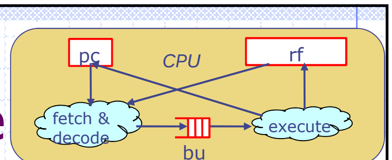
Concurrency and Modularity Issues in Processor pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Concurrency analysis

Two-stage Pipeline




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

For pipelining, we want the behavior in a clock-cycle to be execute < fetch_and_decode

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 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
```

Properties Required of Register File and FIFO for Instruction Pipelining

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

One Element Searchable Pipeline SFIFO

```
module mkSFIFO1#(function Bool findf(tr r, t x))
                                        (SFIFO#(t,tr));
  Reg#(t)       data  <- mkRegU();
  Reg#(Bool)    full  <- mkReg(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 && !deqp));
 endmethod endmodule
```

bu.first < bu.enq
bu.deq < bu.enq

bu.enq < bu.clear
bu.deq < bu.clear

bu.find < bu.enq
bu.deq < bu.find

Register File concurrency properties

- Normal Register File implementation guarantees:
  - rf.sub < rf.upd
    - that is, reads happen before writes in concurrent execution
- But concurrent rf.sub(r1) and rf.upd(r2,v) where r1 ≠ r2 behaves like both
  - rf.sub(r1) < rf.upd(r2,v)
  - rf.sub(r1) > rf.upd(r2,v)
- To guarantee rf.upd < rf.sub
  - Either bypass the input value to output when register names match
  - Or make sure that on concurrent calls rf.upd and rf.sub do not operate on the same register

True for our rules because of stalls but it is too difficult for the compiler to detect

Bypass Register File

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

  RegFile#(RName,Value) rf <- mkRegFileFull();
  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
```

Will work only if the compiler lets us ignore conflicts on the rf made by **mkRegFileFull**

"Config reg file"

Since our rules do not really require a Bypass Register File, the overhead of bypassing can be avoided by simply using the "Config Ref file"

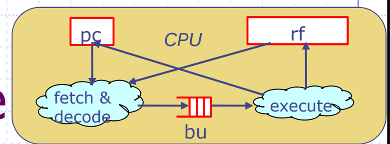
An aside

Unsafe modules

- Bluespec allows you to import Verilog modules by identifying wires that correspond to methods
- Such modules can be made safe either by asserting the correct scheduling properties of the methods or by wrapping the unsafe modules in appropriate Bluespec code

Concurrency analysis

Two-stage Pipeline




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

X

all concurrent cases work

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 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
```


Lot of nontrivial analysis but no change in processor code!

Needed Fifos and Register files with the appropriate concurrency properties

Bypassing

- After decoding the newIt function must read the new register values if available (i.e., the values that are still to be committed in the register file)
  - Will happen automatically if we use bypassRF
- The instruction fetch must not stall if the new value of the register to be read exists
  - The old stall function is correct but unable to take advantage of bypassing and stalls unnecessarily

The stall function for the asynchronous pipeline

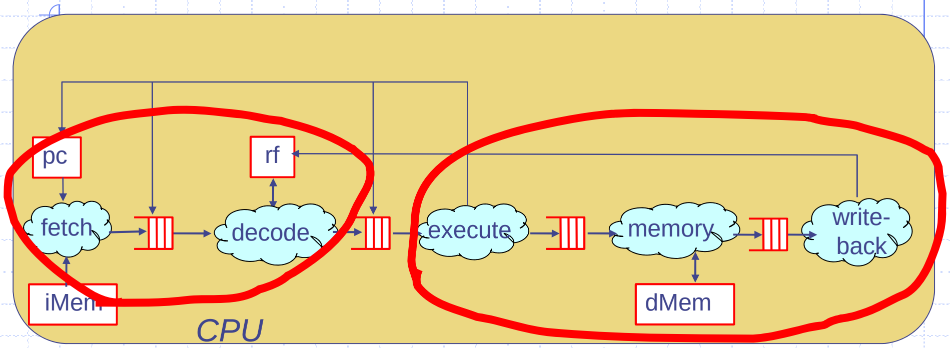
```
function Bool newStallFunc (Instr instr,
        SFIFO#(InstTemplate, RName) bu);
  case (instr) matches
   tagged Add {dst:.rd,src1:.ra,src2:.rb}:
      return (bu.find(ra) || bu.find(rb));
   tagged Bz    {cond:.rc,addr:.addr}:
      return (bu.find(rc) || bu.find(addr));
   …
```

bu.find in our Pipeline SFIFO happens after deq. This means that if bu can hold at most one instruction like in the synchronous case, we do not have to stall. Otherwise, we will still need to check for hazards and stall.

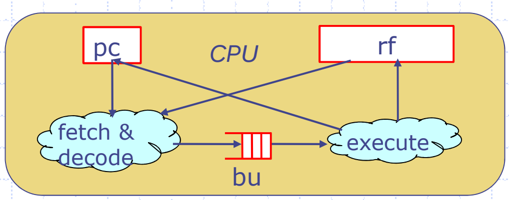
No change in the stall function

Modular Refinement

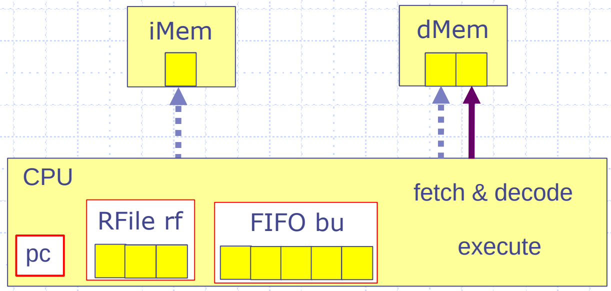
Successive refinement & Modular Structure



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



CPU as one module



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

Read method call

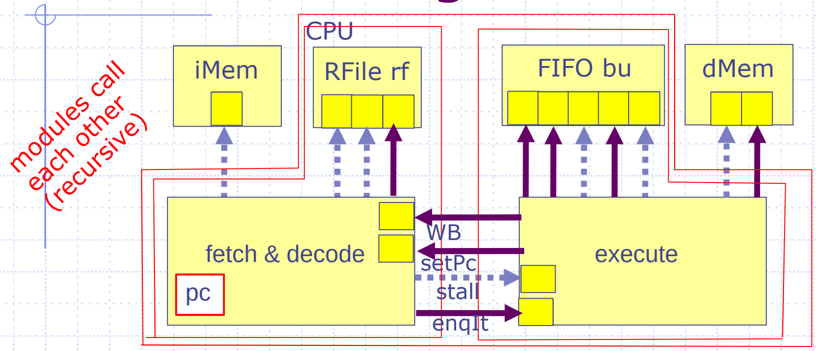
Action method call

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
                          <- mkPipelineSFifo(findf);
// Some definitions
       Instr     instr = iMem.read(pc);
      Iaddress predIa = pc + 1;
// Rules
      rule fetch_decode ...
       rule execute ...
endmodule
```

you have seen this before

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)();
  Execute  execute  <- mkExecute(dMem, fetch);
  Fetch fetch  <- mkFetch(iMem, execute);
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(Instr instr)
endinterface
```

recursive calls

Unfortunately, recursive module syntax is not as simple

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
```

no change

Execute Module

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

   SFIFO#(InstTemplate) bu <- mkSLoopyFifo(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
```


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
```

Modular refinements

- Separating Fetch and Decode
- Replace magic memory by multicycle memory
- Multicycle execution module
- …

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
```

no change