Constructive Computer Architecture:

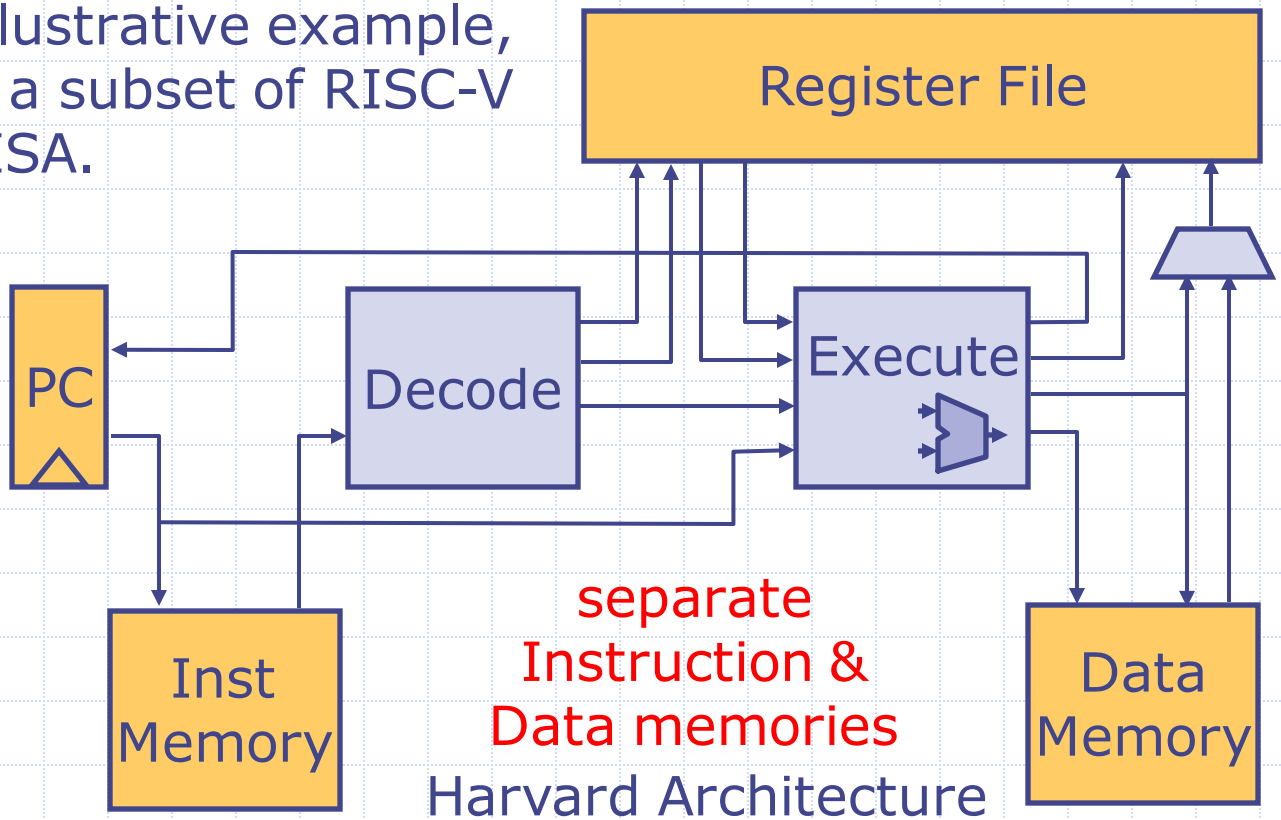
Non-Pipelined Processors

Thomas Bourgeat – EPFL

Slides Prepared with Arvind – 6.1920 spring 2023

Single-Cycle RISC Processor

As an illustrative example, we use a subset of RISC-V 32-bit ISA.



Datapath (arrows in this diagram) are conceptual; the real datapaths are derived automatically from the Bluespec description

RISC-V Register States

- 32 general purpose registers (GPR)
  - x0, x1, …, x31
  - 32-bit wide integer registers
  - x0 is hard-wired to zero
- Program counter (PC)
  - 32-bit wide
- CSR (Control and Status Registers)
  - mcycle
  - minstret
  - mhartid
  - mtohost
  - …

We will not deal with CSRs in this subject

Single-Cycle Implementation

```
module mkProcessor(Empty);
    Reg#(Word)  pc <- mkReg(0);
    RFile2R1W   rf <- mkRFile2R1W;
    MagicMemory iMem <- mkMagicMemory;
    MagicMemory dMem <- mkMagicMemory;
    rule doProcessor;
      let inst <- iMem.req(MemReq{op:Ld, addr:pc, data:dwv});
      let dInst = decode(inst);
// dInst fields: iType, aluFunc, brFunc, dst, src1, src2, imm
      let rVal1 = rf.rd1(dInst.src1);
      let rVal2 = rf.rd2(dInst.src2);
      let eInst = execute(dInst, rVal1, rVal2, pc);
// eInst fields: iType, dst, data, addr, nextPC
      updateState(eInst, pc, rf, dMem);
    endrule
endmodule
```

instantiate the state

extracts the fields

read the register file

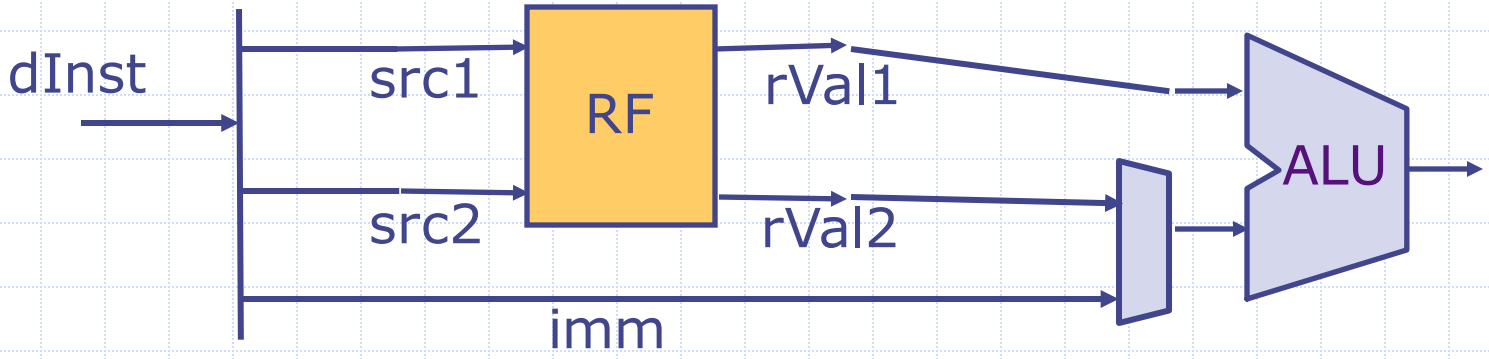
computes values needed to update the processor state

actions to update the processor state

Every BSV program serves two purposes

- It is a description from which a compiler can synthesize a hardware circuit
- Like any program, it is a description which can be executed to produce an answer
  - For example, the Single-cycle implementation can be simulated in software to execute any RISC-V program

Understanding generated hardware (the real datapath)




```
let rVal1 = rf.rd1(dInst.src1);
let rVal2 = rf.rd2(dInst.src2);
```

- Not all instructions have both src1 and src2 fields but there is no harm/cost in reading unused registers; we never use results of undefined fields
- When the same function is called with two different arguments, a mux is generated automatically

Understanding generated hardware - *continued*

```
case (iType)
    OP: data = alu(rVal1, rVal2, aluFunc);
    OPIMM: data = alu(rVal1, imm, aluFunc);
    ...
    LOAD: begin addr = rVal1+imm; nextPc = pc+4; end
    STORE: begin addr = rVal1+imm; data = rVal2;
                  nextPc = pc+4; end ...
```

- How many alu circuits?
  - The two uses of alu are mutually exclusive, so the BSV compiler/backend tools should share the same alu circuit;
  - Can address calculation use the same alu?
- Reuse is not necessarily a good idea because it prevents specialization
  - The circuit for pc+4 has a lot fewer gates than the circuit for pc+imm

Generally, we don't concern ourselves with the sharing of combinational circuits

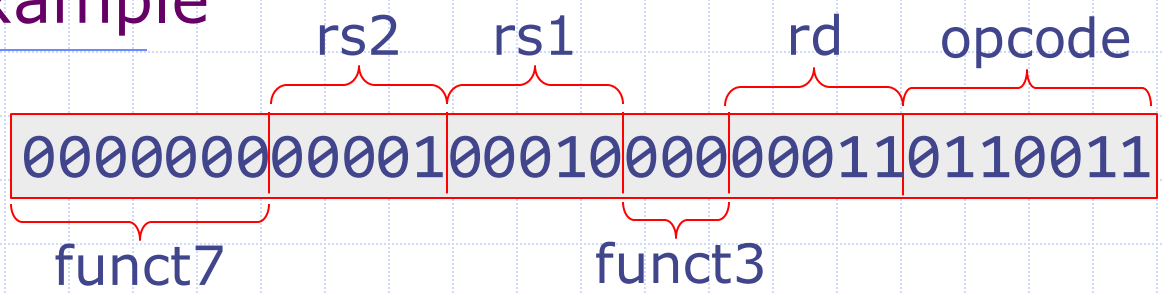
Instruction Decoding

- An instruction can be executed only after each of its fields has been extracted
  - Fields are needed to access the register file, compute address to access memory, supply the proper opcode to ALU, set the pc, ...
- Some 32-bit values may not represent an instruction or may represent an instruction not supported by our implementation
- Many instructions differ only slightly from each other in both decoding and execution

Unlike RISC-V, some instruction sets are extremely complicated to decode, e.g., Intel X86

Decoding instructions

An Example

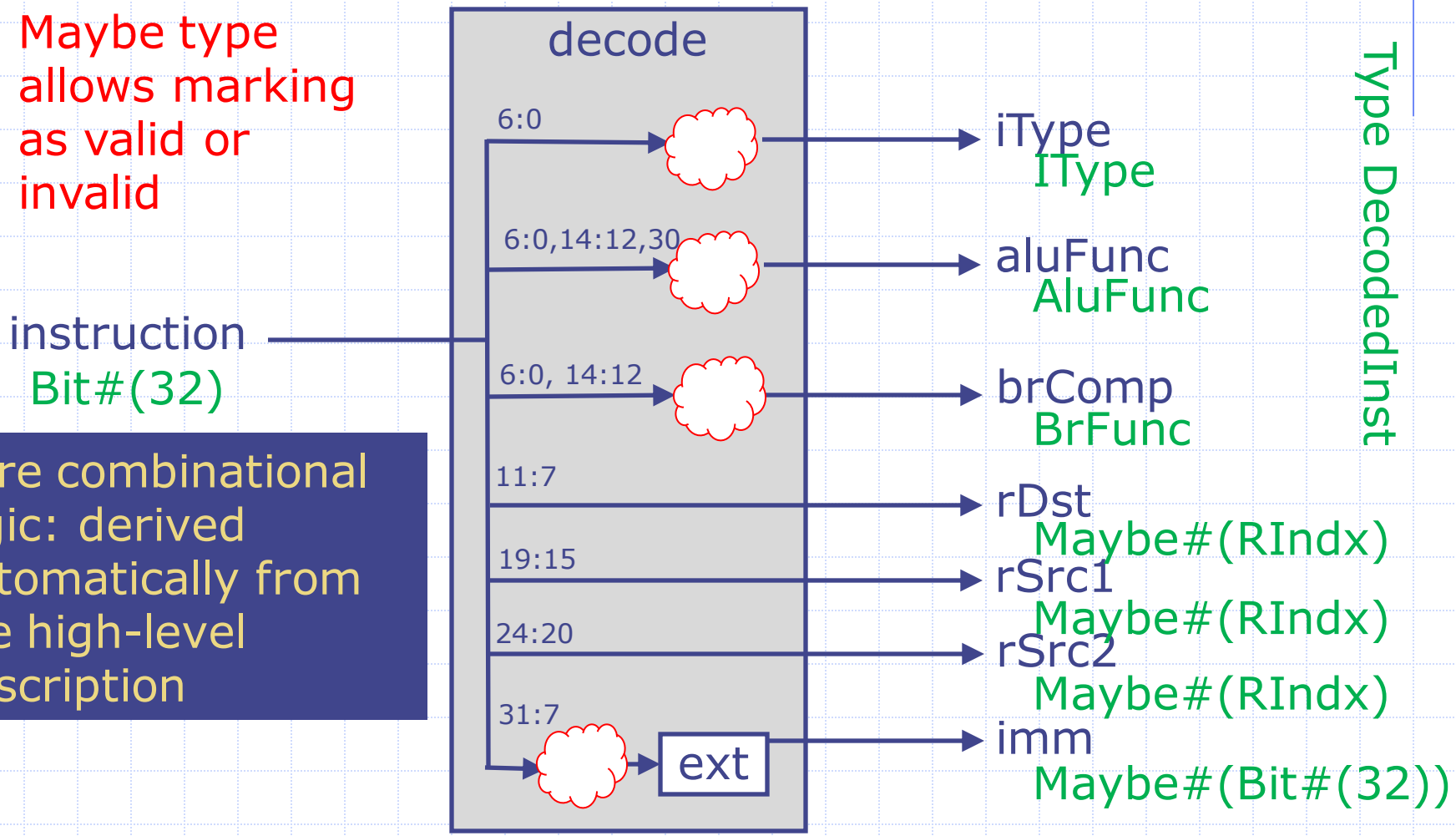


- What RISC-V instruction is represented by these 32 bits?
- Reference manual specifies the fields as follows:
  - `opcode = 0110011` => opCode Op, R-type encoding
  - `funct3 = 000` => ADD
  - `rd = 00011` => x3
  - `rs1 = 00010` => x2
  - `rs2 = 00001` => x1
- What is the meaning of executing this instruction?

```
rf.wr(3, alu(rf.rd1(2), rf.rd2(1), Add)); pc<=pc+4;
```

Decoding Instructions:
extract fields needed for execution

Maybe type allows marking as valid or invalid



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

Decoded Instruction Type

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

typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br, Auipc}
IType deriving(Bits, Eq);
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Sra,
Srl} AluFunc deriving(Bits, Eq);
typedef enum {Eq, Neq, Lt, Ltu, Ge, Geu, AT, NT} BrFunc
deriving(Bits, Eq);
```

Destination register 0 behaves like an Invalid destination

Instruction groups with similar executions paths

Internal names for various opcode and funct3 patterns

names given by us

```
// opcode
Bit#(7) opOpImm  = 7'b0010011; // OP-IMM
Bit#(7) opOp     = 7'b0110011; // OP
Bit#(7) opLui    = 7'b0110111; // LUI
Bit#(7) opAuipc  = 7'b0010111; // AUIPC
Bit#(7) opJal    = 7'b1101111; // JAL
Bit#(7) opJalr   = 7'b1100111; // JALR
Bit#(7) opBranch = 7'b1100011; // BRANCH
Bit#(7) opLoad   = 7'b0000011; // LOAD
Bit#(7) opStore  = 7'b0100011; // STORE
// funct3
Bit#(3) fnADD    = 3'b000; // ADD
Bit#(3) fnSLL    = 3'b001; // SLL
Bit#(3) fnSLT    = 3'b010; // SLT ……
```

Names and associated values are specified in the RISC-V ISA

…

Decode Function

```
function DecodedInst decode(Bit#(32) inst);
   let opcode = inst[  6 :  0 ];
   let rd     = inst[ 11 :  7 ];
   let funct3 = inst[ 14 : 12 ];
   let rs1    = inst[ 19 : 15 ];
   let rs2    = inst[ 24 : 20 ];
   let aluSel = inst[ 30 ]; // Add/Sub, Srl/Sra
   Bit#(32)immI=…; Bit#(32)immS=…; Bit#(32)immB=…;
   Bit#(32)immU=…; Bit#(32)immJ=…; // I/S/B/U/J-imm
   DecodedInst dInst = ?;
   case (opcode)
     opOp
     ...
   endcase
   return dInst;
endfunction
```

initially undefined

Decoding Instructions

```
case (opcode)
    opOpImm: …
    opOp: …
    opLui: …
    opAuipc: …
    opJal: …
    opJalr: …
    opBranch: …
    opLoad: …
    opStore: …
    default: … // Unsupported
endcase
```

```
// opcode
Bit#(7) opOpImm  = 7'b0010011;
Bit#(7) opOp     = 7'b0110011;
Bit#(7) opLui    = 7'b0110111;
Bit#(7) opAuipc  = 7'b0010111;
Bit#(7) opJal    = 7'b1101111;
Bit#(7) opJalr   = 7'b1100111;
Bit#(7) opBranch = 7'b1100011;
Bit#(7) opLoad   = 7'b0000011;
Bit#(7) opStore  = 7'b0100011;
```

Decoding Instructions: Computational Instructions

```
opOpImm:  begin
        dInst.iType = Alu;
        dInst.aluFunc = case (funct3)
            fnAND:  And;
            fnSLTU: Sltu;
            …
            fnADD:  Add;
            fnSR:   aluSel == 0 ? Srl : Sra;
        endcase;
        dInst.brFunc = NT;
        dInst.dst = Valid rd;
        dInst.src1 = Valid rs1;
        dInst.src2 =  Invalid;
        dInst.imm =  Valid ImmI;
    end
```

Decoding instructions with immediate operand (i.e., opcode = OP-IMM) is similar

Decoding Instructions: Unconditional Jumps

```
opJal: begin
    dInst.iType = J;
    dInst.aluFunc = ?;
    dInst.brFunc = AT;
    dInst.dst = Valid rd;
    dInst.src1 = Invalid;
    dInst.src2 = Invalid;
    dInst.imm = Valid immJ;
end
opJalr: begin
    dInst.iType =  Jr;
    dInst.aluFunc =  ?;
    dInst.brFunc =  AT;
    dInst.dst =  Valid rd;
    dInst.src1 =  Valid rs1;
    dInst.src2 =  Invalid;
    dInst.imm =  Valid immI;
end
```

Jump to pc+offset

Jump indirect through register

Decoding Instructions: Conditional Branch

```
opBranch: begin
    Maybe#(BrFunc) brF =
        case(funct3)
        fnBEQ:  Valid Eq;
        …
        fnBGEU: Valid Geu;
        default: Invalid;
    endcase;
    dInst.iType =   isValid(brF) ? Br : Unsupported;
    dInst.aluFunc =   ?;
    dInst.brFunc =  fromMaybe(?, brF);
    dInst.dst = Invalid;
    dInst.src1 = Valid rs1;
    dInst.src2 = Valid rs2;
    dInst.imm = Valid immB;
end
```

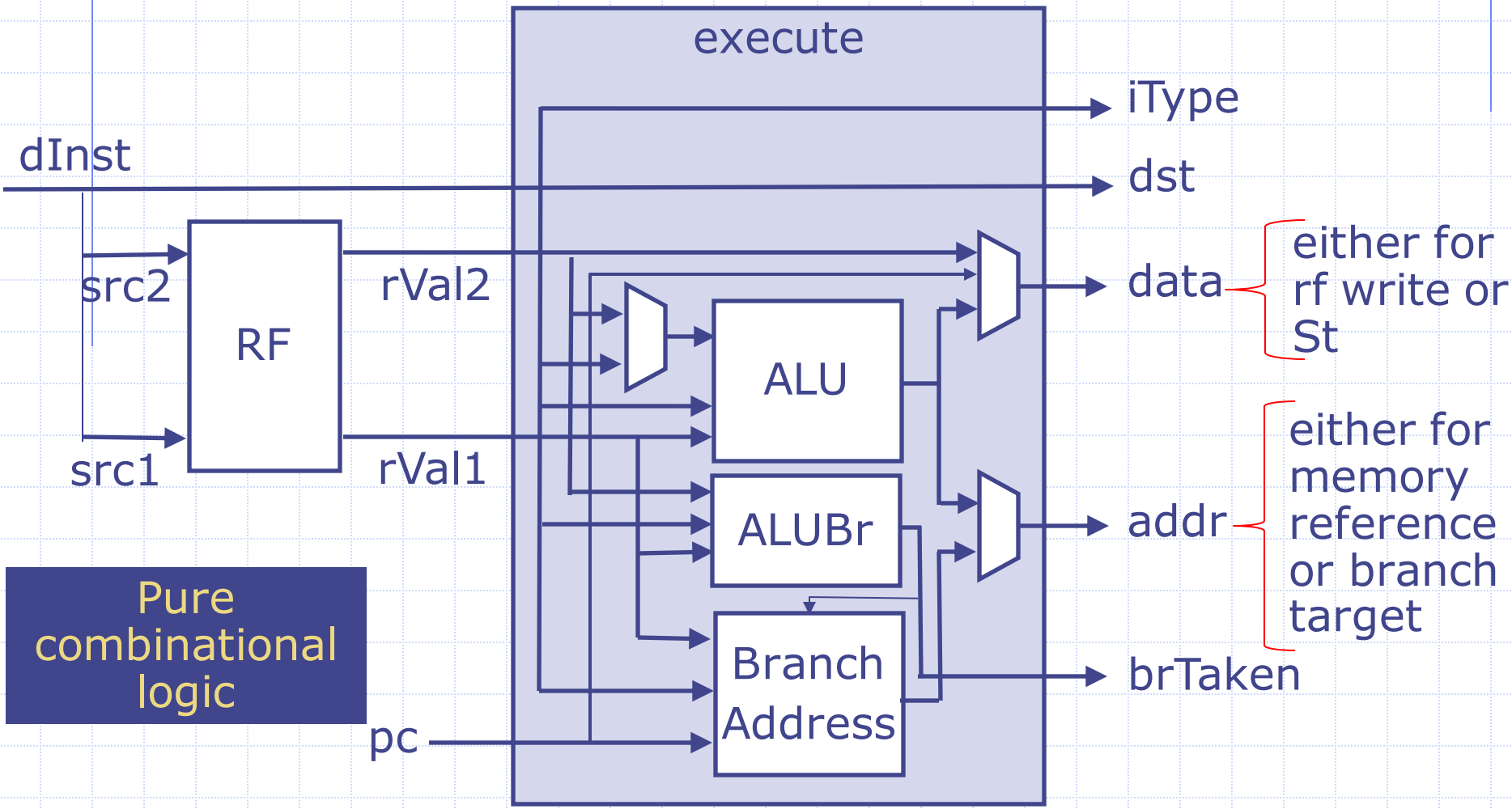
Decoding Instructions: Load & Store

```
opLoad: begin // only support LW
    dInst.iType = funct3 == fnLW ? Ld : Unsupported;
    dInst.aluFunc = Add; // calc effective addr
    dInst.brFunc = NT;
    dInst.dst = Valid rd;
    dInst.src1 = Valid rs1;
    dInst.src2 = Invalid;
    dInst.imm = Valid immI;
end
opStore: begin // only support SW
    dInst.iType = funct3 == fnSW ? St : Unsupported;
    dInst.aluFunc = Add; // calc effective addr
    dInst.brFunc = NT;
    dInst.dst =    Invalid;
    dInst.src1 =   Valid rs1;
    dInst.src2 =   Valid rs2;
    dInst.imm =    Valid immS;
end
```

Decoding instructions: Unsupported

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

Reading Registers and Executing Instructions



Output type of exec function

```
typedef struct {
    IType              iType;
    Maybe#(RIndx)      dst;
    Data               data;
    Addr               addr;
    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):
                    dInst.iType==Auipc ?
                     (pc+fromMaybe(?, dInst.imm)) : aluRes;
  eInst.brTaken  =  aluBr(rVal1, rVal2, dInst.brFunc);
  let brAddr     =  brAddrCalc(pc, rVal1, dInst.iType,
                               fromMaybe(?, dInst.imm));
  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);
   Addr targetAddr = case (iType)
    J  : {pc + imm};
    Jr : {truncateLSB(val + imm), 1'b0};
    Br : pc + imm;
    default: error;  //32'hAAAAAAAA
   endcase;
  return targetAddr;
endfunction
```

Half word aligned;
Maybe needed to support
compressed instruction format

Single-Cycle RISC-V

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(fromMaybe(?, eInst.dst), eInst.data);
    let nextPC = eInst.brTaken ? eInst.addr: pc + 4;
    pc <= nextPC;
endrule
endmodule
```

state updates

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

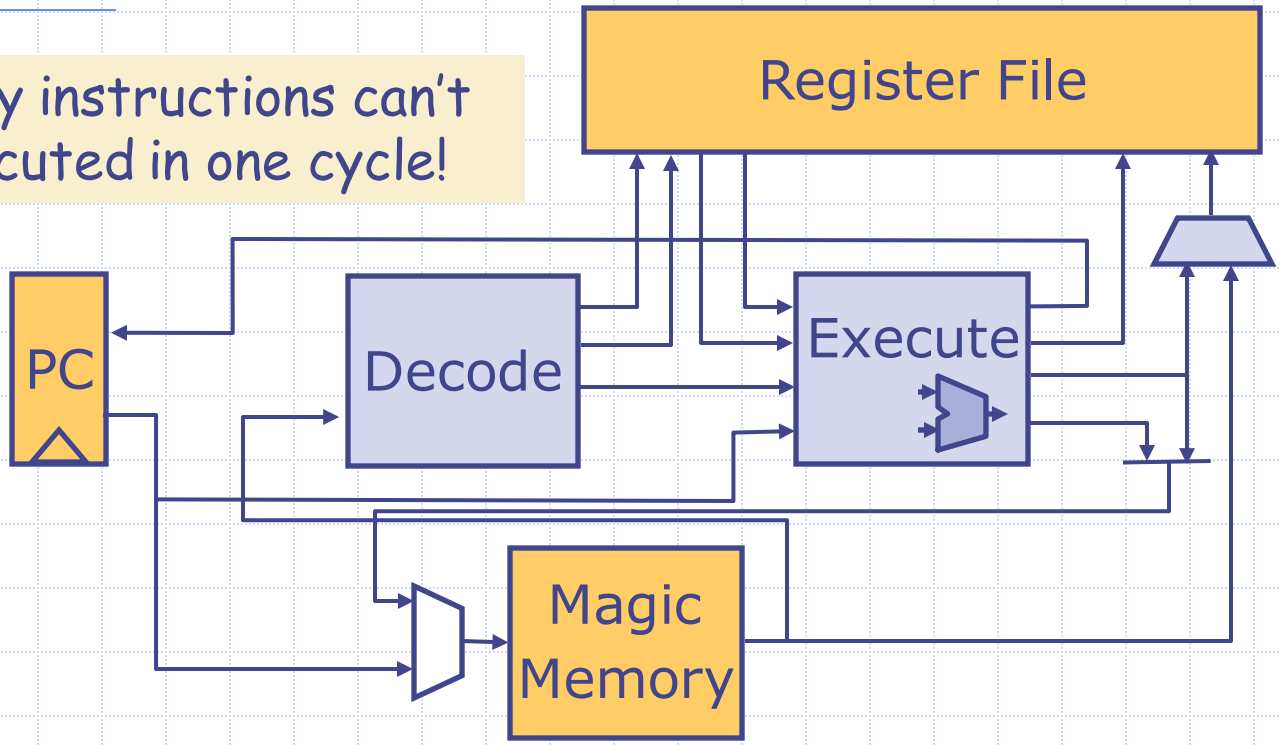
Variations on non-pipelined microarchitectures

An exercise in construction and deconstruction

Princeton Architecture

instructions and data reside in the same memory

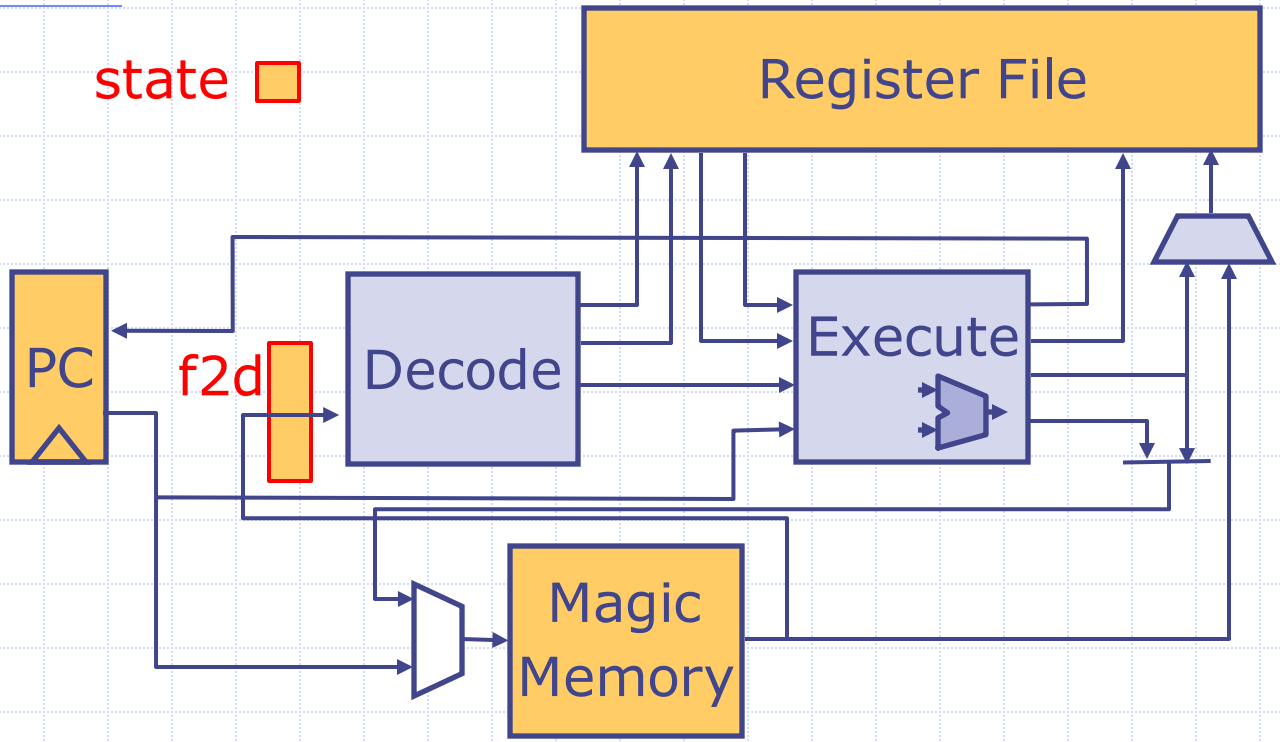
Memory instructions can't be executed in one cycle!



- We can't do instruction fetch and LD/ST at the same time because there is only one memory
- Such resource conflicts are known as *structural hazards* and require multicycle implementation
- Extra registers are required to hold values between cycles

Princeton Architecture

introduce intermediate state



- Every instruction takes two cycles: Fetch followed by Execute
- Insert f2d register to hold the fetched instruction
- A one bit register to record the state of the instruction

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

Princeton Architecture

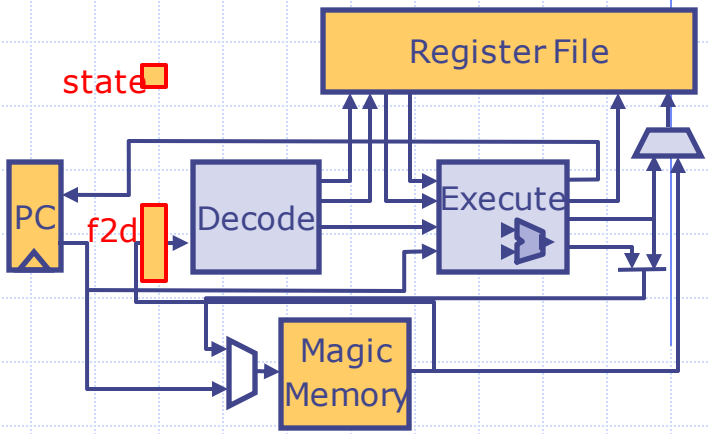
Two-cycle

```
module mkProcPrincetonTwoCycle(Empty);
  Code to instantiate pc, rf, magic mem
  Reg#(Word)  f2d <- mkRegU;
  Reg#(State) state <- mkReg(Fetch);

  rule doFetch if (state == Fetch);
    Code to fetch the instruction at pc
    f2d <= inst;
    state <= Execute;
  endrule

  rule doExecute if (state == Execute);
     Code to decode the instruction in f2d, execute,
updateState
     state <= Fetch;
  endrule
endmodule
```

If state is Fetch then fetch the instruction and put it in f2d, and change the state to Execute

If state is Execute then execute the instruction in f2d, and change the state to Fetch

doExecute rule reexamined

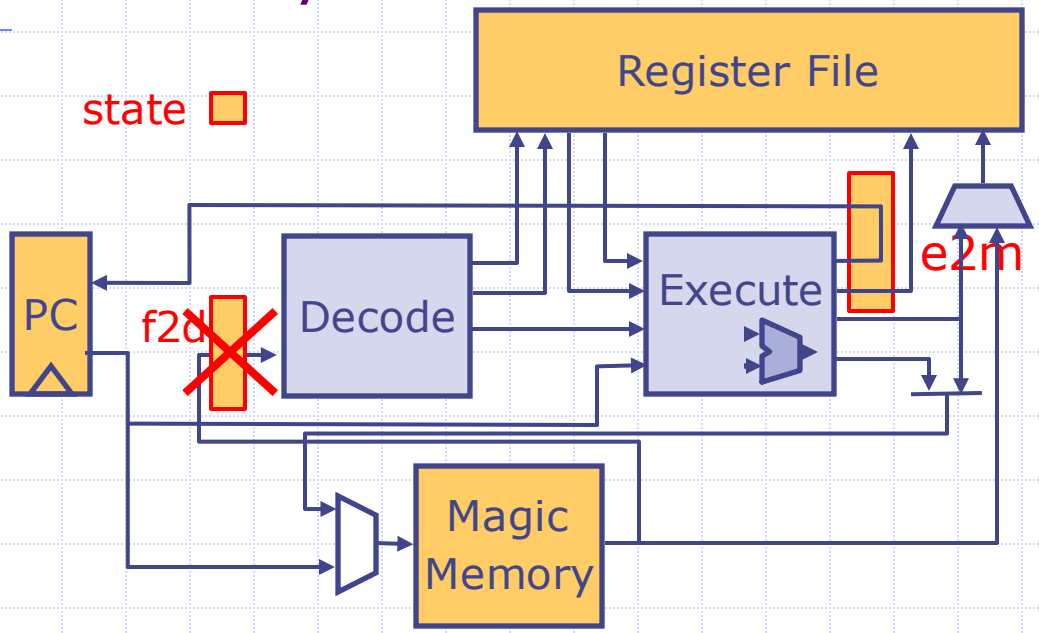



```
rule doExecute if (state == Execute);
   let inst = f2d;
   let dInst = decode(inst);
   let rVal1 = rf.rd1(dInst.src1);
   let rVal2 = rf.rd2(dInst.src2);
   let eInst = execute(dInst, rVal1, rVal2, pc);
   //Extract fileds of eInst: data, addr, dst;
   // memory access
   if (eInst.iType == LOAD) begin
      data <- dMem.req(MemReq{op:Ld, addr:addr, data: dwv});
   end else if (eInst.iType == STORE) begin
      let dummy <- dMem.req(MemReq{op:St, addr:addr,
data:data});
   end
   // register file write
   if (dst.valid) rf.wr(dst.index, data);
   // pc update
   pc <= eInst.nextPc;    state <= Fetch;
```

updateState(eInst,pc,rf,dMem) expanded

Execution of all instructions except LD/ST can be completed in the Fetch cycle

Another Princeton Architecture:
where non-memory instructions take one cycle



The datapath for e2m is not accurate

- Do fetch, decode, and execute in one cycle for all non-memory instructions; no need for f2d
- For a memory instruction:
  - after fetch, decode and execute, put the partially executed instruction in a register (e2m)
  - In the next cycle, do the memory operation, update the register file and pc

Princeton Architecture

where non-memory instructions take one cycle

```
rule doFetch if (state == FetchExecute);
  let inst <- mem.req(MemReq{op: Ld, addr: pc, data: dwv});
  let dInst = decode(inst);
  let rVal1 = rf.rd1(dInst.src1); //similarly rVal2
  let eInst = execute(dInst, rVal1, rVal2, pc);
  if (eInst.iType==Ld) || (eInst.iType==St) begin
    e2m <= eInst;
    state <= MemoryAccess;
  end else begin
  if (eInst.dst.valid) rf.wr(eInst.dst.index, eInst.data);
    pc <= eInst.nextPc;
    state <= FetchExecute;
  end
Endrule
rule doMemoryAccess if (state == MemoryAccess);
    updateState(e2m, pc, rf, mem);
    state <= FetchExecute;
endrule
```

Save the executed instruction state in e2m register

Performance implications

- Suppose fraction f of N executed instructions are memory access instructions
  - Two-cycle Princeton architecture will take 2N cycles
  - How many cycles will the variable-cycle Princeton architecture take?

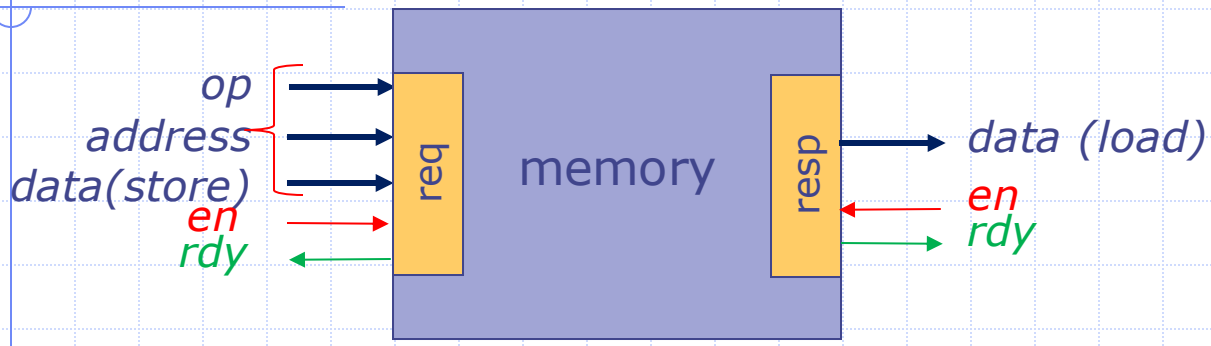
f*2*N + (1-f)*N
= (1+f)*N

As far as the performance is concerned, cycle counts in not the whole story; one needs to compare the cycle times of the two designs as well

More to come later

Realistic Memory Interface

Request/Response methods

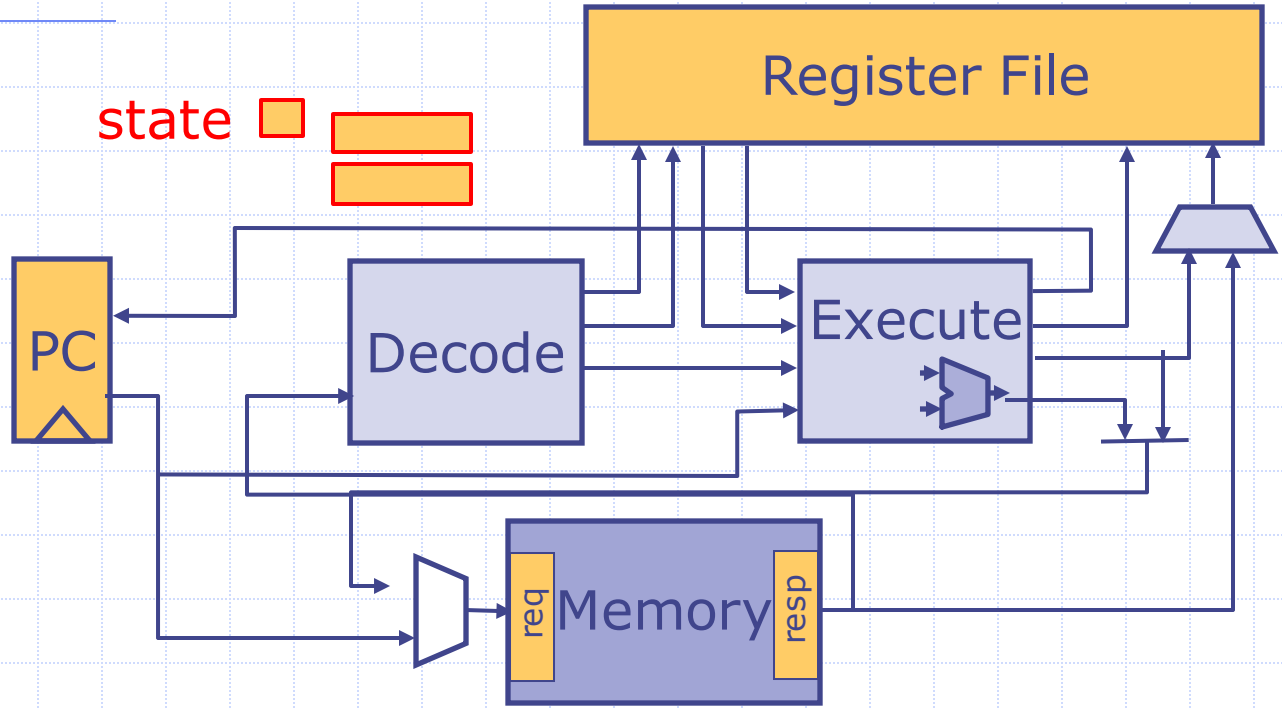


No response for Stores;
Load responses come back in the requested order

```
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp();
endinterface
typedef struct {MemOp op; Word addr; Word data;}
        MemReq deriving(Bits, Eq);
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
```

```
m.req(MemReq{op:Ld, addr:a, data:dwv});
m.req(MemReq{op:St, addr:a, data:v});
let data <- m.resp();
```

Princeton architecture with a realistic memory



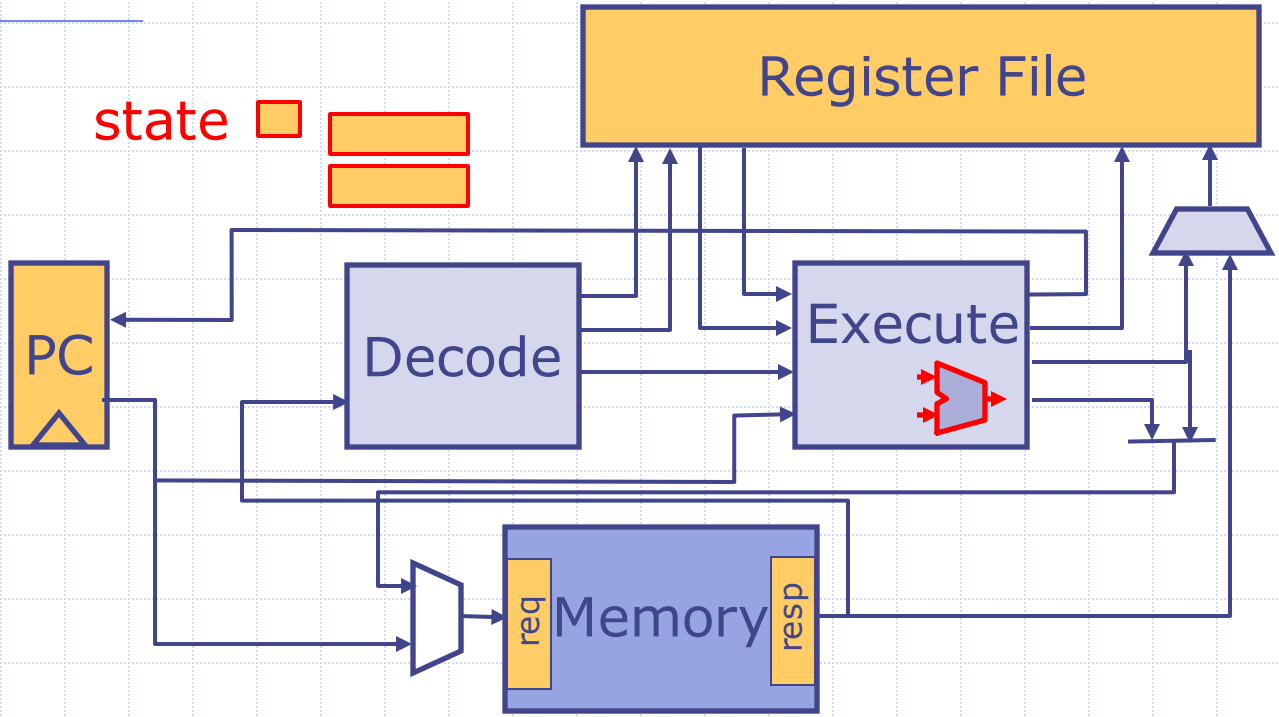
- With request/response memory even instruction fetch cannot be completed in one cycle
  - Instruction fetch must be split into two rules – send request and receive response
  - Need registers to hold the state of a partially executed instructions

Processor with realistic memory

```
module mkProcPrincetonMulticycle(Empty);
    Code to instantiate pc, rf, mem, and registers that hold
    the state of a partially executed instruction
    rule doFetch if (state == Fetch);
       Code to initiate instruction fetch; go to Execute
    rule doExecute if (state == Execute);
       let inst <- mem.resp;
       Code to 1. execute all instructions except memory
         instructions; go to Fetch
       Or 2. initiate memory access;
             go to Fetch (Store) OR go to LoadWait (Load)
    rule doLoadWait if (state == LoadWait);
       Code to wait for the load value, update rf, go to Fetch
endmodule
```

Multicycle ALU's

multicycle or floating point ALU operations

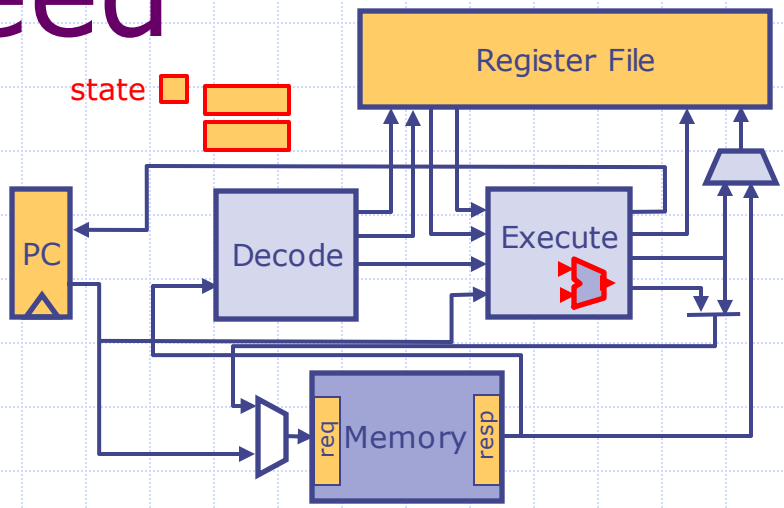


- Multicycle ALU's can be viewed as request/response modules
- Instructions can be further classified after decoding as simple 1 cycle, multicycle (e.g., multiply) or memory access

Processor with realistic memory and multicycle ALUs

```
module mkProcMulticycle(Empty);
    Code to instantiate pc, rf, mem, and registers to hold the
    state of a partially executed instruction
    rule doFetch if (state == Fetch);
       Code to initiate instruction fetch; go to Execute
    rule doExecute if (state == Execute);
       let inst <- mem.resp;
       Code to 1. execute all instructions except
                  memory and multicycle instructions; go to Fetch
       Or 2. initiate memory access
             go to Fetch (Store) OR go to LoadWait (Load)
       Or 3. initiate multicycle instruction; go to MCWait
    rule doLoadWait if (state == LoadWait);
       Code to wait for the load value, update rf, go to Fetch
    rule doMCWait if (state == MCWait);
       Code to wait for MC value, update rf, go to Fetch
endmodule
```

Clock speed



- Clock speed depends upon the longest combinational path between two state elements. Thus, in a single cycle implementation

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

- Clock in a two-cycle implementation may be faster

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

However, this may not improve the performance because now some instructions will take two cycles to execute

Cycle counts

- Different instructions take different number of cycles in our designs
  - Depending upon the type of opcode, an instruction has to go through 1 to 3 processor-rule firings
  - The number of cycles between processor-rule firings depends on how quickly the memory responds or a multicycle functional unit completes its work for the input data

Next, we will study pipelining

Processor Interface

```
module mkProcessor(Empty);
    ...
endmodule
```

- For testing, processor is connected to a host computer* which can read and write the memory of the processor directly
- The processor's memory is preloaded with program and data; it always starts at pc=0
- When the program terminates it writes a 0 in a predetermined location and stops the simulation
  - If the program hits an illegal or unsupported instruction, it dumps the processor state and stops the simulation

Consequently, the processor interface has no methods, ie, its interface is Empty!

*In the simulation environment the host computer is the same computer on which the simulator runs

Extras

Instruction Formats

- R-type instruction

- I-type instruction & I-immediate (32 bits)

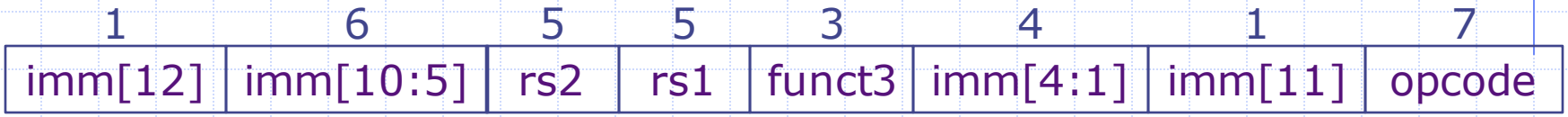
I-imm = signExtend(inst[31:20])

- S-type instruction & S-immediate (32 bits)

S-imm = signExtend({inst[31:25], inst[11:7]})

Instruction Formats *cont.*

- SB-type instruction & B-immediate (32 bits)

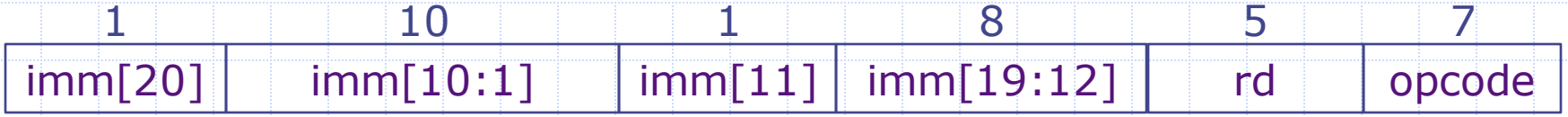


B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

- U-type instruction & U-immediate (32 bits)

U-imm = signExtend({inst[31:12], 12'b0})

- UJ-type instruction & J-immediate (32 bits)



J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})

Computational Instructions *cont.*

- Register-immediate instructions (U-type)

  - opcode = LUI : rd ← U-imm
  - opcode = AUIPC : rd ← pc + U-imm
  - U-imm = {inst[31:12], 12'b0}

Computational Instructions

- Register-Register instructions (R-type)

  - opcode=OP: rd ← rs1 (funct3, funct7) rs2
  - funct3 = SLT/SLTU/AND/OR/XOR/SLL
  - funct3= ADD
    - funct7 = 0000000: rs1 + rs2
    - funct7 = 0100000: rs1 – rs2
  - funct3 = SRL
    - funct7 = 0000000: logical shift right
    - funct7 = 0100000: arithmetic shift right

Computational Instructions *cont*

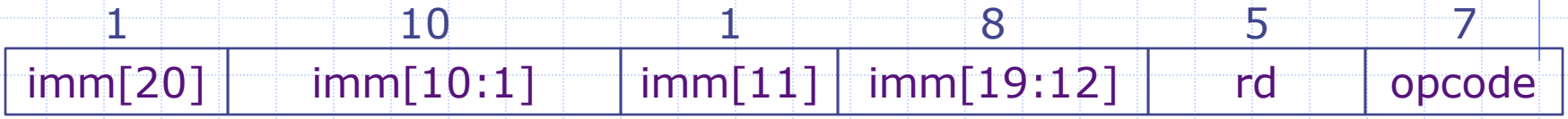
- Register-immediate instructions (I-type)

  - opcode = OP-IMM: rd ← rs1 (funct3) I-imm
  - I-imm = signExtend(inst[31:20])
  - funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI

- A slight variant in coding for shift instructions - SLLI / SRLI / SRAI
  - rd ← rs1 (funct3, inst[30]) I-imm[4:0]

Control Instructions

- Unconditional jump and link (UJ-type)

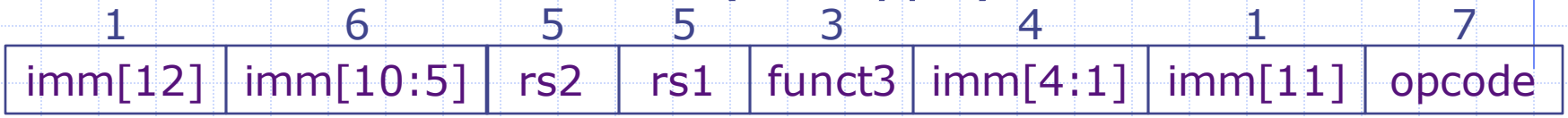


  - opcode = JAL: rd ← pc + 4; pc ← pc + J-imm
  - J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
  - Jump ±1MB range
- Unconditional jump via register and link (I-type)

  - opcode = JALR: rd ← pc + 4; pc ← (rs1 + I-imm) & ~0x01
  - I-imm = signExtend(inst[31:20])

Control Instructions *cont.*

- Conditional branches (SB-type)



  - opcode = BRANCH: pc ← compare(funct3, rs1, rs2) ? pc + B-imm : pc + 4
  - B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})
  - Jump ±4KB range
  - funct3 = BEQ/BNE/BLT/BLTU/BGE/BGEU

Load & Store Instructions

- Load (I-type)

  - opcode = LOAD: rd ← mem[rs1 + I-imm]
  - I-imm = signExtend(inst[31:20])
  - funct3 = LW/LB/LBU/LH/LHU

- Store (S-type)

  - opcode = STORE: mem[rs1 + S-imm] ← rs2
  - S-imm = signExtend({inst[31:25], inst[11:7]})
  - funct3 = SW/SB/SH