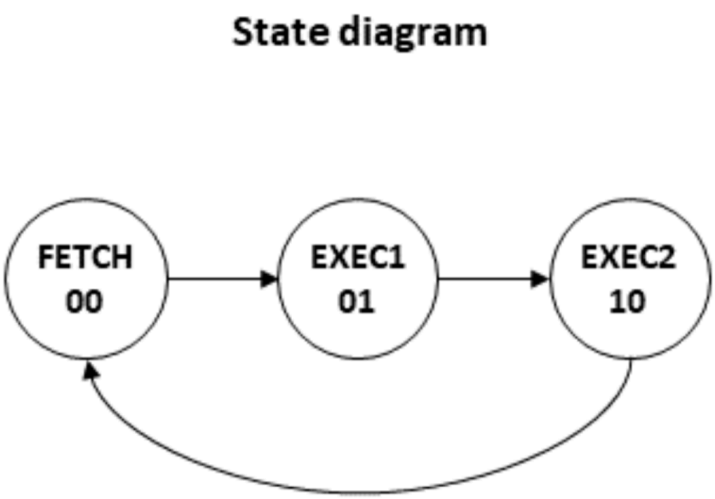
Lab 4

Constructing a MU0 CPU in ISSIE

Liam Browne

Table of Contents:

Finite State Machine:



We would like 3 states:

- FETCH will “fetch” the address for the next instruction and put it in Ram.In
- EXEC1 the machine word, i.e. Ram.Out, into the Instruction Block, in which the 4 most significant instruction bits will be separated from the 12 least significant bits.
- EXEC2 will execute arithmetic on the word that is fed into the ALU Block at the time for 3 cycle instructions. For 2 cycle instructions this state is redundant.

When in:

- FETCH we would like to go to EXEC1.
- EXEC1 we would like to go to EXEC2.
- EXEC2 we would like to go to FETCH.

Since we have 3 states we require a minimum of 2 bits as we need 3 or more combinations. Let these bits be A and B, where A is the MSB and B the LSB.

A_{in} and B_{in} will represent the current state:

A_{out} Karnaugh Map:

$A_{in}\backslash B_{in}$	0	1
0	0	1
1	0	x

B_{out} Karnaugh Map:

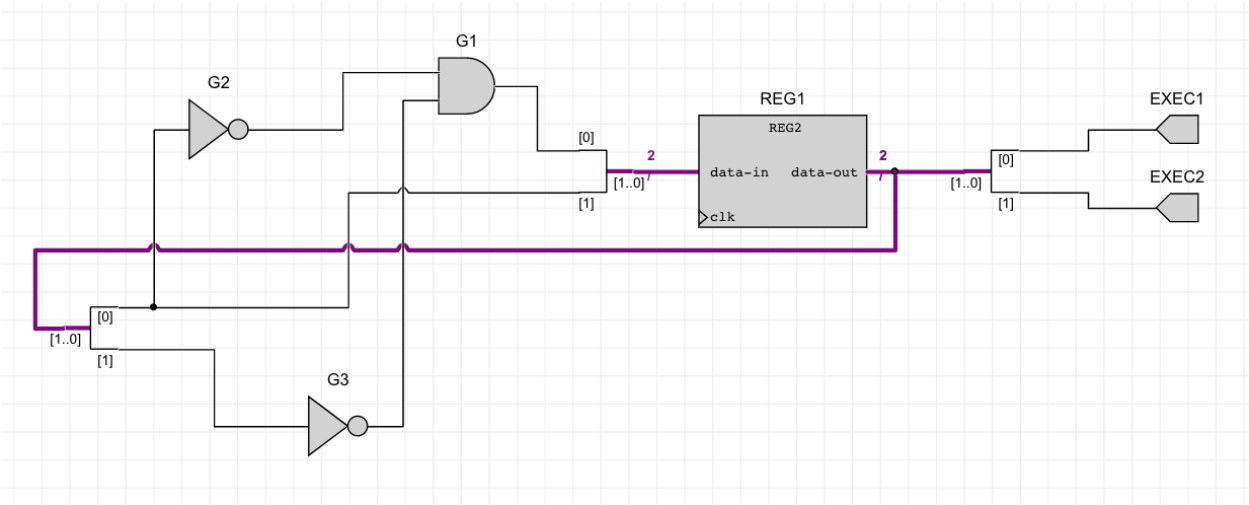
$A_{in}\backslash B_{in}$	0	1
0	1	0
1	0	x

Note: The red highlight represents the solving of the K-Maps

Boolean Expressions:

$$A_{out} = B$$

$$B_{out} = A'B'$$



Boolean Expressions Implemented in ISSIE with a State Machine.

During:

- FETCH, EXEC1 AND EXEC2 will be 0.
- EXEC1, EXEC1 will be 1 and EXEC2 0.
- EXEC2, EXEC1 will be 0 and EXEC2 1.

PC Block:

The PC Block is in charge of keeping the program running sequentially or jumping to a determined address, given a condition. It consists of a register that has the desired address going into PC.Register.In on EXEC2 and the same address going out of PC.Register.Out during the following FETCH cycle.

The input of PC.Register.In can be either PC+1 or N, i.e. the LSB 12 bits from Ram.Out. PC.Register.In will always be PC+1 on EXEC2 except for the following instructions:

- JMP, JMI, JEQ.

In addition, the PC Block has a STP input that will set the EN of the PC.Register to low when STP is high. This has the function of stopping the program by preventing the PC.Register to update.

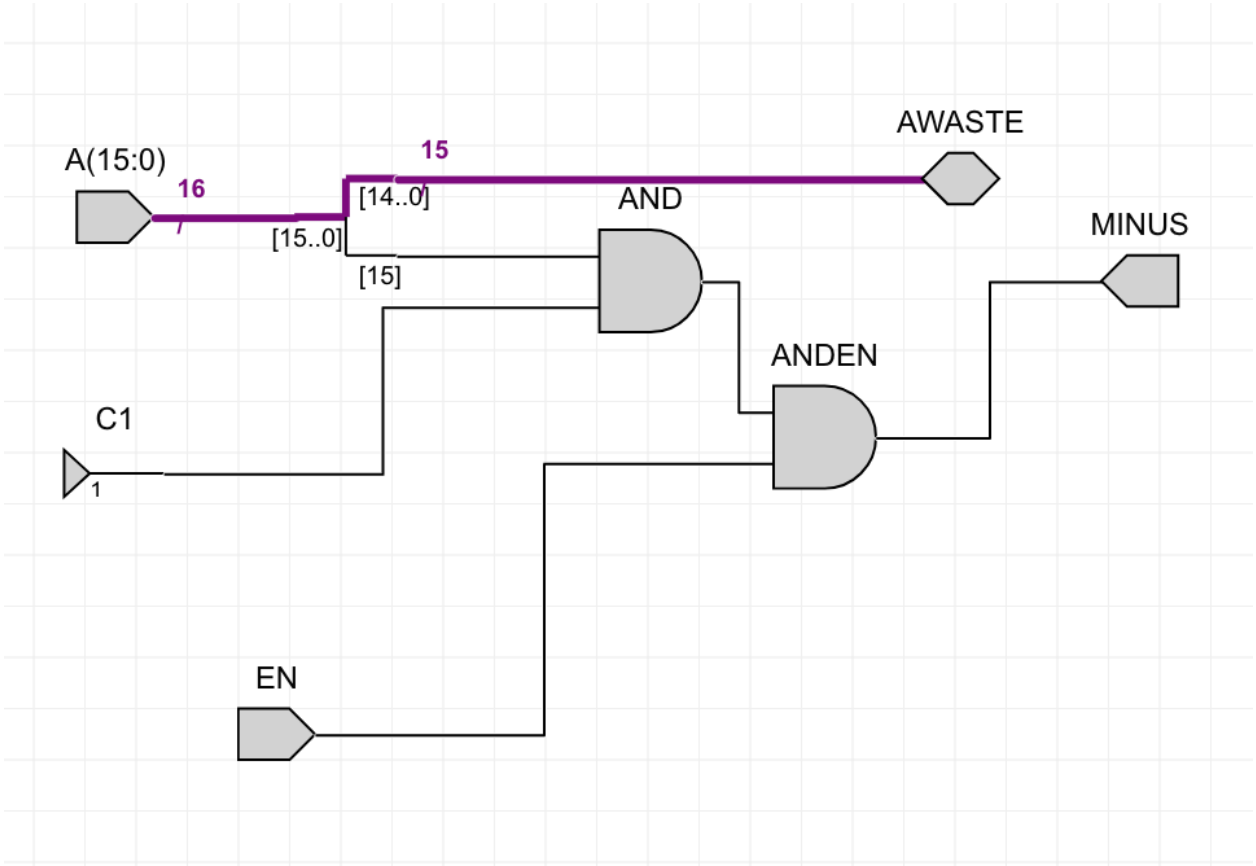
When the instruction is:

- JMP, PC.Register.In will be made to N during EXEC2.
- JMI, PC.Register.In will be made to N during EXEC2 if the value of A is negative.
- JEQ, PC.Register.In will be made to N during EXEC2 if the value of A is 0.

For JMI and JEQ we must create two separate blocks that check their respective conditions.

JMI Block:

Our CPU will be signed. In 2's complement a negative number is indicated by the MSB being 1. Hence, to determine whether A.Out is negative we simply must test whether its MSB is 1 or not.



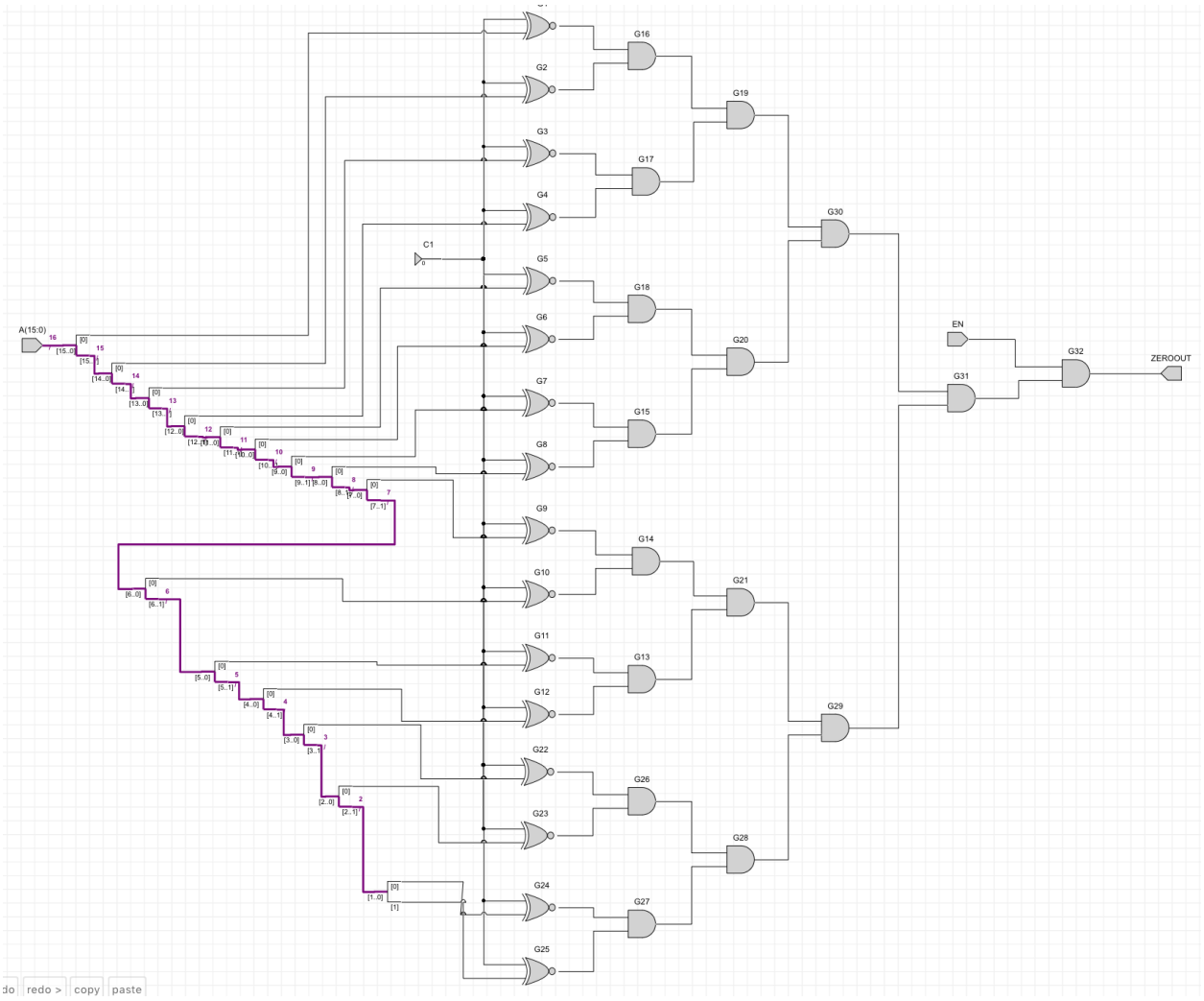
The JMI Block. Tests whether A.Out is negative or not.

The splitwire separates the least significant 15 bits and the most significant bit.

- MSB ANDed with 1.
    - If MSB is 1, MINUS = 1, i.e. A.Out is negative.
    - If MSB is 0, MINUS = 0, i.e. A.Out is positive.
- The EN input assures that this Block is only activated when a JMI instruction is executed.
- Note: During the write-up I realised that one of the AND blocks is redundant.

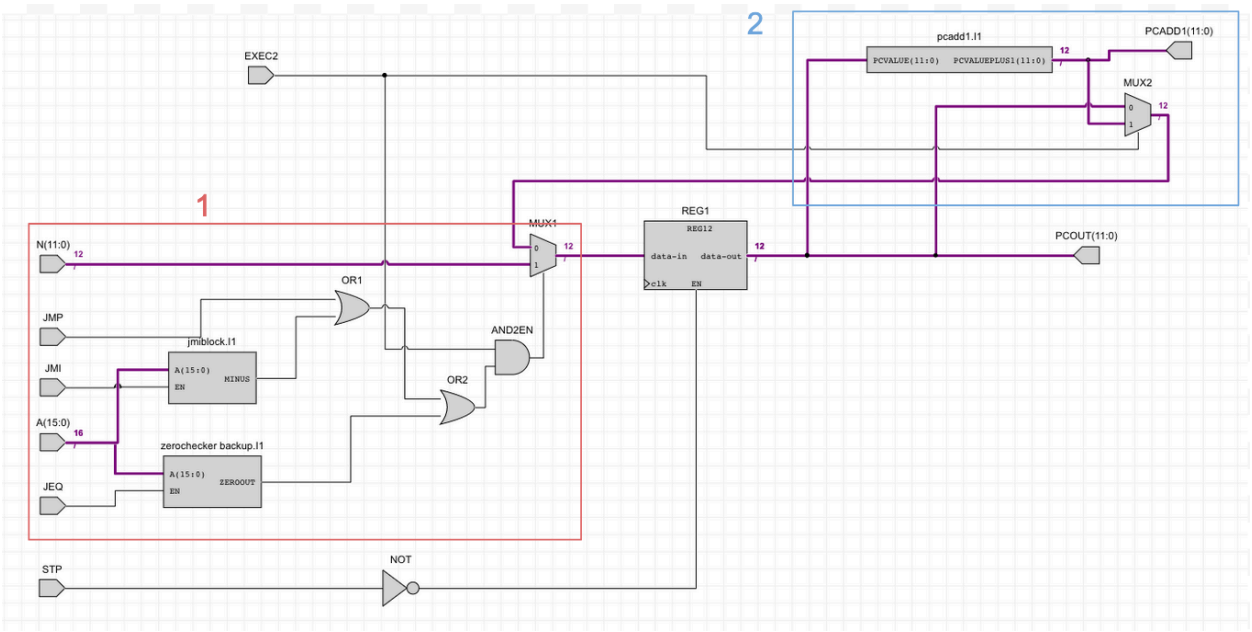
JEQ Block:

This Block must test whether A.Out is equal to zero. This is achieved by XNORing each bit of A.Out with zero. That is, if the numbers are identical, the output will be high. Since one of the inputs is a constant zero, the output will only be high when both numbers are zero. Each of these outputs were ANDed with each other in a chain of AND gates, such that if a single XNOR output is not high, the ZEROOUT output will be zero. If ZEROOUT is equal to 1 then all 16 bits in A.Out are equal to zero.



The JEQ Block. Checks whether A.Out is equal to zero or not.

PC Block:



Box 1:

- If JMP, JMI or JEQ is high, it will test their respective conditions with A.Out if they have one.
- If the conditions are met and the current state is EXEC2, the AND gate's out will be asserted, switching the output of the MUX from PC+1 to N. Thus, the address of RAM will be N during FETCH.

Box 2:

- The PCADD Block simply adds 1 to PC.Register.Out and is asynchronous. That is, it is PC+1.
- Since PC+1 should only be at PC.Register.In during EXEC2 a mux is placed at the output of PCADD such that PC+1 will only go through to PC.Register.In during EXEC2. In any other state PC.Register.In = PC.Register.Out.

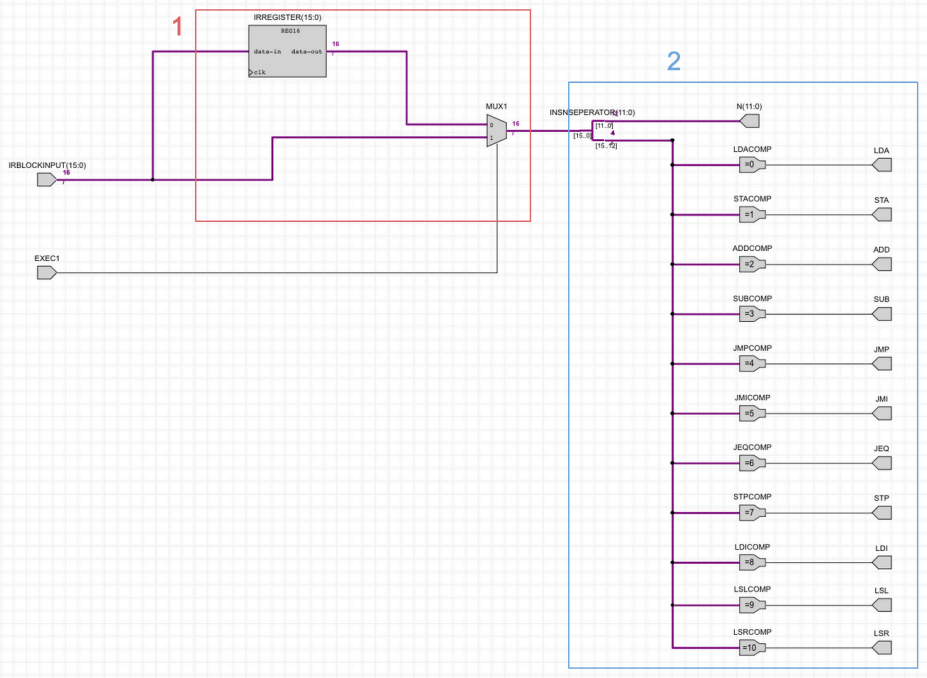
IR Block:

The IR Block is the block containing the Instruction Register. It is in charge of separating the instruction part of the machine word, i.e. the most significant 4 bits, from the least significant 12 bits, N.

- The input of the IR Block is Ram.Out.
- We would like to maintain the same instruction for EXEC1 and EXEC2, such that the IR Block ignores the Ram.Out during EXEC2.
- The 4 instruction bits should assert certain outputs depending on their value. In our case, the desired instructions corresponding to the 4 bits can be found below.

Product term name	IR(15:12)
LDA	0000
STA	0001
ADD	0010
SUB	0011
JMP	0100
JMI	0101
JEQ	0110
STP	0111
LDI	1000
LSL	1001
LSR	1010

The instructions and their corresponding binary code.



Inside the Instruction Register Block.

Box 1:

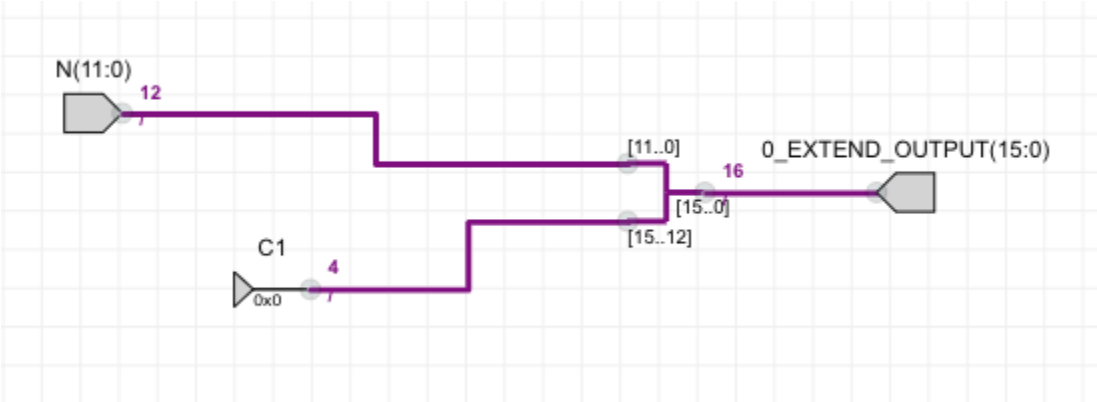
- During EXEC1 Ram.Out is the output of the MUX and also the input of IR.In.
- On EXEC2 the MUX's output is equal to IR.Out, which becomes Ram.Out from EXEC1, thus maintaining the original instruction throughout both cycles.

Box 2:

- A splitwire is used to separate the least significant 12 bits from the instruction word, while the 4 most significant bits go to the bottom of the split wire.
- The 12 LSB feed into the output N.
- Bus Compares are utilised to determine when a certain instruction is activated.
    - The value of the 4 MSB is compared to the decimal number on the Bus Compare. If the binary number corresponds with the decimal number, that instruction will be asserted. Only one instruction can be asserted at a time.
    - E.g. 0010 has a decimal equivalent of 2. Hence, the ADD instruction will be activated.

Zero Extend Block:

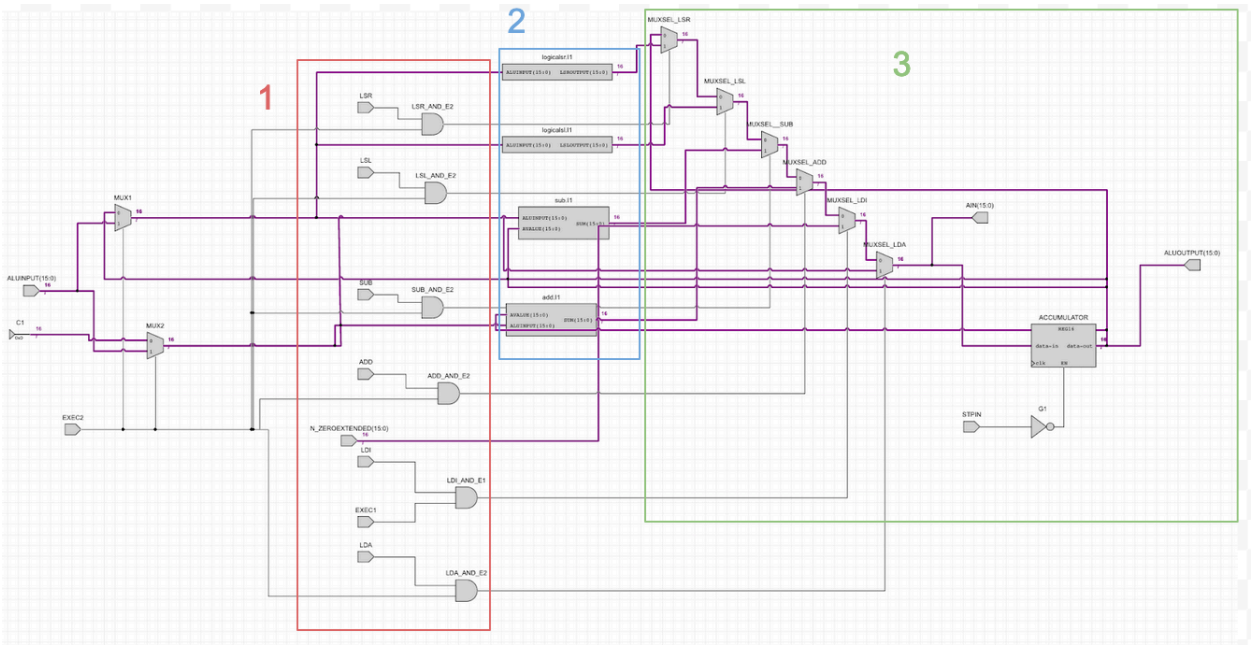
This block is required for the LDI instruction, which consists of loading the N of the machine word into A.In. However, N is only 12 bits while A.In is 16 bits. A solution to this is zero extending N by adding 4 more significant bits that are equal to zero.



4 More significant bits, all equal to zero, are added to N.

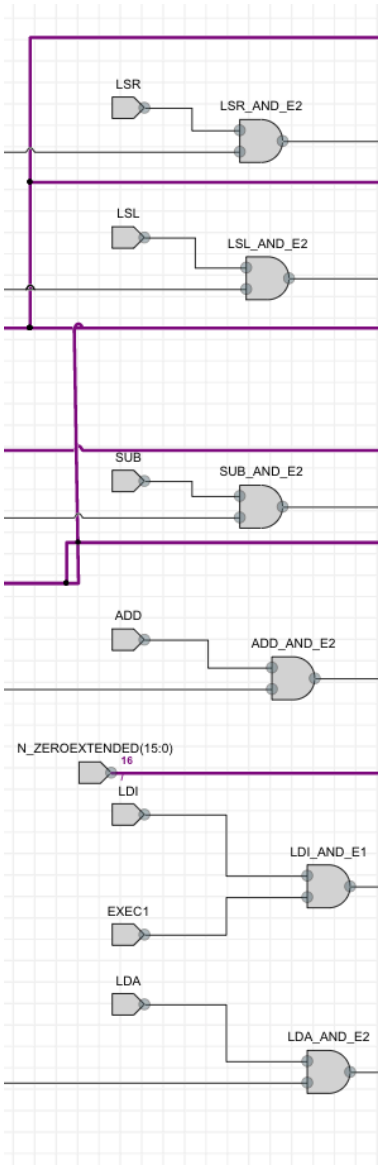
ALU Block:

The ALU Block is in charge of arithmetic operations and changing the accumulator register, A. In our CPU design, the ALU will execute adding, subtracting, logical shifts left (multiplication by 2), logical shifts right (division by 2), loading a number in an address and also loading the zero extended N part of an instruction word.



The entire ALU Block.

Box 1:

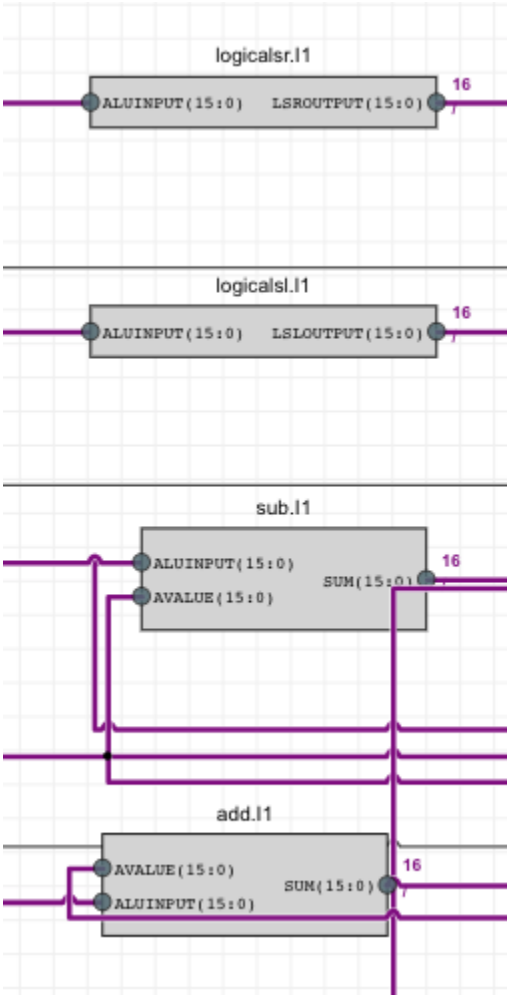


Screenshot of Box 1.

- The inputs belong to the respective instructions. Hence, when a certain instruction is executed, that input will be high.
- There is an AND gate for each instruction in the ALU. Each AND gate consists of an instruction input and an EXEC input. Which EXEC it is can be determined by looking at the last number of and AND gate's name.
- Most activate on EXEC2 as they require a word to be loaded out of Ram and sent into ALU.In during the same cycle. However, LDI is ANDed with EXEC1 since the zero-extended N value is available during EXEC1.

- The AND gates ensure that the accumulator is only changed during the desired cycle and not during other cycles as well (specific cases cannot be mentioned, yet implementing this halted bugs that caused the accumulator to change to an unwanted number during FETCH or when a certain combination of code in RAM was executed).

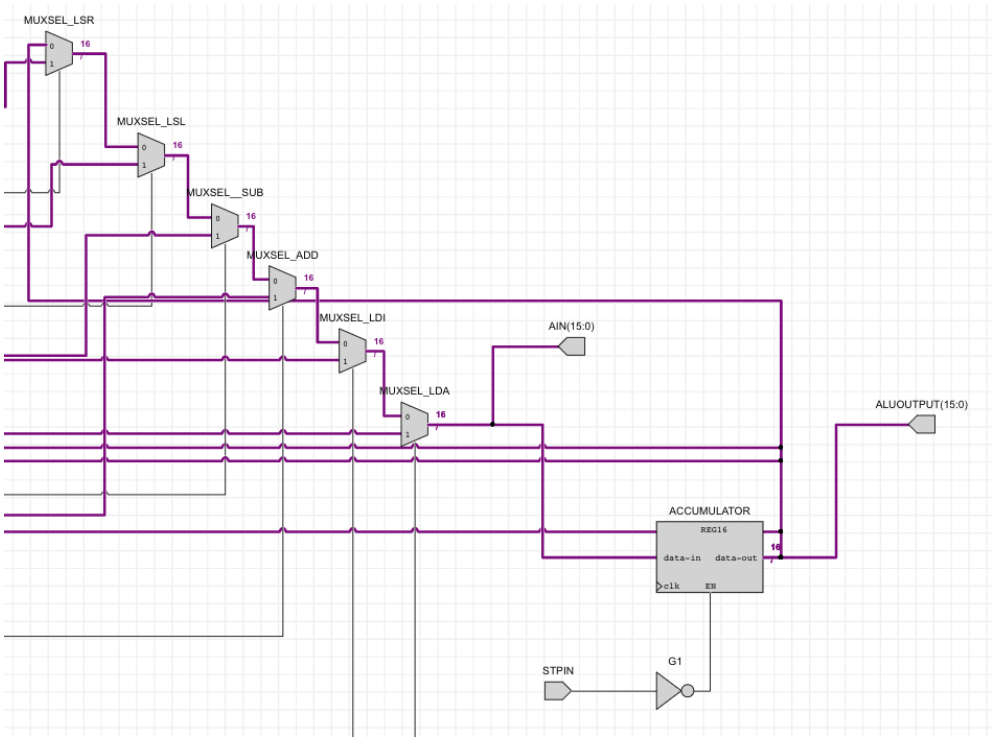
Box 2:



Box 2.

- The LSL Block shifts the bits of ALU.In left by 1 bit.
- LSR Block shifts the bits of ALU.In right by 1 bit.
- The SUB Block subtracts ALU.In from the number currently in the accumulator, i.e. A.Out. Hence the ALUINPUT and AVALUE inputs.
- Similarly, the ADD Block adds ALU.In to A.Out.
- Details of these blocks have been omitted, yet their logic can be found in Appendix 2.

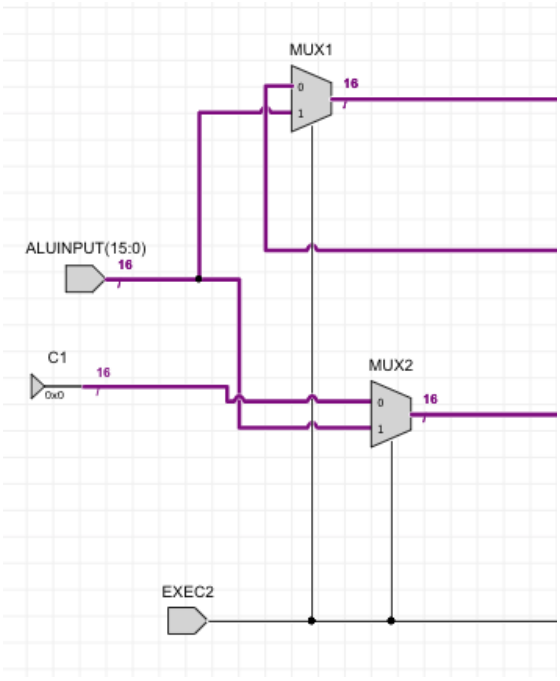
Box 3:



Box 3.

- The MUXs ensure that only the bits corresponding to the current instruction enter A.In, as all MUXs will be set to zero except the one for the asserted instruction (during the right cycle). Thus, the one MUX with its output equal to the input at 1 will feed right into A.In.
- If no instructions are activated or it is not the right cycle for the instructions, all MUXs will select the zero input such that A.In equals A.Out. This ensures that the accumulator does not change during the cycle.
- The STP is supposed to prevent the Accumulator from changing during a Stop instruction. However, it is superfluous as all instructions that feed into the ALU Block will be unasserted, meaning the accumulator will feed into itself. Hence, the A.Out will not change anyway.

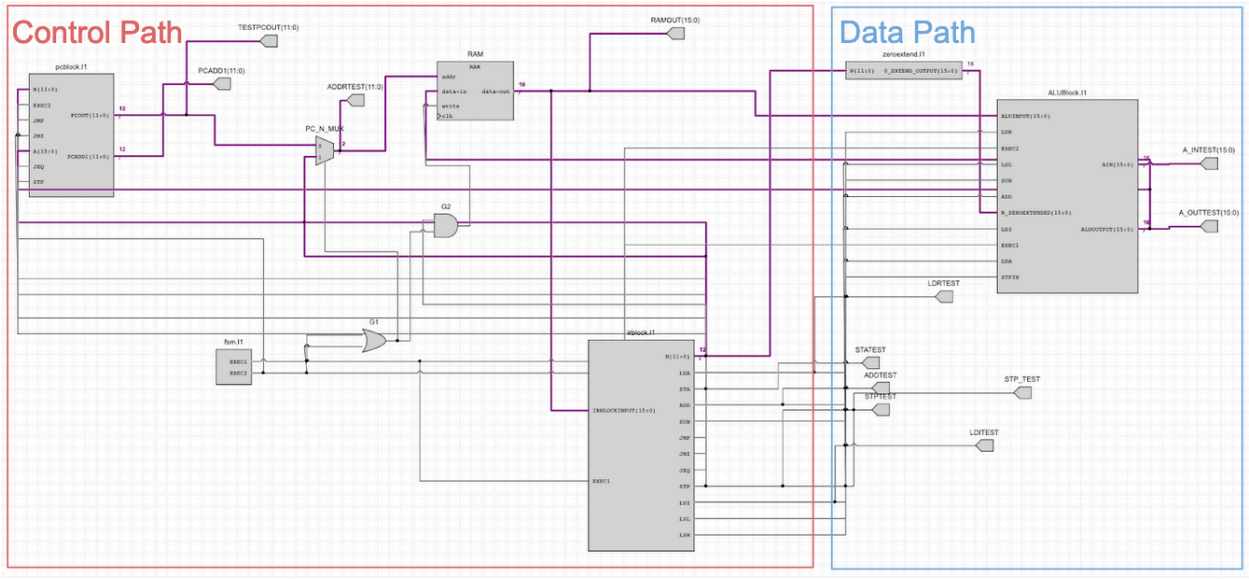
MUXs to left of Page:



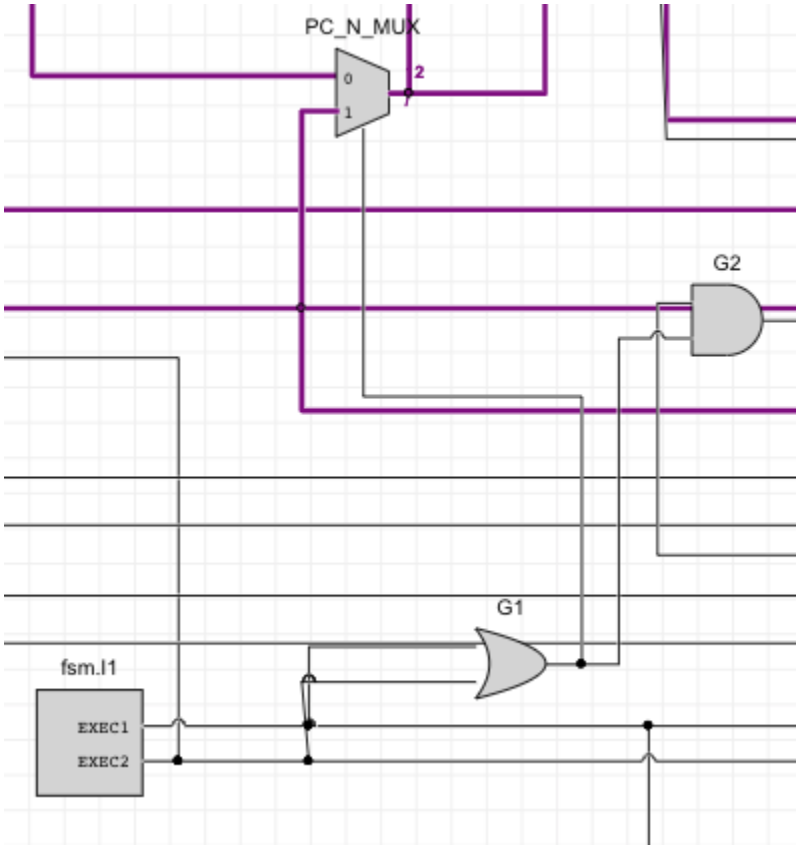
MUX1, MUX2 and C1.

- MUX1's output feeds into the MUX for the LDA instruction. Originally, it was implemented to make certain that the accumulator remains what LDA was. However, this MUX1 may, in fact, be redundant, as during all other cycles other than EXEC2 it will remain LDA due to A.Out feeding into A.In when it is not the right instruction nor cycle. That being said, it was implemented before the AND gates that make it redundant.
- C1 is a constant 16 bits, all of which are equal to zero, that feed into SUB and ADD during all cycles except EXEC2. It ensures that after EXEC2 SUB and ADD do not add/subtract ALU.In to/from A.Out. However, this MUX2 may have also become superfluous following the implementation of the AND gates. The AND gates ensure that even if ALU.In is added/subtracted to/from A.Out, that value will not enter A.In.

Compiling the CPU:



Control Path and Data Path.



Zoom-in of PC_N MUX, FSM, G1 and G2.

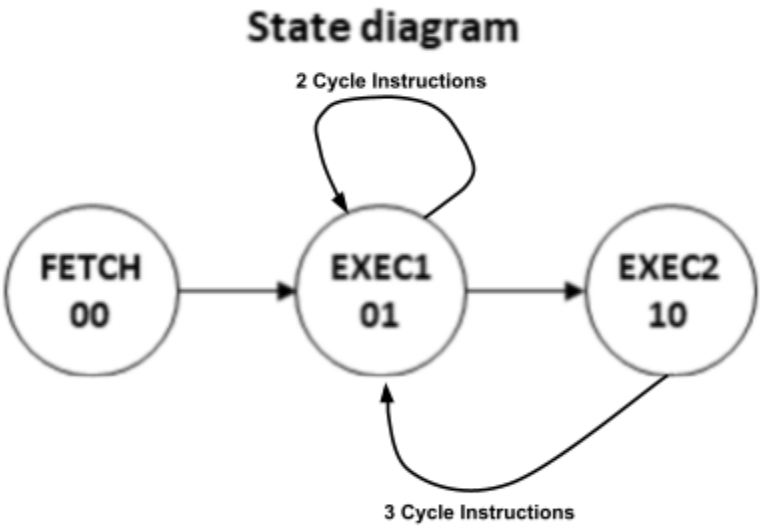
- G1 ORs EXEC1 and EXEC2, such that its output will be high during both of these states and low only during FETCH. Thus, PC_N MUX is PC.Register.Out during FETCH and N during EXEC1 and EXEC2.
- G2 is an AND gate with STA and the output of G1 feeding into it. We desire the RAM to have WRITE enabled during EXEC1, such that it writes the accumulator to the address corresponding to the 12 LSB of the instruction word. In theory, WRITE should be disabled during EXEC2 because we would still be writing into the RAM. However, since the accumulator and N do not change in EXEC2, we can leave WRITE enabled during EXEC2. For this reason is the output of G1 fed into G2. Note that this would not work for the optimised version though as the RAM must read PC.Register.Out during EXEC2 for the STA instruction.

Note: This non-optimised MU0 CPU treats LSL and LSR as instructions that read an address first and shift that number rather than immediately shifting what is in the accumulator. Thus, in this version, they are a 3 cycle instruction rather than 2.

Finite State Machine:

For the non-optimised CPU we had 3 states; FETCH, EXEC1, EXEC2. These states led to the one beside it sequentially and reset to FETCH on EXEC2. In the non-optimised CPU, regardless of the instruction, each instruction will take 3 full cycles – this is inefficient. The majority of instructions only require two cycles, while some require 3 cycles. Hence, the majority of instructions can return to FETCH after EXEC1. Furthermore, FETCH is a state during which the Address, i.e. Ram.In, is updated for the next instruction. However, it is possible to do this during EXEC1 or EXEC2, depending on whether the instruction requires 2 or 3 cycles respectively.

- Program starts at Address 000, meaning that we must start in FETCH.
- During EXEC1 for 2 cycle instructions, the instruction will be carried out and the Address will be updated within the same cycle for the next instruction. Following this, we return to EXEC1, as this is the state in which the new instruction is processed.
- During EXEC1 for 3 cycle instructions, the address for the data word, i.e. the word to undergo arithmetic operations, will feed into RAM.
- During EXEC2 Ram.Out will undergo the arithmetic operations corresponding to the instruction and the address for RAM will update to that of the next instruction. The next state will be EXEC1, for the same reason why EXEC1 is the next state after EXEC1 for 2 cycle instructions.



The desired state diagram.

For:

- 2 cycle instructions which change the accumulator's input, the accumulator will update when EXEC1 enters the new EXEC1. A.Out has updated value in new EXEC1.
- 3 cycle instructions which change the accumulator's input, the accumulator will update when EXEC2 enters EXEC1.

Previously when solving Karnaugh Maps we only had 2 bits that determined the next bit for A or B. However, now that which state we enter next also depends on how many cycles the instruction requires, we must consider a new variable, C, in our solving of Karnaugh Maps.

- For 2 cycle instructions C will be 1, for 3 cycle instructions C will be 0.

Instructions	
2 Cycles:	3 Cycles:
JMP	LDA
JMI	ADD
JEQ	SUB
STP	STA
LDI	RET
JSR	-
LSL	-
LSR	-

A_{out} Karnaugh Map:

$C\backslash A_{in}B_{in}$	00	01	11	10
0	0	1	x	0
1	0	0	x	0

B_{out} Karnaugh Map:

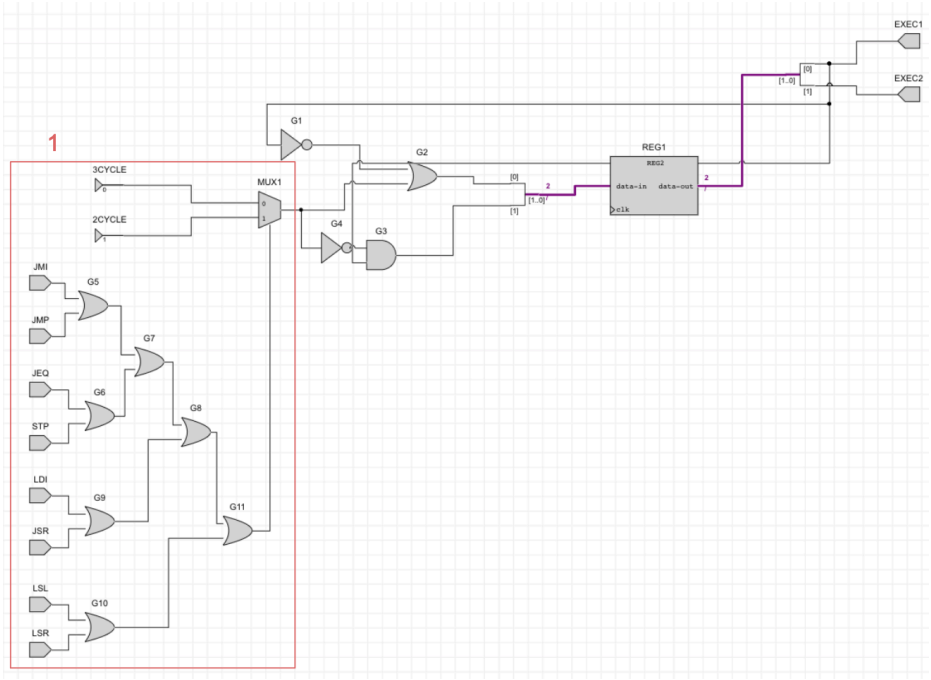
$C\backslash A_{in}B_{in}$	00	01	11	10
0	1	0	x	1
1	1	1	x	1

Note: Solving for 4 or 2 1s does not make a difference for the 1s in red.

Boolean Expressions:

$$A_{out} = BC'$$

$$B_{out} = B' + C$$



The implemented finite state machine.

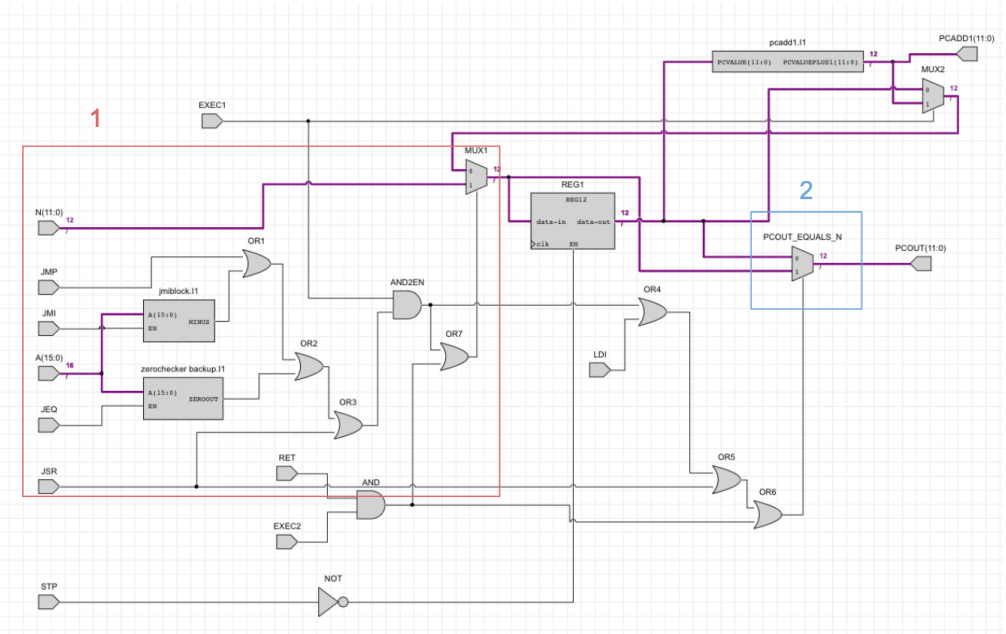
Everything not in Box 1 is the boolean logic for the solved K-Maps and the register for the finite state machine. The FSM's output feeds back into the logic and into EXEC1 and EXEC2.

Box 1:

- The output of MUX1 is the value of C.
- If any of the cycle 2 instructions are high, MUX1's output will be 1. Otherwise, it will be zero.
- Originally, I thought there were fewer 2 cycle instructions than 3 cycle ones. However, I soon found out that LSL and LSR are 2 cycle instructions. Thus, it may have been more efficient to have 1 feed into the zero-input of the MUX and 0 into the one-input, such that the MUX's output is equal to its one-input during a 3 cycle instruction. This would reduce the number of OR gates used.

PC Block:

Originally, the PC Block would send the PC+1 into PC.Register.In during EXEC2 such that PC+1 was the address for RAM during FETCH. This executed the new instruction during EXEC1. In the optimised CPU, we would like PC+1 to feed into address during EXEC1 for 2 cycle instructions, such that the next instruction is executed during EXEC1. For 3 cycle instructions we would like PC+1 to feed into PC.Register.In during EXEC1, such that address is equal to PC+1 in EXEC2. Hence, the new instruction will execute during EXEC1. Furthermore, for 2 cycle jump instructions, we would like Ram.In to be equal to N if the condition for the instruction is met.



Optimised PC Block.

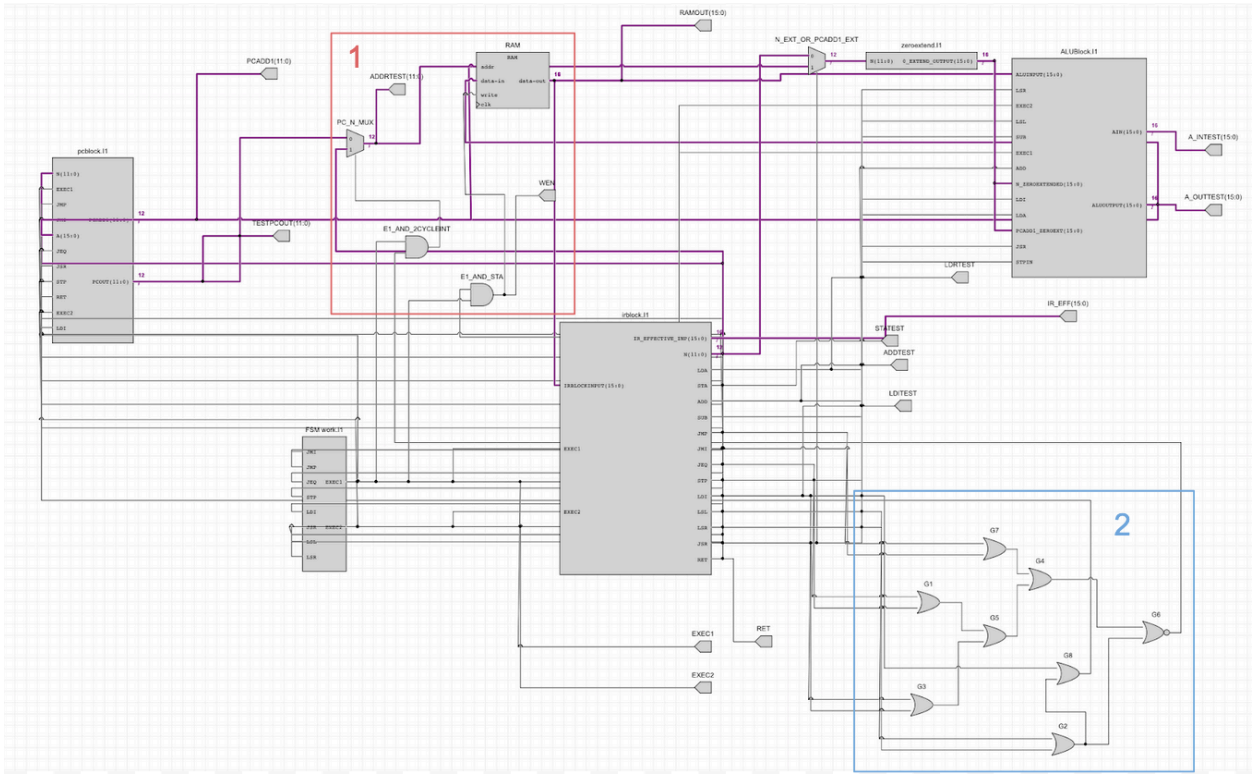
Box 1:

- During a 2 cycle jump instruction, given its conditions are met, the AND2EN gate will ensure that during EXEC1 MUX1 will have output equal to N.
- N now feeds into PC.Register.In and PCOUT_EQUAL_N in Box 2. Assuming the jump condition is met, the MUX in Box 2 will have output equal to N. Hence, the address will be N, meaning that in the next EXEC1 cycle Ram.Out will be equal to the instruction saved in address N.
- **Note Error:** if a condition is not met, PC+1 is supposed to be the output of the MUX in Box 2. However, this is not the case and thus if a, e.g. JMI, condition is not met, it will simply repeat the same instruction again and return the EXEC1. The new address will be PC+1 in this state, yet we go through 1 extra cycle unnecessarily.

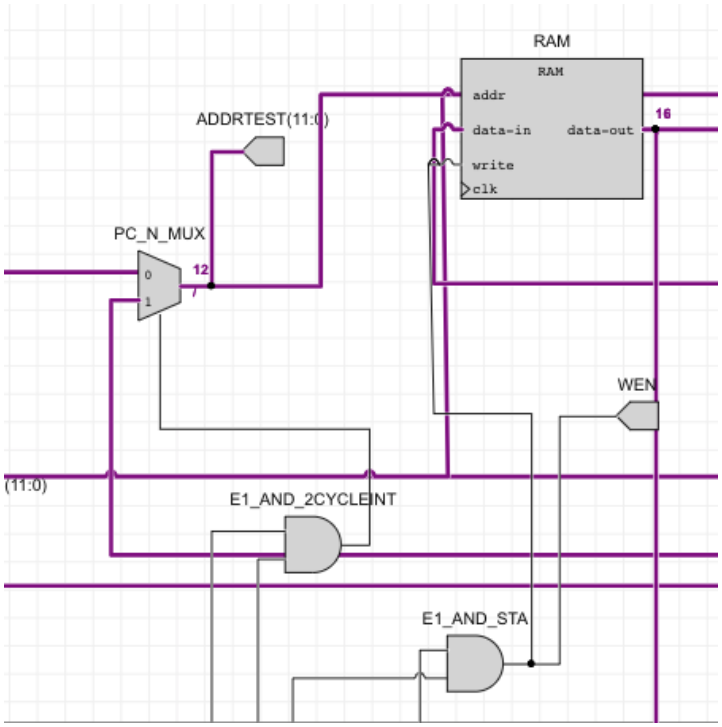
For LDI, LSL and LSR instructions the A.In is changed and we would like to have address be equal to PC+1 – all during the same EXEC1 state.

- During these instructions the MUX in Box 2 will have an output PC+1, making address equal to PC+1, too.

Changes to Compilation:



Compiled, optimised CPU with changes.



Close-up of Box 1.

Box 1:

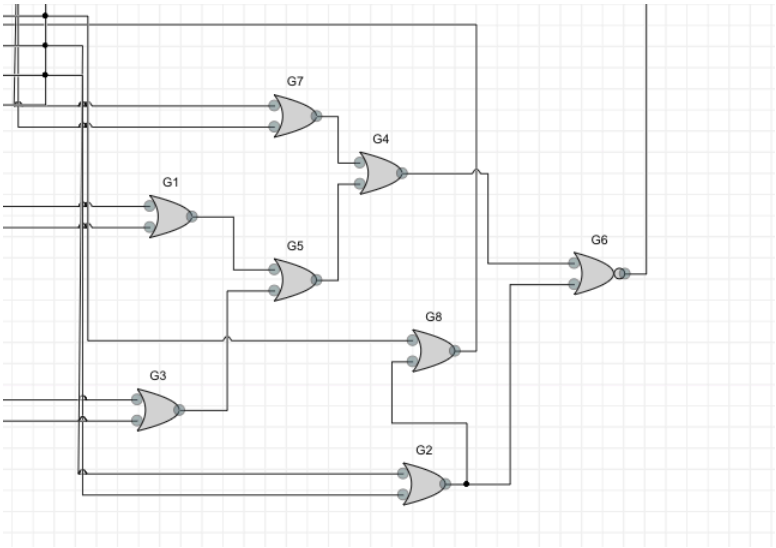
Conditions:

- PC_N_MUX has the output of the PC Block feeding into its zero-input and N into its one-input.
- For 2 cycle instructions, the MUX must have output equal to its zero-input, such that the address into the RAM is that of the next instruction.
- For 3 cycle instructions during EXEC1, the MUX must have output equal to its one-input, such that the address into the RAM is equal to N.
- For 3 cycle instructions during EXEC2, the MUX must have output equal to its zero-input, such that the address into the RAM is that of the next instruction.

We achieve the above requirements by implementing the E1_AND_2CYCLENT AND gate.

- EXEC1 feeds into the top wire and 2CYCLEENT into the bottom wire.
- During 2 cycle instructions in EXEC1, the top wire will be high but the lower wire will be low, such that the MUX will have output equal to its zero-input.
- During 3 cycle instructions in EXEC1, the top wire will be high and the lower while will be high too, such that the MUX will have output equal to its one-input.
- During 3 cycle instructions in EXEC2, the bottom wire will be high, yet the top wire will be high as we are no longer in EXEC1. Thus, the MUX will have output equal to its zero-input, i.e. the address of the next instruction.

The E1_AND_STA AND gate ensures that the RAM is only writing during EXEC1 for an STA instruction, such that PC+1 can be fed into the address during EXEC2.



Close-up of Box 2.

The gates in Box 2 display the logic for 2CYCLENT, i.e. the lower input of the AND gate for the PC_N_MUX.

- ORs all 2 cycle instructions together and NOTs the result.
- Output of G6 will be low for 2 cycle instructions and high for 3 cycle instructions.

Implementing JSR:

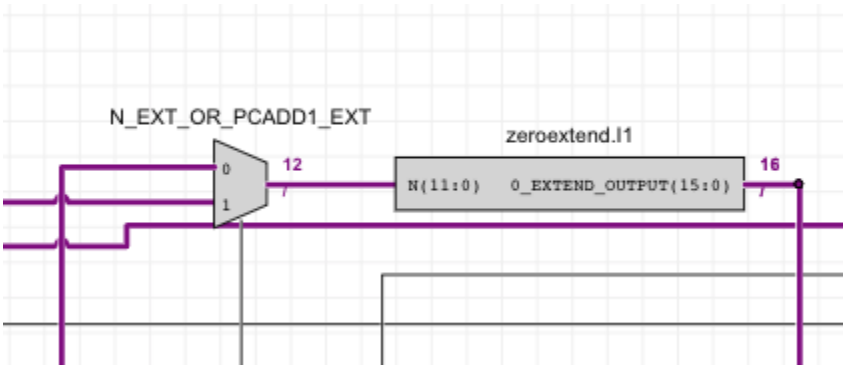
Assembly	Operation	Notes
JSR N	`A:=PC+1 ; PC:=N`	Jump to Subroutine. Jump storing return address in `A`

Description of the Jump to Subroutine instruction.

Jump to Subroutine (JSR) is a 2 cycle instruction that sets A.In equal to PC+1 and the PC Block's output equal to N.

- PC+1 is 12 bits so must be zero extended.
- PC+1 must feed into A.In during EXEC1.
- N must feed into PC.Register.In and into address during EXEC1.
    - For 2 cycle instructions the next state will have N+1 available.
    - For 3 cycle instructions, N+1 will feed into address during EXEC2.

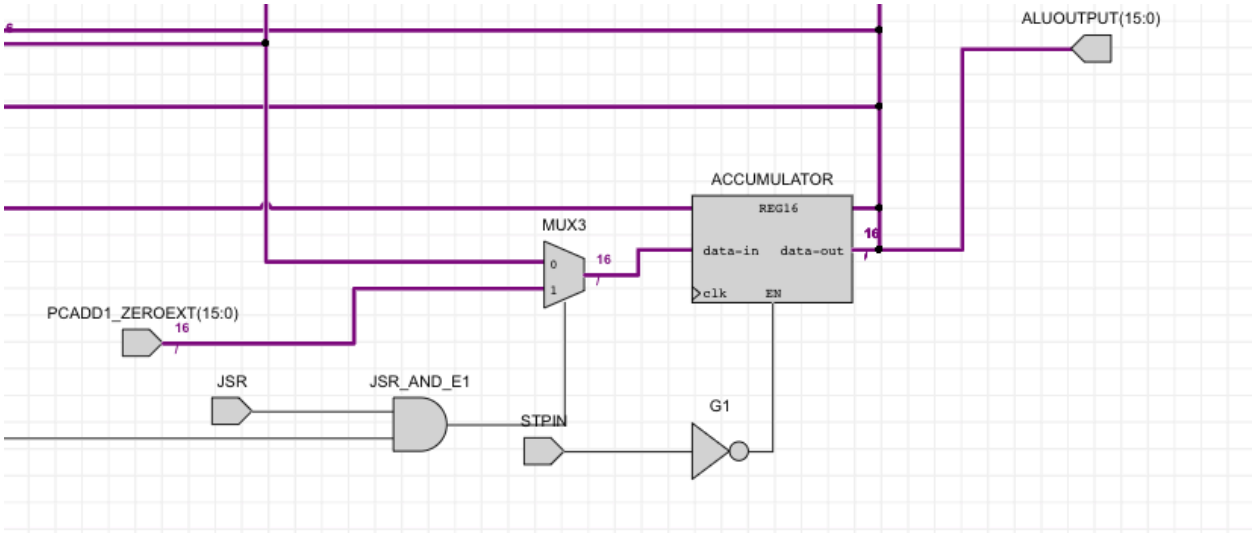
Zero Extending PC+1:



Logic that zero extends PC+1 from 12 bits to 16 bits.

The MUX has N feeding into its zero-input and PC+1 into its one-input. Its selector is the wire connecting the JSR, such that the MUX's output will be PC+1 when JSR is executed. The extended PC+1 is then fed into the ALU block.

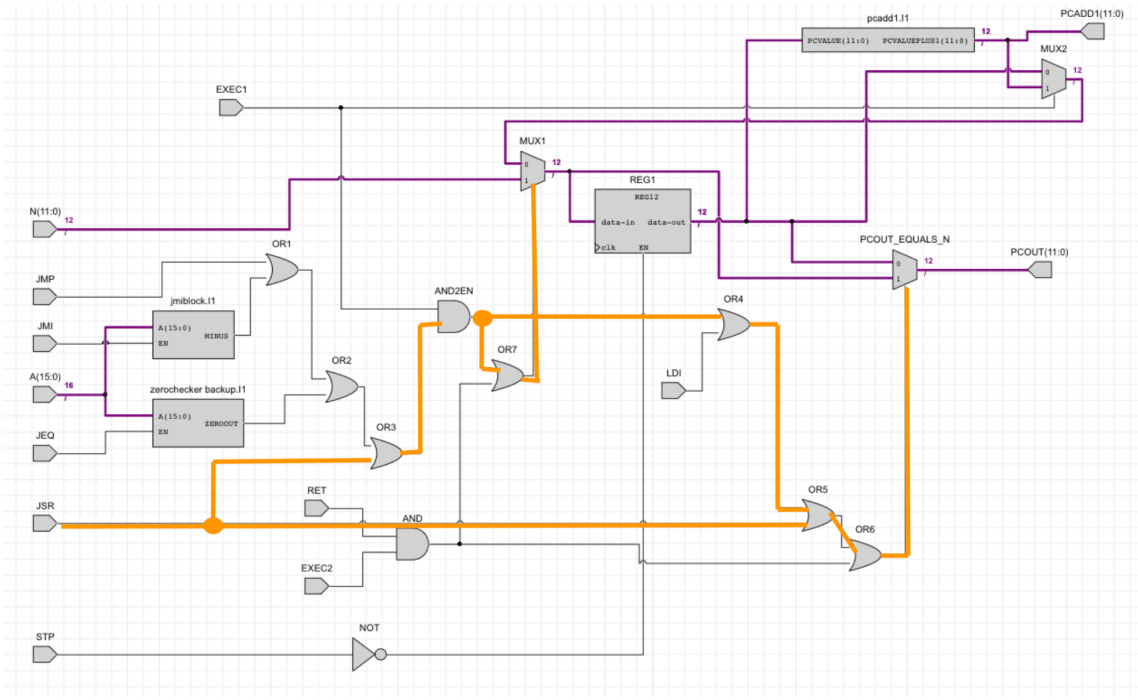
Feeding 16 bit PC+1 into A.In:



JSR logic in ALU.

When the JSR instruction is executed during EXEC1, MUX3 will have output equal to the 16 bit PC+1. This feeds into A.In. When JSR is not being executed, MUX3's output will be equal to its zero-input whose value depends on the current instruction and state. **Note**, for 2 cycle instructions an AND gate for the instruction and EXEC1 may not be necessary, i.e. the instruction input could feed directly into the MUX select. This is because 2 cycle instructions will only be high during EXEC1 and no other state. For 3 cycle instructions the AND gate between the instruction input and EXEC2 is necessary due to the instruction being high during EXEC1 too.

Setting PC equal to N:



PC Block with JSR-relevant wires highlighted in orange.

When JSR is executed, MUX1 will have output equal to N such that:

- PC.Register.In = N.
- Address = N, i.e. PC = N.

Note: JSR feeding into OR5 is redundant as AND2EN's output will always be high at the same time OR5 is high.

- AND2EN may also be redundant for all 2 cycle instructions as the current 2 cycle instruction will always be high at the same time EXEC1 is high.

Implementing RET:

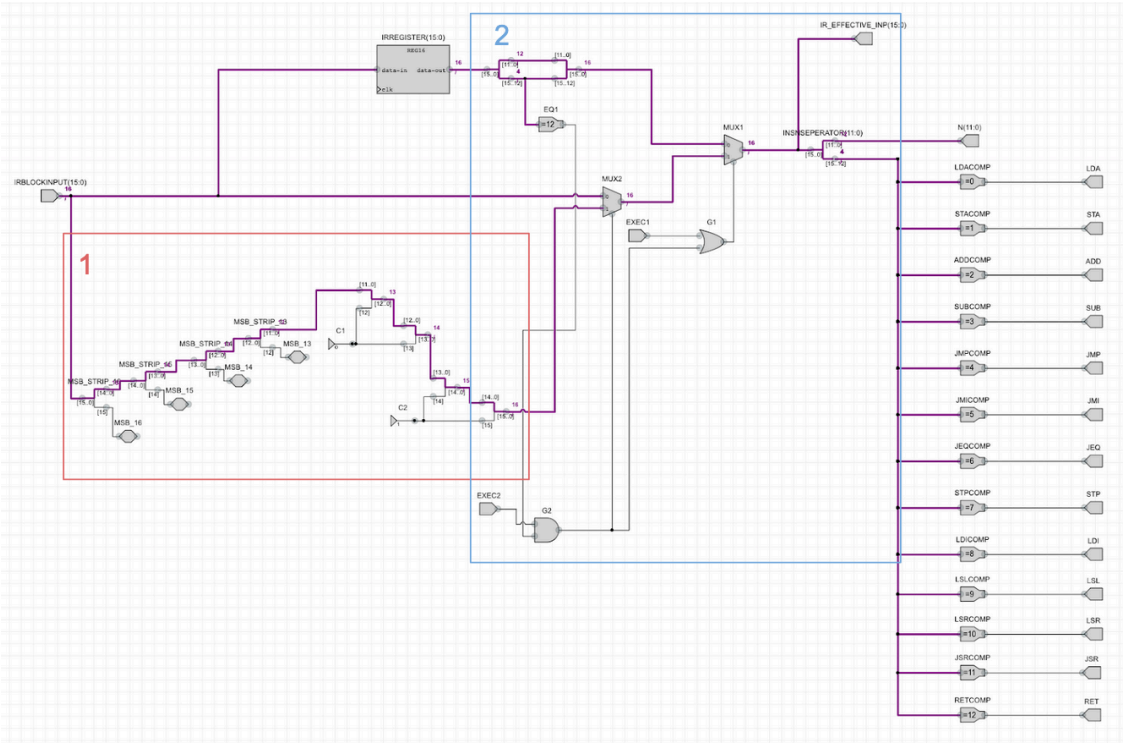
Assembly	Operation	Notes
RET N	`PC := Mem[N]`	Return to instruction whose address is stored in `Mem[N]`

Description of the Return instruction.

Return (RET) is a 3 cycle instruction that looks up the address of the least significant 12 bits of its instruction word and feeds the N of that word into PC.Register.In.

- During EXEC1 the RET wire does not do anything. It is in this cycle that N of the RET instruction is sent into the address of the RAM.
- In EXEC2 the N of the word that has just been looked up is sent into PC.Register. In and into the address of RAM.

Adjustments to the IR Block:



Logic adjusted in IR Block to make certain that RET functions.

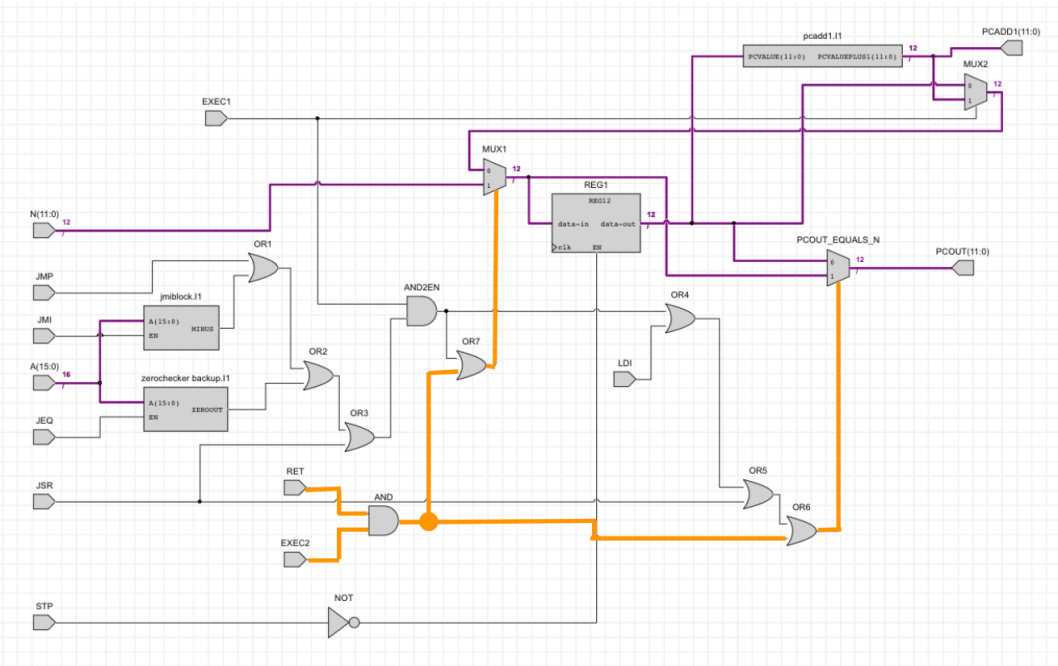
Box 1:

- During EXEC2, the IR Block does not take in Ram.Out and instead has the instruction word from EXEC1 feeding into it via the instruction register.
- However, N of Ram.Out in EXEC2 must be fed in without changing the 4 MSB, i.e. the instruction bits.
- The logic in Box 1 replaces the 4 MSB of the EXEC2 Ram.Out word and replaces them with 1100 – the instruction bits corresponding to RET. Hence, RET will stay executed during EXEC2.

Box 2:

- Box 2 feeds the 12 MSB of the EXEC1 instruction word into a Bus Compare whose output goes high if Instruction.Register.Out's instruction word is that of a RET instruction.
- The Bus's output feeds into an AND gate with EXEC2. Furthermore, the AND gate's output feeds into the select for MUX1 and MUX2. Thus, when executing a RET instruction and one is in EXEC2, MUX1's 4 MSB will correspond to the RET instruction and its 12 LSB will be equal to N of EXEC2's Ram.Out.
- **Note:** A more straightforward solution would have been to separate the 12 LSB from EXEC2's Ram.Out and feed that into the PC Block. However, both methods work.

Adjustments to the PC Block:



PC Block with RET-relevant wires highlighted in orange.

During EXEC1 RET activates nothing in the PC Block. However, during EXEC2 N becomes equal to the 12 LSB of EXEC2's Ram.Out.

- When RET is executed and one is in EXEC2, the AND gate's output will be high, making the output of MUX1 equal to N and the output of the MUX "PCOUT_EQUALS_N" equal to N, too.
- Thus, on EXEC2, the address feeding into RAM will be equal to N, such that the next instruction to be executed in EXEC1 will be that stored in address N. Furthermore, since N is also feeding into PC.Register.In, EXEC1 will have N+1 to work with.

Tidying-up the CPU:

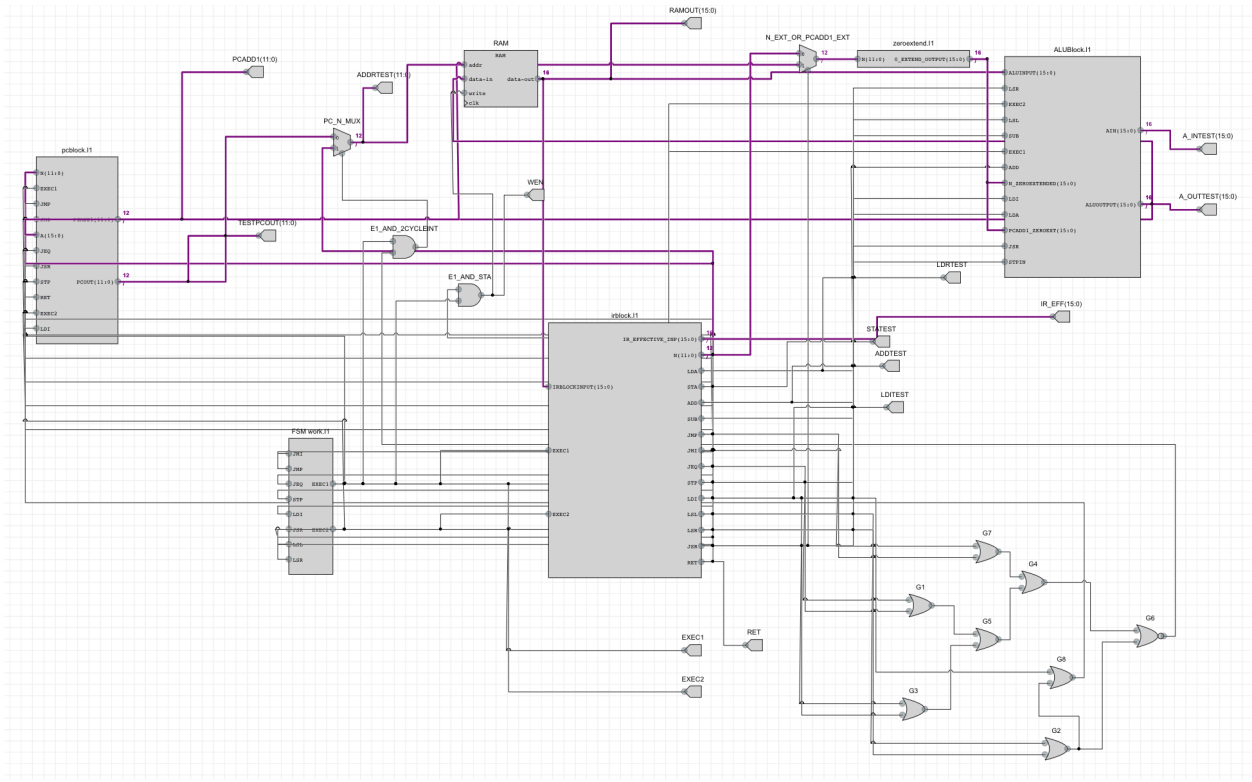


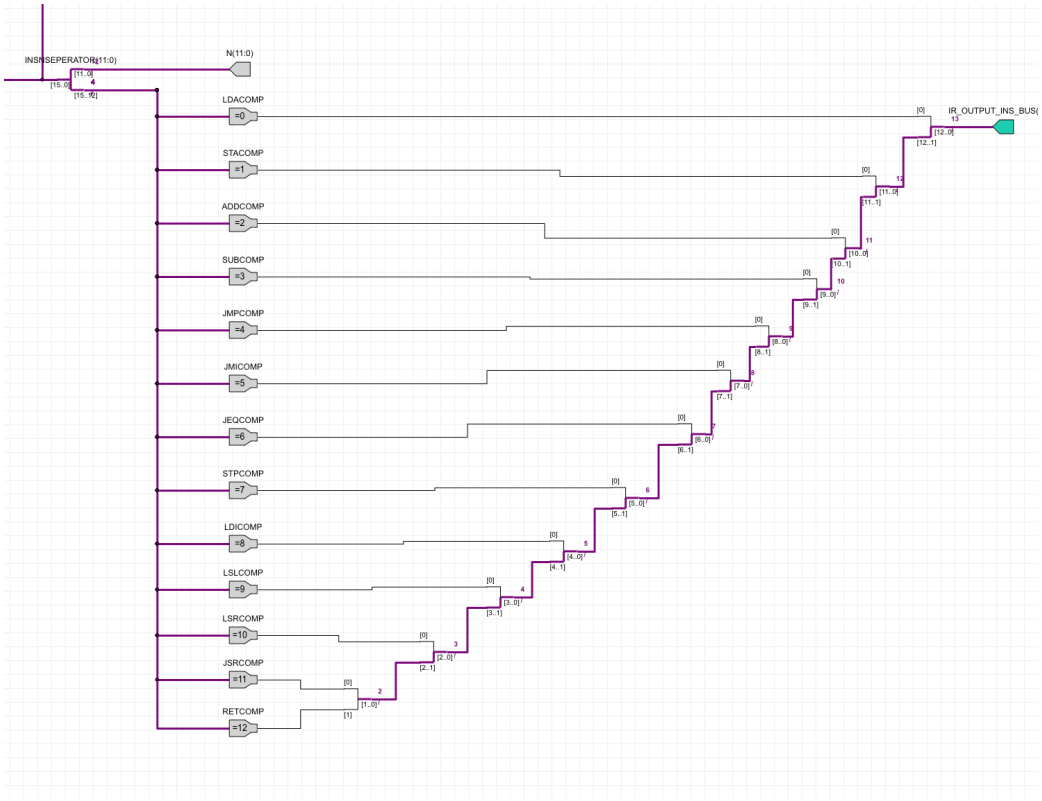
Image of compiled CPU.

Although the CPU that has been assembled may work, it looks quite messy which makes it difficult to take apart and at times can lead to confusion. Hence, I decided to make it look neater. The main untidiness was due to:

- The numerous instruction wires feeding into various blocks.
- Supplementary logic that could go into other blocks, e.g. the MUXs and gates in the above CPU image.

To address these issues, I decided to integrate the stray logic on the CPU page into a variety of blocks. Furthermore, the instruction wires were condensed into a single bus wire that feeds into the other blocks. In each relevant block, these wires are separated out by a block called the “Instruction Separator”, which essentially splits the instruction bus wire into its individual bits.

Condensing the Instruction Wires:

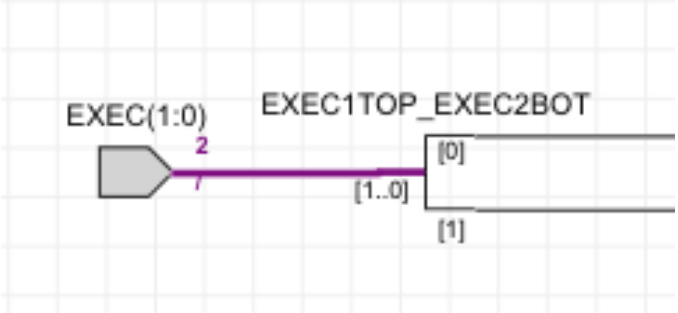


Merge wires merging the individual instructions in a single 13 bit bus.

The above merging of wires was implemented within the IR Block. No two instructions can ever be executed at the same time. Hence, only one or none of the wires will be active during any given cycle. Thus, if the instructions are executed consecutively, the hex code for the 13 bit bus wire will look as follows: 0000, 0001, 0002, 0004, 0008 etc., going up in multiples of 2.

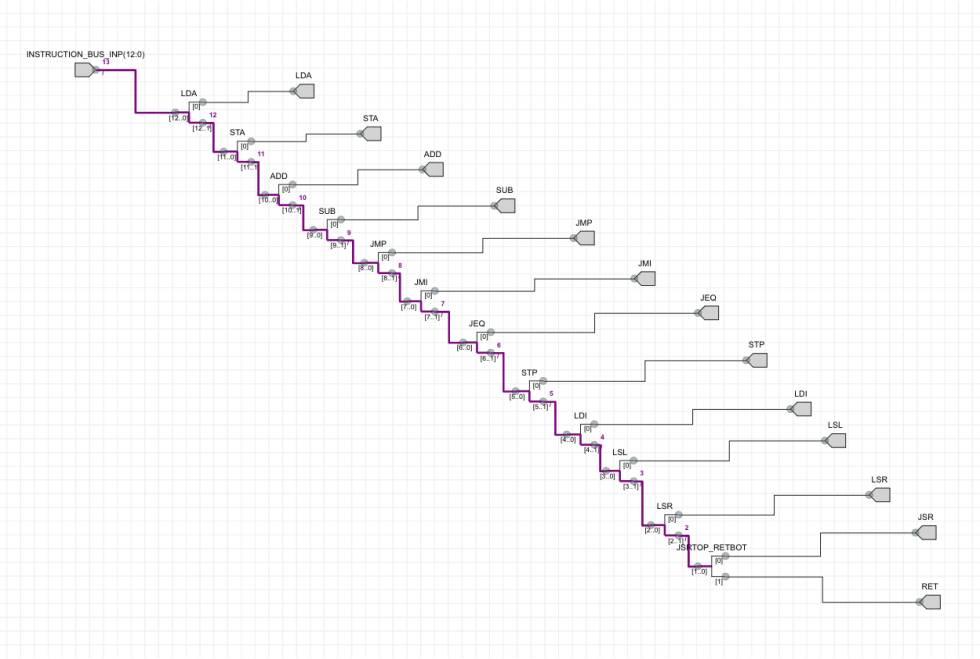
Similarly, EXEC1 and EXEC2 were placed into a single 2 bit bus. This was fed into the blocks that depend on the state. Within these blocks, the EXEC cycles were separated out into EXEC1 and EXEC2 and fed into the required inputs within the blocks.

The 2 bit EXEC bus output that replaced the individual EXEC1/2 outputs.



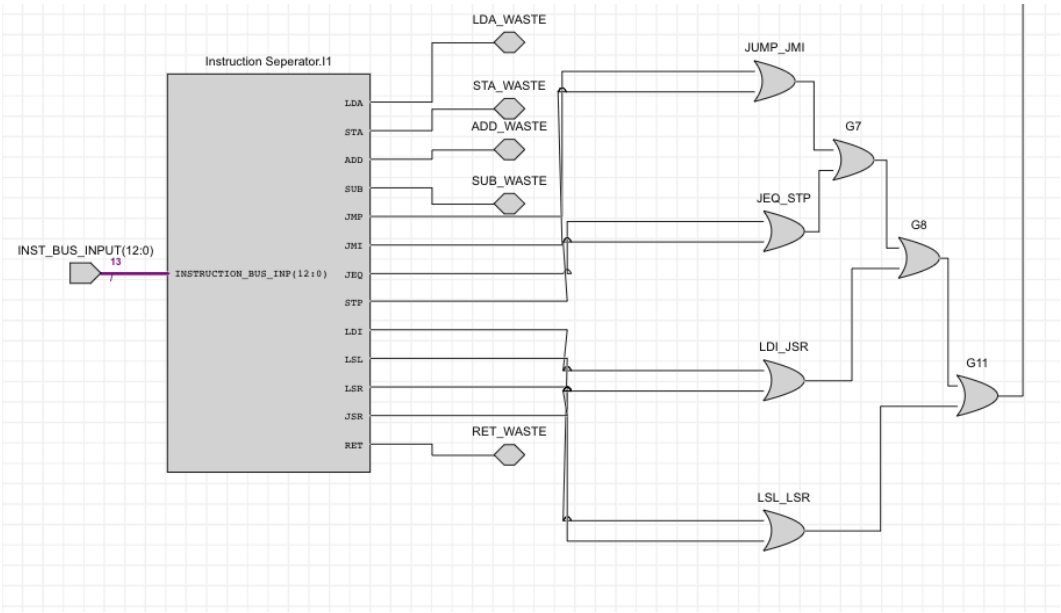
The EXEC bus being separated out into its EXEC1/2 bits in state-dependent blocks.

The Instruction Separator:



The instruction separator.

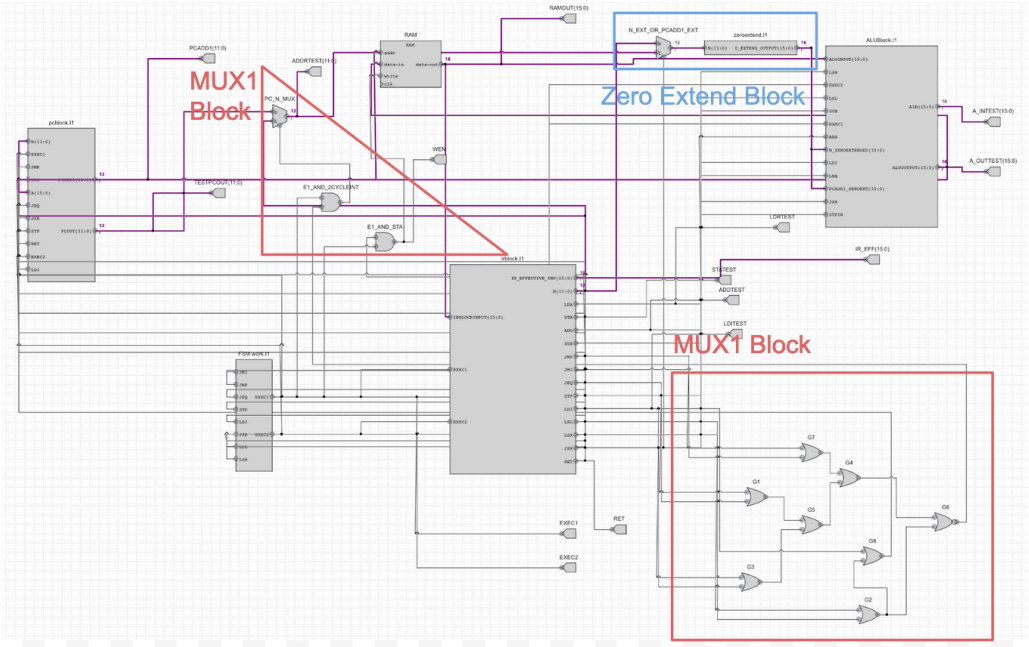
The instruction separator will have the 13 bit instruction bus as its input and separate the bus into its respective instructions from the least to the most significant bit. This block will be placed in the other blocks, as seen for the Finite State Machine below.



Demonstration of the implementation of the Instruction Separator in the FSM.

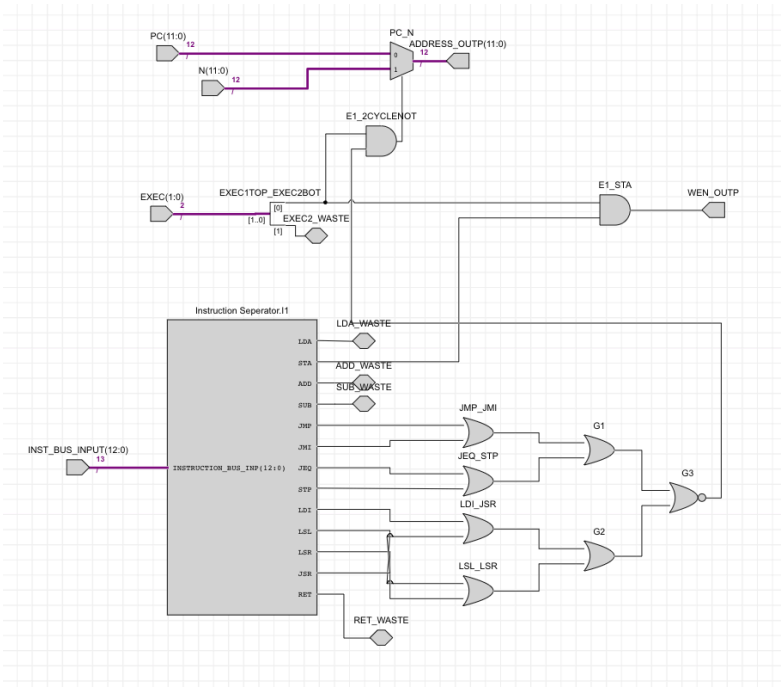
From the above image we can see that the individual inputs that we originally had have been replaced by the Instruction Separator. The inputs that are not being used will feed into a wire label, which leads nowhere.

Integrating Stray Logic into Blocks:



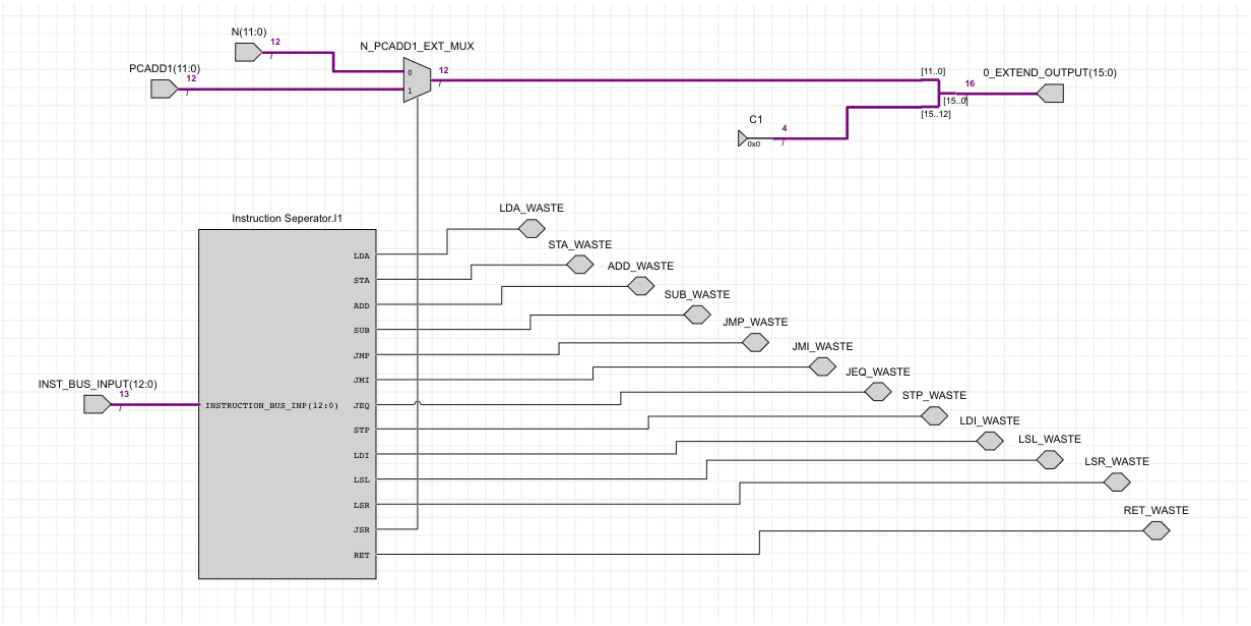
Logic in compiled CPU that will be made into blocks of their own.

MUX1 Block:



MUX1 Block.

Zero Extend Block:



Zero Extend Block.

Testing the CPU:

Addr	Assembly	Notes
0x000	LDI 0x123	check Acc = 0x0123 after LDI
0x001	LDA 0x000	check Acc = 0x8123 after LDA
0x002	STA 0x102	use STA, can't check yet
0x003	LSR	check Acc = 0x4091 after LSR
0x004	ADD 0x002	check Acc = 0x4091+0x1102 = 0x5193 after ADD
0x005	SUB 0x002	check Acc = 0x4091 after SUB
0x006	LDA 0x102	check Acc = 0x8123 after LDA - this checks previous STA
0x007	LSL	check Acc = 0x0246 after LSL
0x008	LDI 0x000	
0x009	JEQ 0x00b	
0x00a	JMP 0x012	executed if JEQ does not work
0x00b	JMI 0x012	executed if JMI does not work
0x00c	LDI 0x001	
0x00d	JEQ 0x012	executed if JEQ does not work
0x00e	LDA 0x013	load 0x8000
0x00f	JMI 0x011	
0x010	JMP 0x012	executed if JMI does not work
0x011	STP	0x11: Stop here if Jump tests PASSED
0x012	STP	0x12 Stop here if Jump tests FAILED
0x013	0x8000	constant to test JMI

Figure 6: Initial RAM contents for program: all other locations are 0

The above figure displays the code that is to be tested.

- Out of the 20 addresses above, only 16 of them should execute as some of the addresses are never executed.
- In the non-optimised CPU each instruction takes 3 cycles to complete. Hence, the program would require around 48 cycles.
- There are around ten 2 cycle instructions and six 3 cycle instructions. Since the cycles of some instructions overlap, i.e. the next instruction's first cycle overlaps with the last cycle of the previous instruction, 2 cycle instructions effectively

require 1 cycle, while 3 cycle instructions effectively require 2 cycles. Thus, the execution of the program should take around 22 cycles.

Testing the CPU			
Address	Instruction	Accumulator Value (A.Out)/Notes	Cycle of A.Out Change
0x000	LDI	0123	2
0x001	LDA	8123	4
0x002	STA	8123	-
0x003	LSR	4091	7
0x004	ADD	5193	9
0x005	SUB	4091	11
0x006	LDA	8123	13
0x007	LSL	0246	14
0x008	LDI	0000	15
0x009	JEQ	Addr := 00B	15
0x00A	JMP	Skipped	-
0x00B	JMI	JMI not exec.	-
0x00C	LDI	0001	19
0x00D	JEQ	JEQ not exec.	-
0x00E	LDA	8000	21
0x00F	JMI	Addr := 011	21
0x010	JMP	Skipped	-
0x011	STP	8000	22
0x012	STP	Never exec.	-
0x013	0x8000	Never exec.	-

The program required 22 cycles to execute, as expected.

Testing JSR:

Address	Content
0x000	0x8321
0x001	0xB006
0x002	0x8EEE
0x003	0x8EEE
0x004	0x8EEE
0x005	0x8EEE
0x006	0x2009
0x007	0x2009
0x008	0x7000
0x009	0x0001

Program used to test JSR.

If the program works, it will:

- Load 0321 onto the accumulator (A.Out) during cycle 2.
- Set A.In to 0002 during cycle 2 and A.Out to 0002 during cycle 3.
- Feed 006 into address during cycle 2, such that 007 is feeding into address during cycle 3.
- A.Out := 0003 on cycle 5 and 0004 on cycle 7.
- Program stops with A.Out equal to 0004.
- If JSR does not work, 0EEE will become the accumulator's output.

Testing JSR			
Address	Instruction	Accumulator Value (A.Out)/Notes	Cycle of A.Out Change
0x000	LDI	0321	2
0x001	JSR	0002	3
0x002	LDI	Skipped	-
0x003	LDI	Skipped	-
0x004	LDI	Skipped	-
0x005	LDI	Skipped	-
0x006	ADD	0003	5
0x007	ADD	0004	7
0x008	STP	0004	8
0x009	0x0001	-	-

Testing RET:

Address	Content
0x000	0x4006
0x001	0x8EEE
0x002	0x3009
0x003	0x7000
0x004	0x8EEE
0x005	0x8EEE
0x006	0x8001
0x007	0xC008
0x008	0x0002
0x009	0x0001

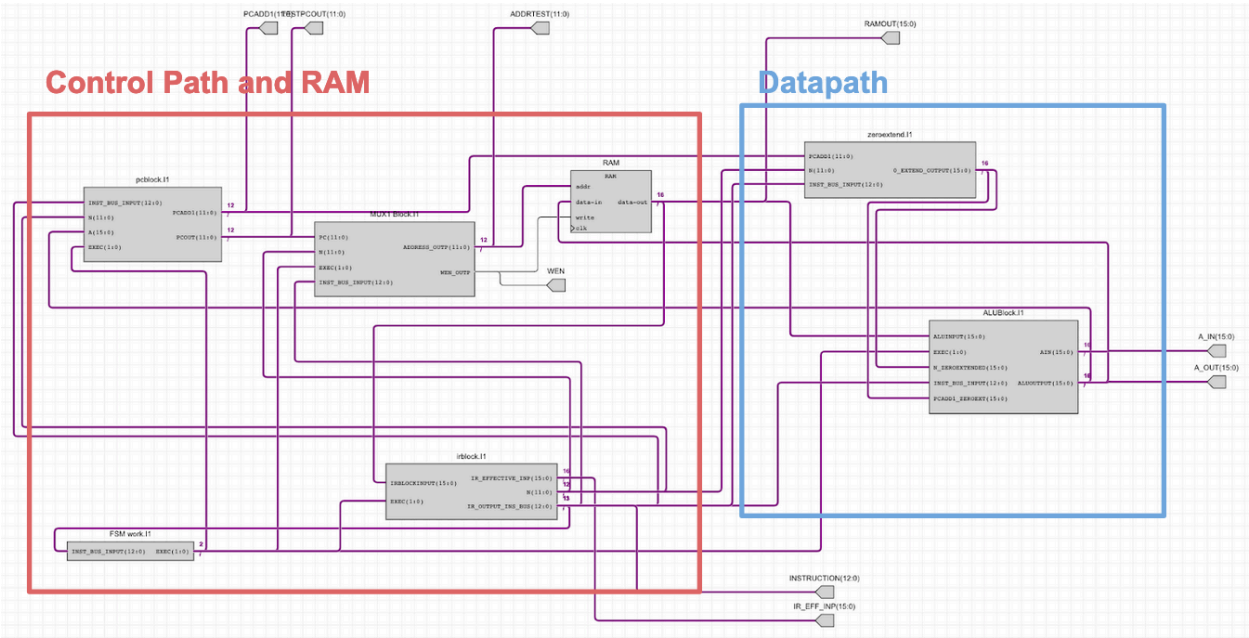
Program used to test RET.

If the program works, it will:

- Jump to address 006 on cycle 2.
- Set A.Out to 0001 on cycle 3.
- Jump to address 002 on cycle 5.
- Set A.Out to 0000 on cycle 7.
- Stop on cycle 8 with A.Out set to 0000.

Testing RET			
Address	Instruction	Accumulator Value (A.Out)/Notes	Cycle of A.Out Change
0x000	JMP	Addr := 006	1
0x001	LDI	Skipped	-
0x002	SUB	0000	7
0x003	STP	0000	8
0x004	LDI	Skipped	-
0x005	LDI	Skipped	-
0x006	LDI	0001	3
0x007	RET	Addr := 002	4
0x008	0x0002	-	-
0x009	0x0001	-	-

Optimised Compiled CPU:



Optimised, neat CPU.

Debugging Log:

Bug 1:

- Issue: The STA instruction would change the value of the accumulator.
- Solution: Feed the output of the accumulator into its input during instructions such as STA.

Bug 2:

- Issue: For an ADD or SUB instruction the ALU input was fed into the respective adder blocks. After the adding cycle, the A input would feed into the adder blocks, causing unwanted addition.
- Solution: Add MUX in such that during the addition cycle the number to be added is fed into the adder and in any other cycle the number feeding into the adder is zero.

Bug 3:

- Issue: Accumulator's output would change to an unwanted number some a couple of cycles after an LDI instruction.
- Solution: Feed accumulator into self when accumulator value should not change.

Bug 4:

- Issue: STP was not stopping the counter.
- Solution: 000 was being fed into the address during the STP instruction, causing a reset of the program. This was fixed by feeding PC.Register.Out into address during the STP instruction, i.e. the address of the STP instruction.

Bug 5:

- Issue: ALU arithmetic was being executed in cycles other than EXEC2, causing unwanted changes in the accumulator.
- Solution: Set up AND gates which ANDed EXEC2 and the instruction, such that any arithmetic for a given instruction would only occur during EXEC2.

Bug 6:

- Issue: LSL and LSR were misinterpreted as 3 cycle instructions, as I thought they shift the value of a number stored in a given address.
- Solution: Made them 2 cycle instructions that shift the number stored in the accumulator.

Bug 7:

- Issue: For RET, the N bits could not feed into the PC Block during EXEC2 due to the IR Block not accepting inputs during EXEC2.
- Solution: Use MUXs to allow the IR Block to have Ram.Out feed into it during EXEC2 of a RET instruction.

Bug 8:

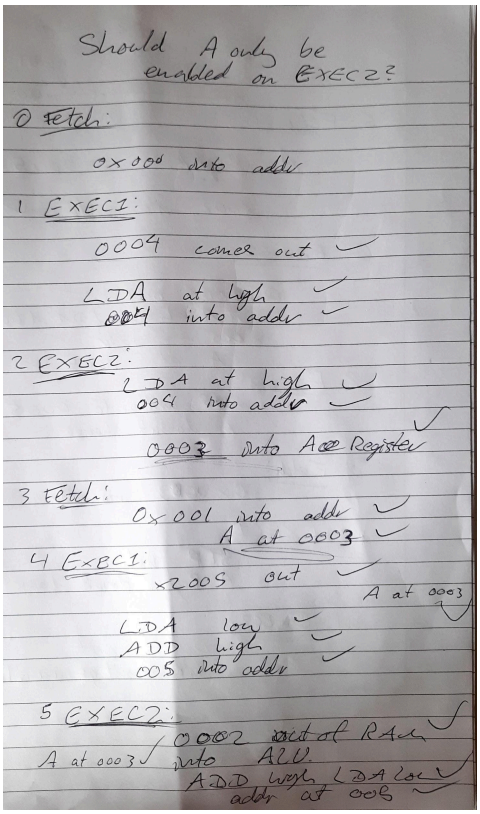
- Issue: SUB values were off by +1 due to an extra carry-in.
- Solution: Removed the extra carry-in.

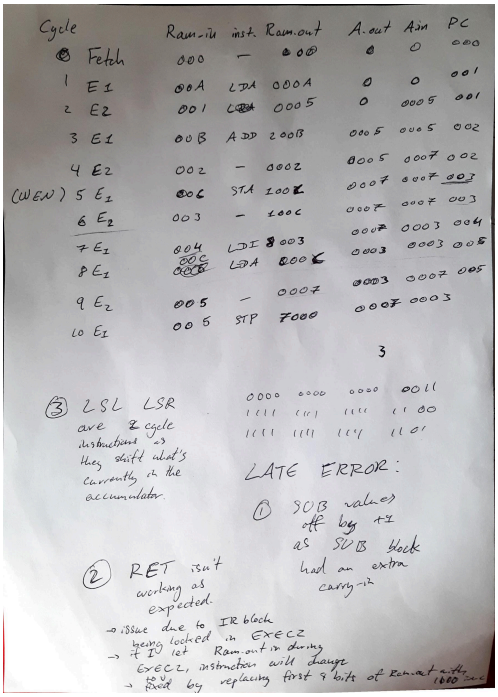
Bug 9:

- Issue: PC Counter would not function after an STA instruction.
- Solution: This issue occurred due to STA being treated as a 2 cycle instruction, which allowed for no cycle for the address of the next instruction to be fed into RAM. The solution was to treat STA as a 3 cycle instruction.

Appendix:

Appendix 1:

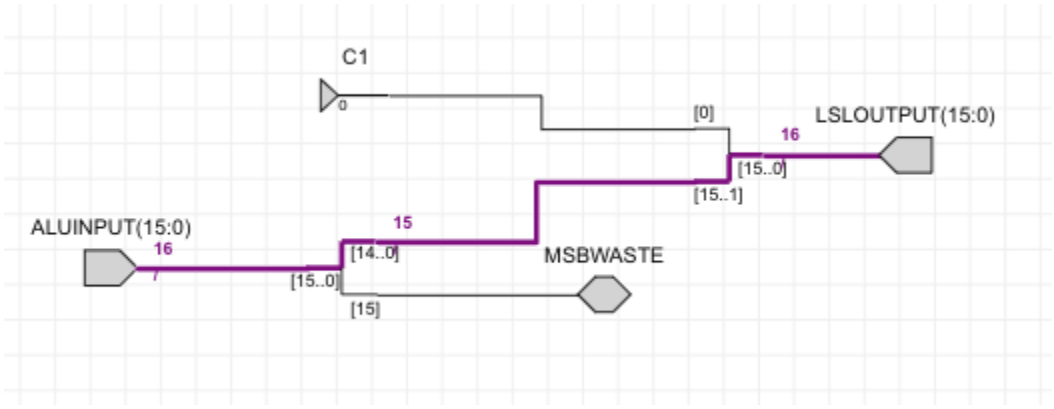




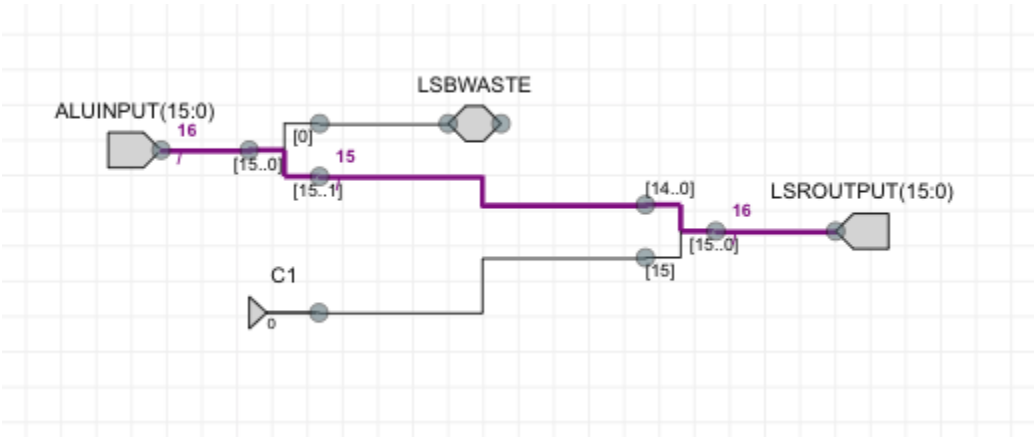
	A	B	C	D	E	F	G	H	I	J	K	L
1	Cycle:	State:		RAM.IN	RAM.OUT	A.IN	A.OUT	LDA	STA	ADD	LDI	Machine Code:
2	**0**	**Fetch:**		000	0000	0000	0000	0	0	0	0	8002
3	1	EXEC1:		002	8002	0002	0000	0	0	0	1	1005
4	2	EXEC2:		002	0005	0002	0002	0	0	0	1	0005
5	**3**	**Fetch:**		001	0005	0002	0002	1	0	0	0	D000
6	4	EXEC1:		005	1005	0002	0002	0	1	0	0	
7	5	EXEC2:		005	0002	0002	0002	0	1	0	0	
8	**6**	**Fetch:**		002	0002	0002	0002	1	0	0	0	
9	7	EXEC1:		005	0005	0002	0002	1	0	0	0	
10	8	EXEC2:		005	0002	0002	0002	1	0	0	0	
11	**9**	**Fetch:**		003	0002	0002	0002	1	0	0	0	
12	10	EXEC1:		000	D000	0002	0002	0	0	0	0	
13	11	EXEC2:		000	8002	0002	0002	0	0	0	0	
14	**12**	**Fetch:**										
15	13	EXEC1:										

I debugged most of my code by writing test programs. For each test program I wrote out exactly what value certain inputs and outputs should have during each cycle of the program. Thus, if there was ever a bug, I could ascertain the issue relatively fast.

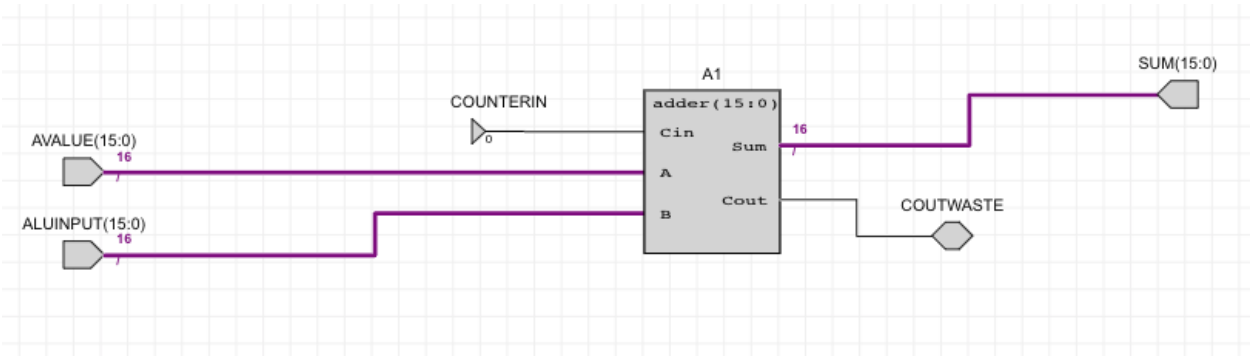
Appendix 2:



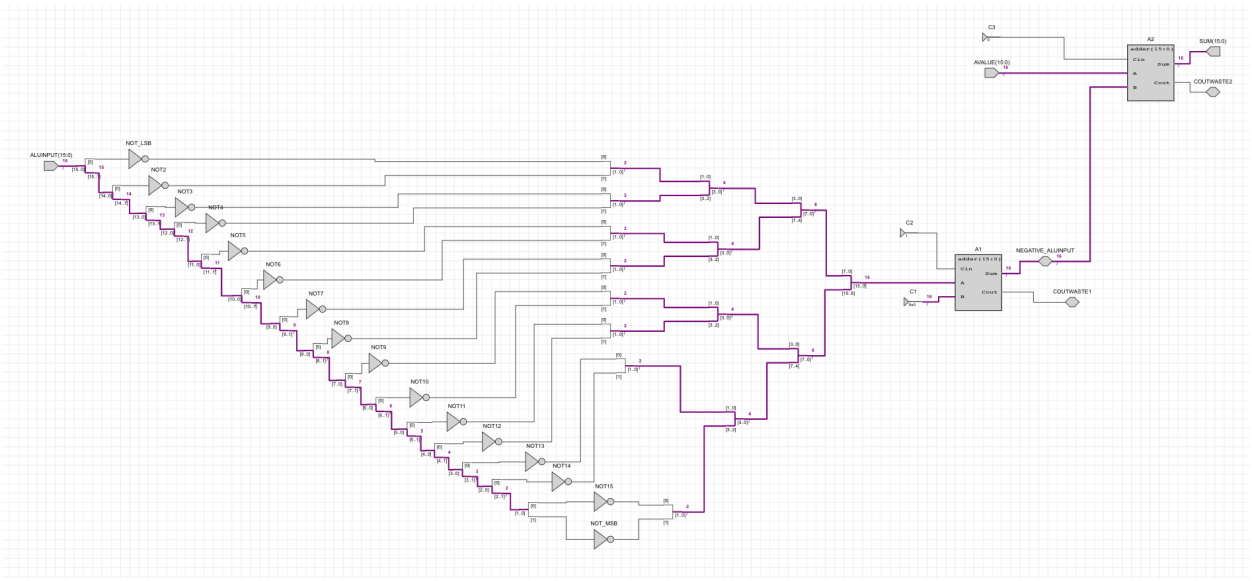
Logic for LSL Block.



Logic for LSR Block.



Logic for ADD Block.



Logic for SUB Block.