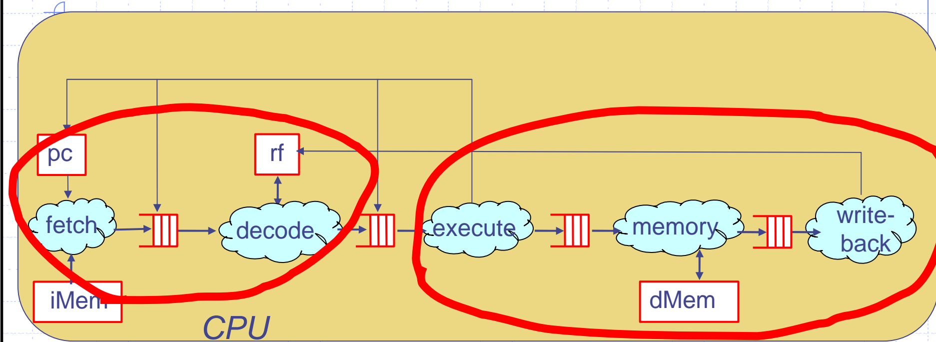
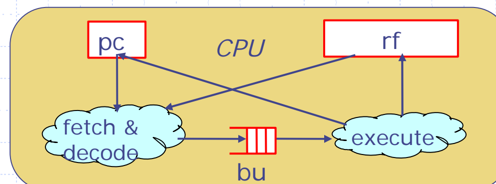
Modular Refinement - 2

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

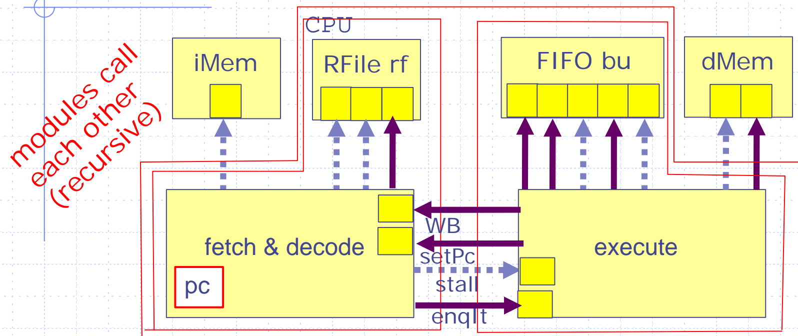
Successive refinement & Modular Structure



Can we derive the 5-stage pipeline by successive refinement of a 2-stage pipeline?



A Modular organization



- Suppose we include rf and pc in Fetch and bu in Execute
- Fetch delivers decoded instructions to Execute and needs to consult Execute for the stall condition
- Execute writes back data in rf and supplies the pc value in case of a branch misprediction

Recursive modular organization

```
module mkCPU2#(Mem iMem, Mem dMem)();
  Execute  execute  <- mkExecute(dMem, fetch);
  Fetch fetch  <- mkFetch(iMem, execute);
endmodule
```

recursive calls

```
interface Fetch;
  method Action setPC (Iaddress cpc);
  method Action writeback (RName dst, Value v);
endinterface

interface Execute;
  method Action enqIt(InstTemplate it);
  method Bool stall(Instr instr)
endinterface
```

Unfortunately, recursive module syntax is not as simple

Fetch Module

```
module mkFetch#(IMem iMem, Execute execute) (Fetch);
   Instr     instr  = iMem.read(pc);
   Iaddress predIa = pc + 1;

   Reg#(Iaddress) pc <- mkReg(0);
   RegFile#(RName, Bit#(32)) rf <- mkBypassRegFile();

   rule fetch_and_decode (!execute.stall(instr));
        execute.enqIt(newIt(instr,rf));
        pc <= predIa;
   endrule

   method Action writeback(RName rd, Value v);
        rf.upd(rd,v);
   endmethod
   method Action setPC(Iaddress newPC);
        pc <= newPC;
   endmethod
endmodule
```


Execute Module

```
module mkExecute#(DMem dMem, Fetch fetch) (Execute);

   SFIFO#(InstTemplate) bu <- mkSLoopyFifo(findf);
   InstTemplate it  = bu.first;

   rule execute …

   method Action enqIt(InstTemplate it);
        bu.enq(it);
   endmethod
   method Bool stall(Instr instr);
        return (stallFunc(instr, bu));
   endmethod
endmodule
```

Execute Module Rule

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
             fetch.writeback(rd, va+vb); bu.deq();
       end
    tagged EBz {cond:.cv,addr:.av}:
       if (cv == 0) then begin
             fetch.setPC(av); bu.clear(); end
       else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
             fetch.writeback(rd, dMem.read(av)); bu.deq();
       end
    tagged EStore{value:.vv,addr:.av}: begin
             dMem.write(av, vv); bu.deq();
       end
  endcase
endrule
```


Fetch Module Refinement

Separating Fetch and Decode

```
module mkFetch#(IMem iMem, Execute execute) (Fetch);
  FIFO#(Instr) buD <- mkLoopyFIFO();
  Instr    instr  = iMem.read(pc);
  Iaddress predIa = pc + 1;

  rule fetch(True);
    pc <= predIa; buD.enq(instr);
  endrule
  rule decode(!execute.stall(buD.first()));
       execute.enqIt(newIt(buD.first(),rf)); buD.deq();
   endrule

   method Action writeback(RName rd, Value v);
       rf.upd(rd,v);
   endmethod
   method Action setPC(Iaddress newPC);
     pc <= newPC; buD.clear();
   endmethod
endmodule
```

Execute Module Refinement

```
module mkExecute#(DMem dMem, Fetch fetch) (Execute);

   SFIFO#(InstTemplate) bu <- mkSLoopyFifo(findf);
   InstTemplate it  = bu.first;

   rule execute …

   method Action enqIt(InstTemplate it);
       bu.enq(it);
   endmethod
   method Bool stall(Instr instr);
       return (stallFunc(instr, bu));
   endmethod
endmodule
```


Execute Module Refinement

```
module mkExecute#(DMem dMem, Fetch fetch) (Execute);
   SFIFO#(InstTemplate)  buE <- mkSLoopyFifo(findfE);

   SFIFO#(InstTemplateM) buM <- mkSLoopyFIFO(findfM);
   SFIFO#(InstTemplateC) buC <- mkSLoopyFIFO(findfC);
   rule execute …
   rule memory …
   rule commit …
   method Action enqIt(InstTemplate it);
       buE.enq(it);
   endmethod
   method Bool stall(Instr instr);
     return(stallFuncE(instr, buE) &&
            stallFuncM(instr, buM) &&
            stallFuncC(instr, buC));
   endmethod
endmodule
```

New Types for Instruction Templates

```
typedef union tagged {
  struct {RName dst; Value val;}        MWB;
  void                                  MNop;
  struct {RName dst; IAddress addr;}    MLoad;
  struct {IAddress addr; Value val}     MStore;
}
  InstTemplateM deriving(Bits, Eq);

typedef union tagged {
  struct {RName dst; Value val;}        CWB;
  void                                  CNop;
}
  InstTemplateC deriving(Bits, Eq);
```


The New Execute Rule

Does not do mem-ops and writebacks

```
rule execute(True);
 case (buE.first()) matches
  tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
    buM.enq(tagged MWB{dst:rd, val:va+vb});
                 buE.deq();end
  tagged EBz {cond:0,addr:.av}:
    begin fetch.setPC(av); buE.clear();
                 buM.enq(tagged MNop{});end
  tagged EBz {cond:.cv,addr:.av}:
    begin buM.enq(tagged MNop{});
                 buE.deq(); end
  tagged ELoad{dst:.rd,addr:.av}:
    begin buM.enq(tagged MLoad{dst:rd,addr:av});
                 buE.deq();end
  tagged EStore{value:.vv,addr:.av}:
    begin buM.enq(tagged MStore{addr:av,val:vv});
                 buE.deq();end
 endcase endrule
```

The Memory Rule

```
rule memory(True);
 case (buM.first()) matches
   tagged MWB{dst:.rd,val:.v}:
      begin buC.enq(tagged CWB{dst:rd, val:v});
            buM.deq(); end
   tagged MNop {}:
      begin buC.enq(tagged CNop);
            buM.deq(); end
   tagged MLoad{dst:.rd,addr:.av}:
     begin buC.enq(tagged CWB{dst:rd,
                             val:dMem.read(av)});
           buM.deq();end
   tagged MStore{addr:.av, val:vv}}:
     begin buC.enq(tagged CNop{});
           dMem.write(av,vv); buM.deq();end
 endcase
endrule
```

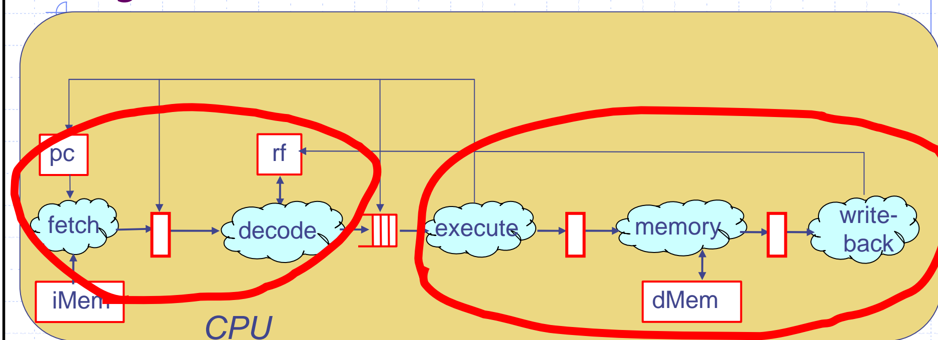

The Commit Rule

```
rule commit(True);
 case (buC.first()) matches
  tagged CWB{dst:.rd,val:.v}:
     fetch.writeback(rd,v);
  tagged CNop {}: noAction;
 endcase
 buC.deq();
endrule
```

Stall Functions

- Same as before, save variation in types in Queue.

Synchronous Refinements



Realistic processor Pipelines often have synchronous pieces connected asynchronously, i.e., using FIFOs

Sync. Execute Module

```
module mkExecute#(DMem dMem, Fetch fetch) (Execute);
   SFIFO#(InstTemplate) buE <- mkSLoopyFifo(findf);
   Reg#(Maybe#(InstTemplateM) buRegM <- mkReg(Invalid);
   Reg#(Maybe#(InstTemplateC) buRegC <- mkReg(Invalid);

    rule executePipe …

   method Action enqIt(InstTemplate it);
       buE.enq(it);
   endmethod
   method Bool stall(Instr instr);
      return(stallFuncE(instr, buE) &&
             stallFuncM(instr, buRegM) &&
             stallFuncC(instr, buRegC));
   endmethod
endmodule
```


Sync. Execute Pipeline

From Lecture 5

```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg1 <= Valid f1(inQ.first()); inQ.deq(); end
  else  sReg1 <= Invalid;
 case (sReg1) matches
  tagged Valid .sx1: sReg2 <= Valid f2(sx1);
  tagged Invalid:    sReg2 <= Invalid; endcase
 case (sReg2) matches
  tagged Valid .sx2: outQ.enq(f3(sx2)); endcase
endrule
```

- Use the following Mapping:
  - inQ -> buE
  - sReg1 -> buRegM
  - sReg2 -> buRegC
  - outQ.enq -> fetch.writeback

Sync. Execute Pipeline

Exec Action / Mem Action / Commit Action

```
rule executePipe (True);
 if (buE.notEmpty())
  begin buRegM <= Valid f1(buE.first()); buE.deq(); end
  else  buRegM <= Invalid;
 case (buRegM) matches
  tagged Valid .sx1: buRegC <= Valid f2(sx1);
  tagged Invalid:    buRegC <= Invalid; endcase
 case (buRegC) matches
  tagged Valid .sx2: fetch.writeback(f3(sx2)); endcase
endrule
```

- In Processor pipelines, the stage functions require additional side effects (actions) e.g. setPC/dMem.write/etc.
- For clarity, we will show these actions one by one

Execute Action

```
if (buE.notEmpty())
 begin buRegM <= Valid f1(buE.first()); buE.deq(); end
 else  buRegM <= Invalid;
```

```
case (buE.first()) matches
  tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
    buRegM <= Valid MWB{dst:rd, val:va+vb}; buE.deq();end
  tagged EBz {cond:0,addr:.av}:
    begin fetch.setPC(av); buE.clear();
          buRegM <= Valid MNop{};end
  tagged EBz {cond:.cv,addr:.av}:
    begin buRegM <= Valid MNop{}; buE.deq(); end
  tagged ELoad{dst:.rd,addr:.av}:
    begin buRegM <= Valid MLoad{dst:rd,addr:av}; buE.deq();end
  tagged EStore{value:.vv,addr:.av}:
    begin buRegM <= Valid MStore{addr:av,val:vv}; buE.deq();end
 endcase
```

"Extra effects"

Memory Action

```
case (buRegM) matches
 tagged Valid .itM: buRegC <= Valid f2(itM);
 tagged Invalid:    buRegC <= Invalid;
```

```
case (itM) matches
  tagged MWB{dst:.rd,val:.v}:
    buRegC <= tagged CWB{dst:rd, val:v};
  tagged MNop {}:
    buRegC <= tagged CNop {};
  tagged MLoad{dst:.rd,addr:.av}:
    buRegC <= tagged CWB{dst:rd, val:dMem.read(av)};
  tagged MStore{addr:.av, val:vv}}:
    begin buRegC <= tagged CNop{};
          dMem.write(av,vv);end
 endcase
```

"Extra effects"

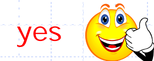
Commit Action

```
case (buRegC) matches
 tagged Valid .itC: fetch.writeback(f3(itC));
endcase
```

```
case (itC) matches
  tagged CWB{dst:.rd,val:.v}:
     fetch.writeback(rd,v);
  tagged CNop {}:
    noAction;
endcase
```

Sync. Execute Pipeline

```
rule executePipe (True);
    Execute Action;
    Memory Action;
    Commit Action;
endrule
```

yes 

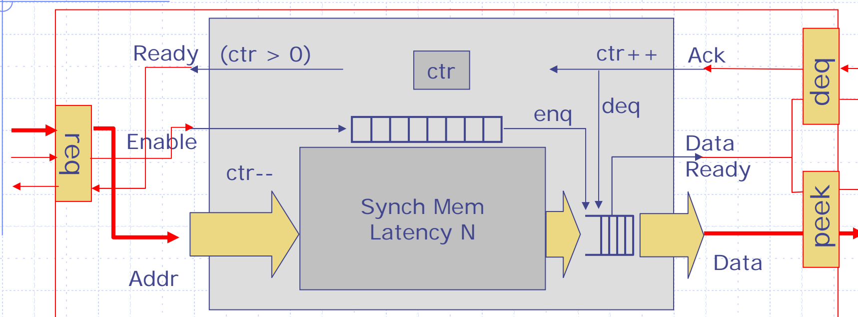
Refining the Memory Subsystem

Non-combinational memory reads

Realistic memory systems

- No combinational reads: The request to read the memory is a different event from the event when the response comes back
- The response to the request should be treated as an asynchronous event because its time can vary (cache misses)
- Request and response events or actions have to be dealt in different rules
  - iMem.read(pc) or dMem.read(addr) are no longer read methods – rules using them have to be split up

Request-Response Interface for Synchronous Memory




```
interface Mem#(type addrT, type dataT);
    method Action req(addrT x);
    method Action deq();
    method dataT peek();
endinterface
```

Making a synchronous component latency-insensitive

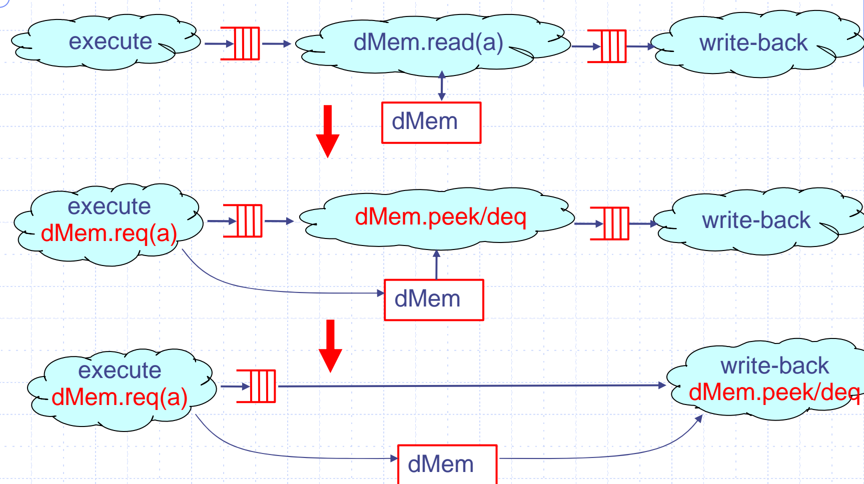
Request-Response Memory Interface

```
interface Mem#(type addrT, type dataT);
      method Action req(MemReq#(addrT,dataT) x);
      method Action deq();
      method dataT peek();
endinterface

typedef union tagged {
    struct {addrT addr; dataT val;} Write;
    struct {addrT addr;}            Read;
  } MemReq#(type addrT, type dataT)
        deriving(Bits, Eq);
```

We will assume that the memory processes read and write requests in the order received

Non-combinational reads



The Modified Execute Rule

```
rule execute(True);
 case (buE.first()) matches
  tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
    buC.enq(tagged CWB{dst:rd, val:va+vb});
                 buE.deq();end
  tagged EBz {cond:0,addr:.av}:
    begin fetch.setPC(av); buE.clear();
                 buC.enq(tagged CNop{});end
  tagged EBz {cond:.cv,addr:.av}:
    begin buC.enq(tagged CNop{});
                 buE.deq(); end
  tagged ELoad{dst:.rd,addr:.av}:
    begin buC.enq(tagged CLoad{dst:rd});
          dMem.req(Read{addr:av});
                 buE.deq();end
  tagged EStore{value:.vv,addr:.av}:
    begin buC.enq(tagged CNop{});
          dMem.req(Write{addr:av, val:vv});
                 buE.deq();end
 endcase endrule
```


The modified Commit Rule

```
rule commit(True);
 case (buC.first()) matches
  tagged CWB{dst:.rd,val:.v}:
     fetch.writeback(rd,v);
  tagged CNop {}: noAction;
  tagged CLoad{dst:.rd}:
   begin
     fetch.writeback(rd,dMem.peek());
     dMem.deq();
   end
 endcase
 buC.deq();
endrule
```

Fetch Module

```
module mkFetch#(IMem iMem, Execute execute) (Fetch);
  FIFO#(void) buD <- mkLoopyFIFO();
  Instr     instr   = iMem.peek();
  Iaddress predIa = pc + 1;
  …
  rule fetch(True);
    pc <= predIa;
    iMem.req(Read{addr:pc});
    buD.enq(void);
  endrule

   rule decode (!execute.stall(instr));
        execute.enqIt(newIt(instr,rf));
       iMem.deq();
        buD.deq();
   endrule
   method Action setPC …
   method Action writeback …
endmodule
```


Fetch Module Refinement

```
module mkFetch#(IMem iMem, Execute execute) (Fetch);
  FIFO#(void) buD <- mkFIFO();
  …
  Iaddress predIa = pc + 1;

  method Action writeback(RName rd, Value v);
      rf.upd(rd,v);
  endmethod
  method Action setPC(Iaddress newPC);
    pc <= newPC;
    buD.clear();
  endmethod
endmodule
```

Does this work?

Fetch Module

```
module mkFetch#(IMem iMem, Execute execute) (Fetch)
  Reg#(Bool) drainingIMem <- mkReg(False); …

  rule fetch(!drainingIMem);
    pc <= predIa; iMem.req(Read{addr:pc}); buD.enq(void);
  endrule

  rule decode(!drainingIMem && !execute.stall(instr));
    iMem.deq(); buD.deq(); execute.enqIt(newIt(instr,rf));
  endrule

  rule drain(drainingIMem);
    if(buD.notEmpty)begin iMem.deq(); buD.deq(); end
       else drainingIMem <= False;
  endrule

  method Action setPC(Iaddress newPC);
     pc <= newPC; drainingIMem <= buD.notEmpty();
  endmethod
  method Action writeback …
endmodule
```


Power of Modular Refinement

- Any Fetch-Decode module refinements we've shown, will work with each of the Execute module refinements we've shown
- Some refinements require interface changes (e.g. magic memory vs. realistic memory)
- The Correctness of refinements often depends on extra-linguistic properties (architectural reasoning)
  - It should be easy to add caches to new memory abstraction