# COEN 122 Project Report

Tian Zhang, Matthew Koken

## Abstract

As we learned about CPU structure and data path design, we are asked to design a structure model of a pipelined CPU with given instruction sets. Given a set of assembly instructions, the SCU instruction set, we are to design a datapath to implement the instructions. The instruction set includes 12 instructions that must be implemented at a hardware level, and an additional instruction that may be implemented using either hardware or software. Using pipelining, we implement a CPU that can perform the 12 instructions and fulfills the 13th instruction through software.

## Description

We designed a 32-bit pipelined CPU which follows SCU Instruction Set Architecture. The CPU has four pipeline stages: `IF/ID`, `ID/EX`, `EX/MEM` and `MEM/WB`.

Please see `Figure 1` for data path design. `Table 1` for control truth table.

Table 1

| Op | Symbol | Opcode | WB | | | MEM | | Ex | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | RegWrt | WriteDataSelect | BranchSelect | MemWrt | MemRd | ALUOp | BranchControl | Jump | Branch |
| **NOOP** | NOP | 0000 | 0 | 00 | 0 | 0 | 0 | 0000 | 0 | 0 | 0 |
| **Load** | LD | 1110 | 1 | 00 | 0 | 0 | 1 | 1111 | 0 | 0 | 0 |
| **Load PC** | LDPC | 1111 | 1 | 01 | 0 | 0 | 0 | 1111 | 0 | 0 | 0 |
| **Store** | ST | 0011 | 0 | 00 | 0 | 1 | 0 | 1111 | 0 | 0 | 0 |
| **Add** | ADD | 0100 | 1 | 10 | 0 | 0 | 0 | 1000 | 0 | 0 | 0 |
| **Increment** | INC | 0101 | 1 | 10 | 0 | 0 | 0 | 0100 | 0 | 0 | 0 |
| **Negate** | NEG | 0110 | 1 | 10 | 0 | 0 | 0 | 0010 | 0 | 0 | 0 |
| **Subtract** | SUB | 0111 | 1 | 10 | 0 | 0 | 0 | 0001 | 0 | 0 | 0 |
| **Jump** | J | 1000 | 0 | 00 | 0 | 0 | 0 | 1111 | 0 | 1 | 0 |
| **Br 0** | BRZ | 1001 | 0 | 00 | 0 | 0 | 0 | 1111 | 1 | 0 | 1 |
| **Jump Mem** | JM | 1010 | 0 | 00 | 1 | 0 | 1 | 1111 | 0 | 1 | 0 |
| **Br Neg** | BRN | 1011 | 0 | 00 | 0 | 0 | 0 | 1111 | 0 | 0 | 1 |

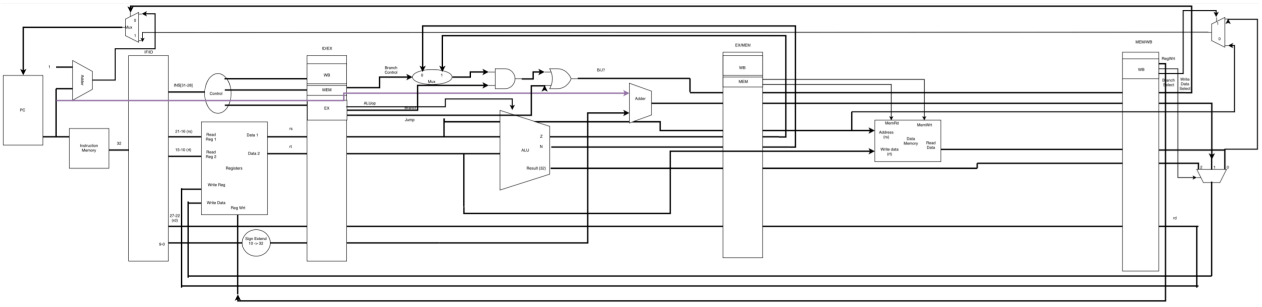


Figure 1

## IF

The initial portion of the CPU handles instruction fetch and decoding. This stage holds the program counter, which keeps track of the current instruction to be executed. It also holds the instruction memory, which holds the set of instructions to be executed. Each instruction holds the information in `Figure 1` along with additional destination address information. When an instruction is fetched, the instruction information is passed along to the next stage. The Program Counter is incremented or written to in order to move to the next instruction or jump to a different instruction.

## ID

The ID stage contains the Register File, which acts as the active data working set. Registers can be written to and can have data read from. The control module decodes the Opcode provided from the previous stage in order to determine the values for alter control signals. Here, instructions are decoded and addresses are passed to the register, which may be written to or read from.

## EX

The EX stage performs the major operations of the CPU. For any operation that requires arithmetic, the ALU here performs the needed operation. The stage also handles determining the next instruction address, should a branch or jump instruction be passed. In the case of BRZ or BRN instructions, the Z and N values from the ALU are stored to the next cycle, where they are used to evaluate for branching.

### MEM

The MEM stage handles operations for data memory - data writes and data reads. This is the main memory storage for the CPU outside of the active working set - the register files. Due to simulator limits, the storage size for data and register memory is more limited, but would be much larger in a production environment.

### WB

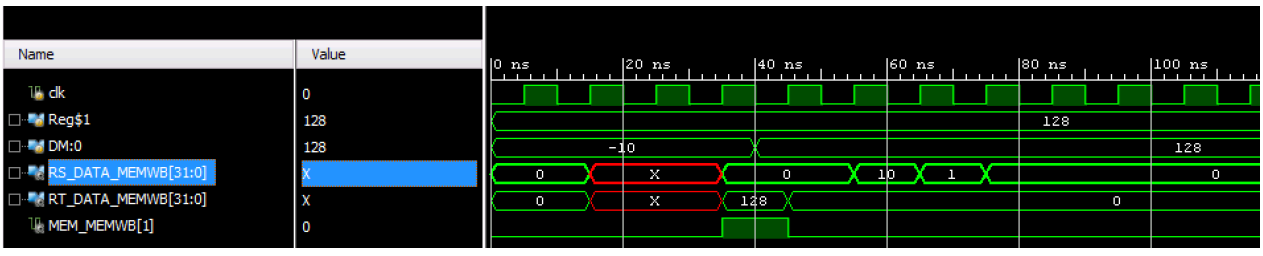
WB is essential for final storage. Operations that must write back to the register file are handled by this stage. The stage also holds the final elements for determining the next instruction address, which is written to the PC.

## Simulation Verification

#### ST $1, $0

```
Binary representation of the instruction
00110000000000000000010000000000 //$ST $1, $0
```

The first instruction we tested is `ST`, and the instruction is placed in `Instruction Memory block 1`. As the waveform shown, `Register $0` has value `0`, `Register $1` has value `128` and `Data Memory 0` has value `-10` at beginning. As instruction being executed, after 4 cycles, `MemWrt` goes to high, `RS` for data memory being set to `0`, and `RT` for data memory being set to value of `Register $1 (128)`. Then at next negative clock, the data being written into `Data Memory 0`.

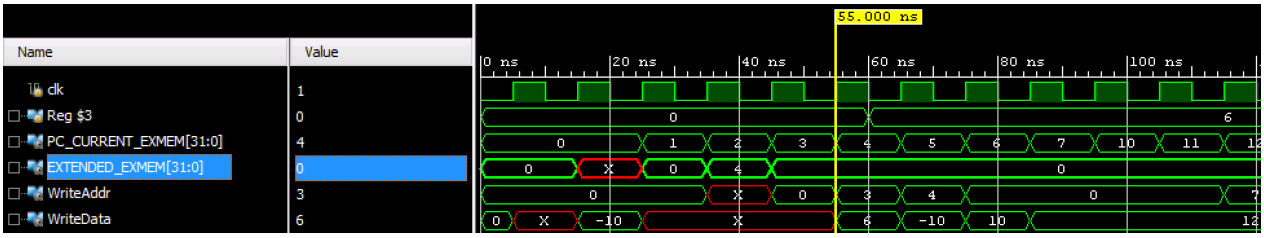



#### LDPC $3, 4

```
Binary representation of the instruction
```

```
11110000110000000000000000000100 //LDPC $3, 4
```

The second instruction we tested is `LDPC`, and the instruction is placed in `Instruction Memory block 2`. As the waveform shown, `Register $3` has value `0` at beginning. As instruction being executed, after 4 cycles, `RegWrt` goes to high, `WriteAddr` for register file being set to `3`, and `WriteData` for register file being set to result from PC adder (2 + 4 = 6). Then at next negative clock, the data being written into `Register $3`.




### MAX Calculation

```
// Binary representation of the instructions (In instruction memory)
assign block[0] = 32'b01010000110000110000000000000000; //INC $3, $3
assign block[1] = 32'b11100000000000010000000000000000; //LD $0, $1
assign block[2] = 32'b0;
assign block[3] = 32'b0;
assign block[4] = 32'b0;
assign block[5] = 32'b0;
// LOOP Start
assign block[6] = 32'b01110001000000100000110000000000; //SUB $4, $2, $3
assign block[7] = 32'b10010000000010000000000000000000; //BRZ END LOOP
assign block[8] = 32'b01000001010000010000110000000000; //ADD $5, $1, $3
assign block[9] = 32'b0;
assign block[10] = 32'b0;
assign block[11] = 32'b0;
assign block[12] = 32'b0;
assign block[13] = 32'b11100001000001010000000000000000; //LD $4, $5
assign block[14] = 32'b0;
assign block[15] = 0;
assign block[16] = 0;
assign block[17] = 0;
assign block[18] = 32'b01110001000001000000000000000000; //SUB $4, $4, $
0
assign block[19] = 32'b10110000000001110000000000000000; //BRN $7
assign block[20] = 0;
```

```
assign block[21] = 0;
assign block[22] = 0;
assign block[23] = 0;
assign block[24] = 32'b11100000000001010000000000000000; //LD $0, $5
// Loop End
// Update Index
assign block[25] = 32'b01010000110000110000000000000000; //INC $3, $3
assign block[26] = 32'b10000000000001100000000000000000; //J $6
assign block[27] = 0;
assign block[28] = 0;
assign block[29] = 0;
assign block[30] = 0;
assign block[31] = 0;
assign block[32] = 0;
```

Initial Register File

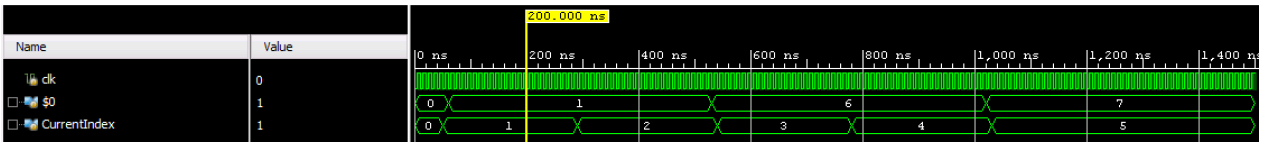
```
assign block[0] = 0;
assign block[1] = 0;  //start at data memory 0
assign block[2] = 5;  //5 elements
assign block[3] = 0;
assign block[4] = 0;
assign block[5] = 0;
assign block[6] = 6;  //Loop Address
assign block[7] = 25; //Update Index Address
assign block[8] = 27; //EndLoop Address
```

Data Memory

```
assign block[0] = 1;
assign block[1] = -4;
assign block[2] = 6;
assign block[3] = 0;
assign block[4] = 7;
```

To test our Max implementation, we used the setup above. The array contains `1, -4, 6, 0, 7` and the actual max value is expected to be stored in `Register 0`. From the waveform we can observe that `CurrentIndex` increases as the program moves to next loop and the content in `Register 0` is being updated with the new

max when needed.




# Assembly Code for Max Calculation

```
# Project MAX ASM

# Tian Zhang
# Matthew Koken

# Logic:
# Load memory
# Store First Element as Current Max
# Loop: Start Loop to go through the array
# Reach End? Jump Final
# Load element
# If Current is greater than Current Max, update Current Max
# Increase Index
# Jump to Loop

# Assume array start pointer being stored at $1, array size being stored at
# Assume all other registers start with value 0
# Current Max is stored at $0
# Index is stored at $3
# $4, $5 used to store temporary vars
# $6 stores address for Loop:
# $7 stores address for UpdateIndex:
# $8 stores address for Final:

Max:
    LD $0, $1       ;Load first element to current max
    INC $3, $3      ;Set start index to 1
Loop:
    SUB $4, $2, $3  ; Array Size - Current Index
    BRZ $8          ; No more elements, end the loop
    ADD $5, $1, $3  ; Adjust array offset
    LD  $4, $5      ; Load element at index
    SUB $4, $4, $0  ; Current Element - Current Max
```

```
    BRN $7          ; Current Element < Current Max, Don't do anything to c
    LD  $0, $5      ; Current Element > Current Max, Update Current Max
UpdateIndex:
    INC $3, $3      ; Update Current Index
    J $6            ; Go to the start of the loop
Final:
    NOP             ; Finish
```

# Estimation for Max Calculation

Duration of each stage

| IF/ID | ID/EX | EX/MEM | MEM/WB |
|---|---|---|---|
| 2ns | 1.5ns | 2ns | 2ns |

Based on the duration of above, the pipeline's cycle time should be 2ns.

Let `n` be number of items in array, `m` be number of items that's larger than current Max.

Total Cycles (based on the setup above):
6 + 6 * ( `n` +1) + 12 * `n` + 1 * `m` + 6 * `n` = m + 24n + 12 cycles

For this case, n = 5, m = 2, total cycles = 134 cycles

Total Time: 134 cycles * 2ns/cycle = 268 ns