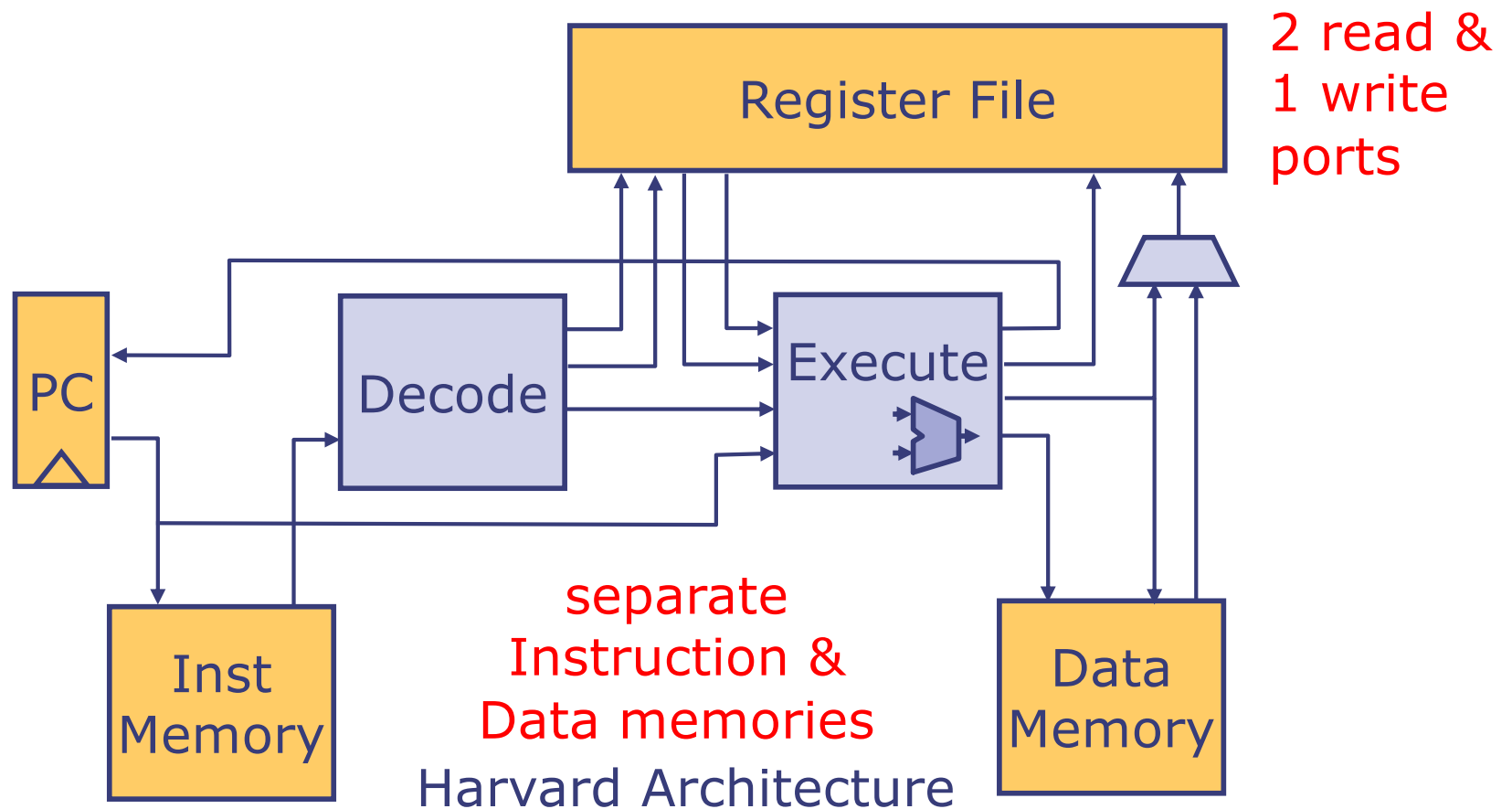


Non-pipelined processors

Arvind

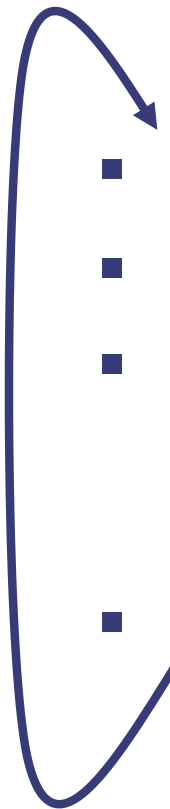
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Processor Components and Datapath



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

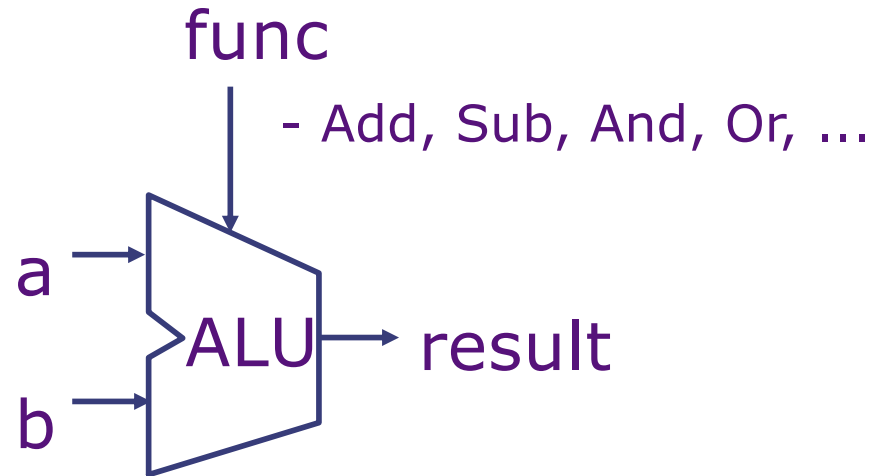
Processor function

- 
- Fetch the instruction at pc
 - Decode the instruction
 - Execute the instruction (compute the next state values for the register file and pc)
 - Access the memory if the instruction is a Ld or St
 - Update the register file and pc

First, we will examine the major components: register file, ALU, memory, decoder, execution unit; and then put it all together to build single-cycle and multicycle nonpipelined implementations of RISC-V

Arithmetic-Logic Unit (ALU)

ALU performs all the arithmetic and logical functions



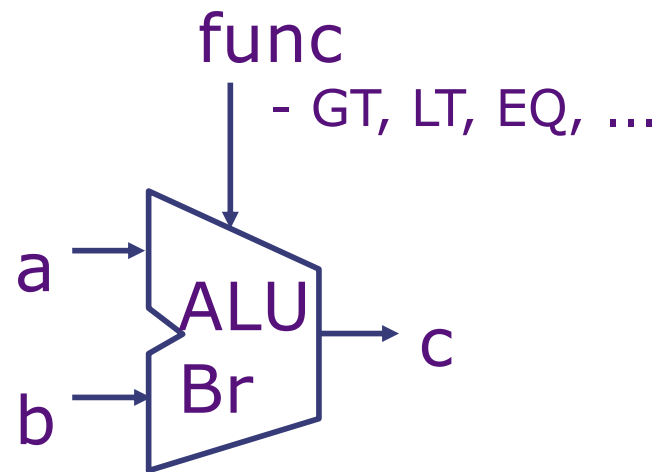
```
function Word alu(Word a, Word b,  
                  AluFunc func);  
typedef Bit#(32) Word;
```

```
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu,  
             Sll, Sra, Srl} AluFunc deriving(Bits, Eq);
```

This is what you implemented in Lab4

ALU for Branch Comparisons

Like ALU but
returns a Bool

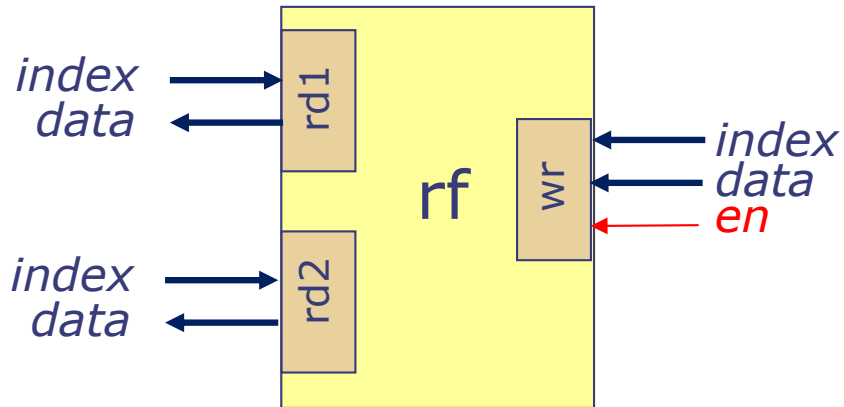


```
function Bool aluBr(Word a, Word b,  
                    BrFunc func);
```

```
typedef enum {Eq, Neq, Lt, Ltu, Ge, Geu}  
BrFunc deriving(Bits, Eq);
```

Register File

2 Read ports + 1 Write port



Registers can be read or written any time, so the guards are always true (not shown)

```
typedef Bit#(32) Word;
typedef Bit#(5) RIndx;

interface RFile2R1W;
    method Word rd1(RIndx index) ;
    method Word rd2(RIndx index) ;
    method Action wr (RIndx index, Word data);
endinterface
```

Register File implementation

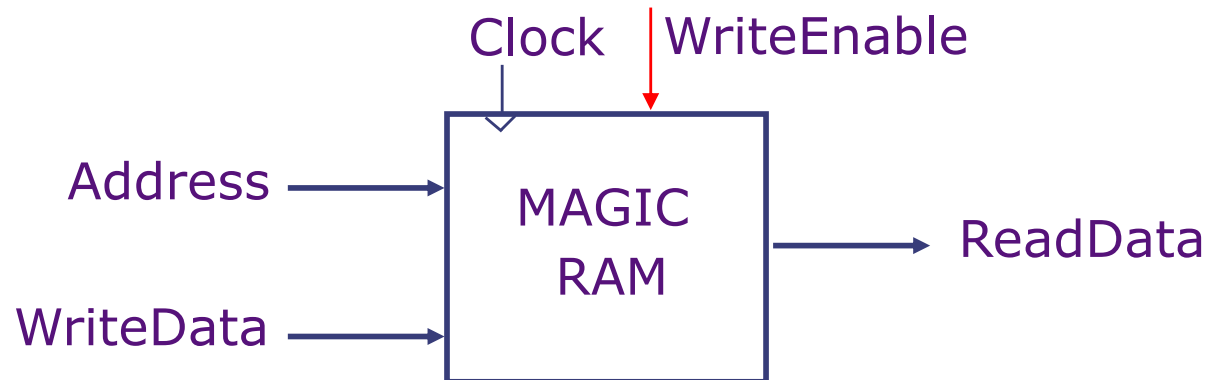
```
module mkRFile2R1W(RFile2R1W);  
  Vector#(32,Reg#(Word)) rfile <- replicateM(mkReg(0));  
  method Word rd1(RIndx rindx) = rfile[rindx];  
  method Word rd2(RIndx rindx) = rfile[rindx];  
  method Action wr(RIndx rindx, Word data);  
    if(rindx!=0) begin rfile[rindx] <= data; end  
  endmethod  
endmodule
```

Register 0 is hardwired to zero and cannot be written

All three methods of the register file can be called simultaneously, and in that case the read methods read the value already in the register file

$\{rd1, rd2\} < wr$

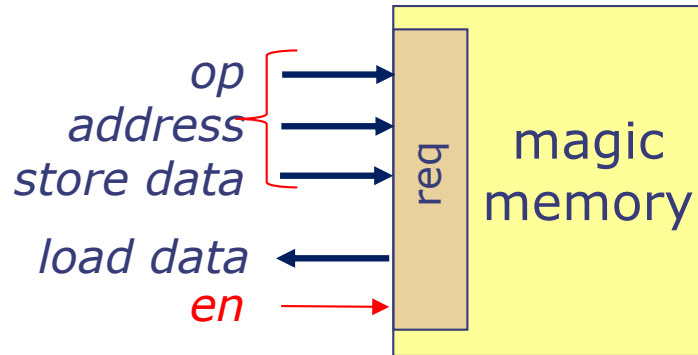
Magic Memory Model



- Reads and writes behave as in a register file (*not true for real SRAM or DRAM*)
 - Reads are combinational
 - Write, if enabled, is performed at the rising clock edge
- However, unlike a register file, there is only one port, which is used either for reading or for writing

Next lecture we will consider more realistic memory systems

Magic Memory Interface



Magic memory can be read or written any time, so the guards are always true (not shown)

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

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

```
// default word value  
Word dwv = 0;
```

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

ALU Instructions

Differ only in the ALU op to be performed

Instruction	Description	Execution
ADD rd, rs1, rs2	Add	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] + \text{reg}[\text{rs2}]$
SUB rd, rs1, rs2	Sub	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] - \text{reg}[\text{rs2}]$
SLL rd, rs1, rs2	Shift Left Logical	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \ll \text{reg}[\text{rs2}]$
SLT rd, rs1, rs2	Set if < (Signed)	$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] <_s \text{reg}[\text{rs2}]) ? 1 : 0$
SLTU rd, rs1, rs2	Set if < (Unsigned)	$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] <_u \text{reg}[\text{rs2}]) ? 1 : 0$
XOR rd, rs1, rs2	Xor	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \wedge \text{reg}[\text{rs2}]$
SRL rd, rs1, rs2	Shift Right Logical	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \gg_u \text{reg}[\text{rs2}]$
SRA rd, rs1, rs2	Shift Right Arithmetic	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \gg_s \text{reg}[\text{rs2}]$
OR rd, rs1, rs2	Or	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \text{reg}[\text{rs2}]$
AND rd, rs1, rs2	And	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \& \text{reg}[\text{rs2}]$

- These instructions are grouped in a category called OP with fields (func, rd, rs1, rs2) where func is the function for the ALU

ALU Instructions

with one Immediate operand

Instruction	Description	Execution
ADDI rd, rs1, immI	Add Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] + \text{immI}$
SLTI rd, rs1, immI	Set if < Immediate (Signed)	$\text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] <_s \text{immI}) ? 1 : 0$
SLTIU rd, rs1, immI	Set if < Immediate (Unsigned)	$\text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] <_u \text{immI}) ? 1 : 0$
XORI rd, rs1, immI	Xor Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \wedge \text{immI}$
ORI rd, rs1, immI	Or Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \text{immI}$
ANDI rd, rs1, immI	And Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \& \text{immI}$
SLLI rd, rs1, immI	Shift Left Logical Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \ll \text{immI}$
SRLI rd, rs1, immI	Shift Right Logical Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \gg_u \text{immI}$
SRAI rd, rs1, immI	Shift Right Arithmetic Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \gg_s \text{immI}$

- These instructions are grouped in a category called OPIMM with fields (func, rd, rs1, immI) where func is the function for the alu

Branch Instructions

differ only in the aluBr operation they perform

Instruction	Description	Execution
BEQ rs1, rs2, immB	Branch =	pc <= (reg[rs1] == reg[rs2]) ? pc + immB : pc + 4
BNE rs1, rs2, immB	Branch !=	pc <= (reg[rs1] != reg[rs2]) ? pc + immB : pc + 4
BLT rs1, rs2, immB	Branch < (Signed)	pc <= (reg[rs1] < _s reg[rs2]) ? pc + immB : pc + 4
BGE rs1, rs2, immB	Branch ≥ (Signed)	pc <= (reg[rs1] ≥ _s reg[rs2]) ? pc + immB : pc + 4
BLTU rs1, rs2, immB	Branch < (Unsigned)	pc <= (reg[rs1] < _u reg[rs2]) ? pc + immB : pc + 4
BGEU rs1, rs2, immB	Branch ≥ (Unsigned)	pc <= (reg[rs1] ≥ _u reg[rs2]) ? pc + immB : pc + 4

- These instructions are grouped in a category called BRANCH with fields (brFunc, rs1, rs2, immB) where brFunc is the function for aluBr

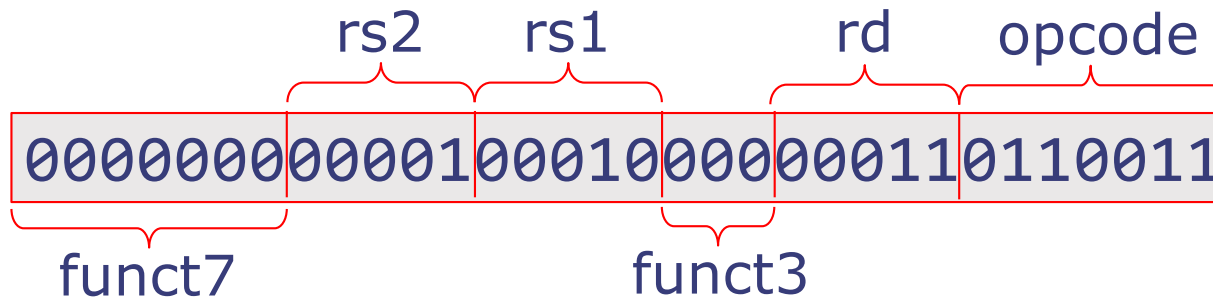
Remaining Instructions

Instruction	Description	Execution
JAL rd, immJ	Jump and Link	$\text{reg}[\text{rd}] \leq \text{pc} + 4$ $\text{pc} \leq \text{pc} + \text{immJ}$
JALR rd, immI(rs1)	Jump and Link Register	$\text{reg}[\text{rd}] \leq \text{pc} + 4$ $\text{pc} \leq \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$
LUI rd, immU	Load Upper Immediate	$\text{reg}[\text{rd}] \leq \text{immU}$
LW rd, immI(rs1)	Load Word	$\text{reg}[\text{rd}] \leq \text{mem}[\text{reg}[\text{rs1}] + \text{immI}]$
SW rs2, immS(rs1)	Store Word	$\text{mem}[\text{reg}[\text{rs1}] + \text{immS}] \leq \text{reg}[\text{rs2}]$

- Each of these instructions is in a category by itself and needs to extract different fields from the instruction
- LW and SW need to access memory for execution and thus, are required to compute an effective memory address

Decoding instructions

An Example



- What RISC-V instruction is represented by these 32 bits?

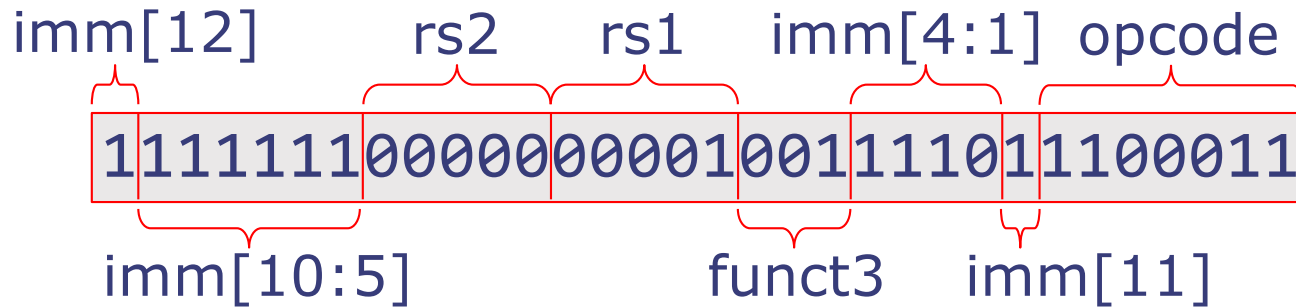
- Reference manual specifies the fields as follows:

- 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 can be complicated!



- What is this instruction?
- Reference manual specifies the fields as follows:
 - opcode = 1100011 => opCode branch, B-type encoding
 - funct3 = 001 => BNEQ
 - rs1 = 00001 => x1
 - rs2 = 00000 => x0
 - imm[12:1] = {1, 1, 111111, 1110};
 - imm[0] = 0; imm[31:13] are sign extended
=> imm[31:0] = -4

```
brAlu(rf.rd1(1), rf.rd2(0), Neq) ? pc <= pc + (-4)
                                   : pc <= pc + 4
```


Instruction decoder

- We need a function to extract the category and the various fields for each category from a 32-bit instruction
- Fields we have identified so far are:
 - Instruction category: OP, OPIMM, BRANCH, JAL, JALR, LUI, LOAD, STORE, Unsupported
 - Function for alu: aluFunc
 - Function for brAlu: brFunc
 - Register fields: rd, rs1, rs2
 - Immediate constants: immI(12), immB(12), immJ(20), immU(20), immS(12) *but* each is used as a 32-bit value with proper sign extension

Notice that no instruction has all the fields

Encoding Examples

- Immediate encodings

31	25	24	20	19	15	14	12	11	7	6	0		
funct7			rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]					rs1		funct3		rd		opcode		I-type
imm[11:5]			rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]			rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]									rd		opcode		U-type
imm[20 10:1 11 19:12]									rd		opcode		J-type

- Sample instruction encodings

imm[31:12]									rd		0110111		LUI
imm[20 10:1 11 19:12]									rd		1101111		JAL
imm[11:0]					rs1		000		rd		1100111		JALR
imm[12 10:5]			rs2		rs1		000		imm[4:1 11]		1100011		BEQ
imm[12 10:5]			rs2		rs1		001		imm[4:1 11]		1100011		BNE
imm[12 10:5]			rs2		rs1		100		imm[4:1 11]		1100011		BLT
imm[12 10:5]			rs2		rs1		101		imm[4:1 11]		1100011		BGE
imm[12 10:5]			rs2		rs1		110		imm[4:1 11]		1100011		BLTU
imm[12 10:5]			rs2		rs1		111		imm[4:1 11]		1100011		BGEU
imm[11:0]					rs1		010		rd		0000011		LW
imm[11:5]			rs2		rs1		010		imm[4:0]		0100011		SW
imm[11:0]					rs1		000		rd		0010011		ADDI
imm[11:0]					rs1		010		rd		0010011		SLTI
imm[11:0]					rs1		011		rd		0010011		SLTIU
imm[11:0]					rs1		100		rd		0010011		XORI

Decoded Instruction Type

```
typedef struct {
```

```
  IType
```

```
  AluFunc
```

```
  BrFunc
```

```
  RDst
```

```
  RIndx
```

```
  RIndx
```

```
  Word
```

```
} DecodedInst deriving(Bits, Eq);
```

```
typedef enum {OP, OPIMM, BRANCH, LUI, JAL, JALR, LOAD, STORE, Unsupported} 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} BrFunc deriving(Bits, Eq);
```

```
typedef struct {Bool valid; RIndx index;} RDst deriving (Bits);
```

Field names

Type of the Field names

Destination register 0 behaves like an Invalid destination

If dst is invalid, register file update is not performed

Single-Cycle Implementation

Putting it all together

```
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.index);
    let rVal2 = rf.rd2(dInst.src2.index);
    let eInst = execute(dInst, rVal1, rVal2, pc);
    // eInst fields: iType, dst, data, addr, nextPC
    updateState(eInst, pc, rf, dMem);
  endrule
endmodule
```

instantiate the state

extract the fields

read the register file

actions to update the processor state

compute values needed to update the processor state

Function execute

```
function ExecInst execute( DecodedInst dInst,
                          Word rVal1, Word rVal2, Word pc );
// extract from dInst: iType, aluFunc, brFunc, imm
// initialize eInst and its fields: data, nextPc, addr to dwv
case (iType) matches
  OP: begin data = alu(rVal1, rVal2, aluFunc);
        nextPc = pc+4; end
  OPIMM: begin data = alu(rVal1, imm, aluFunc);
           nextPc = pc+4; end
  BRANCH: begin nextPc = aluBr(rVal1, rVal2, brFunc) ?
                pc+imm : pc+4; end
  LUI: begin data = imm; nextPc = pc+4; end
  JAL: begin data = pc+4; nextPc = pc+imm; end
  JALR: begin data = pc+4; nextPc = (rVal1+imm) & ~1; end
  LOAD: begin addr = rVal1+imm; nextPc = pc+4; end
  STORE: begin data = rVal2; addr = rVal1+imm;
           nextPc = pc+4; end
endcase // assign to eInst;
endfunction
```

To zero out
the LSB

Updating the state

```
//Extract fields of eInst: data, addr, dst;
  let data = eInst.data;
  // 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;
```

An elegant coding trick: you can package these update actions in a procedure that returns these actions

Update Function

a function that returns an action

```
function Action updateState(ExecInst eInst, Reg#(Word) pc,  
                               RFile2R1W rf, MagicMemory dMem);
```

```
return (action
```

we are passing the register, not its
value, as a parameter

```
//Extract fields of eInst: data, addr, dst;  
let data = eInst.data;  
// 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.data, data);  
// pc update  
pc <= eInst.nextPc;
```

```
endaction);  
endfunction
```

It can be called as follows:

```
updateState(eInst, pc, rf, dMem);
```

Processor Interface

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

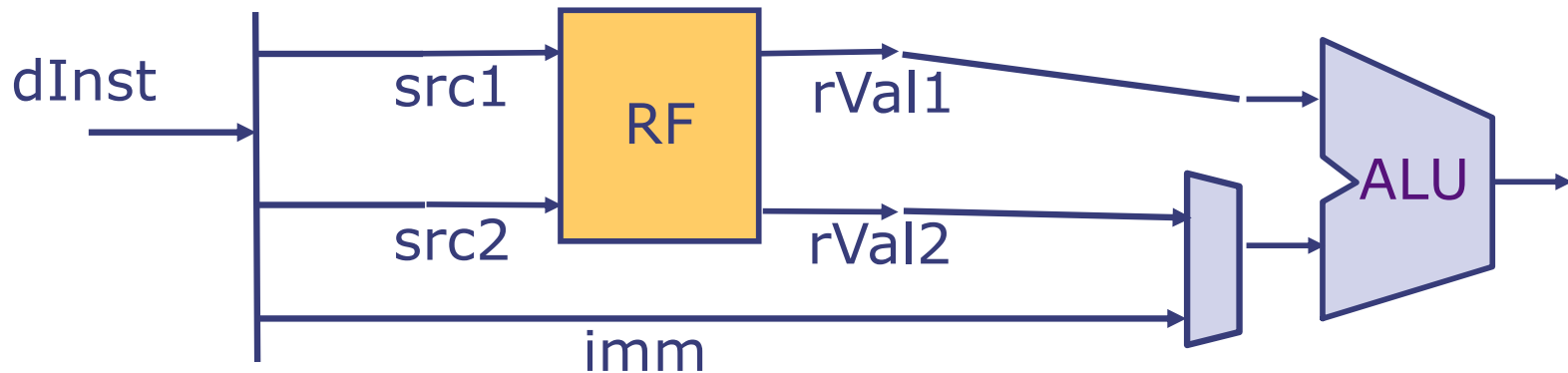


- For testing, processor is connected to a host computer* which can read and write the memory of the processor directly
- The processor's memory is preloaded with program and data; it always starts at pc=0
- When the program terminates it writes a 0 in a predetermined location and stops the simulation
 - 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, it's interface is Empty!

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

Understanding generated hardware (the real datapath)



- Not all instructions have both `src1` and `src2` fields but there is no harm/cost in reading unused registers; we never use results generated by the use of undefined fields

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

- When the same function is called with two different arguments, a mux is generated automatically

Understanding generated hardware

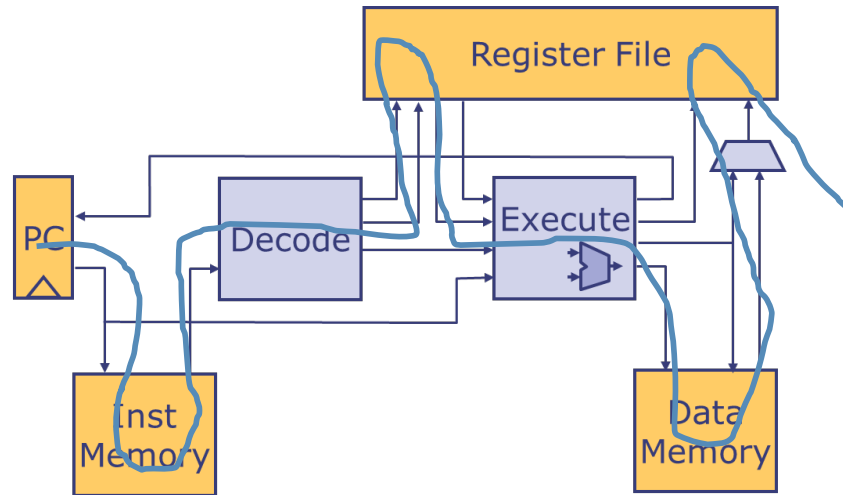
- *continued*

```
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; ones needs to check the output to determine this
 - 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*

Single-cycle processor

Clock Speed



- Clock speed depends upon the longest combinational path between two state elements

- $t_{\text{Clock}} > t_M + t_{\text{DEC}} + t_{\text{RF}} + t_{\text{ALU}} + t_M + t_{\text{WB}}$ *slo...w*

- We can breakdown the execution in multiple phases and execute each phase in one cycle

- $t_{\text{Clock}} > \max \{t_M, t_{\text{DEC}} + t_{\text{RF}}, t_{\text{ALU}}, t_M, t_{\text{WB}}\}$

- Clock will be faster but each instruction will take multiple cycles!

... but some times multicycle implementations are unavoidable

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

Request/Response methods must be called from separate rules

```
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);
```

```
rule doFetch if (state == Fetch);
  m.req(MemReq{op:Ld, addr:pc, data:dwv});
  state <= Execute; x <= ...;
endrule
```

default value



```
rule doExecute if (state == Execute);
  let inst <- mem.resp;
  ... decode(inst);... read x
endrule
```

Often we need to hold the state of a partially executed instruction in new state elements between cycles

Processor with realistic memory multicycle

```
module mkProcMulticycle(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)
```

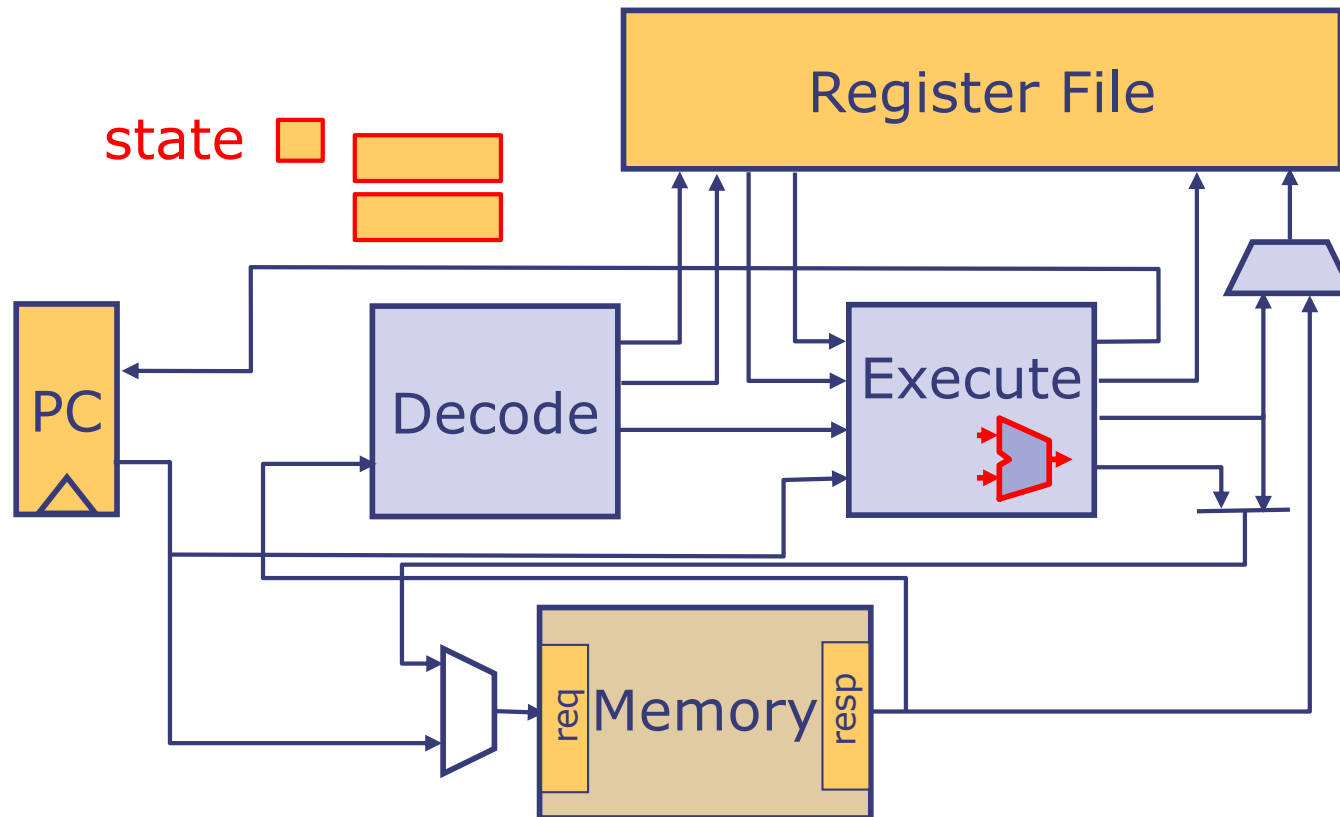
```
rule doLoadWait if (state == LoadWait);
```

```
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
```


Reducing cycle counts further

```
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;
```

Any disadvantage?

Code to 1. execute all instructions except

memory and multicycle instructions; ~~go to Fetch~~

Or 2. initiate memory access

initiate fetch

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);
```

initiate fetch

Code to wait for MC value, update rf, ~~go to Fetch~~

initiate fetch

```
endmodule
```

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 how memory systems are organized internally