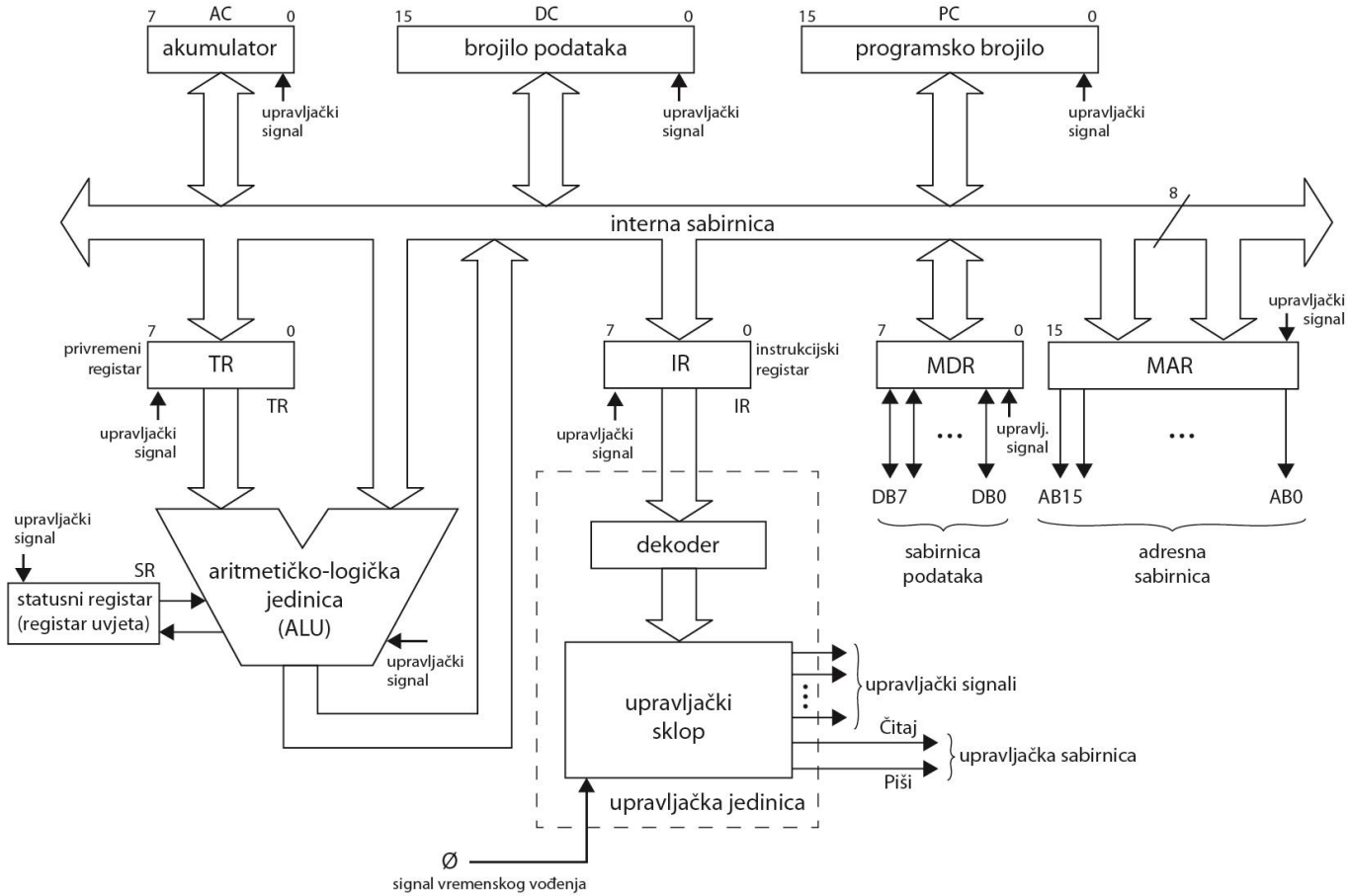
Computer architecture

Elements of ALU

Elements of standard CPU architecture

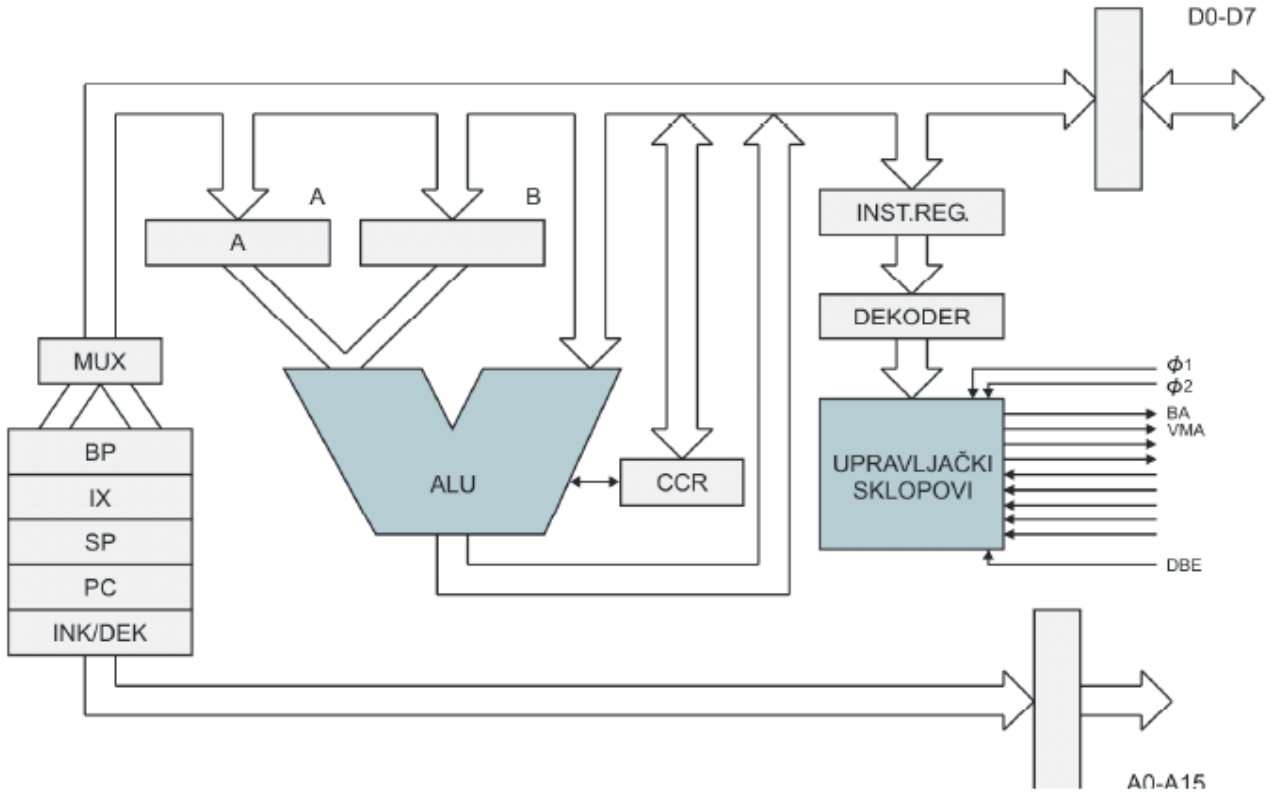
- Control unit
- Arithmetic-Logic unit (ALU)
- One or more accumulators
- General registers
- Address registers
- Internal busses

Simple CPU model



MC 6800

Primjer: MC 6800



CPU sub-systems/units

- CPU has :
  - Data handling units
  - Control units

Data handling units

- Arithmetic Logic unit
- Accumulators
- General purpose registers (GPR)
- Status registers (PSW, CCR)
- Address registers
- Segment registers

The role of ALU

- Common ALU operations:
  - Arithmetic operations (+, -, …)
  - Logical operations (AND, OR, …)
  - Data movement operatins (Load, Store)
- ALU name comes from the basic operations that it does
- We implement ALU by using combination of different types of circuits

ALU design components

- ALUs that perform multiplication and division are designed around the circuits developed for these operations while implementing the desired algorithm.
- More complex ALUs - designed for executing floating point, decimal operations and other complex numerical operations
- These are often times called *coprocessors*
- They work in tandem with the main processor

ALU design specifications

- The design specifications of ALU are derived from the Instruction Set Architecture
- The ALU must have the capability to execute the instructions of ISA
- An instruction execution in a CPU is achieved by the movement of data/datum associated with the instruction. This movement of data is facilitated by the datapath
- For example, a LOAD instruction brings data from memory location and writes onto a GPR

von Neumann computer, yet again!

- Program to be executed be stored in binary format in a memory device so that intermediate results could cause the desired effect to the program.
- This is also known as the Stored Program Concept in Computer design
- The stored program concept says that the program i.e. instructions are stored along with the data in the computer’s memory in machine-readable binary form(machine language)

Why do we care?

- ALL computers work like this
- This is the default way of program execution
- If we didn’t have this concept, all instructions would be executed manually – useless, impractical and completely impossible to use

von Neumann architecture – deeper concepts

- This execution model needs to have certain units, the basic building blocks of von Neumann computer:
  - ALU
  - Memory
  - Input/output
  - Control unit
- ALU has to have a register called Accumulator
- Control unit has to have a register/counter called PC (*Program Counter* – it exists to follow which is the next instruction
- These registers are usually implemented as memory components in CPU, so that any intermediate result could cause the desired effect to the program. This is also known as the Stored Program Concept in Computer design.

How does von Neumann computer work?

- Executes or emulates Fetch-Decode-Execude sequence for program execution:
  - *Fetch* the instruction from memory at the address denoted by the Program Counter.
  - Increment the program counter to point to the next instruction to be fetched.
  - *Decode* the instruction using the control unit.
  - *Execute* the decoded instruction using the data path, control unit and ALU. As part of this step, any data that is to be fetched from memory for executing the instruction is brought. Also, if any results to be updated in memory is written into memory, at the end of the execution of an instruction.
- Go back to beginning after this sequence

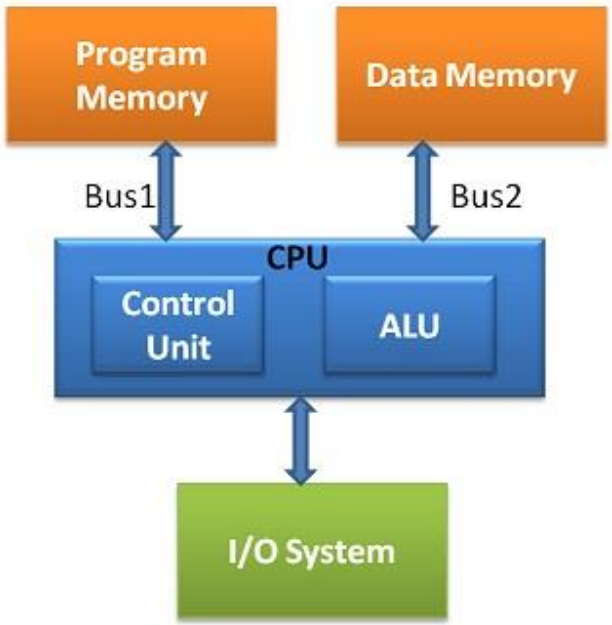
von Neumann computer design issues

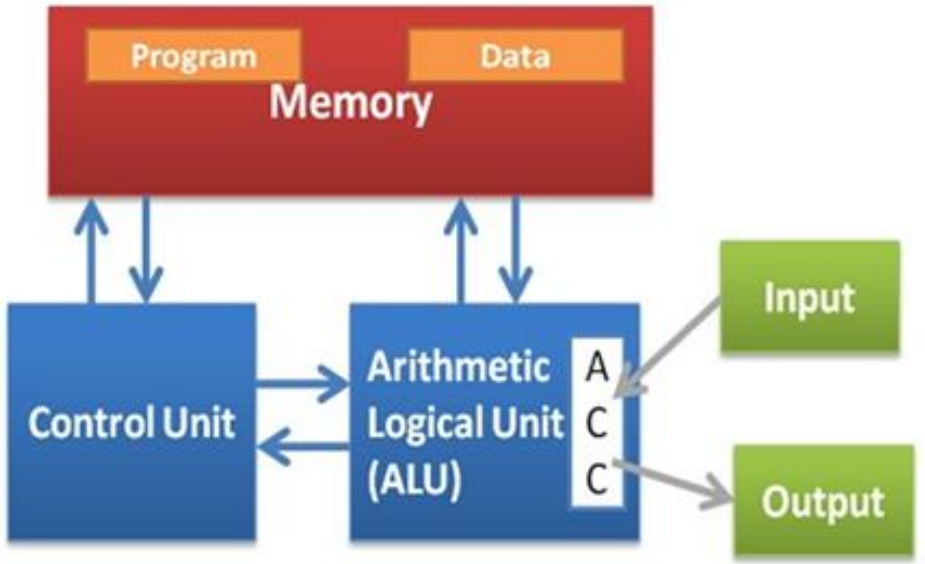
- It was designed to process instruction one by one (no parallelism) – CPU can’t do Fetch and Execute at the same time
- A special, very annoying aspect of that problem is the fact that instructions and data use the same data path -> contention for memory resources
- The fact that memory cycle is way longer than the CPU cycle – CPU waits for a long time for both of them to be “in sync”, which slows the process down even more
- This is what we call the “von Neumann bottleneck”

Solution to these issues?

- The Solution is (yet again) – better design (co-design)
  - Memory and CPU as different busses/interfaces, one for instructions, one for data
  - Design the CPU –Memory interface with two busses, exclusively one for instructions, and the other for data.
  - Designing CPU with Cache increases the bandwidth utilization between CPU and main memory, thereby reducing the waiting time of CPU.
  - Use of modern functional programming and object-oriented programming reduces the need for pushing vast numbers of words back and forth, than the conventional programming languages of the olden days like FORTRAN.
- FYI, one of the alternative architectures that works on these problems is *Harvard architecture* – separate busses for storage, instructions and data

von Neumann computers vs Harvard computer, in picture





Different CPUs – number of bits

- Intel 4004 (1971.) - 4 bit
- 1st gen (kraj 1971.) - 8-bit
- Intel 8086 (1978.) - 16-bit
- 3rd gen - 32-bit
- 4th gen - 64-bit
- GPUs and special purpose processor (VLIW, multimedia processors) – 128-bit and more

Character display

- Alphanumeric characters (A – Z, a – z, 0 – 9, symbols like *, -, +, !, ?, @) are represented by binary code
- **ASCII** (*American Standard Code for Information Interchange)* is The Standard for alphanumeric character coding
- Every AN sign is represented by a 7-bit code
- 128 characters, 96 out of which is “normal” (*printing characters),* ***32*** characters are non-printing ones, for various types of control functionality

ASCII code

		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NULL	DCL	SP	0	@	P	‘	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	’	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	}
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

ASCII kod

Unicode

- ASCII problem – different countries, different code pages
- Unicode uses unique number for every sign, no matter the platform, program or language
- It’s being used to represent ANY sign from EVERY known language
- It uses 16 bits
- We can code 2^{16} = 65536 different characters

Integer display

- Most computers uses positional numbering systems with base 2 (binary system)
- There are special computers that use symbols to display numbers
- Something that’s a type integer encapsulates:
  - Positive integer
  - Zero
  - Negative integer

Integer display

- There are different display methods:
  - Sign-absolute value (signed and magnitude)
  - One’s complement
  - Two’s complement

Floating point numbers

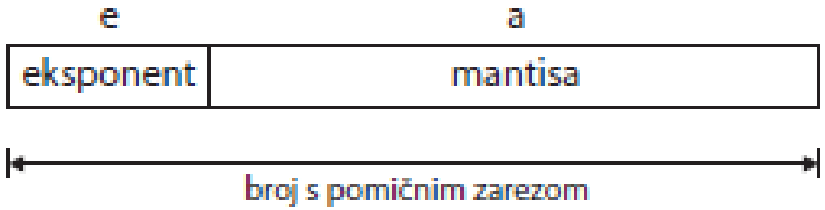
- Most often, we use full or second complement to display floating numbers
- Properties for second complement:
  - it's a real complement as +X +(-X) = 0
  - there's only one zero
  - for positive numbers - MSB 0, for negative numbers - MSB 1
  - range -2^n-1 do $+2^n-1-1$
  - second complement of X is X

Floating-point numbers

- Most common use case - scientific calculations
- Big range - like from 2^{-120} to very big values, like 2^{+120}
- Higher programming languages allow us to use variables of type real, and they can have values between two sequential integers
- These numbers are usually presented as floating binary numbers

Floating-point numbers

- Two parts
  - exponent;
  - mantissa;
- Usually:
- $a \times r^e$
  - a - *mantissa*
  - r - *number system base*
  - e - *exponent*



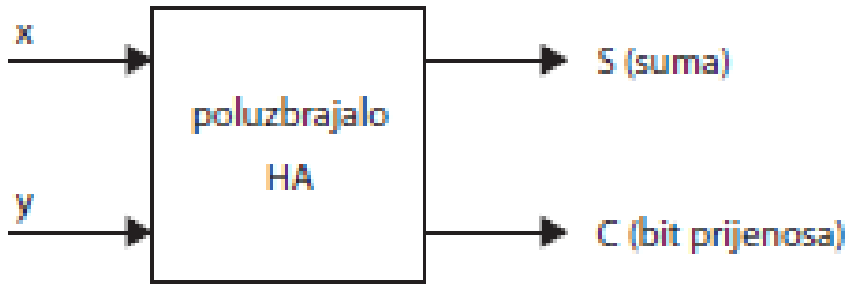
FP numbers

- Usually stored in computer as bit sequence with two fields:
  - Exponent
  - Mantissa (argument)
- Value of the number system base is usually not saved explicitly

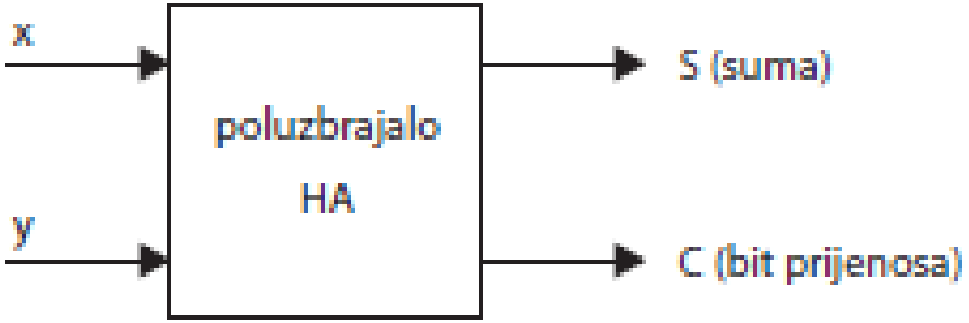
Summing two binary numbers

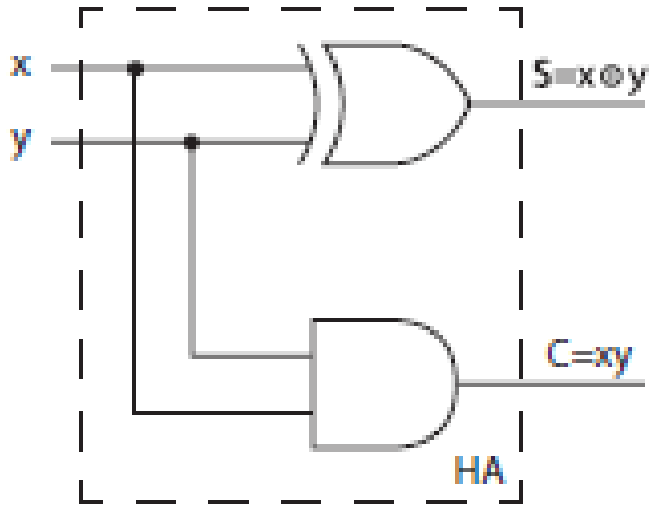
- A circuit that can do this - combination circuit
- We usually call that circuit HA (Half Adder)
- Based on summation rules, we can make a truth table and create this circuit:

Ulaz x	Ulaz y	Izlaz S (suma)	Bit prijenosa C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



HA



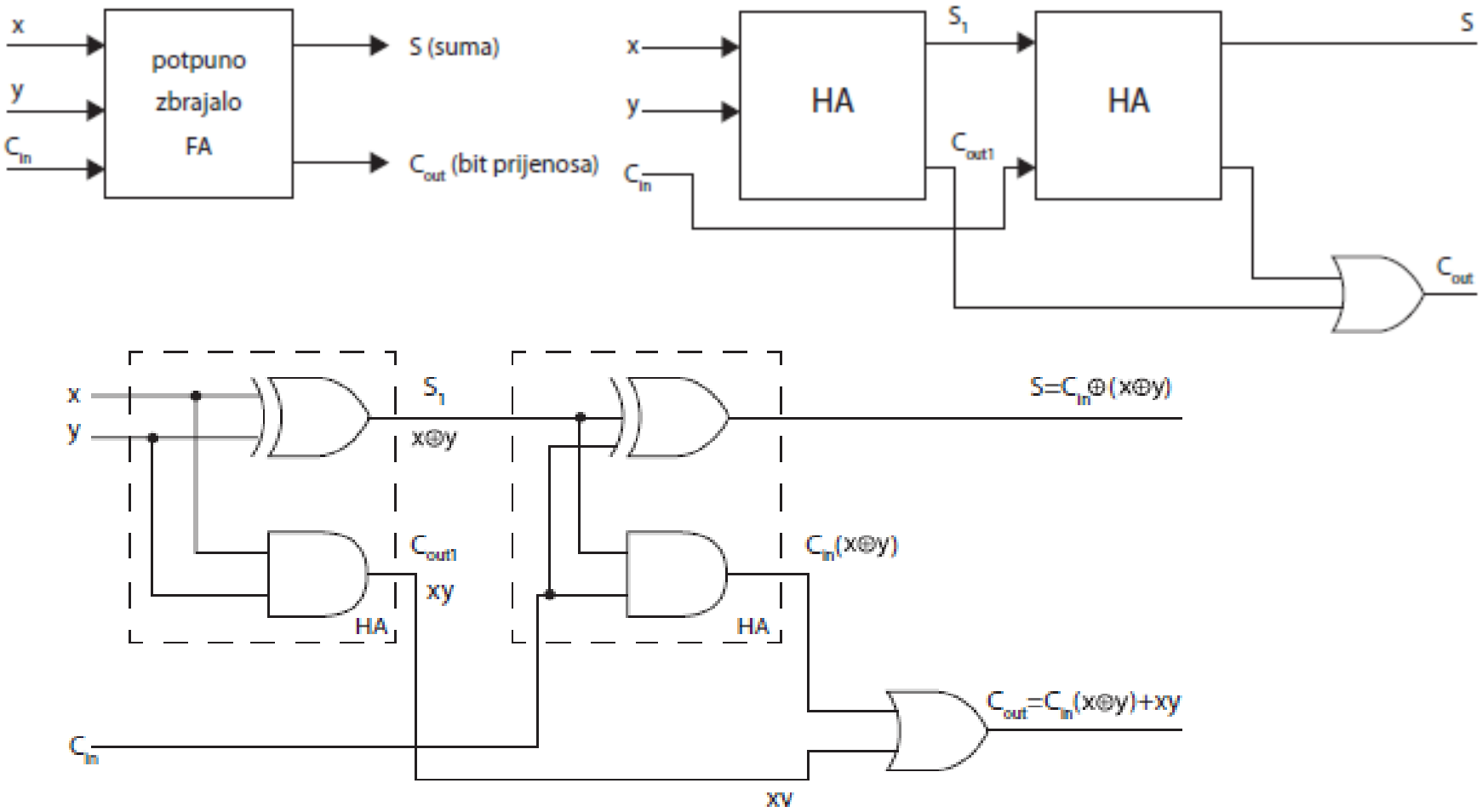


c)

FA (Full Adder) truth table

Ulaz x	Ulaz y	Ulaz C_{in}	Izlaz S (suma)	Izlaz C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

FA circuit

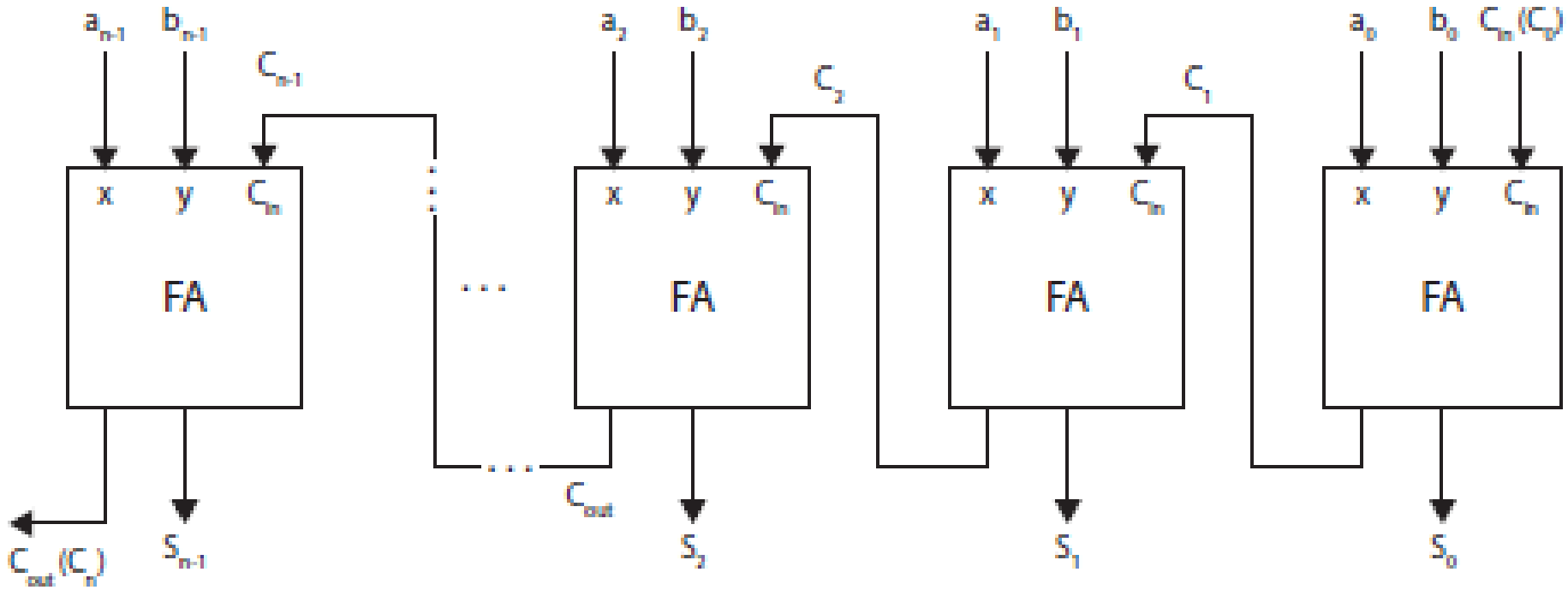

x
y
C_{in}
potpuno
zbrajalo
FA
S (suma)
C_{out} (bit prijenosa)
x
y
C_{in}
HA
S_1
C_{out1}
HA
S
C_{out}
x
y
S_1
x⊕y
C_{out1}
xy
HA
HA
C_{in}(x⊕y)
$S=C_{in}⊕(x⊕y)$
$C_{out}=C_{in}(x⊕y)+xy$
C_{in}
xy

Full Adder

- two-level combination circuit that sums three input bits
  - x and y - operand bits (Ai, Bi)
  - Ci - Carry bit from the previous stage
- two outputs - Fi (single-bit result) and Ci+1 (carry bit to the next stage

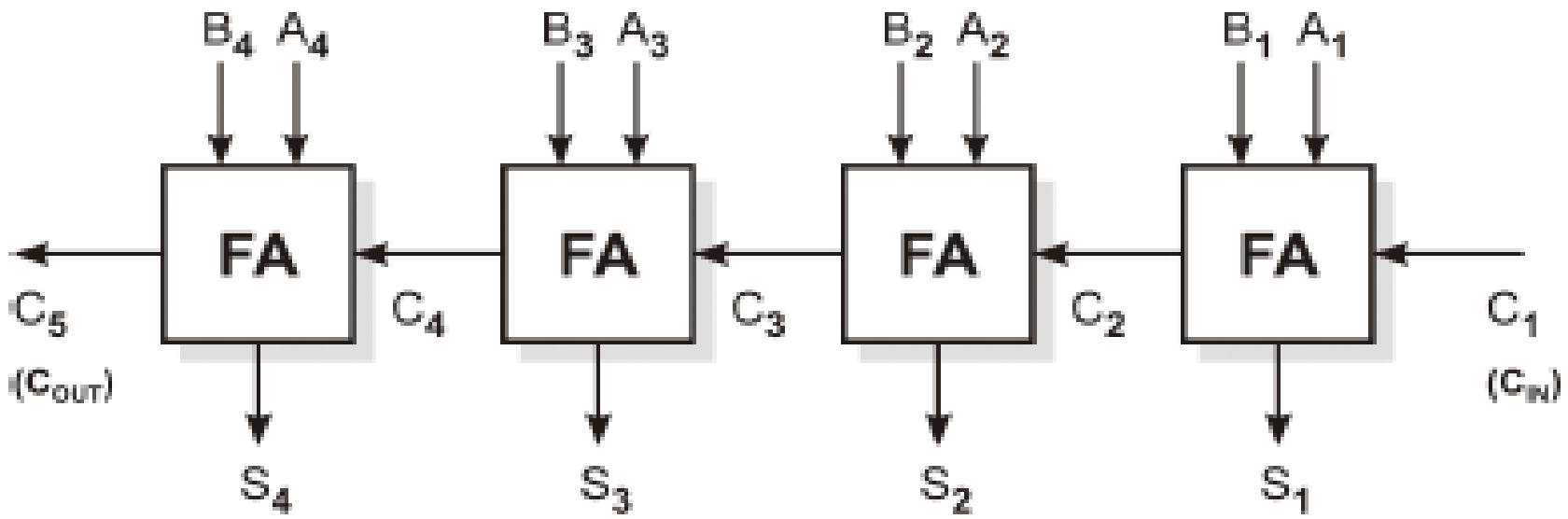
Parallel adder

- adds n-bit word A and n-bit word B in one step



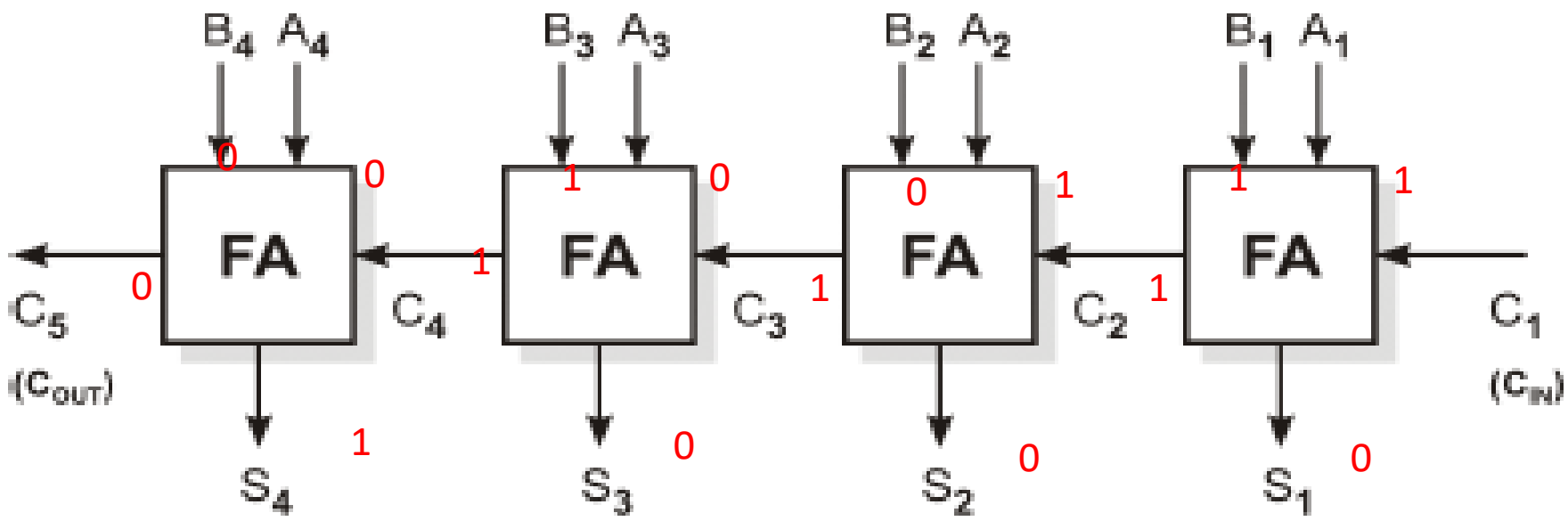
Parallel adder

- The term used (parallel adder) itself suggests that we can do n adds almost simultaneously

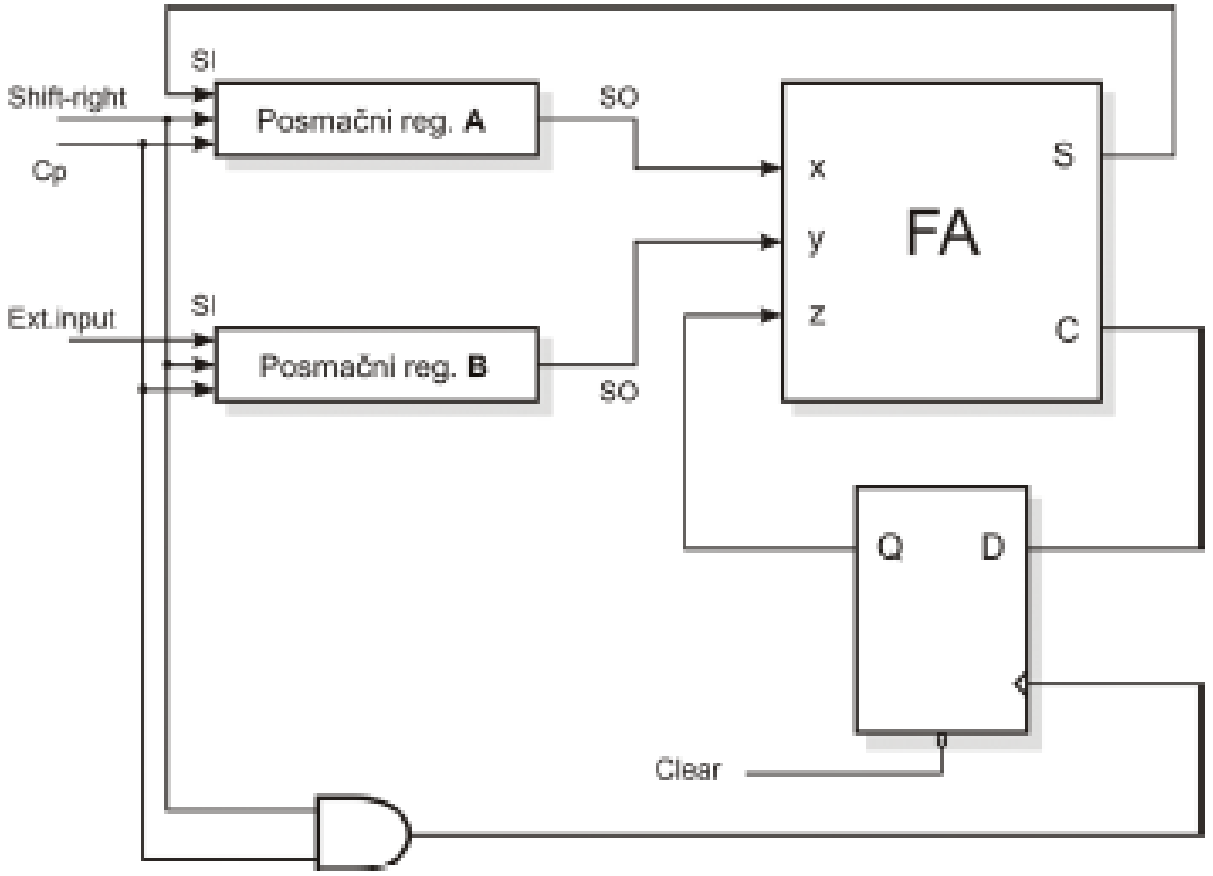


Parallel Adder example

- 5+3 (0101 + 0011)



Serial adder



Serial adder

- Shift registers A and B have n-bit operands
- Bits use shift to the right to be transferred to one stage of full adder circuit
- Result of bit-adding process is stored and shifted in shift register A
- D bistable circuit (flip-flop) is here to store carry bit from the previous bit-add operation

Subtracting two binary numbers

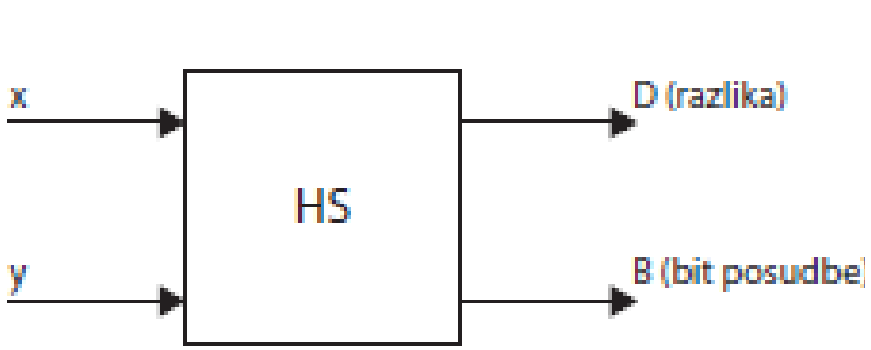
- D equals "difference", or the result of subtraction process
- B equals "borrow"
- Based on basic subtraction rules, this is the truth table:

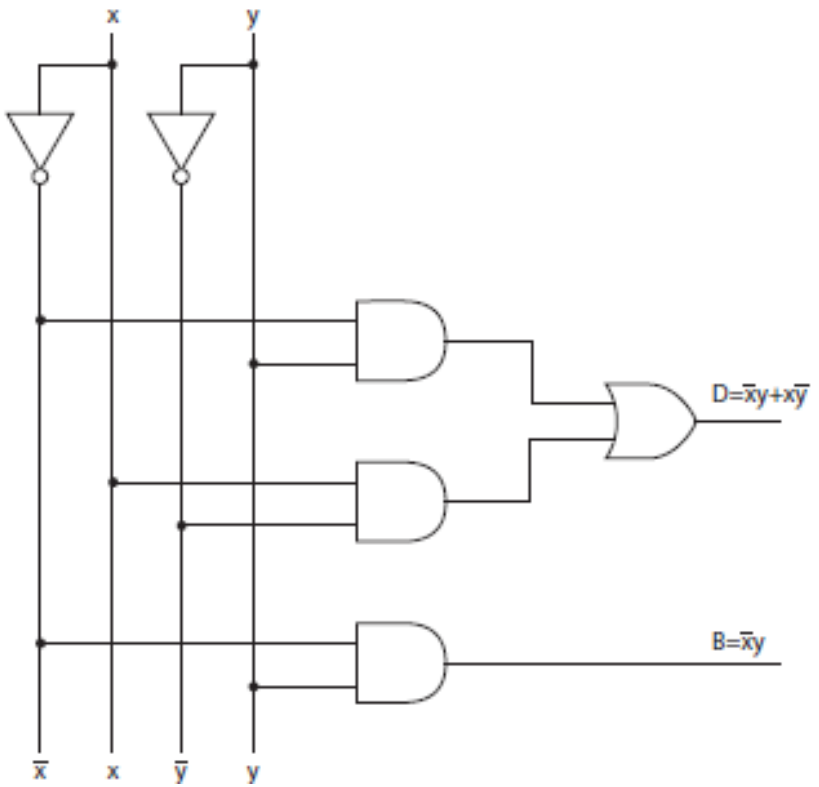
Ulaz x	Ulaz y	Izlaz D (razlika)	Bit posudbe B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Subtracting two binary numbers

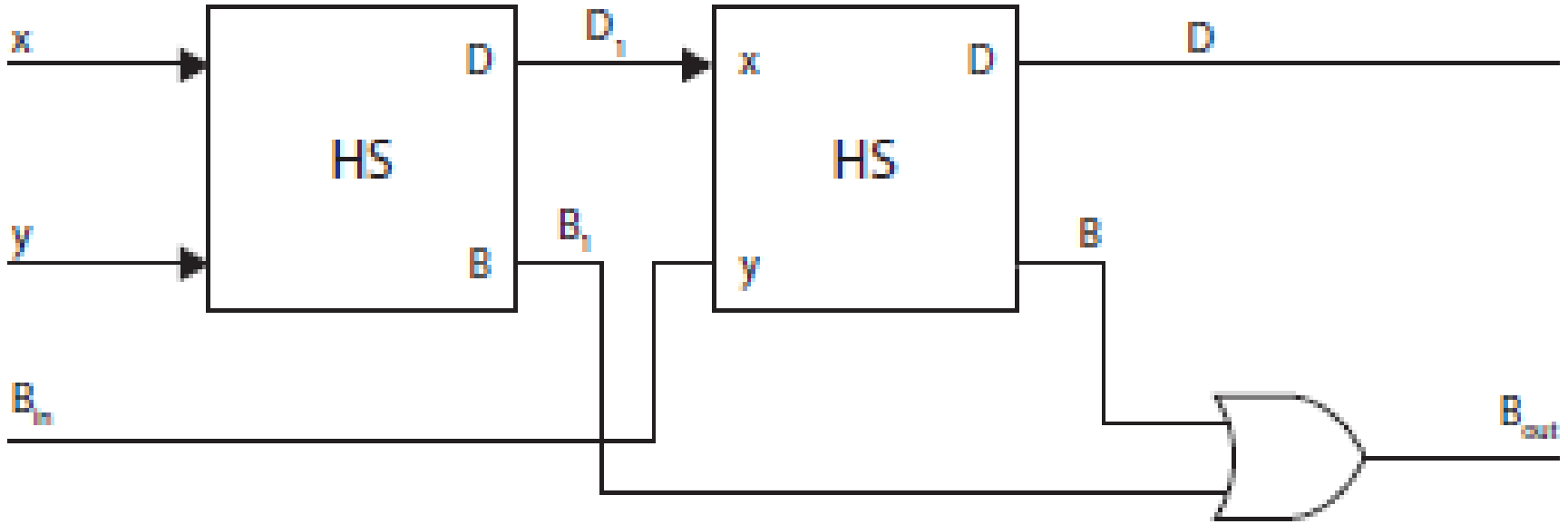
- We can use a circuit called half-subtractor for this (HS)
- Similar to the full-adder, we can use two half-subtractors to create a full subtractor circuit (FS)

Half-subtractor circuit



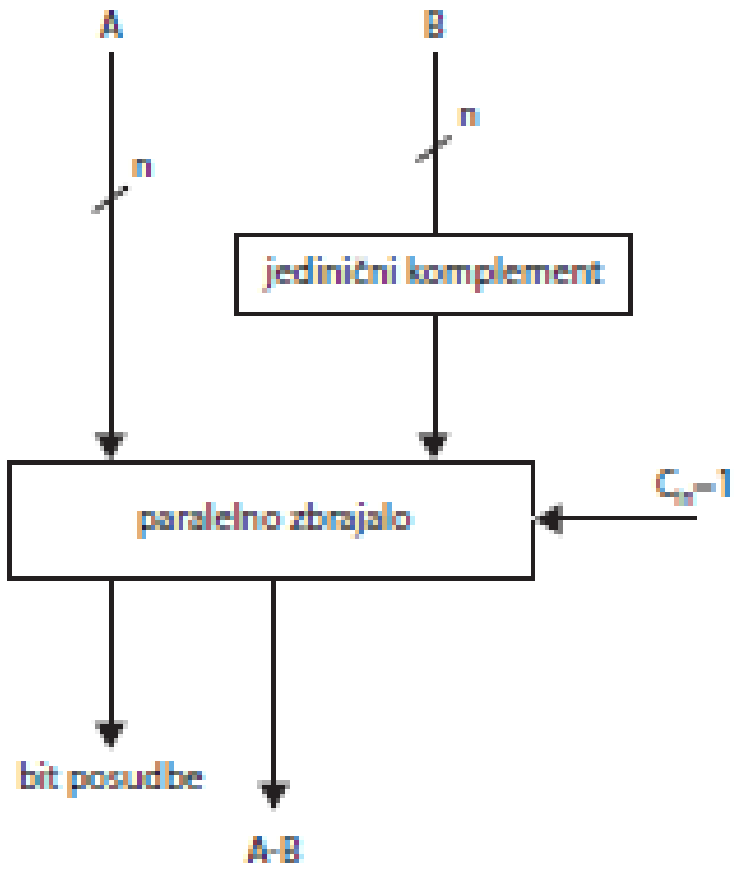


Full subtractor circuit, created out of two half-subtractor circuits



Subtracting by using binary complement

- Subtracting can be done by using summation of second/full complement of subtrahend
- we can get second B operand complement by using single complement and add Cin=1 to it

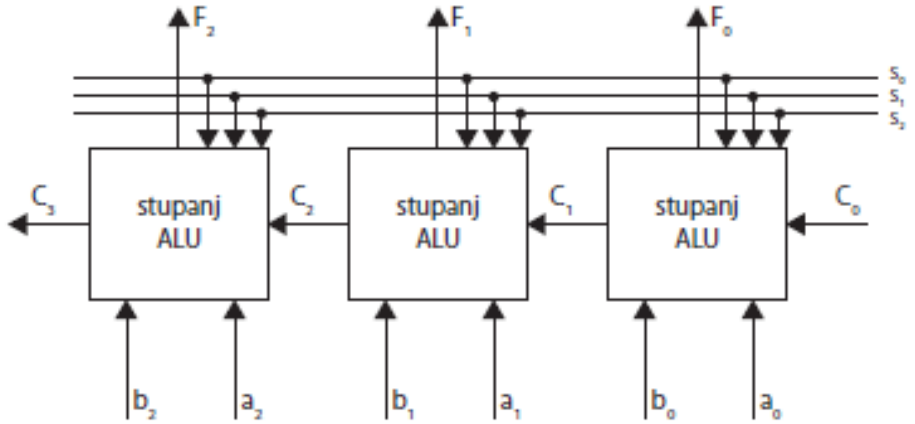


Forming a simple ALU

- ALU is a multi-functional digital circuit
- It can do arithmetic and logical operations
- Often used to calculate effective memory address of operands or results

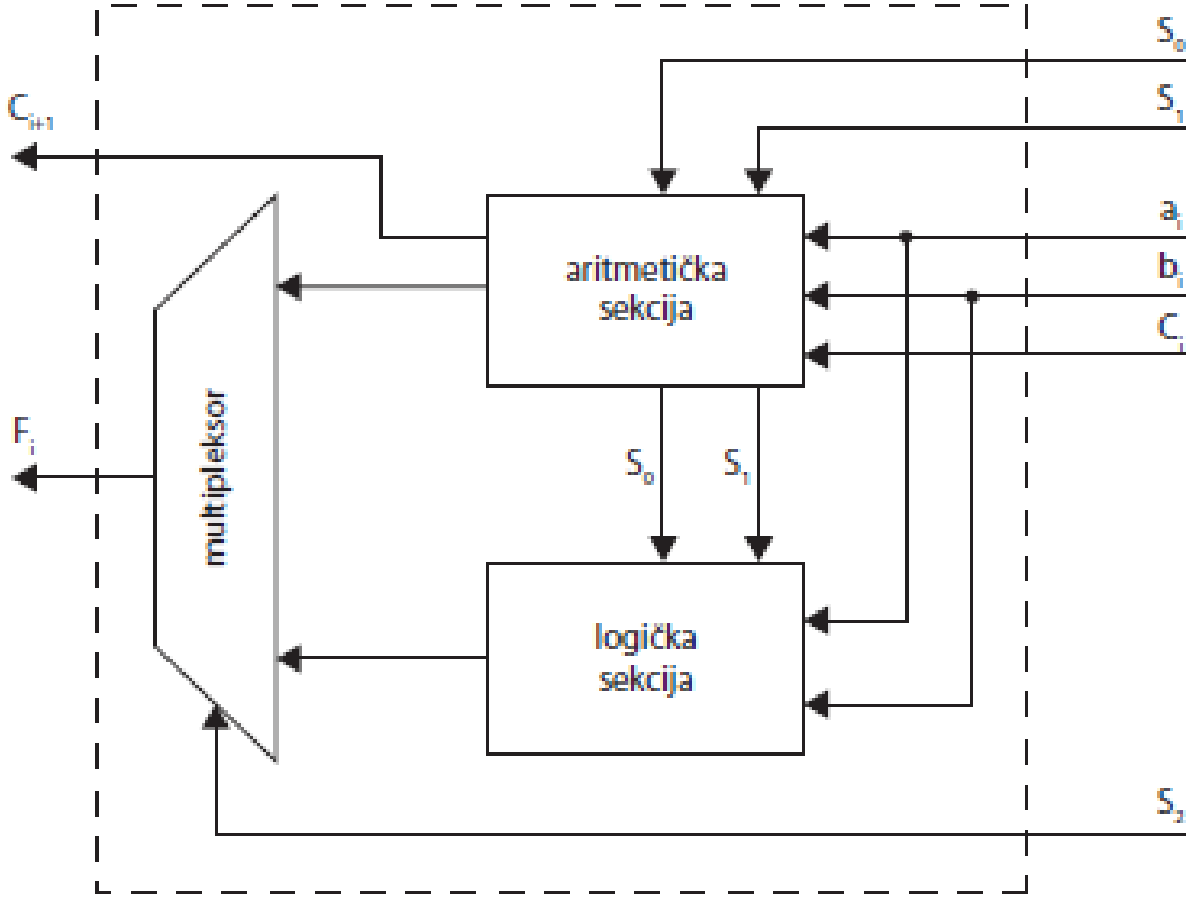
Simple ALU

- Circuit that has a very sequential structure

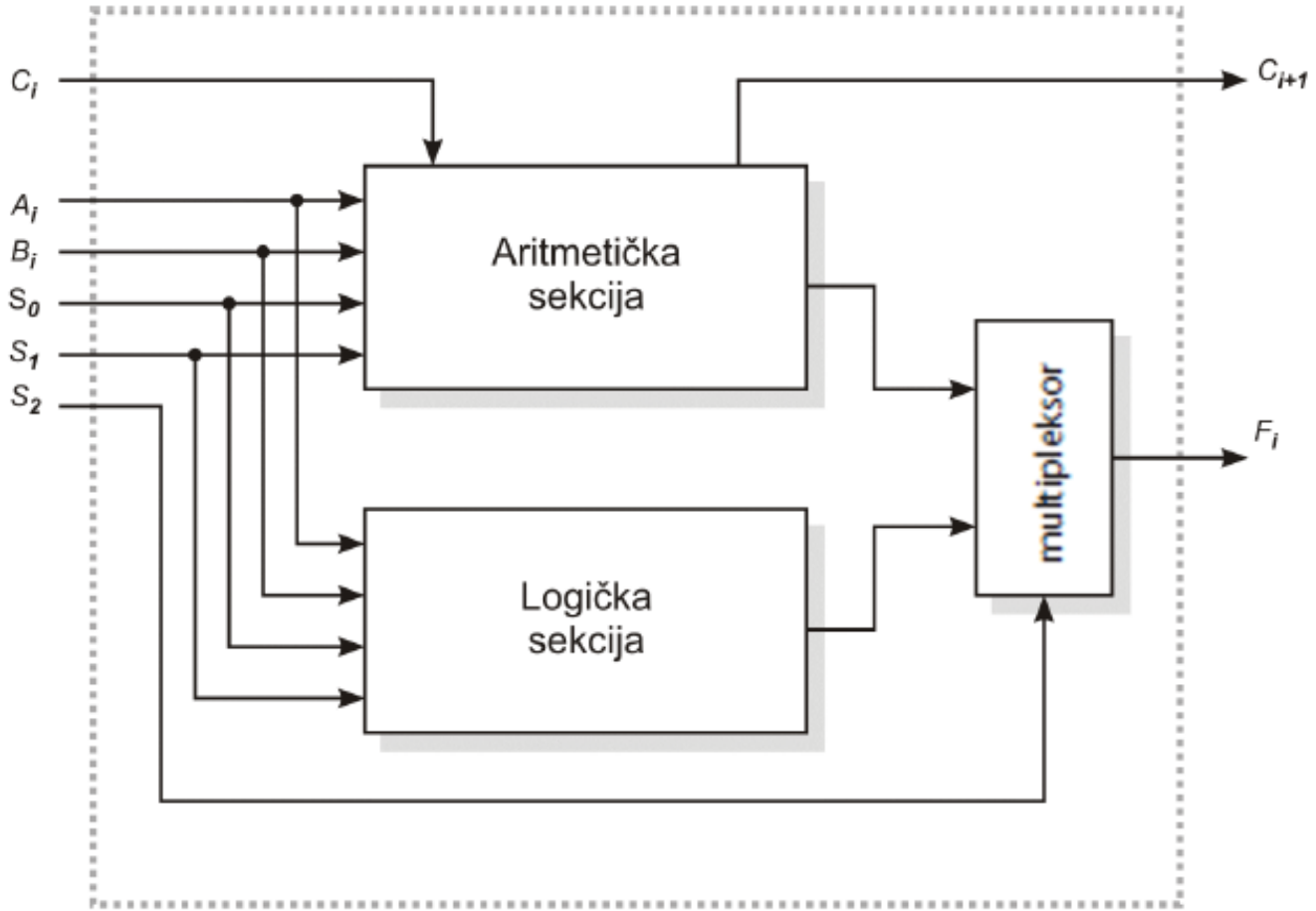


- Multiple cascaded stages
- Every stage does either A or L operations on a single-bit operand
- Usually uses n-bit (integer) operands, with n such stages
- Function-wise, they can be represented as having:
  - arithmetic section
  - logic section

ALU, i-th stage



ALU, i-th stage (arhithmetic and logic sections)



ALU inputs and outputs

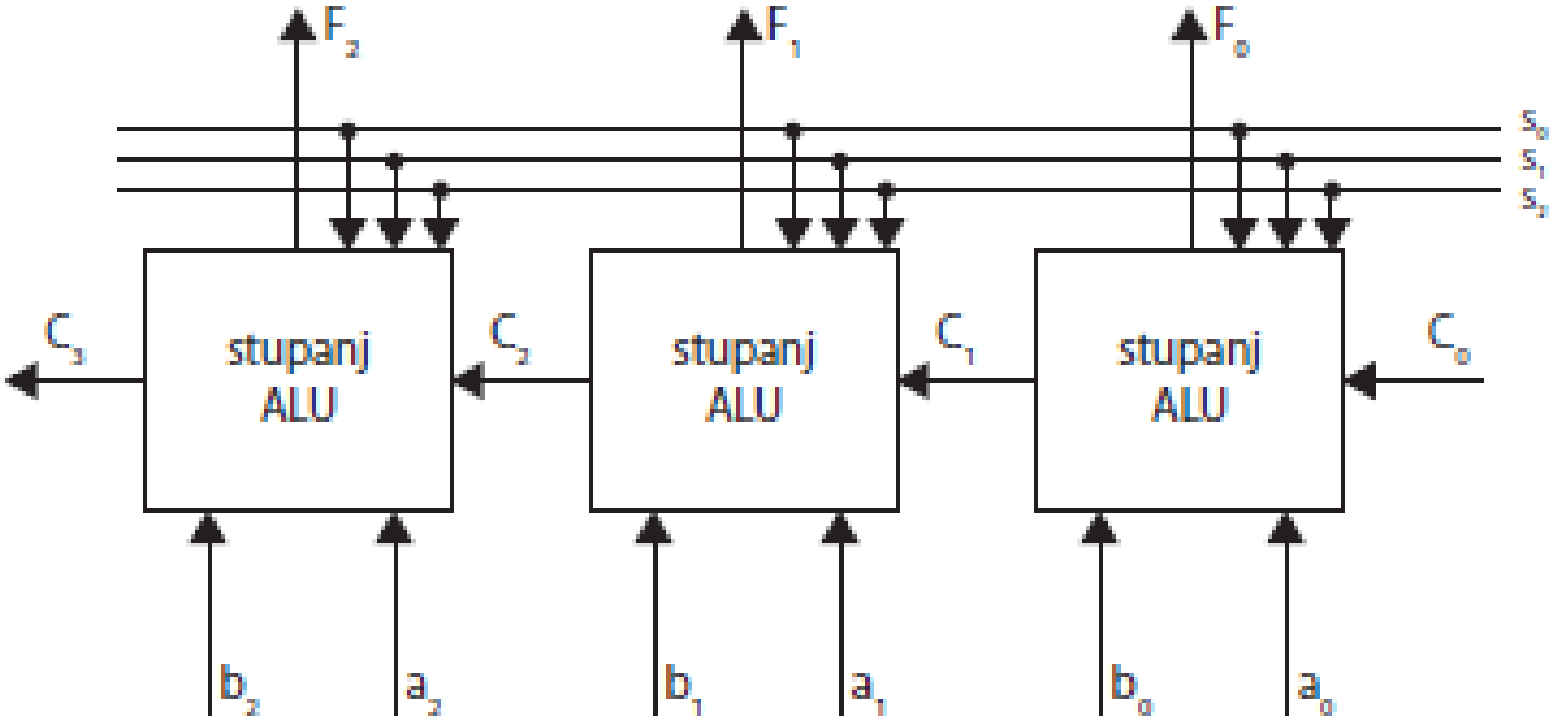
- a_i and b_i are operand inputs
- C_i is a carry bit from the previous stage
- F_i is a single-bit result of A or L operation
- C_{i+1} is a carry bit to the next stage
- Control inputs S_o, S_1 i S_2 are used for different A and L operation selection

Arithmetic section

- Usually done as a full adder (FA)
- We link n stages of them, therefor getting a parallel adder that can sum two words with n-bit length
- Subtraction - full operand complement method
- If we want to do multiple operations, we have to add another circuit, that can prepare B operands
- For example, we want a simple ALU (based on input combinations) to be able to do eight different operations

Connecting stages into a cascade

- made by using carry bits Ci i Ci+1



Forming a simple ALU

- Spliting the ALU stages to A and L is just a functional split
- When forming a simple ALU, we use the following approach:
  - first, we choose A circuits, independent of the L section
  - then, we select L operations that we can do by using the A section
  - we iterate circuit changes so that we create an ALU that can do all L operations that we want

Forming a simple ALU

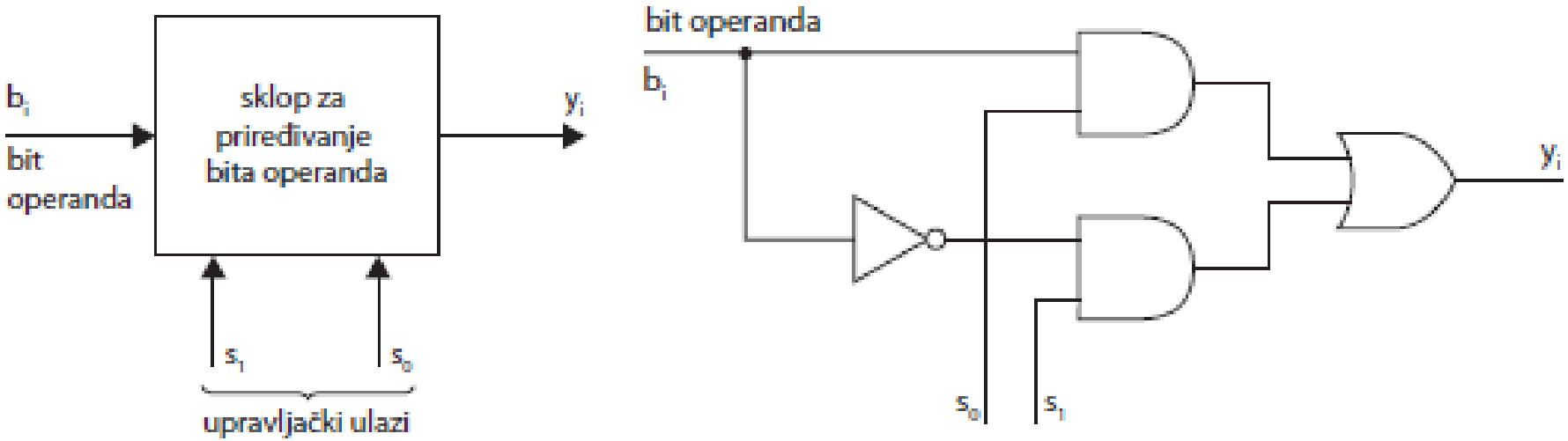
Ulazi			Izlaz F
X	Y	C_{in}	
X	Y	0	Zbrajanje: $F = X + Y$
X	Y	1	Zbrajanje s bitom prijenosa: $F = X + Y + 1$
X	Jedinični komplement: $\overline{Y}$	0	$F = X + \overline{Y}$
X	$\overline{Y}$	1	Oduzimanje: $F = X + \overline{Y} + 1$
X	0	0	Prijenos: $F = X$
X	0	1	Inkrementiranje: $F = X + 1$
X	Sve jedinice: 111...111	0	Dekrementiranje: $F = X - 1$
X	Sve jedinice: 111...111	1	Prijenos $F = X$

Forming a simple ALU

- by adjusting operands at input Y and selecting correct values for the carry bit into the lowest stage of ALU Cin, we can do:
  - summing, subtracting
  - incrementing and decrementing X
  - summing operand X and single Y complement
  - carrying operand X from input to output

Preparing the bit operand at Y input

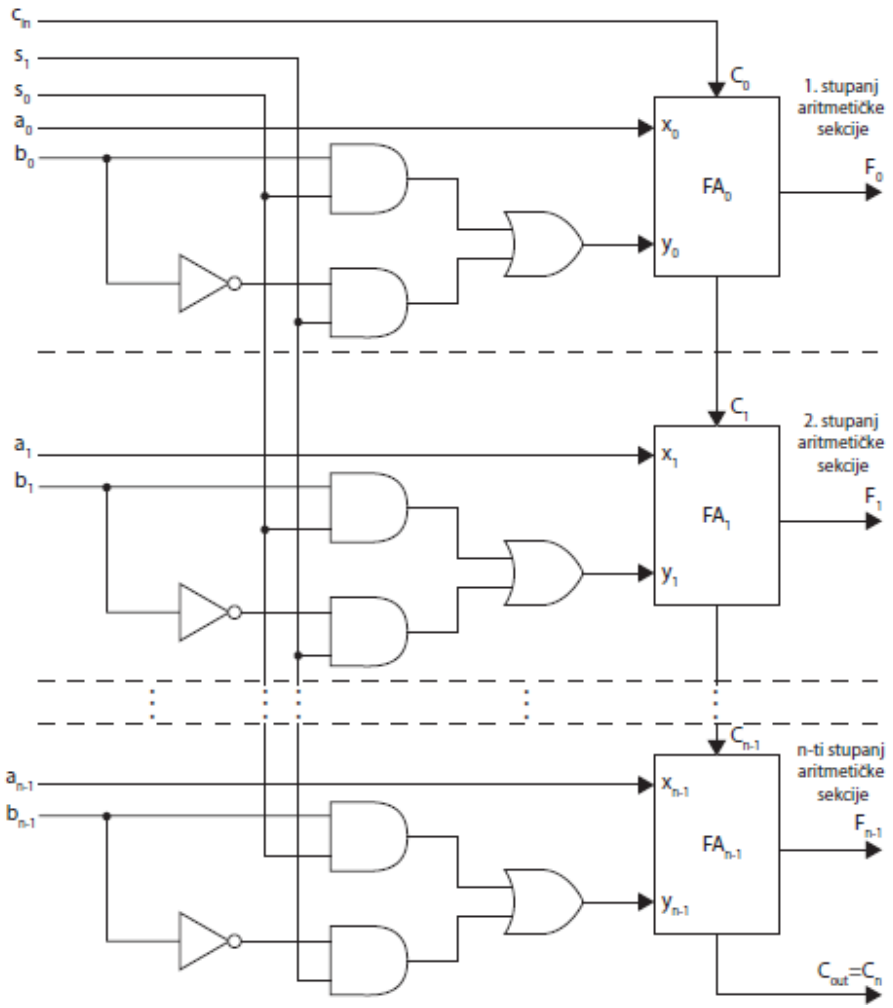
- In order to create a parallel adder that is capable of executing eight operations at the same time we need additional logic circuits on the Y input *to prepare the bit operand*



Preparing of bit operand at Y input

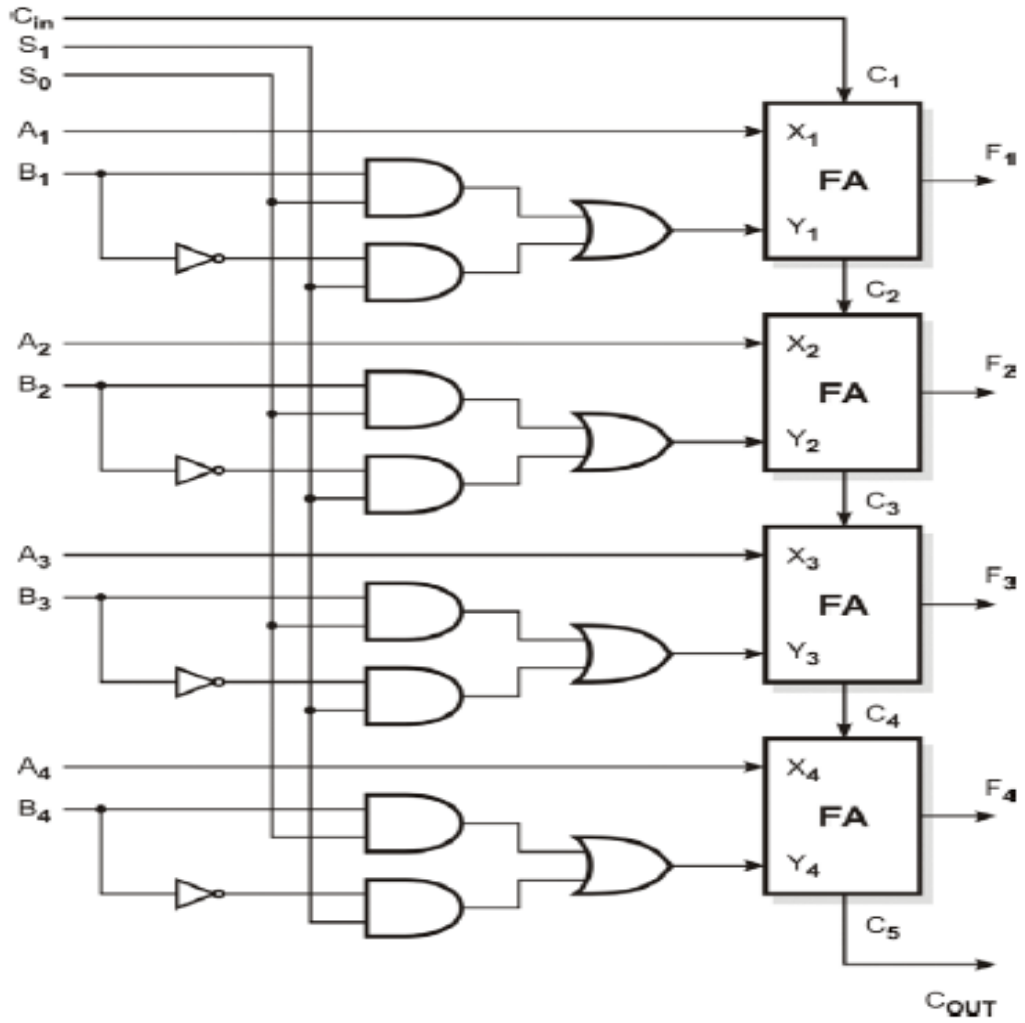
S_1	S_0	Ulaz: bit operanda	Izlaz iz sklopa za pridruživanje y_i
0	0	b_i	0
0	1	b_i	b_i
1	0	b_i	$\overline{b_i}$
1	1	b_i	1

ALU arithmetic section



ALU cascading

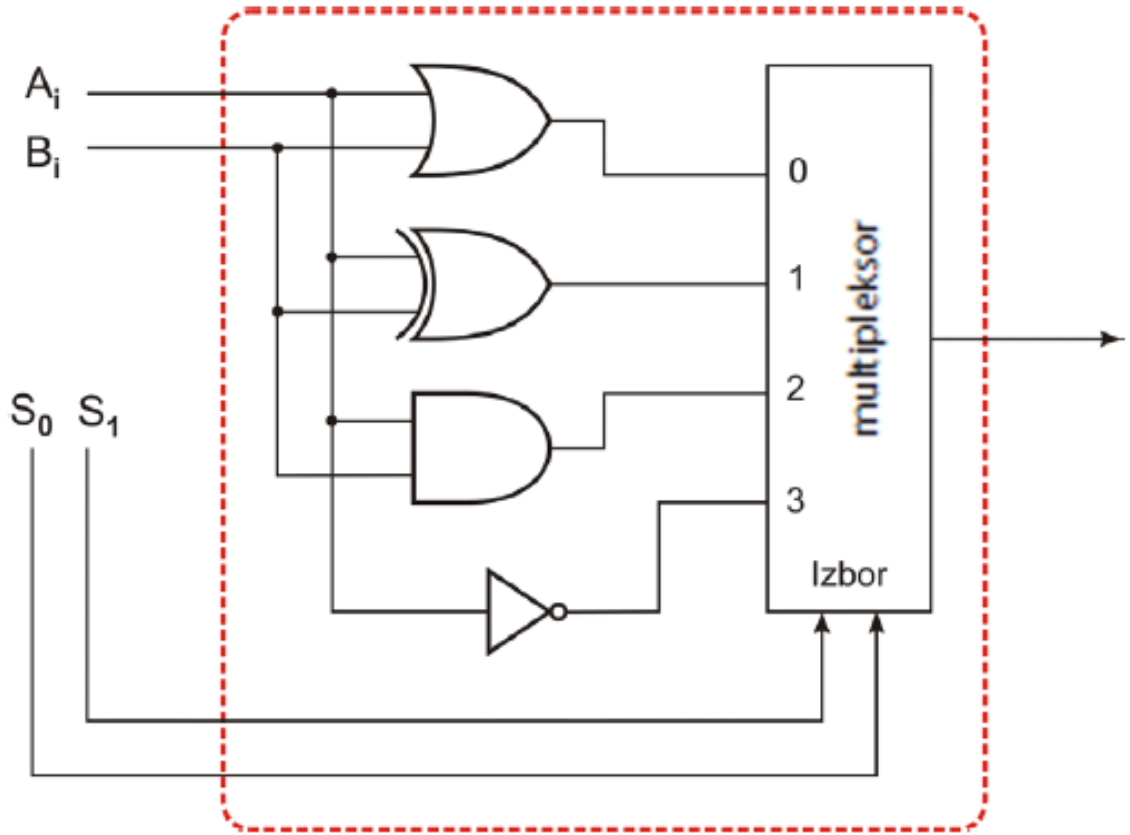
- Arithmetic sections are connected to a cascade
- Last stage has output called Cout that is written to the status register



Logic section

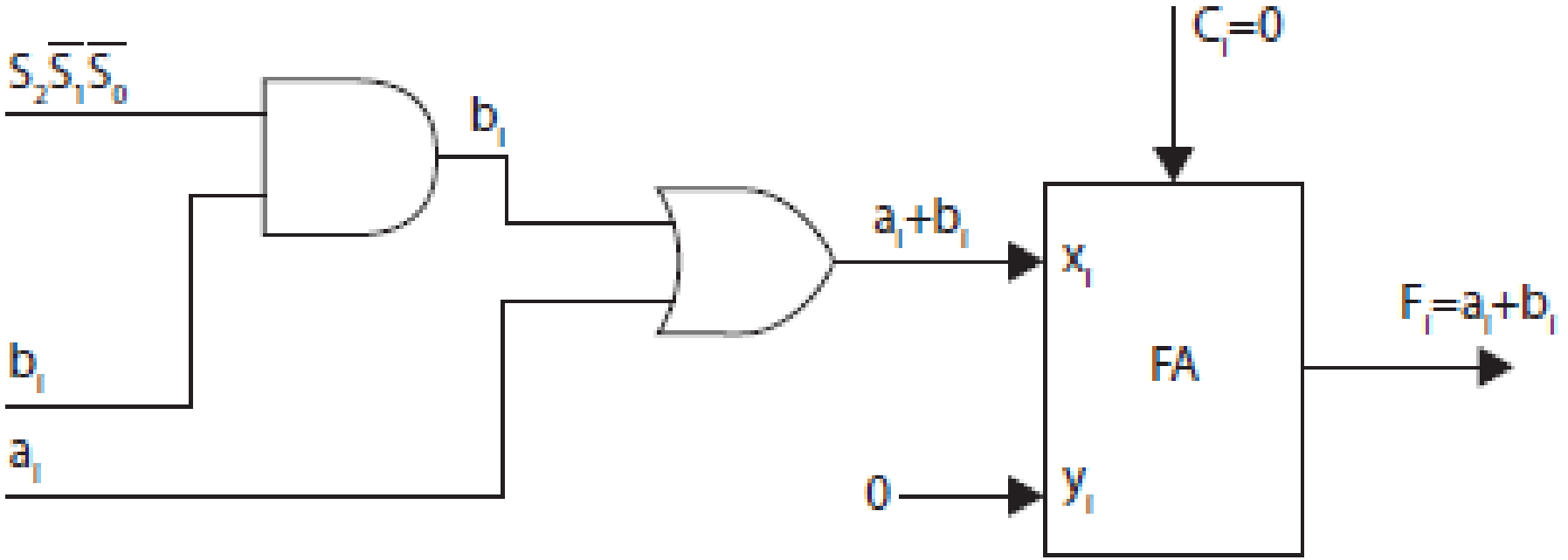
- L operations work with separate operand bits and every operand bit is treated as a logical, binary variable
- It needs to support four basic logical operations - AND, NOT, OR and XOR
- arithmetic and logic section of any given stage are connected to a single section
- Control input S2 is used to select if the operation being done is logical or arithmetic one

L section



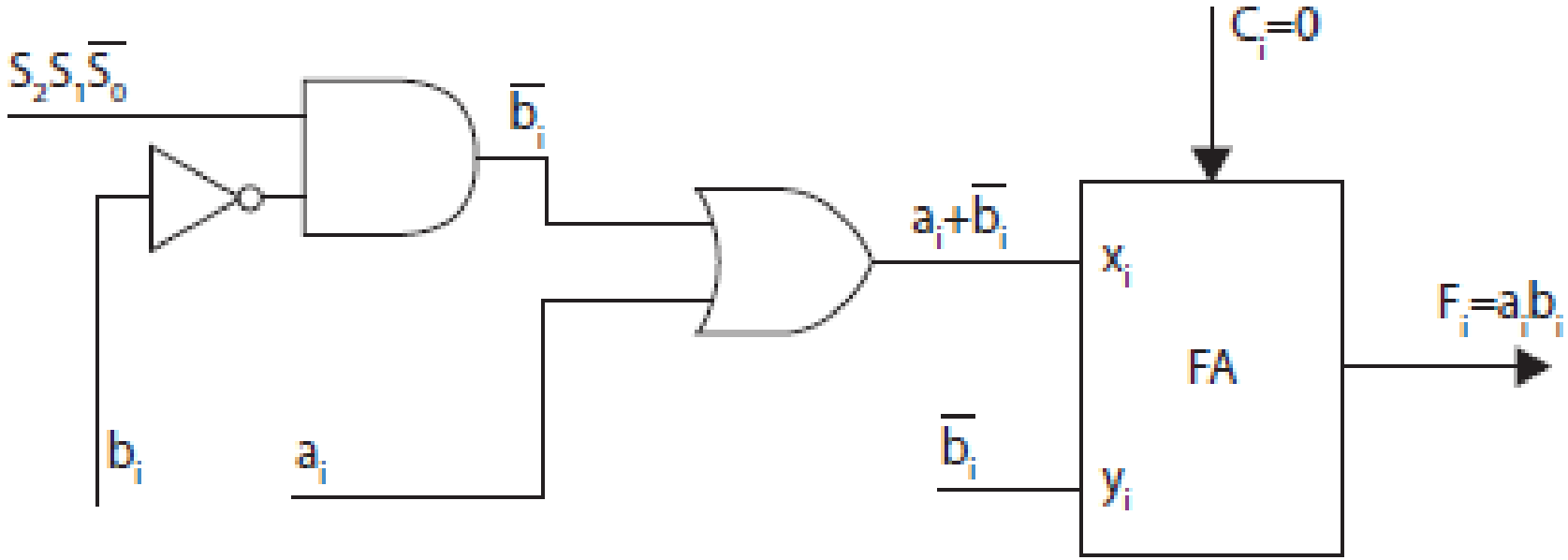
L operation - OR

- Circuit needed to do OR

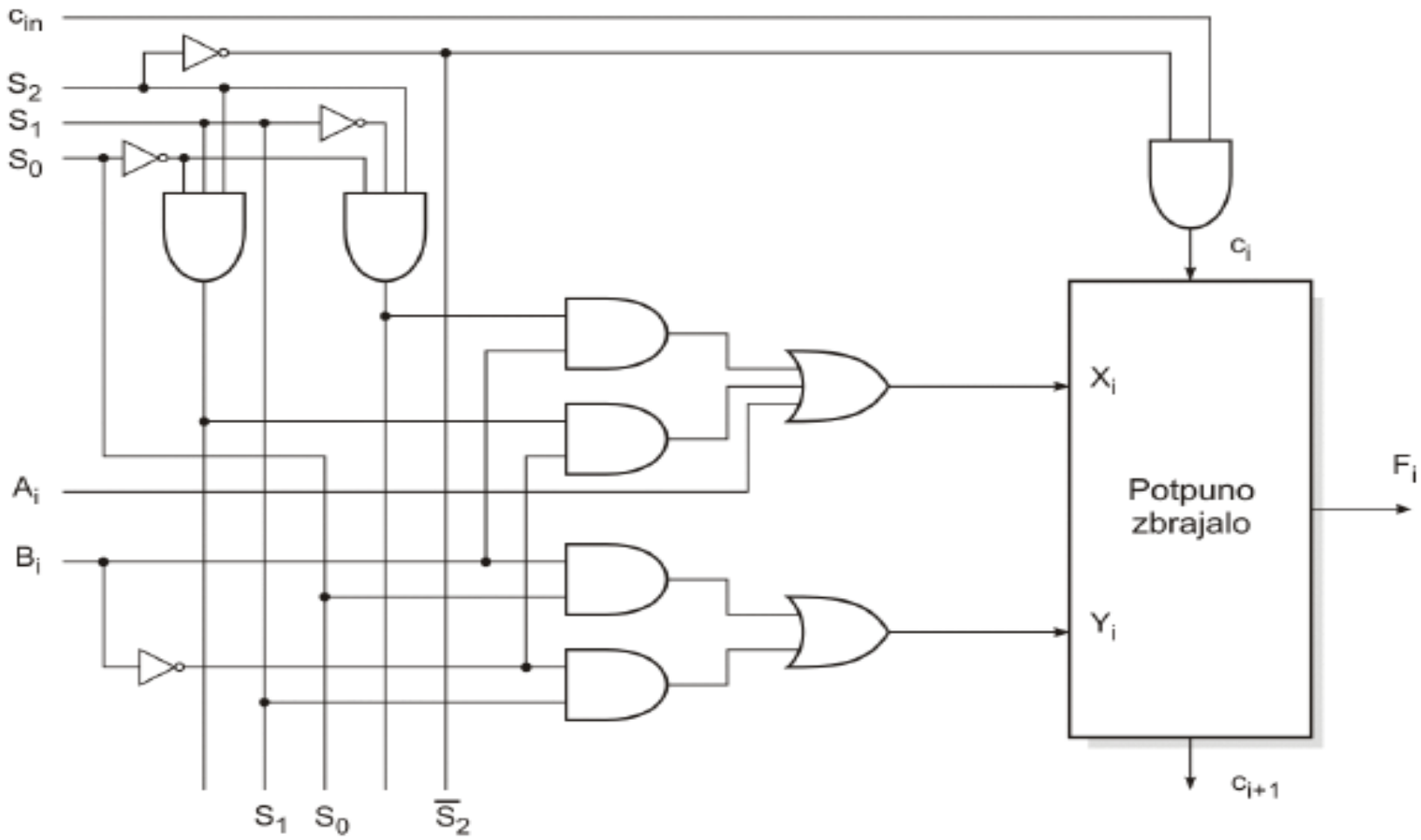


Logic operation - AND

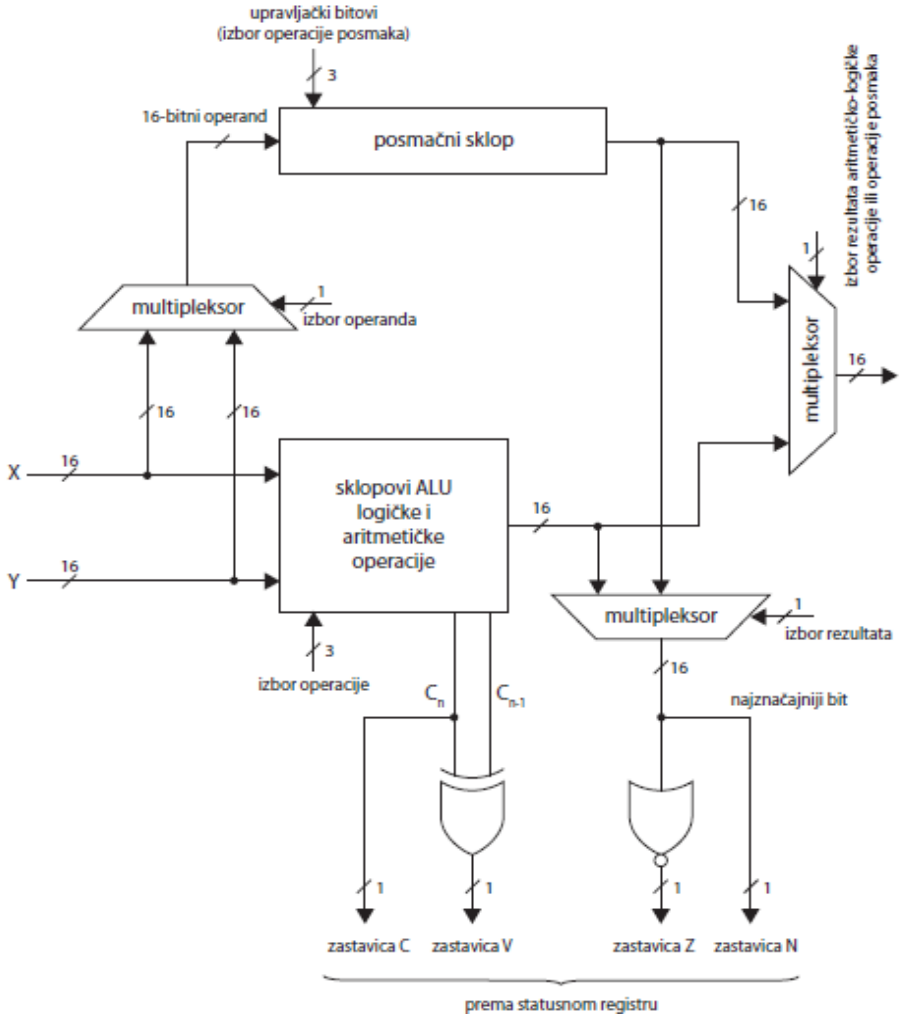
- Changing the i-th stage of simple ALU to get logical AND



Final ALU


c_{in}
S_2
S_1
S_0
c_i
X_i
A_i
Potpuno
zbrajalo
F_i
B_i
Y_i
S_1
S_0
$\overline{S_2}$
c_{i+1}

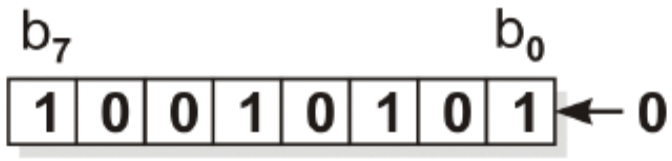
Simple, multi-functional ALU

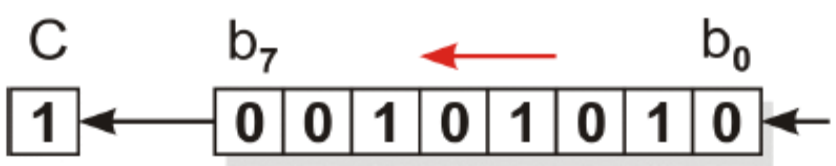


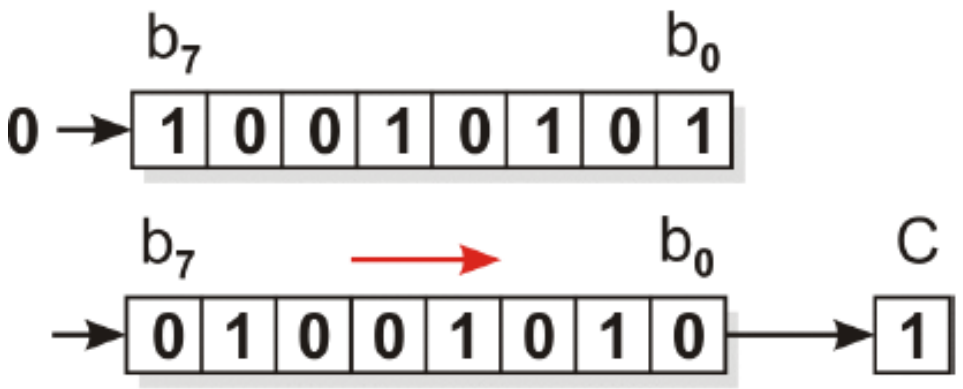
Shift circuit

- Basic building block for an ALU
- It's located at ALU output, it connects it to a bus
- Transfers the A/L operation result to a bus:
  - directly, without shifting
  - by shifting left or right
- These circuits can be made as two-way shift registers or as combination circuits

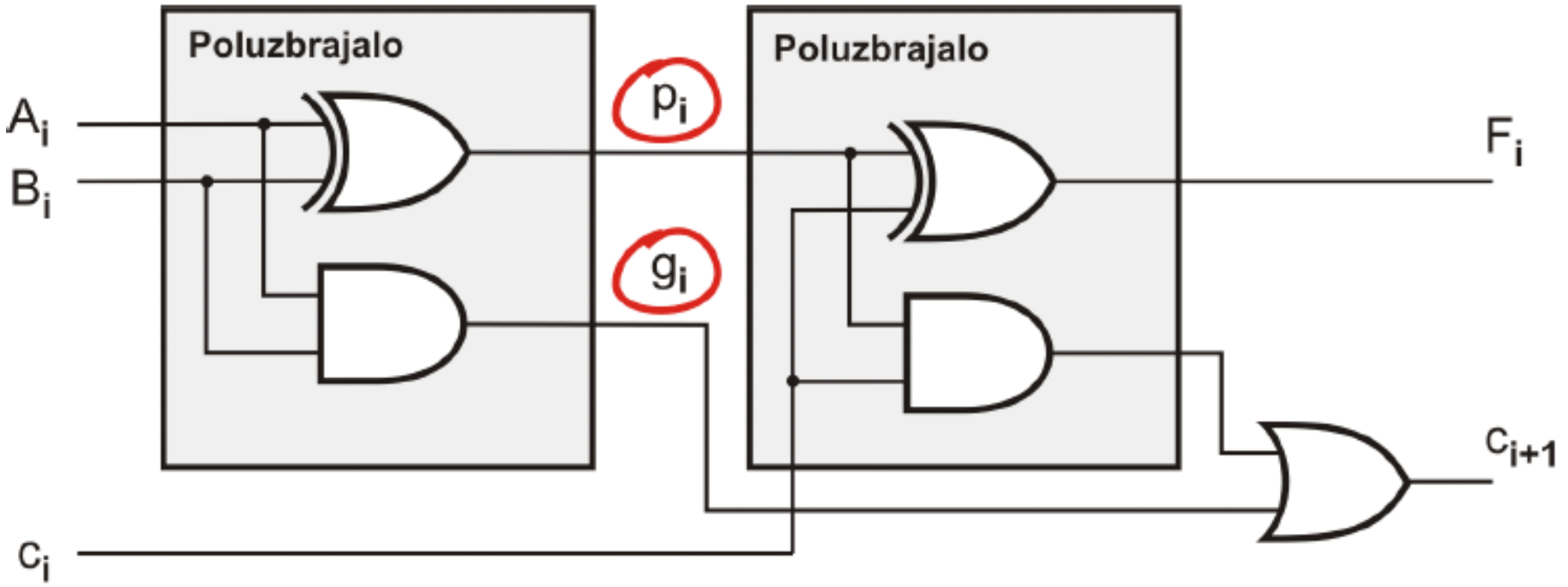
Shift circuits







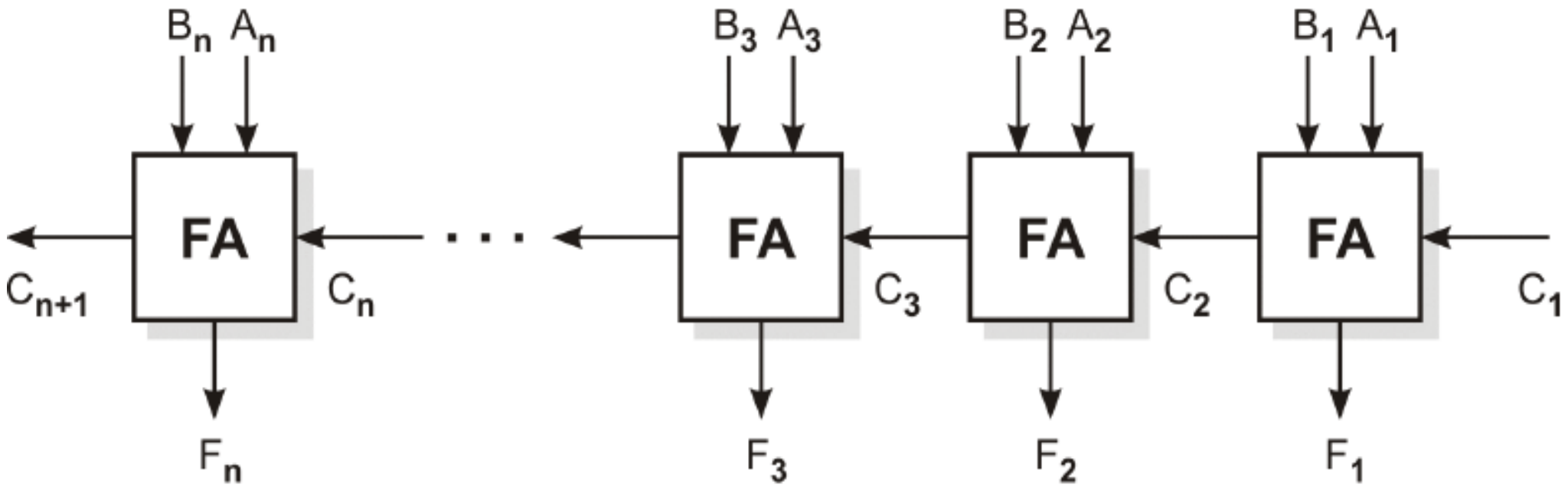
Circuit that predicts carry bit



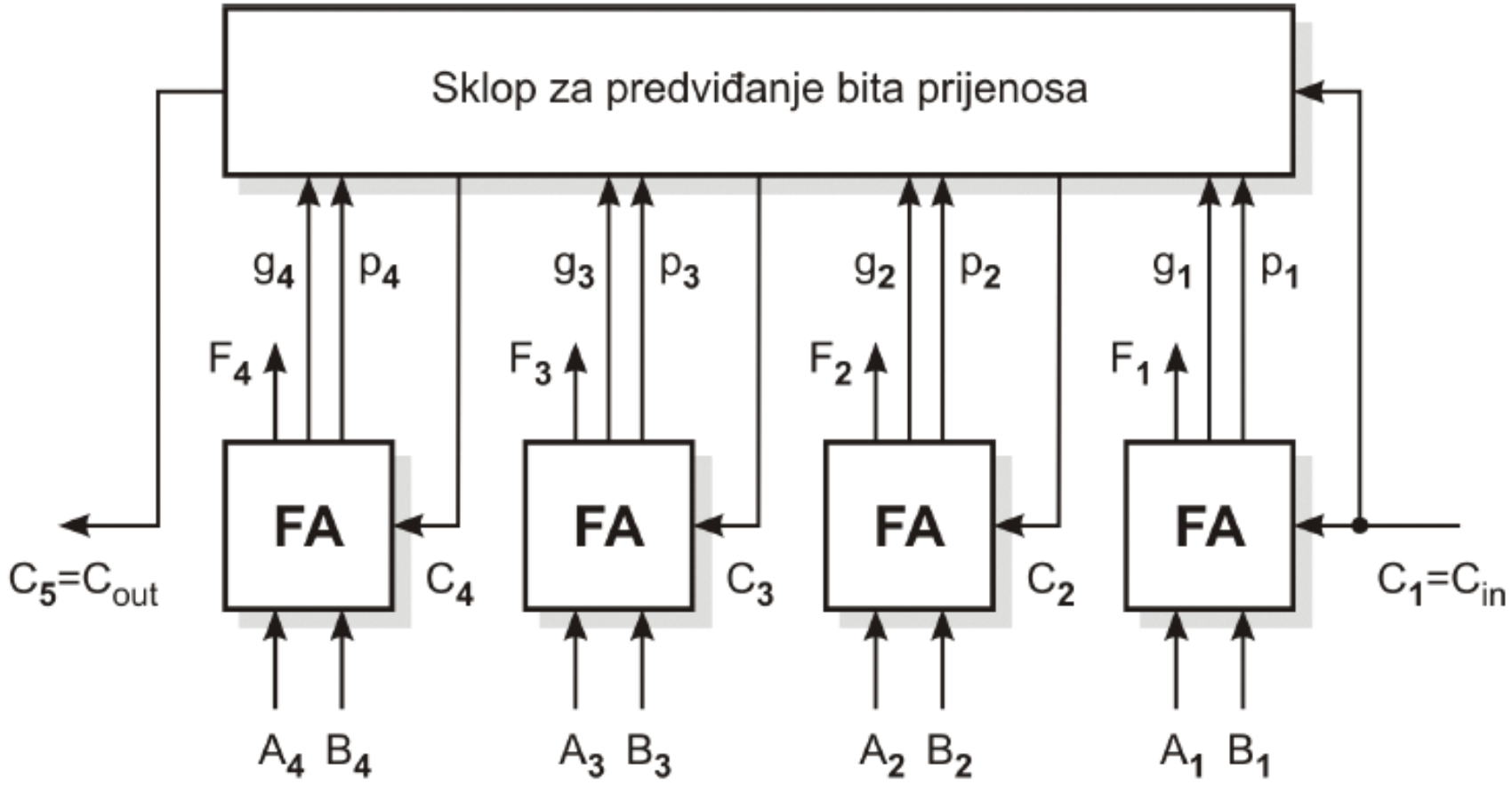
$g_i = A_i B_i$ signal generiranja bita prijenosa

$p_i = A_i \oplus B_i$ signal širenja (propagacije) bita prijenosa

Parallel, n-bit adder latency



Creating a fast adder



Basic multiplication algorithm

				a_3	a_2	a_1	a_0
							x
				b_3	b_2	b_1	b_0
Parcijalni produkt 0				a_3b_0	a_2b_0	a_1b_0	a_0b_0
Parcijalni produkt 1			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
Parcijalni produkt 2		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
Parcijalni produkt 3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			

Multiplication

23_{10} x
19_{10}

---

2 0 7

2 3

---

4 3 7

1 0 1 1 1 x
1 0 0 1 1

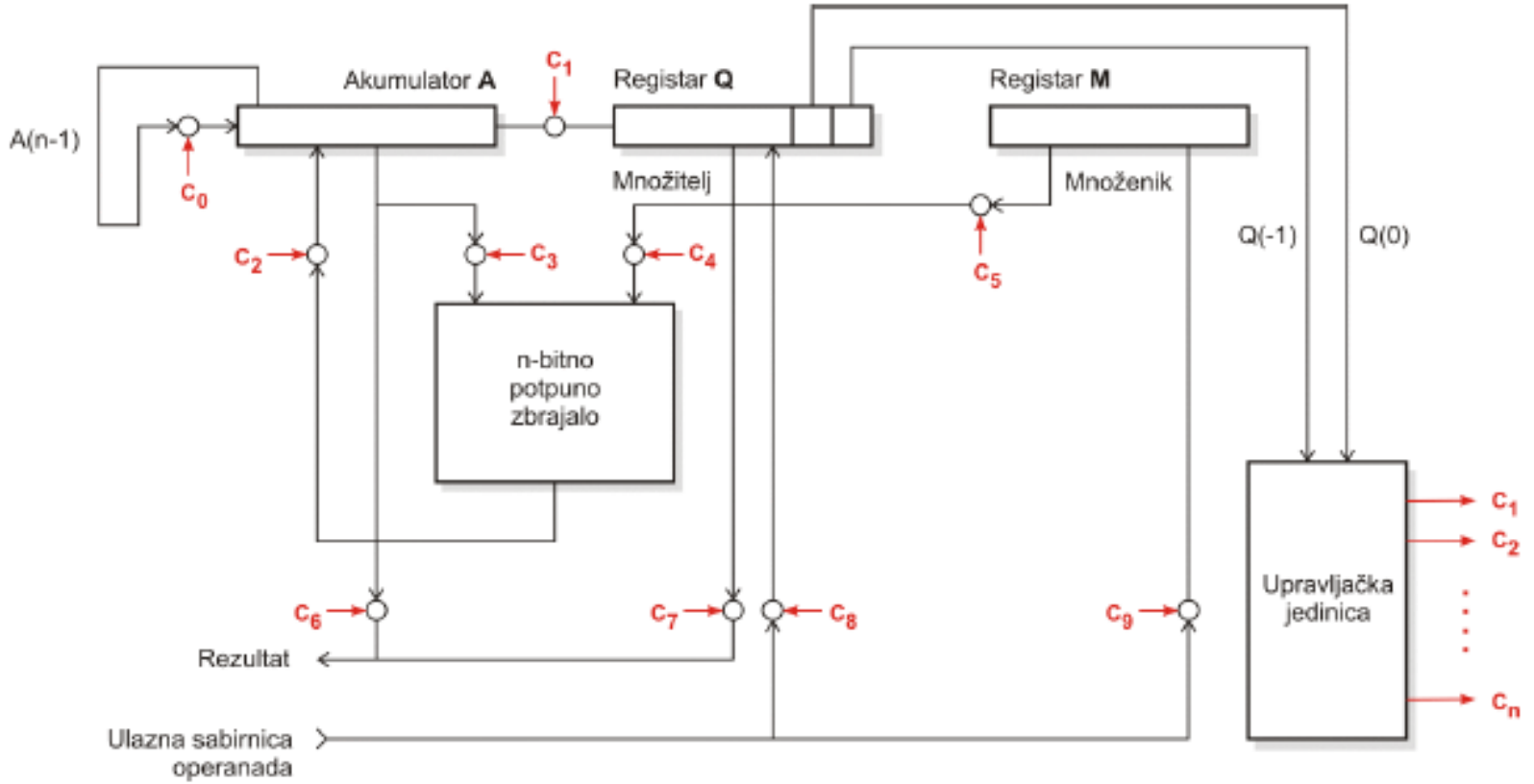
---

1 0 1 1 1
1 0 1 1 1
0 0 0 0 0
0 0 0 0 0
1 0 1 1 1

---

1 1 0 1 1 0 1 0 1

Multiplication block diagram



Thank you for your attention!