M E M O

To: CSL/SSL

Date: September 13, 1974

From: Ben Wegbreit

Location: Palo Alto

Subject: Alto Virtual Memory Update

Organization: PARC/CSL

File: <VANJEPMOND>BWALTOUPDATE.PUB;35

1. SCOPE

This is an incremental update of "Alto Virtual Memory Proposal", 6/25/74, and should be read with that in hand. It discusses refinements to that design, gives additional detail, and specifies hardware at the register-transfer level. A final (draft) memo will be written shortly consolidating material on the virtual memory. Comments please.

2. EXPERIMENTAL DATA ON ADDRESS TRANSLATION FAULT RATES

(2.1) Sources:

For BCPL, see H. Sturgis, "Some Statistics for Virtual Memory Fans (Part 3)" 7/3/74. For Mesa, see "The Implementation of Mesa on Alto", 8/21/74. L.P. Deutsch is currently collecting comparable data for Bytelisp.

(2.2) Summary:

(a)	BCPL	compiler	fault rate = 7.0%
(b)	Mesa	compiler	fault rate = 5.6%
		formatter	fault rate = 3.0%

3. ADDRESSING MODES AND THEIR TRANSLATION

(3.1) Definitions:

VL = 16-bit page # in virtual address space, of the low page of a page group.

CL = 8-bit page # in core, of the low page of a page group.

CH = 8-bit page # in core, of the high page of a page group.

PA = a 24-bit full virtual address (within the page group).

(3.2) Derived quantities:

P = PA + CL·2↑8 - VL·2↑8 (16-bit core address corresponding to PA)
Q = CL - VL (16-bit page #)

(3.3) Addressing modes (c.f. Figure 1.)

(a) D - supplies a 16-bit displacement from PA

(b) E - supplies a 16-bit displacement from VL

(c) F - supplies a 24-bit virtual address purported to be within a page group.

Note: all displacements are positive increments.

(3.4) Translation:

(Bit numbering is from left to right starting at 0 assuming 24-bit quantities.)

(a) the core address corresponding to D is D_c = D+P provided that $CL \leq D_c[8:15] \leq CH$ and that $D_c[0:7]=0$

(b) the core address corresponding to E is E_c = E+CL·2↑8 provided that $E_c[8:15] \leq CH$

(c) the core address corresponding to F is F_c = F+Q·2↑8 provided that $CL \leq F_c[8:15] \leq CH$ and $F_c[0:7]=0$

4. BASE REGISTERS

(4.1) Number of:

The choice is between 8 and 16.

Mesa requires a minimum of 6--using the 2-cache scheme (p. 2 of "Mesa on Alto"). Bytelisp probably requires fewer. Hence we could get by with 8.

However, it may eventually be possible for the Mesa compiler to generate code which uses the base registers more cleverly for user-generated references (e.g. array processing). This would require 4 to 6 rather than 2 registers for pointers to computed quantities. This would raise the total to around 8 or 10.

Conclusion: 8 is enough for current needs. Planned improvements may be able to use more. Since it costs relatively little hardware to do so, we should provide 16.

(4.2) Structure:

Each base register consists of a collection of 4 registers and 3 protection bits.

CL - an 8-bit low core page #
CH - an 8-bit high core page #
P - a 16-bit core address (PA+CL·2↑8-VL·2↑8)

Q - a 16-bit page # (CL·2↑8-VL·2↑8)
R - read permitted bit
W - write permitted bit
E - execute permitted bit

Total # of bits per base register = 51.

5. MANIPULATION OF BASE REGISTERS

(This section merely specifies at the functional level operations to be carried out. Encoding of these functions into fields of the microinstructions has not yet been worked out in detail.)

(5.1) Load Base register from arithmetic unit:

Load P(B):	P(B)←ALU output
Load Q(B):	Q(B)←ALU output
Load Bounds(B):	CH(B),CL(B)←ALU output
Load Protection(B):	R,W,E←ALU output[12:14]

(5.2) Recover computed address:

Recover	BUS←MAR

(5.3) Store base register (analogous to 5.1):

Store P(B):	BUS←P(B)
Store Q(B):	BUS←Q(B)
Store Bounds(B):	BUS←CH(B),CL(B)
Store Protection(B):	BUS[12:14]←R,W,E and clear BUS[0:11] and BUS[15]

Note: As the base registers can be changed only by loading, these last four instructions are for hardware debugging only.

i.e. a long pointer is predigested and mapped when loaded (at target machine level). The munched image in the basereg is never saved/restored.

(5.4) Selecting a Base Register:

All the above operations act on a base register B. This is specified as the logical OR of two quantities:

a set of fields of the microinstruction
the contents of a special loadable SB register if SB is selected; otherwise 0

Loading and storing SB requires two more operations:

Load SB:	SB←ALU output[12:15]
Store SB:	BUS[12:15]←SB

6. BASE REGISTER OPERATION

(6.1) Address Translation Hardware

The memory interface board contains the following components used in address translation (c.f. Figure 2):

(a) a 16X52 memory (which holds P,Q,CH,CL,R,W,E)
(b) a 4-bit Base number register - B
(c) a 16-bit Memory Interface Register - MIR
(d) an 8-bit Word Number Register - WNR
(e) a 16-bit split adder
(f) a 16-bit Memory Address Register - MAR
(g) 3 8-bit comparitors
(h) a PROM hasher

The adder is split in that it will either (1) carry 0 into bit 15 and carry out of bit 8 into bit 7 or (2) it will carry 0 into bit 8 or carry out of bit 0 into bit 15. Hence, it can either (1) perform normal addition or (2) add two numbers with their halves swapped, producing a number with its halves swapped. The second option is useful in F-mode. We denote this form of addition as A $ B.

(6.2) Address Input Format:

Addresses are supplied to the memory interface board from the ALU output as follows:

(a) D-mode - one word

(b) E-mode - one word

(c) F-mode - word number 1 whose high byte is ignored and whose low byte is the first byte of the page number, followed by word number 2 whose high byte is the second byte of the page number and whose low byte is the word number in the page. These two words are supplied on two consecutive minor cycles.

(6.3) Translation in each addressing mode:

(a) D-mode

MIR←ALU output
MAR←MIR+P(B) & fault if overflow
start memory; test CL(B)≤MAR[0:7]≤CH[B] and fault if not

(b) E-mode

same as D-mode except that {CH(B),zeros} i.e. CL·2^8 is used in place of P(B)

(c) F-mode

MIR[8:15]←ALUout[8:15] (first byte of page #)

MIR[0:7]←ALUout[0:7]; WNR←ALUout[8:15]

MAR←{Q[8:15],Q[0:7]} $ MIR and fault if overflow out of bit 8

test CL(B)≤MAR[0:7]≤CH(B) and MAR[8:15]=0 and fault if not

MAR[8:15]←WNR and start memory

(6.4) Fault Processing:

Faults are of two types:

(i) logical errors in attempting to address outside a page group; e.g. faults on a frame pointer will be logical errors in most implementations. Such errors should simply be reported back to the processor.

(ii) translation failure, in that the base register used was a guess at the correct one and if the full virtual address falls outside of the specified page group, then (some) base register should be loaded by consulting the hash table. We assume that only F-mode accesses can cause translation failure. Please note this.

When a translation failure occurs, the contents of MIR (a swapped page #) are transformed into an initial probe address in the hash table. The table consists of 2-word entries, is 4 pages long, and starts on a page boundary whose page number is a multiple of 4. The transformation is carried out by a PROM which (1) hashes the swapped page number contained in MIR producing a 9-bit number - bits [4:14] of the probe, (2) fills bit 15 of the probe with zero, and (3) fills bits [0:5] of the probe with (page #/4) of the table start address. The resulting 16-bit number is the core address of the initial probe and is put back into MIR. Transformation takes one minor cycle. Transferring the probe to MAR and starting the memory takes another minor cycle. The word selected by the probe and the word following that appear on the bus 4 and 5 cycles later, respectively.

The memory interface board makes no other special provisions for hash table lookup. Checking whether the first probe is correct, making subsequent probes if it is not, initiating the reloading of some base register, and finally performing the F-mode memory request again are all the responsibility of the processor. Note that these activities require the original 24-bit virtual address. Note also, that the memory interface board makes provision for recovering only the word number. Hence, it is desirable to store the page number (or the full 24-bit address) in some R register whenever using F-mode addressing. Please note this.

(6.5) Protection:

The three protection bits when on enable, respectively Readable, Writable, and Executable, on a page group basis. (By Readable, we mean a fetch from memory other than instruction fetch.) Each memory request selects a base register and supplies a usage code:

01 - instruction fetch; selects E
10 - other read; selects R
11 - write; selects W
00 - circumvent protection; select no one

If a protection bit is selected and is off, then a protection fault occurs. Code 00 circumvents protection entirely at the microcode level. This

debatable feature is provided so that the hash table can be write protected and yet be written into by the table manager.

7. HASH TABLE

(7.1) Structure:

(a) Each entry in the hash table consists of

PN - 16-bit page # in virtual space

CP - 8-bit page number, in core of the core page which corresponds to PN

CL - 8-bit core page # of the low page of the page group.

CH - 8-bit core page # of the high page of the page group.

R,W,E,Ref Bits for Readable, Writable, Executable, and Referenced.

(b) The table is organized as two arrays

one of 2-word entries, holding PN and CP,R,W,E

one of 1-word entries, holding CH,CL

(c) Hash table is 1/2 full; hence, it requires 2·3·256 = 1.5K words.

(7.2) Looking up a page in the hash table:

The initial probe is computed in the memory interface board and is used to start a memory fetch from the probed entry (c.f. Section 6.4) of the first array. Four and five cycles after the initial probe fetch is initiated, words containing PN and {CP,zeros,R,W,E,Ref} appear on the bus, respectively. The initial probe is correct iff PN=page number of the original memory reference. If not, reprobing is necessary.

I suggest linear-reprobing since it is fastest to compute and with the table 50% loaded, the mean number of probes required with linear reprobing (about 1.5) is not much worse than the mean number of probes required with double hashing (about 1.3). Note, however, that the memory interface hardware makes no committment as to what reprobing technique is used.

In any reprobing method, it is necessary to recover the initial probe. This may be done by Recover (BUS←MAR); c.f. Section 5.2.

(7.3) Loading the base registers from the hash table:

(a) Relations

CP-CL = PN-VL
Q = CL-VL = CP-PN

P = CP*2↑8

(b) Data To Be Moved

(i) extract R,W,E & put into PB(B)
(ii) form CP*2↑8 & put into P(B)
(iii) compute CP-PN & put into Q(B)
(iv) get CH,CL & put into CH,CL(B)

(c) Computation

At the start, PN is in some R-register, say R1, and {CP,zeros, R,W,E,Ref} is in some other register, say R2.

```
1)  MIR←rel adr in 2nd array

2)  T←R2
    PB(B)←R2[12:14]                 ; loads PB(B)

3)  P(B)←L←377,000 and T            ; loads P(B) with CP*2↑8
    MAR←table start + MIR           ; MAR = core adr in 2nd table

4)  R2←L cycle                      ; R2 = CP

5)  T←R1                            ; T = PN

6)  Q(B)←R2-T                       ; loads Q(B) with CP-PN

7)  CH,CL(B)←MD                     ; loads CH,CL(B)
```

(7.4) Referenced bit:

The Ref bit in the hash table is taken over directly from Peter's memo. Its purpose is to assist the page manager in deciding what pages have been recently referenced and which (therefore other) pages should be tossed out. In brief, a queue of recently referenced pages is kept. Every some-milliseconds, all the Ref bits are turned off. Whenever the base-register fault manager loads a base register with a page group for which the Ref bit is off, it turns the bit on. Hence, recently referenced page groups have their Ref bit on. When the some-millisecond mass turnoff is carried out, page groups discovered to have their Ref bits on are moved to the front of the queue. Hence, the most recently referenced pages are in the front of the queue and older references trail off to the end.

8. INTENDED MODES OF USAGE

This section is a collection of suggestions and intentions as to how the virtual memory hardware should be used by the language implementors.

(8.1) Frame Pointers:

A frame, e.g. for procedure activation will reside somewhere in a page group for that frame type. A base register BF will control the page group and P(BF) will point to the core address of the head of the frame. Local variables will be expressed as positive displacements from P(BF) using D-mode addressing. Often, these displacements will only be 4-bits.

Consider transferring control to a new procedure (instance) activation in a co-routine environment such as Mesa (Smalltalk). The storage manager for the page group produces the address of a new frame as a 16-bit displacement relative to the start of the page group (blocks are linked that way). The address is supplied to the interface board as an E-mode reference. The core address E_c corresponding to E is computed as E_c = E+CL'2↑8(Section 3.4) then E_c is put into P(BF) using Recover (Section 5.2) followed by Load P(BF).

(8.2) Code Pointers:

The page group holds all code belonging to a module. Let PC be the base register for code. P(PC) points to the head of a local procedure. Instructions are addressed using D-mode as D+P(PC). If a transfer vector for the module is constructed, it can be placed in a fixed place and reference local procedures in E-mode. Assuming less than 2↑8 local procedures in a module, this allows 8-bit addressing of procedures. The new value of P(PC) for a local procedure called in this way is computed as for frame pointers. (See above.)

(8.3) F-mode addressing:

F-mode addressing allows a full 24-bit virtual address to be supplied to the memory interface board and treated as if it were generated via some base register B. This is only way to map a full 24-bit address, e.g. found in a table. Because of the way F-mode addresses are supplied to the memory interface, it is desirable to store full 24-bit addresses in the right three bytes of a 32-bit word.

9. MULTIPLE BASE REGISTERS

It is possible to decrease the fault rate by assigning multiple base registers to a single function. With 16 base registers, there are enough spares to make this reasonable. In the planned Mesa implementation, faults are caused by two purposes: (1) context changes, i.e. calls to non-local procedures and (2) user-computed addresses (not on the stack). For each of these there is a separate technique:

(9.1) Multiple PCs

Use 4 base registers as PCs. The micro-instruction field merely specifies PC - as xy00. The particular PC to be used is selected by the contents of the SB register, whose contents are OR-ed in to form the full B field. Since memory accesses other than for instructions do not want SB slamming in, it is necessary to be able to disable and enable this OR-ing. To control this, we use another bit in the micro instruction. Also, there must be some policy for managing the pool of PC's. Round robin seems the best choice.

(9.2) Multiple base registers for User-Computed Pointers:

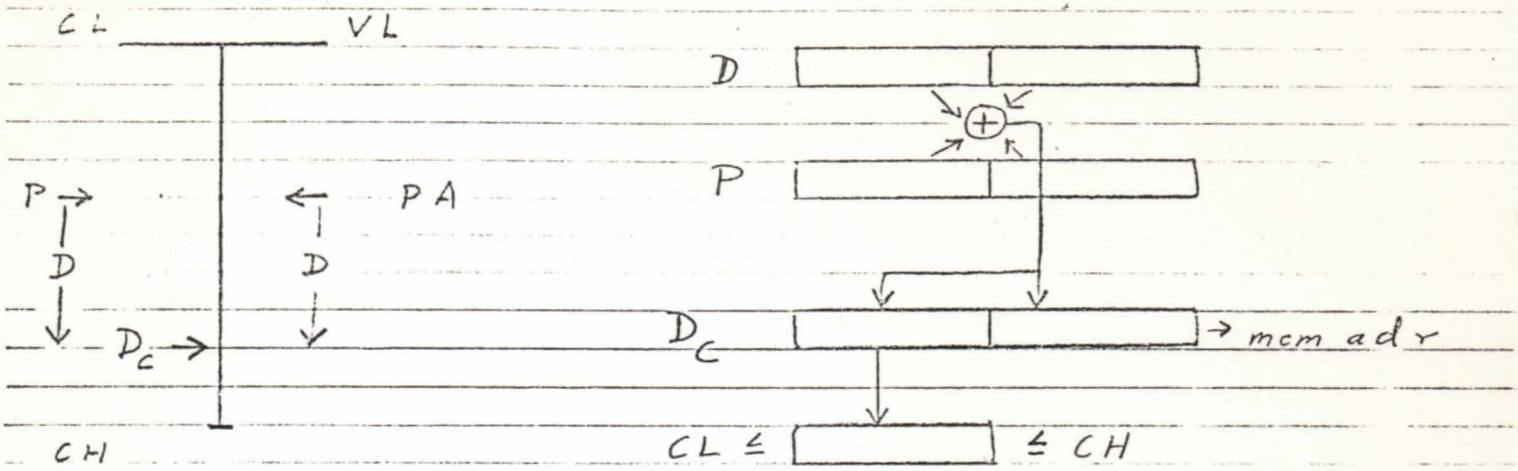
It is currently planned to assign two base registers to user-computed pointers: one for read and one for write; the fault rate for this class of references is about 20-30%. Section 4.1 discusses a better organization which might be adopted by the Mesa compiler at some point. A third organization which may be effective even in the absence of any compiler knowledge, is to use a block of 2 base registers for each of read or write, and check each of these. This is useful only in F-mode. In that mode, it is possible to decide whether or not a fault is occuring after the first 3 minor cycles; subsequently, trying additional base registers takes 2 minor cycles. A fault requires between 8 and 30 minor cycles to hash, check, and reload base registers. Hence, we guess that 2 base registers would be effective, in that the savings in reloading would outweigh the time to check both. Some initial data has tended to confirm this. More careful measurements are now being made.

It is possible, of course, to go to more than 2 base registers, but then additional hardware is required. Consider, for example, using 4-base registers. When a fault through all 4 occurs, it is necessary to reload one. Suppose a round-robin policy is used. Then, on subsequent searches through the set of 4, we want to start with the new one. This requires a register to hold the number of the new one and counter to cycle through the set of four. Finally, we need instructions to load and store the counter.

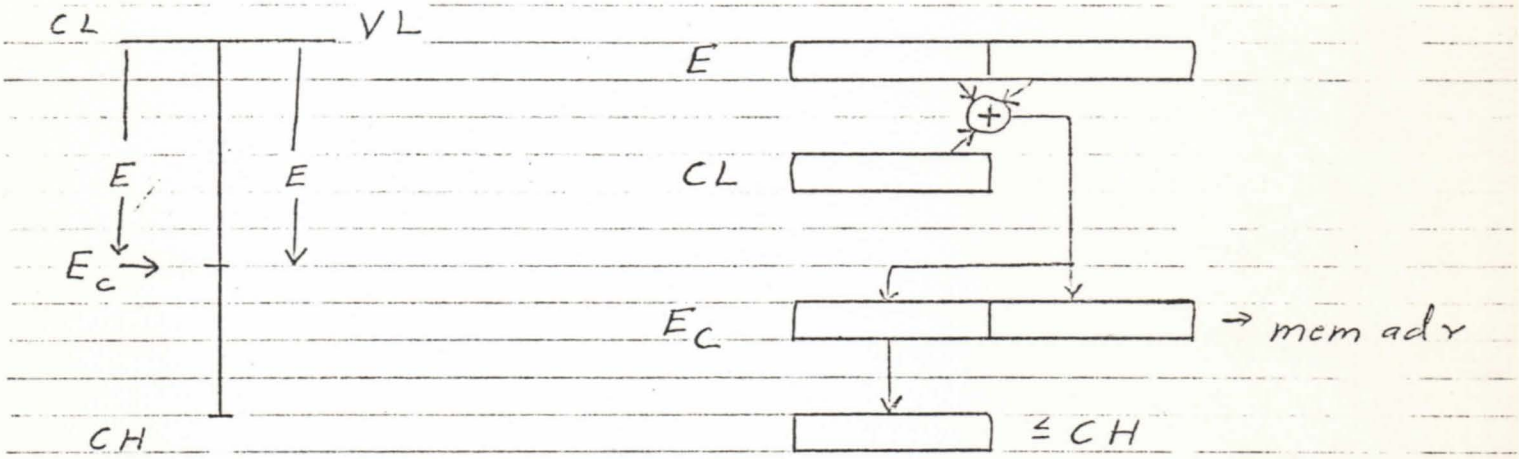
Conclusion:

We need measurements of the effect of multiple (1,2,4) PC's and multiple (1,2,4) base registers for user-computed pointers.

(a) D-mode



(b) E-mode



(c) F-mode

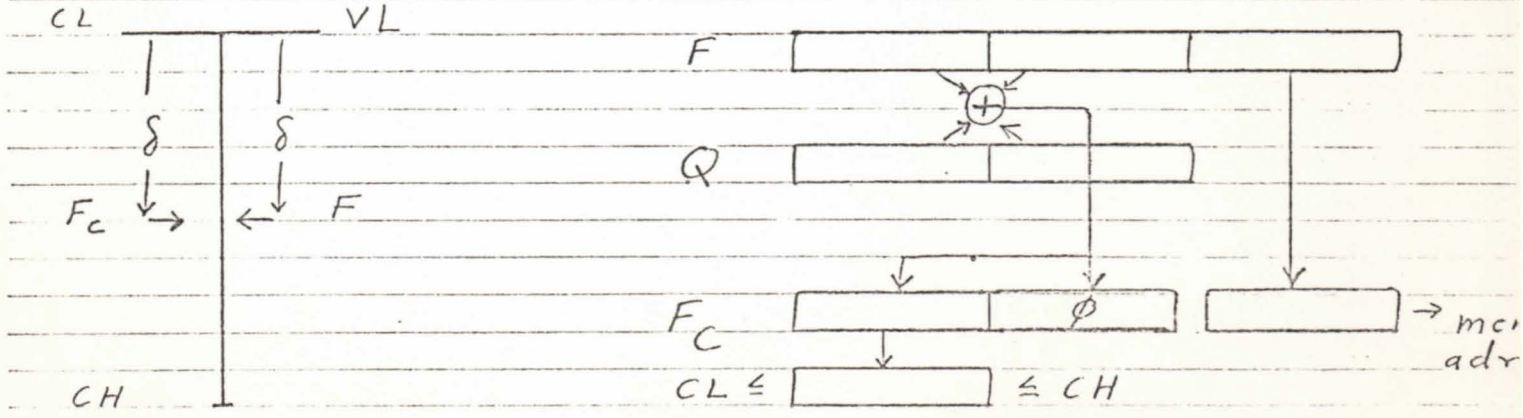


Figure 1. Addressing Modes

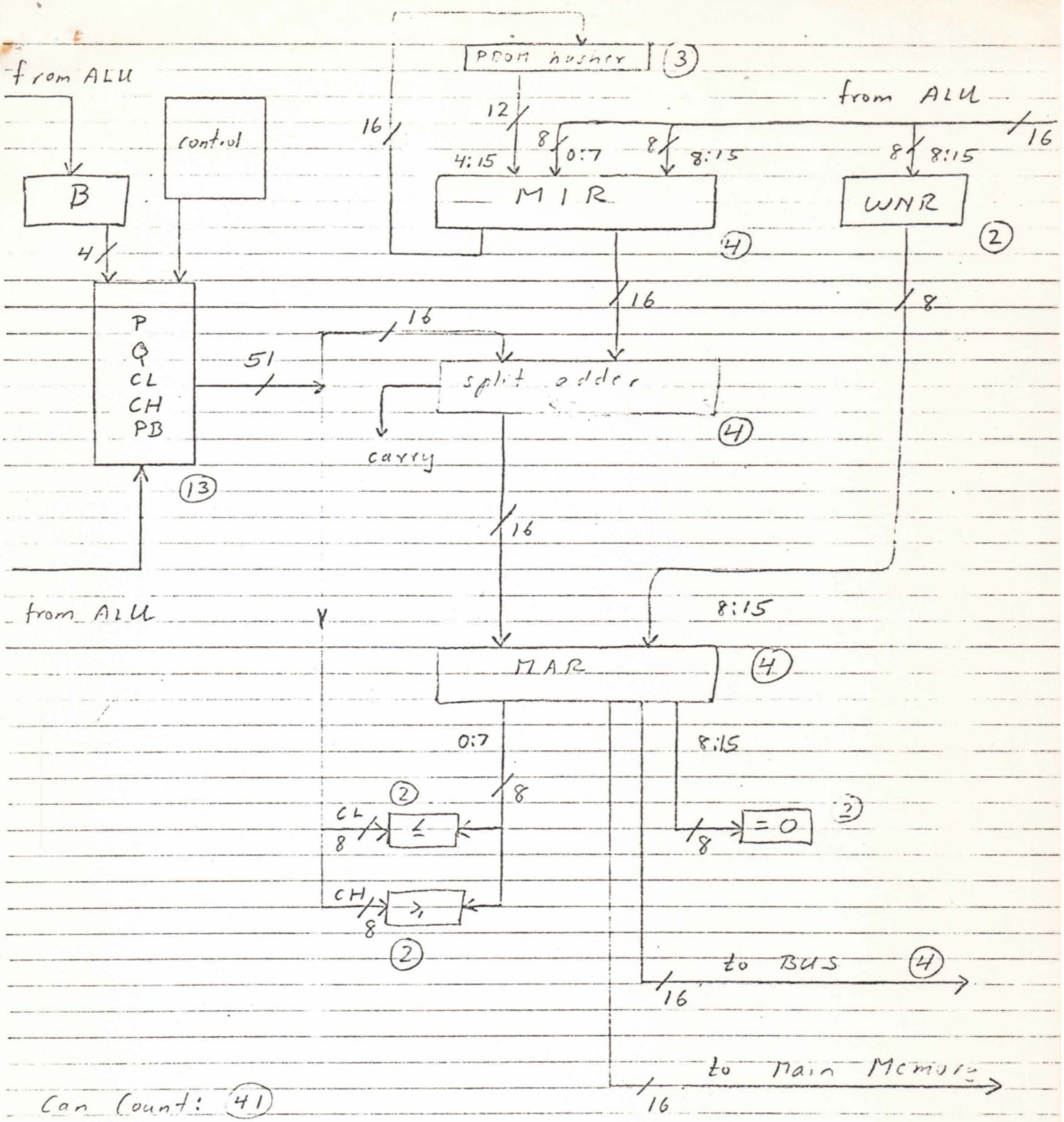


Figure 2. Address Translation Hardware