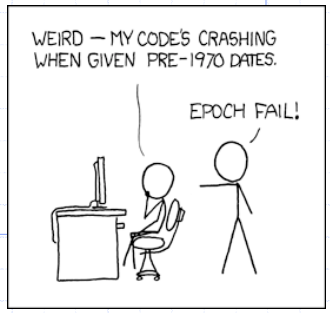
6.175: Constructive Computer Architecture

Tutorial 4

Epochs and Scoreboards

Quan Nguyen
(Hopes to help you avoid epoch fails)



https://xkcd.com/376/
(not *that* kind of epoch)

Outline

- Epochs
- Scoreboards and Bypass RFs
- Branch Prediction

Distributed Epoch Motivation

- Want to redirect instruction stream as soon as known
- Create two epoch registers:
  - Decode (dEpoch)
  - Execute (eEpoch)

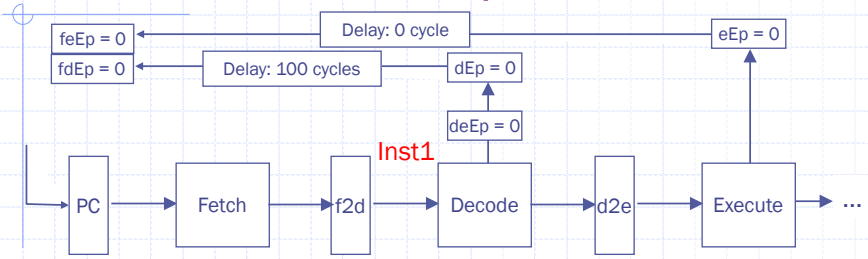
Terminology

- Each instruction has its idea of the:
  - Decode epoch (idEp)
  - Execute epoch (ieEp)
- Decode needs to estimate execute's epoch (deEpoch)
- Fetch has two registers too:
  - Decode (fdEp)
  - Execute (edEp)

Inst1 and Inst2

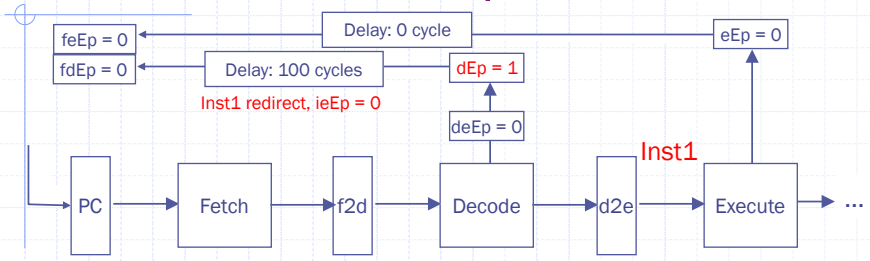
- Inst1 is an instruction causing redirection
  - branch, jump, etc.
- Inst2 is a wrong-path instruction that also produces a redirect
  - the decode stage doesn't know it's a wrong-path instruction
- Furthermore, decode redirects delayed 100 cycles (artificially)

1-bit Distributed Epochs



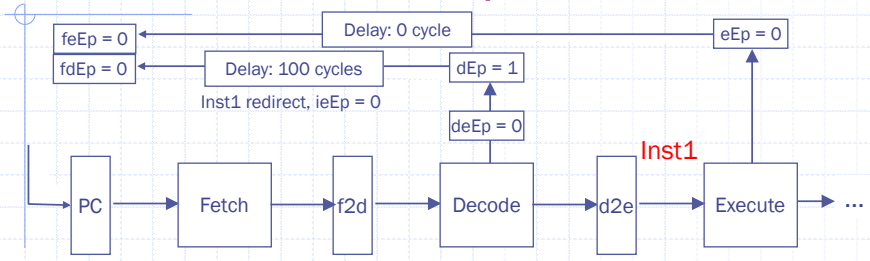
- Decode redirects Inst1 (ieEp = idEp = 0)

1-bit Distributed Epochs



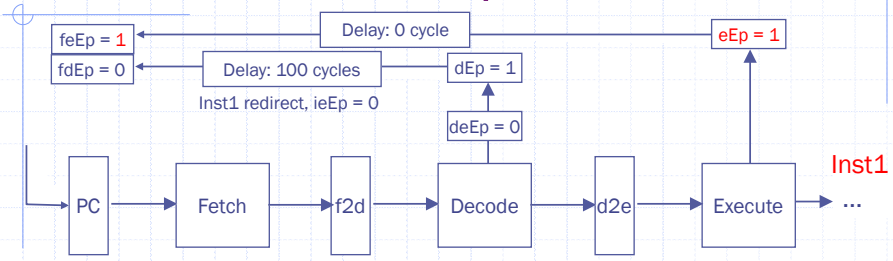
- Decode redirects Inst1 (ieEp = idEp = 0)

1-bit Distributed Epochs



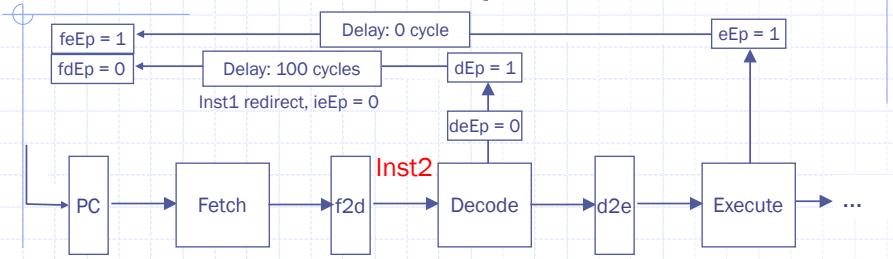
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1

1-bit Distributed Epochs



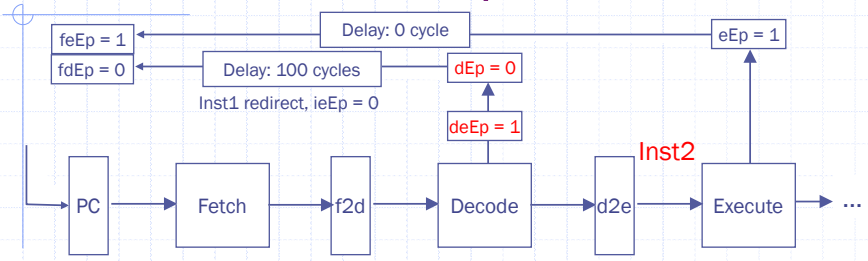
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1

1-bit Distributed Epochs



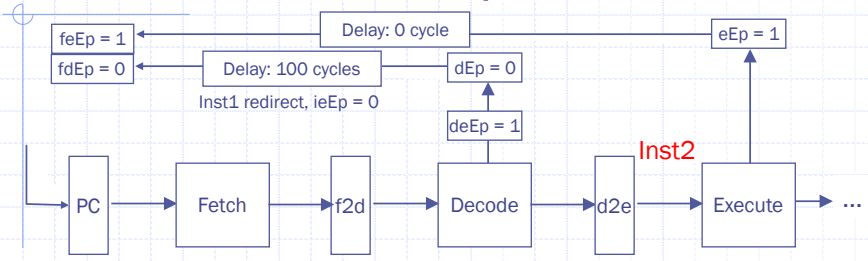
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1
- Correct-path Inst2 (ieEp = 1, idEp = 0) issues

1-bit Distributed Epochs



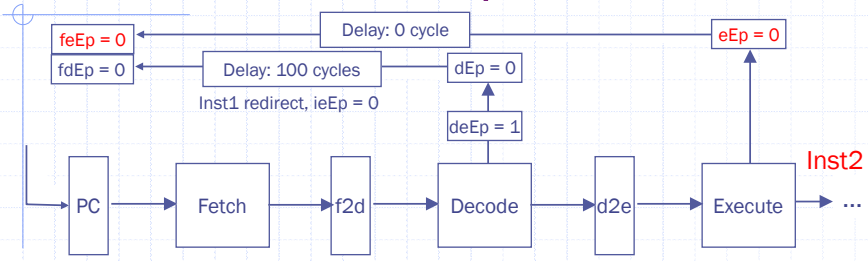
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1
- Correct-path Inst2 (ieEp = 1, idEp = 0) issues

1-bit Distributed Epochs



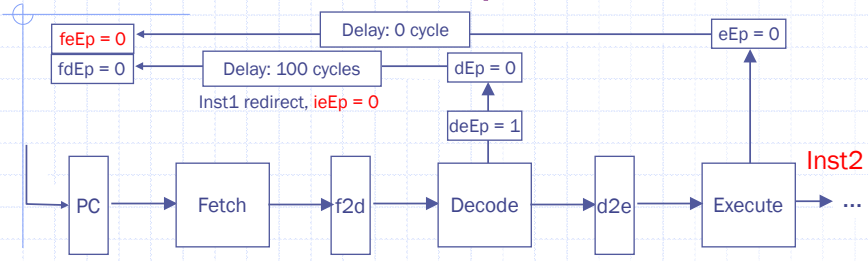
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1
- Correct-path Inst2 (ieEp = 1, idEp = 0) issues
- Execute redirects Inst2

1-bit Distributed Epochs



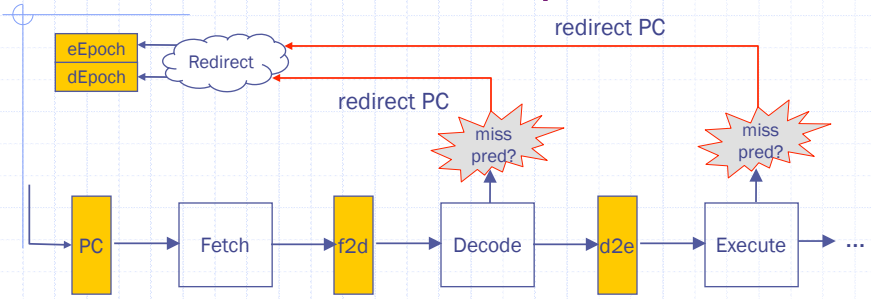
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1
- Correct-path Inst2 (ieEp = 1, idEp = 0) issues
- Execute redirects Inst2

1-bit Distributed Epochs



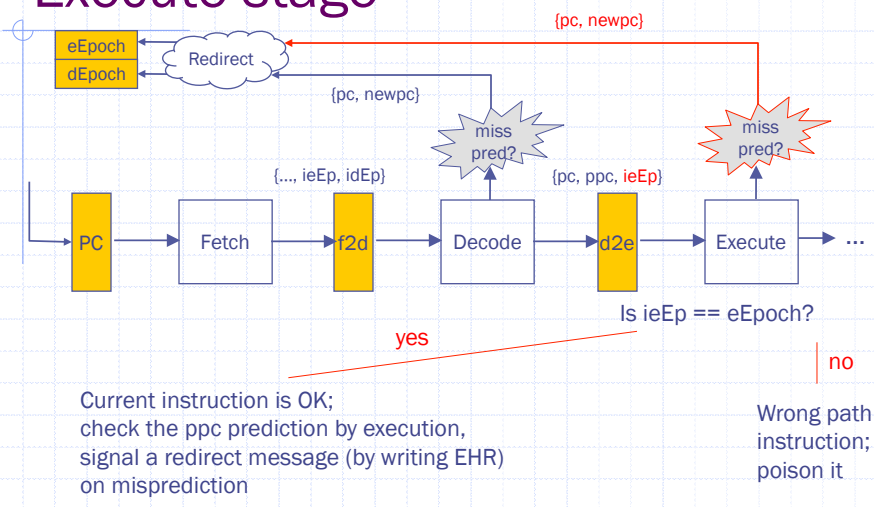
- Decode redirects Inst1 (ieEp = idEp = 0)
- Execute redirects Inst1
- Correct-path Inst2 (ieEp = 1, idEp = 0) issues
- Execute redirects Inst2
- Inst1 redirect arrives at Fetch (ieEp == feEp)
  - change PC to a wrong value

Unbounded Global Epochs

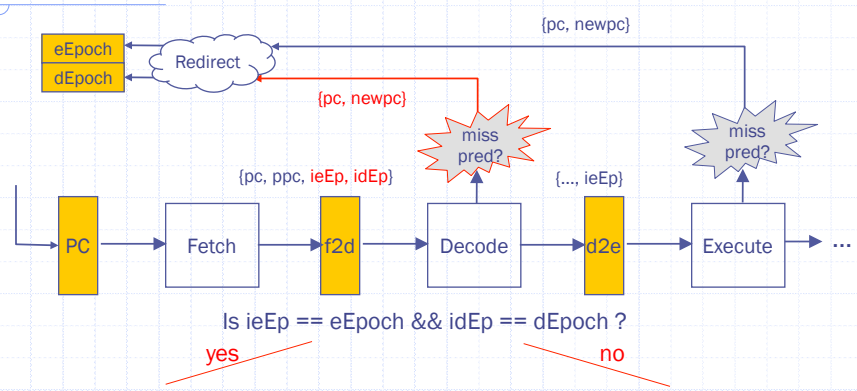


- Both Decode and Execute can redirect the PC
  - Execute redirect should never be overruled
- Global epoch for each redirecting stage
  - eEpoch: incremented when redirect from Execute takes effect
  - dEpoch: incremented when redirect from Decode takes effect
  - Initially set all epochs to 0

Execute stage



Decode Stage



Is ieEp == eEpoch && idEp == dEpoch ?

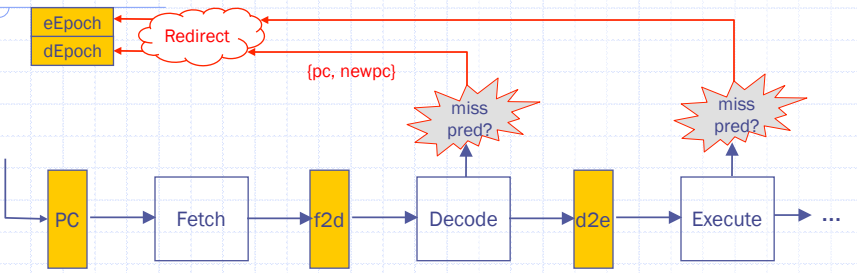
yes

Current instruction is OK;
check the ppc prediction via BHT, signal a redirect message (by writing EHR) on misprediction

no

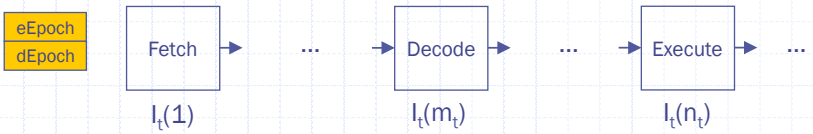
Wrong path instruction; drop it

Fetch stage: Redirect PC, Change Epoch



- If there is redirect message from Execute
  - Set PC to correct value, increment eEp
- Else if there is redirect message from Decode
  - Set PC to correct value, increment dEp
- We can always request IMem to fetch new instruction
  - New instruction will be tagged with current eEpoch and dEpoch
  - PC redirection overrides next-PC prediction
  - When PC redirects, new instruction must be killed at Decode stage

Observation



At cycle t, there are n_t instructions between Fetch and Execute: I_t (1 ... n_t)

- $I_t(1)$ at fetch, $I_t(n_t)$ at exectue, $I_t(m_t)$ at decode

Invariant:

- eEpoch = $I_t(1)$.ieEp ≥ $I_t(2)$.ieEp ≥ ... ≥ $I_t(n_t)$.ieEp ≥ eEpoch – 1
- dEpoch = $I_t(1)$.idEp ≥ $I_t(2)$.idEp ≥ ... ≥ $I_t(m_t)$.idEp ≥ dEpoch – 1

Proved by induction on t

- At cycle t, both Decode and Execute redirect
- $I_t(n_t)$.ieEp = eEpoch → eEpoch = $I_t(1)$.ieEp = $I_t(2)$.ieEp = ... = $I_t(n_t)$.ieEp
- $I_t(m_t)$.idEp = dEpoch → dEpoch = $I_t(1)$.idEp = $I_t(2)$.idEp = ... = $I_t(m_t)$.idEp
- eEpoch ← eEpoch + 1 (invariants still hold at cycle t+1)

1-bit Global Epochs

- The max difference of values of one type of epoch is 1
  - Only need 1 bit to encode each epoch
  - Increment epoch → flip epoch

4-stage Pipeline: Code Sketch

```
module mkProc(Proc);
  // stage 1: Fetch
  // stage 2: Decode & read register
  // stage 3: Execute & access memory
  // stage 4: Commit (write register file)
  Ehr#(2, Addr)     pc <- mkEhr(?);
  IMemory         iMem <- mkIMemory;
  ...
  Fifo#(2, Fetch2Decode)   f2d <- mkCFFifo;
  Fifo#(2, Decode2Execute) d2e <- mkCFFifo;
  Fifo#(2, Execute2Commit) e2c <- mkCFFifo;
  Reg#(Bool) eEpoch <- mkReg(False);
  Reg#(Bool) dEpoch <- mkReg(False);
  Ehr#(2, Maybe#(ExeRedirect)) exeRedirect <- mkEhr(Invalid);
  Ehr#(2, Maybe#(DecRedirect)) decRedirect <- mkEhr(Invalid);
  BTB#(16)   btb <- mkBTB;
  BHT#(1024) bht <- mkBHT;
```


Execute Stage

```
rule doExecute;
    d2e.deq; let x = d2e.first;
    if(x.ieEp == eEpoch) begin
        let eInst = exec(...);
        if(eInst.mispredict)
            exeRedirect[0] <= Valid ({pc: x.pc,
                                      newpc: eInst.addr});
      ...
    end
    else e2c.enq(Exec2Commit{poisoned: True, ...});
         // wrong path instruction, poison it
endrule
```

BHT training will be in lab 6

Decode Stage

```
rule doDecode;
    f2d.deq; let x = f2d.first;
    if(x.ieEp == eEpoch && x.idEp == dEpoch) begin
        let stall = ...; // check scoreboard for stall
        if(!stall) begin
            if(x.iType == Br) begin
                let bht_ppc = ...; // BHT prediction
                if(bht_ppc != x.ppc)
                    decRedirect[0] <= Valid ({pc: x.pc,
                                              newpc: bht_ppc});
            end
            ...
            d2e.enq(Decode2Execute{ieEp: x.ieEp, ...});
        end
    end
    // else: wrong path instruction, killed
endrule
```


Fetch Stage

```
rule doFetch;
    let inst = iMem.req(pc[0]);
    let ppc = btb.predPc(pc[0]);
    pc[0] <= ppc;
    f2d.enq(Fetch2Decode{pc: pc[0], ppc: ppc, inst: inst,
                         ieEp: eEpoch, idEp: dEpoch});
endrule

rule cononicalizeRedirect;
    if(exeRedirect[1] matches tagged Valid .r) begin
        pc[1] <= r.newpc; btb.update(r.pc, r.newpc);
        eEpoch <= !eEpoch;
    end else if(decRedirect[1] matches tagged Valid .r) begin
        pc[1] <= r.newpc; btb.update(r.pc, r.newpc);
        dEpoch <= !dEpoch;
    end
    exeRedirect[1] <= Invalid; decRedirect[1] <= Invalid;
endrule
```

Scoreboarding

- CDC 6600 (1965)
  - *Parallel Operation in the Control Data 6600*
- Capture RAW hazards
- Stepping stone towards out-of-order machines

Normal Register File

```
module mkRFile(RFile);
  Vector#(32, Reg#(Data)) rfile <- replicateM(mkReg(0));


  method Action wr(RIndx rindex, Data data);
    if (rindex != 0) (rfile[rindex]) <= data;
  endmethod
  method Data rd1(RIndx rindx) = rfile[rindx];
  method Data rd2(RIndx rindx) = rfile[rindx];
endmodule
```

{rd1, rd2} < wr

Bypass Register File using EHR

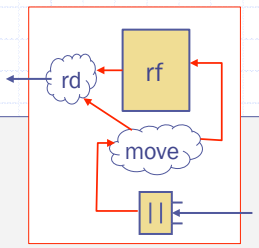
```
module mkBypassRFile(RFile);
  Vector#(32, Ehr#(2, Data)) rfile <-
                              replicateM(mkEhr(0));

  method Action wr(RIndx rindex, Data data);
    if (rindex != 0) (rfile[rindex])[0] <= data;
  endmethod
  method Data rd1(RIndx rindx) = (rfile[rindx])[1];
  method Data rd2(RIndx rindx) = (rfile[rindx])[1];
endmodule
```

wr < {rd1, rd2}

Bypass Register File

with external bypassing




```
module mkBypassRFile(BypassRFile);
  RFile             rf <- mkRFile;
  Fifo#(1, Tuple2#(RIndx, Data))
                 bypass <- mkBypassSFifo;
  rule move;
    begin rf.wr(bypass.first); bypass.deq end;
  endrule
  method Action wr(RIndx rindx, Data data);
    if(rindex!=0) bypass.enq(tuple2(rindx, data));
  endmethod
  method Data rd1(RIndx rindx) =
      return (!bypass.search1(rindx)) ? rf.rd1(rindx)
             : bypass.read1(rindx);
  method Data rd2(RIndx rindx) =
      return (!bypass.search2(rindx)) ? rf.rd2(rindx)
             : bypass.read2(rindx);
endmodule
```

wr < {rd1, rd2}

Scoreboard implementation
using searchable Fifos

```
function Bool isFound
        (Maybe#(RIndx) dst, Maybe#(RIndx) src);
  return isValid(dst) && isValid(src) &&
            (fromMaybe(?,dst)==fromMaybe(?,src));
endfunction

module mkCFScoreboard(Scoreboard#(size));
  SFifo#(size, Maybe#(RIndx), Maybe#(RIndx))
      f <- mkCFSFifo(isFound);
  method insert  = f.enq;
  method remove  = f.deq;
  method search1 = f.search1;
  method search2 = f.search2;
endmodule
```

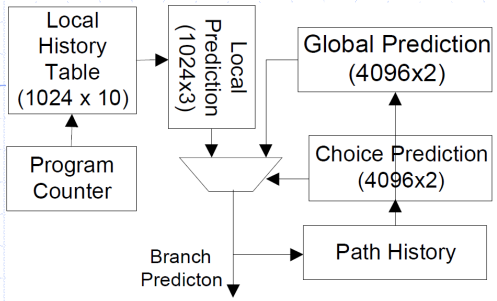
- Will see code in Lab 6

Branch Prediction

Advanced Branch Predictor

- BHT cannot predict very accurately
  - Use combinations from many predictors
- Global branch history
  - taken/not taken for previous branches
- Local branch history
  - taken/not taken for previous occurrences of the same branch
- Tournament branch predictor
  - Use both global and local history
  - Alpha 21264 (*The Alpha 21264 Microprocessor* and The Alpha 21264 Microprocessor Architecture)

Tournament Predictor

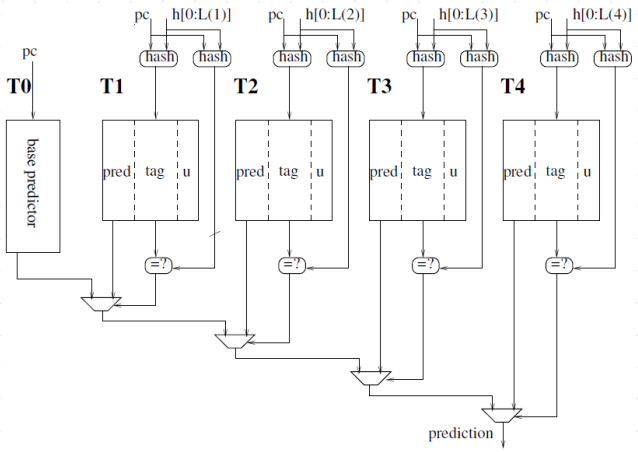


"The Alpha 21264 Microprocessor Architecture"

- 10-bit PC: index 1024 x 10-bit local history table
- 10-bit local history
  - Index 1024 x 3-bit BHT: prediction 1
- 12-bit global history
  - Index 4096 x 2-bit BHT: prediction 2
  - Index 4096 x 2-bit BHT: select between predictions 1, 2

TAGE Branch Predictors

- *A case for partially TAgged GEometric history length branch prediction*



Other Predictors

- Perceptron-based predictors
  - Classification problem in machine learning
  - Recent implementation: Samsung Exynos M1 (HotChips 2016)
- Championship Branch Predictor

Return Address Stack

- Use a stack to store return addresses from function call
  - Function call: push
  - Function return: pop
- Not a “call stack”, only contains return addresses
- Instruction to return from function call
  - JALR: rd = x0, rs1 = x1 (ra)
- Instruction to initiate function call
  - JAL: rd = x1
  - JALR: rd = x1

Credits

- Considerable previous material adapted from last year’s tutorials by Sizhuo Zhang and Andy Wright