Aastha Khare
Madeleine Waldie
Marianne Fuyu Yamazaki Dorr

COEN 122: Final Project Report

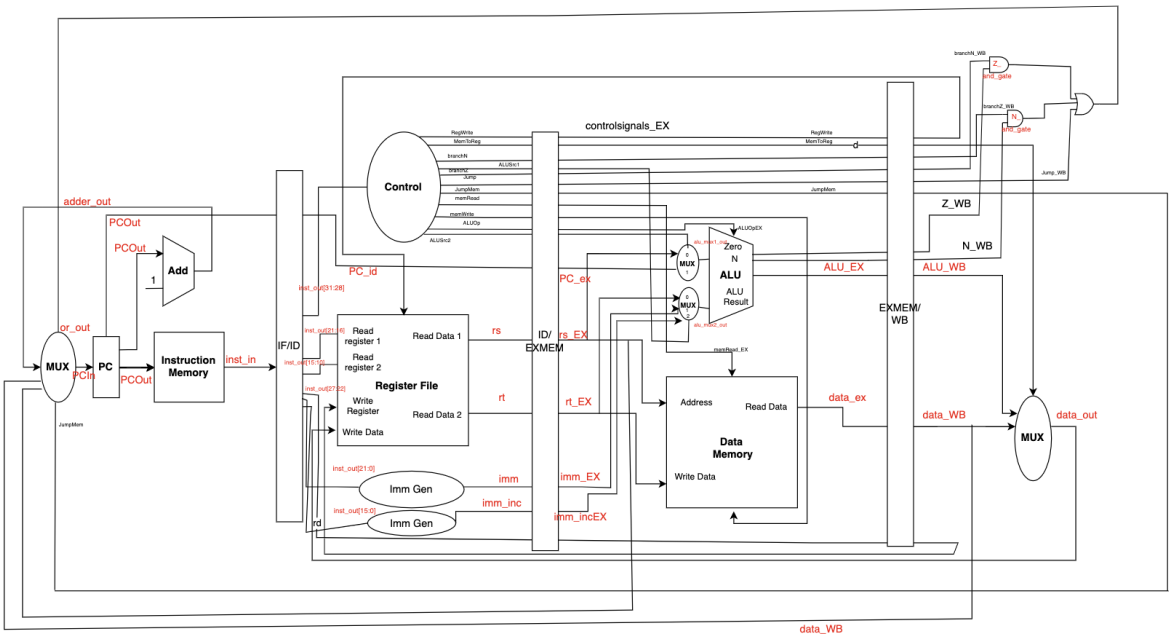
Abstract

In this project, we designed a 32-bit pipelined CPU for the SCU Instruction Set Architecture (SCU ISA), which consists of 13 instructions. Before writing our code, we designed a datapath and created a truth table for our CPU, ensuring that all 13 instructions could be executed. Next, we used the SCU ISA to write Assembly code to execute the following instruction: $A = a_1 + a_2 + ... + a_n$. Finally, we wrote Verilog code to implement the different modules in our data path and ran simulations to verify our datapath's behavior. This report will walk you through the different stages in our CPU creation.

Description of CPU Design

Below, you can see our complete datapath design, as well as a truth table for all 13 of the SCU ISA instructions, using our datapath. Our CPU design comprises of four stages – IF, ID, EXMEM, and WB – and thus has three buffers – IF/ID, ID/EXMEM, and EXMEM/WB. These buffers store results from every stage prior. While our PC helps with incrementing each instruction, those instructions are then fed into and fetched from the instruction memory. In the ID stage, we use the register file to access different registers that need to be read and written to. In this stage, we also use the control unit to determine the control signal logic for the different components in the datapath. Our immediate generators sign-extend immediates to 32-bits before storing it in the ID/EXMEM stage. Next in the EXMEM stage we have two major components: ALU and Data Memory. The Data Memory is designed to handle instructions that require memory accesses. Our ALU performs arithmetic operations based on the ALU opcode to perform operations such as addition, subtraction, negation, etc. For our branch instructions, it also sets the Z and N flags; meaning, if the instruction is branch-if-zero the Z flag will be 1 when our ALU result is 0. The N flag will be 1 when our ALU result is negative. The branch instructions are then determined by the logic gates (using the respective flags and control signals) in the EX/MEM stage.

Datapath



Truth Table

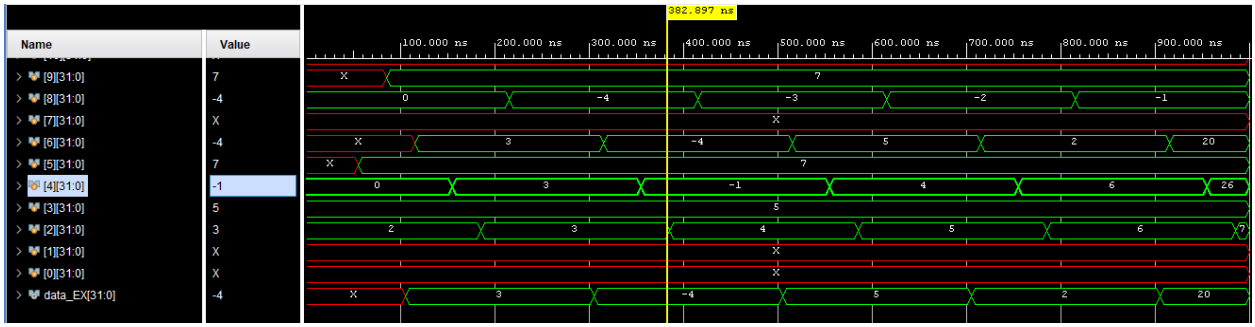
Below, you can find the truth table for the thirteen SCU ISA instructions, based on the datapath above. The different values in the truth table are based upon these definitions:

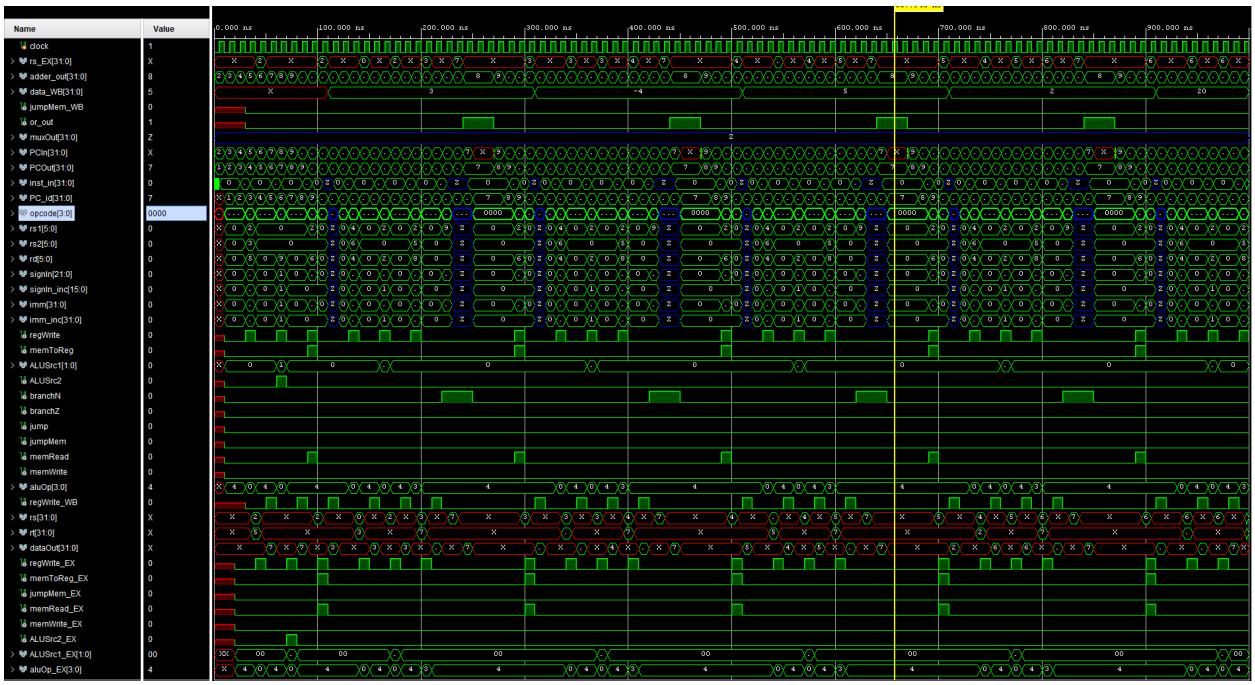
- **regWrite:** whether to write on register file
- **memtoReg:** whether to write data or read from memory on register file
- **PCtoReg:** whether to write PC on register file
- **branchN:** set for branch if negative instruction result
- **branchZ:** set for branch if instruction zero result
- **Jump:** set for jump instruction
- **jumpMem:** set for jump memory instruction
- **memRead:** whether to read from data memory
- **memWrite:** whether to write into data memory
- **aluOp:** selects the needed ALU operation

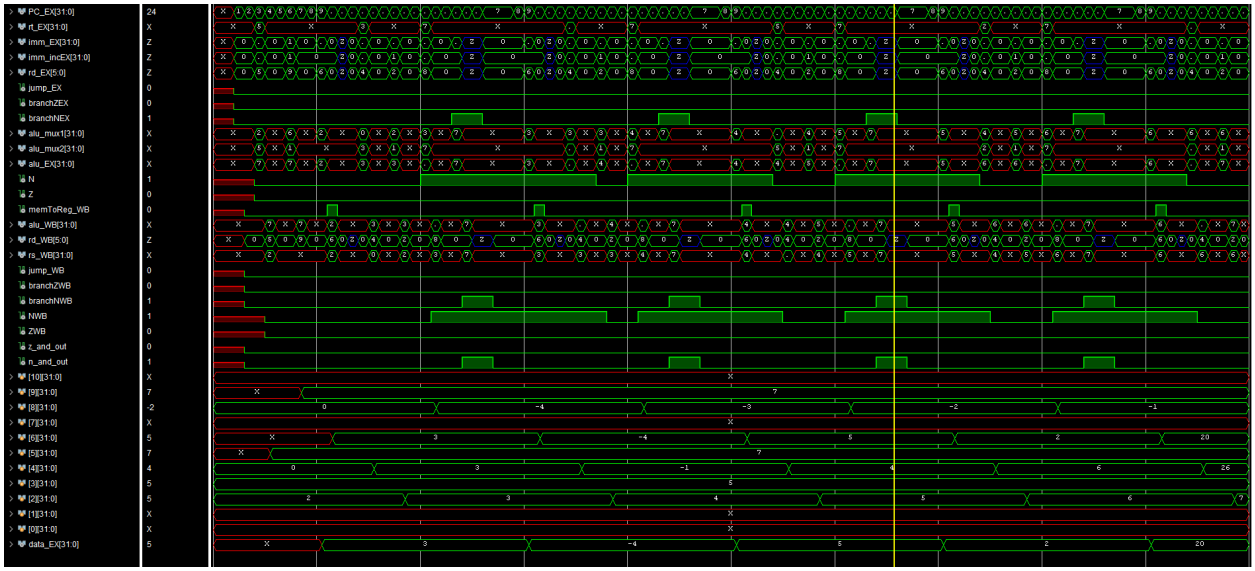
Instruction	Opcode	RegWrite	MemToReg	PCtoReg	branchN	branchZ	Jump	JumpMem	memRead	memWrite	ALUop
No Operation	0000	0	0	0	0	0	0	0	0	0	0100
Save PC	1111	1	0	1	0	0	0	0	0	0	0000
Load	1110	1	1	0	0	0	0	0	1	0	0100
Store	0011	0	0	0	0	0	0	0	0	1	0100
Add	0100	1	0	0	0	0	0	0	0	0	0000

Increment	0101	1	0	0	0	0	0	0	0	0	0000
Negate	0110	1	0	0	0	0	0	0	0	0	0010
Subtract	0111	1	0	0	0	0	0	0	0	0	0011
Jump	1000	0	0	0	0	0	1	0	0	0	0100
Branch if Zero	1001	0	0	0	0	1	0	0	0	0	0100
Jump Memory	1010	0	0	0	0	0	0	1	0	0	0100
Branch if Negative	1011	0	0	0	1	0	0	0	0	0	0100
SUM	0001	1	1	0	0	0	0	0	1	0	0010

Simulation Verification


Name
Value
[9][31:0]
[8][31:0]
[7][31:0]
[6][31:0]
[5][31:0]
[4][31:0]
[3][31:0]
[2][31:0]
[1][31:0]
[0][31:0]
data_EX[31:0]
382.897 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns
800.000 ns
900.000 ns


Name
Value
clock
rs_EX[31:0]
adder_out[31:0]
data_WB[31:0]
jumpMem_WB
or_out
muxOut[31:0]
PCin[31:0]
PCOut[31:0]
inst_in[31:0]
PC_id[31:0]
opcode[3:0]
rs1[5:0]
rs2[5:0]
rd[5:0]
signIn[21:0]
signIn_inc[15:0]
imm[31:0]
imm_inc[31:0]
regWrite
memToReg
ALUSrc1[1:0]
ALUSrc2
branchN
branchZ
jump
jumpMem
memRead
memWrite
aluOp[3:0]
regWrite_WB
rs[31:0]
rt[31:0]
dataOut[31:0]
regWrite_EX
memToReg_EX
jumpMem_EX
memRead_EX
memWrite_EX
ALUSrc2_EX
ALUSrc1_EX[1:0]
aluOp_EX[3:0]


PC_EX[31:0]
rt_EX[31:0]
imm_EX[31:0]
imm_incEx[31:0]
rd_EX[5:0]
jump_EX
branchZEX
branchNEX
alu_mux1[31:0]
alu_mux2[31:0]
alu_EX[31:0]
N
Z
memToReg_WB
alu_WB[31:0]
rd_WB[5:0]
rs_WB[31:0]
jump_WB
branchZWB
branchNWB
NWB
ZWB
z_and_out
n_and_out
data_EX[31:0]

Assembly Code

Assumptions:

- **x2:** base of A
- **x3:** n - 1
- **x4:** $\sum_{i=1}^{n} a_i$, where $n \geq 1$
- **x9:** address of 1st instruction in loop; branch address
- **x5:** address of a_n

Version 1 Assembly Code (using soft-loop):

```
SUB x4, x4, x4          //clear x4 to 0
ADD x5, x2, x3          //save location of last element to be summed
SVPC x9, 1              //save branch address

LD x6, x2               //load first element in memory to register x6
ADD x4, x4, x6          //add first element to x4
INC x2, x2, 1           //go to next element
SUB x8, x2, x5          //set up negative flag
BRN x9                  //if negative flag == 1, jump back to beginning of loop

STOP                    //end of code
```

Version 2 Assembly Code (loop implemented by hardware):

```
SUB x4, x4, x4          //clear x4 to 0
INC x3, x3, 1           //increment x3 by 1 so that x3 = n and not n-1 from our assumption
SUM x4, x2, x3          //call SUM instruction
```

```
// Sum program:
assign instructions[0] = 32'b0111_000100_000100_000100_0000000000; //SUB x4, x4, x4
assign instructions[1] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[2] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[3] = 32'b0100_000101_000010_000011_0000000000;// ADD x5, x2, x3;
assign instructions[4] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[5] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[6] = 32'b1111_001001_00000000000000000001; //SVPC x9, 1
assign instructions[7] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[8] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[9] = 32'b1110_000110_000010_0000000000000000; //LD x6, x2
assign instructions[10] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[12] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[13] = 32'b0100_000100_000100_000110_0000000000; //ADD x4, x4, x6
assign instructions[14] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[15] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[16] = 32'b0101_000010_000010_0000000000000001; //INC x2, x2, 1
assign instructions[17] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[18] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[19] = 32'b0111_001000_000010_000101_0000000000; //SUB x8, x2, x5
assign instructions[20] = 32'b0000_000000_000000_000000_0000000000;
assign instructions[21] = 32'b0000_000000_000000_000000_0000000000;

assign instructions[22] = 32'b1011_000000_001001_0000000000000000; //BRN x9
```

Running Time Estimate

- Estimate of Running Time for sum calculation - Using the component delays defined in the lab handout estimate the number of cycles and the running time required for the execution of your sum assembly code

In the lab document, the component delays for our datapath are as follows:

- Delay of memory (I and D memory): 2 ns
- Delay of register file: 1.5 ns
- Delay of ALU (adders): 2 ns
- Ignore the delays of all other components

Thus, we can compute the running time for our sum assembly code, as follows:

cycle 1	cycle 2	cycle 3	cycle 4
IF	ID	EX/MEM	WB

Our datapath consists of 3 buffers and 4 stages, each of which takes 1 cycle. Therefore, each instruction (total 8, for the sum calculation) takes 4 cycles. Although inefficient, our design also consists of two NOP instructions between every regular instruction. This results in 2 cycles between every instruction. Therefore, overall our design has 88 cycles. As seen in our waveform, each cycle takes 10ns, which brings our total execution time to 880 ns.

Below is our cycle time analysis, according to the values we were given for the component delays. According to this, the instruction that takes the maximum cycle time is 7 ns, which means our cycle time would be 7 ns. If we use this calculation, our total execution time will result in 616 ns.

Cycle time analysis:

Arithmetic: IM + Register file + ALU + RegisterWrite = 2ns + 1.5ns + 2ns + 1.5ns = 7ns
Load: IM + Register file + DM + RegisterWrite = 2ns + 1.5ns + 2ns+ 1.5ns = 7ns
Store: IM + Register file + DM + RegisterWrite = 2ns + 1.5ns + 2ns + 1.5ns = 7ns
SVPC: IM + ALU + RegisterWrite = 2ns + 2ns + 1.5ns = 5.5ns
Branch: IM + Register file + ALU = 2ns + 1.5ns + 2ns = 5.5ns
JumpMem: IM + Register file + DM = 2ns + 1.5ns + 2ns = 5.5ns
Jump: IM + Register file = 2ns + 1.5 ns = 3.5 ns

Max cycle time = 7ns