

Constructive Computer Architecture:

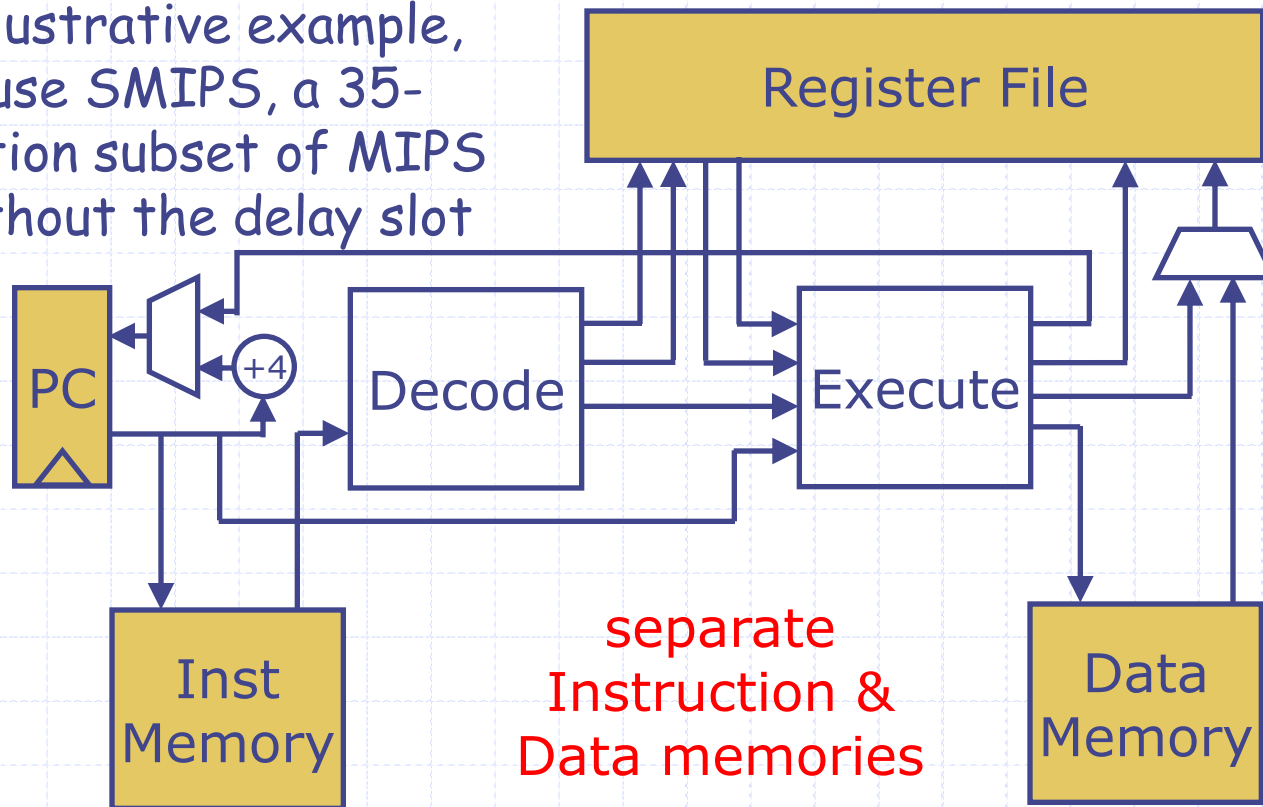
Non-Pipelined Processors

Arvind

Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Single-Cycle RISC Processor

As an illustrative example, we will use SMIPS, a 35-instruction subset of MIPS ISA without the delay slot



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

Single-Cycle Implementation

code structure

```
module mkProc (Proc);
```

```
  Reg# (Addr)  pc <- mkRegU;
```

```
  RFile        rf <- mkRFile;
```

```
  IMemory      iMem <- mkIMemory;
```

```
  DMemory      dMem <- mkDMemory;
```

to be explained later

instantiate the state

```
rule doProc;
```

```
  let inst = iMem.req(pc);
```

```
  let dInst = decode(inst);
```

```
  let rVal1 = rf.rd1(dInst.rSrc1);
```

```
  let rVal2 = rf.rd2(dInst.rSrc2);
```

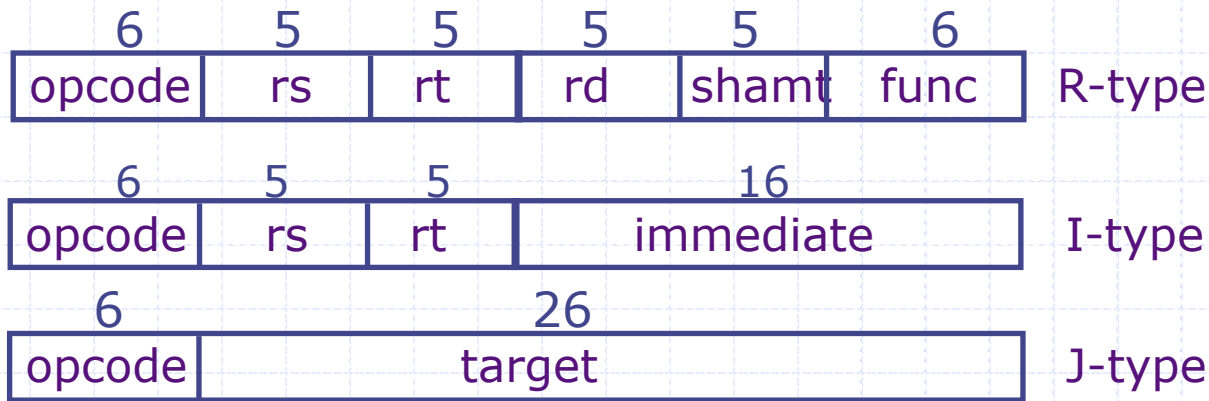
```
  let eInst = exec(dInst, rVal1, rVal2, pc);
```

```
  update rf, pc and dMem
```

extracts fields
needed for
execution

produces values
needed to
update the
processor state

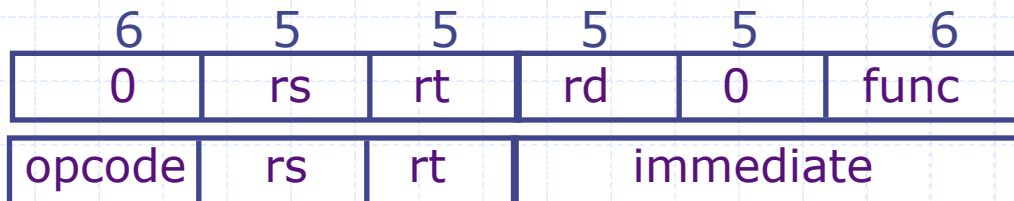
SMIPS Instruction formats



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

Instruction formats *cont*

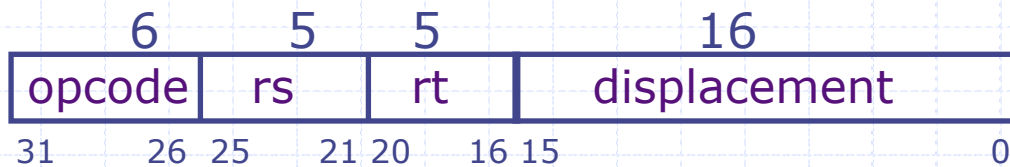
◆ Computational Instructions



$rd \leftarrow (rs) \text{ func } (rt)$

$rt \leftarrow (rs) \text{ op } \text{immediate}$

◆ Load/Store Instructions



addressing mode
 $(rs) + \text{displacement}$

rs is the base register

rt is the destination of a Load or the source for a Store

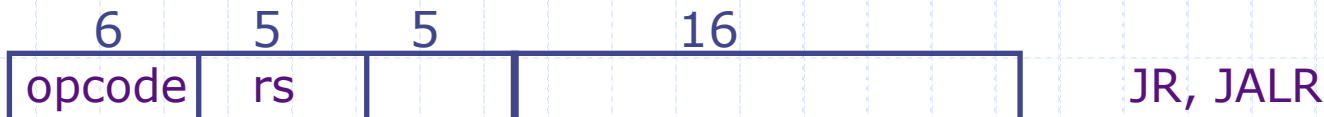
Control Instructions

◆ 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

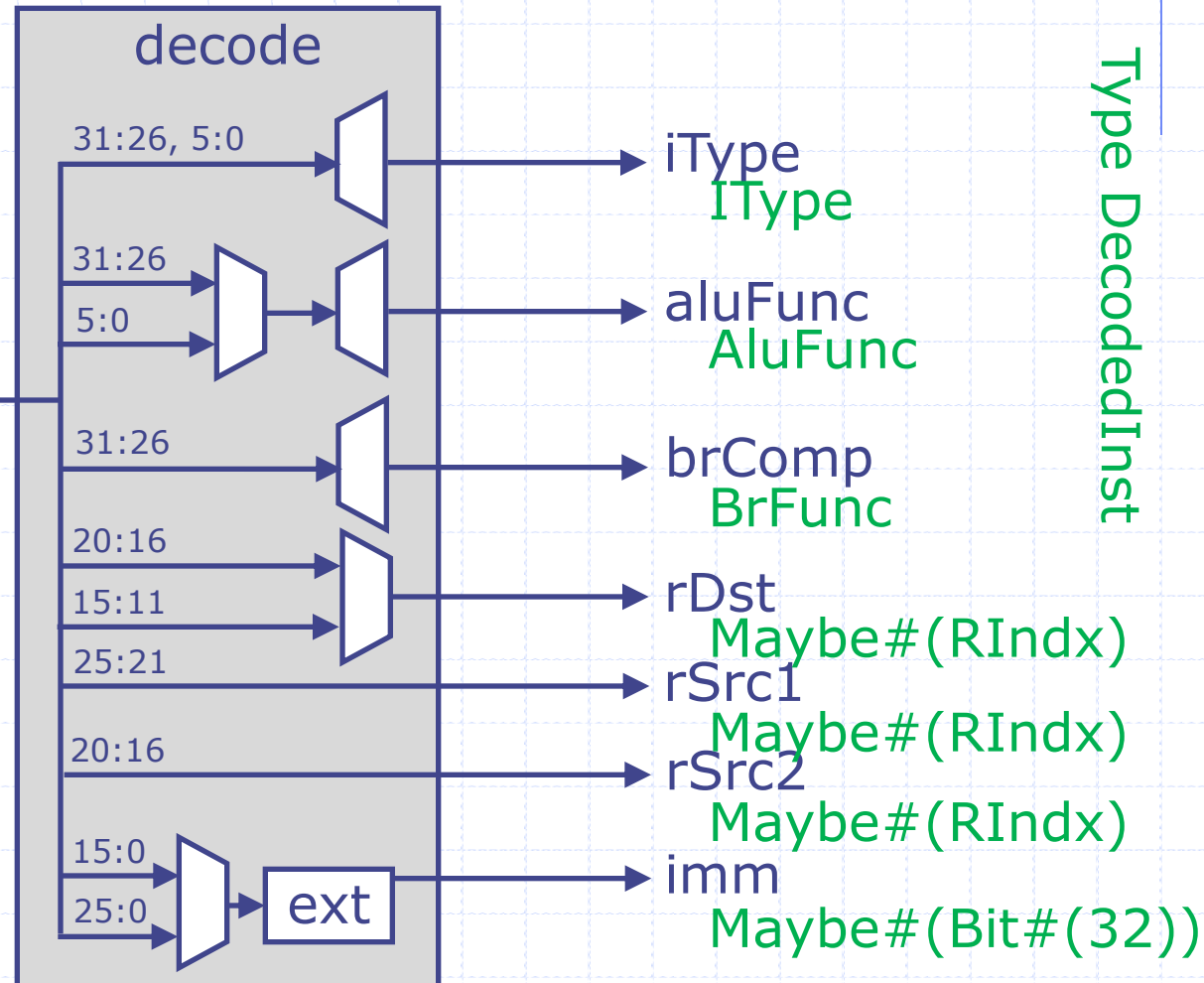
jump-&-link stores PC+4 into the link register (R31)

Decoding Instructions:

extract fields needed for execution

instruction
Bit#(32)

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);
```

FullIndx is similar to RIndx; to be explained later

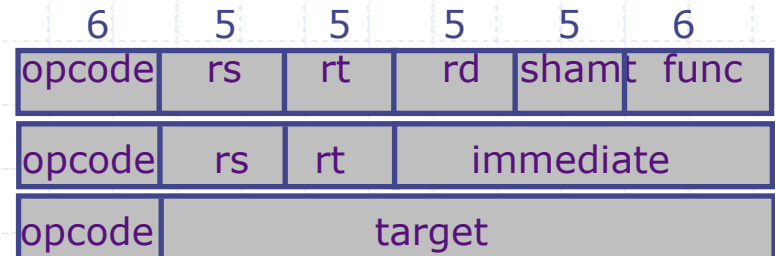
Destination register 0 behaves like an Invalid destination

Instruction groups with similar executions paths

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);      almost like writing
    dInst.src1     = validReg(rs);      (Valid rt)
    dInst.src2     = Invalid;
    dInst.imm      = Valid (case (opcode)
        opADDIU, opSLTI, opSLTIU: signExtend(imm);
        opLUI: {imm, 16'b0};
    endcase);
    dInst.brFunc  = NT;
```

end

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;
```

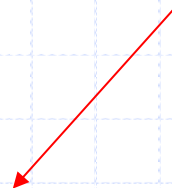
```
      dInst.imm   = Invalid;
```

```
      dInst.brFunc = AT;
```

```
    end
```

```
  fcSLL, fcSRL, fcSRA: ...
```

JALR stores the pc in rd as opposed to JAL which stores the pc in R31



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
```

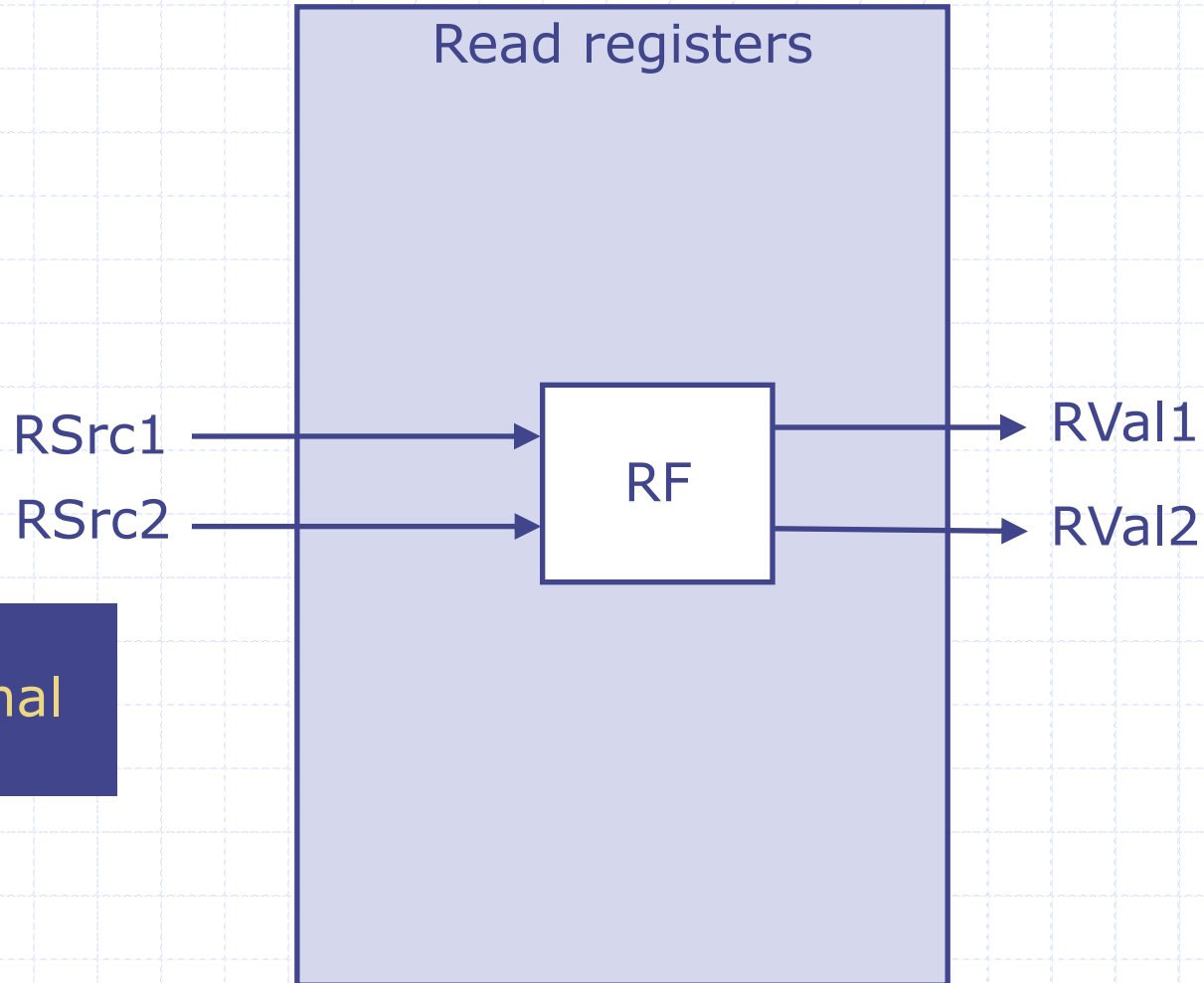
```
default: // Unsupported
```

```
endcase
```

Decoding Instructions: Unsupported instruction

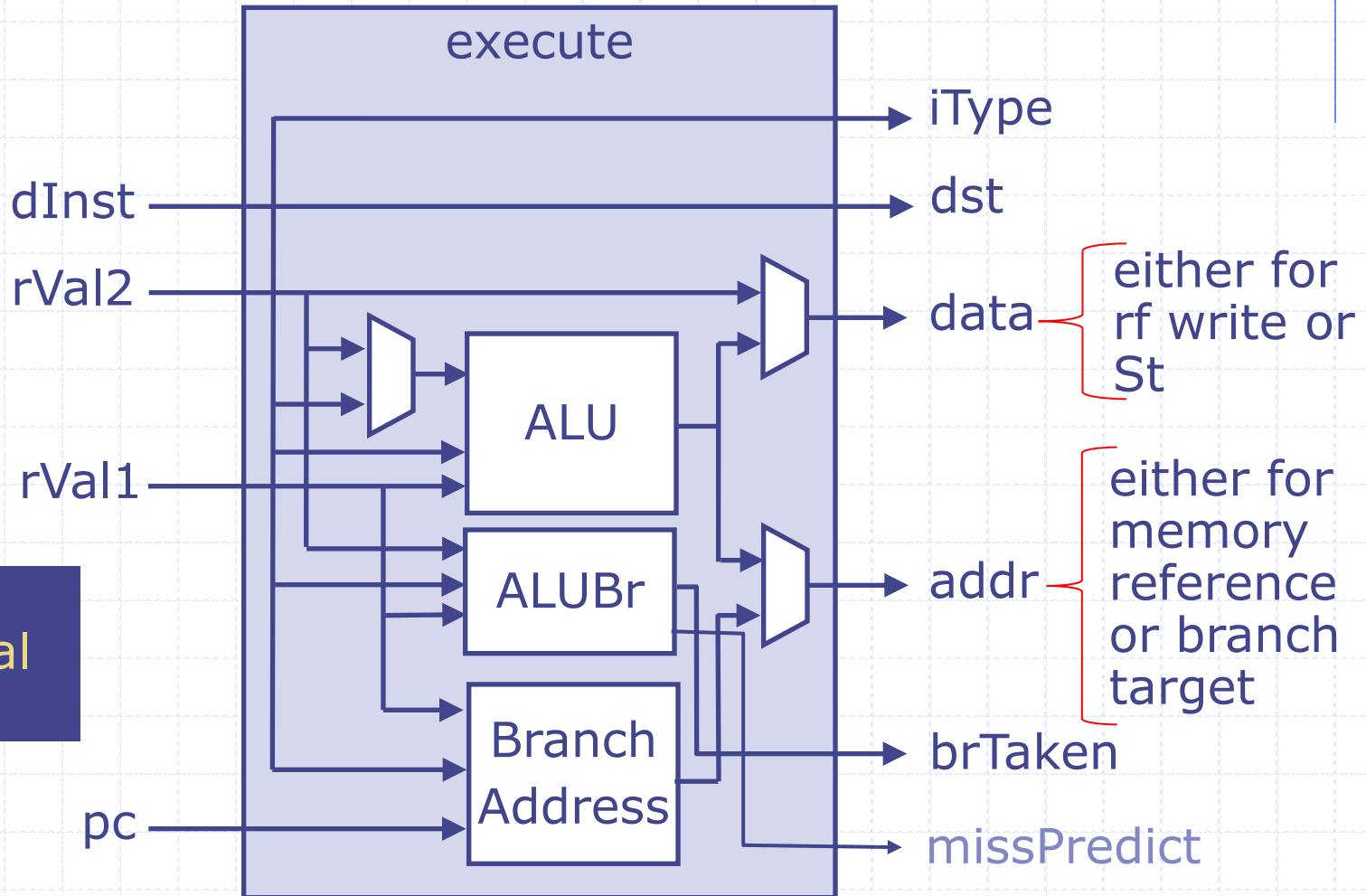
```
default:  
  begin  
    dInst.iType   = Unsupported;  
    dInst.dst     = Invalid;  
    dInst.src1    = Invalid;  
    dInst.src2    = Invalid;  
    dInst.imm     = Invalid;  
    dInst.brFunc  = NT;  
  end  
endcase
```

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 = fromMaybe(rVal2, dInst.imm);

    let aluRes = alu(rVal1, aluVal2, dInst.aluFunc);
    eInst.iType = dInst.iType;
    eInst.data = dInst.iType==St? rVal2 :
                (dInst.iType==J || dInst.iType==Jr)?
                (pc+4) : aluRes;

    let brTaken = aluBr(rVal1, rVal2, dInst.brFunc);
    let brAddr = brAddrCalc(pc, rVal1, dInst.iType,
                           fromMaybe(?, dInst.imm), brTaken);
    eInst.brTaken = brTaken;
    eInst.addr = (dInst.iType==Ld || dInst.iType==St)?
                aluRes : brAddr;
    eInst.dst = dInst.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;
```

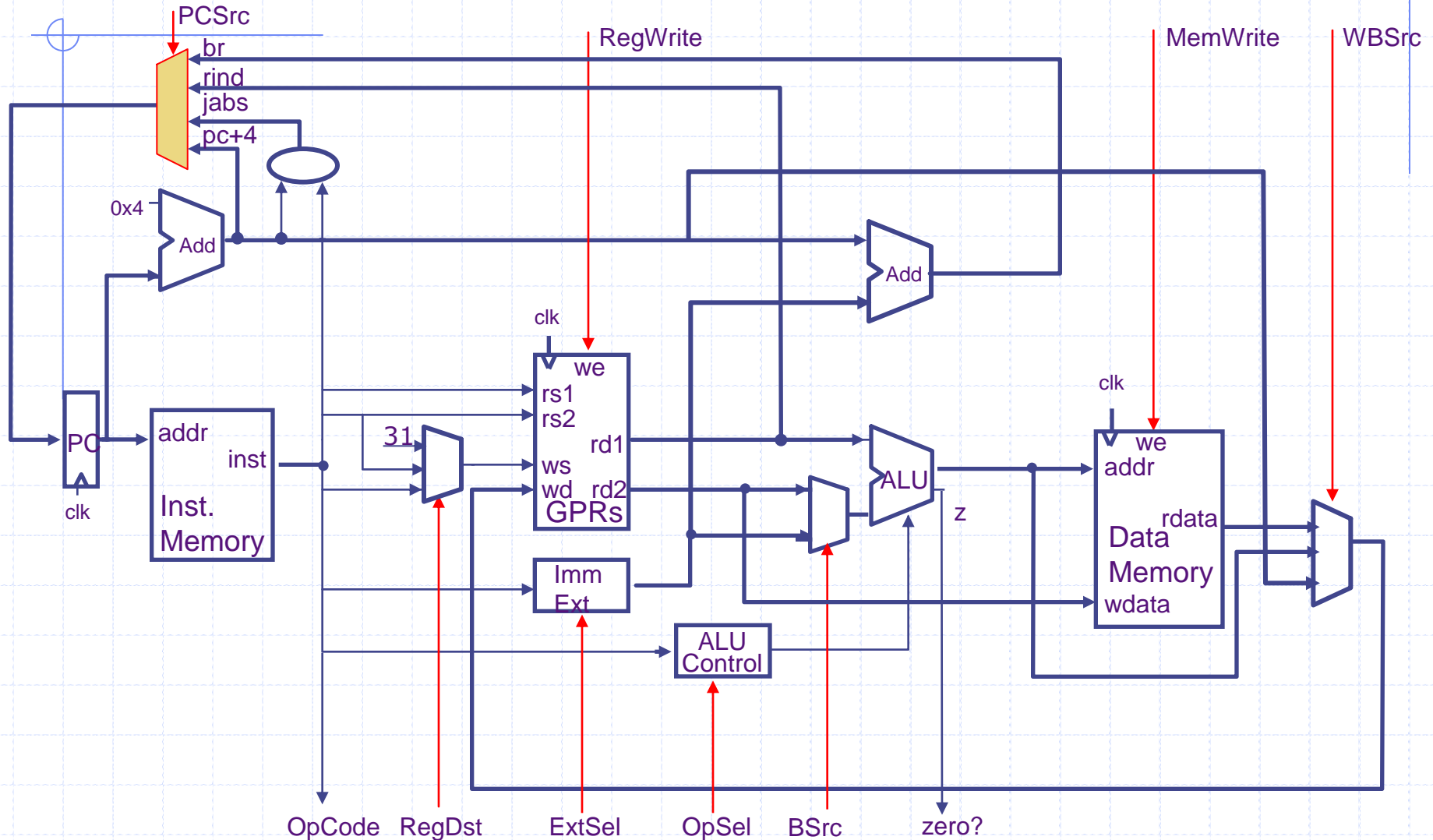
```
endrule
endmodule
```

state updates

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

Harvard-Style Datapath for MIPS

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

RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC

PCSrc = pc+4 / br / rind / jabs