Adding custom instructions to the Simplescalar infrastructure

Somasundaram Meiyappan
HT023601A
School of Computing
National University of Singapore
meisoms@ieee.org

ABSTRACT

Synthesizable processors like the MIPS' Pro[2] series cores provide SoC[1] architects the ability to add user-defined instructions to the MIPS processor. These specialised instructions are normally application specific. These custom instructions can be introduced in the program as assembly instructions manually or using a high-level compiler that chooses this instruction automatically based on the dataflow. This project aims to provide insights into the popular open-source compiler, GNU gcc, in this context. It also implements a simple custom instruction in GCC and also in the Simplescalar infrastructure. This allows us to examine the effect of adding a custom instruction to a processor using a simulator.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Code generation; C.0 [**General**]: Modeling of computer architecture

Keywords

custom instruction,extensible processor[2]

1. INTRODUCTION

Embedded systems are diverse in nature and almost all of them have a limited number of embedded processors and micro-controllers to choose from. Some of the most widely used 32/64-bit embedded processor architectures include MIPS, ARM and ColdFire. Other processors like PowerPC and Pentium are also used in some embedded applications. These processors are actually general purpose processors themselves. However, these processors provide better power-performance and cost-performance ratios than other general purpose processors and hence are very popular in the embedded systems. Recently, processors like Xtensa from Tensilica[4] and ARCTanget from ARC[1] allow system architects to add custom instruction for better performance. Among the processors described above, MIPS provides limited support a custom instruction unit and register files through its *CorExtend* technology.

Companies like Tensilica provide users with almost the complete toolsuite that is tailored for an Xtensa processor that is configured by the user. But this process might not be straight forward for the MIPS Pro series of processors. When a new instruction is added, the designer must either work with his compiler provider or use assembly instructions in the code. Another alternative to this is to change an open-source retargettable compiler like GCC.

[1]System-on-Chip

[2]An extensible processor allows the designer to extend the basic instruction set through the addition of custom application-specific instructions

2. PROBLEM SPACE

There are two kinds of issues to resolve when we deal with extensible processors. One class of them is about identifying opportunities that provide efficient extension and the other class is about implementing the extensions.

2.1 Instruction Synthesis

This is the part of the problem in which the program is given to a black-box program to get the custom instructions to implement. These instructions are normally combinations of some simple instructions that improves the performance of the system when implemented as a single instruction. The black-box will consist of

1. Constraint based pattern selection from *Data Flow Graph*(DFG)
2. Benefit/Cost analysis for each pattern in hardware Vs software
3. Instruction encoding for the chosen instructions

In this project, we do not try to tackle the problem. This active area of research has contributed many publications.

2.2 Instruction Implementation

The instructions are implemented in the hardware by SoC designers. However, the RTL design may be emited by the

black-box program that is discussed in the previous subsection. Similarly, the compiler tool chain may also be emitted by the same program. Tensilica[4] automates both the hardware implementation and the tool chain implementation (to generate its XCC compiler). In this project, we explore the suitability of GCC for this process. Please note that Tensilica already provides GCC based compilers for its Xtensa V processors.

3. GNU GCC

The GNU C Compiler suite is one of the most popular compilers in the world. It is dominant in MacOS and Linux desktops and workstations. It supports a wide variety of processors. GCC is a retargetable compiler suite with front-ends for atleast 4 languages - C, C++, Java and Fortran. Just like any other retargetable compiler, GCC has a decoupled front-end and back-end. The front-end communicates with the back-end through an intermediate representation (IR). The language front-end outputs the parsed data in an IR called `Gimple`. High-level optimisations are done on the Gimple representation. It then converts it into another intermediate representation for the back-end to work on. The back-end of the compiler is almost language independent. The backend of the compiler has many passes like

- Common Sub-expression elimination
- Loop invariant code motion
- Data flow analysis
- *Combine instruction patterns*
- Modulo scheduling
- Local and global register allocation
- Instruction scheduling

GCC uses an intermediate representation (IR) called RTL. RTL is a lisp-like text based language. GCC builds trees and lists of all instructions in the program at runtime. Macros are used to access the various attributes of an instruction in that list or graph. RTL is also used by GCC as the backbone for achieving retargetability. Target specific customisations are present in the *gcc/config* directory in GCC's source directory. For example, ARM targets have their own *gcc/config/arm* directory. The three most important files there are `arm.h`, `arm.c` and `arm.md`.

The file `arm.md` defines all instruction patterns that GCC needs and uses. There are two kinds of instruction patterns defined in the `md` (machine description) file *viz.,* named patterns[3] and un-named patterns[4]. When the RTL output is generated by the compiler after performing source-level optimisations, it needs the machine description file to define certain named patterns. IR *i.e.,* the RTL representation is generated only using these named patterns and operators. These named patterns that are found in the given user program are later combined into larger patterns by the *Combine instruction patterns* phase.

[3]Named patterns are patterns which must be implemented for GCC to work. If a named pattern corresponds to more than one isntruction, then we need to define the pattern accoridngly.

[4]These un-named patterns can actually have names; but names are used for identification/debugging only.

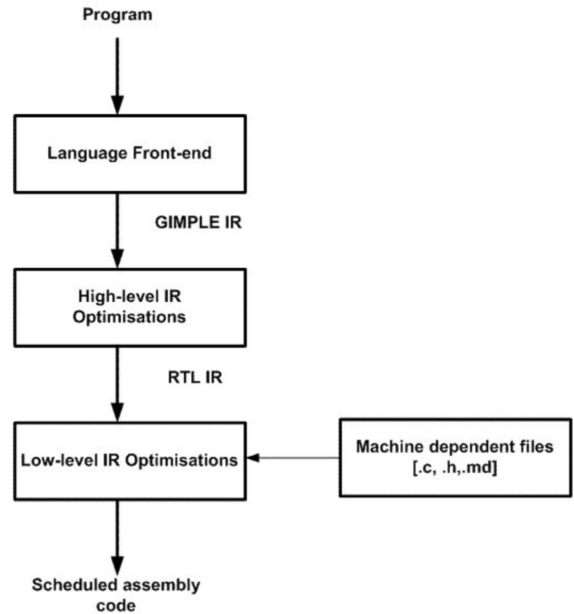

Figure 1: GCC compiler flow

A small example of an instruction/pattern represented by RTL (in the text form) is given below.

```
(insn 48 47 50 (set (reg/v:SI 36)
      (mult:SI (reg:SI 42)
      (reg:SI 41))) -1 (nil)
            (nil))
```

3.1 General flow of GCC

GCC compiles every procedure in a file and optimises it as in figure 1. After every function is optimised, a global optimisation pass (not all optimisations are done again) is done on the entire source file. The assembly file that is produced by the compiler is then assembled with the GNU Assembler (GAS). After all files are assembled into object files, the various object files are linked using the linker. The key takeaway is that all optimisations are done at a procedure level.

3.2 GCC Combiner phase

In the context of this project, the part of the compilation that we are interested in is the *instruction combination* phase. The reason being that it is the phase of the compilation in which complex instruction patterns are indentified (or inferred) in the given program. These complex instruction patterns are defined by the user in the machine description file. GCC's pattern combiner compares a pattern from the program with the list of all patterns defined by the user. The first pattern from that ordered list to match the sub-graph from the DFG and the constraints on the operand will be chosen. Once a pattern is chosen, the

Table 1: Patterns assumed in the illustration

mulsi3	Represents c=a*b
addsi3	Represents c=a+b
mulsi3addsi	Represents d=a*b+c
mulsi4addsi	Represents e=a*b+c*d

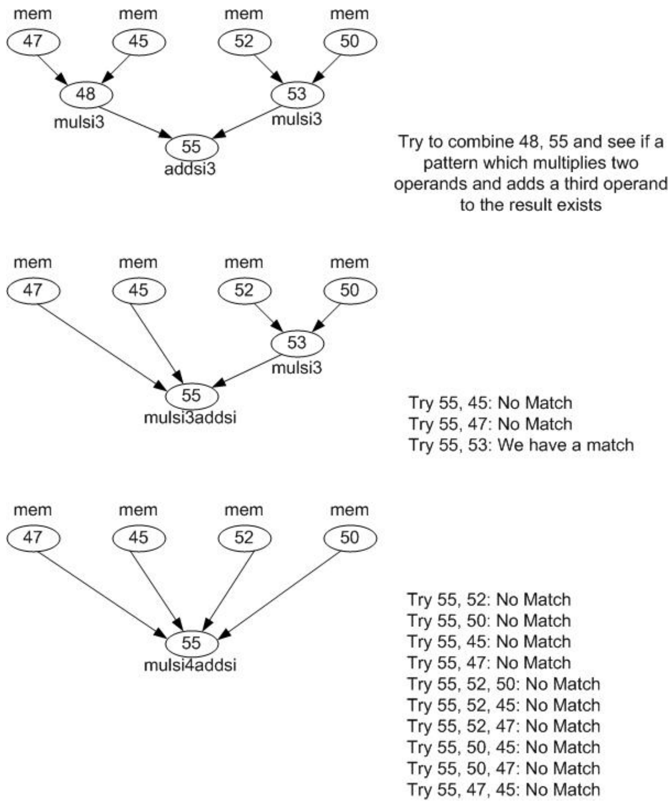

Figure 2: Illustration of GCC combiner on a DFG

sub-graph from the program is replaced with this single pattern. The patterns(`insn` in RTL) that are replaced by this complex pattern are annotated as deleted. Once deleted, these instructions are not considered for any further merges with other instructions.

An illustration of the combiner phase is provided in figure 2. We shall assume that there are 4 patterns, as described in table 1, defined in the `arm.md` file.

GCC's current combiner can try to combine a maximum of 3 instruction patterns together. Please note that these instruction patterns that it can combine today can have only one outgoing edge from the DFG sub-graph. In some architectures, condition registers are updated on an arithmetic operation. In such cases, if another instruction depends on the condition codes, then GCC will make sure that the new pattern also produces the condition register output.

There are two other issues with GCC's combiner that could limit its ability to effectively combine instruction patterns. Since the instructions that match to form a sub-graph are deleted by GCC, it might lose the ability to produce a better match if the selected pattern is not a subset of a possibly beneficial pattern. But this is not a big concern as the current implementation already limits the number of outputs from the combined pattern to one only.

The other issue is the placement of the combiner phase in the flow. The placement is not exactly an issue; but a limitation. When the combiner looks at a sub-graph for combining and matching, it chooses a sub-graph such that it has only one output and its inputs are memory references. Unfortunately, depending upon the way that an expression is written, GCC might insert a memory operation that stores the result. Bringing the patterns mentioned in table 1 into context once again, let us consider two expressions which produce the same result.

ExprA: `f1=a*b+c*d+(e<<4);`
ExprB: `f2=a*b+(e<<4)+c*d;`

From the RTL that is generated for these two C expressions, we can see that GCC will compute these expressions as

ExprA:
`f1=addsi3(mulsi4addsi(a,b,c,d),lshiftsi(e,4));`
ExprB:
`f2=mulsi3addsi(c,d,mulsi3addsi(a,b,lshiftsi(e,4));`

This particular problem has to be carefully looked into. In some architectures, in order to reduce the register pressure, memory references may be made after computing part of the large expression. This might cause the combiner to not pick some portion of the subgraph and thus results in poor optimisation. This also means that the combiner has to look beyond memory references too.

4. GNU ASSEMBLER

The GNU assembler is delivered as a part of the `binutils` package. That package also carried the linker for the target. When we synthesize a new instruction, the very minimal thing that needs to be done is to let the assembler know about the existence of the new instruction, otherwise, it is quite messy to incorporate the new intruction into the program's binary. Of course, another reason why GAS (GNU Assembler) should be updated is because the compiler only outputs assembly code. When a pattern that has the new instruction is chosen by the compiler, the mneumonic for the instruction is placed in the assembly stream. If GAS is not updated, then the compilation is going to fail.

The target dependent assembly file in GAS is located in the `binutils-<version>/gas/config/tc-target.c` file and the corresponding header file. Proper changes need to be made in this file to retarget the assembler when some new instructions are added to the `target`.

5. SIMPLESCALAR

Simplescalar[3] is a widely used high level processor model or simulator. It is written in C. Many researchers have developed models of different processors in SimpleScalar. It can provide a range of different types of simulation from a very fast simulation not taking into account the effect of micro-architectural features to a full processor simulation

with out-of-order execution and memory hierarchy. Simplescalar provides a good mix of traditional instruction set simulator, micro-architectural simulator and a basic C library for the simulator to perform I/O operations like reading from files(s) form the hard-disk, etc.... It is a good tool for profiling an embedded program and any application-level program for that matter. The number of papers that get published referring to Simplescalar is a testimony to its popularity among researchers.

When we are dealing with extensible processors, instruction set simulators become very important than usual as processor with the newly formed custom instruction is not going to be available. An FPGA based verification/emulation may also be helpful. But a simulator is a very convenient and quick way to verify our changes to the extensible processor.

Simplescalar is a simulator that is easy to retarget to any processor architecture. It also allows us to add new instructions to the simulator easily. Simeplescalar picks up the target specific instruction set and decoding information from the target definition file, `machine.def`. Most custom instructions can be just added here to make the simulator capable of interpreting the new instruction when it is encountered in the program and executing it.

5.1 Instruction Decoding

Any basic instruction set simulator (ISS for short) has two functions *viz.,* instruction decoding and instruction execution. In the real processors, architects employ different mechanisms for instruction fetch and decoding from a simple dedicated pipeline stage to pre-decoding and also to a fetch and decode stage is decoupled from the main pipeline.

In Simplescalar, instruction decoding is done as a simple chain of decoders. For the ARM architecture, Simplescalar first interprets the bits 24 to 27 to determine the class of an instruction. Each of the various classes of instructions have their own chain of decoders *i.e.,* the `load/store` instructions could then look into some other set of bits that is different from the bits that the `multiply` class of instructions check. Of course, one of the many leafs in the decode tree will determine which operation to perform.

When a new class of instructions, say `CFU(Custom Functional Unit)` class of instructions, a chain of decoders have to be created in the `machine.def` file so that the new custom instructions can be decoded and executed.

5.2 Input/Output dependencies

One of the main reasons why we design custom instructions is to perform a complex task using a single instruction. This instruction is typically a combination of many smaller operations. Thus, the new instructions may have more input operands than other class of instructions have.

Simplescalar simulators written for ARM have 4 input operands (inclusive of the predicate register). However, if a new predicated instruction takes in 4 input operands, then the simulator has to be extended to take care of the extra input operand. But this is a one time change to the simulator and newer instructions that uses the same or lesser number of input operands can reuse this infrastructure.

Macros (`#define`) form the leaf of the decode tree and it takes various kinds of arguments from the instruction name to the input and output dependencies. Also, each of the top-level simulator C source (files start with the prefix `sim`) would also have to be modified to achieve this capability.

5.3 Other issues

There are two other issues that also have to be dealt with for accurate simulations.

Accurate simulators in Simplescalar like `sim-outorder` need to know which functional unit or resources an instruction needs. This could be like integer multiply unit, integer ALU unit, etc ... Similarly, we might have to create a new resource called custom functional unit so that it can be simulated accurately.

The other issue is register files. There are three kinds of custom instructions that can be designed by an architect. They are

1. Instructions that uses only the general purpose register file
2. Instructions that uses the general purpose register file and a custom register file
3. Instructions that uses only the custom register file

Wherever applicable, this needs to be taken care of too. While it is very easy to implement the first case as all Simplescalar simulators already provide the capability, we should add some new arrays for the other two. Please note that the compiler should also be made known of these restrictions so that it chooses the right registers when it does register allocation.

6. CONCLUSIONS

Given new custom instructions, we can always retarget GCC towards our *extended*[5] processor. The changes to make in the Simplescalar infrastructure to simulate a processor with this new instruction have also been detailed. However, we have identified some reasons why the GCC combiner might not be able to generate optimised code and is very limiting. Since the compiler's combiner phase is rather independent and can be invoked using a function from the top-level, we should consider writing a better combiner phase if we want to use GCC for compiling for extended processors. However, for very basic compiler generation for extended processors, the existing GCC is probably acceptable to some.

7. FUTURE WORK

Automatically adding new instructions in all necessary tools like a Simplescalar simulator and a GCC compiler based on a specification should be the first plausible continuation of this work. Later, a new combiner may be plugged in the place of the current GCC pattern combiner. The combiner should be smart enough to take care of the memory references that are register spills into a variable's home location.

[5]Extensible processor to which new instructions have been added

For this to happen, some new kind of annotations may need to be created in the RTL representation for memory instruction to differentiate between spill code and statements that are a consequence of the high-level language features like `volatile` in C/C++. The ability or limitations of the combiner effectively dictates the optimised nature of the code when we are using complex architectures.

8. ACKNOWLEDGMENTS

I would like to thank Dr.Tulika Mitra and Mr.Pan Yu for their suggestions and support.

9. REFERENCES

[1] http://www.arc.com. Arc tanget processor.

[2] http://www.mips.com/pro. Mips pro - corextend.

[3] http://www.simplescalar/v4test.html. Simplescalar version 4 release page.

[4] http://www.tensilica.com. Xtensa processor.