CENG 311
Computer Architecture

Lecture 4

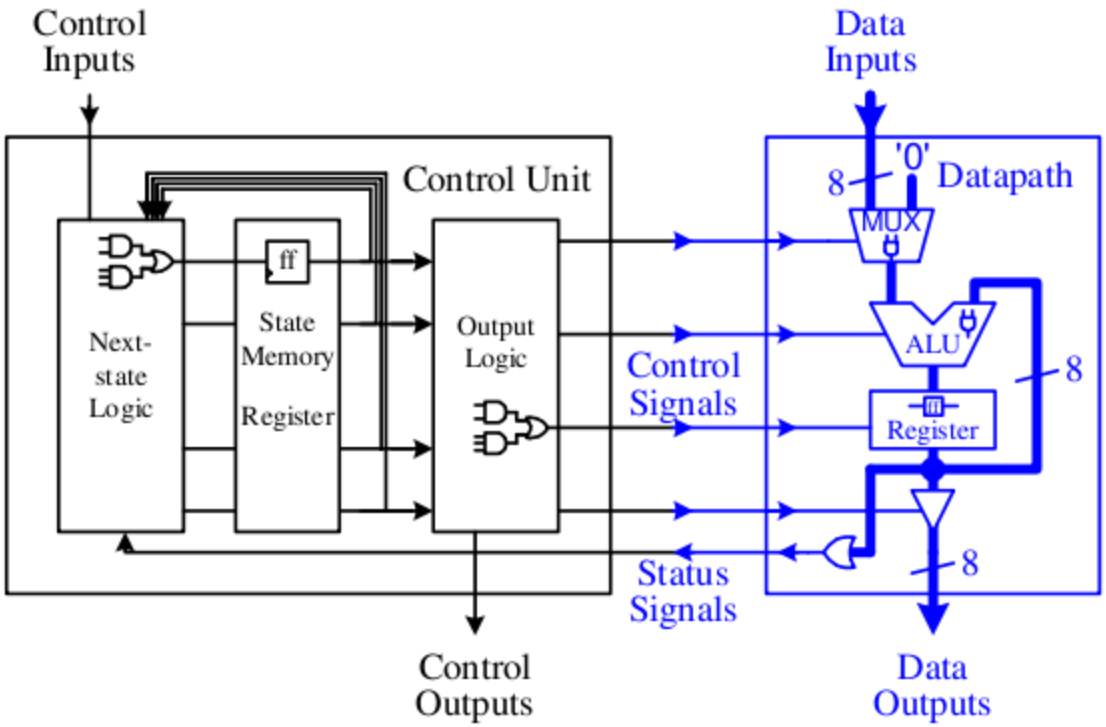
Single and General Purpose µP Design

Design of the Components: Datapath and Control Unit

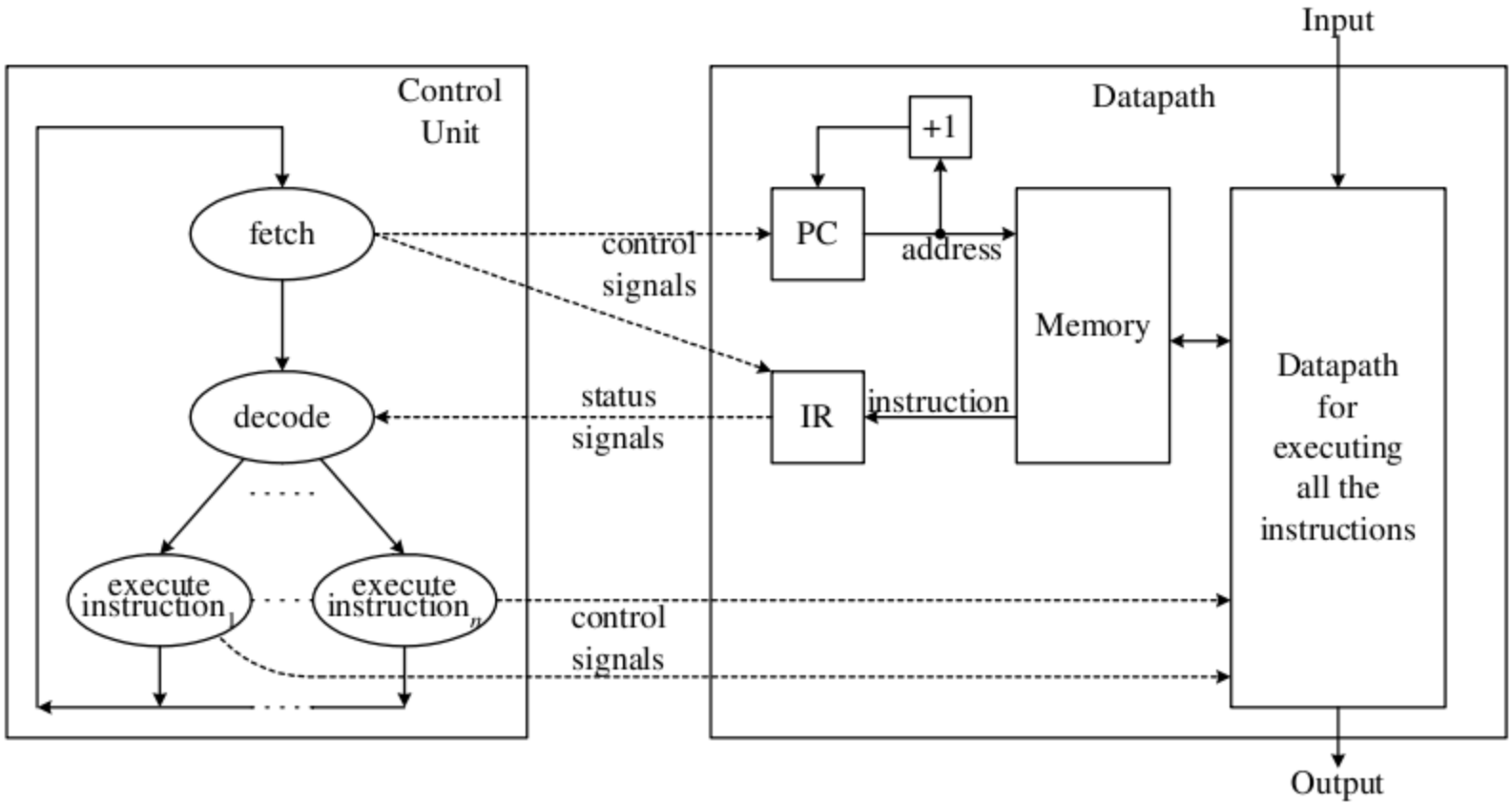
Asst. Prof. Tolga Ayav, Ph.D.

Department of Computer Engineering
İzmir Institute of Technology

General Structure of a Microprocessor



Overview of CPU Design

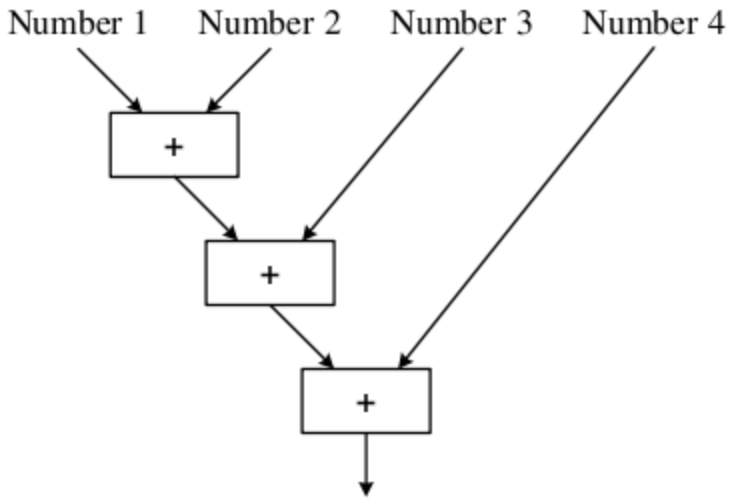


Datapath

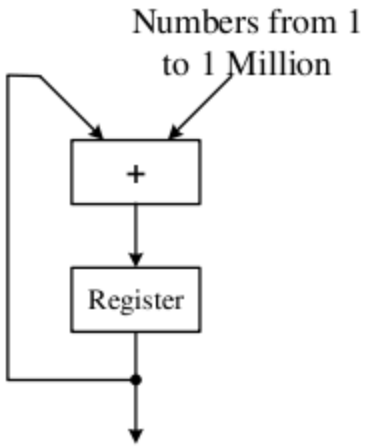
- Manipuates data. It includes:
  - Functional units: Adder, shifter, multiplier, ALU, comparator
  - Registers and other memory elements for the temporary storage of the data
  - Buses, multiplexers and tri-state buffers for the transfer of data between the different components in datapath and the external world.

Why do we use datapath?

- how do we design a circuit for performing more complex data operations or operations that involve multiple steps?



combinational circuit to add four numbers;



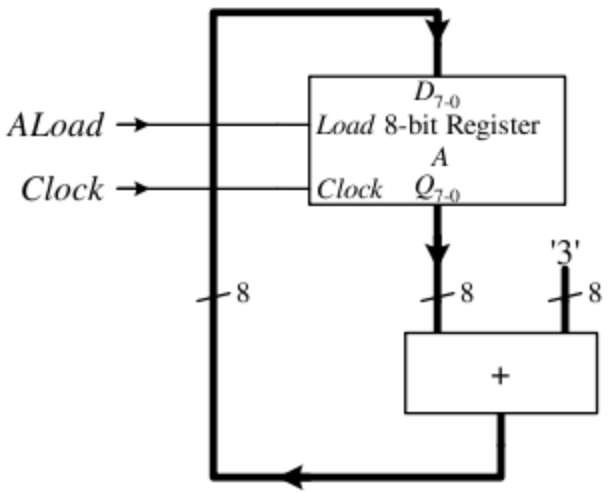
datapath to add one million numbers.

Designing Deticated Datapaths

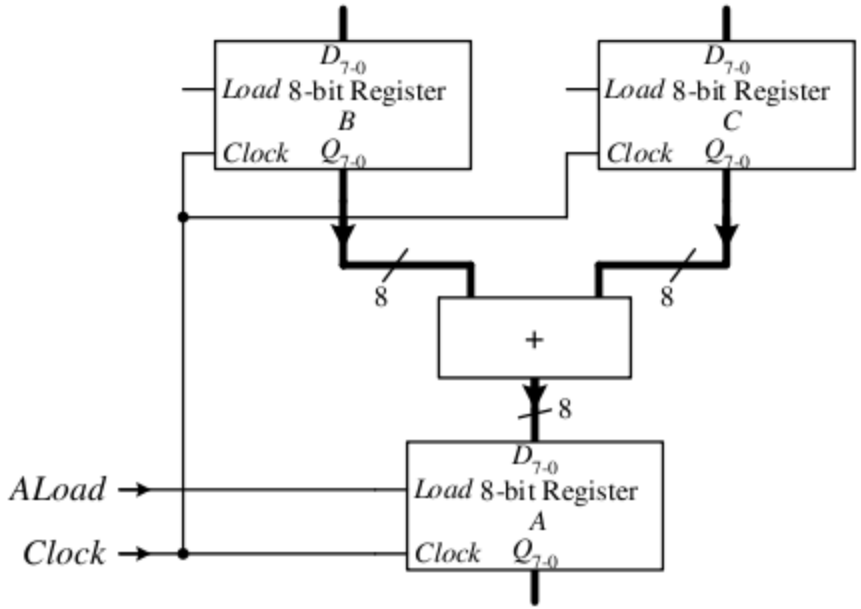
- What kind of registers to use, and how many are needed?
- What kind of functional units to use, and how many are needed?
- Can a certain functional unit be shared between two or more operations?
- How are the registers and functional units connected together so that all of the data movements specified by the algorithm can be realized?

Two sample dedicated datapaths

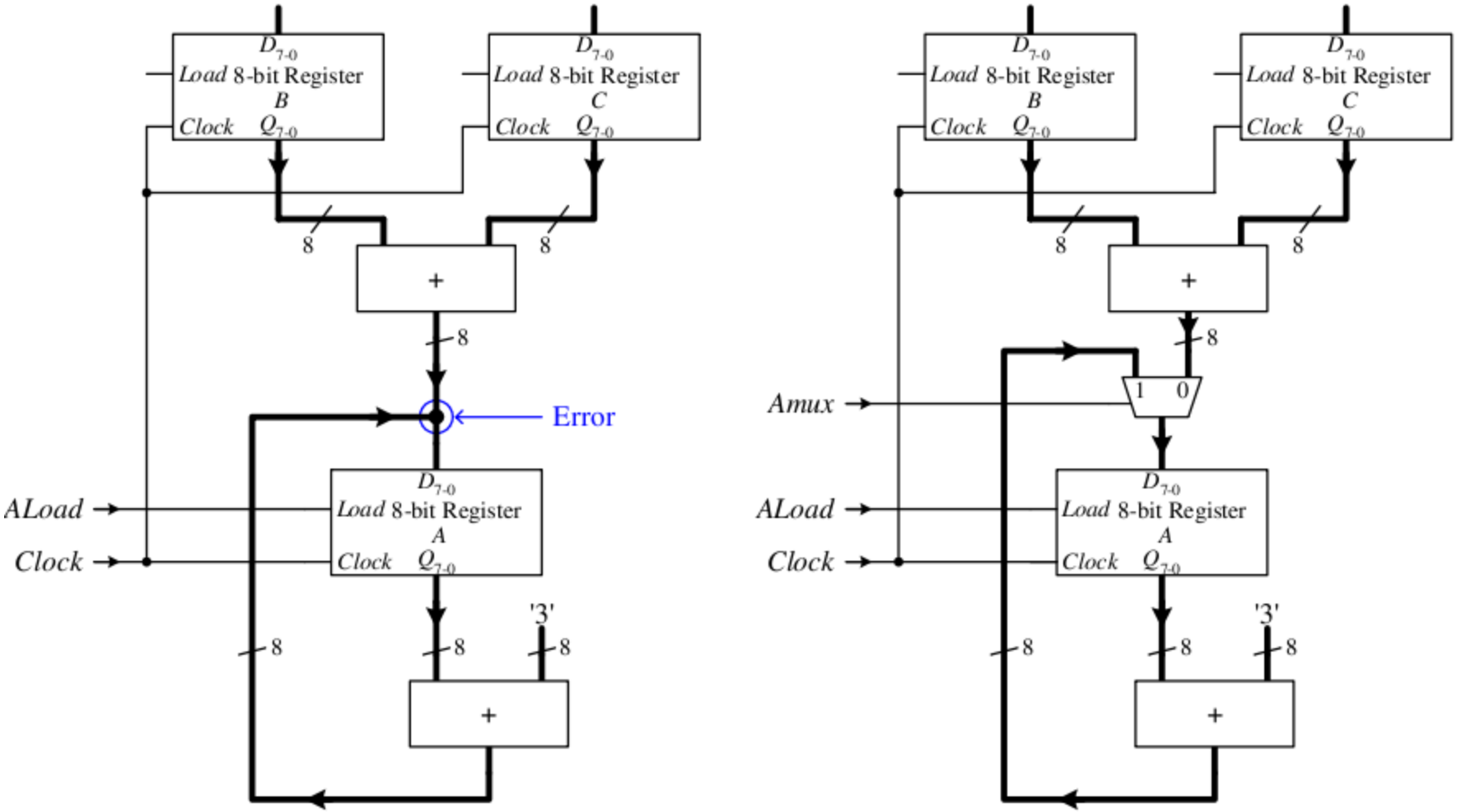
A = A + 3



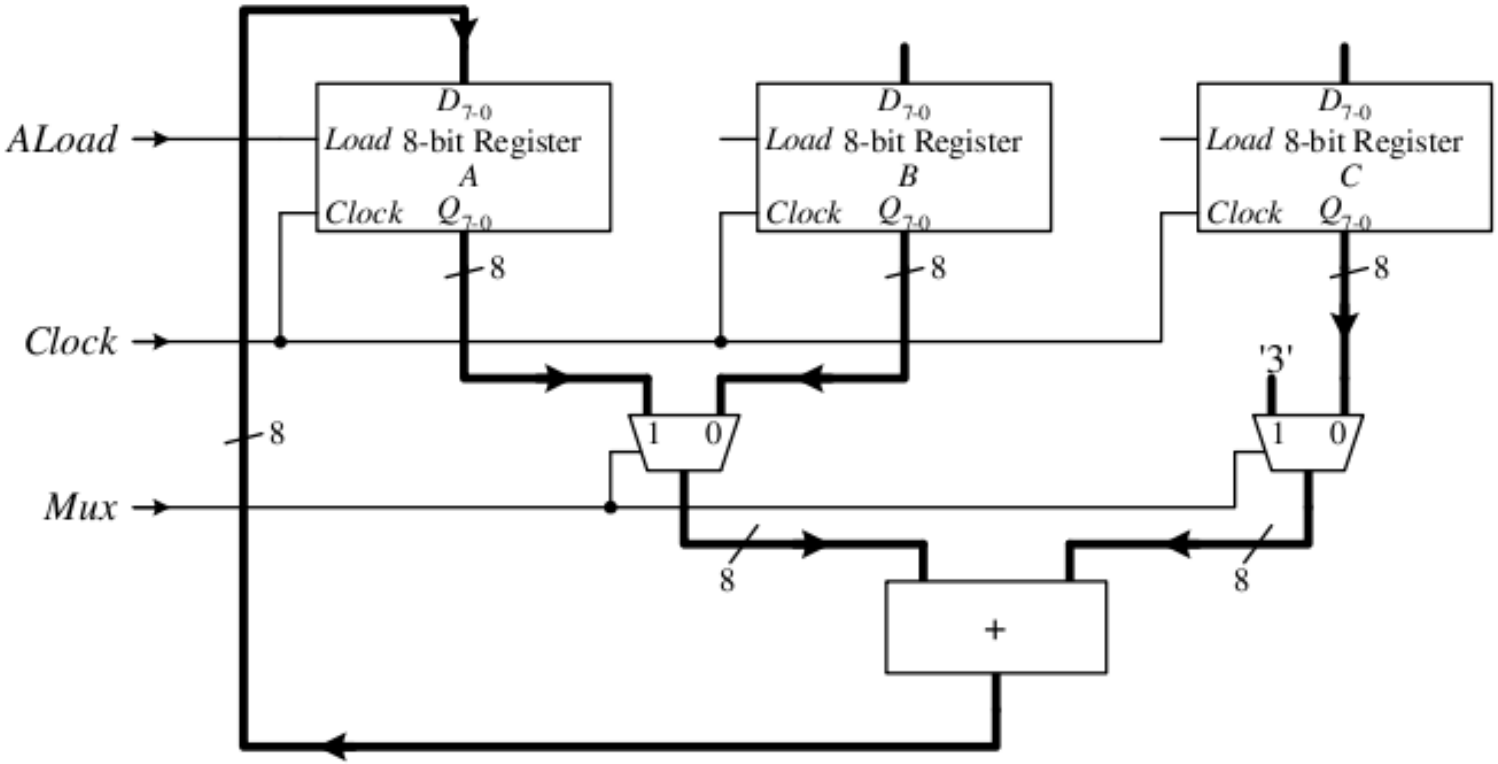
A = B + C



Use of multiplexers



Datapath performing both A=A+3 and A=B+C (we use two adders).

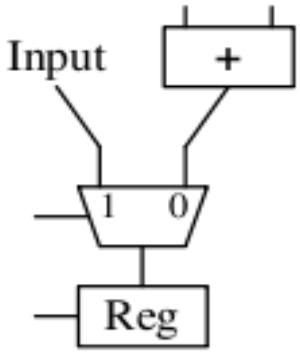


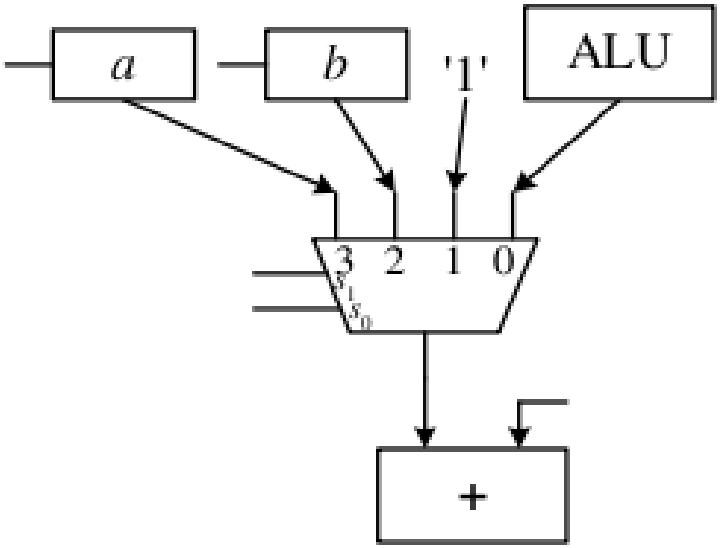
Datapath performing both A=A+3 and A=B+C (we use only one adder).

Data Transfer

Multiple destinations is not a problem. We can connect all of the destinations to the same source (Pay attention to the fan-out).

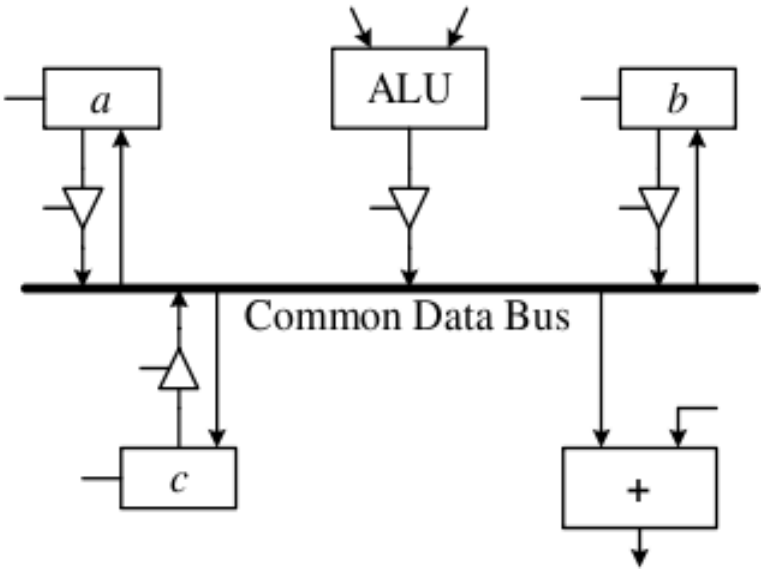
In case of multiple sources, multiplexers are used.





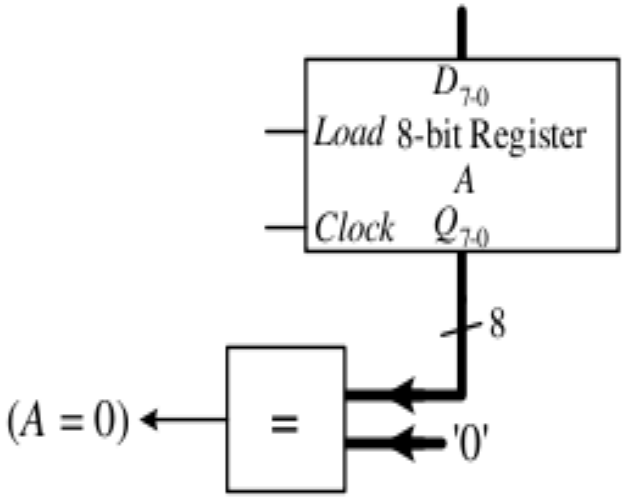
Tri-state Bus

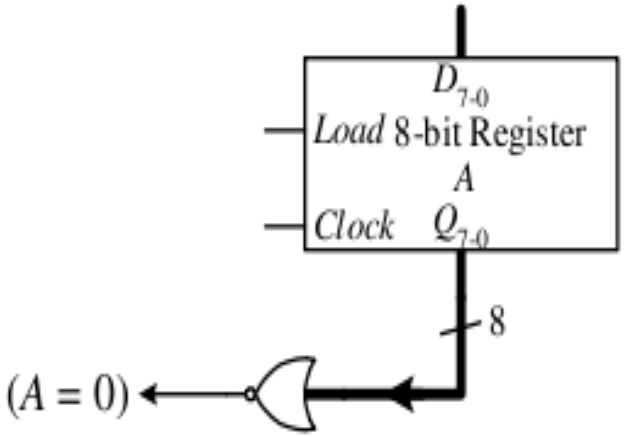
Another scheme where multiple sources and destinations can be connected to the same data bus is through the use of tri-state buffers.



Generating Status Signals

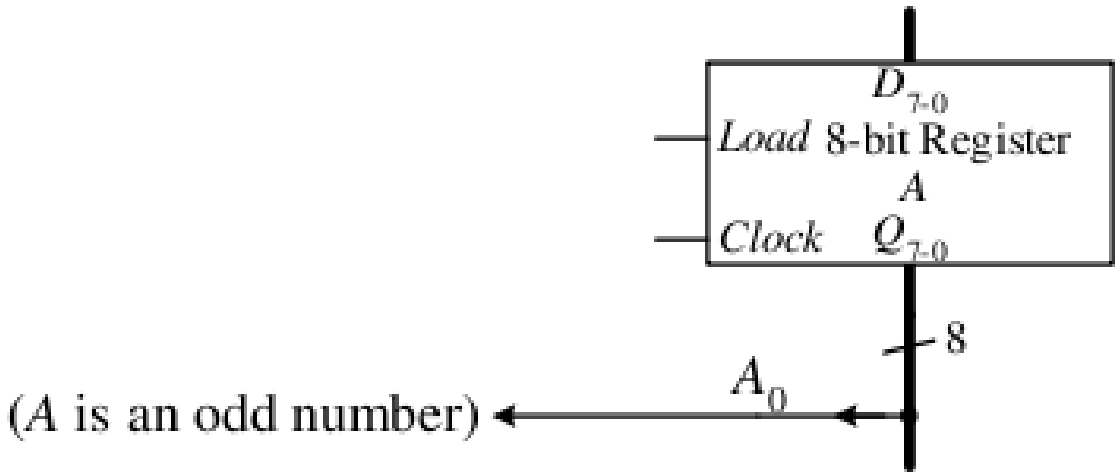
Status signals are the results of the conditional tests that the datapath supplies to the control unit.





Example: Generating Status Signals

IF (A is an odd number) THEN ...



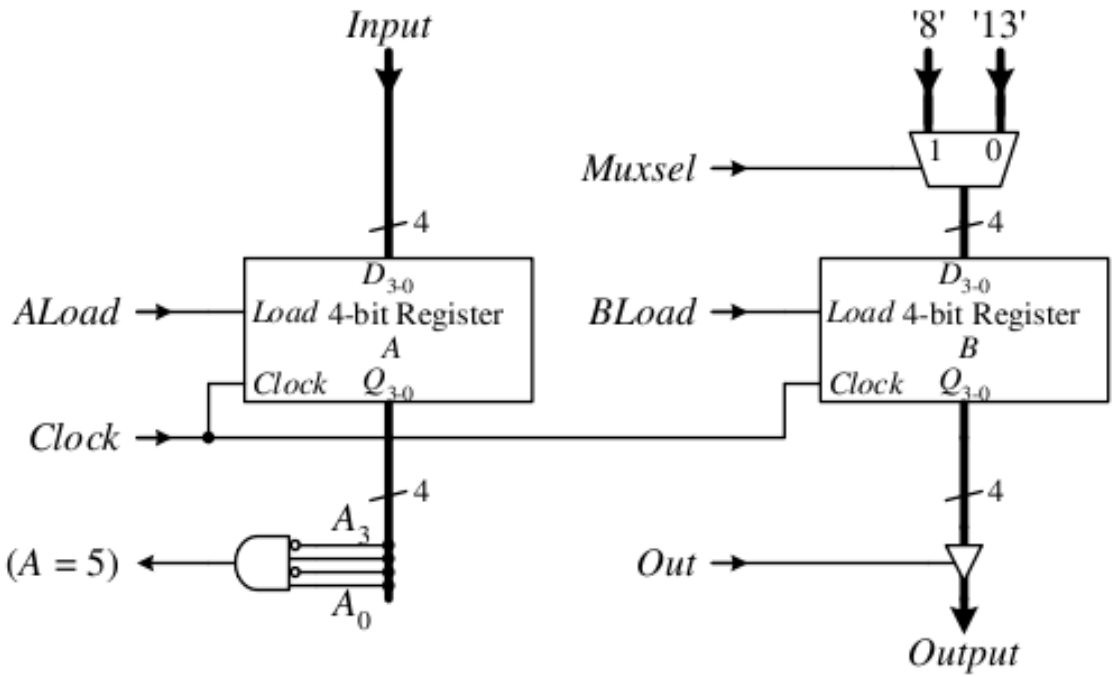
Examples of Dedicated Datapaths

- Simple if-then-else
- Counting 1 to 10
- Summation of n downto 1
- Factorial of n

Simple If-Then-Else

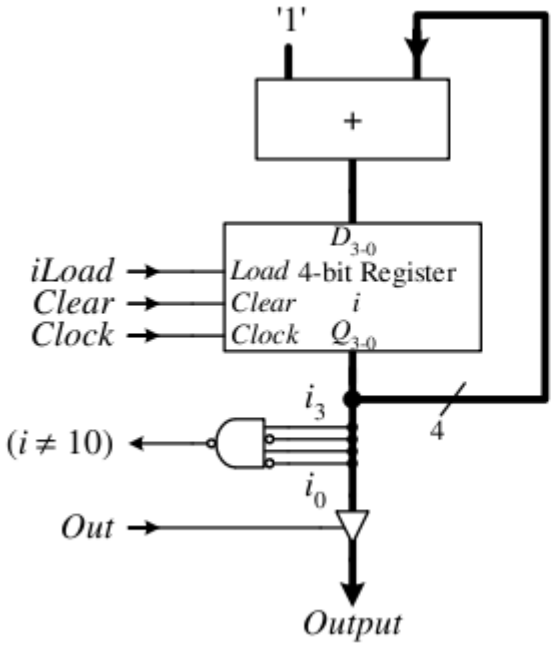
```
1          INPUT A
2          IF (A = 5) THEN
3             B = 8
4          ELSE
5             B = 13
6          END IF
7          OUTPUT B
```

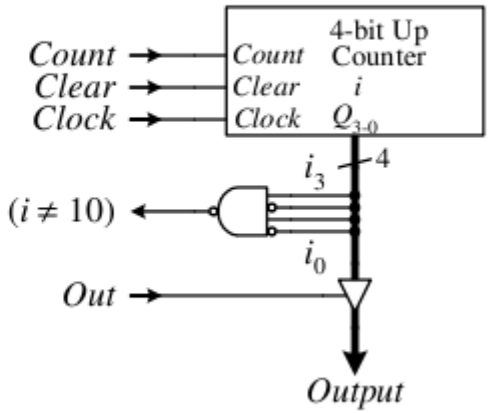
Control Word	Instruction	*ALoad*	*Muxsel*	*BLoad*	*Out*
1	INPUT *A*	1	×	0	0
2	*B* = 8	0	1	1	0
3	*B* = 13	0	0	1	0
4	OUTPUT *B*	0	×	0	1



Counting 1 to 10

```
1          i = 0
2          WHILE (i ≠ 10){
3             i = i + 1
4             OUTPUT i
5             }
```



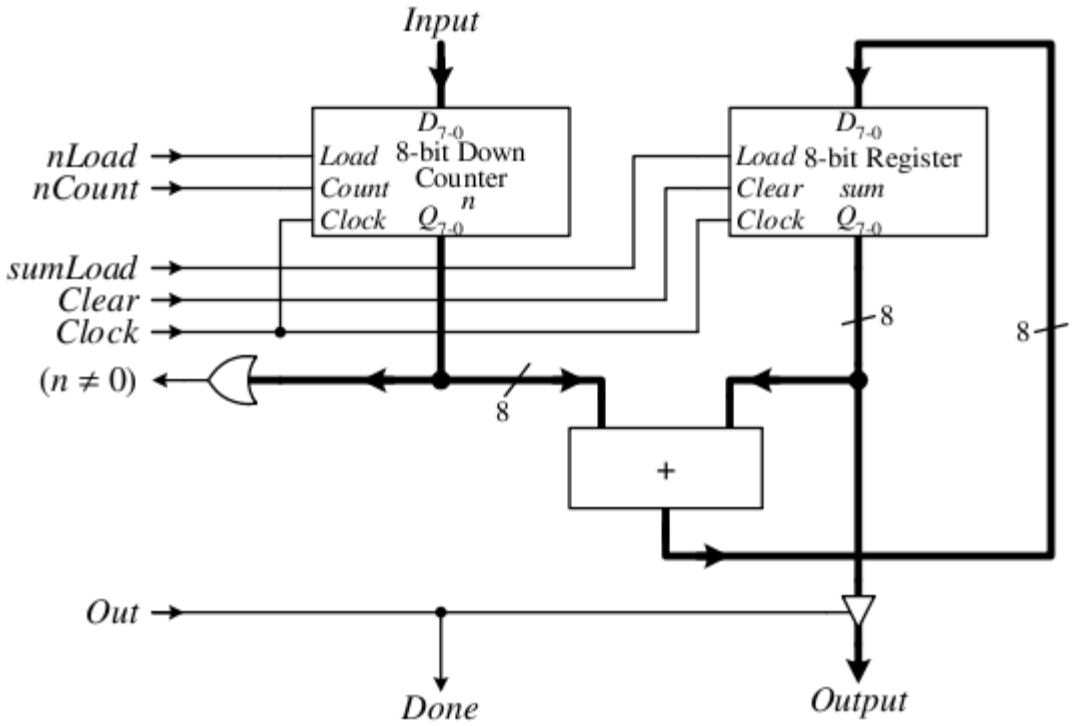


Control Word	Instruction	*iLoad*	*Clear*	*Out*
1	$i = 0$	0	1	0
2	$i = i + 1$	1	0	0
3	OUTPUT i	0	0	1

Control Word	Instruction	*Count*	*Clear*	*Out*
1	$i = 0$	0	1	0
2	$i = i + 1$	1	0	0
3	OUTPUT i	0	0	1

Summation of *n* Downto 1

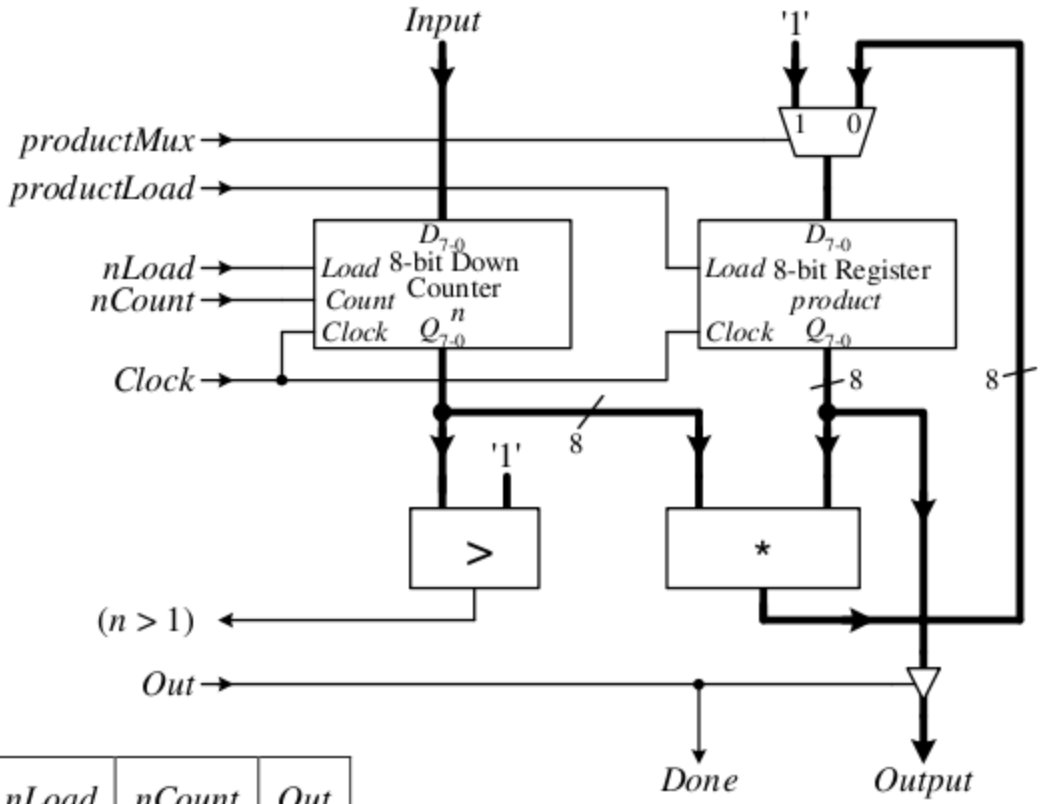
```
1          sum = 0
2          INPUT n
3          WHILE (n ≠ 0){
4             sum = sum + n
5             n = n - 1
6          }
7          OUTPUT sum
```



Control Word	Instruction	*nLoad*	*nCount*	*sumLoad*	*Clear*	*Out*
1	*sum* = 0	0	0	0	1	0
2	INPUT *n*	1	0	0	0	0
3	*sum* = *sum* + *n*	0	0	1	0	0
4	*n* = *n* – 1	0	1	0	0	0
5	OUTPUT *sum*	0	0	0	0	1

Factorial of *n*

```
1           INPUT n
2           product = 1
3           WHILE (n > 1) {
4              product = product * n
5              n = n - 1
            }
6           OUTPUT product
```



Control Word	Instruction	*productMux*	*productLoad*	*nLoad*	*nCount*	*Out*
1	INPUT *n*	×	0	1	0	0
2	*product* = 1	1	1	0	0	0
3	*product* = *product* * *n*	0	1	0	0	0
4	*n* = *n* – 1	×	0	0	1	0
5	OUTPUT *product*	×	0	0	0	1

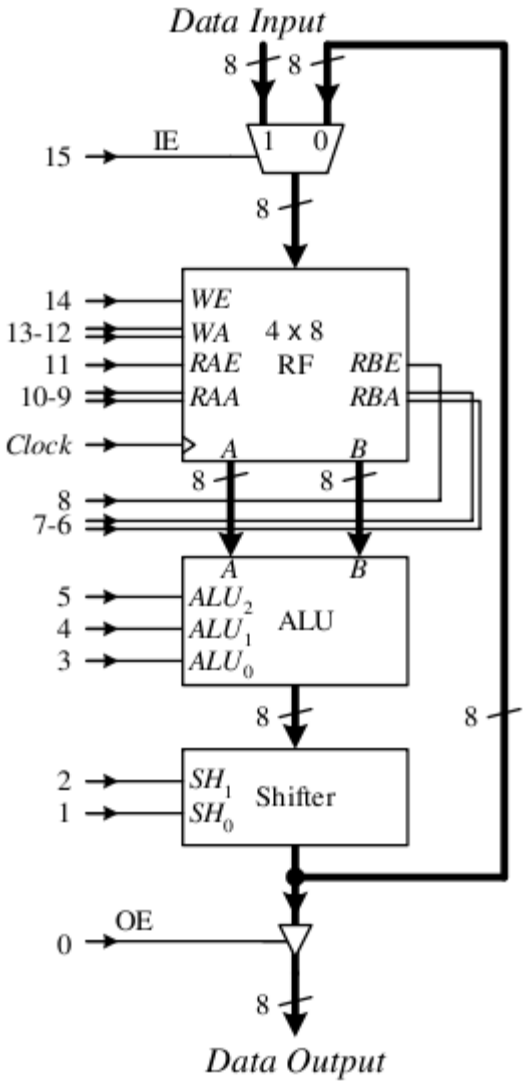
Control Word	Instruction	*productMux*	*productLoad*	*nLoad*	*nCount*	*Out*
1	INPUT *n*, *product* = 1	1	1	1	0	0
2	*product* = *product* * *n*, *n* = *n* – 1	0	1	0	1	0
3	OUTPUT *product*	×	0	0	0	1

General Purpose Datapath

```
1          sum = 0
2          INPUT n
3          WHILE (n ≠ 0){
4              sum = sum + n
5              n = n - 1
6          }
7          OUTPUT sum
```

ALU_2	ALU_1	ALU_0	Operation
0	0	0	Pass through A
0	0	1	A AND B
0	1	0	A OR B
0	1	1	NOT A
1	0	0	$A + B$
1	0	1	$A - B$
1	1	0	$A + 1$
1	1	1	$A - 1$

SH_1	SH_0	Operation
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right



Control Word	Instruction	*IE* 15	*WE* 14	$WA_{1,0}$ 13–12	*RAE* 11	$RAA_{1,0}$ 10–9	*RBE* 8	$RBA_{1,0}$ 7–6	$ALU_{2,1,0}$ 5–3	$SH_{1,0}$ 2–1	*OE* 0
1	$sum = 0$	0	1	00	1	00	1	00	101 (subtract)	00	0
2	INPUT n	1	1	01	0	××	0	××	×××	××	0
3	$sum = sum + n$	0	1	00	1	00	1	01	100 (add)	00	0
4	$n = n - 1$	0	1	01	1	01	0	××	111 (decrement)	00	0
5	OUTPUT *sum*	×	0	××	1	00	0	××	000 (pass)	00	1

Example: Write the control words for manipulating the above circuit to perform the following program

Multiplication of two unsigned numbers:

```
1          prod = 0
2          INPUT A
3          INPUT B
4          WHILE (B ≠ 0){
5             prod = prod + A
6             B = B - 1
7          }
8          OUTPUT prod
```

Control Word	Instruction	IE 15	WE 14	$WA_{1,0}$ 13–12	RAE 11	$RAA_{1,0}$ 10–9	RBE 8	$RBA_{1,0}$ 7–6	$ALU_{2,1,0}$ 5–3	$SH_{1,0}$ 2–1	OE 0
1	$prod = 0$	0	1	00	1	00	1	00	101 (subtract)	00	0
2	INPUT A	1	1	01	0	xx	0	xx	xxx	xx	0
3	INPUT B	1	1	10	0	xx	0	xx	xxx	xx	0
4	$prod = prod + A$	0	1	00	1	00	1	01	100 (add)	00	0
5	$B = B - 1$	0	1	10	1	10	0	xx	111 (decrement)	00	0
6	OUTPUT *prod*	x	0	xx	1	00	0	xx	000 (pass)	00	1

VHDL
for
Datapath

16x8 RAM

```
entity ram_16_8 is port(
        cs:             in std_logic; -- chip select
        wr:             in std_logic; -- write / read enable
        addr:           in std_logic_vector(3 downto 0);
        di:             in std_logic_vector(7 downto 0); -- data in
        do:             out std_logic_vector(7 downto 0)); -- data out
end ram_16_8;

architecture imp of ram_16_8 is

subtype cell is std_logic_vector(7 downto 0);
type ram_type is array(0 to 15) of cell;
signal RAM:     ram_type;

begin
        process(cs, wr)
        variable ctrl:          std_logic_vector(1 downto 0);
        begin
                                ctrl := cs & wr;

                                case ctrl is

                                        when "11" =>
                                                RAM(conv_integer(addr)) <= di;
                                                do <= di;
                                        when "10" =>
                                                do <= RAM(conv_integer(addr));
                                        when others =>
                                                do <= (others => 'Z');
                                end case;
        end process;
end imp;
```

256x8 ROM (Program Memory)

```
entity rom_256_8 is port(
        cs:             in std_logic;
        addr:           in std_logic_vector(7 downto 0);
        data:           out std_logic_vector(7 downto 0));
end rom_256_8;

architecture imp of rom_256_8 is

subtype cell is std_logic_vector(7 downto 0);
type rom_type is array(0 to 6) of cell;

constant ROM:   rom_type :=(
inp & A,
dec & A,
outp & A,
jnz & "1011",
dec & A,
jnz & "1110",
nop);

begin

        process(cs)
        begin

                if(cs = '1') then
                                data <= ROM(conv_integer(addr));
                else
                                data <= (others => 'Z');
                end if;

        end process;

end imp;
```

2 bit Multiplexer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity mux2 is port(
        s:      in std_logic;
        x0, x1: in std_logic;
        y:      out std_logic);
end mux2;

architecture imp of mux2 is
begin
        process(s, x0, x1)
        begin

                if(s = '0') then
                        y <= x0;
                else
                        y <= x1;
                end if;

        end process;

end imp;
```

4 bit Multiplexer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity mux4 is port(
        S:                      in std_logic_vector(1 downto 0);
        x0, x1, x2, x3:         in std_logic;
        y:                      out std_logic);
end mux4;

architecture imp of mux4 is
begin
        process(S, x0, x1, x2, x3)
        begin

        case S is

                when "00"       => y <= x0;
                when "01"       => y <= x1;
                when "10"       => y <= x2;
                when "11"       => y <= x3;
                when others     => y <= 'X';
        end case;

        end process;

end imp;
```

Register File

```
entity regfile is port(
        clk:            in std_logic;
        reset:          in std_logic;                   -- reset
        we:             in std_logic;                   -- write enable
        WA:             in std_logic_vector(1 downto 0); -- write address (4 registers)
        D:              in std_logic_vector(7 downto 0); -- input
        rbe:            in std_logic;
        RBA:            in std_logic_vector(1 downto 0); -- read address (4 registers)
        portA:          out std_logic_vector(7 downto 0);
        portB:          out std_logic_vector(7 downto 0));
end regfile;

architecture imp of regfile is

        subtype reg is std_logic_vector(7 downto 0);
        type regArray is array(0 to 3) of reg;
        signal RF: regArray;
begin
        WritePort: Process(clk ,reset)
        begin
                if(clk'event and clk='1') then
                        if(reset='1') then
                                        RF(0) <= (others => '0'); -- register A (accumulator)
                                        RF(1) <= (others => '0'); -- register B
                                        RF(2) <= (others => '0'); -- register F (Flag)
                                        RF(3) <= (others => '1'); -- register S (Stack Pointer)
                        elsif(we='1') then
                                        RF(conv_integer(WA)) <= D;
                        end if;
                end if;
        end process;

        ReadPortB: Process(rbe, RBA)
        begin
                if(rbe='1') then
                        PortB <= RF(conv_integer(RBA));
                else
                        PortB <= (others => 'X');
                end if;
        end process;

        ReadPortA: PortA <= RF(0); -- PortA always accumulator register
end imp;
```

Instruction Register

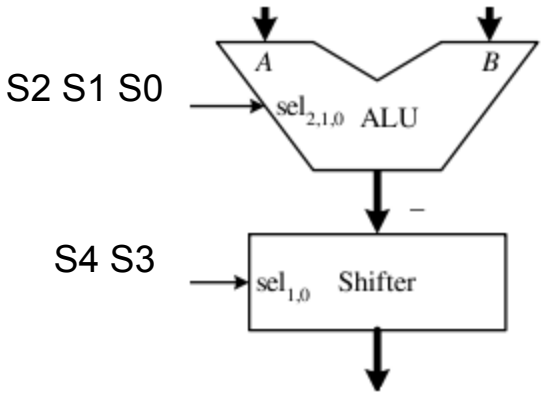
```
entity IR is port(
        clk:            in std_logic;
        reset:          in std_logic;                           -- reset
        load:           in std_logic;
        INPUT :         in std_logic_vector(7 downto 0);        -- input
        OUTPUT:         out std_logic_vector(7 downto 0));      -- output
end IR;

architecture imp of IR is

        subtype reg is std_logic_vector(7 downto 0);
        signal IR8: reg;

begin
        Process(clk ,reset)
        begin
                if(clk'event and clk='1') then
                        if(reset='1') then

                                        IR8 <= (others => '0');

                        elsif(load = '1') then

                                        IR8 <= INPUT;

                        end if;

                OUTPUT <= IR8;

                end if;
        end process;
end imp;
```

Program Counter

```
entity PC is port(
        clk:            in std_logic;
        reset:          in std_logic;                       -- reset
        load:           in std_logic;
        INPUT :         in std_logic_vector(7 downto 0);    -- input
        OUTPUT:         out std_logic_vector(7 downto 0)); -- output
end PC;

architecture imp of PC is

        subtype reg is std_logic_vector(7 downto 0);

        signal PC8: reg := "00000000";

begin

        Process(clk ,reset, load)
        begin
                if(clk'event and clk='1') then
                        if(reset='1') then

                                        PC8 <= (others => '0');

                        elsif(load = '1') then

                                        PC8 <= INPUT;

                        end if;

                end if;
        end process;

OUTPUT <= PC8;

end imp;
```

Components of ALU

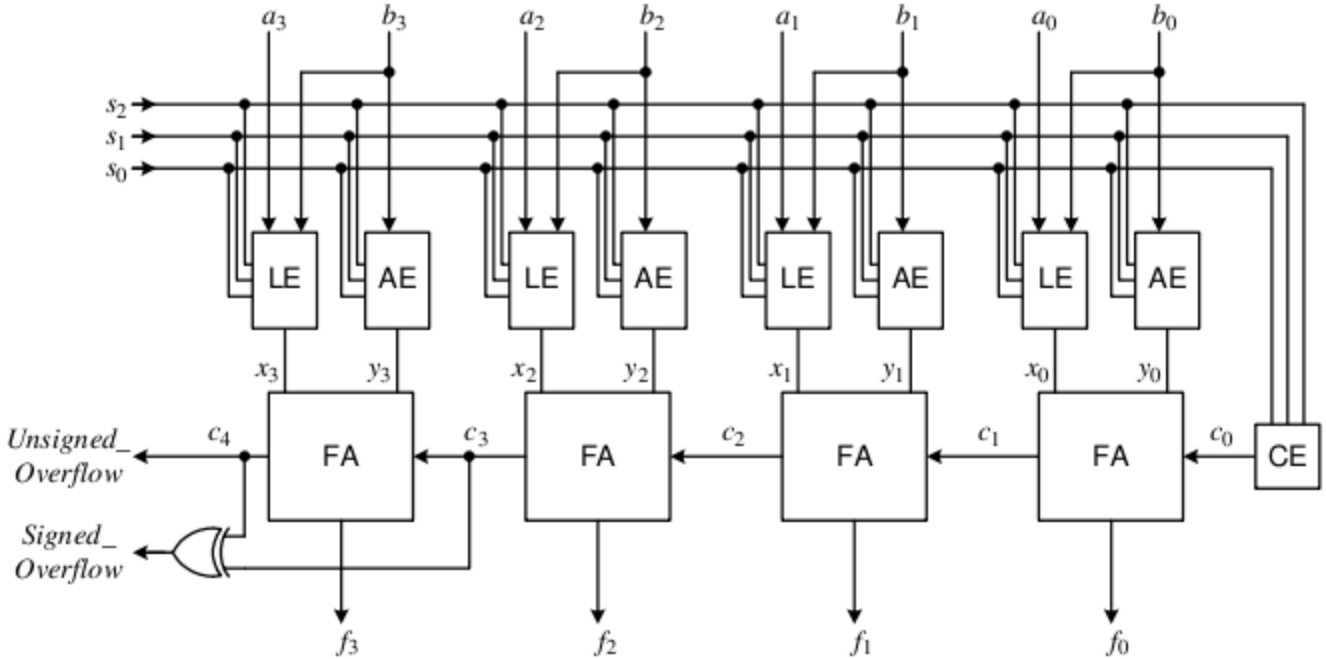
ALU and Shifter



s2	s1	s0	Function
0	0	0	Pass A to output
0	0	1	A and B
0	1	0	A or B
0	1	1	A'
1	0	0	A + B
1	0	1	A – B
1	1	0	A + 1
1	1	1	A – 1

s4	s3	Function
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right

Adder/Subtractor with Arithmetic and Logic Extenders



Full Adder

```
entity FA is port(
        carryIn:                in std_logic;
        carryOut:               out std_logic;
        x, y:                   in std_logic;
        s:                      out std_logic);
end FA;

architecture imp of FA is
begin
        s <= x xor y xor carryIn;
        carryOut <= (x and y) or (carryIn and (x xor y));
end imp;
```

Arithmetic and Logic Extenders

s_2	s_1	s_0	Operation Name	Operation	x_i (LE)	y_i (AE)	c_0 (CE)
0	0	0	Pass	Pass A to output	a_i	0	0
0	0	1	AND	A AND B	a_i AND b_i	0	0
0	1	0	OR	A OR B	a_i OR b_i	0	0
0	1	1	NOT	A'	a_i'	0	0
1	0	0	Addition	$A + B$	a_i	b_i	0
1	0	1	Subtraction	$A - B$	a_i	b_i'	1
1	1	0	Increment	$A + 1$	a_i	0	1
1	1	1	Decrement	$A - 1$	a_i	1	0

(a)

s_2	s_1	s_0	x_i
0	0	0	a_i
0	0	1	$a_i\, b_i$
0	1	0	$a_i + b_i$
0	1	1	a_i'
1	×	×	a_i

s_2	s_1	s_0	b_i	y_i
0	×	×	×	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

s_2	s_1	s_0	c_0
0	×	×	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Logic Extender

```
entity LE is port(
        S:                      in std_logic_vector(2 downto 0);
        a, b:                   in std_logic;
        x:                              out std_logic);
end LE;

architecture imp of LE is
begin
        process(S, a, b)
        begin
                case S is
                        when "000" => x <= a;
                        when "001" => x <= a and b;
                        when "010" => x <= a or b;
                        when "011" => x <= not a;
                        when others => x <= a;
                end case;
        end process;

end imp;
```

Arithmetic Extender

```
entity AE is port(
        S:                      in std_logic_vector(2 downto 0);
        a, b:                   in std_logic;
        y:                      out std_logic);
end AE;

architecture imp of AE is
begin
        process(S, a, b)
        begin
                case S is
                        when "100" => y <= b;
                        when "101" => y <= not b;
                        when "110" => y <= '0';
                        when "111" => y <= '1';
                        when others => y <= '0';
                end case;
        end process;

end imp;
```

8 bit LE

```
entity LE8 is port(
        S:                      in std_logic_vector(2 downto 0);
        A, B:                   in std_logic_vector(7 downto 0);
        X:                      out std_logic_vector(7 downto 0));
end LE8;

architecture imp of LE8 is

        component LE is port(
                S:                      in std_logic_vector(2 downto 0);
                a, b:                   in std_logic;
                x:                      out std_logic);
        end component;

begin

        U0: LE port map(S, A(0), B(0), X(0));
        U1: LE port map(S, A(1), B(1), X(1));
        U2: LE port map(S, A(2), B(2), X(2));
        U3: LE port map(S, A(3), B(3), X(3));
        U4: LE port map(S, A(4), B(4), X(4));
        U5: LE port map(S, A(5), B(5), X(5));
        U6: LE port map(S, A(6), B(6), X(6));
        U7: LE port map(S, A(7), B(7), X(7));

end imp;
```

8 bit AE

```
entity AE8 is port(
        S:                      in std_logic_vector(2 downto 0);
        A, B:                   in std_logic_vector(7 downto 0);
        Y:                      out std_logic_vector(7 downto 0));
end AE8;

architecture imp of AE8 is

        component AE is port(
                S:              in std_logic_vector(2 downto 0); --
                a, b:           in std_logic;
                y:              out std_logic);
        end component;

begin

        U0: AE port map(S, A(0), B(0), Y(0));
        U1: AE port map(S, A(1), B(1), Y(1));
        U2: AE port map(S, A(2), B(2), Y(2));
        U3: AE port map(S, A(3), B(3), Y(3));
        U4: AE port map(S, A(4), B(4), Y(4));
        U5: AE port map(S, A(5), B(5), Y(5));
        U6: AE port map(S, A(6), B(6), Y(6));
        U7: AE port map(S, A(7), B(7), Y(7));

end imp;
```

addsub8

```
entity addsub8 is port(
        A:                      in std_logic_vector(7 downto 0);
        B:                      in std_logic_vector(7 downto 0);
        F:                      out std_logic_vector(7 downto 0);
        carryIn:                in std_logic;
        unsigned_overflow:      out std_logic;
        signed_overflow:        out std_logic);
end addsub8;

architecture imp of addsub8 is

        component FA port(
                carryIn:                in std_logic;
                carryOut:               out std_logic;
                x, y:                   in std_logic;
                s:                      out std_logic);
        end component;

        signal C:               std_logic_vector(7 downto 1);

begin
        U0: FA port map(carryIn, C(1), A(0), B(0), F(0));
        U1: FA port map(C(1), C(2), A(1), B(1), F(1));
        U2: FA port map(C(2), C(3), A(2), B(2), F(2));
        U3: FA port map(C(3), C(4), A(3), B(3), F(3));
        U4: FA port map(C(4), C(5), A(4), B(4), F(4));
        U5: FA port map(C(5), C(6), A(5), B(5), F(5));
        U6: FA port map(C(6), C(7), A(6), B(6), F(6));
        U7: FA port map(C(7), unsigned_overflow, A(7), B(7), F(7));

        signed_overflow <= C(7) xor C(6);
end imp;
```

Shifter

```
entity shifter8 is port(
        S:              in std_logic_vector(1 downto 0);
        A:              in std_logic_vector(7 downto 0);
        Y:              out std_logic_vector(7 downto 0);
        carryOut:       out std_logic;
        zero:           out std_logic);
end shifter8;

architecture imp of shifter8 is

component mux4 is port(
        S:              in std_logic_vector(1 downto 0);
        x0, x1, x2, x3: in std_logic;
        y:              out std_logic);
end component;

begin

        process(S)  -- carry flag
        begin
                if(S="01") then
                        carryOut <= A(7);
                elsif(S="10") then
                        carryOut <= A(0);
                end if;
        end process;

        U0:     mux4 port map(S, A(0), '0', A(1), A(1), Y(0));
        U1:     mux4 port map(S, A(1), A(0), A(2), A(2), Y(1));
        U2:     mux4 port map(S, A(2), A(1), A(3), A(3), Y(2));
        U3:     mux4 port map(S, A(3), A(2), A(4), A(4), Y(3));
        U4:     mux4 port map(S, A(4), A(3), A(5), A(5), Y(4));
        U5:     mux4 port map(S, A(5), A(4), A(6), A(6), Y(5));
        U6:     mux4 port map(S, A(6), A(5), A(7), A(7), Y(6));
        U7:     mux4 port map(S, A(7), A(6), '0', A(0), Y(7));

        process(S) -- zero flag is set if the output of ALU is zero !! control the logic!
        begin
                if(S="00") then
                        if(A=0) then
                                zero <= '1';
                        else
                                zero <= '0';
                        end if;
                end if;
        end process;

end imp;
```

ALU

```
entity ALU is port(
        S:                      in std_logic_vector(4 downto 0);
        A, B:                   in std_logic_vector(7 downto 0);
        F:                      out std_logic_vector(7 downto 0);
        unsigned_overflow:      out std_logic;
        signed_overflow:        out std_logic;
        carry:                  out std_logic;
        zero:                   out std_logic);
end ALU;

architecture imp of ALU is
        component LE8 is port(
                S:                      in std_logic_vector(2 downto 0); -- Selection bits
                A, B:                   in std_logic_vector(7 downto 0);
                X:                      out std_logic_vector(7 downto 0));
        end component;

        component AE8 is port(
                S:                      in std_logic_vector(2 downto 0); -- Selection bits
                A, B:                   in std_logic_vector(7 downto 0);
                Y:                      out std_logic_vector(7 downto 0));
        end component;

        component addsub8 is port(
                A:                      in std_logic_vector(7 downto 0);
                B:                      in std_logic_vector(7 downto 0);
                F:                      out std_logic_vector(7 downto 0);
                carryIn:                in std_logic;
                unsigned_overflow:      out std_logic;
                signed_overflow:        out std_logic);
        end component;

        component shifter8 is port(
                S:                      in std_logic_vector(1 downto 0);
                A:                      in std_logic_vector(7 downto 0);
                Y:                      out std_logic_vector(7 downto 0);
                carryOut:               out std_logic;
                zero:                   out std_logic);
        end component;

        signal X, Y, ShiftInput:        std_logic_vector(7 downto 0);
        signal c0:                      std_logic;
begin
        CarryExtender_ALU: c0 <= (S(0) xor S(1)) and S(2);
        LogicExtender8_ALU:     LE8 port map(S(2 downto 0), A, B, X);
        ArithmeticExtender8_ALU:AE8 port map(S(2 downto 0), A, B, Y);
        AddSub8_ALU:            addsub8 port map(X, Y, ShiftInput, c0, unsigned_overflow, signed_overflow);
        Shifter8_ALU:           shifter8 port map(S(4 downto 3), ShiftInput, F, carry, zero);
end imp;
```