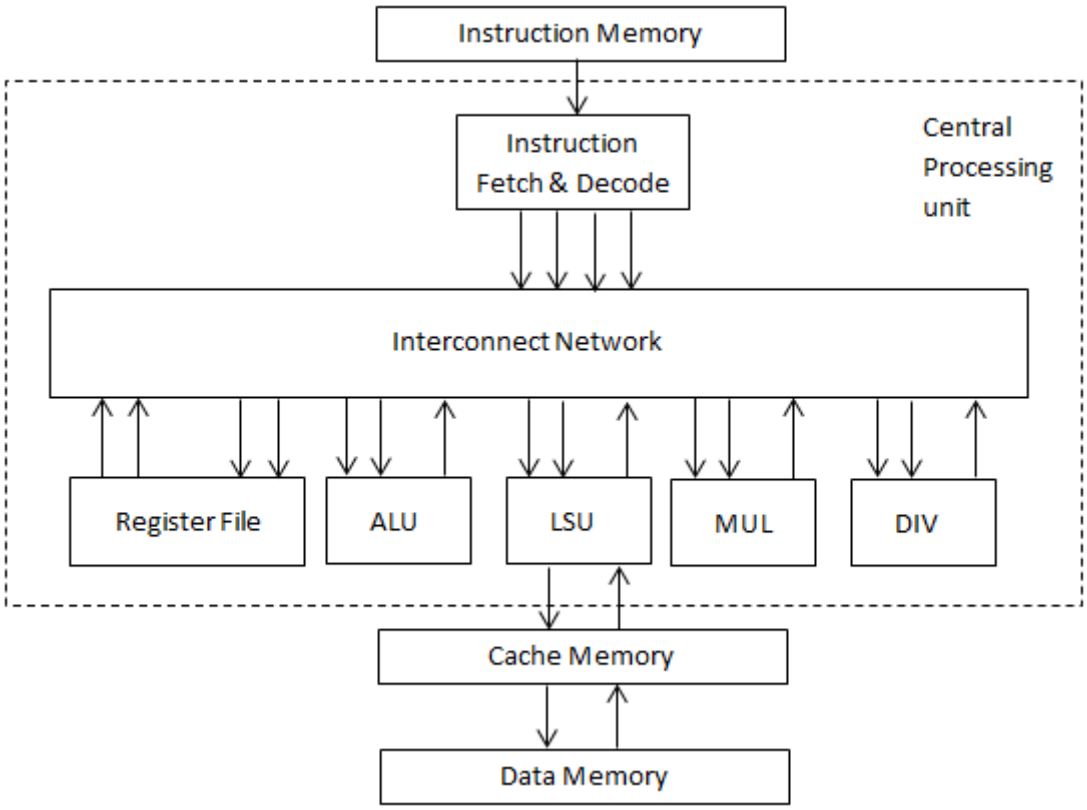


Figure 4.6: Block diagram of latency stalling hardware

cache line is fetched from the main memory and further read / write operations takes place as mentioned in case of a cache hit.

4.4 Latency Stalling

This is one of the first dynamic scheduling techniques proposed in this thesis. The block diagram of this implementation is shown in Figure 4.3

As explained in the previous section, a cache memory is implemented so as to introduce variable latency due to the cache behaviour. The working principle of each blocks are explained in section 4.2.1. In this implementation, the working of the LSU is altered since an additional cache memory introduced. In the block diagram shown in Figure 4.6, the MUL and the DIV units are also a possible variable latency unit, but in this thesis, the results are obtained by only considering LSU as a variable latency unit.

In processors, there are situations known as hazards that prevents the execution of next instruction in the instruction stream in the clock cycle that it is designated. For our implementation, let us understand the following hazards

Structural Hazard

These kind of hazards arise when two instructions wants to use a certain hardware component in the same clock cycle. This leads to a resource conflict.

Data Hazard

These hazards arise when an instruction is dependent on the data of previous instructions and hence attempts to use the data before it is even ready.

4.4.1 Working principle

In this implementation we shall consider a situation where in the instructions will require variable clock cycles to finish the execution. This can usually be the case in terms of complex computational units like multiplier and divider. Load/Store unit also exhibits this behaviour. In this thesis, only the load/store unit is modelled to exhibit the variable latency behaviour.

When the LSU receives a request to load/store data from/to the memory, if there is a cache hit, the data is available/stored right in the next clock cycle. Consider that there is a cache miss. In this case, the cache line has to be first loaded from the memory then the necessary data is read or can be stored further. This would take several cycles to complete based on the memory access latency. In this case, the LSU will not be able to service the request of new instructions until the previous execution has been completed. Hence there is a need to stall the instruction fetch until the previous execution is completed.

The LSU comprises of the LSU_ADR port that receives the memory address, the LSU_DIN that receives the data to be written in the memory, the LSU_WRITE which is the trigger register and receives either a read or write request, and the LSU_DOUT register that contains the data read from the memory in case of load operation.

Generation of LSU Busy Flag

```
loop {
        Event_occurred  = false;
        immediate await (ev_write);
        lsu_busy = true;
        next(Event_occurred) = true;

        r_write = x_write;
        readMem = ((r_write==0b00000000) ? true : false);
        writeMem = ((r_write==0b00000001) ? true : false);
        adrMem = bv2nat(r_adr);
        if(r_write == 0b00000001) dataMem = r_din;

        reqMem = true;
        immediate await (ackMem);
        immediate await (doneMem);
        lsu_busy = false;
        lsu_ack = true;
        Event_occurred  = false;

        if(r_write == 0b00000000) r_dout = dataMem;
        pause;
}
```

Figure 4.7: Generation of LSU Busy signal

Figure 4.7 shows a part of the code that generates lsu_busy signal until the data is stored/ loaded from the memory. Once the lsu receives the trigger (ev_write) then it immediately sets the lsu_busy flag true and further carries on the load/store operation. Once the LSUreceives the doneMem signal, which is sent by the memory module, it sets the lsu_busy flag as false which means that it is ready to accept a new instruction. The lsu_ack flag is used to instruct the CPU to fetch further instructions. There is also another possibility that the LSU receives another trigger signal when the current operation is in progress, to avoid this conflict, we use another flag called Event_occurred that solves this issue.

```
loop {
        immediate await (Event_occurred & (ev_dout or ev_write));
        lsu_ack = false;
        pause;
}
```

Figure 4.8: LSU_Ack signal to stall instruction fetch

In Figure 4.8 the lsu_ack signal is used to instruct the CPU to stall the instruction fetch in case the LSU is busy. When the LSU has received the trigger then the event_occurred signal goes true. Further when the LSU is busy performing the memory access, if a further data read request or data write request occurs, then the next instruction fetch must be stalled until the lsu_busy signal becomes false as shown in Figure 4.4

Once the signals are generated by the LSU, the control unit then further stalls the instruction fetch as shown in Figure 4.9

Program Counter Handling

```
        Loop {
                immediate await (not branch & ack_fetch);
                 next(pc) =pc+1;
                pause;
        }

||

        Loop {
                immediate await (not ack_fetch);
                next(pc) = pc;
                pause;
        }
```

Figure 4.9: Stalling the instruction fetch

In Figure 4.9, the ack_fetch flag is the internal flag of control unit which is either true or false based on the lsu_ack flag as mentioned above. If the ack_fetch is true then the program counter (PC) is incremented. If ack_fetch is false then the PC is not incremented. When the PC is not incremented, the stalling operation is achieved since the further instruction fetch does not happen until the previous instruction has finished the execution and the required hardware is free for the next instruction to execute.

4.5 Dynamic Scheduler

In the previous section we introduced latency stalling as an algorithm for dynamic scheduling. This implementation is carried out as a result of my research on timing anomalies in dynamically scheduled microprocessors. In chapter 1, the concepts related to timing anomalies were introduced. A few examples were presented that showcased the scenerios when timing anomalies can occur.

According to Lundqvist-et-al, timing anomaly did not occur in the presence of in-order resources but occurred only in the presence of out-of order resources [15]. In this implementation we suggest a scheduling method that issues instructions to the functional units in-order. This means that the functional units shall execute instructions in the order of its arrival.

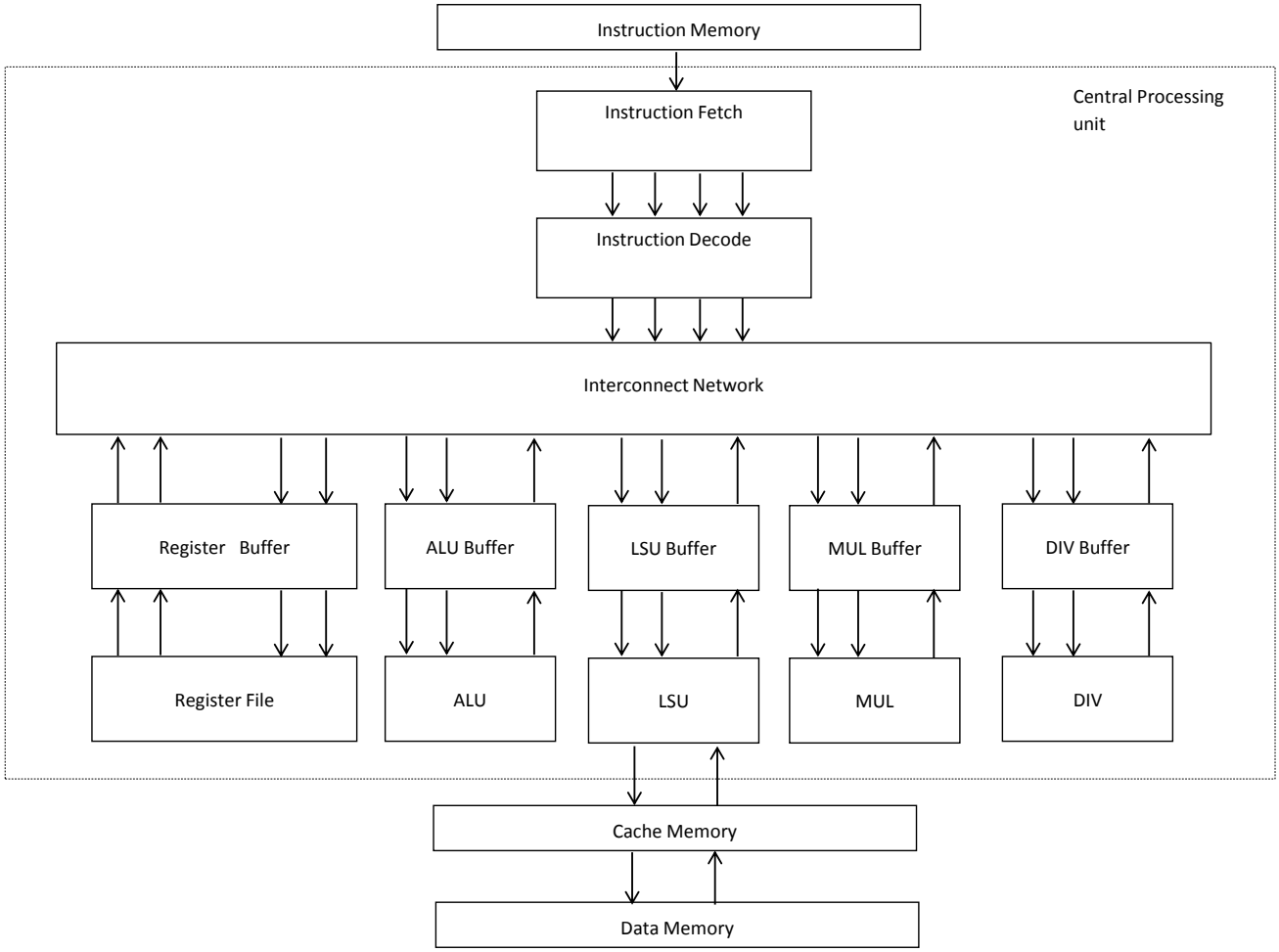


Figure 4.10: Dynamic Scheduler

Figure 4.10 shows the block diagram of the dynamic scheduler

Instruction Fetch and Decode

The instructions present in the instructions memory are fetched and stored in a FIFO fetch buffer. In Figure 4.11(a), the instruction fetch mechanism is shown. All the data transports in an instruction are fetched and stored in the fetch buffer. Figure 4.11(b) shows the decode mechanism. In the next cycle, the instruction from the fetch buffer is decoded in first come first serve basis and in the same cycle the decoded data is transported to the corresponding functional units.

```
for(i=0..NumBus-1) {
instr[i] = prog[pc][i] ;

if(k<buffersize)
{
    fetch_Buffer[k][i] = instr[i];
    fetch_Buffer_Free[k] = false;
}
```

(a) Instruction Fetch

```
for(i=0..NumBus-1) {

    next (decode_instr[i]) = fetch_Buffer[d][i];

    fncode[i] = FnCodeInstr(decode_instr[i]);
    signed[i] = SignedInstr(decode_instr[i]);
}
```

(b) Instruction Decode

Figure 4.11: Instruction Fetch and Decode

Buffer Implementation

In this design of dynamic scheduler, the buffers are implemented inside the functional units. The buffers are implemented in FIFO format. The instruction that is filled first in the buffer is executed first. In Figure 4.12(a), the initial status of the buffer free flag is shown. This flag is used to monitor whether a slot in the buffer is filled or whether it is free. In Figure 4.12(b), the buffer filling mechanism is shown. Since the LSU consists of read/write signal, the data input (din) and the address of memory location (adr), once these signals are detected by the LSU, the respective operands are filled in the buffer slot and then the buffer free flag is set to false indicating that the buffer slot is filled.

```
for(l=0..buffersize-1)
{
   LSU_WR_Buffer_Free[l] = true;
   LSU_ADR_Buffer_Free[l] = true;
   LSU_DIN_Buffer_Free[l] = true;
}
```

(a) Buffer Free Flag

```
immediate await (ev_write);
(LSU_WR_Buffer[lsu_wr_buf]) = x_write;
 LSU_WR_Buffer_Free[lsu_wr_buf] = false;

immediate await (ev_adr);
(LSU_ADR_Buffer[lsu_adr_buf]) = x_adr;
LSU_ADR_Buffer_Free[lsu_adr_buf] = false;

immediate await (ev_din);

(LSU_DIN_Buffer[lsu_din_buf]) = x_din;
(LSU_DIN_Buffer_Free[lsu_din_buf]) = false;
```

(b) Buffer Filling

Figure 4.12: Buffer Implementation

Instruction Execution

When there is a valid entry present in the buffer, the execution of that instruction starts. In Figure 4.13(a), the validation mechanism of a buffer entry is shown. When the corresponding entries of the read/write buffer, address buffer and the data input buffer is filled, then it is considered that a valid data is present and the buffer entry flag is set to true. We also check that the read/write buffer contains only values 0 or 1 to indicate the write or read operation. Else the Buffer entry would not be valid. In Figure 4.13(b) the trigger generation is shown. When the buffer entry is valid, the corresponding operands are passed to the internal registers of the LSU and then the trigger is set to true. After this, the execution mechanism is similar to the one shown in Figure 4.7

```
immediate await (!LSU_ADR_Buffer_Free[trig] & !LSU_WR_Buffer_Free[trig] | !LSU_DIN_Buffer_Free[trig]);
immediate await (((LSU_WR_Buffer[trig] == 0b00000000) | (LSU_WR_Buffer[trig] == 0b00000001)));
Buffer_Entry_Valid = true;
```

(a) Buffer Entry Validation

```
immediate await (Buffer_Entry_Valid);

r_adr = LSU_ADR_Buffer[execution];
r_write = LSU_WR_Buffer[execution];
r_din = LSU_DIN_Buffer[execution];
trigger  = true;
```

(b) Trigger Generation

Figure 4.13: Instruction Execution

Instruction Stalling

When the buffer in a functional unit is filled completely, and if there is a request to access the functional unit whose buffer is filled, then the further instruction decode must be stalled till a slot in the buffer of that functional unit is freed. However, the instruction fetch continues until the fetch buffer has empty slots to accommodate the new instructions. If the fetch buffer is also full, then the instruction fetch also has to be stalled. In Figure 4.14(a), the mechanism to check whether the buffer is free or not is shown. When at least one of the slots in the buffer is free, then the buffer fill flag is set to true. This instructs the CPU that it has empty slots left to accommodate more instructions. In Figure 4.14(b) the mechanism to stall the instruction fetch by the CPU is shown. If the buffer fill flag is true then the PC is incremented and a new instruction is fetched. Similarly if the buffer fill flag is false, then the PC value is not incremented.

```
for(i=0..buffersize-1)
  if((LSU_ADR_Buffer_Free[i] == true) & (LSU_WR_Buffer_Free[i] == true) & (LSU_DIN_Buffer_Free[i] == true))
  {
        buffer_fill = true;
  }
```

(a) Buffer Fill Flag

```
        loop{

                immediate await (not branch & buffer_fill);
                next(pc) = pc + 1;
                pause;
        }
||
        loop{

                immediate await (not buffer_fill);
                next(pc) = pc;
                pause;
        }
```

(b) PC Manipulation

Figure 4.14: Instruction Stalling

4.6 Experimental Results

The experiments were conducted on the latency stalling hardware exhibiting dynamic scheduling containing a variable latency unit and on a simple hardware that schedules these instructions statically. The observations are as shown in the following Table

Specifications

1. Cache - latency = 6 Cycles, FIFO replacement, Write Through
2. No. of test runs = 5000

Test Applications

1. Bubble Sort - 3 Values
2. Insertion Sort - 3 Values
3. Binary Search - 5 Values

4. Reads in Loop- 6 reads - 4 Iterations
5. Writes in Loop - 7 writes - 4 Iterations
6. Read & Modify in Loop- 2 Read & Modify, 1 Read only - 4 Iterations

Applications	Latency Stalling			Static Scheduling		
	Min	Avg	Max	Min	Avg	Max
	No. of Cycles			No. of Cycles		
Bubble Sort	89	97.94	107	122	127.5	138
Insertion Sort	90	98.95	108	123	138.72	155
Binary Search	40	65.72	84	58	88.91	110
Read Loop	95	126.45	179	223	223	223
Write Loop	151	164.32	203	223	223	223
Read - Modify Loop	139	155.62	195	223	223	223

Table 4.1: Experimental Results

From the above table we can observe that the average execution time of the programs in the latency stalling hardware that exhibits dynamic scheduling technique is lesser than that of the ones executed in the statically scheduled hardware.

Chapter 5

Conclusion and Future Scope

In this thesis we discussed about the dynamic scheduling of transport triggered architecture. Efforts were taken to explore the dynamic scheduling possibilities in this class of processor architecture. We presented two algorithms that use the dynamic scheduling techniques to schedule the instructions for execution. We introduced latency stalling where the structural hazard in the variable latency functional unit could be eliminated during run - time by stalling the instruction fetch. We also introduced dynamic scheduler which used FIFO buffers to store the inputs and result. This scheduling technique was proposed as an enhancement to latency stalling mechanism where the stalling could lead to resource wastage. In this design, we also considered the effect of timing anomalies. In addition to the improved performance, the time - predictability was also a design goal.

The implementations of the proposed techniques were confined to the load store unit due to time constraints. In the further researches, these algorithms can be extended to other functional units and can thus be integrated for a possible performance enhancement. To exhibit the variable latency behaviour of the load/store unit, we implemented a cache module with FIFO replacement and write through method. Previous researches in the cache replacement methods have proven that the Least Recently Used (LRU) replacement policy has a better performance than that of the FIFO replacement policy. Hence, the cache unit here implemented can be possibly enhanced using the LRU replacement policy. In the further researches, the cache enhancement along with its integration to these dynamic scheduling hardware could be thought of. In this implementation, we have considered fully connected buses and sockets as the interconnection network. One can also think of implementing the partially connected buses and sockets to connect only the required functional units hence saving the area and power consumption. Other interconnection networks known can also be implemented. The dynamic scheduler was implemented and the basic functionality was tested with the load/store unit but due to time restrictions, the detailed testing of the functionality and the performance

was not possible.

References

[1] Anoop Bhagyanath and Klaus Schneider. Tta as predictable architecture for real-time applications. In *Science Engineering and Management Research (ICSEMR), 2014 International Conference on*, pages 1–9. IEEE, 2014.

[2] Pohua P Chang, William Y Chen, Scott A Mahlke, and Wen-mei W Hwu. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 25–33. ACM, 1991.

[3] Henk Corporaal. Design of transport triggered architectures. In *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV'94, Proceedings., Fourth Great Lakes Symposium on*, pages 130–135. IEEE, 1994.

[4] Henk Corporaal. A different approach to high performance computing. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 22–27. IEEE, 1997.

[5] Henk Corporaal and Hans JM Mulder. Move: A framework for high-performance processor design. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 692–701. ACM, 1991.

[6] Jakob Engblom. Processor pipelines and static worst-case execution time analysis. 2002.

[7] DN Glass. Compile-time instruction scheduling for superscalar processors. In *Compcon Spring'90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 630–633. IEEE, 1990.

[8] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[9] John L Hennessy. Vlsi processor architecture. *Computers, IEEE Transactions on*, 100 (12):1221–1246, 1984.

[10] Jan Hoogerbrugge. *Code generation for transport triggered architectures*. TU Delft, Delft University of Technology, 1996.

[11] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 191–195. ACM, 1994.

[12] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Compiler Construction*, pages 435–449. Springer, 1994.

[13] Johan Janssen. *Compiler strategies for transport triggered architectures*. TU Delft, Delft University of Technology, 2001.

[14] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 119–128. IEEE, 2009.

[15] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 12–21. IEEE, 1999.

[16] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[17] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *OASIcs-OpenAccess Series in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[18] Kevin W Rudd and M Flynn. Instruction-level parallel processors-dynamic and static scheduling tradeoffs. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*, pages 74–81. IEEE, 1997.

[19] Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[20] Klaus Schneider and Tobias Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Conference on Application of Concurrency to System Design (ACSD)*. Citeseer, 2005.

[21] Michael D Smith, Monica S Lam, and Mark A Horowitz. *Boosting beyond static scheduling in a superscalar processor*, volume 18. ACM, 1990.

[22] Ashrit Triambak. Timing anomalies and wcet analysis. *Seminar Report, Embedded Systems Research Group, Dept of Informatik, TU Kaiserslautern*, 2014.

[23] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 295–303. IEEE, 2005.