# Architectural Modeling of Compressed Instruction Set in Retargetable Compiler-Simulator Framework

AUTHOR INFO REMOVED

Software For Embedded Systems
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

March 2000

## Abstract

*Compressed Instruction Set architecture refers to support for coexistence of instructions of different instruction widths. By re-encoding the most frequently used instructions in lesser number of widths code density can be increased at the expense of some extra work in the decode cycle. In this project we have provided support for Compressed Instruction Set in an existing re-targetable compiler-simulator framework, EXPRESSION. We tested the generic implementation on the motorola PowerPC architecture. We decided upon a Compressed Instruction Set for the same, and executed our compiler optimization to estimate the improvements in code size and impact on performance. We found encouraging improvements of upto 20% in the code size for not a setback in the performance.*

# Contents

**1 Introduction** **4**

**2 Compressed Instruction Set Concept** **4**
2.1 ARM Thumb Concept . . . . . 4
2.1.1 Hardware Aspects . . . . . 5
2.1.2 Software Aspects . . . . . 5
2.2 Flavors in Compressed Instruction Set Architecture . . . . . 7

**3 Retargetable Compiler-Simulator Framework** **7**
3.1 ADL based Design Flow . . . . . 8
3.2 Classification of ADLs . . . . . 9
3.3 EXPRESSION . . . . . 9

**4 Compressed Instruction Set Architecture** **10**
4.1 Profiling Results . . . . . 10
4.2 Design of Compressed Instruction Set . . . . . 10
4.3 The Compressed Instruction Set . . . . . 12

**5 Retargetable Compiler** **12**
5.1 Flow of the Compiler . . . . . 12
5.2 Instruction Selection . . . . . 14
5.3 Profitability Function . . . . . 15
5.4 Compression Pass . . . . . 16
5.5 Register Allocation . . . . . 16

**6 Results** **17**
6.1 Code Size Improvements . . . . . 17
6.2 Impact on Performance . . . . . 18

**7 Future Work** **19**

**A Compression Pass** **20**

## List of Figures

1 The Pipeline of ARM architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2 The Translation scheme of ARM Thumb instructions . . . . . . . . . . . . . . . . . . 5
3 Mapping of Thumb instructions onto normal ARM instructions . . . . . . . . . . . 6
4 Instruction Memory Organization for ARM . . . . . . . . . . . . . . . . . . . . . . 6
5 ADL-based Design Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
6 EXPRESSION ADL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
7 Profiling Results on Livermore Loops . . . . . . . . . . . . . . . . . . . . . . . . 11
8 EXPRESS flow for modeling CIS . . . . . . . . . . . . . . . . . . . . . . . . . . 14
9 An example: Iburg tree-pattern matching . . . . . . . . . . . . . . . . . . . . . . 14
10 An example: Control flow graph . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11 The Pipeline of ARM architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 18
12 The Pipeline of ARM architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 19

## List of Tables

1 Compressed Instruction Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

# 1 Introduction

Compressed Instruction Set architecture refers to support for coexistence of instructions of different instruction widths. By re-encoding the most frequently used instructions in lesser number of widths code density can be increased at the expense of some extra work in the decode cycle. A large variety of Embedded Systems differ from each other in a myriad of ways. For a designer to choose the best configuration for his application, rapid Design Space Exploration is a must. Architectural Description Languages aid by automatic generation of software tool kit consisting of a compiler and simulator to automate this process of Design Space Exploration. In this project we have provided support for Compressed Instruction Set in an existing re-targetable compiler-simulator framework. We intend to come up with a generic model for Compressed Instruction Set Architecture, and then specify and model that in a generic way so that all flavors of CIS can be recognized by the compiler. Also the compiler must be able to optimize code for all types of CIS on all architectures. By implementing the support for CIS in a generic way we have enabled the route for a rapid design space exploration of various choices of Compressed Instructions Sets in the same architecture, and over many architectures too. Inspired by the Thumb Instruction set of the ARM processor, we have studied the impact of such a Compressed Instruction Set on Motorola PowerPC. We profiled some benchmarks and chose some instructions that decided to include into the Compressed Instruction Set. The Instruction Selection phase of the compiler now needs to be aware of the presence of Compressed Instructions. Apart from this the Register Allocator might need to map some operands and all their other definitions and uses into a restricted set of registers. This causes an increase in the register pressure. We want to convert the operations

# 2 Compressed Instruction Set Concept

Motivated from the Thumb Concept in ARM processor, the term Compressed Instruction Set Architecture implies support for variable length instructions in the Instruction Set Architecture, and the compiler support for the same.

## 2.1 ARM Thumb Concept

The ARM7TDMI processor is a full scale 32 bit processor.[1] Thumb is an extension to the ARM instruction set. Thumb instructions are 16 bit instructions that co-exist and co-execute with the normal 32 bit instructions. 32 of the most frequently used instructions are re-encoded into 16 bits to make the Thumb instruction set. The most obvious advantage being the code size reduction. On execution, the 16 bit Thumb instructions are expanded into full length instructions and executed as normal 32 bit ARM instructions. Each Thumb instruction is uniquely mapped to normal ARM instruction. Thumb allows to keep the power of a 32 bit instruction set while at the same time benefiting from the advantages of code size improvements of 16 bit instruction set. The fact that there exists a unique ARM instruction for each Thumb instruction means that the decoding logic is very simple, and could be done in the decode cycle. Using the Thumb instruction set the designers are now in a better position to perform a trade-off between code-size and performance. They can

encode the performance critical areas in 32 bit instruction set and code-size critical areas of the code in compressed Thumb instructions.

### 2.1.1 Hardware Aspects

ARM is a simple 3 stage Fetch-Decode-Execute pipeline. The hardware needs to decompress the 16 bit Thumb instructions into 32 bit ARM instructions, decode them, and then execute them as normal ARM instructions. A decompressor logic is added before the decoder for the same purpose.

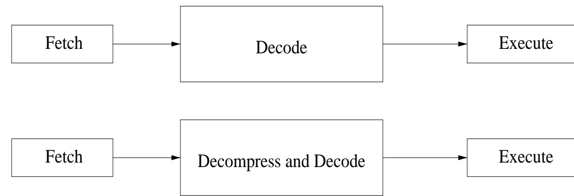


**Figure 1. The Pipeline of ARM architecture**

Since the ARM processor has a very simple three stage pipeline they can fit the decompressor logic into the unused phase of the clock in the decode stage. Therefore there is no timing overhead. But in general this decompressor logic, if not quite simple may cause elongation of cycle time.

The conversion from Thumb instructions to full length ARM instructions is done via a table lookup mechanism. An example of decoding an immediate add instruction is shown in Figure 2.

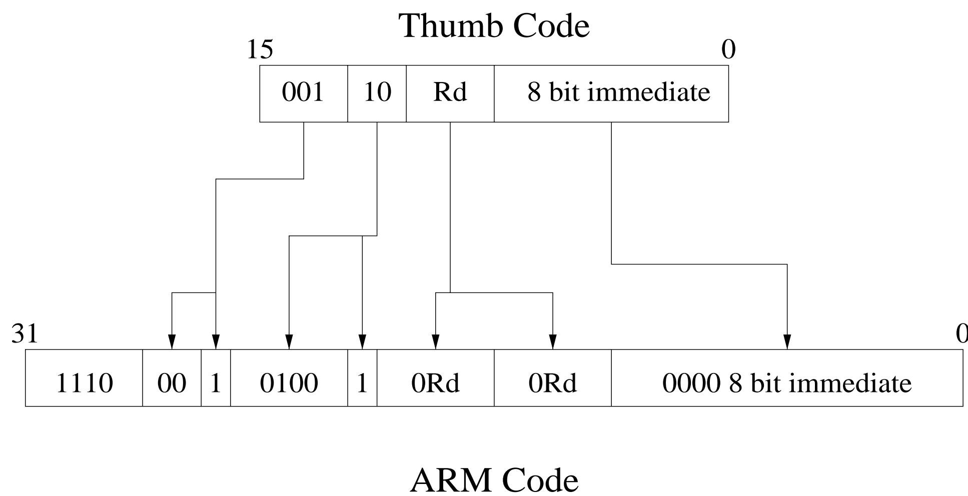


**Figure 2. The Translation scheme of ARM Thumb instructions**

### 2.1.2 Software Aspects

The Thumb instruction set consists of those instructions that

- occur frequently

- can be compressed into 16 bits
- compiler needs in order to get the best code density

Since the instructions are compressed to 16 bits from 32 bits, some instructions can access only a subset of registers. In place of 15 GPR's, Thumb instructions can support only 8 GPR's. The mapping of the Thumb registers onto ARM registers is shown in Figure 3

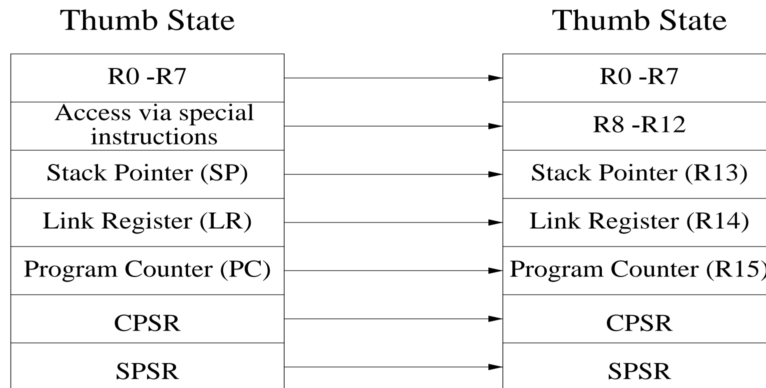


**Figure 3. Mapping of Thumb instructions onto normal ARM instructions**

The ARM processor works in one of two modes, the normal or the Thumb mode. The switching in between the modes is done via a special instruction *BX*. This means that the normal instructions and Thumb instructions can survive in the same memory space. An example of memory organization might look like the one in Figure 4.

**Figure 4. Instruction Memory Organization for ARM**

### 2.2 Flavors in Compressed Instruction Set Architecture

Generalizing on the concept of Thumb architectural support, we have developed the concept of Compressed Instruction Set Architecture. This essentially refers to co-existance of instructions of different length. Compiler needs to be aware of this property and should be able to do appropriate and optimal code generation to take full advantage of this facility both in terms of performance and code-size.

- **Compressed Instruction Length :** Compressed Instruction Sets could differ most primarily in the length of the compressed instruction length. We can use 8 bit compressed instructions, or 16 bit compressed instructions. Although smaller instruction width means that the resulting code size could be reduced, but this does not happen this way because the smaller the instruction width, lesser the instructions that can be represent.
- **Opcode Length :** The length of the opcode directly effects the number of instructions that can be represented. Other variations possible are fixed length opcodes or variable length opcodes. The disadvantage of the later is that the decompressing logic becomes difficult, so it may not possible to implement that in the same cycle as decode without increasing the clock cycle. On the other hand variable instruction length gives the flexibility to encode more and a larger variety of instructions.
- **Register Restrictions :** Since we are compressing the instructions, it might not be possible to have access the whole register file in the compressed instructions. In that case we might have to restrict the operands and/or the destinations to access only a subset of GPR's. There could be two flavors in this. Either the restriction could be imposed only on the destination of the compressed instructions, or all the operands can be restricted. If we constrain the possible register set for an instruction, all the uses of the variable will also have register constraints. This impacts the register allocation stage, and register pressure increases. Which causes more spilling.
- **Implied Operand :** In order to compress instructions, one or more operands could be implied. The implied thing could be a constant or a register, and could be a source or even a destination.

## 3 Retargetable Compiler-Simulator Framework

A huge variety of processors are available in the market today.[1], [10] Right from the full scale Pentium-IV processor to microprocessor chips used for fuel injection in cars. [9] The processors could be generic or very task specific. The processors vary in all possible ways.[7]

- **Architecture :** Harvard or Von Newman architecture. Further the processor could be superscalar or a RISC machine, or even an EPIC architecture.

- **Memory Configurations :** Processors differ in variety of memory hierarchies they support. The accompanying memory might have L1 cache only, or not have it, or even have special flash memory modules.
- **DSP-like features :** for example Split register files, complex addressing modes, all types of processors crop up catering to the specific needs of the application.
- **Special Units :** Some processors may even contains some specific units to perform specialized tasks. It may have a co-processor etc.

### 3.1 ADL based Design Flow

With the huge variety of processors in the market, when one wants to build an application, one might want to find out the processor best suited to their needs. Designer needs to explore all the processor-memory configurations to find out the best combination. The most naive approach to this may be to implement your application using a processor-memory configuration, and then iterate over all possible configurations till you find an acceptable solution. But each of these iterations takes quite a lot of time, which is not feasible in this world of decreasing time-to-market. But even with this approach it is not possible to experiment with the processor-memory design by changing some parameters of the same design. This motivated the need of ADL based design flows. ADL is an acronym for Architectural Description Language. A typical ADl based design flow looks like the one in figure Figure 5

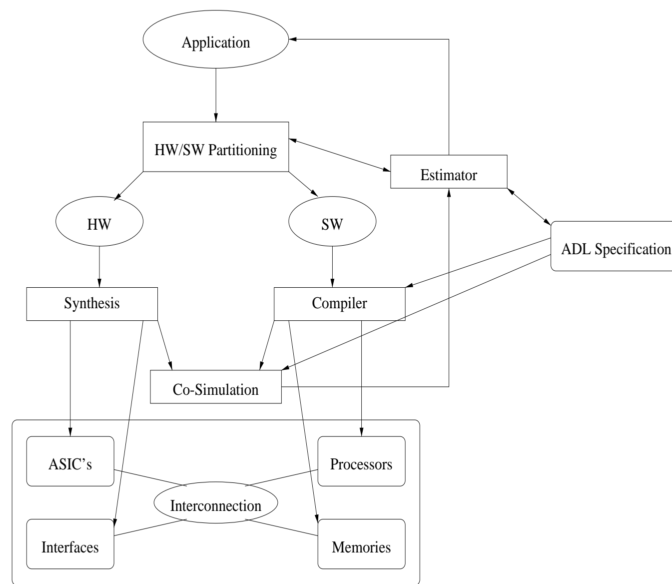


**Figure 5. ADL-based Design Flow**

In the ADL-based design flow[5], an ADL specification of the processor memory configuration is provided, and our objective is to choose the best ADL specification for our application whose

parts have to be partitioned into Hardware, and Software. The ADL description of the processor-memory configuration is used to generate compiler and simulator so that we can do the design space exploration as fast as possible. The feedback loop is the one from co-simulation to ADL specification via estimator. The traditional design flow breaks at this point. Since we have a off-the-shelf processor and it's compiler, we cannot experiment with any changes in the processor design and memory hierarchy. And exactly here the ADL based design flow comes to rescue.

### 3.2 Classification of ADLs

We did a survey on contemporary ADLs and studied their features. Based on the use of ADL's they can be divided into three broad categories.[8]

- **Compiler Oriented ADLs :** These ADLs support the compiler designer. They contain all the information required to build a compiler for the specification. So these ADLs not only contain information about the Instruction Set Architecture of the processor, but also tell about the Datapath, pipeline, the semantics of the operations etc.. MDES and EXPRESSION are some compiler oriented ADLs.

- **Simulator Oriented ADLs :** These ADLs support the automatic generation of simulator. Information might not be enough to generate a compiler, but at least a functional simulator will be easy to build using this description. Examples of Simulator Oriented ADL is LISA.[3]

- **Synthesis Oriented ADLs :** Languages like MIMOLA are synthesis oriented. Such languages usually specify components of hardware in HDL's so that synthesis is simplified. Theoretically it is possible to derive the syntax and semantics of whole system from this description, but is hardly pragmatic.

### 3.3 EXPRESSION

EXPRESSION[4] is a Compiler-Oriented Architectural Description Language. Here the stress is on specifying the structure of the processor at as high level as possible to be able to generate and optimizing compiler and at least a cycle accurate simulator. EXPRESSION description of a processor-memory configuration consists of

- **Behavior Specification :** This section describes both the syntax and semantics of the Instruction Set Architecture of the processor.

- **Structure Specification :** This section describes the structure of the processor. This describes all the other components of the processor, has information about the pipeline structure, datapath, and memory subsystem. The information about the components in the processor is at RTL level, abstracting out the low level details and retaining only those that are must to generate an optimizing compiler.

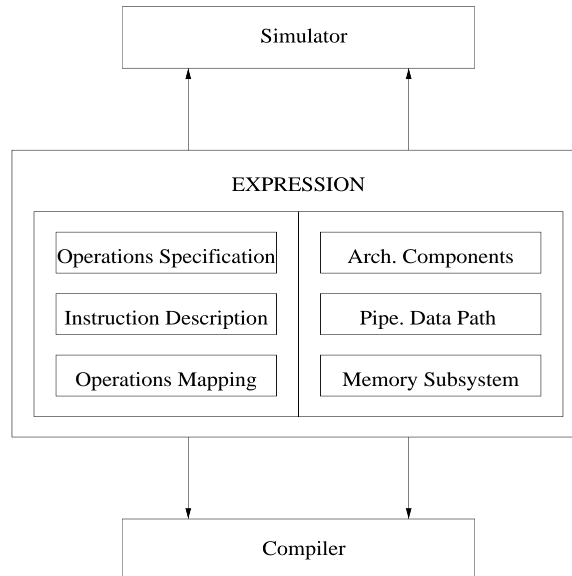


**Figure 6. EXPRESSION ADL**

## 4 Compressed Instruction Set Architecture

We have generalized the concept of Compressed Instruction Set Architecture, and implemented support for that in the Motorola PowerPC. We have tried to keep this model as general as possible. That is, in this framework it is possible to implement a wide variety of flavors in the Compressed Instruction Set Concept.

### 4.1 Profiling Results

We performed a profile run on some Livermore loops to study the frequency of occurrence of functions. The results of the profiling is shown in Figure 7.

The first observation to make was that ADDI was the most frequently used instruction. But ADDI has an immediate operand, so it is not possible to generalize the encoding for ADDI in 16 bits. Actually any given ADDI cannot be compressed into 16 bits. So we divided the ADDI into ADD0, the ones that occur with one operand 0, and ADD4, ones that have one operand as 4. Motorola PowerPC does not implements the move operation using ADDI with a 0 argument. And the ones with one argument as 4 are used for memory address computations. That is why these instructions occur so often. Some other operation like *SLW* do not appear so frequently.

### 4.2 Design of Compressed Instruction Set

The decision to include or exclude an instruction into the Compressed Instruction Set is dependent on various issues.

- **Width of CIS Format** : The first step in designing a CIS is to decide upon the width of CIS. Even this parameter can be changed and the results studied to come up with an optimal

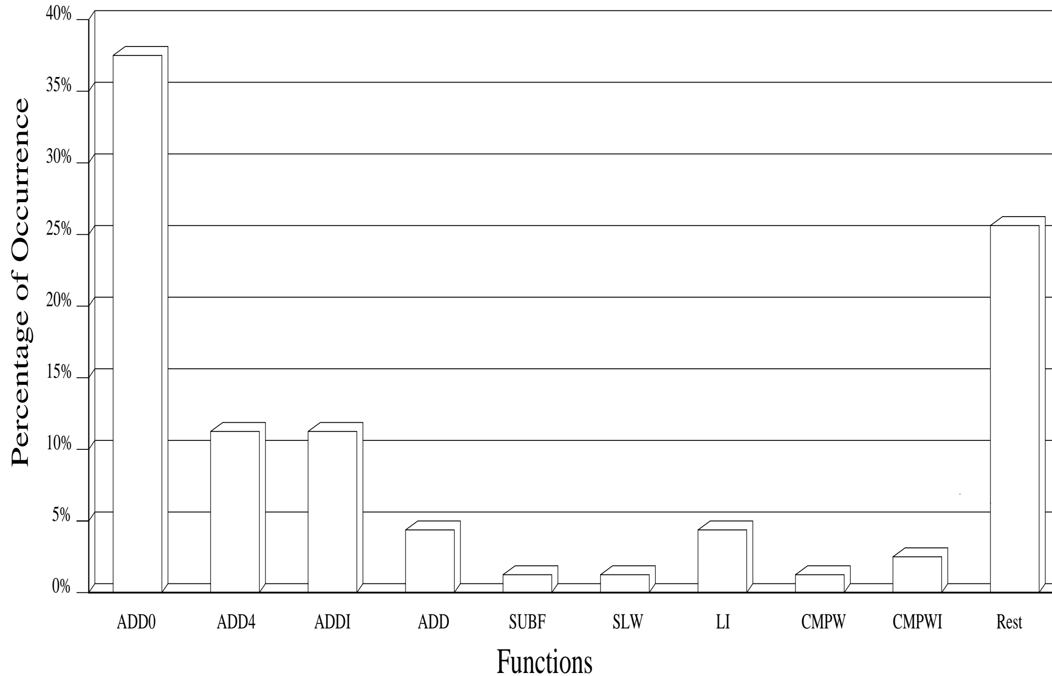


**Figure 7. Profiling Results on Livermore Loops**

width.

- **Safety of CIS** : Some instructions like load, store, divide may produce exceptions when executed. In such a case the processor has to roll back to the state before the instruction was executed. In case of Compressed Instructions the address of instruction does not obey the word boundary. If special care is not taken to address to this issue, such instructions cannot be included in the Compressed Instruction Set.

- **Frequency of occurrence** : Including a more frequently occurring instruction in the Compressed Instruction Set will lead to better reduction in code size.

- **Number of Operands** : The number of operands in an instructions limits its possibility of inclusion in the CIS. In the current implementation we support at the most 3 operands.

- **Size of Immediate Operand** : 16 bits pose a serious constraint on the size of the immediate operand. The constraint for immediate operand is particularly severe, because even with 8 bits of immediate operand, we can only support numbers in the range (-128 to + 127).

- **Grouping of Compressible Instructions** : Instruction like SLW, SUBF, CMPW and CMPWI occur relatively infrequently, but they usually occur with compressible instructions. Including them in the CIS increases the size of the compressible block of instructions. So it is necessary to include them.

- **NOP Compressed Instruction** : A *NOP* operation has to be explicitly included in the CIS. This is because there can be only even sized blocks of compressed instructions. This is because we do intend to respect the word boundaries in the memory. If there are odd number of instructions in a compressed block, this implies that the next normal 32 bit instruction will not start from a word boundary. To eschew from this situation we explicitly include a *NOP* operation in CIS.

- **Mode-Change Compressed Instruction** : The processor works in two modes, normal or the compressed mode. The change of mode is done through a *BX* operation, which changes the mode from normal to compressed mode. But when we need to revert back we need a similar instruction in CIS. The inclusion of last two instructions is specially important because this alleviates the strong constraint on the number of operations that can be included in CIS.

- **Increase in Register Pressure** : In case of a compressed 3 operand instruction it is difficult to incorporate all the registers for all the operands. In this case some of the operands could be restricted to be mapped to only a subset of the completer register set. Once a variable is mapped to a register, all it's other uses and definitions should also be mapped to the same register. In such a case, the register pressure on the subset increases greatly. There should not be many instructions that cause an increase in register pressure.

### 4.3 The Compressed Instruction Set

Taking into consideration the issues mentioned above we selected instructions to be included in CIS. The instructions ADD0T, ADD4T, ADDIT, LI, and ADDT were selected due to their high frequency of occurrence. The other operations like SUBFT, SLWT, CMPWT, and CMPWIT were included because they occurred in between the compressible instructions. Some of the instructions in the chosen instruction set consist of immediate operands. By eye-balling over the profiles we saw that we need at-least 8 bit of immediate field. More than that was not possible, as then we could not have coded the ADDIT instruction. Only a bit was left for the opcode field of ADDIT. Then instructions like CMPWIT and LIT left only 3 bit space for opcode... and so on. In some senses this CIS is quite a compact one, and supports 11 instructions using variable opcode lengths.

## 5 Retargetable Compiler

In the previous sections we have given an overview of how the Compressed Instruction Set impacts the compiler phases. In this section we concentrate on the EXPRESS compiler where we included the support of CIS for the Motorola PowerPC architecture.

### 5.1 Flow of the Compiler

The EXPRESS compiler[6] takes the input C-program as well as the EXPRESSION description of the target architecture and generates assembly instructions for the specified target. The

| Operation | Opcode | Destination | Source1 | Source2 | Immediate |
|---|---|---|---|---|---|
| ADDIT | 0 | 2 | 5 | | 8 |
| CMPWIT | 1-00 | | 5 | | 8 |
| LIT | 1-01 | | 5 | | 8 |
| SUBFT | 11-00 | 2 | 5 | 5 | |
| ADDT | 11-01 | 2 | 5 | 5 | |
| SLWT | 11-10 | 2 | 5 | 5 | |
| ADD0T | 1111-00 | 5 | 5 | | |
| ADD4T | 1111-01 | 5 | 5 | | |
| CMPWT | 1111-10 | | 5 | 5 | |
| BXT | 111111-0 | | | | |
| NOPT | 111111-1 | | | | |

**Table 1. Compressed Instruction Set**

EXPRESSION description captures the Instruction Set of the target architecture on which the generated assembly code will run. The C-program is parsed by a GCC front-end into an intermediate form containing generic instruction set for MIPS. All the target independent optimizations are performed on this intermediate form and then this form is converted by instruction selection into another intermediate representation based on target instruction set. All the target specific optimizations are done on this intermediate representation. The register allocation stage of the compiler extracts the information about register set from the EXPRESSION description and allocates registers using an improved version of Chaitin's algorithm. The parsing of EXPRESSION produces an intermediate file called *operandsMapping.txt* which contains the formats for the target assembly instructions. After register allocation, the code-generator uses this intermediate file to generate the target assembly code.

The EXPRESSION thus first converts the given application in generic Instruction Set, and then using the generic-to-target operations mapping converts the code in target Instruction Set. This conversion is done using a tree-based pattern matching mechanism, which is intrinsically local in nature. On the other hand, the decision whether to convert a group of compressible instructions into Compressed Instruction Set can be done only by doing a profitability analysis on the whole block. So the direct conversion of Generic Instruction Set to Target Instruction Set, including Compressed Instruction Set is not possible. So we augment the target instruction set to have a one to one mapping between Compressed Instruction Set and Augmented Instruction Set. By doing this the instruction selection can convert all the possible instructions into Compressed Instruction Set. Later a profitability analysis can be done as a compiler pass, which will actually map the Augmented Instructions to Compressed Instruction Set or back to Normal Instruction Set. Since the mapping of Compressed Instruction Set and Augmented Instructions is one to one, the conversion of instructions from Augmented Set to Compressed Instruction Set can be implemented as a simple compiler pass over the IR. Conceptually Register Allocation can be done anytime before or after this Instruction Selection. Thus we get the flow as described in Figure 8.

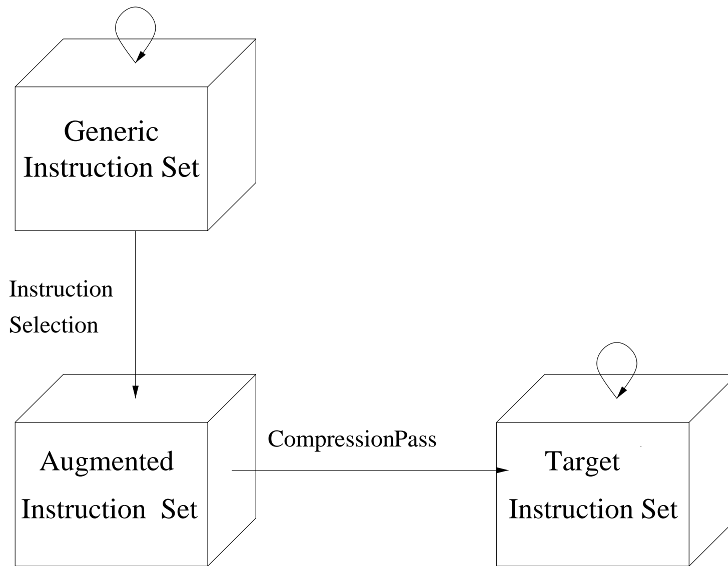


**Figure 8. EXPRESS flow for modeling CIS**

All the compiler passes are modeled in a generic way by fitting them well into the retargetable framework of EXPRESS.

We will now discuss each of the compiler phases that were modified for modeling the compressed instruction set.

## 5.2 Instruction Selection

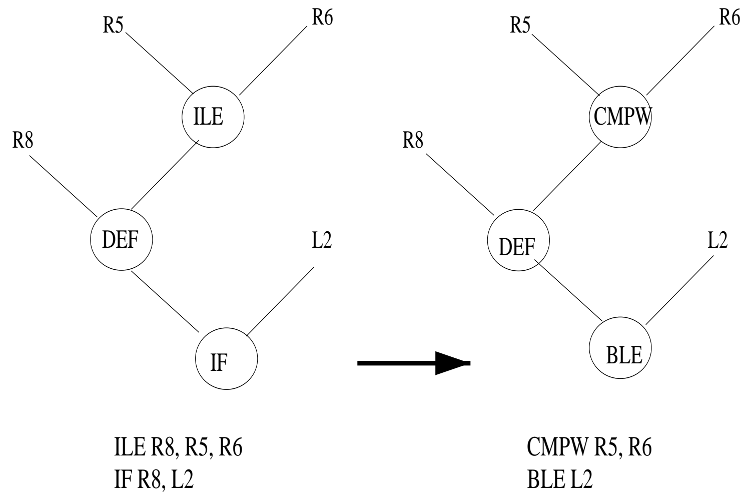


**Figure 9. An example: Iburg tree-pattern matching**

This phase converts all the generic opcodes into their target-specific forms. However, a one to one mapping doesn't always exist. Consider an example given in Figure 9.

The generic instruction *ILE* sets an explicit register **R8** if the value in **R5** is less than or equal to that in **R6**. The conditional opcode *IF* checks the register **R8** for branching to **L2**. However, the powerPC instruction set contains opcode *CMPW* which sets specific bits of an implicit register after comparing **R5** and **R6**. The *BLE* instruction checks the appropriate bit of the implicit register for the less-than-or-equal-to condition for branching to **L2**. Clearly, a one to one mapping between the generic opcodes *IF* and *BLE* does not exist. Consequently, such complex mapping of generic to target instructions can be tackled by mapping a tree of generic opcodes into a tree of target opcodes, the connectivity being established through a def-use chain.

The instruction selection uses an Iburg[2] tree-based pattern matching algorithm to convert the generic instructions into target instructions. ( Refer to Figure 9 ) Once the instructions are obtained in their target-specific form, based on a profitability function, a decision needs to be taken whether each of the compressible instructions to be converted into its compressed form. However, a global analysis of a block of instructions for profitability analysis is not possible while matching tree-based patterns. This calls for another pass for compressing the compressible instructions.

In order to make the compression pass simple in terms of mapping to the target instruction set, the mapping needs to be one to one. To ensure that, we augment the PowerPC instruction set with the full-length(32-bit) forms of the compressed instructions. The instruction selection will be just complex enough to convert all the generic instructions into the augmented instruction set comprising target instructions and full length compressed instructions. An injective mapping exists between the full-length compressed instructions and the actual compressed instructions. The compression pass of the compiler will transform an intermediate representation having augmented instructions into another intermediate representation comprising 32 bit target instructions and 16 bit compressed instructions.

### 5.3 Profitability Function

The Instruction Selection phase converts application in Generic Instructions Set into Augmented Instruction Set for the target machine greedily converting for Compressed Instructions. The decision whether to convert a block of Augmented instructions into Compressed Instructions is taken in the Compression pass made by the compiler. The decision to convert a group of compressible instructions depends on

- **Impact on Performance :**
  - When an instruction is converted to Compressed Instruction the register pressure may increase and cause spilling. The profitability function for the conversion should have some measure of register pressure during this part of code.
  - Conversely now since we have more instructions in a cache line, the cache miss penalty is avoided which could result in a improvement in the performance.
- **Impact on Code Size :**
  - Before each block of Compressed Instructions, there has to be a Mode Change *BX* instruction.

- After the end of each block of Compressed Instructions, there has to be a Mode Change *BXT* instruction.
- Before the end of a block of Compressed Instructions, there could be a *NOPT* instruction to make the number of instructions in the block even.

Currently in our implementation we have assumed a very simple cost function based on code size improvements only. If we are having a code size improvement by converting a block into Compressed Instructions, we convert that. This has been done to maximize the code size improvements.

### 5.4 Compression Pass

Compression Pass converts the Augmented Instruction Set into final Target Instruction Set which includes the Compressed Instruction Set. Since there is a unique Compressed Instruction and Target Instruction for each Augmented Instruction, this conversion can be done "out-of" Instruction Selection pass. Also it is difficult to compress the branch instructions because they include a immediate value for jump in Motorola PowerPC architecture. This means that almost all basic blocks enter and exit in normal mode. This simplifies the code for Compression Pass. The code for Compression Pass is included in the Appendix.

### 5.5 Register Allocation

There is a restricted set of registers meant to be used by compressed instructions as well as other target instructions. The destination register for some of the compressed instructions can only use this restricted set. We identified registers *$24*, *$25*, *$26* and *$27* to form the restricted set. In order to ensure that the destination registers for the selected compressed instruction always maps to the restricted set, we came up with the following strategy. The register set for register allocation is extended to include 32 virtual registers called *temp0* .. *temp31*. In the compression pass, along with the mapping of an augmented target opcode into a compressed target opcode, we change the destination register of the instruction to a new *temp0* register, if the compressed target instruction format allows only restricted register set use. This requires all the uses of the old definition of the compressed instruction to be replaced by the new definition.

One way to resolve this problem might be to traverse the def-use chain of the old definition and replace them by the new definition. However, it doesn't work this way. Let's consider the control flow graph given in Figure 10.

If *$20* in basic block **BB2** is replaced by *$temp0*, all the uses of *$20* is replaced by *$temp0* which changes all *$20* in **BB2** to *$temp0* because of the loop-structure. However, *$20* in instruction number 3 has two definitions : one coming from basic block **BB1** and another from basic block **BB2**. So, the problem doesn't get solved completely by just replacing all the uses of a definition by traversing the def-use chain. In order to solve this problem, we further need to traverse the use-def chain of all the uses of the old definition and replace that by the new definition. The problem is solved completely by recursively changing the def-use and use-def chains starting from an old definition and replace all of them by the new definition.

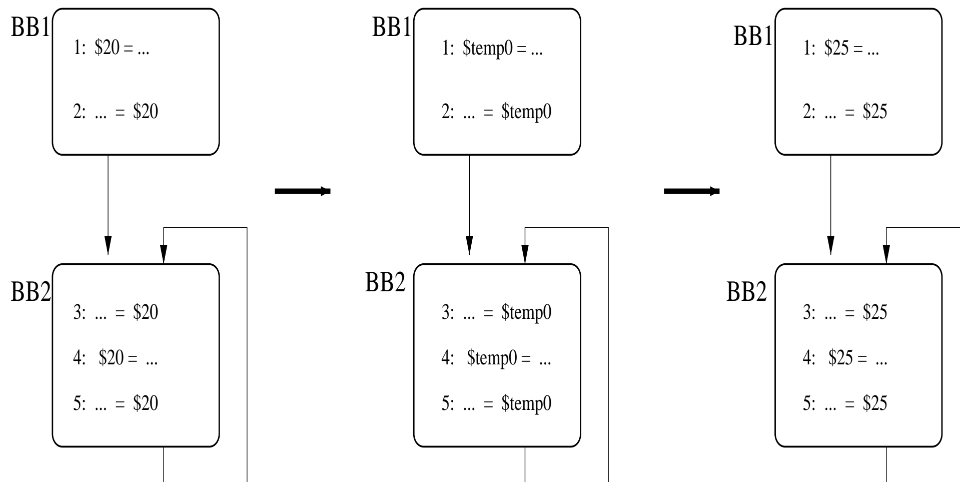


**Figure 10. An example: Control flow graph**

One approach to solve this problem would have been to generate a move instruction for every compressed instruction that moves the value from *$temp0* to *$20* and the correctness of the resulting code is automatically maintained. This is obviously not a good solution because we will have as many move instructions as there are compressed instructions and the main purpose of using the compressed instruction gets defeated.

Another approach would have been to introduce a move instruction at the end of the basic block **BB1** that will move the value from *$20* to *$temp0*. Again, since the same goal can be achieved without introducing a move instruction, there is no reason why we should affect the performance by having the costly move operation.

Once the mapping to the temporary registers is successfully done, the next step is to convert all the temporaries into the restricted register set. This is how we accomplished the mapping of the destination registers of some of the compressed instructions into a well-known restricted register set. As illustrated in Figure 10, *$20* finally gets mapped to *$25*.

# 6 Results

We developed support for Compressed Instruction Set Architecture in EXPRESSION, and tested it by designing a CIS for Motorola PowerPC architecture. We used ten Livermore Loops as our benchmarks.

## 6.1 Code Size Improvements

Figure 11 shows the code size improvement that we achieved on the Livermore loops.

Figure 11 shows the code sizes of Livermore loops before and after using CIS. The orignal code size is scaled to 100. The Livermore Loops are sorted according to increasing code size. The code sizes have reduced by upto 20% of their original size using our CIS, and the EXPRESSION Compression Pass optimization. It might be worth noting that the code size improvements are

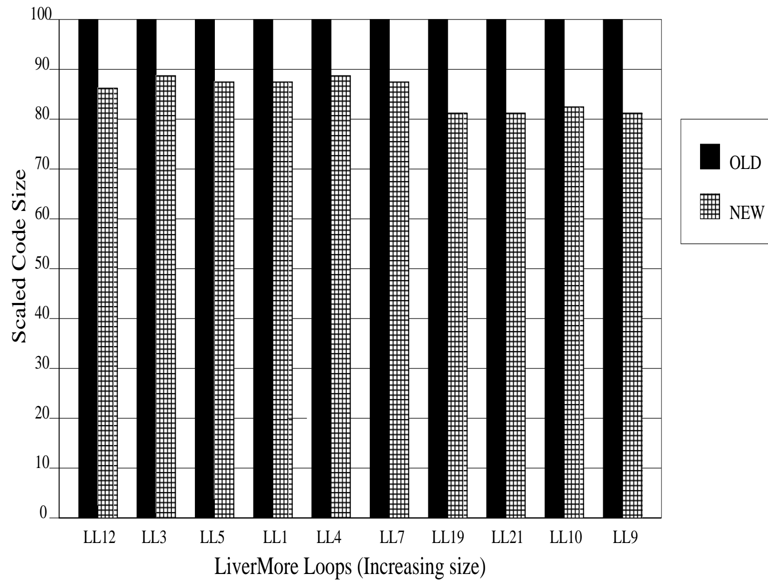


**Figure 11. The Pipeline of ARM architecture**

better for larger code sizes. This could be explained by the observation that in small examples the program initialization and program ending contribute significantly, and these parts generally contains loads and stores to immediate values, which are large and cannot be compressed in the CIS. The observation that for large benchmarks of numerical codes the improvement of about 17% is consistent is a proof for this.

### 6.2 Impact on Performance

The secondary aim of code size reduction is to not to have a negative impact on performance. Now-a-days when memory is a bottleneck rather than the processor speed, smaller codes can lead to better performance. A rough estimate says

- L1 level cache miss penalty 5 cycles.
- L2 level cache miss penalty 100 cycles.

And when we are using VLIW machines, this penalty increases even more, i.e. by a factor of parallelism. Saving even a few cache misses could more than compensate. Due to decrease in code size we can expect better cache performance.

In this project we have not modeled the instruction cache and memory subsystem. So, the numbers we get in Figure 12 do not include the advantage of the cache. These can be considered worst case performance numbers.

Figure 12 compares the cycle counts of a few Livermore Loops. The orignal cycle count is scaled to 100. The Livermore Loops are sorted in order of increasing cycles.

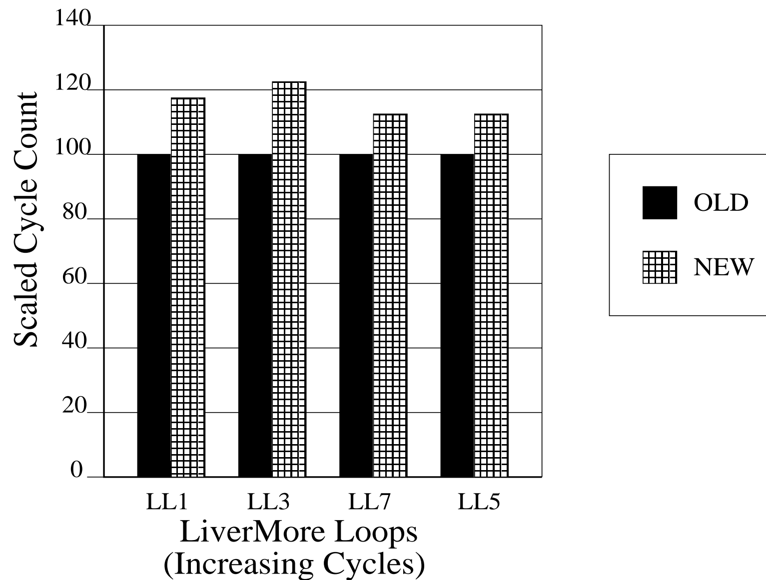


**Figure 12. The Pipeline of ARM architecture**

## 7 Future Work

We have implemented the compiler support for Compressed Instruction Support in the EXPRESS compiler to support quite a variety of flavors in Compressed Instruction Set Architecture. The extensions of this work are

- To build support for Compressed Instruction Set in the Simulator.
- We have to make way for automatic generation of Compiler from EXPRESSION description of Compressed Instruction Set.
- We have to decide upon the changes in the grammar of EXPRESSION to include information about Compressed Instruction Set Architecture.
- Complete the path from EXPRESSION to the data structures used in the Compiler for support of CIS.
- After all this has been done this architectural feature exploration can be tested on various benchmarks over different architectures and different CIS's.

# A Compression Pass

```
//////////////////////////////////////////////////////////
// File:                          ThumbPass.cpp
// Created:                       March 15, 2000.
// Last modified:                 March 15, 2000.
// Author:                        Aviral Shrivastava/Partha Biswas
// Email:                         { aviral, partha } @ics.uci.edu
//
//////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Routine.h"
#include "ControlFlowGraph.h"
#include "BasicBlockNode.h"
#include "ThumbPass.h"
#include "MyString.h"
#include "BBIterator.h"
#include "NormalInstr.h"
#include "ComputeOp.h"
#include "IburgToIR.h"
#include "SemActPrimitives.h"


#include <map>

#ifdef map
#undef map
#endif
#define map std::map

//START: MFC DEBUG NEW: THIS MUST BE AFTER ALL THE #INCLUDES IN A FILE
#ifdef WIN32
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#endif

extern char *prefix(char *s);
```

```
extern char *sufix(char *s);

//END MFC DEBUG NEW:

/*
Notes about the current implementation

        1. Thumb is the technique of recoding instructions into 16 bits,
        so that the code size is reduced.

        2. We do not Thumbify the goto or branch instructions, so the
        - At the entry of a basic block the ThumbMode == 0.
        - At the exit od a basic block the ThumbMode == 1.

        3. Inside the basic block, I collect a list of instructions
        that can be converted into thumb, and then find the size of
        the collection.
        If it seems profitable, I convert the entire collection of
        instructions into thumb instructions. Currently a block of
        4 or more instructions is considered to be profitable to be
        converted into thumb mode.

        4. I need to instructions that can be converted into thumb mode
        into thumb mode, so I need a mapping of target instructions and
        the corresponding thumb instruction.
        We assume we have them in a table right now. Later we need to
        read the expression description and dump this information in a
        file and later read that file into this table


*/

#define TPF 4

int ThumbInit();
NormalInstr *createThumbInstr(MyString opcode);
extern void appendInstrToBB(NormalInstr *newInstr, BasicBlockNode *bb);
extern void prependInstrToBB(NormalInstr *newInstr, BasicBlockNode *bb);
extern void insertInstrBetween(NormalInstr *newInstr, NormalInstr *prevInstr
              NormalInstr *nextInstr);
```

```
NormalToThumbInstrMap NtoTMap;
extern Routine *crtRoutine;

void Routine::ThumbPass()
{

        //make sure program mappings are up to date
        extern void buildProgramMappings();
        buildProgramMappings();

        if (ThumbInit() == 0)
        {
                printf("Could not build the Thumb to Normal Instruction
                         Mapping quitting...\n");
                exit(0);
        }

        crtRoutine = this;

        // print the thumb mapping
        NtoTIter i;
        for ( i = NtoTMap.begin(); i!= NtoTMap.end(); i++ )
        {
                printf("printing Normal to Thumb Instruction Mapping\n");
                printf("%s \t %s\n", ((MyString) (*i).first).getStr(),
                    ((MyString) (*i).second).getStr());
        }

        LinkedListIterator<BasicBlockNode> *iter = _cfg.createBBIterForw();

        while (iter->hasMoreElements())
        {
                (iter->currentElement()).ThumbPass();
                iter->nextElement();
        }
        delete iter;
}


void BasicBlockNode::ThumbPass()
{
        int ThumbMode = 0;
```

```
BaseInstruction *firstThumbInstr = NULL;
BaseInstruction *lastThumbInstr = NULL;
int ThumbCount = 0;


// iterate over the list of basic blocks.
BBForwIterator bbIter(this);

while (bbIter.hasMoreElements())
{
        if ((ThumbMode == 0)
           &&(bbIter.currentElement()->CanbeThumbed()))
        {
                // reset the variables
                ThumbMode = 1;
                ThumbCount = bbIter.currentElement()->OperationCount
                firstThumbInstr = bbIter.currentElement();
        }

        else if ((ThumbMode == 0)
           &&(!bbIter.currentElement()->CanbeThumbed()))
        {
                // do nothing
        }

        else if ((ThumbMode == 1)
           &&(bbIter.currentElement()->CanbeThumbed()))
        {
                // increment the ThumbCount
                ThumbCount +=
                          bbIter.currentElement()->OperationCount();
                lastThumbInstr = bbIter.currentElement();
        }

        else if ((ThumbMode == 1)
           &&(!bbIter.currentElement()->CanbeThumbed()))
        {
               ConvertInstrToThumb(firstThumbInstr,
                          lastThumbInstr, ThumbCount);

                ThumbMode = 0;
                ThumbCount = 0;
```

```
				firstThumbInstr = NULL;
				lastThumbInstr = NULL;
			}

			bbIter.nextElement();
		}

		// We've reached the end of the BB. Check if ThumbMode == 1.
		//
		// If yes, need to do profitability analysis, and if necessary, conv
		// from firstThumbInstr to trailer of bb
		if (ThumbMode == 1)
		{
			ConvertInstrToThumb(firstThumbInstr, lastThumbInstr,
									ThumbCount);
		}
}



void BasicBlockNode::ConvertInstrToThumb(BaseInstruction *firstThumbInstr,
				BaseInstruction *lastThumbInstr, int ThumbCount)
{
		if (ThumbCount >= TPF)
		{
			BaseInstruction *prevInstr, *nextInstr;

			prevInstr = nextInstr = NULL;

			BBForwIterator bbIter(this);

			while(bbIter.currentElement() != firstThumbInstr)
			{
				prevInstr = bbIter.currentElement();
				bbIter.nextElement();
			}

			while(bbIter.currentElement() != lastThumbInstr)
			{
				bbIter.currentElement()->ConvertToThumb();
				bbIter.nextElement();
			}
```

```
bbIter.currentElement()->ConvertToThumb();

// If the lastThumbInstr is not the trailer of this BB then,
// make nextInstr the next element of the bbIter
// (its current element before this statement
// is lastThumbInstr)
//
if (lastThumbInstr != this->_trailer)
{
        bbIter.nextElement();
        nextInstr = bbIter.currentElement();
}


// add a BX instruction before the first Thumb instruction
MyString opcode = "BX";
NormalInstr *BXInstr = createThumbInstr(opcode);
if (firstThumbInstr == this->getFirstInstr())
{
        prependInstrToBB(BXInstr, this);
}
else
{
        // Need to insert BXInstr between prevInstr
        // and firstThumbInstr
        //
        insertInstrBetween((NormalInstr *)BXInstr,
            (NormalInstr *)prevInstr,
            (NormalInstr *)firstThumbInstr);
}

// add the NOPT instruction
if (ThumbCount%2 == 0)
{
        opcode = "NOPT";
        NormalInstr *NOPTInstr = createThumbInstr(opcode);
        if (lastThumbInstr == this->getLastInstr())
        {
                appendInstrToBB(NOPTInstr, this);
        }
        else
```

```
                        {
                                insertInstrBetween((NormalInstr *)NOPTInstr,
                                        (NormalInstr *)lastThumbInstr,
                                        (NormalInstr *)nextInstr);
                        }
                        lastThumbInstr = NOPTInstr;
                }


                // add a BXT instruction  at the last
                opcode = "BXT";
                NormalInstr *BXTInstr = createThumbInstr(opcode);
                if (lastThumbInstr == this->getLastInstr())
                {
                        appendInstrToBB(BXTInstr, this);
                }
                else
                {
                        insertInstrBetween((NormalInstr *)BXTInstr,
                                        (NormalInstr *)lastThumbInstr,
                                        (NormalInstr *)nextInstr);
                }

        }
}

// Is a misnomer.
// This function creates and returns a BX operation, or a
// BXT operation, or a NOPT operation.
NormalInstr *createThumbInstr(MyString opcode)
{
        OpCode *opc;

        ComputeOp *BXOp= new ComputeOp();
        opc = new OpCode(opcode.getStr());
        BXOp->initOpCode(*opc);
        BXOp->setID(0);

        NormalInstr *BXInstr = new NormalInstr();
        BXInstr->initScheduleAndAdd(BXOp, true);
        BXInstr->setID(0);
        return BXInstr;
```

```
}


void NormalInstr::ConvertToThumb()
{
        ArrayIterator<DataOpSlot> *dIter = getDataOpSlotIter();

        while (dIter->hasMoreElements())
        {
                if ((dIter->currentElement()).hasOper())
                {
                  ((dIter->currentElement()).\
                         getPtrToOperation())->ConvertToThumb();
                }
                dIter->nextElement();
        }
        delete dIter;
}


int NormalInstr::CanbeThumbed()
{
        ArrayIterator<DataOpSlot> *dIter = getDataOpSlotIter();
        ArrayIterator<ControlOpSlot> *cIter = getControlOpSlotIter();
        ArrayIterator<FlowOpSlot> *fIter = getFlowOpSlotIter();

        MyString goto_opcode = "GOTO";

        // There should be no FlowOps
        while (fIter->hasMoreElements())
        {
                if ((fIter->currentElement()).hasOper())
                {
                        MyString opcode;
                        ((fIter->currentElement()).\
                                 getPtrToOperation())->getOpcodeName(opcode)

                        if (opcode == goto_opcode)
                        {
                                delete fIter;
                                return 0;
                        }
```

```
                }
                fIter->nextElement();
        }
        delete fIter;

        // There should be no ControlOps
        while (cIter->hasMoreElements())
        {
                if ((cIter->currentElement()).hasOper())
                {
                        if (((cIter->currentElement()).\
                            getPtrToOperation())->CanbeThumbed() == 0)
                        {
                                delete cIter;
                                return 0;
                        }
                }
                cIter->nextElement();
        }
        delete cIter;

        while (dIter->hasMoreElements())
        {
                if ((dIter->currentElement()).hasOper())
                {
                        if (((dIter->currentElement()).\
                             getPtrToOperation())->CanbeThumbed() == 0)
                        {
                                delete dIter;
                                return 0;
                        }
                }
                dIter->nextElement();
        }
        delete dIter;

        return 1;
}


int NormalOp::CanbeThumbed()
{
```

```
        NtoTIter i;
        MyString opcode;
        getOpcodeName(opcode);

        for ( i = NtoTMap.begin(); i!= NtoTMap.end(); i++ )
        {
                if ((*i).first == opcode)
                        return 1;
        }
        return 0;
}


void NormalOp::ConvertToThumb()
{
        MyString opcode;
        getOpcodeName(opcode);

        // check if the opcode is one of
        // ADDIA, SUBF, ADD, SLW
        if( opcode == "ADDIA"
                || opcode == "SUBF"
                || opcode == "ADD"
                || opcode == "SLW"
                )
        {
                // the destination registers use the thumb register set
                BaseArgument *destArg =
                      this->ptrToOperand(_DEST_, _DEST_LIST_);
                BaseArgProperty *prop =
                      destArg->getPtrToProperty(_DUCHAIN_);
                int id = destArg->getUniqID();
                // generate a new temporary
                char *tempName = NEW_TEMP();
                char *pref = prefix(tempName);
                if( destArg->isRealRegArg() ) // will always be true
                {
                        ((RealRegArg*)destArg)\
                                 ->changeReg(pref,atoi(sufix(tempName)),
                                 globalRegFileList->getIndex(pref));
                        //add SSA property with value 1 to reg (temp registe
```

```
BaseArgProperty *ssaP=
          destArg->getPtrToProperty(_SSA_);
if( ssaP == NULL )
{
        SSAArgProperty *ssaP=new SSAArgProperty();
        ssaP->setSSANum(1);
        // add the SSA property
        ((RealRegArg*)destArg)->addProperty(*ssaP);
}

// add the _DUCHAIN_ property
// ((RealRegArg*)destArg)->addProperty(*prop);
// update the DUCHAIN of this dest
MyLinkedListIterator<BaseOperation*>* iter1 =
     ((DUChainProperty*)prop)->useListIteratorForw()
while ( iter1->hasMoreElements() )
{
        BaseOperation *opInDUChain =
                         iter1->currentElement();
        ArgList & al =
                 opInDUChain->sourceOperandList();

        ArgListIter i1;
        for (i1 = al.begin(); i1 != al.end(); i1++)
        {
             BaseArgument *src = (*i1);
             if( src != null )
             {
                 if( src->isRealRegArg() )
                 {
                     if( id == src->getUniqID() )
                     {

                          ((RealRegArg*)src)->\
               changeReg(pref,atoi(sufix(tempName)),
               globalRegFileList->getIndex(pref));
                          BaseArgProperty *udProp =
               src->getPtrToProperty(_UDCHAIN_);
                          MyLinkedListIterator\
                          <BaseOperation*>* iter2 =
               ((UDChainProperty*)udProp)->\
               defListIteratorForw();
```

```
                                                        while( iter2->\
                                                               hasMoreElements() )
                                                        {
                                                            BaseOperation
                                            *opInUDChain = iter2->currentElement();
                                                            if (opInUDChain != thi
                                                            {
                                                             BaseArgument *defReg=
                                     opInUDChain->ptrToOperand(_DEST_,_DEST_LIST_);
                                                if( defReg->isRealRegArg() )
                                                {
                                                   ((RealRegArg*)defReg)->\
                                       changeReg(pref,atoi(sufix(tempName)),
                                       globalRegFileList->getIndex(pref));
                                                }
                                            }
                                            iter2->nextElement();
                                        }
                                     }
                                 }
                              }
                           }

                           iter1->nextElement();
                        }

                        // propagate this change to the global symbol table
                        globalSymbolTable->addNewRegs(((RealRegArg*)destArg),1);
                   }
              }

              changeOpCode(NtoTMap[opcode].getStr());
      }


      int NormalInstr::OperationCount()
      {
              ArrayIterator<DataOpSlot> *dIter = getDataOpSlotIter();
              ArrayIterator<ControlOpSlot> *cIter = getControlOpSlotIter();
              int op_count = 0;

              while (cIter->hasMoreElements())
```

```
        {
                if ((cIter->currentElement()).hasOper())
                        op_count++;

                cIter->nextElement();
        }
        delete cIter;


        while (dIter->hasMoreElements())
        {
                if ((dIter->currentElement()).hasOper())
                        op_count++;

                dIter->nextElement();
        }
        delete dIter;

        return op_count;
}


// returns 1, if this could initialize the NtoTMap
int ThumbInit()
{
        MyString NormalInstr = "ADDIA";
        MyString ThumbInstr = "ADDIT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "ADD0A";
        ThumbInstr = "ADD0T";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "ADD4A";
        ThumbInstr = "ADD4T";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "CMPWIA";
        ThumbInstr = "CMPWIT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "CMPW";
```

```
        ThumbInstr = "CMPWT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "LIA";
        ThumbInstr = "LIT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "SUBF";
        ThumbInstr = "SUBFT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "SLW";
        ThumbInstr = "SLWT";
        NtoTMap[NormalInstr] = ThumbInstr;

        NormalInstr = "ADD";
        ThumbInstr = "ADDT";
        NtoTMap[NormalInstr] = ThumbInstr;

        return 1;
}
```

# References


[1] ARM. *http://www.arm.com*.

[2] D. R. H. Christopher W. Fraser and T. A. Proebsting. Engineering a simple, efficient code generator generator. 1992.

[3] V. Z. et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[4] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for system level design exploration. Technical Report TR 98-29, University Of California, Irvine, 1998.

[5] A. Halambi, N. Dutt, and A. Nicolau. Customizing software toolkits for embedded systems-on-chip. 2000.

[6] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach.* Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.

[8] C. N. F. Mahadevan Ganapathi and J. L. Hennessy. Retargetable compiler code generator. 1982.

[9] Semiconductor Industry Association. *National technology roadmap for semiconductors: Technology needs*, 1998.

[10] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*, 1998.