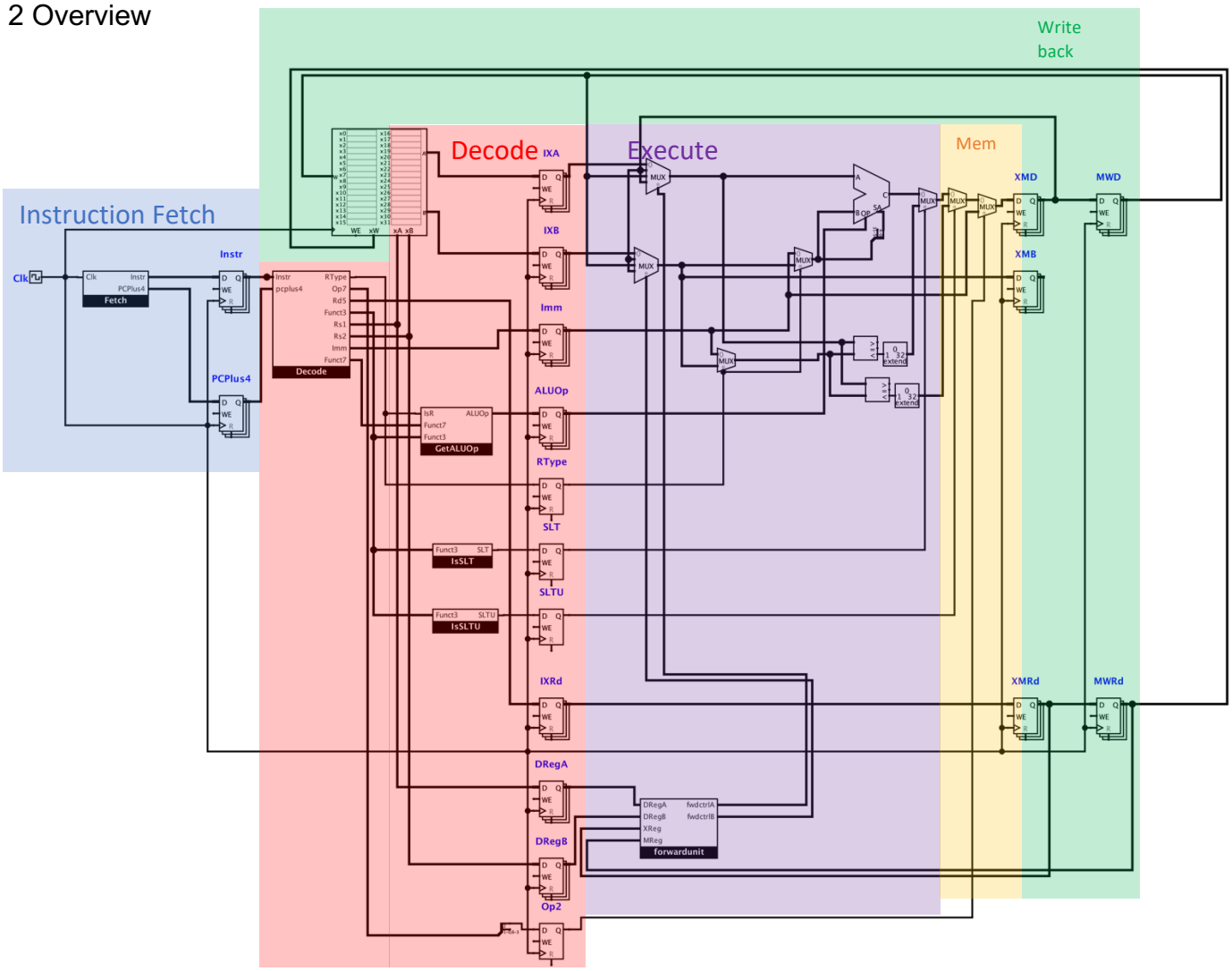
Design Documentation P2

Kevin Klaben (kek228) Jonathan Tong (jt572) February 28, 2019

1 Introduction

- **Purpose:** The purpose of this document is to act as a reference to the processor and explain our design choices
- **Scope:** The scope of this document is everything in a RISC-V pipelined processor that we've covered in class
- **Intended audience:** The intended audience is anyone with at least some basic knowledge of the concepts behind building a processor
- **Summary:** This will provide an overview of each of the components in our processor

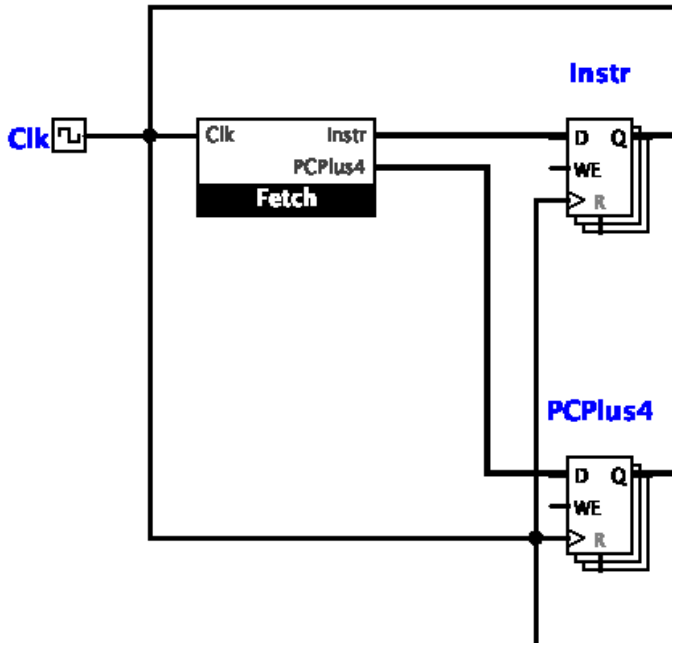
2 Overview



The overall pipelined RISC-V processor design can be seen above. Each stage of the processor is highlighted and will be outlined below. The Pipeline registers at the end of each stage are included as part of that diagram, thus fetch/decode registers are included as part of the Fetch stage and so on.

3 The Fetch Stage

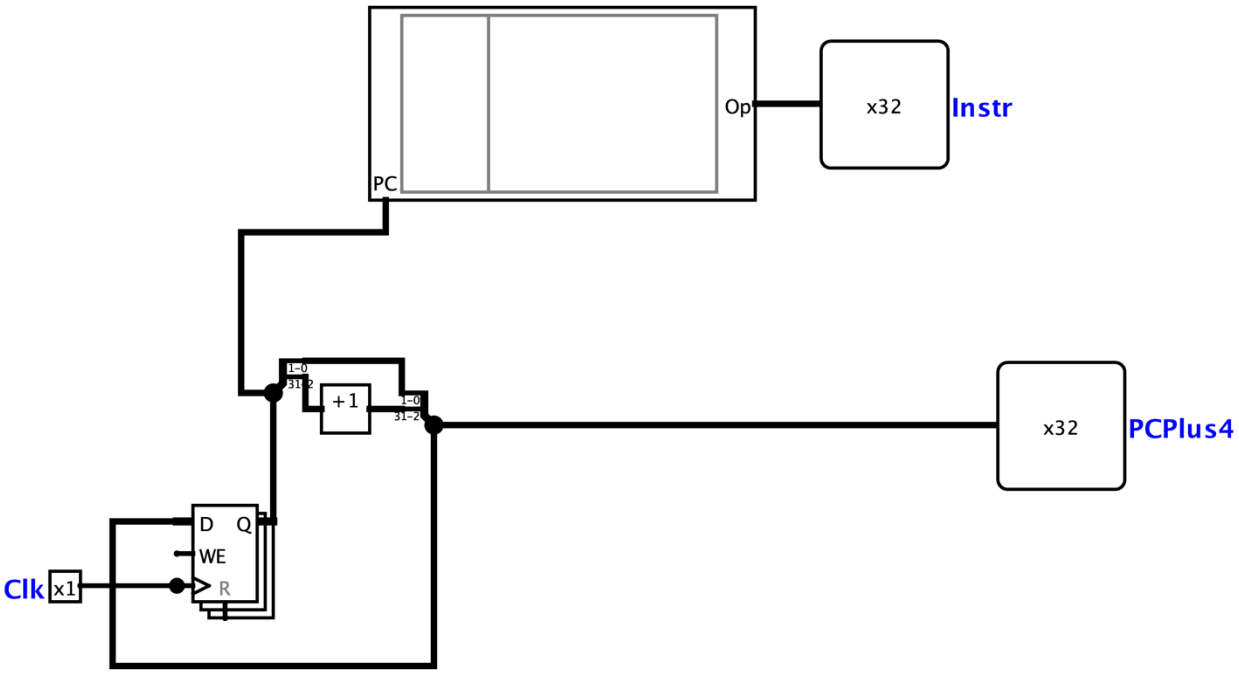
3.1 Circuit Diagram



As can be seen above is the Fetch stage of the processor. There are two pipeline registers going out of this stage. Instr holds the 32bit instruction that was fetched during this clock cycle. PC+4 will store the current PC of this clock cycle plus 4. The clock is used throughout all the stages and this continues on throughout the processor. A breakdown of the Fetch sub-circuit can be seen below.

Instruction Memory holds the instructions to be completed, the pc outputs the address of the next instruction to be completed. Then the instruction is taken from the rom address and output. The PC is incremented by 4 and then muxed with a pc-reg, and pc-rel based on the control bits of the instruction in the decode stage. The selected pc is the pc which is to be output on the next clock cycle.

3.1.1 Sub-Circuit Fetch



The only input to this sub-circuit is Clk which is the clock. The clock is used to increment the PC through the register seen here which stores the PC. Each clock cycle the PC comes out of this register and is input to the Program-ROM which contains all the instructions that were loaded in by the user. This Program-ROM then outputs the instructions stored at PC which then goes to output Instr. The PC is incremented by 4 each clock cycle and this is implemented using splitters and a +1 incrementor. The 30 most significant bits of PC are incremented by 1 each time as this will be equivalent to increasing the entire 32 bit PC by 4. This incremented PC is output to PCPlus4 and is also stored back in the register to be used as the PC for the next clock cycle.

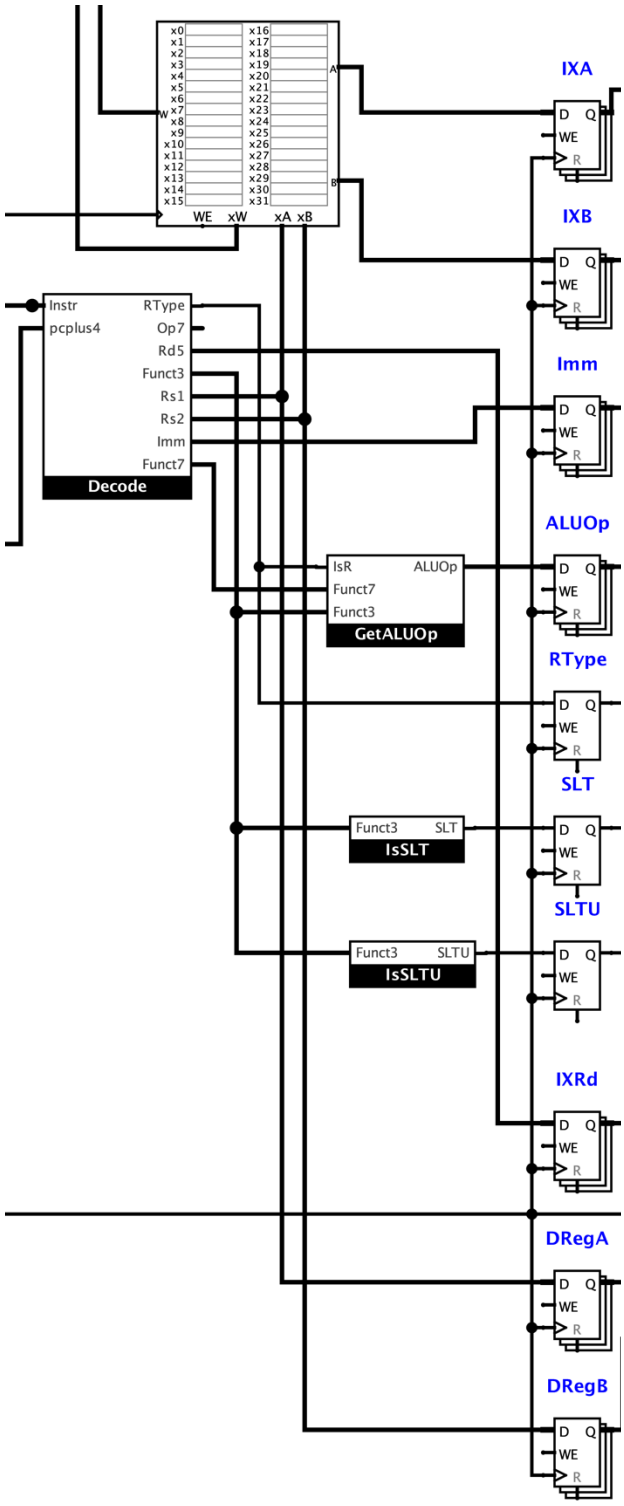
3.2 Correctness Constraints

The objective of the Fetch Stage is to use PC to index Program Memory and output an instruction as well as increment and maintain the PC.

3.3 Testing

4 The Decode Stage

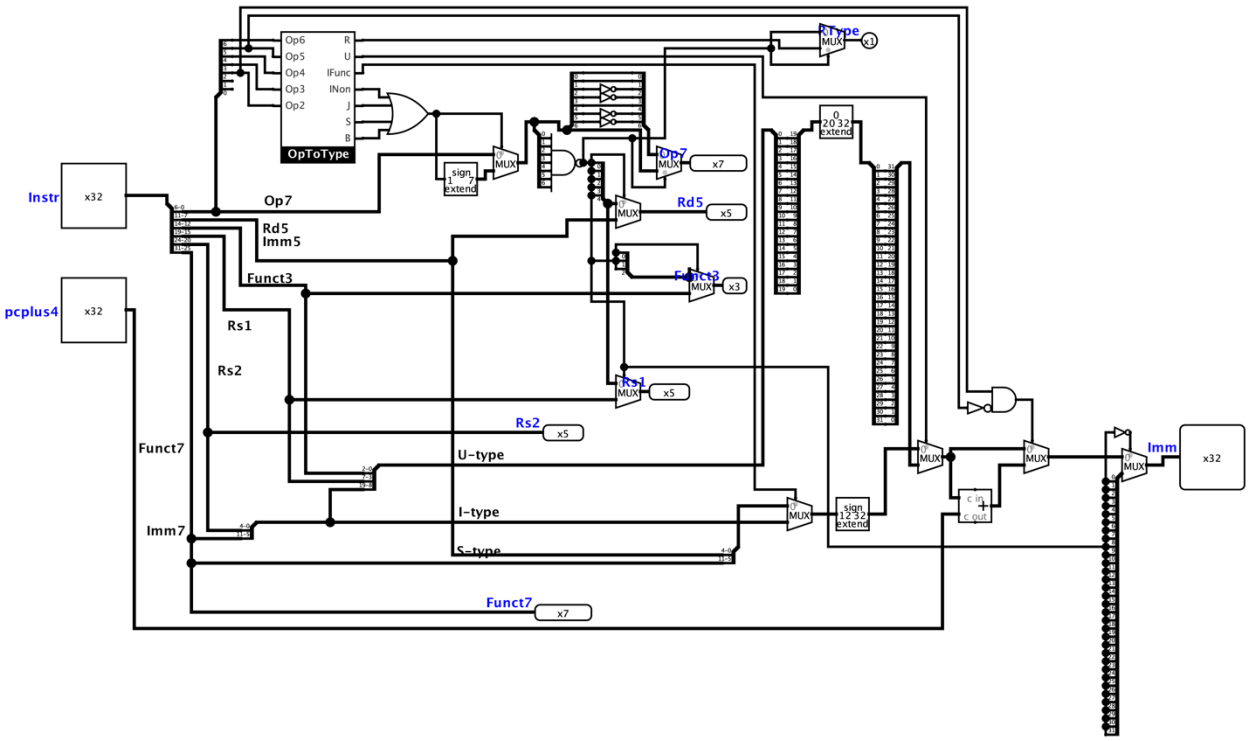
4.1 Circuit Diagram



The decode stage takes in an instruction and the pc+4 and uses them as input to the Decode sub-circuit. The Decode sub-circuit will output several control bits. RType is 1 if the instruction is an RType and 0 otherwise. Rd5 is the register that the instructions operation will be written to. Funct3 is the 3 bit opcode from the instruction which is used as control bits to be used in determining the ALU operation. Rs1 and Rs2 are the registers to be accessed form the register file to get the values to be used in the ALU.

Imm is the 32 bit immediate to be used in operations. Funct7 is the 7 most significant bits of the instruction to be used as controls. Several of these outputs from Decode may be nonsense as different instructions have different data within them. This is ok as the correct information will be contained and using control the nonsense will have no effect on the actual instruction which must be done. In addition, there is a register file which takes in and stores information form past operations and stores it in the correct register as well as it will take in Rs1 and Rs2 as xA and xB and output the values stored at those addresses. In addition, there are several sub-circuits which will be outlined below which are used to further decode which operation is to be done. Many of these bits are stored in the pipeline registers seen in the diagram to be used in later stages.

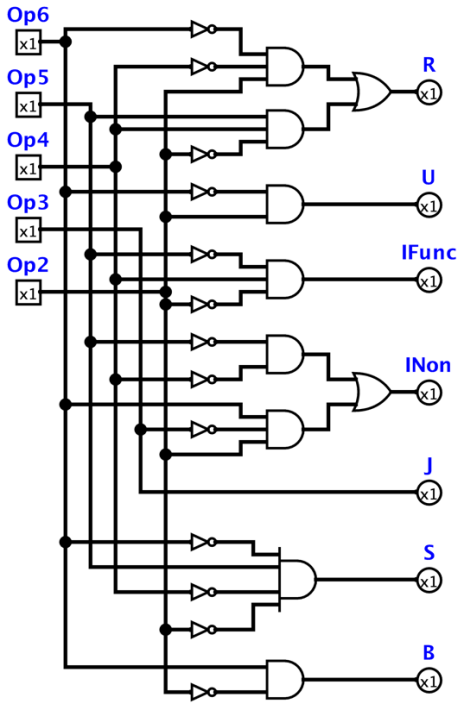
4.1.1 Decode



The inputs to this sub-circuit are Instr and pcplus4 as previously explained. The instruction is split into pieces based on which how the information for an instruction is stored. The first 7 bits are fed into OpToType which will output what type the operation is. Using the outputs from OpToType RType is calculated by muxing between the original signal or in the case where the operation is from table B and the output is then 0 as in such a case the instruction is treated as a Nop and thus ADDI x0 x0 and 0 which is not an RType. Similar controls are put in place to ensure that instruction from Table B will be treated as ADDI x0 x0 0. The 7 bit NAND gate is used to create this Nop detection and then the splitters with several not gates in the middle is to create the ADDI instruction code in that case. Rd5 is similarly muxed between the original signal and 5 zeros in the case of a Table B instruction. The same goes for Rd5, Funct3, and Imm. The large splitters are used to flip and bit extend with zeros the right side of

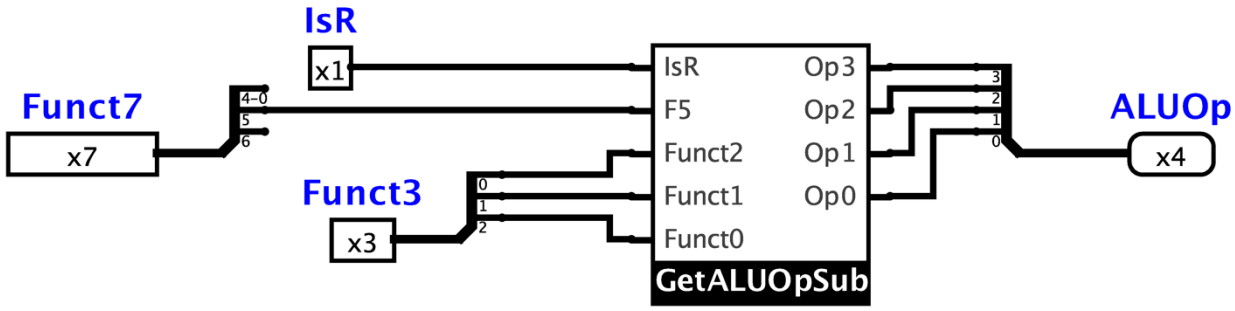
immediate for U type instructions. In most cases the decode stage will work to split off the different control bits for an instruction and output them, and in the case of a Table B it will be the Nop instruction.

4.1.1.1 OpToType



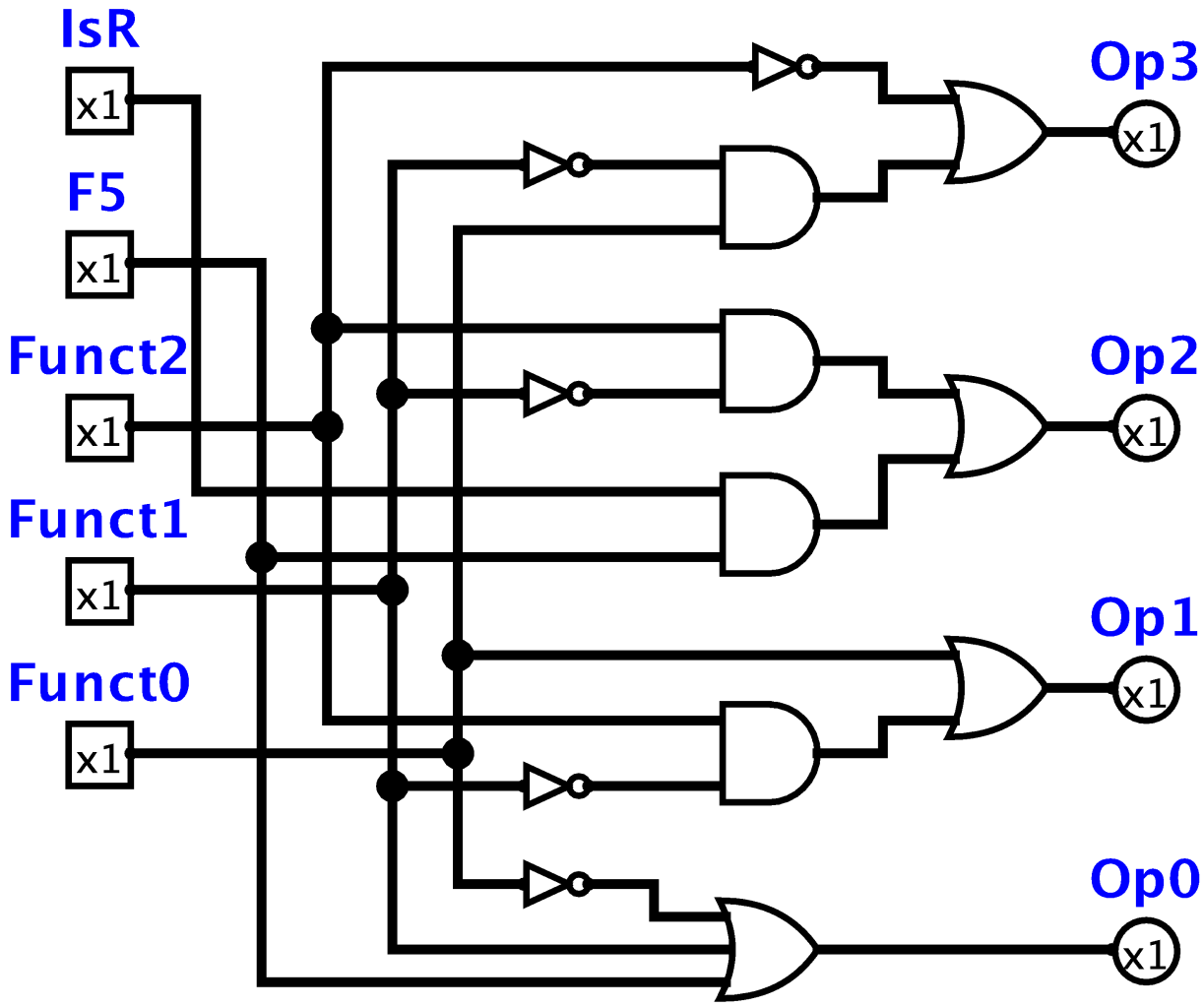
OpToType simply takes in 5 bits from the instruction and decodes them to what type the instruction is and outputs them.

4.1.2 GetALUOp



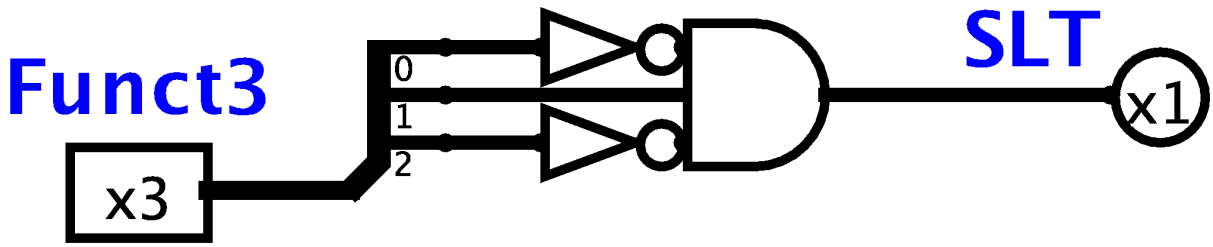
GetALUOp takes in the 6th bit of Funct7, Funct3 and IsR and input them to a decoding subcircuit. This bits are used as controls to develop the ALUOp which can be directly input to the ALU.

4.1.2.1 GetALUOpSub



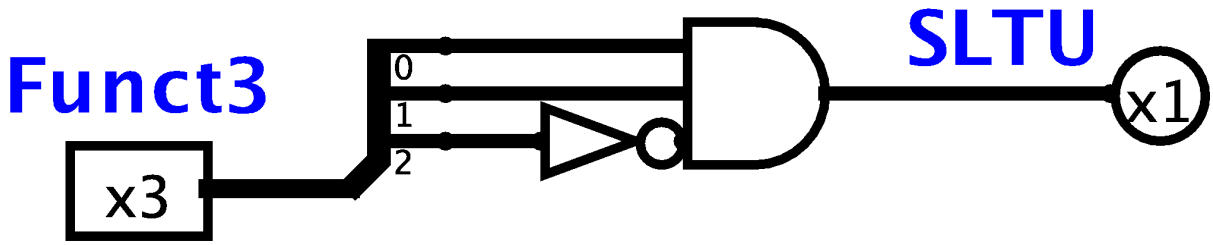
GetALUOpSub is a decode which takes in the control bits from the instruction code and will output the 4 bit op code that the ALU can understand what operation is called for.

4.1.3 IsSLT



This will output whether or not Funct3 calls for a SLT operation.

4.1.4 IsSLTU

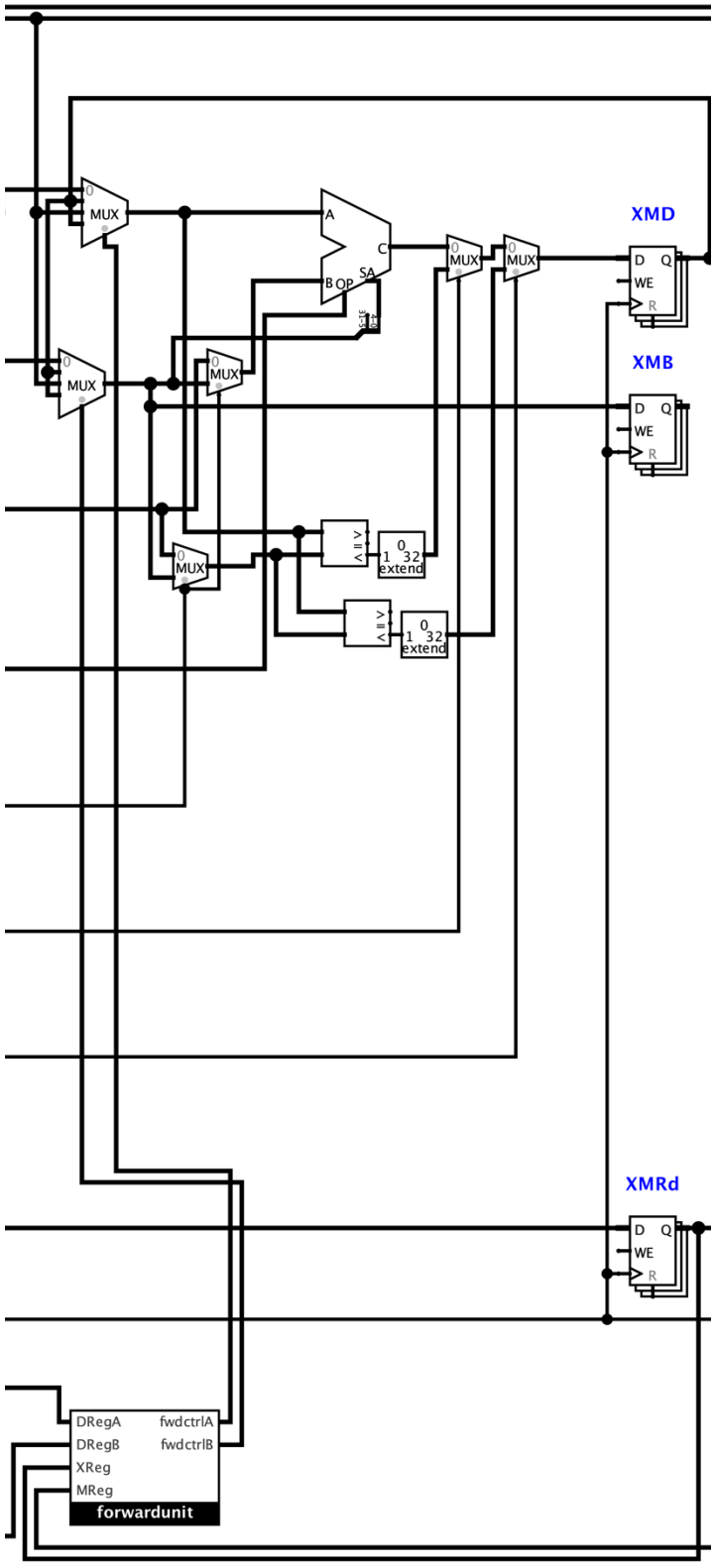


This will output whether or not Funct3 calls for a SLTU operation.

4.2 Correctness Constraints

The main goal of the decode stage Decode instruction, generate control signals, read register file

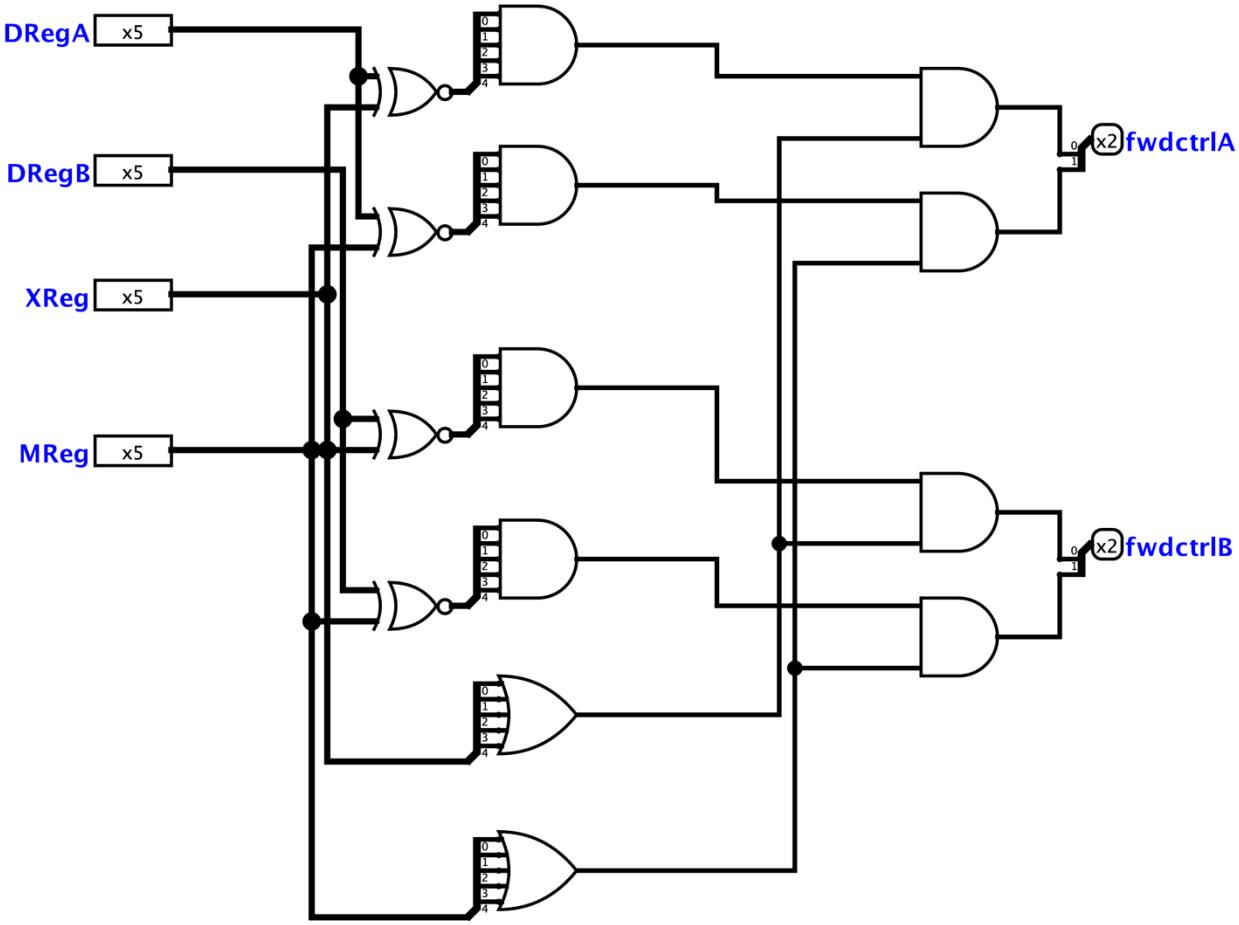
5 The Execute Stage



The execute stage will be used to handle operations with the ALU. Given an immediate value, register A, register B, and the PC, the control bits will decide which of the inputs

will go into the ALU. Control bits from past stored values are used to mux between B and imm and the slt and sltu operations and the ALU output. The forwardunit is used to determine whether or not forwarding is necessary if there is a data hazard and will mux between the original values for A and B gotten in the decode stage and values pulled from the stages ahead in there case where there is a data hazard.

5.1 forwardunit



The forwardunit is used check the conditions of whether or not there is a data hazard and pick which signal is necessary if there is a data hazard. The conditions to be checked for the pairing of each DRegA and DRegB with each XReg and MReg are to check if they are equal and if they are not x0. In the case that these are both true then 1 is output else 0 is output. The first bit for fwdctrl is whether there is a data hazard from write back and the second is whether or not there is a data hazard from memory. In the case where there is a data hazard in both, the closer value from memory is selected and used for the ALU.

5.2 Correctness Constraints

Perform ALU operation, and decide whether or not the value output was needed or if it is nonsense and not called for by the instruction.

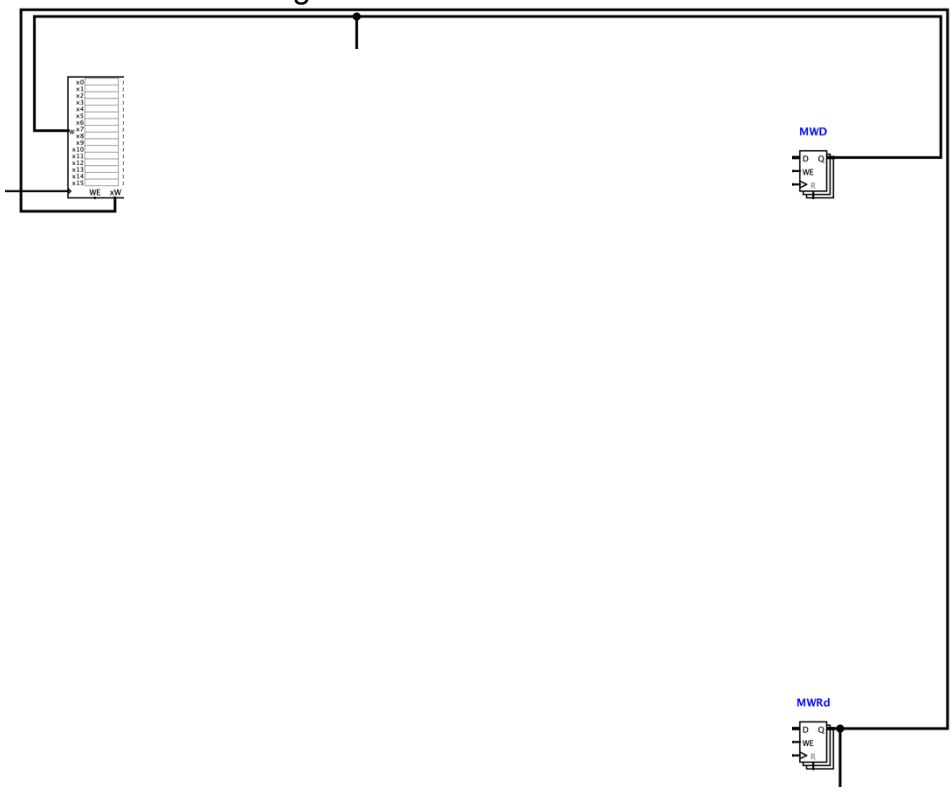
6 The Memory Stage

At this point the memory stage is simply a pass through and does nothing.

6.2 Correctness Constraints

At this point the memory stage is simply a pass through and does nothing.

7 The Writeback Stage



7.2 Correctness Constraints

The Writeback Stage simply takes the value from MWD and stores in in the register file at MWRd.

8 Testing

Our testing includes forwarding testing on both A and B coming from both memory and writeback. In addition, we tets each operation to be sure that if it is on Table A it does the correct operation and if it is on Table B it behaves as a Nop.