Bluespec-7
Out-of-Order Processor

Arvind
Laboratory for Computer Science
M.I.T.

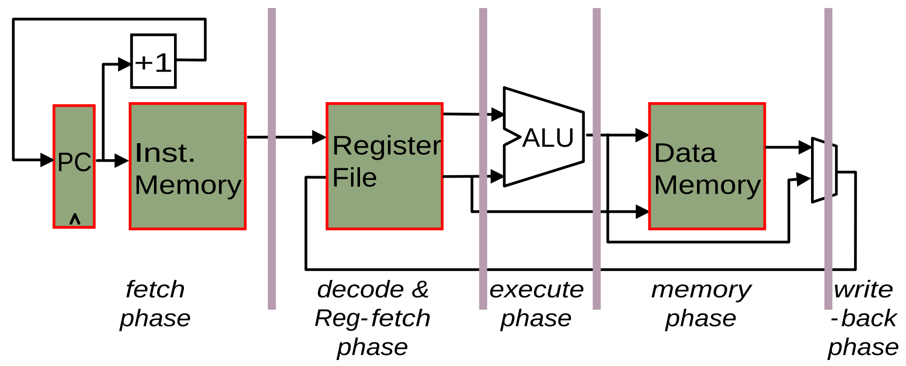
January 16, 2003

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

Outline

- Five-stage pipeline ⇐
  - with and without "bypassing"
- Modeling an out-of-order processor

Five-stage Pipelined Processor



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

five--stage pipeline rules are very closely related to the two-stage pipeline rules

Five-stage Pipeline State

```
mkFiveStage =
  module
      iMem        :: MemIF <- mkMem
      dMem        :: MemIF <- mkMem
      rf          :: RegFile  <- mkRegFile
      pc          :: Reg Ia   <- mkReg 0
      bf          :: FIFO  (Ia, (Bit 32)) <- mkFIFO
      bd          :: BFIFO (Ia, InstTemplate) <- mkBFIFO
      be          :: BFIFO (Ia, InstTemplate) <- mkBFIFO
      bm          :: BFIFO (Ia, InstTemplate) <- mkBFIFO

      rules
             ...
```

Bypass FIFO’s are FIFO’s with the “find” function

Instruction Template

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

type Value  = Bit 32
```

After the execution an instruction template contains a value and the name of the destination register

Five-stage Pipeline Rules:
fetch, execute

```
"fetch":
  when (True) ==>
        action bf.enq (pc, iMem.get pc)
               pc := pc + 1
```

```
"execute_rule":
   when (True) ==>
        let (epc, it) = bd.first
        in   case it of
              EAdd rd va vb ->
                     action  be.enq (epc, EVal rd (va + vb))
                             bd.deq
              EBz  cv av    -> if (cv == 0) then
                                   action  pc := av
                                           bd.clear
                                           bf.clear
                                else bd.deq
              _             -> action be.enq (epc, it)
                                      bd.deq
```

Five-stage Pipeline Rules:
memory, write-back

```
"mem_rule":
 when (True) ==>
   let (mpc, it) = be.first
   in  action
         case it of
           ELoad rd av  -> bm.enq (mpc, EVal rd (dMem.get av))
           EStore vv av -> dMem.put av vv
           _            -> bm.enq (mpc, it)
         be.deq
```

```
"wb_rule":
 when (True) ==>
   let (wpc, it) = bm.first
   in action
        case it of
             EVal rd v -> rf.write rd v
             _         -> noAction
        bm.deq
```

Five-stage Pipeline Rules:
decode&operand-fetch

```
if stall then noAction else action
                                 enqAction
                                 bf.deq
```

where

```
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  = (chkbuf bd r) || (chkbuf be r) || (chkbuf bm r)
```

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

no change needed in the “find” method – only need to adjust “findf”

Five-stage Pipeline Rules:
decode&operand fetch

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

(findf r) is applied to each element of the fifo to determine if it contains r

```
enqAction =
  case instr of
    Add rd ra rb -> bd.enq (dpc, (EAdd  rd (rval ra)(rval rb)))
    Bz cd addr   -> bd.enq (dpc, (EBz   (rval cd) (rval addr)))
    Load rd addr -> bd.enq (dpc, (ELoad   rd (rval addr)))
    Store v addr -> bd.enq (dpc, (EStore (rval v) (rval addr)))

rval r    = rf.sub r
```

what about bypassing values?

Five-Stage Pipeline with Bypassing

Very few changes are needed:

1. Do not stall if rd value is available

```
chkbuf b r = case b.find (findf r) of
                    Nothing             -> False
                    Just (pc, EVal rd _)-> False
                    Just  _             -> True;
```

2. Extract the bypass value

```
bypassVal b r = case b.find (findf r) of
                    Just (pc, EVal rd v) -> Just v
                    _                    -> Nothing
```

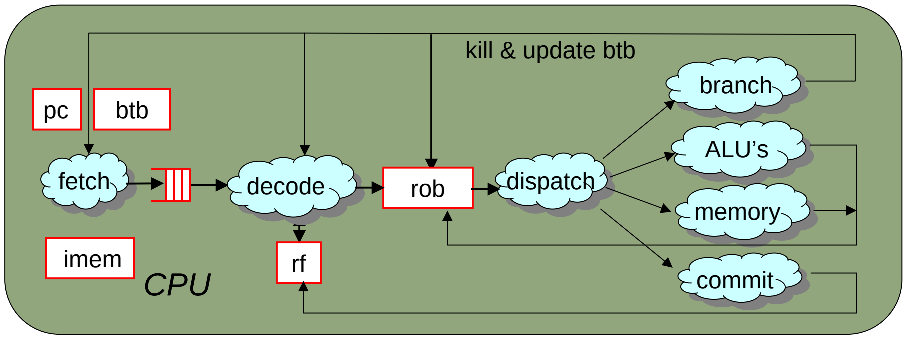
3. Get the r value from the FIFO closest to you

```
rval r = case (bypassVal bd r) of
           Nothing -> case (bypassVal be r) of
                        Nothing -> case (bypassVal bm r) of
                                     Nothing -> rf.sub r
                                     Just v  -> v
                        Just v  -> v
           Just v  -> v
```

Outline

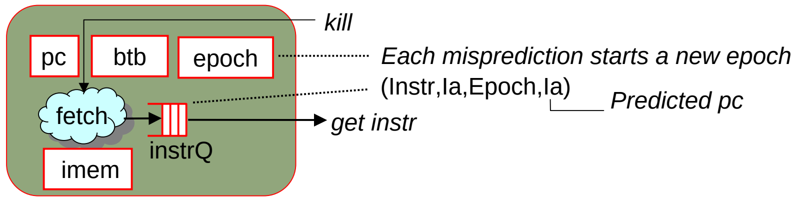
- Five-stage pipeline √
  - with and without “bypassing”
- Modeling an out-of-order processor ⇐

Modern Microprocessors
register renaming and speculative execution



- Capture the essence of the microarchitecture in a high-level model
- Design the details of each subunit
  - Same specification for simulation, verification and synthesis

Fetch Unit



```
mkFetchUnit imem btb =
  module
    curEpoch    :: Reg Epoch <- mkReg 0
    pc          :: Reg Ia    <- mkReg 0
    instrQ      :: FIFO(Instr,Ia,Epoch,Ia) <- mkFIFO
    rules
       …
    interface
        getInstr      = instrQ.get
        kill ia epoch = action
                          pc := ia
                          curEpoch := epoch
```

Fetch Unit Rule

```
mkFetchUnit imem btb =
  module
    curEpoch    :: Reg Epoch <- mkReg 0
    pc          :: Reg Ia    <- mkReg 0
    instrQ      :: FIFO(Instr,Ia,Epoch,Ia) <- mkFIFO
    rules
     when True
      ==> action
           let instr   = imem.get pc
               pIa     = btb.getTarget pc
           instrQ.enq  (instr,pc,curEpoch,pIa)
           pc := pIa
    interface
        getInstr      = instrQ.get
        kill ia epoch = action
                          pc := ia
                          curEpoch := epoch
```

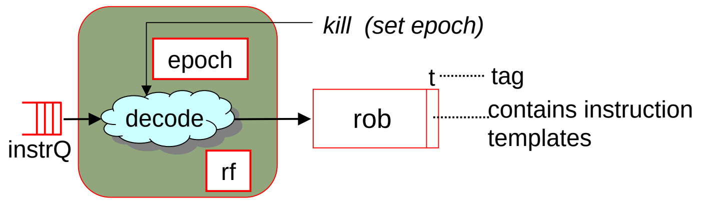
instrQ.notfull check is implicit !

Instruction Templates

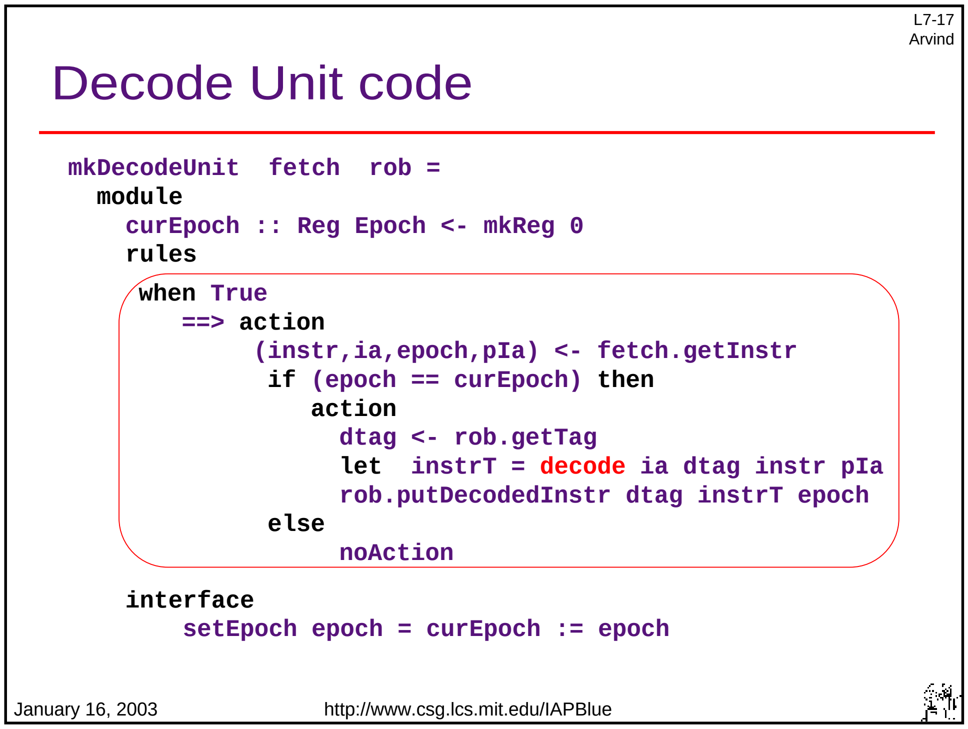
- Decode unit packs instructions into instruction templates by replacing operands either by their value or by their register renaming tag.

```
data Opcode = AddOp | BzOp | LoadOp | StoreOp

data TagOrValue = T  Tag | V  Value

struct InstrTemplate =
    ia        :: Ia
    opcode    :: Opcode
    tv1       :: TagOrValue
    tv2       :: TagOrValue
    dval      :: Value
    destReg   :: Rname
    predIa    :: Ia        -- predicted instr addr
```

Decode Unit



- Decode unit
  - decodes each instruction
  - renames registers
  - gets the operand value or its tag from the reorder buffer (rob) or register file (rf)
  - puts the decoded instruction template into rob

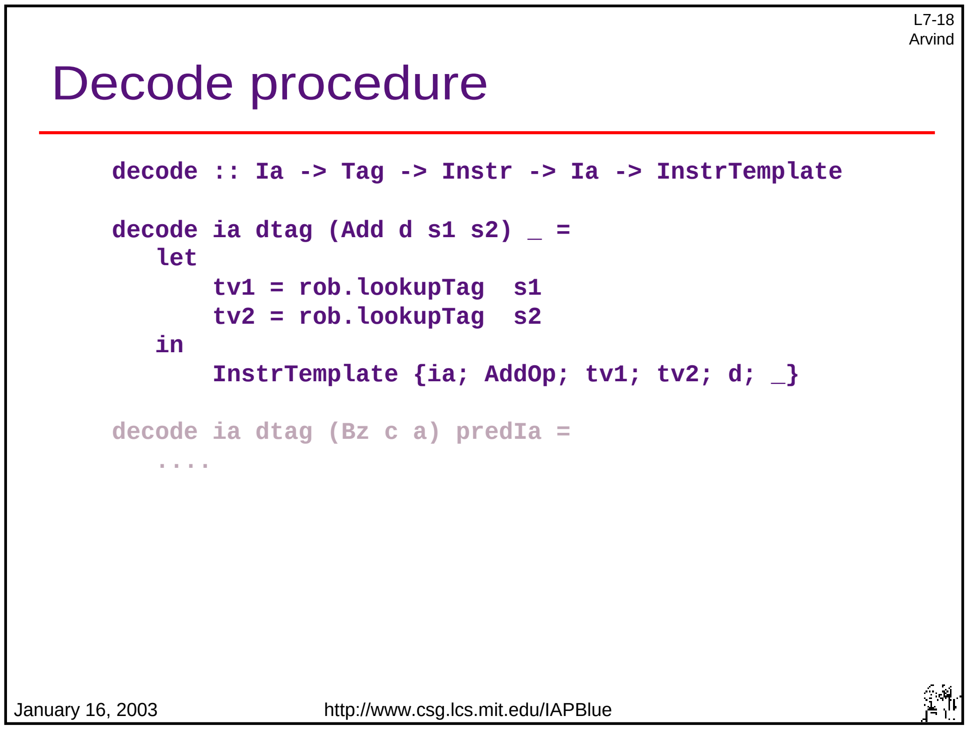


L7-17
Arvind

Decode Unit code

```
mkDecodeUnit  fetch  rob =
  module
    curEpoch :: Reg Epoch <- mkReg 0
    rules
     when True
        ==> action
             (instr,ia,epoch,pIa) <- fetch.getInstr
              if (epoch == curEpoch) then
                 action
                   dtag <- rob.getTag
                   let  instrT = decode ia dtag instr pIa
                   rob.putDecodedInstr dtag instrT epoch
              else
                   noAction

    interface
        setEpoch epoch = curEpoch := epoch
```

January 16, 2003 http://www.csg.lcs.mit.edu/IAPBlue



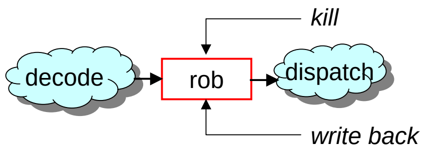
L7-18
Arvind

Decode procedure

```
decode :: Ia -> Tag -> Instr -> Ia -> InstrTemplate

decode ia dtag (Add d s1 s2) _ =
   let
       tv1 = rob.lookupTag  s1
       tv2 = rob.lookupTag  s2
   in
       InstrTemplate {ia; AddOp; tv1; tv2; d; _}

decode ia dtag (Bz c a) predIa =
   ....
```

January 16, 2003 http://www.csg.lcs.mit.edu/IAPBlue

Reorder Buffer: Interface



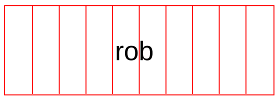
decode unit

dispatch unit

execution unit

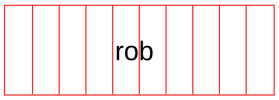
```
interface RoB =
    lookupTag         :: RName -> TagOrValue
    getTag            :: ActionValue Tag
    putDecodedInstr   :: Tag -> InstrTemplate -> Epoch
                                                -> Action
    retireInstr       :: Get InstrTemplate

    getEnabledBranch  :: Get (Tag,InstrTemplate)
    getEnabledAlu     :: Get (Tag,InstrTemplate)
    getEnabledMemOp   :: Get (Tag,InstrTemplate)

    putBranchOutput   :: Tag -> BranchResult -> Action
    putAluOutput      :: Tag -> Value -> Action
    putMemOpOutput    :: Tag -> Value -> Action
```

Reorder Buffer (rob)



- rob is a circular buffer:
  - instructions are assigned tags inorder and are retired in order
- Each rob slot has a state associated with it
  - Empty | Waiting | Dispatched | Killed | Done
  - rules to monitor “Killed/Done” state
  - rules to monitor if all the operands are available, and if they are then to enque them into the appropriate execution unit
- When branch unit returns a result
  - the branch instruction either becomes a nop
  - or it kills itself and all instructions that follow it

Reorder Buffer: Internal state



- A set of complex rules controls the behavior of each slot

```
module
    ...
    slots :: List (Reg (State,InstrTemplate))
    slots <- mapM (\j -> mkReg (Empty, _)) (upto 0 n)

    addRules (map mkSlotRules (upto 0 n))

    interface ...
```

Out-of-order Processor

```
mkOOP :: Module Empty
mkOOP =
  module
    instrMem    <- mkInstrMem
    dataMem     <- mkDataMem
    rf          <- mkRegFile
    btb         <- mkBranchTargetBuffer

    fetchUnit   <- mkFetchUnit instrMem btb
    rob         <- mkRoB  rf
    decodeUnit  <- mkDecodeUnit fetchUnit rob

    branchUnit  <- mkBranchUnit fetchUnit
                                 decodeUnit rob btb
    aluUnit     <- mkAluUnit rob
    memUnit     <- mkMemUnit dataMem  rob
```