Project 1 Exectuion Platforms

Hamdane Brini, Wilches Juan, Barau Elena, Marculescu Tudor

January 2025

1 Introduction to RISC-V Instruction Set Architecture

In this section we are focusing on a decomposition of RISC-V hex instruction into the ASM instruction. The instruction format is determined based on the opcode, funct3 and funct7 fields. Table 1 depicts a detailed translation of each instruction from it's hex code to it's format fields.

1.1 Program Instructions Decomposition

Address	Hex Code	Opcode (6:0)	rd (11:7)	funct3 (14:12)	rs1 (19:15)	rs2 (24:20)	funct7 (31:25)	imm[11:0] (31:20)	imm[X:X] (31:25)	imm[X:X] (11:7)	Type
0x0	0x00050893	0010011	10001	000	01010	-	-	000000000000	-	-	I
0x4	0x00068513	0010011	01010	000	01101	-	-	000000000000	-	-	I
0x8	0x04088063	1100011	-	000	10001	00000	-	-	0000010	00000	SB
0xc	0x04058263	1100011	-	000	01011	00000	-	-	0000010	00100	SB
0x10	0x04060063	1100011	-	000	01100	00000	-	-	0000010	00000	SB
0x14	0x04d05063	1100011	-	101	00000	01101	-	-	0000010	00000	SB
0x18	0x00088793	0010011	01111	000	10001	-	-	000000000000	-	-	I
0x1c	0x00269713	0010011	01110	001	01101	-	-	000000000010	-	-	I
0x20	0x00e888b3	0110011	10001	000	10001	01110	0000000	-	0000000	10001	R
0x24	0x0007a703	0000011	01110	010	01111	-	-	000000000000	-	-	I
0x28	0x0005a803	0000011	10000	010	01011	-	-	000000000000	-	-	I
0x2c	0x01070733	0110011	01110	000	01110	10000	0000000	-	0000000	01110	R
0x30	0x00e62023	0100011	-	010	01100	01110	-	-	0000000	00000	S
0x34	0x00478793	0010011	01111	000	01111	-	-	000000000100	-	-	I
0x38	0x00458593	0010011	01011	000	01011	-	-	000000000100	-	-	I
0x3c	0x00460613	0010011	01100	000	01100	-	-	000000000100	-	-	I
0x40	0xff1792e3	1100011	-	001	01111	10001	-	-	1111111	00101	SB
0x44	0x00008067	1100111	00000	000	00001	-	-	000000000000	-	-	I
0x48	0xfff00513	0010011	01010	000	00000	-	-	111111111111	-	-	I
0x4c	0x00008067	1100111	00000	000	00001	-	-	000000000000	-	-	I
0x50	0xfff00513	0010011	01010	000	00000	-	-	111111111111	-	-	I
0x54	0x00008067	1100111	00000	000	00001	-	-	000000000000	-	-	I

Note: Fields marked with dash (-) are not used in the corresponding instruction type. The Type column determines the instruction format: R, I, S, or SB.

- S-type: imm[X:X] (11:7) = imm[4:0] (11:7) and imm[X:X] (31:25) = imm[11:5] (31:25)
- SB-type: imm[X:X] (11:7) = imm[4:1|11] (11:7) and imm[X:X] (31:25) = imm[12|10:5] (31:25)

Table 1: Decomposition of Hex Codes to RISC-V Instruction Format

A point to note is that the immediate fields for UJ and U type instructions are missing from Table 1. This is because the provided program is missing them. There is also an overlap between the immediates of S and SB type instructions, as they share the same positions in the instruction format.

Address	Hex Code	ASM Instruction (ABI)	ASM Instruction (x-registers)
0x0	0x00050893	addi a7, a0, 0	addi x17, x10, 0
0x4	0x00068513	addi a0, a3, 0	addi x10, x13, 0
0x8	0x04088063	beq a7, zero, 64	beq x17, x0, 64
0xc	0x04058263	beq a1, zero, 68	beq x11, x0, 68
0x10	0x04060063	beq a2, zero, 64	beq x12, x0, 64
0x14	0x04d05063	bge zero, a3, 64	bge x0, x13, 64
0x18	0x00088793	addi a5, a7, 0	addi x15, x17, 0
0x1c	0x00269713	slli a4, a3, 2	slli x14, x13, 2
0x20	0x00e888b3	add a7, a7, a4	add x17, x17, x14
0x24	0x0007a703	lw a4, 0(a5)	lw x14, 0(x15)
0x28	0x0005a803	lw a6, 0(a1)	lw x16, 0(x11)
0x2c	0x01070733	add a4, a4, a6	add x14, x14, x16
0x30	0x00e62023	sw a4, 0(a2)	sw x14, 0(x12)
0x34	0x00478793	addi a5, a5, 4	addi x15, x15, 4
0x38	0x00458593	addi a1, a1, 4	addi x11, x11, 4
0x3c	0x00460613	addi a2, a2, 4	addi x12, x12, 4
0x40	0xff1792e3	bne a5, a7, -28	bne x15, x17, -28
0x44	0x00008067	jalr zero, ra, 0	jalr x0, x1, 0
0x48	0xfff00513	addi a0, zero, -1	addi x10, x0, -1
0x4c	0x00008067	jalr zero, ra, 0	jalr x0, x1, 0
0x50	0xfff00513	addi a0, zero, -1	addi x10, x0, -1
0x54	0x00008067	jalr zero, ra, 0	jalr x0, x1, 0

Table 2: RISC-V Instructions with register numbers, symbolic names and addresses

In order to better understand the provided program, Table 2 is introduced to map the hex codes with their corresponding assembly instructions. As an overall view, the program is making use of RV32I instruction set only. The registers are represented in both their symbolic names (ABI) and x-register numbers. in order to facilitate the understanding of the program.

1.2 Branch Delay Slot Concept

The branch delay slot concept is interesting when discussing pipelined processors. Essentially, when a branch is taken, instructions fetched after the branch become invalid if there is no branch prediction mechanism. In early implementations, this issue was handled by filling the instruction following the branch with a no-operation (NOP) in order to preserve the normal functioning of the pipeline. However, the problem with this solution is that cycles are wasted doing nothing.

A more efficient approach consists of executing a useful instruction right after the branch (the delay slot). This instruction is chosen such that it can be executed safely regardless of the branch decision. As a result, the instruction placed in the delay slot is always executed, whether the branch is taken or not. This technique helps reduce the performance penalty caused by pipeline flushing after branch instructions.

As explained in [1]: "*The idea of the branch shadow or delay slot is to recover part of the clock cycles. If the instruction after a branch is always executed, then when a branch is taken, the instruction in the decode slot is also executed, while the instruction in the fetch slot is discarded. Therefore, one has a single wasted cycle instead of two.*" The responsibility of filling the branch delay slot is handled by the compiler during the compilation phase. To do so, the compiler examines a window of instructions located before and after the branch instruction and selects an instruction that can be safely moved into the delay slot without changing the program's behavior.

Figure 1 illustrates this behavior in a classical five-stage pipeline (IF, ID, EX, MEM, WB) using a concrete instruction sequence. In this example, the `BEQ` instruction is evaluated in the execute stage, and the branch decision is therefore known only after several cycles. The instruction immediately following the branch, `OR r7,r8,r9`, is placed in the delay slot and is executed unconditionally, even when the branch is taken. In contrast, the subsequent instruction `XOR r10,r11,r12`, which has already entered the pipeline, is flushed once the branch outcome is resolved. Execution then continues at the branch target labeled `Label`, illustrating how the delay slot allows useful work to be performed while limiting the number of wasted cycles.

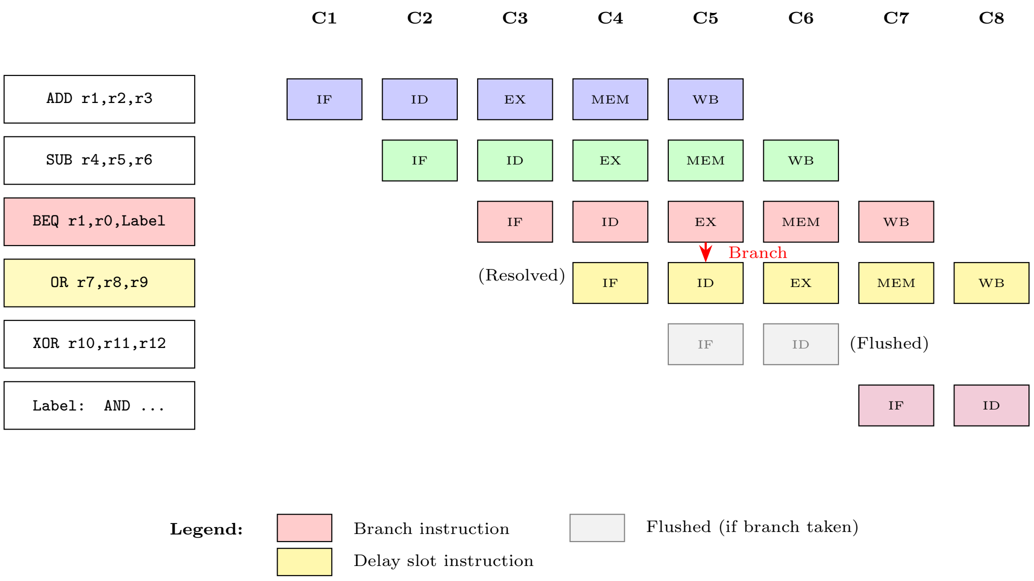

Figure 1: Branch delay slot in a 5-stage pipeline

In terms of advantages and disadvantages, there are several points to consider.
For disadvantages:

- Branch delay slots may create complications in code debugging, since the instruction in the delay slot might have side effects, it may lead to an unexpected state of the registers and memory.
- It adds to the waiting time when trying to execute interruptions, since they will be deffered until the delay slot instruction is executed. This is a problem in the case of real time systems.
- Software compatibility requirements dictate that an architecture may not change the number of delay slots from one generation to the next. This inevitably requires that newer hardware implementations contain extra components to ensure that the architectural behaviour is followed despite no longer being relevant.

Advantages of using branch delay slots include:

- Improved performance in pipelined architectures, since it helps to reduce the number of pipeline stalls caused by branch instructions if no branch prediction is used.
- The use of branch delay slots helps simplify processor design by removing the need for sophisticated branch prediction mechanisms in early architectures. As a result, the hardware becomes easier and less expensive to implement.

Nowadays, the branch delay slot concept bacame obsolete, as modern processors use branch prediction techniques to mitigate the braching performance penalty. This also mitigates the complications of having the compiler finding suitable instructions to fill the delay slots.

1.3 Branch Instructions Analysis

In order to understand better the provided program, it is useful to check the branch instructions and where they lead to.

Address	Conditional branch	Branch to
0x08	beq a7, zero, 64	0x48: addi a0, zero, -1
0x0c	beq a1, zero, 68	0x50: addi a0, zero, -1
0x10	beq a2, zero, 64	0x50: addi a0, zero, -1
0x14	bge zero, a3, 148	0x54: jalr zero, ra, 0
0x40	bne a5, a7, -28	0x24: lw a4, 0(a5)

Table 3: RISC-V Instructions with Addresses

From the table above we can see that there are 4 first conditional branches, that being the addresses 0x08, 0x0c, 0x10 and 0x14, which resemble input value checks. Considering the calling convention for RISC-V, the registers a0-a7 are used for passing function arguments from the caller to the callee. In this case, the supposition of input value checks is valid. The first three jump to a return -1 in case the input values are 0, while the fourth one jumps to a return with the number of elements to be processed, which would be less than or equal to 0.

The last conditional branch at address 0x40 is part of a loop. It checks whether the address stored in register a5, used as a counter, is equal to the last address to be processed, which is stored in register a7. If they are not equal, the program branches back to address 0x24 to continue processing the next elements. If they are equal, the program continues to the return instruction.

1.4 Program Functionality

The program can be divided into three main parts: input validation, processing loop, and return value. The input validation part checks if any of the first three caller arguments are 0. The arguments are passed are actually pointers to the input and output arrays, so they are checked if they are null, which would indicate an invalid memory access, therefore leading to a jump to the section of the program corresponding to return -1. The fourth argument is the number of elements to be processed, which is checked in the register a3 to see if it is less than or equal to 0. In case it is, the program jumps to the section corresponding to return, with the number of elements to be processed.

The processing loop starts at address 0x18 and continues until the branch instruction at address 0x40 It consists of an initial setup for the loop counter in register a5 and for the final value of the counter, which is stored in register a7. Register a7 will store the address of the last element to be processed from one of the input arrays, calculated as the base address plus the number of elements multiplied by 4, the size of each element. Because the elements are 4 bytes long, it can be extrapolated that the arrays are made of 32-bit integers. The loop itself consists of loading the elements from both input arrays, by using registers a5 and a1, which store the current addresses of the elements. Afterwards, the elements are summed and stored in register a4, which is then stored in the output array location pointed by register a2. Finally, the addresses in registers a5, a1 and a2 are incremented by 4 to point to the next elements to be processed. The loop continues until the address in register a5 is equal to the address in register a7. This is ensured by the branch if not equal instruction at address 0x40.

Finally, the return part is reached at address 0x44, where the program jumps back to the calle, using the address stored in register ra. The return value is stored in register a0, which is equal to the number of elements to be processed.

Scenario	**Return value**
input pointers are null	-1
number of elements to be processed ≤ 0	number of elemnts to be processed
processing finished and the result is ready	number of processed elements

Table 4: Return values of the program

2 RISC-V Tool Chain

2.1 Reimagined C Code

The above explanation of the program's functionality allows us to reimagine the C code that could have generated the assembly instructions that were previously described. To do so, one can have an intuition on what certain assembly instructions would mean. For example, for the part of input validation, the conditinal branches checks can be equivalent to the if statements of C language.

```
beq a7, zero, 64
...
addi a0, zero, -1
jalr zero, ra, 0 ⟶ if (in1 == NULL) return -1;
```

There is no direct connection between the register names and C variables, therefore arbitrary names can be assigned to them based on their usage. For example, register a0 can be assigned to the variable "in1", register a1 to "in2", register a2 to "out" and register a3 to "n", representing the two input arrays, the output array and the number of elements to be processed from the arrays.

For the processing loop, one can reimagine it as a do while loop, iterating from 0 to n-1. This is reflected in the assembly instructions which first setup the counter and the final conditional branch to check if it has reached its final value.

```
addi a5, a7, 0
slli a4, a3, 2
add a7, a7, a4
...
bne a5, a7, -28 ⟶ in1_end = in1 + sizeof(int)*n; do { ... } while (in1 != in1_end);
```

As for the operations inside the loop, the lw can be mapped to array accesses in C, the add instruction to the assignment operator and the sw to the assignment operator as well.

```
lw a4, 0(a5)
lw a6, 0(a1)
add a4, a4, a6
sw a4, 0(a2) ⟶ out[i] = in1[i] + in2[i];
```

Therefore, taking into account all of the above observations, a complete C functions that could have generated the provided assembly instructions is represented below:

```c
int addv(int *in1, int *in2, int *out, int n)
{
    if (in1 == NULL)
        return -1;
    if (in2 == NULL)
        return -1;
    if (out == NULL)
        return -1;
    if (n <= 0)
        return n;

    int* in1_end = in1 + sizeof(int)*n;
    do {
        *out = *in1 + *in2;
```

```c
        in1++;
        in2++;
        out++;
    }
    while (in1 != in1_end);
    return n;
}
```

2.2 Compiling the C Code to RISC-V Assembly

By using the RISC-V GCC toolchain, one can compile the above C code into RISC-V object code and then dissasemble it to obtain the assembly instructions. The following command was used in order to produce the object file "se201-prog.o" from the C source file "se201-prog.c":

```
riscv64-linux-gnu-gcc -g -O0 -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -c -o se201-prog.o se201-prog.c
```

A fair mention is that in order to have a compilible C program, one needs to provide a main function as an entry point. In it, the input arrays and the output array are defined, along with the number of elements to be processed. The call to the addv function, which was seen previously, is also made in main.

```c
int main() {
    int n = 50;
    int a[50] = {0};
    int b[50] = {0};
    int result[50] = {0};

    addv(a, b, result, n);
    return 0;
}
```

2.3 Comparison between the generated and the previous instructions

After succefully compiling the C code, one can dissasemble it to obtain the assembly instructions. The following command is used to do so:

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

Then, because the addv function is the one of interest, we can navigate to the corresponding part of the output, containing the label "addv". Below is a snippet of the generated assembly instructions for it:

Generated Assembly Instructions:

```
00000000 <addv>:
   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 <.L2>
    24:   fff00793            li      a5,-1
    28:   0980006f            j       c0 <.L3>

0000002c <.L2>:
    2c:   fd842783            lw      a5,-40(s0)
    30:   00079663            bnez    a5,3c <.L4>
    34:   fff00793            li      a5,-1
    38:   0880006f            j       c0 <.L3>

0000003c <.L4>:
    3c:   fd442783            lw      a5,-44(s0)
    40:   00079663            bnez    a5,4c <.L5>
    44:   fff00793            li      a5,-1
    48:   0780006f            j       c0 <.L3>

0000004c <.L5>:
    4c:   fd042783            lw      a5,-48(s0)
    50:   00f04663            bgtz    a5,5c <.L6>
    54:   fd042783            lw      a5,-48(s0)
    58:   0680006f            j       c0 <.L3>

0000005c <.L6>:
    5c:   fd042783            lw      a5,-48(s0)
    60:   00479793            slli    a5,a5,0x4
    64:   fdc42703            lw      a4,-36(s0)
    68:   00f707b3            add     a5,a4,a5
    6c:   fef42623            sw      a5,-20(s0)

00000070 <.L7>:
    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 <.L7>
  bc:   fd042783        lw      a5,-48(s0)

000000c0 <.L3>:
  c0:   00078513        mv      a0,a5
  c4:   02c12403        lw      s0,44(sp)
  c8:   03010113        addi    sp,sp,48
  cc:   00008067        ret
```

By comparing the generated assembly with the one provided in the first section, one can observe that several differences exist. Firstly, the compiler saves the function arguments on the stack before using them, that being a0, a1, a2 and a3. Then it loads them back from the stack to perform the checks.

Secondly, the check for null pointers, which can be observed in the section from 0x20 – 0x48 is done using the "bnez" instruction instead of a "beq", which instead of jumping to a common return -1, they jump over the return -1 instructions to the next check. This leads to a larger code size and is also less efficient. The efficiency problem arises from the fact that if bnez is taken, when the pointer is valid, then the processor will have to flush the pipeline, if no branch prediction is used.

Thirdly, the loop structure is different, as it can be seen in the 0x5c – 0xb8 section. To compute the end address of the processing loop, the program first takes out of the stack the value of a3, shifts it by 4 then adds it to the address of the first element of the array, taken out of the stack as well. It stores the result back on the stack. In the given assembly, this is done using only three instructions, without using the stack at all. The same phenomenon can be observed for the loop body, in which each each element is taken out of the stack, for example the address of the first input array element, which is used to load the element, then the adress of the second input array element is taken out of the stack to load the element. They are added and then the address of the output array element is taken out of the stack to store the result. Finally, the addresses of the of the three arrays can be retaken out of the stack, incremented by 4 and stored back on the stack.

Finally, the return part is similar, with the difference that actually a ret instruction is used instead of jalr zero, ra, 0.

2.4 Changing the Optimization Level

By switching the optimization level from -O0 to -O, the generated assembly code changes significatly. It is observed that the code size is reduced, and the loop starts to resemble more the code provided in the first section. Below is a snippet of the generated assembly instructions:

```
00000000 <addv>:
   0:   00050793        mv      a5,a0
   4:   00068513        mv      a0,a3

00000008 <.LVL1>:
```

```
     8:    02078e63            beqz    a5,44 <.L4>
     c:    04058063            beqz    a1,4c <.L5>
    10:    04060263            beqz    a2,54 <.L6>
    14:    04d05263            blez    a3,58 <.L2>
    18:    00469893            slli    a7,a3,0x4
    1c:    011788b3            add     a7,a5,a7

00000020 <.L3>:
    20:    0007a703            lw      a4,0(a5)
    24:    0005a803            lw      a6,0(a1)
    28:    01070733            add     a4,a4,a6
    2c:    00e62023            sw      a4,0(a2)
    30:    00478793            addi    a5,a5,4
    34:    00458593            addi    a1,a1,4

00000038 <.LVL4>:
    38:    00460613            addi    a2,a2,4

0000003c <.LVL5>:
    3c:    fef892e3            bne     a7,a5,20 <.L3>
    40:    00008067            ret

00000044 <.L4>:
    44:    fff00513            li      a0,-1
    48:    00008067            ret

0000004c <.L5>:
    4c:    fff00513            li      a0,-1

00000050 <.LVL8>:
    50:    00008067            ret

00000054 <.L6>:
    54:    fff00513            li      a0,-1

00000058 <.L2>:
    58:    00008067            ret
```

Firstly, the stack usage is eliminated completely and the function arguments are used directly from the registers. Secondly, the input validation part is more efficient, as it uses beqz instructions that jump directly to the return -1 section. The same phenomenon happens where a5 and a7 are used as loop counter and address of the last element to be processed, respectively. This version of code is very similar to the one provided in the first section, with only minor differencces in the return sections, that use li instead of addi and ret instead of jalr zero, ra, 0.

Changes in code size occur as well when switching to higher optimization levels, such as -O3.

```
00000000 <addv>:
     0:    00050793            mv      a5,a0
     4:    04050063            beqz    a0,44 <.L8>
```

```
   8:   02058e63        beqz    a1,44 <.L8>
   c:   02060c63        beqz    a2,44 <.L8>
  10:   00469893        slli    a7,a3,0x4
  14:   011508b3        add     a7,a0,a7
  18:   02d05263        blez    a3,3c <.L5>

0000001c <.L4>:
  1c:   0007a703        lw      a4,0(a5)
  20:   0005a803        lw      a6,0(a1)
  24:   00478793        addi    a5,a5,4

00000028 <.LVL2>:
  28:   00458593        addi    a1,a1,4

0000002c <.LVL3>:
  2c:   01070733        add     a4,a4,a6
  30:   00e62023        sw      a4,0(a2)

00000034 <.LVL4>:
  34:   00460613        addi    a2,a2,4

00000038 <.LVL5>:
  38:   fef892e3        bne     a7,a5,1c <.L4>

0000003c <.L5>:
  3c:   00068513        mv      a0,a3
  40:   00008067        ret

00000044 <.L8>:
  44:   fff00513        li      a0,-1

00000048 <.LVL7>:
  48:   00008067        ret
```

Here there are no more output sections that esentially do the same thing, such as in the -O version, in which the li a0, -1 and ret instructions were repeated three times. The instructions are also reordered slightly, but the overall structure remains the same.

3 RISC-V Architecture

4 Procressor Design

4.1 Instruction Set Architecture

For the processor that we are designing, we will have an instruction set based on 16 bit wide instructions. There are 16 registers, all of them being 32 bit wide. The processor is of harvard architecture type.

• R0-R1: Return Registers • R0-R4: Argument Registers (used to pass up to 4 function arguments) • R5-R8: Temporary Registers (not preserved across function calls) • R9-R12: Saved Registers (preserved across function calls) • R13: Stack Pointer (SP) • R14: Return Address Register (RA) • R15: Program Counter (PC) (implicit, cannot be directly modified)

The operands are sign extended.

Arithmetic and logic instructions: - SUM rd, rs1, rs2 ; rd = rs1 + rs2 - DIF rd, rs1, rs2 ; rd = rs1 - rs2 - SHL rd, rs1, rs2 ; rd = rs1 ¡¡ rs2

Load from memory a 32bit value: - MEL rd, rs1, imm ; rd = MEM[rs1 + imm] - Memory load - MES rd, rs1, imm ; MEM[rs1 + imm] = rd - Memory store

Instruction to copy an immediate value to a register: - SET rd, imm ; rd = imm

Define a conditional branch instruction having 1 register operand and 10 bit immediate - BNZ rs1, imm ; if (rs1 != 0) PC = PC + imm * 2

Define an unconditional jump instruction having 1 register operand (read) - GTO rs1 ; PC = rs1

Define a call instruction having 1 imm 9 bits - CALL imm ; RA = PC + 2; PC = PC + imm * 2

Conditional branches, unconditional jumps, and calls in the instruction set have a branch delay slot for a single instruction.

References

[1] James. What is a branch delay slot and why is it used?, 2020. Accessed: June 10, 2024.