ABSTRACT

Aim of the project is to make a VLIW (very large instruction word) processor in an EXPRESSION ADL. Such processors include one or more functional units, each capable of performing a certain class of functions in parallel. Such processors utilize these multiple functional units simultaneously, to execute programs faster.

The first step in a top-down validation methodology is to capture the programmable architectures using a specification language. The language should be powerful enough to specify the wide spectrum of contemporary processor, coprocessor, and memory features.

I have designed and implemented the Machine Description of a VLIW processor in EXPRESSION Architecture Description Language. Advances in semiconductor technology permit increasingly complex applications to be realized using programmable systems-on-chips (SOCs). Architecture Description Language (ADL) is a computer language used to describe software and/or system architectures. EXPRESSION supports architectural design space exploration for embedded Systems-on-Chip (SOC) and automatic generation of a retargetable compiler/simulator toolkit. A VLIW implementation has capabilities to those of a superscalar processor issuing and completing more then one operation at a time. For the VLIW implementation, the long instruction word already encodes the concurrent operations. The processor implemented in ADL presents us with the opportunity to improve on the short comings in design.

Detailed EXPRESSION language manual is also attached with the Thesis for future references. My design includes both processing and control unit.

ACKNOLEDGEMENTS

"All Praise belongs to ALLAH alone, Lord of all Worlds. Who created the heaven and the earth and all that is between the two and indeed in them there are many signs for those who seek."
(**AI-Qur'an)**
First of all, I bow my head in deep gratitude to ALMIGHTY who endowed me with the potential and ability to carry out this task with success and to the Holy Prophet (Peace be upon him) for being a beacon of knowledge and learning for the mankind.
I would like to thank my respected teacher Dr Shoaib who assigned me this project to build my Knowledge and for his constant encouragement, invaluable advice and help. Special thanks to Dr. Khalid, Head of Computer Engineering Department, who has been extremely helpful through out my project. I also appreciate the help of laboratory staff.
I must acknowledge the prayers and well wishes of my parents and family who are behind every success and perhaps is the major stimulating force that facilitates in achieving what I aspire. I am also thankful to my class fellows for motivating comments.

TABLE OF CONTENTS

LIST OF FIGURES:

Chapter #1

1. INTRODUCTION

1.1. ADL overview:

The first step in a top-down validation methodology is to capture the programmable architectures using a specification language. The language should be powerful enough to specify the wide spectrum of contemporary processor, coprocessor, and memory features. On the other hand, the language should be simple enough to allow correlation of the information between the specification and the architecture manual. Specifications widely in use today are still written informally in natural language like English. Since natural language specifications are not amenable to automated analysis, there are possibilities of ambiguity, incompleteness, and contradiction: all problems that can lead to different interpretations of the specification.

Many formal and semi-formal specification languages for describing software and hardware designs have been proposed over the years. The languages range in expressiveness and their different levels of granularity determine their appropriateness for different applications.

Advances in semiconductor technology permit increasingly complex applications to be realized using programmable systems-on-chips (SOCs). Furthermore, shrinking time-to-market demands, coupled with the need for product versioning through software modification of SOC platforms, have led to a significant increase in the software content of these SOCs. However, designer productivity is greatly hampered by the lack of automated software generation tools for the exploration and evaluation of different architectural configurations. Traditional hardware-software co design flows do not support effective exploration and customization of the embedded processors used in programmable SOCs. The inherently application-specific nature of embedded processors and the stringent area, power, and performance constraints in embedded systems design critically require a fast and automated architecture exploration methodology. Architecture description language (ADL)-Driven design space exploration and software toolkit generation strategies present a viable solution to this problem, providing a systematic mechanism for a top-down design and validation of complex systems. The heart of this approach lies in the ability to automatically generate a software toolkit that includes an architecture-sensitive compiler, a cycle-accurate simulator, assembler, debugger, and verification/validation tools.

Architecture Description Language (ADL) is a computer language used to describe software and/or system architectures. This means in case of technical architecture, the architecture must be communicated to software developers. With functional architecture, the software architecture is communicated with stakeholders and enterprise engineers.

ADLs result from a linguistic approach to the formal representation of architectures, and as such they address its shortcomings. Also important, sophisticated ADLs allow for early analysis and feasibility testing of architectural design decisions.

1.1.2. Architecture in the past

Architectures in the past were largely represented by box-and-line drawing. The Nature of the components, component properties, Semantics of connections and behavior of the system is usually defined in such a drawing:

1.1.3. The ADL Must:

ADL should be suitable for communicating architecture to all interested parties and also support the tasks of architecture creation, refinement and validation.

It should also provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL.

ADL should provide the ability to represent most of the common architectural styles and support analytical capabilities or provide quick generating prototype implementations

1.1.4. ADLs have in common:

Architecture description languages have common graphical syntax with often a textual form and a formally defined syntax and semantics, features for modeling distributed systems, little support for capturing design information, except through general purpose annotation mechanisms and ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates

1.1.5 ADLs differ in their ability to:

Architecture description languages can have ability to handle real-time constructs, such as deadlines and task priorities, at the architectural level, support the specification of different architectural styles. Few languages handle object oriented class inheritance or dynamic architectures and also support analysis.

1.1.6. Positive elements of ADL

ADLs represent a formal way of representing architecture, ADLs are intended to be both human and machine readable, ADLs support describing a system at a higher level than previously possible, ADLs permit analysis of architectures – completeness, consistency, ambiguity and ADLs can support automatic generation of software systems

1.1.7. Negative elements of ADL

There is no universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture.

Representations currently in use by ADLs are relatively difficult to parse and are not supported by commercial tools

Most ADLs tend to be very vertically optimized toward a particular kind of analysis

The ADL community generally agrees that Software Architecture is a set of components and the connections among them.

1.1.8. Object Connection Architecture

Configuration consists of the interfaces and connections of an object-oriented system, interfaces specify the features that must be provided by modules conforming to an interface, connections represented by interfaces together with call graph, conformance usually enforced by the programming language.

1.1.9. Interface Connection Architecture

Expands the role of interfaces and connections, interfaces specify both “required” and “provided” features, connections are defined between “required” features and “provided” features.

Consists of interfaces, connections and constraints, constraints restrict behavior of interfaces and connections in an architecture, constraints in an architecture map to requirements for a system.

Most ADLs implement interface connection architecture.

1.2. ADL supports:

An ADL is a language that provides features for modeling a software system’s conceptual architecture, distinguished from the system’s implementation.
An ADL must support the building blocks of an architectural description Components, Interfaces, Connectors, and Configurations.

1.2.1. Desirable features of an ADL are

We should be able to defined specific aspects of components, connectors, configurations and should also have tool support.

1.2.2. Approaches to modeling configurations

There are three types of modeling configurations, Implicit configuration, In-line configuration and Explicit configuration.

1.2.3. Approaches to associating architecture with implementation

There are two types of associating architecture, Implementation constraining and Implementation independent.

1.3. Types of ADL:

There are two types of ADLs: *software* ADLs and *hardware* ADLs.

1.3.1. Software ADLs

For software ADLs, the description is of the software architecture. Therefore, the components are *software processes or modules*. According to Kogut and Clements, ADLs seek to increase the understandability and re-usability of architectural designs, and enable greater degrees of analysis. ADLs are used to define and model system architecture prior to system implementation. Among the issues ADLs address are the following

1.3.1.1. Component behavioral specification.

ADLs typically provide support for specifying both functional and non-functional characteristics of components. (Non-functional requirements include those associated with safety, security, reliability, and performance.) Depending on the ADL, timing constraints, properties of component inputs and outputs, and data accuracy may all be specified.

1.3.1.2. Component protocol specification.

1.3.1.3. Connector specification.

ADLs contain structures for specifying properties of connectors, where connectors are used to define interactions between components.

1.3.2. Hardware ADLs

Hardware ADLs are principally concerned with describing the hardware components. This is often the case when dealing with *application specific instruction-set processor*

(ASIPs) within a -design process. Therefore, the languages describe the processors in terms of their instruction sets. Hence, they are sometimes called *machine description languages*.

1.4. ASIPs:

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Shrinking time-to-market, coupled with short product lifetimes create a critical need for rapid exploration and evaluation of candidate System-on-Chip (SOC) architectures. System architects critically need tools, techniques, and methodologies to perform rapid architectural exploration for a given set of applications to meet the diverse requirements, such as better performance, low power, smaller silicon area, higher clock frequency etc.

The existing approaches are either semi-automatic (expects designers to write data path components manually) or covers a restricted set of architectures. However, none of these approaches are able to capture a wide spectrum of processor features present in DSP, VLIW, EPIC and Superscalar processors, and generate synthesizable RTL from the ADL specification. The main bottleneck has been the lack of an abstraction (covering a diverse set of architectural features) that permits the reuse of the primitives to compose the heterogeneous architectures.

Modern Application Specific Instruction-set Processors (ASIPs) face the demanding task of delivering high performance for a wide range of applications. For enhancing the performance, architectural features e.g. pipelining, VLIW etc are often employed in ASIPs, leading to high design complexity. Integrated ASIP design environments like template-based approaches and language driven approaches provide an answer to this growing design complexity. At the same time, increasing hardware design costs have motivated the processor designers to introduce high flexibility in the processor. Flexibility, in its most effective form, can be introduced to the ASIP by coupling a re-configurable unit to the base processor. Due to its obvious benefits, several re-configurable ASIPs (rASIPs) have been designed in the recent years. These rASIP designs lacked a generic flow from high-level specification, resulting into intuitive design decisions and hard-to-retarget processor design tools. Field-programmable logic (FPL) is rapidly becoming established in markets requiring high-performance, low lead time and the ability to perform soft-upgrades on site. However, few current FPL systems utilize run-time reconfiguration (RTR) and those that do rely on infrequent and coarse grained reconfiguration.

Due to its obvious benefits, several re-configurable ASIPs (rASIPs) have been designed in the recent years. These rASIP designs lacked a generic flow from high-level specification, resulting into intuitive design decisions and hard-to-retarget processor design tools. Although a template-based approach for rASIP design is existent, a clear design methodology especially for the pre-fabrication architecture exploration is not present. In order to address this issue, a high-level specification and design methodology for partially re-configurable VLIW processors is proposed.

Improvements in the processor performance come from two main sources: **FASTER SEMICONDUCTOR TECHNOLOGY** and **PARALLEL PROCESSING**. Parallel processing on multiprocessors, multicomputers & processor clusters has traditionally

involved a high degree of effort in mapping an algorithm to a form that can better exploit multiple processors & threads of execution .Such reorganization has often been productively applied especially for scientific programs. The general purpose microprocessor industry on the other hand has pursued methods of automatically speeding up existing programs without major restructuring effort. This leads to the development of Instruction Level Parallel processor that tries to speed up program execution by overlapping the execution of multiple instructions from an otherwise sequential program.
ILP processors achieve their high performance by causing multiple operations. Some methods which exploit ILP are:

- PIPELINING
- MULTIPLE PROCESSORS
- SUPERSCALAR IMPLEMENTATION
- SPECIFYING MULTIPLE INDEPENDENT INSTRUCTIONS PER CYCLE.

1.4.1. PIPELINING

Pipelining is now universally implemented in high-performance processors. It is a means of introducing parallelism into the essentially sequential nature of a machine instruction. Examples are instruction pipelining and vector processing.

1.4.2. MULTIPLE PROCESSORS

Multi processors improve performance for only a restricted set of applications.

1.4.3. SUPERSCALER IMPLEMENTATION

Superscalar implementation can improve for all types of applications. Superscalar (super beyond; scalar one dimensional) means the ability to fetch, issue to execution units, and complete more than one instruction at a time. Superscalar implementations are required compatibility must be preserved, and they will be used for entrenched architecture with legacy software, such as the X86 architecture that dominates the desktop computers.
Specifying multiple operations per instruction creates a **very long instruction word architecture or VLIW**. A VLIW implementation has capabilities to those of a superscalar processor issuing and completing more then one operation at a time with one important exception: the VLIW hardware is not responsible for discovering opportunities to execute multiple operations concurrently. For the VLIW implementation, the long instruction word already encodes the concurrent operations.
Very long instruction word architecture is suitable alternative for exploiting instruction level parallelism (ILP) in programs that is for executing more then one instruction at a

time. The **VLIW processor** executes the set of operations within a **MultiOp**, which is a long instruction word consist of multiple arithmetic, logic & control operations each of which would probably be an individual operation on a simple **RISC** processor, thereby achieving instruction level parallelism

1.5. VERY LONG INSTRUCTION WORD (VLIW) PROCESSOR:

"Very long instruction word **(VLIW)** describes a computer processing architecture in which a language compiler or pre-processor breaks program instructions down into basic operations that can be performed by the processor in parallel (i.e. at the same time). These operations are put into a very long instruction word which the processor can then take apart without further analysis, handling each operation to an appropriate functional unit. Such architectures have more then one functional units, for parallel processing. The processor fetches different instructions as one long instruction, and then it breaks them and dispatches them accordingly to the different functional units."

By **Joseph Fisher**, VLIW is an architecture which issues one instruction per cycle, where each long instruction called MultiOp consists of many tightly coupled independent operations each of which execute in a small and statically predictable number of cycles. In such a system the task of grouping independent operations into a MultiOp is done by a compiler or binary translator. The processor freed from the cumbersome task of dependence analysis has to merely execute in parallel the operations contained within a **MultiOp**.

The VLIW is a processor with more then one functional unit and the processor with the multiple functional units has the potential to execute several operations in parallel. If the decision about which operations to execute in an overlapped manner is made at the runtime by the hardware, it is called a superscalar processor. To simplify the superscalar a binary program represents a plan of execution. The processor acts as an interpreter that executes the instructions in the program one at a time. From the point of view of a modern superscalar processor, an input program is more like a representation of an algorithm for which several different plans of execution are possible. Each plan of execution specifies when & on which functional unit each instruction from instruction stream is to be executed. **ILP** processors differ in the manner in which the plan of execution is derived, but it typically involves both the compiler & the hardware. In the current breed of high performance processors like the Intel Pentium and the Ultra Sparc, the compiler tries to expose parallelism to the processor by means of several optimizations the net result of which is to place as many independent operations as close to each other in the instruction at a time analyses the Dependences between instructions and keeps track of the availability of data & hardware resources for each instruction. It tries to schedule each decisions are often further complicated by the fact that operations like memory accesses often have variable latencies that depend on whether a memory access hits in the cache or not. Since such processors decide which functional unit should be allocated to which instruction as execution progresses, they are said to be dynamically scheduled.

VLIW processor and superscalar implementations of traditional instruction sets share some characteristics, like multiple execution units and the ability to execute multiple operations simultaneously. The techniques used to achieve high performance, however are different because the parallelism is explicit in VLIW instruction but must be discovered by hardware at run time by superscalar processors. It is simpler then CISC and RISC as it has most simplified hardware, but it requires more compiler's support i.e. more powerful complier.

As VLIW basically is built on the **CISC** and **RISC** architecture or we can say superscalar processors though it is cheaper and simpler then both of them but for the basic understanding we should have some idea of CISC and RICS architectures. From the larger perspective RISC, CISC and VLIW architectures have more similarities than differences. The differences that exists, have however profound effects on the implementation of these architectures.

1.6. COMPARISION OF CISC SUPERSCALAR, RISC SUPER SCALAR AND VLIW:

These architectures use the traditional state machine model of computation. Each instruction effects an incremental change in the state (memory, registers) of the computer, and the hardware fetches and executes instructions sequentially until a branch instruction causes the flow of control to change.

1.6.1. CISC:

CISC instructions vary in size, often specify a sequence of operations and can require serial (slow) decoding algorithms.CISCs tend to have few register may be special-purpose , which restricts the ways in which they can be used. Memory references are typically combined with the other operations (such as add memory to register).

1.6.2. RISC:

RISC instructions specify simple operations, are fixed in size, and are easy (quick) to decode. RISC architectures have a relatively large number of general-purpose registers. Instructions can reference main memory through simple load-register –from-memory and store –register-to-memory operations. RISC instructions sets do not need microcode and are designed to simplify pipelining.

1.6.3. VLIW:

VLIW instructions are like RISC instruction can be longer to allow them to specify multiple independent simple operations. A VLIW instruction can be thought of as several RISC instructions joined together. VLIW architectures tend to be RISC-like in most attribute.

1.6.4. Example:

C-language code:

```
  function (j)
{

long j;

long I;

 j=j+i;
}
```

1.6.4.1. CISC's instruction:

Add 4[r1] <-r2

addMR	4	r1	r5

1.6.4.2. RISC's instruction:

Load r5<-4[r1]
Add r5<-r5+r2
Store 4[r1] <-r5

Load	r5	r1	4

Add	r5	r5	r2

Store	r5	4	r1

1.6.4.3. VLIW instruction:

Load r5<-4[r1]
Add r5<-r5+r2
Store 4[r1] <-r5

-	-	-	-	-	-	-	-	Load	r5	r1	4
-	-	-	-	add	r5	r5	r2	-	-	-	-
-	-	-	-	-	-	-	-	Store	r5	r1	4

1.7. Implementation COMPARISON:

1.7.1. superscalar CISC, superscalar RISC and VLIW:

The difference between CISC, RISC and VLIW architectures manifest themselves in their respective implementations. High performance RISC, CISC designs are called superscalar implementations. Superscalar in this context simply means “beyond scalar” where scalar means one operation at a time. Thus, superscalar means more than one operation at a time.

1.7.2. CISC AND RISC:

Most CISC instruction sets were designed with the idea that an implementation will fetch one instruction executes its operation fully, and then move on to the next instruction. The assumed execution model was thus serial in nature.
RISC architects were aware of the advantages & peculiarities of pipelined processor implementations, and so designed RISC instruction sets with a pipelined execution model in mind. In contrast to the assumed CISC execution model, the idea for RISC execution model is that an implementation will fetch one instruction, issue it into the pipelined and then move on to the next instruction before the previous one has completed its trip through the pipeline.

The assumed RISC execution model-a pipeline-overlaps phases of execution for several instructions simultaneously, but like the CISC execution model, it is scalar, that is at most one instruction is issued at a time.

For either CISC or RISC to reach higher-level of performance than provided by a single pipeline, a superscalar implementation must be constructed. The nature of a superscalar implementation is that it fetches issues & completes more than one CISC or RISC instruction per cycle.

Some more recent RISC architectures have been designed with superscalar implementations in mind. The most notable examples are the **DEC Alpha** and **IBMPOWER (from which PowerPC is derived).** Nonetheless, superscalar RISC and Superscalar CISC implementations share fundamental complexities, the need for the hardware to discover and exploit instruction-level parallelism.

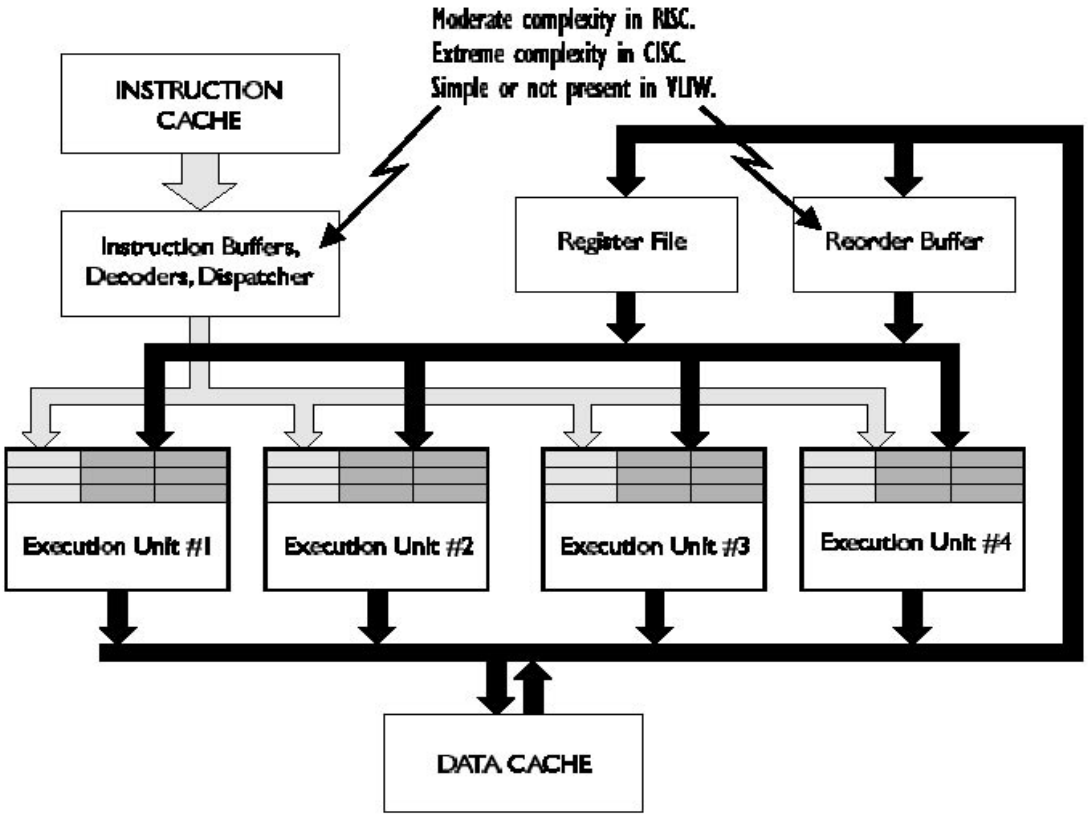


Figure 1.1: high-level block diagram of a superscalar RISC/CISC processor

The implementation consists of a collection of execution units.

- INTEGER ALUs
- FLOATING POINT ALUs
- LOAD/STORE UNITS

- BRANCH UNITS (These are fed from an instruction dispatcher and operands from a register file).

The execution units have reservation stations to buffer waiting operations that have been issued but are not yet executed. The operations may be waiting on operands that are not yet available.

The instruction dispatcher examines a window of instructions contained in a buffer. The dispatcher looks at the instructions in the window and decides which ones can be dispatched to execution units. It tries to dispatch as many instructions at once as possible, i.e. more execution units, require wider windows and a more sophisticated dispatcher.
It is conceptually simple-though expensive to build an implementation with lots of execution units and an aggressive dispatcher, but it is not currently profitable to do so.

The complier for RISC and CISC processor produce code with the certain goals in mind. These goals were typically to minimize code size and runtime. For scalar and very simple superscalar processor implementation, these goals are mostly compatible.

For high performance superscalar implementations on the other hand, the goal of minimizing code size limits the performance that the superscalar implementation can achieve. Performance is limited because minimizing code size results in frequent conditional branches, about every six instructions. Conceptually the processor must wait until the branch is resolved before it can begin to look for parallelism at the target of the branch.

1.8. VLIW architecture detail:

A VLIW implementation achieves the same effect as a superscalar RISC, CISC implementation, but the VLIW design does so without the two most complex parts of a high performance superscalar design.

Because VLIW instruction explicitly specifies several independent operations-i.e. they explicitly specify parallelism-it is not necessary to have decoding and dispatching hardware tries to reconstruct parallelism from a serial instruction stream. Instead of having hardware attempt to discover parallelism, **VLIW processors rely on the complier that generates the VLIW code to explicitly specify parallelism. Relying on the complier has advantage.**

1.8.1. ADVANTAGES:

First the complier has the ability to look at much larger window of instructions than the hardware. For a superscalar processor, a larger hardware window implies larger amount of logic and therefore chip area. At some point, there simply is not enough of either, and window size is constrained. Worse, even before a simple limit on the amount of hardware is reached, complexity may adversely affect the speed of the logic, thus the window size is constrained to avoid reducing the clock speed of the chip. Software windows can be

arbitrarily large. Thus, looking for parallelism in a software window is likely to yield better results.

Second, the complier has the knowledge of the source code of the program. Source code typically contains important information about the program behavior that can be used to help express maximum parallelism at the instruction-set level. A powerful technique called trace-driven compilation can be employed to dramatically improve the quality of code output by the compiler. Trace driven compilation first produces a suboptimal, but correct behavior- which branches are taken, how often, etc.-is then used by the complier during a second compilation to produce code that takes advantage of accurate knowledge of program behavior. Thus, with trace-Driven compilation, the complier has access to some of the dynamic information that would be apparent to the hardware dispatch logic.

Third, with sufficient register, it is possible to mimic the functions of the superscalar implementation's reorder buffer. The purpose of the reorder buffer is to allow a superscalar processor to speculatively execute instructions and then be able to quickly discard the speculatively executed instructions in temporary registers. The complier knows how many instructions will be speculatively executed, so it simply uses the temporary registers along the speculated (predicted) path and ignores the values in those registers along the path that will be taken if the branch turns out to have been miss predicted.

This figure shows a generic VLIW implementation, without the complex reorder buffer and decoding and dispatching logic.

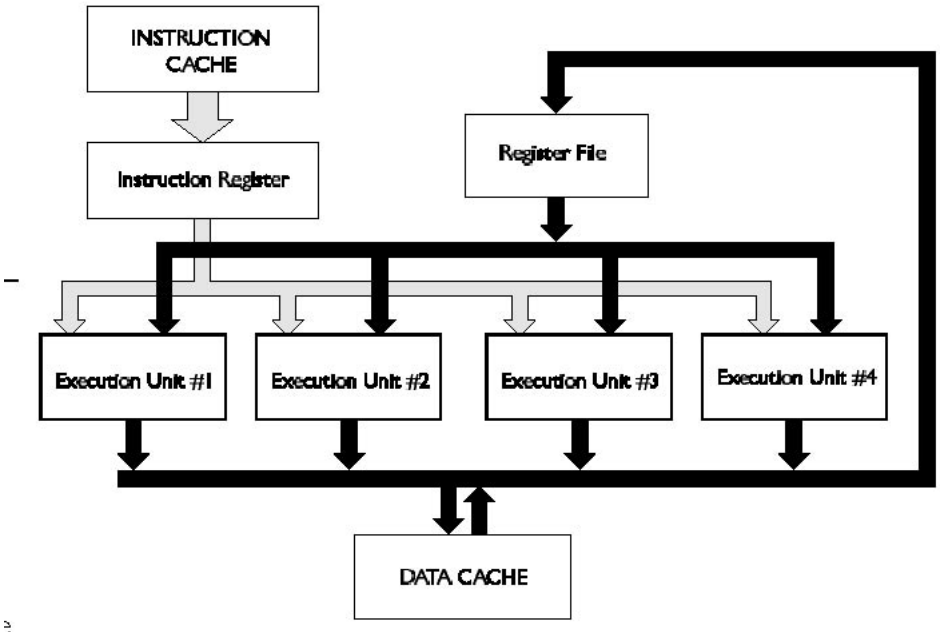


Figure 1.2: A generic VLIW implementation

1.8.2. Target VLIW Processor:

Our target architecture is a VLIW processor augmented with one (or possibly more) RFUs. A VLIW machine is capable of issuing and executing multiple operations per cycle, bundled in a "MultiOp" long-word instruction, and it relies on compile-time scheduling to determine independent operations which can be issued concurrently at execution time. The machine is equipped with multiple functional units so as to exploit the parallelism that has been exposed by the compiler. Many techniques for exploiting instruction-level parallelism have been studied and implemented in optimizing compilers in order to find groups of independent operations in each cycle. Because of the existence of a high number of FUs, advanced VLIW processors are organized in clusters. Every cluster consists of a number of functional units and a register file that these FUs share. Since the cost of register files grows exponentially with the number of read/write ports, the organization in clusters solves the cost problem by keeping the number of ports per register file low. The IS architecture provides special instructions that copy values between register files of different clusters.

Chapter # 2

2. EXPRESSION ADL

2.1. Choosing EXPRESSION ADL:

EXPRESSION is a language supporting architectural design space exploration for embedded Systems-on-Chip (SOC) and automatic generation of a retarget able compiler/simulator toolkit. Key features of this language-driven design methodology include: a mixed behavioral/structural representation supporting a natural specification of the architecture; explicit specification of the memory subsystem allowing novel memory organizations and hierarchies; clean syntax and ease of modification supporting architectural exploration; a single specification supporting consistency and completeness checking of the architecture; and efficient specification of architectural resource constraints allowing extraction of detailed reservation tables for compiler scheduling.

The advent of System-on-Chip (SOC) technology has resulted in a paradigm shift for the design process of embedded systems employing programmable processors with custom hardware. Modern system-level design libraries frequently consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIWs and hybrid ASIPs. Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers, and partitioned register files), allowing a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application. The embedded SOC designer is thus faced with the dual tasks of **1) rapidly exploring and evaluating different architectural and memory configurations, and 2) using a cycle-accurate simulator and retarget able optimizing compiler to adapt the application architecture with the goal of meeting system-level performance, power and cost objectives.**

Furthermore, shrinking time-to-market cycles create an urgent need to perform the traditionally sequential tasks of hardware and software design in parallel. An effective embedded SOC co design flow must therefore support automatic software toolkit generation, without loss of optimizing efficiency. This has resulted in a paradigm shift towards a *language-based design methodology* for embedded SOC optimization and exploration. Consequently there is tremendous interest in using Architectural Description Languages (ADLs) to drive design space exploration and automatic compiler/simulator toolkit generation.

As with an HDL-based ASIC design flow, several benefits accrue from a language-based design methodology for embedded SOC exploration, including the ability to perform (formal) verification and consistency checking, to modify easily the target architecture and memory organization for design space exploration, and to drive automatically the backend toolkit generation from a single specification.

EXPRESSION, an ADL that effectively supports these dual goals of SOC exploration, as well as automatic generation of a high-quality software toolkit for embedded SOC.

2.2. Goals and Approach

SOC designers spend a lot of time and effort exploring candidate processor architectures. The availability of a variety of processor core IP libraries (including DSP, VLIW, SS/RISC and ASIP) presents the system designer with a large exploration space for the choice of base processor architecture. Thus, tool-kits which allow the designer to perform rapid exploration of various processor alternatives are necessary. These tool-kits must provide the designer with quantitative performance measurements in order for him to perform intelligent tradeoffs. Furthermore, the stringent performance, power, code density, and cost constraints mandated by modern embedded systems necessitate the development of a high-quality software tool-kit, including, at a minimum, a cycle-accurate simulator, and an optimizing Instruction-Level Parallelism (ILP) compiler that can exploit novel memory organizations.

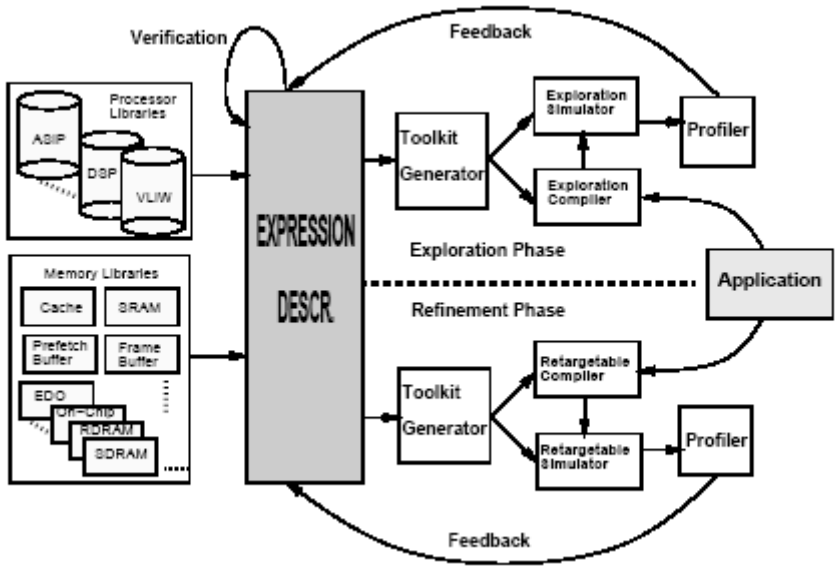


Figure 2.1: EXPRESSION Design Flow

The system designer also requires the ability to customize the base processor by changing parameters of the processor core (e.g. number of functional units, operation latencies). The memory-intensive nature of many embedded applications (e.g. multimedia and network) further exacerbates the traditionally critical memory bottleneck. This requires the ability to explore (and optimize for) novel on-chip and off-chip memory organizations and hierarchies to improve memory bandwidth (examples include the use of on-chip DRAM, frame-buffers, queues, novel cache hierarchies, etc.). An important aspect of such an exploration (not taken into account by most other approaches) is the ability to also customize the compiler concurrently with the processor such that a "best-fit" is

obtained. Figure 1 shows language-based design methodology using EXPRESSION. An EXPRESSION description of an embedded SOC architecture can be used in two modes. In the *Exploration Phase*, the system designer explores and evaluates different base processor candidates (selected from the Processor Libraries), and different memory organizations and hierarchies (with components selected from the Memory Libraries). In the exploration phase, the toolkit generator is used to produce an *Exploration Simulator* and a *Exploration Compiler*. The goal here is to support rapid Design Space Exploration (DSE) with fast (possibly functional) simulation, and using the compiler in an estimation mode for comparative evaluation of candidate base processors and memory organizations. In the *Refinement Phase*, the EXPRESSION description is used to generate a cycle-accurate simulator and an optimizing ILP compiler that allows the system designer to tune the base processor characteristics, as well as to tune the memory subsystem hierarchy. EXPRESSION was designed to provide a natural and easy to specify mechanism for capturing the information needed to support this ADL-based design space exploration and software toolkit generation methodology. As shown in Figure 1, EXPRESSION facilitates the automatic generation of an optimizing, compiler and simulator. The retarget able compiler exploits the parallelism and pipelining available, while the simulator provides accurate timing and utilization information. Furthermore, since the description of complex processors is cumbersome and error-prone, EXPRESSION provides the ability to perform consistency checking and verification of the input specification.

2.3. Keys features of EXPRESSION ADL are

- Ease of specification and modification of architecture from the GUI.
- Mixed behavioral/structural representation supporting a natural, concise specification of the architecture.
- Explicit specification of the memory subsystem allowing novel memory organizations and hierarchies.
- Efficient specification of architectural resource constraints allowing extraction of detailed Reservation Tables (RTs) for compiler scheduling.

2.3.1. Release 1.0 of the EXPRESSION toolkit supports the following exploration features:

2.3.1.1. ISA Exploration

➔ Adding new complex instructions.
➔ Changing register accessibility.

2.3.1.2. Pipeline Exploration

- ➔ Adding a single/multi cycle functional unit.
- ➔ Adding a new pipelined functional unit.
- ➔ Deleting a pipeline path.

2.3.1.3. Memory Subsystem Exploration

- ➔ Modifying access times of caches/ memories.
- ➔ Modifying associatively of caches.
- ➔ Changing sizes of caches/memories.
- ➔ Adding new memory components in the memory subsystem.

2.4. Explanation:

There are two main components in **EXPRESSION**: the **EXPRESS** compiler and the **SIMPRESS** simulator. This tool-kit is implemented with Microsoft Visual C++ 6.0 on an **i686** machine running Microsoft Windows XP. It has also been tested on Microsoft Windows NT and Windows 2000.

A Sparc / Solaris 2.7 machine is also required for preprocessing an input application in C using a GCC-based front-end.

An application in C is preprocessed by the GCC based front-end to generate front-end files, <filename>.procs and <filename>.defs using the generic machine Instruction Set Architecture (ISA). **EXPRESS** then reads the front-end files, builds an Intermediate Representation (IR) amenable to different optimizations and targets the architecture described in an EXPRESSION ADL (Architecture Description Language) description. The output of **EXPRESS** is a special assembly file named <filename>_DUMP_IR_AFTER_REGALLOC.txt.

SIMPRESS reads the special assembly file, simulates the running of assembly on an architecture template generated from the ADL description and finally generates area, power, and performance numbers including cycle count and memory usage statistics. The purpose of the simulator is to assess the efficacy of the code generated by the EXPRESS compiler for the given architecture.

The **EXPRESSION** tool-kit also comes with a GUI front-end to schematically enter the architecture connectivity and instruction set description. The GUI back-end converts the schematic description and instruction set description into EXPRESSION ADL format.

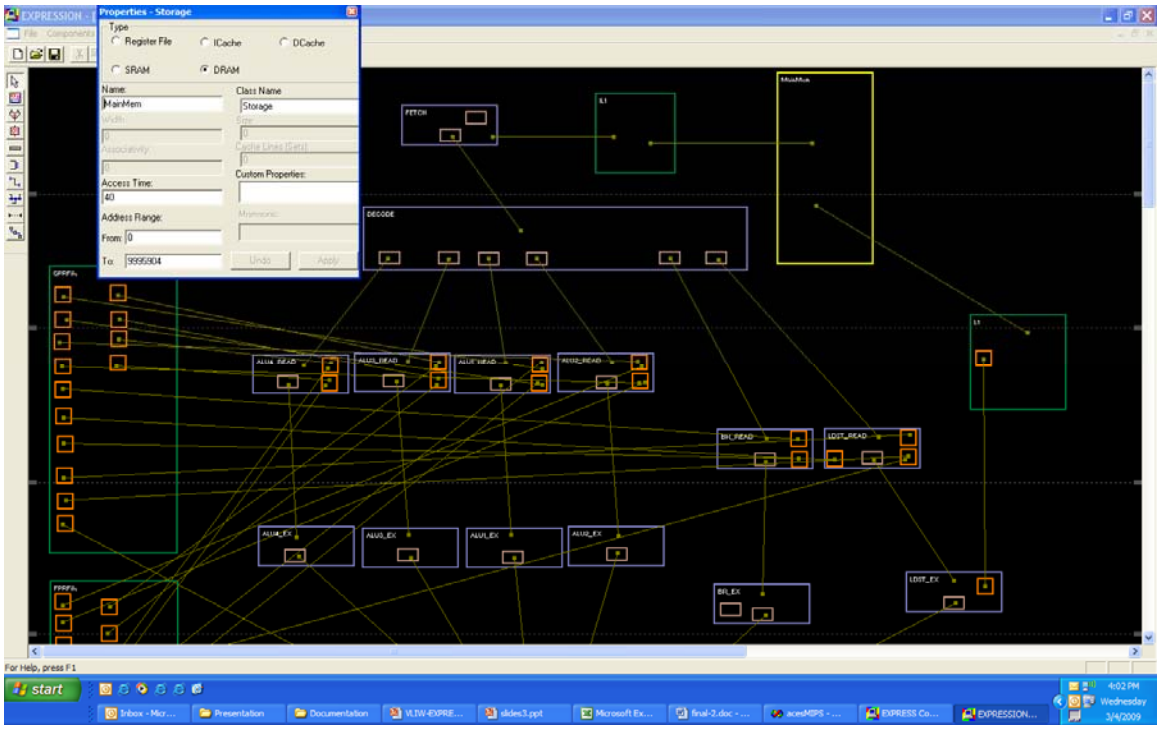


Figure 2.2: EXPRESSION GUI Screen Shot

2.4.1. Expression Design Flow:

The **EXPRESSION** aims to achieve exploration of *Systems-On-Chip* (SOCs) with programmable processor cores and novel memory hierarchies. Effective exploration of such, the compiler and the target application SOCs is possible by considering the interaction between the processor architecture. In this project I use **EXPRESSION**, an Architecture Description Language (ADL), to specify the processor-memory architecture. **EXPRESS**, a highly optimizing, Instruction-Level-Parallelizing (ILP) compiler, and **SIMPRESS**, a cycle-accurate, structural simulator from EXPRESSION. EXPRESSION, EXPRESS and SIMPRESS are integrated under a visual environment, **V-SAT** (Visual Specification and Analysis Tool), to aid rapid *Design Space Exploration* (DSE).

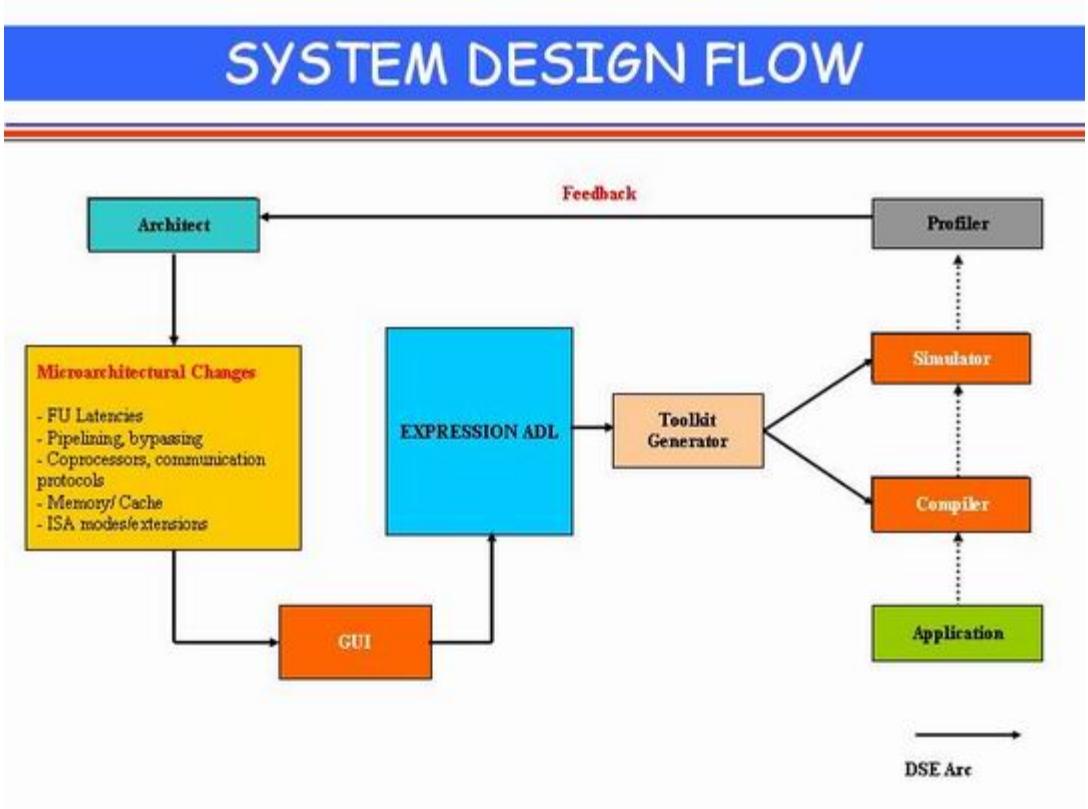


Figure 2.3: EXPRESSION System Design Flow

EXPRESSION was designed with the dual goal of allowing processor description for fast DSE and for automatic generation of detailed/accurate simulation/compilation tools. The novel features of **EXPRESSION** include:

- Integration of the Instruction-Set and Structure to avoid redundancy in specification,
- Automatic generation of resource constraints (as reservation tables),
- Constructs for explicitly specifying novel and traditional memories

EXPRESS was developed with the goal of providing a retarget able compiler platform for Embedded-System/System-on-Chip development. The **EXPRESS** retarget able compiler takes in C programs and produces a highly optimized (and parallel) target specific code using state-of-art Instruction-Level Parallelism (ILP) techniques. The compiler features an extensive set of integrated transformations to perform the traditional compiler tasks of code selection, instruction scheduling and register allocation and memory aware optimizations.

SIMPRESS is a retarget able, cycle-accurate, structural simulator that can be used to evaluate the architecture, the application and the effectiveness of the compiler transformations. It features an extensive set of statistic collector agents that are used to gather information such as resource usage, hazard count, inner-loop execution time, etc.

V-SAT provides a visual environment to graphically specify the architecture and perform Architectural DSE in an intuitive manner. The **EXPRESSION** description of the processor can be automatically generated from the **V-SAT** specification.

2.4.2. System Requirements for Expression ADL:

The EXPRESSION toolkit is available as downloadable source code which needs to be compiled on a host machine before it can be executed. To run EXPRESSION, you will require a machine running Windows and Visual C++ installed on it. The EXPRESSION toolkit has been tested on the following systems:

OS: Microsoft Windows XP Professional, Windows 2000 Server
System Type: X86-based PC
Processor: x86 Family 15 Model 1 Stepping 2 Genuine Intel ~1 Ghz
Total Physical Memory: 512.00 MB
Total Virtual Memory: 1.72 GB
Page File Space: 1.22 GB
Development Platform: Visual C++ 6.0 Enterprise Edition

Additionally, you will also require access to a SUN Sparc workstation if you plan to compile your own C applications.

EXPRESSION can capture a processor memory architecture description and generate a compiler and simulator automatically from this description. Previously many case studies have been undertaken with the goal of architecture exploration using a framework that involves manual specification of architecture in EXPRESSION and subsequent manual intervention at various steps in the ADL to the back-end compiler and simulator flow. In this technical report I present a framework for capturing the EXPRESSION description for processors using a GUI front end tool and transforming the generated description into intermediate code to be used by the compiler and simulator engines.

This automated flow requires no manual intervention at any point and allows rapid Design Space Exploration by modifying the graphical specification of the processor using the GUI.

2.4.3. EXPRESSION ADL to compiler and simulator back-end flow:

The next step after creating the EXPRESSION description file is to process it to generate information for the retarget able compiler and simulator. The EXPRESSION parser builds an IR representation from the description file. This internal representation is processed to generate C++ intermediate files.

2.4.4. EXPRESS retarget able complier

The important phases of EXPRESS compiler are:

- Instruction Selection
- Scheduling
- Register Allocation

2.4.4.1. Instruction Selection

The TREE_MAPPING section in the ADL is used by the Instruction Selection phase to convert a set of generic instructions to a set of target instructions. The order in which the mapping rules are specified determines the priorities of the mappings. The scheduling / pipelined trailblazing is based on the connectivity of the units specified.

2.4.4.2. Scheduler

The scheduler automatically generates the reservation tables from the specified connectivity and maps the target operations effectively. In this way, the scheduler is able to exploit the parallelism present in the architecture.

2.4.4.3. Register Allocation

The register allocation is derived from Chaitin's algorithm, but is based on register classes specified in VAR_GROUPS section. The OPERANDS_MAPPING section maps a register class to the desired set of registers in the register file and thus enables the user to specify a partitioned register file. This is very useful in exploration, as the user will be able to play around with the number of registers available for different operations. EXPRESS has the capability to dump the Intermediate Representation (IR) at any stage for debugging. The backend of EXPRESS is capable of generating the assembly code based on the syntax specified in the ASMFORMAT section. It also generates the dump of IR in a special format recognized by the retarget able simulator called SIMPRESS.

2.4.5. SIMPRESS retarget able simulator:

The intermediate files give information to the simulator about the structural components, pipeline structure, memory hierarchy and how they are linked together in the system. The simulator engine maps appropriate functionality to the components. For example, a functional unit specified to be of type Decode Unit in the GUI has the generic Decode functionality mapped to it. The rest of the attributes of the Decode unit are used to tune this mapping to fit the architecture being modeled – for example changing the Decode unit's reservation buffer size, changing the maximum number of instructions it can issue etc. Once the structural net-list is created and interconnections established, the simulator is ready to accept instructions generated by the EXPRESS compiler for execution and profiling. The SIMPRESS simulator reads an IR dump file that has been generated by the EXPRESS compiler and which contains instructions for execution.

2.5. GUI driven specification capture in EXPRESSION:

While Architecture Description Languages are certainly a powerful mechanism for describing complex processor based systems, there are a few drawbacks in the current textual ADL approaches. Textual descriptions can be tedious and often non intuitive for specifying architectures. The length and repetitive nature of these descriptions also increases the possibility of errors in the specification. To circumvent these limitations I have incorporated a graphical front end tool (V-SAT) that can capture the architecture and data paths of the processor and the memory subsystem, as well as the instruction set description and transform it into a textual EXPRESSION ADL description that is subsequently used in the automatic compiler and simulator tool kit generator phase.

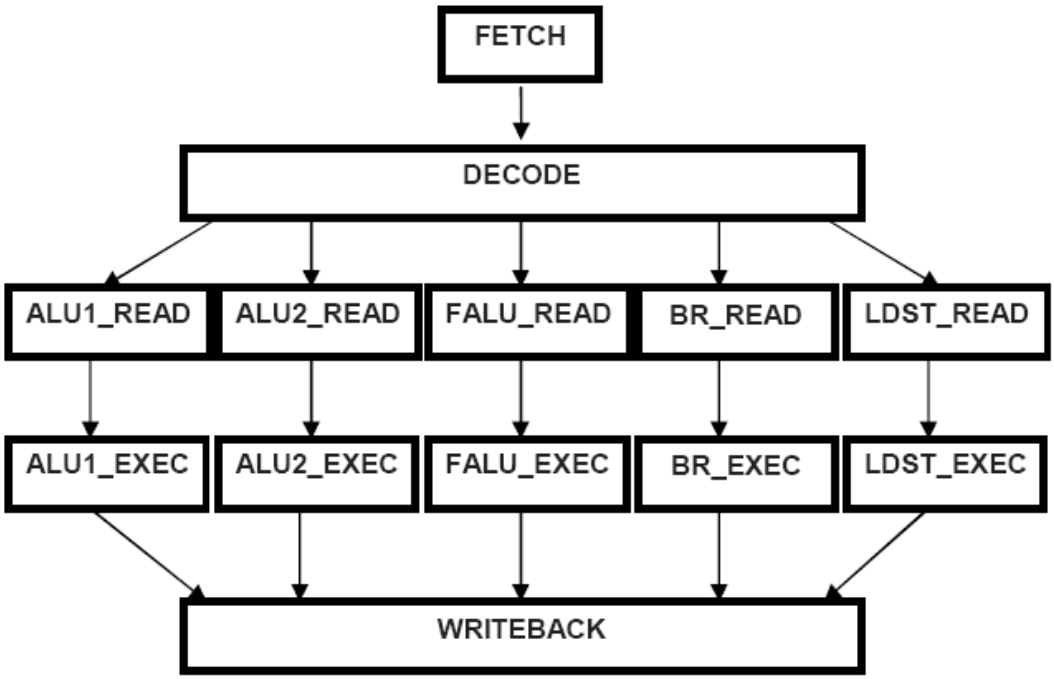


Figure 2.4: Simplified processor architecture

The primary motivation behind graphical specification tool is that it will allow the designer to quickly and accurately specify a particular design configuration and perform Design Space Exploration. For example it is fairly easy to experiment with adding or deleting pipeline stages in the design with the visual specification, generate the EXPRESSION ADL description and then generate a compiler and simulator to test the implications of the design decision with application programs or test bench suites.

2.6. The Processor Architecture

In this report I will demonstrate framework of a VLIW like processor architecture (shown in Fig. 2.4). The architecture contains five pipeline stages – fetch, decode, operand read, execute and write back. There are multiple issue paths corresponding to 4 ALU units, a branch unit and a load/store unit. The memory hierarchy consists of two L1 data caches for instructions and data, a unified L2 cache and a DRAM main memory. There is a 128-bit wide general purpose register file containing 32 registers.

FETCH
DECODE
ALU1_READ ALU2_READ ALU3_READ ALU4_READ FALU_READ BR_READ LDST_READ
ALU1_EXEC ALU2_EXEC ALU3_EXEC ALU4_EXEC FALU_EXEC BR_EXEC LDST_EXEC
WRITEBACK

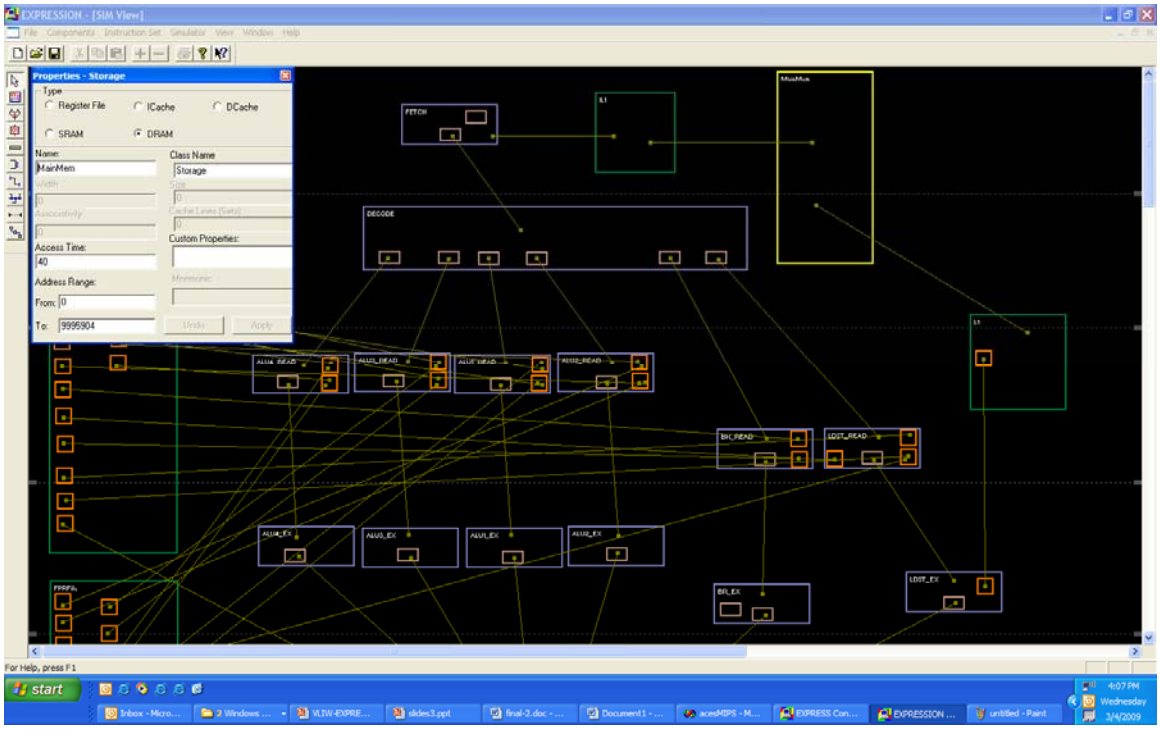


Figure 2.5: VSAT-GUI layout for the proposed architecture

2.6.1. Sections in EXPRESSION:

The EXPRESSION description is composed of two main sections - Behavior and Structure. The Behavior (or Instruction Set) section is divided into three subsections: Operations, Instruction and Operation Mappings.
The Structure section is also divided into three subsections: Components, Pipeline and Data-Transfer Paths and the Memory Subsystem. Each of these subsections is captured using the new version of the VSAT-GUI, stored internally in appropriate data structures and then used to create the textual ADL description file of the processor-memory architecture which will be used by subsequent stages in the framework. The alternative to using the graphical user interface is to specify the entire description using a text editor which would be cumbersome and time consuming. The GUI hides EXPRESSION ADL syntax details allowing the architect to specify the system details quickly and precisely without knowing a whole lot about the EXPRESSION language. Described below are the various EXPRESSION sections and how they are captured using the VSAT-GUI and then used to generate code in the ADL description file.

2.7. Operations Specification:

An instruction in the architecture refers to a VLIW instruction which is composed of more than one operation. All the operations supported in the instruction set are described in this subsection. Each operation is described in terms of its op code, operands, behavior, assembly format and IR dump format. Each operand is classified as either source or destination and is associated with a list of register files which it can access. These operations are grouped together into operation groups, so as to minimize duplication in writing when associating valid operations to functional units.

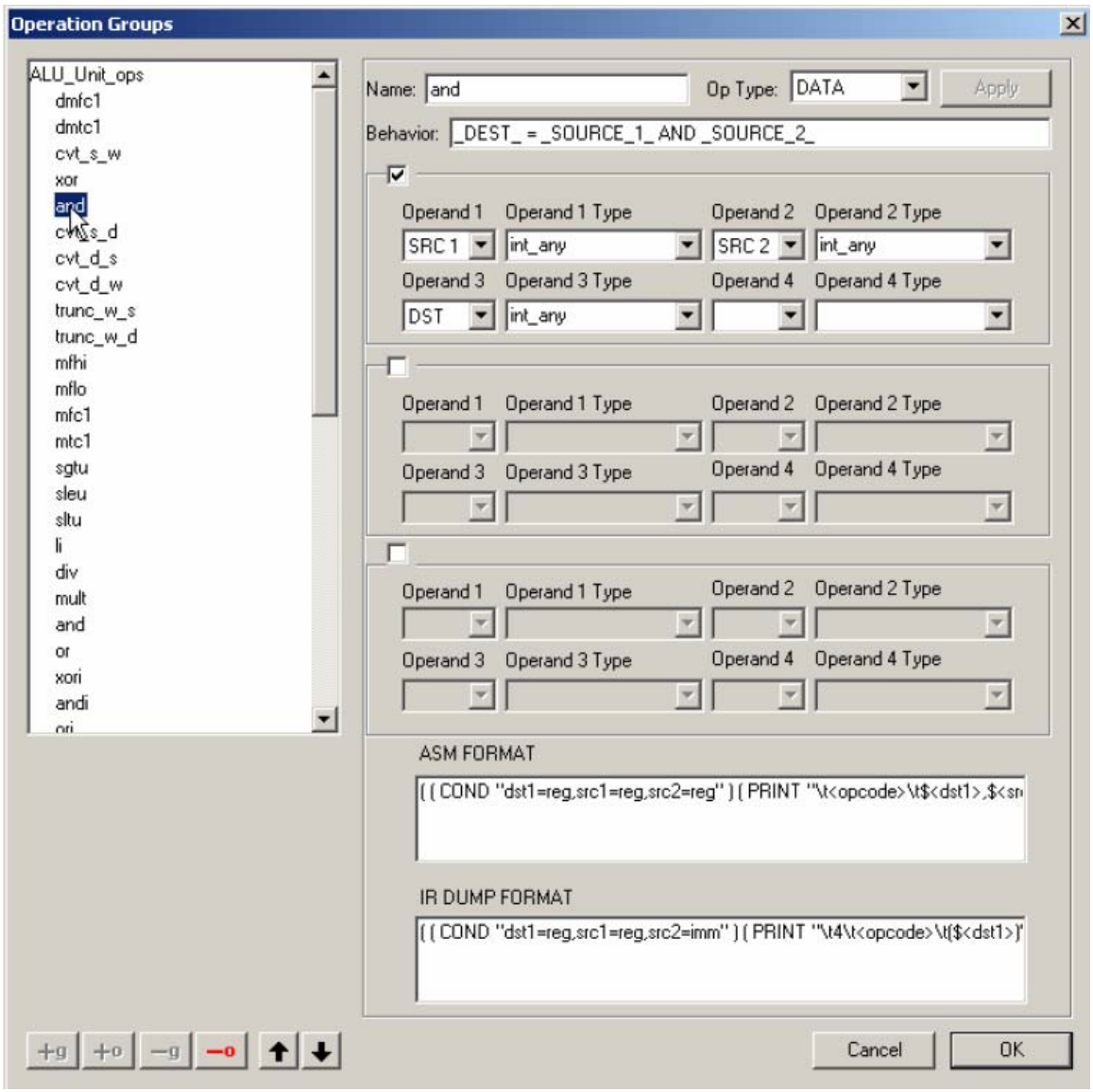


Figure 2.6: Operand Types for an Operation "and"

In the GUI, selecting the **set OP_GROUPS** option in the **Instruction Set** menu brings up the **Operation Groups** dialog box which allows the specification of the operations and their groupings. Fig. 2.6 shows this dialog box when the 'and' instruction is selected. This instruction is part of the ALU_Unit_Ops group, which contains other ALU operations as well. The behavior field indicates that the destination will contain the bitwise AND of the values in the source 1 and source 2 registers. The figure also shows the type of the source and destination registers as 'int_any' which indicates that they are of integer type. The ASM FORMAT textbox is used to specify the standard assembly dump format for the operation while the IR DUMP FORMAT specifies the assembly dump format in a form that is expected by the simulator. The groups and their children operations are stored in a tree structure internally, with all the attributes of the operations stored at the nodes corresponding to the operations. This code generated in

EXPRESSION for the ‘and’ instruction shown in Fig. 2.6 consists of two parts – the OPCODE description and the OP_GROUP grouping. This is illustrated below:

```
(OPERATIONS_SECTION
…
(OPCODE and
(OP_TYPE DATA_OP)
(OPERANDS (_SOURCE_1_ int_any) (_SOURCE_2_ int_any) (_DEST_ int_any))
(BEHAVIOR "_DEST_ = _SOURCE_1_ AND _SOURCE_2_")
(ASMFORMAT ( ( COND "dst1=reg,src1=reg,src2=reg" ) ( PRINT "\t<opcode>\t$<dst1>,$<src1>,$<src2>\n" ) )
)
(IRDUMPFORMAT ( ( COND "dst1=reg,src1=reg,src2=imm" ) ( PRINT "\t4\t<opcode>\t($<dst1>)\t($<src1>,<src2>)\n" )
)
)
…
(OP_GROUP ALU_Unit_ops
 (OPCODE dmfc1 dmtc1 cvt_s_w xor and cvt_s_d cvt_d_s cvt_d_w trunc_w_s trunc_w_d mfhi mflo mfc1 mtc1 sgtu sleu sltu li div mult and or xori andi ori li_s li_d sgeu sne seq sgt sle slt sge sla sll sra srl move subu nop addu)
 )
)
…
)
```

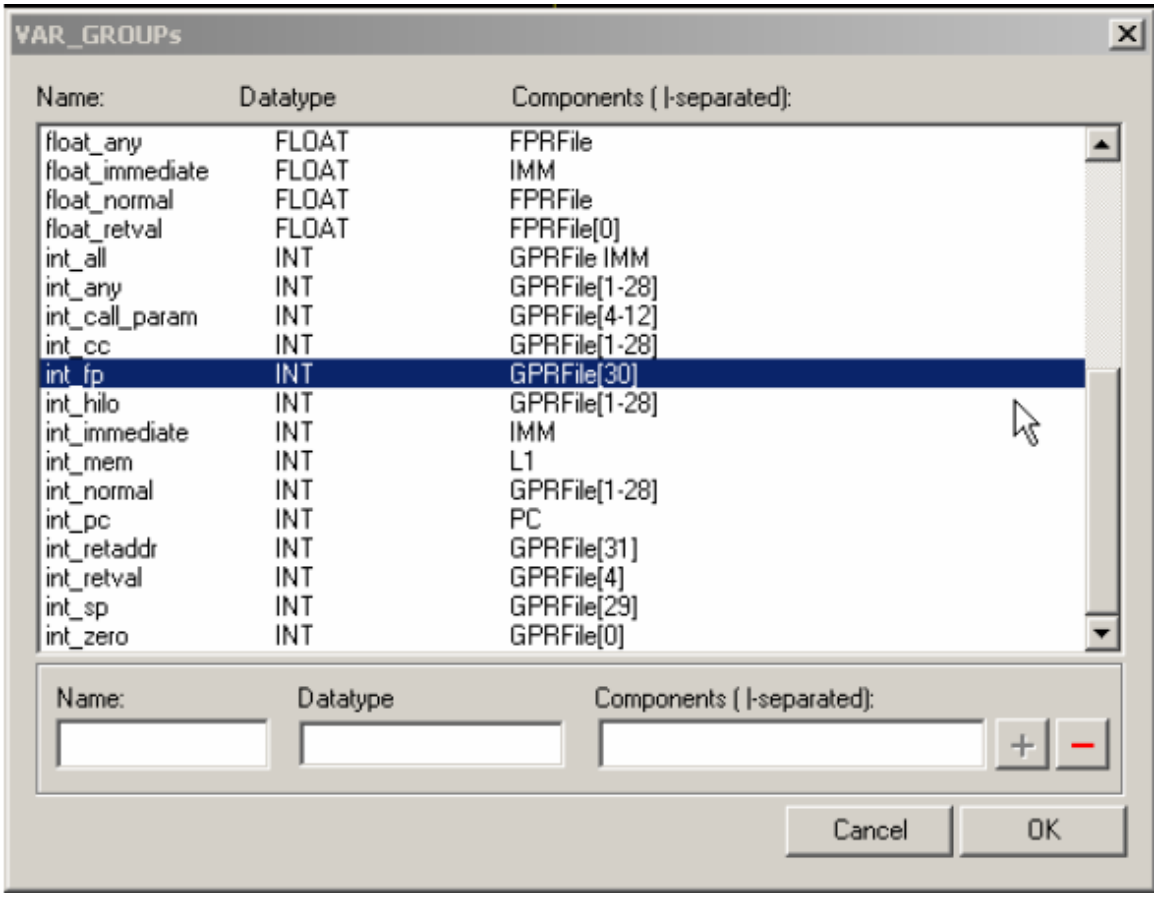


Figure 2.7: Setting VAR_GROUPS

The **set VAR_GROUPS** option in the **Instruction Set** menu brings up the **VAR_GROUPs** dialog box (Fig. 2.7) which allows specification of the accessible register file lists for operands in the operations. The target registers are classified into var_groups based on their data types and mappings with the var_groups in generic register files. For instance, the entry 'int_hilo' is a grouping of registers that are used to hold the output of a multiplication operation. The var_group 'int_fp' refers to the register used as a frame pointer. This section is stored internally in the form of a list with the various register file groups forming the elements of the list. The code generated in EXPRESSION is given below:

```
(OPERATIONS_SECTION
 (VAR_GROUPS
  (any_pc        (DATATYPE  INT)   (REGS PC))
  (double1_retval       (DATATYPE  DOUBLE)     (REGS FPRFile[0]))
  (int_fp        (DATATYPE  INT)   (REGS GPRFile[30]))
  (any_retaddr (DATATYPE  INT)   (REGS GPRFile[31]))
  (double_any  (DATATYPE  DOUBLE)     (REGS FPRFile[0 2 4 6 8 10 12 14 16 18 20
22 24 26 28 30]))
  (float_normal(DATATYPE  FLOAT)        (REGS FPRFile))
```

```
 (int_normal   (DATATYPE  INT)   (REGS GPRFile[1-28]))
 (double1_normal      (DATATYPE  DOUBLE)     (REGS FPRFile[0 2 4 6 8 10 12 14
16 18 20 22 24 26 28 30]))
 (float_all      (DATATYPE  FLOAT)       (REGS FPRFile IMM))
 (int_call_param       (DATATYPE  INT)   (REGS GPRFile[4-12]))
 (double_all   (DATATYPE  DOUBLE)     (REGS FPRFile IMM))
 (double_immediate   (DATATYPE  DOUBLE)     (REGS IMM))
 (double2_normal      (DATATYPE  DOUBLE)     (REGS FPRFile[1 3 5 7 9 11 13 15
17 19 21 23 25 27 29 31]))
 (float_any     (DATATYPE  FLOAT)       (REGS FPRFile))
 (float_retval  (DATATYPE  FLOAT)       (REGS FPRFile[0]))
 (int_retval     (DATATYPE  INT)   (REGS GPRFile[4]))
 (int_sp         (DATATYPE  INT)   (REGS GPRFile[29]))
 (any_fp         (DATATYPE  INT)   (REGS FP))
 (any_hilo       (DATATYPE  INT)   (REGS HILO))
 (int_hilo       (DATATYPE  INT)   (REGS GPRFile[1-28]))
 (any_cc         (DATATYPE  INT)   (REGS CC))
 (int_all        (DATATYPE  INT)   (REGS GPRFile IMM))
 (int_pc         (DATATYPE  INT)   (REGS PC))
 (float_immediate      (DATATYPE  FLOAT)       (REGS IMM))
 (int_immediate        (DATATYPE  INT)   (REGS IMM))
 (any_sp         (DATATYPE  INT)   (REGS SP))
 (int_retaddr    (DATATYPE  INT)   (REGS GPRFile[31]))
 (int_cc         (DATATYPE  INT)   (REGS GPRFile[1-28]))
 (int_mem        (DATATYPE  INT)   (REGS L1))
 (any_call_param       (DATATYPE  INT)   (REGS GPRFile[4-12]))
 (double2_retval       (DATATYPE  DOUBLE)     (REGS FPRFile[1]))
 (int_any        (DATATYPE  INT)   (REGS GPRFile[1-28]))
 (int_zero       (DATATYPE  INT)   (REGS GPRFile[0]))
)
```

2.7.1. Instruction Description

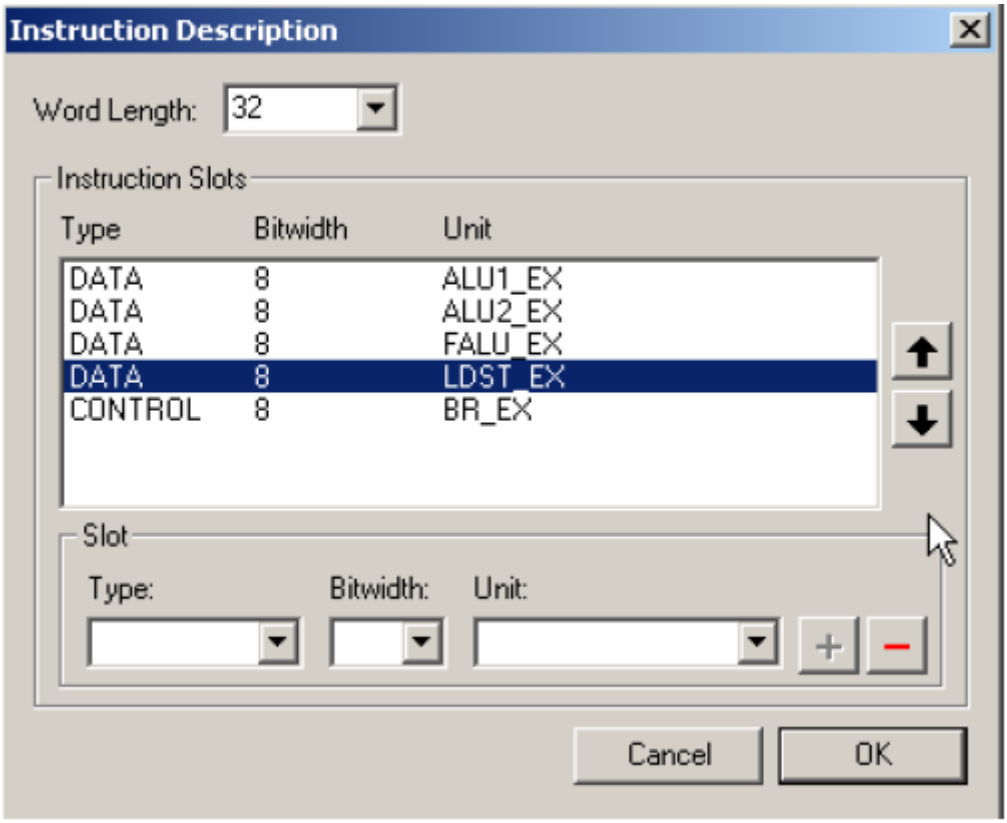


Figure 2.8: VLIW Instruction Template

This subsection captures the parallelism in the architecture by capturing the description of slots in a VLIW instruction. An Instruction contains operations which can be executed in parallel. Each instruction has slots which correspond to a Functional Unit that it will execute on. In the architecture there are 4 slots for data operations (1 addition, 1 subtraction, 1 Multiplication and 1 Shift operation) and 4 slot for Control operations (loop, interrupt, jump and sub routine). A valid VLIW instruction of word length is 76.
The **set Instruction Description** option in the **Instruction Set** menu brings up the Instruction Description dialog (Fig. 2.8) which is used to specify these subsections. The information is stored internally in a list with each element referring to an instruction slot. The code generated in EXPRESSION is illustrated below:

```
(INSTRUCTION_SECTION
  (WORDLEN 64)
  (SLOTS
   ((TYPE DATA) (BITWIDTH 8) (UNIT ALU1_EX))
   ((TYPE DATA) (BITWIDTH 8) (UNIT ALU2_EX))
   ((TYPE DATA) (BITWIDTH 8) (UNIT LDST_EX))
   ((TYPE CONTROL) (BITWIDTH 8) (UNIT BR_EX))
```

```
  ((TYPE DATA) (BITWIDTH 8) (UNIT ALU3_EX))
  ((TYPE DATA) (BITWIDTH 8) (UNIT ALU4_EX))

 )
)
```

2.7.2. Operation Mappings:

This subsection contains information required by the compiler for Instruction Selection and Register Allocation. There are two parts to the Operation Mappings section: Tree Mapping and Operand Mapping.

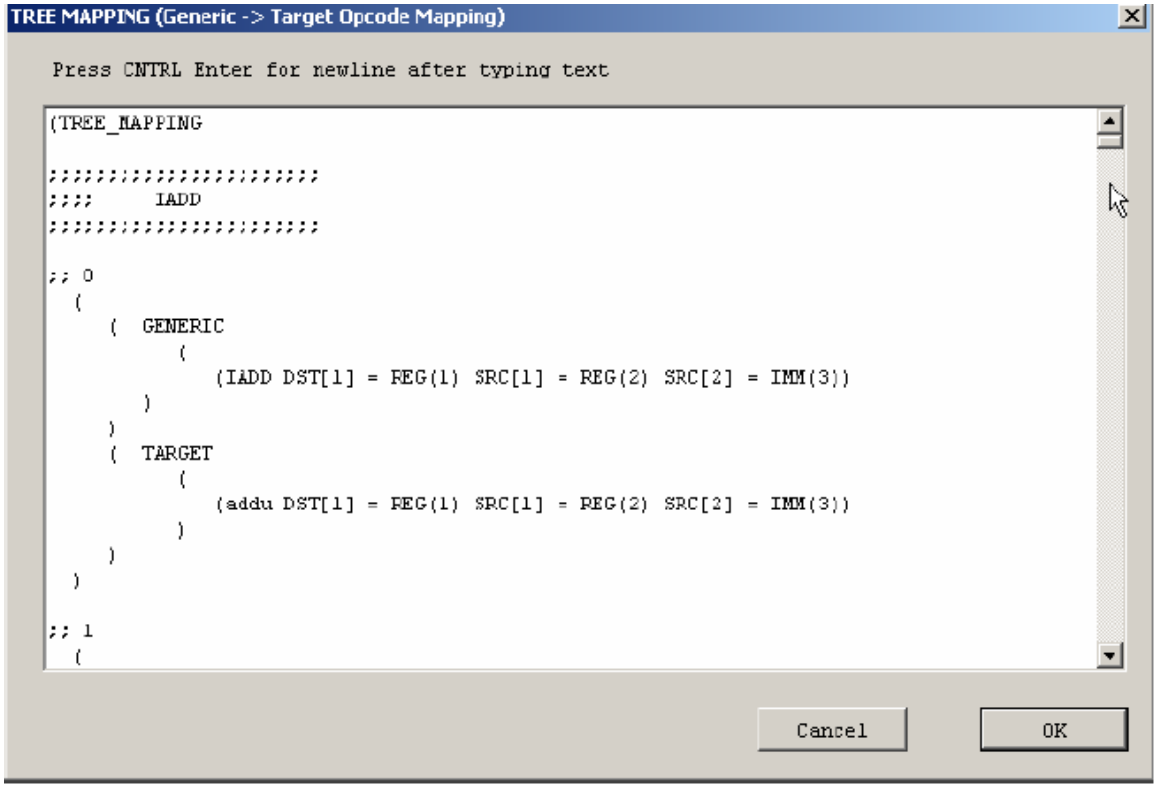


Figure 2.9: Tree Mapping

Tree Mapping is so called because it maps a tree of generic operations to a tree of target operations. The edges of the tree correspond to variable dependencies. This tree mapping could be from a generic compiler operation to a target processor operation, in which case it would be used by the instruction selection algorithm. It is possible to map many operations to one operation, for example in the case of a complex operation like **mac** which is a combination of the generic multiply and add operations.
The **set TREE_MAPPING** option in the **Instruction Set** menu brings up the TREE_MAPPING dialog box (Figure 2.9) which is used to capture this information. The entries are stored in an internal storage structure and reproduced verbatim in the ADL

description file after wrapping the information with the appropriate formatting. The code generated in EXPRESSION is illustrated below:

```
(OPMAPPING_SECTION
…
(TREE_MAPPING

;;;;;;;;;;;;;;;;;;;;;;;
;;;;    IADD
;;;;;;;;;;;;;;;;;;;;;;;

;; 0
  (
    (  GENERIC
            (
          (IADD DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = IMM(3))
      )
    )
    (  TARGET
            (
          (addu DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = IMM(3))
            )
    )
```

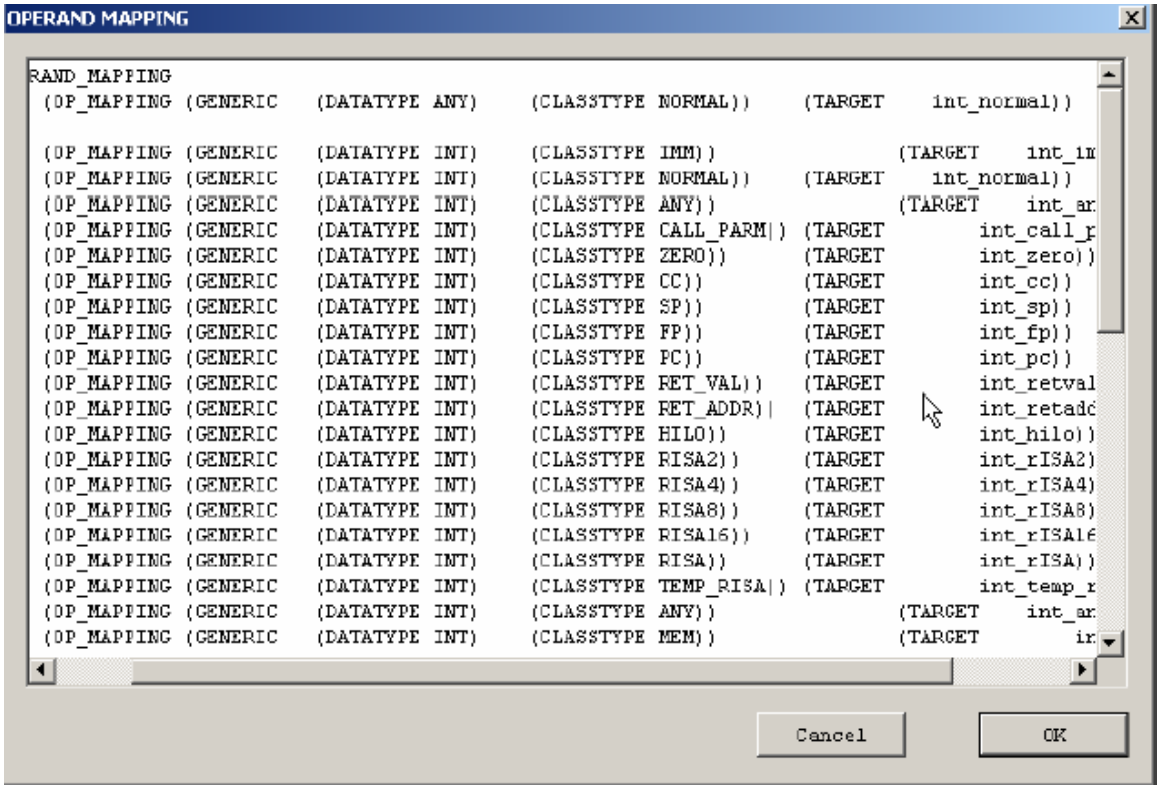


Figure 2.10: Register Class mappings for Operands

In Operand Mapping, the generic register classes are mapped to the target register classes. Each target register class maps to a set of target registers. The **set Operand Mapping** section in the **Instruction Set** menu brings up the Operand Mapping dialog box which is used to specify this information. Just like with the Tree Mapping subsection, the entries are dumped in an internal storage structure and reproduced verbatim in the ADL description file after wrapping the information with the appropriate formatting. The code generated in EXPRESSION is illustrated below:

```
(OPMAPPING_SECTION
  (OPERAND_MAPPING
      (OP_MAPPING (GENERIC (DATATYPE ANY)  (CLASSTYPE NORMAL))
(TARGET  int_normal))

      (OP_MAPPING (GENERIC  (DATATYPE INT)  (CLASSTYPE IMM))
      (TARGET  int_immediate))
      (OP_MAPPING (GENERIC  (DATATYPE INT)  (CLASSTYPE NORMAL))
(TARGET  int_normal))
      (OP_MAPPING (GENERIC  (DATATYPE INT)  (CLASSTYPE ANY))
      (TARGET  int_any))
      (OP_MAPPING (GENERIC  (DATATYPE  INT)  (CLASSTYPE
CALL_PARM)) (TARGET  int_call_param))
```

```
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE
ZERO)) (TARGET int_zero))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE CC))
(TARGET int_cc))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE SP))
(TARGET int_sp))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE FP))
(TARGET int_fp))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE PC))
(TARGET int_pc))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE
RET_VAL)) (TARGET int_retval))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE
RET_ADDR)) (TARGET int_retaddr))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE
HILO)) (TARGET int_hilo))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE ANY))
        (TARGET int_any))
        (OP_MAPPING (GENERIC (DATATYPE INT) (CLASSTYPE
MEM)) (TARGET int_mem))

        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE IMM))
        (TARGET double_immediate))
        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE
DOUBLE1)) (TARGET double1_normal))
        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE
DOUBLE2)) (TARGET double2_normal))
;       (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE
DOUBLE)) (TARGET double_normal))
        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE ANY))
        (TARGET double_any))
        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE
RET_VAL)) (TARGET double1_retval))
        (OP_MAPPING (GENERIC (DATATYPE DOUBLE) (CLASSTYPE
RET_VAL)) (TARGET double2_retval))

        (OP_MAPPING (GENERIC (DATATYPE FLOAT) (CLASSTYPE IMM))
        (TARGET float_immediate))
        (OP_MAPPING (GENERIC (DATATYPE FLOAT) (CLASSTYPE
NORMAL)) (TARGET float_normal))
        (OP_MAPPING (GENERIC (DATATYPE FLOAT) (CLASSTYPE
ANY)) (TARGET float_any))
        (OP_MAPPING (GENERIC (DATATYPE FLOAT) (CLASSTYPE
RET_VAL)) (TARGET float_retval))
```

```
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE
CALL_PARM)) (TARGET any_call_param))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE CC))
(TARGET any_cc))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE SP))
(TARGET any_sp))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE FP))
(TARGET any_fp))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE PC))
(TARGET any_pc))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE
RET_ADDR)) (TARGET any_retaddr))
        (OP_MAPPING (GENERIC (DATATYPE ANY) (CLASSTYPE
HILO)) (TARGET any_hilo))

)
```

2.7.3. Components Specification:

This subsection describes the RT-level components in the architecture. The components can be Pipeline units, Functional units, Storage components, Latches, Ports or Connections. Some of these components have an optional list of attributes, and these are described below. The ADL description file code generated for these structural components is much more involved and requires much more manipulation than in the case of the behavioral specification.

2.7.3.1. Unit (Functional Unit)

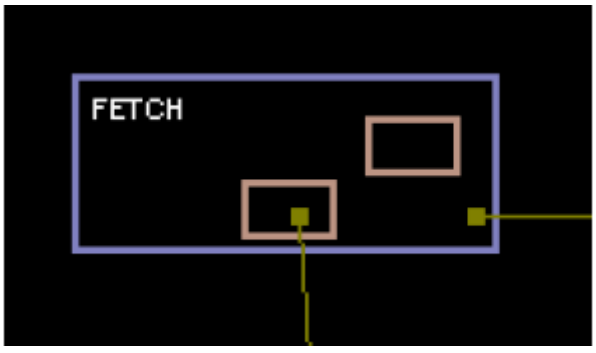


Figure 2.11: Fetch unit

A Functional unit has the following attributes
CAPACITY – size of reservation station
TIMING – time taken for operations to pass through
OPCODES – opcode groups allowed passing through
INSTR_IN – maximum number of simultaneous instructions entering
INSTR_OUT – maximum number of simultaneous instructions leaving
In the GUI, units are graphically represented by purple colored rectangular boxes which can be created from the **Components** menu. Clicking on these boxes brings up the attributes of the unit in the **Properties** window. Fig. 2.11 shows the Fetch unit from the processor architecture. These functional units are stored internally as elements in a global unit list. Each element of this list has the corresponding set of attributes stored with it. Aside from the attributes specified above, each unit also has latches and ports associated with it. The code generated in EXPRESSION is illustrated below:

```
(ARCHITECTURE_SECTION
(SUBTYPE UNIT FetchUnit DecodeUnit OpReadUnit ExecuteUnit BranchUnit
LoadStoreUnit WriteBackUnit ArchUnit ControlUnit )
(FetchUnit FETCH
  (CAPACITY 1)
  (INSTR_IN 6)
  (INSTR_OUT 6)
  (TIMING (all 1))
  (OPCODES all)

  (LATCHES (OUT FetDecLatch) (OTHER pcLatch))
 )

  (InstStrLatch FetDecLatch
  )

  (PCLatch pcLatch
  )


 (DecodeUnit DECODE
  (CAPACITY 12)
  (INSTR_IN 4)
  (INSTR_OUT 4)
  (TIMING (all 1))
  (OPCODES all)
  (LATCHES (OUT DecAlu1ReadLatch) (OUT DecAlu2ReadLatch) (OUT
DecLdStReadLatch) (OUT DecBrReadLatch) (OUT DecAlu3ReadLatch))
  (LATCHES (IN FetDecLatch))
 )

  (InstructionLatch DecAlu1ReadLatch
```

```
 )

 (InstructionLatch DecAlu2ReadLatch
 )

 (InstructionLatch DecLdStReadLatch
 )

 (InstructionLatch DecBrReadLatch
 )

 (InstructionLatch DecAlu3ReadLatch
 )


(OpReadUnit ALU1_READ
 (CAPACITY 1)
 (INSTR_IN 1)
 (INSTR_OUT 1)
 (TIMING (all 1))
 (OPCODES ALU_Unit_ops FALU_Unit_ops)

 (LATCHES (OUT Alu1ReadExLatch))
 (LATCHES (IN DecAlu1ReadLatch))
 (PORTS Alu1ReadPort1 Alu1ReadPort2)
)
```

Here a Fetch unit is declared which can fetch four instructions simultaneously and which has two latches associated with it corresponding to interfaces with other components – FetDecLatch is used to communicate data to the Decode unit while pcLatch is used to interface with the program counter (see Fig. 2.1). InstStrLatch and PCLatch are the types of these latches respectively.

2.7.3.2. Storage (Cache / Memory / Register File)

Storage components are used to represent caches, main memory, buffers and register files in the design. The attributes and connectivity among these storage components is specified in the memory subsystem. However, ports associated with a storage component and used to connect to functional units are specified in this subsection. The GUI however allows specification of storage component information in a unified intuitive manner and then partitions it at the time of writing the ADL description file. Refer to the memory subsystem section.

2.7.3.3. Ports

Functional units and storage components can have ports associated with them. In the GUI, small orange colored square boxes represent these ports. They can be created from the **Components** menu. Clicking on these boxes brings up the attributes of the port in the **Properties** window. These ports are used to connect functional units to storage components. Ports associated with a unit are placed inside the rectangular region of the unit while those associated with storage components are placed within storage boxes in the layout.

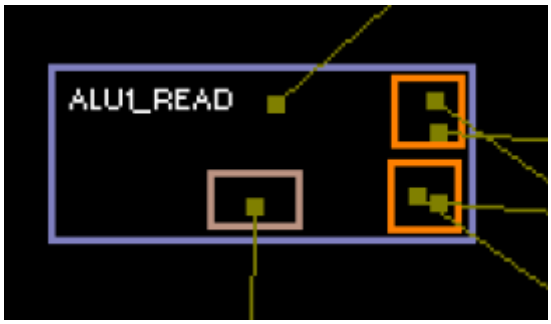


Figure 2.12: Two ports of the ALU1_READ unit

Internally, a global list of ports is maintained. At the time of generation of the ADL description, the coordinates of the ports are checked with those of the units and storage components. If the coordinate range of a port lies within the rectangular region corresponding to a unit or storage component, it is bound to that component. For example, Fig.2.12 shows two ports Alu1ReadPort1 and Alu1ReadPort2 bound to the component named ALU1_READ. The code generated in EXPRESSION for this example is given below:

```
(OpReadUnit ALU2_READ
  (CAPACITY 1)
  (INSTR_IN 1)
  (INSTR_OUT 1)
  (TIMING (all 1))
  (OPCODES ALU_Unit_ops FALU_Unit_ops)
  (LATCHES (OUT Alu2ReadExLatch))
  (LATCHES (IN DecAlu2ReadLatch))
  (PORTS Alu2ReadPort1 Alu2ReadPort2)
 )

  (OperationLatch Alu2ReadExLatch
  )

  (UnitPort Alu2ReadPort1("_READ_")
```

```
  (ARGUMENT _SOURCE_1_)
(CAPACITY 1)  )

(UnitPort Alu2ReadPort2("_READ_")
  (ARGUMENT _SOURCE_2_)
(CAPACITY 1)  )
```

2.7.3.4. Latches

Pipeline latches are associated with units and lie at the interface between two units. One way that a pipeline latch can be associated with a unit is to place it inside the rectangular region of that unit. This would then refer to the latch to which the unit will output its operation data. Another way is to have a connection component starting from the latch in one unit and ending in another unit. In this case, the latch is also bound to the second unit and this unit reads the data put into the latch by the first unit. In the GUI, small pink colored rectangular boxes represent these latches. They can be created from the **Components** menu. Clicking on these boxes brings up the attributes of the latch in the **Properties** window.

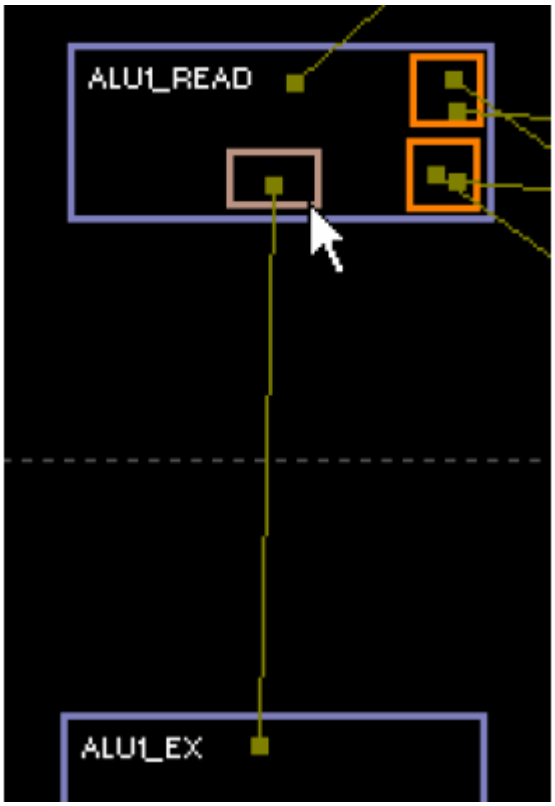


Figure 2.13: Alu1ReadExLatch

All latches in a design are stored in a global latch list. At the time of generation of the ADL description, the coordinates of the latches are checked with those of the units. If the coordinate range of a latch lies within the rectangular region corresponding to a unit, it is bound to that unit. If a connection component from that latch ends in another unit, the latch is bound to the second unit as well. For example, Fig 2.13 shows a latch Alu1ReadExLatch drawn inside the ALU1_READ unit and with a connection component from the latch ending in the ALU1_EX unit. The latch is thus associated with both the units. The code generated in EXPRESSION for this example is given below:

```
(OperationLatch Alu2ReadExLatch
 )

 (UnitPort Alu2ReadPort1("_READ_")
  (ARGUMENT _SOURCE_1_)
 (CAPACITY 1)  )

 (UnitPort Alu2ReadPort2("_READ_")
  (ARGUMENT _SOURCE_2_)
 (CAPACITY 1)  )
```

Here Operation Latch is the type of the latch.

2.7.3.5. Connections

A Connection is a component used to connect two ports, a latch in a unit and another unit or two storage components. In the GUI, a connection component is represented by a line segment (Fig.2.14). These can be created from the **Components** menu. Clicking on the line segment brings up the attributes of the connection in the **Properties** window.

Figure 2.14: A Connection component

All the connections in a system are stored in a global connection list. At the time of generation of the ADL description, the coordinates of the end points of the connection line segment are checked to see where they lie. If one end lies within a port region, then it represents a connection from a port to another port, and the other end must lie within a port too. If an end lies within a latch region, then it represents a latch connection between two units and the other end must lie inside another unit. Finally, if an end lies within a storage component but not inside a port of the component, then it represents a storage connection and the other end must also lie within a storage component.

2.7.3.6. Pipeline and Data-Transfer Paths Description

Recall that architectural pipelining information in EXPRESSION is specified using the notion of pipeline and data transfer paths. This subsection describes the structural net-list of the processor. The pipeline description is used to specify the functional units which make up the pipeline stages. This section is not explicitly specified in the GUI. The only construct needed for generating this section from the GUI is a pipeline stage component, which can be created from the Components menu. A pipeline stage component is represented graphically in the GUI as a horizontal line segment which groups the functional units in the layout into different stages. At the time of generation of the ADL description, a functional unit net-list is built on the fly. This is done by creating a graph with the functional units as nodes, connected with other nodes only if there is a shared latch between two nodes. This graph is further divided by the pipeline stage components in the following way: all the functional units that lie between two line segments corresponding to the pipeline stage components, become part of the pipeline stage whose name is given by the lower of the two pipeline stage components that enclose the unit. This inclusion of units within two pipeline stage components is verified by checking the coordinates of the unit rectangular box and ensuring that the coordinate range lies between the ranges of the two pipeline stage components. For the proposed architecture, the pipeline description generated is given below:

```
(PIPELINE_SECTION
 (PIPELINE FETCH DECODE READ_EXECUTE WB)
 (READ_EXECUTE (ALTERNATE read_execute0 read_execute1 read_execute2
read_execute3 read_execute4))
   (read_execute0( PIPELINE ALU1_READ ALU1_EX ))
   (read_execute1( PIPELINE ALU2_READ ALU2_EX ))
   (read_execute2( PIPELINE BR_READ BR_EX ))
   (read_execute3( PIPELINE LDST_READ LDST_EX ))
   (read_execute4( PIPELINE ALU3_READ ALU3_EX ))
   (read_execute4( PIPELINE ALU4_READ ALU4_EX ))


…
```

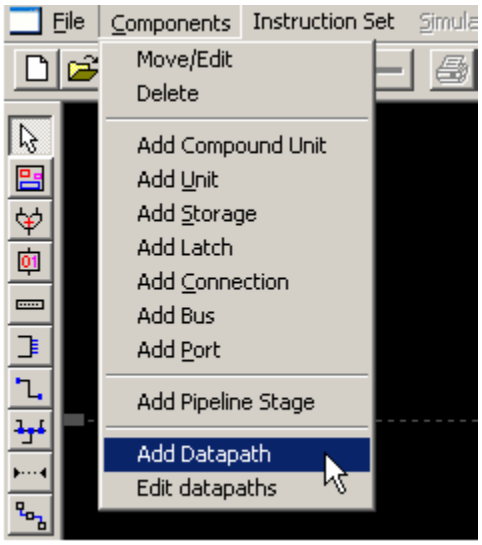


Figure 2.15 (a): Selecting 'Add Datapath' option

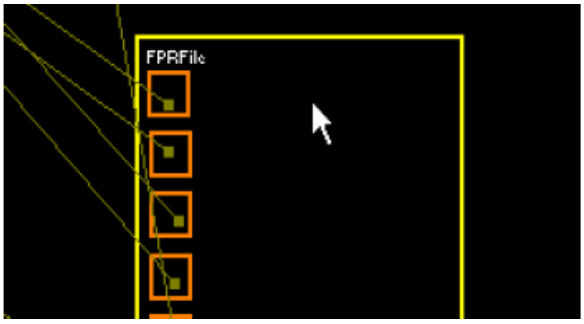


Figure 2.15 (b): Selecting FPRFile

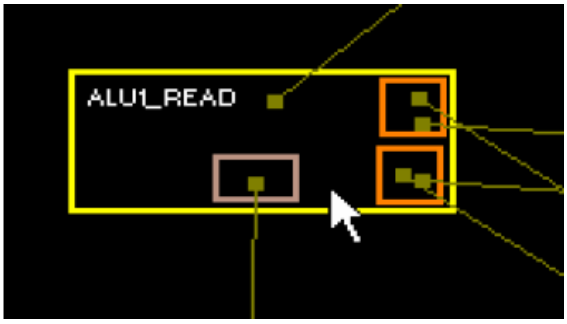


Figure 2.15 (c): Selecting ALU1_READ

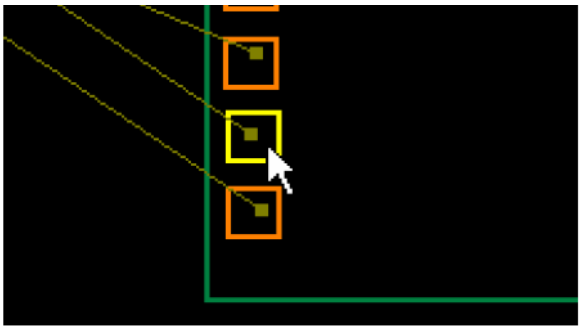


Figure 2.15 (d): Selecting port

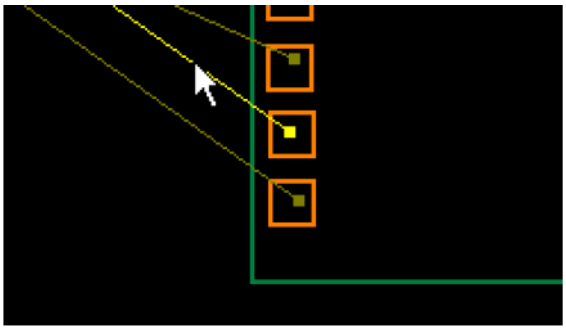
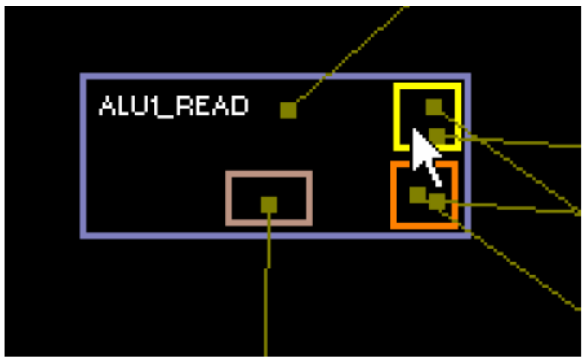


Figure 2.15 (e): Selecting connection element



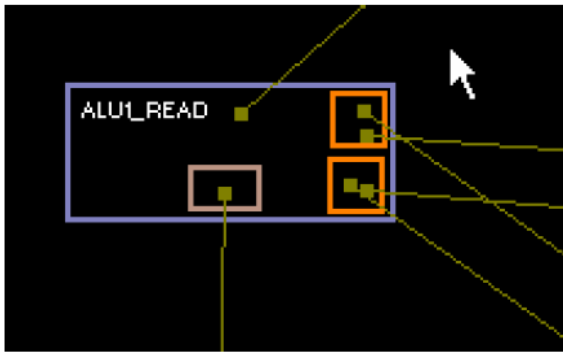


Figure 2.15 (f): Selecting port ALU1Read Figure 2.15 (g): Finish by right clicking

Data-transfer path descriptions specify the valid data transfers in the architecture. There are two kinds of data transfer paths – paths between functional units and storage components, and paths between two storage components. In the GUI, data paths are specified by traversing the path which can be done by selecting the **Add Data path** option from the **Components** menu and clicking on the units, storage components, ports and connections in the order specified by the ADL language. Just like macro recording, the order of clicking the components is recorded and a data path generated, which is stored with other data paths internally in the form of a list. It is important to note that specifying data paths is generally a very error prone and tedious activity. The GUI overcomes this limitation by allowing data paths to be specified conveniently and easily with just a couple of mouse clicks. Figure 2.15 shows the sequence of actions to be performed to
add a data path between the ALU1_READ component and the FPRFile register file. The code generated in EXPRESSION for this data path is given below:

```
(DTPATHS
(TYPE UNI
( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
)
)
)
```

The code generated for all the data paths between functional units and storage components in the
acesMIPS architecture is given below:

```
…
(DTPATHS

 (TYPE UNI
  ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
  ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
  ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
  ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
  ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
```

```
   ( FPRFile ALU1_READ FprReadPort6 FprReadPort6ALU1ReadPort1 Alu1ReadPort1)
(FPRFile LDST_READ FprReadPort5 FprReadPort5LdStReadPort3Cxn LdStReadPort3)
(GPRFileALU3_READGprReadPort10GprReadPort10Alu3ReadPort1CxnAlu3ReadPor1
) ( WB GPRFile WbWritePort WbWritePortGprWritePortCxn GprWritePort)
   ( WB GPRFile WbWritePort WbWritePortGprWritePortCxn GprWritePort)

  )

 )
)
```

…
Both pipeline and data-transfer path descriptions are essential for generating the retarget able simulator and generating reservation tables needed by the scheduler.

2.7.3.6. Memory Subsystem

This section is used to specify the attributes of the various storage components in the memory subsystem. A storage component comprises of the following attributes

WIDTH – Width of register file in bits
SIZE – Number of registers in register file
WORD SIZE – Word size of cache in bytes
LINE SIZE – Number of words in a cache line
ASSOCIATIVITY – Associativity level of cache
CACHE LINES – Number of lines in cache
ACCESS TIME – Time to access storage (in cycles)
ADDRESS RANGE – Range of addresses associated with storage
MNEMONIC – Prefix to be used for the registers in assembly formats

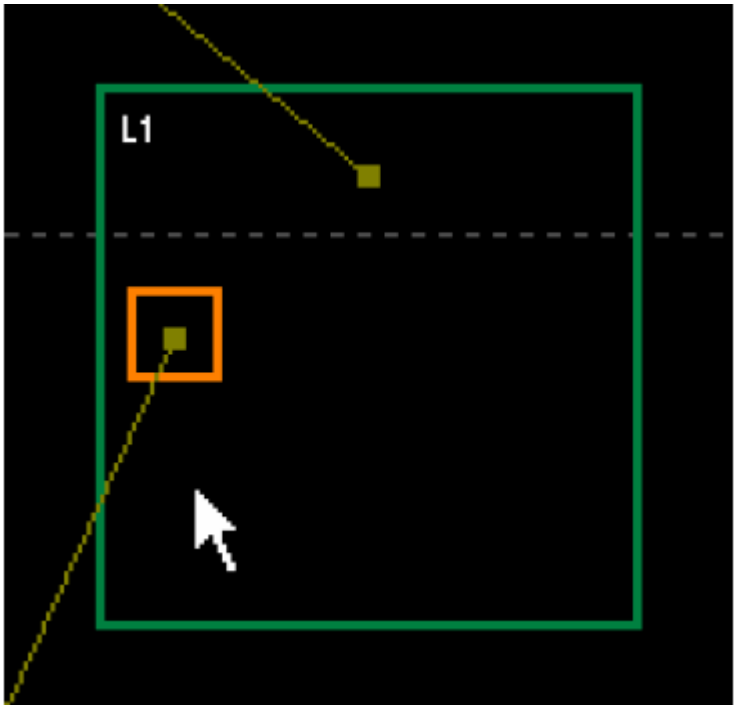


Figure 2.16: L1 cache with port

In the GUI, Storage components are represented graphically by green colored rectangular boxes (distinct from units) which can be created from the **Components** menu. Clicking on these boxes brings up the attributes of the unit in the **Properties** window (Fig. 2.16).Clicking on these boxes brings up the attributes of the unit in the Properties window. Storage components also have ports associated with them. Any port which lies within the rectangular region of a storage component binds to that component. For example, for the L1 data cache in the VLIW architecture (Fig 2.16), the architecture section contains the ports associated with the storage as shown below:

(Storage L1
(PORTS L1ReadWritePort)
(CAPACITY 1)
)

Properties window for L1 cache
Code is also generated for the storage section after specifying the storage attributes (Fig. 2.16) in the Properties window. This generated code is shown below.

(STORAGE_SECTION

(L1
 (TYPE DCACHE)
 (WORDSIZE 4)

```
  (LINESIZE 2)
  (ASSOCIATIVITY 4)
  (NUM_LINES 8)
  (ACCESS_TIMES 1)
  (ADDRESS_RANGE (0 9995904))

(MainMem
  (TYPE DRAM)
  (ACCESS_TIMES 50)
  (ADDRESS_RANGE (0 9995904))
 )
(GPRFile
  (TYPE VirtualRegFile)
  (WIDTH 128)
  (SIZE 32)
  (MNEMONIC "R")
 )

 (FPRFile
  (TYPE VirtualRegFile)
  (WIDTH 128)
  (SIZE 32)
  (MNEMONIC "f")
 )
```

Chapter # 3

3. VLIW ARCHITECTURE DETAIL

3.1. VLIW PROCESSOR

Almost all existing computing platforms act as some form of co processing elements i.e., they implement custom functions in hardware but operate under the control of host platforms.

3.1.2. IMPLEMENTATION:

This chapter includes the design what I have implemented. It is a VLIW processor but with certain limitations. This processor can perform number of different operations. I have designed this processor keeping certain things in mind, and then I have implemented that designed in EXPRESSION ADL using the VSAT GUI front end interface. It is a reconfigurable processor which enables it to adjust its functionalities accordingly. Reconfiguration actually makes it very powerful, increasing its computational power as many times as many is the number of functional units.

3.1.3. RECONFIGUARTION:

Reconfigurable computing is a new and emerging field that makes use of programmable devices to construct “custom computing machinery”. Reconfigurable computing can simply be thought of as an ability to repeatedly configure a machine to perform different and varying functions. The term reconfigurable is broad and can be applied to many scenarios. A reconfigurable custom computing machine makes use of some form of reconfigurable logic that can be changed and configured as demanded by an application.

3.2. CLASSIFICATION OF RECONFIGURABLE ARCHITECTURES:

Although it is believed that reconfigurable computing machines offer the same flexibility as of instruction set architectures type of machines (general purpose microprocessor based platforms) , and can yield performance that is comparable to the custom hardware , it is not a simple task to predict the tradeoffs between performance and the flexibility for reconfigurable computing machines. This is because the performance of the

reconfigurable computing machine can be compared against the performance of the software model of the application but cannot be compared to the custom hardware (ASIC type) implementations very easily. Thus it is not easy to find out the complete performance to flexibility relationships. Reconfigurable computing machines lie somewhere in the performance spectrum that spans between instruction set architectures and the custom hardware. This figure explains this relationship.

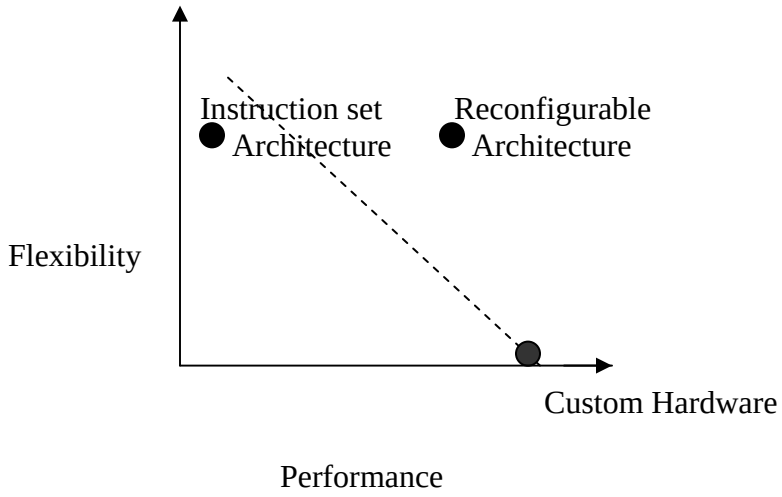


Figure 3.1: Classification of Reconfigurable Architecture

As can be seen from the graph **Re configurable architecture** has got equal **flexibility** as of **instruction set architecture** but still it's **performance** is almost as good as **custom hardware.**

3.2.1. Implementing the reconfigurable hardware:

There are two different ways the reconfiguration can be implemented in:

- STATICALLY
- DYNAMICALLY

3.2.1.1. STATICALLY RECONFIGURABLE HARDWARE:

Statically reconfigurable hardware is where the configuration of a custom application is loaded once and is not changed for the runtime life of the application. The only advantage of using reconfigurable hardware is that, that same platform can be used and re-used for

implementing many different applications. Sometimes, this is also called as compile time reconfiguration based hardware. It should be noted that the reconfiguration overhead for such hardware is a one time penalty that is incurred while loading the configuration data on the hardware.

3.3. A DYNAMICALLY RECONFIGURABLE HARDWARE:

Dynamically reconfigurable hardware is where the configuration of a custom application is loaded once, partially or wholly, and, is allowed to change during the runtime life of the application. This form of computing hardware offers almost infinite resources to speed up the application. This is true because, under partial reconfiguration, the hardware acts more like a paged memory system, and desirable functions can be loaded, swapped, or purged on demand. Also this type of hardware provides means for runtime optimization for speeding up the application. Managing the runtime dynamic reconfiguration is not a trivial task. It requires application profiling, data management, and clever techniques for reducing the overheads associated with the reconfigurable architectures.

3.3.1. RECONFIGURABLE DEVICES:

Reconfigurable devices can be configured after fabrication to solve any computational task. These are best exemplified today by FPGA. In these reconfigurable devices, tasks are implemented by spatially composing primitive operations and operators with the possibility of temporarily changing the hardware of the operators rather then temporarily composing of instructions sequence in Princeton style processors. The reconfigurable processor on FPGA can perform different operations on each bit, sore-configurable can be optimized to the data width of streaming data flows. The central theme of this work is to mix the advantages of Non-von-Neumann architectures with the advantages of re-configurable processing elements.

3.4. Explanation of the target Processor:

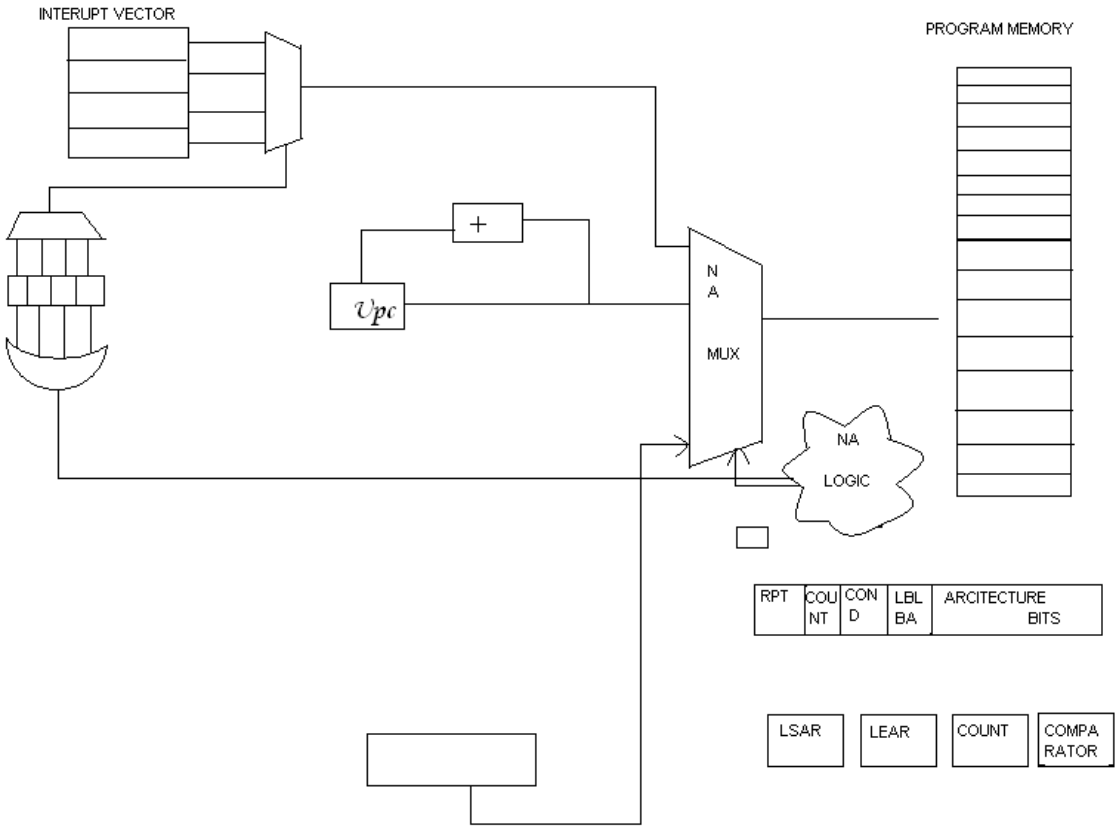


Figure 3.2: Processors Control Unit

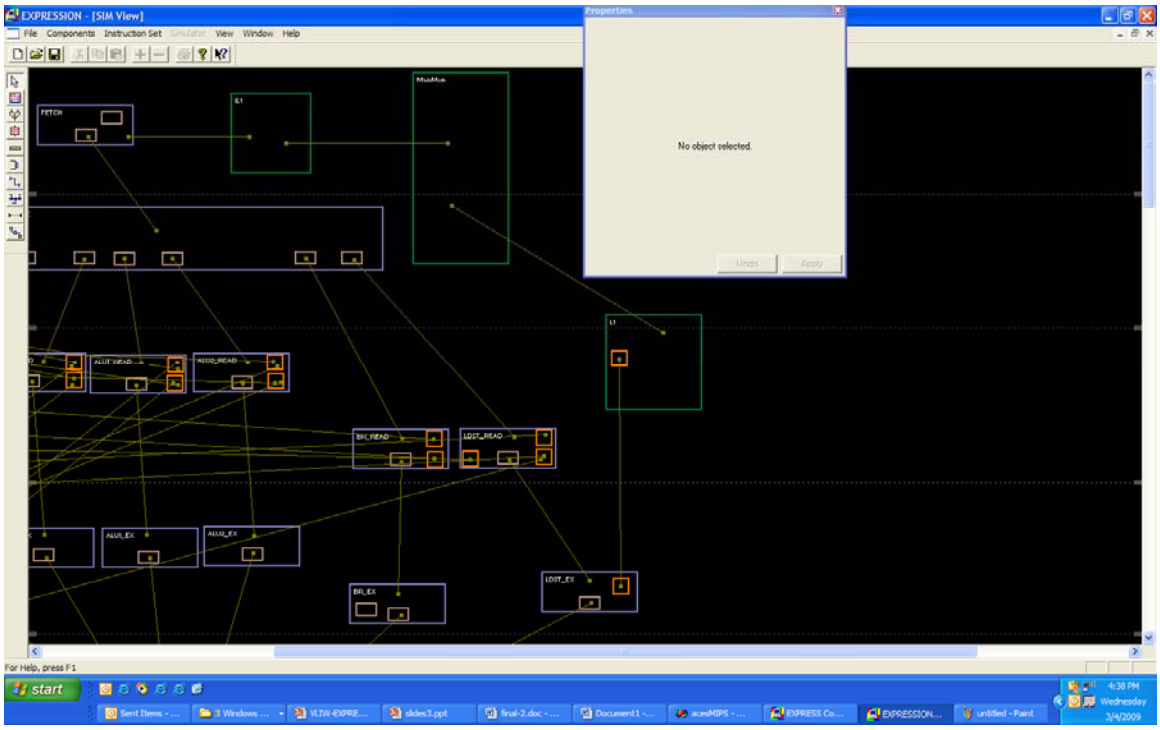


Figure 3.2.1: Processors Control Unit in EXPRESSION VSAT GUI

3.4.1. PROGRAM MEMORY:

It is actually a storage place where the program which has to be performed by the processor will be stored. The instructions will be read from this memory.
In our processor it is a 2-D memory. It has 16 rows, which means it can have, store 16 different instructions. And it has 76 columns which mean that every instruction is of 76 bits.
A user can write 16 different instructions in the program memory through files, then those instructions are coded by an **assembler**.

3.4.2. INSTRUCTION FORMAT:

There are two basic portions of the instructions which are:

- For the control unit.
- For the architecture, i.e. for the functional units.

Loop	Count	Condition	Label	c4	b4	a4	Op4	c3	b3	a3	Op3	c2	b2	a2	Op2

c1	b1	a1	Op1

3.4.3. THE INSTRUCTION DECODING PROCESS:

The instruction decoder identifies the portions of the instruction for the control unit& architecture.

For the control unit:

Loop	Count	Condition	Label

The separately identified control unit portion of the instruction is then further divided into sub-portions which are to categorize loop, sub routine, jump, conditional jump etc.
Now these portions go to the control unit, where they act as the activating signals for the operations like:

- Loop
- Subroutine
- Jump
- Conditional jump
- Internal interrupt

The control unit as obvious from its name controls the processor. It can also be called a controller of the processor, who looks after & monitors all the activities that the processor is doing.

3.4.4. Activities of the control unit:

3.4.4.1. LOOP MACHINE:

It requires the following things from the instruction for its execution:

- **LOOP ENABLE:**

It will be mentioned in the "**loop"** part of the instruction. It will be used to tell whether there is a loop or not. It is **1-bit** long.

- **LOOP END ADDRESS:**

It will be mention in the "**label"** part of the instruction. It is that address till where I want the instructions to be repeated. It is of **4-bit** long.

- **LOOP START ADDRESS:**

It will come from the "**program counter"**, and it is the starting address i.e from where I want to start the loop.

- **COUNT:**

It will be mentioned in the "**count**" part of the instruction. It is amount or number for how many times I want our instructions to be repeated. It is also **4-bit** long.

3.4.4.2. SUBROUTINE:

It is used when one wants to perform number or series of actions at one position. Or I can say when one requires some other routine while processing. That routine will be called a **subroutine.** It has two parts.

- **CALL**
- **RETURN**

CALL:

It is used to call a subroutine.

SYNTAX:

Call label;

RETURN:

After calling a subroutine the **program counter** moves to the **label** & starts performing no. of action till it will get a **return.** And when it gets the return it will return to the position next to, where the sub-routine was called. That will be the end of the subroutine. This is done by a maintaining "**return register"** that will contain the next address.

So the **subroutine requires** two parts of the instruction.

CONDITION:

This part of an instruction is **3-bit** long. It actually determines the **call**, or **return.**

LABEL:

This is the same **label** as that of **loop**, but now this label will act as an address of the branch address .i.e. where it has to move when there will be a call for the subroutine.

3.4.4.3. JUMP:

It is when program want to jump to a certain position or an instruction. Whenever there will be a jump the program counter will move to that instruction whose address would have been mentioned in the **label**. But one important thing to remember, that is that jump does not have a return. After jumping to a certain instruction the program counter does not return to the address where there **jump** was called.

It also requires two parts of the instruction.

CONDITION:

The condition actually tells that there is **jump** here.

LABEL:

This will have the branch address where I want the program counter to move.

3.4.4.4. CONDITIONAL JUMP:

It is same as that of the simple jump, but the only difference is that, that now the jump will depend on some condition.
It also requires those two parts of an instruction.

CONDITION:

This determines the conditional jump.

LABEL:

Where program want to move, or jump.

3.4.4.5. INTERUPTS:

It is an internal or external event that suspends the normal program flow within a computer and causes entry into a special interrupt program (also called **interrupt service routine**) interrupts are provided primarily as a way to improve processing efficiency.
When an interrupts comes the processor suspends execution of the current instruction, saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity. It sets the program counter to the starting address of an interrupt handler routine.

There are two types of the interrupts.

- **EXTERNAL INTERUPTS**
- **INTERNAL INTERUPTS**

EXTERNAL INTERUPTS:

In my processor I have four different types of external interrupts generated from the stimulus. Whenever those interrupts comes, the program counter jumps to the respective location, mentioned in the interrupt vector table corresponding to that specific type of the interrupt. These have nothing to do with the instruction. As these can be generated any time. & after executing that location where it was jumped the program counter will return back to the next address of where it encountered the interrupt.
This processor also maintains an interrupt vector table. It is a reserved memory location where a program counter jumps when an interrupt is detected.

INTERUPT SERVICE ROUTINE:

It is a program that is entered when an external or internal interrupt occurs. Interrupt service routines as usually high priority routines.

INTERNAL INTERUPT:

These are generated from inside the processor. It can be based on some condition; it can be generated when a timer reaches a certain value, etc.
In this VLIW processor I have catered an internal interrupt, which will be generated when the result from the subtractor will be less than 0,.i.e when it will be –ve. So when the interrupt will be generated its interrupt service routine will make that result +ve.

3.5. EXPLANATION OF PROCESSING UNIT

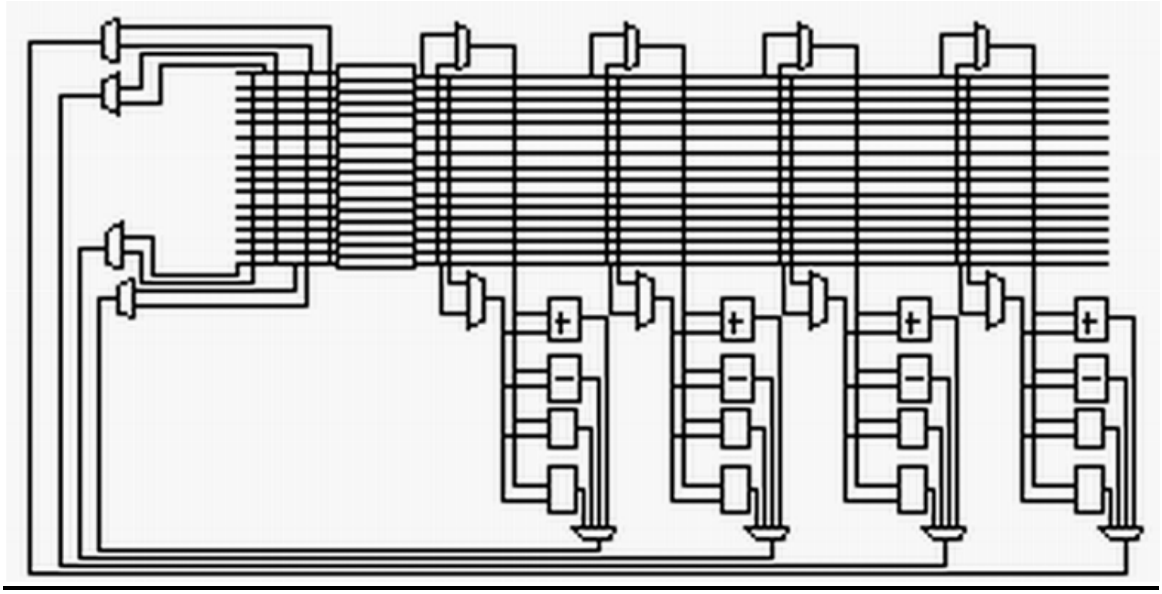


Figure 3.3: Processing Unit

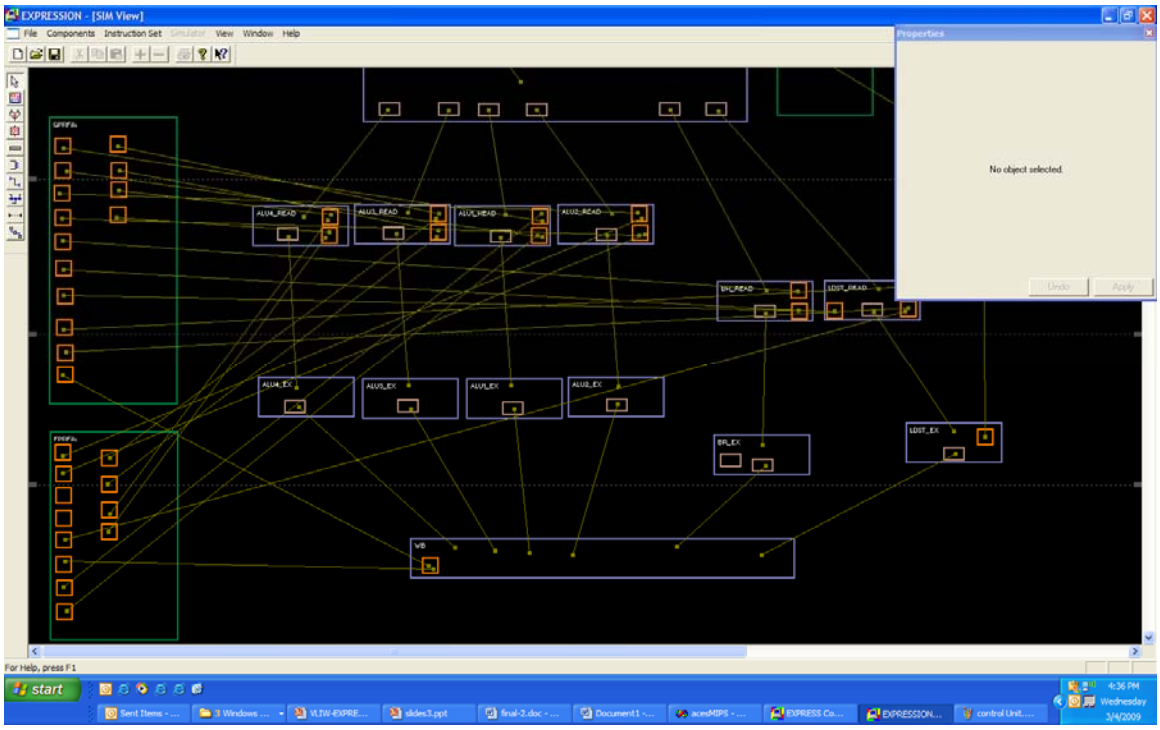


Figure 3.3.1: Processing Unit in EXPRESSION VSAT GUI

3.5.1. ARCHITECTURE BITS:

c4	b4	a4	Op4	c3	b3	a3	Op3	c2	b2	a2	Op2	c1	b1	a1	Op1

3.6. DETAIL OF PROCESSING UNIT

The separately identified portion for the architecture has four main sections, which are also recognized by the instruction decoder.
The four sections are for four functional units, each section has four parts. Which are:

- Opcode
- Operand1
- Operand 2
- Destination

3.6.1.OPCODE:

Operational code is known as **opcode**. It decides which operation is to be performed. In our processor I have the following four functions:

- Addition
- Subtraction
- Multiplication
- Barrel shifter

Each of these has a separate opcode, by which these are identified; the opcodes will be mentioned in the manual of our processor.Op1, Op2, Op3, Op4 are the opcodes for four different functions in each instruction.

3.6.2. OPERANDS:

These are the values on which the certain operations are to be performed. I am maintaining a **register file** which has values stored in it, So the operands will be coming from those registers. b4, a4, b3, a3, b2, a2, b1, a1 shows the addresses of the operands (registers carrying the values) for the four functional units.

3.6.3. DESTINATION:

It is the place, register where the results from the functional units are to be stored or written back. For example c4, c3, c2, c1 shows the four destination address where the results from the functional units will be written back.

3.6.4. FUNCTIONAL UNITS:

This part of the processor comprises of four functional units each having four operating units mentioned underneath.

3.6.4.1.ADDER:

It takes two operands which are identified by the instruction decoder, and gives the sum at next clock cycle. The adders in this processor is a FULL ADDER. I have one module of this full adder in every functional unit.

3.6.4.2. SUBTRACTOR:

It also takes two operands identified by the instruction decoder and inverts the one that is to be subtracted and adds with the other one as a result generating the result in next cycle.

3.6.4.3. MULTIPLIER:

I have two numbers in binary form that are to be multiplied .Result generated is of 16 bits but I only take most significant 8 bits just to standardize the results as when in case of reconfiguration these multiplier, adder, subtractor, shifter may be swapped , doesn't cause any problem.

3.6.4.4. SHIFTER:

Shifter shifts one operand by the amount other operand is. Results are generated in the next cycle.
Our every functional unit has all the above mentioned capabilities and I select only one result from each functional unit at the end using Reconfiguration mux. **This allows my processor to be strongly flexible making computations more easy and appropriate as all the operations are carried out on all the operands selected and providing the required result in the end.**

CHAPTER #4

4. Results

Once the architecture is designed in EXPRESSION front end GUI it converts the schematic description and instruction set description into EXPRESSION ADL format.
EXPRESS is a retargetable compiler centered on a generic machine. **EXPRESS** reads the front-end files, builds an Intermediate Representation (IR) amenable to different optimizations and targets the architecture described in an EXPRESSION ADL (Architecture Description Language) description. **SIMPRESS** reads the special assembly file, simulates the running of assembly on an architecture template generated from the ADL description and finally generates area, power, and performance numbers including cycle count and memory usage statistics. The purpose of the simulator is to assess the efficacy of the code generated by the EXPRESS compiler for the given architecture. The EXPRESSION ADL description of VLIW Architecture is available in <work>\acesMIPSDll\bin\ acesMIPS.xmd. The schematic description of **VLIW Architecture** is stored in acesMIPS.gmd and the instruction set description in acesMIPS.isd.

4.1. EXPRESSION COMPILATION

I compiled EXPRESSION CONSOLE and it takes the EXPRESSION description in <run>\acesMIPS.xmd and generates different intermediate files required to retarget the compiler and the simulator. It also generates <run>\mem.config containing memory configuration. The detail of memory configuration is mentioned below

4.1.1. Memory Configuration as Compiled by EXPRESSION:

```
BEGIN_MEM_MODULES
0 DCACHE 8:2:4:4:1
1 ICACHE 8:2:4:4:1
2 DCACHE 64:2:8:4:5
3 SRAM 9999999:1
4 DRAM 9995904:50
END_MEM_MODULES

BEGIN_CONNECTIVITY
1 2
0 2
2 4
```

END_CONNECTIVITY

BEGIN_MEMORY_MAP
0 0 9995904
3 9995905 9999999
END_MEMORY_MAP

4.1.2. Output of EXPRESS Console compilation

regFileName = L1
regFileMnemonic = (null)
regs :

varName = int_normal
<targetClassType = NORMAL targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28

varName = int_pc
<targetClassType = PC targetDataType = INT>
regFileName = PC
regFileMnemonic = (null)
regs :

varName = int_retaddr
<targetClassType = RET_ADDR targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 31

varName = int_retval
<targetClassType = RET_VAL targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 4

varName = int_sp
<targetClassType = SP targetDataType = INT>
regFileName = GPRFile

```
regFileMnemonic = R
regs :  29


varName = int_zero
<targetClassType = ZERO targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs :  0


######################################################################

######################################################################
varName = any_call_param
<targetClassType = CALL_PARM    targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs :  4 5 6 7 8 9 10 11 12


varName = any_cc
<targetClassType = CC   targetDataType = INT>
regFileName = CC
regFileMnemonic = (null)
regs :


varName = any_fp
<targetClassType = FP   targetDataType = INT>
regFileName = FP
regFileMnemonic = (null)
regs :


varName = any_hilo
<targetClassType = HILO targetDataType = INT>
regFileName = HILO
regFileMnemonic = (null)
regs :


varName = any_pc
<targetClassType = PC   targetDataType = INT>
regFileName = PC
regFileMnemonic = (null)
```

regs :

varName = any_retaddr
<targetClassType = RET_ADDR targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 31

varName = any_sp
<targetClassType = SP targetDataType = INT>
regFileName = SP
regFileMnemonic = (null)
regs :

varName = double1_normal
<targetClassType = DOUBLE1 targetDataType = DOUBLE>
regFileName = FPRFile
regFileMnemonic = f
regs : 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

varName = double1_retval
<targetClassType = RET_VAL targetDataType = DOUBLE>
regFileName = FPRFile
regFileMnemonic = f
regs : 0

varName = double2_normal
<targetClassType = DOUBLE2 targetDataType = DOUBLE>
regFileName = FPRFile
regFileMnemonic = f
regs : 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31

varName = double2_retval
<targetClassType = RET_VAL targetDataType = DOUBLE>
regFileName = FPRFile
regFileMnemonic = f
regs : 1

varName = double_all

```
regFileName = FPRFile
regFileMnemonic = f
regs :  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31
regFileName = IMM
regFileMnemonic = (null)
regs :


varName = double_any
<targetClassType = ANY  targetDataType = DOUBLE>
regFileName = FPRFile
regFileMnemonic = f
regs :  0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30


varName = double_immediate
<targetClassType = IMM  targetDataType = DOUBLE>
regFileName = IMM
regFileMnemonic = (null)
regs :


varName = float_all
regFileName = FPRFile
regFileMnemonic = f
regs :  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31
regFileName = IMM
regFileMnemonic = (null)
regs :


varName = float_any
<targetClassType = ANY  targetDataType = FLOAT>
regFileName = FPRFile
regFileMnemonic = f
regs :  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31


varName = float_immediate
<targetClassType = IMM  targetDataType = FLOAT>
regFileName = IMM
regFileMnemonic = (null)
regs :
```

varName = float_normal
<targetClassType = NORMAL targetDataType = FLOAT>
regFileName = FPRFile
regFileMnemonic = f
regs : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31

varName = float_retval
<targetClassType = RET_VAL targetDataType = FLOAT>
regFileName = FPRFile
regFileMnemonic = f
regs : 0

varName = int_all
regFileName = GPRFile
regFileMnemonic = R
regs : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31
regFileName = IMM
regFileMnemonic = (null)
regs :

varName = int_any
<targetClassType = ANY targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28

varName = int_call_param
<targetClassType = CALL_PARM targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 4 5 6 7 8 9 10 11 12

varName = int_cc
<targetClassType = CC targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R

regs : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

varName = int_fp
<targetClassType = FP targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 30

varName = int_hilo
<targetClassType = HILO targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

varName = int_immediate
<targetClassType = IMM targetDataType = INT>
regFileName = IMM
regFileMnemonic = (null)
regs :

varName = int_mem
<targetClassType = MEM targetDataType = INT>
regFileName = L1
regFileMnemonic = (null)
regs :

varName = int_normal
<targetClassType = NORMAL targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

varName = int_pc
<targetClassType = PC targetDataType = INT>
regFileName = PC
regFileMnemonic = (null)
regs :

varName = int_retaddr
<targetClassType = RET_ADDR targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 31

varName = int_retval
<targetClassType = RET_VAL targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 4

varName = int_sp
<targetClassType = SP targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 29

varName = int_zero
<targetClassType = ZERO targetDataType = INT>
regFileName = GPRFile
regFileMnemonic = R
regs : 0

##

4.2. SIMPRESS SIMULATION

After successful generation of intermediate files from EXPRESSION compilation I set ACESMIPS CONSOLE as the active project. It contains both **EXPRESS** compiler and **SIMPRESS** simulator. The **acesMIPS console** application generates the number of cycles, memory usage and other statistics in <run>/<filename>.pwrStats.

4.2.1. Power stats detail as compiled by EXPRESS AND SIMPRESS

```
Total Cycles: 675

ASSOCIATIVE DCACHE: Cache Accesses: 80 (0.45)
       read hits: 7, read misses: 7 (0.50)
       write hits: 29, write misses: 37 (0.44)
       Energy Dissipation: 0.096 uJ

ASSOCIATIVE DCACHE: Cache Accesses: 147 (0.78)
       read hits: 115, read misses: 32 (0.78)
       write hits: 0, write misses: 0 (0.00)
       Energy Dissipation: 0.176 uJ

ASSOCIATIVE DCACHE: Cache Accesses: 125 (0.49)
       read hits: 12, read misses: 64 (0.16)
       write hits: 49, write misses: 0 (1.00)
       Energy Dissipation: 1.022 uJ

SRAM:       loads: 0, stores: 0
      Energy Dissipation: 0.000 uJ

DRAM:       loads: 64,  stores: 3
       Energy Dissipation:  0.005 uJ
```

4.2.2 Output of ACESMIPS Console simulation

Starting EXPRESS...

-- EXPRESS: Started ----------

Procs file name: LL1.procs

#++++++++++++++++++++++++++++++++#

Reading Compiler Opcodes

Done Reading Compiler Opcodes

#++++++++++++++++++++++++++++++++#

```
#+++++++++++++++++++++++++++++++++#

Intializing Symbol Table

Done Initializing Symbol Table

#+++++++++++++++++++++++++++++++++#

#+++++++++++++++++++++++++++++++++#

Reading Compiler Opcodes

Done Reading Compiler Opcodes

#+++++++++++++++++++++++++++++++++#
-------------------------------------------------------

-- EXPRESS: Reading LL1.procs ------------

-------------------------------------------------------
Name of procedure: _main

#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
Routine: _main

#+++++++++++++++++++++++++++++++++#

Building Control Flow Graph (CFG).

Performing DFS ordering on CFG.

Done Building CFG.

#+++++++++++++++++++++++++++++++++#

-------------------------------------------------------
 Code Size Before All Transformations:
       Num. Instructions = 51
       Num. Operations = 51
-------------------------------------------------------

Warning: RoutParmProperty copy does not work if it is not an empty property
Parsing LL1.defs...

Global Memory Allocation:
```

Local Memory Allocation:

TIME : 0.0 secs

#+++++++++++++++++++++++++++++++++#

Building Static Single Assignment (SSA) Form.

Done Building SSA.

#+++++++++++++++++++++++++++++++++#

#*#
Routine: _main

 Code Size After SSA Before All Other Transformations:
        Num. Instructions = 59
        Num. Operations = 59

#*#
Routine: _main

Performing Def-Use (DU) Analysis.

Performing Use-Def (UD) Analysis.

#+++++++++++++++++++++++++++++++++#

Performing Live-Dead (LD) Analysis.

Done Performing LD Analysis.

#+++++++++++++++++++++++++++++++++#

#+++++++++++++++++++++++++++++++++#

Building Hierarchical Task Graph (HTG).

Done Building HTG.

#+++++++++++++++++++++++++++++++++#

```
Setting instrs ID...
Done setting instrs ID.

#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
Routine: _main
Warning: A pass that is executed only once per routine was called once too many
for routine: _main
Warning: A pass that is executed only once per routine was called once too many
for routine: _main
Warning: A pass that is executed only once per routine was called once too many
for routine: _main

#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
Routine: _main

Setting instrs ID...
Done setting instrs ID.

#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
Routine: _main

Setting instrs ID...
Done setting instrs ID.
      Finished Parsing the ISel input file...

***TIME*** : 0.0 secs

#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
Routine: _main

#+++++++++++++++++++++++++++++++#

Performing Register Allocation.
Recomputing properties...
Coloring interference graph...
IG Statistics:
   133 nodes: 96 registers, 37 multichains,
   edges: 2892
--------------------------------------
-- Num. Data Ops Eliminated: 0 ------
-- Num. Control Ops Eliminated: 0 ---
-- Num. Instructions Eliminated: 0 --
--------------------------------------

Done Performing Register Allocation.
```

```
#+++++++++++++++++++++++++++++++#
Printing Routine:
End Printing Routine
**********************************************
 Start simulation : _DUMP_IR_AFTER_REGALLOC.txt
#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#
DEBUG_PRINT: (I)16784
#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#
The number of cycles is: 675

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        Power Stats:

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

ASSOCIATIVE DCACHE: Cache Accesses: 80 (0.45)
        read hits: 7, read misses: 7 (0.5)
        write hits: 29, write misses: 37 (0.439394)
        Energy Dissipation: 0.0960413 uJ

ASSOCIATIVE DCACHE: Cache Accesses: 147 (0.782313)
        read hits: 115, read misses: 32 (0.782313)
        write hits: 0, write misses: 0 (0)
        Energy Dissipation: 0.176476 uJ

ASSOCIATIVE DCACHE: Cache Accesses: 125 (0.488)
        read hits: 12, read misses: 64 (0.157895)
        write hits: 49, write misses: 0 (1)
        Energy Dissipation: 1.02249 uJ

SRAM:   loads: 0, stores: 0
       Energy Dissipation: 0 uJ

DRAM:   loads: 64,      stores: 3
       Energy Dissipation:  0.00458459 uJ

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

***TIME*** : 0.5 secs
 End simulation : _DUMP_IR_AFTER_REGALLOC.txt
**********************************************

#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#
Number of Cycles (After RA): 675
#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#
```

```
#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#
Number of Cycles (After RA, LockStep): 0
#-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-#

-------------------------------------------------------
 Code Size After All Transformations:
       Num. Instructions = 53
       Num. Operations = 51
-------------------------------------------------------


-- EXPRESS: Finished  ---------------------------------------------------


Leaving EXPRESS...
```

4.2.3. Other Relevant Stat files

There are many other relevant stats generated in the files mentioned below

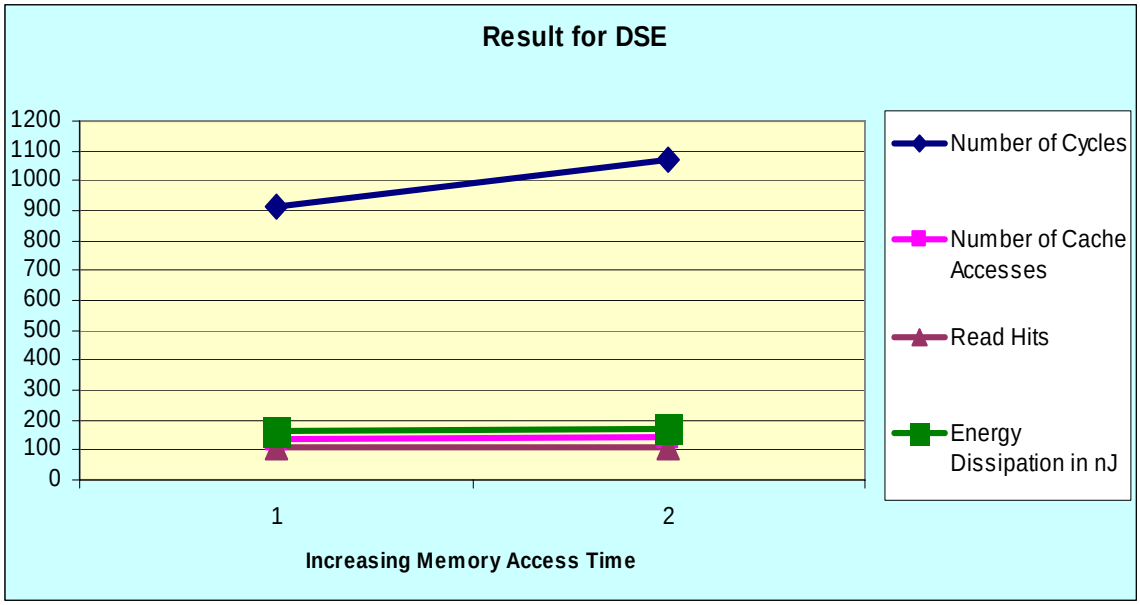
- ➔ DumpIRFormat.txt
- ➔ IselMapping.txt
- ➔ VLIW.pwrStats
- ➔ DUMP_IR_AFTER_REGALLOC.txt
- ➔ MIPS_OpCodeTiming.txt
- ➔ operandsMappingFile.txt
- ➔ SpillReloadTarget.txt
- ➔ targetRegClasses.txt
- ➔ targetRegClassToRegsMapping.txt
- ➔ targetRegFileList.txt

4.3. Design Space Exploration

Detailed design space exploration was carried to optimize and analyze the VLIW architecture designed. Several directions were taken to optimize the processor functionality details of the same are mentioned below.

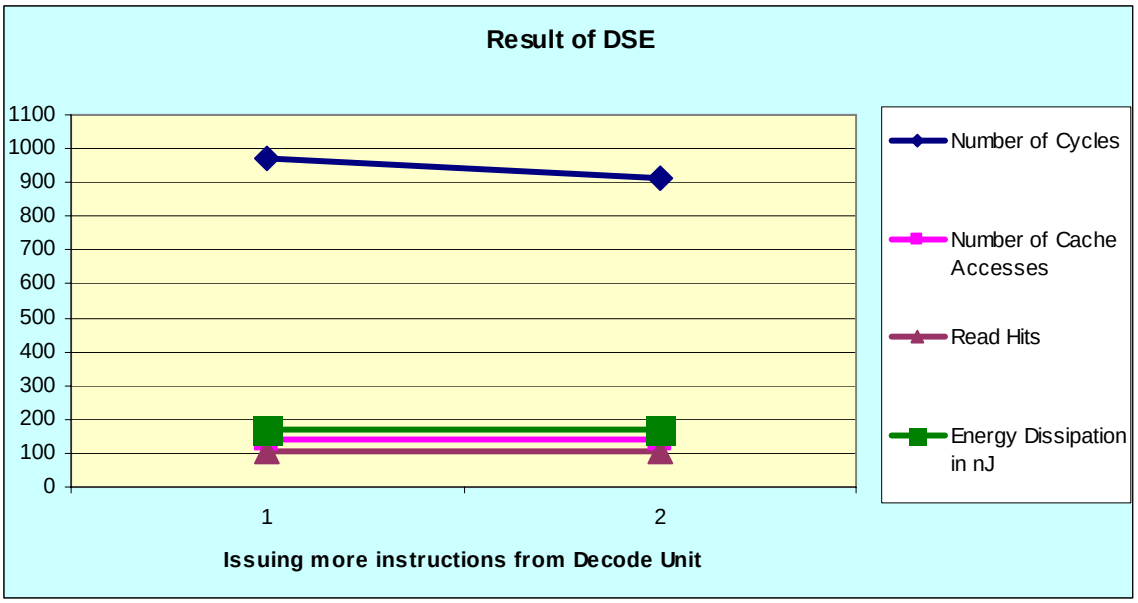
4.3.1 Changing Instruction Memory Access time

The access time of the instruction memory was reduced from 40 cycles to 50 cycles. Ideally the number of cycles required to complete the Livermore loops simulation on VLIW processor was suppose to take more number of cycles. Same was depicted originally in the simulation results. Graph has been plotted and shown below for the same to display results in the same perspective.



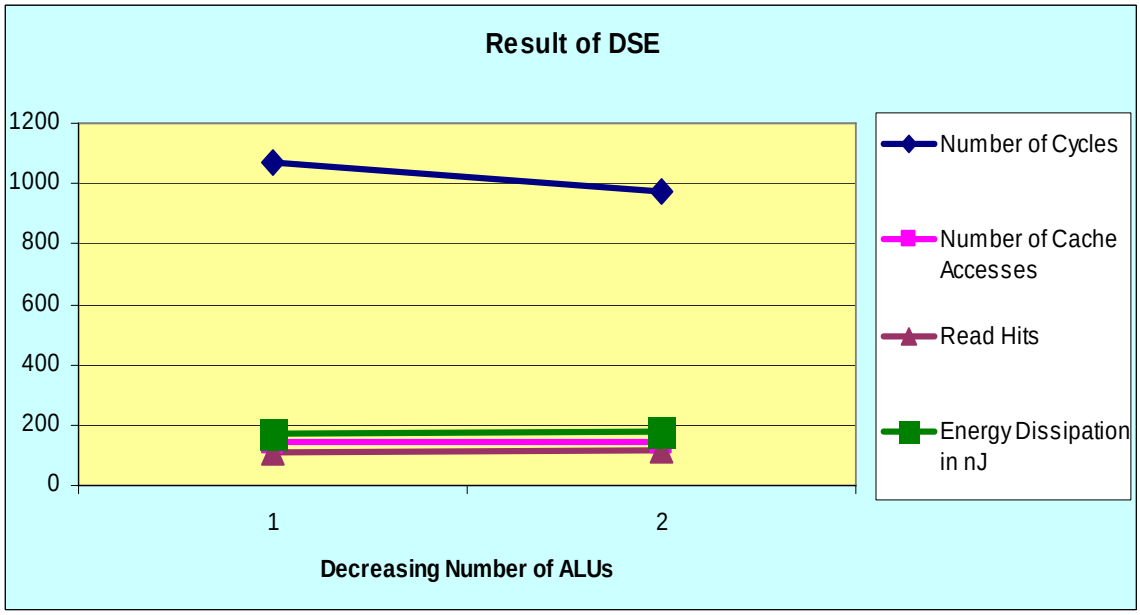
4.3.2 Issuing More Instructions in Parallel from Decode Unit

Initially 4 instructions were issued from processor decode unit in parallel. To explore further I increased the number of parallel instructions to 6.
Ideally the number of cycles required to complete the Livermore loops simulation on VLIW processor was suppose to take lesser number of cycles. Same was depicted originally in the simulation results. Graph has been plotted and shown below for the same to display results in the same perspective.



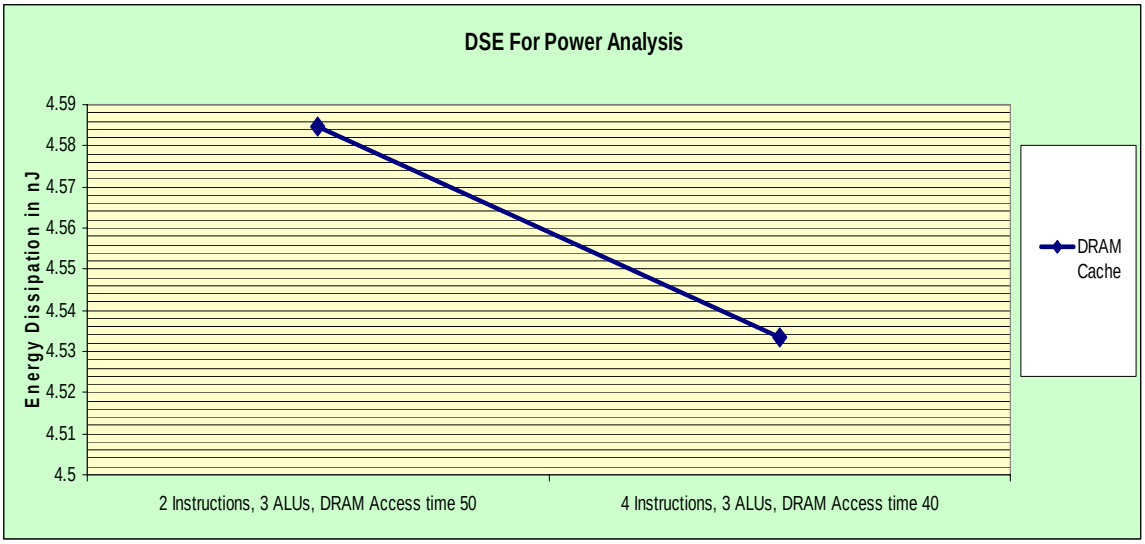
4.3.3 Decreasing number of ALUs from the Processor

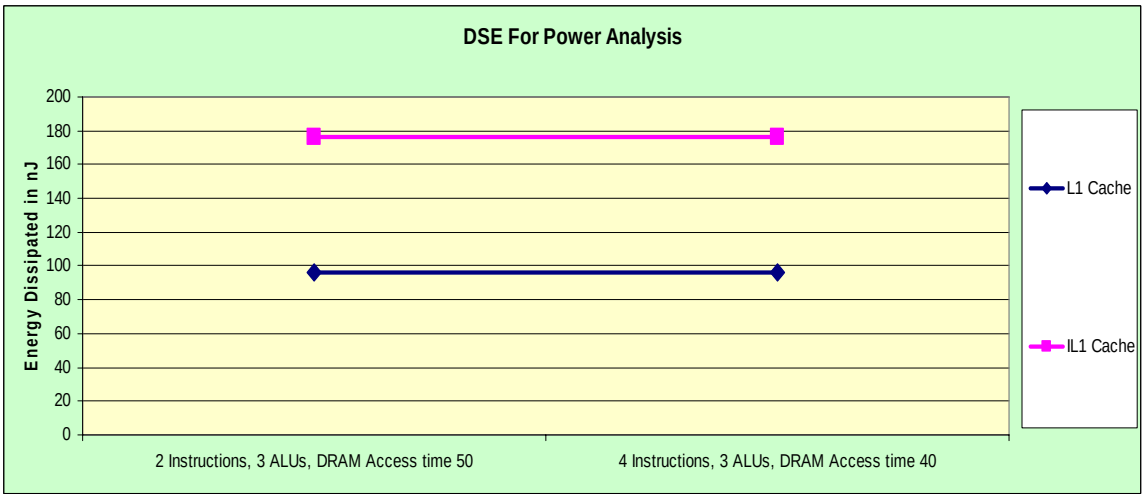
Initially 4 ALUs were part of the processor. To explore further I decreased the number of ALUs to 3.
Ideally the number of cycles required to complete the Livermore loops simulation on VLIW processor was suppose to take more number of cycles since the computing capability of the processor was reduced. However on the contrary interestingly the number of cycles to complete the simulation was reduced. Reason for the same could be that 3 ALU Architecture might be more suited for running the application based upon liver more loops. Graph has been plotted and shown below for the same to display results in the same perspective.



4.3.4 Power Analysis

Initially 2 instructions were issued from processor decode unit in parallel and the memory access time was set to 50 cycles. To explore and optimize the hardware the number of instructions issued in parallel was increased to 4 and the memory access time was reduced to 40 cycles.
Ideally the number of cycles required to complete the Livermore loops simulation on VLIW processor was suppose to take lesser number of cycles and also reduced energy dissipation from Instruction memory. However no impact on the energy dissipated from the cache memories. Same was depicted originally in the simulation results. Graphs for both types of memories have been plotted and shown below to display results in the same perspective.





CHAPTER #5

5. CONCLUSION:

I intended to design a processor in an ADL. EXPRESSION was the ADL which I used to design my processor and complete understanding was developed initially for using the ADL's VSAT GUI.
The design of the processor is elaborate and complete memory hierarchy is also explained. The VLIW processor designed is quite powerful due to availability of four ALUs. Same architecture can be made reconfigurable by using specific functions of ALU units at a time. In turn the processor has wide variety of applications and power requirements accordingly. Designing the processor in ADL helped me to explore more options for optimizing the hardware design. ADL has many more application likes simulating the results for the designer as well. Lots of relevant stats information is generated after compiling the architecture in EXPRESSION which helps in optimizing the architecture.
Complete installation procedure and ADL code is also shared with the thesis which can be used for further learning and knowledge sharing.

CHAPTER # 6

6. FUTURE ENHANCMENTS AND SCOPE:

6.1. FUTURE PERSPECTIVE

I have implemented the VLIW architecture in the ADL.

- Next logical step should be to make the same processor more efficient by exploring different design options.
- EXPRESSION simulates the bench mark of Livermore Loops and also allows different software applications to be simulated on the designed Architecture.
- Using EXPRESSION we can make embedded architecture design most suited and sorted out simulating the target application on the same.
- More improvements can be made in the same architecture by optimizing it's power requirements and making a design which is more power efficient.
- Introducing reconfiguration can also server to improve the power requirements of the same hardware.

RECONFIGURATION IN THE STAND ALONE MODE:

6.2. PROCEDURE:

Getting the specific bit stream can be done using a CPLD (complex programmable logic device) which picks it up from the EPROM also designed in ADL and make it available whenever reconfiguration is required. Reconfiguration logic can be simulated by creating the CPLD aware application and running in on the intended architecture. Keeping in view that the application is smart enough to load new configuration from EPROM and using the architecture accordingly.
Reconfiguration logic being implemented can be optimized with minimum hardware and maximum utility of the resources creating a cutting edge processor both in terms of Speed. Space, Power Consumption and Heat Dissipation.

6.3. COMMENT:

Designing the CPLD is in fact an additional hardware which can interface with existing VLIW processor or it can have use with other hardware design where reconfiguration can be done to improve the efficiency of the design. Interfacing the CPLD with the VLIW architecture proposed or can also be made a generic CPLD which can be used with other reconfigurable architectures as well.
Since using lesser power and optimizing the use of existing resources is the aim of the designer so Architecture Description language provide designers with vast opportunity to find the best possible solution.

Bibliography

A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt and A. Nicolau. “V-SAT: A visual

1. specification and analysis tool for system-on-chip exploration”. *In Proc. EUROMICRO, 1999.*
2. P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt and A. Nicolau. “EXPRESSION: An ADL
3. for System Level Design Exploration”, *Technical Report.*
4. P. Grun, A. Halambi, N. Dutt and A. Nicolau. RTGEN: An algorithm for automatic generation of
5. reservation tables from architectural descriptions. *In ISSS, San Jose, CA, 1999*

A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. *In*

6. *ICPP, St. Charles, IL, 1993*
7. Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau.
8. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator
9. Retargetability ", *DATE 99*
10. Ashok Halambi, Nikil Dutt and Alex Nicolau "Customizing Software Toolkits for Embedded
11. Systems-On-Chip", DIPES 2000
12. Prabhat Mishra, Peter Grun, Nikil Dutt, and Alex Nicolau "Memory Subsystem Description in
13. EXPRESSION ", UCI-ICS Technical Report #00-31
14. H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau "Modeling and Verification of
15. Processor Pipelines in SOC Design Exploration" In Proc. of 4rd International High Level Design
16. Validation and Test Workshop (HLDVT'99), pp. 10--16, Nov. 1999
17. Ashok Halambi, Aviral Shrivastava, Nikil Dutt and Alex Nicolau. "A Customizable Compiler
18. Framework for Embedded Systems", SCOPES 2001
19. A Khare “SIMPRESS: A Simulator Generation Environment for System-on-Chip Exploration”
20. Masters Thesis, UCI-ICS 1999
21. V. Zivojnovic et al. “LISA - machine description language and generic machine model for
22. HW/SW co-design. In VLSI Signal Processing”, 1996.
23. M. Freericks. “The nML machine description formalism.” TR SM-IMP/DIST/08, TU Berlin,
24. 1993.

25. George Hadjiyiannis , Silvina Hanono , Srinivas Devadas, “ISDL: an instruction set
26. description language for retargetability”, Proceedings of the 34th annual conference on Design
27. automation conference, p.299-302, June 1997
28. P. Biswas, S. Pasricha, P. Mishra, A. Shrivastava, N. Dutt, A. Nicolau, “EXPRESSION User
29. Manual version 1.0”, Feb 2003
30. DeHon, Andre, *Re-configurable Architectures for General-Purpose*
31. *Computing*.
32. A.I. Technical Report No. 1586, M.I.T. Artifical Intelligence Lab., Oct. 1996
33. VLIW at IBM Research.
34. N.J. Drew and M.M. Dillinger, “Evolution to reconfigurable user equipment,” *IEEE Communication Magazine*, vol. 39, no. 2, Feb. 2001.
35. Augmenting a Microprocessor with Re configurable Hardware by John Reid Hauser B.S. (North Carolina State University) 1987 M.S. (University of California, Berkeley) 1994 A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science in the GRADUATE DIVISION of the UNIVERSITY OF CALIFORNIA, BERKELEY.
36. Palnitkar, S., Verilog HDL, Sun Microsystems, California, 1996.
37. Gregory, K., Special Edition using Visual C++ 6, Prentice Hall, India, 1998.
38. mbinu@cs.utah.edu
39. express@cecs.uci.edu
40. http://www.cecs.uci.edu/~express

LIST OF ACRONYMS

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuits
CISC	Complex Instruction Set Computers
FPGA	Field Programmable Gate Array
FU	Functional Unit
FPL	Field Programmable Logic
ILP	Instruction level Parallelism
OPCODE	Operational Code
RISC	Reduced Instruction Set computers
RCS	Reconfigurable Computing System
RFU	Reconfigurable Functional Unit
RTR	Runtime Reconfiguration
RC	Reconfigurable Computers
VLIW	Very Large Instruction Word
ADL	Architecture Description Language
EXPRESSION	Name of the ADL used