Automated Pipeline Design

Daniel Kroening*
Computer Science Dept., University of Saarland
Im Stadtwald, B45
D-66041 Saarbruecken, Germany
kroening@cs.uni-sb.de

Wolfgang J. Paul
Computer Science Dept., University of Saarland
Im Stadtwald, B45
D-66041 Saarbruecken, Germany
wjp@cs.uni-sb.de

ABSTRACT

The interlock and forwarding logic is considered the tricky part of a fully-featured pipelined microprocessor and especially debugging these parts delays the hardware design process considerably. It is therefore desirable to automate the design of both interlock and forwarding logic. The hardware design engineer begins with a sequential implementation without any interlock and forwarding logic. A tool then adds the forwarding and interlock logic required for pipelining. This paper describes the algorithm for such a tool and the correctness is formally verified. We use a standard DLX RISC processor as an example.

Keywords

Pipeline, Forwarding, Interlock

1. INTRODUCTION

1.1 Pipeline Design

Besides the mainstream microprocessors, there is a large market for custom designs which incorporate smaller RISC processor cores. These designs used to be simple sequential processors with low complexity, which are easy to verify. Nowadays, customers require cores with higher performance, which requires a pipelined design. Assuming a sequential design is given, the engineers' job is to transform this design into a pipelined machine while maintaining full binary compatibility.

This task is described in many standard textbooks on computer architecture such as [21, 10] or [9]. The task consists of the following four steps:

1) The hardware is partitioned into pipeline stages,
2) structural hazards are resolved by duplicating components where possible,
3) in order to resolve the data hazards, forwarding (bypassing) logic has to be added,
4) interlock hardware has to be added where ever a structural hazard is left in the design or forwarding might fail.

In the open literature, steps 3) and 4), i.e., adding forwarding and interlock hardware, is usually considered the tricky and error prone part. This paper addresses how to automate the steps 3) and 4). We therefore assume that steps 1) and 2) are already done manually, i.e., we take a sequential machine which already has a pipeline structure. This machine is called *prepared sequential* machine [20].

We describe the algorithm used in order to do the transformation of the prepared sequential machine into a pipelined machine. Furthermore, we formally verified the correctness of that transformation using the theorem proving system PVS [8].

We think that critical designs should be a four-tuple: 1) the design itself, 2) a specification, 3) a human-readable proof, and 4) a machine-verified proof. Moreover, we think that there will be a considerable market for such four-tuples. However, the time required to manually write these proofs usually discourages vendors.

In addition to the forwarding and interlock hardware, our tool therefore also generates a proof of correctness for the new hardware. Assuming the correctness of the original sequential design, we conclude that the pipelined machine with forwarding and interlock is correct. The method is limited to in-order designs, out-of-order designs are not covered. The DLX RISC processor [10] serves as an example.

1.2 Related Work

The concept of prepared sequential machines is taken from [20]. Furthermore, [20] describes a manual transformation of a prepared sequential DLX into a pipelined DLX. In [15], Levitt and Olukotun verify pipelined machines by "deconstructing" the pipeline, i.e., by reverting the transformation.

There is much literature on verifying processors with formal methods. Using model-checking [6, 7], one achieves an impressive amount of automatization but one suffers from the state space explosion problem. This is addressed by BDDs (binary decision diagrams) [4, 17]. Velev and Bryant [24] verify a dual-issue in-order DLX with speculation by automating the Burch and Dill pipeline flushing approach. The function units are abstracted by means of uninterpreted functions.

Theorem proving systems such as HOL [5], PVS [8], or ACL2 [13] do not suffer from the state space explosion problem. There has been much success in verifying complete, complex systems using theorem provers [2, 11, 22]. However, theorem proving systems involve much manual work. Recently, Clarke [3], McMillan [18, 19], and Dill et.al. [1] apply classical theorem proving techniques for model- and equivalence-checking.

2. THE SEQUENTIAL MACHINE

*Supported by the DFG graduate program "Effizienz und Komplexität von Algorithmen und Rechenanlagen"

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

We start with a sequential implementation of the design. This design is supposed to be partitioned into stages already. Let n denote the number of stages the design has. A microprocessor design consists of both registers and the (combinatorial) circuits between them.

Each register is assigned to a stage. By convention, a register R is assigned to the stage $k \in \{0,\ldots,n-1\}$ that writes R. Let $R \in out(k)$ denote that R is an output register of stage k.

During step 1) as described above, one introduces instances of specific registers in multiple stages. Thus, let $R.k$ denote the instance of register R written by stage $k-1$. For example, a pipelined microprocessor might have instruction registers in stages two and three which are denoted by IR.2 and IR.3.

Each register has a given domain, i.e., the set of values the register might have. Let $W(R)$ denote this set. In analogy to $out(k)$, let $in(k)$ denote the set of registers a stage takes as inputs. For example, the first stage might want to read the value of the PC (program counter) register, thus $PC \in in(0)$.

The designer is expected to provide a list of the names of the registers, their domain, and the stages they belong to. It is left to specify the data paths of the machine. Let $R_1,\ldots,R_j$ denote the list of inputs registers of stage k. Let $R'_1,\ldots,R'_l$ denote the list of output registers of stage k.

The (combinatorial) data paths of stage k are now modeled as mapping from the set of input values to the set of output values:

$$f_k: \quad W(R_1)\times\ldots\times W(R_j) \longrightarrow W(R'_1)\times\ldots\times W(R'_l)$$

For example, this includes circuits such as the ALU. In addition to that, let

$$f_kRwe: \quad W(R_1)\times\ldots\times W(R_j) \longrightarrow \{0,1\}$$

denote a write enable signal of register $R \in out(k)$. Let ue_k denote the *update enable* signal of stage k. If ue_k is active, the output registers of stage k are updated. The value clocked into a register depends on whether an instance of R is also in the previous stage or not.

- If so, the new value is the value provided by f_k if f_kRwe is active and is provided by the previous stage if the write enable signal is not active. The clock enable signal for such a register is just ue_k.
- If not so, the new value is always provided by f_k. The clock enable signal ce of the register is active iff both the write enable and update enable signals are active:

$$ce = f_kRwe \wedge ue_k$$

If R is part of a register file (e.g., general purpose register file), one needs three signals in order to model the interface to the register file. The function f_k provides the data value (Din), the function f_kRwe the write enable input. Let $\alpha(R)$ be the number of address bits the register file takes and

$$W_a(R) = \{0,1\}^{\alpha(R)}$$

the set of addresses the register file takes. Furthermore, let

$$f_kRwa: \quad W(R_1)\times\ldots\times W(R_j) \longrightarrow W_a(R)$$

denote the signal which is fed into the register file as address for writing data. This is illustrated in figure 1.

In case of a read access to a register file (e.g., operand fetching in decode stage), let

$$f_kRra: \quad W(R_1)\times\ldots\times W(R_j) \longrightarrow W_a(R)$$

denote the signal which is fed into the register file as address.

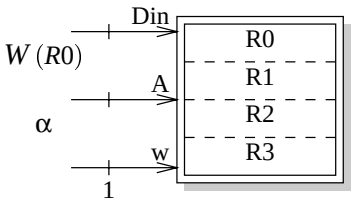

Figure 1: Signals required in order to write into a register file consisting of four registers. In this example, α is two.

Table 1: The sequential scheduling of a three stage pipeline in the absence of stalls

cycle	0	1	2	3	4	5	6...
ue_0^T	1	0	0	1	0	0	1
ue_1^T	0	1	0	0	1	0	0
ue_2^T	0	0	1	0	0	1	0

The signals f_kRwe and f_kRwa are precomputed, as described in [21]. Let $Rwe.j$ and $Rwa.j$ denote the precomputed versions of these signals in stage j.

In addition to the register list, the transformation tool takes the functions (i.e., the signal names in HDL) as described above as inputs.

The circuits that provide the inputs to the functions f_kR and so on are modeled by the input generation function g_k. In case of the prepared sequential machine the function just passes the appropriate register values and does not model any gates. We will later on modify it in order to introduce the forwarding hardware. Each time we mention a function name such as f_kR or f_kRra, we omit the parameter g_kR of the function for sake of simplicity.

By enabling the update enable signals ue_k round robin (table 1), one gets a sequential machine. In the following, we assume that this sequential machine behaves as desired. It will serve as a reference for the correctness proof. However, there is a vast amount of literature on formally verifying sequential machines, e.g., [16, 25].

3. ADDING A STALL ENGINE

In order to realize interlock, we need means to stall the execution in certain stages while the execution proceeds in the stages below. Thus, as first step of the transformation into a pipelined machine, we add a *stall engine*. A stall engine is a module that takes a set of stall signals for each stage as inputs and provides the update enable signals (latch advance signals) as output. This concept is taken from [20]. For this paper, we take a stall engine described in [12] and extend it by a rollback (squashing) mechanism: for each stage $s \in \{1,\ldots,n-1\}$, a one-bit register $fullb.s$ is added. In addition to that, a signal $full_k$ is defined as follows:

$$\begin{aligned} full_0 &= 1 \\ k \in \{1,\ldots,n-1\}: \quad full_k &= fullb.k \end{aligned}$$

The signal $rollback_k$ indicates that a misspeculation is detected in stage k. We will later on describe how this is done. Using the signal $rollback_k$, a set of signals $rollback'_k$ is defined. The signal $rollback'_k$ is active if the instruction in stage k has to be squashed because of misspeculation.

$$rollback'_k = \bigvee_{i=k}^{n-1} rollback_i$$

The signal $stall_k$ is supposed to be active iff the stage k is to be stalled for any reason. We will define it later on. Using the full signal and the stall signal, the update enable signal is defined. A stage is updated if it is full and not stalled and if there is no rollback:

$$ue_k = full_k \wedge \overline{stall_k} \wedge \overline{rollback_k^l}$$

The full bit registers are initialized with zero and updated as follows: A stage becomes full if it is updated or stalled:

$$\begin{array}{l} s \in \{1,\ldots,n-1\}: \\ \quad fullb.s \;:=\; ue_{s-1} \vee stall_s \end{array}$$

The notation ":=" with a register on the left hand side is used in order to denote the inputs of the register.

The signal $stall_k$ is defined using a signal $dhaz_k$, which indicates that a data hazard occurs in stage k, and using a signal ext_k that indicates the presence of any other external stall condition in the stage, e.g., caused by slow memory. Stage k is stalled if there is a data hazard, or an external stall condition or if stage $k+1$ is stalled:

$$\begin{array}{rcl} k \in \{1,\ldots,n-1\}: & & \\ stall_k & = & (dhaz_k \vee ext_k \vee stall_{k+1}) \wedge full_k \\ stall_{n-1} & = & (dhaz_{n-1} \vee ext_{n-1}) \wedge full_k \end{array}$$

Using this stall engine, we can stall the machine in any arbitrary stage and the other stages keep running if possible. This includes removal of pipeline bubbles if possible.

4. FORWARDING

4.1 Generic Approach

The forwarding logic is added as follows: Let R be an input register of stage k. If an instance of R is either output of stage $k-1$ or stage k, nothing needs to be changed, i.e., no forwarding hardware is required. Assume register R is written by stage w, i.e., $R \in out(w)$. For example, in a five stage DLX as in [10], a GPR register operand is read in stage $k = 1$ (decode) but written by stage $w = 4$ (write back). In this case, one needs to add forwarding logic.

In case of a microprocessor, the result of some instructions are already available in an early stage. For example, in a standard five stage DLX the result of ALU instructions is already available in stage 2 (execute). These results are saved in a register. The transformation tool does not try to determine this register automatically. The designer is expected to name the register responsible for forwarding manually instead. We think that this manual effort is very low. In case of the five stage DLX, one needs to specify two registers, one in the execute stage and one in the memory stage.

The register is called *forwarding register*. Furthermore, we assume that as soon as a value is stored in the register, it is the final value as written into the register which is to be forwarded. Let Q denote the forwarding register for forwarding R.

Using the write enable signals of Q, a valid signal is defined as follows: The input is valid iff the register Q is written in stage k (as indicated by f_kQwe) or in any prior stage. In order to determine if it was written in any prior stage, we pipeline the valid bit by adding one-bit registers $Qv.k$. Thus, the valid signal of stage k is:

$$Q_kvalid \;=\; Qv.k \vee f_kQwe$$

The register $Qv.k$ is updated with:

$$Qv.k := Q_{k-1}valid$$

This allows defining signals $R_khit[j]$, which indicate the stage that the instruction writing the desired value of R is in. The hit signal of a stage is then active iff the stage is full, and the instruction in the stage writes the register that is to be forwarded, and the addresses match. For comparison, the precomputed versions of the write enable signal and of the write address of R, i.e., $Rwe.j$ and $Rwa.j$, are used.

$$\begin{array}{l} \forall j \in \{k+1,\ldots,w-1\}: \\ \quad R_khit[j] \;=\; full_j \wedge Rwe.j \wedge \\ \qquad\qquad (f_kRra = Rwa.j) \end{array}$$

The address comparison is realized with an equality tester. It is omitted if the register R is not part of a register file.

If any hit signal of a stage j is active, let top denote the smallest such j:

$$top \;=\; \min\{j \in \{k+1,\ldots,w\} \mid R_khit[j]\}$$

In hardware, one uses a set of multiplexers for this task. If a hit signal is active, the value from the given stage is taken. Let g_kR denote the input value generated by the forwarding logic in stage k for register R. If top is the stage the register R is output of, i.e., $top = w$, one takes the value present at the input of the register:

$$top = w \;\Longrightarrow\; g_kR = f_wR$$

If the hit is in any other stage, one takes the value which is written into the designated forwarding register Q. Note that the write enable signal of that register might or might not be active. We therefore have to select the appropriate value. If the write enable signal is active, we take the value which is written. If not so, the register was written in an earlier stage already. We take the value from the previous stage therefore.

$$top \neq w \;\Longrightarrow\; g_kR = \begin{cases} f_{top}Q & : \; f_{top}Qwe \\ Q.top & : \; \text{otherwise} \end{cases}$$

If no hit signal is active, the value is taken from the register R. If R is part of a register file, let $a = f_kRra$ denote the address.

$$top \text{ undefined} \;\Longrightarrow\; g_kR = R.(w+1)[a]$$

Data Hazards This fails if a hit is indicated but the value forwarded is not valid yet, as defined above. For example, in case of the five stage DLX this happens if one has to forward the result of a load instruction that is in the execute stage. Thus, a data hazard is signaled in this case by activating $dhaz_k$. In addition to that, we enable $dhaz_k$ if the data hazard signal of stage top is active.

Forwarding not only occurs while fetching operands. Many microprocessors, e.g., MIPS, use one or more delay slots for branch instructions, called delayed branch. Given a sequential implementation of a machine with delayed branch, the pipeline transformation tool automatically generates a pipelined machine with one or more delay slots.

4.2 Case Study

As case study, we applied our tool to a five stage DLX RISC machine. The machine does not feature a floating point unit. The machine uses a branch delay slot and therefore does not need speculation for the instruction fetch. The prepared sequential machine reads the two GPR operands in the decode stage (stage 1). Given that $C.2$ and $C.3$ are used as forwarding registers for *GPR*, the tool generates the forwarding hardware depicted in figure 2. The figure shows the forwarding hardware for one operand (called *GPRa*) only. The interlock hardware is not depicted due to lack of space.

Note that this hardware gets slow with larger pipelines. With larger pipelines, one can use a find first one circuit and a balanced tree of multiplexers or an operand bus with tri-state drivers.

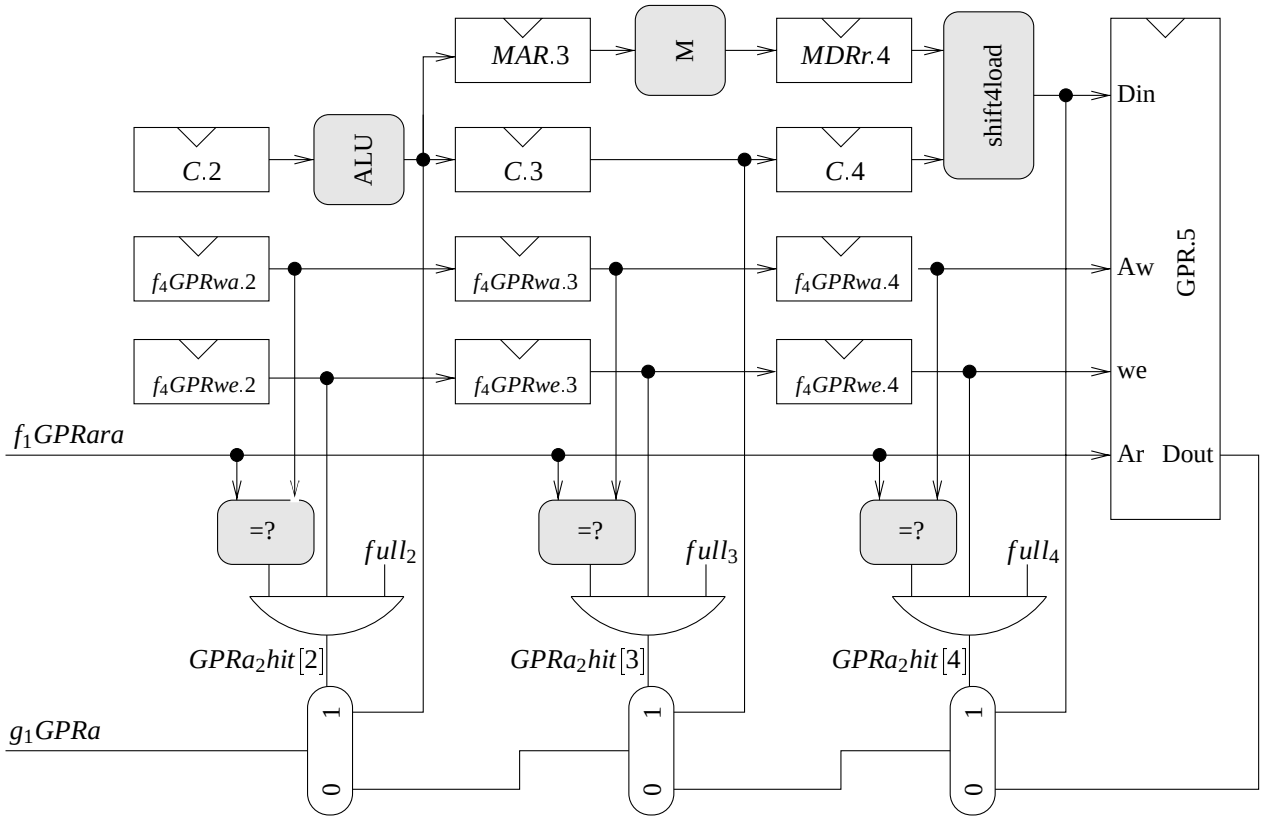

Figure 2: Generated forwarding hardware for the five-stage DLX. Most registers and interlock hardware are omitted.

5. SPECULATION

As described above, the stall engine provides means to evict instructions from the pipe if misspeculation is detected in stage k by enabling the $rollback_k$ signal. The designer is expected to state which input value is speculative and which value is speculated on. The transformation tool adds hardware which compares the guessed value with the actual value as soon as available and enables the rollback signal if the comparison fails. The comparison is done if the stage is full and not stalled in order to ensure that the input operands are valid. In case of a rollback, the correct value is used as input for subsequent calculations. We do not rely on the speculation mechanism to learn from the rollback.

Thus, the guessed value provided by the designer has no influence on the correctness of the design; if the value is always wrong this just slows down the machine. Thus, it is a matter of performance only and not of correctness. We therefore do not have to argue about the value provided by the speculation mechanism.

For example, if one speculates on whether a branch is taken or not taken in stage 0 (instruction fetch), one can implement branch prediction. In addition to that, we implemented precise interrupts in a five stage DLX by speculating that an interrupt does not happen. The truth is detected in stage 4 at the latest. In case of a misspeculation, the pipeline is cleared using the rollback mechanism. This concept is taken from [23].

6. FORMAL VERIFICATION

6.1 Pipeline Properties

We verified the correctness of the generated machines using the theorem proving system PVS [8]. This comprises both data consistency and liveness. The data consistency criterion is taken from [20]: Let

$$I_0, I_1, \ldots$$

denote an instruction sequence. For nonnegative integers i, pipeline stages $k \in \{0, \ldots, n-1\}$, and cycles T we denote by

$$I(k,T) = i$$

the fact that instruction I_i is in stage k in cycle T. This function is called *scheduling function*. In order to simplify some proofs, the domain of the function above is extended to cycles T in which no instruction is in stage k (i.e., the stage is not full). If the stage k was never full before cycle T, $I(k,T)$ is supposed to be zero. If the stage k was full before cycle T, the supposed value of the function $I(k,T)$ is defined using the value the function had in the last cycle $T' < T$ such that $full_k^{T'}$ holds. In this case, $I(k,T)$ is supposed to be $I(k,T')+1$ in anticipation of the next instruction in the stage. In contrast to the definition of the scheduling function in [20], such a scheduling function is total.

Let R_I^T denote the actual value of R in the implementation machine during cycle T. The same notation is used for signals, e.g., $full_k^T$ denotes the value of the signal $full_k$ during cycle T.

For sake of simplicity, we omit rollback in the following arguments. For $T = 0$, $I(k,T)$ is zero for all stages. An inductive defi-

nition for I and $T > 0$ for a pipelined machine is [14]:

$$I(k,T) \quad = \quad \begin{cases} I(k,T-1) & : \quad \overline{ue_k^{T-1}} \\ I(0,T-1)+1 & : \quad ue_k^{T-1} \wedge k = 0 \\ I(k-1,T-1) & : \quad ue_k^{T-1} \wedge k \neq 0 \end{cases}$$

LEMMA 1. *One shows the following properties of this function by induction:*

1. *For $T > 0$, the value of I for a given stage increases by one iff the update enable signal of the stage is active:*

$$I(k,T) = \begin{cases} I(k,T-1) & : \quad ue_k(c^{T-1}) = 0 \\ I(k,T-1)+1 & : \quad otherwise \end{cases}$$

2. *Given a cycle T, the values of the scheduling functions of two adjoining stages are either equal or the value of the scheduling function of the later stage is one higher.*

3. *Iff the values are equal, the full bit of the later stage is not set.*

$$full_k^T = 0 \Leftrightarrow I(k-1,T) = I(k,T)$$

Negating both sides of the last equation results in:

$$full_k^T = 1 \Leftrightarrow I(k-1,T) = I(k,T)+1$$

6.2 Data Consistency

Let R be a register which is visible to the programmer. By R_S^i we denote the correct value of R right before the execution of instruction I_i. Let instruction i be in stage k during cycle T and let $R \in out(k)$ be a visible register. The data consistency claim is:

$$R_I^T \quad \stackrel{!}{=} \quad R_S^i$$

This is shown by induction on T. Due to lack of space, we omit the full proof. The interesting part is how to argue the correctness of the input values generated by the forwarding logic. For this paper, we restrict the proof to the case that a register R is read which is part of a register file and a hit signal is active with $top \neq w$. In the following claims, let stage k the stage for which forwarding is done, and let $I(k,T) = i$, $full_k^T$, and $R \in out(w)$ hold. Furthermore, let x be the address of the operand of instruction I_i which is to be forwarded.

LEMMA 2. *If there is an active hit signal, register $R[x]$ is not modified from instruction $I(top,T)+1$ to instruction i:*

$$R_S^{I(top,T)+1}[x] \quad = \quad R_S^i[x]$$

It is not surprising that one argues about the instruction $I(top,T)+\mathbf{1}$. In case of an active hit signal, the forwarding hardware takes the *output* of the stage top.

Due to lack of space, we omit the full proof of this lemma. It uses the following arguments: Since stage top is the first stage with an active hit signal, all stages above do not have the hit signal set. Let

$$diff = I(k,T) - I(top-1,T)$$

denote the difference between the scheduling functions. Using lemma 1 one can argue that this is also the number of instructions (i.e., full stages) in the pipeline between stage k and top. For an empty stage, nothing has to be shown since the values of the scheduling functions match. For a full stage one argues that the instruction in that stage does not write the register.

LEMMA 3. *During cycle T, let there be an active hit signal and let the data hazard signal be not active. The claim is that the input generated by the forwarding logic during cycle T are correct:*

$$g_k R \quad \stackrel{!}{=} \quad R_S^i[x]$$

Proof The claim is shown inductively beginning with the last stage and proceeding from stage $k+1$ to stage k. In case of the last stage, which is stage $n-1$, there is nothing to show since there is no stage below to forward from. Assuming the claim holds for stages k' with $k < k' < n$, the claim is shown for stage k as follows:

As required in the premise, the data hazard signal $R_k dhaz^T$ is not active. By definition of the data hazard signal, this implies that the valid bit of the stage top is active and that the data hazard signal of stage top is not active.

As described above, one assumes the correctness of the inputs of the stages $k' > k$ in order to show the correctness of the inputs of stage k. Since $top > k$, one can apply the induction premise for stage top. This shows the correctness of the inputs of the stage top.

The claim is now shown by a case split on the value of top (in PVS, a separate lemma is used for the possible values of top).

Let $top \neq w$ hold, i.e., the hit is not in the stage which writes R. As above, let register Q be the designated forwarding register and register A the register with the address. In this case, the forwarding logic returns the value written into $Q.(top+1)$.

One can show show that this value is the value of $R[x]$ after the execution of the instruction in stage top by using that the valid signal is active, i.e.:

$$g_k R \quad = \quad R_S^{I(top,T)+1}[x]$$

Thus, the claim is transformed into:

$$R_S^{I(top,T)+1}[x] \quad \stackrel{!}{=} \quad R_S^i[x]$$

Using lemma 2, the claim is concluded.

6.3 Liveness

The liveness criterion is that a finite upper bound exists such that a given instruction terminates. We omit the proof here. The templates for both the data consistency and the liveness proofs required a large amount of manual work in PVS (about one man-month).

7. CONCLUSION

We describe a method which aids the process of designing pipelined microprocessors by transforming a prepared sequential machine into a pipelined machine by adding forwarding and interlock logic. The transformation is not fully automated but the manual effort is very low, since the designer only has to specify the registers holding intermediate results.

The proof of correctness is limited to the changes made during the transformation; the correctness of the prepared sequential machine is assumed to be shown already. However, automated verification of sequential machines is considered state-of-the art. As case study, we easily verify a sequential DLX without floating point unit using the automated rewriting rules and decision procedures of PVS. This machine is transformed into a pipelined machine. Besides the hardware, the tool generates the proofs necessary in order to verify the forwarding and interlock hardware, which yields a provably correct pipeline with forwarding.

APPENDIX

A. REFERENCES

[1] C. Barrett, D. Dill, and J. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'98)*, pages 522–527. ACM, 1998.

[2] M. Bickford and M. Srivas. Verification of a pipelined microprocessor using CLIO. In *Proceedings of Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Springer, 1989.

[3] A. Biere, E. Clarke, E. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 60–71. Springer, 1999.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[6] E. Clarke and A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *In Logic of Programs: Workshop*. Springer, 1981.

[7] E. Clarke, A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[8] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, pages 203–222. Springer, 1994.

[9] M. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones & Bartlett, 1995.

[10] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., 2nd edition, 1996.

[11] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In *Correct Hardware Design and Verification Methods*, pages 8–22. Springer, 1999.

[12] C. Jacobi and D. Kroening. Proving the correctness of a complete microprocessor. In *Proc. of 30. Jahrestagung der Gesellschaft für Informatik*. Springer, 2000.

[13] M. Kaufmann and J. Moore. ACL2: An industrial strength version of nqthm. In *In Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE, 1996.

[14] D. Kroening, W. Paul, and S. Mueller. Proving the correctness of pipelined micro-architectures. In *Proc. of the ITG/GI/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, pages 89–98. VDE, 2000.

[15] J. Levitt and K. Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *33rd Design Automation Conference (DAC'96)*, pages 558–563. ACM, 1996.

[16] M. McFarland. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):633–654, 1993.

[17] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[18] K. McMillan. Verification of an implementation of Tomasulo's algorithm by composition model checking. In *Proc. 10th International Conference on Computer Aided Verification*, pages 110–121, 1998.

[19] K. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods*, pages 219–233. Springer, 1999.

[20] S. Müller and W. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[21] D. Patterson and J. Hennessy. *Computer Organization and Design – The Hardware / Software Interface*. Morgan Kaufmann Publishers, 1994.

[22] J. Sawada and W. Hunt. Results of the verification of a complex pipelined machine model. In *Correct Hardware Design and Verification Methods*, pages 313–316. Springer, 1999.

[23] J. Smith and A. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[24] M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'00)*, pages 112–117. ACM Press, 2000.

[25] P. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1):54–72, 1995.