Blusepc-5:

Dead cycles, bubbles and Forwarding in Pipelines

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Topics

- Simultaneous enq & deq in a FIFO
- The RWire solution
- Dead cycle elimination in the IP circular pipeline code
- Two-stage processor pipeline
- Value forwarding to reduce bubbles

Implicit guards (conditions)

- Rule

  rule <name> (<guard>); <action>; **endrule**

  where

  <action> ::= r <= <exp>
  | m.g(<exp>)
  | **if** (<exp>) <actions> **endif**

Guards vs If's

- A guard on one action of a parallel group of actions affects every action within the group

  (a1 when p1); (a2 when p2)

  ==> (a1; a2) when (p1 && p2)

- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if (p1) a1); a2

  p1 has no effect on a2 ...

- Mixing ifs and whens

  (if (p) (a1 when q)) ; a2

  ≡ ((if (p) a1); a2) when (p&q | !p)

Example: making guards explicit

```
rule recirculate (True);
   if (p) fifo.enq(8);
   r <= 7;
endrule
```

```
rule recirculate ((p && fifo.engG) || !p);
   if (p) fifo.enqB(8);
   r <= 7;
endrule
```


A problem ... *(from the last lecture)*

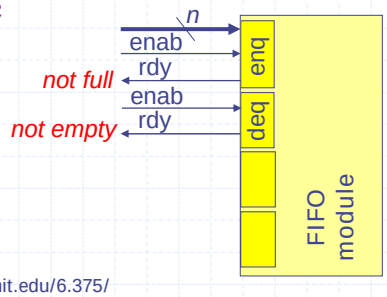
```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .tok} = fifo.first();
   if (isLeaf(p)) cbuf.put(tok, p);
   else begin
        fifo.enq(tuple2(rip << 8, tok));
        ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```

The fifo needs to be able to do enq and deq simultaneously for this rule to make sense

One Element FIFO

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False;
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

enq and deq cannot even be enabled together much less fire concurrently!



RWire to rescue

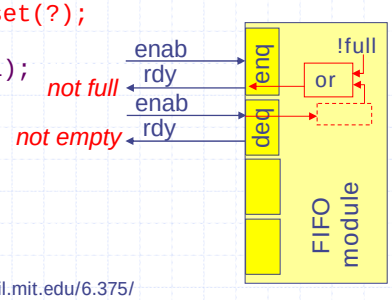
```
interface RWire#(type t);
        method Action wset(t x);
        method Maybe#(t) wget();
endinterface
```

Like a register in that you can read and write it but unlike a register

- read happens after write
- data disappears in the next cycle

One Element "Loopy" FIFO

```
module mkLFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  method Action enq(t x) if
              (!full || isValid (deqEN.wget()));
    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
```

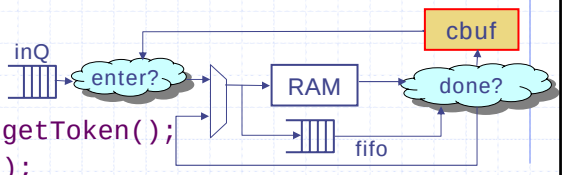


Problem solved!

```
LFIFO fifo <- mkLFIFO;  // use a loopy fifo

rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .tok} = fifo.first();
   if (isLeaf(p)) cbuf.put(tok, p);
   else begin
       fifo.enq(tuple2(rip << 8, tok));
       ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```

The Dead Cycle Problem




```
rule enter (True);
   Token tok <- cbuf.getToken();
   IP ip = inQ.first();
   ram.req(ext(ip[31:16]));
   fifo.enq(tuple2(ip[15:0], tok)); inQ.deq();
endrule
```

Can a new request enter the system simultaneously with an old one leaving?

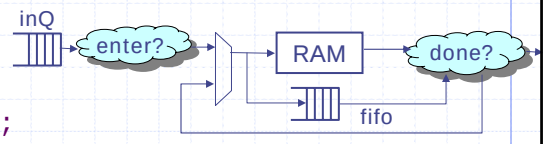
```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .tok} = fifo.first();
   if (isLeaf(p)) cbuf.put(tok, p);
   else begin
        fifo.enq(tuple2(rip << 8, tok));
        ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```


Scheduling conflicting rules

- When two rules conflict on a shared resource, they cannot both execute in the same clock
- The compiler produces logic that ensures that, when both rules are applicable, only one will fire
  - Which one?

source annotations

A slightly simpler example




```
rule enter (True);
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(ip[15:0]); inQ.deq();
endrule
```

```
rule recirculate (True);
   TableEntry p = ram.peek(); ram.deq();
   IP rip = fifo.first();
   if (isLeaf(p)) outQ.enq(p);
   else begin
       fifo.enq(rip << 8);
       ram.req(p + rip[15:8]);
     end
   fifo.deq();
endrule
```


Rule Spliting

```
rule foo (True);
   if (p) r1 <= 5;
   else r2 <= 7;
endrule
```

≡

```
rule fooT (p);
   r1 <= 5;
endrule

rule fooF (!p);
   r2 <= 7;
endrule
```

Spliting the recirculate rule

```
rule recirculate (!isLeaf(ram.peek()));
   IP rip = fifo.first(); fifo.enq(rip << 8);
   ram.req(ram.peek() + rip[15:8]);
   fifo.deq(); ram.deq();
endrule
```

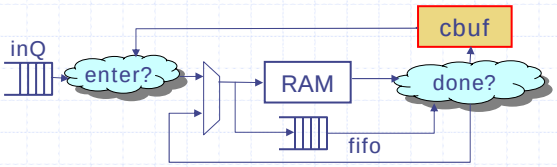
```
rule exit (isLeaf(ram.peek()));
   outQ.enq(ram.peek()); fifo.deq(); ram.deq();
endrule
```

```
rule enter (True);
   IP ip = inQ.first(); ram.req(ip[31:16]);
   fifo.enq(ip[15:0]); inQ.deq();
endrule
```


Sometimes rule splitting is not possible

```
rule recirculate (True);
   TableEntry p <- ram.resp();
   match {.rip, .tok} = fifo.first();
   if (isLeaf(p)) cbuf.put(tok, p);
   else begin
       fifo.enq(tuple2(rip << 8, tok));
       ram.req(p+signExtend(rip[15:8]));
     end
   fifo.deq();
endrule
```

Packaging a module:
Turning a rule into a method

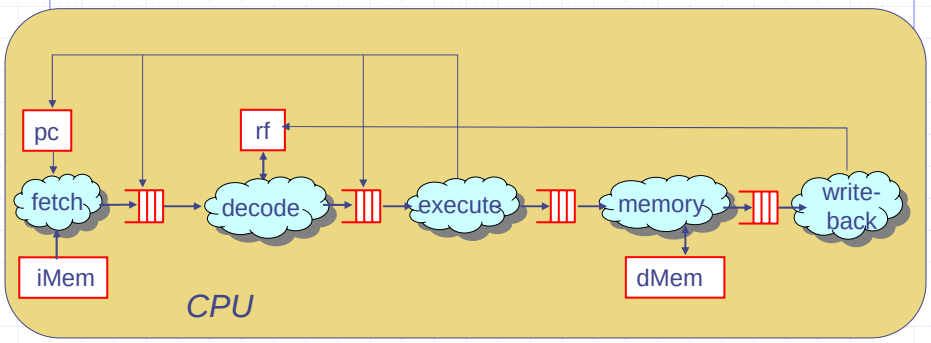



```
rule enter (True);
   Token t <- cbuf.getToken();
   IP ip = inQ.first();
   ram.req(ip[31:16]);
   fifo.enq(tuple2(ip[15:0], t)); inQ.deq();
endrule
```


Processor with a two-stage pipeline

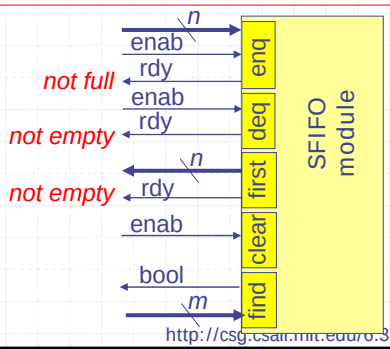
5-minute break to stretch you legs

Processor Pipelines and FIFOs



SFIFO (glue between stages)

```
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
```

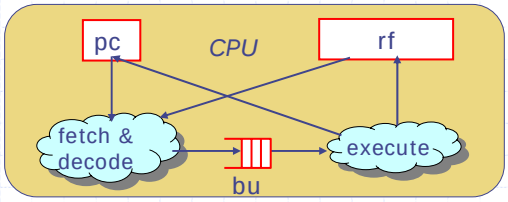


more on searchable FIFOs later

n = # of bits needed to represent the values of type "t"

m = # of bits needed to represent the values of type "tr"

Two-Stage Pipeline




```
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
```

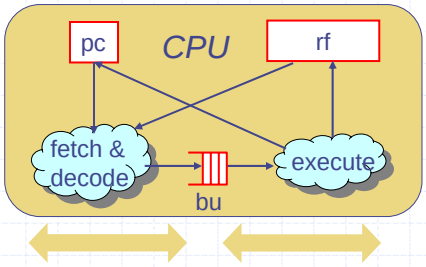

Instructions & Templates

```
typedef union tagged {
  struct {RName dst; RName src1; RName src2} Add;
  struct {RName cond; RName addr}            Bz;
  struct {RName dst; RName addr}             Load;
  struct {RName value; RName addr}           Store;
} Instr deriving(Bits, Eq);

typedef union tagged
{ struct {RName dst; Value op1; Value op2} EAdd;
  struct {Value cond; Iaddress tAddr}      EBz;
  struct {RName dst; Daddress addr}        ELoad;
  struct {Value data; Daddress addr}       EStore;
} InstTemplate deriving(Eq, Bits);

typedef Bit#(32) Iaddress;
typedef Bit#(32) Daddress;
typedef Bit#(32) Value;
```

Rules for Add



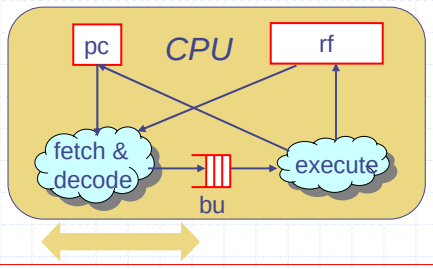

```
rule decodeAdd(instr matches Add{dst:.rd,src1:.ra,src2:.rb})
  bu.enq (EAdd{dst:rd,op1:rf[ra],op2:rf[rb]});
  pc <= predIa;
endrule
```

implicit check:
bu notfull

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

implicit check:
bu notempty

Fetch & Decode Rule: *Reexamined*



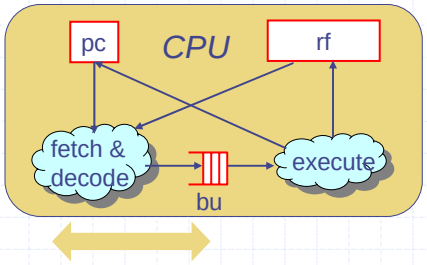

```
rule decodeAdd (instr matches Add{dst:.rd,src1:.ra,src2:.rb})

  bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});

  pc <= predIa;
endrule
```

stall !

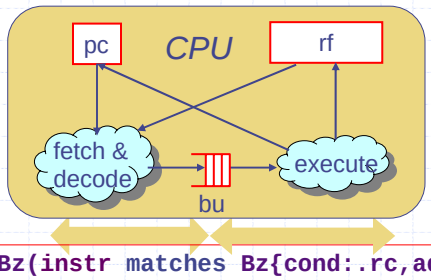
Fetch & Decode Rule: *corrected*




```
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 decodeBz(instr matches Bz{cond:.rc,addr:.addr}) &&&
      !bu.find(rc) &&& !bu.find(addr));
     bu.enq (EBz{cond:rf[rc],addr:rf[addr]});
     pc <= predIa;
  endrule
```

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

The Stall Signal

```
Bool stall  =
  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));
   tagged Load  {dst:.rd,addr:.addr}:
      return (bu.find(addr));
   tagged Store  {value:.v,addr:.addr}:
      return (bu.find(v)) || bu.find(addr));
  endcase;
```

Need to extend the fifo interface with the "find" method where "find" searches the fifo using the **findf** function

Parameterization: The Stall Function

```
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    {cond:.rc,addr:.addr}:
      return (bu.find(rc) || bu.find(addr));
   tagged Load  {dst:.rd,addr:.addr}:
      return (bu.find(addr));
   tagged Store  {value:.v,addr:.addr}:
      return (bu.find(v)) || bu.find(addr));
  endcase
endfunction
```

We need to include the following call in the mkCPU module

```
Bool stall = stallfunc(instr, bu);
```

The findf function

```
function Bool findf (RName r, InstrTemplate it);
  case (it) matches
    tagged EAdd{dst:.rd,op1:.ra,op2:.rb}:
                                  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
```

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

mkSFifo can be parameterized by the search function!

Fetch & Decode Rule

```
rule fetch_and_decode(!stall);
  case (instr) matches
     tagged Add {dst:.rd,src1:.ra,src2:.rb}:
       bu.enq(EAdd{dst:rd,op1:rf[ra],op2:rf[rb]});
     tagged Bz  {cond:.rc,addr:.addr}:
       bu.enq(EBz{cond:rf[rc],addr:rf[addr]});
     tagged Load {dst:.rd,addr:.addr}:
       bu.enq(ELoad{dst:rd,addr:rf[addr]});
     tagged Store{value:.v,addr:.addr}:
       bu.enq(EStore{value:rf[v],addr:rf[addr]});
  endcase
  pc<= predIa;
endrule
```

Fetch & Decode Rule

another style

```
InstrTemplate  newIt =
    case (instr) matches
        tagged Add {dst:.rd,src1:.ra,src2:.rb}:
             return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
        tagged Bz  {cond:.rc,addr:.addr}:
             return EBz{cond:rf[rc],addr:rf[addr]};
        tagged Load {dst:.rd,addr:.addr}:
             return ELoad{dst:rd,addr:rf[addr]};
        tagged Store{value:.v,addr:.addr}:
             return EStore{value:rf[v],addr:rf[addr]};
    endcase;
```

```
rule fetch_and_decode (!stall);
        bu.enq(newIt);
        pc <= predIa;
endrule
```

Conceptually cleaner;
hides unnecessary details

Execute Rule

```
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
```