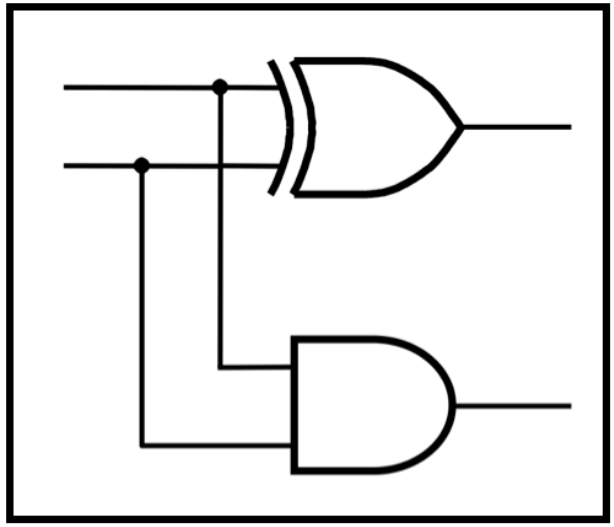


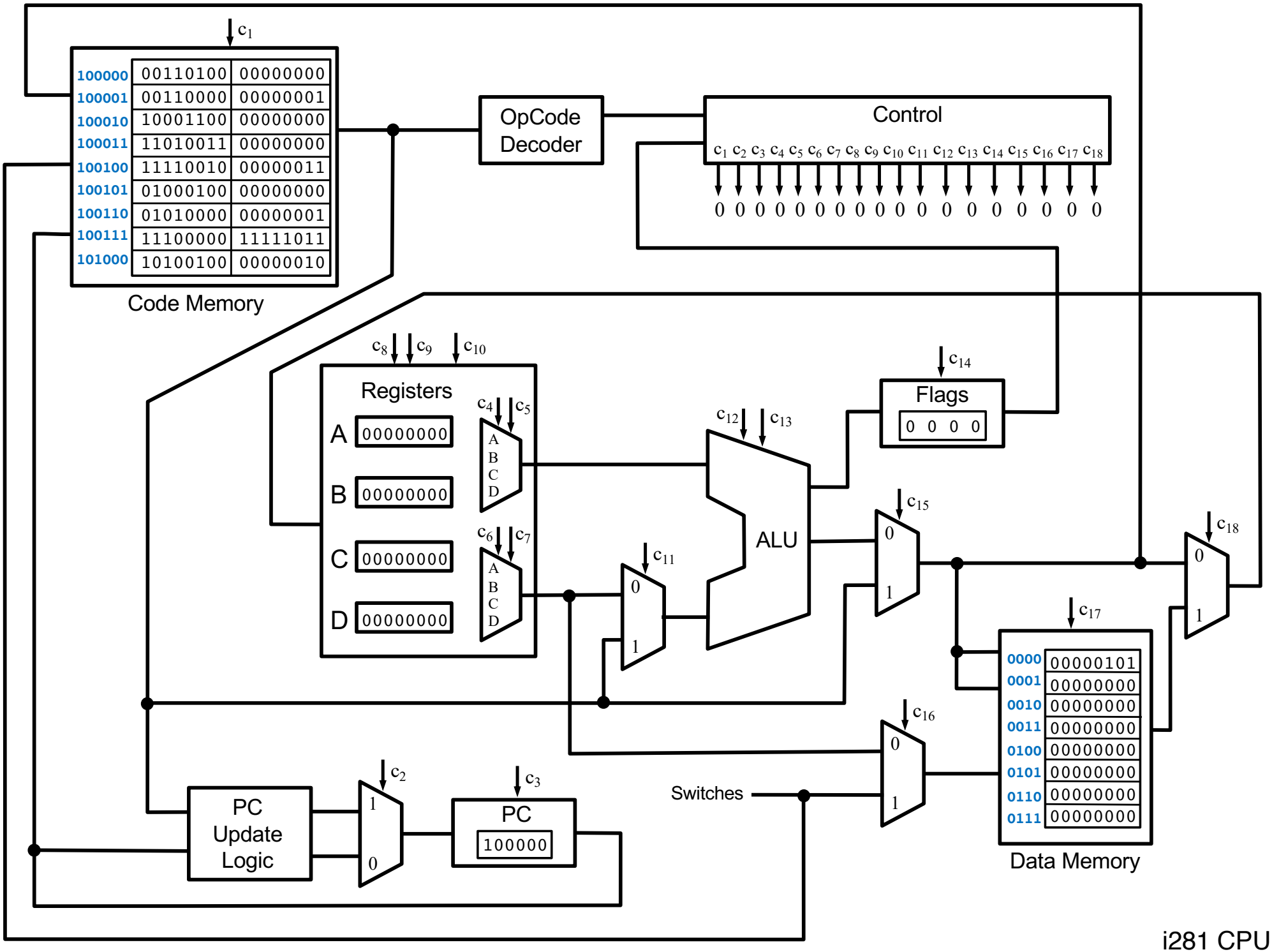
CprE 281: Digital Logic

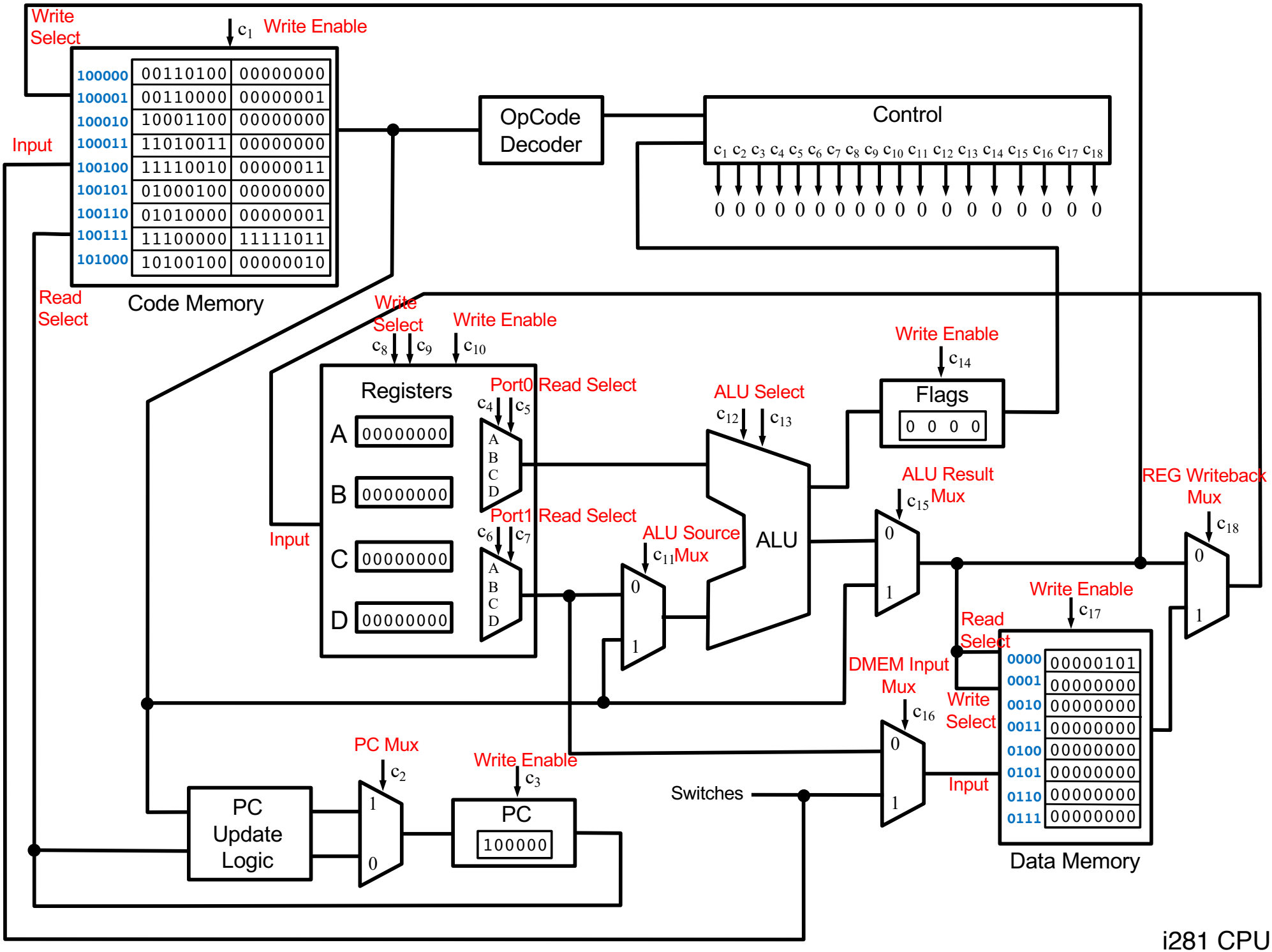
Instructor: Alexander Stoytchev

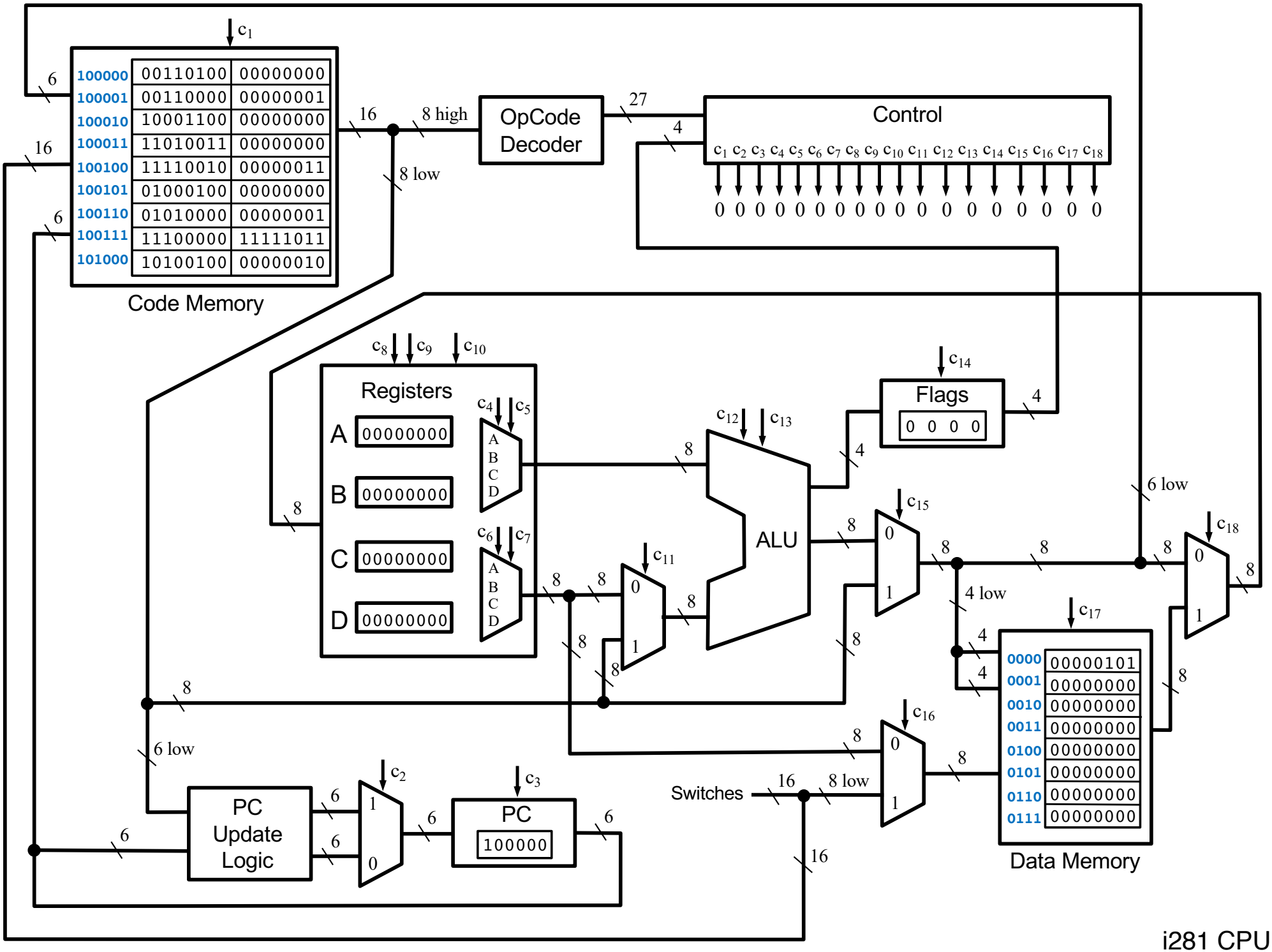
http://www.ece.iastate.edu/~alexs/classes/

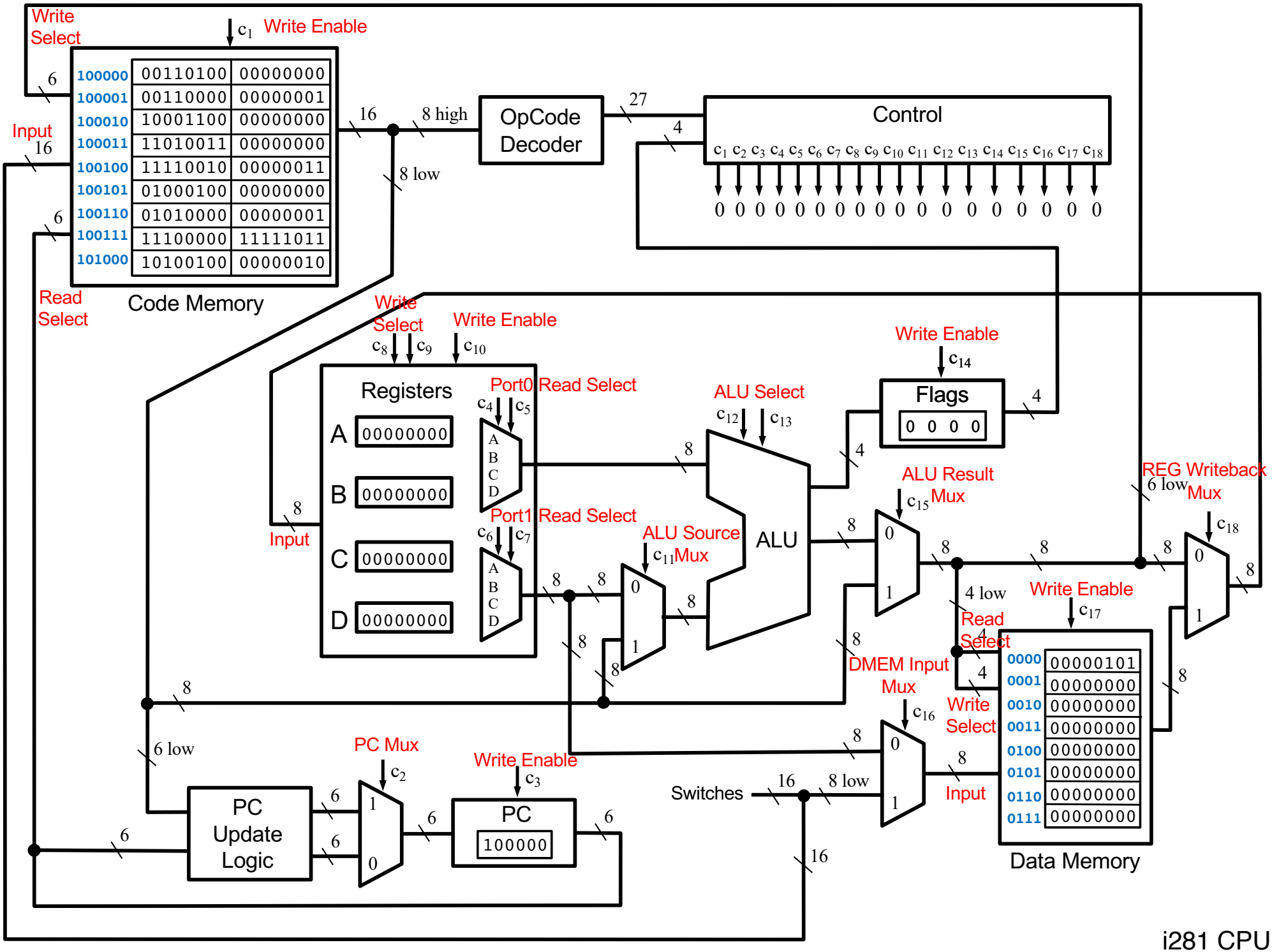
Arithmetic Logic Unit
&
Program Counter

CprE 281: Digital Logic
Iowa State University, Ames, IA

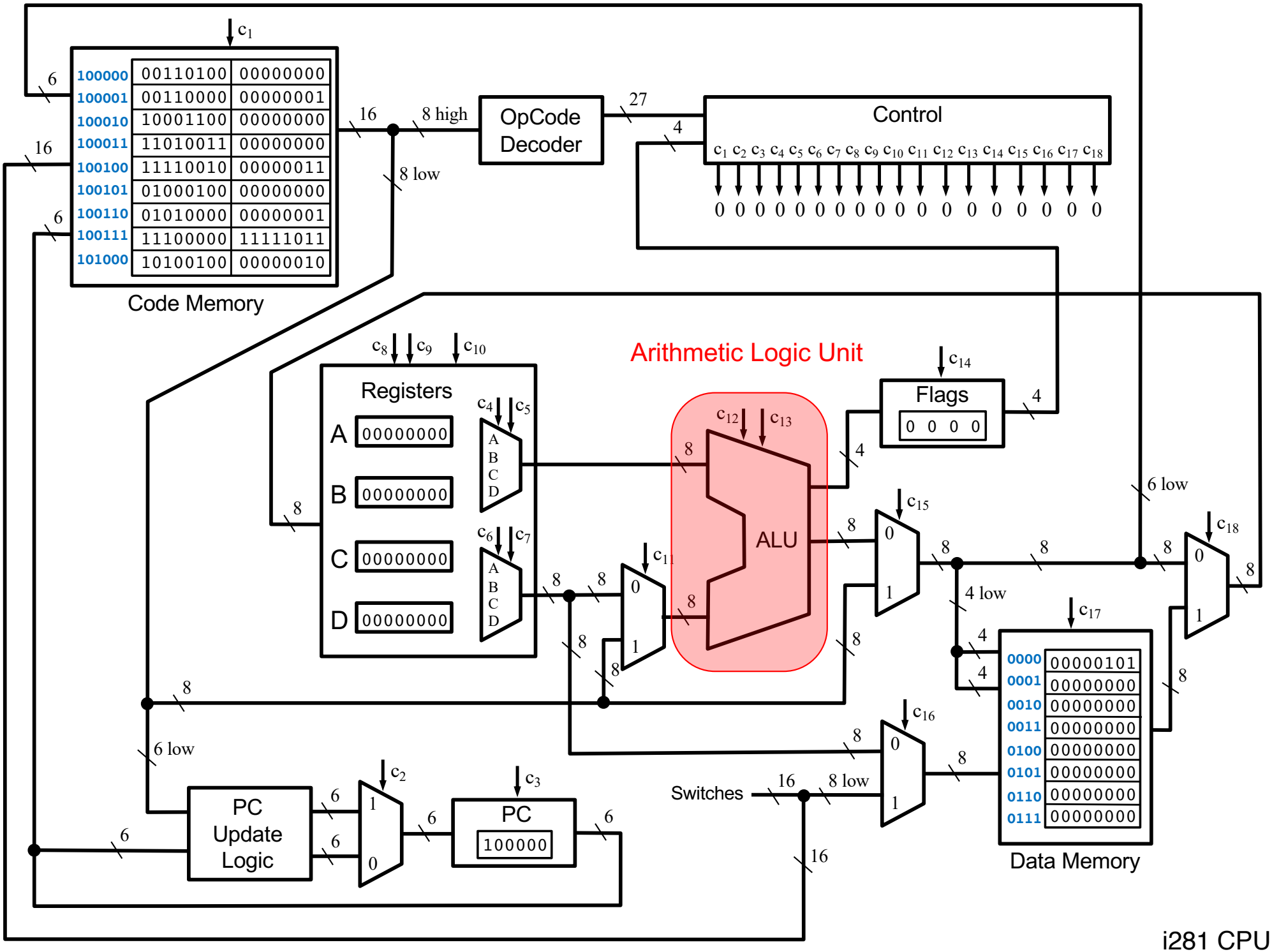

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c14
Flags
0 0 0 0
c12 c13
ALU
c15
c11
0 1
c18
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
c2
c3
PC Update Logic
PC
100000
i281 CPU

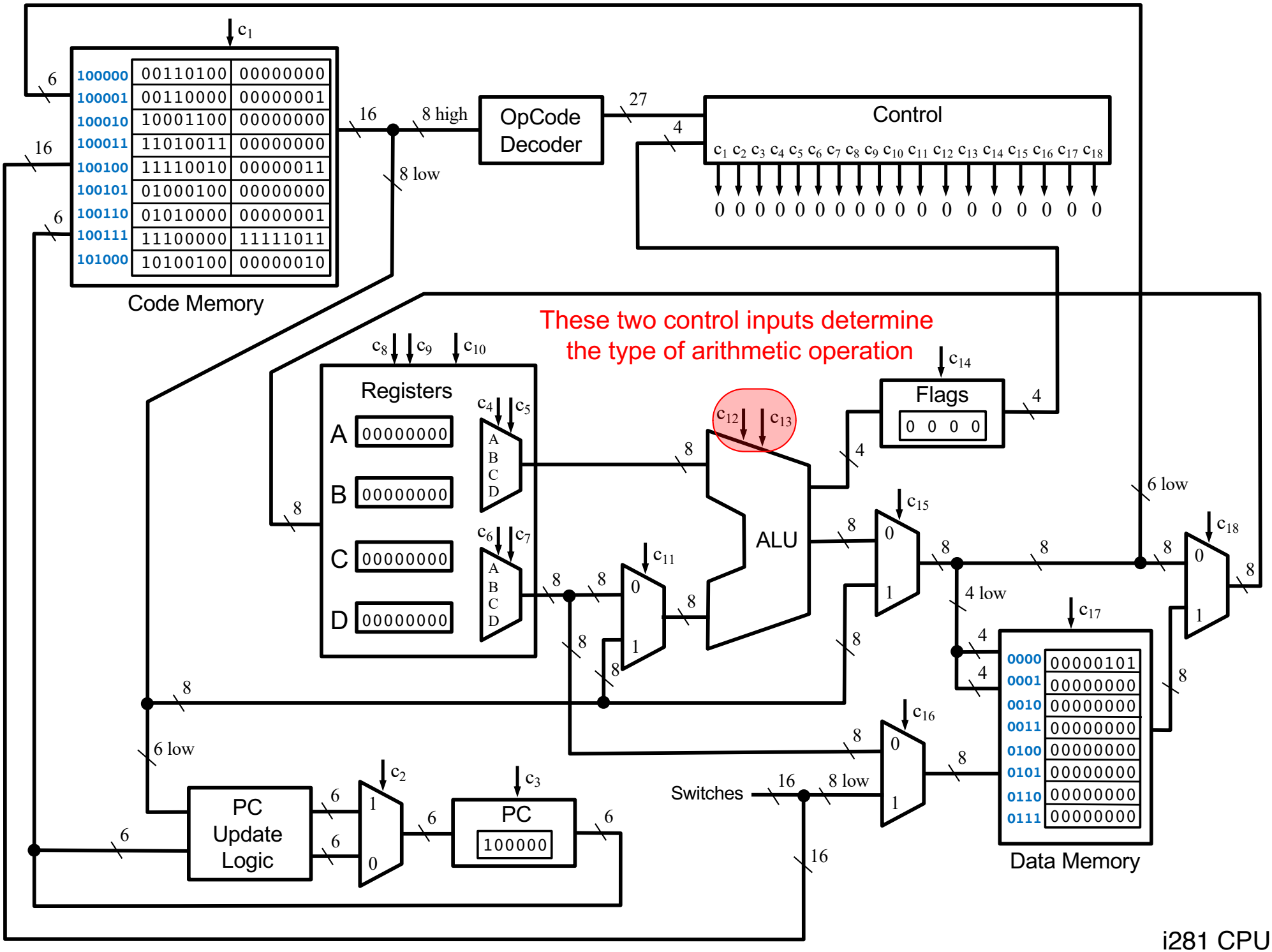

Write Select
c1 Write Enable
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Input
Read Select
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Write Select
c8 c9
Write Enable
c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
Port0 Read Select
c4 c5
Port1 Read Select
c6 c7
A B C D
Input
ALU Select
c12 c13
ALU
ALU Source
c11 Mux
Write Enable
c14
Flags
0 0 0 0
ALU Result
c15 Mux
REG Writeback
Mux
c18
Write Enable
c17
Read Select
Write Select
DMEM Input
Mux
c16
Input
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Mux
c2
Write Enable
c3
PC
100000
PC Update Logic
Switches
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
16
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
6 low
i281 CPU


Write Select
c1 Write Enable
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
Input
16
6
Read Select
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Write Select
c8 c9
Write Enable
c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
Port0 Read Select
c4 c5
Port1 Read Select
c6 c7
A B C D
8
Input
ALU Select
c12 c13
ALU
Write Enable
c14
Flags
0 0 0 0
4
ALU Source
c11 Mux
ALU Result
c15 Mux
REG Writeback
6 low
Mux
c18
4 low
Read Select
Write Enable
c17
Write Select
DMEM Input
Mux
c16
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
6 low
PC Mux
c2
Write Enable
c3
PC Update Logic
PC
100000
Switches
16
8 low
Input
i281 CPU

The Arithmetic Logic Unit (ALU)


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Arithmetic Logic Unit
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c11
c12 c13
ALU
c14
Flags
0 0 0 0
4
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
c2
c3
PC Update Logic
PC
100000
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
8 high
8 low
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
These two control inputs determine
the type of arithmetic operation
Registers
A 00000000
B 00000000
C 00000000
D 00000000
ALU
Flags
0 0 0 0
PC
Update
Logic
PC
100000
Switches
Data Memory
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
i281 CPU

This ALU Can Perform 4 Operations

ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

This ALU Can Perform 4 Operations

Names of these control lines

C_{12} C_{13}

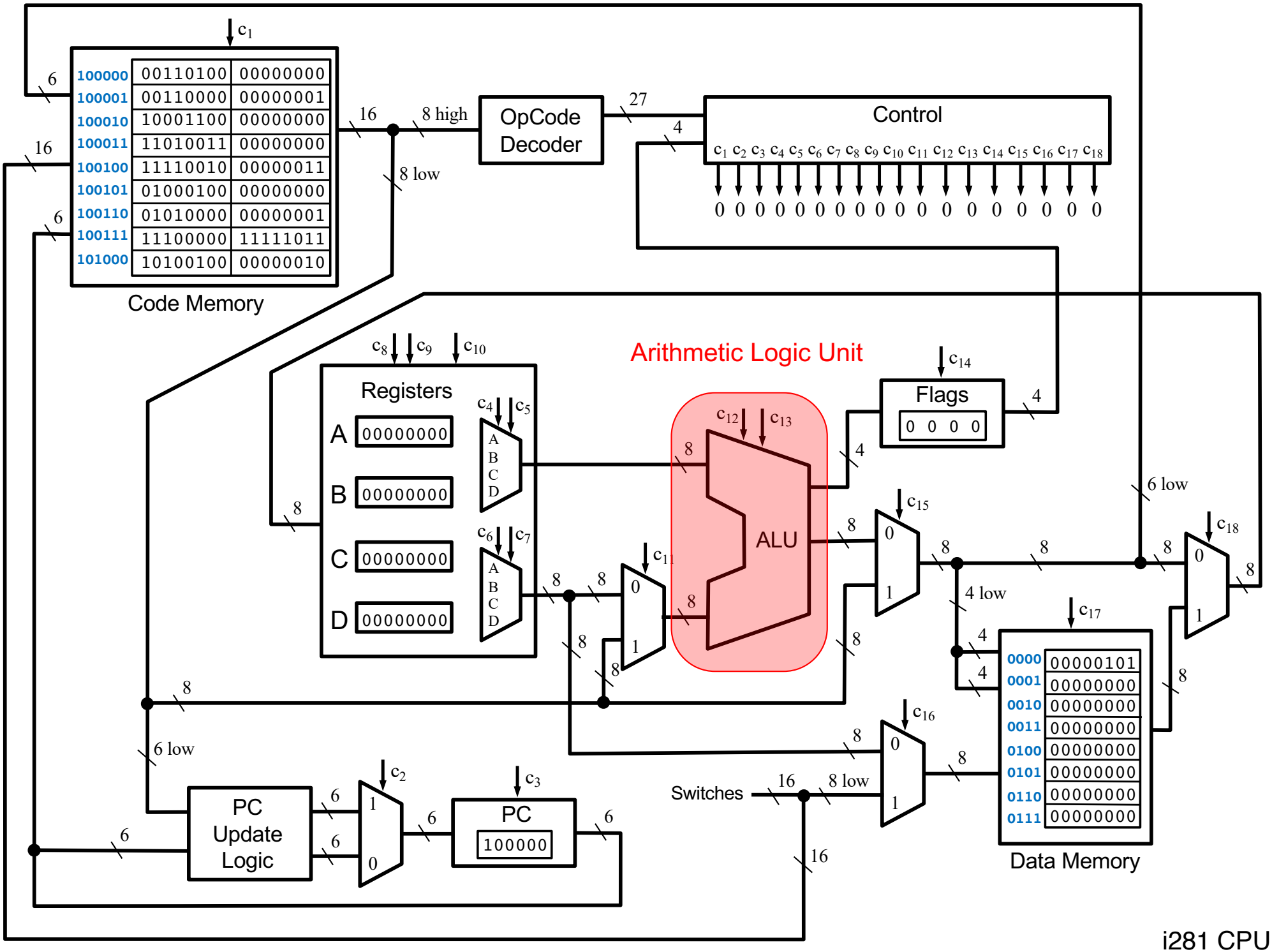
ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

This ALU Can Perform 4 Operations

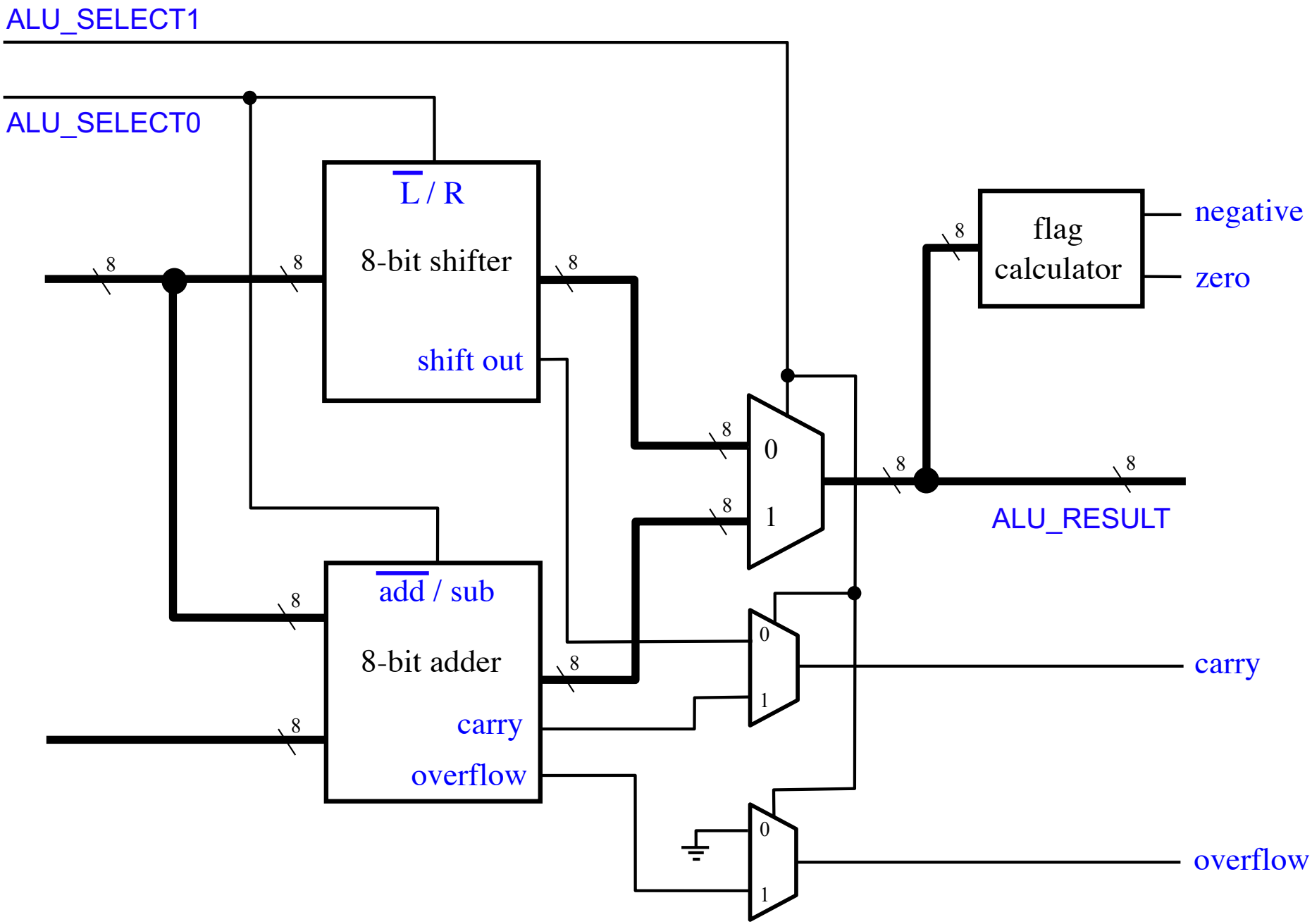
C_{12}	C_{13}	
ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

Both SUB and CMP are implemented as subtraction. They both set the flags.

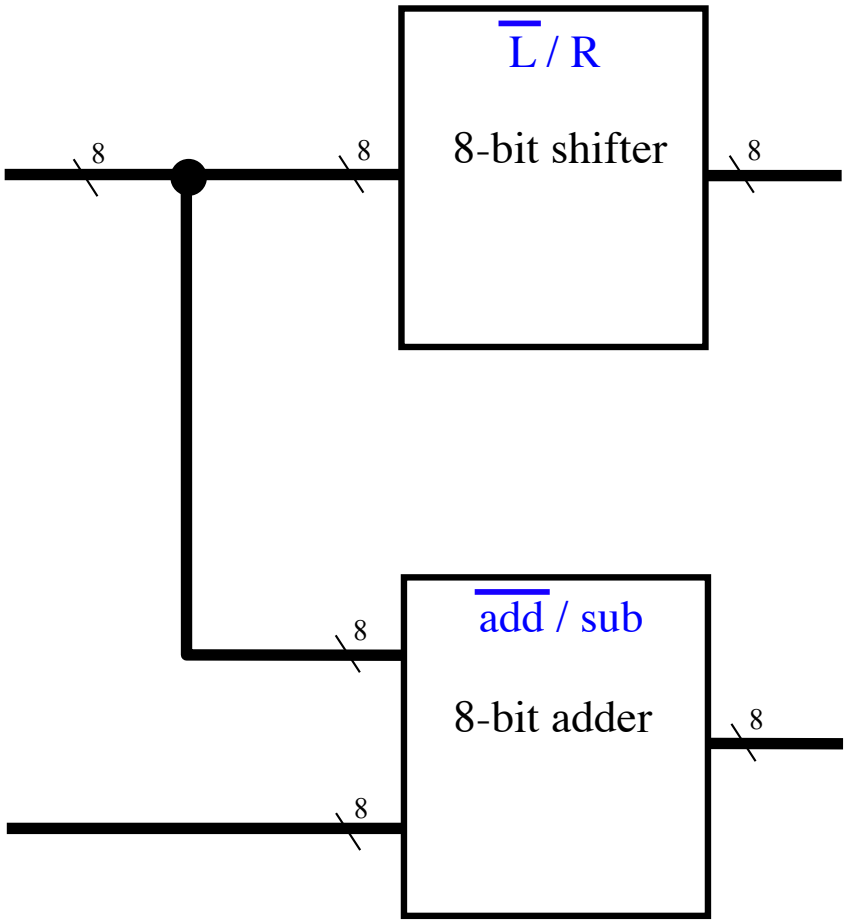
The difference is that CMP does not write back the result of the subtraction to the registers. Only the side effect through the flags remains.


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Arithmetic Logic Unit
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c11
c12 c13
ALU
c14
Flags
0 0 0 0
4
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
c2
c3
PC Update Logic
PC
100000
i281 CPU

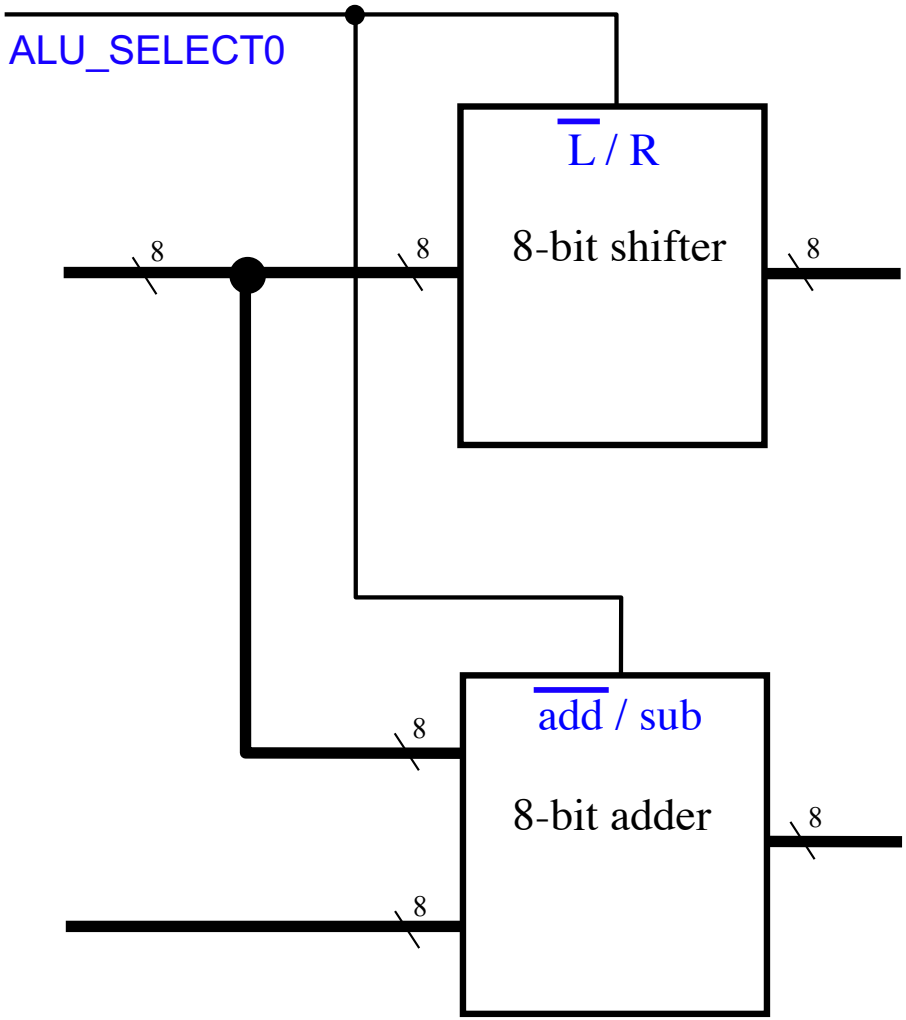
The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
shift out
add / sub
8-bit adder
carry
overflow
flag calculator
negative
zero
ALU_RESULT
carry
overflow
0
1
8

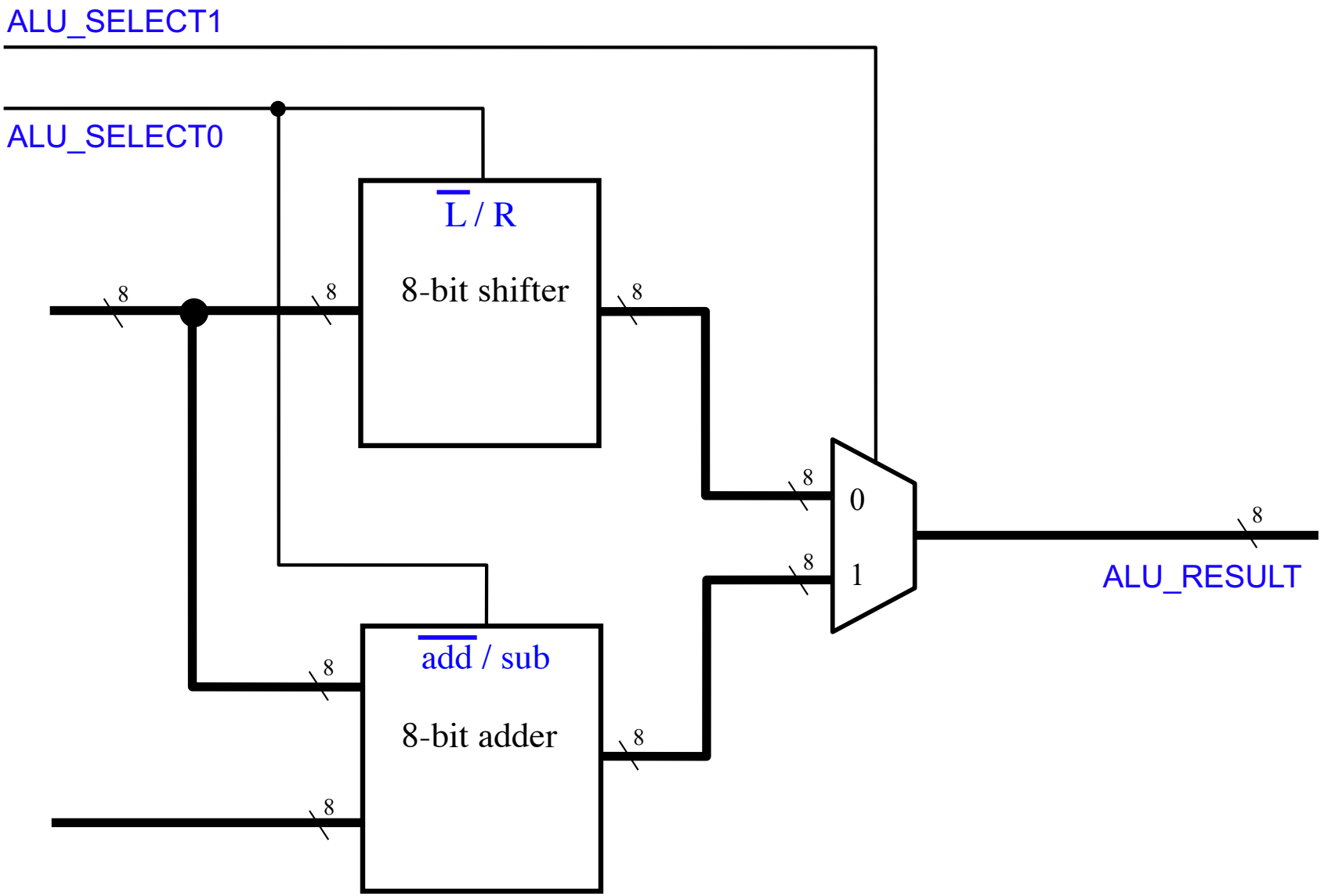
The ALU



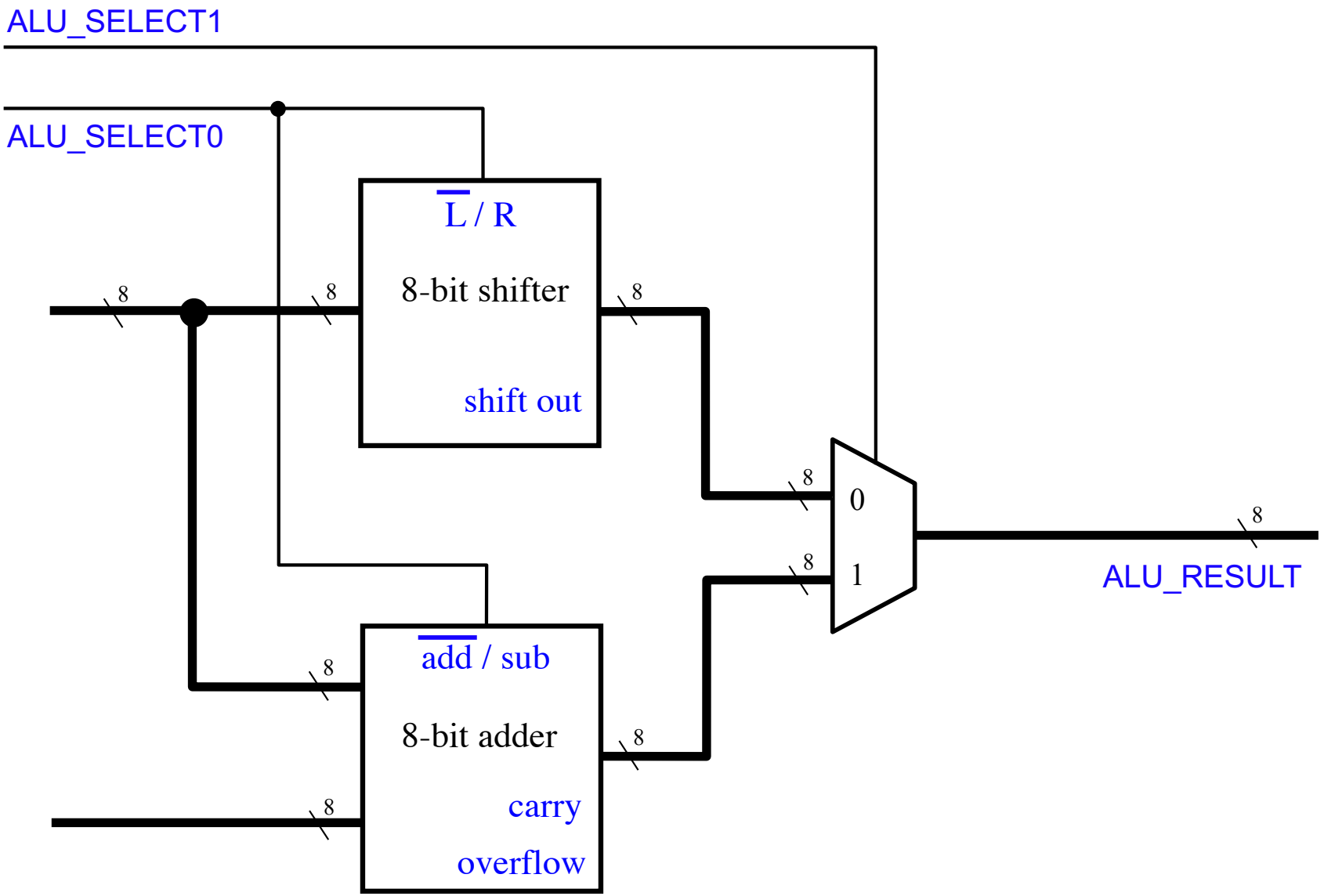
The ALU



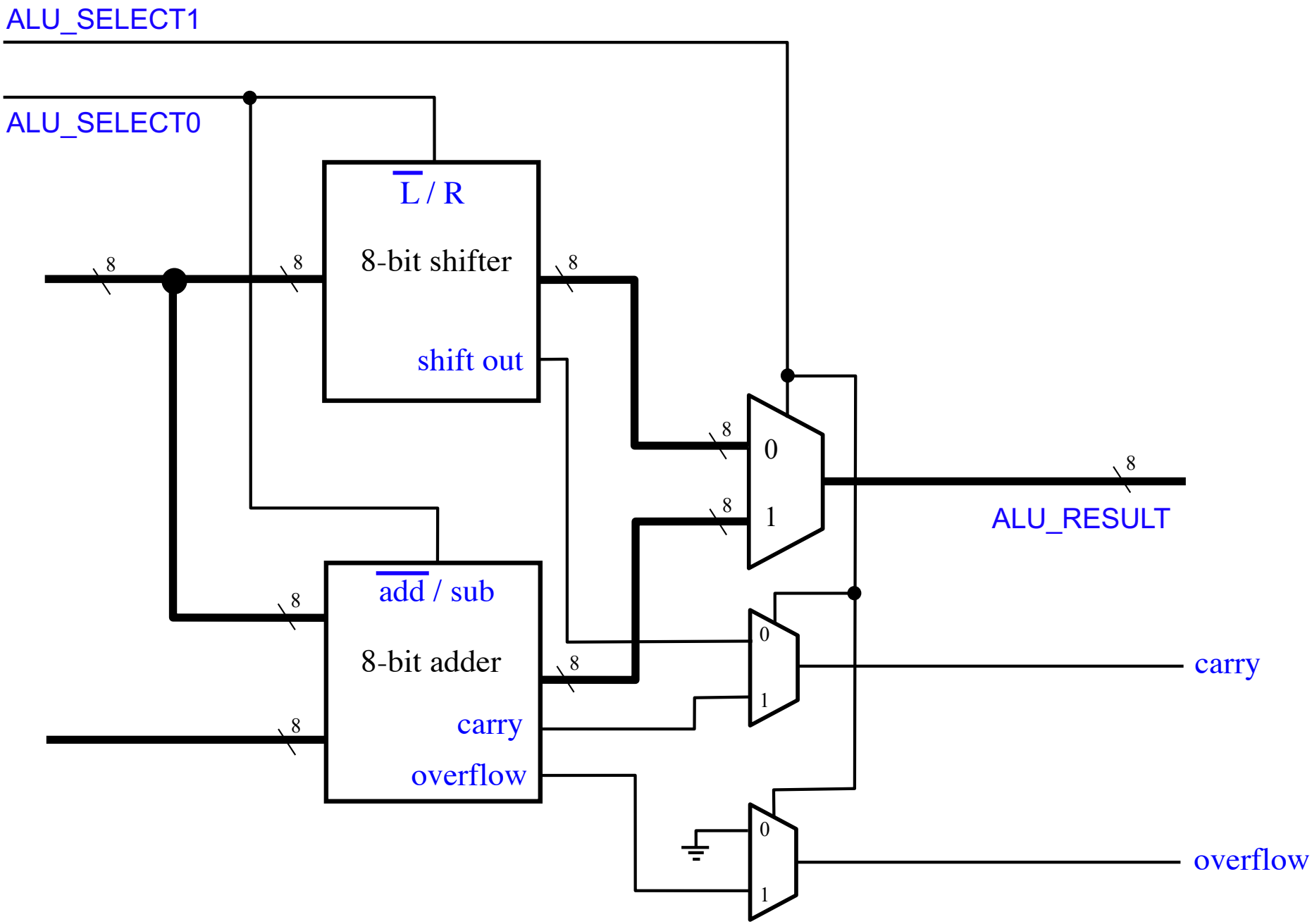
The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
8
8
8
8
8
0
1
8
ALU_RESULT
add / sub
8-bit adder
8
8
8

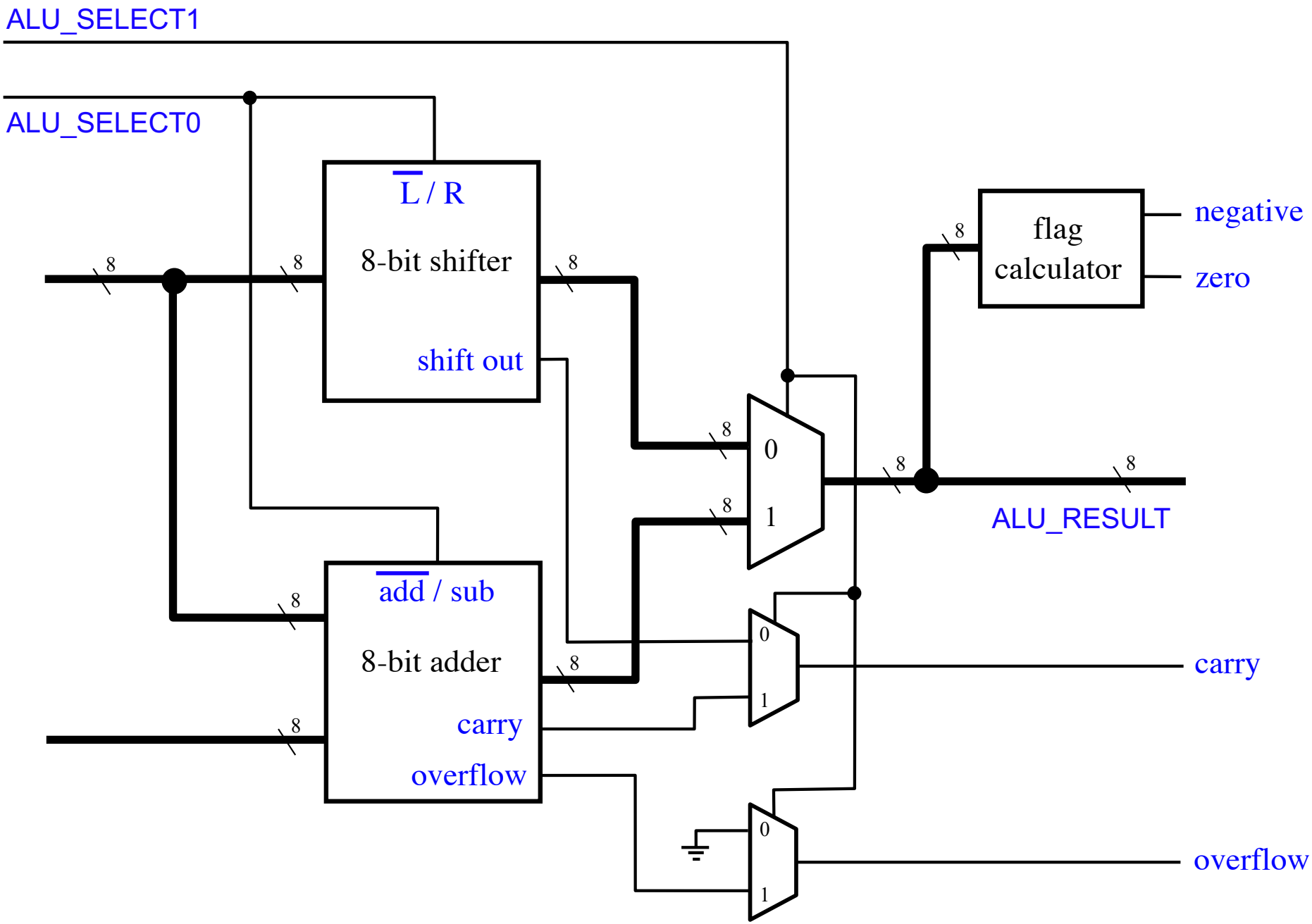
The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
shift out
8
8
8
8
8
0
1
8
ALU_RESULT
add / sub
8-bit adder
carry
overflow
8
8
8

The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
shift out
8
add / sub
8-bit adder
carry
overflow
0
1
ALU_RESULT
carry
overflow

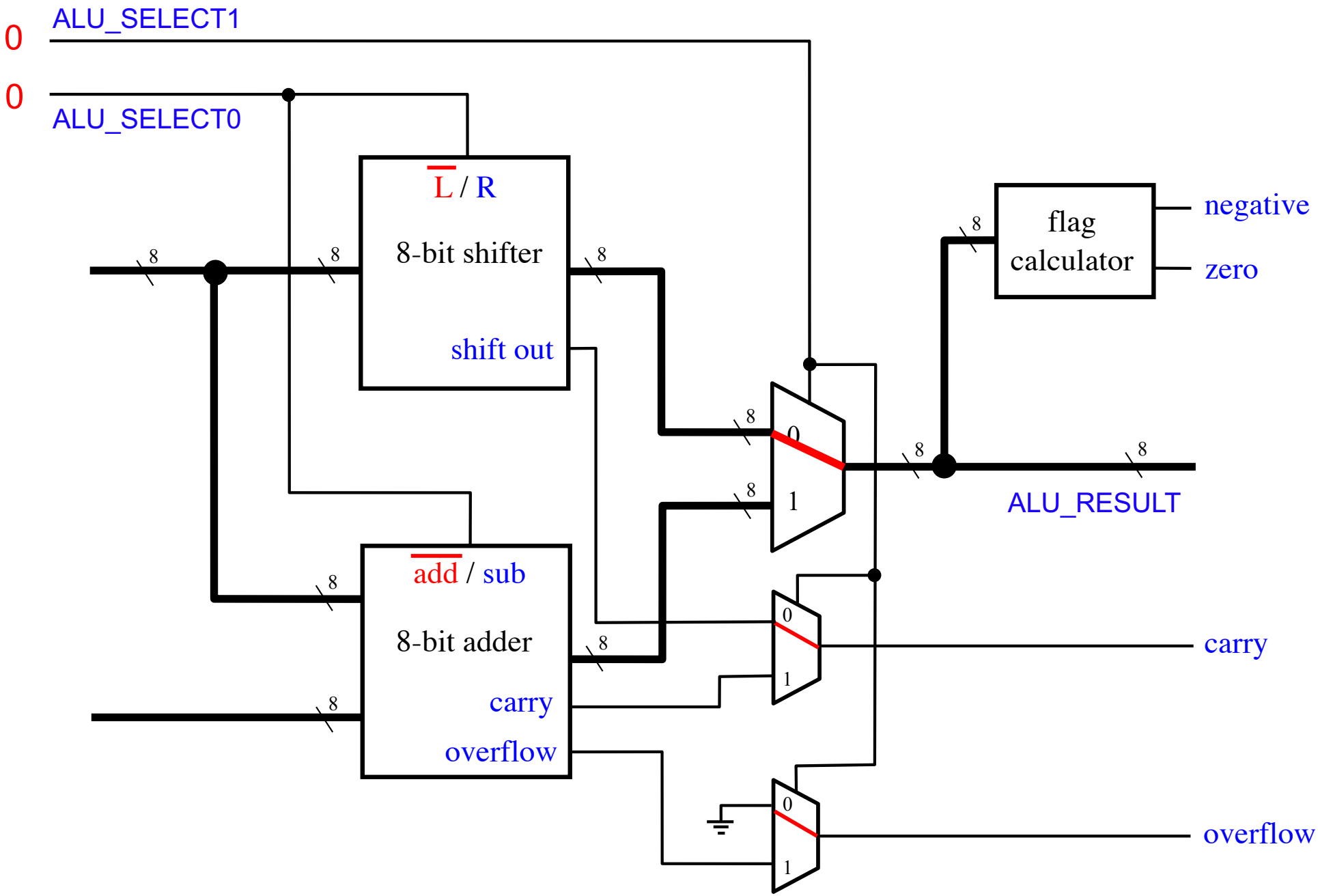
The ALU


ALU_SELECT1
ALU_SELECT0
L / R
8-bit shifter
shift out
add / sub
8-bit adder
carry
overflow
flag calculator
negative
zero
ALU_RESULT
carry
overflow
0
1
8

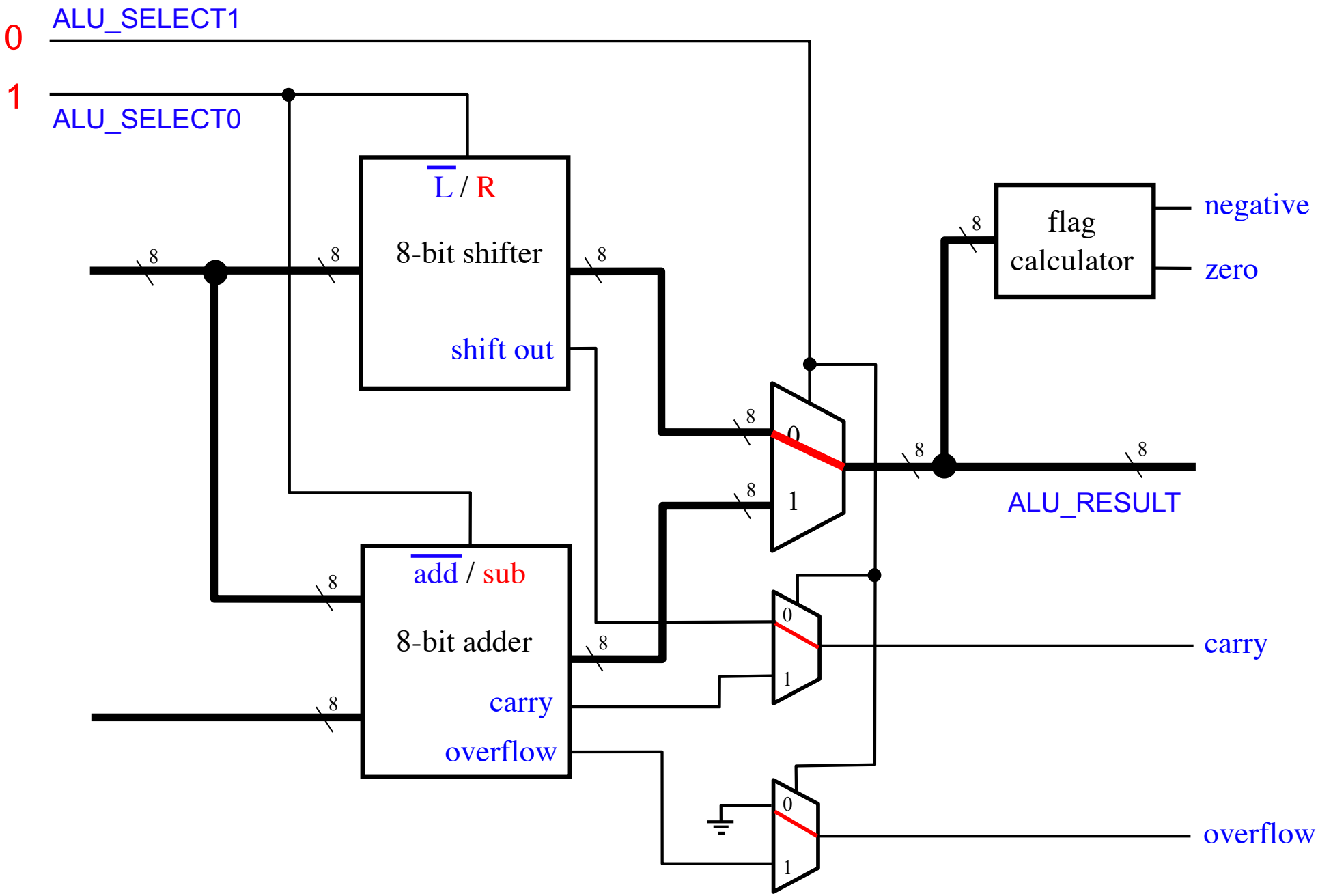
This ALU Can Perform 4 Operations

ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

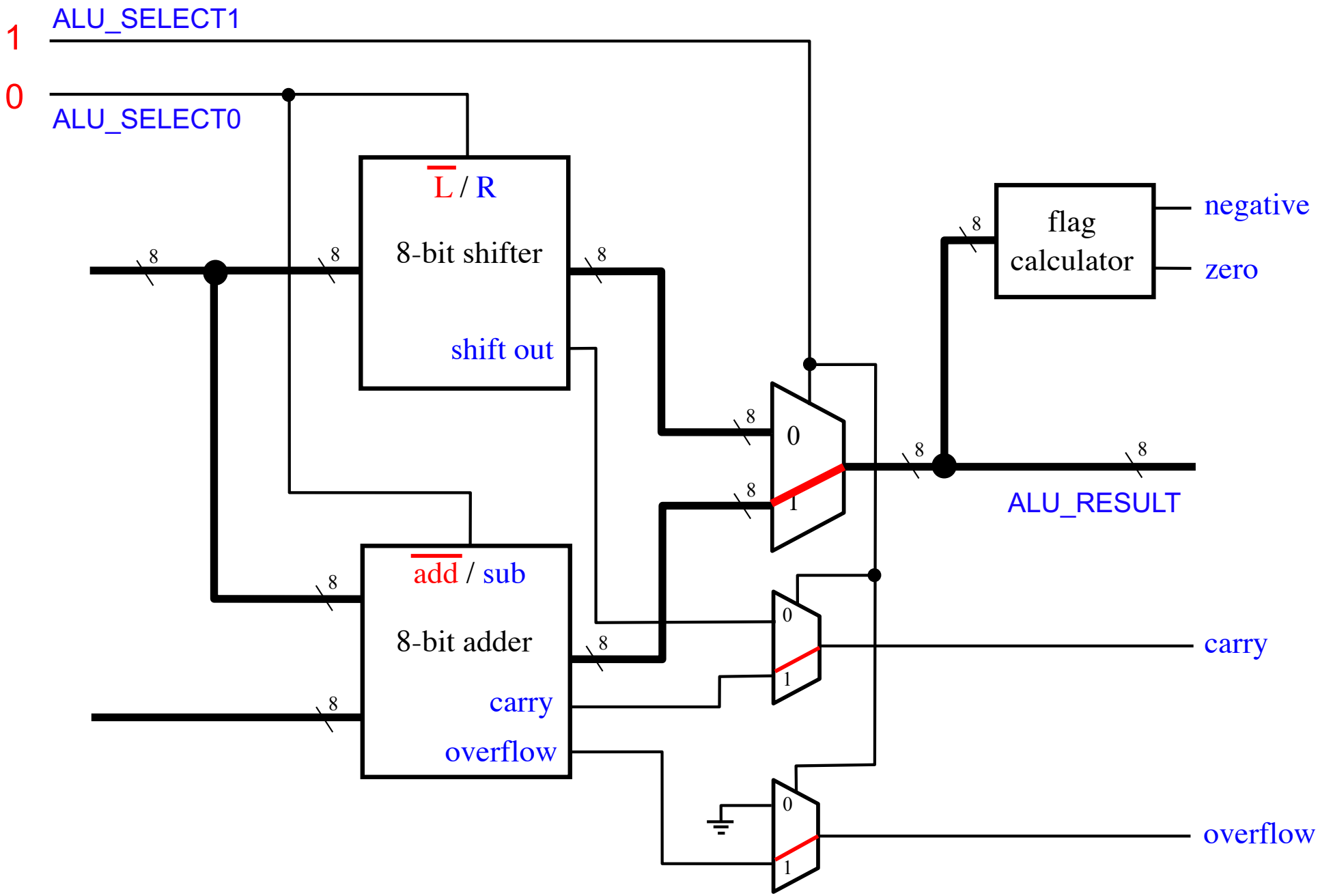
SHIFTL


0
ALU_SELECT1
0
ALU_SELECT0
L / R
8-bit shifter
shift out
8
8
8
flag
calculator
negative
zero
8
0
1
8
8
8
ALU_RESULT
add / sub
8-bit adder
carry
overflow
8
8
8
0
1
carry
0
1
overflow

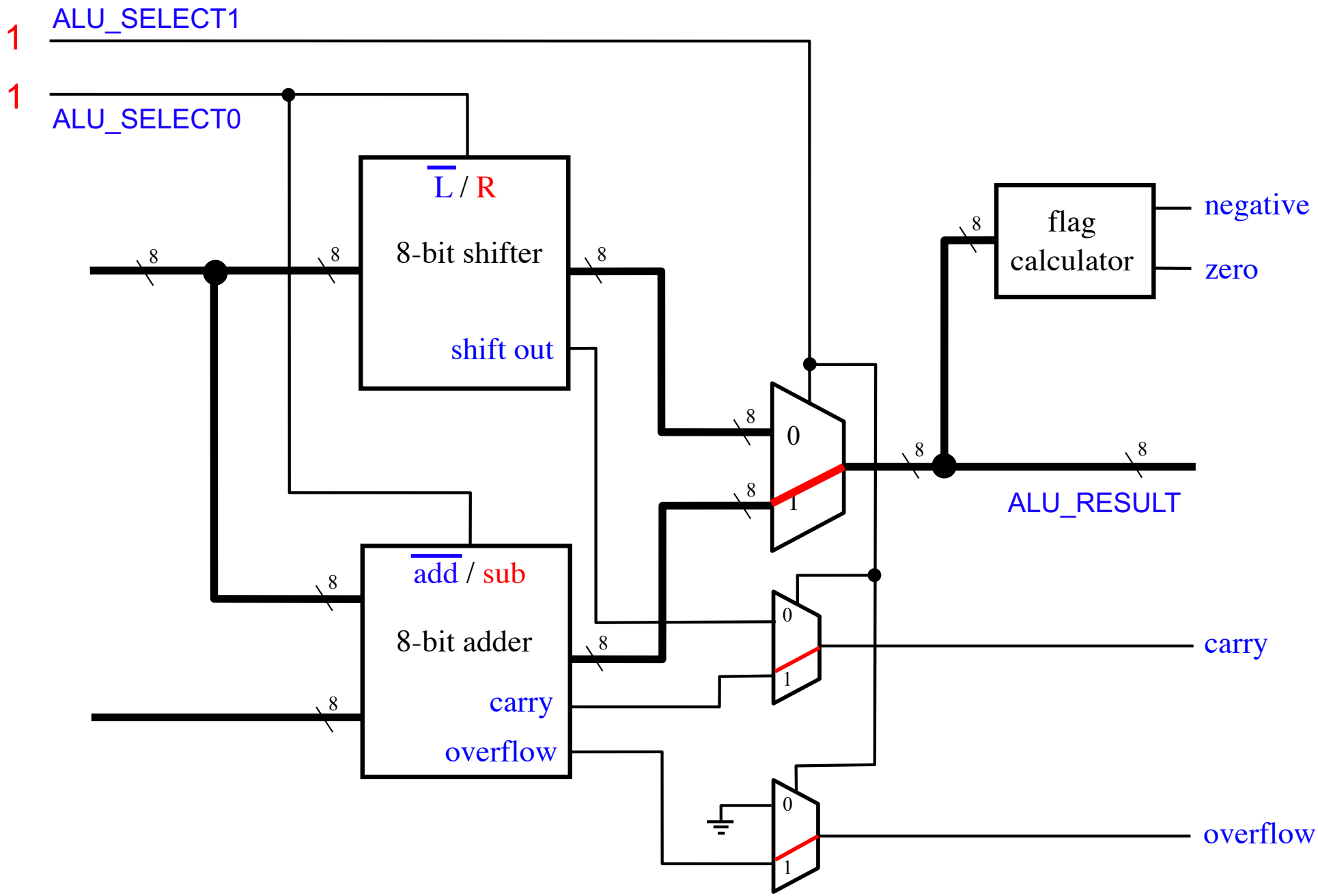
SHIFTR


0
ALU_SELECT1
1
ALU_SELECT0
L / R
8-bit shifter
shift out
8
flag
calculator
negative
zero
0
1
ALU_RESULT
add / sub
8-bit adder
carry
overflow
carry
overflow

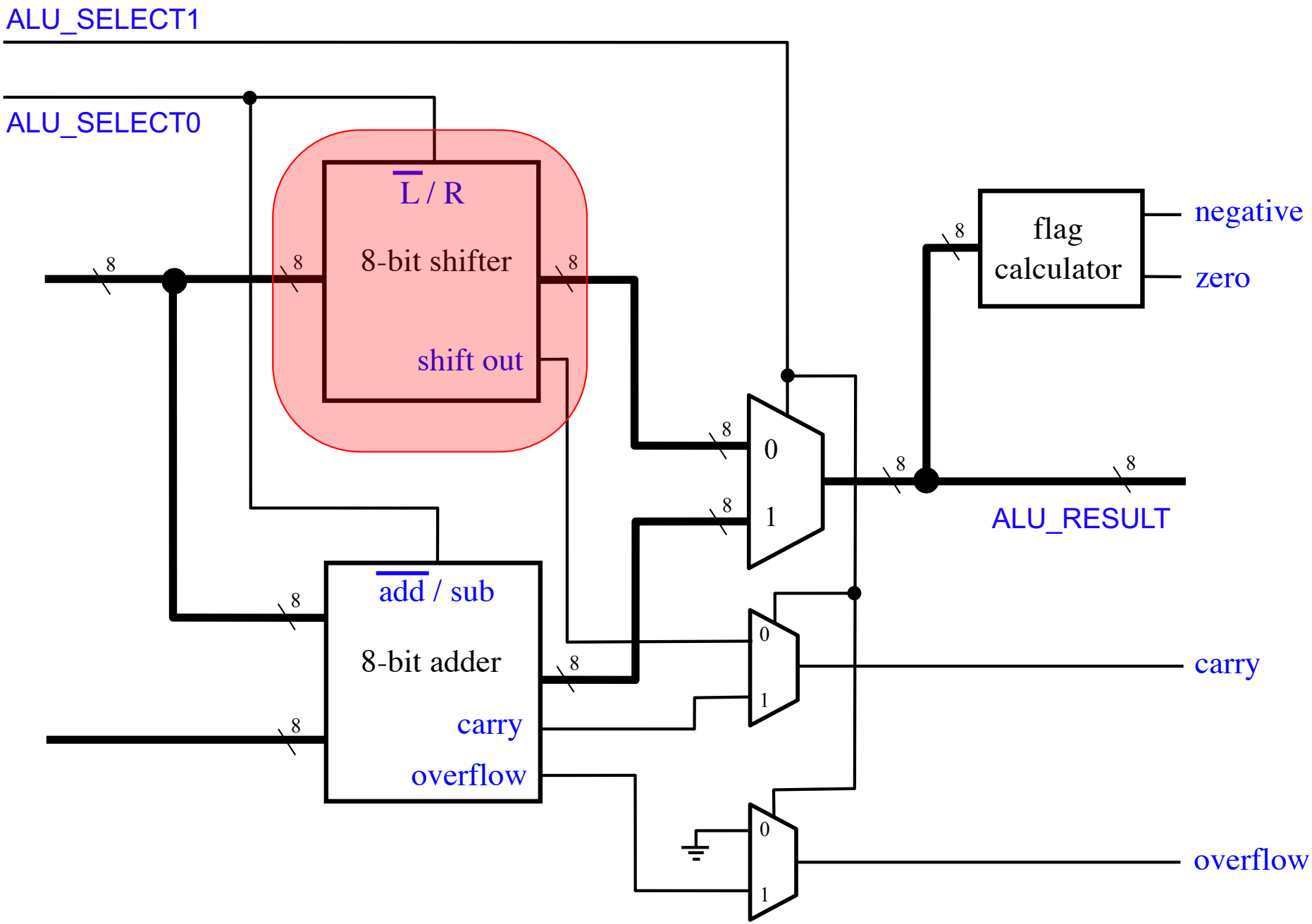
ADD


1
ALU_SELECT1
0
ALU_SELECT0
L / R
8-bit shifter
shift out
add / sub
8-bit adder
carry
overflow
flag calculator
negative
zero
ALU_RESULT
carry
overflow
8
0
1

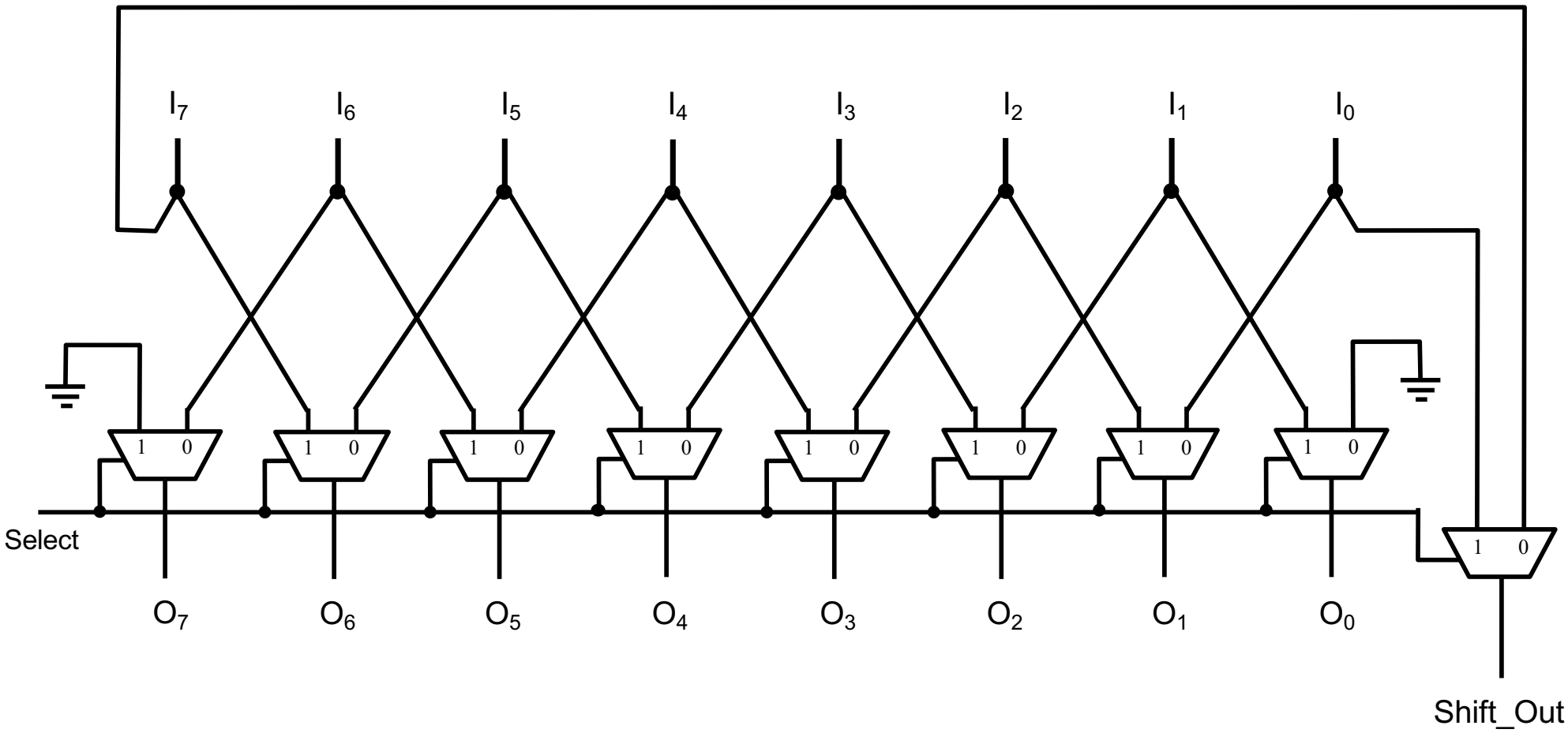
SUB / CMP

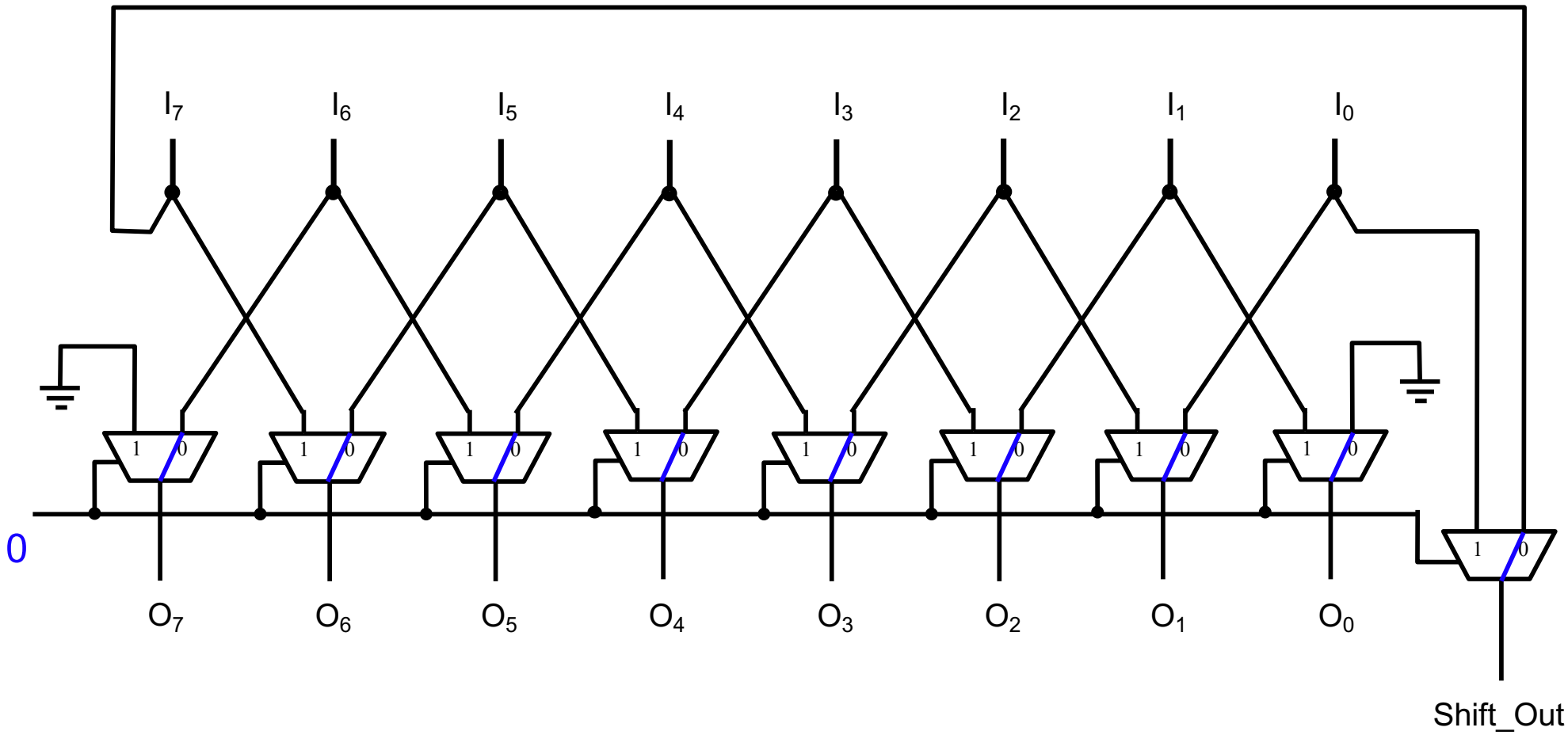

1
ALU_SELECT1
1
ALU_SELECT0
L / R
8-bit shifter
shift out
8
8
8
flag
calculator
negative
zero
0
1
ALU_RESULT
add / sub
8-bit adder
carry
overflow
carry
overflow

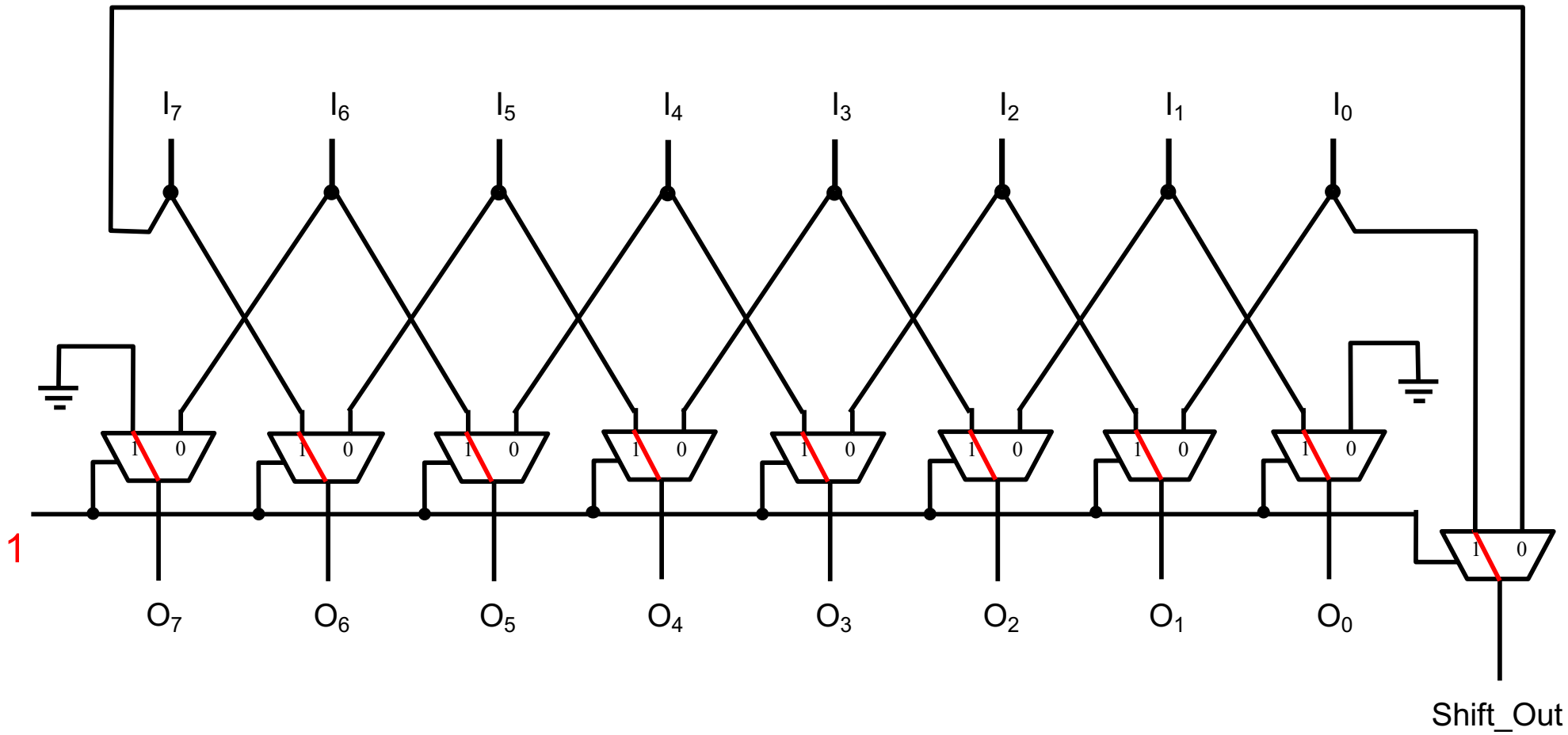
The Shifter Circuit



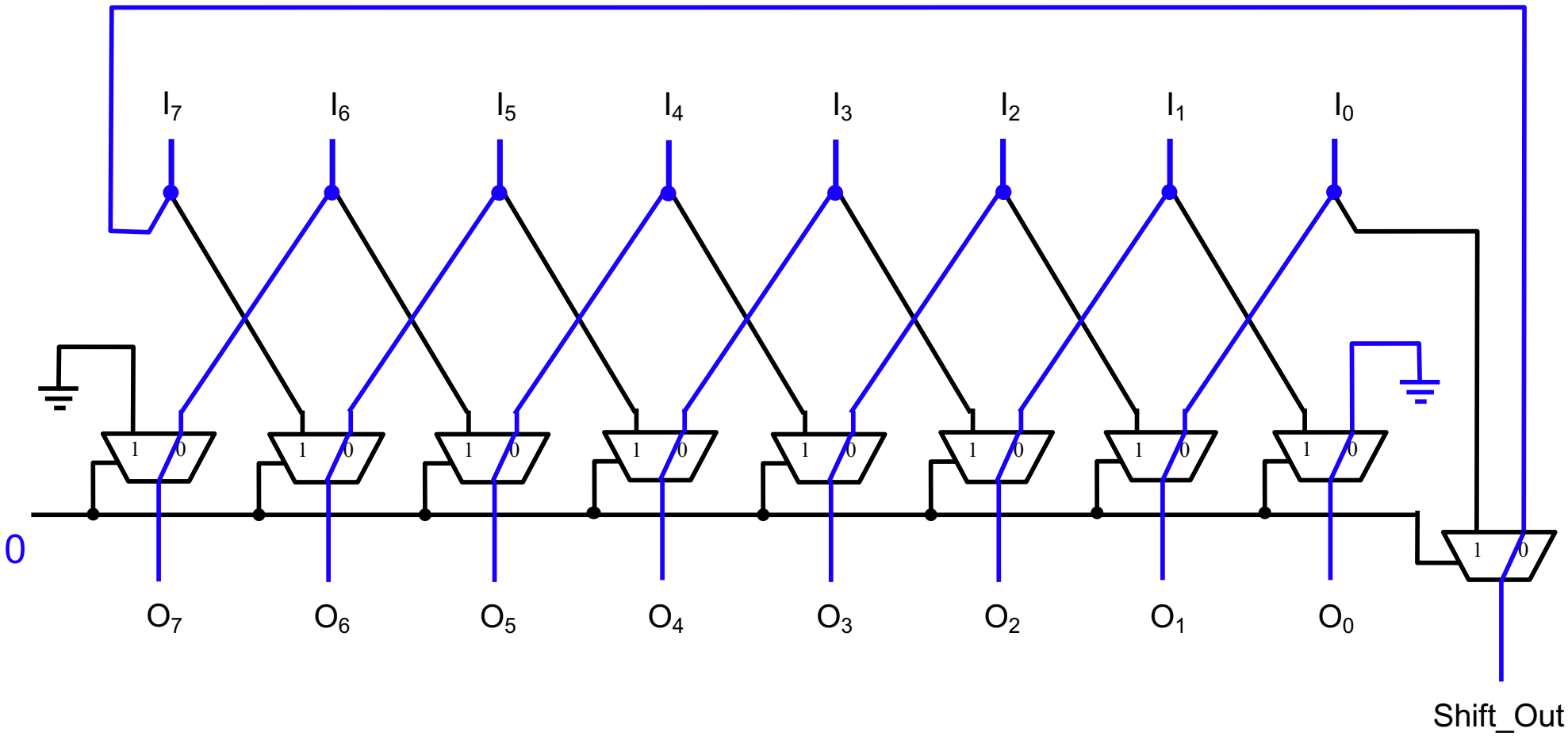
The Shifter Circuit



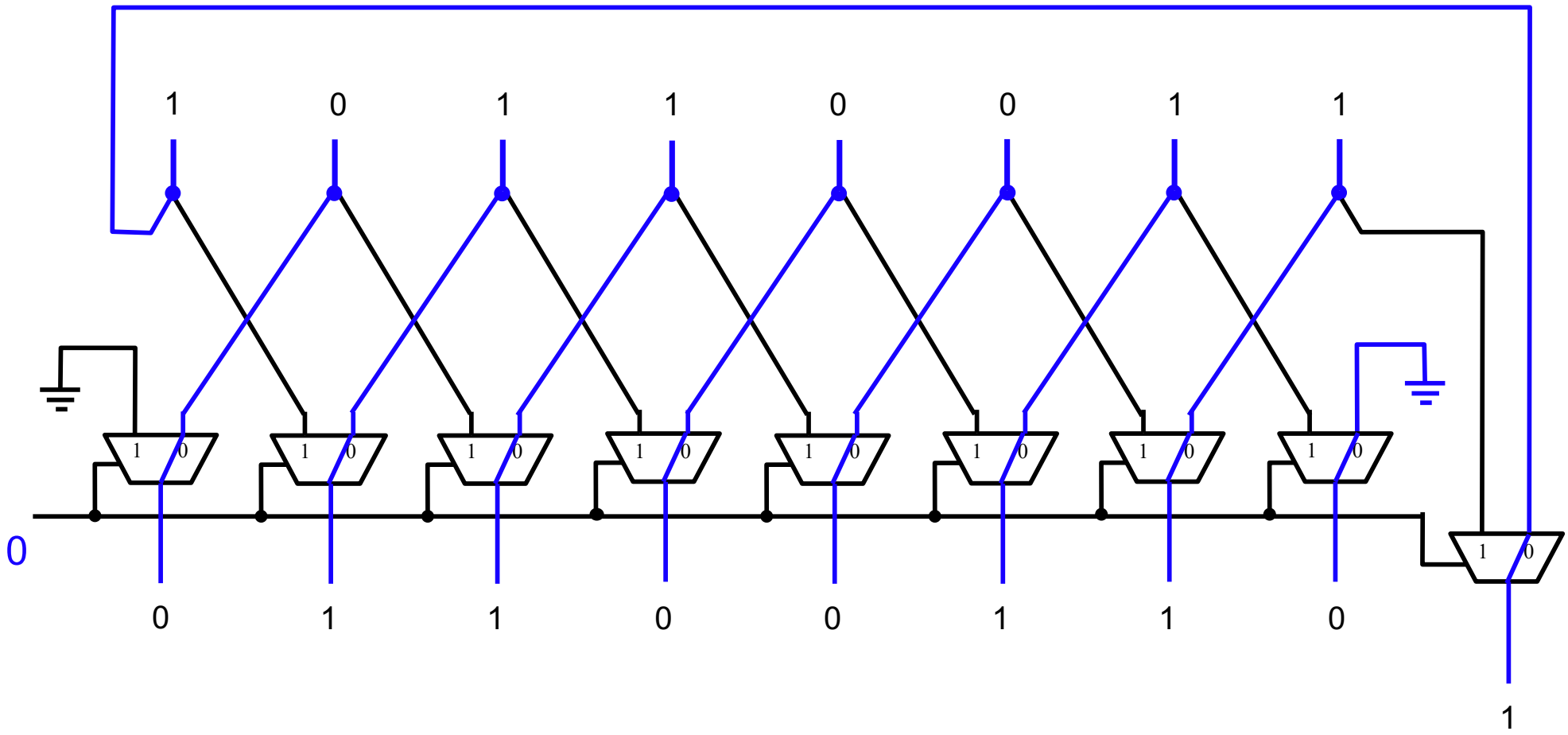

I7
I6
I5
I4
I3
I2
I1
I0
1
0
0
O7
O6
O5
O4
O3
O2
O1
O0
Shift_Out


I7
I6
I5
I4
I3
I2
I1
I0
1
O7
O6
O5
O4
O3
O2
O1
O0
Shift_Out

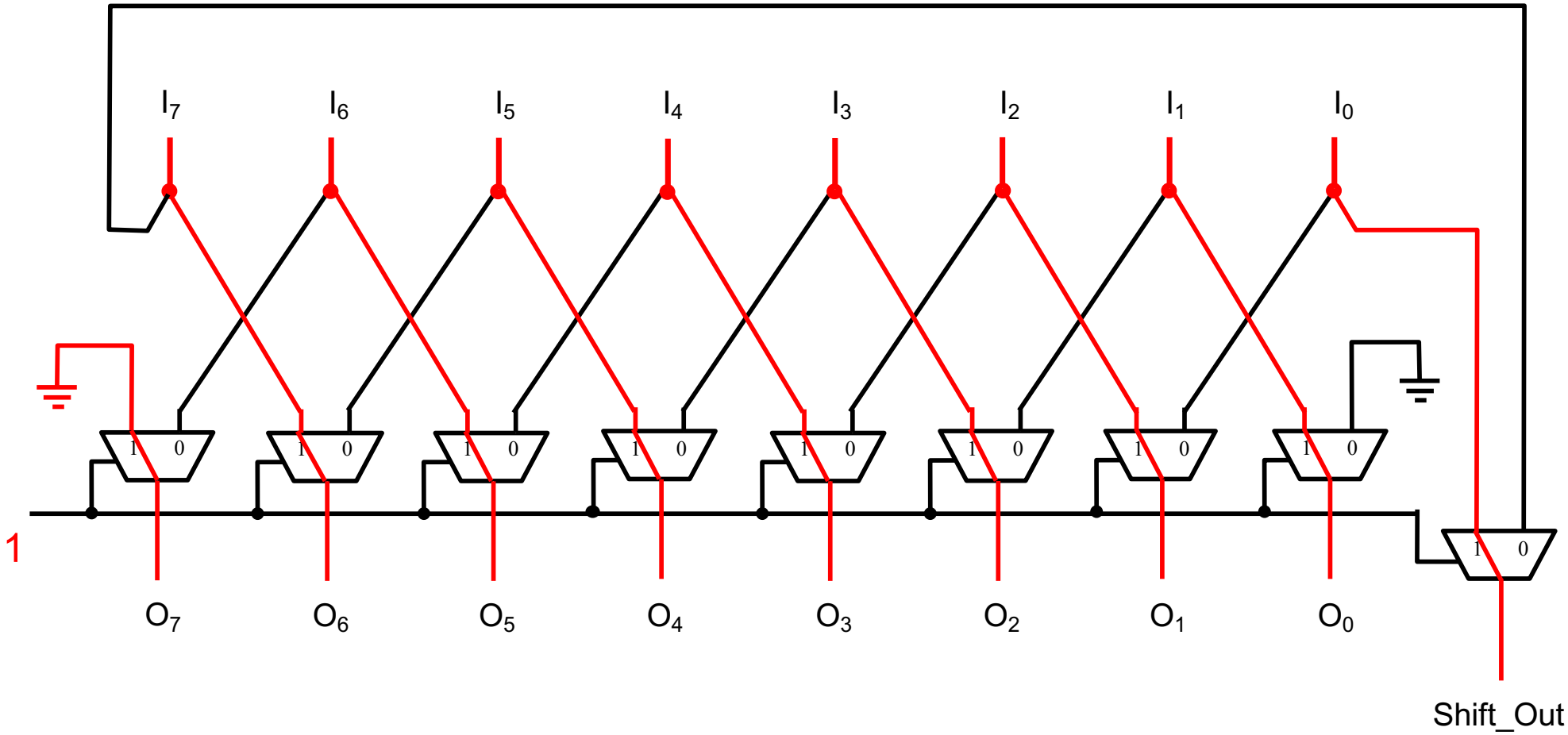
Shift Left



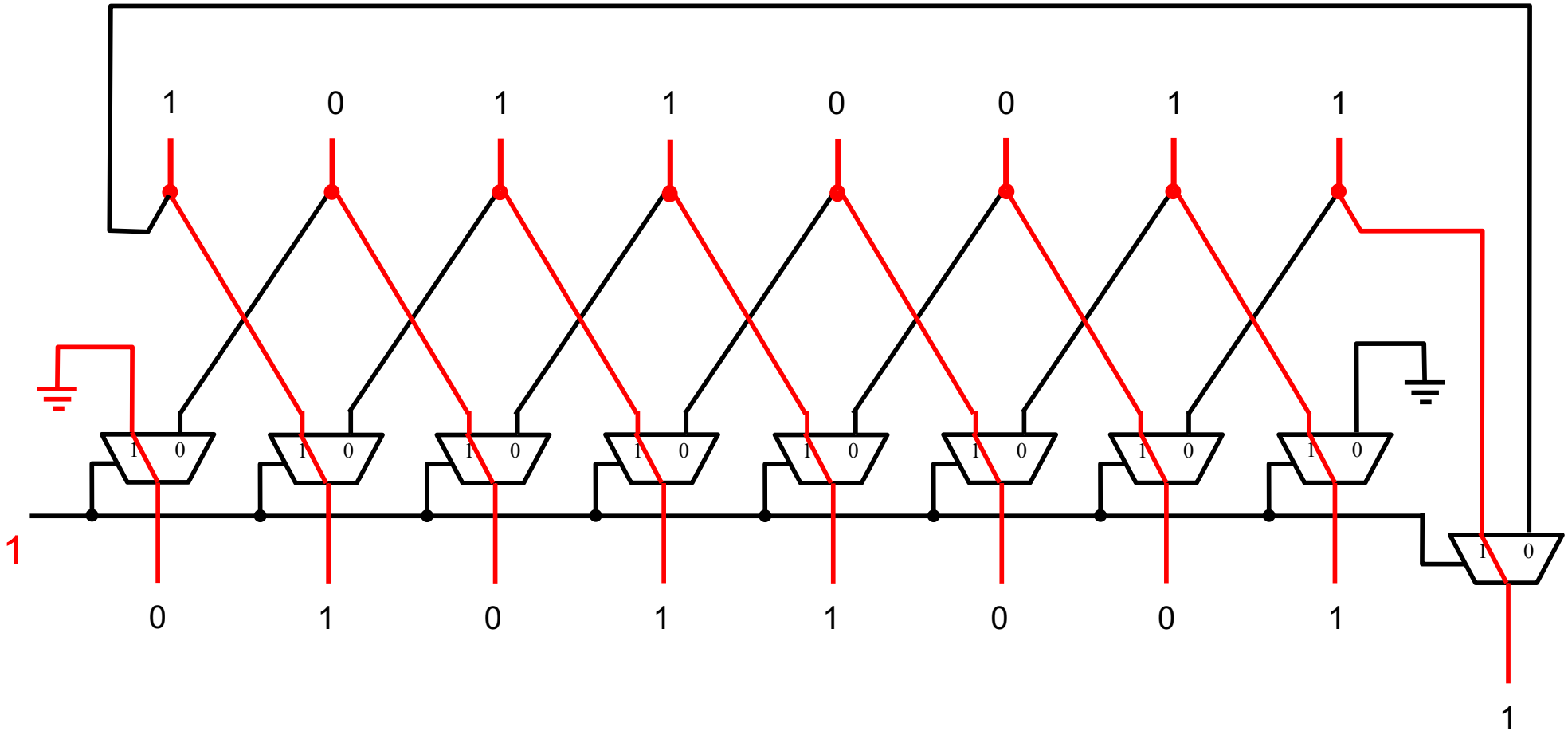
Shift Left



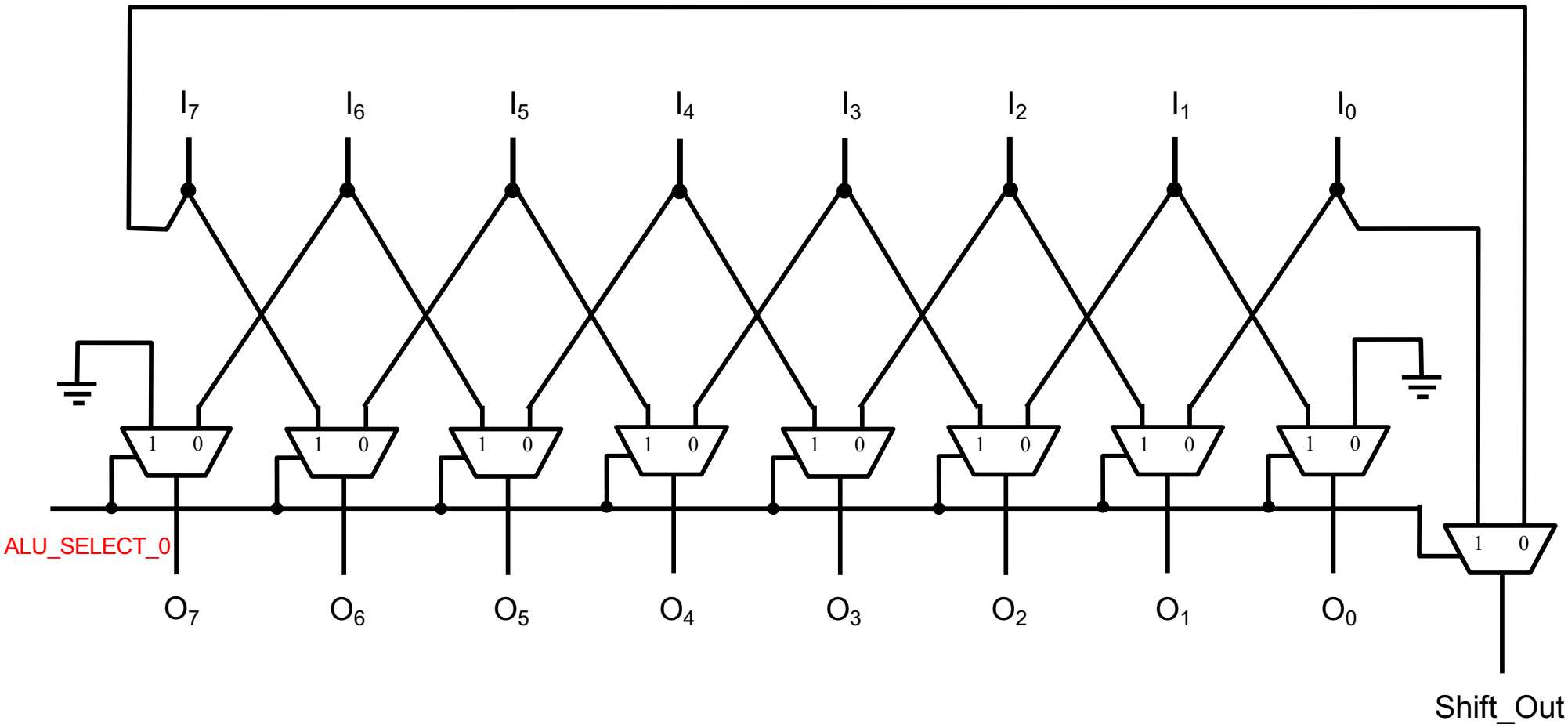
Shift Right



Shift Right

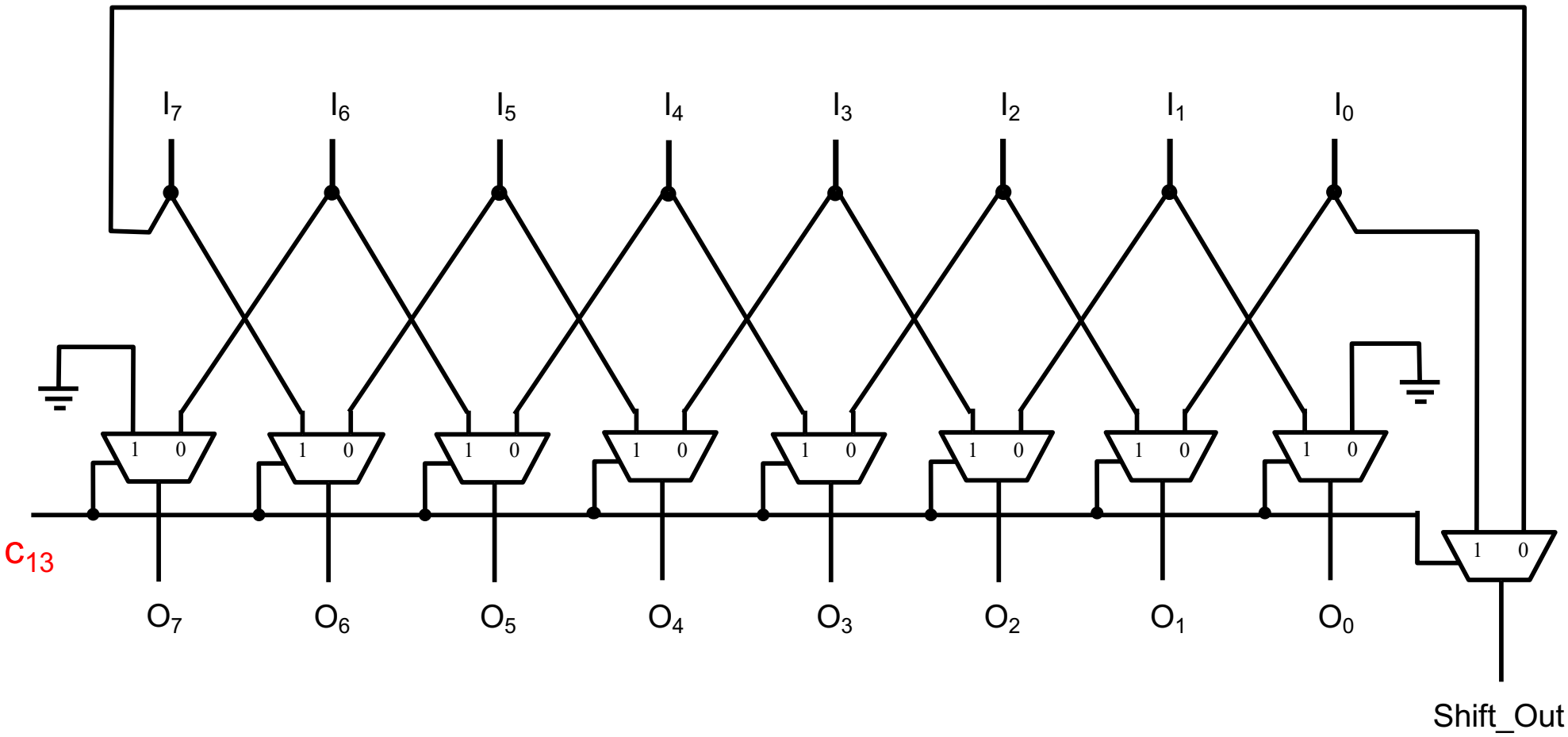


The ALU Shifter Circuit

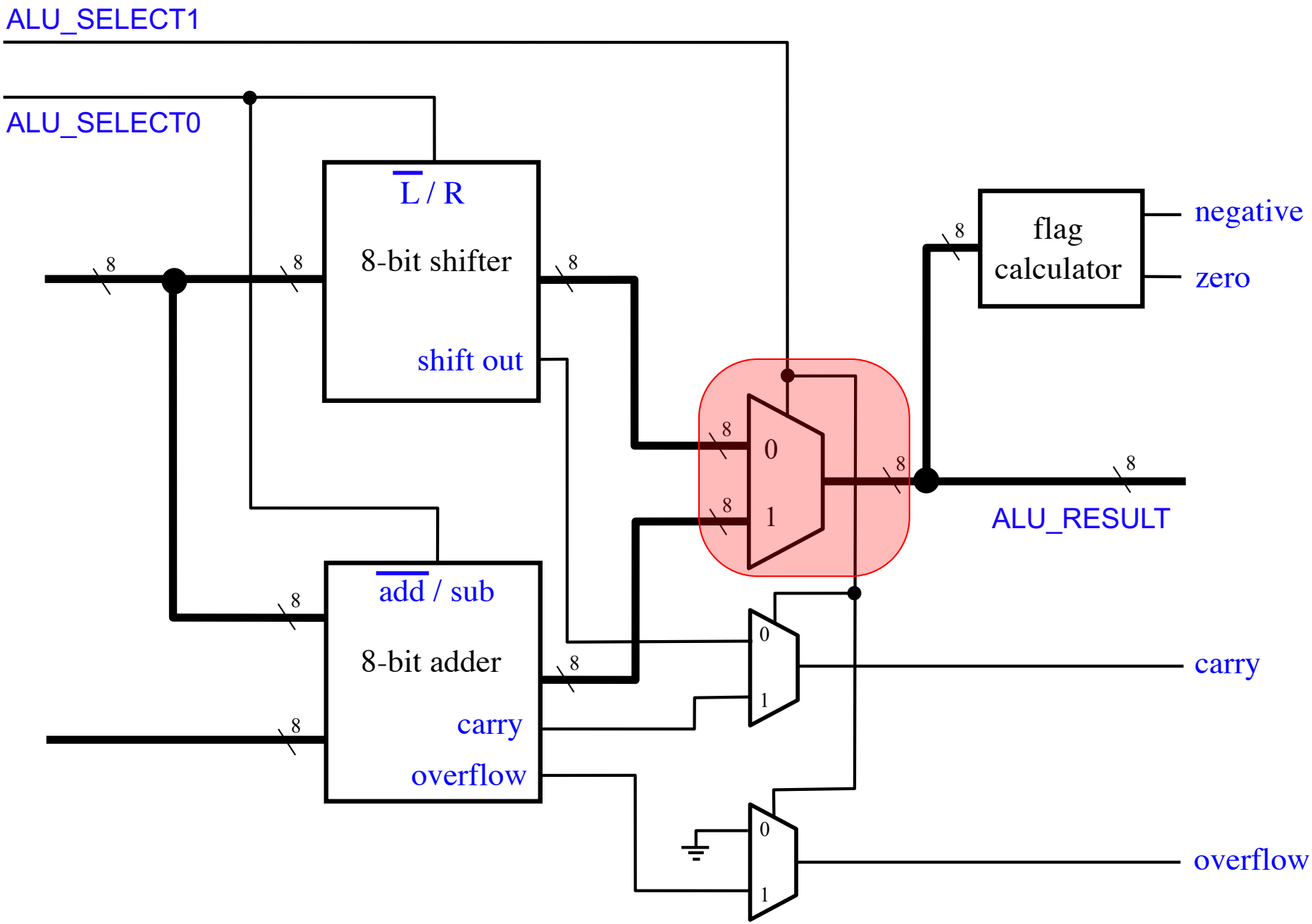


Goes to
Carry Flag

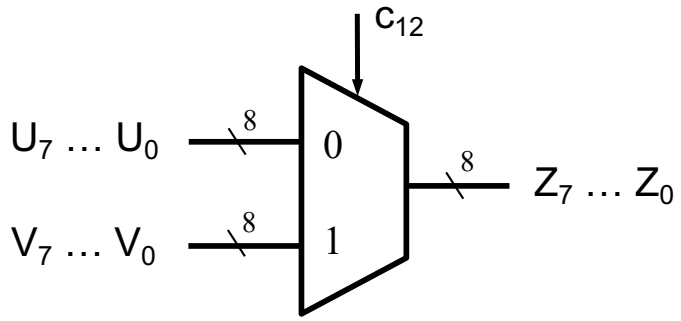
The ALU Shifter Circuit

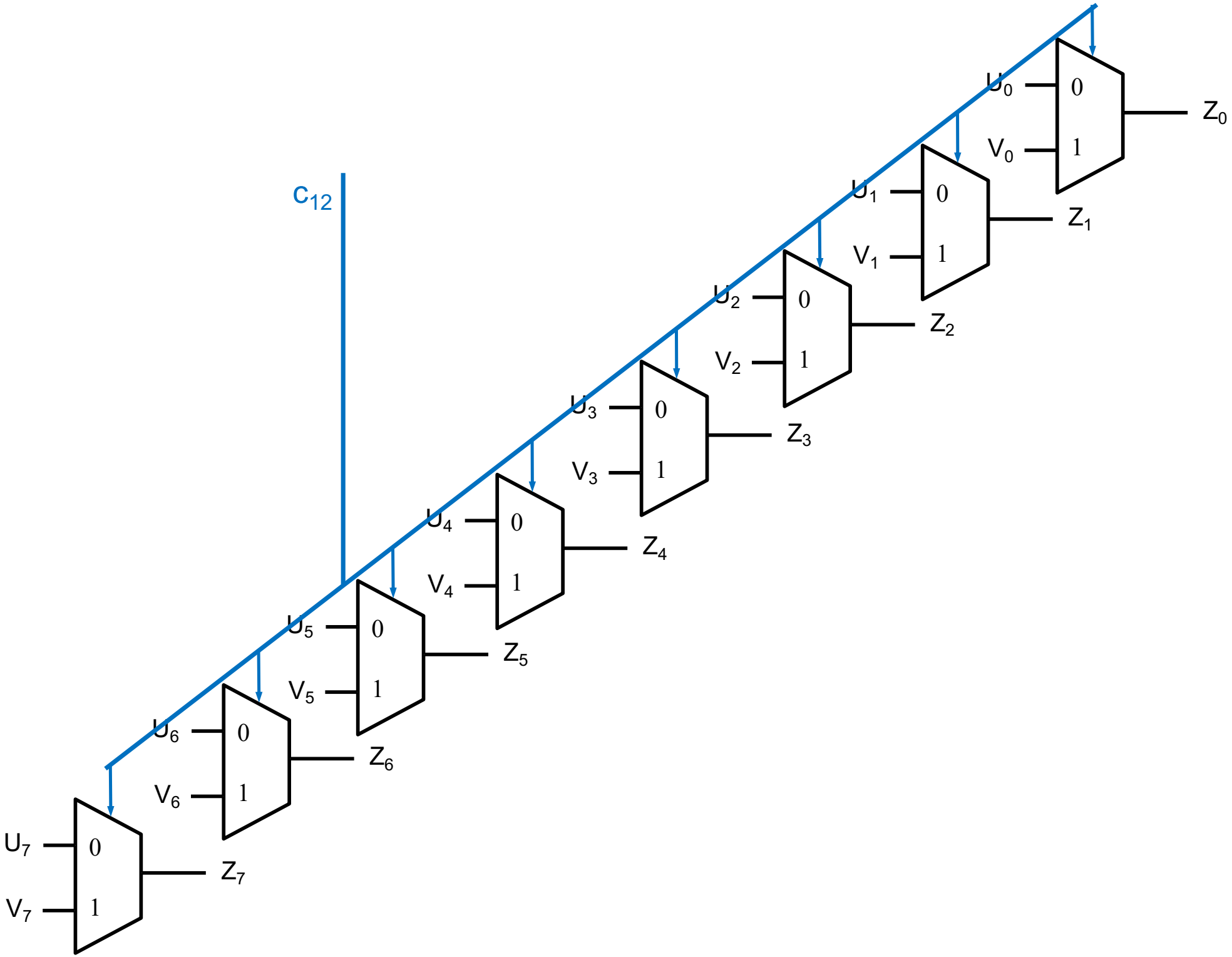


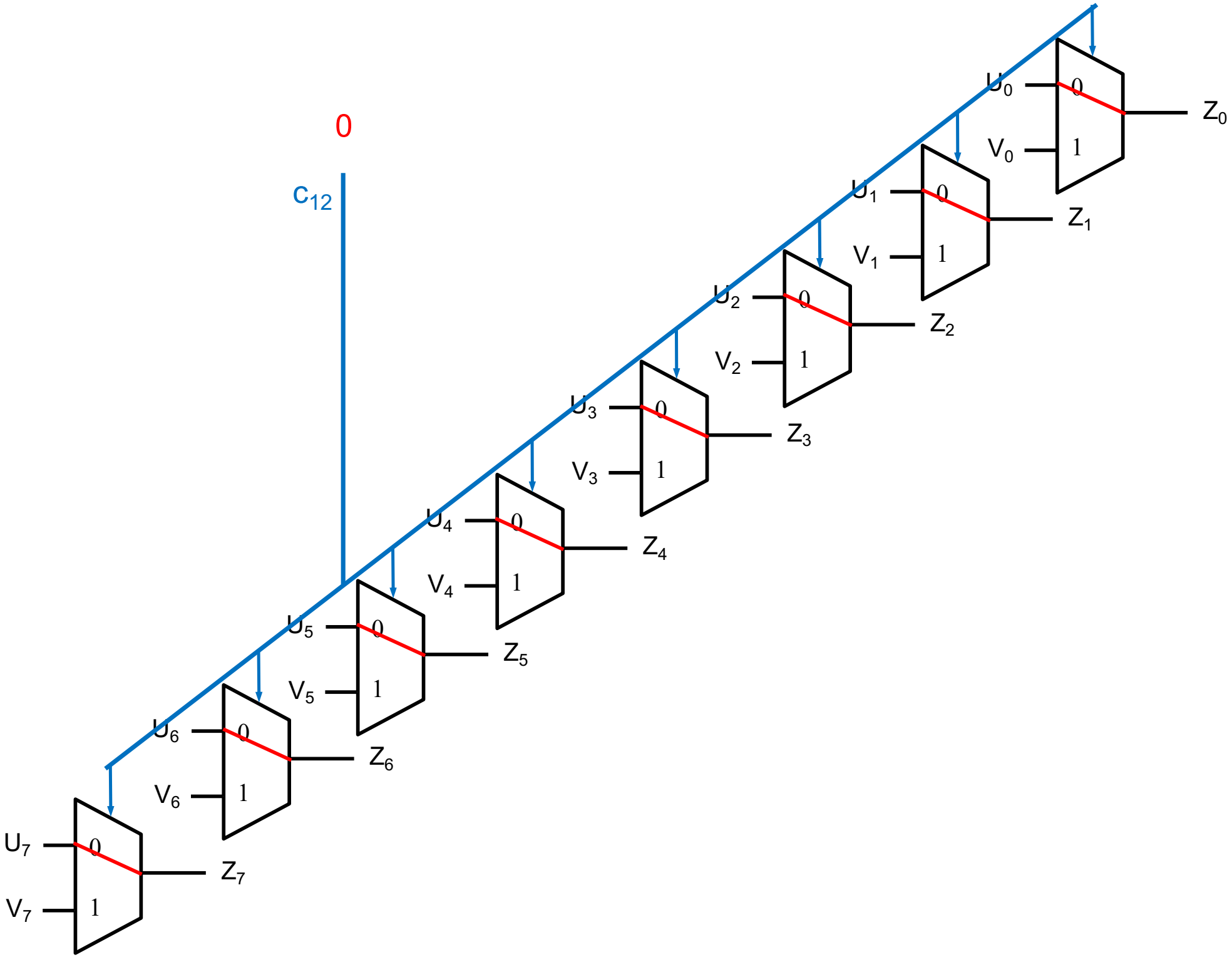
The Internal ALU Bus Multiplexer

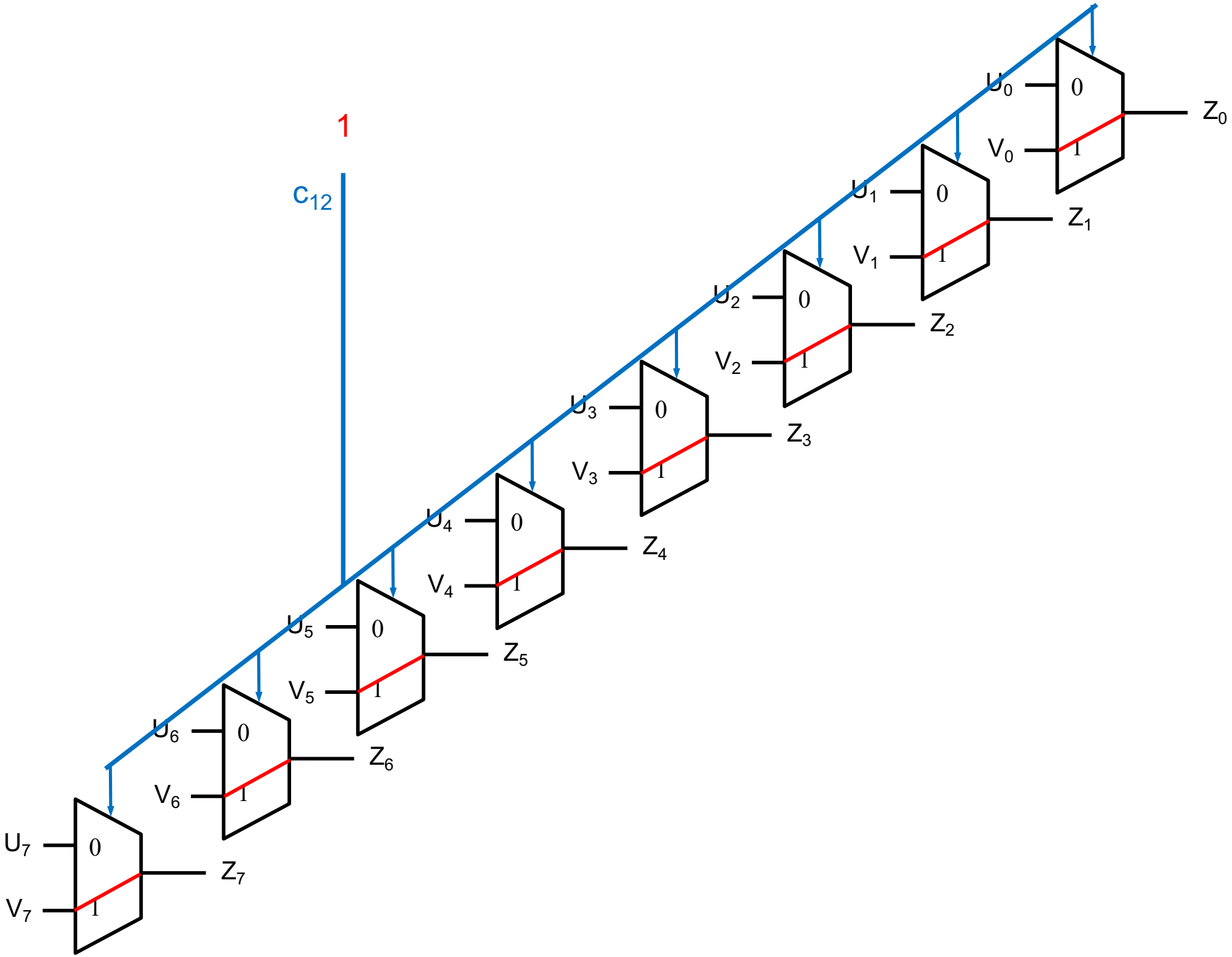


2-to-1 Bus Multiplexer (with 8-bit lines)

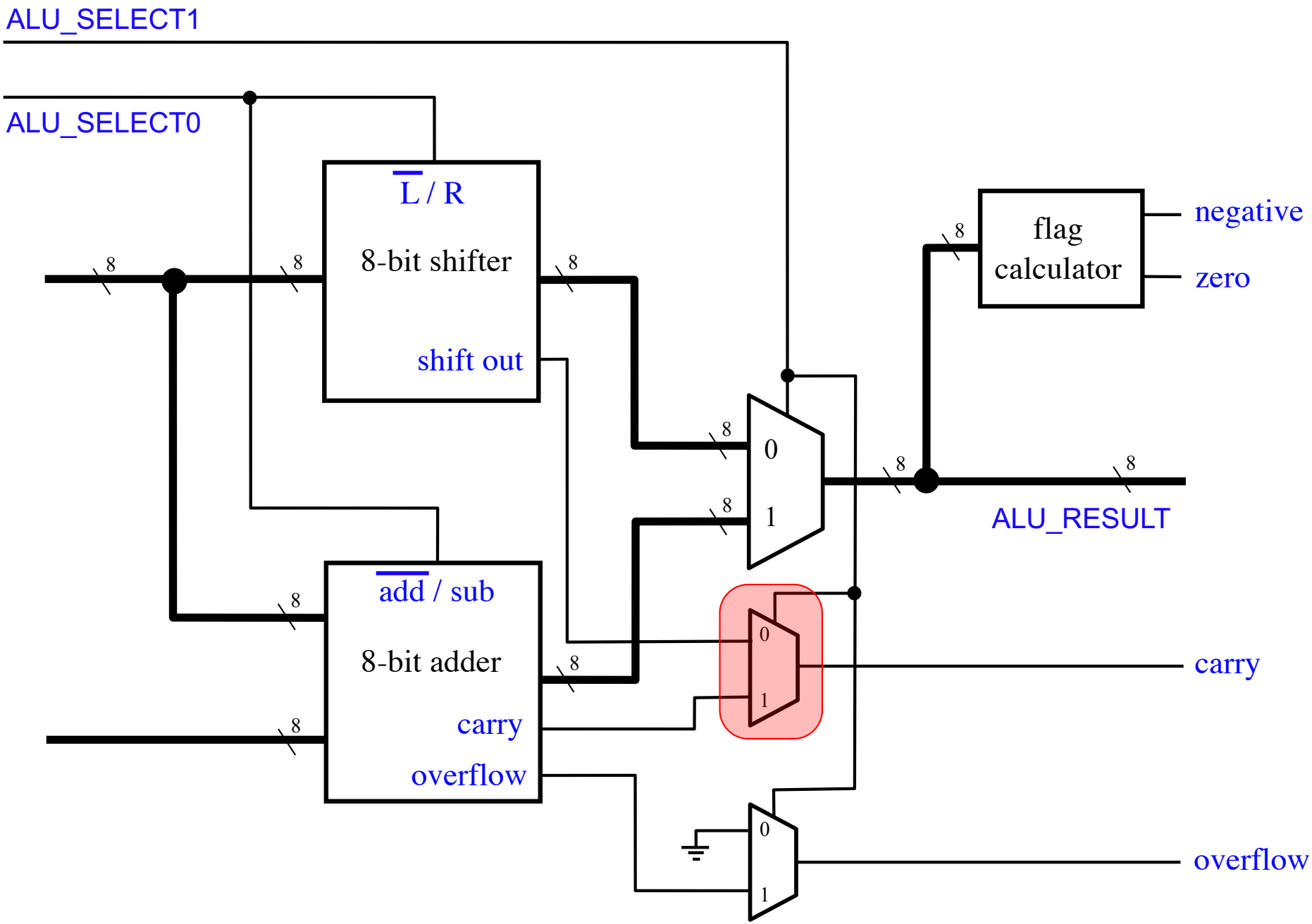



C12
U0
V0
0
1
Z0
U1
V1
Z1
U2
V2
Z2
U3
V3
Z3
U4
V4
Z4
U5
V5
Z5
U6
V6
Z6
U7
V7
Z7

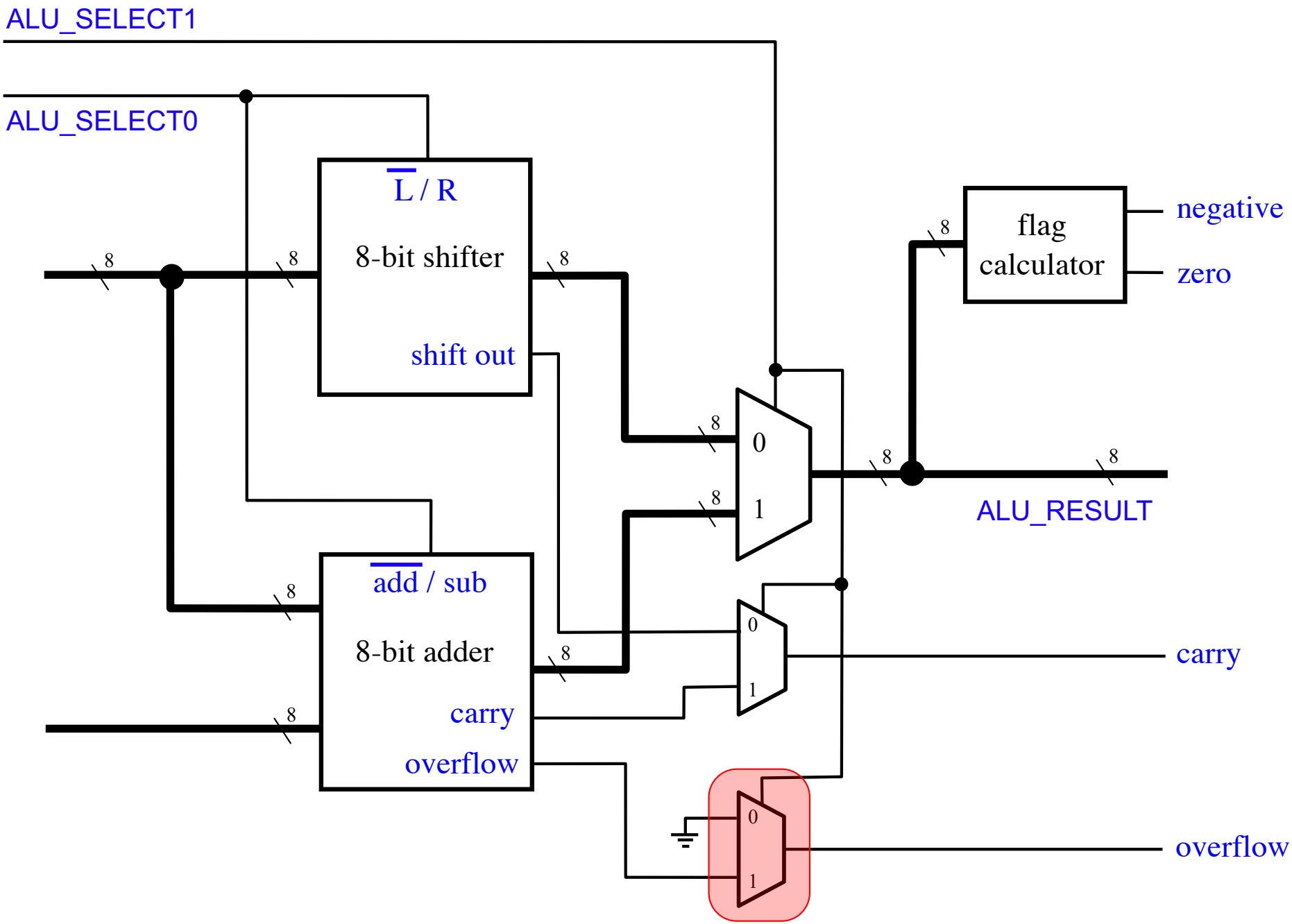

0
c_{12}
U_0
V_0
Z_0
U_1
V_1
Z_1
U_2
V_2
Z_2
U_3
V_3
Z_3
U_4
V_4
Z_4
U_5
V_5
Z_5
U_6
V_6
Z_6
U_7
V_7
Z_7
0
1


1
c_{12}
U_0
V_0
Z_0
U_1
V_1
Z_1
U_2
V_2
Z_2
U_3
V_3
Z_3
U_4
V_4
Z_4
U_5
V_5
Z_5
U_6
V_6
Z_6
U_7
V_7
Z_7
0
1

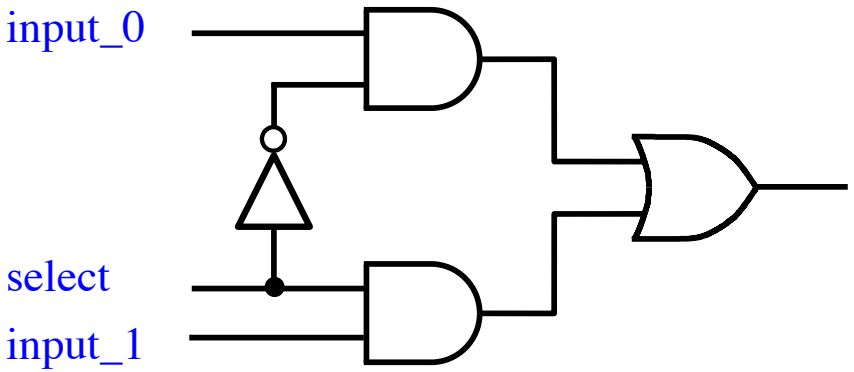
2-to-1 Multiplexer



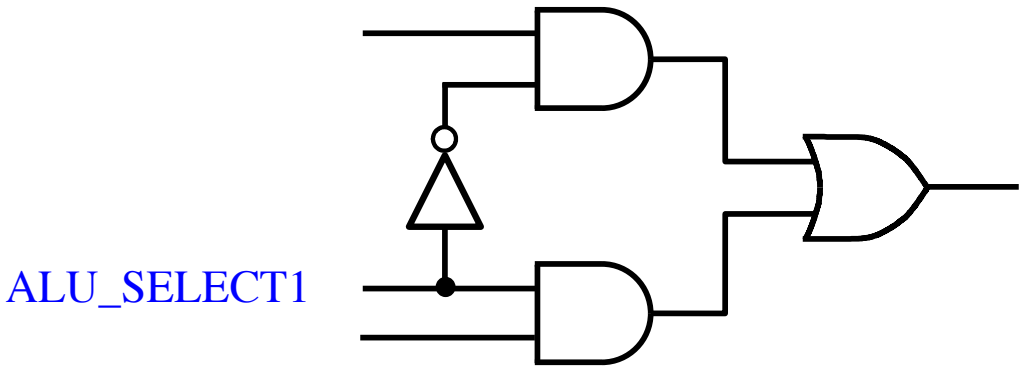
2-to-1 Multiplexer



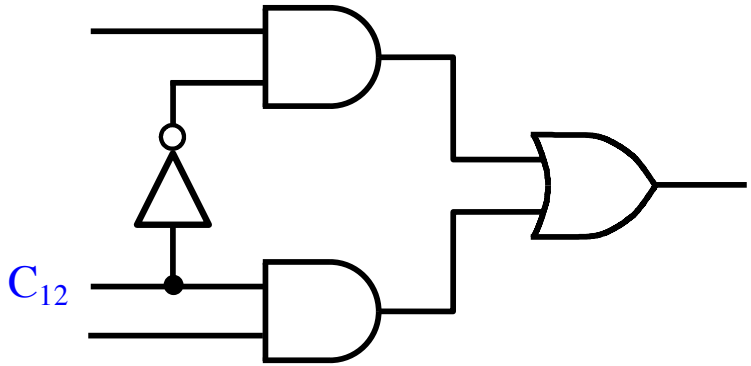
2-to-1 Multiplexer



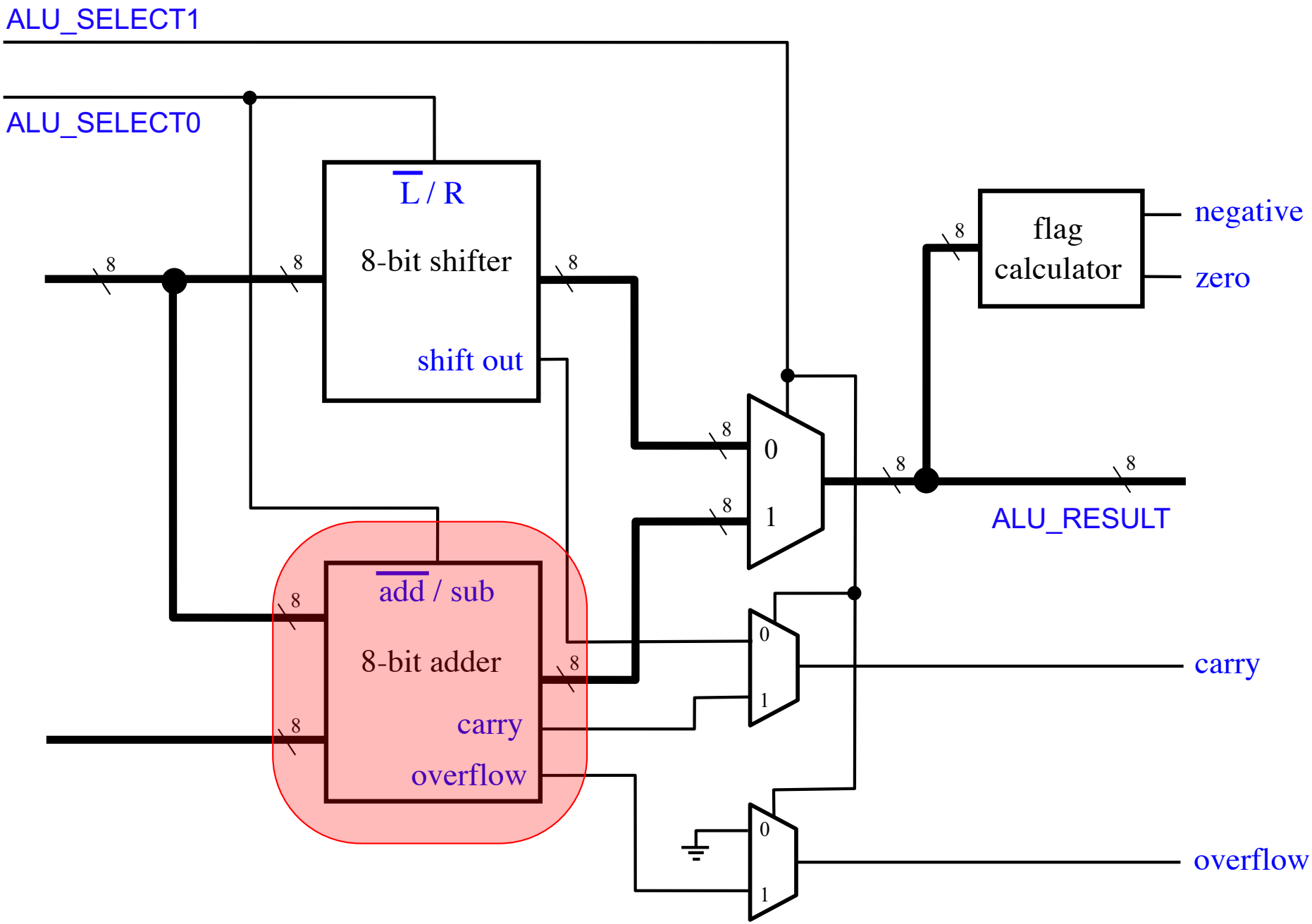
2-to-1 Multiplexer



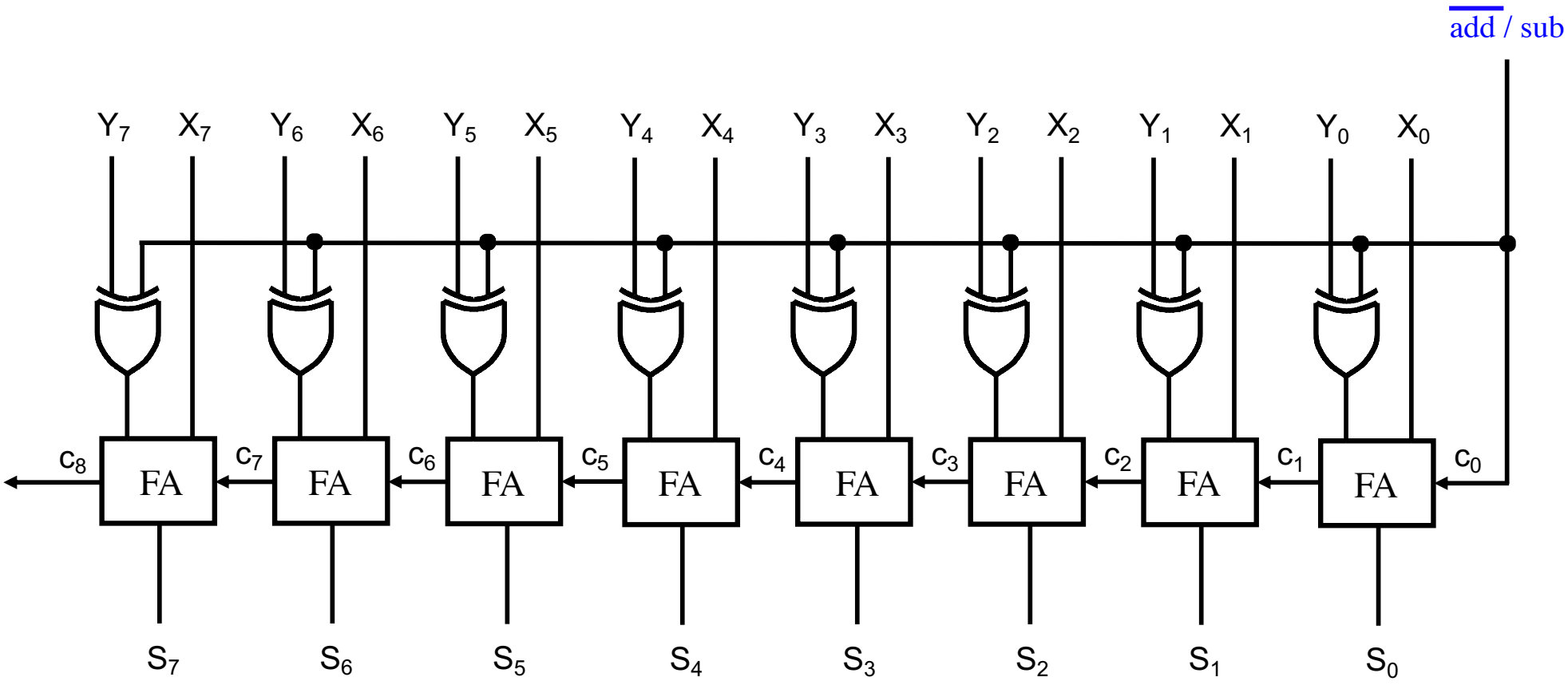
2-to-1 Multiplexer



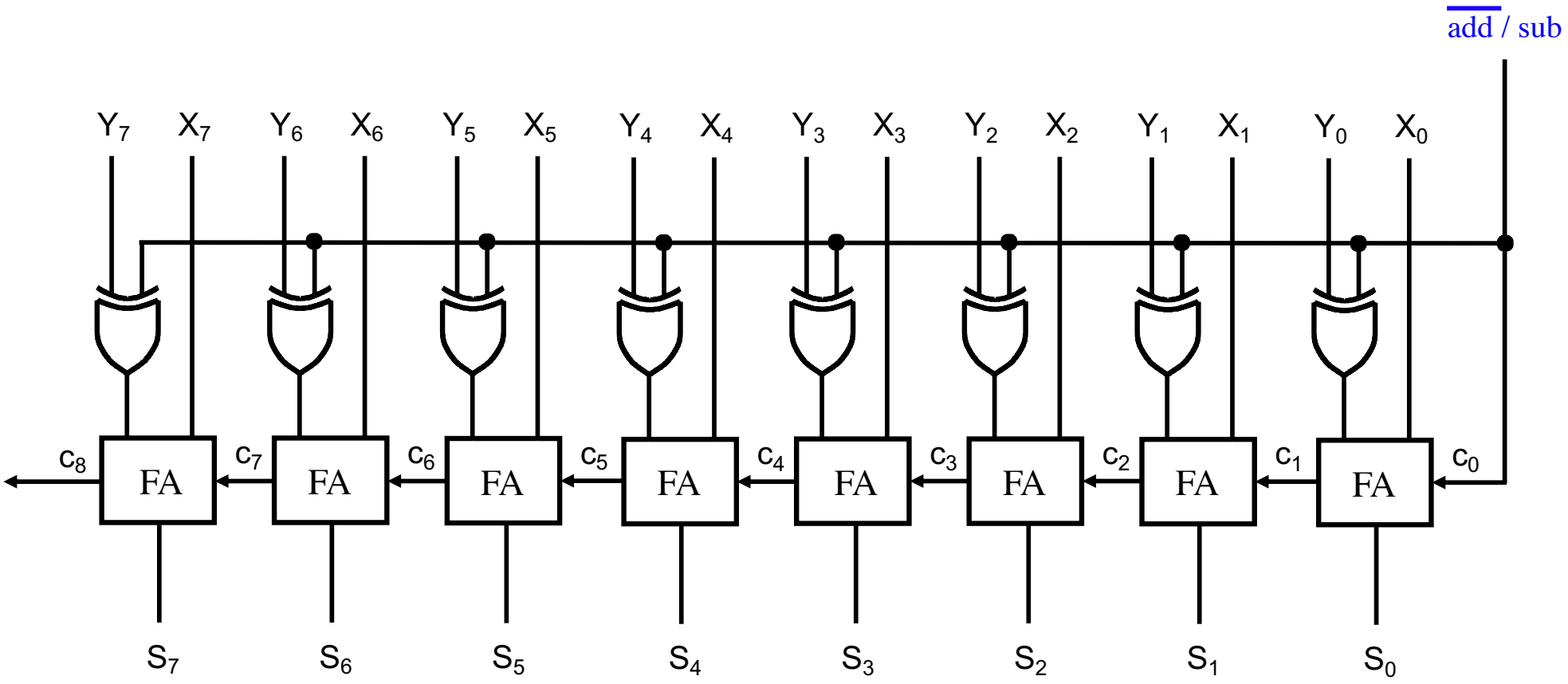
The Adder / Subtractor



The Adder / Subtractor

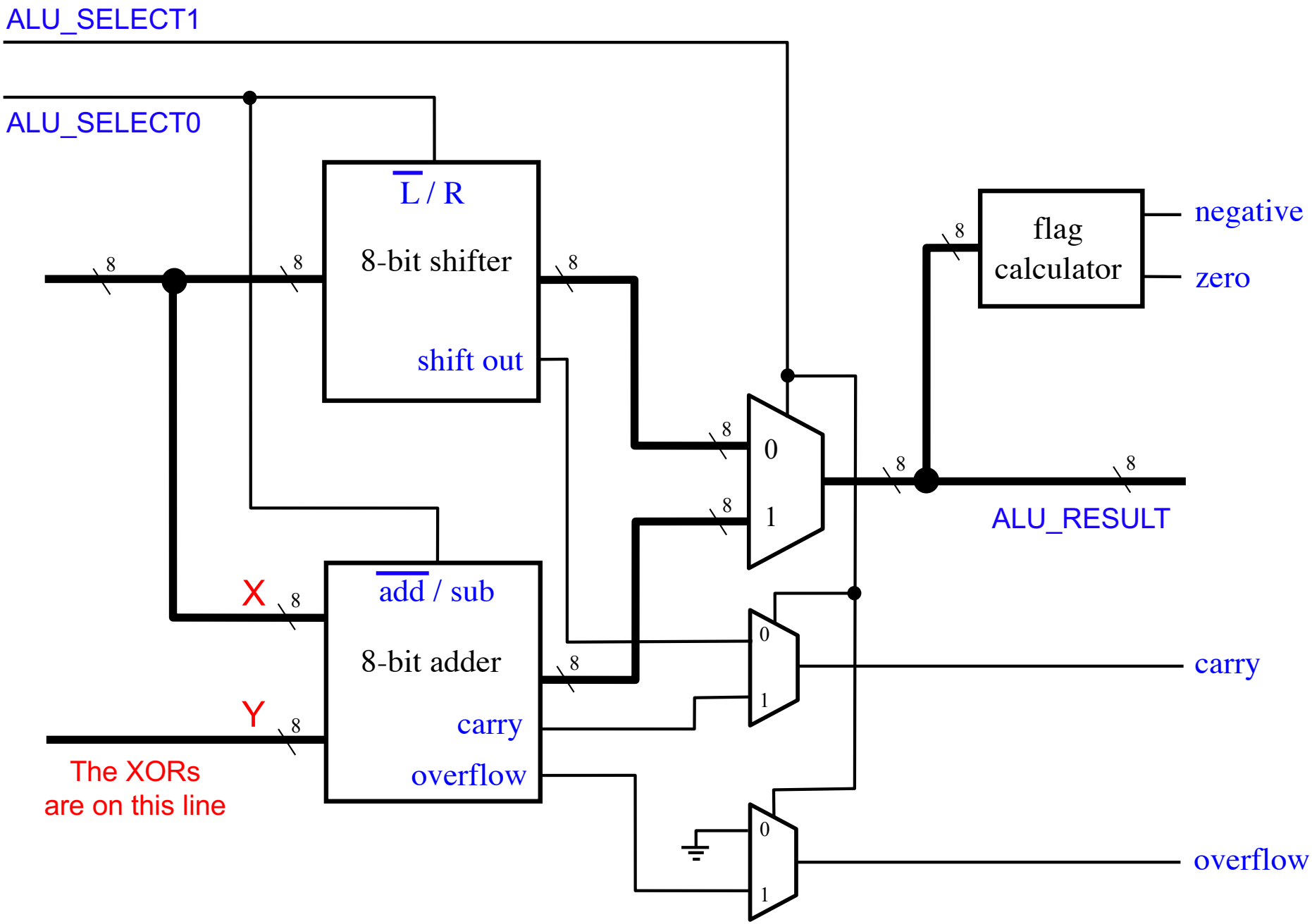


The Adder / Subtractor

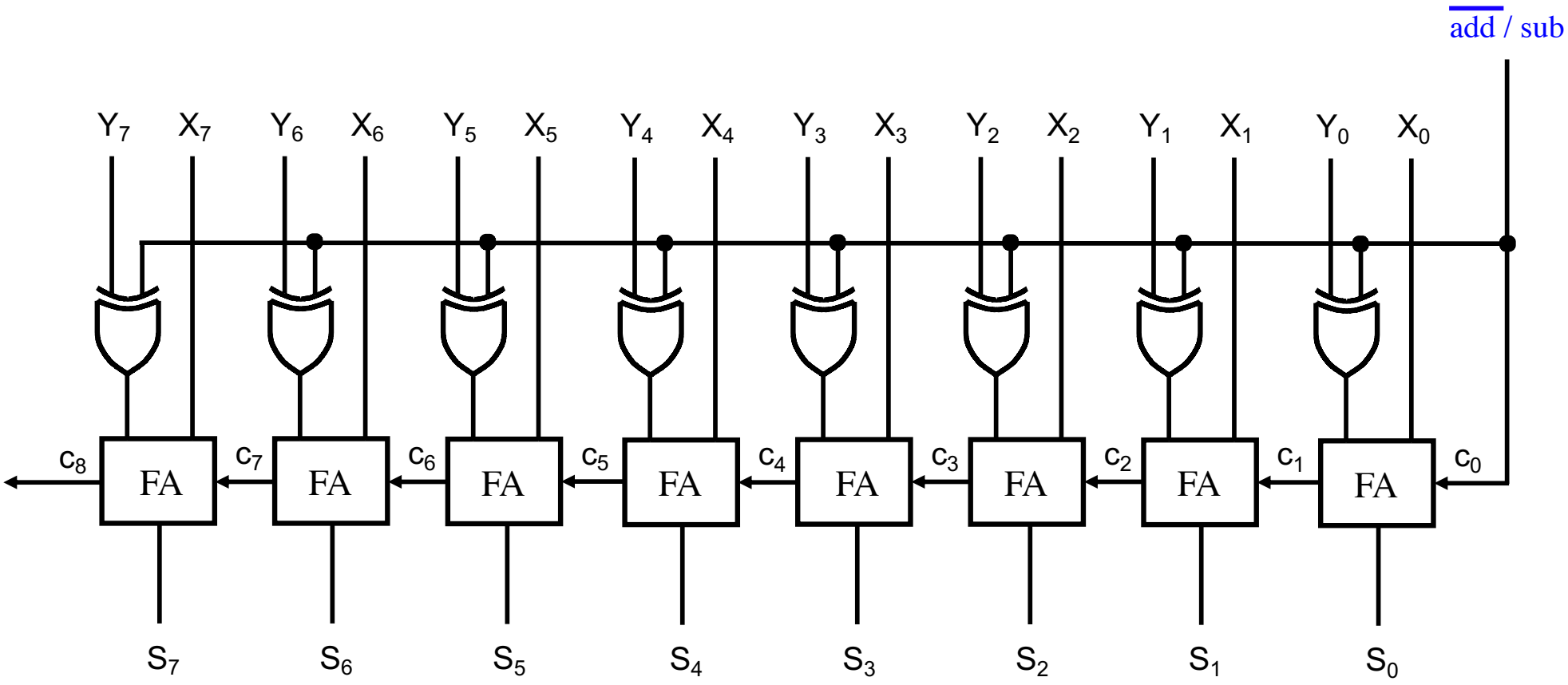


This is an 8-bit ripple-carry adder. Note that the X and Y lines are swapped.

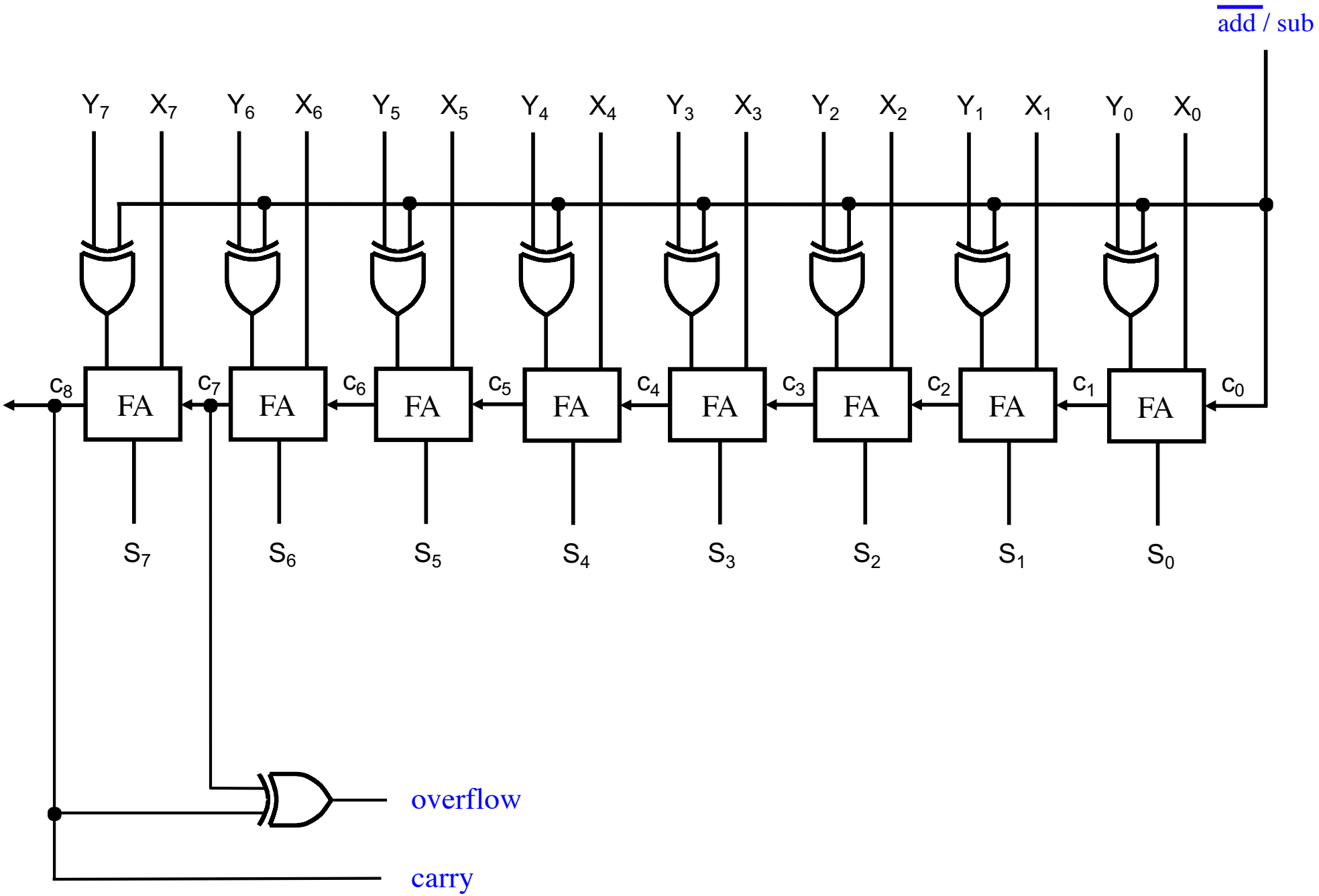
The Adder / Subtractor



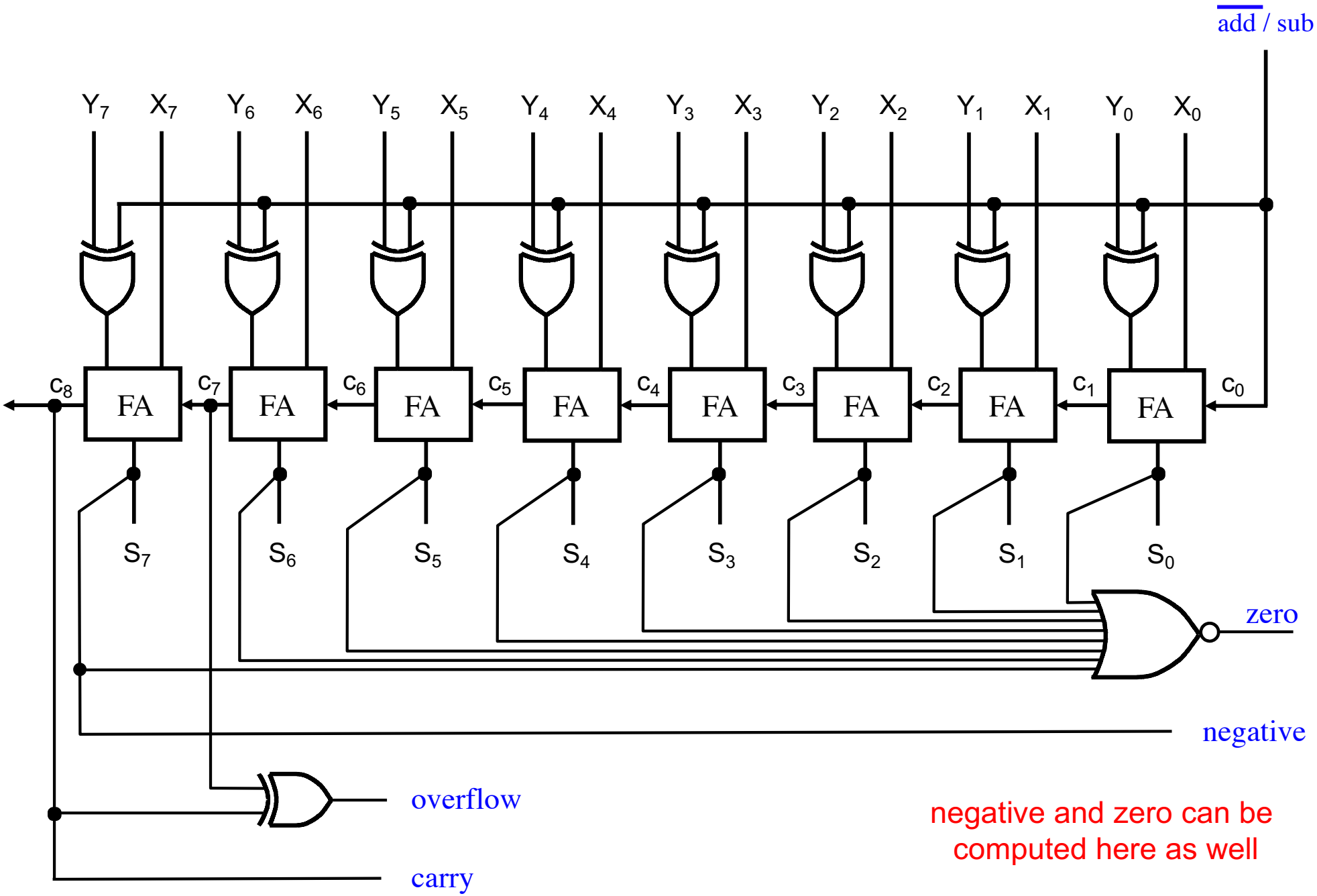
The Adder / Subtractor



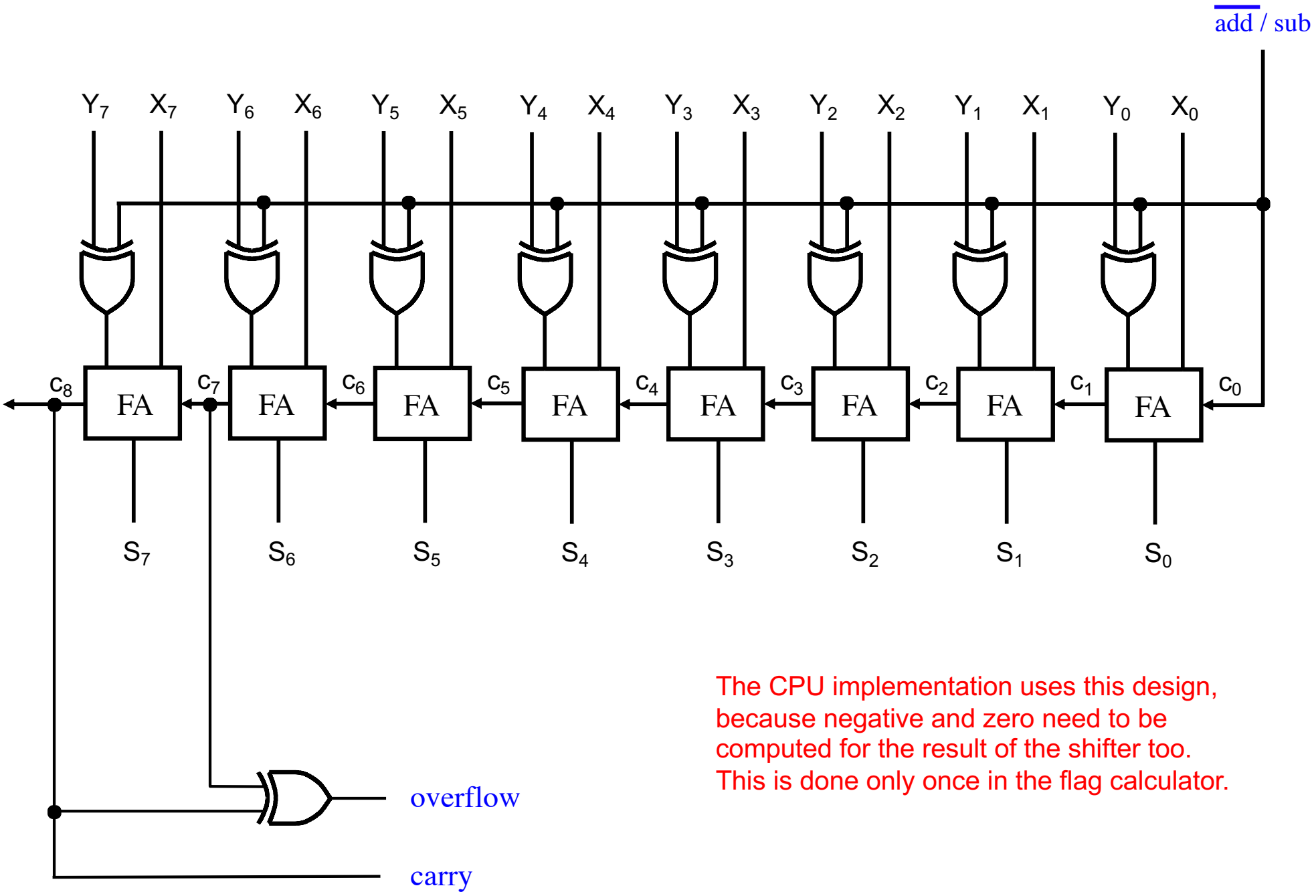
The Adder / Subtractor



The Adder / Subtractor

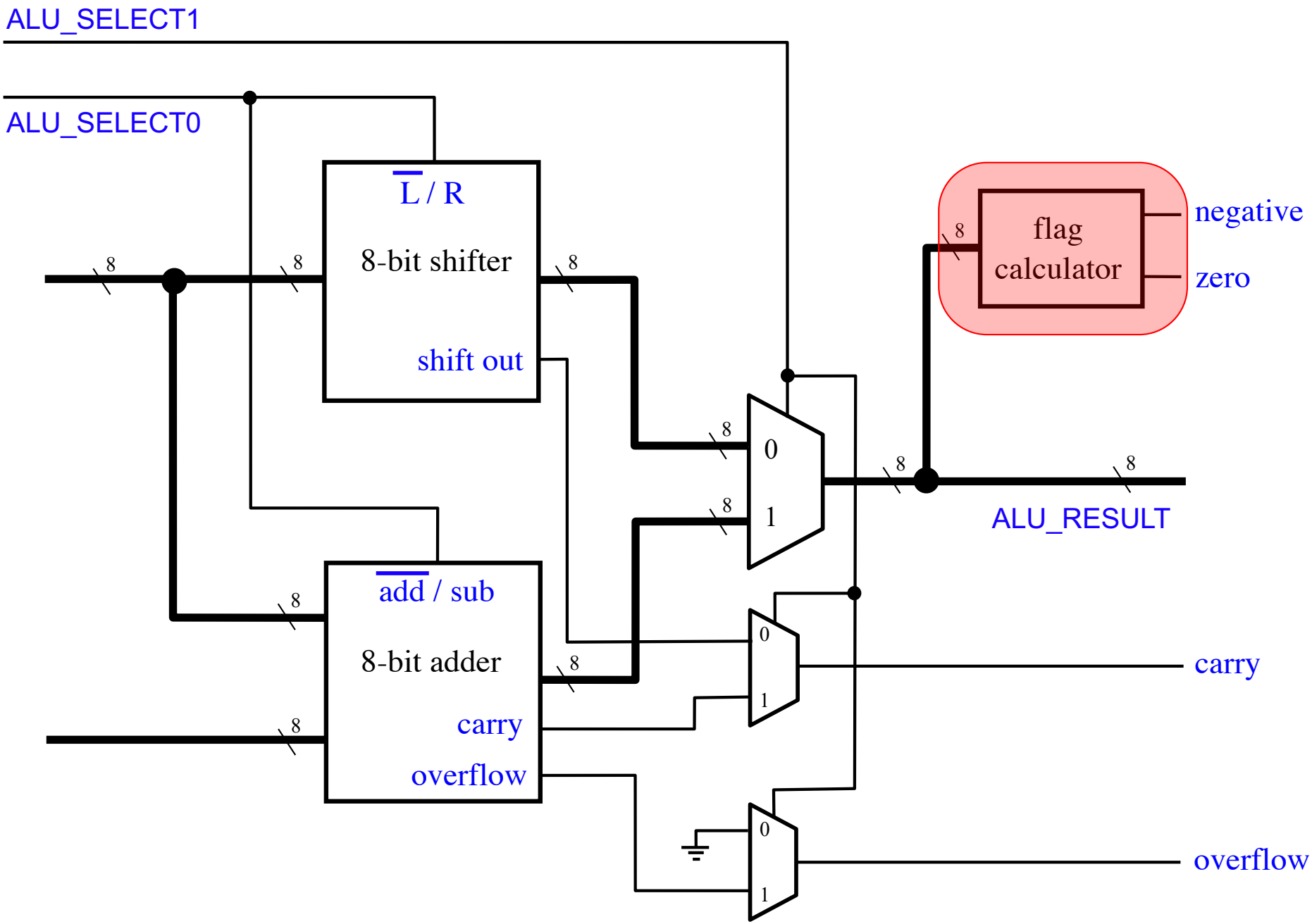


The Adder / Subtractor

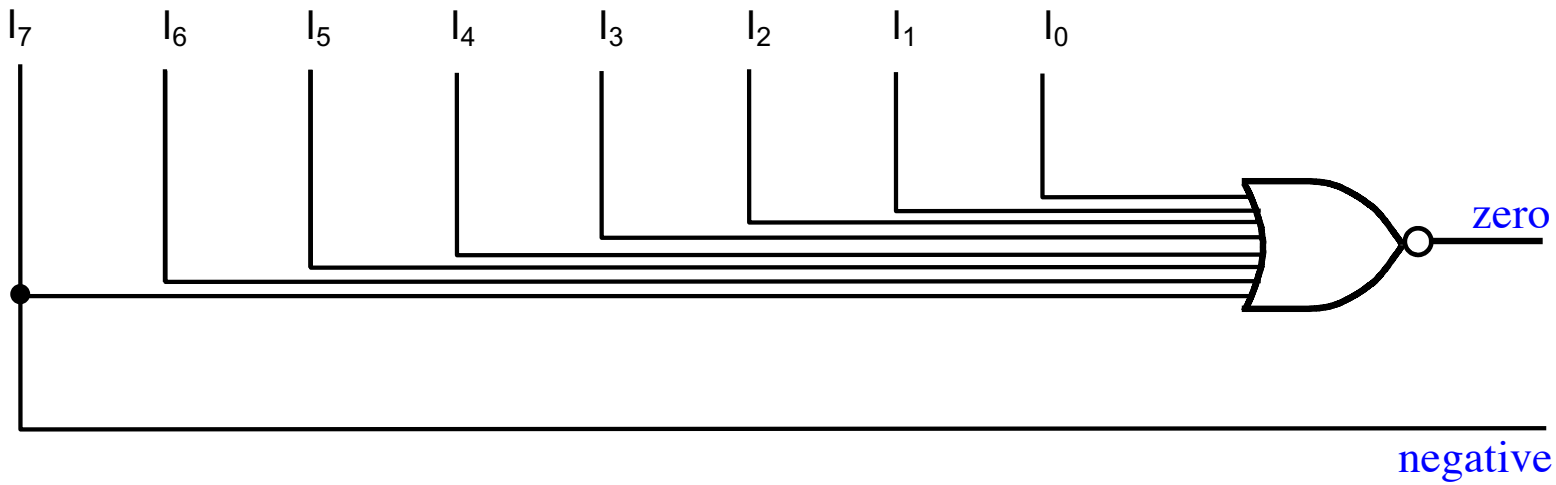


The CPU implementation uses this design, because negative and zero need to be computed for the result of the shifter too. This is done only once in the flag calculator.

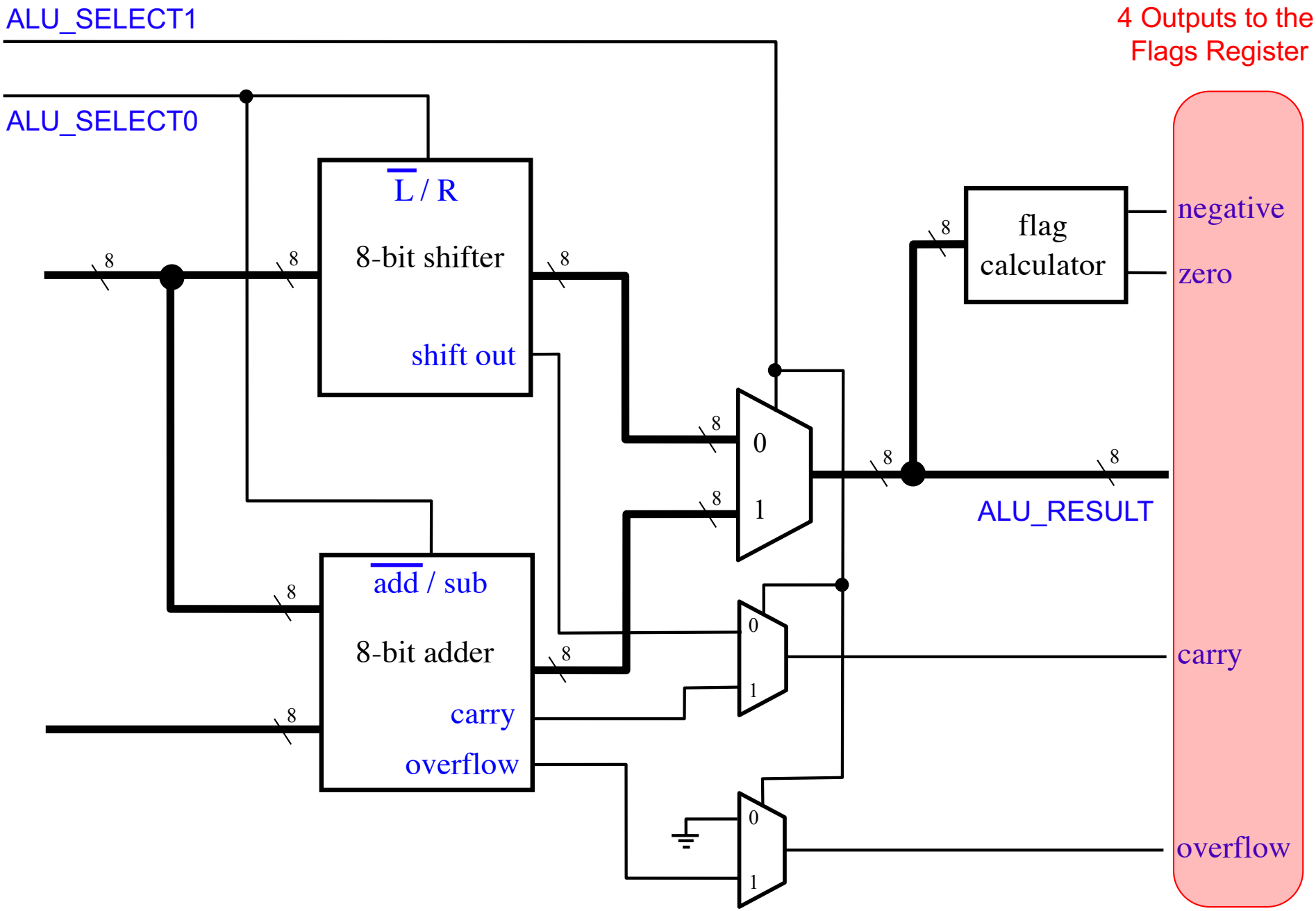
The ALU Flag Calculator



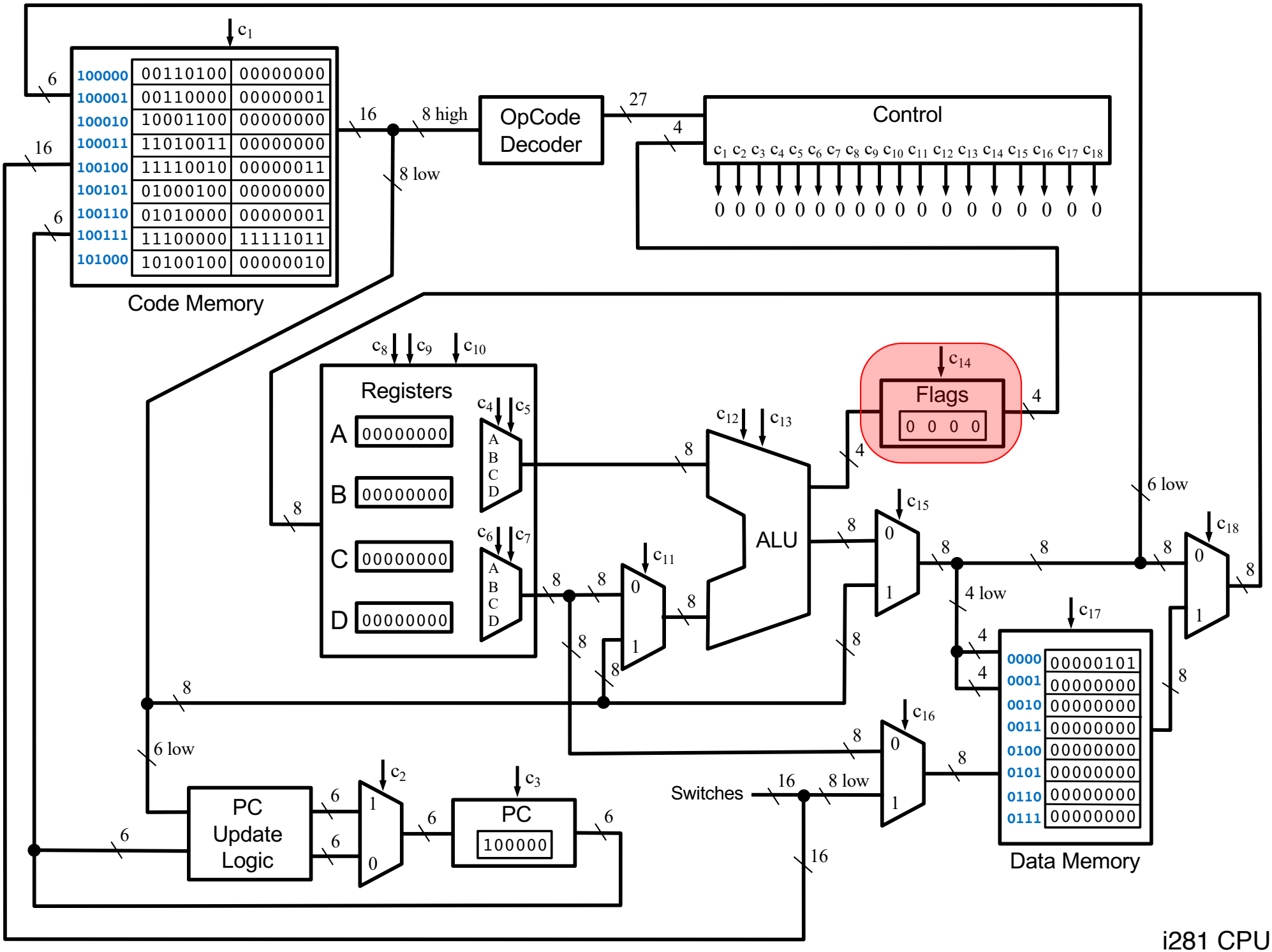
The ALU Flag Calculator



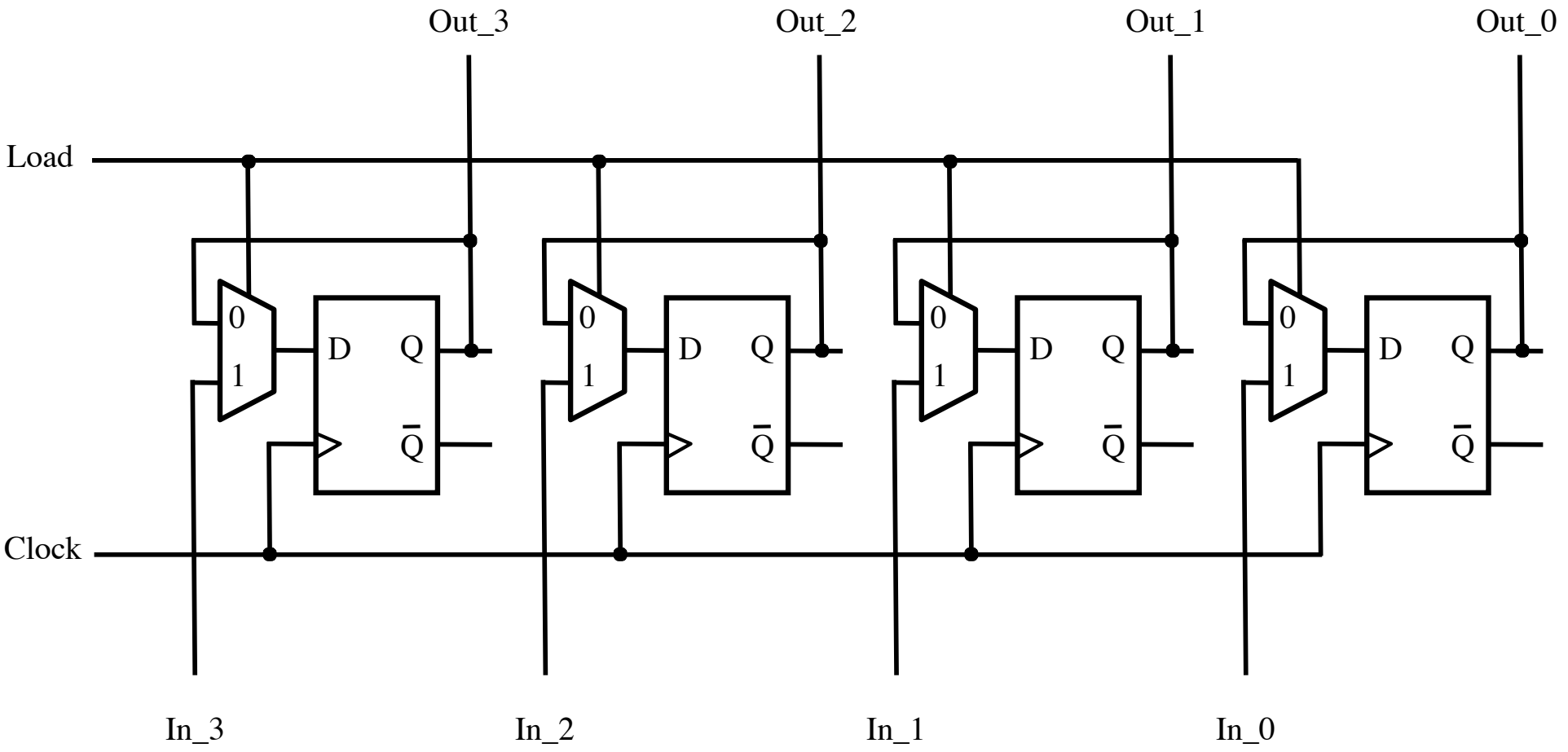
ALU Outputs to the Flags Register



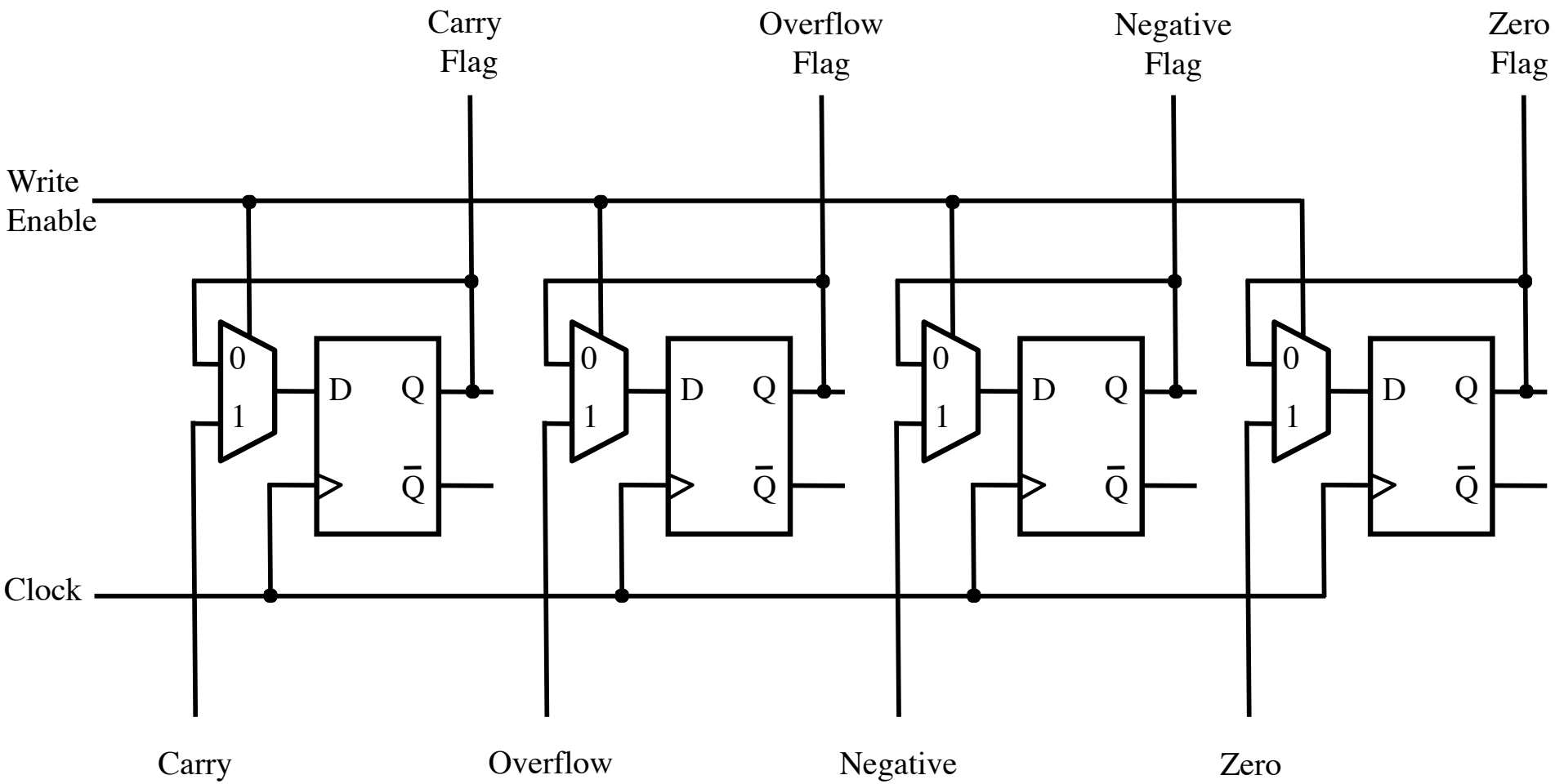
The Flags Register


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c14
Flags
0 0 0 0
4
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c11
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
16
8 low
6 low
c2
c3
PC Update Logic
PC
100000
i281 CPU

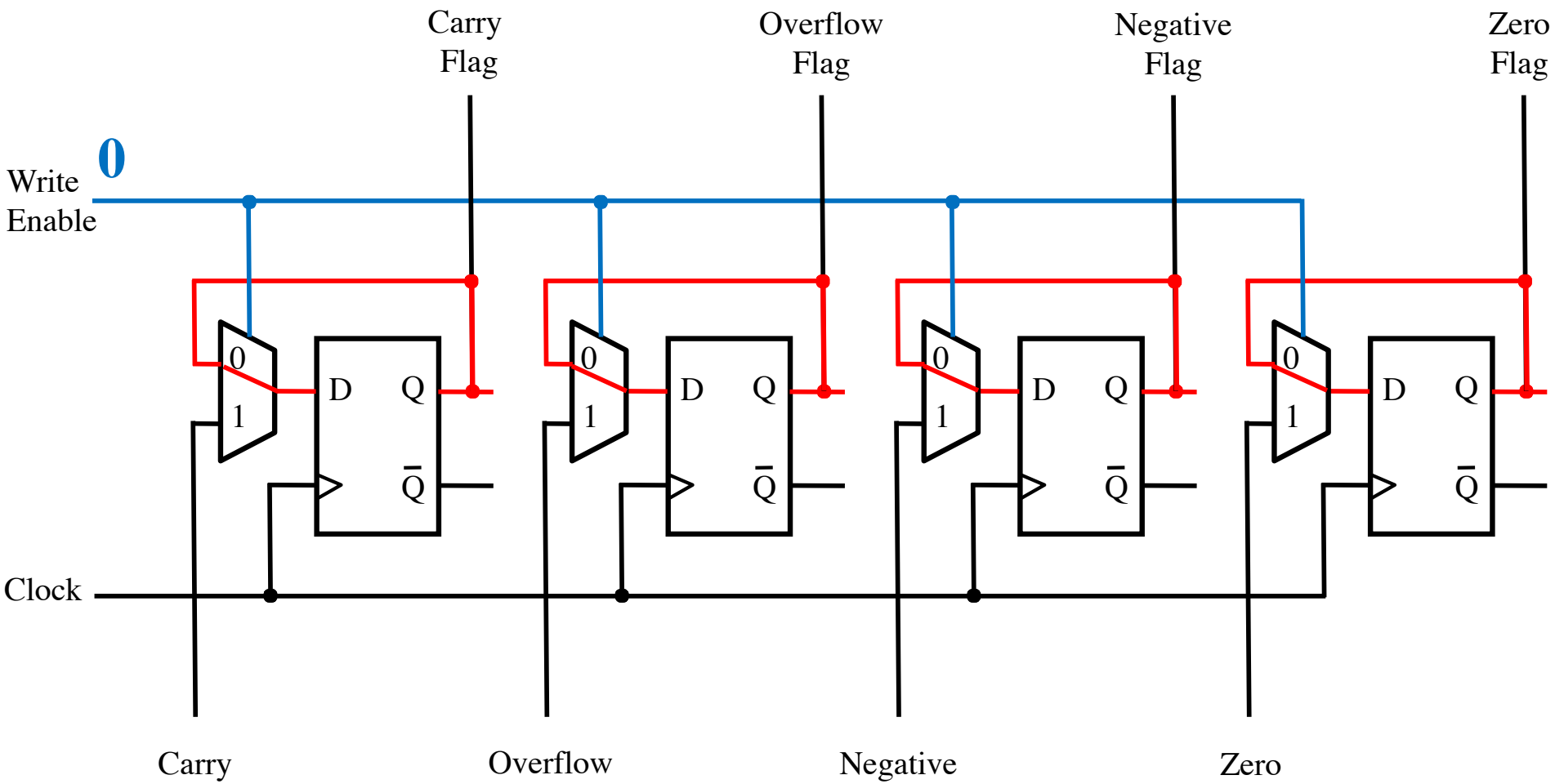
4-Bit Parallel-Access Register



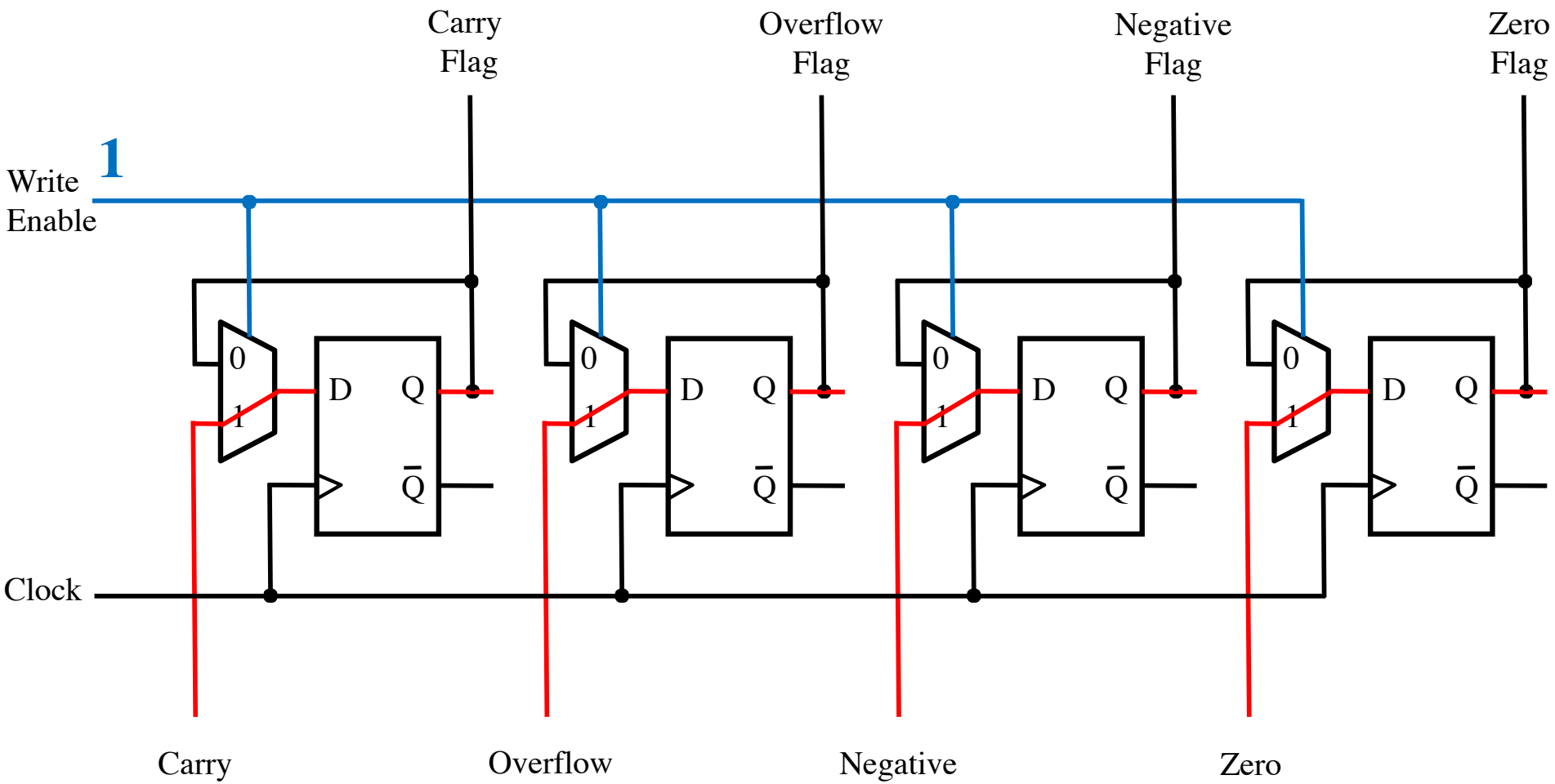
The Flags Register



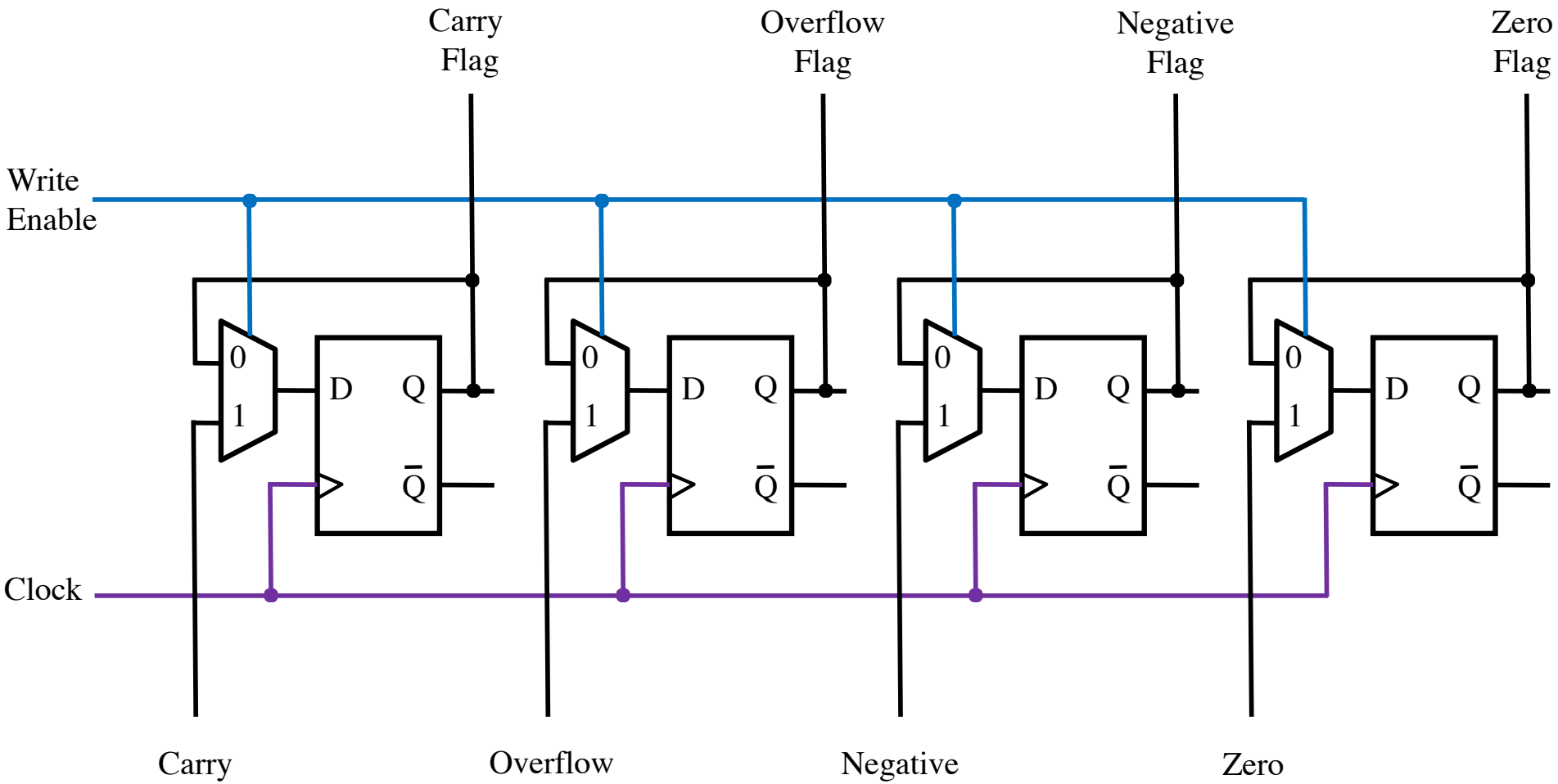
The Flags Register



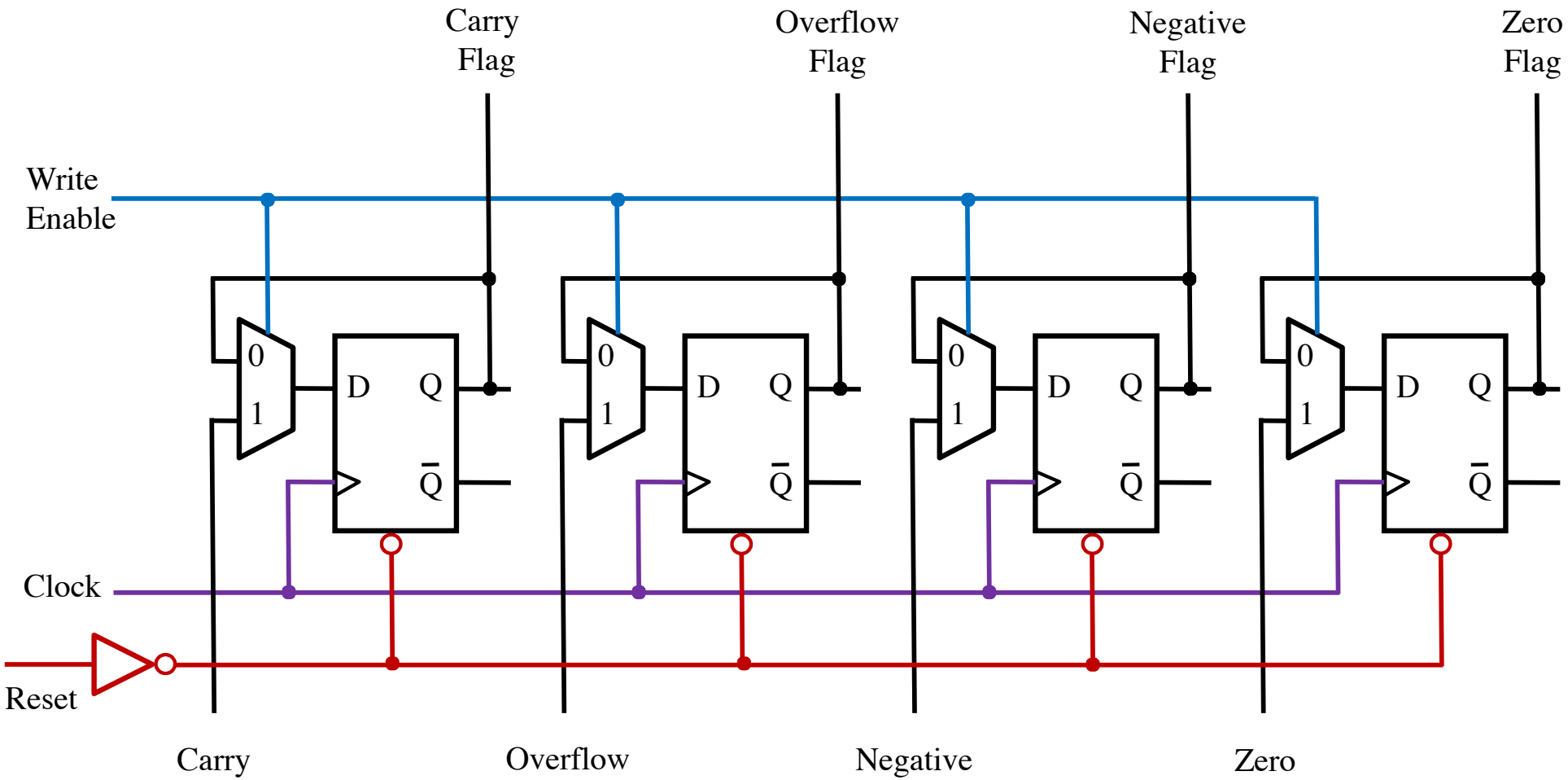
The Flags Register



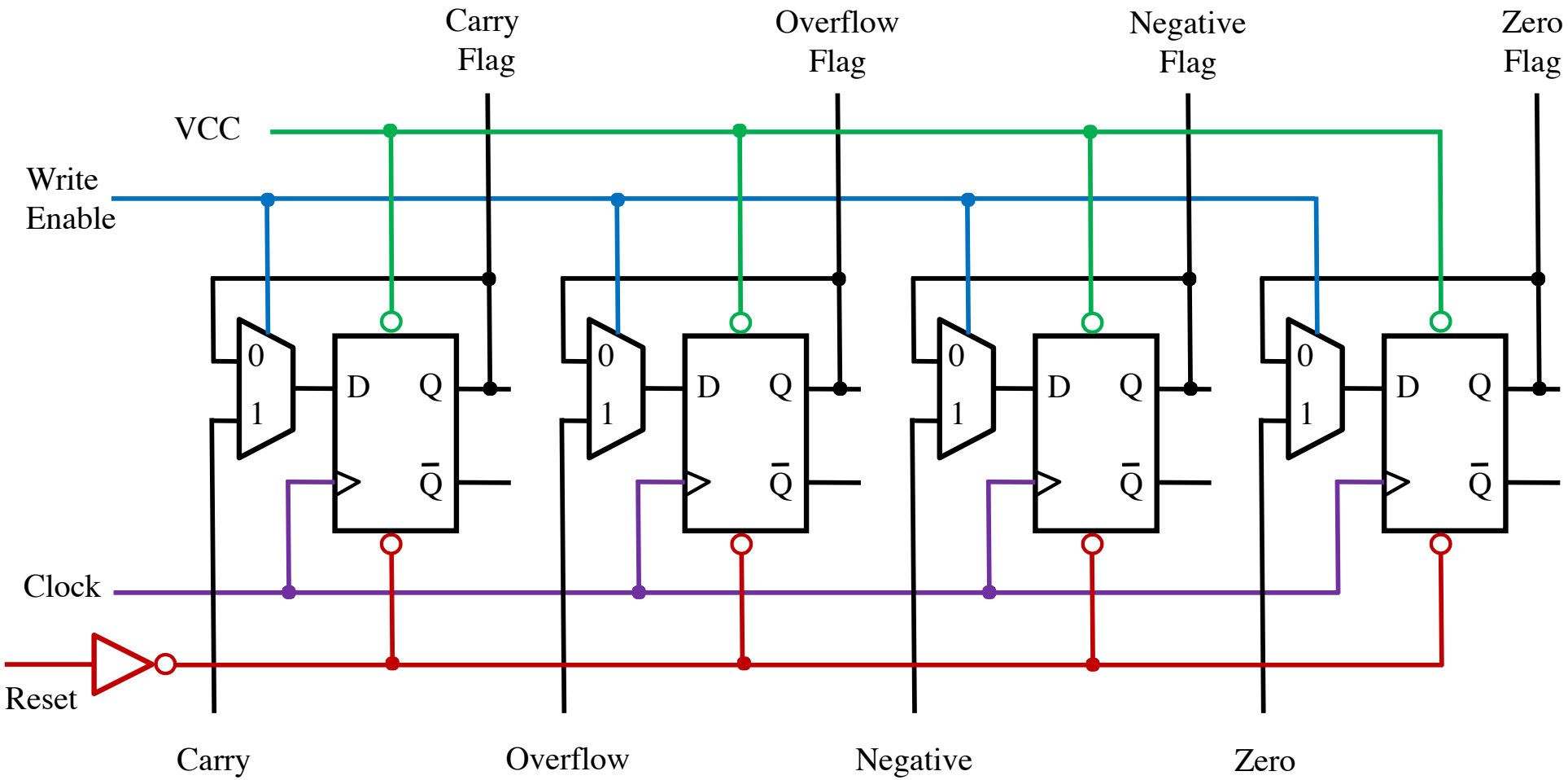
The Flags Register



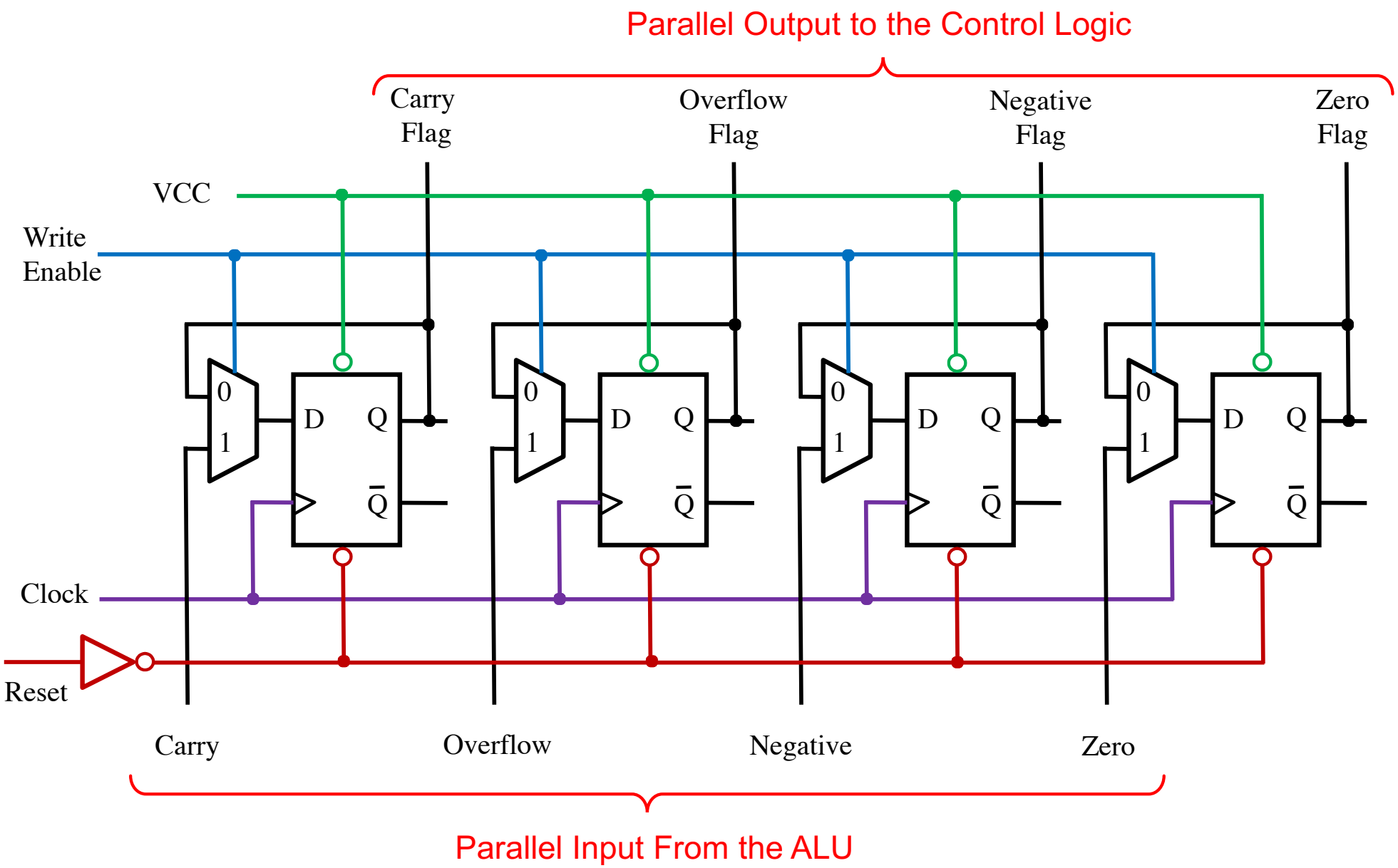
The Flags Register



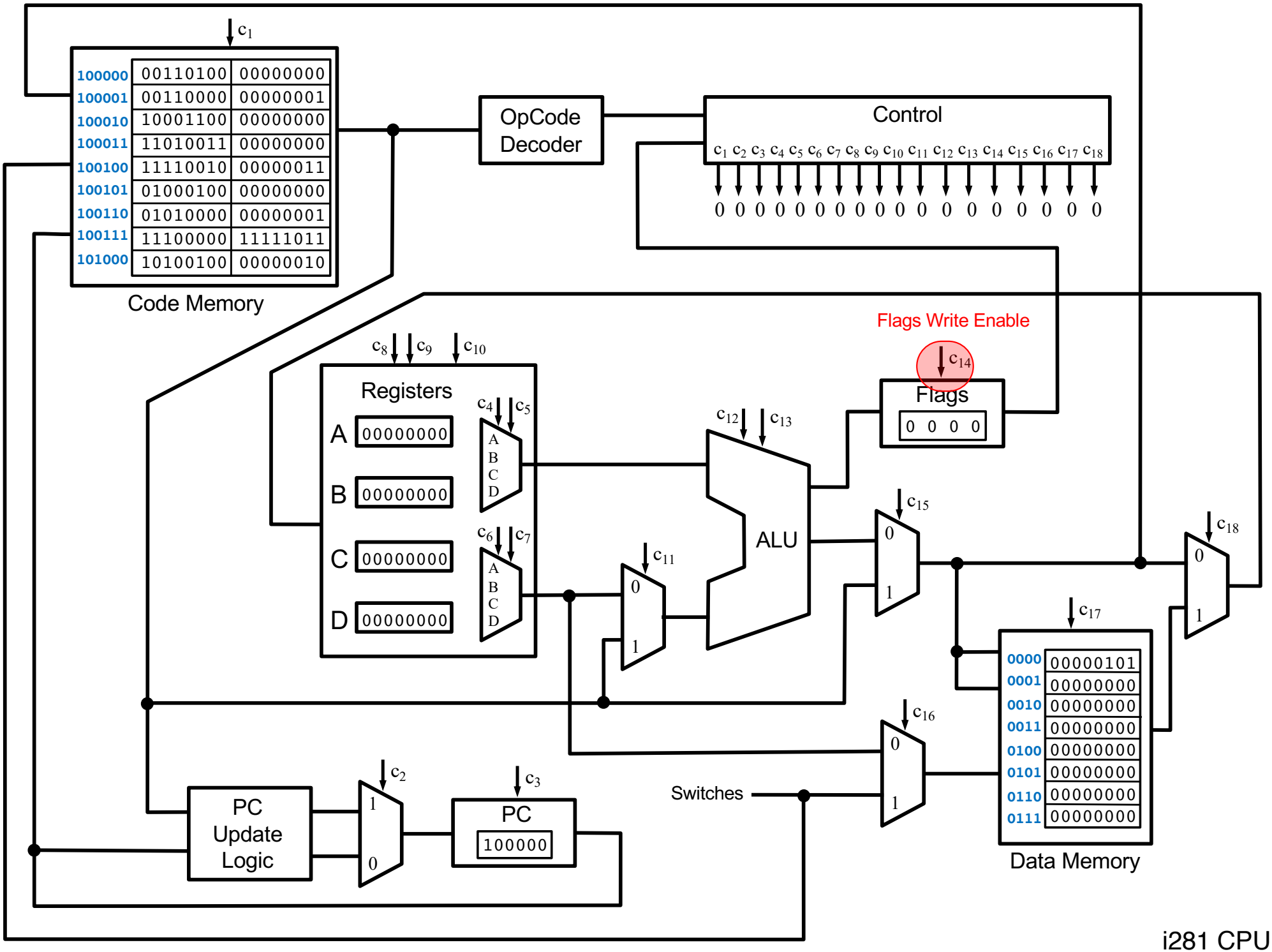
The Flags Register



The Flags Register



Only Seven OPCODEs Update the Flags Register


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Flags Write Enable
c14
Flags
0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c11
c15
c18
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
c2
c3
PC Update Logic
PC
100000
i281 CPU

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

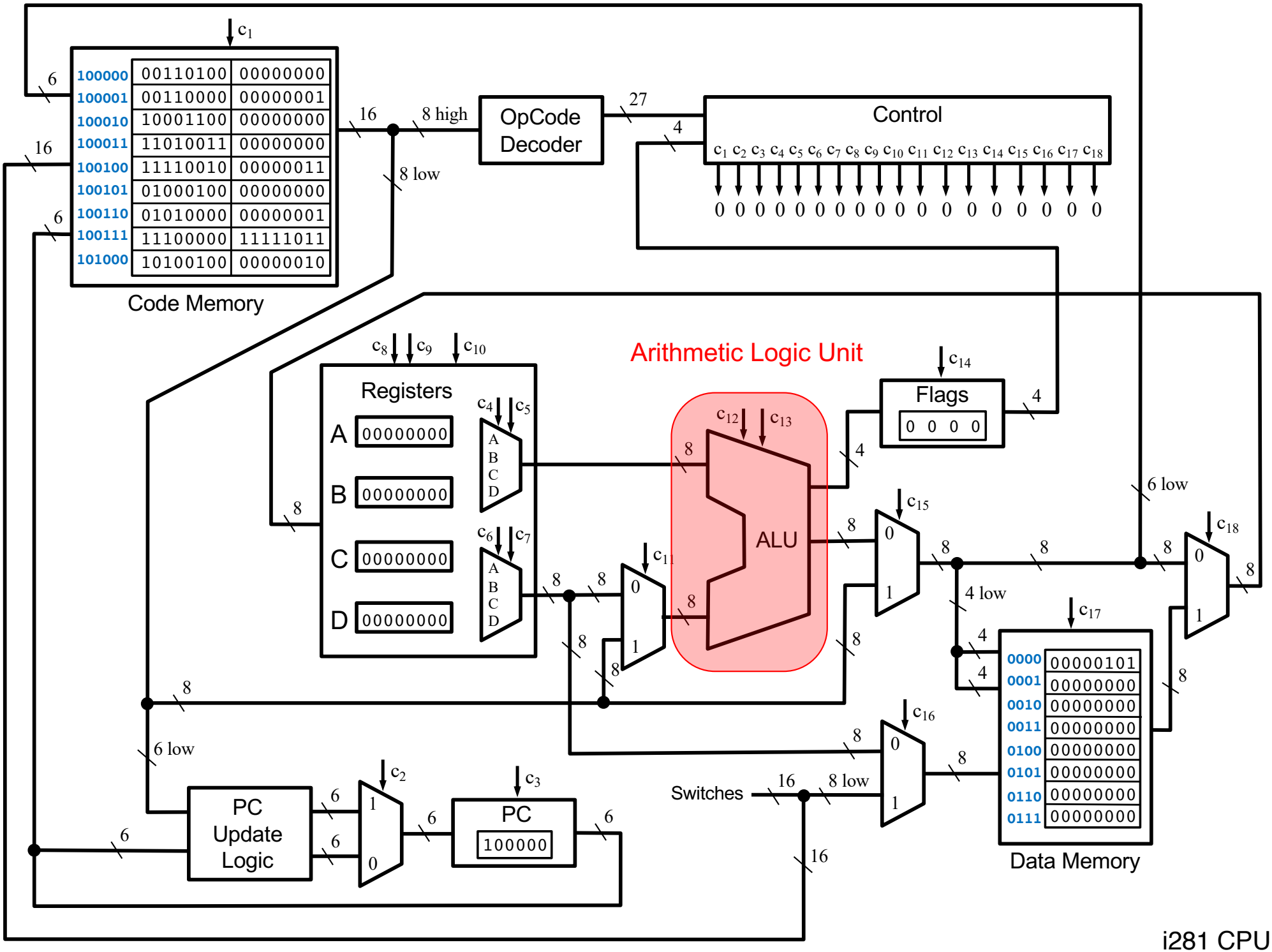
c_{14} is set to 1 for opcodes that update the flags register. Only 7 opcodes do that.

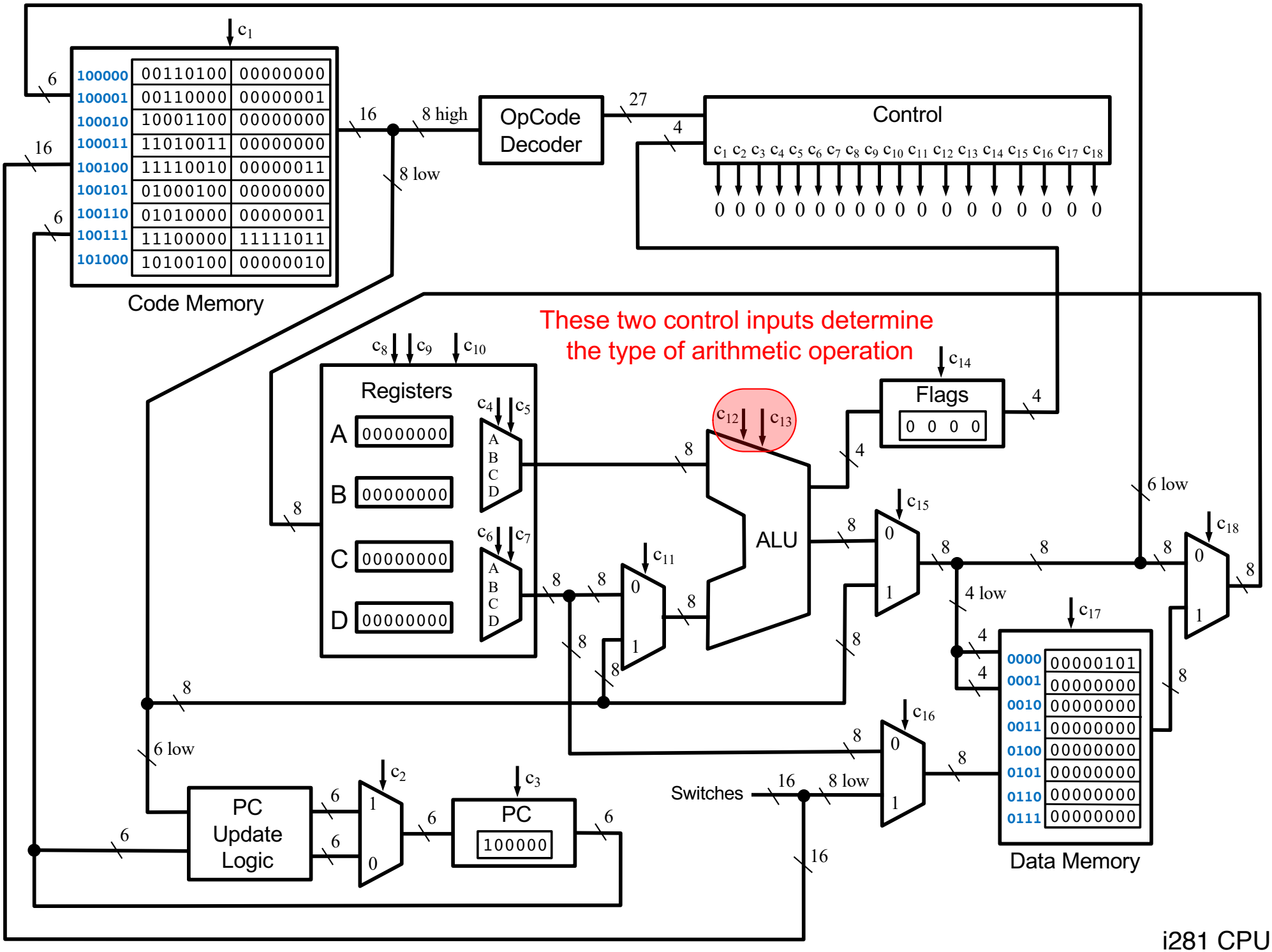
	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

c_{14} is set to 1 for opcodes that update the flags register. Only 7 opcodes do that.

These 7 OPCODEs Update the Flags

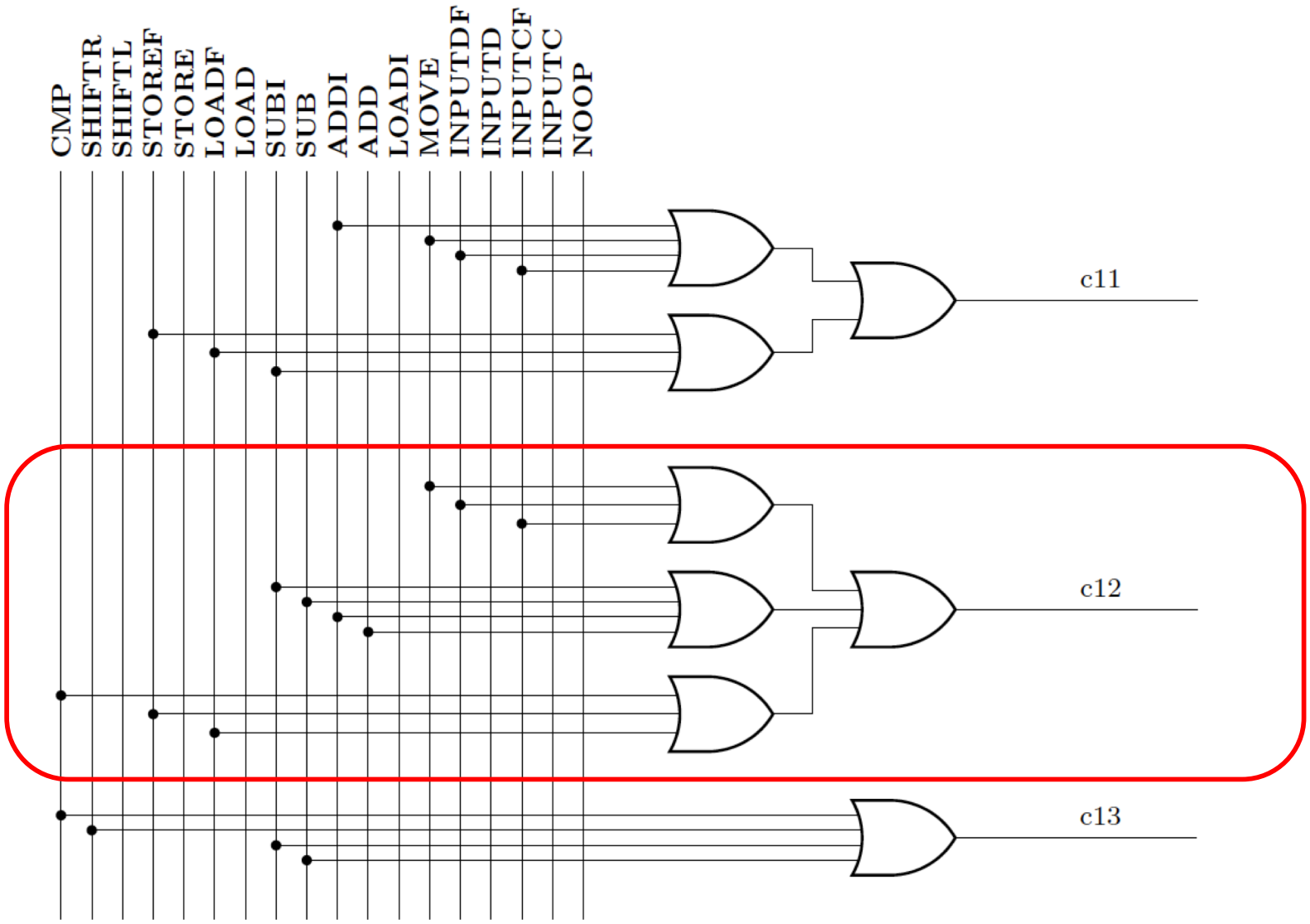
NOOP	NO OPeration
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
8 high
8 low
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Arithmetic Logic Unit
Registers
A 00000000
B 00000000
C 00000000
D 00000000
ALU
Flags
0 0 0 0
6 low
4 low
PC Update Logic
PC
100000
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
These two control inputs determine
the type of arithmetic operation
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c11
c15
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
8 low
c2
c3
PC Update Logic
PC
100000
i281 CPU

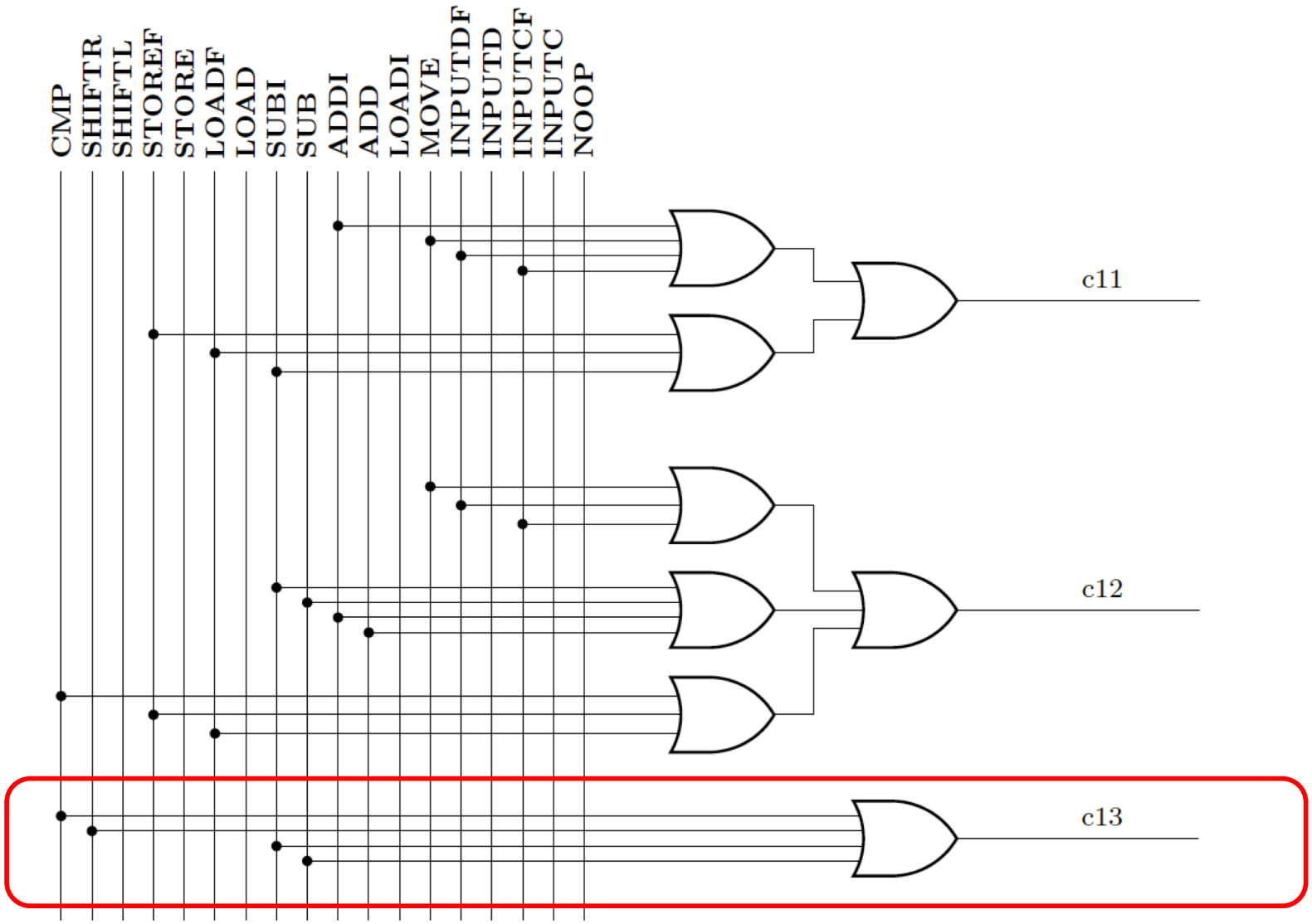
	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

The Wiring Diagram for c_{12}



	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

The Wiring Diagram for c_{13}



This ALU Can Perform 4 Operations

ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

This ALU Can Perform 4 Operations

ALU_SELECT1	ALU_SELECT0	Operation	
0	0	SHIFTL	
0	1	SHIFTR	
1	0	ADD	ADDI
1	1	SUB/CMP	SUBI

These two OpCodes map to here as well

This ALU Can Perform 4 Operations

Names of these control lines

C_{12} C_{13}

ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

This ALU Can Perform 4 Operations

C_{12}	C_{13}	
ALU_SELECT1	ALU_SELECT0	Operation
0	0	SHIFTL
0	1	SHIFTR
1	0	ADD
1	1	SUB/CMP

Both SUB and CMP are implemented as subtraction. They both set the flags.

The difference is that CMP does not write back the result of the subtraction to the registers. Only the side effect through the flags remains.

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

c_{12} and c_{13} are set by these opcodes that need the ALU to compute something.

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1		0	0	1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

SHIFTL also uses the ALU, but in this case the two control lines are both equal to zero.

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1		0	0	1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

In fact, all of these other commands tell
the ALU to shift left. The ALU is never idle!
But they ignore the result!

The MOVE command
is implemented as addition with 0

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

The MOVE command copies the contents of one register into another. But it tells the ALU to do addition?! Why?

MOVE

Full name:

MOVE

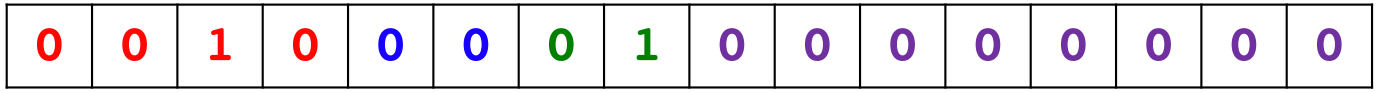
Description:

Move (i.e., copy) the contents of the first register into the second register, overwriting the second register.

Assembly Example:

```
MOVE  A, B
```

Instruction Layout:



MOVE

Full name:

MOVE

Description:

Move (i.e., copy) the contents of the first register into the second register, overwriting the second register.

Assembly Example:

MOVE A, B

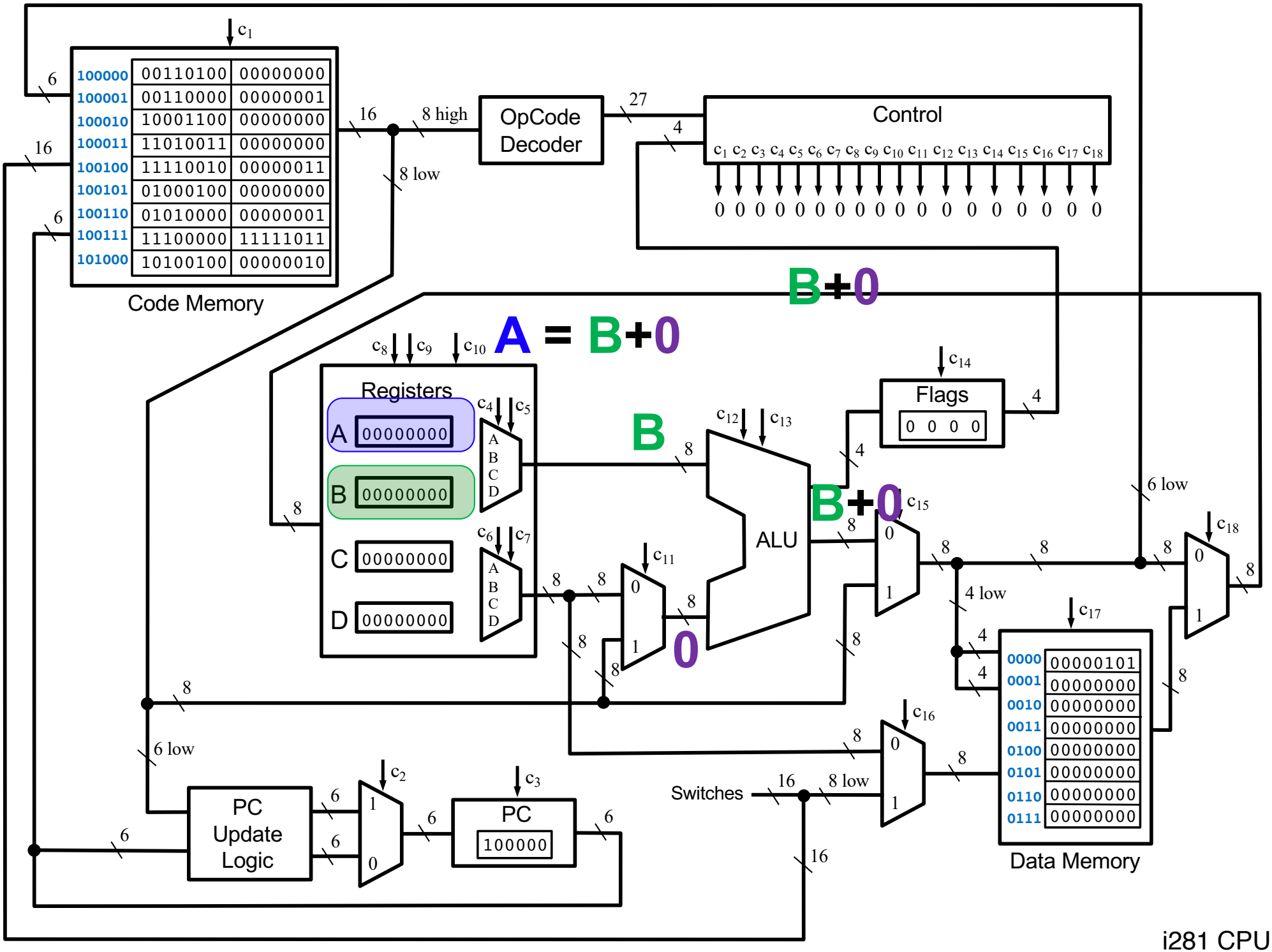
Instruction Layout:

A = B + 0

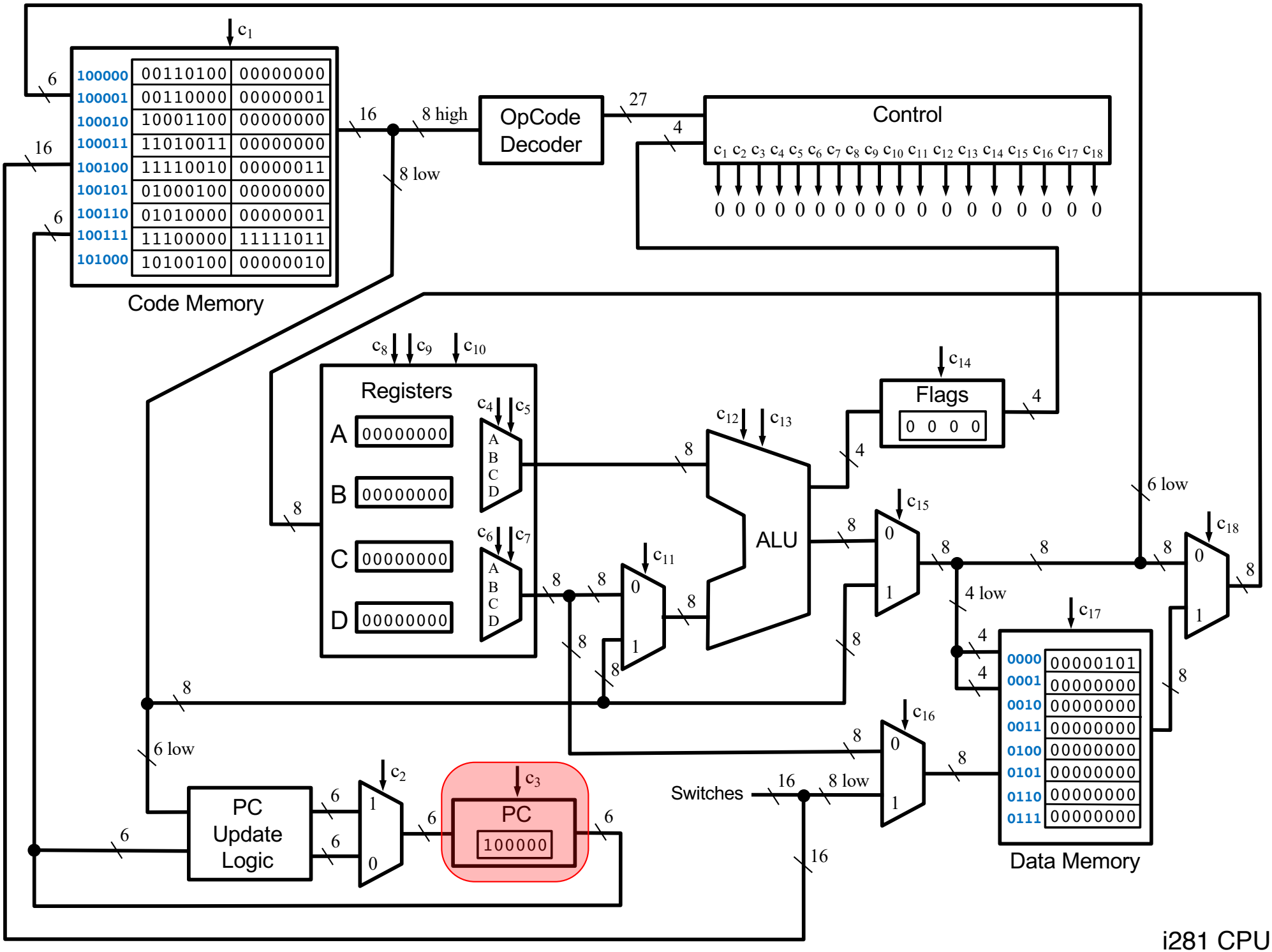
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

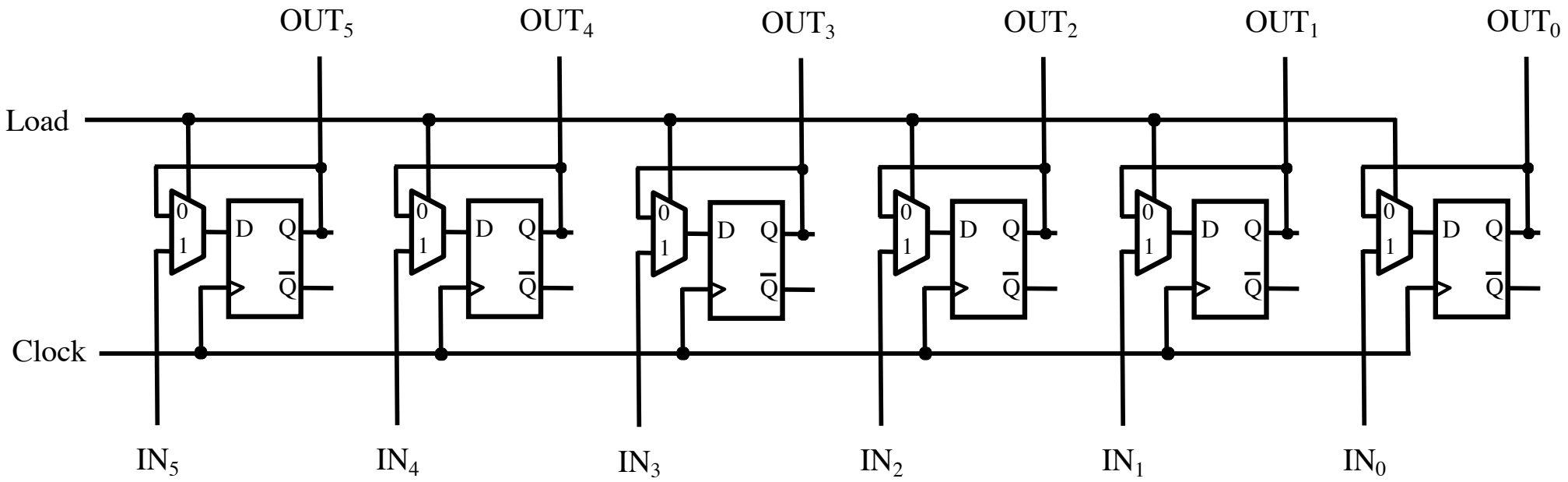
The MOVE command copies the contents of one register into another. But it tells the ALU to do addition?! Why?


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B+0
A = B+0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
B
c12 c13
ALU
c14
Flags
0 0 0 0
B+0
c15
c11
0
1
4 low
6 low
c18
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
Switches
8 low
PC Update Logic
c2
c3
PC
100000
i281 CPU

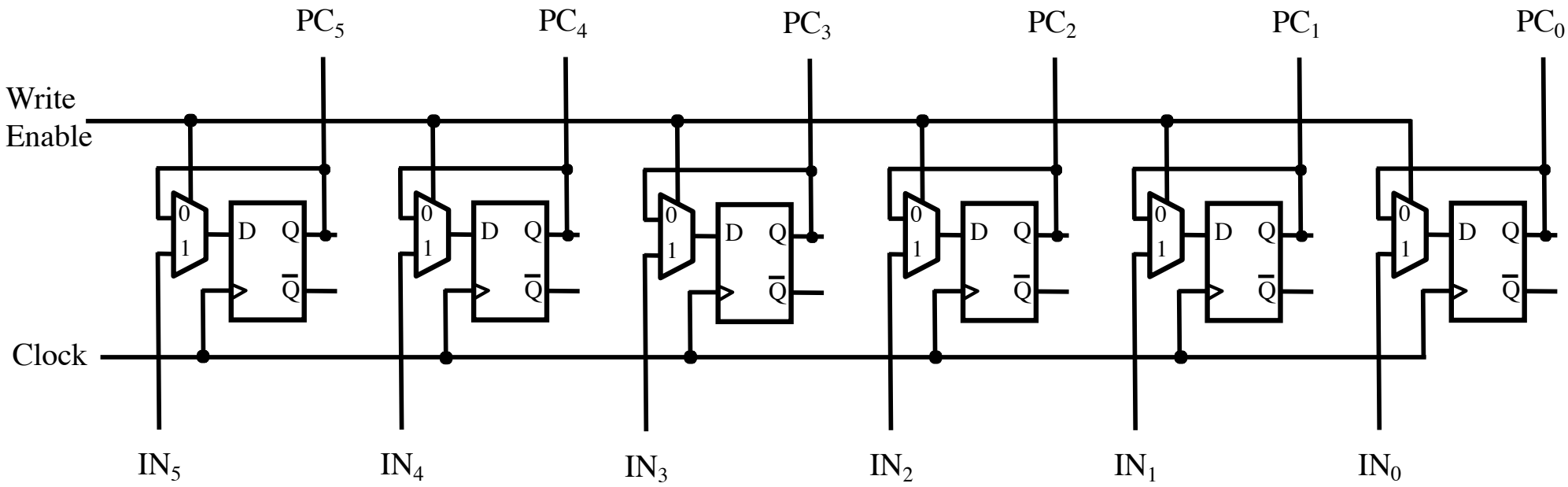
The Program Counter (PC)


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c14
Flags
0 0 0 0
4
c11
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
c2
c3
PC
100000
PC Update Logic
6 low
i281 CPU

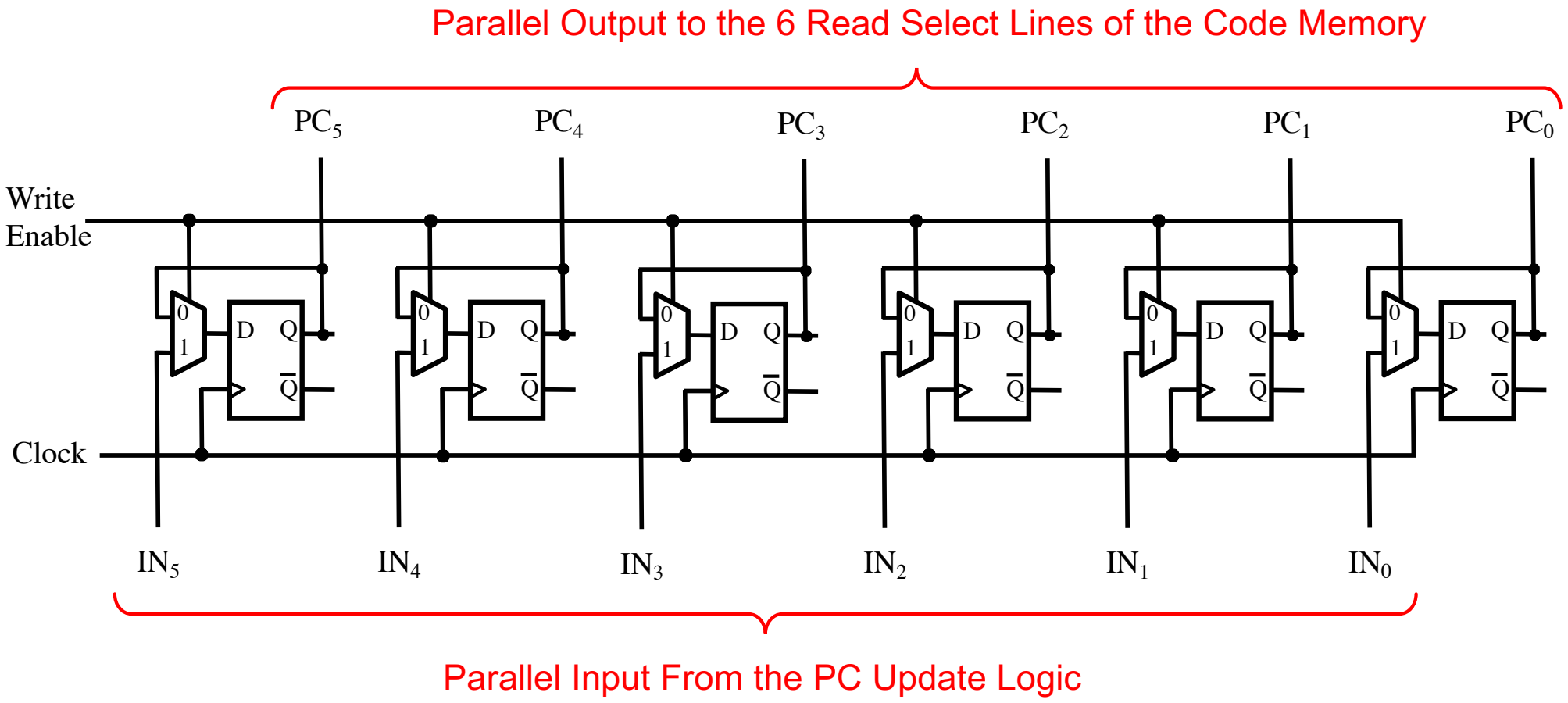
6-Bit Parallel-Access Register



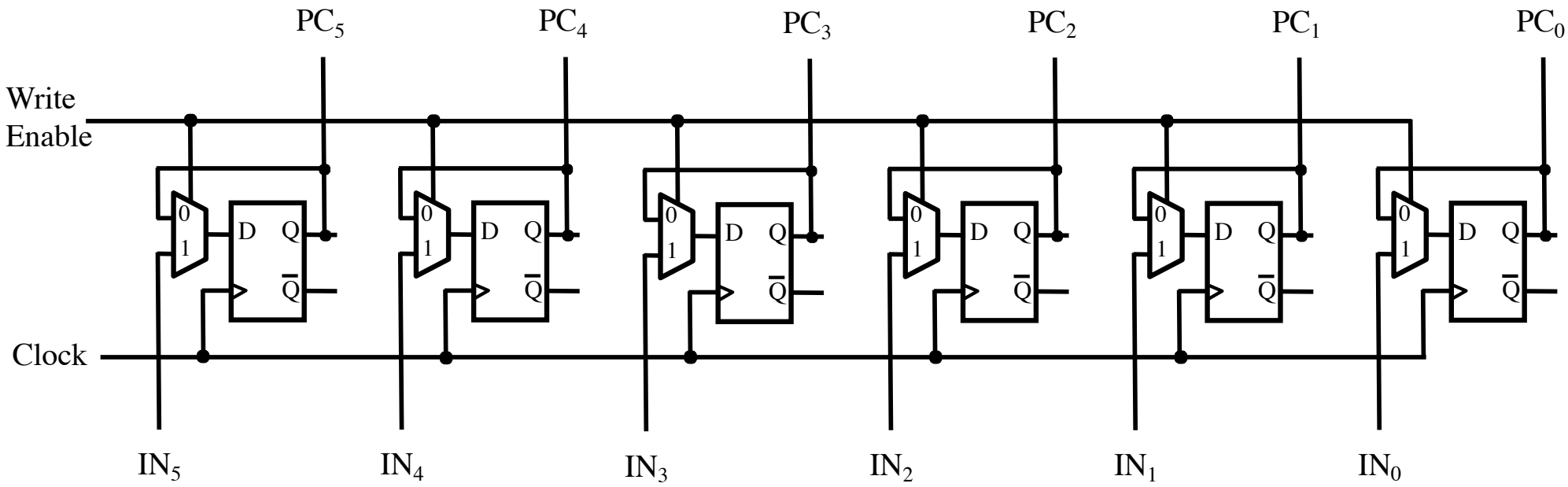
The Program Counter Register



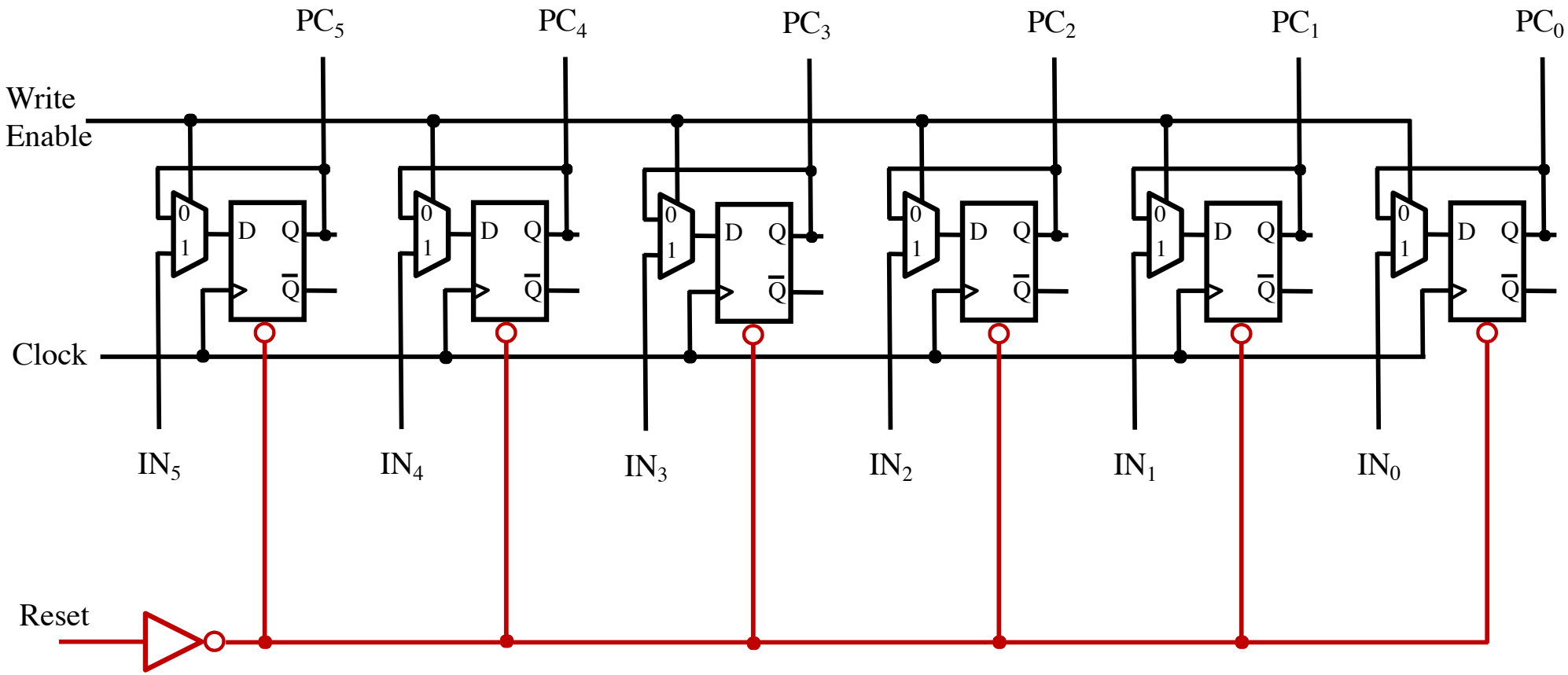
The Program Counter Register



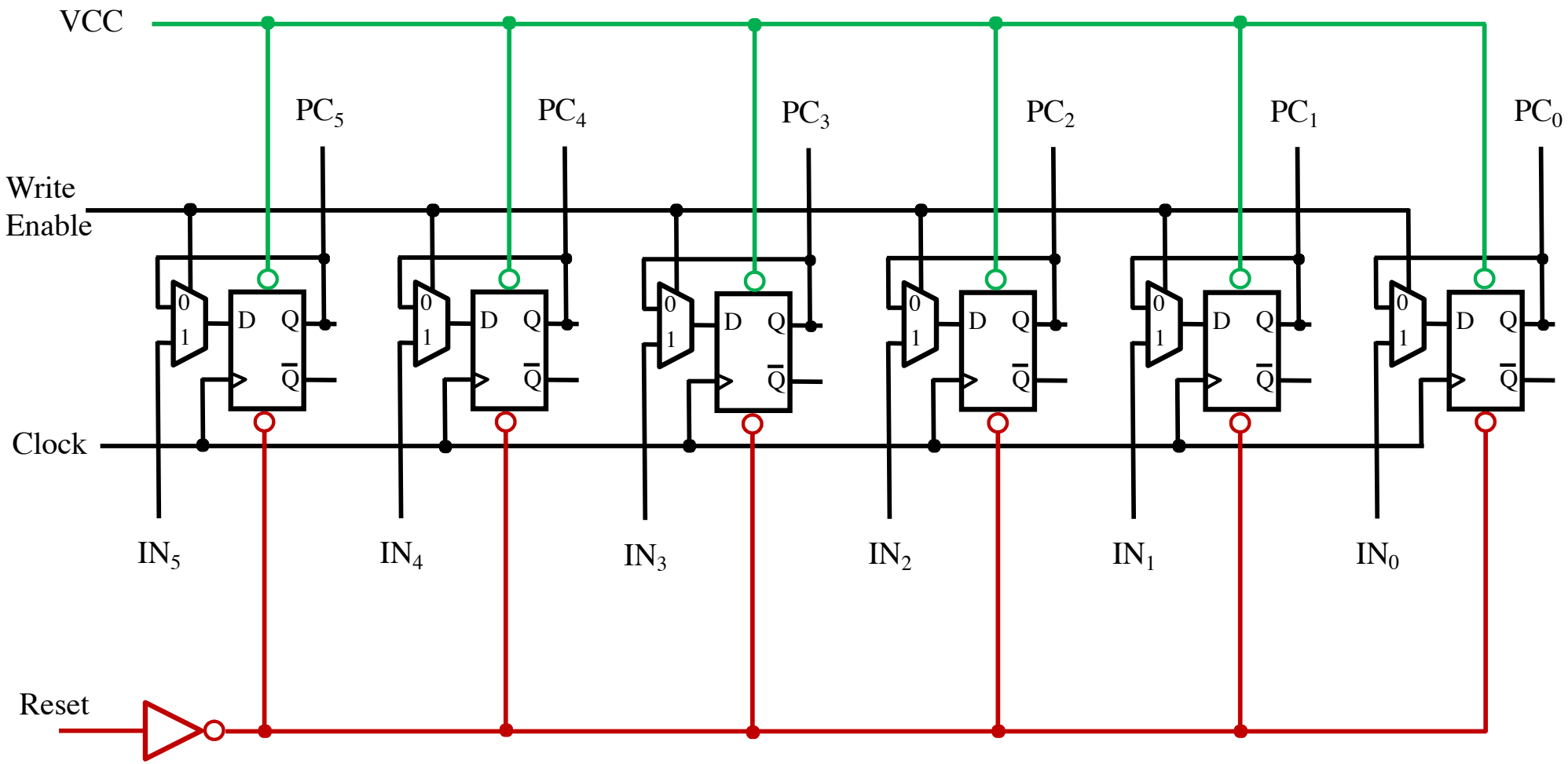
The Program Counter Register



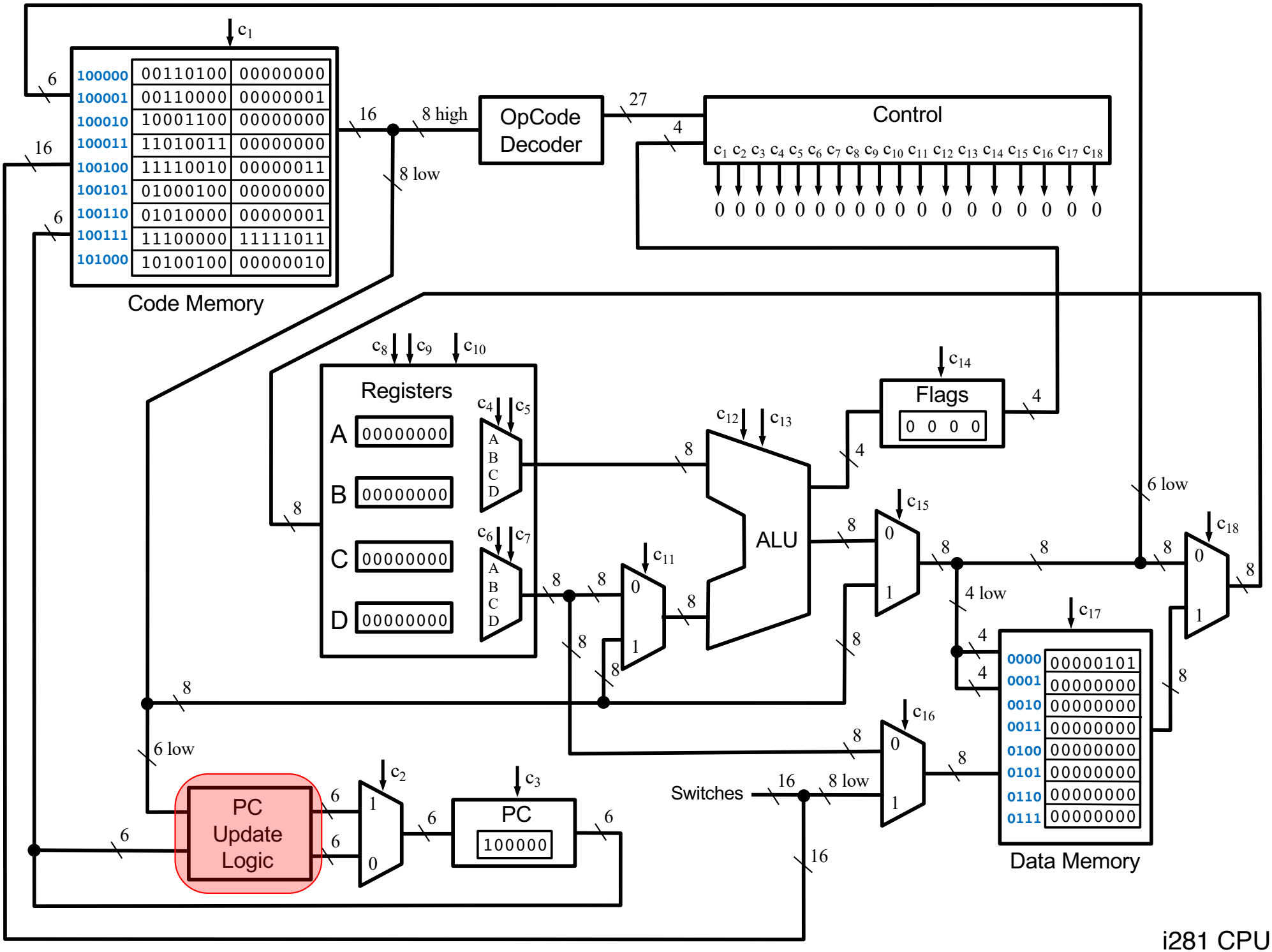
The Program Counter Register

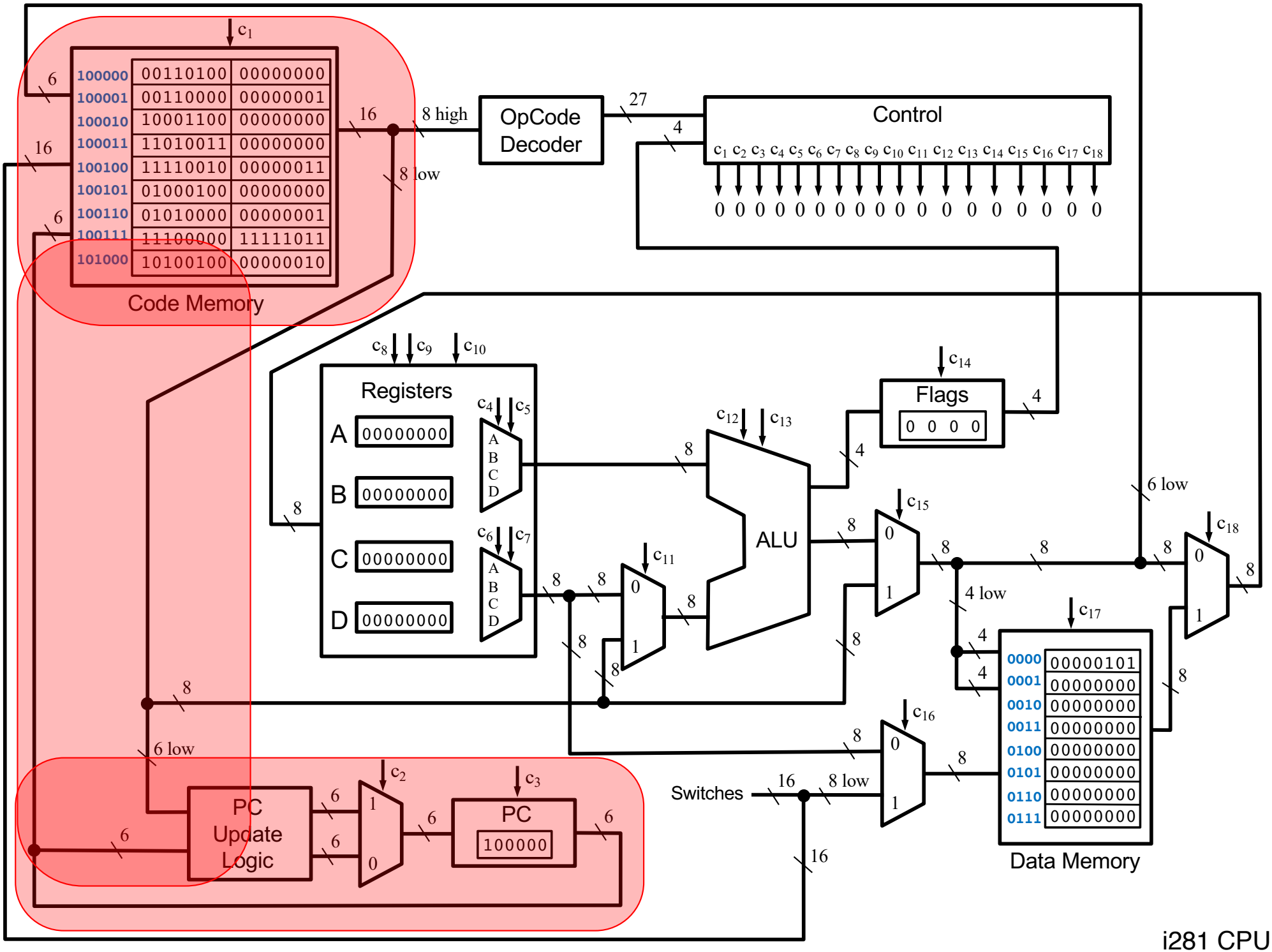


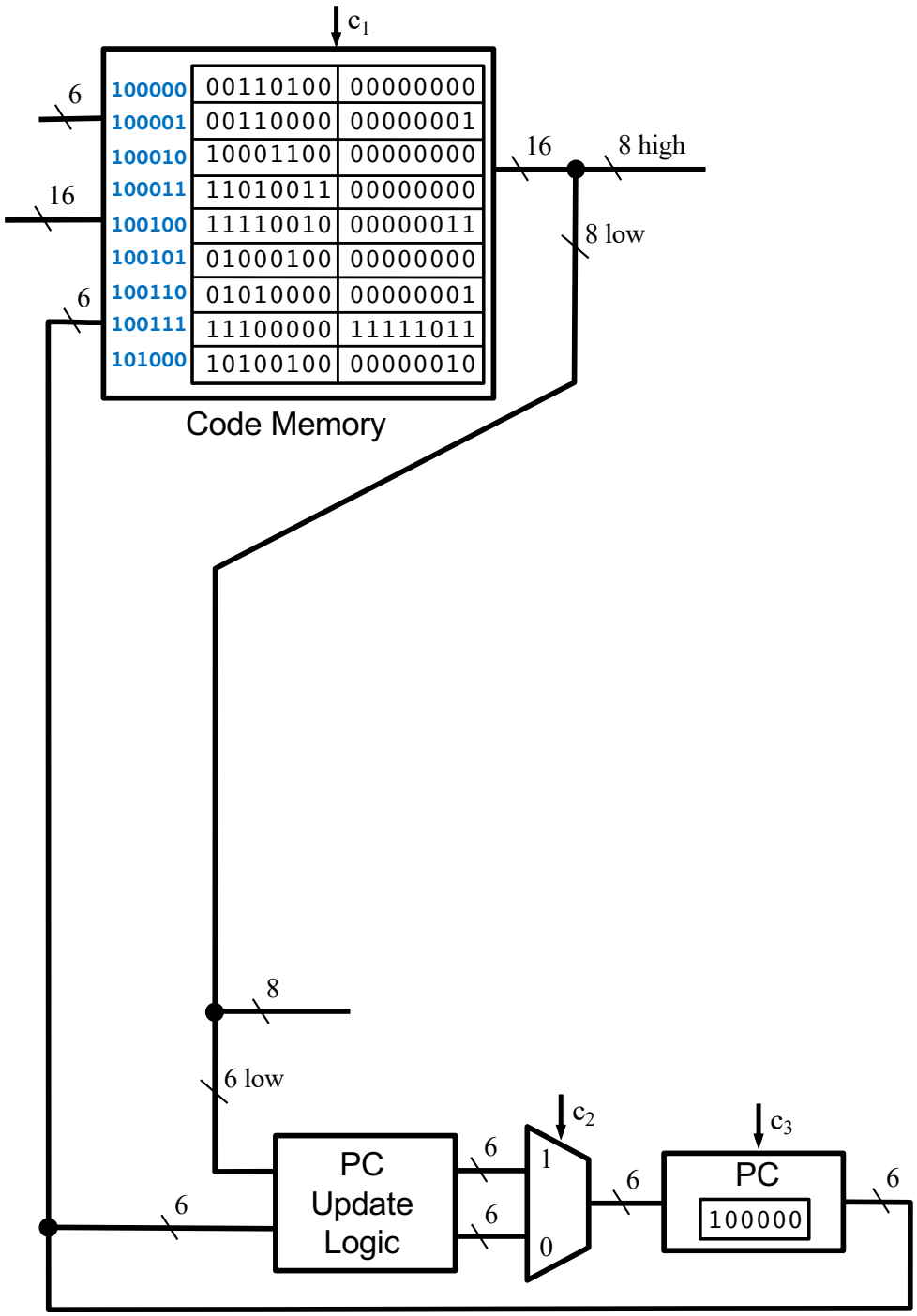
The Program Counter Register

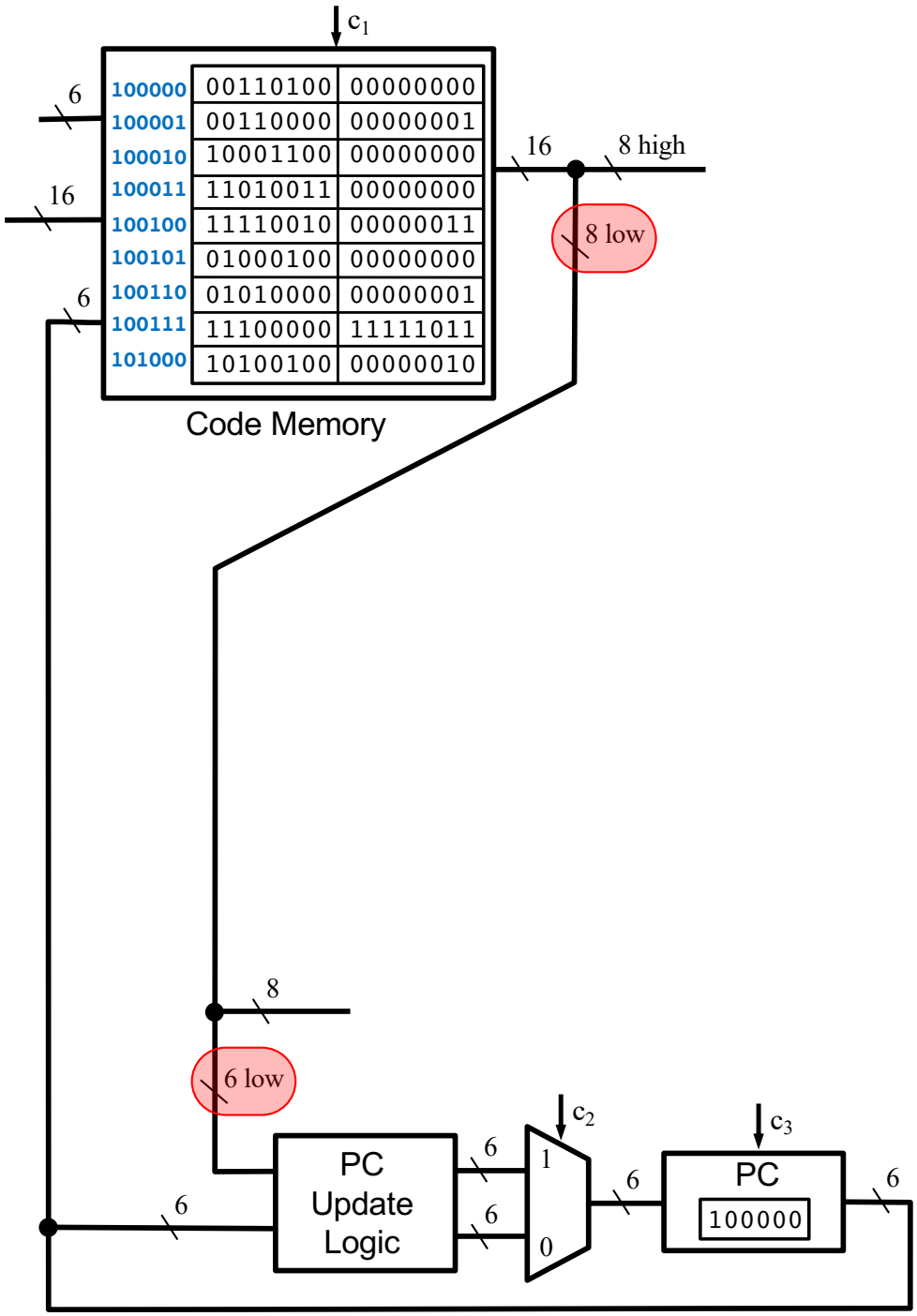


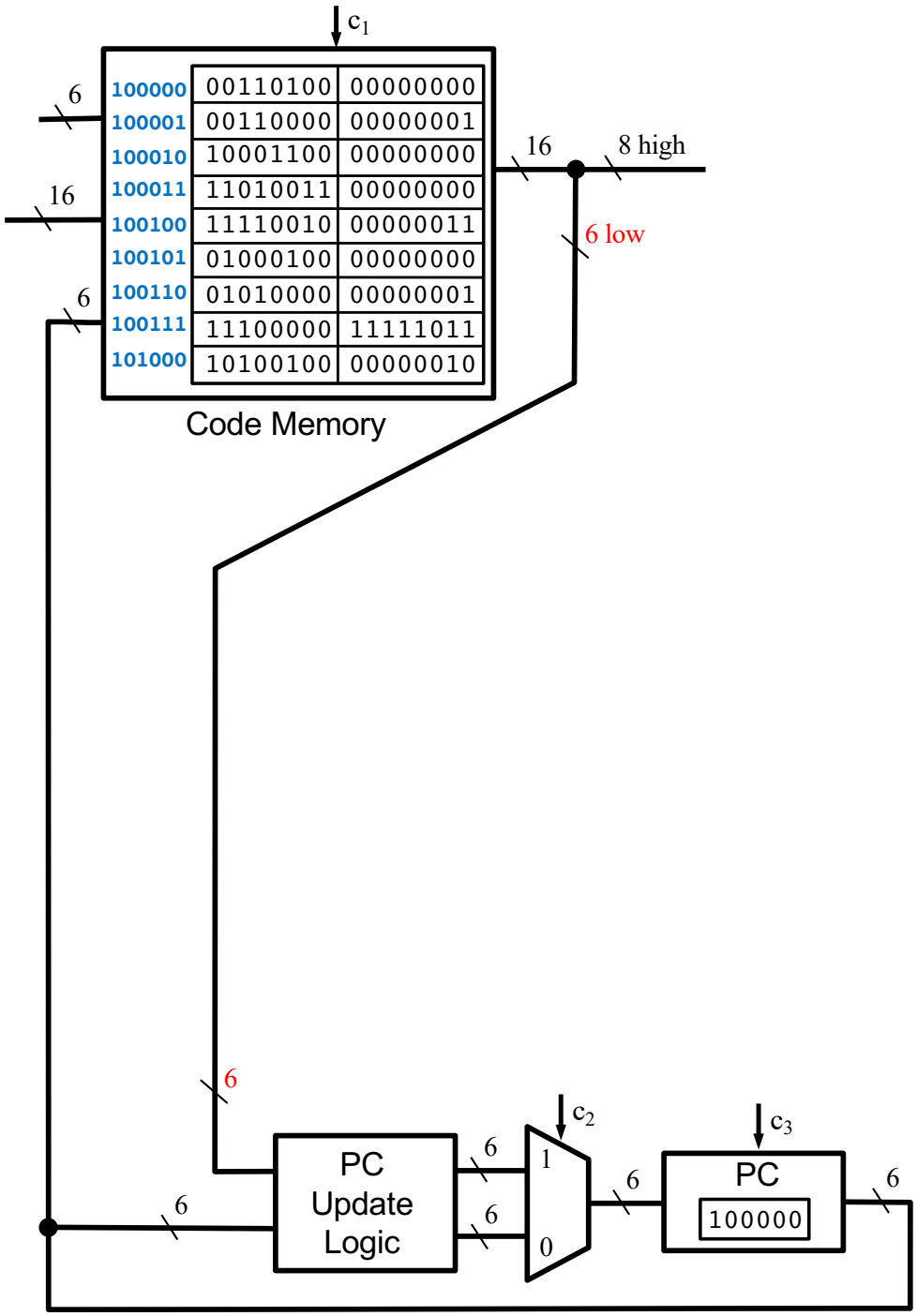
The Program Counter
Update Logic

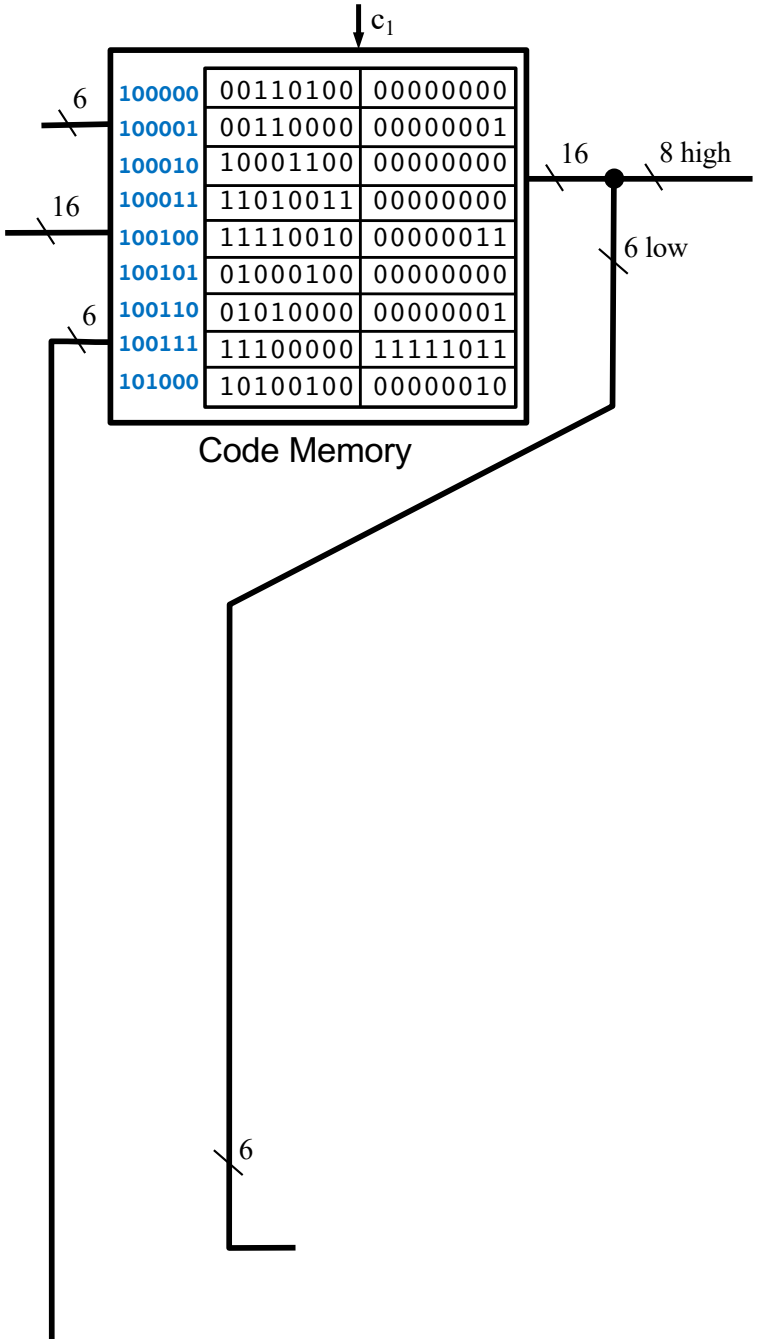

c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c14
Flags
0 0 0 0
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
16
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
i281 CPU

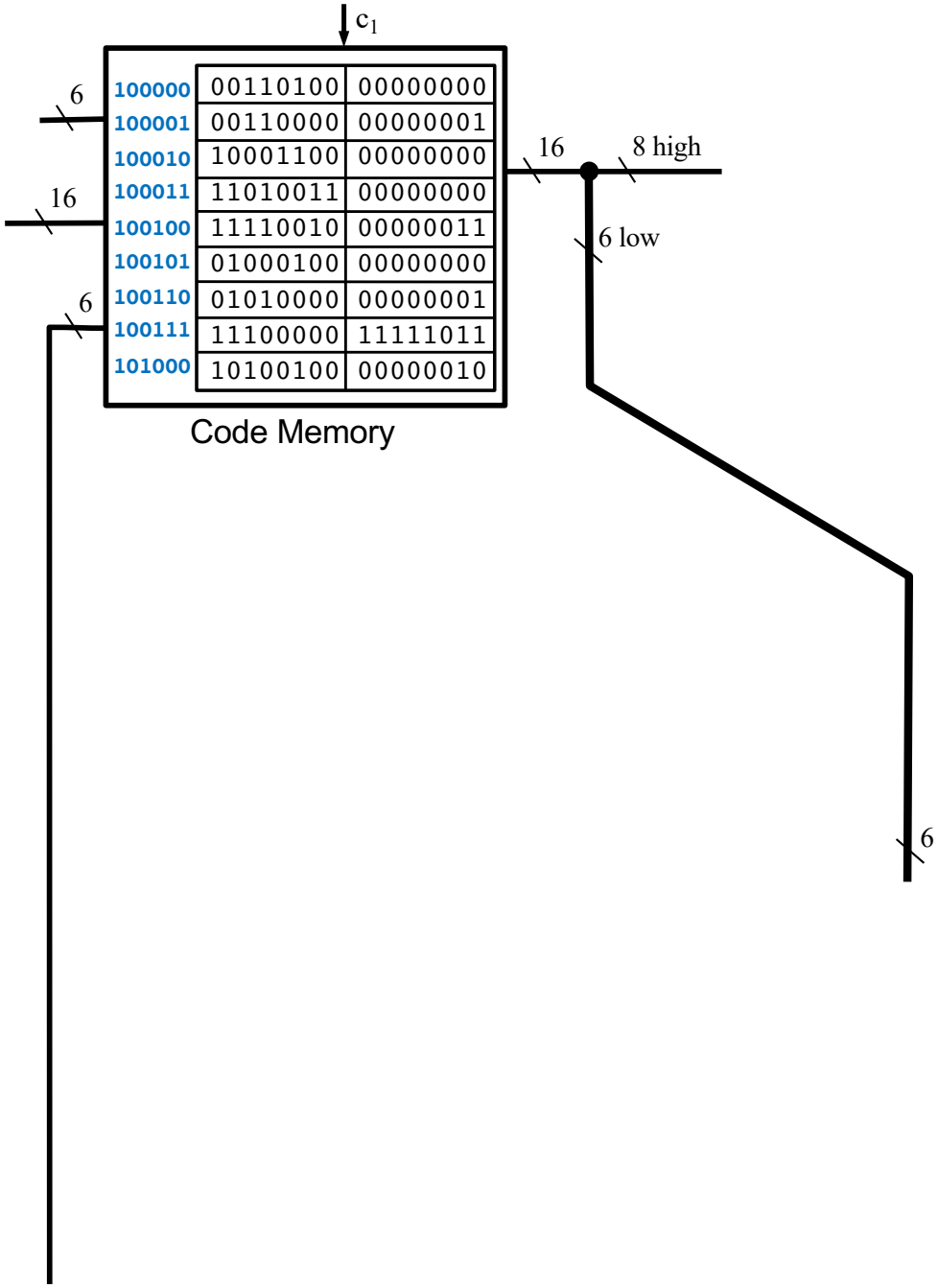

c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
16
6
Code Memory
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
16
8 low
PC Update Logic
c2
c3
PC
100000
6 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
i281 CPU

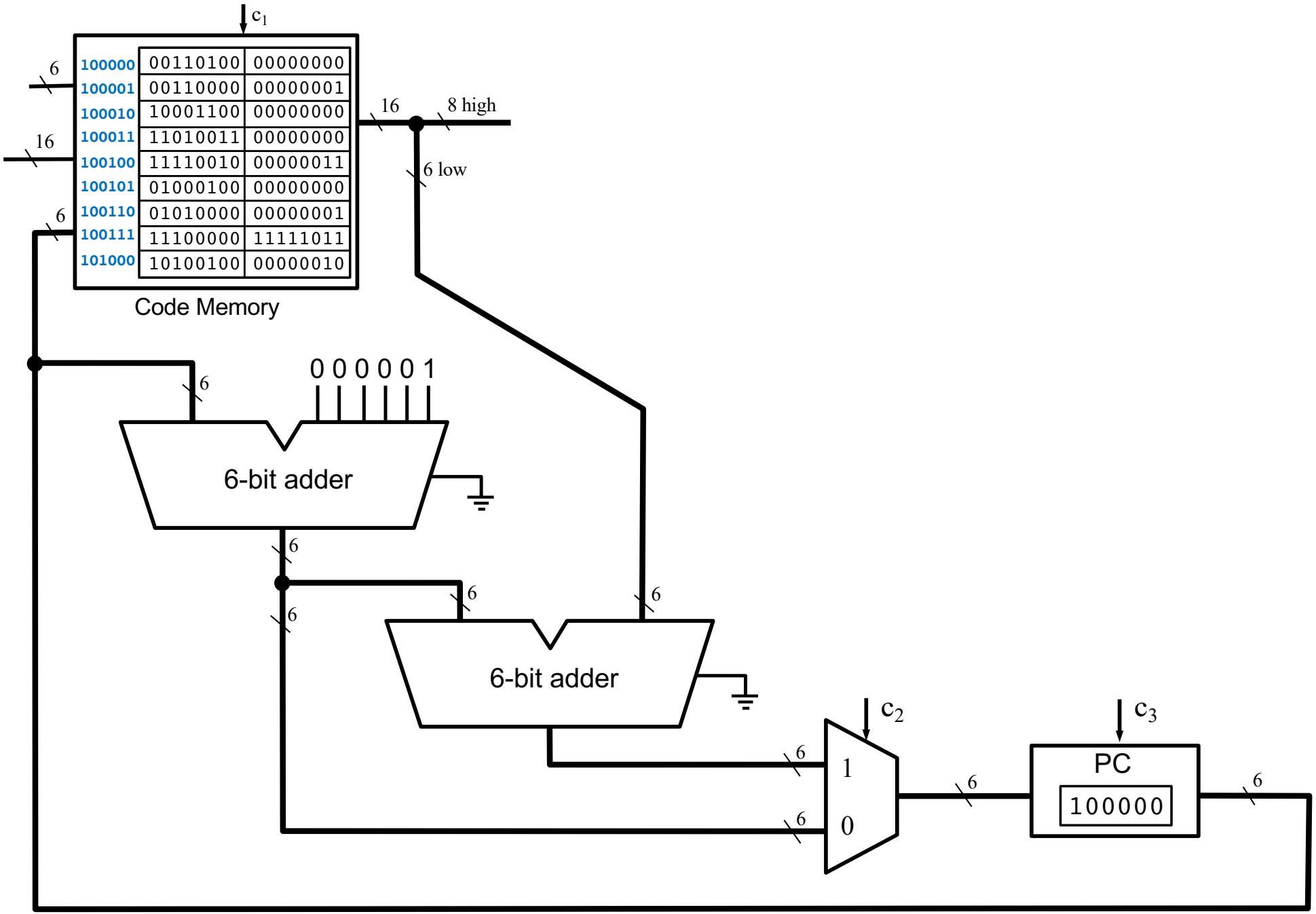

c1
6
16
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
8 low
Code Memory
8
6 low
c2
c3
PC
Update
Logic
6
6
6
1
0
6
PC
100000
6


c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
8 low
Code Memory
8
6 low
c2
c3
PC
Update
Logic
6
1
0
PC
100000

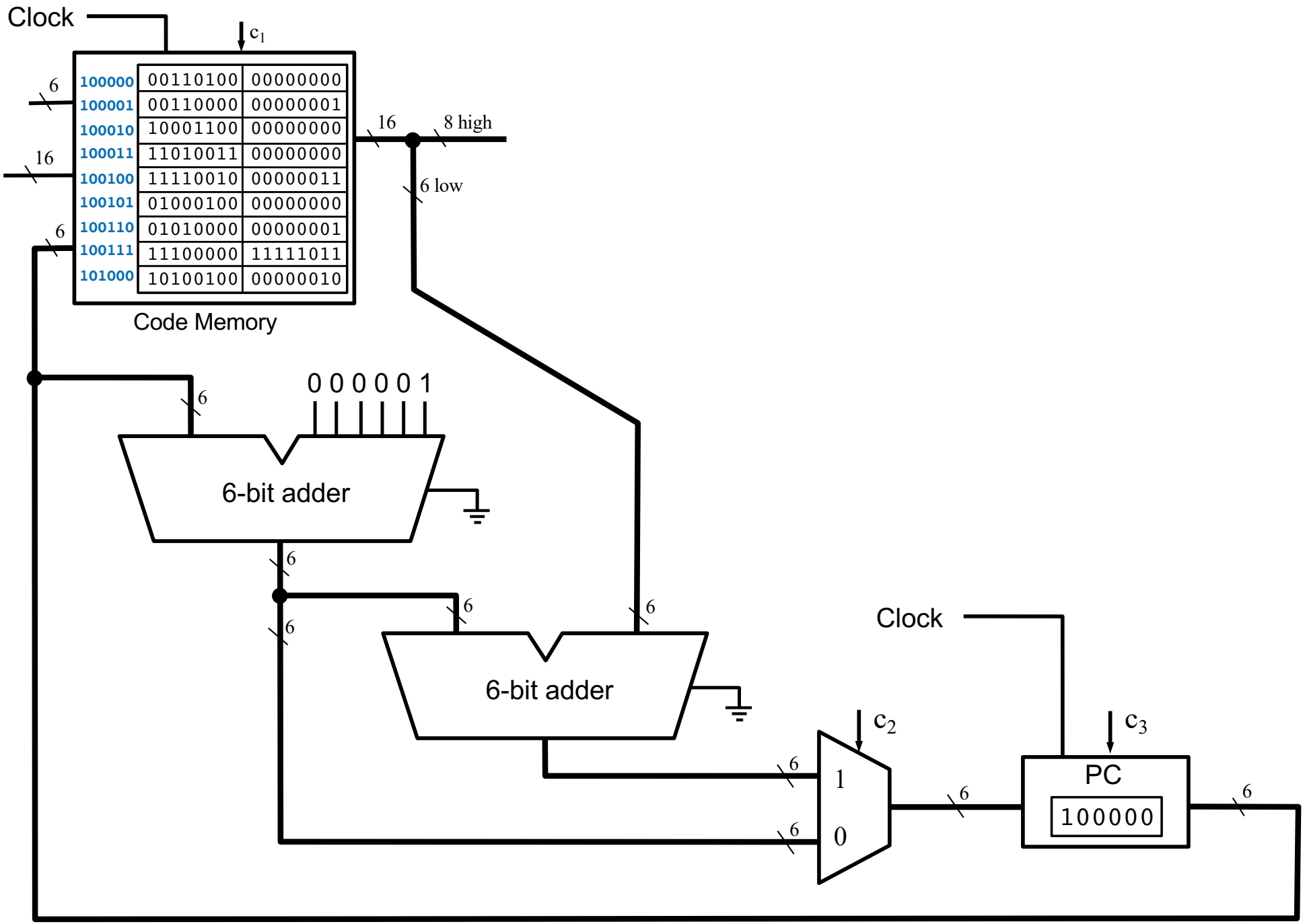

c1
6
16
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
6
c2
c3
PC
Update
Logic
6
6
6
1
0
6
PC
100000
6
6


c1
6
16
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
6

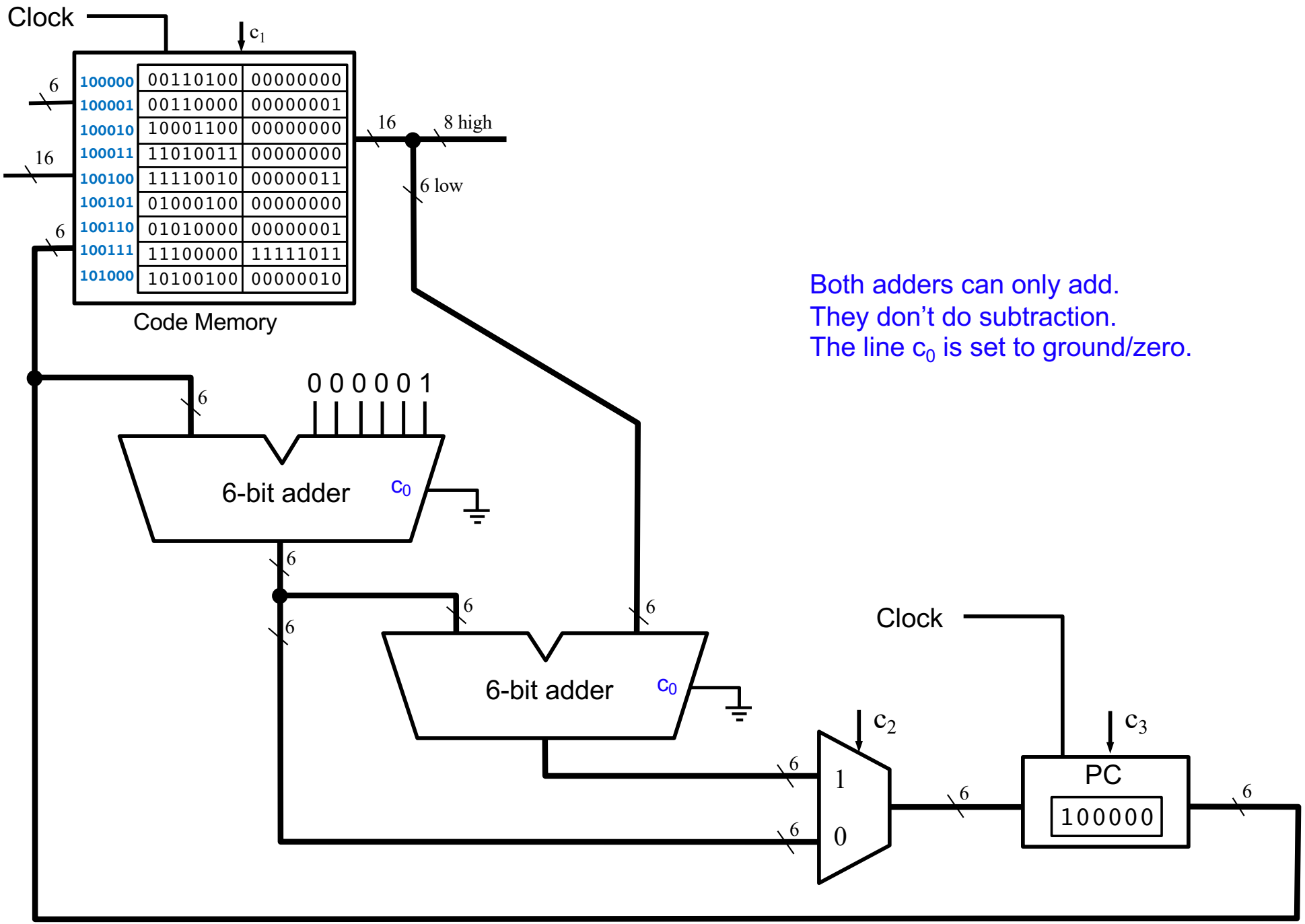

c1
6
16
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
6 low
6

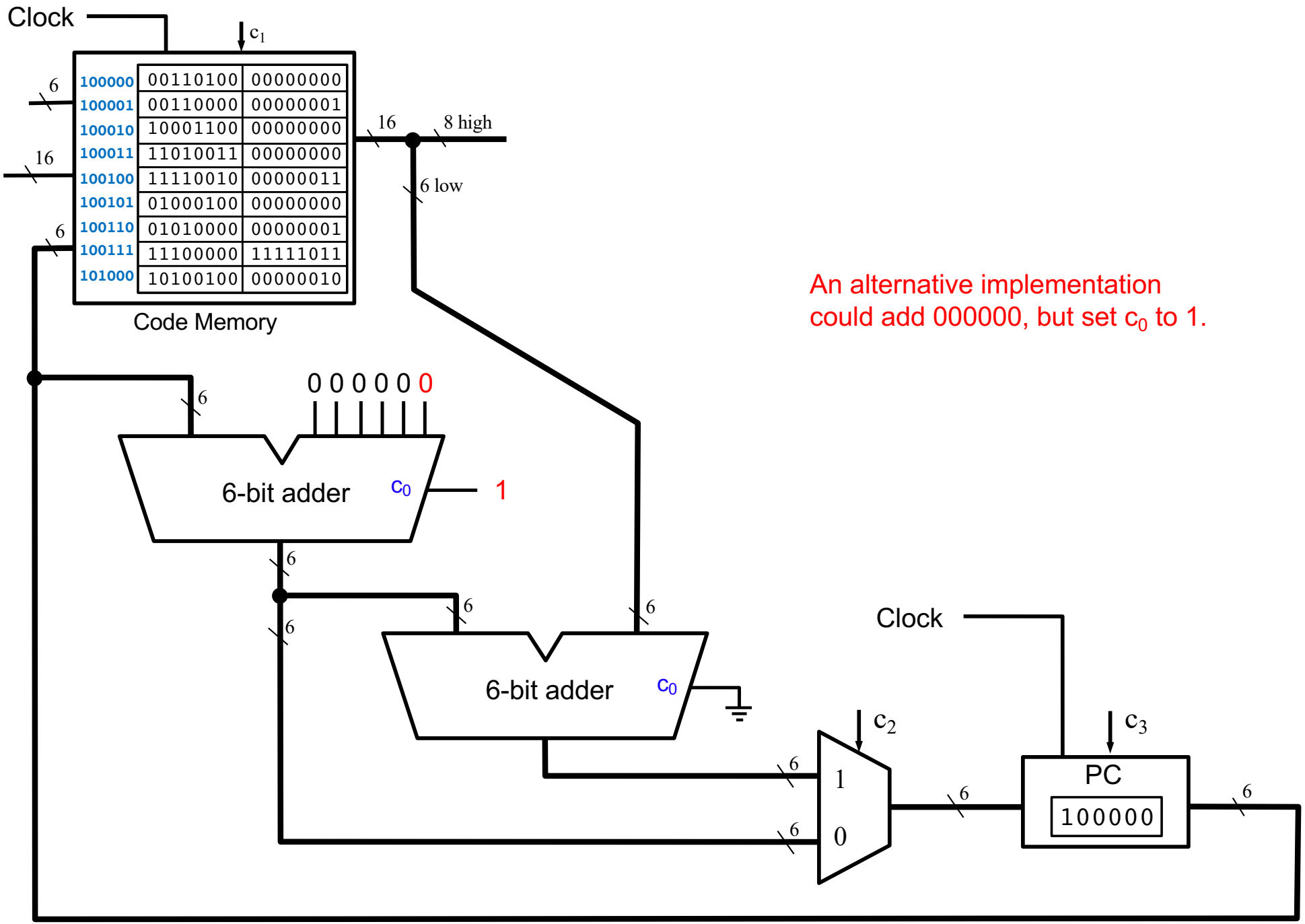

c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
6
0 0 0 0 0 1
6-bit adder
6
6
6
6-bit adder
c2
c3
6
1
0
6
6
PC
100000
6

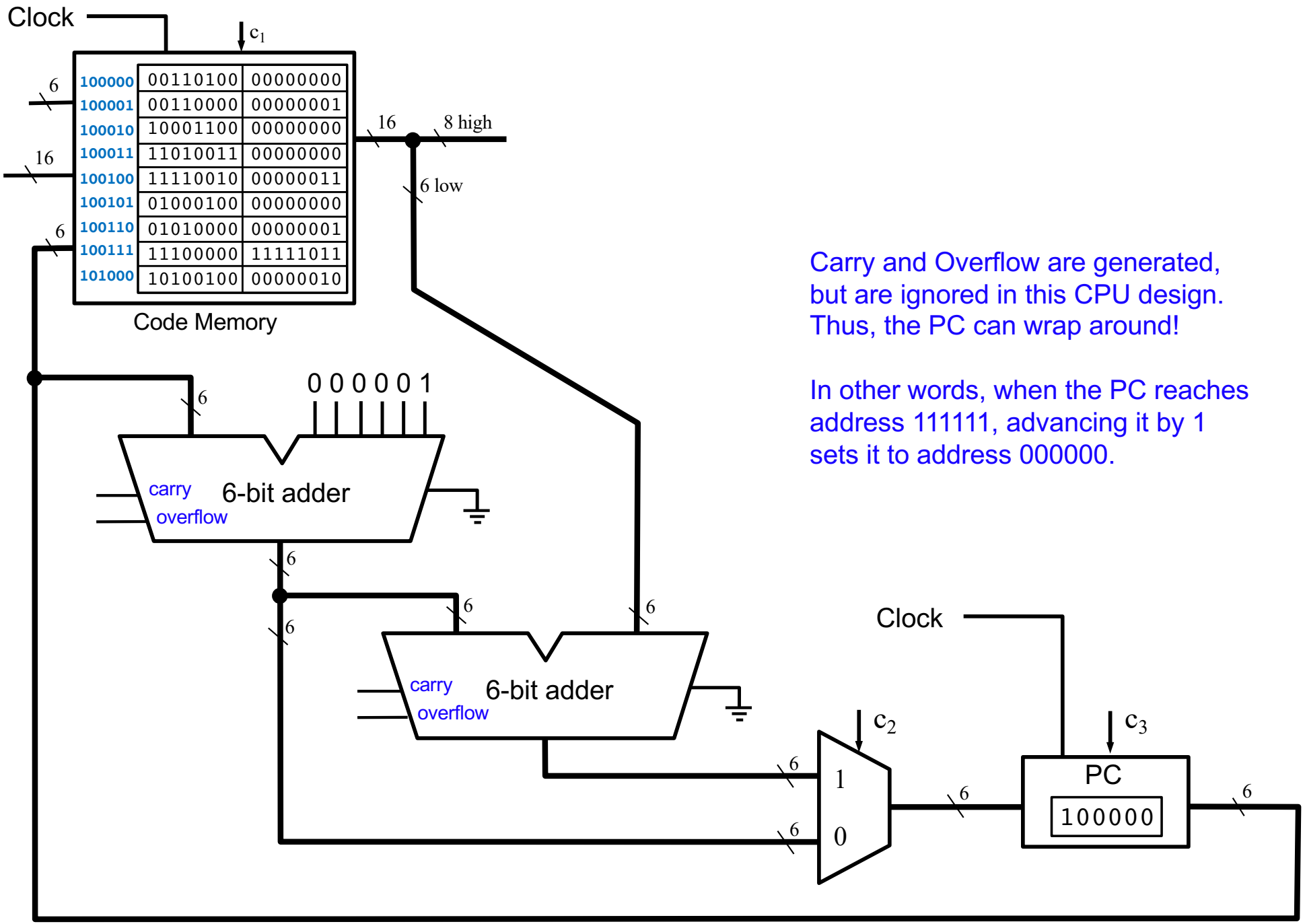
i281 CPU

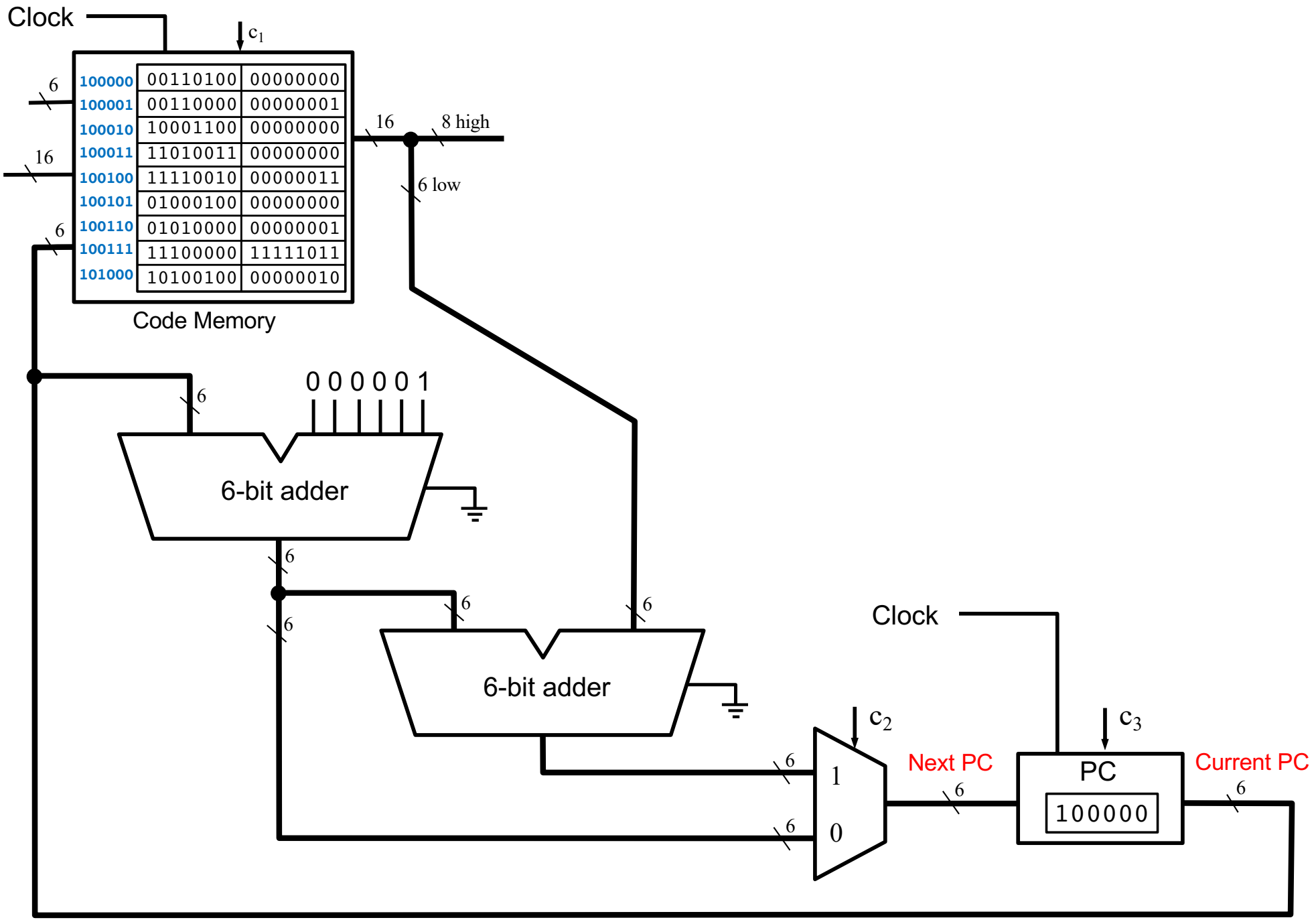

Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
0 0 0 0 0 1
6
6-bit adder
6
6
6
6
6-bit adder
Clock
c2
c3
6
6
1
0
6
PC
100000
6

i281 CPU

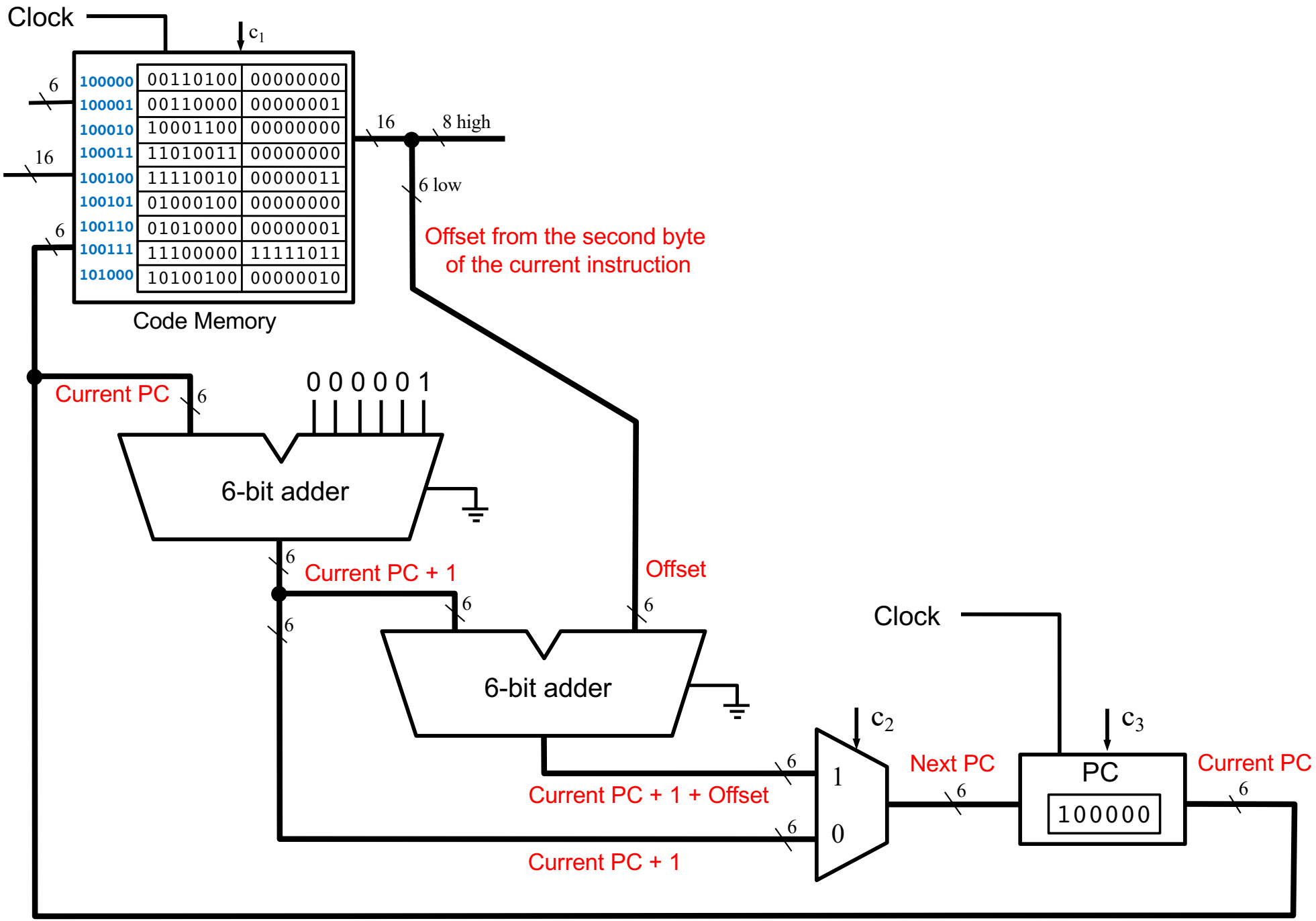

Clock
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
6
16
6
16
8 high
6 low
Code Memory
0 0 0 0 0 1
6-bit adder
c0
6-bit adder
c0
Clock
c2
c3
1
0
PC
100000
Both adders can only add.
They don't do subtraction.
The line c0 is set to ground/zero.


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
An alternative implementation
could add 000000, but set c0 to 1.
0 0 0 0 0 0
6
6-bit adder
c0
1
6
6
6
6-bit adder
c0
Clock
c2
c3
6
6
1
0
6
PC
100000
6

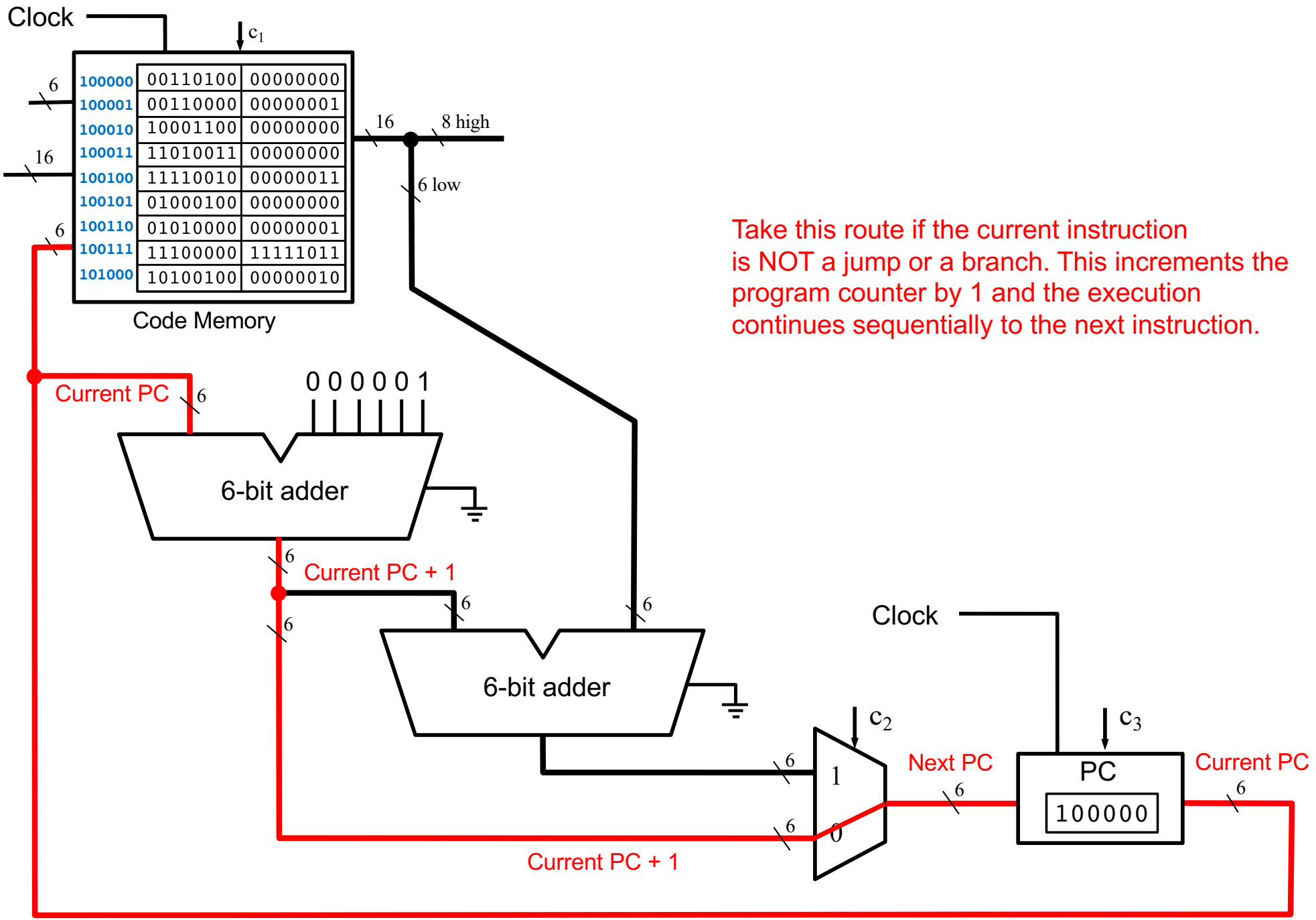

Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
6
0 0 0 0 0 1
carry
overflow
6-bit adder
6
6
6
6
carry
overflow
6-bit adder
Clock
c2
c3
6
1
0
6
6
PC
100000
6
Carry and Overflow are generated,
but are ignored in this CPU design.
Thus, the PC can wrap around!
In other words, when the PC reaches
address 111111, advancing it by 1
sets it to address 000000.

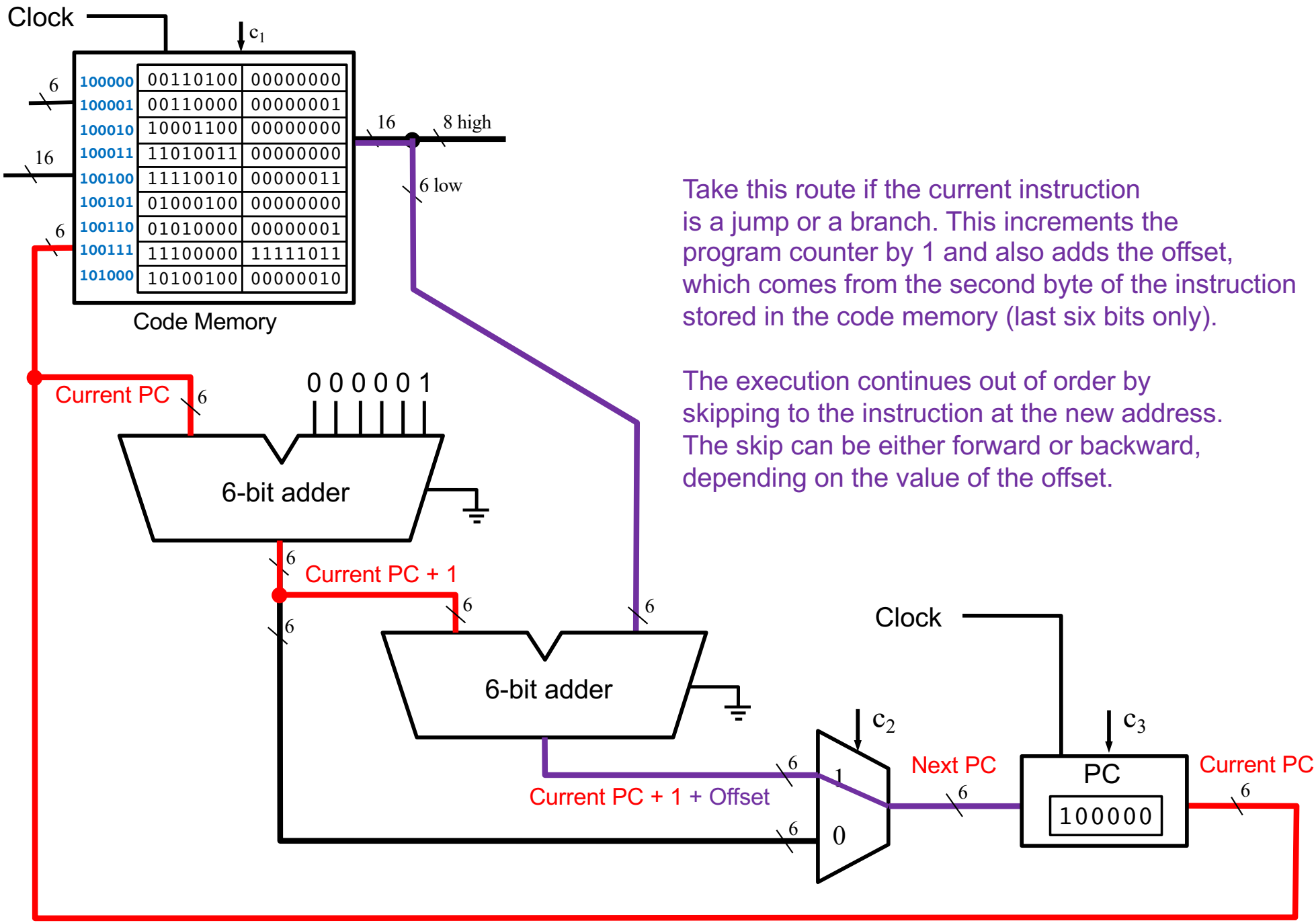

Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
000001
6
6-bit adder
6
6
6
6
6-bit adder
Clock
c2
c3
6
1
0
6
Next PC
6
PC
100000
Current PC
6

i281 CPU


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Offset from the second byte of the current instruction
Code Memory
Current PC
6
0 0 0 0 0 1
6-bit adder
6
Current PC + 1
6
6
Offset
6
6-bit adder
Clock
c2
c3
6
Current PC + 1 + Offset
1
0
6
Current PC + 1
Next PC
6
PC
100000
Current PC
6

i281 CPU


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
Take this route if the current instruction is NOT a jump or a branch. This increments the program counter by 1 and the execution continues sequentially to the next instruction.
Current PC
6
0 0 0 0 0 1
6-bit adder
6
Current PC + 1
6
6
6
6-bit adder
Clock
c2
c3
6
1
0
6
Next PC
6
PC
100000
Current PC
6
Current PC + 1



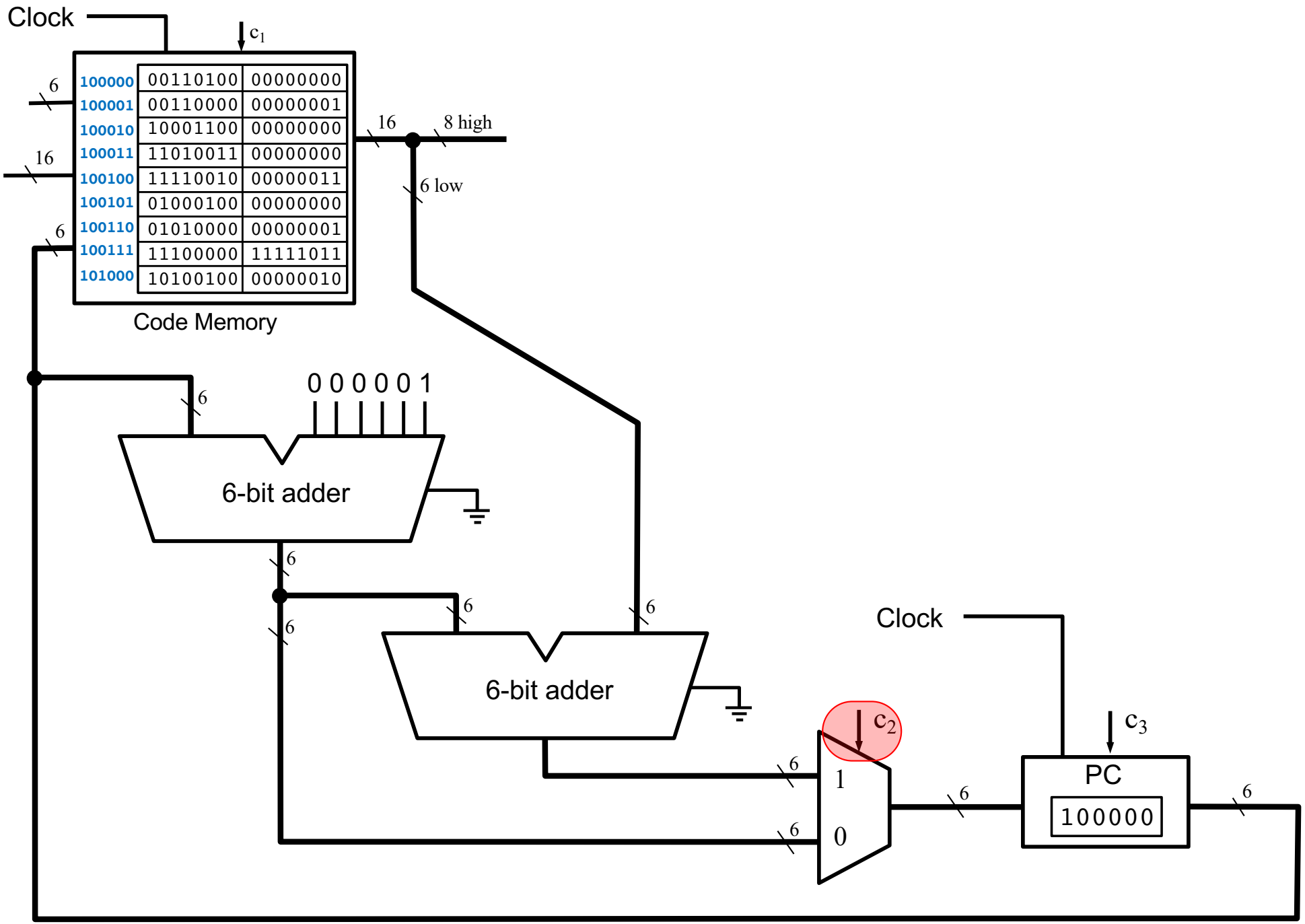
Take this route if the current instruction is a jump or a branch. This increments the program counter by 1 and also adds the offset, which comes from the second byte of the instruction stored in the code memory (last six bits only).

The execution continues out of order by skipping to the instruction at the new address. The skip can be either forward or backward, depending on the value of the offset.

Possible Offsets with +1 Correction

0	11111111
1	00000000
2	00000001
3	00000010
4	00000011
5	00000100
6	00000101
7	00000110
8	00000111
9	00001000
10	00001001
11	00001010
12	00001011
13	00001100
14	00001101
15	00001110
16	00001111
17	00010000
18	00010001
19	00010010
20	00010011
21	00010100
22	00010101
23	00010110
24	00010111
25	00011000
26	00011001
27	00011010
28	00011011
29	00011100
30	00011101
31	00011110
32	00011111

-1	11111110
-2	11111101
-3	11111100
-4	11111011
-5	11111010
-6	11111001
-7	11111000
-8	11110111
-9	11110110
-10	11110101
-11	11110100
-12	11110011
-13	11110010
-14	11110001
-15	11110000
-16	11101111
-17	11101110
-18	11101101
-19	11101100
-20	11101011
-21	11101010
-22	11101001
-23	11101000
-24	11100111
-25	11100110
-26	11100101
-27	11100100
-28	11100011
-29	11100010
-30	11100001
-31	11100000
-32	N/A


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
0 0 0 0 0 1
6
6-bit adder
6
6
6
6
6-bit adder
Clock
c2
c3
6
6
1
0
6
PC
100000
6

i281 CPU

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

computed using the flags register

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

C_2 is the OR of these five times the OPCODE

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

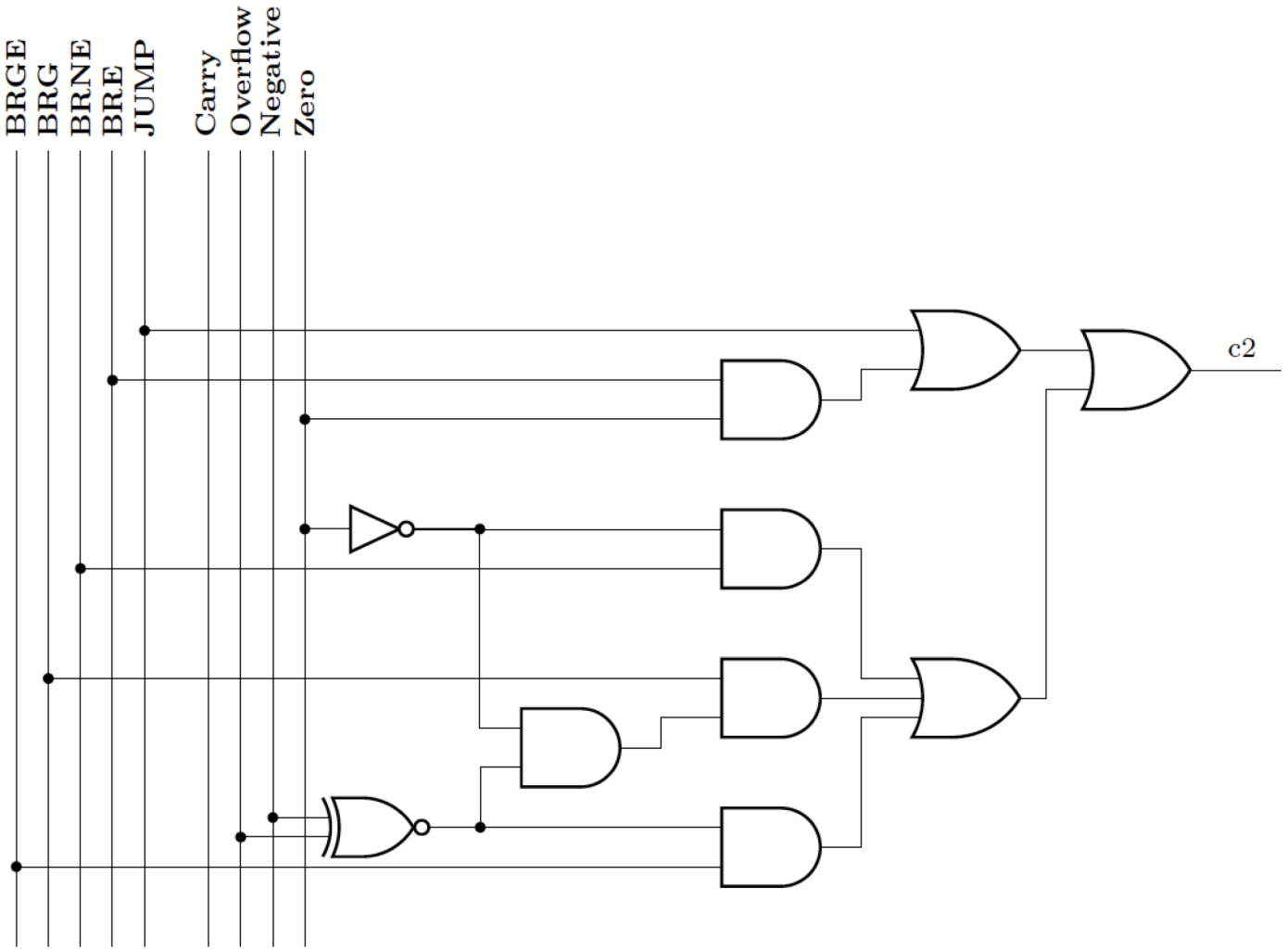
Comparison of Signed Numbers

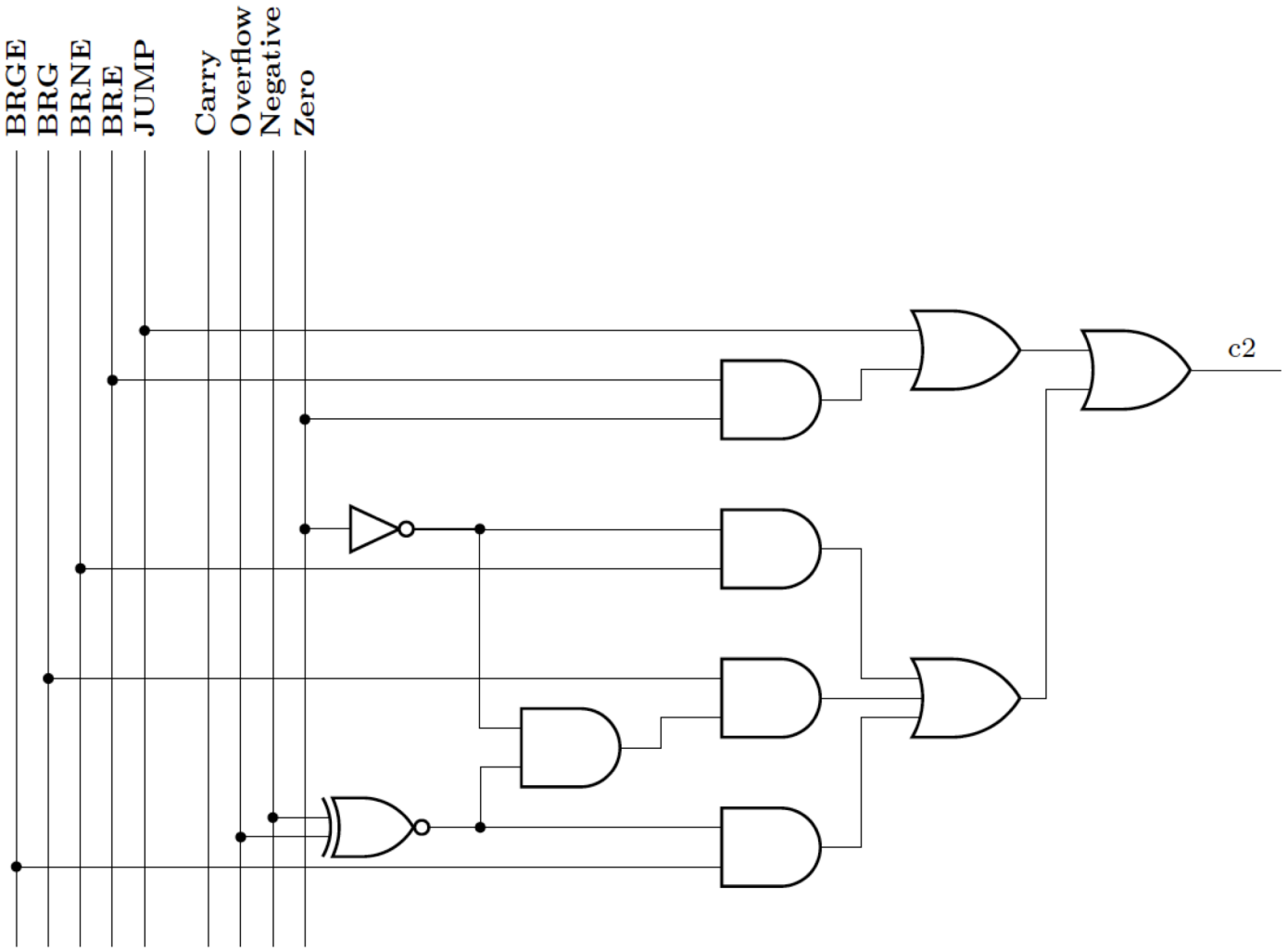
Comparison of Signed Numbers

• Equal	ZF = 1
• Not equal	ZF = 0
• Greater	ZF = 0 and NF = OF
• Greater or Equal	NF = OF
• Less	NF != OF
• Less or Equal	ZF = 1 or NF != OF

Comparison of Signed Numbers

- Equal — ZF
- Not equal — $\overline{ZF}$
- Greater — $\overline{ZF} \bullet XNOR(NF, OF)$
- Greater or Equal — $XNOR(NF, OF)$
- Less — $XOR(NF, OF)$
- Less or Equal — $ZF + XOR(NF, OF)$


BRGE
BRG
BRNE
BRE
JUMP
Carry
Overflow
Negative
Zero
c2

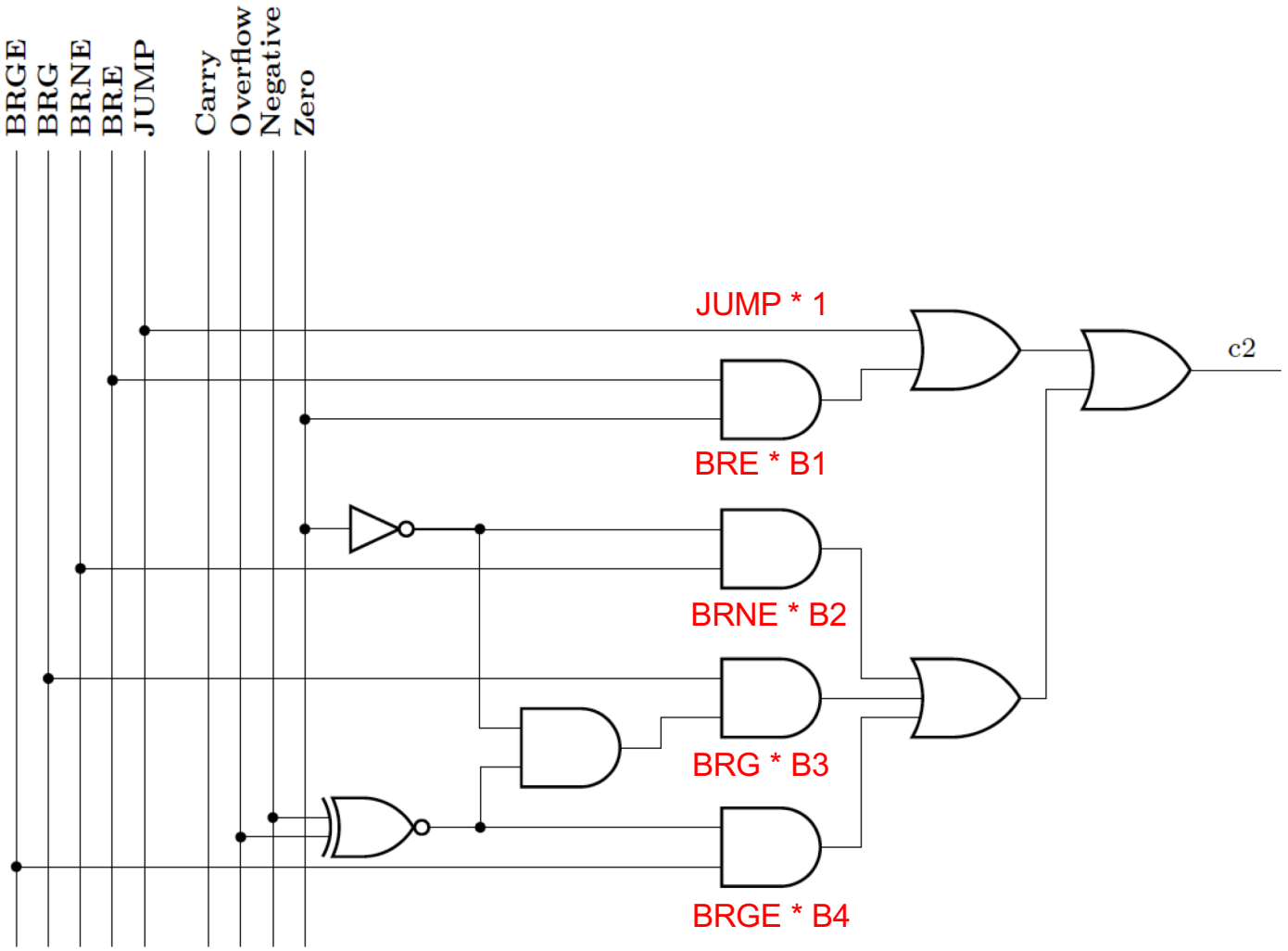


JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

C_2 is the OR of these five times the OPCODE

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)



JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

C_2 is the OR of these five times the OPCODE

```
B1=  ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)
```

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

Some Interesting Dualities

- $\overline{\text{Equal}} = \text{Not Equal}$
- $\overline{\text{Greater}} = \text{Less or Equal}$
- $\overline{\text{Less}} = \text{Greater or Equal}$

Comparison of Unsigned Numbers (not supported by this CPU)

Comparison of Unsigned Numbers

- Equal
- Not equal
- Greater
- Greater or equal
- Less
- Less or Equal

Comparison of Unsigned Numbers

- Equal
- Not equal
- Greater / Above
- Greater or Equal / Above or Equal
- Less / Below
- Less or Equal / Below or Equal

Comparison of Unsigned Numbers

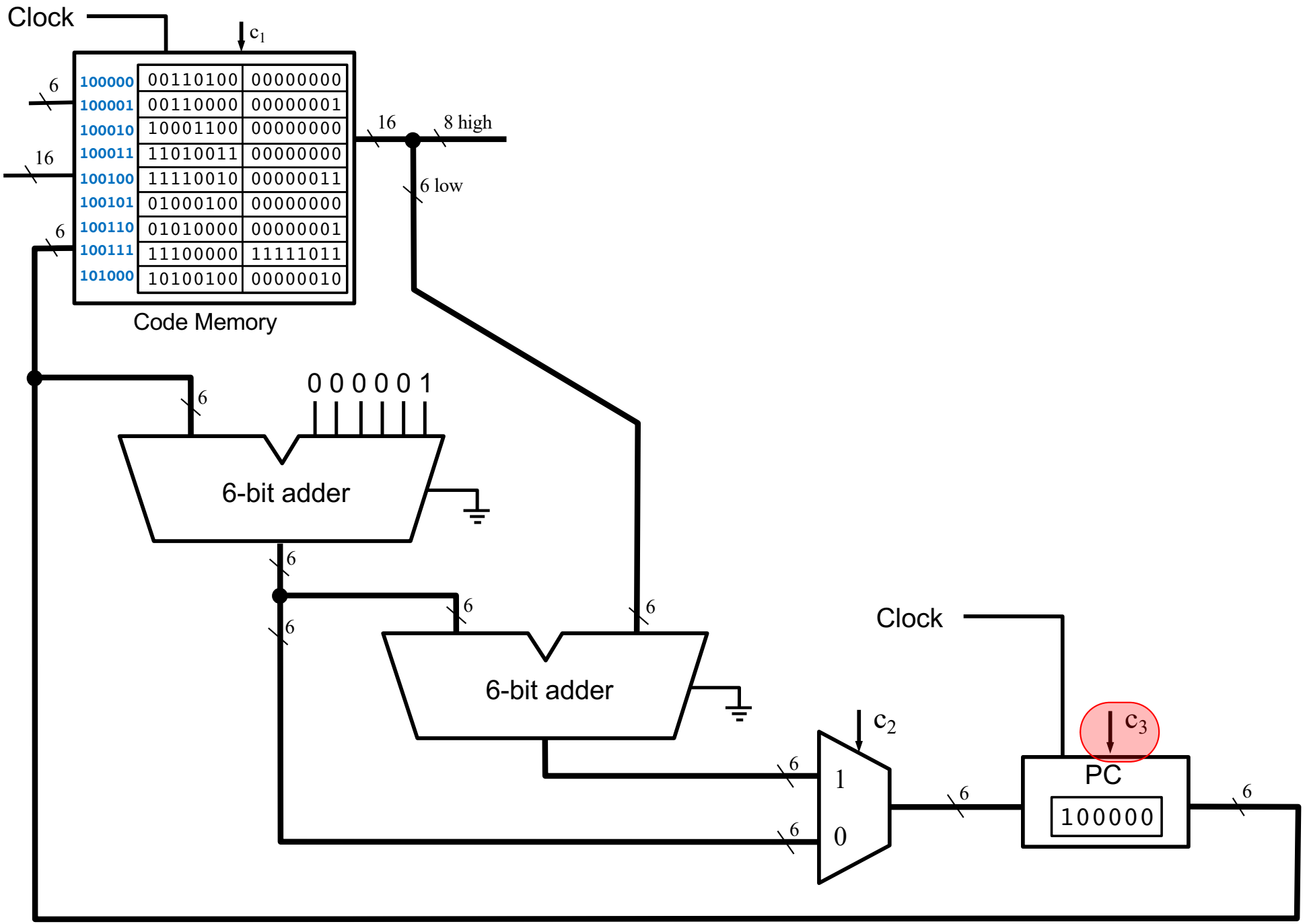
• Equal	ZF = 1
• Not equal	ZF = 0
• Greater	ZF = 0 and CF = 1
• Greater or Equal	CF = 1
• Less	CF = 0
• Less or Equal	ZF = 1 or CF = 0

Comparison of Unsigned Numbers

- Equal — ZF
- Not equal — $\overline{ZF}$
- Greater — $\overline{ZF} \cdot CF$
- Greater or Equal — CF
- Less — $\overline{CF}$
- Less or Equal — $ZF + \overline{CF}$

Comparison of Unsigned Numbers

- Equal — ZF
- Not equal — $\overline{ZF}$
- Above — $\overline{ZF} \cdot CF$
- Above or Equal — CF
- Below — $\overline{CF}$
- Below or Equal — $ZF + \overline{CF}$


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
6
Code Memory
0 0 0 0 0 1
6
6-bit adder
6
6
6
6
6-bit adder
Clock
c2
c3
6
6
1
0
6
PC
100000
6

i281 CPU

	c_1 IMEM_WRITE_ENABLE	c_2 PROGRAM_COUNTER_MUX	c_3 PROGRAM_COUNTER_WRITE_EN	c_4 REGISTERS_PORT0_SELECT1	c_5 REGISTERS_PORT0_SELECT0	c_6 REGISTERS_PORT1_SELECT1	c_7 REGISTERS_PORT1_SELECT0	c_8 REGISTERS_WRITE_SELECT1	c_9 REGISTERS_WRITE_SELECT0	c_{10} REGISTERS_WRITE_ENABLE	c_{11} ALU_SOURCE_MUX	c_{12} ALU_SELECT1	c_{13} ALU_SELECT0	c_{14} FLAGS_WRITE_ENABLE	c_{15} ALU_RESUT_MUX	c_{16} DMEM_INPUT_MUX	c_{17} DMEM_WRITE_ENABLE	c_{18} REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

c_3 is always 1, because this is a single-cycle processor, i.e., it executes one instruction per clock cycle.

Simulation of the For Loop program that adds the numbers from 1 to 5

Add the numbers from 1 to 5

```
// C Version
// using a for loop

int N=5;
int i;
int sum;

int main()
{
     sum=0;
     for(i=1; i<=N; i++) {
        sum+=i;
     }

     // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

Machine Code Version

Data Memory:

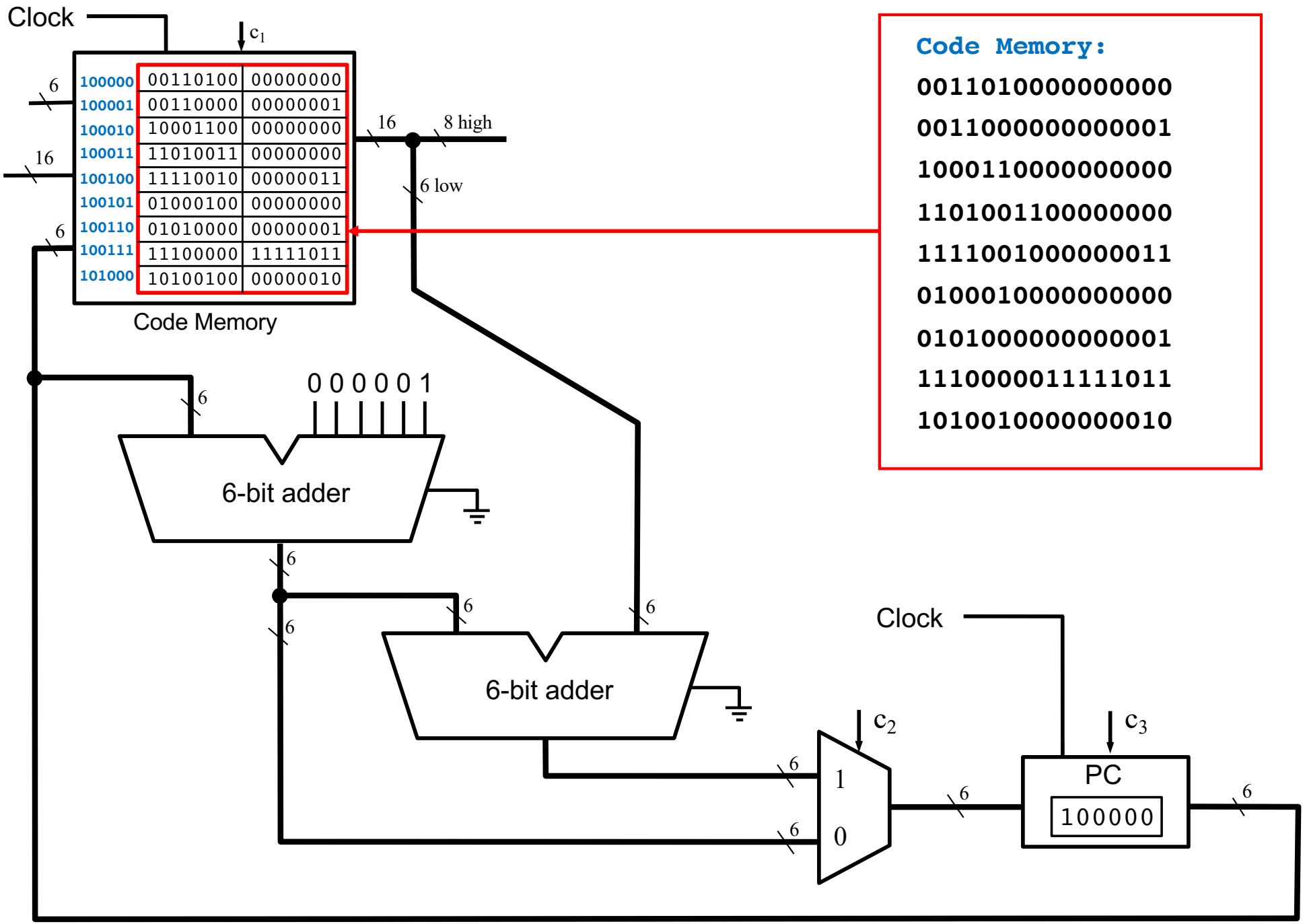
```
00000101
00000000
00000000
```

Code Memory:

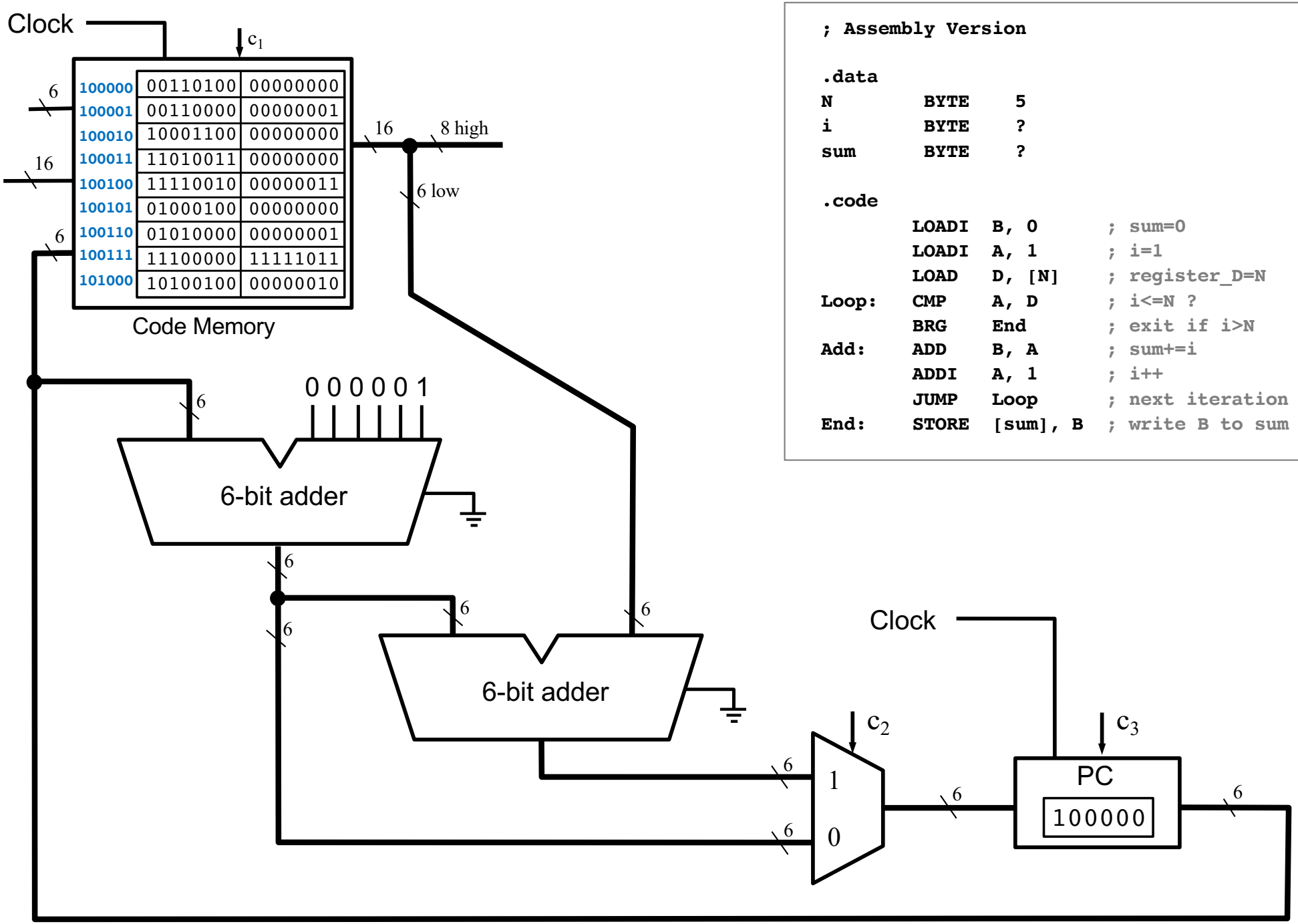
```
0011010000000000
0011000000000001
1000110000000000
1101001100000000
1111001000000011
0100010000000000
0101000000000001
1110000011111011
1010010000000010
```

; Assembly Version

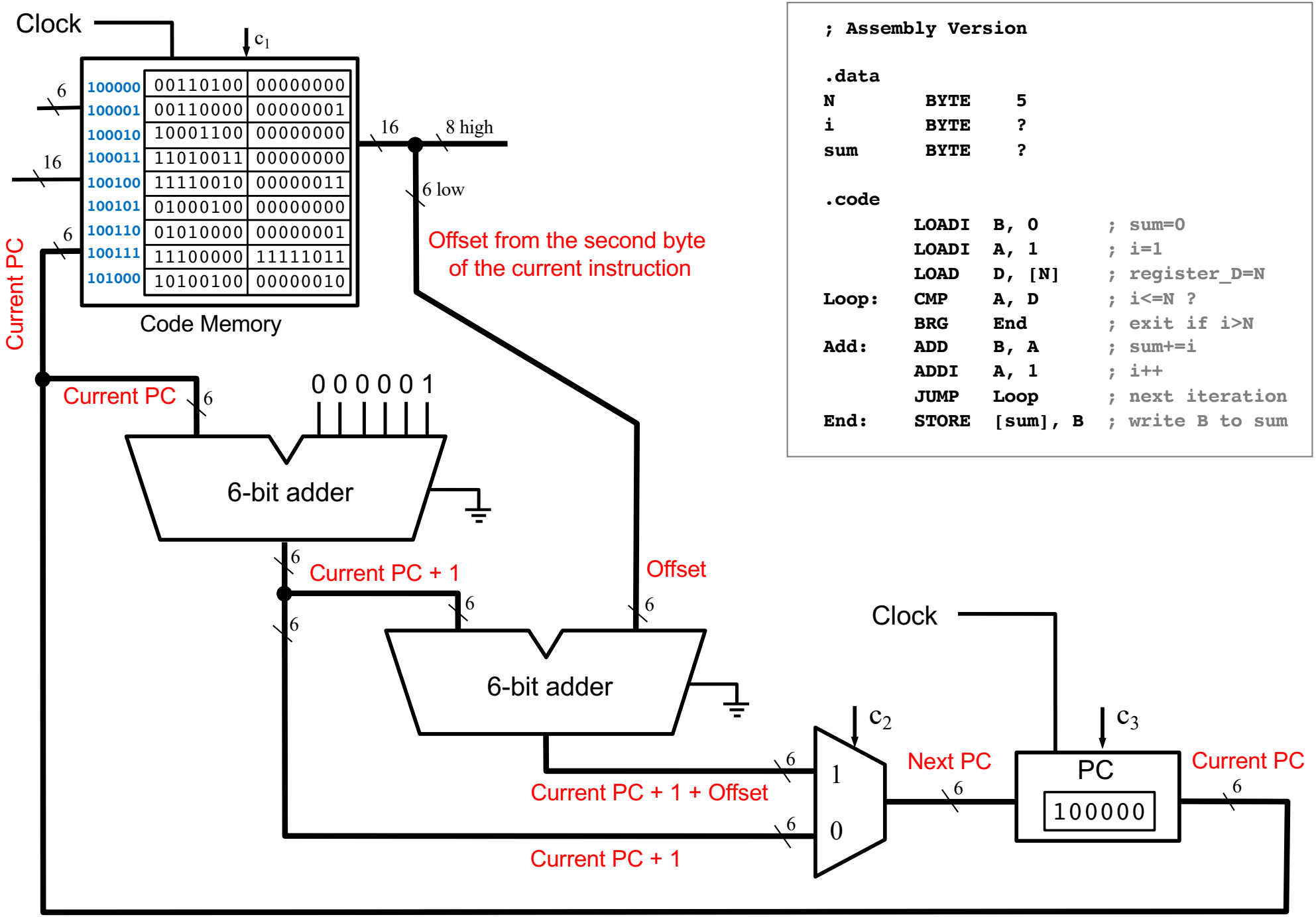
```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```


Clock
c1
6
16
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
Code Memory:
0011010000000000
0011000000000001
1000110000000000
1101001100000000
1111001000000011
0100010000000000
0101000000000001
1110000011111011
1010010000000010
000001
6-bit adder
6-bit adder
Clock
c2
c3
1
0
PC
100000

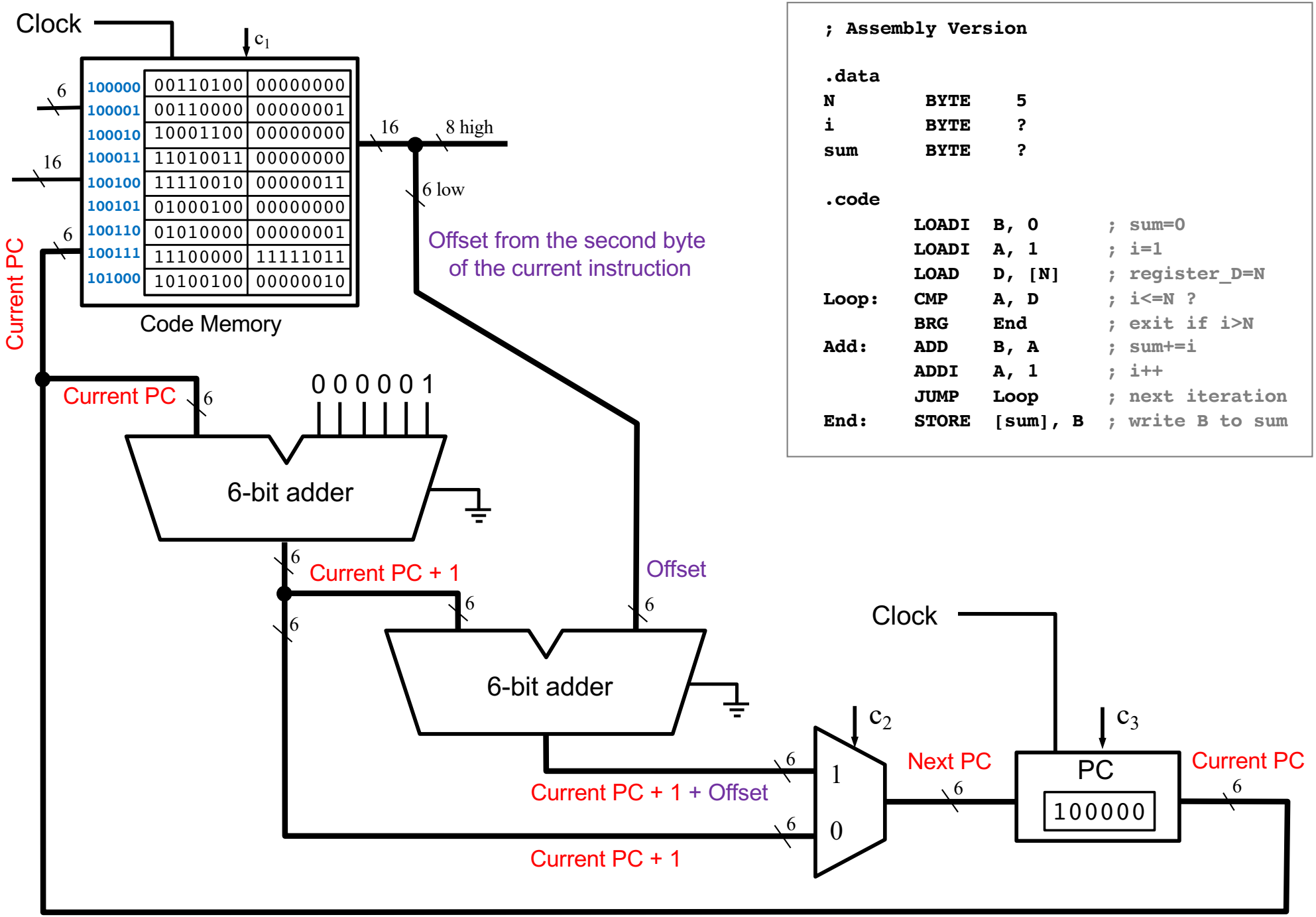
i281 CPU



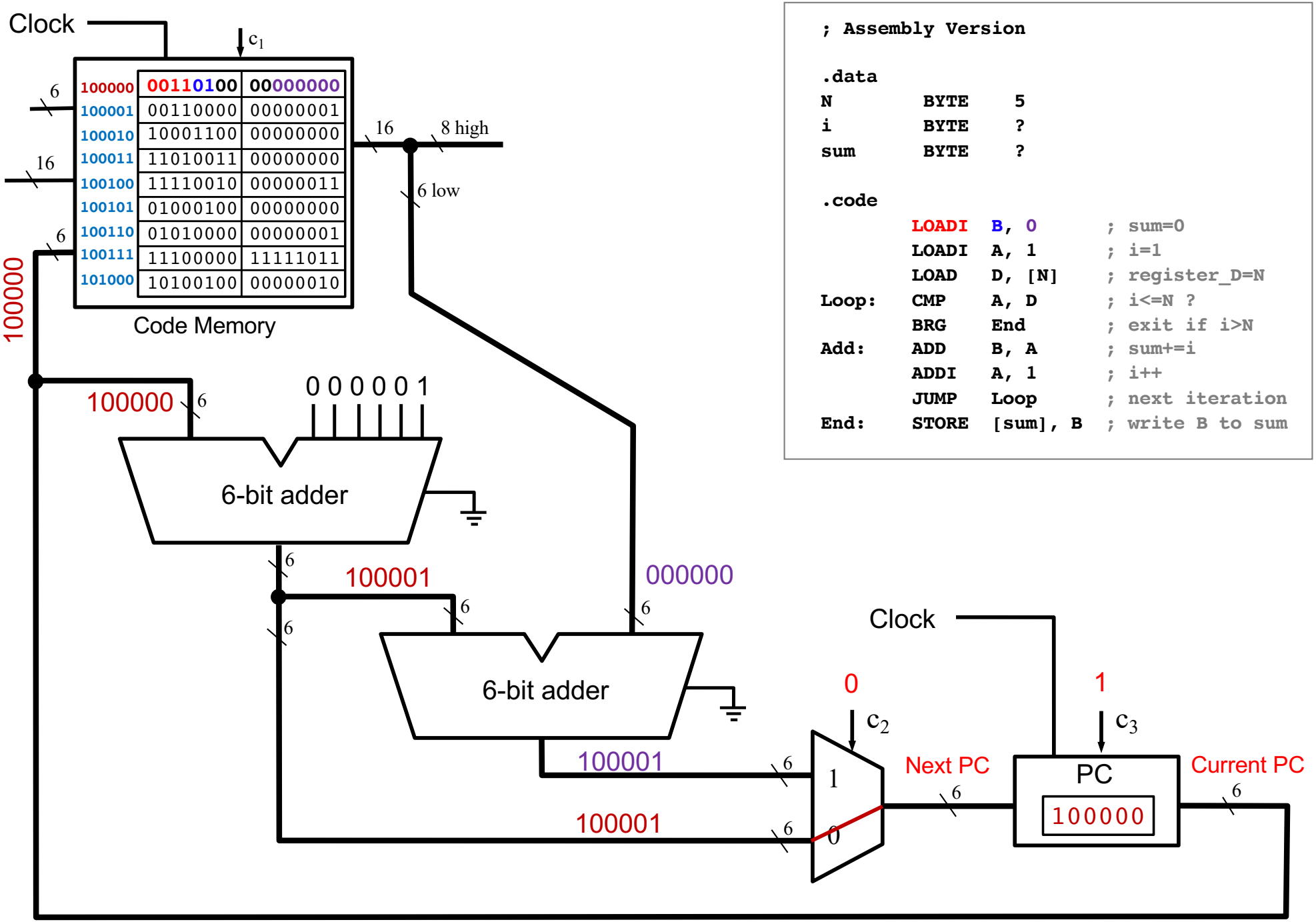

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```



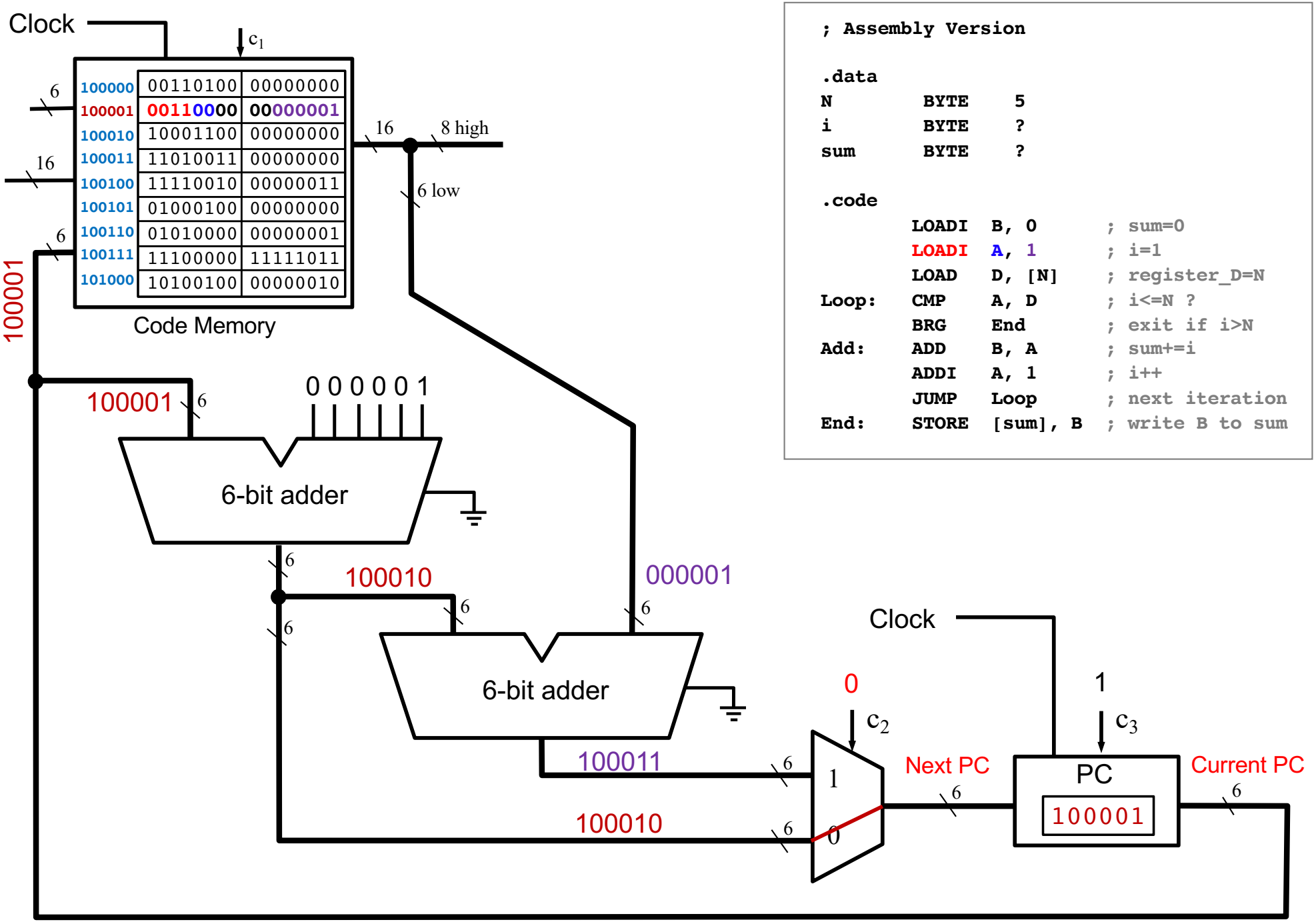

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```




```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

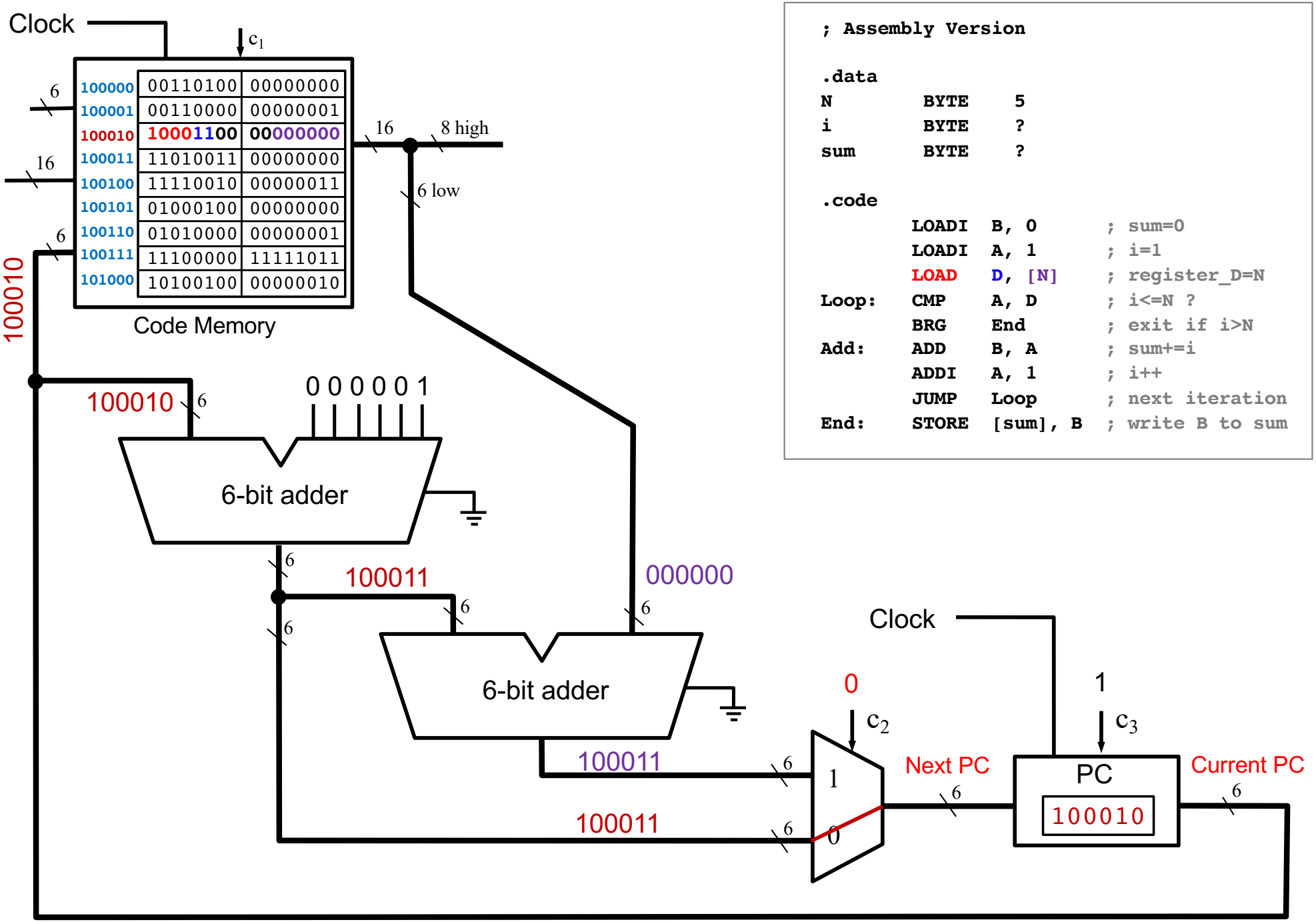



```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

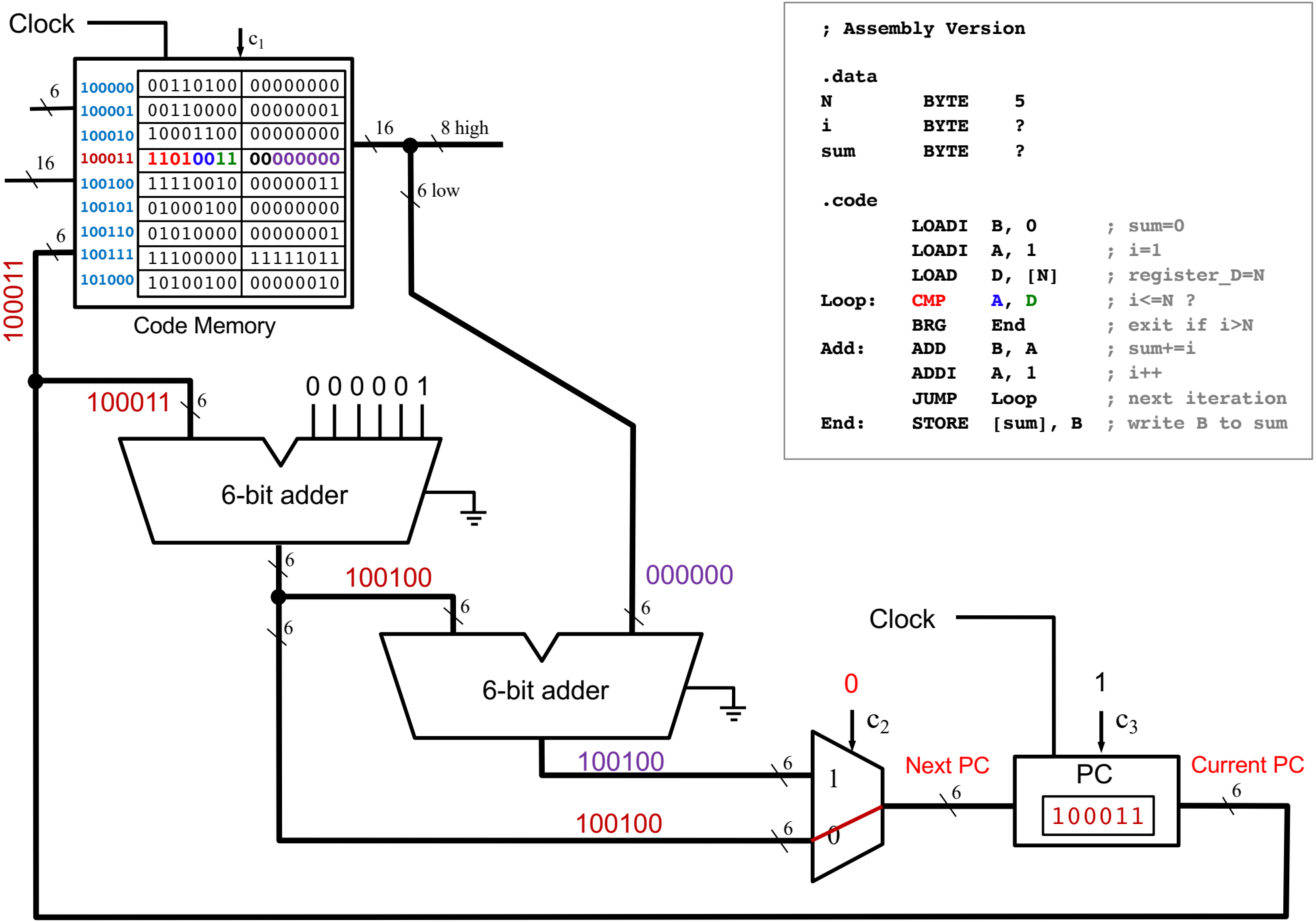



```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

i281 CPU

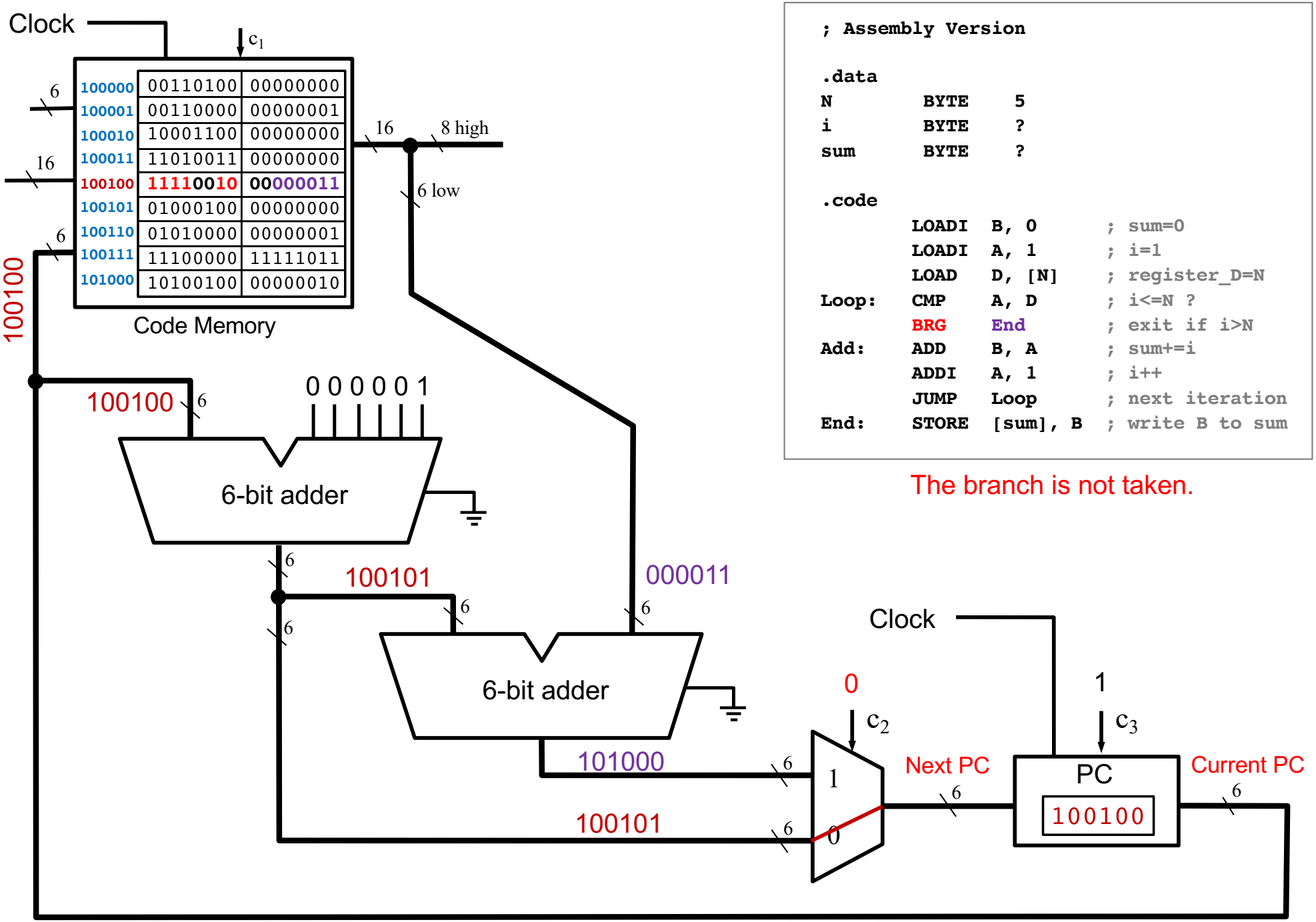



```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```



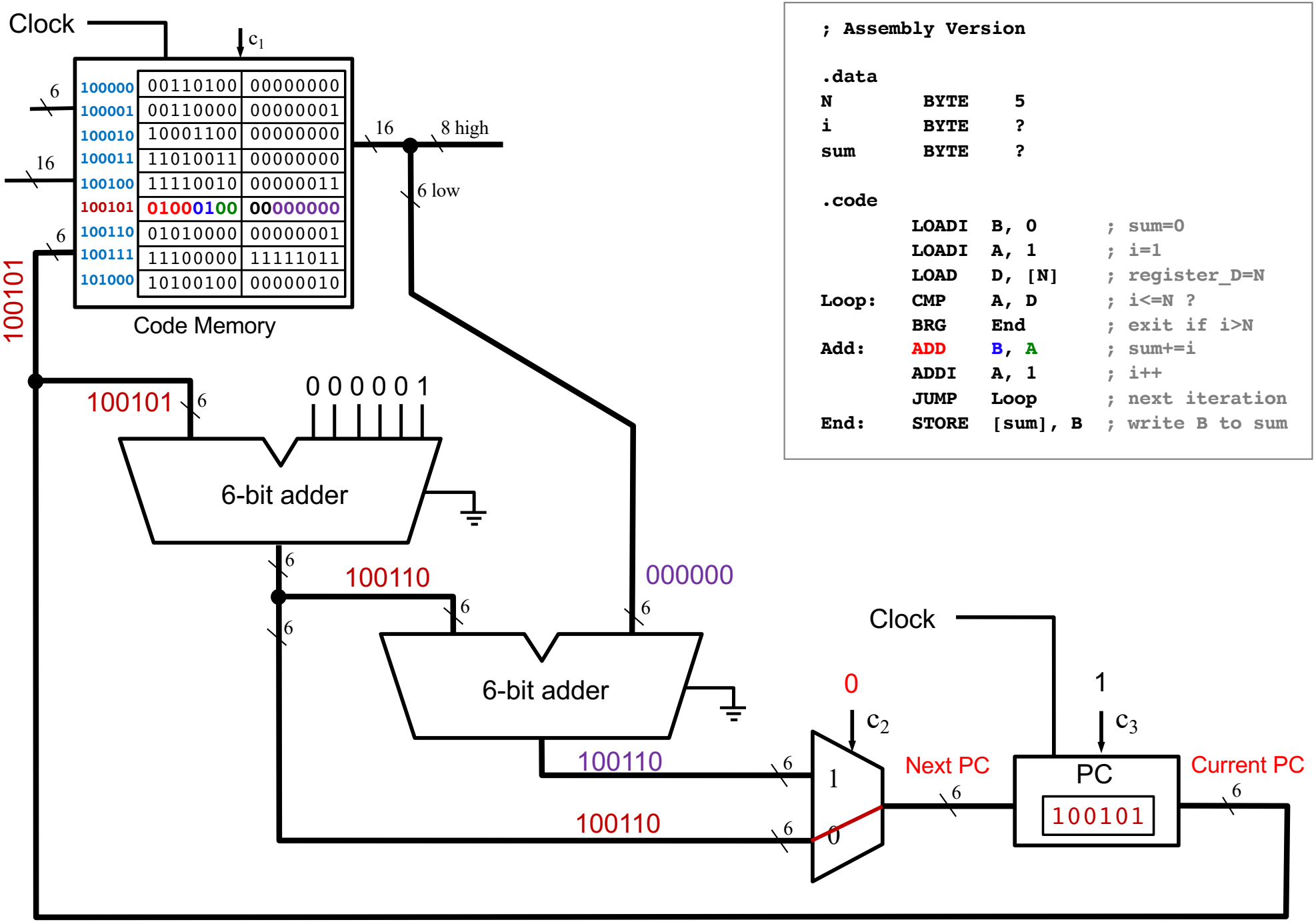

```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

i281 CPU

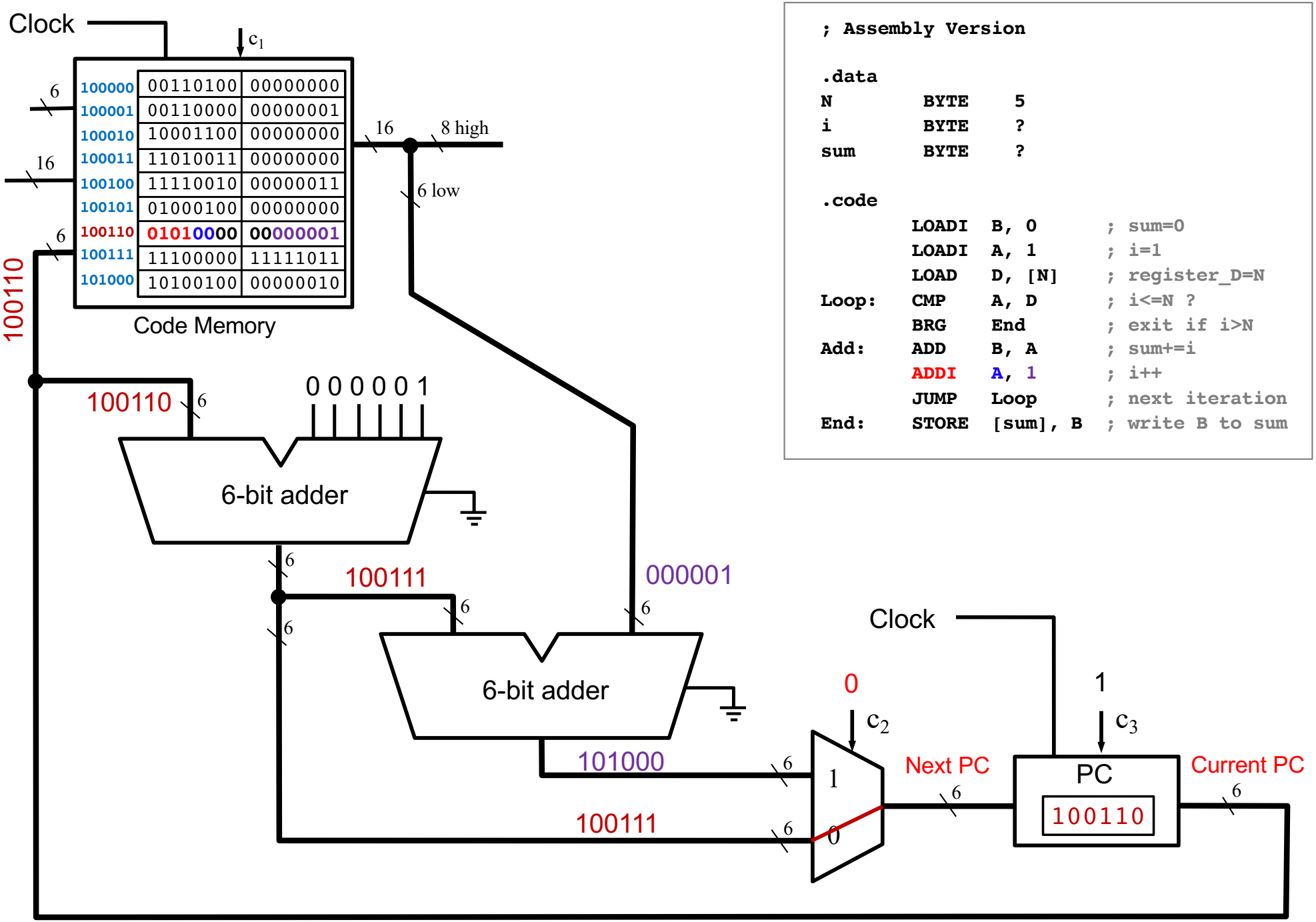



```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

The branch is not taken.

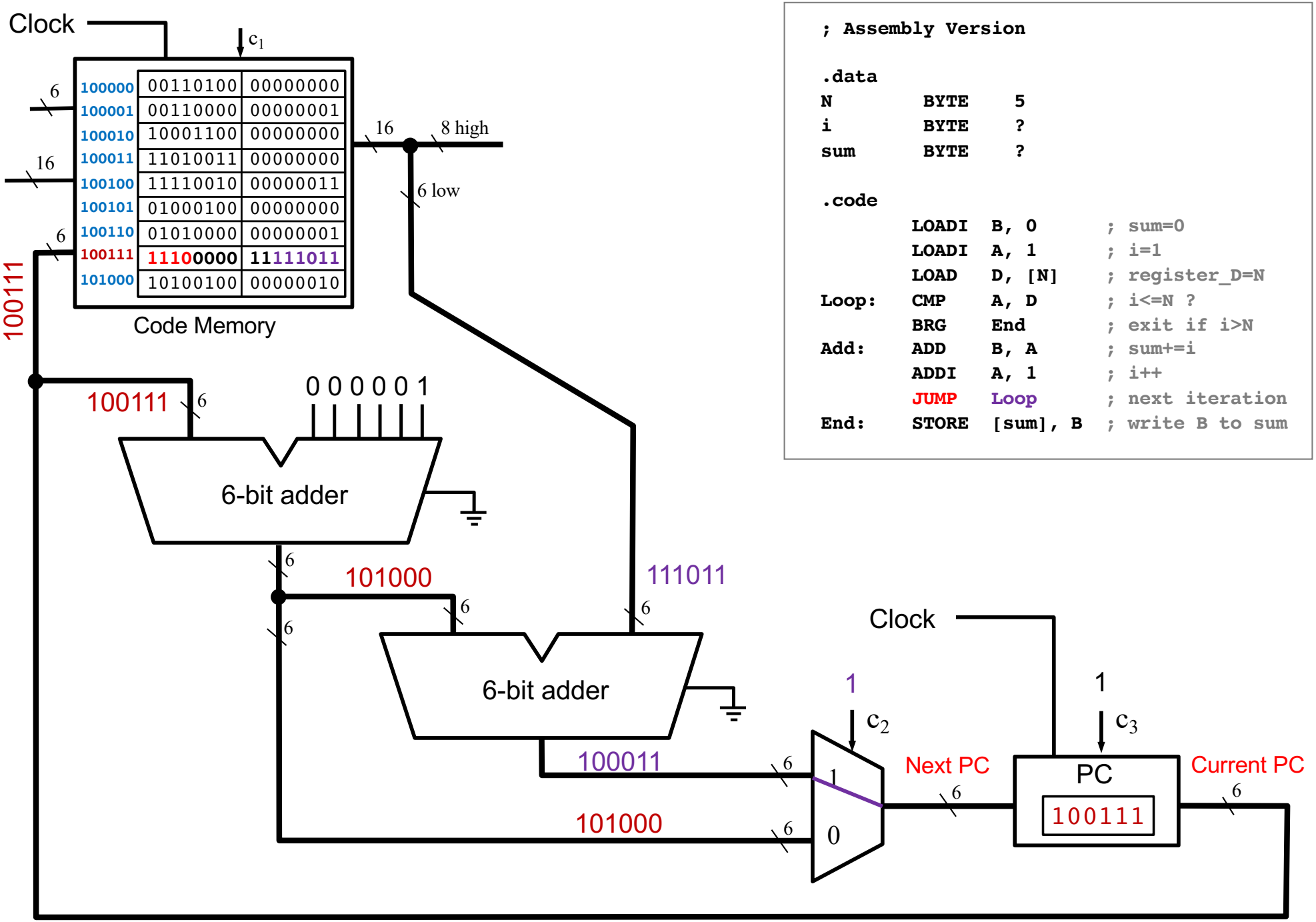



```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```



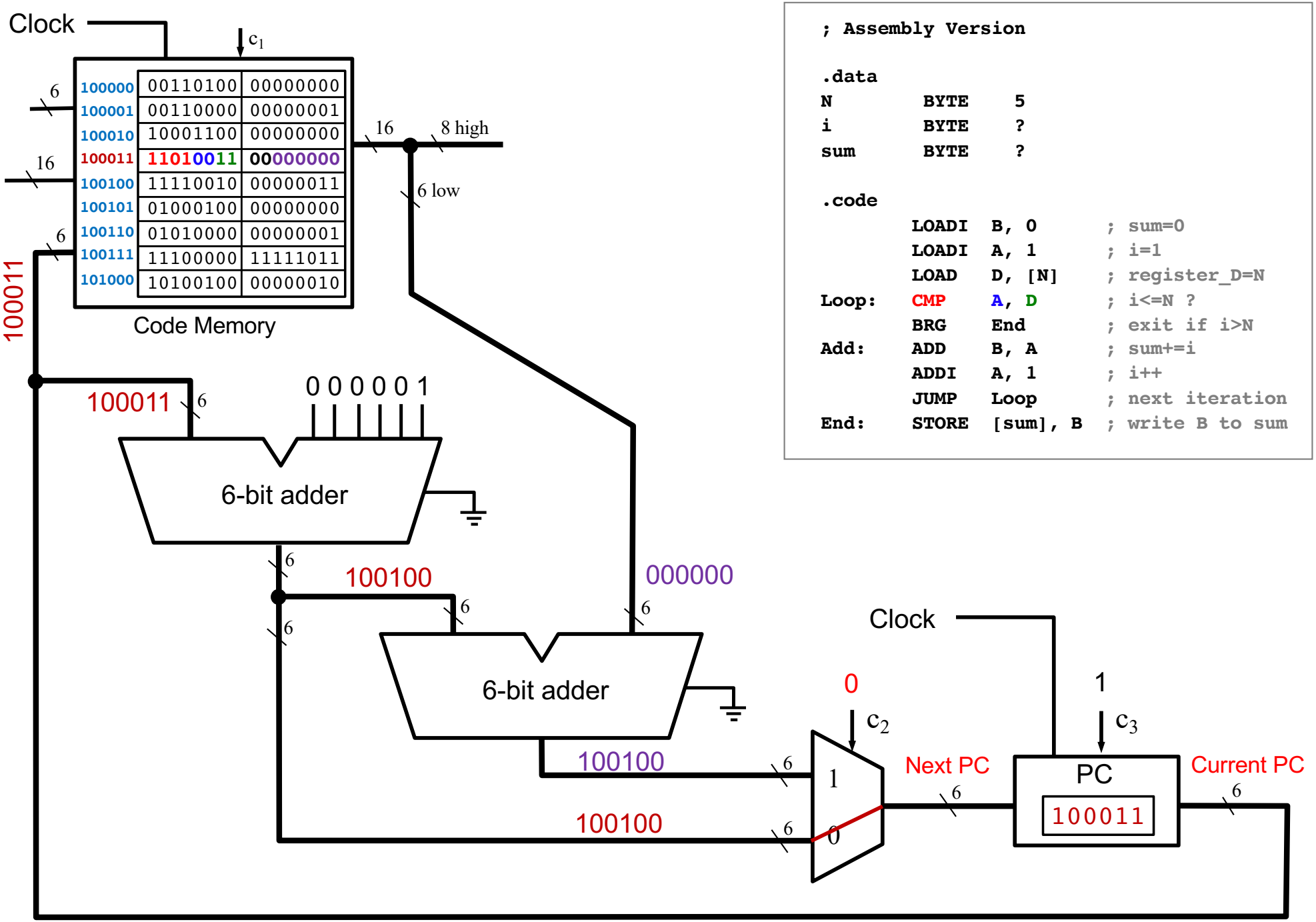

```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

i281 CPU



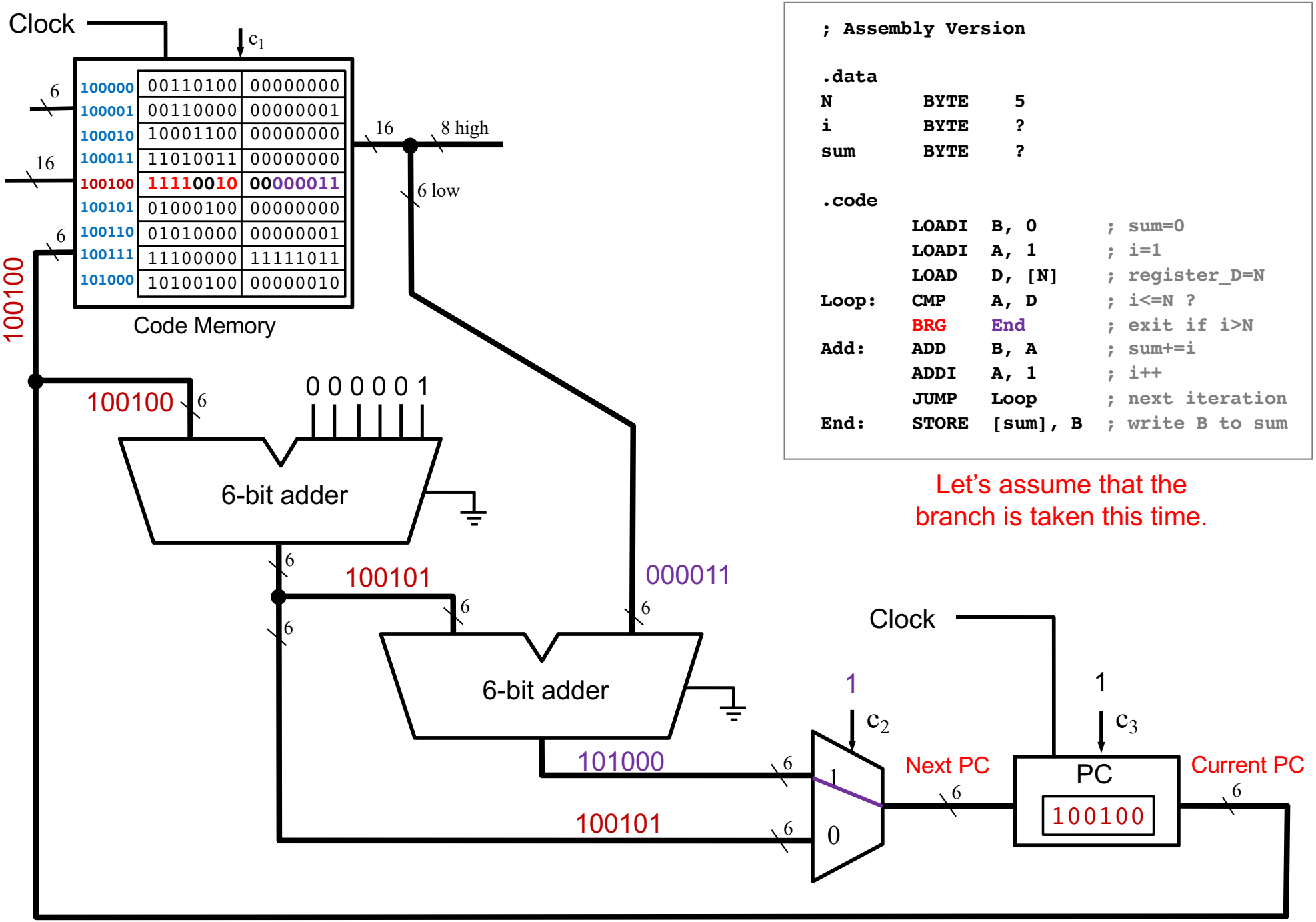

```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

i281 CPU



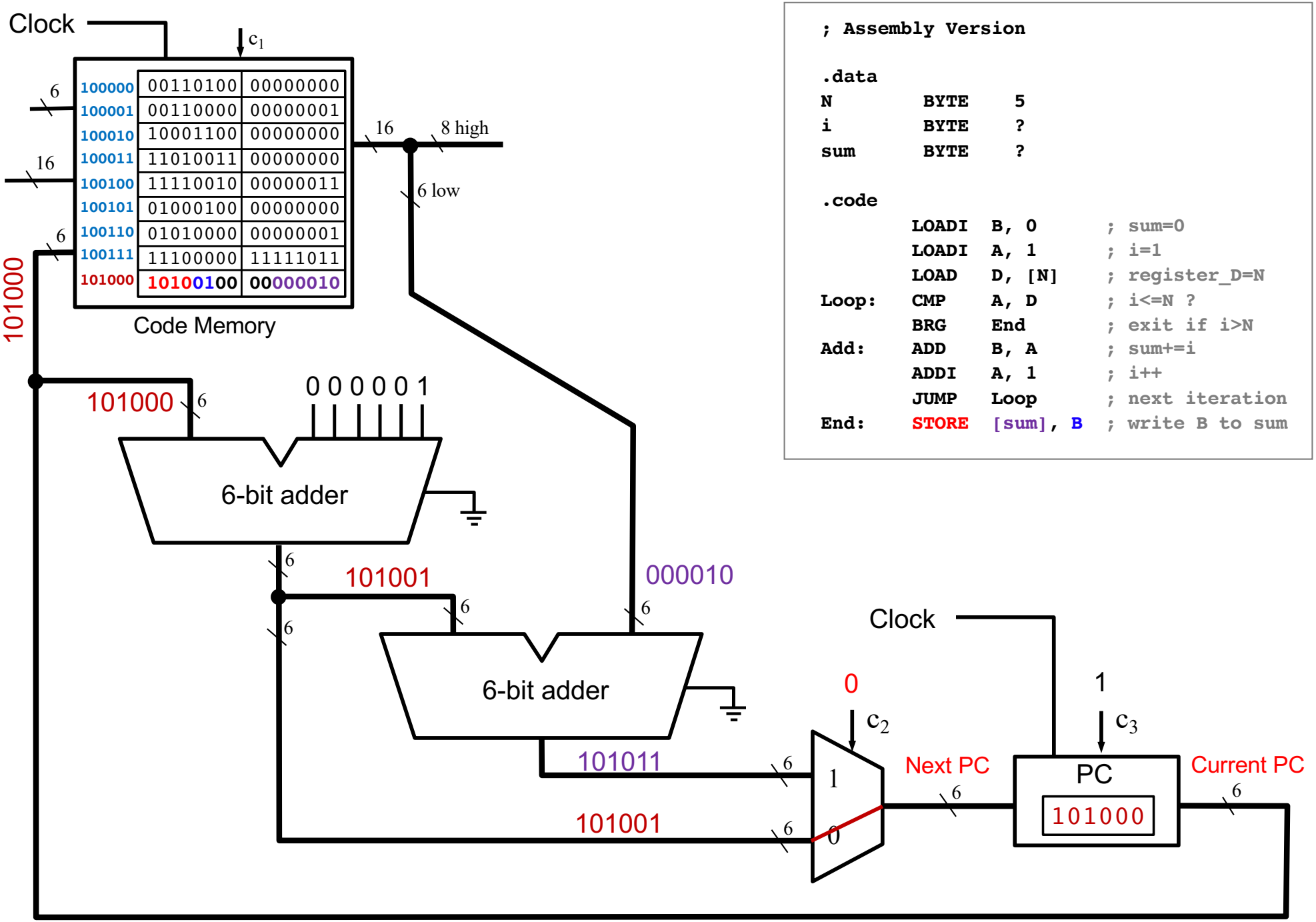

```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

i281 CPU

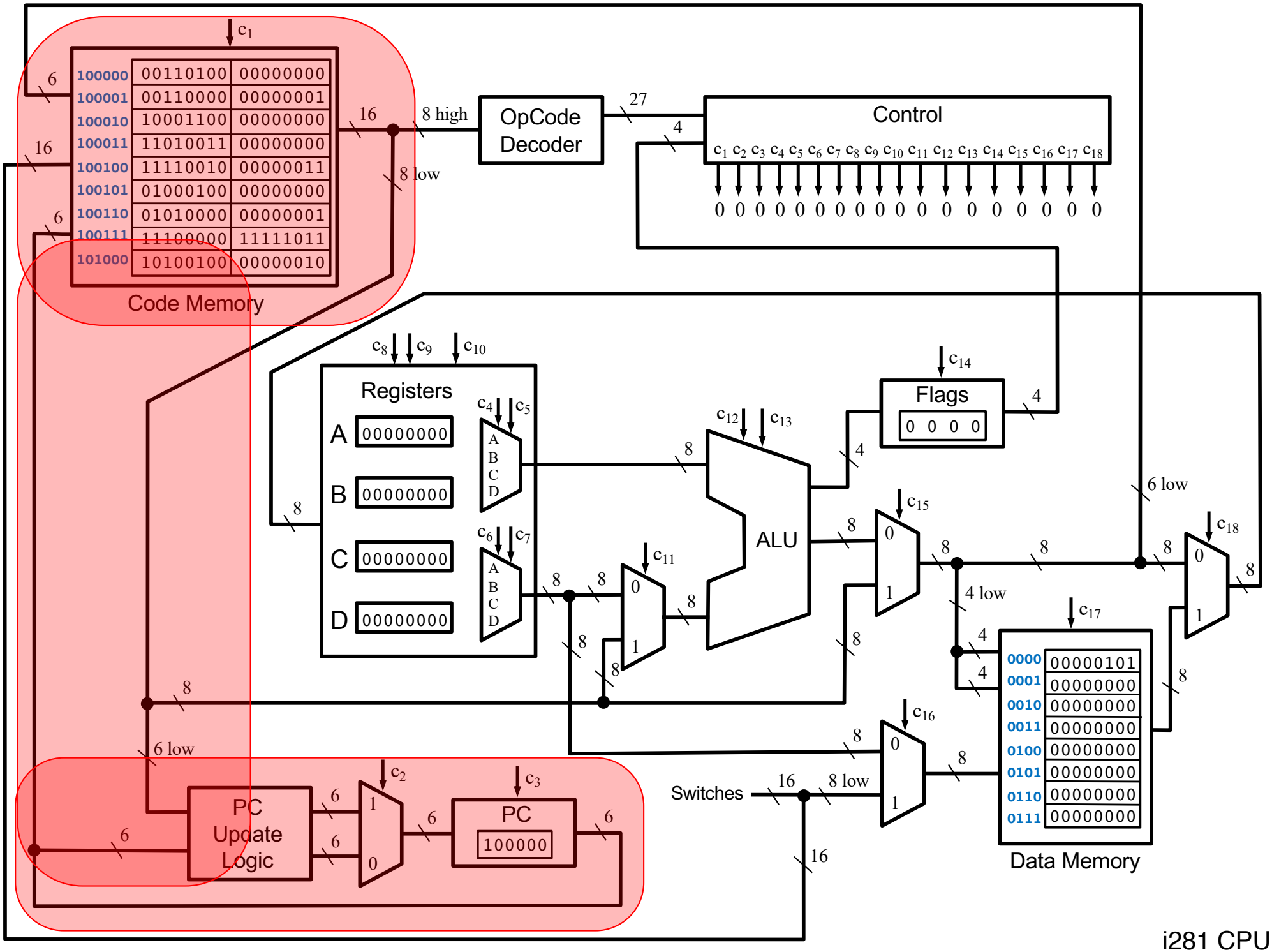


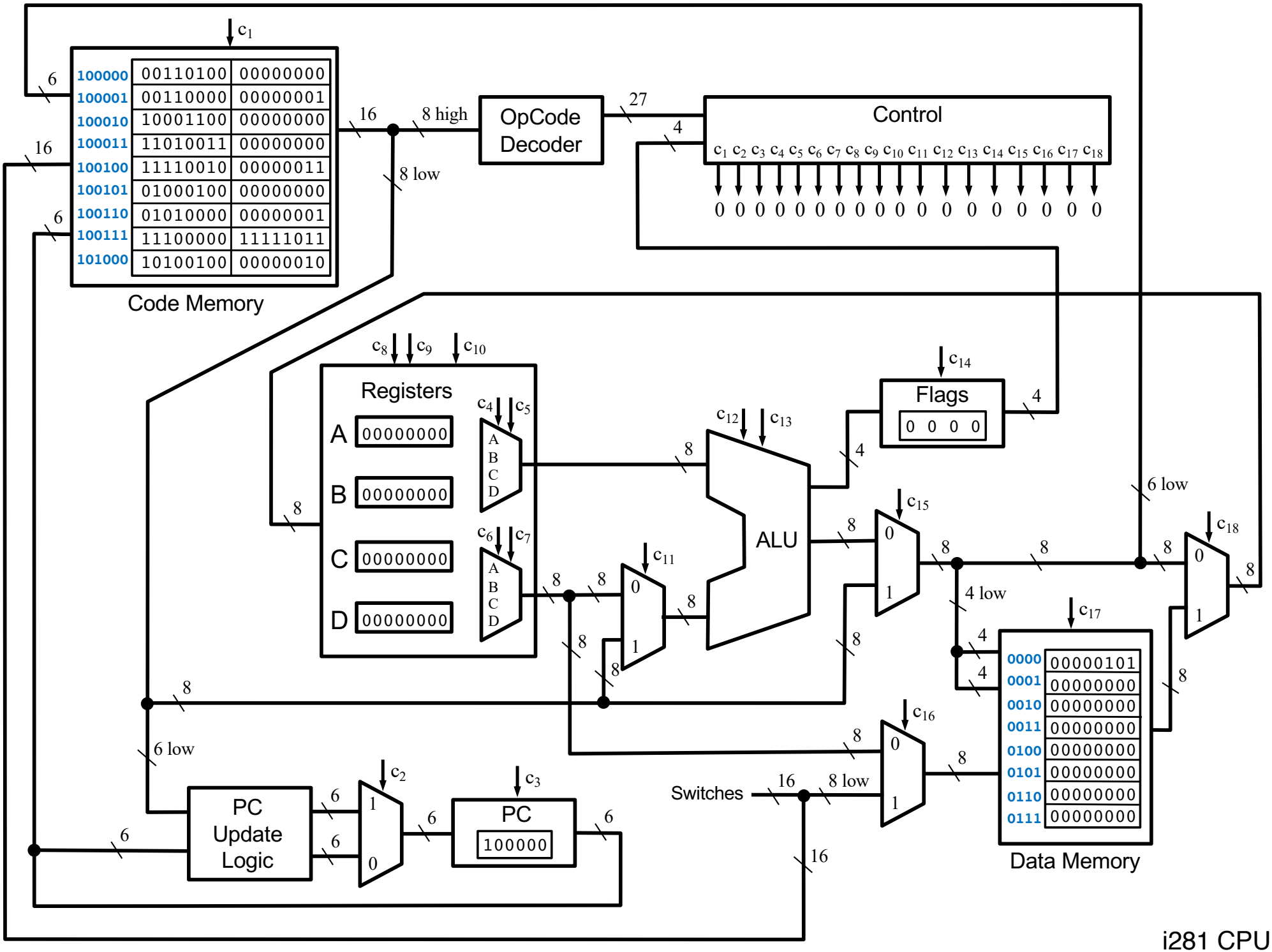

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

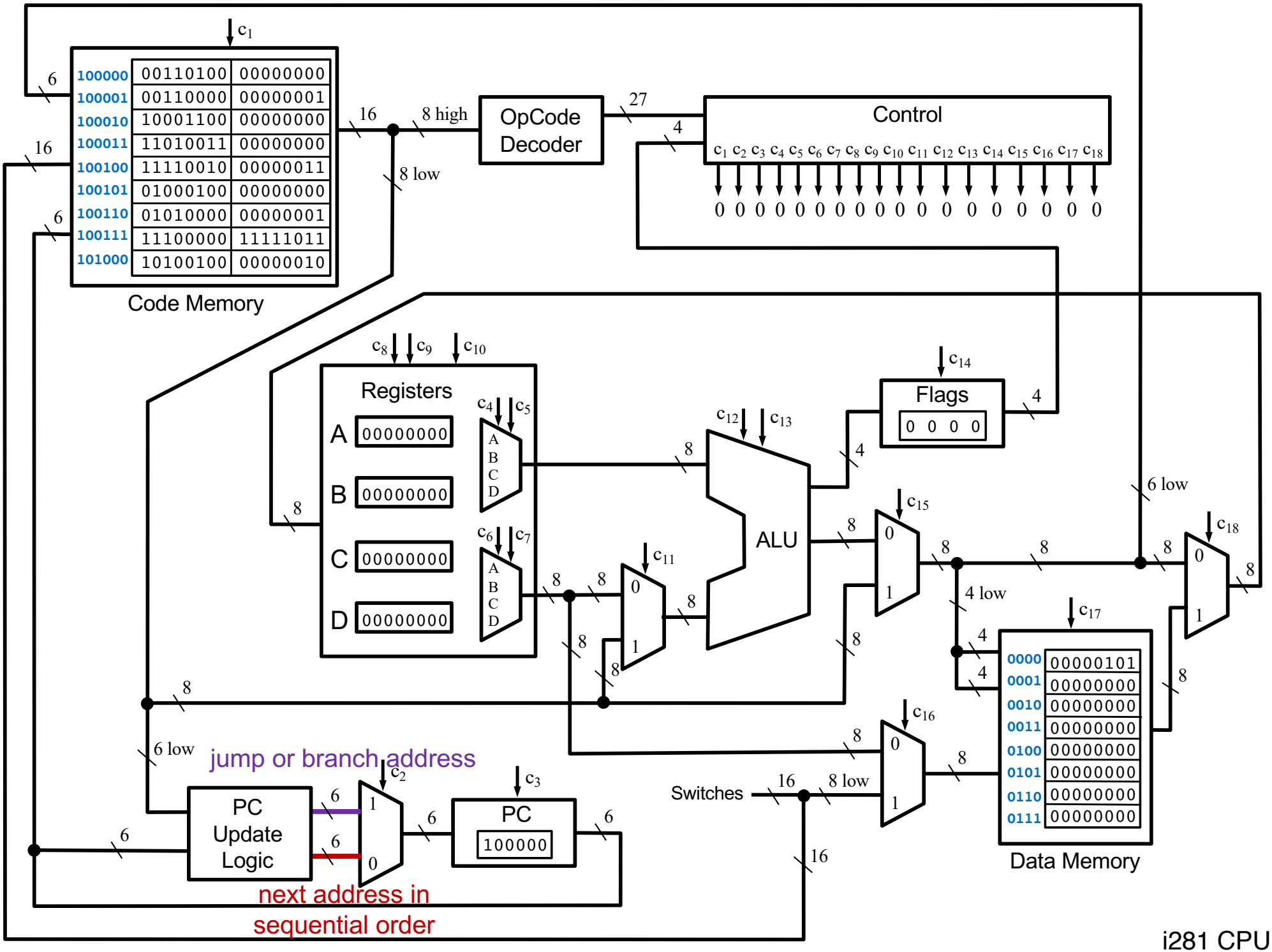
Let's assume that the branch is taken this time.




```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
16
6
Code Memory
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
16
8 low
PC Update Logic
c2
c3
PC
100000
6 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c11
c15
c16
c17
c18
6 low
4 low
Switches
16
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
6 low
i281 CPU


c1
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c14
Flags
0 0 0 0
c12 c13
ALU
c11
c15
c16
c18
6 low
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
jump or branch address
c2
c3
PC Update Logic
PC
100000
next address in sequential order
i281 CPU

Questions?

THE END