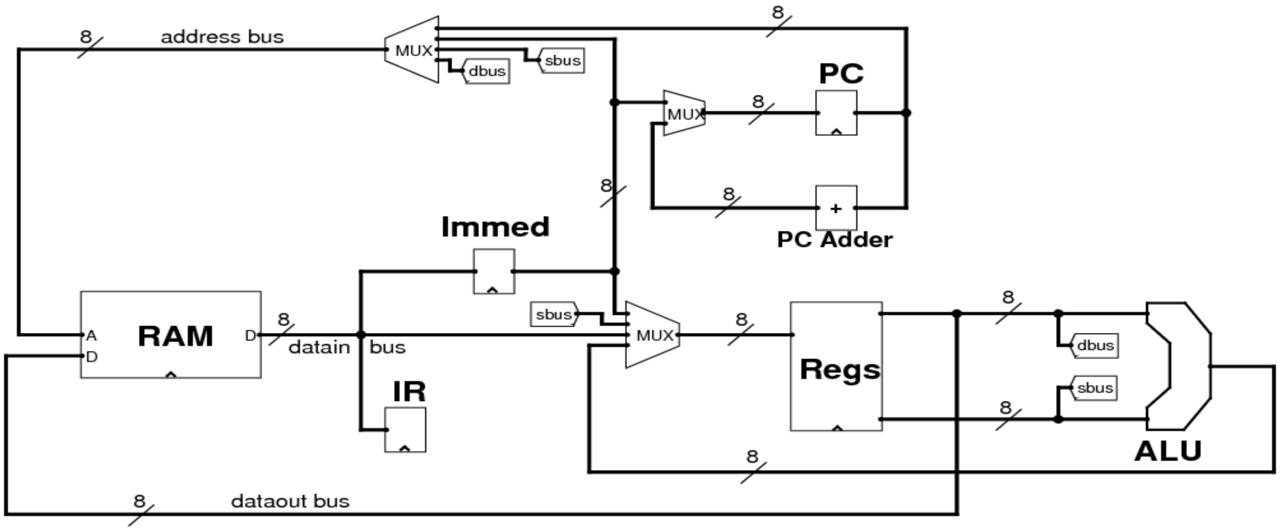
8-Bit CPU

1. Architecture

- The CPU has an 8-bit data bus and an 8-bit address bus, so it can only support 256 bytes of memory to hold both instructions and data.
- Internally, there are four 8-bit registers, R0 to R3, plus an Instruction Register, the Program Counter, and an 8-bit register which holds immediate values.
- The ALU is the same one that we designed last week. It performs the four operations AND, OR, ADD and SUB on two 8-bit values, and supports signed ADDs and SUBs.
- The CPU is a load/store architecture: data has to be brought into registers for manipulation, as the ALU only reads from and writes back to the registers.
- The ALU operations have two operands: one register is a source register, and the second register is both source and destination register, i.e. destination register = destination register OP source register.
- All the jump operations perform absolute jumps; there are no PC-relative branches. There are conditional jumps based on the zeroness or negativity of the destination register, as well as a "jump always" instruction.
- The following diagram shows the datapaths in the CPU:



The *dbus* and *sbus* labels indicate the lines coming out from the register file which hold the value of the destination and source registers.

- Note the data loop involving the registers and the ALU, whose output can only go back into a register.

- The dataout bus is only connected to the *dbus* line, so the only value which can be written to memory is the destination register.
- Also note that there are only 3 multiplexors:

- the address bus multiplexor can get a memory address from the PC, the immediate register (for direct addressing), or from the source or destination registers (for register indirect addressing).
- the PC multiplexor either lets the PC increment, or jump to the value in the immediate register.
- the multiplexor in front of the registers determines where a register write comes from: the ALU, the immediate register, another register or the data bus.

2. Instruction Set

- Half of the instructions in the instruction set fit into one byte:

op1	op2	Rd	Rs
2	2	2	2

- These instructions are identified by a 0 in the most-significant bit in the instruction, i.e. *op1* = 0X.
- The 4 bits of opcode are split into *op1* and *op2*: more details soon.
- *Rd* is the destination register, and *Rs* is the source register.
- The other half of the instruction set are two-byte instructions. The first byte has the same format as above, and it is followed by an 8-bit constant or immediate value:

op1	op2	Rd	Rs	immediate
2	2	2	2	8

- These two-byte instructions are identified by a 1 in the most-significant bit in the instruction, i.e. *op1* = 1X.
- With 4 operation bits, there are 16 instructions:

op1	**op2**	**Mnemonic**	**Purpose**
00	00	AND Rd, Rs	Rd = Rd AND Rs
00	01	OR Rd, Rs	Rd = Rd OR Rs
00	10	ADD Rd, Rs	Rd = Rd + Rs
00	11	SUB Rd, Rs	Rd = Rd - Rs
01	00	LW Rd, (Rs)	Rd = Mem[Rs]
01	01	SW Rd, (Rs)	Mem[Rs] = Rd
01	10	MOV Rd, Rs	Rd = Rs
01	11	NOP	Do nothing
10	00	JEQ Rd, immed	PC = immed if Rd == 0
10	01	JNE Rd, immed	PC = immed if Rd != 0
10	10	JGT Rd, immed	PC = immed if Rd > 0
10	11	JLT Rd, immed	PC = immed if Rd < 0
11	00	LW Rd, immed	Rd = Mem[immed]
11	01	SW Rd, immed	Mem[immed] = Rd
11	10	LI Rd, immed	Rd = immed
11	11	JMP immed	PC = immed

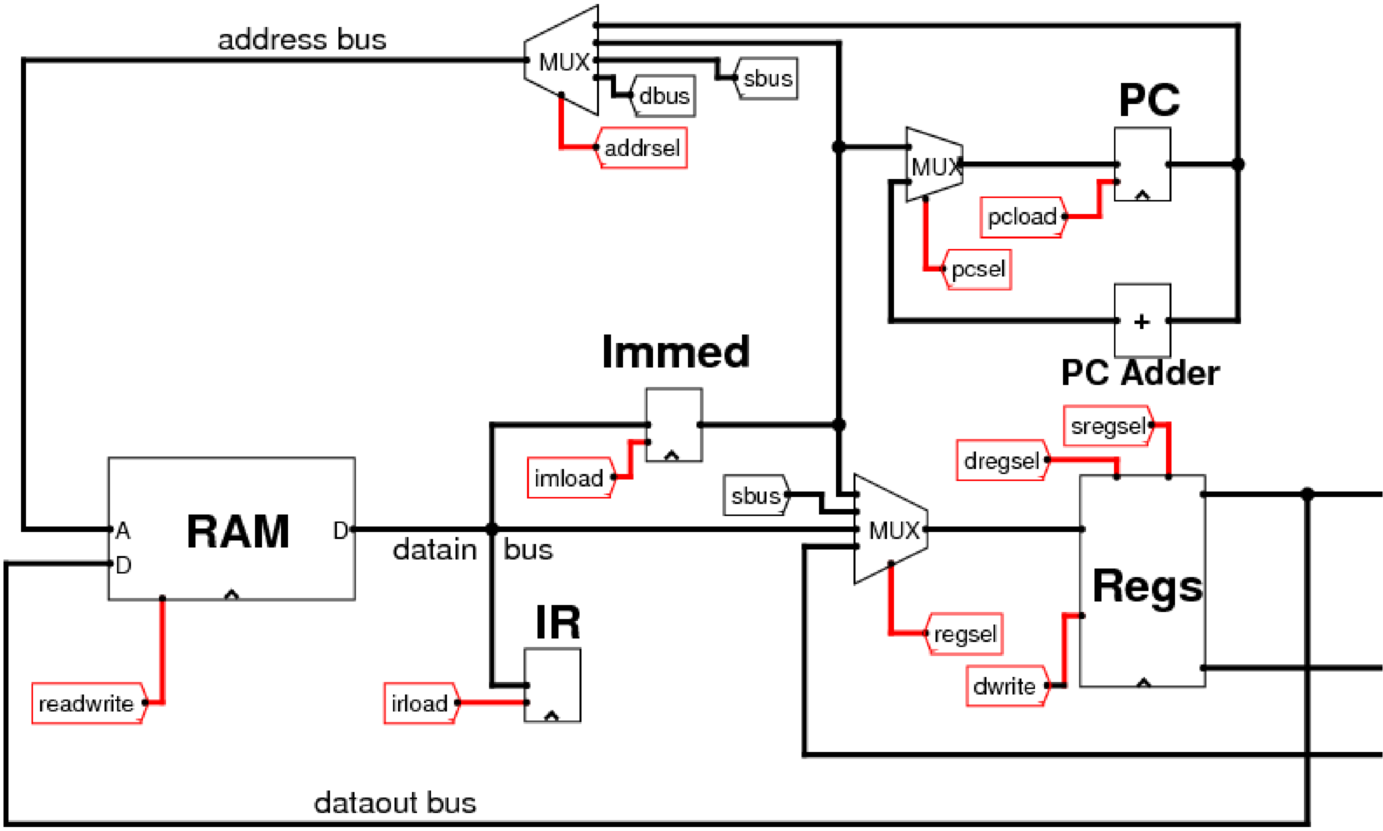
- Note the regularity of the ALU operations and the jump operations: we can feed the *op2* bits directly into the ALU, and use *op2* to control the branch decision.
- The rest of the instruction set is less regular, which will require special decoding for certain of the 16 instructions.

3. Instruction Phases

- The CPU internally has three phases for the execution of each instruction.
- On phase 0, the instruction is fetched from memory and stored in the Instruction Register.
- On phase 1, if the fetched instruction is a two-byte instruction, the second byte is fetched from memory and stored in the Immediate Register. For one-byte instructions, nothing occurs in phase 1.
- On phase 2, everything else is done as required, which can include:
  - an ALU operation, reading from two registers.
  - a jump decision which updates the PC.
  - a register write.
  - a read from a memory location.
  - a write to a memory location.
- After phase 2, the CPU starts the next instruction in phase 0.
- The control logic will be simple for the phase 0 work, not difficult for the phase 1 work, but complicated for the phase 2 work.

4 CPU Control Lines

Below is the main CPU diagram again, this time with the control lines shown.



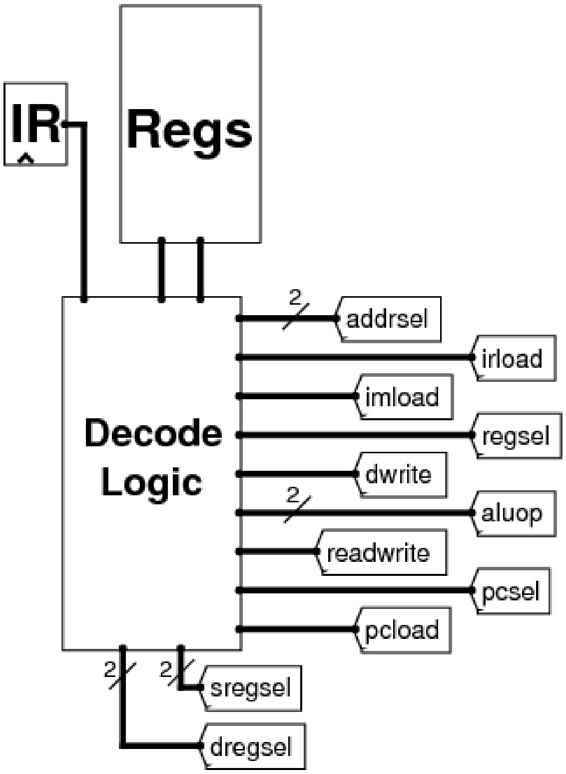
There are several 1-bit control lines:

- *pcsel*, increment PC or load the jump value from the Immediate Register.
- *pcload*, load the PC with a new value, or don't load a new value.
- *irload*, load the Instruction Register with a new instruction.
- *imload*, load the Immediate Register with a new value.

  - *readwrite*, read from memory, or write to memory.
  - *dwrite*, write a value back to a register, or don't write a value.

  There are also several 2-bit control lines:

  - *addrsel*, select an address from the PC, the Immediate Register, the source register or the destination register.
  - *regsel*, select a value to write to a register from the Immediate Register, another register, the data bus or from the ALU.
  - *dregsel* and *sregsel*, select two registers whose values are sent to the ALU.
  - *aluop*, which are the *op2* bits that control the operation of the ALU.
- The values for all of these control lines are generated by the Decode Logic, which gets as input the value from the Instruction Register, and the zero & negative lines of the destination register.



An Example Program to Run in the CPU

- It's time to see an example program written for this CPU.
- In memory starting at location 0x80 is a list of 8-bit numbers; the last number in the list is 0.
- We want a program to sum the numbers, store the result into memory location 0x40, and loop indefinitely after that.
- We have 4 registers to use. They are allocated as follows:
    - R0 holds the pointer to the next number to add.
    - R1 holds the running sum.
    - R2 holds the next number to add to the running sum.
    - R3 is used as a temporary register.

The assembly code for the program.

```
 LI  R1,0x00          # Set running sum to zero
      LI  R0,0x80     # Start at beginning of list
loop: LW  R2, (R0)    # Get the next number
      JEQ R2, end     # Exit loop if number == 0
      ADD R1, R2      # Add number to running sum
      LI  R3, 0x01    # Put 1 into R3, so we can do
      ADD R0, R3      # R0++
      JMP loop        # Loop back
end:  SW  R1, 0x40    # Store result at address 0x40
inf:  JMP inf         # Infinite loop
```

Converting to machine code, here are the hex values to put into memory starting at location 0:

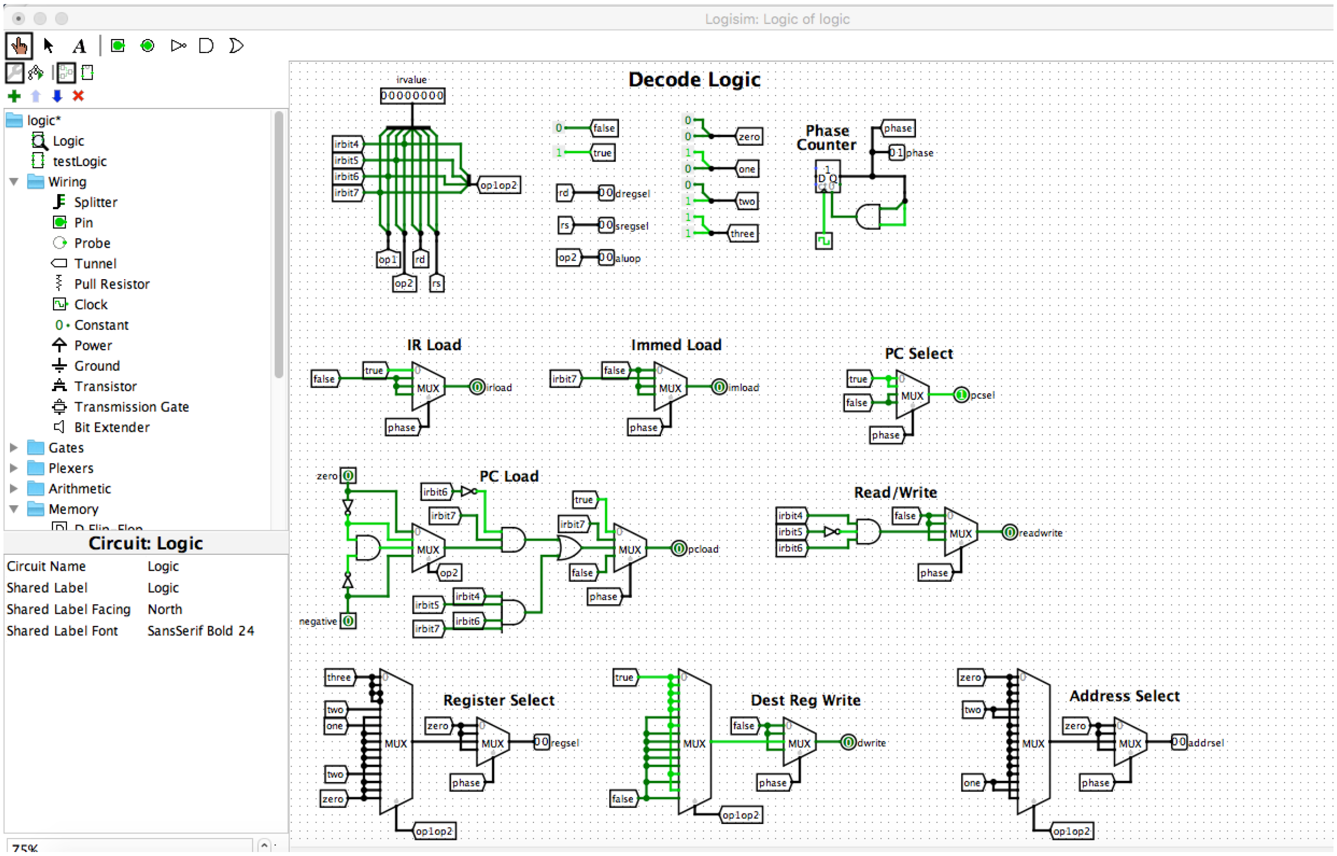
LI R1,0x00	e4 00
LI R0,0x80	e0 80
LW R2, (R0)	48
JEQ R2, end	88 0d
ADD R1, R2	26
LI R3, 0x01	ec 01
ADD R0, R3	23
JMP loop	ff 04
SW R1, 0x40	d4 40
JMP inf	ff 0f

- With the CPU loaded up into Logisim, and the memory loaded with the above data values, we can start the program running.
- Watch the phases of operation. Watch the IR get loaded with an instruction.
- Watch the Immediate Register get loaded with a value.
- On the LW instruction, watch as the *sbus* value is selected to be placed on the address bus, and the datain value is written to the destination register.
- On ALU instructions, watch the *sbus* and *dbus* values, the *aluop*, and the result which is written back into the destination register.

- On the JEQ instruction, watch the value of N and Z into the Decode Logic, and the resulting *pcsel* and *pcload* values.

Create the Logic

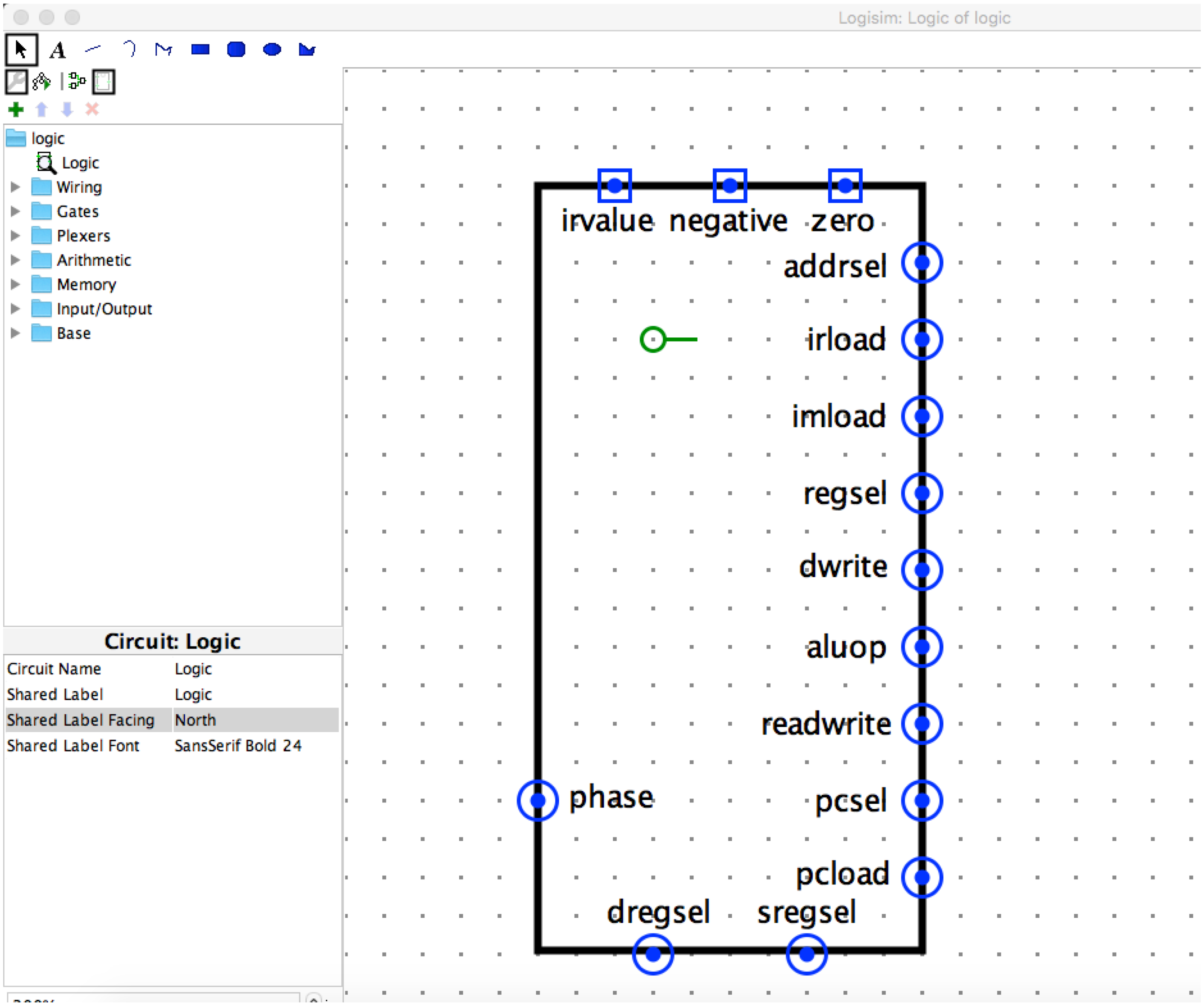
1. Create a new Logisim circuit. Open the given logism file logism.circ so you have the folloing in your screen:



Circuit: Logic

Circuit Name	Logic
Shared Label	Logic
Shared Label Facing	North
Shared Label Font	SansSerif Bold 24

2. Create the following 11x21 chip from it:

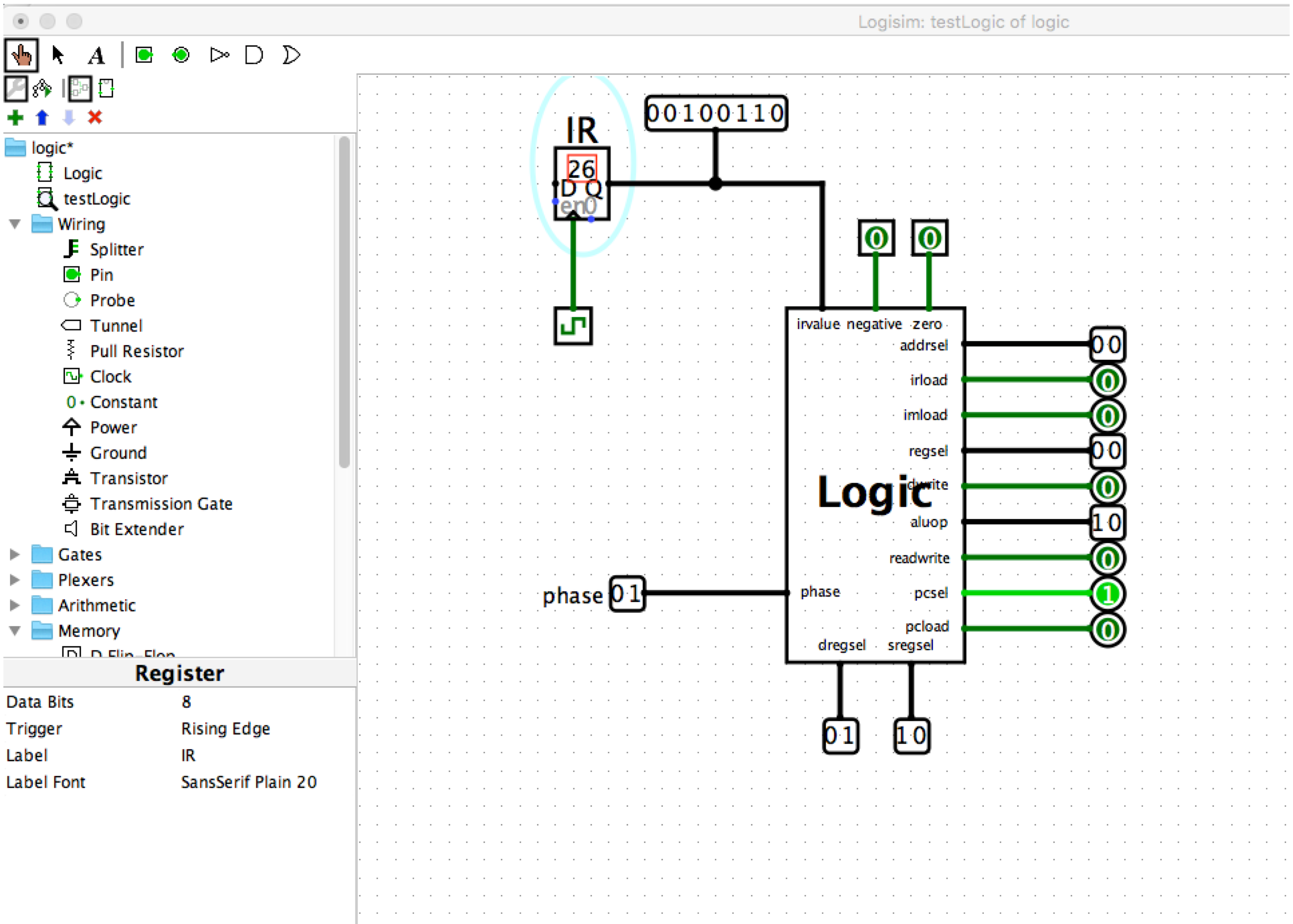


Circuit: Logic

Circuit Name	Logic
Shared Label	Logic
Shared Label Facing	North
Shared Label Font	SansSerif Bold 24

3. Create a new Circuit, call it testLogic.circ, and load (type) into the IR register the instruction 26, which is ADD R1 and R2. Enable the ticks.

4. Take a screen capture with the phase showing 10 Upload the testLOGICphase03.jpeg.



Register

Data Bits	8
Trigger	Rising Edge
Label	IR
Label Font	SansSerif Plain 20