An example on how to use Reactive Modules and Assume-Guarantee Reasoning

Demetrios Demopoulos
Rice University
demopoulos@rice.edu

October 31, 2001

Overview

- ISA & PIPELINE modules
- PIPELINE ≼ ISA
- Using abstraction modules

Assume-Guarantee Reasoning

ISA - A simple instruction set architecture

- 5 external variables (op, inp, src1, src2 & dest)
- 2 interface variables (out & stall)
- isaRegFile is the only private variable
- 1 round - 4 subrounds
- ISARegFile and ISAOut must be executed after ISASTall

PIPELINE

- A three stages pipeline: fetch operands - perform operations - write results
- PIPELINE is a parallel composition of 7 modules.
  - First stage: Pipe1, Opr1, Opr2
  - Second stage: Pipe2
  - Third stage: RegFile
  - Also, PipeOut & Stall
Assume-Guarantee Reasoning

GOAL: Show that PIPELINE is a correct implementation of the instruction set architecture ISA, i.e. PIPELINE \preceq ISA

We will make use of the following two theorems:

Projection refinement theorem: Let P and Q be two modules such that Q is refinable by P. Let W be a compatible with P module such that the interface variables of W include the private variables of Q, and are disjoint from the external variables of P. Then Q^w is projection refinable by P||W, and if P||W \preceq Q^w then P \preceq Q.

Such a module W is called a witness module to the refinement P \preceq Q.

Assume-Guarantee Reasoning - continued

Assume-guarantee rule for modules: Let P = P_1||\cdots||P_n and Q = Q_1||\cdots||Q_m be reactive modules such that Q is projection refinable by P. For all 1 \leq i \leq m, let \Gamma_i be the composition of arbitrary compatible components from P and Q with the exception of Q_i. Suppose \Gamma_i \preceq Q_i for all 1 \leq i \leq m and the relation \triangleright_P \cup \triangleright_Q is acyclic. Then P \preceq Q.

Such a module \Gamma_i is called an obligation model. Our goal is to find such modules \Gamma_i(\preceq Q_i), that have smaller state space than P.

\Gamma_i is parallel composition of two kinds of modules: essential and constraining modules.
The essential modules are chosen from P such that every interface variable of Q_i is an interface variable of some essential module.
In case that are some external variables of Q_i that are not observable for the essential modules, we need to choose constraining modules either from P or from Q to ensure refinability.
Typically, the external variables of \Gamma_i need to be constrained further in order for the refinement check to go through.

Verification of three-stage pipeline (part 1)

GOAL: PIPELINE \preceq ISA

Step 1: ISA is refinable by PIPELINE.

BUT, ISA is not projection refinable because isaRegFile is private.

Step 2: Use ISARegFile as a witness module. Prove that ISARegFile||PIPELINE \preceq ISA^w. Then by the first theorem we get that PIPELINE \preceq ISA.

Step 3: PROOF: Make use of the second theorem; prove that:

ISARegFile||RegFile||Opr1||Opr2||Pipe1||Pipe2||PipeOut||Stall \preceq ISARegFile||ISAOut||ISAStall

Verification of three-stage pipeline (part 2)

Step 3a: It is easy to see that:

ISARegFile<<ISARegFile
Stall<<ISAStall

Step 3b: We would like to find a \Gamma_2 such that \Gamma_2 \preceq ISAOut.
The only interface of ISAOut is out, which is generated by PipeOut. Therefore, PipeOut is the only essential module.

PipeOut<<ISAOut
ISAOut is not refinable by PipeOut. Let's add ISARegFile to get refinability.

ISARegFile||PipeOut<<ISAOut
The input regFile to PipeOut is not constrained. Let's add RegFile then.

ISARegFile||RegFile||PipeOut<<ISAOut
Now the inputs to RegFile are not constrained. We need to add Pipe2.
X ISARRegFile||RegFile||Pipe2||PipeOut \lessdot\ ISAOut

Now the inputs of Pipe2 are not constrained and the refinement check fails. We add Pipe1, Opr1, Opr2 & Stall to constrain them.

\[ \sqrt{\text{ISARRegFile}}||\text{RegFile}||\text{Pipe1}||\text{Pipe2}||\text{Opr1}||\text{Opr2}||\text{Stall}||\text{PipeOut} \lessdot\ ISAOut \] (3)

Finally, the refinement check can go through.

Using the second theorem on 1, 2 & 3 we get that:

\[ \text{ISARRegFile}||\text{RegFile}||\text{Opr1}||\text{Opr2}||\text{Pipe1}||\text{Pipe2}||\text{PipeOut}||\text{Stall} \lessdot\ \text{ISARRegFile}||\text{ISAOut}||\text{ISASTall} \]

\[ \text{PIPELINE}||\text{ISARRegFile}||\text{ISARRegFile\_d} \lessdot\ ISA^n ||\text{ISARRegFile\_d}||\text{AbsOpr1}||\text{AbsOpr2}||\text{AbsRegFile} \]

By the last theorem, we can remove ISARRegFile\_d||AbsOpr1||AbsOpr2||AbsRegFile from the right hand side.

Now we have, PIPELINE||ISARRegFile||ISARRegFile\_d \lessdot\ ISA^n. The module ISARRegFile\_d is a witness. Thus, by using the first theorem, we conclude that PIPELINE \lessdot\ ISA.
To verify an implementation vs. a specification using assumptions guarantee reasoning one needs to:

- Check for refiinability
- Compose implementation with witness modules to get projection refiinability
- Decompose both into components
- Find suitable obligation models for each of the specification components
- Check refinement for each one of the specification components.
- Put all refinement check together to get a proof.
- In case this compositional approach does not give us much advantage, augment the specification with abstraction modules and follow the same procedure.

\textit{Await dependency}: a dependency between the values of variables within a single round.

The module \textit{Q is refinable} by module \textit{P} if every interface variable of \textit{Q} is an interface variable of \textit{P}, and every external variable of \textit{Q} is an observable variable of \textit{P}.

Two modules \textit{P} and \textit{Q} are \textit{compatible} if the controlled variables of \textit{P} and \textit{Q} are disjoint and the await dependencies between variables of \textit{P} and \textit{Q} are acyclic.
module ISA

external op, inp, src1, src2, dest

interface out, stall

private isaRegFile

atom ISAStall controls stall

init update

[]true → stall' := nondet

atom ISARegFile controls isaRegFile

init

[]true → forall i do isaRegFile'[i] := 0

update

[]¬stall' ∧ op' = LOAD → isaRegFile'[dest'] := inp'

[]¬stall' ∧ op' = AND → isaRegFile'[dest'] := isaRegFile[src1'] ∧ isaRegFile[src2']

[]¬stall' ∧ op' = OR → isaRegFile'[dest'] := isaRegFile[src1'] ∨ isaRegFile[src2']

atom ISAOut controls out

init update

[]¬stall' ∧ op' = STORE → out' := isaRegFile[dest']

Figure 3.3: Instruction set architecture
module Opr1
interface opr1
external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src1
atom Opr1 controls opr1
update

[] ¬stall' → opr1' :=
if src1' = pipe1.dest ∧ pipe1.op ≠ NOP ∧ pipe1.op ≠ STORE
then if pipe1.op = LOAD then pipe1.inp else aluOut'
else if src1' = pipe2.dest ∧ pipe2.op ≠ NOP ∧ pipe2.op ≠ STORE
then wbReg else regFile[src1']

module Opr2
interface opr2
external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src2
atom Opr2 controls opr2
update

[] ¬stall' → opr2 :=
if (src2' = pipe1.dest) ∧ ¬(pipe1.op = NOP) ∧ ¬(pipe1.op = STORE)) then
if (pipe1.op = LOAD) then pipe1.inp else aluOut' fi
else if (src2' = pipe2.dest) ∧ ¬(pipe2.op = NOP) ∧ ¬(pipe2.op = STORE)) then wbReg
else regFile[src2'] fi fi

module Pipe1
interface pipe1.op, pipe1.inp, pipe1.dest
external stall, inp, op, dest
atom Pipe1 controls pipe1.op, pipe1.dest, pipe1.inp
init
[] true → pipe1.op' := NOP
update
[true → pipe1.op' := if stall' then NOP else op';
pipe1.dest' := dest'; pipe1.inp' := inp']

Figure 3.4: Pipeline stage 1
module Pipe2
  interface pipe2.op, pipe2.dest, wbReg, aluOut
  external pipe1.op, pipe1.inp, pipe1.dest, opr1, opr2
  atom ALU controls aluOut
    update
      []pipe1.op = AND -> aluOut' := opr1 \land opr2
      []pipe1.op = OR -> aluOut' := opr1 \lor opr2
  atom Pipe2 controls pipe2.op, pipe2.dest
    init
      []true -> pipe2.op' := NOP
    update
      []true -> pipe2.op' := pipe1.op; pipe2.dest' := pipe1.dest
  atom WbReg controls wbReg
    update
      []pipe1.op = AND \lor pipe1.op = OR -> wbReg' := aluOut'
      []pipe1.op = LOAD -> wbReg' := pipe1.inp

module RegFile
  interface regFile
  external pipe2.op, pipe2.dest, wbReg, aluOut
  atom RegFile controls regFile
    init
      []true -> forall i do regFile'[i] := 0
    update
      [] pipe2.op = AND \lor pipe2.op = OR \lor pipe2.op = LOAD ->
        forall i do regFile'[i] := if pipe2.dest = i then wbReg else regFile[i]

Figure 3.5: Pipeline stages 2 and 3
Theorem 3.6, we obtain the following assume-guarantee proof:

\[
\begin{align*}
\text{PipeOut} & \parallel \text{Pipe1} \parallel \text{Pipe2} \parallel \text{Stall} \leq \text{ISAOu} \\
\text{AbsRegFile} & \parallel \text{ISARegFile}_d \parallel \text{ISARegFile} \\
\text{Opr1} & \parallel \text{AbsOpr2} \parallel \text{Pipe1} \parallel \text{Pipe2} \leq \text{AbsOpr1} \\
\text{AbsRegFile} & \parallel \text{ISARegFile}_d \parallel \text{ISARegFile} \\
\text{Opr2} & \parallel \text{AbsOpr1} \parallel \text{Pipe1} \parallel \text{Pipe2} \leq \text{AbsOpr2} \\
\text{AbsRegFile} & \parallel \text{ISARegFile}_d \parallel \text{ISARegFile} \\
\text{AbsOpr1} & \parallel \text{AbsOpr2} \parallel \text{Pipe1} \parallel \text{Pipe2} \leq \text{AbsRegFile}_d \\
\text{RegFile} & \parallel \text{ISARegFile}_d \parallel \text{ISARegFile} \parallel \text{Stall} \\
\text{Stall} & \leq \text{ISASTall} \\
\text{ISARegFile} & \leq \text{ISARegFile}_d \\
\text{ISARegFile} & \parallel \text{ISARegFile}_d \parallel \text{RegFile} \\
\text{Pipe1} & \parallel \text{Pipe2} \parallel \text{Opr1} \parallel \text{Opr2} \parallel \text{PipeOut} \parallel \text{Stall} \\
\text{ISARegFile}_d & \leq \text{AbsOpr1} \parallel \text{AbsOpr2} \parallel \text{AbsRegFile} \\
\end{align*}
\]
module PipeOut
  interface out
  external op, regFile, dest
  atom Out controls out
    update
      []\neg stall' \land op' = STORE \rightarrow out' := regFile[dest']

module Stall
  interface stall
  external op, dest, pipe1.op, pipe1.dest, pipe2.op, pipe2.dest
  atom Stall controls stall
    update
      [] op' = STORE \land pipe1.op \neq NOP \land pipe1.op \neq STORE \land dest' = pipe1.dest \rightarrow
        stall' := true
      [] op' = STORE \land pipe2.op \neq NOP \land pipe2.op \neq STORE \land dest' = pipe2.dest \rightarrow
        stall' := true
      []default \rightarrow stall' := false

Figure 3.6: Pipeline output and stall
module AbsOpr1
    external isaRegFile, src1, stall
    interface opr1
    atom AbsOpr1 controls opr1
    update
        []¬stall' → opr1' := isaRegFile[src1']

module ISARegFile$_d$
    atom ISARegFile$_d$ controls isaRegFile$_d$
    init
        []true → forall i do isaRegFile'$_d$[i] := 0
    update
        []true → forall i do isaRegFile'$_d$[i] := isaRegFile$_d$[i]

module AbsRegFile
    atom AbsRegFile controls regFile
    init
        []true → forall i do regFile'$_d$[i] := 0
    update
        []true → forall i do regFile'$_d$[i] := isaRegFile$_d$[i]