Report

Atharv Sujlegaonkar

2020102025

Mitul Garg

2020102026

Introduction

I have developed SEQ design based on Y86 ISA using Verilog. Below in the report is the detailed description of how each module is created and working along with diagrams for better understanding. I have also later implemented the processor architecture implementation with 5 stage pipeline (without pipeline hazards).

My design approach: I have design my processor in a modular way, hence I have tested each of my modules separately and later in the end combined them all together to work as the processor!

For each module I have implemented a testbench in order to check the functioning of my processor in each step.

Later in this report I have also taken screenshots of the outputs of each module by running several testing commands like “rrmovq rA rB” and more.

I have also written assembly codes for both bubble sort (sorting algorithm) and HCF of two numbers. The results are shown in gtkwave for the bubble sort algorithm. This can be seen in the results section.

Now I have divided this report into two sections:

- SEQ Design
- Pipelined Design

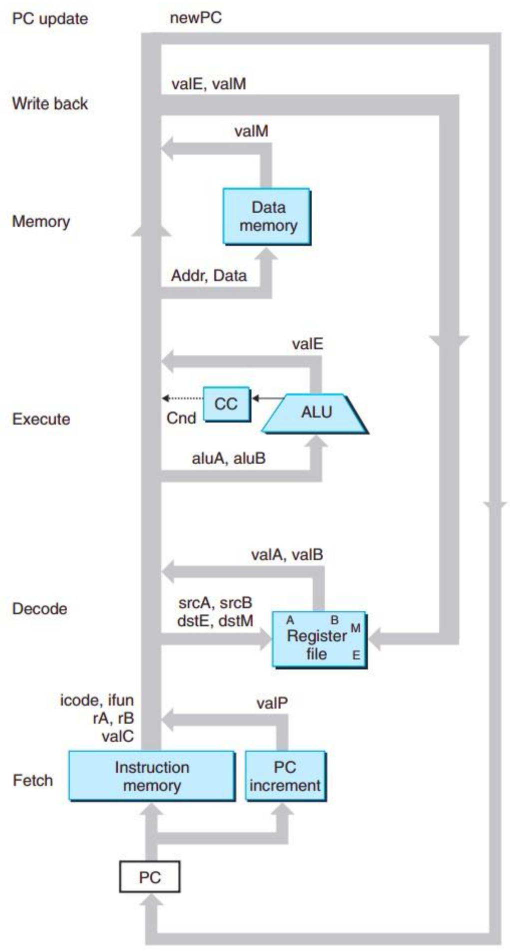
SEQ Design

Module descriptions and architecture diagram

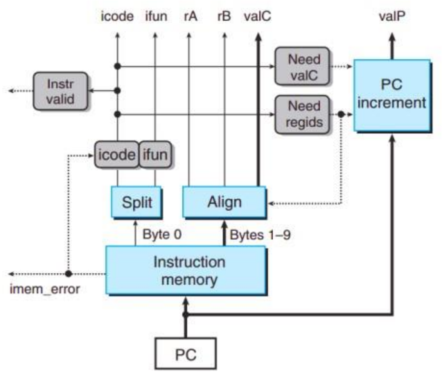
Through this section I will show how the different computation steps in the sequential implementation work for two instructions:

- OPq rA, rB
- rmmovq rA, D(rB)

Abstract view of SEQ implementation:



Fetch Module:



Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.

This unit reads 10 bytes from memory at a time, with the PC serving as the first byte's address (byte 0). This byte is translated as the instruction byte and divided into two 4-bit quantities (by the “Split” unit). From this we get icode and ifun values from which we can predict the next PC increment.

The align module read the other 9 bytes from the encoded instruction and extract valC and rA, rB values.

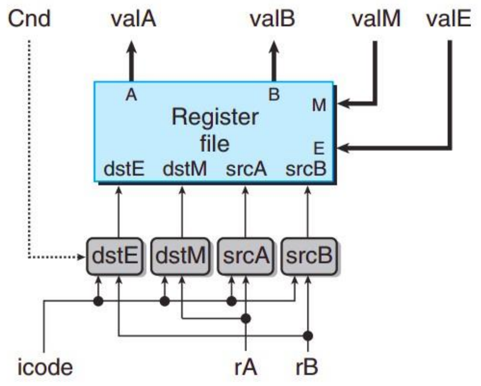
So basically using fetch module we divide the encoded instruction and extract the instruction set from it. We then send these values to the decode block inorder to extract values stored in rA and rB registers.

Here in the diagram you can see three more bit checks:

- Instr_valid: This signal is used to detect an illegal instruction
- Need_valC: Checks for a constant word
- Need_regids: Checks for register specifier byte

Computation	**OPq rA, rB**	**rmmovq rA, D(rB)**
icode, ifun	icode :ifun ← M1[PC]	icode :ifun ← M1[PC]
rA, rB		rA:rB ¬ M 1 [PC+1]
valC	rA :rB ← M1[PC + 1]	valC ¬ M 8 [PC+2]
valP	valP ← PC + 2	valP ¬ PC+10

Decode and Write back Module:



As we can see in the diagram, we input icode, rA and rB values in the decode module. The decode module basically consists of a register file which has 4 ports which supports up to two reads on port A and B and two writes on ports E and M. The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM.

Here in this module I have written many if conditions to extract the values from rA and rB registers in the register file. The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.

We also have two ports coming from the write back stage in the register file, which can change the register values from the write back stage.

Write back:

The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

Outputs of decode module:

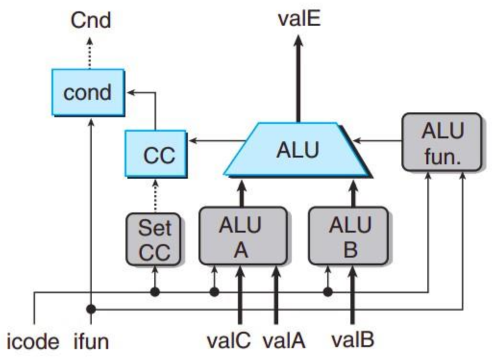
- valA: Value of rA
- valB: Value of rB

We also have two inputs from the write back stage:

- valM: Output value from memory stage
- valE: Output value from the execute stage

Computation	**OPq rA, rB**	**rmmovq rA, D(rB)**
valA, srcA	valA ← R[rA]	valA ← R[rA]
valB, srcB	valB ← R[rB]	valB ← R[rB]

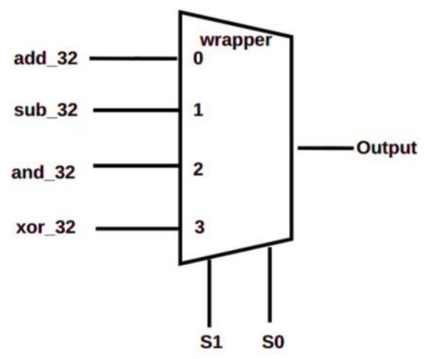
Execute module:



The execute stage uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type.

Here ALU is the core of the execute module.It executes the specified operation for integer operations. It can be used as an adder to compute an incremented or decremented stack pointer, an effective address, or simply to transfer one of its inputs to one of its outputs by inserting zero in other instructions. The three state code bits are stored in the condition code register (CC). The ALU calculates new values for the condition code.

The ALU

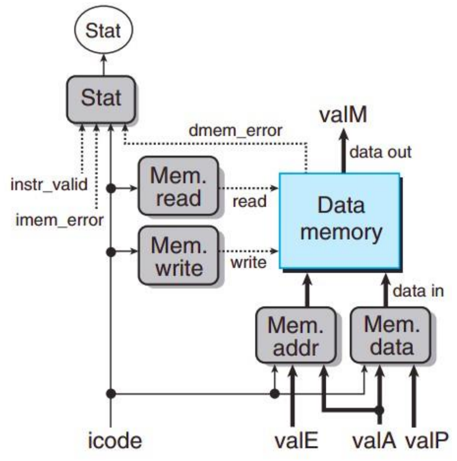


Inbuilt modules Used in the ALU:

Module	Input	Output
and	X,Y (32 bit)	Z = X & Y
or	X,Y (32 bit)	Z = X | Y
xor	X,Y (32 bit)	Z = X ^ Y

When executing a conditional move instruction, the decision as to whether or not to update the destination register is computed based on the condition codes and move condition. Similarly, when executing a jump instruction, the branch signal Cnd is computed based on the condition codes and the jump type.

Memory Module:



Making and implementing this module was one challenging part of this project.

My implementation of memory:

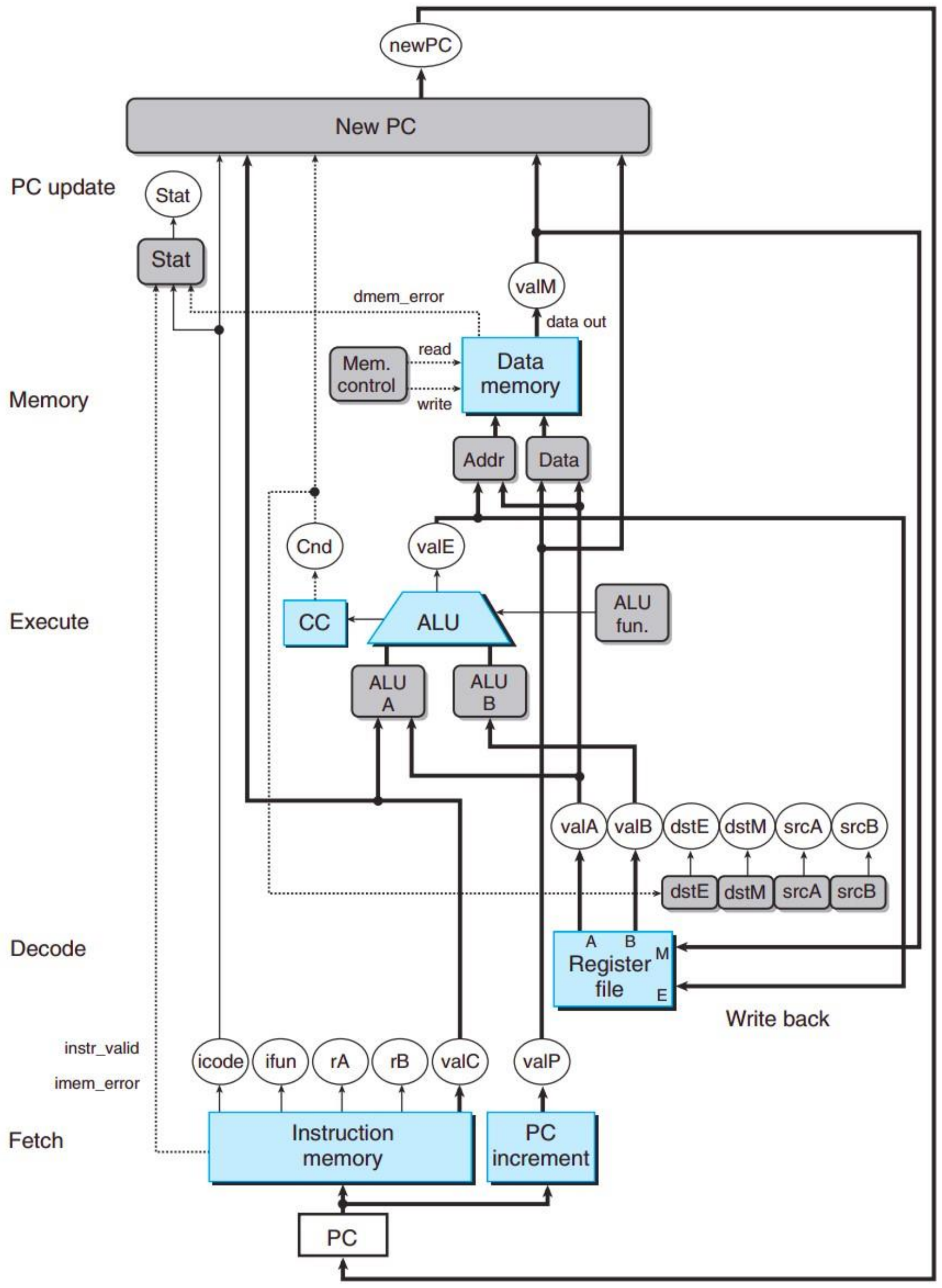
I have implemented the memory using an array of registers for seq design. Here in the array the index refers to the address of the register and the data in the element of the array is value stored at that address.

How does memory work, what does it do:

The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes.

Here two control blocks generate the values for the memory address and the memory input data for write operations. Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM. And for the write operation, the address value is changed accordingly. Later it generates the status code from the values of icode, imem_ error, and instr_valid generated in the fetch stage and the signal dmem_error generated by the data memory. According to the status code the processor either halts or continues processing instructions input in it.

Combined hardware structure of SEQ design implemented:



<u>Instructions supported by my processor</u>

I have implemented all the instructions on my SEQ design **including call and ret functions.**

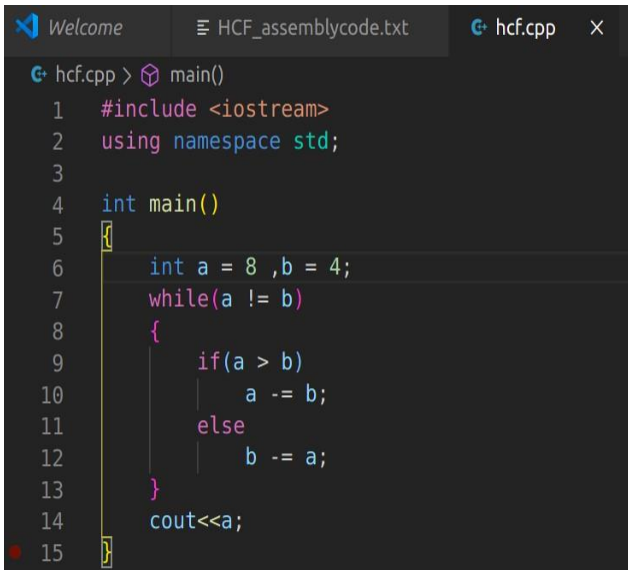
Instruction	**Format**
Halt	halt
No operation	nop
Conditional move	cmovXX rA, rB
Immediate to register	irmovq V, rB
Register to memory	rmmovq rA, D(rB)
Memory to register	mrmovq D(rB), rA
Operation	Opq rA, rB
Jump	jXX Dest
Call	call Dest
Return	ret
Push in stack	pushq rA
Pop from stack	popq rA

Results:

I have written codes for both HCF of two numbers and Bubble sort algorithm.

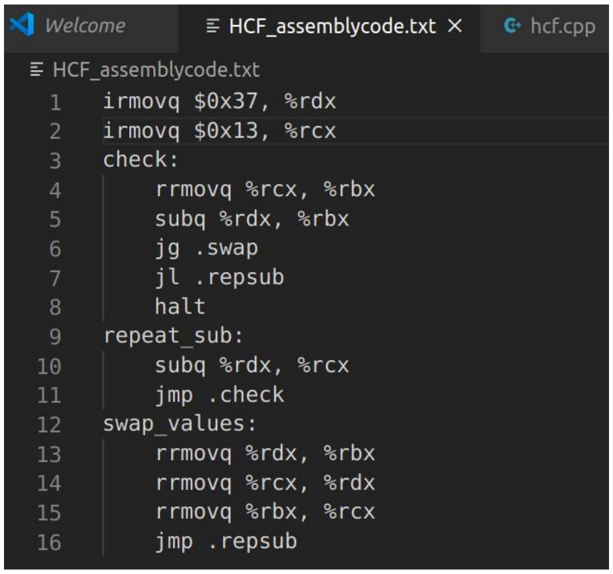
HCF of two numbers:

Code in C++:



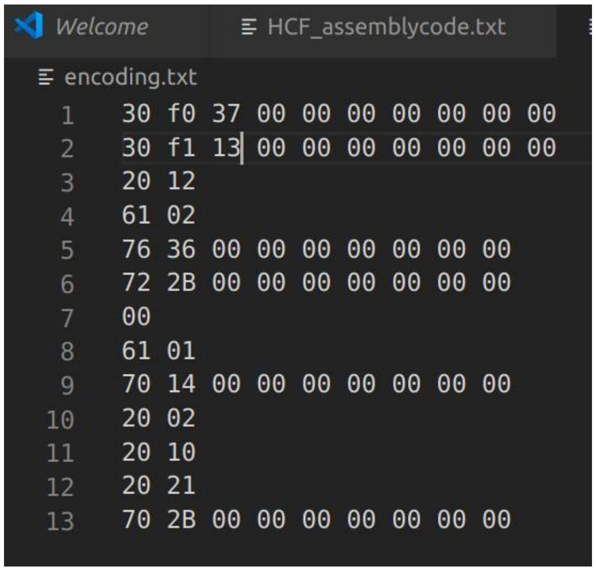
```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 8 ,b = 4;
    while(a != b)
    {
        if(a > b)
            a -= b;
        else
            b -= a;
    }
    cout<<a;
}
```

Assembly Code:



```
irmovq $0x37, %rdx
irmovq $0x13, %rcx
check:
    rrmovq %rcx, %rbx
    subq %rdx, %rbx
    jg .swap
    jl .repsub
    halt
repeat_sub:
    subq %rdx, %rcx
    jmp .check
swap_values:
    rrmovq %rdx, %rbx
    rrmovq %rcx, %rdx
    rrmovq %rbx, %rcx
    jmp .repsub
```

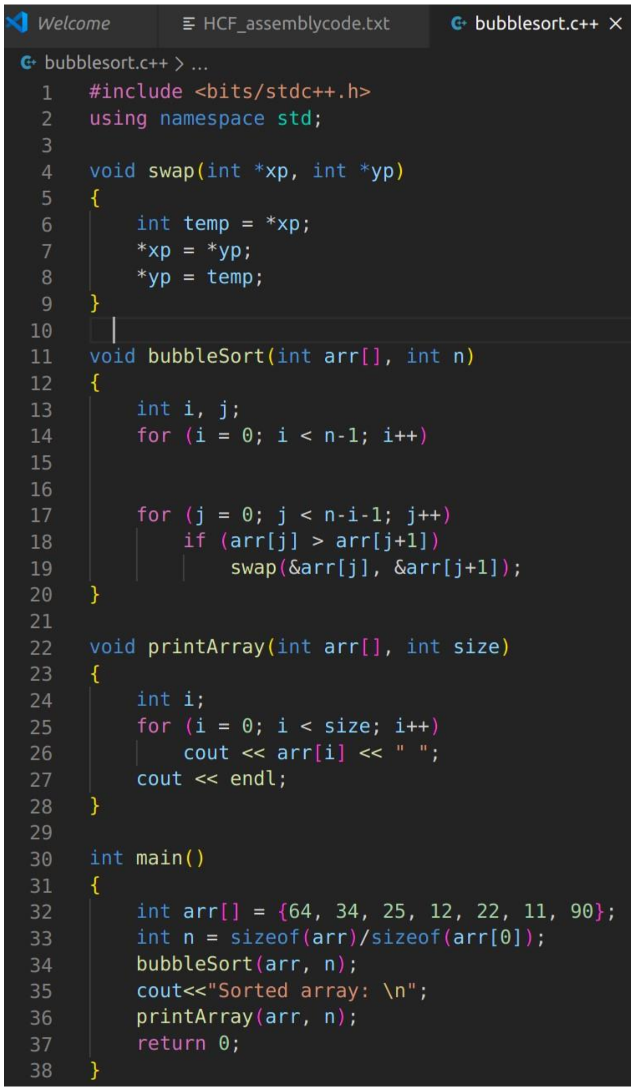
Encoded Code:



```
30 f0 37 00 00 00 00 00 00 00
30 f1 13 00 00 00 00 00 00 00
20 12
61 02
76 36 00 00 00 00 00 00 00
72 2B 00 00 00 00 00 00 00
00
61 01
70 14 00 00 00 00 00 00 00
20 02
20 10
20 21
70 2B 00 00 00 00 00 00 00
```

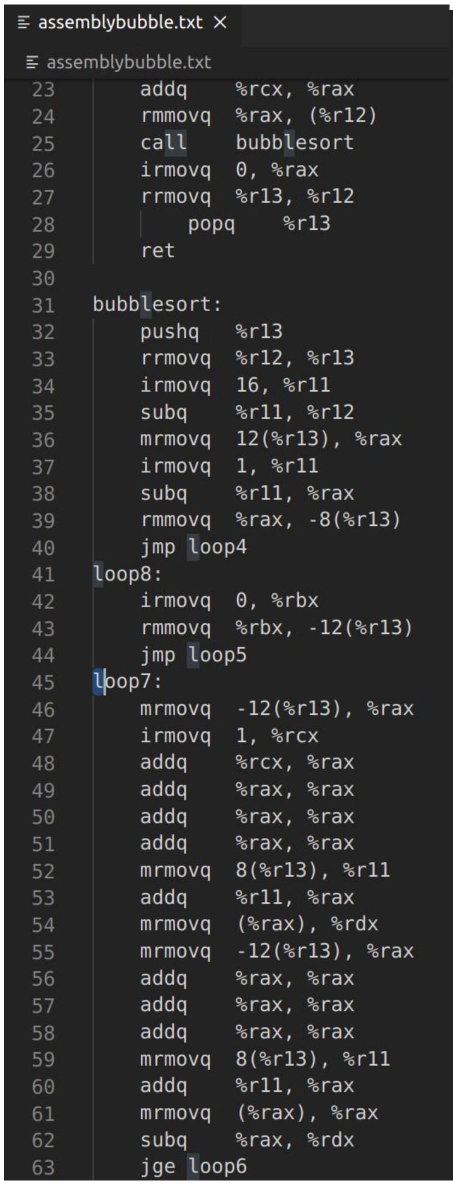
Bubble sort algorithm:

Code in C++:



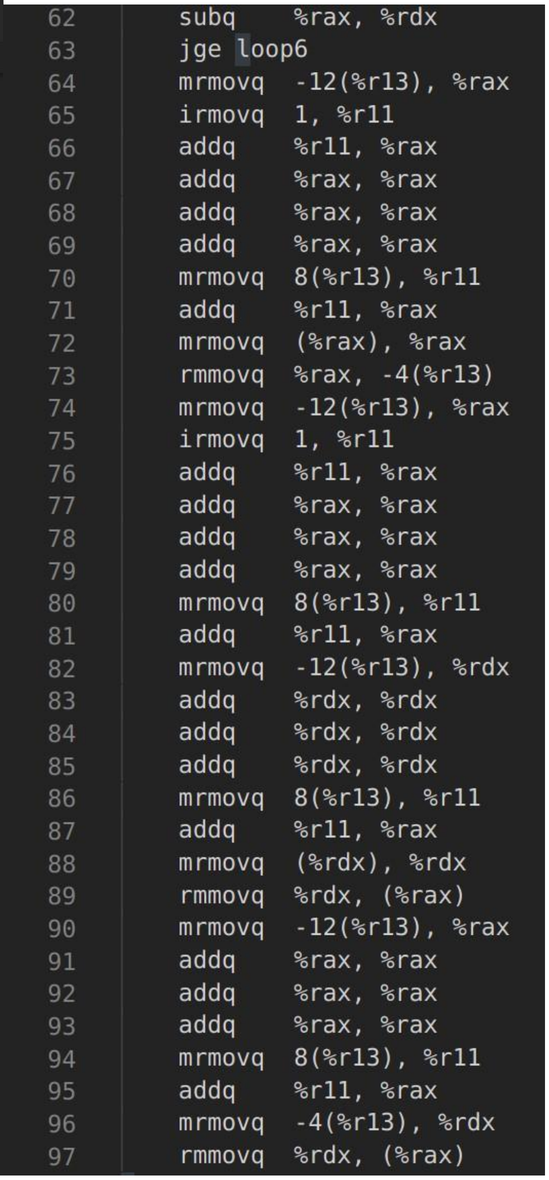
```cpp
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)


    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Assembly code:



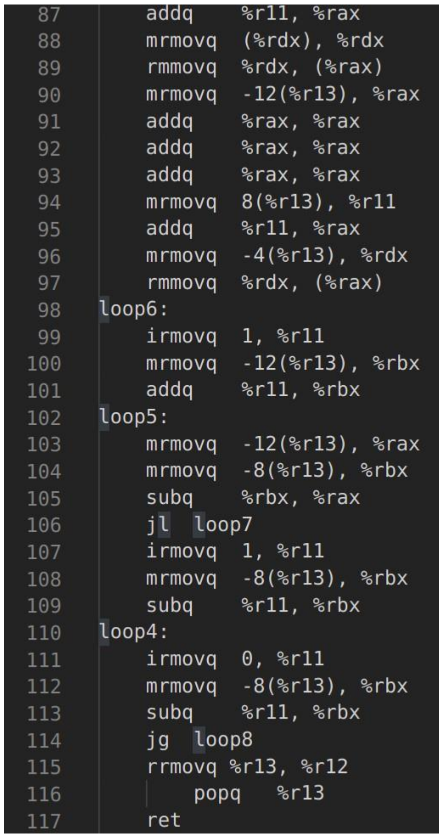
assemblybubble.txt

```
    addq    %rcx, %rax
    rmmovq  %rax, (%r12)
    call    bubblesort
    irmovq  0, %rax
    rrmovq  %r13, %r12
        popq    %r13
    ret

bubblesort:
    pushq   %r13
    rrmovq  %r12, %r13
    irmovq  16, %r11
    subq    %r11, %r12
    mrmovq  12(%r13), %rax
    irmovq  1, %r11
    subq    %r11, %rax
    rmmovq  %rax, -8(%r13)
    jmp loop4
loop8:
    irmovq  0, %rbx
    rmmovq  %rbx, -12(%r13)
    jmp loop5
loop7:
    mrmovq  -12(%r13), %rax
    irmovq  1, %rcx
    addq    %rcx, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  (%rax), %rdx
    mrmovq  -12(%r13), %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  (%rax), %rax
    subq    %rax, %rdx
    jge loop6
```



```
    subq    %rax, %rdx
    jge loop6
```

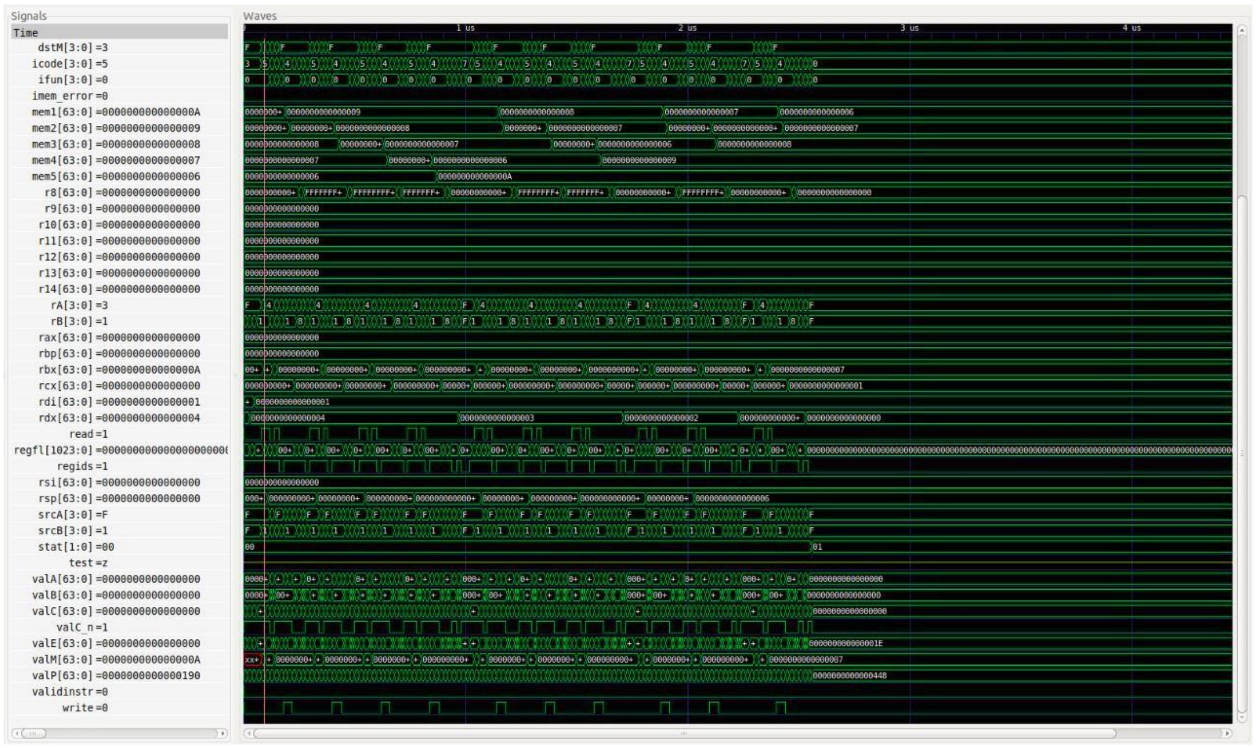
```
    mrmovq  -12(%r13), %rax
    irmovq  1, %r11
    addq    %r11, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  (%rax), %rax
    rmmovq  %rax, -4(%r13)
    mrmovq  -12(%r13), %rax
    irmovq  1, %r11
    addq    %r11, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  -12(%r13), %rdx
    addq    %rdx, %rdx
    addq    %rdx, %rdx
    addq    %rdx, %rdx
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  (%rdx), %rdx
    rmmovq  %rdx, (%rax)
    mrmovq  -12(%r13), %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  -4(%r13), %rdx
    rmmovq  %rdx, (%rax)
```



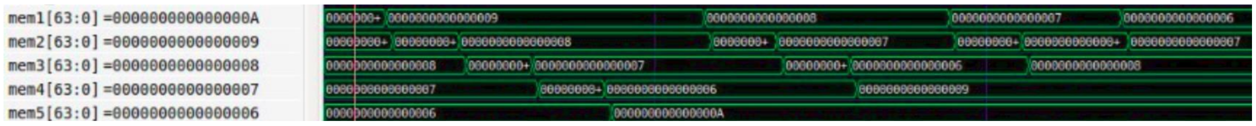
```
    addq    %r11, %rax
    mrmovq  (%rdx), %rdx
    rmmovq  %rdx, (%rax)
    mrmovq  -12(%r13), %rax
    addq    %rax, %rax
    addq    %rax, %rax
    addq    %rax, %rax
    mrmovq  8(%r13), %r11
    addq    %r11, %rax
    mrmovq  -4(%r13), %rdx
    rmmovq  %rdx, (%rax)
loop6:
    irmovq  1, %r11
    mrmovq  -12(%r13), %rbx
    addq    %r11, %rbx
loop5:
    mrmovq  -12(%r13), %rax
    mrmovq  -8(%r13), %rbx
    subq    %rbx, %rax
    jl  loop7
    irmovq  1, %r11
    mrmovq  -8(%r13), %rbx
    subq    %r11, %rbx
loop4:
    irmovq  0, %r11
    mrmovq  -8(%r13), %rbx
    subq    %r11, %rbx
    jg  loop8
    rrmovq %r13, %r12
        popq   %r13
    ret
```

GTKWAVE result for bubble sort:

Please zoom in to checks waves and output values


```
Signals
Time
      dstM[3:0] =3
     icode[3:0] =5
      ifun[3:0] =0
     imem_error =0
     mem1[63:0] =000000000000000A
     mem2[63:0] =0000000000000009
     mem3[63:0] =0000000000000008
     mem4[63:0] =0000000000000007
     mem5[63:0] =0000000000000006
       r8[63:0] =0000000000000000
       r9[63:0] =0000000000000000
      r10[63:0] =0000000000000000
      r11[63:0] =0000000000000000
      r12[63:0] =0000000000000000
      r13[63:0] =0000000000000000
      r14[63:0] =0000000000000000
        rA[3:0] =3
        rB[3:0] =1
      rax[63:0] =0000000000000000
      rbp[63:0] =0000000000000000
      rbx[63:0] =000000000000000A
      rcx[63:0] =0000000000000000
      rdi[63:0] =0000000000000001
      rdx[63:0] =0000000000000004
           read =1
 regfl[1023:0] =0000000000000000000000
         regids =1
      rsi[63:0] =0000000000000000
      rsp[63:0] =0000000000000000
      srcA[3:0] =F
      srcB[3:0] =1
      stat[1:0] =00
           test =z
     valA[63:0] =0000000000000000
     valB[63:0] =0000000000000000
     valC[63:0] =0000000000000000
         valC_n =1
     valE[63:0] =0000000000000000
     valM[63:0] =000000000000000A
     valP[63:0] =0000000000000190
     validinstr =0
          write =0
```

Here please check the mem1, mem2, mem3, mem4, mem5 values.


```
mem1[63:0] =000000000000000A
mem2[63:0] =0000000000000009
mem3[63:0] =0000000000000008
mem4[63:0] =0000000000000007
mem5[63:0] =0000000000000006
```

Initially as we can see I have put in these values:

Mem1 = A

Mem2 = 9

Mem3 = 8

Mem4 = 7

Mem5 = 6

But with few clock cycles you can see these values shifting and changing from one mem to other.

And after final clock cycle we can clearly see in the output that the mems are sorted in ascending order:

Mem1 = 6

Mem2 = 7

Mem3 = 8

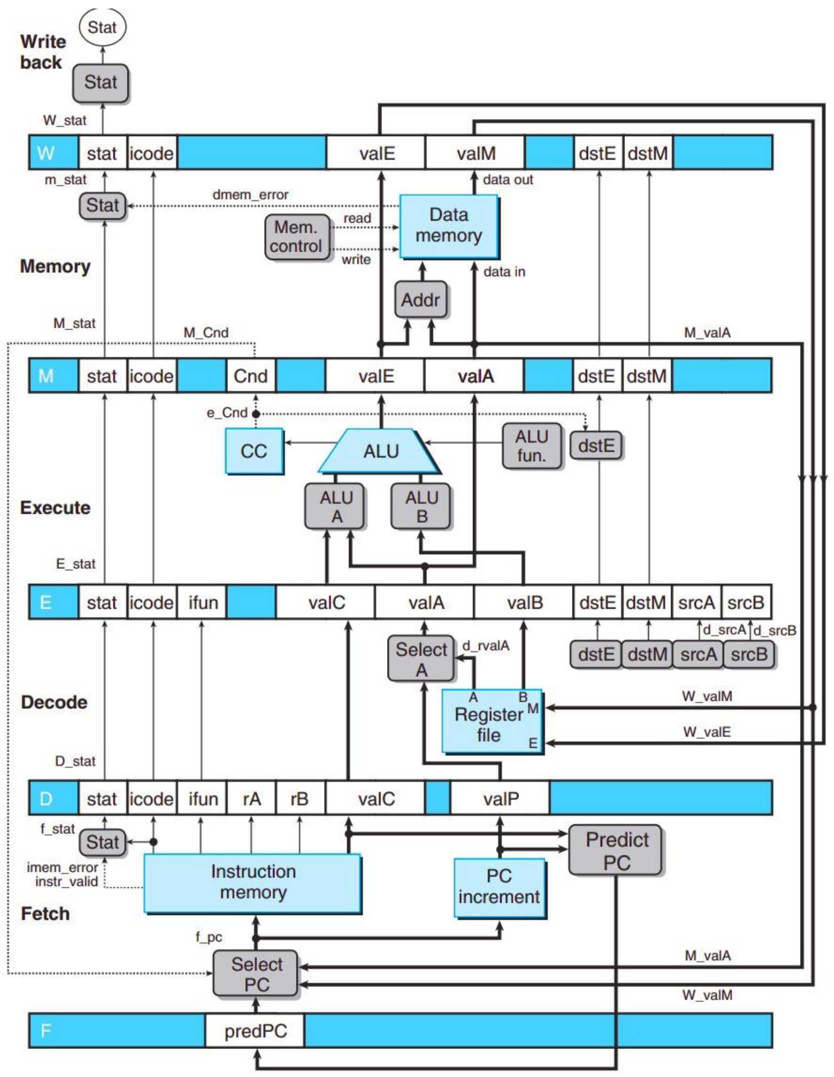
Mem4 = 9

Mem5 = A

Instructions to run your code:

I have made a testbench for each and every module, so please include all the folders in one folder as a whole and run the test bench. Testbench names matches with the module you want to test it on.

Pipeline Design



I have inserted pipeline registers between the stages of SEQ+ and rearrange signals somewhat, yielding the PIPE− processor.

The pipeline registers are labeled as follows:

- F holds a predicted value of the program counter, as will be discussed shortly.
- D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

For this project, I have implemented only the registers in pipeline. I haven't implemented control logic and the overall cpu. Core modules made and their description:

Fetch_reg.v:

Function: Temporarily store the PC value predicted in the previous stage for use by the select_pc module.

Decode_reg.v

Function: Temporarily store the instruction information parsed by the fetch module.

Execute_reg.v

Function: Temporarily store the instruction information of the previous stage.

Alu_args.v

Function: Determine the value and operation type sent to the arithmetic logic operation module according to the instruction code and function code in the execution register.

Mem_reg.v

Function: Temporarily store the instruction information of the previous stage

Registers.v

Function: general register module of the processor.

Define.v

Global macro definition