

## Design of a simple processor

### 1 Architecture

The processor we will design will have the following components:

- 8 32-bit registers  $R_0, R_1, \dots, R_7$  for holding data.
- 256 16-bit registers  $P_0, P_1, \dots, P_{255}$  for holding a program.
- A 32-bit adder.
- An 8-bit register  $PC$  that will serve as a program counter.

It is useful to have the constant values 0 and 1 available for use. So we set registers  $R_0$  and  $R_1$  to hold the values 0 and 1, respectively, permanently.

The processor will have five types of instructions: *add*, *negate*, *load*, and *jump if zero*. A program consists of a sequence of instructions stored in the 256 program registers. Each of these registers holds 16 bits. The 16 bits of an instruction specify the type of instruction and its operands. We need two bits to specify the instruction; the first two of the 16 bits (positions 0 and 1) will specify the instruction type. The formats of the 4 instructions are as follows.

The add instruction adds the contents of two registers  $R_a$  and  $R_b$  and stores the result in register  $R_c$ . The indices  $a$ ,  $b$ , and  $c$  are specified in bit positions 2-4, 5-7, and 8-10, respectively. Bit positions 11-15 will be ignored. The add instruction also increments the program counter by 1. The add instruction

$$\text{add } R_a, R_b \rightarrow R_c$$

is encoded as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	a	a	a	b	b	b	c	c	c	0	0	0	0	0

The negate instruction replaces  $R_a$  with  $-R_a$ , using the two's complement representation. The index  $a$  is specified by bit positions 2-4. The negate instruction also increments the program counter by 1. The negate instruction

$$\text{neg } R_a$$

is encoded as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	a	a	a	0	0	0	0	0	0	0	0	0	0	0

The load instruction loads an 8-bit number  $d$  into the 8 low-order bit positions of register  $R_a$ . The index  $a$  is specified by bit positions 2-4. The value  $d$  is specified in binary by the 8 low-order positions of the instruction; that is, positions 8-15. The effect of the instruction is to set the value in register  $R_a$  to  $d$ ; note that the 24 high-order bits of  $R_a$  are set to 0. Also, the program counter is incremented by 1. The load instruction

$$\text{lod } d \rightarrow R_a$$

is encoded as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	a	a	a	0	0	0	d	d	d	d	d	d	d	d

The jump if zero instruction changes the program counter register to the value specified by an 8-bit number  $d$ , if register  $R_a$  is 0; otherwise the program counter  $PC$  is incremented by 1, as usual. The jump if zero instruction

$$\text{jiz } R_a \rightarrow d$$

is encoded as follows.

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	a	a	a	0	0	0	d	d	d	d	d	d	d	d

## 2 Multiplexers and demultiplexers

Digital circuits are designed in a modular fashion, building more and more complex components out of basic building blocks. A key component of almost every digital circuit is a device that selects any one of many inputs and sends that value to a single output, called a *multiplexer*.

Consider a 2-way 1-bit multiplexer that has three inputs  $X_0$ ,  $X_1$ , and  $Y$ , and a single output  $Z$ . If  $Y$  (called the *control*) is 0, then the output  $Z$  is the same as  $X_0$ ; if  $Y$  is 1, then the output  $Z$  is the same as  $X_1$ . For example, if  $X_0$  is 0,  $X_1$  is 1, and  $Y$  is 1, then  $Z$  equals 1. The truth table for the above multiplexer is as follows.

$X_0$	$X_1$	$Y$	$Z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

We can now express the output  $Z$  as a Boolean formula in terms of the inputs  $X_0$ ,  $X_1$ , and  $Y$ . Simplifying the Boolean formula as much as you can, by applying the laws of logical equivalence, we obtain the following.

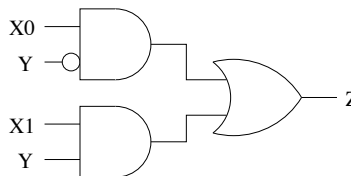
Begin by reading off the disjunctive normal form (DNF) of  $Z$  from the truth table:

$$Z = (\neg X_0 \wedge X_1 \wedge Y) \vee (X_0 \wedge \neg X_1 \wedge \neg Y) \vee (X_0 \wedge X_1 \wedge \neg Y) \vee (X_0 \wedge X_1 \wedge Y)$$

Then simplify using commutativity, distributivity, and the fact that  $A \vee \neg A = T$ :

$$\begin{aligned} Z &= (\neg X_0 \wedge X_1 \wedge Y) \vee (X_0 \wedge \neg X_1 \wedge \neg Y) \vee (X_0 \wedge X_1 \wedge \neg Y) \vee (X_0 \wedge X_1 \wedge Y) \\ &= (\neg X_0 \wedge X_1 \wedge Y) \vee (X_0 \wedge X_1 \wedge Y) \vee (X_0 \wedge X_1 \wedge \neg Y) \vee (X_0 \wedge \neg X_1 \wedge \neg Y) \\ &= ((\neg X_0 \vee X_0) \wedge X_1 \wedge Y) \vee (X_0 \wedge (X_1 \vee \neg X_1) \wedge \neg Y) \\ &= (X_1 \wedge Y) \vee (X_0 \wedge \neg Y) \end{aligned}$$

We are now ready to design a circuit for the multiplexer.

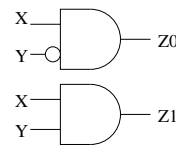


As one can imagine, an equally important device is the *demultiplexer*, that essentially does the opposite of what the multiplexer does: it sends a single input to one of many outputs, depending on a control value. Consider a demultiplexer that has two inputs  $X$  and  $Y$  and two outputs  $Z_0$  and  $Z_1$ . If  $Y$  is 0, then  $Z_0$  takes the value of  $X$ , and  $Z_1$  is 0. On the other hand, if  $Y$  is 1, then  $Z_1$  takes the value of  $X$ , and  $Z_0$  is 0.

We now design a circuit for the demultiplexer, proceeding as above. The DNF is particularly simple for this truth table, so we don't need to simplify the Boolean formulas.

$X$	$Y$	$Z_0$	$Z_1$
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

$$\begin{aligned} Z_0 &= X \wedge \neg Y \\ Z_1 &= X \wedge Y \end{aligned}$$



### 3 Clocked gates and registers

Any circuit built using the gates we studied in class (AND, OR, NOT, etc.) has the property that once the inputs are set, the values on all the wires, and, in particular, the outputs, are instantly computed. In many circuits, e.g., the CPU we are planning to design, we need to control the flow of data through the circuit. This can be achieved by enhancing gates and their associated circuits with clocks.

A *clocked gate* is a regular gate enhanced with a clock. A clock can be viewed as issuing a sequence of pulses at a regular interval. At every clock pulse, the clocked gate applies the gate function to yield the output of the clocked gate. Between two consecutive clock pulses, the output of the clocked gate remains unchanged.

For instance, using a combination of clocked and unclocked NOT gates, we can design registers. Here is a 4-bit register that accepts a 4 input bits before a clock pulse, and puts that value on its 4 output bits. It is easy to generalize this to 8-bit, 32-bit, or larger registers.

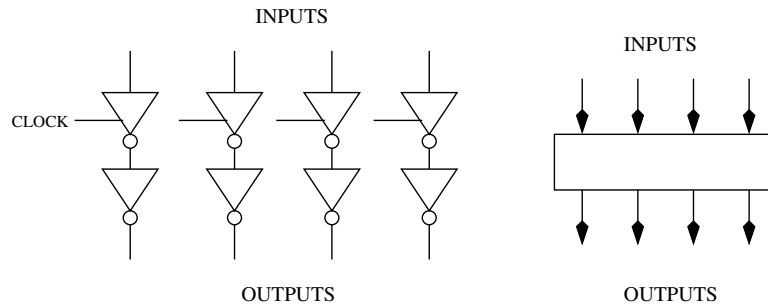


Figure 1: Implementation of a 4-bit register

The above register only holds values for one clock cycle. It is much more helpful if we could design the register so as to hold the values for an extended period of time. We can extend the register circuit from above so that it accepts an extra input  $S$  (for Set). If  $S$  is 1, then the new circuit behaves exactly as above when the clock pulses. However, if  $S$  is 0 when the clock pulses, then the register ignores its inputs, keeping the same outputs as before the clock pulse. It is this notion of a register (with set bits) that will use in the subsequent processor design.

**Exercise:** Design a circuit for the above extension of the 4-bit register. (*Hint:* It may help to use a 2-way 1-bit multiplexer.)

## 4 Design of the processor

We will design the entire processor using the basic gates that we have studied, along with clocks. We have already seen how to build adders from gates. We have also seen how to build multiplexers, demultiplexers, and registers. We will present the design in a modular fashion – building more complex components from simpler components that we have already designed. We assume that the registers  $P_0, \dots, P_{255}$  already contain the instructions to run, and their contents do not need to be changed.

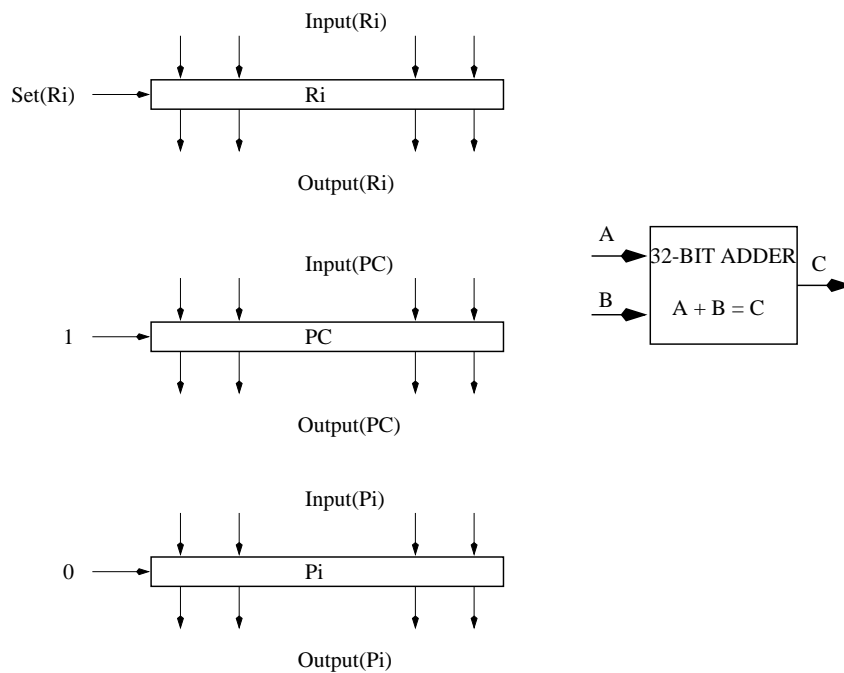
Our design will proceed in steps.

1. We lay out the registers (giving names to the input bits, output bits, and registers for convenience).
2. We design the circuits for extracting the next instruction and incrementing the PC.
3. We design the circuits for implementing the add, negate, load, and jump-if-zero operations.

4. We design the circuit that determines the input and the set bits for the data registers and the PC based on the above circuits.
5. In each of the above circuits, we clearly label the inputs and outputs. These circuits can be put together by matching the appropriate labeled inputs and outputs.

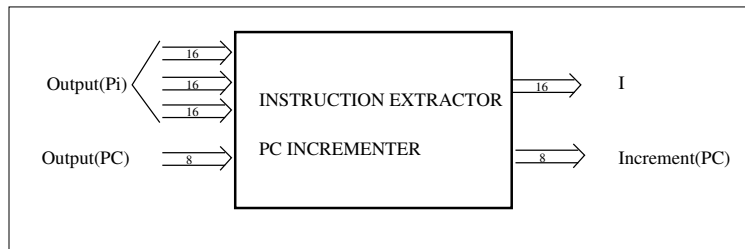
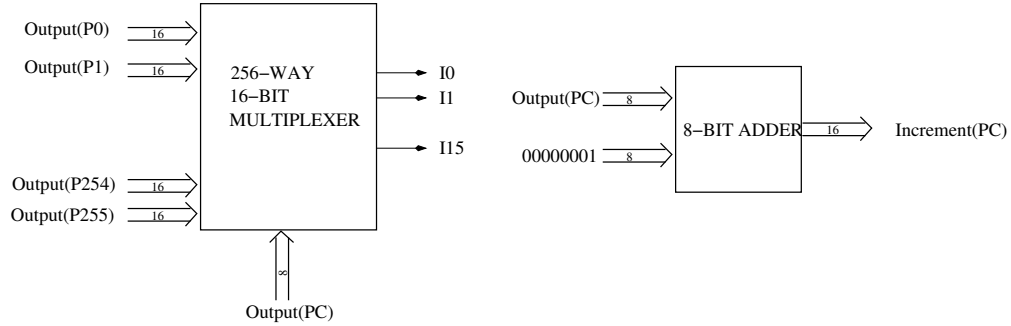
## 4.1 Registers

Here is a layout of the registers. Note that the set bit for PC is always 1 and the set bit of program registers is always 0. So the PC changes at every clock cycle and the program registers never change. The set bits for the data registers will be determined by the instruction.



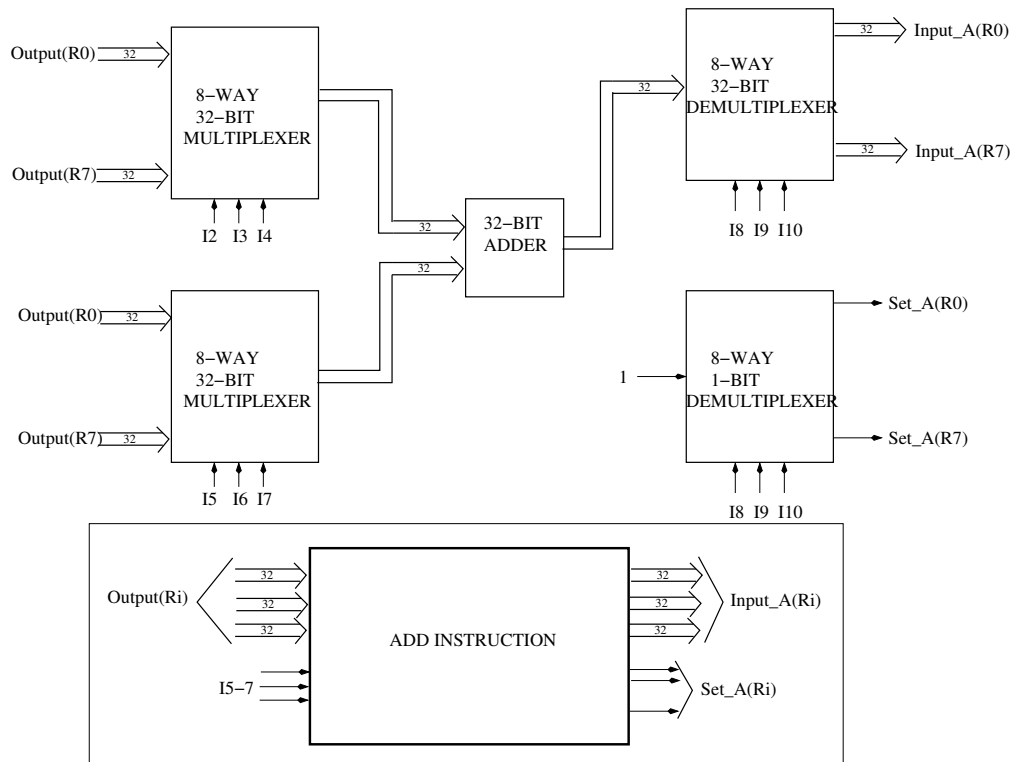
## 4.2 Extracting the instruction and incrementing the PC

The following are the circuits for extracting the next instruction using the PC, and calculating the increment of PC. Note that whether the PC will be actually incremented (or set to something else) will be determined by the instruction, as we will see shortly.



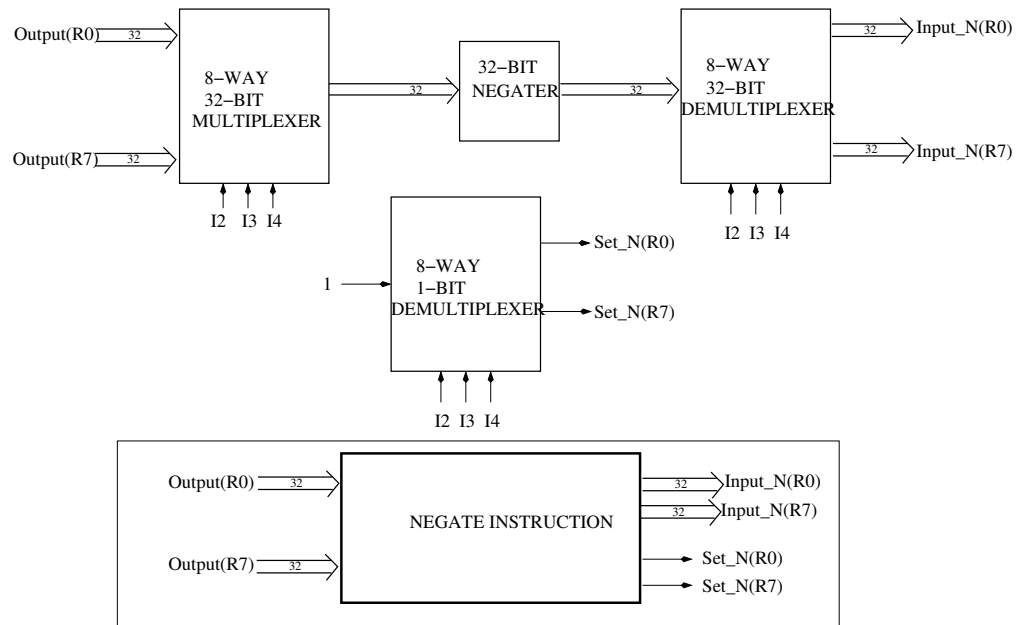
### 4.3 The add instruction

Here is the circuit for implementing the add instruction.



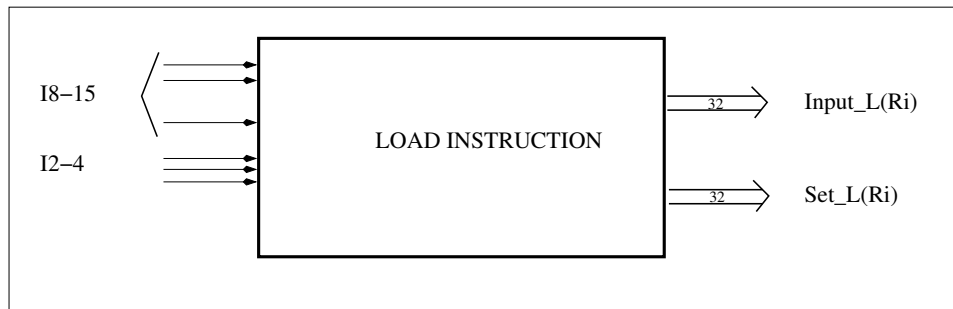
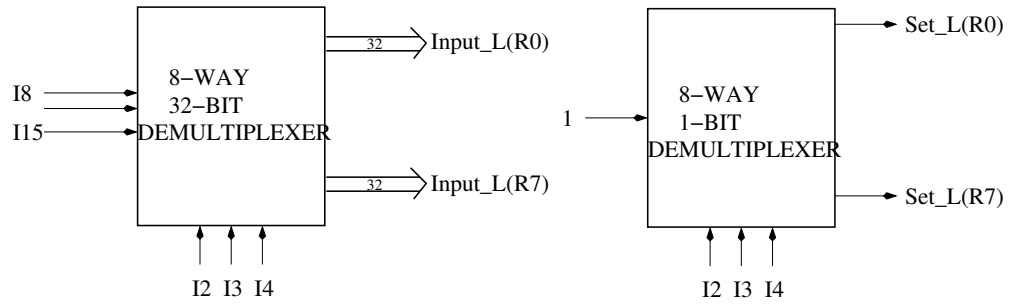
## 4.4 The negate instruction

Here is the circuit for implementing the negate instruction.



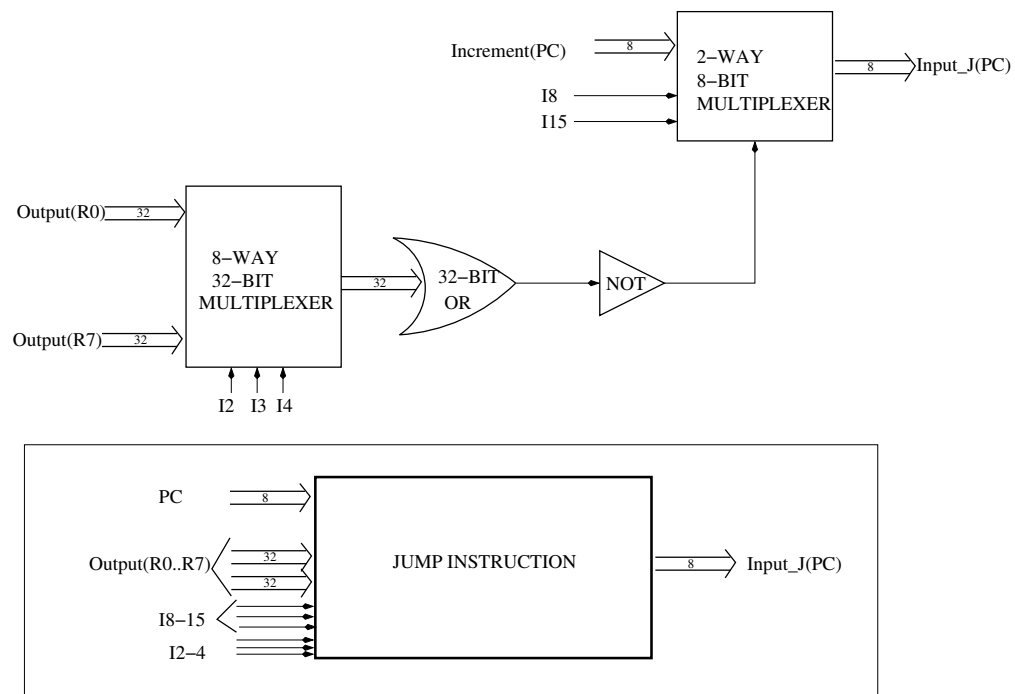
## 4.5 The load instruction

Here is the circuit for implementing the load instruction.



## 4.6 The jump-if-zero instruction

Here is the circuit for implementing the jump-if-zero instruction.



## 4.7 Storing the new values into the data registers and PC

The values computed by the add, negate, load, and jump-if-zero circuits give us potential values that need to be stored in the data registers. Whether the registers need to be updated and what their new values will be is determined by the particular instruction type. The following circuit pushes in the correct value into the registers. It also determines the correct value of the PC. The inputs to the program registers are the same as their outputs (since they are not allowed to change).

