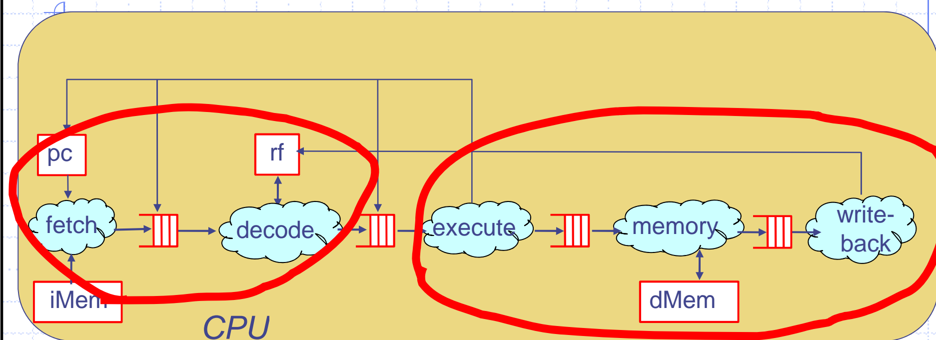
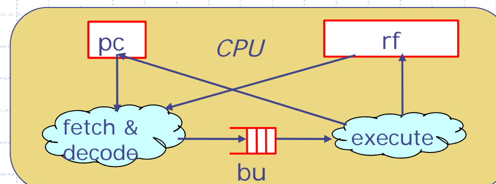
Modular Refinement

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Successive refinement & Modular Structure



Can we derive the 5-stage pipeline by successive refinement of a 2-stage pipeline?

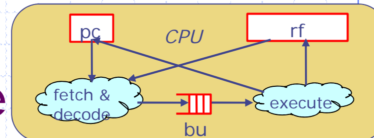


Architectural refinements

- Separating Fetch and Decode
- Replace magic memory by multicycle memory
- Multicycle functional units
- ...

First, let us examine our two-stage pipeline

Two-stage Pipeline




```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule

rule execAdd
   (it matches tagged EAdd{dst:.rd,src1:.va,src2
  rf.upd(rd, va+vb); bu.deq(); endrule
rule BzTaken(it matches tagged Bz {cond:.cv,addr:.av})
             &&& (cv == 0);
     pc <= av; bu.clear(); endrule
rule BzNotTaken(it matches tagged Bz {cond:.cv,addr:.av});
              &&& !(cv == 0);
     bu.deq(); endrule
rule execLoad(it matches tagged ELoad{dst:.rd,addr:.av});
  rf.upd(rd, dMem.read(av)); bu.deq(); endrule
rule execStore(it matches tagged EStore{value:.vv,addr:.av});
  dMem.write(av, vv); bu.deq(); endrule
```

X

fetch rule can execute concurrently with every execute rule except the BzTaken rule

Properties Required of Register File and FIFO for Instruction Pipelining

- Bypass Register File:
  - rf.upd(r1, v) < rf.sub(r2)
- Pipeline SFIFO
  - bu: {first , deq} < {find, enq} ⇒
    - bu.first < bu.find
    - bu.first < bu.enq
    - bu.deq < bu.find
    - bu.deq < bu.enq

Bypass Register File

```
module mkBypassRFFull(RegFile#(RName,Value));

  RegFile#(RName,Value) rf <- mkRegFileFullWCF();
  RWire#(Tuple2#(RName,Value)) rw <- mkRWire();

  method Action upd (RName r, Value d);
    rf.upd(r,d);
    rw.wset(tuple2(r,d));
  endmethod

  method Value sub(RName r);
    case rw.wget() matches
      tagged Valid {.wr,.d}:
                     return (wr==r) ? d : rf.sub(r);
      tagged Invalid:      return rf.sub(r);
    endcase
  endmethod
endmodule
```

"Config reg file"

One Element Searchable Pipeline SFIFO

```
module mkSFIFO1#(function Bool findf(tr r, t x))
                                        (SFIFO#(t,tr));
  Reg#(t)       data   <- mkRegU();
  Reg#(Bool)    full   <- mkConfigReg(False);
  RWire#(void)  deqEN  <- mkRWire();
  Bool          deqp = isValid (deqEN.wget()));
  method Action enq(t x) if (!full || deqp);
    full <= True;      data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
 method Bool find(tr r);
  return (findf(r, data) && full);
 endmethod endmodule
                              (full && !deqp));
```

(In the last method, "&& full);" is struck through and replaced by "(full && !deqp));".)

bu.enq > bu.deq
bu.enq > bu.first
bu.enq < bu.clear

bu.deq > bu.first
bu.deq < bu.clear

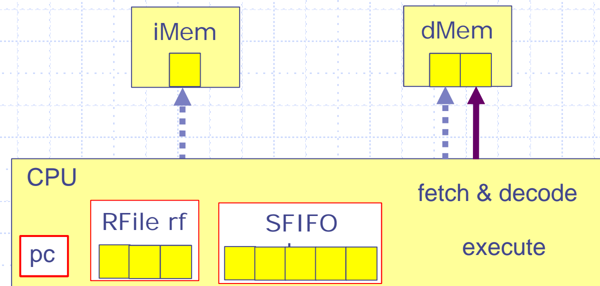
~~bu.find < bu.enq~~
~~bu.find < bu.deq~~
~~bu.find < bu.clear~~

bu.find < bu.enq
bu.find > bu.deq
bu.find < bu.clear

Suppose we used the wrong SFIFO? bu.find < bu.deq

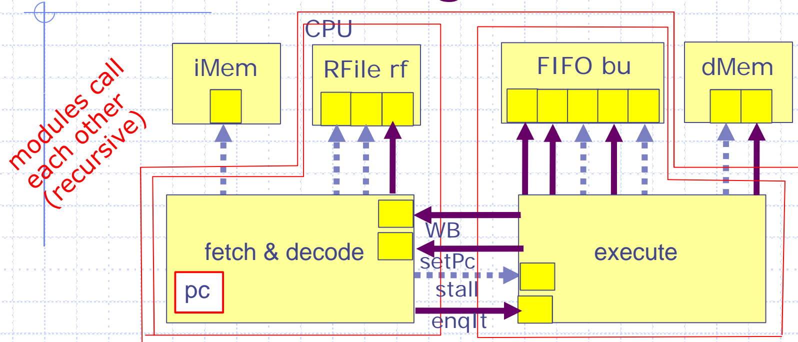
- Will the system produce wrong results?
  - NO because the fetch rule will simply conflict with the execute rules

CPU as one module



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

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

Interface definitions: Fetch and Execute

```
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

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

recursive calls

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

```
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

```
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
```

no change

Execute Module Rule

```
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 **stall**s
  - 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

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

```
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
```

no change (refers to `rf.upd(rd,v);`)

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

`iMem.read(pc)` needs to be split into a pair of request-response actions:

```
method Action iMem.req(Addr pc)
method Actionvalue#(Inst) iMem.res()
```

```
rule fetch_and_decode(True);
     instr <- actionvalue
                    imem.req(pc) <$>
                    i <- imem.resp();
                    return i;
               endactionvalue
     execute.enqIt(newIt(instr,rf))
          when (!execute.stall(instr));
     pc <= predIa; endrule
```

imem req and res change state

a guard is needed to avoid repeatedly requesting the stalled instr

Unfortunately, BSV does not have multicycle rules <$>

Action Connectives: Par vs. Seq

- Parallel compositions (a1;a2)
  - Neither action observes others' updates
  - Writes are disjoint
  - Natural in hardware

(r1 <= r2 ; r2 <= 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 when p1) ; a2 ==> (a1 ; a2) when p1

- The effect of an "if" is local

(if p1 then a1) ; a2 ==> if p1 then (a1 ; a2) else a2

p1 has no effect on a2

Splitting the rule

```
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

```
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
```

All data between actions passed through state (imem + fet2decQ)

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

```
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); 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
       let val <- actionvalue
                    dMem.req(Read {av}) <$>
                    v <- dMem.resp();
                    return v; endactionvalue;
       fetch.writeback(rd, val); bu.deq(); end
    tagged EStore{value:.vv,addr:.av}: begin
       dMem.req(Write {av, vv}); <$>
       let val  <- dMem.resp(); bu.deq(); 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}); end
    tagged EBz {cond:.cv,addr:.av}:
       if (cv == 0) then begin
         fetch.setPC(av); bu.clear(); end;
    tagged ELoad{dst:.rd,addr:.av}: begin
        dMem.req(Read {addr:av});
        exec2wbQ.enq(WLd {dst:rd}); end
    tagged EStore{value:.vv,addr:.av}: begin
        dMem.req(Write {addr:av, val:vv});
        exec2wbQ.enq(WSt {}); end
endcase endrule

rule writeback(True);
 ….
```


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
```

Final step

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

Splitting the Backend Rules:

The execute rule

Stall condition prevents fetched instruction from reading stale data

```
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}); end
    tagged EBz {cond:.cv,addr:.av}:
       if (cv == 0) then begin
         fetch.setPC(av); bu.clear(); end;
    tagged ELoad{dst:.rd,addr:.av}: begin
       dMem.req(Read {addr:av});
       exec2wbQ.enq(WLd {dst:rd}); end
    tagged EStore{value:.vv,addr:.av}: begin
       dMem.req(Write {addr:av, val:vv});
       exec2wbQ.enq(WSt {}); end
endcase endrule

rule writeback(True);
  ….
```

What about exec2wbQ? Should it be cleared?

No because all instructions in exec2wbQ are non speculative and must commit