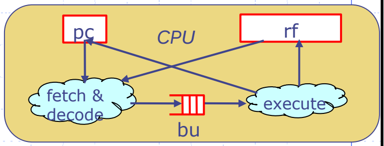
Asynchronous Pipelines: *Concurrency Issues*

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Synchronous vs Asynchronous Pipelines

- In a synchronous 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 asynchronous pipeline:
  - several smaller rules, each easy to write, easier to make changes
  - *downside:* sometimes rules do not fire concurrently when they should

Two-stage Asynchronous Pipeline



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

Can these rules fire concurrently ?

Does it matter?

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
         rf.upd(rd, va+vb); bu.deq(); end
    tagged EBz {cond:.cv,addr:.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{value:.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
- If both rules fire in parallel every cycle when they are enabled, then wrong results would be produced

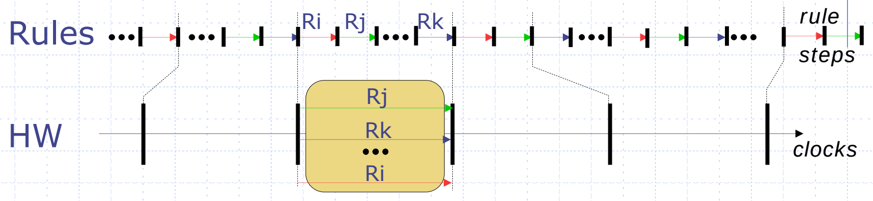
The compiler issue

- Can the compiler detect all the conflicting conditions? yes
  - Important for correctness
- Does the compiler detect conflicts that do not exist in reality? yes
  - False positives lower the performance
  - The main reason is that sometimes the compiler cannot detect under what conditions the two rules are mutually exclusive or conflict free
- What can the user specify easily?
  - Rule priorities to resolve nondeterministic choice

In many situations the correctness of the design is not enough; the design is not done unless the performance goals are met

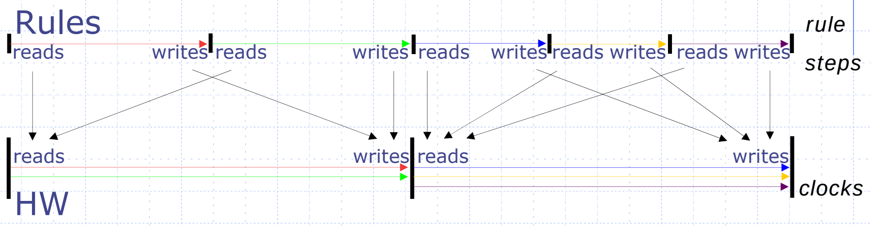
some insight into

Concurrent rule firing



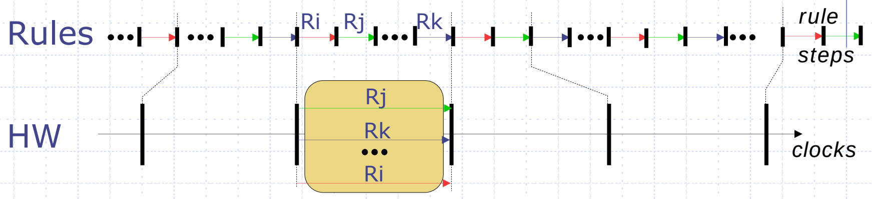
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness



- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution
- Consequence: the HW can never reach a state unexpected in the rule semantics

Executing Multiple Rules Per Cycle: *Conflict-free rules*

```
rule ra (z > 10);
   x <= x + 1;
endrule

rule rb (z > 20);
   y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb or equivalently rb < ra

Rule$_a$ and Rule$_b$ are conflict-free if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow$ 1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$

2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

Mutually Exclusive Rules

- Rule$_a$ and Rule$_b$ are mutually exclusive if they can never be enabled simultaneously

$$\forall s \,.\, \pi_a(s) \Rightarrow \sim \pi_b(s)$$

Mutually-exclusive rules are Conflict-free by definition

Executing Multiple Rules Per Cycle: *Sequentially Composable rules*

```
rule ra (z > 10);
  x <= y + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb

Rule$_a$ and Rule$_b$ are sequentially composable if
$\forall s\ .\ \pi_a(s) \wedge \pi_b(s) \Rightarrow$ 1. $\pi_b(\delta_a(s))$
2. $Prj_{R(Rb)}(\delta_b(s)) == Prj_{R(Rb)}(\delta_b(\delta_a(s)))$

- R(Rb) is the range of rule Rb
- Prj_{st} is the projection selecting st from the total state

Compiler determines if two rules can be executed in parallel

Rule$_a$ and Rule$_b$ are conflict-free if
$\forall s\ .\ \pi_a(s) \wedge \pi_b(s) \Rightarrow$
1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

$D(Ra) \cap R(Rb) = \phi$
$D(Rb) \cap R(Ra) = \phi$
$R(Ra) \cap R(Rb) = \phi$

Rule$_a$ and Rule$_b$ are sequentially composable if
$\forall s\ .\ \pi_a(s) \wedge \pi_b(s) \Rightarrow$
1. $\pi_b(\delta_a(s))$
2. $Prj_{R(Rb)}(\delta_b(s)) == Prj_{R(Rb)}(\delta_b(\delta_a(s)))$

$D(Rb) \cap R(Ra) = \phi$

These conditions are sufficient but not necessary

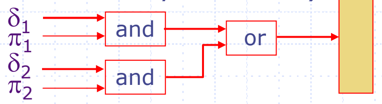
These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

Parallel execution of CF and SC rules does not increase the critical path delay

Muxing structure

- Muxing logic requires determining for each register (action method) the rules that update it and under what conditions

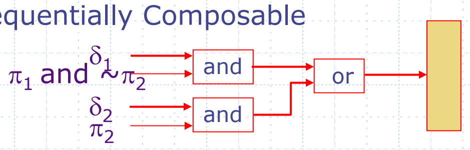
Conflict Free/Mutually Exclusive)



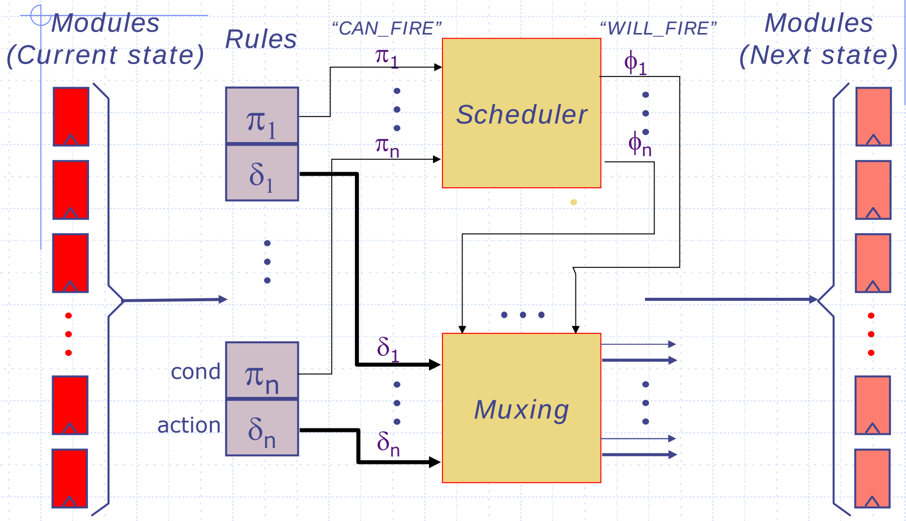
If two CF rules update the same element then they must be *mutually exclusive*

$(\pi_1 \rightarrow \sim\pi_2)$

Sequentially Composable



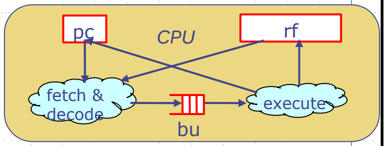
Scheduling and control logic



$\phi_i \wedge \phi_j \Rightarrow R_i$ *and* R_j *are conflict-free or sequentially composable*

Concurrency analysis

Two-stage Pipeline



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

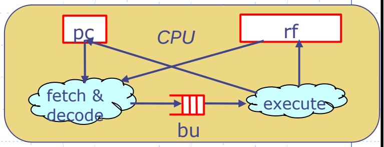
conflicts around:
pc, bu, rf

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
         rf.upd(rd, va+vb); bu.deq(); end
    tagged EBz {cond:.cv,addr:.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{value:.vv,addr:.av}: begin
         dMem.write(av, vv); bu.deq(); end
  endcase endrule
```

Let us split this rule for the sake of analysis

Concurrency analysis

Add Rule



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

rf: sub
bu: find, enq
pc: read,write

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd(rd, va+vb); bu.deq();
endrule
```

- fetch < execAdd ⇒

execAdd
rf: upd
bu: first, deq

- execAdd < fetch ⇒

Do either of these concurrency properties hold ?

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

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
```

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

FIFOs

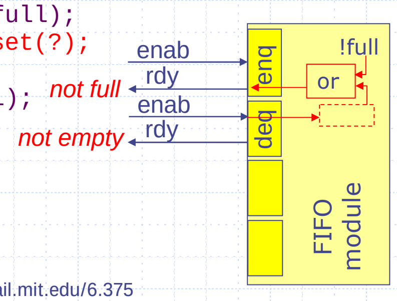
- Ordinary one element FIFO
  - deq & enq conflict – won't do
- Loopy FIFO
  - deq < enq
- Bypass FIFO
  - enq < deq

What about first, clear, find?

Concurrency analysis

One Element “Loopy” FIFO

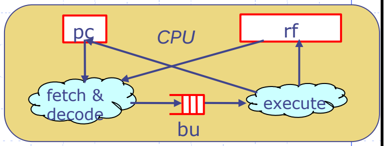
```
module mkLFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool          deqp = isValid (deqEN.wget()));
  method Action enq(t x) if
                (!full || deqp);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```



One Element Searchable FIFO

```
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
```

What concurrency do we want?

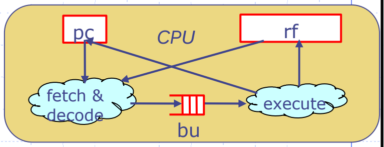


Suppose bu is empty initially

- If fetch and execAdd happened in the same cycle and the meaning was:
  - fetch < execAdd
  - execAdd < fetch

Concurrency analysis

Branch Rules



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

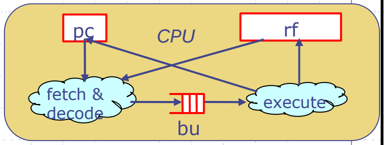
```
rule execBzTaken(it matches tagged Bz {cond:.cv,addr:.av}
             &&& (cv == 0));
     pc <= av; bu.clear(); endrule

rule execBzNotTaken(it matches tagged Bz {cond:.cv,addr:.av}
             &&& !(cv == 0));
     bu.deq(); endrule
```

- execBzTaken < fetch ?
  - Should be treated as conflict – give priority to execBzTaken
- execBzNotTaken < fetch ?

  bu: {first , deq} < {find, enq}

Concurrency analysis

Load-Store Rules



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

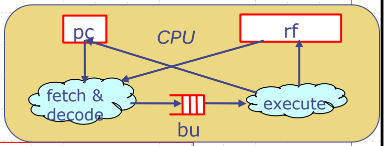
- execLoad < fetch ?
- execStore < fetch ?

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
  - Loopy FIFO

Concurrency analysis

Two-stage Pipeline



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

It all works

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd(rd, va+vb); bu.deq(); endrule

rule execBz(it matches tagged Bz {cond:.cv,addr:.av});
  if (cv == 0) then begin
     pc <= av; bu.clear(); end
  else 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 synchronous pipeline

```
function Bool newStallFunc (Instr instr,
         Reg#(Maybe#(InstTemplate))  buReg);


    return (false);
```

Previously we stalled when ra matched the destination register of the instruction in the execute stage. Now we bypass that information when we read, so no stall is necessary.

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 loopy-searchable FIFO 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