Modeling Processors

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

The Plan

- Non-pipelined processor ⇐
- Two-stage synchronous pipeline
- Two-stage asynchronous pipeline

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

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

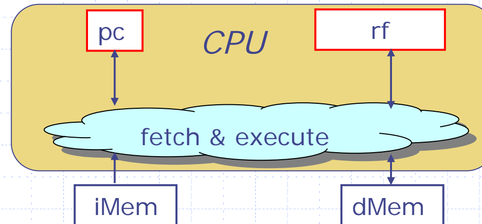
Tagged Unions: *Bit Representation*

```
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);
```

00	dst	src1	src2
01		cond	addr
10		dst	addr
11	dst	imm	

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

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
```

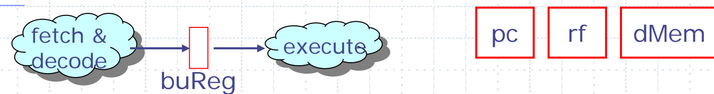
my syntax
rf[r] ≡ rf.sub(r)

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

The Plan

- Non-pipelined processor
- Two-stage synchronous pipeline ⇐
- Two-stage asynchronous pipeline

Two-stage Synchronous 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)

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 data; Daddress addr}       EStore;
} InstTemplate deriving(Eq, Bits);
```


Fetch & Decode Action

Fills the **buReg** *with a decoded instruction*

```
buReg <= newIt(instr);
```

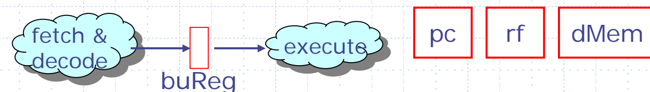
```
function InstrTemplate  newIt(Instr instr);
  case (instr) matches
     tagged Add {dst:.rd,src1:.ra,src2:.rb}:
        return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
     tagged Bz  {cond:.rc,addr:.addr}:
        return EBz{cond:rf[rc],addr:rf[addr]};
     tagged Load {dst:.rd,addr:.addr}:
        return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

no extra gates!

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;

           end
endcase
```


Issues with **buReg**

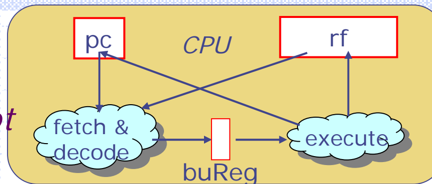


- **buReg** may not always contain an instruction. Why?
  -

- Can't update **buReg** in two concurrent actions

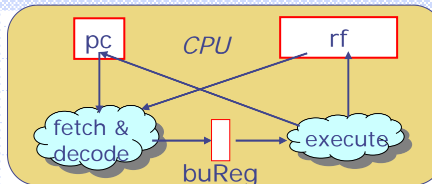
  fetchAction; executeAction

Synchronous Pipeline *first attempt*



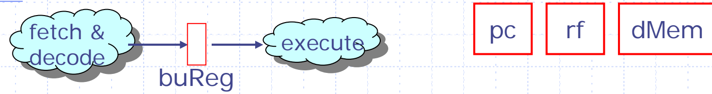

```
rule SyncTwoStage (True);
  let instr = iMem.read(pc);
  let predIa = pc+1;

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

  case (buReg) matches
    …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

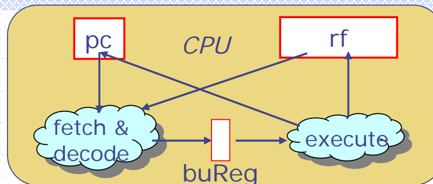


time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_3	FD_4	FD_5			
EXstage			EX_1	EX_2	EX_3	EX_4	EX_5		

I_1 Add(R1,R2,R3)

I_2 Add(R4,R1,R2)

Synchronous 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
```

Synchronous Pipelines

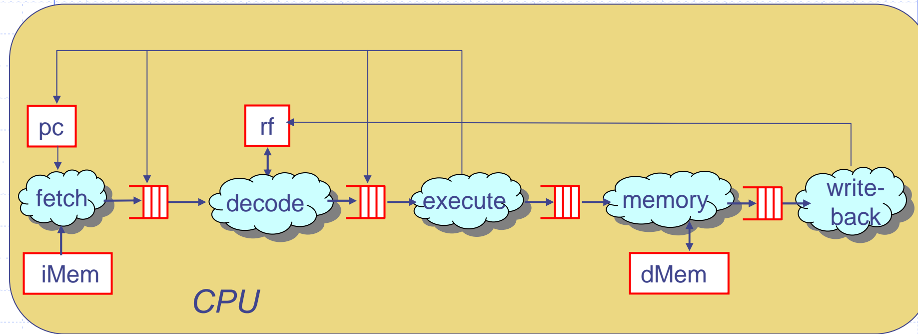
- Notoriously difficult to get right
  - Imagine the cases to be analyzed if it was a five stage pipeline
- Difficult to refine for better clock timing

Asynchronous pipelines

The Plan

- Non-pipelined processor
- Two-stage synchronous pipeline
- Two-stage asynchronous pipeline ⇐

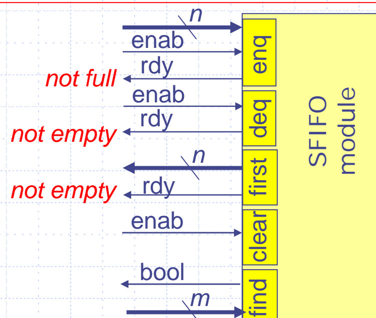
Processor Pipelines and FIFOs



It is better to think in terms of FIFOs as opposed to pipeline registers.

SFIFO (glue between stages)

```
interface  SFIFO#(type t, type tr);
  method Action enq(t);     // enqueue an item
  method Action deq();      // remove oldest entry
  method t first();         // inspect oldest item
  method Action clear();    // make FIFO empty
  method Bool find(tr);     // search FIFO
endinterface
```

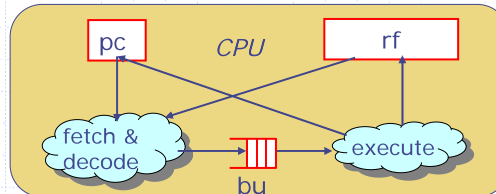


more on searchable FIFOs later

n = # of bits needed to represent the values of type "t"

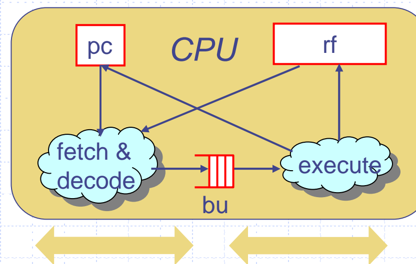
m = # of bits needed to represent the values of type "tr"

Two-Stage Pipeline




```
module mkCPU#(Mem iMem, Mem dMem)(Empty);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    SFIFO#(InstTemplate, RName) bu
                                      <- mkSFifo(findf);
    Instr     instr = iMem.read(pc);
    Iaddress predIa = pc + 1;
    InstTemplate it = bu.first();
    rule fetch_decode ...
endmodule
```

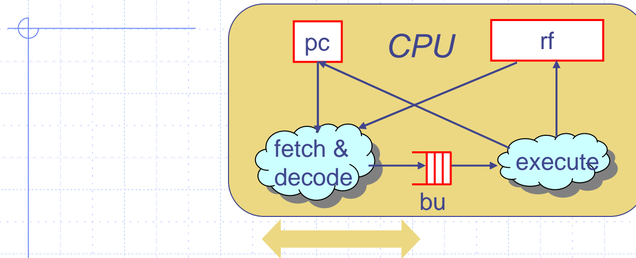

Rules for Add




```
rule decodeAdd(instr matches Add{dst:.rd,src1:.ra,src2:.rb})
  bu.enq (EAdd{dst:rd,op1:rf[ra],op2:rf[rb]});
  pc <= predIa;
endrule
```

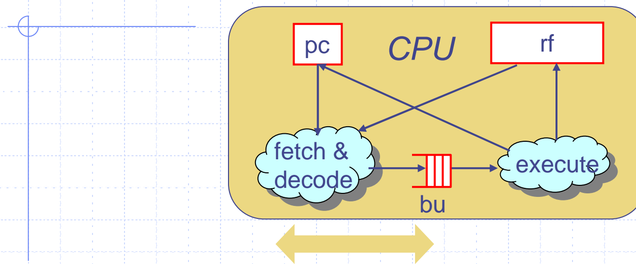
```
rule executeAdd(it matches EAdd{dst:.rd,op1:.va,op2:.vb})
  rf.upd(rd, va + vb);
  bu.deq();
endrule
```

Fetch & Decode Rule: *Reexamined*



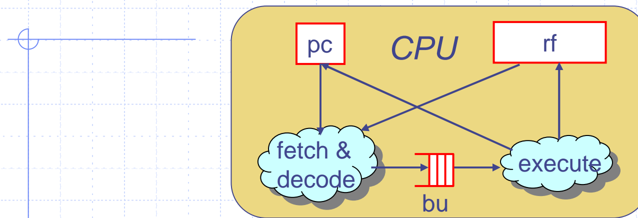

```
rule decodeAdd (instr matches Add{dst:.rd,src1:.ra,src2:.rb})

  bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});

  pc <= predIa;
endrule
```


Fetch & Decode Rule: *corrected*




```
rule decodeAdd (instr matches Add{dst:.rd,src1:.ra,src2:.rb}
     &&& !bu.find(ra) &&& !bu.find(rb))
  bu.enq (EAdd{dst:rd, op1:rf[ra], op2:rf[rb]});
  pc <= predIa;
endrule
```

Rules for Branch




```
rule decodeBz(instr matches Bz{cond:.rc,addr:.addr}) &&&
      !bu.find(rc) &&& !bu.find(addr));
     bu.enq (EBz{cond:rf[rc],addr:rf[addr]});
     pc <= predIa;
  endrule
```

```
rule bzTaken(it matches EBz{cond:.vc,addr:.va}) &&&
                                                (vc==0));
    pc <= va;   bu.clear(); endrule
rule bzNotTaken (it matches EBz{cond:.vc,addr:.va}) &&&
                                              (vc != 0));
    bu.deq; endrule
```


The Stall Signal

```
Bool stall  = stallFunc(instr, bu);
```

```
function Bool stallFunc (Instr instr,
         SFIFO#(InstTemplate, RName) bu);
   case (instr) matches
    tagged Add {dst:.rd,src1:.ra,src2:.rb}:
       return (bu.find(ra) || bu.find(rb));
    tagged Bz     {cond:.rc,addr:.addr}:
       return (bu.find(rc) || bu.find(addr));
    tagged Load   {dst:.rd,addr:.addr}:
       return (bu.find(addr));
    tagged Store  {value:.v,addr:.addr}:
       return (bu.find(v)) || bu.find(addr));
   endcase
endfunction
```

Need to extend the fifo interface with the "find" method where "find" searches the fifo to see if a register is going to be updated

The findf function

- When we make a searchable FIFO we need to supply a function that determines if a register is going to be updated by an instruction template
- mkSFifo can be parameterized by such a search function

```
SFIFO#(InstrTemplate, RName) bu <- mkSFifo(findf);
```

Same as before

```
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
```


Fetch & Decode Rule

```
rule fetch_and_decode (!stallFunc(instr, bu));
        bu.enq(newIt(instr));
        pc <= predIa;
endrule
```

Same as before

```
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 Rule

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


Next time -- Bypassing