EE 3755, Fall 2002

HW # 2 Solutions

EE 3755 [redacted]
HW # 2 Solutions [redacted]

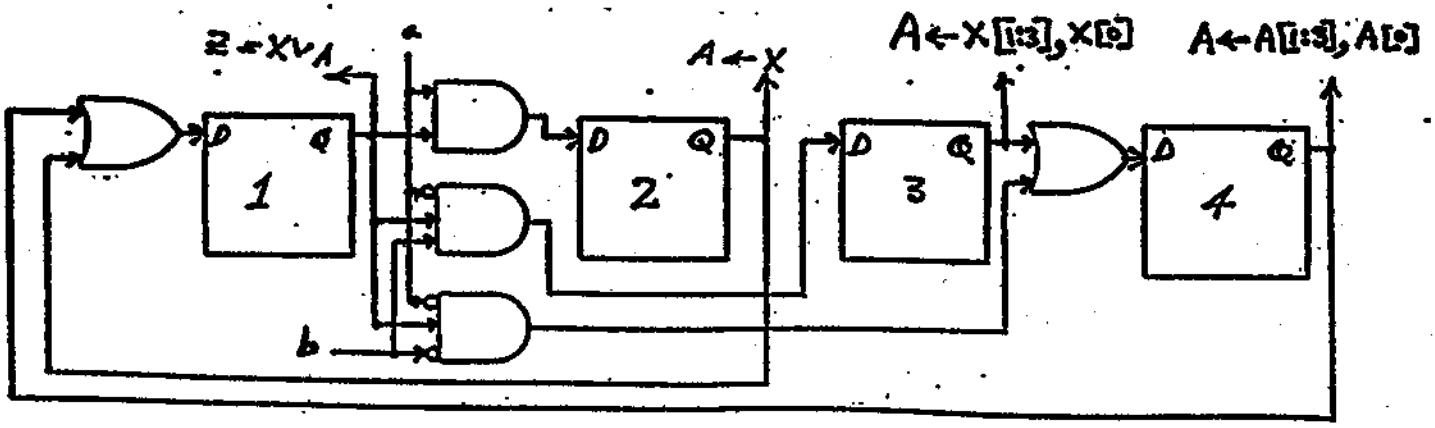
1

1 $B \leftarrow A$;
$\rightarrow (\overline{a})/(3)$.

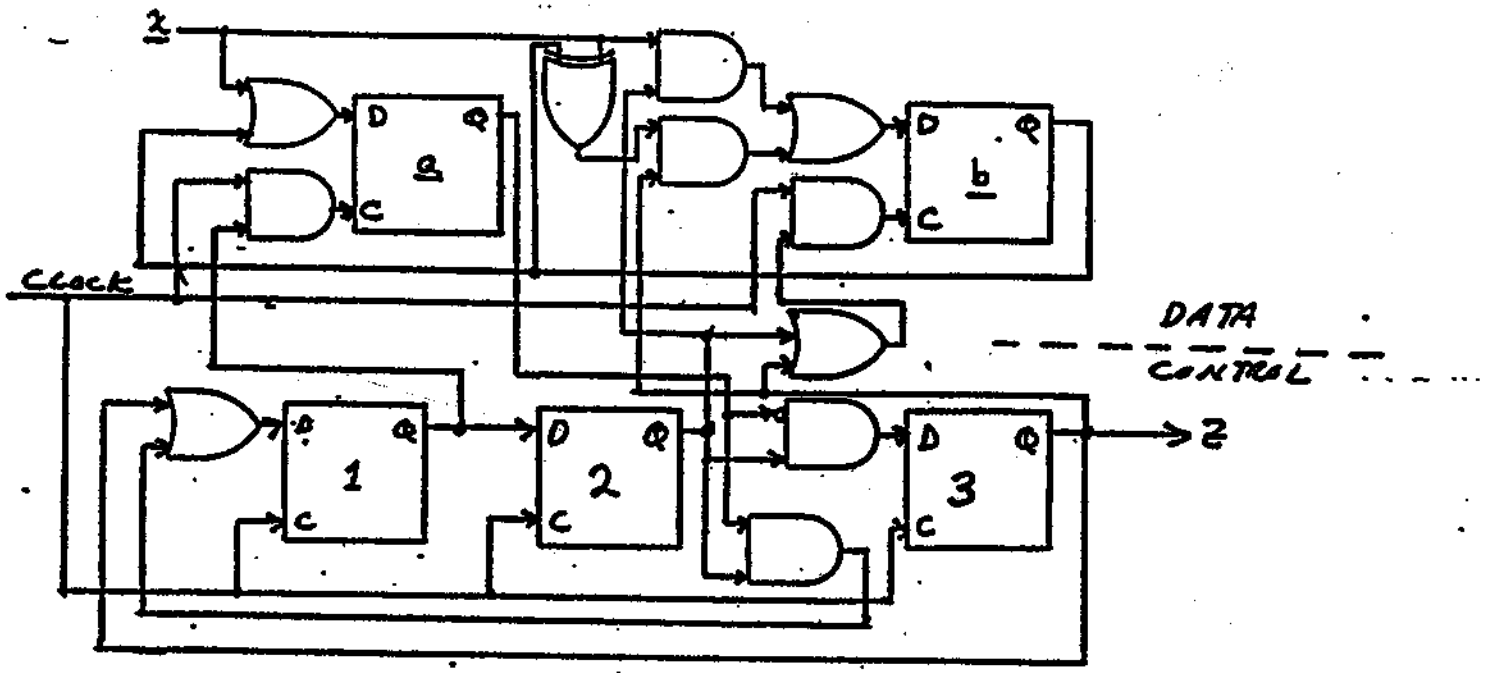
2 $A \leftarrow A[1:3], A[0]$;
$\rightarrow (\overline{b})/(2)$.

3 $B \leftarrow x \oplus A$; $Z = A$;
$\rightarrow (1)$.

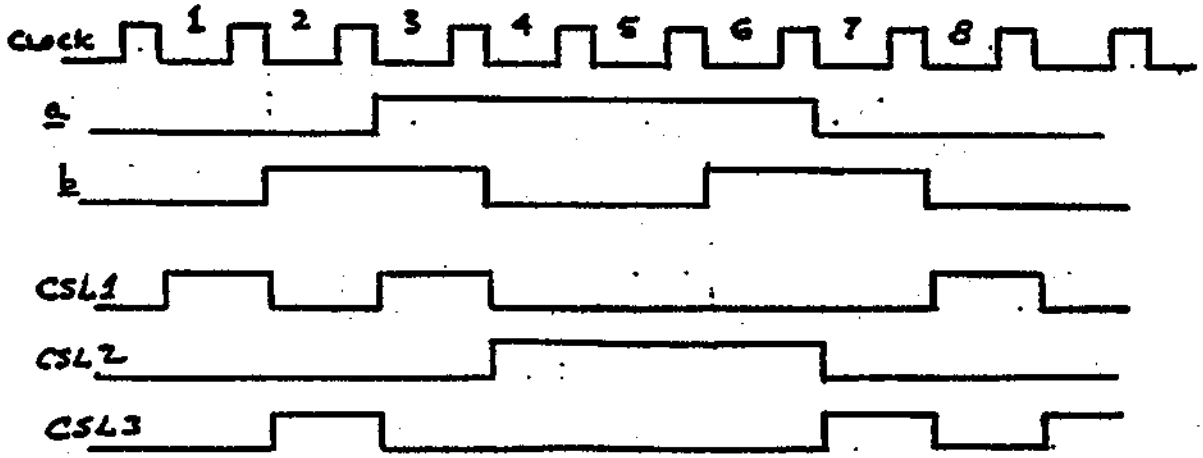
2



3


a
Clock
D
Q
C
a
b
1
2
3
DATA
CONTROL
Z

4


clock
1
2
3
4
5
6
7
8
a
b
CSL1
CSL2
CSL3

5

```
MODULE:
MEMORY: B[3]
INPUTS: X[2]; a
OUTPUTS: z
 1  B[0:1] ← X;
      → (ā)/(3).
 2  B ← B[2], B[0:1];
      → (1).
 3  B[1] ← B[0] ∧ B[1].
 4  B[2] ← B[1] ⊕ B[2];
      → (a, ā)/(1,2).
   END SEQUENCE
      z = B[2].
   END.
```

6.

In the description below P and R are 8-bit registers for storing the previous two X vectors (R is used for the oldest of the previous two vectors). CNT is a 2-bit counting register to count 4 periods for z=1 while d is a flip-flop to be set if current X equals contents of P or R. The vector INC[2] is the 2-bit output of a 2-bit incrementer. The following is a possible description (not the only one neither the most optimal).

```
MODULE:...
MEMORY: P[8]; R[8]; CNT[2]; d
INPUTS: X[8]; ready.
OUTPUTS: z; ask.
CLOCKS: INC[2] <: INCREMENTER{2}.
1 ask=1
2 d*ready ← ∨/(X⊕P) ∨ ∨/(X⊕R);
   → (ready)/(2).
3 R ← P; P ← X; CNT ← 2T0;
  → (d)/(1).
4 z=1; CNT ← INC(CNT);
  → (∧/CNT, ∧/CNT)/(1,4).
END SEQUENCE
END.
```

7 I can reduce the eight-step description into a three-step description. In the new description, the steps 2, 3, 4, 5, 6, 7 of the original description will combine in one step.

The following table can help in the reduction.

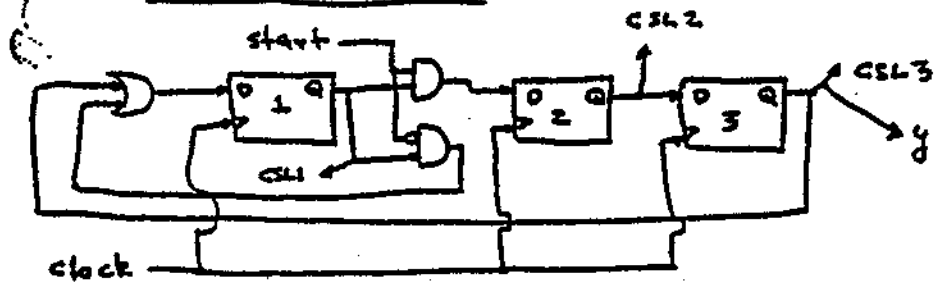
a b c	transfers
0 0 0	q ← p; r ← p
0 0 1	q ← p; r ← p
0 1 0	q ← p; p ← r
0 1 1	q ← p
1 0 0	r ← q; p ← q
1 0 1	r ← q
1 1 0	p ← r
1 1 1	no transfer

The reduced AHPL description follows

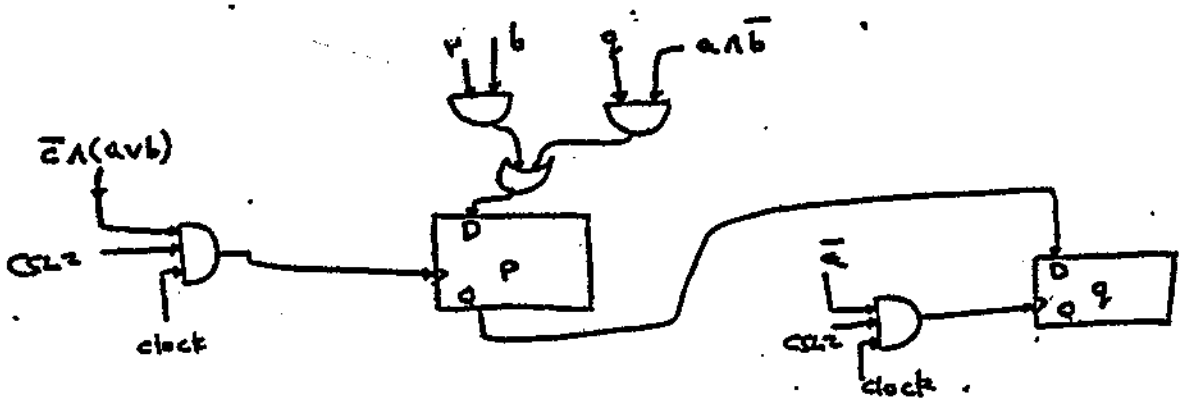
```
MODULE: EASY
MEMORY: p; q; r.
INPUTS: start; a; b; c.
OUTPUTS: y.

1 → (start)/(1).
2 p * (c̄ ∧ (a ∨ b)) ← (r ! q) * (b, a ∧ b̄);
  q * ā ← p;
  r * b̄ ← (p ! q) * (ā, a).
3 y = 1;
  → (1).
  END SEQUENCE
  END.
```

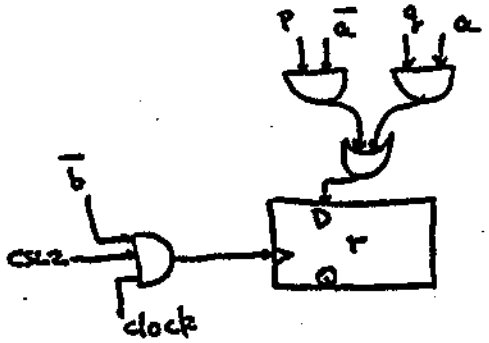
We can't reduce to ~~never~~ less ~~than~~ three steps because we need one step to observe start, one step to complete clocked transfers and one step to create y=1 for one clock period.

Design of controller



Design of data part:





EE 3755 [redacted]
Hw #2 Solutions [redacted]

8

⋮

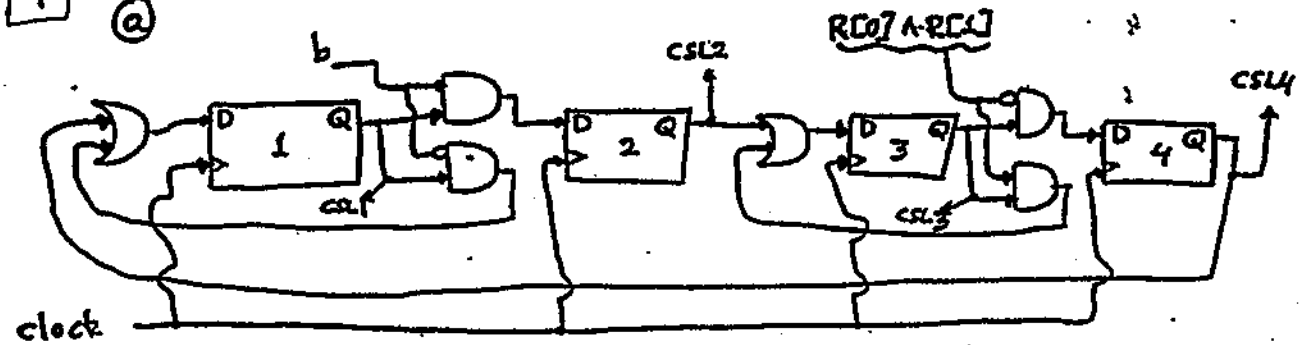
25 P ← s, t, q, r.

26 B* P[0] ← A⊕C; D*(P[0] ∧ P[2]) ← A⊕C⊕D;
F*(P[0] ∧ $\overline{P[3]}$) ← (D⊕E ! A⊕C⊕D⊕E) * ($\overline{P[2]}$, P[2]);
→ ($\overline{P[0]}$ ∧ $\overline{P[1]}$, $\overline{P[0]}$ ∧ P[1])/(28, 29).

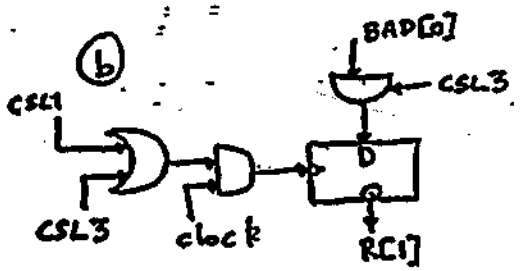
27 ...
28 ...
29 ...

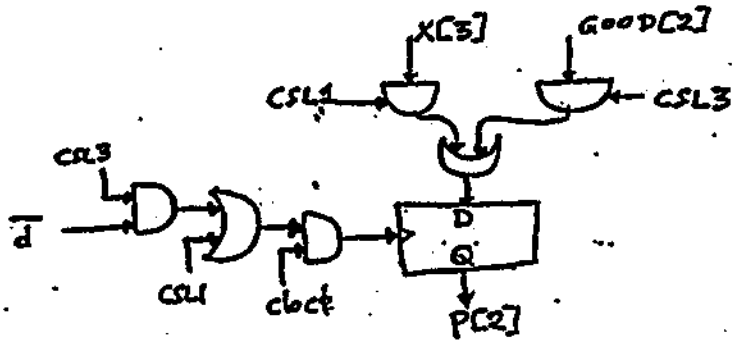
9

@

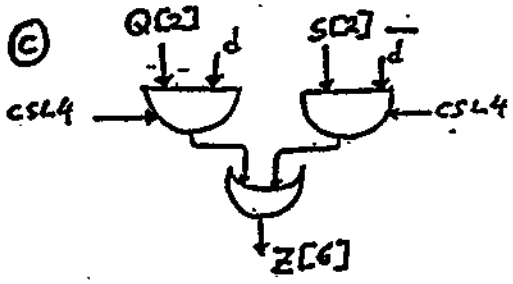

b
CSL2
R[0] ∧ R[1]
CSL4
D
Q
1
2
3
4
CSL1
CSL3
clock

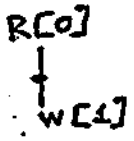
(b)


BAD[0]
CSL3
CSL1
CSL3
clock
D
Q
R[1]

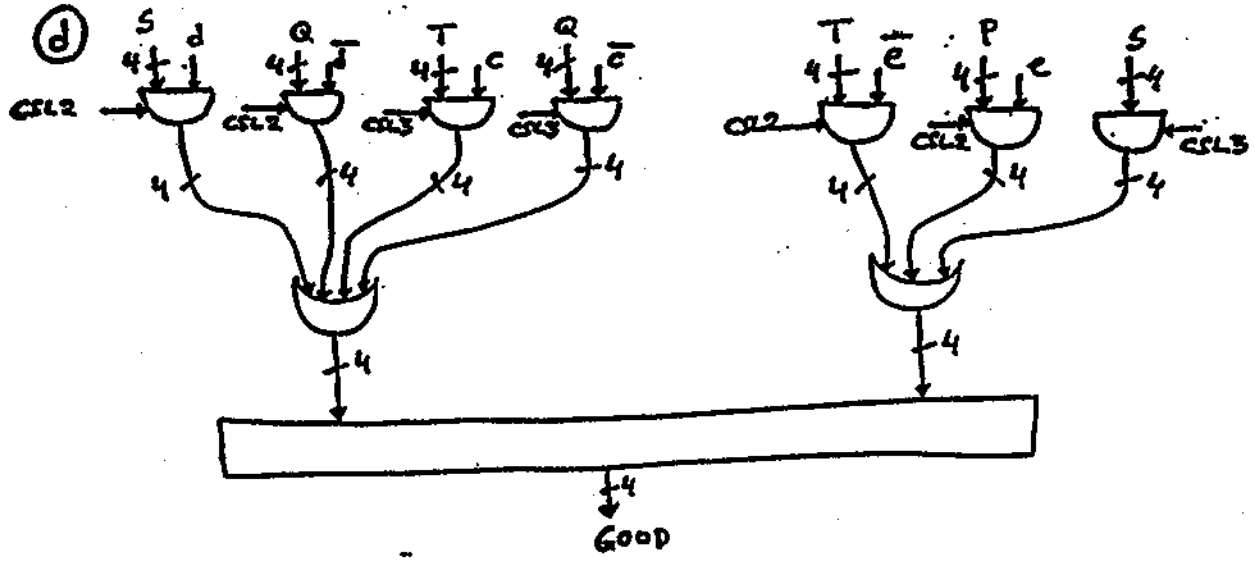

X[3]
GOOD[2]
CSL1
CSL3
CSL3
d̄
CSL1
clock
D
Q
P[2]

(c)


Q[2]
d
S[2]
d̄
CSL4
CSL4
Z[6]


R[0]
W[1]

(d)


S
d
Q
T
c
Q
T
e
P
c
S
CSL2
CSL2
CSL3
CSL3
CSL2
CSL2
CSL3
4
Good

10 Our multiplier will consist of the following components:

- A 34-bit register A. The field A[0:16] (which is a 17-bit field) will be the field left of the multiplier field and will be initialized with zeros. The 16-bit field A[17:32] will be the multiplier field (it will be initialized with the multiplier). The field A[33] will be the dummy field (it will be initialized with zero).
- A 16-bit register B to store the multiplicand.
- A 3-bit counting register CNT to count the eight multiplication cycles.
- A 3-bit combinational incrementer for incrementing the counter. The 3-bit output vector of the incrementer is named INC.
- A 17-bit combinational adder with its 18-bit output vector named ADD. The leftmost line ADD[0] is going to be the carry out line so it is not going to be used.
- A 3-to-8 decoder; (a decoder with a 3-bit input vector and an 8-bit output vector). The input to the decoder is going to be A[31:33] (the two rightmost bits of the multiplier field together with the dummy field). The 8-bit output vector of the decoder is named DCD.

A complete AHPL description including declarations follows.

```
MODULE: MULTIPLIER
INPUTS: X[16]; Y[16].
OUTPUTS: Z[32].
MEMORY: A[34]; B[16]; CNT[3].
CLUNITS: ADD[18]<: ADDER{17}; INC[3]<: INCREMENTER{3};
         DCD[8]<: DECODER{3}.
```

→ continues on next page

1 A[0:16], A[33], CNT ← 21 TO; A[17:32] ← X; B ← Y.

2 A ← ((A[0], A[0], A[0:31])

! (ADD[1](arg1), ADD[1](arg1), ADD[1:17](arg1), A[17:31]))*

* (DCD[0](arg2) V DCD[7](arg2), $\overline{\text{DCD[0](arg2) V DCD[7](arg2)}}$);

CNT ← INC(CNT); → ($\overline{\Lambda/\text{CNT}}$) /(2).

3 Z = A[1:32]; → (1).

END SEQUENCE

END

In the above the input arguments of the adder are

arg1 = A[0:16]; ((B[0], B)!(B, 0)! ($\overline{B}$, 1)! ($\overline{B[0]}$, $\overline{B}$)) *

* (DCD[1](arg2) V DCD[2](arg2), DCD[3](arg2), DCD[4](arg2), DCD[5](arg2) V DCD[6](arg2));

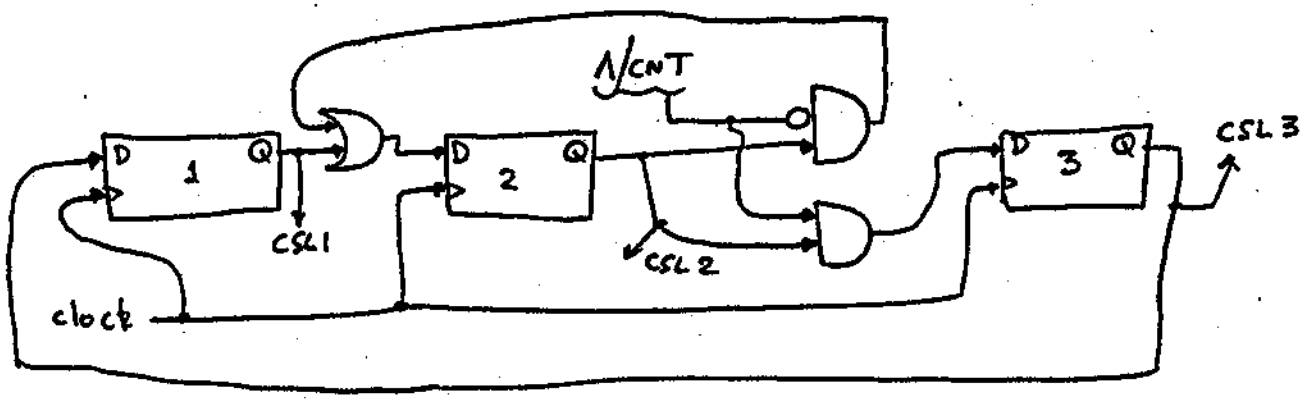
DCD[4](arg2) V DCD[5](arg2) V DCD[6](arg2).

The input argument of the decoder is

arg2 = A[31:33].

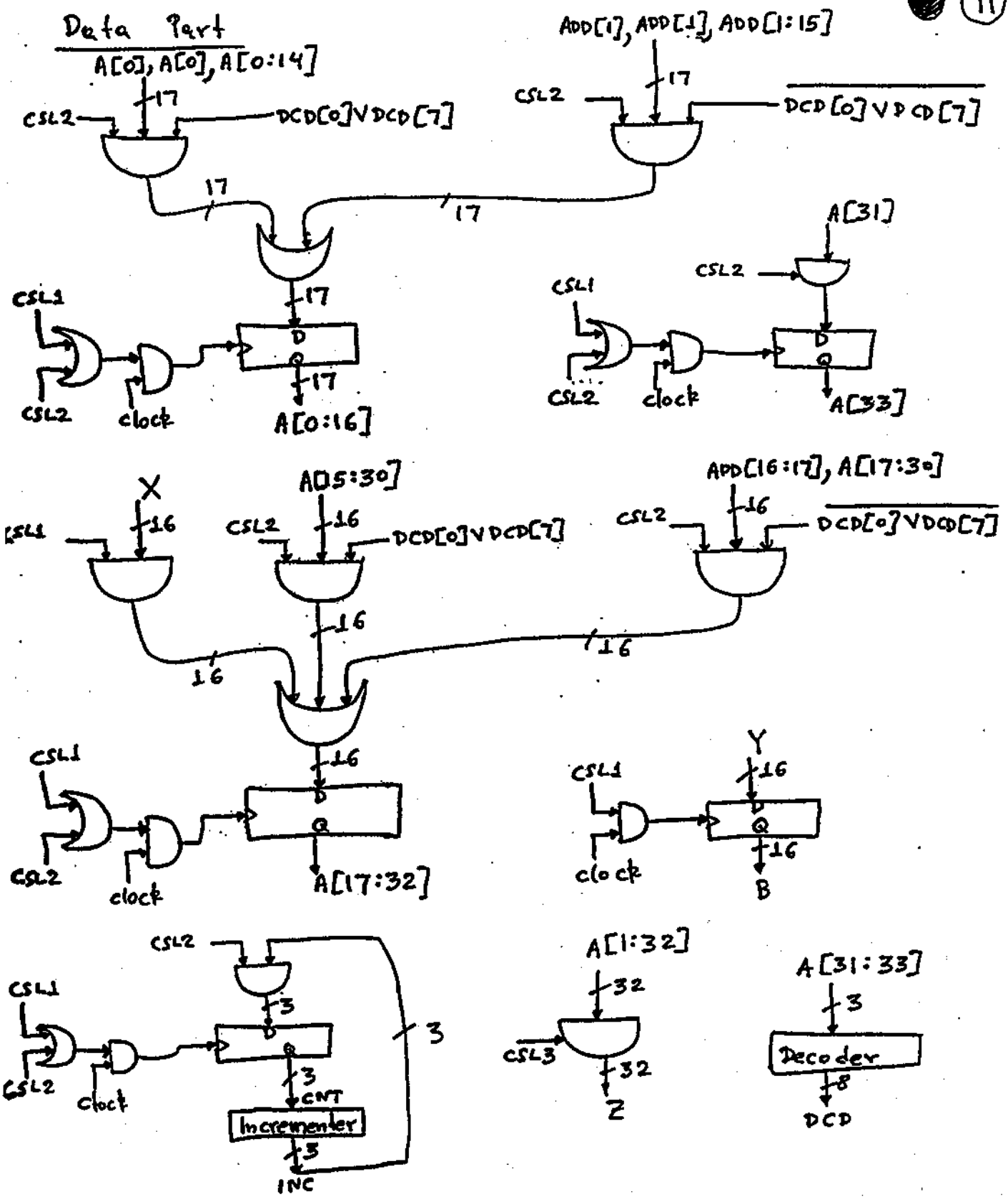
In the above expression for arg1; B[0], B = sign extended multiplicand; B, 0 = 2 x multiplicand; $\overline{B}$, 1 = 1's complement of (2 x multiplicand); $\overline{B[0]}$, $\overline{B}$ = 1s complement of sign extended multiplicand.

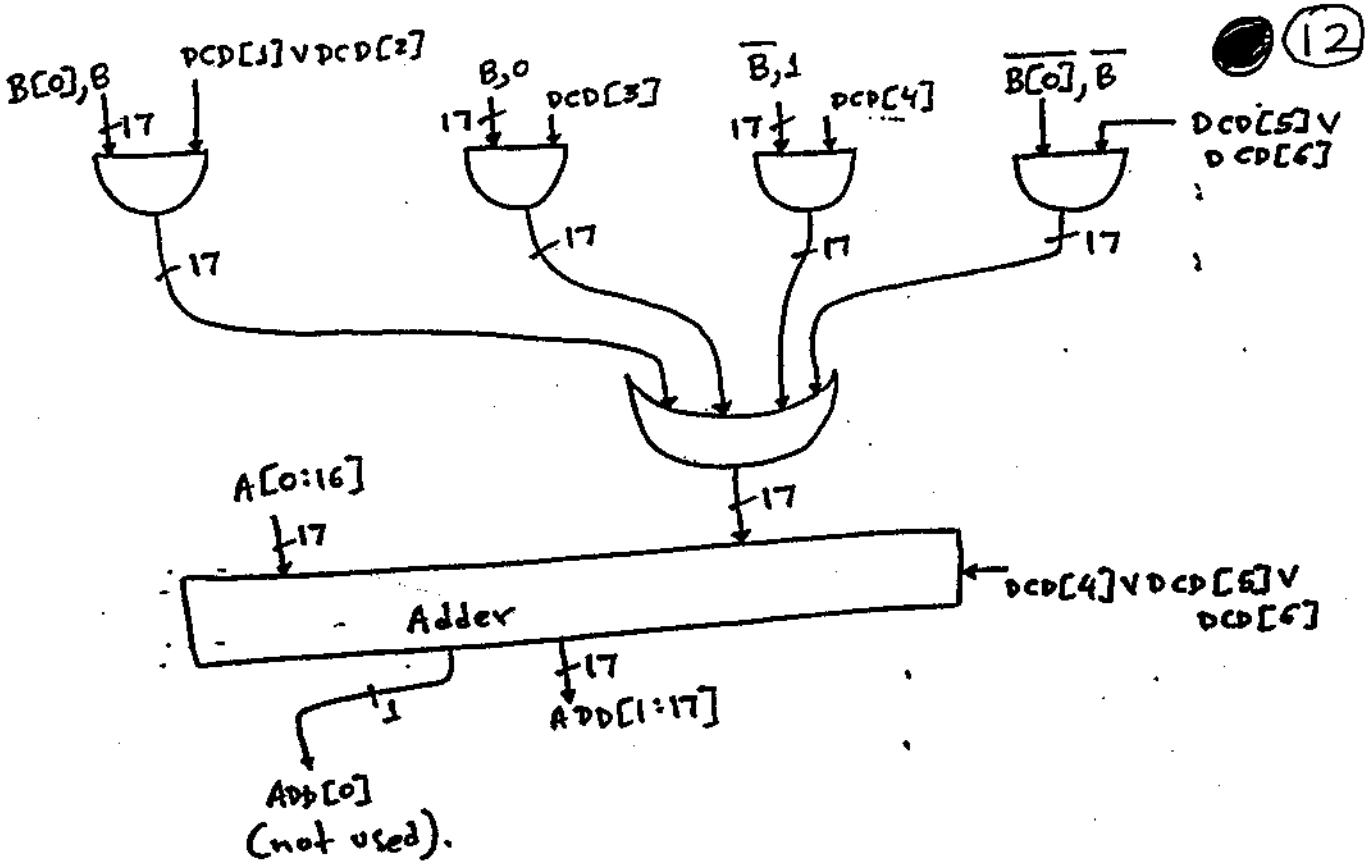
(b)



Controller

Data Part


A[0], A[0], A[0:14]
17
CSL2
DCD[0]VDCD[7]
ADD[1], ADD[1], ADD[1:15]
17
CSL2
DCD[0]VDCD[7]
17
17
17
CSL1
CSL2
clock
D
Q
17
A[0:16]
A[31]
CSL2
CSL1
CSL2
clock
D
Q
A[33]
X
16
CSL1
A[15:30]
16
CSL2
DCD[0]VDCD[7]
ADD[16:17], A[17:30]
16
CSL2
DCD[0]VDCD[7]
16
16
16
CSL1
CSL2
clock
D
Q
A[17:32]
Y
16
CSL1
clock
Q
16
B
CSL2
CSL1
CSL2
clock
3
D
Q
3
CNT
Incrementer
3
INC
3
A[1:32]
32
CSL3
32
Z
A[31:33]
3
Decoder
8
DCD


B[0],B
PCD[1] V PCD[2]
17
B,0
PCD[3]
$\overline{B}$,1
PCD[4]
$\overline{B[0]}$, $\overline{B}$
DCD[5] V DCD[6]
17
A[0:16]
Adder
DCD[4] V DCD[5] V DCD[6]
1
ADD[1:17]
ADD[0]
(not used).

MODULE: UNSIGNEDMULT
MEMORY: A[32]; B[16]; CNT[4].
INPUTS: Same as before
OUTPUTS: Same as before
CLOCKS: same as before
1 A[0:15] ← 16 TO 0; rest of step 1 is the same as before
2 A ← ((0, A[0:30]) ! (ADD(A[0:15]; B:0), A[16:30])) ¥ (A[31], A[31]);
rest of step 2 is the same as before
3 Same as before
END SEQUENCE
END.

12

MODULE: HW# 2
INPUTS: A[8]; B[8]; C[8]; D[8]; e; f.
OUTPUTS: Z[16].
MEMORY: P[17]; Q[16]; R[16]; CNT[3].
CLUNITS: ADD[9] <:ADDER{8}; INC[3] <: INCREMENTER{3}.
1 P ← 8TO, A, 0; CNT ← 3TO.
2 P ← ((P[0], P[0:15]) ! (ADD[1](arg1), ADD[1:8](arg1), P[8:15]))
* ($\overline{P[15] \oplus P[16]}$, P[15] ⊕ P[16]);
CNT ← INC(CNT); → ($\overline{\wedge / CNT}$)/(2).
3 Q ← P[0:15]; P ← 8TO, D, 0; CNT ← 3TO.
4 P ← ((P[0], P[0:15]) ! (ADD[1](arg2), ADD[1:8](arg2), P[8:15]))
* ($\overline{P[15] \oplus P[16]}$, P[15] ⊕ P[16]);
CNT ← INC(CNT); → ($\overline{\wedge / CNT}$)/(4).
5 P[16], Q[8:15] ← ADD(Q[8:15]; (P[8:15] ! $\overline{P[8:15]}$) * ($\overline{f}$, f); f).
6 R ← ADD[1:8](Q[0:7]; (P[0:7] ! $\overline{P[0:7]}$) * ($\overline{f}$, f); P[16]), Q[8:15];
→ (1).
END SEQUENCE
Z = R.
END

In the above

arg1 = (P[0:7]; (B ! $\overline{B}$! C ! $\overline{C}$) × ($\overline{e} \wedge \overline{P[15]}$, $\overline{e} \wedge P[15]$, $e \wedge \overline{P[15]}$, $e \wedge P[15]$); P[15]).

and

arg2 = (P[0:7]; (C ! $\overline{C}$! B ! $\overline{B}$) * ($\overline{e} \wedge \overline{P[15]}$, $\overline{e} \wedge P[15]$, $e \wedge \overline{P[15]}$, $e \wedge P[15]$); P[15]).

For the previous design of MODULE HW #2, P is a 17-bit register used by the Booth multiplication. The field P[8:15] is the multiplier field, P[0:7] is the field left of the multiplier field while P[16] is the dummy field. Q is a 16-bit register in which the first product (A×B or A×C) is stored. R is a 16-bit register in which the final result Z (the sum or difference of the two products) is stored. The CLUNIT ADD is an 8-bit adder used for the Booth multiplications as well as for calculating Z.

Step 1 is responsible for initializing the register P and the counting register CNT. Step 2 is responsible for the Booth multiplication with multiplier A and multiplicand either B or C. We don't have to use registers to store B or C as these inputs are available (they change values every 20 clock cycles). Step 3 is responsible for storing the first product (A×B or A×C) in the register Q and initialize the Booth algorithm so that the multiplication C×D or B×D can follow. Step 4 is responsible for the Booth multiplication with multiplier D and multiplicand either C or B. Finally steps 5 and 6 are responsible for calculating the desired result which gets stored in R.

The controller spends 1 clock cycle over step 1, 8 clock cycles over step 2, 1 clock cycle over step 3, 8 clock cycles over step 4, 1 cycle over step 5 and 1 cycle over step 6 (or totally 1+8+1+8+1+1 cycles = 20 cycles). Thus, the register R gets updated every 20 clock cycles and the output Z changes every 20 clock cycles as well.

MODULE:DIVIDER
INPUTS: X[32]; Y[16].
OUTPUTS: R[16]; Q[16]; ovf.
MEMORY: A[32]; B[16]; c; CNT[4].
CLUNITS: ADD[17]<:ADDER{16}; INC[4]<:INCREMENTER{4}.

1 c←1; A←X; B←Y; CNT←4TO;
→(ADD[0](X[0:15]; $\overline{Y}$; 1))/(3).

2 c, A[0:15]← ADD(A[1:16]; (B ! $\overline{B}$)*($\overline{c}$, c); c);
A[16:31]← A[17:31], ADD[0](A[1:16]; (B ! $\overline{B}$)*($\overline{c}$, c); c);
CNT←INC(CNT); →($\overline{\wedge/CNT}$, $\wedge/CNT$)/(2,4).

3 ovf=1; →(1).

4 Q = A[16:31]; R=(A[0:15] ! ADD[1:16](A[0:15]; B; 0))*(c, $\overline{c}$);
→(1).

END SEQUENCE.
END.

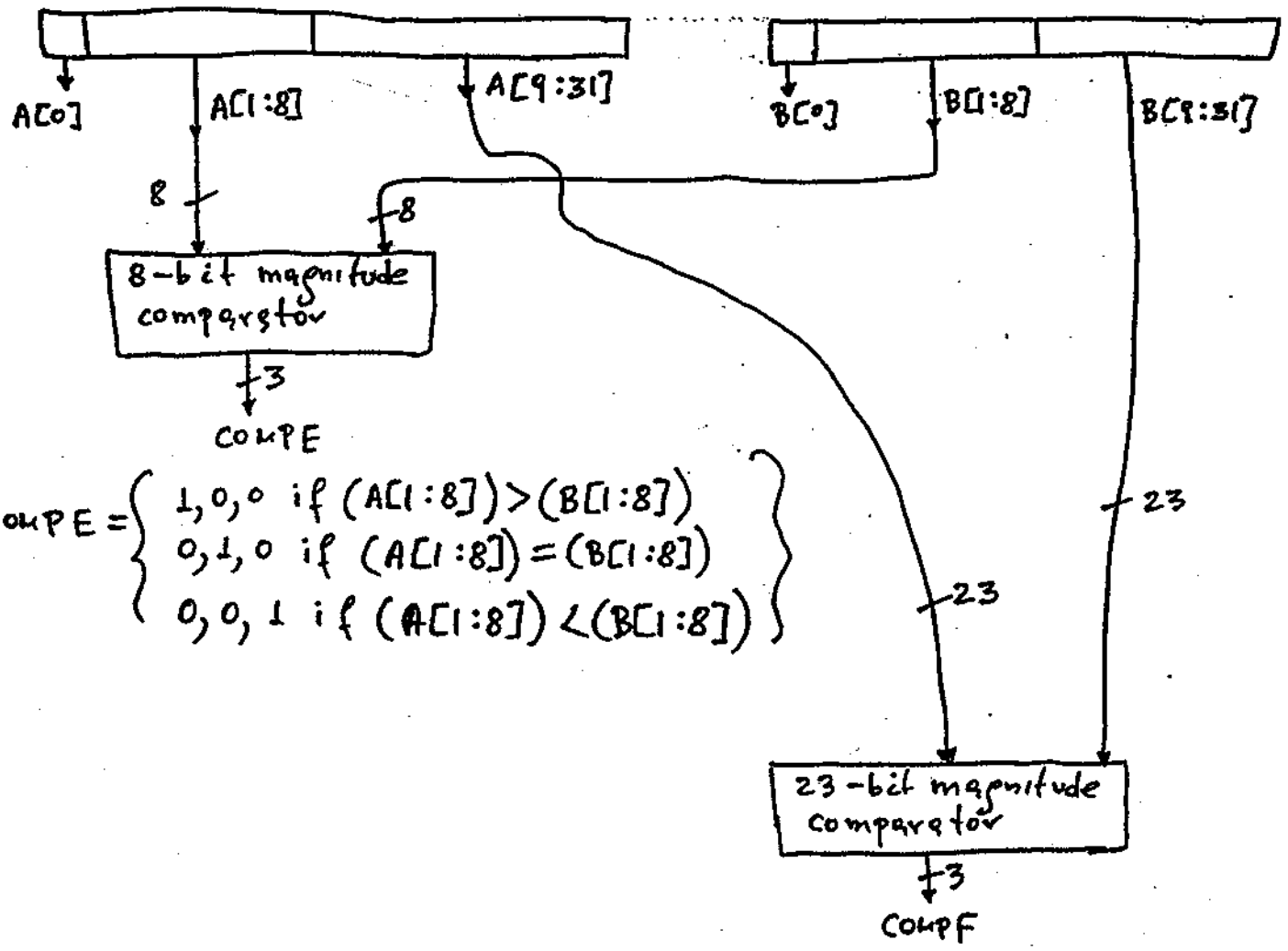
In the above description, A is the dividend register, B is the divisor register, c is the carry-out flip flop, CNT is a 4-bit counting register (used in counting 16 division cycles) while ADD is a 16-bit adder (the line ADD[0] is the carry-out line). The step 1 is responsible for initializing the engine. Also, step 1 is responsible for comparing X[0:15] with Y to determine if a division overflow is to occur. If (X[0:15]) ≥ (Y) then the controller goes to step 3 to signal the occurance of a division overflow. Else, the controller goes to step 2 and the 16 division-cycles take place. Step 4 is responsible for restoring the remainder (if needed) and for providing the quotient and the remainder to the output.

One possible solution is the following.
Let our floating point adder/subtractor consist of the following components:

- Two 32-bit registers A and B for storing the two input operands that enter through input ports X and Y and a 32-bit register C for storing the sum or difference before it becomes available through output port Z.
- One flip flop named "q" for storing the information provided by the input line "a" (the input line that indicates if an addition or a subtraction is to be performed).
- A combinational 8-bit magnitude comparator with its 3-bit output vector named COMPE used for comparing the two exponents.
- A combinational 23-bit magnitude comparator with its 3-bit output vector named COMPF used for comparing the two fractions. Such comparison is needed so that in the case of subtraction, the smaller fraction gets subtracted from the larger one.
- A combinational 23-bit adder with its 24-bit output vector named ADD used for adding/subtracting the fractions.
- A flip flop named "d" for storing the carry out of the addition of the fractions.
- An 8-bit incrementer INC and an 8-bit decrementer DEC for incrementing or decrementing exponents.

- A flip flop named "ovf" to be set to 1 if an exponent overflow has occured and a flip flop named "undfl" to be set to 1 if an exponent underflow has occured.

The following figure explains the meaning of the outputs of the two magnitude comparators.



$$COMPE = \begin{cases} 1,0,0 \text{ if } (A[1:8]) > (B[1:8]) \\ 0,1,0 \text{ if } (A[1:8]) = (B[1:8]) \\ 0,0,1 \text{ if } (A[1:8]) < (B[1:8]) \end{cases}$$

$$COMPF = \begin{cases} 1,0,0 \text{ if } (A[9:31]) > (B[9:31]) \\ 0,1,0 \text{ if } (A[9:31]) = (B[9:31]) \\ 0,0,1 \text{ if } (A[9:31]) < (B[9:31]) \end{cases}$$

MODULE : FLPADDSUB
INPUTS : X[32]; Y[32]; a
OUTPUTS : Z[32]; r; s.
MEMORY : A[32]; B[32]; C[32]; q; d; ovf; undfl.
CLUNITS : ADD[24] <: ADDER{23}; INC[8] <: INCREMENTER{8};
DEC[8] <: DECREMENTER{8}; COMPE[3] <: COMPARATOR{8};
COMPF[3] <: COMPARATOR{23}.

1 A ← X; B ← Y; q ← a. "Load Engine"

2 (A[1:8] ! B[1:8]) * (COMPE[2](arg1), COMPE[0](arg1)) ← INC(arg2)
A[9:31] * COMPE[2](arg1) ← 0, A[9:30];
B[9:31] * COMPE[0](arg1) ← 0, B[9:30];
→ (COMPE[1](arg1), COMPE[1](arg1) ∧ $\overline{q}$, COMPE[1](arg1) ∧ q) / (2, 3, 5)
"Increment corresponding exponent; right shift corresponding fraction; if exponents not equal go back to step 2; if exponents are equal and operation is addition go to 3; if exponents are equal and operation is subtraction go to 5."

3 C[1:8] ← A[1:8]; C[0] ← 0; "Store common exp. in C[1:8]. Store sign bit of sum in C[0]. The sign bit will be zero since initial operands were positive"
d, C[9:31] ← ADD(A[9:31]; B[9:31]; 0); "Add two fractions and put result in d, C[9:31]."
ovf ← ADD[0](A[9:31]; B[9:31]; 0) ∧ (∧/A[1:8]). "Exp. ovf will occur if common exp = 11111111 and addition of fractions produced a carry out!"

4 (d, C[9:31]) * d ← 0, d, C[9:30];
C[1:8] * d ← INC(C[1:8]); → (7)
"Postnormalize if necessary and go to step 7."

5 C[1:8] ← A[1:8] ; "store common exp. in C[1:8]."

C[9:31] ← ADD[1:23]((A[9:31] ! B[9:31]) * (COMPF[2](qvg3), COMPF[2](qvg3)) ;
(B[9:31] ! A[9:31]) * (COMPF[2](qvg3), COMPF[2](qvg3)) ; 1) ;

"The adder performs (A[9:31]) − (B[9:31]) if (A[9:31]) ≥ (B[9:31]) a fact dictated by COMPF[2](qvg3) being zero. The adder performs (B[9:31] − A[9:31]) if (A[9:31]) < (B[9:31]). The carry out of such addition is ignored (ADD[0] is ignored)".

C[0] ← COMPF[2](qvg3). " The sign bit of the difference will be COMPF[2](qvg3) and gets stored in C[0]."

6 undfl ← $\overline{C[9]} \wedge (\overline{v/C[1:8]})$; " Exp. underflow is expected if (C[9])=0 and the exponent (C[1:8]) = 00000000."

C[9:31] * $\overline{C[9]}$ ← ((C[10:31], 0) ! 23 T 0) * (v/C[1:8], $\overline{v/C[1:8]}$) ;

"C[9:31] will be updated only if (C[9])=0. In this case if exponent underflow is not expected C[9:31] will be shifted left by one bit. Else if (C[9])=0 and exponent underflow is expected, the field C[9:31] ~~with~~ will be cleared (recall that in the case of exp undflow we force the result to be zero)"

C[1:8] * $\overline{C[9]}$ ← (DEC(C[1:8]) ! 8 T 0) * (v/C[1:8], $\overline{v/C[1:8]}$) ;

"Similarly, the exponent part will be updated only if (C[9])=0. In this case the update will be the decremented by 1 value if no exp. underflow is expected while in case that exp. undfl. expected the field C[1:8] is cleared".

C[0] * ($\overline{C[9]} \wedge (\overline{v/C[1:8]})$) ← 0 ; " Clear sign flip flop C[0] if exp. undflow is expected

~~[illegible]~~ → ($\overline{C[9]} \wedge (v/C[1:8])$/(6). ~~[illegible]~~

] r, s, Z = ovf, undfl, C ; "output result together with the status of the overflow flags and go back to step 1". '
→ (1).

END SEQUENCE

END

In the above AHPL description of the floating point adder/subtractor I included explanatory comments written within the various steps.

Also, the above solution is not the only one neither is it the most optimal.

The input arguments of the exponent comparator COMPE are

arg1 = A[1:8]; B[1:8]

The input argument of the incrementer INC in step 2 is

arg2 = (A[1:8] ! B[1:8]) * (COMPE[2](arg1), $\overline{\text{COMPE[2](arg1)}}$).

The input arguments of the fraction comparator COMPF are

arg3 = A[9:31]; B[9:31].