A Framework for GUI-driven Design Space Exploration of a MIPS4K-like processor [†]

Sudeep Pasricha, Partha Biswas, Prabhat Mishra, Aviral Shrivastava, Atri Mandal
Nikil Dutt and Alex Nicolau
{sudeep, partha, pmishra, aviral, mandala, dutt, nicolau} @ics.uci.edu

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
http://www.ics.uci.edu/~aces

Technical Report
April 2003

Abstract

EXPRESSION is an ADL (Architecture Description Language) that 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 an 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 we 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. We give the example of a MIPS 4K like processor called acesMIPS which was completely captured using our automated framework and present some sample exploration studies.

[†] This work was supported in part by Motorola Inc, NSF (CCR-020203813, CCR-0205712, ACI-0204028) and DARPA (F33615-00-C-1632).

1. Introduction

EXPRESSION [2] [5] is an ADL (Architecture Description Language) that can capture a processor-memory architecture description and generate a compiler [6] and simulator [10] automatically from this description. It is based on a generic machine model described in [14]. 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 architecture-specific compiler and simulator flow. However, so far we did not have an automated path incorporating a push button GUI-driven processor that comprehensively handled specification capture and subsequent generation of intermediate files. An earlier version of a graphical front-end (V-SAT) [1] was limited to capturing a DLX like architecture and the framework could only generate a subset of the code required for an automated flow. With Systems-on-Chip (SOC) designs becoming ever more complex, the importance of an automated framework that allows quick and reliable architecture exploration and performance evaluation becomes even more critical. Designers need to quickly prototype and evaluate their architecture configurations.

The EXPRESSION toolkit allows the SOC designer to analyze architectural trade offs by simulating the design on the automatically generated simulator. While a number of processor description languages like LISA [11], nML [12] and ISDL [13] support this goal, none of these can boast of generating a retargetable compiler which can exploit the information in the description to generate optimized code for the architecture. Furthermore, to the best of our knowledge none of these frameworks provide a graphical front-end tool which designers can intuitively use to specify the system architecture in.

This report is organized as follows. Section 2 discusses a visual specification and analysis front-end which we have incorporated in our framework for rapid design space exploration. We go over in detail how the ADL specification and description generation takes place in such a framework. Section 3 is concerned with the ADL description to back-end flow and gives a general overview of the compiler and simulator back-ends. Section 4 gives an overview of some of the exploration facets that can be undertaken in such a framework. Section 5 discusses the results obtained on running common benchmarks on our framework. Section 6 concludes the report.

2. 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 we have incorporated a graphical front end tool (V-SAT) [1] 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 toolkit generator phase. We have modified the original V-SAT front end to make it more robust and flexible for capturing processor specifications and have added support for several new features in the EXPRESSION ADL like memory subsystem specification. Of course we can always model the architecture with the ADL, but the point is that simple but non-trivial, realistic processors can be fully described in this system without recourse to ADL.

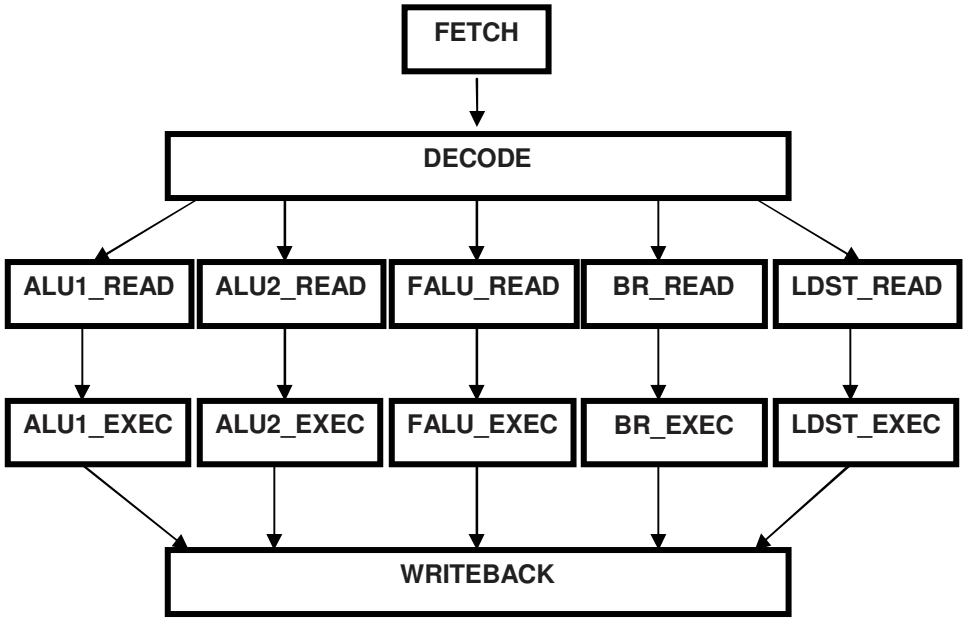

Figure 1: Simplified acesMIPS architecture

The primary motivation behind our 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 testbench suites.

2.1 The acesMIPS architecture

In this report we will demonstrate our framework on a MIPS 4k like processor architecture which we call acesMIPS (shown in Fig. 1). The architecture contains five pipeline stages – fetch, decode, operand read, execute and writeback. There are five parallel issue paths corresponding to two ALU units, one floating point unit, 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 32-bit wide general purpose register file and a 32-bit wide floating point register file, each containing 32 registers.

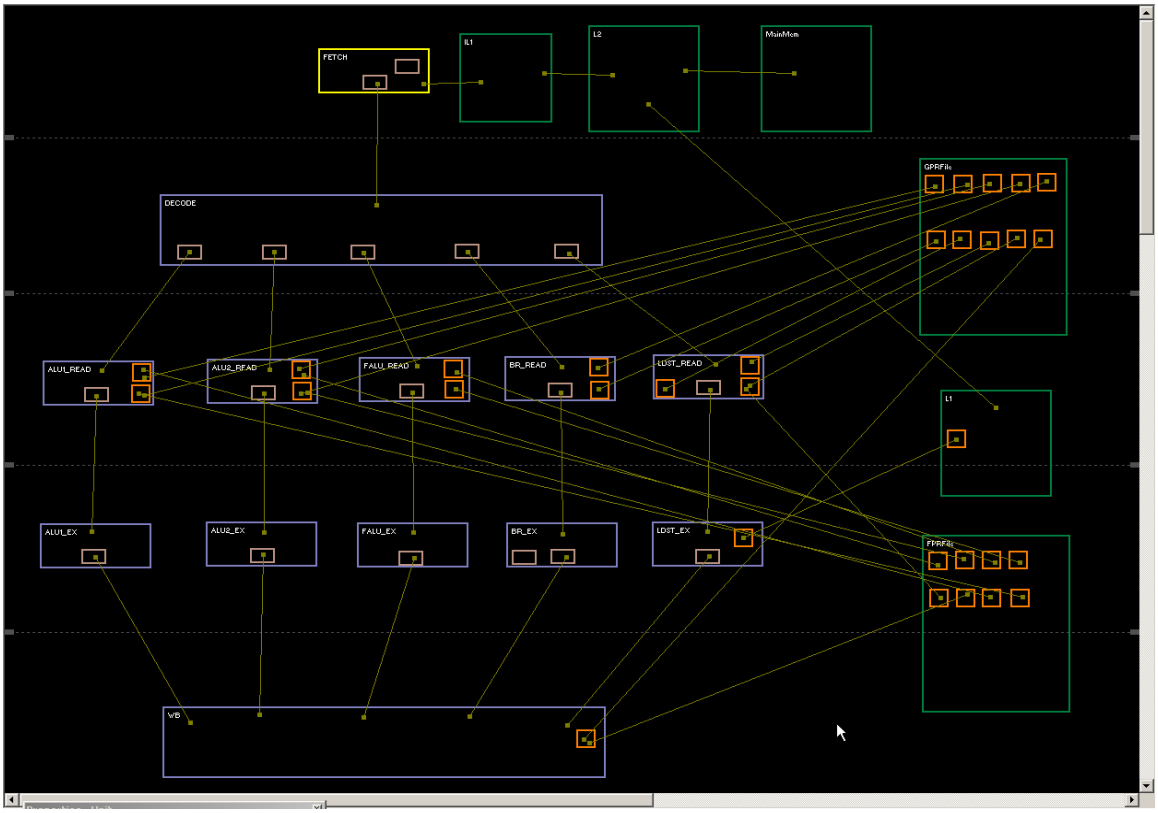

Figure 2: VSAT-GUI layout for the acesMIPS architecture

2.2 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. We describe below the various EXPRESSION sections and how they are captured using the VSAT-GUI and then used to generate code in the ADL description file.

2.2.1 Operations Specification

An instruction in the acesMIPS 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 opcode, 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.

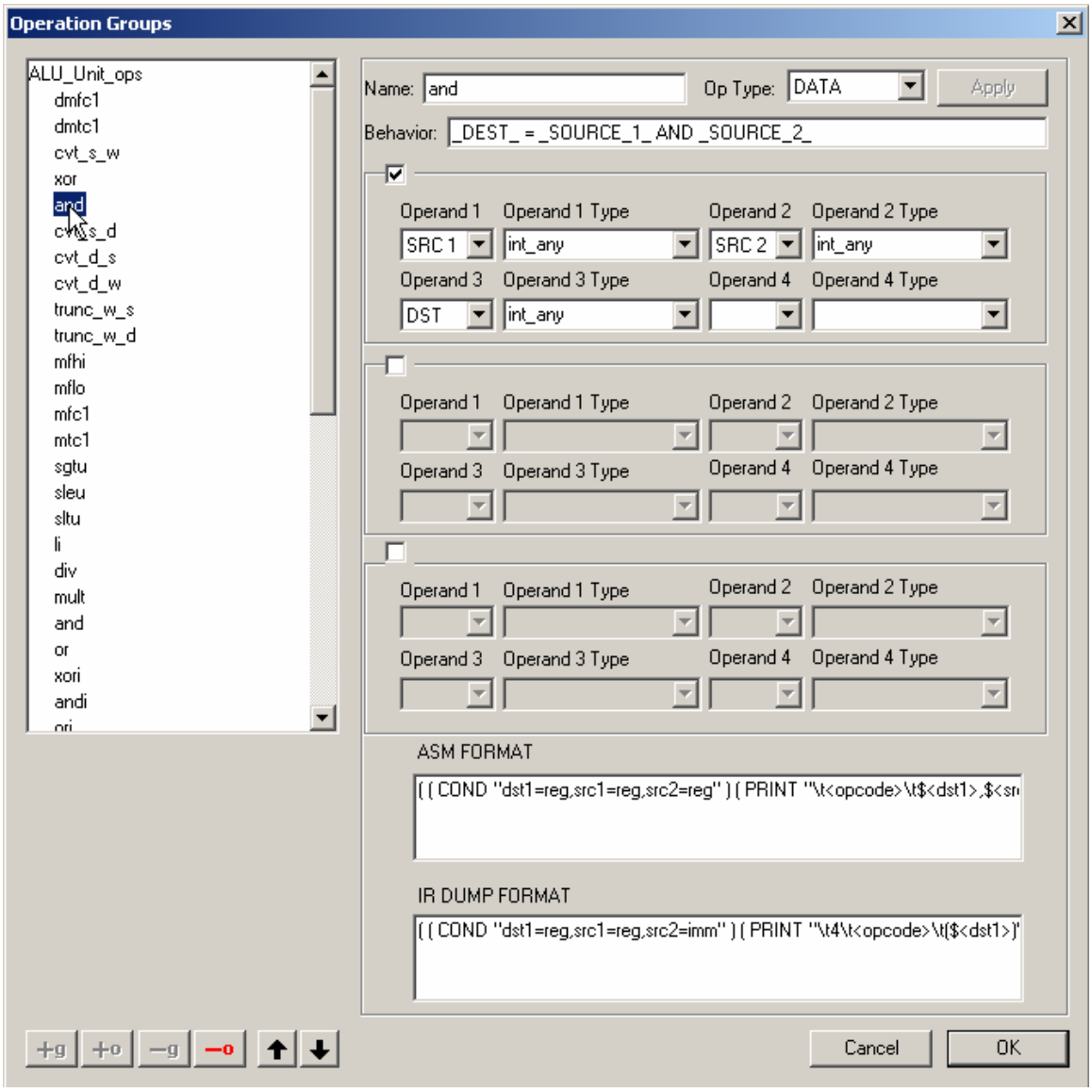

Figure 3: Operand Types for an Operation "and"

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. 3 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. 3 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)
 )
…
)
```

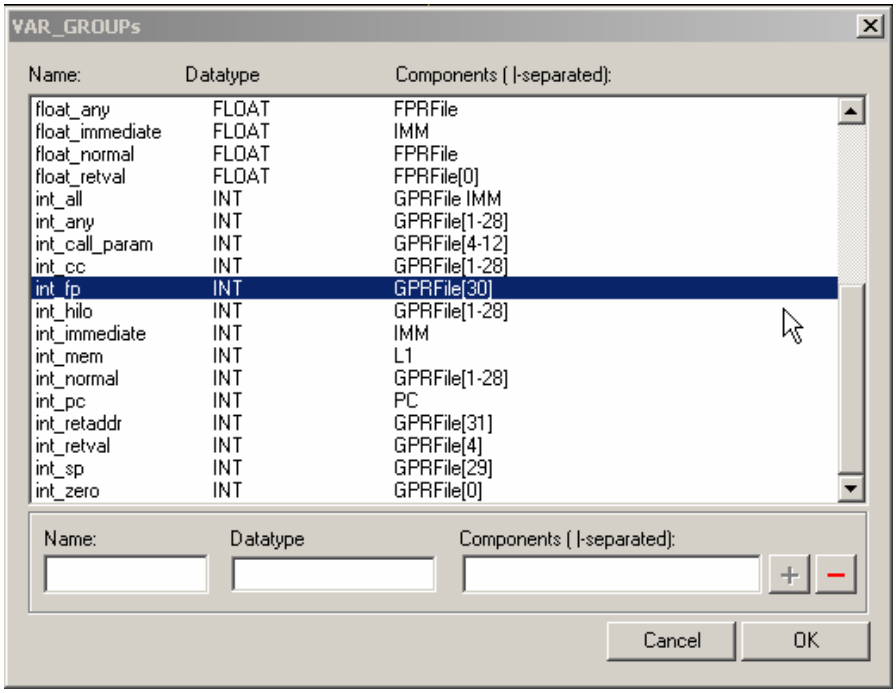

Figure 4: Setting VAR_GROUPS

The **set VAR_GROUPS** option in the **Instruction Set** menu brings up the **VAR_GROUPs** dialog box (Fig. 4) 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 ScratchPad))
 (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]))
)
...
)
```

2.2.2 Instruction Description

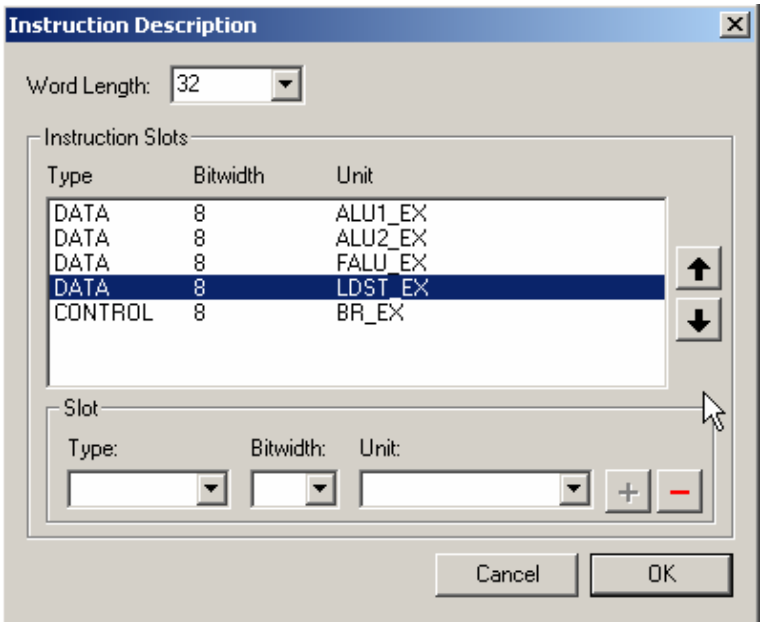

Figure 5: VLIW Instruction Template

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 acesMIPS architecture there are 4 slots for data operations (2 ALU operations, 1 FALU operation, 1 LDST operation) and 1 slot for Control operation. A valid VLIW instruction of word length 32 comprises of any four out of these 5 possible slots.

The **set Instruction Description** option in the **Instruction Set** menu brings up the Instruction Description dialog (Fig. 5) which is used to specify this subsection. 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 32)
  (SLOTS
   ((TYPE DATA) (BITWIDTH 8) (UNIT ALU1_EX))
   ((TYPE DATA) (BITWIDTH 8) (UNIT ALU2_EX))
   ((TYPE DATA) (BITWIDTH 8) (UNIT FALU_EX))
   ((TYPE DATA) (BITWIDTH 8) (UNIT LDST_EX))
   ((TYPE CONTROL) (BITWIDTH 8) (UNIT BR_EX))
  )
)
```

2.2.3 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.

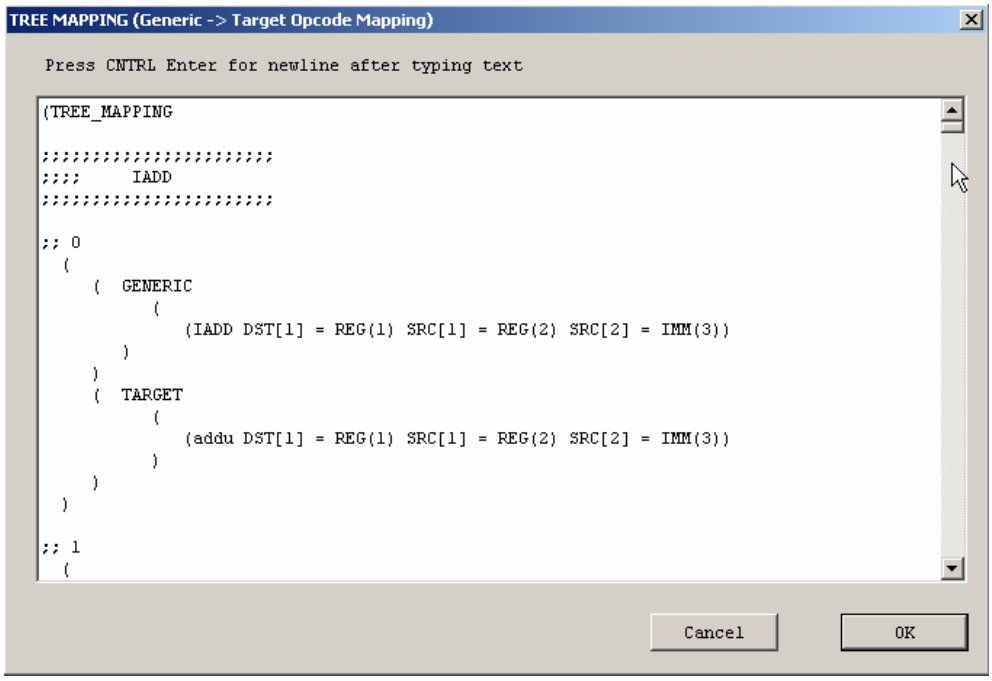

Figure 6: Tree Mapping

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 6) 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))
            )
    )
  )
...
)
)
```

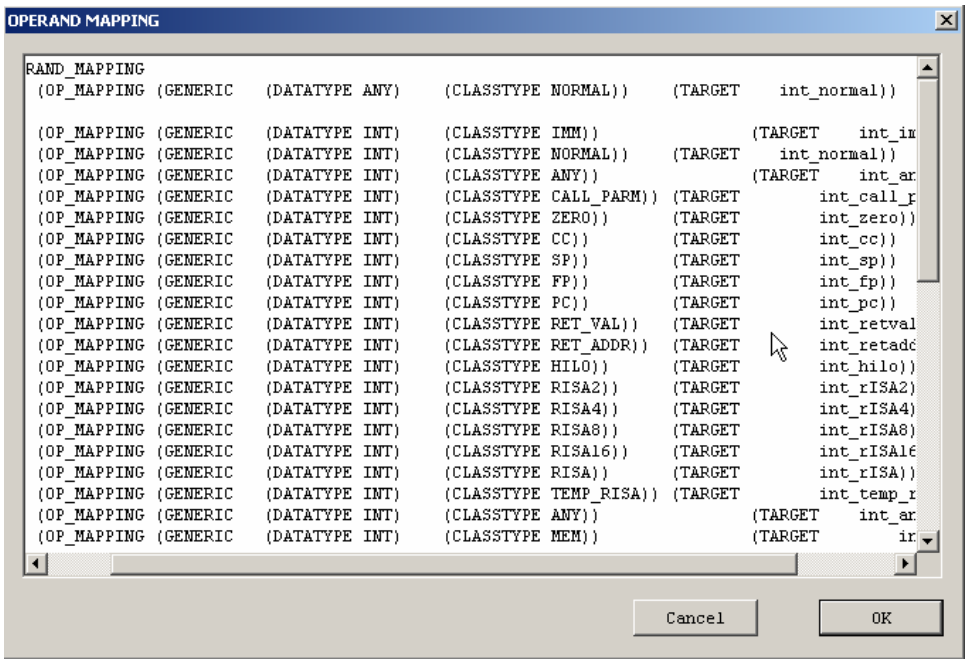

Figure 7: Register Class mappings for Operands

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 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 RISA2))      (TARGET   int_rISA2))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE RISA4))      (TARGET   int_rISA4))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE RISA8))      (TARGET   int_rISA8))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE RISA16))     (TARGET   int_rISA16))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE RISA))       (TARGET   int_rISA))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE TEMP_RISA))  (TARGET   int_temp_rISA))
            (OP_MAPPING (GENERIC   (DATATYPE INT)    (CLASSTYPE ANY))        (TARGET   int_any))
           ...
)
...
)
```

2.2.4 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.

2.2.4.1 Unit (Functional Unit)

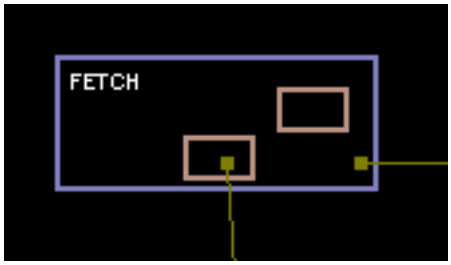

Figure 8: Fetch unit

A Functional unit has the following attributes

CAPACITY – size of reservation station

TIMING – time taken for operations to pass through
OPCODES – opcode groups allowed to pass 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. 8 shows the Fetch unit from the acesMIPS 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 …)
…
 (SUBTYPE STORAGE … InstStrLatch PCLatch …)

 (FetchUnit FETCH
  (CAPACITY 1)
  (INSTR_IN 4)
  (INSTR_OUT 4)
  (TIMING (all 1))
  (OPCODES all)
  (LATCHES (OUT FetDecLatch) (OTHER pcLatch))
 )

  (InstStrLatch FetDecLatch
  )

  (PCLatch pcLatch
  )
```

Here we are declaring a Fetch unit 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. 8). InstStrLatch and PCLatch are the types of these latches respectively.

2.2.4.2 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 (see Section 2.2.6). 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 (Section 2.2.6) for more details.

2.2.4.3 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.

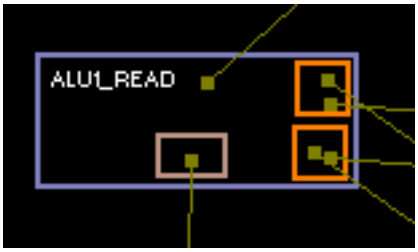

Figure 9: Two ports of the ALU1_READ unit

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. 9 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 ALU1_READ
...
  (PORTS Alu1ReadPort1 Alu1ReadPort2)
 )

 (UnitPort Alu1ReadPort1("_READ_")
  (ARGUMENT _SOURCE_1_)
  (CAPACITY 1)
 )

(UnitPort Alu1ReadPort2("_READ_")
  (ARGUMENT _SOURCE_1_)
  (CAPACITY 1)
 )
```

2.2.4.4 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.

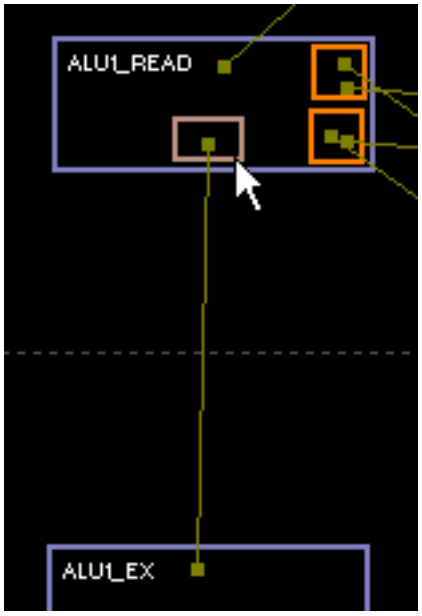

Figure 10: Alu1ReadExLatch

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 10 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:

```
(OpReadUnit ALU1_READ
 ...
  (LATCHES (OUT Alu1ReadExLatch))
 ...
)

  (OperationLatch Alu1ReadExLatch
  )
...
(ExecuteUnit ALU1_EX
 ...
  (LATCHES (IN Alu1ReadExLatch))
 )
```

Here OperationLatch is the type of the latch.

2.2.4.5 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. 11). 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 11: 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.

2.2.5 Pipeline and Data-Transfer Paths Description

Recall that architectural pipelining information in EXPRESSION is specified using the notion of pipeline and data transfer paths [5]. 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 architecture in Figure 1, 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 FALU_READ FALU_EX ))
   (read_execute3( PIPELINE BR_READ BR_EX ))
   (read_execute4( PIPELINE LDST_READ LDST_EX ))
   …
```

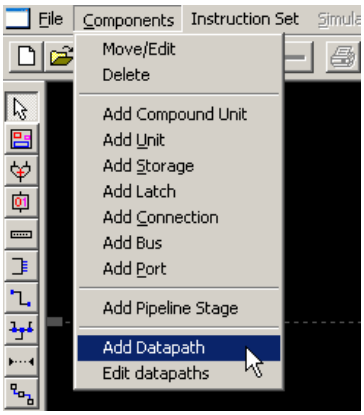

Figure 12 (a): Selecting 'Add Datapath' option

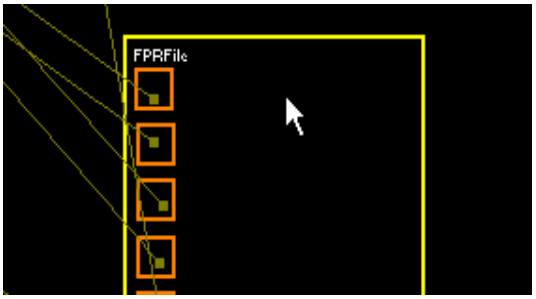

Figure 12 (b): Selecting FPRFile

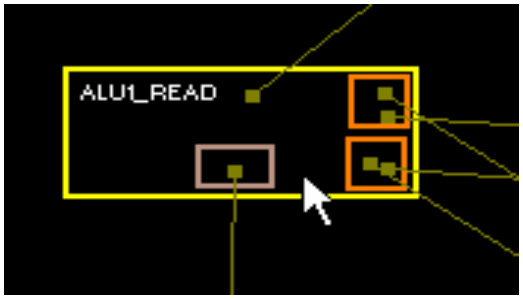

Figure 12 (c): Selecting ALU1_READ

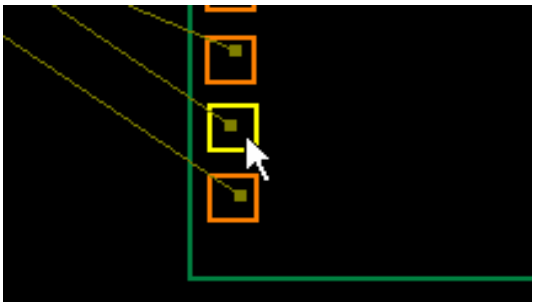

Figure 12 (d): Selecting port FprReadPort6 inside FPRFile

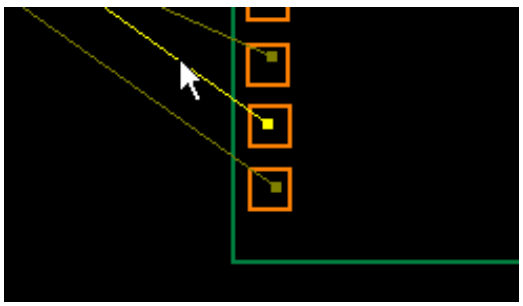

Figure 12 (e): Selecting connection element FprReadPort6ALU1ReadPort1

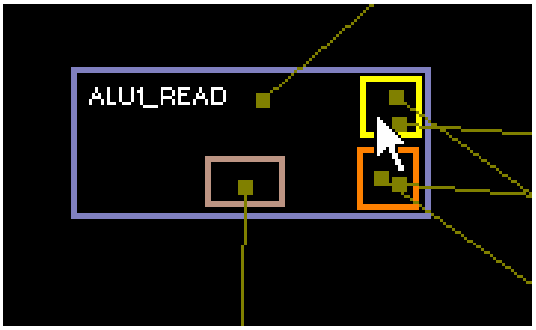

Figure 12 (f): Selecting port ALU1ReadPort1 inside ALU1_READ

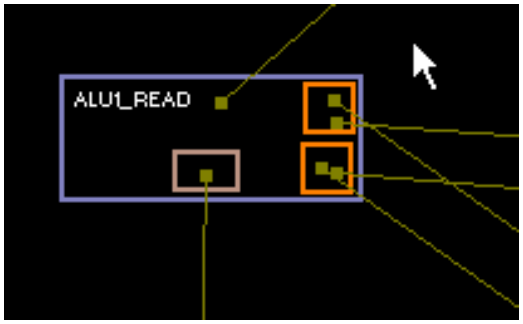

Figure 12 (g): Finish by right clicking mouse anywhere on screen

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 Datapath** 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 12 shows the sequence of actions to be performed to add a datapath between the ALU1_READ component and the FPRFile register file. The code generated in EXPRESSION for this datapath is given below:

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

The code generated for all the datapaths 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 FprReadPort7 FprReadPort7ALU1ReadPort2Cxn Alu1ReadPort2)
   ( FPRFile ALU2_READ FprReadPort1  FprReadPort1Alu2ReadPort1Cxn Alu2ReadPort1)
   ( FPRFile ALU2_READ FprReadPort2 FprReadPort2Alu2ReadPort2Cxn Alu2ReadPort2)
   ( FPRFile FALU_READ FprReadPort3 FprReadPort3FaluReadPort1Cxn FaluReadPort1)
   ( FPRFile FALU_READ FprReadPort4 FprReadPort4FaluReadPort2Cxn FaluReadPort2)
   ( FPRFile LDST_READ FprReadPort5 FprReadPort5LdStReadPort3Cxn LdStReadPort3)
   ( GPRFile ALU1_READ GprReadPort1  GprReadPort1Alu1ReadPort1Cxn Alu1ReadPort1)
   ( GPRFile ALU1_READ GprReadPort2 GprReadPort2Alu1ReadPort2Cxn Alu1ReadPort2)
   ( GPRFile ALU2_READ GprReadPort3 GprReadPort3Alu2ReadPort1Cxn Alu2ReadPort1)
   ( GPRFile ALU2_READ GprReadPort4 GprReadPort4Alu2ReadPort2Cxn Alu2ReadPort2)
   ( GPRFile BR_READ GprReadPort5 GprReadPort5BrReadPort1Cxn BrReadPort1)
   ( GPRFile BR_READ GprReadPort6 GprReadPort6BrReadPort2Cxn BrReadPort2)
   ( GPRFile LDST_READ GprReadPort7 GprReadPort7LdStReadPort1Cxn LdStReadPort1 )
   ( GPRFile LDST_READ GprReadPort8 GprReadPort8LdStReadPort2Cxn LdStReadPort2 )
   ( GPRFile LDST_READ GprReadPort9 GprReadPort9LdStReadPort3Cxn LdStReadPort3)
   ( LDST_EX L1 LdStReadWritePort LdStMemCxn L1ReadWritePort)
   ( WB FPRFile WbWritePort WbWritePortFprWritePortCxn FprWritePort)
   ( WB GPRFile WbWritePort WbWritePortGprWritePortCxn GprWritePort)
  )
 )
)
…
```

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

2.2.6 Memory Subsystem

This section is used to specify the attributes of the various storage components in the memory subsystem [7]. 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

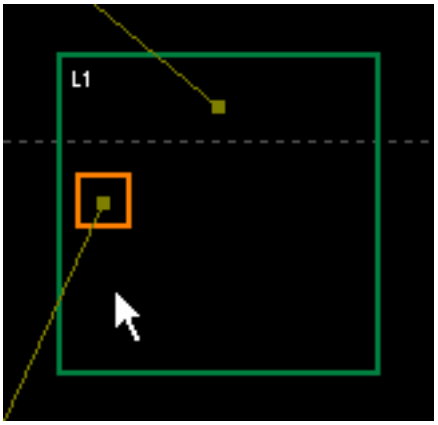

Figure 13: L1 cache with port

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. 14). 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 acesMIPS architecture (Fig 13), the architecture section contains the ports associated with the storage as shown below:

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

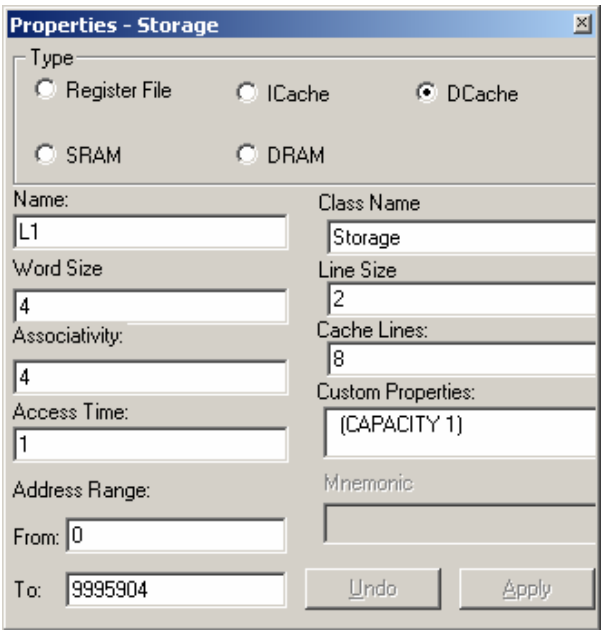

Figure 14: Properties window for L1 cache

Code is also generated for the storage section after specifying the storage attributes (Fig. 13) in the Properties window. This generated code is shown below.

```
(L1
```

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

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 retargetable compiler and simulator. The EXPRESSION parser builds an IR representation from the description file. This internal representation is processed to generate C++ intermediate files.

3.1. EXPRESS retargetable complier

The important phases of EXPRESS compiler are: Instruction Selection, Scheduling and Register Allocation. 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 phase which we call pipelined trailblazing [3] is based on the connectivity of the units specified. 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. 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 back-end 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 retargetable simulator called SIMPRESS.

3.2. SIMPRESS retargetable 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 DecodeUnit 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.

4. Architecture Exploration using the EXPRESSION framework

We present in this section, some of the exploration directions, which are deemed to be important by a system designer. For a step by step description on how to perform these explorations for the acesMIPS architecture, refer to the EXPRESSION User Manual [14].

An architectural modification can affect another architectural change positively or negatively. So, the designer has to do a trade-off between different performance goals, respecting the architectural constraints. The different architecture explorations comprise Instruction Set Architecture exploration, Micro-architecture exploration and Memory architecture exploration.

4.1. ISA Exploration

Most of the instructions of a target machine are obtained from the generic instruction set by one-to-one mapping of generic to target operations. When two or more generic operations combine together to form a target operation, we call the target operation, a complex operation. The target instruction set can be made richer by incorporation of large number of useful complex operations.

Register accessibility plays an important role in instruction set design. The number of supported opcodes can be increased by decreasing the accessibility to registers. However, decreasing the register accessibility can lead to spilling due to increased register pressure. A user can study the instruction set design trade-offs by varying the register accessibility of different operations.

A VLIW instruction consists of a fixed number of slots, which are placeholders for parallel operations. All the operations in a VLIW instruction are issued in parallel because they don't have any inter-dependence and there are resources to execute them in parallel. The slots of a VLIW instruction can be varied and the toolkit can be used to find an instruction template suitable for a given application.

4.1.1 Adding New (Complex) Operations

A complex operation is useful for a given application, when a sequence of operations forming the complex operation is frequently used. A profiler can come up with useful complex operations to be added to a base instruction set. Another advantage of adding a complex operation is that it gets rid of extra fetch delays.

Instruction Selection plays a pivotal role in converting a set of simple generic operations into a complex target operation. This is based on a tree-based mapping rule where the priority of mapping is determined by the order of specified rules. Below we give an example of a rule for the new target operation 'mac' (multiply and accumulate). As shown, this operation is a combination of three simple generic machine operations – IMUL, MFLO and IADD, described in the EXPRESSION generic machine model in [14].

```
(
   ( GENERIC
     (
         (IMUL DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = REG(3))
         (MFLO DST[1] = REG(4) SRC[1] = REG(2))
         (IADD DST[1] = REG(5) SRC[1] = REG(6) SRC[2] = REG(4))
     )
   )
   ( TARGET
     (
         (mac DST[1] = REG(5) SRC[1] = REG(2) SRC[2] = REG(3) SRC[3] = REG(4))
     )
   )
)
```

The rule for mac above should be specified before the rules of mult and addu, which are as follows:

```
(
   ( GENERIC
     (
         (IMUL DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = REG(3))
```

```
        )
    )
    ( TARGET
      (
          (mult DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = REG(3))
      )
    )
)

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

This will ensure that whenever there is an opportunity to generate a 'mac' operation, the compiler will generate it.

A complex operation usually needs more number of input ports than the constituent simple operations. Consequently, addition of new operations may need addition of new read ports in the register file. For instance, unlike the operations already present in the acesMIPS architecture, the 'mac' operation has three sources. However, there are only two input ports in the ALU units. We need to add another read port to each of the ALU units and then bind the operation group containing the 'mac' operation to the units.

4.1.2 Changing Register Accessibility

Individual operands of each operation are mapped to register classes. These register classes effectively partition the register file and have a unique mapping to a particular set of registers. Performance can be affected if the register accessibility of the operations is increased to an extent that results in elimination of register spills.

4.1.3 Changing Slots in an instruction

The slots in an instruction can be changed depending on the availability of resources. Both resources and slots can be added to the base architecture. When there is a new pipeline path owing to the increase in the number of slots, the Pipelined Trailblazing phase will automatically retarget the compiler to take advantage of the extra slot.

4.2 Pipeline Exploration

An architecture can be modified by changing its pipeline. The pipeline changes can be made by simply adding a new functional unit and having a new pipeline path go through the functional unit. One can also add or remove a functional unit from an existing pipeline path.

4.2.1 Adding a Single-cycle/Multi-cycle Functional unit

A new functional unit can be added as a single-cycle, multi-cycle or a pipelined unit. An OP_GROUP containing multi-cycle operations is linked with a multi-cycle functional unit. Adding a multi-cycle functional unit on a separate pipeline path increases the available parallelism, but can also increase the latency of operations since it is not pipelined.

4.2.2 Adding a New Pipelined Functional Unit

Adding a new pipeline path helps increase the parallelism in the datapath. The parallel resource can be a single-cycle/multi-cycle functional unit or a pipelined unit. A multi-cycle operation can be equivalently performed on a pipelined functional unit which will lead to increase in the number of pipeline stages. Pipelining a multi-cycle operation should lead to increase in performance in cases where the multi-cycle operation is extensively used.

4.3 Memory Subsystem Exploration

The memory subsystem consists of data and instruction caches, and main memory (DRAM) modules. All of these components are fully configurable. There are several different explorations possible with the memory subsystem. These possibilities are discussed in the following sections.

4.3.1 Changing Access Times

Access times of caches and main memory have a big impact on system performance. For instance, using a cache with lower access time can drastically improve the latency of the memory subsystem. The access time of every memory subsystem component can be modified by varying the ACCESS_TIMES attribute of the component

4.3.2 Changing Associativity

Associativity in caches is an important parameter which can affect miss rate and hit time. Greater associativity can come at the cost of increased hit time. By varying the ASSOCIATIVITY attribute of caches, the impact of this parameter on system performance can be determined.

4.3.3 Changing Sizes

Sizes of all the memory subsystem components can be varied by changing the SIZE attribute of the component. Increasing component sizes generally improves performance but at a cost – that of increased access time.

4.3.4 Adding/Deleting Memory Modules

It is possible to add new memory modules, for example an L3 cache between the L2 cache and the main memory module. It is also possible to delete any of the existing modules, for example the L2 cache, connecting the L1 caches directly to main memory.

5. Experiments and Results

The benchmarks that were used for testing the EXPRESSION framework comprise the following:

- Livermore Loops. (Benchmarks/LLs)
- Multimedia kernels. (Benchmarks/MMs)

We present here the result of running the Livermore Loops on the **acesMIPS** architecture and performing the architecture explorations discussed in the previous section.

Our first experiment demonstrates an increase in performance when pipeline trailblazing [4] is enabled. Figure 15 shows the percentage decrease in cycle count when trailblazing is enabled.

Trailblazing exploits ILP aggressively and results in a decrease in execution cycle count for the benchmarks.

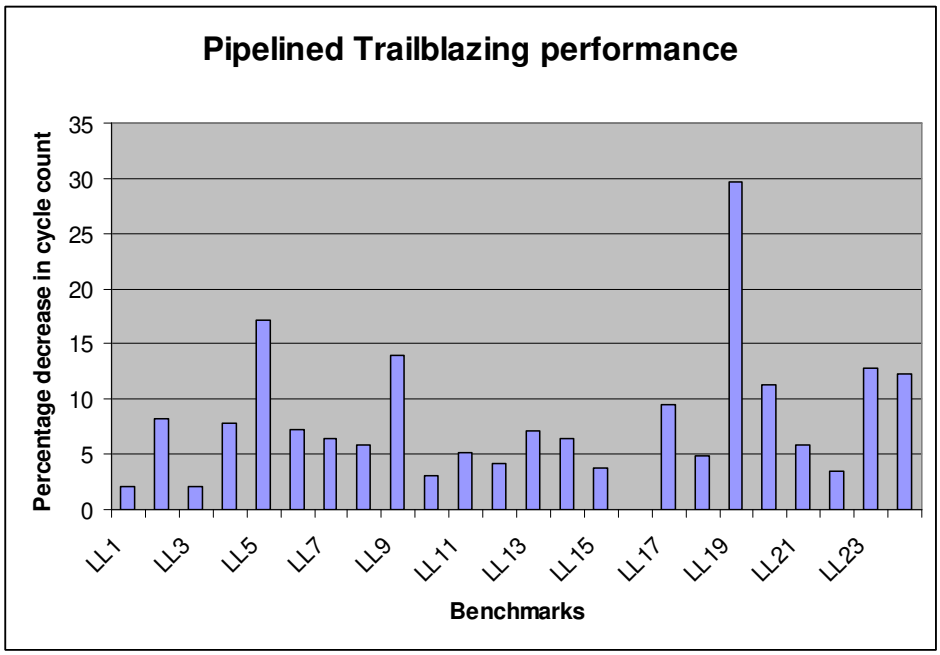

Figure 15: Change in performance after enabling Pipeline Trailblazing

In Figure 16 we show the results of adding a MAC operation to the instruction set. Benchmarks which have several addition and multiplication operations benefit from this instruction which combines the two operations into one instruction, while most of the benchmarks remain unaffected. In LL20, there is degradation in performance because of poor scheduling in the presence of a MAC operation.

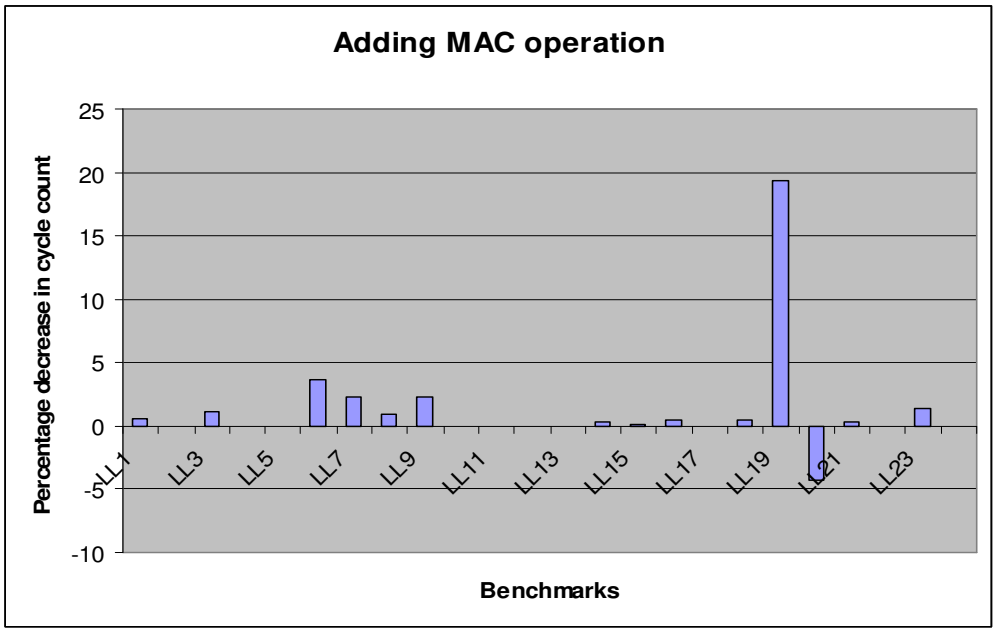

Figure 16: Change in performance after adding a MAC operation

Next we examine the impact of deleting the second ALU functional unit. Figure 17 shows that the performance of most benchmarks deteriorates because the parallelism has been reduced – whereas earlier two ALU operations could be issued simultaneously, now they will be issued one after the other, increasing the cycle count.

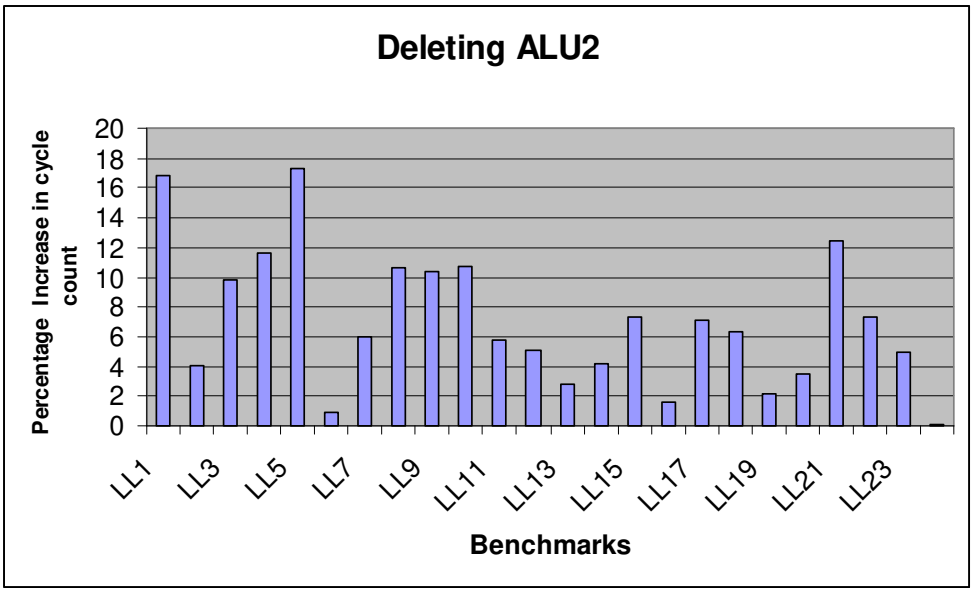

Figure 17: Change in performance after deleting ALU2

If we replace the second ALU functional unit by a 2 cycle multiplier, the latencies of instructions that flow through the unit will increase and as a result, the execution cycle count of the benchmarks increases as shown in Figure 18.

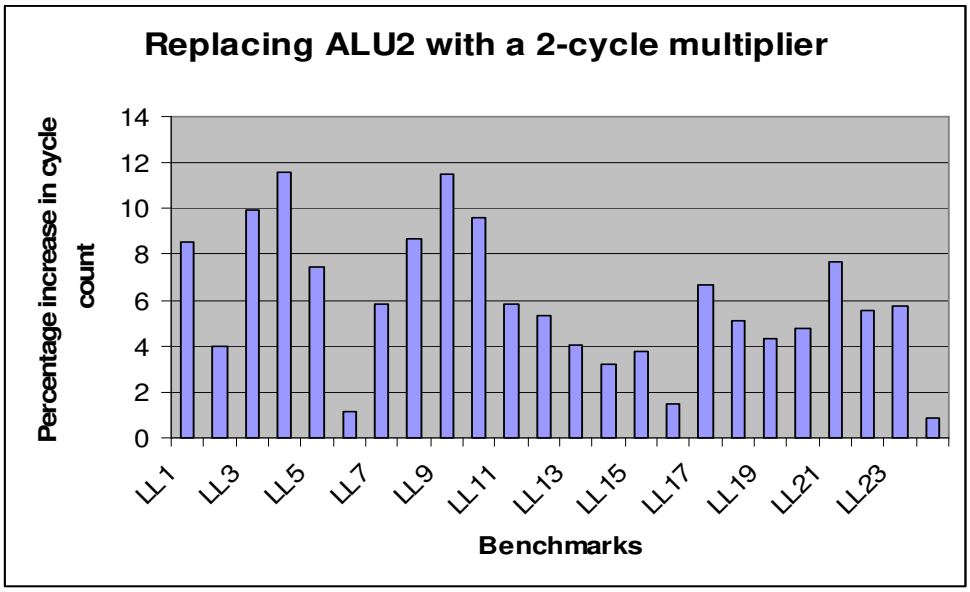

Figure 18: Change in performance after replacing ALU2 with 2-cycle multiplier

Since pipelining increases throughput, it is intuitive to expect that if the two cycle multiplier is pipelined, the performance will improve. Figure 19 shows the improvement in performance on pipelining the 2 cycle multiplier. Some of the benchmarks don't make use of the multiply operation and for these benchmarks there is no change in performance on pipelining.

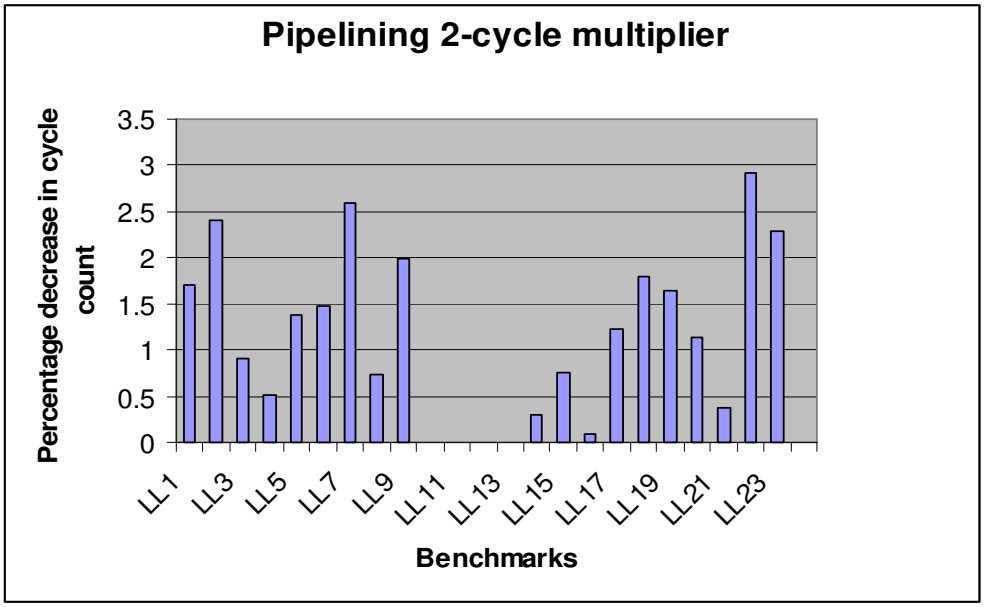

Figure 19: Change in performance after pipelining multi-cycle multiplier

We next explore the effect of decreasing the DRAM access time from 50 cycles to 45 cycles. Figure 20 shows that all of the benchmarks are affected by the decrease, and performance improves because of faster main memory access.

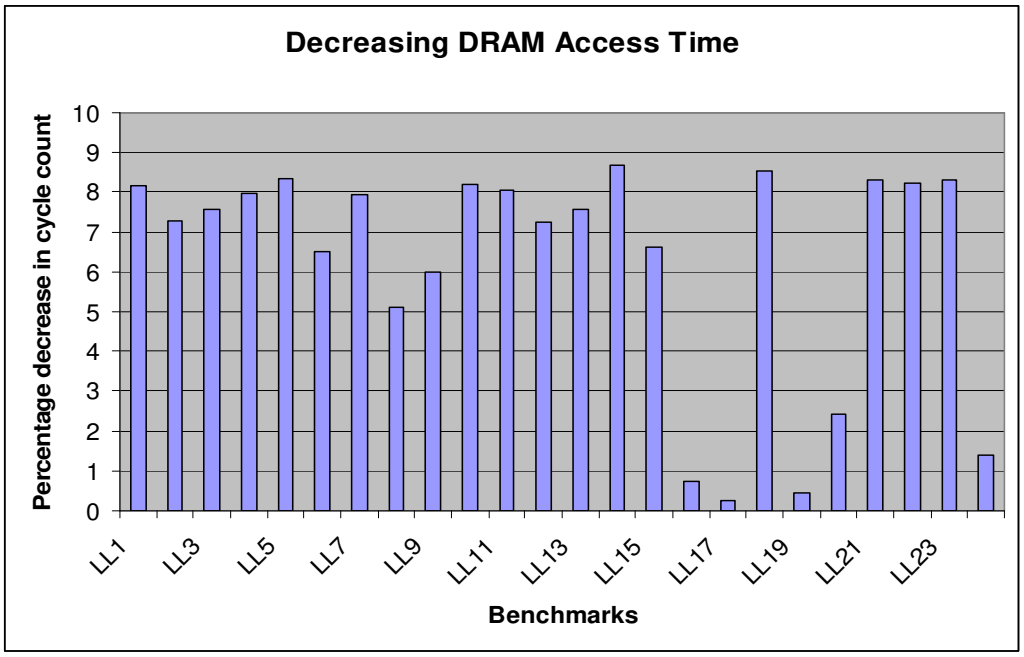

Figure 20: Change in performance after decreasing DRAM access time

Finally, we look at the impact of adding a unified L2 cache to the memory hierarchy. Figure 21 shows that the performance of a majority of the benchmarks improves with the inclusion of a second level of cache. In some cases the performance actually decreases because L2 cache misses dominate.

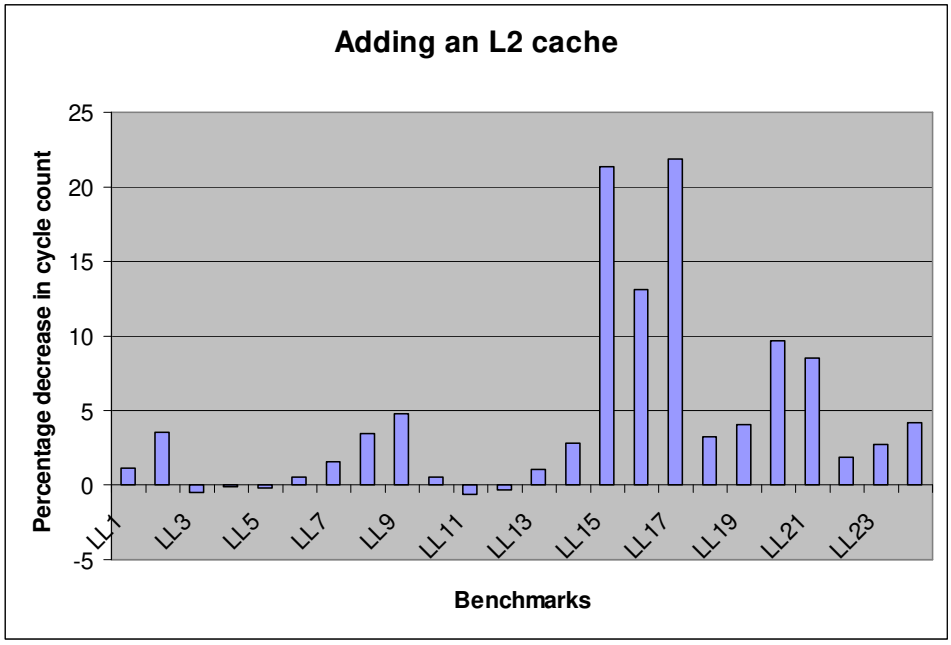

Figure 21: Change in performance after adding L2 cache

6. Conclusion

In this technical report we presented an automated framework for architecture exploration and used this framework to perform various explorations on a MIPS 4K like processor which we call acesMIPS. We first showed how the architecture description can be easily and intuitively captured in EXPRESSION using a graphical front-end tool. We then showed how tweaking the architecture with the help of this graphical front-end tool can allow us to explore different architecture configurations, which is a crucial activity for system architects looking to measure tradeoffs and obtaining optimal system configurations as early as possible. With Systems-on-Chip designs becoming ever more complex, the importance of a framework that allows quick and reliable architecture exploration and performance evaluation cannot be underestimated.

Acknowledgements

We would like to gratefully acknowledge Ashok Halambi, Peter Grun, Asheesh Khare, Nick Savoiu and other ACES team members for their work on the initial EXPRESSION framework. We would also like to thank everyone in the ACES lab for their contribution in testing and giving valuable feedback that helped us improve the framework.

References

[1] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt and A. Nicolau. “V-SAT: A visual 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 for System Level Design Exploration”, *Technical Report*.

[3] P. Grun, A. Halambi, N. Dutt and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. *In ISSS, San Jose, CA, 1999*

[4] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. *In ICPP, St. Charles, IL, 1993*

[5] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability ", *DATE 99*

[6] Ashok Halambi, Nikil Dutt and Alex Nicolau "Customizing Software Toolkits for Embedded Systems-On-Chip", DIPES 2000

[7] Prabhat Mishra, Peter Grun, Nikil Dutt, and Alex Nicolau "Memory Subsystem Description in EXPRESSION ", UCI-ICS Technical Report #00-31

[8] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau "Modeling and Verification of Processor Pipelines in SOC Design Exploration" In Proc. of 4rd International High Level Design Validation and Test Workshop (HLDVT'99), pp. 10--16, Nov. 1999

[9] Ashok Halambi, Aviral Shrivastava, Nikil Dutt and Alex Nicolau. "A Customizable Compiler Framework for Embedded Systems", SCOPES 2001

[10] A Khare “SIMPRESS: A Simulator Generation Environment for System-on-Chip Exploration” Masters Thesis, UCI-ICS 1999

[11] V. Zivojnovic et al. “LISA - machine description language and generic machine model for HW/SW co-design. In VLSI Signal Processing”, 1996.

[12] M. Freericks. “The nML machine description formalism.” TR SM-IMP/DIST/08, TU Berlin, 1993.

[13] George Hadjiyiannis , Silvina Hanono , Srinivas Devadas, “ISDL: an instruction set description language for retargetability”, Proceedings of the 34th annual conference on Design automation conference, p.299-302, June 1997

[14] P. Biswas, S. Pasricha, P. Mishra, A. Shrivastava, N. Dutt, A. Nicolau, “EXPRESSION User Manual version 1.0”, Feb 2003