Bluespec-6: Modeling Processors

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Instruction set

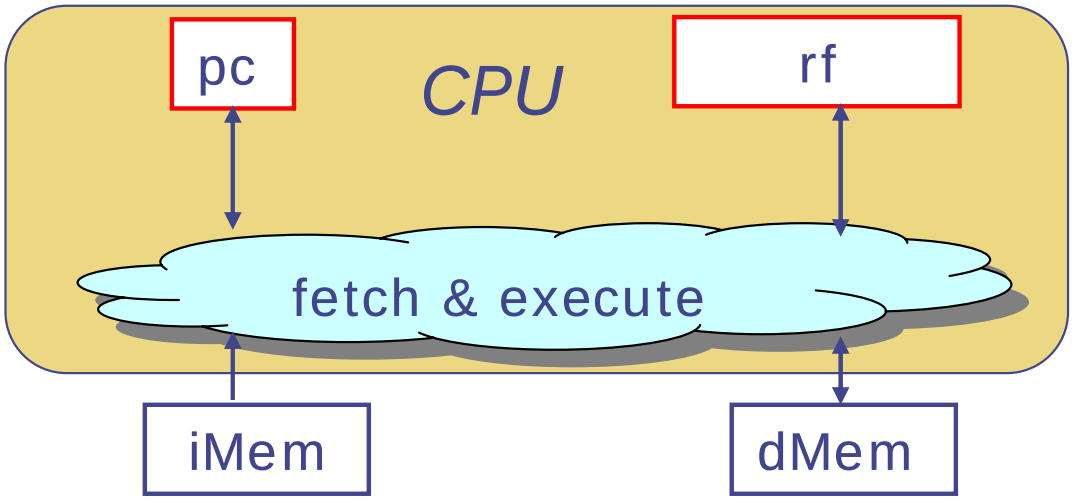
```
typedef enum {R0;R1;R2;…;R31} RName;

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

An instruction set can be implemented using many different microarchitectures

Processors

- Non-pipelined processor ⇐
- Two-stage pipeline
- Performance issues

Non-pipelined Processor




```
module mkCPU#(Mem iMem, Mem dMem)();
        Reg#(Iaddress) pc <- mkReg(0);
        RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
        Instr     instr = iMem.read(pc);
        Iaddress predIa = pc + 1;
        rule fetch_Execute ...
endmodule
```

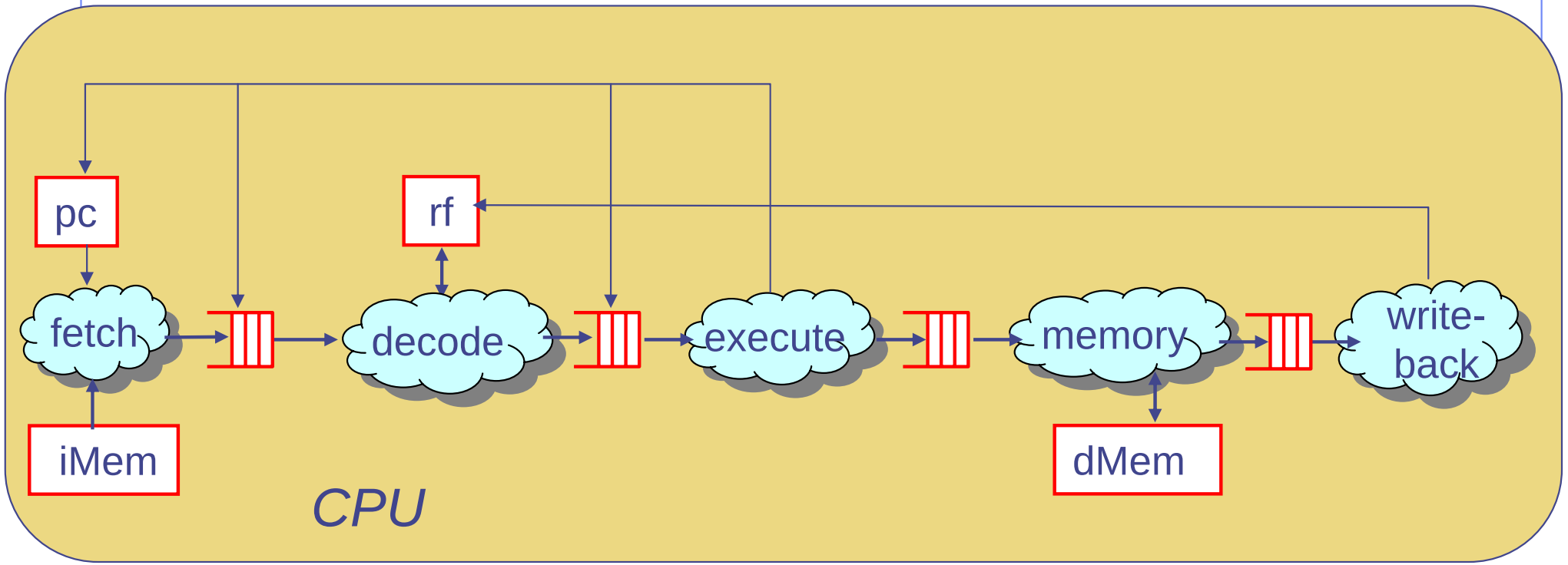
Non-pipelined processor rule

```
rule fetch_Execute (True);
   case (instr) matches
     tagged Add {dst:.rd,src1:.ra,src2:.rb}: begin
                rf.upd(rd, rf[ra]+rf[rb]);
                pc <= predIa
            end
     tagged Bz {cond:.rc,addr:.ra}: begin
                pc <= (rf[rc]==0) ?  rf[ra] :  predIa;
            end
     tagged Load {dest:.rd,addr:.ra}: begin
                rf.upd(rd, dMem.read(rf[ra]));
                pc <= predIa;
            end
     tagged Store {value:.rv,addr:.ra}: begin
                dMem.write(rf[ra],rf[rv]);
                pc <= predIa;
            end
   endcase
endrule
```

my syntax
rf[r] ≡ rf.sub(r)

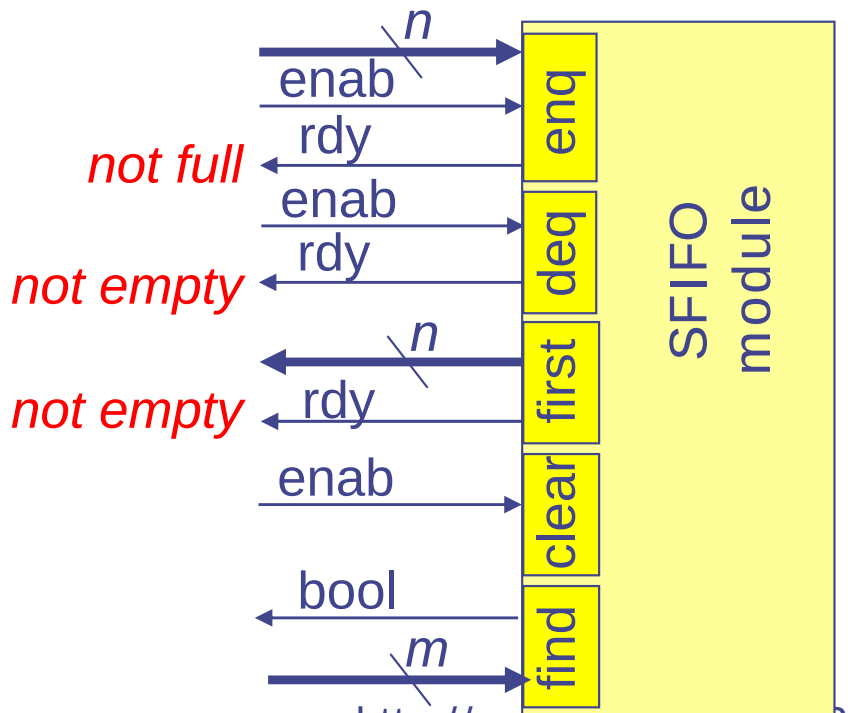
Assume “magic memory”, i.e. responds to a read request in the same cycle and a write updates the memory at the end of the cycle

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
```

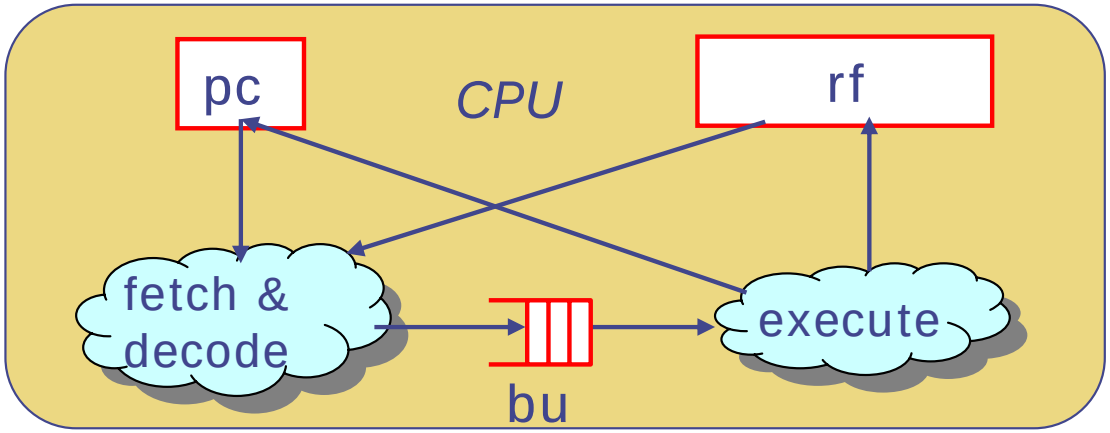


more on searchable FIFOs later

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

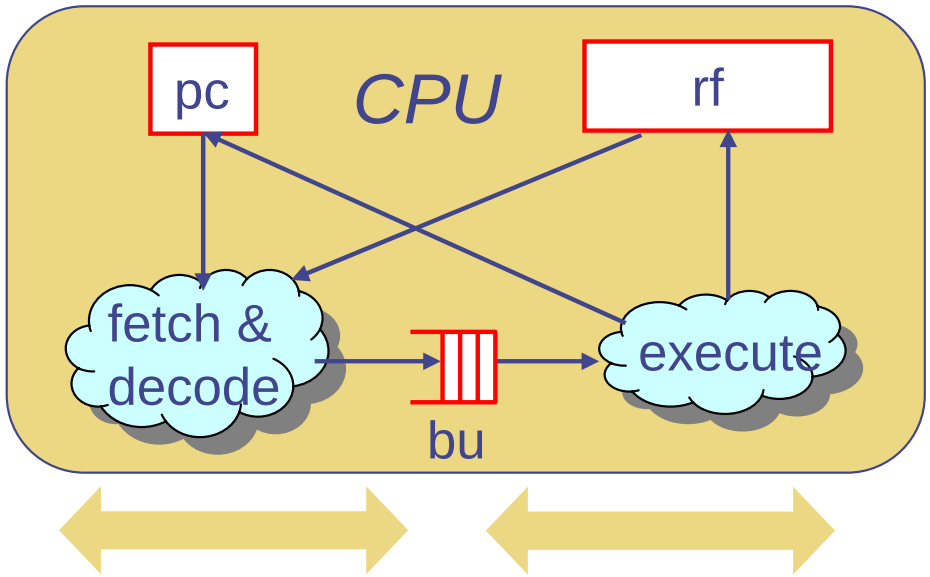



```
module mkCPU#(Mem iMem, Mem dMem)(Empty);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    SFIFO#(InstTemplate, RName) bu
                                    <- mkSFifo(findf);
    Instr     instr = iMem.read(pc);
    Iaddress predIa = pc + 1;
    InstTemplate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



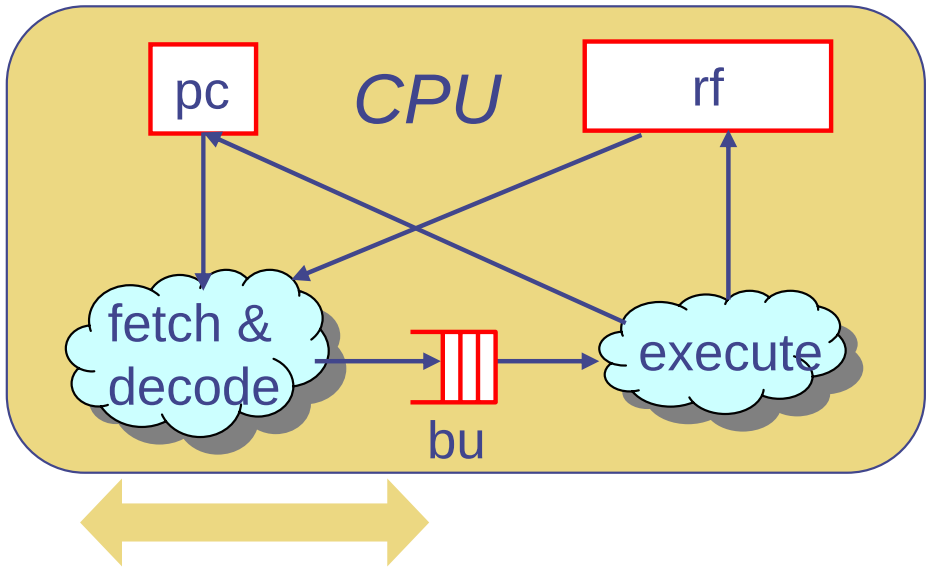

```
rule decodeAdd(instr matches Add{dst:.rd,src1:.ra,src2:.rb})
  bu.enq (EAdd{dst:rd,op1:rf[ra],op2:rf[rb]});
  pc <= predIa;
endrule
```

implicit check:

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

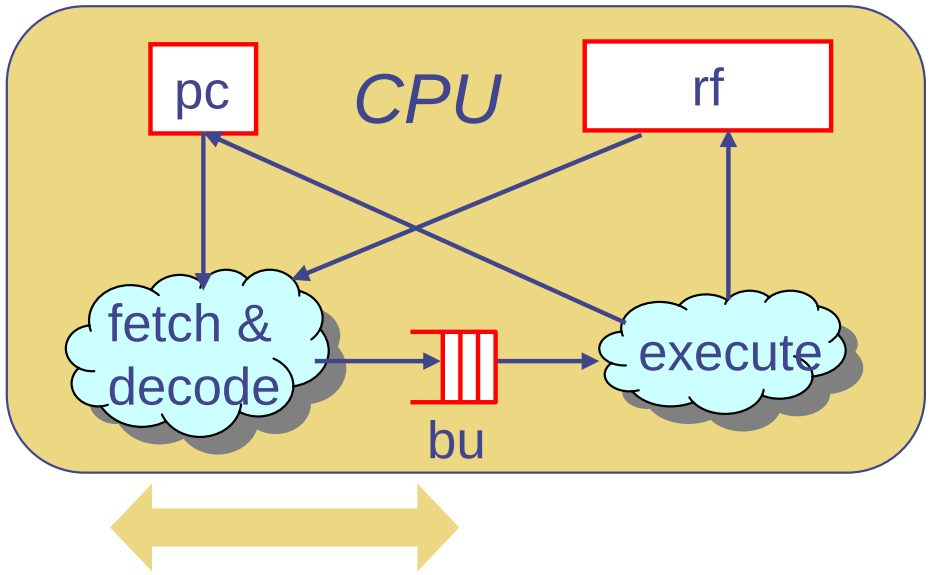
implicit check:

Fetch & Decode Rule: *Reexamined*



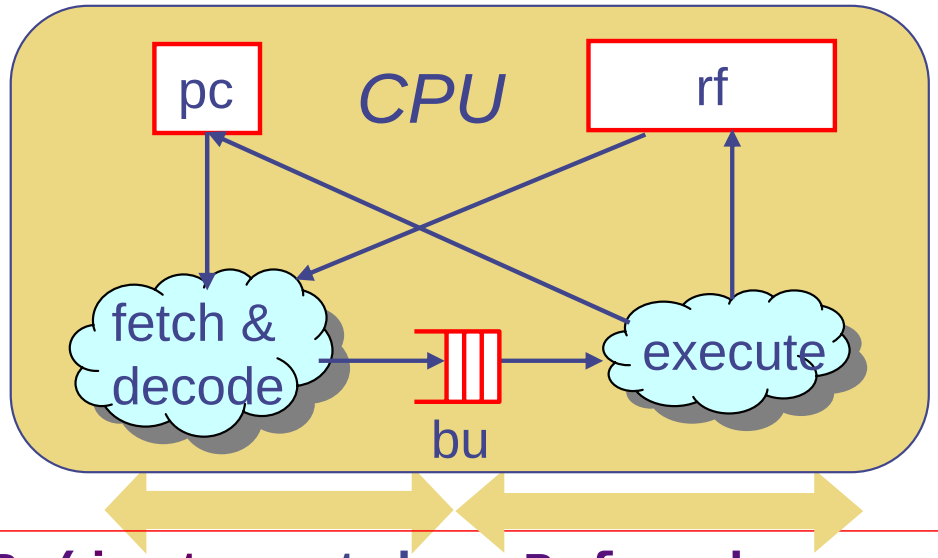

```
rule decodeAdd (instr matches Add{dst:.rd,src1:.ra,src2:.rb})

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

  pc <= predIa;
endrule
```

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

```
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);
```

no extra gates!

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!

no extra gates!

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
```

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
```

Searchable FIFOs and Concurrency Issues

One-Element Searchable FIFO

parameterized by findf

```
module mkSFIFO1#(function Bool findf(tr rd, t x))
                                (SFIFO#(type t, type tr));
  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
  method Bool find(tr rd);
    return full && findf(rd, data);
  endmethod
endmodule
```

Conflict Matrix

	enq2	first2	deq2	clear2	find2
enq1	C	>	C	<	>
first1	<	CF	<	<	CF
deq1	C	>	C	<	>
clear1	>	>	>	C	>
find1	<	CF	<	<	CF

Two-Element Searchable FIFO

```
module mkSFIFO2#(function Bool findf(tr rd, t x))
                (SFIFO#(type t, type tr));
  Reg#(t) data0 <-mkRegU; Reg#(Bool) full0 <- mkReg(False);
  Reg#(t) data1 <-mkRegU; Reg#(Bool) full1 <- mkReg(False);
  method Action enq(t x) if (!full1);
    if (!full0) begin data0 <= x; full0 <= True; end
          else begin data1 <= x; full1 <= True; end
 endmethod // Shift register implementation;
  method Action deq() if (full0);
    full0 <= full1;  data0 <= data1; full1 <= False;
  endmethod
  method t first() if (full0);
    return data0;
  endmethod
  method Action clear();
    full0 <= False; full1 <= False;
  endmethod
  method Bool find(tr rd);
     return ((full0 && findf(rd, data0) ||
              (full1 && findf(rd, data1));
  endmethod endmodule
```

Conflict Matrix is the same as one-element SFIFO

Concurrency requirements

```
rule fetch_and_decode (!stall);
        bu.enq(tuple2(pc, newIt));
        pc <= predIa;
endrule
```

The two rules must fire together whenever possible, *and* execute < fetch_and_decode

```
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
```

Intra-Rule Requirements

- fetch_and_decode rule
  - two read methods (ports) on the rf register file (rf[ra],rf[rb])
  - two find methods on the bu SFIFO (bu.find(ra), bu.find(rb))
  - read methods must come before action methods, e.g. bu.find < bu.enq
- execute rule
  - read methods must come before action methods, e.g., bu.first < bu.deq

Inter-rule concurrency requirements: case analysis

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}:
     begin rf.upd(rd, va+vb); bu.deq(); end
    ...
  endcase endrule
```

```
rule fetch_and_decode (!stall);
        bu.enq(tuple2(pc, newIt));
        pc <= predIa;
endrule
InstrTemplate  newIt =
  case (instr) matches
    tagged Add {dst:.rd,src1:.ra,src2:.rb}:
     return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
     ...
    endcase;
```

Numbering the methods

worksheet

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

execute < fetch_and_decode

➔ $rf.upd^0 < rf.sub^1$

$bu.first^0 < \{bu.deq^0, bu.clear^0\} < bu.find^1 < bu.enq^1$

```
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
```