Notes on Safety, Liveness, Simulation, Bisimulation
CS 6110, Formal Methods in System Design
December 4, 2007

1 Safety and Liveness

We worked with pretty acceptable definitions of safety and liveness so far. It pays to see some more depth wrt these definitions.

Alpern/Schneider, "Defining Liveness," Info.Proc.Let.,21,4(Oct'85):

Safety: Let S be a set of program states, S^ω the set of infinite sequences of program states, and S^* the set of finite sequences of program states. An execution of any program can be modeled as a member of S^ω. Members of S^* are partial executions. We write $\sigma \models P$ when execution σ is in property P. σ_i denotes the partial execution consisting of the first i states in σ.

P is a safety property if and only if

$$(\forall \sigma \in S^\omega : \ \sigma \not\models P \Rightarrow \exists i : \ 0 \leq i : \ (\forall \beta : \ \beta \in S^\omega : \ \sigma_i \beta \not\models P)))$$

Lynch ("Distributed Algorithms") offers these additional points which are implied, or convenient:

- Safety properties are non-empty (allows the empty string)
- They are prefix-closed
- They are limit-closed

Liveness has many definitions. One from Alpern/Schneider:
P is a liveness property if and only if

$$(\forall \alpha : \ \alpha \in S^* : \ (\exists \beta : \ \beta \in S^\omega : \ \alpha\beta \models P))$$

Lynch ("Distributed Algorithms") offers these additional points which are implied, or convenient:

- It is any property such that any arbitrary finite sequence α has an extension within P.

Lynch then goes on to prove the oft-stated theorem:

Theorem 8.8 (Lynch): If P is a safety property and a liveness property, then it is $S^* \cup S^\omega$.

Proof: Since P is a liveness property, any arbitrary partial execution σ has an extension in P. Since it is also a safety property, it is also prefix closed. Thus, for any σ, all its prefixes are in P. Since σ is arbitrary, all finite extensions of σ are also in P. Since P is limit-closed, we have our result.

Machine Closure (Lamport, DEC SRC TR 152): A pair $\langle S, L\rangle$ of properties (safety and liveness) is machine closed iff every finite sequence satisfying S can be extended to an infinite sequence satisfying $S \wedge L$. In other words, the liveness property does not "collide" with the safety property. Lamport points out that non machine-closed specifications of systems are natural (clear, expressive). He writes memory model specifications in this manner (e.g., his Alpha and Itanium memory model specifications).

2 Kozen, on Collapsing NFA, Supplementary Lecture B

Let M and N be two NFA:

$M = (Q_M, \Sigma, \Delta_M, S_M, F_M)$

$N = (Q_N, \Sigma, \Delta_N, S_N, F_N)$

Let $\approx$ be a subset of $Q_M \times Q_N$. For $B \subseteq Q_N$, define

$$C_{\approx}(B) = \{p \in Q_M \mid \exists q \in B\; p \approx q\},$$

the set of states of M that are related via $\approx$ to some state in B. Similarly, for $A \subseteq Q_M$, define

$$C_{\approx}(A) = \{q \in Q_N \mid \exists p \in A\; p \approx q\}.$$

The relation $\approx$ can be extended in a natural way to subsets of Q_M and Q_N: for $A \subseteq Q_M$ and $B \subseteq Q_N$,

$A \approx B$ iff $A \subseteq C_{\approx}(B)$ and $B \subseteq C_{\approx}(A)$,

iff

$\forall p \in A : \exists q \in B : p \approx q$, and $\forall q \in B : \exists p \in A : p \approx q$.

Definition B.1: The relation $\approx$ is called a bisimulation if all the following hold:

1. $S_M \approx S_N$
2. if $p \approx q$, then for all $a \in \Sigma$, $\Delta_M(p, a) \approx \Delta_N(q, a)$, and
3. if $p \approx q$, then $p \in F_M$ iff $q \in F_N$.

The bisimilarity class of M is the family of all NFA that are bisimilar to M.

- All NFA in the bisimilarity class accept the same set, and
- There is a unique minimal NFA in this set that can be constructed through a collapsing construction.

3 Equivalences and Preorders between Structures (Clarke, Ch11)

- Bisimulation relation between two structures (p 171), written $M \equiv M'$.
- Bisimulation equivalent structures (p 171, exs Fig 11.1 and 11.2)
- Two paths correspond (def) if states are pairwise bisimilar

Lemma 31, CGP: if $B(s, s')$, then for every path beginning at s, there is a corresponding path beginning at s', and vice versa.

Theorem 13, CGP: if $B(s, s')$, then for every CTL* formula f,

$$s \models f \Leftrightarrow s' \models f.$$

Theorem 14, CGP: if $M \equiv M'$, then for every CTL* formula f,

$$M \models f \Leftrightarrow M' \models f.$$

The converse of Theorem 14 is also true.

Interesting result: if two structures are *distinguished* by a CTL* formula, they can also be distinguished by a CTL formula.

Simulation Relation: Used as the basic means of comparing systems whose behaviors cannot be easily captured using TL properties alone.

Simulation equivalence is tantamount to ACTL* equivalence (ACTL* lacks the E quantifier... of course write formulae in negation-normal form with negation pushed innermost.) Two simulation equivalent structures need not be bisimilar (see Figure 11.4 of CGP).

4 Illustration of Simulation: Proofs of Correctness of Pipelined Processors

The term "datapath circuits" denotes all those circuits that perform computations similar to a processor. Members of this family include general purpose processors as well as special purpose processors such as priority queues, LRU units, microprogram engines, etc.

There are many different ways to model datapath circuits. One conceptually clear way (developed by many, including myself during my PhD dissertation) is to view them as *abstract data types*. For example, an LRU unit viewed as an abstract data type (ADT) has the following *constructors* that create its states: `init` to initialize the LRU, and `use(S,a)` to record the usage of address `a` with respect to state `S`. An observer for the LRU is `least(S)` that observes a given state `S` and returns the LRU value with respect to this state.

Once upon a time, researchers in this area used to advocate a purely *algebraic* style to specifying datatypes. However, their examples seldom went beyond stacks and queues. The author considers the LRU as an eminent example of a system whose purely algebraic style specification is pretty interesting (and nearly useless in terms of clarity). One interesting fact about LRU is that it resembles a queue except it needs to "know" the oldest item inserted into it and check if that item has been inserted since! (Try writing a purely algebraic specification.)

Suppose we take a more practical approach where we *specify* the LRU unit through one simple (and concrete) implementation. Suppose we also specify a more involved LRU as another concrete implementation (say, realized using a matrix of bits). How do we ensure that the implementation of the LRU is faithful to its specification? One verification criterion commonly used is based on the notion of algebraic homomorphism, expressed with the help of a commuting diagram:

Captured as a mathematical equation, Figure 1 specifies the following verification condition:

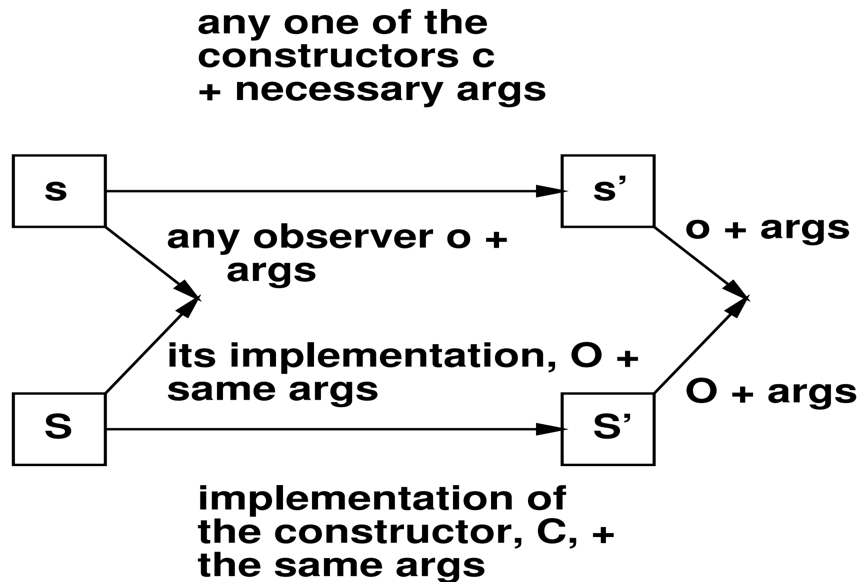


Figure 1: ADT verification criterion. States s and S "match" or "correspond" if all observers yield the same value when applied to s and S. Here, s and S match as do s' and S'. The proof technique illustrated in this diagram is based on induction on contructor application.

```
o(s,args) = O(S,args)    =>    o(c(s,args'),args) = O(C(S,args'),args)
```

As a concrete example, suppose we specify the LRU unit using a queue data structure and implement it using a matrix data structure, as shown in Figure 2:

To keep the examples simple, we will return to a familiar (albeit hackneyed) one: a FIFO queue. We will discuss a non pipelined queue in Section 4.0.1 and a pipelined queue in Section 4.1.

4.0.1 A Simple Non-pipelined FIFO Queue

Let's take a simple circulating pointer queue, specify it in the usual way (using two pointers "chasing each other") and implement it in a VLSI efficient way (using MLS counters for the pointers, as explained later). We will use a textual language for specifying behaviors, whose constructs will be introduced as needed. The basic descriptive style will be that of state transition systems, except we shall specify things in a *symbolic* fashion. The tacit convention is that we will be working with a universally quantified fragment of first-order logic.

Specification

We will write the FIFO queue specification using the tuple data type `<M, F, R, f, e>` where `M` is the memory, `F` is the front-pointer of the queue, `R` is the rear-pointer, `f` is the "full flag", and "e" is the "empty flag". Call this FIFO specification F1, standing for "FIFO version 1". We will subject `M` to the operation `M[v/a]` which means "update `M` such that its address `a` contains value `v`". The update specification (`[v/a]`) can be cascaded; thus, `M[v/a][v1/a1]` means "update `M` such that its address `a` has `v`, and update the resulting memory such that `a1` has `v1`". Obviously, if `a = a1`,

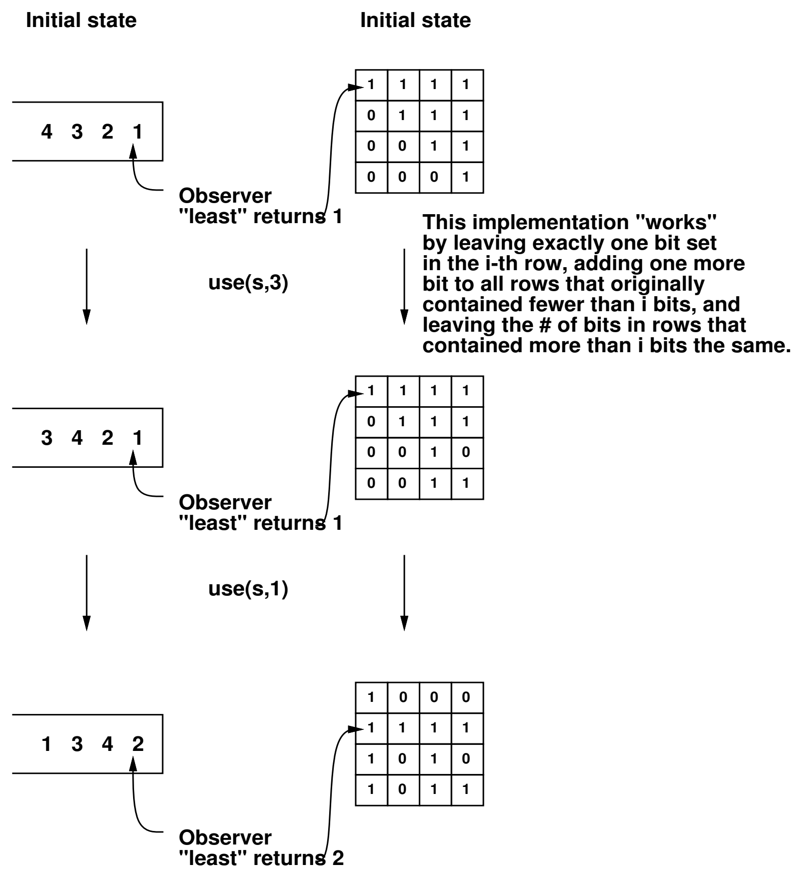


Figure 2: An LRU specification and its implementation

then the value `v` "gets over-written" by the value `v1`. We will subject `F` and `R` to the `+1` operation, which is tacitly assumed to occur under the modulus of an unspecified (but fixed) `capacity`, with the condition that `capacity` is $2^N - 1$ for some N.

We will describe three constructor operations on the queue: `init(s)` which initializes the queue, `ins(s,v)` which inserts `v` at the rear of the queue in state `s`, and `rem(s)` which removes the element at the head of the queue. We will describe three observers on the queue: `front(s)` which returns the front value of the queue, `full(s)` which returns true if `s` is in a full state, and `empty(s)` which returns true if `s` is in an empty state.

```
Specification of the operations of FIFO F1:

init(<M, F, R, f, e>) = <M, 0, 0, false, true>

empty(<M, F, R, f, e>) = e

full(<M, F, R, f, e>) = f

ins(<M, F, R, false, e>, v) = <M[v/R], F, R+1, (F == R+1), false>

rem(<M, F, R, f, false>) = <M, F+1, R, false, (F+1 == R)>

front(<M, F, R, f, e>) = M[F]
```

Implementation

Let us implement this FIFO queue by implementing `F` and `R` using highly efficient hardware counters called *maximum length sequence* (MLS) counters. An N-bit MLS counter is a shift-register with bits numbered, say, 0 to N-1 from left to right, and with the shift register input set to the XOR of bits N-2 and N-1. It counts thus:

```
starting at 0000 (for N=4)
remain stuck in 0000 forever

starting in 0001, count as follows

0001, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111, 0111, 0011
and back to 0001
```

Thus an N-bit MLS counter cannot be initialized in 0000. Initialized in any other states, it counts "higgeldy piggeldy" over $2^N - 1$ states.

The implementation of the FIFO is achieved by replacing F and R with MLS counters of the appropriate lengths, initializing them in a non-zero state, and for each `F+1` or `R+1`, replace them by the "shift" command on the MLS counter. Call this implementation of FIFO "F2" standing for FIFO, version 2.

Correctness of Implementation

We can now apply the previous verification criterion based on constructor induction. It will be clear that we need to *strengthen* the former verification condition to the following (taking `s` to

represent the queue specification states that employ "ordinary" counters and S to represent the queue implementation states that employ MLS counters:

```
[o(s,args) = O(S,args)] /\ good(s) /\ GOOD(S)
=>
[o(c(s,args'),args) = O(C(S,args'),args)] /\ good(c(s,args')) /\ GOOD(C(S,args'))
```

where `GOOD(S,...)` is true if `F<>0` and `R<>0` and the full and empty bits `f, e` are consistent with the counters `F` and `R`, while `good(s,...)` is true if `f, e` are consistent with the counters `F` and `R`.

Notice that we still have the same "commuting diagram" as before, except that the transitions are defined only on a subset of the implementation states.

4.1 A Simple Pipelined FIFO Queue

Let us now define a version of F1 called F3 that is implemented in a pipelined fashion. More specifically, F3 will carry out the queue operations following a time schedule defined by a clock oscillator. Below, we will use the notation `c(S1, S2, ...)` to denote that a hardware module (F3) is in control state `c` and data state `<S1, S2, ...>`. We will use the notation

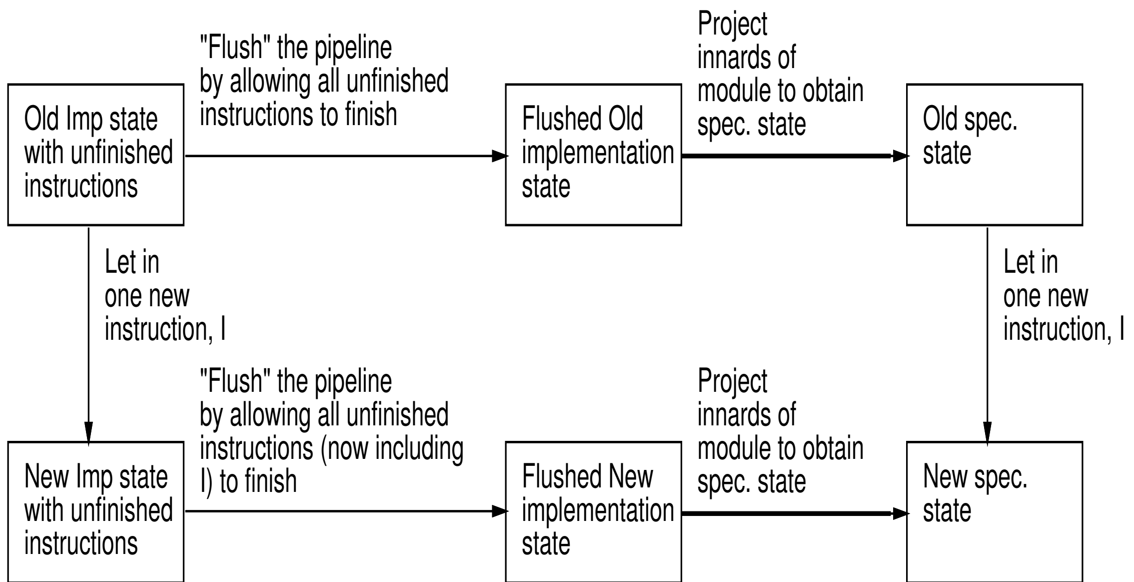
```
c(S1, S2, ...) -- op(args) --> c'(S1', S2', ...)
```

to indicate that the hardware module in question performs operation `op` with arguments `args` in one clock cycle, reaching the "next state" `c'(S1',S2', ...)`. A collection of such state transition rules will define the FIFO queue F3. The control state names used are `q` for *quiescent* and `p` for *pipelined*.

```
q(M, F, R, f,     e)     -- nop     --> q(M,      F,   R,   f,          e)

q(M, F, R, f,     e) where e = false
                         -- rem     --> q(M,      F+1, R,   false,      (F+1 == R))

q(M, F, R, f,     e) where f = false
                         -- ins(v)  --> p(M[v/R], F,   R,   (F == R+1), false)

p(M, F, R, f,     e)  -- nop     --> q(M,      F,   R+1, f,          e)

p(M, F, R, f,     e) where e = false
                         -- rem     --> q(M,      F+1, R+1, false,      (F+1 == R+1))

p(M, F, R, f,     e) where f = false
                         -- ins(v1) --> p(M[v1/R+1], F, R+1, (F == R+2), false)
```

Does F3 implement F1 faithfully? For this, we can make an attempt to apply the verification criterion shown in Figure 1. However this attempt fails because for a pipelined processor, there exists no point of clear correspondence. However we can apply the technique proposed by Burch and Dill illustrated in Figure 3. This figure also illustrates a slight modification of the original verification criterion for handling cases where the the specification state cannot be obtained by merely projecting the innards from the implementation state.

Notice from Figure 3 that this verification criterion also is based on induction on the length of instruction traces. In effect, it proves (basis case) that a single instruction followed by an infinity



The above verification criterion may also be generalized for handling Spec states that can't be derived from Imp states by mere projection (e.g. LRU). In this case, the Flushed New Imp state and New Spec state must correspond assuming that the Flushed Old Imp state and Old spec state do so.

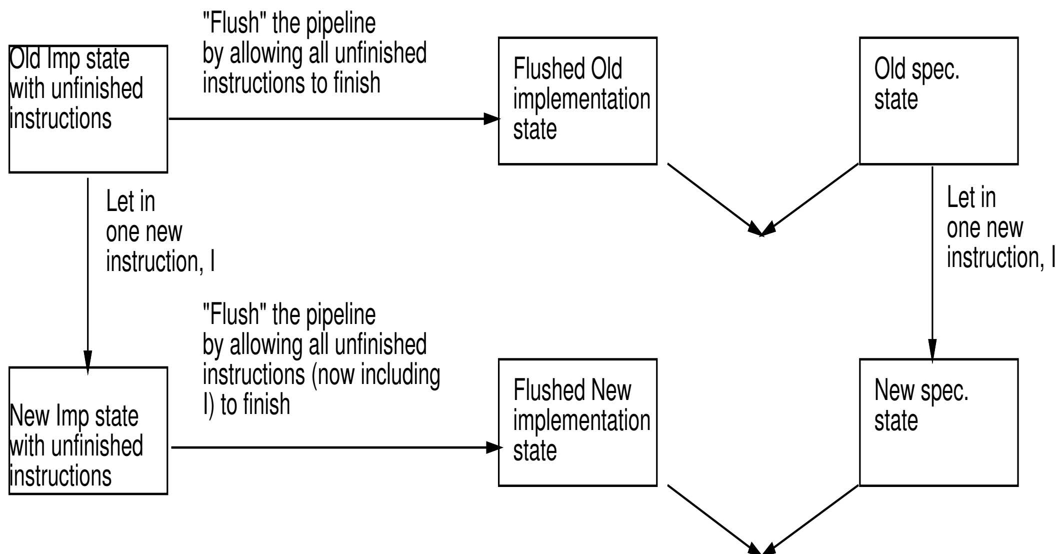


Figure 3: A verification criterion for pipelined processors

of NOPs is handled correctly by both systems, and assuming that an instruction stream of length N followed by an infinity of NOPs is handled correctly, then so is an instruction stream of length N+1 followed by an infinity of NOPs correctly handled.

Let us apply this verification criterion and generate a few of the VCs. Suppose the specification machine for this verification is a non-pipelined FIFO with identical internal state representation. More specifically, the specification machine doesn't have the pipelined state `p` and its `ins` operation is as follows:

```
q(M, F, R, f,     e) where f = false
                    -- ins(v)  --> q(M[v/R], F, R+1, (F == R+1), false)
```

As one example of generating a VC, consider the old implementation state

```
p(M, F, R, f, e).
```

"Flushing" this implementation state by executing a `nop` results in the old specification state

```
q(M, F, R+1, f, e).
```

Now, we assume that `f` is false, and execute one instruction (e.g. an `ins(v1)`) from the old implementation state, resulting in

```
p(M[v1/R+1], F, R+1, (F == R+2), false).
```

Flushing this state results in

```
p(M[v1/R+1], F, R+2, (F == R+2), false).
```

If one were to execute `ins(v1)` from the old specification state

```
q(M, F, R+1, f, e),
```

one would get

```
q(M[v1/R+1], F, R+2, (F == R+2), e)
```

which agrees with the flushed new implementation state.

In general, doing the above style of reasoning requires a theorem prover. Fortunately, Burch, Dill and most others have been able to use automatic decision procedures for fragments of first-order logic to accomplish much of their reasoning automatically.

4.2 Verifying pipelined processors

In this section, we will consider a toy pipelined processor. For the sake of simplicity, we consider exactly one instruction class, namely the `alu` class. The instruction set level specification of the processor is

```
State: RF -- register file

State Transition (called a_step below):

      RF
        -- (op s1 s2 d st) -->

      if st
      then RF
      else wr(RF, d, alu(op, rd(RF, s1), rd(RF, s2)))
```

The implementation uses a 4-stage pipeline (as opposed to previous versions of this example which say that there are 3 stages). A diagram showing this pipelined implementation is in Figure 4.

The pipelined machine state and the stage behaviors are as follows:

```
State: (RF, all the other registers which are mentioned below)

State Transitions per stage:

        Instruction fetch stage:

                opcode  := op
                dstn    := d
                s1reg   := s1
                s2reg   := s2
                stall   := st

        Operand fetch stage:

                dstnd   := dstn
                stalld  := stall
                opcoded := opcode
                opreg1  := if (s1reg = dstnd) and not(stalld)
                                then aluout
                           else if (s1reg = dstndd) and not(stalldd)
                                then wb
                           else rd(RF, s1reg)
                           endif
                opreg2  := if (s2reg = dstnd) and not(stalld)
                                then aluout
                           else if (s2reg = dstndd) and not(stalldd)
                                then wb
                           else rd(RF, s2reg)
                           endif

        Alu op stage:

                dstndd  := dstnd
                stalldd := stalld
                wb      := alu(opcoded, opreg1, opreg2)


        Writeback stage:

                RF      := if stalldd then RF
                           else wr(RF, dstndd, wb)
```

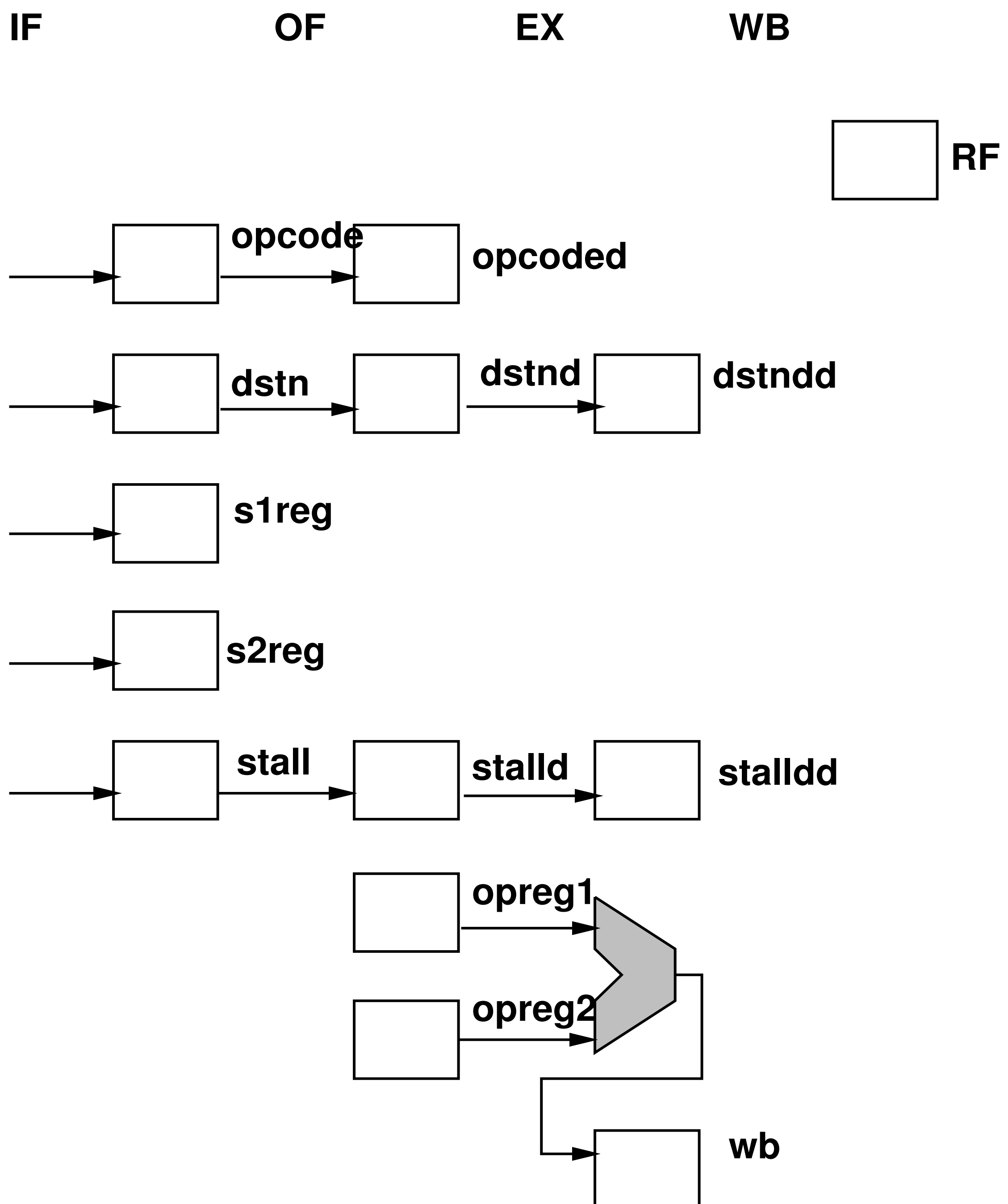


Figure 4: A simple 4-stage pipeline

```
endif
```

The verification condition using the "standard" (e.g. Burch/Dill) method would look like in Figure 5:

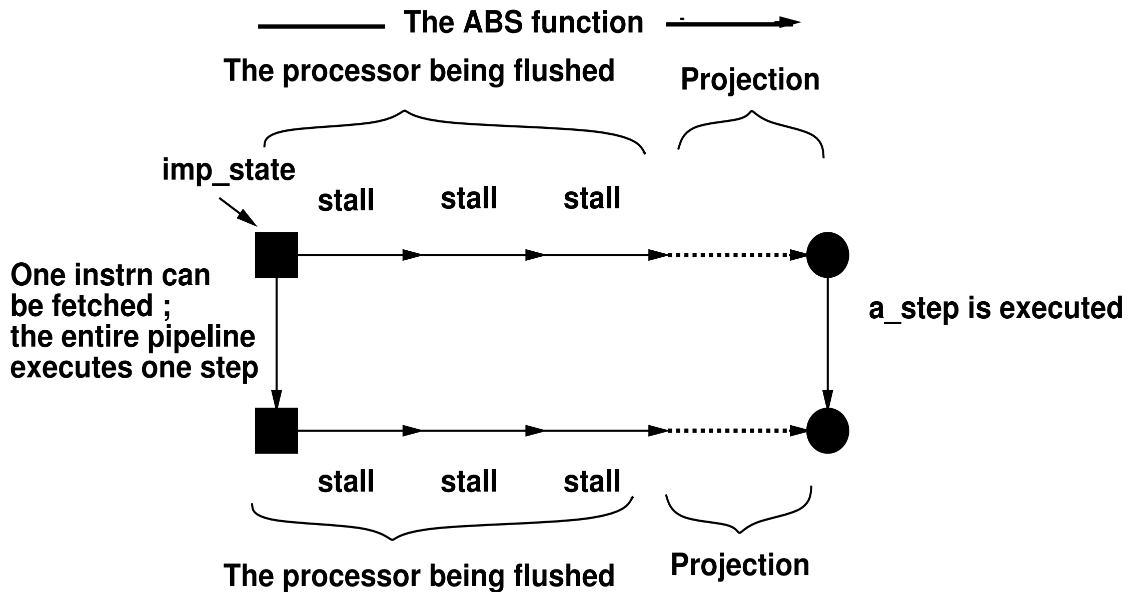


Figure 5: The usual verification criterion for pipelined processors

The verification conditions generated are:

```
imp_state is an arbitrary implementation state meeting the system
invariant for ''legal'' states  (in our example, the invariant is ''true''):

VC : a_step(stall(stall(stall(imp_state)))) =

stall(stall(stall(istep(imp_state))))
```

This usually generates in gigantic "if-then-elses" besides inviting other problems when iterative loops, interlocks, etc. are present in pipeline stages.

The verification condition using the Hosabettu/Srivas method would look like in Figure 6:

The verification conditions generated are:

```
VC1: RF(c_wb(imp_state)) = RF(istep(imp_state))

VC2: RF(c_ex(c_wb(imp_state))) = RF(c_wb(istep(imp_state)))

VC3: RF(c_of(c_ex(c_wb(imp_state)))) = RF(c_ex(c_wb(istep(imp_state))))

VC4: RF(A_step(c_of(c_ex(c_wb(imp_state))))) = RF(c_of(c_ex(c_wb(istep(imp_state)))))
```

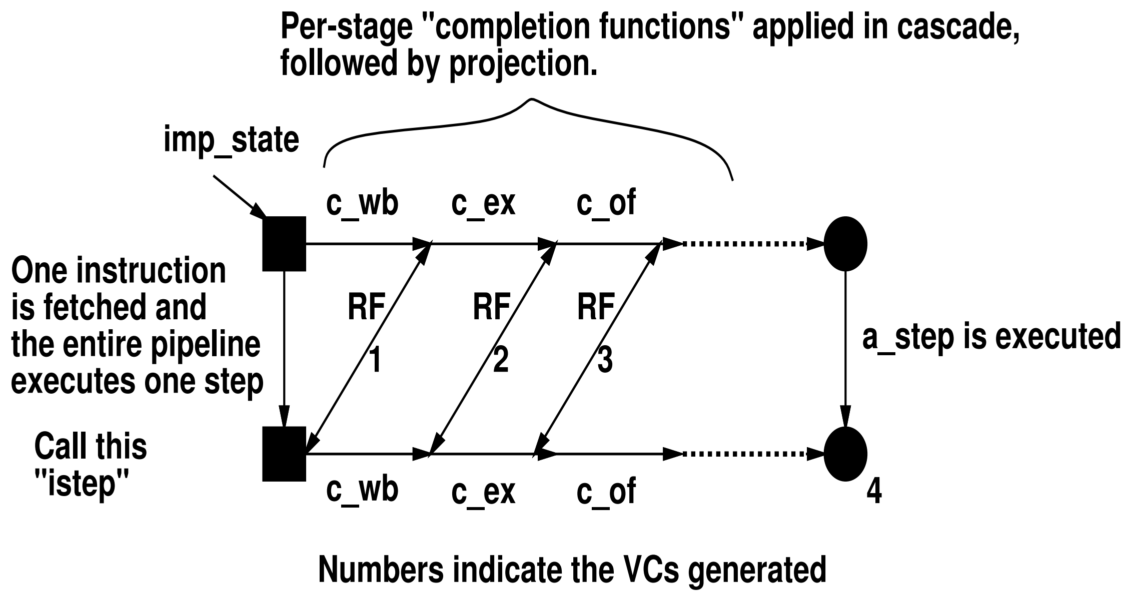


Figure 6: A better verification criterion for pipelined processors

The proof has been found to decompose in a modular fashion, besides circumventing problems due to interlocks, loops, etc. Essentially, for embedded stage loops, the user is forced to state an I/O behavior while writing the "completion" function for that stage.