Lab 05 - Pipelined Datapath

Table of Contents

1. single cycle datapath Results

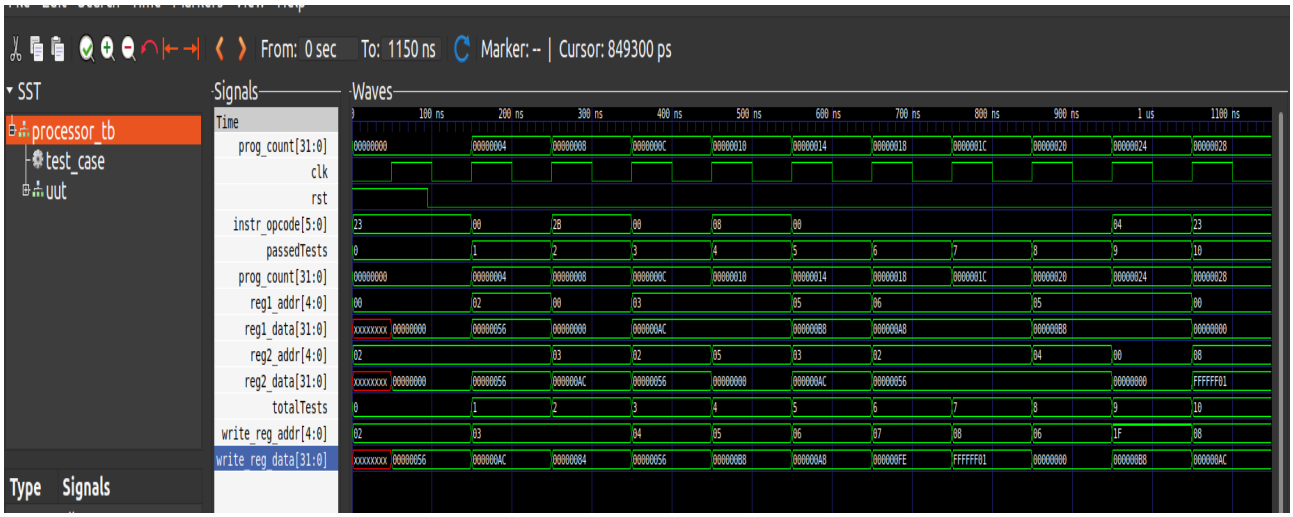
If we run the code in `init.asm` in the schematic capture design of the single cycle datapath from the lab4 over the course of running the 11 instructions in 11 cycles we are getting the following outputs.

Table

clk	PC	Opcode	src1_addr	src1_out	src2_addr	src_out	dst_addr	dst_data
1	0	0x23	0	0x00000000	2	0x00000000	2	0x00000056
2	4	0x00	2	0x00000056	2	0x00000056	3	0x000000AC
3	8	0x2B	0	0x00000000	3	0x000000AC	3	0x00000084
4	12	0x00	3	0x000000AC	2	0x00000056	4	0x00000056
5	16	0x08	3	0x000000AC	5	0x00000000	5	0x000000B8
6	20	0x00	5	0x000000B8	3	0x000000AC	6	0x000000A8
7	24	0x00	6	0x000000A8	2	0x00000056	7	0x000000FE
8	28	0x00	6	0x000000A8	2	0x00000056	8	0xFFFFFF01
9	32	0x00	5	0x000000B8	4	0x00000056	6	0x000000

clk	PC	Opcode	src1_addr	src1_out	src2_addr	src_out	dst_addr	dst_data
10	36	0x04	5	0x000000B8	0	0x00000000	31	0x000000B8
11	40	0x23	0	0x00000000	8	0xFFFFFF01	8	0x000000AC

Simulation



2. Pipelined datapath Results

Section I: Tables and run result

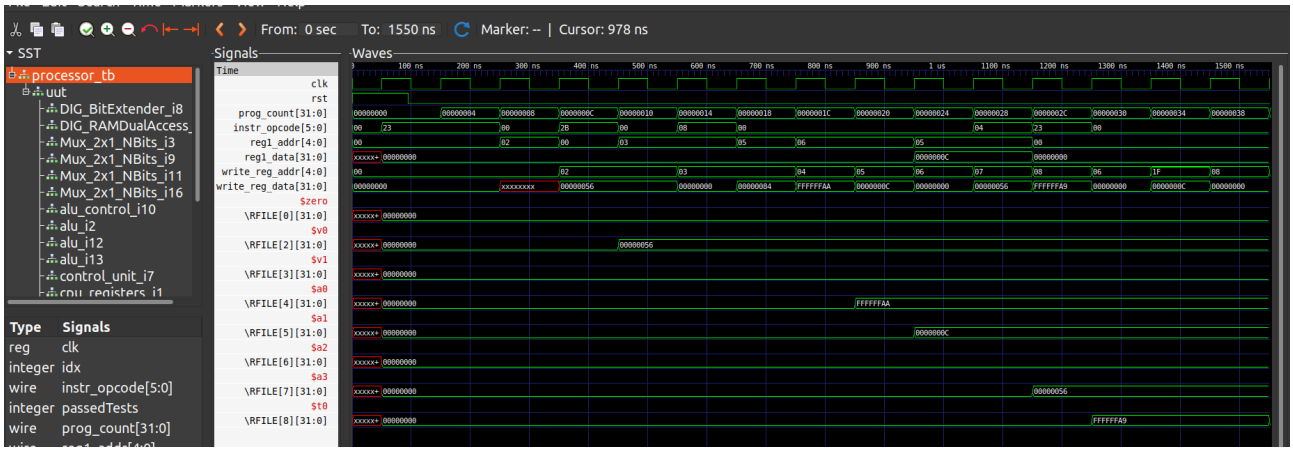
For lab5, same table has been produce for running the program in `labo5_pipeliend.dig` in Digital. However, for this version of the datapath, the program is run for 15 cycles to execute the 11 instructions. Why? Because the pipeline is causing the execuction of the program to take 4 cycles longer as can be seen in the tabel and simulation below.

Table

clk	PC	Opcode	src1_addr	src1_out	src2_addr	src_out	dst_addr	dst_data
1	0	0	0	0x00000000	0	0x00000000	0	0x00000000
2	4	0x23	0	0x00000000	2	0x00000000	0	0x00000000

clk	PC	Opcode	src1_addr	src1_out	src2_addr	src_out	dst_addr	dst_data
3	8	0x00	2	0x00000000	2	0x00000000	0	0x00000000
4	12	0x2B	0	0x00000000	3	0x00000000	0	0x00000000
5	16	0x00	3	0x00000000	2	0x00000000	2	0x00000056
6	20	0x08	3	0x00000000	5	0x00000000	3	0x00000000
7	24	0x00	5	0x00000000	3	0x00000000	3	0x00000084
8	28	0x00	6	0x00000000	2	0x00000056	4	0x00000000
9	32	0x00	6	0x00000000	2	0x00000056	5	0x0000000C
10	36	0x00	5	0x0000000C	4	0x00000000	6	0x00000000
11	40	0x24	5	0x0000000C	9	0x00000000	7	0x00000056
12	44	0x23	0	0x00000000	8	0x00000000	8	0xFFFFFFA9
13	48	0x00	0	0x00000000	0	0x00000000	6	0x00000000
14	52	0x00	0	0x00000000	0	0x00000000	31	0x0000000C
15	56	0x00	0	0x00000000	0	0x00000000	8	0x00000000

Simulation:



Section 2: Explanation of the Difference Between Pipelined and Non-Pipelined Outputs

In the pipelined datapath, instructions are fetched, decoded, executed, and written back in an overlapping manner. This means that multiple instructions are in different stages of execution

simultaneously. While this increases throughput, it introduces hazards that must be managed to ensure correct program execution.

Data Hazards: These occur when an instruction depends on the result of a previous instruction that has not yet completed. For instance, in the given code, `add $v1, $v0, $v0` depends on the result of `lw $v0, 31($zero)`. Without proper handling, this can cause incorrect values to be used.

Control Hazards: These arise from the need to make a decision based on the results of an instruction that has not yet completed, such as branch instructions. For example, `beq $a2, $zero, -8` depends on the outcome of the `slt` instruction.

In the absence of hazard detection and forwarding mechanisms, these hazards necessitate the insertion of stalls (NOPs) to wait for the necessary data to be available. This causes the pipelined version to take more cycles to complete compared to the non-pipelined version.

Section 3: Hazard Mitigation

To avoid data hazards between the `lw $v0, 31($zero)` and `add $v1, $v0, $v0` instructions, we can insert a no-operation (NOP) instruction. This will prevent the processor from attempting to use the result of the `lw` instruction before it is available.

lw $v0 31($zero)
nop
add $v1 $v0 $v0
sw $v1 132($zero)
sub $a0 $v1 $v0
addi $a1 $v1 12
and $a2 $a1 $v1
or $a3 $a2 $v0
nor $t0 $a2 $v0
slt $a2 $a1 $a0
beq $a2 $zero -8
lw $t0 132($zero)

Inserted Instruction: nop
Purpose: This instruction helps to avoid a data hazard by giving the `lw` instruction time to complete and write the value to $v0 before the `add` instruction tries to read it. It effectively introduces a stall, ensuring that the pipelined datapath behaves correctly.

By adding this NOP instruction, the program avoids the data hazard without changing the overall outcome of the program.