Constructive Computer Architecture:

Non-Pipelined and Pipelined Processors

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

interface Proc;
    method Action hostToCpu(Addr startpc);
    method Action Value #(Tuple2#(RIndx, Data)) cpuToHost;
endinterface

Stream of register values from the CPU
Coprocessor Registers

MIPS allows extra sets of 32-registers each to support system calls, floating point, debugging etc. These registers are known as coprocessor registers

- The registers in the \( n \)th set are written and read using instructions \( \text{MTCn} \) and \( \text{MFCn} \), respectively
- Set 0 is used to get the results of program execution (Pass/Fail), the number of instructions executed and the cycle counts
- Type \text{FullIndx} is used to refer to the normal registers plus the coprocessor set 0 registers
- function \text{validRegValue} (\text{FullIndx} r) returns index of \( r \)

```plaintext
typedef Bit#(5) RIndx;
typedef enum {Normal, CopReg} RegType deriving (Bits, Eq);
typedef struct {RegType regType; RIndx idx;} FullIndx;
deriving (Bits, Eq);
```

October 1, 2014  
http://csg.csail.mit.edu/6.175  
L10-3
Code with coprocessor calls

```javascript
let copVal = cop.rd(validRegValue(dInst.src1));
let eInst = exec(dInst, rVal1, rVal2, pc, copVal);

pass coprocessor register values to execute MFC0

cop.wr(eInst.dst, eInst.data);

write coprocessor registers (MTC0) and indicate the completion of an instruction

We did not show these lines in our processor to avoid cluttering the slides
```
Single-Cycle SMIPS: 

*Clock Speed*

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

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

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
Two-Cycle SMIPS

Introduce register “f2d” to hold a fetched instruction and register “state” to remember the state (fetch/execute) of the processor
Two-Cycle SMIPS

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);
endmodule

rule doFetch (state == Fetch);
    let inst = iMem.req(pc);
    f2d <= inst;
    state <= Execute;
endrule
Two-Cycle SMIPS

```verilog
rule doExecute (stage == Execute);

let inst = f2d;

let dInst = decode (inst);
let rVal1 = rf.rd1 (validRegValue (dInst.src1));
let rVal2 = rf.rd2 (validRegValue (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(validRegValue(eInst.dst), eInst.data);
pc <= eInst.brTaken ? eInst.addr : pc + 4;
state <= Fetch;
endrule
```

no change from single-cycle
Two-Cycle SMIPS: 

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.

- 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.

\[\Rightarrow \text{must deal with hazard}\]
Control Hazards

Inst\textsubscript{i+1} is not known until Inst\textsubscript{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 SMIPS – singlerule

rule doPipeline;

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

if(isValid(ir)) begin
let x = validValue(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