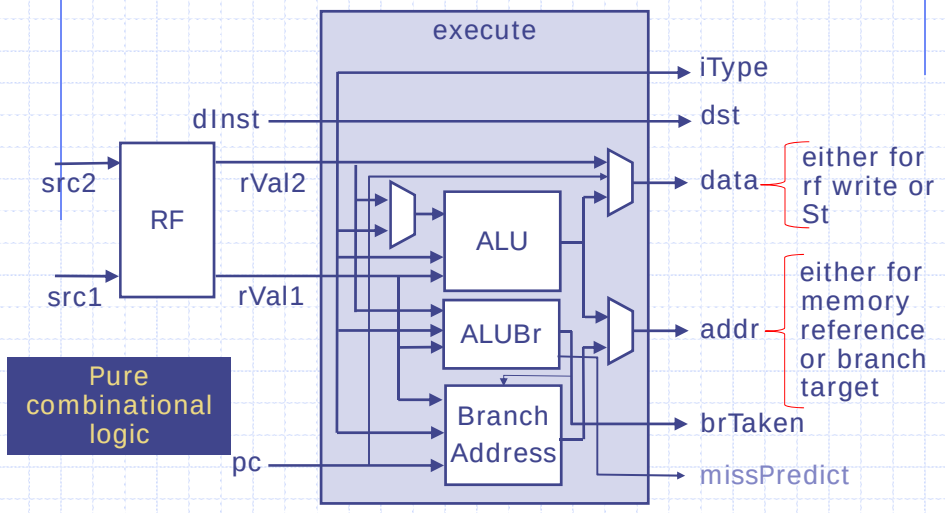
Constructive Computer Architecture:

Non-Pipelined Processors - 2

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Reading Registers and Executing Instructions



Output type of exec function

```
typedef struct {
  IType             iType;
  Maybe#(RIndx)     dst;
  Data              data;
  Addr              addr;
  Bool              mispredict;
  Bool              brTaken;
} ExecInst deriving(Bits, Eq);
```


Execute Function

```
function ExecInst exec(DecodedInst dInst, Data rVal1,
                       Data rVal2, Addr pc);
  ExecInst eInst = ?;
  Data aluVal2   =

  let aluRes     =
  eInst.iType    =
  eInst.data     =


  let brTaken    =
  let brAddr     =

  eInst.brTaken  =
  eInst.addr     =

  eInst.dst      =
  return eInst;
endfunction
```

Single-Cycle SMIPS *atomic state updates*

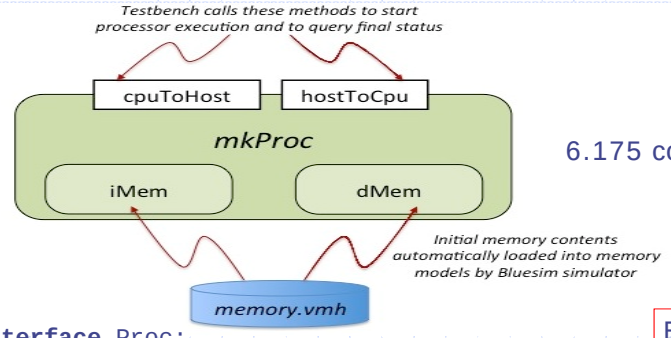
```
    if(eInst.iType == Ld)
      eInst.data <- dMem.req(MemReq{op: Ld,
                      addr: eInst.addr, data: ?});
    else if (eInst.iType == St)
      let dummy <- dMem.req(MemReq{op: St,
                      addr: eInst.addr, data: data});

    if(isValid(eInst.dst))
        rf.wr(fromMaybe(?, eInst.dst), eInst.data);

    pc <= eInst.brTaken ? eInst.addr : pc + 4;

endrule
endmodule
```


Processor interface



6.175 convention

```
interface Proc;
   method Action hostToCpu(Addr startpc);
   method ActionValue#(CpuToHost) cpuToHost;
endinterface
typedef struct {CpuToHostType c2hType; Bit#(16) data;}
CpuToHost deriving(Bits, Eq);
typedef enum {ExitCode, PrintChar, PrintIntLow,
PrintIntHigh} CpuToHostType deriving(Bits, Eq);
```

Exit code & stdout

Control and Status Registers (CSR)

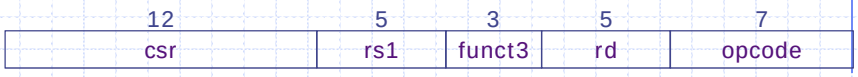
- CSRs are used to record and control the machine state
  - cycle (clock cycles) // read only
  - instret (instruction counts) // read only
  - hartid (hardware thread ID) // read only
  - mtohost (output to host) // write only
  - mepc, mcause etc. will be used for exception handleling later

```
typedef Bit#(12) CsrIndx; // CSR index is 12-bit
```

CSR is needed as an additional field in `DecodedInst` and `ExecInst` types

```
Maybe#(CsrIndex) csr;
```


Instructions to Read and Write CSR



- opcode = SYSTEM
- CSRW rs1, csr (funct3 = CSRRW, rd = x0): csr ← rs1
- CSRR csr, rd (funct3 = CSRRS, rs1 = x0): rd ← csr
- New enums in IType: `Csrr, Csrw`

Code with CSRs

```
// csrf: module that implements all CSRs
let csrVal = csrf.rd(fromMaybe(?, dInst.csr));
let eInst = exec(dInst, rVal1, rVal2, pc, csrVal);
```

pass CSR values to execute CSRR

```
csrf.wr(eInst.iType == Csrw ? eInst.csr : Invalid,
eInst.data);
```

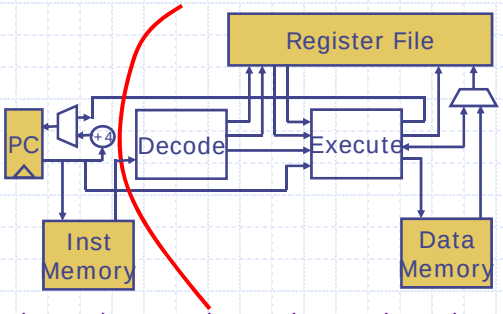
write CSR (CSRW instruction) and indicate the completion of an instruction

We did not show these lines in our processor to avoid cluttering the slides

Communicating with the host

- We will provide you C library functions like print and you will almost never encode anything directly to communicate with tthe host

Single-Cycle RISC-V: *Clock Speed*



$t_{Clock} > t_M + t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB}$

We can improve the clock speed if we execute each instruction in two clock cycles

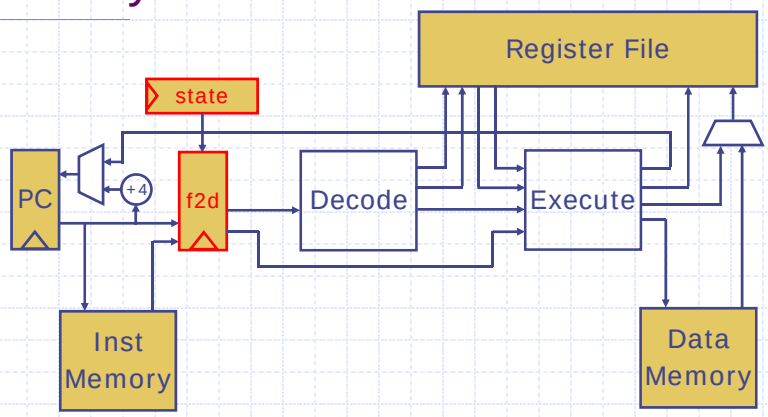
$t_{Clock} > \max \{t_M , (t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB})\}$

However, this may not improve the performance because each instruction will now take two cycles to execute

Structural Hazards

- Sometimes multicycle implementations are necessary because of resource conflicts, aka, *structural hazards*
  - Princeton style architectures use the same memory for instruction and data and consequently, require at least two cycles to execute Load/Store instructions
  - If the register file supported less than 2 reads and one write concurrently then most instructions would take more than one cycle to execute
- Usually extra registers are required to hold values between cycles

Two-Cycle RISC-V



Introduce register "f2d" to hold a fetched instruction and register "state" to remember the state (fetch/execute) of the processor

Two-Cycle RISC-V

```
module mkProc(Proc);
  Reg#(Addr)  pc <- mkRegU;
  RFile       rf <- mkRFile;
  IMemory     iMem <- mkIMemory;
  DMemory     dMem <- mkDMemory;
  Reg#(Data)  f2d <- mkRegU;
  Reg#(State) state <- mkReg(Fetch);

  rule doFetch (state == Fetch);
      let inst = iMem.req(pc);
      f2d <= inst;
      state <= Execute;
  endrule
```

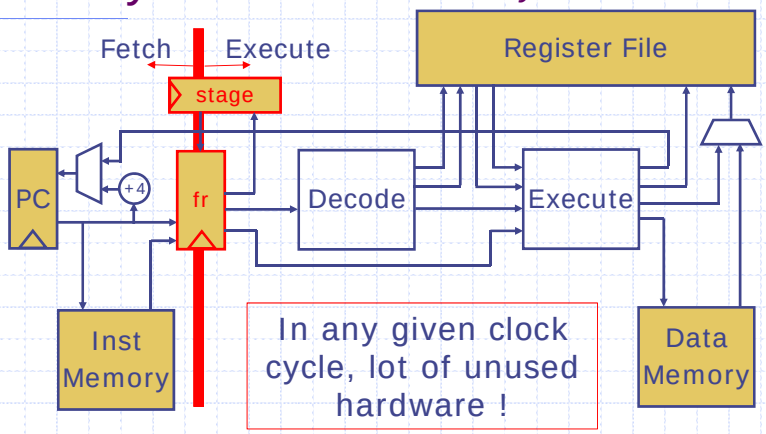
Two-Cycle RISC V

The Execute Cycle

```
rule doExecute(stage==Execute);
   let inst = f2d;
   let dInst = decode(inst);
   let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
   let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
   let eInst = exec(dInst, rVal1, rVal2, pc);
   if(eInst.iType == Ld)
      eInst.data <- dMem.req(MemReq{op: Ld, addr:
            eInst.addr, data: ?});
   else if(eInst.iType == St)
      let d <- dMem.req(MemReq{op: St, addr:
            eInst.addr, data: eInst.data});
   if (isValid(eInst.dst))
      rf.wr(fromMaybe(?, eInst.dst), eInst.data);
   pc <= eInst.brTaken ? eInst.addr : pc + 4;
   state <= Fetch;
endrule endmodule
```

no change from single-cycle

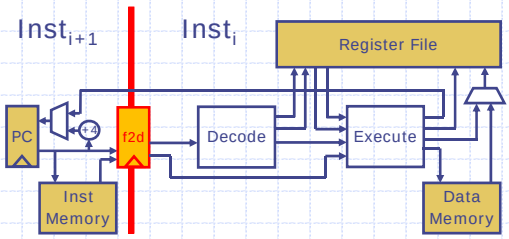
Two-Cycle RISC-V: *Analysis*



In any given clock cycle, lot of unused hardware !

Pipeline execution of instructions to increase the throughput

Problems in Instruction pipelining

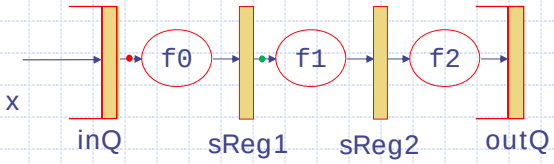


- *Control hazard:* Inst$_{i+1}$ is not known until Inst$_i$ is at least decoded. So which instruction should be fetched?
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory
- *Data hazard:* Inst$_i$ may affect the state of the machine (pc, rf, dMem) – Inst$_{i+1}$ must be fully cognizant of this change

none of these hazards were present in the FFT pipeline

Arithmetic versus Instruction pipelining

- The data items in an arithmetic pipeline, e.g., FFT, are independent of each other

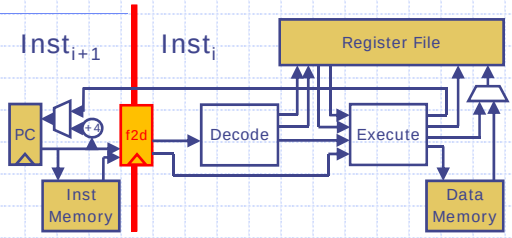


- The entities in an instruction pipeline affect each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls
  - Processor pipelines are significantly more complicated than arithmetic pipelines

The power of computers comes from the fact that the instructions in a program are *not* independent of each other

⇒ must deal with hazard

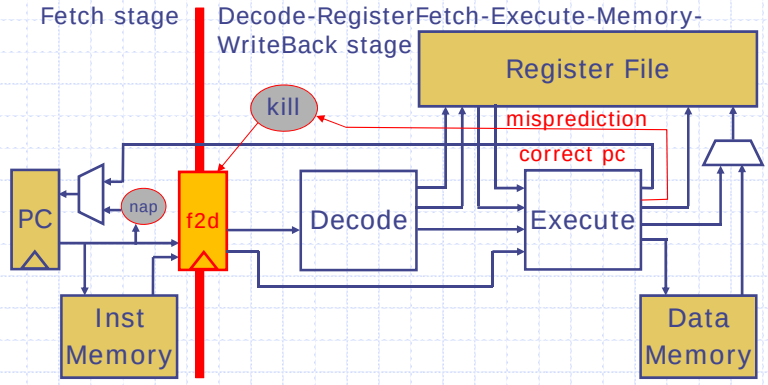
Control Hazards



Inst$_{i+1}$ is not known until Inst$_i$ is at least decoded. So which instruction should be fetched?

- General solution – *speculate*, i.e., predict the next instruction address
  - requires the next-instruction-address prediction machinery; can be as simple as pc+4
  - prediction machinery is usually elaborate because it dynamically learns from the past behavior of the program
- What if speculation goes wrong?
  - machinery to kill the wrong-path instructions, restore the correct processor state and restart the execution at the correct pc

Two-stage Pipelined SMIPS



Fetch stage must predict the next instruction to fetch to have any pipelining

In case of a misprediction the Execute stage must kill the mispredicted instruction in f2d

Pipelining Two-Cycle RISC-V

singlerule

```
rule doPipeline ;
  let newInst = iMem.req(pc);                                   fetch
  let newPpc = nap(pc); let newPc = ppc;
  let newIr=Valid(Fetch2Decode{pc:newPc,ppc:newPpc,
                               inst:newIinst});
  if(isValid(ir)) begin                                         execute
   let x = fromMaybe(?, ir); let irpc = x.pc;
   let ppc = x.ppc; let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch ...;
   let eInst = exec(dInst, rVal1, rVal2, irpc, ppc);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict) begin newIr = Invalid;
                               newPc = eInst.addr;  end
                  end
    pc <= newPc; ir <= newIr;
endrule
```