Intro to Processor Architecture

5-Staged Pipelined Y86 Processor

Report

Kartik Agarwal
2018102017

1) Fetch Logic and PC Selection

There are four modules in this stage:

A. Instruction Memory
B. Split
C. Align
D. PC increment

A) Instruction Memory:

The aim of this block is to read the instructions from the memory. This block then divides the 10 bytes input into two sets, sending the 1st byte (byte 0) into the “Split” module and the remaining 9 bytes (bytes 1-9) into the “Align” module as the output.

B) Split:

The first byte through the “Instruction Memory” is taken as input in this module. This input instruction byte is then split into two 4-bit quantities and passed into the output sub-modules, “icode” and “ifun”. These two control logic blocks are used to compute the instruction and function codes, and also to check if the values are either read from the memory or if the instruction address is not valid (as shown by signal “imem_error”), that is if the values correspond to the nop instruction. The “icode” values can be useful to compute three 1-bit signals.

C) Align:

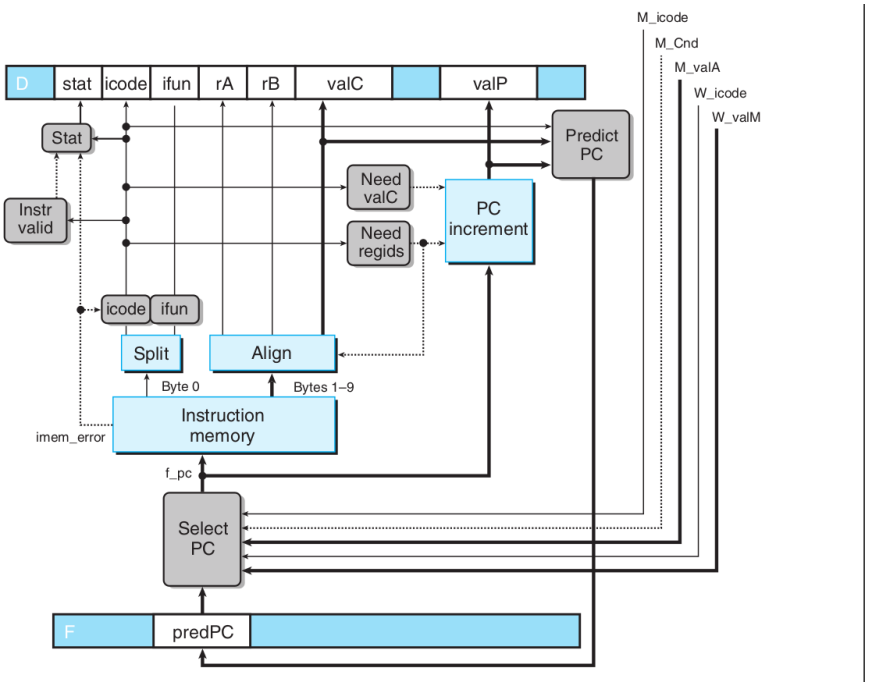
The input 9 bytes taken from the instruction memory encodes the specific combination of the register specifier bytes and the constant word after processing through this module. 1st byte is split into register specifiers rA and rB if the signal “need_regids” is true (valued 1). Else, both of the register specifiers are set to 0xF (RNONE), showing that no registers are specified by the instruction. Thus, we can say that signals rA and rB are either encoded registers, that we want to access, and if it is “RNONE”, it shows that register access is not required. Also, this module generates the constant word “valC”. This can be either byte 1-8 or bytes 2-9 (that is the set of 8 contiguous bytes), and is decided based on the value of the signal “need_regids”.

D) PC Increment:

The “PC Increment” module aims for generating the signal “valP” based on the present values of the PC, and the signals “need_regids” and “need_valC”. The value generated is ***”p+1+r+8i”,*** where p is the PC value, r and i are “need_regids” and “need_valC” signals values respectively.

The PC selection logic chooses based on the three program counter sources. If a mispredicted branch enters the memory stage, the value of valP for this instruction is read from pipeline register M (signal M_valA). When a ret instruction enters the write-back stage,

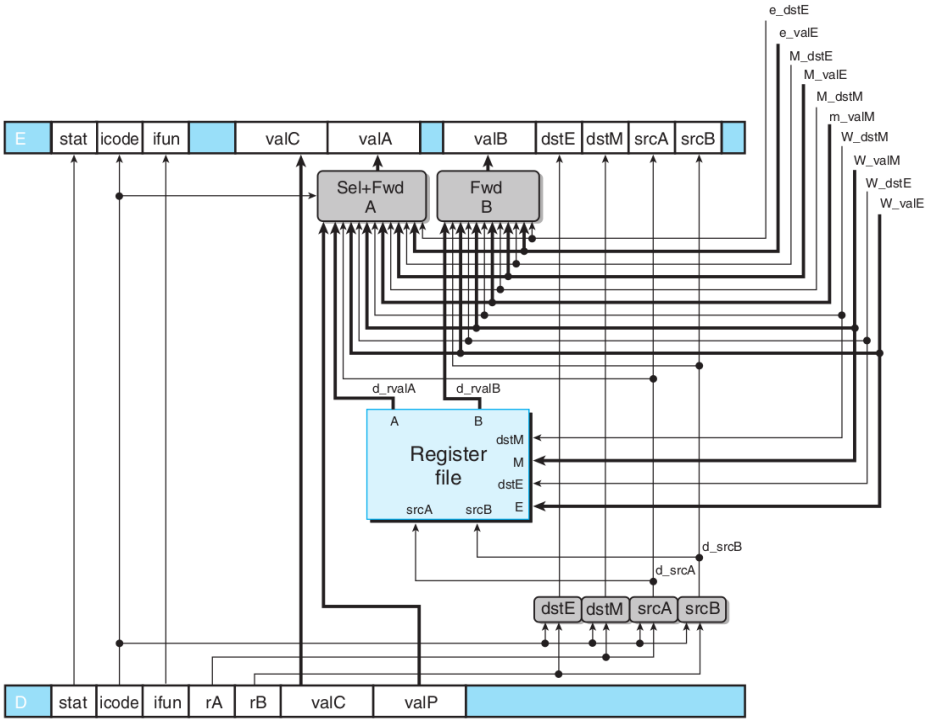
the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC). The PC prediction logic chooses valC if it is either a call or a jump, otherwise, it's valP.



The control logic blocks labeled as "instr_valid," "need_regids," and "need_valC" are similar to that of SEQ. Also, we can test for a memory error because of out-of-range instruction addresses, in the fetch stage, and the detection of an illegal or halt instruction is also done here. And the detection of an invalid data address must be deferred to the memory stage.

2) Decode Stage

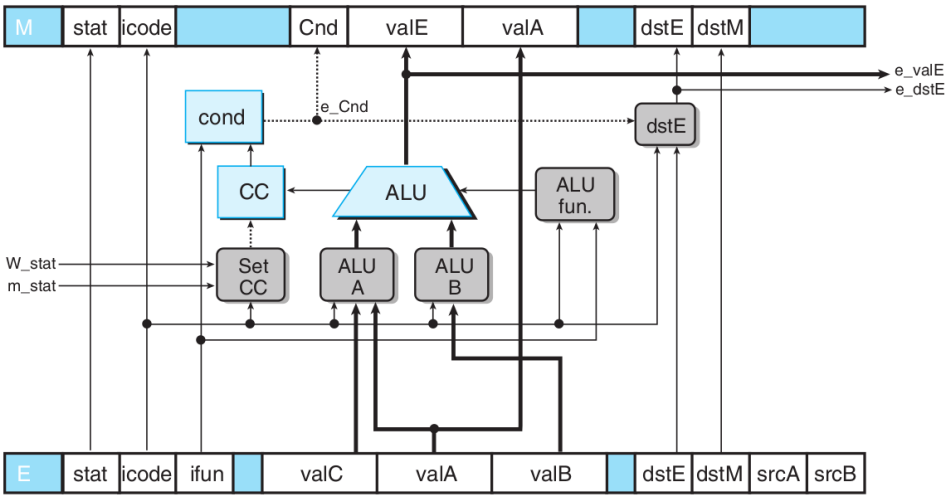
The two stages Decode and Write-back are combined together because they both access the register file. The register file consists of four ports. The two ports A and B support simultaneous reads and the other two ports E and M support simultaneous writes. Read ports address is defined as srcA and srcB whereas write ports address is defined as dstE and dstM . Each port consists of an address connection (register id) and a data connection (set of 64 wires which serves either as input or output word of the register file). The address port has a special identifier 0xF (RNONE) which indicates that no register should be accessed. The write-back stage i.e. signals W_dstE and W_dstM supply register IDs because the writes to occur to the destination registers which are specified in the write-back stage. The complexity of this stage is associated with the forwarding logic. The block "Sel+Fwd A" can perform two roles. The first role is to merge the valP and valA signal for the upcoming stages which helps in reducing the number of states in the pipeline. The second role is to implement the forwarding logic for source operand valA. In later stages, only the jump and call instructions require the value of valP, so these signals do not require reading the value from port A. This is controlled by the signal called icode.



3) Execution Stage

The execute stage includes the arithmetic/logic unit (ALU). This part performs the operation add, subtract, and, or exclusive-or on inputs aluA and aluB based on the setting of the alucom signal. The data and control signals are generated by three control blocks.

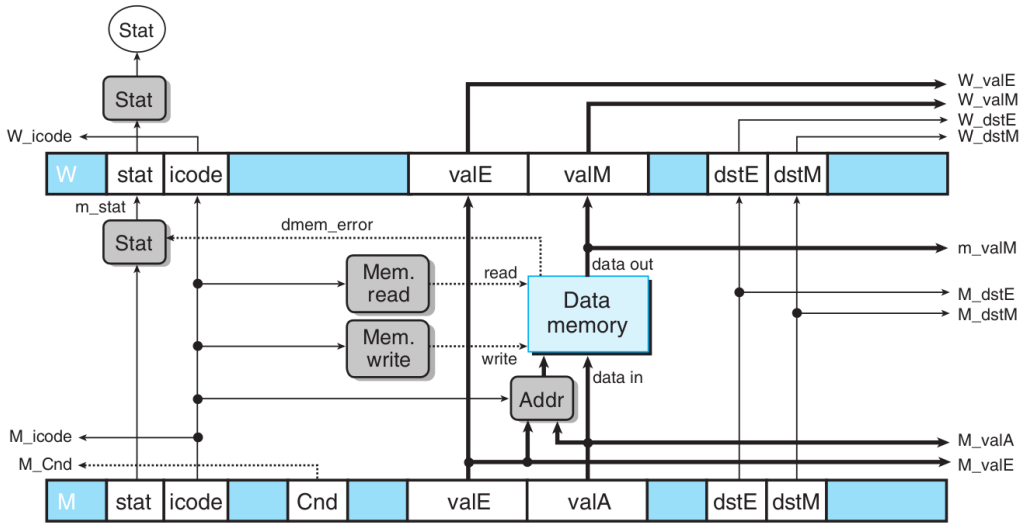
aluB is listed before the aluA in the operands to ensure that the subq instruction subtracts the valA from the valB and not vice-versa. We can also see that the value of aluA can be any of these valA, valc or either -8 or +8, respective to instruction type. By looking at the performed operations we can deduce that it mostly works as an adder. For the OPq instructions, we want it to use the operation encoded in the ifun field of the instruction. The execute stage also includes the condition code register. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an OPq instruction is executed.



The hardware unit labeled “cond” uses a combination of the condition codes and the function code to determine whether a conditional branch or data transfer should take place or not. It generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches. We can see signals e_valE and e_dstE directed towards the decode stage as one of the forwarding sources. One difference from sequential in pipelined 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. These signals can be used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

4) Memory Stage

The memory stage has the task of either reading or writing program data. Two control blocks generate the values for the memory address and the memory input data. 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. The address for memory reads and writes is always valE or valA. We want to set the control signal mem_read only for instructions that read data from memory.



A final function for the memory stage is to compute the status code Stat resulting from the instruction execution according to the values of icode, imem_error, and instr_valid generated in the fetch stage and the signal dmem_error generated by the data memory.

5) Pipeline Control Logic

We are now ready to complete our design for PIPE by creating the pipeline control logic. This logic must handle the following four control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

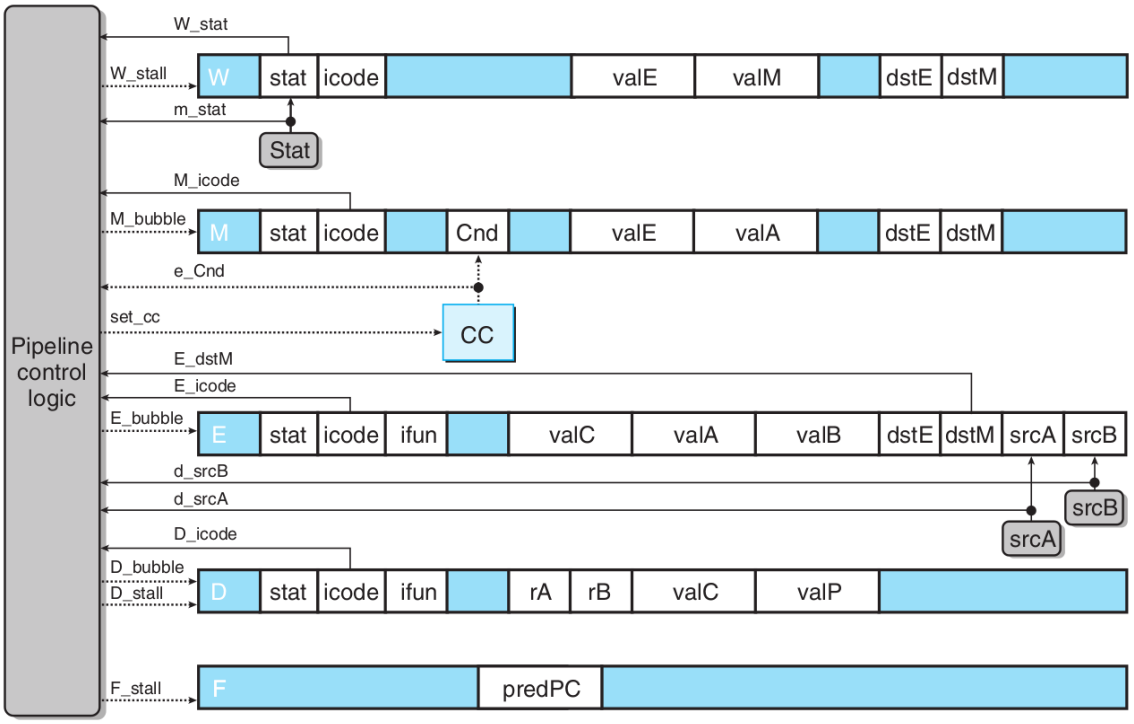
Load/use hazard: The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.

Processing ret: The pipeline must stall until the ret instruction reaches the write-back stage.

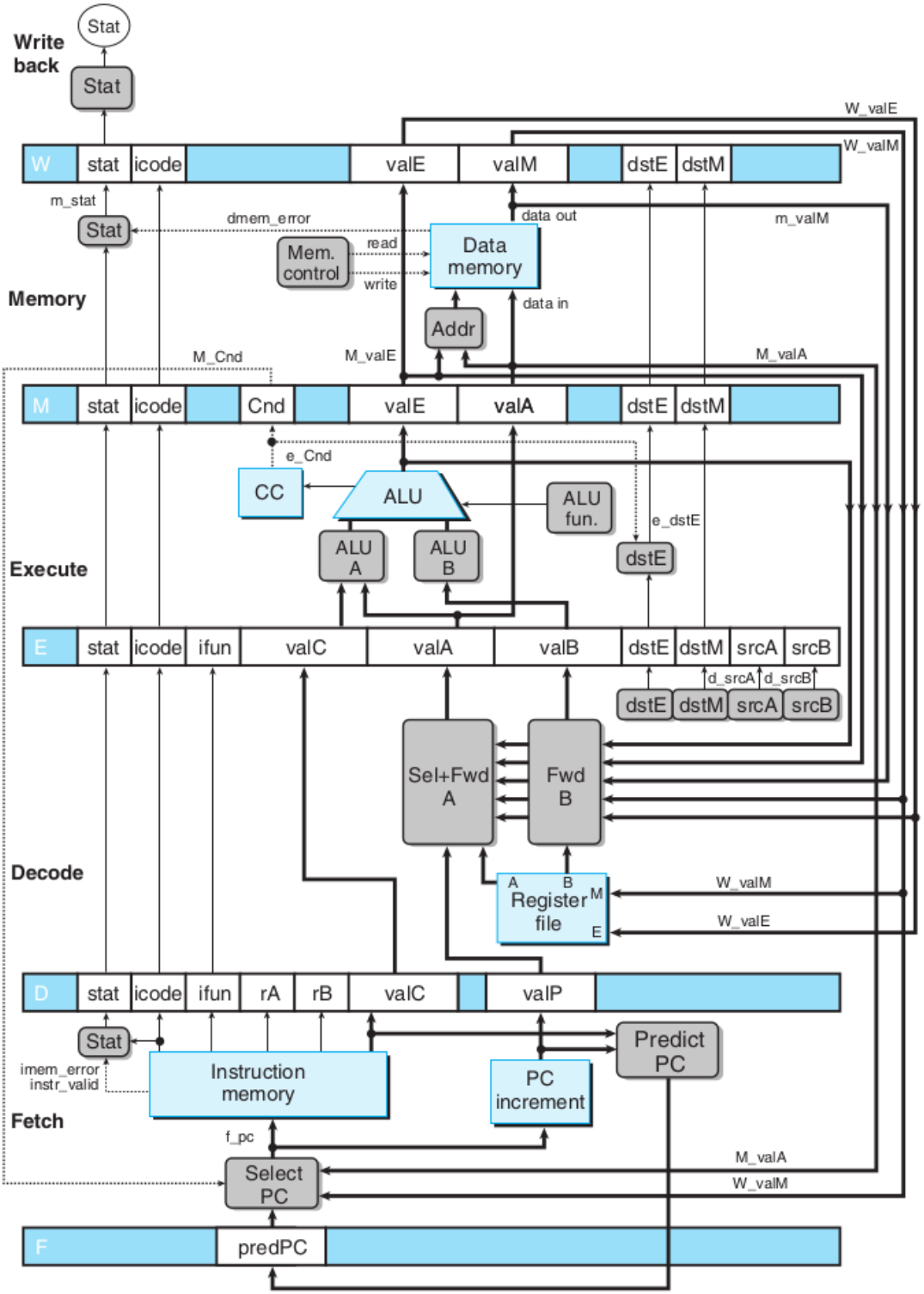
Mispredicted branches**:** By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline.

These instructions must be stopped, 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.



Complete Pipelined Processor



Instructions Supported By Processor:

- halt
- nop
- rrmovq
- irmovq
- rmmovq
- mrmovq
- OPq
- jXX
- cmovXX

- call
- ret
- pushq
- popq

HCF Code in Assembly Language:

```
rmovq 58, %rax
   irmovq 98, %rbx
   clock:
   rrmovq %rbx, %rcx
   subq %rax, %rcx
   jg .swap
   jl .repsub
   halt

   repsub:
   subq %rax, %rbx
   jmp .check

   swap:
   rrmovq %rax, %rcx
   rrmovq %rbx, %rax
   rrmovq %rbx, %rbx
   jmp .repsub
```

Instructions to run code:

1. Download the codes folder from the git to your machine.
2. To install verilog: (in Terminal)

   sudo apt-get install verilog
3. To install Gtkwave:

   sudo apt-get install gtkwave
4. To compile the code:

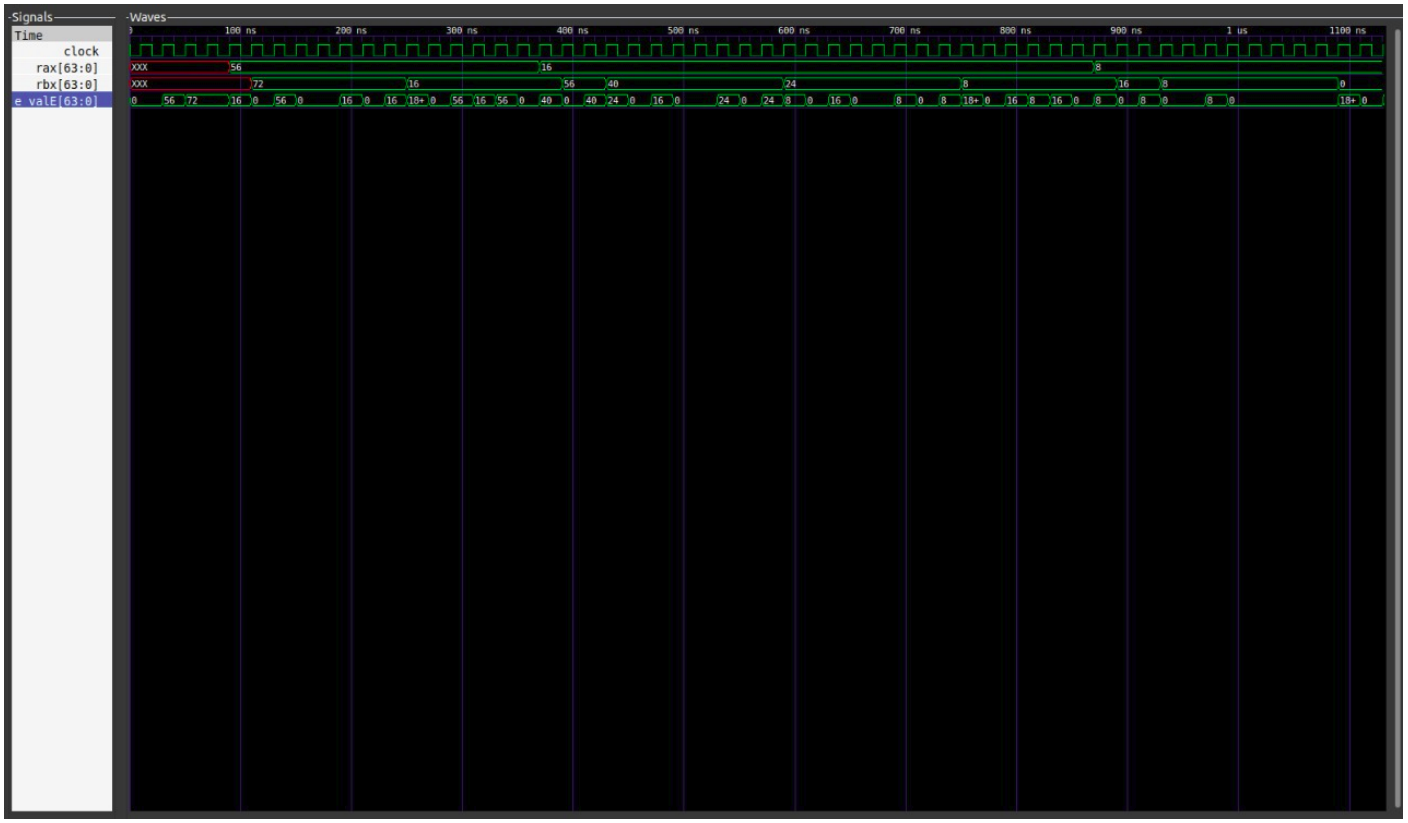
   iverilog main_temp.v main_temp_tb.v
5. To run:

   ./a.out
6. To show output in gtkwave:

   gtkwave main_temp_tb.vcd

GTKWave Output:

rax and rbx are the first and second registers respectively shown in the GTK wave screenshot.


Signals
Waves
Time
clock
rax[63:0]
rbx[63:0]
e valE[63:0]
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1 us
1100 ns