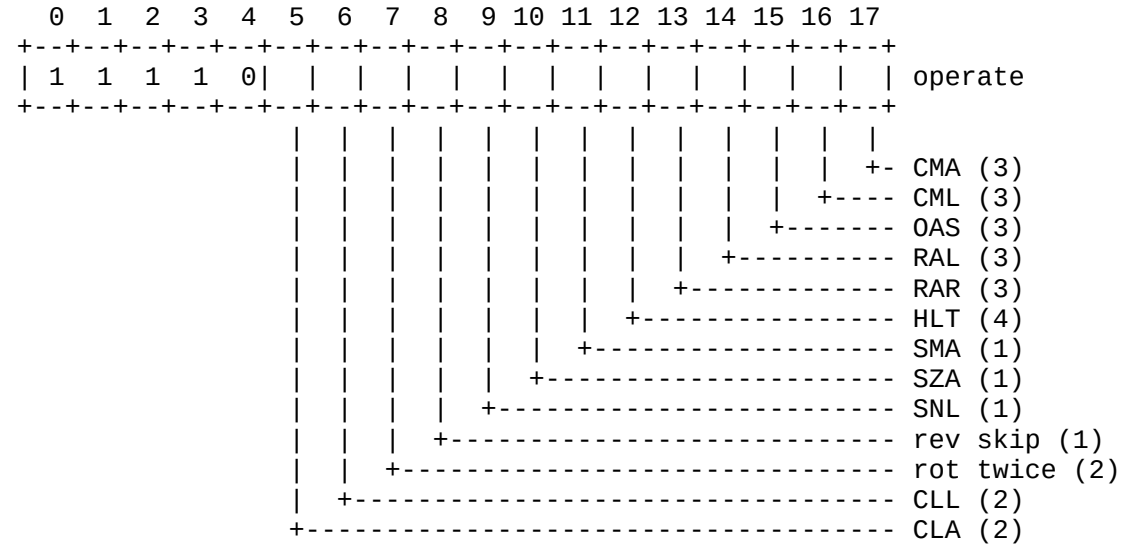
Decoding The H316/H516 “Generic A” Instructions

Bob Supnik, 23-Jul-2001 [revised links 18-Feb-2005]

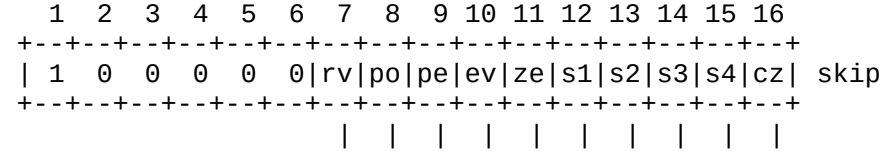
Introduction

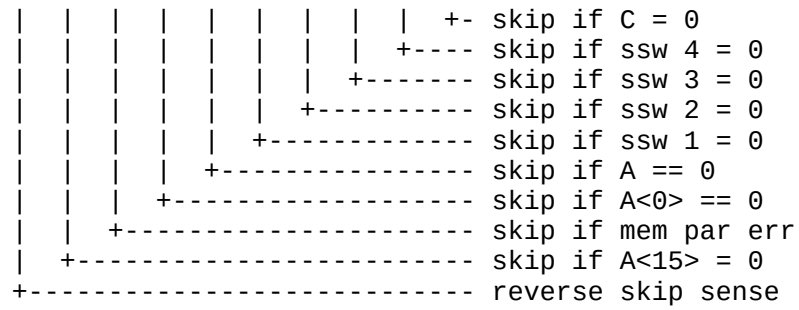
The Honeywell Series 16 (H116, H316, H416, H516, H716) was a family of 16b minicomputers sold from the mid-60’s to the mid-70’s. The series was originally built by Computer Controls Corporation and designated the DDP family. In 1969, Honeywell purchased Computer Controls and renamed the family the Series 16. Historically, the most famous model in the series was the H516, which was used as the original Arpanet “Interface Message Processor” or IMP – the world’s first router. This paper is concerned with the H316 and H516, which were logically identical.

Like many 1960’s minicomputers, the H316/H516 was accumulator rather than general register based. It had an instruction group (known in the hardware documentation as the Generic A group) for manipulating the accumulator (A) and the carry flag (C). In many contemporary machines, the accumulator manipulation instruction was microcoded; that is, individual bits or fields in the instruction controlled individual functions in the data path. For example, the PDP-7/9 operate instruction was decoded as follows:



In the H316/H516, the skip instruction group was also microcoded:





But the generic A instruction group were not, apparently, microcoded. In addition, the generic A group was very sparsely encoded: only 16 combinations out of a possible 1024 were defined. What did the undefined instructions do? How did the group really work?

Prior Work

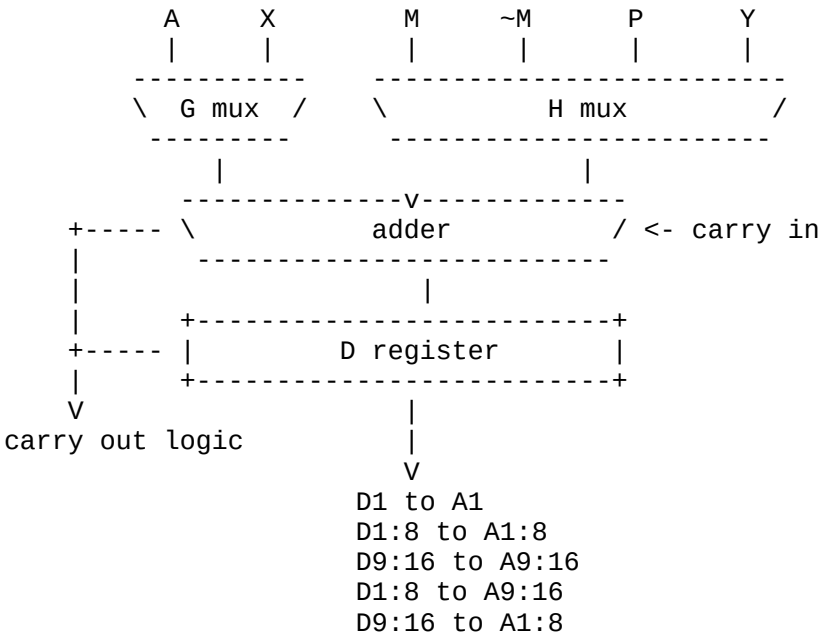
In 1971, Donald Bell, at the National Physics Laboratory in the UK, wrote a technical note on “Micro-coding the DDP-516 Computer” [1]. By scanning all possible 1024 generic A instructions, he demonstrated that:

1. All of the generic A instructions had reproducible results.
2. Instruction bit 7 had no effect on operation, effectively halving the number of possible unique instructions.
3. The 512 potential remaining instructions fell into groups, with up to 46 different instructions producing the same result.

Bell offered a partial explanation of how the generic A group was implemented; but his explanation was insufficient, as Adrian Wise demonstrated in his 1999 H316/H516 simulator [2].

Generic A Decoding

The implementation of the generic A group depends on the particular details of the H316/H516 data path. The data path consists of a two input adder, multiplexors on the adder inputs, a results distribution register (D), logic for storing part or all of D back into A, and logic for manipulating the carry flag:



Some points to note:

1. The G mux selects A, X, or no input. If there is no input, the output is 0. For the generic A instructions, the only available choices are A or 0.
2. The H mux selects M (memory input), ~M (memory input complemented), P, Y, or no input. If there is no input, the output is 177777. For the generic A instructions, the only available choices are M and ~M together, producing 0, or no input, producing 177777.
3. The adder performs either a true add or, if carries are suppressed, an exclusive OR.
4. Unless a register is explicitly cleared, a transfer OR's new information into the register. If multiple sources are transferred simultaneously, all the sources are OR'd together.
5. The adder lacks a ~A input. Any instruction requiring the complement of A must use the adder to perform the operation A XOR 177777.

Generic A instructions are performed in four or six phases. A four phase instruction consists of:

T1	decoding
T2	setup
T3	adder
T4	distribution, carry

A six phase instruction repeats phases 2 and 3, with special overrides on the arithmetic unit during the repeated cycles:

T1	decoding
T2	setup
T3	arithmetic
T2 repeat	distribution
T3 repeat	adder, forced add
T4	distribution, carry

The data path and timing is controlled by hard-wired decode logic, as follows:

phase	signal	decoding	function
2-3 (tlate)	EASTL	(((m12+m16)x!AZZZZ) + (m9+m11+AZZZZ)	Enable A to adder input G (else, input 1 = 0)
	EASBM	m9+m11+AZZZZ	Enable 0 to adder input H (else, input 2 = ‘177777)
	JAMKN	(m12+m16)x!AZZZZ	Force adder carry network to 0 (adder generates exclusive OR)
	EIKI7	(m15x(C+!m13))x!JAMKN	Enable 1 to adder carry in (else, adder carry in = 0)
3	SETAZ	m8xm15x!AZZZZ	Set AZZZZ
	CLDTR	always	Clear D
	ESDTS	always	Enable adder output to D
If AZZZZ			
2	CLATR	t2xAZZZZ	Clear A
	EDAHS	t2xAZZZZ	Enable D high to A high
	EDALS	t2xAZZZZ	Enable D low to A low
2-3 (tlate)	EASTL	(((m12+m16)x!AZZZZ) + (m9+m11+AZZZZ)	Enable A to adder input 1 (else, input 1 = 0)
	EASBM	m9+m11+AZZZZ	Enable 0 to adder input 2 (else, input 2 = ‘177777)
	JAMKN	(m12+m16)x!AZZZZ	Force adder carry network to 0 (adder generates exclusive OR)
	EIKI7	(m15x(C+!m13))x!JAMKN	Enable 1 to adder carry in (else, adder carry in = 0)
3	CLDTR	always	Clear D
	ESDTS	always	Enable adder output to D
End if AZZZZ			
4	CLATR	t4x(m11+m15+m16)	Clear A
	CLA1R	t4x(m10+m14)	Clear A1
	EDAHS	t4x((m11xm14)+m15+m16)	Enable D high to A high
	EDALS	t4x((m11xm13)+m15+m16)	Enable D low to A low
	ETAHS	t4x(m9xm11)	Enable D low to A high
	ETALS	t4x(m10xm11)	Enable D high to A low
	EDA1R	t4x((m8xm10)+m14)	Enable D1 to A1
	CBITL	t4x(m9x!m11)	Clear C, conditionally set C from adder overflow
	CBITG	D1xm10xm12	Conditionally set C if D1 = 1
	CBITE	m8xm9	Unconditionally set C

Generic A Instructions

The logic in the previous section was implemented as part of the SIMH simulator [3] for the H316/H516. Using a special test harness, the simulator produced a decomposition of the generic A group into unique instructions. This was compared to the output of the original instruction scan program, executing on a real H316; the results were identical. Thus, the simulated logic accurately reproduced the generic A implementation of a real H316.

The following table lists the unique instructions within the generic A group. Where Bell provided mnemonics, they are used. Where he did not, the instruction function is shown in a C-like notation.

NOP: no operation
140000 140010 140020 140030 140041 140043 140045 140047
140051 140053 140054 140055 140057 140061 140062 140063
140065 140066 140067 140071 140072 140073 140074 140075
140076 140077 140400 140410 140420 140430 140441 140445
140451 140454 140455 140461 140465 140471 140474 140475

CMA (complement accumulator): A ← ~A
140001 140003 140005 140007 140011 140013 140015 140017
140021 140022 140023 140025 140026 140027 140031 140032
140033 140035 140036 140037 140101 140103 140105 140107
140111 140113 140115 140117 140401 140405 140411 140415
140421 140425 140431 140435 140501 140505 140511 140515

CRA (clear A): A ← 0
140002 140006 140040 140060 140102 140106 140440 140460

SSM (set sign minus): A1 ← 1
140004 140014 140104 140114 140404 140414 140500 140504
140510 140514

CM1: A ← C - 1
140012 140016 140112 140116

CHS (change sign): A1 ← ~A1
140024 140034 140424 140434

AD1 (add 1 to A, do not change C): A ← A + 1
140042 140046 140443 140447 140462 140463 140466 140467

CAR (clear A right): A ← A & 177400
140044 140064 140444 140464

CAL (clear A left): A ← A & 377
140050 140070 140450 140470

ADC (add C to A, do not change C): A ← A + C
140052 140056 140453 140457 140472 140473 140476 140477

SSP (set sign plus): A1 ← 0
140100 140110

C ← C | ~A1, A1 ← 0
140120 140130

CMA/ORC: A ← ~A, C ← C | A1
140121 140122 140123 140125 140126 140127 140131 140132
140133 140135 140136 140137 140521 140525 140531 140535

CHS/ORC: A1 ← ~A1, C ← C | A1
140124 140134 140520 140524 140530 140534

ICL (interchange and clear left): A ← A >> 8
140140

BTR (OR left to right): A ← A | (A >> 8)
140141 140143 140145 140147 140151 140153 140154 140155
140157 140541 140545 140551 140554 140555

A ← (A + 1) | ((A + 1) >> 8)
140142 140146 140543 140547

LTR (copy left to right): A ← (A & 177400) | (A >> 8)
140144 140544

BCL (OR to right, clear left): A ← (A & 377) | (A >> 8)
140150

A ← (A + C) | ((A + C) >> 8)
140152 140156 140553 140557

ORC/ICL: C ← C | A1, A ← A >> 8
140160

ORC/BTR: C ← C | A1, A ← A | (A >> 8)
140161 140162 140163 140165 140166 140167 140171 140172
140173 140174 140175 140176 140177 140561 140565 140571
140574 140575

ORC/LTR: C ← C | A1, A ← (A & 177400) | (A >> 8)
140164 140564

ORC/BCL: C ← C | A1, A ← (A & 377) | (A >> 8)
140170

RCB (reset C bit): C ← 0
140200 140201 140203 140204 140205 140207 140210 140211
140213 140214 140215 140217 140220 140221 140222 140223
140224 140225 140226 140227 140230 140231 140232 140233
140234 140235 140236 140237 140301 140303 140304 140305
140307 140311 140313 140314 140315 140317

AOA (add 1 to A): A ← A + 1, C ← overflow
140202 140206 140302 140306

ACA (add C to A): A ← A + C, C ← overflow
140212 140216 140312 140316

ICR (interchange and clear right): A ← A << 8
140240 140260

BTL (OR right to left): A ← A | (A << 8)
140241 140243 140245 140247 140251 140253 140254 140255
140257 140261 140262 140263 140265 140266 140267 140271
140272 140273 140274 140275 140276 140277

A ← (A + 1) | ((A + 1) << 8)
140242 140246

BCR (OR to left, clear right): A ← (A & 177400) | (A << 8)
140244 140264

RTL (copy right to left): A ← (A & 377) | (A << 8)
140250 140270

A ← (A + C) | ((A + C) << 8)
140252 140256

RCB/SSP: C ← 0, A1 ← 0
140300 140310

CSA (copy sign and set plus): C ← A1, A1 ← 0
140320 140330

CPY (copy sign): C ← A1
140321 140322 140323 140324 140325 140326 140327 140331
140332 140333 140334 140335 140336 140337

ICA (interchange A): A ← byteswap (A)
140340

BTB (OR to both halves): A ← A | byteswap (A)
140341 140343 140345 140347 140351 140353 140354 140355
140357

A ← (A + 1) | byteswap (A + 1)
140342 140346

A ← A1 | byteswap (A)
140344

A ← (A & 0377) | byteswap (A)
140350

A ← (A + C) | byteswap (A + C)
140352 140356

ORC/ICA: C ← C | A1, A ← byteswap (A)
140360

ORC/BTB: C ← C | A1, A ← A | byteswap (A)
140361 140362 140363 140365 140366 140367 140371 140372
140373 140374 140375 140376 140377

C ← C | A1, A ← A1 | byteswap (A)
140364

C ← C | A1, A ← (A & ‘377) | byteswap (A)
140370

LD1 (load 1): A ← 1
140402 140406 140502 140506

TCA (two’s complement A): A ← -A
140403 140407 140422 140423 140426 140427 140503 140507

ISG (inverse sign): A ← 2*C - 1
140412 140416 140512 140516

CMA/ADC: A ← ~A + C
140413 140417 140432 140433 140436 140437 140513 140517

A2A (add 2 to A): A ← A + 2
140442 140446

A2C (add 2*C to A): A ← A + 2*C
140452 140456

TCA/ORC: A ← -A, C ← C | A1
140522 140523 140526 140527

CMA/ADC/ORC: A ← ~A + C, C ← C | A1
140532 140533 140536 140537

ICS (interchange, clear left, keep sign bit): A ← A1 | (A >> 8)
140540

A ← (A + 2) | ((A + 2) >> 8)
140542 140546

A ← A1 | (A & 0377) | (A >> 8)
140550

A ← (A + 2*C) | ((A + 2*C) >> 8)
140552 140556

A ← A1 | (A >> 8), C ← C | A1
140560

A ← (A + 1) | ((A + 1) >> 8), C ← C | A1
140562 140563 140566 140567

A ← A1 | (A & 377) | (A >> 8), C ← C | A1
140570

A ← (A + C) | ((A + C) >> 8), C ← C | A1
140572 140573 140576 140577

SCB (set C bit): C ← 1
140600 140601 140604 140605 140610 140611 140614 140615
140620 140621 140624 140625 140630 140631 140634 140635
140700 140701 140704 140705 140710 140711 140714 140715
140720 140721 140724 140725 140730 140731 140734 140735

A2A/SCB: A ← A + 2, C ← 1
140602 140606 140702 140706

AOA/SCB: A ← A + 1, C ← 1,
140603 140607 140622 140623 140626 140627 140703 140707
140722 140723 140726 140727

A2C/SCB: A ← A + 2*C, C ←1
140612 140616 140712 140716

ACA/SCB: A ← A + C, C ← 1
140613 140617 140632 140633 140636 140637 140713 140717
140732 140733 140736 140737

ICR/SCB: A ← A << 8, C ← 1
140640 140660

A ← A | (A << 8), C ← 1
140641 140645 140651 140654 140655 140661 140665 140671
140674 140675

A ← (A + 2) | ((A + 2) << 8), C ← 1
140642 140646

A ← (A + 1) | ((A + 1) << 8), C ← 1
140643 140647 140662 140663 140666 140667

A ← (A & 177400) | (A << 8), C ← 1
140644 140664

RTL/SCB: A ← (A & 377) | (A << 8), C ← 1
140650 140670

A ← (A + 2*C) | ((A + 2*C) << 8), C ← 1
140652 140656

A ← (A + C) | ((A + C) << 8), C ← 1
140653 140657 140672 140673 140676 140677

A ← A1 | byteswap (A), C ← 1
140740 140760

BTB/SCB: A ← A | byteswap (A), C ← 1
140741 140745 140751 140754 140755 140761 140765 140771
140774 140775

A ← (A + 2) | byteswap (A + 2), C ← 1
140742 140746

A ← (A + 1) | byteswap (A + 1), C ← 1
140743 140747 140762 140763 140766 140767

A ← (A & 177400) | byteswap (A), C ← 1
140744 140764

A ← A1 | (A & 377) | byteswap (A), C ← 1
140750 140770

A ← (A + 2*C) | byteswap (A + 2*C), C ← 1
140752 140756

A ← (A + C) | byteswap (A + C), C ← 1
140753 140757 140772 140773 140776 140777

This chart differs from Bell's in one case. Bell identified 140413 as CMA/ACA, with equivalent encodings 140417, 140432, 140433, 140436, 140437, 140513, 140517, 140532, 140533, 140536, 140537. On the H316, 140413 is actually CMA/ADC (C is not changed), and the equivalent encodings are 140417, 140432, 140433, 140436, 140437, 140513, 140517. The four instructions 140532, 140533, 140536, 140537 are a separate group implementing CMA/ADC/ORC. This does not mean that Bell was wrong: he ran his experiment on an H516, while this table is derived from an H316. The machines are supposedly equivalent, but without H516 logic prints, or access to a real system, we can't be sure.

Acknowledgements

As is often the case in computer history work, this paper would not have been possible without the help of colleagues whom I know mostly or exclusively through the Internet. Adrian Wise created and maintains an invaluable set of web pages on the computers, transcribed software and manuals, and wrote the first H316/H516 simulator. Al Kossow provided online documentation. Mike Umbricht provided the hardware prints that unlocked the secrets of the generic A logic. Finally, Adrian closed the loop between simulated logic and real machine by running the instruction scan on his H316.

References

[1] On the web at http://www.series16.adrianwise.co.uk/computers/microcode.html.
[2] On the web at http://www.series16.adrianwise.co.uk/computers/emulator.html. The current version (1.4) reflects the results of this paper.

[3] On the web at http://simh.trailing-edge.com.