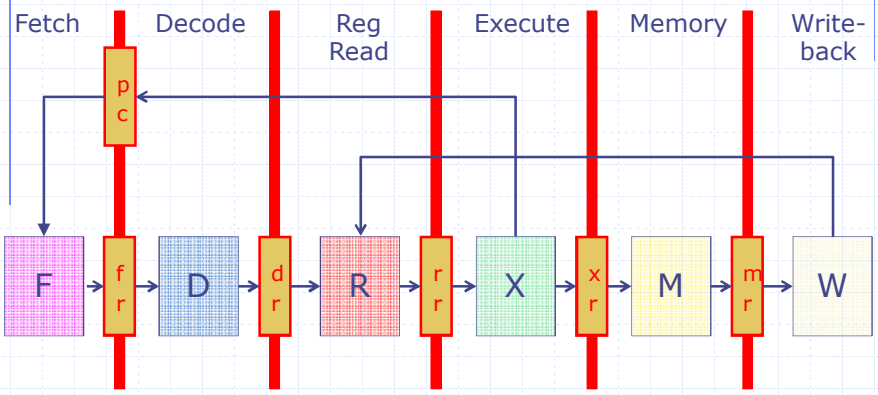
Computer Architecture: A Constructive Approach

Bypassing

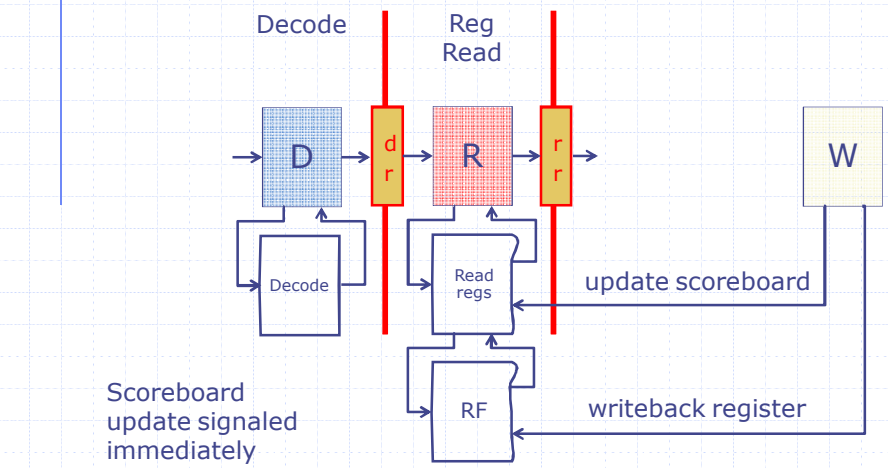
Joel Emer
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Six Stage Pipeline



Writeback feeds back directly to ReadReg rather than through FIFO

Register Read Stage



Per stage architectural state

```
typedef struct {
  MemResp instResp;
} FBundle;

typedef struct {
    Rindx    r_dest;
    Rindx    op1;
    Rindx    op2;
    ...
} DecBundle;

typedef struct {
    Data      src1;
    Data      src2;
} RegBundle;
```

```
typedef struct {
    Bool   cond;
    Addr   addr;
    Data   data;
} Ebundle;

No MemBundle –
  data added to Ebundle

No WbBundle –
   nothing added in WB
```

Per stage micro-architectural state

```
typedef struct {
   FBundle   fInst;
   DecBundle decInst;
   RegBundle regInst;
   Inum      inum;
   Epoch     epoch;
} RegData deriving(Bits,Eq);

function RegData newRegData(DecData decData);
   RegData regData = ?;
   regData.fInst   = decData.fInst;
   regData.decInst = decData.decInst;
   regData.inum    = decData.inum;
   regData.epoch   = decData.epoch;
   return regData;
endfunction
```

Architectural state from prior stages

New architectural state for this stage

Passthrough and (optional) new micro-architectural state

Utility function

Copy architectural state from prior stages

Copy uarch state from prior stages

In "uarch_types"

Example of stage state use

```
rule doRegRead;
  let regData = newRegData(dr.first());
  let decInst = regData.decInst;

  case (reg_read.readRegs(decInst)) matches
    tagged Valid .rfdata:
     begin
       if( writes_register(decInst))
         reg_read.markUnavail(decInst.r_dest);
       regData.regInst.src1 = rfdata.src1;
       regData.regInst.src2 = rfdata.src2;
       rr.enq(regData);
       dr.deq();
     end
  endcase
endrule
```

Initialize stage state

Set utility variables*

Set architectural state

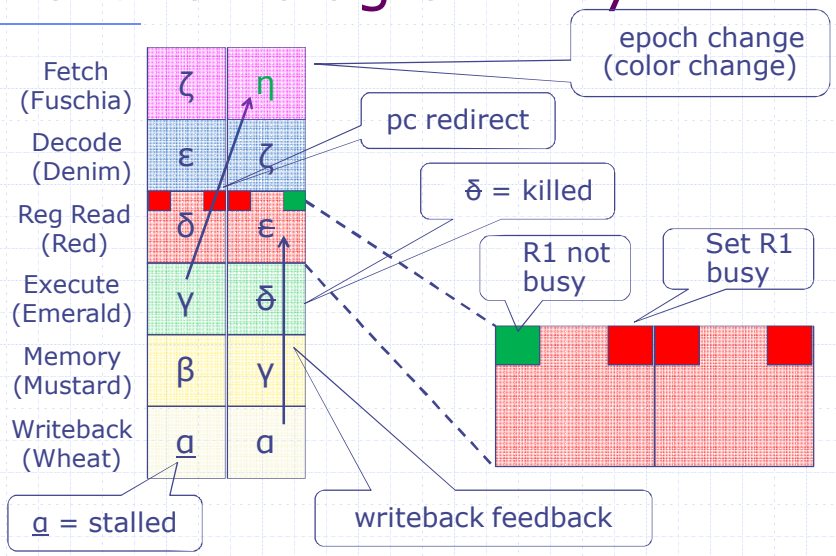
Set uarch state if any

Dequeue incoming state

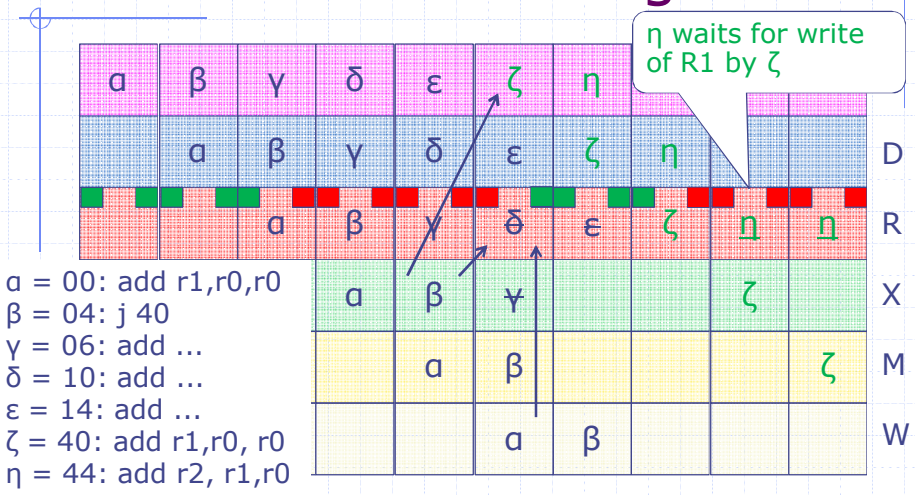
Enqueue outgoing state

*Don't try to update!!!

Waterfall diagram key



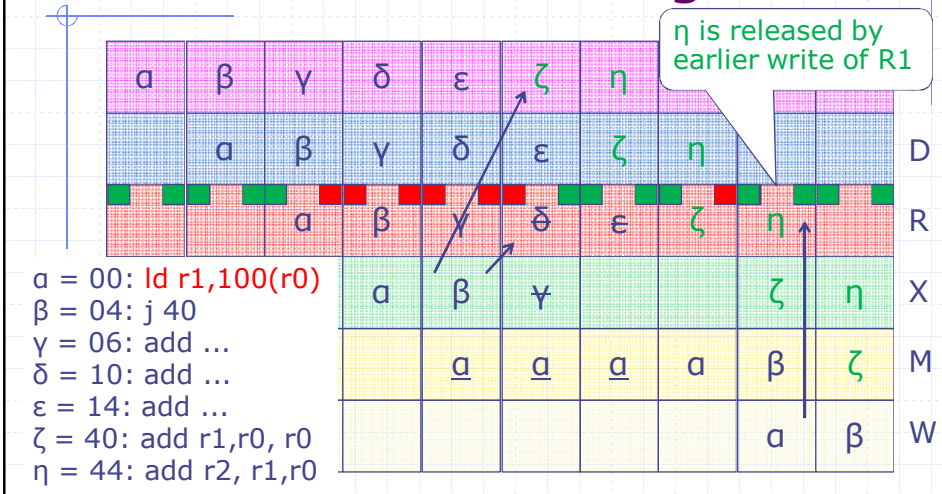
Scoreboard clear bug!



α = 00: add r1,r0,r0
β = 04: j 40
γ = 06: add ...
δ = 10: add ...
ε = 14: add ...
ζ = 40: add r1,r0, r0
η = 44: add r2, r1,r0

Looks okay, but what if α is delayed?

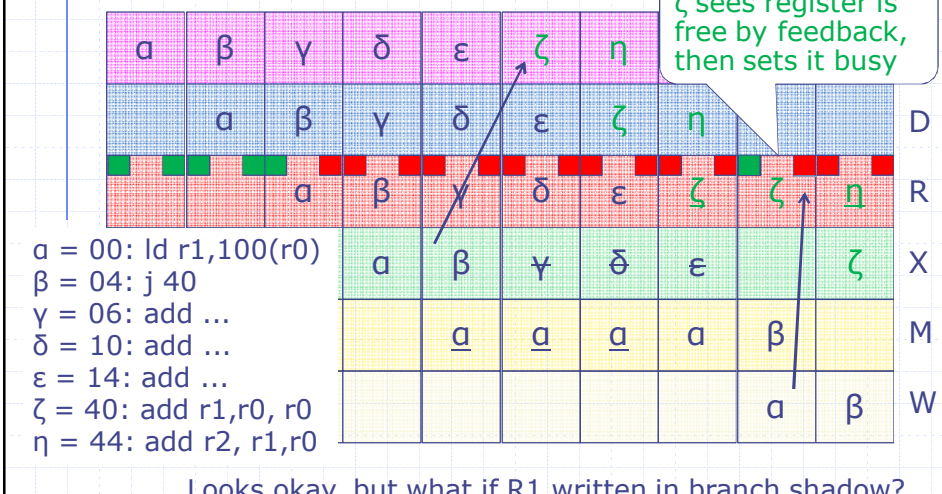
Scoreboard clear bug!



α = 00: ld r1,100(r0)
β = 04: j 40
γ = 06: add ...
δ = 10: add ...
ε = 14: add ...
ζ = 40: add r1,r0, r0
η = 44: add r2, r1,r0

Mistreatment of what did of data dependence?

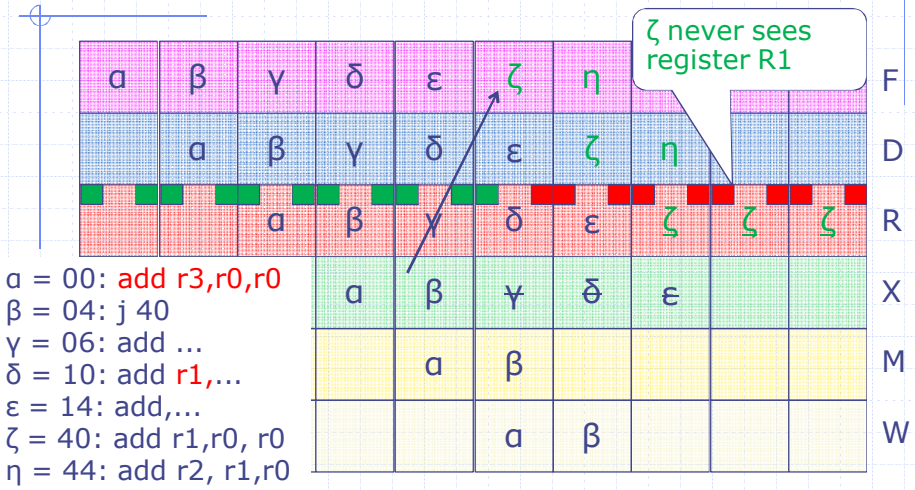
So don't clear scoreboard!



α = 00: ld r1,100(r0)
β = 04: j 40
γ = 06: add ...
δ = 10: add ...
ε = 14: add ...
ζ = 40: add r1,r0, r0
η = 44: add r2, r1,r0

Looks okay, but what if R1 written in branch shadow?

Dead instruction deadlock!

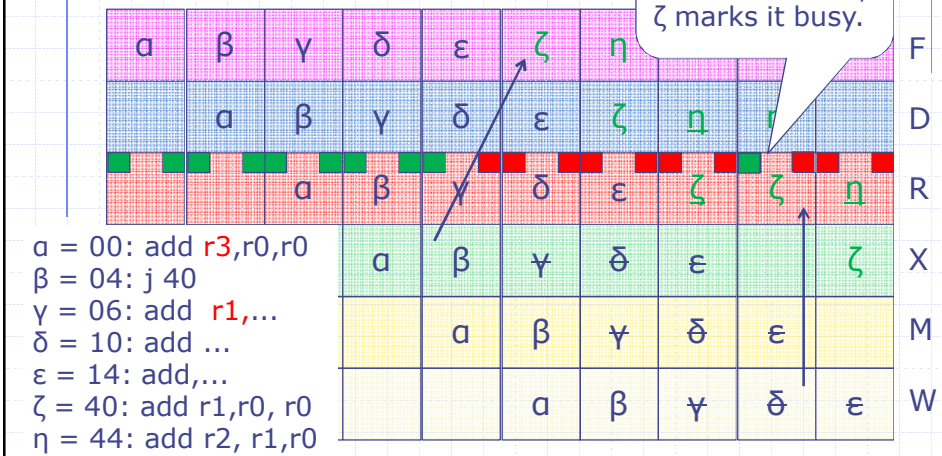


So, what can we do?

Poison bit uarch state

- In exec:
  - instead of dropping killed instructions mark them 'poisoned'
- In subsequent stages:
  - **DO NOT** make architectural changes
  - **DO** necessary bookkeeping

Poison-bit solution



α = 00: add r3,r0,r0
β = 04: j 40
γ = 06: add r1,...
δ = 10: add ...
ε = 14: add,...
ζ = 40: add r1,r0, r0
η = 44: add r2, r1,r0

What stage generates poison bit?

Poison-bit generation

```
rule doExecute;
  let execData = newExecData(rr.first);
  let ...;
  if (! epochChange)
  begin
    execData.execInst = exec.exec(decInst,src1,src2);
    if (execInst.cond) ...
  end
  else
  begin
    execData.poisoned = True;
  end
  xr.enq(execData);
  rr.deq();
endrule
```

Initialize stage state

Set utility variables

Set architectural state

Set uarch state

Enqueue outgoing state

Dequeue incoming state

What stages need to handle poison bit?

Poison-bit usage

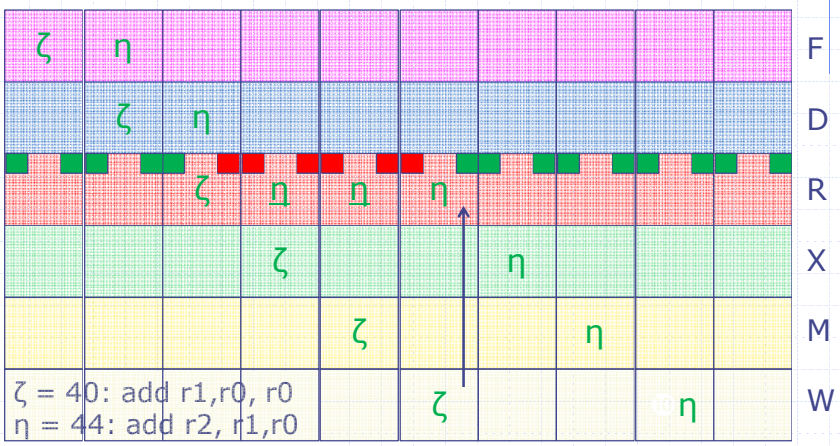
```
rule doMemory;
  let memData  = newMemData(xr.first);
  let ...;
  if(! poisoned && memop(decInst))
    memData.execInst.data <- mem.port(...);
  mr.enq(memData);
  xr.deq();
endrule
rule doWriteBack;
  let wbData = newWBData(mr.first);
  let ...;
  reg_read.markAvailable(rdst);
  if (!poisoned && regwrite(decInst)
    rf.wr(rdst, data);
  mr.deq;
endrule
```

Architectural activity if not poisoned

Unconditionally do bookkeeping

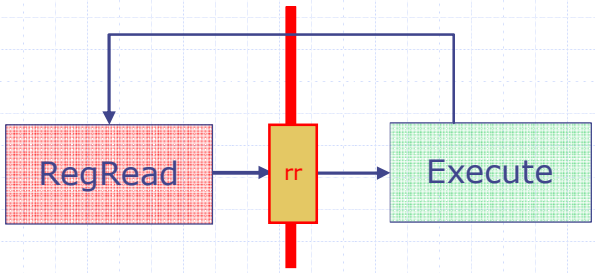
Architectural activity if not poisoned

Data dependence stalls



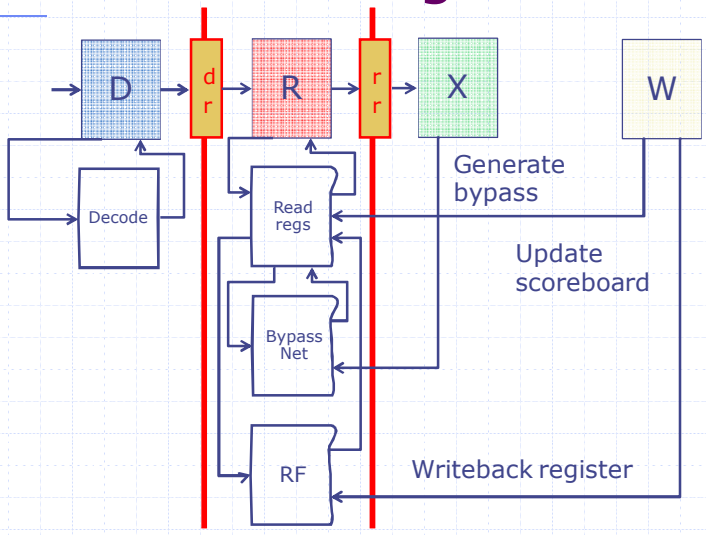
This is a data dependence. What kind?

Bypass



What does RegRead need to do?

Register Read Stage



Bypass Network

```
typedef  struct { Rindx regnum; Data  value;} BypassValue;

module mkBypass(BypassNetwork)
     Rwire#(BypassValue) bypass;

     method produceBypass(Rindx regnum, Data value);
       bypass.wset(BypassValue{regname: regnum, value:value});
     endmethod

     method Maybe#(Data) consumeBypass1(Rindx regnum);
       if (bypass matches tagged Valid .b && b.regnum == regnum)
           return tagged Valid b.value;
       else
           return tagged Invalid;
     endmethod
endmodule
```

Does bypass solve WAW hazard

Register Read (with bypass)

```
module mkRegRead#(RFile rf, BypassNetwork bypass)(RegRead);
  method Maybe#(Sources) readRegs(DecBundle decInst);
    let op1 = decInst.op1;       let op2 = decInst.op2;
    let rfdata1 = rf.rd1(op1);   let rfdata2 = rf.rd2(op2);
    let bypass1 = bypass.consumeBypass1(op1);
    let bypass2 = bypass.consumeBypass2(op2);
    let stall = isWAWhazard(scoreboard) ||
          (isBusy(op1,scoreboard) && !isJust(bypass1)) ||
          (isBusy(op2,scoreboard) && !isJust(bypass2)));
    let s1 = fromMaybe(rfdata1, bypass1);
    let s2 = fromMaybe(rfdata2, bypass2);
    if (stall) return tagged Invalid;
    else return tagged Valid Sources{src1:s1, src2:s2};
  endmethod
```

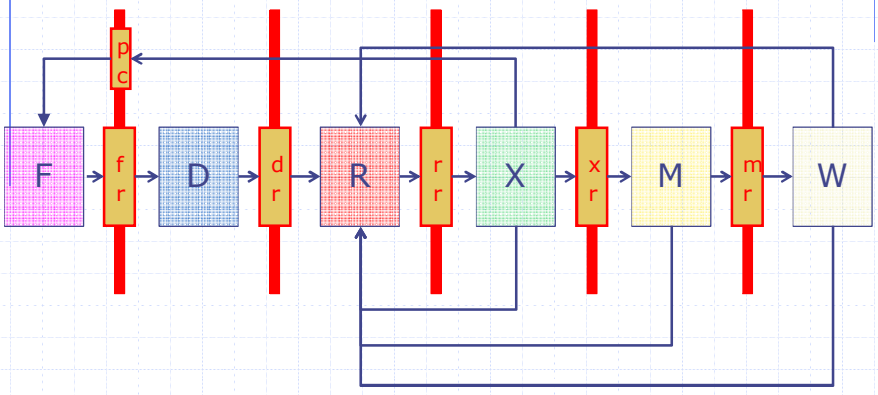
Bypass generation in execute

```
rule doExecute;
  let execData = newExecData(rr.first);
  let ...;
  if (! epochChange) begin
    execData.execInst = exec.exec(decInst,src1,src2);
    if (execInst.cond) ...
    if ( decInst.i_type == ALU )
      bypass.generateBypass(decInst.r_dst, execInst.data);
  end else begin
    execData.poisoned = True;
  end
  xr.enq(execData);
  rr.deq();
endrule
```

If have data destined for a register bypass it back to register read

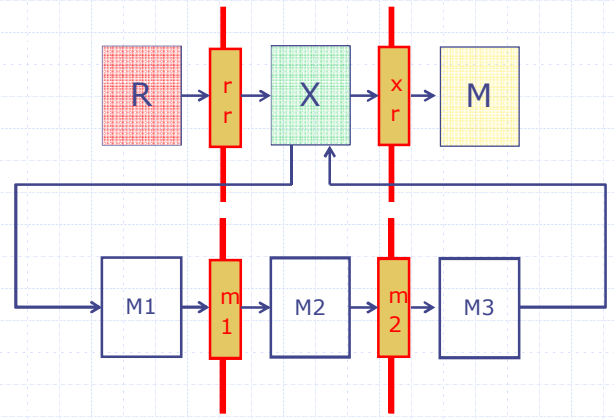
Does bypass solve all RAW delays?

Full bypassing



Every stage that generates register writes can generate bypass data

Multi-cycle execute



Incorporating a multi-cycle operation into a stage

Multi-cycle function unit

```
module mkMultistage(Multistage);
  State1 m1 <- mkReg();  State2 m2 <- mkReg();

  method Action request(Operand operand);
    m1.enq(doM1(operand));
  endmethod

  rule s1;
    m2.enq(doM2(m1.first)); m1.deq();
  endrule

  method ActionValue Result response();
     return doM3(m2.first); m2.deq();
  endmethod
endmodule
```

Perform first stage of pipeline

Perform middle stage(s) of pipeline

Perform last stage of pipeline and return result

Single/Multi-cycle pipe stage

```
rule doExec;
 ... initialization
  let done = False;
  if (! waiting) begin
    if(isMulticycle(...)) begin
       multicycle.request(...); waiting <= True;
    end else begin
      result = singlecycle.exec(...); done = True;
    end
  end else begin
    result <- multicycle.response();
    done = True; waiting <= False;
  end
  if (done)
    ...finish up
endrule
```

If not busy start multicycle operation, or…

…perform single cycle operation

If busy, wait for response

Finish up if have a result