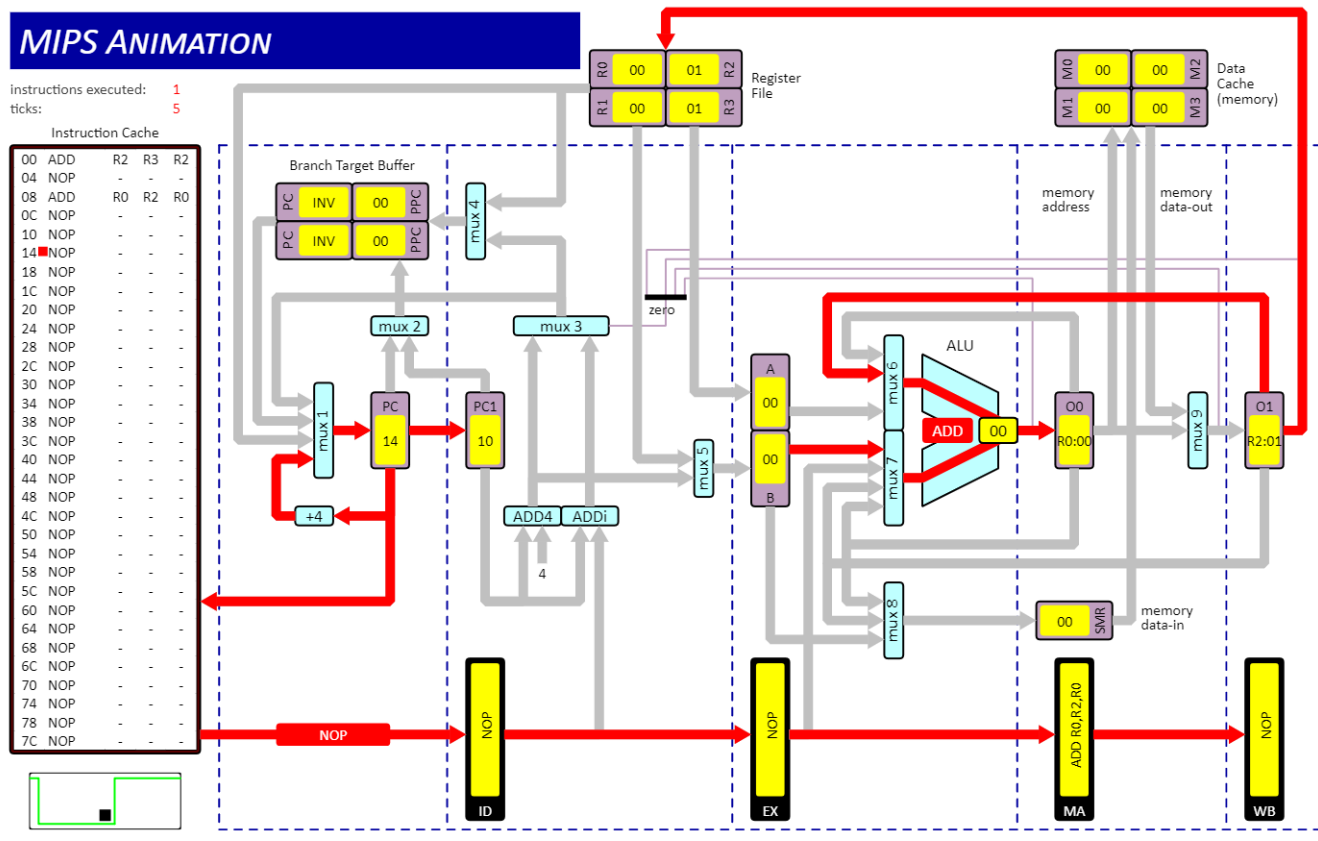
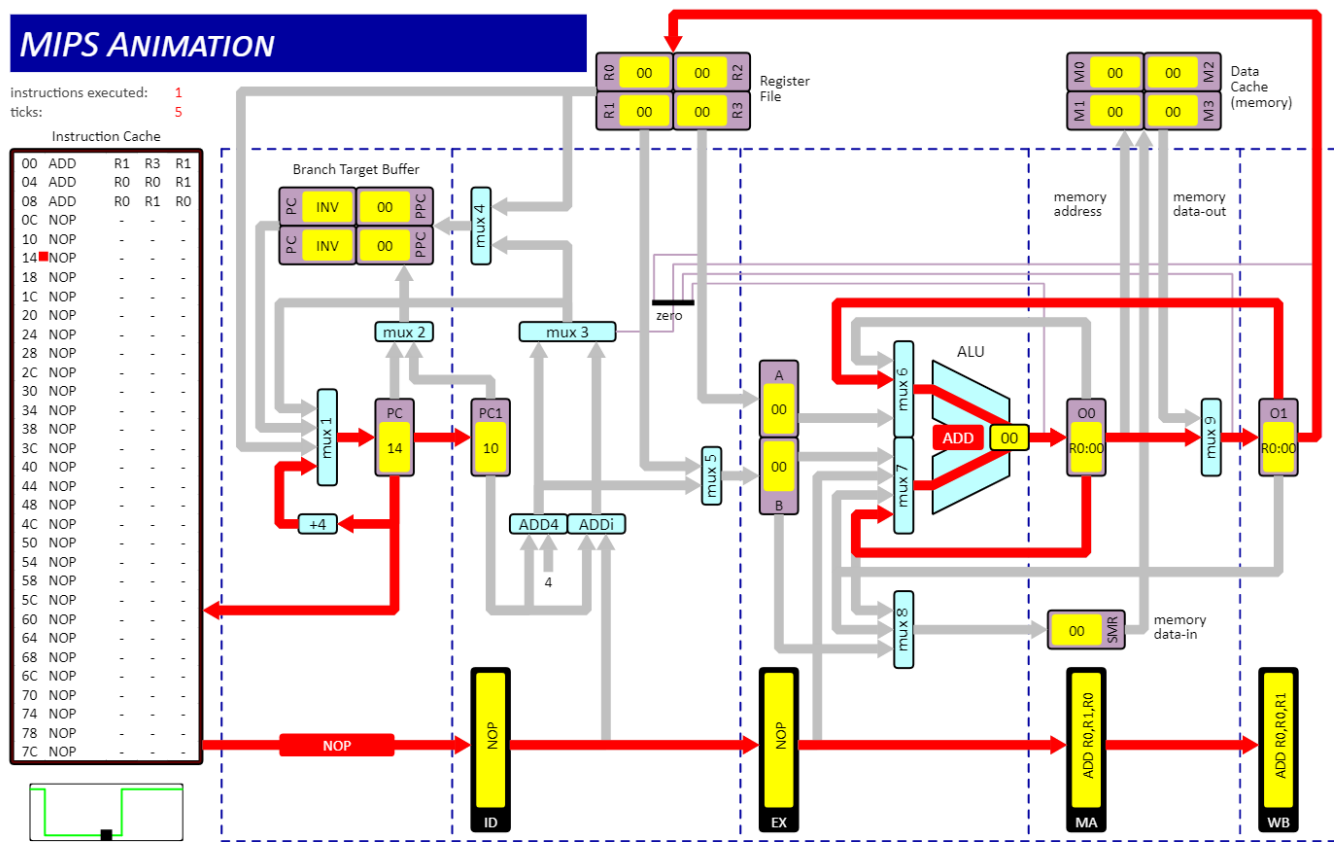


1. O1 to MUX6



2. O0 to MUX7 and O1 to MUX6 (simultaneously)



MIPS ANIMATION

Instructions executed: 1
ticks: 5

Instruction Cache

00	ADD	R0	R3	R1
04	NOP	-	-	-
08	ST	R0	R0	O1
0C	NOP	-	-	-
10	NOP	-	-	-
14	NOP	-	-	-
18	NOP	-	-	-
1C	NOP	-	-	-
20	NOP	-	-	-
24	NOP	-	-	-
28	NOP	-	-	-
2C	NOP	-	-	-
30	NOP	-	-	-
34	NOP	-	-	-
38	NOP	-	-	-
3C	NOP	-	-	-
40	NOP	-	-	-
44	NOP	-	-	-
48	NOP	-	-	-
4C	NOP	-	-	-
50	NOP	-	-	-
54	NOP	-	-	-
58	NOP	-	-	-
5C	NOP	-	-	-
60	NOP	-	-	-
64	NOP	-	-	-
68	NOP	-	-	-
6C	NOP	-	-	-
70	NOP	-	-	-
74	NOP	-	-	-
78	NOP	-	-	-
7C	NOP	-	-	-

The diagram illustrates the MIPS architecture and its execution flow. Key components include:

- Instruction Cache:** A table of instructions indexed by PC values. The first instruction is `ADD R0, R3, R1` at PC 00.
- Branch Target Buffer:** A buffer for branch targets, containing PC, INV, and PPC values.
- Register File:** A set of registers (R0-R3) storing data. R0 and R3 contain 00, while R1 and R2 contain 01.
- ALU:** The Arithmetic Logic Unit performing operations like ADD. It takes inputs from registers and performs operations like ADD, INV, and SWR.
- Data Cache (memory):** A cache for memory data, containing M0-M3. M0 and M1 contain 00, while M2 and M3 contain 01.
- Pipeline Stages:** The execution is divided into five stages: ID (Instruction Decode), EX (Execute), MA (Memory Access), and WB (Write Back). Each stage is represented by a box at the bottom.
- Execution Flow:** A red path shows the execution of the first instruction (`ADD R0, R3, R1`) and the second instruction (`ST R0, R0, O1`). The first instruction is executed in the ID stage at tick 0, then moves through EX, MA, and WB stages. The second instruction is fetched at tick 4 and moves through the ID stage.

MIPS ANIMATION

Instructions executed: 1
ticks: 5

Instruction Cache

00	ADD	R0	R3	R1
04	NOP	-	-	-
08	ST	R0	R0	O1
0C	NOP	-	-	-
10	NOP	-	-	-
14	NOP	-	-	-
18	NOP	-	-	-
1C	NOP	-	-	-
20	NOP	-	-	-
24	NOP	-	-	-
28	NOP	-	-	-
2C	NOP	-	-	-
30	NOP	-	-	-
34	NOP	-	-	-
38	NOP	-	-	-
3C	NOP	-	-	-
40	NOP	-	-	-
44	NOP	-	-	-
48	NOP	-	-	-
4C	NOP	-	-	-
50	NOP	-	-	-
54	NOP	-	-	-
58	NOP	-	-	-
5C	NOP	-	-	-
60	NOP	-	-	-
64	NOP	-	-	-
68	NOP	-	-	-
6C	NOP	-	-	-
70	NOP	-	-	-
74	NOP	-	-	-
78	NOP	-	-	-
7C	NOP	-	-	-

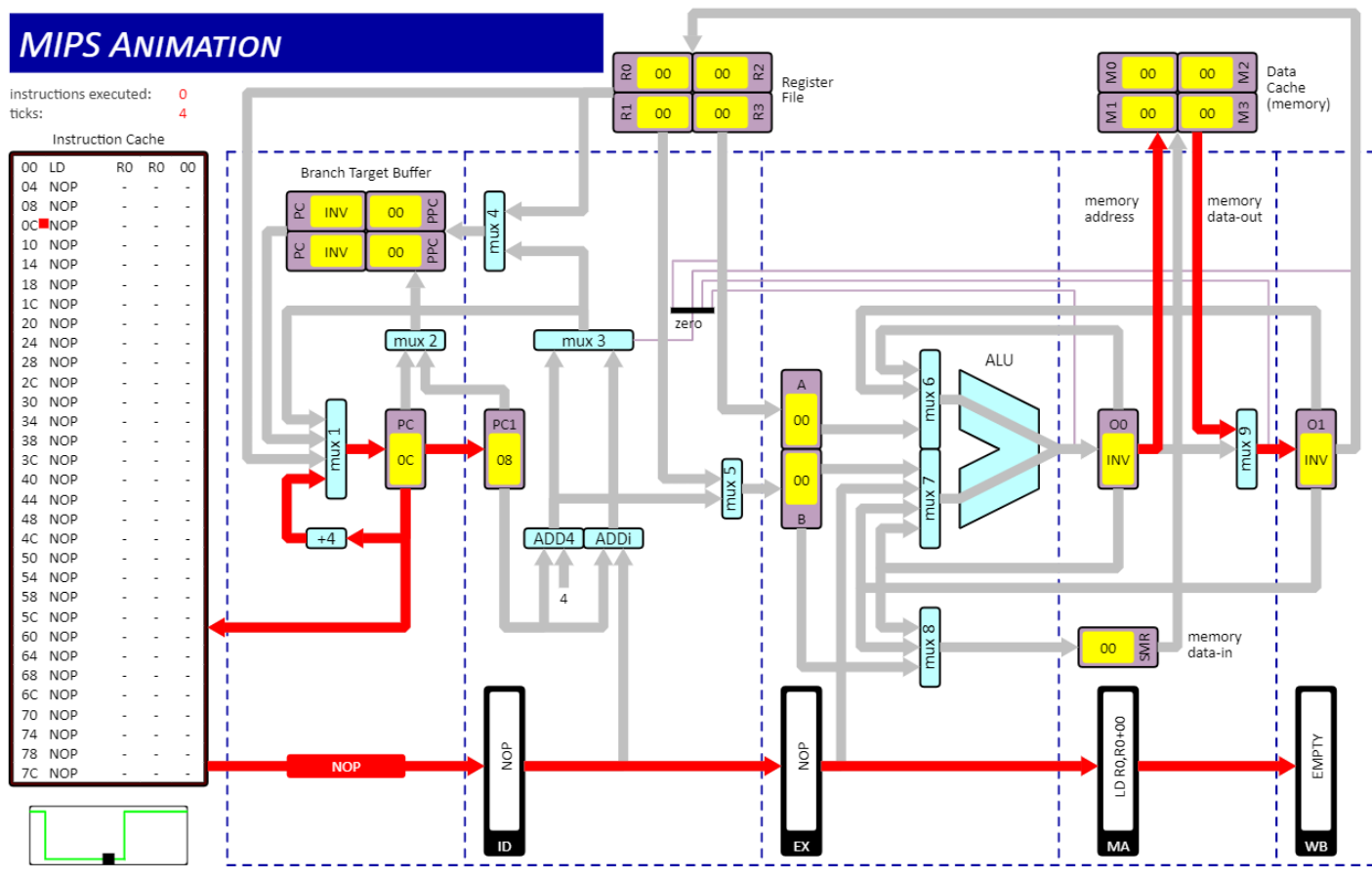
The diagram illustrates the MIPS architecture and the execution of the first instruction (NOP) at tick 5. The components shown include:

- Instruction Cache:** A table of instructions. The first instruction (00) is ADD R0, R3, R1. The current instruction being executed is NOP (10).
- Register File:** A table of registers. R0, R1, R2, R3 are shown. R0 and R1 contain 00.
- Branch Target Buffer:** A table of branch targets. The first entry (00) is INV 00. The current entry being used is INV 00.
- ALU:** The Arithmetic Logic Unit. It takes inputs from registers R0 and R1 (both 00) and performs an ADD operation, resulting in 00.
- Data Cache (memory):** A table of memory locations. M0, M1, M2, M3 are shown. M0 and M1 contain 00.
- Control Signals:** Various signals like 'zero', 'O1', and 'NOP' are shown, indicating the state of the processor.

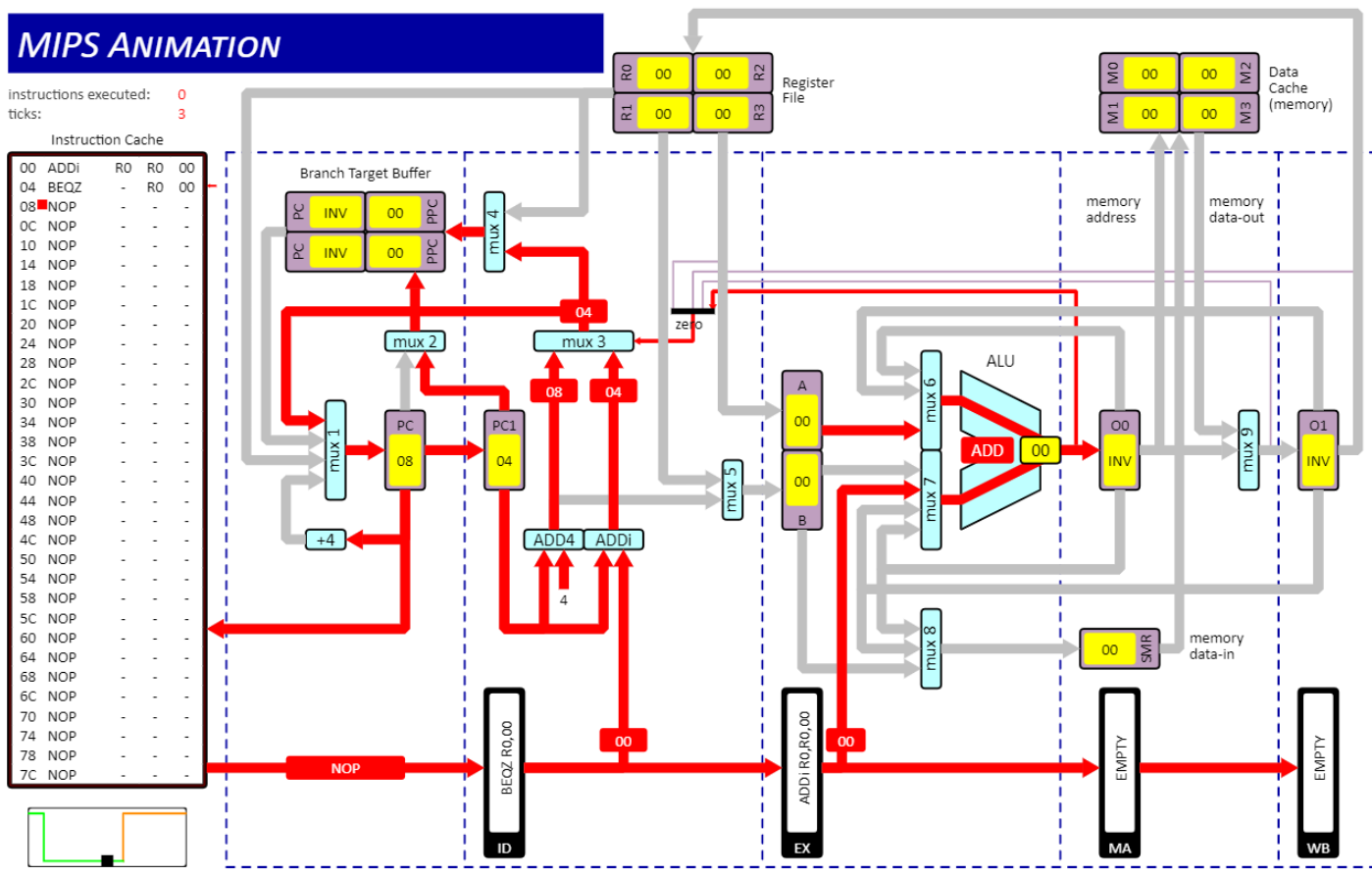
The execution flow is as follows:

- The Instruction Cache provides the instruction NOP (10) to the PC (Program Counter).
- The PC (10) is used to calculate the next instruction address (14) using the ADD4 and ADDi logic.
- The ALU takes inputs from registers R0 and R1 (both 00) and performs an ADD operation, resulting in 00.
- The ALU result (00) is stored in the Register File (R0).
- The Data Cache (memory) contains 00 at address 00.

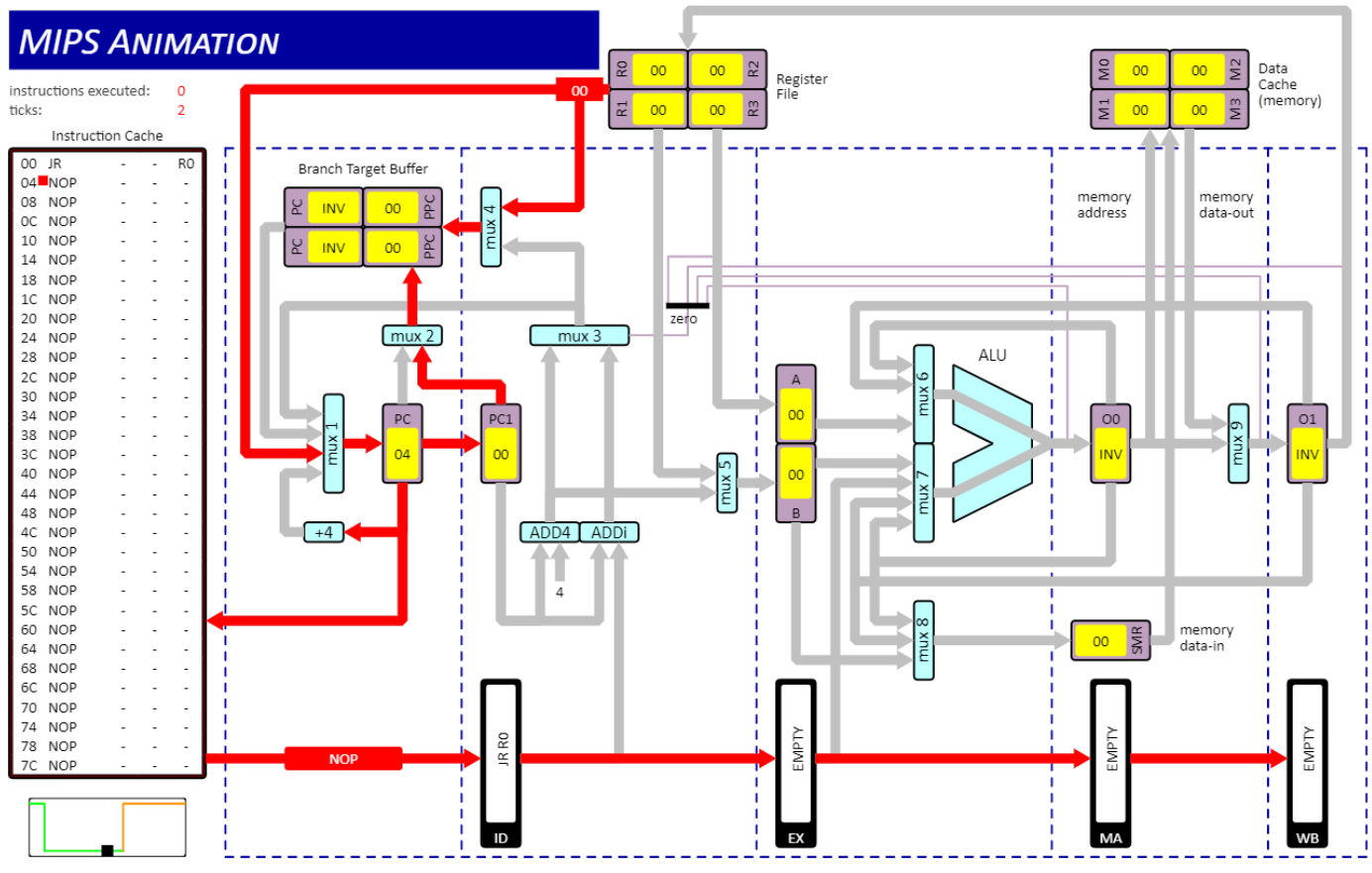
5. Data cache to MUX9 (memory data-out)



6. 00 to Zero detector



7. Register File to MUX1



Q2

- i) ALU forwarding: $r1 = 15$, Clock Cycles = 10
- ii) No ALU forwarding, CPU interlocks enabled: $r1 = 15$, Clock Cycles = 18
- iii) ALU forwarding, CPU interlocks disabled: $r1 = 6$, Clock Cycles = 10

The number of clock cycles is different with no ALU forwarding because the result from one instruction cannot be fed back into the ALU as an input. When ALU forwarding is enabled, the processor keeps the result stored in the immediate registers Out0 then Out1 for one clock cycle each. If one of the register operands in a subsequent instruction is the same as the register value of Out0 or Out1, this value will be sent to MUX6 or MUX7 depending on its positioning in the instruction. The benefit of this is we can avoid stalling due to the Memory Access and Write Back stages in the pipeline. When forwarding is absent, these stages must be carried out in full, leading to an increase in the amount of clock cycles required to carry out the same amount of instructions.

In the case of part iii) $R1 = 6$ because there are no interlocks in place to prevent data hazards from occurring. R1 and R2 are used in subsequent instructions, before these registers can be properly written back into memory.

Q3

i) Instructions = 39, Clock Cycles = 51

These numbers are not equal due to the design and nature of the pipeline itself. For the first 4 clock cycles no instruction is executed in full (despite having passed the “Execution stage”), as the result still needs to be written back into memory to be considered ‘complete’. An instruction cannot be executed if it hasn’t been fetched or decoded yet. Similarly, when the processor passes the Load or Store instructions, the pipeline is stalled for another clock cycle. This is to avoid a data hazard when storing and loading values to and from memory to registers that are subsequently used. Enabling interlocks prevents this hazard from occurring at the cost of lower throughput.

ii) Instructions = 39, Clock Cycles = 53

When branch prediction is exchanged for branch interlock, the processor takes more clock cycles due to the Jump instructions. Whenever the Jump instruction is taken the processor stalls for an additional clock cycle to prevent a control hazard from occurring. The next logical instruction stored in memory may not be the next instruction to be executed as the processor may branch to a different one.

iii) Instructions = 39, Clock Cycles = 47

When the Logical Shift instructions are exchanged the code takes shorter to execute because R2 is being shifted right one instruction later. This has a knock-on effect on the loop in the program as it terminates when $r2 = 0$. The BEQZ is operating on R2 before it can be written into memory .