IPA Project Report
Y86-64 Sequential and Pipelined implementation

GROUP- 42: ARCHITECTS

KESHAV AGARWAL – 2020102048

MAULESH GANDHI – 2020112009

Index

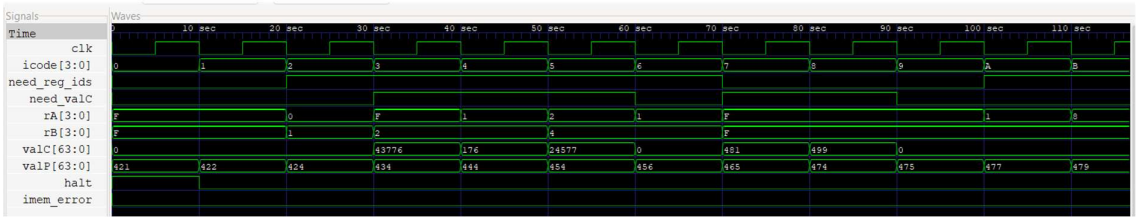
Sequential Implementation:

\Fetch:

We defined a 1kb instruction memory as a register array named *instr_mem*. We are initialising this memory with the instruction codes we want to test. Based on the initialised value of the program counter *PC*, we read the first 10 bytes from *PC* to a register *instr*. If the value of *PC* lies outside instruction memory, we assign *imem_error* as 1. Based on the read values of *icode* and *ifun* which will be present in the first byte, we assign values to *rA*, *rB*, *valC*, *valP*, *halt* and *instr_valid* from *instr*.

* Y86-64 instructions give valC in order of least significant byte first. But for this project we are giving it in form of most significant byte first.

GTKwave simulation of a testbench

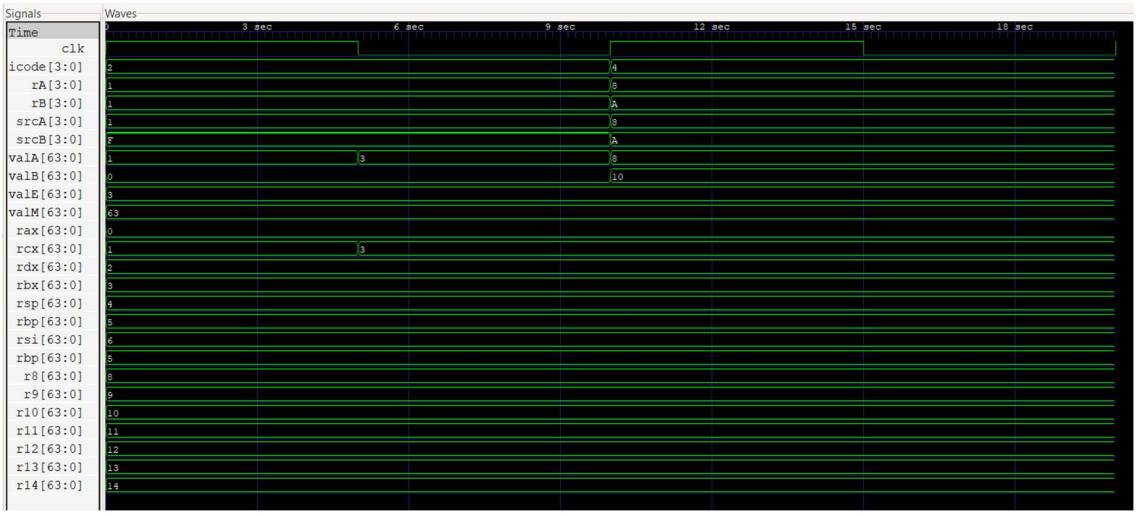


Decode_WriteBack:

Decode part is supposed to give the value stored in the given registers rA and rB as valA and valB. icode is used here to determine whether rA or rB are required for the given operation. if they are not required then we make srcA or srcB into 4'hf accordingly which were otherwise rA and rB. Also, having 4'hf in rA or rB means that we don't have to do any processing to get the new value of valA or valB accordingly.

Writeback part is supposed to write the value of valE or valM in rA, rB, or rsp based on the icode.

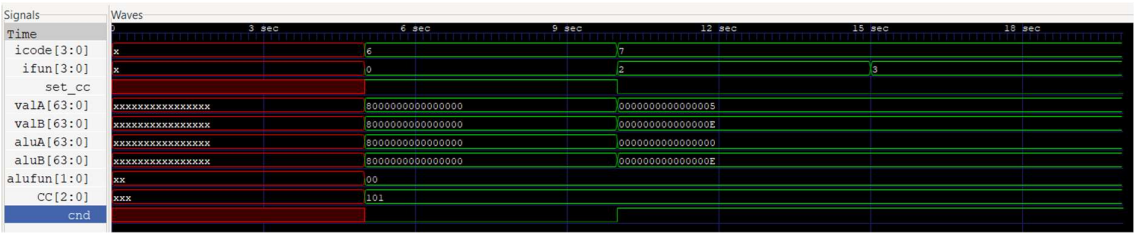
GTKwave simulation of a testbench



Execute:

The purpose of the Execute module is to assign value to valE based on the value of inputs icode, ifun, valA, valC and valC. The execute module also includes the ALU module which allows us to add, and, subtract or xor two 64-bit inputs. A 3 bit register CC is set whenever an OPq instruction is carried out. For determining value of Cnd in case of cmovxx or jxx instructions, we use this CC register.

GTKwave simulation of a testbench

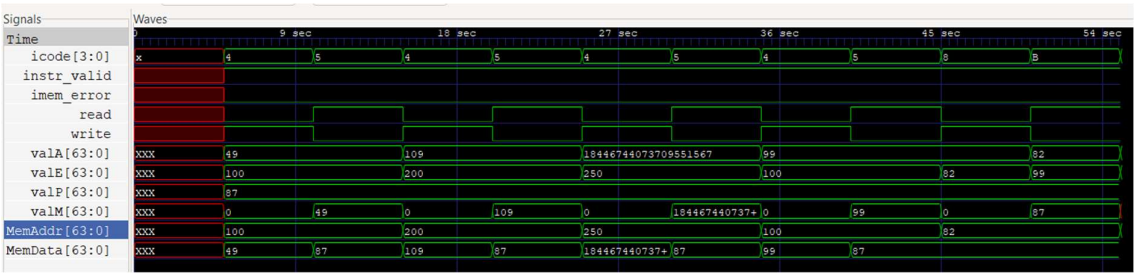


Memory:

The memory stage is responsible for reading and writing memory. Declared data memory as a memory level register array. Only here you can access the data storage you need.

You can decide to read or write based on icode. The data address is calculated using icode, valE, valA and the data The input is calculated using icode, valA, and valP. For rmmovq, call and pushq, you need to write to memory. For mrmovq, ret, popq, it needs to be read from memory.

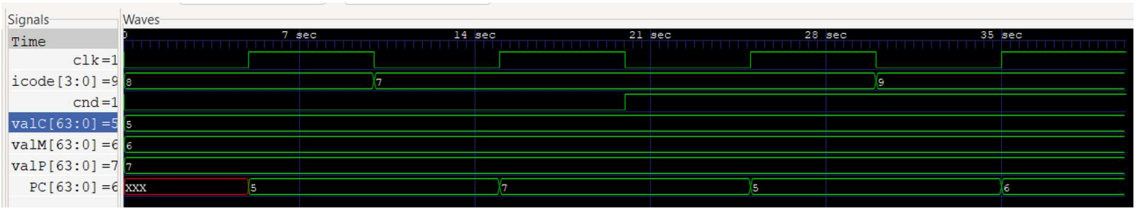
GTKwave simulation of a testbench



PC_Update:

The role of the PC_Update module is to choose the correct value of next PC to be read in fetch. It chooses between inputs valP, valC and valM based on the value of icode and cnd.

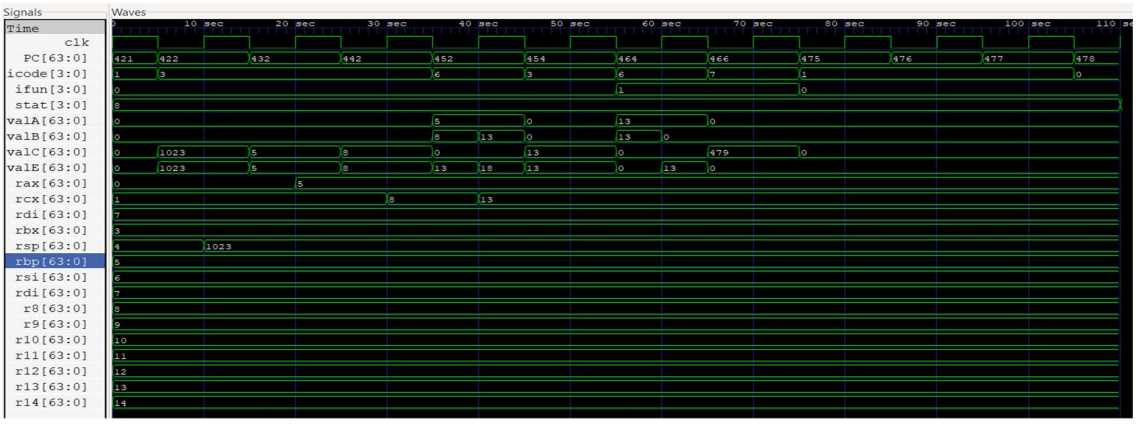
GTKwave simulation of a testbench



Processor:

The processor module is the wrapper module including all the previous modules.

GTKwave simulation of a testbench



Supported Features:

1. halt
2. nop
3. cmovXX
4. irmovq
5. rmmovq
6. mrmovq
7. OPq
8. jXX
9. call
10. ret
11. Pushq
12. Popq

Problems Faced:

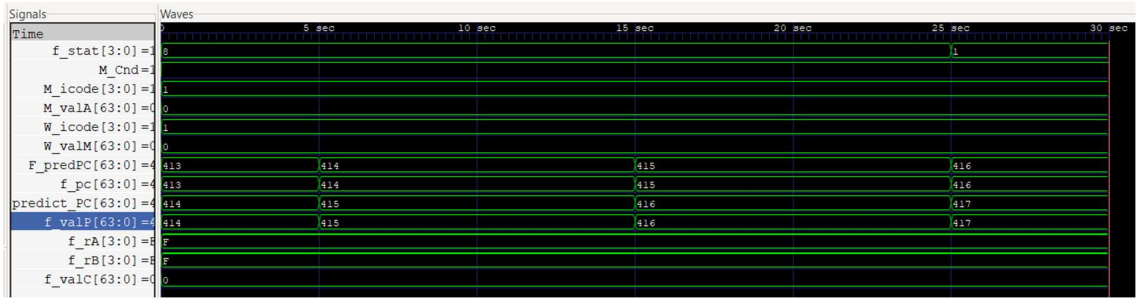
1. We had to check AOK only at edges since while taking jump, instr_valid was becoming 0 for some miniscule time period and the process was getting halted.
2. Call and Ret functions work fine when operated normally, but when pop and push are used inside them, the code is not able to reassign the correct value on returning.

Pipelined Implementation:

Fetch:

This is the first stage of the pipelined implementation containing Fetch_Pipe as main logic module. Here, we first use register F_predPC , which is stored in another module named F_regs,to store the value of predicted PC based on the previous PC and icode and considering all Cnd=1, that is, considering all conditional jumps taken. After this level, we have normal fetch implementation where it takes the 10 bytes from instruction memory and give the icode, ifun, rA, rB, valC and valP where each of these have prefix f_ for this stage to differentiate them from the same terms of other stages.We pass these on to the registers of Decode level. We also have a F_stall in the F registers from pipeline control logic which tells if we have to stall the registers.

GTKwave simulation of a testbench



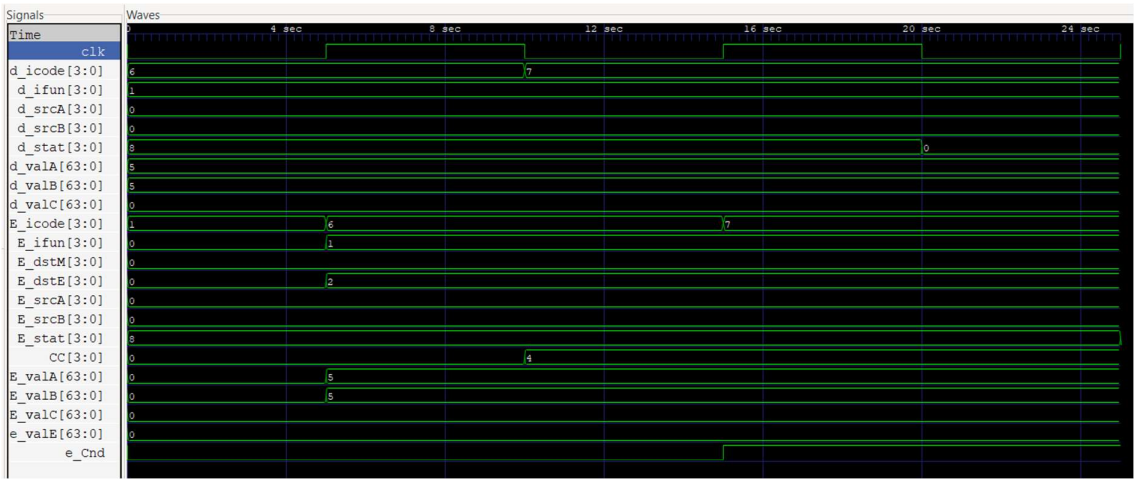
Decode:

This contains two modules D_regs and Decode_WriteBack_Pipe. The first module contains the vregisters which store the values from the previous cycle for this clock cycle. These values have D_ as their prefix and then when they are given as output in Decode_WriteBack_Pipe, they have d_ as prefix.We are implementing the D_stall and D_bubble commands from PCL in the D_regs file only. The Decode functionality is same as in SEQ. The additional feature hers is the data forwarding logic which helps us counter data hazards.

Execute :

This contains only one module named Execute_Pipe which contains both its register files and its combinational logic.We also e=implement E_bubble from PCL here as well. The other execute functionality are same as in SEQ just the outputs of the registers hav prefix E_ and outputs of this module have prefix e_.

GTKwave simulation of a testbench



Memory:

This stage contains one individual module and one part in the wrapper module processor (also in a separate file called M_regs). Its register file is stored in the wrapper module and we bubble implementation in processor as well for this. The combinational logic is same as SEQ. The outputs of the registers have the prefix M_ and the combination logic output have prefix f_. This module mainly interacts with the data memory by reading from and writing to it.

Writeback:

This stage contains two parts which are stored one individual combinational logic module and other register files in Processor (also in a separate file called W_regs for its testbench). It takes stall from PCL and is implemented in the wrapper module. It's registers output has prefix W_ and since its combinational loic gives no output we don't need to worry about them, Here the output of the register files goes to the registers in the Decode stage to get written into. So, it basically sends the data to the specified register and doesn't send if that register is "4'hf".

PCL:

This is the Pipeline Control Logic Module, which tells other modules when they have to stall or bubble based on instructions in all the modules. The main instructions are F_stall, D_stall, D_bubble and E_bubble.

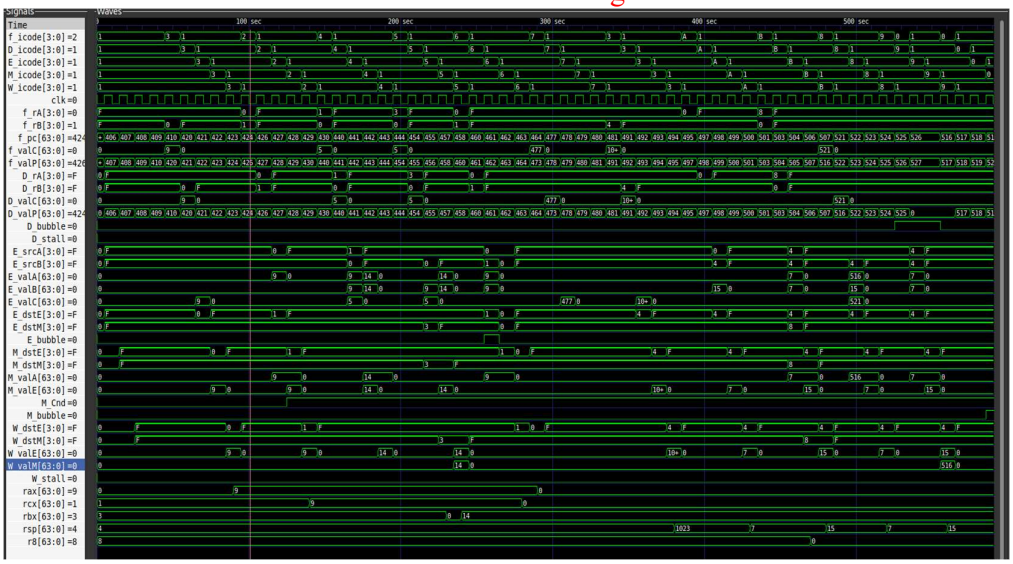
1. F_stall is 1 when we try to access memory locations before they are written, that is, the Load/use hazard or when there is ret in icode in Decode, Execute or Memory stage.
2. D_stall is active only in Load/use hazard

3. D_bubble is active for mis predicted branch, Load/use hazard and ret command in icode in Decode, Execute or Memory stage.
4. E_bubble is active for mis predicted branch or Load/use hazard.

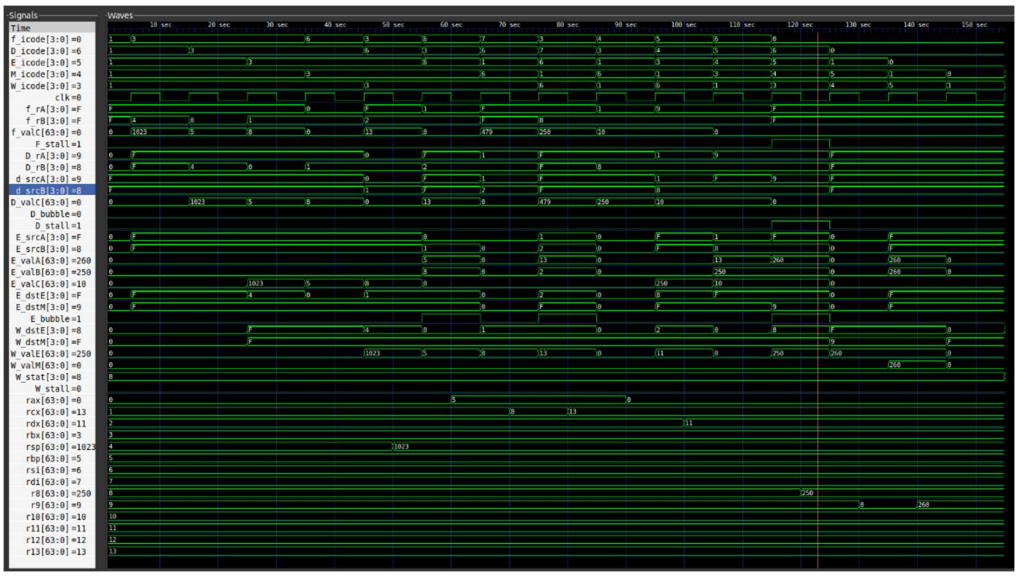
Processor:

This is the Wrapper module Which combines all the other modules into one, give initialisations and monitor everything. This is just a wrapper and doesn't have much functionality to itself but has only one but important functionality. It checks for status in Writeback stage and based on its value, decides whether to stop the processor.

GTKwave simulation of a testbench to test working of each individual function



GTKwave simulation of a testbench to test working of consecutive instructions



Supported Features:

From the previous GTKwave simulations, we can see that all the features are supported.

1. halt
2. nop
3. cmovXX
4. irmovq
5. rmmovq
6. mrmovq
7. OPq
8. jXX
9. call
10. ret
11. Pushq
12. Popq

Problems Faced:

1. Making sure that everything happens at the correct time. Even the slightest delay made the code incorrect.
2. Order in which code was written also mattered a lot. For example, assign statements and always blocks.
3. Initialising the value of PC to start the instructions at. To solve this problem, we initialised all the registers with nop values.
4. We were not able to make the registers follow the PCL logic correctly.