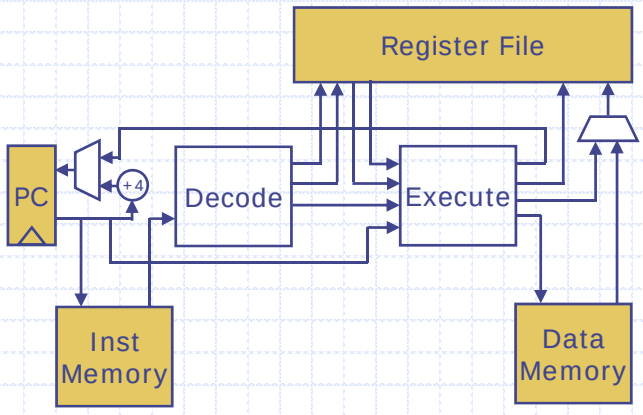
Non-Pipelined Processors

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

Single-Cycle RISC Processor

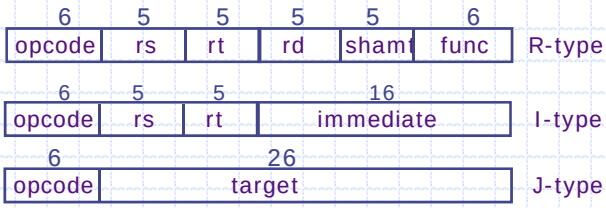


Single-Cycle Implementation

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
```

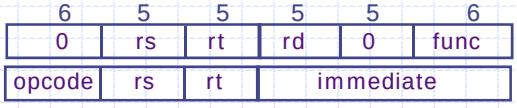

SMIPS Instruction formats



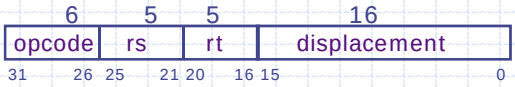
- Only three formats but the fields are used differently by different types of instructions

Instruction formats *cont*

- Computational Instructions



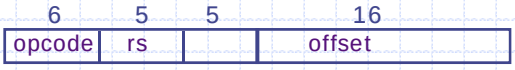
- Load/Store Instructions



rs is the base register
rt is the destination of a Load or the source for a Store

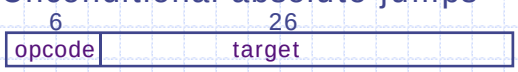
Control Instructions

- Conditional (on GPR) PC-relative branch



  - target address = (offset in words)×4 + (PC+4)
  - range: ±128 KB range
- Unconditional register-indirect jumps

- Unconditional absolute jumps

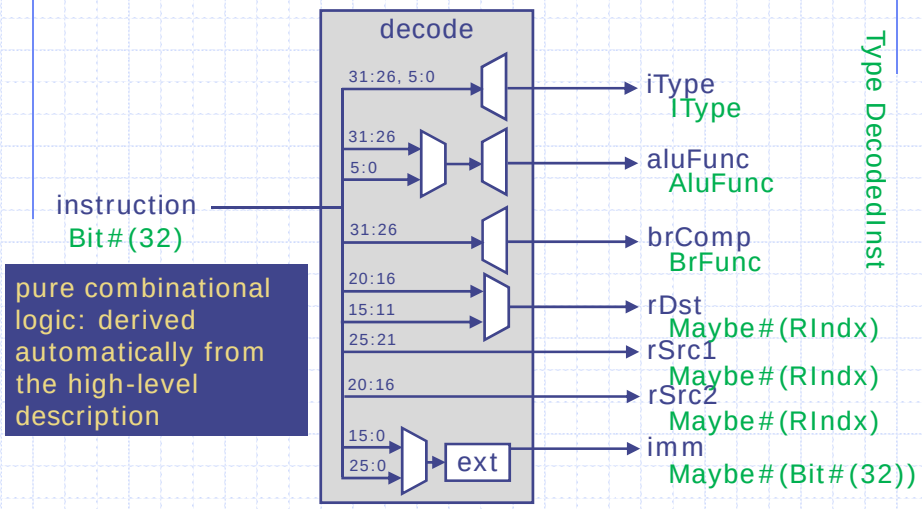


  - target address = {PC<31:28>, target×4}
  - range : 256 MB range

jump-&-link stores PC+4 into the link register (R31)

Decoding Instructions:

extract fields needed for execution



pure combinational logic: derived automatically from the high-level description

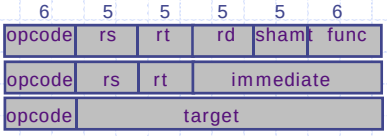
Decoded Instruction

```
typedef struct {
  IType             iType;
  AluFunc           aluFunc;
  BrFunc            brFunc;
  Maybe#(FullIndx) dst;
  Maybe#(FullIndx) src1;
  Maybe#(FullIndx) src2;
  Maybe#(Data)      imm;
} DecodedInst deriving(Bits, Eq);

typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br}
IType deriving(Bits, Eq);
typedef enum {Add, Sub, And, Or, Xor, Nor, Slt, Sltu,
LShift, RShift, Sra} AluFunc deriving(Bits, Eq);
typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT} BrFunc
deriving(Bits, Eq);
```

Decode Function

```
function DecodedInst decode(Bit#(32) inst);
   DecodedInst dInst = ?;
   let opcode = inst[ 31 : 26 ];
   let rs     = inst[ 25 : 21 ];
   let rt     = inst[ 20 : 16 ];
   let rd     = inst[ 15 : 11 ];
   let funct  = inst[  5 :  0 ];
   let imm    = inst[ 15 :  0 ];
   let target = inst[ 25 :  0 ];
   case (opcode)
      ...
   endcase
   return dInst;
endfunction
```

initially undefined



Naming the opcodes

```
Bit#(6) opADDIU = 6'b001001;
Bit#(6) opSLTI  = 6'b001010;
Bit#(6) opLW    = 6'b100011;
Bit#(6) opSW    = 6'b101011;
Bit#(6) opJ     = 6'b000010;
Bit#(6) opBEQ   = 6'b000100;
…
Bit#(6) opFUNC  = 6'b000000;
Bit#(6) fcADDU  = 6'b100001;
Bit#(6) fcAND   = 6'b100100;
Bit#(6) fcJR    = 6'b001000;
…
Bit#(6) opRT    = 6'b000001;
Bit#(6) rtBLTZ  = 5'b00000;
Bit#(6) rtBGEZ  = 5'b00100;
```

bit patterns are specified in the SMIPS ISA

Instruction Groupings

instructions with common execution steps

```
case (opcode)
  opADDIU, opSLTI, opSLTIU, opANDI, opORI, opXORI, opLUI: …
  opLW: …
  opSW: …
  opJ, opJAL: …
  opBEQ, opBNE, opBLEZ, opBGTZ, opRT: …
  opFUNC: case (funct)
       fcJR, fcJALR: …
       fcSLL, fcSRL, fcSRA: …
       fcSLLV, fcSRLV, fcSRAV: …
       fcADDU, fcSUBU, fcAND, fcOR, fcXOR,
                fcNOR, fcSLT, fcSLTU: … ;
      default: // Unsupported
          endcase
  default: // Unsupported
endcase;
```

These groupings are somewhat arbitrary

Decoding Instructions: I-Type ALU

```
opADDIU, opSLTI, opSLTIU, opANDI, opORI, opXORI, opLUI:
begin
      dInst.iType   =Alu;
      dInst.aluFunc =case (opcode)
                       opADDIU, opLUI: Add;
                       opSLTI: Slt;    opSLTIU: Sltu;
                       opANDI: And;    opORI: Or;
                       opXORI: Xor;
                     endcase;
      dInst.dst     = validReg(rt);
      dInst.src1    = validReg(rs);
      dInst.src2    = Invalid;
      dInst.imm     = Valid (case(opcode)
           opADDIU, opSLTI, opSLTIU: signExtend(imm);
           opLUI:                    {imm, 16'b0};
                              endcase);

      dInst.brFunc = NT;
end
```

almost like writing (Valid rt)

Decoding Instructions: Load & Store

```
opLW: begin
        dInst.iType   = Ld;
        dInst.aluFunc = Add;
        dInst.rDst    = validReg(rt);
        dInst.rSrc1   = validReg(rs);
        dInst.rSrc2   = Invalid;
        dInst.imm     = Valid (signExtend(imm));
        dInst.brFunc  = NT;          end
opSW: begin
        dInst.iType   = St;
        dInst.aluFunc = Add;
        dInst.rDst    = Invalid;
        dInst.rSrc1   = validReg(rs);
        dInst.rSrc2   = validReg(rt);
        dInst.imm     = Valid(signExtend(imm));
        dInst.brFunc  = NT;          end
```


Decoding Instructions: Jump

```
opJ, opJAL:
 begin
    dInst.iType  = J;
    dInst.rDst   = opcode==opJ ? Invalid :
                                 validReg(31);
    dInst.rSrc1  = Invalid;
    dInst.rSrc2  = Invalid;
    dInst.imm    = Valid(zeroExtend(
                           {target, 2'b00}));
    dInst.brFunc = AT;
 end
```

Decoding Instructions: Branch

```
opBEQ, opBNE, opBLEZ, opBGTZ, opRT:
begin
   dInst.iType  = Br;
   dInst.brFunc = case(opcode)
     opBEQ: Eq;          opBNE: Neq;
     opBLEZ: Le;         opBGTZ: Gt;
     opRT: (rt==rtBLTZ ? Lt : Ge);
   endcase;
   dInst.dst    = Invalid;
   dInst.src1   = validReg(rs);
   dInst.src2   = (opcode==opBEQ||opcode==opBNE)?
                    validReg(rt) : Invalid;
   dInst.imm    = Valid(signExtend(imm) << 2);
 end
```


Decoding Instructions: opFUNC, JR

```
opFUNC:
case (funct)
  fcJR, fcJALR:
      begin
        dInst.iType  = Jr;
        dInst.dst    = funct == fcJR? Invalid:
                                      validReg(rd);
        dInst.src1   = validReg(rs);
        dInst.src2   = Invalid;
        dInst.imm    = Invalid;
        dInst.brFunc = AT;
      end

  fcSLL, fcSRL, fcSRA: …
```

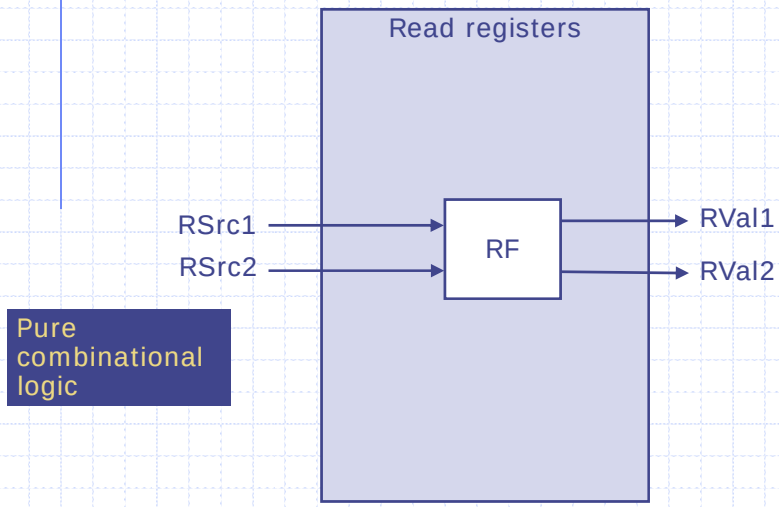
Decoding Instructions: opFUNC- ALU ops

```
fcADDU, fcSUBU, fcAND, fcOR, fcXOR, fcNOR, fcSLT, fcSLTU:
begin
    dInst.iType   = Alu;
    dInst.aluFunc = case (funct)
          fcADDU: Add;       fcSUBU: Sub;
          fcAND : And;       fcOR  : Or;
          fcXOR : Xor;       fcNOR : Nor;
          fcSLT : Slt;       fcSLTU: Sltu;  endcase;
    dInst.dst    = validReg(rd);
    dInst.src1   = validReg(rs);
    dInst.src2   = validReg(rt);
    dInst.imm    = Invalid;
    dInst.brFunc = NT
end
default: // Unsupported
endcase
```


Decoding Instructions: Unsupported instruction

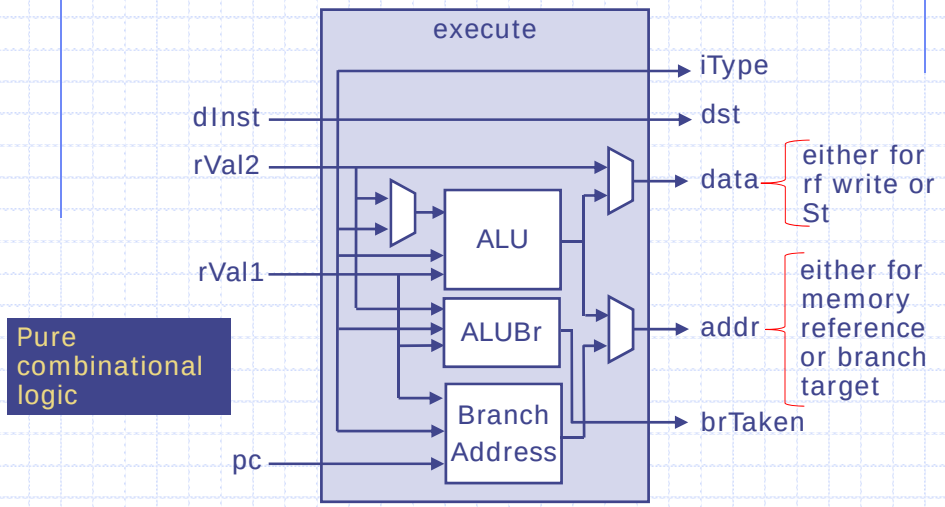
```
default:
    begin
      dInst.iType  = Unsupported;
      dInst.dst    = Invalid;
      dInst.src1   = Invalid;
      dInst.src2   = Invalid;
      dInst.imm    = Invalid;
      dInst.brFunc = NT;
    end
endcase
```

Reading Registers



Pure combinational logic

Executing Instructions



Pure combinational logic

Output of exec function

```
typedef struct {
  IType              iType;
  Maybe#(FullIndx) 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
```

Branch Address Calculation

```
function Addr brAddrCalc(Addr pc, Data val,
                         IType iType, Data imm, Bool taken);
  Addr pcPlus4 = pc + 4;
  Addr targetAddr = case (iType)
    J  : {pcPlus4[31:28], imm[27:0]};
    Jr : val;
    Br : (taken? pcPlus4 + imm : pcPlus4);
    Alu, Ld, St, Unsupported: pcPlus4;
  endcase;
  return targetAddr;
endfunction
```

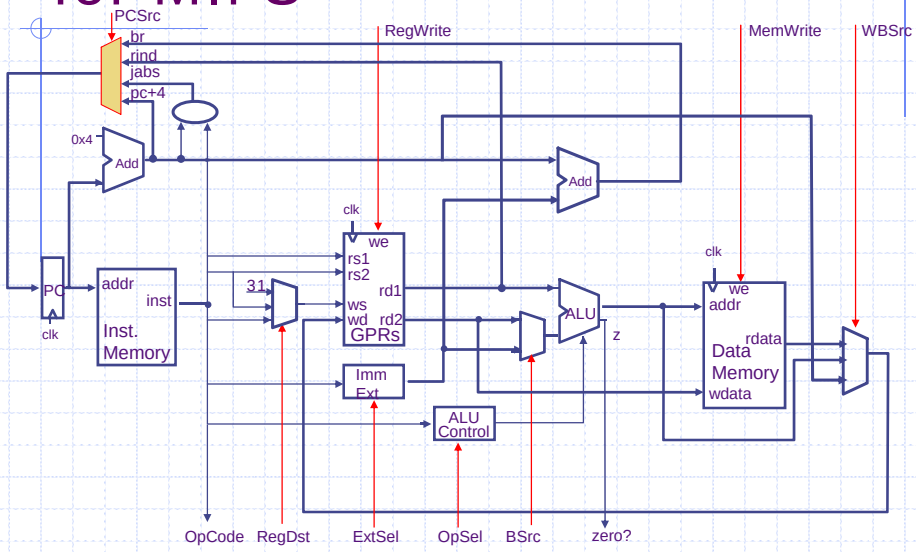

Single-Cycle SMIPS *atomic state updates*

```
    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(validRegValue(eInst.dst), eInst.data);

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

endrule
endmodule
```

state updates

Harvard-Style Datapath for MIPS

old way



old way

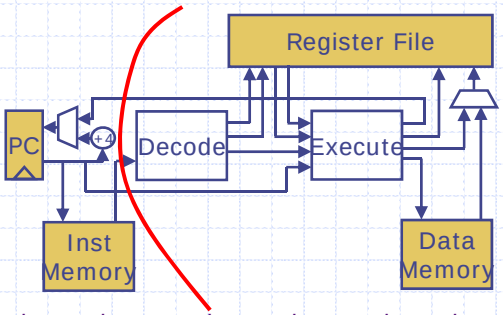
Hardwired Control Table

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

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

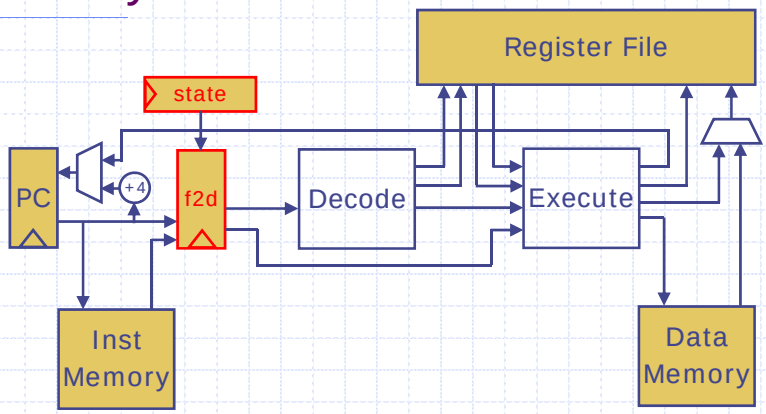
$t_{Clock} > \max \{t_M , (t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB})\}$

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

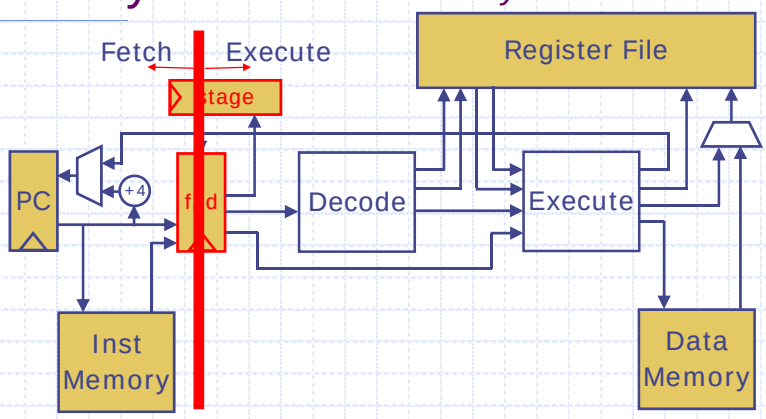
```
module mkProc(Proc);
  Reg#(Addr)  pc <- mkRegU;
  RFile       rf <- mkRFile;
  IMemory     iMem <- mkIMemory;
  DMemory     dMem <- mkDMemory;
  Reg#(Data)  f2d <- mkRegU;
  Reg#(State) state <- mkReg(Fetch);

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

Two-Cycle SMIPS

```
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;
  stage <= Fetch;
endrule endmodule
```

no change from single-cycle

Two-Cycle SMIPS: *Analysis*



In any given clock cycle, lots of unused hardware!

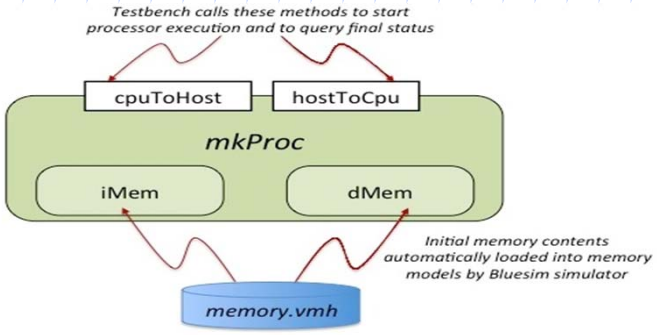
next lecture: Pipelining to increase the throughput

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 nth set are written and read using instructions MTCn and 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 `FullIndx` is used to refer to the normal registers plus the coprocessor set 0 registers
  - function `validRegValue(FullIndx r)` returns index of `r`

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


Processor interface




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

refers to coprocessor registers

Code with coprocessor calls

```
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