SE201 Project 1

Drobotun Valerii, Daniel Vahos Mendoza, Luis Peña

December 2021 – January 2022

1 RISC-V Instruction Set

1.1 A reverse engineering task

We are working in a company that have a specialisation in the reverse engineering. This winter we received a new device for re-engineering, and our team have to understand what is happening in the firmware of that device.

We got a dump from device's ROM, and now we have to understand what this code do. Everything that we know about it is that device uses RISC-V architecture, so the first step to understand the code is to disassemble it.

```
0:  00050893
4:  00068513
8:  04088063
c:  04058263
10: 04060063
14: 04d05063
18: 00088793
1c: 00269713
20: 00e888b3
24: 0007a703
28: 0005a803
2c: 01070733
30: 00e62023
34: 00478793
38: 00458593
3c: 00460613
40: ff1792e3
44: 00008067
48: fff00513
4c: 00008067
50: fff00513
54: 00008067
```

Listing 1: ROM dump

1.2 Binary code analysis

According to 32-bit instruction encoding for RISC-V processors, each command can be divided into its counterparts, as `opcode` (OPeration CODE), `fun3` (additional definition of command), `fun7` (Additional definition of command/immediate value), `rd` (Register for Data), `rs1` (Register Source 1), `rs2` (Register Source 2).

```
    fun7    rd    rs1   fun3 rs2  opcode
0:  0000000 00000 01010 000 10001 0010011
4:  0000000 00000 01101 000 01010 0010011
8:  0000010 00000 10001 000 00000 1100011
c:  0000010 00000 01011 000 00100 1100011
10: 0000010 00000 01100 000 00000 1100011
14: 0000010 01101 00000 101 00000 1100011
18: 0000000 00000 10001 000 01111 0010011
1c: 0000000 00010 01101 001 01110 0010011
20: 0000000 01110 10001 000 10001 0110011
24: 0000000 00000 01111 010 01110 0000011
28: 0000000 00000 01011 010 10000 0000011
2c: 0000000 10000 01110 000 01110 0110011
30: 0000000 01110 01100 010 00000 0100011
34: 0000000 00100 01111 000 01111 0010011
38: 0000000 00100 01011 000 01011 0010011
3c: 0000000 00100 01100 000 01100 0010011
```

```
40: 1111111 10001 01111 001 00101 1100011
44: 0000000 00000 00001 000 00000 1100111
48: 1111111 11111 00000 000 01010 0010011
4c: 0000000 00000 00001 000 00000 1100111
50: 1111111 11111 00000 000 01010 0010011
54: 0000000 00000 00001 000 00000 1100111
```

Listing 2: Binary command representation

This work was performed by a simple python script, that just divides 32-bit codewords into according parts of a command.

The next task was to decode this binary representation into the commands. We used yet another python script to collect the statistics of the opcodes, which helped us not to re-create full decoding tree, but only necessary commands. The resulting statistics is presented below.

```
opcode  : [fun7, fun3]
0010011 : [0000000 000] [0000000 001] [1111111 000]
1100011 : [0000010 000] [0000010 101] [1111111 001]
0110011 : [0000000 000]
0000011 : [0000000 010]
0100011 : [0000000 010]
1100111 : [0000000 000]
```

Listing 3: Opcode statistics

Thus, we have to support only 6 opcodes and some variations of `fun3` and `fun7`. Decoding can be performed via following decoding dictionaries:

```
opcode    : {fun3  : 'asm'}
0b0010011 : {0b000 : 'addi', 0b001 : 'slli'}
0b1100011 : {0b000 : 'beq',  0b001 : 'bne', 0b101 : 'bge'}
0b0110011 : {0b000 : 'add/sub'}
0b0100011 : {0b010 : 'sw'}
0b1100111 : {0b000 : 'jalr'}
0b0000011 : {0b010 : 'lw'}
```

Listing 4: Command decoding

It should be noted that decoding of `add` and `sub` relies also on `fun7`, which raises complexity of a code. Also we can see here that only `I`, `R`, `S`, `SB` instructions are used.

We omit the calculations of the immediate values for the `I` and `SB`, it is rather simple, if all binary arithmetic is calculated correctly. It should be also noted that immediate values are signed, and has different bit order for `I` and `SB`. Intermediate result is presented below, at the right in brackets we added instruction type.

```
addi a7, a0, 0 [I]
addi a0, a3, 0 [I]
beq a7, zero, _label72 [SB]
beq a1, zero, _label80 [SB]
beq a2, zero, _label80 [SB]
bge zero, a3, _label84 [SB]
addi a5, a7, 0 [I]
slli a4, a3, 2 [I]
add a7, a7, a4 [R]
_label36:
lw a4, a5, 0 [I]
lw a6, a1, 0 [I]
add a4, a4, a6 [R]
sw a4, 0(a2) [S]
addi a5, a5, 4 [I]
addi a1, a1, 4 [I]
addi a2, a2, 4 [I]
bne a5, a7, _label36 [SB]
jalr zero, ra, 0 [I]
_label72:
addi a0, zero, -1 [I]
jalr zero, ra, 0 [I]
_label80:
addi a0, zero, -1 [I]
_label84:
jalr zero, ra, 0 [I]
```

Listing 5: Assembler instructions

But in the attempt of disassembling it is quite hard to stop yourself at the intermediate results. We added some assembler "sugar" to the commands, and finally got this refined listing:

```
1  mv a7, a0
2  mv a0, a3
3  beqz a7, _label72
4  beqz a1, _label80
5  beqz a2, _label80
6  blez a3, _label84
7  mv a5, a7
8  slli a4, a3, 2
9  add a7, a7, a4
10 _label36:
11 lw a4, 0(a5)
12 lw a6, 0(a1)
13 add a4, a4, a6
14 sw a4, 0(a2)
15 addi a5, a5, 4
16 addi a1, a1, 4
17 addi a2, a2, 4
18 bne a5, a7, _label36
19 ret
20 _label72:
21 li a0, -1
22 ret
23 _label80:
24 li a0, -1
25 _label84:
26 ret
```

Listing 6: Refined assembler instructions

1.3 Assembler analysis

As we can see from the first line, register `a0` is stored into the `a7`, and it can be a sign that `a0` can be used to return the value from the function. Then we see copying of `a3` to `a0`, so *maybe* now `a0` is storing thee return value.

After the move, there are a bunch of branch instructions, which lead to some addresses next to the end of the function. Register `a7` (Previously `a0`, or the first operand) is checked to be zero. If check is failed, then `a0` is loaded with -1, and function returns. Certainly, `a0` holds an error code now.

Same situation for `a1` and `a2`, if they are zero, function returns -1. However, this is not the case for the `a3` register, if it is below zero, function returns just the value of `a0`, well, with previously stored value of `a3`, or the fourth operand of the function.

Then we see yet another move from `a7` to `a7`, so now we can find here the value of the first operand of the function. Further, in `a4` stored the value of `a3`, multiplied by 4, and later in the program it can be seen that `a4` is used as a temporary register for load from `a5`, and its initial value is not preserved, so that means that the function use only 4 operands (`a0` – `a3`).

Now let's move to the `_label36` loop: as an indexer **`a5`** is used, and every cycle `a5`, `a1`, `a2` are incremented by 4, and also `lw`/`sw` operations are used, so it's definitely reading of words and storing of words without interleaving.

In general, it seems like 2 words are loaded from addresses, stored in `a5` and `a1`, then added, and the result is stored at the address, defined by `a2`.

Now it's clear why at the line 8 register `a4` is a `a3`, multiplied by 4: `a3` defines the number of words to process, and then `a7` stores the final address.

So if the loop ends, function returns only the value of `a3`, or, total number of processed words. In case of error, such as 0 addresses it returns -1, and in case of length below 0 it returns just that length.

What is the function actually doing? It sums up two arrays into the third one, with given length. E.g. in C the prototype is `int32_t sum(int32_t* a, int32_t* b, int32_t* res, int32_t length)`. Return value is -1, if any of pointers (addresses) is equal to 0, and `length` in any other case.

Also it can be noted that `_label72` and `_label80` are pointing to different instructions, which, however, do the same thing: loading -1 to `a0` and returning. That could mean that this function is a compiled one (probably from C), not written by hand. If it was written by hand, programmer could economize 2 instructions, or 8 bytes!

1.4 Branch delay slot

This technique was widely used on a MIPS machines, which allowed to execute the next instruction after the branching instruction *no matter if the branch is taken or not*. However, not all instructions can be executed in a delay slot, e. g. branching cannot.

The disadvantages are obvious, the machine code is hard to read and understand, since the programmer should always take in mind that command(s) after the branch is(are) executed. Moreover, some versions of processors can have 1, or 2 branch delay slots, so that means that those processors are incompatible, even if registers and assembler commands are the same.

Also this is a headache for the compiler developers, they always should take in mind this 'unique' feature while compiling a program, which turns into instruction reordering and `nop` inserting.

However, some advantages can be found. In some cases, code can be a bit more compact. Also this feature can be used for code obfuscating, if we are writing specific cryptography soft.

Main advantage is – less pipeline stalling. E.g. for a 1 delay slot processor, we save 1 clock period of it, performing a command in that delay slot.

In fact, modern ARM and other RISC systems (RISC-V also) tend not to use branch delay slots because of described earlier side effects.

2 RISC-V Tool Chain

2.1 Matching it to a C program

To transform this assembler instructions into a higher programming language like C we have to first understand what the assemble code it's doing.

We got the variables `a` and `b` into the function as pointers to an address of memory. We have the requirement that if either `a7`, `a1` are equal to 0, we return the -1 error. However, if we are fine with pointers, we check if length is positive, and if not, we stop the function execution, and returning this length.

Now, we have to compute the final address which is equal to the sum of `a7` + `a4` or in our C code to `a` + `length`. Note that there's no multiplication by 4 since it's done automatically while compiling, regarding to pointed data type, which is 4 bytes wide.

Then we have a loop where values are consecutively loaded from addresses `a` and `b`, added together and stored to memory address, defined by `res`. This is followed by the pointer incrementation, however, C programmers would write it in one line, like `*res++ = *a++ + *b++`.

This code is actually quite strange, because we are checking if length is lower that 0. That means, in a 32-bit addressing, this function won't work for the upper 2GB of memory, at addresses from 0x10000000 to 0xFFFFFFFF, which looks like more not checking if length is negative, but a sort of memory protection.

```c
#include<stdint.h>
#include<memory.h>

int32_t array_sum(int32_t* a, int32_t* b, int32_t* res, int32_t length)
{
    if (a == NULL)
        return -1;
    if (b == NULL)
        return -1;
    if (res == NULL)
        return -1;
    if (length <= 0)
        return length;

    int32_t* final_addr = a + length;

    do {
        *res = *a + *b;

        a++;
        b++;
        res++;
    } while (a != final_addr);

    return length;
}
```

Listing 7: C code

2.2 Comparing objdump output to the original code

We compiled our program via

```
riscv64-linux-gnu-gcc -g -Ox -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -c
-o se201-prog.o func.c
```

where **x** was 0,1 and 3. Then performed disassembly, which was done by the `objdump` tool:

```
riscv64-linux-gnu-objdump -d se201-prog.o
```

This is the output of the objdump output:

```
se201-prog.o:     format de fichier elf32-littleriscv
Deassemblage de la section .text :
00000000 <array_sum>:
   0: 04050063            beqz  a0,40 <.L0 >
00000004 <.L0 >:
   4: 04058263            beqz  a1,48 <.L0 >
00000008 <.L0 >:
   8: 04060463            beqz  a2,50 <.L0 >
0000000c <.L0 >:
   c: 02d05663            blez  a3,38 <.L0 >
00000010 <.L0 >:
  10: 00269893            slli  a7,a3,0x2
00000014 <.L0 >:
  14: 011508b3            add a7,a0,a7
00000018 <.L0 >:
  18: 00052703            lw  a4,0(a0)
  1c: 0005a803            lw  a6,0(a1)
  20: 01070733            add a4,a4,a6
00000024 <.L0 >:
  24: 00e62023            sw  a4,0(a2)
00000028 <.L0 >:
  28: 00450513            addi  a0,a0,4
0000002c <.L0 >:
  2c: 00458593            addi  a1,a1,4
00000030 <.L0 >:
  30: 00460613            addi  a2,a2,4
00000034 <.L0 >:
  34: fea892e3            bne a7,a0,18 <.L0 >
00000038 <.L0 >:
  38: 00068513            mv  a0,a3
0000003c <.LVL6>:
  3c: 00008067            ret
00000040 <.L0 >:
  40: fff00693            li  a3,-1
00000044 <.LVL8>:
  44: ff5ff06f            j 38 <.L0 >
00000048 <.L0 >:
  48: fff00693            li  a3,-1
0000004c <.LVL10>:
  4c: fedff06f            j 38 <.L0 >
00000050 <.L0 >:
  50: fff00693            li  a3,-1
00000054 <.LVL12>:
  54: fe5ff06f            j 38 <.L0 >
```

Listing 8: Objdump output

If we clean it a bit to match the assembly code we obtained in the first stage of our investigation, it could be compared more freely:

```

   0: 04050063            beqz  a0,40 <.L0 >
   4: 04058263            beqz  a1,48 <.L0 >
   8: 04060463            beqz  a2,50 <.L0 >
   c: 02d05663            blez  a3,38 <.L0 >
  10: 00269893            slli  a7,a3,0x2
  14: 011508b3            add a7,a0,a7
  18: 00052703            lw  a4,0(a0)
  1c: 0005a803            lw  a6,0(a1)
  20: 01070733            add a4,a4,a6
  24: 00e62023            sw  a4,0(a2)
  28: 00450513            addi  a0,a0,4
  2c: 00458593            addi  a1,a1,4
  30: 00460613            addi  a2,a2,4
  34: fea892e3            bne a7,a0,18 <.L0 >
  38: 00068513            mv  a0,a3
  3c: 00008067            ret
  40: fff00693            li  a3,-1
  44: ff5ff06f            j 38 <.L0 >
  48: fff00693            li  a3,-1
  4c: fedff06f            j 38 <.L0 >
```

```
  50: fff00693            li  a3,-1
  54: fe5ff06f            j 38 <.L0 >
```

Listing 9: Refined Objdump output without -O option

While not using the optimization option, compiler generates a code that's somewhat similar to the original, taking into account the flow of the code and how it implements the same structure. Some of the changes are seen in the lack of additional `mv` commands and how it returns to branches; in the original we use branches to jump to the return commands and here plenty of `j` commands are used and only one `ret`.

However, we also can see that the lines from 0x40 to 0x54 are just copies, which can be optimized.

Now, lets have a look at O0 optimisation:

```
   0: fd010113            addi  sp,sp,-48
   4: 02812623            sw  s0,44(sp)
   8: 03010413            addi  s0,sp,48
   c: fca42e23            sw  a0,-36(s0)
  10: fcb42c23            sw  a1,-40(s0)
  14: fcc42a23            sw  a2,-44(s0)
  18: fcd42823            sw  a3,-48(s0)
  1c: fdc42783            lw  a5,-36(s0)
  20: 00079663            bnez  a5,2c <.L0 >
  24: fff00793            li  a5,-1
  28: 0980006f            j c0 <.L0 >
  2c: fd842783            lw  a5,-40(s0)
  30: 00079663            bnez  a5,3c <.L0 >
  34: fff00793            li  a5,-1
  38: 0880006f            j c0 <.L0 >
  3c: fd442783            lw  a5,-44(s0)
  40: 00079663            bnez  a5,4c <.L0 >
  44: fff00793            li  a5,-1
  48: 0780006f            j c0 <.L0 >
  4c: fd042783            lw  a5,-48(s0)
  50: 00f04663            bgtz  a5,5c <.L0 >
  54: fd042783            lw  a5,-48(s0)
  58: 0680006f            j c0 <.L0 >
  5c: fd042783            lw  a5,-48(s0)
  60: 00279793            slli  a5,a5,0x2
  64: fdc42703            lw  a4,-36(s0)
  68: 00f707b3            add a5,a4,a5
  6c: fef42623            sw  a5,-20(s0)
  70: fdc42783            lw  a5,-36(s0)
  74: 0007a703            lw  a4,0(a5)
  78: fd842783            lw  a5,-40(s0)
  7c: 0007a783            lw  a5,0(a5)
  80: 00f70733            add a4,a4,a5
  84: fd442783            lw  a5,-44(s0)
  88: 00e7a023            sw  a4,0(a5)
  8c: fdc42783            lw  a5,-36(s0)
  90: 00478793            addi  a5,a5,4
  94: fcf42e23            sw  a5,-36(s0)
  98: fd842783            lw  a5,-40(s0)
  9c: 00478793            addi  a5,a5,4
  a0: fcf42c23            sw  a5,-40(s0)
  a4: fd442783            lw  a5,-44(s0)
  a8: 00478793            addi  a5,a5,4
  ac: fcf42a23            sw  a5,-44(s0)
  b0: fdc42703            lw  a4,-36(s0)
  b4: fec42783            lw  a5,-20(s0)
  b8: faf71ce3            bne a4,a5,70 <.L0 >
  bc: fd042783            lw  a5,-48(s0)
  c0: 00078513            mv  a0,a5
  c4: 02c12403            lw  s0,44(sp)
  c8: 03010113            addi  sp,sp,48
  cc: 00008067            ret
```

Listing 10: Refined Objdump output with -O0 option

Here we can observe how disabling the optimization in the options affects greatly the structure of the assembly code, adding many instructions that are not required and a bizarre structure, this is due to the compiler doesn't optimize the structure of the program and it's "translating" the lines from the C code we wrote.

More interesting thing, is that this program uses stack for the function guard.

And finally compiled code with O3 optimisation:

```
   0: 04050063            beqz  a0,40 <.L0 >
   4: 02058e63            beqz  a1,40 <.L0 >
   8: 02060c63            beqz  a2,40 <.L0 >
```

```
   c: 00269813            slli  a6,a3,0x2
  10: 01050833            add a6,a0,a6
  14: 02d05263            blez  a3,38 <.L0 >
  18: 00052783            lw  a5,0(a0)
  1c: 0005a703            lw  a4,0(a1)
  20: 00450513            addi  a0,a0,4
  24: 00458593            addi  a1,a1,4
  28: 00e787b3            add a5,a5,a4
  2c: 00f62023            sw  a5,0(a2)
  30: 00460613            addi  a2,a2,4
  34: fea812e3            bne a6,a0,18 <.L0 >
  38: 00068513            mv  a0,a3
  3c: 00008067            ret
  40: fff00513            li  a0,-1
  44: 00008067            ret
```

Listing 11: Refined Objdump output with -O3 option

In the case of using option `-O3` meaning optimization level 3 we can observe a much better optimized code that actually looks similar to the original code we saw in the previous chapter. Actually compiler optimizes our code even more using less lines than the original thanks to the removing repetitive code at the end of the function, as we noted before.

3 RISC-V Architecture

3.1 Program Flow

Program flow can be seen in the attached files (`pipeline-ex.pdf` and `pipeline.pdf`). Initial and final state of the affected memory as well as register values are presented below.

Memory initial state

Address	Value
0x200	0x61
0x204	0x20
0x208	0x62
0x20C	0x00

Register initial state

Register	Value
a0	0x200
a1	0x200
a2	0x200
a3	0x2

Memory final state

Address	Value
0x200	0xC2
0x204	0x40
0x208	0x62
0x20C	0x00

Briefly, for the given parameters, the function does not take conditional branches at the beginning, as soon as parameters are correct. Then, it performs only 2 loops, and return 2 at the `a0` register as only one return value.

4 Processor Design

4.1 Instruction Set Architecture

Now we have to describe the list of commands of our own processor that can run that *useful* code.

It should have 16-bit wide instruction set with 16 registers, so for the three-registers command we rest only 4 bits for the command definition, if we want to make the registers used fully.

Registers are 32-bit wide, as well as `PC` and `SP`. We also need load/store commands.

> *I would like to use an instruction set from the PDP-11, it is almost "full orthogonal", 16-bit wide, with easy encoding and can execute whatever you want. But, it uses only 8 registers...*

4.1.1 ISA Realisation

First things first, we have to think about our registers, and what do we need from our processor. It is useful to have a zero register as on RISC-V, let it be `r0` too. Then, link register is useful for storing return address when calling a function, let it be `r1`. Let the third (`r2`) register be a stack pointer, which is a useful conception too.

Now we state that our processor is kind of stupid, so that's all for the list of non-general purpose registers. Let `r3`–`r4` be function arguments/return values and `r5`–r10 for the function arguments. The rest 5 registers are temporary, which results in a following table:

Now, we have to define 3-operands instructions. As soon as we start from the scratch, the first bit of the command can be used to define if it's a branch/call/load immediate commands, in other words – commands that use long immediate values.

Table 1: Registers of processor

Register	Name	Description	Saver
r0	zero	Contains 0.	N.A.
r1	ra	Return address.	Caller
r2	sp	Stack pointer.	Callee
r3-r4	a0-a1	Function parameters/return values.	Caller
r5-r10	a2-a7	Function parameters.	Caller
r11-r15	t0-t5	Temporary registers.	Caller

Second bit defines if a command is used to work with memory (if bit is set), in our case – to load or store word.

Then, three 3-registers commands are defined – to add and subtract registers, and one exotic instruction – `slt` (see description), which can be used together with branching `bnz` instruction to support more complex branching.

Finally, if the `op1` – `op4` is zero, command is then uses 2 or 1 registers only. In our case, we use a bunch of 2-operand commands, that are, however, 3-operand on RISC-V. This was achieved by modifying the initial register value, and in case if value should be preserved, we need an additional move command. *Probably*, there's no other way to support this amount of commands on a 16-bit wide command set.

The reason why `add/sub` commands are made 3-operands commands disregarding the previous observation, is that with those command it is possible to create a move command, in other words, command of copying of a register.

Instruction of accessing processor state register is also present. However, there's no *direct* access to `PC`, so only PIC-code can be executed. As well it should be noted that PC points on the current instruction, not the following one, as it made on ARM cores. But, here we have one branch delay slot.

Moreover, some new 2-operand and 1-operand commands can be added, and the first operations which come to mind are `push` and `pop`. However, they can be realised with `add`, `sw` and `lw` commands, so they somehow can be emulated by the assembler.

Table 2: Command types & encoding

	[15] [14]	[13] [12]	[11:8]	[7:4]	[3:0]
SB:	opcode	I[9:8]	rs1	I[7:0]	
UJ :	0000		rs1	opcode	I[3:0]
I :	opcode	I[8:0]			rd
I/2 :	opcode	I[4]	rs1	I[3:0]	rd
I/3 :	opcode	I[4]	rs1	rs2	I[3:0]
R :	0000		opcode		rd
R/2 :	0000		rs1	opcode	rd
R/3 :	opcode		rs1	rs2	rd

This instruction set also allows to define some useful commands (sugar):

- `mov x, y` as `add x, y, r0`
- `nop` as a `add r0, r0, r0`
- `ret` as a `jalr r1`.

Table 3: Processor ISA

Mnemonics	**op1** [15]	**op2** [14]	**op3** [13]	**op4** [12]	**rs1/opc2** [11:8]	**rs2/opc1** [7:4]	**rd** [3:0]
bnz	1	0	I[9:8]		rs1	I[7:0]	
jal	1	1	0	I[12:0]			
ldi	1	1	1	I[8:0]			rd
sw	0	1	0	I[4]	rs1	rs2	I[3:0]
lw	0	1	1	I[4]	rs1	I[3:0]	rd
add	0	0	0	1	rs1	rs2	rd
sub	0	0	1	0	rs1	rs2	rd
slt	0	0	1	1	rs1	rs2	rd
and	0	0	0	0	rs1	1000	rd
or	0	0	0	0	rs1	1001	rd
xor	0	0	0	0	rs1	1010	rd
sll	0	0	0	0	rs1	1011	rd
slr	0	0	0	0	rs1	1100	rd
not	0	0	0	0	0000	0001	rd
csrr	0	0	0	0	0000	0010	rd
csrw	0	0	0	0	0000	0011	rd
slli	0	0	0	0	I[3:0]	0100	rd
slri	0	0	0	0	I[3:0]	0101	rd
jalr	0	0	0	0	rs1	0000	I[3:0]

Table 4: ISA Description

Mnemonics	**Type**	**Description**
bnz	SB	if (R[rs1] != 0) PC = PC + {imm, 1'b0}
jal	UJ	R[1] = PC; PC = PC + {imm, 1'b0}
ldi	I	R[rd] = imm
sw	I/3	M(R[rs2]) = R[rs1]
lw	I/2	R[rd] = M(R[rs1])
add	R/3	R[rd] = R[rs1] + R[rs2]
sub	R/3	R[rd] = R[rs1] - R[rs2]
slt	R/3	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0
and	R/2	R[rd] = R[rd] & R[rs1]
or	R/2	R[rd] = R[rd] | R[rs1]
xor	R/2	R[rd] = R[rd] ^ R[rs1]
sll	R/2	R[rd] = R[rd] << R[rs1]
slr	R/2	R[rd] = R[rd] >> R[rs1]
not	R	R[rd] = ¬ R[rd]
csrr	R	R[rd] = CSR
csrw	R	CSR = R[rd]
slli	R	R[rd] = R[rd] << imm
slri	R	R[rd] = R[rd] >> imm
jalr	R	PC = R[rs1] + {imm , 1'b0}

4.1.2 Assembly programs

It's obvious that this processor can run some programs, as soon as it has conditional branching and logical instructions. Moreover, some complex logic can be written, as soon as it has an access to processor state flag register.

For the first example, we can show a simple sum function (which looks like one for ARM or RISC-V):

```
1 _sum:
2     add a0, a0, a1
3     ret
```

Listing 12: Sum function

We also can create a function that calls another one (subroutine), for example, to sum up 65408 and 134.

It should be noted that constant 65408 doesn't fit into the immediate part of `ldi` command since it has only 9-bit space for it, which means that immediate value is in range [-256; 255]. Of course this problem can be hidden with assembler sugar, that will be translated into a code like in the following example.

```
: Adding 65408 and 134
_sum_call:
    ; Preserving current ra
    ldi a5, 4
    mov a6, sp
    add sp, sp, a5
    sw ra, 0(a6)

    ; 65408 = (255 << 1 + 1) << 7
    ldi a0, 255
    slli a0, 1
    ldi a5, 1
    add a0, a0, a5
    slli a0, 7

    ldi a1, 134

    jal _sum

    ; Returning back link register
    ldi a5, -4
    add sp, sp, a5
    lw  ra, 0(sp)
    ret

_sum:
    add a0, a0, a1
    ret
```

Listing 13: Sum function call

The next program is a functional copy of one from the first chapter. Here, we found `slt` function very useful, since our pitiful processor have only one type of conditional branch.

```
    mov a4, a0
    bnz a4, _pass1 ; Going to next test if ok
    ldi a0, -1
    ret
_pass1:
    bnz a1, _pass2 ; Well, duplicating code...
    ldi a0, -1
    ret
_pass2:
    mov a0, a3
    slt a2, zero, a5
    bnz a5, _return
; Main part begins
    slli a3, 2
    add a5, a4, a3
    ldi a6, 4
_loop:
    lw a3, 0(a4)
    lw a7, 0(a1)
    add a3, a3, a7
    sw a3, 0(a2)
    add a4, a4, a6 ; Increment pointers
    add a1, a1, a6
    add a2, a2, a6
    sub a7, a5, a4 ; Test if a5-a4 is zero (addresses are the same)
    bnz a7, _loop
_return:
    ret
```

Listing 14: Sum function from chapter 2

4.2 Processor diagram

Instruction Fetch stage is largely the same as for simplified RISC-V processor. There belongs instruction memory (or cache), program counter `PC` and a simple ALU that just adds 2 to current program address.

Instruction Decoding stage is the most complex one, because we have to handle all types of branches on this stage, via additional ALU, comparator and bunch of multiplexers. Decoder generates plenty of signals:

- `Reg_Wr` is an index of register which will be written at the end of pipeline.
- `Reg_WrEn` Specifies if the register will be written.
- `ALU_Op` specifies current ALU operation.
- `ALU_Src` specifies second ALU operand, e. g. register or immediate value.
- `MEM_Rd` specifies if processor is performing memory read.
- `MEM_Wr` specifies if processor is performing memory write.
- `MEM_En` Enables memory, allows to put/get data by selected address.
- `Jump` Enables PC overwrite, used in unconditional branching.
- `Jlink` Enables PC preservation, usually to `ra`.
- `Branch` Enable flag for conditional branching.
- `PC_Src` Selection of PC source: register or current PC.

On this stage also we use sign extender for immediate operands. This module is rather complex, since immediate values for different types of instructions have different length. Moreover, it is able to perform a logical shift left by one bit to make the output value used in the PC calculations.

On this stage Stalling controller can be seen, but it works just like the one in the simplified RISC-V pipeline.

On a **Execution stage** we can see main ALU, which is driven by decoder's signal, having two operands – one direct from register file, and second can be chosen from register and immediate value. We also merged here a Data memory, which is accessed by address, calculated by ALU (which allows memory shift by immediate value), with write, read, memory enable commands.

Processor diagram can be seen in the attached file (`diagram.pdf`).

For a call command execution explanation there's an additional file (`call.pdf`).

Red color here goes for the first clock: we do not care much about current state of `ID` and `EX` stages, we assume that the executed commands are not of branching type. We are performing ordinary fetch on this clock pulse.

Orange color stands for second clock pulse, on the `ID` stage decoder is parsing the command, passing the immediate to the sign extender and generate signals used on this stage: `Jlink` is 1 to store current `PC` in `ra`, `PC_Src` is 0, that means that it will be modified by the immediate value, `Jump` is 1, which means we performing unconditional jump. ALU the calculates jump address, which is the will be stored to `PC`. `Reg_Wr` is then selected to 1, which is `ra` address and `Reg_WrEn` is 1 to write the register. In the same time, command in delay slot is fetched from the memory (remember that we have one).

Violet color stands for the third clock. Branch delay slot command is being executed on the `ID` stage, but, current `PC` is already changed, so desired branched command is being executed at the `IF` stage.

4.2.1 Hazards

This processor can meet Data and Structural hazards, which are resolved by stalling, preformed at the decoding stage. This caused by the minimum amount of stages in the pipeline, however, it seems that this processor cannot work at high frequencies.

Taking a branch on our processor does not need flushing of the next instruction(s), since we have only three pipeline stages and one branch delay slot. That still results in executing a command after the branch one, so sometimes to avoid unexpected behaviour, we should insert `nop`'s in our code.