Formal Verification of a Superscalar Execution Unit[1]

Kyle L. Nelson
IBM Corporation
AS/400 Division
Rochester, MN 55901
email: kln@vnet.ibm.com

Alok Jain
Department of ECE
Carnegie Mellon University
Pittsburgh, PA 15213
email: alok.jain@ece.cmu.edu

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
email: randy.bryant@cs.cmu.edu

Abstract. Many modern systems are designed as a set of interconnected reactive subsystems. The subsystem verification task is to verify an implementation of the subsystem against the simple deterministic high-level specification of the entire system. Our verification methodology, based on Symbolic Trajectory Evaluation, is able to bridge the wide gap between the abstract specification and the implementation specific details of the subsystem. This paper presents a detailed description of an industrial application of this methodology to the fixed point execution unit of the PowerPC processor. We were able to verify a representative instruction under all possible stall, bypass, pipeline conditions and under all possible timings for interface to other functional units in the processor.

1. Introduction

Some modern systems with a simple deterministic high-level specification have implementations that exhibit highly nondeterministic behaviors. A large class of systems that exhibit such behavior are processors. At the high-level the sequencing model inherent in processors is the sequential execution model. However, at the low level, these processors are implemented as a set of interconnected reactive subsystems. The subsystems have complex interfaces and use nondeterministic protocols to interact with each other. In addition, the underlying implementation of subsystems uses features such as pipelines and dispatching multiple instruction per cycle in an effort to increase performance. The interaction among instructions results in increased interlock and resource conflict problems which leads to nondeterminism in the subsystem. Such subsystems contain many subtle features with the potential for serious design error.

A methodology for formal verification of such subsystems presents a unique set of challenges. The goal is to verify the implementation of the subsystem against the more natural high-level specification of the entire system. The verification methodology has to incorporate the ability of defining the environment around the subsystem. The environment defines the set of restrictions and requirements placed on the subsystem by the rest of the system. The restrictions and requirements are usually in the form of a set of nondeterministic protocols defined on the interface signals. In addition to defining these interfaces, the methodology has to account for complex features such as instruction pipelines, pipeline interlocks, multiple instruction issue, multiple cycle instructions and speculative execution. Though formal verification tools have started gaining acceptance in the industry[8][9][10][11], they do not provide a rigorous methodology for subsystem verification.

Our verification methodology is able to bridge the wide gap between the abstract specification of the entire system and the subsystems' often radical deviation from the sequential execution model. This paper focuses on applying our methodology to verify the fixed point unit of a PowerPC processor. The fixed point unit represents a subsystem with a complex interface and several of the performance enhancing features found in modern day processors.

A high level overview of our methodology for subsystem verification and some of the related work is presented in Section 2. Section 3 discusses the implementation details of the fixed point unit. The steps required by our methodology to verify the fixed point unit are detailed in Section 4. The results of the verification are presented in Section 5.

2. Overview of Verification Methodology

The goal is to develop a methodology with which a designer can show that an implementation of the subsystem correctly fulfills an abstract specification of the desired system behavior. The abstract specification describes the high-level behavior of the system independent of any timing or implementation details. As an example, the natural specification of a processor is the instruction set architecture. The specification is a set of *abstract assertions* defining the effect of each operation on user-visible state elements. The verification process must bridge a wide gap between the detailed subsystem implementation and the abstract specification. In spanning this gap, the verifier must account for issues such as system clocking, pipelines and interfaces with other subsystems. To bridge this gap, the verification process requires some additional mapping information. The *implementation mapping* relates the abstract state elements in the assertions to signals in the subsystem. The implementation mapping is a nondeterministic mapping defined in terms of state diagrams. State diagrams allow users to create an environment around the subsystem and define complex nondeterministic interface protocols. The state diagrams corresponding to the inputs can be viewed as generators that generate low-level signals required for the operation of the subsystem. State diagrams corresponding to outputs can be viewed as acceptors that recognize low-level signals on the outputs of the subsystem. In addition to defining the environment, the mapping also has information about how to stitch instructions together to create infinite execution sequences.

The abstract specification and the implementation mapping are used to generate the *trajectory specification*. The trajectory specification consists of a set of *trajectory assertions*. Each abstract assertion gets mapped into a trajectory assertion. The trajectory assertion captures all possible sequences of circuit state that arise due to nondeterministic interactions of the signals in the environment around the subsystem. A modified form of symbolic simulation called Symbolic Trajectory Evaluation (STE)[1] is used to verify the trajectory assertions on the subsystem.

The reader is referred to [4] for a more detailed description of our verification methodology.

2.1. Related Work

Beatty[2] laid down the foundation for our methodology for formal verification of systems. However his work had one basic limitation. The methodology could handle only bounded single behavior sequences. We have extended the methodology to handle a greater level of nondeterministic behavior required for subsystem verifica-

[1]This work partially funded by Semiconductor Research Corporation #96-DC-068

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06..$3.50

tion.

STE has been used earlier to verify trajectory assertions. Beatty [2] mapped each abstract assertion into a set of symbolic patterns. STE was used to verify the set of symbolic patterns on the circuit. The set of symbolic patterns corresponded to a single sequence of states in a state diagram. Seger[3] extended STE to perform fixed point computations to verify a single sequence of states augmented with a limited set of loops. We have extended STE to deal with arbitrary state diagrams.

Our work has some resemblance to the capabilities provided by the Symbolic Model Verifier (SMV)[5][6]. SMV requires a closed system. The environment is modeled as a set of machines. The state diagrams in our mapping correspond to creating an environment around the system. However, there is one essential difference. Though SMV does provide the capability of describing the environment, it does not provide a methodology for rigorously defining these machines and stitching them together to reason about infinite execution sequences. The other difference is that the model-checking algorithm in SMV requires the complete next-state relation. It would be impossible to obtain the entire next-state relation for a complex subsystem. On the other hand, we use STE to evaluate the next-state function on-the-fly and only for that part of the subsystem that is required by the specification.

Kurshan[7][8] has used the concept of mappings to perform hierarchical verification. Kurshan uses reduction transformations as a basis of complexity management and hierarchical verification. Reductions are homomorphic transformations which correspond to abstraction mappings. Inverse reductions (called refinements) correspond to our implementation mappings. However Kurshan does not have the concept of stitching tasks together to create an infinite sequence of tasks. Also Kurshan uses language containment as opposed to STE as the verification task.

3. Overview of the Fixed Point Unit

The fixed point unit being verified is part of a superscalar implementation of the PowerPC architecture[12] used in IBM's AS/400 Advanced 36 computer[13]. The FXU interfaces with the branch processing unit (BPU) and the load store unit (LSU). The FXU is responsible for executing all fixed point instructions other than loads and stores. Instructions are received from the BPU. The BPU can dispatch a maximum of two instructions per cycle depending on the number of valid prefetched instructions and the interlock conditions set by previously dispatched instructions. All instructions are pre-decoded and steered to and acknowledged by the correct functional unit. If the unit is busy and can not receive the instruction, then no acknowledgment will be given to the BPU. The instruction will stall in the dispatch stage, and the BPU will try to dispatch it again in the following cycle.

The FXU processes instructions in two stages, the decode stage and the execute stage. In the decode stage, the latched instruction is decoded into the instruction fields. In this stage, requests for all required source operands are made, and a target operand, if required, is reserved. The required register source operands will come from one of two places. If the source's register address is equivalent to the target's register address of the execute stage's instruction, then the source data will be forwarded to the decode stage, bypassing the register file. Otherwise, if there is no register match or if there is some special circumstance in which the hardware does not support register bypassing, a request for the data is sent to the LSU, where the general purpose register (GPR) file resides. Part of the LSU's function is to manage all the interlocks involved in the allocation of register resources. Instructions will stall in the decode stage until all of the source operands are available and the execute stage is available. The execute stage is considered available when either it does not currently contain a valid instruction or the instruction in the execute stage is going to be completed in the current cycle.

```
WHEN (ra != rb) || (dataA == dataB)                          (1)
  (op is OR) and (RA is ra) and (RB is rb) and (RT is rt) and (2)
  (Reg[ra] is dataA) and (Reg[rb] is dataB)                  (3)
        LEADSTO                                              (4)
  (Reg[rt] is dataA | dataB)                                 (5)
```

Figure 1. Abstract assertion for the `OR` instruction

When the execute stage begins, all required operands are available, and the current instruction is latched into the execute stage instruction register. Most instructions will complete the execute stage in a single cycle. Multiply and divide instructions are the exceptions. Result data is provided to the LSU to be stored in the GPR file, and the instruction is complete when the LSU acknowledges the store. This acknowledgment may be delayed, causing the instruction to stall, if there are previous instructions in the instruction stream that still have the possibility of causing an interrupt.

The complexity resulting from both the interlock logic that enforces pipeline stalls and the non-deterministic interfaces between the FXU and the other functional units are common in modern processors. These features are a significant source of errors and increase the difficulty of the verification task.

4. Fixed Point Unit Verification

4.1. Specification

The initial step in verifying the FXU is to formally document its specification. The specification can be directly taken from the portion of the PowerPC's instruction set architecture that it implements. Each instruction that the FXU executes can be expressed as an abstract assertion. In our methodology, an abstract assertion is of the form: *P* **LEADSTO** *Q*, where *P* serves as the precondition and *Q* as the postcondition. The conditions *P* and *Q* are a conjunction of clauses where each clause is an assignment to an abstract state element. The precondition expresses some assumed conditions over the system state and the postcondition expresses the condition that should result. The set of all the assertions form the functional unit's specification.

The abstract system level assertion of a specific instruction, the bitwise `OR` instruction, is shown in Figure 1. This instruction computes the bitwise `OR` of two source registers and stores the result in a target register. The assertion is completely implementation independent. Lines 2-3 contain the precondition of the assertion. First it specifies that the opcode must be that for the `OR` instruction. Next, by using symbolic variables, it specifies that the two source operands, `RA` and `RB`, and the target operand, `RT`, can be any register address. It also specifies the contents of the two source registers to be the symbolic values `dataA` and `dataB`. The assertion's postcondition is specified in line 5. It is simply that the contents of the target register will contain the bitwise `OR` of the data in the two source registers. Line 1 of the assertion is a condition that filters out illegal input patterns. An illegal pattern would be when the two source operands refer to the same register address and the data contained in the two source registers is different.

4.2. Implementation Mapping

The implementation mapping consists of a *main machine* and a set of *map machines*. The main machine defines the flow of control for individual system operations. The map machines define a mapping for the state elements in the assertion. Each state element is associated with a single map machine. The main machine and the map

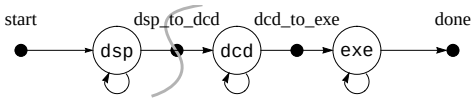

Figure 2. Main machine

machines are nondeterministic machines that are modeled as *control graphs. C*ontrol graphs are state diagrams with the capability of synchronization at specific time points. A control graph has two sets of vertices: 1) State vertices that represent some non-zero duration of time and 2) Event vertices that represent instantaneous time points. A control graph has a *source*, an event vertex with no incoming edges, and a *sink*, an event vertex with no outgoing edges. Nondeterminism is modeled as multiple outgoing edges from a vertex.

The main machine is a control graph with a cutset of event vertices that denotes the nominal end of the current operation and start of the successive operation. The main machine for the FXU is shown in Figure 2. The vertices labelled `dsp`, `dcd` and `exe` are state vertices that represent the three pipeline stages in the FXU. The self loops on each state vertex represent that an instruction can stall in each pipe stage for a nondeterministic period of time. The rest of the vertices in the figure are event vertices. The cutset `dsp_to_dcd` represents the fact that the successive instruction can be started at this point thus overlapping the decode stage of the current instruction with the dispatch stage of the successive instruction.

Once the main machine is defined, then a map machine for each state element is defined. The map machine is a control graph with node formulas and synchronization points. Node formulas are on the state vertices and refer to assignments to actual nodes or signals in the circuit. Node formulas can either be *antecedent node formulas* or *consequent node formulas*. Antecedent node formulas define the stimuli and current state for the circuit. Consequent node formulas define the desired response and state transitions. Both types of node formulas can be associated with a single state vertex. Synchronization is on the event vertices and synchronizes event vertices in the map and main machines.

The map machine can be used to define the protocols on the interface signals of the FXU. A single clause in the assertion is often expanded to protocols involving the interface signals of the interacting functional units. In the case of the FXU and the `OR` assertion, the (`op` **is** `OR`) clause is mapped into a protocol involving interface signals between the BPU and the FXU. This protocol is carried out during the instruction dispatch stage. Similarly, the clause (`Reg[ra]` **is** `dataA`) is mapped into a protocol between the LSU and the FXU. Here the FXU must request the register data from the LSU since the LSU manages the allocation of registers.

The implementation mapping must also consider the flow of the instruction through the particular functional unit. This often exposes some of the functional unit's internal states. Figure 3 shows the high level flow of an instruction through the three pipeline stages. First, the BPU must successfully dispatch the instruction to the FXU. In the dispatch stage, the instruction is predecoded and it is determined whether the source operand data will bypass the register file. This determination is based on the dispatch stage's pre-decode, the current decode stage instruction, and instruction dispatch information from the LSU. In the decode stage, data for the source operands is obtained. If bypassing was determined to occur, this data comes from the execute stage; otherwise it comes from the LSU. Also, the target register is reserved during the decode stage. Additionally, the instruction will stall in the decode stage until the execute stage is available. In the execute

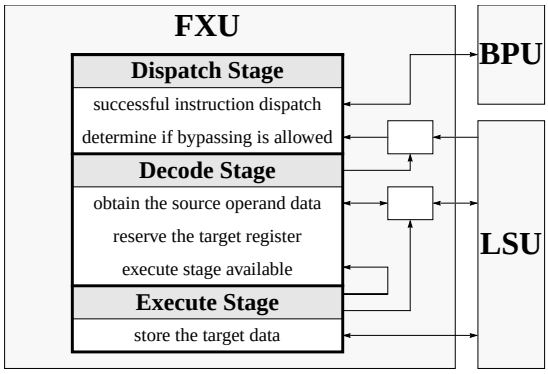

Figure 3. Instruction flow and interactions through the FXU

stage, the target data is calculated and stored in the register file. The execute stage completes when the LSU acknowledges this store.

The implementation mapping for each clause in the abstract assertion will take into account one of more of these details. The focus of the discussion here will be on the mappings for the `OR` assertion's clauses (`Reg[ra]` **is** `dataA`) and (`Reg[rt]` **is** `dataA|dataB`).

The A operand data is received during the decode stage and will originate from one of two sources: either from the LSU, where the register file is located, or from the FXU's execute stage instruction. The signals that are involved in this transaction are shown in Figure 4. If the data originates from the LSU, the FXU will assert `fx_a_req` and set `fx_a_sel` to `ra`, the register address. When the register data is available, the LSU will assert `ls_a_valid` and `ls_a_data`. If instead, the data originates from the FXU's execute stage, the data will be received when the execute stage completes its computation. The execute stage will assert `ex_trgt_valid` and the target data that is to be forwarded will be in `ex_trgt_data`. After the decode stage is complete, the instruction moves into the execute stage. When the target data is computed, it will attempt to store the data into the target register. The execute stage will assert `ex_trgt_valid`, `ex_trgt_data`, and `ex_trgt_sel` until the LSU completes the handshake with `ls_trgt_ack`.

This transaction is captured in the map machine for the clause (`Reg[ra]` **is** `dataA`), shown at the top of Figure 5. The protocol is defined by associating antecedent and consequent node formulas with each state in the control graph. The antecedent node

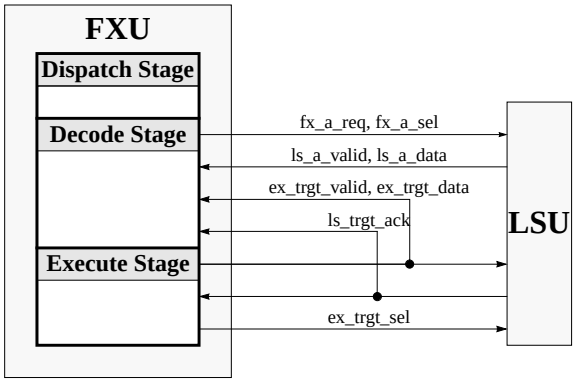

Figure 4. A operand data and target data interface signals

formulas are shown in the upper half of the shadowed boxes and consequent node formulas are shown in the lower half. The control graph for `Reg[ra]` is synchronized with the decode stage of the main machine as indicated by the dashed lines at the control graph's source and sink.The control graph has two legs, the top leg maps the case where the data comes from the LSU, and the bottom leg maps the bypass case. The consequent node formulas of the first two states, s1 and s2, of the top leg indicate that the FXU is requesting the data from register `ra`. Potentially the instruction can wait an indefinite period of time in the state s1 before `ls_a_valid` is asserted in the state s2. The register data is received from the LSU in state s2. Once the data is received, the FXU latches it up for use in the execute stage. After receiving the data, the instruction may remain in the decode stage while other resources are being attained. This is represented in state s5. During this period, the FXU drops its request to the LSU since the data is being internally stored.

The bottom leg, the bypass case, of the graph has exactly the same flow, only the node expressions are different. On this path, `fx_a_req` is never asserted because the data is not originating from the LSU. Instead, the data will be supplied by the signal `ex_trgt_data` in state s4.

The implementation mapping needs to expose some of the internal signals in the FXU. The mapping for the bypass case of the A operand data clause is dependent upon two execute stage related signals, `ex_valid` and `ex_trgt_valid`. Their map machines are shown in Figure 5. The signal `ex_valid` is asserted when the execute stage contains a valid instruction, and `ex_trgt_valid` is asserted when the execute stage has completed the computation of the target data. These signals represent the state of the preceding instruction and can be considered inputs into the decode stage. As a result, their node formulas are antecedent node formulas, and the control graph is synchronized to the decode stage. Later in the discussion of the mapping for the target data clause, these signals will be considered as outputs of the execute stage, in which case their control graphs will be synchronized to the execute stage of the main machine and the antecedent node formulas will become consequent node formulas. In general, state elements that are contained in both the precondition and the postcondition are always mapped by the same control graph, only the role of the antecedent node formulas and the consequent node formulas are reversed.

As shown in Figure 5, the top leg of the control graph for `ex_valid` is the case where the execute stage does not have a valid instruction. In state s2 on the bottom leg, the execute stage is valid. It is possible that the execute stage instruction completes while the current instruction is still stalling in the decode stage. This explains the path from state s2 to state s1. A key point of this graph is the synchronization line between itself and the `Reg[ra]` graph. Synchronizing the bottom legs of these two machines guarantee that if the A operand data is being forwarded by the execute stage, then the execute stage must contain a valid instruction.

Similarly, the mapping for (`Reg[ra]` **is** `dataA`) is dependent upon `ex_trgt_valid`. The top leg of this control graph is the case where the execute stage does not contain a valid instruction as indicated by the antecedent node formula. On the bottom leg, the antecedent node formula for state s2 indicates that the target data is not yet valid. In state s3 the execute stage has completed its computation, and the data is valid. This signal will remain asserted until the LSU acknowledges receiving the target data for the execute instruction by asserting `ls_trgt_ack`. Once the acknowledgment is received, then the execute stage completes. As shown in the mapping for `ls_trgt_ack`, this acknowledgment from the LSU is received in the last occurrence of `ex_trgt_valid`'s state s3. The first two synchronization lines ensure that the top leg is taken only when the execute stage does not have a valid instruction and that the bottom leg is only taken when the execute stage does have a valid instruction. The next synchronization line guarantees that the A operand data is received on the first cycle that `ex_trgt_valid` is asserted. The final synchronization line ensures that when `ex_trgt_valid` is de-asserted, `ex_valid` is also de-asserted because the execute stage instruction has completed.

The last graph of Figure 5 is the mapping for the target data clause (`Reg[rt]` **is** `dataA | dataB`). This control graph is synchronized to the execute stage of the main machine. Its node formulas relate to FXU output signals, therefore they are consequent node formulas. Two other output signals of the execute stage must also be mapped during the execute stage. These signals are `ex_valid` and `ex_trgt_valid`. These signals have been discussed with respect to the decode stage where they were considered as inputs. The execute stage mapping of these signals was automatically derived from its decode stage map machines. Additionally, antecedent node formulas have become consequent node formulas. In effect, the execute stage control graph for `ex_valid` becomes just the bottom leg of its decode stage mapping. This is because the instruction being verified is in the execute stage, so clearly the execute stage contains a valid instruction.

The execute stage mapping of `ex_trgt_valid` is in effect just state s3. This is because the `OR` instruction being considered here requires a single cycle to compute its target. As a result, `ex_trgt_valid` will be asserted in the first cycle of the execute stage.

The signal `ls_trgt_ack` is an input to both the decode and execute stages. Its execute stage mapping is automatically derived from the decode stage map machine. The control graph in the execute stage is the leg of the decode stage's mapping in which the execute stage has a valid instruction. Because this signal is an input from the LSU in both the decode and the execute stages, its node formulas are antecedent node formulas.

5. Results

The implementation mapping was specified for the FXU. It defines 24 control graphs representing inputs, outputs and internal states of the FXU. Only five of these control graphs have been shown here. The remaining control graphs map the other interactions outlined in Figure 3. The abstract assertion for the `OR` instruction and the implementation mapping were used to automatically generate the trajectory assertion. STE was used to verify the `OR` trajectory assertion on a gate-level model of the FXU. The gate-level representation of the FXU contains approximately 28000 multi-input gates and over 1000 latches.

5.1. Trajectory Generation

The trajectory assertion corresponds to the composition of the 24 map machines defined in the implementation mapping for the bitwise `OR` assertion. Composition amounts to taking the cross-product of these aligned map machines under restrictions placed by the synchronization function. Figure 6 shows the trajectory assertion that was generated for the `OR` assertion. The trajectory assertion corresponds to all possible cases that can arise due to interactions between the map machines. The trajectory assertion has 488 vertices, 34 loops and 28,602 paths. Each path represents a unique ordering of events for a particular set of inputs and current states.

While it is not feasible to go into details about the composition of each state in the trajectory assertion, we are able to make some intuitive sense out of it. First notice the two horizontal cutsets in the graph. These correspond to the nodes `dsp_to_dcd` and `dcd_to_exe` shown in the main machine, Figure 2. The difference is that the state vertices `dsp`, and `dcd` have been expanded,

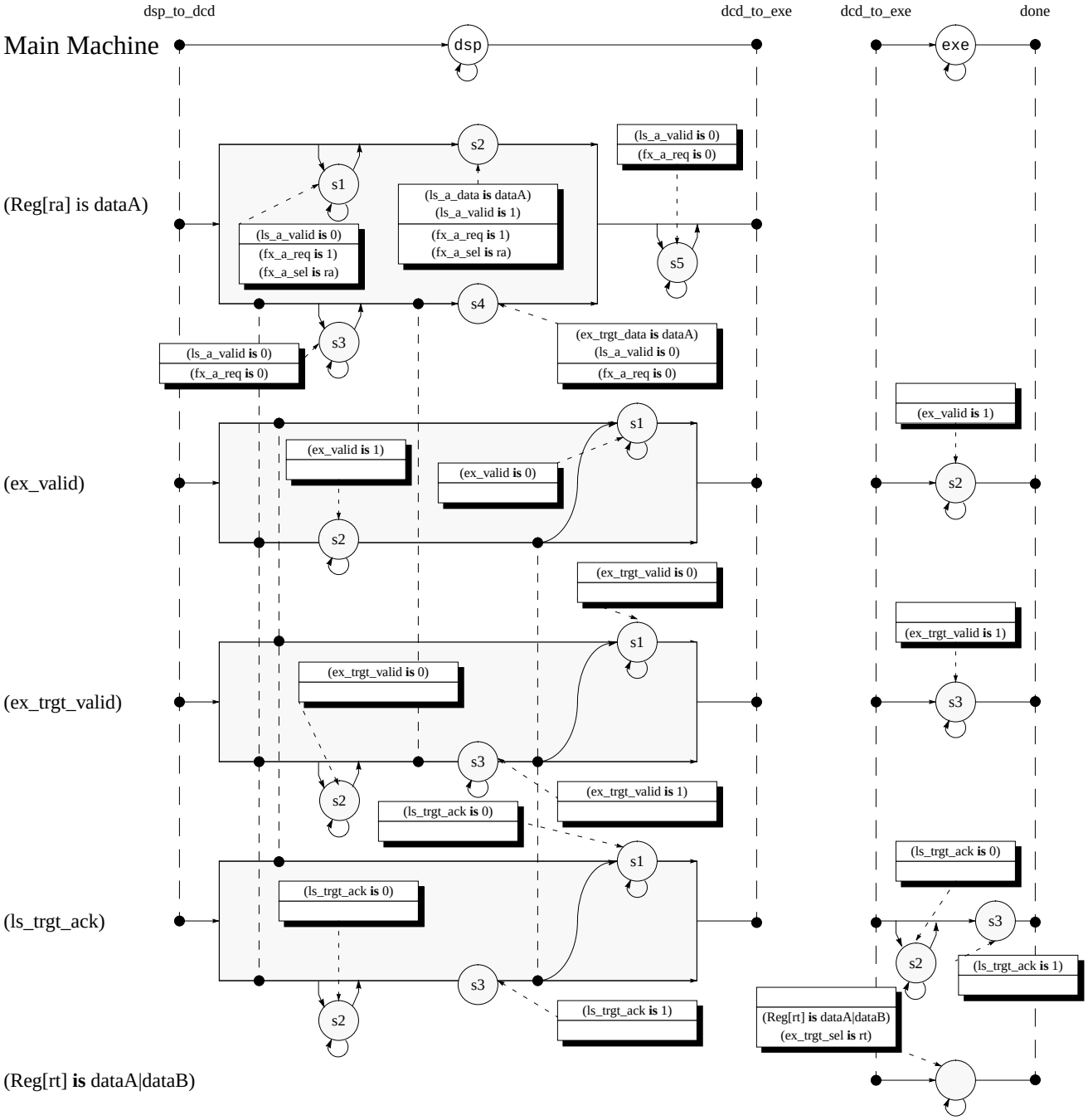

Figure 5. A source operand data mapping and target data mapping

showing all possible interactions that can occur. Also, the decode stage of the graph is significantly more complex than the other two pipeline stages. The dispatch and execute stages require only an acknowledgment in order to complete the pipeline stage. The decode stage must attain multiple resources to advance. The set of states in which the circuit can enter the dispatch stage and the many orderings in which resources can be obtained account for the complexity of the decode stage portion of the graph.

Also, in the dispatch stage the graph fans out into four paths. Path 1 and path 4 correspond to the case where one of the source operands is being bypassed. Path 2, the path where neither operand is being bypassed, is the most complex path. This is because all the resources that need to be obtained in the decode stage are completely independent. Path 3, the path where both operands are being bypassed, is the simplest path. On this path the arrival of the A and B operands coincide with the signal `ex_trgt_valid`.

Another observation that can be made is that path 2 actually consists of multiple paths through the dispatch stage. This is because when both operands are not being bypassed, then the instruction is allowed to stall in the dispatch stage. The remaining three paths do not stall in the dispatch stage because a condition for bypassing to occur is that the two instructions involved must be dispatched in consecutive cycles.

5.2. Symbolic Trajectory Evaluation

The trajectory assertion in Figure 6 is very complex and has in

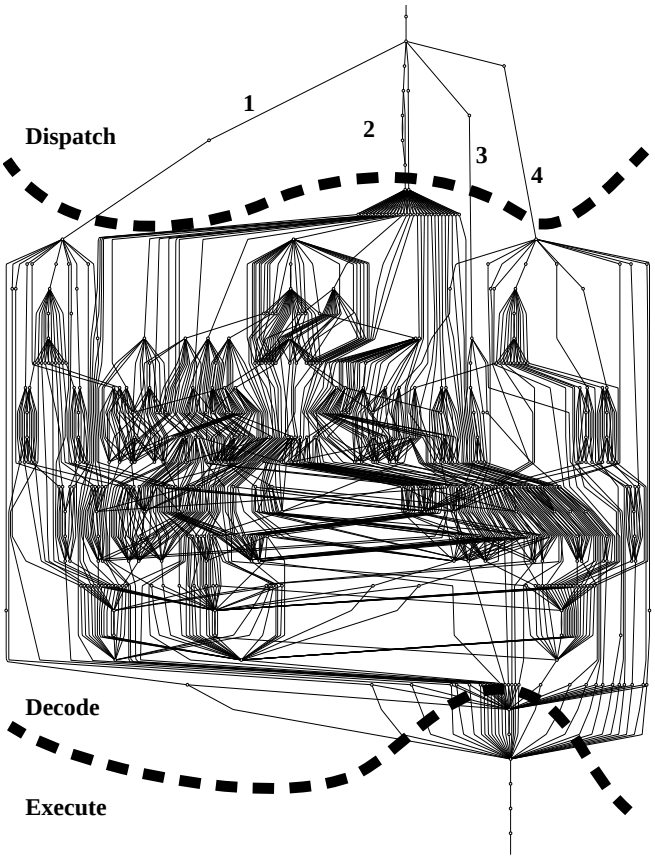

Figure 6. Trajectory assertion for the OR instruction

excess of 28000 paths in the corresponding acyclic component graph. It would be computationally infeasible to enumerate all the paths and use STE separately on each path. Instead the paths were encoded using 456 path variables and each of the states needs to be simulated only once. STE was used to verify the entire trajectory assertion in a single verification run. Cycles in the trajectory assertion were dealt by recursively identifying the strongly connected components in the graph and performing a greatest fixed point computation to deal with the strongly connected components[4]. The verification of the OR assertion took nearly 50 hours of CPU time and 185 MBytes of memory on an IBM RS/600 43P Model 140. This may seem to be a considerable amount of time and memory, but taking into account the enormous number of paths that are being simulated makes the required amount of resources appear more reasonable. In fact, on average the simulator is able to verify 10 paths a second. Verification of the complete trajectory assertion would most likely be run as regression tests. During the development of the assertions, implementation mapping, and the actual design, the simulator can be run interactively to debug each of these components by focusing the verification on problematic paths.

Additionally, it is not necessary to have a one-to-one relationship between instructions and assertions. Instead, a single assertion could be used to specify a class of instructions with the same instruction format. This would significantly reduce the number of assertions to be verified while not significantly increasing the amount of symbolic manipulation or CPU time.

6. Conclusion

We have shown the application of our methodology for the verification of the fixed point unit of a PowerPC processor. The specification was kept abstract at the level of the instruction set architecture. A separate implementation mapping provided the complex implementation specific details for the FXU. STE was used to verify that the FXU subsystem correctly fulfilled the abstract specification of the processor. In some sense, the mapping merely served as hints to guide the verification task.

At first glance the mapping might seem to be too complex. However, note the fact that most of the complexity in the mapping is in defining the environment around the FXU. And the reality is, modern systems are designed as a set of reactive subsystems with complex interfaces and protocols. Therefore any technique for formal verification of subsystems would have to deal with the same level of complexity. The ultimate aim of this project is the verification of the entire system i.e., the PowerPC processor. However the current set of methodology and tools cannot deal with the level of complexity of an entire processor. Our initial focus is to verify each subsystem i.e., each functional unit (FXU, LSU, BPU) separately and then reason about the interactions amongst the subsystems. We feel that our work on the FXU is a significant step in that direction.

References

[1] R. E. Bryant, D. L. Beatty and C. J. H. Seger, “Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation,” *28th Design Automation Conference*, pp. 397-402, June 1991.

[2] D. L. Beatty and Randal E. Bryant, “Formally Verifying a Microprocessor Using a Simulation Methodology,” *31st Design Automation Conference*, pp. 596-602, June 1994.

[3] C. J. H Seger and R. E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories,” *Formal Methods in System Design 6*, pp. 147-189, 1994.

[4] A. Jain, K. Nelson and R. E. Bryant, "Verifying Nondeterministic Implementations of Deterministic Systems," *Formal Methods in CAD*, November 1996.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill, “Sequential Circuit Verification Using Symbolic Model Checking,” *27th Design Automation Conference*, pp. 46-51, June 1990.

[6] K. L. McMillan, “Symbolic Model Checking,” *Kluwer Academic Publishers*, 1993.

[7] R. P. Kurshan, “Analysis of Discrete Event Coordination,” *Lecture Notes in Computer Science 430*, pp. 414-453, 1990.

[8] R. P. Kurshan, “Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach,” *Princeton University Press*, 1994.

[9] I. Beer, S. Ben David, C. Eisner and A. Landver, “RuleBase: an industry oriented formal verification tool,” *33th Design Automation Conference*, pp. 655-660, June 1996.

[10] E. M. Clarke and R. P. Kurshan, “Computer-aided verification,” *IEEE Spectrum*, pp. 61-67, June 1996.

[11] B. Plessier and C. Pixley, “Formal verification of a commercial serial bus interface,” *14th Annual International Phoenix Conference on Computers and Communications*, pp. 378-382, March 1995.

[12] C. May, E. Silha, R. Simpson and H. Warren, “The PowerPC Architecture: A Specification for a New Family of RISC Processors,” Morgan Kaufmann Publishers, 1994.

[13] F. G. Soltis, “Inside the AS/400,” Duke Press, 1996.