COEN 122L: Computer Architecture Lab
Final Project Report

Justin Bracket & Tamir Enkhjargal

Winter 2020 - 2:15p Thursday

Abstract

Our task for the final project for COEN 122 was to take a 32-bit data-path structure and turn it into a working pipelined CPU. The data-path shows us how data flows from given instruction inputs into each module in the system. Though this seems like a very tough task, the lab was broken into individual tasks, and these smaller modules allowed us to become very familiar with each step of the project. Once completing the code for each module, we were able to go through the wires in the data-path and link up each module together. This task proved difficult due to some of the inputs and outputs requiring a different amount of bits. In the end, we were able to piece everything together to get a working CPU.

Project Design

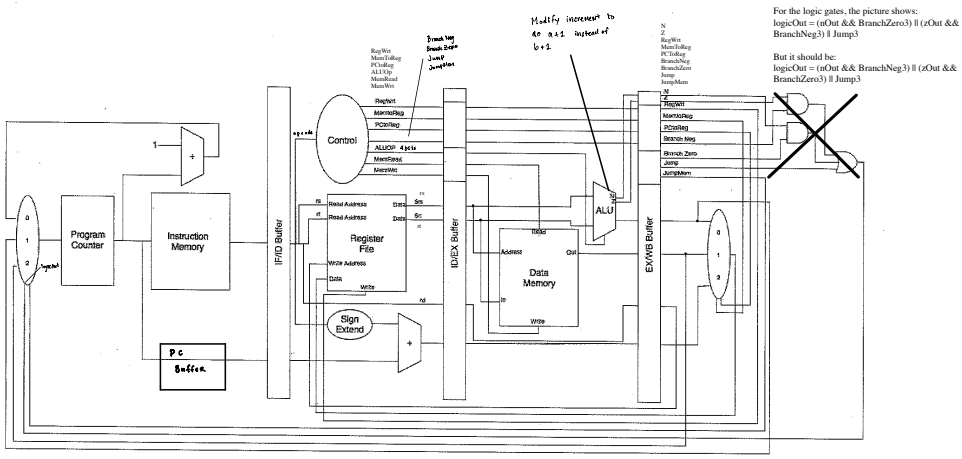

Fig. 1: Working Datapath of our Project

The system is controlled under the `Clock`, updating data and wires of the system on both the positive and negative edge of the `Clock`. Then, the system begins with the `Program Counter`, which tracks the address of the next instruction that the CPU is currently processing. The program counter tells which instruction to run in the `Instruction Memory`. There is also an `ALU` that increments the `PC`. From this, the `Program Counter` and `Instruction` is stored into the `IF/ID Buffer`. Next, the `Control Block` gets the instruction to send control lines over the CPU to set bits such as `MemWrt, MemRead, RegWrt, etc.`. The `Register File` at this time also takes the `rs` and `rt` bits from the 32-bit instruction and moves them through the `ALU` which sets potential `N` and `Z` flags. These outputs are then passed through the `ID/EX Buffer` which has the same role as the `IF/ID Buffer`. Then, all of the data is passed into the `Data Memory`. In conjunction with the `ALU2`, it is then stored in memory into the `EX/WB Buffer`. It goes through the final `ALU` where portions of the information are transferred back into the original `MUX` which allows us to increment, or add an offset to the `Program Counter`. This completes the cycle, which will continue until each instruction is completed in the `Instruction Memory`. Each buffer in the system allows the data to become synced and allows for pipelining.

```
At the start, set all bits to 0:
ALUOp       = 4'b0100; // Pass A
RegWrt      = 0;
MemToReg    = 0;
PCtoReg     = 0;
MemRead     = 0;
MemWrt      = 0;
BranchNeg   = 0;
BranchZero  = 0;
Jump        = 0;
JumpMem     = 0;

Keep everything at these standard bits (above) unless stated otherwise.

if(opcode == 4'b0000) //NOP keep everything @ zero

if(opcode == 4'b1111) // SavePC
    ALUOp       = 4'b0100;
    RegWrt      = 1;
    PCtoReg     = 1;

if(opcode == 4'b1110) // Load
    RegWrt      = 1;
    MemToReg    = 1;
    MemRead     = 1;

if(opcode == 4'b0011) // Store
    ALUOp       = 4'b0000;
    MemWrt      = 1;

if(opcode == 4'b0100) // Add
    ALUOp       = 4'b0000;
    RegWrt      = 1;

if(opcode == 4'b0101) // Increment
    RegWrt      = 1;
    ALUOp       = 4'b0001;
```

```
if(opcode == 4'b0110) // Negate
    RegWrt      = 1;
    ALUOp       = 4'b0010;

if(opcode == 4'b0111) // Subtract
    RegWrt      = 1;
    ALUOp       = 4'b0011;

if(opcode == 4'b1000) // Jump
    Jump        = 1;

if(opcode == 4'b1001) // BranchifZero
    BranchZero  = 1;

if(opcode == 4'b1010) //JumpMemory
    MemRead     = 1;
    JumpMem     = 1;

if(opcode == 4'b1011) //BranchifNegitive
    BranchNeg   = 1;

if(opcode == 4'b0001) //MAX
    RegWrt      = 1;
    MemRead     = 1;
```

Test Benchmarks

Assembly Code

```
assign instruction[0]  = 32'b11110000010000000000000011111111; // LDPC $1, 0x0FF
assign instruction[1]  = 32'b00000000000000000000000000000000; // NOP
assign instruction[2]  = 32'b00000000000000000000000000000000; // NOP
assign instruction[3]  = 32'b01010000100000010000000000000000; // INC $r2, $r1
assign instruction[4]  = 32'b01100000110000010000000000000000; // NEG $r3, $r1
assign instruction[5]  = 32'b11110010100000000000000000001000; // LDCPC $r10, label1 = 8
assign instruction[6]  = 32'b00000000000000000000000000000000; // NOP
assign instruction[7]  = 32'b00000000000000000000000000000000; // NOP
assign instruction[8]  = 32'b10110000000010100000000000000000; // BRN $10
assign instruction[9]  = 32'b00000000000000000000000000000000; // NOP
assign instruction[10] = 32'b00000000000000000000000000000000; // NOP
assign instruction[11] = 32'b00000000000000000000000000000000; // NOP
assign instruction[12] = 32'b01010000100000010000000000000000; // INC $r2, $r2
assign instruction[13] = 32'b00110000000000001000001000000000; // ST  $r1, $r1
assign instruction[14] = 32'b00000000000000000000000000000000; // NOP
assign instruction[15] = 32'b00000000000000000000000000000000; // NOP
assign instruction[16] = 32'b11100001000000010000000000000000; // LD $r4, $r1=
assign instruction[17] = 32'b00000000000000000000000000000000; // NOP
assign instruction[18] = 32'b00000000000000000000000000000000; // NOP
assign instruction[19] = 32'b01000001010000010000100000000000; // ADD $r5, $r1
assign instruction[20] = 32'b01110001100001000000010000000000; // SUB $r6, $r4
assign instruction[21] = 32'b11110010110000000000000000001000; // LDPC $11, label2 = 8
assign instruction[22] = 32'b00000000000000000000000000000000; // NOP
assign instruction[23] = 32'b00000000000000000000000000000000; // NOP
assign instruction[24] = 32'b10010000000010110000000000000000; // BRZ $r11
assign instruction[25] = 32'b00000000000000000000000000000000; // NOP
assign instruction[26] = 32'b00000000000000000000000000000000; // NOP
assign instruction[27] = 32'b00000000000000000000000000000000; // NOP
assign instruction[28] = 32'b01010000100000010000000000000000; // INC $r2, $r2
assign instruction[29] = 32'b10100000000000001000000000000000; // JM $r1
assign instruction[30] = 32'b00000000000000000000000000000000; // NOP
assign instruction[31] = 32'b00000000000000000000000000000000; // NOP
assign instruction[32] = 32'b00000000000000000000000000000000; // NOP
assign instruction[33] = 32'b10000000000000001000000000000000; // 0xFF: J $r1 =$
```

CPU Time Data

Our system has 15 non-`NOP` instructions, which are each 4 cycles, however, in the code, the two lines `INC $r2, $r2` should not be executed, meaning that we only have 13 non-`NOP` instructions *actually executed*. This results in 52 cycles, and added on 19 `NOPs`, which are a single cycle each, leads to a total of 71 CPU clock cycles. Theoretically, the amount of time required to get through each step in the data-path and instruction memory should be 710ns, since we have a clock size set to 10ns. We can confirm this with our Test Benchmark Waveform, since the last time any data that was updated was exactly at 710ns.