Kiril Tofiloski
[Ulica, mesto, poštna številka]
Tel. [Telefon]
Faks [Faks]
[Spletno mesto]

MIMO V2

2023

MiMo v2: Advanced, pipelined version of MiMo v1 with forwarding and predictions

ChangeLog:

- v01: starting text by Kiril Tofiloski
- v02: v01 enhanced by Robi
- v03: new version from Kiril (slightly corrected by Robi)

CONTENT

Content

MiMo pipeline transformation

This document describes the changes to the new versions of the pipelined CPU based on the original MiMo CPU model, developed in Logisim Evolution.

MiMo model used so far is now denoted as »mimo_v1«, and new versions as »mimo_32bit_v2[.1,2,3]«.

CPU OVERVIEW

This document describes the changes to the new versions of the pipelined CPU based on the original MiMo CPU model (v1), developed in Logisim Evolution.

General features and changes:

- 5 stage pipeline (Instruction fetch - **IF**, Instruction Decode - **ID**, Execute - **EX**, Memory Access - **MA**, Write Back - **WB**)
- Harvard Architecture (Separate Instruction and Operand RAM)
- 32-bit Instruction Set
- 16-bit addresses (*Logisim RAM modules have limited address sizes so I left it at 16-bits for now).*
- Instruction set has been modified to be more similar to ARM ISA (*More details on **Instructions** page).*
- Added Overflow flag **V**
- Added Link Register
- **mimo_32bit_v2**: no pipeline hazard optimization methods added yet. **Nop** commands need to be added manually to instruction scripts to avoid hazards.
- **mimo_32bit_v2.x**: Multiple pipeline hazard optimization methods added (*Described thoroughly on Pipeline Hazard Optimizations page*).

MIMO PIPELINE TRANSFORMATION

INSTRUCTION SET ARCHITECTURE TRANSFORMATION

I made the instruction set and CPU in general function much more like a classic ARM CPU, while also combining it with some of the features of the previous MiMo model that made it simpler for learning purposes.

Added optional condition bits to every instruction

In the previous version of MiMo, each conditional operation would have to be made as a separate instruction in the micro-assembler.

In this version each instruction has 4 bits reserved for a condition code (*e.g. mov**eq** r1, #5 - Equal condition)*.

The condition is checked in the **ID** stage in the **check_condition** circuit. All conditions are based on the ARM ISA conditional codes. They go as follows:

Mnemonic	Description	Flags state
EQ	Equal	Z set
NE	Not equal	Z clear
CS	Carry set/unsigned higher or same	C set
CC	Carry clear/unsigned lower	C clear
MI	Minus/Negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N = V
LT	Signed less than	N != V

Mnemonic	Description	Flags state
GT	Signed greater than	Z clear and N = V
LE	Signed less than or equal	Z set and N != V
AL	Always/unconditional	Irrelevant

Added 'Set Flags' (S) bit

In the previous version (v1) of MiMo, all instructions would change the ALU flags, now each instruction has an extra bit reserved for the 'set flag' bit, and only changes the state of the flags if it is set.

This was another change made to bring MiMo closer to a traditional ARM processor.

Instruction example:

*add**s** r1, r2, r3*

Added immediate as part of instruction

In the previous version of MiMo, all immediates would be contained in a separate instruction following the original instruction (Format 2 in v1).

In a pipelined CPU, this would slow things down tremendously. Since we are also increasing the instruction size to 32 bits, I decided to change it to work as a classic ARM CPU would, where the immediate value would occupy a part of the original instruction, 10 bits in our case.

The downside of this approach is of course that we are more limited in the size of our immediate values

In a classic ARM Instruction set, there are also no separate instructions for immediate operands.

I decided to implement this in MiMo by having the **imload** signal, which was previously a control signal provided by the micro-assembler, be part of the machine instruction instead.

We reserve 1 bit in the machine instruction marked as the **imload** bit. When this is active, the CPU knows that the instruction uses the immediate operand provided within it. This way we do not need separate instructions for using immediate values.

Instruction format

Instruction includes several bit groups :

- Operation Code - 7 bits
- Condition Code - 4 bits
- imload - 1 bit
- Set flags - 1 bit
- Rd - 3 bits
- Rs - 3 bits
- Rt - 3 bits
- Immediate value - 10 bits

Instruction Syntax

Changed instruction syntax to a simplified ARM Assembly syntax.

Symbol	Function	Example
/* */	Multiline comment	/*not read by assembler*/
@	Single-line comment	@also not read
#	Indicates immediate value	mov r1, #1
.text	Indicates start of instruction section	add r1,r2,r3 @this instruction @would not be read .text add r1,r2,r3 @this one would
.data	Indicates start of data section	.word 0xff, 0x23 @this data would not @be read .data .asciiz "Hello World" @this data would

Combinations in the form of [r1, r2] have NOT been added for any instructions except ldr (More on this at the end of the next section).

The .data section outputs a second ram file for the operand RAM, when used.

You first must declare .data then followed by the data you want to add. Afterwards, before declaring instructions, you must declare .text similar to standard ARM assembly.

```
.data

.word 1,2, 3 , 4
.word 0x1d, 0xff,0x23

.space 12

.ascii "Hello World", "aaa","bbb"
.asciiz "testing", "ccc"
```

Example .data section of code

The **.word** directive declares words to be added to the operand RAM. They can be written as decimal or hexadecimal type.

The **.space** directive declares an amount of space(bytes) to be left empty in operand RAM.

The **.ascii** and **.asciiz** directives declare ascii strings to be added to the operand RAM. They are not aligned and take up as much space as needed. The only difference is .asciiz adds a terminating null character(empty byte) to each string.

Examples of current proper syntax can be seen in the **Assembler/tests** folder.

Instruction Set SUMMARY

For the instruction set I implemented most of the standard ARM instructions along with some additions from the previous MiMo model.

Instruction Code	Arguments	Function
mov	Rd, Rs/Immediate	Rd <- Rs/Immediate
mvn	Rd, Rs/Immediate	Rd <- NOT Rs/Immediate
add	Rd, Rs, Rt/Immediate	Rd <- Rs + Rt/Immediate
sub	Rd, Rs, Rt/Immediate	Rd <- Rs - Rt/Immediate
rsb	Rd, Rs, Rt/Immediate	Rd <- Rt/Immediate - Rs
mul	Rd, Rs, Rt/Immediate	Rd <- Rs * Rt/Immediate
div	Rd, Rs, Rt/Immediate	Rd <- Rs / Rt/Immediate
rem	Rd, Rs, Rt/Immediate	Rd <- Rs % Rt/Immediate
and	Rd, Rs, Rt/Immediate	Rd <- Rs AND Rt/Immediate
orr	Rd, Rs, Rt/Immediate	Rd <- Rs OR Rt/Immediate

MIMO PIPELINE TRANSFORMATION

Instruction Code	Arguments	Function
eor	Rd, Rs, Rt/Immediate	Rd <- Rs XOR Rt/Immediate
nand	Rd, Rs, Rt/Immediate	Rd <- Rs NAND Rt/Immediate
nor	Rd, Rs, Rt/Immediate	Rd <- Rs NOR Rt/Immediate
bic	Rd, Rs, Rt/Immediate	Rd <- Rs AND NOT(Rt/Immediate)
cmp	Rs, Rt/Immediate	Rs - Rt Set flags
cmn	Rs, Rt/Immediate	Rs + Rt Set flags
tst	Rs, Rt/Immediate	Rs XOR Rt Set flags
teq	Rs, Rt/Immediate	Rs AND Rt Set flags
lsl	Rd, Rs, Rt/Immediate	Rd <- Rs << Rt/Immediate
lsr	Rd, Rs, Rt/Immediate	Rd <- Rs >> Rt/Immediate
asr	Rd, Rs, Rt/Immediate	Rd <- Rs >> Rt/Immediate, Filled bits are sign bit
ror	Rd, Rs, Rt/Immediate	Rd <- Rs ROLL RIGHT Rt/Immediate
rol	Rd, Rs, Rt/Immediate	Rd <- Rs ROLL LEFT Rt/Immediate

Instruction Code	Arguments	Function
j	Rs/Immediate/Label	PC <- Rs/Immediate/Label
b	Immediate/Label	PC <- PC + Immediate/Label
bl	Label?	Jump to subroutine
ldr	Rd, Immediate	Rd <- M[Immediate]
str	Rd, Immediate	M[Immediate] <- Rd
nop	None	No operation (Used to skip cycle and avoid pipeline hazards)

The **ldr** instructions supports 3 different notations that correspond to different addressing modes taht are supported in the CPU:

- **ldr Rd, [Rs]** - Register indirect addressing
- **ldr Rd, [Rs, Rt/immed]** - Base plus offset/Base plus index addressing
- **ldr Rd, immed** - Absolute addressing

Added Link Register and Subroutine Calls

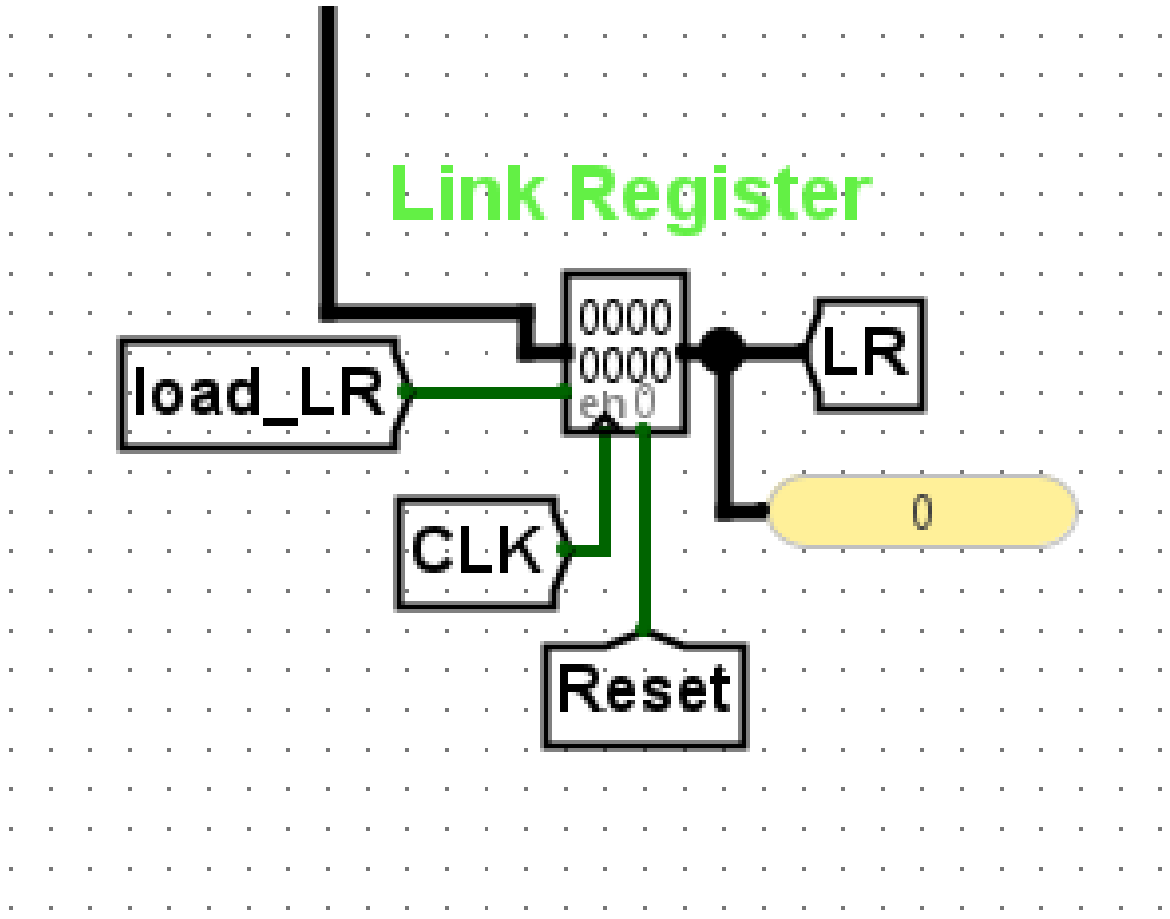
I added the Link register as a special register, outside of the Register Bank. This register is used to store the address for returning from subroutine calls with the **bl** (branch with link) instruction.

Sometimes a branch is executed to call a subprogram where the instruction sequence should return to the calling sequence when the subprogram terminates.

These are called **subroutine calls** and are sufficiently common that most architectures include specific instructions to make them efficient.

In our case, we use the **bl** and **rts** instructions.

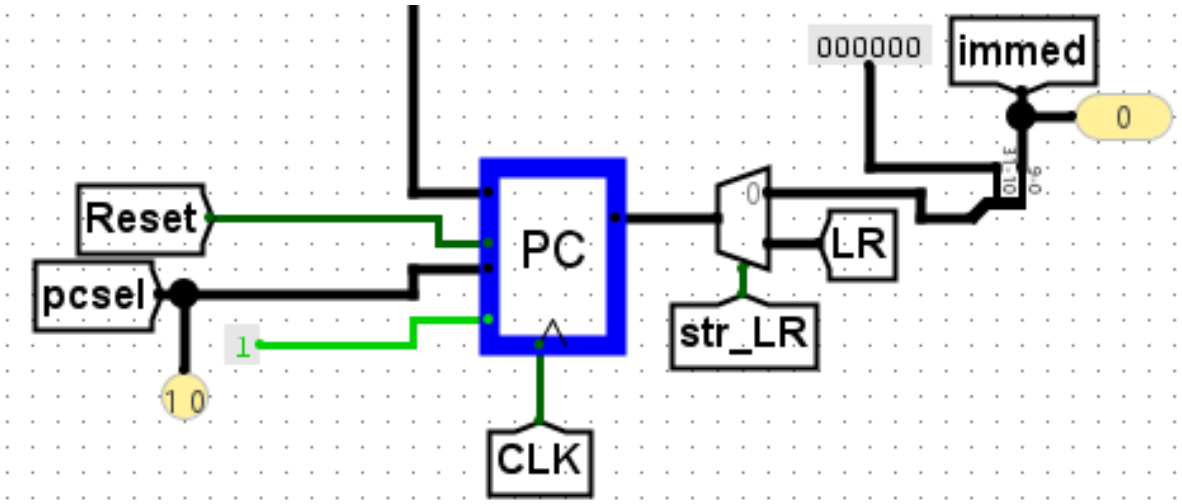
It does not make sense to use one of the normal registers in our pipelined CPU because the Write Back stage for normal registers happens at the end, and since branch with link is a branch instruction, it needs to happen immediately after decoding like the other branch/jump instructions.



Link register

I added 2 new microinstructions, **loadlink** and **strlink**.

When the first signal is active it saves the current IF stage address in the link register, the second stores the current Link register value inside the **immed** port of the IF stage, as pictured below.



Link register connected to IF stage

The **rts** instruction is used to return from a subroutine call, i.e. it stores the address that was saved in our link register back into the PC, so the CPU may continue fetching instructions from where it left off when the subroutine was entered.

An example can be seen and tested in the **Assembler/tests/test7.txt** file.

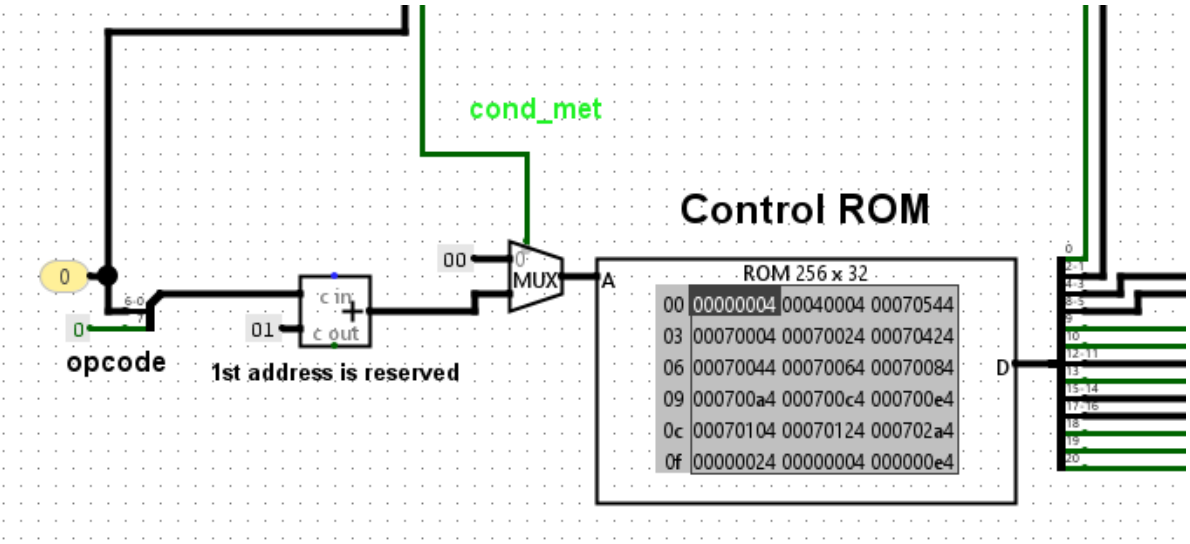
CONTROL SIGNALS

Conditional Execution and Removed Decision ROM

In the previous version of MiMo, conditional instructions would have to be specified and specifically programed for every condition in the micro-assembler.

Given that we changed it so every instruction can have an optional condition code now, programming each one separately seems unnecessary.

I found that a more elegant solution is to remove the decision ROM and have the conditional checking be done in hardware.



The Control ROM

MIMO PIPELINE TRANSFORMATION

The new conditional execution is done in the following way:

1. A **condition_met** signal along with the **opcode** is received from the **ID** stage.

2. The first address in our Control ROM is reserved for when a condition is not met. This throws the following **EX,MA,WB** and **IF** stages in their idle/default state as no instruction will be coming in.

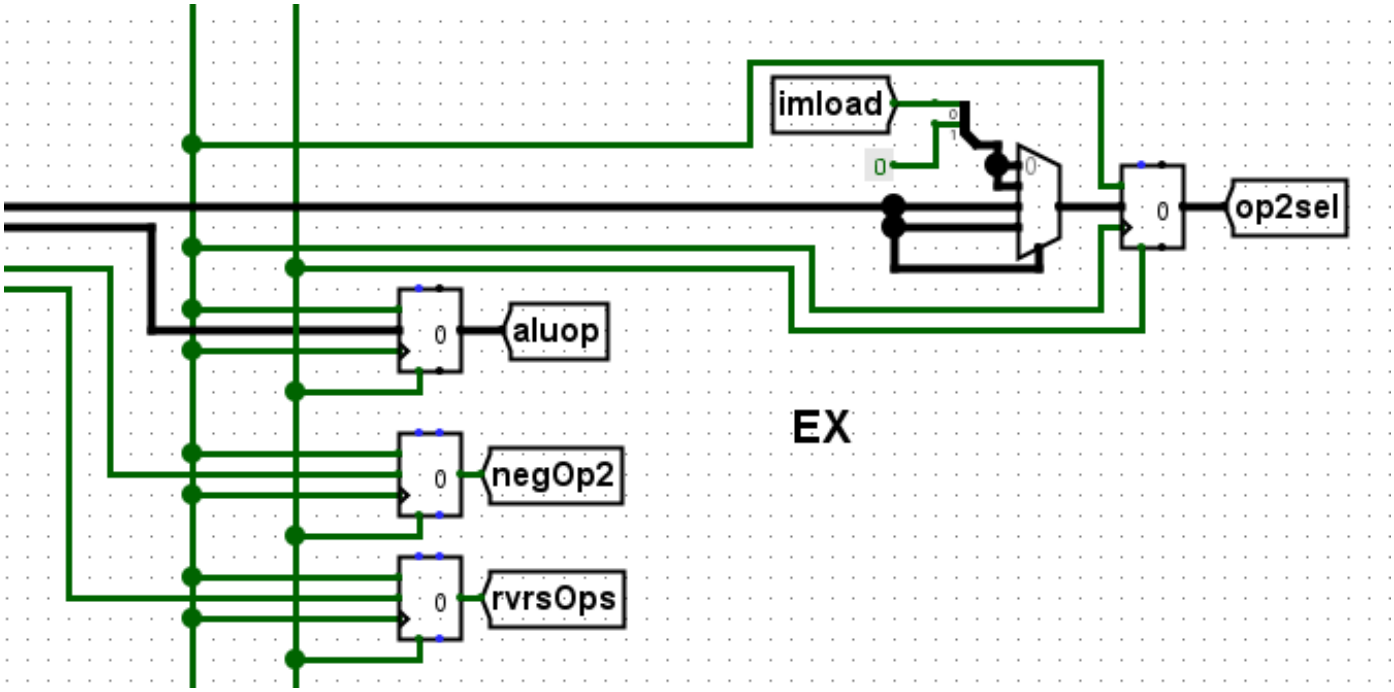
3. The **condition_met** signal tells us when a condition has been met, so we may execute the instruction. When it is active, the Control ROM takes the **opcode** + 1 (+1 because of reserved first address) as it's address, when it is inactive it takes the first idle/default address as the condition has not been met.

4. Each ROM address contains the values that each control signal should have for the current microinstruction.

Control signal buffering

Because it is a pipelined CPU of course, some signals will be needed in later stages than others.

To accommodate for this, signals are buffered using different shift registers,



Buffering of control signals

For example, signals needed for the **EX** stage will be needed right after the **ID** stage, so they are entered in a 1 stage shift register. Signals needed for the **MA** stage require a 2 stage shift register and signals needed for the **WB** stage require a 3 stage shift register.

The only exceptions are signals needed for the **IF** stage. They are immediately processed as they are always jump/branch signals and the flow of execution needs to be immediately changed.

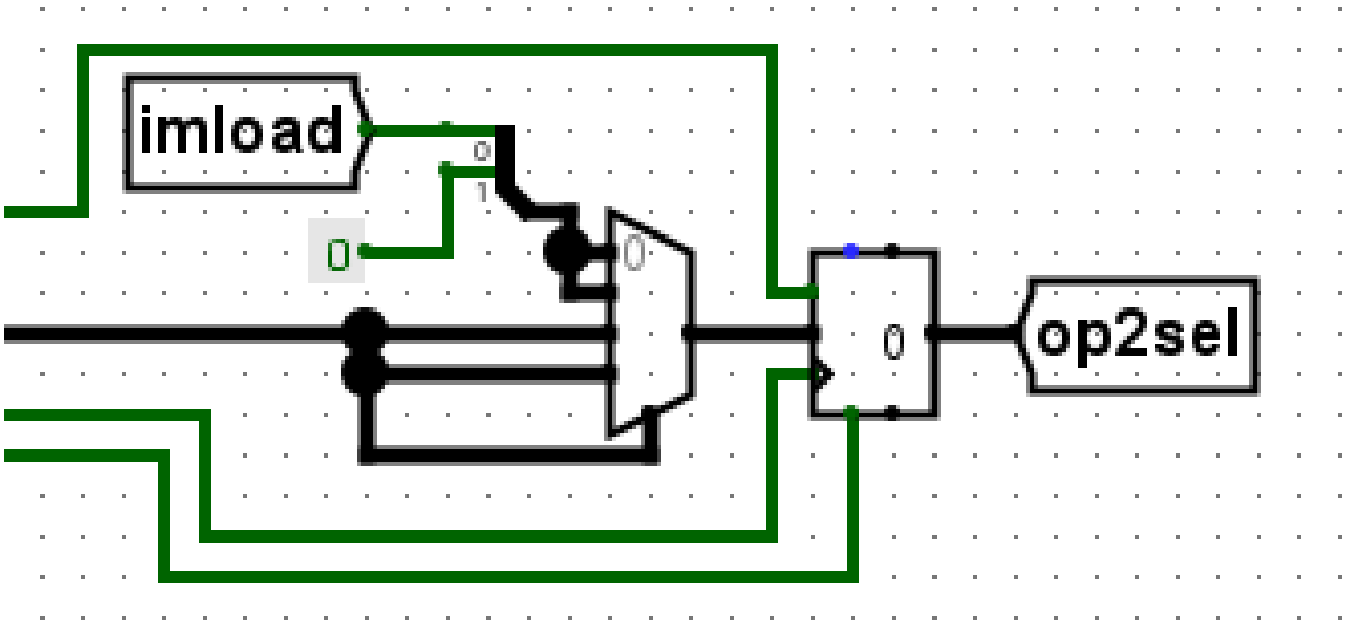
Control signals with imload

As mentioned in the **Instructions section**, some instructions can either take an immediate or a register as an argument now and we do not need separate instructions for immediates.

Since the micro-assembler does not know if a given instruction will use an immediate or a register, a change for this situation needed to be made.

For example, take the **op2sel** signal. Previously the possible values were **treg(0), immed(1), const0(2), const1(3)**.

For an instruction like **add**, that can use a third register or an immediate value, we do not know which value for **op2sel** (**treg** or **immed)** to program into the micro-assembler.

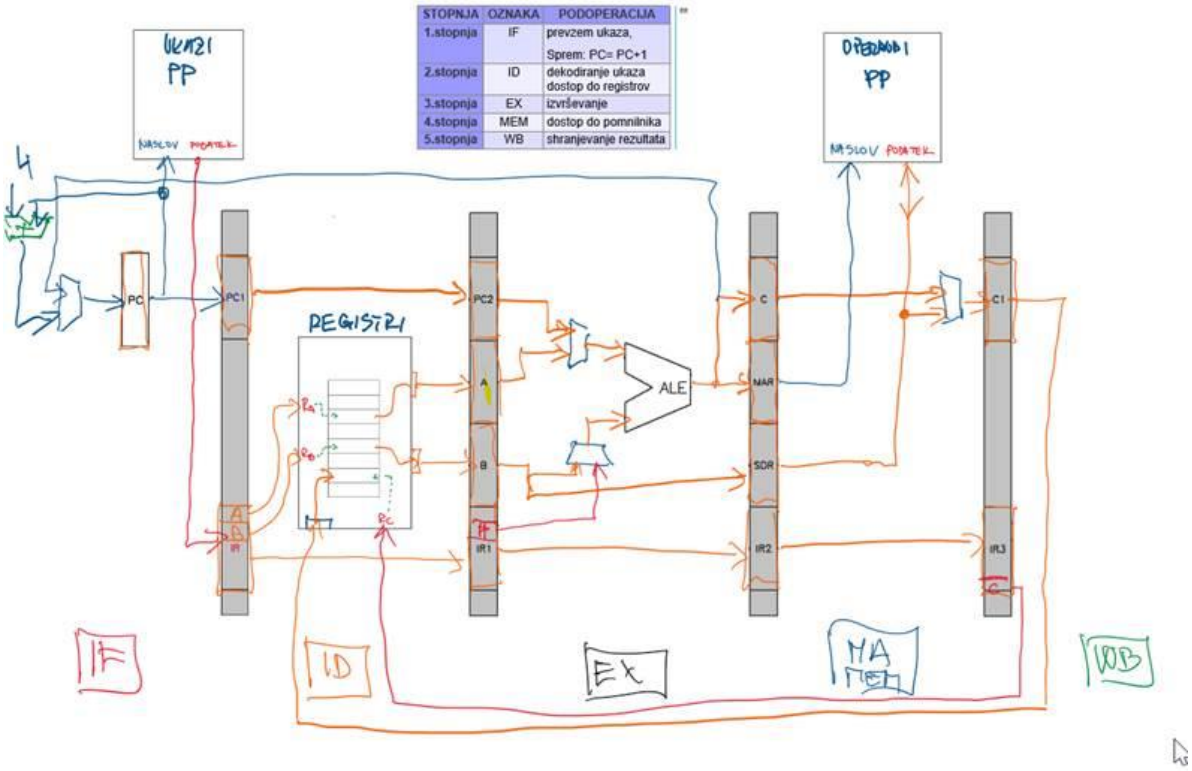


Instead of having **treg** and **immed**, we replace them with **op2(0),** and use the MUX above to properly determine the signal. It works as follows:

- The signal acts as the select signal to the MUX.
- When it is 0 or 1, if **imload** is 1, effectively **immed(1)** is loaded as the signal, when **imload** is 0, effectively **treg(0)** is loaded.
- When it is 2 or 3, **const0(2)** and **const1(3)** are simply loaded.

OVERVIEW OF PIPELINE STAGES

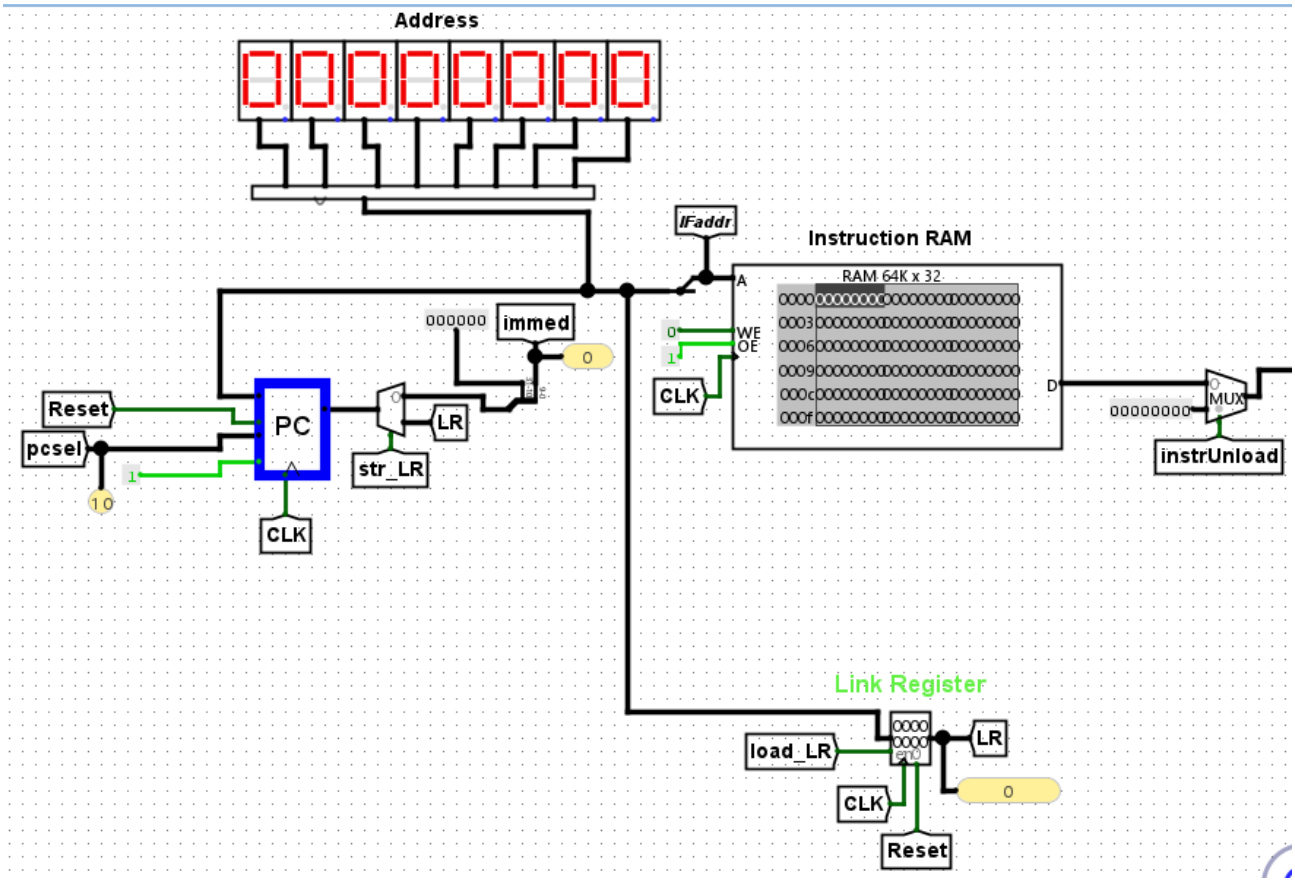
CPU Layout



OR class sketch

The general layout of the CPU and it's stages has been made to mimic the OR class sketch in it's layout. All of the stages and their parts are displayed on the main circuit with added "transition blocks" between them (like IF->ID, ID->EX), that store the data going in and leaving a stage.

Instruction Fetch - IF



IF Stage along with Instruction RAM

Inputs:

- immed
- Rs
- pcsel
- instrUnload
- str_LR
- load_LR

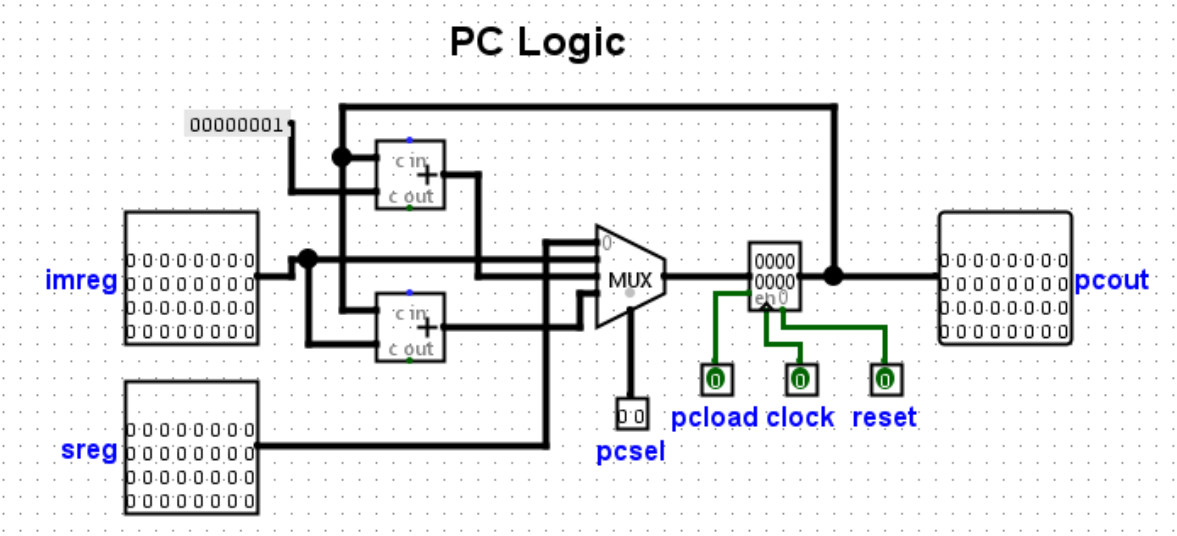
Outputs:

- instruction

The Instruction Fetch Stage has the task of fetching a new instruction each cycle.

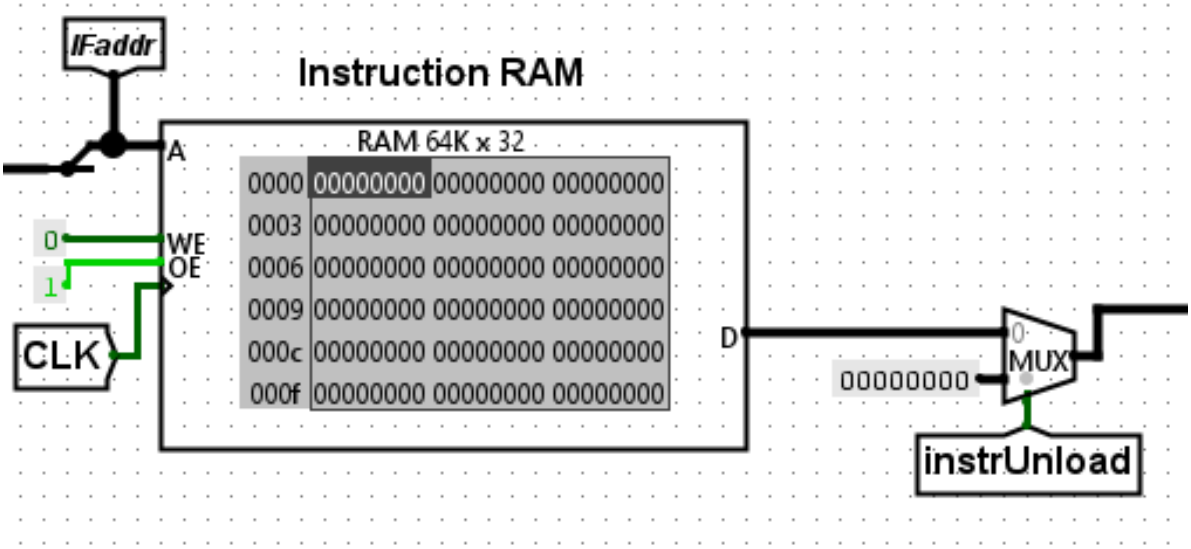
The immed and Rs values are directly taken from the most recently processed ID Stage immediate and register values.

The pcsel signal dictates whether the PC should increment, jump to immediate address(immed), jump to PC + immediate or jump to register address(Rs).



Inside of PC

After processing the correct address, the Instruction RAM gives us the corresponding instruction.

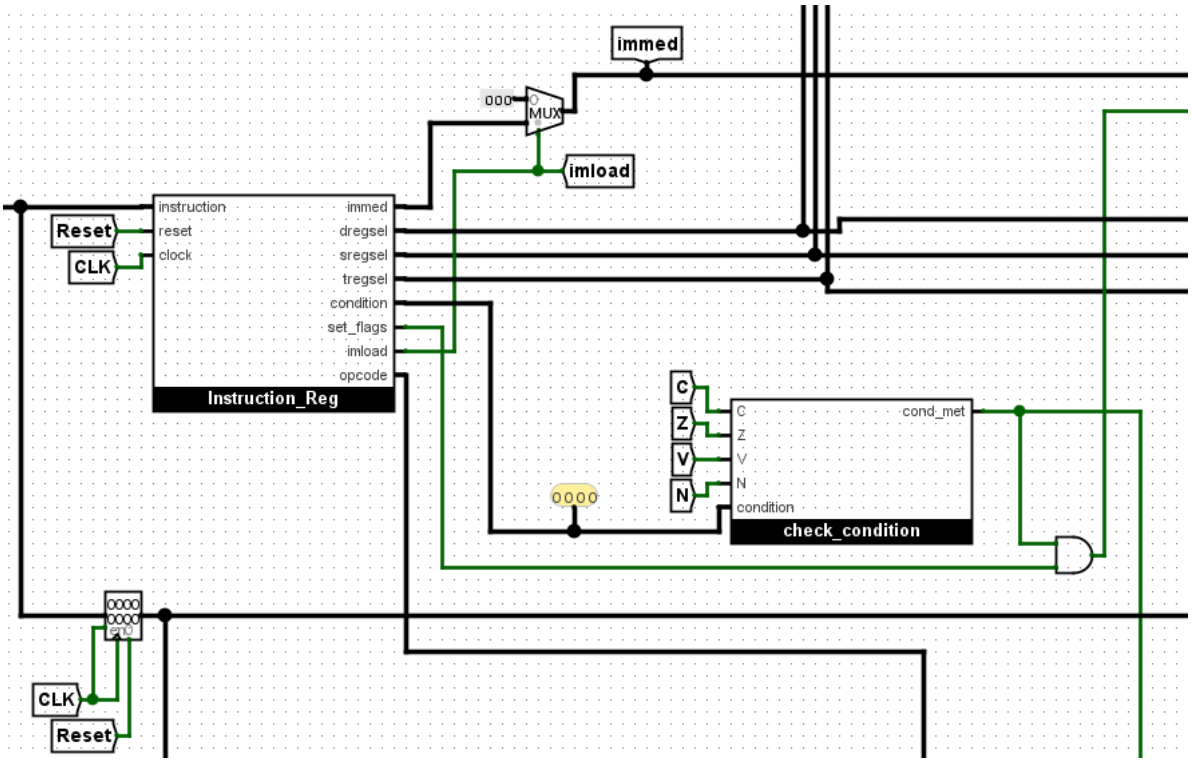


instrUnload MUX

The **instrUnload** signal decides whether the instruction passes through to the ID stage. It's purpose is to stop the instruction that is right after a jump/branch command from passing through.

The Link register works as described on the Instructions page.

Instruction Decode - ID



ID Stage

Inputs

- instruction
- Flags (C, Z, V, N)

Outputs

- immed
- imload
- dregsel
- tregsel
- sregsel
- cond_met
- set_flags

- opcode

The purpose of this stage is to decode the given instruction, decide if it should be executed, and access the needed register values.

The instruction enters the instruction register, from which we receive:

- the immediate **immed** that is only loaded if **imload** bit is also active. This immediate is then passed to the next stage.
- **dsel, ssel** and **tsel** which indicate which registers to access
- the **opcode** that will go to the Control ROM
- the **set_flags** bit that indicates whether the instruction should influence the flags or not.
- the **condition** bits that indicate the condition for which this instruction should execute

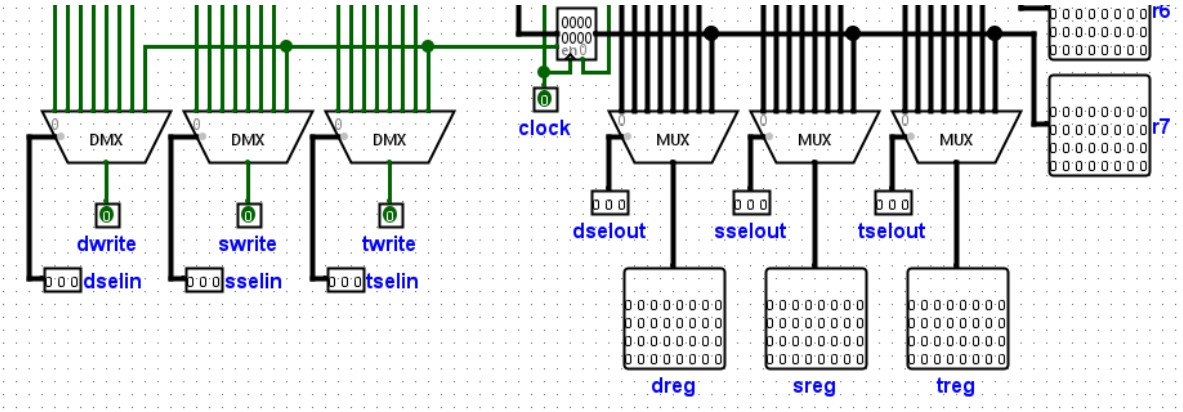
The **condition** bits enter the **check_condition** circuit along with the condition flags (N, Z, C, V). The output is the **condition_met** signal which indicates whether this instruction passes the given condition.

This **cond_met** signal is then passed to the Control ROM along with the **opcode**.

The **set_flags** signal from our Instruction Register passes through an AND gate with the **condition_met** signal and is then passed to the next stage (Execute).

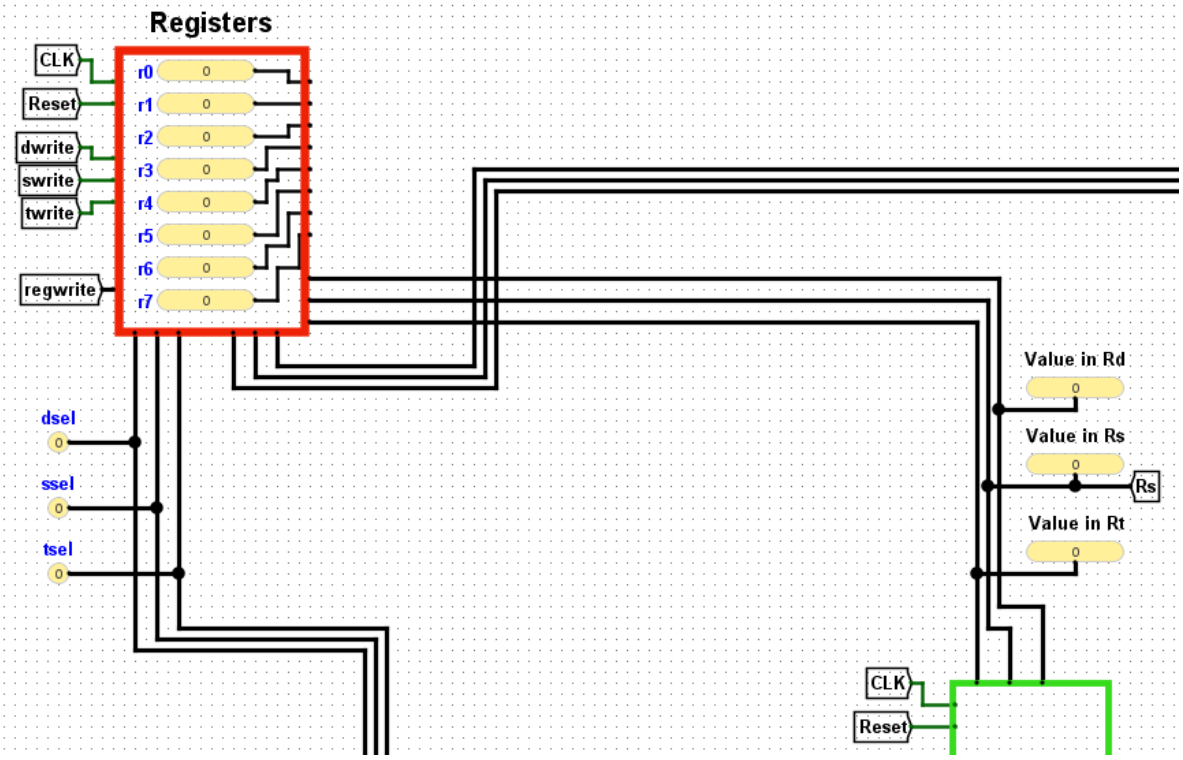
The Register Bank needed to be changed to allow 2 selections at the same time, as both the ID and the WB stage need access to it.

dsel, ssel, tsel have been changed to **dselin, sselin, tselin** for the WB stage and **dselout, sselout, tselout** for the ID stage.



Register Bank change

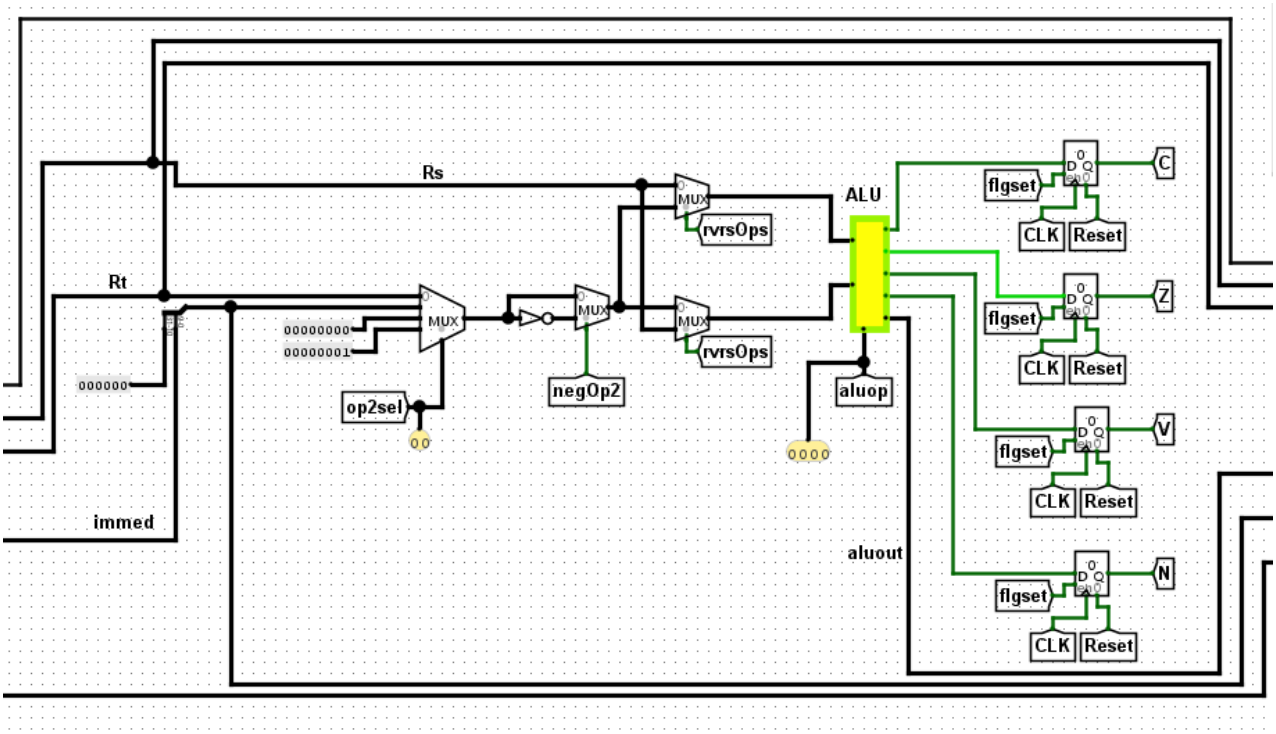
The **dsel, ssel** and **tsel** signals enter the **dselout, sselout, tselout** signals of the Register Bank and from the **dreg, sreg** and **treg** outputs, the values for **Rd**, **Rs** and **Rt** are transferred to the next stage.



Register Bank connected to the ID stage

The **dselin, sselin** and **tselin** inputs of the Register Bank are read from the WB stage, as this is when register write back happens. The values of the current instruction's **dsel, ssel** and **tsel** signals are passed on to everey proceding stage, so the same values may be used in the WB stage 3 cycles from now.

Instruction Execute - EXE



EX Stage

Inputs:

- Rd
- Rs
- Rt
- Immediate (**immed**)
- flags_set
- op2sel
- aluop
- negOp2
- rvrsOps
- dregsel
- tregsel
- sregsel

Outputs:

- aluout
- Flags (C, Z, V, N)
- Rd
- Rs
- Rt
- immed
- dregsel
- tregsel
- sregsel

The purpose of the Execute stage is to transform the data using an ALU operation if needed and change the condition signals (C, Z, V, N) if instructed to.

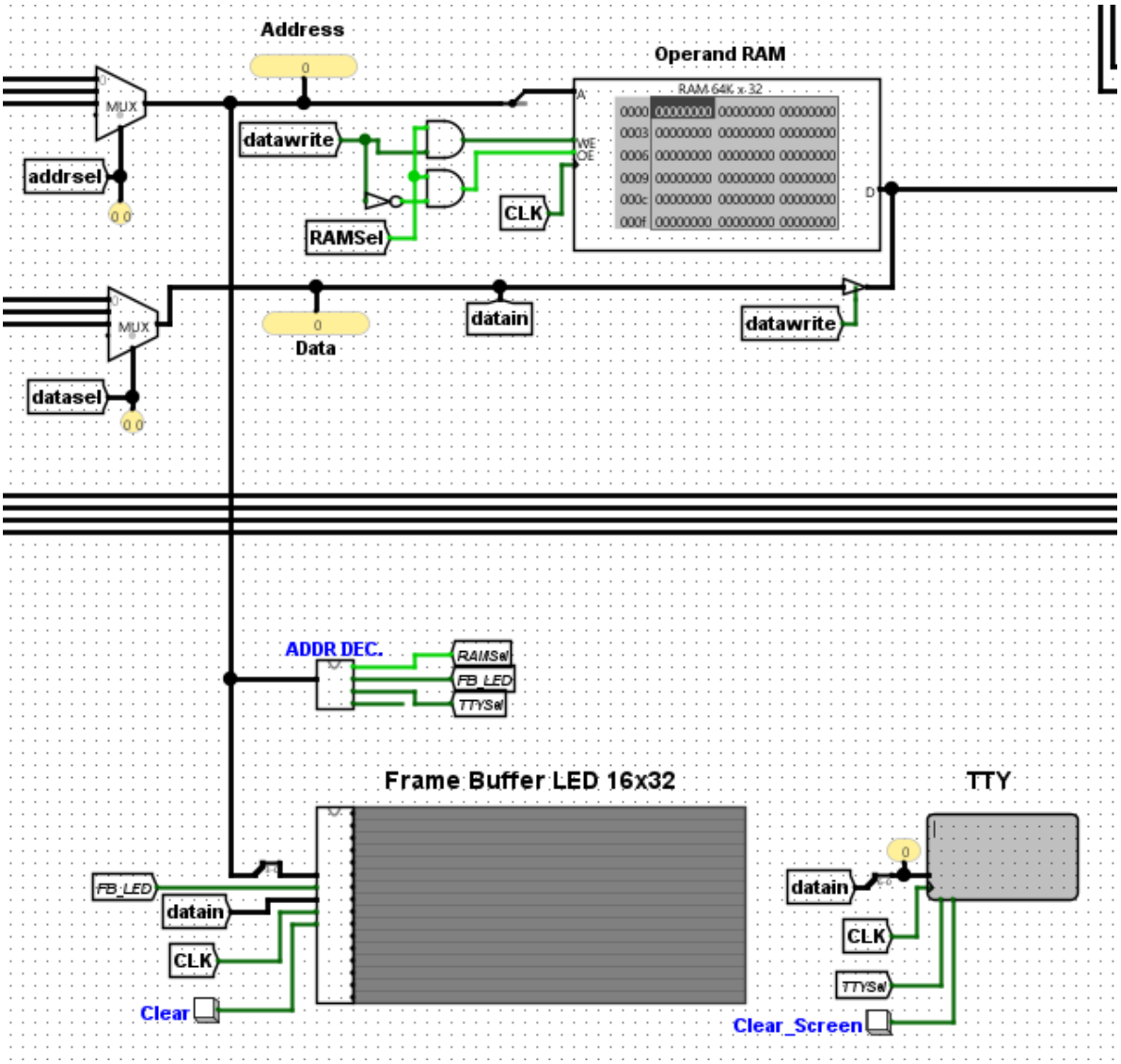
The Execute stage takes the register values and immediate fetched in the ID stage as it's operands for data transformation.

Rs is always taken as the first operand. The **op2sel** signal dictates which values to take as the second operand: **Rt**, the immediate, constant 0 or constant 1. The **negOp2** signal dictates whether we want the second operand to pass through a NOT gate. The **rvrsOps** signal dictates whether we should swap the first and second operand in the ALU.

The **aluop** signal dictates which ALU operation we want to perform. Each of the condition signals are only changed if the **flags_set** signal is active.

The resulting ALU result is passed through to the next stage along with the register and immediate values for this instruction.

Memory Access - MA



MA Stage

Inputs:

- aluout
- Rd
- Rs
- Rt

- immed
- addrsel
- datasel
- datawrite
- dregsel
- tregsel
- sregsel

Outputs:

- aluout
- immed
- Rs
- dregsel
- tregsel
- sregsel

The purpose of the Memory Access stage is to decide whether to read from or write to memory and select data to be written/address to read from.

The **addrsel** signal decides which address in the Operand RAM to read from/write to. It's options are **immed, aluout** and **Rs**.

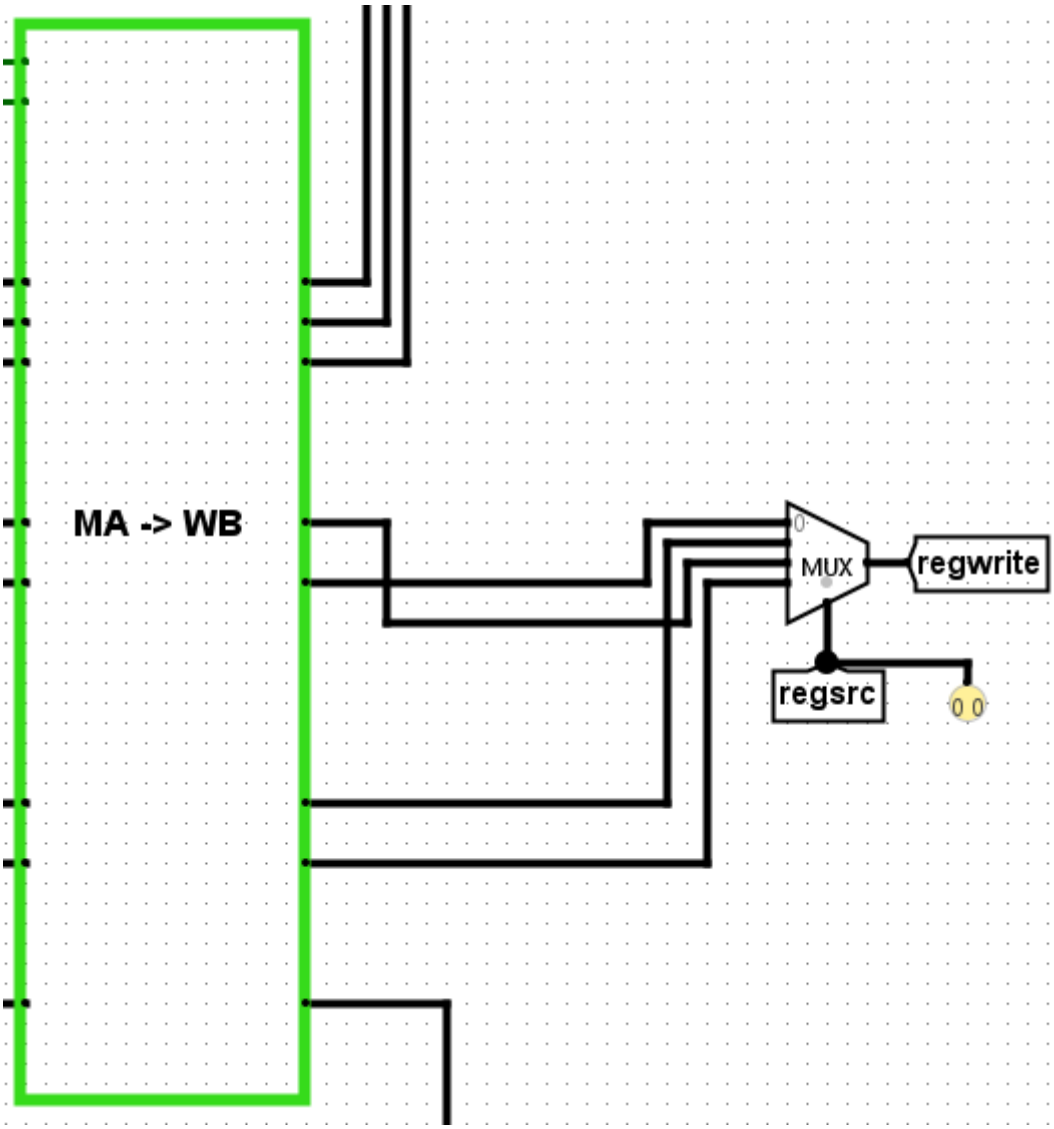
The **datasel** signal decides what data to write to the Operand RAM in the case of a write command. It's options are **Rd, Rt** and **aluot**.

The **datawrite** signal dictates wether data should be written to the RAM.

The selected address and data (**datain**) are passed to the Operand RAM. The Operand RAM transfers the **operand** (that the address points to in the case of a read operation) to the next stage.

The data can also be written to any of the 2 other peripherals, the Frame Buffer LED and the TTY, based on the original MiMo model's address decoding scheme.

Write Back - WB



WB Stage

Inputs:

- Rs
- aluout
- immed
- operand

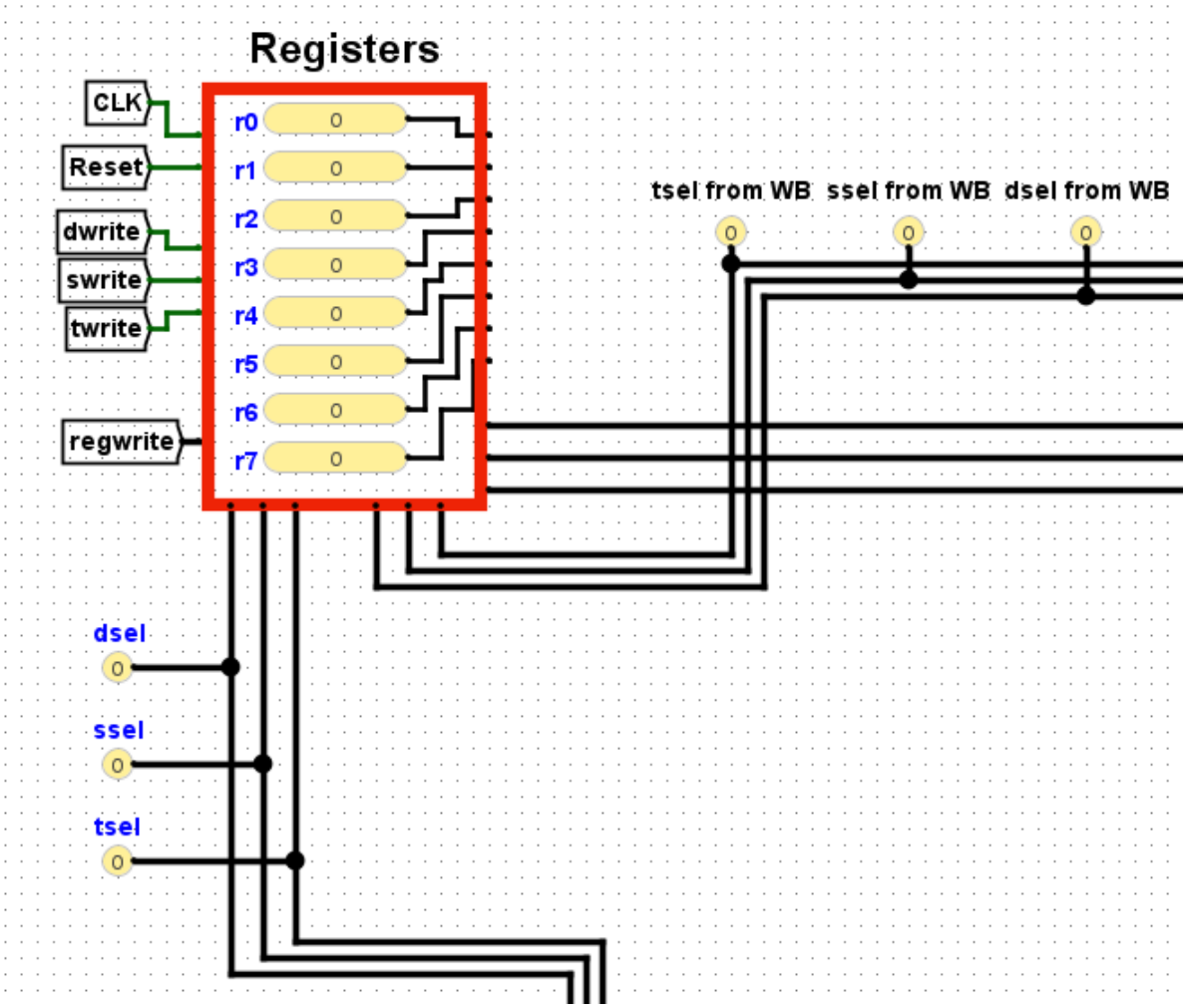
- regsrc
- dwrite
- swrite
- twrite
- dregsel
- tregsel
- sregsel

Outputs:

- regwrite

The purpose of the Write Back stage is to write back to the registers (if needed) any transformed or fetched data in the previous stages.

The **regsrc** signal dictates what data is to be written to the register. It's options are **Rs, immed, operand** (taken from Operand RAM) and **aluout**.



Register Bank connected to the WB stage

The result is the **regwrite** output which enters the Register Bank along with the **dwrite, swrite** and **twrite** signals which indicate whether to write or not.

The **dregsel, sregsel** and **tregsel** signals from 3 cycles before now exit the pipeline and indicate which registers to write to.

The CPU has now completely finished executing the given instruction and it has exited the pipeline in 5 stages, while concurrently processing 4 other instructions in each of the previous stages. This encapsulates the efficiency of the pipelined model over the previous version.

ASSEMBLERS

The assembler and micro assembler files can be found inside the Assembler folder.

Microcode for all instructions is already written in the microcode.txt file

Running them is similar as before:

- We open command prompt and enter: assembler.exe filename.txt
- This produces a filename.iram file intended for the Instruction RAM and a filename.oram file intended for the Operand RAM.
- The same goes for the microassembler.exe file which produces a single *.rom file.

Pipeline Hazard Optimizations

We have created several versions of MiMo pipelined CPU model (v2 and beyond. All changes described so far have been included in first pipelind version, denoted as v2. Then, we also implemented various pipeline hazard optimizations, elaborated in following chapters.

MIMO_32BIT_V2: PURE PIPELINE VERSION

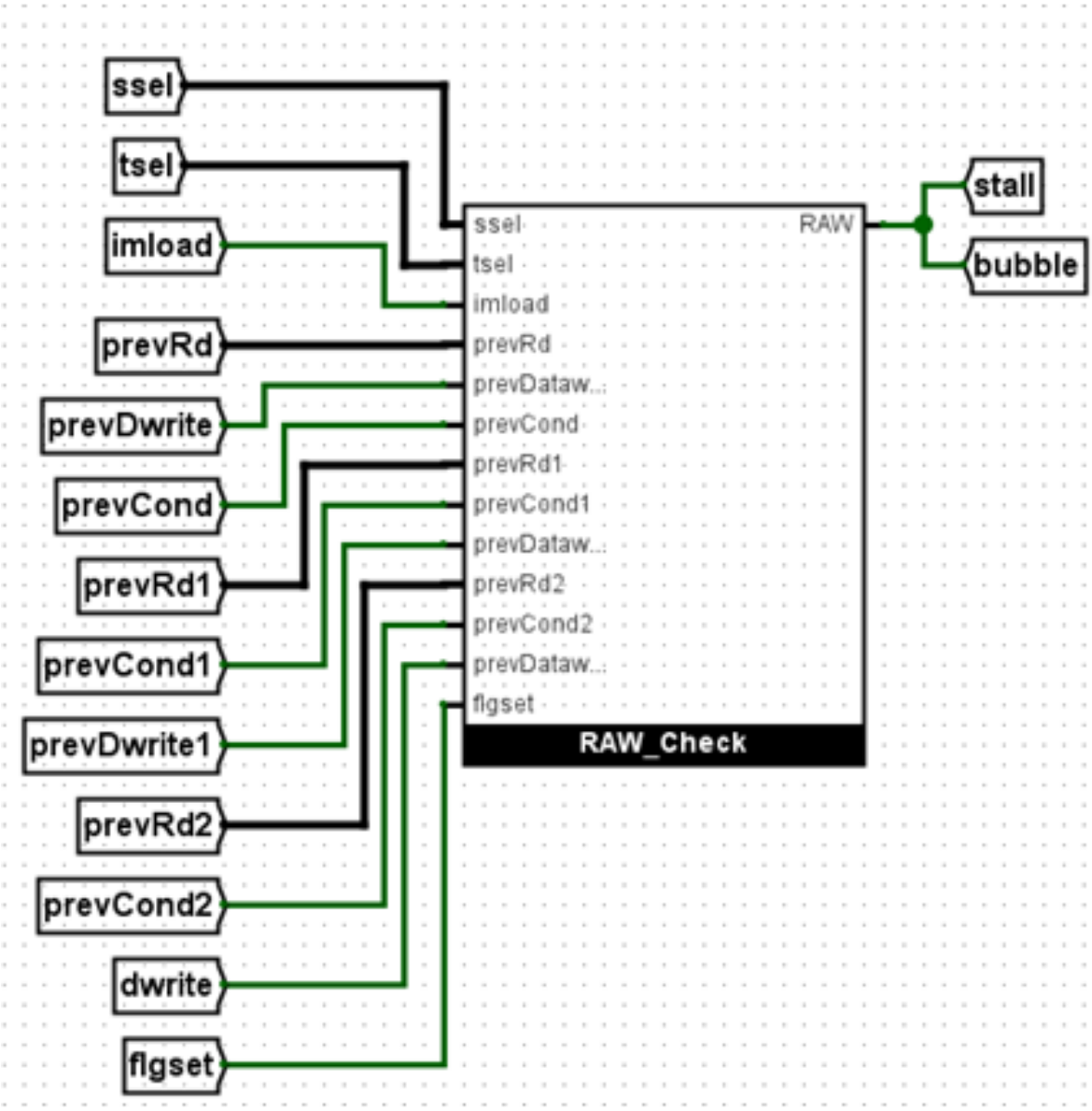
All of the previously described features with no pipeline hazard optimizations have been implemented in mimo_32bit_v2.circ.

While our new pipelined model is more efficient than the previous microprogrammed version v1, it still runs into pipeline hazards that drop it's efficiency by a lot when they occur. The following 3 versions implement various ways to avoid these hazards either completely or as often as possible.

MIMO_32BIT_V2.1: PIPELINE STALL VERSION

The aim of this version is to show how an implementation with pipeline stalls works. The CPU detects when a Read-After-Write error is about to occur and stalls the pipeline accordingly to avoid it.

I added a simple RAW-check circuit in the ID section.



RAW_Check circuit

The circuit checks whether any of the 3 previous instructions:

Wrote to the register bank (**dwrite**=1)

A RAW error occurs when an instruction reads from the Register Bank before a instruction preceding it writes to it.

We check the previous 3 instructions because it takes 3 cycles and pipeline stages for an instruction to go from the ID stage to the end of the WB stage.

Their condition was met

If the condition of the previous instruction is not met, it can not write to the Register Bank and a RAW therefore can not occur.

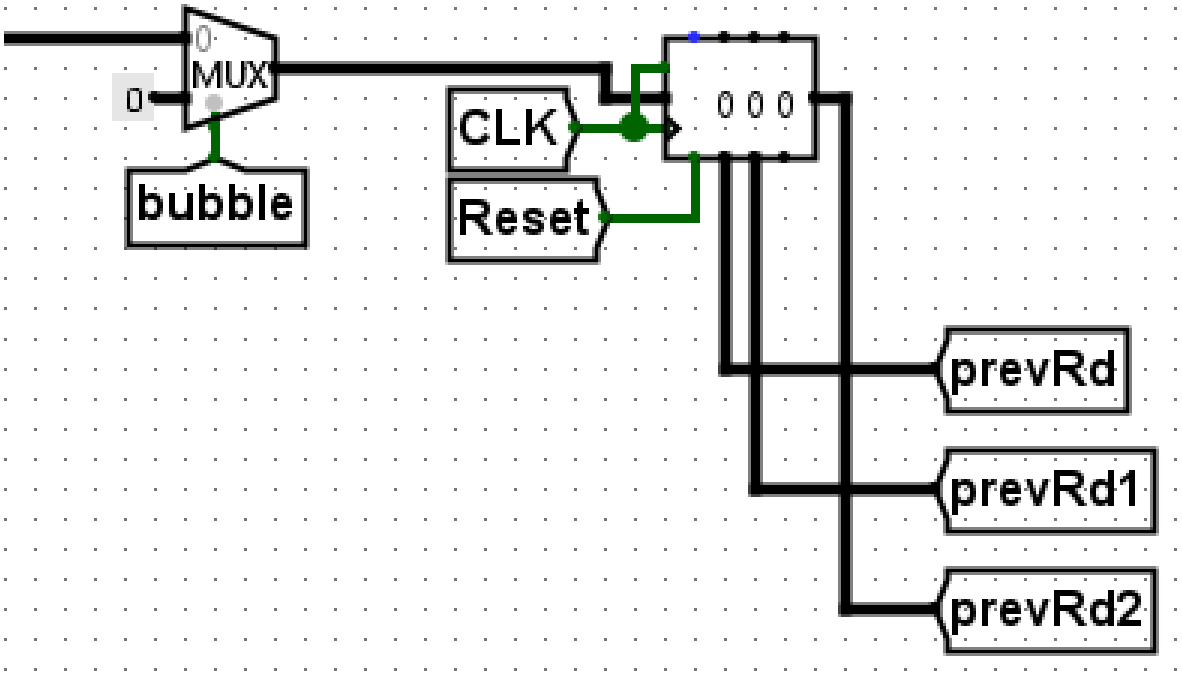
The **dsel** is equal to the current **ssel** or is equal to the current **tsel** while **imload**=0

The destination register of the previous instruction must be equal to one of he registers being used in the current instruction(tsel is only active when not using an immediate value).

The circuit produces the **stall** and **bubble** signals when a incoming RAW is detected.

The **stall** signal stops the **pcload** and **irload** signals, so that no new instruction is loaded into the IF or ID stage.

The **bubble** signal replaces all signals inside the **ID -> EX** intermediate circuit with zeros(bubbles), so that the previous instructions that were causing the RAW may travel through the pipeline until they are no longer a problem.



Bubble connection to avoid infinite loop

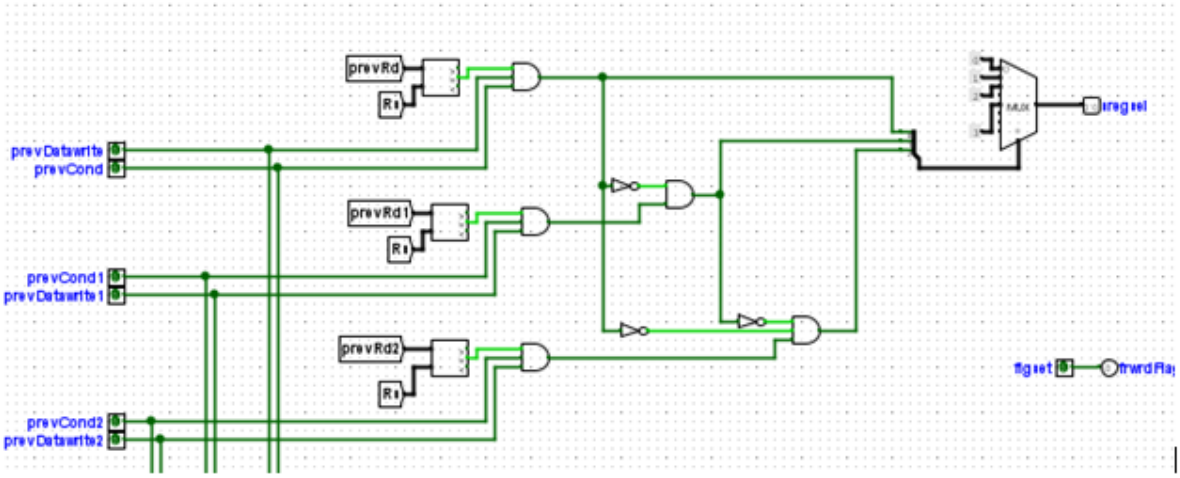
The **bubble** signal is also connected to the shift registers that keep previous values of **Rd** and **cond_met**. This is to ensure that when we encounter multiple RAWs one after another, an infinite loop doesn't arise and a zero(bubble) is shifted into the shift register instead of the same value of the instruction.

An example can be seen and tested in the **Assembler/tests/test10.txt** file.

MIMO_32BIT_V2.2: OPERAND FORWARDING PIPELINE VERSION

The aim of this version is to show how an implementation with operand forwarding works. The CPU detects when a Read-After-Write error is about to occur and forwards operands to the **ID->EX** transitional circuit, thus we always avoid it.

I based it on the previous version **v2.1 – Zaklenitev**, but I changed multiple things.



Changes to RAW_Check curcuit

The RAW checking circuit was changed to produce 2 signals. One to decide from what pipeline stage to forward **Rs** and one to decided from what pipeline stage to forward **Rt**.

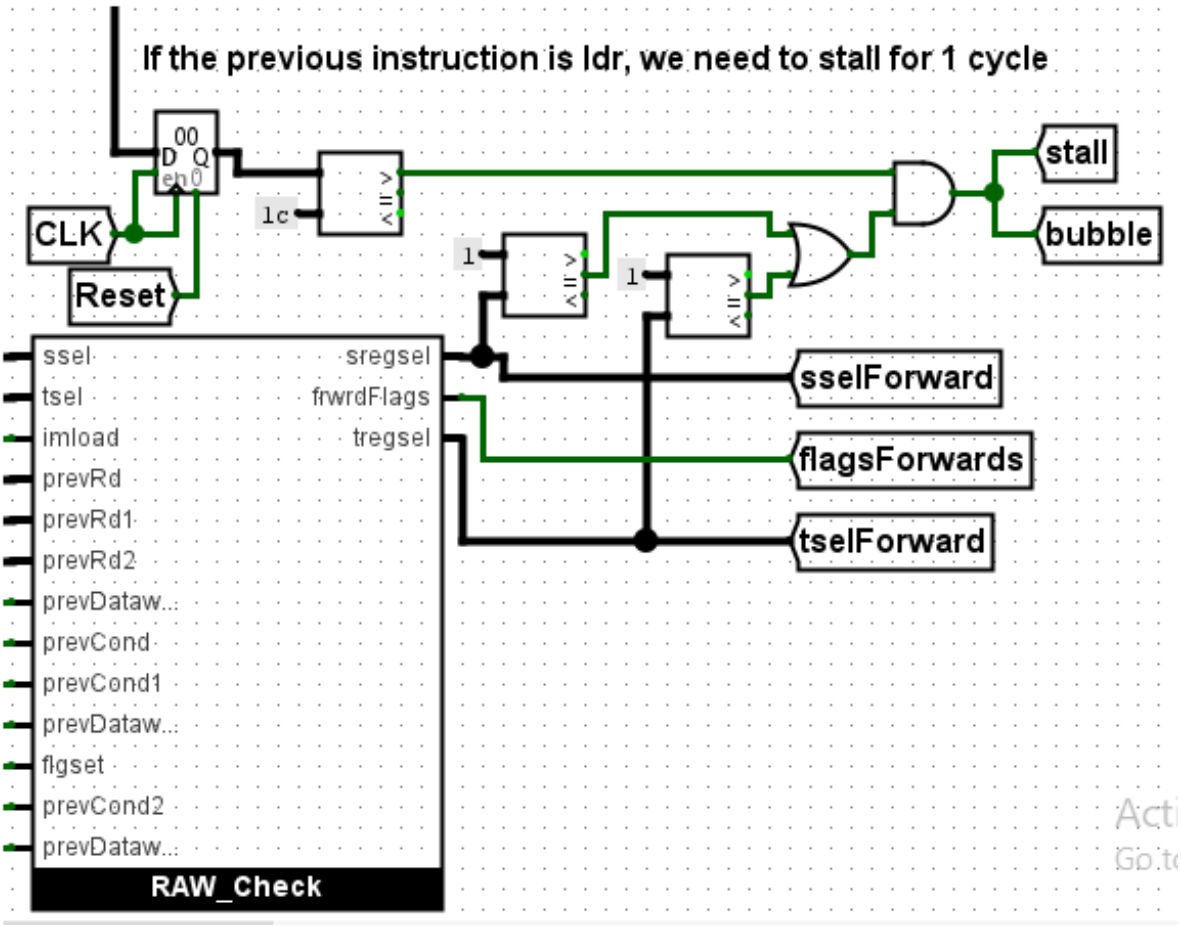
It works on the following principle: If a RAW is detected in the first previous instruction, forward the operand from before the **EX -> MA** stage.

If no RAW is detected in the 1st previous instruction and a RAW is detected in the 2nd previous instruction, forward the operand from before the **MA -> WB** stage.

If no RAW is the detected in the 1st and 2nd previous instructions and a RAW is detected in the 3rd previous instruction, forward the operand from after the **MA -> WB** stage.

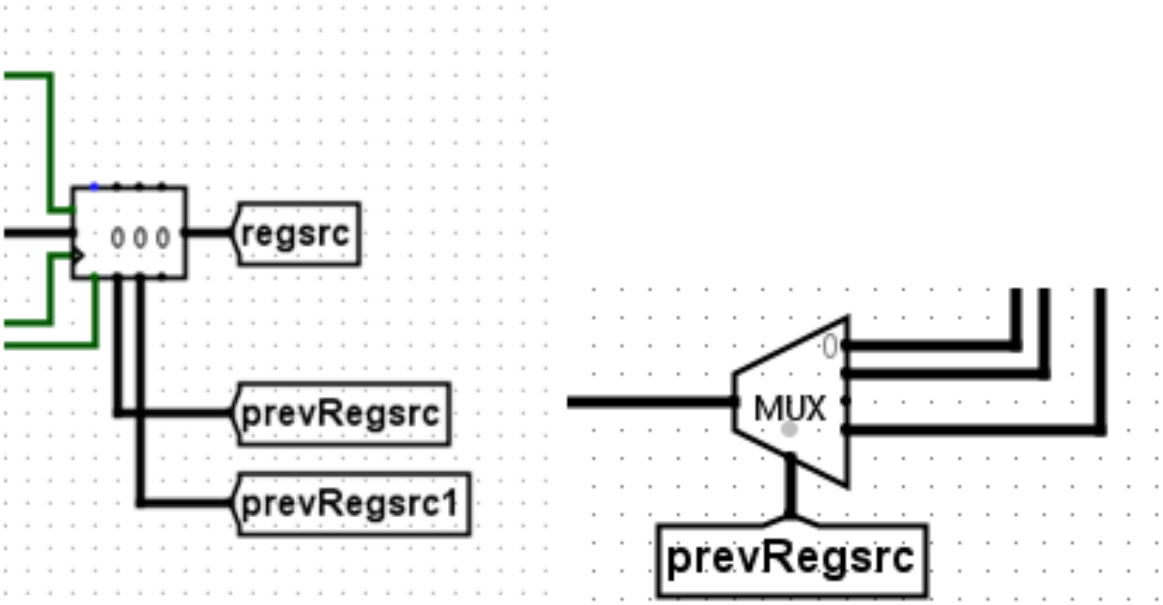
If no RAW is detected at all, no operand forwarding is done.

PIPELINE HAZARD OPTIMIZATIONS



Stall needed for ldr instruction

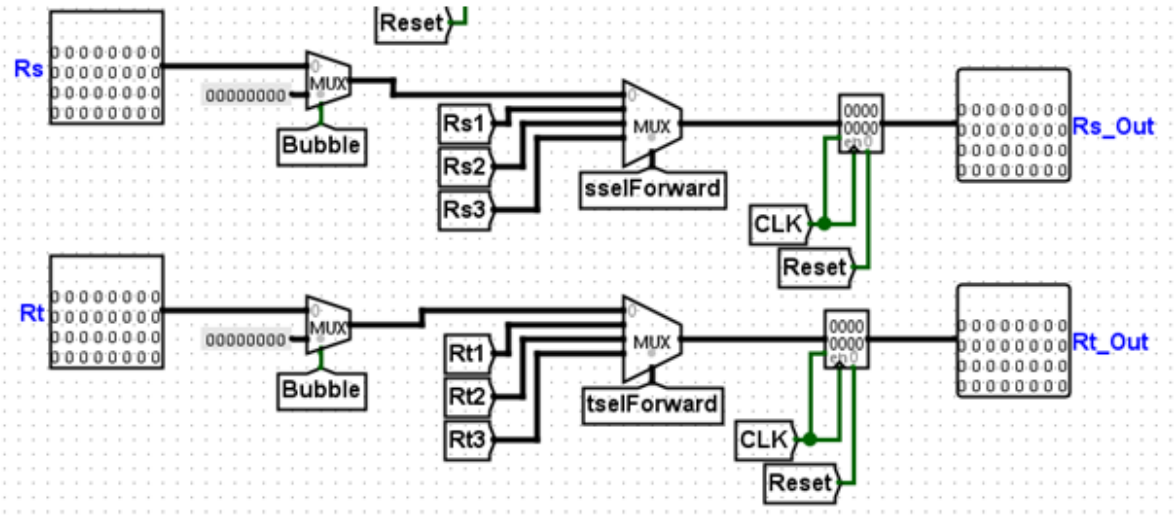
No stall cycles are needed for ALU operations with this method, the only time when a stall is needed is when the 1st previous instruction is a **ldr** instruction and a RAW is detected. Because operand memory access occurs in the 4th stage (MA), we do not know the value to be loaded inside the register to use in the EX stage, so we must stall one cycle to know the value.



Wiring to determine register source for forwarding

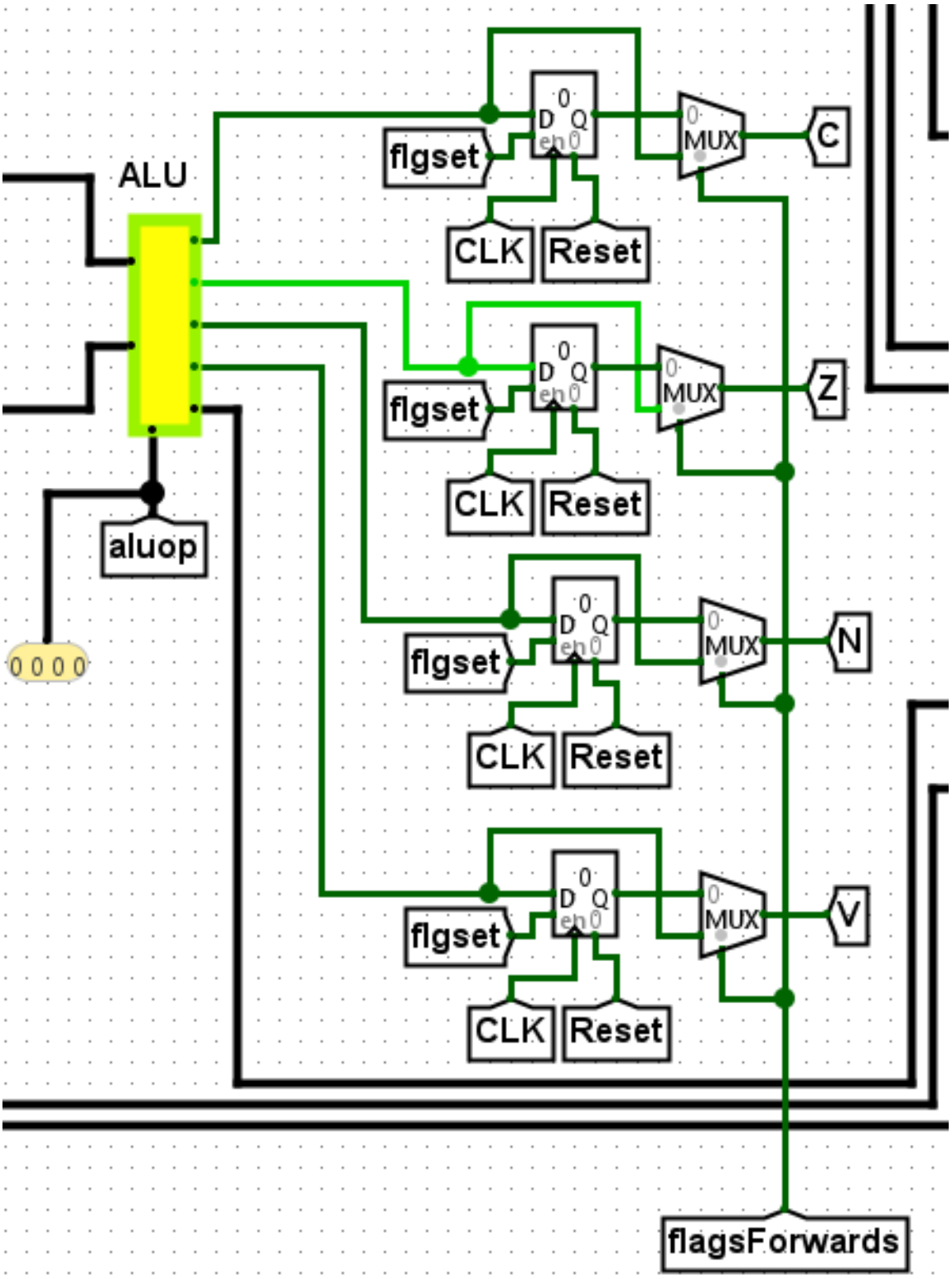
We determine what value (**operand, immed, sreg, aluout**) to forward to the next EX stage by looking at that instruction's **regsrc** signal.

If a RAW was detected in the 1st previous instruction, we take that instruction's **regsrc** signal (**prevRegsrc**) to determine what value to forward. If it was detected in the 2nd previous instruction, **prevRegsrc1** is used, and if it was detected in the 3rd previous instruction, that means the operands are exiting the **MA->WB** transition block but haven't been written to the Register Bank yet, so we take the natural **regsrc** signal.



Forwarding circuitry inside the ID->EX transition block

Inside the **ID -> EX** stage, the value to be loaded into the next EX stage is determined by the **sselforward** and **tselforward** signals produced by the **RAW_Check** circuit.



Flag forwarding

We also forward the flags whenever a flag setting signal is activated the previous instruction, as if we do not do this, we would need to stall one cycle for the proper flags to be loaded into their registers.

An example can be seen and tested in the **Assembler/tests/test11.txt** file.

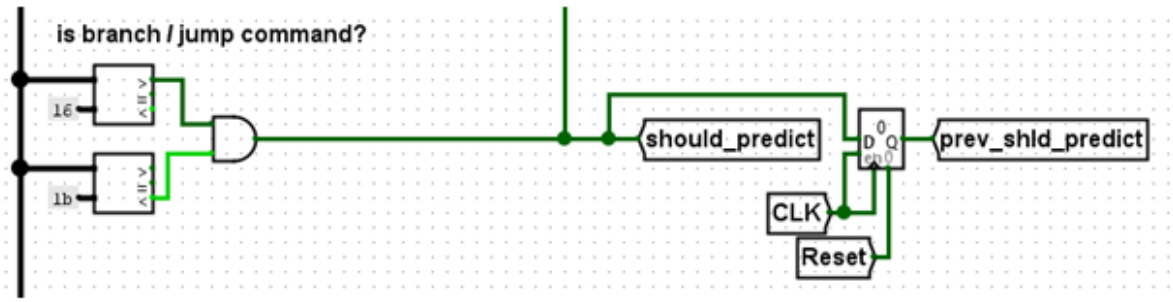
MIMO_32BIT_V2.3.2: BRANCH PREDICTIONS VERSION

The aim of this version is to present an implementation using prediction methods to mitigate delays because of branches.

The model is based on **v2.2 - Premoscanje** and builds upon it.

To implement predictions in our CPU model it only made sense to make the predictions in the IF stage, since naturally we already now the result of the prediction in the ID stage (since we are using **conditions based on flags already set**), so we only lose 1 cycle for every jump naturally. With predictions we want to minimize that to 0 cycles lost as often as possible.

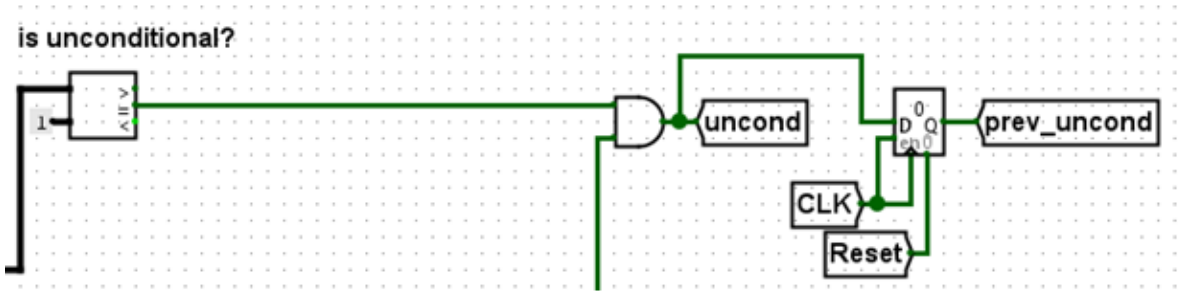
Firstly we underline the mechanisms and changes needed for predicting and missing predictions.



Circuit for checking on branch/jump

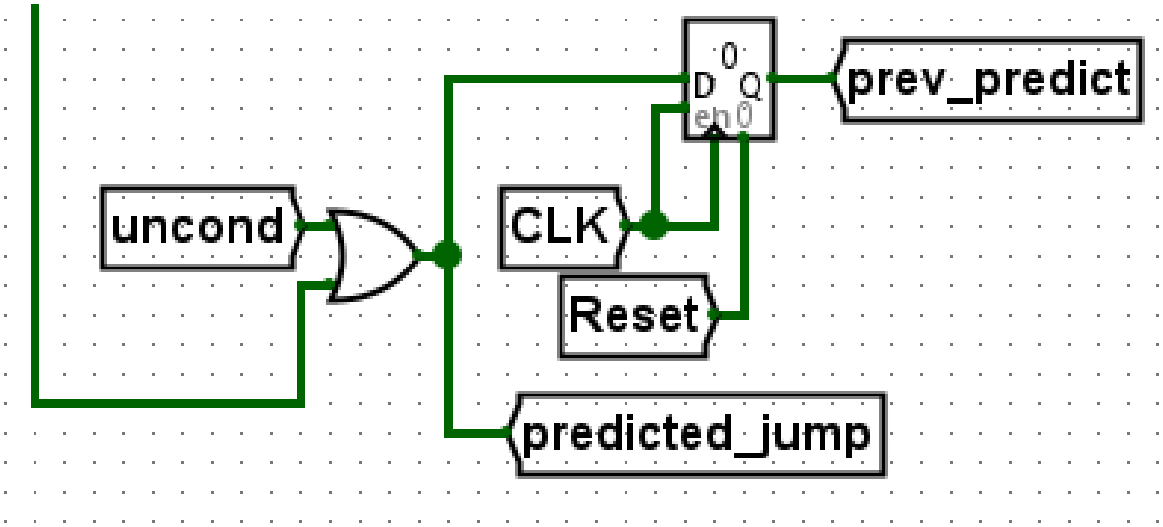
In the IF stage after accessing the RAM, we check whether the instruction is a jump/branch which decides whether we should initialize predictions for that instruction.

We also run it through a register once to know next cycle if the previous instruction was predicted or not.



Circuitry for checking if unconditional branch

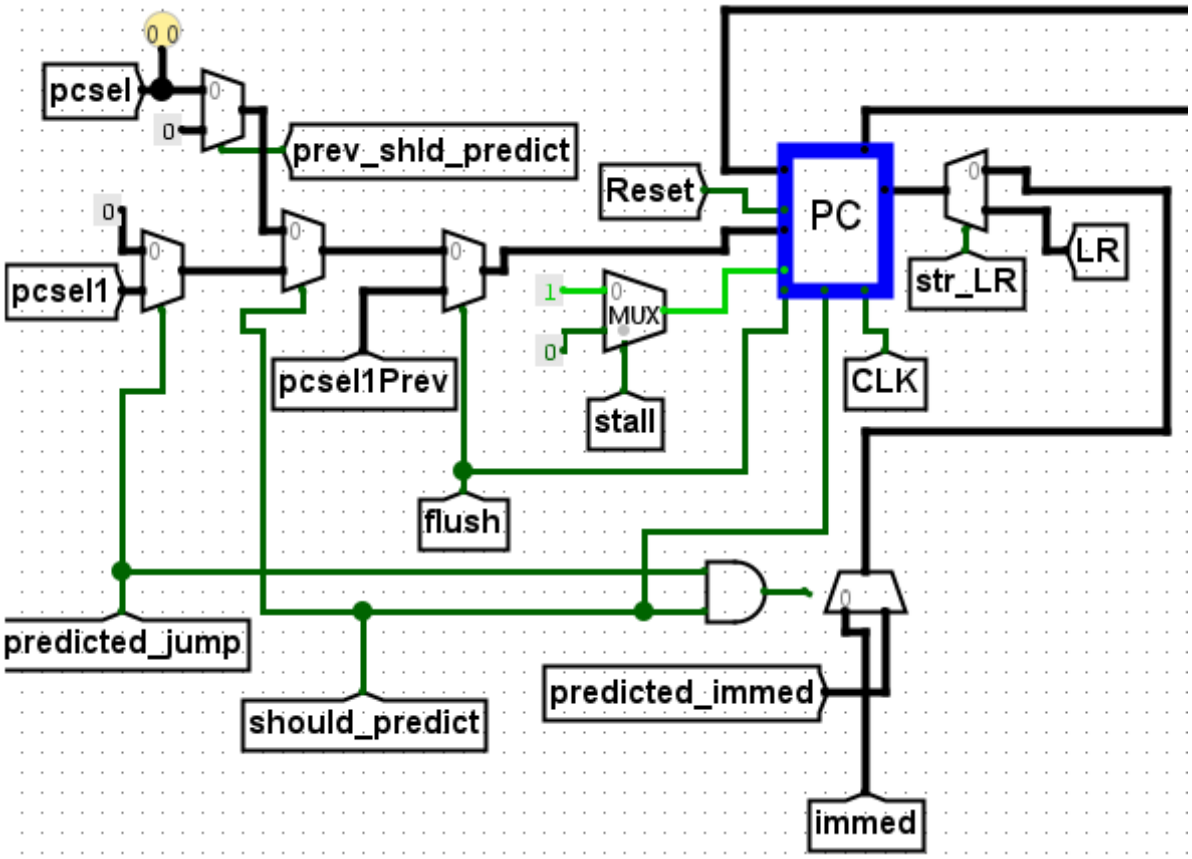
We also check if it a unconditional branch/jump, since if it is, the branch will always be taken.



Prediction

After this we run the instruction through our desired prediction method (described later) and get our prediction. We also run it through a register to know what our prediction was next cycle.

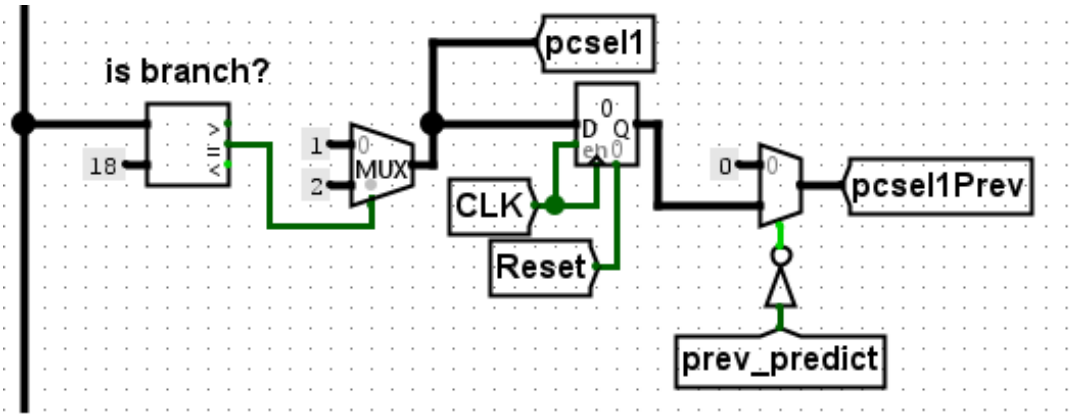
Then, if we predicted a jump, we need to change the **pcsel** for the next fetch.



Changes to pcsel needed for prediction

If we conclude that we should predict (**should_predict**) and if **we did not predict a jump**, the **pcsel** is set to 0 (PC = PC+1), **if we predicted a jump**, the **pcsel** is set to **pcsel1** which is derived from the instruction being processed, explained bellow.

PIPELINE HAZARD OPTIMIZATIONS



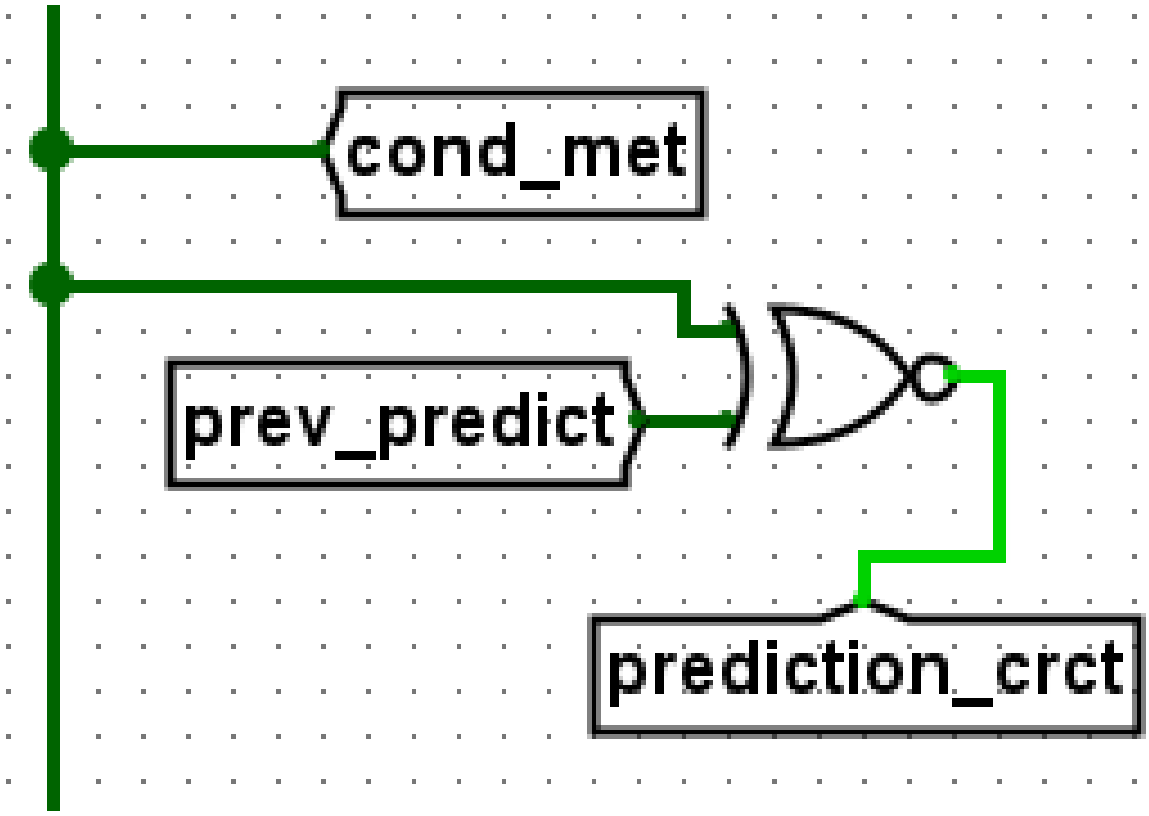
Circuitry for checking if branch command and saving pcsel of branch or jump for prediction

Since if it is a branch or if it is a jump, the value of **pcsel** changes, and we don't know the value of **pcsel** yet since that is derived normally in the ID stage. So we have to derive it as well in the IF stage.

If we should predict and it is predicted to jump, we store the value of the **predicted_immed** inside the **immed** field of the PC. We also derive the **predicted_immed** from the instruction when we decide if we should predict or not.

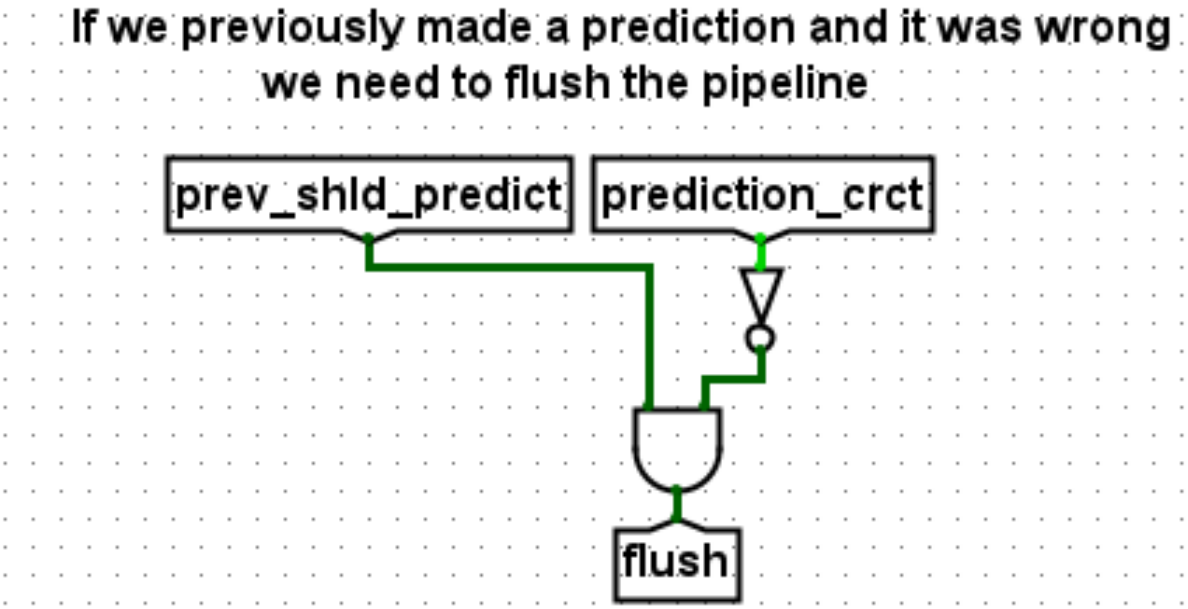
If we should not predict this cycle, the natural **immed** is stored inside the **immed** field of the PC, and the **pcsel** is either the natural **pcsel** derived from the control ROM or it is PC+1, based on whether the previous cycle we predicted an outcome. Since if we previously predicted an outcome, the natural **pcsel** will be that of the jump instruction, instead of the instruction proceeding it.

After we make our prediction the **next cycle we must check if it was correct**.



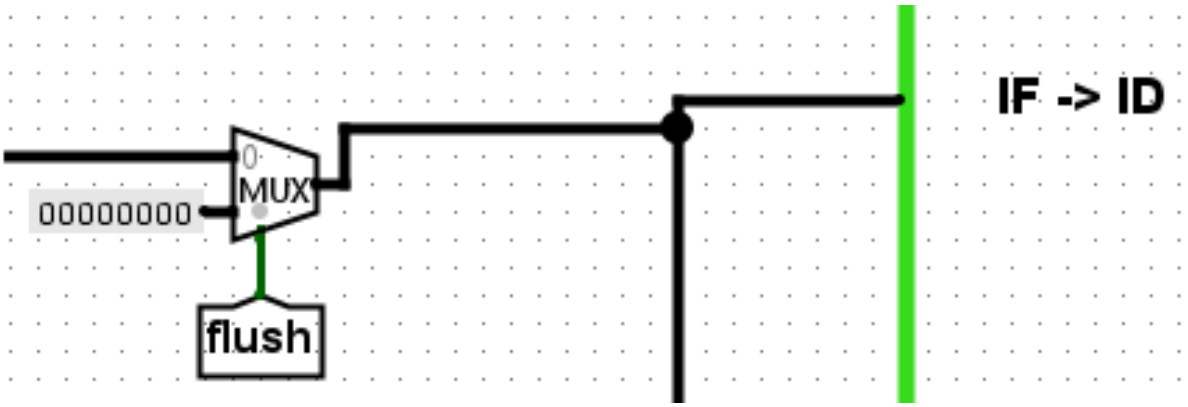
Circuitry for checking if prediction was correct

We check to see if the condition was met for that instruction and whether we previously predicted a jump or not. **If they are the same our prediction was correct**.



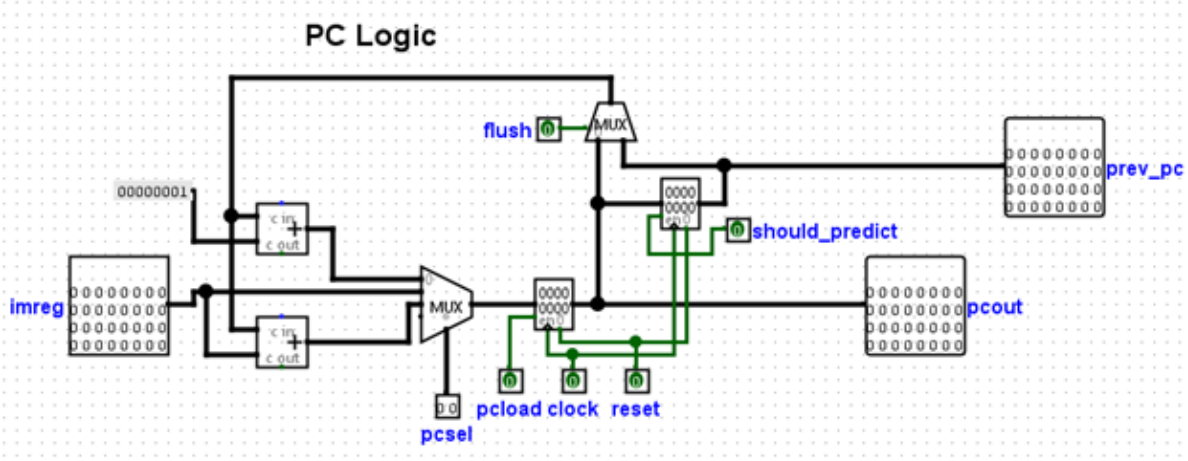
Circuitry for checking for incorrect prediction

But if we previously made a prediction and it was wrong, we need to **flush the pipeline**.



Flushing instruction coming into IF->ID

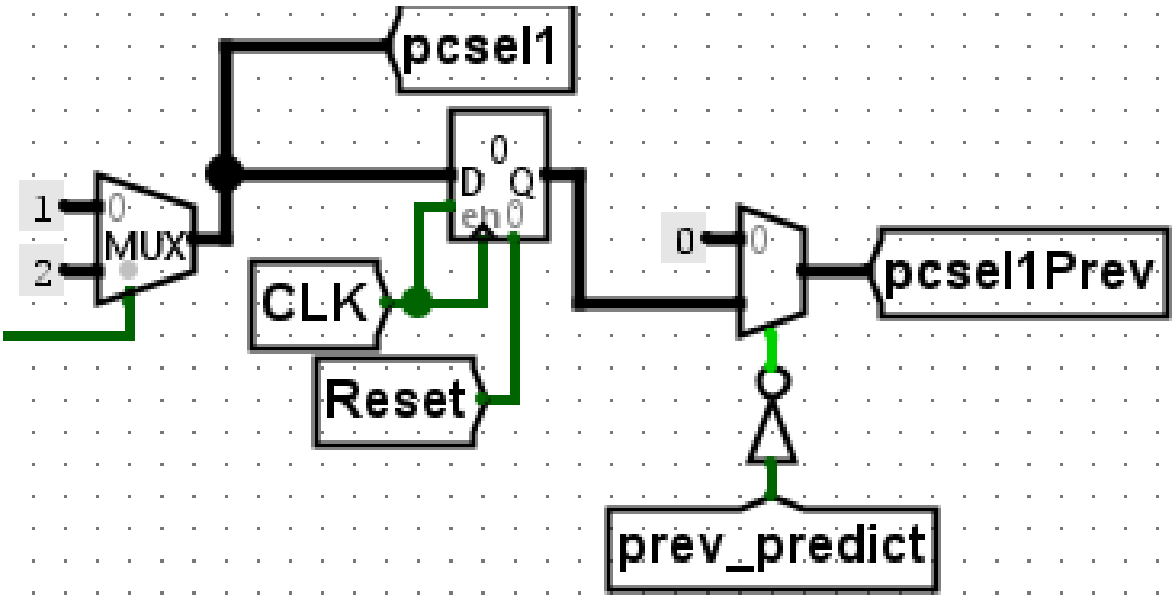
This means we need to remove the instruction coming into the ID stage currently.



Edited PC logic for predictions

We need to set the PC back to what it was previously when we made the prediction. This is why we change the PC logic for this version.

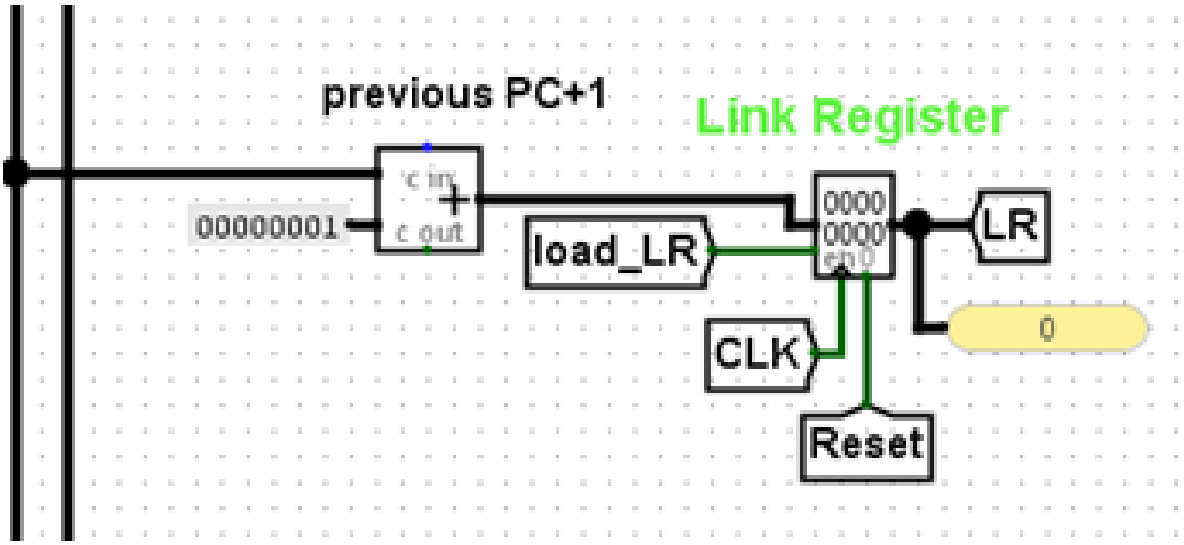
When we detect that we should predict an outcome, the **prev_pc** register is loaded. When a pipeline flush is needed, the **flush** signal directs the **prev_pc** signal instead of the **pcout.**



Circuit for pcsel change for incorrect prediction

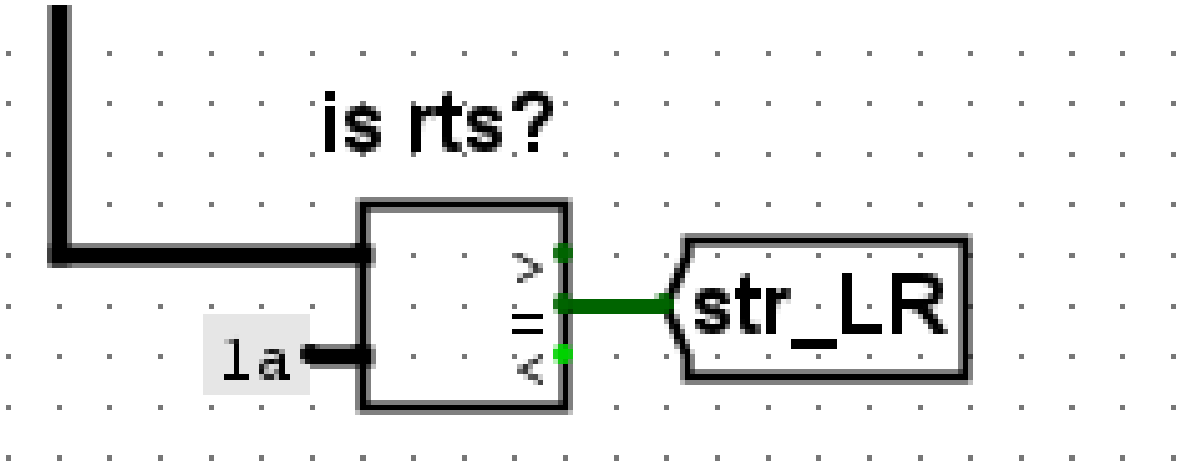
We need to set **pcsel** to the **opposite** of what we predicted previously.

Changes made for bl instruction (branch with link/subroutine call)



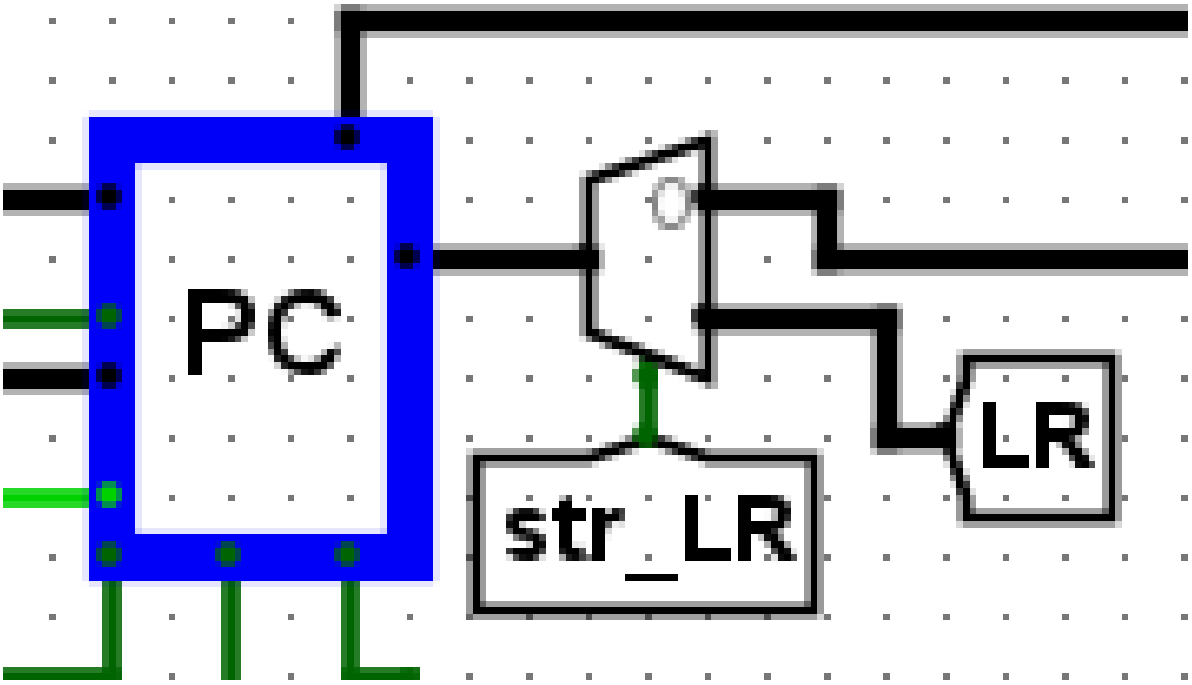
Changes to Link register for predictions

During the ID stage, if the **load_LR** signal is read as true, **the next address after the address that contained the subroutine call** is loaded in the Link Register, i. e. **the previous PC value + 1.** This is taken instead of the current PC value because the Link Register must contain the return address after a subroutine call. In our prediction based model, if we made a pre-emptive jump in the IF stage, the PC value taken in the ID stage would be a completely wrong return address for the subroutine.



»rts« is read and processed right away

Regarding **rts** since it is always an unconditional jump, there is no need to make any prediction. so we read the **str_LR** signal directly from the instruction in the IF stage through the circuitry pictured above.

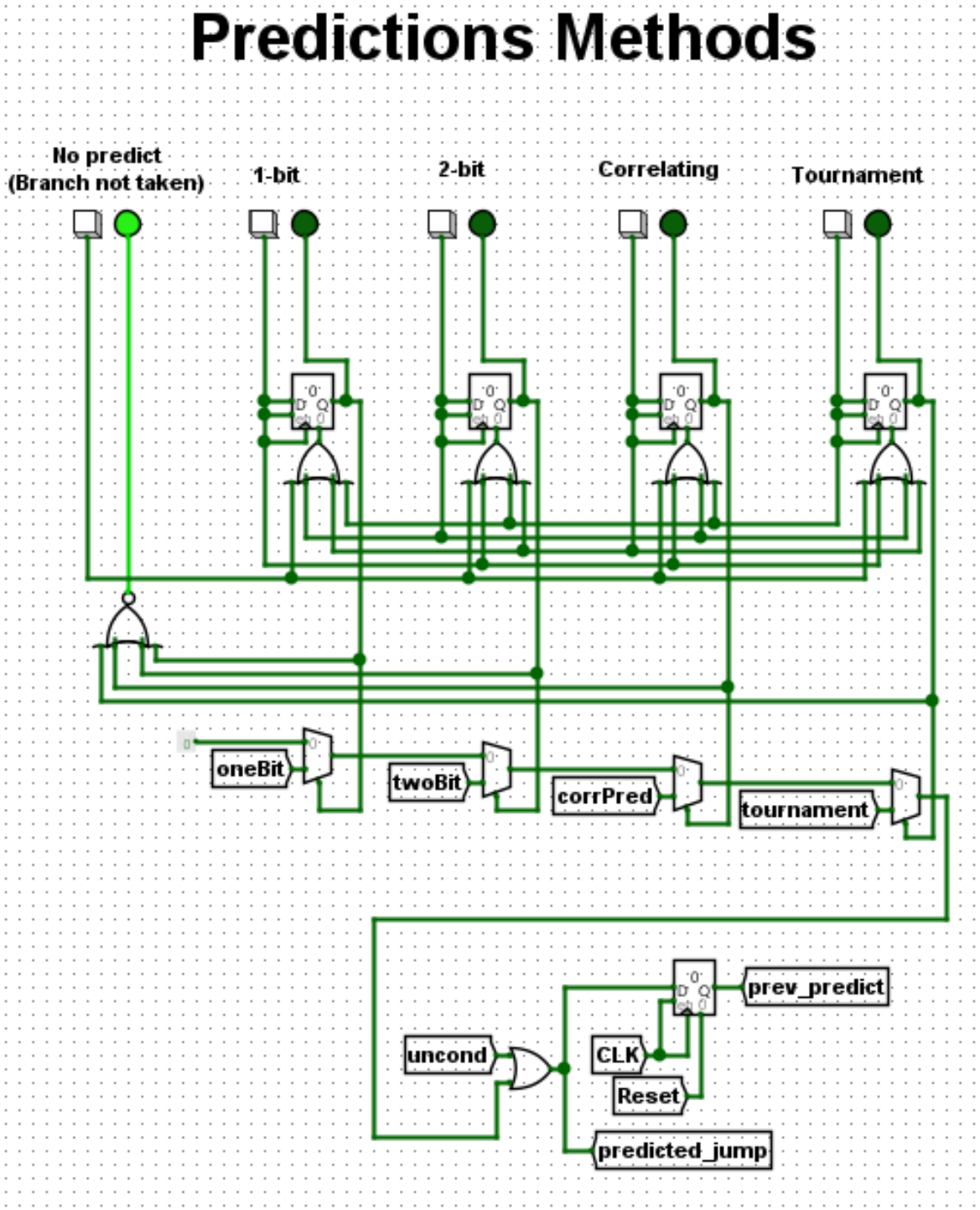


»str_LR« signal directing the PC to take the Link Register value

This **str_LR** signal then directs the PC to take the value stored in the Link Register at the PC's immediate port. Since there are then no predictions to be made, the instruction flows normally and the PC is loaded with the return address next cycle.

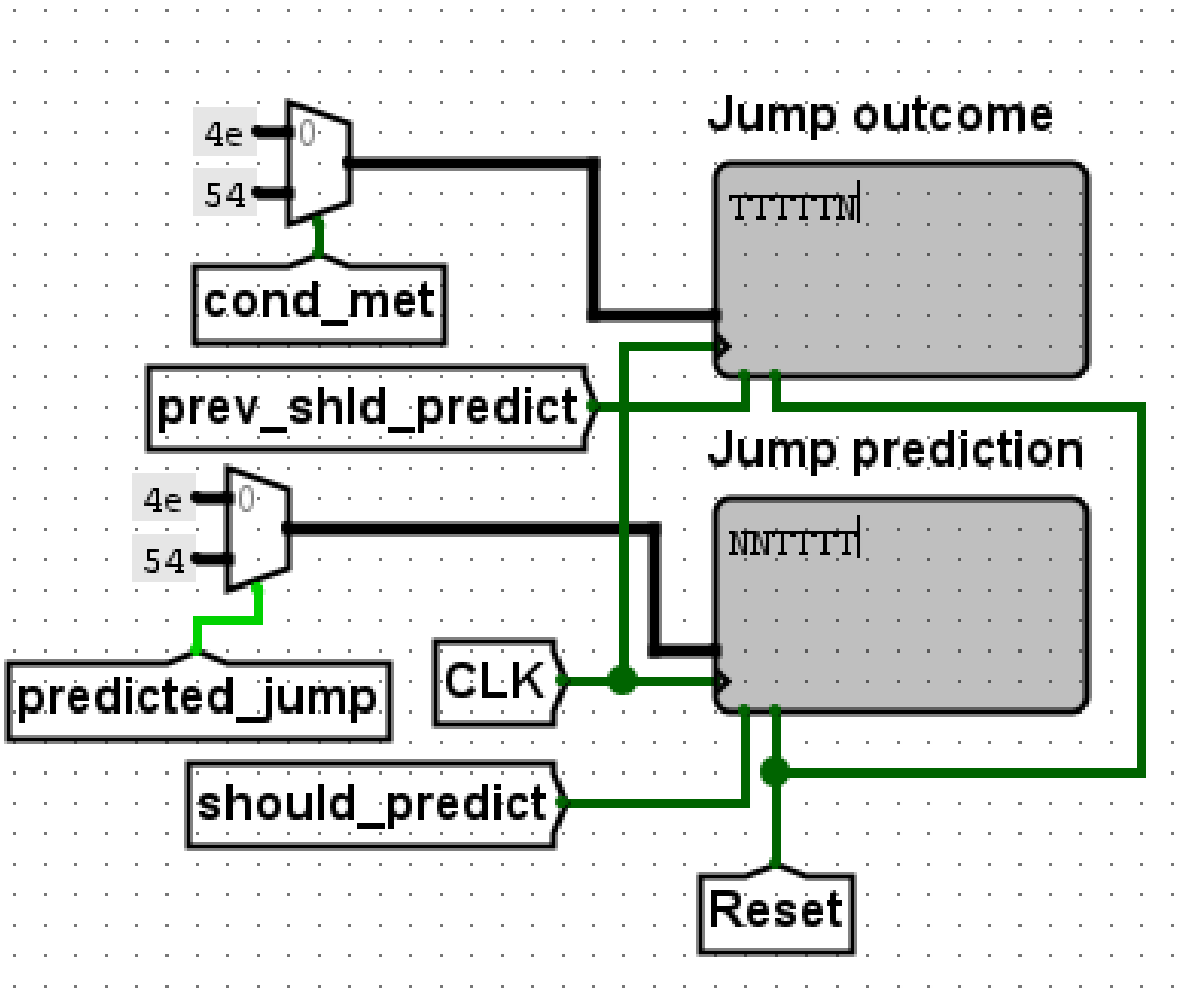
Both the **instrUnload** and **str_LR** control signals are not needed in this version of the CPU.

Branch Prediction Methods



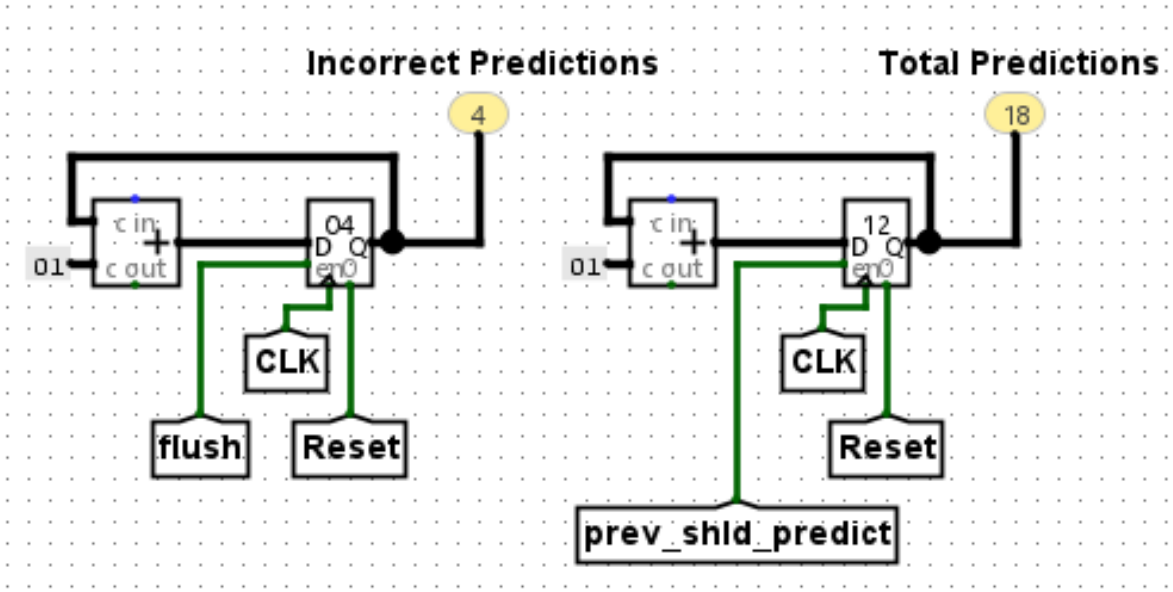
Prediction method toggle circuit

There are 4 prediction methods that we employ with varying rates of success. The method used can be toggled using the buttons pictured above.



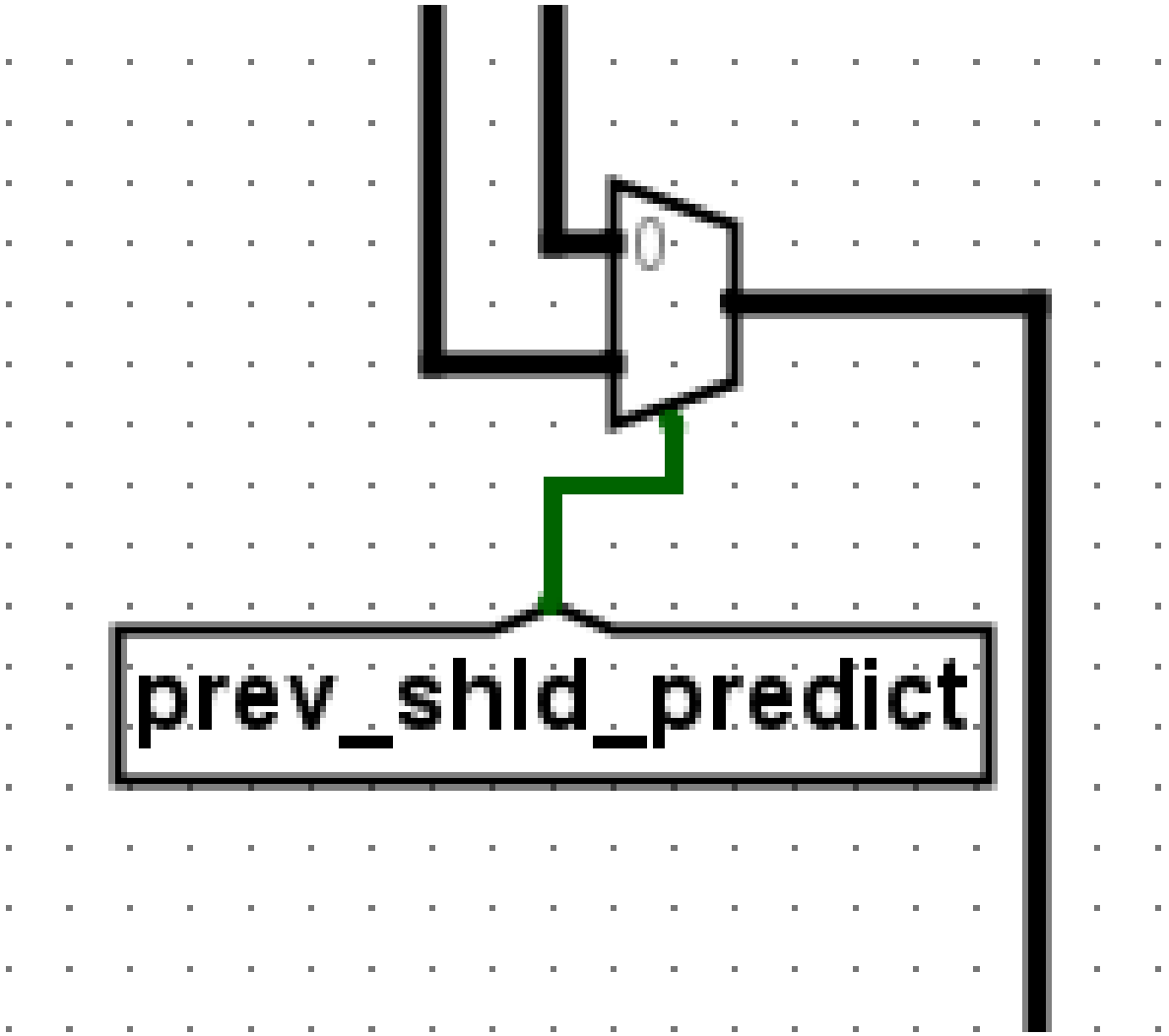
Visual representation of prediction success rate

We also use 2 TTYs to visually represent branches taken(T) and not taken(N) and we can see how the predictions vary from the outcome.



Prediction counters

We also display counts of incorrect and total predictions to see a methods success rate..



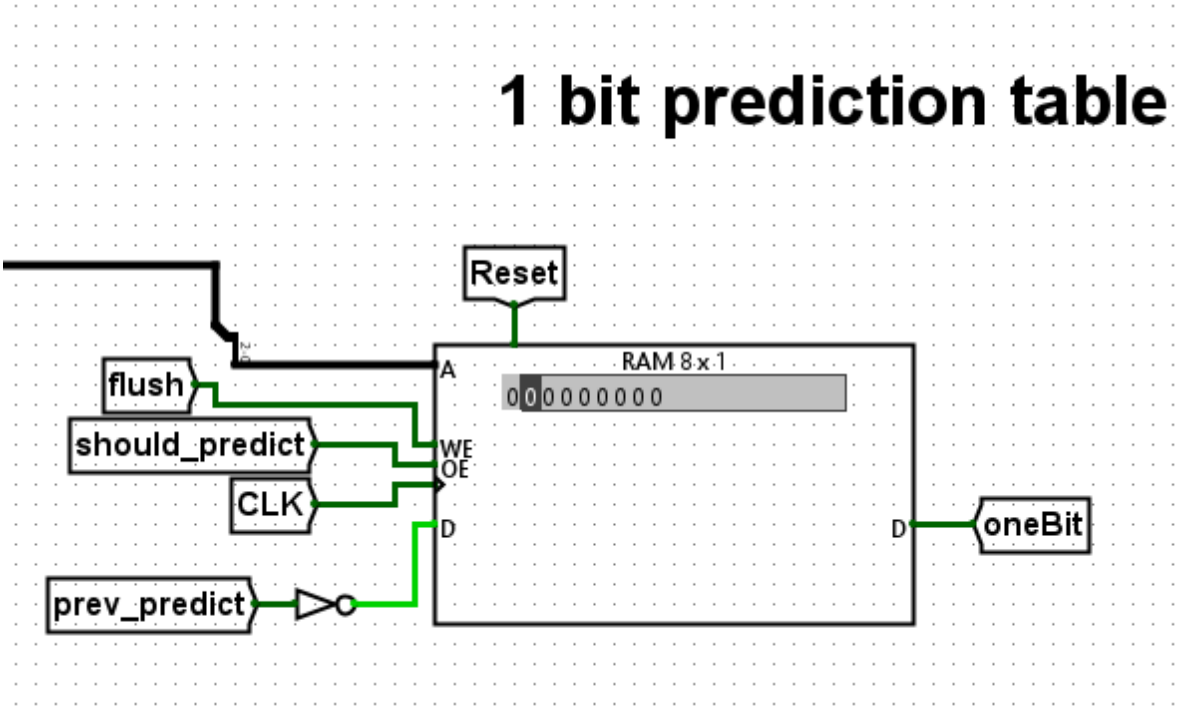
prev_shld_predict *signal indicating prediction result update*

Addressing each prediction method requires the address of the jump/branch instruction. As previously mentioned, there are 2 stages to predictions when encountering a condition jump/branch instruction:

1. In the first stage, the instruction is in the IF stage and we need to make a prediction for the instruction, so when addressing the prediction methods, the current PC value is taken as the address of the instruction.

2. In the second stage, we need to check if our prediction was correct. The instruction is now in the ID stage, so the PC value has changed. All of our prediction methods need to be updated once we know the result of the jump, so we must address them again. So to address our prediction methods, we need to take previous PC value as the address. The **prev_shld_predict** signal indicated that we made a prediction the previous cycle, so we use it to toggle between the current PC and the previous one.

1-bit Prediction Table



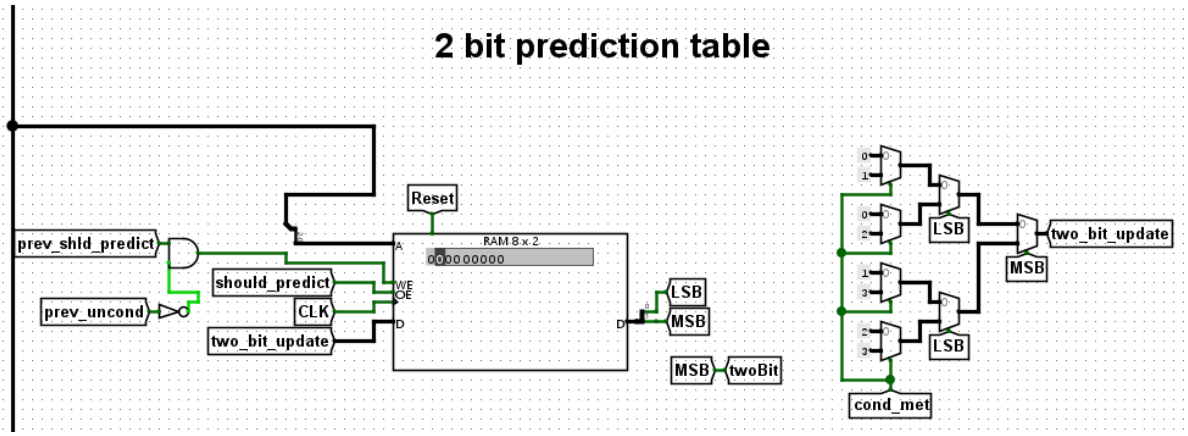
1-bit Prediction Table

The **1-bit prediction table** is the simplest prediction method. We have a 8 address memory that we address by the last 3 bits of our address, each memory location contains a 0 – Don't jump or a 1 – Jump. After each prediction we update the memory based on whether we jumped or not.

This is the simplest method but it is also the least accurate. For every loop there are 2 mandatory missed predictions, one on loop enter, and one on loop exit.

An example using this method can be found in **Assembler/tests/test14-1_bit.txt**

2-bit Prediction Table



2-bit Prdiction Table

The **2-bit prediction table** works in a similar way, but each memory location is comprised of 2 bits, and there are 4 possible states:

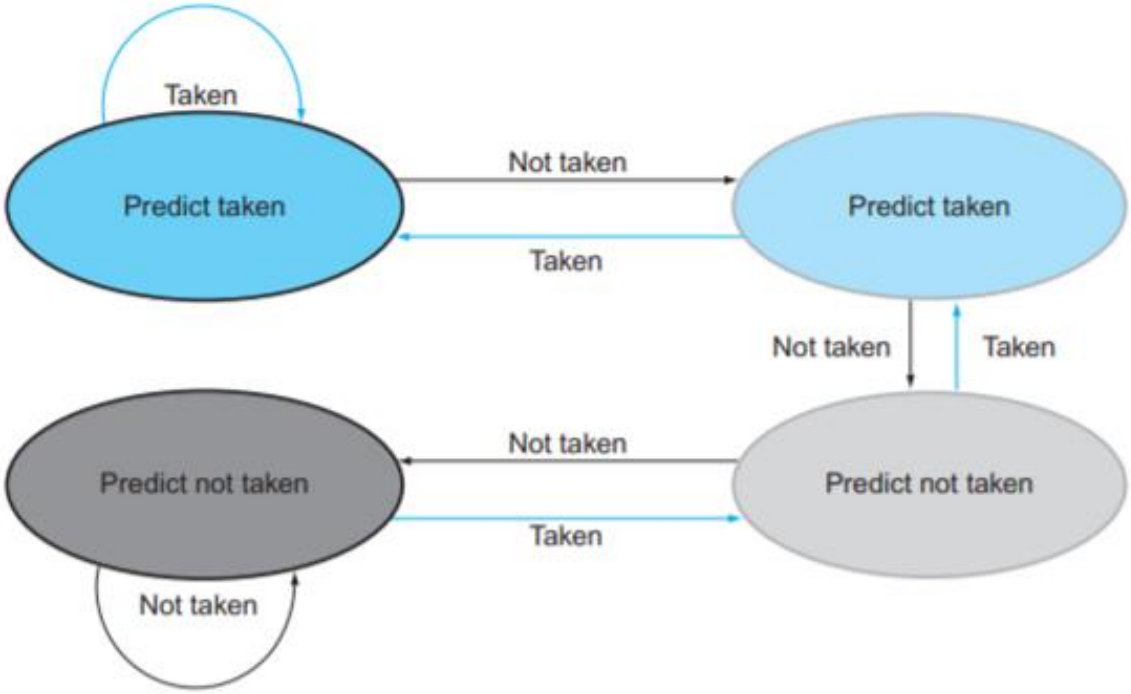
00 – Strong Not Taken

01 – Weak Not Taken

10 – Weak Taken

11 – Strong Taken

The most significant bit signifies whether the branch should be taken or not. The transitions between states are listed in the image bellow:

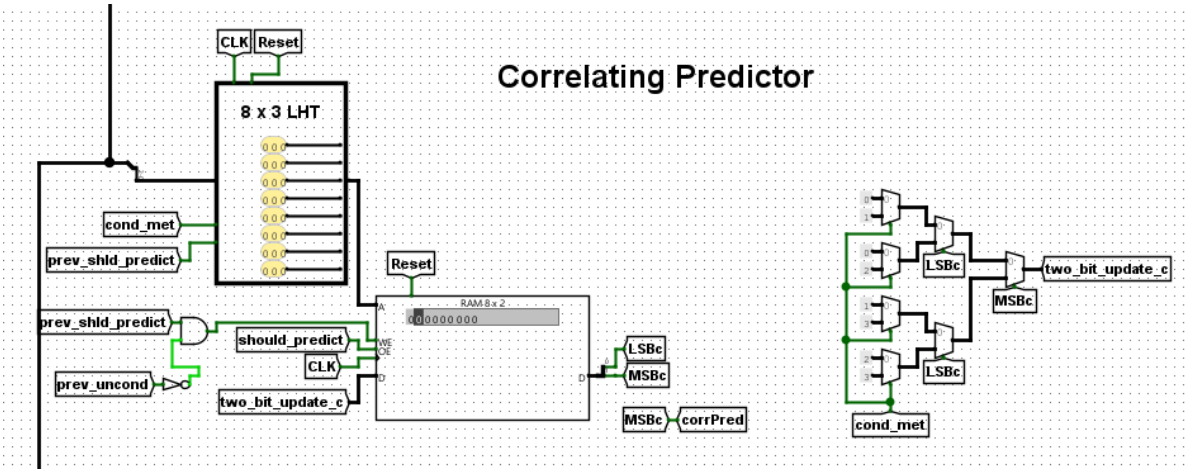


Transition between states

This is more accurate than 1-bit predictions since there is only one mandatory missed prediction on loops, on loop reenters it correctly predicts the reenter.

An example using this method can be found in **Assembler/tests/test15-2_bit.txt**

Correlating Predictor



Correlating Predictor

One problem with the previous prediction methods is that because we only address by the last 3 bits of our address (and we don't have much address space for predictions), sometimes 2 jump instructions can have the same last 3 bits and therefore the same spot in the prediction table.

The correlating predictor fixes this by first running the address through a 8x3 Local History Table(LHT), that reroutes the address to a different prediction table location in our 2-bit predictor, which in this case we call the Local Prediction Table or LPT for short. After a prediction is made, the LHT address is updated by shifting a 1 or 0(Branch Taken or Not Taken) into the addressed LHT spot, based on what was the actual result of the branch. This then points to a different spot in the prediction table(LPT), that will be accessed on the next access of the LHT i.e., the next jump that has the same last 3 bits will not access the same prediction as the one before it.

To illustrate how this works better, we can view the example bellow:

```
while (true){
        if(a % 2 == 0){jump1}    TNTNTNTNTNT
        a++
        if(b == 1){jump2}        NNNNNNNNNNN
}
```

Example code where a Correlating predictor thrives

In the code above, jump1 occurs every second loop, while jump2 never occurs. Let's say that the address of the jump1 instruction is 0010 and the address of the jump2 instruction is 1010. Both addresses share their last 3 bits, 010.

Using a 2-bit prediction table, where we address by the last 3 bits, these 2 instruction would have to share an address space in our prediction table, so their predictions would clash between one another.

```
Branch predictions when using only 2 bit table:
jump1   NNNNNN
jump2   NNNNNN
```

Branch prediction of code above with 2-bit predictor

We end up getting an incorrect prediction for jump1 every second loop.

Using a correlating predictor, our 2 jump instructions would take up different spots in our Local Prediction Table. Using our example code above:

1. The LHT address of 010 would first have a value of 000. This then points to address 000 of our LPT. The LPT address by default would have a value of 00(Strong Not Taken).
2. Jump1 is predicted to not be taken, but in actuality is taken. Our prediction was incorrect.
3. The LHT address 010 is updated with the actual outcome of the branch, in this case since it is taken, a 1 is shifted into the value, so the value goes from

000 to 100. Now this LHT address points to address 100 of the LPT, which is 00(Strong Not Taken) by default. The value in LPT address 000 is updated to 01 (Weak Not Taken) for the future.

4. Jump2 is predicted to not be taken and it is not. Our prediction is correct.
5. The LHT address 010 is again updated with the actual outcome of the branch, now since the branch was not taken, a 0 is shifted into the value, it goes from 100 to 010. Now our LHT address points to address 010 of the LPT.

```
Branch predictions when using correlating predictor:
jump1: NNNNTNTNTNTN
jump2: NNNNNNNNNNNN
```

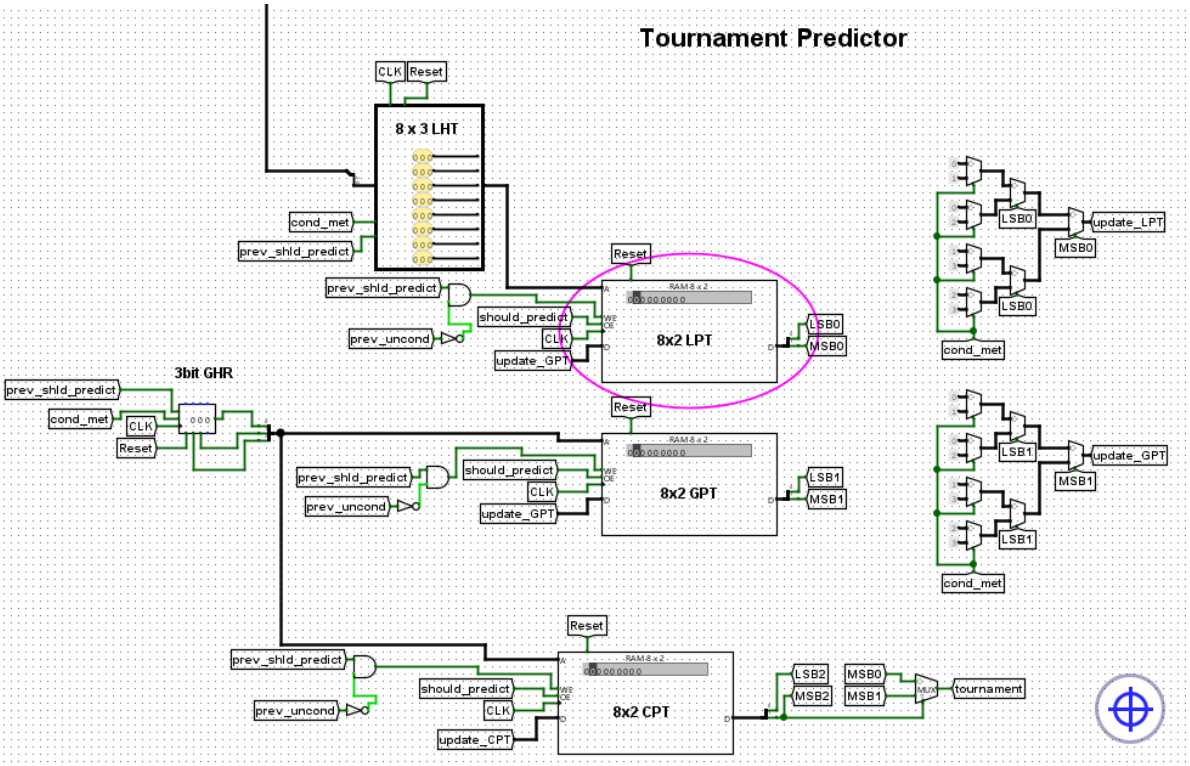
Branch prediction of code above with correlating predictor

Eventually, after the 4th iteration in our example above, we predict correct jumps every time for each of our jump instructions.

The predictions are on average more correct using a correlating predictor when 2 jumps take up the same last 3 bits of the address.

This example can be found in **Assembler/tests/test16-correlating.txt**

Tournament Predictor



Tournament Predictor

One problem with the correlating predictor is that it is not always better than a standard 2-bit predictor. Sometimes the LHT rerouting can interfere with jumps that do not share the same last 3 bits.

The tournament predictor fixes this by **dynamically choosing whether to predict by local memory or global memory**.

We have a Local History Table (LHT) and Local Prediction Table (LPT) that keep track the same as a correlating predictor, the Global History Register is updated after every prediction (1 is shifted into it if branch taken, 0 if not taken), the address from the GHR is used to address the Global Prediction Table (GPT), that acts as our standard 2-bit predictor, and the GHR is also used to address the Choice Prediction Table (CPT), which is updated the same way as a 2-bit prediction table. The result

from the CPT is used to choose whether to take the prediction from the LPT or the GPT when they differ.

Let's look at an expanded form of the example used previously to better understand this.

```
while (true){
        if(a % 2 == 0){jump1}    TNTNTNTNTNT
        a++
        if(b == 1){jump2}        NNNNNNNNNNN

        if(c % 2 == 0){jump3}    TNTNTNTNTNT
        c++
}
```

Expanded code example

We add another jump instruction to our code, jump3, that functions the same as jump1 but importantly **does not share** the last 3 bits of it's address with jump1 and jump2.

Our code flows as follows:

1. AT the start, all of our prediction tables and registers (LHT, LPT, GPT, CPT, GHR) have values of 0. By default the first branch will be predicted as not taken, since the CPT will point to the value in the LPT and that will be 0.
2. When jump1 occurs, the branch is taken and our initial prediction is incorrect.
3. The LHT and LPT are updated the same as with the correlating predictor above, the LHT address 010 is updated with the value 100, the LPT address 000 is updated to 01 (Weak Not Taken).
4. The Global Prediction Table (GPT) address 000 is updated from 00 to 01 as well. This acts is a standard 2-bit predictor.

5. The Choice Prediction Table(CPT) address 000 is updated from 01 to 01 as well. This will again point to the LPT value next iteration.
6. The Global History Register (GHR) is updated with the actual outcome of the branch, so a 1 is shifted in this case, it goes from 000 to 100. This now points to a different address in our GPT and CPT.
7. Now jump2 occurs, The value in our GHR is 100 so it points to addresses 100 in our CPT and GPT. CPT address 100 contains the value of 00, so it again points to the value in our LPT.
8. The LHT is addressed by address 010 again, but it now points to address 100 in the LPT, which contains the value 00.
9. jump2 is predicted as not taken and is not. Our prediction was correct.
10. The LHT address 010 is updated with the value 010. The LPT address 100 value stays 00. The GPT address 100 is stays 00. The CPT address 100 also stays 00. The GHR is updated to 010.

```
Branch outcomes:
jump1                TNTNTNTNTNTNTNTN
jump2                NNNNNNNNNNNNNNNN
jump3                TNTNTNTNTNTNTNTN

Correlating predictor predictions:
jump1                NNNNNNTNTNTN
jump2                NNNNNTNTNTNT
jump3                NNNNTNTNTNTN

Tournament predictions:
jump1                NNNNNNTNTNTN
jump2                NNNNNNNNNNNN
jump3                NNNNTNTNTNTN
```

Branch predictions of code above

Because of the extra independent jump, our Local History Table is being modified and we are incorrectly predicting every second jump2 using the correlating predictor. After the 5th iteration of jump2, the tournament predictor is much more correct.

The tournament predictor has the highest rate of correct prediction out of every method used and is the most advanced method here.

This example can be found in **Assembler/tests/test17-tournament.txt**