Top-level Refinement in Processor Verification

Sava Krstić, Byron Cook, John Launchbury, and John Matthews

Oregon Graduate Institute
{krstic, byron, jl, johnm}@cse.ogi.edu

Abstract. We provide a framework for the specification and verification of high-performance processors. As an example, we give a high-level specification and correctness proof for a processor that uses speculation, register renaming, superscalar out-of-order execution, and resolution of memory dependencies. The specifications of its three concurrently operating units are very general and can be refined independently, so that our proof covers a whole family of microarchitectures. Abstract treatment of data, representation of on-the-fly instructions as transactions, and a history table containing the full information of a processor's run are the main features of the proof.

1 Introduction

A variety of formal verification tools are now in use in various phases of hardware design; [2, 8, 17] are but a few notable examples. At the microarchitectural level, however, the real use of verification is limited, mostly due to the immaturity of the available techniques. Indeed, proving the correctness of a combination of aggressive strategies to resolve inter-instruction dependencies is extremely difficult. Still, it is an important verification aspect because microarchitectural defects can impact a large fraction of the design and so are hard to fix. Engineers close to current processor design teams inform us that designers purposefully forgo promising optimizations because they cannot guarantee the optimizations preserve correctness.

Following the top-down approach, we address the question of specifying and verifying processors at a high level. On a worked out example, we show how to abstract the specification as much as possible in order to clearly and concisely specify a complex microarchitecture with the following package of features: speculation, register renaming, superscalar out-of-order execution with in-order retirement, and resolution of memory dependencies. We present only the essentials of the microarchitecture, just enough to make the correctness proof possible. The lower-level details are left to further refinement.

Our example is based on an executable processor model expressed using *Hawk*, a specification language with stream transformer semantics [7, 15]. This example microarchitecture is close to Intel's PentiumPro [10] and AMD's K6 [20]. It is partitioned into three major units for which we provide independent axiomatic specifications. We show that the visible output computed by this microarchitecture is equivalent to that of a simple reference machine implementing the instruction set architecture. This approach exhibits a very desirable form of modularity where the three units can be independently refined further without affecting global correctness. Moreover, since the units are to a large extent underspecified, our proof covers a whole family of microarchitectures that can significantly vary in implementation details.

To write the specifications and organize the proof, we use a small number of concepts and structures of a general nature. For example, our correctness criterion can be used for any model with in-order retirement. Next, *transactions* (a formalized notion of partially computed instructions) seem to be just the right microarchitectural abstraction that provides uniformity in the description of the data path. Transactions come with a natural partial order (progress in computation of an instruction) that enhances their expressiveness and can be effectively used in reasoning. The proof itself revolves around a *history table* which contains all crucial information about a single run of a processor.

After a brief discussion of related work, the rest of the paper is organized by sections, as follows: we specify a reference machine, introduce transactions and (informally) our processor model, describe the correctness criterion, explain the history table and the structure of the proof, and give formal specifications of the three processor components. The full definition of the history table and a proof of the correctness theorem are relegated to the Appendix.

2 Related Work

The complexity of verified processor models described in the literature varies, largely in connection with the level of proof automation. Highly automated methods show a promising trend of consistent increase of applicability, including impressive recent proofs of out-of-order execution [5, 16]. Still, the models verified by these methods are rather limited. This paper belongs to the other end of the spectrum: our processor model is one of the most complex, but at the price of having been specified in a rather unconstraned mathematical style, and verified by a pencil-and-paper proof. The same can be said of the work of Arvind and Shen [4], whose appealing processor model is defined as a term-rewriting system. While our specifications allow refinement in the most obvious sense, it is not clear how the correctness result of [4] that relies on being able to apply the rewrite rules in any order would translate to a lower-level implementation that lacks that property.

With Pnueli and Arons [18] we share the insistence on maximal abstraction and modularity stemming from specifying the processor as a simple composition of concurrent subsystems. There is also some similarity in the correctness criterion, based on the idea of refinement. Their model, however, assumes a restricted instruction set, without branches and memory instructions.

The correctness criterion adopted in most processor verification papers is the "commutative diagram" condition of Burch and Dill [6], or some version thereof (*cf.* [4, 12, 14, 19]). Along with [18], we avoid dealing with explicit synchronization and abstraction functions that match the states of the verified processor with the states of the reference machine. Instead, our criterion requires that the two sequences of retired instructions arising from running the same program on the two machines are equivalent.

Dealing with memory instructions combined with out-of-order execution has only recently come into the scope of processor verification efforts; *cf.* [4, 12, 19]. Our execution unit allows multiple refinements with arbitrarily sophisticated treatment of memory operations (load bypassing, for example).

A remarkably detailed model, including a treatment of exceptions, is verified by Sawada and Hunt [19] using a methodology which has many similarities to our work. The key structure they use, the *Microarchitectural Execution Trace Table*, contains entries that are much like our transactions. This table represents the current computational state of the processor like a row of our history table does. A global invariant relates the table with the corresponding microarchitectural state. Since it references most of the state elements, this invariant presents a difficult proof obligation, which unfortunately is only briefly discussed in [19].

Our paper promotes hierarchical verification by providing a very general and non-deterministic model and a straightforward reduction to verification of components. At this level, the assume-guarantee style takes a simple form: all that the components assume of the environment are type-correct values on their input wires; *cf.* [11].

3 Standard machine (ISA)

Our reference model is an abstract *standard machine*, defined as a state machine whose states consist of values for the program counter, register file and memory. Most of the common instruction set architectures are instances of it when we ignore the treatment of external exceptions.

Definition 1. *Given a state* $(\mathtt{pc}, \mathtt{rf}, \mathtt{mem})$*, the standard machine (executing a fixed program* $\mathtt{pgm}$*) makes a transition to the state* $(\mathtt{pc}', \mathtt{rf}', \mathtt{mem}')$ *defined by the following set of equalities.*

$$\begin{aligned}
I &= \mathtt{pgm}(\mathtt{pc}) \\
(\mathtt{opcode}, \mathtt{rSources}, \mathtt{rDest}) &= \mathtt{decode}(I) \\
\mathtt{rOps} &= \mathtt{rf}(\mathtt{rSources}) \\
(\mathtt{mSource}, \mathtt{mDest}) &= \mathtt{getAddr}(\mathtt{opcode}, \mathtt{rOps}) \\
\mathtt{mOp} &= \mathtt{mem}(\mathtt{mSource}) \\
(\mathtt{pc}', \mathtt{rRes}, \mathtt{mRes}) &= \mathtt{compute}(\mathtt{pc}, \mathtt{opcode}, \mathtt{rOps}, \mathtt{mOp}) \\
\mathtt{rf}' &= \begin{cases} \mathtt{rf}[\mathtt{rDest} \mapsto \mathtt{rRes}] & \textit{if}\ \mathtt{rDest} \in \mathbf{Reg} \\ \mathtt{rf} & \textit{if}\ \mathtt{rDest} = () \end{cases} \\
\mathtt{mem}' &= \begin{cases} \mathtt{mem}[\mathtt{mDest} \mapsto \mathtt{mRes}] & \textit{if}\ \mathtt{mDest} \in \mathbf{Addr} \\ \mathtt{mem} & \textit{if}\ \mathtt{mDest} = () \end{cases}
\end{aligned}$$

The function `decode` extracts the opcode, source registers and the destination register from an instruction. The function `getAddr` computes the addresses `mSource` for loads and `mDest` for stores. Finally, the results of `compute` are the new value for the program counter and the values to be written back to the register file or memory.

The standard machine is totally data-insensitive. It uses abstract basic types **IAddr**, **Instr**, **Opcode**, **Value**, **Reg** and **Addr**, and the rest is typed as follows:

$$\begin{array}{r}
\texttt{pc}\colon \mathbf{IAddr},\ \texttt{pgm}\colon \mathbf{IAddr} \to \mathbf{Instr},\ \texttt{rf}\colon \mathbf{Reg} \to \mathbf{Value},\ \texttt{mem}\colon \mathbf{Addr} \to \mathbf{Value} \\
\texttt{decode} : \mathbf{Instr} \to \mathbf{Opcode}, \mathbf{RegSeq}, \mathbf{Reg}^\sharp \\
\texttt{getAddr} : \mathbf{Opcode}, \mathbf{ValueSeq} \to \mathbf{Addr}^\sharp, \mathbf{Addr}^\sharp \\
\texttt{compute} : \mathbf{IAddr}, \mathbf{Opcode}, \mathbf{ValueSeq}, \mathbf{Value}^\sharp \to \mathbf{IAddr}, \mathbf{Value}^\sharp, \mathbf{Value}^\sharp
\end{array}$$

where we follow the convention to write product types using commas and function types using arrows. The notation $\mathbf{Type}^\sharp$ is a shorthand for the sum type $\mathbf{Type} + \{()\}$, where the element $()$ indicates a value that does not need computation. For example, the first component of the result of `getAddr` is $()$ unless the first argument is the opcode of a load instruction. Note that our definition allows a single instruction to have the combined behavior of a branch, alu-instruction, load and store, if desired. Particular instructions may of course choose to only implement a subset of this functionality.

4 An example processor

When reasoning about the execution process of complex processors one normally thinks of instructions as entities that come into being at a certain cycle and evolve thereafter. *Transactions* formalize this notion of partially computed instructions. Informally, a transaction is a package of information which (directly or indirectly) contains the identity of the unique (static) instruction it is associated with plus various data extracted from the processor's state that are relevant for the execution of that instruction.

Guided by the standard machine specification, we define a *standard transaction* as a record with the following eleven fields:

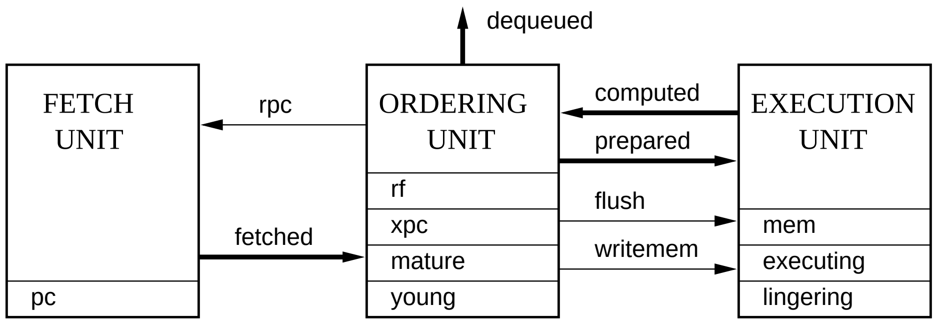
instr	: **Instr**	rDest	: $\mathbf{Reg}^\sharp$
opcode	: **Opcode**	mSource, mDest	: $\mathbf{Addr}^\sharp$
rSources	: **RegSeq**	npc	: **IAddr**
rOps	: **ValueSeq**	mOp, rRes, mRes	: $\mathbf{Value}^\sharp$

We assume that all our basic types contain a value $\bot$, indicating an uncomputed value. We will also use the notation $\mathsf{rOp}_i(T)$ for the i^{th} member of the sequence $\mathsf{rOps}(T)$. The functions `decode`, `getAddr` and `compute` treat $\bot$ as an argument in a lazy fashion: a component of their result is $\bot$ only if some crucial arguments needed for computation of that result are $\bot$.

A natural idea, introduced in [3] and paradigmatic for the *Hawk* specification language [15], is to use transactions as a unifying concept in microarchitectural specifications. Transactions are passed along wires and manipulated by processor components. In addition to the above standard fields, any specific microarchitecture adds fields appropriate for the description of its execution algorithm. Our example processor adds five new fields: the *instruction address* addr, the *speculative next program counter* spc, the *name* (alias) name, the *register providers* rProvs and the *most recent store* mrSt:

$$\begin{array}{llll} \mathsf{addr}, \mathsf{spc} & : \mathbf{IAddr} & \mathsf{rProvs} & : \mathbf{NameOptSeq} \\ \mathsf{name} & : \mathbf{Name} & \mathsf{mrSt} & : \mathbf{NameOpt} \end{array}$$

The fields rProvs and mrSt will record dependencies among instructions. Here **NameOpt** = **Name** + {NONE} is the type of an optional name field, where NONE serves to indicate the lack of dependency.



`pc`, `xpc`	: **IAddr**	`rpc`	: $\mathbf{IAddr}^{\sharp}$
`rf`	: **Reg** → **Value**	`fetched`, `dequeued`, `prepared`	: **TransSeq**
`mature`, `young`	: **TransSeq**	`computed`	: **TransSet**
`mem`	: **Addr** → **Value**	`flush`, `writemem`	: **Bool**
`executing`, `lingering`	: **TransSet**		

Fig. 1. Top-level specification with the types of wires (right) and state components (left). Thick wires represent transaction sets or sequences. At each cycle, units update their state and output wires depending on the values on their input wires and state elements at the previous cycle.

The processor consists of three major units and seven wires as depicted in Fig. 1. The *fetch unit* provides multiple instructions at each cycle. This unit outputs along the `fetched` wire transactions with filled in fields instr, addr and spc. The fetching of instructions begins at the address `pc` if the current value of `rpc` (requested program counter) is (); otherwise `rpc` is used. The fetching proceeds by unconstrained speculation.

The *ordering unit* maintains the sequential programming model of the ISA by using a queue made by concatenating the sequences `mature` and `young` (Fig. 2). It takes a prefix of the sequence `fetched` to form a transaction sequence `enqueued` to be added to the back of the queue. The transactions of `fetched` that do not belong to the chosen prefix are discarded. Each transaction added to the queue gets its name field filled in, unique in the queue. The `mature` part of the queue corresponds to transactions already sent to the execution unit. Transactions in `prepared` are taken from the beginning of the `young` part of the queue and possibly also from `enqueued`; they all have their rOps, rProvs and mrSt fields filled in. The elements of rOps obtain values from `rf` when there is no dependency on previous transactions; if there are dependencies, they are recorded in the elements of rProvs, which contain the names of the transactions that will provide the appropriate values when computed. The field mrSt contains the name of the last preceding store in the queue; it is used only by loads and stores for future resolution of dependencies among them. The `mature` part of the queue is updated by transactions arriving along the `computed` wire, then a prefix of the resulting sequence consisting entirely of complete transactions is retired, that is, sent along the `dequeued` wire while updating `rf`. When a retired transaction is a mispredicting branch, then the queue is emptied, the Boolean wire `flush` is asserted and `rpc` set equal to the address of the last retired transaction. The wire `rpc` is also given a non-trivial value when not all fetched transactions are enqueued. In this case the `rpc` is set to the spc of the last enqueued transaction.

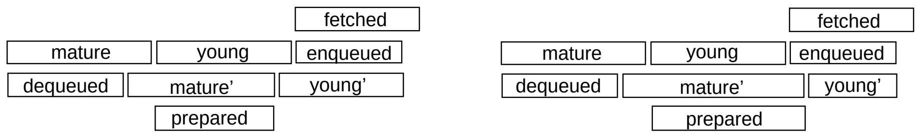


Fig. 2. Two possible scenarios for the relationship between transaction sequences involved in a transition of the ordering unit. The inputs are `fetched`, `mature` and `young`, and the outputs are `dequeued`, `prepared`, `mature`′ and `young`′. The sequences are aligned so that if two transactions are on the same vertical line, then the higher one is less than or equal to the lower (in the progress ordering defined below).

The *execution unit* is an out-of-order component that computes the results rRes and mRes of transactions contained within it and determines which of these transactions are mispredicting (by computing npc for each and comparing it with spc). It may also execute a memory store if the value on the wire `writemem` indicates that it is right time to do so. A number of completed transactions are sent out along the `computed` wire, while placing them in the set `lingering`, where each of them will remain intact until the moment when an equally named transaction comes along the `prepared` wire and takes its place. When a transaction is sent to `computed` (or sooner), the values in its result fields are forwarded to all other transactions in `executing`. There are no requirements on the number of transactions executed at each cycle and the only requirement on the order of their execution is that the data-flow order is respected.

5 Correctness criterion

One can slightly extend the definition of the standard machine so that at each cycle it outputs a complete transaction (corresponding to the instruction completed at that cycle). A run of the standard machine then defines a sequence of "retired" transactions from which the corresponding sequence of states of the standard machine can easily be reconstructed.

A transition of a complex processor cannot, in general, be associated with a unique transaction, but with a sequence, possibly empty, of transactions retired on that transition. So, suppose P is a processor and denote by ρ_n the sequence of transactions retired by P on its n^{th} cycle. Concatenating these sequences we obtain $\rho_\infty = \rho_1\rho_2\cdots$. Replacing every transaction in ρ_∞ with the corresponding standard transaction (which amounts to ignoring its "non-standard" fields), we obtain a sequence of standard transactions ρ_∞^{std}, which, if P does implement the standard machine, should be identical to the appropriate execution sequence of the standard machine. This gives us the following correctness criterion.

Definition 2. *A processor P is* correct with respect to the standard machine *if for any given program* `pgm` *and a state σ_0 of the standard machine, there exists an initial state of P such that the execution of* `pgm` *on P produces a sequence of retired transactions ρ_∞ with the associated sequence ρ_∞^{std} equal to the execution sequence defined by the program* `pgm` *and the initial state σ_0.*

The notion of the execution sequence is made precise below, after a brief elaboration of the type of transactions.

5.1 The progress ordering of transactions

We define the *progress ordering* $\preceq$ on the set of transactions so that $T_1 \preceq T_2$ will mean that T_2 is a computationally more advanced ("closer to retirement") version of T_1. The relation $\preceq$ is the product of 16 partial orders (all denoted $\preceq$)—one for each record component. These component orders are defined as follows. For each basic type (including **Name**), we make $\bot$ the smallest element and all other elements, including (), incomparable. In **NameOpt**, NONE is the largest element. Finally, two sequences are comparable if and only if they have the same length and the elements of one of them are all less than or equal to the corresponding elements of the other.

The partial order just introduced allows us to define the notion of intrinsic consistency of transactions. Intuitively, a transaction is consistent if the contents of its fields do not contradict any of the equations occurring in the definition of the standard machine. Of these equations, the ones that do not involve the components of the machine state (program counter, register file and memory) give rise to consistency criteria:

$$\langle \mathsf{opcode}(T), \mathsf{rSources}(T), \mathsf{rDest}(T)\rangle \preceq \mathtt{decode}(\mathsf{instr}(T))$$

$$\langle \mathsf{mSource}(T), \mathsf{mDest}(T)\rangle \preceq \mathtt{getAddr}(\mathsf{opcode}(T), \mathsf{rOps}(T))$$

$$\langle \mathsf{npc}(T), \mathsf{rRes}(T), \mathsf{mRes}(T)\rangle \preceq \mathtt{compute}(\mathsf{addr}(T), \mathsf{opcode}(T), \mathsf{instr}(T), \mathsf{rOps}(T), \mathsf{mOp}(T))$$

By definition, a transaction is *consistent* if its fields satisfy these inequalities. We define **Trans** to be the set of all consistent transactions. Note that consistency of a transaction depends entirely on the contents of its "standard" fields and that all strictly increasing chains in the poset (**Trans**, $\preceq$) are of finite length.

Maximal transactions with respect to the ordering $\preceq$ will be called *complete*; a transaction is complete if none of its fields is $\perp$, and mrSt and all component fields of rProvs are NONE.

5.2 Execution sequences

For every transition of the standard machine there is an associated complete standard transaction. To define it, just use the left-hand sides of the equations in Definition 1. Thus, together with every run of the standard machine, one can consider the corresponding transaction sequence $\langle T_1, T_2, \ldots\rangle$, where T_i corresponds to the i^{th} transition. Characterizing properties of such sequences are collected in Definition 3 below.

If τ is a (finite or infinite) sequence of transactions or standard transactions and T a transaction in τ, we define the i^{th} *register provider* of T to be the transaction U of τ which precedes T and has the property that $\mathsf{rSource}_i(T) = \mathsf{rDest}(U)$, while $\mathsf{rSource}_i(T) \neq \mathsf{rDest}(V)$ for all transactions V between U and T. Similarly, we define U to be the *store provider* of T if T is a load and U is the last store among the transactions that precede T in τ and satisfy $\mathsf{mSource}(T) = \mathsf{mDest}(U)$.

Definition 3. *An infinite sequence $\tau = \langle T_1, T_2, \ldots\rangle$ is an* execution sequence *corresponding to the program* $\mathtt{pgm}$ *and the initial state* $(\mathtt{pc}_{init}, \mathtt{rf}_{init}, \mathtt{mem}_{init})$ *if every T_m is a complete transaction and*

$$\mathsf{instr}(T_m) = \begin{cases} \mathtt{pgm}(\mathtt{pc}_{init}) & \text{if } m = 0 \\ \mathtt{pgm}(\mathsf{npc}(T_{m-1})) & \text{if } m > 0 \end{cases}$$

$$\mathsf{rOp}_i(T_m) = \begin{cases} \mathsf{rRes}(T_k) & \text{if } T_k \text{ is the } i^{\text{th}} \text{ register provider for } T_m \text{ in } \tau \\ \mathtt{rf}_{init}(\mathsf{rSource}_i(T_m)) & \text{if } T_m \text{ does not have an } i^{\text{th}} \text{ provider in } \tau \end{cases}$$

$$\mathsf{mOp}(T_m) = \begin{cases} \mathsf{mRes}(T_k) & \text{if } T_k \text{ is the store provider for } T_m \text{ in } \tau \\ \mathtt{mem}_{init}(\mathsf{mSource}(T_m)) & \text{if } T_m \text{ does not have a store provider in } \tau \end{cases}$$

6 History Table (Structuring the proof)

Reasoning about the execution of processors can be conveniently organized around a *history table*. Two simple observations are behind its definition. First, if $I_1, I_2, \ldots$ is the sequence of instructions considered by the processor during a run, then each transaction T found anywhere in the processor at any time is associated with a unique fetched instruction I_j; we say that j is the *ordinal* of T. The second observation is that there are only finitely many essentially different execution patterns for an instruction and that one can define a finite transition diagram describing those patterns. Each node of this *transaction flow diagram* Γ corresponds to a distinguished "pipeline stage" and will be called a *status*.

A history table is defined for every run of the processor. At the n^{th} row and the i^{th} column of the table one finds a pair $H_n^i = (T, X)$, where T is the transaction that represents the state of computation of I_i at the n^{th} cycle and X is the status of that computation. Formally, H_n^i is defined in terms of the set of transactions with ordinal i which are present in the processor at the n^{th} cycle, and the values of "control" variables at that cycle; normally, T is the maximal of those transactions and the status X corresponds to the set of locations in which they are found.

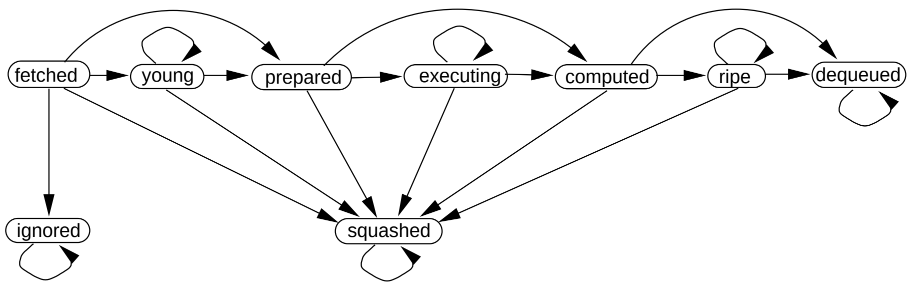


Fig. 3. Transaction flow diagram Γ. The transitions to `squashed` occur only when `flush` = TRUE.

For our example processor, Γ is given in Fig. 3. The top row represents the execution patterns of successfully completed instructions. Looping at `young` means waiting to be sent to the execution unit; the loops at `executing` and `ripe` have similar meaning. The status `ripe` corresponds to the set of complete transactions contained in `mature`. The final statuses `ignored` and `squashed` are for transactions aborted because of the overflow in the ordering unit (inability to enqueue all fetched transactions) and misprediction, respectively.

The rows of the history table are finite; the length of the n^{th} row is equal to the total number of fetched instructions in the first n cycles. All columns stabilize: for each i, we have $H_{n+1}^i = H_n^i$ for all large n. This follows since both **Trans** and Γ are posets in which strictly increasing chains are finite. We define the *limit row* H_∞ as the sequence of the limit values of columns: $H_\infty^i = \lim_n H_n^i$.

For any $n \leq \infty$, denote by τ_n the sequence of transactions occurring in the n^{th} row H_n of H. Let also τ_n^{D} denote the sequence consisting of only those transactions occuring in H_n whose corresponding status component is `dequeued`. The correctness of the processor can then be restated as follows.

Theorem. *τ_∞^{D} is an execution sequence.*

In view of Definition 3, this presents us with four proof obligations.

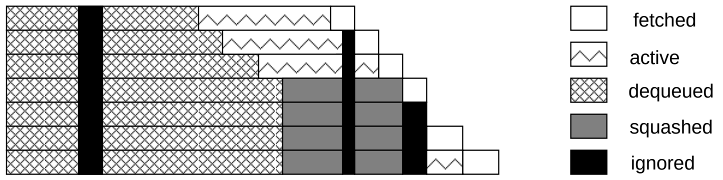


Fig. 4. Seven consecutive rows in the middle of a history table. The second depicts a cycle when only part of the fetched transactions is enqueued. The first misprediction is seen in the fourth row; transactions fetched at this cycle are ignored at the next, when also, due to the misprediction, the fetching unit was unable to output. ("Active" stands for statuses that are neither initial nor final and reflects the queue in the ordering unit.)

Proposition 1. *The sequence* $\tau^{\mathsf{D}}_{\infty}$ *is infinite.*

Proposition 2. *If* U *and* T *are two consecutive elements of* $\tau^{\mathsf{D}}_{\infty}$*, then* $\mathsf{npc}(U) = \mathsf{addr}(T)$*. Also, the value or the* addr *field of the first transaction of* $\tau^{\mathsf{D}}_{\infty}$ *id* $\mathtt{pc}_{init}$*.*

Proposition 3. *Let* T *be a transaction in* $\tau^{\mathsf{D}}_{\infty}$*. If* U *is the* r^{th} *register provider of* T *in* $\tau^{\mathsf{D}}_{\infty}$*, then* $\mathsf{rOp}_r(T) = \mathsf{rRes}(U)$*, and if* T *does not have an* r^{th} *provider in* $\tau^{\mathsf{D}}_{\infty}$*, then* $\mathsf{rSources}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$*.*

Proposition 4. *Let* T *be a transaction in* $\tau^{\mathsf{D}}_{\infty}$*. If* U *is the store provider of* T *in* $\tau^{\mathsf{D}}_{\infty}$*, then* $\mathsf{mOp}(T) = \mathsf{mRes}(U)$*, and if* T *does not have a store provider in* $\tau^{\mathsf{D}}_{\infty}$*, then* $\mathsf{mOp}(T) = \mathtt{mem}_{init}(\mathsf{mSource}(T))$*.*

The proof of Proposition 1 uses the liveness conditions of components. The major results one needs to establish are the infinity of the sequences of fetched and enqueued transactions, and the absence of livelock, expressed as the statement that all locations in H_{∞} are final. Proving the remaining three propositions involves a rather straightforward but tedious chasing around the history table.

7 Formal specification

Staying close to the *Hawk* specification style, we model processors and their components as state machines, which use sets of input wires, output wires, and states, each wire and each piece of state having a prescribed type. The machine is then defined by a function whose arguments are the values for input wires and states, and whose results are values for the output wires and states in the next clock cycle. Consequently, the machine acts as a signal transformer: for any given signals (infinite sequences) of inputs and initial values of states, it produces uniquely determined signals of outputs.

An axiomatic specification of a state machine could consist of a list of its input, output and state variables, an initial condition, an invariance condition, and a liveness condition. Without making these notions precise, we note that an invariant is a propositional formula written in terms of input variables, output variables, state variables and primed state variables, and a liveness condition is a property of signals expressible by a suitable formula in temporal logic.

Again without going into technicalities, state machines can be composed by identifying each output wire of the constituent machines with some (zero or more) input wires. At the level of signals, which is how it is done in Hawk, composition amounts to writing a system of equations, each corresponding to a component machine.

The input, output and state variables of the three components of our processor can be read off from Fig. 1, which also tells how the wires are joined to give a specification of the processor as a composition of its components. The formulas for specifications of components are given below, after introducing notational conventions.

The values $\mathtt{pgm}$, $\mathtt{pc}_{init}$, $\mathtt{rf}_{init}$ and $\mathtt{mem}_{init}$ are constants.

We restrict the type **TransSeq** to "uniquely named" sequences: if two transactions in a sequence have names x and y, none of which is $\perp$, then $x \neq y$. The concatenation of sequences α and β is denoted $\alpha \mathbin{+\!\!\!+} \beta$. A partial order on the set of transaction sequences is defined by $\alpha \preceq \beta$ if and only if $|\alpha| = |\beta|$ and $\alpha[i] \preceq \beta[i]$ for every i. A transaction is *mispredicting* if its spc and npc fields are not equal, and none is equal to $\perp$. A transaction is *decoded* if none of its fields opcode, $\mathsf{rSources}$, rDest contains $\perp$. A transaction is *independent* if its mrSt and and rOps fields are maximal (the first is NONE and the second does not contain $\perp$). A transaction T *depends on* another transaction U if $\mathsf{rProv}_i(T) = \mathsf{name}(U)$ or $\mathsf{mrSt}(T) = \mathsf{name}(U)$. If T is a transaction in a transaction sequence α, then the *most recent store* of T in α is the last store in α that precedes T. Finally, if A is a transaction set and T is a transaction, then the *store chain* of T in A is the maximal sequence $\langle S_k, \ldots, S_1\rangle$ with the properties $\mathsf{mrSt}(T) = \mathsf{name}(S_1)$ and $\mathsf{mrSt}(S_i) = \mathsf{name}(S_{i+1})$ for $1 \leq i < k$.

Transaction sets have the property that different elements of a set have distinct names; we use the type $\mathbf{TransSet} = (\mathbf{Name} - \{\perp\}) \to \mathbf{Trans}^\sharp$ to represent such sets. For A and B in **TransSet**, we denote by $A \cup B$ the union of A and B with A having the higher priority; that is, if A and B both have a transaction named x, then the transaction named x of $A \cup B$ is that of A. (This union operation is associative, but not commutative.) The notation $A \preceq B$ means by definition that $A(x) \preceq B(x)$ for every $x \in \mathbf{Name}$. Note that there is a canonical map $\mathbf{TransSeq} \to \mathbf{TransSet}$, so every transaction sequence can be regarded as a transaction set.

For $\mathtt{rf} \in \mathbf{Reg} \to \mathbf{Value}$, $\mathtt{mem} \in \mathbf{Addr} \to \mathbf{Value}$, $v \in \mathbf{Value}$, $r \in \mathbf{Reg}$ and $a \in \mathbf{Addr}$, the values of the updated register files and memories are denoted by $\mathtt{rf}[r \mapsto v]$ and $\mathtt{mem}[a \mapsto v]$. Note the role of $\perp$ in updating functions: if $\mathtt{rf}' = \mathtt{rf}[\perp \mapsto v]$, then $\mathtt{rf}'(r) = \perp$ for every r, but if $\mathtt{rf}' = \mathtt{rf}[r \mapsto \perp]$ then $\mathtt{rf}'(s) = \mathtt{rf}(s)$ for every $s \neq r$. Updating of a register file and memory by a transaction is defined by

$$\mathtt{rf} \cdot T = \begin{cases} \mathtt{rf}[\mathsf{rDest}(T) \mapsto \mathsf{rRes}(T)] & \text{if } \mathsf{rDest}(T) \in \mathbf{Reg} \\ \mathtt{rf} & \text{if } \mathsf{rDest}(T) = () \end{cases}$$

$$\mathtt{mem} \cdot T = \begin{cases} \mathtt{mem}[\mathsf{mDest}(T) \mapsto \mathsf{mRes}(T)] & \text{if } \mathsf{mDest}(T) \in \mathbf{Addr} \\ \mathtt{mem} & \text{if } \mathsf{mDest}(T) = () \end{cases}$$

The results $\mathtt{rf} \cdot \tau$ and $\mathtt{mem} \cdot \tau$ of updating $\mathtt{rf}$ and $\mathtt{mem}$ by a finite transaction sequence τ are then defined in a straightforward manner.

Fetching Unit

Let $\mathtt{pc\text{-}rpc} = \mathtt{pc}$ if $\mathtt{rpc} = ()$; otherwise $\mathtt{pc\text{-}rpc} = \mathtt{rpc}$.

Fetch-Init. *The initial values of* $\mathtt{pc}$ *and* $\mathtt{fetched}$ *are* $\mathtt{pc}_{init}$ *and* $\langle\rangle$ *respectively.*

Fetch-Inv 1. $\mathsf{instr}(T) = \mathsf{pgm}(\mathsf{addr}(T))$, *for every transaction* T *occurring in* $\mathtt{fetched}$*.*

Fetch-Inv 2 (Speculation). *If* $\mathtt{fetched} = \langle T_1, \ldots, T_k \rangle$, *then* $\mathsf{addr}(T_1) = \mathtt{pc\text{-}rpc}$, *and* $\mathsf{addr}(T_{i+1}) = \mathsf{spc}(T_i)$ *for every* $i \in \{1, \ldots, k-1\}$*.*

Fetch-Inv 3 (Next PC). $\mathtt{pc}' = \mathsf{spc}(T)$ *if* T *is the last transaction of* $\mathtt{fetched}$, *and* $\mathtt{pc}' = \mathtt{pc\text{-}rpc}$ *if* $\mathtt{fetched} = \langle\rangle$*.*

Fetch-Inv 4 (Empty fields). *A field of a transaction in* $\mathtt{fetched}$ *has a value different from* $\bot$ *if and only if that field is* instr, spc *or* addr*.*

Fetch-Liv. *The formula* $\mathtt{rpc} \neq () \vee \mathtt{fetched} \neq \langle\rangle$ *is true infinitely often.*

Ordering Unit

Denote $\mathsf{queue} = \mathtt{mature} \# \mathtt{young}$.

Ord-Init. *The initial values of* $\mathtt{xpc}$, $\mathtt{rf}$, queue, $\mathtt{flush}$, $\mathtt{prepared}$ *and* $\mathtt{rpc}$ *are* $\mathtt{pc}_{init}$, $\mathtt{rf}_{init}$, $\langle\rangle$, FALSE, $\langle\rangle$ *and* $()$ *respectively.*

Ord-Inv 1 (Naming). *All transactions in* queue *have distinct names.*

Ord-Inv 2 (Queue). *Let* $\mathtt{mature}^*$ *be the sequence obtained from* $\mathtt{mature}$ *by replacing every transaction in it with an equally named transaction of* $\mathtt{computed}$, *if it exists. If* $\mathtt{flush} = $ TRUE *then* $\mathsf{queue}' = \mathtt{prepared} = \langle\rangle$ *and* $\mathsf{dequeued}$ *is a prefix of* $\mathtt{mature}^*$*. If* $\mathtt{flush} = $ FALSE, *then there exists a prefix* $\mathsf{enqueued}$ *of* $\mathtt{fetched}$ *such that*

$$\mathtt{young} \# \mathsf{enqueued} \preceq \mathtt{prepared} \# \mathtt{young}',$$
$$\mathtt{mature}^* \# \mathtt{prepared} = \mathsf{dequeued} \# \mathtt{mature}'.$$

Ord-Inv 3 (Enqueueing). *If* T *is the first transaction of* $\mathsf{enqueued}$, *then* $\mathsf{addr}(T) = \mathtt{xpc}$*. Finally, if* $\mathsf{queue} = \langle\rangle$ *and* $\mathtt{xpc} = \mathsf{addr}(T)$, *where* T *is the first transaction of* $\mathtt{fetched}$, *then* $\mathsf{enqueued} \neq \langle\rangle$*.*

Ord-Inv 4 (Preparation). *Let* T *be a transaction in* $\mathtt{prepared}$*. Then*

1. *T is decoded,* $\mathsf{rRes}(T) = \bot$*, and* $T \in \mathtt{mature}'$*.*
2. $\langle \mathsf{rOp}_i(T), \mathsf{rProv}_i(T)\rangle = \langle \bot, \mathsf{name}(U)\rangle$ *if U is the i^{th} register provider of T in* queue'*, and* $\langle \mathsf{rOp}_i(T), \mathsf{rProv}_i(T)\rangle = \langle \mathtt{rf}'(\mathsf{rSource}_i(T)), \text{NONE}\rangle$ *if T does not have the i^{th} register provider in* queue'*.*
3. $\mathsf{mrSt}(T) = \mathsf{name}(S)$ *if S is the most recent store for T in* queue'*, and* $\mathsf{mrSt}(T) = \text{NONE}$ *if this most recent store does not exist. The value of* $\mathsf{mOp}(T)$ *is $\bot$ or $()$, depending on whether T is a load or not.*

Ord-Inv 5 (Dequeueing). *All transactions of* $\mathsf{dequeued}$ *are complete and none of them, except possibly the last one, is mispredicting.*

Ord-Inv 6 (Register File). $\mathtt{rf}' = \mathtt{rf} \cdot \mathsf{dequeued}$.

Ord-Inv 7 (Flush). $\mathtt{flush} = \text{TRUE}$ *if and only if the last transaction in* $\mathsf{dequeued}$ *is mispredicting.*

Ord-Inv 8 (Enabling a memory write). $\mathtt{writemem} = \text{TRUE}$ *if and only if the first transaction of* queue' *is an incomplete store.*

Ord-Inv 9 (Requested PC).

$$\mathtt{rpc} = \begin{cases} \mathsf{npc}(D) & \textit{if } \mathtt{flush} = \text{TRUE} \\ \mathsf{addr}(E) & \textit{if } \mathtt{flush} = \text{FALSE} \textit{ and } |\mathsf{enqueued}| < |\mathtt{fetched}| \\ () & \textit{otherwise} \end{cases},$$

where D is the last transaction of $\mathsf{dequeued}$ *and* $E = \mathtt{fetched}(|\mathsf{enqueued}| + 1)$*.*

Ord-Inv 10 (Expected PC).

$$\mathtt{xpc}' = \begin{cases} \mathtt{rpc} & \textit{if } \mathtt{rpc} \neq () \\ \mathsf{spc}(T) & \textit{if } \mathtt{rpc} = () \textit{ and } \mathsf{enqueued} \neq \langle\rangle \\ \mathtt{xpc} & \textit{otherwise} \end{cases},$$

where T is the last transaction of $\mathsf{enqueued}$*.*

Ord-Liv. *If the first transaction of* $\mathtt{queue}$ *is complete, then eventually* $\mathtt{dequeued} \neq \langle\rangle$*. If* $\mathtt{mature} = \langle\rangle$ *and* $\mathtt{young} \neq \langle\rangle$*, then eventually* $\mathtt{prepared} \neq \langle\rangle$*.*

Execution Unit

Exec-Init. *The initial value of* $\mathtt{mem}$ *is* $\mathtt{mem}_{init}$*, and $\emptyset$ is the initial value of* $\mathtt{executing}$, $\mathtt{lingering}$ *and* $\mathtt{computed}$.

Exec-Inv 1 (Flushing). *If* $\mathtt{flush} = \text{TRUE}$*, then* $\mathtt{executing}' = \mathtt{lingering}' = \emptyset$ *and* $\mathtt{mem}' = \mathtt{mem}$*.*

Exec-Inv 2 (Contents). *The sets* $\mathtt{executing}$ *and* $\mathtt{lingering}$ *are disjoint. If* $\mathtt{flush} = \text{FALSE}$ *then*

$$\mathtt{executing} \cup \mathtt{prepared} \cup \mathtt{lingering} \preceq \mathtt{executing}' \cup \mathtt{lingering}'. \qquad (1)$$

If T is an element of the left-hand side of (1) and T' is the corresponding element of the right-hand side, we will say that T' is the *descendant* of T. Note that the only transactions of $\mathtt{executing} \cup \mathtt{lingering}$ without a descendant are members of `lingering` whose name occurs in a transaction of `prepared`.

Exec-Inv 3 (Lingering). *Assume* $\mathtt{flush} = \textsc{false}$*. Then all transactions in* `lingering` *are complete and no transaction in* `executing` *depends on any transactions of* `lingering`*. Also, a transaction belongs to* $\mathtt{lingering}'$ *if and only if it either belongs to* `computed`*, or is a descendant of a transaction in* `lingering`*.*

If L is a load in $\mathtt{executing} \cup \mathtt{prepared}$ and ϕ is the store chain of L in this set, then

$$\mathsf{mOp}(L) \preceq (\mathtt{mem} \cdot \phi)(\mathsf{mSource}(L)) \qquad \text{(LC)}$$

is a condition that should be satisfied by the execution unit. Note that the value on the right-hand side is $\perp$ if $\mathsf{mDest}(S) = \perp$ for some S in ϕ. If $\mathsf{mDest}(S) \neq \perp$ for all S in ϕ, then the value on the right-hand side is either (1) $\mathsf{mRes}(S)$, where S is the last transaction in ϕ with $\mathsf{mDest}(S) = \mathsf{mSource}(L)$, or (2) $\mathtt{mem}(\mathsf{mSource}(L))$, if no such S exists.

Exec-Inv 4 (Load Correctness). *If L' is the descendant of a load L which satisfies the condition* (LC)*, then L' satisfies* (LC) *too.*

Exec-Inv 5 (Forwarding). *If T' is the descendant of T, then* $\langle \mathsf{rProv}_i(T'), \mathsf{rOp}_i(T')\rangle = \langle \mathsf{rProv}_i(T), \mathsf{rOp}_i(T)\rangle$*, or* $\langle \mathsf{rProv}_i(T'), \mathsf{rOp}_i(T')\rangle = \langle \textsc{none}, \mathsf{rRes}(U)\rangle$*, where* $U \in \mathtt{executing} \cup \mathtt{lingering}$*,* $\mathsf{rProv}_i(T) = \mathsf{name}(U)$*, and* $\mathsf{rRes}(U) \neq \perp$*.*

Exec-Inv 6 (Memory). *1. If* $\mathtt{mem}' \neq \mathtt{mem}$*, then* $\mathtt{writemem} = \textsc{true}$ *and* $\mathtt{mem}' = \mathtt{mem} \cdot S$*, where S is a complete store in* `executing`*.*
2. If `computed` *contains a store S, then* $\mathtt{mem}' = \mathtt{mem} \cdot S$ *and* $\mathtt{writemem} = \textsc{true}$*.*

Exec-Inv 7 (Most Recent Store). *If T' and U' are descendants of T and U, and if* $\mathsf{mrSt}(T) = \mathsf{name}(U)$*, then* $\mathsf{mrSt}(T') = \mathsf{name}(U')$ *unless* $U' \in \mathtt{computed}$ *or T is a load with* $\mathsf{mOp}(T) \neq \perp$*.*

Exec-Liv. *Let T be an independent transaction in* `executing`*. If T is a store, assume also that* $\mathtt{writemem} = \textsc{true}$*. Then eventually* $\mathtt{flush} = \textsc{true}$ *or* $\mathsf{name}(T)$ *occurs among names of transactions in* `computed`*.*

8 Conclusions

In an attempt to bring the power of verification closer to the complexity of commercial processors, we have specified a general microarchitectural design and proved its correctness. Our axiomatization can be satisfied by a family of microarchitectures; therefore, it retains a good deal of flexibility as the structure of the individual components is developed. Since each component is specified independent of other components, the implementation and proof of components can be carried out independently. Furthermore, our specifications and proof are independent of many considerations that affect performance. For example, we do not need to set the number and latencies of subunits of our execution units, the width of instruction-carrying wires, the accuracy of branch prediction etc. Therefore, many design decisions based on simulation may be made without adversely affecting the global correctness proof. Note also that the wires present in our top-level specification are just what is necessary for interunit communication. The units are free to communicate through extra channels; for example, an extra wire allows implementation of a branch target buffer within the fetching unit.

Most of the advantages of our approach come as a consequence of using a severely minimized axiomatization. This approach is not quite common, probably because coming up with a reasonably complete set of invariants for an algorithm is generally difficult. Considerable skill is required to extract the axioms, but in a limited domain, such as that of hardware design, it could be feasible. We plan to explore the axiomatics for hardware components and develop a library of specifications and typical proofs.

We intend to construct various refinements of our component specifications and thus to show that our axiomatizations can be related to specific microarchitectures. We have already developed executable PentiumPro-like specifications in *Hawk* using the same structure described here (see [1]); we plan to prove the correctness of these executable models by checking their three units satisfy our axioms. Transactions, as we have demonstrated, are a useful microarchitectural abstraction, but they also come with a substantial overhead that should be eliminated in lower-level refinements. We plan to develop a methodology for shrinking the interfaces of our top-level specifications.

We expect that further research will confirm that reasoning around the history table is a promising proof technique, applicable to pipeline designs in general. Also left to further research is rewriting our axiomatics in a more stringent specification style, and mechanization of the proofs.

Acknowledgments. For their contributions to this research, we thank Mark Aagaard, Borislav Agapiev, Robert Jones, and John O'Leary of Intel Strategic CAD Labs; Tito Autrey, Nancy Day, Dick Kieburtz and Thomas Nordin of OGI; and Arvind of MIT.

The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews receives support from a graduate research fellowship with the National Science Foundation.

References

[1] Hawk Web page: `http://www.cse.ogi.edu/PacSoft/Hawk/`.

[2] M. Aagaard, R. Jones, and C.-J. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In 35th *Design Automation Conference (DAC '98)*, pages 538–541. Association for Computing Machinery, 1998.

[3] M. Aagaard and M. Leeser. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[4] Arvind and X. Shen. Design and verification of processors using term rewriting systems. *IEEE Micro*, 1999. to appear.

[5] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In [9], pages 369–386.

[6] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–70. Springer-Verlag, 1994.

[7] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.

[8] Á. P. Eiríksson. The formal design of 1M-gate ASICs. In [9], pages 49–63.

[9] G. Gopalakrishnan and P. Windley, editors. *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[11] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In [13], pages 440–451.

[12] R. Hosabbettu, M. Srivas, and G. Gopalakrihnan. Decomposing the proof of correctness of pipelined microprocessors. In [13], pages 122–134.

[13] A. J. Hu and M. Y. Vardi, editors. *Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[14] R. B. Jones, J. U. Skakkebaek, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In [9], pages 2–17.

[15] J. Matthews, J. Launchbury, and B. Cook. Specifying microprocessors in Hawk. In *1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society, 1998.

[16] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In [13], pages 110–121.

[17] J. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD K86. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[18] A. Pnueli and T. Arons. Verification of data-insensitive circuits: An in-order-retirement study. In [9], pages 351–568.

[19] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In [13], pages 135–146.

[20] B. Shiver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society, 1998.

A Appendix: Correctness Proof

In Sect. 6 we gave a brief and incomplete description of the history table associated to a run of our processor model. A precise definition is given below in Subsect. A.3. In particular, we prove that the columns of the history table stabilize (Lemma 12), so that the sequence τ_∞ of limit values is defined. Recall that the sequence τ_∞^{D} is obtained by removing from τ_∞ all transactions whose corresponding status is not `dequeued`. We prove that this sequence is equal to the concatenation of all sequences of transactions dequeued by our processor in the run being considered (Lemma 16). Thus the correctness of the processor can indeed be expressed as in Theorem stated in Sect. 6. We repeat it here:

Theorem. *τ_∞^{D} is an execution sequence.*

We also repeat the four Propositions which, in view of Definition 3, imply the theorem.

Proposition 1. *The sequence τ_∞^{D} is infinite.*

Proposition 2. *If U and T are two consecutive elements of τ_∞^{D}, then $\mathsf{npc}(U) = \mathsf{addr}(T)$. Also, the value or the addr field of the first transaction of τ_∞^{D} id $\mathtt{pc}_{init}$.*

Proposition 3. *Let T be a transaction in τ_∞^{D}. If U is the r^{th} register provider of T in τ_∞^{D}, then $\mathsf{rOp}_r(T) = \mathsf{rRes}(U)$, and if T does not have an r^{th} provider in τ_∞^{D}, then $\mathsf{rSources}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$.*

Proposition 4. *Let T be a transaction in τ_∞^{D}. If U is the store provider of T in τ_∞^{D}, then $\mathsf{mOp}(T) = \mathsf{mRes}(U)$, and if T does not have a store provider in τ_∞^{D}, then $\mathsf{mOp}(T) = \mathtt{mem}_{init}(\mathsf{mSource}(T))$.*

The proofs of the propositions are given in Subsections A.5–A.8. The definition and some basic properties of the history table are given in Subsection A.3. The first two subsections contain notational preliminaries and key lemmas about the relationships among the processor's components.

A.1 Terminology

Regular and singular cycles. For a given run of the processor, the value of any state variable v at the cycle n ($n \geq 1$) will be denoted by v^n. Define n to be *regular* or *singular* depending on whether $\mathtt{flush}^n$ is FALSE or TRUE. Note that n is singular if and only if $\mathtt{dequeued}^n$ is non-empty and the last transaction in it is mispredicting (Ord-Inv 5). Note also that if n is singular, then $\mathtt{queue}^n$, $\mathtt{executing}^{n+1}$ and $\mathtt{executing}^{n+1}$ are empty, by Ord-Inv 2 and Exec-Inv 1 respectively. As a consequence, we have that two consecutive numbers cannot be both singular.

Locations. Let us use the term *location* for the four wires ($\texttt{fetched}$, $\texttt{prepared}$, $\texttt{computed}$, $\texttt{dequeued}$) and the four state elements ($\texttt{young}$, $\texttt{mature}$, $\texttt{executing}$, $\texttt{lingering}$) that serve as transaction holders in our processor's specification. In addition to these, we will also consider a few more defined "locations", some of which have previously been defined or just mentioned. First we have $\texttt{queue}^n = \texttt{mature}^n \,\#\, \texttt{young}^n$ and $\texttt{contents}^n = \texttt{executing}^n + \texttt{lingering}^n$, the full contents of the ordering and the execution units respectively. Then we have $\texttt{enqueued}^n$, a prefix of $\texttt{fetched}^n$, defined when n is regular and with properties given in Ord-Inv 2 and Ord-Inv 3. We define $\texttt{enqueued}^n = \langle\rangle$ when n is singular. Furthermore, we define $\texttt{ignored}^n$ by $\texttt{fetched}^n = \texttt{enqueued}^n \,\#\, \texttt{ignored}^n$ when n is regular, and $\texttt{ignored}^n = \langle\rangle$ when n is singular. $\texttt{ripe}^n$ is the transaction set consisting of complete transactions in $\texttt{mature}^n$. Finally, when n is regular, we define $\texttt{squashed}^n = \langle\rangle$, and when n is singular, we define $\texttt{squashed}^n$ to be the suffix of $\texttt{queue}^{n-1} \,\#\, \texttt{fetched}^{n-1}$ of length complementary to $|\texttt{dequeued}^n|$.

Note that the nine location names are used to name the nodes of the transaction diagram Γ in Fig. 3. If X and Y are two nodes of Γ we will write $X \le Y$ if $X = Y$ or there exists a sequence of arcs in Γ leading from X to Y. There are no non-trivial cycles in Γ, so this is a partial order relation.

Ancestors and ordinals. A simple fundamental observation is that any transaction present in the processor at any cycle in any of the eight basic locations except $\texttt{fetched}$ has a uniquely determined *immediate ancestor* among transactions present in the processor at the previous cycle. Note, however, that it is not realistic to assume that this relationship is "one-to-one". For example, in the model we are considering, each transaction in $\texttt{prepared}^n$ wire has a copy of itself saved in $\texttt{mature}^n$ and each transaction in $\texttt{executing}^n$ or $\texttt{computed}^n$ also has a copy of its ancestor waiting in $\texttt{mature}^n$. Choosing a unique "descendant" of a fetched instruction in all subsequent cycles is tantamount to the definition of the history table; see A.3.

Since the initial value X^1 is empty for every $X \ne \texttt{fetched}$, it follows that starting with any transaction T belonging to a location X^n one can define a sequence of transactions in which each is the immediate ancestor of the previous one and which terminates at a transaction T_0 belonging to $\texttt{fetched}^k$ for some $k \le n$. This T_0 is a uniquely defined progenitor of T. The *ordinal* of T is defined to be the ordinal of T_0 in the sequence $\texttt{all-fetched} = \texttt{fetched}^1 \,\#\, \texttt{fetched}^2 \,\#\, \cdots$ of all fetched transactions.

It remains to give a precise definition of immediate ancestors. So suppose X is a basic location, $X \ne \texttt{fetched}$, and $T \in X^n$. We define the ancestor T' of T and its location Y^{n-1}. Consider first the possibilites $\texttt{executing}$, $\texttt{lingering}$ and $\texttt{computed}$ for X. If $n-1$ is regular, then T' and Y are found from the inequality

$$\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1} \cup \texttt{lingering}^{n-1} \preceq \texttt{contents}^n \qquad (2)$$

of Exec-Inv 2. If $n-1$ is singular, then $\texttt{executing}^n$, $\texttt{lingering}^n$ and $\texttt{computed}^n$ are empty, so there is nothing to define. Turning to the possibilities $\texttt{young}$, mature, prepared and dequeued for X, we obtain the corresponding T' and Y easily from the relations

$$\texttt{young}^{n-1} \mathbin{\#} \texttt{enqueued}^{n} \preceq \texttt{prepared}^{n} \mathbin{\#} \texttt{young}^{n}, \tag{3}$$

$$\texttt{mature}^{*} \mathbin{\#} \texttt{prepared}^{n} = \texttt{dequeued}^{n} \mathbin{\#} \texttt{mature}^{n}. \tag{4}$$

of Ord-Inv 2, provided that n is regular. And if n is singular, then $\texttt{prepared}^n$, $\texttt{mature}^n$ and $\texttt{young}^n$ are empty (Ord-Inv 2) so there is nothing to do for them, while for $\texttt{dequeued}^n$ we have that it is a prefix of a sequence $\texttt{mature}^*$, where each member of $\texttt{mature}^*$ belongs to either $\texttt{mature}^{n-1}$ and $\texttt{computed}^{n-1}$.

Note that in all cases we have $T' \preceq T$.

A.2 Between processor units

From the informal specification of the ordering unit (Sect. 4) we expect that transactions in $\texttt{mature}^n$ should fall into four well-defined classes: for each T in $\texttt{mature}^n$, T is either complete and waiting for its turn to be dequeued, or there is a unique transaction associated (by name) with T in $\texttt{prepared}^n$, $\texttt{executing}^n$, or $\texttt{computed}^n$. Lemma 2 below confirms this basic relationship between the contents of the ordering and the execution units. Lemmas 3 and 4 state two important relationships between what comes in and what goes out. They refer to the execution unit and the ordring unit respectively, but neither can be derived from the axiomatics of a single unit.

First we need to extend our notation about transaction sets. Transaction sets are *disjoint* if their domains are disjoint as sets; we will write $A+B$ for $A \cup B$ in the case when we know A and B are disjoint. Define $A \setminus B$ to be the restriction of A on the set difference of the domains of A and B. Define A to be a *subset* of B if $A(x) = B(x)$ whenever $A(x) \neq ()$. We will write $A - B$ for $A \setminus B$ when we know that A is a subset of B.

Lemma 1. *If n and $n-1$ are regular, then*

$$\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1} \preceq \texttt{executing}^{n} + \texttt{computed}^{n}.$$

Proof. Since n is regular, Exec-Inv 2 implies

$$(\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1}) + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}) \preceq \texttt{executing}^{n} + \texttt{lingering}^{n}.$$

Since $n-1$ is regular, Exec-Inv 3 implies

$$\texttt{lingering}^{n} = \texttt{computed}^{n} + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}).$$

The lemma immediately follows from these relations. □

Lemma 2. *For every regular n, the sets $\texttt{ripe}^n$, $\texttt{computed}^n$, $\texttt{executing}^n$ and $\texttt{prepared}^n$ are disjoint, and*

$$\texttt{mature}^{n} \preceq \texttt{ripe}^{n} + \texttt{computed}^{n} + \texttt{executing}^{n} + \texttt{prepared}^{n}. \tag{5}$$

Moreover, the corresponding elements on the two sides have the same ordinal.

Proof. The proof is by induction. Since the initial values of all the sets involved are empty, the initial case is true. The induction step splits into two cases, depending on whether $n-1$ is regular or not.

Assume first $n-1$ is not regular. By Ord-Inv 2, we have $\mathtt{mature}^{n-1} = \mathtt{young}^{n-1} = \langle\rangle$ and then $\mathtt{prepared}^{n} = \mathtt{dequeued}^{n} \# \mathtt{mature}^{n}$. This implies $\mathtt{mature}^{n} = \mathtt{prepared}^{n}$ because all transactions in $\mathtt{dequeued}^{n}$ are complete and so cannot occur in $\mathtt{prepared}^{n}$, which (by Ord-Inv 4) contains only incomplete transactions. It remains only to prove that the sets $\mathtt{ripe}^{n}$, $\mathtt{computed}^{n}$ and $\mathtt{executing}^{n}$ are empty. For $\mathtt{ripe}^{n}$ it is true because all elements of $\mathtt{mature}^{n}$ are incomplete. The other two are subsets of $\mathtt{contents}^{n}$ which is empty by Exec-Inv 1.

Assume now that $n-1$ is regular. By Ord-Inv 2, we have

$$\mathtt{mature}^{*} + \mathtt{prepared}^{n} = \mathtt{dequeued}^{n} + \mathtt{mature}^{n}, \tag{6}$$

where $\mathtt{mature}^{*}$ is obtained by replacing every transaction in $\mathtt{mature}^{n-1}$ with an equally named transaction of $\mathtt{computed}^{n-1}$. By induction hypothesis, all names of $\mathtt{computed}^{n-1}$ occur among names of $\mathtt{mature}^{n-1}$, so we have

$$\mathtt{mature}^{*} = \mathtt{computed}^{n-1} + (\mathtt{mature}^{n-1} \setminus \mathtt{computed}^{n-1}). \tag{7}$$

Combining (6) and (7), and the induction hypothesis in the form

$$\mathtt{mature}^{n-1} \setminus \mathtt{computed}^{n-1} \preceq \mathtt{ripe}^{n-1} + \mathtt{executing}^{n-1} + \mathtt{prepared}^{n-1},$$

we obtain

$$\begin{aligned}\mathtt{dequeued}^{n} + \mathtt{mature}^{n} \preceq{} & \mathtt{computed}^{n-1} + \mathtt{ripe}^{n-1} + \mathtt{executing}^{n-1} \\ & + \mathtt{prepared}^{n-1} + \mathtt{prepared}^{n}.\end{aligned} \tag{8}$$

Observe now that $\mathtt{ripe}^{n-1} + \mathtt{computed}^{n-1}$ is the set of complete transactions in $\mathtt{mature}^{*}$; this follows from (7), the fact that all transactions in $\mathtt{computed}^{n-1}$ are complete, and the induction hypothesis implying that the complete transactions in $\mathtt{mature}^{n-1} \setminus \mathtt{computed}^{n}$ are precisely those of $\mathtt{ripe}^{n-1}$. Since no transaction of $\mathtt{prepared}^{n}$ is complete (Ord-Inv 4) and all transactions of $\mathtt{dequeued}^{n}$ are complete (Ord-Inv 5), it follows from (6) that the same set of complete transactions of $\mathtt{mature}^{*}$ can also be written as $\mathtt{dequeued}^{n} + \mathtt{ripe}^{n}$. Thus, (8) rewrites into

$$\begin{aligned}\mathtt{dequeued}^{n} + \mathtt{mature}^{n} \preceq{} & \mathtt{dequeued}^{n} + \mathtt{ripe}^{n} + \mathtt{executing}^{n-1} \\ & + \mathtt{prepared}^{n-1} + \mathtt{prepared}^{n},\end{aligned}$$

and the desired result follows immediately from Lemma 1.

It remains to go back and check that the ordinals are the same for any two correspondind members of the two sides of any equality and inequality that was used in the proof. This is done by a straighforward inspection. □

Lemma 3. *If n and $n-1$ are regular, then*

$$\mathtt{executing}^{n-1} + \mathtt{prepared}^{n-1} \preceq \mathtt{executing}^{n} + \mathtt{computed}^{n}.$$

Proof. This is a strengthening of Lemma 1; that $\texttt{prepared}^{n-1}$ and $exec^{n-1}$ are disjoint is a part of Lemma 2. □

Lemma 4. *If n and $n-1$ are regular, then*

$$\texttt{computed}^n + \texttt{ripe}^n = \texttt{dequeued}^{n+1} + \texttt{ripe}^{n+1} \tag{9}$$

and all transactions of this set belong to $\texttt{lingering}^{n+1}$.

Proof. The equation is proved in the course of proving Lemma 2. As in the proof of Lemma 1, we have

$$\texttt{lingering}^n = \texttt{computed}^n + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}),$$

so all we need to prove is that $\texttt{ripe}^n$ is a subset of $\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}$. Arguing by induction, the problem reduces to showing that the sets $\texttt{ripe}^n$ and $\texttt{prepared}^{n-1}$ are disjoint. Indeed, by Exec-Inv 2 and Exec-Inv 3, every transaction in $\texttt{prepared}^{n-1}$ has a descendant in $\texttt{executing}^n$ or $\texttt{computed}^n$, and by Lemma 2, these two sets are disjoint from $\texttt{ripe}^n$. □

A.3 Definition of the history table

The top row of Fig. 3 depicts all possible paths through selected processor locations that a normally completed transaction can have, form fetching through retiring. A transition from X to Y in most cases should be interpreted as "it is possible that a transaction in X^n has a corresponding transaction in Y^{n+1}". The diagram also suggests that all transactions in X^n should have a corresponding transaction in some Y^{n+1} for some Y, the target node of an arc coming from X. "Corresponding" here means having the same ordinal, *i.e.*, being related to the same fetched instruction. Our goal is to define the history of execution of any fetched instruction, so we would like to define "transitions" $(T, X) \rightsquigarrow (T', Y)$ with (T', Y) uniquely determined by (T, X). When more than one such transition is possible, we select the right one according to the values of "control variables" (`flush` in our case).

Transaction flow. The subgraphs of Γ defined in Figs. 5–7 represent the transaction flow between cycles n and $n+1$, depending on whether these numbers are regular or singular. The following lemma states this in precise terms.

Lemma 5. *Let $n \geq 2$ and*

$$\Gamma_n = \begin{cases} \Gamma_{rr} & \textit{if both } n-1 \textit{ and } n \textit{ are regular} \\ \Gamma_{rs} & \textit{if } n \textit{ is singular} \\ \Gamma_{sr} & \textit{if } n-1 \textit{ is singular} \end{cases}$$

and let

$$\begin{aligned} In_n &= \{(T, X) \mid T \in X^{n-1} \textit{ and } X \textit{ is the source of an arrow of } \Gamma_n\}, \\ Out_n &= \{(T', Y) \mid T' \in Y^n \textit{ and } Y \textit{ is the target of an arrow of } \Gamma_n\}. \end{aligned}$$

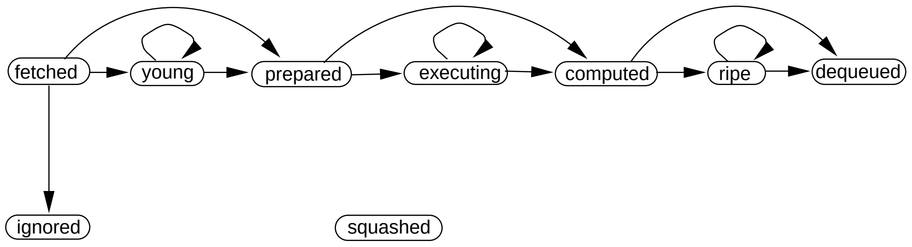


Fig. 5. Γ_{rr}.

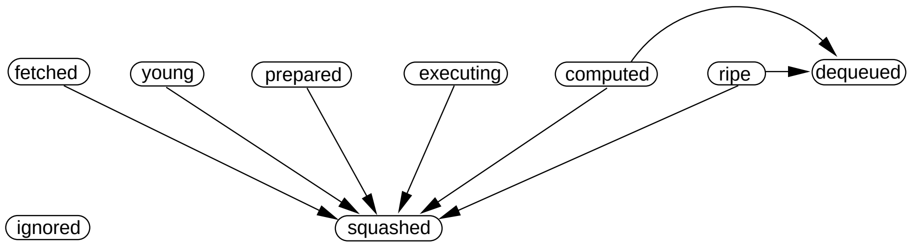


Fig. 6. Γ_{rs}.

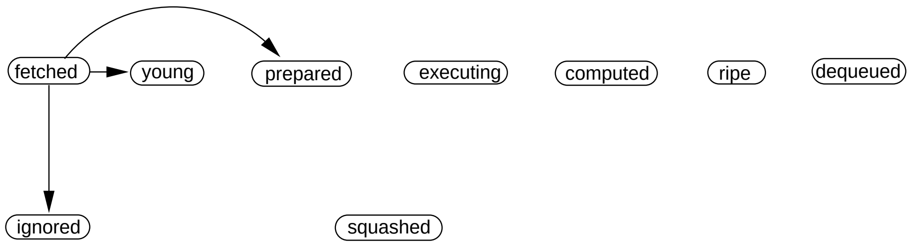


Fig. 7. Γ_{sr}.

Then the relation "have the same ordinal" defines a bijection $\delta_n : In_n \to Out_n$. Moreover, if $\delta_n(T,X) = (T',Y)$, then X and Y are joined by an arrow in Γ_n, and, in the cases $\Gamma_n = \Gamma_{rr}$ and $\Gamma_n = \Gamma_{sr}$, $T \preceq T'$.

Proof. We claim that if $n+1$ is regular, then

$$\texttt{young}^n \# \texttt{fetched}^n \preceq \texttt{prepared}^{n+1} \# \texttt{young}^{n+1} \# \texttt{ignored}^{n+1}, \qquad (10)$$

and if both n and $n+1$ are regular, then

$$\texttt{prepared}^n + \texttt{executing}^n \preceq \texttt{executing}^{n+1} + \texttt{computed}^{n+1}, \qquad (11)$$

$$\texttt{computed}^n + \texttt{ripe}^n = \texttt{dequeued}^{n+1} + \texttt{ripe}^{n+1}. \qquad (12)$$

Indeed, (10) follows from (3) and $\texttt{fetched}^n = \texttt{enqueued}^{n+1} \# \texttt{ignored}^{n+1}$, and (11) and (12) follow from Lemma 3 and Lemma 4 respectively. The case of the lemma when $\Gamma_n = \Gamma_{rr}$ immediately follows from these relations. Since $\texttt{young}^n = \langle\rangle$ when n is singular, the case $\Gamma_n = \Gamma_{rs}$ follows from (10) alone. Finally, in the case when $\Gamma_n = \Gamma_{sr}$ we have that $\texttt{squashed}^{n+1}$ is the suffix of $\texttt{mature}^n \# \texttt{young}^n \# \texttt{fetched}^n$ of length complementary to the length of $\texttt{dequeued}^{n+1}$ and that $\texttt{dequeued}^{n+1}$ is a prefix of $\texttt{mature}^*$, the sequence obtained by updating $\texttt{mature}^n$ with transactions of $\texttt{computed}^n$. The lemma now easily follows from Lemma 2. $\square$

History table. Recall the definition of ordinals of transactions. In particular, transactions in the sequence $\texttt{all-fetched} = \texttt{fetched}^1 \# \texttt{fetched}^2 \# \cdots$ have distinct ordinals. For every $i \geq 1$, we define the *nascency rank* $\texttt{nr}(i)$ to be the number n such that $\texttt{fetched}^n$ contains a transaction with ordinal i. (If $\texttt{all-fetched}$ is finite, then $\texttt{nr}(i)$ would be defined only for $i \leq |\texttt{all-fetched}|$, but we will prove that $\texttt{all-fetched}$ is infinite, so $\texttt{nr}$ is defined for every positive integer.)

Definition 4. *For a given run of the processor and every n and i such that $n \geq \texttt{nr}(i)$ define H^i_n inductively as follows:*

1. *If $n = \texttt{nr}(i)$, then $H^i_n = (T, \texttt{fetched})$, where T is the transaction in $\texttt{fetched}^n$ whose ordinal is i.*
2. *If $H^i_{n-1} = (T,X)$ and X is a final location, then $H^i_n = H^i_{n-1}$.*
3. *If $H^i_{n-1} = (T,X)$ and X is not final, then $H^i_n = \delta_{n-1}(H^i_{n-1})$.*

The history table *H is the table whose element belonging to the nth row and the ith column is H^i_n.*

The sequence of elements occuring in the nth row of H will be denoted by H_n. The transaction and the status components of H^i_n will be denoted T^i_n and X^i_n respectively. The sequence of transaction components of H_n will be denoted τ_n and the sequence of the status components of H_n will be denoted ξ_n.

Lemma 6. *The definition of the history table is correct.*

Proof. The only thing that needs to be checked is that if $H^i_{n-1} = (T, X)$ and X is not final, then (T, X) belongs to In_n, the domain of δ_n. If $X = \texttt{fetched}$, then this is obvious. Otherwise, $(T, X) = \delta_{n-1}(T', X')$, so $(T, X) \in Out_{n-1}$. Thus, X is a target of an arrow in Γ_{n-1} and (by inspection of the eight possibilities for Γ_{n-1} and Γ_n) it follows that X is a source of an arrow of Γ_n, finishing the proof. □

A.4 Basic properties of the history table

Lemma 7. *If H^i_n is defined then*

1. $X^i_n \le X^i_{n+1}$;
2. $T^i_n \preceq T^i_{n+1}$, *provided* $X^i_{n+1} \neq \texttt{squashed}$.

Proof. The proof follows immediately from Definition 4 and Lemma 5. □

Lemma 8. *All elements of Out_n occur in H_n.* □

Proof. The proof is obtained by strengthening the last argument in the proof of Lemma 6 by using bijectivity of δ_n. □

Note that the statuses related to the execution unit (`prepared`, `executing`, `computed`) do not occur in H_n when n is singular, so that the descendancy relation of Sect. 7 is not exactly reflected in the history table. In transitions between regular cycles, however, the descendancy in the execution unit can be seen in the table, as stated in the following lemma, easily derived from definitions.

Lemma 9. *If n and $n+1$ are both regular and X^i_n is* `prepared` *or* `executing`*, then T^i_{n+1} is the descendant of T^i_n (in the sense of Sect. 7).* □

Let **Act** denote the set consisting of the five nodes of Γ that are neither initial nor final. Let $\texttt{active}^n$ be the sequence obtained from τ_n by removing all transactions whose corresponding location is not in **Act**.

Lemma 10. *For every n, $\texttt{queue}^n \preceq \texttt{active}^n$. Moreover, if n is regular, then $\texttt{active}^n$ is the sequence obtained by replacing every transaction in $\texttt{mature}^n$ with an equally named transaction in the set $\texttt{computed}^n + \texttt{executing}^n$.*

Proof. Suppose first n is singular. Since $\texttt{queue}^n = \langle\rangle$, we need to show that $\texttt{active}^n = \langle\rangle$ too. Indeed, all elements of Out_n are of the form (T, X), where X is either `dequeued` or `squashed` (Fig. 6), and by Definition 4, the status of all elements in the nth row of H is final.

Suppose now n is regular. By Lemma 8, if $X \in \textbf{Act}$ and $T \in X^n$, then (T, X) occurs in H_n. Then, by Lemma 2, there exists a bijection $T \mapsto T'$ between elements of $\texttt{active}^n$ and $\texttt{queue}^n$ such that $T' \preceq T$. All that remains to prove is that the elements of $\texttt{queue}^n = \texttt{mature}^n \,\#\, \texttt{young}^n$ have increasing ordinals, and that follows easily from the definitions of ancestors and ordinals. □

Now we can derive an often used lemma that guarantees existence of regular number intervals.

Lemma 11. *If $X_n^i <$ `dequeued` and $\mathtt{nr}(i) < m < n$, then m is regular.*

Proof. Since $m > nr(i)$, we have $X_m^i >$ `fetched`. Since $m < n$, we have $X_m^i <$ `dequeued` (Lemma 7). Thus, $X_m^i \in \mathbf{Act}$ and so $\mathtt{active}^n \neq \langle\rangle$. Since $\mathtt{queue}^n = \langle\rangle$ when n is singular (Ord-Inv 2), the result follows from Lemma 10. □

The following is a stabilization lemma for columns of the history table.

Lemma 12. *For every i, the sequence H_n^i is eventually constant.*

Proof. Let $H_n^i = (T_n, X_n)$. By Lemma 7 $X_n \leq X_{n+1}$ in Γ. Since Γ is finite and the only cycles in it are loops at nodes, it follows that the sequence X_n stabilizes at n_0, say. Let X be its limit value. If X is final, then, again by Definition 4, H_n^i stabilizes as well. The remaining possibility is that X is `young`, `executing` or `ripe`. By Lemma 11, all numbers greater than n_0 are regular. By Lemma 5, we then have $T_n \preceq T_{n+1}$ for all $n > n_0$. By definition of progress ordering, all strictly increasing chains of transactions are finite, so the sequence T_n is eventually constant. □

The cycle at which the sequence H_n^i assumes its stable value will be denoted $\mathtt{sr}(i)$, the *stabilization rank*. The *limit row* H_∞ is the sequence of stable values: $H_\infty^i = \lim_n H_n^i$. The sequence of transactions and the sequence of statuses occurring in H_∞ will be denoted τ_∞ and ξ_∞.

Lemma 13. *The sequence* $\mathtt{dequeued}^{n+1}$ *is a prefix of* $\mathtt{active}^n$.

Proof. By Ord-Inv 2, $\mathtt{dequeued}^{n+1}$ is a prefix of $\mathtt{mature}^*$, the sequence obtained by replacing transactions in $\mathtt{mature}^n$ with equally named transactions of $\mathtt{computed}^n$. When n is singular, $\mathtt{dequeued}^{n+1}$ is empty because $\mathtt{mature}^n$ is empty. When n is regular, the corollary follows from Lemma 10. □

Let τ_n^+ be the sequence obtained from τ_n by deleting all members whose corresponding status is `ignored`. Let also τ_n^{DS} be the sequence obtained from τ_n by keeping only its members whose status is `dequeued` or `squashed`.

Lemma 14. $\tau_n^+ = \tau_n^{\mathsf{DS}} \, \# \mathtt{active}^n \, \# \mathtt{fetched}^n$.

Proof. The proof is by induction. Assume τ_n^+ has the given form. By Definition 4, $\mathtt{fetched}^{n+1}$ is a suffix of τ_{n+1}^+. The prefix τ_n^{DS} remains intact in τ_{n+1}^+, by the same definition.

By Lemma 13, $\mathtt{dequeued}^{n+1}$ occurs as a prefix in $\mathtt{active}^n$ and so, by Definition 4, it will occur at the corresponding places in τ_{n+1}. Therefore, the sequence $\tau_n^{\mathsf{DS}} \, \# \mathtt{dequeued}^{n+1}$ is a prefix of τ_{n+1}^+. Now, if $n+1$ is regular, then, for each T_n^i of $\mathtt{active}^n$ which does not occur in the prefix $\mathtt{dequeued}^{n+1}$, we have $X_{n+1}^i \in \mathbf{Act}$ (diagram Γ_{rr} or Γ_{sr}, though in the latter case there are no such elements T_n^i).

Also, if T_n^i is in $\texttt{fetched}^n$, then $X_{n+1}^i \in \mathbf{Act}$ or $X_{n+1}^i = \texttt{ignored}$. This finishes the proof if $n+1$ is regular. If $n+1$ is singular, then for every T_n^i in $\texttt{active}^n \,\#\, \texttt{fetched}^n$ that does not belong to the prefix $\texttt{dequeued}^{n+1}$, one has $X_{n+1}^i = \texttt{squashed}$ (diagram Γ_{rs}). $\square$

As an immediate consequence of this lemma and its proof, we obtain the following.

Lemma 15. *For every $n > 1$, $\tau_n^{\mathsf{DS}} = \tau_{n-1}^{\mathsf{DS}} \,\#\texttt{dequeued}^n \,\#\texttt{squashed}^n$. If n is singular, then $\tau_n^{\mathsf{DS}} = \tau_n^+$.* $\square$

Lemma 16. $\tau_n^{\mathsf{D}} = \texttt{dequeued}^1 \,\# \cdots \#\texttt{dequeued}^n$ *and τ_n^{D} is a prefix of τ_∞^{D}.*

Proof. By induction, using Lemma 15. $\square$

Let ξ_n^+ be the sequence obtained from ξ_n by deleting all members equal to $\texttt{ignored}$.

Lemma 17. *For every n, the sequence ξ_n^+ regarded as a string, belongs to the set defined by the regular expression*

$$\{\texttt{dequeued}, \texttt{squashed}\}^*\{\texttt{executing}, \texttt{computed}, \texttt{ripe}\}^*\{\texttt{prepared}\}^*\{\texttt{young}\}^*\{\texttt{fetched}\}^*.$$

Moreover, if n is singular, then the regular expression can be restricted to

$$\{\texttt{dequeued}, \texttt{squashed}\}^*\{\texttt{fetched}\}^*.$$

Proof. The lemma follows from Lemma 15 and a simple observation that the sequence $\texttt{prepared}^n \,\#\texttt{ignored}^n$ is a suffix of $\texttt{queue}^n$ that occurs also as a suffix in $\texttt{active}^n$ (see Lemma 10). $\square$

All transactions in $\texttt{fetched}^n$ have maximal values in their fields instr, addr and spc, and the field name is maximal in transactions of $\texttt{young}^n$. In $\texttt{prepared}^n$, all transactions have maximal values in their fields opcode, $\mathsf{rSources}$ and rDest. In $\texttt{computed}^n$ transactions are complete and so have maximal values in all fields. Combining these observations with Lemma 7, we obtain the following lemma, often used without being explicitly mentioned.

Lemma 18. *Fix i, let $p, q \geq \texttt{nr}(i)$ and denote $H_p^i = (T_p, X_p)$, $H_q^i = (T_q, X_q)$.*

1. $\mathsf{field}(T_p) = \mathsf{field}(T_q) \neq \bot$ *for any* $\mathsf{field} \in \{\mathsf{instr}, \mathsf{addr}, \mathsf{spc}\}$
2. *If $p, q > \texttt{nr}(i)$, then* $\mathsf{name}(T_p) = \mathsf{name}(T_q) \neq \bot$.
3. *If* $\texttt{prepared} \leq X_p, X_q \leq \texttt{dequeued}$*, then* $\mathsf{field}(T_p) = \mathsf{field}(T_q) \neq \bot$ *for any* $\mathsf{field} \in \{\mathsf{opcode}, \mathsf{rSources}, \mathsf{rDest}\}$
4. *If* $\texttt{computed} \leq X_p, X_q \leq \texttt{dequeued}$*, then $T_p = T_q$.*

$\square$

Lemma 19. *If* $i \le j$, *then* $\mathtt{nr}(i) \le \mathtt{nr}(j)$. *If* $i \le j$ *and* $X^j_\infty \ne \mathtt{ignored}$, *then* $\mathtt{sr}(i) \le \mathtt{sr}(j)$.

Proof. The first statement is an immediate consequence of the definition of $\mathtt{nr}$. For the second statement, if $X^i_\infty = \mathtt{ignored}$, then $\mathtt{sr}(i) = \mathtt{nr}(i) + 1$ and $\mathtt{sr}(i) < \mathtt{sr}(j)$ easily follows. The interesting case, when neither of X^i_∞, X^j_∞ is $\mathtt{ignored}$, follows from Lemma 16. □

Lemma 20. *Suppose* $i < j$ *and* $X^i_n, X^j_n \in \mathbf{Act}$. *Then* $X^j_\infty = \mathtt{dequeued}$ *implies that* $X^i_\infty = \mathtt{dequeued}$.

Proof. Suppose the lemma is not true. By Lemma 17, we must have $X^i_n = \mathtt{squashed}$. Consequently, $\mathtt{sr}(i)$ is singular. By Lemma 19, $n < \mathtt{sr}(i) \le \mathtt{sr}(j)$. Lemma 11 discards all but the possibility $\mathtt{sr}(i) = \mathtt{sr}(j)$. This, however, contradicts the definition of $\mathtt{squashed}^n$ (dequeued transactions precede transactions squashed at the same cycle). □

A.5 Proof of Proposition 1

Lemma 21. $\Box\Diamond\ \mathtt{fetched} \ne \langle\rangle$.

Proof. Assume the contrary: $\Diamond\Box\ \mathtt{fetched} = \langle\rangle$. It follows then from (10) that $\Diamond\Box\ \mathtt{prepared} = \langle\rangle$. Then (11) implies $\Diamond\Box\ \mathtt{computed} = \langle\rangle$, and then (12) implies $\Diamond\Box\ \mathtt{dequeued} = \langle\rangle$. Now from Ord-Inv 9 we deduce $\Diamond\Box\ \mathtt{rpc} = ()$ and reach a contradiction with Fetch-Liv. □

Lemma 22. $\Box\Diamond\ \mathtt{queue} \ne \langle\rangle$.

Proof. Assume, on the contrary, that $\Diamond\Box\ \mathtt{queue} = \langle\rangle$. Then, by Ord-Inv 2, $\Diamond\Box\ \mathtt{dequeued} = \langle\rangle$ and then, by Ord-Inv 7, $\Diamond\Box\ \mathtt{flush} = \textsc{false}$. Also by Ord-Inv 2, $\Diamond\Box\ \mathtt{enqueued} = \langle\rangle$. By Lemma 21, there exists i such that $\mathtt{fetched}^i \ne \langle\rangle$, while $\mathtt{queue}^k = \mathtt{enqueued}^k = \mathtt{dequeued}^k = \langle\rangle$ and $\mathtt{flush}^k = \textsc{false}$ for all $k \ge i$. Let $x = \mathtt{rpc}^{i+1}$. By Ord-Inv 9, $x \ne ()$, while $\mathtt{rpc}^{i+1} = ()$ for all $k > i + 1$. By Ord-Inv 10, $\mathtt{xpc}^k = x$ for all $k > i$. Now let j be the smallest number greater than i such that $\mathtt{fetched}^j \ne \langle\rangle$; it exists by Lemma 21. We have $\mathtt{pc\text{-}rpc}^{i+1} = x$ and, by repeated application of Fetch-Inv 3, $\mathtt{pc\text{-}rpc}^j = x$ as well. If T is the first transaction of $\mathtt{fetched}^j$, then $\mathtt{addr}(T) = x$ (Fetch-Inv 2), and then Ord-Inv 3 implies that $\mathtt{enqueued}^{j+1} \ne \langle\rangle$, which is a contradiction. □

Lemma 23. *All locations occurring in the entries of* H_∞ *are final.*

Proof. Since each of $\mathtt{fetched}$, $\mathtt{prepared}$ and $\mathtt{computed}$ can occur at most once in any given column of the history table, none of them can occur in ξ_∞. We need to eliminate the possibility of occurrences of $\mathtt{young}$, $\mathtt{executing}$ and $\mathtt{ripe}$. Assuming the contrary, let k be the smallest integer such that $X^k_\infty = X$ is one of these three and let $m = \mathtt{sr}(k)$. Then $H^k_\infty = (T, X)$ for some T and, by Lemma 10, $\mathtt{queue}^n$ begins with a transaction T_n such that $T_n \preceq T$, for all $n > m$.

Case 1: $X = \texttt{young}$. Now we have $\texttt{young}^m \neq \langle\rangle$ and so $\texttt{mature}^n = \langle\rangle$ for all $n \geq m$. This implies $\texttt{prepared}^n = \langle\rangle$ for all $n \geq m$, directly contradicting Ord-Liv.
Case 2: $X = \texttt{ripe}$. We have $\texttt{dequeued}^n = \langle\rangle$ for all $n \geq m$. By Lemma 2, T_m equals T and so is complete. This again contradicts Ord-Liv.
Case 3: $X = \texttt{executing}$. Note first that, by Ord-Inv 8, $\texttt{writemem}^m = \textsc{true}$ if T is a store; indeed, T_n is a store and is not complete since that would imply $X = \texttt{ripe}$. Secondly, by Lemma 11, all numbers greater than m are regular. Thirdly, since $\texttt{executing}^n$ and $\texttt{computed}^n$ are disjoint, $\mathsf{name}(T)$ does not occur in $\texttt{computed}^n$ for any $n \geq m$. These three facts, combined with Exec-Liv imply that T is not independent. Thus, we have (1) $\mathsf{rOp}_i(T) = \perp$ for some i, or (2) $\mathsf{mrSt}(T) \neq \textsc{none}$. Since $m = \texttt{sr}(k)$ and $X^k_m = \texttt{computed}$, it follows that $X^k_{m-1} = \texttt{prepared}$.

If (1) holds, then by Ord-Inv 4, there exists U in $\texttt{queue}^{m-1}$ such that $\mathsf{rProv}_i(T^k_{m-1}) = \mathsf{name}(U)$. If (2) holds, then, again by Ord-Inv 4, there exists U in $\texttt{queue}^n$ such that $\mathsf{mrSt}(T^k_{m-1}) = \mathsf{name}(U)$. In both cases we have that T^k_{m-1} depends on T^j_n for some $j < k$. Since $j < k$ and all numbers greater than n are regular, we have $X^j_\infty = \texttt{dequeued}$ and so there exists n such that T^j_n is in $\texttt{computed}^n$. Then T^j_n also occurs in $\texttt{lingering}^n$ (Exec-Inv 3). By Lemma 18, $\mathsf{name}(T^j_n) = \mathsf{name}(T^j_{m-1})$, so $T^k_n = T$ depends on T^j_n, contradicting the axiom that a transaction in $\texttt{executing}^n$ cannot depend on any transaction in $\texttt{lingering}^n$ (Exec-Inv 3). □

Proof of Proposition 1. If i is the ordinal of a transaction in $\texttt{queue}^n$, Lemma 23 implies that X^i_∞ is either $\texttt{dequeued}$ or $\texttt{squashed}$. It follows then from Lemma 22 that ξ_∞ contains infinitely many entries equal to $\texttt{dequeued}$ or $\texttt{squashed}$. In other words, the sequence $\tau^{\mathsf{DS}}_\infty$ is infinite. By Lemma 15, this sequence is the concatenation of all sequences $\texttt{dequeued}^n \# \texttt{squashed}^n$. Since $\texttt{dequeued}^n \neq \langle\rangle$ whenever $\texttt{squashed}^n \neq \langle\rangle$ (by definition of $\texttt{squashed}$ and Ord-Inv 7), it follows that $\texttt{dequeued}^n \neq \langle\rangle$ for infinitely many values for n, and therefore ξ^{D}_∞ is infinite.

A.6 Proof of Proposition 2

Let T^i_∞ and T^j_∞ be two consecutive elements of τ^{D}_∞. We need to prove that $\mathsf{npc}(T^i_\infty) = \mathsf{addr}(T^j_\infty)$. Let $m = \texttt{sr}(i)$, $n = \texttt{sr}(j)$, $m' = \texttt{nr}(i)$, and $n' = \texttt{nr}(j)$; by Lemma 19, we have $m \leq n$ and $m' \leq n'$.

First we show that every p between m and n (if it exists) is regular. Assume the contrary: there exists a singular p such that $m < p < n$. Then $\texttt{dequeued}^p \neq \langle\rangle$, by Ord-Inv 5. Thus, there exists l such that $\texttt{sr}(l) = p$ and $X^l_\infty = \texttt{dequeued}$. By Lemma 19, it follows that $i < l < j$, contradicting the assumption that T^i_∞ and T^j_∞ are consecutive in τ^{D}_∞.

Assume first that T^i_∞ is not mispredicting; the other case will be considered separately. Since now $\mathsf{npc}(T^i_\infty) = \mathsf{spc}(T^i_\infty)$, all we need to show is $\mathsf{spc}(T^i_\infty) = \mathsf{addr}(T^j_\infty)$. If $m' = n'$ then $T^i_{m'}$ and $T^j_{m'}$ are members of $\texttt{fetched}^{m'}$ and both belong to $\texttt{enqueued}^{m'+1}$. Using Lemma 20, we deduce that these two transactions must be consecutive in $\texttt{fetched}^{m'}$.Therefore, $\mathsf{spc}(T^i_{m'}) = \mathsf{addr}(T^j_{m'})$, by Fetch-Inv 2. Now assume $n' > m'$. Then $T^i_{m'+1}$ is the last element of $\texttt{enqueued}^{m'+1}$, $T^j_{n'+1}$ is the first element of $\texttt{enqueued}^{n'+1}$, and $\texttt{enqueued}^r = \langle\rangle$ for all r between $m'+1$ and $n'+1$ (if there are any such r). We claim that all r between m' and n' are regular. We know that all numbers between m' and m are regular (Lemma 11), and that all numbers between m and n are regular (proved above). Thus, the claim fails only if m is singular and $n' > m$. Then T^i_m would be the last transaction in $\texttt{dequeued}^m$ (because $n > m$ now and $\texttt{dequeued}^r = \langle\rangle$ for all r between m and n), contradicting the assumption that T^i_∞ is not mispredicting.

We can conclude that $\texttt{xpc}^{m'+1} = \mathsf{spc}(T^i_{m'})$ from Ord-Inv 10, that $\texttt{xpc}^{m'+1} = \cdots = \texttt{xpc}^{n'}$ (also from Ord-Inv 10), and that $\texttt{xpc}^{n'} = \mathsf{addr}(T^j_{n'+1})$ from Ord-Inv 3. This finishes the proof in the case when T^i_∞ is not mispredicting.

Assume finally that T^i_∞ is mispredicting. It follows from Ord-Inv 5 that m is singular and also that $\texttt{xpc}^{m+1} = \mathsf{npc}(T^i_m)$ (Ord-Inv 9 and Ord-Inv 10). It follows also that $n' \geq m$; otherwise T^j_m would exist and would be in $\texttt{active}^m$, which is absurd because this sequence must be empty since m is singular.

It follows that $T^j_{n'+1}$ is the first element of $\texttt{enqueued}^{n'+1}$ and that $\texttt{enqueued}^r = \langle\rangle$ for every r between m and $n'+1$. We already know that the numbers between m and $n'+1$ are all regular, and it follows from the Ord-Inv axioms, similarly as in the previous case, that $\texttt{xpc}^{m+1} = \cdots = \texttt{xpc}^{n'+1} = \mathsf{addr}(T^j_{n'+1})$.

We also need to prove that $\mathsf{addr}(T^1_\infty) = \texttt{pc}_{init}$. We do have $\texttt{pc}^1 = \texttt{pc}_{init}$ by Fetch-Init. Let $n = \texttt{nr}(1)$. By Fetch-Inv 2, $\mathsf{addr}(T^1_n) = \texttt{pc}^{n-1}$. Since $\mathsf{addr}(T^1_\infty) = \mathsf{addr}(T^1_n)$ (Lemma 18), it suffices to check that $\texttt{pc}^{n-1} = \texttt{pc}^1$. In view of Fetch-Inv 3, this reduces to proving $\texttt{rpc}^m = ()$ for $1 \leq m < n-1$. The last claim is a consequence of Ord-Inv 9 and simple facts $\texttt{flush}^m = \text{FALSE}$ and $\texttt{queue}^m = \langle\rangle$ for all $m < n-2$.

A.7 Proof of Proposition 3

Lemma 24. *For every* $\texttt{reg} \in \mathbf{Reg}$ *and* $n \geq 1$,

$$\texttt{rf}^n(\texttt{reg}) = \begin{cases} \mathsf{rRes}(T) & \textit{if } T \textit{ is the last element in } \tau^{\mathsf{D}}_n \textit{ such that } \mathsf{rDest}(T) = \texttt{reg} \\ \texttt{rf}_{init}(\texttt{reg}) & \textit{if such } T \textit{ does not exist} \end{cases}.$$

Proof. The proof follows from Lemma 16 and Ord-Inv 6. □

Denote by τ°_n the sequence obtained from τ_n by removing all its elements τ^i_n such that X^i_n is $\texttt{ignored}$ or $\texttt{squashed}$.

Lemma 25. *Let* $\textsf{prepared} \leq X^i_m, X^i_n \leq \textsf{dequeued}$. *If* $m < n$ *and* T^j_n *is the* r^{th} *register provider of* T^i_n *in* τ°_n*, then* T^j_m *is the* r^{th} *provider of* T^i_m *in* τ°_m*. Also, if* T^i_n *does not have the* r^{th} *register provider in* τ°_n*, then* T^i_m *does not have the* r^{th} *provider of in* τ°_m*.*

Proof. We prove only the first assertion of the lemma. The proof of the second is analogous.

Since $X^i_m \leq X^i_n$ and $X^j_m \leq X^j_n$, neither of X^i_m, X^j_m is $\texttt{ignored}$ or $\texttt{squashed}$, so T^i_m, T^j_m are in τ°_m. By Lemma 18, $\mathsf{rSource}_r(T^i_m) = \mathsf{rSource}_r(T^i_n)$ and $\mathsf{rDest}(T^j_m) = \mathsf{rDest}(T^j_n)$. Thus, $\mathsf{rSource}_r(T^i_m) = \mathsf{rDest}(T^j_m)$.

Suppose now k is such that $j < k < i$, T^k_m is in τ°_m, and $\mathsf{rSource}_r(T^i_m) = \mathsf{rDest}(T^k_m)$. Again, by Lemma 18, we have $\mathsf{rSource}_r(T^i_n) = \mathsf{rDest}(T^k_n)$, so T^k_n does not belong to τ°_n. Thus, X^k_n is $\texttt{ignored}$ or $\texttt{squashed}$. The only possibility for $X^k_n = \texttt{ignored}$ would be that $m = n-1$ and $X^k_m = \texttt{fetched}$, but that contradicts Lemma 17. If $X^k_n = \texttt{squashed}$, then it would follow that there exists p such that T^k_p belongs to $\texttt{squashed}^p$. This would imply that T^i_p belongs to $\texttt{squashed}^p$, which is not true. $\square$

Proof of Proposition 3. Suppose T and U are transactions in τ^{D}_∞ such that U is the r^{th} register provider of T. Let i and j be the ordinals of T and U respectively. Denote $H^i_k = (T_k, X_k)$ and $H^j_k = (U_k, Y_k)$. Let $n = \texttt{sr}(i)$ and let m be the unique integer such that $X_m = \texttt{prepared}$. From Lemma 11 we have that every k such that $m \leq k < n$ is regular.

By Lemma 16, U_n is the r^{th} provider of T_n in τ^{D}_n. Then, by Lemma 25, U_m is the r^{th} provider of T_m in τ°_m. By Lemma 17, $Y_m \geq \texttt{prepared}$. Note also that, being an element of $\texttt{prepared}^m$, T_m belongs to $\texttt{queue}^m$.

Case 1: $Y_m = \texttt{dequeued}$. By Lemma 17, T_m does not have an r^{th} register provider in $\texttt{active}^m$. It follows, using Lemma 10, that T_m does not have an r^{th} provider in $\texttt{queue}^m$. Let $\texttt{reg} = \mathsf{rSource}_r(T_m)$. By Ord-Inv 4, $\texttt{reg} \neq \bot$ and $\mathsf{rOp}_r(T_m) = \texttt{rf}^m(\texttt{reg})$. Since U_m is the r^{th} provider of T_m in τ°_m, it follows that U_m is the last transaction in τ^{D}_m whith rDest field equal to $\texttt{reg}$. It follows from Lemma 24 that $\mathsf{rOp}_r(T_m) = \mathsf{rRes}(U_m)$ and so $\mathsf{rSources}(T) = \mathsf{rRes}(U)$, as required.

Case 2: $Y_m \neq \texttt{dequeued}$. Now Y_m belongs to $\texttt{active}^m$. Since U_m is the r^{th} provider of T_m in τ°_m, it follows that U_m is the r^{th} provider of T_m in $\texttt{active}^m$ as well. From Lemma 10 we deduce that U'_m is the r^{th} provider of T_m in $\texttt{queue}^m$, where $U'_m \preceq U_m$. It follows from Ord-Inv 4 that $\mathsf{rOp}_r(T_m) = \bot$ and $\mathsf{rProv}_r(T_m) = \mathsf{name}(U'_m)$, which immediately implies $\mathsf{rProv}_r(T_m) = \mathsf{name}(U_m)$.

Since $T_m \preceq \cdots \preceq T_n$, $\mathsf{rOp}(T_m) = \bot$ and $\mathsf{rOp}(T_n) \neq \bot$, there exists a unique number p such that $m \leq p < n$, $\mathsf{rOp}(T_p) = \bot$, and $\mathsf{rOp}(T_{p+1}) \neq \bot$. Since T_p is incomplete, it belongs to $\texttt{executing}^p$ or $\texttt{prepared}^p$. From Lemma 9 we conclude that T_{p+1} is the descendant of T_p. Furthermore, Exec-Inv 5 implies that there exists a transaction V in $\texttt{executing}^p \cup \texttt{lingering}^p$ such that $\mathsf{rProv}_r(T_p) = \mathsf{name}(U)$ and $\mathsf{rOp}_r(T_{p+1}) = \mathsf{rRes}(V) \neq \bot$. It follows that $\mathsf{name}(V) = \mathsf{name}(U_m)$. We claim that $V = U_p$, which then implies $\mathsf{rOp}_r(T) = \mathsf{rOp}_r(T_{p+1}) = \mathsf{rRes}(U_p) = \mathsf{rRes}(U)$, finishing the proof.

Suppose the claim is not true. Then U_p cannot belong to $\texttt{active}^p$ because this sequence contains V and cannot contain two transactions with the same name. It follows that $X^j_p = \texttt{dequeued}$, so there exists q such that $p < q \leq m$ and U_q is in $\texttt{computed}^q$ and so in $\texttt{lingering}^q$. Since T_q belongs to $\texttt{executing}^q$

and $\mathsf{rProv}_r(T_q) = \mathsf{name}(U_q)$, it follows that T_q depends on U_q. This contradicts Exec-Inv 3, finishing the proof of the claim.

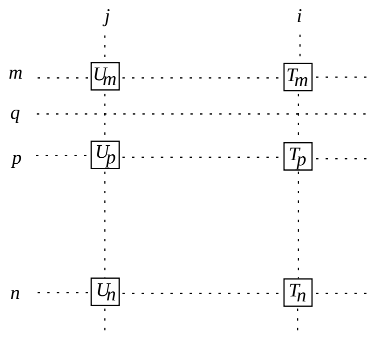


Fig. 8. Transactions involved in the resolution of a register dependency (Case 2 of the proof of Proposition 3).

We also need to prove $\mathsf{rOp}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$ in the case when $T = T^i_\infty$ does not have the r^{th} register provider in τ^{D}_∞. Let again m be the integer such that $X_m = \mathtt{prepared}$. By Lemma 25, T^i_m does not have the r^{th} provider in τ°_m. As in Case 1 above, we obtain $\mathsf{rOp}_r(T^i_m) = \mathtt{rf}^m(\mathtt{reg})$, where $\mathtt{reg} = \mathsf{rSource}_r(T^i_m)$. Using Lemma 24, we deduce $\mathtt{rf}^m(\mathtt{reg}) = \mathtt{rf}_{init}(\mathtt{reg})$, finishing the proof.

A.8 Proof of Proposition 4

Lemma 26. *Every load in* $\mathtt{prepared}^n + \mathtt{executing}^n$ *satisfies the condition* (LC).

Proof. By Ord-Inv 4, the mOp field of every load in $\mathtt{prepared}^n$ is $\perp$, so (LC) is true for such loads. Furthermore, every load in $\mathtt{executing}^n$ is a descendant of a load in $\mathtt{prepared}^{n-1} \cup \mathtt{executing}^{n-1}$, so by induction on n and using Exec-Inv 4, it follows that these loads also satisfy (LC). □

Lemma 27. *If* $\mathtt{computed}^n$ *contains a store, then this store is the first transaction in* $\mathtt{active}^n$*.*

Proof. Suppose the lemma is not true and pick the minimal n that provides a counter-example. Suppose T^i_n is a store in $\mathtt{computed}^n$ and T^j_n is the first transaction in $\mathtt{active}^n$, and $j < i$. Pick i so that $i - j$ is smallest.

Let U be the first transaction of $\mathtt{queue}^n$. By Lemma 10 $U \preceq T^j_n$. By By Exec-Inv 6, $\mathtt{writemem}^n = \textsc{true}$ and by Ord-Inv 8, U is an incomplete store. Using Lemma 17, we conclude that T^j_n belongs to $\mathtt{executing}^n$.

Let m be such that T^i_m belongs to $\mathtt{prepared}^m$. By Lemma 19, T^j_m belongs to $\mathtt{active}^m$ and so, by Ord-Inv 4, $\mathsf{mrSt}(T^i_m) = \mathsf{name}(T^k_m)$ for some k such that

$j \leq k < i$. Since $\mathsf{mrSt}(T^i_n) \neq \mathsf{mrSt}(T^i_m)$, it follows from Exec-Inv 7 that for some p such that $m < p \leq n$ one has T^k_p in $\mathtt{computed}^p$.

Since T^j_n is in $\mathtt{executing}^n$, T^j_p is not in $\mathtt{computed}^p$, so $k \neq j$. Thus, T^k_p is a store in $\mathtt{computed}^p$ and is not a first transaction in $\mathtt{active}^p$. By minimality of n, we have $p = n$ and then a contradiction with the minimality assumption on i. □

Corollary 28. *If* $\mathtt{active}^n$ *contains a complete store, then it is its first transaction. If* $\mathtt{dequeued}^n$ *contains a store, then it contains only one and it is its first transaction.* □

Proof. The first statement is an immediate consequence of Lemma 28. For the second, use also Lemma 13 □

Corollary 29. *If* $\mathtt{mem}^{n+1} \neq \mathtt{mem}^n$ *then the first transaction S in* $\mathtt{active}^n$ *is a complete store in* $\mathtt{executing}^n$ *and* $\mathtt{mem}^{n+1} = \mathtt{mem}^n \cdot S$.

Proof. The proof follows directly from Exec-Inv 6 and Corollary 28. □

Lemma 30. *If* $\mathtt{computed}^n + \mathtt{ripe}^n$ *contains a store S, then* $\mathtt{mem}^n = \mathtt{mem}^n \cdot S$.

Proof. Suppose $S = T^i_n$. Then, for some $m \leq n$, $T^i_m = S$ is in $\mathtt{computed}^m$, and so, by Exec-Inv 6, $\mathtt{mem}^m = \mathtt{mem}^{m-1} \cdot S$. Therefore, $\mathtt{mem}^m \cdot S = \mathtt{mem}^m$. For every p such that $m < p \leq n$, T^i_p is in $\mathtt{ripe}^p$ and is the first transaction in $\mathtt{active}^p$. It follows from Corollary 29, that $\mathtt{mem}^p = \mathtt{mem}^m$ for all such p. In particular, $\mathtt{mem}^n = \mathtt{mem}^m$ and the lemma follows. □

Lemma 31. *For every n,* $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n$ *or* $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot S$*, where S is a store and is a first transaction of* $\mathtt{active}^n$.

Proof. We argue by induction. The initial case is clearly true. For the induction step, suppose first that $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n$. If $\mathtt{dequeued}^{n+1}$ is non-empty and contains no store, then $\mathtt{mem}^{n+1} = \mathtt{mem}^n$ by Lemma 29, and $\mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_{n+1} = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_{n+1}$ is clear. If $\mathtt{dequeued}^{n+1}$ contains a store S, then by Lemma 28, $\mathtt{dequeued}^{n+1}$ begins with S and contains no other stores. Being an element of $\mathtt{dequeued}^{n+1}$, S belongs to $\mathtt{computed}^n$ or $\mathtt{ripe}^n$ (Eq. 12), so by Lemma 29, $\mathtt{mem}^{n+1} = \mathtt{mem}^n$. On the other hand, Lemma 30 implies $\mathtt{mem}^n = \mathtt{mem}^n \cdot S$ and so $\mathtt{mem}^n = \mathtt{mem}^n \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_{n+1}$. Finally, if $\mathtt{dequeued}^{n+1}$ is empty, then $\tau^{\mathsf{D}}_{n+1} = \tau^{\mathsf{D}}_n$ and both $\mathtt{mem}^{n+1} = \mathtt{mem}^n$ and $\mathtt{mem}^{n+1} \neq \mathtt{mem}^n$ are possible. The desired result in the first case follows immediately, and in the second case it follows from Lemma 29.

Assume now the second possibility for the inductive hypothesis: $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot S$, where S is a store and is a first transaction of $\mathtt{active}^n$. Lemma 29 implies $\mathtt{mem}^{n+1} = \mathtt{mem}^n$. If $\mathtt{dequeued}^{n+1}$ is empty, the result immediately follows. If $\mathtt{dequeued}^{n+1}$ is non-empty, then it begins with S and contains no other stores, so $\mathtt{mem}^{n+1} = \mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot S = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau^{\mathsf{D}}_{n+1}$. □

Lemma 32. *If T is a load or store in* $\texttt{prepared}^n + \texttt{executing}^n$ *and U is the most recent store of T in* $\texttt{active}^n$*, then either (1) U is in* $\texttt{prepared}^n + \texttt{executing}^n$ *and* $\mathsf{mrSt}(T) = \mathsf{name}(U)$*, or (2) U is in* $\texttt{computed}^n + \texttt{ripe}^n$ *(and therefore is the first transaction in* $\texttt{active}^n$*) and* $\mathsf{mrSt}(T) = \text{NONE}$*.*

Proof. Let $T^i_n = T$ and $T^j_n = U$. Let m be such that T^i_m is in $\texttt{prepared}^m$. Then T^j_m is in $\texttt{active}^m$ and so $\mathsf{mrSt}(T^i_m) = \mathsf{name}(T^k_m)$, where T^k_m is the most recent store for T^i_m in $\texttt{active}^m$. We claim that $k = j$. Otherwise, using Exec-Inv 7 we would obtain T^k_p in $\texttt{computed}^p$ for some $p \leq n$, contradicting Corollary 28. The lemma now follows from Exec-Inv 7 and Corollary 28. □

Corollary 33. *Let T be a load or store in* $\texttt{prepared}^n + \texttt{executing}^n$ *and let ϕ be the store chain of T in this set. Then ϕ is equal to the sequence of stores in* $\texttt{prepared}^n + \texttt{executing}^n$ *that precede T in* $\texttt{active}^n$*.* □

Lemma 34. *Let L be a load in* $\texttt{prepared}^n + \texttt{executing}^n$ *and let ψ be the prefix of* $\texttt{active}^n$ *consisting of transactions preceding L. Then* $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \psi)(\mathsf{mSource}(L))$*.*

Proof. By Lemma 26, $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \phi)(\mathsf{mSource}(L))$, where ϕ is the store chain of L in $\texttt{prepared}^n + \texttt{executing}^n$. Let ψ_0 be the sequence obtained by deleting from ψ all transactions which are not stores. Clearly, $\texttt{mem}^n \cdot \psi = \texttt{mem}^n \cdot \psi_0$. By Corollary 28, all transactions in ψ_0 are in $\texttt{prepared}^n + \texttt{computed}^n$, except possibly the first store (say, S), which may belong to $\texttt{computed}^n + \texttt{ripe}^n$. By Lemma 33, we have $\psi_0 = \phi$ in the first case, and $\psi_0 = \langle S \rangle \mathbin{\#} \phi$ in the second. By Lemma 30, $\texttt{mem}^n \cdot \phi = \texttt{mem}^n \cdot \psi_0$, finishing the proof. □

Lemma 35. *Let α and β be transaction sequences such that $\alpha \preceq \beta$. Let* $\texttt{mem}$ *be an element of type* **IAddr** $\rightarrow$ **Value** *and* $\texttt{addr}$ *an element of type* **IAddr***. Then* $(\texttt{mem} \cdot \alpha)(\texttt{addr}) \preceq (\texttt{mem} \cdot \beta)(\texttt{addr})$*.*

Proof. By direct examination. □

Proof of Proposition 4. Suppose L is a load in τ^{D}_∞. Let θ be the prefix of τ^{D}_∞ consisting of transactions that precede L. We will prove that $\mathsf{mOp}(L) = (\texttt{mem}_{init} \cdot \theta)(\mathsf{mSource}(L))$. It is easy to see that this would imply Proposition 4.

Let i and be such that $L = T^i_\infty$ and and let n be the largest number such that T^i_n is in $\texttt{prepared}^n + \texttt{executing}^n$. Thus, T^i_{n+1} is in $\texttt{computed}^{n+1}$ and it follows from Exec-Inv 7 that $\mathsf{mOp}(T^i_n) \neq \perp$. Thus, $\mathsf{mOp}(T^i_n) = \mathsf{mOp}(L)$ and $\mathsf{mSource}(T^i_n) = \mathsf{mSource}(L)$.

From Lemma 34 we then obtain $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \psi)(\mathsf{mSource}(L))$, where ψ is the prefix of $\texttt{active}^n$ consisting of transactions preceding L. By Lemma 31, $\texttt{mem}^n$ is equal to either $\texttt{mem}_{init} \cdot \tau^{\mathsf{D}}_n$ or $\texttt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot S$, where the store S is the first transaction of $\texttt{active}^n$. Since ψ is a prefix of $\texttt{active}^n$ (and L is not a store), it follows that $\texttt{mem}^n \cdot \psi = \texttt{mem}_{init} \cdot \tau^{\mathsf{D}}_n \cdot \psi$.

By Lemma 20, all transactions of ψ are eventually dequeued. Thus, $\tau^{\mathsf{D}}_n \mathbin{\#} \psi \preceq \theta$. Using Lemma 35, we finally obtain $\mathsf{mOp}(L) \preceq (\texttt{mem}_{init} \cdot \theta)(\mathsf{mSource}(L))$, which must be equality because $\mathsf{mOp}(L) \neq \perp$.