Modeling Processors: *Concurrency Issues*

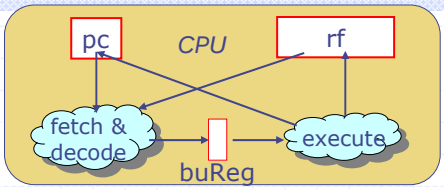
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

The Plan

- Two-stage synchronous pipeline ⇐
  - Bypassing issues
- Two-stage asynchronous pipeline
  - Concurrency Issues

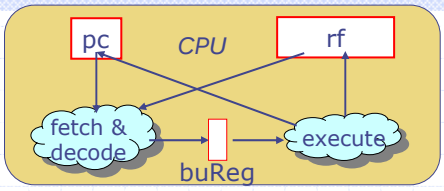
Some understanding of simple processor pipelines is needed to follow this lecture

Synchronous Pipeline




```
rule SyncTwoStage (True);
  let instr = iMem.read(pc);
  let stall = stallfuncR(instr,buReg);

  let fetchAction = action
    if(!stall) pc <= predIa;
    buReg <= (stall) ? Invalid : Valid newIt(instr);
    endaction;

  case (buReg) matches
    …


  endcase
endcase endrule
```


Synchronous Pipeline




```
rule SyncTwoStage (True);
…
  case (buReg) matches
    tagged Invalid: fetchAction;
    tagged Valid .it: begin
     case (it) matches
      tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
         rf.upd(rd, va+vb); fetchAction; end
      tagged EBz {cond:.cv,addr:.av}:
         if (cv == 0) then begin
            pc <= av; buReg <= Invalid; end
         else fetchAction;
      tagged ELoad{dst:.rd,addr:.av}: begin
         rf.upd(rd, dMem.read(av)); fetchAction; end
      tagged EStore{value:.vv,addr:.av}: begin
         dMem.write(av, vv); fetchAction; end
  endcase
endcase endrule
```

The Stall Function

```
function Bool stallfuncR (Instr instr,
        Maybe#(InstTemplate) buReg);
   case (buReg) matches
     tagged Invalid: return False;
     tagged Valid .it:
        case (instr) matches
          tagged Add   {dst:.rd,src1:.ra,src2:.rb}:
            return (findf(ra,it) || findf(rb,it));
          tagged Bz    {cond:.rc,addr:.addr}:
            return (findf(rc,it) || findf(addr,it));
          tagged Load  {dst:.rd,addr:.addr}:
            return (findf(addr,it));
          tagged Store {value:.v,addr:.addr}:
       return (findf(v,it) || findf(addr,it));
  endcase
endfunction
```


The findf function

```
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,addr:.a}:
                                return (False);
    tagged ELoad{dst:.rd,addr:.a}:
                                return (r == rd);
    tagged EStore{value:.v,addr:.a}:
                                return (False);
  endcase endfunction
```

Bypasses

Bypassing will affect ...

- The newIt function: After decoding it must read the new register values if available (i.e., the values that are still to be committed in the register file)
- The Stall function: The instruction fetch must not stall if the new value of the register to be read exists
  - In our specific design we never stall because the new register value will be available

The bypassRF function

```
function bypassRF(r,tobeCommitted);
case (tobeCommitted) matches
 tagged (Valid {.rd, .v} &&& (r==rd)): return (v);
 tagged  Invalid: return (rf[r]);
endcase;
endfunction
```


Modified Decode function

```
function InstrTemplate  newItBy (instr,tobeCommitted);
    let bRf(x) = bypassRF(x, tobeCommitted);
    case (instr) matches
        tagged Add {dst:.rd,src1:.ra,src2:.rb}:
           return EAdd{dst:rd,op1:bRF(ra),op2:bRF(rb)};
        tagged Bz  {cond:.rc,addr:.addr}:
           return EBz{cond:bRF(rc),addr:bRF(addr)};
        tagged Load {dst:.rd,addr:.addr}:
           return ELoad{dst:rd,addr:bRF(addr)};
        tagged Store{value:.v,addr:.addr}:
           return EStore{value:bRF(v),addr:bRF(addr)};
    endcase endfunction
```

Replace each registerfile read by function bypassRF(ra) which will return the newly written value if it exists

Synchronous Pipeline *with bypassing*

```
rule SyncTwoStage (True);
  let instr = iMem.read(pc);
  let stall = newstallfuncR(instr,buReg);
  let fetchAction(tobeCommitted) = action
       if(!stall) pc <= predIa;
          buReg <= (stall) ? Invalid :
                   Valid newByIt(instr,tobeCommitted);
        endaction;
  case (buReg) matches
    …

  endcase
endcase endrule
```


Synchronous Pipeline *with bypassing*

```
rule SyncTwoStage (True); …
  case (buReg) matches
    tagged Invalid: fetchAction(Invalid);
    tagged Valid .it: begin
     case (it) matches
      tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
       let v = va + vb;
       rf.upd(rd,t); fetchAction(Valid tuple2(rd,v));end
      tagged EBz {cond:.cv,addr:.av}:
       if (cv == 0) then begin
          pc <= av; buReg <= Invalid; end
       else fetchAction(Invalid);
      tagged ELoad{dst:.rd,addr:.av}: begin
       let v = dMem.read(av);
       rf.upd(rd,v); fetchAction(Valid tuple2(rd,v));end
      tagged EStore{value:.vv,addr:.av}: begin
        dMem.write(av, vv); fetchAction(Invalid);end
  endcase endcase endrule
```

The New Stall Function

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


   return (false);
```

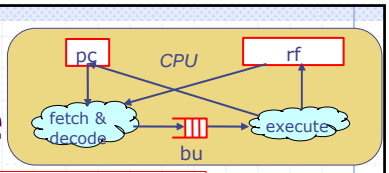
Previously we stalled when ra matched the destination register of the instruction in the execute stage. Now we bypass that information when we read, so no stall is necessary.

The Plan

- Two-stage synchronous pipeline
  - Bypassing issues
- Two-stage asynchronous pipeline ⇐
  - Concurrency Issues

Some understanding of simple processor pipelines is needed to follow this lecture

Two-stage 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 every cycle they are enabled, then wrong results would be produced

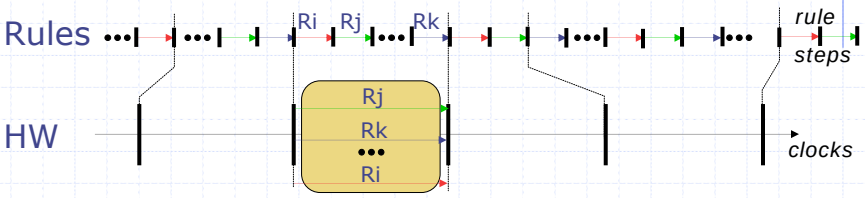
The compiler issue

- Can the compiler detect all the conflicting conditions?
  - Important for correctness
- Does the compiler detect conflicts that do not exist in reality?
  - 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 some situations correctness of the design is not enough; the design is not done unless the performance goals are met

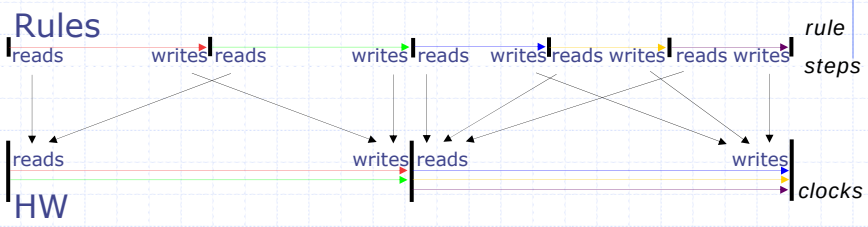
some insight into

Concurrent rule firing



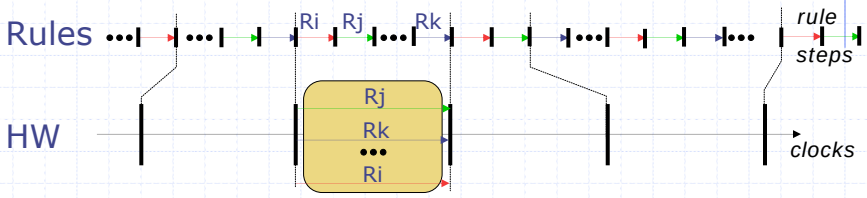
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

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 (i.e., CF or SC)
- Consequence: the HW can never reach a state unexpected in the rule semantics

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

$\forall s \; . \; \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$

and $\delta_{ab}(s)$ is implemented as $\delta_b(\delta_a(s))$

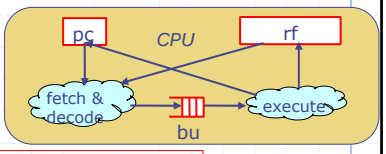
$D(\pi_b) \cap R(Ra) = \phi$

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

These conditions are sufficient but not necessary.
Parallel execution of CF and SC rules does not increase the critical path delay

Concurrency analysis

Two-stage Pipeline




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

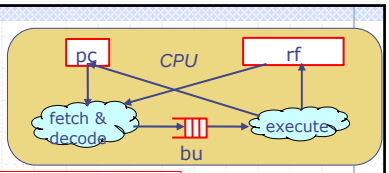
conflicts around: pc, bu, rf

```
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 ⇒

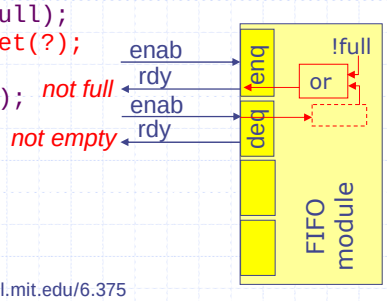
Do either of these concurrency properties hold ?

Concurrency analysis

One Element "Loopy" FIFO

From Lecture L07

```
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

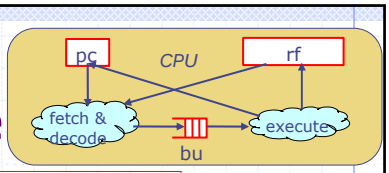
```
module mkSFIFO1#(function Bool findf(tr r, t x))
                                        (SFIFO#(t,tr));
  Reg#(t)      data  <- mkRegU();
  Reg#(Bool)   full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool         deqp = isValid (deqEN.wget()));
  method Action enq(t x) if (!full || 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
```


Register File concurrency properties

- Register File implementation would 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)

Concurrency analysis

Two-stage Pipeline




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

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

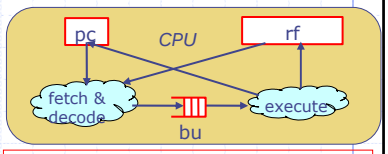
```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd(rd, va+vb); bu.deq();
endrule
```

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

execAdd
rf: upd
bu: first, deq

Do either of these concurrency properties hold ?

What concurrency do we want?

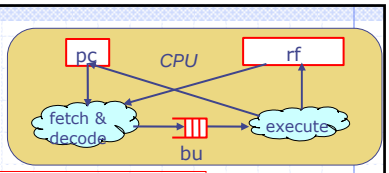


Suppose bu is empty initially

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

Concurrency analysis

Branch Rules



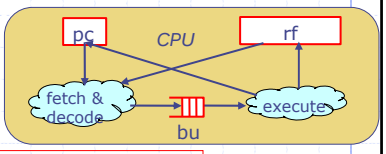

```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

```
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 ?
- execBzNotTaken < fetch ?

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
```

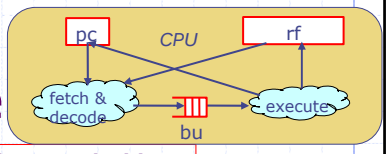
- execLoad < fetch ?
- execStore < fetch ?

Properties Required of Register File and FIFO for Instruction Pipelining

- Register File:
  - rf.upd(r1, v) < rf.sub(r2)
  - Our construction of stall guarantees that r1 ≠ r2 in concurrent calls
  - We can assert no conflict here
- FIFO
  - bu: {first , deq} < {find, enq} ⇒
    - bu.first < bu.find
    - bu.first < bu.enq
    - bu.deq < bu.find
    - bu.deq < bu.enq

Concurrency analysis

Two-stage Pipeline




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

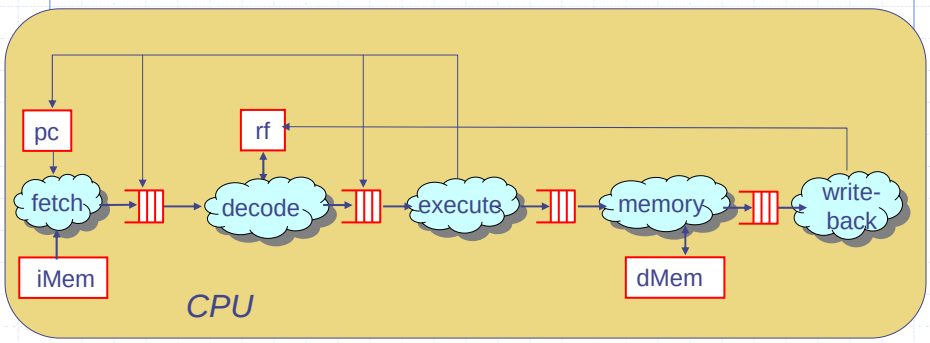
It all works

```
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

Processor Pipelines



The problem of exploiting the right amount of concurrency is quite difficult in complex processor pipelines