**Simulating and implementing a specific pipeline MIPS processor using Logisim**

* **Eslam Hussein Mohamed (Leader)**
* **Mohamed Gamal Mohamed**
* **Abdelrahman Badr**

**Email:** eslamhussein579@gmail.com

* **Table of contents:**
* Chapter one phase (1) Design and implementing of a single cycle MIPS processor…………………………………………………………………
* The main control unit………………………………………………..
* Register file………………………………………………………….
* The arithmetical and logical unit…………………………………….
* The overall datapath…………………………………………………
* Chapter two phase (2) Design and implementing of a pipelining MIPS processor………………………………………………………………...
* The pipelining registers……………………………………………..
* Pipelining hazards…………………………………………………..
* Hazard unit part (1)…………………………………….……………
* Hazard unit part (2)…………………………………….…………….
* Hazard unit part (3)…………………………………….…………….
* Main Hazard unit....…………………………………….…………….
* Execution stage forwarding unit (A)…………………………………
* Execution stage forwarding unit (B)…………………………………
* Decoding stage forwarding unit (C)…………………………………
* Decoding stage forwarding unit (D)…………………………………
* The overall datapath…………………………………………………
* Chapter three testing and simulations……………………………….….
* Test code (1)…………………………………………...……………
* Test code (2)…………………………………………….……………
* Test code (3)…………………………………………...……………
* **phase (1) Design and implementing of a single cycle MIPS processor ⮘**
* **The main control unit**

The start point of designing any processor is the control unit design. It is a circuit that takes the instruction’s operation code and brings out the control signals that related with performing and processing the current instruction.

* **Main control unit design procedures:**

1. constricting the control signals truth table of the control unit that support all instructions of the given processor as shown below:







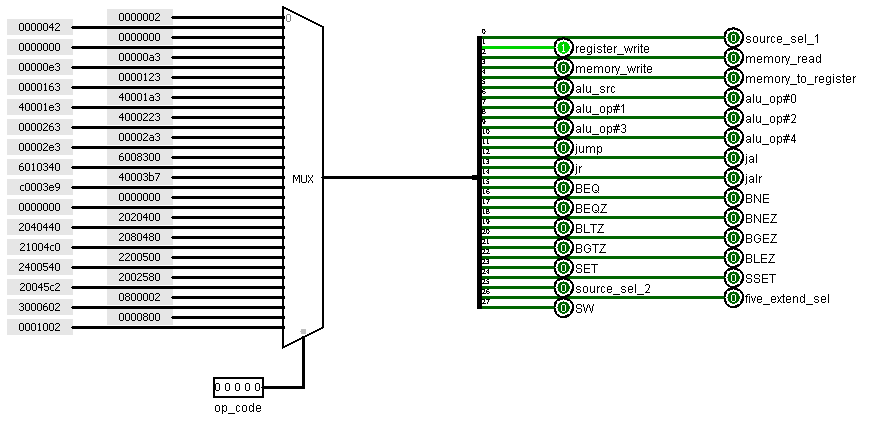
1. In the above table we treat each row (each instruction = each operation code) of the table as a binary format and we converted it into a unique hexadecimal code as shown in the following diagram:

**0x0000002**

Note all don’t cares are considered zeros.

Note this diagram related to the instructions whose Op.code = zero only and the other instruction would be calculated in the following table:

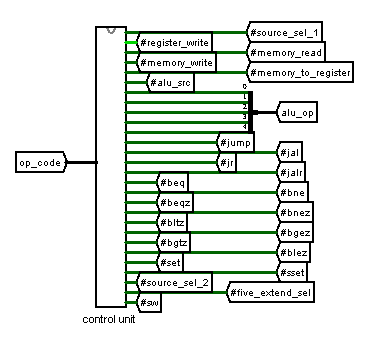


1. ****We use a 32x1 multiplexer with the instruction operation code as a selector to set up the main control unit as shown below:

The inputs of the multiplexer are the hexadecimal codes which have been described in step (2).

The output of the multiplexer is connected with a splitter to connect each control signal with its position (weigh) in the truth table that belongs to the control unit.

1. The circuit appearance after design:



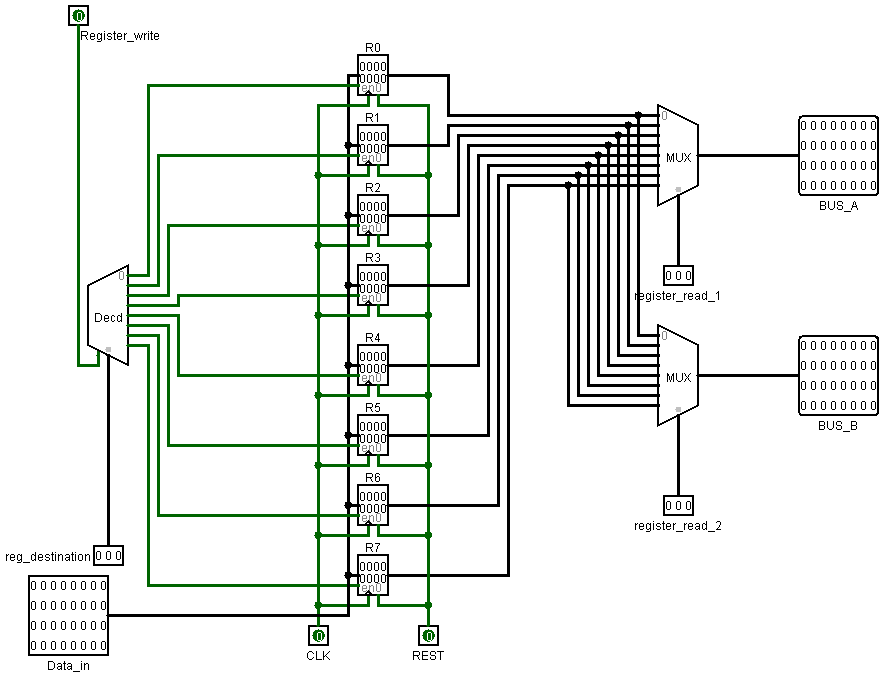
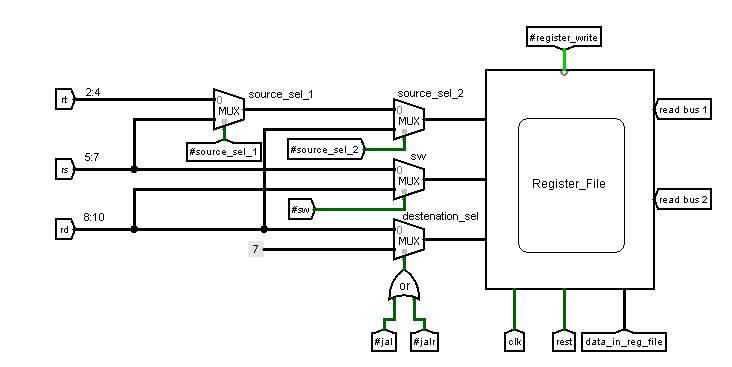
* **Register file circuit design:**

The register file is a means of memory storage within a computer's central processing unit (CPU). We used it to store data after executing the instruction.

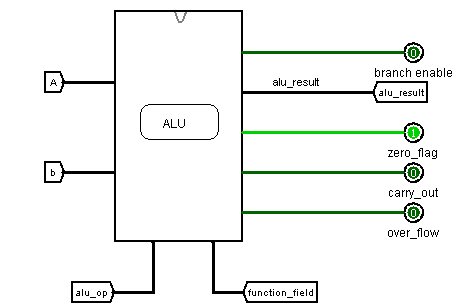
The register file exists in this processor consists of eight general purpose registers each of them 32 bits data storage working at the rising edge of the clock.

* **The inputs to the register file:**
* Data that will be written in the register file.
* register write control signal that comes from the control unit
* Clock Pin
* Rest pin
* RT …Three address bits for the first source (bits 2:4 of the instruction)
* RS …Three address bits for the second source (bits 5:7 of the instruction)
* RD…Three address bits for the destination register (bits 8:10 of the instruction)
* **The inputs of the register file:**
* 32 bits for read bus 1
* 32 bits for read bus 2
* **The design procedures of the register file:**

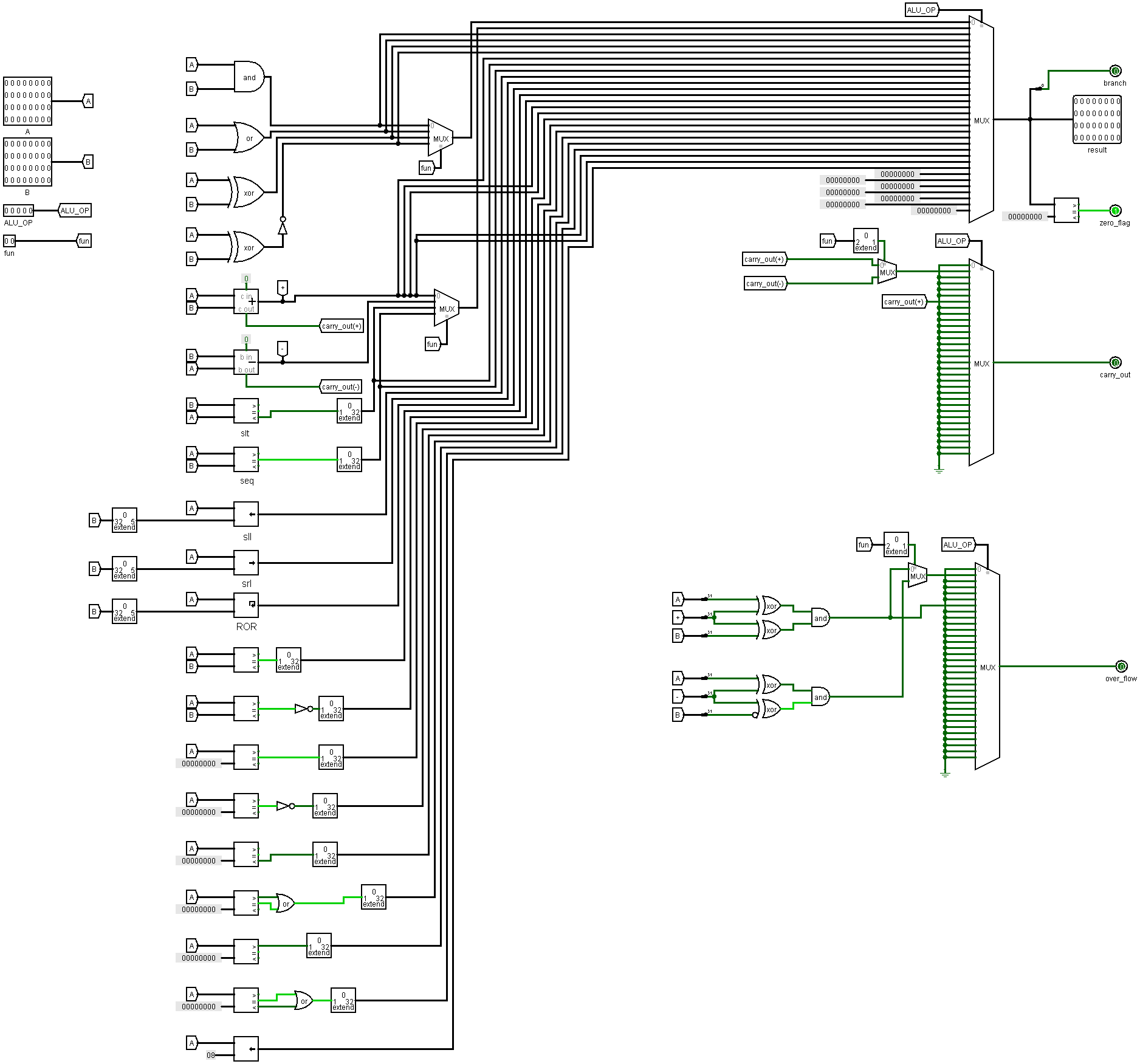
1. Specify the number and the size of the registers.
2. Give each of these register a unique label or number that will represent its address.
3. Make all registers synchronized with each other by connecting them with the same clock pin.
4. all registers’ clear pins must be connect with each to form the rest pin (when this pin is active all the registers will be zeros at the same time )
5. we drive the date that results of the write back stage to the all registers and to select which register we will write in we use a decoder with three address bits for the destination register (bits 8:10 of the instruction) as selectors of the decoder whose enable pin is the register write control signal that comes from the control unit.
6. The output data of these registers are driven to two multiplexers.
7. For the upper multiplexer: the selectors are the address of the register that we will take out its contents on bus 1 of the register file.
8. For the lower multiplexer: the selectors are the address of the register that we will take out its contents on bus 2 of the register file.

* **The register file as a sub circuit.**
* **The circuit appearance of the register file.**
* **The arithmetical and logical unit design**

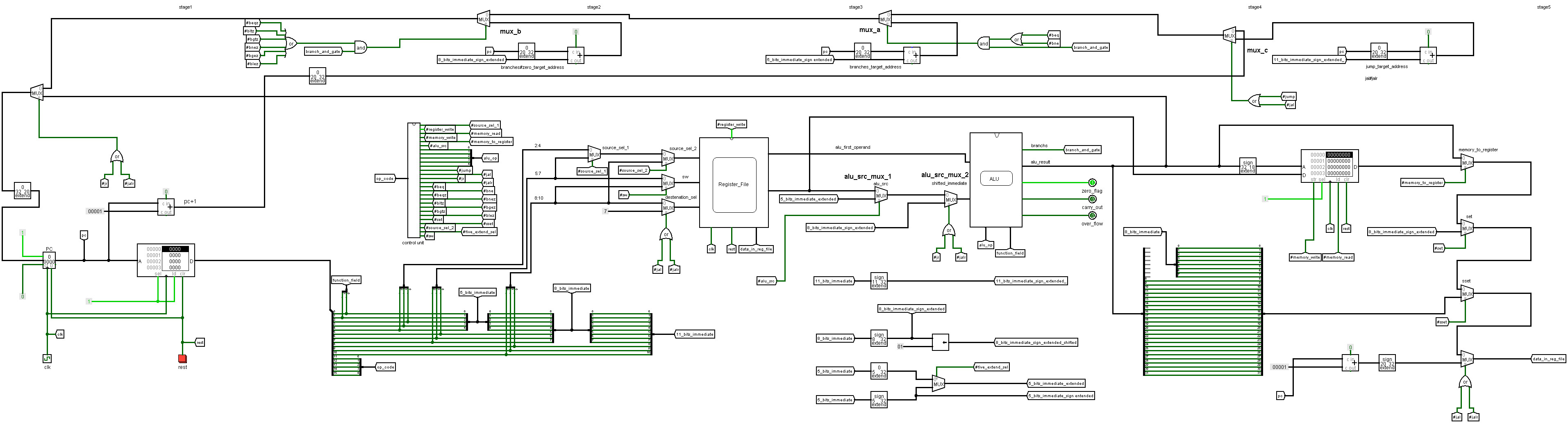
The arithmetic logic unit (ALU) is a combinational circuit that performs arithmetic and bitwise operations on integer binary numbers.

* **The inputs of (ALU)**
* ALU\_OP
* Function field for R\_type instructions
* First operand
* Second operand
* **The inputs of (ALU)**
* ALU result
* Overflow pin
* Carry out pin
* Zero flag
* Branches enable
* **The operation that the (ALU) circuit supports**
* bitwise operations such ( OR , AND, XOR ,EQV)
* Shifting operations such (SLL,ROR ,SRL)
* Arithmetical operations such (ADD,SUB)
* Other operations based on comparisons (SLT,SEQ)
* Comparative operations for branch instructions.
* **The design procedures of the (ALU):**

1. We specify the operations required that that the ALU circuit have to perform.
2. Make a special combinational circuit for each operation.
3. Use a Multiplexer to set up the (ALU), where the selectors of this Multiplexer will be the (ALU) operation code bit that come from the main control unit, and the inputs of this multiplexer are the results of the combinational circuits we have been built under the constraint that “each combinational circuit’s output should enter in the crossponding location to its (ALU) operation code.
4. We use two multiplexers to differ between the instructions (AND – OR – XOR - EQV -ADD – SUB – SLT – SEQ) that have the same instruction operation code (same (ALU) operation code). The selectors of these multiplexers will be the function field bits (0:1 of the instruction), and the output of these multiplexers enter the final result multiplexer.



Note: The zero flag pin is active when the result of the (ALU) is equal to zero

* **The overall data path of the single cycle processor.**

We can divide the overall datapath into four parts to make description clear:

* **First part (next pc).**

This stage consists of:

* 20 bits Program counter point to the address of the instruction will be fetched
* 16 bits instruction memory(word addressable) holds all instruction that will be executed, its maximum capacity is 220 words
* Next pc could be:
* Normal mode pc+1.
* Branch (BEQ, BNE) target address (pc+ 5 bits sign extended immediate value) restricted by **mux\_a**, its selector is an AND\_gate with inputs (branch enable pin anded with branches control signals (BEQ, BNE in an OR\_gate).
* Zeroes Branches (BEQZ, BNEZ, BLTZ, BGEZ, BGTZ, BLEZ) target address (pc +8 bits sign extended immediate value) restricted by **mux\_b**, its selector is an AND\_gate whose inputs are (branch enable pin anded with branches control signals (BEQZ, BNEZ, BLTZ, BGEZ, BGTZ, BLEZ in an OR\_gate).
* Jump and JAL target address (pc + 11 bits sign extend immediate) restricted by **mux\_c** with (jump control signal pin ored with JAL control signal through OR\_GATE) as its selector.
* JR and JALR target address (pc + 8 bits sign extend immediate) restricted by **mux\_d** with (JR control signal pin ored with JALR control signal through OR\_GATE) as its selector.
* **Second part (instruction decoding)**

IN this part the fetched instruction is decoded to four format:

* R\_type format:
* 5-bits Op.code (Op) (11:15)…………..to control unit.
* 3-bits destination register Rd (8:10)…..to the register file.
* 3-bits source registers Rs (5:7)………..to the register file.
* 3-bits source registers RT (2:4)……......to the register file.
* 2-bits function field (0:1)…………..….to (ALU).
* I\_type format:
* 5-bits Op.code (Op) (11:15)………...….to control unit.
* 3-bits destination register Rd (8:10)……to the register file.
* 3-bits source register Rs (5:7)…………..to the register file.
* 5-bits immediate (0:4)…………………..to (BEQ-BNE) target address adder.
* 5-bits immediate (0:4)…………………..to (LW-SW) ALU adder.
* B\_type format:
* 5-bits Op.code (Op) (11:15)…..………….to control unit.
* 3-bits register number Rd (8:10)…………to the register file.
* 8-bits Immediate (0:7).to (BEQZ, BNEZ, BLTZ, BGEZ, BGTZ, BLEZ) target address adder.
* 8-bits Immediate (0:7).to (JR-JALR). ALU target address adder
* J\_type format:
* 5-bits (11:10) Op.code (Op)…………..….to control unit
* 11-bits immediate (0:10)….… to (J-JAL). ALU target address adder.
* **Third part (instruction execution)**

In this part Alu perform operations on each instruction’s operands which selected by two multiplexers (ALU\_SRC\_MUX\_1and ALU\_SRC\_MUX\_2).

* ALU\_SRC\_MUX\_1:

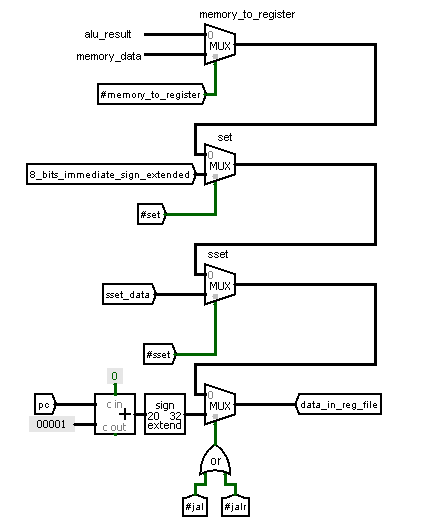
Selects between five bits extend (I\_TYPE INSTRUCTIONS) and bus b of the register file.

* ALU\_SRC\_MUX\_2:

Selects between eight bits extend (SET-SSET-JALR-JR- BEQZ- BNEZ- BLTZ-BGEZ- BGTZ- BLEZ) and ALU\_SRC\_MUX\_1’s output.

* **Fourth part (Data memory)**
* This part related with instructions that deals with data memory (word addressable) load word and store word only.
* Load word instruction loads data from memory to register file.
* Store word instruction loads data from register file to memory.
* The accessed addresses by these instruction are calculated in the arithmetic and logical unit.
* **Fourth part (write back in the register file)**

In this part we select which data will be written in the register file through four multiplexers:

* Memory to register mux differs between data memory output and Alu results , its selector is the memory to register pin from control unit
* Set mux activates set instruction data in the register file.its selector is the set pin from control unit.
* Sset mux activates sset instruction data in the register file.its selector is the sset pin from control unit.
* Jal and jalr mux to save the return address in register R7.its selector is the JAL control signal pin ored with JALR control signal through an OR\_GATE.
* **phase (2) Design and implementing of a Pipeline MIPS processor ⮘**

In the single cycle processor the CPI equal one that means “the processor performs only one instruction each clock cycle”. The main problem of this technique is that there are many parts and devices may be stand free for long time period and this affects the performance of the processor.

We can exploit these circuits by using another technique which is called “pipelining “.

This technique improves the performance of the processor.

The main difference between the single cycle processor and the Pipelined processor is that the instructions are overlapped through the different stages in the datapath.

The only major advantage is performance improvement. By pipelining instructions, we can pump in more instructions in the processor and then we get a significant improvement in processor speed. This happens because we can execute parts of instructions in parallel to parts of other instructions.

For long sequences of instructions, this technique is extremely effective because in every step the processors different parts busy with an instruction.

The only major disadvantage is complicity process in designing the processor.

All instructions must enter sequence of five stages each of them needs one clock cycle so each instruction is completely performed after five clock cycles.

* **The pipelining registers**

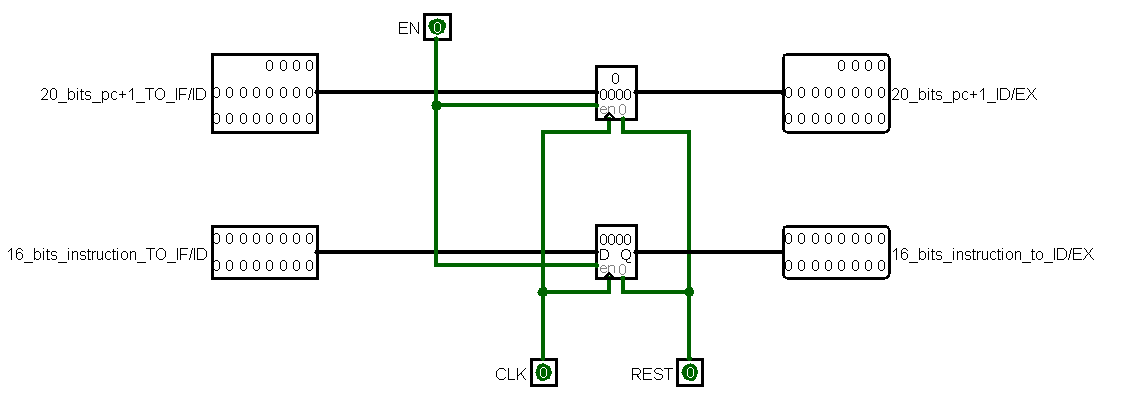
In this technique we don’t directly deal with the pins and wires (the output of the different circuits in different stages are not connect to each other) such as in the single cycle processor but we deal with the pipelining registers’ outputs and inputs. The main reason for this is that we need locations to hold the output data of the different stages combinational circuit to use it in the next stage until the instruction is completely performed.

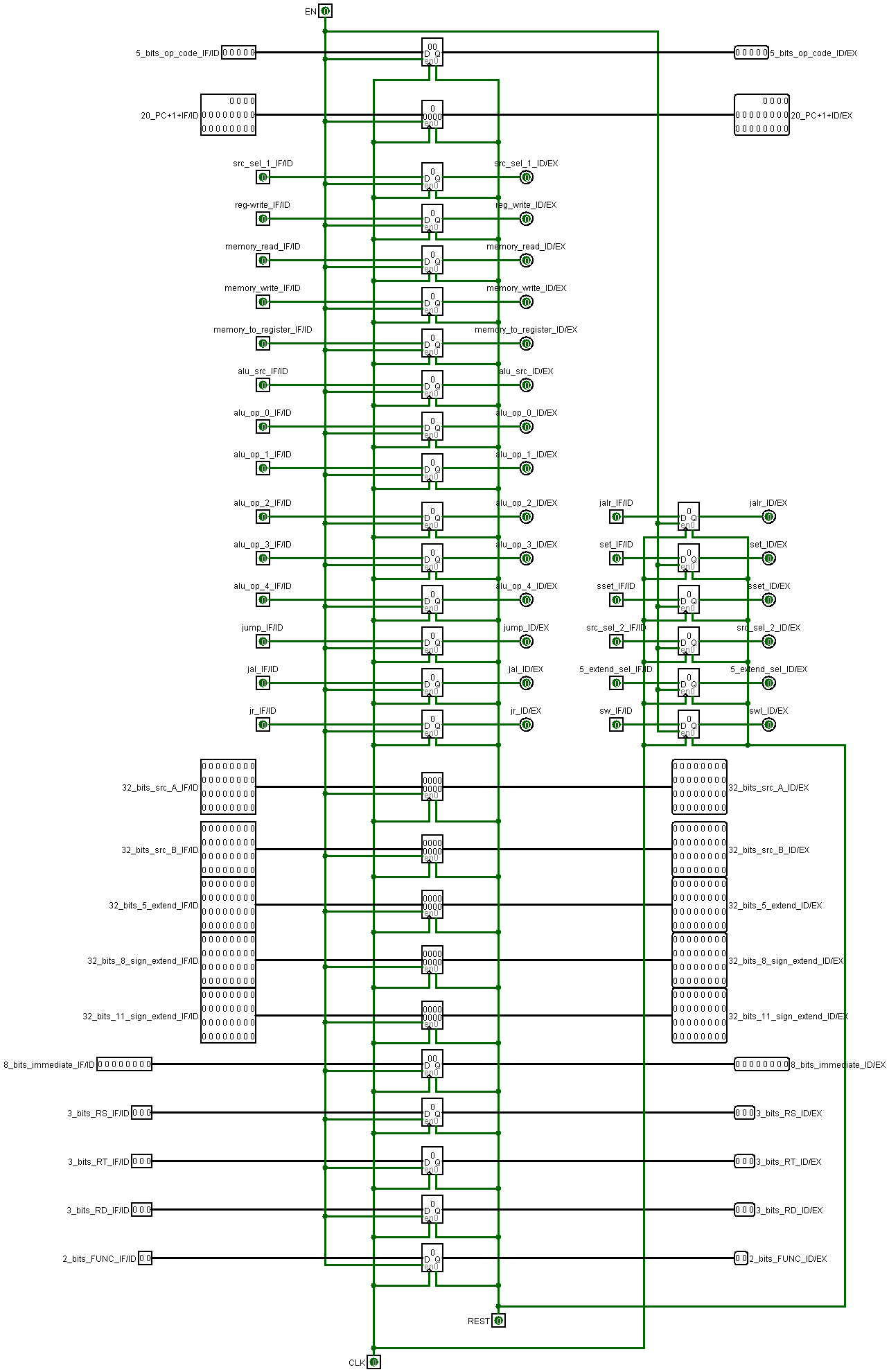
These registers are:

* IF/ID.
* ID/EX.
* EX/MEM.
* MEM/WB

1. **IF/ID.**

**Contents:**

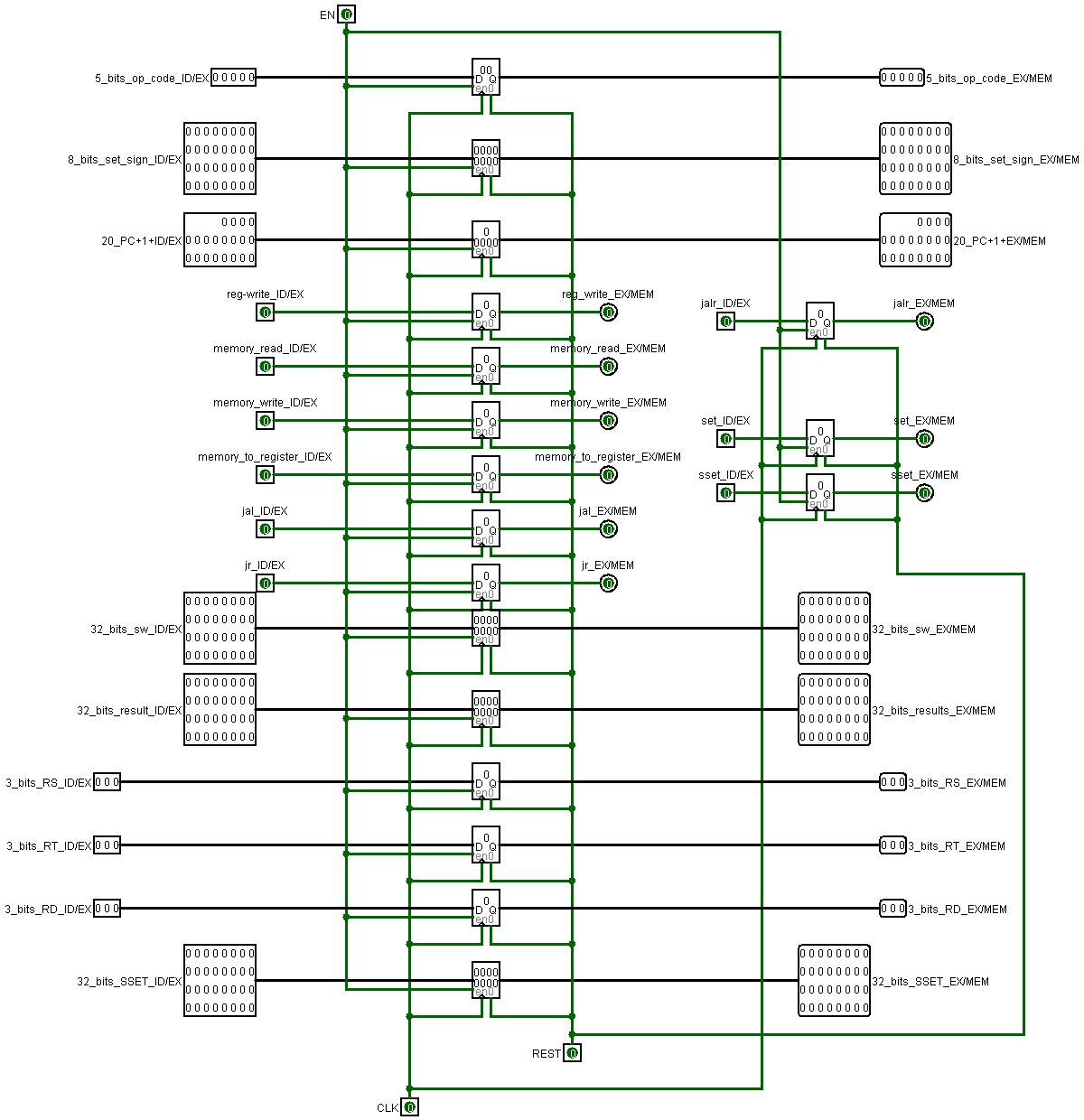
* Program Counter’s contents…………for (BEQZ , BNEZ , BLTZ , BGEZ , BGTZ , BLEZ,BEQ,BNE,JUMP,JAL,JR,JALR) instructions
* 16 bit instruction



1. **ID/EXE.**

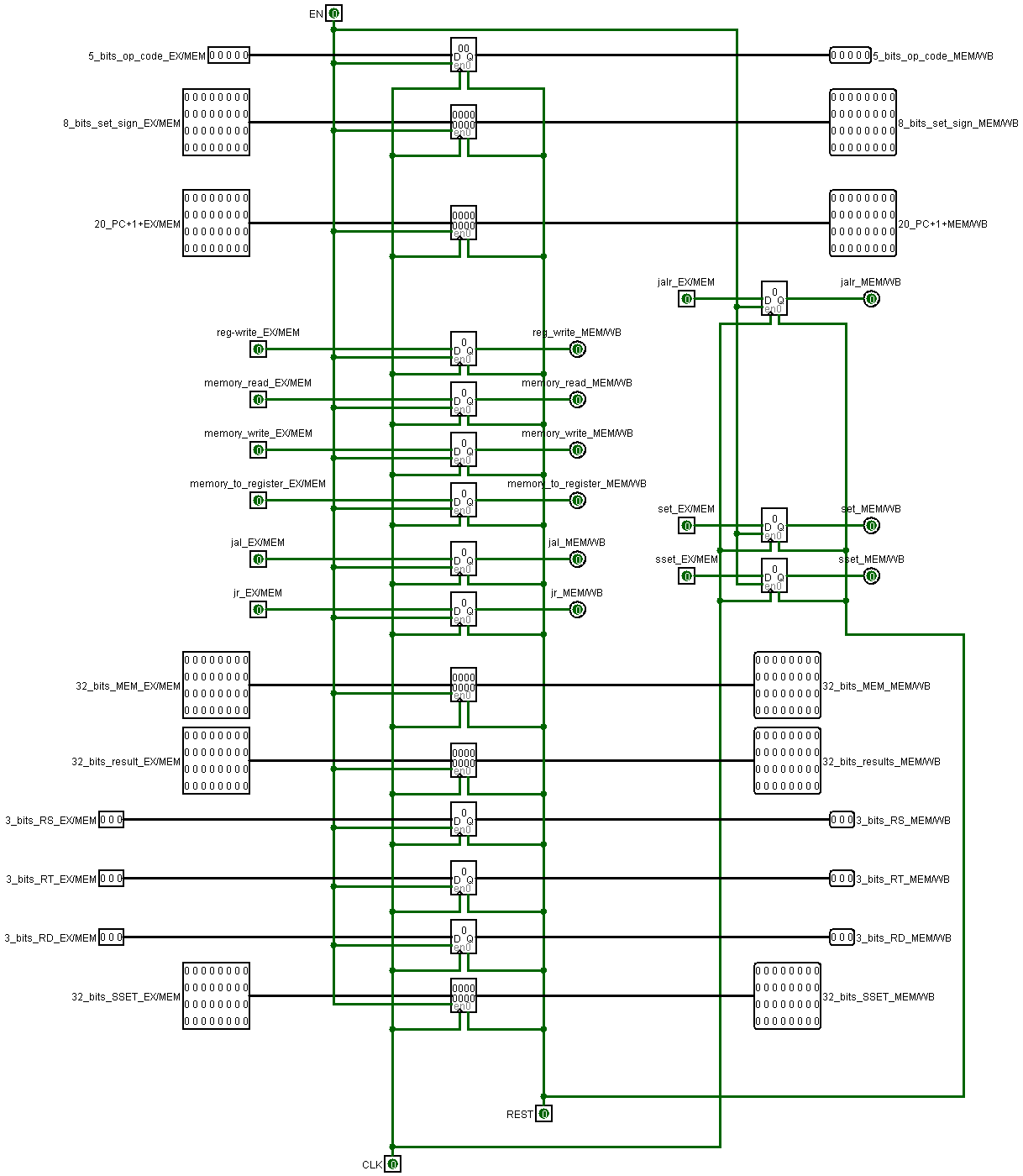
**Contents:**

* PC+1
* Op code
* Control signals
* Signed extend 5
* Signed extend 8
* Signed extend 11
* Function field
* Number of used “Rs, Rt ,Rd”
* Bus 1,2

1. **EXE/MEM.**

**Contents:**

* PC+1
* Op code
* Control signals
* ALU result
* sset result
* set result
* SW data
* Number of used “Rs, Rt ,Rd”



1. **MEM/WB.**

**Contents:**

* PC+1
* Op code
* Control signals
* ALU result
* sset result
* set result
* SW data
* Number of used “Rs, Rt ,Rd”
* **Pipeline Hazards**

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types:

**Structural hazard:** When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

**Data hazard:** When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

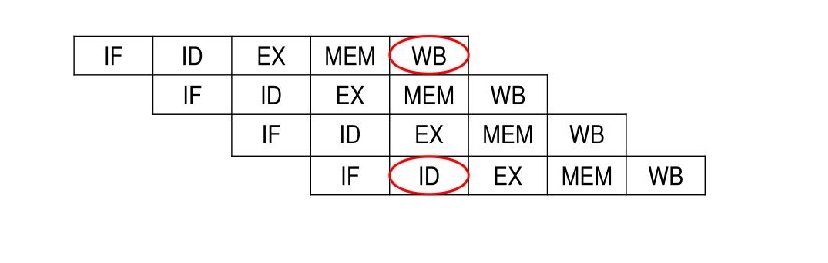
**Control hazard:** When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not then one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

We can solve these hazards using three techniques:

* Add more hardware.
* Dividing clock principle.
* Stalling instructions.
* Forwarding data (by passing).

Adding more hardware is not used here because Structural hazards are limited in our processor.

* **Dividing clock principle**

For example if we fall in a situation like this:

LW $r1, 4($r2)

ADD $r5, $r6, $r4

SUB $r3, $r7, $r0

SUB $r3, $r1, $r0

**Solution:**

Always Write to register ($r1) during first half of clock cycle.

Always Read ($r1) from Register file during second half of clock cycle.

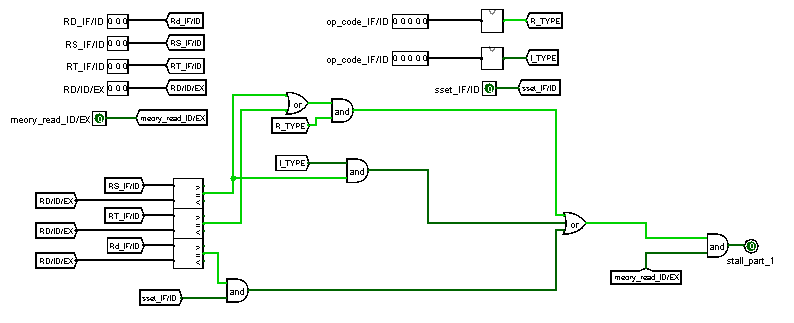
* **Hazard detection Unit:**

We have designed three parts for this unit:

* Hazard unit part 1:
* Hazard unit part 2:
* Hazard unit part 3:
* **Hazard unit part 1:**

This part handles the data hazards results from load word instruction such this case:

LW $r1, 4($r2)

ADD $r5, $r1, $r4

**Inputs:**

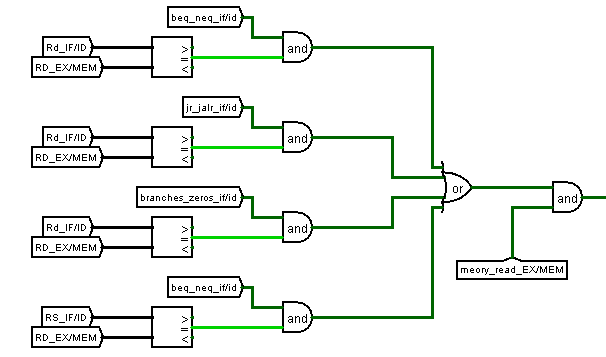
* Register RS (IF/ID)
* Register RT (IF/ID)
* Register RD (IF/ID)
* Register RD (ID/EXE)
* MEM/READ (ID/EX)
* Op.code (IF/ID)

**Outputs:**

* stall pin

**Conditions:**

* Memory read ID/EXE = 1 in addition to one of the following:
* IF/ID Register Rs =ID/EXE register Rd ………..for R\_type and I\_type instructions
* IF/ID Register Rt = ID/EXE register Rd………...for R\_type
* IF/ID Register Rd= ID/EXE register Rd ………..for sset instruction
* **Hazard unit part 2:**

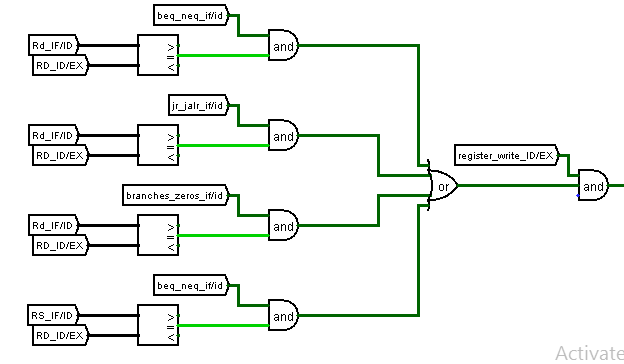
****This part handles the data hazards related with branches, JR AND JALR instructions like this case:

LW $r1, 4($r2)

ADD $r1, $r5, $r4

BEQ $r1, $r6, 4

**Inputs:**

* Register RS (IF/ID)
* Register RT (IF/ID)
* Register RD (IF/ID)
* Register RD (ID/EXE)
* Register RD (EXE/MEM)
* MEM/READ (ID/EX)
* MEM/READ (EXE/MEM)
* Op.code (IF/ID)
* Op.code (ID/EX)
* SET/SSET/REG\_WRITE(ID/EX)

**Outputs:**

* stall pin

**Conditions:**

* Memory read EXE/MEM = 1 and
* IF/ID Register Rs = EXE/ME register Rd
* IF/ID Register Rs = EXE/ME register Rd
* IF/ID Register Rd = EXE/ME register Rd
* Register write ID/EXE =1 and
* IF/ID Register Rs = EXE/ME register Rd
* IF/ID Register Rs = ID/EXE register Rd
* IF/ID Register Rd =ID/EXE register Rd
* **Hazard unit part 3:**

This part handles the data hazards related with the store word instruction such this case:

ADD $r5, $r1, $r4

SW $r5, 4($r7)

SW $r5, 4($r3)

**Inputs:**

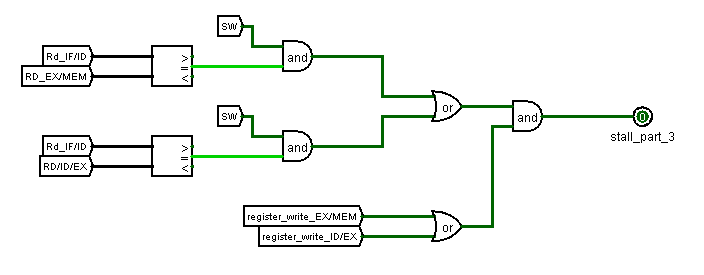
* Register RD (IF/ID)
* Register RD (ID/EXE)
* Register RD (EXE/MEM)
* Op.code (IF/ID)
* REG\_WRITE(EXE/MEM)
* REG\_WRITE(ID/EX)

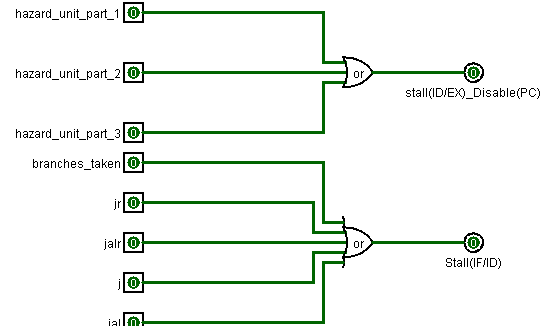
**Outputs:**

* stall pin

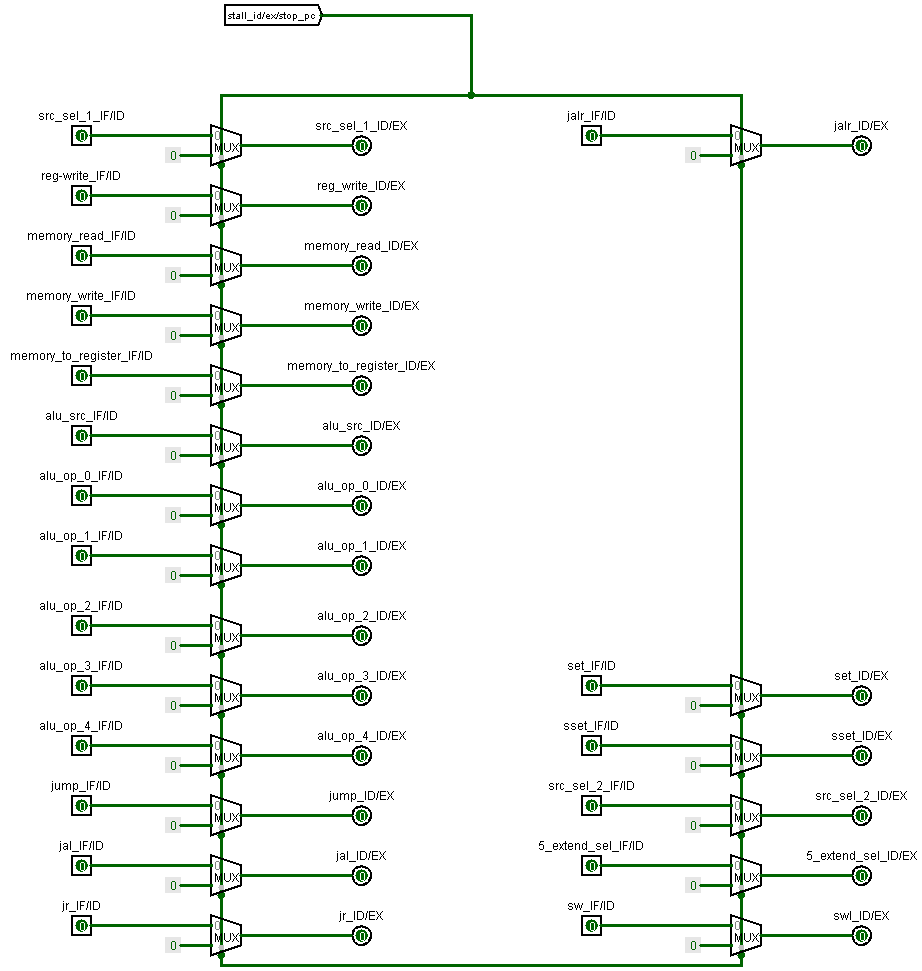
**Conditions:**

* Register write EXE/MEM = 1 or Register write ID/EXE = 1 and
* SW signal = 1
* IF/ID Register Rd=EXE/ME register Rd
* IF/ID Register Rd=ID/EXE register Rd



* **The main Hazard unit:**

This circuit collect the three parts to form two bits reasonable for stalling the ID/EX, flushing instruction in IF/ID and disable the program counter register.

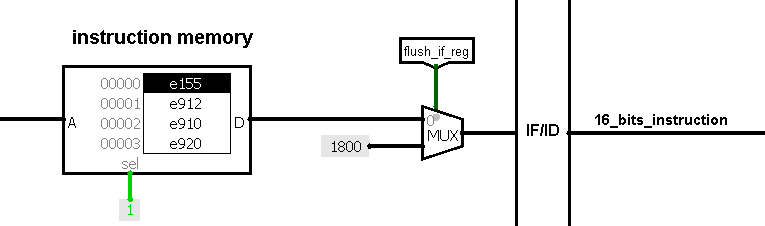
**Stalling the ID/EX:**

We can insert “bubbles” through ID/EX by passing zeros instead of the instruction’s control signals that are responsible for changing the state of the processor’s register file or PC.

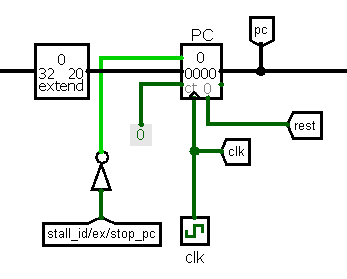
In order to do that we make the stall (ID/EX) pin a selector to a multiplexer whose inputs are the control signals (location zero) and zeros at (location one) the outputs of the multiplexer go as inputs for the (ID/EX) registers as shown in the figure.

**Flushing instruction in IF/ID**

For the taken branches and jump instructions:

They calculate their target address in the decoding stage and at that moment the processor is fetching a new instruction (its address = pc+1) but this instruction leads into wrong results so we need to replace this instruction with any instruction whose operation code is one of these (2 - 3-18-19) as they are not used in this processor (their control signals equal zero)

We can do this by using a multiplexer whose inputs are 1800 (Op.code equals 3) at location one and the 16\_bits instruction at location zero. The selector of the mux is the (IF/ID) stall pin

**Disable the program counter register:**

In order to do that we path the stall (ID/EX) pin through a converter and connect it with the enable pin of the program counter register to keep the current instruction in the decoding stage by fetching it again as shown in the figure.

* **Execution Forwarding Unit:**

This unit is design to handles the data hazards (read after write data dependency) in the execution stage.

This unit consists of two parts:

* **Forwarding unit A**
* **The reason for forwarding**

An instruction in the execution stage needs data (dependency) from an instruction in the memory access stage (EXE/MEM register) or write back stage (MEM/WB register)

* **The Forwarding locations :**
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from EX/MEM pipeline register (Alu results).
* Data that belongs to set instruction from EX/MEM pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from EX/MEM pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from EX/MEM pipeline register (pc+1).
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from MEM/WB pipeline register (Alu results).
* Data that belongs to set instruction from MEM/WB pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from MEM/WB pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from MEM/WB pipeline register (pc+1).
* **The Forwarding conditions:**

**The forwarding from (EXE/MEM) register (ALU RESULTS) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,R\_type or I\_type in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt ID/EXE & R\_type in the execution stage).
* Rd EXE/MEM = (Rs ID/EXE & I\_type in the execution stage).
* Rd EXE/MEM = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from EXE/MEM register (SET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SET instruction in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt ID/EXE & R\_type in the execution stage).
* Rd EXE/MEM = (Rs ID/EXE & I\_type in the execution stage).
* Rd EXE/MEM = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from EXE/MEM register (SSET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SSET instruction in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt ID/EXE & R\_type in the execution stage).
* Rd EXE/MEM = (Rs ID/EXE & I\_type in the execution stage).
* Rd EXE/MEM = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from EXE/MEM register (JAL/JALR DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) , JAL/JALR instruction in the memory access stage and one of the following conditions:
* Rd (R7) EXE/MEM = (Rt (R7) ID/EXE & R\_type in the execution stage).
* Rd (R7) EXE/MEM = (Rs (R7) ID/EXE & I\_type in the execution stage).
* Rd (R7) EXE/MEM = (Rd (R7) ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from (MEM/WB) register (ALU RESULTS) conditions:**

* Register write signal in the MEM/WB register equals (1) ,R\_type or I\_type instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rt ID/EXE & R\_type in the execution stage).
* Rd MEM/WB = (Rs ID/EXE & I\_type in the execution stage).
* Rd MEM/WB = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from (MEM/WB) register (SSET data) conditions:**

* Register write signal in the MEM/WB register equals (1) ,SSET instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rt ID/EXE & R\_type in the execution stage).
* Rd MEM/WB = (Rs ID/EXE & I\_type in the execution stage).
* Rd MEM/WB = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

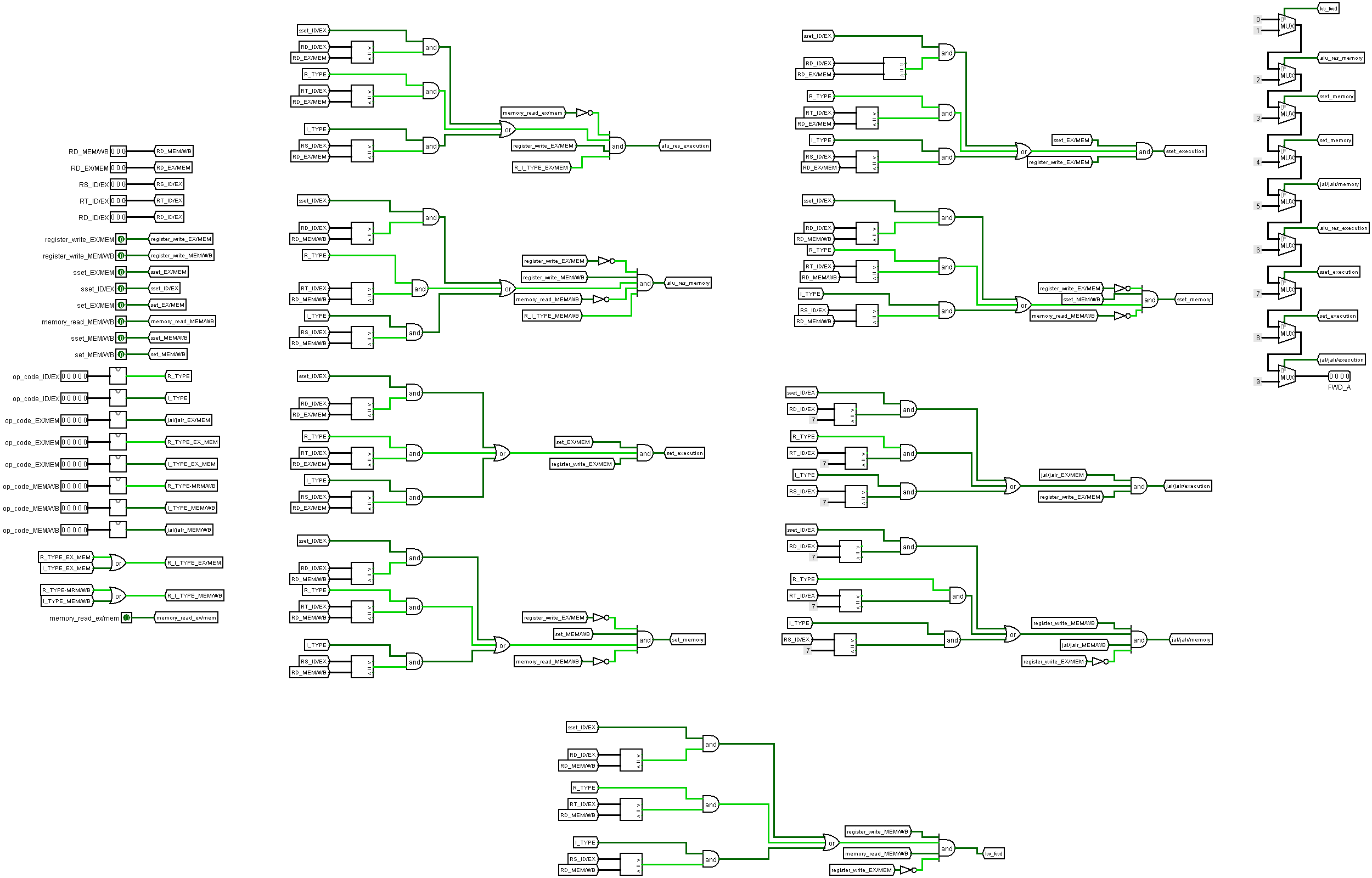
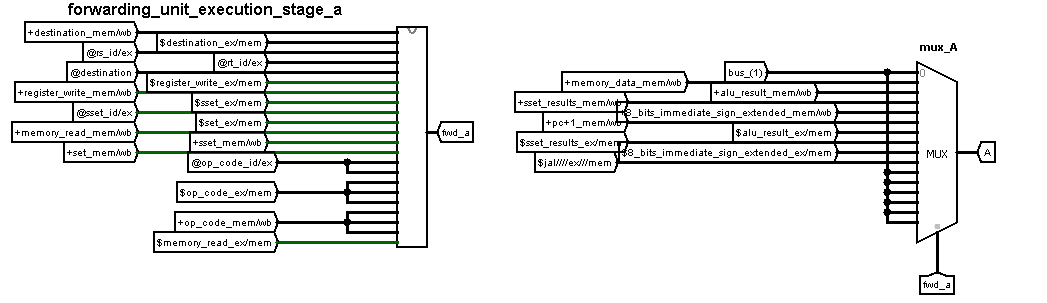
**The forwarding from (MEM/WB) register (SET data) conditions:**

* Register write signal in the MEM/WB register equals (1) ,SET instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rt ID/EXE & R\_type in the execution stage).
* Rd MEM/WB = (Rs ID/EXE & I\_type in the execution stage).
* Rd MEM/WB = (Rd ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from (MEM/WB) register (JAL/JALR DATA) conditions:**

* Register write signal in the MEM/WB register equals (1) ,JAL/JALR instruction in the write back stage and one of the following conditions:
* Rd (R7) MEM/WB = (Rt (R7) ID/EXE & R\_type in the execution stage).
* Rd (R7) MEM/WB = (Rs (R7) ID/EXE & I\_type in the execution stage).
* Rd (R7) MEM/WB = (Rd (R7) ID/EXE & SSET signal equals (1) in the execution stage).

**The forwarding from (MEM/WB) register (DATA MEMORY) conditions:**

* Register write signal in the MEM/WB register equals (1) ,memory read instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rt ID/EXE & R\_type in the execution stage).
* Rd MEM/WB = (Rs ID/EXE & I\_type in the execution stage).
* Rd MEM/WB = (Rd ID/EXE & SSET signal equals (1) in the execution stage).
* **Forwarding unit inputs:**
* (RD / ID-EX)
* (RT / ID-EX)
* (RS / ID-EX)
* (RD / EX-MEM)
* The op code and the control signals.
* **Forwarding unit outputs:**
* Four bits as a selector (A) for the mux (A) in the datapath.
* **Forwarding unit B**
* **The reason for forwarding**

R\_TYPE instruction in the execution stage needs data (dependency) from an instruction in the memory access stage (EXE/MEM register) or write back stage (MEM/WB register)

* **The Forwarding locations :**
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from EX/MEM pipeline register (Alu results).
* Data that belongs to set instruction from EX/MEM pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from EX/MEM pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from EX/MEM pipeline register (pc+1).
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from MEM/WB pipeline register (Alu results).
* Data that belongs to set instruction from MEM/WB pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from MEM/WB pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from MEM/WB pipeline register (pc+1).
* Data that belongs to load word instructions from MEM/WB pipeline register.
* **The Forwarding conditions:**

**The forwarding from (EXE/MEM) register (ALU RESULTS) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,R\_type or I\_type in the memory access stage and the following conditions:
* Rd EXE/MEM = (RS ID/EXE & R\_type in the execution stage).

**The forwarding from EXE/MEM register (SET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SET instruction in the memory access stage and the following conditions:
* Rd EXE/MEM = (RS ID/EXE & R\_type in the execution stage).
* **The forwarding from EXE/MEM register (SSET DATA) conditions:**
* Register write signal in the EXE/MEM register equals (1) ,SSET instruction in the memory access stage and the following conditions:
* Rd EXE/MEM = (RS ID/EXE & R\_type in the execution stage).
* **The forwarding from EXE/MEM register (JAL/JALR DATA) conditions:**
* Register write signal in the EXE/MEM register equals (1) , JAL/JALR instruction in the memory access stage and one of the following conditions:
* Rd (**R7**) EXE/MEM = (Rs (**R7**) ID/EXE & R\_type in the execution stage).

**The forwarding from (MEM/WB) register (ALU RESULTS) conditions:**

* Register write signal in the MEM/WB register equals (1) ,R\_type or I\_type instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rs ID/EXE & R\_type in the execution stage).

**The forwarding from (MEM/WB) register (SSET data) conditions:**

* Register write signal in the MEM/WB register equals (1) ,SSET instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rs ID/EXE & R\_type in the execution stage).

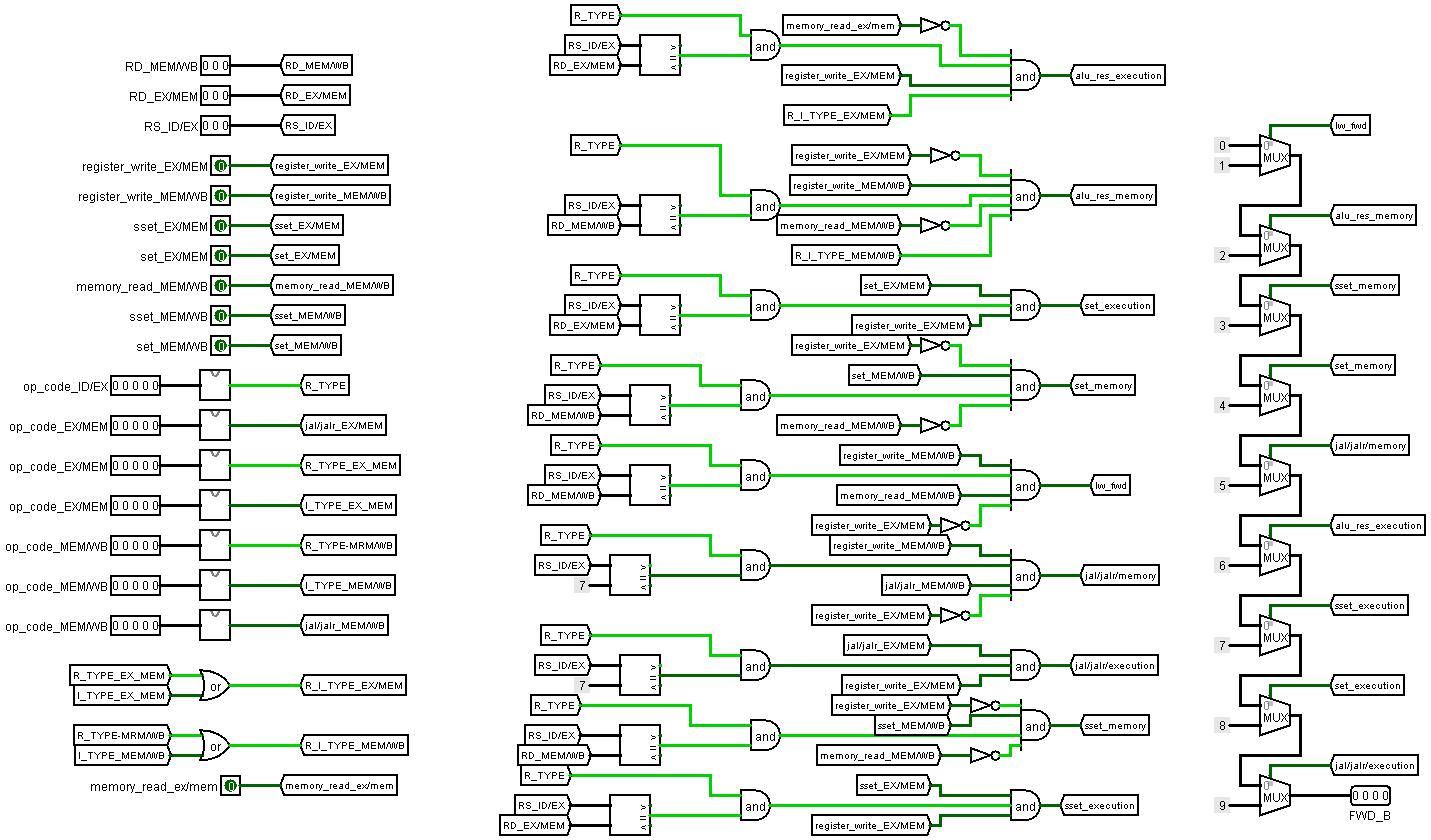
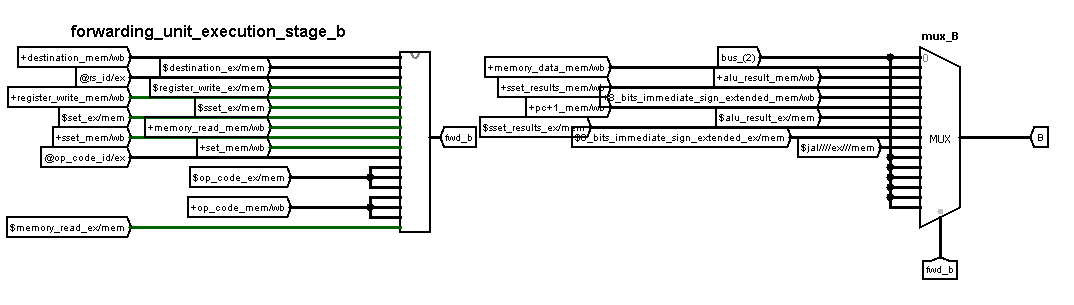
**The forwarding from (MEM/WB) register (SET data) conditions:**

* Register write signal in the MEM/WB register equals (1) ,SET instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rs ID/EXE & R\_type in the execution stage).

**The forwarding from (MEM/WB) register (JAL/JALR DATA) conditions:**

* Register write signal in the MEM/WB register equals (1) ,JAL/JALR instruction in the write back stage and one of the following conditions:
* Rd (**R7**) MEM/WB = (Rs (**R7**) ID/EXE & R\_type in the execution stage).

**The forwarding from (MEM/WB) register (DATA MEMORY) conditions:**

* Register write signal in the MEM/WB register equals (1) ,memory read instruction in the write back stage and one of the following conditions:
* Rd MEM/WB = (Rs ID/EXE & R\_type in the execution stage).
* **Forwarding unit inputs:**
* (RS / ID-EX)
* (RD / EX-MEM)
* The op code and the control signals.
* **Forwarding unit outputs:**
* Four bits as a selector (B) for the mux (B) in the datapath
* **Decoding Forwarding Unit:**

This unit is design to handles the data hazards (read after write data dependency) in the decoding stage.

This unit consists of two parts:

* **Forwarding unit C**
* **The reason for forwarding**

An instruction in the decoding stage needs data (dependency) from an instruction in the memory access stage (EXE/MEM register).

* **The Forwarding locations :**
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from EX/MEM pipeline register (Alu results).
* Data that belongs to set instruction from EX/MEM pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from EX/MEM pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from EX/MEM pipeline register (pc+1).
* **The Forwarding conditions:**

**The forwarding from (EXE/MEM) register (ALU RESULTS) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,R\_type or I\_type in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt IF/ID & BEQ/BNE in the decoding stage).
* Rd EXE/MEM = (Rd IF/ID & JR/JALR/ZERO BREZNCHES in the decoding stage).

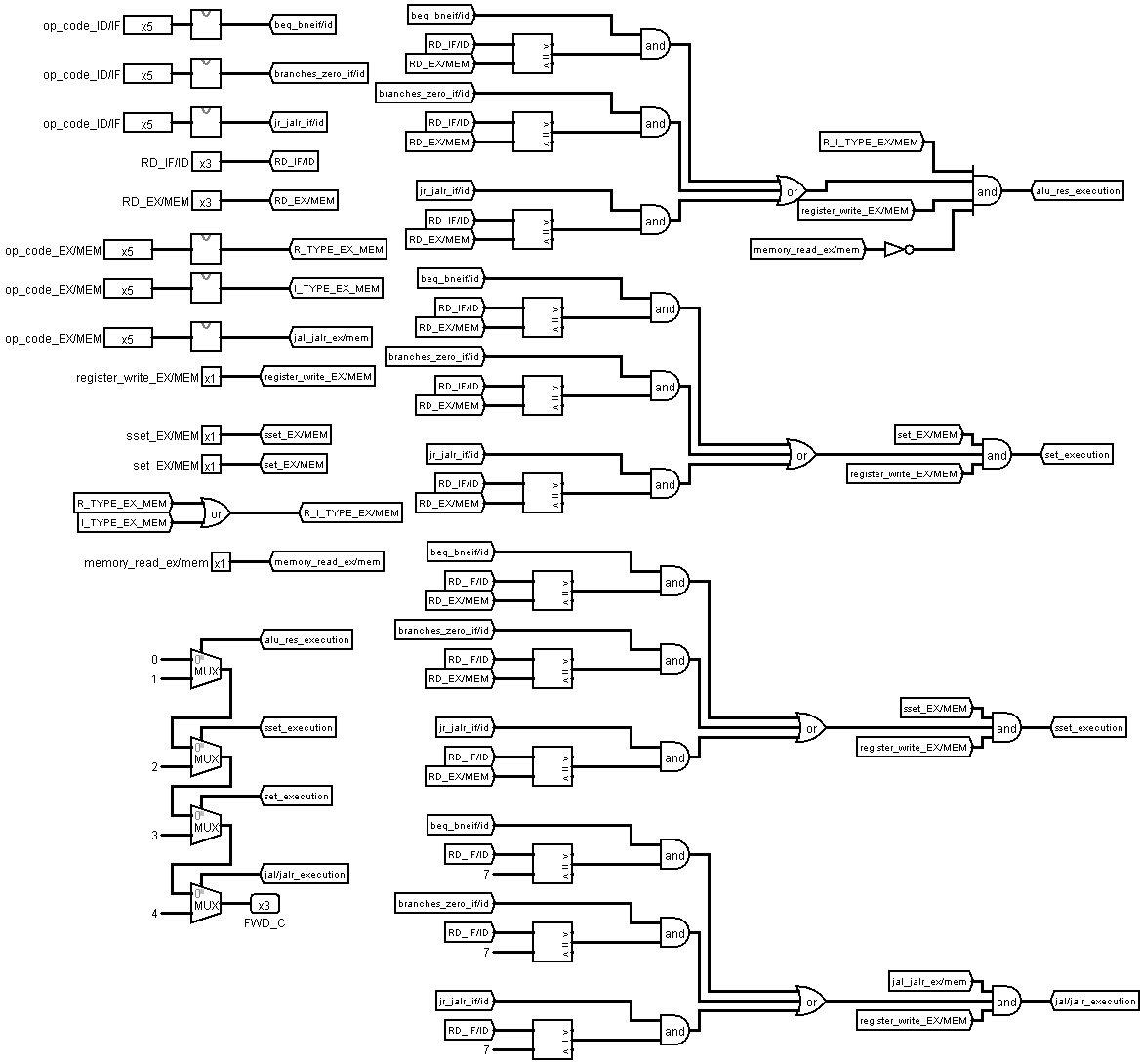
**The forwarding from EXE/MEM register (SET DATA) conditions:**

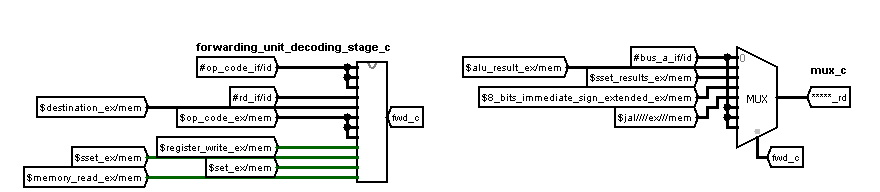
* Register write signal in the EXE/MEM register equals (1) ,SET instruction in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt IF/ID & BEQ/BNE in the decoding stage).
* Rd EXE/MEM = (Rd IF/ID & JR/JALR/ZERO BREZNCHES in the decoding stage).

**The forwarding from EXE/MEM register (SSET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SSET instruction in the memory access stage and one of the following conditions:
* Rd EXE/MEM = (Rt IF/ID & BEQ/BNE in the decoding stage).
* Rd EXE/MEM = (Rd IF/ID & JR/JALR/ZERO BREZNCHES in the decoding stage).

**The forwarding from EXE/MEM register (JAL/JALR DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) , JAL/JALR instruction in the memory access stage and one of the following conditions:
* Rd (R7) EXE/MEM = (Rt (R7) ID/EXE & BEQ/BNE in the decoding stage).
* Rd (R7) EXE/MEM = (Rd (R7) IF/ID & JR/JALR/ZERO BREZNCHES in the decoding stage).

* **Forwarding unit inputs:**
* (RD / IF/ID)
* (RS / IF/ID)
* (RD / EX-MEM)
* The op code and the control signals.
* **Forwarding unit outputs:**
* Four bits as a selector (C) for the mux (C) in the datapath.
* **Forwarding unit D**
* **The reason for forwarding**

BEQ/BNE instruction in the execution stage needs data (dependency) from an instruction in the memory access stage (EXE/MEM register) or write back stage (MEM/WB register)

* **The Forwarding locations :**
* Data that belongs to R\_type and I\_type except for instructions LW, SW, BEQ, BNE instructions from EX/MEM pipeline register (Alu results).
* Data that belongs to set instruction from EX/MEM pipeline register (8 bits sign extended).
* Data that belongs to sset instruction from EX/MEM pipeline register ({Rd [23:0] <<8, Imm8}).
* Data that belongs to Jal and jalr instructions from EX/MEM pipeline register (pc+1).
* **The Forwarding conditions:**

**The forwarding from (EXE/MEM) register (ALU RESULTS) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,R\_type or I\_type in the memory access stage and the following conditions:
* Rd EXE/MEM = (Rs IF/ID & BEQ/BNE in the decoding stage).

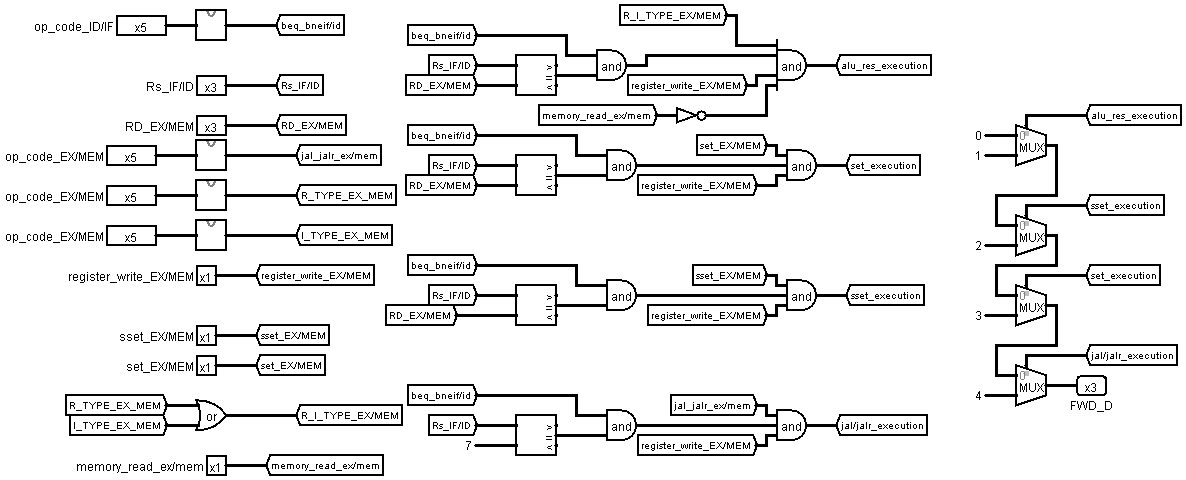
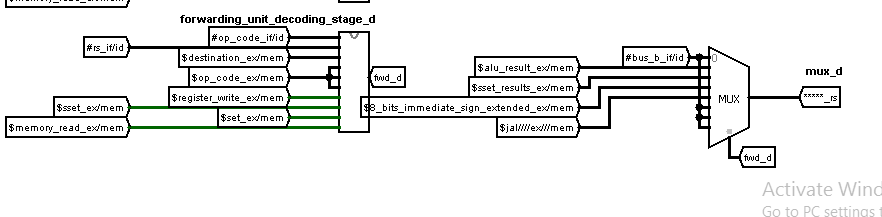
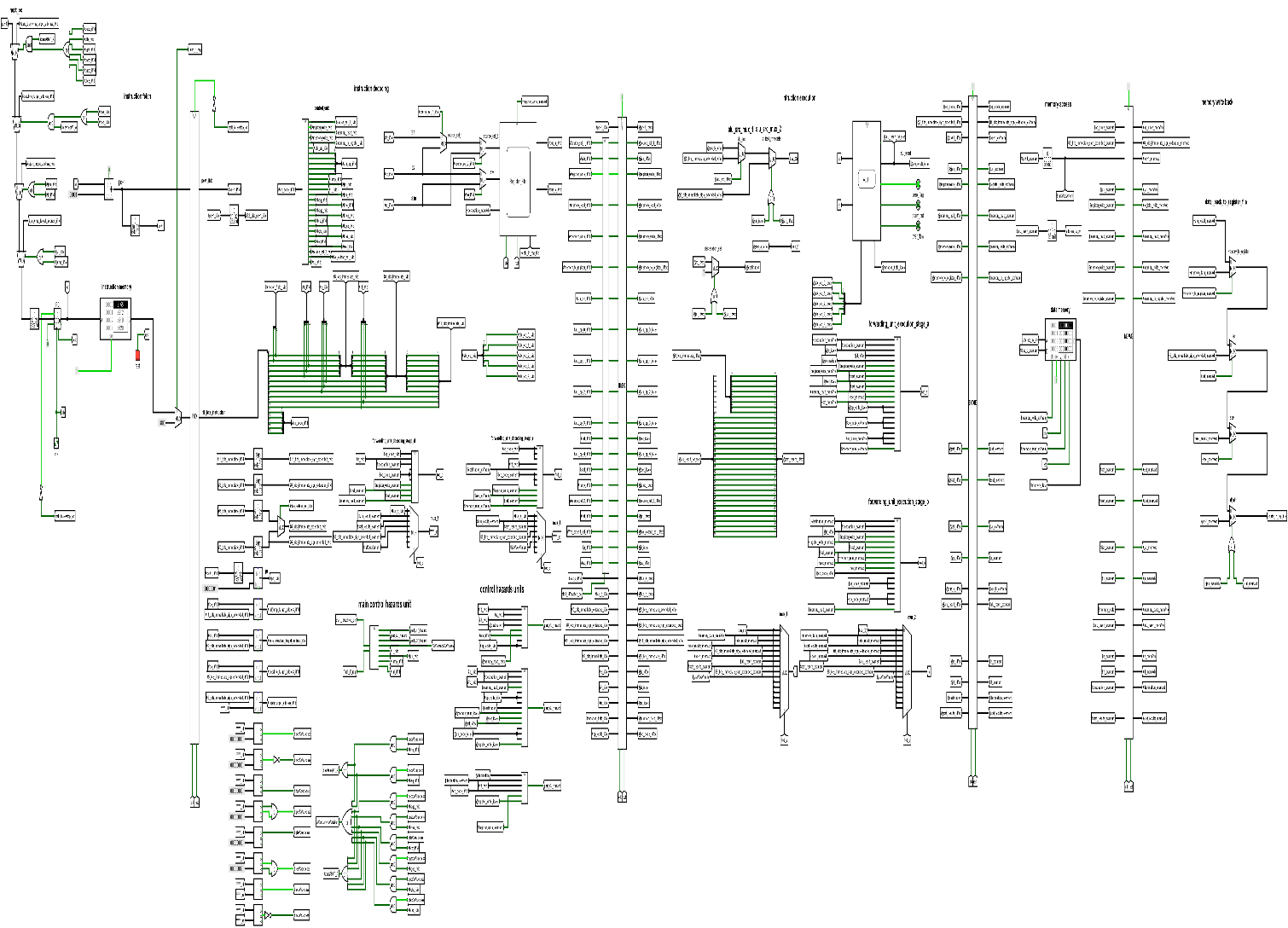
**The forwarding from EXE/MEM register (SET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SET instruction in the memory access stage and the following conditions:
* Rd EXE/MEM = (Rs IF/ID & BEQ/BNE in the decoding stage).

**The forwarding from EXE/MEM register (SSET DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) ,SSET instruction in the memory access stage and the following conditions:
* Rd EXE/MEM = (Rs IF/ID & BEQ/BNE in the decoding stage).

**The forwarding from EXE/MEM register (JAL/JALR DATA) conditions:**

* Register write signal in the EXE/MEM register equals (1) , JAL/JALR instruction in the memory access stage and one of the following conditions:
* Rd (**R7**) EXE/MEM = (Rs (**R7**) IF/ID & BEQ/BNE in the decoding stage).
* **Forwarding unit inputs:**
* (RS / ID-EX)
* (RD / EX-MEM)
* The op code and the control signals.
* **Forwarding unit outputs:**
* Four bits as a selector (B)
* **The pipelining over all datapath.**

The pipelined over all datapath is very similar to the single cycle processor overall datapath:

Same (control unit - arithmetic and logical unit -control signals - register file – data memory –instruction memory).

But the main difference is the existence of the pipelining registers, decoding forwarding units, execution forwarding unit and the hazards detection unit.

**⮚ Simulation and testing ⮘**

* **Test code (1):**
* **The main target of the code:**

Counts the number of 1's in a 32-bit register

* **The Assembly code:**

Set $R0, 1 # Initialize the test register

Sset $R0, 32

Sset $R0, 16

Sset $R0, 8

Set $R1, 0 # "1's counter” register

Sset $R1, 0

Sset $R1, 0

Sset $R1, 0

Set $R2, 0 # "BNEQZ test" register

Sset $R2, 0

Sset $R2, 0

Sset $R2, 0

Set $R3, 0 # the test register size = 32 bits

Sset $R3, 0

Sset $R3, 0

Sset $R3, 32

Start:

Andi $R2, $R0, 1 # check the first right bit in the test register if it is "1" or not

Srl $R0, 1 # shift right to let the next bit ready for test

BNEQZ $R2, Label # If the first right bit is "1" then R2=1 and the condition is true then we branch to label IF not it will continue normally

Addi $R3, $R3,-1 # decrease the size of test bits by 1

BEQZ $R3, Finish # If we finish testing all 32 bits it branch to finish

J start

Label:

Addi $R1, $R1, 1 # the counter increase by 1 if condition is true

Addi $R3, R3,-1 # decrease the size of test bits by 1

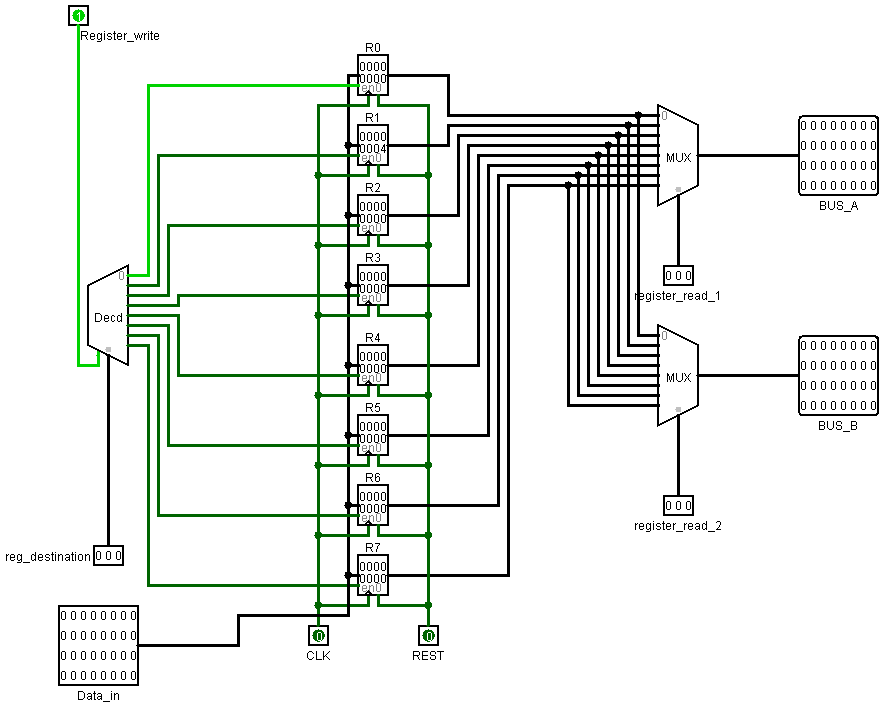
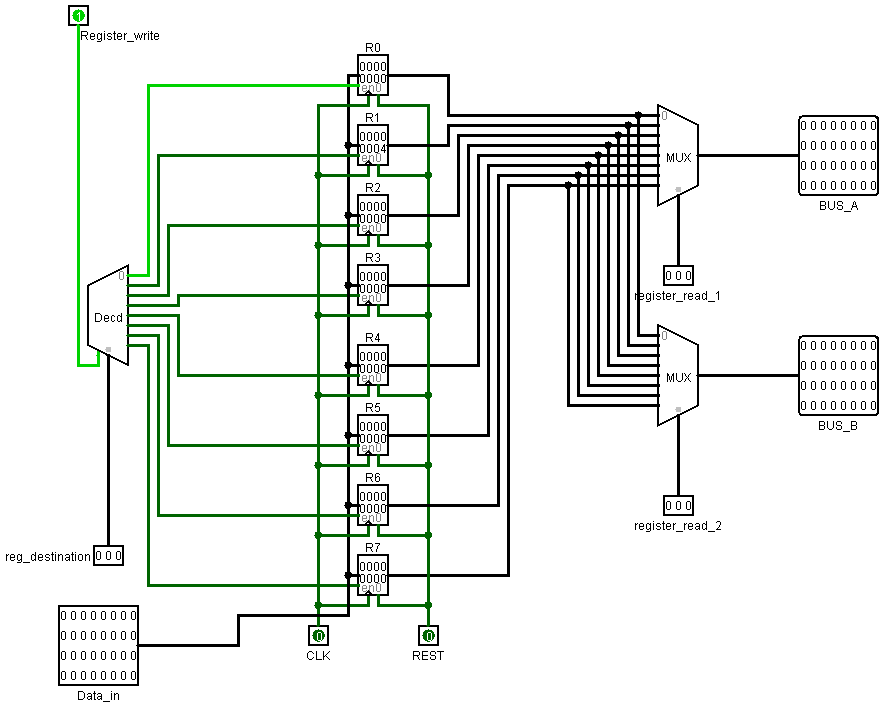
BEQZ $R3, Finish # If we finish testing all 32 bits it branch to finish

J start

Finish:

#inputs: 0000 0001 0010 0000 0001 0000 0000 1000

# expected output "number of 1's”: 4 at $R1

* **The outputs of the program on the single cycle processor:**
* **The outputs of the program on the pipelined processor:**
* **Test code (2):**
* **The Assembly code:**

Set $R0, 0 #array element no.1 =12

Sset $R0, 0

Sset $R0,0

Sset $R0, 12

Set $R1, 0 #array element no.2 =30

Sset $R1, 0

Sset $R1, 0

Sset $R1, 30

Set $R2, 0 #array element no.3 =15

Sset $R2, 0

Sset $R2, 0

Sset $R2, 15

Set $R3, 0

Sset $R3, 0 # array size = 3

Sset $R3, 0

Sset $R3, 3

Set $R4, 0 # size test register

Sset $R4,0

Sset $R4, 0

Sset $R4, 0

Set $R5, 0 #sum register

Sset $R5, 0

Sset $R5, 0

Sset $R5, 0

Set $R6, 0 # "load in" register

Sset $R6, 0

Sset $R6, 0

Sset $R6, 0

Set $R7, 0 #Base address of the first element = 2

Sset $R7, 0

Sset $R7, 0

Sset $R7, 2

SW, $R0, 0($R7) # store R0 in the memory address=2

SW $R1, 1($R7) # store R1 in the memory address=3

SW $R2, 2($R7) # store R2 in the memory address=4

Start:

Beq $R3 ,$R4 ,finish # If we reach the array size "3" then the condition is true and the program branch to finish

LW $R6, 0($R7) # load the first element in R6 and the others later

Add $R5, $R5, $R6 # add the loaded element in R6 to the one at R5

Addi $R7, $R7, 1 # increase the base address by 1 to get the next element

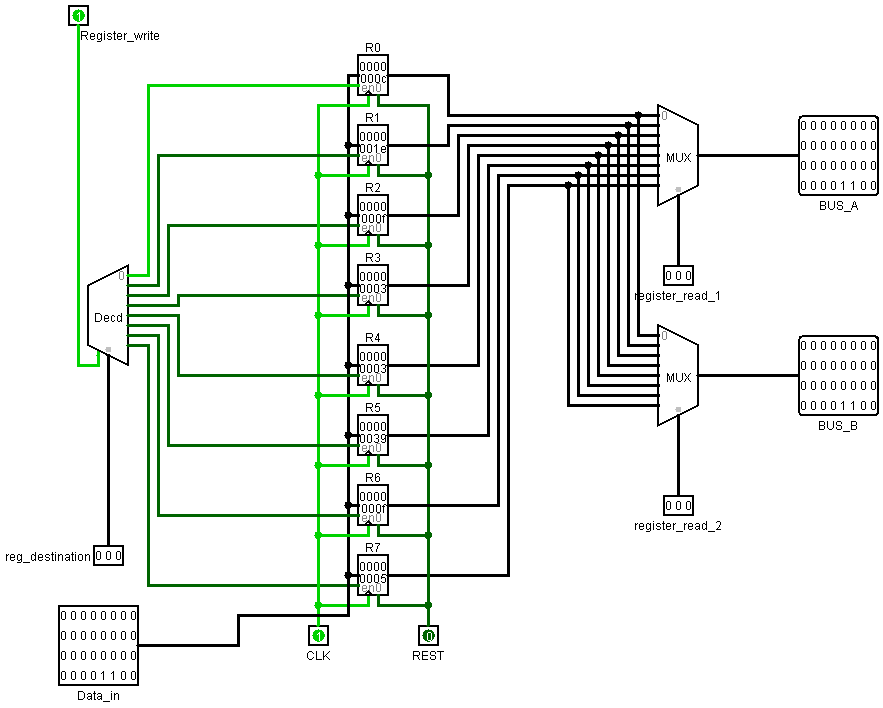
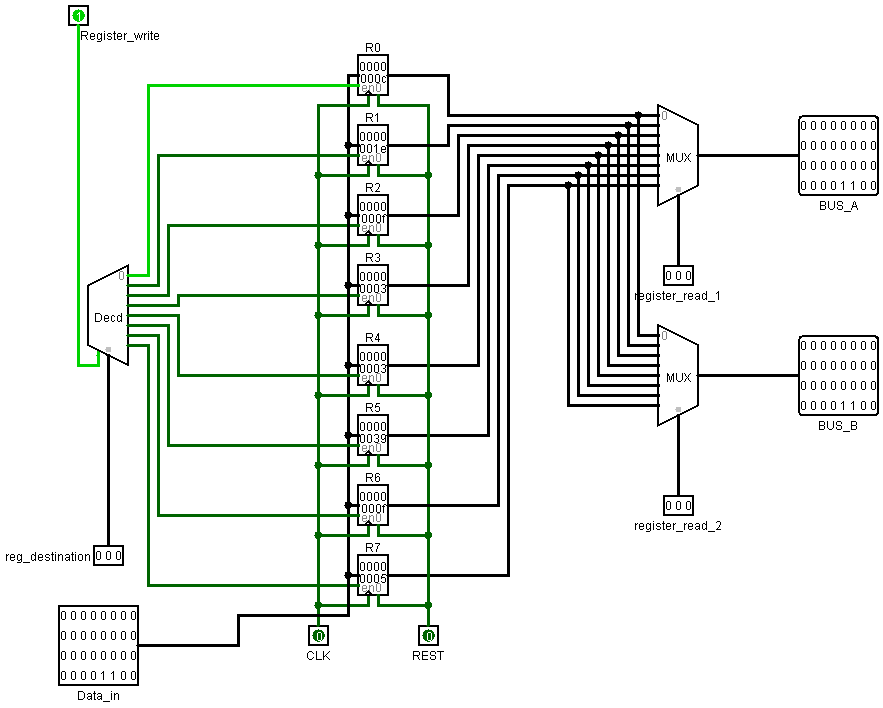
Addi $R4, $R4, 1 # increase the size test register by 1

J start

Finish:

# inputs 12, 30, and 15

# expected output “sum of array elements" 57 in $R5

* **The outputs of the program on the single cycle processor:**
* **The outputs of the program on the pipelined processor:**
* **Test code (3):**
* **The Assembly code:**

Set r1, 0x55; set Reg [1] to 0x12102000

Sset r1, 0x12

Sset r1, 0x10

Sset r1, 0x20

Sset r1, 0x00

Set r6, 0x22; set Reg [6] to 0x22060211

Sset r6, 0x06

Sset r6, 0x02

Sset r6, 0x11

Eqv r3, r6, r1

Sub r0, r0, r0; set Reg [0] to 0, use as base

LW r1, 0(r0); Reg [1] <- mem [0]

LW r2, 1(r0); Reg [2] <- mem [1]

LW r3, 2(r0); Reg [3] <- mem [2]

Addi r4, r4, 10

Sub r4, r4, r4; Reg [4] <- 0, running total

Add r4, r2, r4; Reg [4] + = A

SLT r5, r2, r3; Reg [5] <- A < B

Beq r5, r0, 3; if Reg [5] = FALSE, go forward 2 instructions

Add r2, r1, r2; A++

Beq r0, r0,-4; go back 5 instructions

Sw r4, 0(r0); mem [0] <- Reg [4]

Jal func

Ror r6, r6, 8

lw r4, 1(r0)

lw r5, 2(r0)

add r6, r6, r5

Beq r0, r0,-1; program is over, keep looping back to here

Func: sub r0,r0,r0 ; set Reg[0] to 0

LW r1, 0(r0); Reg [1] <- mem [0]

LW r2, 0(r1); Reg [2] <- mem [55]

LW r3, 1(r1); Reg [3] <- mem [56]

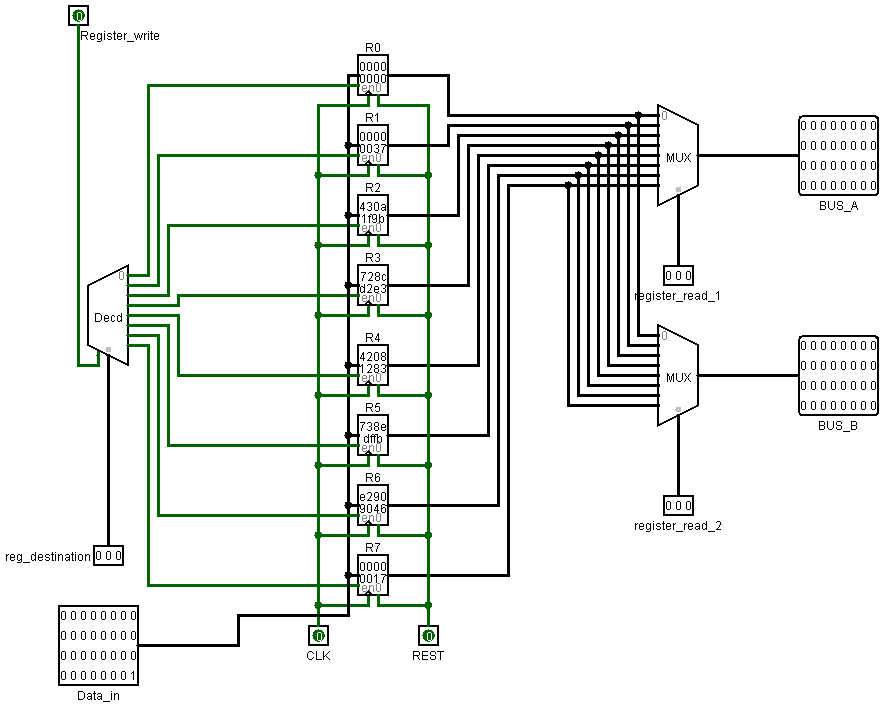
And r4, r2, r3; Reg [4] <- Reg [2] AND Reg [3]

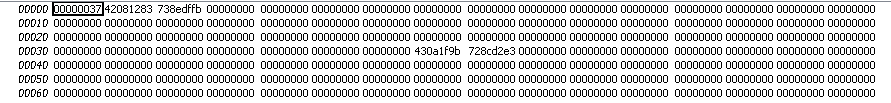
Or r5, r2, r3 ; Reg[5] <- Reg[2] OR Reg[3]

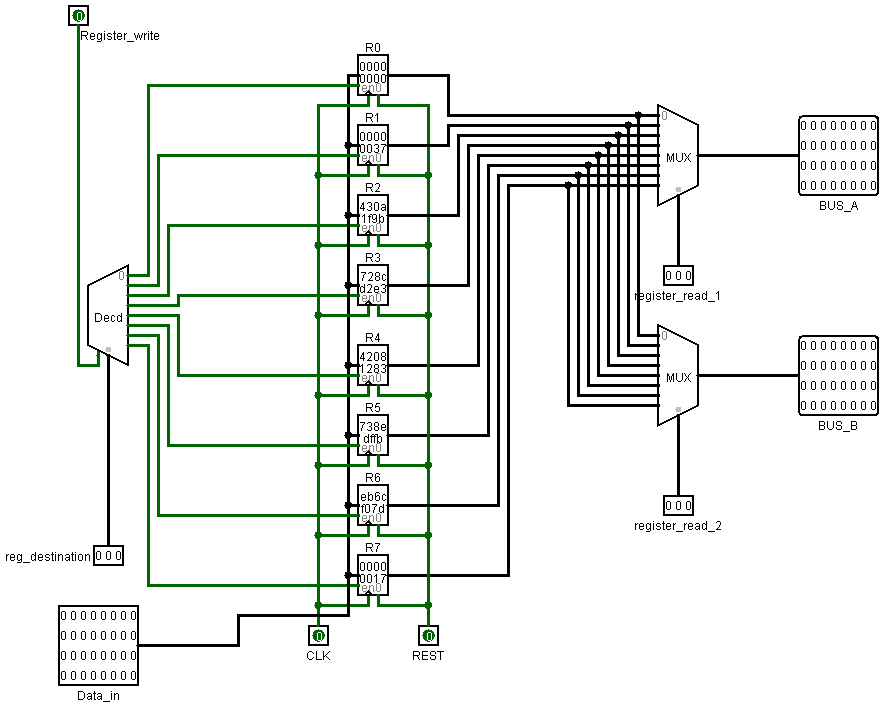
Sw r4, 1(r0); mem [1] <- Reg [4]

Sw r5, 2(r0); mem [2] <- Reg [5]

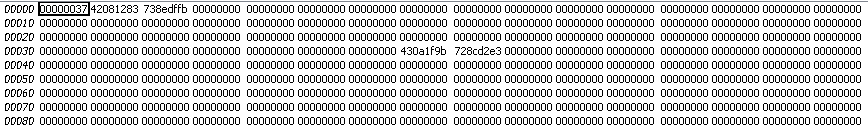
Jr r7

* **The outputs of the program on the single cycle processor:**

**Data memory state:**

* **The outputs of the program on the pipelined processor:**

**Data memory state:**

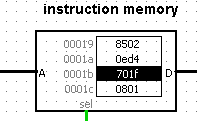
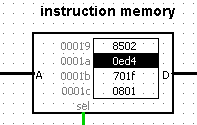


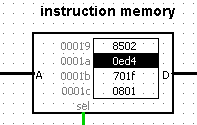
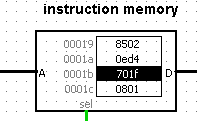
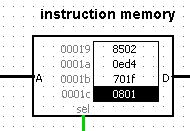
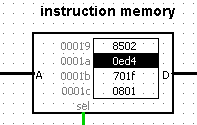
**Comments on this codes:**

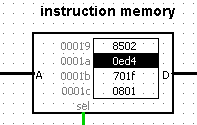
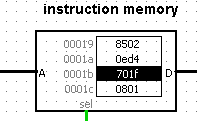
The sequence make the processor keep looping and change the contents of r6 .This is reason for the difference in results

Add r6, r6, r5

Beq r0, r0,-1

**In Single cycle case:**

**In pipelining case:**

The reason for this difference is that the branchinstruction always taken so the processor has to flush the fetched insteruction.