Constructive Computer Architecture:

Non-Pipelined Processors - 2

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Single-Cycle Implementation

code structure

module mkProc(Proc);

Reg#(Addr) pc <- mkRegU;
RFile rf <- mkRFile;
IMemory iMem <- mkIMemory;
DMemory dMem <- mkDMemory;

rule doProc;

let inst = iMem.req(pc);
let dInst = decode(inst);
let rVal1 = rf.rd1(dInst.rSrc1);
let rVal2 = rf.rd2(dInst.rSrc2);
let eInst = exec(dInst, rVal1, rVal2, pc);
update rf, pc and dMem

instantiate the state

exacts fields needed for execution

produces values needed to update the processor state
Single-Cycle RISC-V atomic state updates

```haskell
if(eInst.iType == Ld) // Load from memory
  eInst.data <- dMem.req(MemReq{op: Ld,
    addr: eInst.addr, data: ?});
else if (eInst.iType == St) // Store to memory
  let dummy <- dMem.req(MemReq{op: St,
    addr: eInst.addr, data: data});

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

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

The whole processor is described using one rule; lots of big combinational functions.
Processor interface

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);

communication is done via CSRs or memory
6.175 convention uses a CSR (mtohost) to communicate with the host

- 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

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

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

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

csr.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, which use CSR to communicate with the host; you will almost never encode anything directly to communicate with the host.
Single-Cycle RISC-V: Clock Speed

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

\[
t_{\text{Clock}} > t_M + t_{\text{DEC}} + t_{\text{RF}} + t_{\text{ALU}} + t_M + t_{\text{WB}}
\]

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
Introduce register “f2d” to hold a fetched instruction and register “state” to remember the state (fetch/execute) of the processor.
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
rule doExecute (stage==Execute);
    let inst = f2d;
    let dInst = decode(inst);
    ... Copy the code from slides 2 and 3 ...
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
    state <= Fetch;
endrule endmodule
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**: \( \text{Inst}_{i+1} \) is not known until \( \text{Inst}_i \) 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**: \( \text{Inst}_i \) may affect the state of the machine (pc, rf, dMem) – \( \text{Inst}_{i+1} \) 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.

In processors, older instructions in the pipeline may affect the younger ones:
- This causes pipeline stalls or requires other fancy tricks to avoid stalls.
- Processor pipelines are significantly more complicated than arithmetic pipelines.
Hazards can’t be wished away

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

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 RISC-V

- 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
Synchronous Pipeline, singlerule

**rule** doPipeline;

**Fetch phase** -
fetch an instruction to be put into register \( ir \); and
guess the next pc

**Execute phase** -
execute the instruction in \( ir \) if it has a valid one;
determine if the next pc is what we had guessed;
if the guess was correct then assign the newly fetched instruction and
the guessed pc into \( ir \) and \( pc \), respectively;
if the guess was incorrect then put Invalid in \( ir \) and the correct pc into
the pc

**endrule**
Pipelining Two-Cycle RISC-V synchronous pipeline, i.e., singlerule

rule doPipeline;

let newInst = iMem.req(pc);
let newPpc = nap(pc); let newPc = ppc;
let newIr = Valid(Fetch2Decode{pc:newPc, ppc:newPpc,
                        inst:newIinst});

if(isValid(ir)) begin
  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
Elastic two-stage pipeline

We replace f2d register by a FIFO to make the machine more elastic, that is, Fetch keeps putting instructions into f2d and Execute keeps removing and executing instructions from f2d.

Fetch passes the pc and predicted pc in addition to the inst to Execute; Execute redirects the PC in case of a miss-prediction.