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.

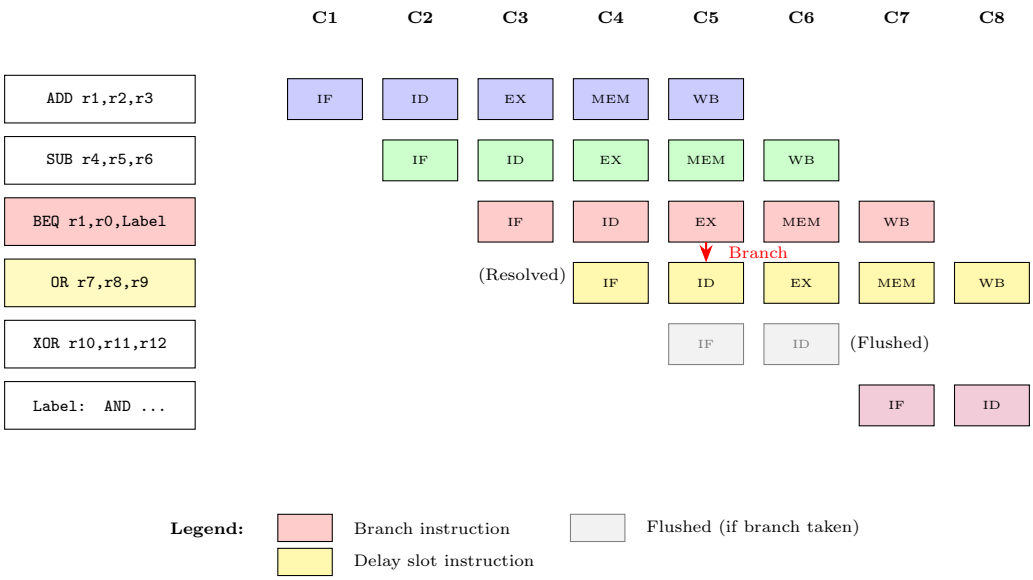

Figure 1: Branch delay slot in a 5-stage pipeline

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 $\longrightarrow$ 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) $\longrightarrow$ 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;
        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

For the following part, a simple RISC-V processor is assumed. It consists of 5 stages Instruction Fetch, Instruction Decode, Execute, Memory Access and Write Back (IF, ID, EX, MEM, WB). Register operands are read during the ID stage and written back in the WB stage. The arithmetic operations, effective address calculations for loads and stores and branch condition evaluation are performed in the EX stage, while data memory accesses for load and store instructions occur in the MEM stage. The design forwards ALU results to dependent instructions in order to not stall the pipeline.

When a load is consumed immediately by the next instruction, ID inserts a one-cycle stall. It is assumed that the memory replies in one cycle. Branches resolve in EX stage, where when taken, the next two fetched instructions are flushed and the PC jumps to the branch target.

3.1 Program Flow (5-stage Pipeline)

Table **??** below presents the complete sequence of instructions executed until the function terminates with a ret instruction. For each instruction, the current architectural state is shown together with a brief explanation of the program behavior. Pipeline hazards, stalls, forwarding events and branch effects are explicitly identified as they occur. The table provides a brief explanation of what happens in the processor and program state. At first the processor has the registers a0, a1 and a2 set to value 0x200. Register a3 has the value 0x2 and all the other registers are set to 0.

The memory contents at the addresses 0x200 through 0x210 is given as follows:

Address	Value
0x200	0x61
0x204	0x20
0x208	0x62
0x20C	0x00
0x210	0x00

Table 5: Initial memory contents

Address	Value
0x200	0xc2
0x204	0x40
0x208	0x62
0x20C	0x00
0x210	0x00

Table 6: Final memory contents after program execution

PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7	Explanation	Hazard/Resolution
0x00	addi a7, a0	0x200	0x200	0x200	0x2	0x0	0x0	0x0	**0x200**	Copy value of a0 (return register) into a7 (limit address).	-
0x04	addi a0, a3	**0x2**	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Copy value of a3 (element count) into a0 (return register).	-
0x08	beq a7, x0, 0x48	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Branch evaluated in EX stage and found not taken ($R[a7] \neq R[x0]$); normal sequential PC progression, no pipeline flush required. Target: 0x48 = 0x08 + 64.	-
0x0c	beq a1, x0, 0x4c	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Branch evaluated in EX stage and found not taken ($R[a1] \neq R[x0]$); normal sequential PC progression, no pipeline flush required. Target: 0x50 = 0x0c + 68.	-
0x10	beq a2, x0, 0x50	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Branch evaluated in EX stage and found not taken ($R[a2] \neq R[x0]$); normal sequential PC progression, no pipeline flush required. Target: 0x50 = 0x10 + 64.	-
0x14	bge x0, a3, 0x54	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Branch evaluated in EX stage and found not taken ($R[x0] < R[a3]$); normal sequential PC progression, no pipeline flush required. Target: 0x54 = 0x14 + 64.	-
0x18	addi a5, a7, 0	0x2	0x200	0x200	0x2	0x0	**0x200**	0x0	0x200	Initialize source pointer: copy value of a7 into a5 (loop cursor).	-
0x1c	slli a4, a3, 2	0x2	0x200	0x200	0x2	**0x8**	0x200	0x0	0x200	Compute byte count for n elements: shift a3 left by 2; value in a4 becomes 8.	-
0x20	add a7, a7, a4	0x2	0x200	0x200	0x2	0x8	0x200	0x0	**0x208**	Compute end pointer: a7 = base + (n*4) = 0x208.	-
0x24	lw a4, 0(a5)	0x2	0x200	0x200	0x2	**0x61**	0x200	0x0	0x208	Compute address in EX stage (0x200); read word in MEM stage (0x61); write result into a4 in WB.	-
0x28	lw a6, 0(a1)	0x2	0x200	0x200	0x2	0x61	0x200	**0x61**	0x208	Compute address in EX stage (0x200); read word in MEM stage (0x61); write result into a6 in WB. Next instruction uses a6 immediately, so one cycle stall in ID.	The pipeline encounters a load-use hazard when the next instruction uses a6; the ID stage inserts one stall cycle and then forwards a6 to the EX stage.
0x2c	add a4, a4, a6	0x2	0x200	0x200	0x2	**0xc2**	0x200	0x61	0x208	Add with forwarding: a4 = a4 + a6 = 0xc2.	The forwarding network supplies a6 to the EX stage, so the ALU completes without stalling.
0x30	sw a4, 0(a2)	0x2	0x200	0x200	0x2	0xc2	0x200	0x61	0x208	Compute address in EX stage (0x200); write word in MEM stage (0xc2); memory [0x200] is updated; store uses forwarded a4.	The store uses the forwarded a4 and writes memory without stalling.
0x34	addi a5, a5, 4	0x2	0x200	0x200	0x2	0xc2	**0x204**	0x61	0x208	Advance loop cursor: a5 = a5 + 4.	-

PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7	Explanation	Hazard/Resolution
0x34	addi a5, a5, 4	0x2	0x200	0x200	0x2	0xc2	**0x204**	0x61	0x208	Advance loop cursor: a5 = a5 + 4.	-
0x38	addi a1, a1, 4	0x2	**0x204**	0x200	0x2	0xc2	0x204	0x61	0x208	Advance source pointer: a1 = a1 + 4.	-
0x3c	addi a2, a2, 4	0x2	0x204	**0x204**	0x2	0xc2	0x204	0x61	0x208	Advance destination pointer: a2 = a2 + 4.	-
0x40	bne a5, a7, -0x1c	0x2	0x204	0x204	0x2	0xc2	0x204	0x61	0x208	Branch evaluated in EX stage and found taken ($R[a5] \neq R[a7]$); flush the next two instructions; set PC to 0x24. Target: 0x24 = 0x40 + (-0x1c).	The EX stage takes the branch; the fetch unit flushes two instructions and sets the PC to 0x24.
0x24	lw a4, 0(a5)	0x2	0x204	0x204	0x2	**0x20**	0x204	0x61	0x208	Compute address in EX stage (0x204); read word in MEM stage (0x20); write result into a4 in WB.	-
0x28	lw a6, 0(a1)	0x2	0x204	0x204	0x2	0x20	0x204	**0x20**	0x208	Compute address in EX stage (0x204); read word in MEM stage (0x20); write result into a6 in WB. Next instruction uses a6 immediately, so one cycle stall in ID.	The pipeline encounters a load-use hazard when the next instruction uses a6; the ID stage inserts one stall cycle and then forwards a6 to the EX stage.
0x2c	add a4, a4, a6	0x2	0x204	0x204	0x2	**0x40**	0x204	0x20	0x208	Add with forwarding: a4 = a4 + a6 = 0x40.	The forwarding network supplies a6 to the EX stage, so the ALU completes without stalling.
0x30	sw a4, 0(a2)	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208	Compute address in EX stage (0x204); write word in MEM stage (0x40); memory [0x204] is updated; store uses forwarded a4.	The store uses the forwarded a4 and writes memory without stalling.
0x34	addi a5, a5, 4	0x2	0x204	0x204	0x2	0x40	**0x208**	0x20	0x208	Advance loop cursor: a5 = a5 + 4.	-
0x38	addi a1, a1, 4	0x2	**0x208**	0x204	0x2	0x40	0x208	0x20	0x208	Advance source pointer: a1 = a1 + 4.	-
0x3c	addi a2, a2, 4	0x2	0x208	**0x208**	0x2	0x40	0x208	0x20	0x208	Advance destination pointer: a2 = a2 + 4.	-
0x40	bne a5, a7, -0x1c	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208	Branch evaluated in EX stage and found not taken ($R[a5] = R[a7]$); normal sequential PC progression, no pipeline flush required. Target: 0x24 = 0x40 + (-0x1c), not taken.	The EX stage resolves the branch as not taken; the PC advances sequentially and no flush occurs.
0x44	ret	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208	Return to `ra`; a0 already holds the element count (n=0x2).	-

Table 7: Program flow with pipeline notes and hazards (changed registers highlighted)

Note: In the actual code, `jalr zero, ra, 0` encodes the `ret` pseudoinstruction, returning to the address held in `ra`.

3.2 Pipeline Diagram

The following diagrams show the 5-stage pipeline progression for all executed instructions. Colors denote pipeline stages; stalls and flushed wrong-path instructions are highlighted.

Stage	Color	Role and Notes
Instruction Fetch	IF	Fetches the instruction at the current PC; sequentially increments PC unless a taken branch in EX overrides it. Wrong-path instructions after a taken branch become flush.
Instruction Decode	ID	Reads source registers and decodes opcode; detects hazards. On a load-use dependency, ID inserts a single stall bubble before the dependent instruction can enter EX.
Execute	EX	Performs ALU operations, effective address calculation, and branch resolution. Forwarding supplies operands from prior EX/MEM results to avoid stalls for ALU dependencies.
Memory	MEM	Reads or writes data memory for loads/stores using the effective address computed in EX. Load results are available for forwarding after MEM.
Writeback	WB	Writes results to destination registers (for ALU and loads). Stores do not write back.
Stall	stall	Represents a bubble introduced to wait for data (e.g. after a load when the very next instruction consumes the loaded value).
Flush	flush	Marks wrong-path instructions fetched before a taken branch resolves in EX; these do not affect architectural state.

Pipeline Overview
The processor uses a classic five-stage in-order pipeline. Instructions advance one stage per cycle unless a hazard forces a stall or a taken branch flushes wrong-path work.

Forwarding sends ALU results from EX and MEM to subsequent instructions, allowing dependent additions (e.g. `add a4,a4,a6`) to proceed without stalling.

A load-use hazard occurs when the very next instruction needs a loaded value (`lw` immediately followed by `add`). The ID stage inserts a one-cycle stall so that MEM can produce the value and the result is then forwarded into EX.

Branch resolution happens in EX. When the branch is taken, the fetch unit flushes the next two wrong-path instructions and redirects the PC to the computed target. When the branch is not taken, the PC advances sequentially.

Stores consume forwarded data (e.g. the new `a4` value) and update memory in MEM. They do not introduce additional stalls in this design.

In the diagrams, each row shows an instruction's stages by cycle (C1, C2, ...). stallcells indicate bubbles that are inserted and flushrows show wrong-path instructions that are discarded after a taken branch.

Setup (PC 0x00–0x20)

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
0x00 addi a7,a0	IF	ID	EX	MEM	WB							
0x04 addi a0,a3		IF	ID	EX	MEM	WB						
0x08 beq a7,zero,64 (NT)			IF	ID	EX							
0x0c beq a1,zero,68 (NT)				IF	ID	EX						
0x10 beq a2,zero,64 (NT)					IF	ID	EX					
0x14 bge zero,a3,64 (NT)						IF	ID	EX				
0x18 addi a5,a7,0							IF	ID	EX	MEM	WB	
0x1c slli a4,a3,2								IF	ID	EX	MEM	WB
0x20 add a7,a7,a4									IF	ID	EX	MEM

Table 8: Setup pipeline overview (PC 0x00–0x20): initial instructions advance without stalls; establishes pointers and byte count for the loop.

Iteration 1 (PC 0x24–0x40, branch taken)

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
0x24 lw a4,0(a5)	IF	ID	EX	MEM	WB						
0x28 lw a6,0(a1)		IF	ID	EX	MEM	WB					
0x2c add a4,a4,a6			IF	ID	stall	EX(EX *fwd*)	MEM	WB			
0x30 sw a4,0(a2)				IF	ID	EX	MEM				
0x34 addi a5,4					IF	ID	EX	MEM	WB		
0x38 addi a1,4						IF	ID	EX	MEM	WB	
0x3c addi a2,4							IF	ID	EX	MEM	WB
0x40 bne a5,a7,-28 (T)								IF	ID	EX(EX *branch*)	
0x44 ret (flushed)									IF	flush	
0x48 addi a0,-1 (flushed)										IF	flush

Table 9: Iteration 1 pipeline overview (branch taken): shows load-use stall and forwarding into add; store updates memory; branch resolves taken and flushes wrong-path instructions.

Iteration 2 and return (PC 0x24–0x44, branch not taken)

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
0x24 lw a4,0(a5)	IF	ID	EX	MEM	WB							
0x28 lw a6,0(a1)		IF	ID	EX	MEM	WB						
0x2c add a4,a4,a6			IF	ID	stall	EX(EX *fwd*)	MEM	WB				
0x30 sw a4,0(a2)				IF	ID	EX	MEM					
0x34 addi a5,4					IF	ID	EX	MEM	WB			
0x38 addi a1,4						IF	ID	EX	MEM	WB		
0x3c addi a2,4							IF	ID	EX	MEM	WB	
0x40 bne a5,a7,-28 (NT)								IF	ID	EX(EX *branch*)		
0x44 ret									IF	ID	EX	WB

Table 10: Iteration 2 pipeline overview (branch not taken): repeats the sequence with forwarding; branch resolves not taken; execution returns without additional flushes.

4 Procressor Design

4.1 Instruction Set Architecture

The processor designed in this project uses a fixed instruction width of 16 bits. It includes 16 general-purpose registers, each 32 bits wide, allowing operations on 32-bit data values.

The processor follows a Harvard architecture, with separate instruction and data memories. This design enables simultaneous instruction fetch and data access, which is well suited for pipelined execution.

4.1.1 Register Usage Convention

Name	Functionality	Description
R0	Zero Register	Always holds the constant value zero; write operations are ignored.
R1	Return Value Register	Hold values returned by functions
R2–R5	Parameter Registers	They carry the arguments that are passed to the functions
R6–R8	Temporal Registers	May be overwritten during function execution
R9–R12	Saved Registers	Retain their contents across function
R13	Stack Pointer (SP)	Indicates the current top of the stack
R14	Link Register (LR)	Contains the address to resume execution call a function
R15	Program Counter (PC)	Control the flow of execution

Table 11: Register Usage and Functionality

4.1.2 Instruction Format and Decoding

The processor is implemented using a fixed 16-bit instruction set architecture that supports six instruction formats: R, I, S, SB, U, and UJ. Each format defines a specific allocation of bits for operands, immediates, and control fields, allowing efficient decoding and execution. In all instruction formats, bits are listed from the most significant bit (MSB) to the least significant bit (LSB), following standard architectural conventions.

- **R-Type (Register Instructions):** These instructions operate exclusively on registers. The format includes a 2-bit `Function2` field, a 4-bit source register `RS2`, a 4-bit source register `RS1`, a 4-bit destination register `RD`, and a 2-bit `Opcode` field.

- **I-Type (Immediate Instructions):** Immediate instructions combine register operands with a constant value. This format consists of a 1-bit `Function1` field, a 5-bit immediate value, a 4-bit source register `RS1`, a 4-bit destination register `RD`, and a 2-bit `Opcode`.
- **S-Type (Store Instructions):** Store instructions write data from a register to memory. The format includes a 1-bit `Function1` field, a 5-bit immediate value used for address calculation, a 4-bit source register `RS2` containing the data to be stored, a 4-bit base register `RS1`, and a 2-bit `Opcode`.
- **SB-Type (Store Branch Instructions):** Conditional branch instructions modify the control flow based on a register value. This format contains a 10-bit immediate value representing the branch offset, a 4-bit source register `RS1`, and a 2-bit `Opcode`.
- **U-Type (Upper Immediate Instructions):** Upper immediate instructions are used to load larger constant values. The format consists of a 10-bit immediate field, a 4-bit destination register `RD`, and a 2-bit `Opcode`.
- **UJ-Type (Unconditional Jump Instructions):** Unconditional jump instructions support control transfer operations such as jumps and procedure calls. This format includes a multi-bit `Function2` field, a 1-bit `Function1` field, and an immediate field of either 11 bits (for call instructions) or a combination of fixed 7 zero bits and a source register `RS1`, along with a 2-bit `Opcode`.

The following table presents the instruction formats to clearly illustrate the bit-level structure of each instruction type.

Type	**Bit Position**															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	Funct2		RS2				RS1				RD				OPCODE	
I	Fun1	Imm[5:0]					RS1				RD				OPCODE	
S	Fun1	Imm[5:0]					RS2				RS1				OPCODE	
SB	Imm[10:0]										RS1				OPCODE	
U	Imm[10:0]										RD				OPCODE	
UJ CALL	Funct2		Fun1	Imm[11:0]											OPCODE	
UJ GTO	Funct2		Fun1	0							RS1				OPCODE	

Table 12: Instruction Formats and Bit Allocation (16-bit ISA)

4.1.3 Instruction Definition

The operands are sign extended.

R-Type (Register Instructions)

Addition between registers - SUM
The `SUM` instruction performs integer addition between two source registers. It follows the instruction format shown below:

`SUM rd, rs1, rs2`: $rd \leftarrow rs1 + rs2$

And the instruction has the following binary representation.

Funct2 [15:14]	RS2 [13:10]	RS1 [9:6]	RD [5:2]	Opcode [1:0]
0 0	4 bits	4 bits	4 bits	1 1

subtract between registers - DIF
The `DIF` instruction performs integer subtract between two source registers. It follows the instruction format shown below:

`DIF rd, rs1, rs2`: $rd \leftarrow rs1 - rs2$

And the instruction has the following binary representation.

Funct2 [15:14]	RS2 [13:10]	RS1 [9:6]	RD [5:2]	Opcode [1:0]
0 1	4 bits	4 bits	4 bits	1 1

Free Instruction

shift right between registers - SHR
The `SHR` instruction performs a logical right shift operation between two source registers. The value stored in `rs1` is shifted right by the number of bit positions specified in `rs2`. Zeros are shifted into the most significant bits.

It follows the instruction format shown below:

`SHR rd, rs1, rs2`: $rd \leftarrow rs1 \gg rs2$

And the instruction has the following binary representation.

Funct2 [15:14]	RS2 [13:10]	RS1 [9:6]	RD [5:2]	Opcode [1:0]
1 0	4 bits	4 bits	4 bits	1 1

I-Type (Immediate Instructions)

Memory Load - MEL

The `MEL` instruction loads a 32-bit value from memory into a destination register. It follows the instruction format shown below:

$$\texttt{MEL rd, rs1, imm: } rd \leftarrow MEM[rs1 + imm]$$

And the instruction has the following binary representation.

Funct1 [15]	Imm5 [14:10]	RS1 [9:6]	RD [5:2]	Opcode [1:0]
1	5 bits	4 bits	4 bits	0 1

S-Type (Store Instructions)

Memory Store Instruction - MES

The `MES` instruction stores a 32-bit value from a register into memory. It follows the instruction format shown below:

$$\texttt{MES rs2, rs1, imm: } MEM[rs1 + imm] \leftarrow rs2$$

And the instruction has the following binary representation.

Funct1 [15]	Imm5 [14:10]	RS1 [9:6]	RD [5:2]	Opcode [1:0]
0	5 bits	4 bits	4 bits	0 1

SB-Type (Store Branch Instructions):

conditional branch - BNZ Instruction

The `BNZ` instruction performs a conditional branch based on the value of a source register. If the content of the register is not zero, the program counter is updated according to the immediate offset.

$$\texttt{BNZ rs1, imm: if } (rs1 \neq 0) \text{ then } PC \leftarrow PC + imm \times 2$$

And the instruction has the following binary representation.

Imm [15:6]	RS1 [5:2]	Opcode [1:0]
10 bits	4 bits	1 0

U-Type (Upper Immediate Instructions):

Move Immediate - SETI Instruction
The `SETI` instruction loads an immediate value into a destination register. It follows the instruction format shown below:

$$\texttt{SETI rd, imm: } rd \leftarrow imm$$

And the instruction has the following binary representation.

Imm [15:6]	RD [5:2]	Opcode [1:0]
10 bits	4 bits	1 1

UJ-Type (Unconditional Jump Instructions)

Unconditional Jump - GTO Instruction
The `GTO` instruction performs an unconditional jump to the address stored in a register. It follows the instruction format shown below:

$$\texttt{GTO rs1: } PC \leftarrow rs1$$

And the instruction has the following binary representation.

Funct2 [15:14]	Funct1 [13]	Fixed 0 [12:6]	RS1 [5:2]	Opcode [1:0]
1 1	0	7 bits in zero	4 bits	1 1

CALL Instruction
The `CALL` instruction performs a procedure call by storing the return address and transferring control to a new program location. It follows the instruction format shown below:

$$\texttt{CALL imm: } RA \leftarrow PC + 2;\ PC \leftarrow PC + imm \times 2$$

And the instruction has the following binary representation.

Funct2 [15:14]	Funct1 [13]	Imm [12:2]	Opcode [1:0]
1 1	1	11 bits	1 1

Additional instructions

NOP

A no-operation (NOP) instruction is implemented using the `SUM R0, R0, R0` instruction. Since register `R0` is hardwired to zero, the operation produces no side effects and does not modify no register.

RET

A return (RET) operation is implemented by performing an unconditional jump to the address stored in the link register. This behavior is achieved using the `GTO R14` instruction, where register `R14` (LR) holds the return address saved during a procedure call.

4.1.4 Test functions in ASM

Add 2 numbers

```
AddNumbers:
    SUM R1, R2, R3    ; R1 = Argument1 + Argument2
    RET               ; Return using link register (R14)
    NOP               ; Delay slot
```

Call the AddNumbers function

```
CallSum:
    SETI  R2, 65408     ; Load first operand (65408) into R2 (First Argument)
    SETI  R3, 134       ; Load second operand (134) into R3 (Second argument)
    CALL  AddNumbers    ; Call AddNumbers function (return address saved in R14)
    NOP                 ; Branch delay slot after CALL
    RET                 ; Return to caller using link register
    NOP                 ; Branch delay slot after RET
```

4.1.5 Translation of C code to our processor

This routine receives three pointers (`a`, `b`, `out`) and a fourth argument `n` indicating the number of elements. It first validates that `a`, `b`, and `out` are non-null and that `n` is a valid (non-negative) length. If the inputs are valid, it iterates from index 0 to $n-1$ and stores `out[i] = a[i] + b[i]` for each position. The loop stops after processing n elements, and the routine returns the original length `n`.

Important details

- The function signature is `int addv(int *in1, int *in2, int *out, int n)`.
- Function arguments are passed as follows: `in1` in `R2`, `in2` in `R3`, `out` in `R4`, and `n` in `R5`.
- The return value is stored in register `R1`. The function returns the value of `counter` unless an error occurs, in which case it returns `-1`.

- It is assumed that integers are 32 bits wide.
- Due to the branch delay slot, a `NOP` instruction is inserted after control-flow instructions that implement branching.
- For the `SETI RS1, LABEL` instruction, it is assumed that the compiler computes and provides the corresponding offset value.

```
1
2  addv:
3      BNZ R2 , 4       ; Check whether input pointer in1 is non-null
4      SETI R12, ERROR  ; Due to the branch delay slot, store the ERROR label address for later
        use
5      GTO R12          ; Jump in case of a null pointer
6      NOP              ; Branch delay slot
7
8      BNZ R3 , 4       ; Check whether the second input pointer is non-null
9      NOP              ; Branch delay slot
10     GTO R12          ; Jump to error handler
11     NOP              ; Branch delay slot
12
13     BNZ R4, 4        ; Check whether the third input pointer is non-null
14     NOP              ; Branch delay slot
15     GTO R12          ; Jump in case of error
16     NOP              ; Branch delay slot
17
18     SETI R10, RETURN ; Store return address in R10
19     BNZ R5, 4         ; Check whether the length argument is non-zero
20     NOP               ;
21     GTO  R10          ; If the 4th argument is zero, return immediately
22     NOP               ; Branch delay slot
23
24     SETI R6, 31       ; Load the number of bits used for the shift operation
25     SHR  R6, R5, R6   ; Shift all bits except the MSB to determine if n is positive or
       negative
26     BNZ R6, 8         ; If the result is 1, the value is negative and execution jumps to
       return
27                       ; Otherwise, execution continues normally
28
29     SUM R6, R2, R0    ; Copy the initial input pointer into another register
30     SETI R8, 1        ; Initialize loop counter
31     SETI R11, 32      ; Size of each vector element, assuming 32-bit integers
32     SUM R12, R5, R0   ; Create a copy of the vector length
33
34     SETI R7, InEND    ; Load address of the routine that computes the end position of the
       pointer
35     GTO  R7           ; Jump to function that finds the last element address
36     NOP               ; Branch delay slot
37
38     GTO R10           ; Jump to return sequence
39     NOP               ; Branch delay slot
40
```

```
InEND:
    DIF R12, R12, R8    ; Decrement the counter by 1
    SUM R6, R6, R11     ; Advance the pointer to the next vector element
    BNZ R12, 4          ; Check whether the end of the vector has been reached
    SETI R10, LOOP      ; Store loop entry address
    GTO  R10            ; End position found, jump back to LOOP
    NOP                 ; Branch delay slot

    GTO R7              ; Continue with the next iteration of the InEnd routine
    NOP                 ; Branch delay slot

LOOP:
    MEL R8, R2, 0     ; Load vector1[i] into a register
    MEL R9, R3, 0     ; Load vector2[i] into a register
    SUM R10, R9, R8   ; Add the loaded values
    MES R4, R10, 0    ; Store the result into out[i]

    SUM R2, R2, R11   ; Advance pointer to the next element of vector1
    SUM R3, R3, R11   ; Advance pointer to the next element of vector2
    SUM R4, R4, R11   ; Advance pointer to the next element of output vector

    DIF R7, R6, R2    ; Subtract initial vector position from the end position
    BNZ R7, 4         ; If the result is non-zero, continue the loop
    SETI R12, RETURN  ; Store return address
    GTO R12           ; Jump to return to finish execution
    NOP               ; Branch delay slot

    SETI R12, LOOP    ; Jump back to the loop to continue copying values
    GTO R12           ;
    NOP               ; Branch delay slot

RETURN:
    SUM R1, R5, R0   ; Return the counter value in R1
    RET              ; Return from function
    NOP              ; Branch delay slot

ERROR:
    SETI R1, -1      ; Return -1 in R1 due to an error
    RET              ; Return from function
    NOP              ; Branch delay slot
```

References

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