Instruction Set Commutivity *

Phillip J. Windley
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow ID 83843

Abstract

We present a state property called congruence *and show how it can be used to demonstrate commutivity of instructions in a modern load–store architecture. Our analysis is particularly important in pipelined microprocessors where instructions are frequently reordered to avoid costly delays in execution caused by hazards. Our work has significant implications to safety and security critical applications since reordering can easily change the meaning and an instruction sequence and current techniques are largely ad hoc. Our work is done in a mechanical theorem prover and results in a set of trustworthy rules for instruction reordering. The mechanization makes it practical to analyze the entire instruction set.*

1 Introduction.

Instruction pipelining [1] is critical to good performance in modern microprocessors. Almost every microprocessor developed in the last several years contains an instruction pipeline. Significant attention has been given to the development of scheduling algorithms to reduce the the occurrence of pipeline hazards because of the performance degradation that they cause. Typically, scheduling involves reordering the instruction stream produced by a compiler.

The approaches to code reordering are largely *ad hoc* with little or no analysis showing whether the rules are correct and under what conditions they should be avoided. Indeed, when asked about the rules that he gave for avoiding semantic changes one researcher, who developed the code scheduling algorithms for the C compiler in a widely available commercial UNIX system, stated "Its all *ad hoc*. It never occurred to me that there might be any other way to do it."

Clearly, the current approach to code scheduling is unacceptable in safety and security critical applications. On one hand, modern pipelined architectures perform poorly without the aid of a compiler that is smart enough to reschedule code to avoid pipeline hazards. On the other hand, the

*This work was sponsored by the Department of Defense under University Research Program contract No. MDA904–91–C–7054

[1] See [HP90] for an excellent introduction to pipelining and pipeline hazards.

reordering that the scheduler performs has the potential to initiate semantic changes in the code stream. We must either give up performance or live with untrustworthy code. Neither of these approaches is satisfactory.

This paper describes the analysis of a microprocessor instruction set for commutivity. We are interested in establishing, by analysis, under what circumstances the instructions can be reordered while avoiding semantic changes. The next section describes related work and the following sections present our analysis.

2 Related Work

The formal analysis of code reordering is related to at least three active areas of research: microprocessor verification, compiler verification, and the automatic generation of optimizers. This section discusses these three areas of research and relates them to the work presented in this paper.

Microprocessor Verification There have been numerous efforts to verify microprocessors. These efforts have been mostly for research purposes and none have included any kind of analysis of their instruction sets regarding code reordering. Only one formally verified general purpose microprocessor has been fabricated and it has so few features as to be impractical for real use. Descriptions of these efforts can be found in [Coh88, Joy89a, Joy88, Hun89]. It is important to note that none of these projects involved verification of a pipelined processor.

In [SB90], Srivas *et al.* describe the formal verification of a pipelined microprocessor called Mini Cayuga, comparable in complexity of design to that of Hunt's FM8501. However, the structure and behavior of the pipeline were hidden from the abstract specification. Only prefetching of the next instruction was incorporated into the specification. This precluded the possibility of formally reasoning about pipeline hazards and instruction scheduling.

We are designing, specifying, and verifying a microprocessor called *AVM–2*. *AVM–2* has a load store architecture and will be pipelined. The architecture of *AVM–2* is largely the same as that of *AVM–1*, but the design is substantially different. We described early results of this research in [Win91] where we demonstrated the integrity of the supervisory mode of *AVM–1*. In this paper, we describe results regarding instruction set commutivity for *AVM–2*.

Compiler Verification There has been much work on verified compilers. Space considerations do not present a full treatment here. Joyce [Joy89b] gives an excellent review. Most of the early work [Rus77, Coh80, CM86] on the compiler correctness problem used idealizations of the hardware. Recent work by Joyce [Joy89b], Moore [Moo88], and Young [You89] have looked at compiler verification under the constraints of a real instruction set. None of these efforts has addressed code reordering although Young states that initial work on an optimizer has begun. Even so, our approach is different from Young's due to the nature of the specification. The specifications in Young's work are operational while ours are denotational.

Optimizer Generation Current approaches to instruction scheduling is largely *ad hoc*. Current compiler technology utilizes rule–based, heuristic algorithms for optimizing code sequences. Representative of the state of the art, the IBM RISC System/6000 XL compiler family uses special flags associated with opcodes to indicate instructions which are "dangerous" to move [War90, War92]. To date, there has not been any published results describing the application of formal methods to pipeline scheduling.

There has been some work on the automatic generation of optimizers from specifications of one sort or another.

In [Kes84], Kessler describes a tool called Peep. Peep is an architectural description driven peephole optimizer. The description of the architecture, given in LISP, is used to generate a table of optimizable instructions that can be used in a an optimizing compiler.

In [DF84], Davidson and Fraser present a system that generates peephole optimizations called PO. PO uses productions which describe the effect of assembly language instructions in a simulator to determine substitutions for 2 and 3 assembly instruction sequences.

While this work is interesting and related to the work presented here, these efforts differ in several important ways:

- The descriptions used for generating optimizations are not related to the implementation in anyway. Our work uses a specification that is related through proof to the implementational specification.
- It is not clear whether or not any kind of theory underlies the generation of the optimizers. As we will show later, there are concepts regarding reordering that can be generalized and used as a basis for reasoning about reordering.
- it is not clear how much faith can be placed on the simulation used to determine equivalent sequences. Our work will be done in a widely accepted theorem proving environment.

3 *AVM–2*

We have designed a computer designated *AVM–2* (*A Verified Microprocessor*). *AVM–2* is a second generation design that will be implemented in CMOS. The design, specification, and verification of *AVM–1*, the predecessor to *AVM–2*, are given in [Win90] where it is used as an example to demonstrate the utility of generic models in hardware verification. *AVM–2*, like *AVM–1*, will feature a RISC–like instruction set and a large register file. Unlike *AVM–1*, *AVM–2* will have a pipelined implementation.

The Registers. *AVM–2* has a load–store architecture based on a large register file. The register file is divided into seven supervisor–mode registers and twenty–four general purpose registers.

Two additional registers are visible at the architectural level: the program counter and the program status word. The program counter (denoted `pc`) is used to sequence the computer—it indicates which instruction in memory to execute next. The program status word (denoted `psw`) is used to keep track of the status of the last ALU operation, whether or not interrupts are enabled, and the privilege level of the CPU.

The Instruction Set. *AVM–2* has 30 programming level instructions. There is a group of eight, 3–argument (source A, source B, and destination) arithmetic and logical instructions and another group of 8 arithmetic and logical instructions that use two arguments and a 16–bit immediate value. There are 4 instructions for loading and storing registers. Only the load and store instructions communicate with memory. In addition, there are instructions for performing user interrupts, jumps, subroutine calls, and shifts.

4 Instruction Set Specification

The instruction set for *AVM–2* has been formally specified as part of the design process. The specification represents a denotational description of the machine language. In this section, we present several of the state transition functions representing instructions. These examples will be used in later sections describing the analysis.

The instructions are modeled by state transition functions. In general, each function operates on a state tuple and an environment tuple. The state tuple, contains variables representing the register file, `reg`, the program status word, `psw`, the program counter, `pc`, and the memory, `mem`. The environment tuple contains variables representing the interrupt vector, `ivec`, the interrupt line, `int`, and the reset line, `reset`. Each function returns a state tuple updated to reflect the behavior of the instruction being modeled.

The `NOOP` instruction updates the state tuple by incrementing the program counter. No other actions are performed.

```
⊢def NOOP (reg, psw, pc, mem)
          (ivec, int, reset) =
       let new_pc = inc pc in
       (reg, psw, new_pc, mem)
```

Note that `NOOP` is *not* an identity function, although it is often thought of that way. The fact that `NOOP` does affect the state and resides in memory affects its commutivity.

Other instructions are quite a bit more complicated than the `NOOP` instruction. For example, the `ADD` instruction is shown below:

```
⊢def ADD (reg, psw, pc, mem)
         (ivec, int, reset) =
    let a = EL (GetSrcA pc mem) reg and
        b = EL (GetSrcB pc mem) reg and
        d = GetDest pc mem in
    let result = add (a, b) in
    let cflag  = addp (a, b, result) and
        vflag  = aovfl (a, b, result) and
        nflag  = negp result and
        zflag  = zerop result and
        sm     = get_sm psw and
        ie     = get_ie psw in
    let new_reg = UPDATE_REG psw d reg result and
        new_psw = mk_psw(sm, ie, vflag,
                         nflag, cflag, zflag) and
        new_pc = inc pc in
    (new_reg, new_psw, new_pc, mem)
```

The ADD instruction updates every member of the state tuple except the memory. The primary action, summing two registers and updating the register file accordingly is reflected by the updated register file, reg. The instruction also calculated new values for the overflow, carry, negative, and zero flags of the program status word, psw. The supervisory mode bit and the interrupt enable bit remain unchanged, as expected. The program counter, pc is incremented.

5 State Congruence

Our notion of state correspondence is motivated by the denotational description of the instruction set. Because the specification is denotational, we are interested in showing that state transitions made by one sequence of instructions are equivalent to the state transitions made by another sequence of instructions.

Unfortunately, the instructions sequences themselves are part of the state and so we cannot use equivalence as the relation between states. Instead, we relate the states using a property we call congruence. We call this relationship *congruence* because the states are equivalent except for (i.e. *modulo*) the ordering of the instruction sequences in memory.

Figure 1 illustrates congruence. As the figure shows, we start with two states, S1 and S2, which are related by the relation ρ. After they have been transformed by a sequence of instructions, we are left with two modified states S1' and S2'. Ideally, these new states are still related by ρ, but as we will see in Section 5.2, this is not always the case and so we show them related by ρ'.

The primary congruence relation that we use in the examples is given by the following predicate:

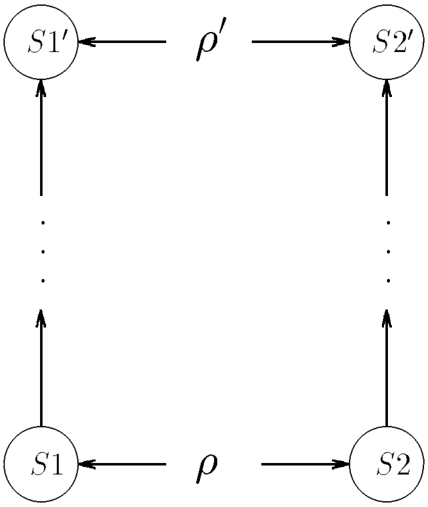

Figure 1: State Congruence

```
⊢def Congruent loc (reg1,psw1,pc1,mem1)
                   (reg2,psw2,pc2,mem2) =
   (reg1=reg2) ∧ (psw1=psw2) ∧ (pc1=pc2) ∧
   (∀  a .  ¬(a = address loc) ∧
            ¬(a = address (inc loc)) ⇒
            (fetch (mem1,a) = fetch (mem1,a))) ∧
   (fetch (mem1,address loc) =
             fetch (mem2,address (inc loc))) ∧
   (fetch (mem2,address loc) =
             fetch (mem1,address (inc loc)))
```

The predicate operates over a location and two state tuples. We say that the state tuples are *congruent* if their register files, `reg1` and `reg2`, program status words, `psw1` and `psw2`, and program counters, `pc1` and `pc2`, are the same. Additionally, we require that the memories, `mem1` and `mem2` be the same except that the words located at `loc` and `loc+1` in `mem1` are swapped in `mem2`.

5.1 Swapping NOOP

To start with, we examine the commutivity of the `NOOP` instruction. While this may seem like a trivial problem, the problem is not as straightforward as it seems since `NOOP` does affect the state. Also, the difficulties encountered in `NOOP` commutivity are typical of other commutivity proofs.

The following theorem shows that `NOOP` can be commuted with any instruction that does not modify memory or alter program flow:

```
⊢ let s1  = (reg1,psw1,pc1,mem1) and
      s2  = (reg2,psw2,pc2,mem2) and
      e = (ivec,ireq,reset) in
   ∀ inst.
   (inst = macro_inst (Opcode s1 e)) ⇒
   Congruent pc2 s1 s2 ∧
   NON_MEM_INST (Opcode s1 e) ⇒
   let s1′ = (NOOP (inst s1 e) e) and
       s2′ = (inst (NOOP s2 e) e) in
   Congruent pc2 s1′ s2′
```

We assume that the environment does not change during execution of the two instructions (e, representing the environment, is used as an argument for both of them). The theorem states that for every instruction in the instruction set, if the initial states are congruent the modified states are also congruent.

The theorem is not true for instructions that alter flow control because the NOOP would never execute in one case and some other instruction would execute in its place. For instructions that modify memory, we can prove a more restricted version that assumes that the memory instruction does not interfere with the program (instructions and data are stored in the same memory). We will see an example using a non–interference condition in the next section.

5.2 Swapping Arithmetic Instructions

The arithmetic instructions of *AVM–2* that do not use the carry flag commute under weak congruence. Weak congruence is the same as strong congruence (defined above), except that it does not require equivalence for the entire program status word. Rather we require only that the supervisory mode bit and the interrupt enable bit be the same. Most instructions that modify the overflow, carry, negative, and zero flags of the program status word do so without regard to their previous values, so the value of the two program status words cannot be equal.

```
⊢def Weak_Congruent loc (reg1,psw1,pc1,mem1)
                        (reg2,psw2,pc2,mem2) =
     (reg1 = reg2) ∧
     (pc1 = pc2) ∧
     ((get_sm psw1) = (get_sm psw2)) ∧
     ((get_ie psw1) = (get_ie psw2)) ∧
     (∀ a . ¬(a = address loc) ∧
            ¬(a = address (inc loc)) ⇒
            (fetch (mem1,a) = fetch (mem2,a))) ∧
     (fetch (mem1,address loc) =
               fetch (mem2,address (inc loc))) ∧
     (fetch (mem2,address loc) =
               fetch (mem1,address (inc loc)))
```

In order to commute two arithmetic instructions, we require that they be non–interfering. That is, the destination registers cannot be the same as the source registers. For example, the following instructions do not commute:

```
a := b + c
d := a + e
```

Since the second instruction uses the value computed in the first, we cannot swap them without changing the resulting state. In addition, we require that the destination registers be different. The following predicate defines the non–interference property in terms of the functions used by the instruction set definition to retrieve the source and destination register indices from memory.

```
⊢def Non_Interfering pc mem  =
    ¬(GetSrcA (inc pc) mem = (GetDest pc mem)) ∧
    ¬(GetSrcB (inc pc) mem = (GetDest pc mem)) ∧
    ¬(GetDest (inc pc) mem = (GetDest pc mem))
```

Using the weak congruence predicate and the non–interference predicate, we can show, for example, that ADD and SUB commute:

```
⊢   let s1  = (reg1,psw1,pc1,mem1) and
        s2  = (reg2,psw2,pc2,mem2) and
        e = (ivec,ireq,reset) in
     Congruent pc2 s1 s2 ⇒
     Non_Interfering pc1 mem1 ∧
     Non_Interfering pc2 mem2 ⇒
     let s1′ = (ADD (SUB s1 e) e) and
         s2′ = (SUB (ADD s2 e) e)) in
     Weak_Congruent pc2 s1′ s2′
```

Note that we use strong congruence in the assumptions, but can only show weak congruence between the resulting states.

We have presented only two theorems regarding instruction commutivity in *AVM–2*. We can prove more general theorems about the commutivity of arithmetic instructions. We can also show arithmetic instructions commute with load and store instructions provided they are non–interfering.

6 Discussion

We have presented only a few small theorems regarding the analysis of the *AVM–2* instruction set. A more thorough analysis is presently underway. Even so, we believe the results to be interesting.

The fact that we can only show weak congruence when commuting arithmetic instructions is a function of the design of the instruction set. Other instruction sets would provide different results. The contribution of formal analysis is that this property is clearly and unambiguously stated in the resulting theorem.

Also, the weak congruence result affects when we can actually commute arithmetic instructions in a program. We cannot, for example, commute two arithmetic instructions that are followed by a conditional jump since the values of the flags are changed. Strong congruence is required to maintain the program meaning in this case.

We should note that our initial efforts in this area have had an affect on the architecture of *AVM–2*. In light of the results presented here regarding weak congruence, we have undertaken a modification of the instruction set semantics so that arithmetic instructions commute under strong congruence. This will allow greater freedom in code reordering to avoid pipeline hazards.

Certainly none of our specific discoveries regarding the *AVM–2* instruction set will surprise veteran compiler writers. The rules that we have demonstrated for code reordering in the *AVM–2* instruction set are well known. What is important, however, is that we are *not* veteran compiler writers. Analysis allowed us to *show* that they were correct rather than relying on years of experience and intuition. This, it seems, is the heart and soul of engineering [Sha90].

7 Future Work

We plan to extend our analysis of commutivity to explore code motion for larger code fragments. Our intent is to automate a complete analysis of instruction commutivity for all instruction pairs and use these results to determine when large instruction sequences are congruent.

As mentioned earlier, the behavioral specification of *AVM–2* will be verified against its implementation. Because of the hierarchical nature of the specification (see [Win90]), the phase–level mode of the pipeline will have a similar structure to the behavioral model of the top–level. We believe that the techniques demonstrated in this paper will allow us to perform an analysis of the pipeline to identify hazards. This work is underway.

8 Conclusion

This paper has presented examples from an analysis of commutivity in a modern instruction set. Commutivity is start at a more general notion of instruction reordering that in important to both compiler optimizations and pipeline scheduling. Analysis of instruction set reordering is important in both safety and security critical applications because of the danger *ad hoc* approaches present to the semantic integrity of the instruction stream.

Our analysis could have been performed without the benefit of a formal specification of the instruction set or a formal statement of the desired properties. However, the formal analysis has several benefits:

- The theorems about commutivity form a set of rules for commuting instructions. These rules have a demonstrated correctness.
- The formal analysis was helpful in finding interferences between instructions that were not immediately obvious.
- The assumptions and results are unambiguously stated and can be used for further reasoning about optimization and scheduling.

- In providing a set of rules about commutivity, the instructions must be analyzed on a case–by–case basis whether the analysis is done by hand or automated. The formal analysis provides a tool for quickly analyzing the instructions.

References

[CM86] Laurian M. Chirica and David F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions On Programming Languages And Systems*, 8:185–214, April 1986.

[Coh80] Avra Cohn. *Machine Assisted Proofs of Recursion Implementation*. PhD thesis, University of Edinburgh, April 1980.

[Coh88] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 27–72. Kluwer Academic Publishers, 1988.

[DF84] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *ACM SIGPLAN 84 Symposium on Compiler Construction*. ACM, June 1984.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hun89] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[Joy88] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[Joy89a] Jeffrey J. Joyce. *Multi–Level Verification of Microprocessor–Based Systems*. PhD thesis, Cambridge University, December 1989.

[Joy89b] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In Miriam Leeser and Geoffrey Brown, editors, *Proceedings of the Mathematical Sciences Institute's Workshop on Hardware Specification, Verification, and Synthesis*, July 1989.

[Kes84] Robert R. Kessler. Peep—an architectural description driven peephole optimizer. In *ACM SIGPLAN 84 Symposium on Compiler Construction*. ACM, June 1984.

[Moo88] J. Strother Moore. A mechanically verified language implementation. Technical Report 30, University of Texas at Austin, 1988.

[Rus77] Bruce D. Russell. Implementation correctness involving a language with `goto` statements. *SIAM Journal of Computing*, 6(3), September 1977.

[SB90] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *Software Engineering*, 7(6):15–24, November 1990.

[War90] Henry S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.

[War92] Henry S. Warren. IBM T.J. Watson Research Center, Private communication, February 1992.

[Win90] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[Win91] Phillip J. Windley. Using correctness results to verify behavioral properties of microprocessors. In *Proceedings of the IEEE Computer Assurance Conference*, June 1991.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5, 1989.