CS 3410 Project 1

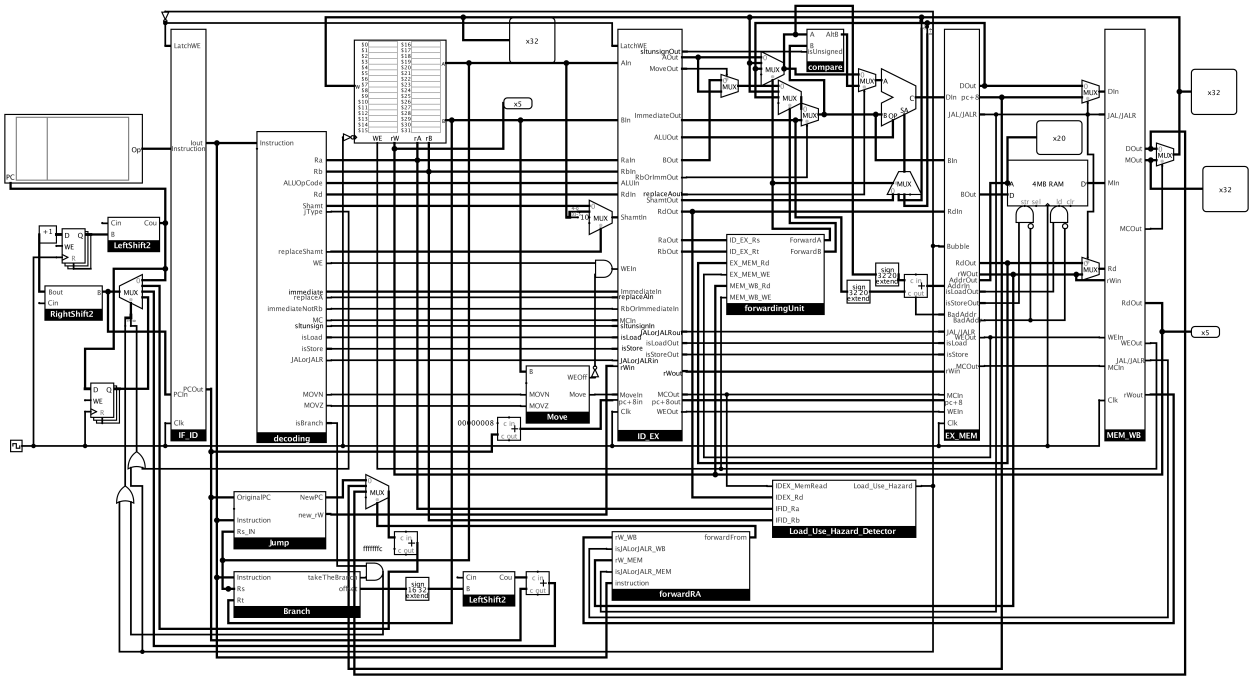
Alex Leong

March 15, 2016

Overview

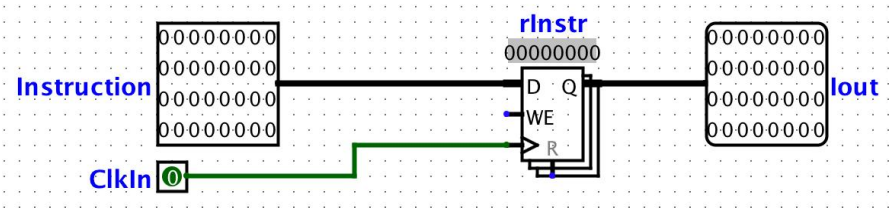
In project 1 I implemented my design for a five stage pipelined MIPS processor into Logisim. This processor is capable of handling data and control hazards, and it executes the following MIPS instructions:

Immediate arithmetic	`ADDIU, ANDI, ORI, XORI, SLTI, SLTIU`
Register arithmetic	`ADDU, SUBU, AND, OR, XOR, NOR, SLT, SLTU`
Move	`MOVN, MOVZ`
Shifts (constant and variable)	`SLL, SRL, SRA, SLLV, SRLV, SRAV`
Immediate load	`LUI`



Stage 1: Instruction Fetch

In the instruction fetch stage, I implemented my program rom with a register which increments by 1 each clock cycle. To achieve an increment of 4 into the PC input, the PC register is left shifted by 2 before entering the program rom. The program instruction is then passed into the first pipeline registers, IF_ID. This separator has two inputs and one output: instruction[32] and clock, and Iout[32]. No control signals needed.

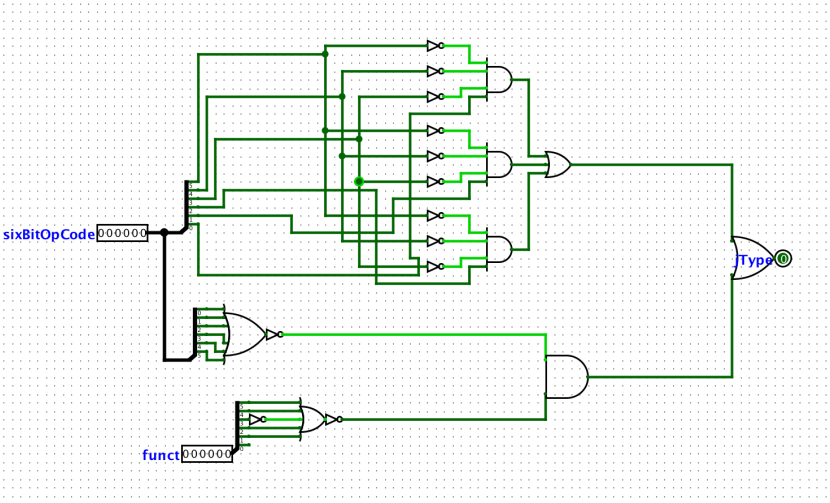


Stage 2: Instruction Decode

Some instructions are very straightforward and and be directly mapped to one of the ALU operations, while others need some manipulation to be “translated” to a type of ALU operation. Most of the decoding stage is completed within the sub-circuit named “decoding”, while some completed outside it.

How every types of instruction in the Table A are actually implemented are explained as follows:

1. J-type instructions or any Load/Store instructions
   Use a sub-circuit to detect the existence of J-type instructions.

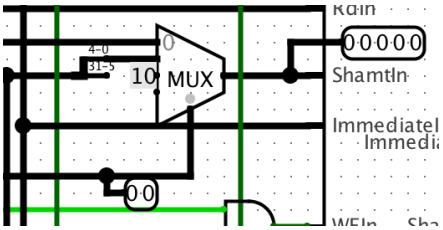


If the 32-bit instruction is determined as J-type instruction, OR the most significant bit is 1 (meaning that it is Load/Store), then WE is automatically set to 0;

2. The six shifting-related instructions (SLL,SRL,SRA,SLLV,SRLV,SRAV)

For SLL, SRL and SRA, the matter is simple. Their corresponding ALU opcodes are naturally 0000, 0100, 0101 respectively, with the shamt being the 10th to 6th bit.

For SLLV, SRLV and SRAV, one extra step is needed. If the 6-bit funct code of the 32-bit instruction is “0xx1xx”, then the instruction must be one of SLLV, SRLV and SRAV. Then the “replaceShamt” output will be set to “01”. Once this output is 01, the multiplexer right before the “ShamtIn” will choose the last five bits from the A value just read to be the shamt.



3. ADD, ADDU, SUBU in the R-type instructions

These three instructions are very straightforward. They correspond to ALU opcodes 001x, 001x and 011x respectively, and the A and B values input into the ALU are just the A and B read from the register file.

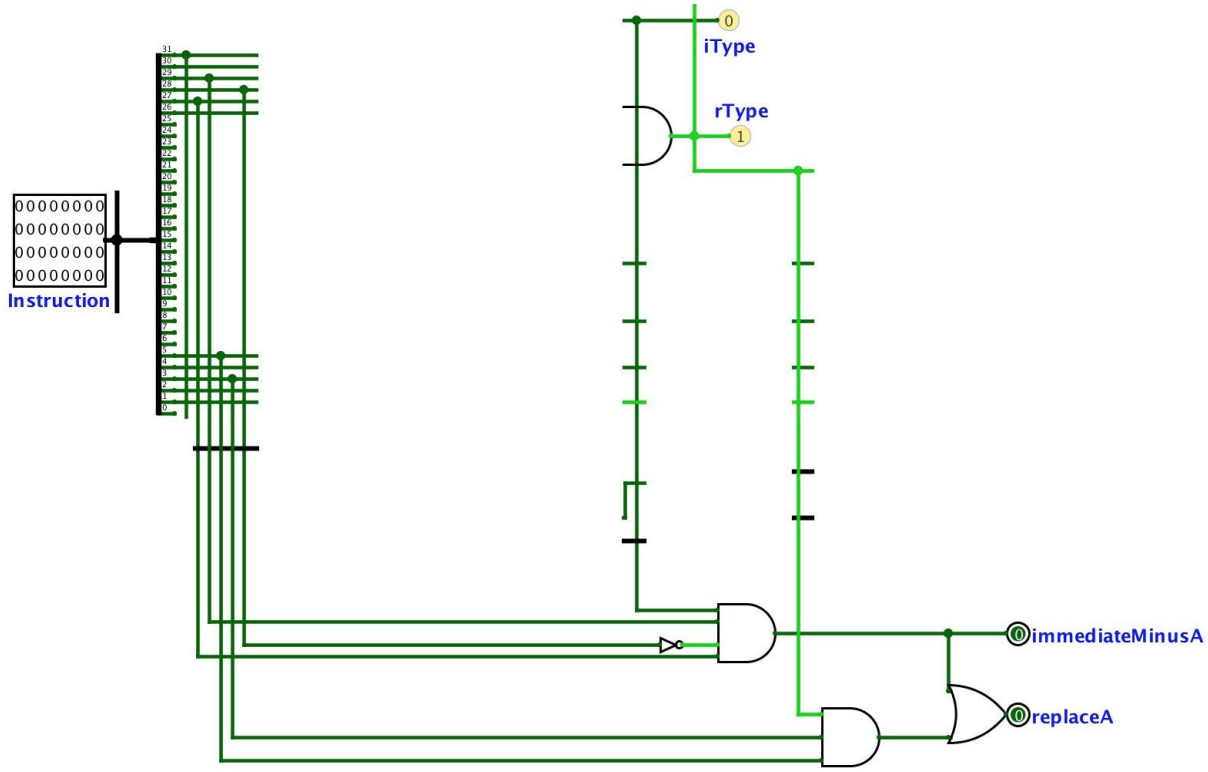
4. AND, OR, XOR, NOR

These are also very straightforward. They have a direct mapping in the ALU functions, so they correspond to 1000, 1010, 1100, 1110 of the ALU opcode respectively.

5. SLT, SLTU (R-type instructions) / SLTI, SLTIU (I-type instructions)

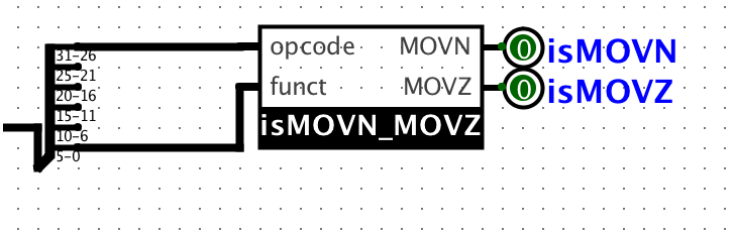
For the I-type SLTI and SLTIU, the written back output depends on the difference of the immediate minus A. This difference is passed into the ALU gtz function which determines if the difference is greater than zero. If it is, WB 1, if not WB 0. In order to find the difference of the immediate and A without passing through the ALU, two 1-bit controls are implemented: “immediateMinusA” and “replaceA”. These controls control

multiplexers which route the immediate - A into the ALU.

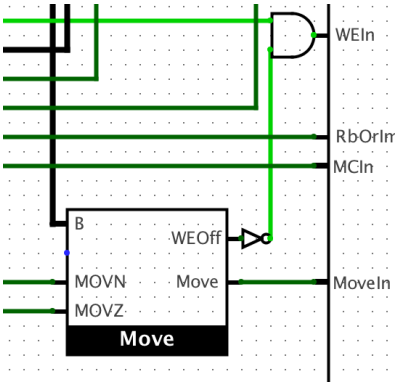


6. MOVN/MOVZ instructions

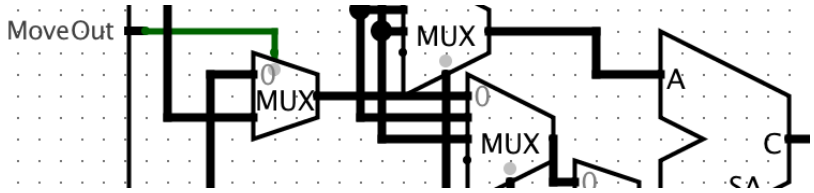
Firstly, a simple subcircuit was used to determine whether an instruction is a MOVN/MOVZ instruction.



By utilizing this simple sub-circuit, the decoding circuit will now have 2 outputs: MOVN and MOVZ. These two outputs are now in the main circuit.



The “Move” circuit takes in three inputs B, MOVN, MOVZ and outputs 2 values, namely WEOff and Move. The WEOff flag will be set to 1 iff the instruction is either MOVN or MOVZ but the condition for making the move is not met (e.g. rt=0 in MOVN, rt!=0 in MOVZ). When the WEOff is set to 1, the WE signal being passed to the execution stage will be set to 1 even if the current instruction is neither J-type nor Load/Store. The output flag named “Move” will be set to 1 if the instruction is either MOVN or MOVZ and the condition for making the move is met.



The “Move” flag will then pass through the pipeline and be used in the execution stage. If it is 1, the A value will replace B value so that both operands of the ALU will be A. According to the conversion table which will be introduced later, the 4-bit opcode will be set to 1010, so that the actual computation performed by the ALU will be A OR A. Ultimately, the value stored in rs (namely A) will be copied to rd, while the original value will not be changed.

7. ADDIU

Similar to the ADDU of the R-type, the only difference is that since it will be detected to be an I-type instruction, the “useImmmediateNotB” flag will be set to 1, so the multiplexer before the ALU will choose the immediate (instead of B value) to do the addition with A.

8. ANDI, ORI, NORI

Similar to AND, OR, NOR in R-type instructions, but the “useImmediateNotB” will be set to 1.

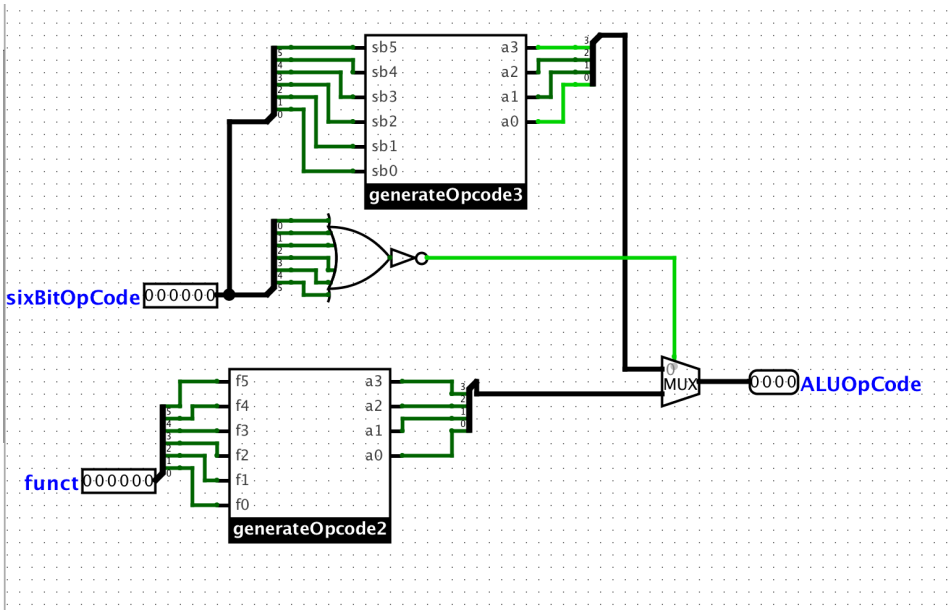
9. LUI

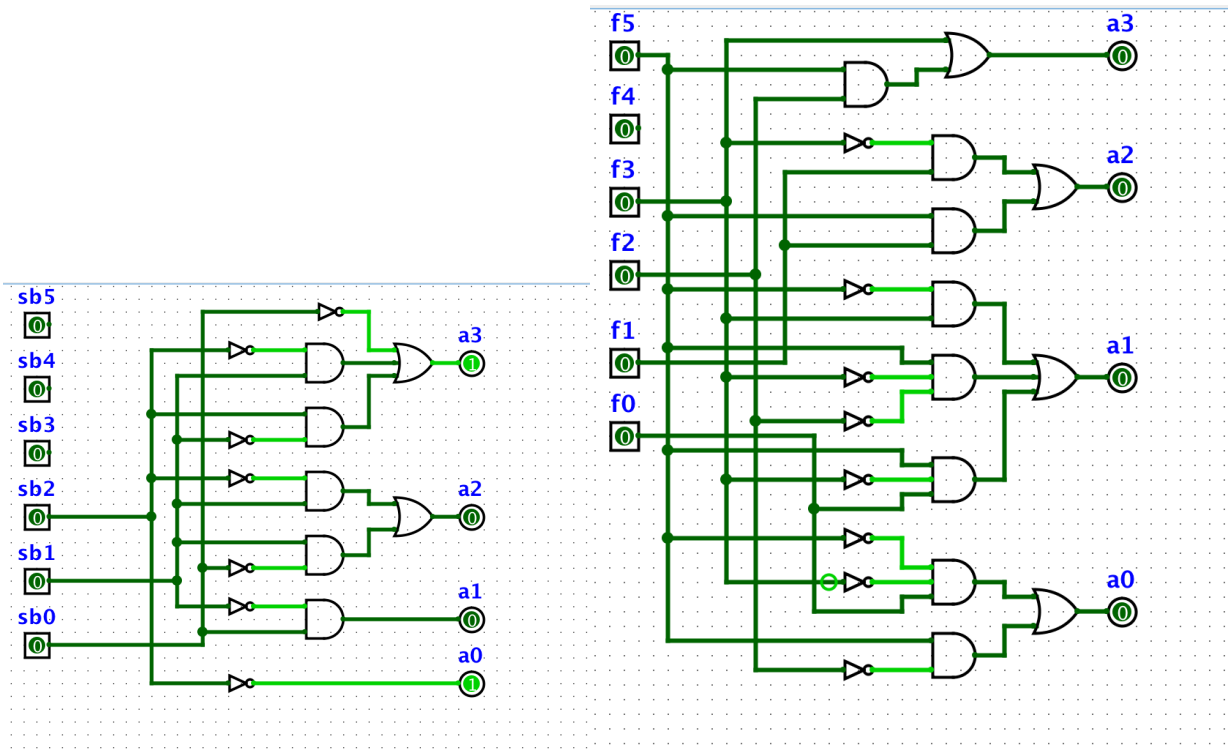
This instruction will be basically translated to a SLL instruction, with the shamt set to a constant of 16 bits. In order to set the shamt to be 16, a constant was used, and the “replaceShamt” output from the decoding sub-circuit has been set to “10” to let the multiplexer choose 16. Apparently, the corresponding ALU opcode is 0000.

After determining how every instruction in Table A and B are handled, I could now have the following “ALU opcode generating table”. It is a surjective function from the 32-bit instruction to the 4-bit ALU opcode.

Functio	6-bit op	funct (if a	ALU (4)
ADDIU	001001	N.A.	001x
SLTI	001010	N.A.	1101
SLTIU	001011	N.A.	1101
ANDI	001100	N.A.	1000
ORI	001101	N.A.	1010
XORI	001110	N.A.	1100
LUI	001111	N.A.	000x
SLL	000000	000000	000x
SRL	000000	000010	0100
SRA	000000	000011	0101
SLLV	000000	000100	000x
SRLV	000000	000110	0100
SRAV	000000	000111	0101
ADD	000000	100000	001x
ADDU	000000	100001	001x
SUBU	000000	100011	011x
AND	000000	100100	1000
OR	000000	100101	1010
XOR	000000	100110	1100
NOR	000000	100111	1110
SLT	000000	101010	1101
SLTU	000000	101011	1101
MOVN	000000	001011	1010
MOVZ	000000	001010	1010

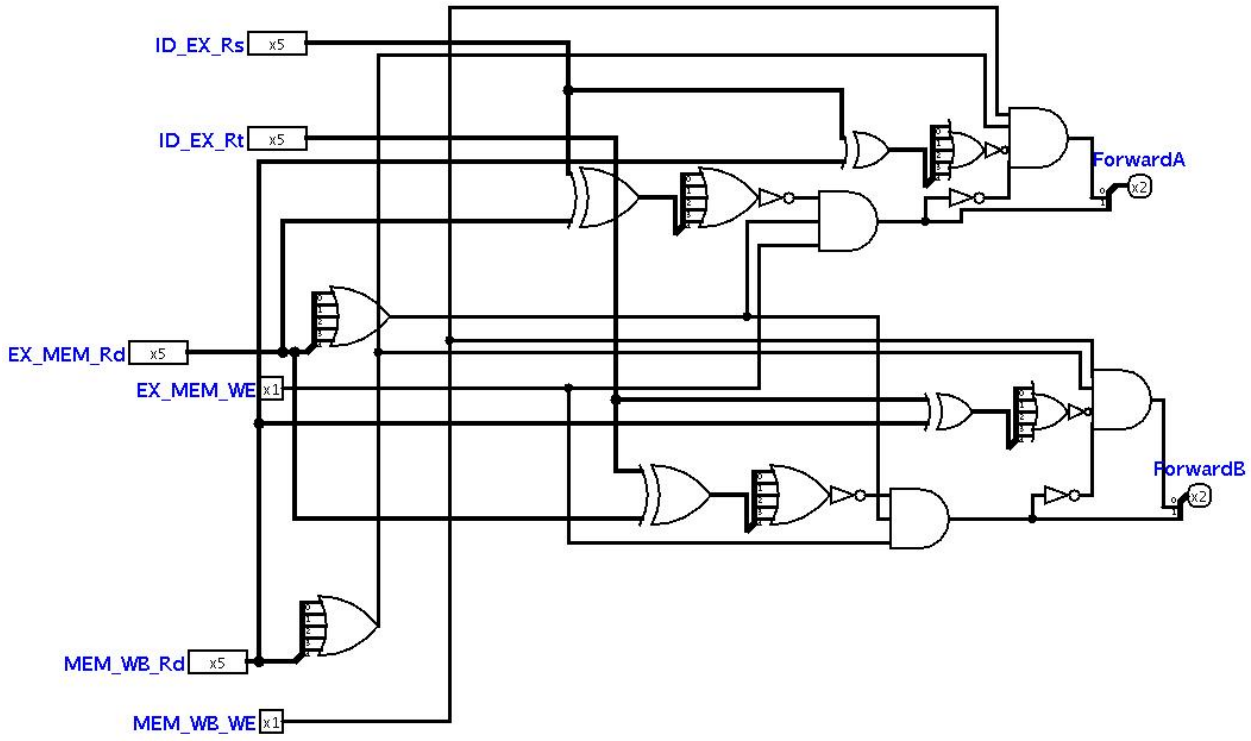
With this table, I used the “Analyze Circuit” functionality of Logism to generate the 4-bit ALU opcodes. The circuits are as follows:





Stage 3: Execution

Forwarding Components:
Two multiplexers are implemented which will input A and B from ID/EX as well as the ALU computed D value and from the memory and writeback stages.
The forwarding unit inputs Rs and Rt from ID/EX and Rd and WE from EX/MEM and MEM/WB. These inputs are the basis of the hazard detection logic which was explained in lecture. In the subcircuit basic gates are be enough to achieve this logic. The forwarding unit has two outputs which control the multiplexers described above.



A multiplexer will enable the choice of some B and the Immediate in the ALU for R-type/I-type instructions, controlled by a control from ID/EX.
Other multiplexers implementations in the execution stage are trivial.

>>What information will be passing through pipeline registers;

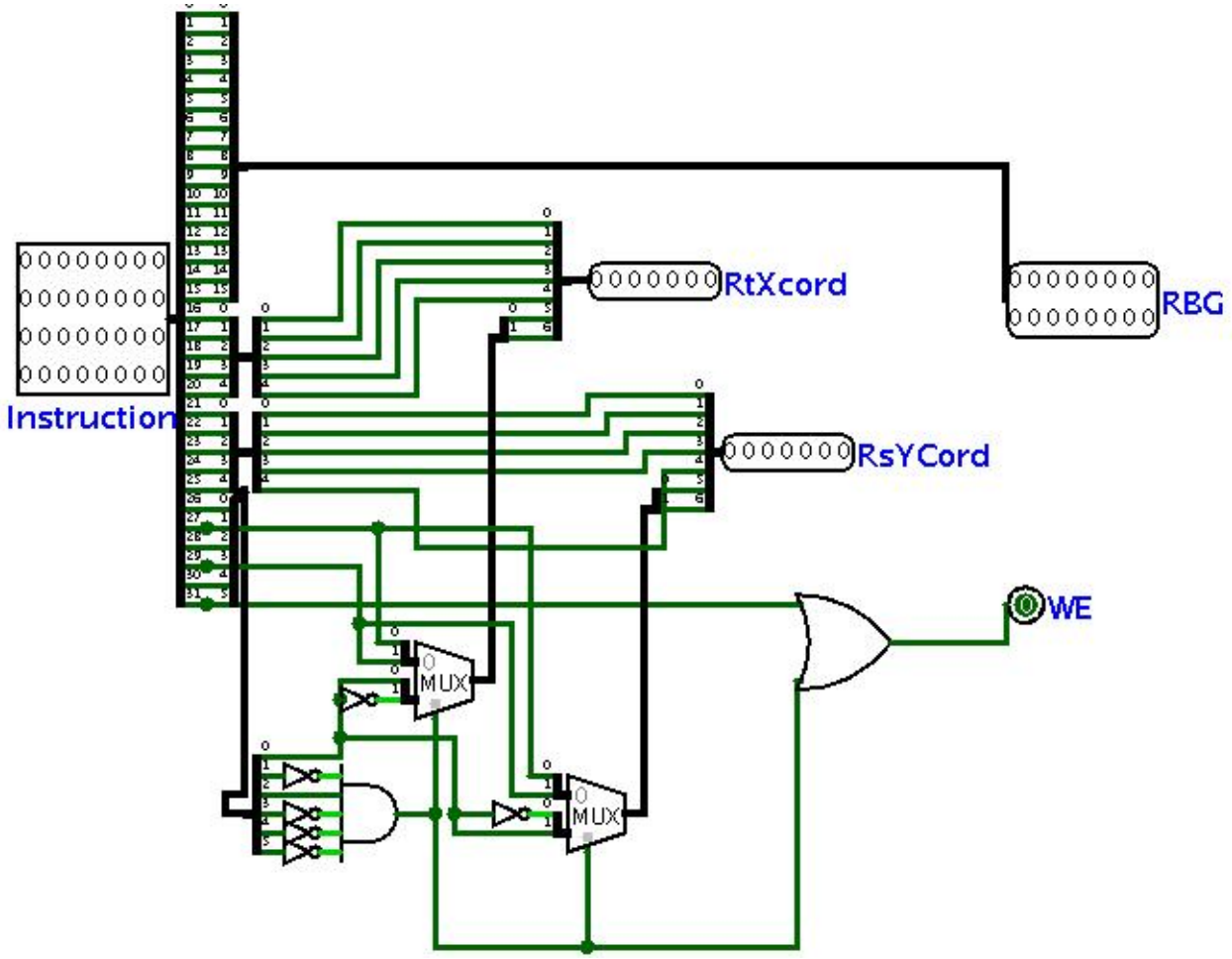
The EX/Mem pipeline registers will hold WE, MC, Rd, B, and D (ALU computation).

Stage 4: Memory

This stage is simply a passthrough stage for project 1. The data value computed from the ALU is wired back to the execution stage for forwarding in the case of a data hazard.

The LCD screen is implemented physically in the memory stage because it was the most convenient location, and the screen writes on load, store, and branch instructions. As it follows,

here is the subcircuit for the LCD control decode.



The RGB (5-5-5) output is simply the signed offset value. The first 5 bits of the x-coordinate are taken from rs, bits 21-25 and the first 5 bits of the y-coordinate are taken from rt 16-20. The two most significant bits of each coordinate output are computed from the opcode with special logic for two non-load/store instructions used: BEQ (000100)and BNE(000101). These were chosen from the unimplemented jump and branch instructions for their similar instruction layout.

>>What information will be passing through pipeline registers;

The Mem/WB pipeline registers will hold MC, WE, M (memory data) and D.

Stage 5: Writeback

After the first four stages, the execution result(DOut), register number(RdOut) and the writeback enable control (WEOut) are sent back to the register file.