Induction over instruction sequences

- Many systems we deal with processes “instructions” sequentially.

- We verify them by showing that for all instruction sequences the systems behave identically.

- While the conceptual proof is by induction, the actual proof “goes by a commutative diagram.”

- This module will expose you to three simple examples:
  - A least recently use (LRU) unit
  - A pipelined FIFO queue
  - A ‘pico-processor’
The basic theme: “Commute diagrams”

- The basic idea is to capture induction

- Suppose the spec and imp are
  
  - in a pair of states satisfying their respective invariants, and
  
  - these states are “observationally equivalent” (to be clarified)

- Suppose we make the spec and imp execute all possible one-step moves; then
  
  - do the systems end up in states satisfying their invariants? and
  
  - are these states observationally equivalent also?
Illustration of verification criterion

- Constructors are state-changing operations
- Observers are value-producing operations
- After all sequences of constructor applications resulting in legal states, all observers yield the same value.

```
any one of the constructors c
+ necessary args

s ↦ s'
any observer o + args
its implementation, O + same args
S ↦ S'
implementation of the constructor, C, + the same args
```
Example of an LRU unit

Spec invariant: The set of entries = \{1, 2, 3, 4\}
Imp invariant: The set of entries = \{0001, 0011, 0111, 1111\}

Implementation of "use(s,i)":
move "i" to the tail.
Implementation of "use(S,i)":
reset ith row; set ith column

Implementation of "least(s)":
front of the queue.
Implementation of "least(S)":
look for row containing all "1"s

Observer
"least" returns 1
use(s,3)

Observer
"least" returns 1
use(s,1)

Observer
"least" returns 2
Verifying pipelined systems

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.

Old Imp state with unfinished instructions

“Flush” the pipeline by allowing all unfinished instructions to finish

Let in one new instruction, I

New Imp state with unfinished instructions

“Flush” the pipeline by allowing all unfinished instructions (now including I) to finish

Let in one new instruction, I

Old Spec state

Flushed Old implementation state

Old Spec state

Project innards of module to obtain spec. state

New Spec state

Flushed New implementation state

New Spec state

Flushed New implementation state

Let in one new instruction, I
Example: A Pico-processor

IF          OF          EX          WB

opcoded  

dstnd  dstndd

doct

s1reg

s2reg

stalld  stalldd

opreg1  opreg2

wb

RF
ISA level specification of the Pico-processor

specification: THEORY
BEGIN
IMPORTING instructions, implementation_state
% -- The type of the Abstract State
% as: VAR regfile

% -- The input consists of the instruction and ‘stall’
%i: VAR inputs_type

% -- The only instruction of the processor
% A_step(as,i): regfile =
assign(as,dest(ins_in(i))),
    alu_reg(op(ins_in(i))),
    select(as,src1(ins_in(i))),
    select(as,src2(ins_in(i))))

% -- This is used in the commute diagram
%m: VAR bool
A_step_new(as,i,m): regfile =
    IF not(m) THEN as    %--stall due to bubble
    ELSE A_step(as,i)
    ENDIF
END specification
Implementation State of the Pico-processor

implementation_state: THEORY
BEGIN
IMPORTING instructions
state_I: TYPE = [#
    opcode, opcoded : opcode,
    dstn,  dstnd,  dstndd : reg_addr,
    s1reg,  s2reg : reg_addr,
    stall,  stalld,  stalldd : bool,
    opreg1, opreg2, wb : value,
    rf: regfile #]  % -- the only observable

% -- Implementation invariant = TRUE

inputs_type : TYPE+
    = [# ins_in: instruction, stall_in : bool #]
END implementation_state
Implementation transitions

implementation_trans: THEORY
BEGIN
IMPORTING implementation_state

% Initial state.
% Init(q:state_I): state_I = q % invariant = TRUE

% -- Defining the transition function.
% q: VAR state_I
i: VAR inputs_type

% --------------------- WB stage.
% next_rf(q) : regfile =
    IF stalldd(q) then rf(q)
    ELSE assign(rf(q), dstndd(q), wb(q))
    ENDIF

% --------------------- ALU-op stage.
% next_dstndd(q) : reg_addr = dstnd(q)
next_stalldd(q): bool = stalld(q)
next_wb(q): value
   = alu_reg(opcoded(q), opreg1(q), opreg2(q))

% ------------------------ Opnd fetch stage.
%
next_dstnd(q): reg_addr = dstn(q)
next_stalld(q): bool    = stall(q)
next_opcoded(q): opcode = opcode(q)
next_opreg1(q): value =
   IF (s1reg(q) = dstnd(q)) AND NOT(stalld(q))
      THEN alu_reg(opcoded(q), opreg1(q), opreg2(q))
         % read the ALU o/p wires
   ELSE IF (s1reg(q) = dstndd(q)) AND NOT(stalldd(q))
      THEN wb(q)
      ELSE select(rf(q), s1reg(q))
   ENDIF
ENDIF

next_opreg2(q): value =
   IF (s2reg(q) = dstnd(q)) AND NOT(stalld(q))
      THEN alu_reg(opcoded(q), opreg1(q), opreg2(q))
   ELSE IF (s2reg(q) = dstndd(q)) AND NOT(stalldd(q))
      THEN wb(q)
      ELSE select(rf(q), s2reg(q))
   ENDIF
END IF

% ------------------------ Instrn fetch stage.
%
next_opcode(q,i): opcode = op(ins_in(i))
next_dstn(q,i): reg_addr = dest(ins_in(i))
next_s1reg(q,i): reg_addr = src1(ins_in(i))
next_s2reg(q,i): reg_addr = src2(ins_in(i))
next_stall(q,i): bool = stall_in(i)
Implementation step, I\_step

\%
% -- Define the next state transition function.
%
I\_step(q,i) : state\_I = (#
    opcode := next\_opcode(q,i),
opcoded := next\_opcoded(q),
dstn := next\_dstn(q,i),
dstnd := next\_dstnd(q),
dstndd := next\_dstndd(q),
s1reg := next\_s1reg(q,i),
s2reg := next\_s2reg(q,i),
stall := next\_stall(q,i),
stall\_dd := next\_stall\_dd(q),
stall\_ddd := next\_stall\_ddd(q),
opreg1 := next\_opreg1(q),
opreg2 := next\_opreg2(q),
w\_b := next\_w\_b(q),
rf := next\_rf(q)
    #)

END implementation\_trans
The “standard” commute diagram for pipelined processors

The ABS function

The processor being flushed

Projection

imp_state

stall stall stall

One instrn can be fetched; the entire pipeline executes one step

The processor being flushed

Projection

a_step is executed

stall stall stall

Project
Commutativity - the standard one
(Established automatically by (grind) in a minute!)

commutativity: THEORY
BEGIN

IMPORTING implementation_state,
     implementation_trans,
     abstraction,
     specification

q: VAR state_I
i: VAR inputs_type

%-----------------
% Synchronization function. If the stall register is
% set, then return false, else return true
% sync_fun(q:state_I,i:inputs_type) : bool = not(stall_in(i))

%-----------------
% The abstraction map
% ABS(q:state_I) : regfile =
    projection(Complete_till_OF(q))
commutativity: THEOREM

\[ \text{ABS}(I_{\text{step}}(q,i)) = A_{\text{step\_new}}(\text{ABS}(q),i,\text{sync\_fun}(q,i)) \]

END commutativity
The completion functions approach to commutativity

Per-stage "completion functions" applied in cascade, followed by projection.

One instruction is fetched and the entire pipeline executes one step

Call this "istep"

Numbers indicate the VCs generated

a_step is executed
Layered Definition of the Abstraction Map

abstraction: THEORY
BEGIN

IMPORTING implementation_state

% Complete the instruction in the WB stage
%
Action_WB(q : state_I) : state_I =
  IF stalldd(q)
  THEN q
  ELSE
    q with [(rf) := assign(rf(q), dstndd(q), wb(q))]
  ENDIF

% Complete the instruction in the EX stage
%
Action_EX(q : state_I) : state_I =
  IF stallld(q)
  THEN q
  ELSE
    q WITH
    [(rf) :=
assign(rf(q),
stnd(q),
    alu_reg(opcoded(q), opreg1(q), opreg2(q)))
]
ENDIF

%-----
%
% Complete the instruction in the operand fetch (OF) stage
Action_OF(q : state_I) : state_I =
    IF stall(q)
        THEN q
    ELSE
        q WITH
        [ (rf) :=
            assign(rf(q),
                dstn(q),
                alu_reg(opcode(q),
                    select(rf(q), s1reg(q)),
                    select(rf(q), s2reg(q))))
        ]
    ENDIF
% Complete_till function definitions. ‘‘Till’’ counts
% FROM the tail TILL the mentioned stage.

Complete_till_WB(q : state_I): state_I =
    Action_WB(q)

Complete_till_EX(q : state_I): state_I =
    Action_EX(Complete_till_WB(q))

Complete_till_OF(q : state_I): state_I =
    Action_OF(Complete_till_EX(q))

projection(q:state_I) : regfile = rf(q)

ABS(q:state_I) : regfile =
    projection(Complete_till_OF(q))

% Synchronization function. If the stall register is
% set, then return false, else return true

sync_fun(q:state_I,i:inputs_type) : bool = not(stall_in(i))

END abstraction
VCs in the completion-functions approach
(permit stagewise verification and proof reuse)

commutativity: THEORY
BEGIN
IMPORTING implementation_state, implementation_trans,
      abstraction, specification
q: VAR state_I
i: VAR inputs_type
%----------
VC1_rf: CLAIM
rf(I_step(q,i)) = rf(Complete_till_WB(q))

%----------
VC2_rf: CLAIM
rf(Complete_till_WB(I_step(q,i))) = rf(Complete_till_EX(q))

%----------
VC3_rf: CLAIM
rf(Complete_till_EX(I_step(q,i))) = rf(Complete_till_OF(q))

%----------
VCn_rf: CLAIM
NOT (stall_in(i)) IMPLIES
rf(Complete_till_OF(I_step(q,i))) =
A_step(projection(Complete_till_OF(q)),i)
%%-------------------

commutativity: THEOREM

    ABS(I_step(q,i)) = A_step_new(ABS(q),i, sync_fun(q,i))

END commutativity
Summary of instruction sequence processor verification

• In general, one needs to state non-trivial invariants.

• Some invariants relate to feedback logic while others relate to register integrity.

• Completion functions can be a “golden C” simulation model - thus providing an approach to bring FV ideas into simulation-based validation.
Concluding Remarks for the course

- If we all successfully made it this far, we probably got a good glimpse of
  - how “logic” gets used in a variety of ways (propositional, first-order, program logics, various ways to induct, ...)
  - that induction plays a central role in reasoning
  - that invariants play a central role in reasoning

- The academic examples we saw do have more serious counterparts in real-life SW/HW design

- Hope you enjoy your follow-on tools course to see some of that!

- **Questions/comments appreciated!**
  (even after I get back to Utah).