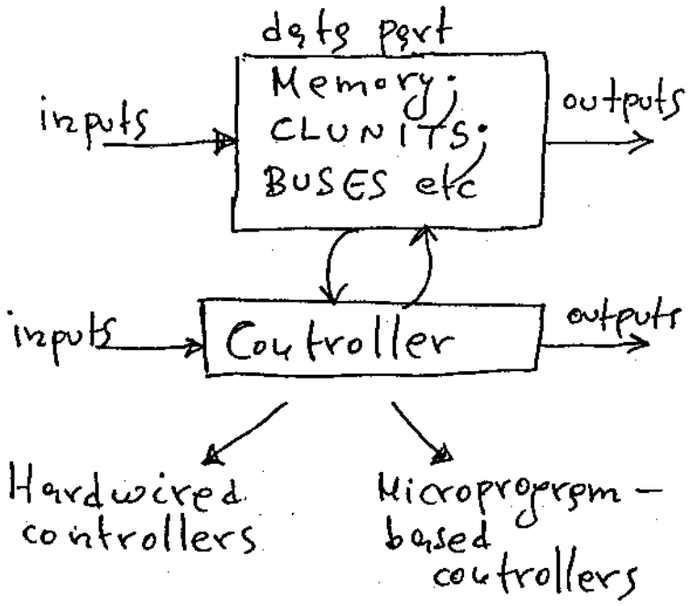
EE 3755

• Digital Systems



The system has a behaviour described by a sequence of control steps; (states).

• Description tool for digital systems:

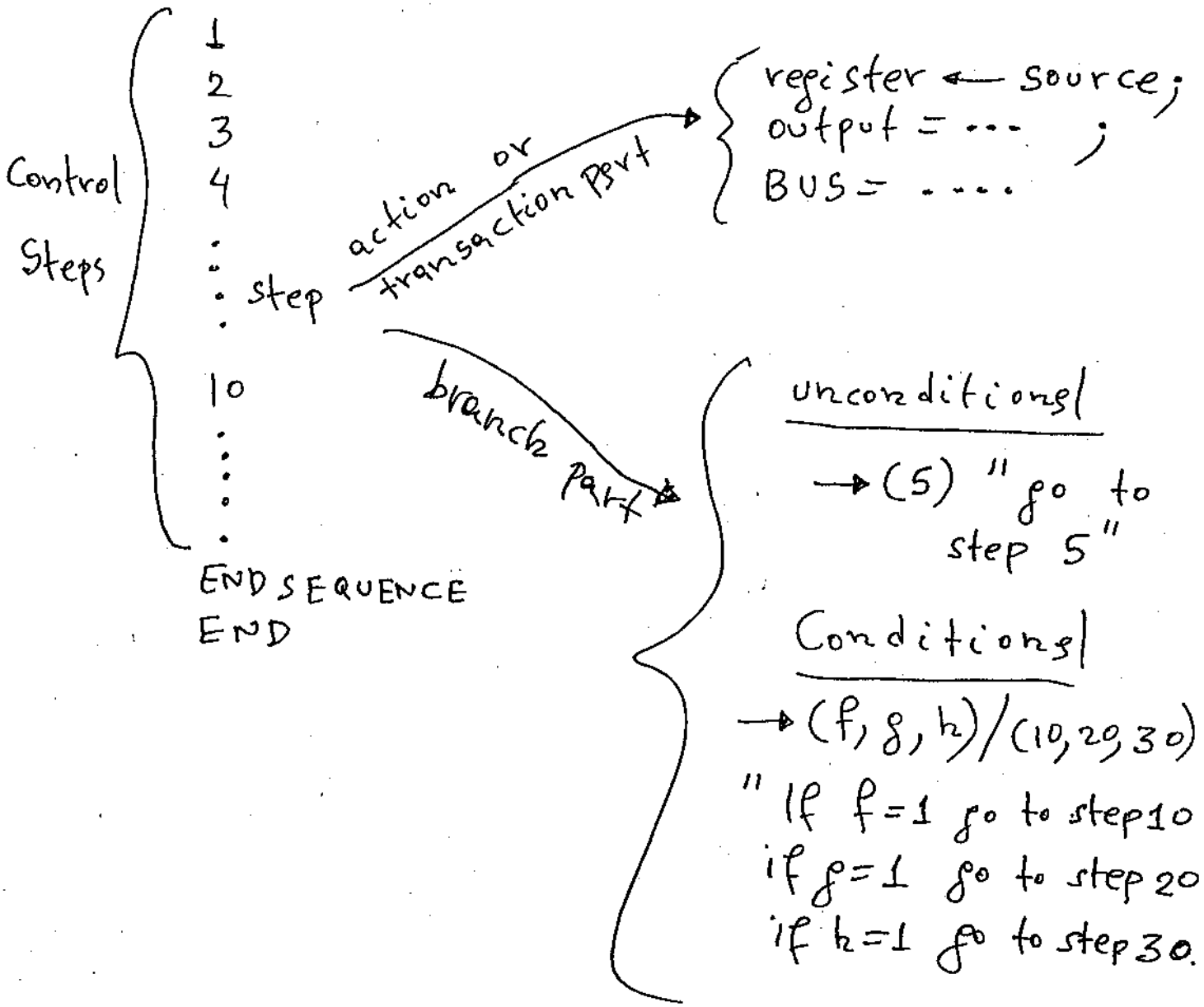
AHPL; (A Hardware Programming Language or a Hardware Description Language).

• AHPL description of Digital Systems

```
MODULE: NAME
INPUTS: ....
OUTPUTS: ....
MEMORY: ...
CLUNITS: ....
BUSES: ...
```

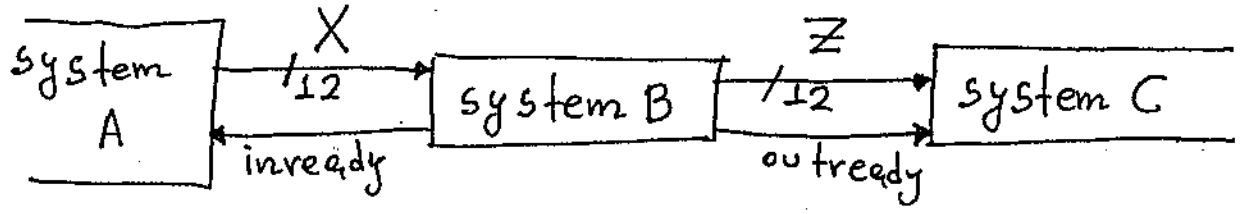
Declarations; (everything declared by name and dimension).

→ go to next page

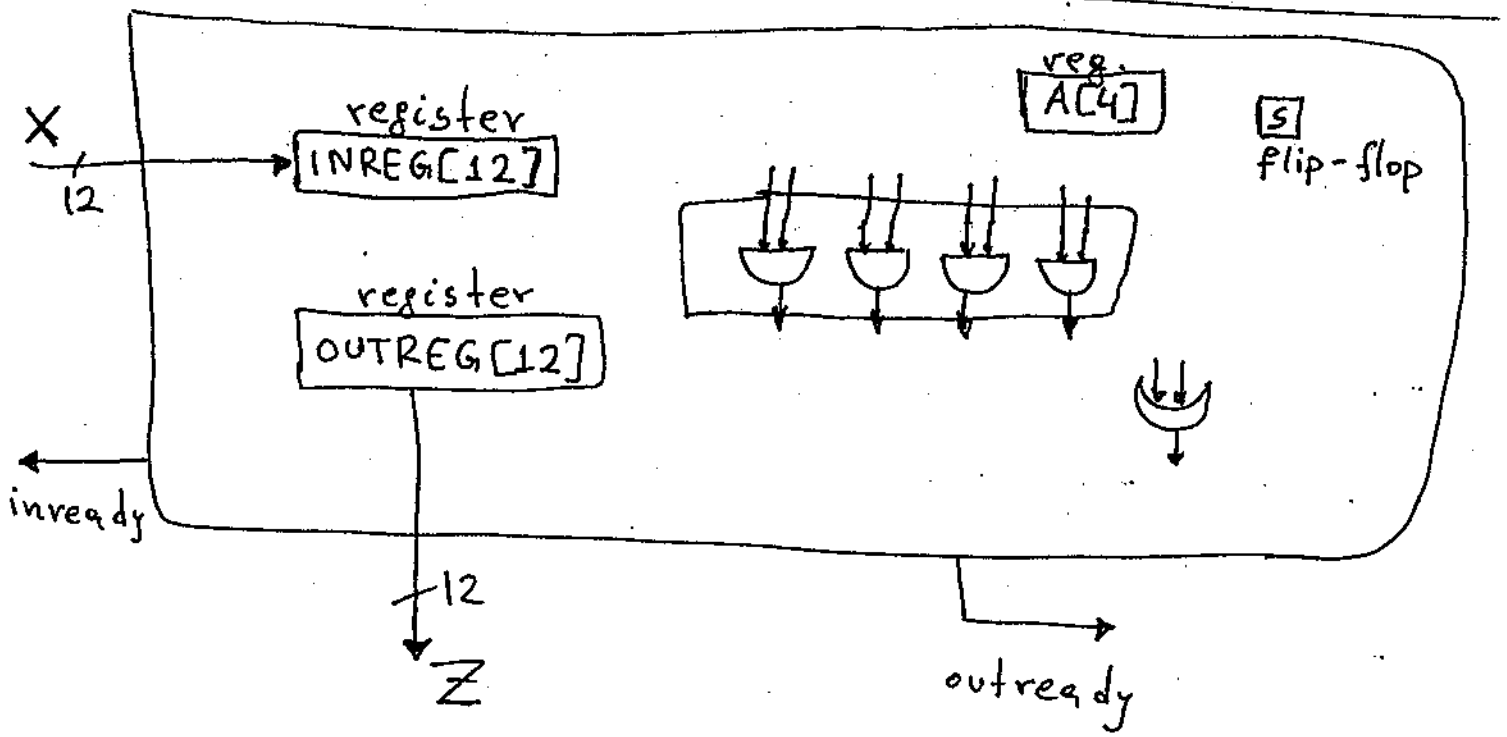

Control
Steps
1
2
3
4
10
step
action or
transaction part
register ← source;
output = ... ;
BUS = ...
branch part
unconditional
→ (5) " go to
step 5 "
Conditional
→ (f, g, h)/(10, 20, 30)
" if f=1 go to step 10
if g=1 go to step 20
if h=1 go to step 30.
END SEQUENCE
END

EE 3755

(1) d



- inready = 1 for one clock period
- then valid data appear on X input and get stored in system B.
- Then data are tested and if $X[0:3] \wedge X[4:7] \wedge X[8:11] = 0,0,0,0$ the controller goes back to its initial state etc... Else the system B outputs the data through Z output after outready = 1 for one clock period; then the controller goes back to its initial step etc....

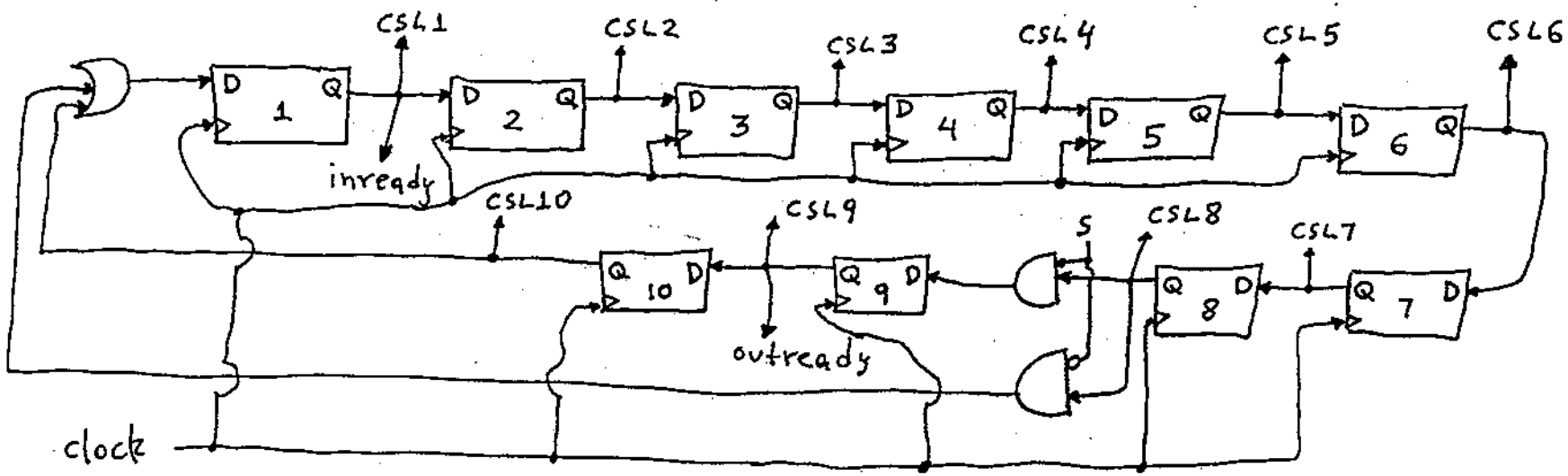


EE 3755 (2)d

```
MODULE: DATASELECTOR
MEMORY: INREG[12]; OUTREG[12]; A[4]; S.
INPUTS: X[12].
OUTPUTS: Z[12]; inready; outready.
1  inready = 1.
2  INREG ← X.
3  A ← INREG[0:3] ∧ INREG[4:7].
4  A ← INREG[8:11] ∧ A.
5  S ← A[0] ∨ A[1].
6  S ← S ∨ A[2].
7  S ← S ∨ A[3].
8  → (S, S̄)/(9, 1).
9  OUTREG ← INREG;
   outready = 1.
10 Z = OUTREG;
   → (1).
   END SEQUENCE.
   END.
```

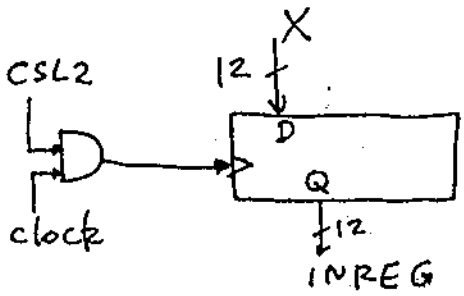
(3)

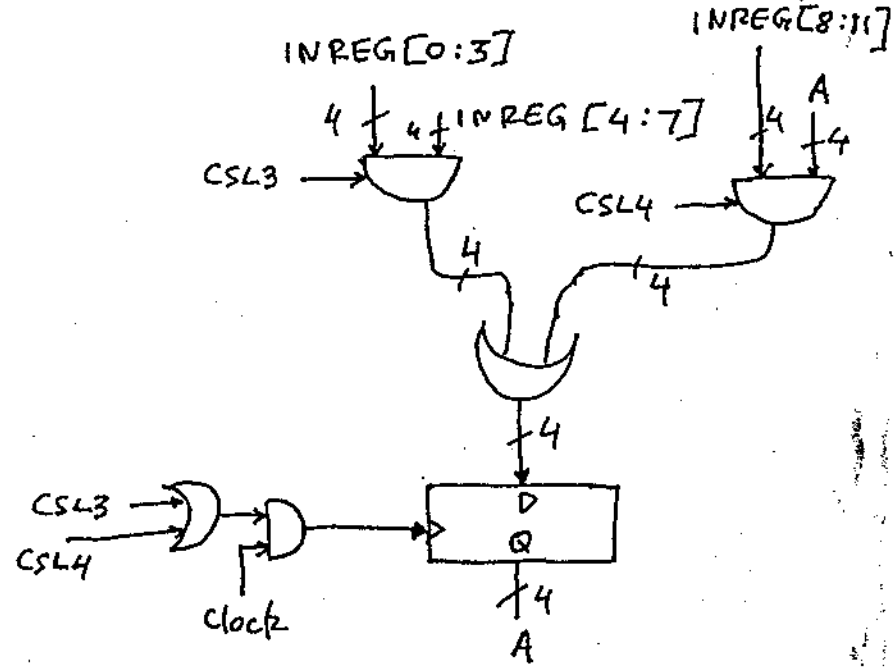
Hardwired Controller

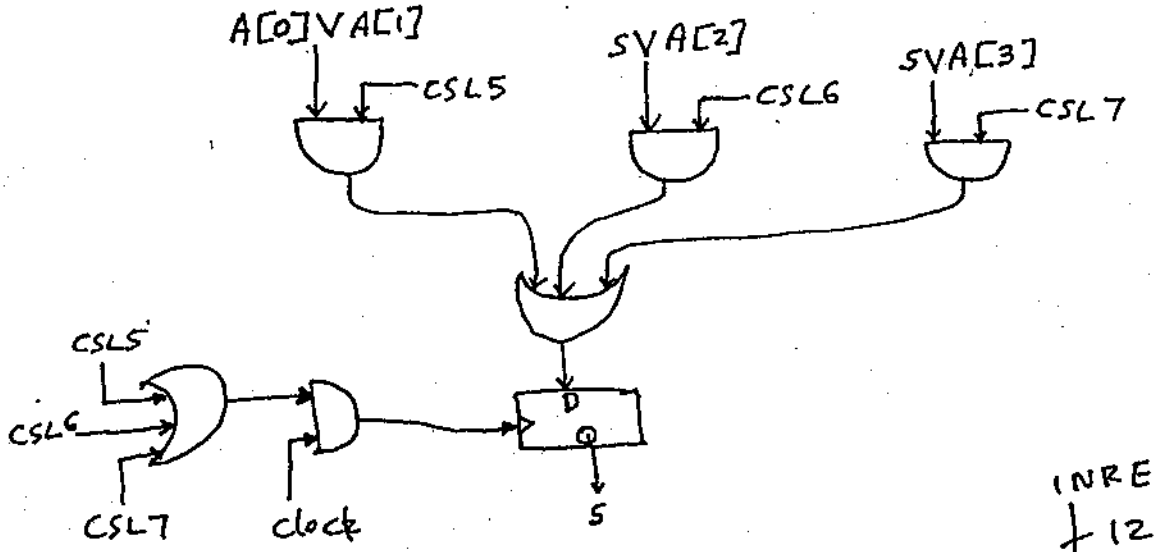


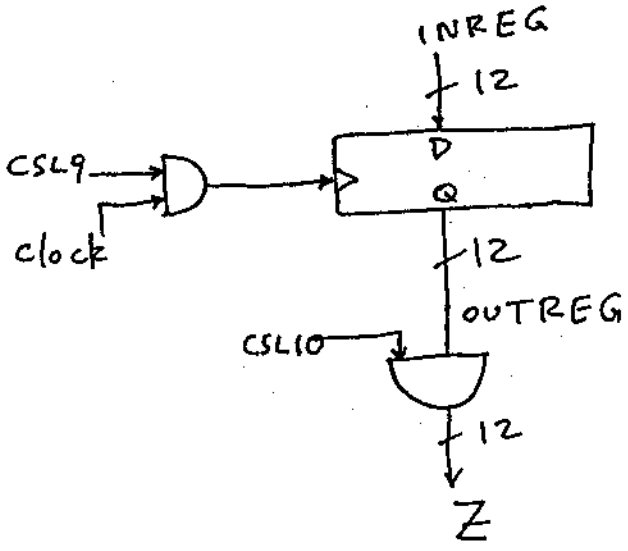
④d

Data Part


X
12
CSL2
D
Q
clock
12
INREG


INREG[0:3]
INREG[4:7]
INREG[8:11]
A
4
CSL3
CSL4
CSL3
CSL4
clock
D
Q
A

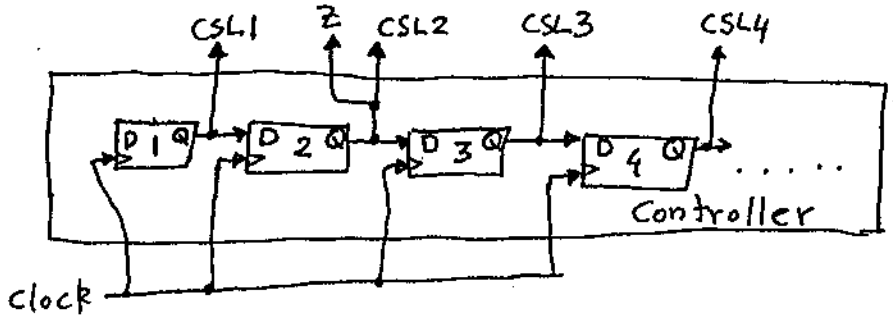

A[0]
A[1]
A[2]
A[3]
CSL5
CSL6
CSL7
CSL5
CSL6
CSL7
clock
D
Q
S

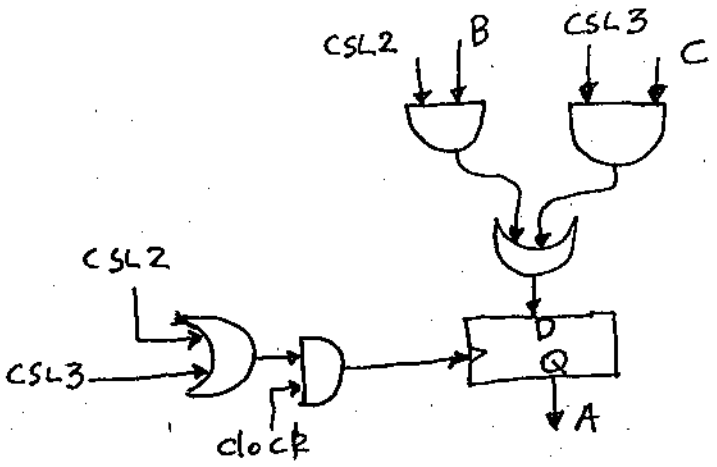

INREG
12
CSL9
clock
D
Q
12
OUTREG
CSL10
12
Z

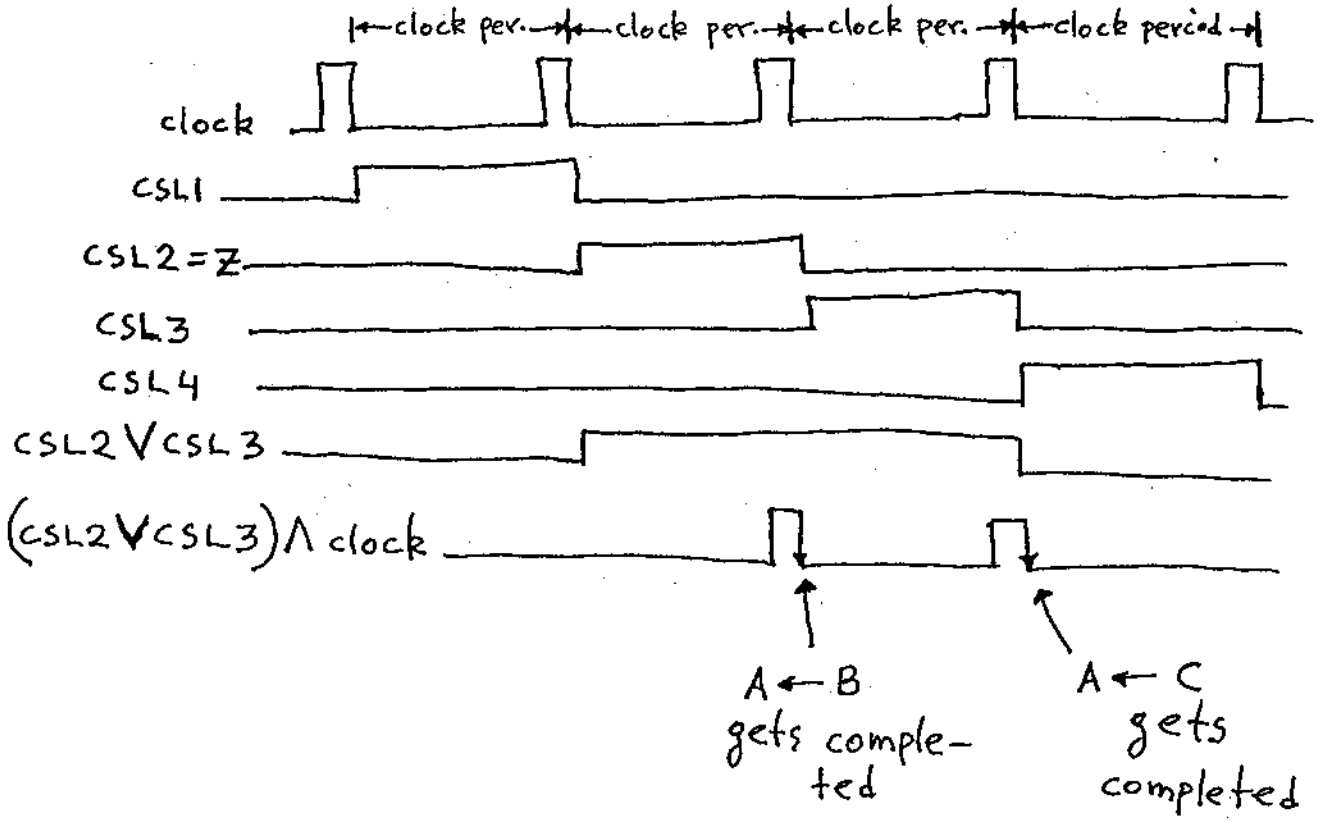
EE 3755

(1) d*

```
1 ....
2 A ← B; Z=1.
3 A ← C.
4 ....
:
```



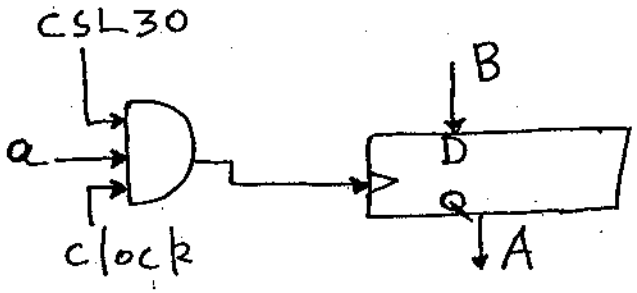




Conditional Transfers

30 $A * a \leftarrow B$

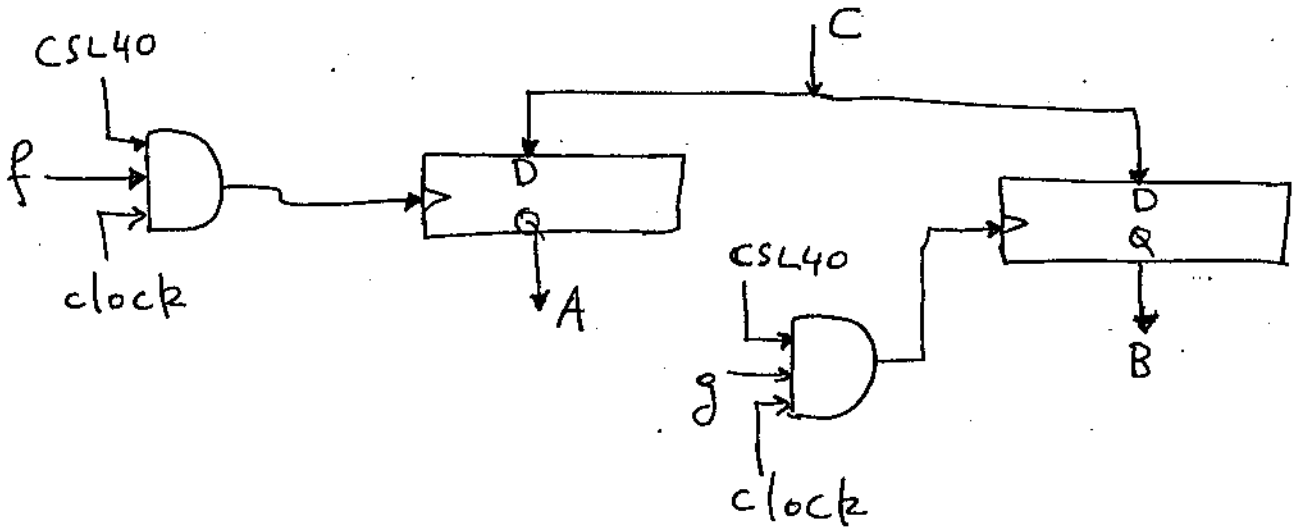
The meaning of the above is
$A \leftarrow B$ if $a = 1$;
Do not clock A if $a = 0$.



---

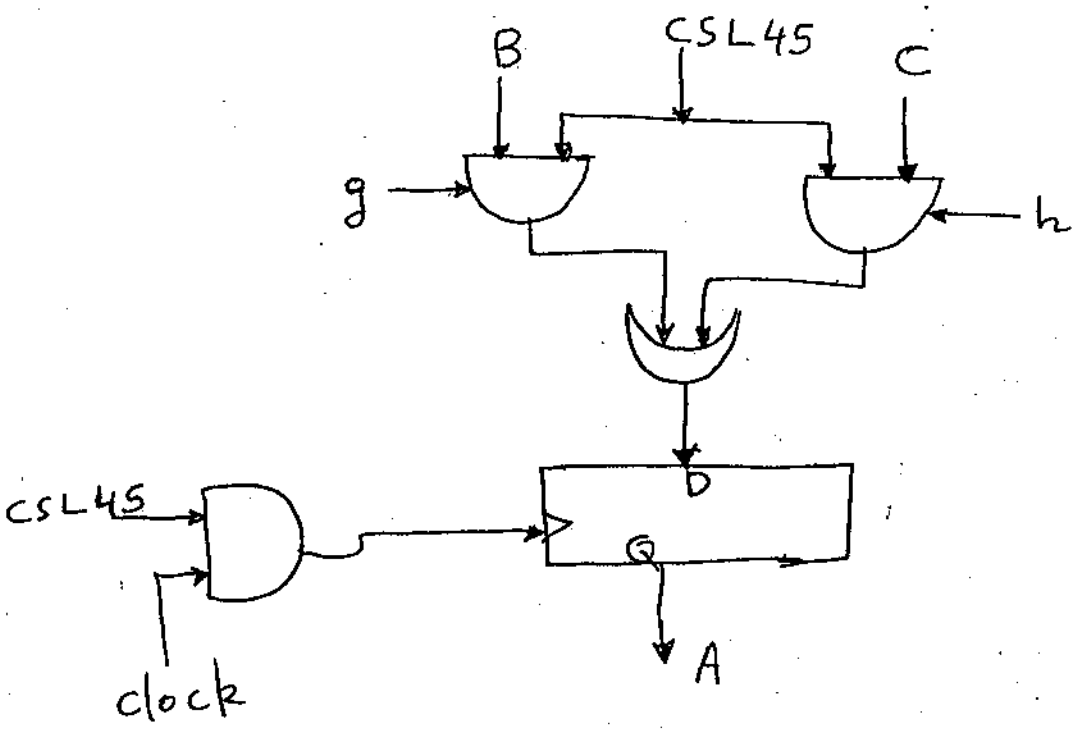
40 $(A \,!\, B) * (f, g) \leftarrow C$

The above is the same as
40. $A * f \leftarrow C$; $B * g \leftarrow C$



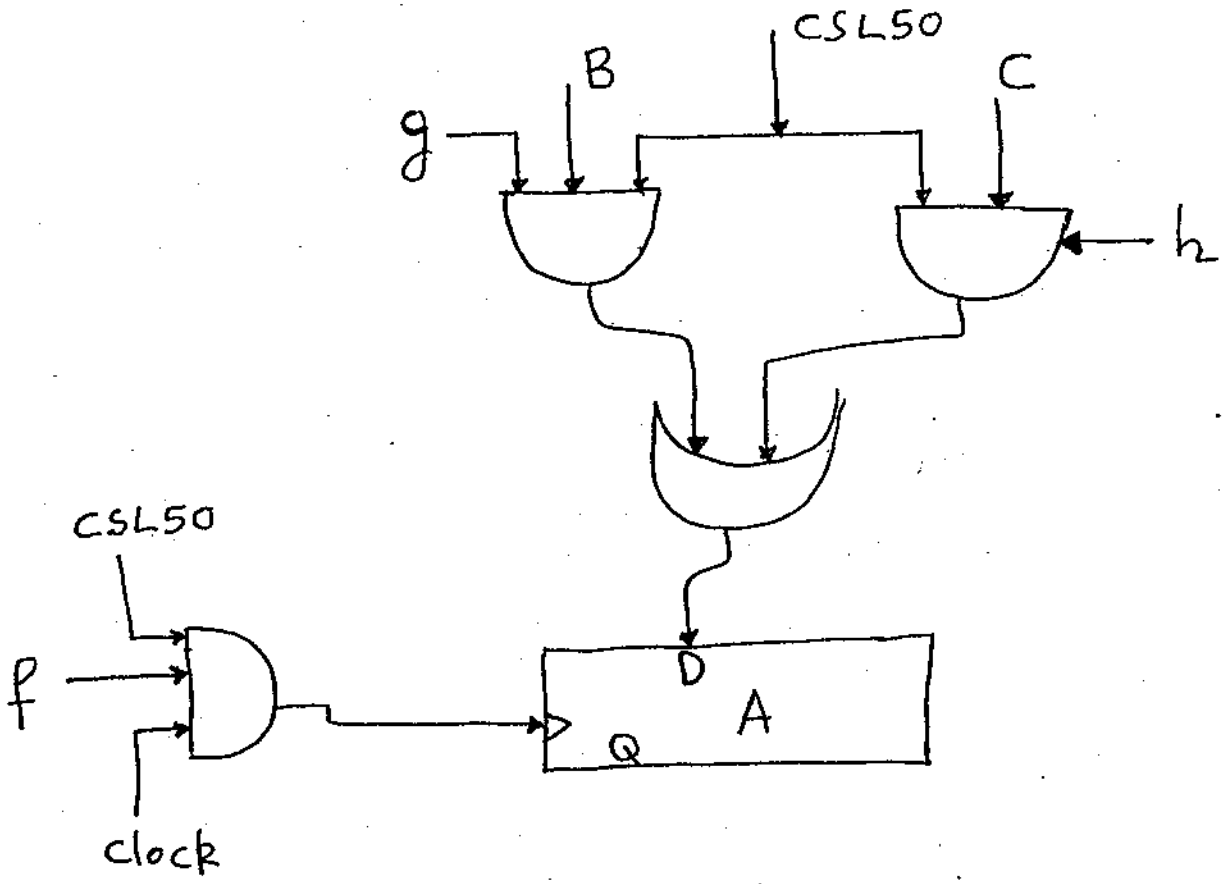
(2) e

45 A ← (B ! C) * (g, h).



(3)e

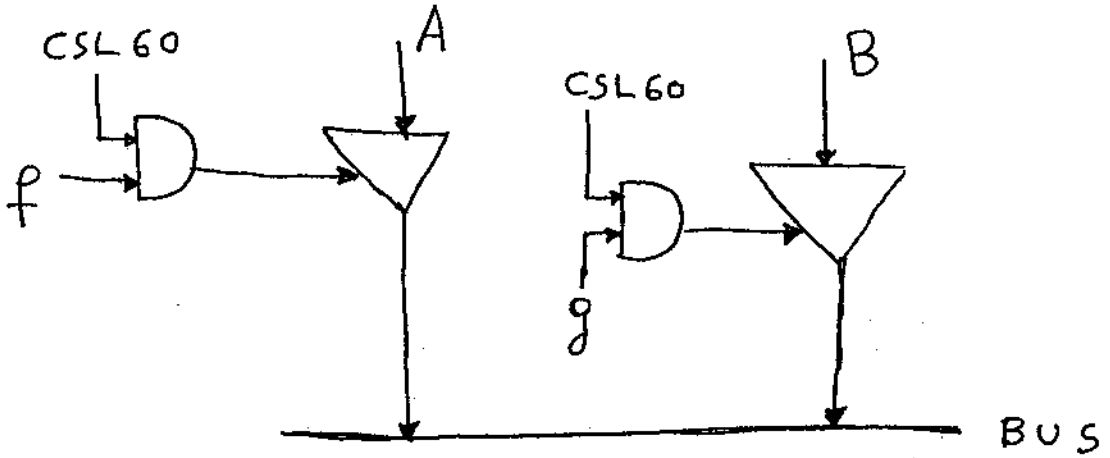
50 A * f ← (B ! C) * (g, h)



Note: Conditions on the left control the clock inputs to the registers. Conditions on the right specify busing networks for the data vectors.

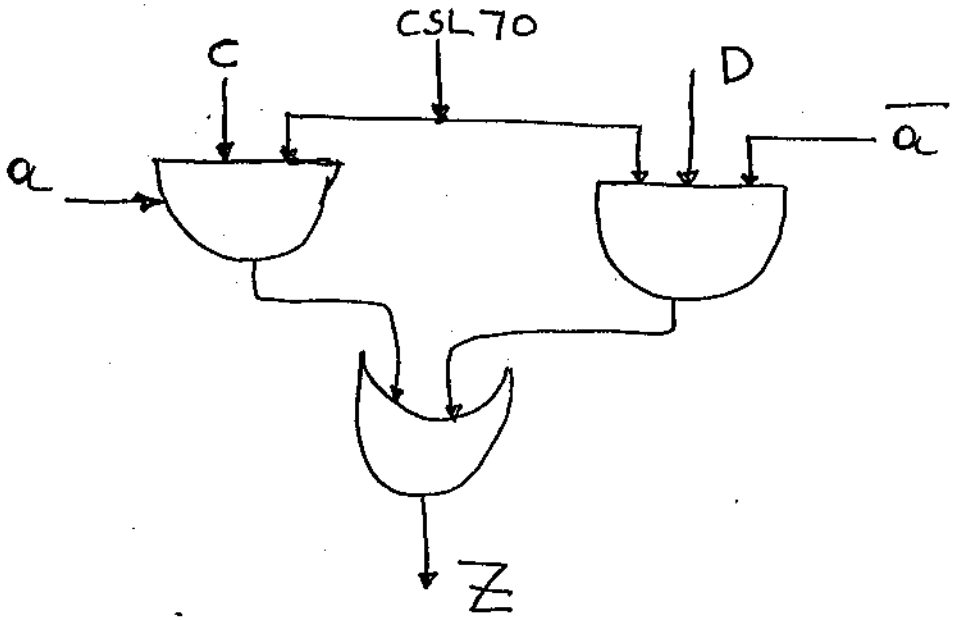
(4)e

60 BUS = (A ! B) * (f, g)



70 Z = (C ! D) * (a, $\overline{a}$)

Z is an output



Problem: Consider the system described below

```
MODULE:
MEMORY: A[8]; B[8]; D[8].
INPUTS: X[8]; a; b; c.
OUTPUTS: z.
1 → (ā)/(1).
2 → (b̄)/(4).
3 B ← X ∨ B;
  → (c̄)/(5).
4 D ← A ∧ B.
5 A ← X ⊕ B.
6 z = 1;
  → (1).
END SEQUENCE.
END.
```

Operation of the system starts after input a goes to 1 for one clock period. Then the inputs b and c get tested and the appropriate clocked transfers take place. After the

transfers get completed, an output $z=1$ is created for one clock period. Then control goes back to step 1 etc...

Reduce the sequence to an equivalent three-step sequence by employing conditional transfers. Is it possible to describe the system in fewer then three steps? Yes? No? Why?

Solution: The six-step description can be reduced into a three-step description. Step 1 of the reduced description will be step 1 of the original description, step 2 of the reduced description will be the result of combining steps 2, 3, 4, 5 of the original description while step 3 of the reduced description will be step 6 of the original description. We cannot reduce to less than 3 steps because we need one step to observe input a, one step to complete clocked transfers and one step to create $z=1$ for one clock period.

As already explained, the steps 3,4,5 of ③ f the original description are going to be combined together with step 2. These steps 3,4,5 do not deserve separate existence (separate clock periods for execution) and will be eliminated. The steps 3, 4, 5 are then called NO DELAY steps as shown below

MODULE: .....
MEMORY: .....
INPUTS: ....
OUTPUTS: .....
} same as before

1 → $(\overline{a})/(1)$.

2 → $(\overline{b})/(4)$.

3 NO DELAY
B ← X∨B;
→ $(\overline{c})/(5)$.

4 NO DELAY
D ← A∧B.

5 NO DELAY
A ← X⊕B.

6 z = 1;
→ (1).

END SEQUENCE.
END.

We now reduce the original six-step description into a three-step description. The following table can help in the reduction.

b c	transfers as done in the original description.	transfers done concurrently in one step (the same step).
0 0	$D \leftarrow A \wedge B$ followed by $A \leftarrow X \oplus B$	$D \leftarrow A \wedge B$; $A \leftarrow X \oplus B$
0 1	$D \leftarrow A \wedge B$ followed by $A \leftarrow X \oplus B$.	$D \leftarrow A \wedge B$; $A \leftarrow X \oplus B$
1 0	$B \leftarrow X \vee B$ followed by $A \leftarrow X \oplus B$	$B \leftarrow X \vee B$; $A \leftarrow X \oplus (X \vee B)$.
1 1	$B \leftarrow X \vee B$ followed by $D \leftarrow A \wedge B$ followed by $A \leftarrow X \oplus B$	$B \leftarrow X \vee B$; $D \leftarrow A \wedge (X \vee B)$; $A \leftarrow X \oplus (X \vee B)$.

The three-step description follows:

MODULE: ....
MEMORY: ....
INPUTS: ....
OUTPUTS: ...
} Same as before

1 → $(\overline{a})/(1)$.

2 $B * b \leftarrow X \vee B$;

$D * (\overline{b \wedge \overline{c}}) \leftarrow (A \wedge B \,!\, A \wedge (X \vee B)) * (\overline{b}, b)$;

$A \leftarrow (X \oplus B \,!\, X \oplus (X \vee B)) * (\overline{b}, b)$.

3 $z = 1$; → (1).

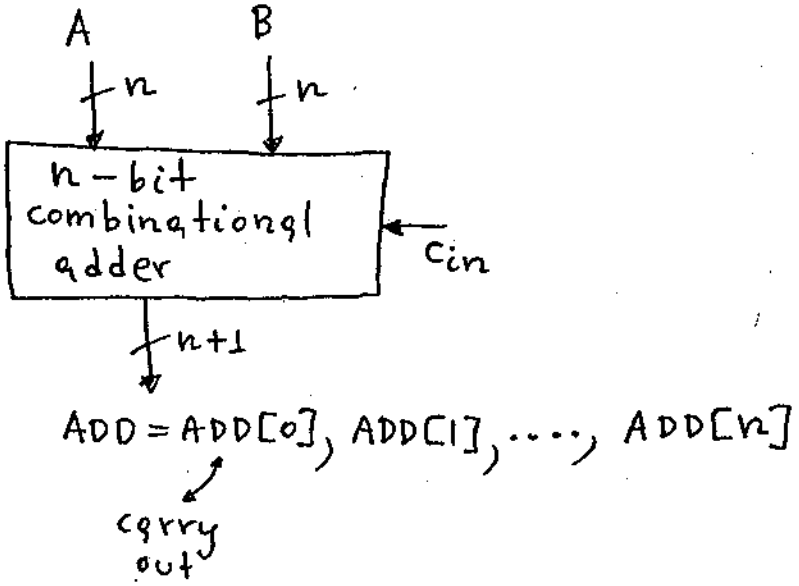
END SEQUENCE.

END.

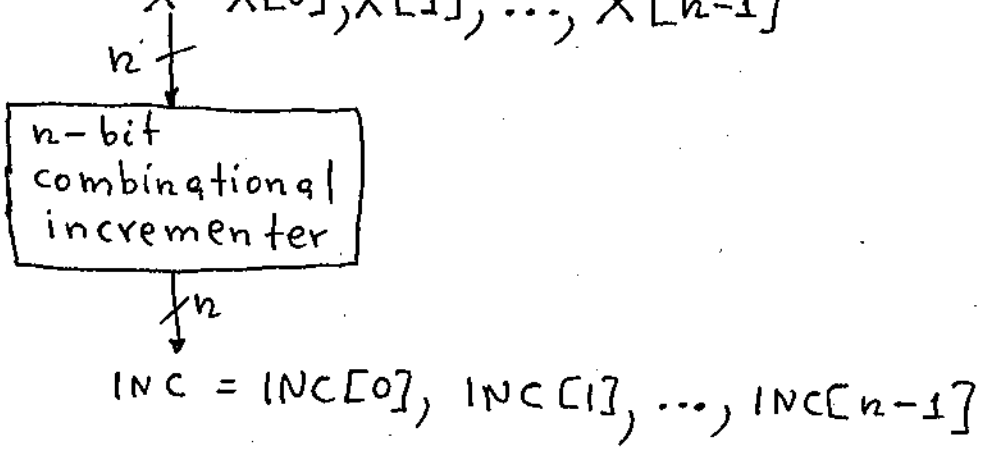
EE 3755

Some useful combinational logic units (CLUNITS).

- Adder



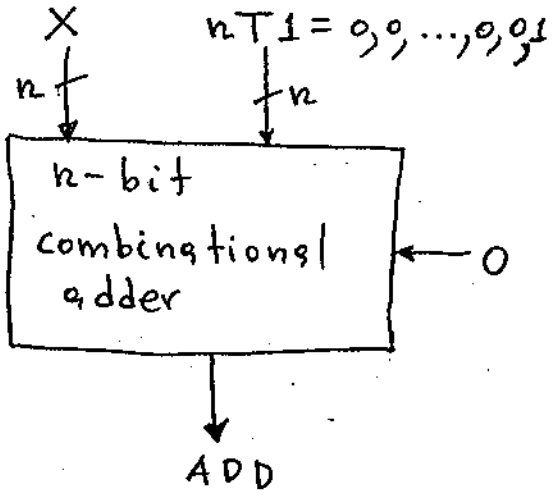
- Incrementer (by one)



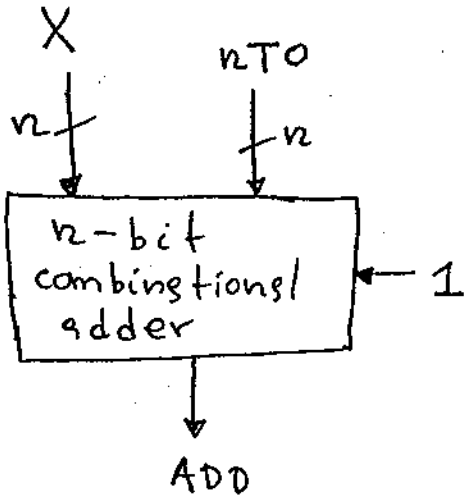
(2) g

X	INC
00...00	00...01
00...01	00...10
00...10	00...11
⋮	⋮
11...11	00...00

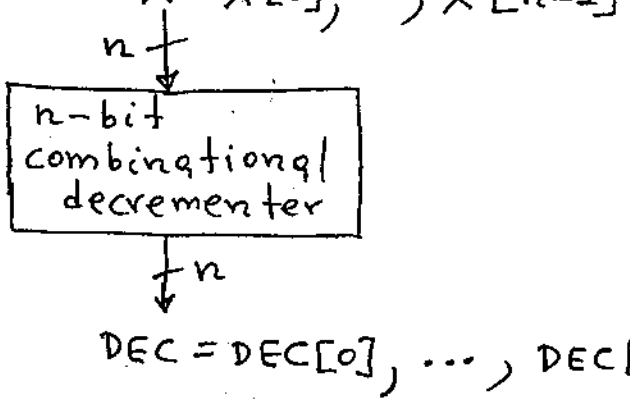
Of course, an adder can also be used to perform incrementation as shown below:



Or

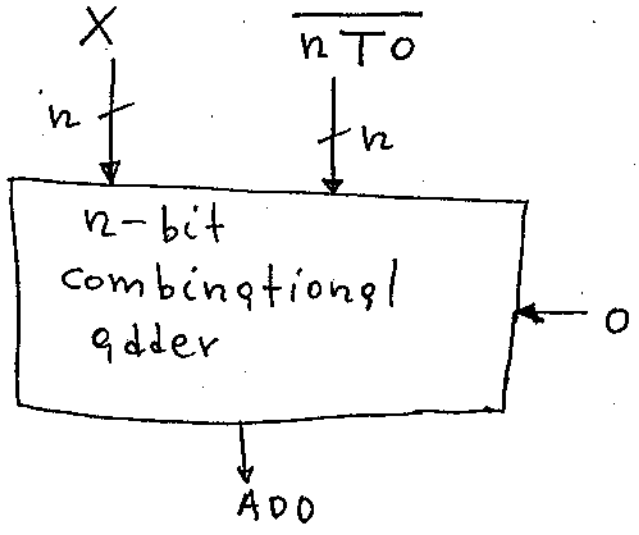


- **Decrementer (by one)**



X	DEC
00...00	11...11
00...01	00...00
00...10	00...01
⋮	⋮
11...11	111..10

Of course, an adder can also be used to perform decrementation as shown below:



Problem: You are asked to design a multiplier for multiplying two 16-bit unsigned numbers to produce their 32-bit product. Inputs to your multiplier are two 16-bit input vectors X and Y (X is used for the multiplier; Y for the multiplicand). The output of your multiplier is a 32-bit vector Z (the product).

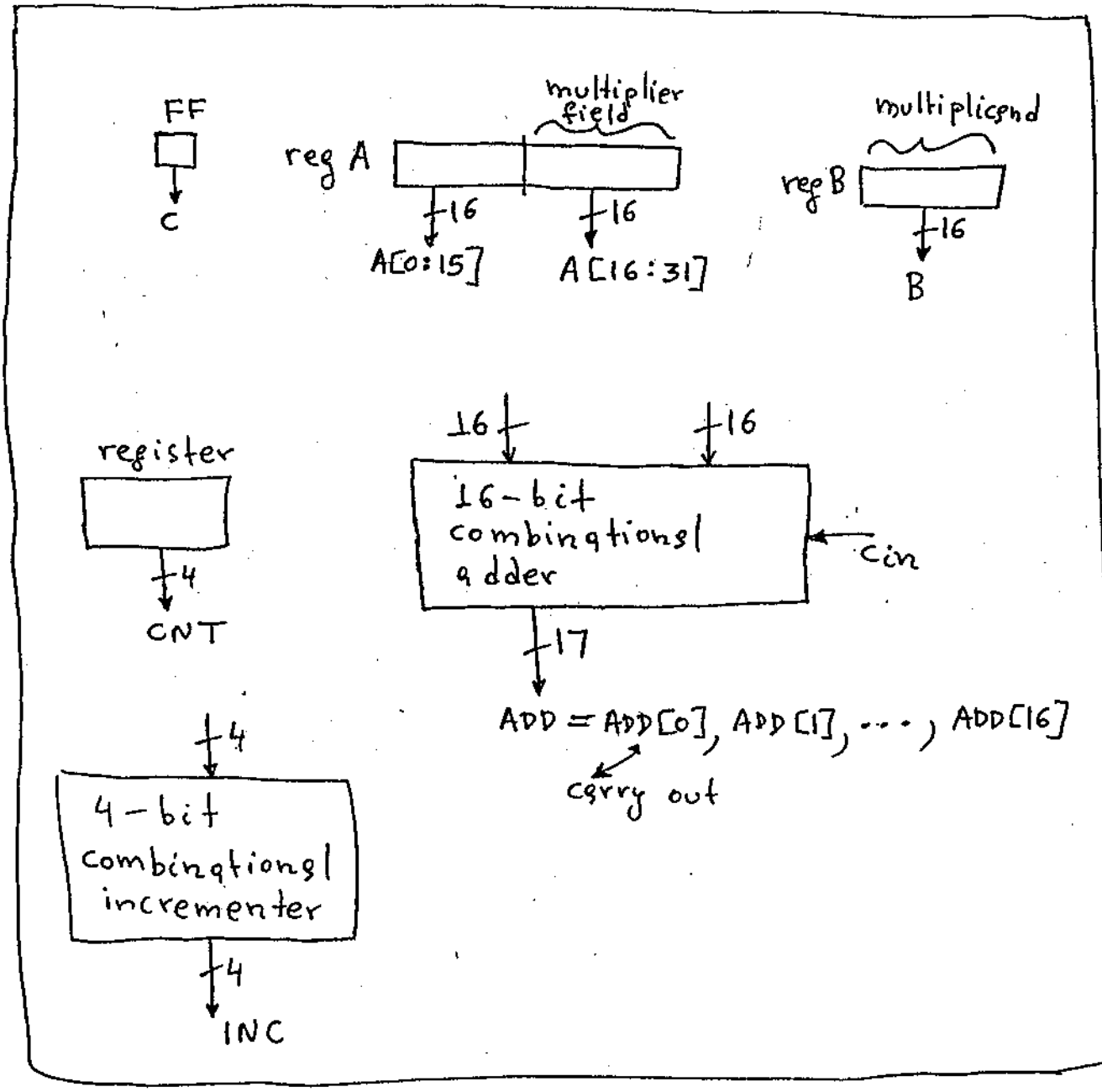
You are asked to:

- Provide a complete AHPL description of your design including declarations.
- Show in logic block diagram form the design of the control part and dats part.

Solution: The multiplier will consist of the following components.

- A 32-bit register named A. The right half part of it (A[16:31]) will be initialized with the multiplier. The part A[0:15] will be initialized with zeros.
- A 16-bit register named B to store the multiplicand.
- A flip-flop c, for storing the carry-out of the addition.
- A 16-bit combinational adder. The output of this adder is the 17-bit vector named ADD with ADD[0] being the carry out.

- A 4-bit counting register named CNT to be used in counting the 16 add/shift cycles.
- A 4-bit combinational incrementer used in incrementing by one the contents of the counter CNT. The output of the incrementer is a 4-bit vector named INC.



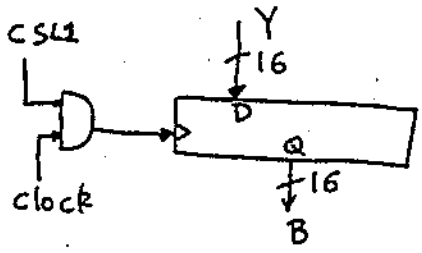
⑥ g

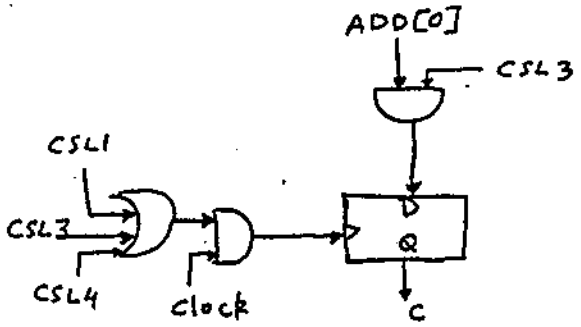
The following shows an AHPL description of the unsigned multiplier:

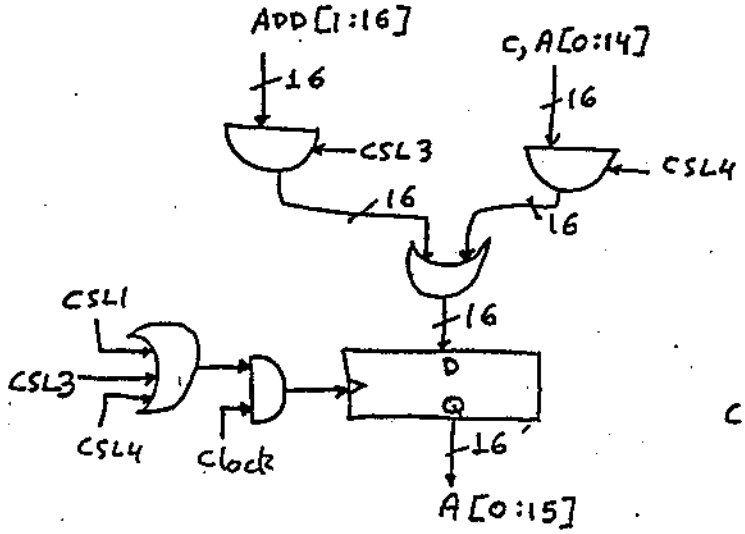
```
MODULE: UNSIGNEDMULT.
MEMORY: A[32]; B[16]; CNT[4]; C.
INPUTS: X[16]; Y[16].
OUTPUTS: Z[32].
CLUNITS: ADD[17] <: ADDER{16};
         INC[4] <: INCREMENTER{4}.
1 C, A[0:15] ← 17TO; A[16:31] ← X; B ← Y;
  CNT ← 4TO.
2 → (~A[31])/(4).
3 C, A[0:15] ← ADD(A[0:15]; B; 0).
4 C, A ← 0, C, A[0:30]; CNT ← INC(CNT);
  → (~(CNT[0] ∧ CNT[1] ∧ CNT[2] ∧ CNT[3]))/(2).
5 Z = A; → (1).
END SEQUENCE.
END.
```

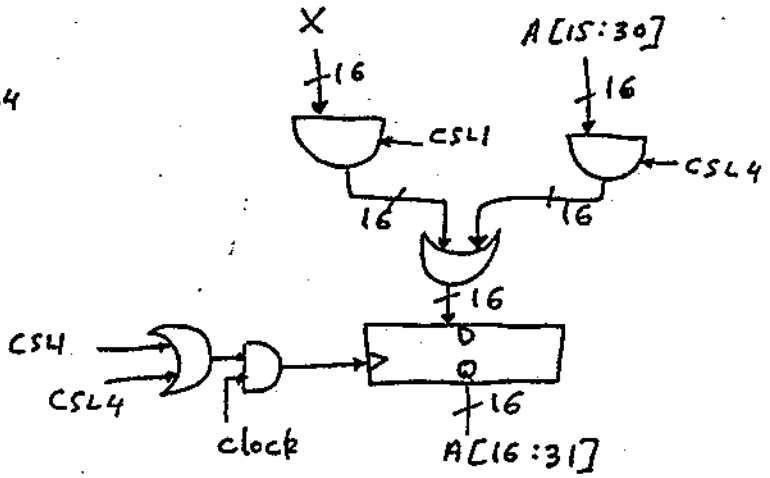
(8) g

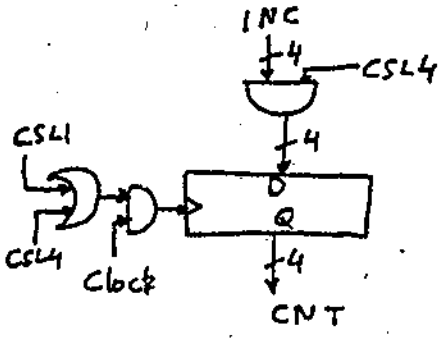
Data Part

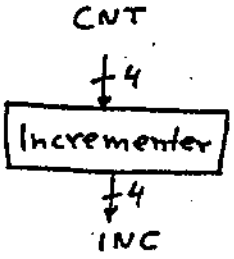

CSL2
Y
16
D
Q
16
clock
B

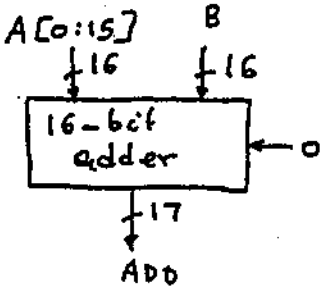

ADD[0]
CSL3
CSL1
CSL3
CSL4
clock
D
Q
C

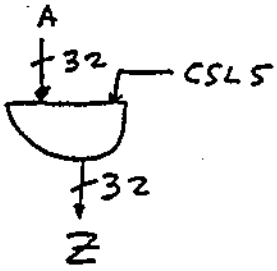

ADD[1:16]
16
CSL3
C, A[0:14]
16
CSL4
16
16
CSL1
CSL3
CSL4
clock
D
Q
16
A[0:15]


X
16
CSL1
A[15:30]
16
CSL4
16
16
16
CS4
CSL4
clock
D
Q
16
A[16:31]

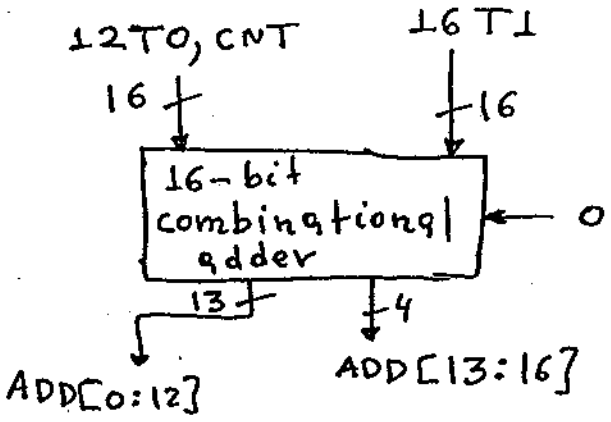

INC
4
CSL4
CSL1
4
CSL4
Clock
D
Q
4
CNT


CNT
4
Incrementer
4
INC


A[0:15]
B
16
16
16-bit
adder
0
17
ADD


A
32
CSL5
32
Z

For the previous design, (the AHPL description on page 6), one can use the combinational adder to perform the counter (CNT) incrementation in step 4. The situation looks as follows:

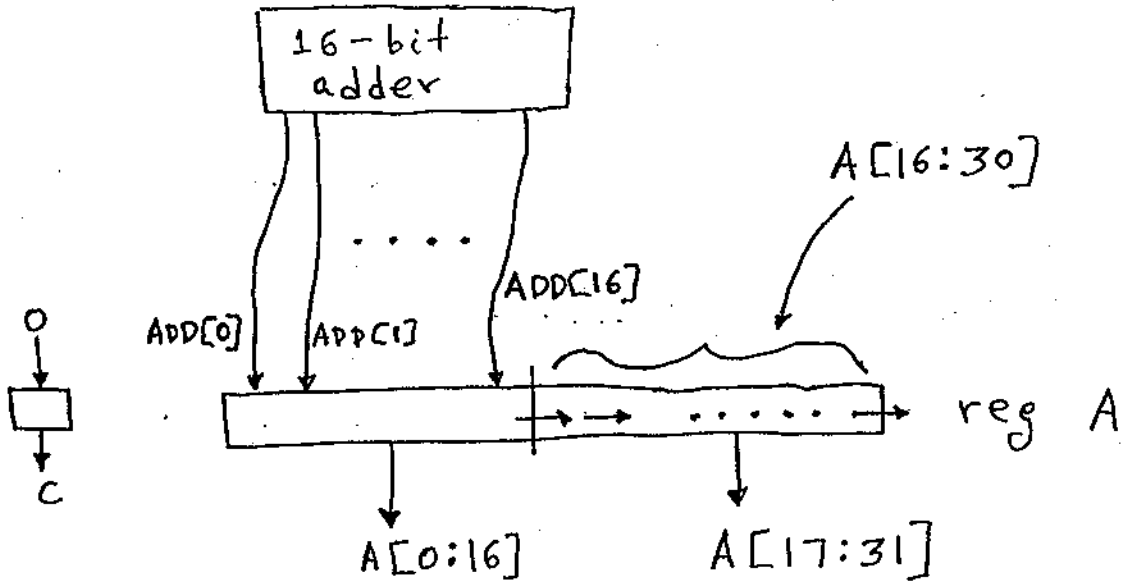


The AHPL description of page 6 can then be rewritten as:

```
MODULE: UNSIGNEDMULT.
MEMORY:
INPUTS:
OUTPUTS:        } same as before.
CLUNITS: ADD[17]<: ADDER{16}.
1 ....
2 ....          } same as before.
3 ...
4 C, A ← 0, C, A[0:30];
  CNT ← ADD[13:16](12TO, CNT; 16T1; 0);
  → (/(1/CNT))/(2).
5 ... same as before.
  END SEQUENCE.
  END.
```

We can now reduce the number of steps in the AHPL description of page 6. Steps 2, 3, 4 can be combined into one step. The responsibilities associated with this new step will be: testing A[31] and doing either shift only or add/shift; affecting CNT and testing CNT.

The add/shift done in one cycle is clarified below:

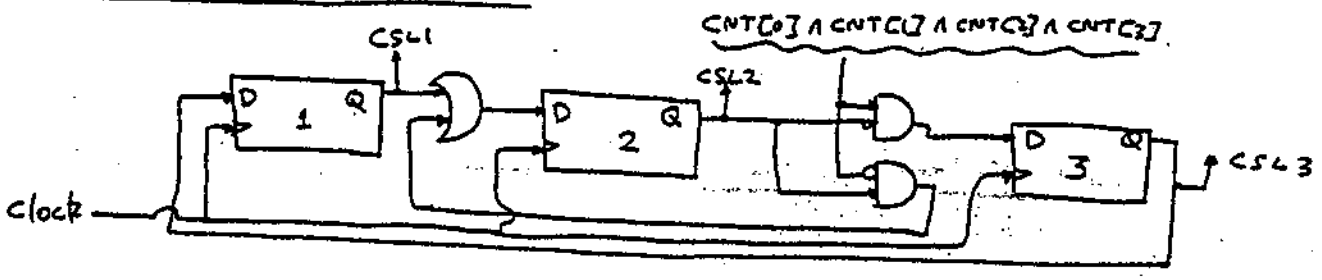


The reduced AHPL description follows.

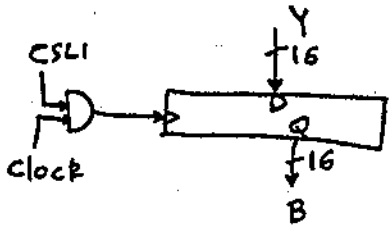
```
MODULE: UNSIGNEDMULT.
MEMORY: A[32]; B[16]; CNT[4]; C.
INPUTS: X[16]; Y[16].
OUTPUTS: Z[32].
CLUNITS: ADD[17]<: ADDER{16};
         INC[4]<: INCREMENTER{4}.

1 C, A[0:15]←17TO; A[16:31]←X; B←Y;
  CNT←4TO.

2 C; A←((0,C,A[0:30])!(0,ADD(A[0:15]; B; 0), A[16:30]))*
                                    (Ā[31], A[31]);
  CNT←INC(CNT); →(∧/CNT)/(2).

3 Z=A; →(1).
END SEQUENCE
END.
```

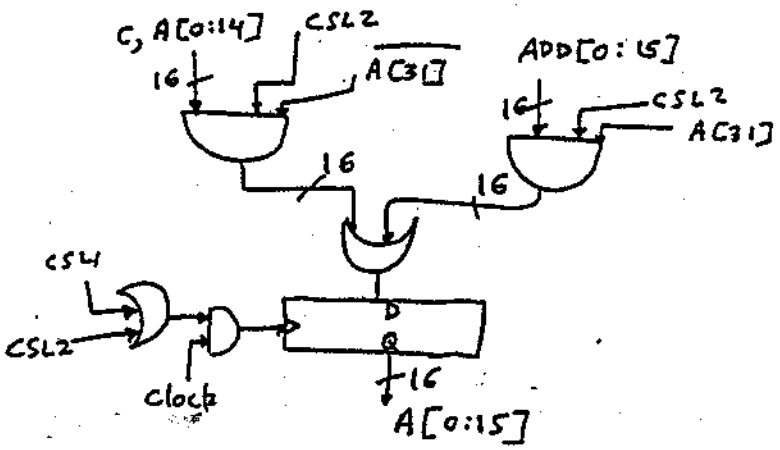
Diagrams for the control and data part are shown on the next page.

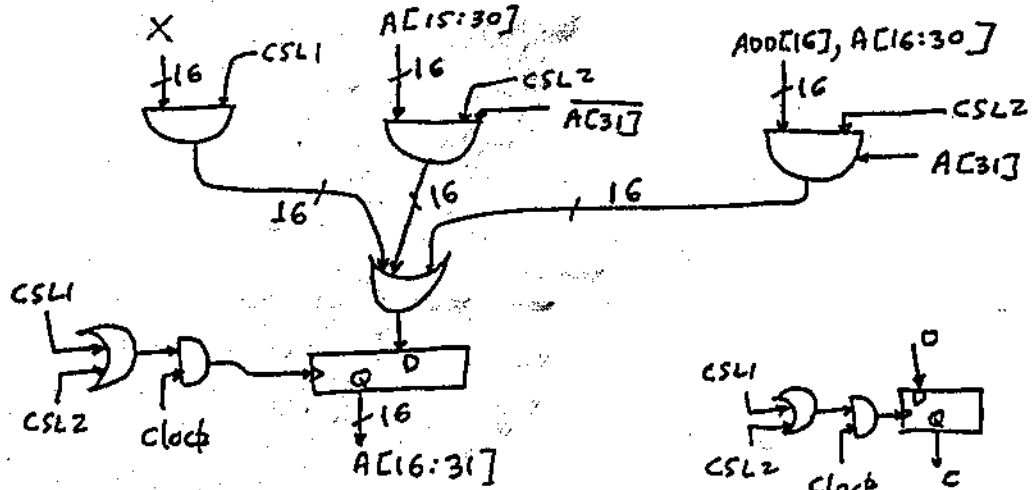
Hardwired Controller

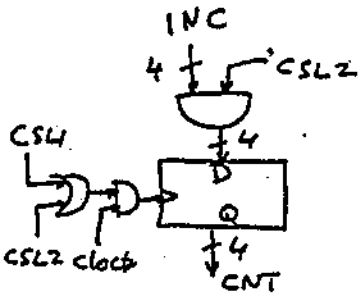


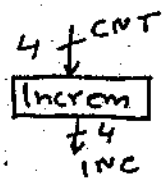
Data part

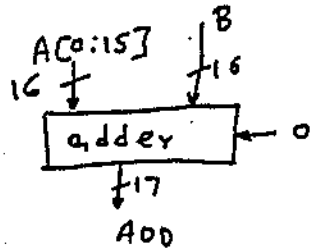












* You can easily notice that the flip flop C is not actually needed (it is useless).

Additional useful combinational logic units (CLUNITS)

- **Decoder**

$X = X[0], X[1], \cdots, X[n-1]$

n

[n-to-2^n decoder]

2^n

$DCD = DCD[0], DCD[1], \cdots, DCD[2^n-1]$

$DCD[0] = 1$ if $X[0], X[1], \cdots, X[n-1] = 0, 0, \cdots 0, 0$

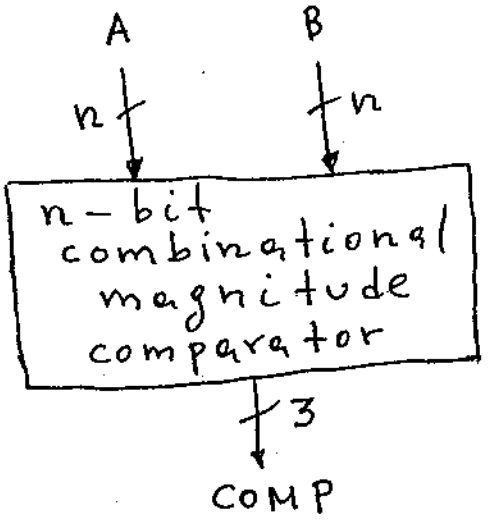
$DCD[1] = 1$ if $X[0], X[1], \cdots, X[n-1] = 0, 0, \cdots 0, 1$

$DCD[2] = 1$ if $X[0], X[1], \cdots, X[n-1] = 0, 0, \cdots 1, 0$

$\vdots$

$DCD[2^n-1] = 1$ if $X[0], X[1], \cdots, X[n-1] = 1, 1, \cdots 1, 1$

- Magnitude comparator



Here, the vectors A and B represent magnitudes.

COMP = COMP[0], COMP[1], COMP[2] = 1, 0, 0 if (A) > (B)
= 0, 1, 0 if (A) = (B)
= 0, 0, 1 if (A) < (B)

In the above, (A) means value of number A while (B) means value of number B.

Of course, an adder can also be used to perform magnitude comparison.

EE 3755

Problem: You are asked to design a multiplier for multiplying two 16-bit signed numbers to produce their 32-bit product. The system used for representing signed numbers is the 2's complement system. The algorithm used should be the simplest version of the Booth algorithm (examining two bits at a time). Inputs to your multiplier are two 16-bit input vectors X and Y (X is used for the multiplier; Y for the multiplicand). The output of your multiplier is a 32-bit vector, Z (the product).

You are asked to:

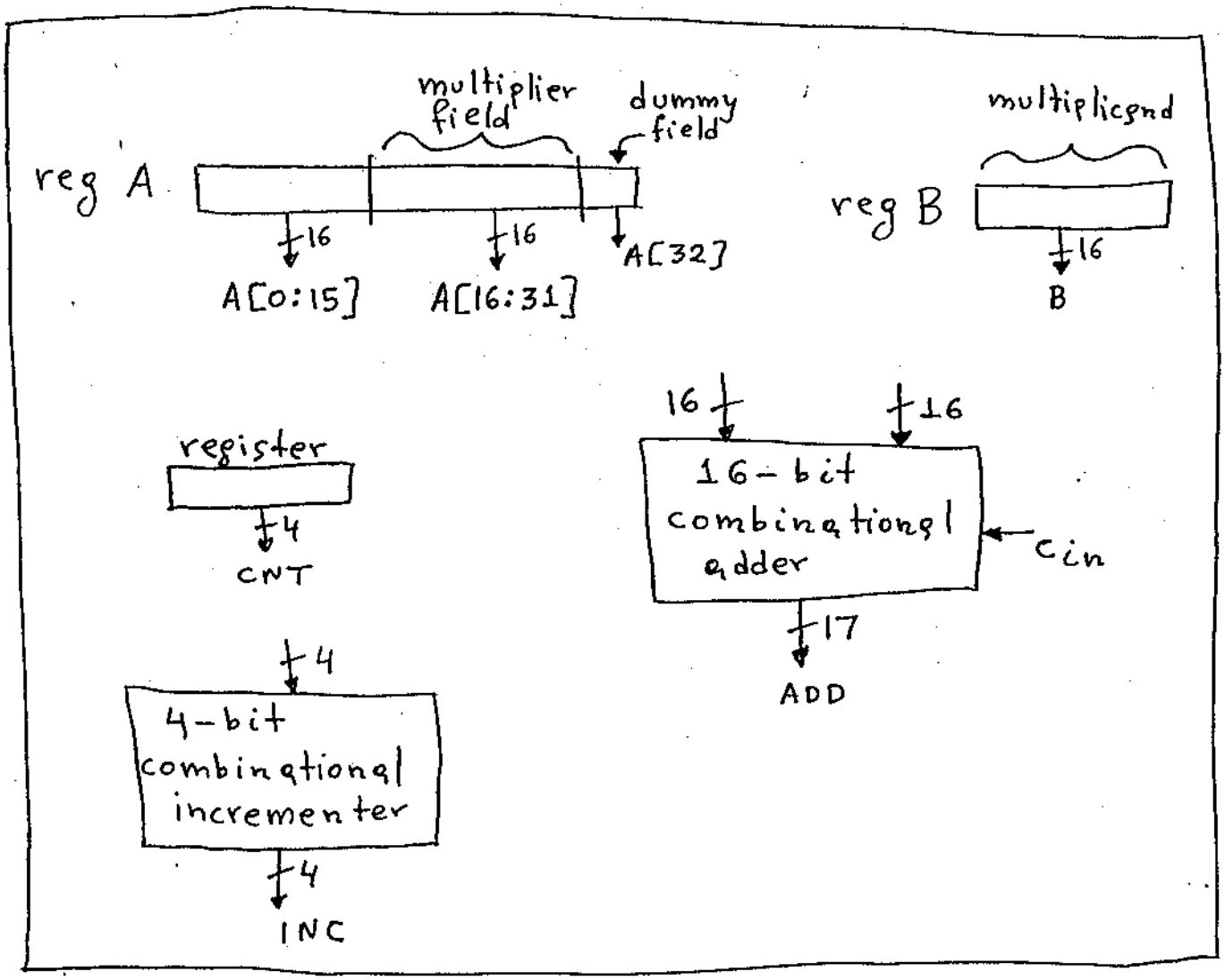
- Provide a complete AHPL description of your design including declarations.
- Show in logic block diagram form the design of the control part and data part.

Solution: The multiplier will consist of the following components:

- A 33-bit register named A. The part A[0:15] will be initialized with zeros, the part A[16:31] will be initialized with the multiplier while A[32] will be the dummy bit flip-flop (initialized with zero).
- A 16-bit register B to store the multiplicand.

(2) i

- A 16-bit combinational adder. The output of the adder is a 17-bit vector named ADD with ADD[0] being the carry out; (the line ADD[0] will not be used).
- A 4-bit counting register CNT used in counting the 16 Booth cycles.
- A 4-bit combinational incrementer used in incrementing by one the contents of CNT. The output of the incrementer is a 4-bit vector named INC.



(3) i

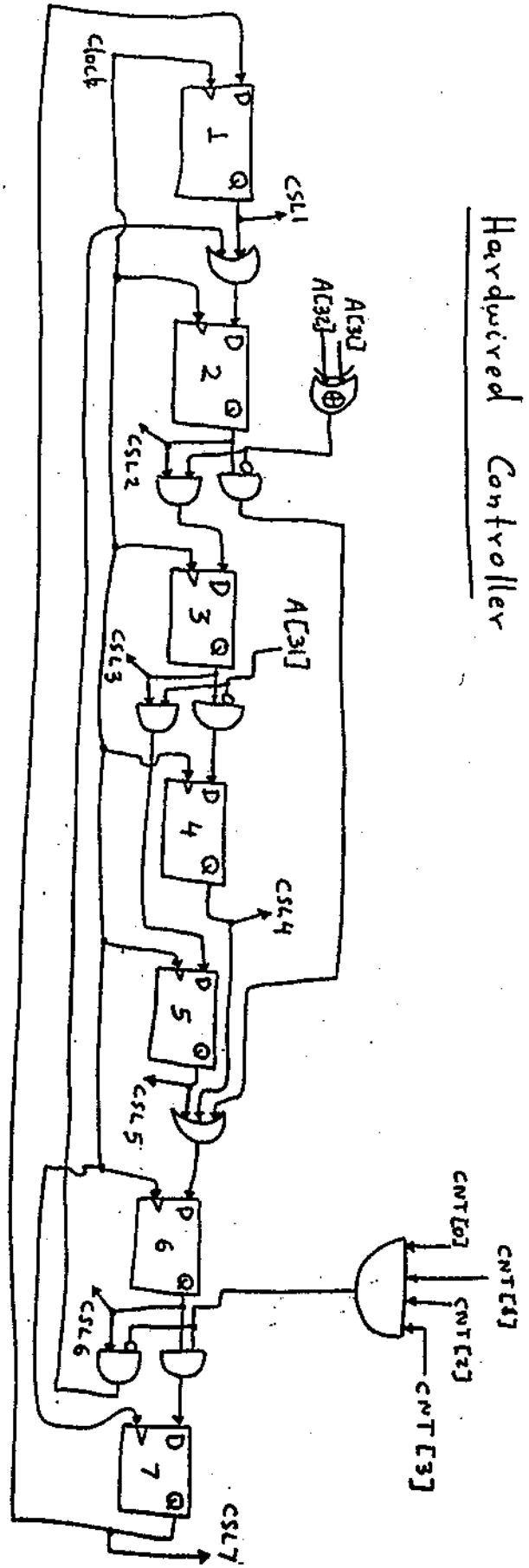
The following shows an AHPL description of the Booth multiplier:

```
MODULE: BOOTHMULT.
INPUTS: X[16]; Y[16].
OUTPUTS: Z[32].
MEMORY: A[33]; B[16]; CNT[4].
CLUNITS: ADD[17]<:ADDER{16}; INC[4]<:INCREMENTER{4}.
1 A[0:15], A[32]← 17TO; A[16:31]← X; B← Y;
  CNT ← 4TO.
2 → (~(A[31]⊕A[32]))/(6).
3 → (A[31])/(5).
4 A[0:15]← ADD[1:16](A[0:15]; B; 0); →(6).
5 A[0:15]← ADD[1:16](A[0:15]; ~B; 1).
6 A← A[0], A[0:31]; CNT← INC(CNT);
  →(~(∧/CNT))/(2).
7 Z = A[0:31]; →(1).
  END SEQUENCE
  END
```

Diagrams for the control and data part follow.

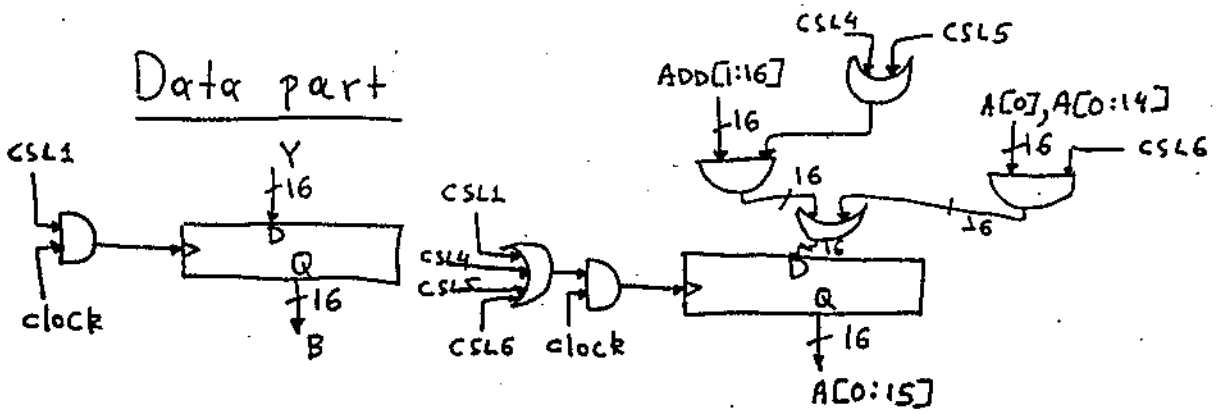
④ i

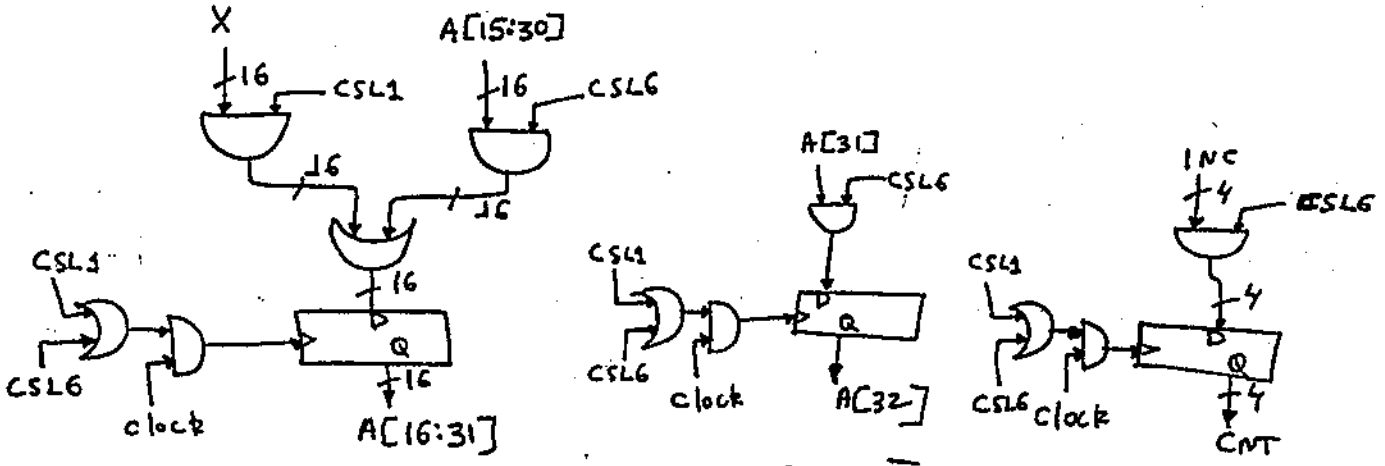
Hardwired Controller

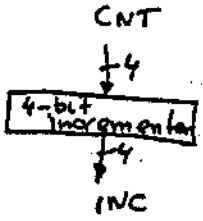


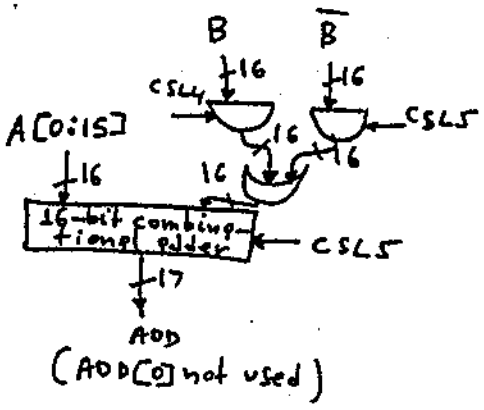
⑤ i

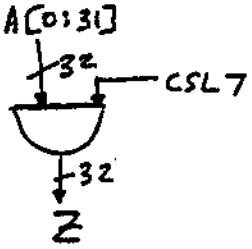
Data part


CSL1
Y
16
D
Q
16
B
clock
CSL1
CSL4
CSL5
CSL6
clock
ADD[1:16]
16
CSL4
CSL5
16
A[0], A[0:14]
16
CSL6
16
16
16
D
Q
16
A[0:15]


X
16
CSL1
A[15:30]
16
CSL6
16
16
CSL1
CSL6
clock
16
D
Q
16
A[16:31]
A[31]
CSL6
CSL1
CSL6
clock
D
Q
A[32]
INC
4
CSL6
CSL1
CSL6
clock
4
D
Q
4
CNT


CNT
4
4-bit incrementer
4
INC


B
B̄
16
16
CSL4
CSL5
A[0:15]
16
16
16
16-bit combinational adder
CSL5
17
ADD
(ADD[0] not used)


A[0:31]
32
CSL7
32
Z

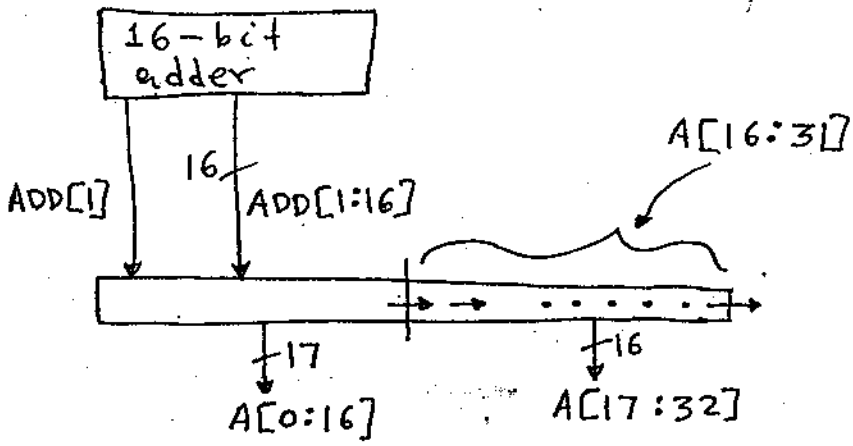
(6) We can now reduce the number of steps in the AHPL description of page 3. Steps 2, 3, 4, 5, 6 can be combined into one step. The reduced AHPL description follows.

```
MODULE: BOOTHMULT.
INPUTS: X[16]; Y[16].
OUTPUTS: Z[32].
MEMORY: A[33]; B[16]; CNT[4].
CLUNITS: ADD[17]<: ADDER{16}; INC[4]<: INCREMENTER{4}.
1 A[0:15], A[32] ← 17TO; A[16:31] ← X; B ← Y; CNT ← 4TO.
2 A ← ((A[0], A[0:31]) ! (ADD[1](A[0:15]; (B ! B̄) * (A[31]‾, A[31]); A[31]),
  ADD[1:16](A[0:15]; (B ! B̄) * (A[31]‾, A[31]); A[31]), A[16:31])) *
    (A[31] ⊕ A[32])‾, A[31] ⊕ A[32]);
  CNT ← INC(CNT); → (∧/CNT)‾/(2).
3 Z = A[0:31]; → (1).
END SEQUENCE.
END.
```

Explanation of step 2 of AHPL description of page 6.

The responsibilities associated with step 2 of the description on page 6 are: testing A[31:32] and performing either shift or add/shift or subtract/shift; affecting CNT and testing CNT.

The add/shift or subtract/shift done in one cycle is clarified by the figure below.



step 2 now looks like:

2 A ← ((A[0], A[0:31]) ! (ADD[1](arg), ADD[1:16](arg), A[16:31]))
* ($\overline{A[31] \oplus A[32]}$, A[31] ⊕ A[32]);
CNT ← INC(CNT); → ($\overline{\wedge/CNT}$)/(2).

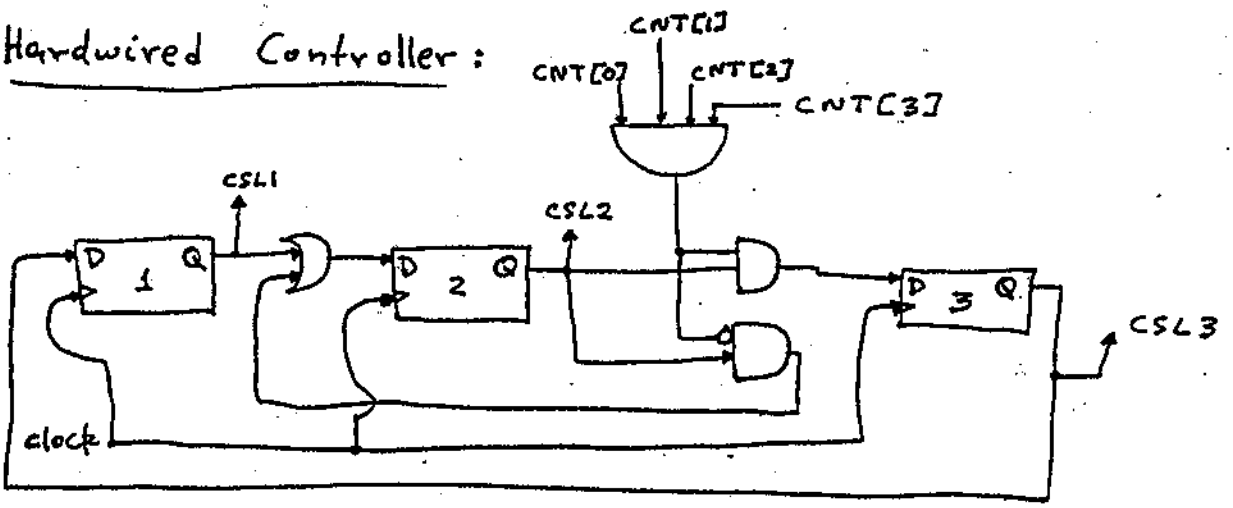
where

arg = A[0:15]; (B ! $\overline{B}$) * ($\overline{A[31]}$, A[31]); A[31]

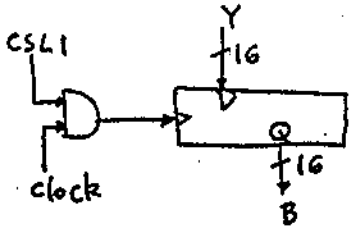
(8) i

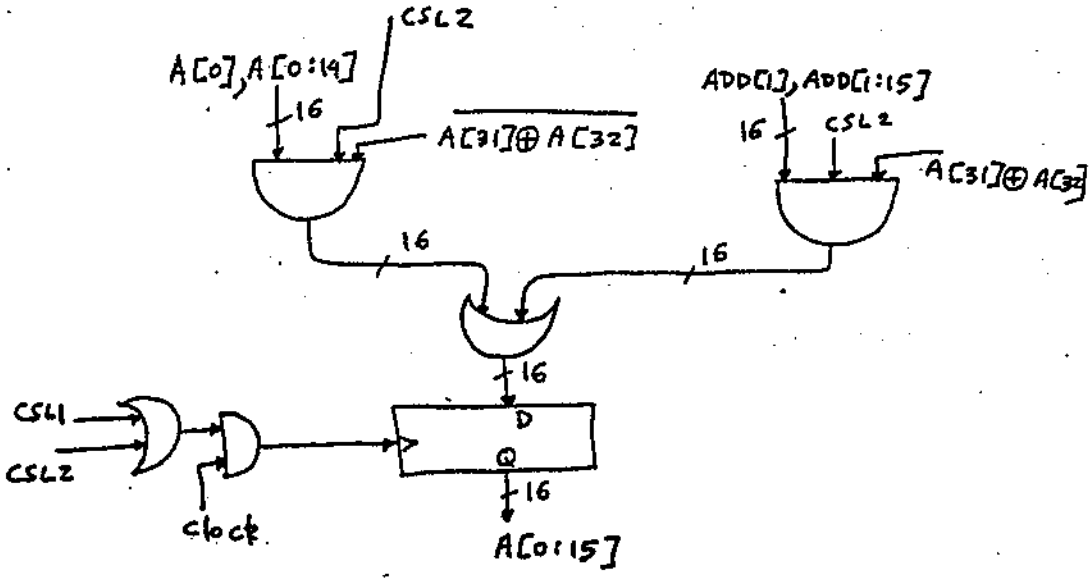
Diagrams for the control and data part follow.

Hardwired Controller:



Data Part:





Data part cont

(9) i.

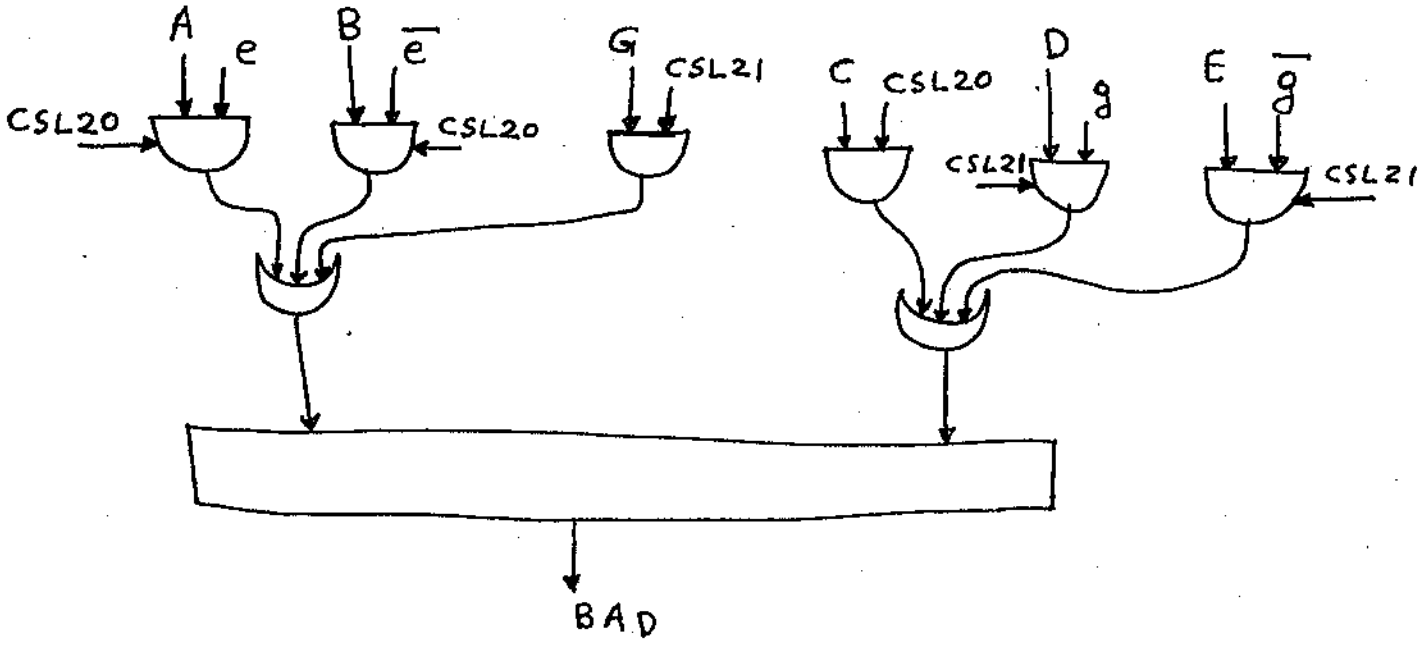
EE 3755

(1) j

Consider the following incomplete AHPL description

```
MODULE:.....
INPUTS:....
  ⋮
CLUNITS: BAD[ ]<: ; ....
 1...
 2...
 ⋮
20 D ← BAD((A ! B) * (e, ē); C).
21 E ← BAD( G; (D ! E) * (g, ḡ)).
22 ...
 ⋮
```

Show in logic block diagram form the connections to the inputs of the CLUNIT BAD.



(1) k

```
MODULE: ....
   :
1 ...
2 ...
3 A ← B
4 ...
5 Z = R ; ω = 1
6 A ← D
   :
END SEQUENCE
P = Q ; C ← D
END
```