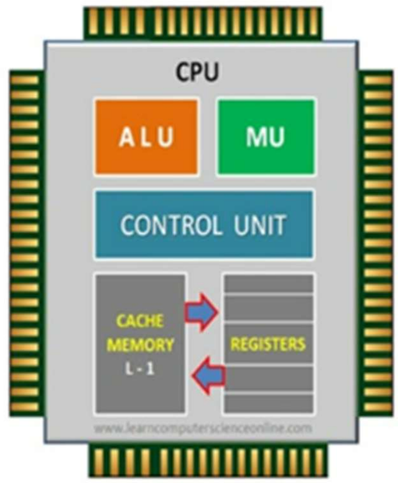
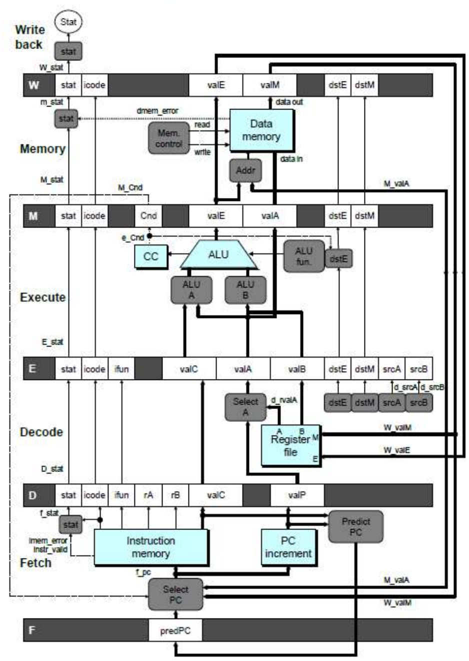




Project Report

ALU + SEQ + PIPE

Pronoy (2021112019) and Ashish (2021102016)
Information and Processor Architecture, Spring 2023
08th March 2023

Introduction

In this project, we implemented a processor architecture design based on the Y86 ISA using Verilog. Our design is a full-fledged processor architecture implementation which includes various stages of the processor architecture, viz., sequential, 5 stage pipeline, and is successfully able to execute all the instructions from Y86 ISA.

We have used modular approach for our design i.e., each stage is coded as separate modules and tested independently to help the integration without too many issues.

All the supported features were individually and thoroughly tested to satisfy all the specification requirements and the corresponding simulation snapshots are attached in the report.

An assembly program for sorting algorithm using Y86 ISA was written and the corresponding encoded instructions was used to test the integrated design.

The challenges encountered in the implement of this processor has also been discussed.

ALU Implementation

We built an ALU unit with following functionality:

- ADD – 64 bits
- SUB – 64 bits
- AND – 64 bits
- XOR – 64 bits

We didn't use +, -, &, ^ directly on the 64-bit inputs for 64-bit operations. We wrote each of the above modules from scratch (structural).

All input and output were signed.

Wrapper ALU unit

We created a ALU unit to call the other modules mentioned above based on the control input. The ALU unit takes as input the control signal, and two 64-bit inputs, and returns the 64-bit output corresponding to the control signal chosen. An example with 64-bit inputs x and y:

Control 0 - ADD x and y

Control 1 – Subtract y from x

Control 2 – AND x and y

Control 3 – XOR x and y

To verify the functioning of the module created by us, we have created a testbench which takes two random inputs and calculate values directly and using the designed module to verify if both give the same results or not.

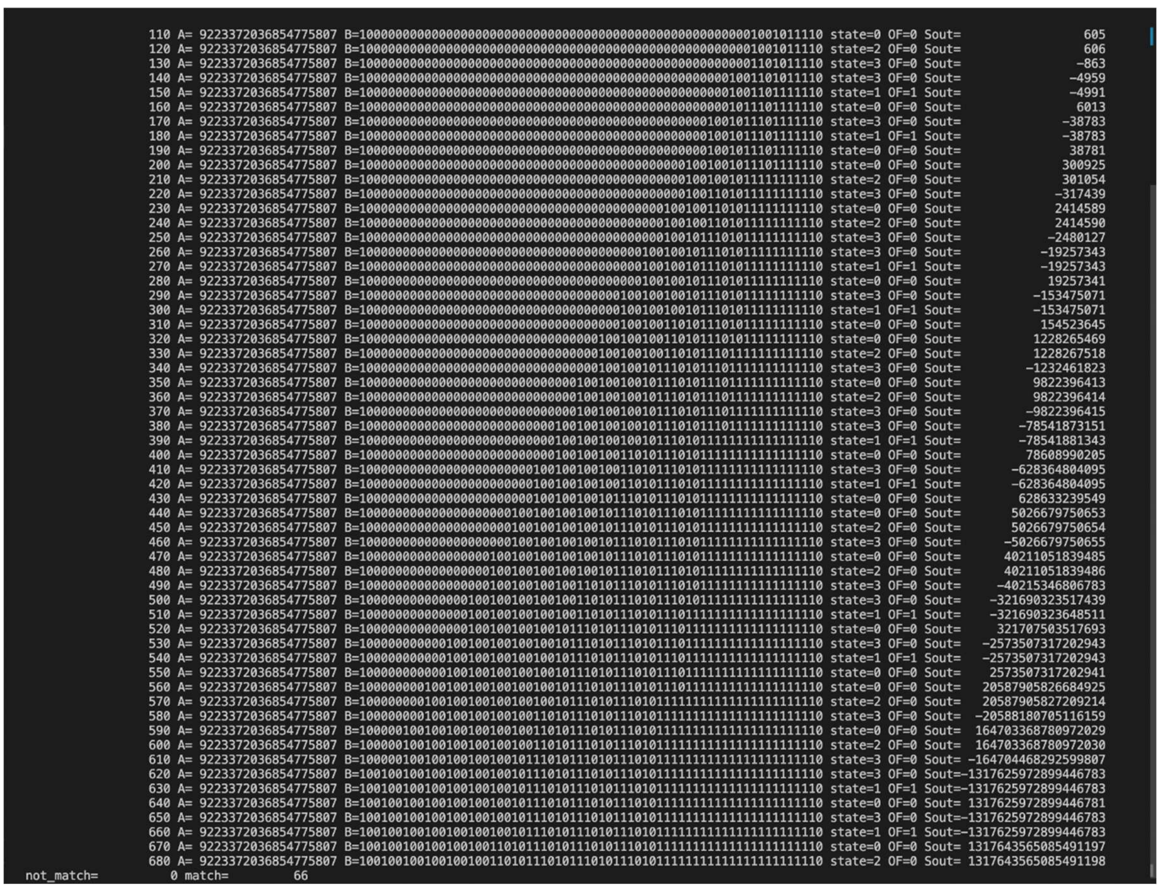


Fig 1.1 Output for different Inputs.

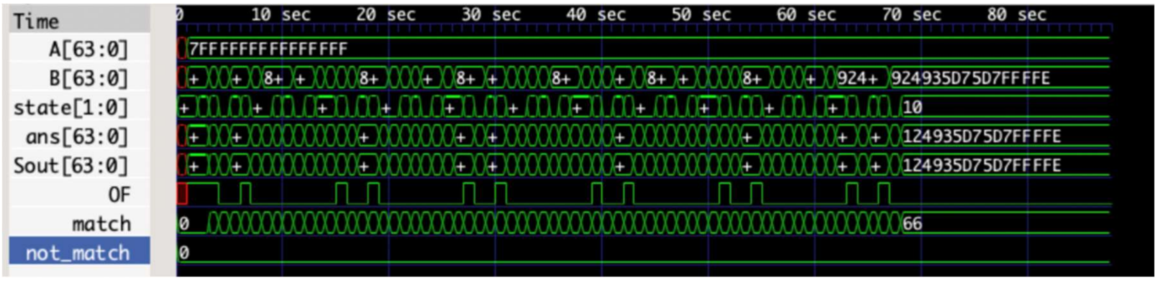


Fig 1.2 Output waveform of the results.

ADD

To implement this module in structural method, we first created a full adder module to add 3 bits. Then, we called this module 64 time to find the addition of two 64 bits binary number.

So, basically we used the Ripple Carry Adder (RCA) method to add the two numbers.

The overflow can be found by XORing the 64^{th} and 63^{rd} Carry bit.

To verify the working of our module, we have written a test bench where we have taken random values and calculated values directly and using the module made by us to verify if both are giving the same results not.

```
                    10 A= 9223372036854775807 B=-9223372036854775808 OF=0 Sout=                   -1
                    30 A= 9223372036854775807 B=-9223372036854775806 OF=0 Sout=                    1
                    40 A= 9223372036854775807 B=-9223372036854775802 OF=0 Sout=                    5
                    50 A= 9223372036854775807 B=-9223372036854775794 OF=0 Sout=                   13
                    70 A= 9223372036854775807 B=-9223372036854775778 OF=0 Sout=                   29
                    80 A= 9223372036854775807 B=-9223372036854775714 OF=0 Sout=                   93
                   110 A= 9223372036854775807 B=-9223372036854775202 OF=0 Sout=                  605
                   130 A= 9223372036854775807 B=-9223372036854774946 OF=0 Sout=                  861
                   140 A= 9223372036854775807 B=-9223372036854770850 OF=0 Sout=                 4957
                   150 A= 9223372036854775807 B=-9223372036854770818 OF=0 Sout=                 4989
                   160 A= 9223372036854775807 B=-9223372036854769794 OF=0 Sout=                 6013
                   170 A= 9223372036854775807 B=-9223372036854737026 OF=0 Sout=                38781
                   200 A= 9223372036854775807 B=-9223372036854474882 OF=0 Sout=               300925
                   210 A= 9223372036854775807 B=-9223372036854474754 OF=0 Sout=               301053
                   220 A= 9223372036854775807 B=-9223372036854458370 OF=0 Sout=               317437
                   230 A= 9223372036854775807 B=-9223372036852361218 OF=0 Sout=              2414589
                   250 A= 9223372036854775807 B=-9223372036852295682 OF=0 Sout=              2480125
                   260 A= 9223372036854775807 B=-9223372036835518466 OF=0 Sout=             19257341
                   290 A= 9223372036854775807 B=-9223372036701300738 OF=0 Sout=            153475069
                   310 A= 9223372036854775807 B=-9223372036700252162 OF=0 Sout=            154523645
                   320 A= 9223372036854775807 B=-9223372035626510338 OF=0 Sout=           1228265469
                   330 A= 9223372036854775807 B=-9223372035626508290 OF=0 Sout=           1228267517
                   340 A= 9223372036854775807 B=-9223372035622313986 OF=0 Sout=           1232461821
                   350 A= 9223372036854775807 B=-9223372027032379394 OF=0 Sout=           9822396413
                   380 A= 9223372036854775807 B=-9223371958312902658 OF=0 Sout=          78541873149
                   390 A= 9223372036854775807 B=-9223371958312894466 OF=0 Sout=          78541881341
                   400 A= 9223372036854775807 B=-9223371958245785602 OF=0 Sout=          78608990205
                   410 A= 9223372036854775807 B=-9223371408489971714 OF=0 Sout=         628364804093
                   430 A= 9223372036854775807 B=-9223371408221536258 OF=0 Sout=         628633239549
                   440 A= 9223372036854775807 B=-9223367010175025154 OF=0 Sout=        5026679750653
                   470 A= 9223372036854775807 B=-9223331825802936322 OF=0 Sout=       40211051839485
                   490 A= 9223372036854775807 B=-9223331821507969026 OF=0 Sout=       40215346806781
                   500 A= 9223372036854775807 B=-9223050346531258370 OF=0 Sout=      321690323517437
                   510 A= 9223372036854775807 B=-9223050346531127298 OF=0 Sout=      321690323648509
                   520 A= 9223372036854775807 B=-9223050329351258114 OF=0 Sout=      321707503517693
                   530 A= 9223372036854775807 B=-9220798529537572866 OF=0 Sout=     2573507317202941
                   560 A= 9223372036854775807 B=-9202784131028090882 OF=0 Sout=    20587905826684925
                   570 A= 9223372036854775807 B=-9202784131027566594 OF=0 Sout=    20587905827209213
                   580 A= 9223372036854775807 B=-9202783856149659650 OF=0 Sout=    20588180705116157
                   590 A= 9223372036854775807 B=-9058668668073803778 OF=0 Sout=   164703368780972029
                   610 A= 9223372036854775807 B=-9058667568562176002 OF=0 Sout=   164704468292599805
                   620 A= 9223372036854775807 B=-7905746063955329026 OF=0 Sout= 1317625972899446781
                   670 A= 9223372036854775807 B=-7905728471769284610 OF=0 Sout= 1317643565085491197
   not_match=           0 match=          66
```

Fig 2.1 Output for different Inputs.

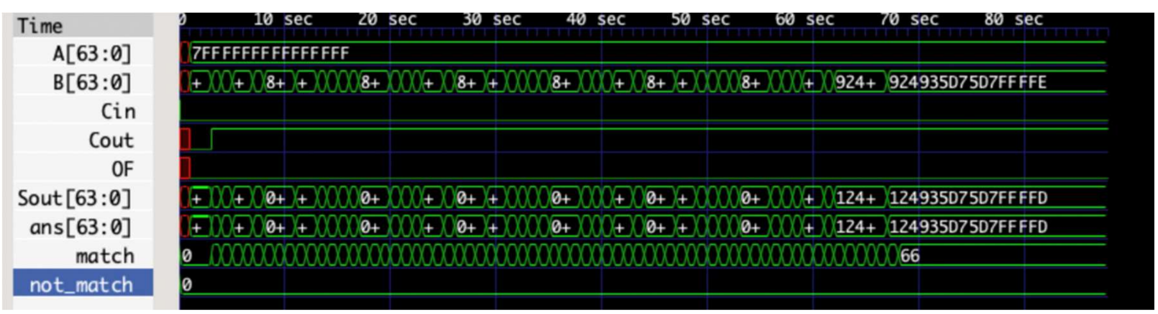


Fig 2.2 Output waveform of the results.

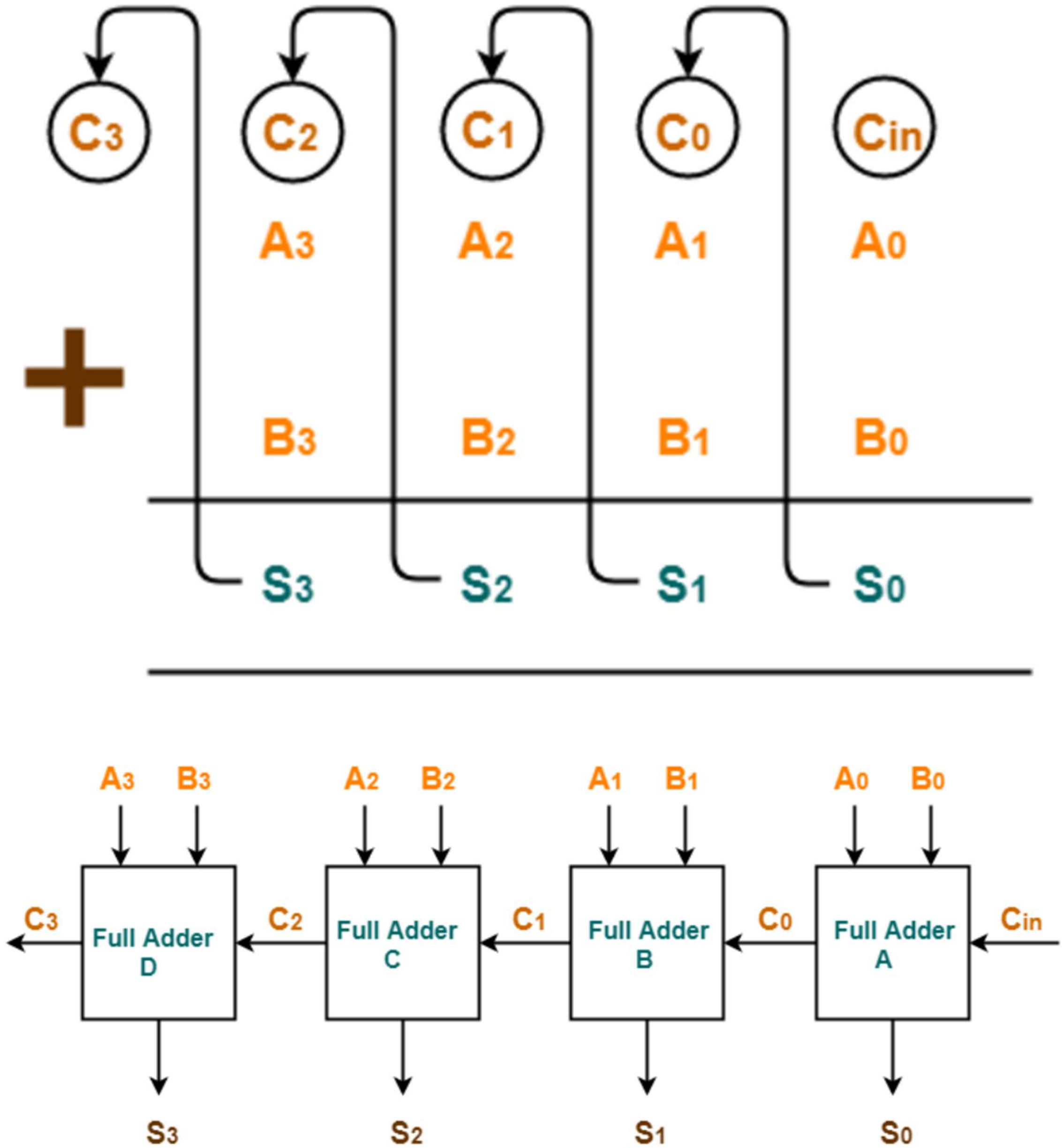


Fig 1.3 Illustration of method used.

SUBTRACT

We used 2's complement for the subtraction. Binary subtraction of two binary numbers can be done by adding the 2's complement of the second number to the first number.

The methodology used is described as below:

- First, we took 1's complement of the second number. This is taken by inverting all the binary digits in the number, i.e., replacing 0's with 1's and 1's with 0's.
- Then, we took 2's complement of that number. That can be found by adding a 1 to the 1's complement of the number.
- The above step was done by calling the ADD module which added the first number, the 1's complement of the second number and 1 as a carry.

To verify the working of our module, we have written a test bench where we have taken random values and calculated values directly and using the module made by us to verify if both are giving the same results not.

```
                  10 A= 9223372036854775807 B=-9223372036854775808 OF=1 Sout=                   -1
                  30 A= 9223372036854775807 B=-9223372036854775806 OF=1 Sout=                   -3
                  40 A= 9223372036854775807 B=-9223372036854775802 OF=1 Sout=                   -7
                  50 A= 9223372036854775807 B=-9223372036854775794 OF=1 Sout=                  -15
                  70 A= 9223372036854775807 B=-9223372036854775778 OF=1 Sout=                  -31
                  80 A= 9223372036854775807 B=-9223372036854775714 OF=1 Sout=                  -95
                 110 A= 9223372036854775807 B=-9223372036854775202 OF=1 Sout=                 -607
                 130 A= 9223372036854775807 B=-9223372036854774946 OF=1 Sout=                 -863
                 140 A= 9223372036854775807 B=-9223372036854770850 OF=1 Sout=                -4959
                 150 A= 9223372036854775807 B=-9223372036854770818 OF=1 Sout=                -4991
                 160 A= 9223372036854775807 B=-9223372036854769794 OF=1 Sout=                -6015
                 170 A= 9223372036854775807 B=-9223372036854737026 OF=1 Sout=               -38783
                 200 A= 9223372036854775807 B=-9223372036854474882 OF=1 Sout=              -300927
                 210 A= 9223372036854775807 B=-9223372036854474754 OF=1 Sout=              -301055
                 220 A= 9223372036854775807 B=-9223372036854458370 OF=1 Sout=              -317439
                 230 A= 9223372036854775807 B=-9223372036852361218 OF=1 Sout=             -2414591
                 250 A= 9223372036854775807 B=-9223372036852295682 OF=1 Sout=             -2480127
                 260 A= 9223372036854775807 B=-9223372036835518466 OF=1 Sout=            -19257343
                 290 A= 9223372036854775807 B=-9223372036701300738 OF=1 Sout=           -153475071
                 310 A= 9223372036854775807 B=-9223372036700252162 OF=1 Sout=           -154523647
                 320 A= 9223372036854775807 B=-9223372035626510338 OF=1 Sout=          -1228265471
                 330 A= 9223372036854775807 B=-9223372035626508290 OF=1 Sout=          -1228267519
                 340 A= 9223372036854775807 B=-9223372035622313986 OF=1 Sout=          -1232461823
                 350 A= 9223372036854775807 B=-9223372027032379394 OF=1 Sout=          -9822396415
                 380 A= 9223372036854775807 B=-9223371958312902658 OF=1 Sout=         -78541873151
                 390 A= 9223372036854775807 B=-9223371958312894466 OF=1 Sout=         -78541881343
                 400 A= 9223372036854775807 B=-9223371958245785602 OF=1 Sout=         -78608990207
                 410 A= 9223372036854775807 B=-9223371408489971714 OF=1 Sout=        -628364804095
                 430 A= 9223372036854775807 B=-9223371408221536258 OF=1 Sout=        -628633239551
                 440 A= 9223372036854775807 B=-9223367010175025154 OF=1 Sout=       -5026679750655
                 470 A= 9223372036854775807 B=-9223331825802936322 OF=1 Sout=      -40211051839487
                 490 A= 9223372036854775807 B=-9223331821507969026 OF=1 Sout=      -40215346806783
                 500 A= 9223372036854775807 B=-9223050346531258370 OF=1 Sout=     -321690323517439
                 510 A= 9223372036854775807 B=-9223050346531127298 OF=1 Sout=     -321690323648511
                 520 A= 9223372036854775807 B=-9223050329351258114 OF=1 Sout=     -321707503517695
                 530 A= 9223372036854775807 B=-9220798529537572866 OF=1 Sout=    -2573507317202943
                 560 A= 9223372036854775807 B=-9202784131028090882 OF=1 Sout=   -20587905826684927
                 570 A= 9223372036854775807 B=-9202784131027566594 OF=1 Sout=   -20587905827209215
                 580 A= 9223372036854775807 B=-9202783856149659650 OF=1 Sout=   -20588180705116159
                 590 A= 9223372036854775807 B=-9058668668073803778 OF=1 Sout= -164703368780972031
                 610 A= 9223372036854775807 B=-9058667568562176002 OF=1 Sout= -164704468292599807
                 620 A= 9223372036854775807 B=-7905746063955329026 OF=1 Sout=-1317625972899446783
                 670 A= 9223372036854775807 B=-7905728471769284610 OF=1 Sout=-1317643565085491199
  not_match=           0 match=          66
```

Fig 3.1 Output for different Inputs.

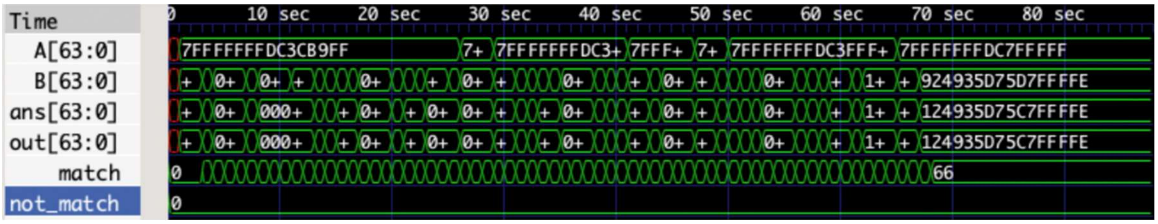


Fig 3.2 Output waveform of the results.

AND

We wrote a module to do AND of two 64 bits signed binary numbers. The module was written in a structural way where each bit was ANDed with the corresponding bit.

```
                 10 A= 9223372036254775807 B=                    8 out=                    8
                 30 A= 9223372036254775807 B=                   10 out=                   10
                 40 A= 9223372036254775807 B=                   14 out=                   14
                 70 A= 9223372036254775807 B=                   30 out=                   30
                 80 A= 9223372036254775807 B=                   94 out=                   94
                110 A= 9223372036254775807 B=                  606 out=                   94
                130 A= 9223372036254775807 B=                  862 out=                  350
                140 A= 9223372036254775807 B=                 4958 out=                 4446
                150 A= 9223372036254775807 B=                 4990 out=                 4478
                160 A= 9223372036254775807 B=                 6014 out=                 4478
                170 A= 9223372036254775807 B=                38782 out=                37246
                200 A= 9223372036254775807 B=               300926 out=               299390
                210 A= 9223372036254775807 B=               301054 out=               299518
                220 A= 9223372036254775807 B=               317438 out=               299518
                230 A= 9223372036254775807 B=              2414590 out=              2396670
                250 A= 9223372036254775807 B=              2480126 out=              2396670
                260 A= 9223372036254776319 B=             19257342 out=              2397182
                290 A= 9223372036254777343 B=            153475070 out=            136615934
                310 A= 9223372036254777343 B=            154523646 out=            137664510
                320 A= 9223372036254777343 B=           1228265470 out=           1211406334
                330 A= 9223372036254777343 B=           1228267518 out=           1211408382
                340 A= 9223372036254777343 B=           1232461822 out=           1211408382
                350 A= 9223372036254777343 B=           9822396414 out=           9801342974
                380 A= 9223372036254777343 B=          78541873150 out=          78520819710
                390 A= 9223372036254777343 B=          78541881342 out=          78520827902
                400 A= 9223372036254777343 B=          78608990206 out=          78587936766
                410 A= 9223372036254793727 B=         628364804094 out=         628343767038
                430 A= 9223372036254793727 B=         628633239550 out=         628612202494
                440 A= 9223372036254793727 B=        5026679750654 out=        5026658713598
                470 A= 9223372036254859263 B=       40211051839486 out=       40211030867966
                490 A= 9223372036254859263 B=       40215346806782 out=       40215325835262
                500 A= 9223372036254990335 B=      321690323517438 out=      321690302545918
                510 A= 9223372036254990335 B=      321690323648510 out=      321690302676990
                520 A= 9223372036254990335 B=      321707503517694 out=      321707482546174
                530 A= 9223372036254990335 B=     2573507317202942 out=     2573507296231422
                560 A= 9223372036254990335 B=    20587905826684926 out=    20587905805713406
                570 A= 9223372036254990335 B=    20587905827209214 out=    20587905806237694
                580 A= 9223372036254990335 B=    20588180705116158 out=    20588180684144638
                590 A= 9223372036254990335 B=   164703368780972030 out=   164703368760000510
                610 A= 9223372036254990335 B=   164704468292599806 out=   164704468271628286
                620 A= 9223372036254990335 B= 1317625972899446782 out= 1317625972878475262
                650 A= 9223372036259184639 B=-7905746063955329026 out= 1317625972882669566
                670 A= 9223372036259184639 B=-7905728471769284610 out= 1317643565068713982
  not_match=          0 match=          66
```

Fig 4.1 Output for different Inputs.

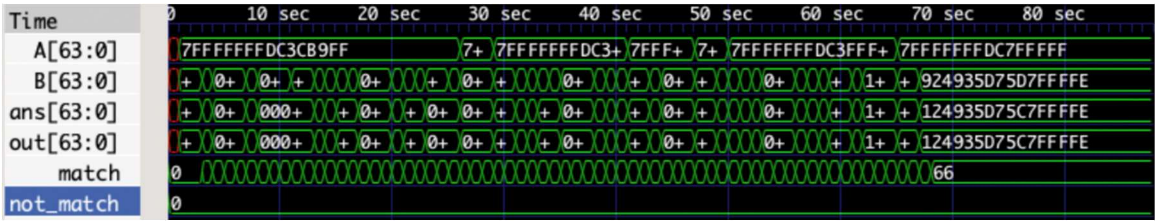


Fig 4.2 Output waveform of the results.

To verify the working of our module, we have written a test bench where we have taken random values and calculated values directly and using the module made by us to verify if both are giving the same results not.

XOR

We wrote a module to do XOR of two 64 bits signed binary numbers. The module was written in a structural way where each bit was XORed with the corresponding bit.

```
                  10 A= 9223372036854775807 B=                    8 out= 9223372036854775799
                  30 A= 9223372036854775807 B=                   10 out= 9223372036854775797
                  40 A= 9223372036854775807 B=                   14 out= 9223372036854775793
                  70 A= 9223372036854775807 B=                   30 out= 9223372036854775777
                  80 A= 9223372036854775807 B=                   94 out= 9223372036854775713
                 110 A= 9223372036854775807 B=                  606 out= 9223372036854775201
                 130 A= 9223372036854775807 B=                  862 out= 9223372036854774945
                 140 A= 9223372036854775807 B=                 4958 out= 9223372036854770849
                 150 A= 9223372036854775807 B=                 4990 out= 9223372036854770817
                 160 A= 9223372036854775807 B=                 6014 out= 9223372036854769793
                 170 A= 9223372036854775807 B=                38782 out= 9223372036854737025
                 200 A= 9223372036854775807 B=               300926 out= 9223372036854474881
                 210 A= 9223372036854775807 B=               301054 out= 9223372036854474753
                 220 A= 9223372036854775807 B=               317438 out= 9223372036854458369
                 230 A= 9223372036854775807 B=              2414590 out= 9223372036852361217
                 250 A= 9223372036854775807 B=              2480126 out= 9223372036852295681
                 260 A= 9223372036854775807 B=             19257342 out= 9223372036835518465
                 290 A= 9223372036854775807 B=            153475070 out= 9223372036701300737
                 310 A= 9223372036854775807 B=            154523646 out= 9223372036700252161
                 320 A= 9223372036854775807 B=           1228265470 out= 9223372035626510337
                 330 A= 9223372036854775807 B=           1228267518 out= 9223372035626508289
                 340 A= 9223372036854775807 B=           1232461822 out= 9223372035622313985
                 350 A= 9223372036854775807 B=           9822396414 out= 9223372027032379393
                 380 A= 9223372036854775807 B=          78541873150 out= 9223371958312902657
                 390 A= 9223372036854775807 B=          78541881342 out= 9223371958312894465
                 400 A= 9223372036854775807 B=          78608990206 out= 9223371958245785601
                 410 A= 9223372036854775807 B=         628364804094 out= 9223371408489971713
                 430 A= 9223372036854775807 B=         628633239550 out= 9223371408221536257
                 440 A= 9223372036854775807 B=        5026679750654 out= 9223367010175025153
                 470 A= 9223372036854775807 B=       40211051839486 out= 9223331825802936321
                 490 A= 9223372036854775807 B=       40215346806782 out= 9223331821507969025
                 500 A= 9223372036854775807 B=      321690323517438 out= 9223050346531258369
                 510 A= 9223372036854775807 B=      321690323648510 out= 9223050346531127297
                 520 A= 9223372036854775807 B=      321707503517694 out= 9223050329351258113
                 530 A= 9223372036854775807 B=     2573507317202942 out= 9220798529537572865
                 560 A= 9223372036854775807 B=    20587905826684926 out= 9202784131028090881
                 570 A= 9223372036854775807 B=    20587905827209214 out= 9202784131027566593
                 580 A= 9223372036854775807 B=    20588180705116158 out= 9202783856149659649
                 590 A= 9223372036854775807 B=   164703368780972030 out= 9058668668073803777
                 610 A= 9223372036854775807 B=   164704468292599806 out= 9058667568562176001
                 620 A= 9223372036854775807 B= 1317625972899446782 out= 7905746063955329025
                 650 A= 9223372036854775807 B=-7905746063955329026 out=1712911810081010483 3
                 670 A= 9223372036854775807 B=-7905728471769284610 out=17129100508624060417
  not_match=          0 match=          66
```

Fig 5.1 Output for different Inputs.

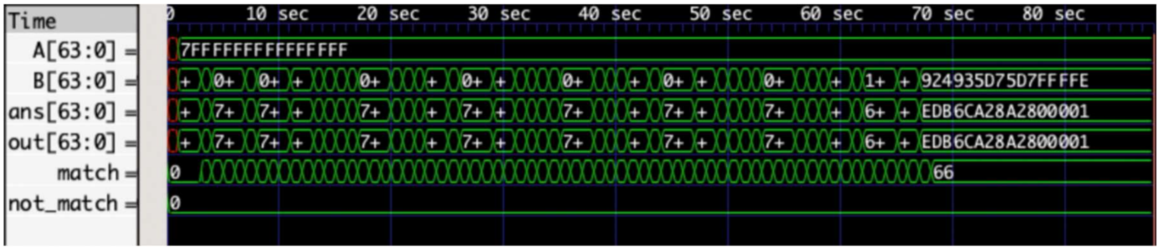
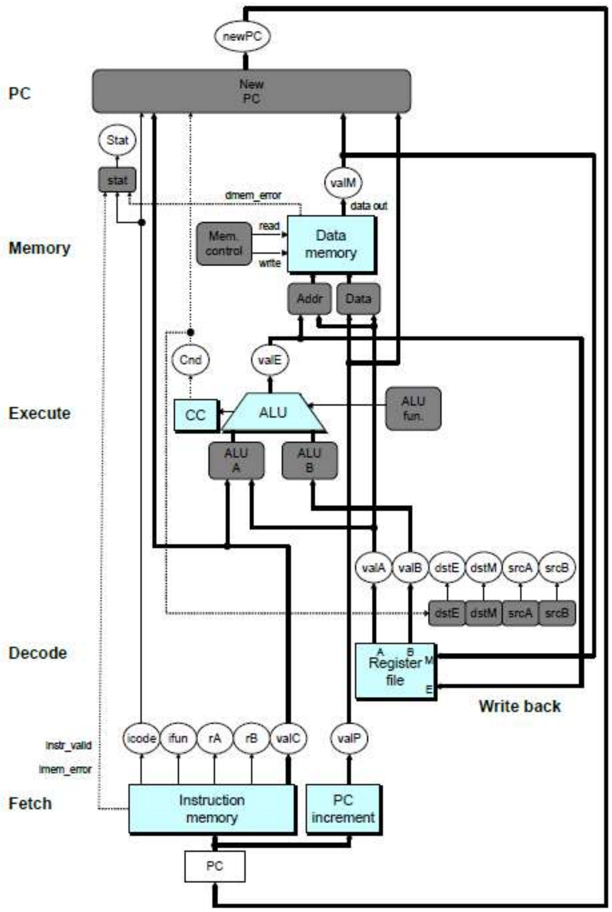


Fig 5.2 Output waveform of the results.

To verify the working of our module, we have written a test bench where we have taken random values and calculated values directly and using the module made by us to verify if both are giving the same results not.

Sequential implementation

Here, all the instructions are processed one after the other, and hence the name sequential. In this implementation, all the 5 stages are executed one after the other for the execution of the complete instruction.

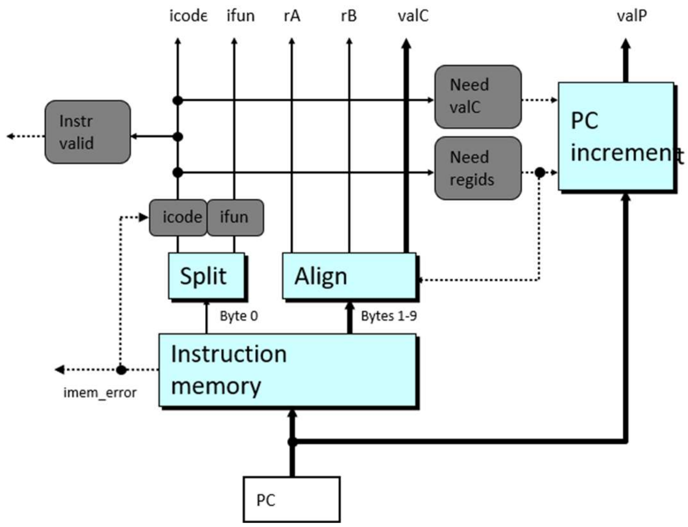


Sequential Stages

There are 5 stages in the Sequential implementation which are described below.

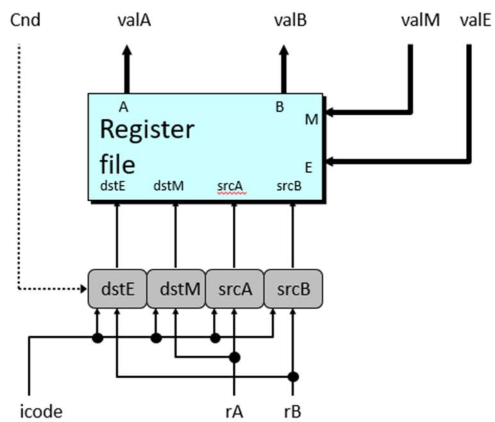
Fetch

- Reads bytes of an instruction from memory using the PC value as address and extracts the two 4-bit portions of instruction specifier byte referred to as **icode** and **ifun.**
- Fetches the register specifier byte to get rA and rB.
- Fetches 8-byte constant word valC to computes valP as the address of the next instruction in the sequence , i.e. valP = PC + length of fetched instruction



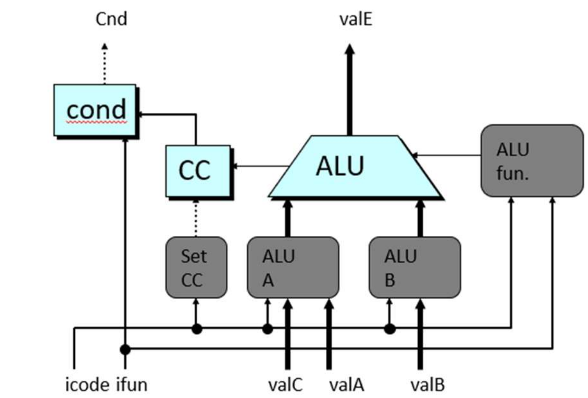
Decode

- Reads operands from the register file giving values valA and valB.



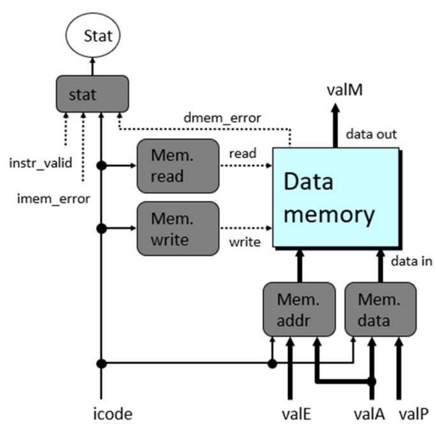
Execute

- ALU either performs operation given by ifun, computes effective address of a memory reference, or increments or decrements the stack pointer and store the resulting value in valE.
- Condition codes are set.
- Determines if a branch is taken or not for a jump instruction by testing the condition code and branch condition.



Memory

Read or write data from/to memory respectively. Value read referred to as valM.

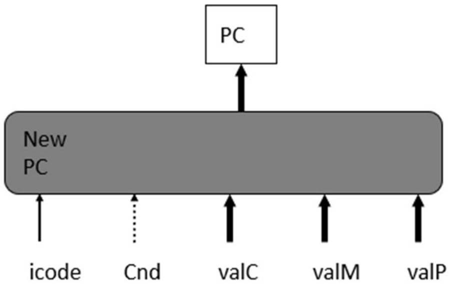


Write Back

Update register file.

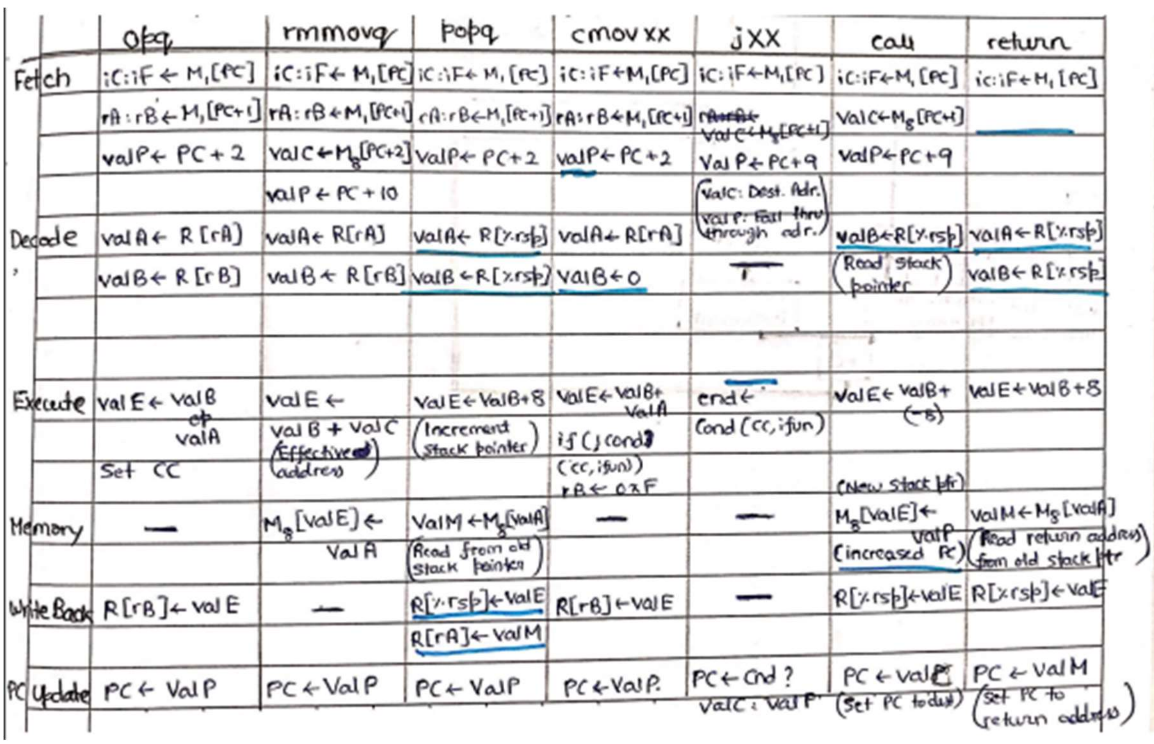
PC Update

PC is updated with the next instruction or valP.



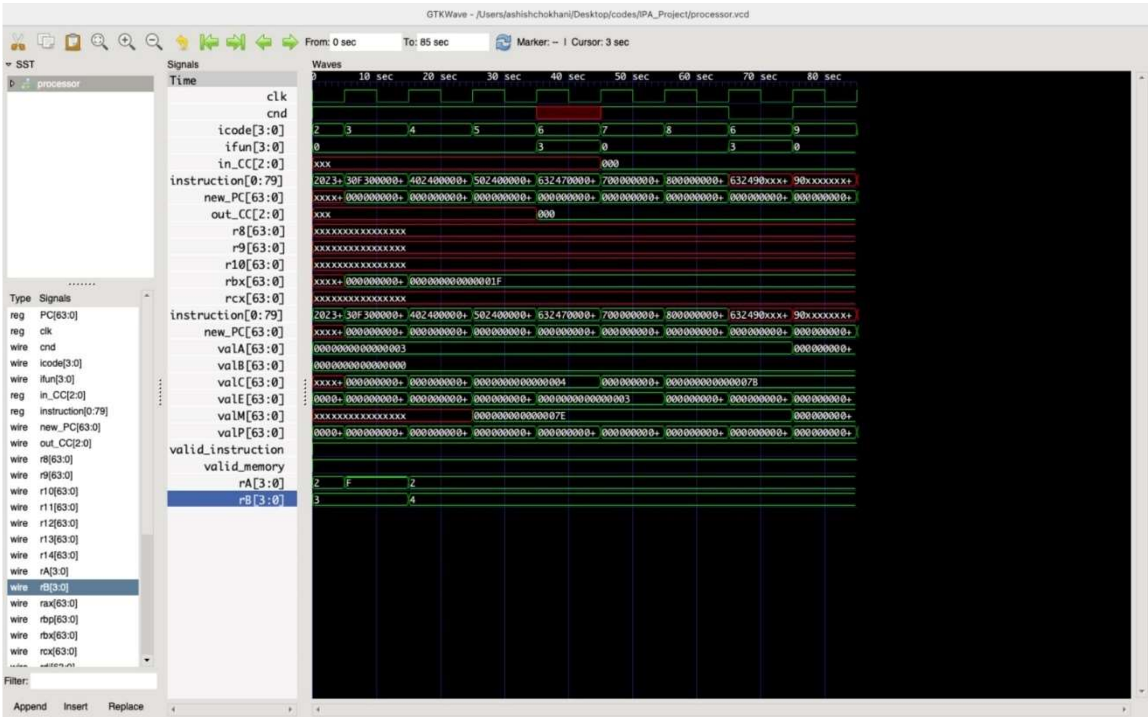
Testing

For verifying the implementation of our modules written in the Sequential stage, all the instructions in Y86 processor architecture was written in encoded form.



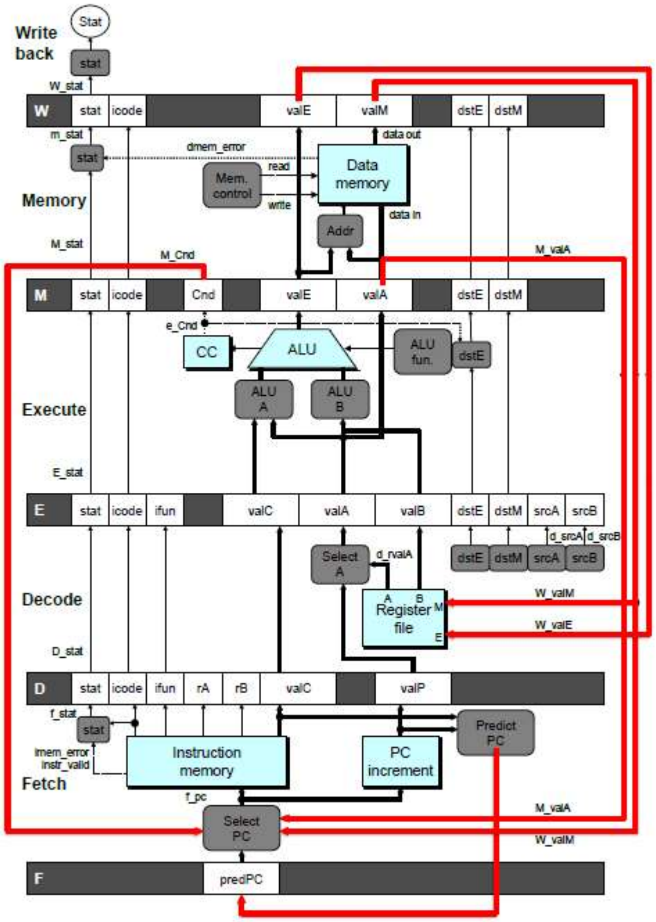
	OPq	rmmovq	popq	cmovXX	jXX	call	return
Fetch	iC:iF ← M1[PC]	iC:iF ← M1[PC]	iC:iF ← M1[PC]	iC:iF ← M1[PC]	iC:iF ← M1[PC]	iC:iF ← M1[PC]	iC:iF ← M1[PC]
	rA:rB ← M1[PC+1]	rA:rB ← M1[PC+1]	rA:rB ← M1[PC+1]	rA:rB ← M1[PC+1]	~~rA:rB~~ ValC ← M8[PC+1]	ValC ← M8[PC+1]	
	valP ← PC+2	ValC ← M8[PC+2]	ValP ← PC+2	valP ← PC+2	Val P ← PC+9	ValP ← PC+9	
		valP ← PC+10			(ValC: Dest. Adr. ValP: Fall thru through adr.)		
Decode	val A ← R[rA]	valA ← R[rA]	ValA ← R[%rsp]	valA ← R[rA]		valB ← R[%rsp]	valA ← R[%rsp]
	valB ← R[rB]	valB ← R[rB]	valB ← R[%rsp]	ValB ← 0	—	(Read stack pointer)	valB ← R[%rsp]
Execute	val E ← valB op valA	val E ← val B + ValC	ValE ← ValB+8	ValE ← ValB+ ValA	cnd ←	ValE ← ValB+ (−8)	ValE ← ValB+8
		(Effective address)	(Increment stack pointer)	if (cond)	Cond (cc, ifun)		
	Set CC			(cc, ifun)) rB ← 0xF		(New stack ptr)	
Memory	—	M8[ValE] ← ValA	ValM ← M8[ValA]	—	—	M8[ValE] ← ValP	ValM ← M8[ValA]
			(Read from old stack pointer)			(increased PC)	(Read return address from old stack ptr)
Write Back	R[rB] ← val E	—	R[%rsp] ← ValE	R[rB] ← ValE	—	R[%rsp] ← valE	R[%rsp] ← ValE
			R[rA] ← ValM				
PC Update	PC ← ValP	PC ← ValP	PC ← ValP	PC ← ValP.	PC ← Cnd ? ValC : ValP	PC ← ValC (Set PC to dest)	PC ← ValM (Set PC to return address)

The GTK Wave the Test case used is shown hereby.



GTKWave - /Users/ashishchokhani/Desktop/codes/IPA_Project/processor.vcd
Time
clk
cnd
icode[3:0]
ifun[3:0]
in_CC[2:0]
instruction[0:79]
new_PC[63:0]
out_CC[2:0]
r8[63:0]
r9[63:0]
r10[63:0]
rbx[63:0]
rcx[63:0]
valA[63:0]
valB[63:0]
valC[63:0]
valE[63:0]
valM[63:0]
valP[63:0]
valid_instruction
valid_memory
rA[3:0]
rB[3:0]

Pipeline Implementation



In processor architecture, a pipeline is a technique used to improve the performance of processors by allowing multiple instructions to be processed simultaneously.

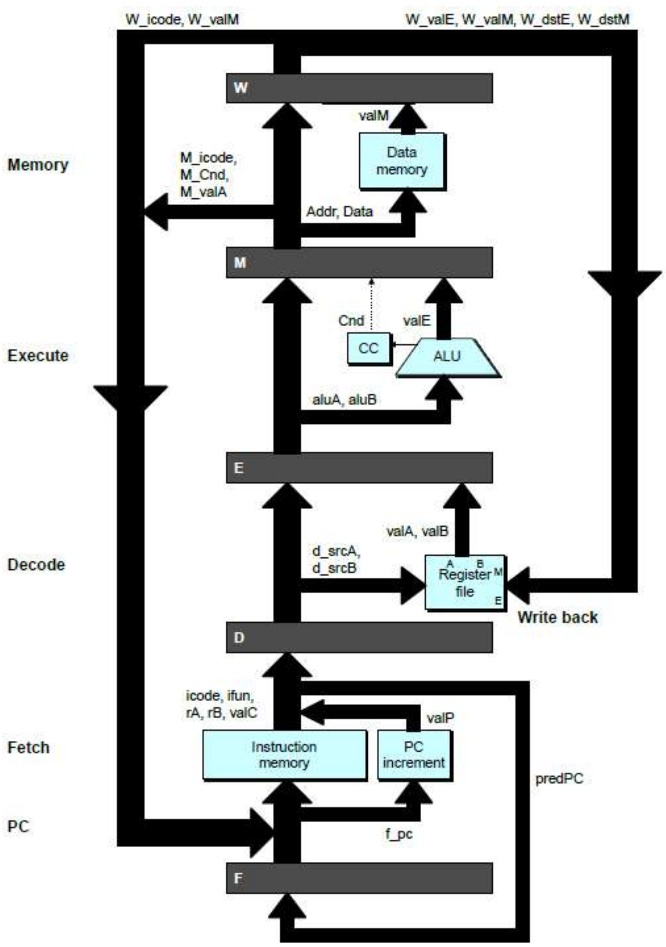
The utility of a pipeline in processor architecture lies in its ability to increase the efficiency and speed of instruction processing. By breaking down instructions into smaller tasks and processing them in parallel, pipelines can dramatically reduce the time required to execute a single instruction.

Pipelines can also improve the overall throughput of the processor by allowing multiple instructions to be processed simultaneously, which can result in a significant increase in overall processing speed.

However, the use of pipelines can also introduce some challenges, such as the potential for data dependencies between instructions and the possibility of pipeline stalls or delays. These issues must be carefully managed to ensure that the benefits of the pipeline are realized without introducing additional problems.

Pipeline Stages

A pipeline consists of several stages, each of which performs a specific task. When an instruction enters the pipeline, it is split into a series of smaller tasks that can be processed in parallel in different stages of the pipeline. This allows multiple instructions to be processed simultaneously, improving the overall throughput of the processor.



Fetch

Select current PC

Read instruction

Compute incremented PC

Decode

Read program registers

Execute

Operate ALU

Memory

Read or write data memory.

Write Back

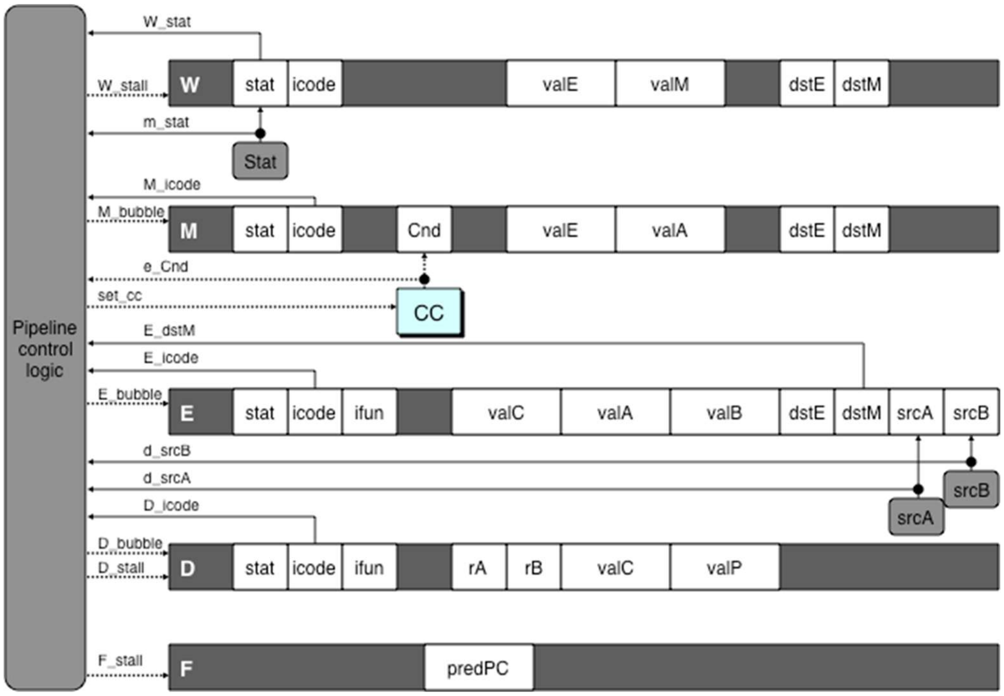
Update register file.

Implementation

- Stalling:

Stalling instruction is required when the data required in the decode stage isn't yet updated with the new value. It holds back instruction in decode stage and the ollowing instruction stays in fetch stage.

It injects Bubbles into execute stage like dynamically generated nop's.

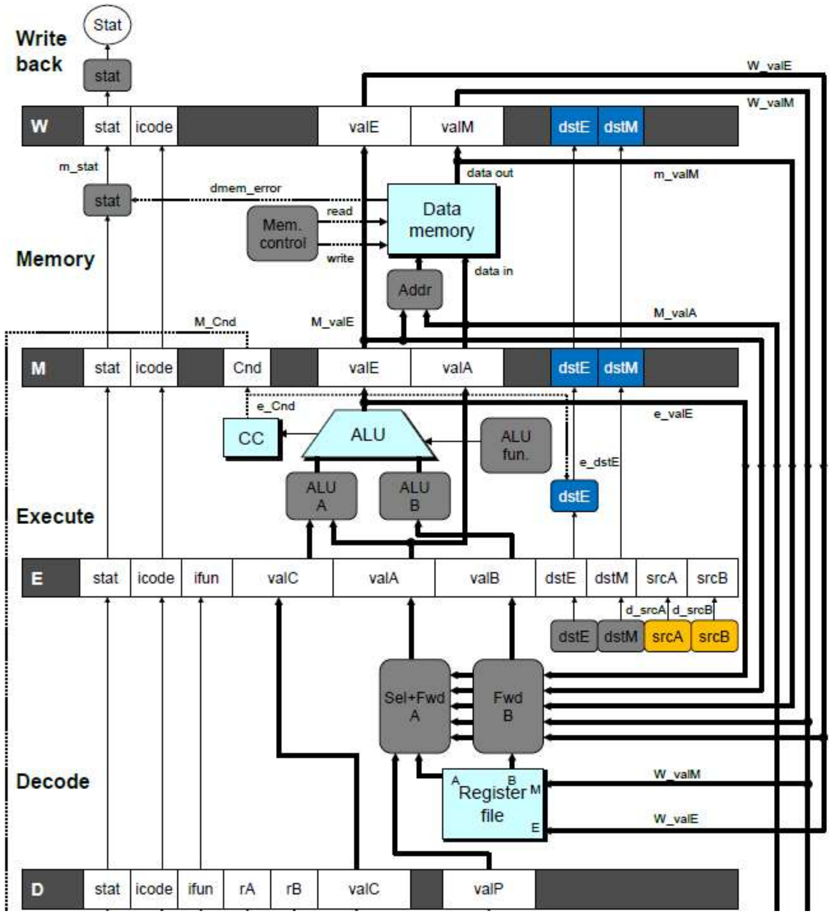


To implement Stalling in Pipeline Control, Combinational logic is used which detects stall condition. And then mode signals are set for how pipeline registers should update.

- ## Data Forwarding

Here, the values are passed directly from generating instruction to decode stage which needs to be available at end of decode stage.

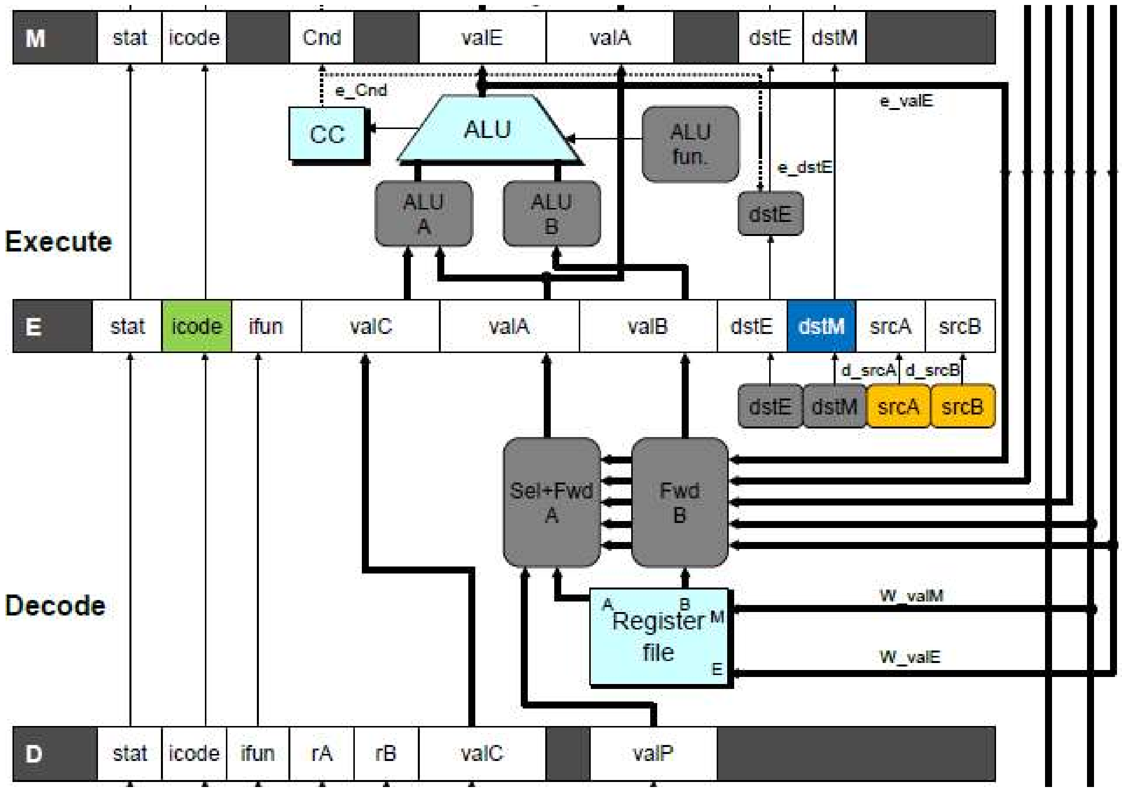
Forwarding logic selects valA and valB from various Forwarding sources such as valE from Execute stage, valM from Memory stage, valE and valM from Write-back stage from later pipeline stage.



To implement Data Forwarding, we add additional feedback paths from Execute, Memory, and Write-back pipeline registers into decode stage. Logic blocks are created to select from multiple sources for valA and valB in decode stage.

- ## Load/Use Hazard

We need to wait till the Memory stage in mrmovq instruction to get the updated memory values. So, we need to wait for the instruction to pass the Memory stage.



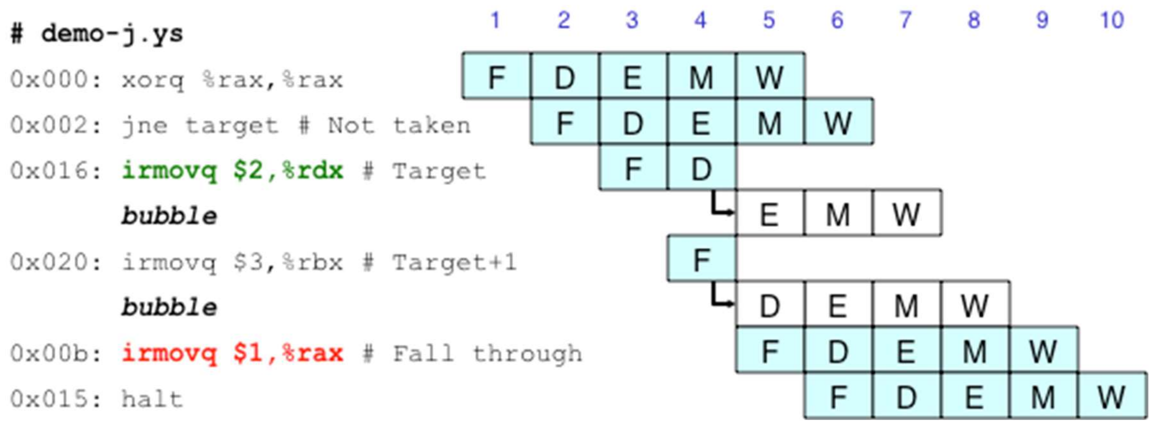
So, we Stall instruction for one cycle so that we can then pick up loaded value by forwarding from memory stage.

Testing

To test the correctness of the modules written, test cases were created for various Data Hazards.

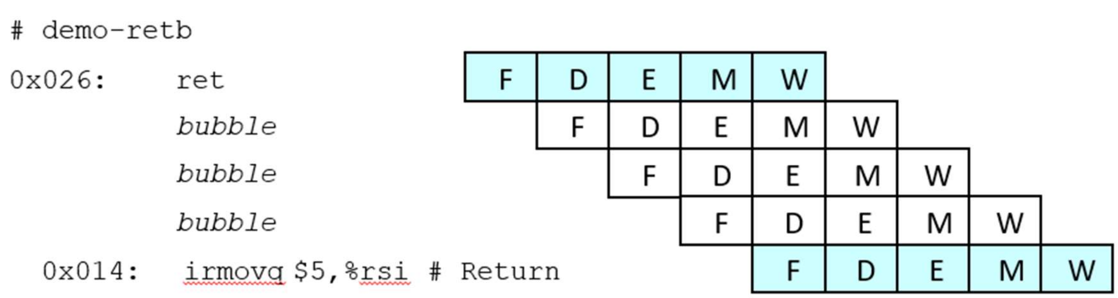
- Branch Misprediction

```
0x000:      xorq %rax,%rax
0x002:      jne   t              # Not taken
0x00b:      irmovq $1, %rax      # Fall through
0x015:      nop
0x016:      nop
0x017:      nop
0x018:      halt
0x019: t: irmovq $3, %rdx      # Target
0x023:      irmovq $4, %rcx      # Should not execute
```

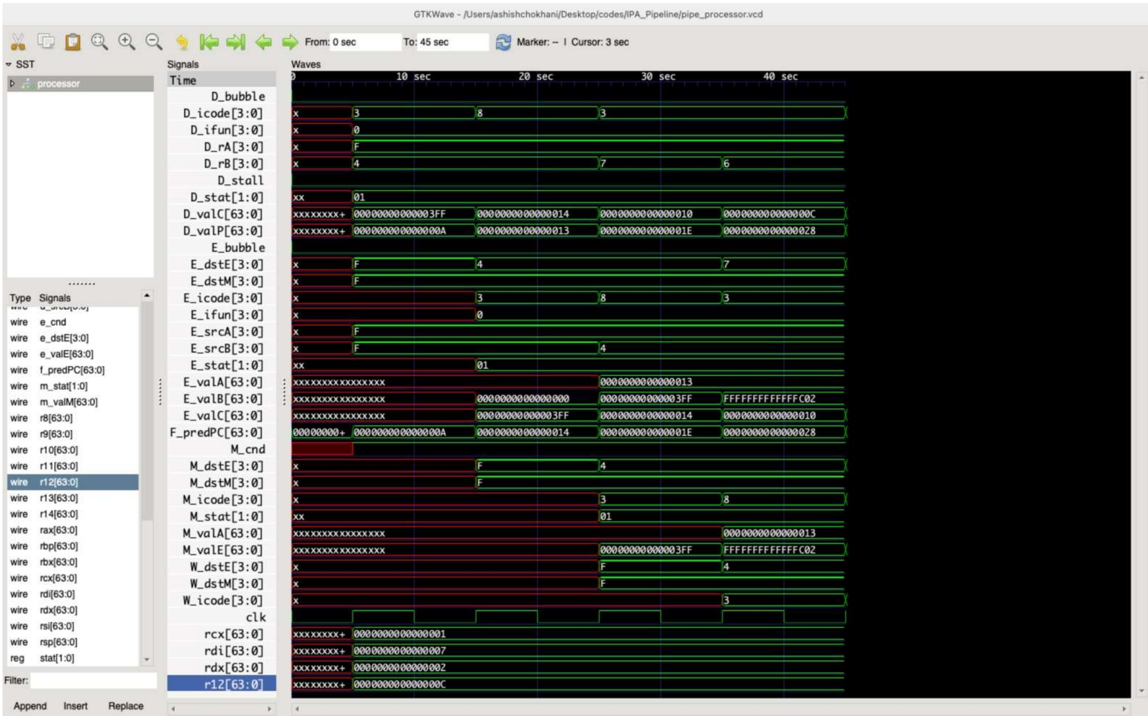


- Return

```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020: .pos 0x20
0x020: p: irmovq $-1,%rdi     # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100: .pos 0x100
```



The GTK Wave of the corresponding test cases is shown below:



Problems encountered.

Considering the inter relationship and feedbacks among the 5 stages in pipeline registers, it was very complicated to remove errors.

In sequential stage as well, data computed in one stage was used in other later stages. So, overall data dependencies were high making it difficult to write the correct code.

Future developments

We can create an automated testbench to verify the design efficiently which will automatically verify the state of the processor and memory after execution of each instruction in the program.