Validation of Pipelined Processor Designs using Esterel Tools: A Case Study* (Extended Abstract)

S. Ramesh[1] and Purandar Bhaduri[2] **

[1] Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Powai, Mumbai 400 076, INDIA
Email: ramesh@cse.iitb.ernet.in
[2] Applied Technology Group, Tata Infotech Ltd
Seepz, Andheri (E), Mumbai 400 096, INDIA
Email: purandar.bhaduri@tatainfotech.com

Abstract. The design of control units of modern processors is quite complex due to many speed-up techniques like pipelining and out-of-order execution. The existing approaches to formal verification of processor designs are applicable to very high level descriptions that ignore timing details of control signals. In this paper, we propose an approach for verification of detailed design of processors. Our approach suggests the use of Esterel language which has rich constructs for succinct and modular description of control. The Esterel simulation tool Xes and verification tools Xeve and FcTools can be used effectively to catch minor bugs as well as subtle timing errors. As an illustration, we have developed an Esterel implementation of DLX pipeline control and verified certain crucial properties.

1 Introduction

Modern processors employ many techniques like pipelining, branch prediction and out-of-order execution to enhance their performance. The design and validation of these processors, especially their control circuitry, is a challenging task [6, 7].

Formal verification techniques, emerging as a viable approach to validation [10], are still inadequate in verification of large systems like processors. Recently many new techniques have been proposed specifically for processor verification [1, 7, 6, 9, 11]. These techniques verify that the given implementation is equivalent to a simpler sequential model of execution, as described by the instruction set architecture. But in these approaches, the implementation is at

* Partial support for this work came from the Indo-US Project titled Programming Dynamical Real-time systems and Tata Infotech Research Laboratory, IIT Bombay.
** This author's current address: TRDDC, 54 B, Hadapsar Industrial Estate, Pune 411 013, INDIA. Email: pbhaduri@pune.tcs.co.in

a very high level of abstraction ignoring details of finer timing constraints on control signals. These details are to be introduced to arrive at the final implementation that can be realized in hardware. Even if the design at the higher level of abstraction is proved to be equivalent to a sequential model, later refinements may introduce timing errors.

The aim of this paper is to propose a verification method for detailed processor implementations containing timing constraints of control signals. We suggest the use of Esterel language [3, 2] and its associated verification tools for describing the implementations and verifying their properties. Esterel has a number of attractive features that come in handy for our purpose. It provides a nice separation between data and control. It offers a rich set of high level constructs, like preemption, interrupts and synchronous parallelism, that are natural for hardware systems and that enable modular and succinct description of complex controllers. Besides simulation, Esterel descriptions can be rigorously verified using the tools Xeve [4] and FcTools [5]. Finally, Esterel programs can be directly translated into hardware.

In this paper we illustrate our approach by developing an Esterel model of the DLX pipelined processor control unit [8]. The model has been debugged using the simulator tool Xes and has been verified to satisfy a number of desired properties using the verification tools.

2 Esterel Specification of Pipelined Control Unit

The specification is based upon the informal description of DLX processor given in [8]. We confine ourselves to the control unit specification; the data path specification can be trivially given using a host language like C.

2.1 The Main Controller

The execution of an instruction in the DLX processor goes through five stages: *Instruction Fetch (IF)*, *Instruction Decode/Register Fetch (ID)*, *Execution/Effective Address Calculation (EX)*, *Memory Access/Branch Completion (MEM)* and *Write-Back (WB)*. The introduction of pipelining leads to increased complexity in design in terms of additional registers and control logic due to various *hazards*. *Pipeline registers* are required to store the intermediate values produced by different stages. DLX uses the *branch-not-taken* prediction scheme and hence to handle the control hazard that occurs when a branch is taken (determined in the EX stage), the instruction in the ID stage must be squashed; the handling of interrupts requires even more complex control logic. Appropriate actions like data forwarding or *stalling* have to be taken to handle *data hazards*, for instance when an instruction updates a register or memory location that is read by a subsequent instruction.

Figure 1 gives an Esterel module that models a generic pipeline stage of the DLX controller. An Esterel program in general consists of one or more modules. Each module has an input-output interface and reactive code that is executed

```
module XXUnit:
input GoPrev, Stall, Restart;
output GoNext, StallPrev, RestartPrev;

loop  % execute the 'loop' body repeatedly
 do % the 'body' of the 'do-watching' statement starts here
  signal Go in  %Go is a local signal
    [
      suspend  % stop execution
      [
       loop
         await immediate Go; %wait till the other component emits 'Go'
         emit GoNext;   % generate the signal GoNext
         run XX; % execute the module named XX
         await tick % wait for one reaction
       end loop
      ||
       loop
         await immediate GoPrev; %wait till 'GoPrev' is present in the input
         await tick; % wait one reaction step
         emit Go % generate 'Go' signal
       end loop
      ]
      when immediate Stall  % stop execution of the 'suspend' body when 'Stall'
                            % is present
    ||
     loop
       await  tick;
       await immediate Stall;
       emit StallPrev
     end
    ]
  end signal % end of scope of local signal declaration
 watching Restart; % abort the 'watching' body when 'Restart' is present
 emit RestartPrev
end loop % end of the outermost loop

end module % end of the module
```

Fig. 1. A pipeline stage in Esterel

periodically at the phase of the built-in signal `tick`. Every time a module is executed, it reads input signals and depending upon the state of the module generates appropriate output signals and changes the state. Every such execution is called a *reaction*. A reaction is assumed to be instantaneous so that there is no time delay between input consumption and output generation. All Esterel statements are instantaneous excepting the 'halt' statement which does not terminate at all. The control of an Esterel program resides at one or more halt statements (more than one when there are concurrent components) which decide the state of the program. A reaction, besides generating outputs, results in a change of state with the movement of control points from one set of halt statements to another.

Esterel possesses a rich set of constructs for describing control. Here we give a very brief explanation of some of these constructs. The statement `await S` is a simple 'wait construct' that delays termination until the signal `S` is present in the input; `await immediate S` is a variant which can terminate even in the very first instant when control reaches the construct. The statement `do watching stat S` continues to execute `stat` as long as the signal `S` is not present; the moment `S` appears on the input, the whole statement terminates aborting the computation inside `stat`. The statement `suspend stat till S` suspends the execution of `stat` in all reactions in which `S` is present; execution continues where it got suspended when `S` is not present.

Now we will describe the behavior of the module in Figure 1. For the sake of simplicity, we have taken the `tick` signal to define the clock of the processor. Suppose the signals `Stall` and `Restart` are not present in a reaction, corresponding to the uninterrupted flow of an instruction through the pipeline stages. Then the submodule `XX` (in the first branch of the parallel operator within the `suspend` statement) is executed in the cycle when the local signal `Go` is present; the `Go` signal is present in this cycle provided the `GoPrev` signal was present in the previous cycle (in the second branch of the parallel operator within the `suspend` statement). At the end of execution of `XX`, which is assumed to be instantaneous, the module generates `GoNext`.

Suppose that `Stall` is present in a cycle, representing a hazard in the pipeline stage `XX`. Then the execution of `XX` is suspended by the `suspend` statement and the signal `GoNext` is not generated; the signal `StallPrev` is generated (in the second branch of the outer parallel operator). If on the other hand the `Restart` signal is present, representing an interrupt or a taken branch, then the body of the outer watchdog primitive is killed and the execution is restarted because of the presence of the outer loop construct. This results in the loss of information about the presence of the `GoPrev` signal in the previous cycle. Also a `Restart` triggers a `RestartPrev` signal.

Thus, `XXUnit` executes the submodule `XX` in every cycle in which `GoPrev` is present and generates `GoNext`, as long as `Stall` or `Restart` are not present. A `Stall` in a cycle suspends the execution of `XX` while a `Restart` restarts the execution of whole module afresh resetting its internal state, i.e., it squashes the execution of `XX`.

```
module CONTROL:
input IssueNextInstr;
output InstrCompleted;
output WritePCn : integer,WritePCb: integer;
inputoutput Restart0, RestartIF, RestartID,
            RestartEX, RestartMEM, RestartWB;
inputoutput Stall0, StallIF, StallID, StallEX, StallMEM, StallWB;

signal GoIF, GoID, GoEX, GoMEM
in

  [
   run IFUnit [ signal  IssueNextInstr / GoPrev, GoIF / GoNext,
                        StallIF / Stall, Stall0 / StallPrev,
                        RestartIF /  Restart, Restart0 /  RestartPrev]
      ||
   run IDUnit [ signal  GoIF / GoPrev, GoID / GoNext,
                        StallID / Stall, StallIF / StallPrev,
                        RestartID /  Restart, RestartIF /  RestartPrev]
      ||
   run EXUnit [ signal  GoID / GoPrev, GoEX / GoNext,
                        StallEX / Stall, StallID / StallPrev,
                        RestartEX /  Restart, RestartID /  RestartPrev]
      ||
   run MEMUnit [ signal GoEX / GoPrev, GoMEM / GoNext,
                        StallMEM / Stall, StallEX / StallPrev,
                        RestartMEM /  Restart, RestartEX /  RestartPrev]
      ||
   run WBUnit [ signal  GoMEM / GoPrev, InstrCompleted / GoNext,
                        StallWB / Stall, StallMEM / StallPrev,
                        RestartWB /  Restart, RestartMEM /  RestartPrev]
   end signal

end module
```

Fig. 2. The control unit for the DLX pipeline stages

The Esterel module in Figure 2 models the behavior of the entire pipeline controller. Each pipeline stage is an instantiation of the generic module `XXUnit` given in Figure 1; for example, `IFUnit` is obtained from `XXUnit` by replacing the command `run XX` by `run IF` where the module `IF`, shown in Figure 3, describes the behavior of the instruction fetch stage.

In the module `CONTROL`, the renaming of the `Go`, `Stall` and `Restart` signals leads to the establishment of a forward `Go`-chain and two reverse `Stall` and `Restart`-chains. When there is no `Stall` signal (none of `StallIF`,$\cdots$,`StallWB` is present), the input `IssueNextInstr` signal triggers the execution of the five stages, with the execution of each stage in a cycle triggering via the `Go`-chain the execution of the next stage in the next cycle. When `StallXX` is present, it stalls the pipeline up to stage `XX`; this is achieved by the instantaneous transmission of the various `Stall` signals to the preceding stages via the `Stall`-chain. The succeeding stages are not affected by this stall. Similarly, a `Restart` signal triggers the restart of all the earlier stages up to the current stage using the `Restart`-chain.

2.2 The Pipeline Stages

The Esterel specification of the various pipe stages which instantiate `XX` in Figure 1 can now be described. Because of space constraints, we describe only the IF and EX stages.

```
module IF:
input ReadPC : integer, BranchTaken;
output WritePCn: integer, IfOut : integer;
function FetchInstr (integer) : integer;
function IncrPC (integer) : integer;

  emit IfOut(FetchInstr(?ReadPC));
  present BranchTaken
  else
    emit WritePCn(IncrPC(?ReadPC))
  end present;
end module
```

Fig. 3. IF Stage

The module `IF` in Figure 3 emits a signal `IfOut` with a value representing the current instruction and a signal `WritePCn` whose value indicates the new value of PC. The signal `BranchTaken` indicates a taken branch, and the `IF` stage writes a PC value only if this signal is absent, indicating a normal flow of execution. If the `BranchTaken` signal is present the PC value is written by the `EX` stage, shown in Figure 4, through a signal called `WritePCb` to indicate a branch in instruction

execution. The external functions `FetchInstr` and `IncrPC` abstract the actions corresponding to fetching an instruction and incrementing the PC.

```
module EX:
input BranchTaken, Bypass, MemInAdr:integer, MemInVal : integer,
      ExInOpcode : integer, ExInOpnd : integer;
output ExOutAdr : integer, ExOutVal : integer, WritePCb:integer;
function AluOpAdr (integer, integer) : integer;
function AluOpVal (integer, integer) : integer;

  present Bypass then
    emit ExOutAdr(AluOpAdr(?ExInOpcode, ?MemInVal));
    emit ExOutVal(AluOpVal(?ExInOpcode, ?MemInVal))
  else
    emit ExOutAdr(AluOpAdr(?ExInOpcode, ?ExInOpnd));
    emit ExOutVal(AluOpVal(?ExInOpcode, ?ExInOpnd))
  end present;

  present BranchTaken then
    emit WritePCb(AluOpAdr(?ExInOpcode, ?ExInOpnd))
  end present
end module
```

Fig. 4. EX Stage

The module `EX` in Figure 4 emits two signals `ExOutAdr` and `ExOutVal`, corresponding to the address and value computed by the ALU by operations abstracted by the external functions `AluOpAdr` and `AluOpVal`. The presence of the input signal `Bypass` indicates that there is a data hazard and hence that the inputs to ALU are to be taken through a forwarding process from the output of the EX/MEM pipe stage; in the absence of this signal, the inputs come from the ID/EX pipe stage. The `BranchTaken` signal indicates a taken branch and triggers the signal `WritePCb` which writes the new branch address into PC.

The above Esterel model of the DLX processor has abstracted away details about the data path, instruction decoding, alternative actions based on various types of instructions (such as load/store) and hazard detection. This is the reason that the signals `Bypass`, `Restart`, `BranchTaken` and `Stall` have been modeled as external input signals, rather than being generated internally (by hazard detection units).

3 Validation using Esterel tools

In this section we outline the validation of the design of the DLX processor control unit using the Esterel simulation tool Xes and verification tools Xeve and FcTools. We focus on the micro-properties of the control unit, such as smooth

flow of instructions through the pipeline, absence of deadlock, proper issuing of stall and restart instructions, and correct behavior of the pipeline with respect to these signals. We are able to verify that for example, in case of a taken branch (determined in the EX stage) the instruction following the branch (in its ID stage) is restarted or aborted. Similarly, we can verify that a stall signal sent to some stage propagates as a bubble through the pipeline.

The properties verified by us are finer than the macro-property verified in [7], namely that the pipelined machine has the same effect on visible state as the sequential one for the same input. The latter property, in its full glory, cannot be verified using existing Esterel tools because they deal with only control states. However, the property restricted to control states is still verifiable (see the paragraph titled Stall in Section 3.1).

3.1 Verification

The simple properties of the DLX pipeline controller mentioned above can be verified using the Esterel tools Xeve [4] and FcTools [5]. They are verification environments for Esterel programs modeled as *finite state machines* (FSMs) with a user-friendly graphical interface.

The Esterel compiler generates FSMs implicitly in the form of boolean equations with latches. One of the verification tasks performed by Xeve is to take an implicit FSM and perform a state minimization using the notion of bisimulation equivalence. Before minimization a set of input /output signals can be hidden. This results in a nondeterministic FSM where some transitions may be labeled by τ, a hidden internal action. Xeve generates minimized FSMs, that can be further reduced using some abstraction criterion by FcTools and can be graphically explored using the tool ATG.

FcTools is a verification tool set for networks of communicating FSMs. Its capabilities include graphical depiction of automata, reduction of automata and verification of simple modal properties by observers, counterexample production and visualization.

In our verification process the original FSM produced by Xeve had about 1500 states, which after making some irrelevant interface signals local got reduced to 543 reachable states. This was reduced to 16 states and 72 transitions after applying the observational equivalence minimization procedure available in FcTools. Still the automaton could not be inspected due to the large number of transitions. So we used the powerful abstraction technique available in FcTools to further reduce the size of the automaton. An *abstraction criterion* defines a new set of action symbols that are regular expressions on the action symbols in the original automaton. The reduction involves abstraction of sequences of old actions into new actions so that the reduced automaton contains only new action symbols; further, certain paths in the original automaton are eliminated, thereby resulting in a small automaton that can be checked easily.

Depending upon the property to be checked, we applied different criteria to get small automata which we verified with respect to appropriate properties.

Criterion	States	Transitions
Initial	16	72
Smooth Flow	8	12
Stall	16	32
Branch	1	1

Table 1. Sizes of Reduced Automata

Table 1 summarizes the sizes of the various reduced automata obtained for different criteria. The details about the criteria 'Smooth Flow' and 'Stall' are given below. The criterion 'Branch' checks for proper updation of the PC value at any cycle by abstracting paths into two abstract actions 'success' and 'failure'. The reduced automaton has only one transition with the label 'success'.

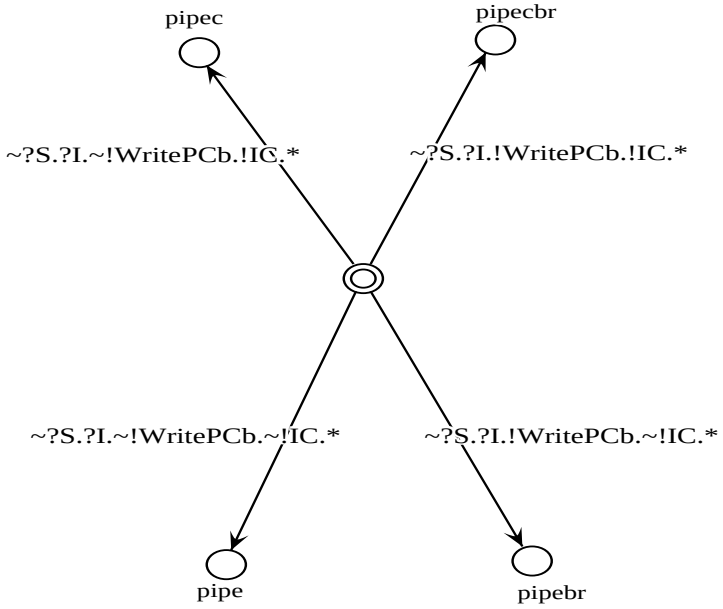

Fig. 5. Abstraction Criterion for Smooth Flow

Smooth flow of instructions This criterion verifies that every instruction issued is completed after four cycles in the absence of stalls and branches. The criterion depicted in Figure 5, defines four abstract actions `pipe`, `pipec`, `pipebr` and `pipecbr` which rename the edges satisfying the corresponding regular expressions, eg., `pipebr` renames any edge in which a branch has been taken and no instruction is completed; in the regular expressions, `.` denotes synchronous product of input and output events (prefixed by `?` and `!` respectively) and their

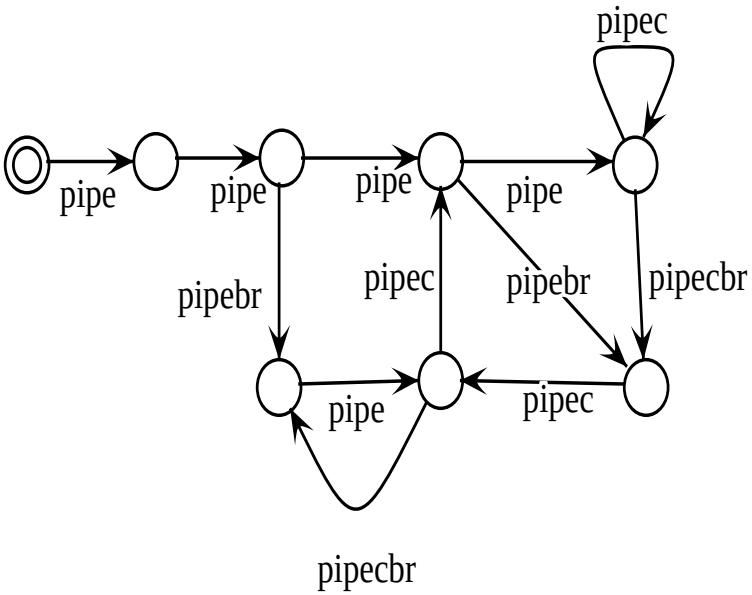

Fig. 6. Reduced Automaton for Smooth Flow

negations (prefixed by ˜); the event * matches any event. Figure 6 gives the reduced automaton which can be verified with respect to the desired property by inspection.

For the sake of clarity in the figures, the signals `StallIF`, `IssueNextInstr`, and `InstrCompleted` of the original automaton are renamed as `S`, `I` and `IC` respectively; further the `WritePCb` signal is treated as being synonymous with `BranchTaken` for technical reasons.

Stall The property verified here is that the `StallIF` signal stalls the `IF` stage for a cycle: no instruction is completed four cycles after a `StallIF` assuming later stages are not stalled or squashed in the intervening period. The abstraction criterion for this is shown in Figure 7 and the reduced automaton in Figure 8. In the reduced automaton there is no path of length five starting with a `stall` or a `stallc` that ends with a `ic` or `stallc` edge. Another interesting thing to note from this automaton is that from every state there is a sequence of 'stalls' that leads to the initial state; this property corresponds to the sequential equivalence property of [7] for control states.

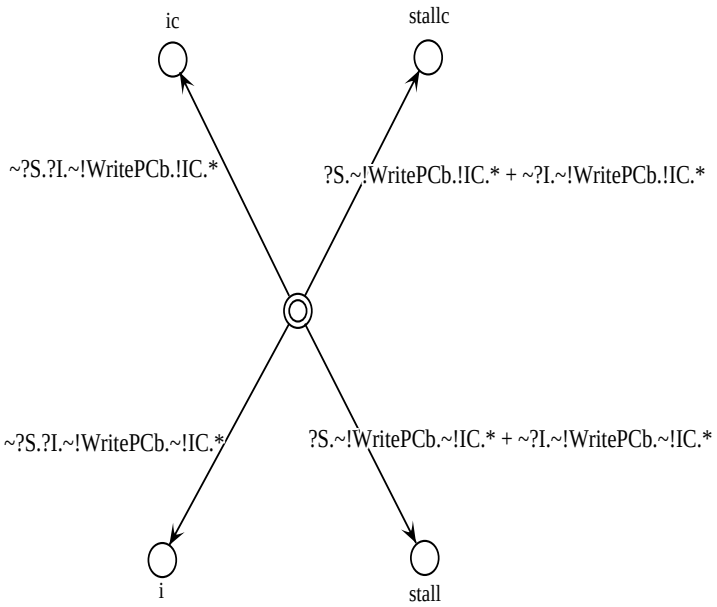

Fig. 7. Abstraction Criterion for Stall

4 Conclusion

We have proposed the use of Esterel language and tools for verification of modern processors. Esterel can be used to describe, in sufficient detail and in a modular and succinct way, control units of processors using its rich set of constructs. Complex timing properties of Esterel descriptions can be verified using powerful tools.

We have illustrated the use of Esterel tools for the description of DLX processor. The initial results are encouraging. The verification tools Xes, Xeve and FcTools were found to be quite useful in detecting anomalies ranging from simple bugs to complex timing errors. We plan to extend our investigation to more complex processors involving superscalar features like out-of-order executions. We also plan to investigate, in greater detail, the relative merits of Esterel for describing control units of processors with respect to the traditional HDLs.

References

1. S. Berezin, A. Biere, Ed. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out of order processor verification. In G. Gopalakrishnan and P. Windley, editors, *FMCAD'98, LNCS 1522*. Springer Verlag, 1998.

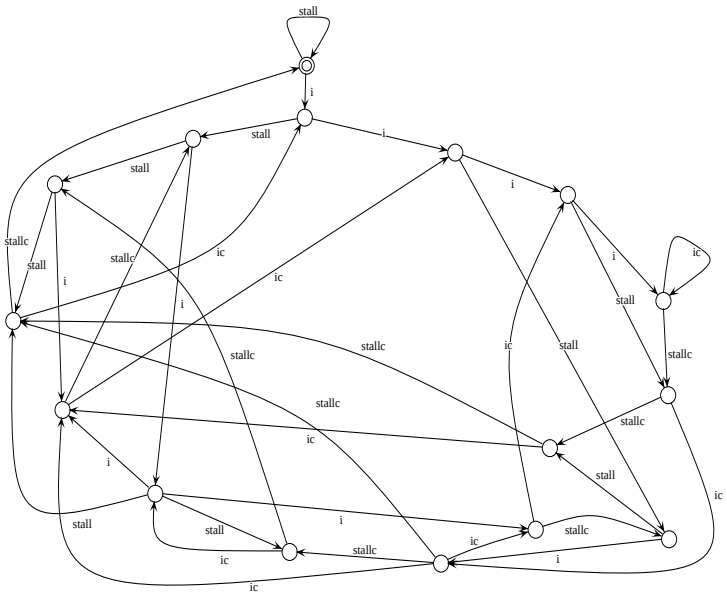

Fig. 8. Reduced Automaton for Stall

2. G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
3. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2), 1992.
4. A. Bouali. XEVE: An Esterel Verification Environment. Available by ftp from `ftp-sop.inria.fr` as file `/meije/verif/xeve-doc.ps.gz`.
5. A. Bouali, A. Ressouche, R. de Simone, and V. Roy. The FcTools User Manual. Available by ftp from `ftp-sop.inria.fr` as file `/meije/verif/fc2userman.ps`.
6. R. E. Bryant. Formal Verification of Pipelined Processors. In *Proc. TACAS 98, LNCS 1384*. Springer Verlag, March-April 1998.
7. J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proc. CAV'94, LNCS 818*. Springer Verlag, June 1994.
8. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufman Publishers Inc., 1995.
9. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the Proof of Correctness of Pipelined Microprocessors. In *Proc. CAV'98, LNCS 1427*. Springer Verlag, June/July 1998.
10. T. Kropf, editor. *Formal Hardware Verification, LNCS 1287*. Springer Verlag, 1997.
11. J. U. Skakkebaek, R. B. Jones, and D. L. Dill. Formal Verification of Out-of-order Execution using Incremental Flushing. In *Proc. CAV'98, LNCS 1427*. Springer Verlag, June 1998.