P1-ALU Design Document

Kevin Klaben kek228

1 Overview

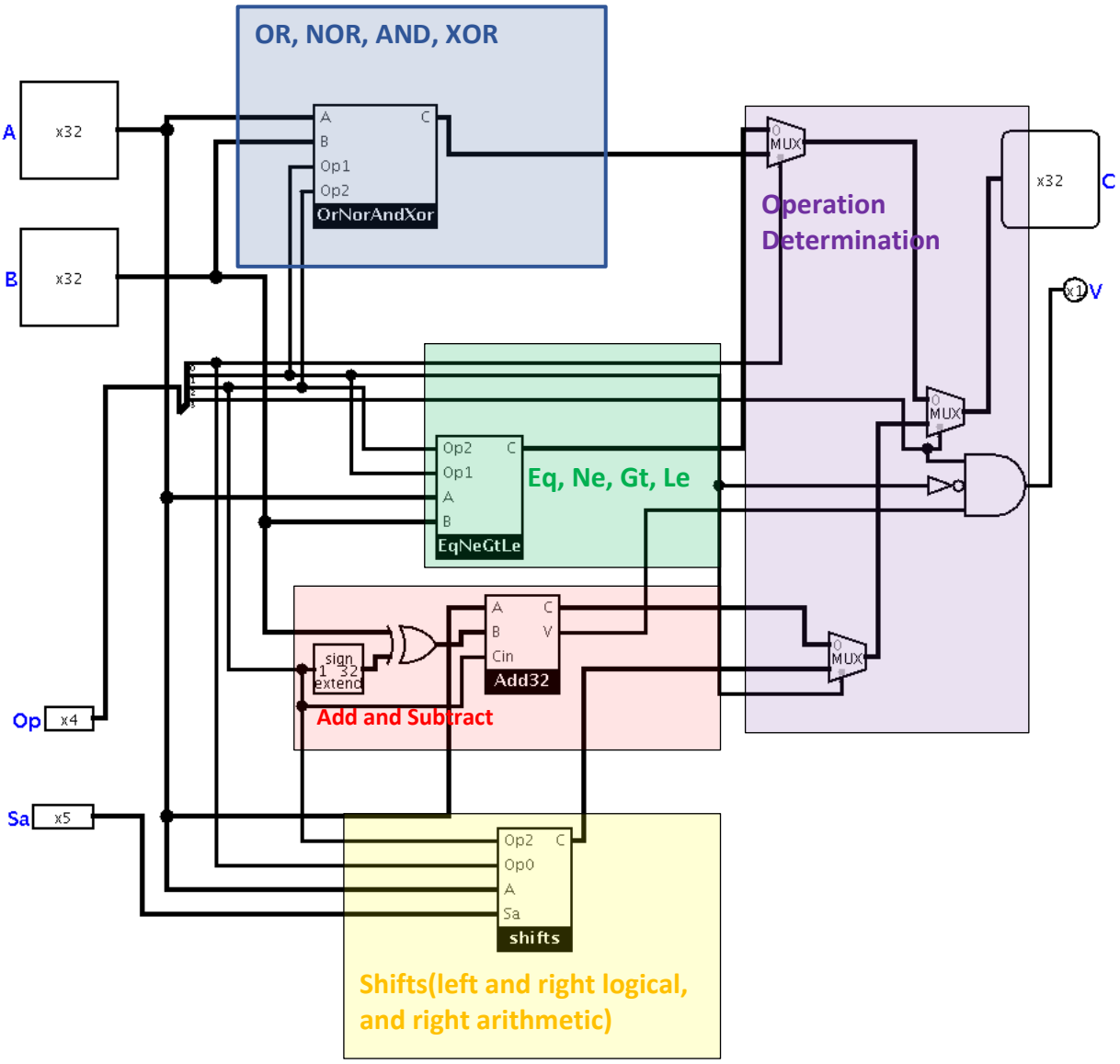
This project was focused on designing the first part of the RISC32 architecture specifically the Arithmetic Logic Unit (ALU). The ALU's purpose is to perform simple operations on two 32 bit input numbers A and B. Which operation is performed is determined by the 4 bit operation code labelled Op. The output of the ALU is the resulting one 32 bit number (C) and a single bit V indicating if there was overflow or not when the operation was performed. The ALU was implemented using simple gates, multiplexors, bit extenders, and splitters.

2 Component Design Documentation

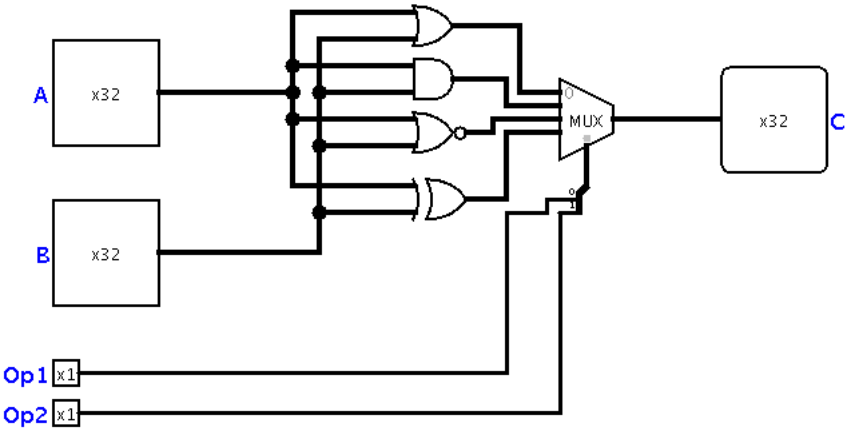
The ALU is made up of several major components including:

1. The OrNorAndXor which performs either an Or, Nor, And, or Xor operation depending on the Op code.
2. The EqNeGtLe which performs either Eq, Ne, Gt, or Le operation depending on the Op code.
3. The Add 32 which performs additions on two 32 bit numbers and out puts their sum and a value V saying whether or not overflow occurred.
4. The shifts which performs Logical Left, Logical Right, and Arithmetic Right shifts depending on the Op code.
5. In addition there are several multiplexors and gates that are not included in one of these sub circuits, these are used to determine which operation should be done and output to C and V based on the Op code.

Overall ALU diagram.



OrNorAndXor

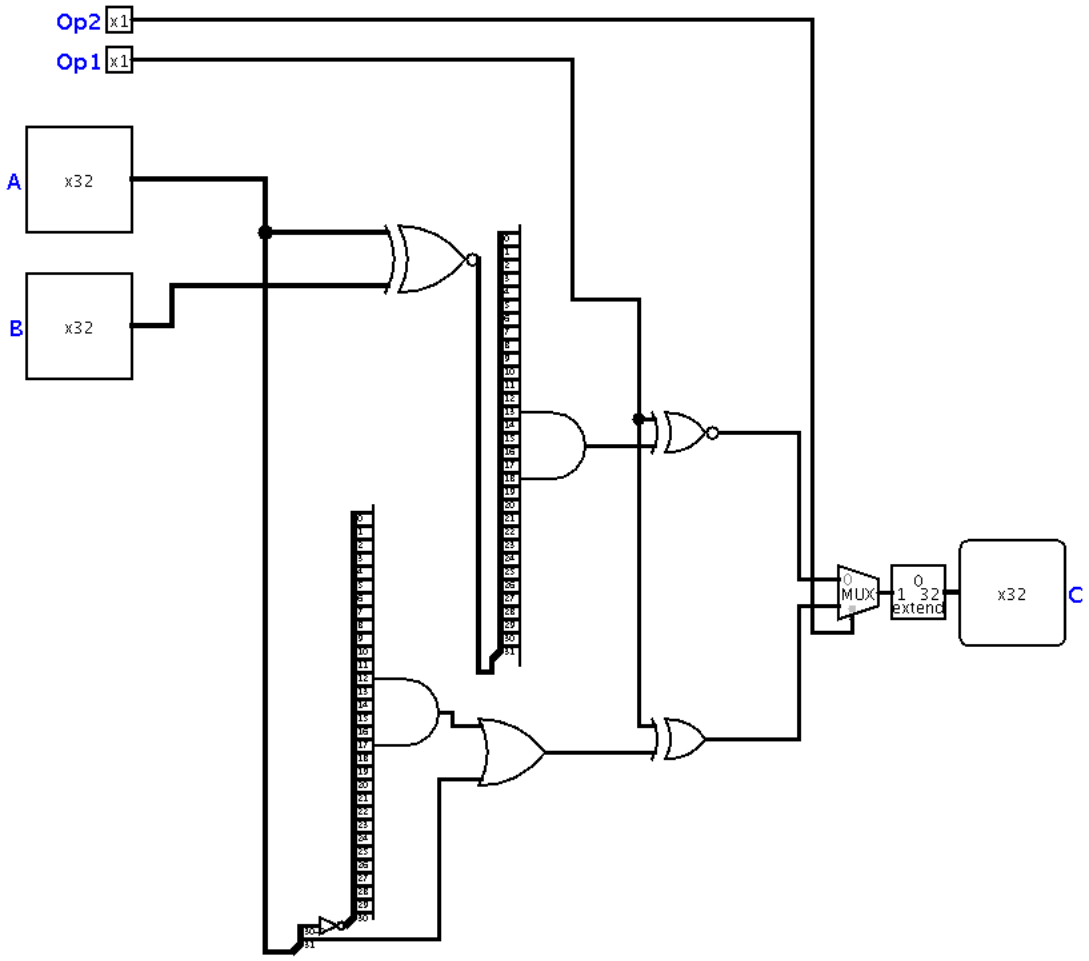


The OrNorAndXor performs the Or, Nor, And, and Xor operations depending on the Op code specifically the middle two bits of Op, specifically Op2 (second most signifcant bit of Op) and Op1 (second least signifcant bit of Op). The OrNorAndXor performs the each operation by simply connecting A and B to their corresponding gate (Ex- Or operation means A and B connected to an Or gate). The outputs from each of these operations is connected to a 4 to 1 multiplexor where the select bits are Op2 and Op1. A truth table below shows which operations signal is selected based on a given input Op2 and Op1.

Op2	Op1	Operation signal selected
0	0	Or
0	1	And
1	1	Xor
1	0	Nor

A 4-to-1 multiplexor was used because the four operations just happened to work out that they all can be distinguished by simply two bits. Although, several 2-to-1 multiplexors could have been substituted for this multiplexor, this we preferred as it is far simpler.

EqNeGtLe



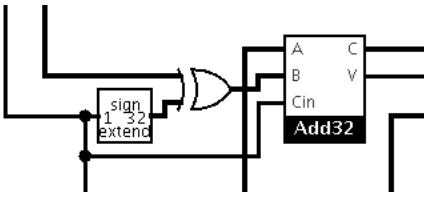
The EqNeGtLe takes in four inputs, A and B which are 32 bits each and Op2 (the second most significant bit of Op) and Op1(the second least significant bit of Op) which are each 1 bit. The output is C which is a 32 bit number which is 31 zeros followed by either a 0 or a 1 in the least significant bit. The EqNeGtLe performs the Eq, Ne, Le, and Gt operations. In the top half of the diagram the Eq and Ne operations are performed, A and B are put into an Xnor initially, this will mean that on any bit where they match a 1 will be output and on any bit where they do not match a 0 will be output. Thus, in the case where A and B are equal, 32 1's will be output from the Xnor gate and if A and B are not equal then at least one zero will be present in the output. Next this output is split and each bit is input to a 32 And gate. This And gate will out put 1 in the case where each bit is 1 and 0 if at least one of the inputs was a 0. Thus, 1 is output in the case where A and B were equal and 0 is output in the case where A and B were not equal. This output is then input to a Xnor with the other input being Op1. Op1 will be a 1 in the case where we are to be checking for equality and 0 in the case where we are to be checking for inequality. Thus the output from this Xnor (shown in the table below) is the correct output for the Eq/Ne operation.

Output from equality operation of A==B	Op1	Output form Xnor
0	0	1
0	1	0
1	1	1
1	0	0

The other half of the EqNeGtLe is the operation of Gt and Le. This is performed first by splitting off the 32nd bit of A from the rest of A as this tells us the sign of A. In the case where the sign bit of A that was broken off is 1 then we know that A is less than 0. The only other case that also must be checked is if A is equal to zero. In this case we know that every bit of A will be a zero. Thus, we take the remaining 31 bits of A that were split off and pass them through a not gate followed by a 31 bit And gate. In the case where these 31 bits were all zeros, they will be flipped all to ones and the output of the And gate will be 1, in other cases (where A is not zero) the output of this And gate will be 0. This when the sign bit and the output of this And gate are passed through an Or gate the output will be 1 if A is less than or equal to zero and 0 if A is greater than 0. Next Op1 and this output are put through a Xor gate. Op1 will be 1 if greater than was to be performed and 0 if less than or equal was to be performed. Thus, the output form this gate will be correct for whichever operation was specified either Gt or Le.

Together with the output from the Eq/Ne operation and the output from the Gt/Le operation, they are put through as mux with the control bit being Op2. In the case where Op2 is 1 the Gt/Le signal will be selected and in the case where Op2 was 0 the Eq/Ne will continue through. In order to satisfy the correct output form this final output bit for EqNeGtLe is extended with 31 0's in from of it and is the final output to C.

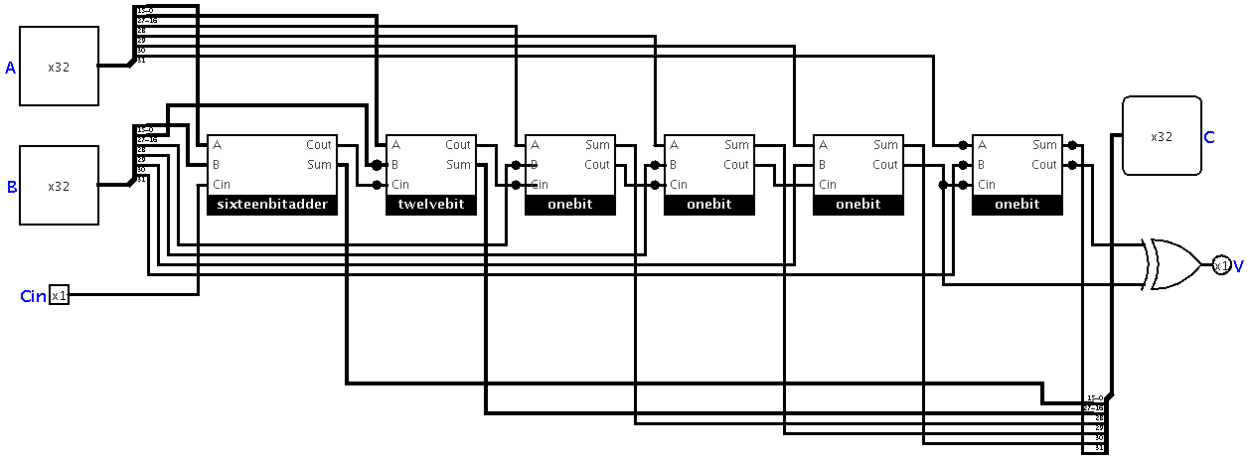
Add and Subtract



The add and subtract operation here takes in the second most significant bit of Op (Op2), B and A. In the case where the specified operation is A+B, Op2 will be 0. In the case where the specified operation is A-B, Op2 will be 1. Op2 is fed into a 1 to 32 bit extender which will create a repeat of 32 of whichever number Op2 already was (Ex- 0 -> 0000…0000 and 1-> 1111…1111). This extended version of Op2 is then fed into an Xor with B. The result of this gate is that in the case where Op2 was 0 and thus we do not want to modify B, all 32 digits of extended Op2 will be 0 and thus combined in an Xor, each bit of B will remain the same when combined with a 0 in the Xor. The converse is true for 1, in the case where Op2 was 1 and thus subtraction was to be done. Op2 will be extended to be a 32 1's and thus each bit of B will be swapped to the opposite bit when put in an Xor with a 1. Thus, in the case of subtraction, each bit of B is flipped which is exactly how subtraction works for two's complements, with the addition of 1 after the bits have been flipped. In

order to have this addition of 1 when the bits are flipped, the Cin the Add32 is simply Op2. In the case of addition, no additional 1 is needed and thus Op2 is fed in as a Cin of 0. In the case of Subtraction where the bits are flipped and an addition of 1 is needed, we Cin Op2 which in the case of Subtraction will be 1. The output of the Add 32 will be the result of the addition or subtraction specified to C and V will be 1 if the operation caused overflow and 0 if no overflow was caused.

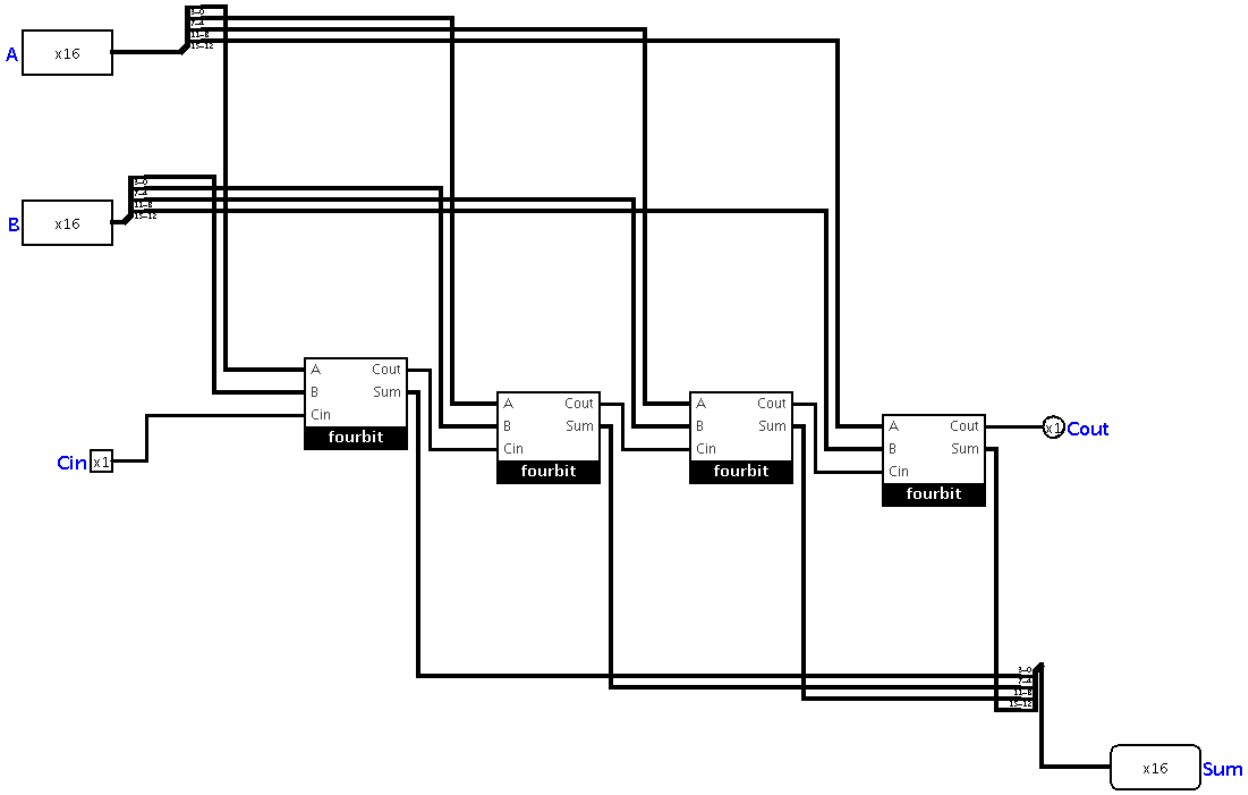
Add32



The Add32 takes in 3 inputs, A and B which are 32 bits and Cin which is the carry-in and is one bit. The output of Add32 is Sum which is the sum of A and B and V which is 1 bit indicating overflow. The implementation of Add32 is implemented by cascading a sixteenbitadder, a twelvebitadder, and four onebit adders. Each bit of A and B is added together bitwise through an individual adders, the least significant 16 bits in A is added to the least significant 16 bits in B through the sixteenbitadder, the next least significant 12 bits in A is added to the next least significant 12 bits in Busing the twelvebitadder. The remaining four most significant bits for A and B are added bitwise in individual onebit adders. The Sums from each individual adder are then combined together in order and output to sum as 32 bits. The Cout from each adder is used as the Cin for the next most significant bit adder, with the input Cin being the Cin used for the least significant bits. The Cout of the most significant onebit adders are used to determine the output to V. V is the output which is 1 if there is an issue with overflow, and 0 if there is no overflow issue. V is simple the output of an Xor with the most significant Couts as the two inputs. This because the only case where there is an overflow issue when adding two complement numbers is when the Couts of the two most significant bits are different, then an overflow issue is present.

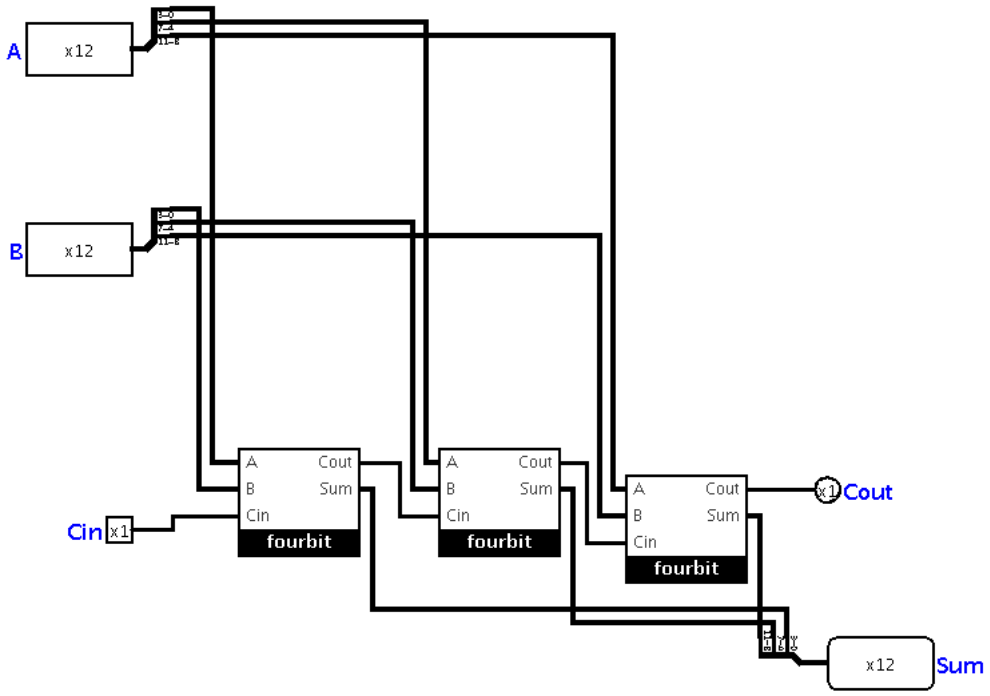
Cout1 most significant	Cout2 2nd most significant	V	Situation
0	0	0	No Overflow
0	1	1	Overflow
1	1	0	No Overflow
1	0	1	Overflow

Sixteenbitadder



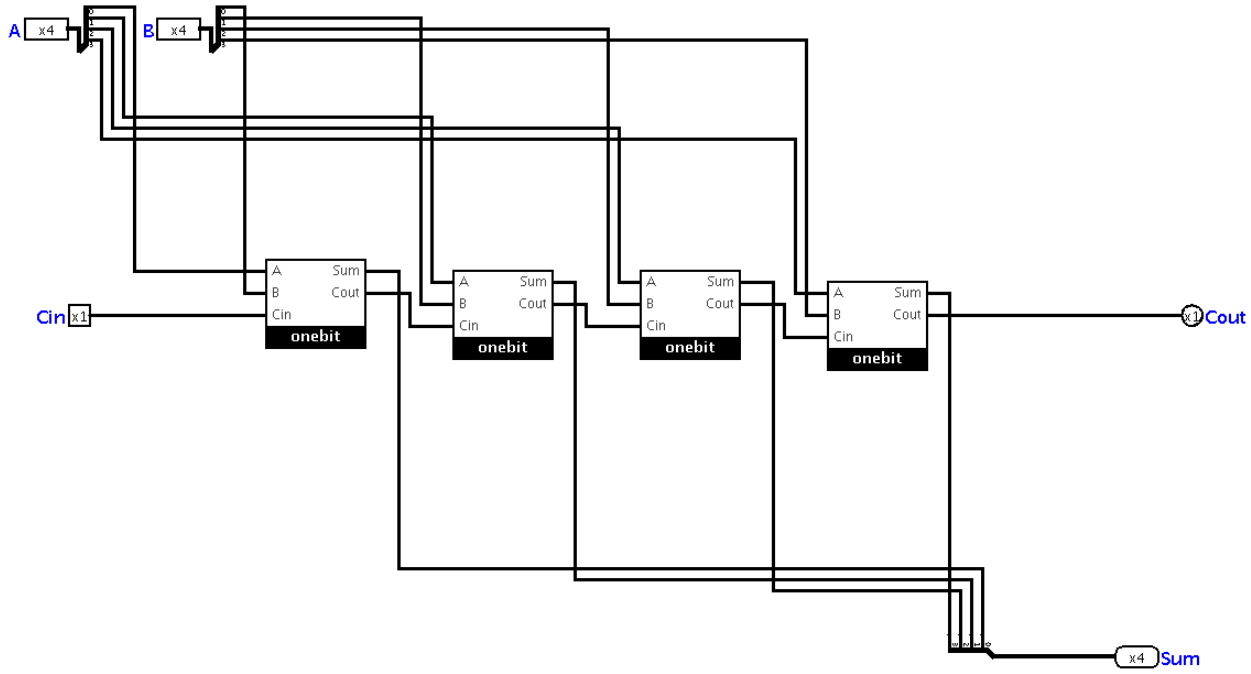
The sixteenbitadder takes in 3 inputs, A and B which are 16 bits and Cin which is the carry-in and is one bit. The output of sixteenbitadder is Sum which is the sum of A and B and Cout which is 1 bit, the carry out bit. The implementation of the sixteenbitadder is implemented by cascading four fourbit adders. Each bit of A and B is added together bitwise through an individual fourbit adder, the least significant 4 bits in A is added to the least significant 4 bits in B, the second least significant 4 bits in A is added to the second least significant 4 bits in B, and so on for all 16 bits. The Sums from each individual fourbit adder are then combined together in order and output to sum. The Cout from each one bit adder is used as the Cin for the next most significant bit adder, with the input Cin being the Cin used for the least significant bits and the Cout of the most significant bits being the Cout overall for the sixteenbitadder as a whole.

Twelvebitadder



The twelvebitadder takes in 3 inputs, A and B which are 12 bits and Cin which is the carry-in and is one bit. The output of twelvebitadder is Sum which is the sum of A and B and Cout which is 1 bit, the carry out bit. The implementation of the twelvebitadder is implemented by cascading three fourbit adders. Each bit of A and B is added together bitwise through an individual fourbit adder, the least significant 4 bits in A is added to the least significant 4 bits in B, the second least significant 4 bits in A is added to the second least significant 4 bits in B, and so on for all 12 bits. The Sums from each individual fourbit adder are then combined together in order and output to sum. The Cout from each one bit adder is used as the Cin for the next most significant bit adder, with the input Cin being the Cin used for the least significant bits and the Cout of the most significant bits being the Cout overall for the twelvebitadder as a whole.

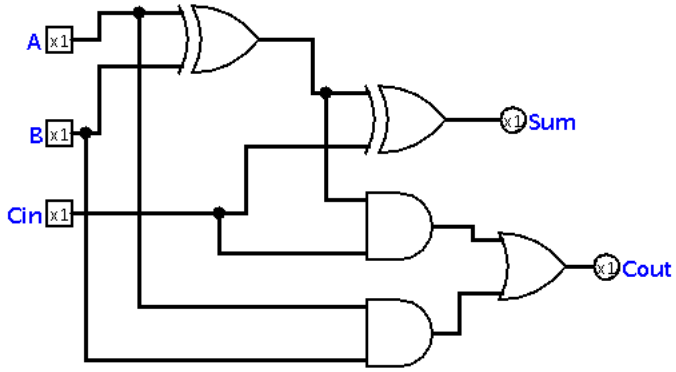
Fourbit



The fourbit adder takes in 3 inputs, A and B which are 4 bits and Cin which is the carry-in and is one bit. The output of Fourbit is Sum which is the sum of A and B and Cout which is 1 bit, the carry out bit. The implementation of the four bit adder is implemented by cascading onebit adders. Each bit of A and B is added together pairwise through an individual one bit adder, the least significant bit in A is added to the least significant bit in B, the second least significant bit in A is added to the second least significant bit in B, and so on for all four bits. The Sums from each individual one bit added are then combined together in order and output to sum. The Cout from each one bit adder is used as the Cin for the next most significant bit adder, with the input Cin being the Cin used for the least significant bit and the Cout of the most significant bit being the Cout overall for the four bit adder as a whole.

The four bit adder is a ripple carry adder which is not as efficient or fast as a loop-aheadcarry adder however, a ripple carry adder is far simpler to built and thus this tradeoff was made. All of the adding components (fourbit, twelvebitadder, sixteenbitadder, and Add32) are ripple carry adders for this reason, simplicity.

Onebit

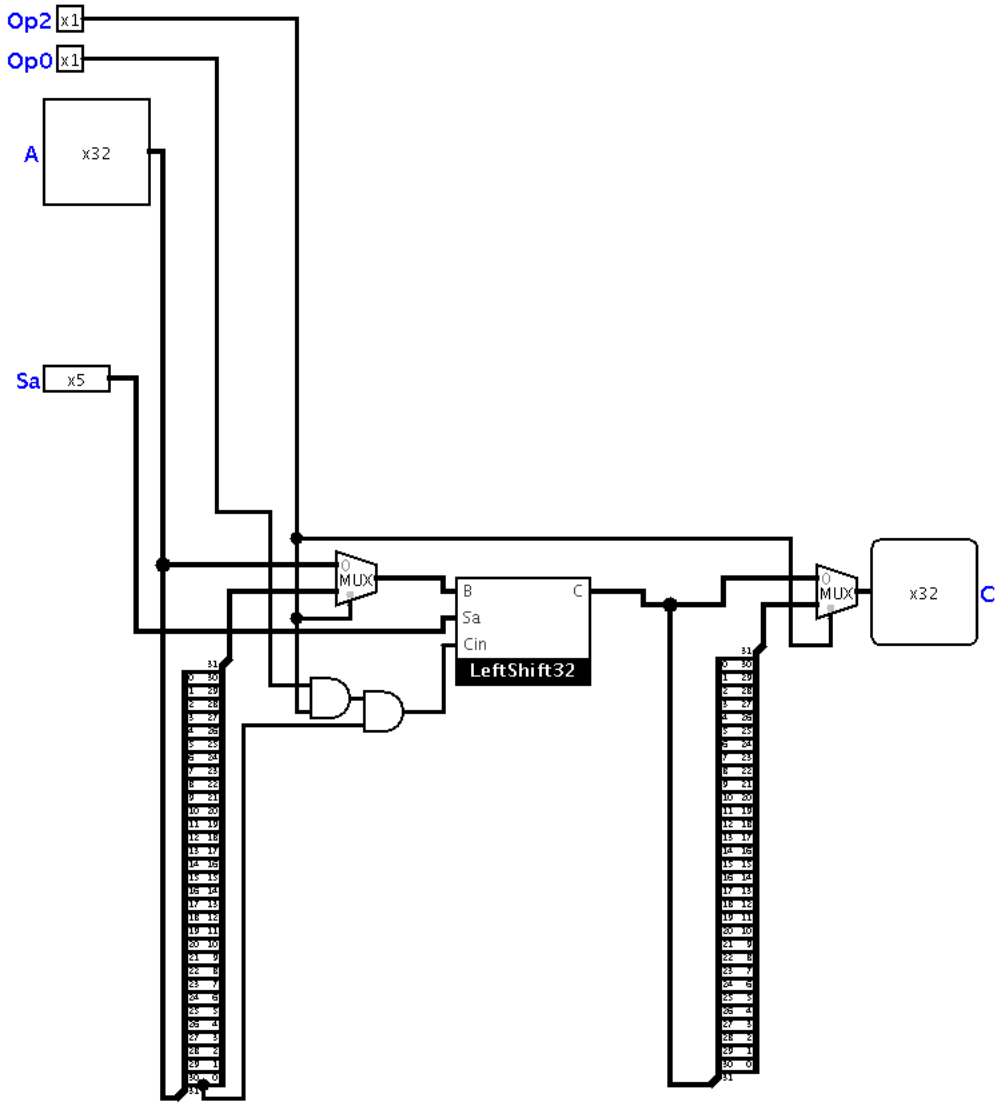


In the above diagram, A and B are the 1-bit inputs, while C_{in} is the carry-in bit. S is the output bit, while C_{out} is the carry-out bit. This is the optimal circuit for the following truth table:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Our circuit implements the optimal logical expression for both the resulting bit and the carry-out bit. As a result, our solution did not have any tradeoffs and is the optimal circuit for this purpose.

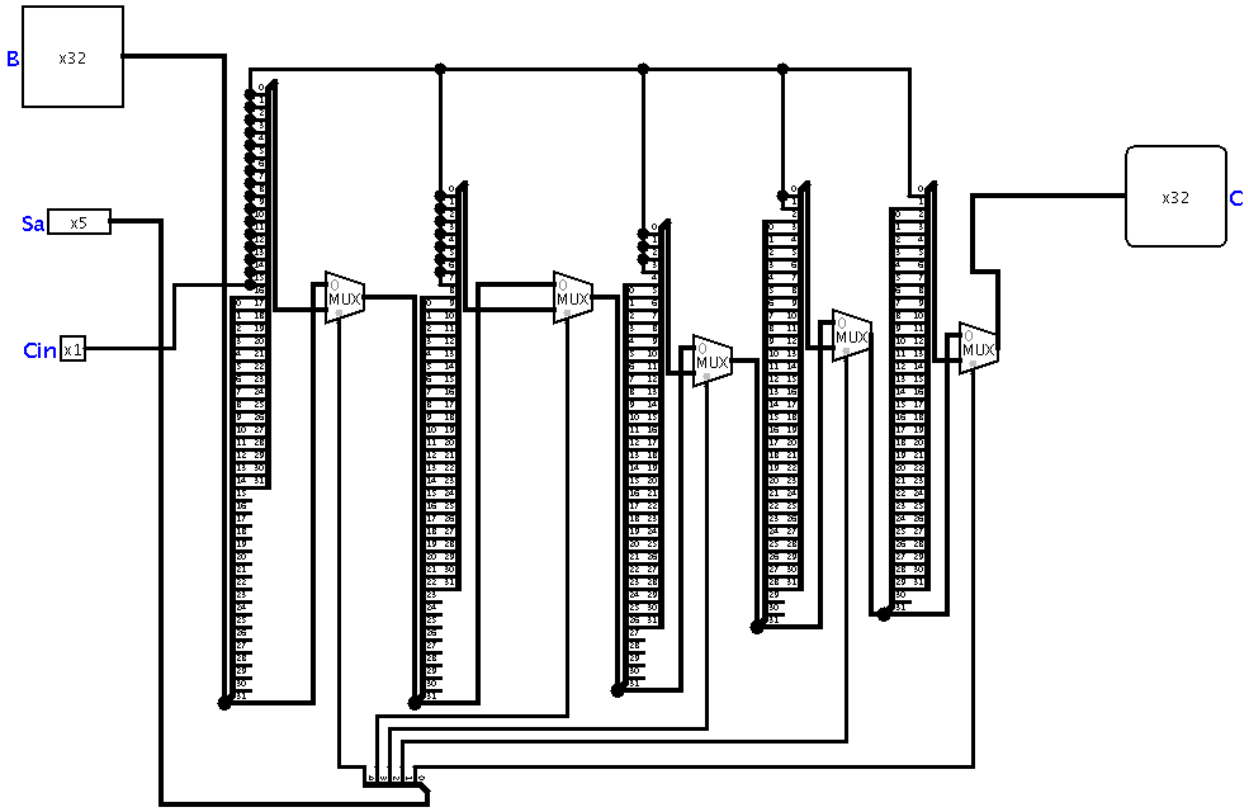
Shifts



The Shifts component consists of a main unit LeftShift32 which does an actual shift of B to the left by a certain amount (Sa) replacing the shifted out bits with a new bit (Cin). The other components seen here are used to decide which operation is being called for and preparing the inputs so that they can be used by the LeftShift32. Sa is input directly to the LeftShift32 as all three different shifts use Sa exactly the same. To prepare A for being shifted, A is fed into splitter which is then connected to a reversed splitter. This will have the effect of reversing the order of each bit in A. This reversed A is necessary as it can be used when the operation calls for a right shift. When a right shift is called for we can shift a reversed A to the left by the same amount and then reverse the order of the output to achieve the right shift result. To accomplish this the reverse A and the normal A are used as the signal inputs to a multiplexor whose control bit is Op2. In the case where Op2 is 0 then a left shift is called for and the normal A is output, while if Op2 is 1 a right shift is called for and the reversed A is output. This output from this multiplexor is fed into LeftShift32 as input B. To determine Cin for the shift operations we can see that in the case of a logical shift 0 is used as the Cin and in the case of an Arithmetic shift,

the sign bit or the 32nd bit of A is used as the Cin. Thus the only case where a 1 could potentially be needed is when the sign bit, the Op2 and the Op0 bits are all 1. Thus, these three signals are input to a series of And gates and the output is used as the Cin for the LeftShift32. The output of LeftShift32 must simply be reversed and then again using multiplexor with Op2 as the control bit, the reversed output or normal output is selected based on whether or not a left or right shift was called for. This multiplexor's output is the final output from shifts C.

LeftShift32



The LeftShift32 takes in three inputs B which is 32 bits, Sa which is 5 bits, and Cin which is 1 bit. Sa is split into its five bits as it determines the number of places that must be shifted. The most significant bit is handled first. In the case that this most significant bit is a 1 this indicated that B must be shifted by 16 places. This is accomplished by passing B into a splitter which is connected to another splitter such that the 16 lest significant bits on the first splitter are connected to the 16 most significant bits on the second splitter maintaining order. The 16 least significant bits on the second splitter are replaced by the value of Cin. The original unshifted signal of B and the now shifted and replaced signal B are then passed into a multiplexor where the control bit is the most significant bit of Sa. If this control bit is 1 the shifted signal of B is output and in the case where the control signal is 0 the unshifted B is output. The remaining bits of Sa are handled the same way cascading shifts of different amounts according to what a 1 in that place of Sa would indicate. The output of the final shift is used as the output to C.

Other Control multiplexors and Gates(Operation Determination)

After the four subcircuits have done their operations we are given an output signal from each (as well as a V signal for the Add32), the remaining work to be done is determining which of the four was called for. This is done through three multiplexors. The first decides between the EqNeGtLe and the OrNorAndXor signals. All the operations of the OrNorAndXor have a 0 as their most significant bit Op code and all the operations in the EqNeGtLe are indicated by a 1 in the most significant bit. Thus, a multiplexor is used with the most significant bit of Op as the control signal. The second decides between the Add32 and the shifts signals. All the operations of the Add32 have a 0 as their second least significant bit of the Op code and all the operations in the shifts are indicated by a 1 in the second least significant bit. Thus, a multiplexor is used with the most significant bit of Op as the control signal. For the third multiplexor we have the signals of EqNeGtLe\OrNorAndXor and Add32\shifts as the two signals to be selected and the control signal is the most significant bit of Op. The resulting output from this multiplexor is the final output to the ALU32 and is output to C. All that remains is to determine the value of V. The only case in which V is not 0 is when an addition/subtraction operation was done and the V value output from this operation was 1. We know that a subtraction/addition was called for if the most significant bit of Op is a 1 and the second least significant bit of Op is 0. Thus by using an and gate that takes in the v output from the Add32, the most significant bit of Op, and the signal of second least significant bit of Op after being passed through a not gate. This will output a 1 if an Add32 operation was called for and then the output from the operation was 1 otherwise 0 will be output. This output will indicate overflow and thus will be output to V.

Gate Count:

The total gate count is approximately 1482 gates.

Critical Path:

The critical Path is through the Add32 path and it is 70 gates.