

Bluespec-6: Modularity and Performance

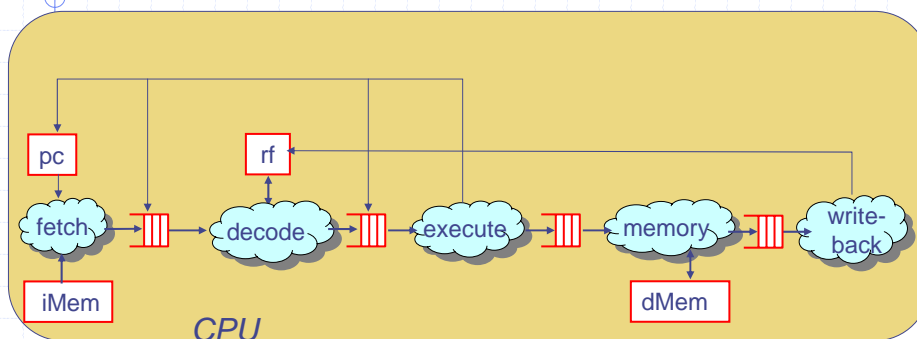
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-1

Processor Pipelines and FIFOs



How should designs be modularized?

Does modularization affect performance?

For performance it is necessary that all pipeline stages be able to fire concurrently

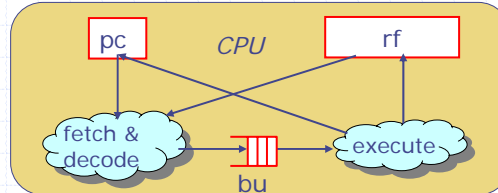
Does the FIFO implementation permit this?

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-2

Two-Stage Pipeline



```

module mkCPU#(Mem iMem, Mem dMem)(Empty);
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  SFIFO#(Tuple2#(Iaddress, InstTemplate)) bu
    <- mkSFifo(findf);

  Address i32 = iMem.get(pc);
  Instr instr = unpack(i32[16:0]);
  Address predIa = pc + 1;
  match{.ipc, .it} = bu.first;
  rule fetch_decode ...

```

endmodule

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-3

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;

```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-4

Fetch & Decode Rule

```
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(tuple2(pc, newIt));
  pc <= predIa;
endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-5

Execute Rule

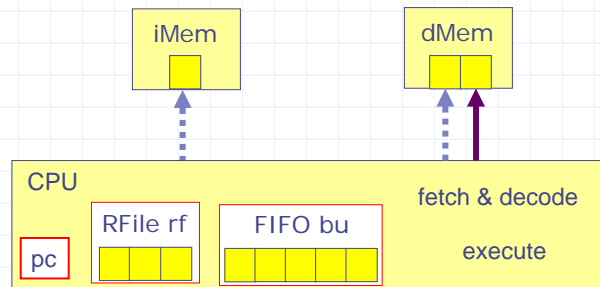
```
rule execute_rule (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
      rf.upd(rd, va+vb); bu.deq();
    end
    tagged EBz {cond:.cv,addr:.av}:
      if (cv == 0) then begin
        pc <= av; bu.clear(); end
      else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
      rf.upd(rd, dMem.get(av)); bu.deq();
    end
    tagged EStore{value:.vv,addr:.av}: begin
      dMem.put(av, vv); bu.deq();
    end
  endcase
endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-6

CPU as one module



Method calls embody both data and control (i.e., protocol)

.....> Read method call

————> Action method call

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-7

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

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-8

Parameterization: The Stall Function

```
function Bool stallfunc (InstTemplate instr,
                        SFIFO#(Tuple2#(Iaddress, InstTemplate)) 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!
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-9

The findf function

```
function Bool findf (RName r,
                    Tuple2#(Iaddress, InstrTemplate) tup);
case (snd(tup)) 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#(Tuple2(Iaddress, InstrTemplate)) bu
    <- mkSFifo(findf)
```

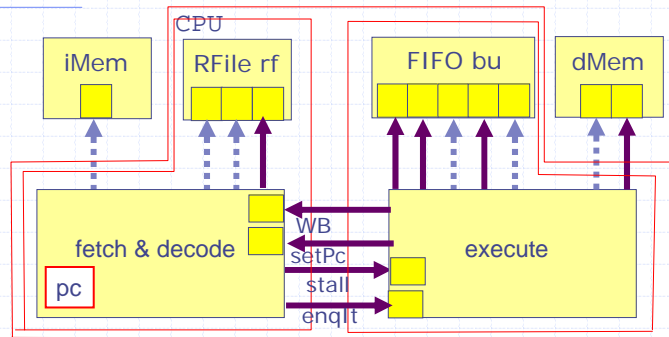
mkSFifo can be parameterized by the search function!
No extra gates? ... more on this later

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-10

A Modular organization: recursive modules *(not allowed but ...)*



Modules call each other

- bu part of Execute
- rf and pc part of Fetch&Decode
- fetch delivers decoded instructions to Execute

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-11

Recursive modular organization

```

module mkCPU#(Mem iMem, Mem dMem)(Empty);
  Execute execute <- mkExecute(dMem, fetch);
  Fetch fetch <- mkFetch(iMem, execute);
endmodule

interface Fetch;
  method Action setPC (Iaddress cpc);
  method Action writeback (RName dst, Value v);
endinterface

interface Execute;
  method Action enqIt(Tuple2#(Iaddress,InstrTemplate) x);
  method Bool stall(Instr instr)
endinterface
    
```

recursive calls

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-12

Fetch & Decode Module

```
module mkFetch(Mem dMem, Execute execute)(Fetch);
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  Address i32=iMem.get(pc);Instr instr=unpack(i32[16:0]);
  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(tuple2(pc, newIt)); pc <= pc+1;
  endrule
  method...
endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-13

Fetch & Decode Methods

```
method Action setPC(Iaddress ia);
  pc <= ia;
endmethod

method Action writeback(RName dst, Value v);
  rf.upd(dst,v);
endmethod
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-14

Execute Module

```
module mkExecute#(Mem dMem,Fetch fetch) (Empty);
  SFIFO#(Tuple2(Iaddress, InstTemplate)) bu
    <- mkSFifo(findf);
  match {.ipc, .it} = bu.first;

  rule ...
  method Bool stall (Instr instr);
    return (stallfunc(instr,bu));
  endmethod

  method Action enqIt(Tuple2#(Iaddress,InstrTemplate) x)
    bu.enq(x);
  endmethod

endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-15

Execute Module Rule

```
rule execute_rule (True);
  case (it) matches
    tagged EAdd{dst:.rd,op1:.va,op2:.vb}: begin
      fetch.writeback(rd, va + vb); bu.deq();
    end
    tagged EBz {cond:.cv,addr:.av}:
      if (cv == 0) then begin
        fetch.setPC(av); bu.clear(); end
      else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
      fetch.writeback(rd, dMem.get(av)); bu.deq();
    end
    tagged EStore{value:.vv,addr:.av}: begin
      dMem.put(av,vv); bu.deq();
    end
  end
  emdcase
endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-16

Is this a good modular organization?

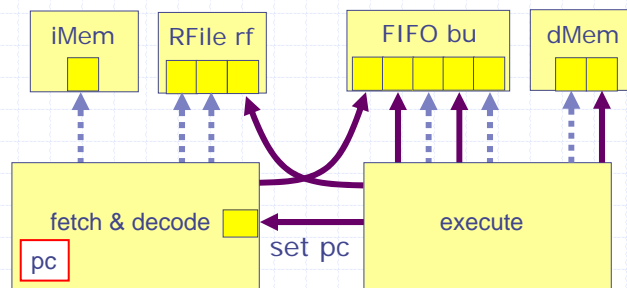
- ◆ Separately compilable?
- ◆ Separately refinable?
- ◆ Good for verification?
- ◆ Physical properties:
 - Few connecting wires?
 - ...
- ◆ ...

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-17

Modular organizations



- make fetch&decode separately compilable (shown)
- make execute separately compilable

.....> Read method call
————> Action method call

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-18

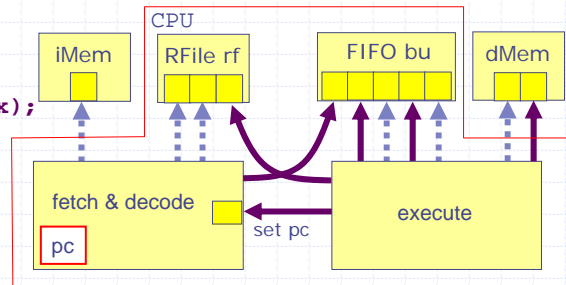
A nonrecursive modular organization

```

module mkCPU#(Mem iMem, Mem dMem)(Empty);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  SFIFO#(Tuple2(Iaddress, InstTemplate)) bu
    <- mkSFifo(findf);
  Fetch fetch <- mkFetch(iMem, bu, rf); ← no recursion
  Empty exec <- mkExecute(dMem, bu, rf, fetch);
endmodule

interface Fetch;
  method Action
    setPC(Iaddress x);
endinterface

```



March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-19

Fetch & Decode Module

```

module mkFetch(Mem dMem, SFIFO#(Tuple2(Iaddress,
  InstTemplate)) bu, RegFile#(RName, Bit#(32)) rf)(Fetch);
  Reg#(Iaddress) pc <- mkReg(0);
  Iaddress i32=iMem.get(pc); Instr instr=unpack(i32[16:0]);
  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(tuple2(pc, newIt)); pc <= pc+1;
  endrule
  method Action setPC(Iaddress ia);
    pc <= ia;
  endmethod
endmodule

```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-20

Execute Module

```

module mkExecute#(Mem dMem,
                 SFIFO#(Tuple2(Iaddress,InstTemplate) bu,
                 RegFile#(RName, Bit#(32)) rf, Fetch fetch)(Empty);
match {.ipc, .it} = bu.first;
rule execute_rule (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,addr:.av}:
    if (cv == 0) then begin
        fetch.setPC(av); bu.clear(); end
    else bu.deq();
tagged ELoad{dst:.rd,addr:.av}: begin
    rf.upd(rd, dMem.get(av)); bu.deq();
end
tagged EStore{value:.vv,addr:.av}: begin
    dMem.put(av,vv); bu.deq();
end
endcase endrule endmodule

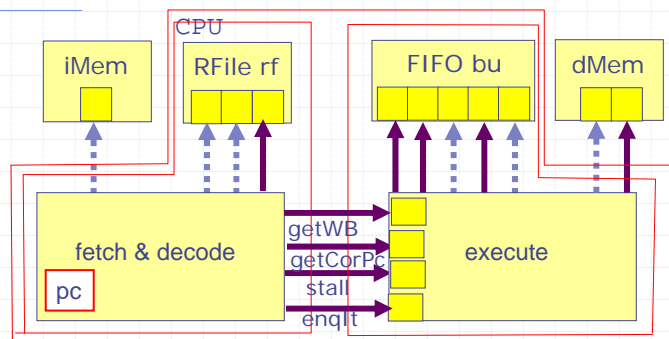
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-21

Another Modular organization



Make Execute separately compilable

- make bu part of Execute
- make rf part of Fetch&Decode
- let fetch deliver decoded instructions to Execute

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-22

Another modular organization

```
module mkCPU#(Mem iMem, Mem dMem)(Empty);
  Execute execute <- mkExecute(dMem); ← no recursion
  Empty fetch <- mkFetch(iMem, execute);
endmodule

interface Execute;
  method Action enqIt(Tuple2#(Iaddress,InstrTemplate) x);
  method Bool stall(Instr instr);
  method ActionValue#(Iaddress) getCorPC();
  method ActionValue#(Tuple2#(RName, Value)) getWriteback();
endinterface
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-23

Fetch & Decode Module

```
module mkFetch(Mem dMem, Execute execute) (Empty);
  Reg#(Iaddress) pc <- mkReg(0);
  RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
  Iaddress i32=iMem.get(pc);Instr instr=unpack(i32[16:0]);
  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 ...
endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-24

Fetch & Decode Module Rules

```
rule fetch_and_decode (!execute.stall(instr));
    execute.enqIt(tuple2(pc, newIt));
    pc <= predIa;
endrule

rule setPC(True);
    pc <= execute.getCorPC();
endrule

rule writeRF(True);
    match {.rd, .v} = execute.getWriteback();
    rf.upd(rd, v);
endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-25

Execute Module

```
module mkExecute#(Mem dMem) (Empty);
    SFIFO#(Tuple2(Iaddress, InstTemplate)) bu
        <- mkSFifo(findf);
    match {.ipc, .it} = bu.first;

    rule ...
    method Bool stall (Instr instr);
        return (stallfunc(instr, bu));
    endmethod

    method Action enqIt(Tuple2#(Iaddress, InstTemplate) x)
        bu.enq(x);
    endmethod

    method getCorPC ...
    method getWriteback ...
endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-26

getCorPC Method

```
method ActionValue#(Iaddress) getCorPC (it matches
      EBz {cond:.cv, addr:.av} &&& (cv == 0));
  return (av);
  bu.clear();
endmethod
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-27

getWriteback Method

```
function Bool writeRF(InstrTemple it);
  case (it) matches
    tagged EAdd: return (True);
    tagged ELoad: return (True);
    default: return (False);
  endcase
endfunction

method ActionValue#(Tuple2#(RName, Value)) getWriteback()
      if (writeRF(it));
  bu.deq();
  case (it) matches
    tagged EAdd{dst:.rd,op1:.va,op2:.vb}:
      return(tuple2(rd, va + vb));
    tagged ELoad{.rd,.av}:
      return (tuple2(rd, dMem.get(av)));
  endcase
endmethod
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-28

Execute Module Rules

```
rule execute_rule (True);
  case (it) matches
    tagged EBz{cond:.cv,addr:.av}:
      if (cv != 0) bu.deq();
      tagged EStore{value:.vv, addr:.av}:begin
        dMem.put(av,vv); bu.deq();
      end
    endcase
  endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-29

Summary

- ◆ Recursive modular organizations are natural but
 - not supported!
 - theoretical complications
 - ◆ e.g., Can you write a self-inhibiting action ?
- ◆ Non recursive structures may be difficult to express at times (try JAL)
- ◆ Do different modular structures generate the “same hardware”?
 - probably but we need to investigate further

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-30

Implementing FIFOs

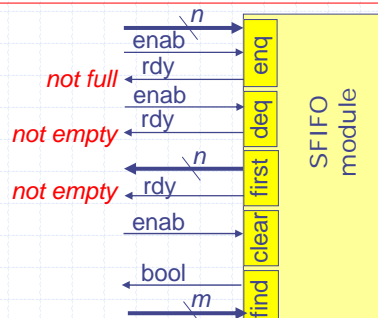
March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-31

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
```



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

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

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-32

One Element FIFO

```
module mkSFIFO1#(function Bool findf(t ele, tr rd))
  (SFIFO#(type t, type tr));
  Reg#(t) data <- mkRegU();
  Reg#(Bool) used <- mkReg(False);
  method Action enq(t) if (!used);
    used <= True; data <= t;
  endmethod
  method Action deq() if (used);
    used <= False;
  endmethod
  method t first() if (used);
    return (data);
  endmethod
  method Action clear();
    used <= False;
  endmethod
  method Bool find(tr);
    return ((used)? findf(data, val): False);
  endmethod
endmodule
```

parameterization by findf
is straightforward

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-33

Two-Element FIFO

```
module mkSFIFO2#(function Bool findf(t ele, tr rd))
  (SFIFO#(type t, type tr));
  Reg#(t) data0 <- mkRegU(); Reg#(t) data1 <- mkRegU();
  Reg#(Bool) used0 <- mkReg(F); Reg#(Bool) used1 <-mkReg(F);
  method Action enq(t) if (!(used0 && used1));
    data1 <= t; used1 <= True;
    if (used1) then begin data0 <= data1; used0 <= True; end
  endmethod
  method Action deq() if (used0 || used1);
    if (used0) used0 <= False; else used1 <= False;endmethod
  method t first() if (used0 || used1);
    return ((used0)?data0:data1); endmethod
  method Action clear();
    used0 <= False; used1 <= False; endmethod
  method Bool find(tr);
    return ((used0 && findf(data0, val) ||
            (used1 && findf(data1, val)));
  endmethod
endmodule
```

Shift register
implementation

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-34

Concurrency & other Issue

- ◆ The design is parameterized by the find function
- ◆ It is possible to write a FIFO module that is parameterized by the number of elements but it requires a lot of BSV expertise
- ◆ Concurrency: Can enq and deq be done simultaneously? CF?

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-35

Concurrency: One Element FIFO

```
module mkSFIFO1#(function Bool findf(t ele, tr rd)
                 (SFIFO#(type t, type tr)));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) used <- mkReg(False);
  method Action enq(t) if (!used);
    used <= True;    data <= t;
  endmethod
  method Action deq() if (used);
    used <= False;
  endmethod
  method t first() if (used);
    return (data);
  endmethod
  method Action clear();
    used <= False;
  endmethod
  method Bool find(tr);
    return ((used)? findf(data, val): False);
  endmethod
endmodule
```

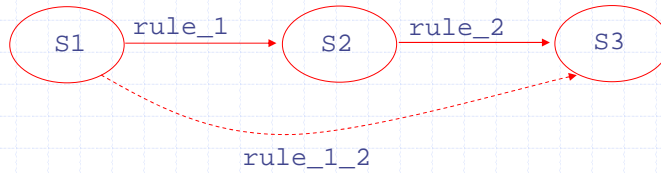
Both enq and deq read and write the used bit!

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-36

Rule composition



```
rule rule_1(p1(r)); r <= f1(r);endrule
rule rule_2(p2(r)); r <= f2(r);endrule
rule rule_1_2(p1(r) && p2(r'); r<= f2(r'); endrule
where r' = f1(r);
```

Guarded atomic actions guarantee that rule_1_2 which takes s1 to s3 is correct

Such composed rules are mechanically derivable

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-37

Composing rules:

Fetch & Decode + Execute Rule

```
rule executeAdd(it is EAdd{rd,va,vb})
  rf.upd(rd, va + vb);
  bu.deq();
endrule
```

```
rule decodeAdd (instr is Add{rd,ra,rb}
  && !bu.find(ra) &&& !bu.find(rb))
  bu.enq (tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
  pc <= predIa;
endrule
```

```
rule exeFetchAdd(it is EAdd{rd,va,vb} &&
  instr' is Add{rd,ra,rb} &&&
  !bu'.find(ra) &&& !bu'.find(rb))
  rf.upd(rd, va + vb);
  bu.deq();
  bu'.enq(tuple2(pc', EAdd{rd, rf[ra], rf[rb]}));
  pc' <= predIa';
endrule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-38

Composing rules: Effect on interfaces

```
rule exeFetchAdd(it is EAdd{rd,va,vb} &&
                instr is Add{rd,ra,rb} &&&
                !bu.deqfind(ra) &&& !bu.deqfind(rb))
  rf.upd(rd, va + vb);
  bu.deqenq(tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
  pc <= predIa;
endrule
```

instr' is the same as instr
pc' is the same as pc
predIa' is the same as predIa
bu' is the state of the fifo after the deq

```
interface SFIFO#(type t, type tr);
  methods enq(t); deq(); first(); clear(); find(tr);

  method Action deqenq(t) // dequeue and then enqueue
  method Bool deqfind(tr); // find after deq occurs
endinterface
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-39

Composing methods

One-element FIFO

```
module mkSFIFO1#(function Bool findf(t ele, tr rd)
                (SFIFO#(type t, type tr)));
```

...

```
method Action enq(t) if (!used);
  used <= True; data <= t;
endmethod
method Action deq() if (used);
  used <= False;
endmethod
```

```
method Action deqenq() if (used);
  data <= t;
endmethod
method Action enqdeq() if (!used);
  noAction;
endmethod
```

These methods are derivable from enq and deq.

```
methods first() clear() find(tr) ...
endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-40

Composing methods

Two-element FIFO

```
module mkSFIFO1#(function Bool findf(t ele, tr rd)
  (SFIFO#(type t, type tr));

  ...
  method Action enq(t) if (!(used0 && used1));
    data1 <= t; used1 <= True;
    if (used1) then begin data0 <= data1; used0 <= True; end
  endmethod
  method Action deq() if (used0 || used1);
    if (used0) used0 <= False; else used1 <= False; endmethod
  method Action deqenq if (used0 || used1);
    data1 <= t; used1 <= True;
    if (used1) then begin data0 <= data1; used0 <= True; end
  endmethod
  method Action enqdeq if (!(used0 && used1));
    data1 <= t; used1 <= (used0 || used1);
    used0 <= False;
  endmethod
  methods first() clear() find(tr) ...
endmodule
```

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-41

Bypassing involves similar
issues

March 11, 2005

<http://csg.csail.mit.edu/6.884/>

L13-42