Modeling Processors

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Instruction set

```
typedef enum {R0;R1;R2;…;R31} RName;

typedef union tagged {
  struct {RName dst; RName src1; RName src2;} Add;
  struct {RName cond; RName addr;}            Bz;
  struct {RName dst; RName addr;}             Load;
  struct {RName value; RName addr;}           Store
}
  Instr 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

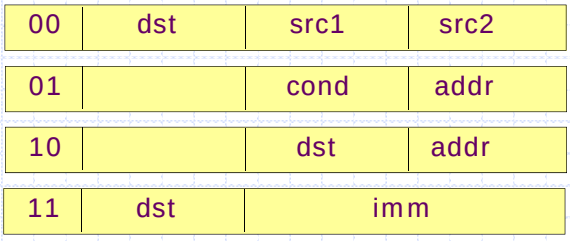
Deriving Bits

```
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 cond; RName addr;}               Bz;
  struct {RName dst; RName addr;}              Load;
  struct {RName dst; Immediate imm;}         AddImm;
} Instr deriving(Bits, Eq);
```



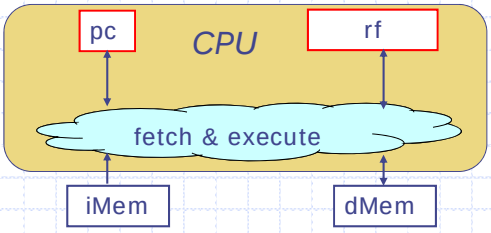
Automatically derived representation; can be customized by the user written pack and unpack functions

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




```
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

```
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
```

my syntax
rf[r] ≡ rf.sub(r)

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

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_3	FD_4	FD_5			
EXstage			EX_1	EX_2	EX_3	EX_4	EX_5		

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 cond; RName addr}            Bz;
  struct {RName dst; RName addr}             Load;
  struct {RName value; RName addr}           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 value; Daddress addr}      EStore;
} InstTemplate deriving(Eq, Bits);
```


Fetch & Decode Action

Fills 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  {cond:.rc,addr:.addr}:
        return EBz{cond:rf[rc],addr:rf[addr]};
     tagged Load {dst:.rd,addr:.addr}:
        return ELoad{dst:rd,addr:rf[addr]};
     tagged Store{value:.v,addr:.addr}:
        return EStore{value:rf[v],addr:rf[addr]};
  endcase
endfunction
```

Execute Action: *Reads* **buReg** *and modifies state* **(rf,dMem,pc)**

```
case (buReg) matches
   tagged EAdd{dst:.rd,src1:.va,src2:.vb}:
      begin rf.upd(rd, va+vb);
            pc <= predIa; end
   tagged ELoad{dst:.rd,addr:.av}:
      begin rf.upd(rd, dMem.read(av));
            pc <= predIa; end
   tagged EStore{value:.vv,addr:.av}:
      begin dMem.write(av, vv);
            pc <= predIa; end
   tagged EBz {cond:.cv,addr:.av}:
      if (cv != 0) then pc <= predIa;
      else begin pc <= av;
                 Invalidate buReg
           end
endcase
```

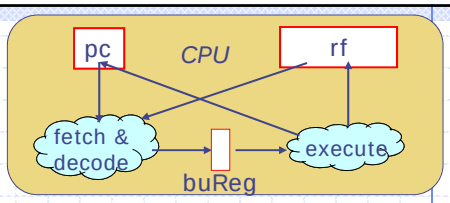
What does this mean?

Issues with **buReg**

- **buReg** may not always contain an instruction. Why?
  - start cycle
  - Execute stage may kill the fetched instructions because of branch misprediction
- Can't update **buReg** in two concurrent actions

`fetchAction; executeAction`

Inelastic Pipeline *first attempt*



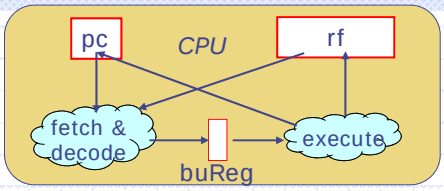

```
rule SyncTwoStage (True);
  let instr = iMem.read(pc);
  let predIa = pc+1;

  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,src1:.va,src2:.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{value:.vv,addr:.av}: begin
         dMem.write(av, vv); fetchAction; end
      tagged EBz {cond:.cv,addr:.av}:
         if (cv != 0) then fetchAction;
         else begin pc <= av; buReg <= Invalid; end
     endcase
  tagged Invalid: fetchAction;
endcase
```

Pipeline Hazards

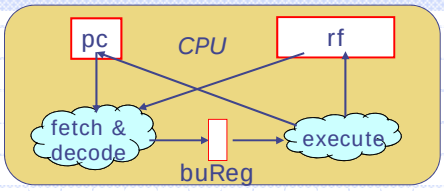
time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_3	FD_4	FD_5			
EXstage			EX_1	EX_2	EX_3	EX_4	EX_5		

I_1 Add(R1,R2,R3)
I_2 Add(R4,R1,R2)

I_2 must be stalled until I_1 updates the register file

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_2	FD_3	FD_4	FD_5		
EXstage			EX_1		EX_2	EX_3	EX_4	EX_5	

Inelastic Pipeline *corrected*




```
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
    … no change …
  endcase
endcase endrule
```

How do we detect stalls?

The Stall Function

```
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    {cond:.rc,addr:.addr}:
            return (findf(rc,it) || findf(addr,it));
          tagged Load  {dst:.rd,addr:.addr}:
            return (findf(addr,it));
          tagged Store {value:.v,addr:.addr}:
       return (findf(v,it) || findf(addr,it));
  endcase
endfunction
```


The findf function

```
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,addr:.a}:
                                return (False);
    tagged ELoad{dst:.rd,addr:.a}:
                                return (r == rd);
    tagged EStore{value:.v,addr:.a}:
                                return (False);
  endcase endfunction
```

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)

Updated newIt

```
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  {cond:.rc,addr:.addr}:
        return EBz{cond:nrf(rc),addr:nrf(addr)};
     tagged Load {dst:.rd,addr:.addr}:
        return ELoad{dst:rd,addr:nrf(addr)};
     tagged Store{value:.v,addr:.addr}:
        return EStore{value:nrf(v),addr:nrf(addr)};
  endcase
endfunction
```

New fetchAction

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


New Execute Action

```
case (buReg) matches
  tagged Valid .it:
     case (it) matches
      tagged EAdd{dst:.rd,src1:.va,src2:.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{value:.vv,addr:.av}: begin
         dMem.write(av, vv);
         fetchAction(Invalid, ?); end
      tagged EBz {cond:.cv,addr:.av}:
         if (cv != 0) then fetchAction(Invalid, ?);
         else begin pc <= av; buReg <= Invalid; end
     endcase
  tagged Invalid: fetchAction(Invalid, ?);
endcase
```

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

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

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

elastic pipelines