1. Instruction set

The 5'th September 1992 and I begin by starting to think about the instruction set. I decide to have 16 registers. This way the register number will fit in 4 bits. The computer will be a 40-bit machine. Hopefully I will be able to fit 2 instructions in each 40-bit "word", each one taking only 20 bits. This way I can execute 2 instructions per instruction fetch. On this page the right column indicates the number of bits the instruction operands require.

The instruction groups are:

o NOP (No operation)
o Arithmetic (Multiply, Add, Subtract), integer and floating point
o Logical
o Shifts and rotates
o Loads
o Individual bit testing, setting and resetting
o Jumps, Calls, Branch (relative jump) and returns
o I/O instructions and control

5/9/92 1

NOP	NOP		0	0
Mult	Integer		S1, S2, R	12
	F.P	*	S1, S2, R	12
Add	Int NC		S1, S2, R	12
	Int C		S1, S2, R	12
	F.P	*	S1, S2, R	12
Sub	Int NC		S1, S2, R	12
	Int C		S1, S2, R	12
	F.P.	*	S1, S2, R	12
Logical	AND		S1, S2, R	12
	OR		S1, S2, R	12
	NOT		S, R	8
	XOR		S1, S2, R	12
	INC		S, R	8
	DEC		S, R	8
Shifts	Shift n bits Rotate left		S, R	8 + n
	Rotate Right		S, R	8 + n.
	Shift logical Right		S, R	8
	Shift Arithmetic ~~left~~ right		S, R	8
	Shift left.		S, R	8
Loads:	Mem load reg, mem		r, mem	≥ 22
	Store reg, mem		r, mem	≥ 22
	Move reg, reg		r, r	8
	Push reg		r	4
	Pop reg		r	4
Bit:	Test		r, bit	9
	set		r, bit	9
EXTRAS ABS value	reset		r, bit	9
Change sign	CLC		0	0
	SEC		0	0
	Load r, reg + disp		r, r, disp	≥16
	Store r, reg + disp		r, r, disp	≥16

PTO

2. Instruction set (continued)

The Jump, Call, Branch and Return instructions. I note that the Jump and Call instructions will not fit inside the 20 bits size I hope to use for each instruction. The destination address must be 20 bits or more, if the memory is to be a reasonable size.

(cont). 5/9/92 2

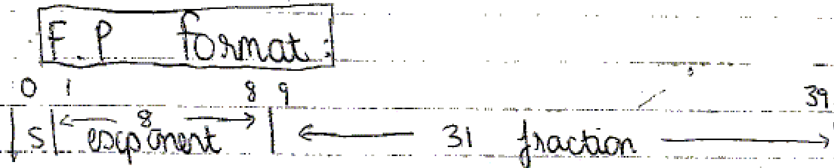
Jump /branch /sub: /Ret	Jump		mem.	≥ 18
	Jump	C	mem, condition	≥ 20
	Jump	NC	"	≥ 20
	Jump	Z	"	"
	Jump	NZ	"	"
	Jump	P	"	"
	Jump	N	"	"
	Call		mem	≥ 18
	Call	C	mem, condition	≥ 20
	Call	NC	"	"
	Call	Z	"	"
	Call	NZ	"	"
	Call	P	"	"
	Call	N	"	"
	Branch		disp	≥ 10
	Br	C	disp, condition	≥ 12
	Br	NC	"	"
	Br	Z	"	"
	Br	NZ	"	"
	Br	P	"	"
	Br	N	"	"
	Ret		0	0
	Ret	C	cond	2
	Ret	NC	"	2
	Ret	Z	"	2
	Ret	NZ	"	2
	Ret	P	"	2
	Ret	N	"	2
I/O	In	reg		
	Out	reg		
Control	Halt, & interupt host.			

3. Floating Point format and Execution Units

Next I decided the format to use for floating point (none of this IEEE compliant nonsense). I use 1 sign bit, 8 exponent bits and a 31-bit fractional part. This I consider should be a reasonable accuracy for most applications, and of course it fits the 40-bit word size nicely.

I regroup the instructions and decide on what execution units I will need to build. Each unit will handle a particular type of instructions. The list of execution units doesn't seem complete.

5/4/92 3



Instruction types:

1) NOP
2) Loads,
3) Multiply
4) Add /Sub/ logical
5) Shifts
6) Bit manipulation
7) Jumps & calls
8) I/O/ control.

New groupings:

0 : Control, incl NOP / I/O.
1 : Load/store
2 : Reg→Reg
~~2~~ 3 : Multiply
~~3~~ 4 : Add /Sub / logic
~~4~~ 5 : Shift
~~5~~ 6 : Bit manipulation
~~6~~ 7 : Jumps & calls
~~7 I/O~~

Units:

I/O,
Mult,
Add/sub/~~shift~~ logic
Shift
Load / store to mem
" to reg
Stack reg.

4. Detailed Instruction Codes

The next day I consider in detail the format the instructions will take. The first 3 bits will always code the unit required. The following bits specify the opcode and operands (register numbers for source and destination data, etc). The load and store to memory instructions will require a whole 40 bits.

6/9/92 4

Group 0: Control

0 1 2 3 4
|000| |←

00 NOP
01 HALT
~~10 In r~~
~~11 Out r~~
} Come under Mov reg, reg.

1: Load/store:

0 1 2 3 4 5 6 10
|0|0|1|000| r |← mem → } Load reg, mem
|001| r |← mem → Store reg, mem. } Whole 40 bits.
|010| r | x Pop r
|011| r | x Push r
|10| r | r |13 14 15 16 17 18 19 20| Indexed load
|11| r | |13 ← 8/7 disp →| Indexed store

NO, put r in 6-10

2: Reg → reg.

|0|1|1|0| source |7 dest 10| Move reg → reg.

3: Multiply

|0 1 1 |3 |4 source 1 |7 source 2 |11 dest 14|
0 F.P
1 Int.

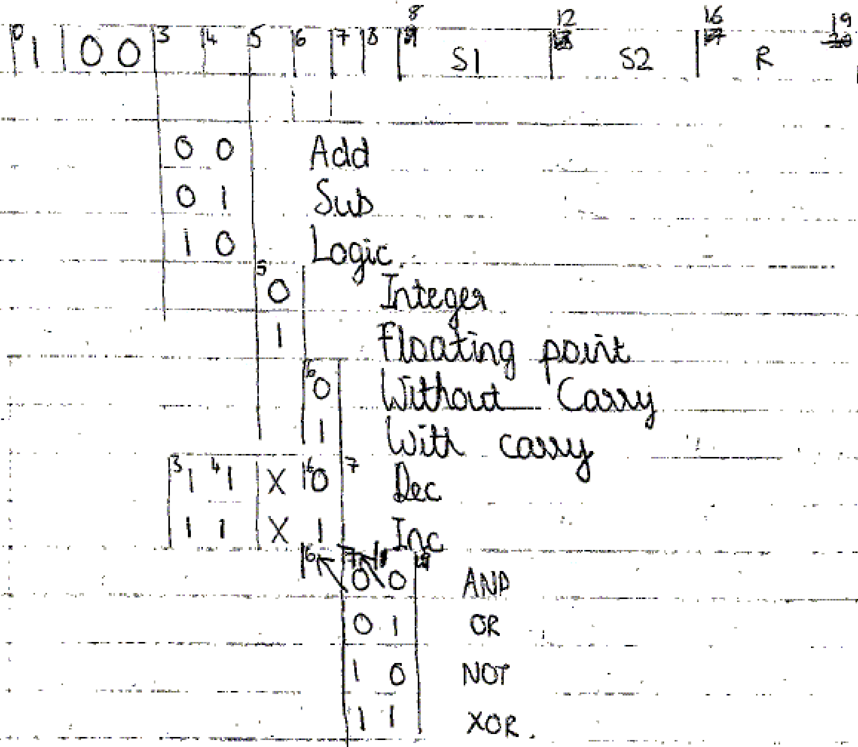
5. Detailed Instruction Codes: Add/Sub/Logic, Shift

The opcode for the add/sub/logic instructions fits nicely into 5 bits, so there are 12 left for the two source registers and the destination register. Neat (now you may begin to realise why I chose 40 bits for my word size).

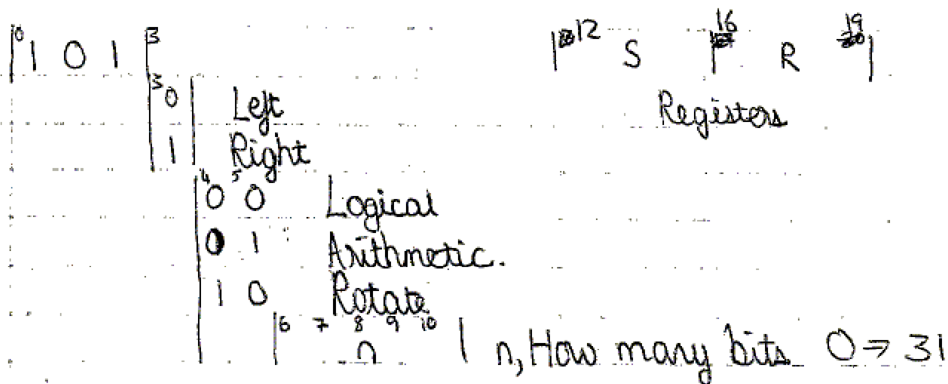
The shift instructions also have a source and destination register. I have marked 5 bits to specify the amount of shift but I think I would really need 6 (for 0-39 bits of shift).

6/9/92. 5

4: Add / Sub / Logic.



5: Shift



6. Detailed Instruction Codes: Bit manimpulation and jumps

Bit manipulations also fit exactly and neatly into 20 bits. 3 bits specify the action required, 6 bits the number of the bit in the word (0..39), and 8 bits the source and destination registers.

The jumps and calls require the full 40 bits for specification of the target address. Branches fit in 20 bits, they have 12 bits available to specify the displacement from the current location. The Jump, Branch, Call and return instructions all have conditional variants, which test the state of the Carry, Zero and Sign flags.

6/9/92 6

6: Bit manipulation:

1110 | S | R

Bits 3–5	Action
000	Test (Doesn't use R)
010	Reset
011	Set
100	CLC (Doesn't use S/R/n)
101	SEC //

Bits 6–11: n ; n; Bit no, 0 → 39.

7: Jumps etc.

0111

Type		
00	Jump	(Only in first field) 2nd field = address
01	Branch	(2nd field ignored)
10	Call	(2nd field ignored) 2nd field = address
11	Ret.	(2nd field ignored)

Condition	
000	No condition
001	Loop, dec reg ? until = 0.
010	NC
011	C
100	NZ
101	Z
110	P
111	N

Bits 8–19: Disp

ZR, CR, I%, ZI, CI, A, B, DX, DY, X, Y, AR, AI.

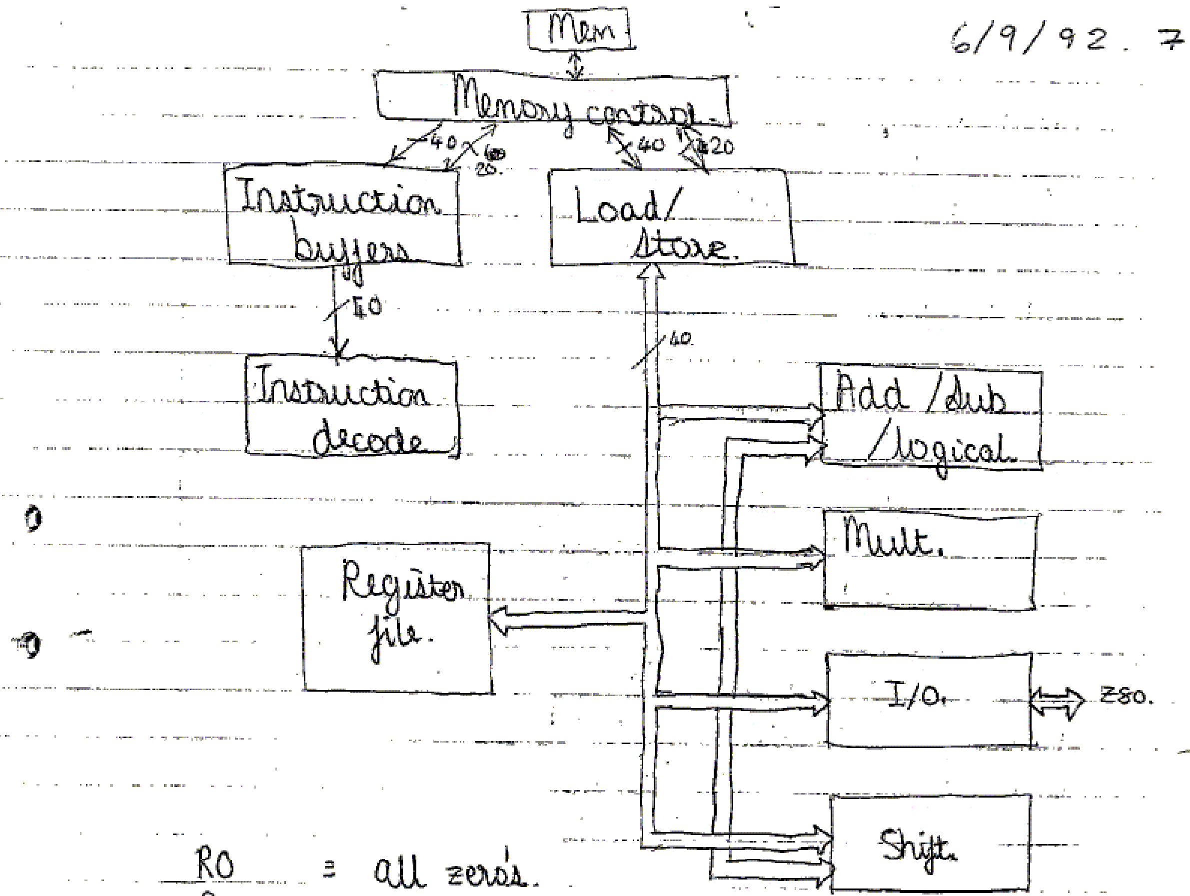
12 4096.

* Need 18 bits to specify full 256K addresses.

7. Block Diagram and Register descriptions

A small block diagram showing the interconnection if the units in the CPU. The CPU incldes its own memory and memory control, and all input/output is intended to be via a host Z80 processor.

There are 9 general purpose 40-bit registers, register 0 is always zero, and the remaining 6 registers are for the stack pointer, program counter, loop counter, In and Out registers, and index register.



R0 = all zero's.
R1 = A
R2 = B
R3 = C
R4 = D
R5 = E
R6 = F
R7 = G
R8 = H
R9 = I
R10 = Index Register (Write only, 20 bits).
R11 = IN register 40 bit Read only.
R12 = OUT register. 40 bit Write only.
R13 = LOOP COUNTER (20 BIT) Write only.
R14 = STACK POINTER (20 BITS) Write only.
R15. = PC (20 BITS). Write only.

20 Bits can address 1 M addresses, = 5 M Bytes total.

8. Scoreboarding and Bus Arbitration

Now I give some consideration to the scheduling and organisation of the units. The Scoreboard ensures that registers and execution units are locked while they are awaiting a result write or in use.

On this page I also make a few calculations relating to the mandelbrot set, concerning the number of instructions that must be executed in each iteration of the mandelbrot calculation. Yes, drawing mandelbrot sets is considered to be the first application for the computer...

6/9/92. 8

Source 1: 8 – 11 or 6–9
Source 2: 12 – 15
Result : 16 – 19 or 6–9

Scoreboarding and busy unit:

On decoding of instruction, checks must be made to ensure
i) function unit is available
ii) Source registers available
iii) Destination registers available.

If not:
Label function unit "busy"
Indicate destination unit "busy".
Load sources into unit,
Issue start signal to unit.

Bus arbitration:

1st come first serve.
Some sort of priority to results from units:

ZI = CI
ZR = CR
I = 0

LOOP STARTS →

A = ZR×ZR	1
B = ZI×ZI	1
ZR = 2×ZR×ZI + CR	3
ZI = A − B + CI	2
C = A + B	1
Test if C>4	1
DJNZ if NOT	1
	10

ZR ZI
A B
CR CI
C
Inter Inter

Have ~1µs = 25 instr cycles

9. Instruction set codes again

A review of the instruction set, in which things change slightly.

7/9/92 9

0: Control:

|0 0 0 0|3
- 0 NOP
- 1 HALT & interupt Z80.

1: Load/store

|0 0 0 1|3 ● |5 |6 reg 9|10 ← disp → 19|20 ← address → 39|
(Indexed) (absolute)

- 00 To a specified address.
- 01 Push/Pop : save address & dec SP
- 10 Indexed Load/store using IX.
  - 5 0 Load
  - 1 Store.

2: Reg → Reg.

|0 0 1 0|3 × | × |12 S 15|16 R 19 ●|

3: Multiply

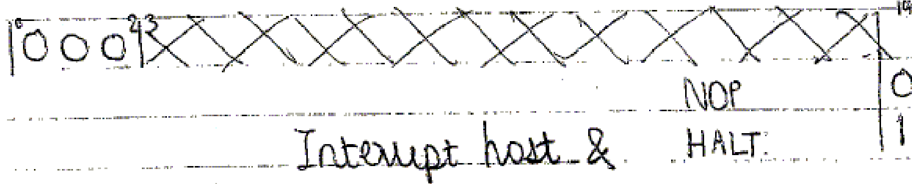
|0 0 1 1|3 ● | |8 Si 11|12 S2 15|16 R 19|

- 0 4 F.P
- 1 INT

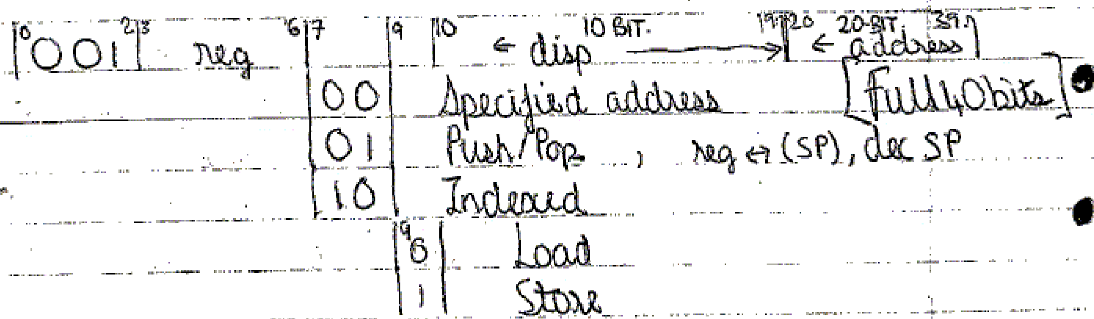
10. Instruction set codes again (Control, Loa/Store, Multiply, Add/Sub/Logic)

8/9/92 10

0: Control

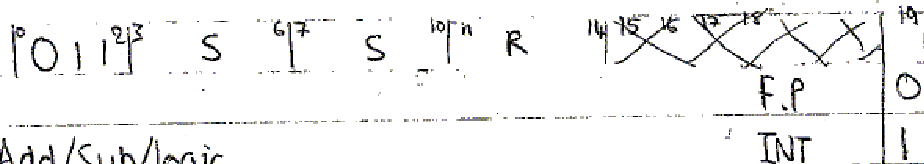


1: Load/Store

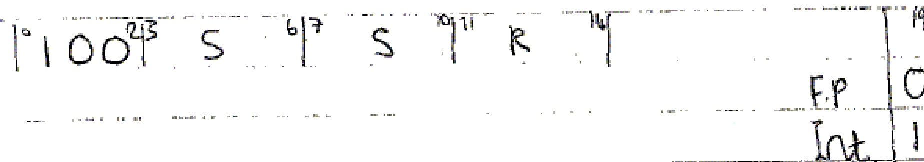


2: Reg → Reg

3: Multiply



4: Add/Sub/logic



	18
Dec OR Without Carry	0
/Inc With Carry	1

	15 16	17 18	
Add	00	00	AND
Sub	01	01	OR
Logic	10	10	NOT
Inc/Dec.	11	11	XOR

11. Instruction set codes again (shifts, bit manipulation, jumps)

8/9/92. 11

5: Shifts

0–2	3–6	7–10	11–14	15–19 (5 bits)
1 0 1	S	(7–8, 9, 10 ⊠)	R	n

7 8	
0 0	Logical
0 1	Arithmetic
1 0	Rotate

9	
0	Left
1	Right

n = No. bits.

6: Bit manipulation

0–2	3–6	7–10	11–14	15–16	17–19
1 1 0	S	$n_5 n_4 n_3 n_2$	R	$n_1 n_0$	

n = Bit no. (0-39)

	17–19
Test	0 0 0
Reset	0 1 0
Set	0 1 1
CLC	1 0 0
SEC.	1 0 1

7: Jumps etc.

0–2	3–4	5–7	8–19	20–39
1 1 1			12-bit disp	address

3 4		
0 0	Jump	→ [Full 40 bits]
0 1	Branch	} 2nd instr. field ignored.
1 0	Call	
1 1	Ret	

5 6 7	
0 0 0	No condition
0 0 1	Dec reg R13 & loop while > 0
0 1 0	NC
0 1 1	C
1 0 0	NZ
1 0 1	Z
1 1 0	P
1 1 1	N

12. More on Scoreboarding

Some more thoughts on the practical implementation of the scoreboarding. Also, consideration of an alternative way to code the instructions.

8/9/92. 12

Scoreboarding:

Register access → 4 → 74LS 154 → 16

"Busy" flip flops — 16 → 16-1 line MUX → "Waits" if selected reg is busy.

"R" field of instruction

- Have to check on source & destination registers, & function units.

Alternative instruction format...

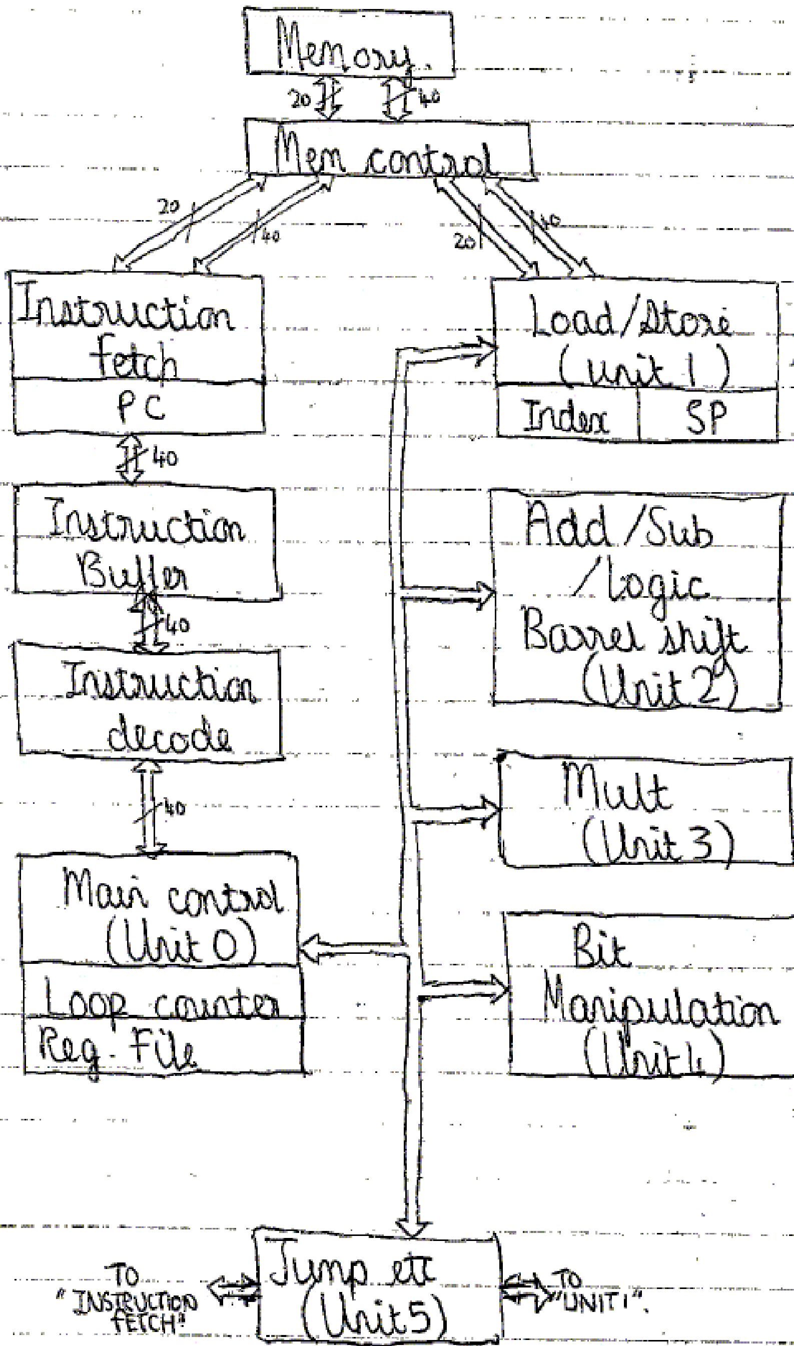
0 3 | 4 3 ... 8 — OPERATION UNIT, OPERAND NO./TYPE

0 ... 2 3 ... 8 ... 11 12 ... 15 16 ... 19 — UNIT

Types: No operands;
Load/store type
S, R
S, S, R
S, R, n
Displ. / abs (Jumps etc)

13. Block Diagram of CPU

A nice block diagram of the CPU, as it stands so far.



14. DRAM refreshing

The 9th September 1992 and I decide to think about refreshing of dynamic memories (DRAM), and begin by calculating how often I will need to do it, and how long it will take.

9/9/92 14

Refresh: Every 4ms,
Refresh 1-3% of time

120 + 100
250 say
4 every μs
∴ takes 128μS for all 512 addresses;
4000 / 128   128 / 40 ~ 3 %.

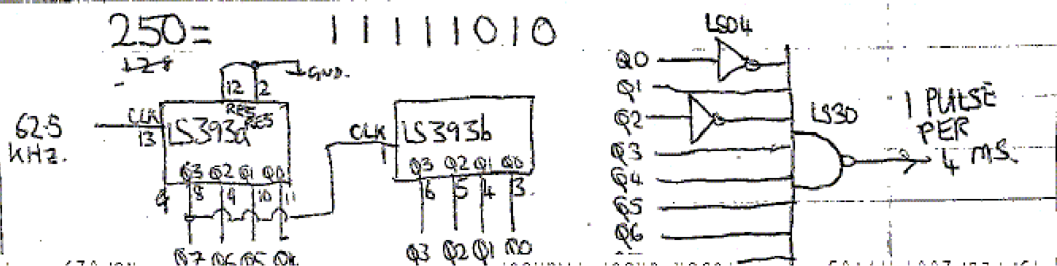
This is for 120nS chip:
70nS one would be 1.87%.
80nS ⇒ 2.13 %.

Refresh control:

4 mS = 4000 μS = 4 000 00. 100 MHz CLKS.
4 000 000
400 000   65536 × 2 = 131072 × 2 = 262144
262 144

Using 4 MHz clock:
4 mS = 4000 μS = 16 000 4 MHz CLKS.
1 MHz = 1μS
500 KHz = 2μS
250 KHz = 4μS
125 KHz = 8μS
62.5 KHz = 16μS (250).
16) 4000 = 250

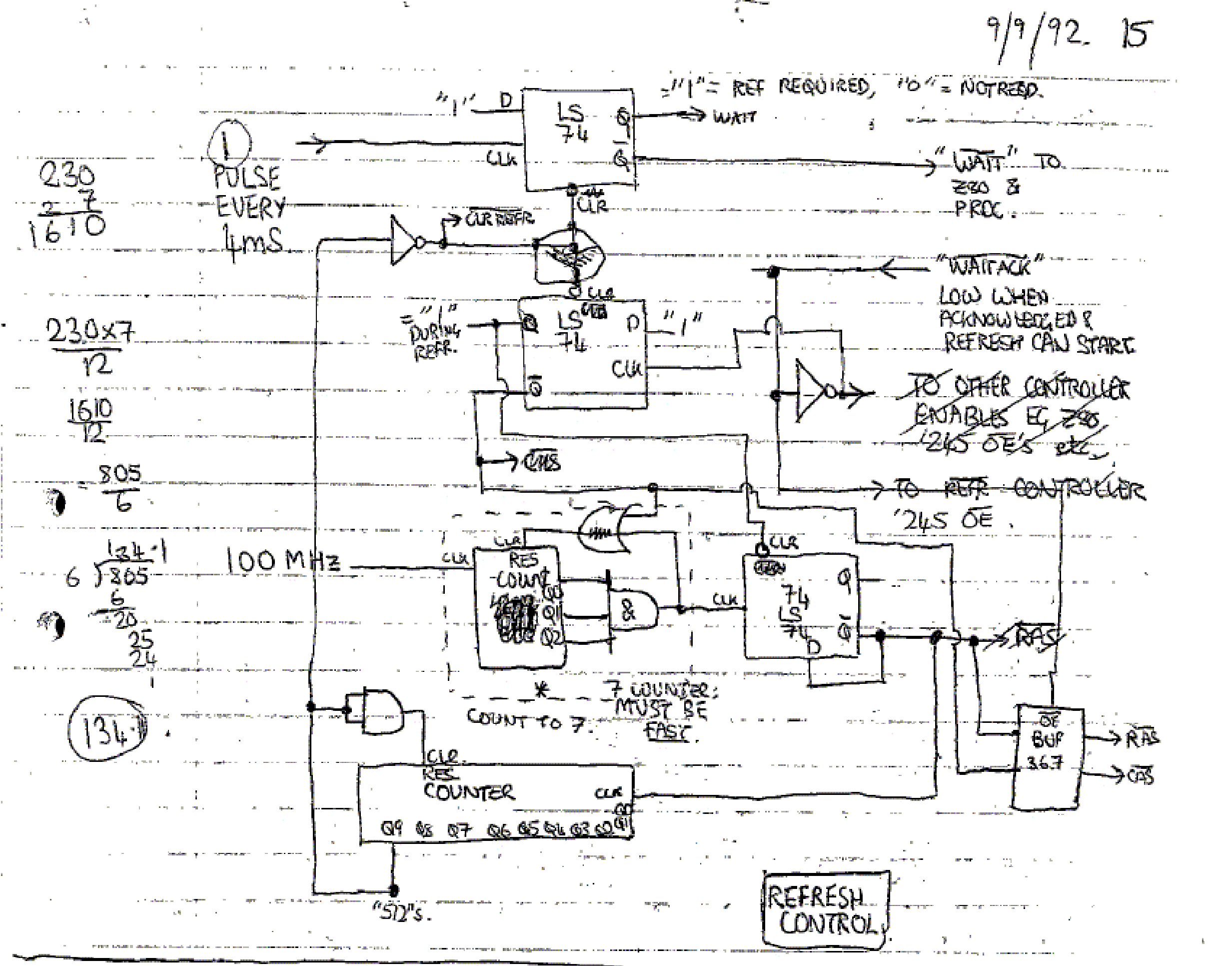
1 PULSE per 4 MS.

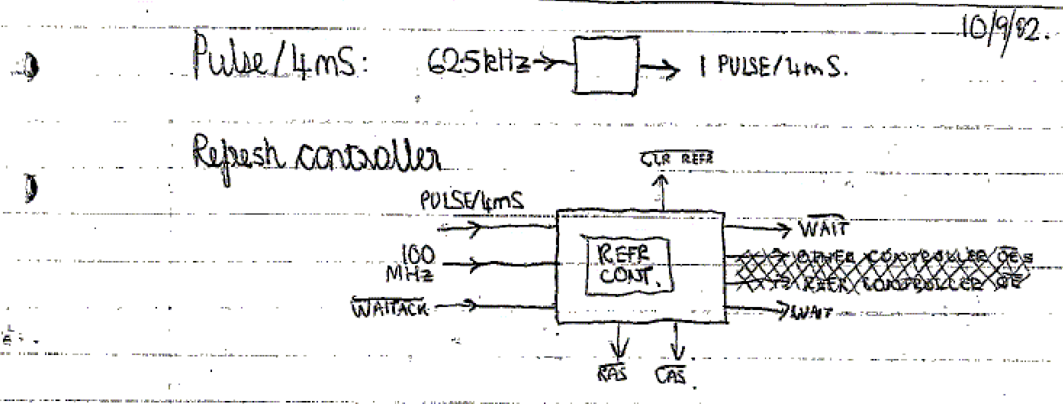


15. DRAM refresh circuit

A real circuit diagram of how I will take care of the refreshing. At the bottom of page 14 is a circuit to generate one pulse every 4 mS, which is how often I will be doing a refresh. This is generated from a 62.5 kHz signal, which would in fact come from the host Z80 computer's video controller circuit. The refresh controller also requires a 100 MHz clock.

During the refresh, the CPU gets suspended, and the hosting Z80 also has to WAIT. The refresh operation doesn't start until the rest of the CPU acknowledges the refresh request.





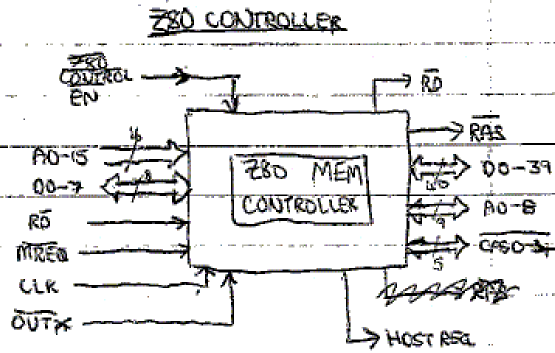
* COMBINED PROP. DELAY HERE MUST BE <10nS IN THE COUNT, &, and OR.

16. Z80 Memory Controller

This CPU design only has I/O via the host Z80 computer, and so the contents of the memory are programmed by the Z80. The circuit on this page allows the host Z80 to read and write the memory. At the bottom write I draw a nice diagram indicating the external connections of this unit.

10/9/92. 16

Z80 controller



How to 3-state like the various CAS, RAS etc signals to the memory?

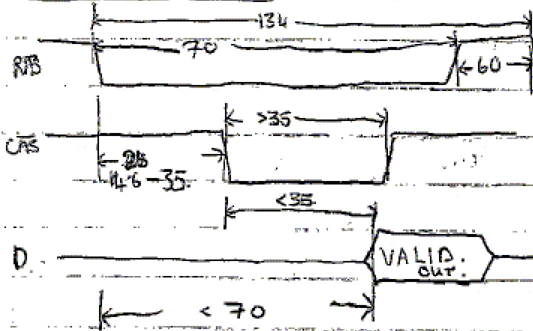
17. DRAM page mode

I devote some attention to the issue of dynamic RAM page mode access. Page mode is much quicker than a fully random access. In page mode, the row address of the memory location is locked, and columns read from that same row.

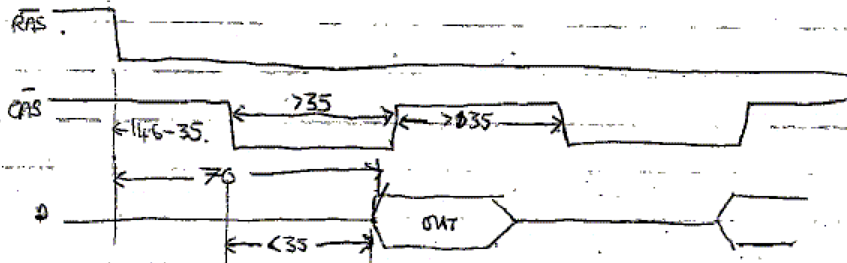
I feel that if I use page mode wherever possible, my CPU will run twice as fast as it would otherwise, because I can get my instructions twice as quickly.

10/9/92 . 17

NUM controller

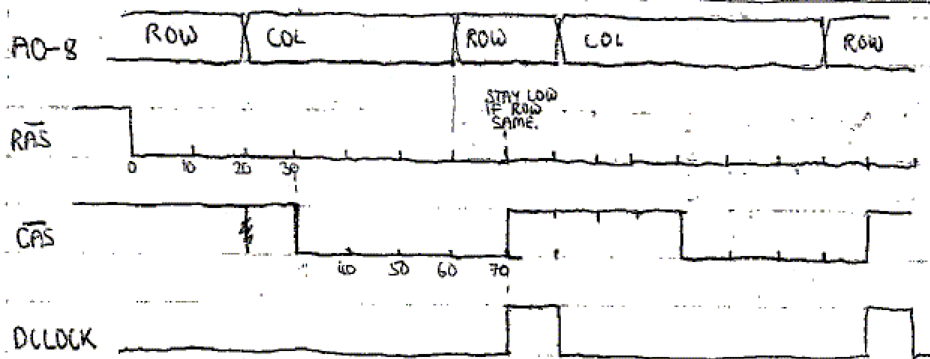


Page mode:



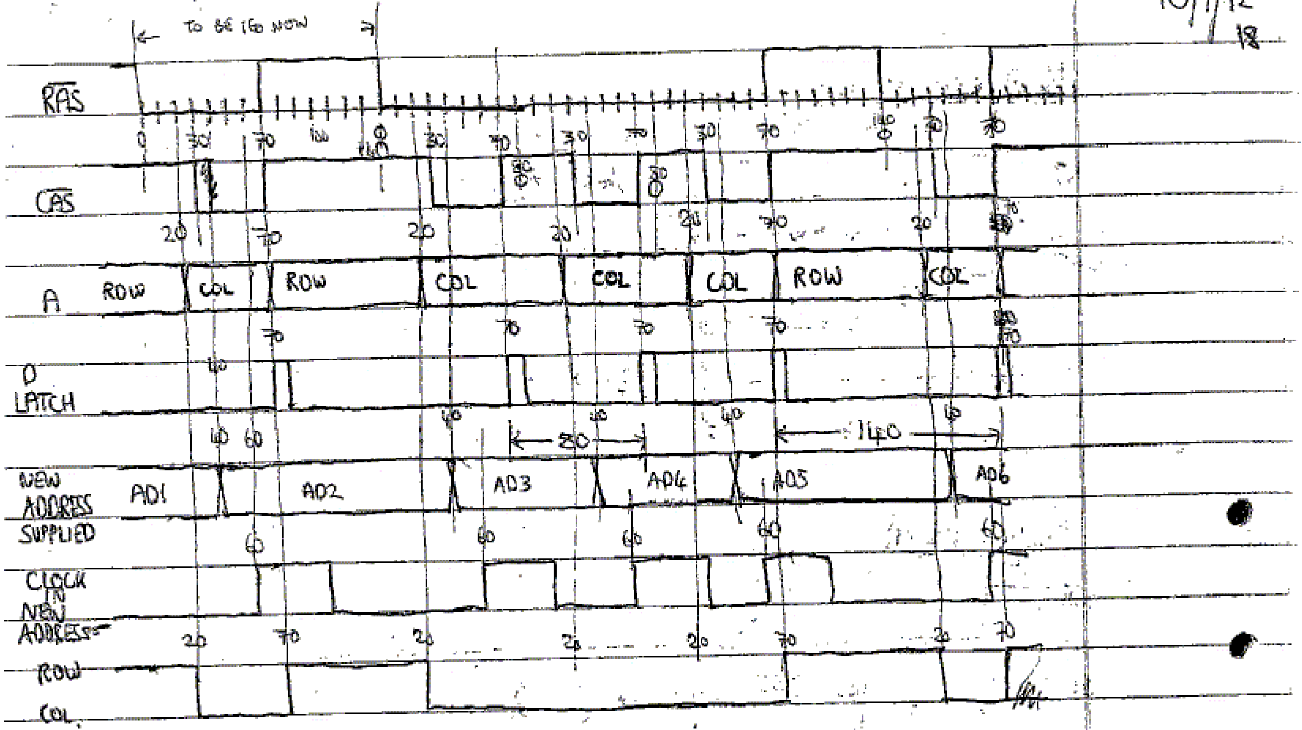
RAS address hold: >15ns for -120ns 4464 (4464-120).

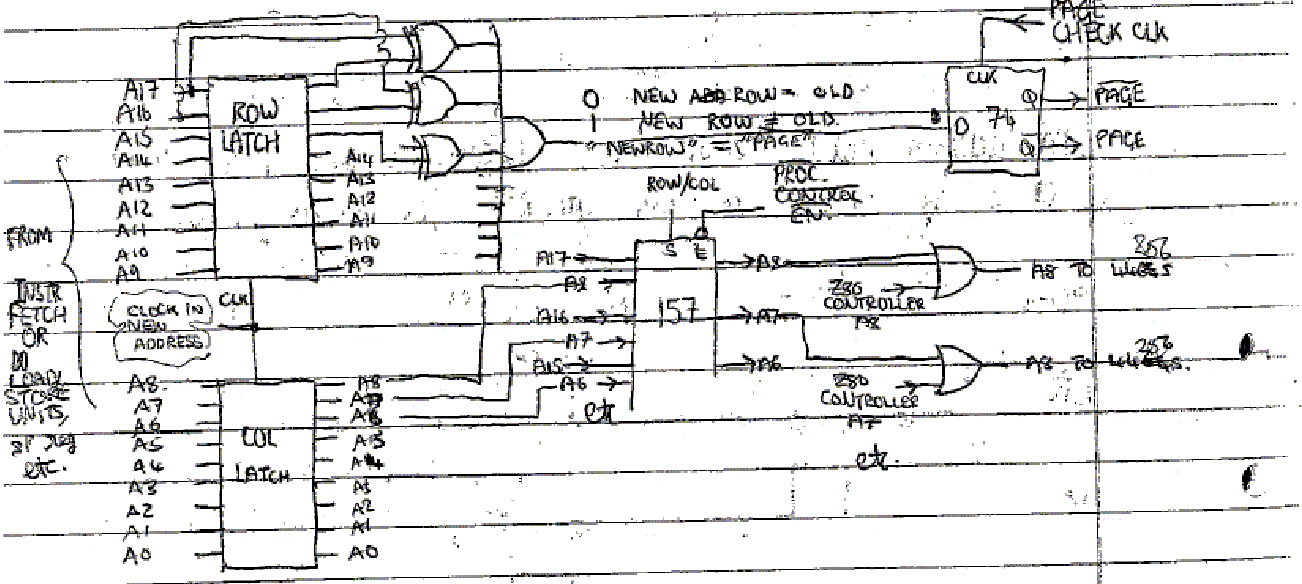
Col address hold: 46-67ns after RAS for 70ns 4464.

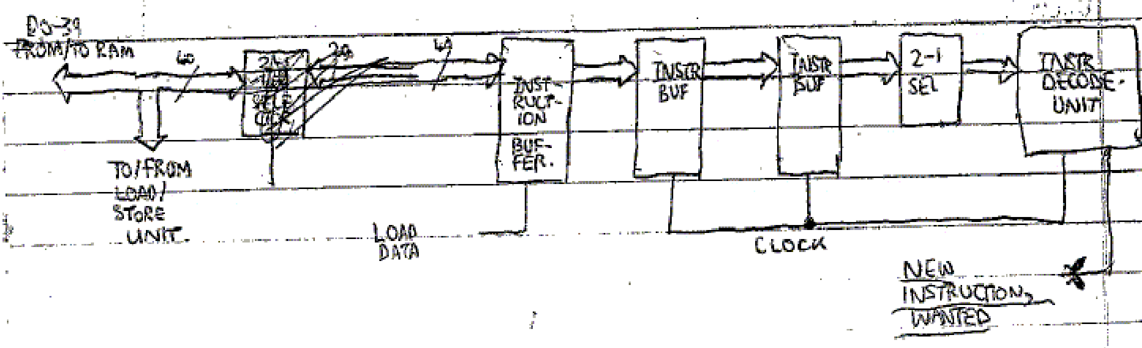


18. Instruction fetch and pipeline

More details of the precise memory timing I will need to generate in order to arrange for page mode. Here are some designs for circuits to arrange this timing, and ideas for the instruction buffer (pipeline).



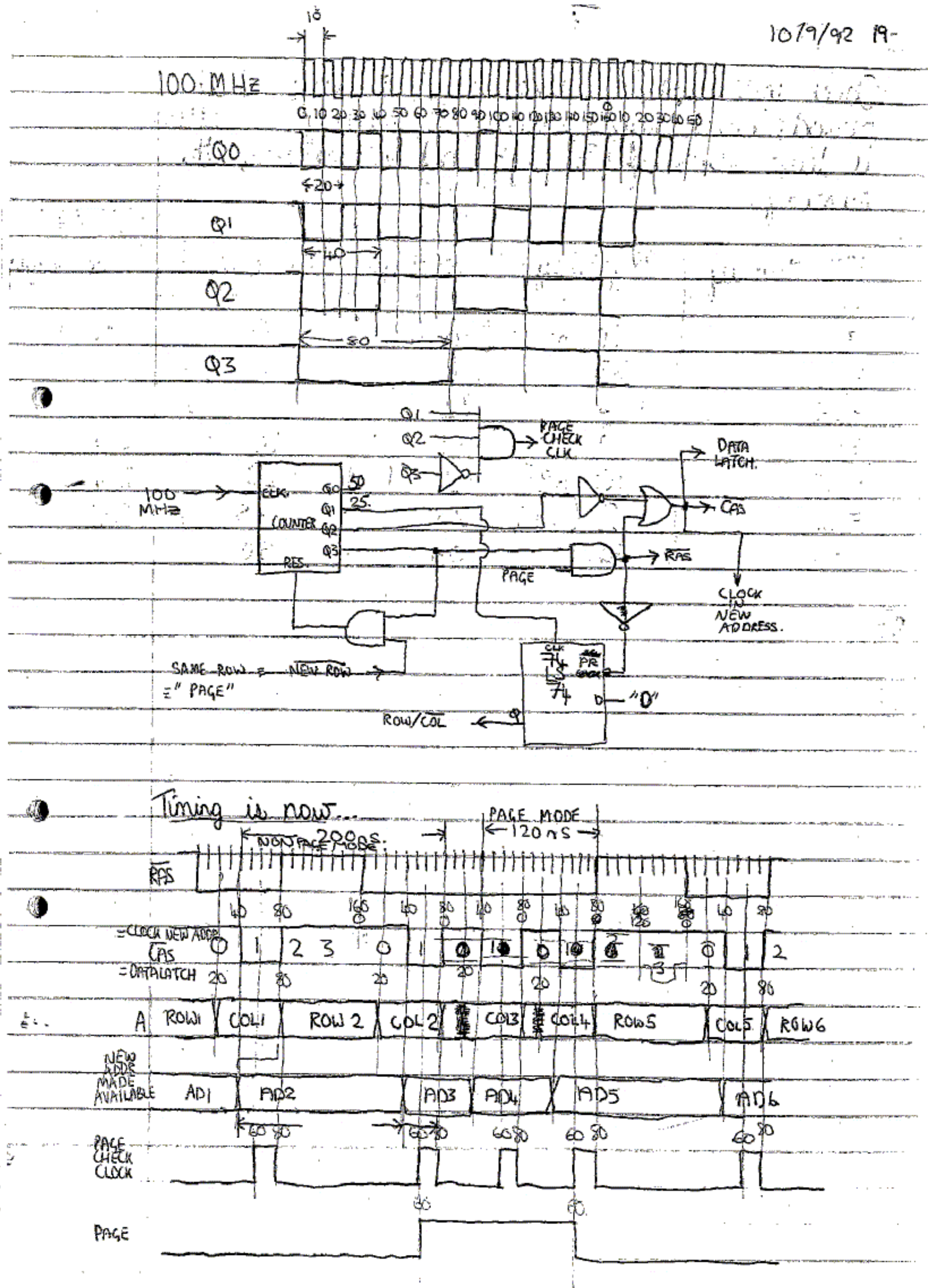




19. DRAM timing for Page-mode access

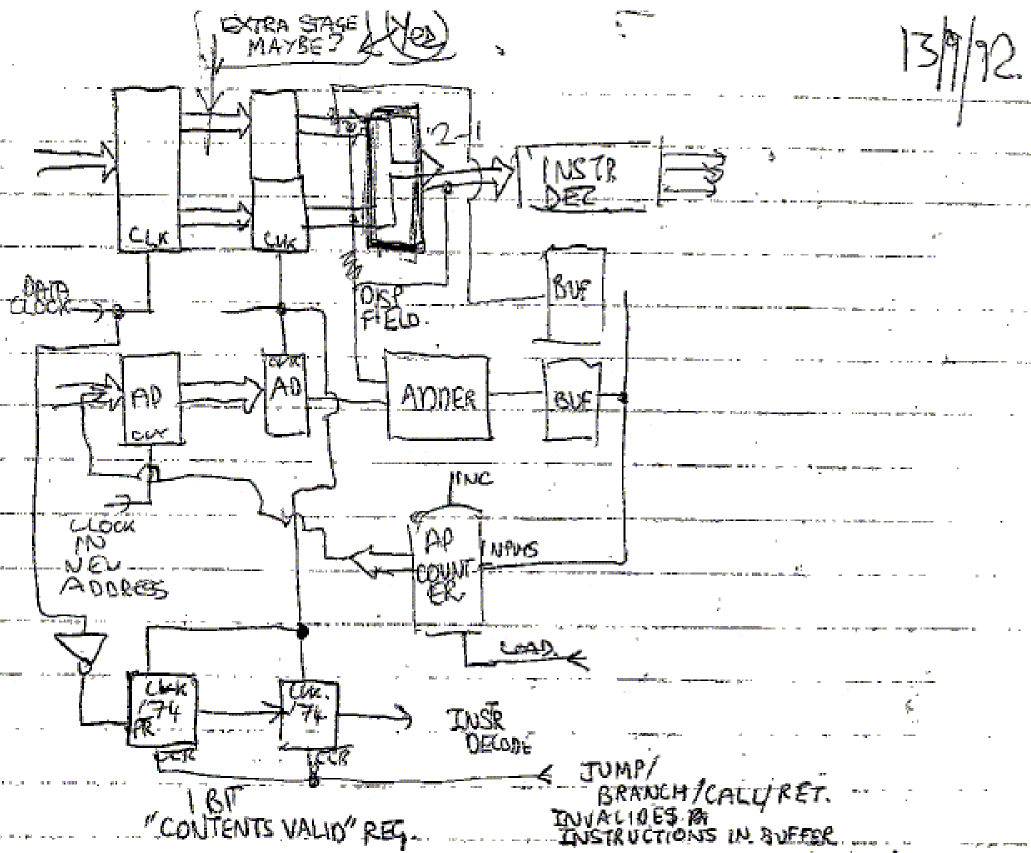
More on the timing of the DRAM page-mode access. Note that the indicated "100 MHz" is for thought purposes only: the CPU is asynchronous so in reality no such clock exists. In practice I will ensure that propagation delays in the instruction fetch circuit will be long enough so that the memory will always have the required amount of time to access and return its data.

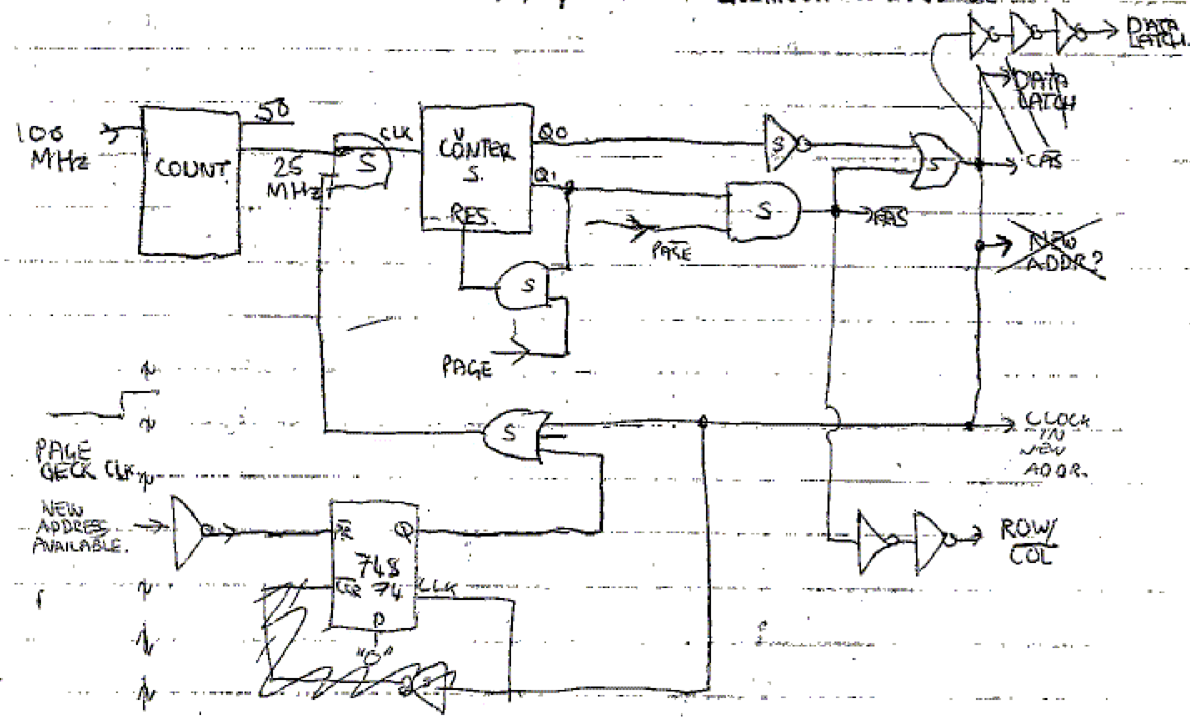
[Or is this true? Did I in fact intend to use a 100 MHz clock for the memory timing, in order to ensure precise timing? In this case I would have considered the resultant 10 nS period very short so that effectively the clock was only be used to ensure precise timing, not synchronise any other part of the processor].



20. Instruction buffer

Early sketches concerning the operation of the instruction buffer and memory timing.





21. Propagation Delays of TTL families

13th September 1992 and I carry out some library research into the propagation delays of various TTL types.

11/9/92 - 21

Check initialisation procedure, following refresh period; check refresh circuit & modify to allow at least 80nS of RAS high before releasing the memory.

13/9/92

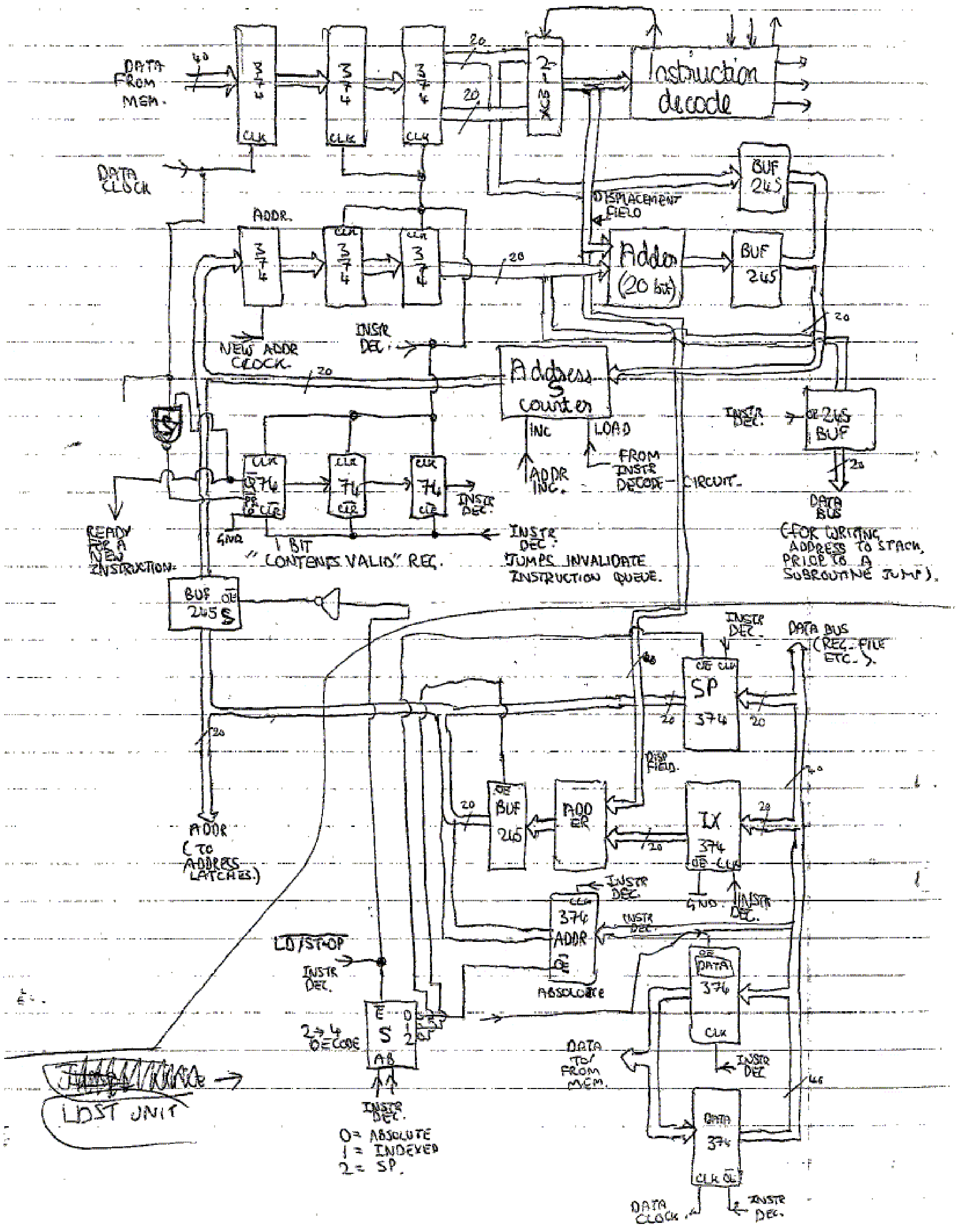
TTL family	Propdelay	Power diss	Fan out	Noise margin
74	10 nS	10 mW	10	0-4 V
74L	33	1	?	
74H	6	22	?	
74S	3	19	10	0-4
74LS	10 (or 9.5)	2	20	0.4
ECL	2 (or 1)	25	25	0.2
CMOS	~~30~~ 25	0-1	50	3

RAS

CAS

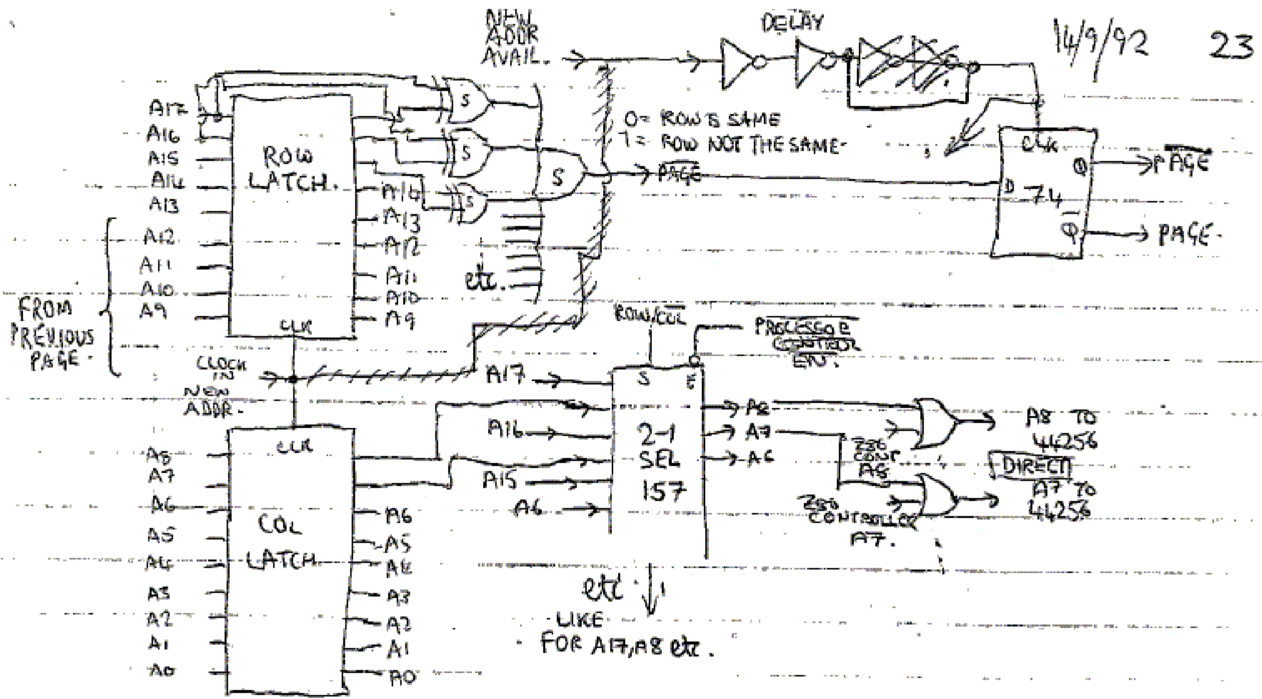
22. Instruction Pipeline and Load/Store Unit

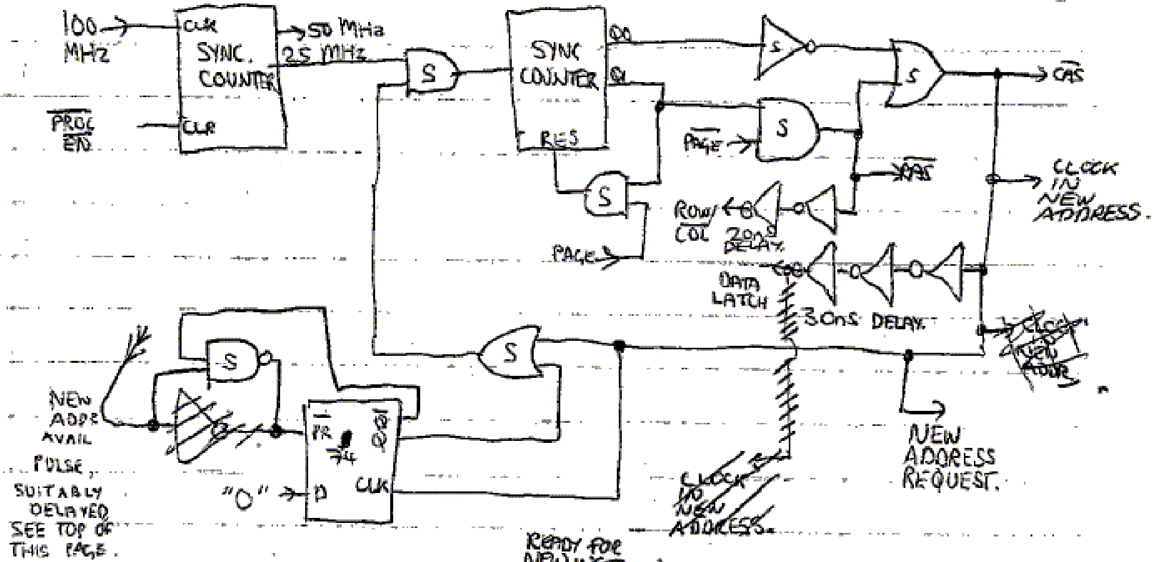
Here is a detailed diagram of the instruction pipeline and load/store units, which are closely connected.

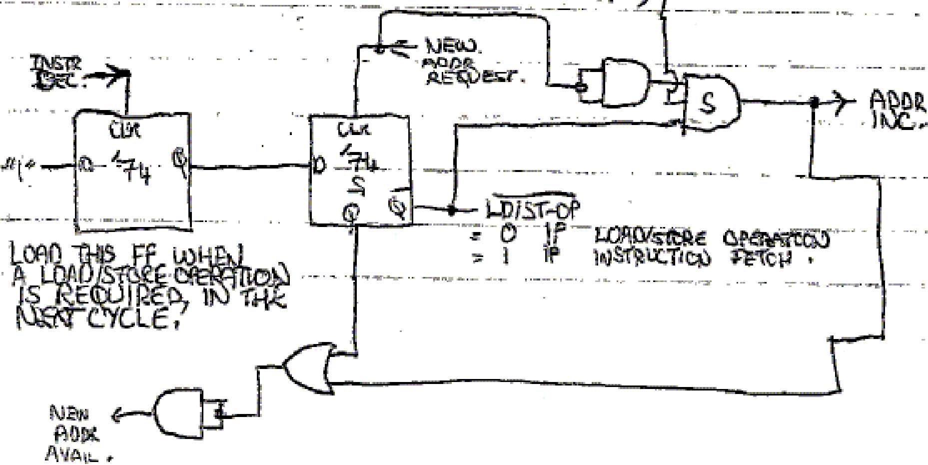


23. Instruction Pipeline (continued)

More of the instruction fetch and memory timing control circuits. The second circuit here relates to the memoryu timing, but has evolved to a more carefully thought-out version.



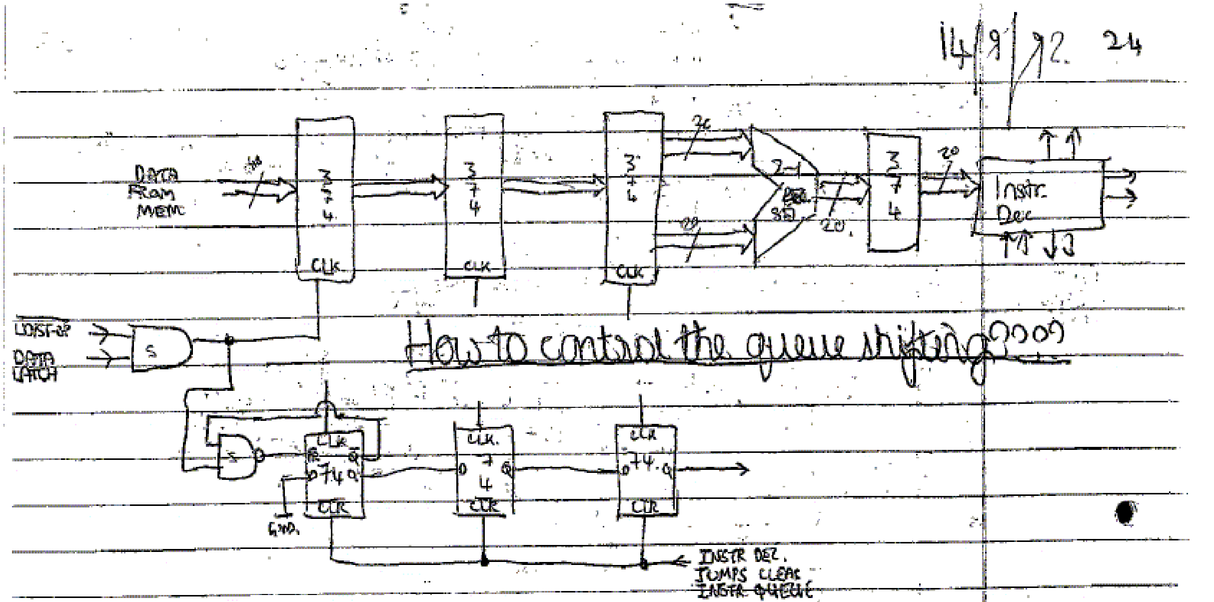




24. Shift unit thoughts

Now I spend some mental energy considering the controversial topic of shifting. I want a fast barrel shifter, that will be able to shift by any number of bits. This must be done in stages, but using what TTL chips? To try to work out the best configuration, I consider designs including cascaded stages consisting of a number of the following TTL chips:

o 74LS151: Single 8-1 line multiplexer
o 74LS153: Dual 4-1 line multiplexer
o 74LS157: Quad 2-1 line multiplexer



SHIFTER

Design using 157's has 5 levels, 32 selectors each level, = 8 chips/level, = 40 157's in total. | Prop delay is 5 gates

16
8
4
2
1

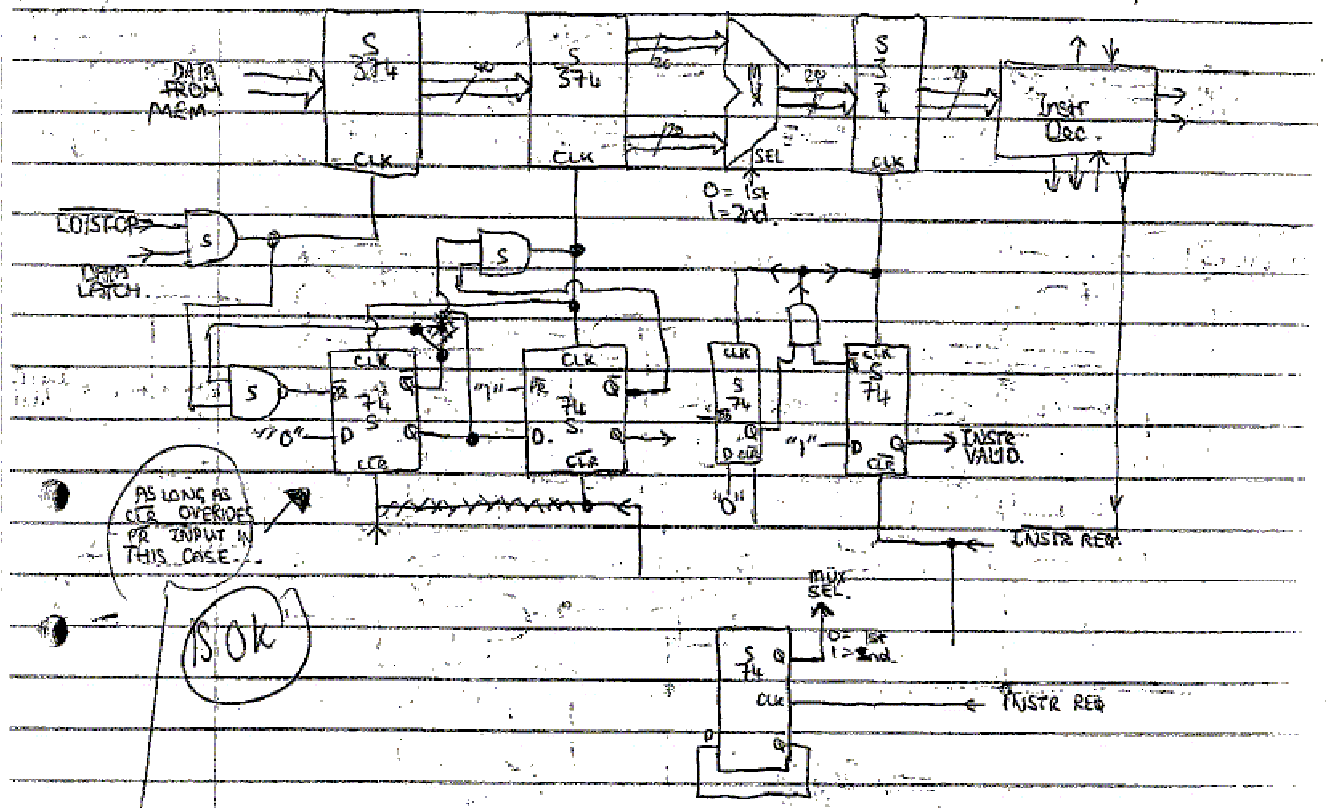
Design using 151's uses 2 levels of 32 157's gates, = 16
1 level of 32 151's
= 48 in total

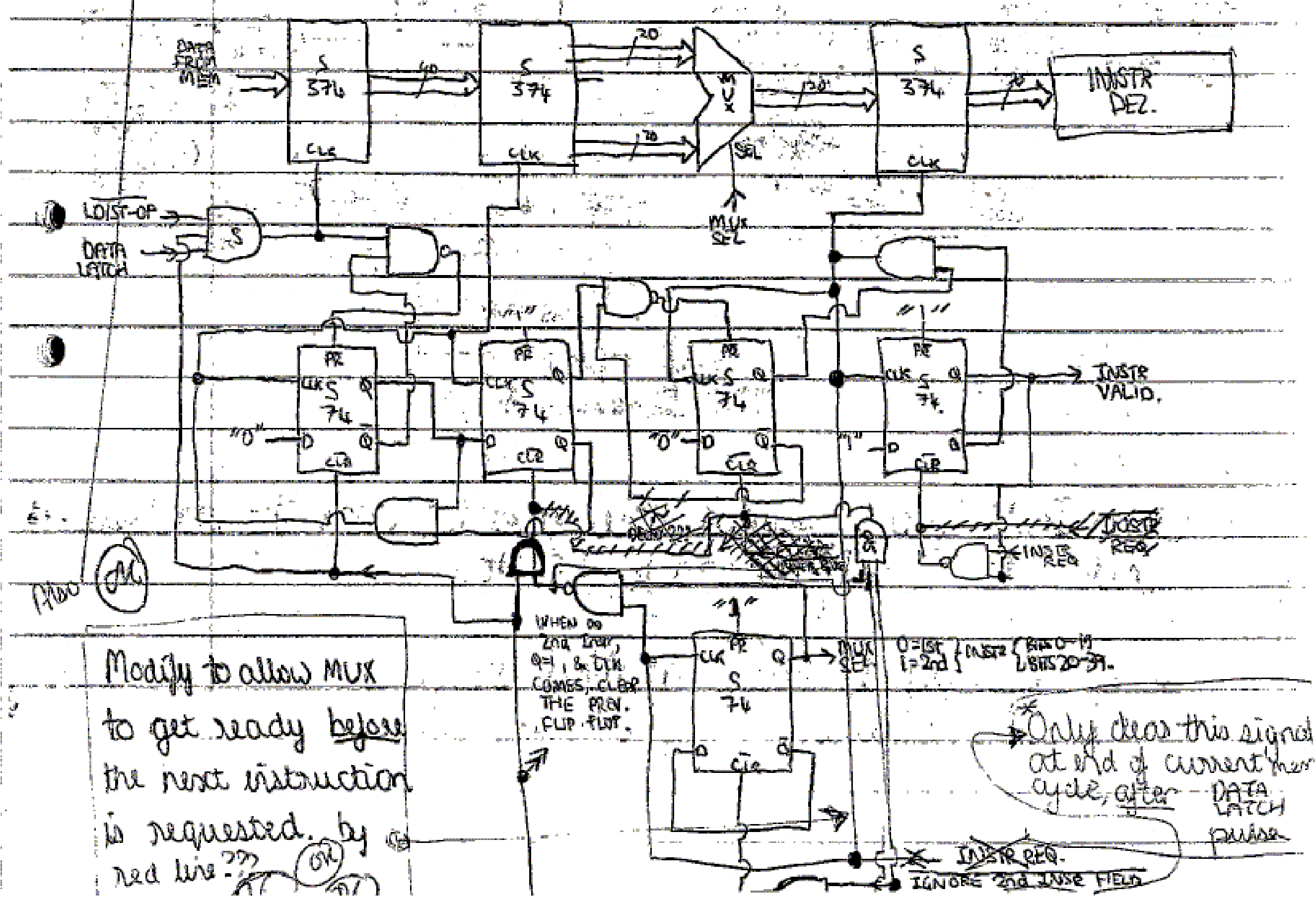
First level shifts by 16 or 0
2nd by 8 or 0
3rd by 7,6,5,4,3,2,1,0
Prop delay is 3 gates.

Design using 151s and 153s:
1 level of 153s, shifts by 24,16,8,0 ; 16 chips
1 level of 151s, shift by 7,6,5,4,3,2,1,0 32 chips
Total 48 chips.
Prop delay 2 gates.

25. Instruction Fetch Pipeline evolution

15th September 1992. After that brief interlude with the barrel shifter, it's back to work on the instruction fetch pipeline. These circuits show the evolution of the circuit, as I iron out the problems one by one...





26. Instruction Queue Unit

Finally a more complete version of the instruction queue unit, which I believe might work.

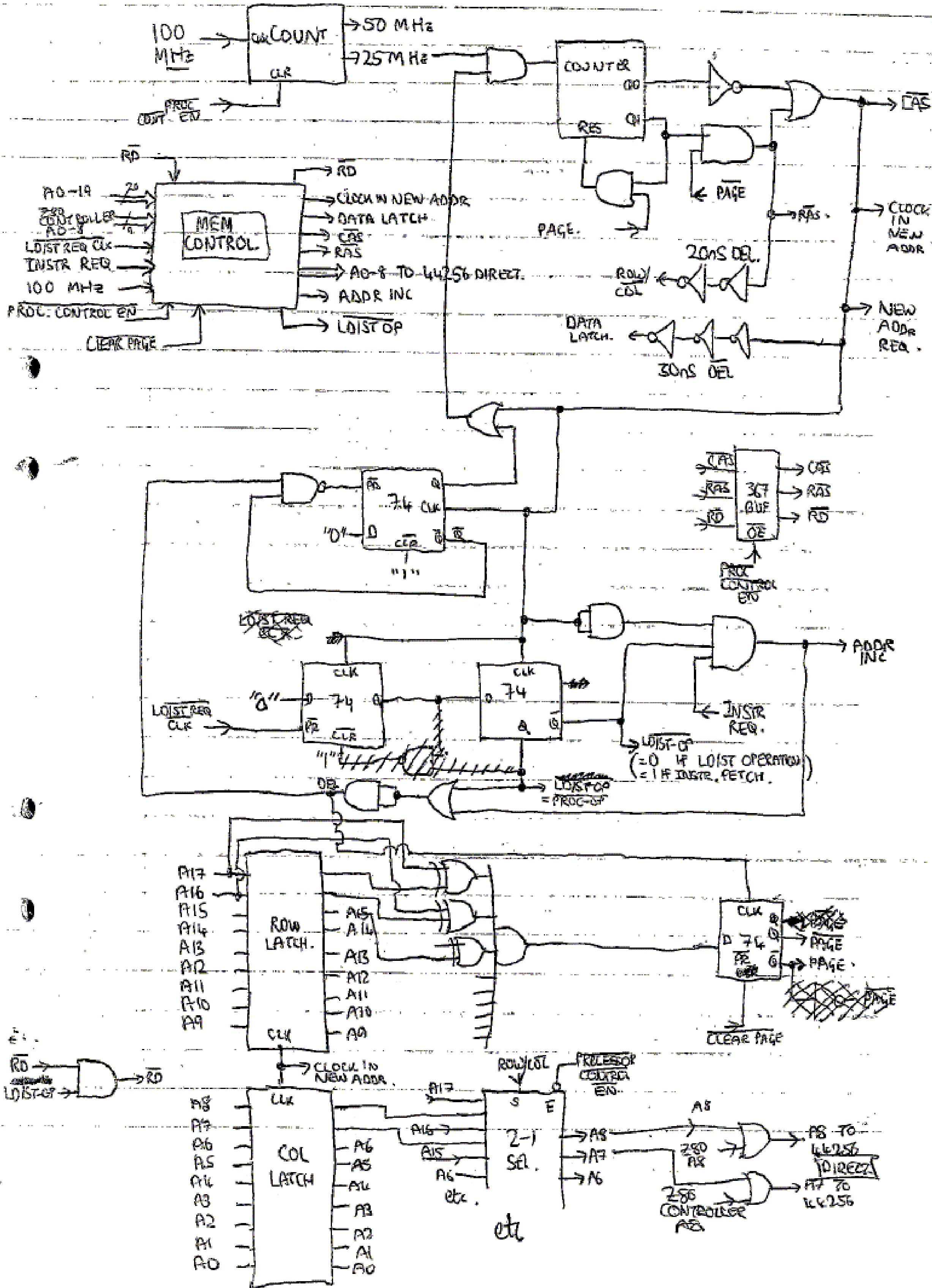
15/9/92. '26.

(Clear Instr Queue signal also goes to clear the address unit etc, adjust page EF etc.)

Is it necessary to have another stage in Instruction queue? Maybe it is ...

27. Memory Controller

The final version of the memory control circuit. This turns the 20-bit address bus into the multiplexed Row and Column form required for the DRAM's, and generates the RAS and CAS signals. It is the only part of the CPU having a clock, the 100 MHz clock is used to generate the precise timing required by the DRAM. The rest of the CPU from then on is entirely asynchronous.



28. Address unit; Memory arbitration

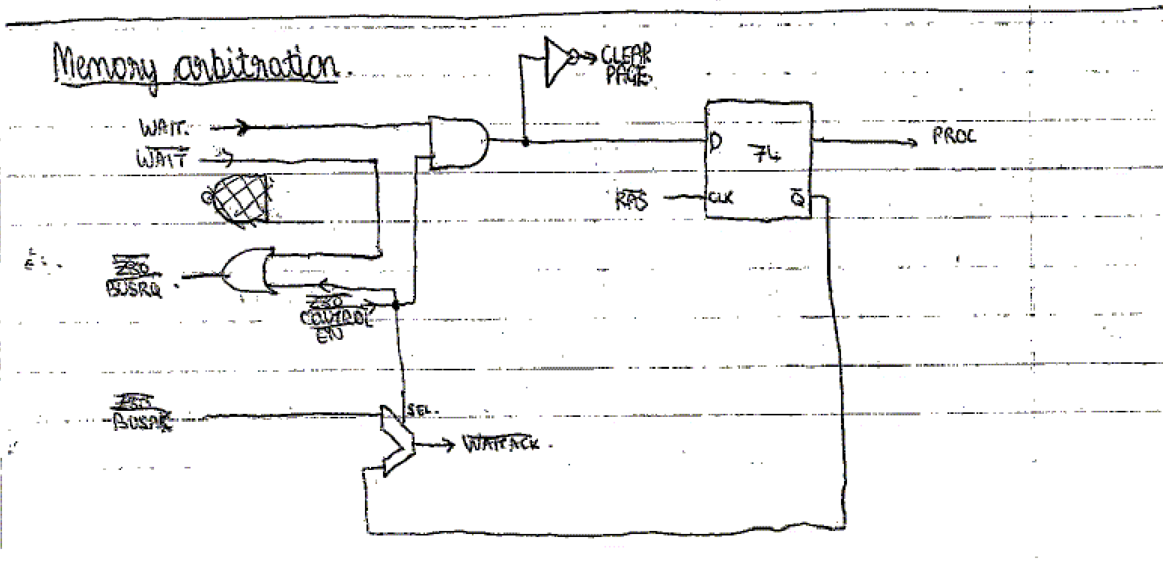
The address unit generates the memory address for the memory controller unit to use to access the DRAM. This address can come from the program counter, the second instruction field of the Jump and Call instructions, or by adding the displacement address of a branch instruction to the current location.

Later I start to think about bus arbitration. This circuit will decide when the CPU can control its own memory, and when it should yield control to the host Z80 so that the host can read/write the memory.

17/9/92 28

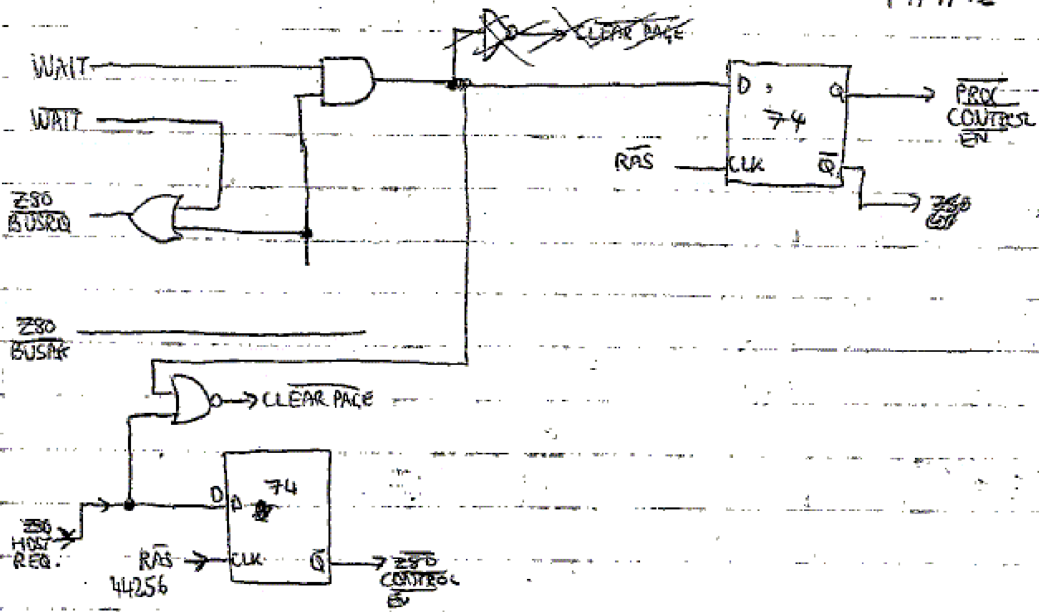
Does come from here as it is the current address that must be stored not the one currently being read/written

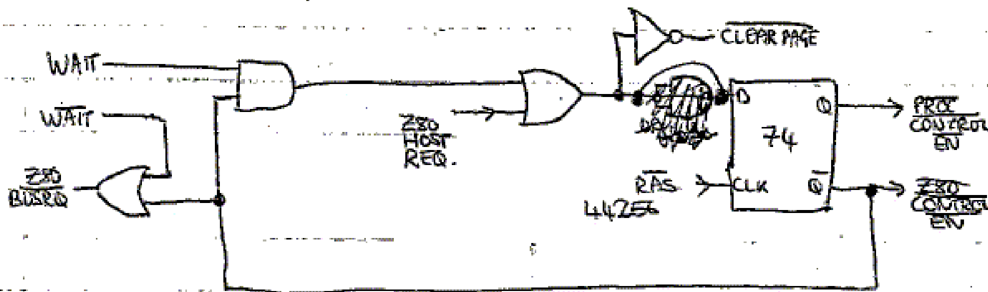
374
374
EXTRA STAGE MAYBE.
OE "0"
CLK
LD/ST-OP
NEW ADDR CLOCK
QUEUE CLEAR
QUEUE CLOCK
ADD
DISP. FIELD.
245 BUF
READ ADDR
(TO DATA BUS)
DATA BUS 0-19
2nd INSTR FIELD
ABS/REL
ADDRESS (0-19)
Address Counter
LOAD INPUTS.
ADDR INC
LOAD ADDR COUNTER
LD/ST-OP
OR
ADDRESS UNIT
CLOCK NEW ADDR
QUEUE CLEAR
QUEUE CLOCK
READ ADDR
ABS/REL
ADDR INC
LOAD ADDR COUNTER
DISP FIELD
2nd INSTR FIELD.
A0-19.
DATA BUS 0-19

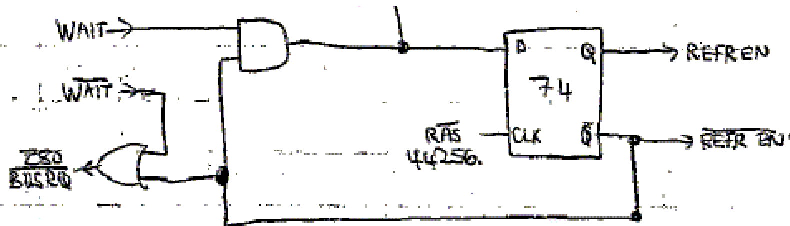


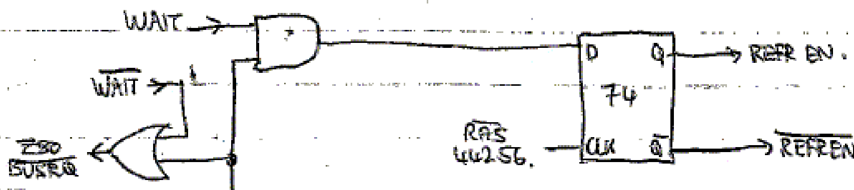
29. Evolution of the Memory Arbitration Circuit

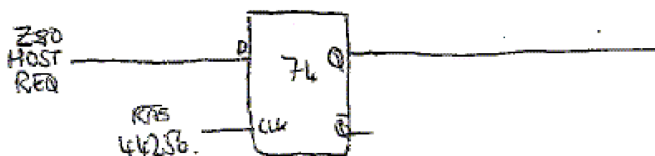
A few more attempts at designing a working circuit for the memory arbitration.





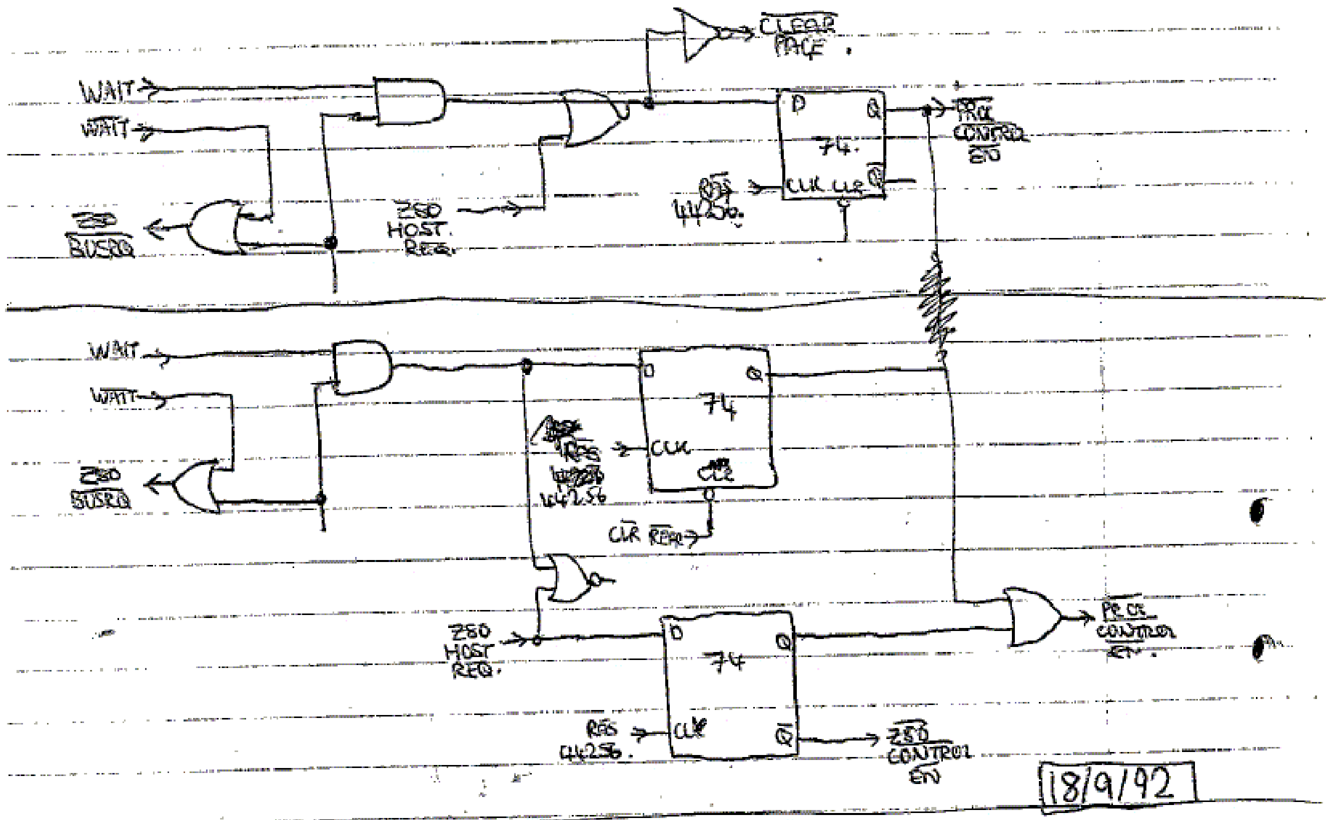


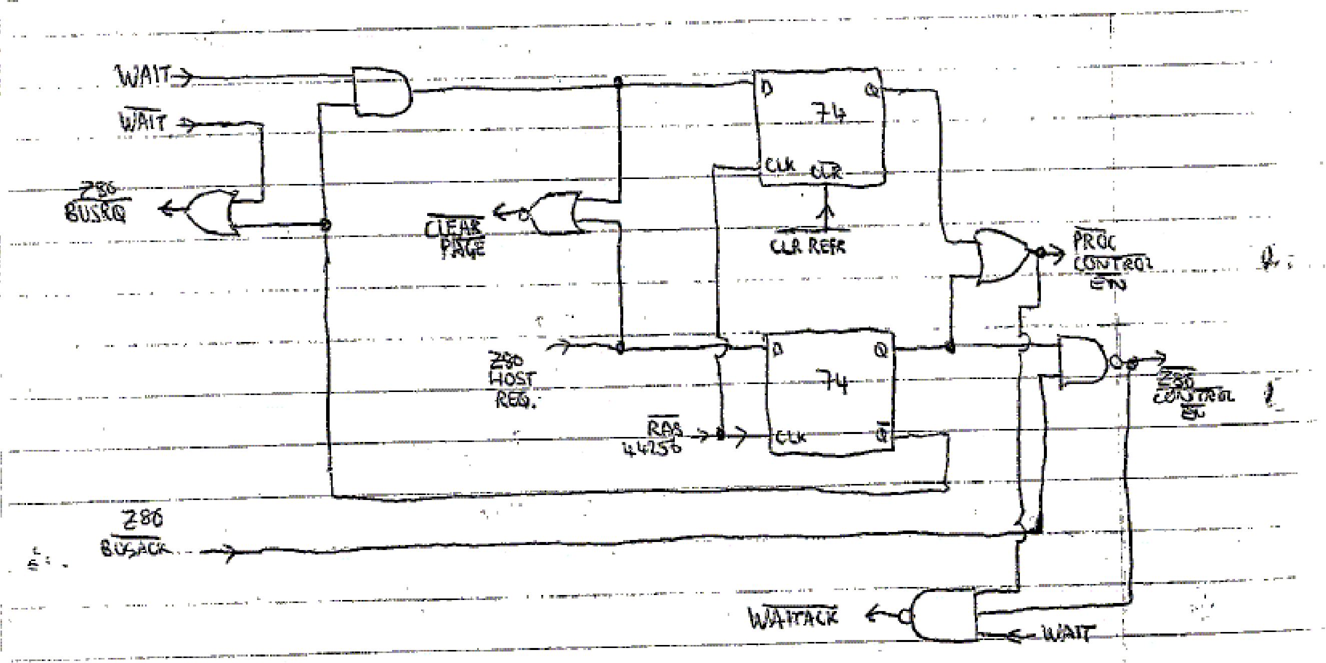


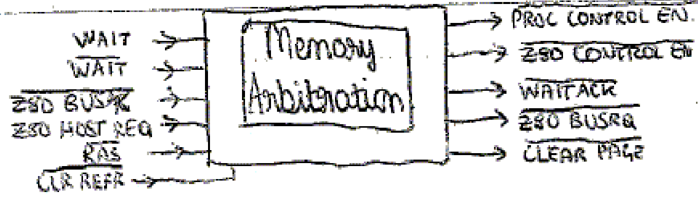


30. Final Circuit for Memory Arbitration

18'th September 1992 rolls around and I decide that my memory arbitration circuit design is complete. It controls the Z80 BUSRQ and WAIT signals and arranges for a clean handover of control when the Z80 requests it.







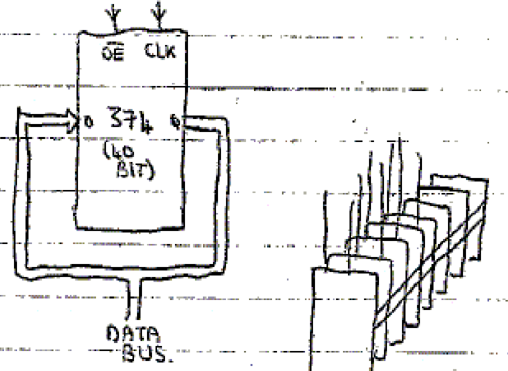
31. Register File Definition

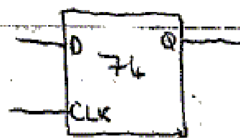
A reminder of the register naming and useage. Each register also has its own scoreboard bit (half a dual D-type 74LS74 flip flop), whose state is "0" if the register is available or "1" if it is busy.

18/9/92 31

Register file:

Register		Use	Access
R0	=	All zeros	
R1	=	A	
R2	=	B	
R3	=	C	
R4	=	D	
R5	=	E	
R6	=	F	
R7	=	G	
R8	=	H	
R9	=	I	
R10	=	Index (20)	Write only
R11	=	IN register	Read only
R12	=	OUT register	Write only
R13	=	Loop counter (20)	~~Write only~~
R14	=	Stack pointer (20)	
R15	=	PC (20)	





"0" = Available
"1" = Busy.

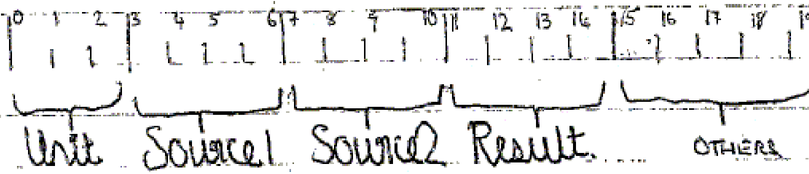
19/9/92

32. Instruction Decoding and Scheduling

The instruction decode is conceptually simple. The first 3 bits of every instruction specify the unit to be used for the instruction. The instruction is "issued" when the unit scoreboard indicates the required execution unit is available, and the scoreboard also indicates each of the source and destination registers is available.

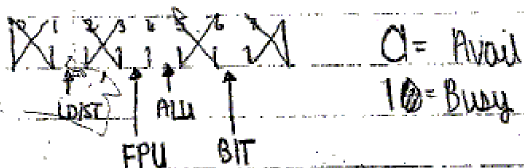
19/9/82 32

Instr decode:

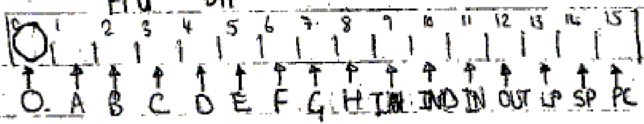


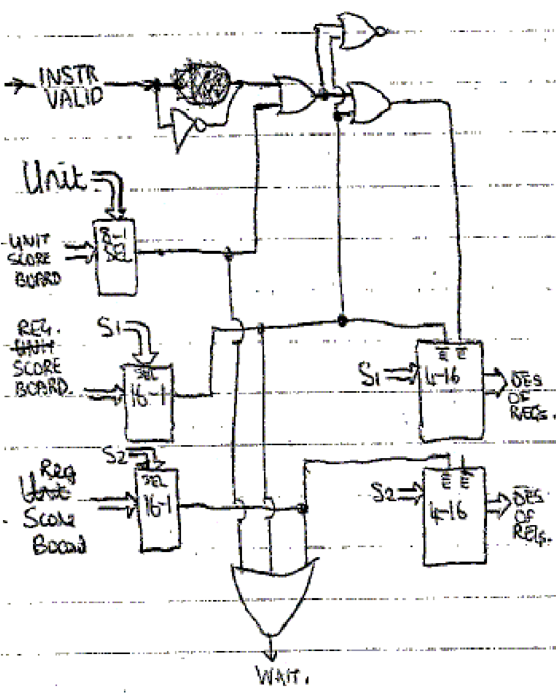
→ INSTR VALID
← INSTR REQ
← IGNORE 2nd FIELD
← CLEAR INSTR QUEUE.
⇒ INSTR.
← LOAD ADDR COUNTER (PC)
← READ ADDR (PC)
← ABS/REL

Unit Scoreboard:



Reg. scoreboard





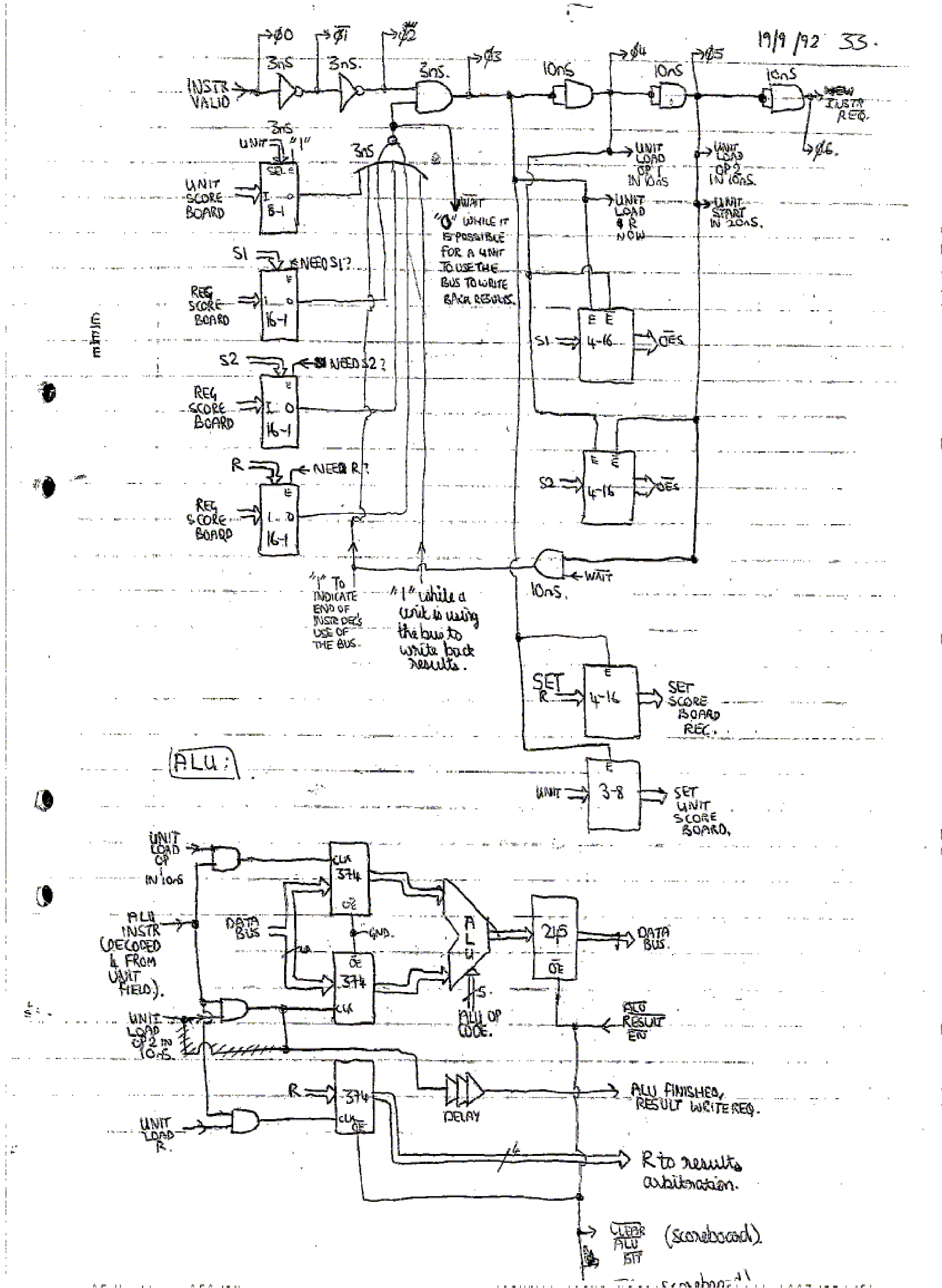
Unit/op codes

	0	NOP/HALT
*	1	LD/ST
	2	Reg.
*	3	F.P. incl ADD/SUB etc Shift, normalise
*	4	ALU
	5	
*	6	Bit manipulation.
*	7	Jumps etc.

33. Instruction Scheduler / First ALU thoughts

Final diagram of the scheduler. Given an "Instruction Valid" signal from the instruction queue, it checks register and unit availability. If everything is Ok it enables the correct source register outputs and generates signals for the execution units to start processing. The scoreboard bits then get set (unit and result register).

Also shown here are my first thoughts on the ALU design.



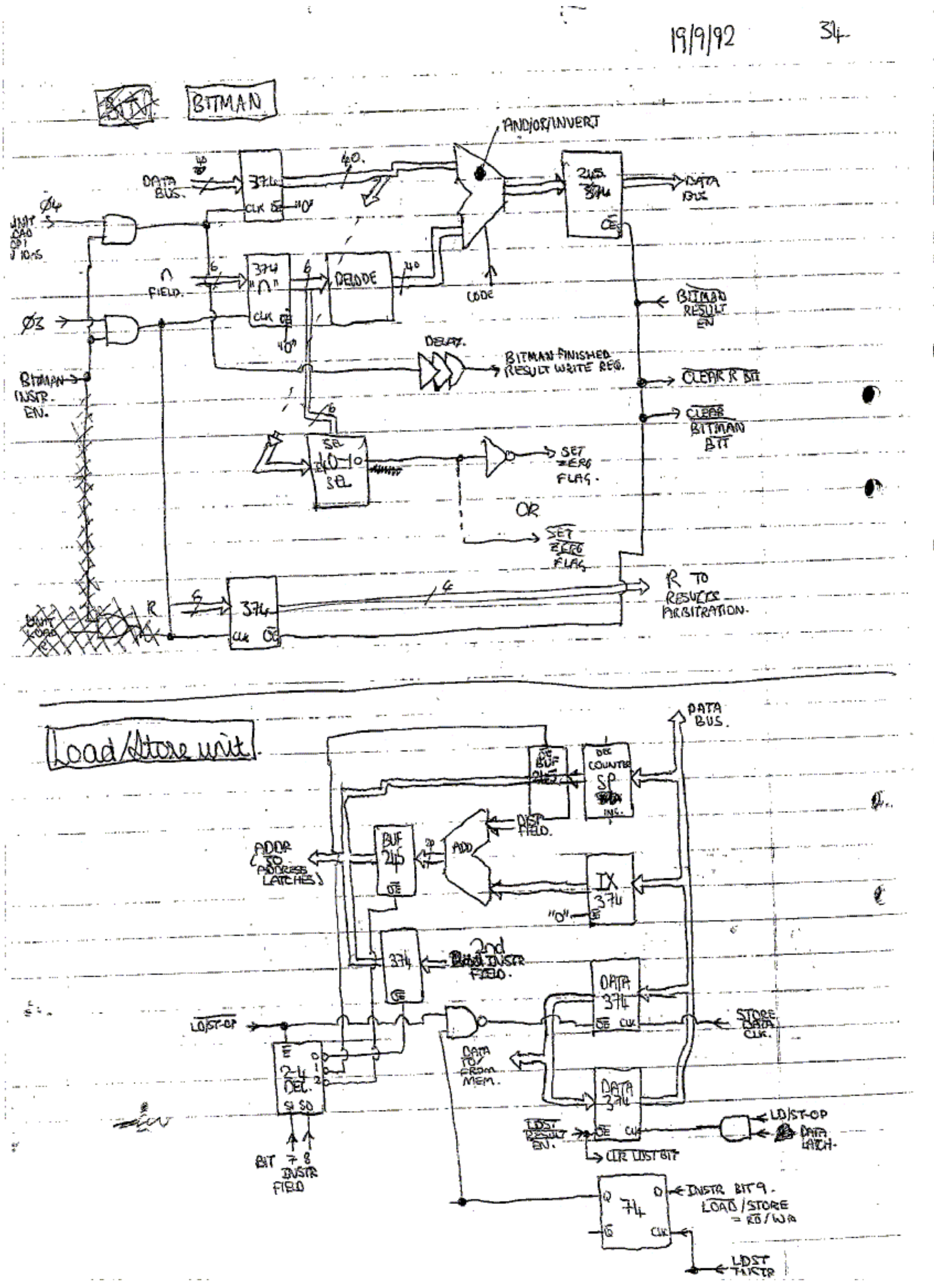
34. Sketches for the BitMan and Load/Store Units

Some preliminary consideration of the Bit Management (test/set/reset) unit. This one can reset any specified bit to "0", set it to "1" or test its state.

The load/store unit is responsible for generating the required memory address. Depending on the state of instruction bits 10, 11 and 12, this address is one of

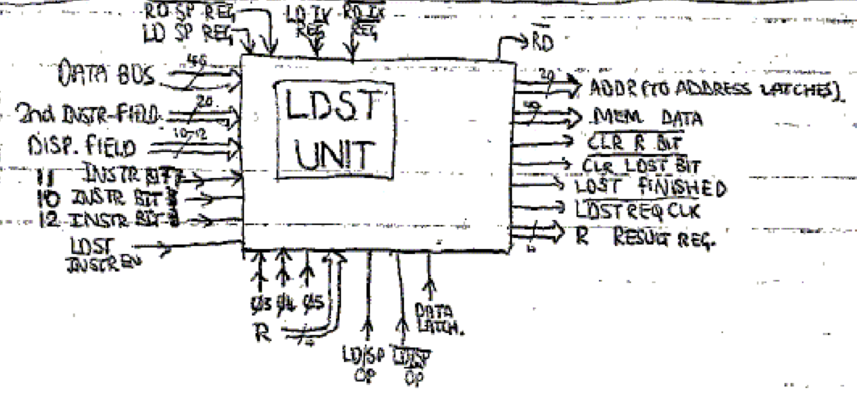
o 0: an absolute address specified in the 2nd instruction field (40-bit instruction)
o 1: Stack Pointer: The address specified by the stack pointer is used
o 2: Index Register + displacement specified in the instruction

When the stack pointer is used, it is subsequently incremented or decremented automatically by this unit, corresponding to a stack "POP" or "PUSH".



35. Load/Store Unit

Final diagram of the Load/Store Unit. Note the nice block representation at the bottom of the page showing all input and output signals connected to this unit.

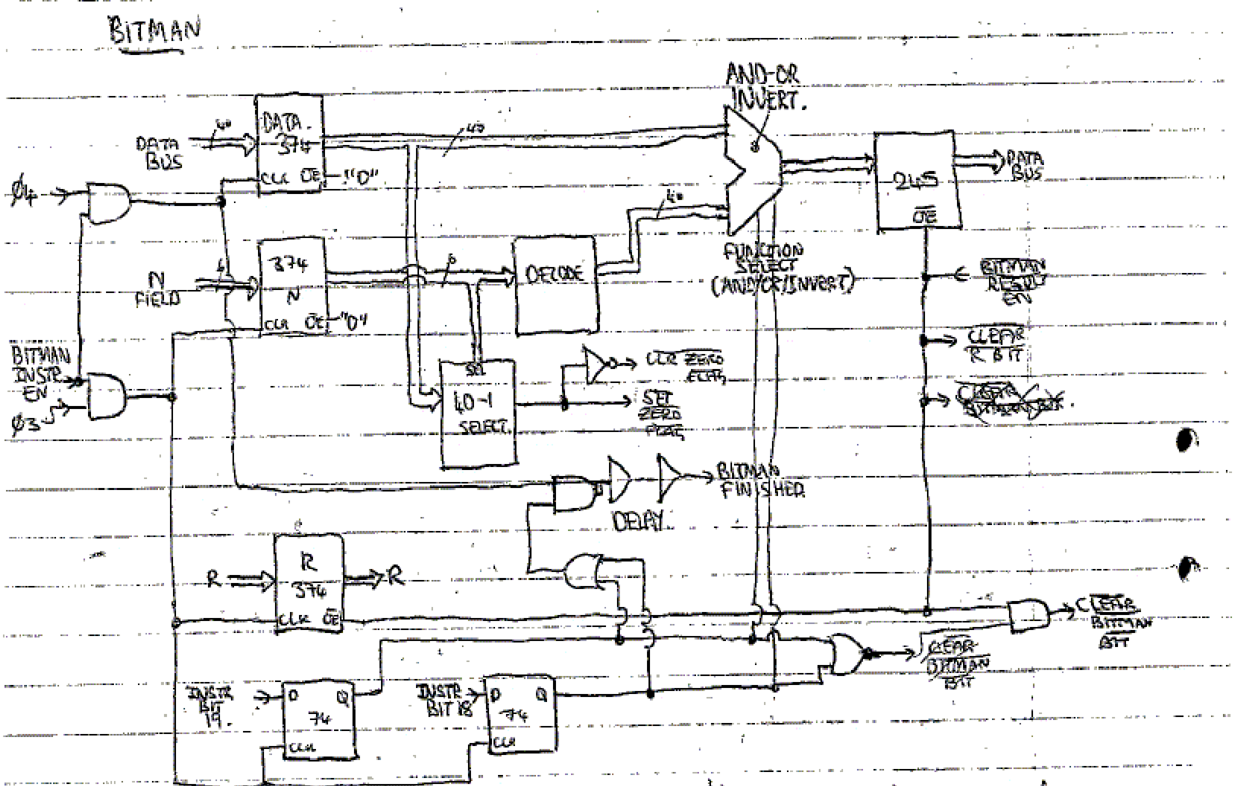


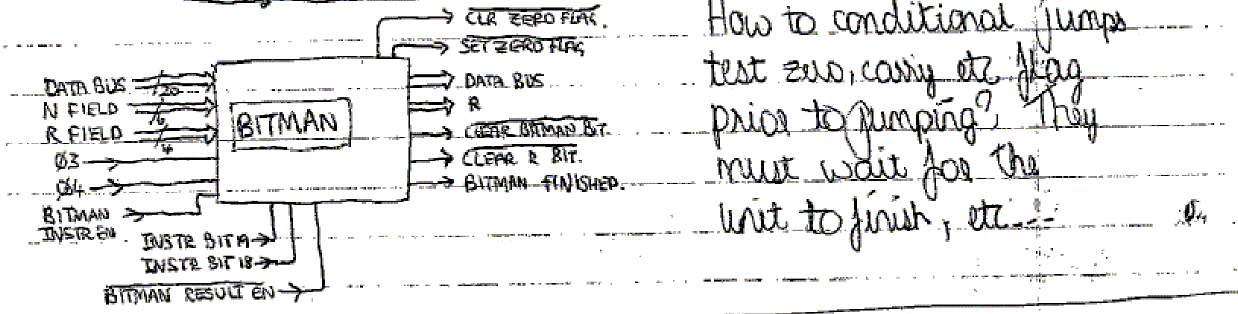
36. Bit Management Unit; ALU

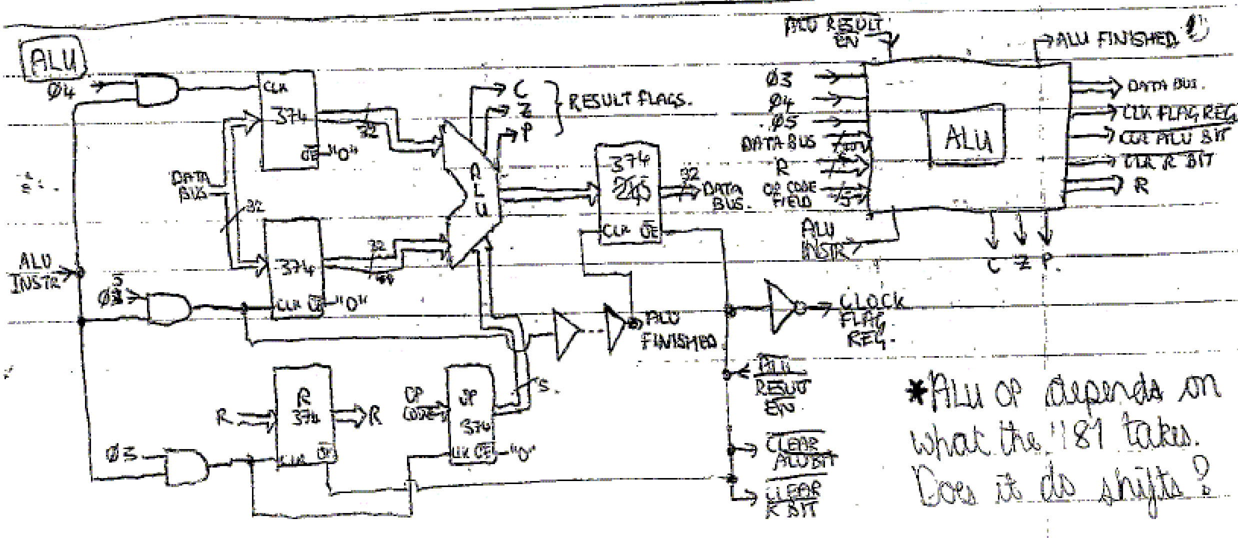
The final version of the bit management unit, responsible for reset to "0", set to "1", or testing of any bit.

In the lower part of the page, a diagram of the ALU. This is rather simple, mainly because it uses the 74LS181 4-bit ALU (8 of them).

20/9/92 36







37. Result Arbitration

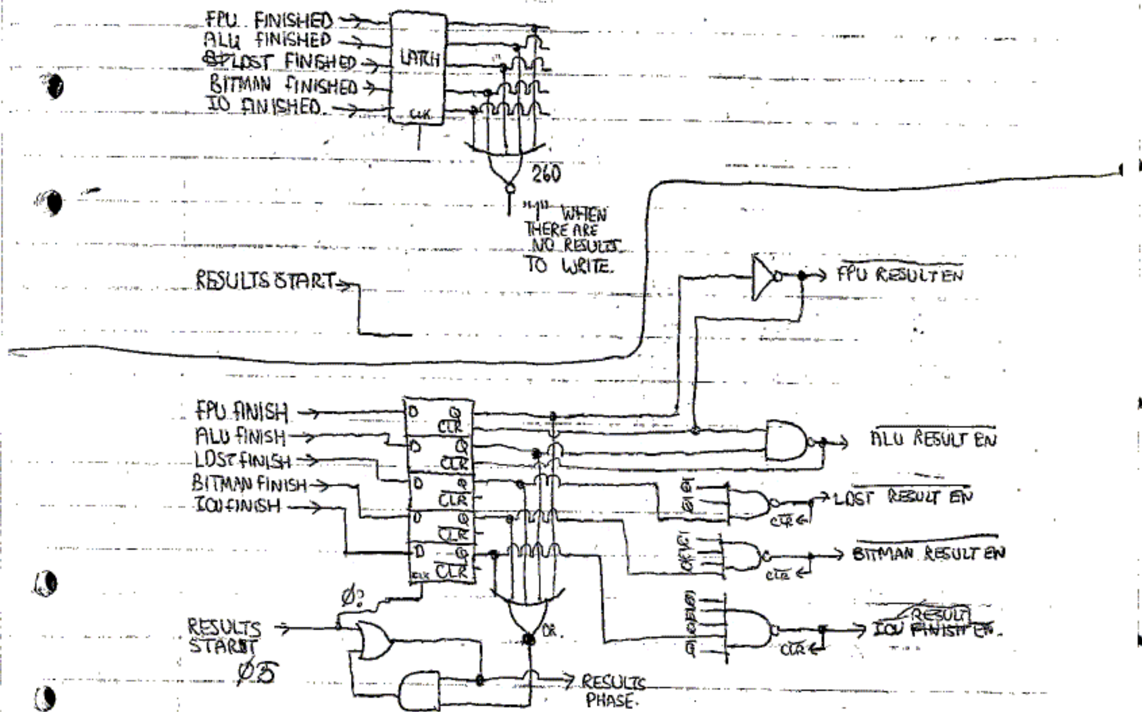
One of the most important parts of the CPU design! All the units operate asynchronously, and can be operating in parallel. Yet they all ultimately want to write the results of their computation to the register file using the internal bus. The result arbitration circuit helps to decide who can use the bus at what time.

There are 5 units which may need to write back results. In addition the instruction issue unit will require the bus to load source data from the register file to the units. I therefore arrange a stack of 5 flip flops. There is a "result phase" during which the instruction issue unit allows results to be written. In this phase, each of the 5 possible units are checked and in turn allowed to write their results if they need to.

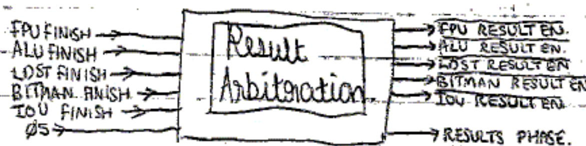
20/9/92 37.

Results Arbitration.

Result returning units:		Priority
	FPU	1
	ALU	2
	LDST	3
	BitMan	4
	IOU	5.



* Will this work, regarding OE propagation delays etc; in this circuit & in requesting unit's OE of '374/'245.

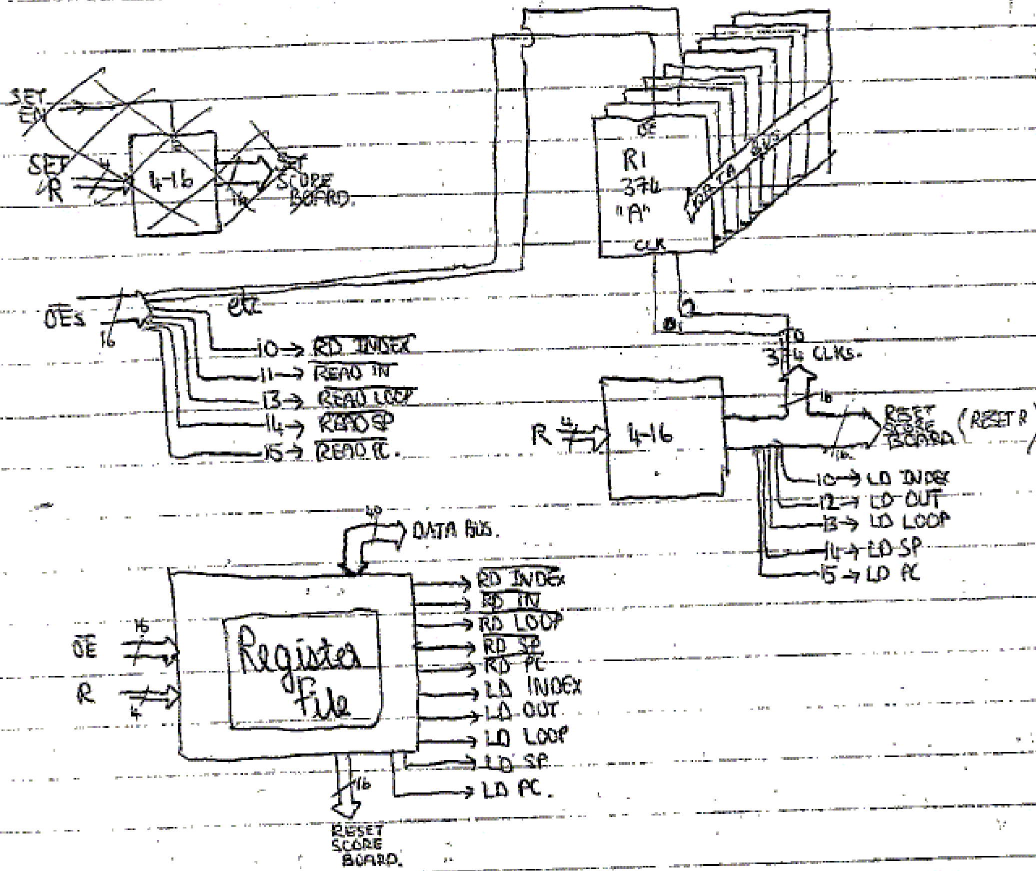


NB. IOU counts as a register not a unit, so arbitration is not necessary.

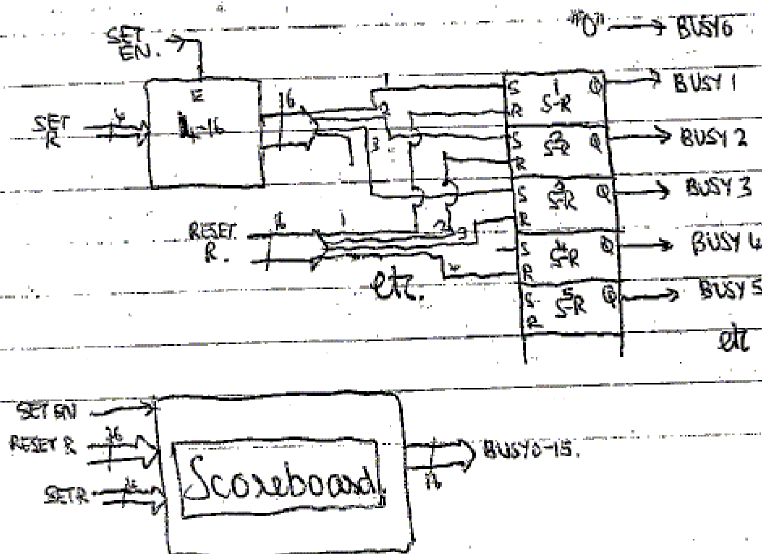
38. Register File and Scoreboard

This page shows the construction of the register file and scoreboard. Register 0 is always zero by definition, so it is NEVER busy.

Register file:



Scoreboard

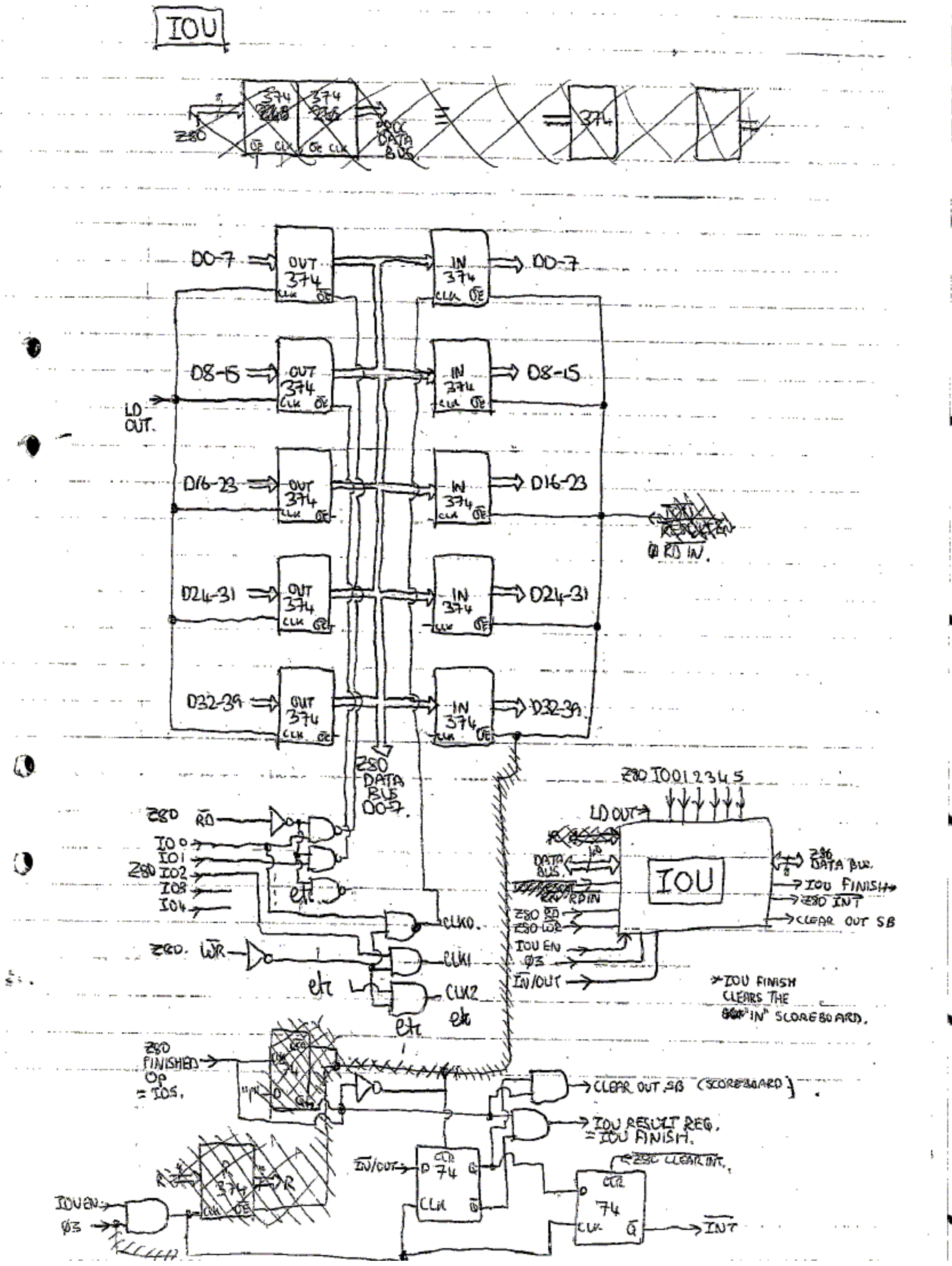


39. IOU, the Input/Output Unit

This is a set of 74LS374 octal D-type latches, 40-bits wide, one bank for the input and on e for the output. Using these the host Z80 can send data to and from the CPU.

When the CPU executes an IN instruction, a Z80 interrupt gets generated. The IOU is then marked as busy on the unit scoreboard until the Z80 has loaded all 5 8-bit registers, so a complete 40-bit CPU word is ready. Only then does the IOU send a "finish" signal to the Result Arbitration circuit.

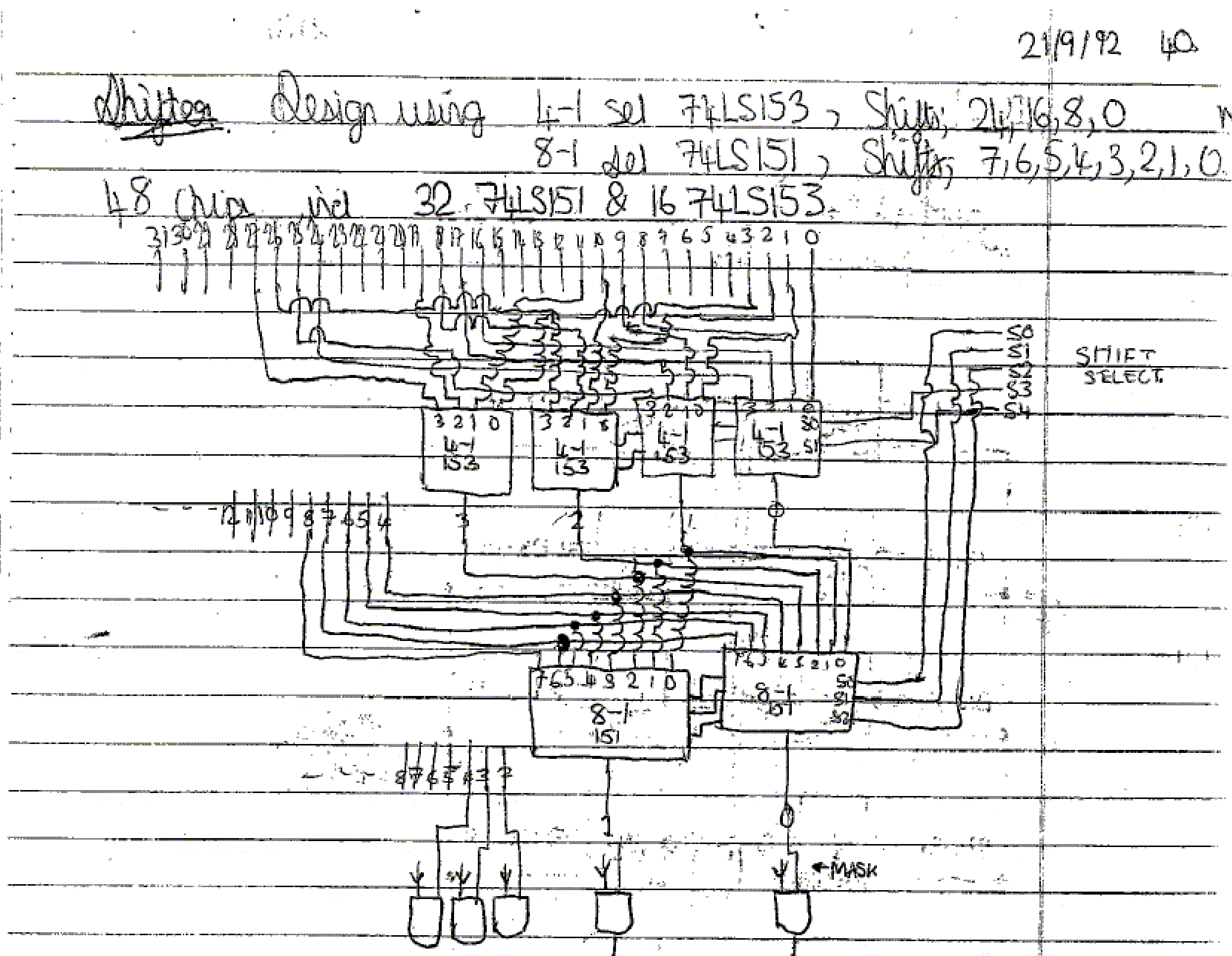
When the CPU executes an OUT instruction, a Z80 interrupt is also generated. The IOU busy scoreboard bit remains set until the Z80 has read all the 5 8-bit chunks of the 40-bit word.



40. Barrel Shifter sketch and FPU addition notes

Here is a design for a barrel shifter using two states: first a set of 74LS153 dual 4-1 multiplexers, followed by a set of 74LS151 single 8-1 multiplexers. This operates on 32 bits and can shift by any number of bits in just these two stages of logic. Only 32 bits are considered, cause this barrel shifter is meant to be used in the floating point unit to normalise the fractional part.

Later I start to work out what is involved in floating point addition. A few worked examples are required...



Or do without mask, just ground all unwanted inputs to the 153s & 151s; could result in lower chip count.

F.P Addition: 11+3

i) 11 = 1011 integer = 1011 e 00 F.P

3 = 0011 integer = 1100 e -10 F.P

11+3 exponents subtracted give 0--2 = 2 so shift of 2 is reqd, in 2nd operand: ⇒ 0011, Now add:

⇒ 1110 e 0 Normalise.. stays same.

ii) 17+13: 17 = 00010001 e 128 = 10001000 e 125

13 = 00001101 e 128 = ~~1000~~

= 11010 e 124

Sub. exp. ⇒ 125-124 = 1 ; shift 2nd right by 1:

41. On the subject of Floating Point Addition

More deliberations and worked examples as I attempt to understand how to add two floating point numbers together.

21/9/92 41.

10001000 e125
01101000 e 125
11110000 e 125 No norm req.
= 00011110 e128 = 16+8+4+2 = 30 ok.

iii) 30+3 30= 11110000 e125
3= 00000011 e 128 = 11000000 e 122

Sub exp: 125-122=3 ∴ shift right 3:
11110000
00011000
100001000 e 125
Norm: shift right 1 ⇒ 10000100 e 126
= 00100001 e 128 = 33 ok.

iv) 4+48 4= 00000100 e0 = 10000000 e123
48= 00110000 e0 = 11000000 e126

Sub exp. 123-126 = -3. Shift 1st by 3, take 2nd exp
11000000 e126
00010000 e126 = 00110100 e128
11010000 e126 = 32+16+4 = 52.

125-122 125= 01111101
122 122= 01011010 01111101
122 = 10000101 +1 = 10000110
100000011
= 3.

123-126 126=
123= 01111010 01111101
126= 01111101 +1,00000101
126 = 10000010 100000010 = 2.
2C = 10000011 01111010
10000011
11111101
c 00000010
+1 00000011

125
-64
61
-32
29
-16
13
-8
5

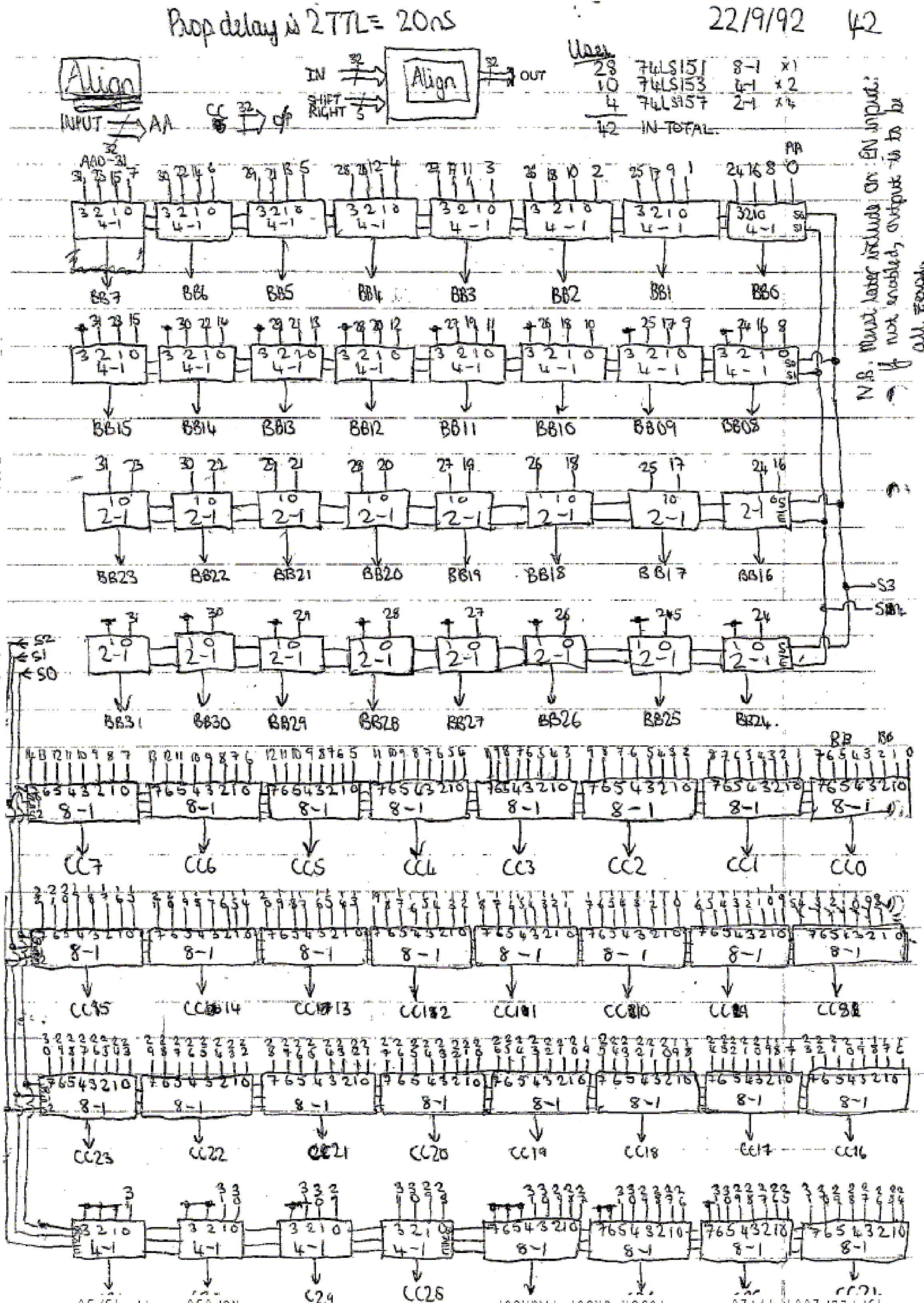
122
-64
58
-32
26
-16
10

123
-64
59
-32
27
-16
11
-8
3

42. Align Block

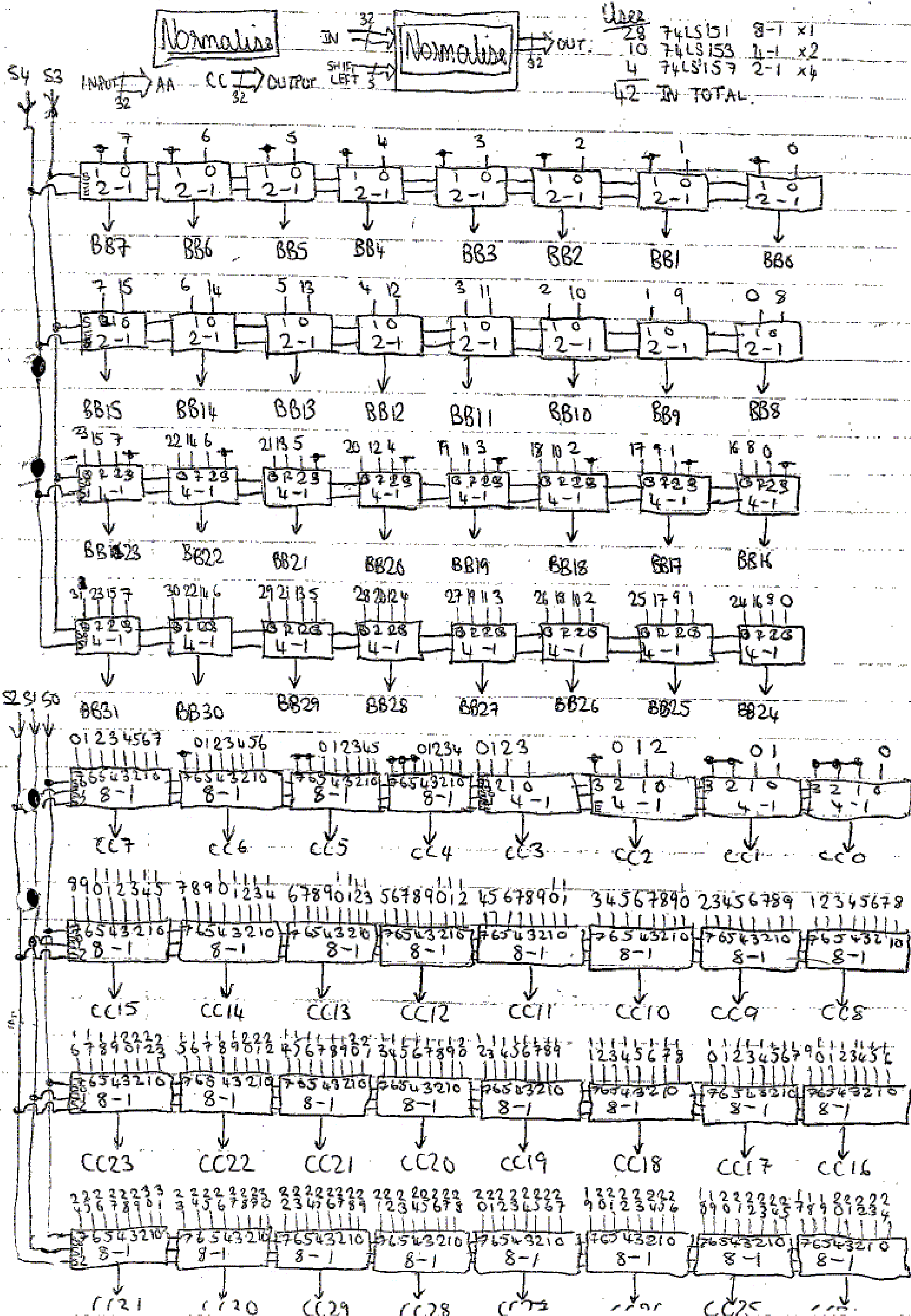
This is a development of the shifter sketched previously. Alignment of one of the operands in a floating point addition operation is required prior to the actual addition. The smallest operand is shifted right by the difference in the exponent fields. What I are really doing here is lining up the decimal points so I can later add the two numbers.

The align block can shift the 32-bit fractional part of a floating point number (well actually 31 bits), by any number of bits in the range 0..31. It does this in only two stages of logic. To accomplish this feat requires a mere 28 74LS151 chips, 10 74LS153 and 4 74LS157...



43. Normalise Block

Very similar to the Align Block, but in reverse. The normalise block is used after a floating point operation to left-shift the result, incrementing the exponent field accordingly, so that the most significant bit of the floating point fractional field is a "1".



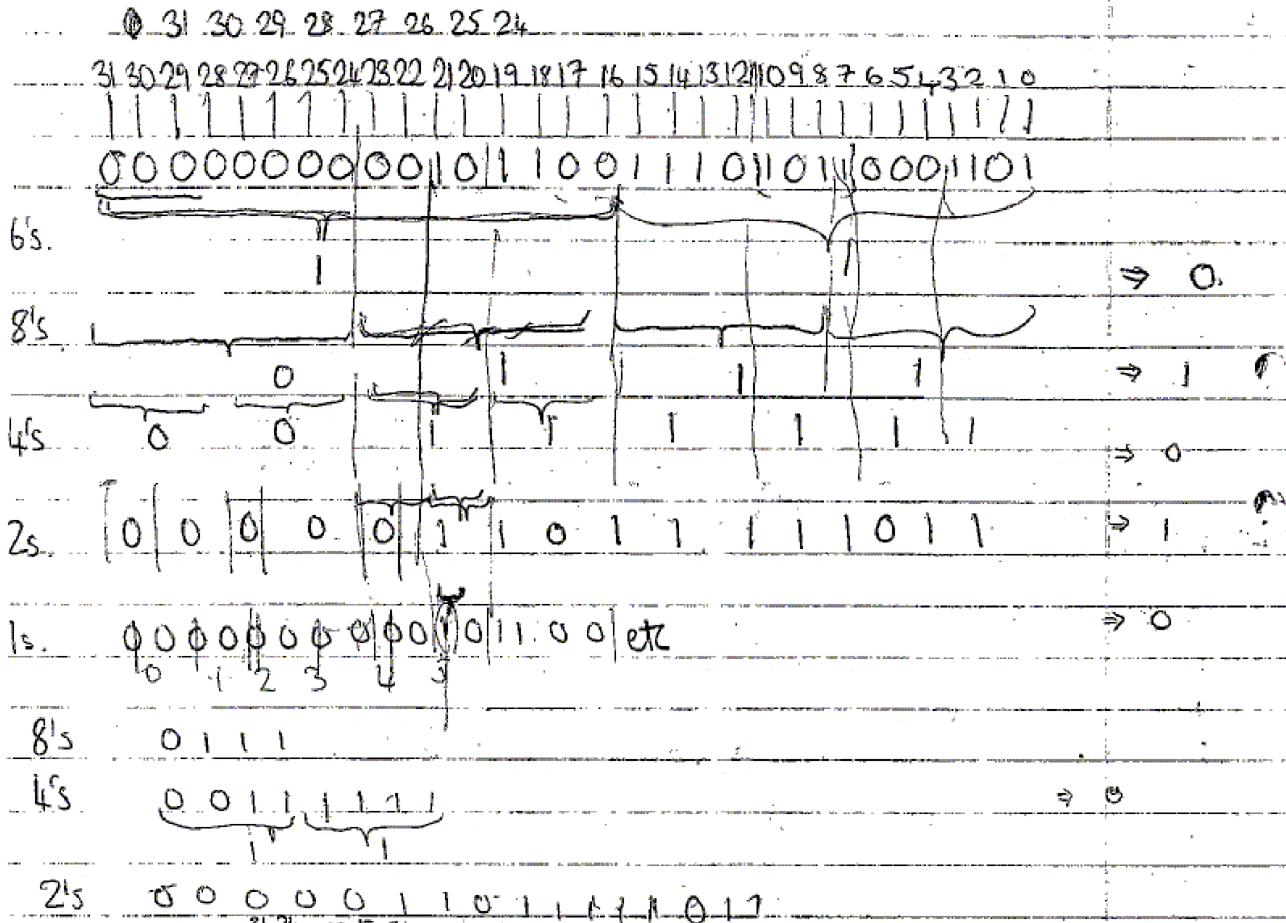
44. On the counting of leading zeros

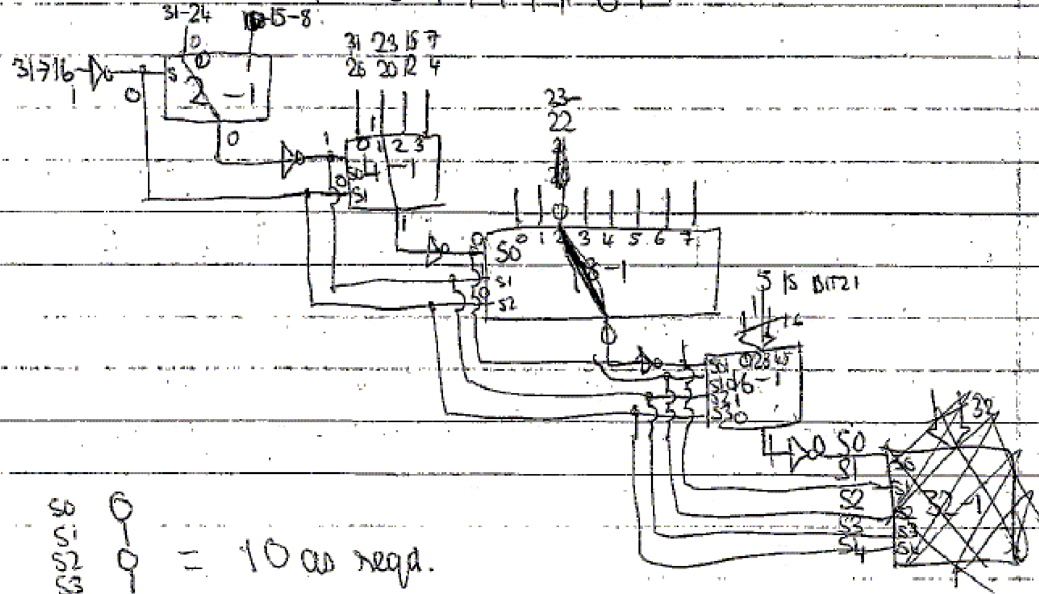
Before I can normalise, I have to count the number of leading zeros in the result's fractional part, so that I can tell the normalise block how many bits to shift left by. Here I have a few thoughts about this task.

Leading Zeros count

LEADING ZEROS COUNT

22/9/92 44



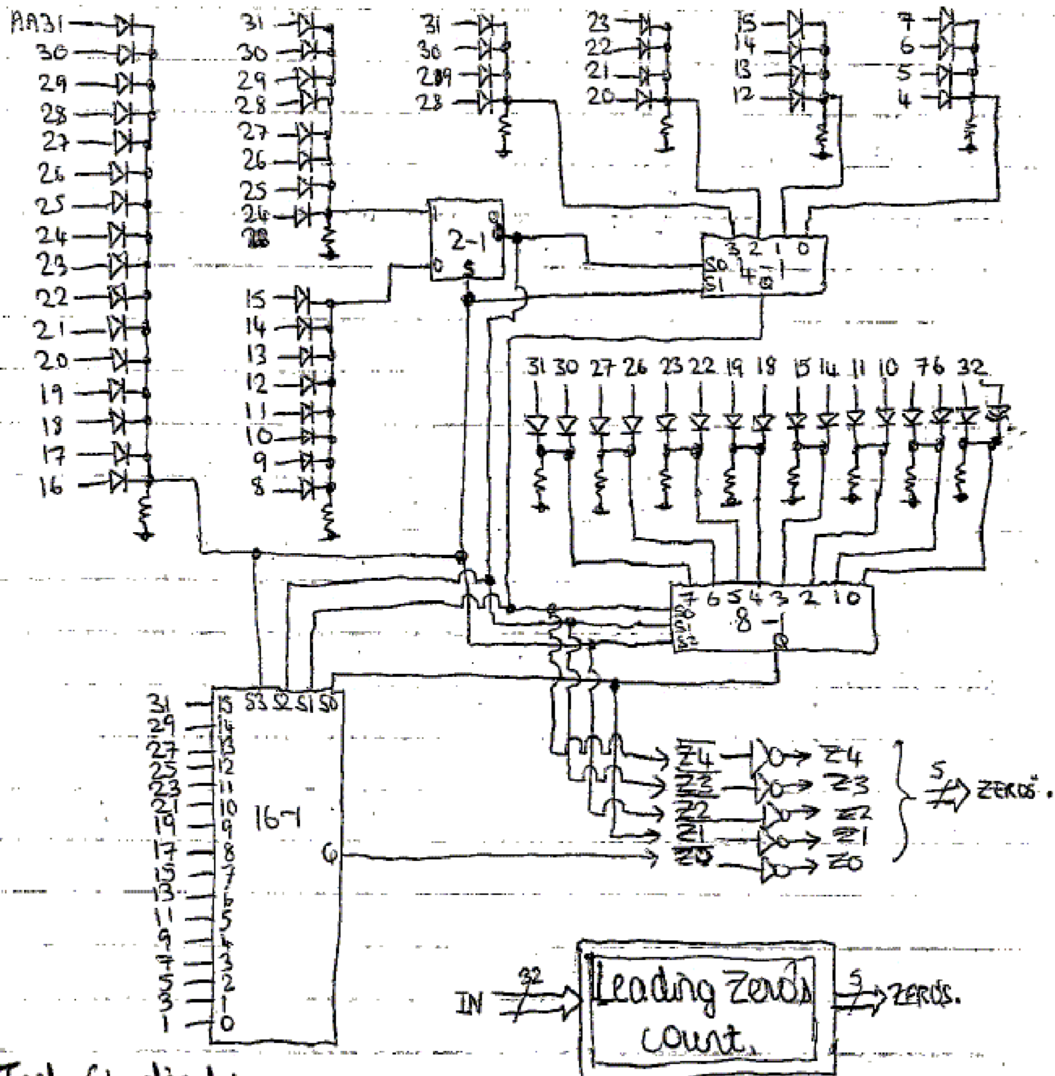


45. Leading Zero Count

The final diagram for the leading zero counter. Note the large number of diodes, which replace TTL OR-gates. I always thought if you needed a large number of inputs to an OR-gate, why not use a set of diodes?

22/9/92 45.

IN ⇒ AA0-31.



*Incl. 64 diodes.

*Propagation delay is 1 diode + 4 TTL.

*If delay in 1 diode is 1nS and 74S series TTL is used with delay 3nS, TOTAL = 13nS.

*+ further 3nS for the final inverting, gives 16nS.

*Alternatively use encoder TTLs if there are any.

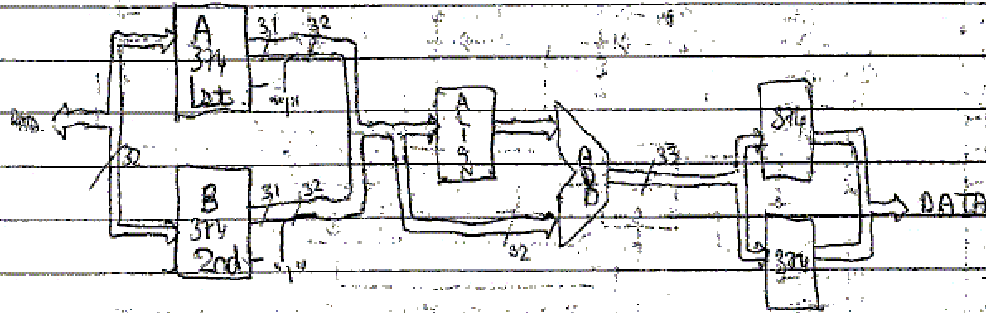
What happens if the count is all zeros?

46. Floating Point Addition Sketches

It's now 25'th September 1992 and I need to remind myself of my newfound understanding of how to add floating point numbers, and how I planned to implement this in TTL.

25/9/92 46.

Add F.P

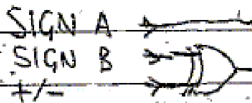


15+3: 0000 1111
0000 0011
0001 0010 = 18

15-3: 15 = 1111
3 = 0011, $\bar{3}$ = 1100, -3 = 1101

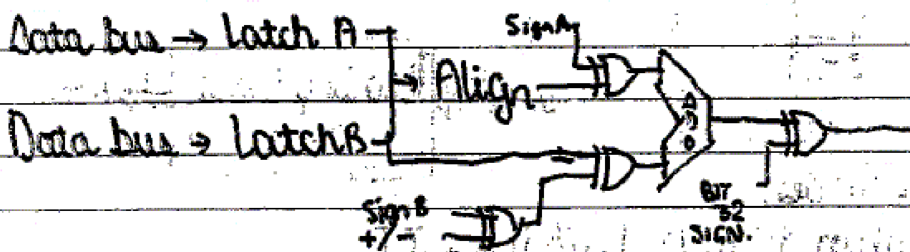
0000 1111
1111 1101
1 0000 1100 = 12

-3-15 = 0000 0011
1111 0001
1111 0100 = -12
0000 1100 = +12



0 = +
1 = -

-3-15 1111 0001
1111 1101
1 1110 1110 = -18
0001 0010 = +18



Sub: Zero count → Normalise → latch
Add: No carry → latch 32
Carry → latch C+31 and Inc. exp, → latch.

47. Add/Subtract Floating Point: First design

Now I add the complication of being able to subtract as well. To do this I use a bank of exclusive OR gates (XOR) before and after my adder. This first sketch design needs considerable further work before it can be said to be anywhere near complete.

27/9/92 47.

Mult F.P.

Latch A

Latch B

Latch → O/P.

Add exponents

ADD/SUB F.P.

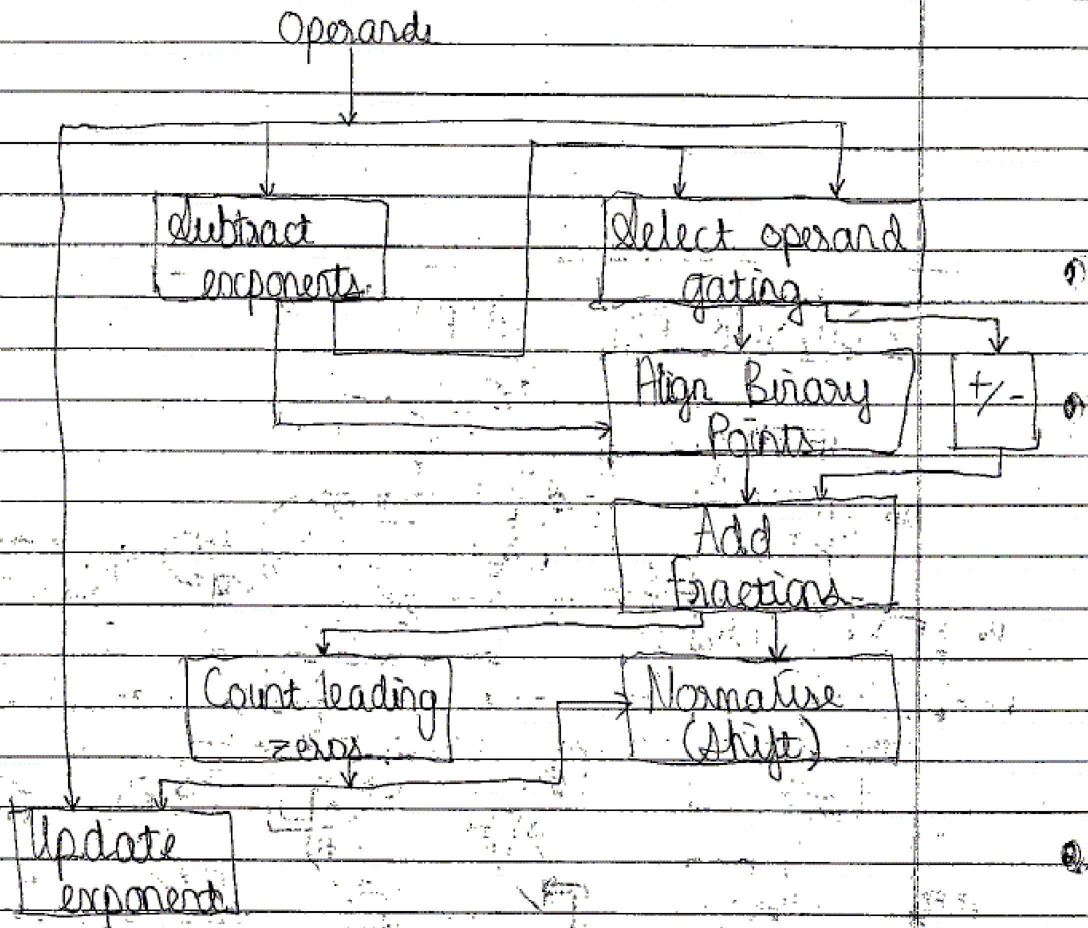
ASFP

48. Library Research

27'th September 1992 and I found myself in a wonderful library. I am not Swedish (despite by name) but on this date I had occasion to be present at the KTH technical university library in Stockholm. Here I found various books about computer architecture and the floating point implementations of the IBM 360/91. On this page I jotted down some useful information.

27/9/92 48

- Clock millisecond; 36 bits times ≈2 years (795 days)
- Instruction Control Unit ICU fetches, decodes, and indexes instrs
- IBM 360/91 FP Adder:



- Divide Algorithm, IBM 360/91:
Uses multiplier unit in iterative algorithm with rapid (quadratic convergence).
Performed by sequence of multiplications; multipliers chosen so that divisor is driven towards 1; quad. conv, ie no. of bits of precision doubles at each iteration. Each time divisor multiplied, dividend multiplied by the same value. Dividend & divisor are like numerator & denominator of a fraction, (the quotient). As denominator → 1,

49. More Library Research

On the subject of floating point viision and multiplication. Oh no! I have only recently understood how to add floating point numbers let alone multiply or divide them.

27/9/82 . 49.

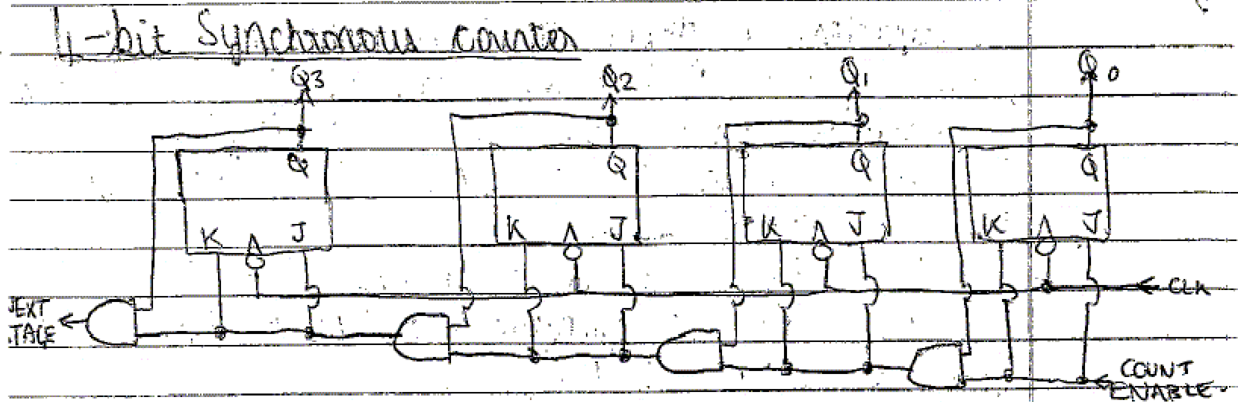
The dividend is driven to the quotient value. (mantissas.)
A & B = dividend & divisor; Answer = A / B = Quotient Q
Do sequence $(A/B)(M_0/M_0)(M_1/M_1)\ldots(M_N/M_N) = (Q/1)$
In normalised form, $0{\cdot}5 \leq B < 1$ so (B = divisor mant)
$B = 1-b$ where $b < 0{\cdot}5$
$(1-b)(1+b) = 1-b^2$ leads to choosing
$M_0 = 1+b$, the 2's complement of the divisor, $1-b$.
This results in $(B)(M_0) = 1-b^2$
Similarly, $(1-b^2)(1+b^2) = 1-b^4$
and $M_1 = 1+b^2$, 2's comp. of the divisor's current value. For kth iteration,
$(1-b^{2e^k})(1+b^{2e^k}) = 1-b^{2e^{k+1}}$
ie $M_k = 1+b^{2e^k}$, the 2's comp of the divisor's current value, $1-b^{2e^k} = (B)(M_1)(M_2)\ldots(M_{k-1})$
Since $b < 0{\cdot}5$, $1-b^{2e^k}$ converges quadratically to 1. With B normalised, mantissa's high-order bit is a "1", guaranteeing initial precision of at least one bit; After 5 iterations, precision is 32 bits as read; (Final multiply of the divisor is not necessary, as it would result in a value of "1" for denominator, leaving quotient in numerator)
Further improvement is table look-up: provides value of 1st multiplier (for 1st 3 multiplies) & assures precision of 7 bits, etc.
Multipliers may be truncated to improve performance, as full precision products not needed. Multiplications of numerator & denominator are performed concurrently in alternate stages of the pipeline.
[From "Supercomputer Architecture", B Paul B. Schneck].

50. A 4-bit Synchronous counter and Floating Point Multiplication

Here I note down the circuit for a 4-bit synchronous counter. I also found this in the library and thought it might come in useful. I did mention sometime earlier in these CPU notes that I needed a fast synchronous counter, well here it is.

I also decide to try to understand floating point multiplication. Once again, a few worked examples are the best way...

27/9/92 50



Multiply F.P, MFP MULT

1000 x 1000 = 8x8=64 = 0100 0000
1111 x 1111 = 15x15=225 = 1110 0001
1111
00001111 = 15 = .11110000 e +4
00001111 = 15 = .11110000 e +4
x ~~00000000~~
11110000 x 11110000 = 240^2 = 57600
= .1110000100000000 e +8

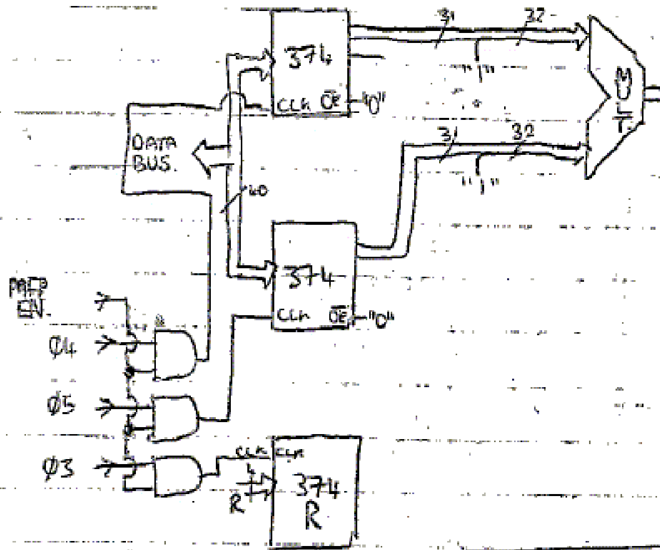
8 x 8 :
00001000 = 0.10000000 e 4
.10000000 x .10000000
= .0100 0000 0000 0000 e 8
= 0.1000 0000 e 7

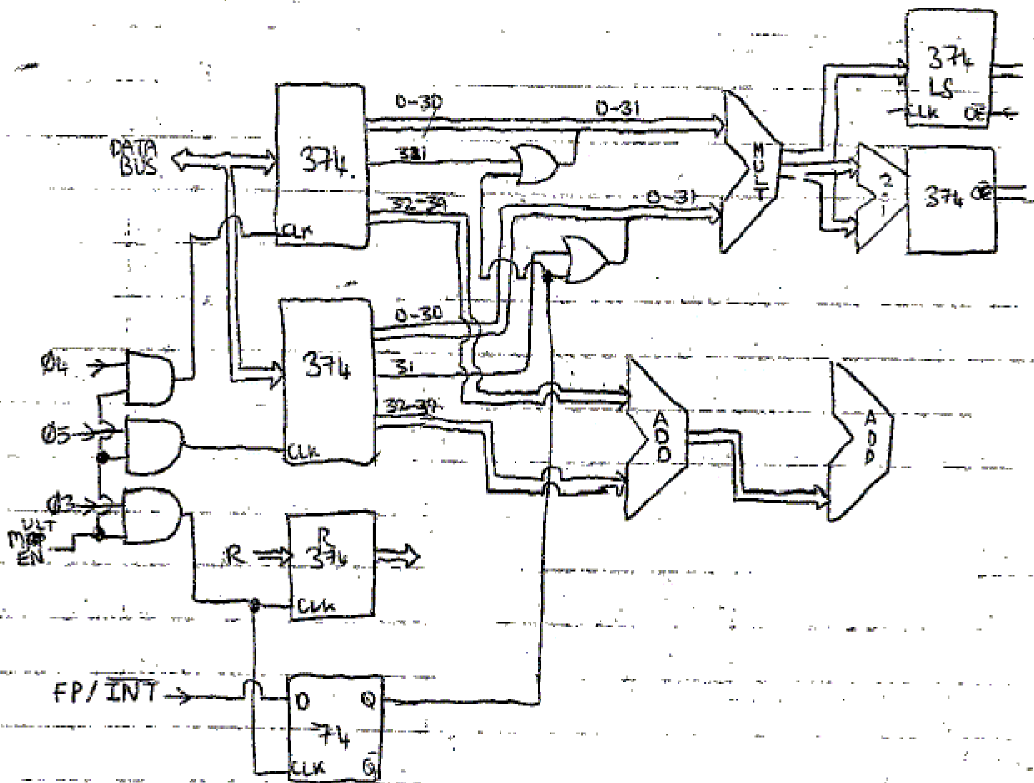
½ x ½ :
0.10000000 e0 ²
.01 e0
= .10000000 e -1
-1 = 11111111

51. First sketches for a floating point multiplier

Some first preliminary ideas for my floating point multiplication implementation. I also want this multiplier to be able to multiply integers, which is an easier task.

27/9/92. 51.





52. Multiplier

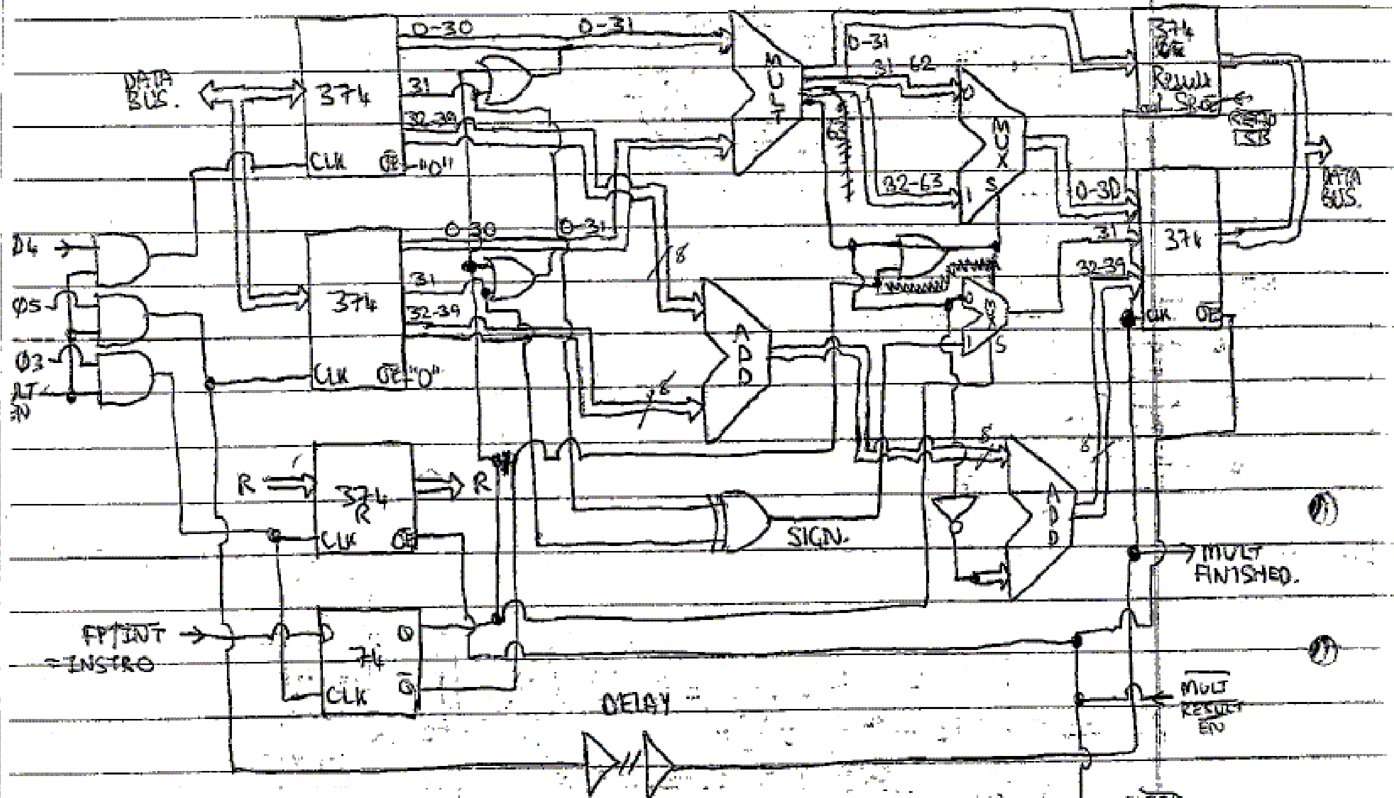
Complete design for the multiplier.

When multiplying two floating point numbers together, it isn't necessary to align them first. Just multiply them, and take the most significant bits of the result. Meanwhile, add the exponents. Post-multplication normalisation will only ever result in a shift of 1 bit, so I don't need a full barrel shifter, instead I just have a multiplexer with its inputs both connected to the multiplcation result but displaced by one bit.

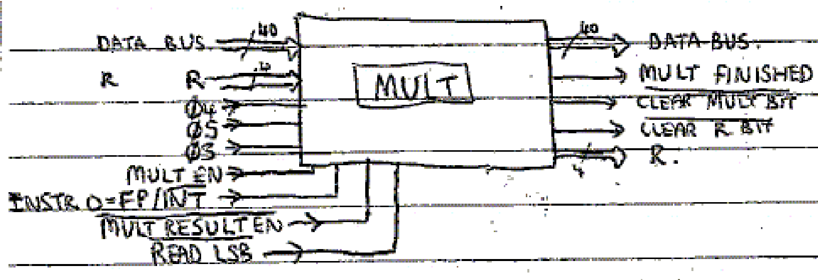
This unit can also multiply integers. In this case, the product is twice as wide as the operands. I decide that I must store this extra result word somewhere, it would be a shame to waste it. Register 15 seems like the ideal place. Previously I had specified register 15 to be the Program Counter. However, the Program Counter does not need to be accessible for read or write by the executing program, it is entirely under the control of the instruction decode related units. So I hide it from the register bank and use register 15 for the least significant word of the result. The most significant word goes to the specified result register.

The multiply circuit itself is not shown here, it's just drawn as a block with the MULT caption. I intend to use a fast parallel multiplier, I had some papers from the libary about how to build one of those, but unfortunately it requires a large number of chips. I decide not to draw it here, but just to consider it as a functional block: give it two numbers and it will return the product some time later.

I also make the controverial decision to pipeline this unit, since it has a parallel multiplier. However, I don't pipeline it in the conventional way, with a register between each processing stage and clock the results along the pipeline. Instead I use ripple-through (wave) pipelining. Just shove the inputs in one after the other, wait the right amount of time, and take the output. Believe it or not, wave-pipelined parallel multipliers have been built in practice. Whether or not I could ever get it working is debateable.



Pipeline this unit by putting in a "pipe of R-registers and permanently removing the "MULT BUSY" bit; + inserting possible latches in the exponent lines & sign bit line, & FP/$\overline{\text{INT}}$.



How will R15 be indicated busy? perhaps use INSTR1 to tell MULT if it should allow pipelining, or normal "unit busy" operation, this gives time for R15 to be read if read?

Possibly make the LSB register one of the "16", suggest R15, instead of PC being R15, since PC should be under the control of the instruction decode unit.
Should be possible to fully pipeline this unit, without additional latches inside the multiplier, etc. (Just ripple-through pipelining.

53. Instruction set once more

So the 27-th September 1992 was a heavy day. All that library research and then the design of the multiplier. The following day I decide to revisit the instruction set again. This time I summarise the whole thing on just one page.

Instructions

0: Control — bits 19–16: 0 0 0 0, remaining bits don't care, bit 0:

Bit 0	
NOP	0
HALT.	1

1: Load/store — bits 19–17: 0 0 1 | 16–13: REG. | 12 11 10 | 9–0: ←10-BIT DISP→

Bit 12		Bits 11 10	
		00	Absolute (Next 20)
Load	0	01	Push/Pop
Store.	1	10	Indexed

2: Reg Move — bits 19–17: 0 1 0 | 16–13: Source | 12–9: don't care | 8–5: Result | 4–0: don't care

3: Multiply — bits 19–17: 0 1 1 | 16–13: Source1 | 12–9: Source2 | 8–5: Result | 4–1: don't care | bit 0:

Bit 0	
Integer	0
Floating pt.	1

4: ALU — bits 19–17: 1 0 0 | 16–13: Source1 | 12–9: Source2 | 8–5: Result | 4–0: ALU-OP

5: ADD/SUB — bits 19–17: 1 0 1 | 16–13: Source1 | 12–9: Source2 | 8–5: Result | 4–1: don't care | bit 0:

Bit 0	
Add	0
Subtract	1

6: BitMan — bits 19–17: 1 1 0 | 16–13: Source | 12–9: n n n n | 8–5: Result | 4 3: n n | 2: don't care | 1 0:

n = No. of bit, 0-39.

Bits 1 0	
Test	00
Set	01
Reset	10
Invert.	11

7: Jumps — bits 19–17: 1 1 1 | 16 15 14 | 13 12 | 11–0: ←12 BIT DISPLACEMENT→

Condition	Bits 16–14	Bits 13 12	
No condition	000	00	Jump (Next 20)
Dec R13, loop>0	001	01	Branch
NC	010	10	Call (Next 20)
C	011	11	Ret
NZ.	100		
Z	101		
P	110		
N.	111		

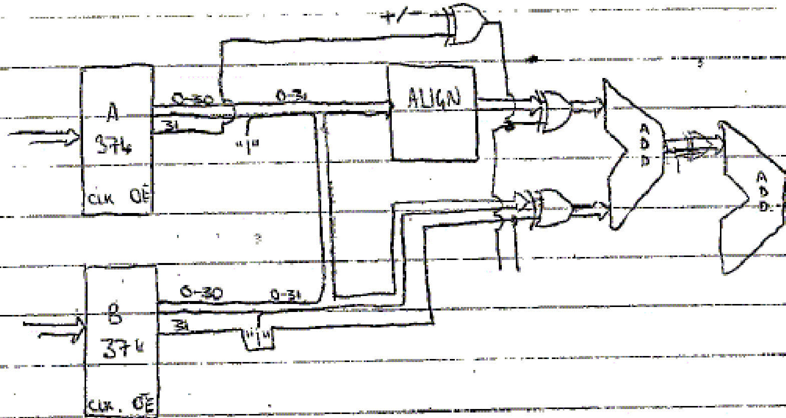
Jumps group instructions ignore 2nd field if they occur in the 1st and are tested true.

N.B. ALU OP depends on the 74LS181 functions available, does it include shifts?

54.More on Floating Point Add/Subtract methods

The next day, the 29'th September 1992 I decide I really have to do something about the design of the floating point add/subtract circuit. I start to draw the circuit but find my understanding is still lacking, or rather, that my previous understanding has been temporarily erased by my exploits with the multiplier. So I start on some more simple worked examples to get things straight in my head.

29/9/92 54



4-3   0100 = 4 $\bar{3}$ = 1100   -3 = 1101
1101   0100
10001 = 1.   10000 = 0

4-5:   0100 = 4 $\bar{5}$ = 0101 = 1010   -5 = 1011
1011   0100
01111   1010
= 0000 + 1   01110
= 0001 = 1.   = 0001

5-3:   5: 0101   3: 0011
-$\bar{3}$: 1100 +1
5-3   10001 +1   +1 reqd.
= +2

3-5   3: 0011   5: 0101
-5: 1010 +1.
3-5:   01101 +1 = 01110
= - 10010   = - 10001 + 1.
No + reqd.

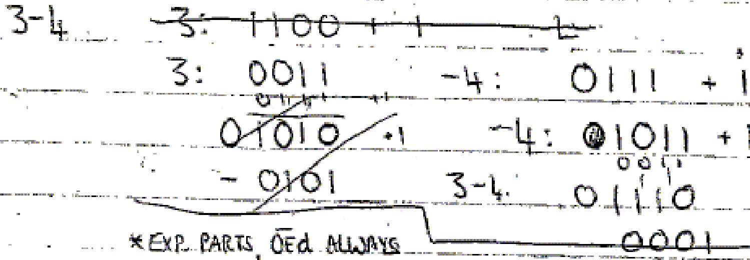
-3-5:   3: 0011   5: 0101
-3: 1100 +1   -5: 1010 +1
-3-5: 10110 +2.   = 11000
= - 01001 -1   = - 00111 +1
= - 0 1000 = - 8.

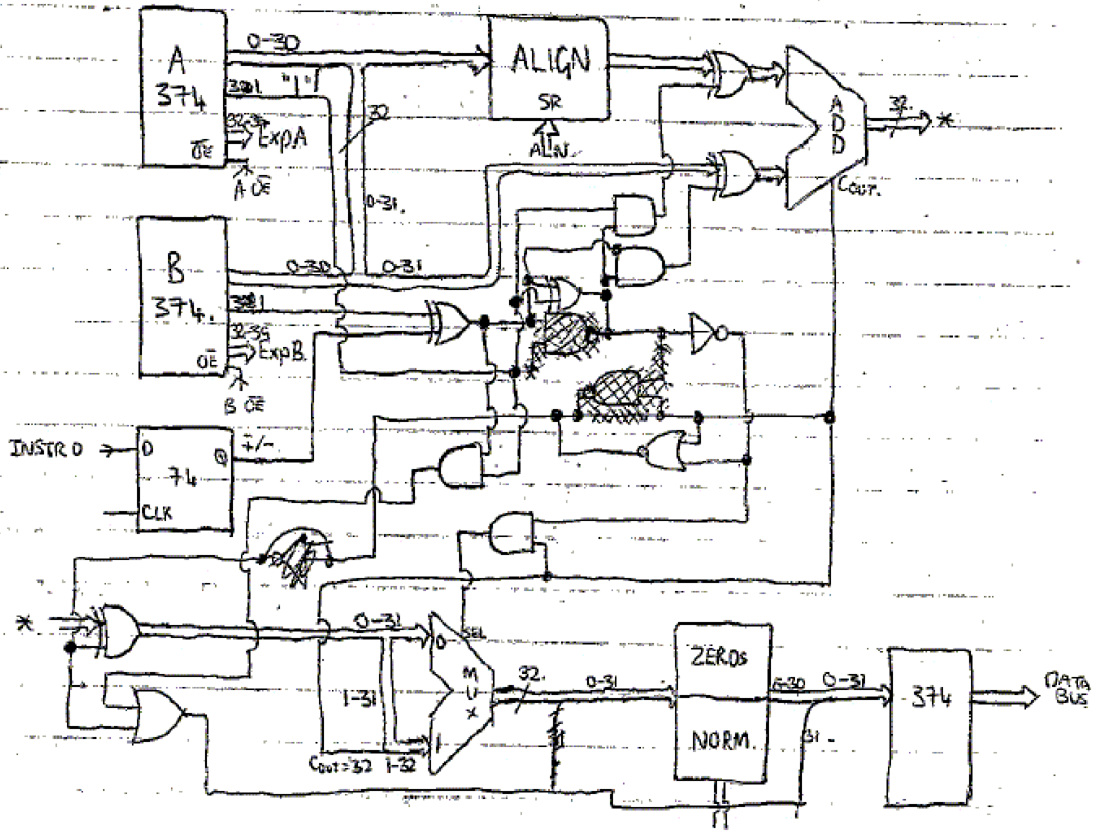
4-3   4: 0100   -3: 1100 +1
4-3   10000
+10000 +1 error!

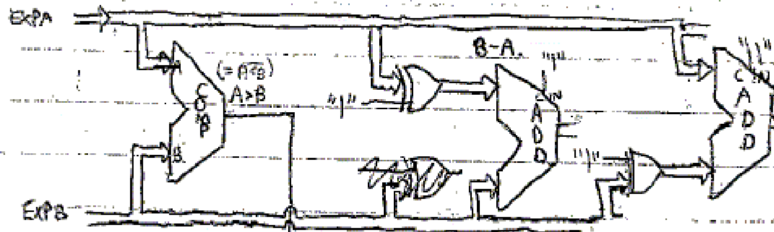
·100 e2 =   ·0100 ≈ $\frac{1}{4}$ 100 e0
·100 e+1   <   ·1000 e-1 = $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$.   1 e2
L SHIFT DEC. EXP.

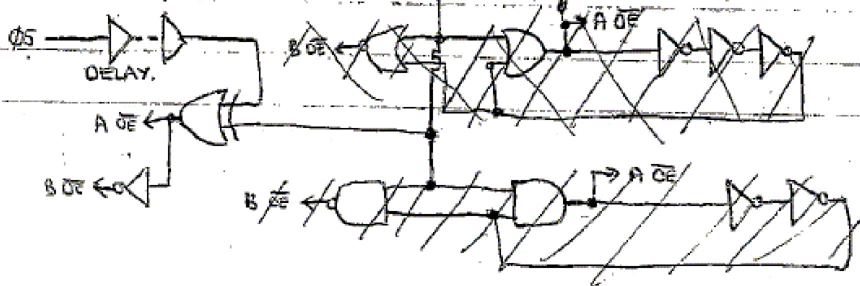
55. Floating Point Add/Subtract sketch

A first attempt at a design for the floating point add/subtract unit.



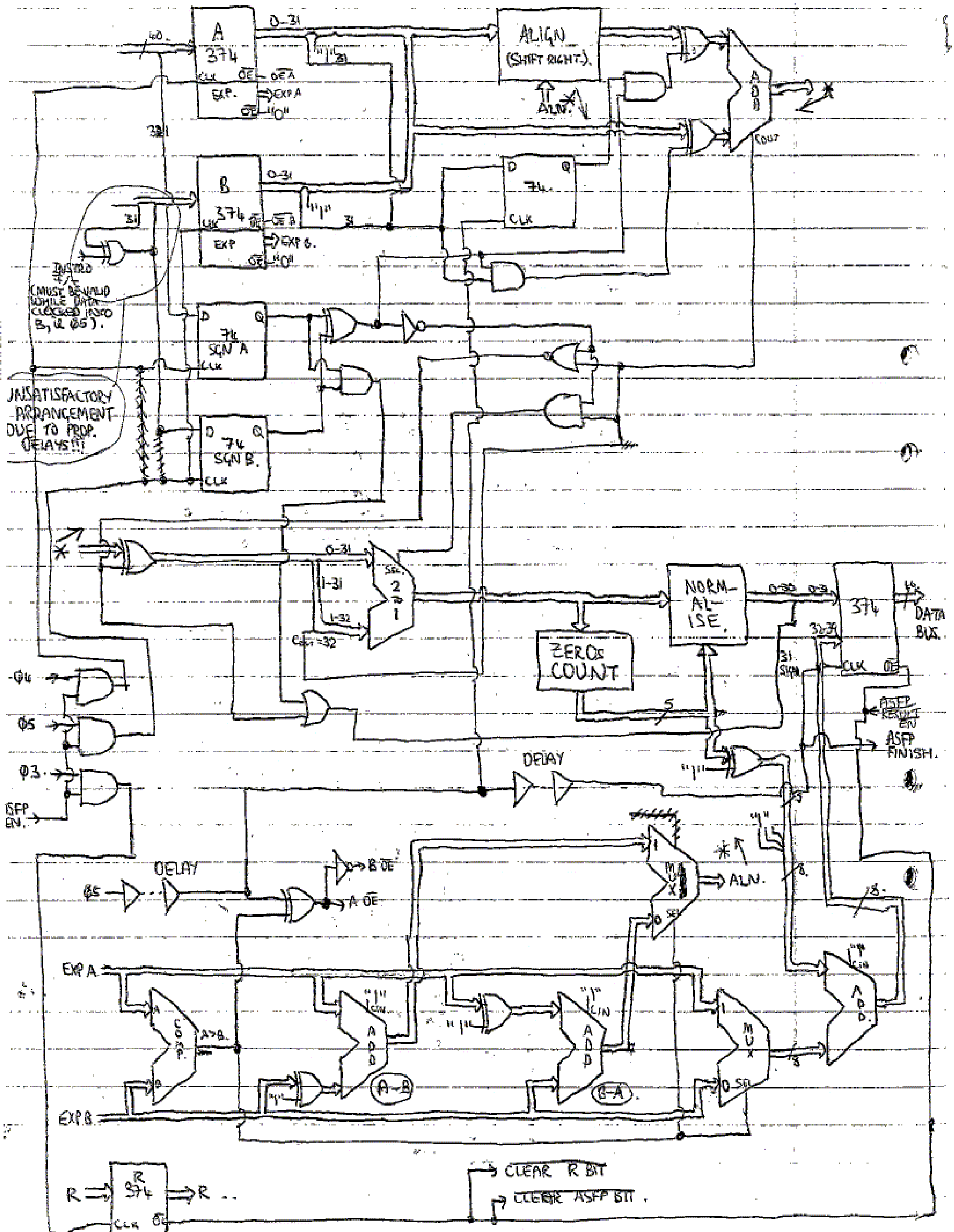






56. Final Floating Point Add/Subtract design

Finally on the 1'st October 1992 I find myself in a position to complete the design of the add/subtract unit. This diagram includes as labelled blocks the Align, Normalise and Zero-Count blocks described previously.



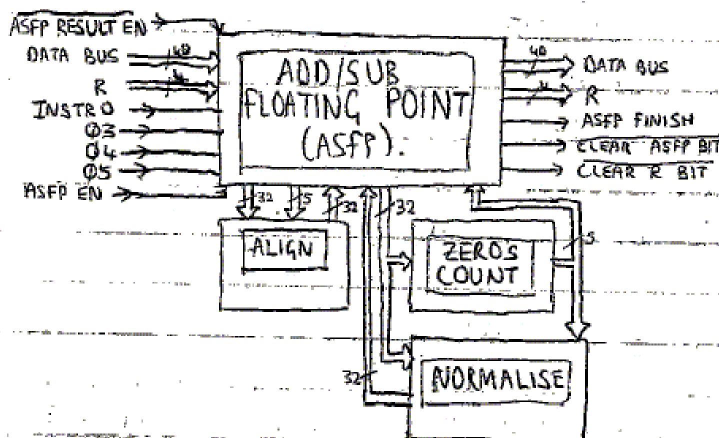
57. Add/Subtract block diagram and Questions

It's a complicated unit, this Add/Subtract, and I STILL have some questions about its operation, which I carefully write down here in order that I may return to them at some later time.

Below that I draw the Add/Sub Floating Point unit as a block diagram showing its interconnections to the Align, Normalise and Zero Count blocks.

1/10/92. 57.

* Align enable: If the required shift is 32 or more, the right shifter must give an output of zero; ie disable it; use 'EN' inputs on the 157, 153, 151s etc in the align subunit.
* +/- operation must be properly sorted out at an early stage, without causing propagation delay problems.
* Pipelining will probably need intermediate latches at strategic points!
* Do errors ever occur and if so, how do you detect them, and signify them?
* How to synchronise all the various parts of the circuit?



* In certain cases of subtraction a zero fraction part may occur; how to cope with this?

58. Another listing of the Registers

Which now also shows register 15 as the multiplier low word. Notice that I have also replaced the separate I/O registers with a single one which is used for both IN and OUT data. There are therefore now 10 general purpose registers available.

The idea of separating the input and output busses of the register file is a good one. It means that the instruction decode unit can access the output of the registers whenever it wants, without stopping the write-back of results to the register file by the units. Effectively then the bus arbitration circuit is always in the result phase, which should dramatically speed up the CPU.

2/10/92 58.

Register list

R0 Zeros register.
R1 A
R2 B
R3 C
R4 D
R5 E
R6 F
R7 G
R8 H
R9 I
R10 J
R11 IN/OUT register.
R12 Index register (20
R13 Loop Counter (20)
R14 Stack pointer (20)
R15. Multiplier low word; Read only.

How will R15 be indicated busy? See Page 52

How 'bout using separate input & output buses on the registers? Eliminating conflicts due to simultaneous unit and Instr. decode accesses.

59. IOU Redesign

So, 2'nd October 1992 and this is the last page of my asynchronous TTL CPU design. I add a control and status register. I am unsure if there were any other changes since the last one (page 39).

2/10/92 59.

IOU Redesign.