Design of a Five Stage Pipelined RISC-V Single-Cycle Processor Supporting Integer Instructions and Hazard Handling (RV32I)

by Felix Tse

2025

BASE INTEGER SET DATAPATH AND CONTROL DESIGN

- Instruction and data memories separate to avoid structural hazards of handling simultaneous access (Harvard Architecture)
- Implement overflow for signed arithmetic operations

Load/Store Word

The microarchitecture begins with four **architectural state elements**: the Program Counter (PC), the Instruction Memory (IM), the register file (RF), and the Data Memory (DM). We have five pipeline registers to split the processor into five stages: *Fetch, Decode, Execute, Memory Access*, and *Writeback*. The *lw* and *sw* instructions are the basis for the datapath. In the *Fetch* stage, the PC sends an address to IM, which fetches the instruction to be processed. In the *Decode* stage, the instruction is decoded into its specific bit fields. The RF reads the data from the source register while simultaneously sign extending the immediate.In the *Execute* stage, the ALU performs an operation on the two operands and sends the result to DM, either an address for *lw* or a data word for *sw*. If the instruction is *lw*, the data is read from DM back to the RF in the *Writeback* stage.

R-Type Instructions

We create a four bit ALUControl signal to decode signals for all the R-type operations including arithmetic, shifts, and logical operations, shown to the right. R-type operations do not access memory nor do they use the Extend unit, so we do not need to implement any additional hardware or control signals. However, R-type (and some I-type) instructions need to account for **overflow**, which is discussed in the section directly below.

ALU Operation	ALUControl
add	0000
sub	0001
and	0010
or	0011
xor	0100
slt	0101
sltu	0110
sll	0111
srl	1000
sra	1001
32'bx	default

Later we will find it useful to combine addition and subtraction into a single signal “Sum”. We take advantage of the fact that subtraction in two’s complement involves inverting the second operand and adding 1. Therefore, we produce the following SystemVerilog snippet:

```
5   assign Sum      = SrcA + (ALUControl[0] ? ~SrcB : SrcB) + ALUControl[0];
```

Overflow

We can detect signed overflow during add/sub by checking the sign bits of the MSB of the two operands, and checking the sign bit of the result. Overflow in signed addition occurs when the two operands with the same sign produce a result with the opposite sign, while overflow in signed subtraction occurs when two operands with opposite signs produce a result with the same sign as the subtrahend. This can easily be implemented with the following SystemVerilog snippet, which first checks for the ALU operation, checks the signs of two operands, then checks the signs of result and operand:

```
1  logic isAdd = (ALUControl == 4'b0000)
2  logic isSub = (ALUControl == 4'b0001)
3
4  assign Overflow = (isAdd & ~(SrcA[31] ^ SrcB[31]) & (SrcA[31] ^ Sum[31])) |
5                    (isSub &  (SrcA[31] ^ SrcB[31]) & (SrcA[31] ^ Sum[31]));
```

By checking the ALU operation, we can disable overflow detection for ALU operations that are not addition or subtraction. The table to the right lists the overflow conditions for add and subtract on signed two's complement numbers.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Immediates and Shifts

addi adds a sign-extended 12-bit immediate to register *rs1*.
slti places 1 in register *rd* if register *rs1* is less than sign-extended immediate when both are treated as signed numbers.
sltiu also sign-extends the immediate but treats both the immediate and operand as unsigned numbers. If *rs1* equals zero, *rd* is set to 1. This is used when unsigned numbers are the correct interpretation of the data, such as in memory addresses, indexing, and flags.

Left shifts always fill the least significant bits with zeros. Right shift logical shifts zeros into most significant bits, while right shift arithmetic shifts sign bit into most significant bits. Immediate shifts fill the first 7 bits of the instruction field with 0000000 and the next 5 bits with the immediate shamt. Although the ALU only uses the lower 5 bits of the operand for shift operations, the ALU still expects 32-bit operands, and thus we sign-extend the 5-bit immediate.

Branch Control

We implement *beq, bne, blt, bge* instructions in this section.

Let us create two output flags from the ALU: ZERO and SIGN, where SIGN is ALUResult[31]. Observing funct3 bits from the instruction field (bits 14:12), we can derive the following:

$$beq = \overline{14} \cdot \overline{12} \cdot ZERO \text{ (000 funct3, rs1 - rs2 = ZERO)}$$
$$bne = \overline{14} \cdot 12 \cdot \overline{ZERO} \text{ (001 funct3, rs1 - rs2} \neq \text{ZERO)}$$
$$blt = 14 \cdot \overline{12} \cdot SIGN \text{ (100 funct3, ALUResult[31] = 1)}$$
$$bge = 14 \cdot 12 \cdot \overline{SIGN} \text{ (101 funct3, ALUResult[31] = 0)}$$

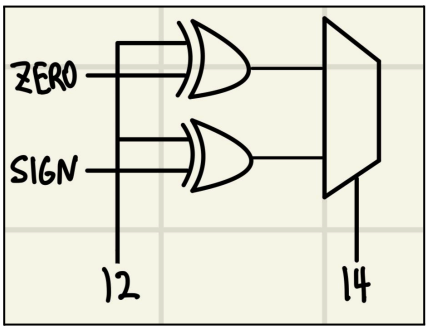
Using the above equations as product of sums, basic boolean algebra yields the following result:

$$Y = \overline{14}(12 \oplus ZERO) + 14(12 \oplus SIGN)$$

This yields three levels of gates: XOR, AND, then OR. Observe the similarities with the above result and the logic for a 2:1 multiplexer:

$$Y = (\overline{S} \cdot A) + (S \cdot B)$$

Therefore, we replace two levels of logic with a 2:1 mux, yielding the following two level gate representation:



The result from the multiplexer is then AND with Branch signal to determine whether or not to branch instruction.

Jump Control

Although *jalr* is an I-type instruction, for clarity we will treat it as a J-type instruction. The control signal for J-type (Jump) will be asserted for either *jal* or *jalr*, which is routed through an OR gate that takes the Jump and Branch control signals as inputs, and outputs a control signal PCSrc. *jalr* uses a 21-bit sign extended immediate, and therefore it becomes necessary to add an additional control signal (100) for ImmSrc that sign-extends 21-bit values.

U-Type Instructions

The *auipc* instruction adds an upper immediate (20-bit value left shifted by 12) with the current PC. The *lui* instruction stores an upper immediate (also 20-bit value left shifted by 12) in a destination register. *lui* chooses the sign extended immediate for ALUSrcB and adds it to 0, therefore it becomes necessary to create a mux for ALUSrcA, one being *rs1* and the other being *32'b0.* It also becomes necessary to add an additional control signal (101) for ImmSrc that zero-extends the LSBs of a 20-bit value. The table below summarizes the control signals for ImmSrc:

Hazards

Multiple instructions are handled concurrently in a pipelined processor, which inevitably leads to hazards. Data hazards occur when an instruction tries to read a register that has not yet been updated by a previous instruction, while control hazards occur when a fetch occurs before the decision of which instruction to fetch next has been made. We can solve these two issues with **forwarding** and **stalling**.

Forwarding involves forwarding a value from the *Memory* or *Writeback* stage to a dependent instruction in the *Execute* stage. Forwarding is used when a source register in the *Execute* stage matches the destination register currently in the *Memory* or *Writeback* stage. We add multiplexers in front of ALUSrcA and ALUSrcB to select between RF, *Memory*, or *Writeback* stage. If the current source register in the *Execute* stage matches the current destination registers in either the *Memory* or *Writeback* stages, *ForwardAE* and/or *ForwardBE* is asserted. If the destination registers match for both *Memory* and *Writeback* stages, the *Memory* stage takes priority because it is the more recently executed instruction.

Stalling involves disabling a pipeline register until its data is available. All previous stages are also stalled, and the pipeline stage directly after the stalled register must be **flushed**. Stalling is used when a *lw* is in the *Execute* stage (indicated by $ResultSrcE_0 = 1$), and the load's destination register matches the source operands of the instruction currently in the *Decode* stage. We add EN inputs to the *Fetch* and *Decode* registers, with control signals *StallF* and *StallD* are asserted to stall the *Fetch* and *Decode* pipeline registers, while *FlushE* clears the contents of the *Execute* stage pipeline register. Stalling degrades performance so should only be used when necessary.