ALU

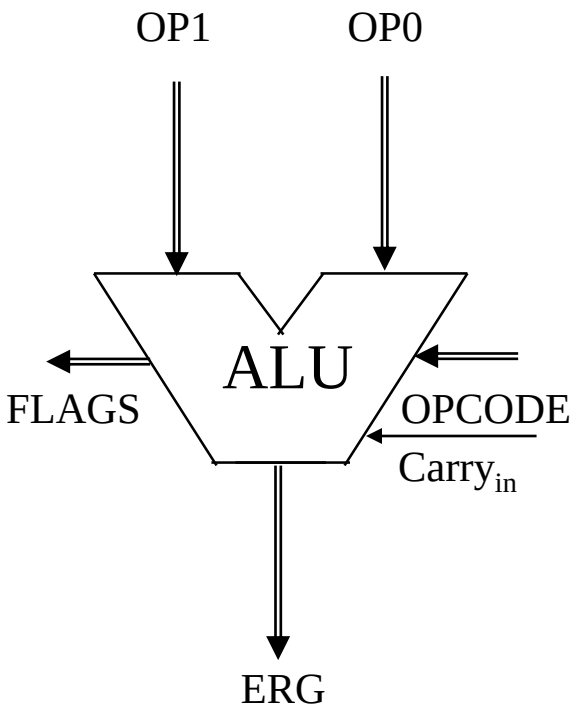
ALU-Assembly

An ALU (Arithmetic Logic Unit) by rule of thumb, consists of

- Adder
- Logical Unit
- Shifter

Inputs of an ALU: Two operands, Instruction code

Outputs of an ALU: Results, Flags



Adder

The core of every ALU is an adder. We can see a ripple carry adder in the next slide.

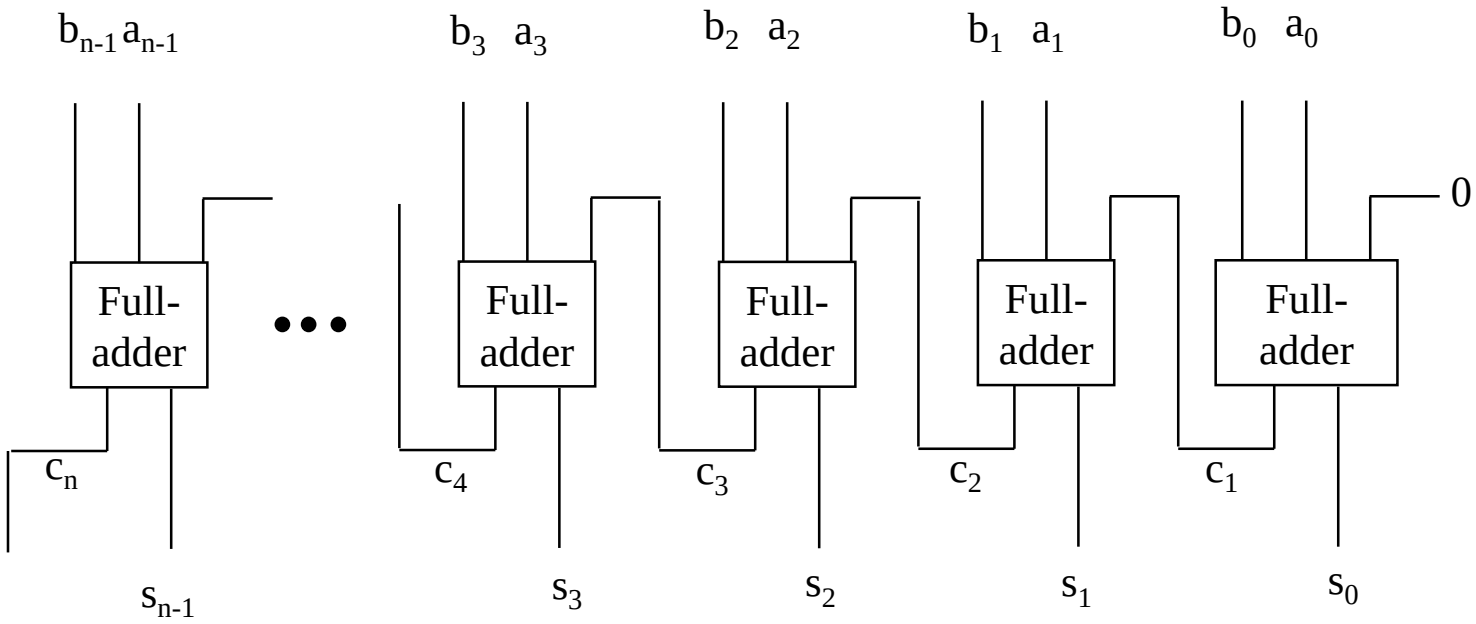
How can we use this to perform a subtraction?

We create the 2's complement on one of the operands and connect it to one input of the adder.

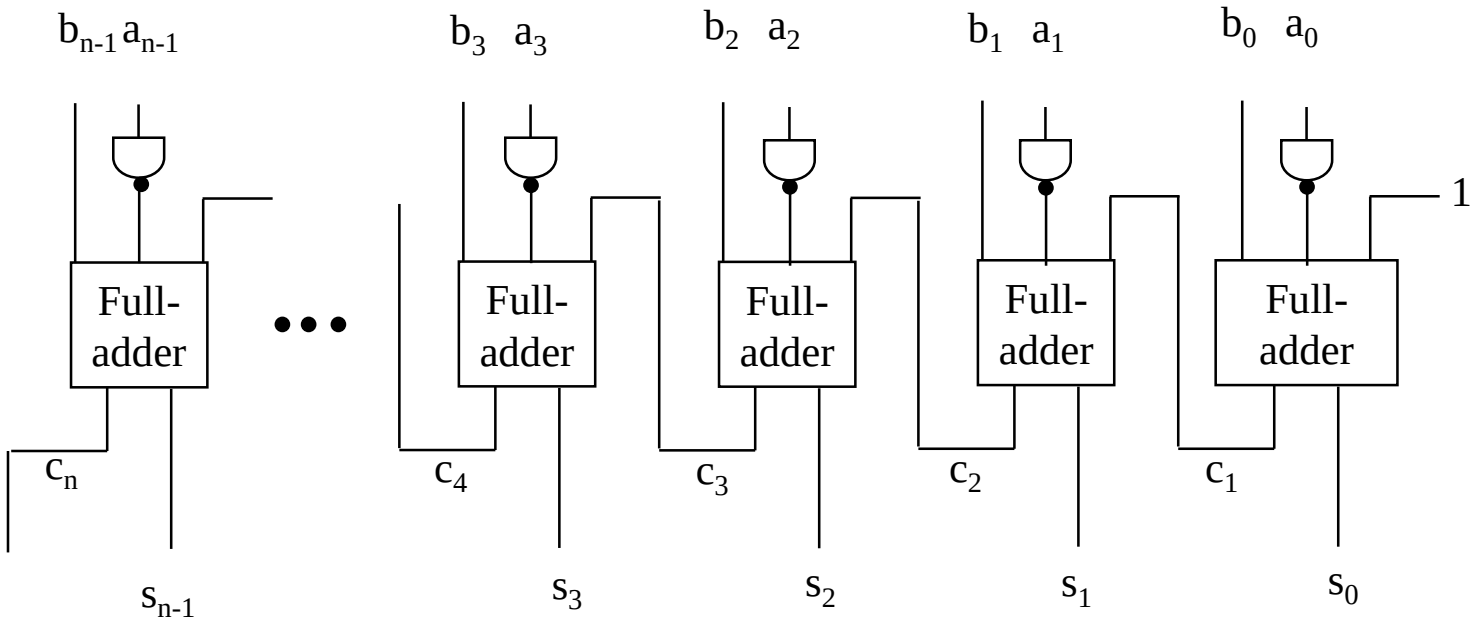
How can we create the 2's complement?

We invert every bit (1's complement) and then add a "1". To avoid using an additional adder, we can simply misuse the carry-input of the adder, in which we enter a 1 instead of a 0 for subtraction. We can see a corresponding combinatorial circuit in the two slides after this. Later, we will execute further operations using the adder. Thus, the interconnection of its input and output will get more extensively.

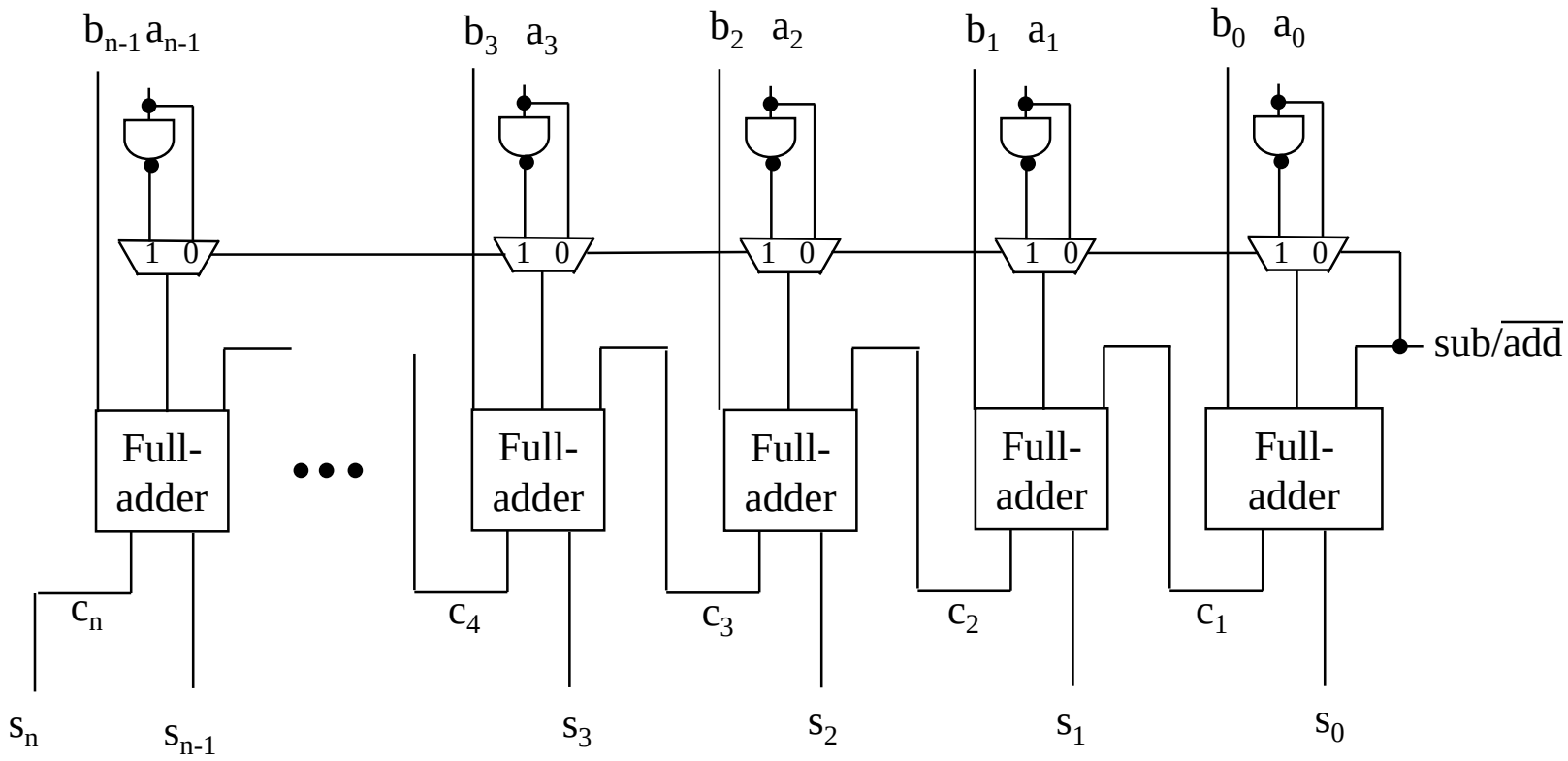
Adder


b_{n-1} a_{n-1}
b_3 a_3
b_2 a_2
b_1 a_1
b_0 a_0
0
Full-adder
Full-adder
Full-adder
Full-adder
Full-adder
c_n
c_4
c_3
c_2
c_1
s_{n-1}
s_3
s_2
s_1
s_0

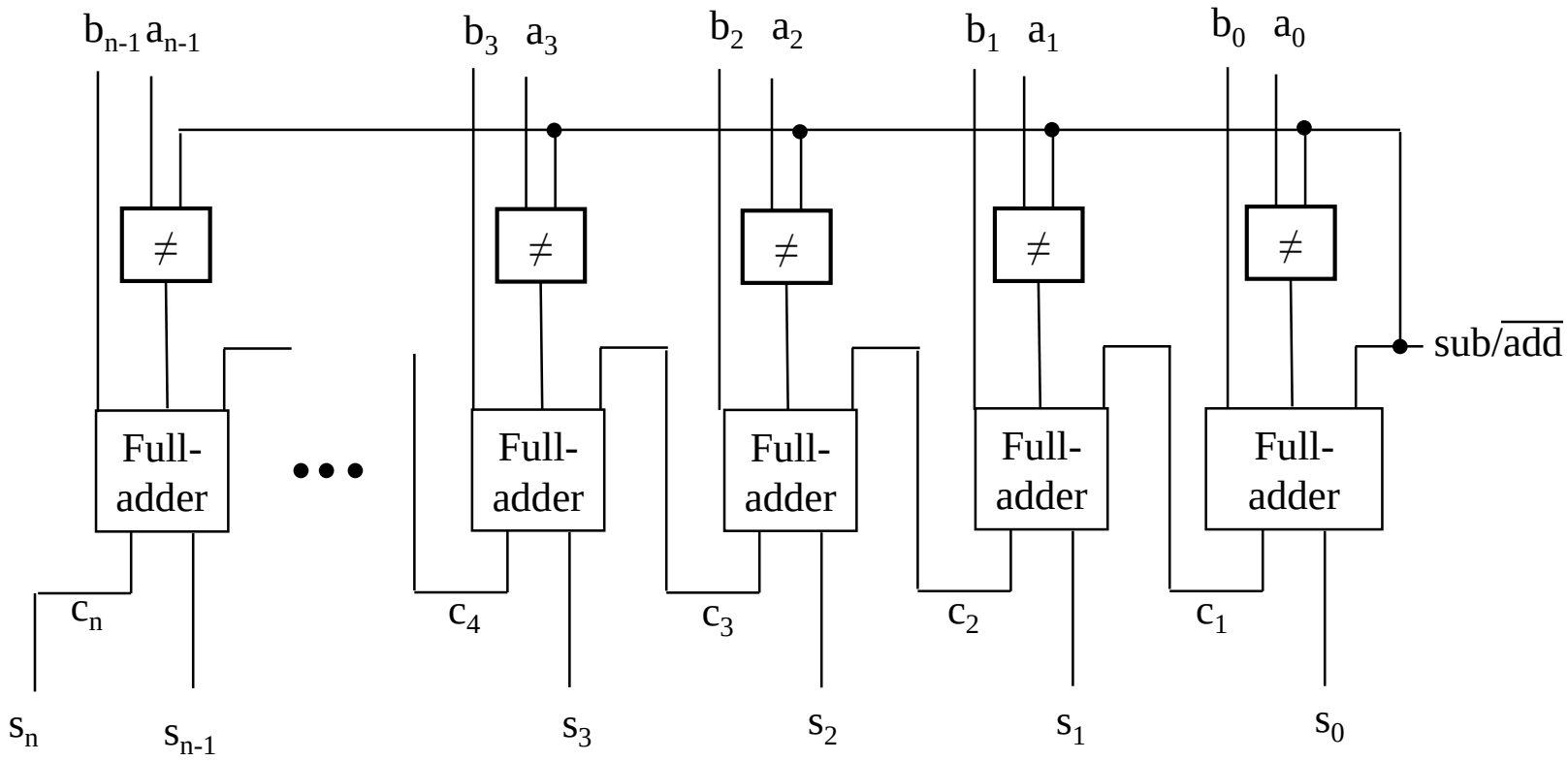
Subtractor


b_{n-1} a_{n-1}
b_3 a_3
b_2 a_2
b_1 a_1
b_0 a_0
1
Full-adder
Full-adder
Full-adder
Full-adder
Full-adder
c_n
c_4
c_3
c_2
c_1
s_{n-1}
s_3
s_2
s_1
s_0

Adder/Subtractor



Adder/Subtractor



Instruction Set

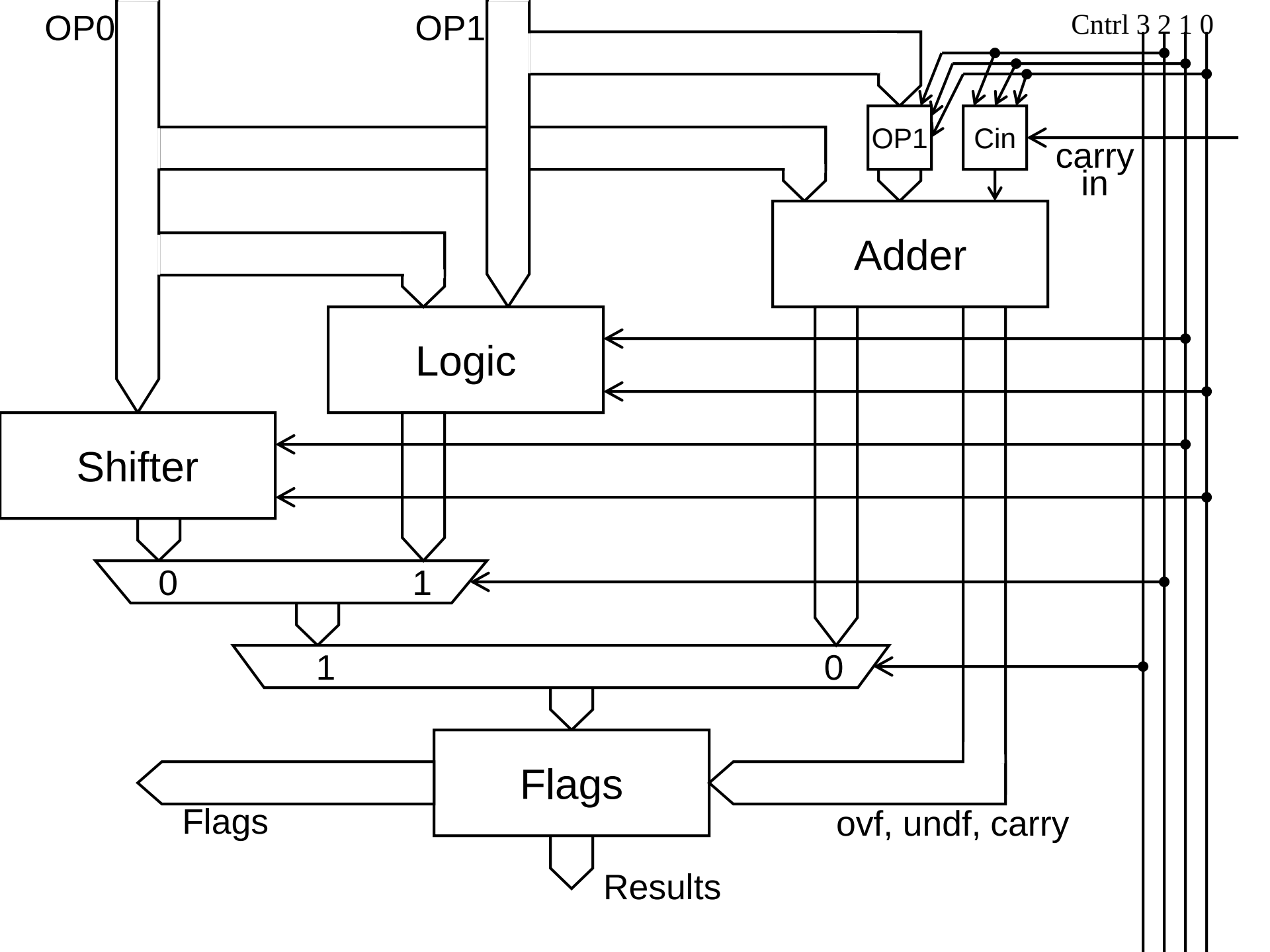
Now, we know which instruction set we would want our ALU to be able to execute. The following slide shows a typical selection of operations which a modern RISC processor can execute on the ALU. One can observe that the instruction selection contains some redundancy (the SET instruction exists twice whereby the logic operations can be coded different etc.). We decide to use this simple version in order to keep the implementation as clear as possible.
The 16 instructions are coded in four bits $cntrl_3$,...,$cntrl_0$. $cntrl_3$ decides if it is an arithmetic operation and $cntrl_2$ decides if it is a shift or logical operation and an addition or subtraction operation.

Instruction Set:

Instruction	Represents	Coding			
		cntrl3	cntrl2	cntrl1	cntrl0
SET	Result:=OP0	0	0	0	0
DEC	Result :=OP0-1	0	0	0	1
ADD	Result :=OP0+OP1	0	0	1	0
ADC	Result :=OP0+OP1 with $Carry_{in}$	0	0	1	1
SET	Result :=OP0	0	1	0	0
INC	Result :=OP0+1	0	1	0	1
SUB	Result :=OP0-OP1	0	1	1	0
SBC	Result :=OP0-OP1 with $Carry_{in}$	0	1	1	1
SETF	Result :=0	1	0	0	0
SLL	Result :=2*OP0	1	0	0	1
SRL	Result :=OP0 div 2	1	0	1	0
SETT	Result :=-1	1	0	1	1
NAND	Result :=OP0 NAND OP1	1	1	0	0
AND	Result :=OP0 AND OP1	1	1	0	1
NOT	Result :=NOT OP0	1	1	1	0
OR	Result :=OP0 OR OP1	1	1	1	1

Structure of the ALU

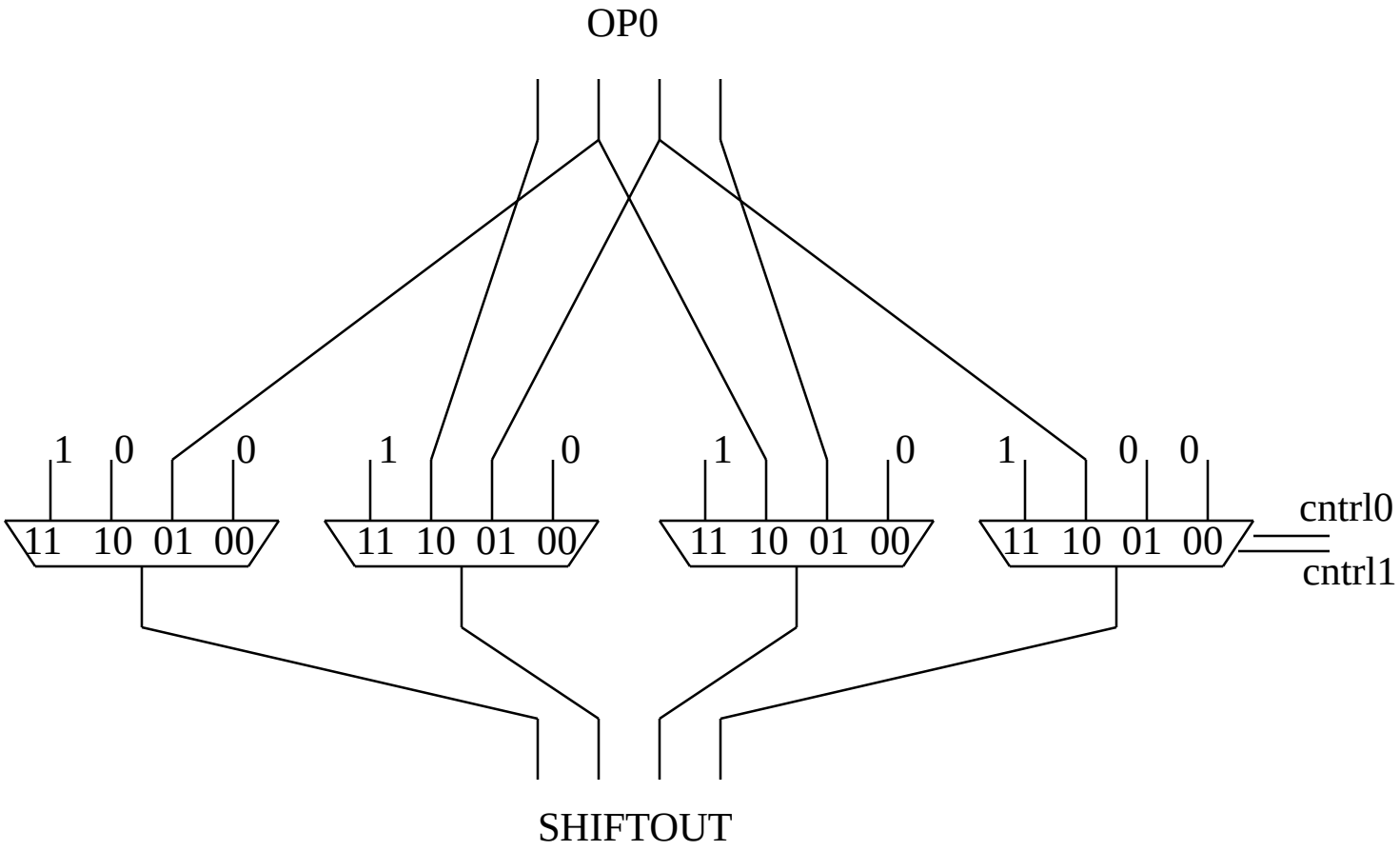
The next slide shows the fundamental structure of the ALU. The control inputs, on one hand, operates the units (shifter, logic, adder) itself and on the other hand, it selects the results through the control of two data-path multiplexers in order to forward the results of the corresponding unit to the ERG-output of the ALU.


OP0
OP1
Cntrl 3 2 1 0
OP1
Cin
carry in
Adder
Logic
Shifter
0
1
1
0
Flags
Flags
ovf, undf, carry
Results

The Shifter

We have already known what is a wire and a data-path multiplexer. The task remains to fill the units with "life", from which we have only a functional description based on the instruction set.
We begin with the simplest unit, the shifter. Its tasks are the instructions SETF (sets as FALSE, set to 0), SLL (shift logical left), SLR (shift logical right) and SETT (sets as TRUE, set to -1). These functions can be perceived with simple 4-to-1 multiplexer, one for every bit of the results. For both of the shift instructions, it respectively sets a 0 for the new bit that was shifted in and discard the bit that was shifted out.
If a rotation should be realised, one can realise a corresponding modification in the unit.
The following slide shows the shifter for a data width of four bits.

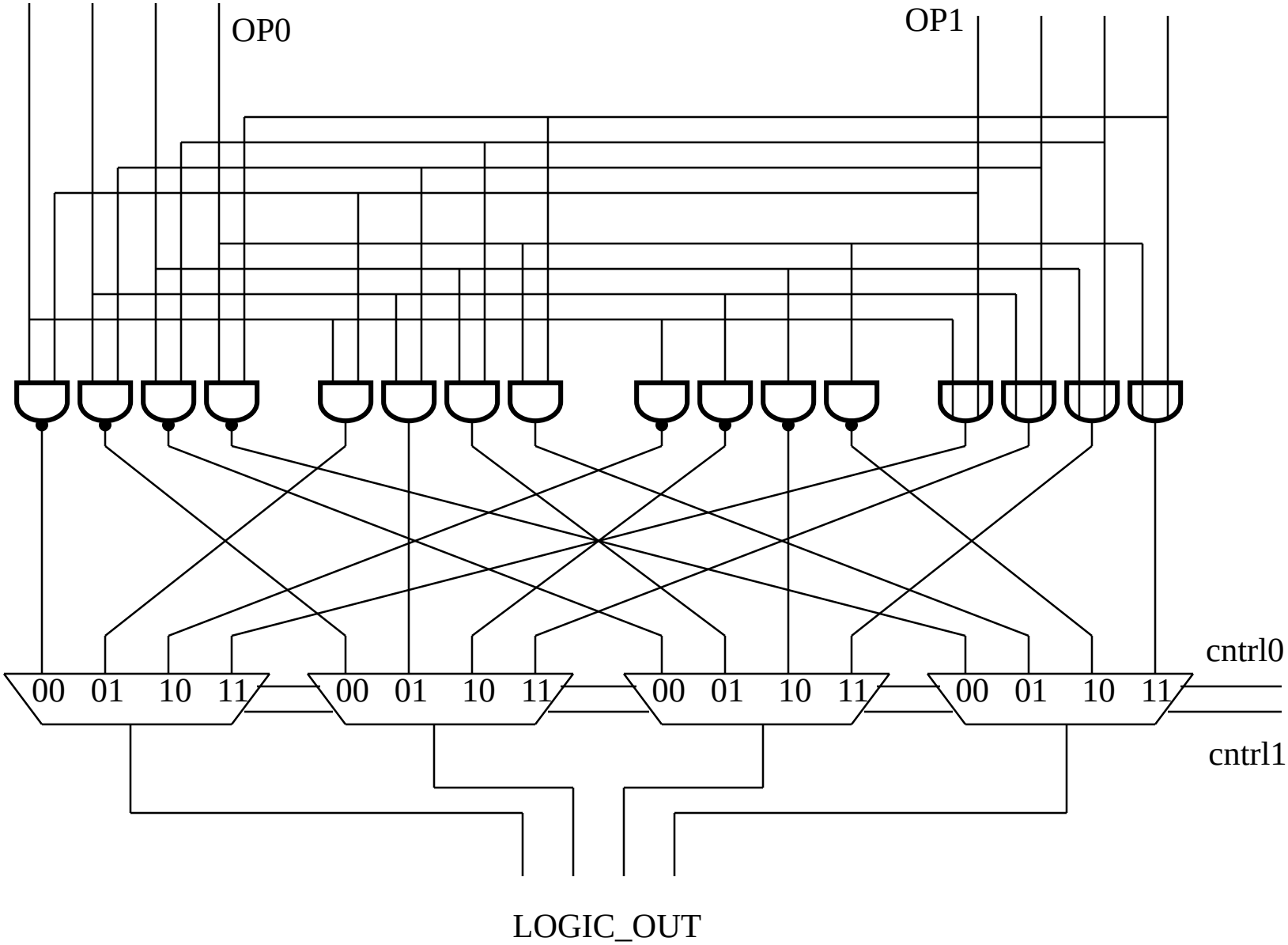
Shifter


OP0
1 0 0
1 0
1 0
1 0 0
11 10 01 00
11 10 01 00
11 10 01 00
11 10 01 00
cntrl0
cntrl1
SHIFTOUT

The Logic Unit

The following slide shows one (from many possible equivalent) realization of the logical instructions. NAND, AND, NOT, or OR are calculated separately bit-wise and through a multiplexer, one can select the results from the current instructions.

The Logic Unit



Inputs of the Adders

The adder has an operand that is always OP0. The second which can be added, can be OP1, 0, -1, 1, or -OP1, depending on the operation: addition, substraction incrementation, decrementation or unchanged. These cases are described as follows:

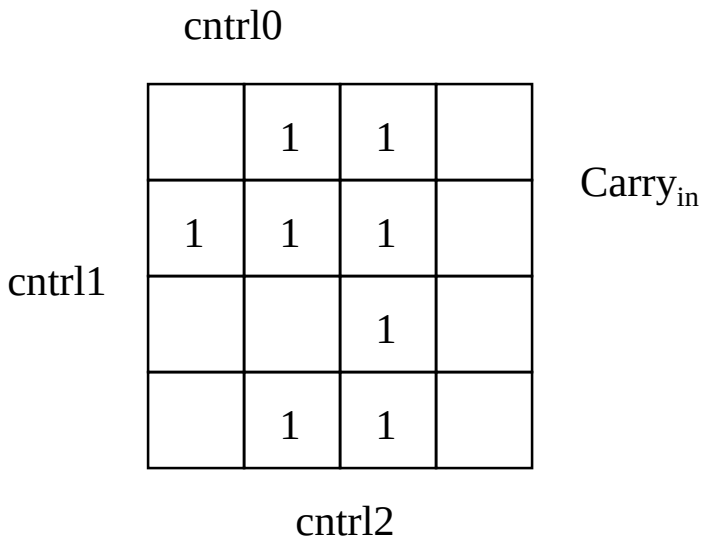
For a simple addition, the second input is equal to OP1 and the carry input is equal to 0. For an addition with considerations of the old $Carry_{in}$, the input is equal to OP1, the carry input is equal to $Carry_{in}$. For a subtraction, all bits of OP1 is inverted and the carry input is 1. In this way, the 2's complement of OP1 is added to OP0. For subtraction with carry, the bits of OP1 are inverted and $Carry_{in}$ is applied at the carry input of the adder.
For SET instructions, OP0 + 0 is calculated. There are two SET instructions. For the first a +0 is added and for the second, a -0 is subtracted. Thus, for the first operation (+0), the second adder input is equal to 0 and the carry is 0 and for the second operation, the second adder input is -1 (all bits are 1) and the carry is 1.
For increment, the second adder input is 0 and the carry input is 1. For decrement, the second adder input is -1 (all bits are 1) and the carry input is 0.
The combinatorial circuit for the carry input and the second input of the adder are derived on the following slides. The combinatorial circuit for the second input of the adder is allocated for each individual bit.

Inputs of the Adder

Instruction	$Carry_{in}$	cntrl2	cntrl1	cntrl0	C	OP1-Input
set	0	0	0	0	0	0
dec	0	0	0	1	0	1
add	0	0	1	0	0	OP1
adc	0	0	1	1	0	OP1
set	0	1	0	0	1	1
inc	0	1	0	1	1	0
sub	0	1	1	0	1	~OP1
sbc	0	1	1	1	0	~OP1
set	1	0	0	0	0	0
dec	1	0	0	1	0	1
add	1	0	1	0	0	OP1
adc	1	0	1	1	1	OP1
set	1	1	0	0	1	1
inc	1	1	0	1	1	0
sub	1	1	1	0	1	~OP1
sbc	1	1	1	1	1	~OP1

C-Input of the Adder

Instruction	$Carry_{in}$	cntrl2	cntrl1	cntrl0	C-Input
set	0	0	0	0	0
dec	0	0	0	1	0
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	1
sub	0	1	1	0	1
sbc	0	1	1	1	0
set	1	0	0	0	0
dec	1	0	0	1	0
add	1	0	1	0	0
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	1
sub	1	1	1	0	1
sbc	1	1	1	1	1



C-Input = $\overline{cntrl1}\ cntrl2 + \overline{cntrl0}\ cntrl2 + cntrl0\ cntrl1\ Carry_{in}$

Implementation as Complex Gate

We first observe the p-side: The disjunctive minimal form delivers immediately a fast and simple realization of the value, whose function is 1, in which we apply the inverted inputs as control of p- transistors and use these in series connection for a "AND" connection and parallel for an "OR" connection.

For the value, by which the function has the value 0, the n-side is responsible. In this case, the conjunctive minimal form is more suitable.

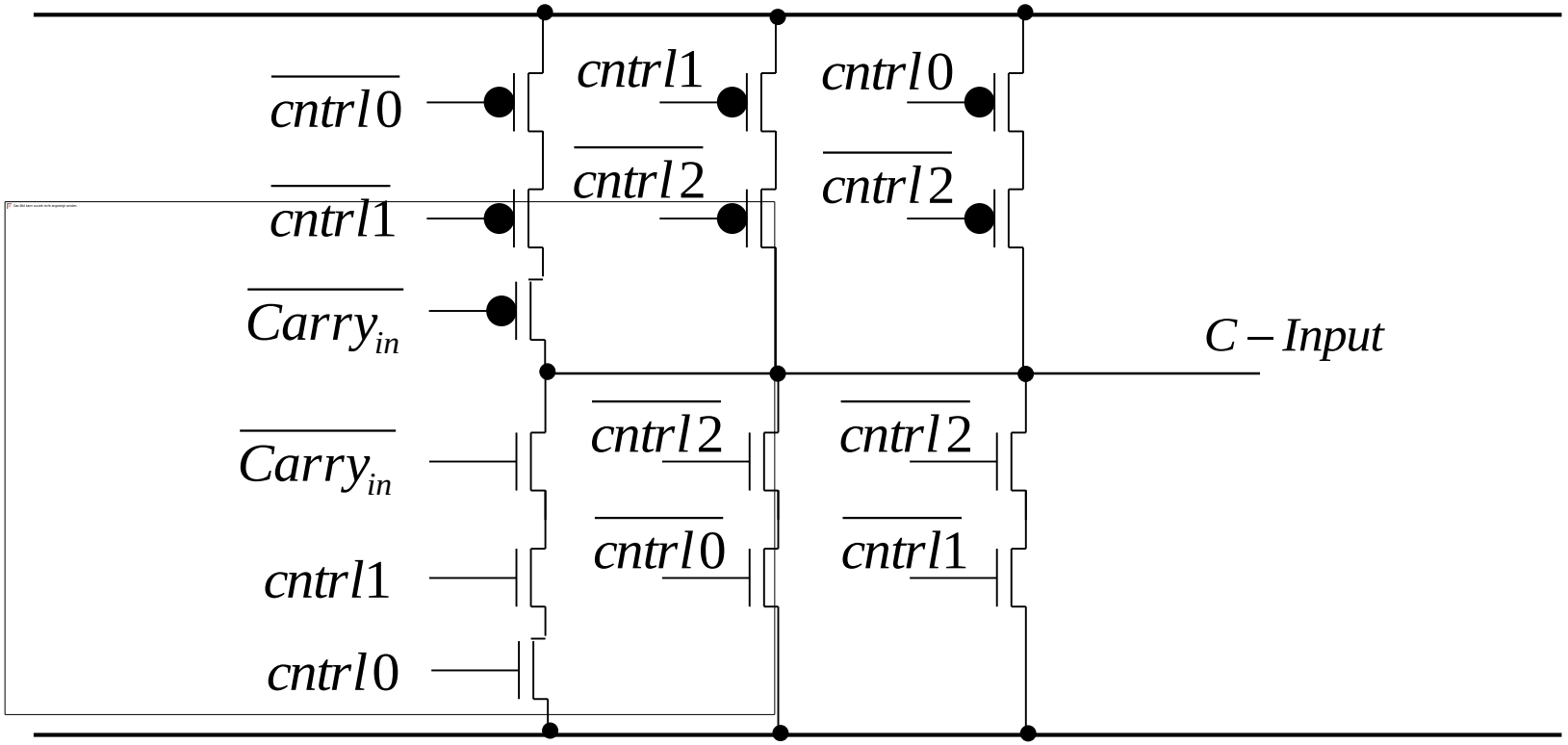
$$\overline{C-Input} = \overline{(cntrl0+cntrl2)\cdot(cntrl1+cntrl2)\cdot(\overline{cntrl0}+\overline{cntrl1}+Carry_{in})} =$$

$$= \overline{(cntrl0+cntrl2)} + \overline{(cntrl1+cntrl2)} + \overline{(\overline{cntrl0}+\overline{cntrl1}+Carry_{in})} =$$

$$= \overline{cntrl0}\cdot\overline{cntrl2} + \overline{cntrl1}\cdot\overline{cntrl2} + cntrl0\cdot cntrl1\cdot\overline{Carry_{in}}$$

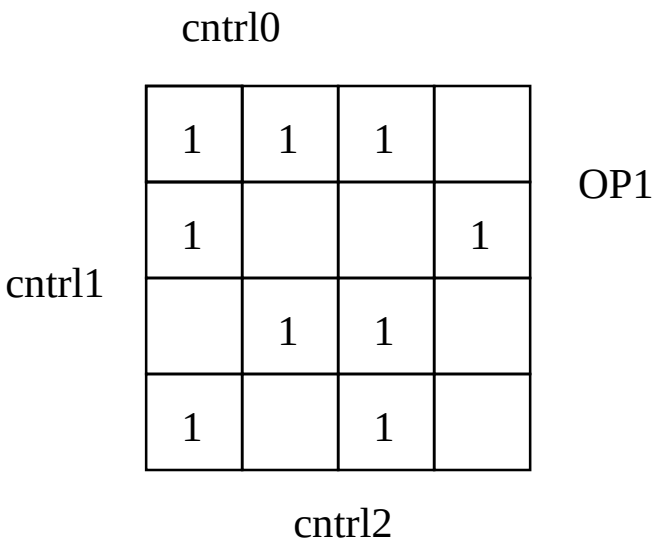
Implementation as Complex Gate

With that, we obtain the following complex gate:



OP1-Input of the Adder

Instruction	OP1	cntrl2	cntrl1	cntrl0	OP1-Input
set	0	0	0	0	0
dec	0	0	0	1	1
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	0
sub	0	1	1	0	1
sbc	0	1	1	1	1
set	1	0	0	0	0
dec	1	0	0	1	1
add	1	0	1	0	1
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	0
sub	1	1	1	0	0
sbc	1	1	1	1	0

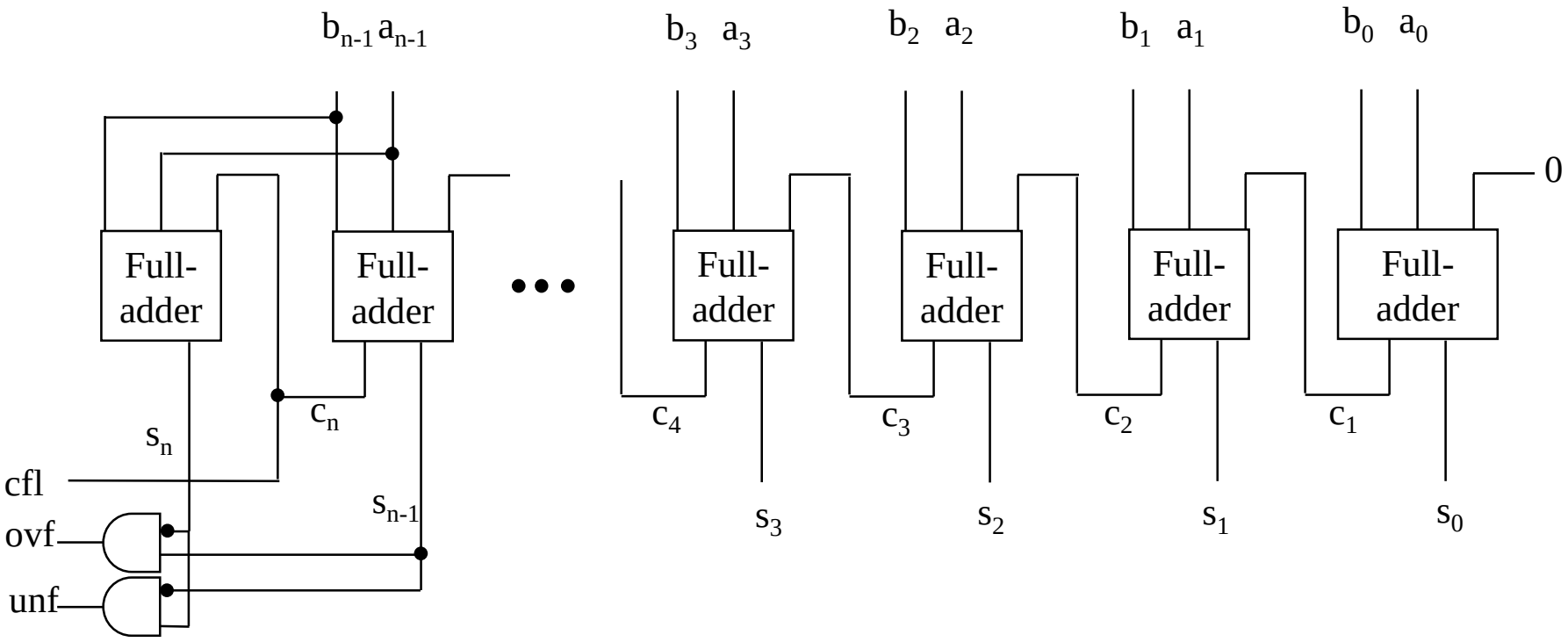


$$\text{OP-Input} = \overline{\text{cntrl1}}\ \overline{\text{cntrl2}}\ \text{OP1} + \text{cntrl1}\ \text{cntrl2}\ \overline{\text{OP1}} + \overline{\text{cntrl1}}\ \text{cntrl2}\ \text{OP1} + \text{cntrl0}\ \overline{\text{cntrl1}}\ \overline{\text{cntrl2}} + \text{cntrl0}\ \text{cntrl1}\ \text{cntrl2}$$

Overflow-Recognition of the Adder

As shown already in the first lecture, the over- and under flow of the addition can be known with the help of one additional security bit which represents a copy of the most significant bit. One uses now for an n-bit addition, a n+1 bit adder, whereby the securing position is added as per normal. The result of the addition is therefore the summation of s_{n-1}, s_{n-2}, ..., s_1, s_0, the outgoing carry bit (= carry-flag) c_n, as well as an artificial sum bit s_n. The artificial carry bit c_{n+1} has no meaning and will not be used. An overflow (result is greater than the greatest representable number) is now triggered, if s_{n-1} is equal to 0 and s_n is equal to 1. When s_n is equal to 0 and s_{n-1} is equal to 1, an underflow has been found (result is smaller than the smallest representable number). When s_{n-1} is equal to s_n, neither the overflow nor the underflow has occurred, and thus the addition is executed without any errors. The combinatorial circuit is shown on the following slide where it shows the recognition of over- and underflow conditions that complement our adders and they are represented as flags (ovf (overflow) and unf (underflow)). These flags can then be transmitted to the flag unit.

Signals for Flags



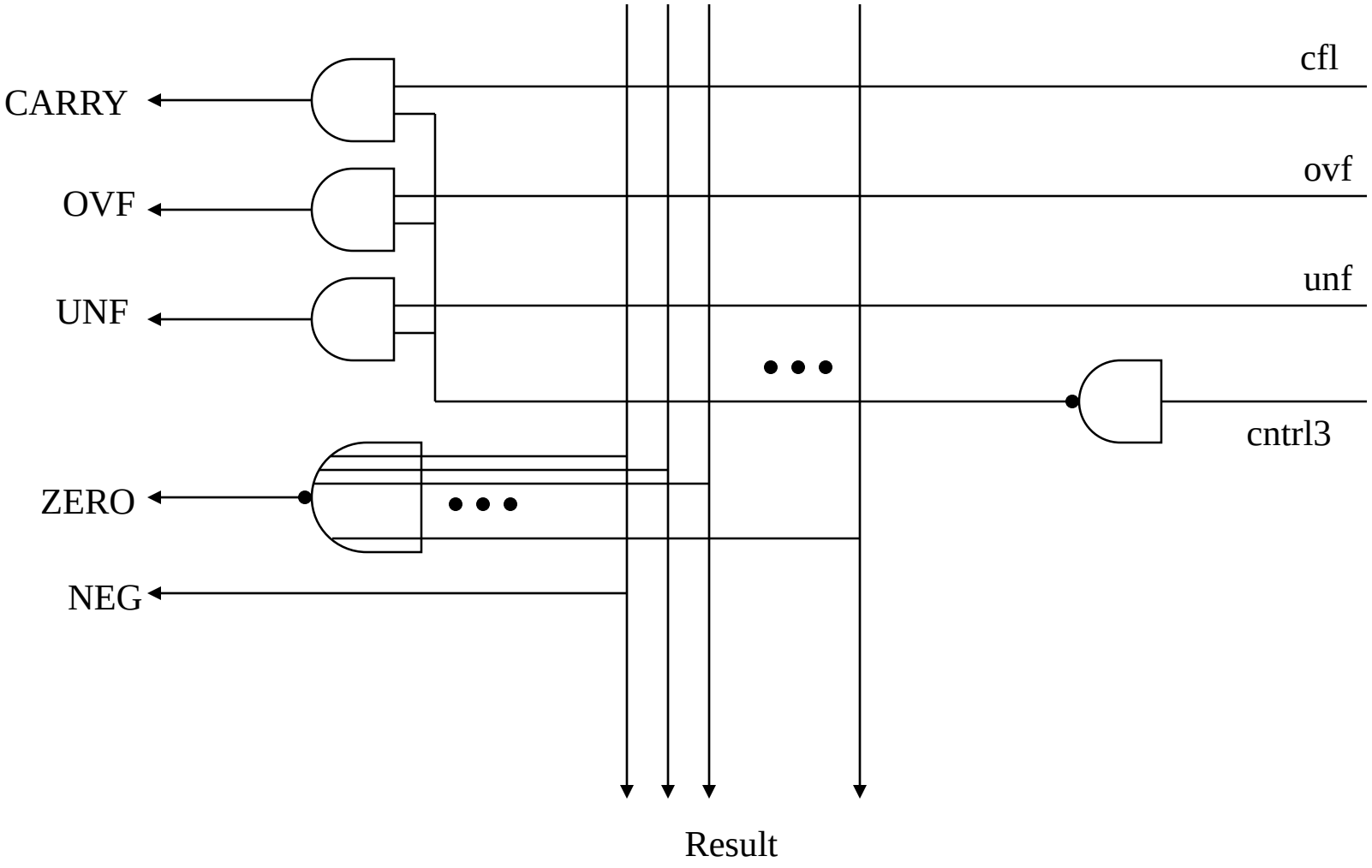
Flag-Test

Five flags shall be generated:

- Carry-Flag (=1, in the case when an arithmetic operation has generated a Carry)
- Neg-Flag (=1, when the result represents a negative number)
- Zero-Flag (=1, when the result is equal to 0)
- ovf-Flag (=1, when an arithmetic operation has generated an overflow)
- unf-Flag (=1, when an arithmetic operation has generated an underflow)

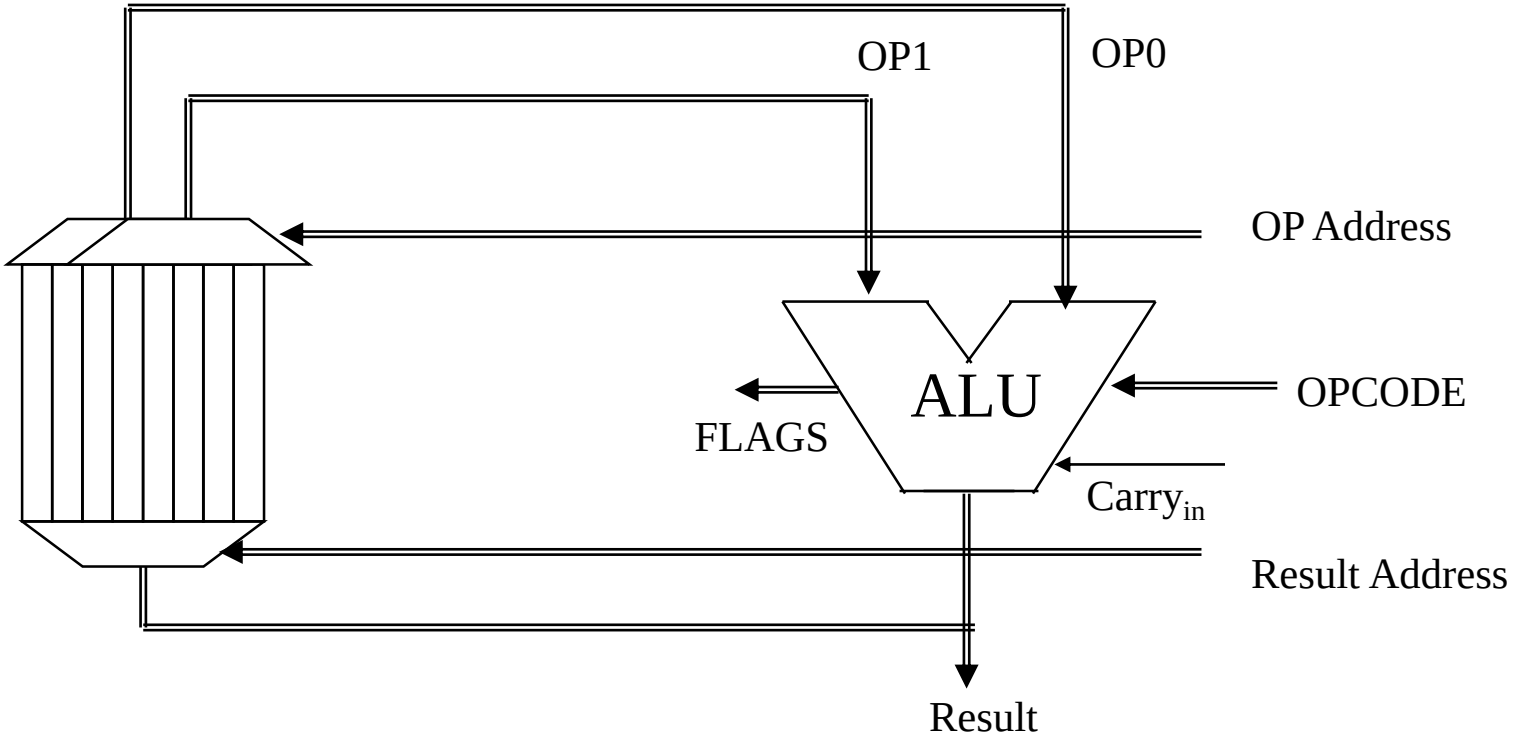
In each individual case, the requirements on the flag unit can be very complicated, especially when it concerns a processor with implicit condition handling. For our purpose, it is sufficient to use such a simple unit, as represented in the following slide.

Flag-Test



Adding of Registers

When we now add a register set to our ALU, we have thus already built the core part of a (very simple) processor:



Example: Multiply Two Positive Numbers

We want to see how a program can calculate something using this "processor". We want to calculate the product of two positive numbers and the two numbers shall be stored in the register R0 and R1 at the beginning. The result shall be in register R6 and R7 at the end. Positive 4-bit numbers can take on a value between one and seven. Therefore, the result is between one and 49. With that, one register is insufficient to represent it. R6 shall, at the end contain the high significant part and R7 shall contain the low significant part of the result.

We need some help register:
R2: Contains the 1 and it is necessary to check if the LSB of R0 is equal to 1.
R3: Contains the 0 or exactly a 1 in differentiating bit positions. It is necessary to generate the bit order 0000 or 1111 in R4.
R4: It is necessary to be used as a mask register in order to execute the 4-bit multiplication with R1. Furthermore, it acts as a help register for the double length addition.
R5: Contains the partial product for the respective bit R0 which is to be modified. It is used to add this partial product to the current sum in R6 and R7.
R6: High significant result register.
R7: Low significant result register.

```
SETF    R6          Initialization of R6 with 0
INC     R2,R4       R2 is now 1


AND     R3,R0,R2    Is the last bit of R0 = 1 ?
SUB     R4,R6,R3    R4 is now 1111 or 0000

AND     R7,R1,R4    Multiplication of R1 with the last bit of R0
SHR     R0          Next bit of R0
SETF    R4          Prepare next mask register

AND     R3,R0,R2    Is the last bit of R0 = 1 ?
SUB     R4,R4,R3    R4 is now 1111 or 0000
```

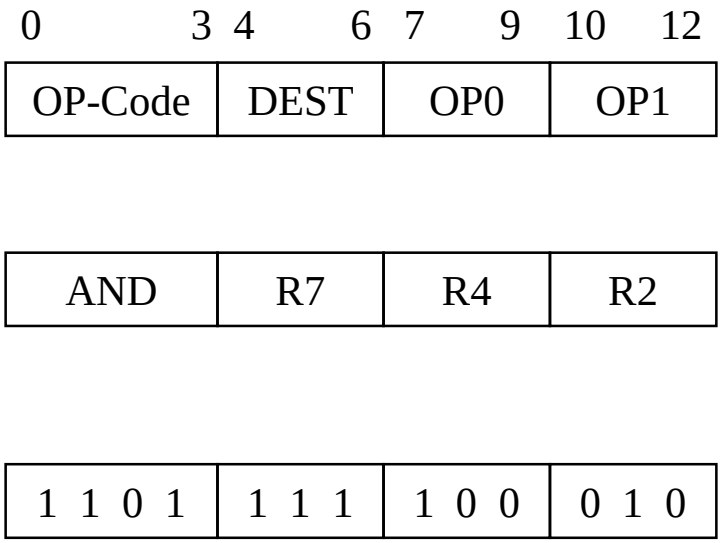
```
AND     R5,R1,R4    Multiply the second bit of R0 with R1
SETF    R4          Initialization of R4 with 0
ADD     R5,R5,R5    Shifting of R5 by one bit to the left (with carry)
ADC     R4,R4,R4    Detect a possible incoming carry
ADD     R7,R5,R7    Add to present sum (least significant part)
ADC     R6,R4,R6    Add to present sum (high significant part)

SHR     R0          next bit of R0
SETF    R4          prepare next mask register

AND     R3,R0,R2    Is the last bit of R0 = 1 ?
SUB     R4,R4,R3    R4 is now 1111 or 0000

AND     R5,R1,R4    Multiply the second bit of R0 with R1
SETF    R4          Load R4 with 0
ADD     R5,R5,R5    Shifting of R5 by one bit to the left (with carry)
ADC     R4,R4,R4    Detect a possible incoming carry
ADD     R5,R5,R5    Shifting of R5 still by one bit to the left (with carry)
ADC     R4,R4,R4    Detect a possible incoming carry
ADD     R7,R5,R7    Add to the current sum (least significant part)
ADC     R6,R4,R6    Add to the current sum (high significant part)
```

Example: R0=5, R1=6
Register assignment:

R0	R1	R2	R3	R4	R5	R6	R7
5	6	0	1	0	0	0	0
2		1	2	1	6	0	6
1			4	3	C	1	6
			8	7	8		E
			0	F			
			0	0			
			0	0			
			0	0			
			1	0			
			2	1			
			4	3			
			8	7			
				F			
				0			
				0			
				1			

Instruction Format

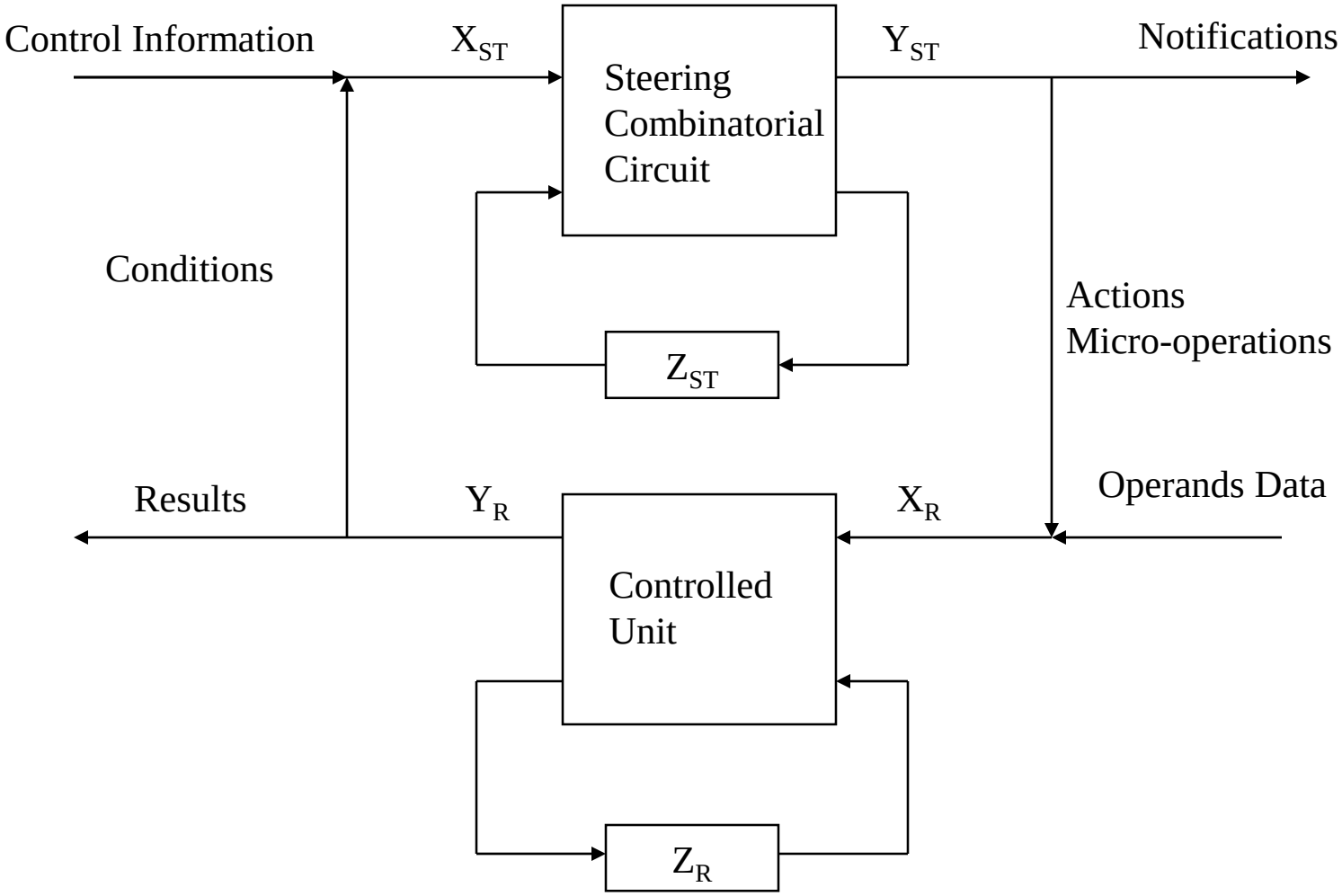
How should an instruction for such a processor be assembled?



Control Unit

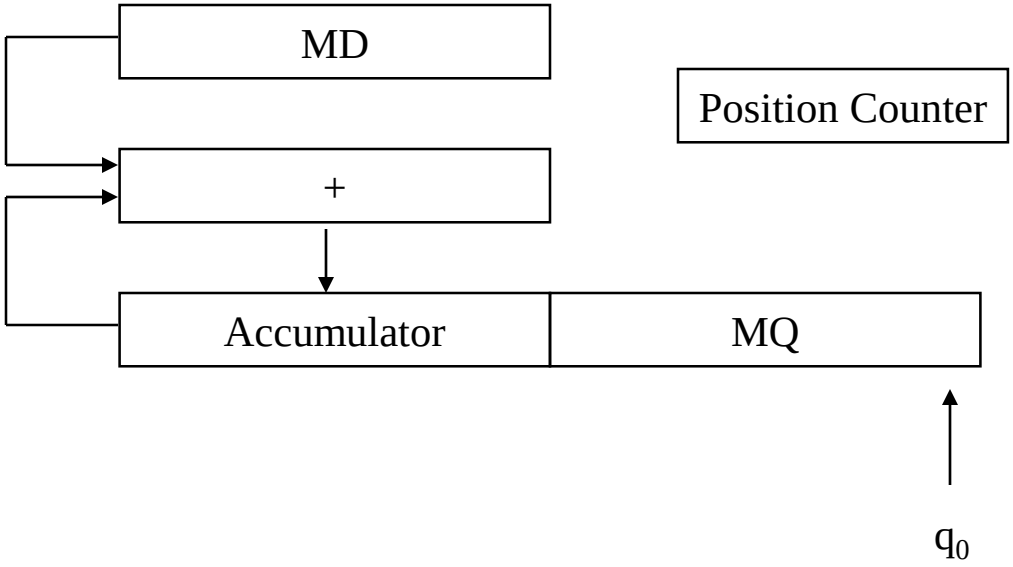
We have now known about the assembly of the arithmetic logic units, e.g. an adder or a logical unit. In a computer, these units must however, be activated at the right time and the data must be forwarded to the correct units over the multiplexer. In other words: the flow must be controlled. These functions behave likewise as sequential circuits, the so-called control units. We want to study a simple example of this function of a control unit in this section.
The following slide shows the basic principles of a control unit: It has a steering combinatorial circuit and a controlled unit. The controlled unit is often an arithmetic logic unit. The control unit has – like every combinatorial circuit – inputs, outputs and states. The inputs are on one hand, the instructions of the overriding instances (e.g. the users or an overriding control unit), and on the other hand, they are also the **conditions**. These are the outputs of the arithmetic logic unit, with which the controlled unit provides information to the controlling combinatorial circuit about the states of the controlled task. The tasks of the steering unit are on one hand, the notifications to the overriding instances and on the other hand, the **micro-instructions (actions)**, which enter the controlled unit as inputs and from there the operations are supervised.

Control Unit Principles



Multiplication Unit

The following example shall illustrate the principles of the control unit:
A multiplication unit is to be controlled so that it can multiply two numbers (e.g. with four bits each). The multiplication unit consists of a multiplicand register MD, an accumulator register AKK, which acts as a parallel register and can be used as a shift register, a multiplicator register MQ with the same properties, a counter, a so-called position counter (SZ) and an adder. The last position of MQ is shown using q_0.



Multiplication Unit

For a multiplication, the multiplicand and multiplicator are loaded in the corresponding register and the accumulator would be preset by a 0. Then, the control sends a micro instruction "-n", whereby the position counter is then set to -n. n shall be the number of positions of the multiplication operands. If q_0 is 1, the micro-operation "+" is then generated, which results in the addition of the existing accumulator to the register MD and the result will be saved in the accumulator. Thereafter, the micro-operations, "S" and "SZ" will be generated. S will cause a shift by one bit to the right in the shift register which consists of an accumulator and MQ. SZ causes an increment in the position counter. When the value in the position counter has reached a 0, the process terminates and the result is available in the shift register. When the position counter is not 0, q_0 will once again be interpreted. When q_0 is 0, the micro-operation "0" will be generated which modifies no register.

The following page shows an example for the multiplication of the numbers 0101 and 1011
(5 * 11 = 55).

Sequence in the Multiplication Unit

MD = 0101

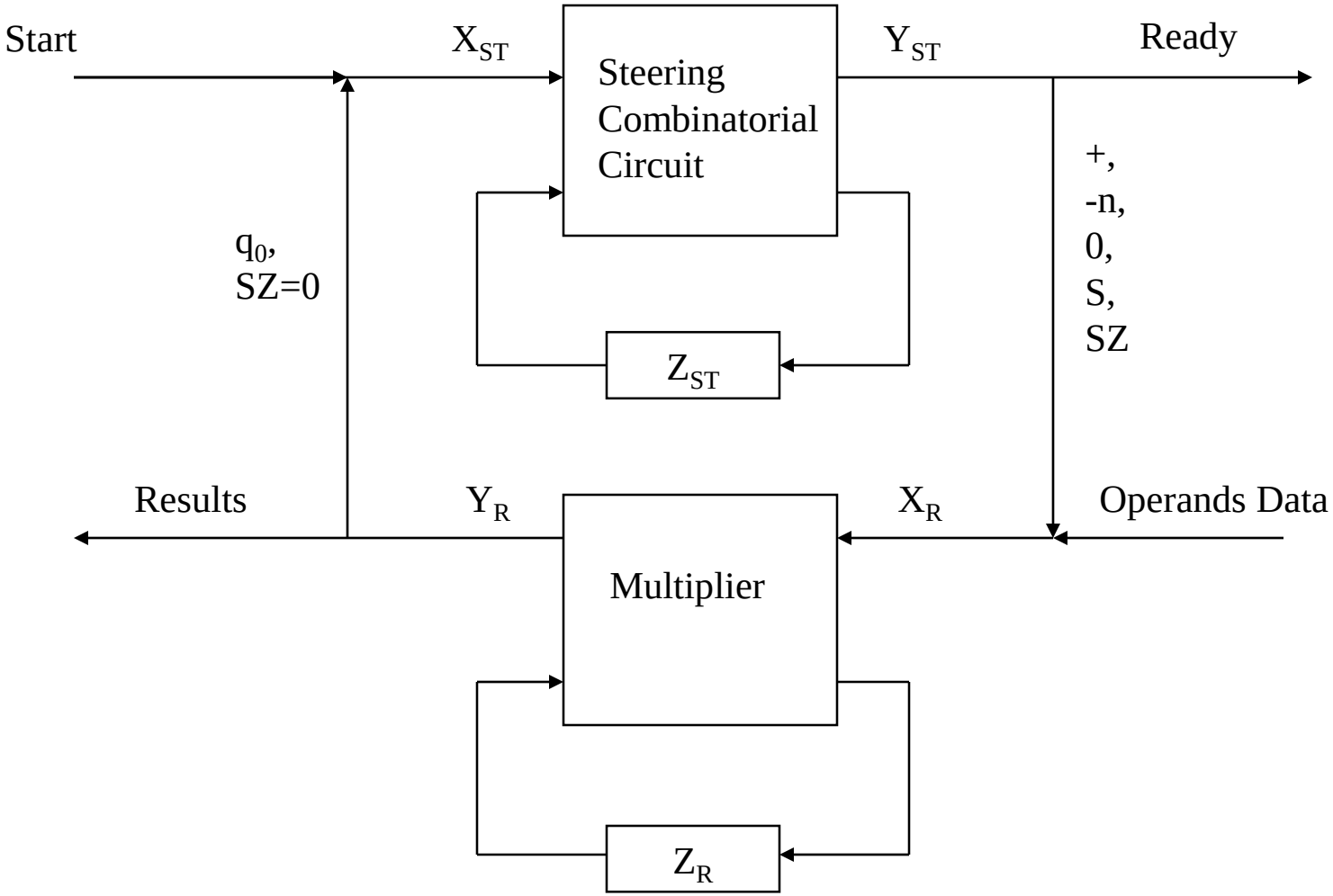
ACCU	MQ	SZ	q0	SZ=0	Start	Micro-operation
0 0 0 0	1 0 1 1	000	1	1	1	-n
0 0 0 0	1 0 1 1	100	1	0	0	+
0 1 0 1	1 0 1 1	100	1	0	0	S, SZ
0 0 1 0	1 1 0 1	101	1	0	0	+
0 1 1 1	1 1 0 1	101	1	0	0	S, SZ
0 0 1 1	1 1 1 0	110	0	0	0	0
0 0 1 1	1 1 1 0	110	0	0	0	S, SZ
0 0 0 1	1 1 1 1	111	1	0	0	+
0 1 1 0	1 1 1 1	111	1	0	0	S, SZ
0 0 1 1	0 1 1 1	000	1	1	0	

We already know how we must build such an arithmetic logic unit, it consists solely of the components known to us (Register, shift register, counter, adder). Now, it interests us as in how we can generate the micro-operation with the right timing and thus, how we can control this arithmetic logic unit. With that, we learn:

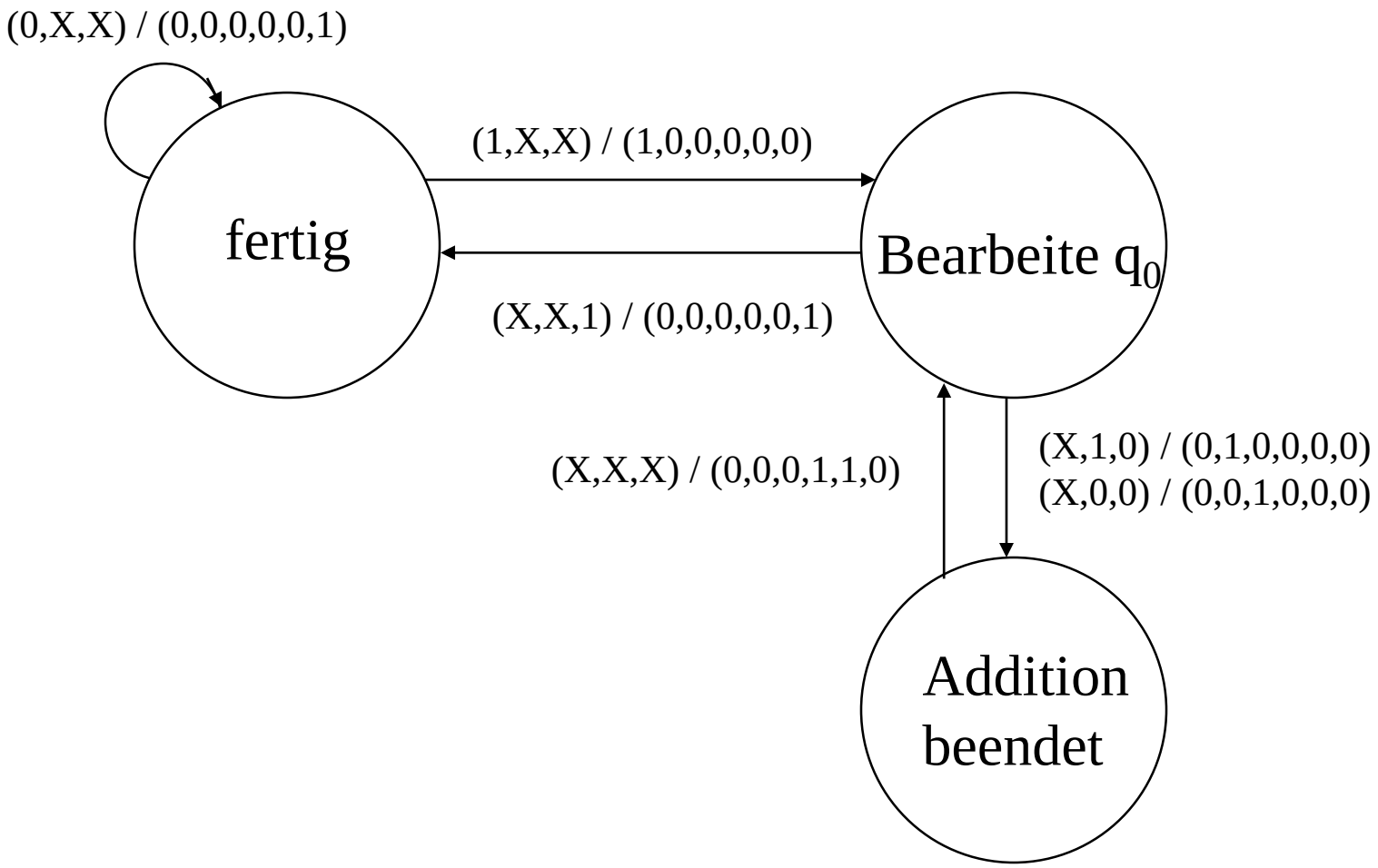
- When a "Start"-Signal is obtained, the position counter must be initialized with "-n".
- When q_0 would be interpreted, MD must then be added to the accumulator exactly when $q_0 = 1$.
- After every such addition step, the accumulator and the MQ must shift by one bit and the position counter must be incremented by one.
- Whenever the position counter has reached the value 0, the multiplication has ended and the result is available in the shift register of the accumulator and MQ.

The following slide shows which signals are inputs and which signals are outputs and to which unit. With this information, we can then design an Automatengraph.

Control Unit



Inputs: (Start, q_0, SZ=0) Outputs (-n, +, 0, S, SZ, ready)



We code the states with

00: Ready
01: Start Counter
10: Addition Ended
11: Shifting Ended

Thus, the following truth table represents this Automat

Start	q0	SZ=0	z1	z0	z‘1	z‘0	-n	+	0	S	SZ	Ready
0	X	X	0	0	0	0	0	0	0	0	0	1
1	X	X	0	0	0	1	1	0	0	0	0	0
X	1	X	0	1	1	0	0	1	0	0	0	0
X	0	X	0	1	1	0	0	0	1	0	0	0
X	X	X	1	0	1	1	0	0	0	1	1	0
X	1	0	1	1	1	0	0	1	0	0	0	0
X	0	0	1	1	1	0	0	0	1	0	0	0
X	X	1	1	1	0	0	0	0	0	0	0	1

After minimization, this results in the circuit realization on the next slide:

Realization as FPLA: