Can we derive the 5-stage pipeline by successive refinement of a 2-stage pipeline?
A 2-Stage Processor in RTL

1) Design Microarchitecture
2) Locate Datapaths/Memories and create modules
3) Identify Input/Output ports
4) Design the Controller (FSM)

Designing a 2-Stage Processor with GAA

1) Design Microarchitecture
2) Locate Datapaths/Memories and create modules
3) Define Interface methods: read, action, action value
CPU as one module

Method calls embody both data and control (i.e., protocol)

Read method call
Action method call

CPU as one module

module mkCPU#(Mem iMem, Mem dMem)();
  // Instantiating state elements
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Value) rf <- mkBypassRF();
  SFIFO#(InstTemplate, RName) bu <- mkLoopyFifo(findf);

  // Some definitions
  Instr instr = iMem.read(pc);
  Iaddress predIa = pc + 1;

  // Rules
  rule fetch_decode ...
  rule execute ...
endmodule
A Modular organization

Suppose we include rf and pc in Fetch and bu in Execute
Fetch delivers decoded instructions to Execute and needs to consult Execute for the stall condition
Execute writes back data in rf and supplies the pc value in case of a branch misprediction

Recursive modular organization

module mkCPU2#(Mem iMem, Mem dMem)();
  Fetch fetch <- mkFetch(iMem, execute);
  Execute execute <- mkExecute(dMem, fetch);
endmodule

interface Fetch;
  method Action setPC (Iaddress cpc);
  method Action writeback (RName dst, Value v);
endinterface

interface Execute;
  method Action enqIt(InstTemplate it);
  method Bool stall(Insn instr)
endinterface

Unfortunately, recursive module syntax is not as simple
Issue

- A recursive call structure can be wrong in the sense of “circular calls”; fortunately the compiler can perform this check.
- Unfortunately recursive call structure amongst modules is supported by the compiler in a limited way.
  - The syntax is complicated
  - Recursive modules cannot be synthesized separately

Syntax for Recursive Modules

```
module mkFix#(Tuple2#(Fetch, Execute) fe)
    (Tuple2#(Fetch, Execute));
match{.f, .e} = fe;
  Fetch  fetch <- mkFetch(e);
  Execute execute <- mkExecute(f);
return(tuple2(fetch,execute));
endmodule

(* synthesize *)
module mkCPU(Empty);
match {.fetch, .execute} <- moduleFix(mkFix);
endmodule
```

`moduleFix` is like the Y combinator

\[ F = Y F \]
Passing parameters

module mkCPU#(IMem iMem, DMem dMem)(Empty);
module mkFix#(Tuple2#(Fetch, Execute) fe)

  (Tuple2#(Fetch, Execute));
  match{.f, .e} = fe;
  Fetch   fetch <- mkFetch(iMem,e);
  Execute execute <- mkExecute(dMem,f);
  return(tuple2(fetch,execute);
endmodule

  match {.fetch, .execute} <- moduleFix(mkFix);
endmodule

Fetch Module

module mkFetch#(IMem iMem, Execute execute) (Fetch);
    Instr instr = iMem.read(pc);
    Iaddress predIa = pc + 1;

    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkBypassRegFile();

    rule fetch_and_decode (!execute.stall(instr));
        execute.enqIt(newIt(instr,rf));
        pc <= predIa;
    endrule

    method Action writeback(RName rd, Value v);
        rf.upd(rd,v);
    endmethod

    method Action setPC(Iaddress newPC);
        pc <= newPC;
    endmethod
endmodule
Execute Module

```verilog
module mkExecute#(DMem dMem, Fetch fetch) (Execute);

  // Instantiate the SLoopyFifo and obtain the first element
  SFIFO#(InstTemplate) bu <- mkSLoopyFifo(findf);
  InstTemplate it = bu.first;

  // Define the execute rule
  rule execute ...
  
    // Define the enq method
    method Action enqIt(InstTemplate it);
    bu.enq(it);
    endmethod

    // Define the stall method
    method Bool stall(Instr instr);
    return (stallFunc(instr, bu));
    endmethod

  endmodule
```

Execute Module Rule

```verilog
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
      fetch.writeback(rd, va+vb); bu.deq();
    end
    tagged EBz {cond:.cv,addr:.av}:
      if (cv == 0) then begin
        fetch.setPC(av); bu.clear(); end
      else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
      fetch.writeback(rd, dMem.read(av)); bu.deq();
    end
    tagged EStore{value:.vv,addr:.av}: begin
      dMem.write(av, vv); bu.deq();
    end
  endcase
endrule
```
Subtle Architecture Issues

interface Fetch;
    method Action setPC (Iaddress cpc);
    method Action writeback (RName dst, Value v);
endinterface

interface Execute;
    method Action enqIt(InstTemplate it);
    method Bool stall(Instr instr)
endinterface

After setPC is called the next instruction enqueued via enqIt must correspond to iMem(cpc)

stall and writeback methods are closely related;
- writeback affects the results of subsequent stalls
- the effect of writeback must be reflected immediately in the decoded instructions

Any modular refinement must preserve these extra linguistic semantic properties

Fetch Module Refinement
Separating Fetch and Decode

module mkFetch#(IMem iMem, Execute execute) (Fetch);
    FIFO#(Instr) fetchDecodeQ <- mkLoopyFIFO();
    Instr instr = iMem.read(pc);
    Iaddress predIa = pc + 1;

rule fetch(True);
    pc <= predIa;
    fetchDecodeQ.enq(instr);
endrule

rule decode (!execute.stall(fetchDecodeQ.first()));
    execute.enqIt(newIt(fetchDecodeQ.first(), rf));
    fetchDecodeQ.deq();
endrule

method Action setPC …
method Action writeback …
endmodule

Are any changes needed in the methods?
Fetch Module Refinement

module mkFetch#(IMem iMem, Execute execute) (Fetch);
    FIFO#(Instr) fetchDecodeQ <- mkFIFO();
    ...
    Instr instr = iMem.read(pc);
    Iaddress predIa = pc + 1;
    method Action writeback(RName rd, Value v);
        rf.upd(rd,v);
    endmethod
    method Action setPC(Iaddress newPC);
        pc <= newPC;
        fetchDecodeQ.clear();
    endmethod
endmodule

More refinements to consider

- Multicycle execution module
- Multicycle memory
- Bypassing

Next time ...