Anqa

A RISC-V Processor

Yousef Alaa Awad

February 2025

1 Opening Words

The objective of this processor is to implement the basic 40 instruction ISA (of which 36 are explained below) of RISC-V RV32I of which is explained in my own words, as well as explain how I went about doing it myself. I plan to make the actual processor in Vivado 2024.2 using the Verilog language.

Contents

2 RV32I ISA Explained

Immediate-type Operations

Add Immediate

The 'addi' instruction is an instruction that simply adds 2 immediates (12 bits) to the register rs1, and puts it in the destination-register. The assembly is as shown below:

```
addi destination-register,rs1,immediate-value
```

Set Less Than Immediate

The 'slti' instruction is an instruction that checks if the value in register rs1 is less than the immediate-value, and if so places a value 1 in the destination-register, else it is 0. The assembly is as follows:

```
slti destination-register,rs1,immediate-value
```

Set Less Than Immediate Unsigned

The 'sltiu' instruction is an instruction that checks if the value in register rs1 is less than the immediate-value treating both values as unsigned numbers, and if so places a value 1 in the destination-register, else it is 0. The assembly is as follows:

```
sltiu destination-register,rs1,immediate-value
```

XOR Immediate

The 'xori' instruction is an instruction that performs an XOR bitwise operation on register rs1, and the signed 12-bit immediate-value, and places the result in the destination-register. The assembly is as follows:

```
xori destination-register,rs1,immediate-value
```

OR Immediate

The 'ori' instruction is an instruction that performs an OR bitwise operation on register rs1, and the signed 12-bit immediate-value, and places the result in the destination-register. The assembly is as follows:

```
ori destination-register,rs1,immediate-value
```

AND Immediate

The 'andi' instruction is an instruction that performs an AND bitwise operation on register rs1, and the signed 12-bit immediate-value, and places the result in the destination-register. This assembly is as follows:

```
andi destination-register,rs1,immediate-value
```

Shift Left Logical Immediate

The 'slli' instruction is an instruction that performs a logical left shift on the value given in register rs1 by the amount held in the last 5 bits of the immediate-value. The result is placed in the destination-register. The assembly is as follows:

```
slli destination-register,rs1,immediate-value
```

Shift Right Logical Immediate

The 'srli' instruction is an instruction that performs a logical right shift on the value given in register rs1 by the amount held in the last 5 bits of the immediate-value. The result is placed in the destination-register. The assembly is as follows:

```
srli destination-register,rs1,immediate-value
```

Shift Right Arithmetic Immediate

The 'srai' instruction is an instruction that performs an arithmetic left shift on the value given in register rs1 by the amount held in the last 5 bits of the immediate-value. The result is placed in the destination-register. The assembly is as follows:

```
srai destination-register,rs1,immediate-value
```

Load Byte

The 'lb' instruction is an instruction that loads an 8-bit value from memory (from the address taken from register rs1, with an offset of offset bytes) and sign-extends it to XLEN (32 bits in Anqa's case) and then stores it into the destination-register. The assembly is as follows:

```
lb destination-register,offset(rs1)
```

Load Halfword

The 'lh' instruction is an instruction that loads a 16-bit value from memory (from the address taken from register rs1, with an offset of offset bytes). It then sign-extends it to 32-bits and then stores it into the destination-register. The assembly is as follows:

```
lh destination-register,offset(rs1)
```

Load Word

The 'lw' instruction is an instruction that loads a 32-bit value from memory (from the address taken from register rs1, with an offset of offset bytes). It then sign-extends it to 32-bits and then stores it into the destination-register. The assembly is as follows:

```
lw destination-register,offset(rs1)
```

Load Byte Unsigned

The 'lbu' instruction is an instruction that loads an 8-bit value from memory (from the address taken from register rs1, with an offset of offset bytes). It then zero-extends it to 32-bits and then stores it into the destination-register. The assembly is as follows:

```
lbu destination-register,offset(rs1)
```

Register-Register Operations

ADDition

The 'add' instruction is an instruction that simply adds the registers, rs1 and rs2, and puts the result in the destination-register. All arithmetic overflows are ignored. The assembly is as follows:

```
add destination-register,rs1,rs2
```

SUBtraction

The 'sub' instruction is an instruction that simply subtracts register rs2 from register rs1, and puts the result in the destination-register. All arithmetic overflows are ignored. The assembly is as follows:

```
sub destination-register,rs1,rs2
```

Shift Left Logical

The 'sll' instruction is an instruction that logical left shifts the data in register rs1 by the shift amount held in the last 5 bits of register rs2, and puts the result in the destination-register. The assembly is as follows:

```
sll destination-register,rs1,rs2
```

Shift Right Logical

The 'srl' instruction is an instruction that logical right shifts the data in register rs1 by the shift amount held in the last 5 bits of register rs2, and puts the result in the destination-register. The assembly is as follows:

```
srl destination-register,rs1,rs2
```

Shift Right Arithmetic

The 'sra' instruction is an instruction that arithmetic right shifts the data in register rs1 by the shift amount held in the last 5 bits of register rs2, and puts the result in the destination-register. The assembly is as follows:

```
sra destination-register,rs1,rs2
```

Set Less Than

The 'slt' instruction is an instruction that checks if register rs1 is less than register rs2, and if so places the value 1 in the destination-register, else it places the value 0. The assembly is as follows:

```
slt destination-register,rs1,rs2
```

Set Less Than Unsigned

The 'sltu' instruction is an instruction that checks if register rs1 is less than register rs2 treating both values in rs1 and rs2 as if they were unsigned numbers, and if so places the value 1 in the destination-register, else it places the value 0. The assembly is as follows:

```
sltu destination-register,rs1,rs2
```

bitwise XOR

The 'xor' instruction is an instruction that performs the XOR bitwise operation on registers rs1 and rs2 and places the result in the destination-register. The assembly is as follows:

```
xor destination-register,rs1,rs2
```

bitwise OR

The 'or' instruction is an instruction that performs the OR bitwise operation on registers rs1 and rs2 and places the result in the destination-register. The assembly is as follows:

```
or destination-register,rs1,rs2
```

bitwise AND

The 'and' instruction is an instruction that performs the AND bitwise operation on registers rs1 and rs2 and places the result in the destination-register. The assembly is as follows:

```
and destination-register,rs1,rs2
```

Store-type Instructions

Store Byte

The 'sb' instruction is an instruction that stores the least significant byte (8 bits) from the register rs2 to the memory address calculated by adding the sign-extended immediate known as offset below to the value of the register rs1. The assembly is as follows:

```
sb rs2,offset(rs1)
```

Store Halfword

The 'sh' instruction is an instruction that stores the least significant halfword (2 bytes or 16 bits) from the register rs2 to the memory address calculated by adding the sign-extended immediate known as offset below to the value of the register rs1. The assembly is as follows:

```
sh rs2,offset(rs1)
```

Store Word

The 'sw' instruction is an instruction that stores the word (4 bytes or 32 bits) from the register rs2 to the memory address calculated by adding the sign-extended immediate known as offset below to the value of the register rs1. The assembly is as follows:

```
sw rs2,offset(rs1)
```

Branch-type Instructions

Branch if EQual

The 'beq' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 and rs2 are equal. The assembly is as follows:

```
beq rs1,rs2,offset
```

Branch if Not Equal

The 'bne' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 and rs2 are not equal. The assembly is as follows:

```
bne rs1,rs2,offset
```

Branch if Less Than

The 'blt' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 is less than rs2. The assembly is as follows:

```
blt rs1,rs2,offset
```

Branch if Greater than or Equal

The 'bge' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 is greater than or equal to rs2. The assembly is as follows:

```
bge rs1,rs2,offset
```

Branch if Less Than Unsigned

The 'bltu' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 is less than rs2, with both values being treated as if they were unsigned integers. The assembly is as follows:

```
bltu rs1,rs2,offset
```

Branch if Greater than or Equal Unsigned

The 'bgeu' instruction is an instruction that branches to the target address (that being offset), if and only if the values in registers rs1 is greater than or equal to rs2, with both values being treated as if they were unsigned integers. The assembly is as follows:

```
bgeu rs1,rs2,offset
```

Upper Immediate-type Instructions

Load Upper Immediate

The 'lui' instruction is an instruction that loads the 20-bit immediate value into the upper 20 bits of the destination-register, and sets the lower 12 bits to 0. The assembly is as follows:

```
lui destination-register,immediate-value
```

Add Upper Immediate to Program Counter

The 'auipc' instruction is an instruction that adds the 20-bit immediate value, shifted to the left by 12 bits, to the program counter. The assembly is as follows:

```
auipc destination-register,immediate-value
```

Jump-type Instructions

Jump And Link

The 'jal' instruction is an instruction that performs an unconditional (also known as immediate) jump to a target address that is calculated by adding the sign-extended immediate, shifted left by 1 bit, to the current program counter. The address of the next instruction is then saved to the destination-register. The assembly is as follows:

```
jal destination-register,immediate-value
```

Jump And Link Register

The 'jalr' instruction is an instruction that performs an unconditional (also known as immediate) jump to a target address that is calculated by adding a 12-bit sign-extended immediate, to the value in registter rs1. The least significant bit of the computed address is also cleared to ensure that it is aligned in the process. The address of the instruction following the jump (program counter + 4 bits) is then saved to the destination-register. The assembly is as follows:

```
jalr destination-register,rs1,immediate-value
```

3 Part 1: The 32-bit Adder

First, to even design a CPU, alongside any real digital circuit, I first had to understand a basic adder, with our end goal being the 32-bit adder. However, to even understand a 32-bit adder, I first needed to understand the even simpler half adder.

Part 1A: The Half Adder

A half adder is simply just, well, a device that adds 2 bits and outputs its simple addition along with a carry bit incase of an overflow, that has the following properties of which are taken from the Discrete World:

2-Bit Adder Properties

$Bit1$	$Bit2$	Sum
0	0	0
0	1	1
1	0	1
1	1	10

As you can see, when you add 2 bits, there are 3 unique cases that can happen:

1. You get an output of 0.
2. You get an output of 1.
3. You get an output of 0 but an overflow of 1, as the output can only be 1 bit in length.

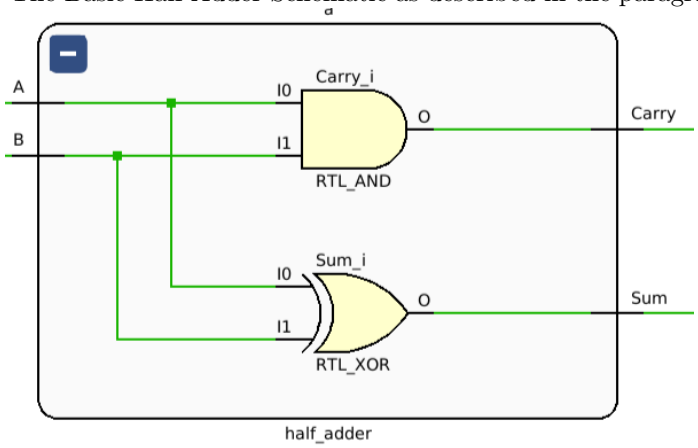
And now you may be wondering, how in the world do you convert that to simple circuits using something like ANDs and ORs and XORs and NOTs??? Well, let's think, what does the table itself look like?

According to the table, the output bit is only true, if and only if, only one of the inputs is on/true, else it is 0. Now, when converted to discrete terms, we know that we have the following basic blocks of relations:

1. The AND gate, of which only outputs a 1 if and only if all inputs are 1.
2. The OR gate, of which only outputs a 1 if at least one of the inputs are 1.
3. The XOR gate, of which only outputs a 1 if and only if only one input is 1, and not both.
4. The NOT gate, of which only outputs 1 if the input is 0, and a 0 if the input is 1.

Now, the only option that aligns with the statement above on the half adder's truth table for its output, would have to be the XOR (The pointy one with a line out of its inputs) gate. But what of the carry? Well, when is that a 1? It's only 1 when BOTH Bit1 and Bit2 are on, therefore isn't it just the definition of an AND gate (The one that looks like an U shaded in)? And therefore, wouldn't the circuit look something like this?

Figure 1: The Basic Half Adder Schematic as described in the paragraph above



And, since this documentation isn't explaining Verilog, and its syntax, the above schematic would convert to look like the following verilog code:

```
module half_adder(
        input Bit1, Bit2,
        output Sum, Carry
    );

    assign Sum = Bit1 ^ Bit2; // ^ is XOR
    assign Carry = Bit1 & Bit2; // & is AND

endmodule
```

Listing 1: Half Adder with 2 input bits, and outputs, sum and carry, as explained

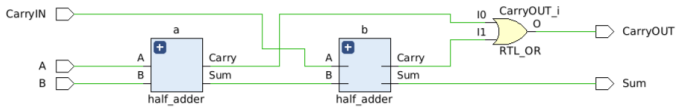
Part 1B: The Full Adder

Now, currently we simply have a half adder. But, what if we want to take into account the carry from another addition before it, for example if we are adding the numbers 0 and 1 with an input carry of 1. Well, with our half adder that we made above, we only get a carry out, and can't exactly input a carry in. Therefore, wouldn't we need to take into account 3 inputs, and not just 2?

Now, how would we actually construct this mystical "ultimate adder" (or more commonly known as the full adder)? Well, again, let's think on it. What exactly happens when given an input carry into the addition above? Well, it'd be treated simply as a three way addition of $1 + 1 + 0$. And, when broken down even further, wouldn't that just simply be 2 addition operations, of which don't necessarily need anything but the half adder we designed above? Which, when chained together, is the addition of the carry input and the first bit, then of that result to the second bit.

But what of the carry out signal? Well, that would just simply only be true if and only if at least one of those additions produced a carry in the first place. Therefore, we can derive the schematic below:

Figure 2: A Full Adder with 2 half adders chained together, alongside an or gate to determine if a carry is to outputted




```
module full_adder(
        input A, B, CarryIN,
        output Sum, CarryOUT
    );

    wire carry1, carry2, sum1;

    half_adder a(.Bit1(A), .Bit2(B), .Carry(carry1), .Sum(
        sum1));
    half_adder b(.Bit1(CarryIN), .Bit2(sum1), .Carry(carry2)
        , .Sum(Sum));

    assign CarryOUT = carry1 | carry2;

endmodule
```

Listing 2: Verilog code of the Full Adder schematic above

Part 1C: Chaining Adders via Ripple Carry Adders

And now we get to a fun part, a part where we get to chain the adders together into 2 bit adders, and 4 bit adders, etc etc until we get to a 32 bit adder. To do this, though, I shall be taking it in the approach that, for now, it will be made up of two 16 bit adders, of which is made up of two 8 bit adders, and so on. I'll only be showing the code for the 2 bit adder though, as after

that it is the same structure with different names.

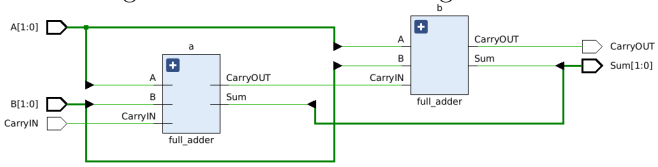
Therefore, how in the world do we chain the two bits together into a result that lets you add two 2 bit numbers together? Well, let's think about it, we have 2 bits in an array, number 1, of which is two bits long, and number 2, of which is also two bits long. Wouldn't we have to add the least significant bit first (of which is, if you didn't know, the right-most bit), and then output the result to the output bit, as well as the carry out to the carry in of the bit to the left of it? So, let's try that out in with the following verilog code, and guess what it's correct:

```
1  module two_bit_adder(
2          input[1:0] A, B,
3          input CarryIN,
4          output[1:0] Sum,
5          output CarryOUT
6      );
7
8      wire carry;
9
10     full_adder a(.A(A[0]), .B(B[0]), .CarryIN(CarryIN), .
           CarryOUT(carry), .Sum(Sum[0]));
11     full_adder b(.A(A[1]), .B(B[1]), .CarryIN(carry), .
           CarryOUT(CarryOUT), .Sum(Sum[1]));
12
13 endmodule
```

Listing 3: 2 Bit Adder Verilog Code

Which generates the following schematic:

Figure 3: 2 Bit Adder as designed above



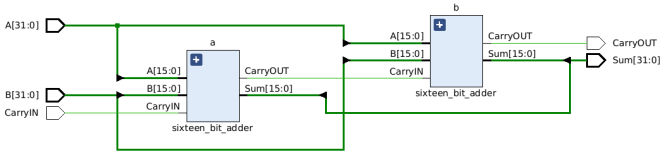
And now, to just spoil the fun for you, I'll just tell you that, guess what, it's the exact same thing for every addition to it, from 4 bits to 32 bits to even 128 bits if you wanted. And so, I'll just show you the 32 bit encapsulation, and to find the submodules in it head to the code proper in sources...

```
module thirtytwo_bit_adder(
        input [31:0] A, B,
        input CarryIN,
        output [31:0] Sum,
        output CarryOUT
    );

    wire carry;

    sixteen_bit_adder a(.A(A[15:0]), .B(B[15:0]), .CarryIN(
        CarryIN), .CarryOUT(carry), .Sum(Sum[15:0]));
    sixteen_bit_adder b(.A(A[31:16]), .B(B[31:16]), .CarryIN
        (carry), .CarryOUT(CarryOUT), .Sum(Sum[31:16]));

endmodule
```

Listing 4: 32 Bit Adder Verilog Code

And generates the following schematic:

Figure 4: 32 Bit Adder as designed above



Now, you may be wondering, why in the world do we even need a 32-bit adder in the firstplace? And, that's quite simple. We need it due to the fact that the entire system is meant to be a 32-bit processor, of which takes in 32 bits of information, then modulate it with shifting and additions and subtractions, and then outputs a 32 bit result, either to be stored or to be placed onto the next instruction that it is told to do.

4 The Arithmetic Logic Unit

Now that we have a basic 32 bit adder designed,

5 Important Definitions

Arithmetic Shifting

This is a bitwise operation that, like logical shifting, moves all of the bits of a binary of bit-length X, left or right by a specificied number of positions/bits. Unlike logical shifting, however, you would preserve the sign of the number and therefore when shifting a negative number would fill with 1s or if positive would be 0s.

Immediate Values

An immediate value is a constant that is embeded directly into an instruction. This means that it is not stored in a register or memory location, and therefore does not need to access anything before using the value, making the value usable immediately after its declaration.

Logical Shifting

This is a bitwise operation that moves all of the bits of the binary number of bit-length X, left or right by a specificied number of positions/bits. The vacant bits, if any are created, are subsequently filled with 0s.

Program Counter (PC)

The program counter is a special register that holds the memory address of the next instruction that is going to be executed.

Sign Extension

This is the process of increasing the bit-width of a binary number while preserving its value and sign from the original. To do this, you duplicate the most significant bit into the new empty bits that are higher order than the original bit-width.

Zero Extension

This is the process of increasing the bit-width of a binary number but, unlike sign extensions, is only used for unsigned binary numbers and only simply adds 0s to the higher order bits, therefore only preserving the original value.