

Project 1 Exectuion Platforms

Hamden 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. Some of them not used in certain instruction types, and are marked with a dash (-) in the table. The ultimate factor that determines the instruction type is the "Type" column, which indicates instruction format label R, I, S or SB.

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[11:5] (31:25)	imm[4:0] (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	0xffff00513	0010011	01010	000	00000	-	-	111111111111	-	-	I
0x4c	0x00008067	1100111	00000	000	00001	-	-	000000000000	-	-	I
0x50	0xffff00513	0010011	01010	000	00000	-	-	111111111111	-	-	I
0x54	0x00008067	1100111	00000	000	00001	-	-	000000000000	-	-	I

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. The table head includes just the immediate fields for S out of simplicity, but in the case of SB, they were calculated as depicted in the RISC-V specification from the Project 1 description [1].

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	0xffff00513	addi a0, zero, -1	addi x10, x0, -1
0x4c	0x00008067	jalr zero, ra, 0	jalr x0, x1, 0
0x50	0xffff00513	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 instruction is taken, the instructions that were fetched after the branch instructions become invalid if there is no branch prediction. To avoid this, the branch delay slot declares that the instruction immediately following a branch instruction is always executed, regardless of whether the branch is taken or not. This helps to mitigate the performance penalty associated with branch instructions that invalidate subsequent instructions in the pipeline. As explained in [2]: "The idea of the branch shadow or delay slot is to recover a part of the clocks. If you declare that the instruction after a branch is always executed then when a branch is taken, then the instruction in the decode slot also gets executed, while the instruction in the fetch slot is discarded. Therefore one has a hole of time not two." The branch delay slot will be filled with an instruction that is independent of the branch outcome by the compiler in the compilation phase of the program. It will look in a window of instructions before and after the branch instruction to find a suitable candidate that will come right after it.

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 deferred 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 hardware 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 became obsolete, as modern processors use branch prediction techniques to mitigate the branching 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 callee, 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.

Situation	Return value
input pointers are null	-1
number of elements to be processed $i = 0$	number of elements 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 in C. For example, for the part of input validation, the conditional branches checks can be equivalent to if statements in C.

```
beq a7, zero, 64
...
addi a0, zero, -1
jalr zero, ra, 0  $\rightarrow$  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 for loop in C, 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.

```
bne a5, a7, -28
addi a5, a7, 0
slli a4, a3, 2
add a7, a7, a4  $\rightarrow$  for (int i = 0; i < n; i++) { ... }
```

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:

```
1 int addv(int *in1, int *in2, int *out, int n)
2 {
3     if (in1 == NULL)
4         return -1;
5     if (in2 == NULL)
6         return -1;
7     if (out == NULL)
8         return -1;
9     if (n <= 0)
10        return n;
11
12     for (int i = 0; i < n; i++) {
13         out[i] = in1[i] + in2[i];
14     }
15
16     return n;
17 }
```

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 dissassemble 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 -mcmmodel=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.

```
1 int main() {  
2     volatile int n = 50;  
3     volatile int a[50] = {0};  
4     volatile int b[50] = {0};  
5     volatile int result[50] = {0};  
6  
7     addv(a, b, result, n);  
8  
9     return 0;  
10 }
```

The volatile keyword is used to prevent the compiler from optimizing away the arrays and the variable n, as they are used in the function call to addv. Also, it is important to force the compiler to use load store instruction when operating on the elements of the arrays, such that the generated assembly can be compared with the one provided in the first section.

2.3 Comparison between the generated and the previous instructions

After successefully compiling the C code, one can dissassemble 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:

```

1 00000000 <addv>:
2      0: fd010113      addi    sp,sp,-48
3      4: 02812623      sw      s0,44(sp)
4      8: 03010413      addi    s0,sp,48
5      c: fca42e23      sw      a0,-36(s0)
6     10: fcb42c23      sw      a1,-40(s0)
7     14: fcc42a23      sw      a2,-44(s0)
8     18: fcd42823      sw      a3,-48(s0)
9     1c: fdc42783      lw      a5,-36(s0)
10    20: 00079663      bnez    a5,2c <.L2>
11    24: fff00793      li      a5,-1
12    28: 0980006f      j       c0 <.L3>
13
14 0000002c <.L2>:
15    2c: fd842783      lw      a5,-40(s0)
16    30: 00079663      bnez    a5,3c <.L4>
17    34: fff00793      li      a5,-1
18    38: 0880006f      j       c0 <.L3>
19
20 0000003c <.L4>:
21    3c: fd442783      lw      a5,-44(s0)
22    40: 00079663      bnez    a5,4c <.L5>
23    44: fff00793      li      a5,-1
24    48: 0780006f      j       c0 <.L3>
25
26 0000004c <.L5>:
27    4c: fd042783      lw      a5,-48(s0)
28    50: 00f04663      bgtz    a5,5c <.L6>
29    54: fd042783      lw      a5,-48(s0)
30    58: 0680006f      j       c0 <.L3>
31
32 0000005c <.L6>:
33    5c: fe042623      sw      zero,-20(s0)
34    60: 0500006f      j       b0 <.L7>
35
36 00000064 <.L8>:
37    64: fec42783      lw      a5,-20(s0)
38    68: 00279793      slli    a5,a5,0x2
39    6c: fdc42703      lw      a4,-36(s0)
40    70: 00f707b3      add     a5,a4,a5
41    74: 0007a683      lw      a3,0(a5)
42    78: fec42783      lw      a5,-20(s0)
43    7c: 00279793      slli    a5,a5,0x2
44    80: fd842703      lw      a4,-40(s0)
45    84: 00f707b3      add     a5,a4,a5
46    88: 0007a703      lw      a4,0(a5)
47    8c: fec42783      lw      a5,-20(s0)
48    90: 00279793      slli    a5,a5,0x2
49    94: fd442603      lw      a2,-44(s0)
50    98: 00f607b3      add     a5,a2,a5
51    9c: 00e68733      add     a4,a3,a4

```

```

52  a0: 00e7a023          sw    a4,0(a5)
53  a4: fec42783          lw    a5,-20(s0)
54  a8: 00178793          addi  a5,a5,1
55  ac: fef42623          sw    a5,-20(s0)
56
57  000000b0 <.L7>:
58      b0: fec42703          lw    a4,-20(s0)
59      b4: fd042783          lw    a5,-48(s0)
60      b8: faf746e3          blt   a4,a5,64 <.L8>
61
62  000000bc <.LBE2>:
63      bc: fd042783          lw    a5,-48(s0)
64
65  000000c0 <.L3>:
66      c0: 00078513          mv    a0,a5
67      c4: 02c12403          lw    s0,44(sp)
68      c8: 03010113          addi  sp,sp,48
69      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 puts the code in different sections, with different labels, such as .L2, .L3, .L4 etc, which correspond to parts of the addv function.

Secondly, the check for null pointers, which can be observed in the section .L2, .L4 and .L5 is done using the "bnez" instruction instead of a "beq", which instead of jumping to a common return -1, each "bnez" contains its own jump to the return -1 section. 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 .L8 and .L7 sections. The loop counter is stored on stack, instead of in a register, which leads to additional load and store instructions. It is increment only by 1, instead of 4, and then multiplied by 4 when calculating the address of the element to be processed.

2.4 Changing the Optimization Level

By switching the optimization level from -O0 to -O1, the generated assembly code changes significantly. 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:

```

1  00000000 <addv>:
2      0: 00050813          mv    a6,a0
3      4: 00068513          mv    a0,a3
4
5  00000008 <.LVL1>:
6      8: 04080063          beqz  a6,48 <.L4>
7      c: 04058263          beqz  a1,50 <.L5>
8      10: 04060463          beqz  a2,58 <.L6>
9      14: 04d05463          blez  a3,5c <.L2>
10     18: 00080793          mv    a5,a6
11     1c: 00269713          slli  a4,a3,0x2

```

```

12 20: 00e80833          add    a6,a6,a4
13
14 00000024 <.L3>:
15 24: 0007a703          lw     a4,0(a5)
16 28: 0005a883          lw     a7,0(a1)
17 2c: 01170733          add    a4,a4,a7
18 30: 00e62023          sw     a4,0(a2)
19 34: 00478793          addi   a5,a5,4
20 38: 00458593          addi   a1,a1,4
21 3c: 00460613          addi   a2,a2,4
22 40: ff0792e3          bne    a5,a6,24 <.L3>
23 44: 00008067          ret
24
25 00000048 <.L4>:
26 48: fff00513          li     a0,-1
27 4c: 00008067          ret
28
29 00000050 <.L5>:
30 50: fff00513          li     a0,-1
31
32 00000054 <.LVL5>:
33 54: 00008067          ret
34
35 00000058 <.L6>:
36 58: fff00513          li     a0,-1
37
38 0000005c <.L2>:
39 5c: 00008067          ret

```

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

```

1 00000000 <addv>:
2 0: 00050793          mv     a5,a0
3 4: 04050063          beqz   a0,44 <.L8>
4 8: 02058e63          beqz   a1,44 <.L8>
5 c: 02060c63          beqz   a2,44 <.L8>
6 10: 00269893          slli   a7,a3,0x2
7 14: 011508b3          add    a7,a0,a7
8 18: 02d05263          blez   a3,3c <.L5>
9
10 0000001c <.L4>:
11 1c: 0007a703          lw     a4,0(a5)
12 20: 0005a803          lw     a6,0(a1)
13 24: 00478793          addi   a5,a5,4
14 28: 00458593          addi   a1,a1,4
15 2c: 01070733          add    a4,a4,a6
16 30: 00e62023          sw     a4,0(a2)
17 34: 00460613          addi   a2,a2,4
18 38: ff1792e3          bne    a5,a7,1c <.L4>
19
20 0000003c <.L5>:
21 3c: 00068513          mv     a0,a3

```

```
22 |
23 | 00000040 <.LVL2>:
24 |     40:  00008067             ret
25 |
26 | 00000044 <.L8>:
27 |     44:  fff00513             li      a0,-1
28 |
29 | 00000048 <.LVL4>:
30 |     48:  00008067             ret
```

3 RISC-V Architecture

4 Processor 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 5: 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 6: 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:

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

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:

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

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.

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

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:

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

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:

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

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:

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

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

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

Call the AddNumbers function

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

4.1.5 Translation of C code to our processor

```
1  int *in1 R2, int *in2 R3, int *out R3, int n R4
2
3  addv:
4      BNZ R2 , 4
5      NOP
6      GTO ERROR
7      NOP
8      BNZ R3 , 4
9      NOP
10     GTO ERROR
11     NOP
12     BNZ R4, 4
13     NOP
14     GTO ERROR
```

```

15     NOP
16     BNZ R5,
17
18     SETI R6, 31
19     SHR R6, R5, R6
20     BNZ R6,
21     NOP
22     SUM R6,R2,R0
23     SETI R8,1
24     SETI R9,16
25     SUM R12,R5,R0
26
27
28     GTO RETURN
29     NOP
30
31 InEND:
32     DIF R7, R12, R8
33     SUM R6, R6, R9
34     BNZ R7, 4
35     NOP
36     GTO InEnd
37     NOP
38
39 LOOP:
40     MEL R8, R2, 0
41     MEL R9, R3, 0
42     SUM R10, R9, R8
43     MES R4, R10, 0
44
45     SUM R2, R2, R9
46     SUM R3, R3, R9
47     SUM R4, R4, R9
48
49     DIF R7, R6, R2
50     BNZ R7, 4
51     NOP
52     GTO RETURN
53     NOP
54     GTO LOOP
55     NOP
56
57 RETURN
58     SUM R1, R5, R0
59     RET
60     NOP
61
62 ERROR:
63     SETI R1, -1
64     RET

```

References

- [1] *PR1-SE201-21*, 2021. Unpublished internal document.
- [2] James. What is a branch delay slot and why is it used?, 2020. Accessed: June 10, 2024.