Jordan Cottle, Michael Russell, Harshil Suthar

# ECGR 3183 Project

## Instruction Set

Each of our instructions is encoded in a 32 bit word. The opcode portion of the instruction maps to a 5 bit value at the most significant bits. Following that the destination register and the two source register addresses follow taking up 4 bits each. The first source register specified corresponds to the 'top' register in the design. This register will always be sent to the ALU, and is used for instructions that only require one input or who's values need to be passed through the ALU unmodified. The second register can be used as a second value for the ALU, but also can be redirected into the data memory for storage. For branching instructions the label will be encoded into the final 10 bits of the instruction, allowing for a branch destination to be any available memory address. That leaves 5 unused bits between the final source register and the beginning of the branch label for a total of 32 bits.

Our instruction set is as follows:

| Instruction | Op Code | Syntax | Description | Condition Codes |
|---|---|---|---|---|
| Set | 10000 | SET Rd #Value | Sets a specified value into a register | None |
| Load | 10001 | LOAD Rd Ri | Takes a value from memory at address in register Ri and stores it in register Rd | None |

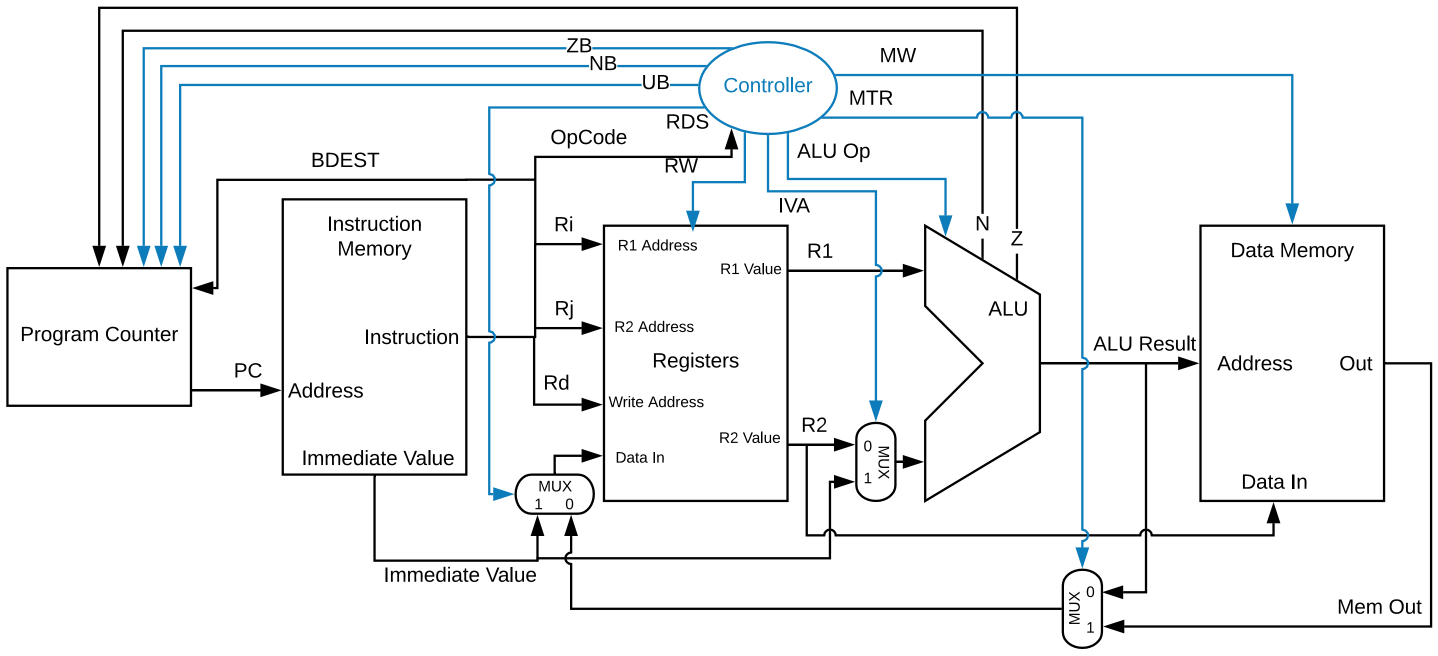
| | | | | |
|---|---|---|---|---|
| Store | 10010 | STORE Rd Ri | Stores a value located in register Ri into memory at address stored in Rd | None |
| Move | 10011 | MOVE Rd Ri | Moves the value in register Ri into register Rd | None |
| Add* | 00000 | ADD Rd Ri Rj | Adds the values in registers Ri and Rj together and stores the result in Rd | Z N |
| Subtract* | 00001 | SUB Rd Ri Rj | Subtracts the values in register Ri from the one in Rj and stores the result in Rd | Z N |
| Negate* | 00010 | NEG Rd Ri | Multiplies the value in Ri by -1 and stores the result in Rd | Z N |
| Multiply | 00011 | MUL Rd Ri Rj | Multiplies the values in Ri and Rj together and stores the result in Rd | Z N |
| Divide | 00100 | DIV Rd Ri Rj | Divides the value in Ri by the one in Rj and stores the result in Rd | Z N E |
| Floor* | 00101 | FLOOR Rd Ri | Rounds the value in Ri down to the nearest integer and stores the result in Rd | Z N |
| Ceiling* | 00110 | CEIL Rd Ri | Rounds the value in Ri up to the nearest integer and stores the result in Rd | Z N |
| Round* | 00111 | ROUND Rd Ri | Rounds the value in Ri to the nearest even | Z N |

| | | | | |
|---|---|---|---|---|
| | | | integer and stores the result in Rd | |
| Absolute* | 01000 | ABS Rd Ri | Takes the absolute value of the value in Ri and stores the result in Rd | Z N |
| Minimum* | 01001 | MIN Rd Ri Rj | Stores the smaller value in Ri and Rj into Rd | Z N |
| Maximum* | 01010 | MAX Rd Ri Rj | Stores the larger of the values in Ri and Rj into Rd | Z N |
| Power | 01011 | POW Rd Ri #Value | Raises the value in register Ri to the immediate value and stores the result in Rd | Z N |
| Exponent | 01100 | EXP Rd Ri | Calculates the value of e raised to the power in Ri and stores the result in Rd | Z N |
| Square Root | 01101 | SQRT Rd Ri | Takes the square root of the value in Ri and stores the result in Rd | Z N |
| Unconditional Branch | 11010 | UB LABEL | Moves the PC to the line marked with the specified label | None |
| Negative Branch | 11000 | BN Ri LABEL | Moves the PC to the line marked with the specified label if the value in Ri is negative | N |
| Zero Branch | 11001 | BZ Ri LABEL | Moves the PC to the line marked with the specified label if the value in Ri is zero | Z |
| No-op | 11111 | PASS | Does nothing for one cycle | None |

| | | | | |
|---|---|---|---|---|
| Halt | 10101 | HALT | Stops execution of the program | None |

* Implemented at the bit level

## Single Cycle



Data Path

Controller Signals:

- UB – Unconditional Branch {1: Branch unconditionally, 0: Don’t do that}
- ZB – Branch on Zero {1: Branch if zero flag from ALU is set, 0: Don’t do that}
- NB – Branch on Negative {1: Branch if negative flag from ALU is set, 0: Don’t do that}
- ALUC – ALU Control Signal {4 bit signal for ALU ops}
- RW – Register Write {1: Write data to Rw address, 0: Don’t do that}
- MW – Memory Write {1: Write data to memory address, 0: Read data from memory address}
- MTR – Memory to Register {0: Send ALU result to register, 1: Send memory result to register}
- RDS – Register Data Source {1: Immediate Value, 0: Memory/ALU}

- IVA – Immediate Value to ALU {1: Use immediate value, 0: use register output}

## Controller Signals

| Op Code | ALUC | UB | ZB | NB | RW | MW | MTR | RDS | IVA |
|---|---|---|---|---|---|---|---|---|---|
| SET | 1111* | 0 | 0 | 0 | 1 | 0 | X | 1 | 0 |
| LOAD | 1111* | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| STORE | 1111* | 0 | 0 | 0 | 0 | 1 | X | 0 | 0 |
| MOVE | 1111* | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| ADD | 0000 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| SUB | 0001 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| NEG | 0010 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| MUL | 0011 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| DIV | 0100 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| FLOOR | 0101 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| CEIL | 0110 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| ROUND | 0111 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| ABS | 1000 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| MIN | 1001 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| MAX | 1010 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| POW | 1011 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| EXP | 1100 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| SQRT | 1101 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| UB | 1111* | 1 | 0 | 0 | 0 | 0 | X | 0 | 0 |
| ZB | 1111* | 0 | 1 | 0 | 0 | 0 | X | 0 | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| NB | 1111* | 0 | 0 | 1 | 0 | 0 | X | 0 | 0 |
| NOP | 1111* | 0 | 0 | 0 | 0 | 0 | X | 0 | 0 |
| HALT | 1111* | 1 | 0 | 0 | 0 | 0 | X | 0 | 0 |

* 1111 is PASS op to ALU. It will pass whatever is on its first input through to its output. Doubles as no-op/don't care command as output value can be ignored when not needed

Notes:

- The PC receives the various control signals it needs to compute its next value. Since immediate values are provided by the instruction memory for all instructions, the PC jumps by 2 addresses by default.
- HALT operation is implemented as an unconditional branch to the same address. Once the HALT command is reached, the PC will remain locked to that memory address. (Probably a bad idea)
- MTR in the chart above are set to don't care (X) when overriding signals around the register negate their impact. For example, if the register write signal is off (no data is being written to the register), the value that reaches the register through the muxes doesn't matter.

-

# Memory Elements

**Instruction Memory, Register File, Data Memory & Stage Registers**

## Instruction Memory:

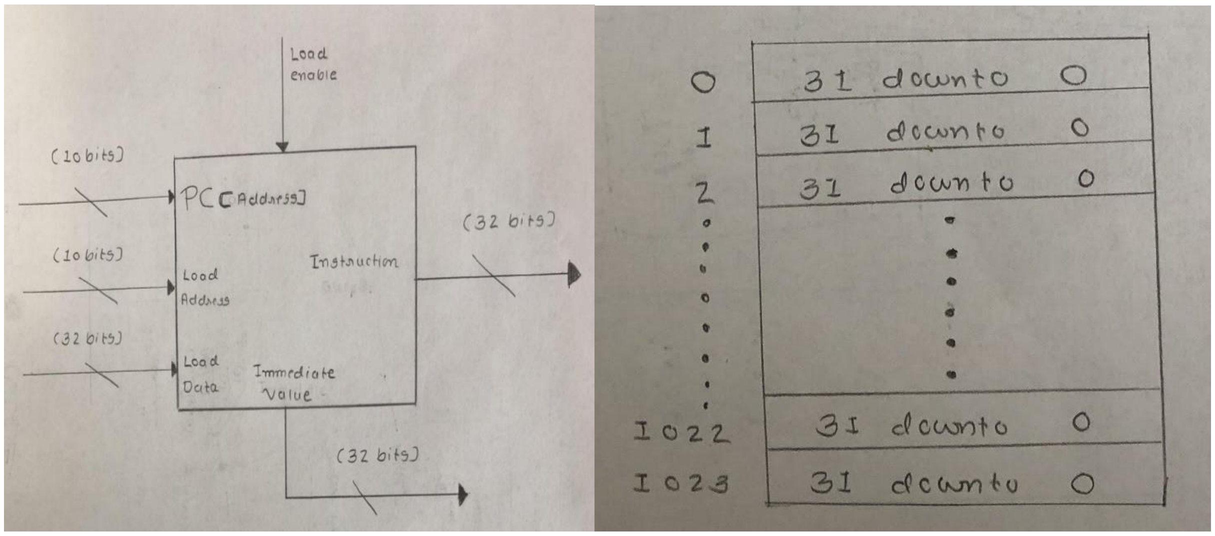


**Figure 1: Black Box Diagram of Instruction Memory**

Above image 1, shows the diagram of the instruction memory. Instruction memory is designed to have a capacity of storing 1024 words maximum at once (4096 bytes). As shown in figure, it has four input ports and two output ports. For the Program Counter and load address ports 10 bits signal ports are used to index the particular word of data from 1024 different word(1 word = 32 bits) values. Load Data port is used to transfer the data to instruction memory when load enable signal is high (once while loading the program). For output, two ports of 32 bits are assigned. Instruction output signal is used to omit out the pointed instruction data by PC. While the Immediate value signal carry outs the immediate value stored on instruction memory.

For the stimulation of the instruction memory, 1024 arrays each of 32 bits were created using the VHDL code. Instruction memory is programmed to omit out one word of instruction data and reading new PC value on every rising edge of clock. For loading the new data to instruction memory, load data port is used to transfer the new data and load address is used for pointing the load location for new data. While load enable port is used to activate the load data and load address ports. Finally, the immediate values according to ISA uses 32 bits (entire block of word) so, the instruction consisting of immediate values omits out the entire block of word data followed by the one indexed by PC (PC+1) indexed value of particular instruction.

**Register File:**

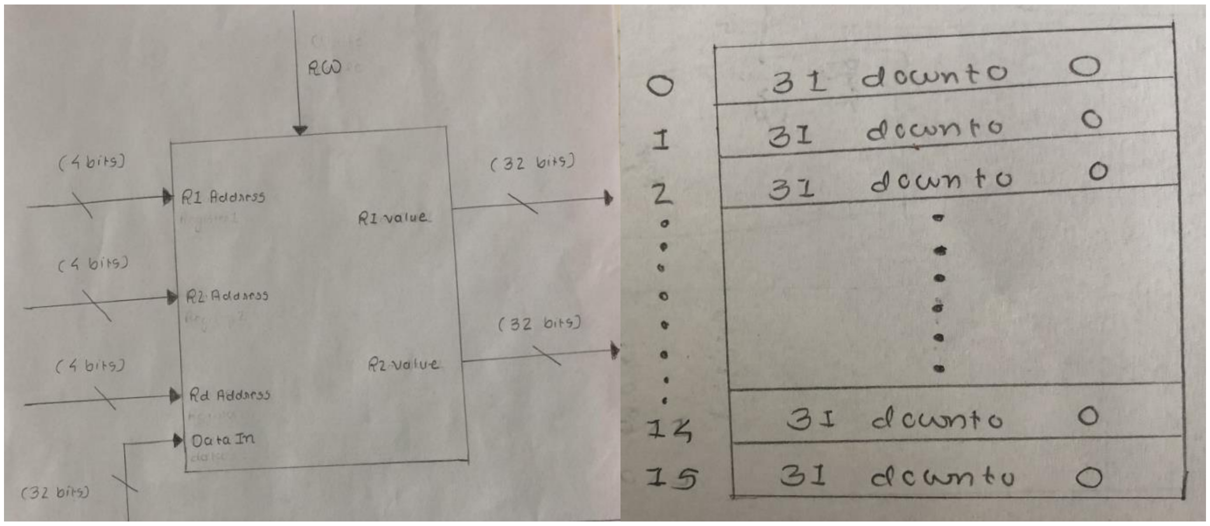


**Figure 2: Black Box Diagram of Register File**

Above image 2, shows the diagram of the register file. Register file is designed for the processor consist of the sixteen different registers each of 32 bits. As demonstrated in diagram, it has five input ports and two output ports. ‘R1 Address’ and ‘R2 Address’ ports each with four-bit bus signal are used to point the two source registers from the register file similarly, ‘Rd Address’ port is used to point the destination register for the particular instruction. ‘Data In’ port with 32-bit bus signal is used for transferring(writing) the data to the destination register, when ‘RW’ signal is high. For output ‘R1 value’ and ‘R2 value’ ports each assigned with 32 bits bus signal for passing the source register data to ALU for execution stage.

For stimulating the register file,16 arrays each of 32 bits were created using the VHDL code. Register memory is programmed to omit out data at output ports on every rising edge of the clock cycle and read new input values pointing the other new two source registers. For writing the data to destination register on the register file, 'RW' signal is activated then, 'Rd Address'' port is used to point the destination register on the file and data is written through the 'Data In' port.

**Data Memory:**

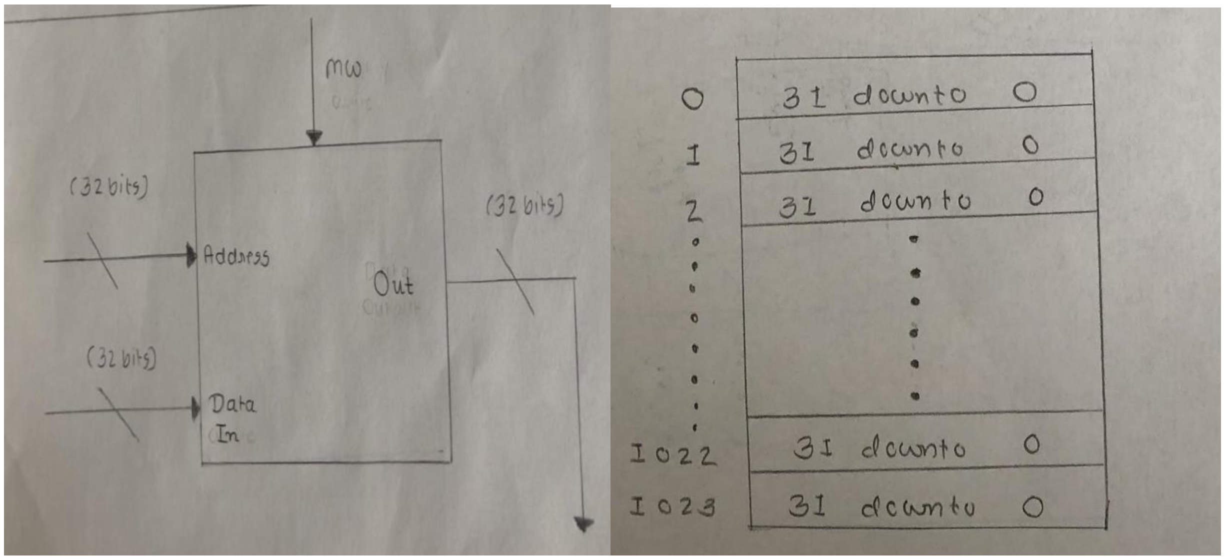


**Figure 3: Black Box Diagram of Data Memory**

Image 3 shows the diagram of the data memory. Data memory is designed to have capacity of storing 1024 words maximum at once (4096 bytes). As shown in the diagram, it has three input ports and one output port. 'Address' port is assigned the signal of 32 bits bus for indexing the particular word in data memory. 'Data In'' port is used to transfer/write the data in the data memory when 'MW' signal is high. For the output 'Out' port is used, it also has 32-bit bus signal assigned to it. Output data then is written to destination register.

For stimulation of the data memory, 1024 arrays each of 32 bits were created using the VHDL coding. Data memory is programmed to omit out one word (1 word = 32 bits) of data and read new ‘Address’ on every rising edge of clock cycle. For storing the new data in data memory ‘Data In’ port is used for writing the new data and ‘Address’ port is used to point location in memory. ‘MW’ signal is used to activate the both of the other input ports. At last, data omitted by ‘Out’ port is send to the ‘Data In’ port of the register file. The only issue faced while stimulating the data memory was that the ‘Address’ value calculated by the ALU was in format of floating-point numbers, so it became hurdle for indexing the location in memory. To fix that problem, function that converts the floating-point number to integer format was utilized.

**Stage Registers:**

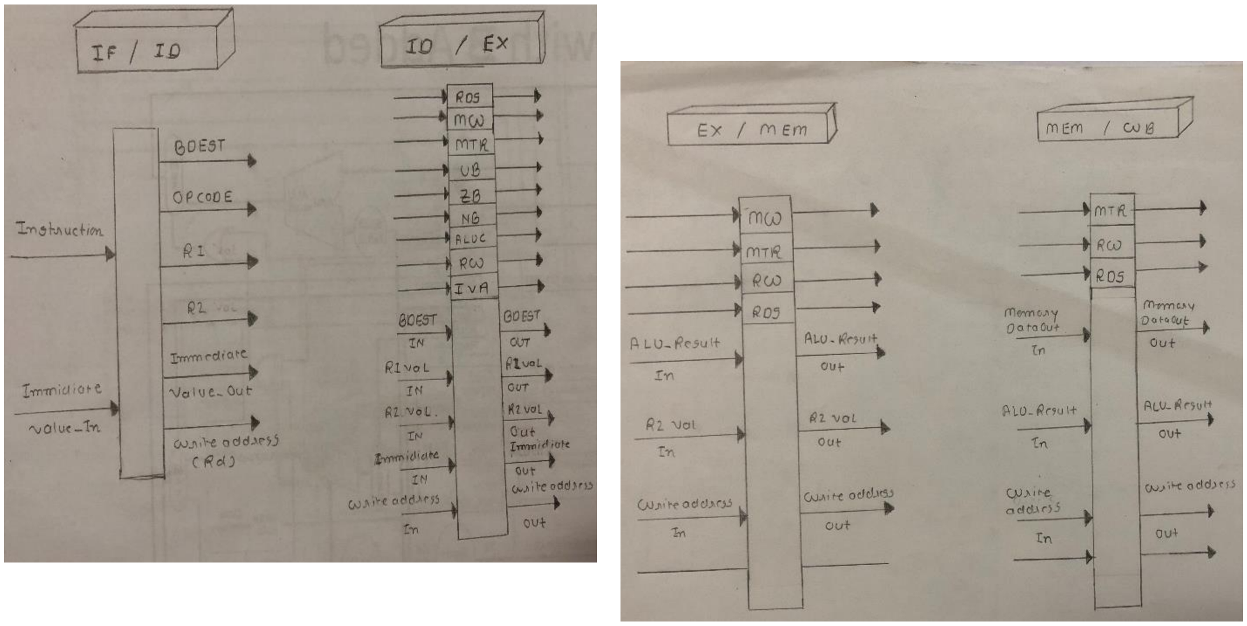


**Figure 4: Black Box Diagram of four Stage Registers**

As shown in figure 4, four stage registers are used while applying the pipeline on the processor. Each stage in processor pass its output signals along with the control signals to this stage registers and then stage registers pass those signals to next stage as per needed (by detecting hazards). For stimulation of these registers VHDL code was used. Basically, registers were designed to take the input in form of signal on rising edge of clock of clock, and omitted the signals for the rest. Below is the description of each stage register in brief:

IF/ID: This stage register receives the inputs from fetch stage components like instruction memory and passes output to decode stage component such as register file.

ID/EX: This stage register receives the inputs from decode stage components like register file and passes output to execution stage component such as ALU.

EX/MEM: This stage register receives the inputs from execution stage components like ALU and passes output to memory stage component such as data memory.

MEM/WB: This stage register receives the inputs from memory stage components like data memory and passes output back to destination register in register file.

## PC

The program counter keeps track of the program's execution. It receives various control signals from the controller and ALU to determine branches. Since our instruction memory outputs both the instruction and any potential immediate values, the default increment for the value in the PC each is cycle is two. On the single cycle design this is not necessary, but it avoids a lot of potential control hazards in the pipelined version.

## ALU

Overall the ALU acts as a large multiplexer that takes two 32 bit inputs and performs different arithmetic operations on them based on the control signal it receives. After the result if computed, it also send out a zero and negative flags for the PC to use to determine whether or not to branch.

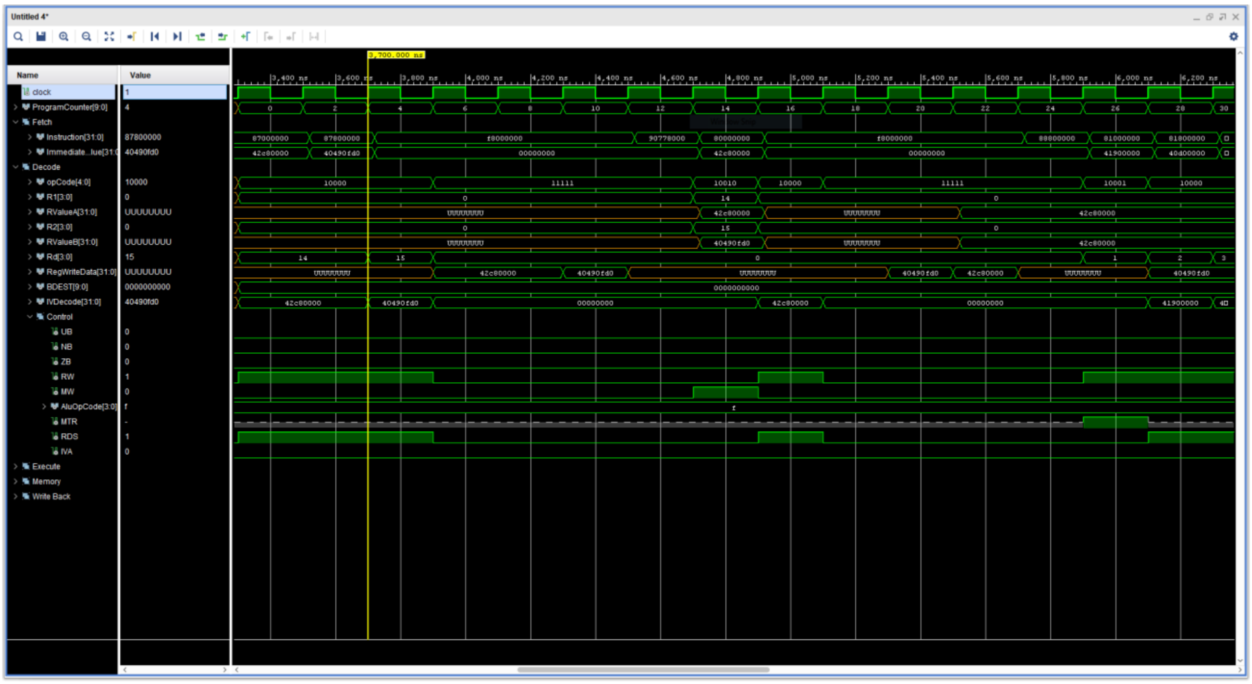
The add function was easily one of the most harrowing tasks that our group encountered.  One of the biggest issues with designing the addition function was handling the special cases of infinity and denormalized values. It also required a fair number of auxiliary functions such as shifting, two's complimenting, and regular unsigned addition/subtraction to be designed to complete the task.

Jordan Cottle, Michael Russell, Harshil Suthar

# Pipeline

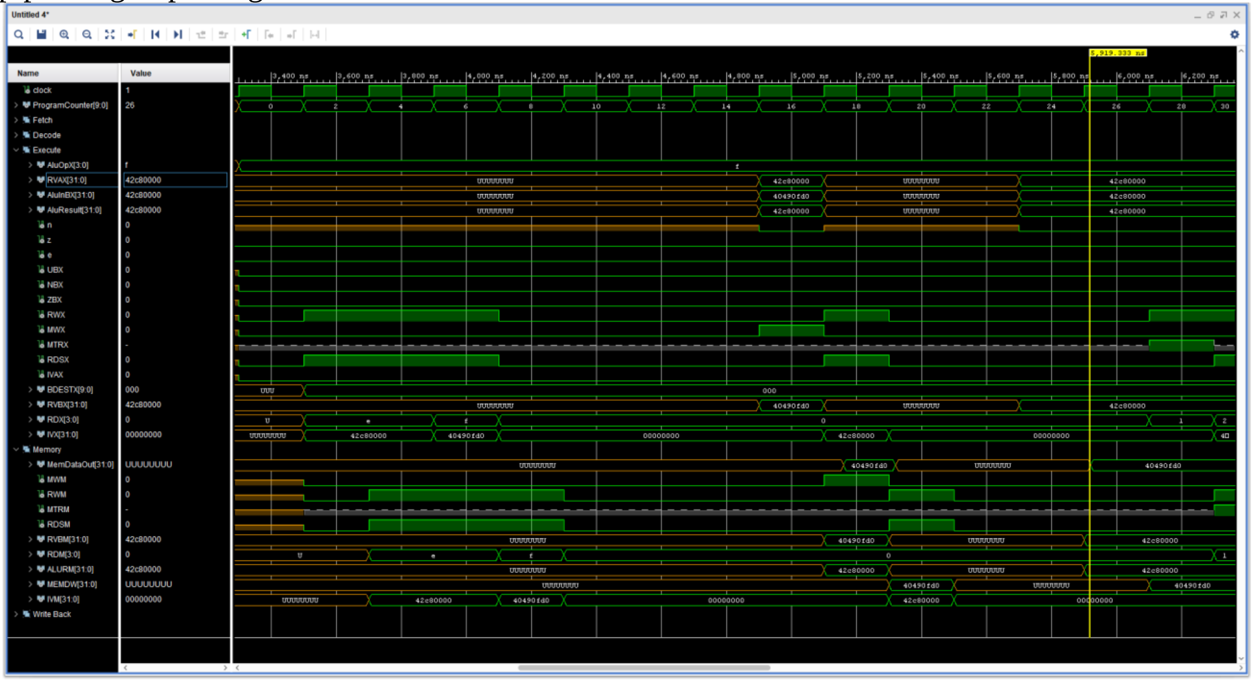
***Waveforms:***

Fetch/Decode: Shows the various signals from the fetch and decode stages of the pipeline grouped together



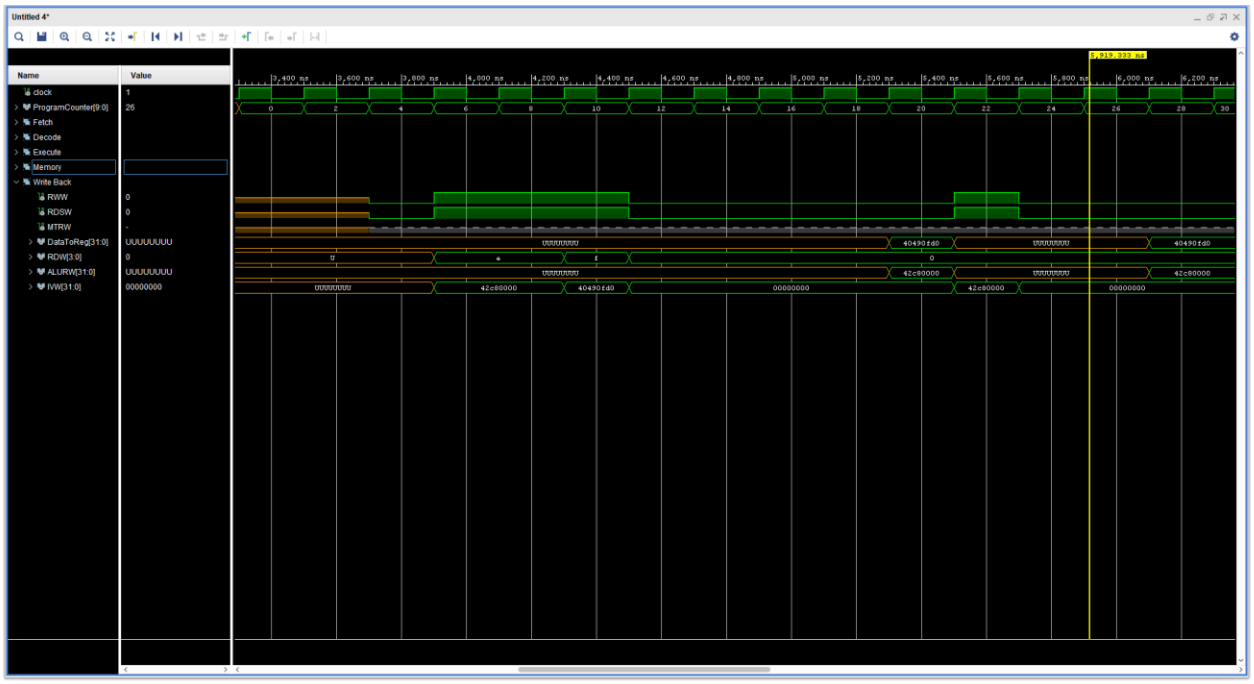
| Name | Value |
|---|---|
| clock | 1 |
| ProgramCounter[9:0] | 4 |
| Fetch | |
| Instruction[31:0] | 87800000 |
| Immediate...lue[31:0] | 40490fd0 |
| Decode | |
| opCode[4:0] | 10000 |
| R1[3:0] | 0 |
| RValueA[31:0] | UUUUUUUU |
| R2[3:0] | 0 |
| RValueB[31:0] | UUUUUUUU |
| Rd[3:0] | 15 |
| RegWriteData[31:0] | UUUUUUUU |
| BDEST[9:0] | 0000000000 |
| IVDecode[31:0] | 40490fd0 |
| Control | |
| UB | 0 |
| NB | 0 |
| ZB | 0 |
| RW | 1 |
| MW | 0 |
| AluOpCode[3:0] | f |
| MTR | - |
| RDS | 1 |
| IVA | 0 |
| Execute | |
| Memory | |
| Write Back | |

Execute/Memory: Shows the various signals from the execute and memory stages of the pipeline grouped together



```
Untitled 4*                              5,919.333 ns
Name                      Value
clock                     1
> ProgramCounter[9:0]     26
> Fetch
> Decode
v Execute
  > AluOpX[3:0]           f
  > RVAX[31:0]            42c80000
  > AluInBX[31:0]         42c80000
  > AluResult[31:0]       42c80000
    n                     0
    z                     0
    e                     0
    UBX                   0
    NBX                   0
    ZBX                   0
    RWX                   0
    MWX                   0
    MTRX                  -
    RDSX                  0
    IVAX                  0
  > BDESTX[9:0]           000
  > RVBX[31:0]            42c80000
  > RDX[3:0]              0
  > IVX[31:0]             00000000
v Memory
  > MemDataOut[31:0]      UUUUUUUU
    MWM                   0
    RWM                   0
    MTRM                  -
    RDSM                  0
  > RVBM[31:0]            42c80000
  > RDM[3:0]              0
  > ALURM[31:0]           42c80000
  > MEMDW[31:0]           UUUUUUUU
  > IVM[31:0]             00000000
> Write Back
```

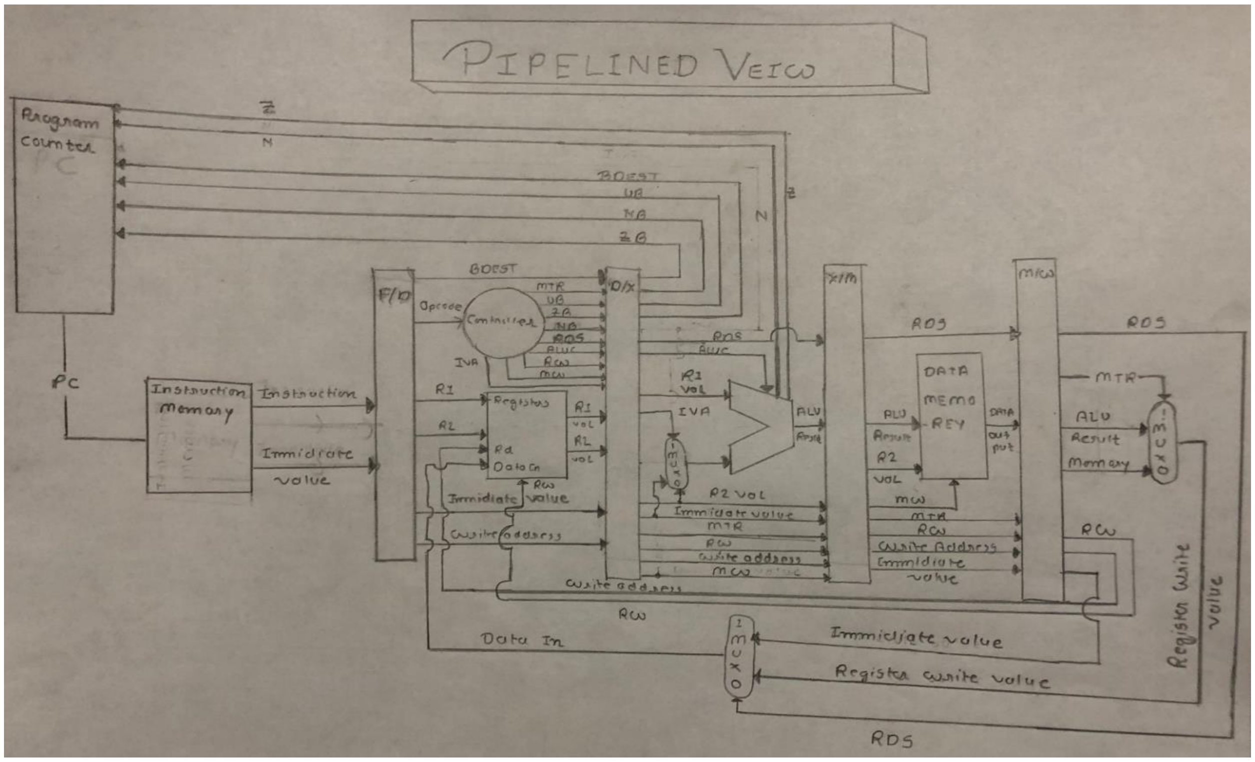
Writeback: Shows the various signals from the write back stage of the pipeline grouped together



```
Untitled 4*                              5,919.333 ns
Name                      Value
clock                     1
> ProgramCounter[9:0]     26
> Fetch
> Decode
> Execute
> Memory
v Write Back
    RWW                   0
    RDSW                  0
    MTRW                  -
  > DataToReg[31:0]       UUUUUUUU
  > RDW[3:0]              0
  > ALURW[31:0]           UUUUUUUU
  > IVW[31:0]             00000000
```

Jordan Cottle, Michael Russell, Harshil Suthar

# Pipelined Architecture



Note: The pipelined design was completed after the demo and now functions properly. It does not have hazard detection or forwarding so dependent instructions must have noops inserted between them to give the processor time to update the values required in the registers before they are used. (There wasn't enough time to write more about the pipeline).