Bluespec-6
Simple Pipelined Processor

Arvind
Laboratory for Computer Science
M.I.T.

January 15, 2003

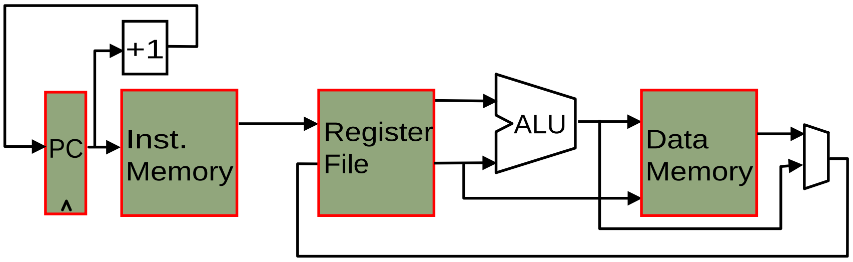
http://www.csg.lcs.mit.edu/IAPBlue

Instruction set

```
data RName = R0 | R1 | R2 | … | R31

type Src        = RName
type Dest       = RName
type Cond       = RName
type Addr       = RName
type Val        = RName

data Instr =      Add     Dest Src Src
                | Bz      Cond Addr
                | Load    Dest Addr
                | Store   Val  Addr
```

An instruction set can be implemented using many different microarchitectures

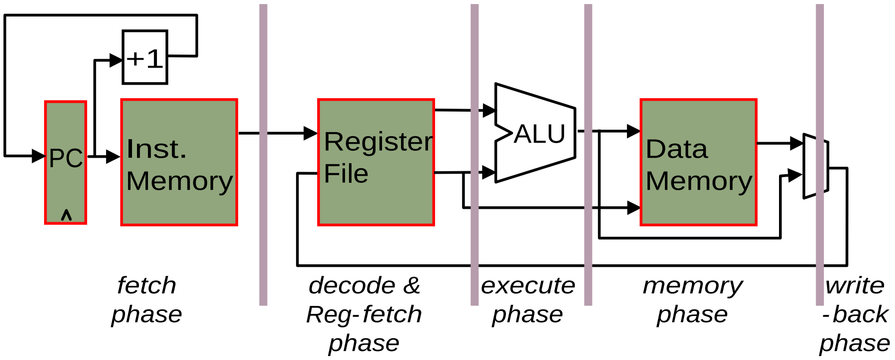
Non-pipelined Processor



Each instruction executes in one cycle!

slow ... slow ....slow ......

N-stage Pipelined Processor

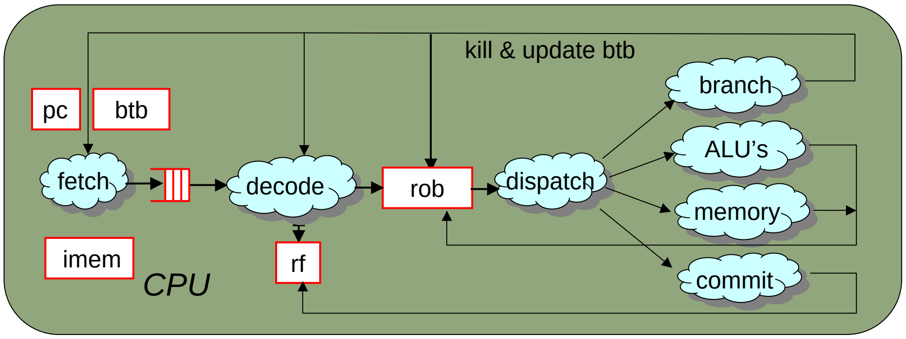


- Stages are separated by *FIFO buffers*
- In-order issue and completion

Do these processors produce the same behaviors as the non-pipelined processor?

Modern Microprocessors

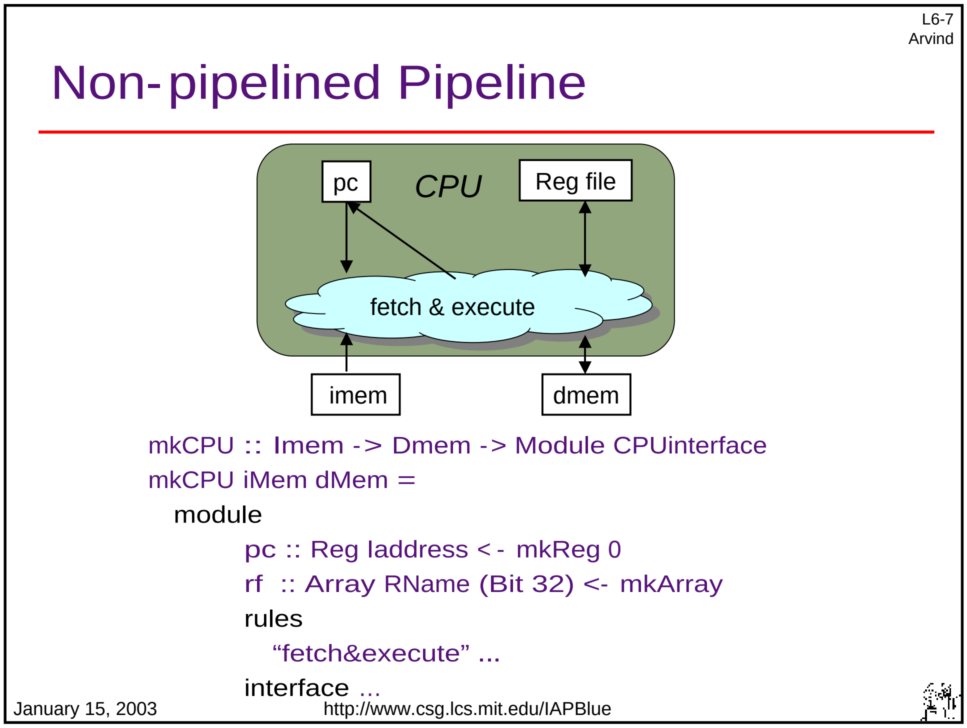
register renaming and speculative execution



Does a speculative processor produce the same behaviors as a non-pipelined processor?

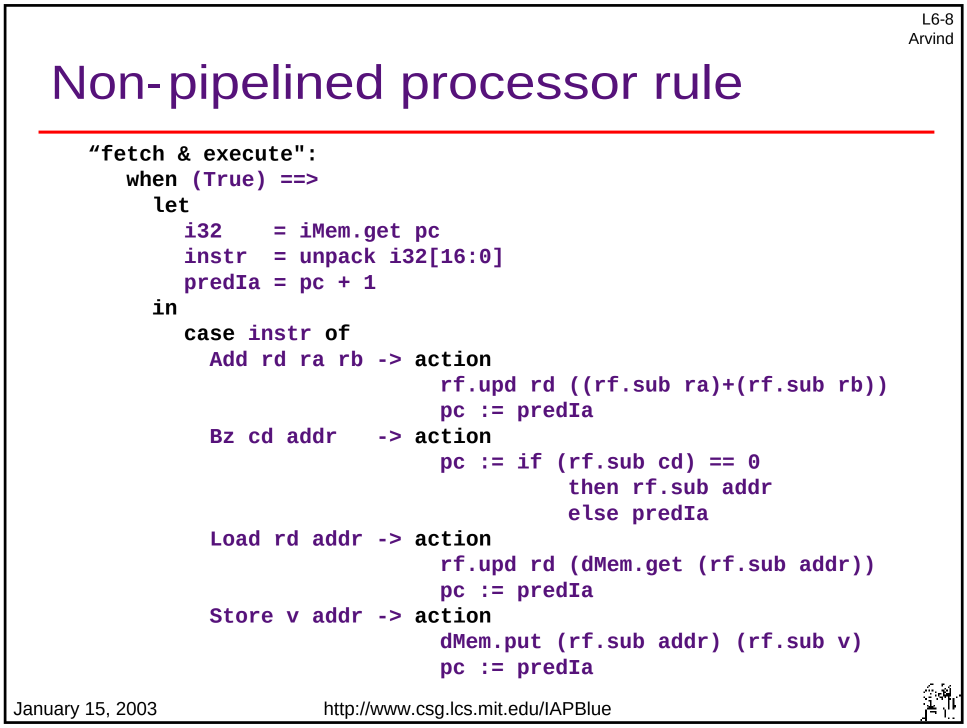
Outline

- Microacritectures√
- Unpipelined processor⇐
- Two-stage pipeline
- Bypass FIFO



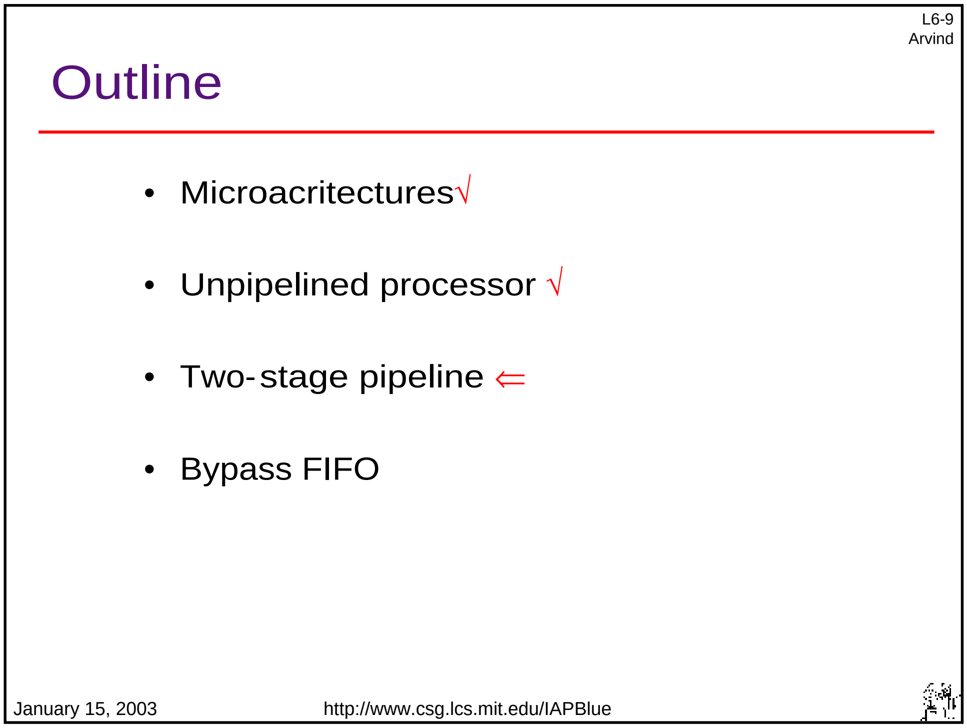
Non-pipelined Pipeline

```
mkCPU :: Imem -> Dmem -> Module CPUinterface
mkCPU iMem dMem =
  module
        pc :: Reg Iaddress <- mkReg 0
        rf  :: Array RName (Bit 32) <- mkArray
        rules
          “fetch&execute” ...
        interface ...
```



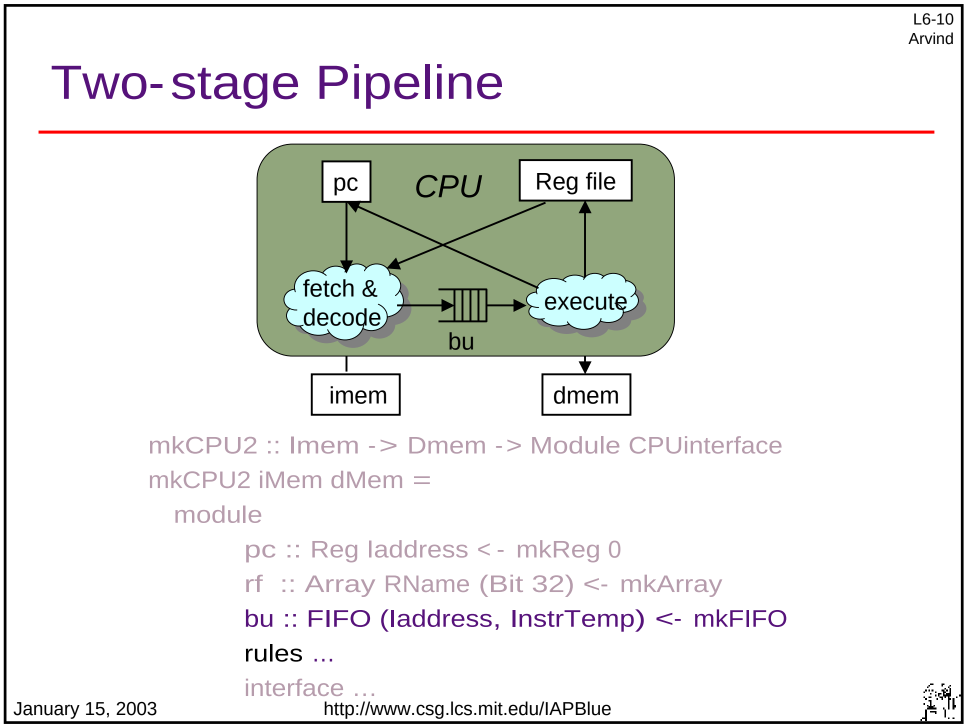
Non-pipelined processor rule

```
“fetch & execute":
   when (True) ==>
     let
       i32    = iMem.get pc
       instr  = unpack i32[16:0]
       predIa = pc + 1
     in
       case instr of
         Add rd ra rb -> action
                           rf.upd rd ((rf.sub ra)+(rf.sub rb))
                           pc := predIa
         Bz cd addr   -> action
                           pc := if (rf.sub cd) == 0
                                   then rf.sub addr
                                   else predIa
         Load rd addr -> action
                           rf.upd rd (dMem.get (rf.sub addr))
                           pc := predIa
         Store v addr -> action
                           dMem.put (rf.sub addr) (rf.sub v)
                           pc := predIa
```



Outline

- Microacritectures√
- Unpipelined processor √
- Two-stage pipeline ⇐
- Bypass FIFO



Two-stage Pipeline

```
mkCPU2 :: Imem -> Dmem -> Module CPUinterface
mkCPU2 iMem dMem =
  module
        pc :: Reg Iaddress <- mkReg 0
        rf  :: Array RName (Bit 32) <- mkArray
        bu :: FIFO (Iaddress, InstrTemp) <- mkFIFO
        rules ...
        interface ...
```

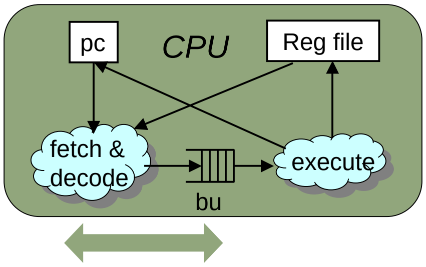
Instruction Template

```
data Instr =
                 Add     Dest Src Src
               | Bz      Cond Addr
               | Load    Dest Addr
               | Store   Val  Addr
        deriving (Bits)
```

decoded instruction with operands

```
data InstTemplate =
                EAdd     Dest  Value Value
              | EBz      Value Value
              | ELoad    Dest  Value
              | EStore   Value Value
        deriving (Bits)

type Value  = Bit 32
```

Fetch & Decode Rule: Add

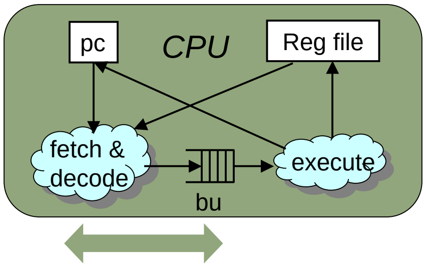


```
"fetch_and_decode_Add":

when (Add rd ra rb) <- instr ==>
   action
       bu.enq (pc,(EAdd rd (rf.sub ~~ra) (rf.sub rb)~~))
       pc:= pc+1
```

Wrong! Because instructions in bu may be modifying ra or rb

stall !

Fetch & Decode Rule: Add *corrected*



```
"fetch_and_decode_Add":

when (Add rd ra rb) <- instr, ((chk ra) || (chk rb)) ==>
   action
       bu.enq (pc,(EAdd rd (rf.sub ra) (rf.sub rb)))
       pc:= pc+1
```

The Stall Signal

```
stall  = case instr of
             Add rd ra rb -> (chk ra) || (chk rb)
             Bz  cd addr  -> (chk cd) || (chk addr)
             Load rd addr -> (chk addr)
             Store v addr -> (chk v)  || (chk addr)
```

```
chk r  = case (bu.find (findf r)) of
             Just _  -> True;
             Nothing -> False;
```

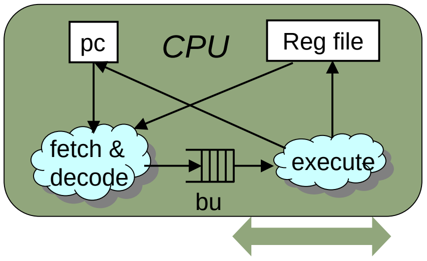
```
findf r it =
       let (pc, i) = it
       in   case i of
             EAdd rd _ _ -> (r == rd)
             EBz    _ _  -> False
             ELoad rd _  -> (r == rd)
             EStore _ _  -> False
```

Need to extend the fifo interface with the “find” method

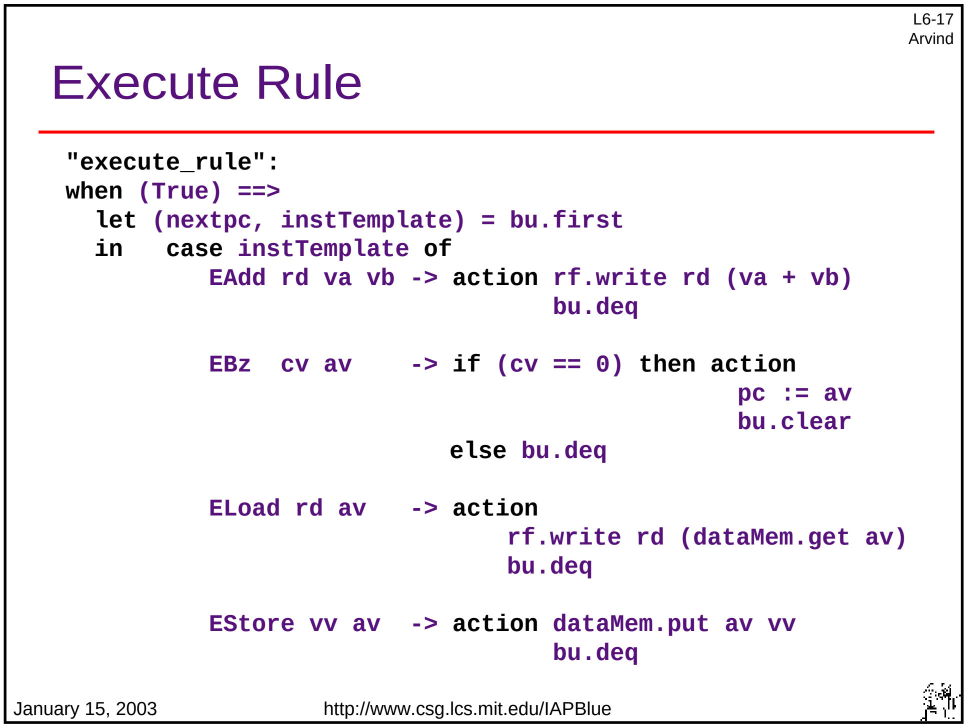
Fetch & Decode Rule

```
"fetch_and_decode_rule":
when (True) ==>
   if stall then noAction
   else
    action
       case instr of
         Add rd ra rb -> bu.enq (pc,
                           (EAd rd (rf.sub ra) (rf.sub rb)))
         Bz cd addr   -> bu.enq (pc,
                           (EBz (rf.sub cd) (rf.sub addr)))
         Load rd addr -> bu.enq (pc,
                           (ELd rd (rf.sub addr)))
         Store v addr -> bu.enq (pc,
                           (ESt (rf.sub v) (rf.sub addr)))
       pc:= pc+1
```

Execute Rule: Add

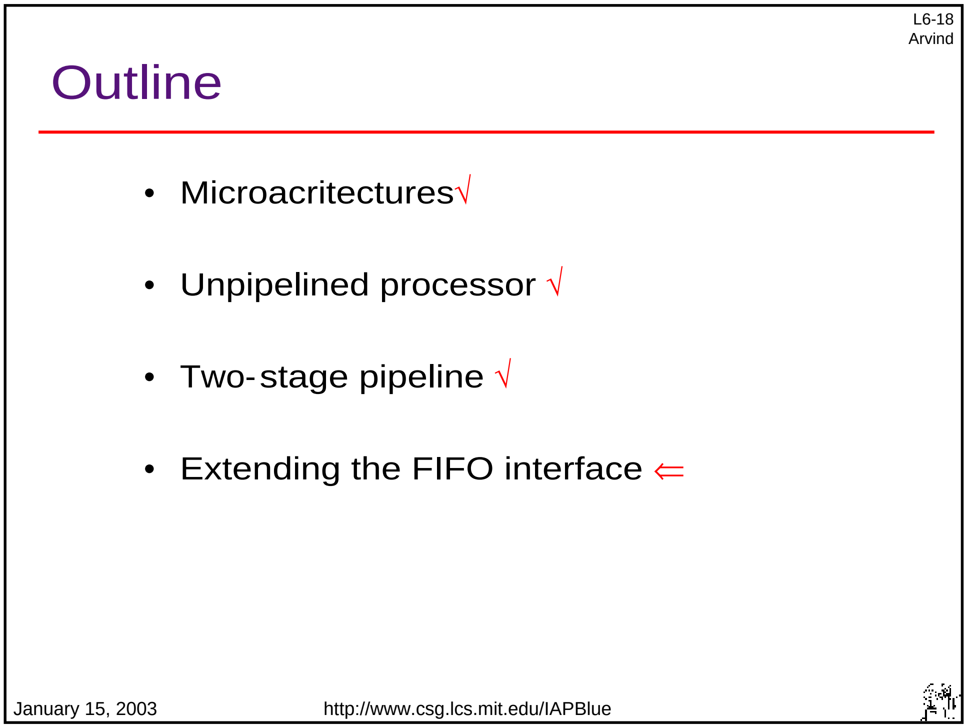


```
when (nextpc, EAdd rd va vb) <- bu.first
  ==>
   action
       rf.upd rd (va + vb)
       bu.deq
```



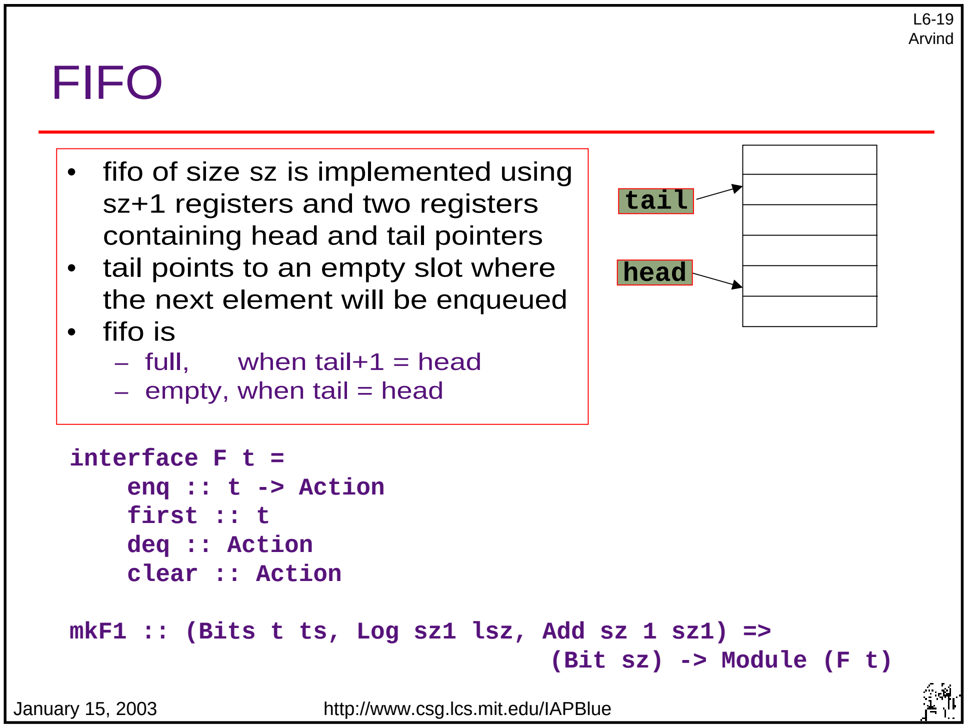
Execute Rule

```
"execute_rule":
when (True) ==>
  let (nextpc, instTemplate) = bu.first
  in   case instTemplate of
          EAdd rd va vb -> action rf.write rd (va + vb)
                                  bu.deq

          EBz  cv av    -> if (cv == 0) then action
                                                 pc := av
                                                 bu.clear
                           else bu.deq

          ELoad rd av   -> action
                               rf.write rd (dataMem.get av)
                               bu.deq

          EStore vv av  -> action dataMem.put av vv
                                  bu.deq
```



Outline

- Microacritectures√
- Unpipelined processor √
- Two-stage pipeline √
- Extending the FIFO interface ⇐



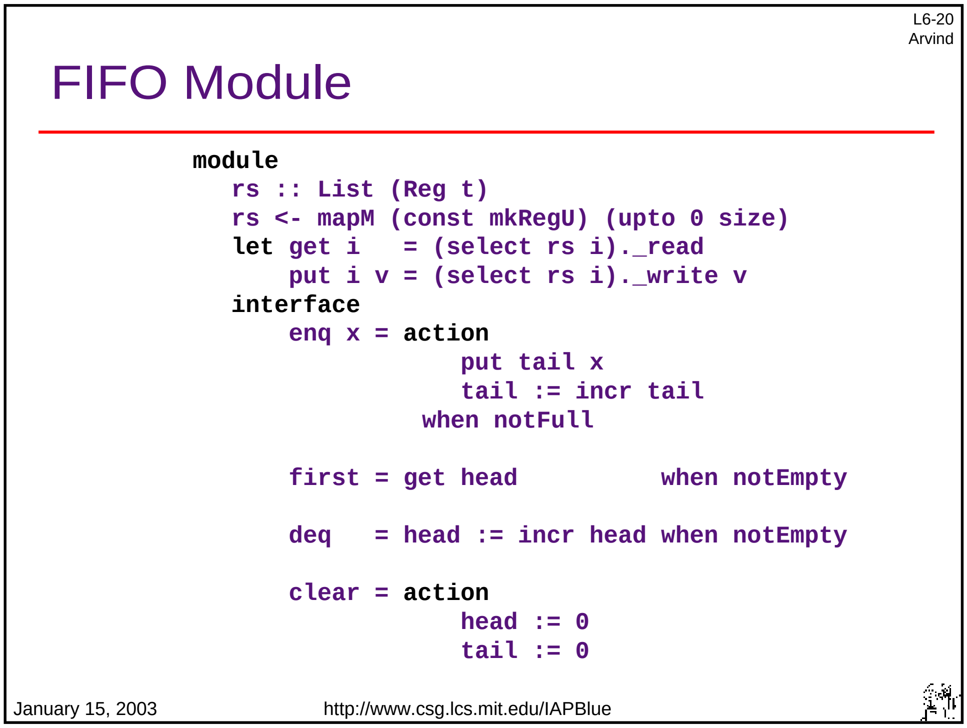
FIFO

- fifo of size sz is implemented using sz+1 registers and two registers containing head and tail pointers
- tail points to an empty slot where the next element will be enqueued
- fifo is
  - full, when tail+1 = head
  - empty, when tail = head

tail

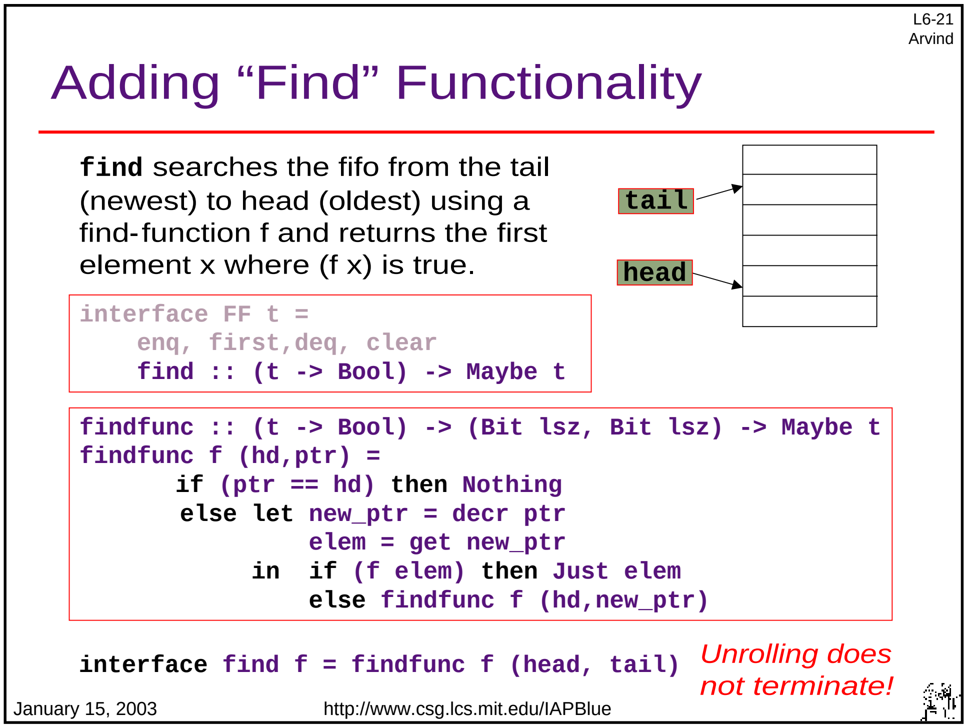
head

```
interface F t =
    enq :: t -> Action
    first :: t
    deq :: Action
    clear :: Action

mkF1 :: (Bits t ts, Log sz1 lsz, Add sz 1 sz1) =>
                                  (Bit sz) -> Module (F t)
```



FIFO Module

```
module
   rs :: List (Reg t)
   rs <- mapM (const mkRegU) (upto 0 size)
   let get i   = (select rs i)._read
       put i v = (select rs i)._write v
   interface
       enq x = action
                   put tail x
                   tail := incr tail
                when notFull

       first = get head           when notEmpty

       deq   = head := incr head when notEmpty

       clear = action
                   head := 0
                   tail := 0
```



Adding “Find” Functionality

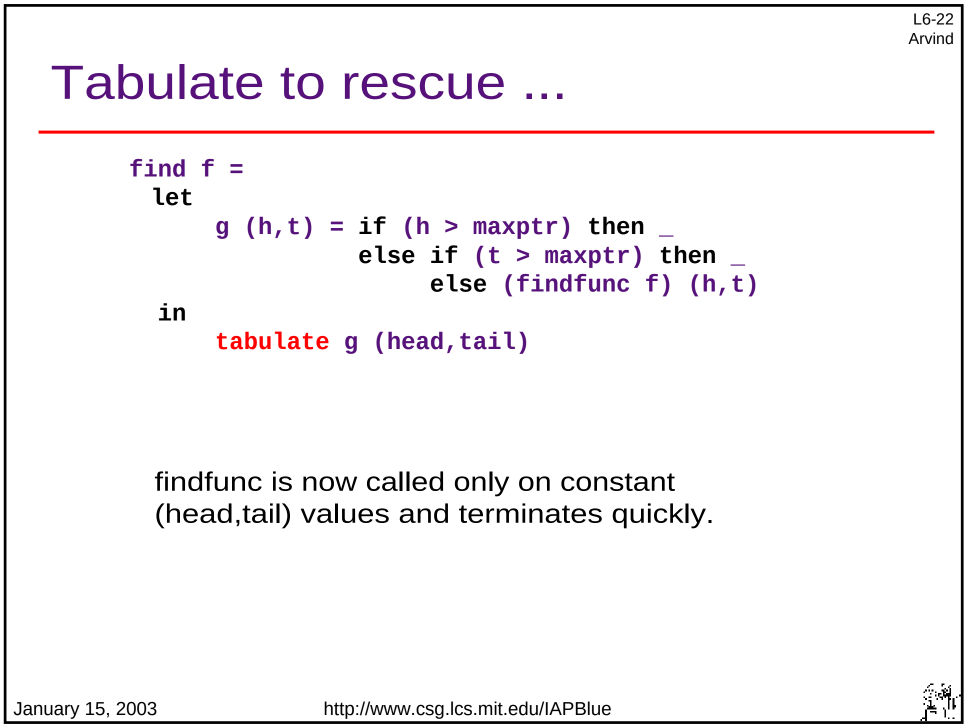
find searches the fifo from the tail (newest) to head (oldest) using a find-function f and returns the first element x where (f x) is true.

```
interface FF t =
    enq, first,deq, clear
    find :: (t -> Bool) -> Maybe t
```

```
findfunc :: (t -> Bool) -> (Bit lsz, Bit lsz) -> Maybe t
findfunc f (hd,ptr) =
       if (ptr == hd) then Nothing
        else let new_ptr = decr ptr
                 elem = get new_ptr
             in  if (f elem) then Just elem
                 else findfunc f (hd,new_ptr)
```

```
interface find f = findfunc f (head, tail)
```

Unrolling does not terminate!



Tabulate to rescue ...

```
find f =
  let
       g (h,t) = if (h > maxptr) then _
                 else if (t > maxptr) then _
                      else (findfunc f) (h,t)
   in
       tabulate g (head,tail)
```

findfunc is now called only on constant (head,tail) values and terminates quickly.