

## Instruction Memory

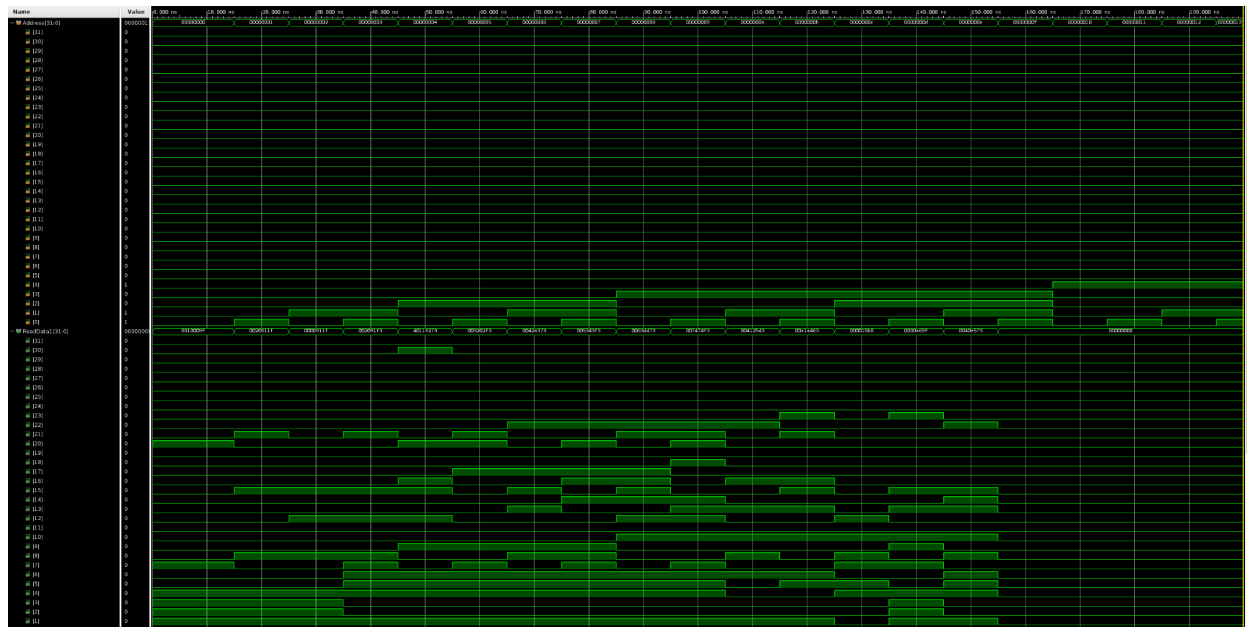
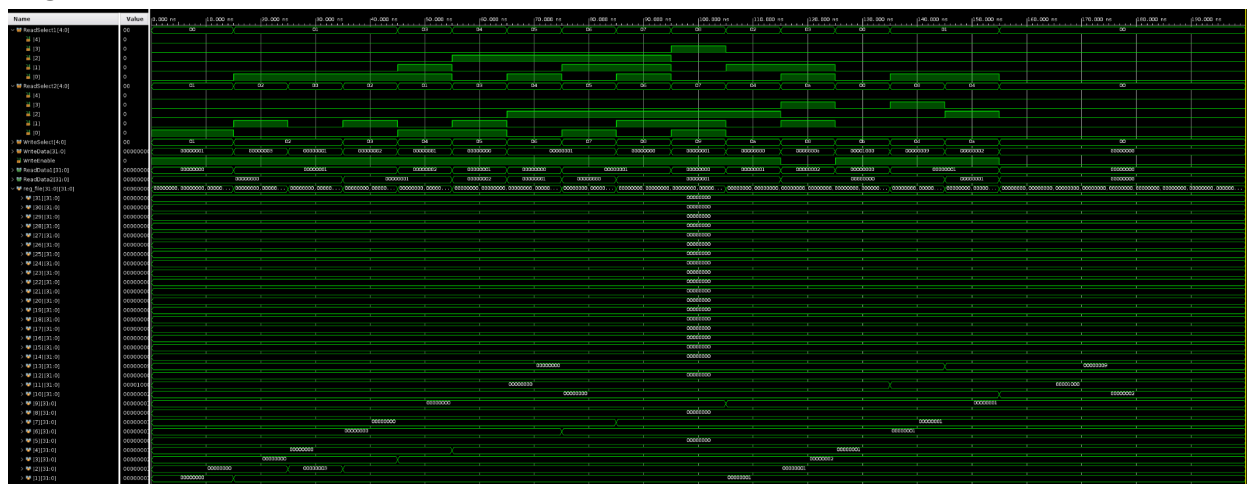


Figure 1: Instruction Memory Waveform.

In Figure 1, we can see the instruction memory waveform. The instruction memory includes all of the test scenarios necessary to execute the single-cycle data path. We can see that each of the instructions of the datapath is output through readData. The address is derived from the pc and we can see that it is counting by 1 which means it receives the signal properly from the PC. The output readData needs to be read by the control logic subsequently.

## Register File



The register file waveform needs to correctly store the results from each instruction in memory. To verify this functionality, we can test a range of inputs and observe whether the registers accurately store each value after execution. By reading the specific registers that require modification from the instruction memory, we can track which registers are populated with values, confirming the accuracy of our code. The output is directed to the ALU, with a MUX integrated into the register output, which also feeds into the ALU.

	Value	0x00000000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007	0x00000008	0x00000009	0x0000000A	0x0000000B	0x0000000C	0x0000000D	0x0000000E	0x0000000F	0x00000010	0x00000011	0x00000012	0x00000013	0x00000014	0x00000015	0x00000016	0x00000017	0x00000018	0x00000019	0x0000001A	0x0000001B	0x0000001C	0x0000001D	0x0000001E	0x0000001F	0x00000020	0x00000021	0x00000022	0x00000023	0x00000024	0x00000025	0x00000026	0x00000027	0x00000028	0x00000029	0x0000002A	0x0000002B	0x0000002C	0x0000002D	0x0000002E	0x0000002F	0x00000030	0x00000031	0x00000032	0x00000033	0x00000034	0x00000035	0x00000036	0x00000037	0x00000038	0x00000039	0x0000003A	0x0000003B	0x0000003C	0x0000003D	0x0000003E	0x0000003F	0x00000040	0x00000041	0x00000042	0x00000043	0x00000044	0x00000045	0x00000046	0x00000047	0x00000048	0x00000049	0x0000004A	0x0000004B	0x0000004C	0x0000004D	0x0000004E	0x0000004F	0x00000050	0x00000051	0x00000052	0x00000053	0x00000054	0x00000055	0x00000056	0x00000057	0x00000058	0x00000059	0x0000005A	0x0000005B	0x0000005C	0x0000005D	0x0000005E	0x0000005F	0x00000060	0x00000061	0x00000062	0x00000063	0x00000064	0x00000065	0x00000066	0x00000067	0x00000068	0x00000069	0x0000006A	0x0000006B	0x0000006C	0x0000006D	0x0000006E	0x0000006F	0x00000070	0x00000071	0x00000072	0x00000073	0x00000074	0x00000075	0x00000076	0x00000077	0x00000078	0x00000079	0x0000007A	0x0000007B	0x0000007C	0x0000007D	0x0000007E	0x0000007F	0x00000080	0x00000081	0x00000082	0x00000083	0x00000084	0x00000085	0x00000086	0x00000087	0x00000088	0x00000089	0x0000008A	0x0000008B	0x0000008C	0x0000008D	0x0000008E	0x0000008F	0x00000090	0x00000091	0x00000092	0x00000093	0x00000094	0x00000095	0x00000096	0x00000097	0x00000098	0x00000099	0x0000009A	0x0000009B	0x0000009C	0x0000009D	0x0000009E	0x0000009F	0x000000A0	0x000000A1	0x000000A2	0x000000A3	0x000000A4	0x000000A5	0x000000A6	0x000000A7	0x000000A8	0x000000A9	0x000000AA	0x000000AB	0x000000AC	0x000000AD	0x000000AE	0x000000AF	0x000000B0	0x000000B1	0x000000B2	0x000000B3	0x000000B4	0x000000B5	0x000000B6	0x000000B7	0x000000B8	0x000000B9	0x000000BA	0x000000BB	0x000000BC	0x000000BD	0x000000BE	0x000000BF	0x000000C0	0x000000C1	0x000000C2	0x000000C3	0x000000C4	0x000000C5	0x000000C6	0x000000C7	0x000000C8	0x000000C9	0x000000CA	0x000000CB	0x000000CC	0x000000CD	0x000000CE	0x000000CF	0x000000D0	0x000000D1	0x000000D2	0x000000D3	0x000000D4	0x000000D5	0x000000D6	0x000000D7	0x000000D8	0x000000D9	0x000000DA	0x000000DB	0x000000DC	0x000000DD	0x000000DE	0x000000DF	0x000000E0	0x000000E1	0x000000E2	0x000000E3	0x000000E4	0x000000E5	0x000000E6	0x000000E7	0x000000E8	0x000000E9	0x000000EA	0x000000EB	0x000000EC	0x000000ED	0x000000EE	0x000000EF	0x000000F0	0x000000F1	0x000000F2	0x000000F3	0x000000F4	0x000000F5	0x000000F6	0x000000F7	0x000000F8	0x000000F9	0x000000FA	0x000000FB	0x000000FC	0x000000FD	0x000000FE	0x000000FF	0x00000100	0x00000101	0x00000102	0x00000103	0x00000104	0x00000105	0x00000106	0x00000107	0x00000108	0x00000109	0x0000010A	0x0000010B	0x0000010C	0x0000010D	0x0000010E	0x0000010F	0x00000110	0x00000111
--	-------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------	------------

The ALU opcodes, provided in the lab handout, specify how different instructions should be processed. To ensure correct functionality, we need to verify that the ALU outputs can translate accurately to addresses used by the data memory. Integrating the ALU with the datapath can allow us to properly see the ALU's arithmetic capabilities.

[illegible]

In Figure 2, we can see our control logic waveform. Our control logic combines both the ALU control and control unit to simplify code. The input we can see properly reflects the output of the

instruction memory from the previous figure. Each 32-bit instruction specifies which operation to perform, and the ALU control outputs for each instruction reflect the necessary operations. For each instruction, an associated ALU opcode determines if additional steps are required for execution. For I-type instructions, an immediate value is included. We aim for each instruction to output its corresponding ALU control signal, opcode, and immediate. We can then test this by using instruction memory to verify if it accurately reads these outputs as inputs to the control logic.

## Immediate Generator

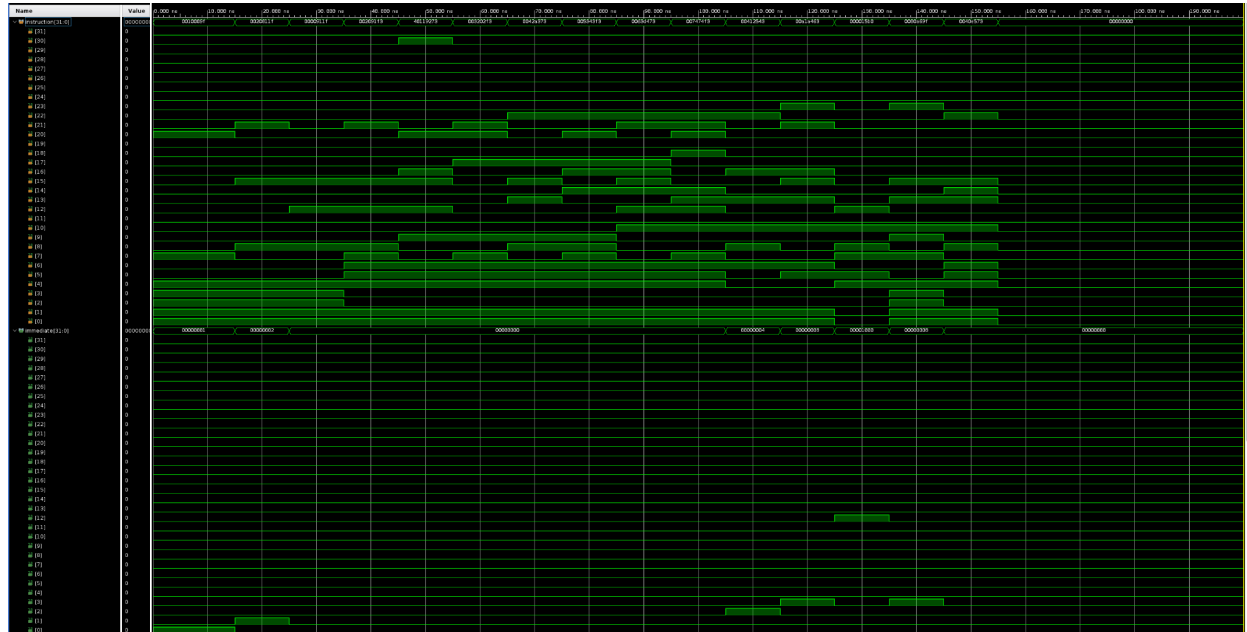


Figure 5: Immediate Generator Waveform

The immediate generator waveform also receives input from instruction memory. It interprets this input and outputs an immediate value whenever needed, which we can observe in the immediate output signal. This immediate value is then passed to the branch component.

## Data Memory



Figure 6: Data Memory Waveform

Figure 3 shows the waveform for data memory. The mux is implemented into the data memory to reduce the amount of modules needed in the datapath. Data memory operates only when a memory read is triggered. We can confirm its correct behavior: whenever MEMread is set to 1, readData runs; otherwise, it remains inactive. In the datapath, later on, we can test the functionality of the MEMread and see if it behaves as intended with the register file.

## Branch

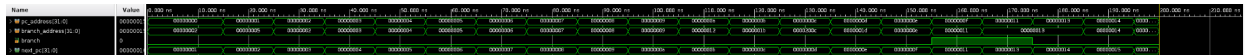


Figure 7: Branch Control Waveform

The branch we can test its viability by feeding it a beq or blt instruction if the address changes with the branch command we can confirm its effectiveness. The branch also receives zero flags to indicate if it's needed or not.

PC



### Figure 8: Program Counter Waveform

The PC is simple, all it needs to do is increment by 1 and we know it properly functions. The adder for incrementing is implemented into this module for simplification. The PC also needs to be synced properly to the clock which we can see it working perfectly in the waveform. In the datapath, later on, we can further test the PC by seeing if it properly increments whenever the instruction is a branch. The address needs to then be fed to the instruction memory.

## Single Cycle Data Path

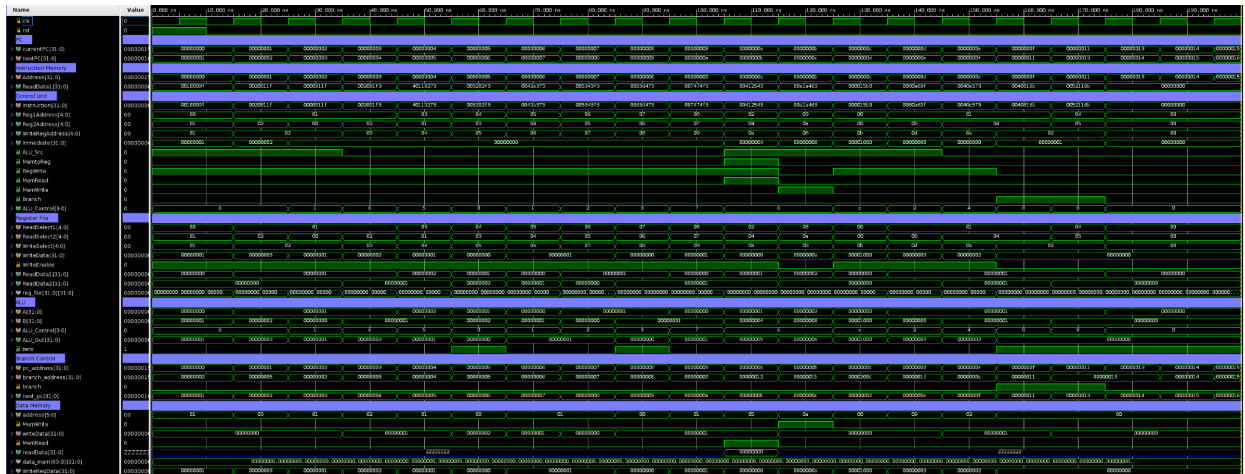


Figure 9: Single Cycle Datapath Waveform 1

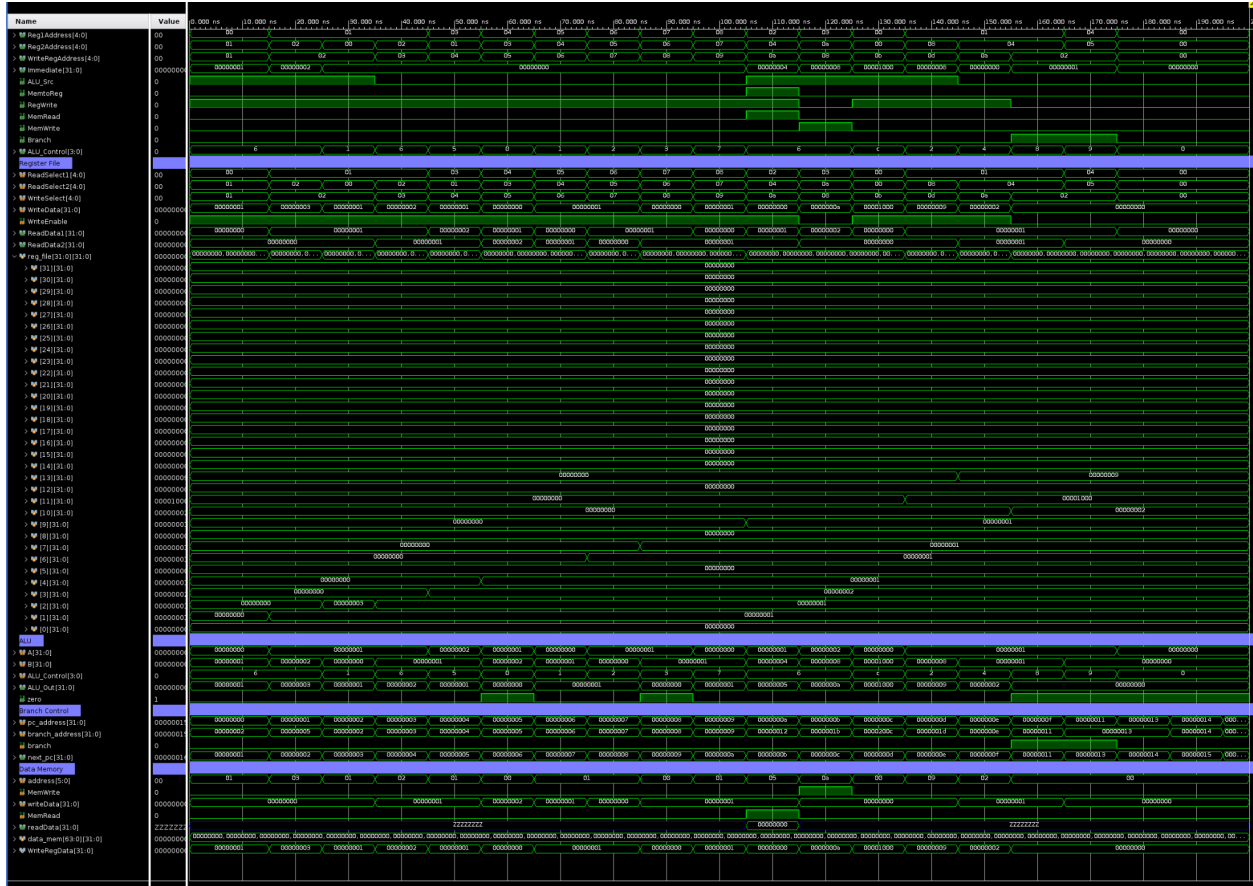


Figure 10: Single Cycle Datapath Waveform 2

The figure above shows the final single-cycle datapath waveform, demonstrating all instructions outlined in the lab handout. The top module's only inputs are `clk` and `rst`, where the datapath completes a single cycle with each clock tick, and the positive clock edge triggers the execution of a new instruction. At the start, we initialize the module with `rst` set to 1.

In this program, all registers are initialized to 0. We began with the instructions `addi x1, x0, 1` and `addi x2, x1, 2`. After the first cycle, `x1` is set to 1, and after the second cycle, `x2` becomes 3. Next, we execute `ori x2, x1, 0`, which results in `x2` holding the value 1. Then, the instruction `add x3, x1, x2` is executed, making `x3` equal to 2. Following that, `sub x4, x3, x1` sets `x4` to 1 in the register, and `and x5, x4, x3` results in `x5` storing the value 0. The subsequent `or x6, x5, x4` instruction sets `x6` to 1, and `xor x7, x6, x5` makes `x7` hold the value 1. We then ran `slt x9, x8, x7`, resulting in `x9` being set to 1 since `x8` is 0 and `x7` is 1.

The following instructions involve data memory, starting with `lw x10, 4(x2)`. At 105 ns, we see an ALU output address of 5, which is correct as `x2` holds 1, causing data memory to load a 0 into `x10` (since data memory at that address is 0). The next instruction, `sw x10, 8(x3)`, stores 0 at address 10 in data memory. Afterward, `lui x11, 0x(1)` loads a 1 followed by 12 zeros, which we see in the register file as 00001000 in hexadecimal. Then, `xori x13, x1, 8` stores 9 in `x13`. The

instruction `sll x10, x1, x4` results in `x10` holding 2 in hexadecimal as it performs a left logical shift on binary 1 by one position.

The final two instructions use branch control. First, `beq x1, x4, 1` increments the program counter (PC) by 2, as `x1` and `x4` are equal, causing the PC to jump from 15 to 17. The last instruction, `blt x5, x4, 1`, at PC address 17, is executed, and since `x5` is less than `x4`, the PC increments again to reflect this branch.

With these tests complete, we are confident in the accuracy of our final single-cycle datapath.