Elastic Pipelines: 
Concurrency Issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 28, 2011
http://csg.csail.mit.edu/6.375

Inelastic vs Elastic Pipelines

◆ In a Inelastic pipeline:
  - typically only one rule; the designer controls precisely which activities go on in parallel
  - downside: The rule can get too complicated -- easy to make a mistake; difficult to make changes

◆ In an Elastic pipeline:
  - several smaller rules, each easy to write, easier to make changes
  - downside: sometimes rules do not fire concurrently when they should
It is better to think in terms of FIFOs as opposed to pipeline registers.

Fetch & Decode Rule: corrected

```
rule decodeAdd (instr matches Add(dst:.rd, src1:.ra, src2:.rb) 
    &&& !bu.find(ra) &&& !bu.find(rb))
    bu.enq (EAdd(dst:rd, op1:rf[ra], op2:rf[rb]));
    pc <= predIa;
endrule
```
**SFIFO (glue between stages)**

```verilog
interface SFIFO#(type t, type tr);
method Action enq(t); // enqueue an item
method Action deq(); // remove oldest entry
method t first(); // inspect oldest item
method Action clear(); // make FIFO empty
method Bool find(tr); // search FIFO
endinterface
```

---

**Two-Stage Pipeline**

```verilog
module mkCPU#(Mem iMem, Mem dMem)(Empty);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    SFIFO#(InstTemplate, RName) bu <- mkSFifo(findf);
    Instr instr = iMem.read(pc);
    Iaddress predIa = pc + 1;
    InstTemplate it = bu.first();
    rule fetch_decode ...
endmodule
```
Rules for Add

rule decodeAdd (instr matches Add{dst:.rd, src1:.ra, src2:.rb})
    bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});
    pc <= predIa;
endrule

rule executeAdd (it matches EAdd{dst:.rd, op1:.va, op2:.vb})
    rf.upd(rd, va + vb);
    bu.deq();
endrule

Fetch & Decode Rule:

Reexamined

rule decodeAdd (instr matches Add{dst:.rd, src1:.ra, src2:.rb})
    bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});
    pc <= predIa;
endrule
Rules for Branch

rule-atomicity ensures that pc update, and discard of pre-
fetched instrs in bu, are done consistently

rule decodeBz(instr matches Bz{condR:.rc,addrR:.addr}) &&&
!bu.find(rc) &&& !bu.find(addr));
  bu.enq (EBz{cond:rf[rc],tAddr:rf[addr]});
endrule

rule bzTaken(it matches EBz{cond:.vc,tAddr:.va})
  pc <= va;   bu.clear(); endrule
rule bzNotTaken (it matches EBz{cond:.vc,tAddr:.va}) &&&
(vc != 0));
  bu.deq; endrule

Fetch & Decode Rule

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

definition newIt(Instr instr);
  case (instr) matches
    tagged Add {dst:.rd,src1:.ra,src2:.rb}:
      return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
    tagged Bz {condR:.rc,addrR:.addr}:
      return EBz{cond:rf[rc],tAddr:rf[addr]};
    tagged Load {dst:.rd,addrR:.addr}:
      return ELoad{dst:rd,addrR:rf[addr]};
    tagged Store{valueR:.v,addrR:.addr}:
      return EStore{val:rf[v],addr:rf[addr]};
  endcase
endfunction
The Stall Signal

Bool stall = stallFunc(instr, bu);

function Bool stallFunc (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   {condR:.rc,addrR:.addr}:
            return (bu.find(rc) || bu.find(addr));
        tagged Load {dst:.rd,addrR:.addr}:
            return (bu.find(addr));
        tagged Store {valueR:.v,addrR:.addr}:
            return (bu.find(v)) || bu.find(addr));
    endcase
endfunction

This need to search the contents of the FIFO is why we need an SFIFO, not just a FIFO

The findf function

When we make a searchable FIFO we need to supply a function that determines if a register is going to be updated by an instruction template. mkSFifo can be parameterized by such a search function.

SFIFO#(InstrTemplate, RName) bu <- mkSFifo(findf);

function Bool findf (RName r, InstrTemplate it);
    case (it) matches
        tagged EAdd{dst:.rd,op1:.v1,op2:.v2}:
            return (r == rd);
        tagged EBz {cond:.c,tAddr:.a}:
            return (False);
        tagged ELoad{dst:.rd,addr:.a}:
            return (r == rd);
        tagged EStore{val:.v,addr:.a}:
            return (False);
    endcase
endfunction
**Execute Rule**

```
rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd, op1:.va, op2:.vb}:
            begin rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv, tAddr:.av}:
            if (cv == 0) then
                begin pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd, addr:.av}:
            begin rf.upd(rd, dMem.read(av)); bu.deq(); end
        tagged EStore{val:.vv, addr:.av}:
            begin dMem.write(av, vv); bu.deq(); end
    endcase
endrule
```

**Concurrency**

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

rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd, op1:.va, op2:.vb}:
            begin rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv, tAddr:.av}:
            if (cv == 0) then
                begin pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd, addr:.av}:
            begin rf.upd(rd, dMem.read(av)); bu.deq(); end
        tagged EStore{val:.vv, addr:.av}:
            begin dMem.write(av, vv); bu.deq(); end
    endcase
endrule
```
The tension

- If the two rules never fire in the same cycle then the machine can hardly be called a pipelined machine
  - Scheduling cannot be too conservative

- If both rules are enabled and are executed together then in some cases wrong results would be produced
  - Too aggressive a scheduling would violate one-rule-at-time-semantics

Case 1: Back-to-back dependencies?

Case 2: Branch taken?

Execution rules

Split the execution rule for analysis

```
rule execAdd(it matches tagged EAdd{dst:.rd,op1:.va,op2:.vb});
rf.upd(rd, va+vb); bu.deq(); endrule

rule bzTaken(it matches tagged EBz {cond:.cv,tAddr:.av});
  &&& (cv == 0);
  pc <= av; bu.clear(); endrule

rule bzNotTaken(it matches tagged EBz {cond:.cv,tAddr:.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{val:.vv,addr:.av});
  dMem.write(av, vv); bu.deq(); endrule
```
**Concurrency analysis**

**Add Rule**

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

```haskell
rule execAdd(it matches tagged EAdd{dst:.rd,op1:.va,op2:.vb});
    rf.upd(rd, va+vb); bu.deq();
endrule
```

- **fetch < execAdd**
  - rf: sub < upd
  - bu: {find, enq} < {first, deq}
- **execAdd < fetch**
  - rf: sub > upd
  - bu: {find, enq} > {first, deq}

---

**What concurrency do we want?**

- Suppose bu is empty initially

- If fetch and execAdd happened in the same cycle and the meaning was:
  - fetch < execAdd
    - instructions will fly through the FIFO (No pipelining!)
    - rf and bu modules will need the properties:
      - rf: sub < upd
      - bu: {find, enq} < {first, deq}
  - execAdd < fetch
    - execAdd will make space for the fetched instructions (i.e., how pipelining is supposed to work)
    - rf and bu modules will need the properties:
      - rf: upd < sub
      - bu: {first, deq} < {find, enq}
Concurrency analysis

Branch Rules

\[
\text{rule fetch\_and\_decode (\neg stallfunc(instr, bu)):
\begin{align*}
\text{bu}.\text{enq}(\text{newIt}(\text{instr}, \text{rf})); \\
\text{pc} &\leftarrow \text{predIa};
\end{align*}
\text{endrule}
\]

\text{Rule bzTaken(it matches tagged EBz \{cond:.cv,tAddr:.av\})} \land (\text{cv} = 0)); \\
\text{pc} &\leftarrow \text{av}; \text{bu}.\text{clear}(); \text{endrule}

\text{rule bzNotTaken(it matches tagged EBz \{cond:.cv,tAddr:.av\})} \land (\text{cv} = 0)); \\
\text{bu}.\text{deq}(); \text{endrule}

\text{bzTaken} < \text{fetch} \Rightarrow \\
\text{bu: \{first, deq\} < \{find, enq\}}

\text{bzNotTaken} < \text{fetch} \Rightarrow \\
\text{bu: \{first, deq\} < \{find, enq\}}

Load-Store Rules

\text{rule fetch\_and\_decode (\neg stallfunc(instr, bu)):
\begin{align*}
\text{bu}.\text{enq}(\text{newIt}(\text{instr}, \text{rf})); \\
\text{pc} &\leftarrow \text{predIa};
\end{align*}
\text{endrule}

\text{rule execLoad(it matches tagged ELoad\{dst:.rd, addr:.av\})}:
\begin{align*}
\text{rf}.\text{upd}(\text{rd}, \text{dMem}.\text{read}(\text{av})); \text{bu}.\text{deq}();
\end{align*}
\text{endrule}

\text{rule execStore(it matches tagged EStore\{val:.vv, addr:.av\})}:
\begin{align*}
\text{dMem}.\text{write}(\text{av}, \text{vv}); \text{bu}.\text{deq}();
\end{align*}
\text{endrule}

\text{execLoad} < \text{fetch} \Rightarrow \\
\text{rf: upd < sub; bu: \{first, deq\} < \{find, enq\}}

\text{execStore} < \text{fetch} \Rightarrow \\
\text{bu: \{first, deq\} < \{find, enq\}}
Properties Required of Register File and FIFO for Instruction Pipelining

Register File:
- \( rf.\text{upd}(r1, v) < rf.\text{sub}(r2) \)
- Bypass RF

FIFO
- \( \text{bu: } \{\text{first }, \text{deq}\} < \{\text{find}, \text{enq}\} \Rightarrow \)
  - \( \text{bu.first} < \text{bu.find} \)
  - \( \text{bu.first} < \text{bu.enq} \)
  - \( \text{bu.deq} < \text{bu.find} \)
  - \( \text{bu.deq} < \text{bu.enq} \)
- Pipeline SFIFO

One Element Searchable Pipeline SFIFO

```verilog
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
```
Suppose we used the wrong SFIFO? \( \text{bu.find} < \text{bu.deq} \)

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

Register File concurrency properties

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

True for our rules because of stalls but it is too difficult for the compiler to detect
Since our rules do not really require a Bypass Register File, the overhead of bypassing can be avoided by simply using the “Config Regfile”
Concurrency analysis

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:.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 elastic pipeline

```pseudocode
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 inelastic case, we do not have to stall. Otherwise, we will still need to check for hazards and stall.

No change in the stall function