The purpose of our CPU is to perform Moving Window Integration on data read from the Data Memory.

To be able to perform Moving Window Integration we need the following arithmetic operations: Addition, subtraction, and division by 30. On top of that we need to be able to read and write from memory, and we need to jump to specific instruction memory addresses, so that we can loop through the same instructions again. To realize the above requirements, we decided to have our processor be able to perform the instructions as listed below.

**Operations:**

|  |
| --- |
| NOP |
| LOAD |
| STORE |
| ADD |
| ADDI |
| SUB |
| SR5 |
| BRANCH |

The NOP operation is a null operation, which does nothing at all, other than wasting clock cycles if called. This operation is used to halt the processor while waiting for Data Memory etc.  
Gezel also has a bug at the start of execution, where it performs the first operation twice, therefore we call NOP as the first operation in the beginning of our instruction set, to simply neutralize this bug.

LOAD is used to read from data memory. It reads a value from a data memory address, and places it into a register.

STORE does the opposite of LOAD, and is used to write to data memory. It specifies a data memory address and places writes data to that address.

ADD is addition of two registers, which sum is placed into a third register.

SUB acts similarly to ADD, except it performs subtraction instead of addition.

Where ADDI is simply addition, but adding a constant to a register value, rather than adding two registers values together, which ADD does.

SR5, which stands for Shift-Right-5, is how we implemented division. Right shifting the bits 5 times is the same as dividing by 32, which is close to dividing by 30. Of course, this does have some consequences, but it is a much faster implementation of division, than alternate methods. An explanation and the extend of the consequences will be given in section \ref{implementation}.

Finally, BRANCH is the operation used to jump to a specified instruction memory address. It simply sets the PC value to 3, since our instruction set is designed such that we will only ever BRANCH to address 3.

**OP-code structure:**

[OP] [ARG] [ARG] [ARG]

Where each [ARG] can be either a constant, zero value, a register, or a memory address. Not all operations/instructions takes three arguments. It can range all the way from zero to three arguments, depending on the operation.

The following table shows how each individual 32-bit instruction would be used, where [n:m] indicates bit n to bit m inclusive, of the instruction for which the argument is assigned.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operation /  Instruction | OP  [31:29] | Arguments  [28:26] | [25:23] | [22:20] | [19:15]…[14:0] |
| NOP | 000 | 0 | 0 | 0 | 0 |
| LOAD | 001 | [REG] | [ADR] | 0 | 0 |
| STORE | 010 | 0 | [REG] | [ADR] | 0 |
| ADD | 011 | [REG] | [REG] | [REG] | 0 |
| ADDI | 100 | [REG] | [REG] | [CONST] | …[CONST]… 0 |
| SUB | 101 | [REG] | [REG] | [REG] | 0 |
| SR5 | 110 | [REG] | [REG] | 0 | 0 |
| BRANCH | 111 | 0 | 0 | 0 | 0 |

STORE has a filler argument as its first argument, which value it ignores. This is simply because of the way our instruction is sliced into different wires so that the [REG] will be the [25:23] bits rather than the [28:26] bits, and have [ADR] as [22:20] since these are the bits that are used for command/address to the Data memory, when writing.

The constant for the ADDI instructions ranges from the [22:15] bits of the instruction.

**Moving Window Integration**

From all of the operations above and the 8 registers available, our instruction set for Moving Window Integration was planned as follows:

R0 always contains the value 0.  
R1 contains x\_i. The i'th input value.  
R2 contains x\_i+31. The i+31th input value.  
R3 contains the i’th Data Memory address. Index number i.  
R4 contains the i+31’th Data Memory address. Index number i+31  
R5 contains the sum of most 32 most recent input values.  
R6 contains the average, R5 divided by 32 (shifted 5). I.e. the i'th output, mwi\_i.  
R7 contains the output Data Memory address. The address that we want to store the output at.

Where RN is the N’th register out of 8.

And so our instruction set was constructed:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Instruction (Assembly) | | | | Instruction (OP-code) 32-bit | | | |
| 0 | NOP |  |  |  | 000 | 000 | 000 | 000…0 |
| 1 | ADDI | R4 | R0 | -31 | 100 | 100 | 000 | 11100001…0 |
| 2 | ADDI | R7 | R0 | 251 | 100 | 111 | 000 | 11111011…0 |
| 3 | LOAD | R1 | (R3) |  | 001 | 001 | (011) | 000…0 |
| 4 | LOAD | R2 | (R4) |  | 001 | 010 | (100) | 000…0 |
| 5 | ADD | R5 | R5 | R1 | 011 | 101 | 101 | 001…0 |
| 6 | SUB | R5 | R5 | R2 | 101 | 101 | 101 | 010…0 |
| 7 | SR5 | R6 | R5 |  | 110 | 110 | 101 | 000…0 |
| 8 | STORE | 000 | R6 | (R7) | 010 | 000 | 110 | (111) |
| 9 | ADDI | R3 | R3 | 1 | 100 | 011 | 011 | 00000001…0 |
| A | ADDI | R4 | R4 | 1 | 100 | 100 | 100 | 00000001…0 |
| B | ADDI | R7 | R7 | 1 | 100 | 111 | 111 | 00000001…0 |
| C | BRANCH |  |  |  | 111 |  |  |  |

Address 1 and 2 is used to initialize the initial indexes. R4 is set to -31, as the i-31’th index. R7 is set to 251 which means we store our output values at address 251 and above. This is because address 0-250 contains the ECG data we use.

On 3 and 4 we load the x\_i, and x\_i-31, into R1 and R2.  
On 5, x\_i is added to the sum, and on 6, x\_i-31 is subtracted from the sum.  
Then on 7, the sum in R5 is shifted right 5 times (divided by 32) and placed into R6.  
Finally, on address 8, the output (R6) is then written to the data memory address stored in R7.  
Next, all the indexes are incremented on 9, A, B. And, on address C, we branch back to address 3, and so the loop keeps going.