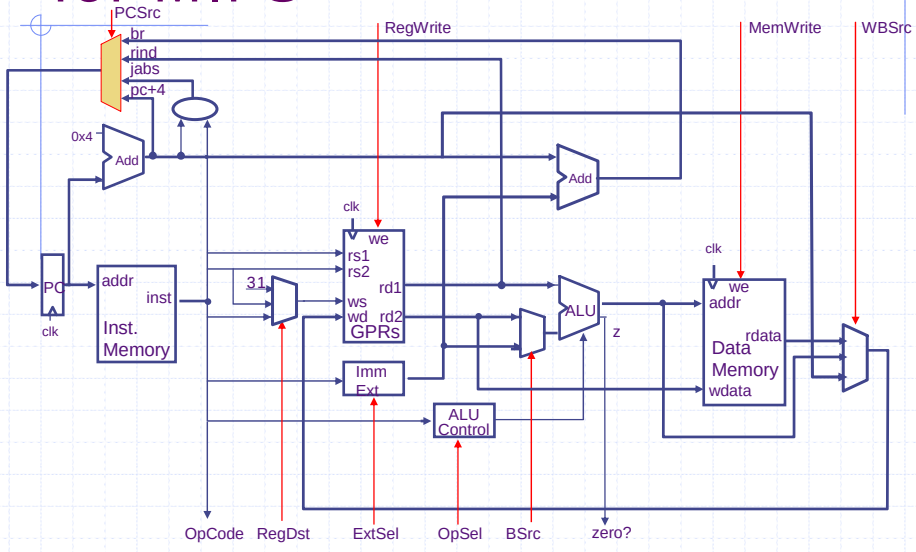
Computer Architecture: A Constructive Approach

Multi-cycle SMIPS Implementations

Joel Emer
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Harvard-Style Datapath for MIPS

old way



old way

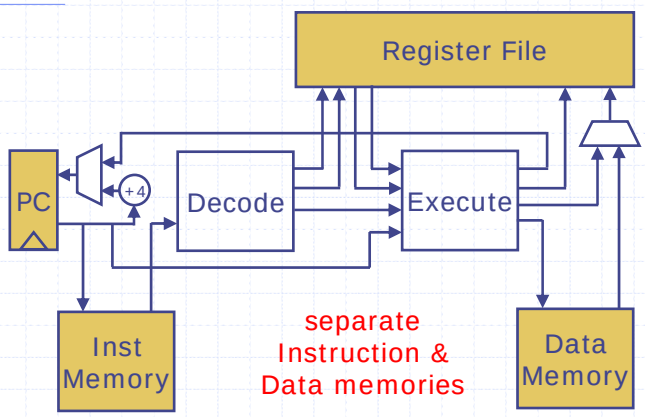
Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	$sExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
ALUiu	$uExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
LW	$sExt_{16}$	Imm	+	no	yes	Mem	rt	pc+4
SW	$sExt_{16}$	Imm	+	yes	no	*	*	pc+4
$BEQZ_{z=0}$	$sExt_{16}$	*	0?	no	no	*	*	br
$BEQZ_{z=1}$	$sExt_{16}$	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

BSrc = Reg / Imm WBSrc = ALU / Mem / PC
RegDst = rt / rd / R31 PCSrc = pc+4 / br / rind / jabs

new way

Single-Cycle SMIPS



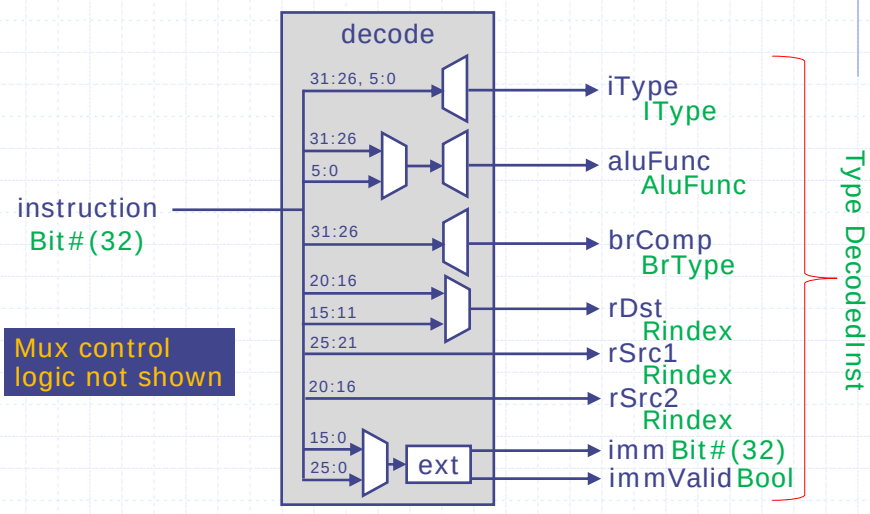
Datapath and control were derived automatically from a high-level rule-based description

Single-Cycle SMIPS *code structure*

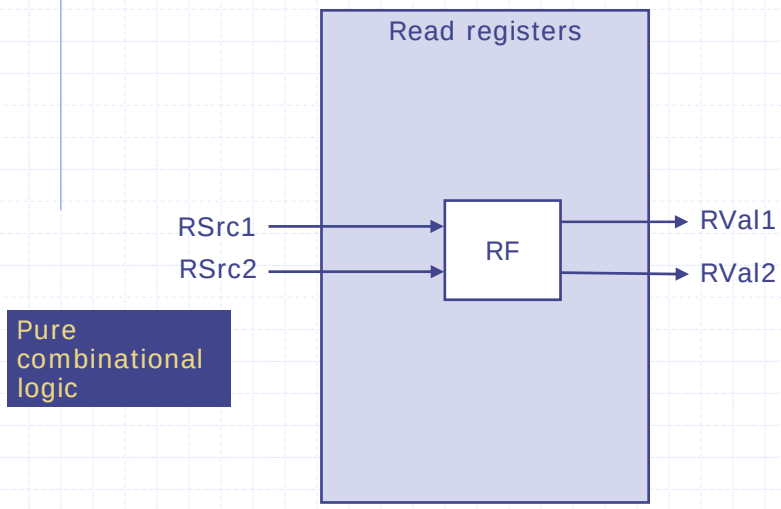
```
module mkProc(Proc);
  Reg#(Addr)  pc <- mkRegU;
  RFile       rf <- mkRFile;
  Memory     mem <- mkTwoPortedMemory;
  let iMem = mem.iport ; let dMem = mem.dport;

  rule doProc;
    let inst = iMem(MemReq{op:Ld, addr:pc, data:?});
    let dInst = decode(inst);
    Data rVal1 = rf.rd1(dInst.rSrc1);
    Data rVal2 = rf.rd2(dInst.rSrc2);
    let eInst = exec(dInst, rVal1, rVal2, pc);
```

update rf, pc and dMem

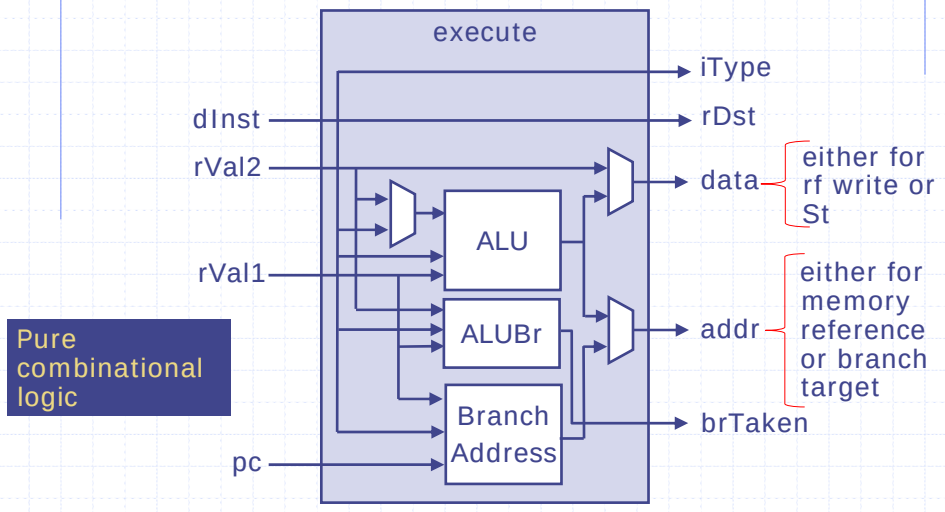
Decoding Instructions:
input-output types



Reading Registers



Executing Instructions



Branch Address Calculation

```
function Addr brAddrCalc(Address pc, Data val,
                         IType iType, Data imm);
 let targetAddr = case (iType)
                    J, Jal   : {pc[31:28], imm[27:0]};
                    Jr, Jalr : val;
                    default  : pc + imm;
                  endcase;
 return targetAddr;
endfunction
```


Some Useful Functions

```
function Bool memType (IType i)
  return (i==Ld || i == St);
endfunction

function Bool regWriteType (IType i)
 return (i==Alu || i==Ld || i==Jal || i==Jalr);
endfunction

function Bool controlType (IType i)
 return (i==J || i==Jr || i==Jal || i==Jalr || i==Br);
endfunction
```

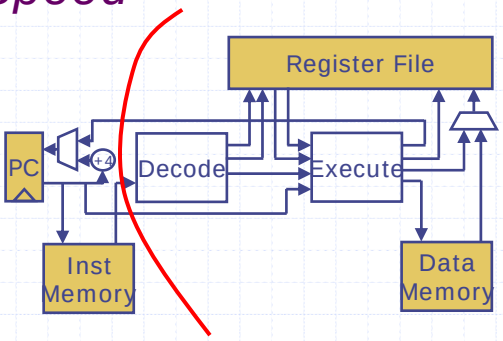
Execute Function

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

  Data aluVal2 = (dInst.immValid)? dInst.imm : rVal2
  let aluRes = alu(rVal1, aluVal2, dInst.aluFunc);
  let brAddr = brAddrCal(pc, rVal1, dInst.iType,
                         dInst.imm);
  einst.itype = dInst.iType;
  einst.addr = (memType(dInst.iType)? aluRes : brAddr;
  einst.data = dInst.iType==St ? rVal2 : aluRes;
  einst.brTaken = aluBr(rVal1, aluVal2, dInst.brComp);
  einst.rDst = dInst.rDst;
  return einst;
endfunction
```


Single-Cycle SMIPS *atomic state updates*

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

    if(regWriteType(eInst.iType))
      rf.wr(eInst.rDst, eInst.data);

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

  endrule
endmodule
```

Single-Cycle SMIPS: *Clock Speed*

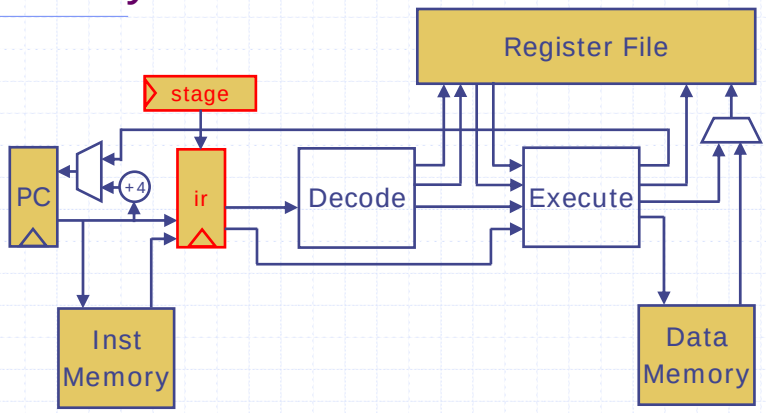


$t_{Clock} > t_M + t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB}$

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

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

Two-Cycle SMIPS



Introduce register "ir" to hold a fetched instruction and register "stage" to remember which stage (fetch/execute) we are in

ir: The instruction register

- You may recall from our earlier discussion of pipelining that when we take multiple cycles to perform some operation (e.g., IFFT), there is a possibility that intermediate registers do not contain any meaningful data in some cycles
- It is straight forward to convert ir into a pipeline register
  - We can associate (Valid/Invalid) bit with ir
  - Equivalently, we can think of ir as a single-element FIFO

Additional Types

```
typedef struct {
  Addr     pc;
  Bit#(32) inst;
} TypeFetch2Decode deriving (Bits, Eq);

typedef enum {Fetch, Execute}
        TypeStage deriving (Bits, Eq);
```

Two-Cycle SMIPS

```
module mkProc(Proc);
  Reg#(Addr)  pc <- mkRegU;
  RFile       rf <- mkRFile;
  Memory     mem <- mkTwoPortedMemory;
  let iMem = mem.iport;  let dMem = mem.dport;
  Reg#(TypeFetch2Decode)    ir <- mkRegU;
  Reg#(TypeStage)        stage <- mkReg(Fetch);

  rule doFetch (state==Fetch);
      let inst = iMem(MemReq{op:Ld, addr:pc, data:?});
      ir <= TypeFetch2Decode{pc: pc, inst: inst};
      stage <= Execute;
  endrule
```


Two-Cycle SMIPS

```
  rule doExecute(stage==Execute);
    let irpc  = ir.pc;
    let inst  = ir.inst;
    let dInst = decode(inst);
    Data rVal1 = rf.rd1(dInst.rSrc1);
    Data rVal2 = rf.rd2(dInst.rSrc2);
    let eInst = exec(dInst, rVal1, rVal2, irpc);
    if(memType(eInst.iType))
      eInst.data <- dMem(MemReq{
                op: eInst.iType==Ld ? Ld : St,
                addr: eInst.addr, data: eInst.data});
    if(regWriteType(eInst.iType))
      rf.wr(eInst.rDst, eInst.data);
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
    stage <= Fetch;
  endrule    endmodule
```

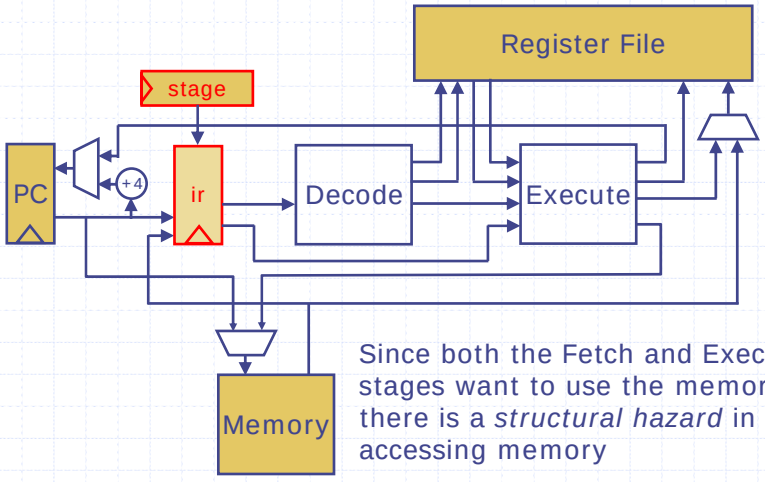
no change from single-cycle

Princeton versus Harvard Architecture

- Harvard architecture uses different memories for instructions and data
  - needed for a single-cycle implementation
- Princeton architecture uses the same memory for instruction and data and thus, requires at least two cycles to execute Load/Store instructions

The two-cycle implementations of Princeton and Harvard architectures are almost the same

SMIPS Princeton Architecture



Since both the Fetch and Execute stages want to use the memory, there is a *structural hazard* in accessing memory

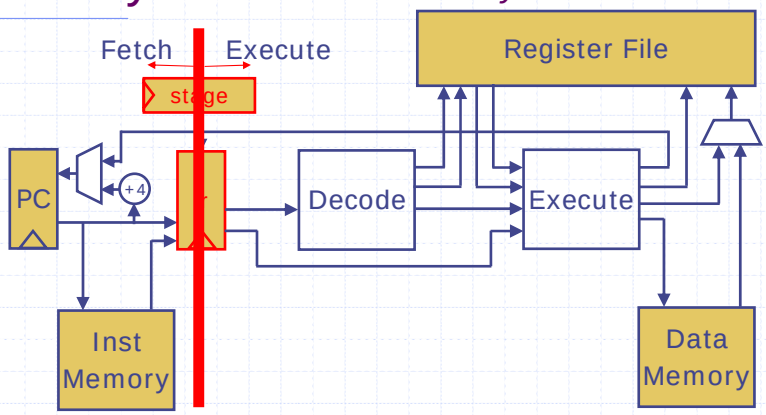
Two-Cycle SMIPS Princeton

```
module mkProc(Proc);
  Reg#(Addr)  pc <- mkRegU;
  RFile       rf <- mkRFile;
  Memory     mem <- mkOnePortedMemory;
  let uMem = mem.port;
  Reg#(TypeFetch2Decode)   ir <- mkRegU;
  Reg#(TypeStage)       stage <- mkReg(Fetch);

  rule doFetch (stage==Fetch);
      let inst <- uMem(MemReq{op:Ld, addr:pc, data:?});
      ir <= TypeFetch2Decode{pc: pc, inst: inst};
      stage <= Execute;
  endrule
```


Two-Cycle SMIPS Princeton

```
  rule doExecute(stage==Execute);
    let irpc  = ir.pc;
    let inst  = ir.inst;
    let dInst = decode(inst);
    Data rVal1 = rf.rd1(dInst.rSrc1);
    Data rVal2 = rf.rd2(dInst.rSrc2);
    let eInst = exec(dInst, rVal1, rVal2, irpc);
    if(memType(eInst.iType))
      eInst.data <- uMem(MemReq{
                op: eInst.iType==Ld ? Ld : St,
                addr: eInst.addr, data: eInst.data});
    if(regWriteType(eInst.iType))
      rf.wr(eInst.rDst, eInst.data);
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
    stage <= Fetch;
  endrule    endmodule
```

Two-Cycle SMIPS: *Analysis*



In any given clock cycle, lots of unused hardware!

next lecture: Pipelining to increase the throughput