6.004 Recitation Problems
L14 – RISC-V Processor

Single-Cycle RISC-V Processor

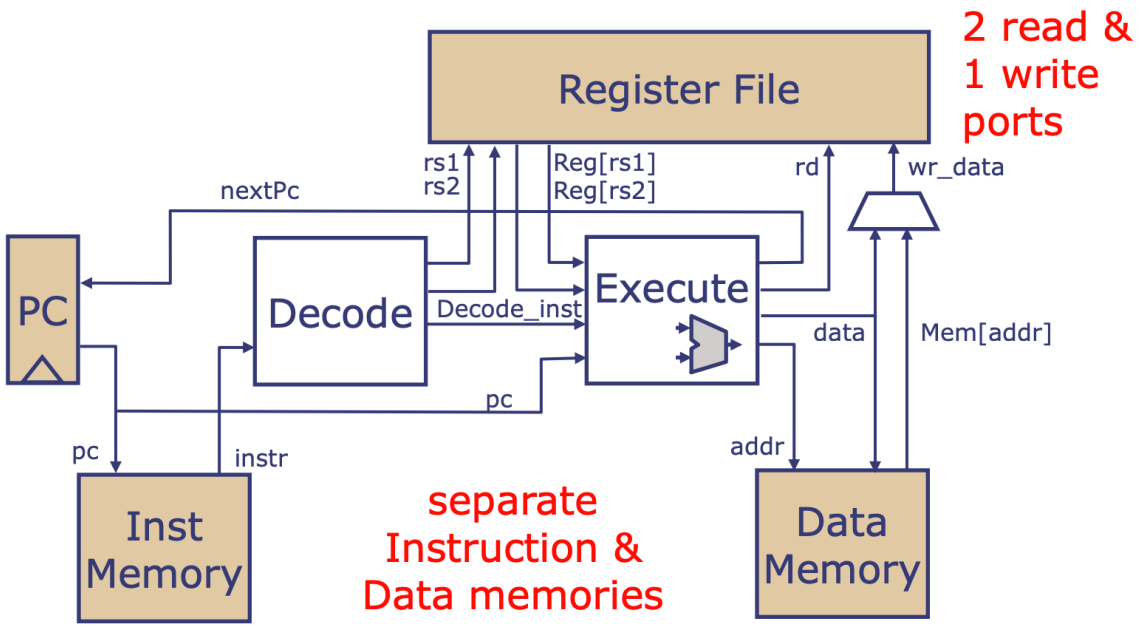


Diagram 1: Abstract Version

- Contains all major components: PC Register, Instruction and Data memories, Decode logic, Execute logic (including the ALU), and Register file
- Skips some details in the wiring of signals to increase readability

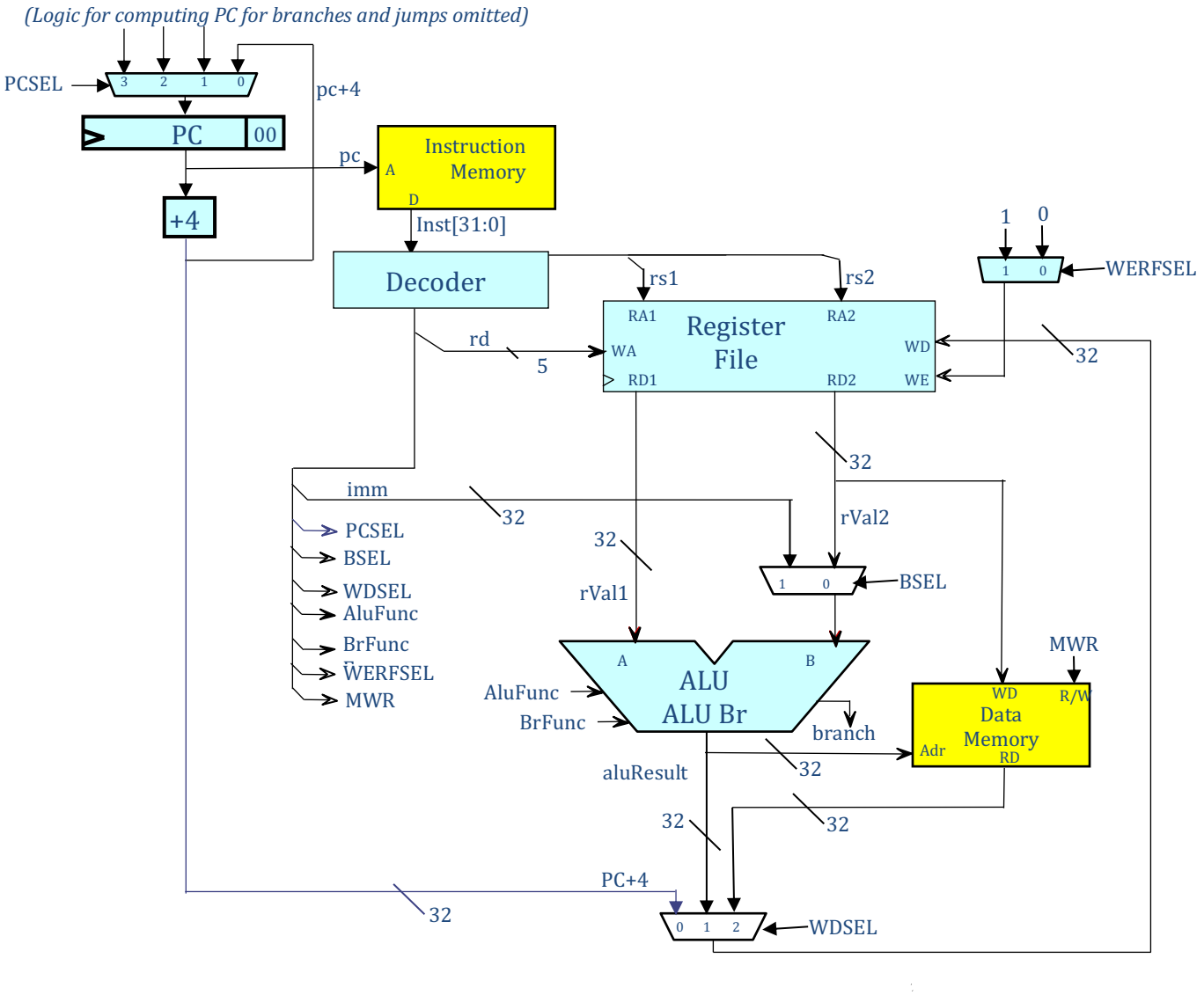


Diagram 2: Detailed Version

- Also contains all of the major components, as well as additional details like muxes on inputs, descriptions of various signal contents and bit widths

- Both diagrams are equivalent! Differ only in level of detail

RISC-V Processor: Components

Refer to the 6.004 ISA Reference Tables (Website > Resources) for details about each instruction.

ProcTypes

```
typedef Bit#(32) Word;

// Branch function enumeration
typedef enum {
    Eq,
    Neq,
    Lt,
    Ltu,
    Ge,
    Geu,
    Dbr
} BrFunc;


typedef enum {
    OP,
    OPIMM,
    BRANCH,
    LUI,
    JAL,
    JALR,
    LOAD,
    STORE,
    AUIPC,
    Unsupported
} IType;
```

```
// Return type for decode()
typedef struct {
    IType iType;
    AluFunc aluFunc;
    BrFunc brFunc;
    Maybe#(RIndx) dst;
    RIndx src1;
    RIndx src2;
    Word imm;
} DecodedInst;

// Register Index Type
typedef Bit#(5) RIndx;

// Register File writes
typedef struct {
    RIndx index;
    Word data;
} RegWriteArgs;

// Return type for execute()
typedef struct {
    IType iType;
    Maybe#(RIndx) dst;
    Word data;
    Word addr;
    Word nextPc;
} ExecInst;

// Memory writes
typedef struct {
    Word addr;
    Word data;
} MemWriteReq;
```

```
    // Opcode
    Bit#(7) opOpImm  = 7'b0010011;
    Bit#(7) opOp     = 7'b0110011;
    Bit#(7) opLui    = 7'b0110111;
    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;
    Bit#(7) opAuipc  = 7'b0010111;

    // funct3 - ALU
    Bit#(3) fnADD   = 3'b000;
    Bit#(3) fnSLL   = 3'b001;
    Bit#(3) fnSLT   = 3'b010;
    Bit#(3) fnSLTU  = 3'b011;
    Bit#(3) fnXOR   = 3'b100;
    Bit#(3) fnSR    = 3'b101;
    Bit#(3) fnOR    = 3'b110;
    Bit#(3) fnAND   = 3'b111;
```

```
    // funct3 - Branch
    Bit#(3) fnBEQ   = 3'b000;
    Bit#(3) fnBNE   = 3'b001;
    Bit#(3) fnBLT   = 3'b100;
    Bit#(3) fnBGE   = 3'b101;
    Bit#(3) fnBLTU  = 3'b110;
    Bit#(3) fnBGEU  = 3'b111;
    // funct3 - Load
    Bit#(3) fnLW    = 3'b010;
    Bit#(3) fnLB    = 3'b000;
    Bit#(3) fnLH    = 3'b001;
    Bit#(3) fnLBU   = 3'b100;
    Bit#(3) fnLHU   = 3'b101;
    // funct3 - Store
    Bit#(3) fnSW    = 3'b010;
    Bit#(3) fnSB    = 3'b000;
    Bit#(3) fnSH    = 3'b001;
    // funct3 - JALR
    Bit#(3) fnJALR  = 3'b000;
```

Register File

```
module RegisterFile;
    Vector#(32, Reg#(Word)) regs(0);

    method Word rd1(RIndx rindx) = regs[rindx];
    method Word rd2(RIndx rindx) = regs[rindx];

    input Maybe#(RegWriteArgs) wr default = Invalid;

    rule rfWrite;
        if (isValid(wr)) begin
            RegWriteArgs rwd = fromMaybe(?, wr);
            if (rwd.index != 0)
                regs[rwd.index] <= rwd.data;
        end
    endrule
endmodule
```

Decode

```
function DecodedInst decode(Bit#(32) inst);
    let opcode = inst[6:0];
    let funct3 = inst[14:12];
    let funct7 = inst[31:25];
    let dst    = inst[11:7];
    let src1   = inst[19:15];
    let src2   = inst[24:20];

    Maybe#(RIndx) validDst = Valid(dst);
    Maybe#(RIndx) dDst = Invalid; // default value
    RIndx dSrc = 5'b0;

    // DEFAULT VALUES - Use the following for your default values:
    // dst: dDst, src1: dSrc, src2: dSrc, imm: immD, BrFunc: Dbr, AluFunc: ?

    // We have provided a default value and done immB for you.
    Word immD32 = signExtend(1'b0); // default value
    Bit#(12) immB = { inst[31], inst[7], inst[30:25], inst[11:8] };
    Word immB32 = signExtend({immB, 1'b0});
    Bit#(20) immU = 0; // TODO
    Word immU32 = 0; // TODO
    Bit#(12) immI = 0; // TODO
    Word immI32 = 0; // TODO
    Bit#(20) immJ = 0; // TODO
    Word immJ32 = 0; // TODO
    Bit#(12) immS = 0; // TODO
    Word immS32 = 0; // TODO

    DecodedInst dInst = unpack(0);
    dInst.iType = Unsupported; // unsupported by default

    case (opcode)
        opAuipc: begin
            dInst = DecodedInst {
                iType: AUIPC,
                dst: validDst,
                src1: dSrc,
                src2: dSrc,
                imm: immU32,
                brFunc: Dbr,
                aluFunc: ?
            };
        end
        opLui: // TODO
```

Decode (continued)

```
        opOpImm: begin
            dInst.iType = OPIMM;
            dInst.src1  = src1;
            dInst.imm   = immI32;
            dInst.dst   = validDst;

            case (funct3)
                fnAND : dInst.aluFunc = And; // Decode ANDI instructions
                fnOR  : dInst.iType = Unsupported; // TODO
                fnXOR : dInst.iType = Unsupported; // TODO
                fnADD : dInst.iType = Unsupported; // TODO
                fnSLT : dInst.iType = Unsupported; // TODO
                fnSLTU: dInst.iType = Unsupported; // TODO
                fnSLL : case (funct7)
                    7'b0000000: dInst.aluFunc = Sll;
                    // Otherwise we must say the instruction is invalid:
                    default:    dInst.iType = Unsupported;
                endcase
                fnSR  : // TODO
                    dInst.iType = Unsupported;
                    default: dInst.iType = Unsupported;
            endcase
        end
        opOp: // TODO
        opBranch: // TODO
        opJal: // TODO
        opLoad: // TODO
        opStore: // TODO
        opJalr: // TODO
    endcase
    return dInst;
endfunction
```

Branch ALU (Execute.ms)

```
function Bool aluBr(Word a, Word b, BrFunc brFunc);
    Bool res = case (brFunc)
        Eq:     (a == b);
        Neq:    (a != b);
        Lt:     signedLT(a, b);
        Ltu:    (a < b);
        Ge:     signedGE(a, b);
        Geu:    (a >= b);
        default: False;
    endcase;
    return res;
endfunction
```

Execute

```
function ExecInst execute(DecodedInst dInst,Word rVal1,Word rVal2,Word pc);
    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;
    let aluVal2 = dInst.iType == OPIMM ? imm : rVal2;

    Word data = case (dInst.iType)
        AUIPC:     pc + imm;
        LUI:       0; // TODO
        OP, OPIMM: 0; // TODO
        JAL, JALR: 0; // TODO
        STORE:     0; // TODO
        default:   0;
    endcase;

    Word nextPc = case (dInst.iType)
        BRANCH: 0; // TODO Replace 0 with the correct expression
        JAL:    0; // TODO Replace 0 with the correct expression
        JALR: (rVal1 + imm) & ~1; // "& ~1" clears the bottom bit.
        default: pc + 4;
    endcase;

    Word addr = 0; // TODO Replace 0 with the correct expression

    return ExecInst{iType: dInst.iType, dst: dInst.dst, data: data,
                    addr: addr, nextPc: nextPc};
endfunction
```

Magic Memory

```
module MagicMemory;
    // 64 KB magic memory array
    MagicMemoryArray magicMem("mem.vmh");

    method Word read(Word addr) = magicMem.sub(truncate(addr >> 2));

    input Maybe#(MemWriteReq) write default = Invalid;

    rule doWrite;
        if (isValid(write)) begin
            MemWriteReq req = fromMaybe(?, write);
            if (req.addr == 'h4000_0000) begin
                // Write character to stdout
                $write("%c", req.data[7:0]);
            end else if (req.addr == 'h4000_0004) begin
                // Write integer to stdout
                $write("%0d", req.data);
            end else if (req.addr == 'h4000_1000) begin
                // Exit simulation
                if (req.data == 0) begin
                    $display("PASSED");
                end else begin
                    $display("FAILED %0d", req.data);
                end
                $finish;
            end else begin
                // Write memory array
                magicMem.upd = ArrayWriteReq{idx: truncate(req.addr >> 2),
                                             data: req.data};
            end
        end
    endrule
endmodule
```

Processor

```
module Processor;
    Reg#(Word) pc(0);
    RegisterFile rf;
    MagicMemory iMem; // Memory for loading instructions
    MagicMemory dMem; // Memory for loading and storing data

    rule doSingleCycle;
        // Load the instruction from instruction memory (iMem)
        Word inst = 0; // TODO

        // Decode the instruction
        DecodedInst dInst = unpack(0); // TODO

        // Read the register values used by the instruction
        Word rVal1 = 0; // TODO
        Word rVal2 = 0; // TODO

        // Compute all outputs of the instruction
        ExecInst eInst = unpack(0); // TODO

        if (eInst.iType == LOAD) begin
            // TODO: Load from data memory (dMem) if needed
        end else if (eInst.iType == STORE) begin
            // TODO: Store to data memory (dMem) if needed
        end

        if (isValid(eInst.dst)) begin
            // TODO: Write to a register if the instruction requires it
        end

        // TODO: Update pc to the next pc
    endrule
endmodule
```

Problem 1.

Decode the following 32-bit RISC-V instructions:

1. `0100000 00001 00100 000 00011 0110011`

   0100000 rs2 rs1 000 rd 0110011

   SUB x3, x4, x1

   The first step is to check the opcode, or inst[6:0]. In this case, 0110011 tells us that this is an OP type instruction. Next, 'funct3', or inst[14:12], informs us that in particular we have an ADD or SUB. Finally, since inst[30] = 1 this is a sub rather than add operation. Now, we can convert the rs1, rs2, and rd parts into 4, 1, and 3 respectively, giving x4, x1, and x3 as the operand registers.

2. `0100000 00101 00010 101 00111 0010011`

   0100000 shamt rs1 101 rd 0010011

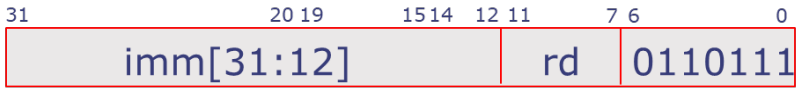
   SRAI x7, x2, 5

   Again, the opcode informs us the broad instruction class, this time an OPIMM. Next, 'funct3' tells us that we're looking at a right shift, and 'funct7' reveals that it's an arithmetic one. Note that shift instructions only use the bottom 5 bits of the immediate as the actual shift amount, so from that we get the value 5.

Problem 2. ★

The RISC-V `LUI` instruction, Load Upper Immediate, loads a 20-bit immediate into the upper 20 bits of the specified destination register (and sets the lower 12 bits to zero). What modifications to the datapath are needed to implement the `LUI` instruction?

LUI rd, immU reg[rd] <= immU << 12



Modifications needed:

1. Extract 20-bit immediate from instruction [in decode]
2. Shift left 12 bits (just add 12 zeros on the right) [in decode]
3. Extend final datapathmux (WDSEL) to take new value

In terms of implementing the processor in Minispec, we'll need the Execute module to output that immediate that was shifted in the Decode module

```
function ExecInst execute(DecodedInst dInst,Word rVal1,Word rVal2,Word pc);
    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;
    let aluVal2 = dInst.iType == OPIMM ? imm : rVal2;

    Word data = case (dInst.iType)
        AUIPC:     pc + imm;
        LUI:       imm;
        OP, OPIMM: 0; // TODO
        JAL, JALR: 0; // TODO
        STORE:     0; // TODO
        default:   0;
    endcase;
```

Problem 3. ★

We want to add the following instruction to the RISC-V ISA:

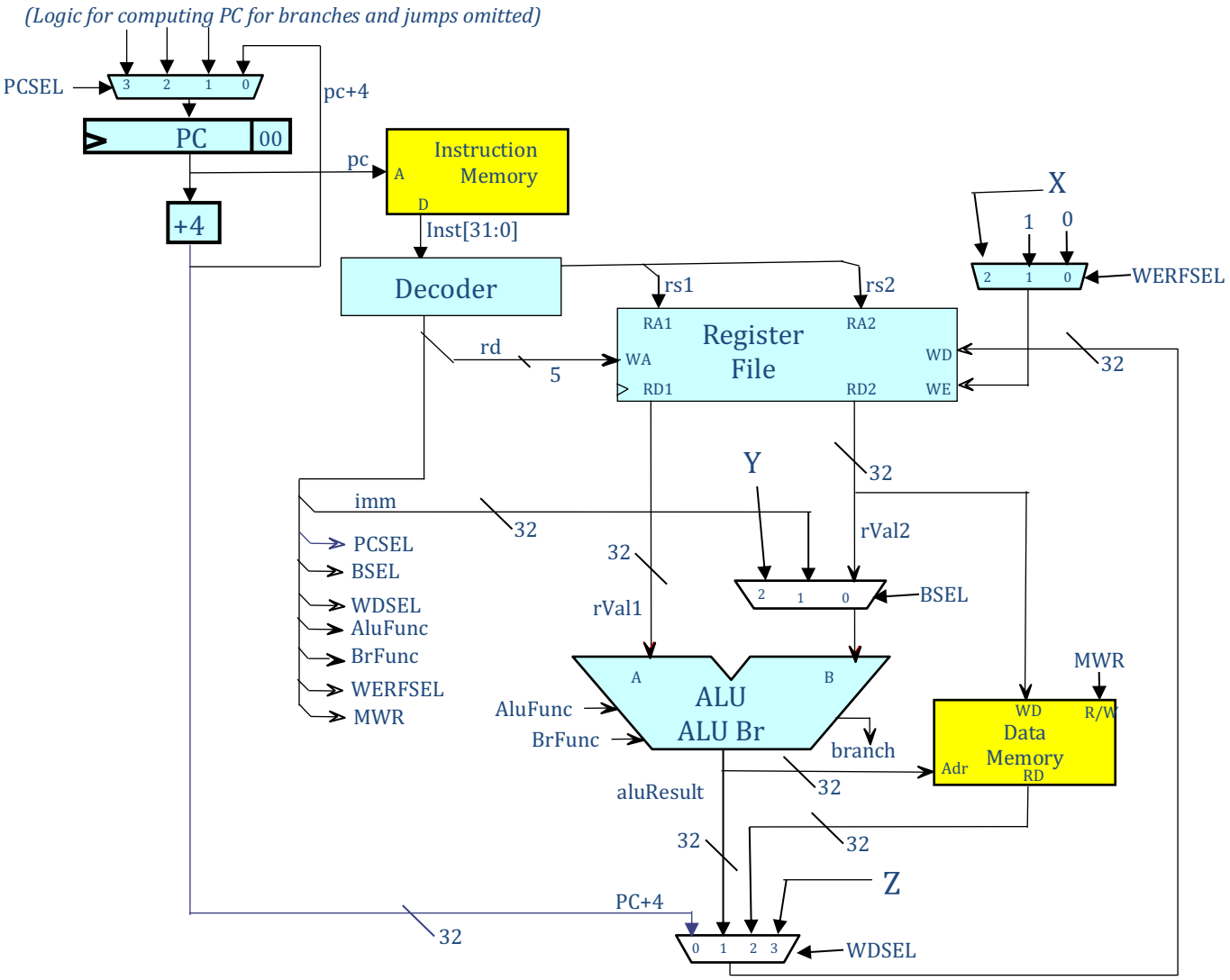
```
cmvz rd, rs1, rs2
```

This new `cmvz` instruction is a *conditional move*: it checks the contents of source register `rs1`, and if it's 0, copies the contents of source register `rs2` into `rd`. In other words:

```
if (reg[rs1] == 0) begin
  reg[rd] <= reg[rs2];
end
```

We would like to update the hardware diagram of the single-cycle processor to support this new instruction, reusing the existing **branch ALU** comparison logic to check if `rs1` is `0`. In the diagram below, we have added extra inputs to the BSEL and WDSEL, and WERFSEL muxes to support reusing the branch ALU in this way.

Assume that when the decoder decodes CMVZ, it outputs BrFunc = Eq, BSEL = 2, WDSEL = 3, and WERFSEL = 2. It also outputs PCSEL = 0 so that the next value of PC will be PC + 4.


(Logic for computing PC for branches and jumps omitted)
PCSEL
3 2 1 0
pc+4
PC
00
+4
pc
A
Instruction Memory
D
Inst[31:0]
Decoder
rs1
rs2
X
1 0
2 1 0
WERFSEL
RA1
RA2
WA
Register File
WD
RD1
RD2
WE
rd
5
32
Y
32
imm
32
rVal2
PCSEL
BSEL
WDSEL
AluFunc
BrFunc
WERFSEL
MWR
32
rVal1
2 1 0
BSEL
A
B
ALU
ALU Br
AluFunc
BrFunc
branch
MWR
WD
R/W
Data Memory
Adr
RD
aluResult
32
32
32
Z
PC+4
32
0 1 2 3
WDSEL

Summary of some of the relevant processor components in the above diagram:

- One register in the register file can be written to each cycle by providing the register index in WA and the data to write in WD. The data is only written if the write enable signal, WE, is 1.
- The ALU receives two inputs, A and B. It computes an arithmetic operation on them specified by AluFunc, outputting the 32-bit result to aluResult, and a comparison specified by BrFunc, outputting the 1-bit result to branch.

(a) For each of the missing mux inputs X, Y, and Z, write down either a constant or the name of a different wire or port in the processor hardware diagram that should be connected to each input, so that the processor executes CMVZ correctly.

X: _____branch_____

Y: _____0_____

Z: _____rVal2_____

By setting Y to 0, we make sure that the Branch ALU that is being reused properly compares rs1 to 0. As a result, we can use the 'branch' signal at X to use this result to decide if rd should have its value updated. The value we're supposed to write it rVal2, so that's what Z is set to.

(b) We would like to check that our processor can still decode and execute the old RISC-V instructions correctly. Suppose the processor is decoding an OPIMM instruction, such as ADDI. For each of the select signals we changed (shown in **bold** in the diagram), write down the correct signals that the decoder should produce for such an instruction.

Decoded value of BSEL: ____1_____

Decoded value of WDSEL: ____1_____

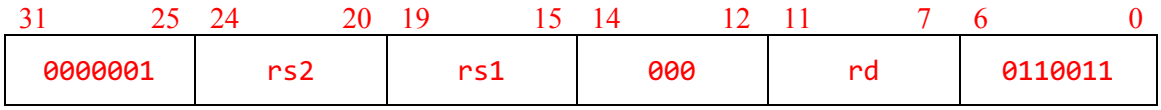
Decoded value of WERFSEL: ____1_____

OPIMM instructions operate on the 'immI' immediate, hence BSEL being 1. Additionally, we do want to write the result back to a register, and in particular we're writing the output of the ALU, so WDSEL and WERFSEL are also both 1.

Problem 4.

(a) The RISC-V `MUL` instruction multiplies two 32-bit values together and places the **lower** 32 bits of the product in the destination register. It has an opcode of `7'b0110011`, which is the same as the other OP type instructions, and a funct7 of `7'b0000001`, and a funct3 of `3'b000`. What changes to the datapath would be needed to support the `MUL` instruction? Use the combinational multiplier from Lecture 12.

MUL rd, rs1, rs2 reg[rd] <= (reg[rs1] × reg[rs2])[31:0]



Modifications needed:

1. Add multiplier to ALU [in execute]
2. Select lower 32 bits for writeback data

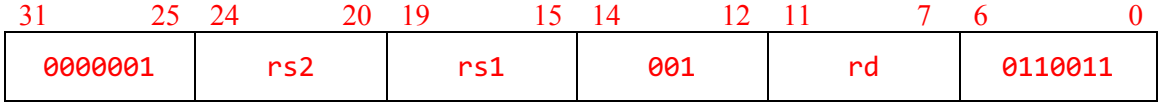
Adding this to the Minispec Execute module would look something like this:

```
function ExecInst execute(DecodedInst dInst,Word rVal1,Word rVal2,Word pc);
    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;
    let aluVal2 = dInst.iType == OPIMM ? imm : rVal2;
    Bit#(64) product = multiply_by_adding(rVal1, rVal2);

    Word data = case (dInst.iType)
        // other cases not shown
        OP: case (funct7)
            7'b0000001: case (funct3)
                3'b000: product[31:0];
                default: ?;
                endcase;
            default: ?;
            endcase;
        default: 0;
    endcase;
```

(b) The RISC-V `MULH` instruction performs the same multiplication in (a), but returns the **upper** 32 bits of the product in the destination register. It has the same opcode and funct7 codes, but its funct3 is `3'b001`. What modifications to 3(a) would be needed to support the `MULH` instruction?

MULH rd1, rs1, rs2 reg[rd] <= (reg[rs1] × reg[rs2])[63:32]



Modifications needed:

1. Same as MUL, except select higher 32 bits for writeback data with multiplexer.
2. Add the following to `case (funct3)` above: `3'b001: product[63:32];`

This has a slightly different implementation:

```
function ExecInst execute(DecodedInst dInst,Word rVal1,Word rVal2,Word pc);
    let imm = dInst.imm;
    let brFunc = dInst.brFunc;
    let aluFunc = dInst.aluFunc;
    let aluVal2 = dInst.iType == OPIMM ? imm : rVal2;
    Bit#(64) product = multiply_by_adding(rVal1, rVal2);

    Word data = case (dInst.iType)
        // other cases not shown
        OP: case (funct7)
            7'b0000001: case (funct3)
                3'b001: product[63:32];    ← HERE
                default: ?;
                endcase;
            default: ?;
            endcase;
        default: 0;
    endcase;
```

Problem 5. ★

Add a branch if greater-than (`BGT`) instruction to the provided RISC-V processor. The instruction encoding should match other branch instructions, but have `funct3 = 3'b010`.

```
Bit#(7) opBranch = 7'b1100011;
Bit#(3) fnGt = 3'b010;          ← Required funct3

function DecodedInst decode(Bit#(32) inst);
    let opcode = inst[6:0];
    let funct3 = inst[14:12];
    let funct7 = inst[31:25];
    let dst    = inst[11:7];
    let src1   = inst[19:15];
    let src2   = inst[24:20];

    Maybe#(RIndx) validDst = Valid(dst);
    Maybe#(RIndx) dDst = Invalid; // default value
    RIndx dSrc = 5'b0;

    // DEFAULT VALUES - Use the following for your default values:
    // dst: dDst, src1: dSrc, src2: dSrc, imm: immD, BrFunc: Dbr, AluFunc: ?

    // We have provided a default value and done immB for you.
    Word immD = signExtend(1'b0); // default value
    // decode branch immediate
    Word immB = signExtend({inst[31],inst[7],inst[30:25],inst[11:8],1'b0});
    Word immU = 0; // TODO
    Word immI = 0; // TODO
    Word immJ = 0; // TODO
    Word immS = 0; // TODO

    DecodedInst dInst = unpack(0);
    dInst.iType = Unsupported; // unsupported by default

  case (opcode)
       // Lots of omitted code from Lab 6
        opBranch: // TODO
               case (funct3):
                      // put together all the fields of dInst
                      fnGt: dInst = DecodedInst{dst: Invalid, src1: src1,
                      src2: src2, imm: immB, brFunc: Gt, aluFunc: ?,
                      iType: BRANCH };

        // Lots of omitted code from Lab 6
    endcase
    return dInst;
endfunction
```

```
// Branch function enumeration
typedef enum {Eq, Neq, Lt, Ltu, Ge, Geu, Gt, Dbr} BrFunc;

function Bool aluBr(Word a, Word b, BrFunc brFunc);
    Bool res = case (brFunc)
        Eq:     (a == b);
        Neq:    (a != b);
        Lt:     signedLT(a, b);
        Ltu:    (a < b);
        Ge:     signedGE(a, b);
        Geu:    (a >= b);
        Gt:     signedGT(a, b);     ← Using a new 'signedGT' function
        default: False;
    endcase;
    return res;
endfunction

function Bool signedGT(Word a, Word b);
    Int#(32) aInt = unpack(a);
    Int#(32) bInt = unpack(b);
    return aInt > bInt;
endfunction
```

Problem 6.

Assume that `aluBr` has been replaced with the new branch ALU function, `newAluBr`, shown below. This new branch ALU is controlled by two control signals: `newBrFunc` and `negate`. When the result of this function is true, the next PC is going to be computed as `pc + imm`.

```
typedef enum {Eq, Lt, Ltu} NewBrFunc;

function Bool newAluBr(Word a, Word b, NewBrFunc newBrFunc, Bool negate);
    Bool res = case (newBrFunc)
        Eq:     (a == b);
        Lt:     signedLT(a, b);
        Ltu:    (a < b);
    endcase;
    return negate ? !res : res;
endfunction
```

A) Fill in the decoding table below to specify what the control signals should be for each funct3. Write an "X" in the table for entries that don't matter. (Use the ISA reference card from the course website.)

funct3	newBrFunc	negate	Instruction
3'b000	Eq	False	BEQ
3'b001	Eq	True	BNE
3'b010	X	X	
3'b011	X	X	
3'b100	Lt	False	BLT
3'b101	Lt	True	BGE
3'b110	Ltu	False	BLTU
3'b111	Ltu	True	BGEU

Problem 7. (Quiz Problem 6 From Spring 20)

Giuseppe is writing a large RISC-V assembly program, and is tired of getting confused by how to properly maintain the stack pointer **sp**. He's heard that other ISAs have **push** and **pop** instructions, which handle both allocating/freeing the space for a word on the stack, and storing it from/loading it to a register. Giuseppe decides that he'd like to implement them in the RISC-V ISA. Their syntaxes are as follows:

```
push rs2                pop rd
```

And the following is a Python-like description of how each works:

```
push:
        reg[sp] = reg[sp] – 4
        Mem[reg[sp]] = reg[rs2]

pop:
        reg[rd] = Mem[reg[sp]]
        reg[sp] = reg[sp] + 4
```

(A) (4 points) Giuseppe hopes to be able to simply add some new signals as inputs to the selection muxes in the existing RISC-V processor diagram. For each instruction being added, can this be done with **the existing processor components as described in lecture and shown on the following page? If not, describe the limitation that prevents it.**

(Label: 6A_push) Can push be implemented with existing components?

Yes, the rs1 port of the register file can be used to read reg[sp] and the ALU can be used to subtract 4 from it. This value is sent to the Memory as an address and reg[rs2] is sent to the Memory as data.

(Label: 6A_pop) Can pop be implemented with existing components?

No, the proposed `pop` instruction needs to write to two different registers but our register file only has one write port.

Giuseppe's friend Jeffrey tells him that the **push** and **pop** instructions can cause issues when pipelining his processor later on. Giuseppe, knowing nothing about processor pipelining, decides to take his word for it and find a different solution to his problem.

His goal now is to remove a large amount of the uses of stack, so that he doesn't have as many opportunities to become confused. The main limitation that causes him to use the stack so much is that he's running out of registers to hold temporary values. In order to overcome this, he wants to combine multiple steps into a single instruction. In particular, he wants to combine the `AND`ing of two registers and `XOR`ing the result with an immediate into a single instruction. All immediates he uses are 10 bits or less, so he comes up with the following instruction syntax, definition, and encoding:

`andxori rd, rs1, rs2, imm`

`andxori:`
reg[rd] = ((reg[rs1] & reg[rs2]) ^ signExtend(imm))

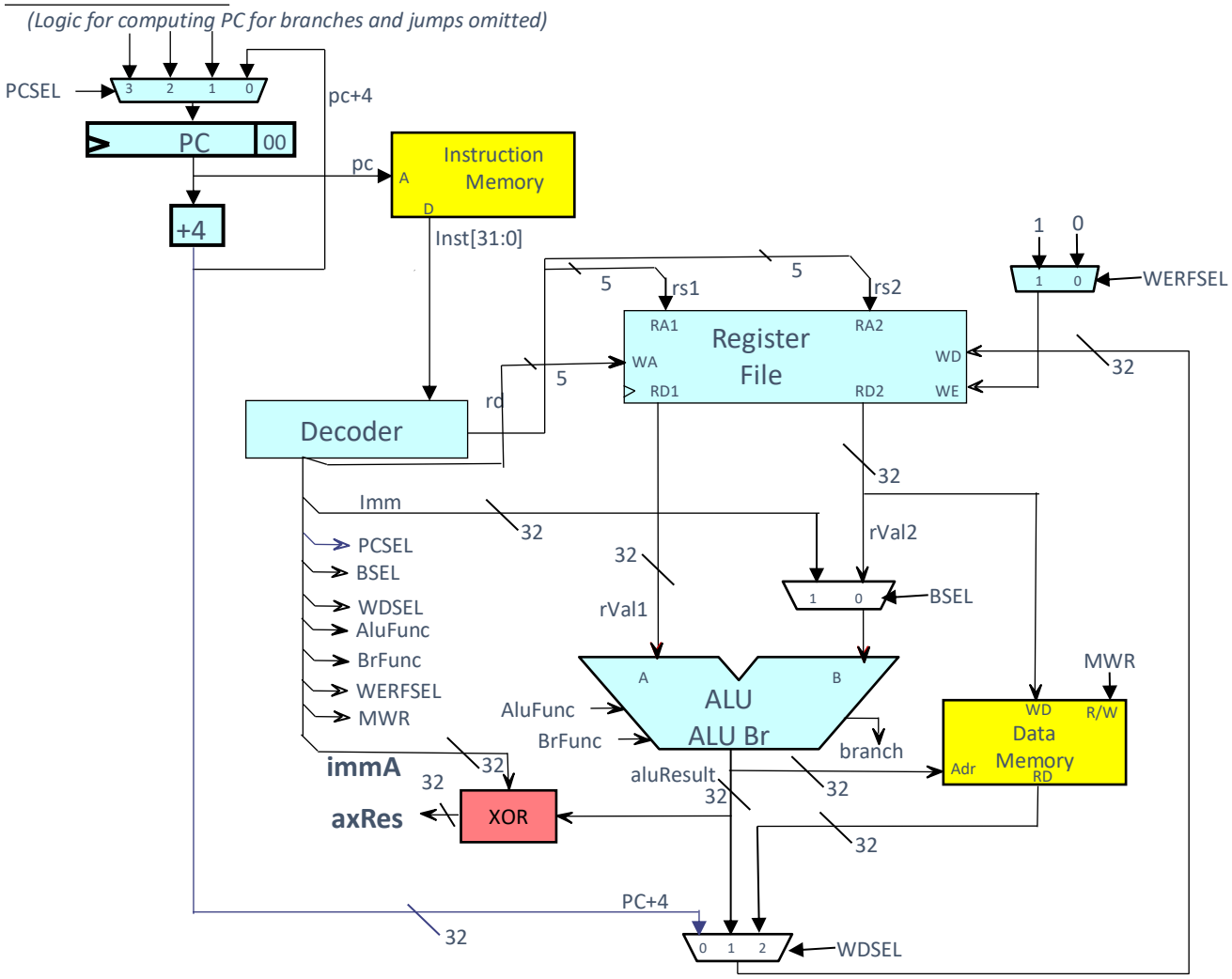
31...25	24...20	19...15	14...12	11...7	6...0
imm[9:3]	rs2	rs1	imm[2:0]	rd	0110100

(B) (2 points) Encode the following instructions as 32-bit binary words:

```
andxori x7, x1, x16, 0x1BC
```

(Label: 6B) Encoding in hexadecimal 0x: ___6F00C3B4___

In the diagram below, Giuseppe has decoded and sign extended the immediate used for the **`andxori`** instruction, labelling this signal **`immA`**. He also added an additional dedicated XOR module used in the instruction's computation. The output of this new XOR module is labelled **`axRes`**.



(C) (3 points) For each of the following signals, does the mux being controlled by that signal need an extra input to accommodate the new instruction? If so, indicate the name of **the signal that needs to be added as an input to the mux**. If not, indicate which existing value of **the mux control signal** is required to make the instruction work properly.

(Label: 6C_BSEL_1) BSEL: **Needs new input?** **YES** **(NO)**

(Label: 6C_BSEL_2) New input/Existing control signal: ____0 or rVal2____

(Label: 6C_WDSEL_1) WDSEL: **Needs new input?** **(YES)** **NO**

(Label: 6C_WDSEL_2) New input/Existing control signal: _____axRes_____

(Label: 6C_WERFSEL_1) WERFSEL: **Needs new input?** **YES** **(NO)**

(Label: 6C_WERFSEL_2) New input/Existing control signal: _____1_____

(D) (3 points) Additionally, decide for each of the following control signals what their values should be when executing the **`andxori`** instruction. If the value of the signal doesn't matter, then put **N/A**. The possible values for each signal are provided below.

AluFunc: Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra

BrFunc: Eq, Neq, Lt, Ltu, Ge, Geu

MWR: Read, Write

(Label: 6D_alufunc) AluFunc: _____And_____

(Label: 6D_brfunc) BrFunc: _____N/A_____

(Label: 6D_mwr) MWR: _____Read_____