## Objective:

- Design a 10 bit single-cycle CPU that has separate Data and Instruction memory.
- Make the ISA general-purpose enough to be able to run provided general programs.
- Run a benchmark to test out the ISA, through various code examples.

## Design:

- ***Architecture:*** Our ISA will follow a Register-Register/Load-Store addressing class. It will follow the RISC (Reduced Instruction Set Computing) design style with fixed instruction length at 10 bits.

- ***Memory:*** The registers will store 8-bits each. There will be 8 ($2^3$) registers. There are 4 temporary data registers, 3 saved registers, and 1 register ($0) hardwired to the ground. There will be separate Data and Instruction memory. In total there will be 256 ($2^8$) Data memory cells that store 8-bits each and 256 ($2^8$) Instruction memory cells that store 10-bits each. The instruction will thus be read by the computer by breaking it into 4-bits, the opcode, and 6-bits, the rest. Which is possible to do in a single cycle.

- ***Input and Output:*** The CPU will be connected to, for example, a 7-segment display, acting as the output device and, for example, a keyboard, acting as the input device.
  Of the 256 data memory, one memory cell (0x00) will be dedicated to store the values from the input device. This will have the most current input given by the input device (Keyboard) and old input stored here will be overwritten by any new input.
  And there will be one memory cell (0x01) dedicated to storing the values for the output device. New output will overwrite the old output memory. The output device will output whatever is stored in the output memory cells.

- ***Operands:*** Each instruction will contain at most 2 explicit operands. There will be 2 types of operands, register and immediate (3-bit and 6-bit).

- ***Instruction Format:*** We plan to have 3 different formats in order to utilize the 10 bits to the maximum. 4 bits will be dedicated to represent the 16 operation codes and 3 bits will be dedicated to represent the 8 8-bit registers. The 3 formats are described below:
  a) R-Type

| 9 - 6 | 5 - 3 | 2 - 0 |
|---|---|---|
| Opcode (4 bits) | Register1 (3 bits) | Register2 (3 bits) |

  b) I-Type

| 9 - 6 | 5 - 3 | 2 - 0 |
|---|---|---|
| Opcode (4 bits) | Register1 (3 bits) | Immediate (3 bits) |

c) T-Type

| 9 - 6 | 5 - 0 |
|---|---|
| Opcode (4 bits) | Target (6 bits) |

- ***Operations:*** We plan to have 15 operations in total. This number provides a reasonable balance between the functionality and complexity of the ISA and will cover the basic functionalities:
    a) 2 operations are for arithmetic
    b) 3 operations are for logic
    c) 2 operations for branching (Conditional/Unconditional)
    d) 2 operation for conditional statement
    e) 4 operations are for data transfer
    f) 2 operations for IO (Input/Output)

## Instruction Table

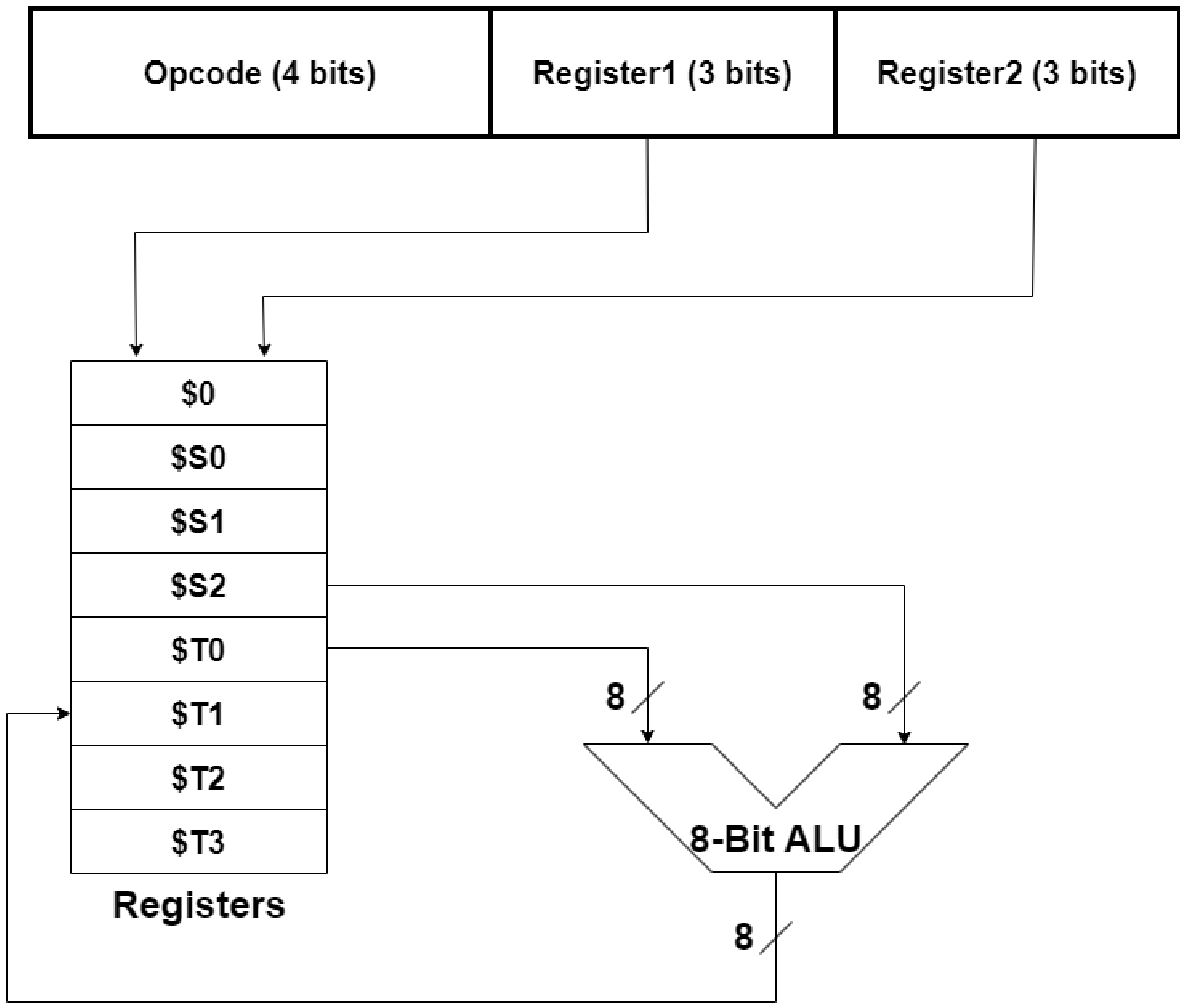
| No. | Instruction | Type | Format | Example | Meaning | Opcode |
|---|---|---|---|---|---|---|
| 1 | Add | Arithmetic | R | Add $S1, $S2 | $S1 = $S1 + $S2 | 0000 |
| 2 | Sub | Arithmetic | R | Sub $S1, $S2 | $S1 = $S1 - $S2 | 0001 |
| 3 | NAND | Logical | R | NAND $S1, $S2 | $S1 = $S1 NAND $S2 | 0010 |
| 4 | Sll | Logical | I | Sll $S1, 3 | $S1 = $S1 << 3 | 0011 |
| 5 | Srl | Logical | I | Srl $S1, 6 | $S1 = $S1 >> 6 | 0100 |
| 6 | J | Branch (Unconditional) | T | J L1 | Branch to the instruction labeled L1 | 0101 |
| 7 | Beqz | Branch (Conditional) | T | Beqz L1 | If ($T1 == 0)<br>Then branch to the instruction labeled L1<br>Else continue to the next instruction | 0110 |
| 8 | Seq | Conditional | R | Seq $S1, $S2 | If ($S1 == $S2)<br>Then $T1 = 1<br>Else $T1 = 0 | 0111 |
| 9 | Slt | Conditional | R | Slt $S1, $S2 | If ($S1 < $S2)<br>Then $T1 = 1<br>Else $T1 = 0 | 1000 |
| 10 | LW | Data Transfer | R | LW $S1, $S2 | $T0 = Mem($S2 + $S1) | 1001 |
| 11 | SW | Data Transfer | R | SW $S1, $S2 | Mem($S2 + $S1) = $T0 | 1010 |
| 12 | Init | Data Transfer | T | Init 20 | $T0 = 20 | 1011 |
| 13 | Cpy | Data Transfer | R | Cpy $S1, $T0 | $S1 = $T0 | 1100 |
| 14 | In | IO (Input) | I | In $S1 | $S1 = Mem(0x00)<br>0 will be stored as the immediate. | 1101 |
| 15 | Out | IO (Output) | I | Out $S1 | Mem(0x01) = $S1<br>1 will be stored as the immediate. | 1110 |

## Register Table

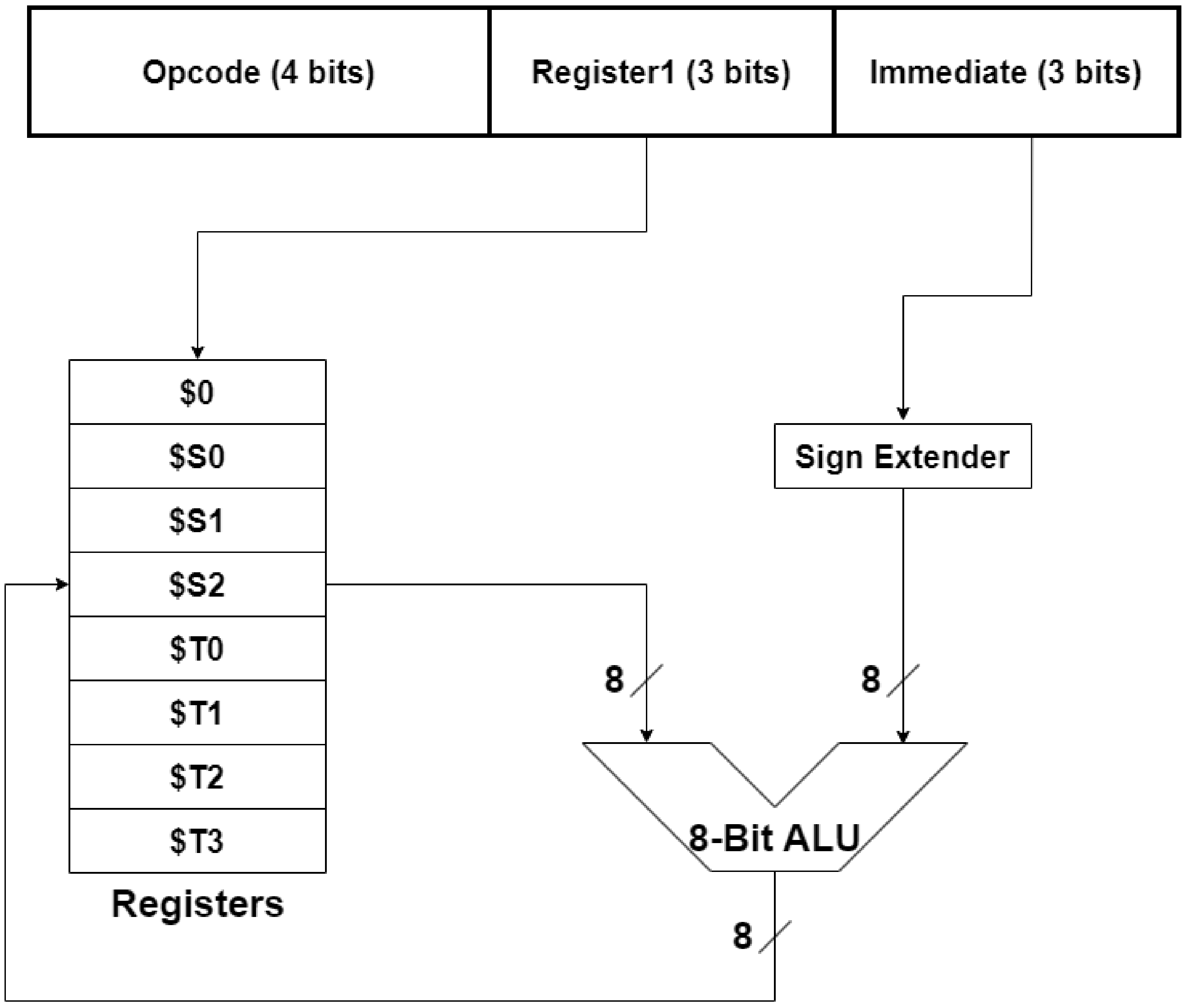
| Register Number | Conventional Name | Code | Usage |
|---|---|---|---|
| $0 | $Zero | 000 | Hard-wired to ground or 0 |
| $1 | $S0 | 001 | Saved registers, preserved by subprograms. |
| $2 | $S1 | 010 | |
| $3 | $S2 | 011 | |
| $4 | $T0 | 100 | Temporary registers, not preserved by subprograms.<br><br>$T0 is special, since it is hardcoded to be used by LW, SW, and Init. In the same way $T1 is used as default by Beqz, Seq, and Slt.<br>However, these can still be used normally like the other temporary registers. |
| $5 | $T1 | 101 | |
| $6 | $T2 | 110 | |
| $7 | $T3 | 111 | |

## Addressing Modes:

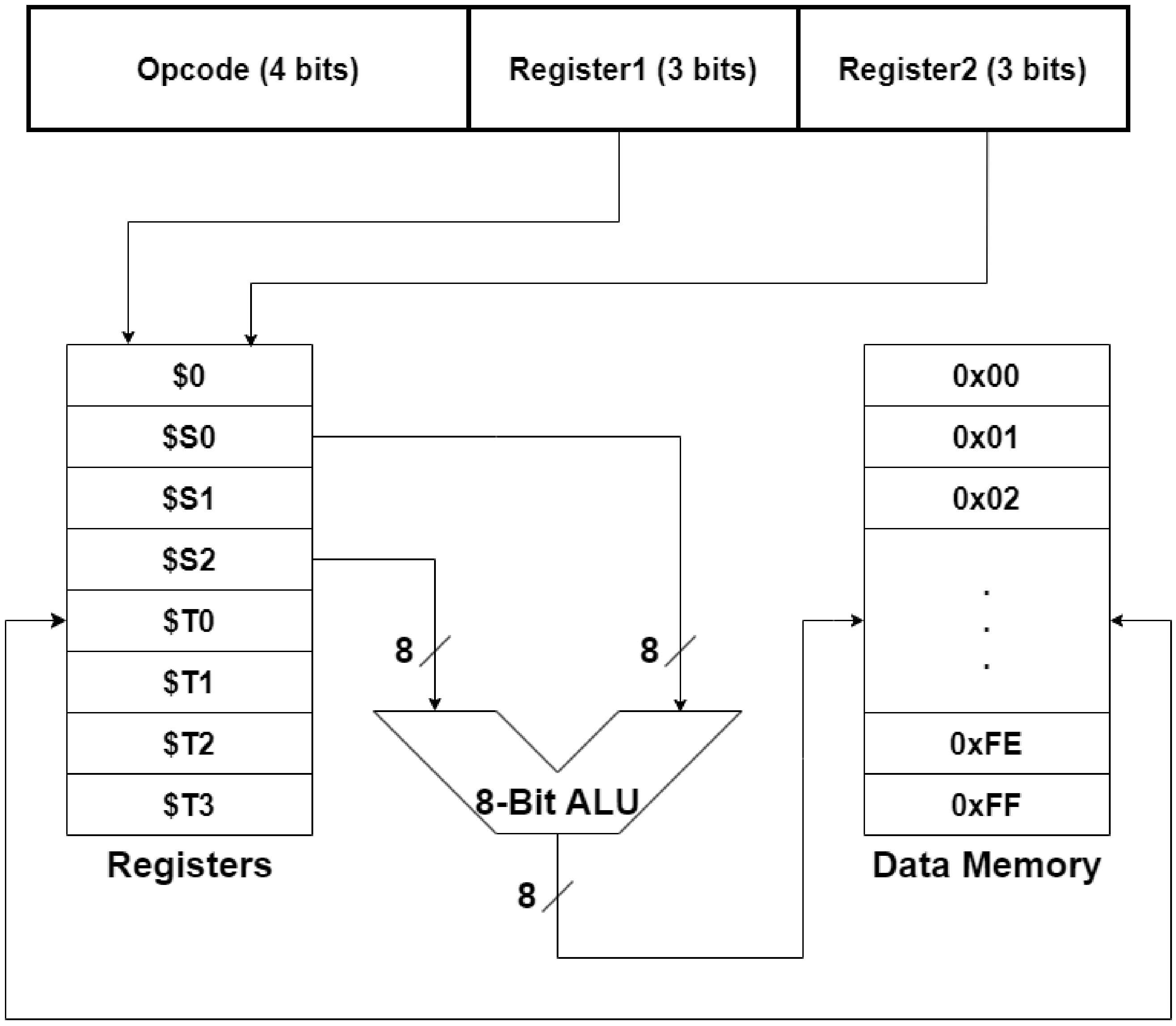
- ***Register Addressing:*** Used by the instructions Add, Sub, NAND, and Cpy. Here the instruction points directly to the registers that contain the operands and stores the result in register1. This addressing is also used by Seq and Slt, but the destination is hardwired to $T1. For Cpy, register2 is the destination and register1 is one of the input to the ALU like normal, but the other input to the ALU is 0.



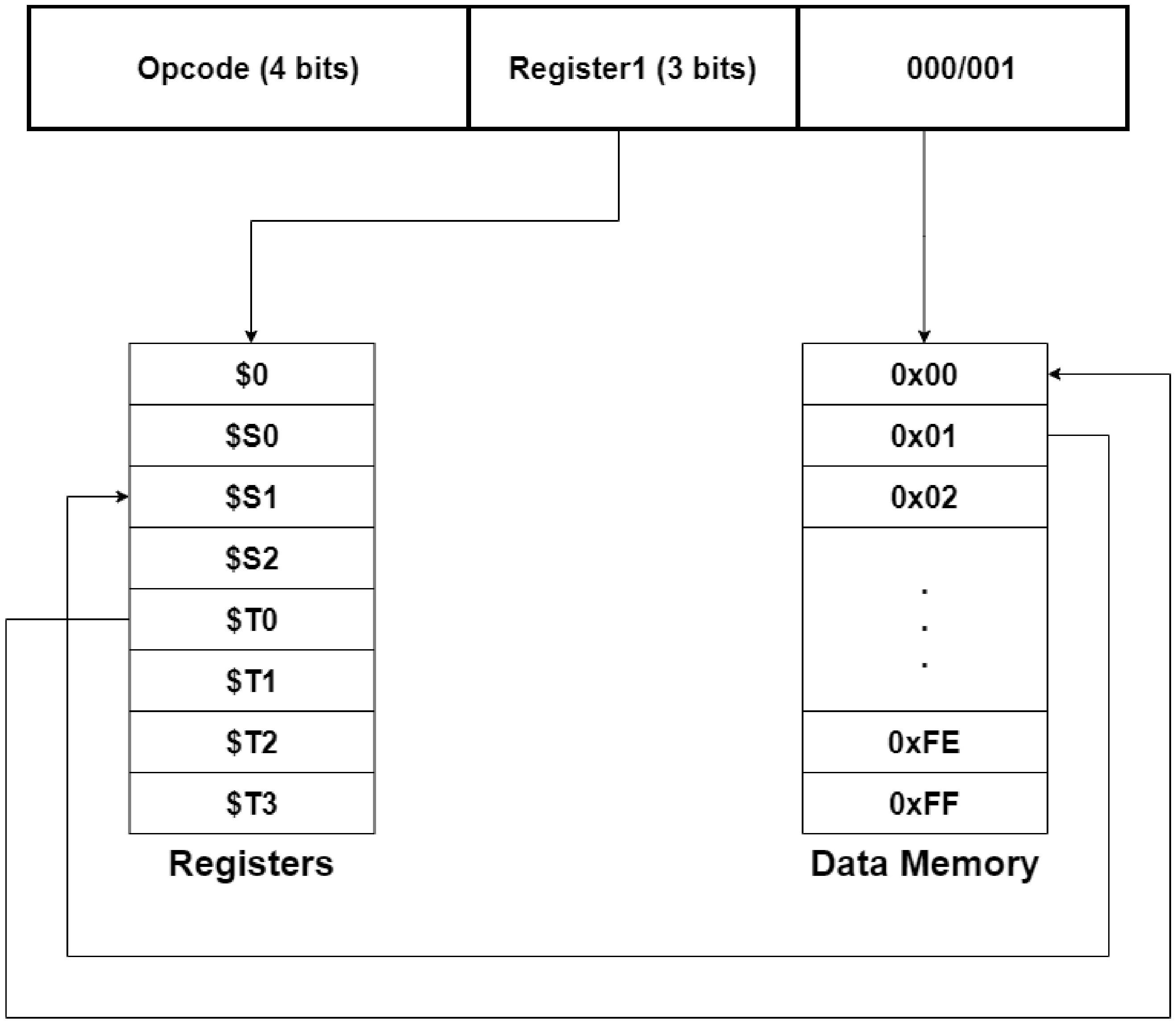
- ***Immediate-Register Addressing:*** Used by instructions like Sll and Srl. Here one of the operands is explicitly mentioned in the instruction, and the instruction points to the other operand. The result is then stored in register1.



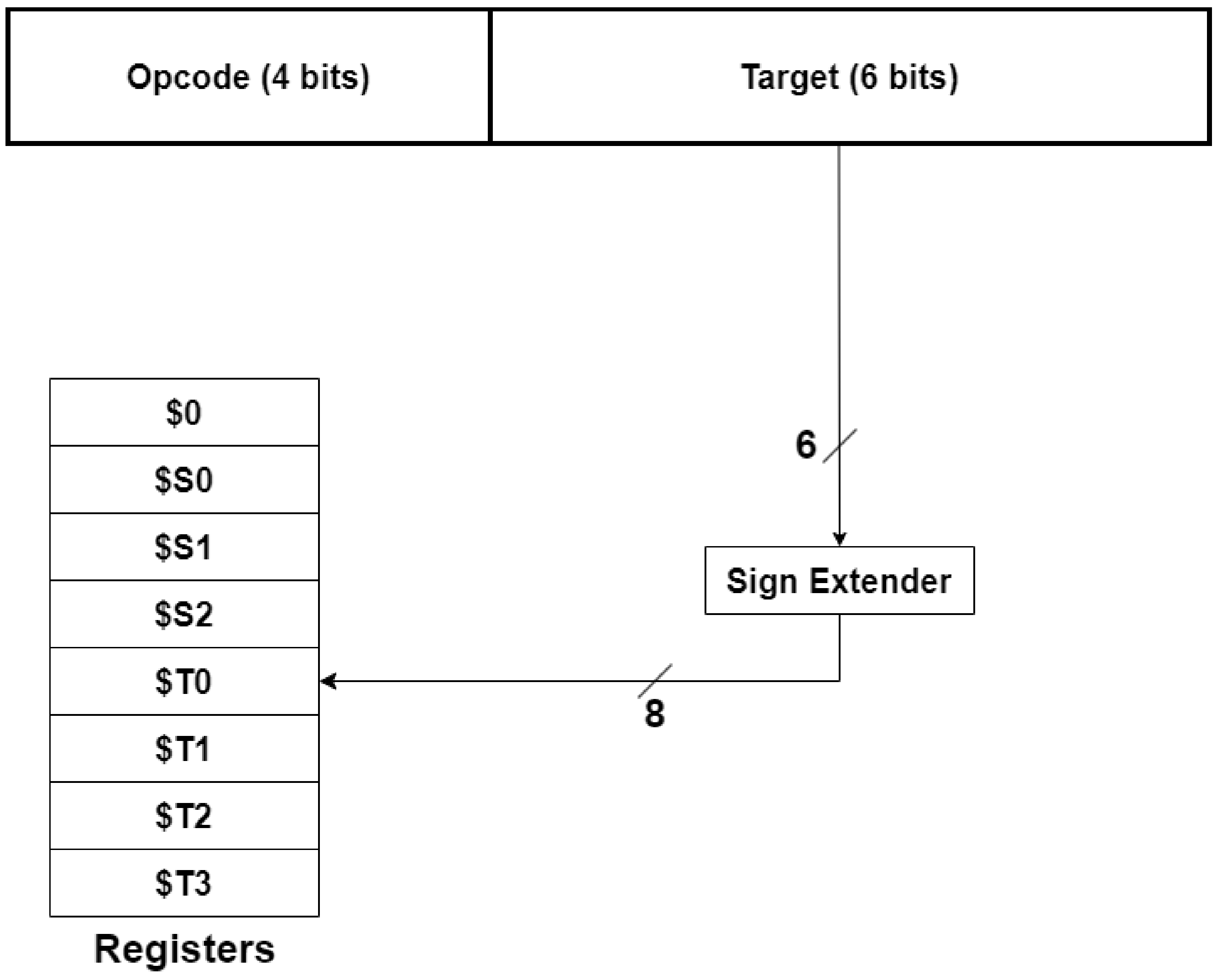
- ***Memory-Register Addressing:*** Used by LW and SW. Here the address of the operand, in the data memory, is found by adding the offset to the pointer given in the instruction. Then this operand is stored in the register $T0 for LW and the value in the register $T0 is stored in the destination for SW.



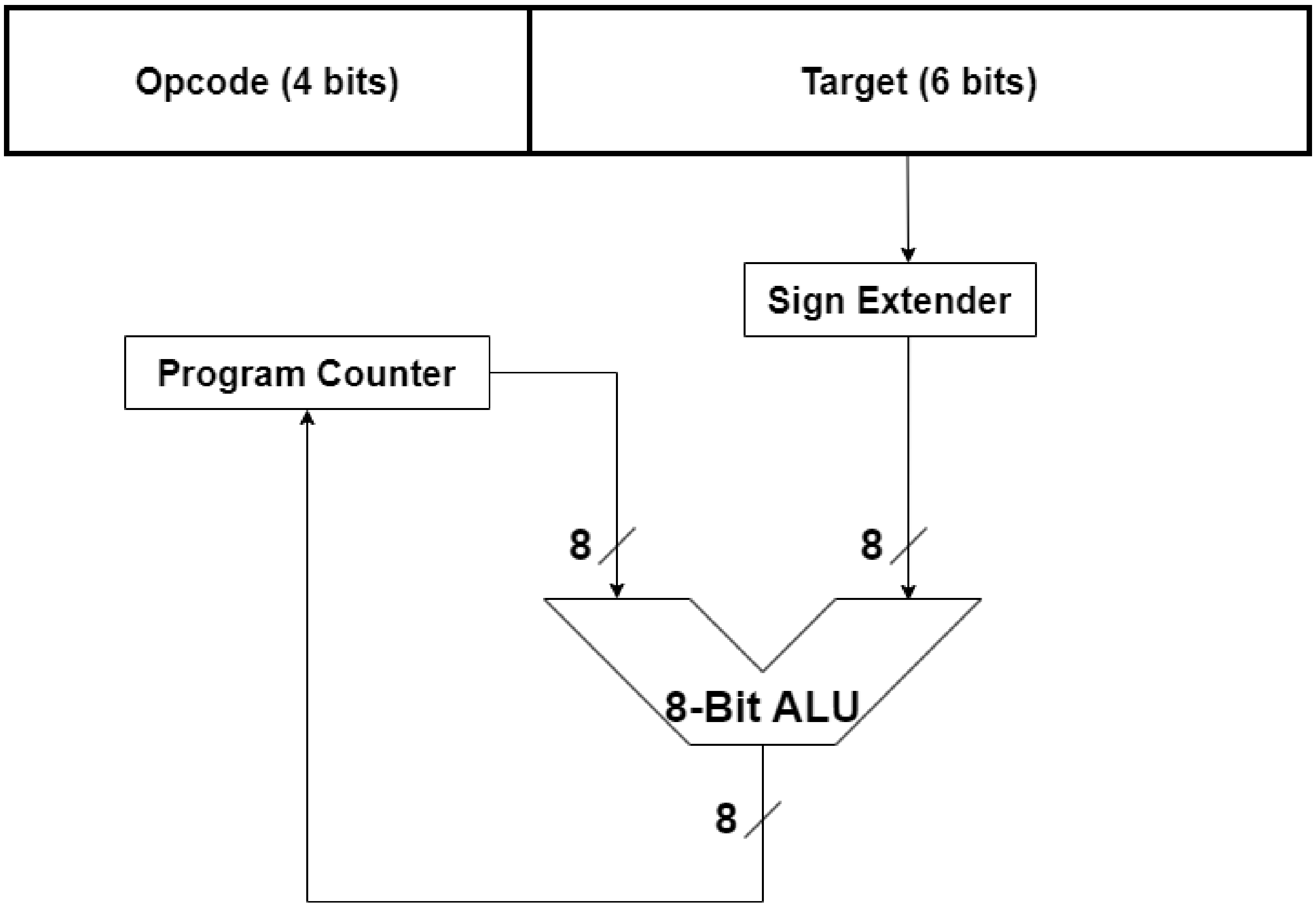
- ***Input/Output Addressing:*** This is used by In and Out. In basically just stores the content of the register into the memory cell 0x00. And Out basically just stores the content of the memory cell 0x01 into the desired register.



- ***Initialization Addressing:*** Used by Init. It just transfers the value from the immediate to register $T0.



- ***Branch Addressing:*** Used by J and Beqz. J adds the value in the target unconditionally but Beqz only adds the target only if their condition is met.



## Benchmark:

We had to design our ISA focusing on 3 categories of programs:

- Simple arithmetic and logic operations
- Programs that require checking conditions
- Programs with loops

Below we provided examples alongside explanations of the purpose of each instruction, to prove that our ISA works on all 3 categories of programs.

***Simple Arithmetic operations:***

C = (A+B) - 6

//Where A, B, C are $S0, $S1, $S2 respectively.

| Code | Explanation |
|---|---|
| Add $S0, $S1 | A = A + B |
| Cpy $S2, $S0 | C = A |
| Init 6 | $T0 = 6 |
| Sub $S2, $T0 | C = C - $T0 or C = C - 6 |

Here, the calculation is done fairly easily and stored in A, and then copied over to C. And since there are no instructions like Subi, we just initialize $T0 and use Sub like normal. Thus it is possible to do all simple arithmetic operations since we can do addition, subtraction, and assignment.

***Simple Logical operations:***

A = (A AND B) << 2

//Where A and B are $S0 and $S1 respectively.

| Code | Explanation |
|---|---|
| NAND $S0, $S1 | A = A NAND B |
| NAND $S0, $S0 | A = A NAND A or A = NOT A |
| Sll $S0, 2 | A = A << 2 |

Our ISA does not explicitly have the AND operation, but we do have the universal gates NAND implemented. And so the logical operation was easily done. This alongside the shifting operations prove that all types of logical operations are possible in this ISA.

***<u>Programs that require checking conditions:</u>***

```
If (A < B)
  Output A
Else
  Output B
//Where A, B are $S0, $S1 respectively.
```

| Code | Explanation |
|---|---|
| Slt $S0, $S1 | If (A < B) $T1 = 1 Else $T1 = 0 |
| Beqz L1 | If ($T1 == 0) branch to L1 |
| Out $S0 | Mem(0x01) = $S0 or Output A |
| J L2 | Branch to L2 |
| L1: Out $S1 | Mem(0x01) = $S1 or Output B |
| L2: | Label |

Here the program basically wants us to output the minimum of A and B. This can be easily done using Slt, Beq, and J. This example thus showcases that our ISA can do all types of conditional programs.

***<u>Programs with loop:</u>***

```
For (i = Input; i <= 5; ++i)
  Output i
//Where i is $S0
```

| Code | Explanation |
|---|---|
| In $S0 | i = Mem(0x00) or i = Input |
| Init 6 | $T0 = 6 |

| | |
|---|---|
| Cpy $T2, $T0 | $T2 = $T0 or $T2 = 6 |
| Loop: Slt $S0, $T2 | If ($S0 < 6) $T1 = 1 Else $T1 = 0 |
| Beqz Exit | If ($T1 == 0) branch to Exit |
| Out $S0 | Mem(0x01) = $S0 or Output i |
| Init 1 | $T0 = 1 |
| Add $S0, $T0 | i = i + $T0 or i = i + 1 |
| J Loop | Branch to Loop |
| Exit: | Label |

It is thus possible to include loops through our ISA. Although the code becomes very large and cumbersome for our ISA, it is a necessary sacrifice for the increased versatility. One may notice that most of the temporary registers are used for looping, and may state that the ISA cannot do complicated operations inside the loop. But that is not true since the temporary registers can be used normally for the operations and then initializing the correct number to any one register right before branching.

## Reasoning:

Below are the reasoning for the individual operations.

| No. | Instruction | Reasoning |
|---|---|---|
| 1 | Add | Basic implementations of Add and Sub.<br>We did not include Addi and Subi since we saw that it is not very practical to include this when our immediate is only 3-bits |
| 2 | Sub | |
| 3 | NAND | NAND is included in this ISA since it is a universal gate. |
| 4 | Sll | Basic implementations of left and right bit shifting.<br>And since the registers and data memory only holds 8-bits, a 3-bit immediate, which can hold 0 - 7, to represent the amount of shifting is perfect. |
| 5 | Srl | |
| 6 | J | Basic implementation of the Jump instruction.<br>The amount to jump is calculated in the assembler and stored in the T format. Done this way to make the programming process easier. Example:<br>J L1<br>L1: Add $T2, $S0 |
| 7 | Beqz | Beqz stands for Branch if Equal to Zero. Basic implementation of Beq but it only |

| | | |
|---|---|---|
| | | branches if $T1 is equal to zero.<br>We decided to hardcode the equality check to be done by $T1 since 6 bits are needed to store where to branch.<br>This was added to streamline conditional branching, and works normally in combination with Seq and Slt. |
| 8 | Seq | Seq stands for Set if Equal. We decided not to go with Beq since implementing it with our instruction formats leads to branching being tedious. Since, branching is supposed to be used commonly, we decided to go with this and added Beqz. |
| 9 | Slt | Basic implementation of Slt. Since we could not have 3 operands, we made it to change $T1 depending on the answer. We decided to go with $T1 since we wanted $T1 to be associated with conditional statements and branching. |
| 10 | LW | Basic implementation of LW and SW. These provide the only communication between the registers and the memory. Since we could not have 3 operands, it is hardcoded to access $T0. We did not hardcode different instruction access different registers in order to avoid confusion.<br>We also made the decision to not make the offset, an immediate value, for better versatility and range. |
| 11 | SW | |
| 12 | Init | Since, we removed Addi and Subi for good reasons, we added these in order to deal with registers and immediate better and more effectively. |
| 13 | Cpy | |
| 14 | In | The only instruction connecting registers and the input and output device. Only one memory was allocated for the input and output each since we thought that having more could complicate the architecture, especially since we can just use our other instructions to store the input and outputs to other data memory cells if needed.<br>The immediate stores the memory address needed for the instruction (0 for In and 1 for Out) |
| 15 | Out | |

We decided to have 256 separate data and instruction memory since that is the maximum number that the values stored in our registers can help access. Overall we think it is the correct decision since programs may have many instructions and may have big arrays and other data.

Our other decisions regarding the ISA are rather self explanatory or explained earlier. After many iterations of the ISA, this is the final ISA that we think our CPU would be able to tackle any high-level programs thrown at it.

We are also designing the assembler to be case insensitive. The Assembler will convert the assembly instruction in Input.txt to machine code in Output.txt by default. But this can be configured through command line arguments, which will allow us to choose a specific input file or a specific input and output file. The assembler has a lot of checks to prevent errors in assembling. If any error is present in the input assembler instruction, it will point it out along with the erroneous line in the output file. And will also print in the console whether the assembling was done with or without any errors. A blank line in the input file does not count as an error since we figured blank lines are useful for writing assembly code more orderly. The machine code will also be outputted in hexadecimal for ease of use.

Thank you for taking the time to read our ISA Design (1st milestone).

[END]