Modular Refinement

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

March 2, 2011
http://csg.csail.mit.edu/6.375

Successive refinement & Modular Structure

Can we derive the 5-stage pipeline by successive refinement of a 2-stage pipeline?
Architectural refinements
- Separating Fetch and Decode
- Replace magic memory by multicycle memory
- Multicycle functional units
- ...

Nirav Dave, M.C. Ng, M. Pellauer, Arvind [Memocode 2010]
A design flow based on modular refinement

CPU as one module

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

Read method call
Action method call
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.

Interface definitions:

**Fetch and Execute**

```plaintext
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
```

Recursive modular organization

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

Unfortunately, the recursive module syntax is not so 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

```verbatim
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

```verbatim
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
```


March 2, 2011  http://csg.csail.mit.edu/6.375  L09-10
**Fetch Module**

```verilog
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);
    SFIFO#(InstTemplate) bu <- mkSPipelineFifo(findf);
    InstTemplate it = bu.first;
    rule execute ...

        method Action enqIt(InstTemplate it);
            bu.enq(it);
        endmethod
        method Bool stall(Instr instr);
            return (stallFunc(instr, bu));
        endmethod
endmodule
```
Execute Module Rule

```vHDL
rule execute (True);
  case (it) matches
    tagged EAdd\{dst::rd,srcl::va,srcl2::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

```vHDL
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
```

Modular refinement: 
Separating Fetch and Decode

module mkFetch#(IMem iMem, Execute execute) (Fetch);
FIFO#(Instr) fetch2DecodeQ <- mkPipelineFIFO();
Instr instr = iMem.read(pc);
Iaddress predIa = pc + 1;
...
rule fetch(True);
   pc <= predIa;
   fetch2DecodeQ.enq(instr);
endrule
rule decode(!execute.stall(fetch2DecodeQ.first()));
   execute.enqIt(newIt(fetch2DecodeQ.first(),rf));
   fetch2DecodeQ.deq();
endrule
method Action setPC ...
method Action writeback ...
endmodule

Fetch Module Refinement
Separating Fetch and Decode

module mkFetch#(IMem iMem, Execute execute) (Fetch);
FIFO#(Instr) fetch2DecodeQ <- mkPipelineFIFO();
Instr instr = iMem.read(pc);
Iaddress predIa = pc + 1;
...
rule fetch(True);
   pc <= predIa;
   fetch2DecodeQ.enq(instr);
endrule
rule decode(!execute.stall(fetch2DecodeQ.first()));
   execute.enqIt(newIt(fetch2DecodeQ.first(),rf));
   fetch2DecodeQ.deq();
endrule
method Action setPC ...
method Action writeback ...
endmodule

Are any changes needed in the methods?
Fetch Module Refinement

```verilog
module mkFetch#(IMem iMem, Execute execute) (Fetch);
  FIFO#(Instr) fetchDecodeQ <- mkFIFO();
  ...
  rule fetch ...
  rule decode ...

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

  method Action setPC(Iaddress newPC);
    pc <= newPC;
    fetch2DecodeQ.clear();
  endmethod
Endmodule
```

The stall signal definition guarantees that any order for the execution of the decode rule and writeback method is valid.

---

Modular refinement:
Replace magic memory by multicycle memory
The desired behavior

The desired behavior

\[ \text{fetch_and_decode}(\text{True}) \]

\[ \text{instr} \leftarrow \text{actionvalue} \]

\[ \text{imem.req}(\text{pc}) \triangleright \]

\[ \text{i} \leftarrow \text{imem.resp}(); \]

\[ \text{return i;} \]

\[ \text{endactionvalue} \]

\[ \text{execute.enqIt(newIt(instr,rf))} \]

\[ \text{when} \left( \text{!execute.stall(instr)} \right) ; \]

\[ \text{pc} \leftarrow \text{predIa}; \]

endrule

Action Connectives:

Par vs. Seq

- **Parallel compositions** (a1; a2)
  - Neither action observes others’ updates
  - Writes are disjoint
  - Natural in hardware
  
  \[ (\text{r1} \leftarrow \text{r2} ; \text{r2} \leftarrow \text{r1}) \]
  
  swaps r1 & r2

- **Sequential Connective** (a1 <$$> a2)
  - a2 observes a1’s updates
  - Still atomic
  - Not present in BSV because of implementation complications

We need to split the rule to get rid of <$$>
Predicating Actions:
Guards vs. Ifs

- **Guards affect their surroundings**

  \[(a1 \text{ when } p1) \ ; \ a2 \implies (a1 \ ; \ a2) \text{ when } p1\]

- **The effect of an “if” is local**

  \[(\text{if } p1 \text{ then } a1) \ ; \ a2 \implies \text{if } p1 \text{ then } (a1 \ ; \ a2) \text{ else } a2\]

  \textit{p1 has no effect on a2}

## Splitting the rule

```plaintext
rule fetch(True);
  imem.req(pc)
  pc <= predIa;
endrule

rule decode(True);
  let instr <- imem.resp();
  execute.enqIt(newIt(instr,rf)) when
  (!execute.stall(instr));
endrule
```

Suppose the PC was also needed to decode the instruction
Passing data from Fetch to Decode

```verilog
rule fetch(True);
    imem.req(pc);
    fet2decQ.enq(pc);
    pc <= predIa;
endrule

rule decode(True);
    let instr <- imem.resp();
    let pc = fet2decQ.first();
    fet2decQ.deq();
    execute.enqIt(newIt(instr,rf,pc))
        when (!execute.stall(instr));
endrule
```

Methods of Fetch module

To finish we need to change method setPC so that the next instruction sent to the Execute module is the correct one

```verilog
method setPC(Iaddress npc);
    pc <= npc;
    fet2decQ.clear();
endmethod
```
Multicycle memory:

Refined Execute Module Rule

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
      fetch.writeback(rd, va+vb); end
    tagged EBz {cond:.cv,addr:.av}:
      if (cv == 0) then begin
        fetch.setPC(av); end
      else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
      let val <- actionvalue
        dMem.req(Read {addr:av});
        v <- dMem.resp();
        return v; end
    tagged EStore{value:.vv,addr:.av}: begin
      dMem.req(Write {addr:av, val:vv});
      let val <- dMem.resp();
      end;
  endcase endrule
```

Splitting the Backend Rules:

The execute rule

```
rule execute (True);
  bu.deq();
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
      exec2wbQ.enq (WWB {dst: rd, val: va+vb});
    tagged EBz {cond:.cv,addr:.av}:
      if (cv == 0) then begin
        fetch.setPC(av); end
      else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
      let val <- actionvalue
        dMem.req(Read {addr:av});
        exec2wbQ.enq(WLd {dst:rd});
      end
    tagged EStore{value:.vv,addr:.av}: begin
      exec2wbQ.enq(WSt {}); end
  endcase endrule
```
Splitting the Backend Rules

The writeback rule

```
rule execute (True);
  ...
endrule

rule writeback(True);
  exec2wbQ.deq();
  case exec2wbQ.first() matches
    tagged WWB {dst: .rd, val: .v}: fetch.writeback(rd,v);
    tagged WLd {dst: .rd}:
      begin let v <- dMem.resp();
        fetch.writeback(rd,v);
      end
    tagged WSt {}:
      begin let ack <- dMem.resp(); end
    default: noAction;
  endcase
endrule
```

Correctness

- Stepwise refinement makes the verification task easier but does not eliminate it
- We still need to prove that each step is correct