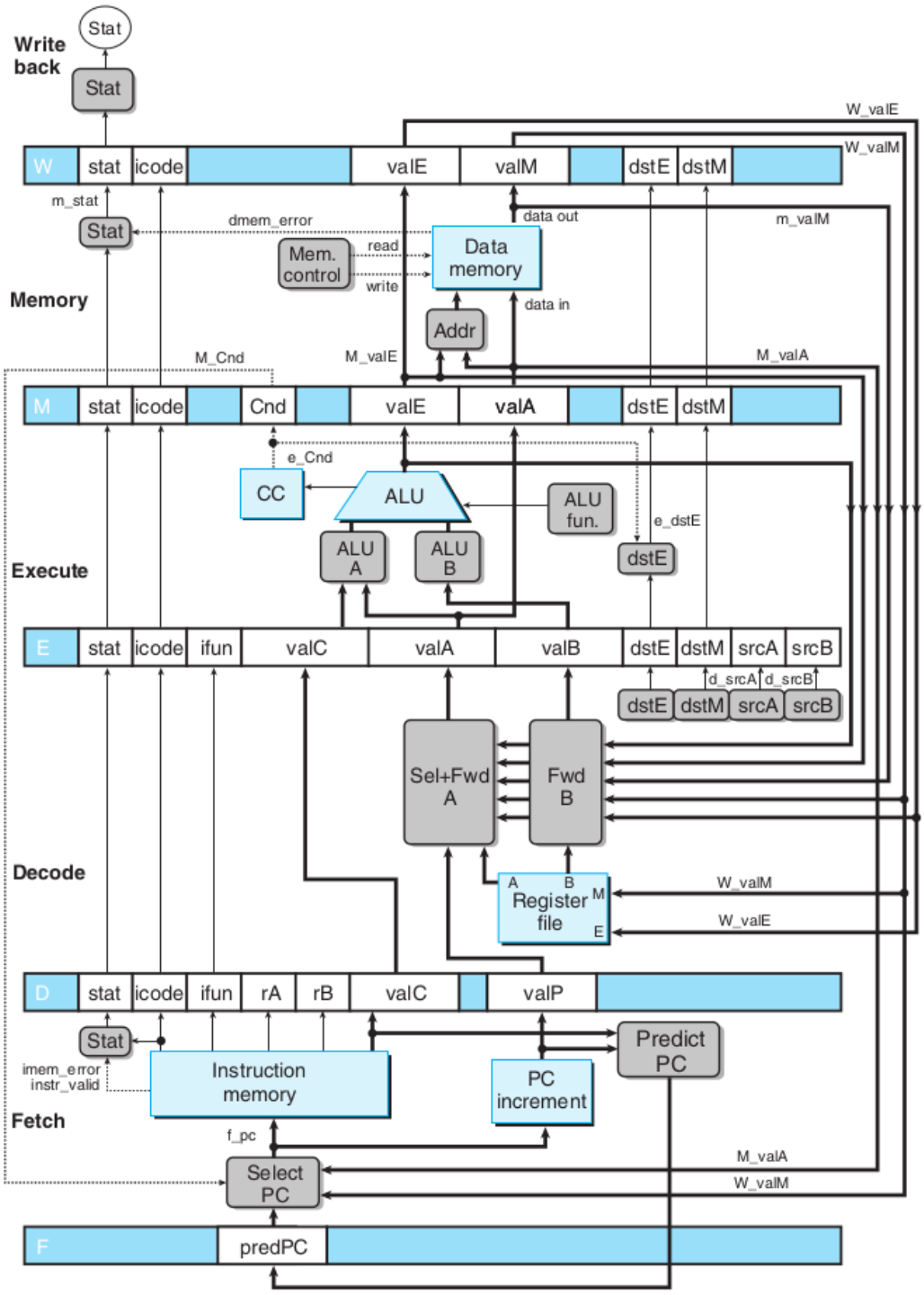
IPA Project report

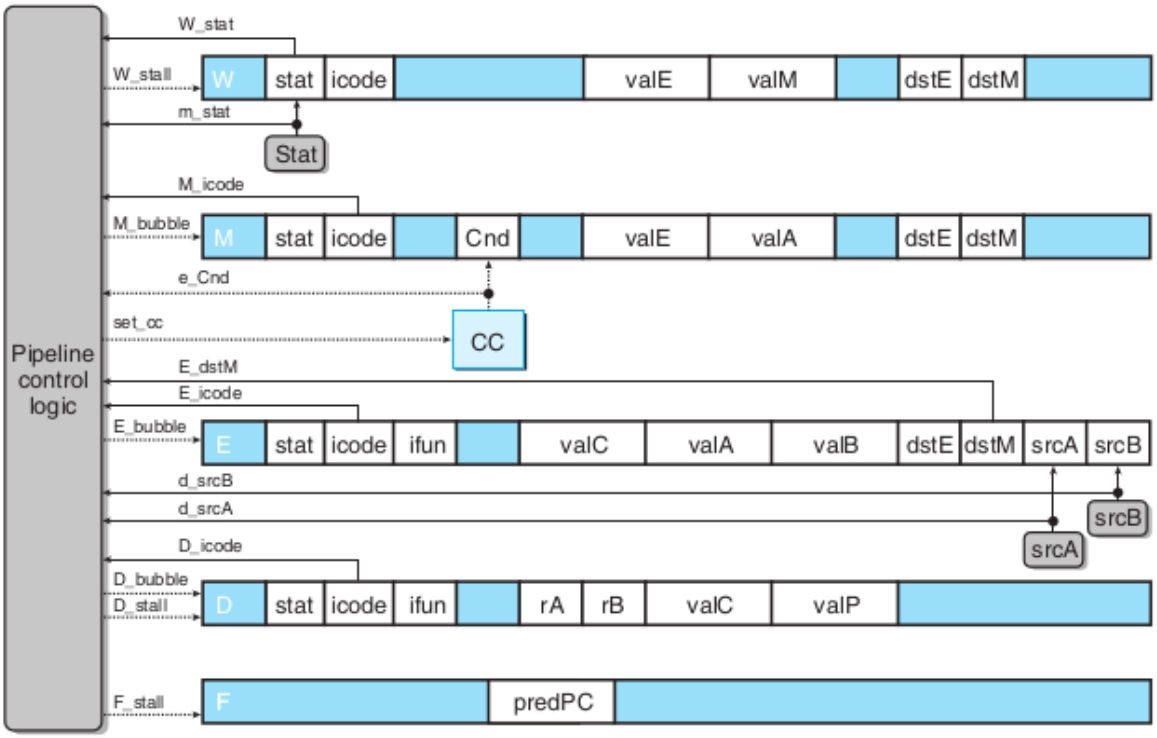
shaantanu kulkarni
2019102031

The SEQ implementation of our processor was too slow and so we modified the design of our processor by adding registers between each stage and by using techniques such as forwarding , stalling and addition of bubbles to rectify the predicted PC.
The diagram below shows how each stage in connected with other using pipelining registers.



Pipeline control logic

The figure below shows how the Pipeline control logic is connected with the pipelining registers.

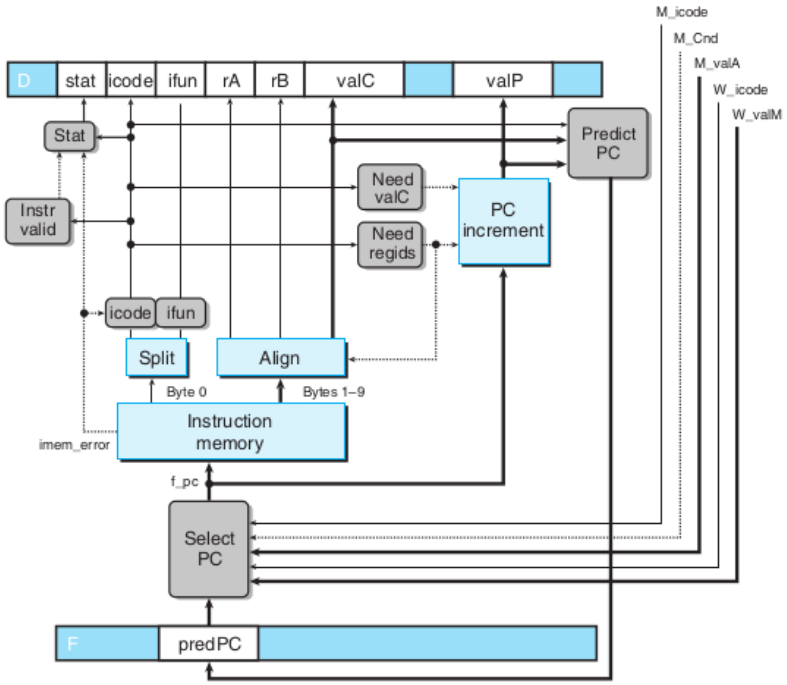


My processor handles these four case with the help of this pipeline logic :

- Load/use hazards. The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
- Processing ret. The pipeline must stall until the ret instruction reaches the write-back stage.
- Mispredicted branches. By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.
- Exceptions. When an instruction causes an exception, we want to disable the
  updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

Individual stages of the pipelined architecture

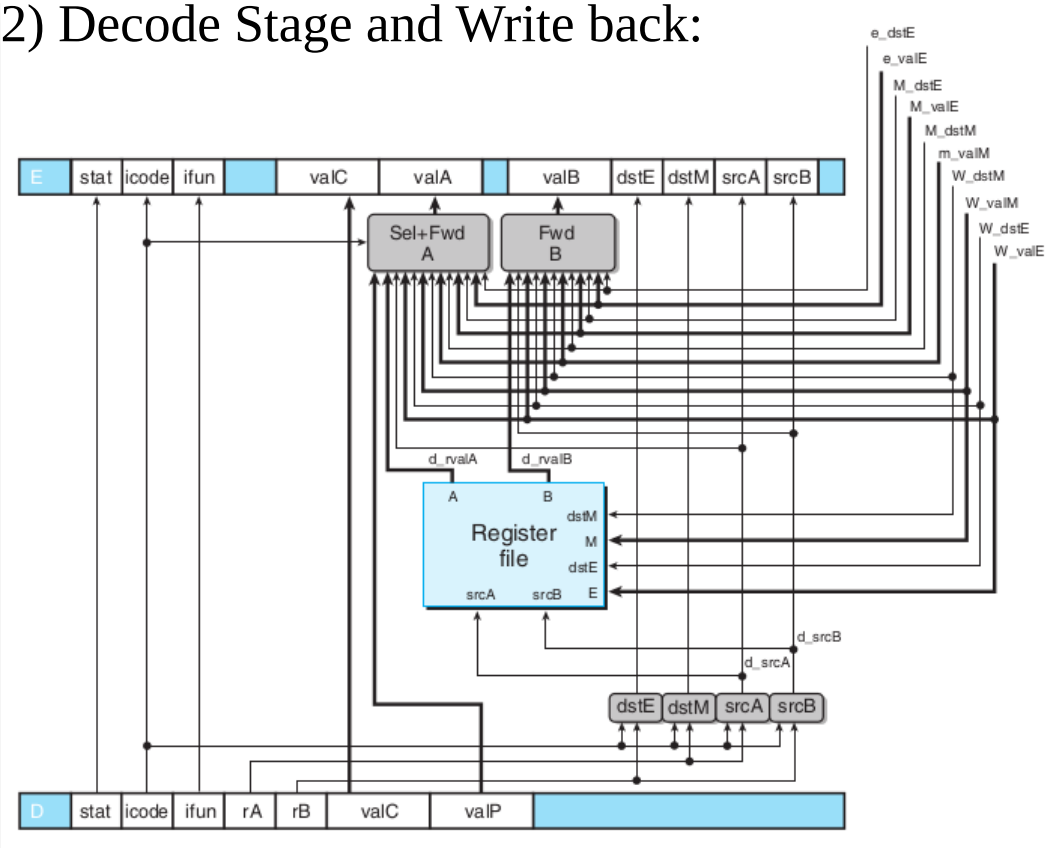
1) Fetch and PC selection:



This stage, depending on f_pc , will read 10 bytes and decide how many bytes it actually has to read. On the basis of this decision, the Predict PC block predicts the location of the next instruction in the instruction memory.
The instruction memory is implemented by using a 8 X 2048 memory array and the values in the memory are initiallized at the begining of the first cycle of the processor.

```
module InstructionMemory(f_pc,f_ibyte,f_ibytes,imem_error);
    input [63:0] f_pc;
    output reg[7:0] f_ibyte;
    output reg[71:0] f_ibytes;
    output reg imem_error;

    reg [7:0] instruction_mem [2047:0];

    initial
        begin
            $readmemh("./rom.mem", instruction_mem);
            $display("init");
        end
    always @(f_pc)
    begin
        $display(instruction_mem[f_pc]);
        f_ibyte <= instruction_mem[f_pc];
        f_ibytes[71:64] <= instruction_mem[f_pc+1];
        f_ibytes[63:56] <= instruction_mem[f_pc+2];
        f_ibytes[55:48] <= instruction_mem[f_pc+3];
        f_ibytes[47:40] <= instruction_mem[f_pc+4];
        f_ibytes[39:32] <= instruction_mem[f_pc+5];
        f_ibytes[31:24] <= instruction_mem[f_pc+6];
        f_ibytes[23:16] <= instruction_mem[f_pc+7];
        f_ibytes[15:8] <= instruction_mem[f_pc+8];
        f_ibytes[7:0] <= instruction_mem[f_pc+9];

        imem_error <= (f_pc < 64'd0 || f_pc > 64'd2047 ) ? 1'b1:1'b0;

    end

endmodule
```

2) Decode Stage and Write back:



This stage, on the basis of the D pipelining registers, decides what value to pass to the E pipelined registers, ie,

- d_ValA
- d_ValB
- dstE
- dstM
- srcA
- srcB

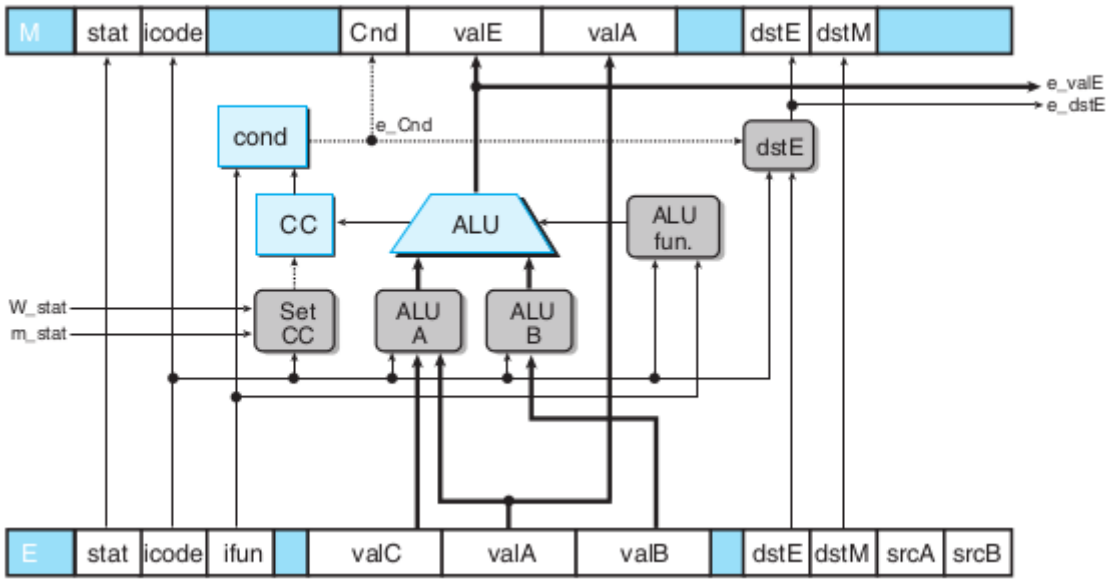
Also, since there can be load/use hazards we need to add two more blocks namely

- Sel + Fwd A
- Fwd B

The block labeled “Sel+Fwd A” serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA. Similar functionality is of Fwd B.
The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP.
The priority order for Sel + Fwd A is

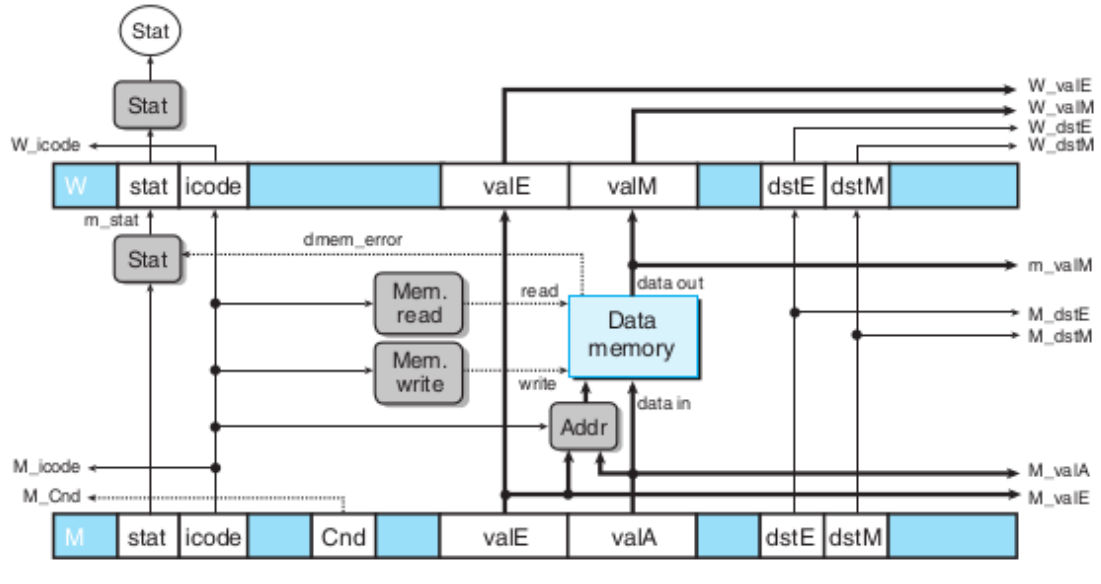
Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

3) Execute Stage:



The hardware units and the logic blocks are identical to those in SEQ architechture. We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled “Set CC,” which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs.
The “Cond” block decides whether e_Cnd should be ‘1’ or ‘ 0’ on the basis of E_ifun and CC.

4) Memory Stage:

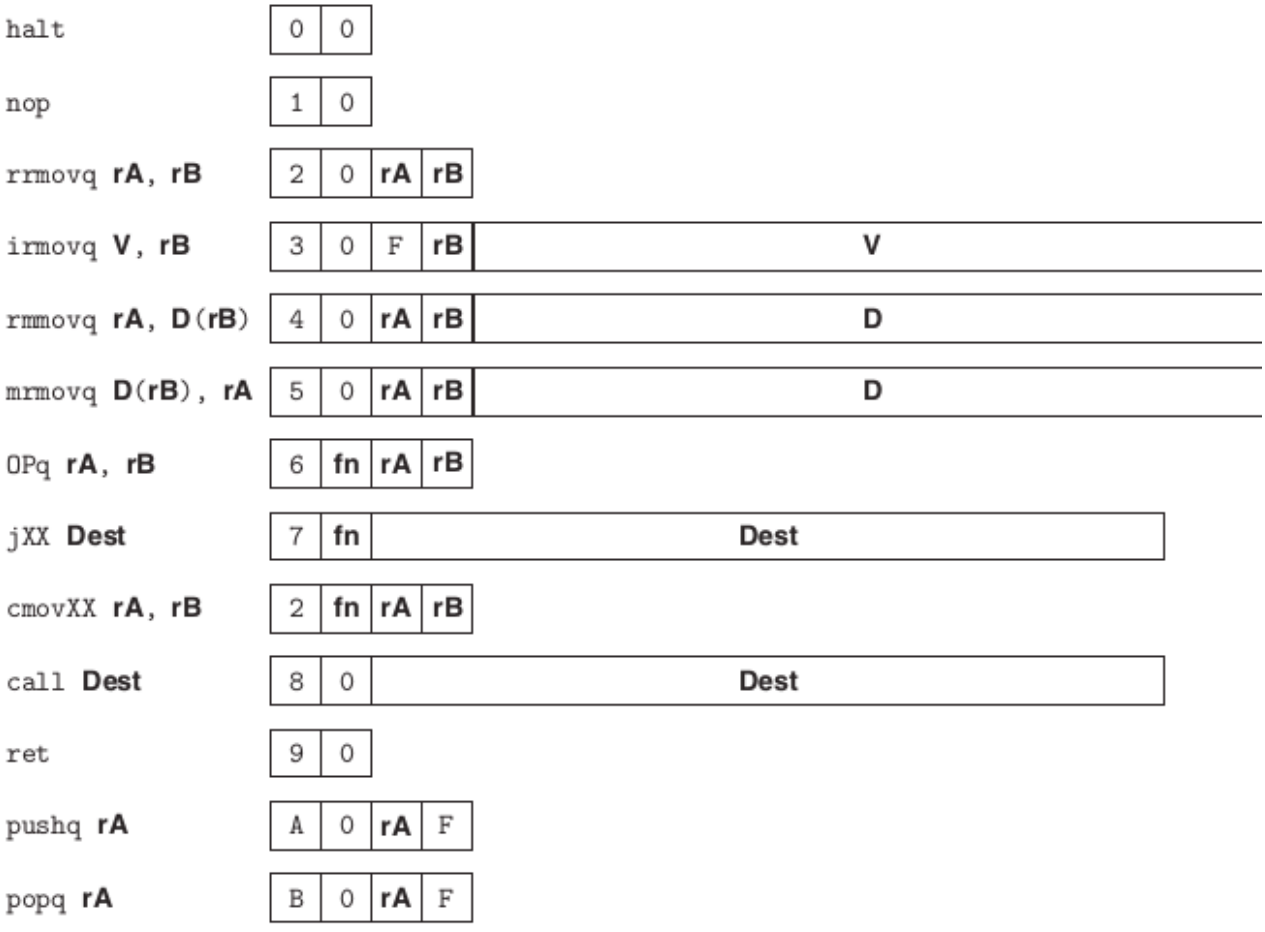


This stage has the main component that is Data memory where data can be stored . The read and write wires determines which operation has to be followed and mem_addr is the address of the memory location where data has to be written ie “M_ValA” or read ,ie, to m_ValM. It also passes the necessary data to the next pipeline registers.

Set of Instructions supported by this processor

- rrmovq
- irmovq
- rmmovq
- mrmovq
- Opq
- jXX
- cmovXX
- pushq
- popq
- halt
- nop
- call
- ret

All the instruction in the Y86 instruction set are supported in the processor. Each of the stages have been tested on individual testbench before integrating all the modules together into one main module namely ‘cpu.v’ .
The figure below shows the OPCODES of each of these instructions.



Testing the processor on standard algorithms

The processor has been tested on the following algorithms for calculating:

- HCF of two numbers
- Bubble sort

The following table shows the C++ code for the two algorithms

Algorithm	HCF of two numbers	Bubble Sort

Code	#include <iostream>
	using namespace std;
	int main()
	{
	int n1, n2;
	cout << "Enter two numbers:
	";
	cin >> n1 >> n2;
	while(n1 != n2)
	{
	if(n1 > n2)
	n1 -= n2;
	else
	n2 -= n1;
	}
	cout << "HCF = " << n1;
	return 0;
	}
		#include<iostream>
		using namespace std;
		int main ()
		{
		int i, j,temp,pass=0;
		int a[10] = {10,2,0,14,43,25,18,1,5,45};
		cout <<"Input list ...\n";
		for(i = 0; i<10; i++) {
		cout <<a[i]<<"\t";
		}
		cout<<endl;
		for(i = 0; i<10; i++) {
		for(j = i+1; j<10; j++)
		{
		if(a[j] < a[i]) {
		temp = a[i];
		a[i] = a[j];
		a[j] = temp;
		}
		}
		pass++;
		}
		cout <<"Sorted Element List ...\n";
		for(i = 0; i<10; i++) {
		cout <<a[i]<<"\t";
		}
		cout<<"\nNumber of passes taken to sort the list:"<<pass<<endl;
		return 0;
		}

Simulation specific note:

- **The output of each registers are displayed on the terminal after each clock cycle. This technique proved to be a good debugging tool.**
- **The Final state of the Data memory is outputted in ‘data_out.mem’**
- **The initial state of the Date memory can be provided in ‘data.mem’ before the simulation is executed**
- **The instruction memory is loaded with instructions stored in ‘rom.mem’.**

```
initial
    begin
        $readmemh("./rom.mem", instruction_mem);
        $display("init");
    end
```

Assembly code and simulations

HCF of two numbers:

The following assembly program calculates the HCF of two numbers , for the current test case we take it to be 56 and 76.
HCF(56,76) = 4.

HCF_assembly.txt

```
1   irmovq $56 %rax
2   irmovq $76 %rbx
3   check:
4       rrmovq %rbx %rcx
5       subq %rax %rbx
6       jg swap
7       jl subtract
8       halt
9   subtract:
10      subq %rax %rbx
11      jmp check
12  swap:
13      rrmovq %rax %rcx
14      rrmovq %rbx %rax
15      rrmovq %rcx %rbx
16      jmp subtract
```

The encoded HEX values

30 F0 0000000000000038
30 F30000000000000062
20 31
61 01
72 0000000000000036
76 000000000000002B
00
61 03
70 0000000000000014
20 01
20 30
20 13
70 000000000000002B

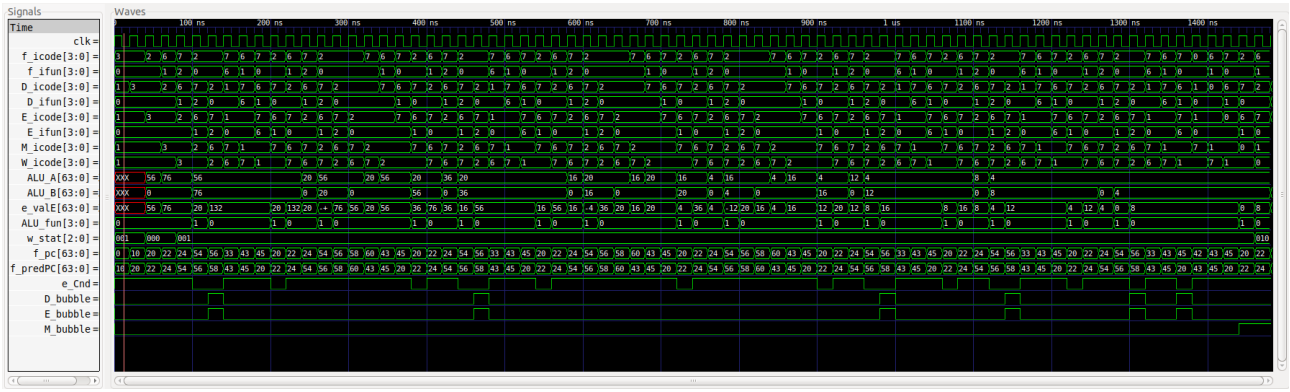
The complete hex codes is in ‘HCF.mem’

- *The corresponding HEX code can be found in ‘HCF.mem’. During conversion to HEX code , the labels were removed and exact memory locations were hardcoded for the conditional/nonconditional jump instructions*

As expected , the HCF value will be stored in both the registers rax and rbx (thats the condition for halt ,ie, when rax == rbx)
The figure below shows the values in rax and rbx

```
Current register status:
rax :                4   rsp:                0   r8  :                0  r12 :                0
rcx :                0   rbp:                0   r9  :                0  r13 :                0
rdx :                0   rsi:                0   r10 :                0  r14 :                0
rbx :                4   rdi:                0   r11 :                0  F   : XX
```

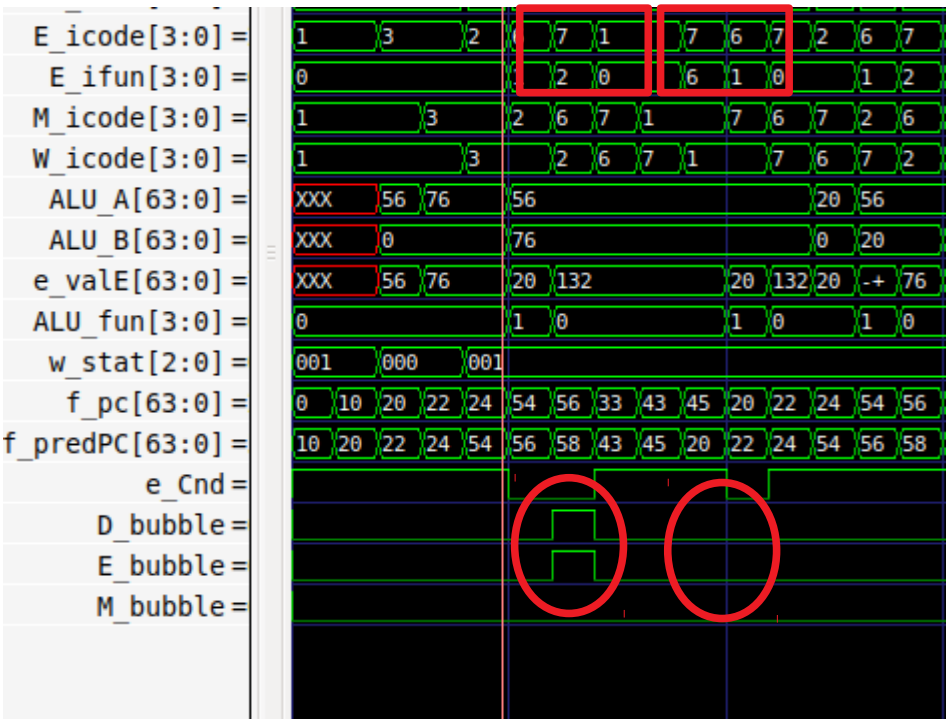
Outputs of GTKWAVE:



The gtkwave output for complete simulation time ,ie, about 1500 ns.
The clock value toggles after every 10ns and thus the timeperiod is 20ns.

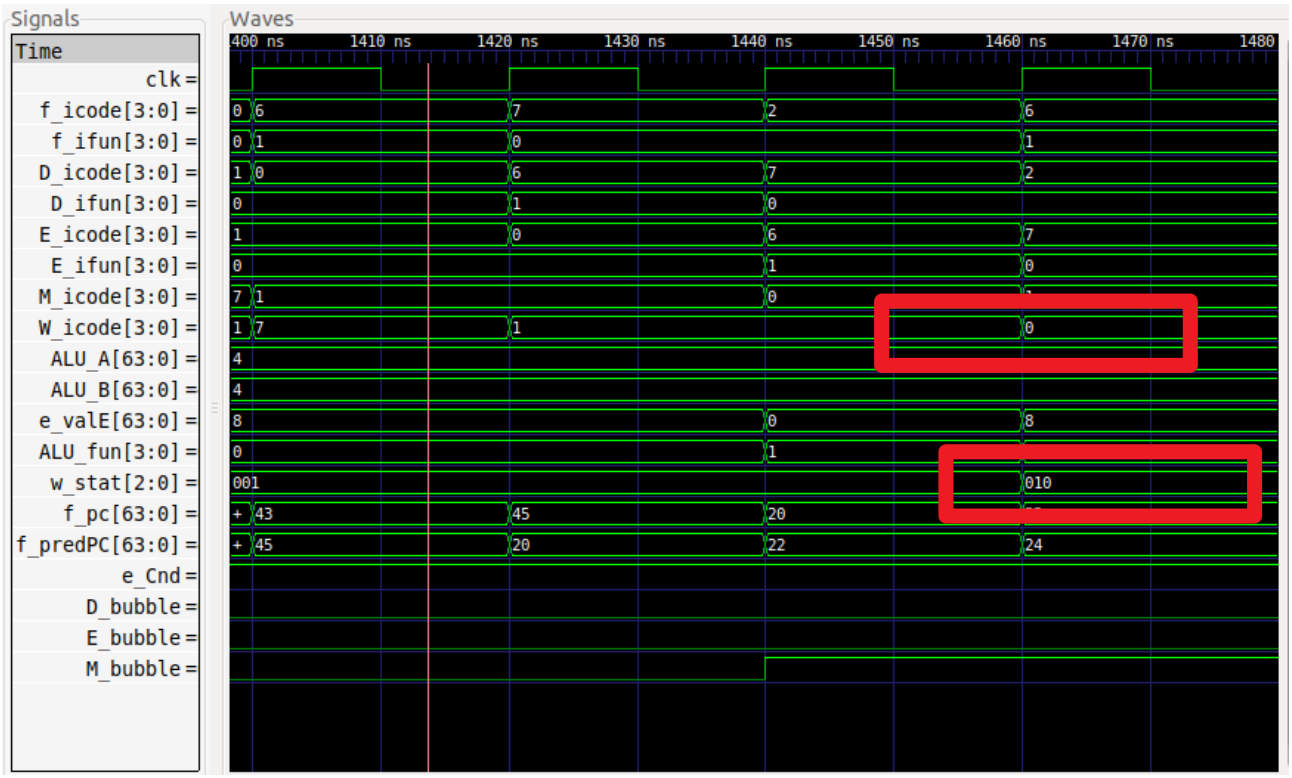
We know that the conditional jump happens even before checking if the conditions were true
If it is found that the jump we took was a misprediction for the PC value, we generated bubble signal ,ie, basically NOPs in decode and execution because by the time the processor realises that it took wrong jump, two operations would have already been fetched by the fetch stage .
In the assembly code, we are using four jump instructions, two conditional and two non conditional.
If the prediction was correct and the jump had to be taken, none of E_bubble or D_bubble signal is set '1'. and the processor continues with normal execution.
If the predition was false, E_bubble and D_bubble are set true and the processor fixes its PC value and normal operation starts thereafter.

In our assembly code, the first jump condition for the very first time when is fetched, it compares %rax and %rbx and jumps if %rax > %rbx
Since this is not true for the first time, E_bubble and D_bubble will be generated.
The second jmp checks if %rax> $rbx , and since its true E_bubble and D_bubble wont be generated. This can be seen from the figure below.



Also , we can see the following NOP caused by the bubbles in following instruction.

We also see that the simulation is terminated when W_stat = SHLT (010 it is three bit register in this processor) and '0' in W_icode.



Some other Test cases:

HCF(579,8764) = 1

```
Current register status:

rax :                   1   rsp:                   0   r8  :                   0  r12 :
     0

rcx :                   0   rbp:                   0   r9  :                   0  r13 :
     0

rdx :                   0   rsi:                   0   r10 :                   0  r14 :
     0

rbx :                   1   rdi:                   0   r11 :                   0  F   : XX
```

HCF(150,225) = 75

```
Current register status:

rax :                  75   rsp:                   0   r8  :                   0  r12 :
     0

rcx :                   0   rbp:                   0   r9  :                   0  r13 :
     0

rdx :                   0   rsi:                   0   r10 :                   0  r14 :
     0

rbx :                  75   rdi:                   0   r11 :                   0  F   : XX
```

Sorting of an array:

The following program takes the number of elements in array and stores in %rcx and then uses bubble sort to sort the array stored in the Data memory initially and store it back in data memory.

sort_assembly.txt

```
1   irmovq $1 %r11
2   irmovq $1 %rax
3   irmovq $0 %rbx
4   mrmovq 0(%rbx) %rcx
5   irmovq $1 %rdx
6   outloop:
7       subq %rdx %rcx
8       rrmovq %rcx %rdx
9       rrmovq %rax %rbx
10  inloop:
11      mrmovq 0(%rbx) %r8
12      mrmovq 1(%rbx) %r9
13      rrmovq %r9 %r10
14      subq %r8 %r10jl .swap
15  restore:
16      rmmovq %r8 0(rbx)
17      rmmovq %r9 0(rbx)
18      addq %r11 %rbx
19      subq %r11 %rdx
20      je .inloop
21      subq %r11 %rcx
22      je .outloop
23      halt
24  swap:
25      rrmovq %r8 %r10
26      rrmovq %r9 %r8
27      rrmovq %r10 %r9
28      jmp .restore
29
```

The encoded HEX values

30 FB 0000000000000001
30 F0 0000000000000001
30 F3 0000000000000000
50 13 0000000000000000
30 F2 0000000000000001
61 21
20 12
20 03
50 83 0000000000000000
50 93 0000000000000001
20 9A
61 8A
72 000000000000008A
40 83 0000000000000000
40 93 0000000000000001
60 B3
61 B2
61 72
76 0000000000000038
61 B1
61 71
76 0000000000000032
00
20 8A
20 98
20 A9
70 0000000000000059

- *The corresponding HEX code can be found in 'sorting.mem'. During conversion to HEX code , the labels were removed and exact memory locations were hardcoded for the conditional/nonconditional jump instructions*

The input array that is loaded in the Data memory is stored in data.mem and the final state of the memory is stored in data_out.mem

INPUT array

data.mem

```
1   00000006
2   00000004
3   00000003
4   00000002
5   00000001
6   0000000A
7   00000000
```

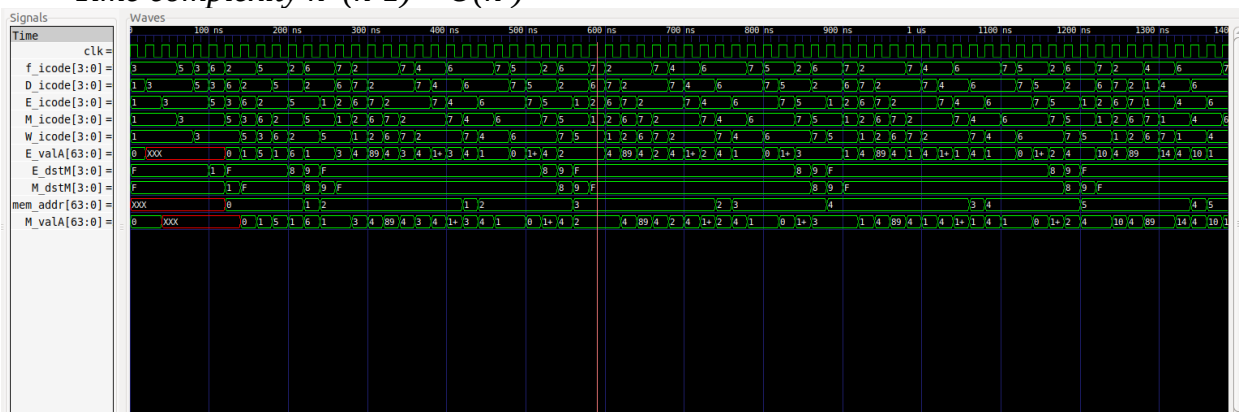
OUTPUT array

data_out.mem

```
1   // 0x00000000
2   0000000000000006
3   0000000000000000
4   0000000000000001
5   0000000000000002
6   0000000000000003
7   0000000000000004
8   000000000000000a
9   xxxxxxxxxxxxxxxx
```

From the above screenshots we can see that the array is sorted properly and stored in memory back.

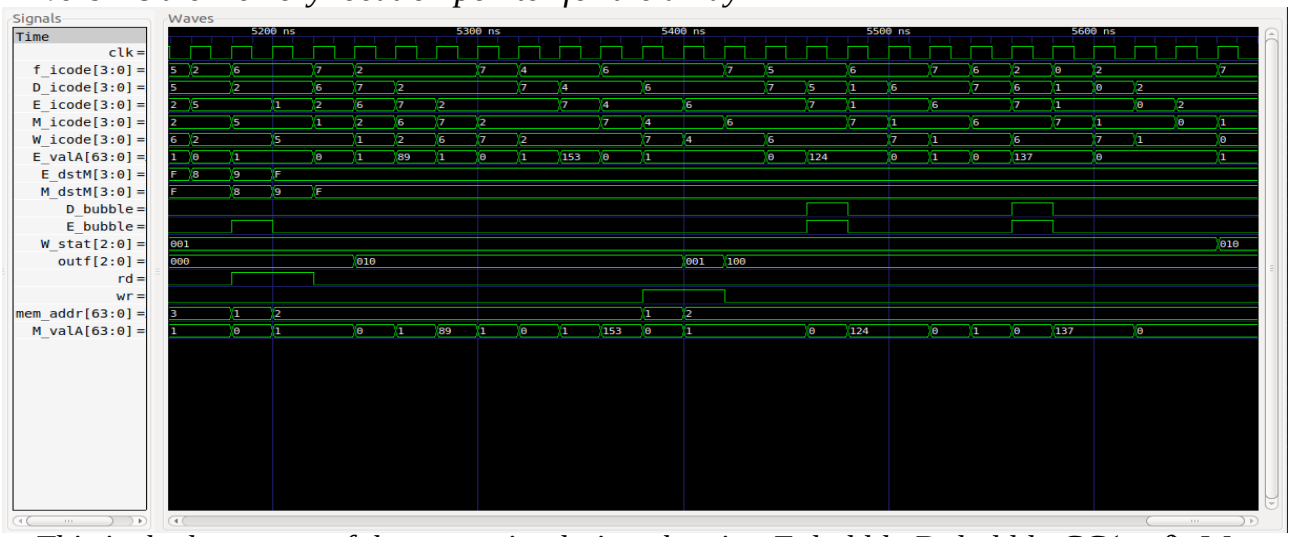
- *The number of elements in array is stored at location 0x00000000 say n and the following n locations store the unsorted elements of the array*
- *The simulation executes 'bubble sort ' to compare each consecutive element of this array and the swap them accordingly*
- *Time complexity n*(n-1) ~ O(n^2)*



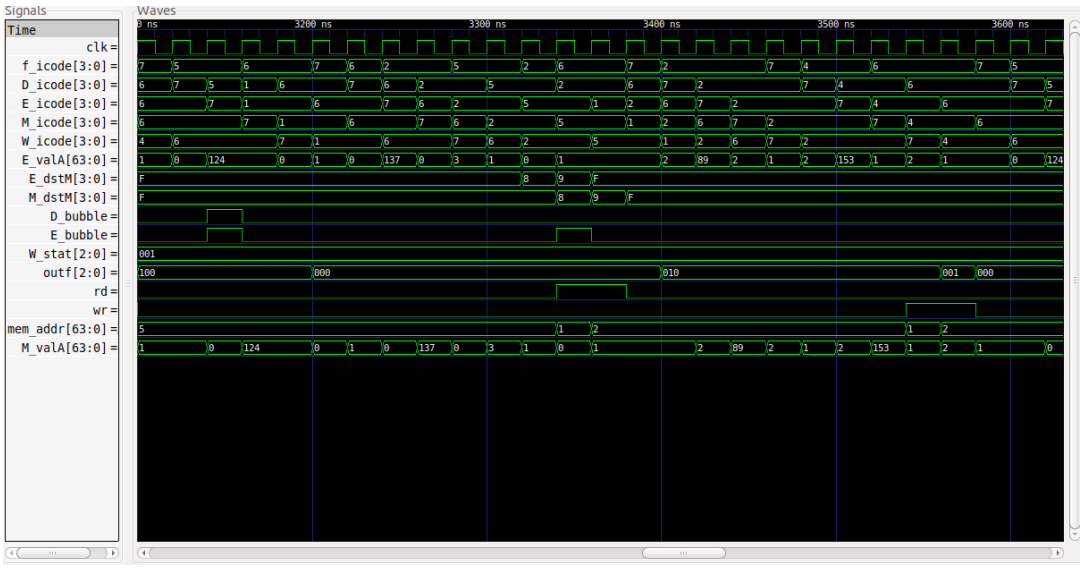
The above figure is not the complete simulation time as it wasn't fitting in one frame. It shows first few cycles from the start.

- *The M_ValA stores the data that has to be written in the Data memory*
- *mem_addr has the address from where the data has to be read or where it has to be written*
- *M_dstM shows the register address for write back*
- *We can see that mem_addr ranges from 1-6 because our data is stored from 0x00000001 to 0x00000006*
- *M_dstM has values 8,9 because we are using %r8 and %r9 for comparing during sorting*

- %rcx stores the number of elements and acts as counter for outer loop
- %rdx stored inner loop counter
-% r11 , %rax are uesd to store constant values
-%rbx is the memory location pointer for the array



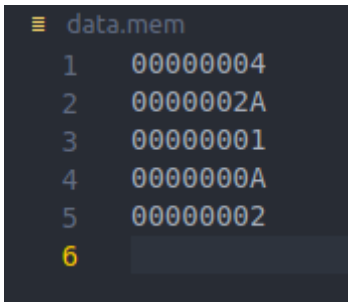
This is the later part of the same simulation showing E_bubble,D_bubble,CC(outf), W_stat, read and write enables

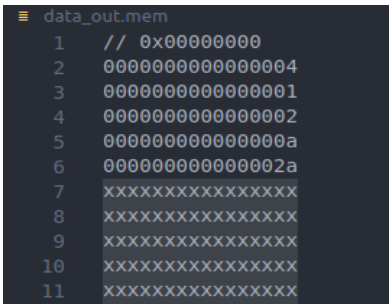


Some intermediate cycles of the same simulation

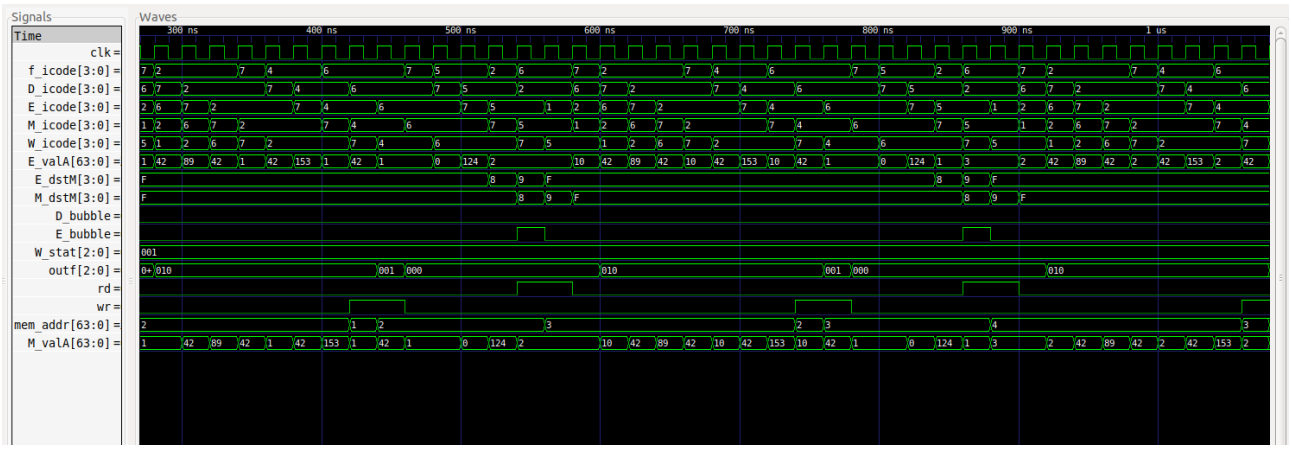
Some other test cases:

1)

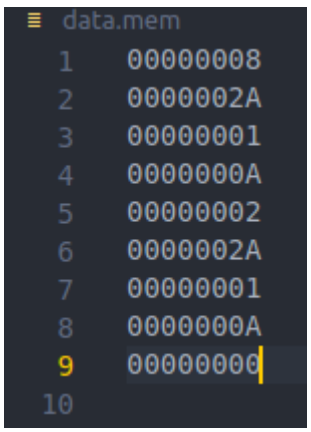


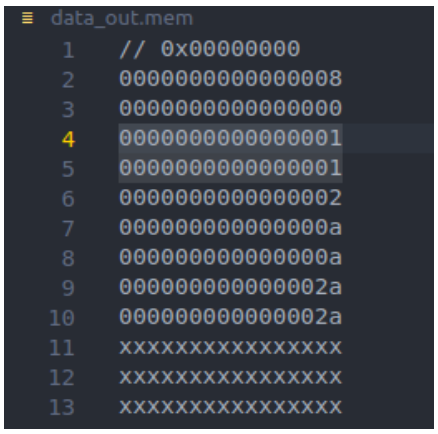


gtkwave output of intermediate cycles



2)





For further testing, the data input can be changed dynamically in data.mem and on running the simulation , the output of sorted array can be found in data_out.mem.

Instructions to run the top module:

- *The code that has to be tested needs to be written in rom.mem*
- *The HCF hexcodes are written in HCF.mem and that for sorting are in sorting.mem*
- *Copy paste the code into rom.mem for testing*
- *The commands to compile verilog module and show output are written in cpu.sh*
- *make sure you give necessary permisions to cpu.sh*
- *run ./cpu.sh*
- *You can see the outputs on gtkwave clearly*
- *For sorting algorithm , initiallize the data.mem with input array in the following format*

0x00000000 : n (number of elements)
0x00000001 : a1
0x00000002 : a2
.
.
$0x(n)_2$: an

Store a_is in hexadecimal format.

NOTE:

The Sequential architechture of the processor had also been implemented and can be found in 'sequential folder'.
All the modules in sequential architechture were tested and then it was extended to pipelined architechture making the necessary changes.

Thank you!