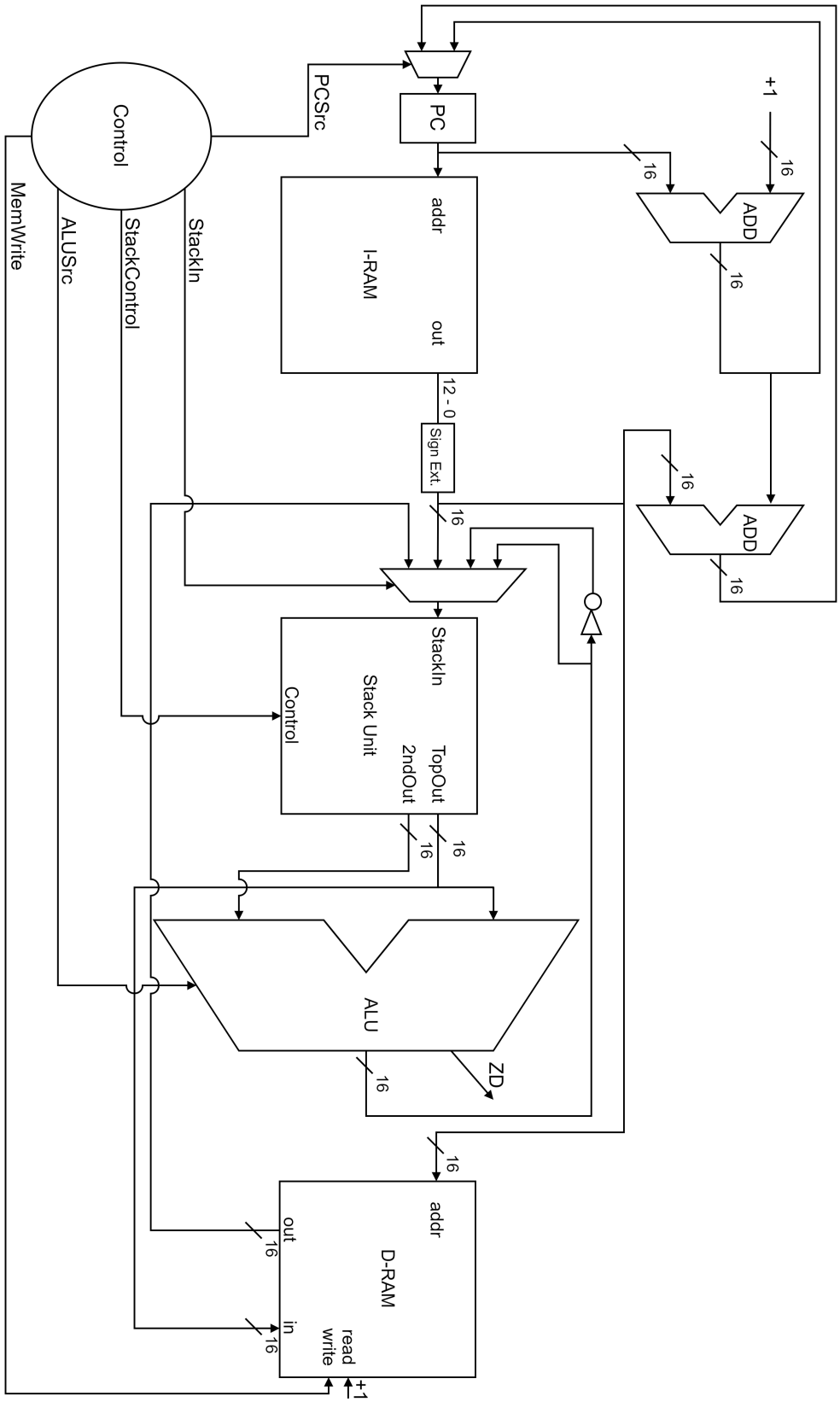
**Van with a Plan (vSaw2) Processor Design:**



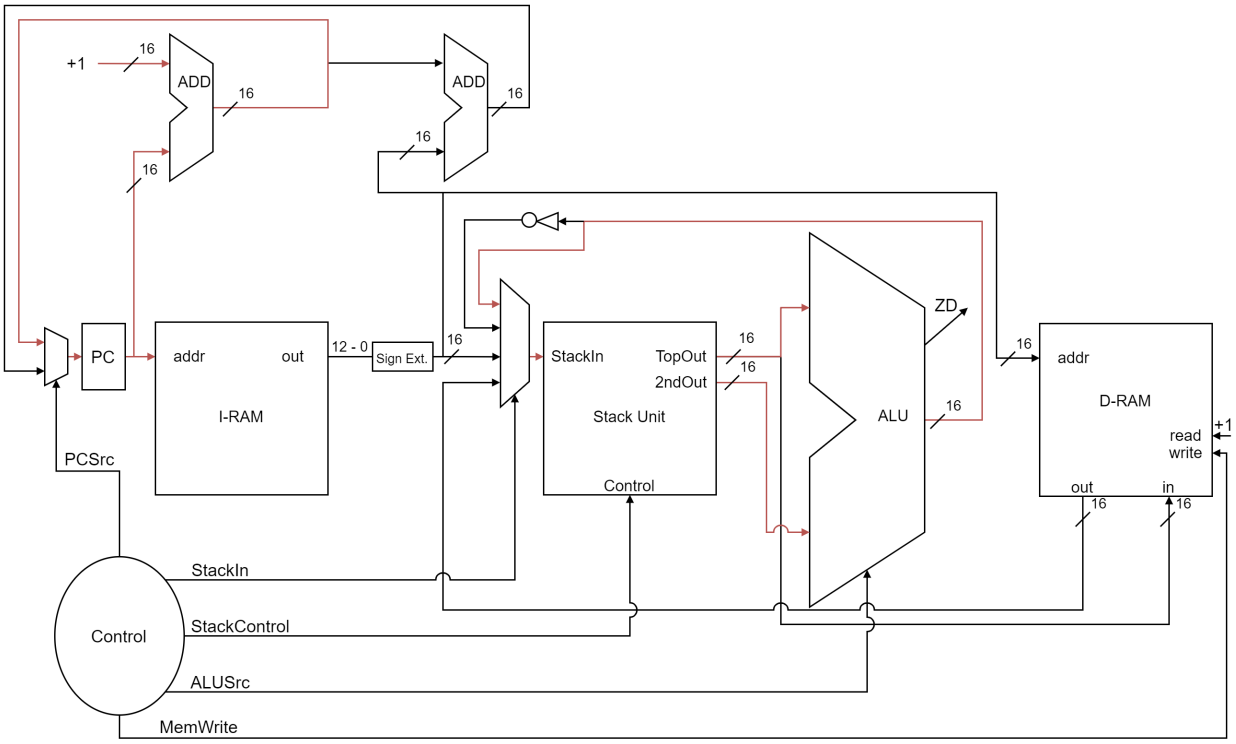
**vSaw2 Control Table:**

| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| add | 0 | 00 | 100 | 010 | 0 |
| nand | 0 | 01 | 100 | 000 | 0 |
| push | 0 | 11 | 010 | 000 | 0 |
| pop | 0 | 00 | 001 | 000 | 1 |
| pushi | 0 | 10 | 010 | 000 | 0 |
| beq | 0/1 | 00 | 000/011 | 110 | 0 |
| noop | 0 | 00 | 000 | 000 | 0 |

For the beq operation, the PCSrc will be the result of the zero detect form the ALU. Depending on the value of the zero detect, the StackControl signal will either be 000 for a ZD of 0, or 011 for a ZD of 1.

---

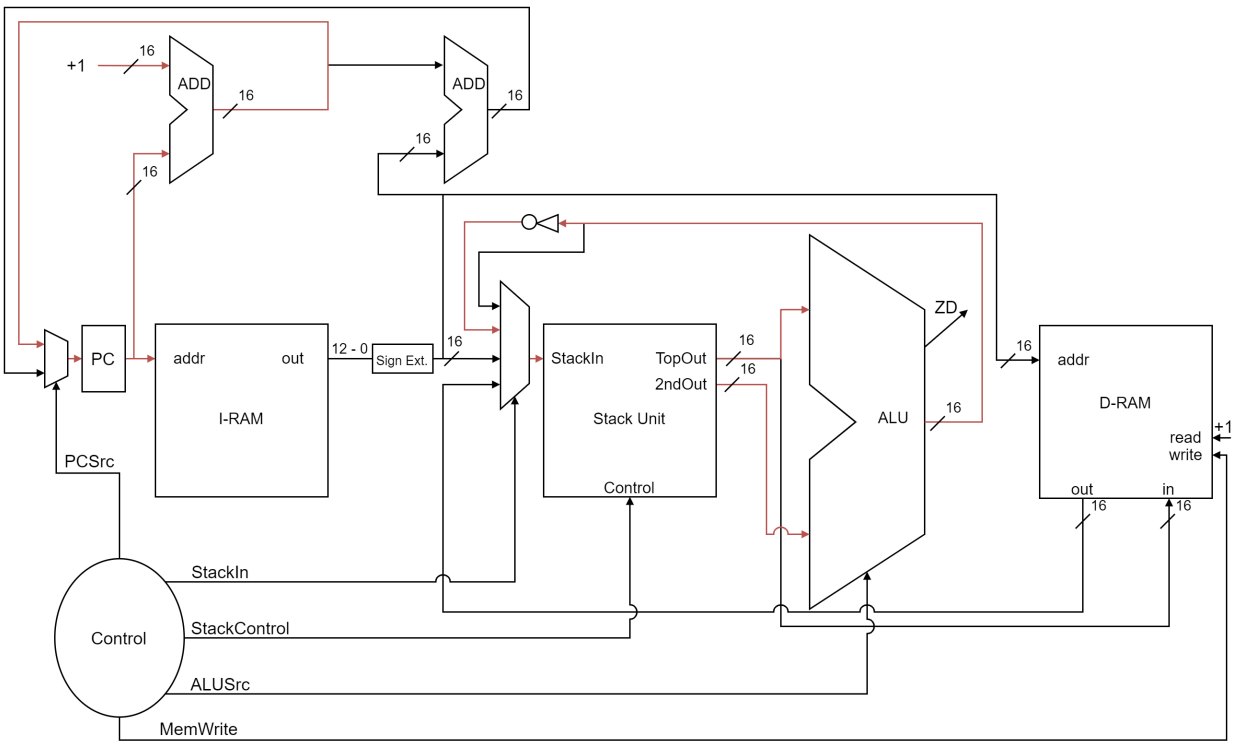
**ADD Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| add | 0 | 00 | 100 | 010 | 0 |

For the add operation we need to take the top two values from the stack unit, combine them in the ALU with an add function, and then return the output back to the stack unit to be placed on the stack. We give PCSrc 0 for PC+1. We give StackIn 00 to select our ALU output. We give StackControl 100 to pop the top two then add StackIn to the top of the stack. We give ALUSrc 010 for an add operation. We give MemWrite 0 so we don't change values in memory.

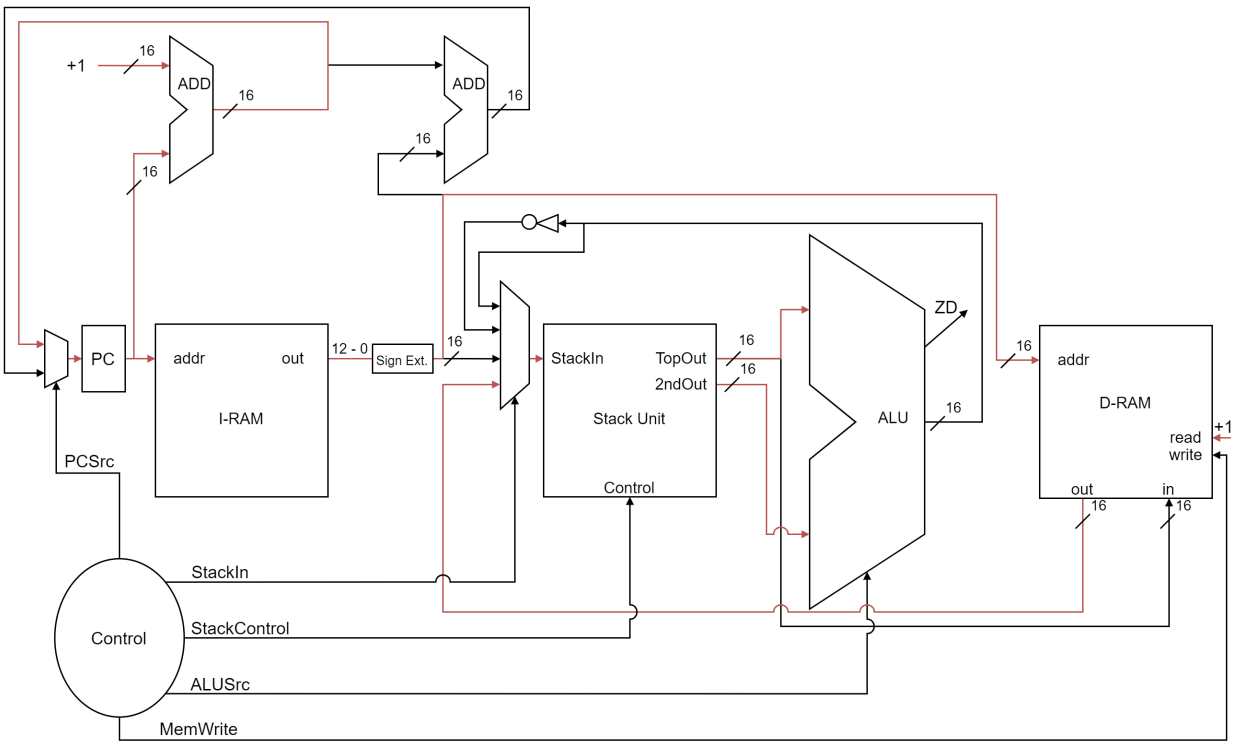
**NAND Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| nand | 0 | 01 | 100 | 000 | 0 |

For the nand operation we need to take the top two values from the stack unit, combine them in the ALU with an and function, and then return the output back to the stack unit with our 16 bit answer going through not gates to get the complement, which is placed on the stack. We give PCSrc 0 for PC+1.We give StackIn 01 to select the complement of the ALU output. We give the StackControl 100 to pop the top two then add the StackIn to the top of the stack. We give ALUSrc 000 for an and operation. We give MemWrite 0 so we don't change values in memory.

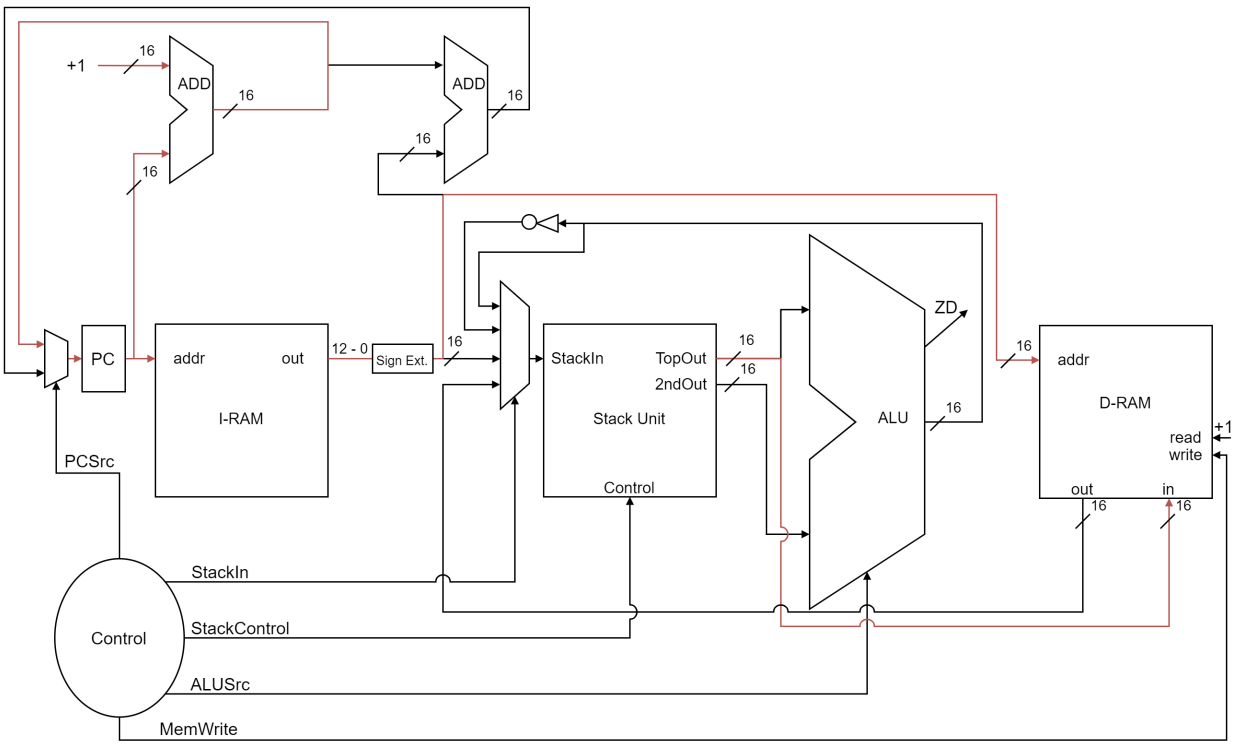
**PUSH Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| push | 0 | 11 | 010 | 000 | 0 |

For the push operation we need to take sign extended z value from our instruction, feed it to the address selection on the data memory, and return the value at that address back to the stack unit to be placed on the stack. We give PCSrc 0 for PC+1. We give StackIn 11 to select the output from the data memory. We give StackControl 010 to push the StackIn value onto the stack. We give the ALUSrc 000 since it is not needed. We give MemWrite 0 so we don't change values in memory.

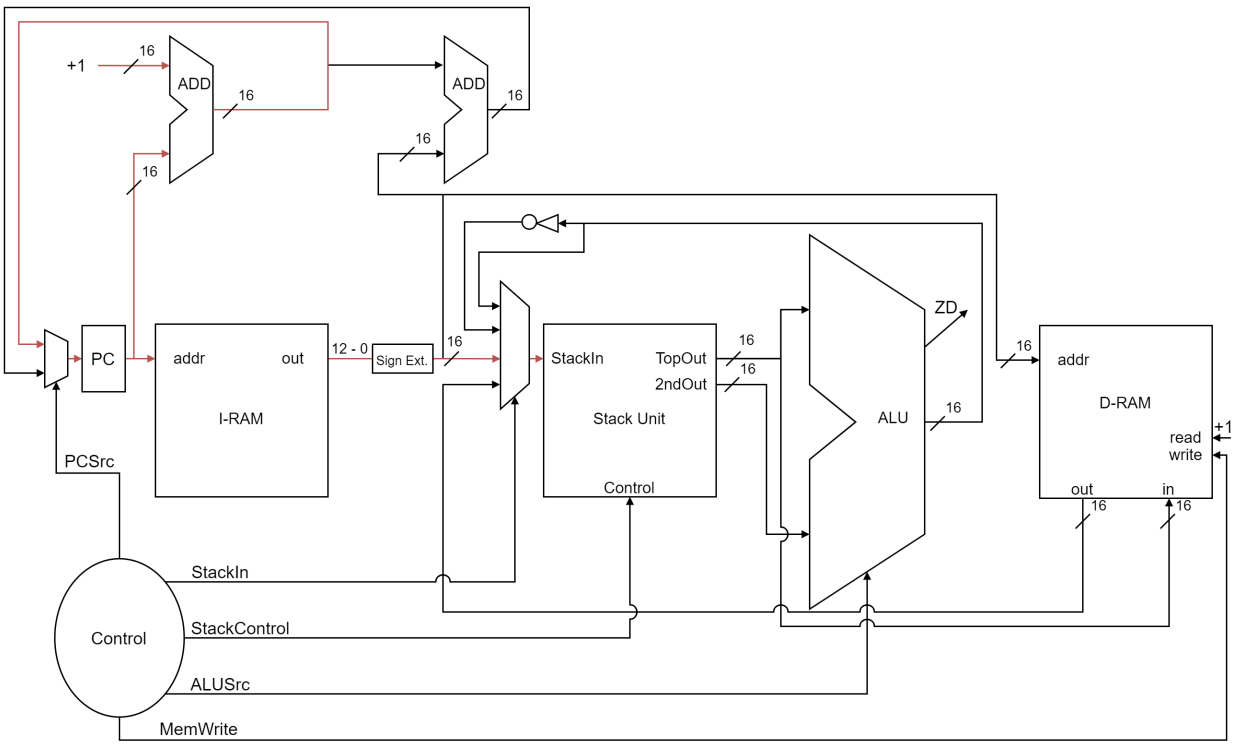
**POP Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| pop | 0 | 00 | 001 | 000 | 1 |

For the pop operation we need to take the value off of the top of the stack and put it into data memory at an address specified by our sign extended Z value from our instruction. We give PCSrc 0 for PC+1. We give StackIn 00 since it is not needed. We give StackControl 001 to pop off the top of the stack. We give the ALUSrc 000 since it is not needed. We give MemWrite 1 to write the input data to the data memory at the address specified by our Z value.

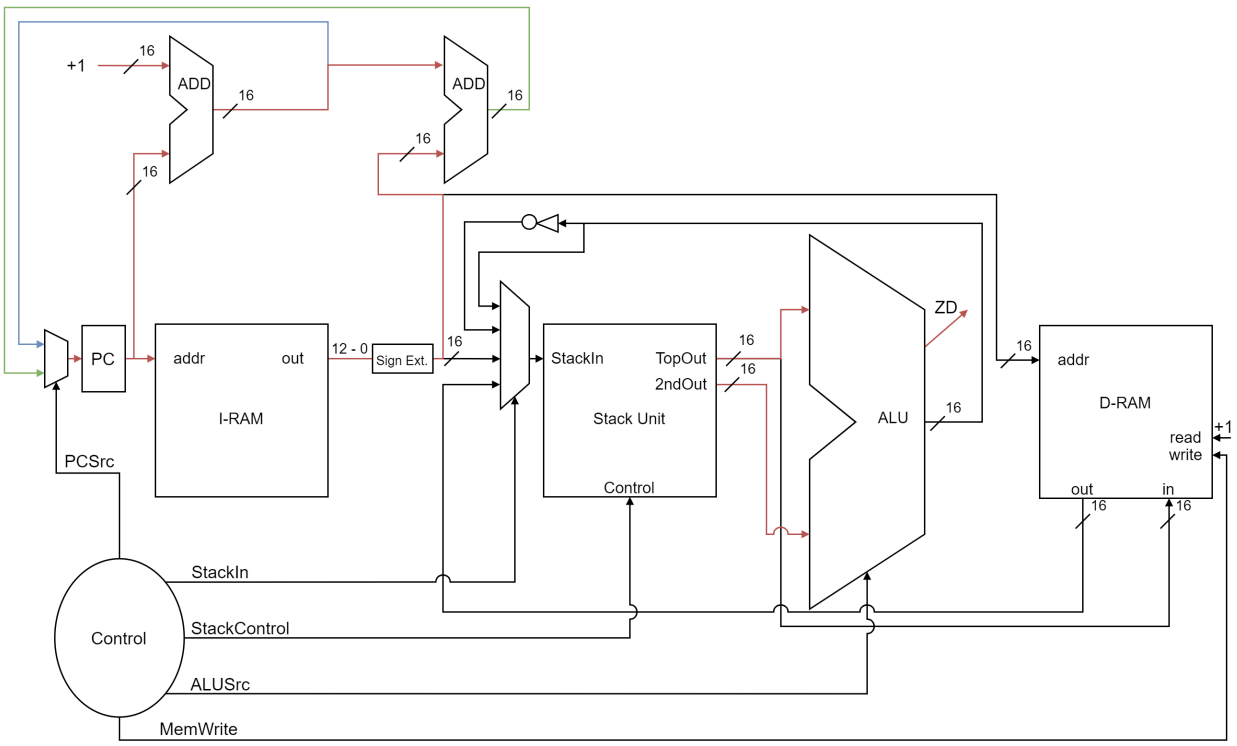
**Push Immediate Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| pushi | 0 | 10 | 010 | 000 | 0 |

For the push immediate operation we need to take sign extended z value from our instruction and give it to the stack unit to be placed on the stack. We give PCSrc 0 for PC+1. We give StackIn 10 to select the sign extended Z value from our instruction. We give StackControl 010 to push the StackIn value onto the stack. We give the ALUSrc 000 since it is not needed. We give MemWrite 0 so we don't change values in memory.

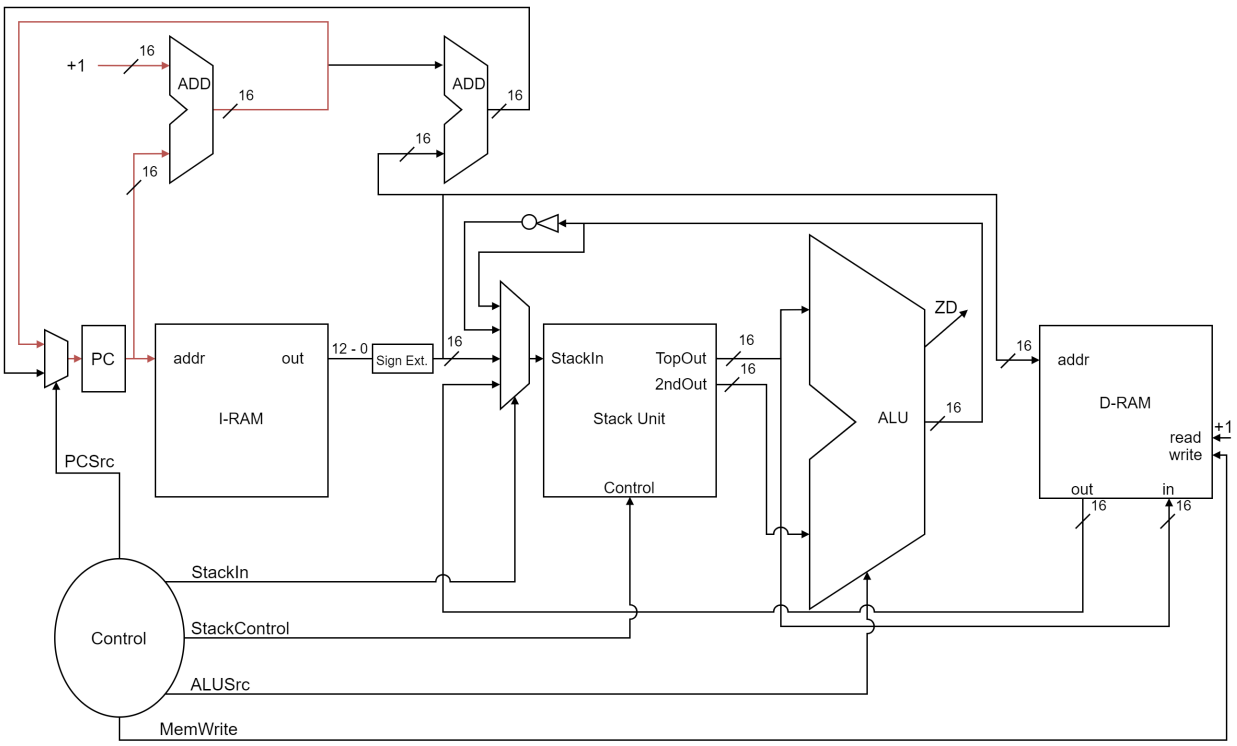
**Branch On Equals Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| beq | 0/1 | 00 | 000/011 | 110 | 0 |

For the branch on equals operation we need to take the top two values from the stack, subtract one from the other, and use the zero detect from that operation to either set the PC to PC+1, or set it to PC+1+Zvalue. We give PCSrc either 0 for a ZD of 0 (blue path) to select PC+1, or a 1 for a ZD of 1 (green path) to select PC+1+Z. We give StackIn 00 since it is not needed. We give StackControl either 000 for a ZD of 0 to do nothing, or 011 for a ZD of 1 to pop the top two values off the stack. We give the ALUSrc 110 for a subtract operation to check for equality using the zero detect. We give MemWrite 0 so we don't change values in memory.

**No Operation:**




| Instruction | PCSrc | StackIn | StackControl | ALUSrc | MemWrite |
|---|---|---|---|---|---|
| noop | 0 | 00 | 000 | 000 | 0 |

For no operation we simply need to make sure the stack and the data memory do not change, while setting PC to PC+1. We give PCSrc 0 for PC+1. We give StackIn 00 since it is not needed. We give StackControl 000 for no operation. We give the ALUSrc 000 since it is not needed. We give MemWrite 0 so we don't change values in memory.

**Design Justification:**

The Van with a Plan (vSaw2) processor I have designed works by executing instructions from a dedicated instruction memory module, with each instruction being only of a single type, and therefore all processed in the same manner. The way each instruction is handled is the first 3 bits for the op code are striped off and used to inform the control logic, while the remaining 13 bits (our Z value) are sign extended and used for operations that require them.

The reason we only ever need to use the 12-0 bits from the instruction in an all or nothing fashion is because there is only one format of instruction. Since we only have one type of instruction, there is no need to handle different sections of bits differently depending on the instruction, and we can simply take those same 13 bits every time, and everything else will be handled by the control logic.

We always sign extend the 13 bit Z value because there is no use for a 13 bit number in a 16 bit machine. No matter what operation we are doing, we will need a signed extend version of the number, so feeding it directly into the sign extender makes it easy to then simply use the sign extended number wherever it is needed.

The TopOut from the Stack Unit needs to be used for both ALU operations and being written to memory, so we have it branch to both locations, while 2ndOut goes straight into the ALU, since it only is ever used for operations that go through the ALU.

I have chosen to tie the read control bit on the data memory to 1 so we are always reading. This is correct because there is no reason in our design to not always be reading, since the bits will be selected for by the StackIn control logic if they are needed, and will be ignored if they are not. Choosing to not output these bits is irrelevant to our design, and by tying this control bit to 1, it allows us to simplify our control logic and remove an unnecessary column from our control table.

The MemWrite bit allows us to select to write to memory. The fact that we are always reading will not affect our ability to write since always reading will simply output the current value in the spot in memory we are about to write to, and then at the end of the clock cycle, that value

will be overwritten by the value on the input to the data memory.

The output from the ALU only goes back into the Stack Unit. This is correct because the output from the ALU is never written into memory, and it's not needed for branching since we only need the zero detect from the ALU to branch correctly. The jump calculation is handled by a separate adder that takes in the PC+1 and the sign extended Z value to calculate the offset for the branch.

It is necessary to have the ALU and the two adders. This is because for the branch on equals function, we need the ALU to perform the check on the branch condition, and we need the other two adders to produce the two possible next PC values for us to select between depending on the result of the zero detect on the ALU. It is not possible to remove the ALU or either of the two adders and still have a branch on equals functionality.

This design uses a minimal number of muxes, control bits, and other hardware to maximize the cost efficiency of the vSaw2 chip. It has 5 control bits that correctly implement the 8 operations of the chip meeting the required functionality of our ISA.