

TP208

STATEMENT OF PURPOSE

THIS NOTEBOOK IS MEANT TO BE AN ALL-PURPOSE DIARY OF DESIGN IDEAS AND PHILOSOPHIES. SOME ATTEMPT WILL BE MADE TO RESTRICT THE TOPIC TO ~~SOME~~ DEDICATED INSTRUMENTATION MICROCOMPUTERS, BUT NO PROMISE IS MADE THAT THIS RESTRICTION SHALL BE RELIGIOUSLY ADHERED TO.

Ruehl's

770208 CAN DESIGN ECONOMIES BE REALIZED
THRU MICRO-PROGRAM CONTROL OF ANALOG
AND DIGITAL PERIPHERAL/FUNCTIONAL
CIRCUITS IN A MICRO-PROCESSOR BASED
INSTRUMENT SUCH AS TIMS?

HISTORICALLY, IN THE DEVELOPMENT OF THE HP4942A,
HP 4942B, AND HP 4943A TRANSMISSION IMPAIRMENT
MEASURING SETS (TIMS), THE INSTRUMENT'S
ANALOG CIRCUITRY HAS BEEN REGARDED (FROM
THE POINT OF VIEW OF THE μ COMPUTER HARDWARE
AND THE SYSTEM SOFTWARE) AS BEING THE
TIMS EQUIVALENT OF COMPUTER PERIPHERAL
EQUIPMENT (TAPE-PUNCH, LINE PRINTER, ETC.). TO
THE SOFTWARE DESIGNER, THIS HAS MEANT
THAT CONTROL OF THE ANALOG CIRCUITS HAS
BEEN ACCOMPLISHED THRU SOFTWARE DRIVER
ROUTINES. 8

FROM A HARDWARE POINT OF VIEW, EACH
PERIPHERAL DRIVER IS ACCESSED THRU AN I/O
PORT (TYPICALLY AN 8-BIT LATCH) CONNECTED
TO THE DATA BUS.

CONCEPTUALLY, FROM A HARDWARE BLOCK DIAGRAM
POINT OF VIEW, THIS ARRANGEMENT LOOKS SOMETHING
LIKE THE DRAWING ON PAGE 7. OMITTED FROM
THIS DRAWING ARE VARIOUS BUFFERS, INVERTERS,
BUS-DRIVERS, ETC AS WELL AS THE μ PROCESSOR
CONTROL LINES.

TO IMPLEMENT THE VARIOUS CONTROL FUNCTIONS
REQUIRED BY HIS DESIGN, THE CIRCUIT DESIGNER
ADDS A LATCH TO HIS BOARD (TYPICALLY AN
8-bit latch; SELDOM LARGER THAN 16 BITS; OCCASSIONALLY
ONLY 4 bits). THE DESIGNER THEN MAKES ARRANGEMENTS
WITH THE ENGINEER RESPONSIBLE FOR THE MPU
BOARD TO PROVIDE AN ENABLE SIGNAL FOR HIS
NEW I/O PORT.

THE TYPE OF FUNCTIONS TYPICALLY PERFORMED
BY THE HARDWARE LATCH INCLUDE:

1. SELECTING AND CONNECTING ONE ^{OF MANY} DIGITAL
SIGNAL PATHS TO ANOTHER
2. SELECTING AND CONNECTING ONE OF
MANY ANALOG SIGNAL PATHS TO ANOTHER
(see page 8)

RBWells

770208

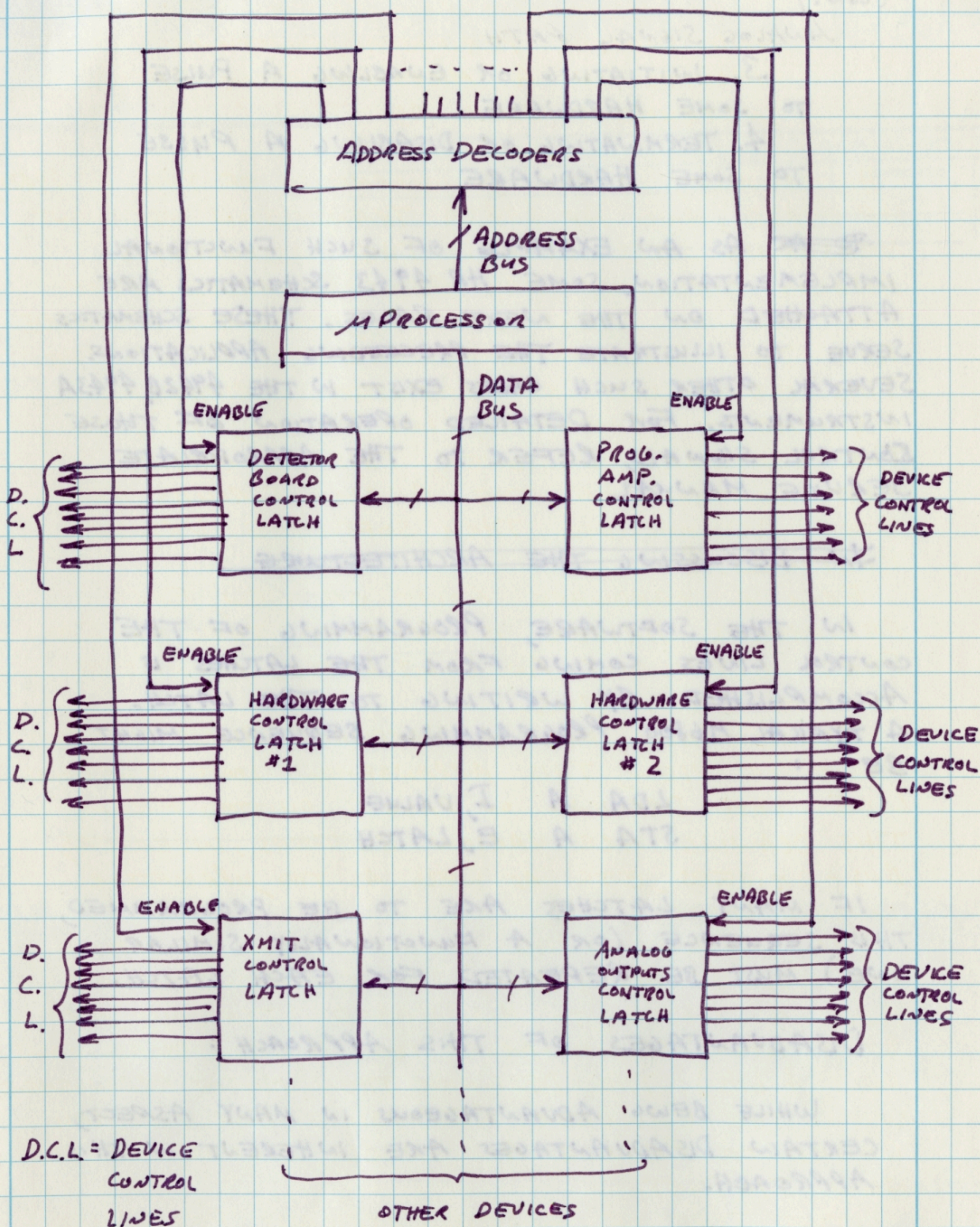


FIGURE 770208.1 CONTROL CONFIGURATION OF THE WRITE-ONLY I/O PORTS IN THE HP 4943

RBWells

(CONT)

ANALOG SIGNAL PATH

3. INITIATING OR ENABLING A PULSE TO SOME HARDWARE
4. TERMINATING OR DISABLING A PULSE TO SOME HARDWARE

~~TO~~ AS AN EXAMPLE OF SUCH FUNCTIONAL IMPLEMENTATION, SOME HP 4943 SCHEMATICS ARE ATTACHED ON THE NEXT PAGES. THESE SCHEMATICS SERVE TO ILLUSTRATE THE PRECEEDING APPLICATIONS. SEVERAL OTHER SUCH CASES EXIST IN THE 4942B, 4943A INSTRUMENTS. FOR DETAILED OPERATION OF THOSE CONTROL SIGNALS, REFER TO THE APPROPRIATE SERVICE MANUAL

IN DISCUSSING THE ARCHITECTURE

IN THE SOFTWARE, PROGRAMMING OF THE CONTROL LINES COMING FROM THE LATCHES IS ACCOMPLISHED BY WRITING TO THE LATCH. A TYPICAL M6800 PROGRAMMING SEQUENCE MIGHT BE :

```
LDA A I, VALUE  
STA A E, LATCH
```

IF MANY LATCHES ARE TO BE PROGRAMMED, THIS SEQUENCE (OR A FUNCTIONALLY SIMILAR ONE) MUST BE REPEATED FOR EACH LATCH.

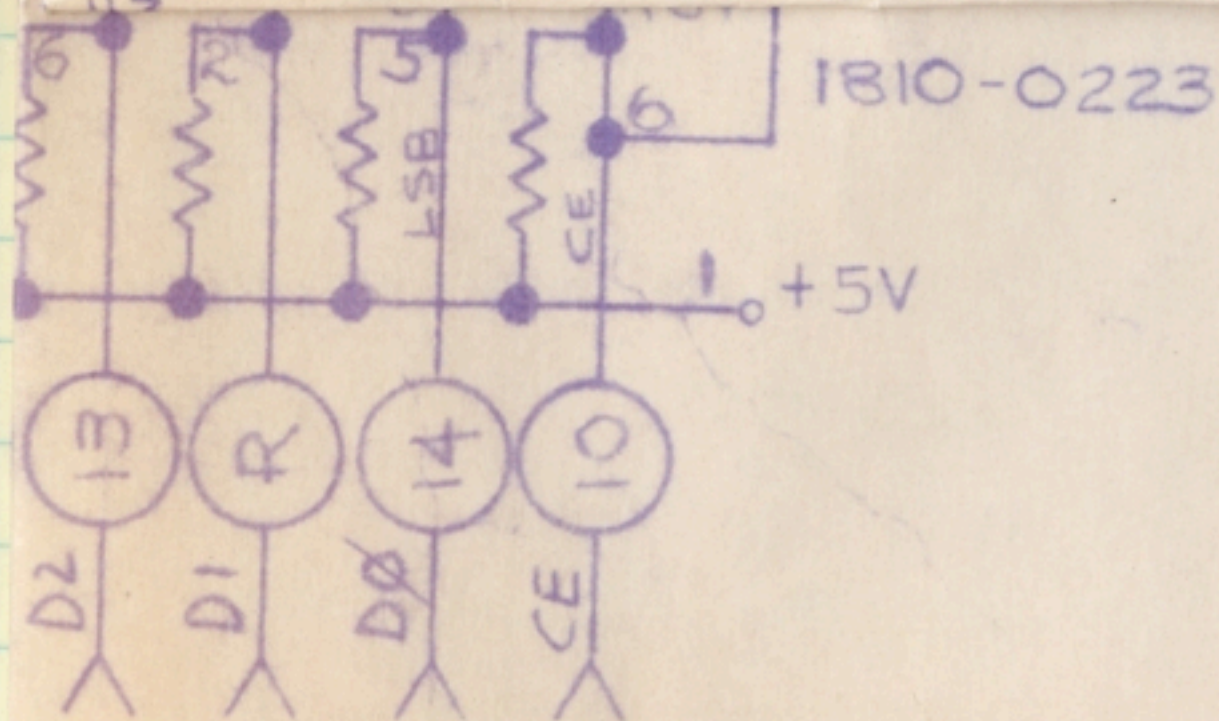
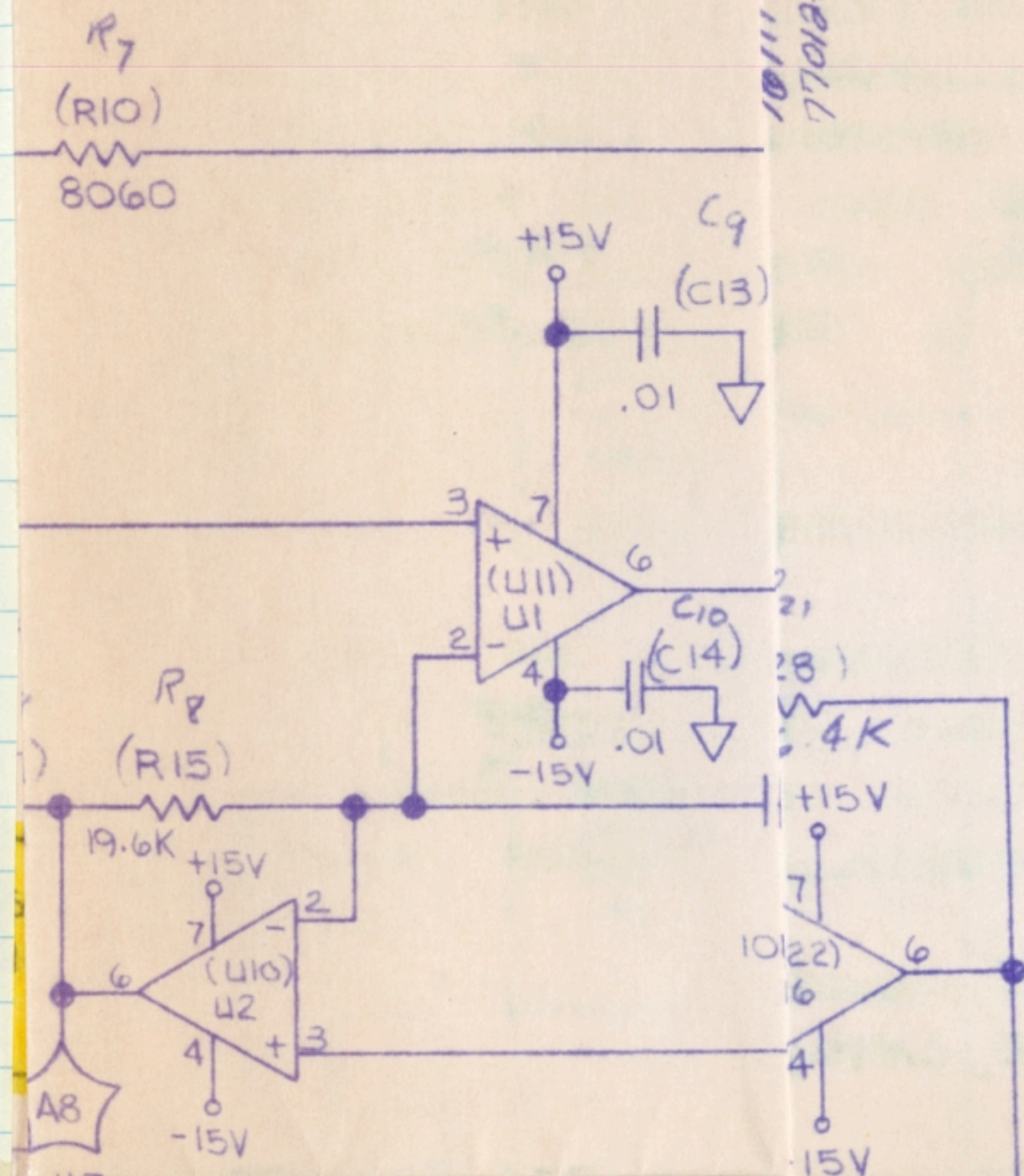
DISADVANTAGES OF THIS APPROACH :

WHILE BEING ADVANTAGEOUS IN MANY ASPECT, CERTAIN DISADVANTAGES ARE INHERENT IN THIS APPROACH.

1. THE LATCHES TAKE UP BOARD SPACE WHICH OTHERWISE COULD HAVE BEEN ALLOCATED ELSEWHERE ;
2. EXTENSIVE ADDRESS DECODING IS REQUIRED; IF AN ATTEMPT IS MADE TO KEEP THE PARTS COUNT REQUIRED DOWN, (see pg 13)

RICKLETS

770208



ARTO
RANGE

D-04943-60017-		
REVISIONS	APPROVED	DATE


DESCRIPTION

MAT'L PART NO.

MAT'L DWG NO.

MAT'L SPEC.

A17 NOTCH AND
PROG AMP

HEWLETT  PACKARD

EXT ASSEMBLY

PART NUMBER

4943-60017

FINISH

NONE

SCALE

NONE

D-04943-60017-

SHEET 1 OF 1

Kickwell

770611

Operation	Mnemonic OP Code	Machine Code	Function
Memory Reference Instructions			
1. Load Accumulator	LAC	000001	$ACC \leftarrow c(EA)$
2. Load X Register	LRX	000010	$RX \leftarrow c(EA)$
3. Load L Register	LRL	000011	$RL \leftarrow c(EA)$
4. Load I/O Register	LIO	000100	$IO \leftarrow c(EA)$
5. Load Immediate Acc.	LIAC	000101	$ACC \leftarrow EA$
6. Load Immediate L Reg.	LIRL	000110	$RL \leftarrow EA$
7. Load Immediate Index Reg.1	LIX1	000111	$IX1 \leftarrow EA$
8. Load Immediate Index Reg.2	LIX2	001000	$IX2 \leftarrow EA$
9. Store Accumulator	STAC	001001	$c(EA) \leftarrow ACC$
10. Store L Register	STRL	001010	$c(EA) \leftarrow RL$
11. Store I/O Register	STIO	001011	$c(EA) \leftarrow IO$
12. Store Index Reg.1	STIX1	001100	$c(EA) \leftarrow IX1$
13. Store Index Reg.2	STIX2	001101	$c(EA) \leftarrow IX2$
Arithmetic Instructions			
1. Add Accumulator	ADAC	001110	$ACC \leftarrow ACC + c(EA)$
2. Subtract Accumulator	SUAC	001111	$ACC \leftarrow ACC - c(EA)$
3. Multiply and Accumulate	MAC	010000	$ACC \leftarrow RX \times c(EA) + ACC$
4. Multiply	MPY	010001	$ACC \leftarrow RX \times c(EA)$
5. Shift Left Acc.	SLAC	010010	$ACC \leftarrow ACC \times 2$
6. Shift Right Acc.	SRAC	010011	$ACC \leftarrow ACC \times 2^{-1}$
7. Shift Left X Reg.	SLRX	010100	$RX \leftarrow RX \times 2$
8. Shift Right X Reg.	SRRX	010101	$RX \leftarrow RX \times 2^{-1}$
Branch Instructions			
1. Jump Unconditionally	JMP	010110	$NIR \leftarrow EA$
2. Branch on Zero in L Reg.	BZRL	010111	$NIR \leftarrow EA \text{ if } RL=0$
4. Branch on Zero in Acc.	BZAC	011000	$NIR \leftarrow EA \text{ if } ACC=0$
5. Branch on Positive in Acc.	BRPA	011001	$NIR \leftarrow EA \text{ if } ACC \geq 0$
Logical Instructions			
1. AND Accumulator	ANDA	011010	$ACC \leftarrow ACC \cap c(EA)$
2. OR Accumulator	ORAC	011011	$ACC \leftarrow ACC \cup c(EA)$
3. Exclusive Or Accumulator	EXAC	011100	$ACC \leftarrow ACC \oplus c(EA)$
Miscellaneous Instructions			
1. Increment L Register	IRL	011101	$RL \leftarrow RL + EA$
2. Increment Index Reg.1	IIX1	011110	$IX1 \leftarrow IX1 + EA$
3. Increment Index Reg.2	IIX2	011111	$IX2 \leftarrow IX2 + EA$
4. Compare Index Reg.2	CIX2	100000	$CP = 1 \text{ if } IX2 > c(EA)$
5. Conditional Subtract IX2	CSX2	100001	$IX2 \leftarrow IX2 - c(EA) \text{ if } CP=1$
6. No Operation	NOP	100010	Do nothing
7. Halt	HLT	100011	Stop

Note: c(EA) denotes the contents of the data memory at the effective address.

Table 4.8 The Instruction Set of DISP.

Some of the features of their design are :

1. A distinction is made between instruction memory and data memory. This has many advantages including :

A) Data physical length is independent of instruction physical length

B) This division lends itself to storing the user program in ROM

Rich Wells

770611

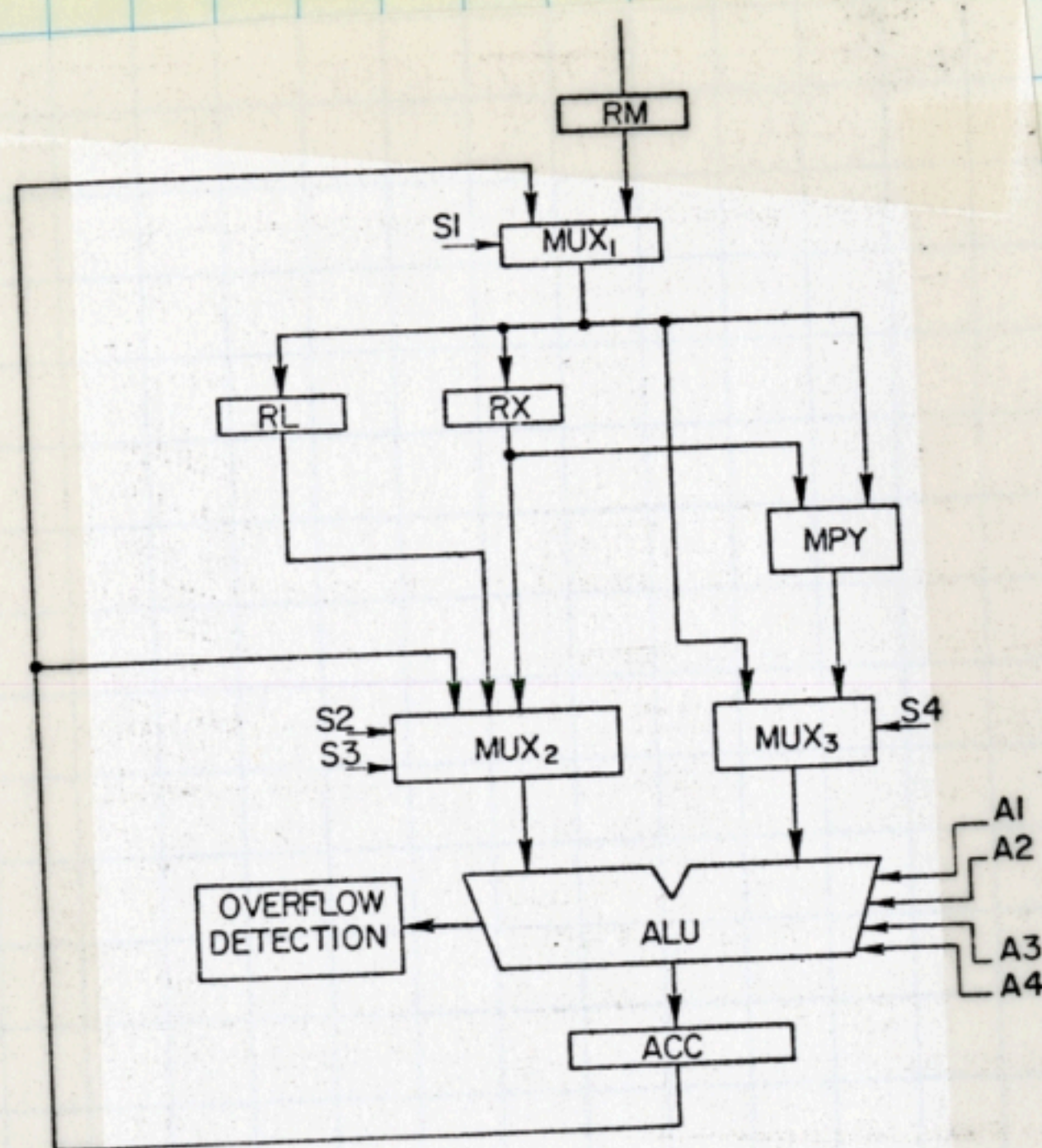


Figure 4.17 The arithmetic unit.

2. A provision is made for loading and storing directly from the I/O Register into memory. This facilitates data acquisition during real time operation. At the same time, it in no way reduces the I/O capability of the machine since memory-mapped I/O may be utilized.

3. Two ALU units exist: one is for data and the other is for instructions. This facilitates pipelining which in turn increases throughput.

4. Special instructions exist (see Table 4.8 for high-level op codes) which facilitate computation of operations of the form

$$y = \sum_{n=1}^N a_n x_n$$

5. The multiplicity of index registers facilitates the calculation of convolutional sums.

A sample program was written by Peled & Liu for performing non-recursive filtering. This program and some of their comments are attached:
Rick Wells

770611

• NONRECURSIVE DIGITAL FILTER PROGRAM •
 • THE NUMBER OF TAPS, N, IS IN LOCATION 99 •
 • THE FILTER COEFFICIENTS ARE IN LOCATIONS 200 - 200+N-1 •
 • THE DATA POINTS ARE IN LOCATIONS 100 - 100+N-1 •
 • EACH NEW DATA POINT REPLACES THE N-1 DATA POINT (FIFO) •

000	LIX2	0	100	IX2 ← 100	Initialize pointers to the top and bottom of the input data list.
001	STIX2	0	098	c(98) ← 100	
002	LAC	0	098	ACC ← c(98)	
003	ADAC	0	099	ACC ← ACC + c(99)	
004	STAC	0	098	c(98) ← ACC	Initialize the loop and index registers and accumulator.
005	LRL	0	099	RL ← c(99)	
006	LIX1	0	000	IX1 ← 0	
007	LIAC	0	000	ACC ← 0	
008	LRX	1	200	RX ← c(200+IX1)	Compute an output and advance pointers modulo the number of taps.
009	MAC	2	000	ACC ← RX × c(IX2) + ACC	
010	IRL	0	-01	RL ← RL - 1	
011	JMPL	0	017	Goto 017 if RL=0	
012	IIX1	0	001	IX1 ← IX1 + 1	
013	IIX2	0	001	IX2 ← IX2 + 1	
014	CIX2	0	098	CP=1 if IX2>c(98)	
015	CSX2	0	099	IX2 ← IX2 - c(99) if CP=1	
016	JMP	0	008	Goto 008	Store output sample, load new input sample and restart if limit is not reached.
017	STAC	0	097	c(97) ← ACC	
018	STIO	2	000	c(IX2) ← IO	
019	LIO	0	097	IO ← c(97)	
020	LIAC	0	-01	ACC ← -1	
021	ADAC	0	095	ACC ← c(95) - 1	
022	STAC	0	095	c(95) ← ACC	
023	BRPA	0	005	Goto 005 if ACC>0	
024	HLT				

Table 4.9 A Nonrecursive Filter Program on DISP.

ADDRESS	DATA MEMORY CONTENTS				
	5	4	3	2	1
95					
96					
97					
98		y ₃	y ₄	y ₅	y ₆
99	104	104	104	104	104
100	4	4	4	4	4
101	x ₃	x ₃	x ₃	x ₃	x ₇
102	x ₂	x ₂	x ₂	x ₆	x ₇
103	x ₁	x ₁	x ₅	x ₅	x ₅
•	x ₀	x ₄	x ₄	x ₄	x ₄
•					
200	a ₀	a ₀	a ₀	a ₀	a ₀
201	a ₁	a ₁	a ₁	a ₁	a ₁
202	a ₂	a ₂	a ₂	a ₂	a ₂
203	a ₃	a ₃	a ₃	a ₃	a ₃
Loop	0	1	2	3	4

Table 4.10 Data Memory Contents During the Execution of the Nonrecursive Filter Program, with N = 4.

Rak Wells

770611

Table 4.9 contains the listing of the DISP program that will perform the nonrecursive filtering as outlined above. The reader is strongly urged to go through the program carefully, executing each command manually to understand fully its operation. While doing this he will realize the importance of the index registers and appreciate the flexibility they provide in manipulating the addresses, so as to make the memory look like a shift register. Table 4.10 shows the changes in memory contents for this example, in the simple case of $N=4$, as the computation progresses. As we see x_n are maintained in a circular list. To compute the correct address, instructions CIX2 and CSX2 effectively allow us to increment the index register modulo an arbitrary number. This is again an example of the matching required between the architecture and the algorithms.

We see from Table 4.9 that a total of 25 instructions are required for this program. To evaluate the execution time of the program we consider a concrete example, where $L=1000$, and $N=20$. Therefore instructions 000 to 004 and 024 are executed once, instructions 005 to 007 and 017 to 023 are executed a 1000 times, and instructions 008 to 016 are executed 20×1000 times each, and hence a total of 190,006 instructions are executed, or an average of about 190 per output. The actual execution time depends on the time required to execute one instruction (if all instructions take the same time).

A total of 190 instructions per output sample seems a large number, considering that all we do is multiply 20 numbers and add them up. A closer examination of the program shows that indeed we only do an average of 40 instructions to do the actual arithmetic, and the other 150 instructions are required to take care of the indexing for looping and memory addressing.

At this point we notice that we could significantly improve the running time of this program if we had some additional instructions in the DISP repertoire. As an example, suppose we add the instruction LRXL whose effect is to load the RX register from memory and decrement the L register by -1. This can obviously be done at the same time, since while we load RX from memory the ALU or the AU can decrement the L register. In this case we could replace instruction 008 by LRXL 1200 and eliminate instruction 010 IRL 0 -1. This will reduce the average number of instructions per output to 170, that is, a more than 10% reduction. We note however that this instruction (LRXL) is somewhat restrictive, in that it does not allow us to specify the size of the increment (it is taken as -1), which makes this program less flexible.

In the excerpt above, L is the "window length" of the data stream $[and, therefore, the number of outputs]$ and N is the number of taps in the filter. Assuming an average instruction execution rate of 250 nsec/instruction, their machine (in principle) is capable of doing real time filtering on a data stream sampled at a 10 KHz rate.

Of particular importance is to note how important the "overhead" tasks of indexing, looping and memory management are. These tasks account for

Rick Wells

710611

79% of the instructions executed. The remaining 21% constitute the "real work" desired by the programmer. My previous commentary on the various addressing modes and use of index registers is seen ~~to~~ in this light to represent an architectural issue of prime importance.

Worth noting from the DISP instruction set is the "Multiply and Accumulate" instruction. In terms of real work, this instruction is seen to be of great importance and the work on pages 35-37 should perhaps be modified to include a MAC operation [which, for the case of the previously discussed architecture, would accumulate the result at the top of stack].

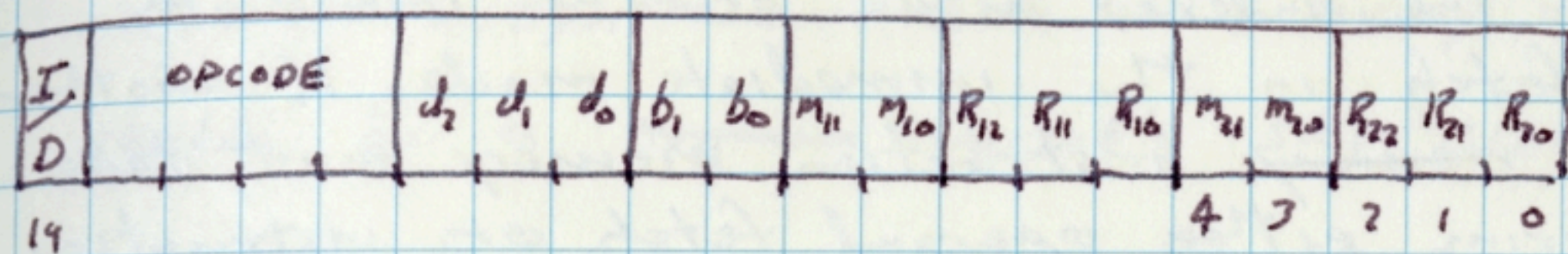
A prime feature of the Peled-Liu architecture is the fact that all instructions are contained in a single word of instruction memory. This is a major reason they achieve the speed of operation they do since the required number of memory accesses is reduced. In my prior discussion of addressing, such single-reference operation is achieved with the auto-increment and auto-decrement modes. However, indexed and immediate addressing modes require multiple memory references, and it may be deemed desirable to allow the physical length of the instruction word to grow to accommodate the memory reference address or immediate value with the instruction fetch. The dual addressing nature of the machine makes one hesitant to do so, however, because provisions must be made to accommodate the worst case required word length. For a dual operand machine, this ~~word~~ ^{length} extra word length is twice that of a single operand machine. Also, including the immediate value in the instruction word places an implied dependency of the data physical length on the instruction physical length. This is undesirable.

Note that a wealth of index registers for auto-increment & auto-decrement decreases the need for having an index + offset feature.

Rick Wells

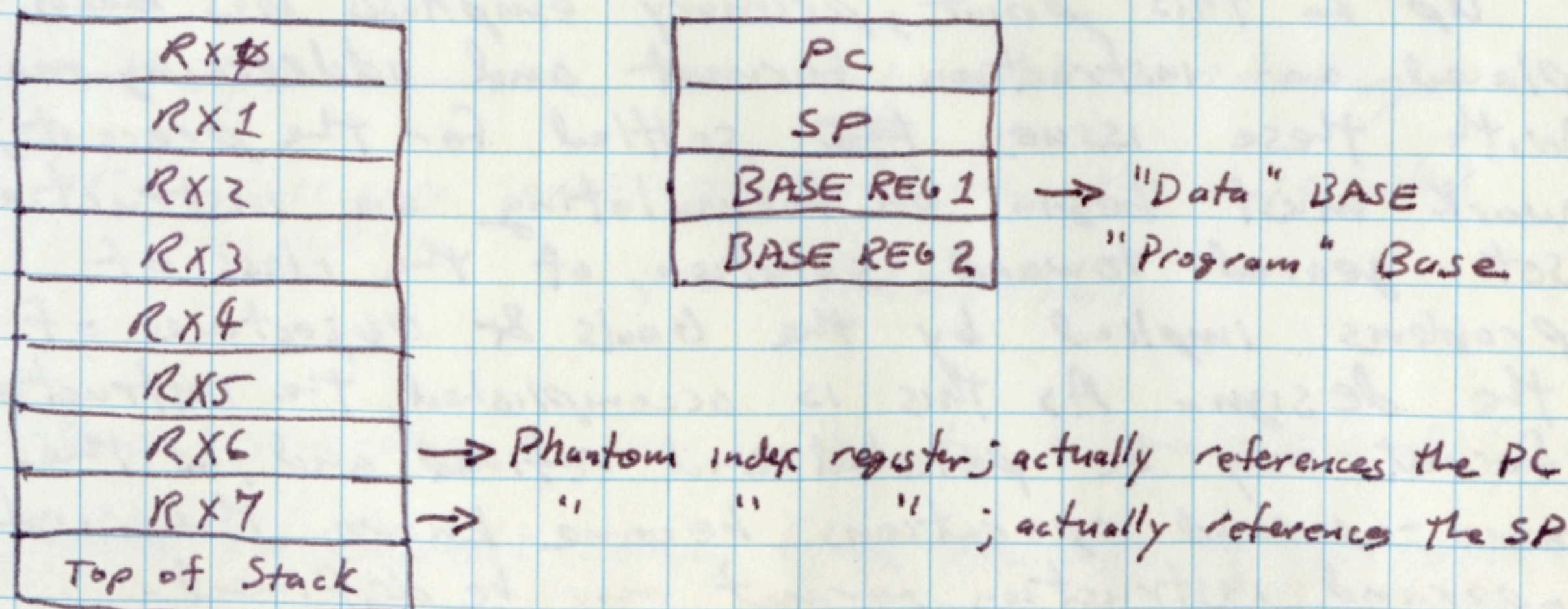
770611

Allowing the instruction word to grow by four bits does increase the power of the machine substantially. Consider:



d_2	d_1	d_0	DESTINATION	b_1	b_0	BASE
0	0	0	Bit Bucket	0	0	PC
0	0	1	OPERAND 1	0	1	BASE REG. 1
0	1	0	OPERAND 2	1	0	BASE REG 2
0	1	1	TOP OF STACK	1	1	SP
1	0	0	PC BASE REG 1			
1	0	1	SP BASE REG 2			
1	1	0	BASE REG 1 PC			
1	1	1	BASE REG 2 SP			
				m_{11}	m_{10}	MODE
				0	0	Autoincrement
				0	1	Register
				1	0	Autodecrement
				1	1	Indexed

Eight registers are now available for indexing, and program jumps are possible by a MOV to the PC. Similarly, a MOV to the SP changes stacks. Whenever a MOV destination is not operand 2, the MOV is actually a member of the single operand class. Therefore, in this case instruction bits $i < 4 : 0$ could be used as an offset value (if $m_1 \Rightarrow$ indexing), an immediate value by which to autodecrement or autoincrement R_1 , or an immediate value by which to load R_1 . The structure looks like:



Rick Wells