CS 3410 P2 MINI-MIPS Processor Design Document

Oren Michaely (om72) and Pablo Llambias (pl438)

1. Overview

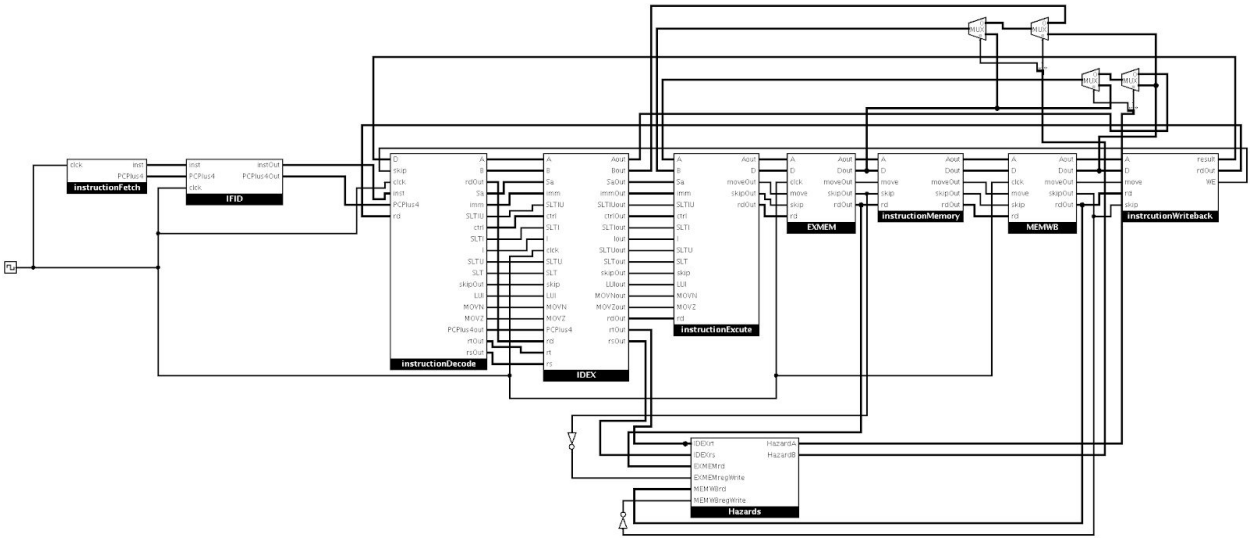
The purpose of the minMIPS is to design a processor which will perform a subset of the operations performed by the full MIPS processor. The purpose of the MIPS processor is to effectively and efficiently take in a 32-bit instruction and store the correct result to the correct register file, based on the given instruction. Additionally, it is important to make sure that this process of evaluating an instruction is done as efficiently as possible. In order to maximize the efficiency, we will implement this MIPS processor using the pipelining paradigm. When implementing with pipelining, we will need to consider data hazards, specifically hazards that require Writeback to execute forwarding and Memory to execute forwarding. Different techniques will be used to avoid complications.

2. Component Design Documentation

2.1) MIPS processor

The minMIPS processor circuit will be split into five stages for pipelining. Additionally, control logic is implemented to avoid data hazards, as well as to decide which operation will be performed and which stage is accessed at each time.

2.1.1) Implementation Details



The MIPS Processor has 5 stages. The first stage is Instruction Fetch; in addition to fetching instructions, this will increment the PC counter by 4 each clock cycle in order to read the next instruction. The second stage is the Instruction Decode Stage, where the 32 bit instruction is “broken up”, and corresponding control signals are output. The third is the Execute stage, where the control signals determine what is inputted into the ALU and what operations go into the comparators and then the correct computed output is chosen as the result. The fourth stage is the Memory stage, which is in charge of storing the values in memory. In this project no values are stored to memory so this stage in the pipeline does nothing. The fifth and last stage is the Write-Back stage which writes back the output from the ALU to the correct register file. Additionally, between each stage are pipeline registers which take in as inputs the outputs of the stage before and output the inputs of the stage to come next. There are thus subcircuits for the IF/ID, ID/EX, EX/MEM, MEM/WB pipeline registers. Each of these subcircuits also have a clock input that comes from the clock in this top-level circuit telling the pipeline register subcircuits when to write to the registers. The last subcircuit in this circuit is the one that handles hazards. This circuit performs the logic that produces outputs that indicate if there is a hazard with A, B, both, or neither. Finally, muxes are used above with the hazard outputs as control bits in order to decide what to forward in the case that forwarding is necessary. There are 2 muxes to decide what the input to A will be and two muxes to decide what the input to B will be (these are inputs into the execute stage). Truth table:

Control (bit 0 from hazard A)	Output
0	A
1	D (from MEM/WB pipeline registers)

Then the output is passed as the 0 input for the next mux. The 1 input for the next mux is D from EX/MEM pipeline registers. Truth table:

Control(bit 1 from hazard A)	Output
0	Output of prev. Mux
1	D(from EX/MEM pipeline registers)

This type of implementation is used as it will mean that if there are neither mem hazards or ex hazards just A will be input, if there are both the mem hazards will be “prioritized” and thus the result from the EX/MEM pipeline register will be input, and if there are only mem or only ex hazards, then the proper input will be chosen with the logic above.

The exact same logic is used for the other two muxes, simply with B replacing A and the control bits coming from the 2-bit hazard B output instead of the hazard A output. The result is then placed into the execute stage as the input for B.

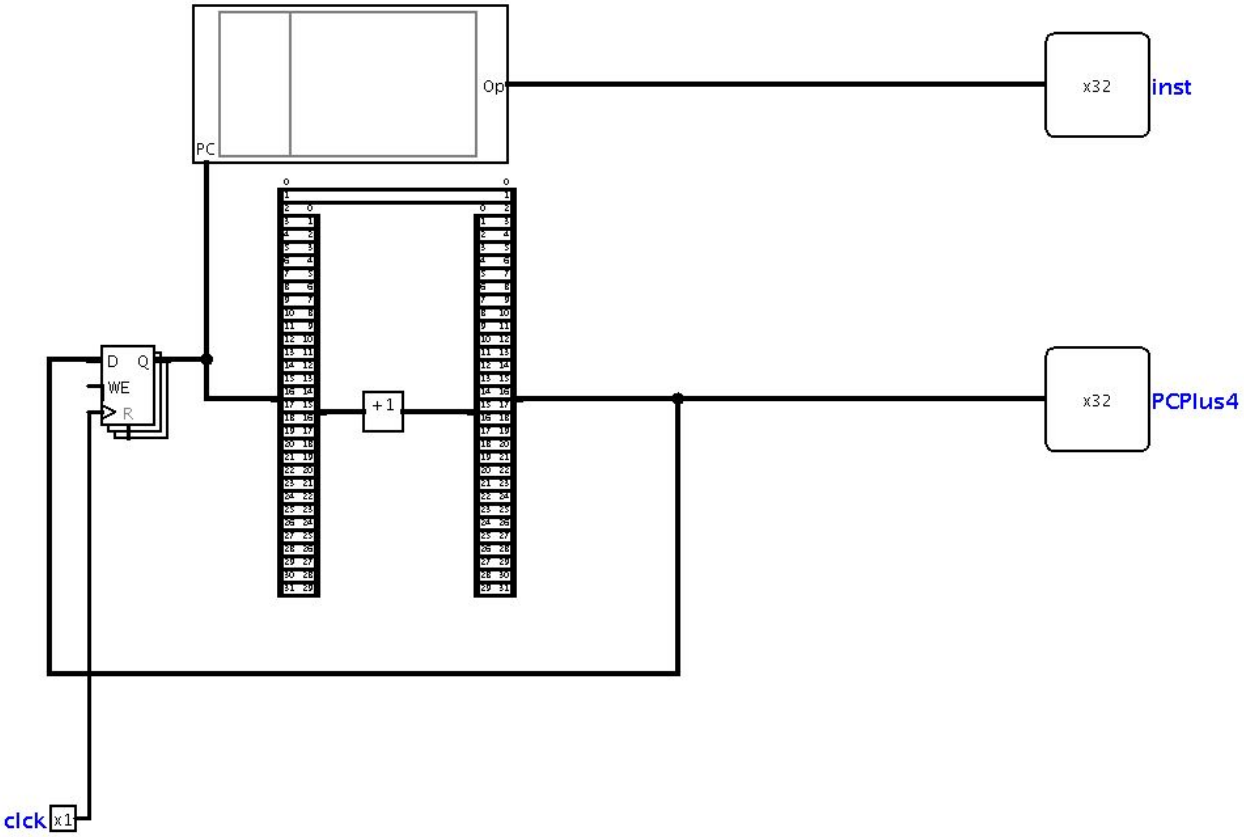
2.1.2) Evaluation

Pipelining improves the throughput by an amount proportional to the amount of stages in the pipeline. After the initial instruction is written in 5 clock cycles, subsequent instructions are written in 1 clock cycle.

2.2) Instruction Fetch

The PC calculates the new target address that the next instruction is stored in in the ROM(Read-Only Memory). That instruction will be read and stored in a register in the IF/ID register file. The PC will then be incremented by 4 to prepare for the next instruction.

2.2.1) Implementation Details



In the Instruction fetch stage, we fetch a new instruction every cycle. Every cycle the PC points to the index of the instruction in the ROM memory. At the end of every clock cycle the PC is incremented by 4. We read the instruction bits from the ROM on the rising edge of the clock cycle (we will need it later for decoding) and the PC+4 (will be passed along, passed along version will be used in next project) to the pipeline register (IF/ID).In order to increment the PC by 4, we will take the 30 most significant bits and pass them through the incrementer. Then, we will simply append the other two bits and have therefore incremented it by 4.

2.2.2) Evaluation

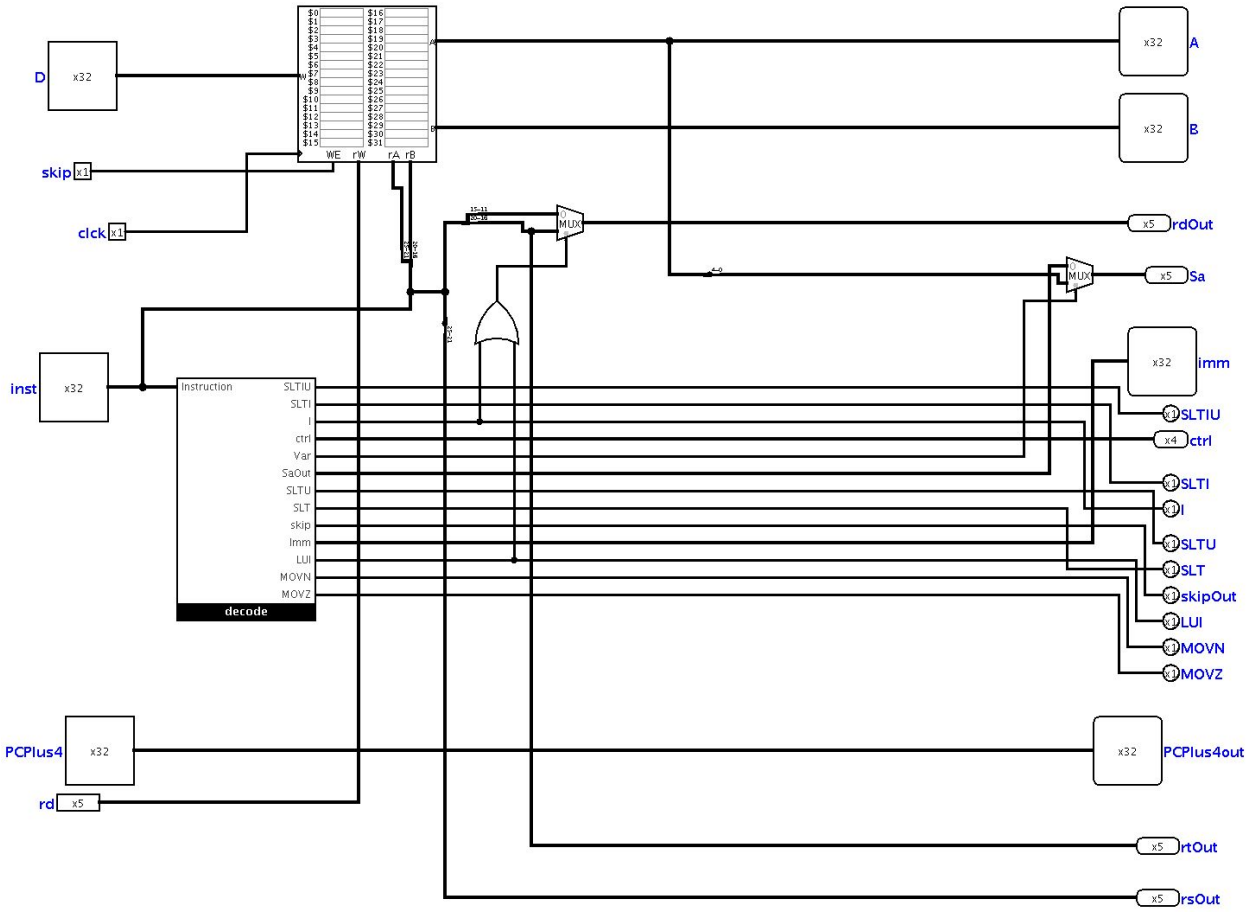
No gates used, efficient, will run on rising edge of clock inputted from clock of the top level circuit.

2.3) Instruction Decode

The instruction decode stage is essential in the overall performance and correctness of the processor. It will take in an instruction, which is read form the IF/ID pipeline register, and will decode it, generating control signals that will determine which operation is performed in the execute state. It will pass these control signals to the ID/EX register. Additionally, it will pass controls to the Reg file to determine what will be written, when,

what will be read, when, and what register something will be written to or read from.. Additionally, depending on the type of instruction, it will take the 16 bit immediate portion of the instruction from the IF/ID RF and zero-extend it into a 32-bit value that will be passed to a register in the ID/EX register file.

2.3.1) Implementation Details



The input D determines what will be written to the register file (this comes from the writeback stage). The skip input indicates if writing will be enabled or not in this clock cycle. The clock feeds in an input, and writing or reading is done on the falling edge for this register file, due to hazard considerations (will be explained later). Specific bits from the instruction input are fed into the reg file to tell the reg file which registers to read from for A and B outputs. An rd output represents where the result of the operation will eventually write to: a mux is used to determine, based on the operation to be done from control logic, which bits to use from the instruction to represent the register to be write to, since different operations sometimes write to different registers. Another mux is used to determine what should go into the shift amount output, picking between a variable

amount (from last 5-bits of A) or an input amount value from the instruction bits. The PC counter is passed through for the next project. There is also an rd input, which tells the register file where to write to, fed in from the writeback stage. Rt and rs are also outputs as they will be needed later for hazard logic. Then there is a ctrl output, which will be the opcode that is fed into the ALU, and the rest of the outputs are indicators on whether a certain operation is being performed and come from our decode subcircuit.

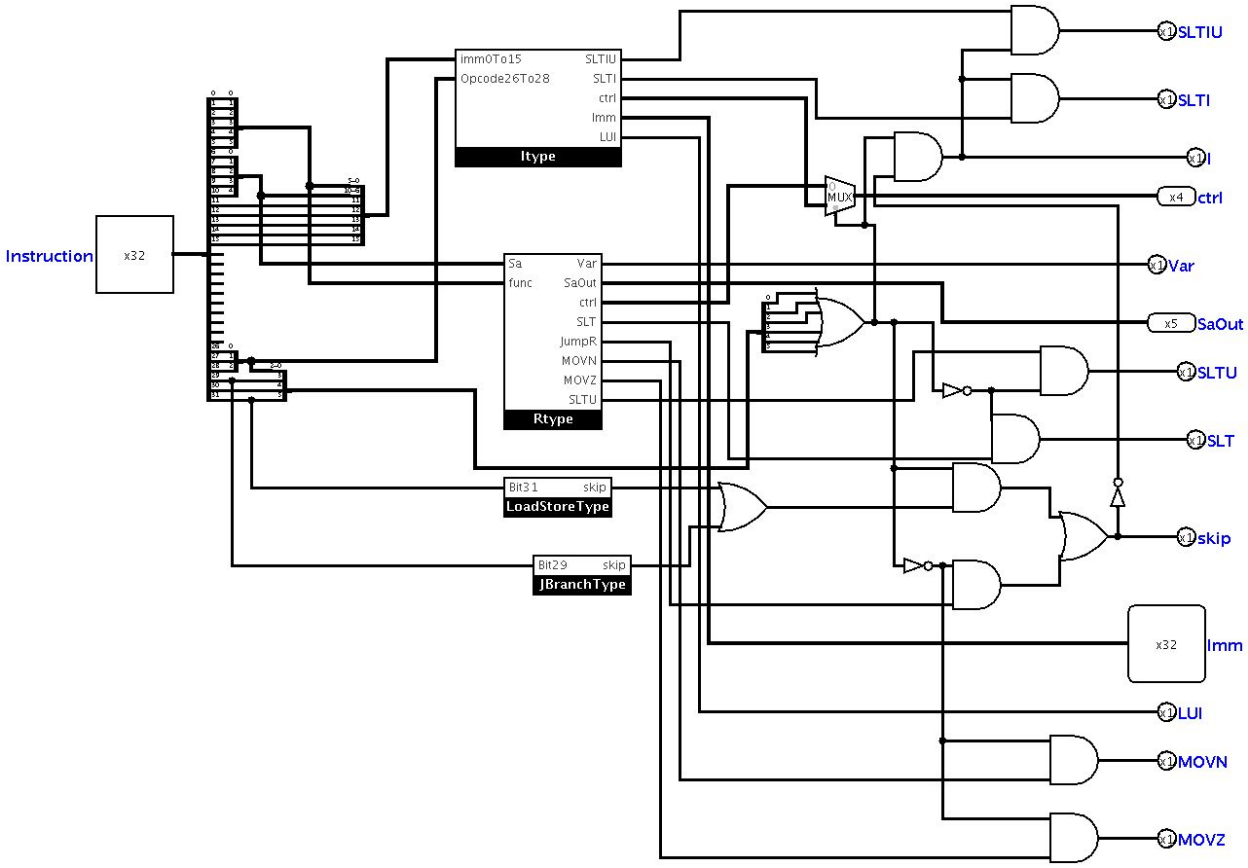
2.3.2) Evaluation

The two muxes used are necessary to pick between certain inputs, overall an efficient circuit.

2.4) Decode

In this subcircuit, most of the decoding logic is done. It uses 4 subcircuits: Itype, Rtype, LoadStore Type, and JBranch type. Overall, it uses outputs from these subcircuits and using gates determines what the output of this circuit will be.

2.4.1) Implementation Details



The 32-bit instruction is input into this subcircuit. The corresponding bits of the instruction are fed into the 4 subcircuits named above. The subcircuits within differentiate between instructions of the same type, i.e. I-type, R-type, etc. In order to differentiate between types, we first differentiate between I and R type operations by

ORing the 6 most significant bits. If any of those bits are 1, then we know that it cannot be R-type. Then, if it is also not a skip (not skip and the result of the OR gate go to an AND gate), then the I-type output will be a 1. The skip output is if it is not I type and not R-type, i.e. either J-type or a Load-type. Thus this produces the skip output (term skip is used to denote that for this project, if it is a “skip”, we must assure that the registers do not change). The ctrl output must choose between the ctrl output of the R-type circuit and the ctrl output of the I-type circuit, as neither of the other two “types” produce a ctrl output. The control bit for the mux that chooses which one is the result of the OR gate mentioned above. The output is the fed into the ctrl output of the circuit. The immediate output simply comes from the I-type subcircuit, and the rest of the outputs are 1-bit indicators produced in the 4 subcircuits.

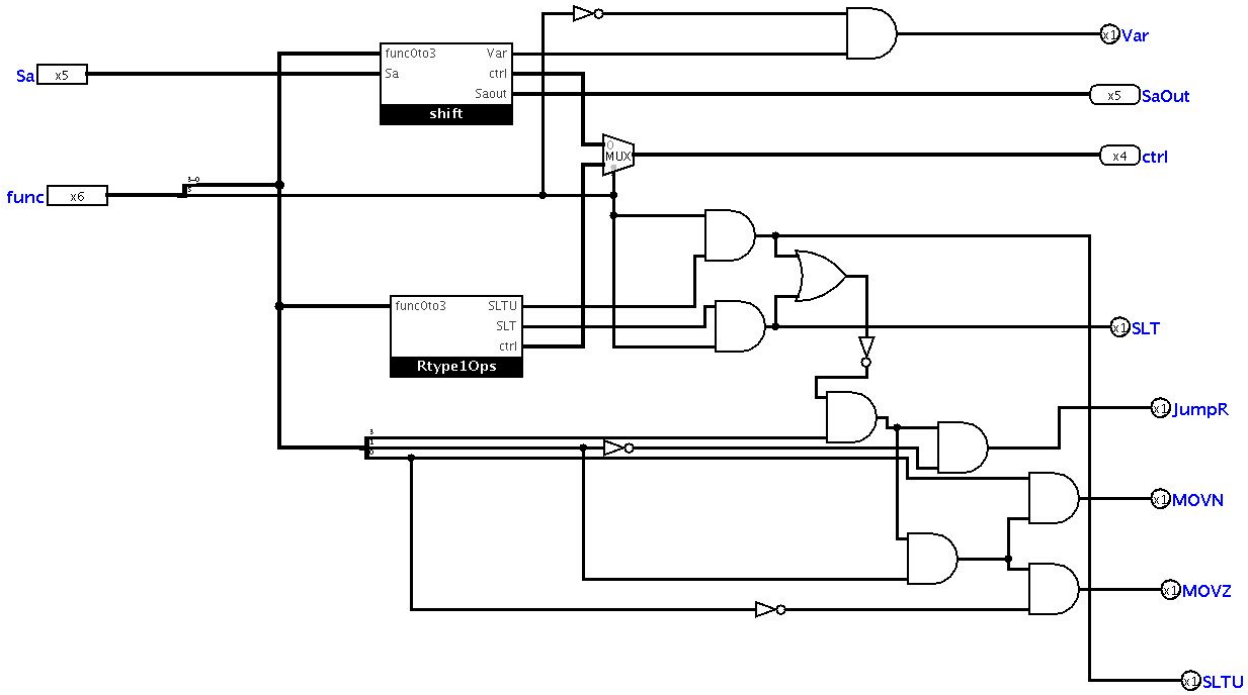
2.4.2) Evaluation

Many gates are used, but it is necessary in order to perform the logic correctly.

2.5) R-type

This subcircuit is used in the decode stage. It differentiates between R-type shifts and R-type operations whose function code begin with a 1. It produces outputs to help with the logic of the decode circuit.

2.5.1) Implementation Details



The input to this circuit is Sa, the shift amount taken from the corresponding instr. bits and the function code, i.e. the 6 least significant bits from the original instruction. Below is the R-type comp. instructions.

R-Type Computational Instructions

						Name	Mnemonic
000000	000000	src	dest	shamt	000000	Shift Left Logical	SLL rd, rt, shamt
000000	000000	src	dest	shamt	000010	Shift Right Logical	SRL rd, rt, shamt
000000	000000	src	dest	shamt	000011	Shift Right Arithmetic	SRA rd, rt, shamt
000000	src1	src2	dest	000000	100000	Add (with overflow)	ADD rd, rs, rt
000000	src1	src2	dest	000000	100001	Add Unsig. (no overflow)	ADDU rd, rs, rt
000000	src1	src2	dest	000000	100010	Subtract	SUB rd, rs, rt
000000	src1	src2	dest	000000	100011	Subtract Unsigned	SUBU rd, rs, rt
000000	src1	src2	dest	000000	100100	And	AND rd, rs, rt
000000	src1	src2	dest	000000	100101	Or	OR rd, rs, rt
000000	src1	src2	dest	000000	100110	Xor	XOR rd, rs, rt
000000	src1	src2	dest	000000	100111	Nor	NOR rd, rs, rt
000000	src1	src2	dest	000000	101010	Set Less Than	SLT rd, rs, rt
000000	src1	src2	dest	000000	101011	Set Less Than Unsig.	SLTU rd, rs, rt

If the func bits most significant bit is a 0 then we know we have some kind of shift operation (SLL,SRL,SRA,SLLV,SRLV,SRAV). If the func bits most significant bit is 1 we have an arithmetic operation (ADDU, SUBU, AND, OR, XOR, NOR, SLT, SLTU). We produce a ctrl output that will be fed into the ALU inside the two subcircuits of this circuit. For operations that do not require the ALU, we produce 1-bit indicators. In this circuit, many gates are used in order to ensure that illegal combinations of outputs are not produced. For example, SLT and SLTU cannot both be 1 for a given operation, MOVN and MOVZ cannot both be 1, Var cannot be 1 if the most significant bit of the func code is 0, etc. The bit that decides the output of the mux that takes in the ctrl output of the two sub circuits is the most significant bit of the func code, as described above.

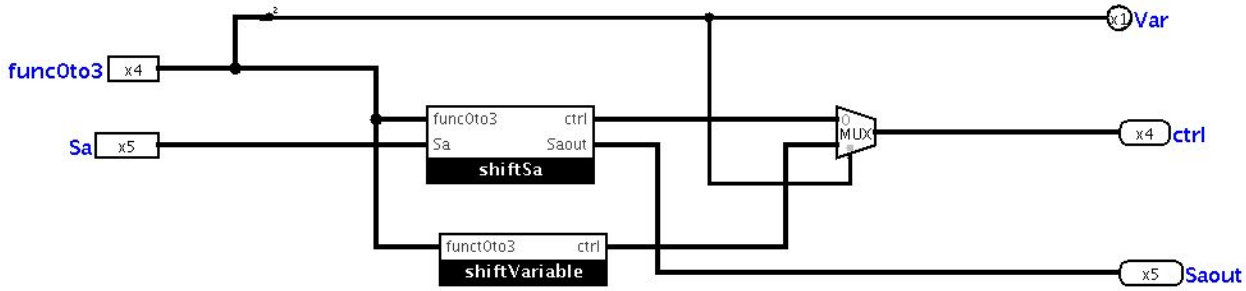
2.5.2) Evaluation

Gates must be used in order to avoid illegal combinations. A mux must be used to pick between two ctrl outputs.

2.6) Shift

This circuit is used to choose between the output of the two shift subcircuits.

2.6.1) Implementation Details



A mux is used to pick between the ctrl output from shiftSa and the control output from shiftVariable. The control bit for the mux is bit 2 from the funct code. Additionally, if bit 2 is 1, the var output also gets a 1, indicating we will be shifting by a variable amount.

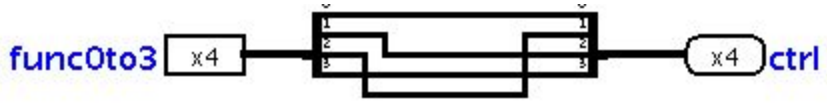
2.6.2) Evaluation

Just one mux.

2.7) shift Sa

The shift Sa manipulates bit 0 - bit 3 of the function opcode in order to turn it into the proper ALU opcode that will perform the correct shift operation.

2.7.1) Implementation Details



This specific manipulation of bits will always result in the correct ALU opcode being passed into the ALU.

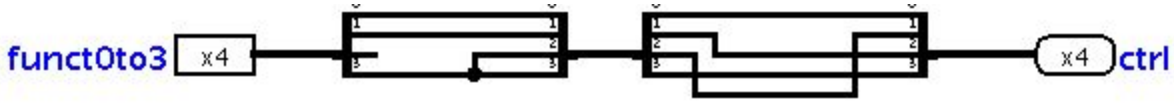
2.7.2) Evaluation

No gates.

2.8) shift Variable

The shift Variable manipulates bit 0 - bit 3 of the function opcode in order to turn it into the proper ALU opcode that will perform the correct shift operation.

2.8.1) Implementation Details



In order to manipulate the bits correctly, copy bit 3 into bit 2. Then perform the same manipulation as the shiftSa circuit above. This will result in the correct opcode being fed into the ALU if the operation is a shift variable.

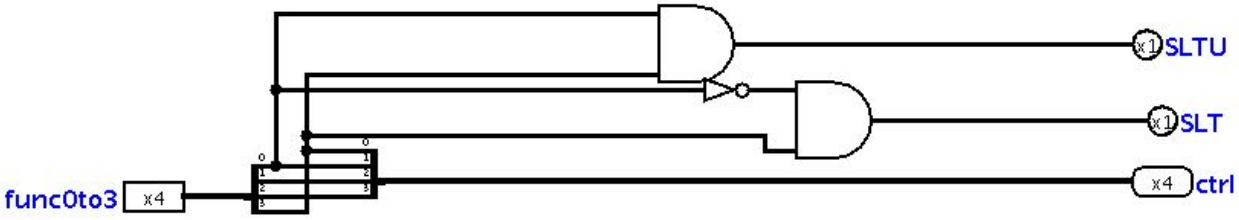
2.8.2) Evaluation

No gates.

2.9) Rtype 1ops

The goal of this sub circuit is to produce the correct control opcode for the ALU for Rtype operations whose function code bit 5 = 1, or to identify an SLTU or SLT operation.

2.9.1) Implementation Details



If bit 0 and bit 3 are 1, then we know it is an SLTU operation. If bit 0 = 0 and bit 3 = 1, then SLT = 1. Else, to produce the ctrl opcode, the bits are all shifted left and then bit 3 replaces bit 0.

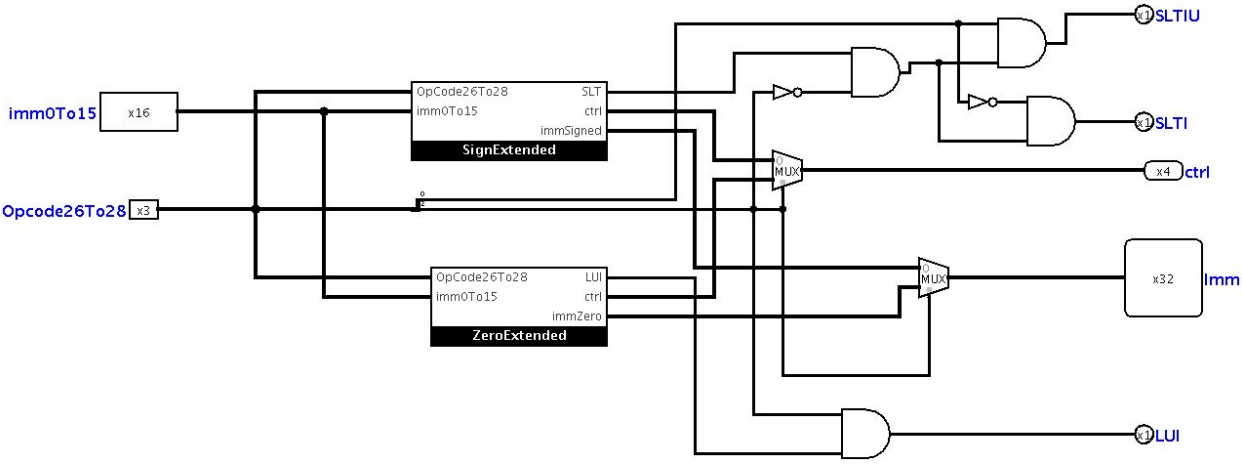
2.9.2) Evaluation

Efficient.

2.10) I-type operations

The purpose of the I-type operations sub circuit is to differentiate between operations who are in the I-type category. The outputs of this circuit are used in the decode circuit.

2.10.1) Implementation Details



I-Type Computational Instructions				Name	Mnemonic
001001	src	dest	signed immediate	Add Imm. Unsigned	ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate	Set Less Than Imm.	SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate	Set Less Than Imm. Unsig.	SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate	And Immediate	ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate	Or Immediate	ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate	Xor Immediate	XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate	Load Upper Imm.	LUI rt, zero-ext-imm.

The I-type operations can be broken down into two cases: one in which bit 28 = 0, the other in which bit 28 = 1. If bit 28 = 0, we will consider I-type operations which have a signed immediate component (as opposed to a zero-ext immediate component). Thus, we have the two subcircuits within this circuit. Their outputs for both the ctrl and the immediate output are put into muxes, with the control bit being bit 28. Additional gates are used to assure that SLTI, SLTIU, and LUI outputs produce a 1 when they are supposed to.

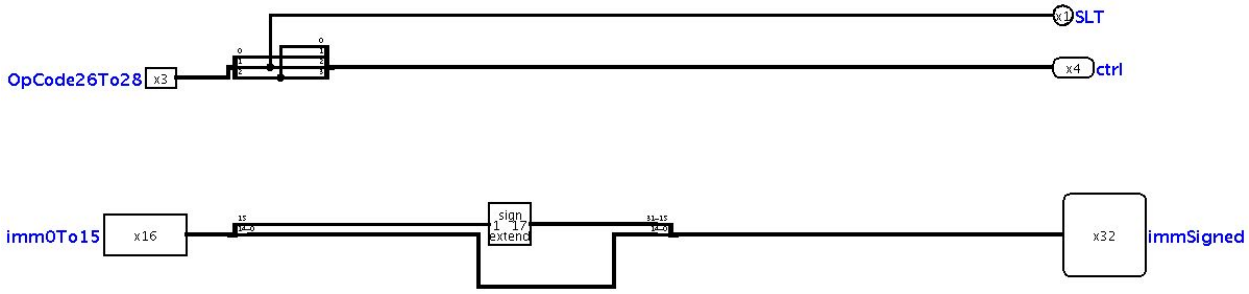
2.10.2) Evaluation

This circuit only uses two muxes, so efficient.

2.11) SignExtended

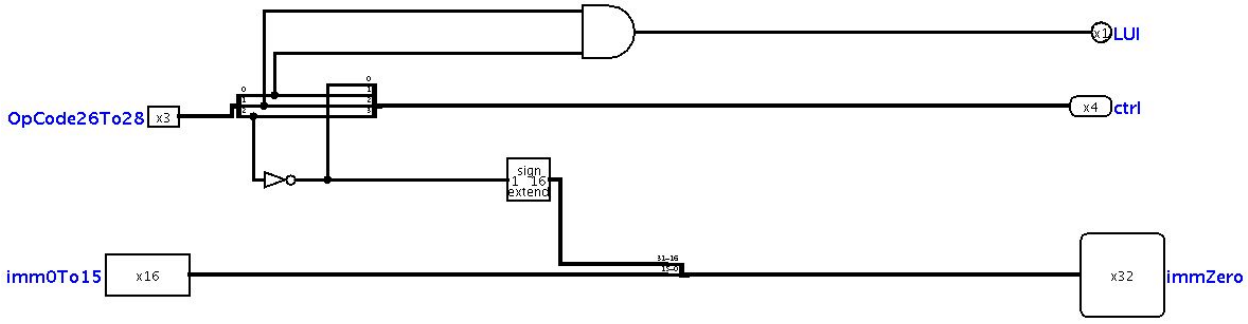
The goal of this subcircuit is to differentiate between the operations which require the sign extension of the immediate input.

2.11.1) Implementation Details



In this subcircuit, bit 0-14 will be concatenated with a bit extension of the signed bit of the immed input, namely bit 15. Thus bit 15 will also be fed into the the resulting bit 31 place, and this output will be passed to the ID/EX register file. Additionally, if bit 26 = 0, then we know it will be an ADDIU operation, so we will output bit 27, bit 26, bit 25 and a 0 into a 4-digit opcode that will be fed into the ID/EX pipeline register file. The 0 will come from bit 28 and will go into the least significant spot of the opcode. If instead bit 26 = 1, then we know it will either be a set less than immediate or set less than immediate unsigned.

2.11.2) Evaluation
Only a sign extension is used.

2.12) Zero Extended
The goal of this subcircuit is to differentiate between the operations which require the zero extension of the immediate input.
2.12.1) Implementation Details


In this subcircuit, bit 0-15 will be concatenated with a 16 bit zero-extension. The zero to be extended will come from the NOT of bit 28. The zero's will go into the 16 most significant bits, and the immed input will go into the 15 least significant bits. This output will be passed to the ID/EX register file. Additionally, if bit 25 and bit 26 both 1, then we know that the operation to be done is the Load Upper Immediate. We will thus pass an opcode into the output of this subcircuit that contains bit 25 and bit 26, signifying that within the zero-ext immed. Subcircuit, the operation to be done is an LUI.
2.12.2) Evaluation
Only one and gate and a zero extender.

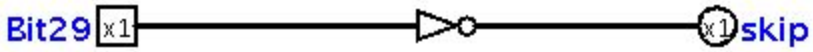
2.13) Load Store Type
This circuit simply has bit 31 as an input fed directly into a skip output. It means that we have a Table B operation and thus for this project nothing should be done.
2.13.1) Implementation Details


2.13.2) Evaluation
Wire.

2.14) JBranch Type

The purpose of this operation is to provide a NOP.

2.14.1) Implementation Details



If bit 29 = 0, then we have a J/Branch instruction, so must ensure that registers are not changed.

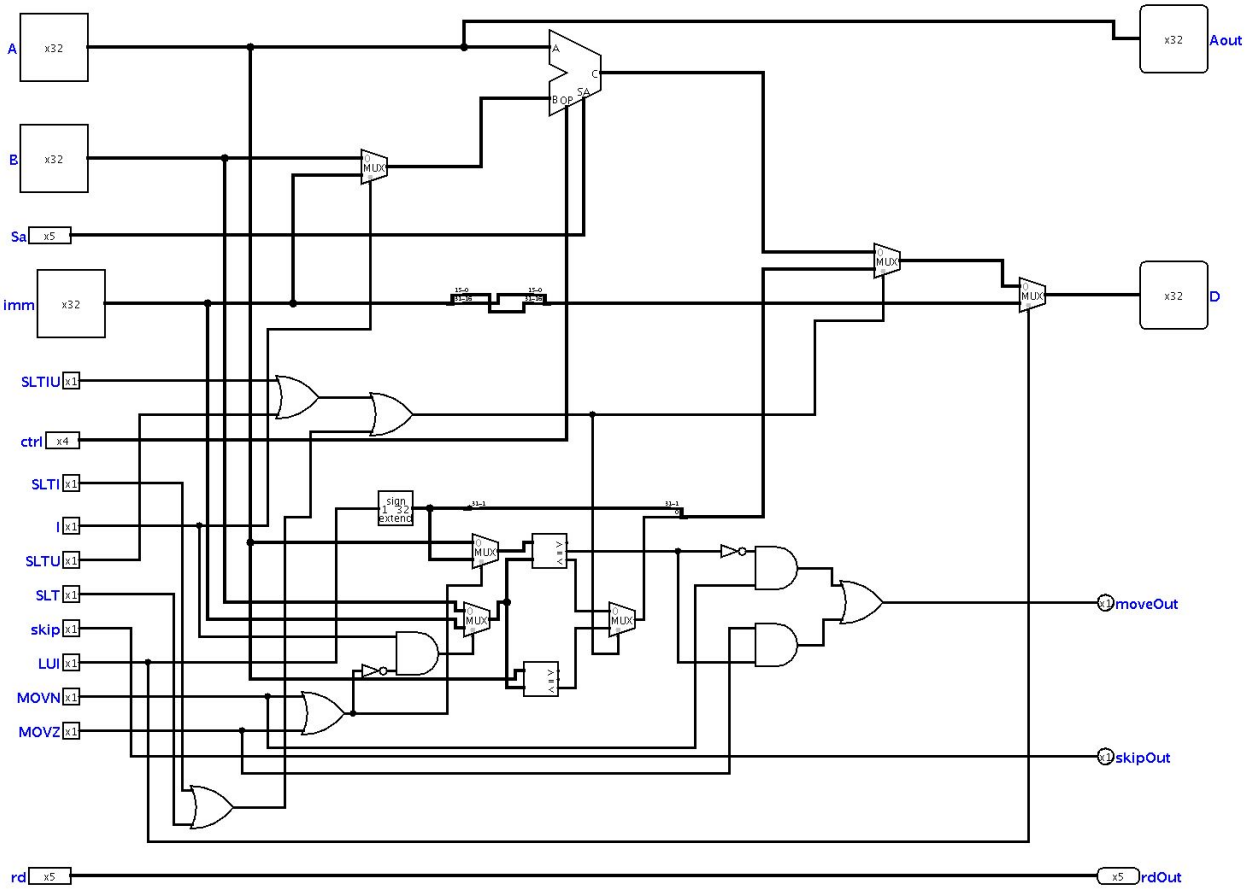
2.14.2) Evaluation

Not gate.

2.15) Instruction Execute

The purpose of the instruction execute stage is to compute the result of the operation, and, based on the control logic signals that are inputted, select the correct result to be executed.

2.15.1) Implementation Details



The inputs in this circuit come from the outputs of the ID/EX registers. Input A is always fed into the ALU, as any operation that requires the ALU will always use at least one register. The B input and the immediate input are both put into a mux and then the one

is chosen to be the second input for the ALU. The control bit for this mux is the I input, thus if it is 1, the immediate input is chosen, if 0, the B input is chosen. Additionally, the ctrl input is passed as the ALU opcode, telling the ALU which operation to perform, and the Sa input is passed into the ALU as the shift amount input. On the bottom half of the diagram, gates are used to correctly determine the output of operations that do not require the ALU. One mux is used to choose between an input of all 0's (used when we need to compare a register value to 0) and the A input. The control bit for this mux comes from the result of an OR gate with MOVN and MOVZ indicators as its input. This is because if we have a move operation, we need to pick the all 0's input to be fed into one of the inputs of the top comparator. The other input of that comparator is chosen from a mux that has B and immediate as its input. The control bit for this mux is the result of an AND gate between the I type input and the NOT of the MOVN/MOVZ OR gate. The other comparator takes in as inputs A and the result of the previous mux just described. The results of the two comparators are also put into a mux, whose control bit determines the correct output to be chosen. The resulting output of this mux is then extended by 31 bits, with bit 1-bit 31 of this potential output being extended with 0's. This output, and the output of the ALU, are put into a mux whose control bit chooses the ALU output if it is any operation that requires the ALU (i.e. most of the I type and R type operations) or chooses the comparator output if it is any operation that does not require the ALU and is not a Load Upper Immediate. Finally, this chosen output is passed in as one input of a mux, with the other input being the result of the LUI operation, which just involves swapping the 16 most significant bits of the imm input with the 16 least significant bits of the imm input. The control bit for this mux naturally is the LUI indicator input, which if 1 chooses the LUI input, or if 0 chooses the result of the previous mux. Finally, this output is passed as the output to the D output, which represents the result of the instruction execute stage. The A input is also passed directly as an output in this circuit, as it will be necessary in the case of a move operation in the writeback stage. For that same reason, a move indicator is also output. The rd register is passed directly through as an output as it will be needed to know which register to write the result to in the writeback stage. Finally, a skip indicator is also output, which will be used in a later stage to completely disregard the result of the instruction execute stage in the case that we do not want anything written to the registers. All of the outputs of this stage will be passed to the EX/MEM pipeline registers.

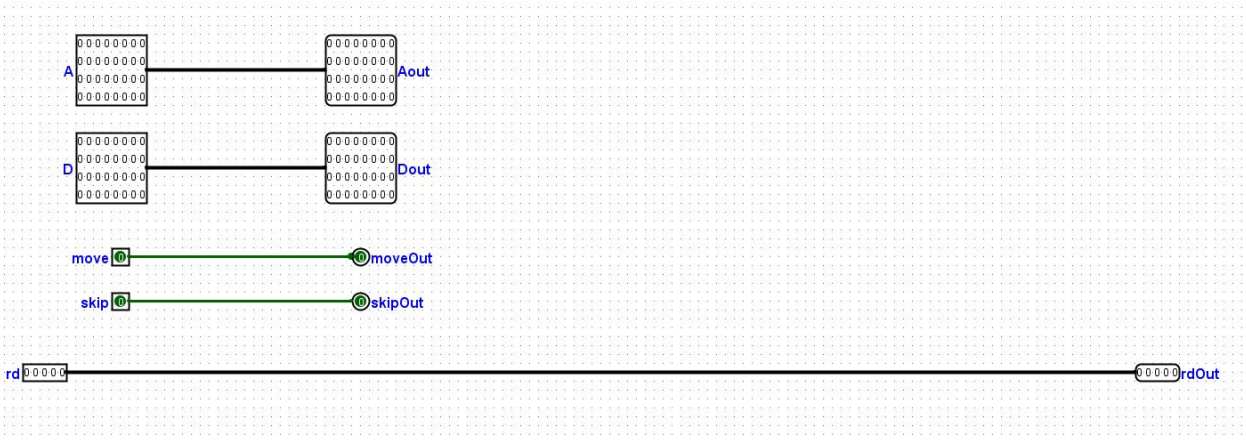
2.15.2) Evaluation

We tried to use the minimum number of muxes we could, but in certain cases we had to to correctly choose the right output.

2.16) Memory

In this project we are not implementing the functionality of writing back to memory. In this project the memory stage just passes the value on to the next stage.

2.16.1) Implementation Details

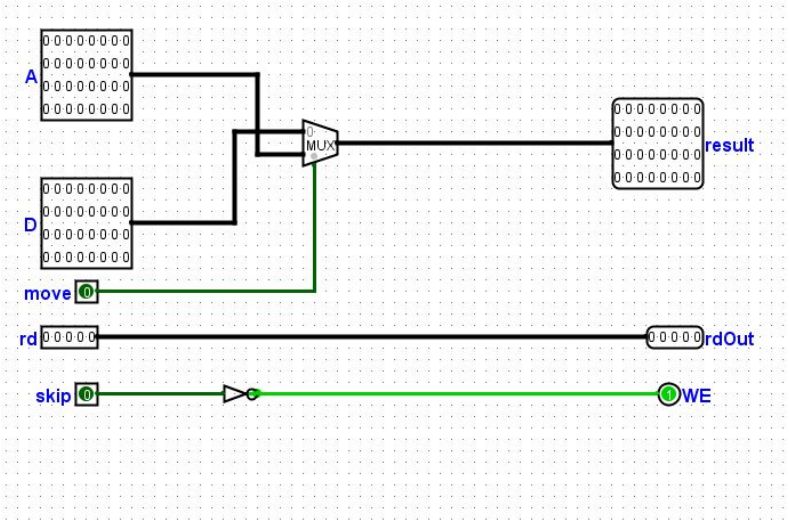


In this stage we have two 32 bit inputs A and D, two 1 bit inputs skip and move and a five bit input rd. All of the inputs are directly passed to corresponding outputs and nothing changed in this stage.

2.17)Write Back

In this stage, we set and write the final result of the instruction to the correct register file, and if nothing needs to be written or the instruction was one that is not implemented in this project then we set the write enabled on the register file to 0 so that no value is written or changed.

2.17.1) Implementation Details



As we see above, the write back stage has two 32 bit inputs A and D, two one bit inputs move and skip, and a 5 bit input rd. rd is the location in the register file that the 32 bit output result needs to be written to. We have a mux deciding between A and D, if the instruction is MOVN or MOVZ then move will be one and the value in A will be written to result, in all other cases the value of D will be written to result. The skip input is 1 if the instruction is of a type that we did not implement in this project, so we use the inverse as control for the write enabled in the register.

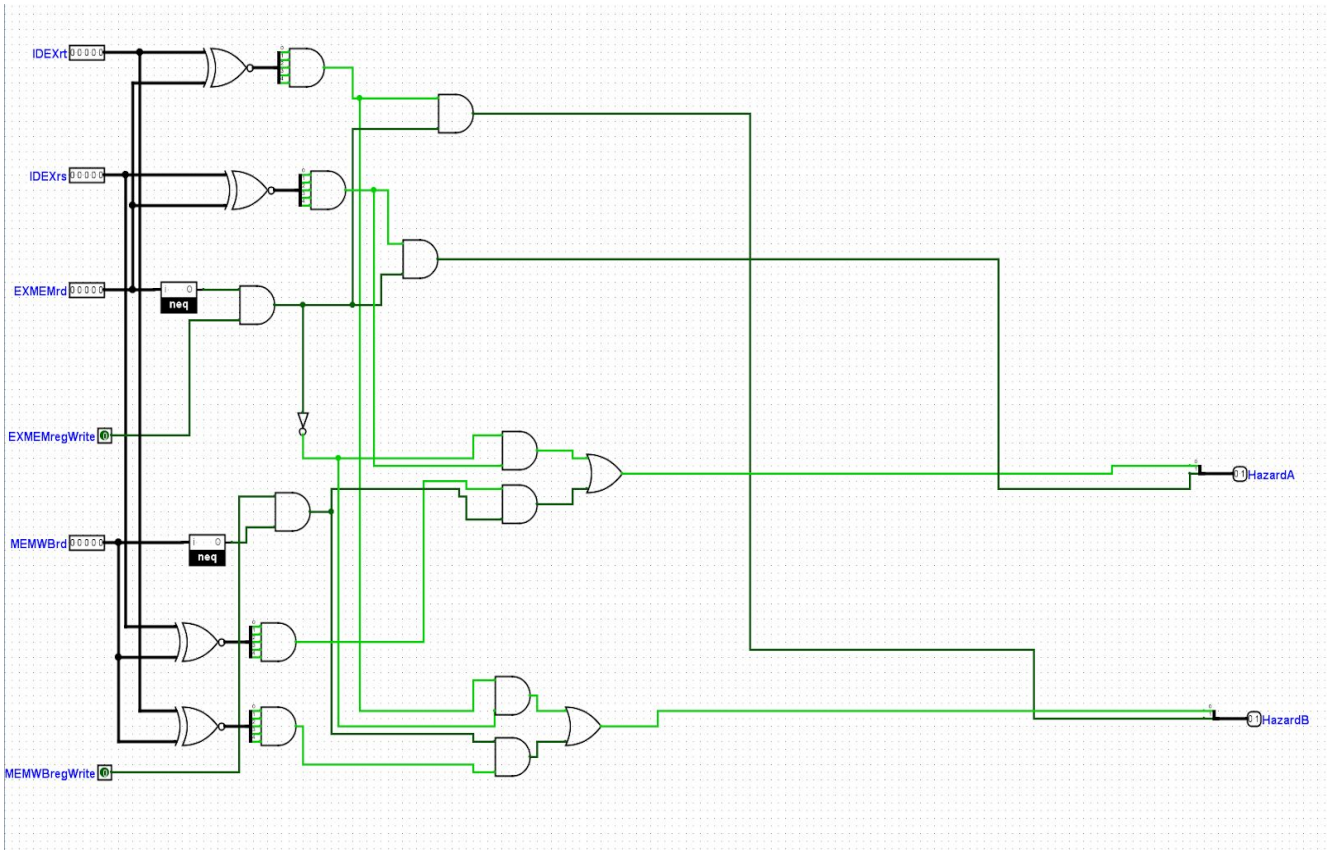
2.17.2) Evaluation

1 mux and no gates. As far as we can tell, this is the optimal way to implement this circuit.

2.18) Hazards

One of the ways to avoid hazards is forwarding. Forwarding is the only hazard control that we are implementing in this project. The Hazard sub-circuit checks if the the current instruction in the execute stage needs the value of that is still in the pipeline and has not been written back to the register yet. Depending on where that value is we input it into the execute stage either from the MEM stage or the WB stage.

2.18.1) Implementation Details



In the Hazards circuit we implement the following conditional statements that check if there is a hazard or not.

Ex Hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs)) ForwardA = 10

If this statement is true, it means that the current instruction that is in the Ex stage needs its input A to be the value of the instruction that is currently in the memory stage. In this case we forward D from the memory stage into A in the execute stage.

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRt)) ForwardB = 10

If this statement is true, it means that the current instruction that is in the Ex stage needs its input B to be the value of the instruction that is currently in the memory stage. In this case we forward D from the memory stage into B in the execute stage.

These results always go into the significant bit of the forward A and B outputs, and the significance of this was explained in the logic of the MIPS32 when deciding which type of forwarding we need.

MEM Hazard

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
    ForwardA = 01
```

If this case is true then we need to forward the value from D in the WB stage to A in the Execute stage

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
    ForwardB = 01
```

If this case is true then we need to forward the value from D in the WB stage to B in the Execute stage.

2.18.2) Evaluation

We directly followed the conditional statements given to us in the hand out, so this implementation is optimal.

2.19.1) Pipeline registers IF/ID, ID/EX, EX/MEM, MEM/WB.

In these sub-circuits we pass the values into the registers and out as outputs. They all have clock signals. We implemented these in sub-circuits so that our MIPS32 would have a cleaner look to it.