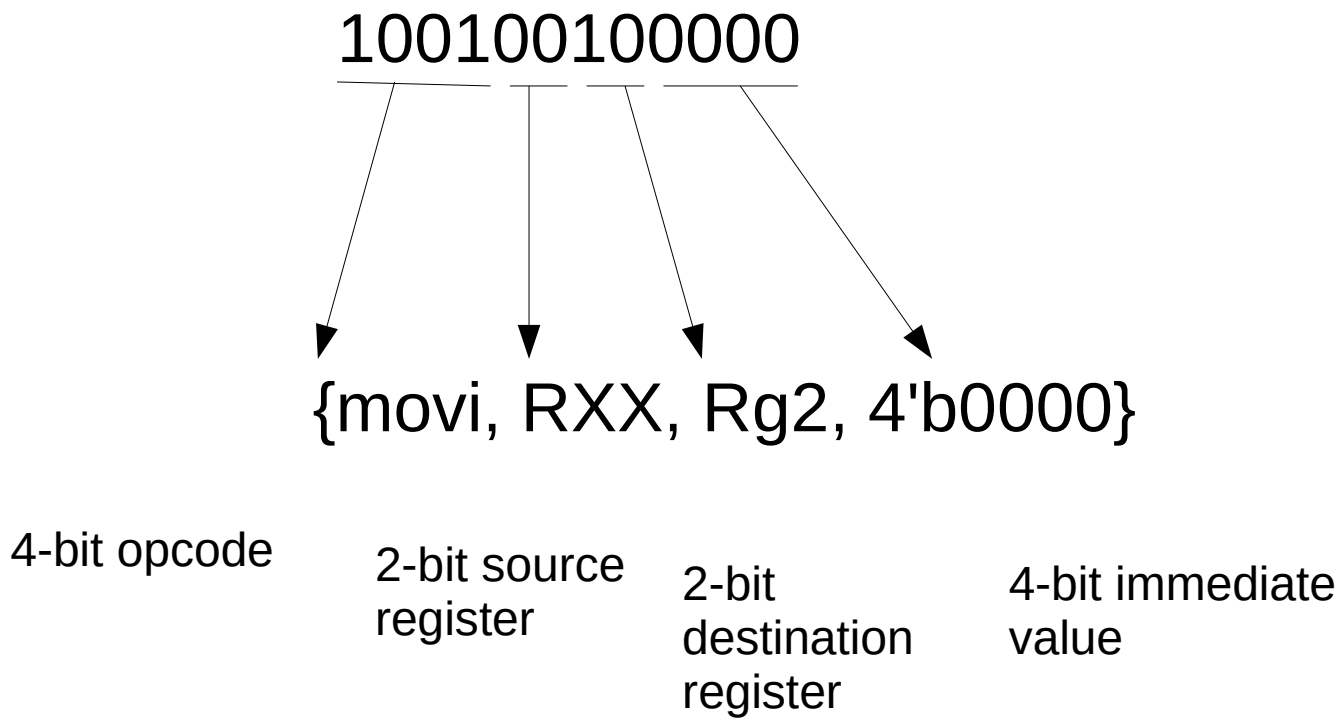
E15 Processor

Machine language

bin	hex
100100100000	920
100100010001	911
101001100000	a60
111100100111	f27
001100001110	30e
000000000000	000

Machine language bin	hex	Assembly language
100100100000	920	{movi, RXX, Rg2, 4'b0000}
100100010001	911	{movi, RXX, Rg1, 4'b0001}
101001100000	a60	{add, Rg1, Rg2, 4'b0000}
111100100111	f27	{cmpi, RXX, Rg2, 4'b0111}
001100001110	30e	{jnz, RXX, RXX, 4'b1110}
000000000000	000	{jmp, RXX, RXX, 4'b0000}


100100100000
{movi, RXX, Rg2, 4'b0000}
4-bit opcode
2-bit source register
2-bit destination register
4-bit immediate value

This E15 assembly language....

```
{movi, RXX, Rg2, 4'b0000}
{movi, RXX, Rg1, 4'b0001}
{add,  Rg1, Rg2, 4'b0000}
{cmpi, RXX, Rg2, 4'b0111}
{jnz,  RXX, RXX, 4'b1110}
{jmp,  RXX, RXX, 4'b0000}
```

... corresponds (roughly) to the following pseudocode:

```
int rg2 = 0;
int rg1 = 1;
while (true) {
   rg2 += rg1;
   if (rg2 == 7)
      break;
}
while (true) {}
```

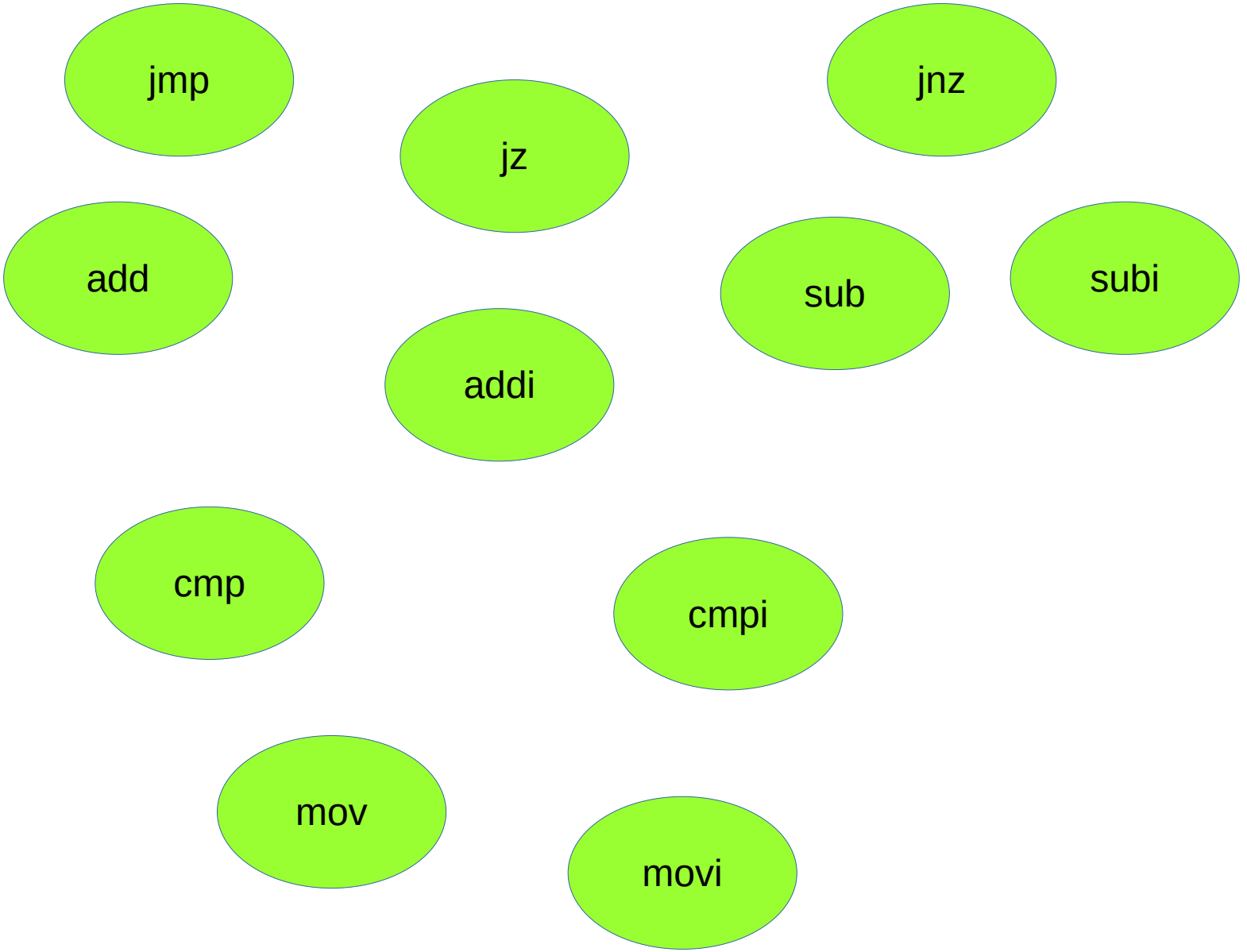
E15 instruction set

```
jmp  = 4'b0000,
jz   = 4'b0010,
jnz =  4'b0011,
movi = 4'b1001,
mov  = 4'b1000,
addi = 4'b1011,
add  = 4'b1010,
subi = 4'b1101,
sub  = 4'b1100,
cmpi = 4'b1111,
cmp  = 4'b1110;
```

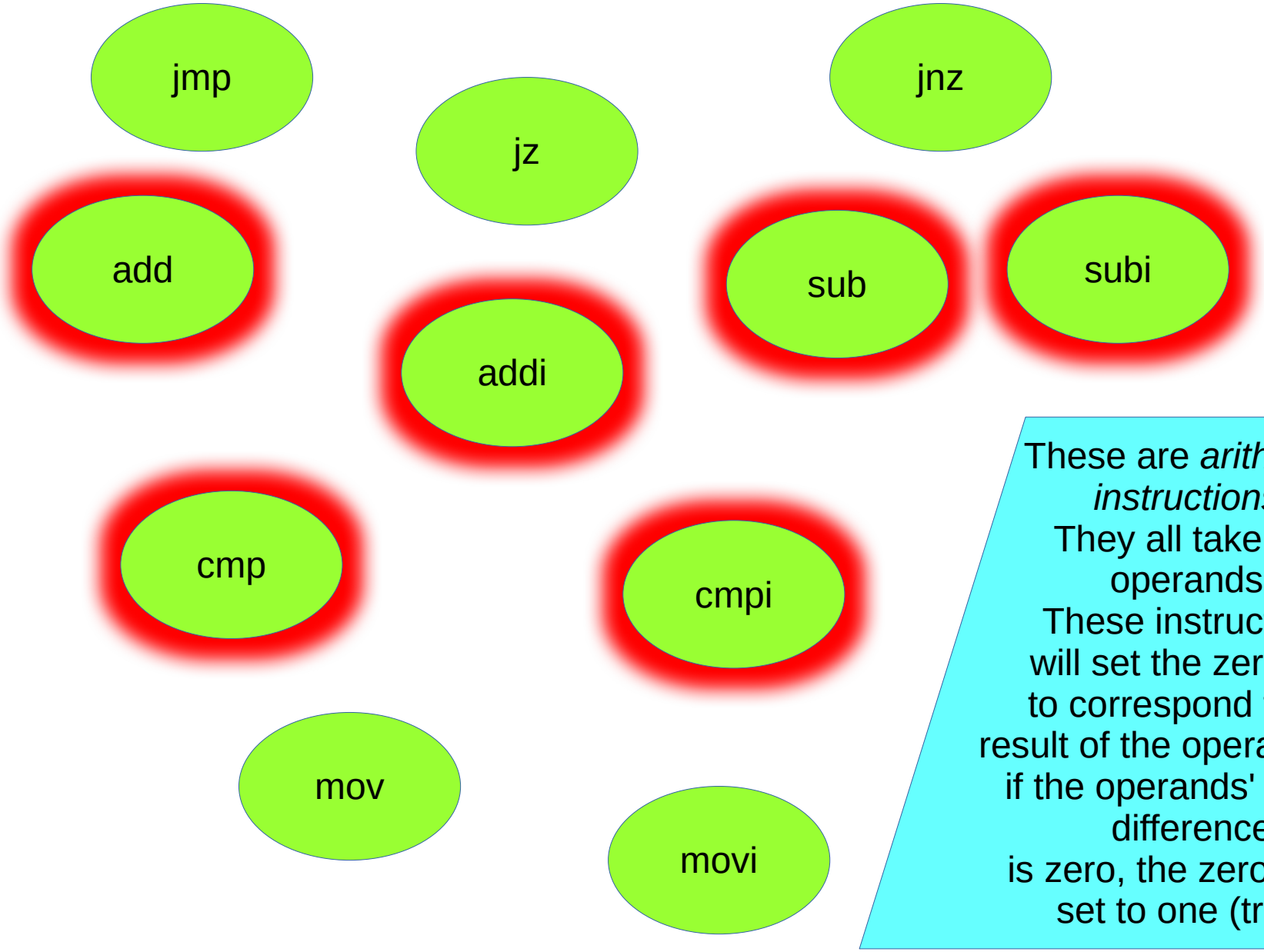
jump
jump if zero
jump if not zero
move immediate
move
add immediate
add
subtract immediate
subtract
compare immediate
compare

```
jmp  = 4'b0000,
    Add the given immediate value to the program counter
jz   = 4'b0010,
    Add the given immediate value to the program counter if  the ZERO flag is true
jnz =  4'b0011,
    Add the given immediate value to the program counter if  the ZERO flag is
    false
movi = 4'b1001,
    Move the given immediate value into the given destination
mov  = 4'b1000,
    Move the given source register value into the given destination
addi = 4'b1011,
    Add the given immediate value into the given destination
add  = 4'b1010,
    Add the given source register value into the given destination
subi = 4'b1101,
    Subtract the given immediate value from the given destination
sub  = 4'b1100,
    Subtract the given source register value from the given destination
cmpi = 4'b1111,
    Compare the given immediate value to the given destination register and set
    the ZERO flag accordingly
cmp  = 4'b1110;
    Compare the given source register value to the given destination register and
    set the ZERO flag accordingly
```

A tour of the E15 opcodes

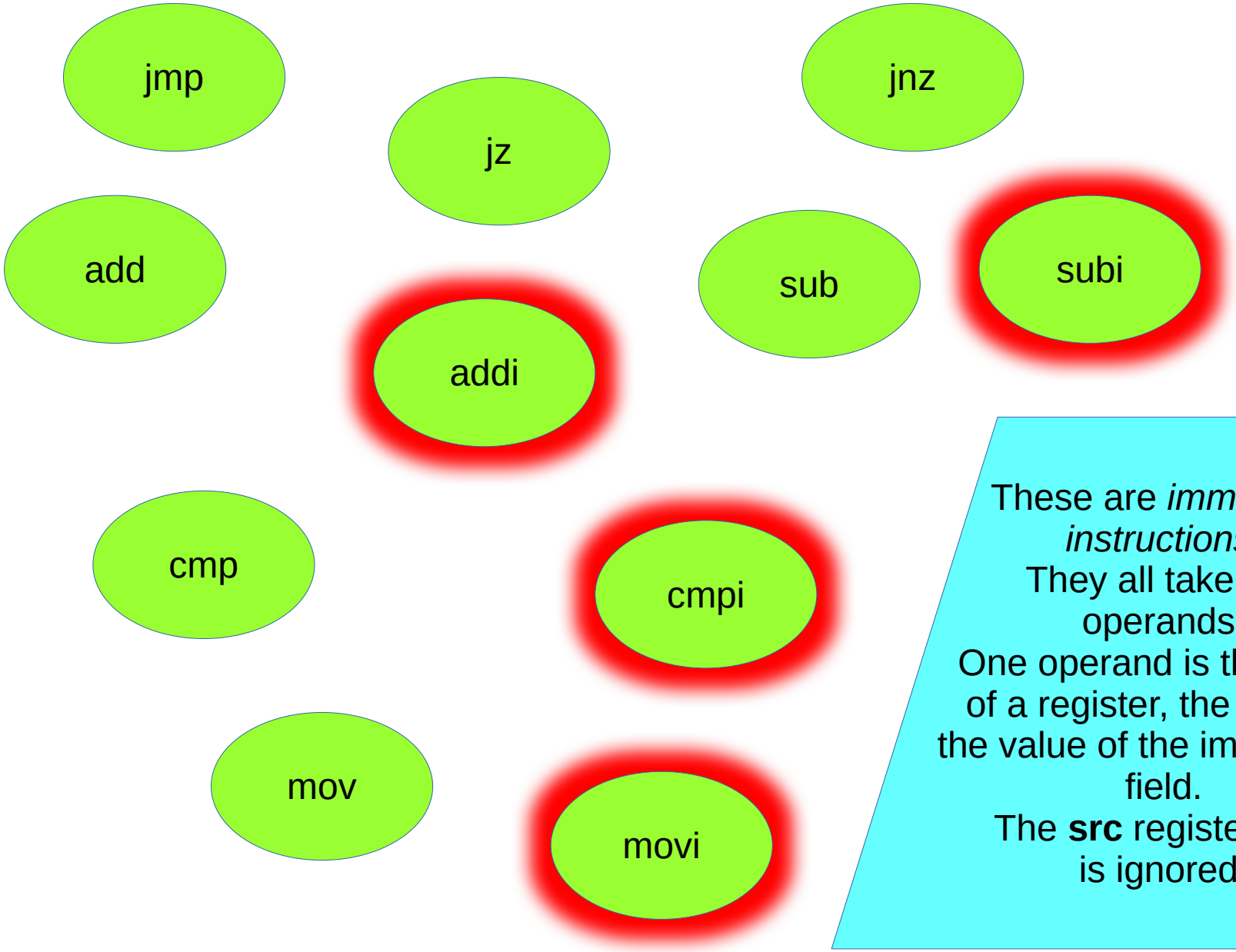


A tour of the E15 opcodes

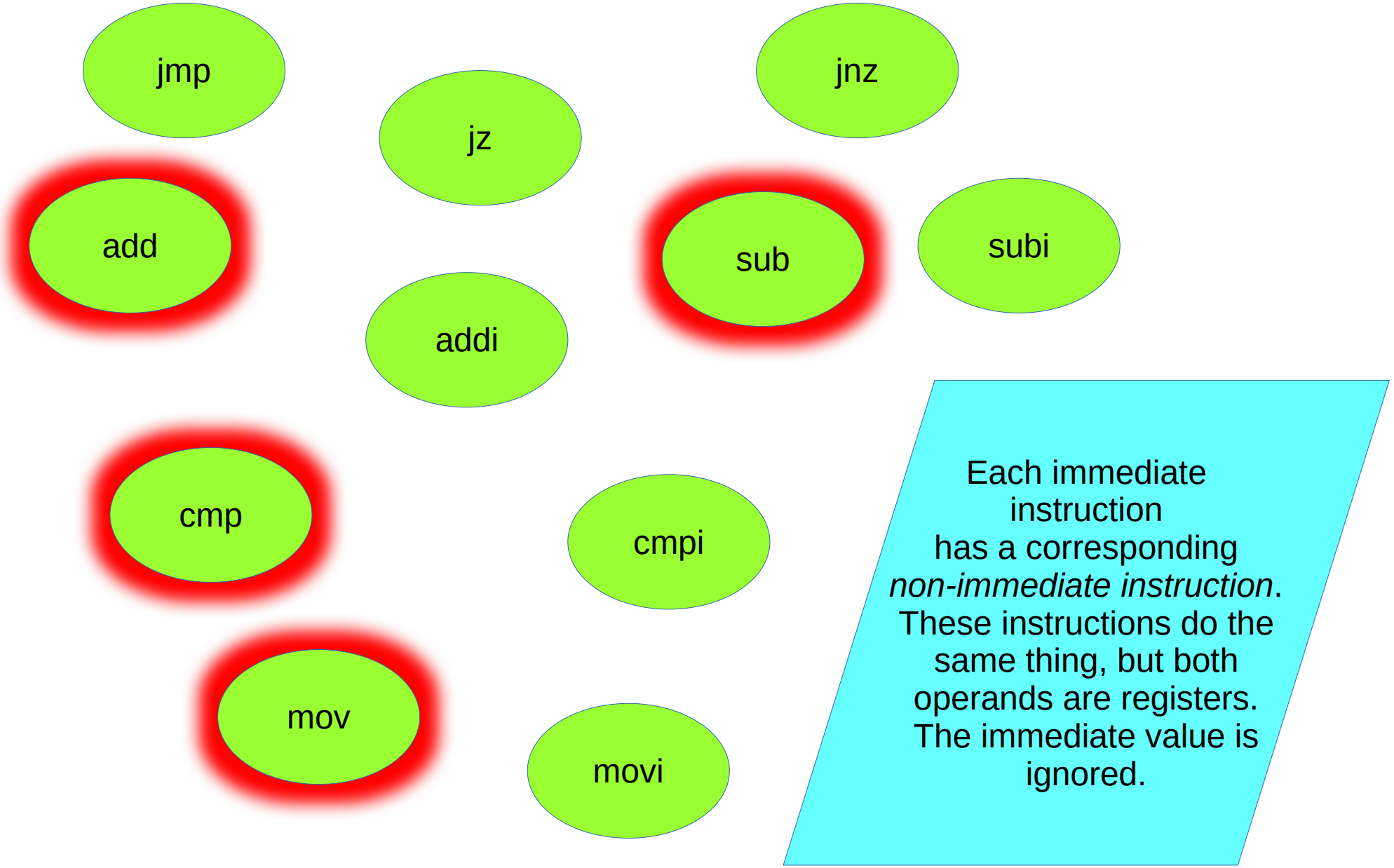


These are *arithmetic instructions*.
They all take two operands.
These instructions will set the zero flag to correspond to the result of the operation, i.e. if the operands' sum or difference is zero, the zero flag is set to one (true).

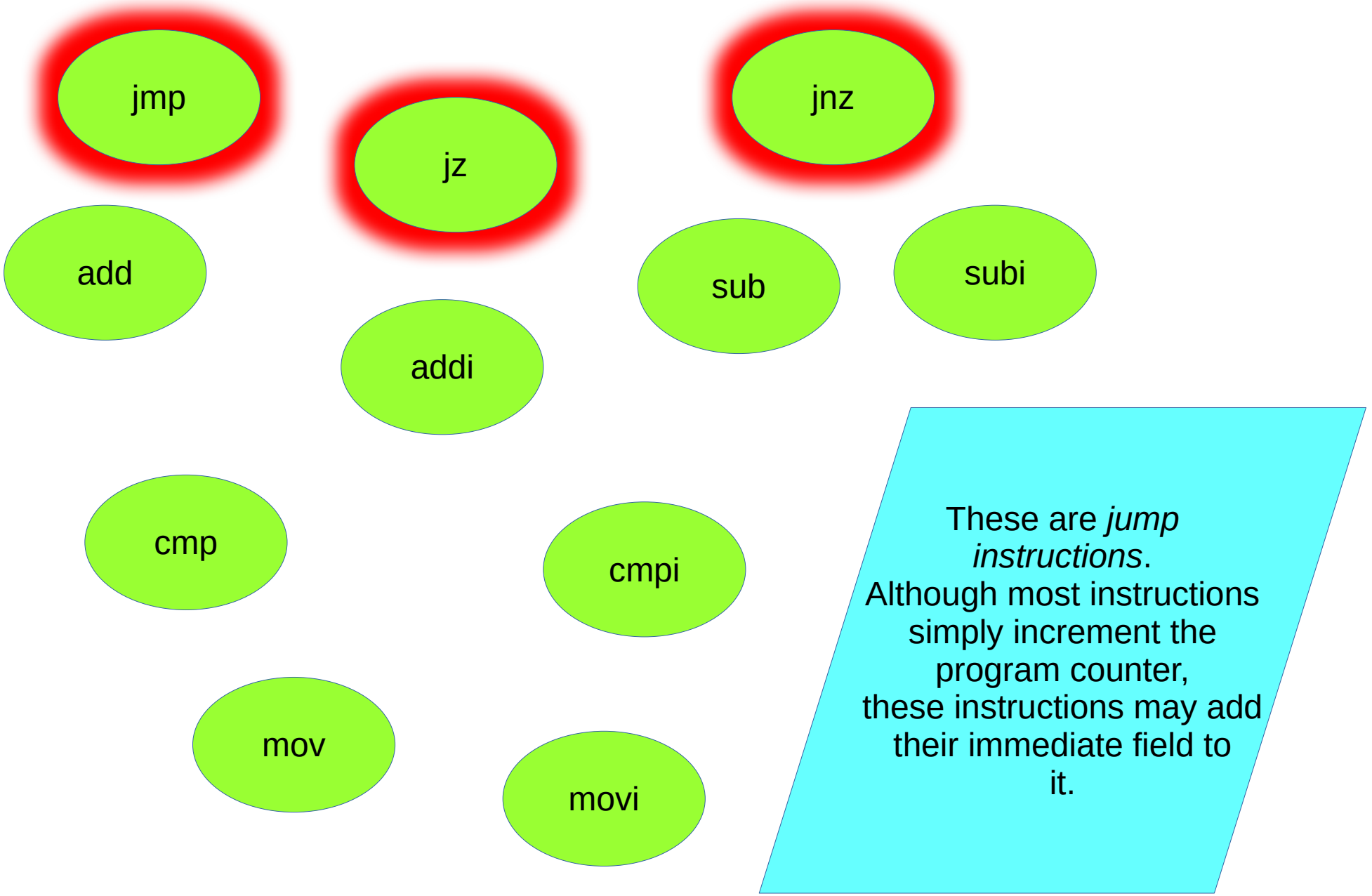
A tour of the E15 opcodes



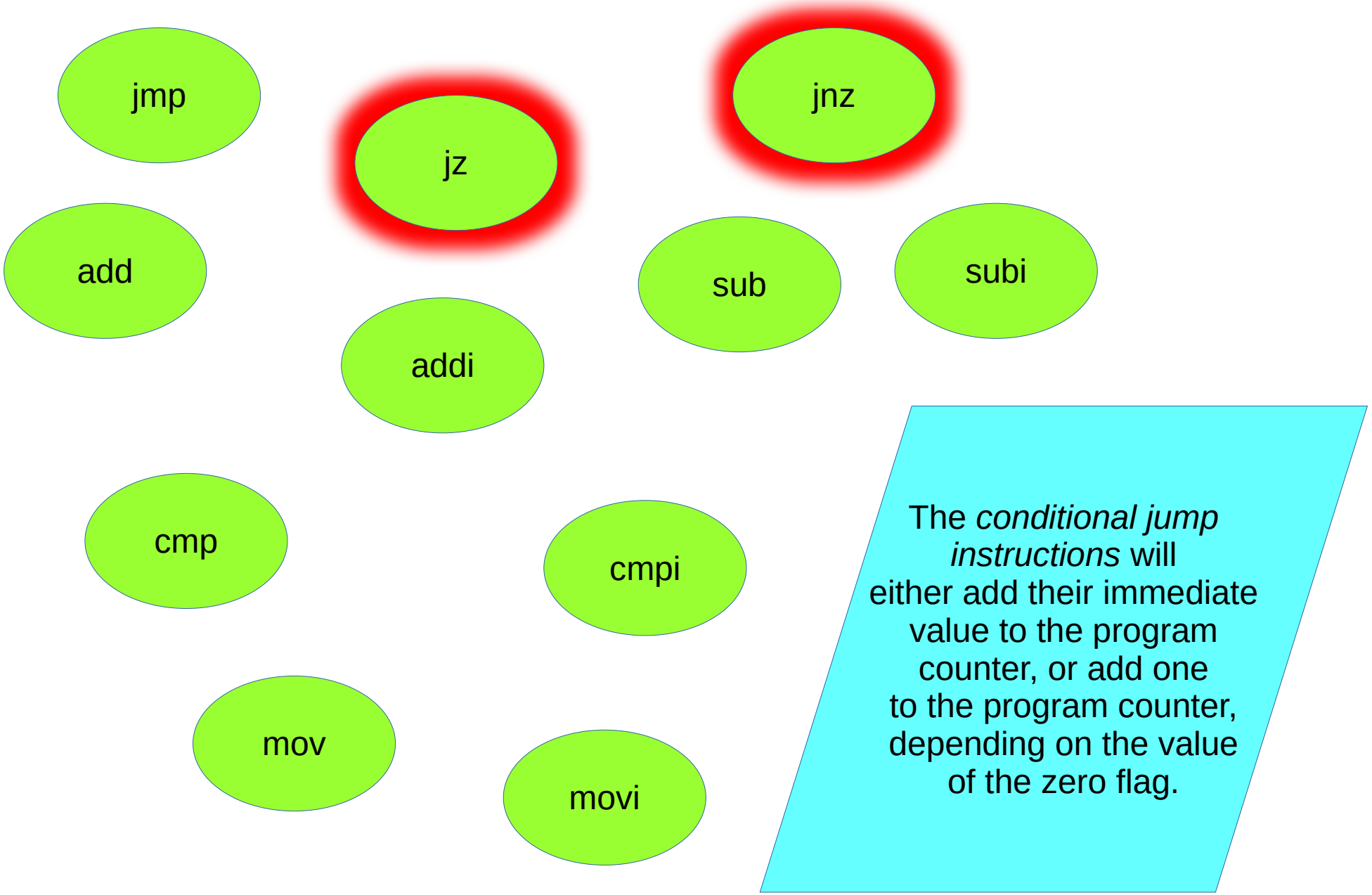
A tour of the E15 opcodes



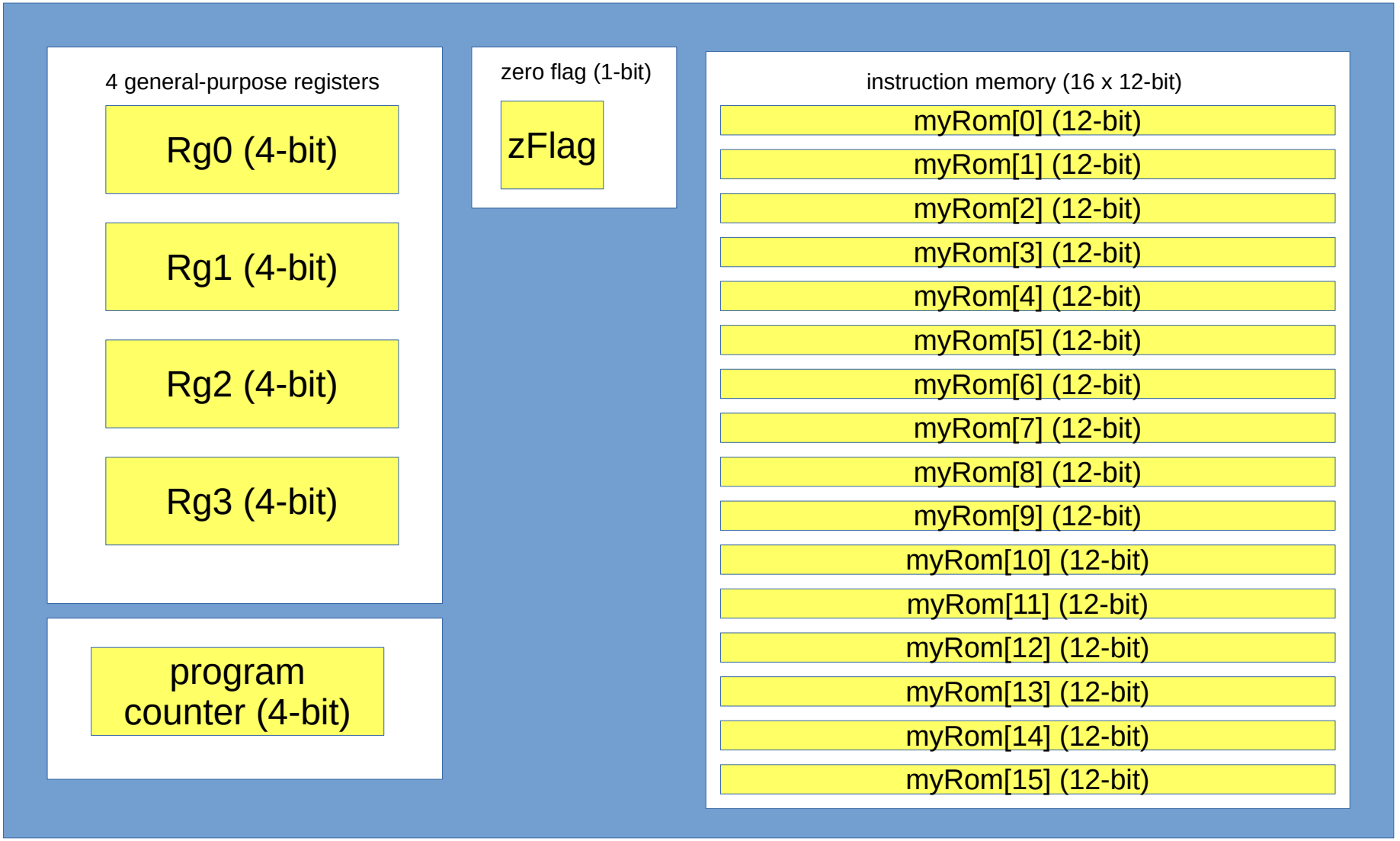
A tour of the E15 opcodes



A tour of the E15 opcodes



What's inside the E15 processor?

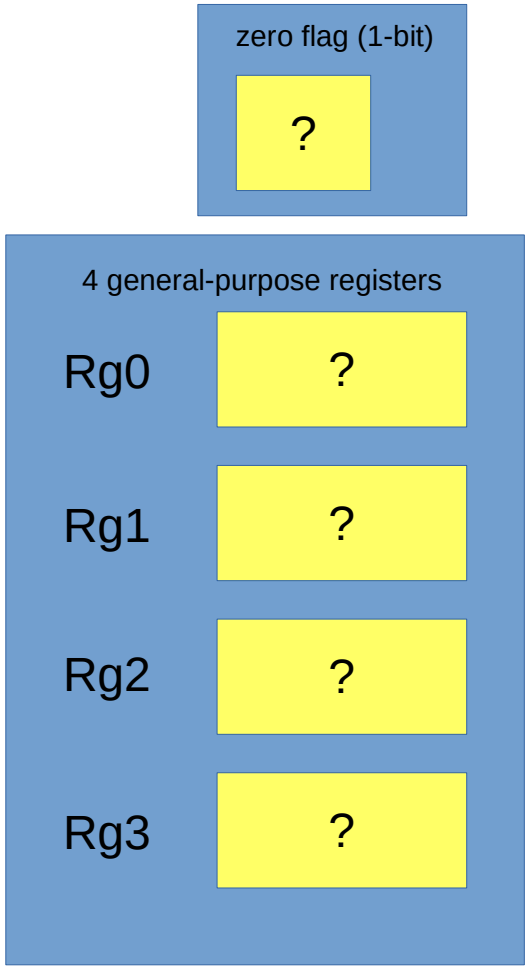


Here's a more complicated E15 program. What does it do?

```
/*           OPCODE SRC  DST  IMMDATA */
myROM[0] = {movi, RXX, Rg1, 4'b0011};
myROM[1] = {add,  Rg1, Rg1, 4'b0000};
myROM[2] = {add,  Rg1, Rg1, 4'b0000};
myROM[3] = {movi, RXX, Rg2, 4'b1100};
myROM[4] = {cmp,  Rg1, Rg2, 4'b0000};
myROM[5] = {jz,   RXX, RXX, 4'b0011};
myROM[6] = {movi, RXX, Rg2, 4'b0001};
myROM[7] = {jmp,  RXX, RXX, 4'b0000};
myROM[8] = {movi, RXX, Rg2, 4'b0000};
myROM[9] = {jmp,  RXX, RXX, 4'b0000};
myROM[10] = {jmp,  RXX, RXX, 4'b0000};
myROM[11] = {jmp,  RXX, RXX, 4'b0000};
myROM[12] = {jmp,  RXX, RXX, 4'b0000};
myROM[13] = {jmp,  RXX, RXX, 4'b0000};
myROM[14] = {jmp,  RXX, RXX, 4'b0000};
myROM[15] = {jmp,  RXX, RXX, 4'b0000};
```

Note that we assign instruction values to all 16 ROM cells, not just those that our program will actually use.

How does a program execute?

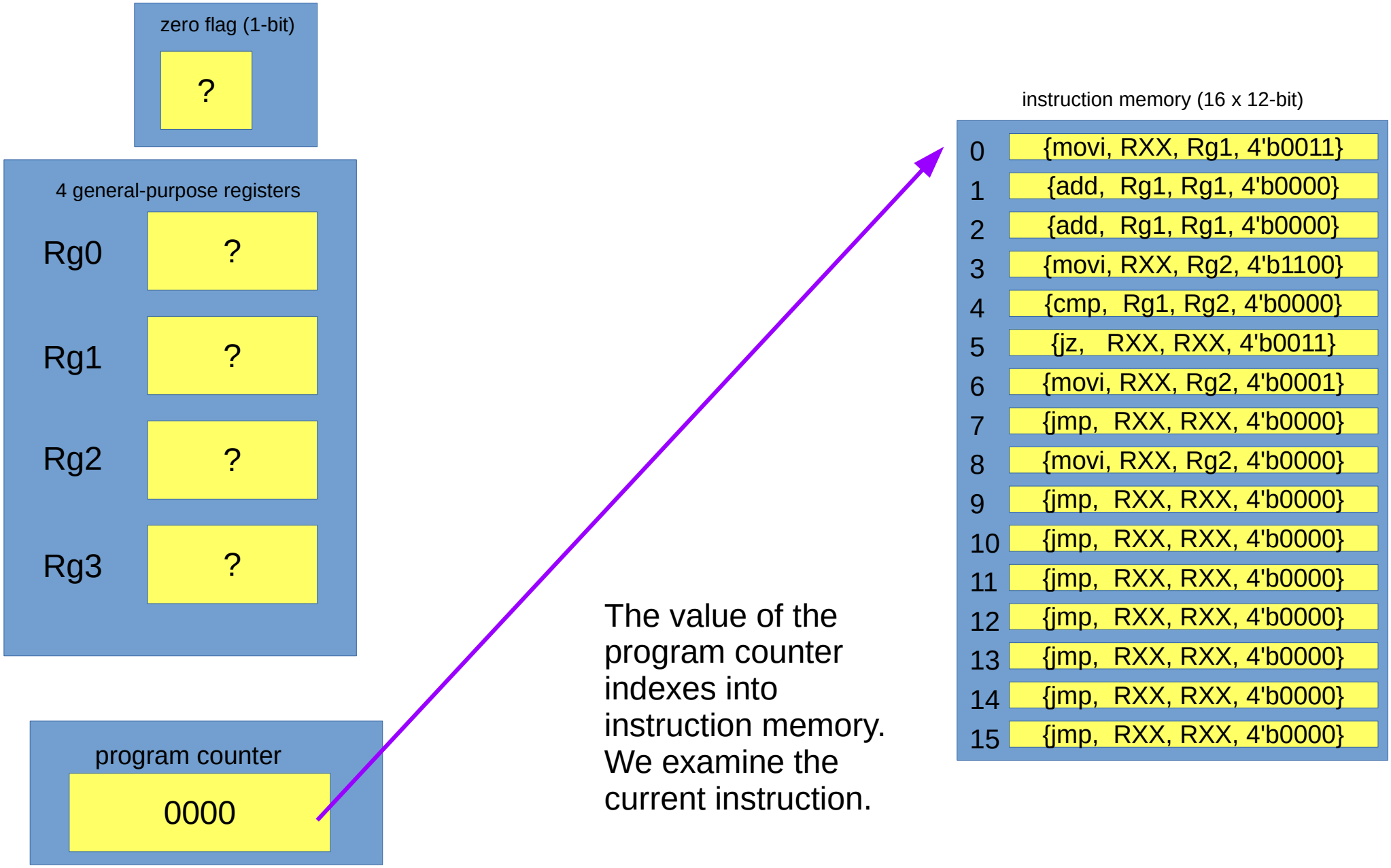


Initially, the program counter is zero and the ROM contains our program.
Other registers have unknown values.

instruction memory (16 x 12-bit)

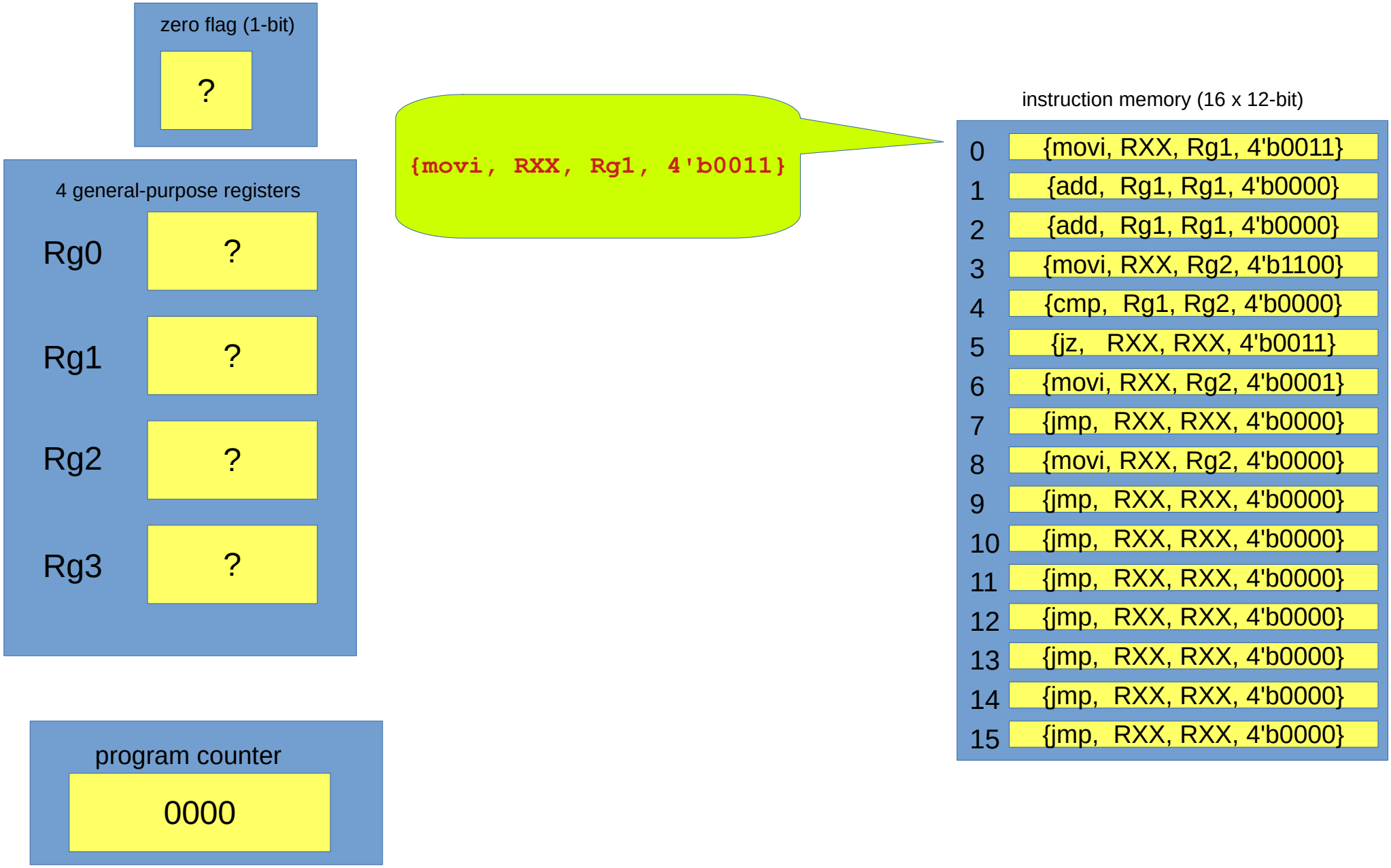
0	{movi, RXX, Rg1, 4'b0011}
1	{add, Rg1, Rg1, 4'b0000}
2	{add, Rg1, Rg1, 4'b0000}
3	{movi, RXX, Rg2, 4'b1100}
4	{cmp, Rg1, Rg2, 4'b0000}
5	{jz, RXX, RXX, 4'b0011}
6	{movi, RXX, Rg2, 4'b0001}
7	{jmp, RXX, RXX, 4'b0000}
8	{movi, RXX, Rg2, 4'b0000}
9	{jmp, RXX, RXX, 4'b0000}
10	{jmp, RXX, RXX, 4'b0000}
11	{jmp, RXX, RXX, 4'b0000}
12	{jmp, RXX, RXX, 4'b0000}
13	{jmp, RXX, RXX, 4'b0000}
14	{jmp, RXX, RXX, 4'b0000}
15	{jmp, RXX, RXX, 4'b0000}

How does a program execute?

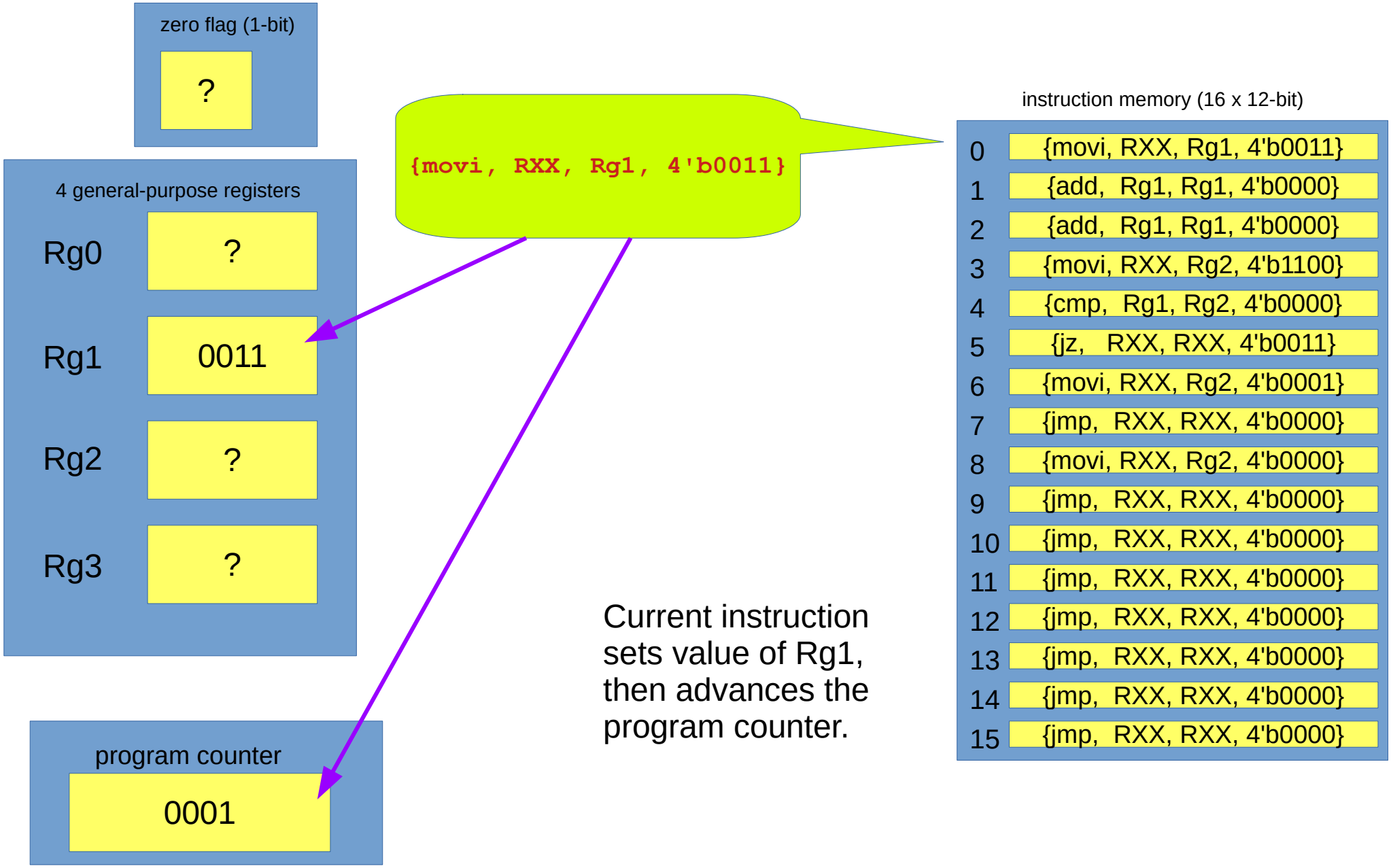


The value of the program counter indexes into instruction memory. We examine the current instruction.

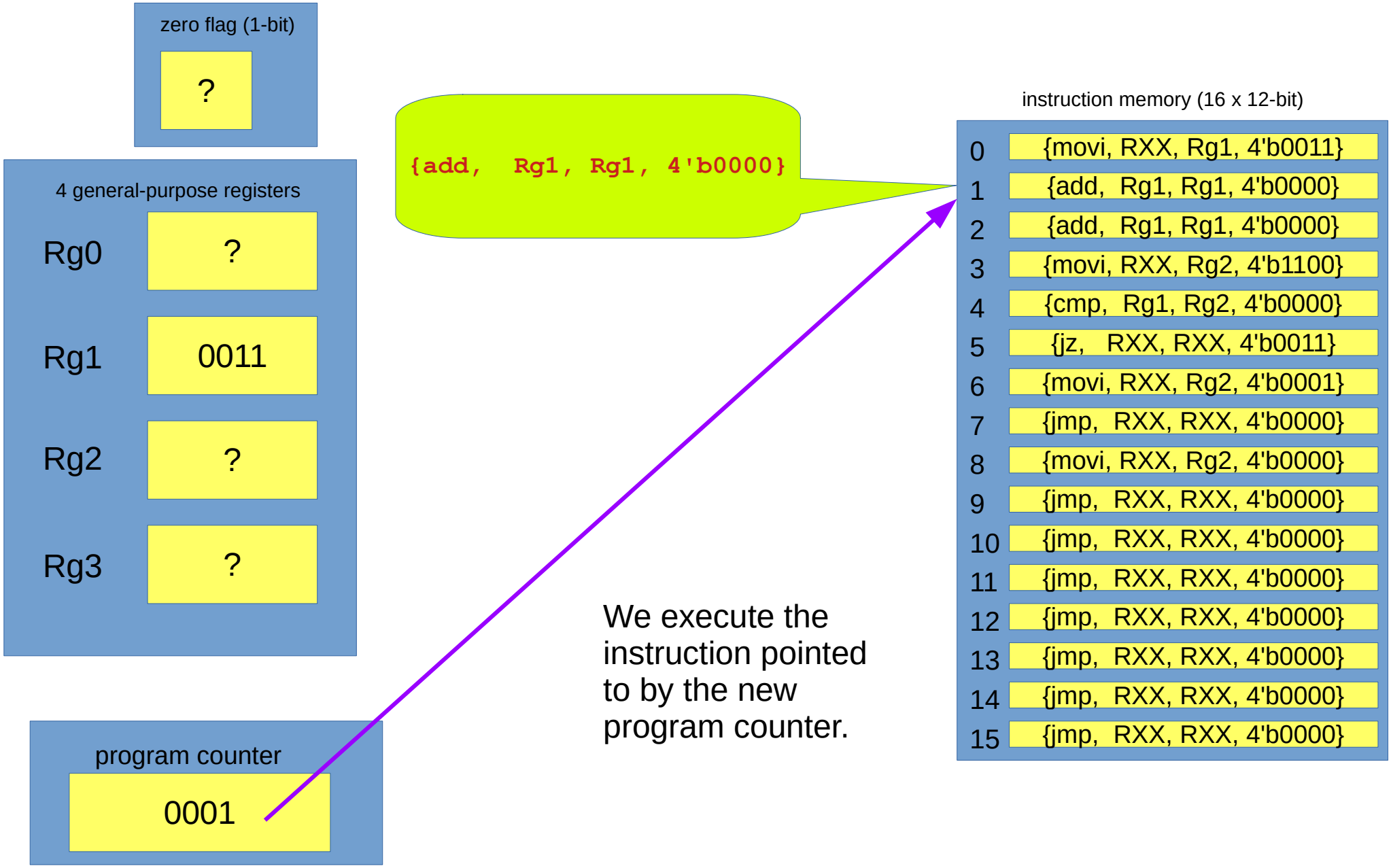
How does a program execute?



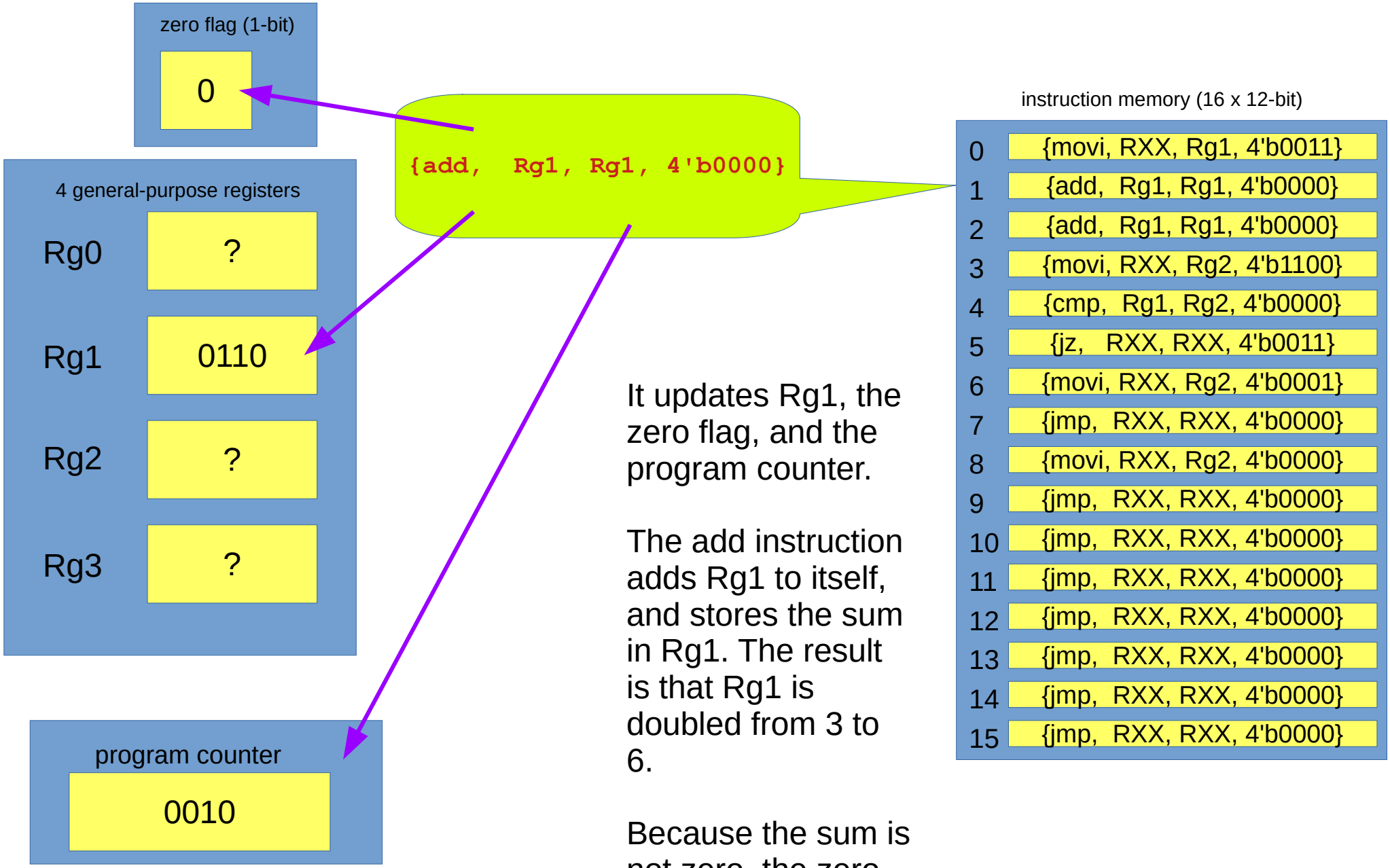
How does a program execute?



How does a program execute?



How does a program execute?

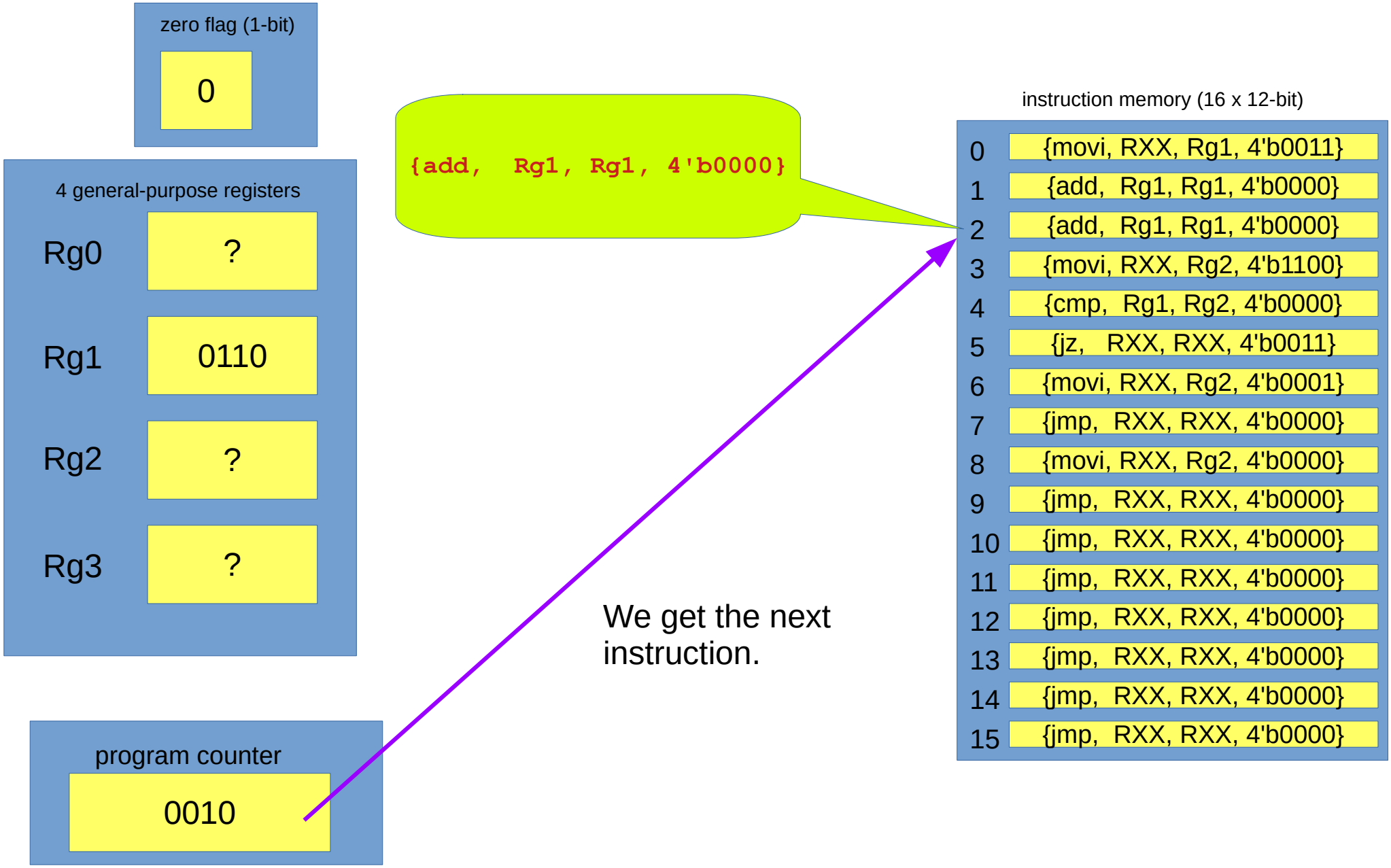


It updates Rg1, the zero flag, and the program counter.

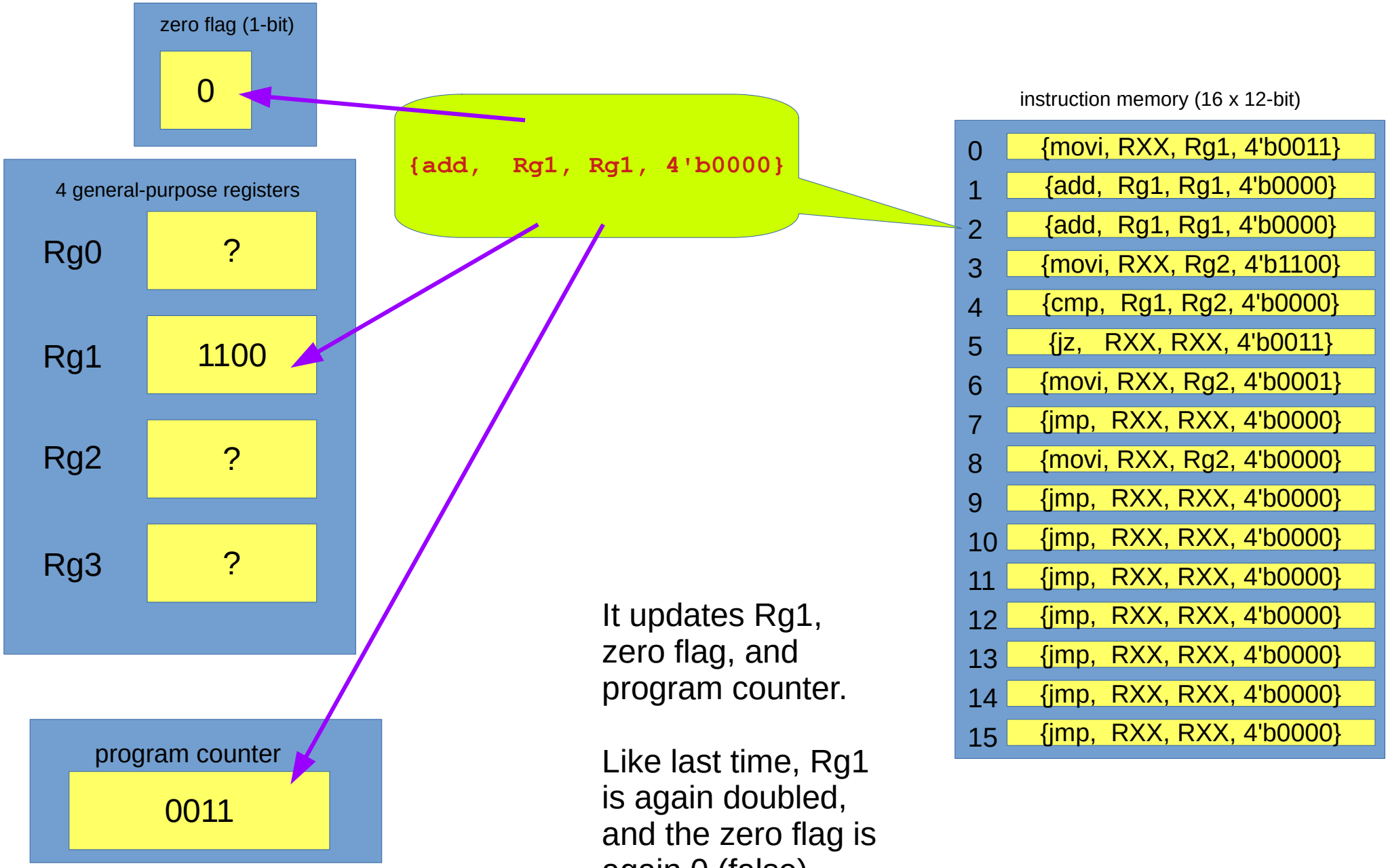
The add instruction adds Rg1 to itself, and stores the sum in Rg1. The result is that Rg1 is doubled from 3 to 6.

Because the sum is not zero, the zero flag is set to zero (false).

How does a program execute?



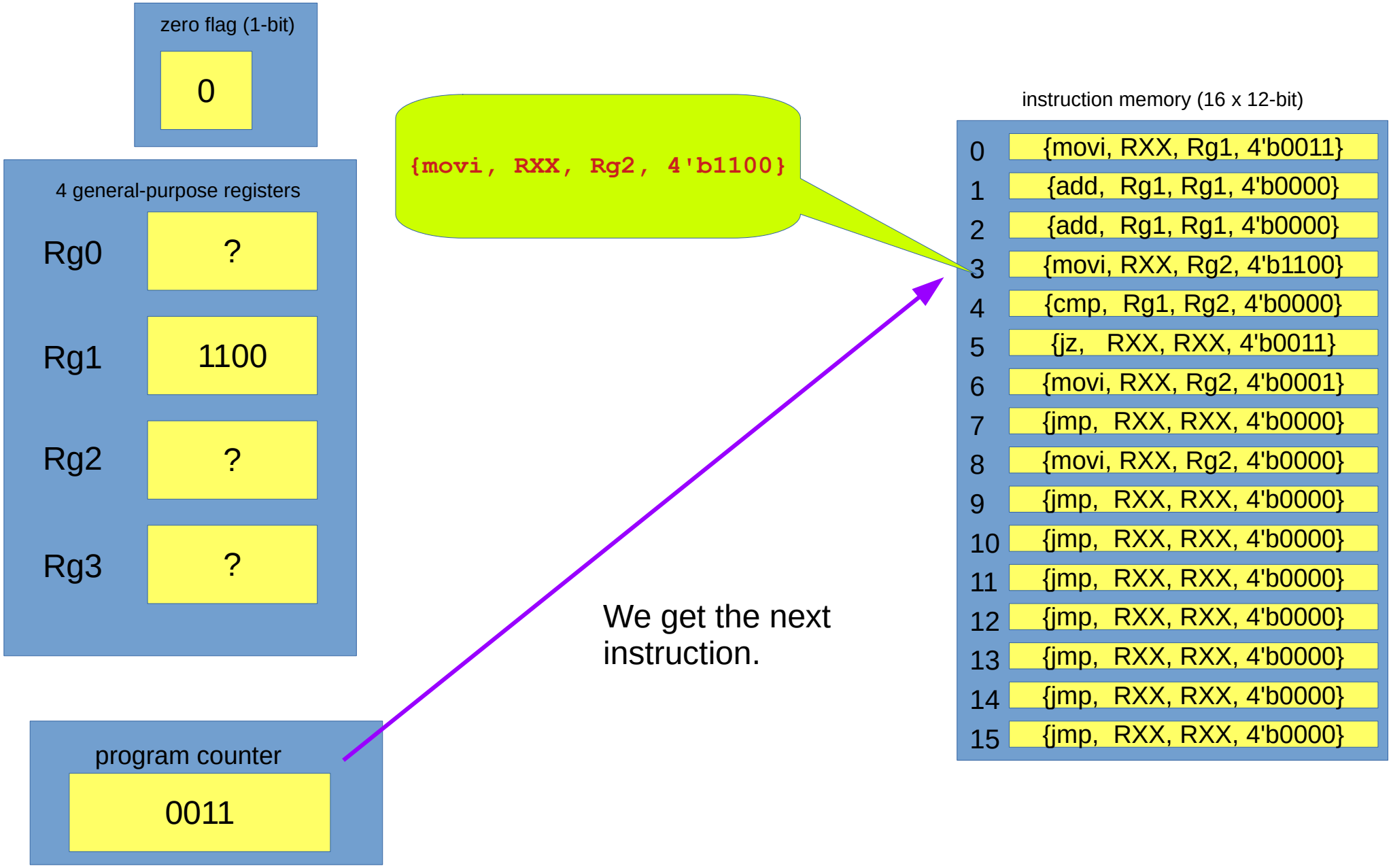
How does a program execute?



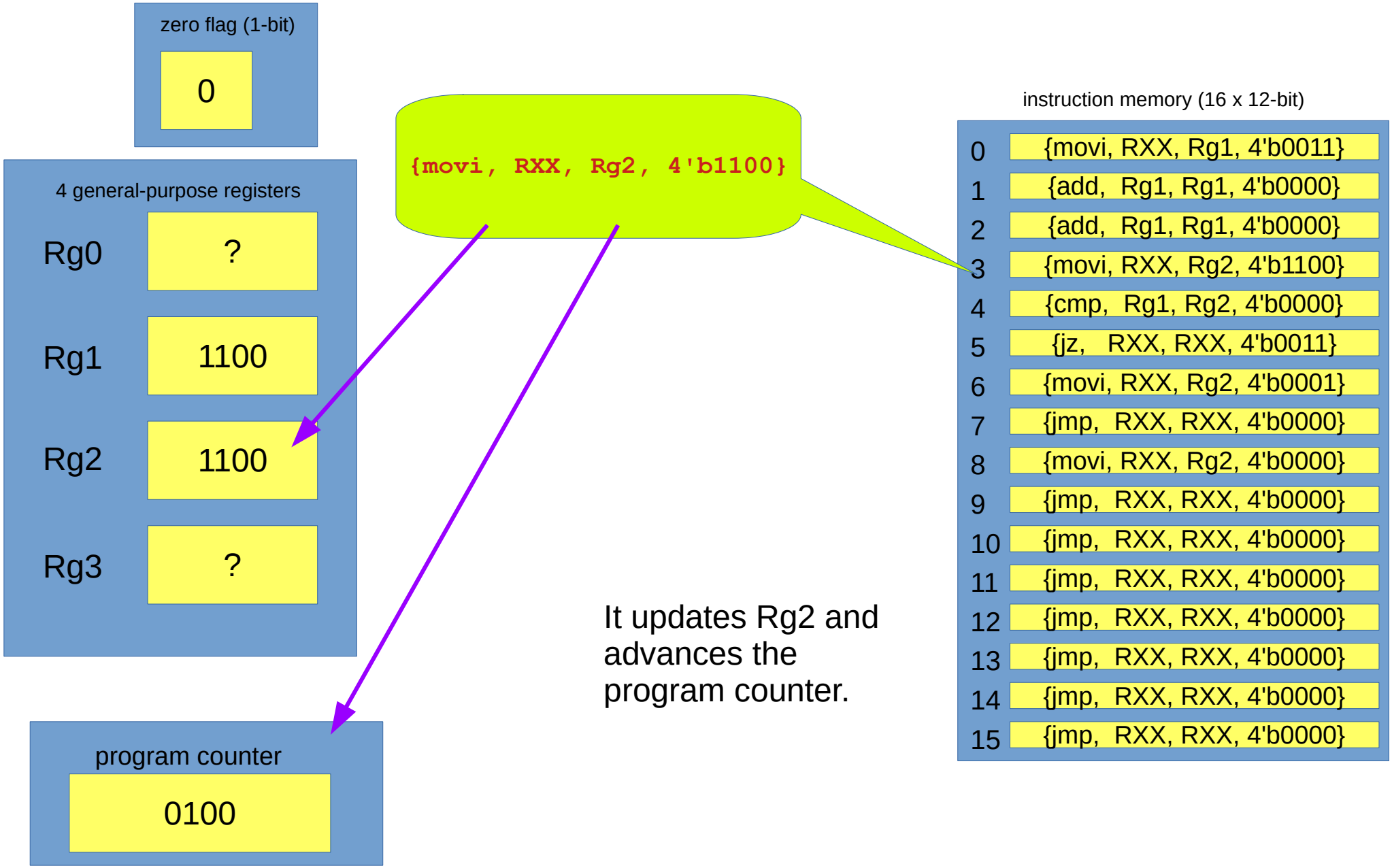
It updates Rg1, zero flag, and program counter.

Like last time, Rg1 is again doubled, and the zero flag is again 0 (false).

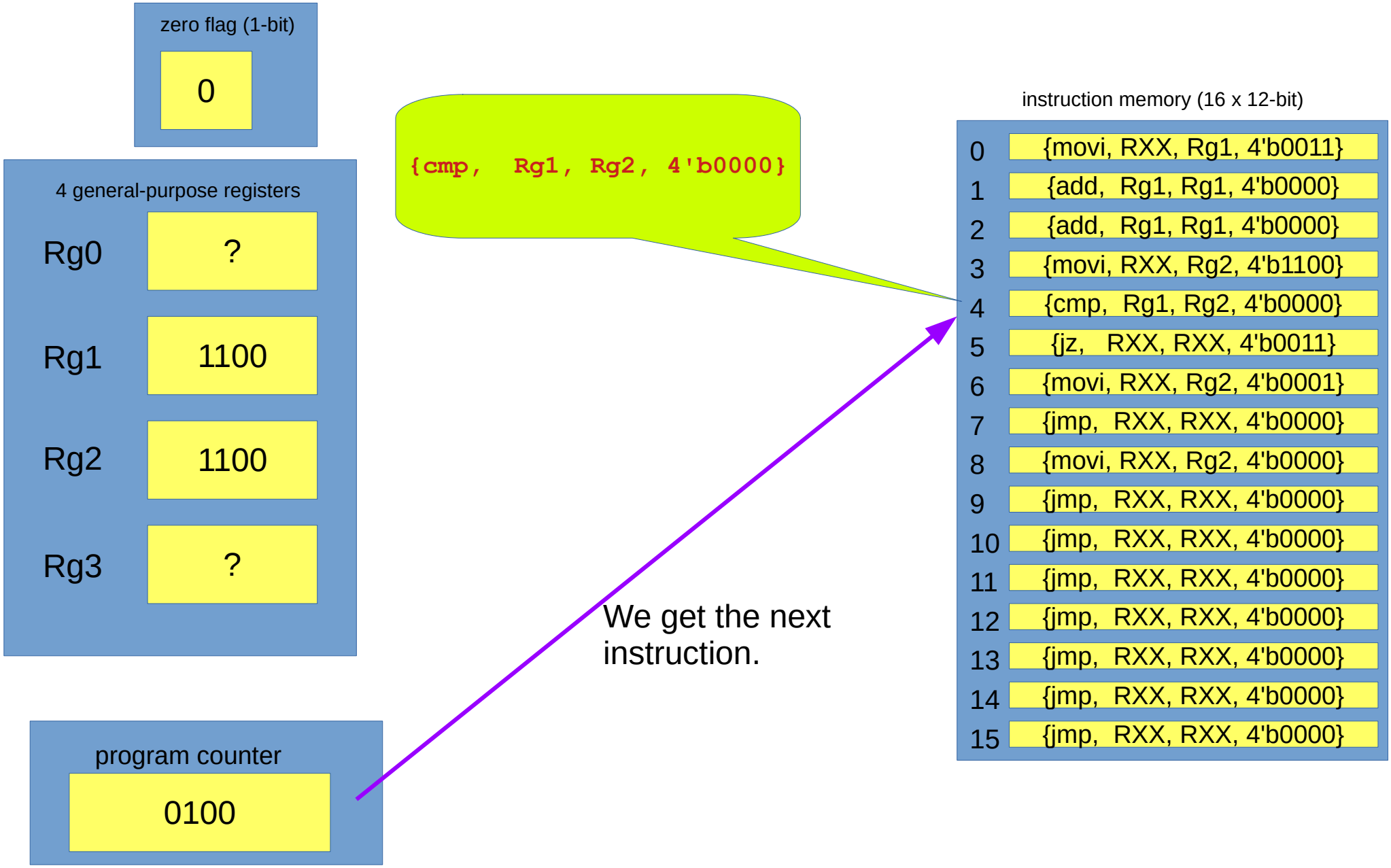
How does a program execute?



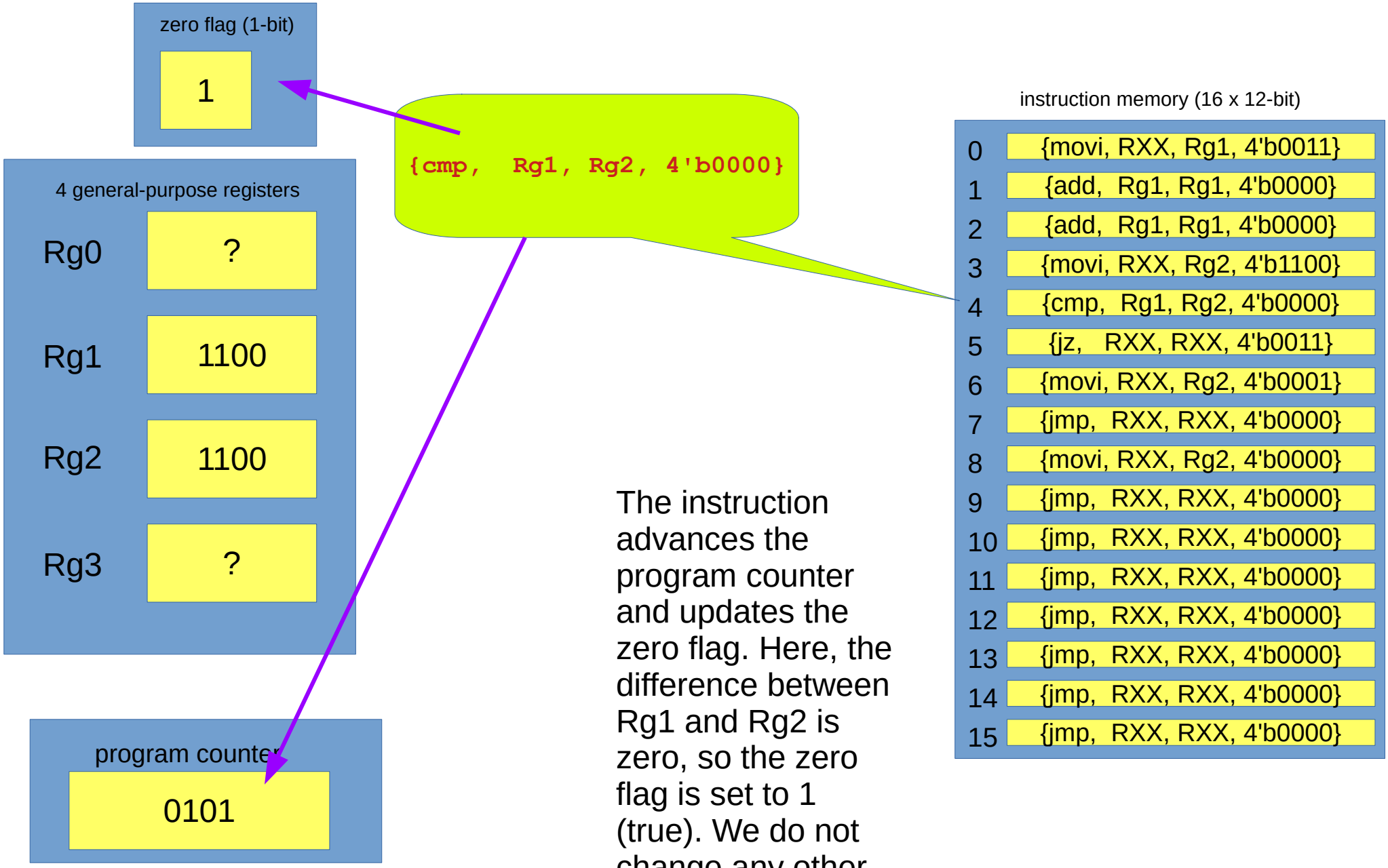
How does a program execute?



How does a program execute?

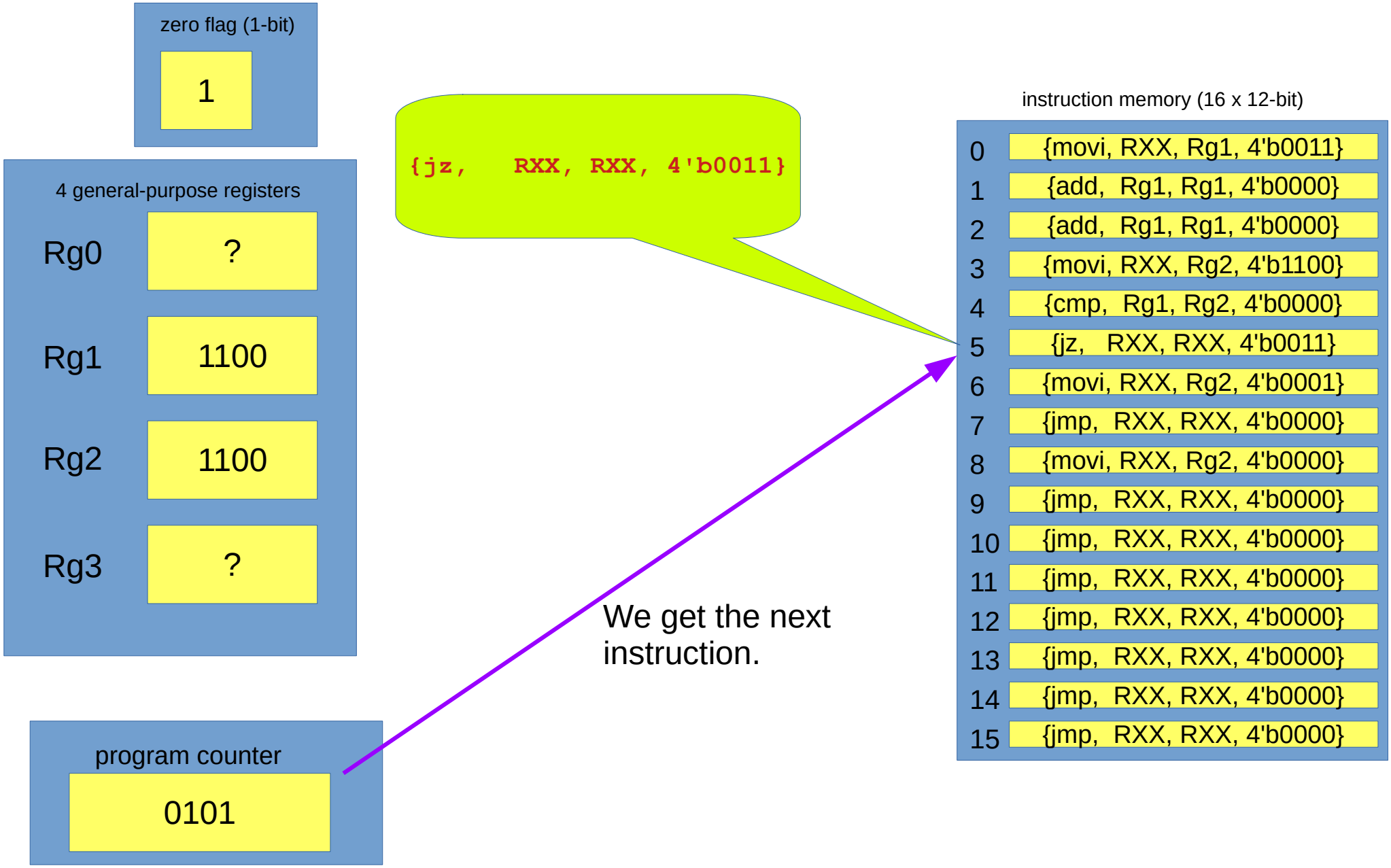


How does a program execute?

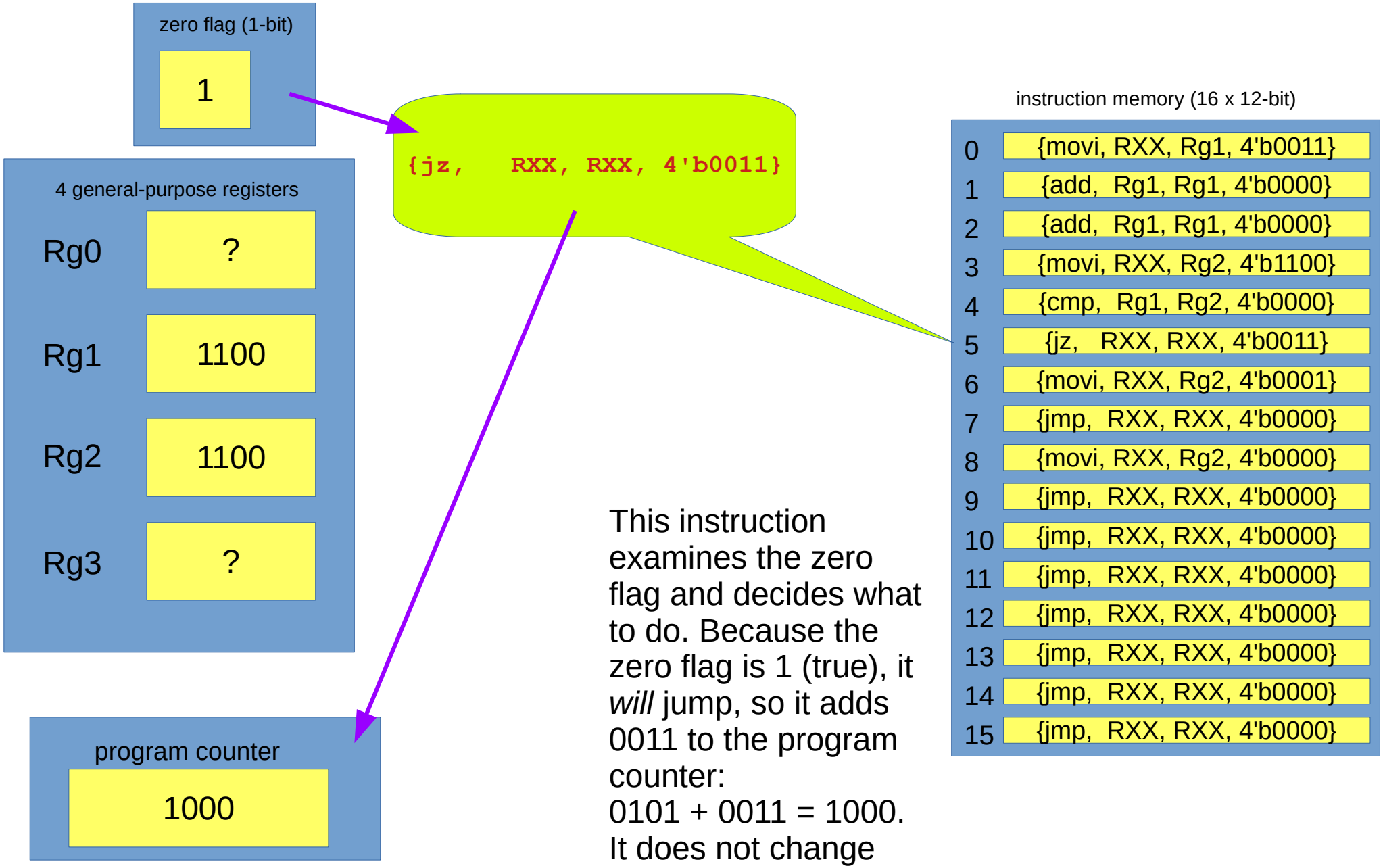


The instruction advances the program counter and updates the zero flag. Here, the difference between Rg1 and Rg2 is zero, so the zero flag is set to 1 (true). We do not change any other register.

How does a program execute?

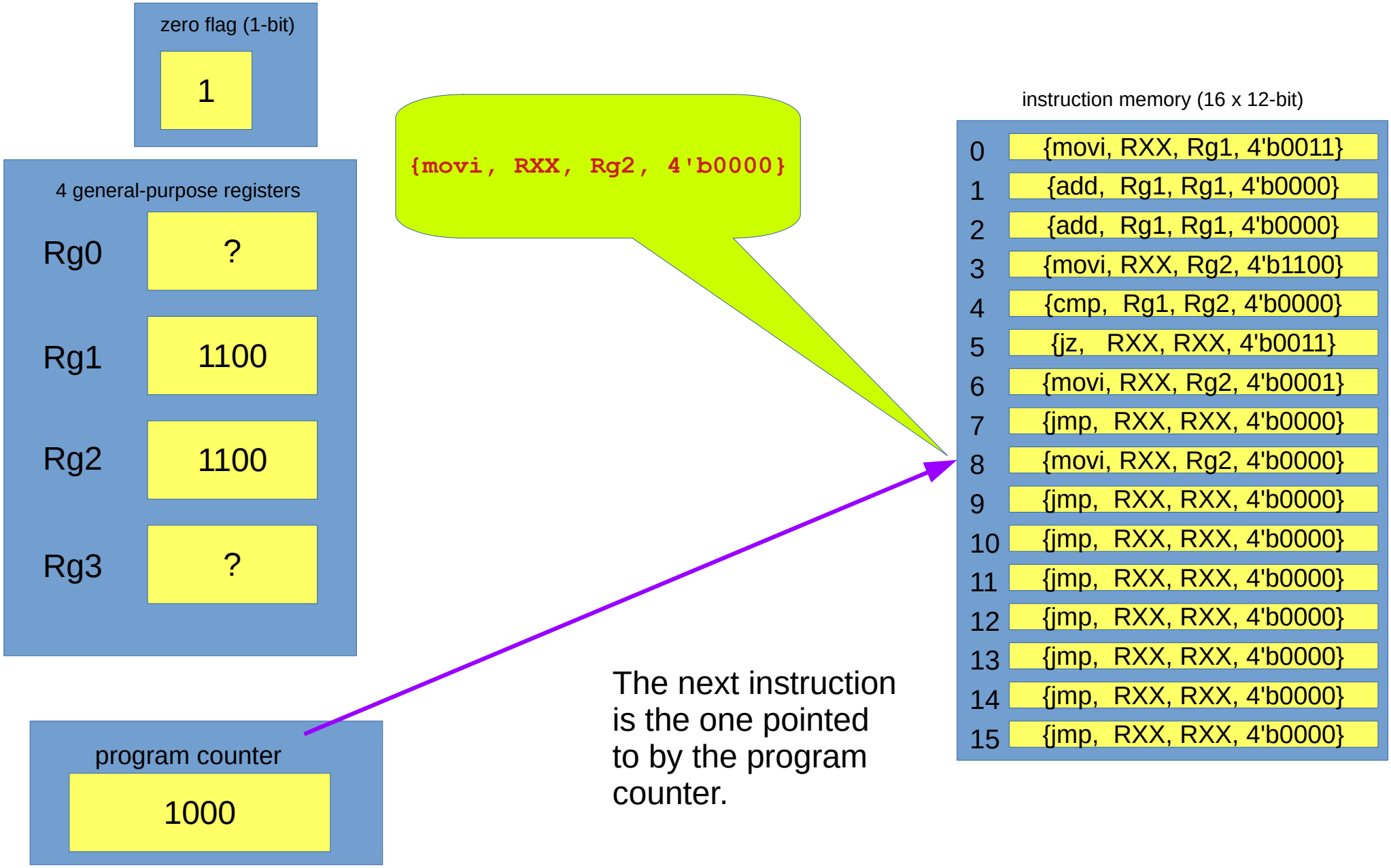


How does a program execute?

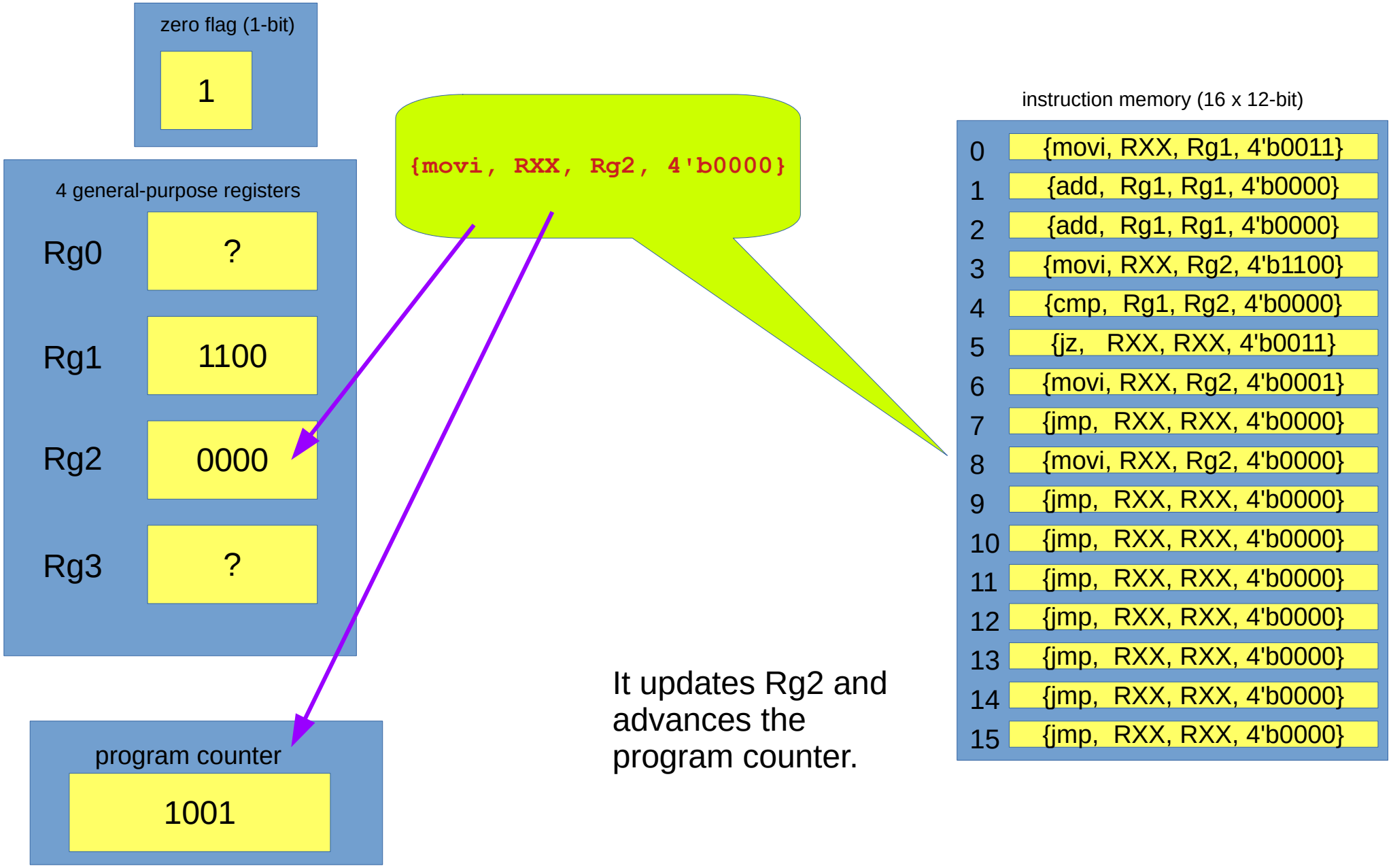


This instruction examines the zero flag and decides what to do. Because the zero flag is 1 (true), it *will* jump, so it adds 0011 to the program counter:
0101 + 0011 = 1000.
It does not change the zero flag or any other register.

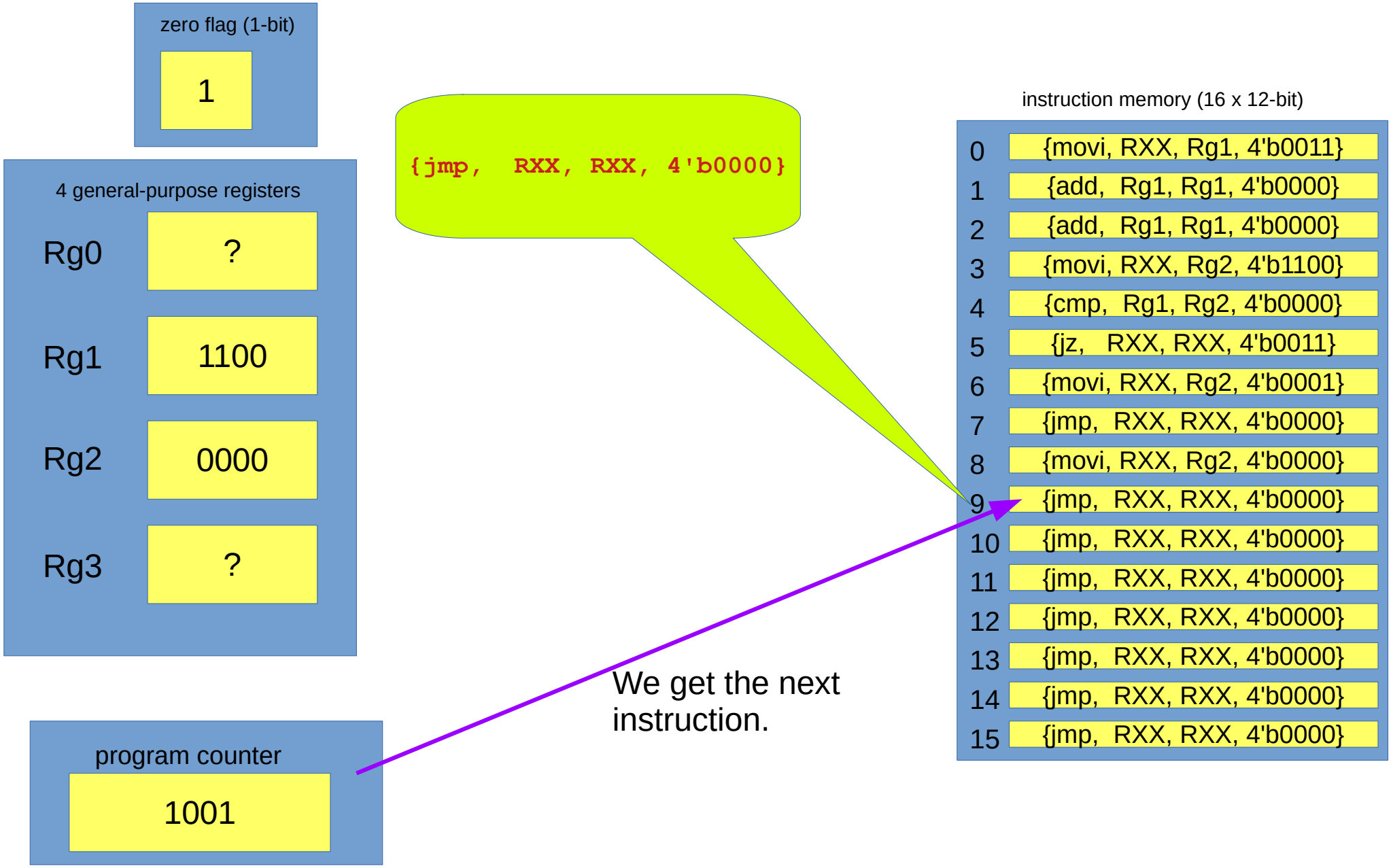
How does a program execute?



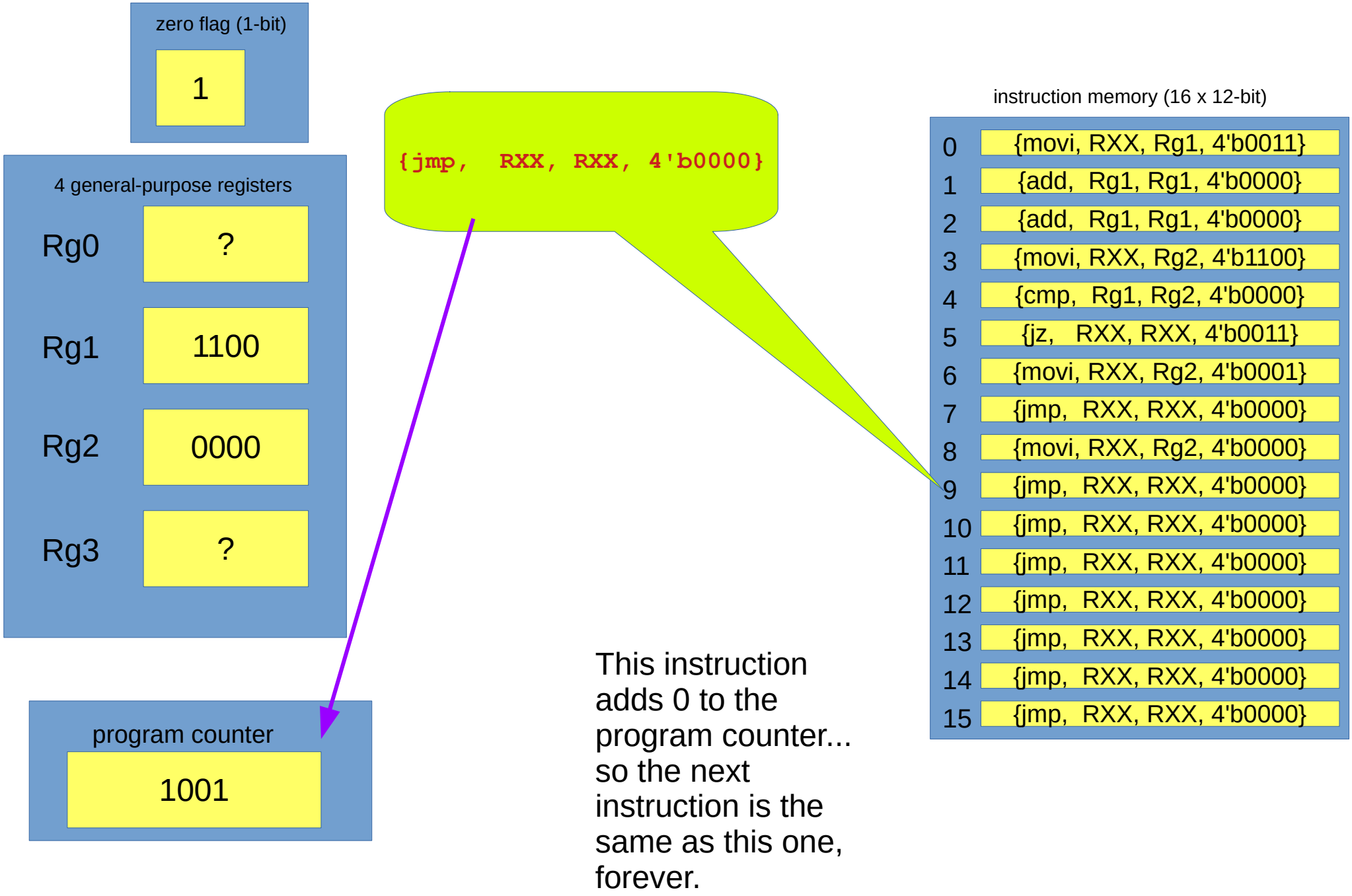
How does a program execute?



How does a program execute?



How does a program execute?



We can now understand this E15 program in terms of (roughly) equivalent pseudocode.

```
/*           OPCODE SRC  DST  IMMDATA */
myROM[0] = {movi, RXX, Rg1, 4'b0011};
myROM[1] = {add,  Rg1, Rg1, 4'b0000};
myROM[2] = {add,  Rg1, Rg1, 4'b0000};
myROM[3] = {movi, RXX, Rg2, 4'b1100};
myROM[4] = {cmp,  Rg1, Rg2, 4'b0000};
myROM[5] = {jz,   RXX, RXX, 4'b0011};
myROM[6] = {movi, RXX, Rg2, 4'b0001};
myROM[7] = {jmp,  RXX, RXX, 4'b0000};
myROM[8] = {movi, RXX, Rg2, 4'b0000};
myROM[9] = {jmp,  RXX, RXX, 4'b0000};
myROM[10] = {jmp,  RXX, RXX, 4'b0000};
myROM[11] = {jmp,  RXX, RXX, 4'b0000};
myROM[12] = {jmp,  RXX, RXX, 4'b0000};
myROM[13] = {jmp,  RXX, RXX, 4'b0000};
myROM[14] = {jmp,  RXX, RXX, 4'b0000};
myROM[15] = {jmp,  RXX, RXX, 4'b0000};
```

```
int rg1 = 3;
rg1 += rg1;
rg1 += rg1;
int rg2 = 12;
if (rg1==rg2)
  rg2 = 0;
else
  rg2 = 1;
while (true) {}
```

Overflow

Consider this E15 program:

```
myROM[0] = {movi, RXX, Rg0, 4'b0000};  // Rg0 = 0;
myROM[1] = {movi, RXX, Rg1, 4'b1111};  // Rg1 = 15;
myROM[2] = {subi, RXX, Rg0, 4'b0001};  // Rg0 = Rg0 - 1;
myROM[3] = {addi, RXX, Rg1, 4'b0001};  // Rg1 = Rg1 + 1;
myROM[4] = {jmp,  RXX, RXX, 4'b0000};  // end program
```

What will be the final value of Rg0?
What will be the final value of Rg1?

Keep in mind that these are each 4-bit registers.

Overflow

```
myROM[0] = {movi, RXX, Rg0, 4'b0000};  // Rg0 = 0;
myROM[1] = {movi, RXX, Rg1, 4'b1111};  // Rg1 = 15;
myROM[2] = {subi, RXX, Rg0, 4'b0001};  // Rg0 = Rg0 - 1;
myROM[3] = {addi, RXX, Rg1, 4'b0001};  // Rg1 = Rg1 + 1;
myROM[4] = {jmp,  RXX, RXX, 4'b0000};  // end program
```

E15 uses the usual algorithm for binary addition when adding and subtracting registers. In the case of subtraction, we must recall that subtraction is defined as addition of a negative.
In both cases the result must fit in four bits!

Calculating the final value of Rg0

```
   0000
 - 0001
---------
   ????
```

equivalent

```
   0000
 + 1111
---------
   ????
```

Calculating the final value of Rg1

```
   1111
 + 0001
---------
   ????
```

Overflow

```
myROM[0] = {movi, RXX, Rg0, 4'b0000};  // Rg0 = 0;
myROM[1] = {movi, RXX, Rg1, 4'b1111};  // Rg1 = 15;
myROM[2] = {subi, RXX, Rg0, 4'b0001};  // Rg0 = Rg0 - 1;
myROM[3] = {addi, RXX, Rg1, 4'b0001};  // Rg1 = Rg1 + 1;
myROM[4] = {jmp,  RXX, RXX, 4'b0000};  // end program
```

E15 uses the usual algorithm for binary addition when adding and subtracting registers. In the case of subtraction, we must recall that subtraction is defined as addition of a negative.
In both cases the result must fit in four bits!

Calculating the final value of Rg0

```
   0000
 - 0001
---------
   1111
```

equivalent

```
   0000
 + 1111
---------
   1111
```

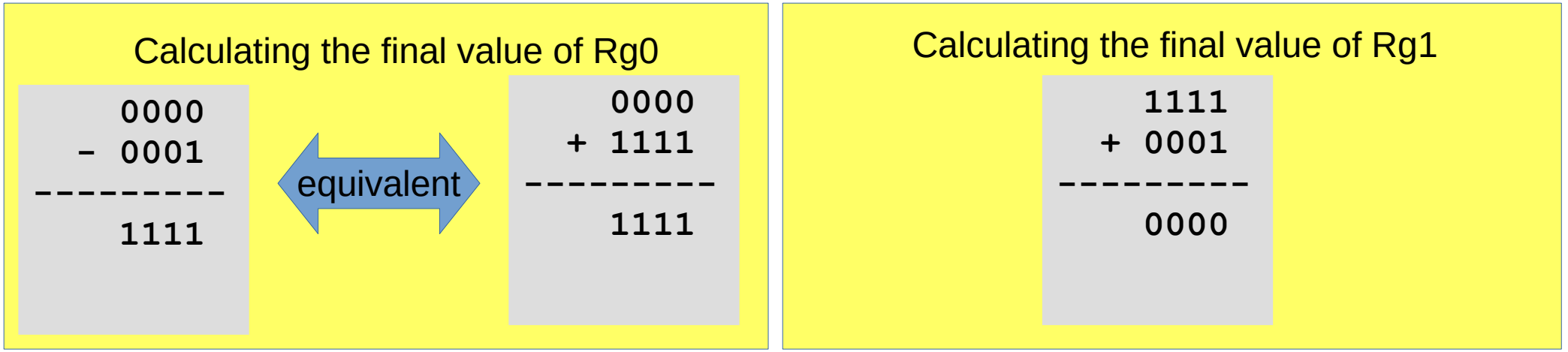
Subtracting 1 is the same as adding -1

Calculating the final value of Rg1

```
   1111
 + 0001
---------
   0000
```

This addition produces a carry-out bit, which we ignore.

Overflow



This is the phenomenon of *wraparound*.

E15 registers are 4-bit. This means they can store unsigned decimal values between 0 and 15, inclusive.

Because numbers are confined to a fixed bit width, adding or subtracting beyond the expressible range will cause the value to "wrap":

- too large values (such as 15+1=16) will become small (here, 0)
- too small values (such as 0-1=-1) will become large (here, 15)

The extreme ends of the number line connect.

Overflow

```
.....
myROM[12] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[13] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[14] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[15] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
```

Now consider this partial program. We show only the last four instructions, occupying addresses 12 through 15. What will happen *after* the instruction at address 15 executes?

- Program stops?
- Crash?
- Error message?
- Go back to beginning, at address 0?
- Go to address 16?

Remember, the E15 memory unit has only 16 cells, numbered 0 through 15.

Overflow

```
.....
myROM[12] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[13] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[14] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
myROM[15] = {addi, RXX, Rg0, 4'b0011};  // Rg0 += 3;
```

Now consider this partial program. We show only the last four instructions, occupying addresses 12 through 15. What will happen *after* the instruction at address 15 executes?

Calculating the next value of program counter

```
   1111
 + 0001
---------
   0000
```

=15

increment

=new pc

Therefore, the program continues executing at address 0.
That's why we need to explicitly mark the end of the program with a **`{jmp, RXX, RXX, 4'b0000}`** instruction!
Remember the the program counter is a 4-bit register, just like the others.

E15 in Verilog

Let's look at some interesting parts of the (incomplete) single-cycle E15 implementation in Verilog.

You can follow along in the source code you've been given.

```
module simpleALU(
    input addNotSub,
    input [3:0] src, dst,
    output zFlag,
    output [3:0] res);
    wire Cout;

    fourbit_adder the_adder(dst, addNotSub ? src : ~src,
          ~addNotSub, res, Cout);

    assign zFlag = !(res);

endmodule
```

Before we get into the E15 processor itself, let's define an ALU module.

This ALU supports only two operations: add and subtract. The addNotSub input selects the operation.

```
module simpleALU(
    input addNotSub,
    input [3:0] src, dst,
    output zFlag,
    output [3:0] res);
    wire Cout;

    fourbit_adder the_adder(dst, addNotSub ? src : ~src,
          ~addNotSub, res, Cout);

    assign zFlag = !(res);

endmodule
```

We use our **fourbit_adder** module to do the math for us. Note that for subtraction, we invert the second operand and add one via **Cin**, exactly as we previously discussed how subtractors work.
The result is passed out in **res**.

```
module simpleALU(
    input addNotSub,
    input [3:0] src, dst,
    output zFlag,
    output [3:0] res);
    wire Cout;

    fourbit_adder the_adder(dst, addNotSub ? src : ~src,
          ~addNotSub, res, Cout);

    assign zFlag = !(res);

endmodule
```

The ALU's **zFlag** output will be set to 1 when the result of the arithmetic operation is zero. We use the logical not operator **!** to do this.

```
simpleALU dataALU(addNotSub, operand1, operand2,
       aluOutputZero, aluOutput);
simpleALU pcALU(1'b1, pc, pcIncr, pcz, pcRes);
```

The E15 processor contains not one, but *two* ALUs: **dataALU** and **pcALU**.

```
// Register names
parameter
   Rg0 = 2'b00, Rg1 = 2'b01,
   Rg2 = 2'b10, Rg3 = 2'b11,
   RXX = 2'b00;

// Opcodes
parameter
  jmp  = 4'b0000, jz   = 4'b0010,
  movi = 4'b1001, mov  = 4'b1000,
  addi = 4'b1011, add  = 4'b1010,
  subi = 4'b1101, sub  = 4'b1100,
  cmpi = 4'b1111, cmp  = 4'bi
  jnz  = 4'b0011;
```

Now here's the E15 processor itself.

We're given values of constant symbols used in E15 assembly code. The Verilog **parameter** syntax is similar to C++ **const**, in that for convenience we are simply giving a name to a numeric value.

For example, the symbol **subi** maps to the 4-bit value 1101.

```
/*Processor state*/
reg [3:0] pc;                 // Program Counter
reg       zFlag;              // Zero flag
reg [3:0] r0, r1, r2, r3; // Registers


/*Program storage*/
reg [11:0] myROM [15:0];    // ROM (holds program)
```

Here we define the mutable storage capacity of the processor: the program counter, the zero flag, the four general-purpose registers, and the instruction memory (ROM).

Note that these components correspond to the elements described in the slide titled "What's inside the E15 processor?"

```
/*Processor state*/
reg [3:0] pc;                 // Program Counter
reg       zFlag;              // Zero flag
reg [3:0] r0, r1, r2, r3; // Registers


/*Program storage*/
reg [11:0] myROM [15:0]; // ROM (holds program)
```

The syntax defining **myROM** is worth discussing.

myROM is an array of 16 elements. Each element is a 12-bit value (i.e. one instruction).

Therefore **myROM[0]** is the first instruction of our program. **myROM[0][0]** is the MSB of the first instruction.

```
initial
  begin

     `include "program1.v"
        // load the program

     pc = 4'b0000;
        // initialize the program counter

  end
```

This code is run when the processor is "turned on." Here we set the initial value of registers.

The include line brings in an external file where our program is stored. That code in turn will assign values to each element of the **myROM** array. After this **myROM** will never change

Then, we initialize our program counter to zero, the address of the first instruction.

Other registers (**Rg0**, **Rg1**, **Rg2**, **Rg3**, **zFlag**) are not initialized and have no defined initial value.

```
   /*Parts of the instruction*/
   wire [3:0] opCode;          // op code
   wire [1:0] src, dst;        // src and dst register
   wire [3:0] immData;         // "Immediate" data


assign {opCode, src, dst, immData} = myROM[pc];
```

Here we define wires corresponding to each field of an instruction.

The opcode is 4 bits, the source and destination register names are 2 bits each, and the immediate data field is four bits.

The **assign** statement accesses the program memory, retrieving the instruction pointed to by the program counter. Then we map that value to each of the four fields.

Because this is *continuous assignment*, the field wires are updated whenever the program counter is changed.

```
always @(posedge clk)
   begin

      // Update zero flag
      case(opCode)
        addi, add, subi, sub, cmpi, cmp:
          begin
             zFlag <= aluOutputZero;
          end
      endcase

      // update destination register
      case(opCode)
        movi, mov, add, addi, sub, subi:
          case(dst)
            Rg0: r0 <= storeVal;
            Rg1: r1 <= storeVal;
            Rg2: r2 <= storeVal;
            Rg3: r3 <= storeVal;
          endcase
      endcase

      // Update program counter
      pc <= pcRes;

  end
```

This is the heart of the processor. This code defines a loop that executes at every clock cycle.

```
always @(posedge clk)
   begin

      // Update zero flag
      case(opCode)
        addi, add, subi, sub, cmpi, cmp:
          begin
             zFlag <= aluOutputZero;
          end
      endcase

      // update destination register
      case(opCode)
        movi, mov, add, addi, sub, s
          case(dst)
            Rg0: r0 <= storeVal;
            Rg1: r1 <= storeVal;
            Rg2: r2 <= storeVal;
            Rg3: r3 <= storeVal;
          endcase
      endcase

      // Update program counter
      pc <= pcRes;

   end
```

Here, we need to save the ALU's zero flag output for those instructions that set it.

The **case** syntax lets us provide conditional actions for certain opcodes, similar to C++'s **switch**.

```
always @(posedge clk)
   begin

      // Update zero flag
      case(opCode)
        addi, add, subi, sub, cmpi, cmp:
          begin
             zFlag <= aluOutputZero;
          end
      endcase

      // update destination register
      case(opCode)
        movi, mov, add, addi, sub, subi:
          case(dst)
            Rg0: r0 <= storeVal;
            Rg1: r1 <= storeVal;
            Rg2: r2 <= storeVal;
            Rg3: r3 <= storeVal;
          endcase
      endcase

      // Update program counter
      pc <= pcRes;

  end
```

Then, we need to update the destination register for those instructions that set it. Depending on the value of the **dst** field, we will assign **storeVal** to one of the four general-purpose registers.

Note that other instructions, such as **cmp** and **jmp**, do not modify these registers.

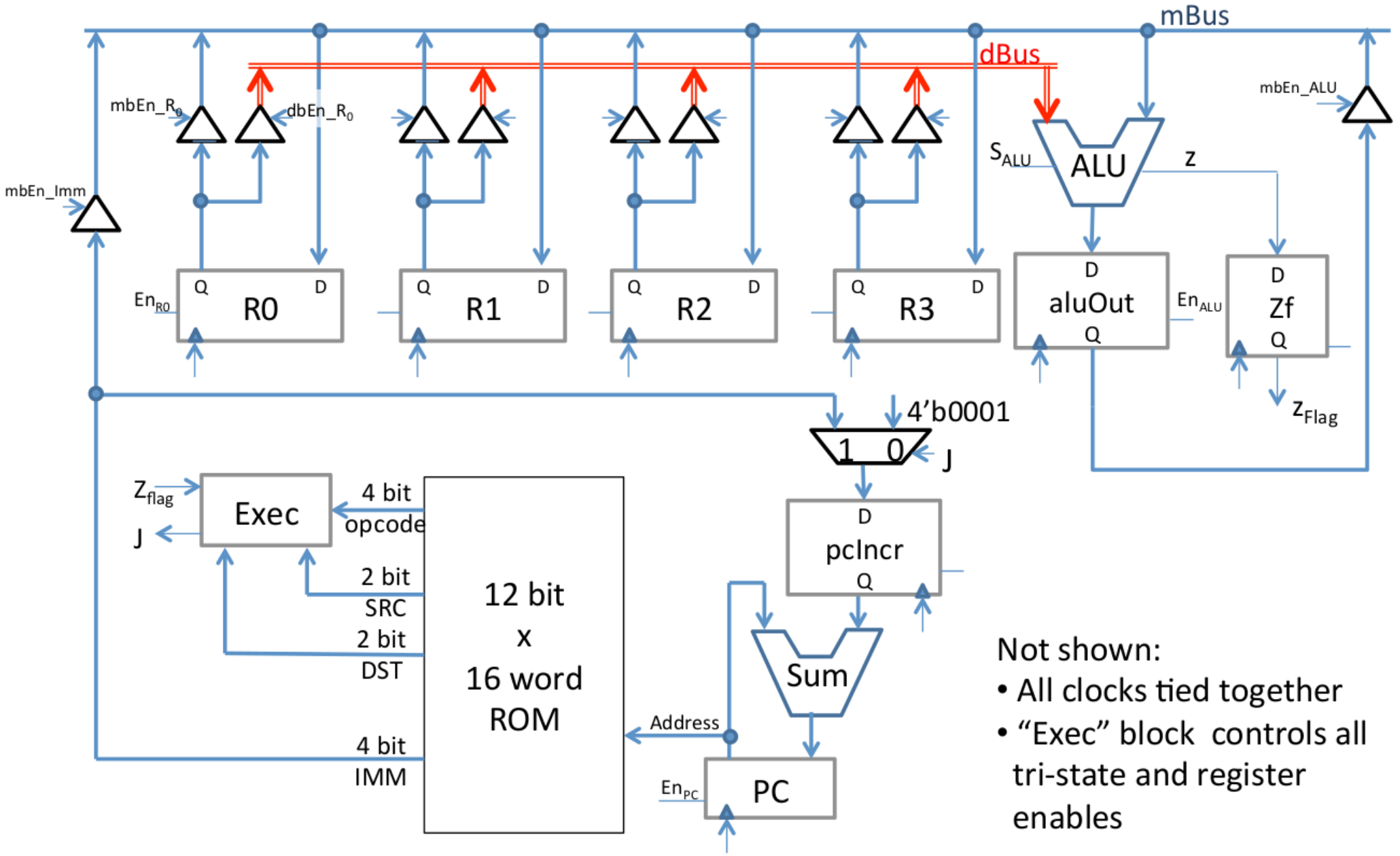
```
always @(posedge clk)
   begin

      // Update zero flag
      case(opCode)
        addi, add, subi, sub, cmpi, cmp:
          begin
             zFlag <= aluOutputZero;
          end
      endcase

      // update destination register
      case(opCode)
        movi, mov, add, addi, sub, subi:
          case(dst)
            Rg0: r0 <= storeVal;
            Rg1: r1 <= storeVal;
            Rg2: r2 <= storeVal;
            Rg3: r3 <= storeVal;
          endcase
      endcase

      // Update program counter
      pc <= pcRes;

  end
```

Finally, we update the program counter with the new value calculated by pcALU.

After that, the clock cycle is finished and we move on to the next instruction.

E15 multicycle

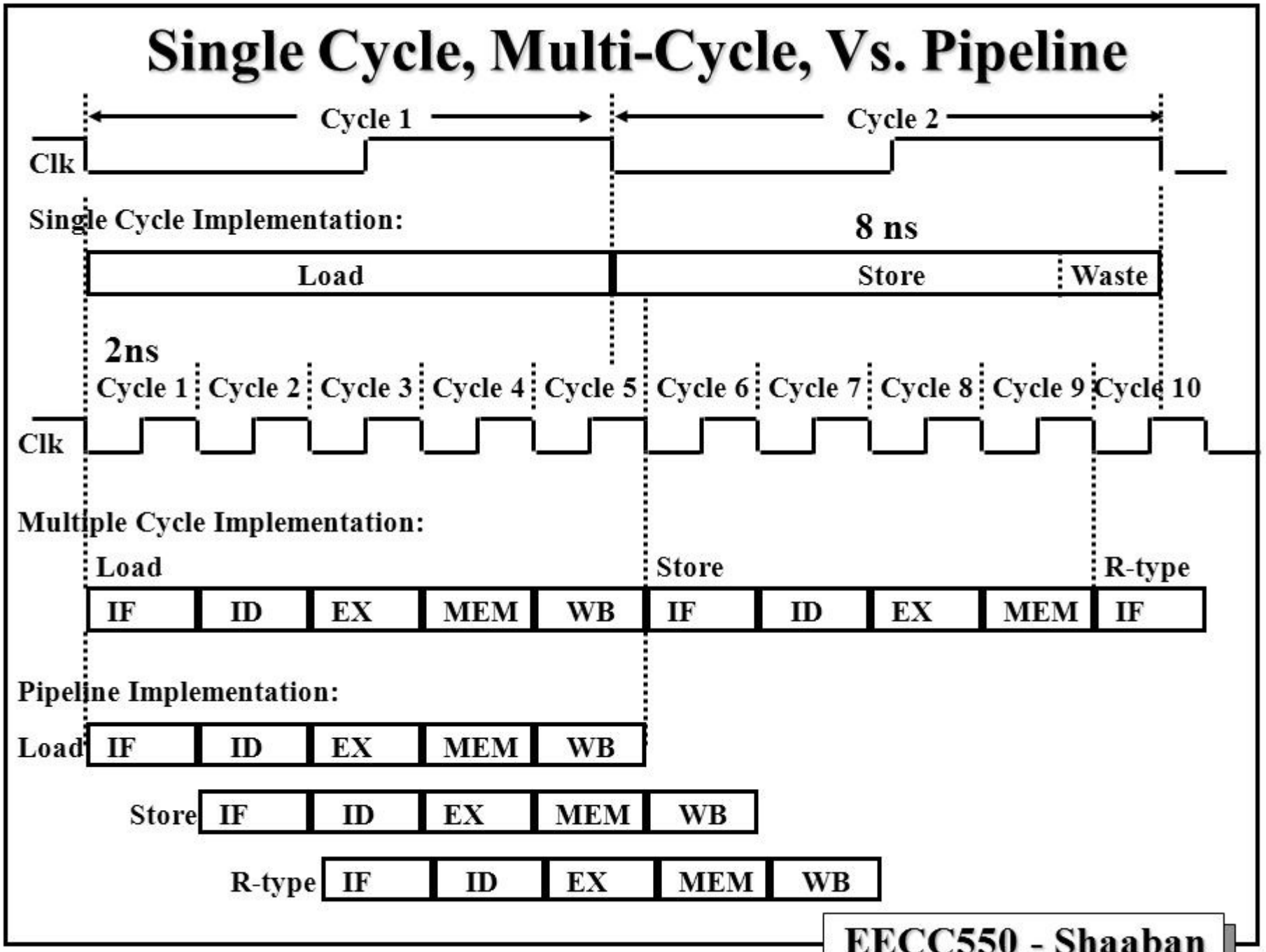
E15 Processor



Not shown:

- All clocks tied together
- "Exec" block controls all tri-state and register enables

Single Cycle, Multi-Cycle, Vs. Pipeline



What happens in each cycle? E15

fetch
    opcode, src, dst, immData <= myROM[pc]

decode
    mBus <= sourceValue1
    dBus <= sourceValue2
    addNotSub <= addOrSubtract?

exec
    pcIncr <= nextInstruction
    mBus <= aluResult

store
    destinationValue <= mBus
    pc <= pc + pcIncr