COMP2611: Computer Organization

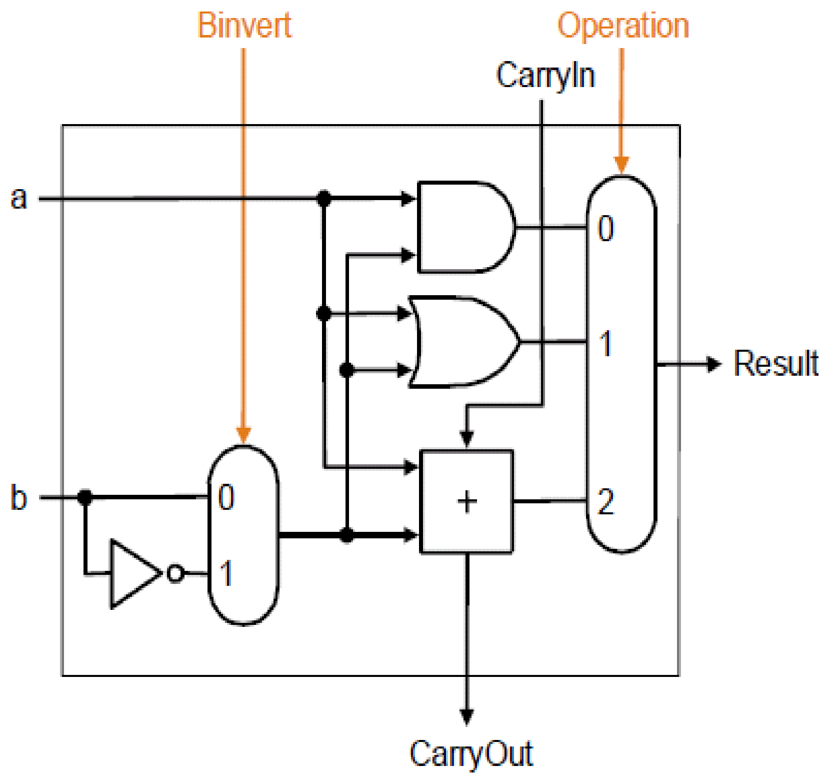
4-bit ALU and processor in Logisim

Overview

- You will learn the following in this lab:
  - ❑ building a 4-bit ALU,
  - ❑ an implementation of the single-cycle MIPS processor in Logisim,
  - ❑ executing instructions in that processor implementation.

Reminder : an 1-bit ALU that does AND, OR, Addition, Subtraction

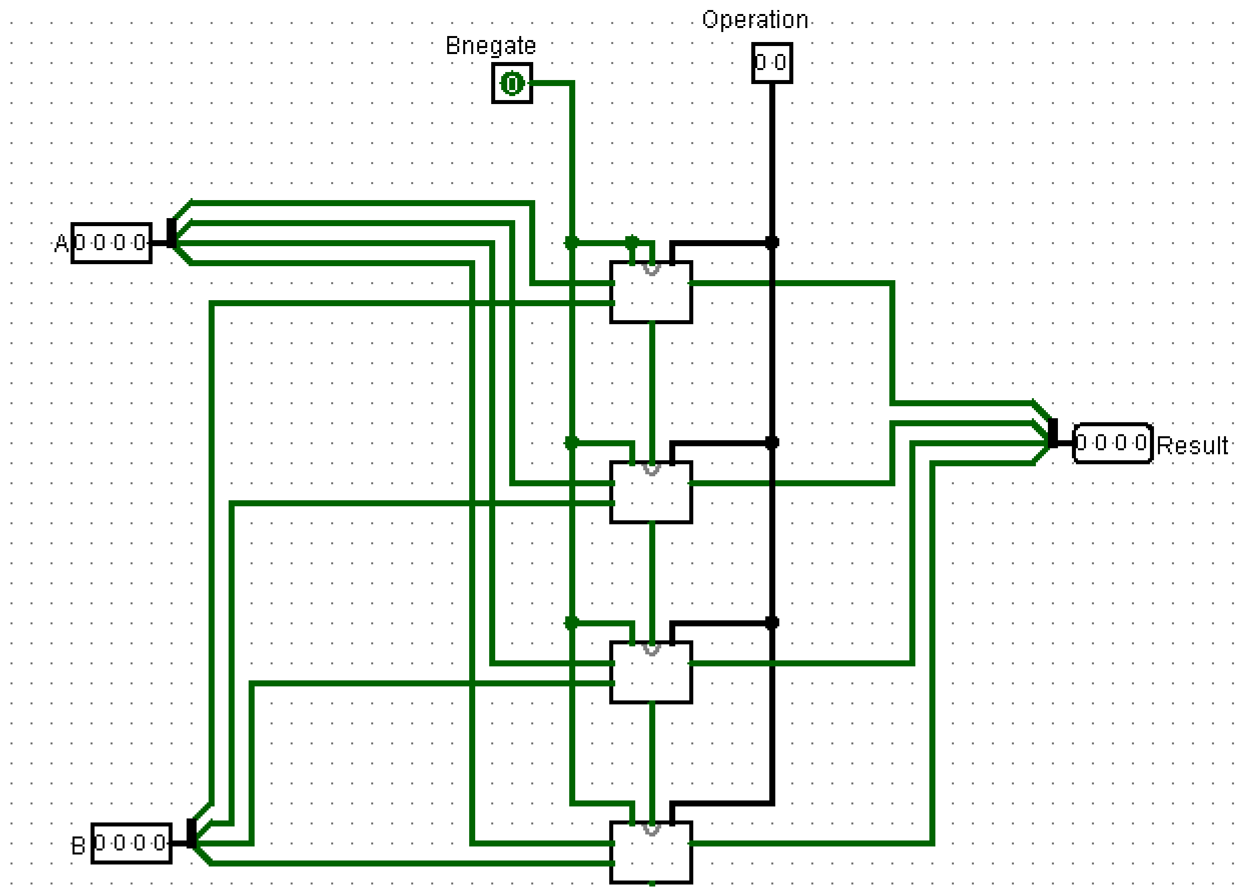
- The 1-bit ALU can perform AND, OR, Addition and Subtraction operations on two 1-bit inputs.



- A 4-bit ALU can be built using four 1-bit ALUs shown above. Each 1-bit ALU will take care of the operations for exactly one bit.

Building the 4-bit ALU 1/4

- **One possible implementation of a 4-bit ALU.**



Building the 4-bit ALU 2/4

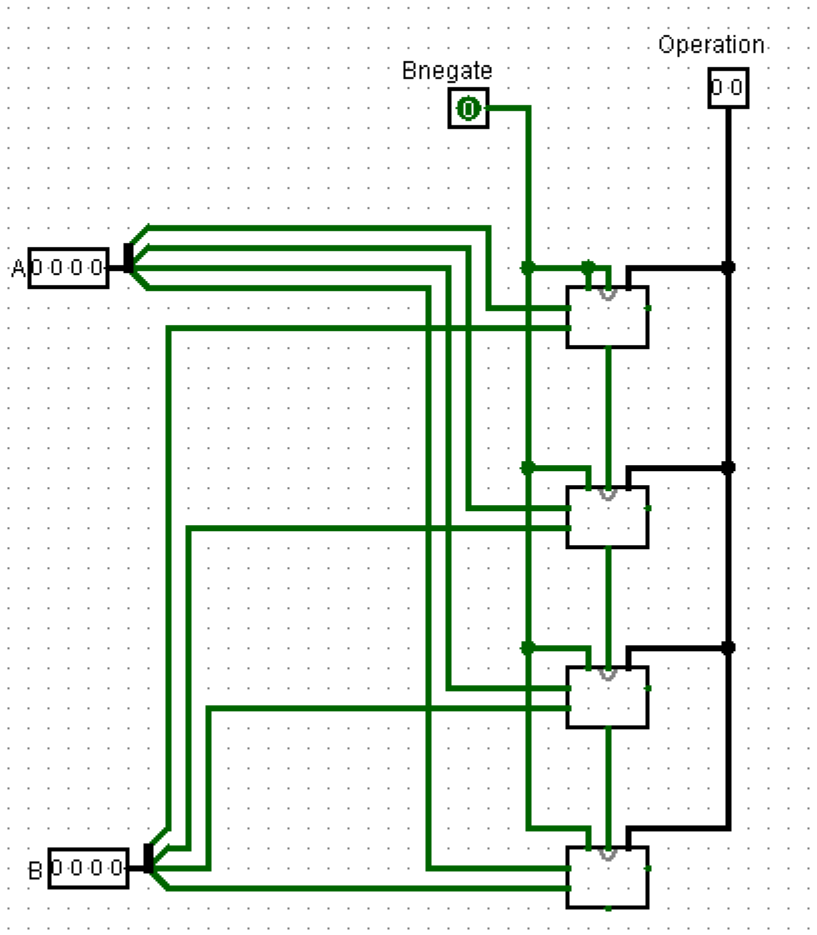
- Download the logisim files 1-bit-adder.circ, 3-input-multiplexor.circ and 1-bit-alu.circ created from the last lab to the same folder.

- Add the 1-bit-alu.circ in the current Logisim project by clicking "Project->Load Library->Logisim Library".

- Add the 1-bit-alu circuit four times into the canvas for connections:
  - ❑Pay attention to the connections of CarryIn/CarryOut between 1-bit ALUs.
  - ❑Pay attention to the connections of Operation bits between 1-bit ALUs.
  - ❑Pay attention to the connections of Binvert bits between 1-bit ALUs.

Building the 4-bit ALU 3/4

- Add two input pins "A" and "B", each of which is an operand with 4-bit data.
- Connect these two input pins with four 1-bit ALUs correctly (note that splitters may be used).
- Add an output pin as the "Carry_out", and connect it correctly.



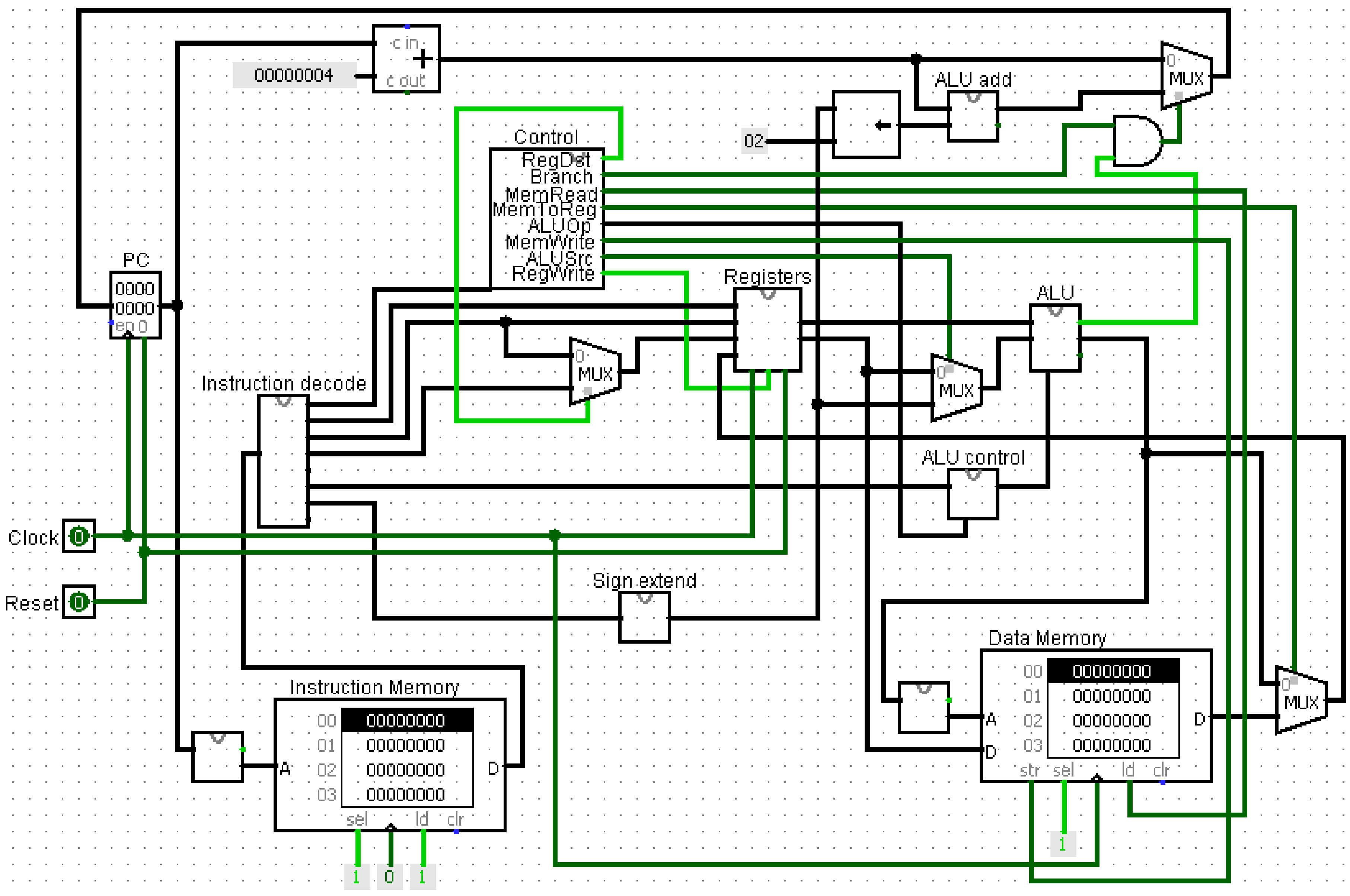
Building the 4-bit ALU 4/4

- Add an output pin as the "Result" and connect it correctly (note that a splitter may also need to be used to facilitate the 4-bit output).
- After finishing all the connections, test the 4-bit ALU to check for its correctness.

Single-cycle MIPS processor

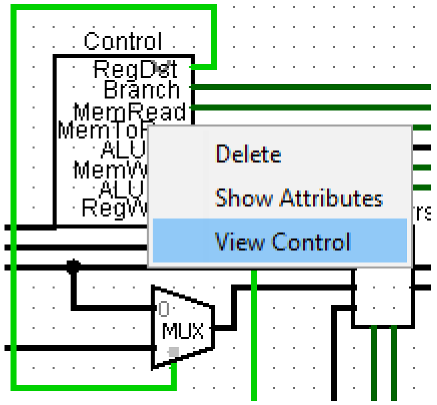
- Download the logisim files processor.circ, cpu32.circ and misc32.circ to the same folder.
- Open the processor.circ as a new project in Logisim.
- This file shows an implementation of a 32-bit single-cycle MIPS processor as shown on the next page.

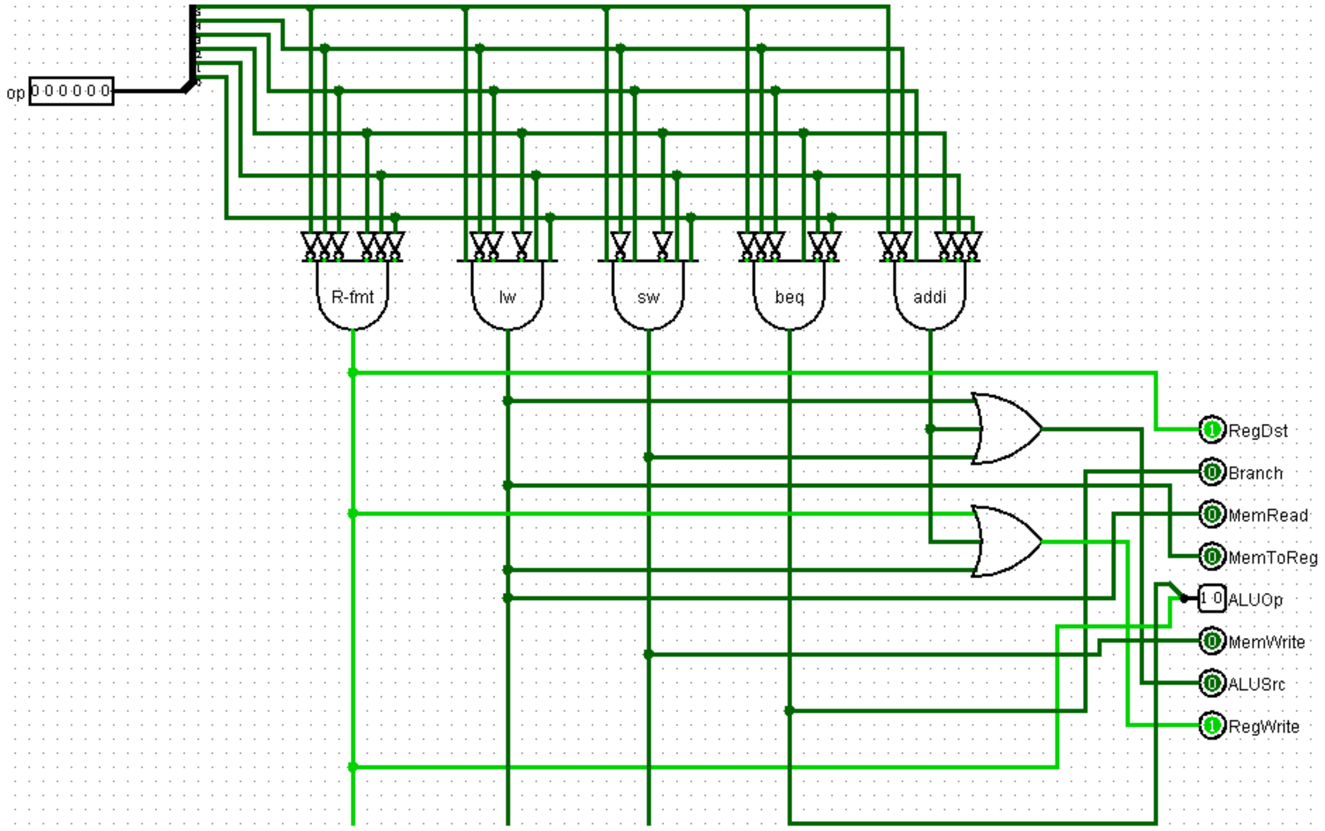
Single-cycle MIPS processor



Black-box sub-circuits

- For each black-box sub-circuit, you can view its implementation by right-clicking on it and select the "View CircuitName" command.
- For example, right-click on the circuit "Control" which is the Control Unit of the processor and select the command "View Control".
- This shows the circuit's implementation.

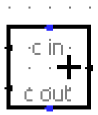


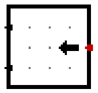
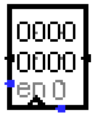


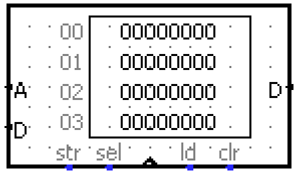
Supported MIPS instructions

- You can also view the implementation of the circuit “ALU control”.
- These two circuits show that the processor implementation only supports these MIPS instructions: lw, sw, beq, add, sub, and, or, slt, addi.

Built-in Logisim circuits

- The processor uses the following built-in Logisim circuits not taught in the past labs (see the help manual of Logisim if you want to know their details):
  - Adder


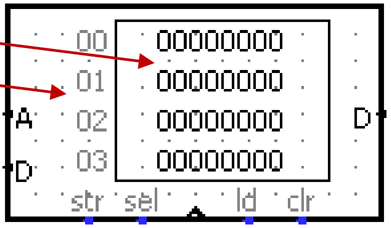
  - Shifter

  - Register


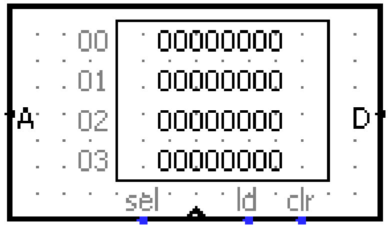
    - Its value is displayed on the circuit using the hexadecimal format and can be updated to its data input only when its input "en" is 1 or is not connected.
    - For all the registers in the processor, any value update is set (by an attribute of Register) to be triggered at the rising edge of the clock.
  - RAM


    - It is a re-writable memory unit.
    - Data is loaded from or stored to it by setting its input "ld" or "str", respectively, to 1.

RAM

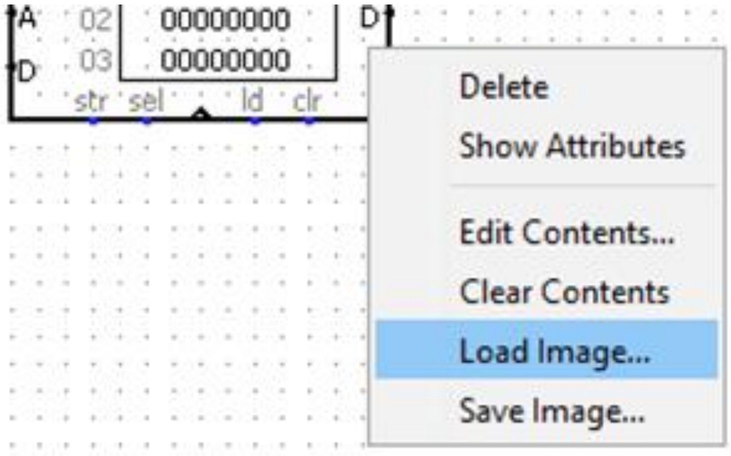
- The data at each memory address in RAM can be set (by an attribute of RAM) to have a certain bit width.
- The first few addresses and the datum in them are displayed on the RAM circuit using the hexadecimal format.
- Input A is the input address.
- By setting an attribute of RAM, the circuit can have:
  - an output D on the right side for outputting the data at the address A and an input D on the left side for the input data to be stored at the address A, or
  - only one D for the output data or input data depending on the input "ld" (e.g. set to 1 for the output data).





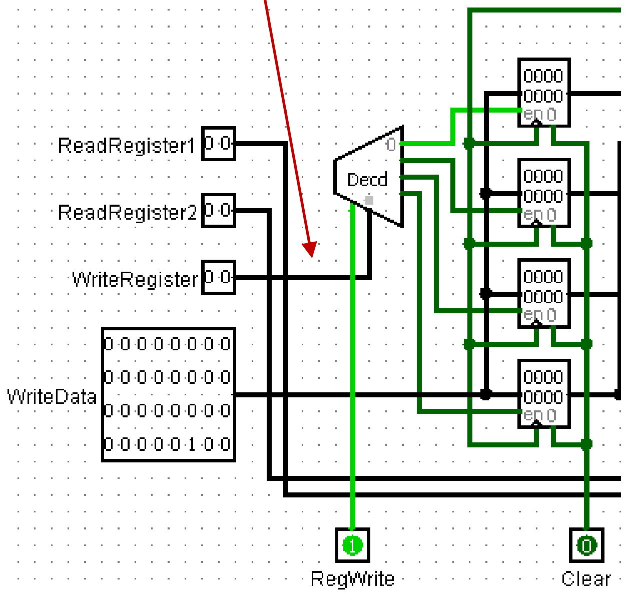
Setting RAM data

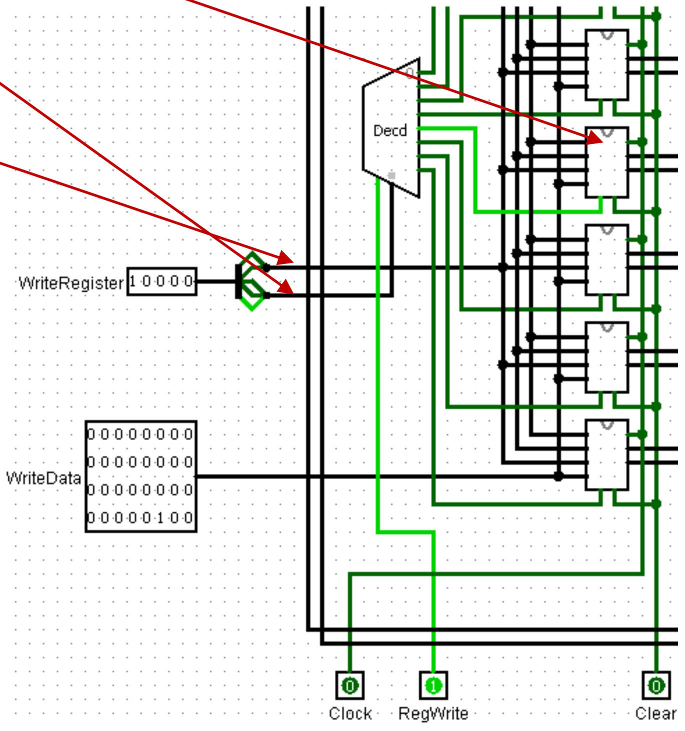
- By right-clicking on RAM,
  - all its datum can be reset to zero by selecting the command “Clear Contents”,
  - they can be displayed for editing by selecting the command “Edit Contents”,
  - they can also be set using an memory image file by selecting the command “Load Image”,
- Memory image file format:
  - the first line is always “v2.0 raw” for the file type,
  - the data at each memory address is put in each line afterward using the hexadecimal format,
  - if fewer lines than the available addresses are used, the datum at all the remaining addresses are set to zero.



Register file

- The circuit "Registers" on the processor is the register file.
- It consists of 8 black-box sub-circuits, each of which contains 4 registers.
  - For a register no., its most three significant bits are used to determine which sub-circuit contains that register, and
  - then its other 2 bits are used to select that register among the fours inside the sub-circuit.





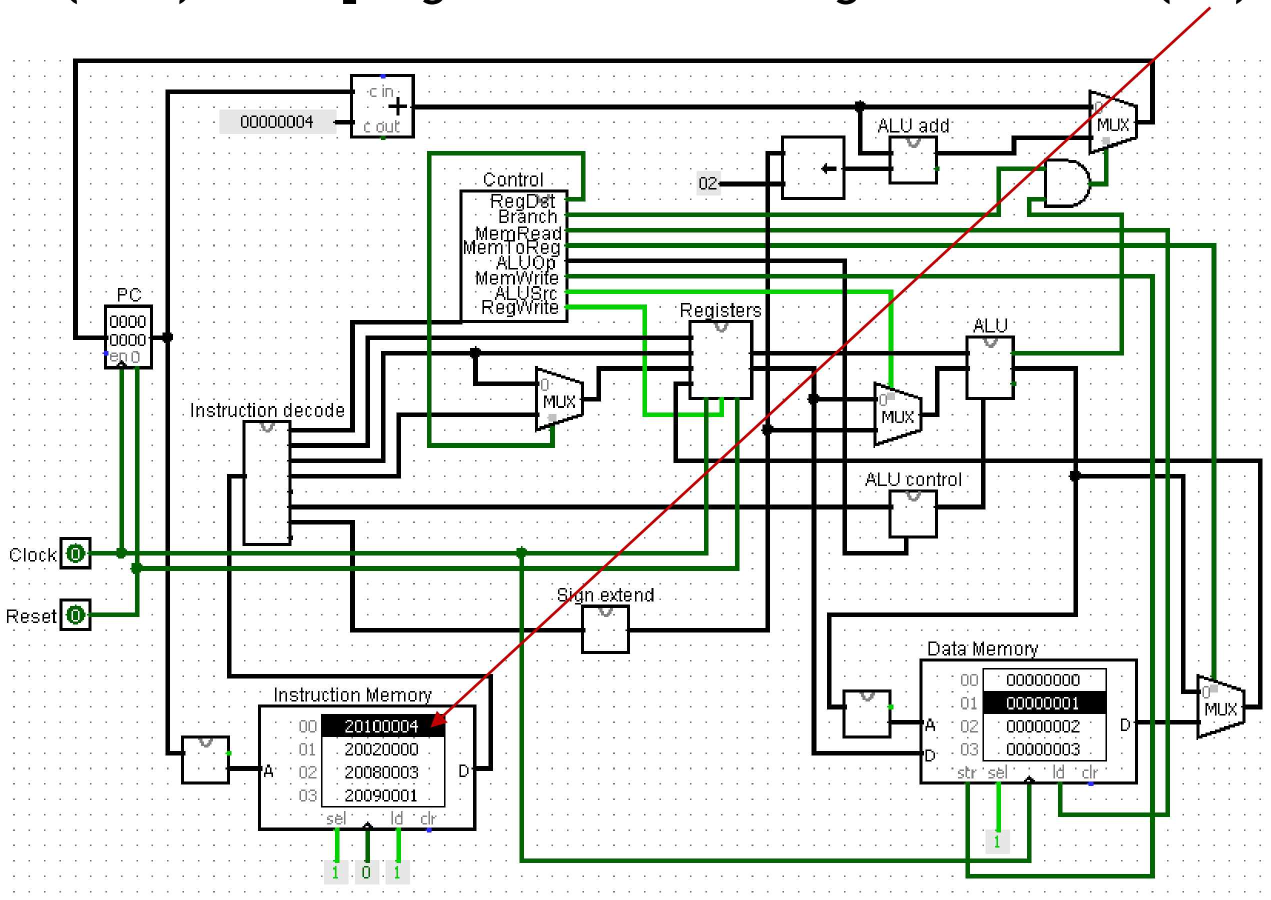
Example MIPS program

- **We now demonstrate the execution of the MIPS program below in the processor.**

`.data`		Load the memory image file data.txt to the Data Memory of the processor. This stores arr in the memory's 8-bit address 0x01.
`arr: .word 1 2 3`		
`.text`		Load the memory image file instruction.txt to the Instruction Memory of the processor. This stores the machine codes of the program's instructions in the memory's address 0x00.
`#use 32-bit address 0...0100 of arr`	Address	
`addi $s0, $zero, 0x4`	0x00	
`addi $v0, $zero, 0   #sum of arr elements`	0x01	
`addi $t0, $zero, 3   #remaining loop steps`	0x02	
`addi $t1, $zero, 1`	0x03	
`loop: lw $s1, 0($s0)`	0x04	
`add $v0, $v0, $s1`	0x05	
`sub $t0, $t0, $t1`	0x06	
`addi $s0, $s0, 4`	0x07	
`slt $t2, $t0, $t1    #remaining steps < 1?`	0x08	
`beq $t2, $zero, loop`	0x09	
`sw $v0, -4($s0)`	0x0A	

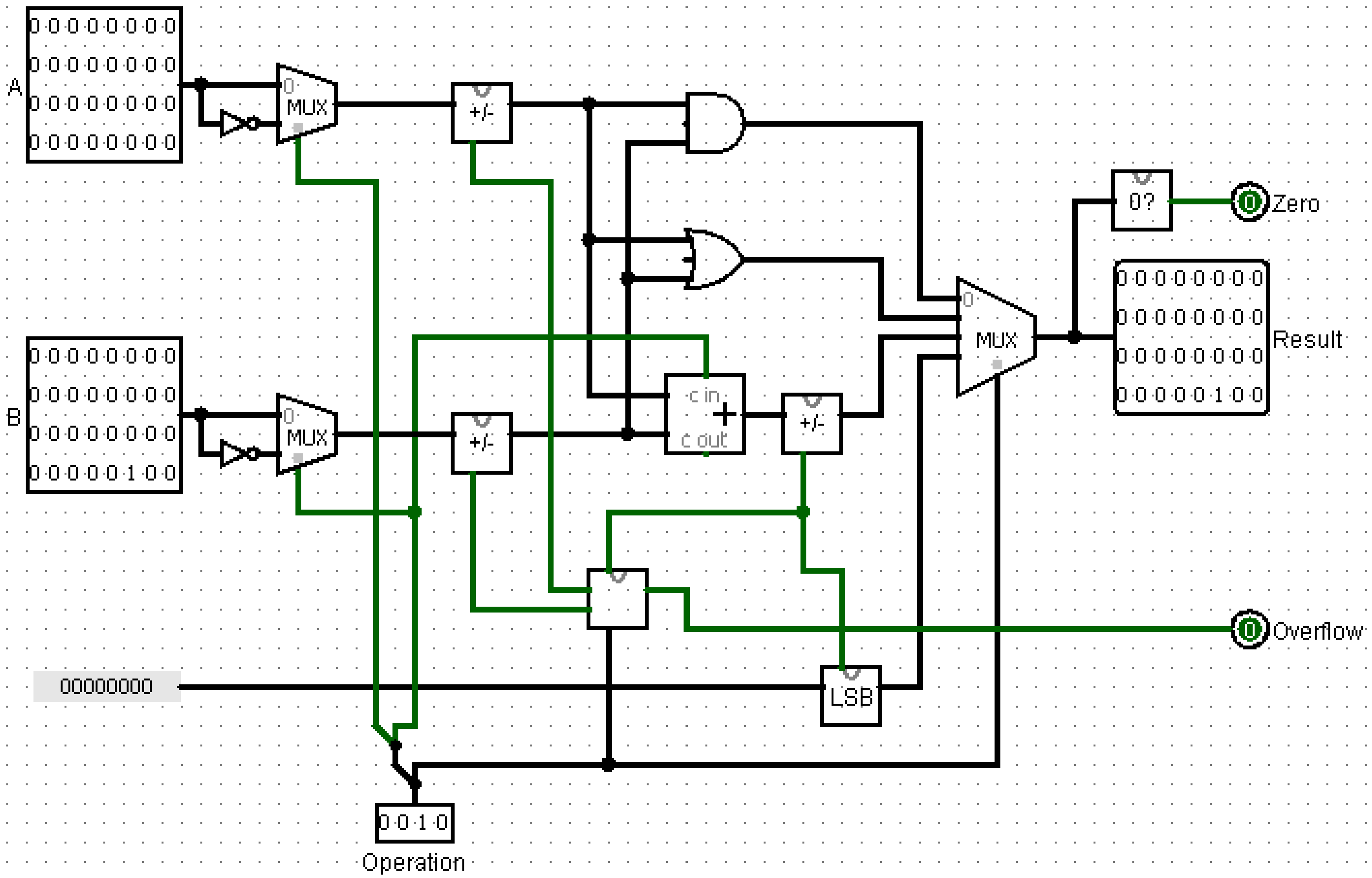
New program execution

- Before starting a new program execution in the processor, set the input pin "Reset" to 1 to clear the values of all the registers and then set it back to 0.
- The signals on the datapath of the processor are now set for the first instruction (addi) of the program when the Program Counter (PC) is 0.



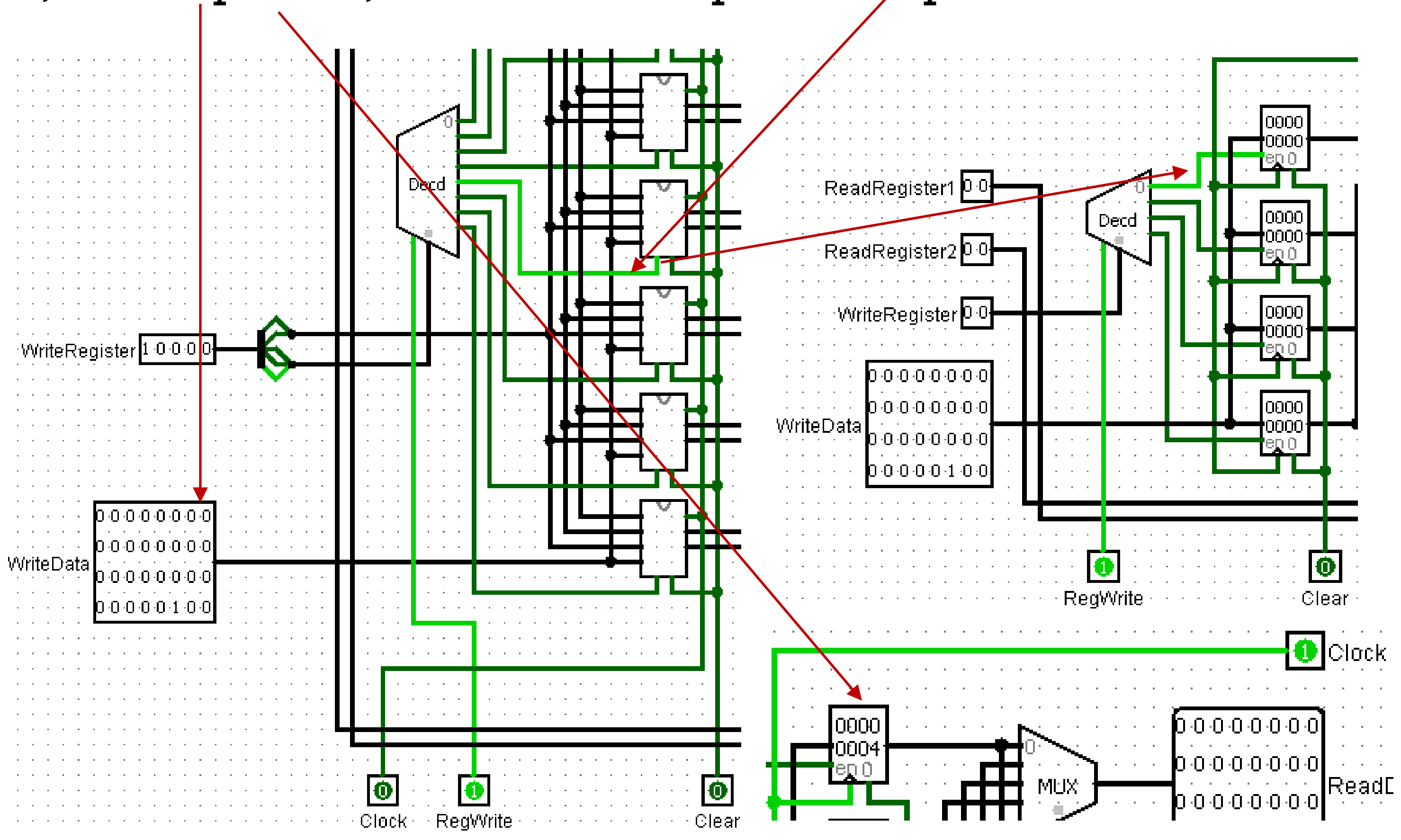
ALU signals for "addi $s0, $zero, 0x4" (instr addr 0x00)

- Since ALUSrc is 1, the immediate number 4 in the instruction is forwarded to the input B of the ALU. The value of the $zero register is forwarded from the register file to the input A.
- The Result in the ALU is set to the sum of A and B (Operation is 0010).



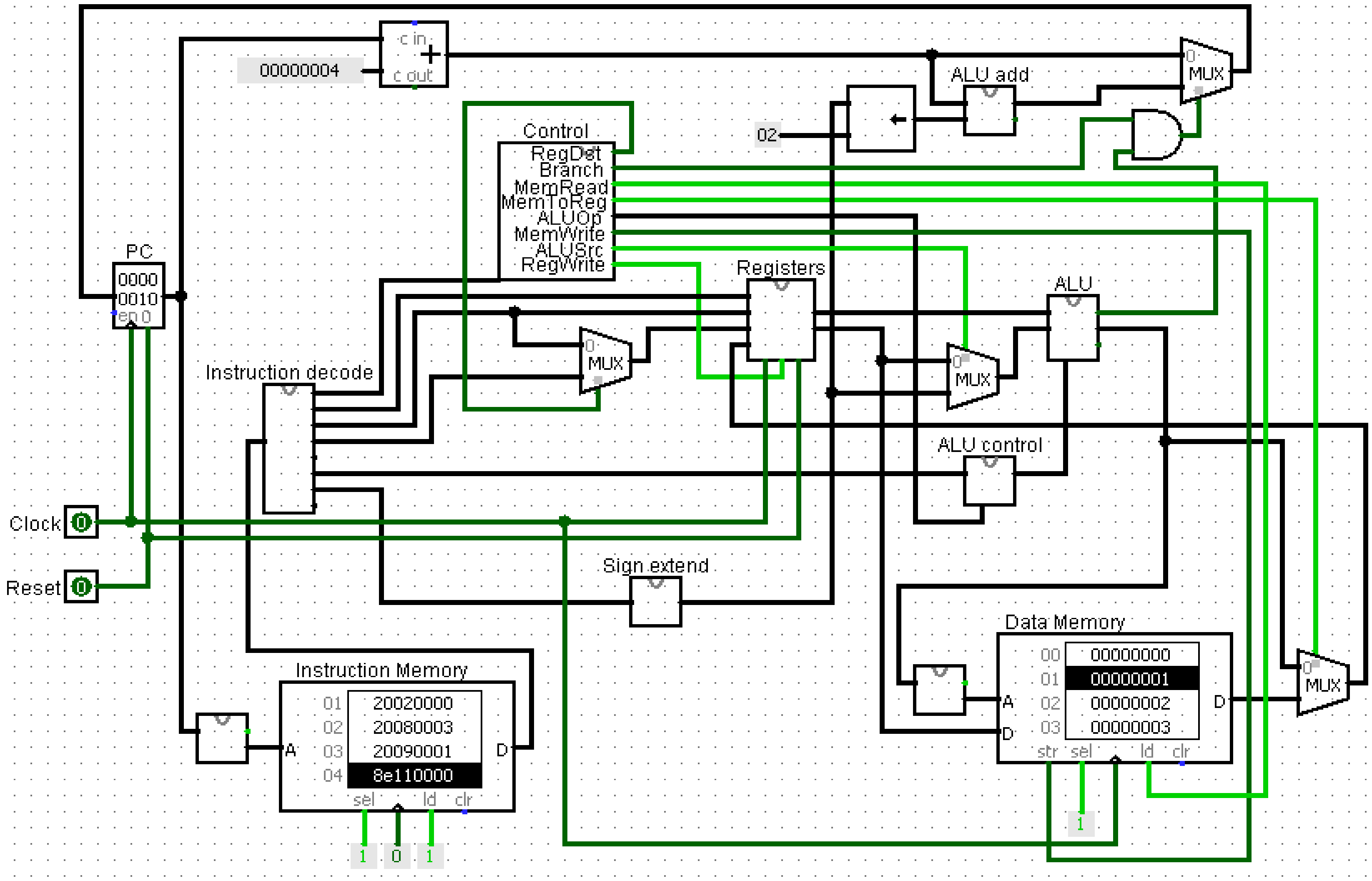
Writing register

- In the circuit "Registers", the register $s0 (or $16) is enabled for writing.
- We now set the clock (input pin "Clock") of the processor so that a rising edge occurs in its signal to complete the instruction execution.
- Then, $s0 is updated, and PC is also updated to point to the next instruction.



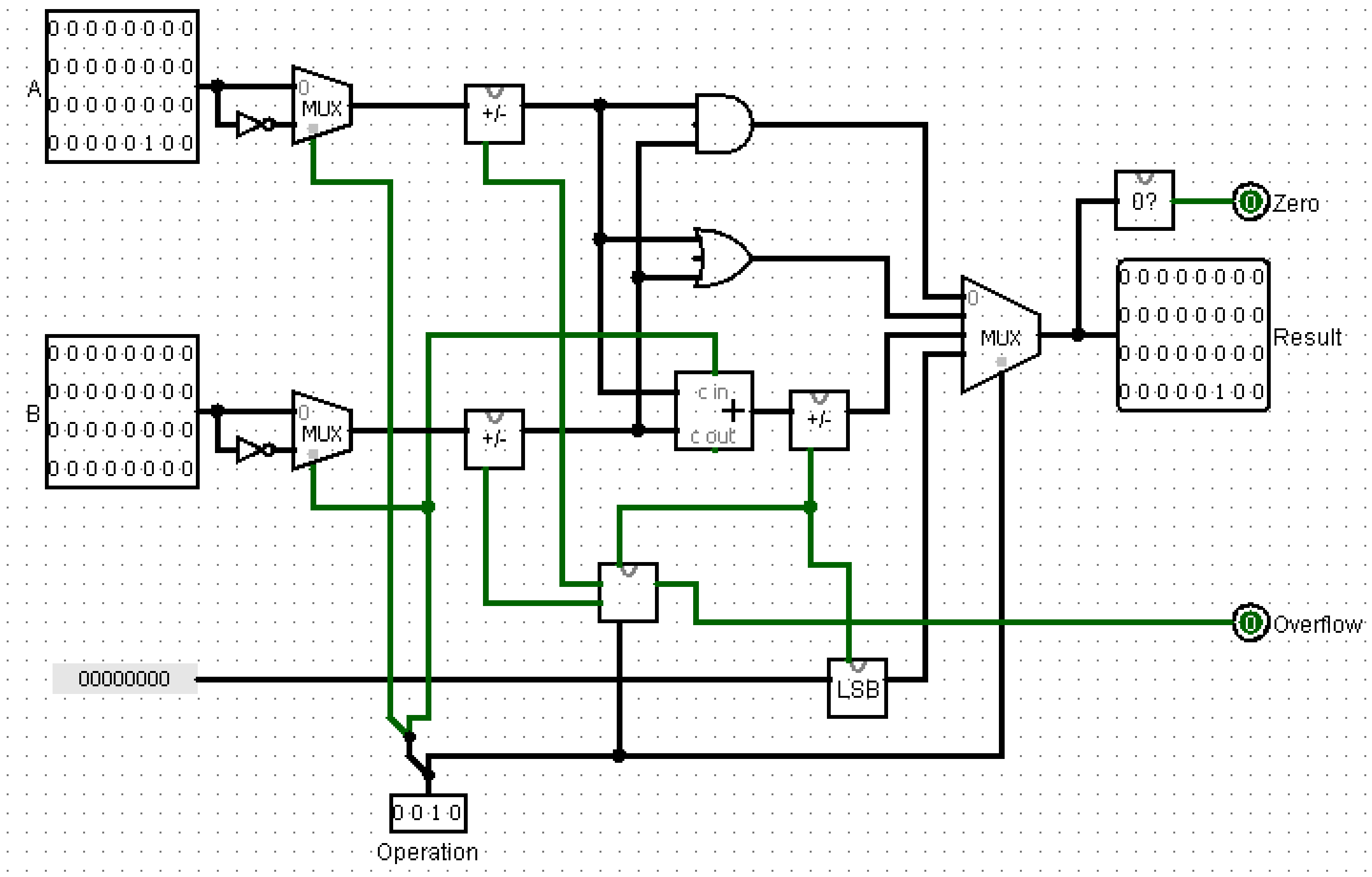
Signals for “lw $s1, 0($s0)” (instr addr 0x04)

- **Continue the program execution until PC is pointing to the lw instruction at the address 0x04 in the Instruction Memory.**



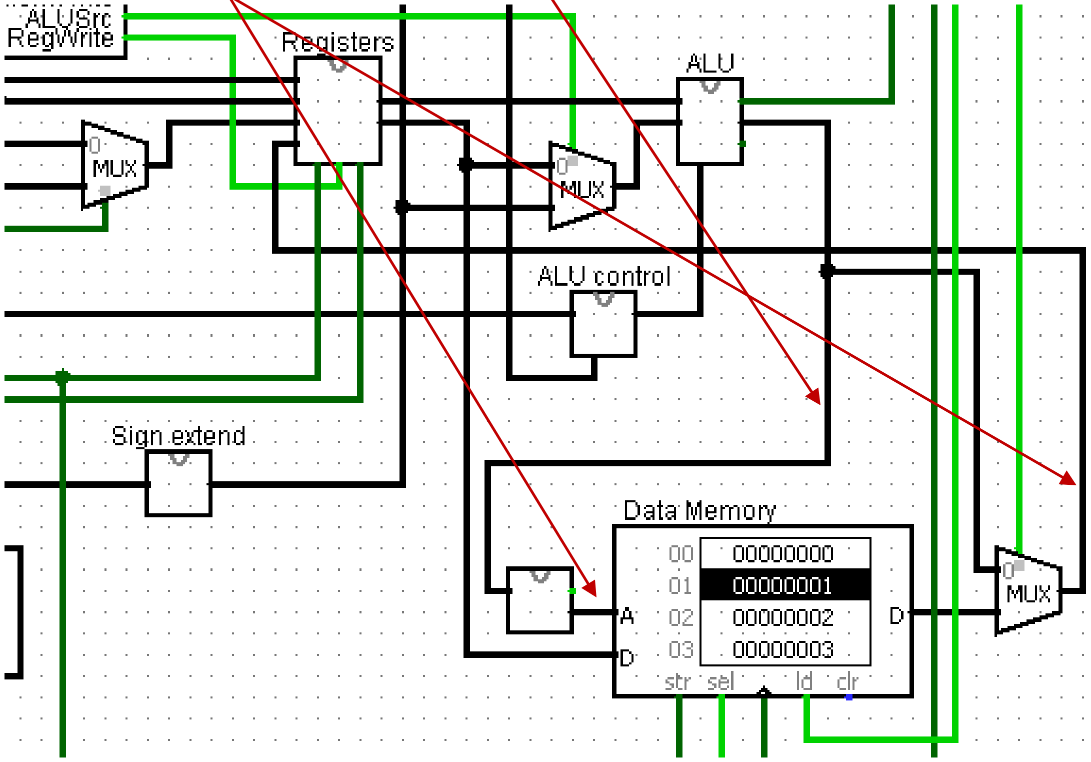
ALU signals

- **In the ALU, lw's memory loading address is computed in the Result as the sum of A ($s0 value) and B (the address offset 0 in lw).**



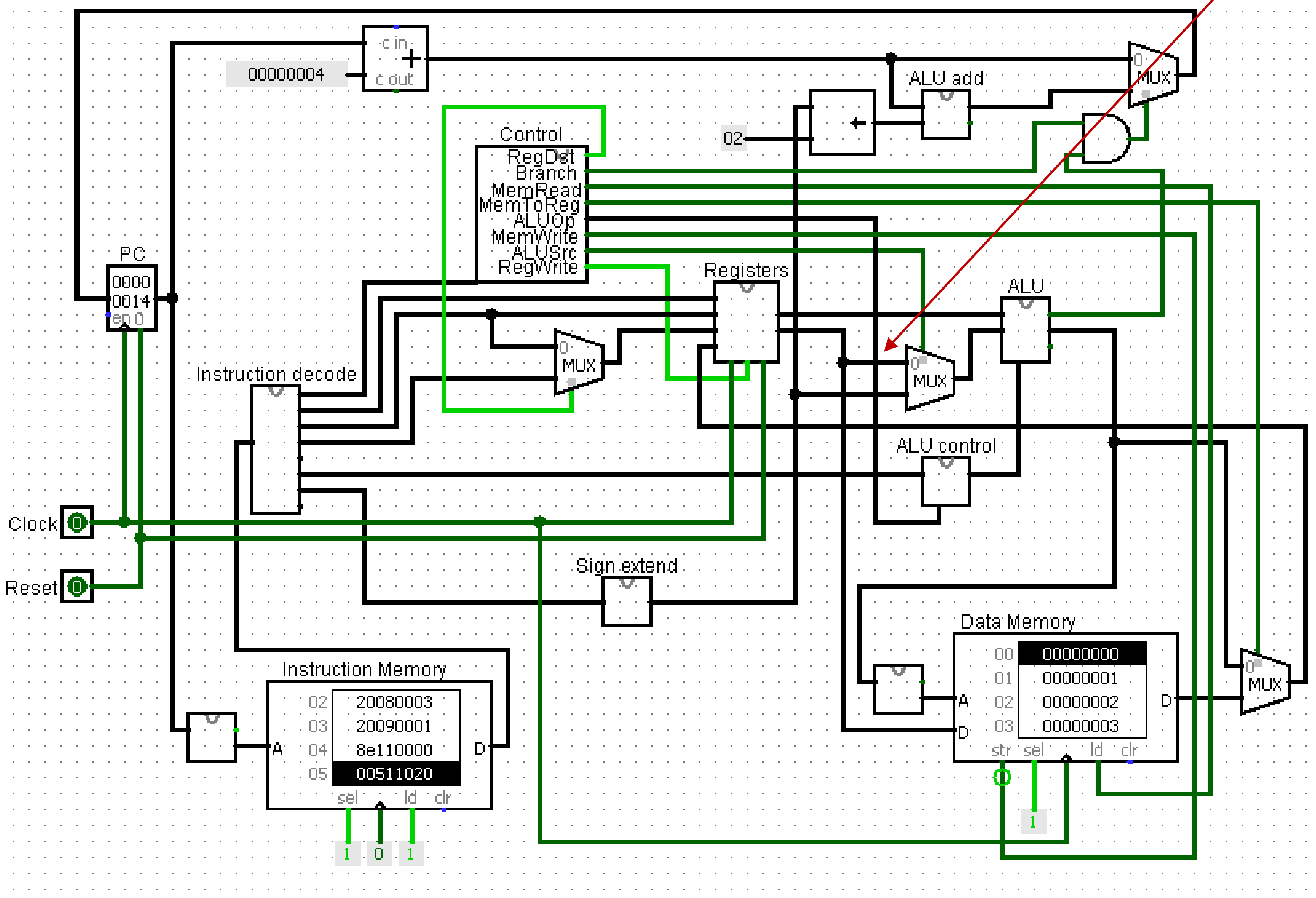
Data Memory signals

- The loading address 0…0100 from the ALU is then used to select the word at the address 0x01 in the Data Memory for reading.
- The word is read and is forwarded to the register file for writing it to $s1.



Signals for "add $v0, $v0, $s1"   (instr addr 0x05)

- The signals of the next instruction (add) are like those of addi, but $s1 value instead of an immediate number is forwarded to B of the ALU.



Instruction Memory:

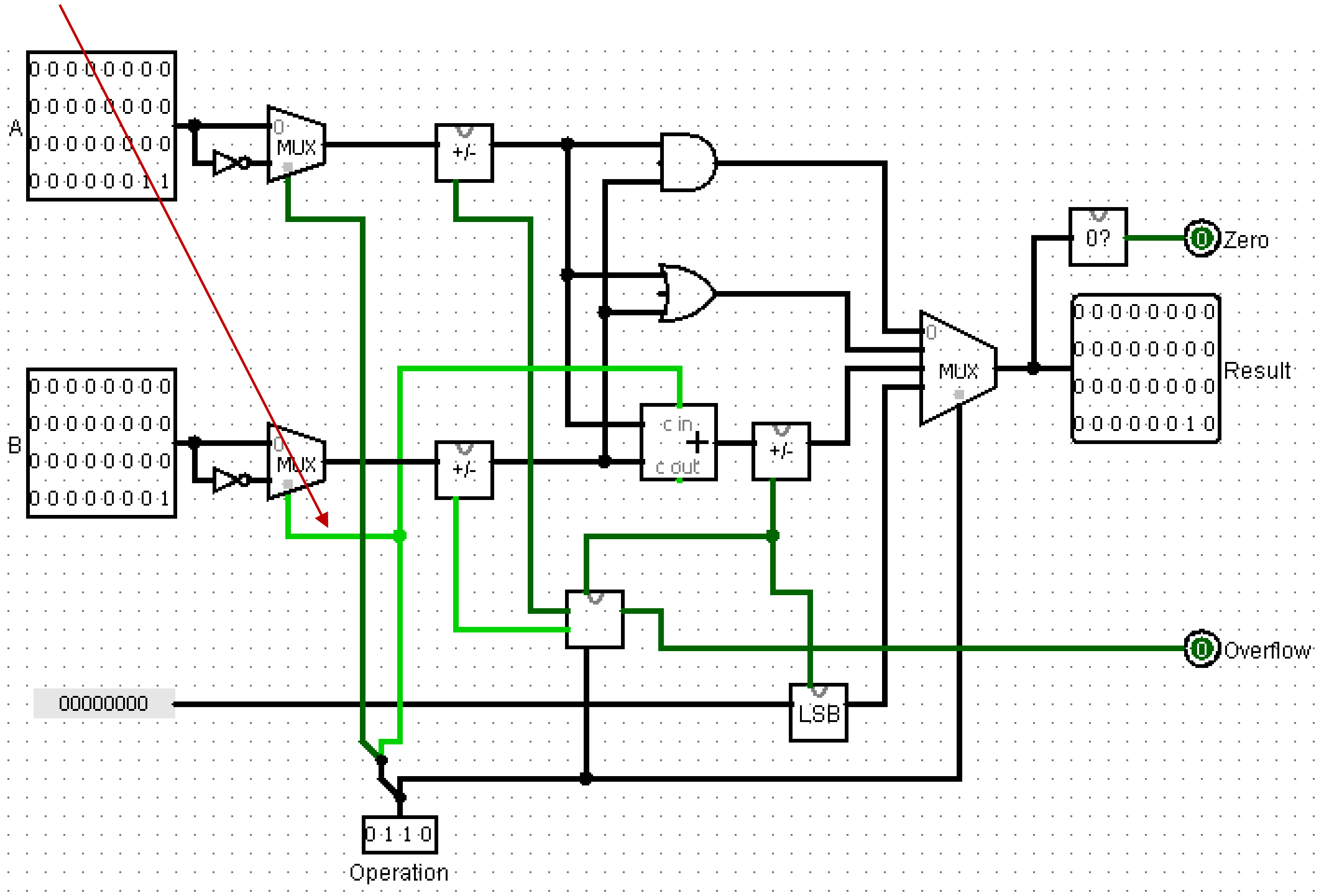
02	20080003
03	20090001
04	8e110000
05	00511020

Data Memory:

00	00000000
01	00000001
02	00000002
03	00000003

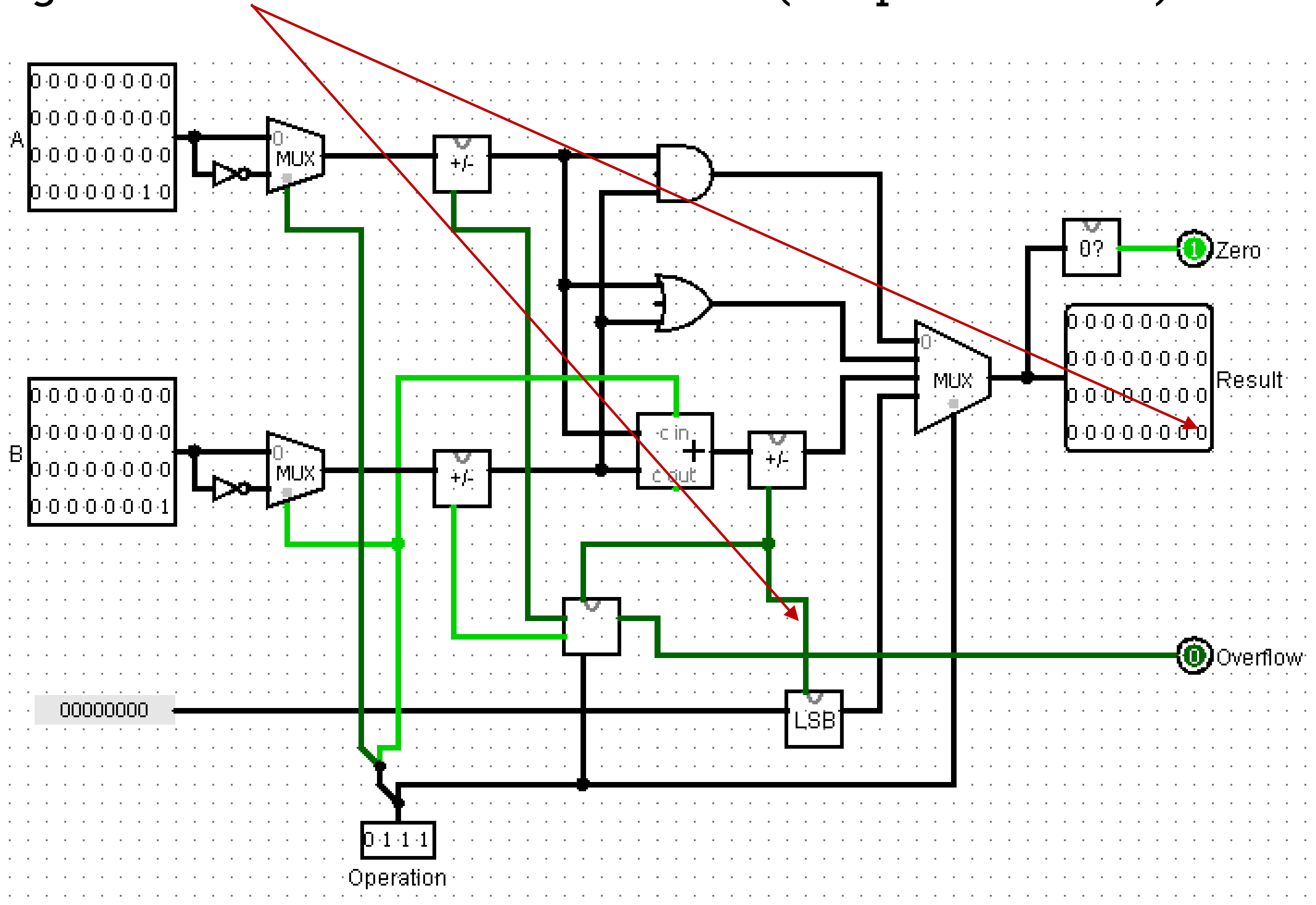
ALU signals for “sub $t0, $t0, $t1” (instr addr 0x06)

- **The signals of the next instruction (sub) are like those of add, but B of the ALU is negated.**



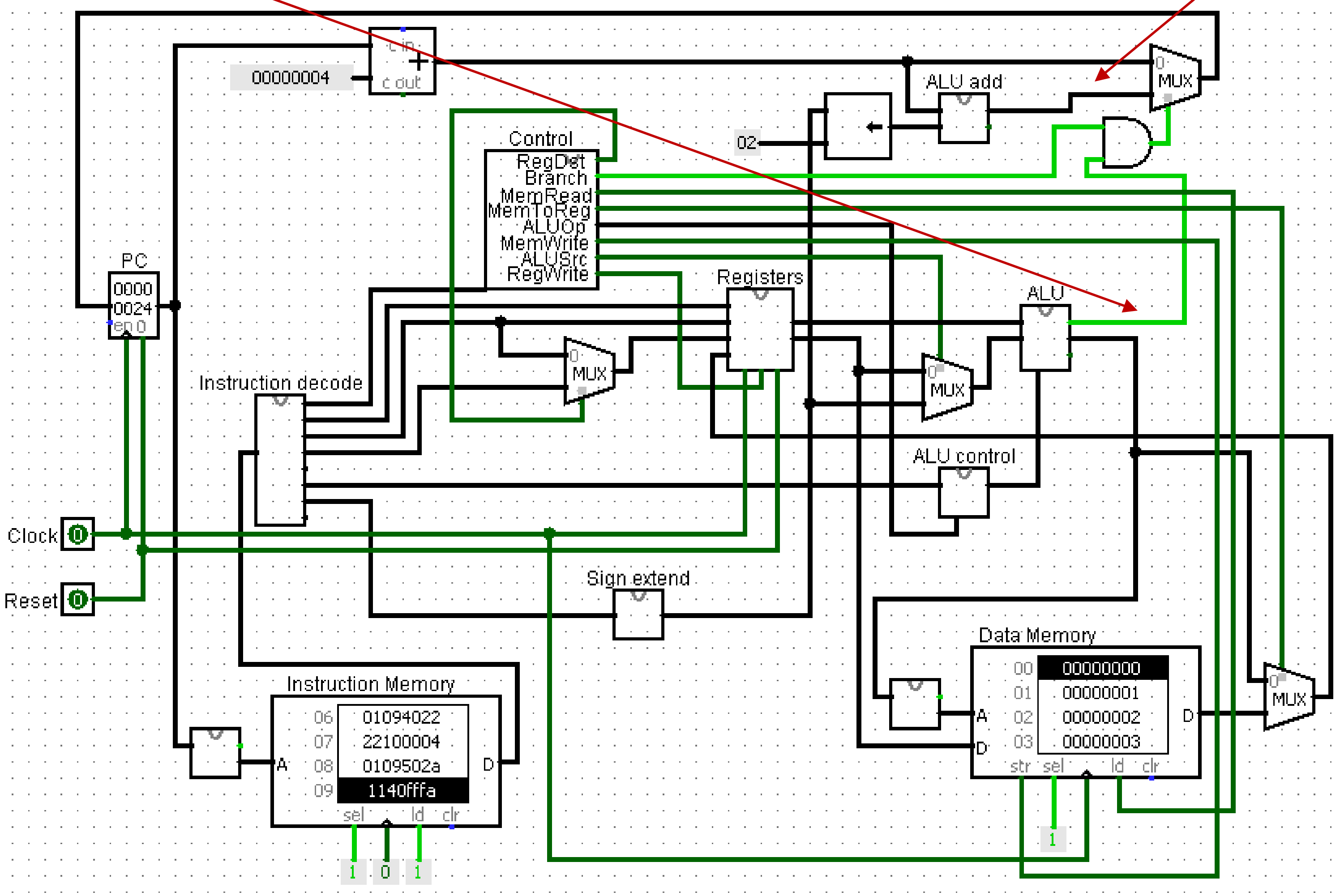
ALU signals for “slt $t2, $t0, $t1” (instr addr 0x08)

- Continue the program execution until the slt instruction at the address 0x08.
- Its signals are like those of sub, but the Result of the ALU is set based on the most significant bit of the subtraction result (in Operation 0111).



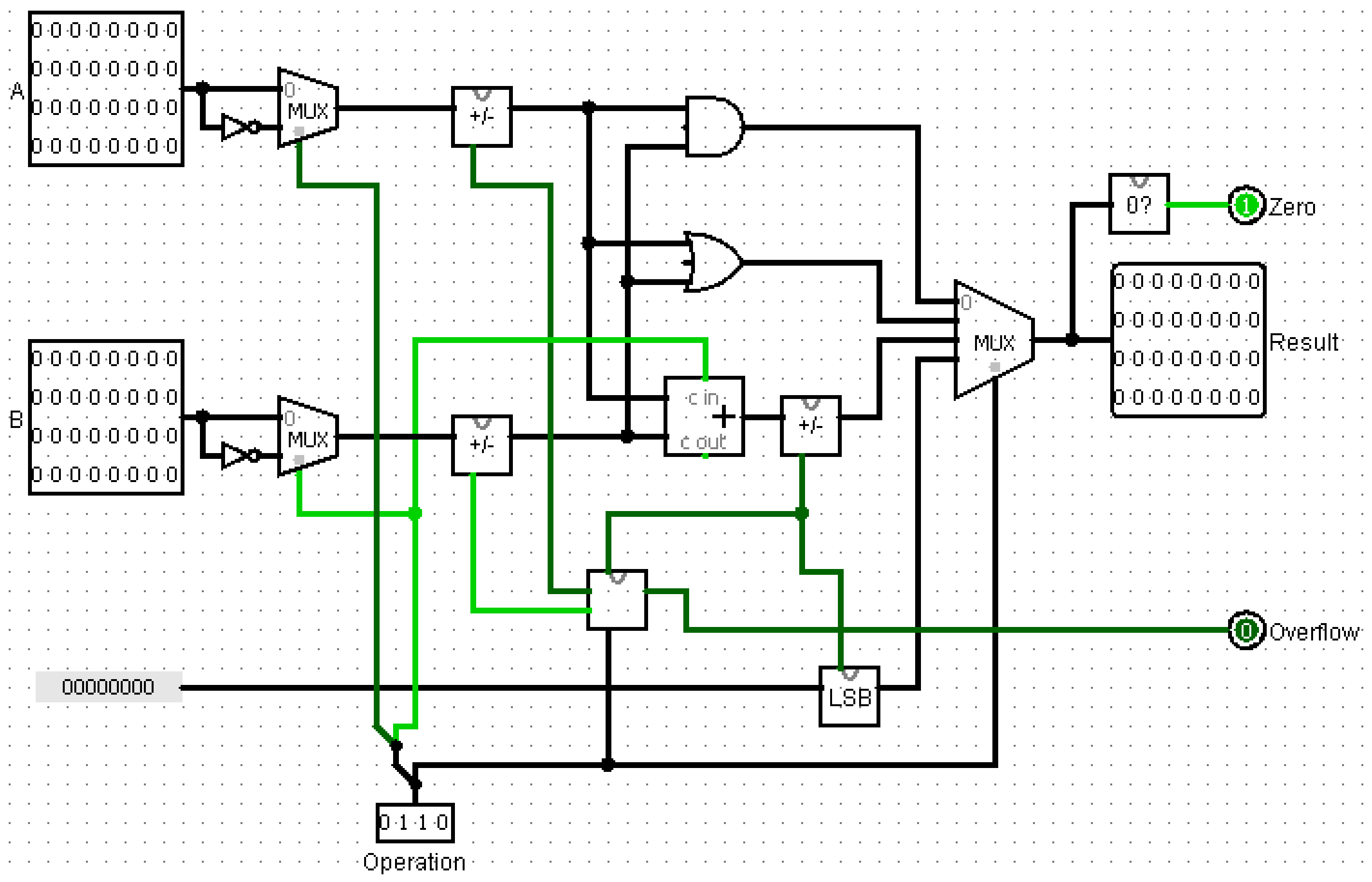
Signals for “beq $t2, $zero, loop” (instr addr 0x09)

- For the next instruction (beq), PC is updated to PC+4 or the branch address if the output Zero of the ALU is 1.



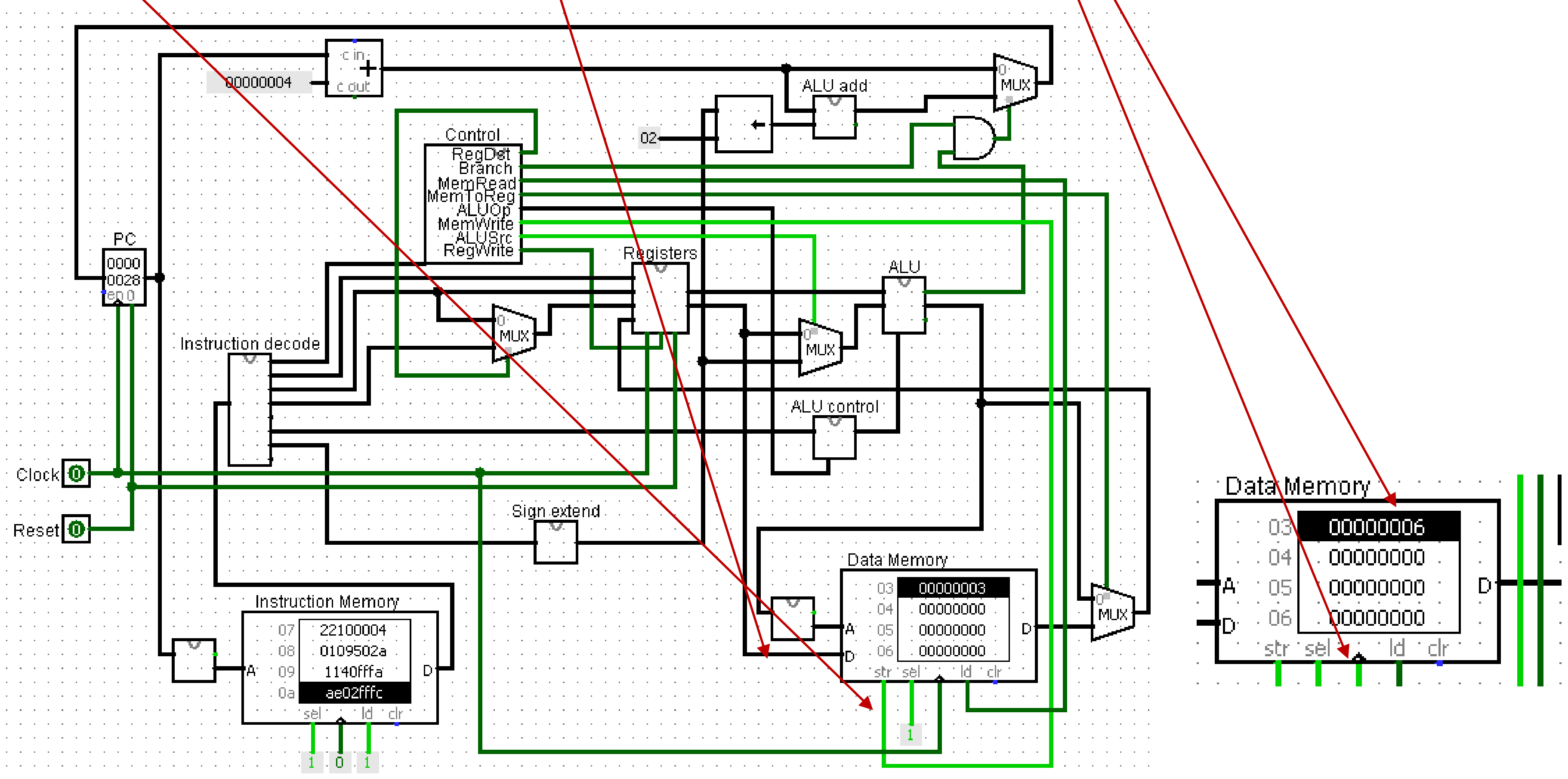
ALU signals for “beq $t2, $zero, loop”

- **$t2 value in A is the result (0) of the previous slt instruction.**



Signals for "sw $v0, -4($s0)"   (instr addr 0x0A)

- Continue the program execution until its last instruction (sw) at the address 0x0A (after the loop execution is completed).
- Summation result of arr (6 in $v0) is forwarded to the Data Memory to be stored at the address 0x03 (at the next rising clock edge).



Conclusions

- You have learnt:
  - ❑building a 4-bit ALU,
  - ❑an implementation of the single-cycle MIPS processor in Logisim,
  - ❑executing instructions in that processor implementation.

Appendix

ALU is used for

- Load/Store: F = add
- Branch: F = subtract
- R-type: F depends on funct field

ALU Control Input	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set on less than
1100	NOR

Appendix

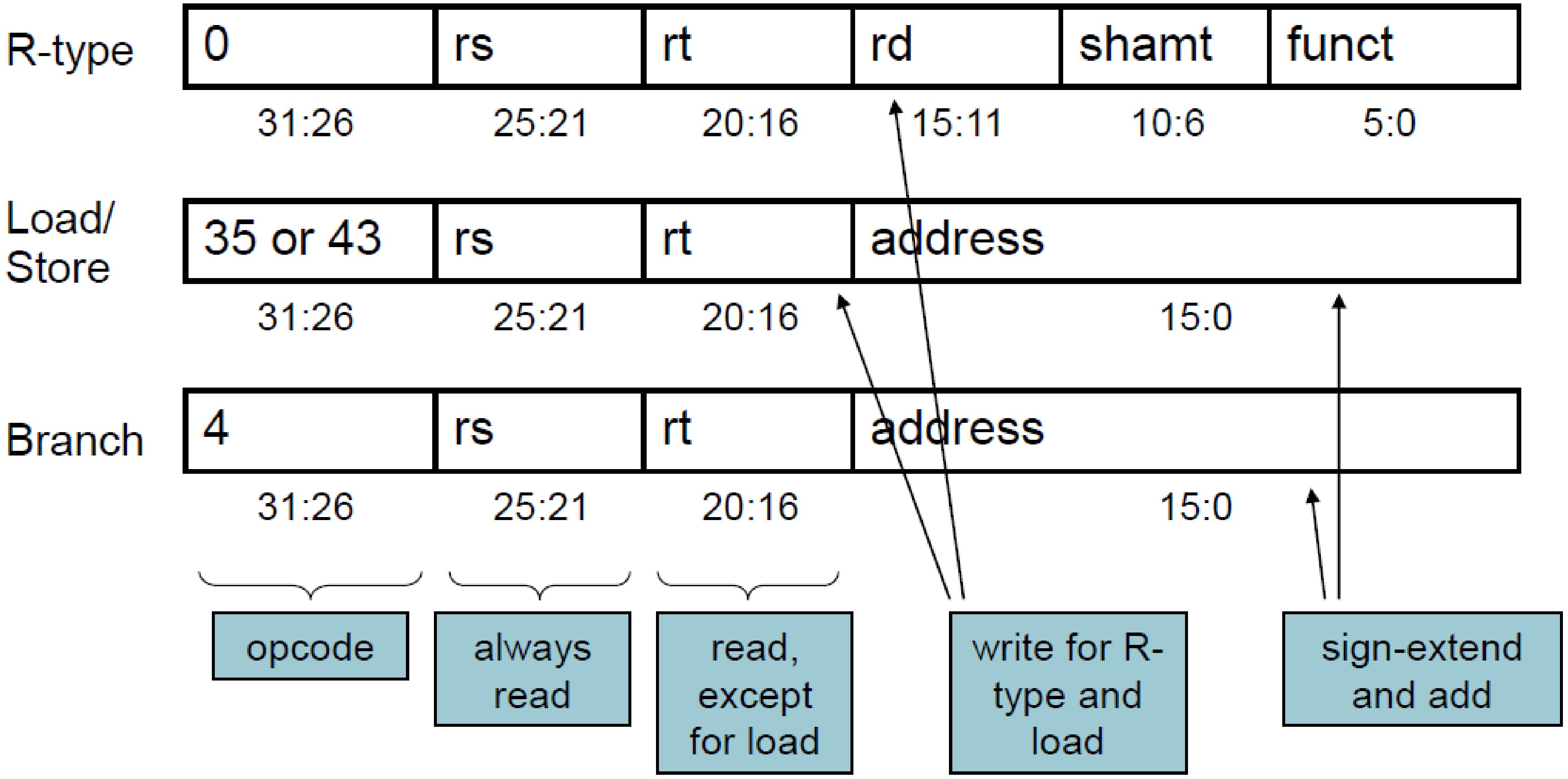
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

input

input

output

Appendix



R-type
0
rs
rt
rd
shamt
funct
31:26
25:21
20:16
15:11
10:6
5:0
Load/ Store
35 or 43
rs
rt
address
31:26
25:21
20:16
15:0
Branch
4
rs
rt
address
31:26
25:21
20:16
15:0
opcode
always read
read, except for load
write for R-type and load
sign-extend and add

Appendix

- ❑ Setting of control lines (output of control unit):

Instruction	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

`sw` & `beq` will not modify any register, it is ensured by making RegWrite to 0
So, we don't care what write register & write data are

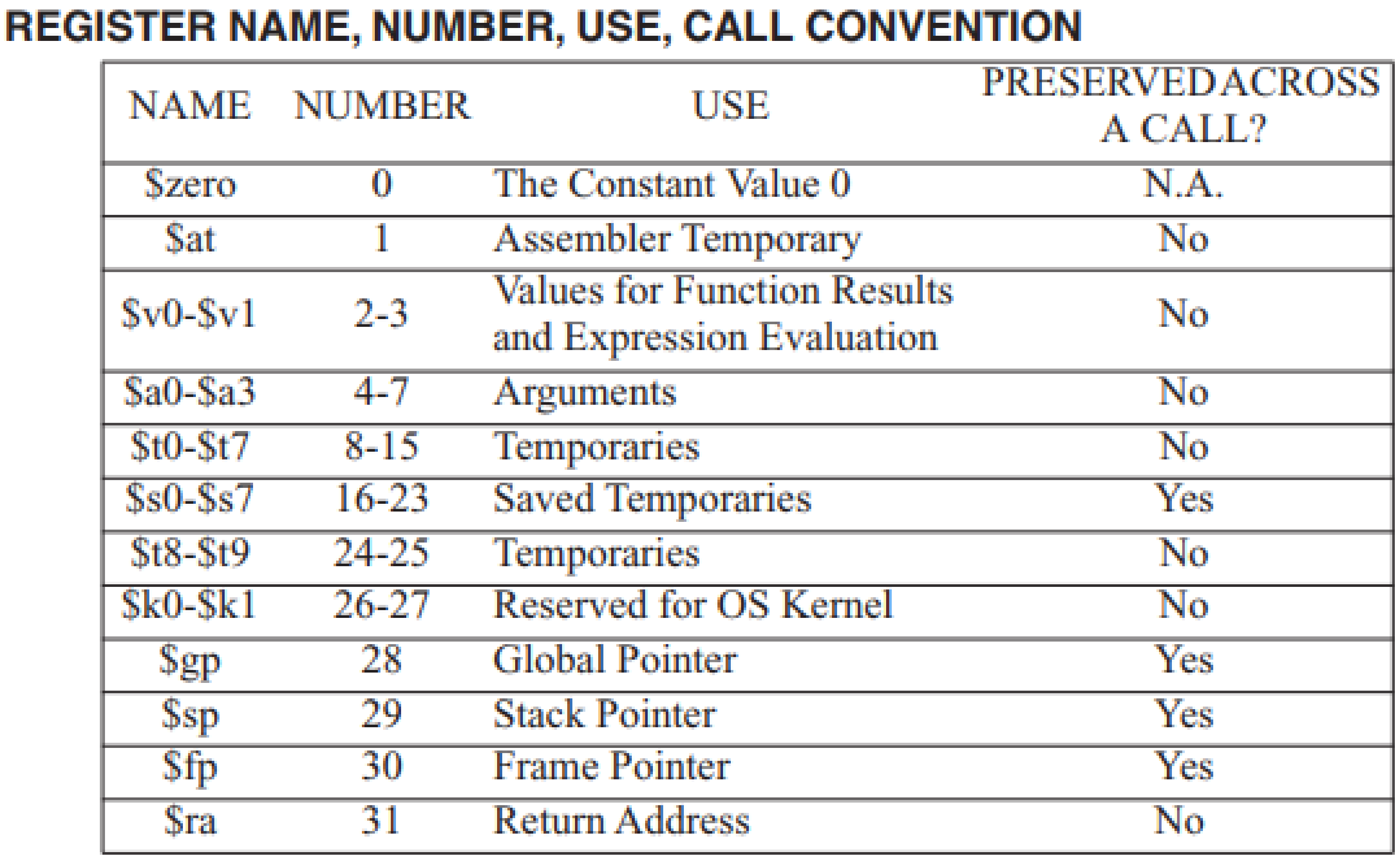
- ❑ Input to control unit (i.e. opcode determines setting of control lines):

Instruction	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0	0	0	0	0	0	0
`lw`	35	1	0	0	0	1	1
`sw`	43	1	0	1	0	1	1
`beq`	4	0	0	0	1	0	0

Appendix

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from **rt** field (bits 20-16)	The register destination number for the Write register comes from **rd** field (bits 15-11)
RegWrite	None	Enable data write to the register specified by the register destination number
ALUSrc	The second ALU operand comes from the second register file output (Read data port 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The next PC picks up the output of the adder that computes PC+4	The next PC picks up the output of the adder that computes the branch target
MemRead	None	Enable read from memory. Memory contents designated by the address are put on the Read data output
MemWrite	None	Enable write to memory. Overwrite the memory contents designated by the address with the value on the Write data input
MemtoReg	Feed the Write data input of the register file with output from ALU	Feed the Write data input of the register file with output from memory

Appendix (from green card)



REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No