Instruction set

typedef enum {R0;R1;R2;…;R31} RName;
typedef union tagged {
    struct {RName dst; RName src1; RName src2;} Add;
    struct {RName condR; RName addrR;}          Bz;
    struct {RName addrR;}            Load;
    struct {RName valueR; RName addrR;}         Store
} Iinstr deriving(Bits, Eq);

typedef Bit#(32) Iaddress;
typedef Bit#(32) Daddress;
typedef Bit#(32) Value;

An instruction set can be implemented using many different microarchitectures
**Deriving Bits**

typedef struct {...} Foo
deriving (Bits);

- To store datatypes in register, fifo, etc. we need to know how to represent them as bits (pack) and interpret their bit representation (unpack)
- Deriving annotation automatically generates the “pack” and “unpack” operations on the type (simple concatenation of bit representations of components)
- It is possible to customize the pack/unpack operations to any specific desired representation

---

**Tagged Unions: *Bit* Representation**

typedef union tagged {
  struct {RName dst; RName src1; RName src2;} Add;
  struct {RName condR; RName addrR;} Bz;
  struct {RName dst; RName addrR;} Load;
  struct {RName dst; Immediate imm;} AddImm;
} Instr deriving(Bits, Eq);

- 00 dst src1 src2
- 01 condR addrR
- 10 dst addrR
- 11 dst imm

Automatically derived representation; can be customized by the user written pack and unpack functions
The Plan

- Non-pipelined processor
- Two-stage Inelastic pipeline
- Two-stage Elastic pipeline – next lecture

Some understanding of simple processor pipelines is needed to follow this lecture.

Non-pipelined Processor

```verilog
module mkCPU#(Mem iMem, Mem dMem)
  (Reg#(Iaddress) pc <- mkReg(0);
   RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
   Instr instr = iMem.read(pc);
   Iaddress predIa = pc + 1;
   rule fetch_Execute ...
  endmodule
```

Non-pipelined processor rule

```verilog
rule fetch_Execute (True);
  case (instr) matches
    tagged Add (dst:.rd, src1:.ra, src2:.rb): begin
      rf.upd(rd, rf[ra]+rf[rb]);
      pc <= predIa;
    end
    tagged Bz (cond:.rc, addr:.ra): begin
      pc <= (rf[rc]==0) ? rf[ra] : predIa;
    end
    tagged Load (dest:.rd, addr:.ra): begin
      rf.upd(rd, dMem.read(rf[ra]));
      pc <= predIa;
    end
    tagged Store (value:.rv, addr:.ra): begin
      dMem.write(rf[ra], rf[rv]);
      pc <= predIa;
    end
  endcase
endrule
```

Assume "magic memory", i.e. responds to a read request in the same cycle and a write updates the memory at the end of the cycle.

Register File

- How many read ports?
- How many write ports?
- Concurrency properties?
The Plan

- Non-pipelined processor
- Two-stage Inelastic pipeline
- Two-stage Elastic pipeline

Two-stage Inelastic Pipeline

- Fetch & decode
- Execute
- pc, rf, dMem

Actions to be performed in parallel every cycle:
- Fetch Action: Decodes the instruction at the current pc and fetches operands from the register file and stores the result in buReg
- Execute Action: Performs the action specified in buReg and updates the processor state (pc, rf, dMem)

rule InelasticPipeline2(True);
fetchAction; executeAction; endrule
Instructions & Templates

buReg contains instruction templates, i.e., decoded instructions

typedef union tagged {
    struct {RName dst; RName src1; RName src2} Add;
    struct {RName condR; RName addrR} Bz;
    struct {RName dst; RName addrR} Load;
    struct {RName valueR; RName addrR} 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 val; Daddress addr} EStore;
} InstTemplate deriving(Eq, Bits);


Fetch & Decode Action
Fill the buReg with a decoded instruction

buReg <= newIt(instr);

function InstrTemplate newIt(Instr instr);
    case (instr) matches
        tagged Add {dst:.rd,src1:.ra,src2:.rb}:
            return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
        tagged Bz {condR:.rc,addrR:.addr}:
            return EBz{cond:rf[rc],tAddr:rf[addr]};
        tagged Load {dst:.rd,addrR:.addr}:
            return ELoad{dst:rd,addrR:rf[addr]};
        tagged Store{valueR:.v,addrR:.addr}:
            return EStore{val:rf[v],addr:rf[addr]};
    endcase
endfunction

Execute Action: Reads \( \text{buReg} \) and modifies state \((\text{rf}, \text{dMem}, \text{pc})\)

\[
\text{case (buReg) matches}
\]

- \( \text{tagged EAdd}\{\text{dst:.rd, op1:.va, op2:.vb}\}: \)
  \[
  \begin{align*}
  &\text{begin rf.upd(rd, va+vb);} \\
  &\text{pc <= predIa; end}
  \end{align*}
\]

- \( \text{tagged ELoad}\{\text{dst:.rd, addr:.av}\}: \)
  \[
  \begin{align*}
  &\text{begin rf.upd(rd, dMem.read(av));} \\
  &\text{pc <= predIa; end}
  \end{align*}
\]

- \( \text{tagged EStore}\{\text{val:.vv, addr:.av}\}: \)
  \[
  \begin{align*}
  &\text{begin dMem.write(av, vv);} \\
  &\text{pc <= predIa; end}
  \end{align*}
\]

- \( \text{tagged EBz }\{\text{cond:.cv, tAddr:.av}\}: \)
  \[
  \begin{align*}
  &\text{if (cv != 0) then pc <= predIa;} \\
  &\text{else begin pc <= av;} \\
  &\text{Invalidat \( \text{buReg} \)}
  \end{align*}
\]

endcase

What does this mean?

Issues with \( \text{buReg} \)

- \( \text{buReg} \) may not always contain an instruction.
  
  Why?
  - start cycle
  - Execute stage may kill the fetched instructions because of branch misprediction

  Maybe type to the rescue ...

- Can’t update \( \text{buReg} \) in two concurrent actions

fetchAction; executeAction

Fold them together
Inelastic Pipeline first attempt

rule SyncTwoStage (True);
  let instr = iMem.read(pc);
  let predIa = pc+1;

  Action fetchAction =
    action
      buReg <= Valid newIt(instr);
      pc <= predIa;
    endaction;

  case (buReg) matches
    each instruction execution calls fetchAction
    or puts Invalid in buReg ...
  endcase
endcase endrule

Execute

  case (buReg) matches
    tagged Valid .it:
      case (it) matches
        tagged EAdd{dst:.rd,op1:.va,op2:.vb}: begin
          rf.upd(rd, va+vb); fetchAction; end
        tagged ELoad{dst:.rd,addr:.av}: begin
          rf.upd(rd, dMem.read(av)); fetchAction; end
        tagged EStore{val:.vv,addr:.av}: begin
          dMem.write(av, vv); fetchAction; end
        tagged EBz {cond:.cv,tAddr:.av}:
          if (cv != 0) then fetchAction;
          else begin pc <= av; buReg <= Invalid; end
      endcase
    tagged Invalid: fetchAction;
  endcase
Pipeline Hazards

Stall condition

Suppose the fetched instruction needs to read register \( r \) and the instruction in \( \text{buReg} \) is going to write in \( r \) then the Fetch unit must stall.

A function to find register \( r \) in an instruction template \( \text{it} \):

```
function Bool findf (RName r, InstrTemplate it);
    case (it) matches
        tagged EAdd{dst:.rd,op1:.v1,op2:.v2}:
            return (r == rd);
        tagged EBz {cond:.c,tAddr:.a}:
            return (False);
        tagged ELoad{dst:.rd,addr:.a}:
            return (r == rd);
        tagged EStore{val:.v,addr:.a}:
            return (False);
    endcase endfunction
```
The Stall Function
Decides if instruction \texttt{instr} should stall given the state of the \texttt{buReg}

```plaintext
function Bool stallFunc (Instr instr, Maybe#(InstTemplate) mit);
    case (mit) matches
        tagged Invalid: return False;
        tagged Valid .it:
            case (instr) matches
                tagged Add {dst:.rd,src1:.ra,src2:.rb}:
                    return (findf(ra,it) || findf(rb,it));
                tagged Bz {condR:.rc,addrR:.addr}:
                    return (findf(rc,it) || findf(addr,it));
                tagged Load {dst:.rd,addrR:.addr}:
                    return (findf(addr,it));
                tagged Store {valueR:.v,addrR:.addr}:
                    return (findf(v,it) || findf(addr,it));
            endcase
        endcase
    endfunction
```

Inelastic Pipeline

```plaintext
rule SyncTwoStage (True);
    let instr = iMem.read(pc);
    let predIa = pc+1;

    Action fetchAction =
        action
            if stallFunc(instr, buReg) then buReg <=Invalid
            else begin
                buReg <= Valid newIt(instr);
                pc <= predIa; end
        endaction;

case (buReg) matches
    The execute rule (no change)
endcase
endcase endrule
```
Bypassing

- After decoding the newIt function must read the new register values if available (i.e., the values that are still to be committed in the register file)

- We pass the value being written to decoding action (the newIt function)

Generation of bypass register value

```
rule inelasticProcessor2 (True);
case (buReg) matches
tagged Valid .it:
case (it) matches
tagged EAdd{dst:.rd,op1:.va,op2:.vb}: begin
  rf.upd(rd, va+vb); fetchAction; end
tagged ELoad{dst:.rd,addr:.av}: begin
  rf.upd(rd, dMem.read(av)); fetchAction; end
tagged EStore{val:.vv,addr:.av}: begin
  dMem.write(av, vv); fetchAction; end
tagged EBz {cond:.cv,tAddr:.av}:
  if (cv != 0) then fetchAction;
  else begin pc <= av; buReg <= Invalid; end
endcase
tagged Invalid: fetchAction;
endcase endrule
```
Bypassing values to Fetch

```plaintext
Bypassing values to Fetch

case (buReg) matches
  tagged Valid .it:
  case (it) matches
    tagged EAdd{dst:.rd,op1:.va,op2:.vb}: begin
      rf.upd(rd, va+vb);
      fetchAction(Valid rd, va+vb); end
    tagged ELoad{dst:.rd,addr:.av}: begin
      rf.upd(rd, dMem.read(av));
      fetchAction(Valid rd, dMem.read(av)); end
    tagged EStore{val:.vv,addr:.av}: begin
      dMem.write(av, vv);
      fetchAction(Invalid, ?); end
    tagged EBz {cond:.cv,tAddr:.av}:
      if (cv != 0) then fetchAction(Invalid, ?);
      else begin pc <= av; buReg <= Invalid; end
  endcase
  tagged Invalid: fetchAction(Invalid, ?);
endcase
```

New fetchAction

```plaintext
function Action fetchAction(Maybe#(RName) mrd,
                           Value val);
  action
    if stallFunc(instr, buReg) then
      buReg <= Invalid;
    else begin
      buReg <= Valid newIt(mrd, val, instr);
      pc <= predIa; end
  endaction
endfunction
```
Updated newIt

```haskell
function InstrTemplate newIt(Maybe#(RName) mrd,
                                Value val,
                                Instr instr);
    let nrf(a)=(Valid a == mrd) ? val: rf.sub(a);
    case (instr) matches
        tagged Add {dst:.rd,src1:.ra,src2:.rb}:
            return EAdd{dst:rd,op1:nrf(ra),op2:nrf(rb)};
        tagged Bz {condR:.rc,addrR:.addr}:
            return EBz{cond:nrf(rc),tAddr:nrf(addr)};
        tagged Load {dst:.rd,addrR:.addr}:
            return ELoad{dst:rd,addr:nrf(addr)};
        tagged Store{valueR:.v,addrR:.addr}:
            return EStore{val:nrf(v),addr:nrf(addr)};
    endcase
endfunction
```

Bypassing

- Now that we’ve correctly bypassed data, we do not have to stall as often

- The current stall function is correct, but inefficient
  - Should not stall if the value is now being bypassed
The stall function for the Inelastic pipeline

```haskell
function Bool newStallFunc (Instr instr,
                       Reg#(Maybe#(InstTemplate)) buReg);

    return (false);
```

Previously we stalled when ra matched the destination register of the instruction in the execute stage. Now we bypass that information when we read, so no stall is necessary.

Inelastic Pipelines

- Notoriously difficult to get right
  - Imagine the cases to be analyzed if it was a five stage pipeline

- Difficult to refine for better clock timing