Project Report

Processor Architecture

Nikhil Agrawal -- 2020102021

Siddhant garg -- 2020112006

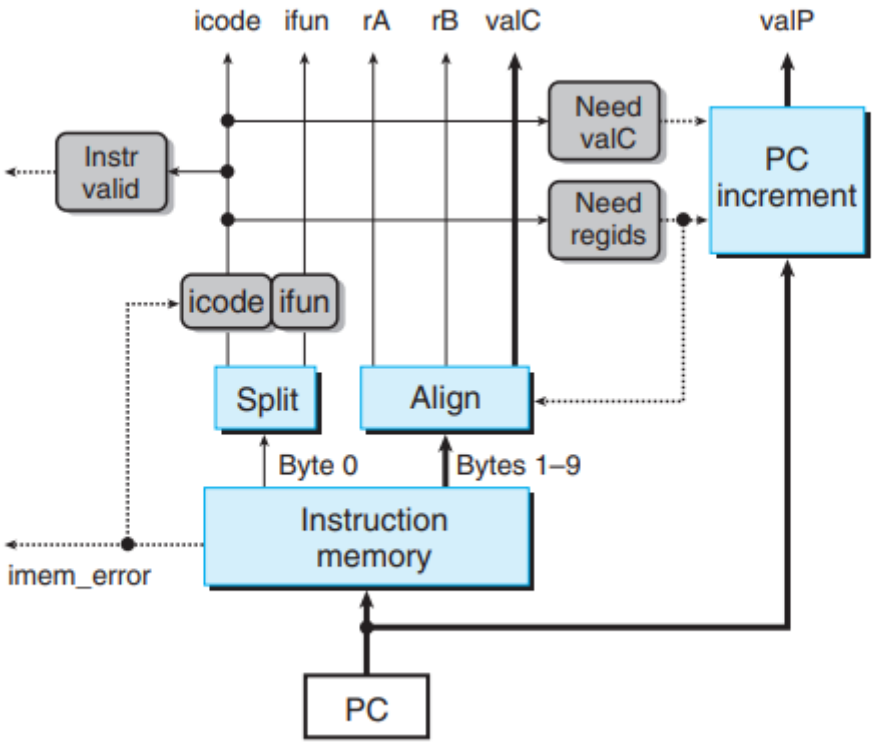
Module Descriptions and Architecture Diagram

- **Stages**

  Note about implementation of memory modules :

  Both the instruction and data memory for this processor are constructed as a collection of constant sized registers. In case of instruction memory, it is a collection of 15 registers of size `80 bits`. In case of data memory, it is a collection of 15 registers which can store `64 bit` numbers. So, the instruction will be of the form `icode-ifun-ra-rb-valC`. If an instruction doesn't involve any field in the structure, it is defaulted to be 0. For example, `JXX` instruction will be encoded as `7-ifun-0-0-valC` and it will contain 10 bytes.

  1. Fetch



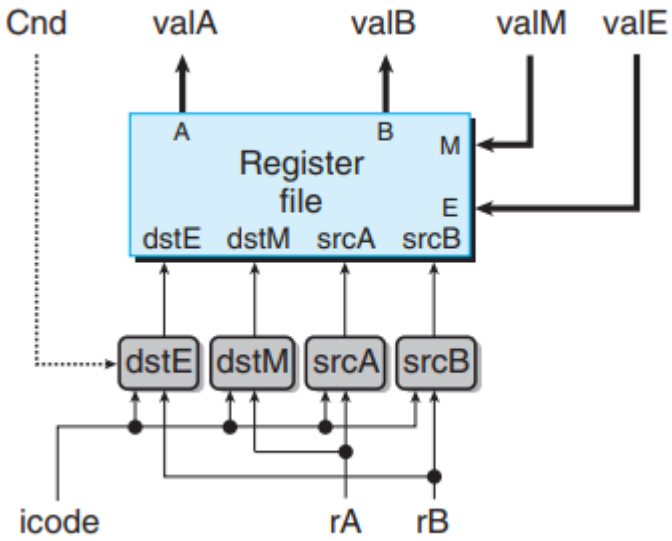
Fetch is the first stage in execution of an instruction. In this stage, the instruction is first fetched from the instruction memory. PC, which stands for program counter is a register which contains the address of current instruction in the instruction memory. So to fetch an instruction from memory, we use PC as the address to reach the instruction. I have designed the instruction memory such that each instruction occupies `10 bytes`. So if there are no jumps, the PC must increment by 10 after every instruction. This value is stored as `valP` after passing through the `pc_inc` block.

The first byte in the instruction contains the information about the type of instruction and the functionality. To get them, the first byte is passed through the split block which return `icode` and `ifun` which are the code for the instruction and the functionality respectively.

The remaining `9 bytes` contain the information about the registers involved in the instruction and constant vale involved in the instruction. So these 9 bytes are passed through the align block where it outputs the register codes `ra` and `rb` along with the constant value `valC`. If the

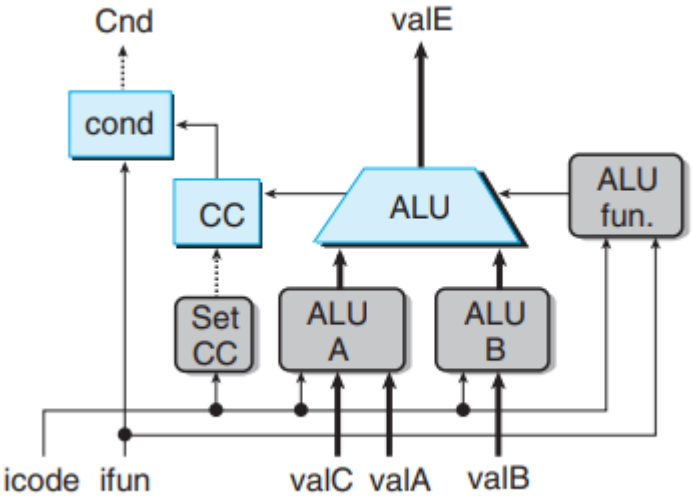
instruction doesn't involve any register or constant, it is defaulted to `zero`.

2. Decode



The decode stage gets its inputs `ra` and `rb`. This stage is to decode the register codes and output the values contained in these registers namely `valA` and `valB`. This block also acts as a write back stage as this is the stage where all the processor registers can be accessed (read / write). A maximum of two values can be written back to the registers. `valM` and `valE` are written back at locations `dstM` and `dstE` respectively. These locations are register locations (register codes).

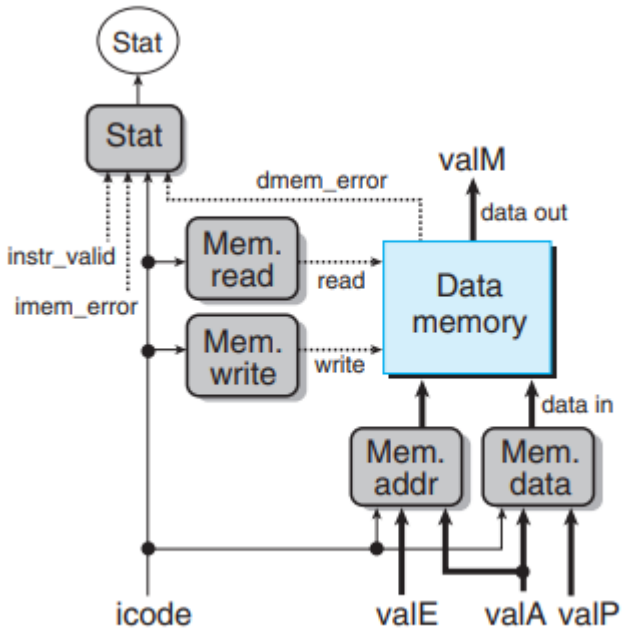
3. Execute



The execute stage contains the arithmetic and logic unit (ALU). The inputs are conditional and depends on the type of instructions. The conditional assignment of inputs are clearly mentioned in the code. The ALU can perform `+, -, & and ^` . The functionality will be mentioned by an input parameter called `fun`. The output of this unit is `valE`. This stage also outputs the new set of condition codes. Condition codes are set in accordance to the output of execute stage. The condition codes contain 3 bits `zf, of and sf` which are set if the output is 0 or there is an overflow or the calculated number is negative respectively.

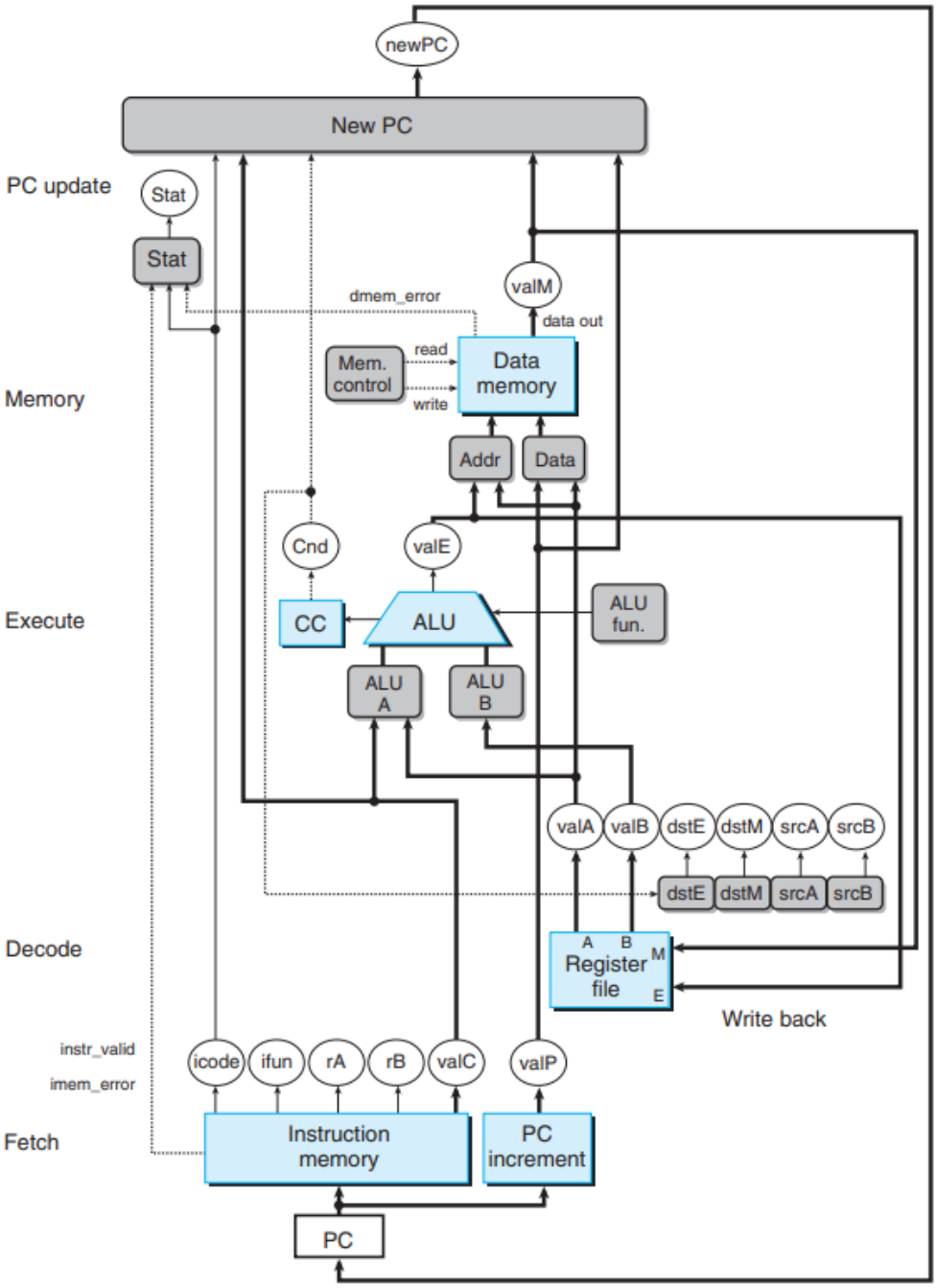
The updated condition codes are set in the condition code register only if the current instruction is a `OPQ` instruction.

4. Memory



The data memory is organised as a collection of `64 bit` registers. Each register has an address. So, this module can read or write values from / to the memory. For read instructions, `valM` is updated such that it contains the read value.

- **Processor**



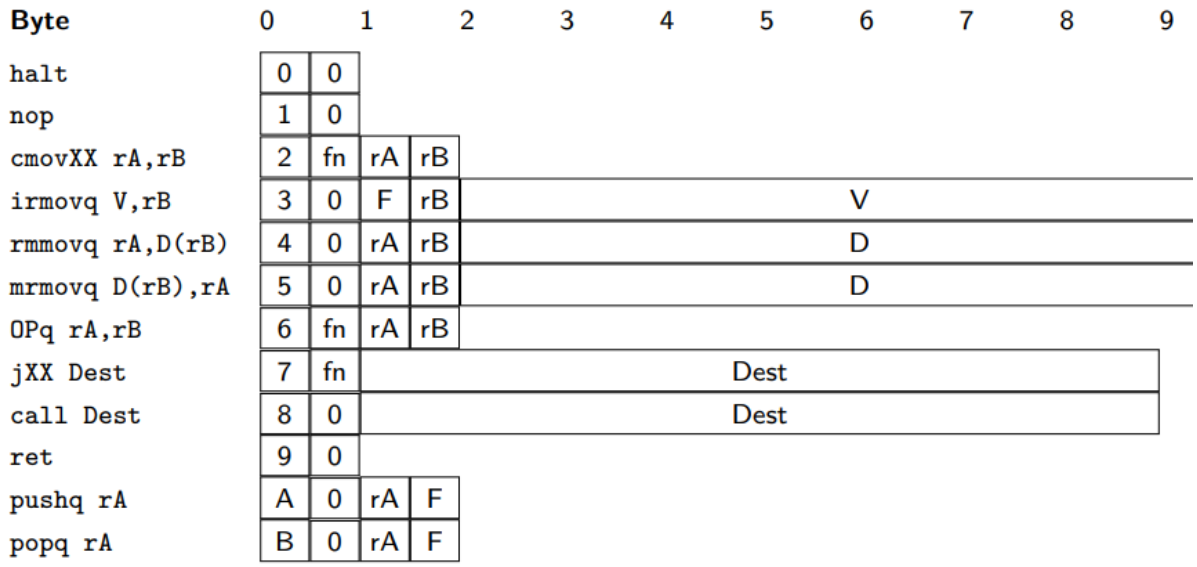
The overall processor can be implemented by combining all the individual modules and conditionally providing the input to each of the stages according to the type of the instruction and the functionality.

The clock period used for the simulation is `1 sec`.

Instructions Supported by the Processor

1. HALT
2. NOP

3. CMOVXX
4. IRMOVQ
5. RMMOVQ
6. MRMOVQ
7. OPQ
8. JXX
9. PUSHQ
10. POPQ



Testing the Processor

Here, we are testing the processor on a code that finds the HCF of two numbers.

```cpp
#include<bits/stdc++.h>
using namespace std;

int find_gcd(int a, int b)
{
    if(a == 0)
        return b;
    if(b == 0)
        return a;

    if (a == b)
        return a;

    if(a > b)
        return find_gcd(a-b, b);
    return find_gcd(a, b-a);
}

int main(void)
{
    int a = 36;
    int b = 24;
    cout<<find_gcd(a, b)<<endl;
    return 0;
}
```

C++ code to find the HCF of two numbers recursively.

```
//considering %rbx and %rsp as the inputs
//finding gcd of 24 and 36

//assemlbly code
0   irmovq $36 %rbx   //loading the value to %rbx
10  irmovq $24 %rsp   //loading the value to %rsp
20  rrmovq %rbx %rbp    //moving %rbx and %rsp to temp variables in the program
30  rrmovq %rsp %rsi
40  subq %rsp %rbp
50  cmovg %rbp %rsp     //update %rsp as %rsp - %rbx if r4 > %rbx
60  subq %rbx %rsi
70  cmovg %rsi %rbx     //else update %rbx as %rbx - %rsp
80  jmpne 20        //continue in the loop untill it is 0
90  rrmovq %rbx %r8 //storing the output in %r8
100 halt            //stopping the instructions
```

assembly code for the function above

```
//instructions --encoded
0   80'h30f30000000000018
10  80'h30f40000000000024
20  80'h20350000000000000
30  80'h20460000000000000
40  80'h61450000000000000
50  80'h26540000000000000
60  80'h61360000000000000
70  80'h26630000000000000
80  80'h74000000000000014
90  80'h20380000000000000
100 80'h0
```

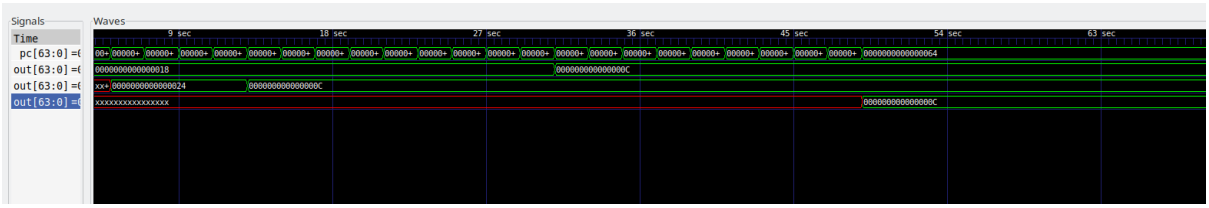
encoding the assembly code into instructions which are added to instruction memory.

The final output will be stored in register 8, $\%r8$

GTK-Wave Output

- TEST - ONE

$A = 36$ and $B = 24$.



The first row is the program counter. We can see how it changes in every clock cycle.

The second row is input A ($\%rbx$)

The third row is the second input B ($\%rsp$)

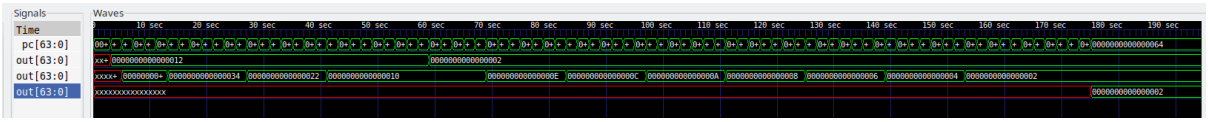
The fourth row is the final output ($\%r8$). It is updated only when the complete function is complete and we arrive at an answer for HCF of A and B.

Since the final instruction is a HALT, the register carry the same value for eternity once the program has completed.

We can see the output value at $\%r8$ being $64h'0000000000000000c$, which is 12. We know HCF of 24 and 36 is indeed 12.

- TEST - TWO

$A = 18$ and $B = 70$.



The first row is the program counter. We can see how it changes in every clock cycle.

The second row is input A ($\%rbx$)

The third row is the second input B ($\%rsp$)

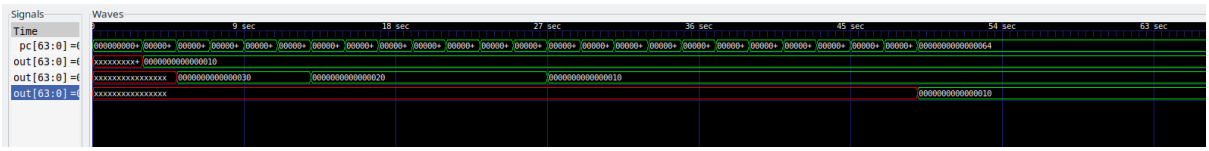
The fourth row is the final output ($\%r8$). It is updated only when the complete function is complete and we arrive at an answer for HCF of A and B.

Since the final instruction is a HALT, the register carry the same value for eternity once the program has completed.

We can see the output value at $\%r8$ being $64h'0000000000000002$, which is 2. We know HCF of 18 and 70 is indeed 2.

- TEST - THREE

$A = 48$ and $B = 16$.



The first row is the program counter. We can see how it changes in every clock cycle.

The second row is input A ($\%rbx$)

The third row is the second input B ($\%rsp$)

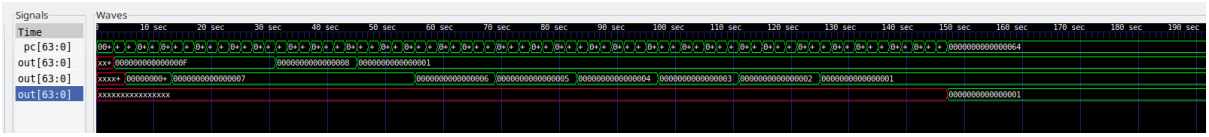
The fourth row is the final output ($\%r8$). It is updated only when the complete function is complete and we arrive at an answer for HCF of A and B.

Since the final instruction is a HALT, the register carry the same value for eternity once the program has completed.

We can see the output value at $\%r8$ being $64h'0000000000000010$, which is 16. We know HCF of 16 and 48 is indeed 16.

- TEST - FOUR

$A = 22$ and $B = 15$.



The first row is the program counter. We can see how it changes in every clock cycle.

The second row is input A ($\%rbx$)

The third row is the second input B ($\%rsp$)

The fourth row is the final output ($\%r8$). It is updated only when the complete function is complete and we arrive at an answer for HCF of A and B.

Since the final instruction is a HALT, the register carry the same value for eternity once the program has completed.

We can see the output value at $\%r8$ being $64h'0000000000000001$, which is 1. We know HCF of 15 and 22 is indeed 1.

Limitations of the design

- The memory size for instructions is `150 bytes`. It is a collection 15 `10-byte` registers. Similarly, for data memory, it is a collection of 15 `8-byte` registers.

Hence it is `120 bytes`. Also every instruction consumes `10 bytes`. Therefore some space in memory is wasted.

- Also, the instructions have to be hard coded into these registers in the memory prior to running the instructions. Instructions can be read and executed, but not written to memory. So the sequence of instructions to be executed have to be initialised in the instruction memory first before running the processor. But this is not the case for data memory. Values can be written and read from data memory.
- This is not a pipelined processor. Hence time taken to complete the set of instructions is generally more.

Instructions to run the code

```
iverilog -o comp.vvp seq_test.v seq.v split.v align.v  decode.v  alu.v  condn.v clocke
d_reg.v  instr_mem.v pc_inc.v data_mem.v && vvp comp.vvp
```

Running this will run the entire processor module along with its test bench. However, if individual modules have to be tested, it can be run with the respective test-bench and dependent files.

Since the instructions have to be loaded before running the processor, the sequence of instructions that have to be executed must be hard coded to the instruction memory registers with proper structure.

- Expected output

  The test bench replies all the register values (15 registers which are in the processor) after a delay of 300 seconds. It has been set such all the tested instructions should get over by this so that we can get the proper register status.