Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 26

Programming the 816

type d'upgrade de ce ducument : 3
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 50 page(s) environ

Domaine : 816

VERSION :
DATE : 1.05.85

Programming the 65816

GOALS

- Provide enough information about the overall architecture, interrupts, addressing modes, instruction set, and programming techniques to enable you to write significant application and system programs in 65816 native mode assembly language.

- Provide the foundation for a future class on machine level details of the Apple //-16.

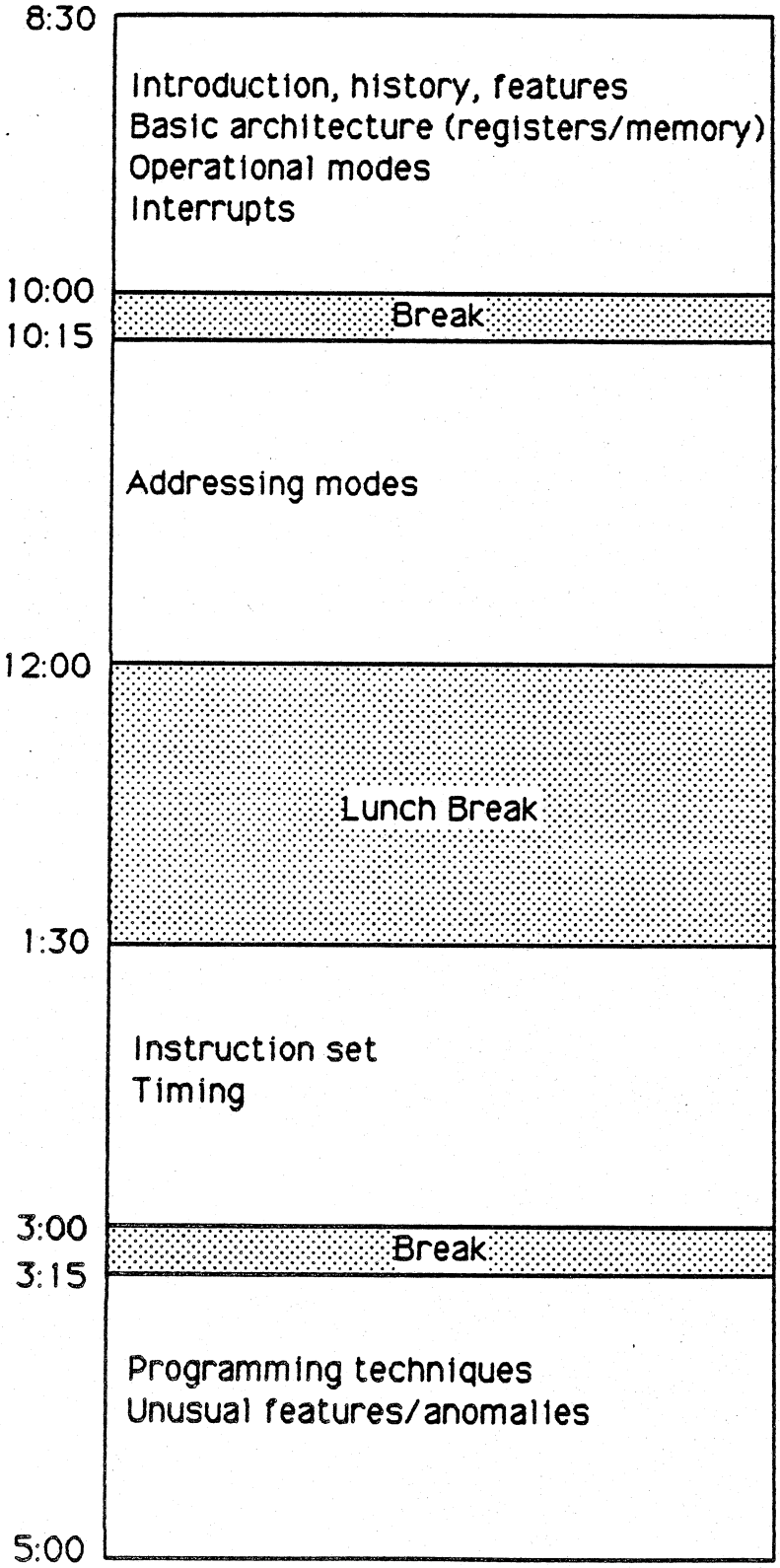
- Stimulate discussion about "good" and "bad" programming techniques for the 65816.

OMISSIONS

The course does not cover any of the following:

- Emulation mode operation of the 65816 (it works like a 6502, and I assume everyone already knows how to program a 6502).
- Apple //-16 machine details such as soft switches, memory layout, control registers, etc. (these will be the subject of a followon course to be given when details are 99 44/100% firmed up).
- Hands-on programming (no time, not enough machines).

SCHEDULE

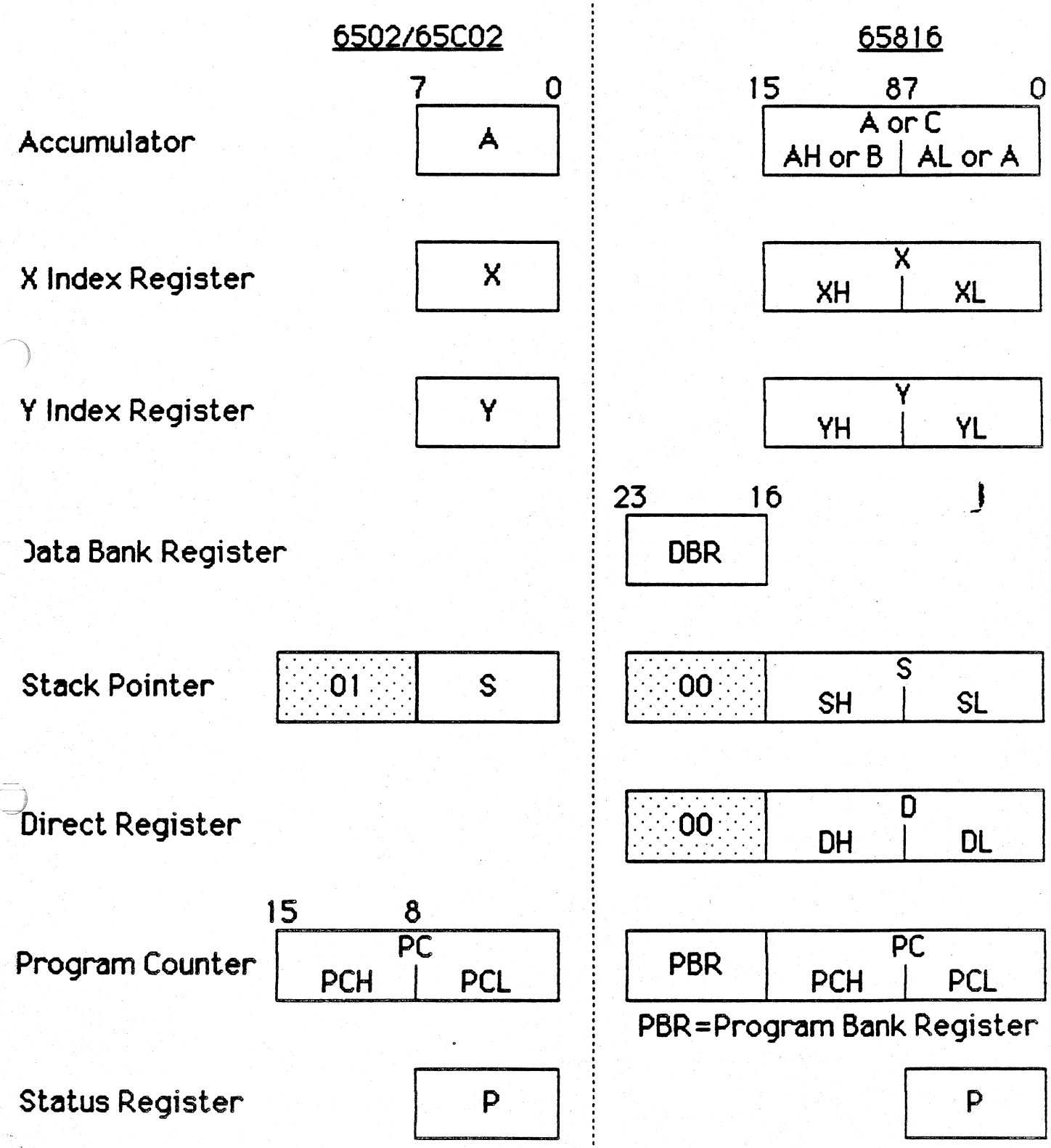


65XXX FAMILY RELATIONSHIPS

Characteristic	6502	65C02	65C802	65C816
Year available	1975	1983	1985	1985
Technology	NMOS	CMOS	CMOS	CMOS
ALU width (bits)	8	8	16	16
Address bus width (bits)	16	16	16	24*
Data bus width (bits)	8	8	8	8
Maximum memory size (by)	64 K	64 K	64 K	16 M
Maximum stack size (by)	256	256	64 K	64 K
Number of defined opcodes	151	178	256	256
Number of addressing modes	13	15	24	24
Relocatable zero page?	No	No	Yes	Yes
Software compatible w/ 6502?	Yes	Almost	Yes	Yes
Pin compatible w/ 6502?	Yes	Yes	Yes	No
Fast block move instructions?	No	No	Yes	Yes

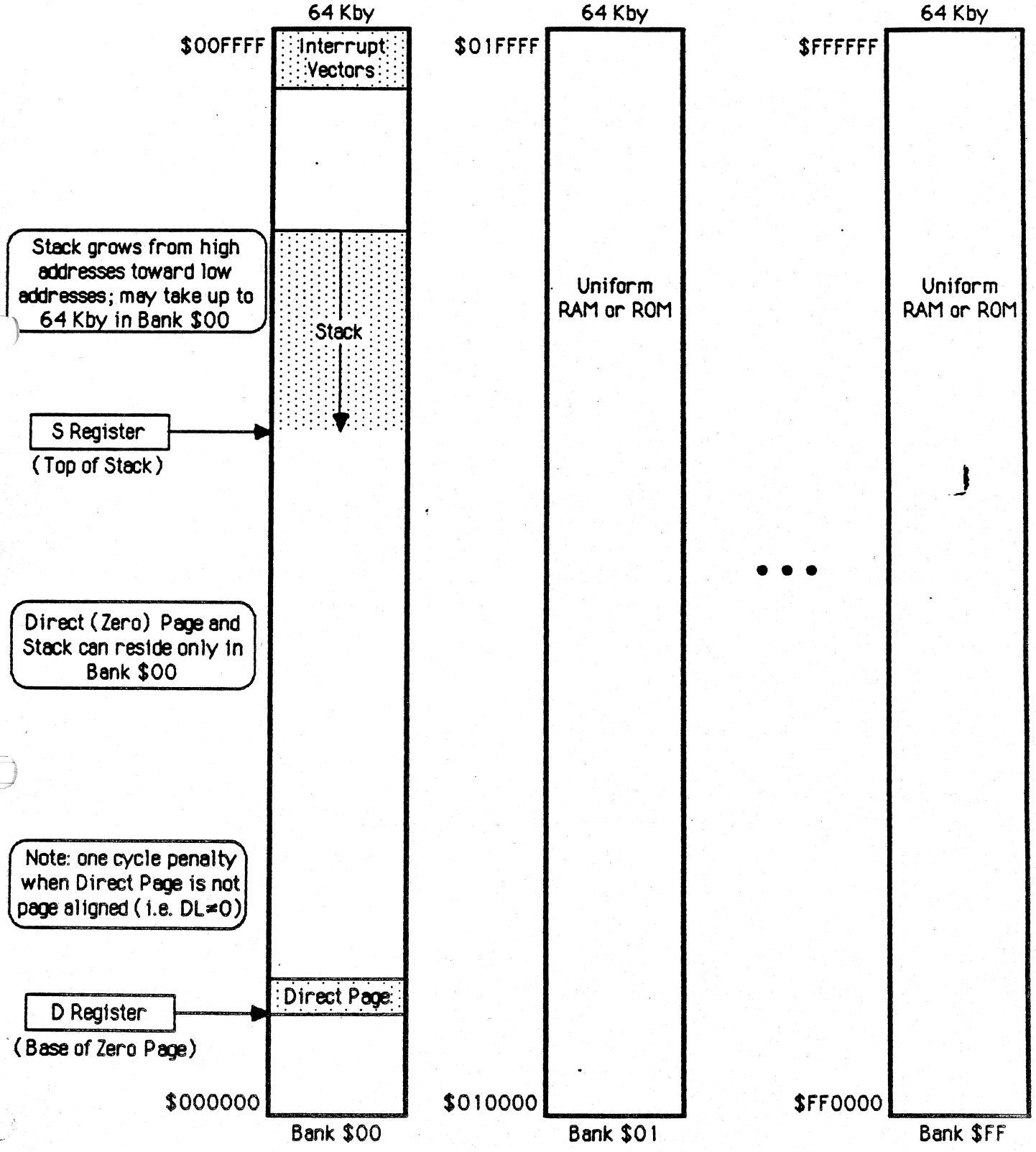
* Note: Upper 8 bits of address bus are multiplexed on data bus.

6502/65C02/65816 REGISTERS



65816 MEMORY

(Not to scale)



REGISTER DESCRIPTIONS

ACCUMULATOR–A

- Stores one operand and result of most arithmetic/logical operations
- 16 bits wide when e=0 and m=0
- 8 bits wide when e=1 or when e=0 and m=1
- XBA (SWA) instruction eXchanges upper and lower halves of 16-bit accumulator

INDEX REGISTERS–X, Y

- Generally provide index values for effective address calculation
- Hold operands for a restricted set of arithmetic/logical operations
- 16 bits wide when e=0 and x=0
- 8 bits wide when e=1 or when e=0 and x=1

DATA BANK REGISTER–DBR

- Provides the upper 8 bits of effective address in addressing modes that otherwise generate only the lower 16 bits–e.g. absolute

STACK POINTER–S

- Indicates the next available location on the stack
- Pushes decrease the value in S; pops increase it
- Used in formulation of effective address for stack-relative addressing modes
- Used to store context for subroutine calls and interrupts

REGISTER EXPLANATIONS

DIRECT–D

- "Relocates" direct page (zero page) anywhere in Bank $00
- Thus, affects all addressing modes containing "direct" references
- One cycle timing penalty on all direct references when direct page is not aligned on a page boundary (i.e. DL≠0)

PROGRAM BANK REGISTER–PBR

- Normally serves as the upper 8 bits of the 24-bit address of the next instruction
- Used in conjunction with the PC to provide a full 24-bit instruction address
- But, when the PC is incremented, there is no carry from the high bit of the PC into the low bit of the PBR
- Also provides the high byte of the effective address for certain addressing modes

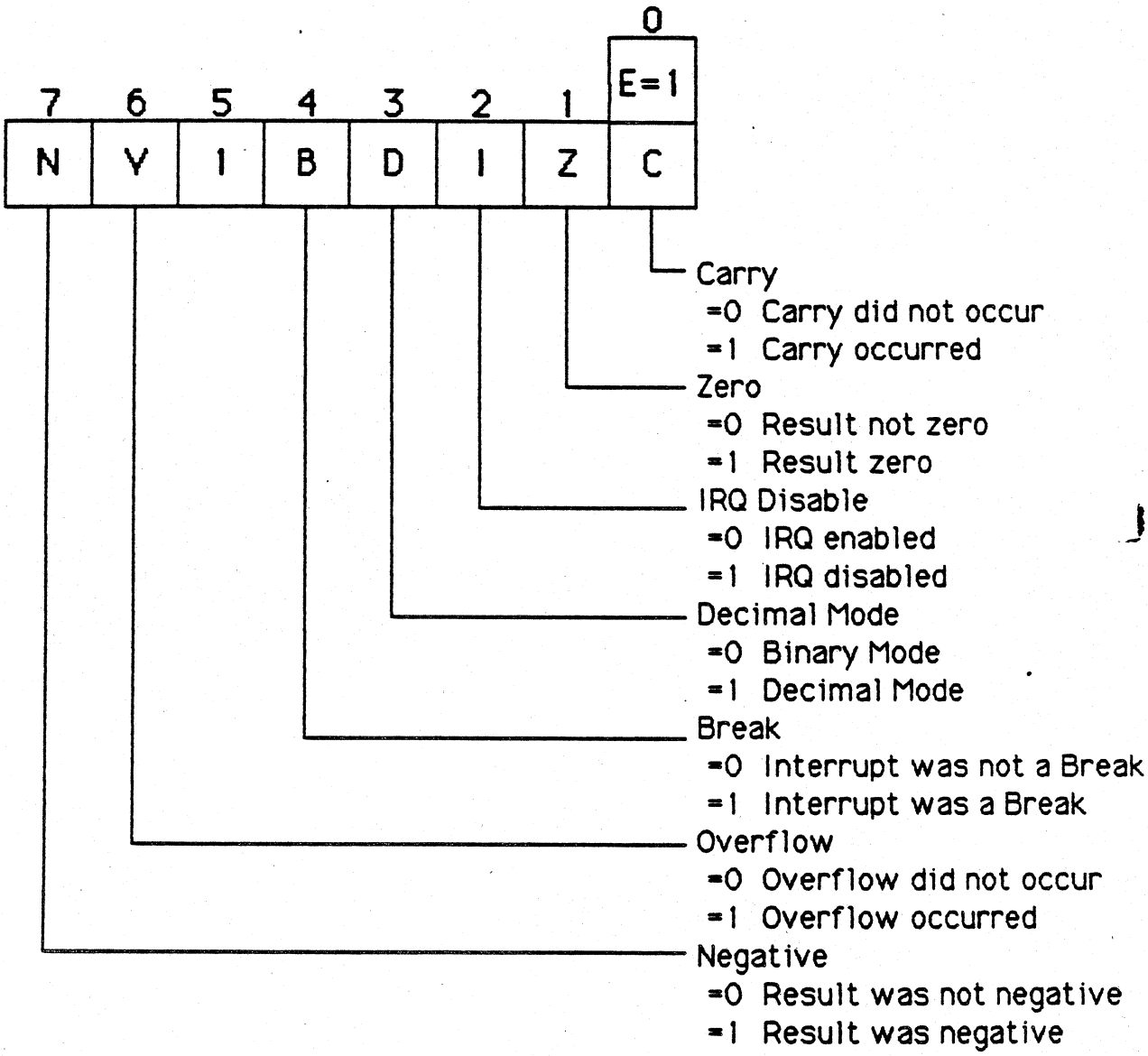
PROGRAM COUNTER–PC

- Holds the low order 16 bits of the 24-bit address of the next instruction
- Used in conjunction with the PBR to provide a full 24-bit instruction address
- Carry out of high bit is not propagated into the PBR

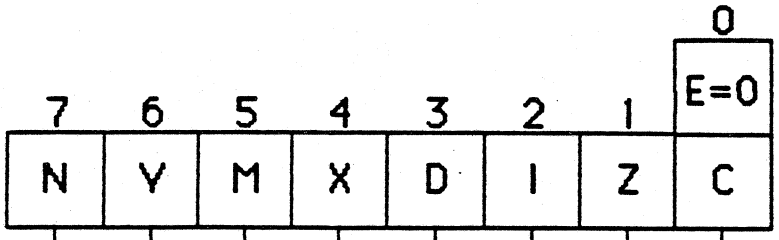
PROCESSOR STATUS–P

- Contains status flags and mode select bits

P (STATUS AND CONTROL) REGISTER–6502/65C02 EMULATION MODE



P (STATUS AND CONTROL) REGISTER–65816 NATIVE MODE



Carry
=0 Carry did not occur
=1 Carry occurred

Zero
=0 Result not zero
=1 Result zero

IRQ Disable
=0 IRQ enabled
=1 IRQ disabled

Decimal Mode
=0 Binary Mode
=1 Decimal Mode

Index Register Select
=0 Index registers are 16 bits
=1 Index registers are 8 bits

Memory/Accumulator Select
=0 Memory/Accumulator references are 16 bits
=1 Memory/Accumulator references are 8 bits

Overflow
=0 Overflow did not occur
=1 Overflow occurred

Negative Result
=0 Result was not negative
=1 Result was negative

65816 PROCESSOR MODES

EMULATION MODE–(e=1)

- Executes all 6502 instructions exactly like a 6502, including cycle-by-cycle bus contents. FROM THE SOFTWARE STANDPOINT, IT IS A 6502 (with some additional instructions).
- Executes all additional 65C02 instructions exactly like a 65C02, including cycle-by-cycle bus contents.
- Executes new 65816 instructions, including new addressing modes and 24-bit addressing. However, many instructions don't make sense (e.g. MVN, MVP).

NATIVE MODE AND VARIANTS–(e=0)

- Implements the full 65816 architecture including 16 Mby address space and full set of registers (A, X, Y, DBR, S, D, PBR, PC, P).
- "Full native mode"–(e=0, m=0, x=0): Accumulator, index registers, and memory references are all 16 bits.
- "Mixed native modes"–(e=0 but m=1 or x=1 or both): restricts accumulator/memory and/or index register/memory references to 8 bits.

EFFECT OF m AND x BIT CHANGES ON REGISTERS

- Switch m bit from 8 to 16 bits or from 16 to 8 bits:

  No effect on AH (B) or AL (A)

  In 8 bit mode, AH (B) is accessible via XBA

  Even in 8 bit mode, instructions that transfer A register—e.g. TCS—always move 16 bits

- Switch x bit from 16 to 8 bits

  XH and YH set to zero—old contents lost

- Switch x bit from 8 to 16 bits

  XH=00, XL unchanged, YH=00, YL unchanged

OBSERVATIONS

- Some instructions perform exactly the same operation no matter how the e, m, and x control bits are set. For example, PEI always pushes two bytes from zero page onto the stack.

- Some instructions operate sensibly only in native mode. For example, MVN and MVP use the 16 bit A, X, and Y registers to specify operands, so they don't work in other than full native mode.

- Some instructions change their native mode operation depending on the values of the m and/or x bits. For example, LDX # loads a 1-byte value if x=1 and a 2-byte value if x=0. LDA # loads a 1-byte value if m=1 and a 2-byte value if m=0.

- Some instructions that you would expect to be affected by the m and x bit settings are not. For example, TCS and TSC always transfer 16 bits.

SUGGESTIONS

- Mixed native modes (m=1 or x=1 or both) may be dangerous to your sanity and the sanity of those who call the procedures you write. Try to use either emulation mode or full native mode. Try to restrict use of m=1 or x=1 to short code sequences. Assume that m=0, x=0 are the standard native mode settings, and always restore these values if you change them.

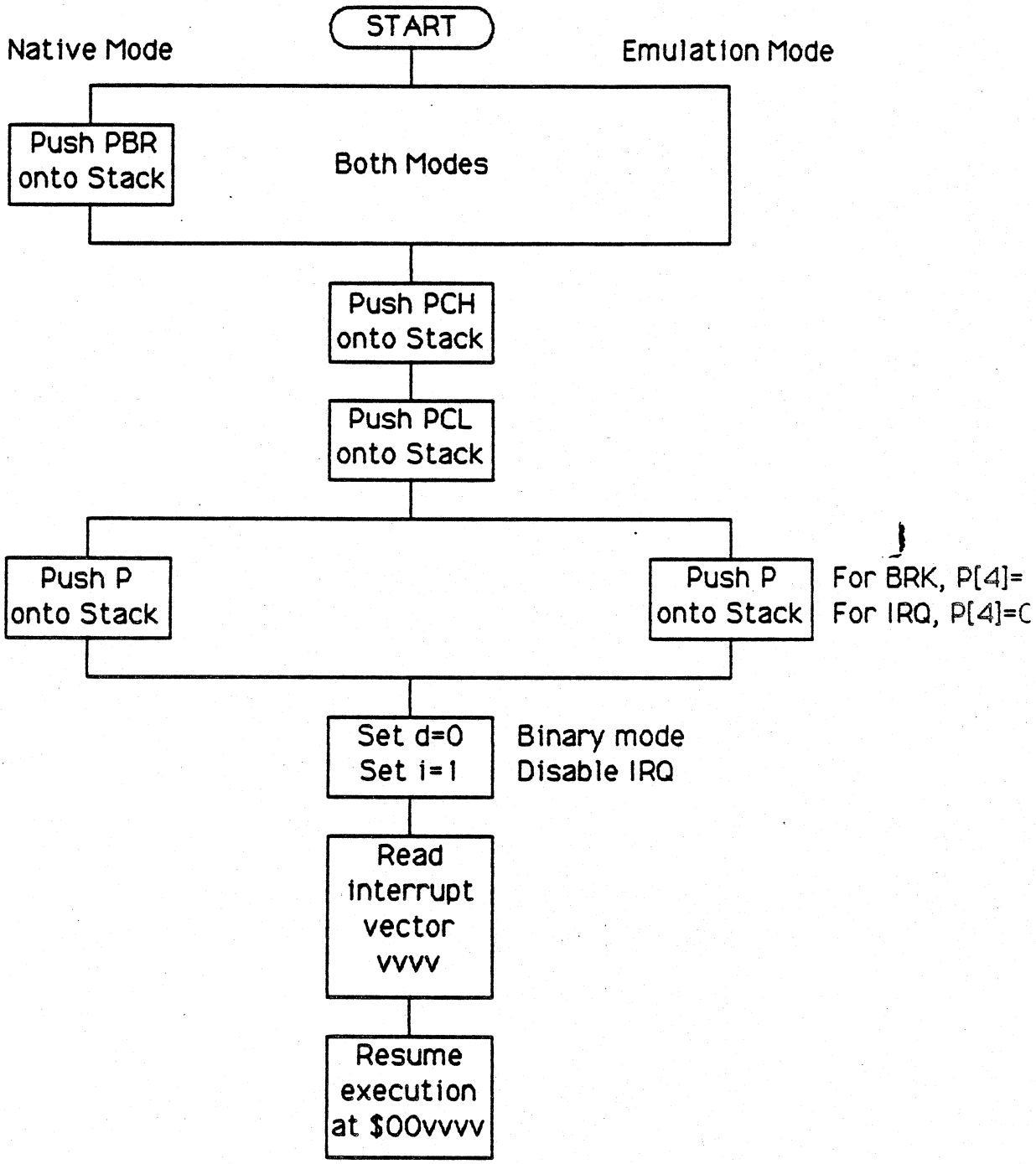
- There are few reasons for setting m=1 and/or x=1 in native mode. Almost any algorithm will be faster if it manipulates words rather than bytes.

CLASSIFICATION OF INTERRUPTS

Software Interrupts–Instructions in the normal instruction stream
- BRK–BReaK
- COP–"COProcessor" interrupt

Hardware Interrupts–External signals
- IRQ–Interrupt ReQuest
- NMI–Non-Maskable Interrupt
- ABORT–ABORT instruction such that it can be restarted
- RESET–RESET the processor into its initial state

INTERRUPT PROCESSING



Timing:
Emulation mode: 7 cycles
Native mode: 8 cycles

Note: ABORT pushes address of aborted instruction onto stack rather than address of next instruction.

INTERRUPT VECTOR LOCATIONS

Location	Interrupt
00FFFE,F	BRK and IRQ (emulation)
00FFFC,D	RESET (emulation and native)
00FFFA,B	NMI (emulation)
00FFF8,9	ABORT (emulation)
00FFF6,7	-
00FFF4,5	COP (emulation)
00FFF2,3	-
00FFF0,1	-
00FFEE,F	IRQ (native)
00FFEC,D	-
00FFEA,B	NMI (native)
00FFE8,9	ABORT (native)
00FFE6,7	BRK (native)
00FFE4,5	COP (native)

PROCESSOR STATE AFTER RESET

If the ~RES pin is held low for at least two clock cycles after VDD reaches operating voltage, the processor performs an initialization sequence which sets the registers, status bits, and control bits as follows:

Register	Initial Value
XH	$00
YH	$00
DB	$00
D	$0000
SH	$01
PB	$00

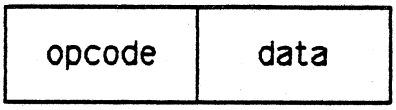
Control Bit	Initial Value	Meaning
e	1	Emulation mode
m	1	Always true for e=1
b, x	1	Break command
d	0	Binary mode
i	1	IRQ disabled

The remaining registers, status, and control bits are undefined on reset.

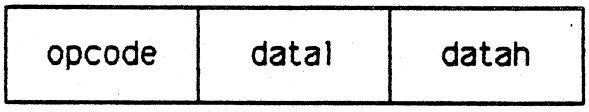
INTRAFAMILY COMPATIBILITY ISSUES

Issue	65802/65816	65C02	6502
Status and control bits in P register	n, v, z flags valid in decimal mode	n, v, z flags valid in decimal mode	n, v, z flags invalid in decimal mode
	0->d on reset or interrupt	0->d on reset or interrupt	reset: d undefined interrupt: d unchg.
Read/Modify/Write a,x no page cross Write Memory Lock	 7 cycles last 2 cycles last 3 cycles [rmw]	 6 cycles last cycle last 2 cycles [mw]	 7 cycles last two cycles not available
Jump indirect Cycles Operand=XXFF	 5 cycles Correct	 6 cycles Correct	 5 cycles Invalid page cross
Branch or index across page bound Execution time for such a branch	Read last program byte 4 cycles (e=1) 3 cycles (e=0)	Read last program byte 4 cycles	Read invalid address 4 cycles
Decimal mode ADC and SBC	No added cycles	Add 1 cycle	No added cycles
Unused opcodes	Only one, WDM, is a no-op	All are no-ops	Undefined-some hang processor

[1] IMMEDIATE-#



e=1 or e=0 and m/x=1



e=0 and m/x=0

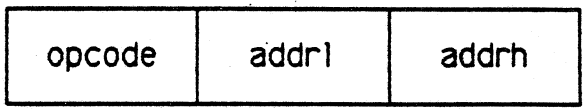
The 8-bit or 16-bit operand is taken from the instruction.

Examples:

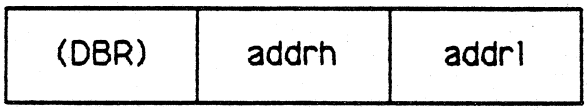
```
LDA #$E1
LDX #$FFFE
```

[2] ABSOLUTE-a

Instruction



Operand Address (data reference)



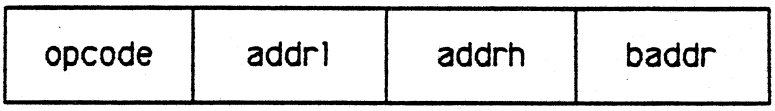
Operand Address (JMP or JSR)

Examples:

```
LDA LOCA
LDA |LOCB
JMP |DEST
```

[3] ABSOLUTE LONG-al

Instruction

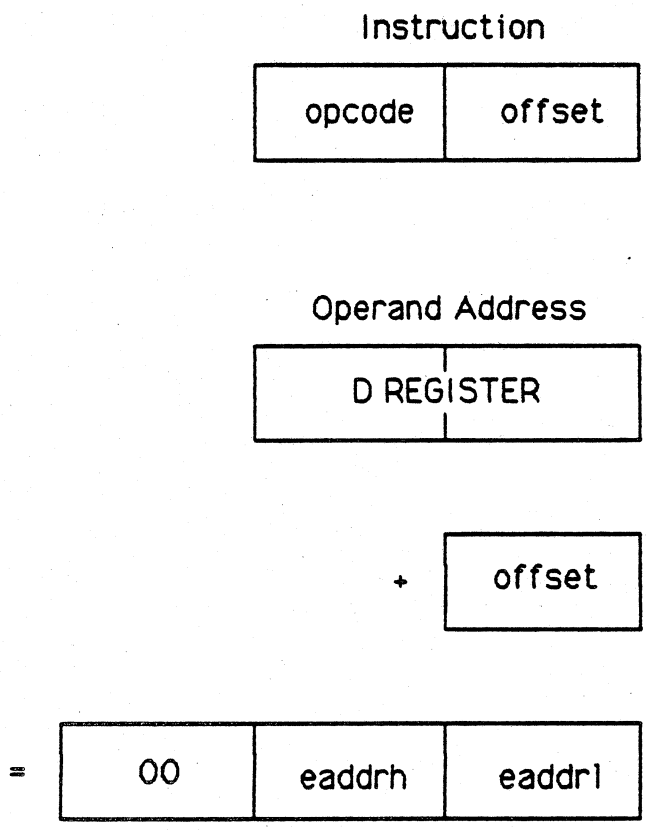


Operand Address

Examples:

```
AND LOC
AND >LOC
```

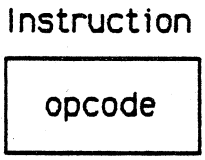
[4] DIRECT-d



Examples:

```
EOR LOC
EOR <LOC
```

[5] ACCUMULATOR-A

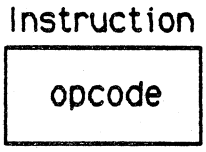


The operand is the accumulator.

Examples:

```
ASL A
INC A
```

[6] IMPLIED-i

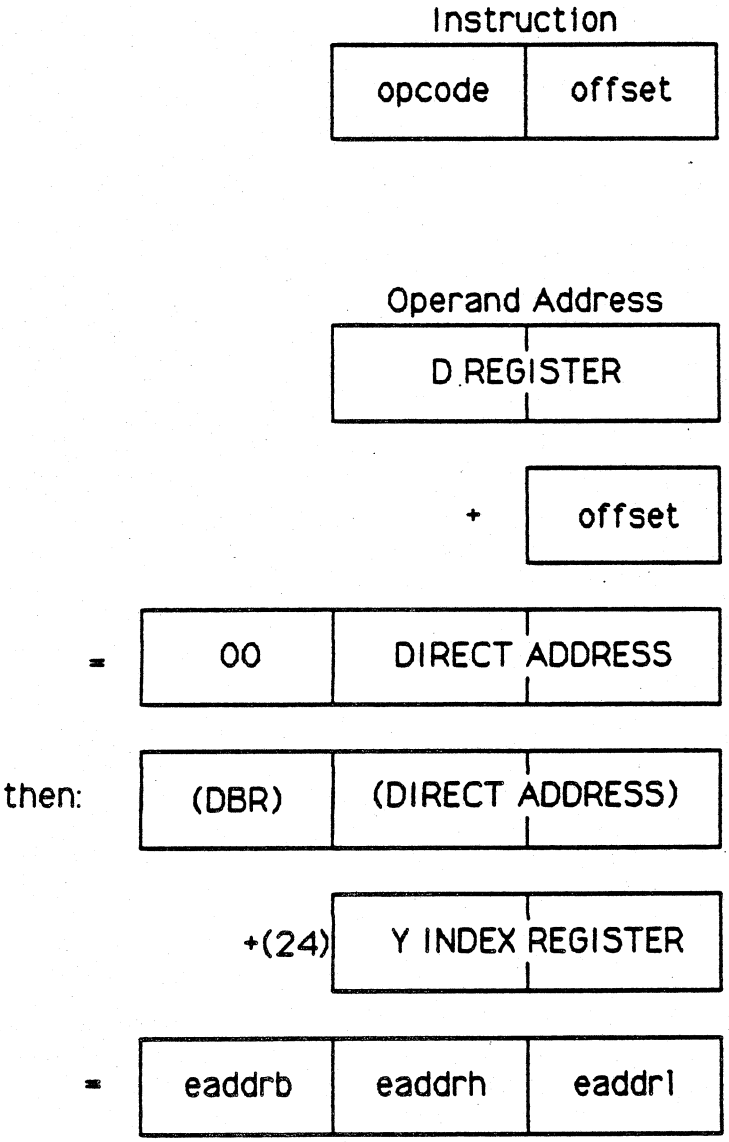


The operand is implied by the instruction.

Examples:

```
DEY
INX
SEC
TAY
```

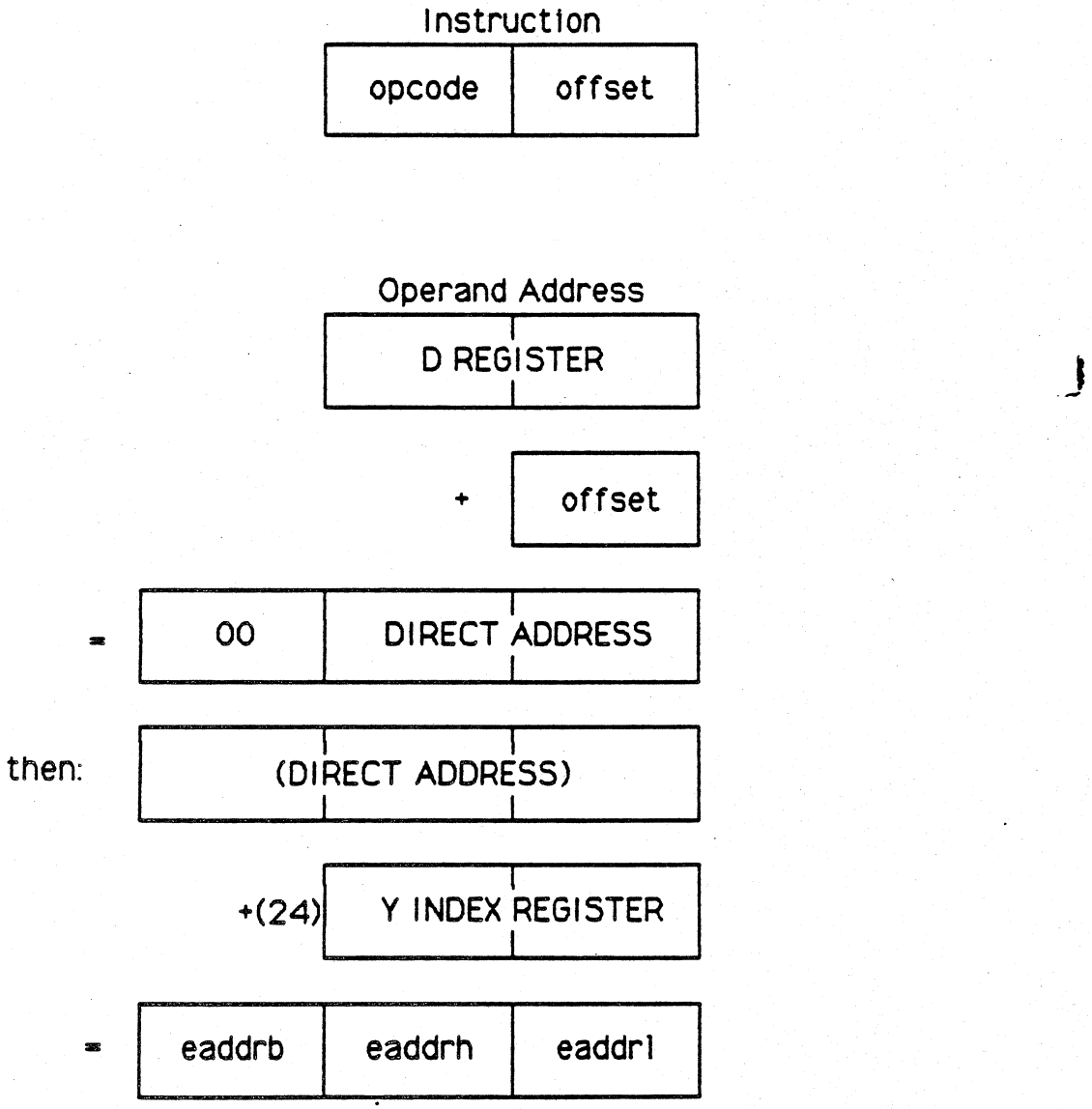
[7] DIRECT INDIRECT INDEXED–(d),y



Examples:

```
LDA (LOC),Y
LDA (<LOC),Y
```

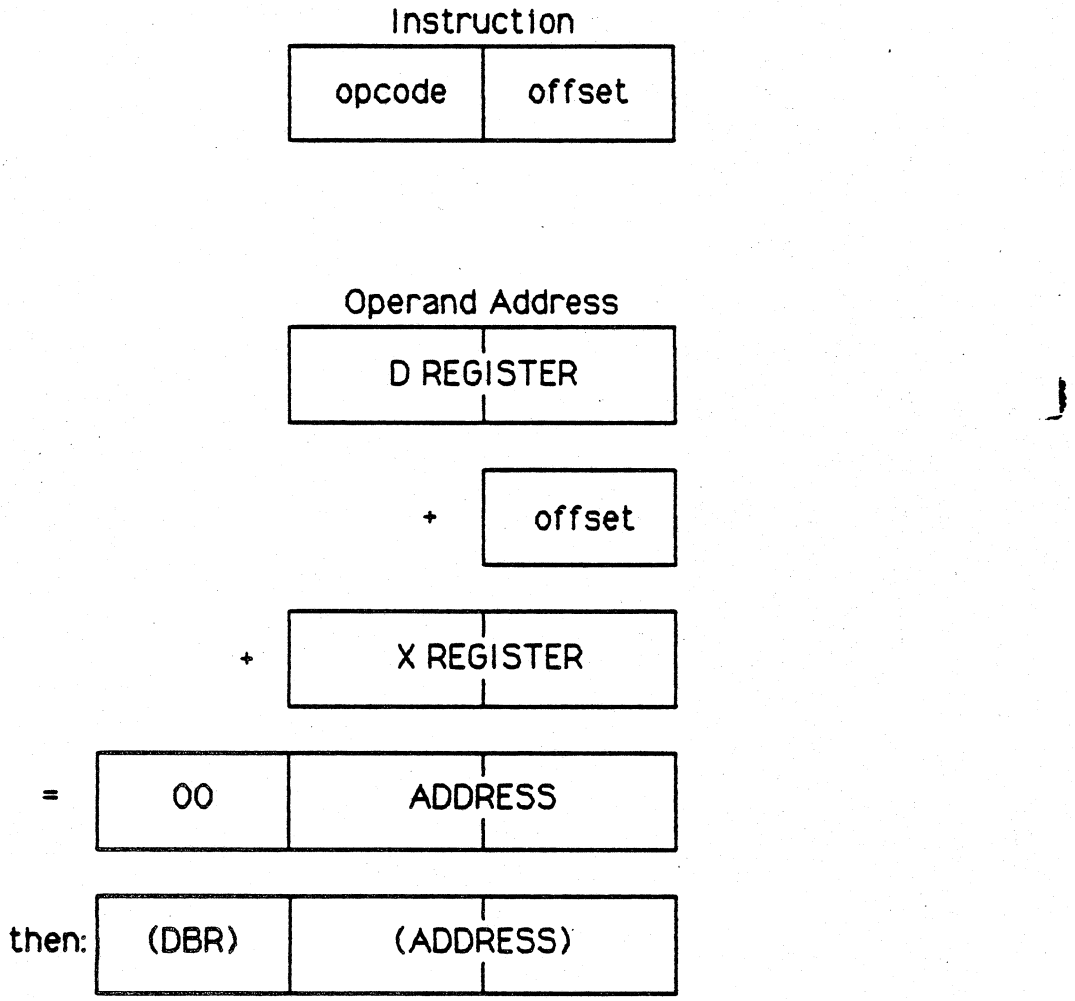
[8] DIRECT INDIRECT LONG INDEXED–[d],y



Examples:

```
LDA [LOC],Y
LDA [<LOC],Y
```

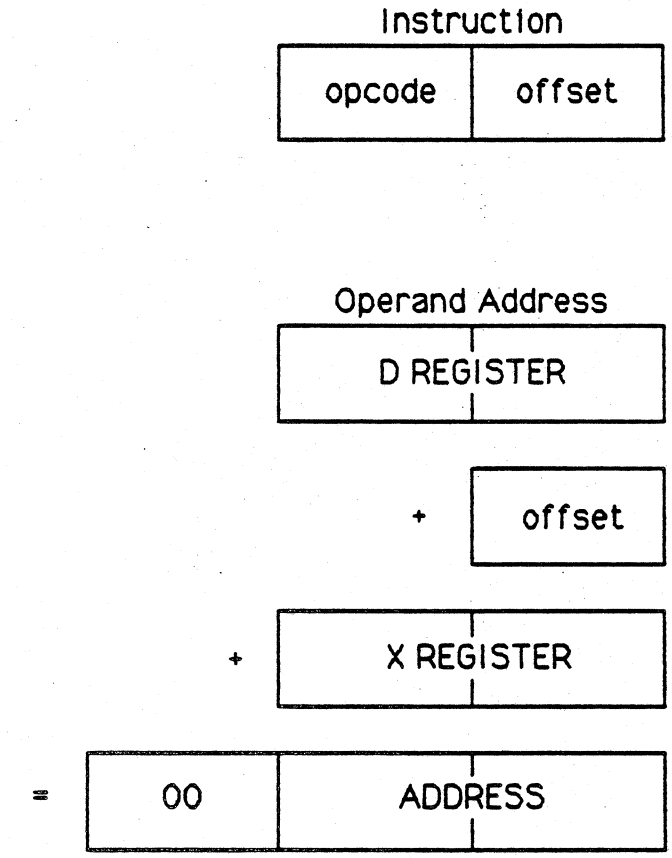
[9] DIRECT INDEXED INDIRECT–(d,x)



Examples:

```
LDA (LOC,X)
LDA (<LOC,X)
```

[10] DIRECT INDEXED WITH X-d,x
[11] DIRECT INDEXED WITH Y-d,y

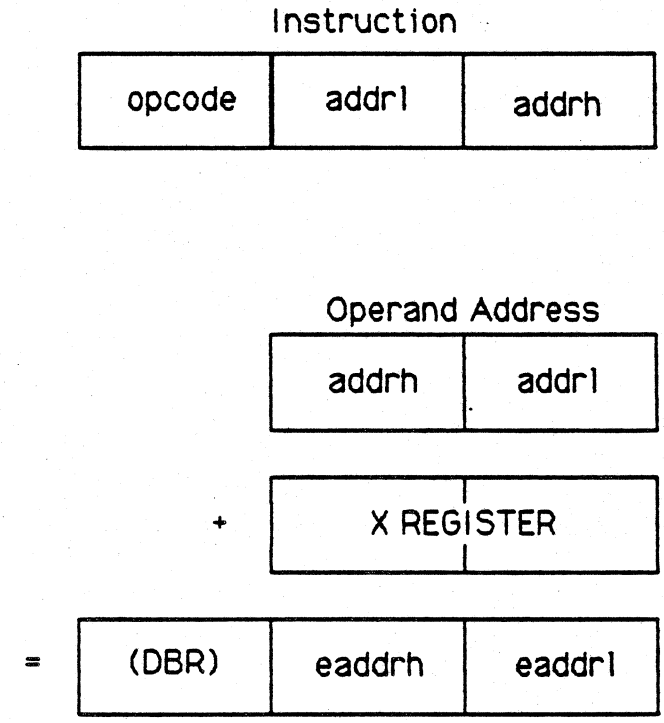


Note:
Mode 10 uses X REGISTER
Mode 11 uses Y REGISTER

Examples:

```
LDA LOC,X
LDA <LOC,X
LDA LOC,Y
LDA <LOC,Y
```

[12] ABSOLUTE INDEXED WITH X–a,x
[14] ABSOLUTE INDEXED WITH Y–a,y

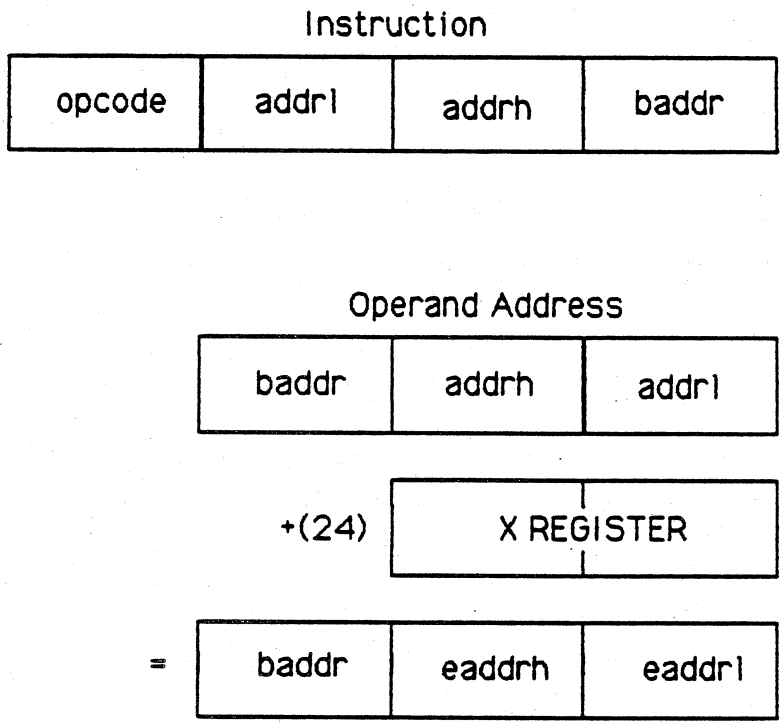


Note:
Mode 12 uses X REGISTER
Mode 14 uses Y REGISTER

Examples:

```
LDA LOC,X
LDA |LOC,X
LDA LOC,Y
LDA |LOC,Y
```

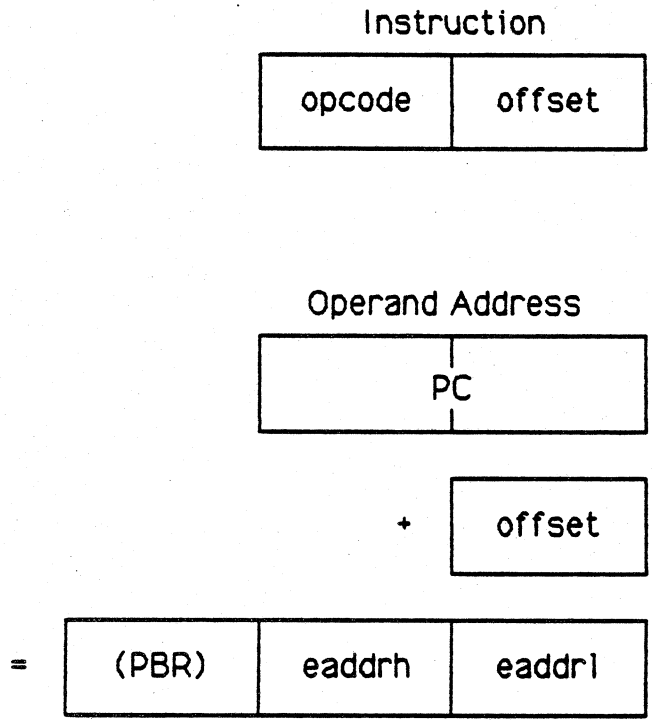
[13] ABSOLUTE LONG INDEXED WITH X-al,x



Examples:

```
LDA LOC,X
LDA >LOC,X
```

[15] PROGRAM COUNTER RELATIVE-r

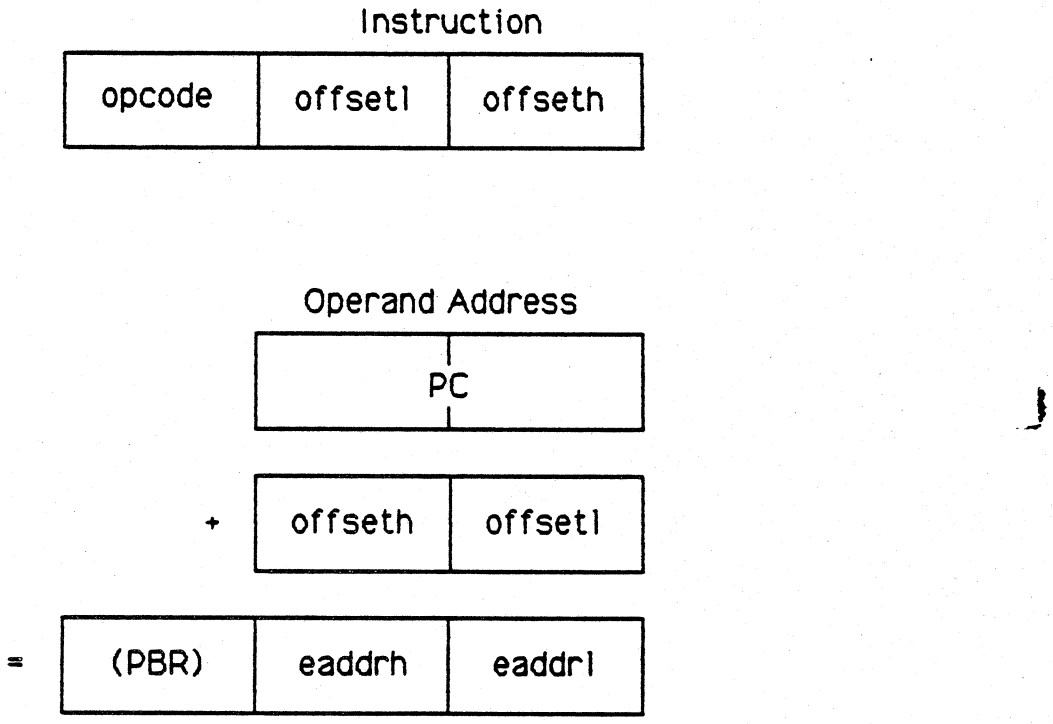


Offset is a two's complement number which is added to the contents of the PC, which have been updated to point to the opcode of the next instruction.

Examples:

```
BCC BLAH
BRA START
```

[16] PROGRAM COUNTER RELATIVE LONG-rl

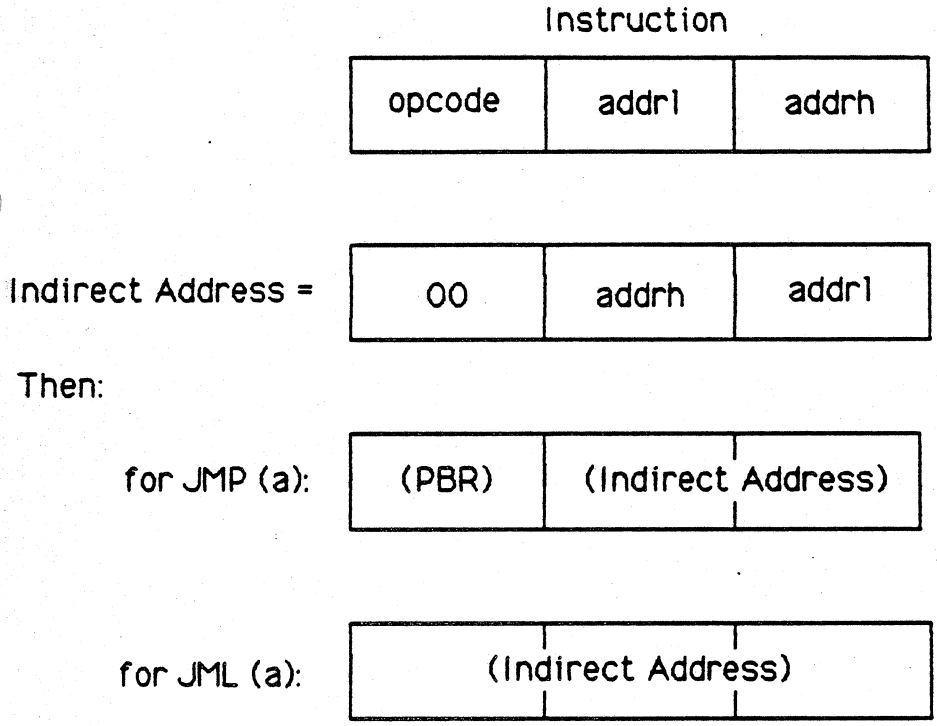


Offset is a two's complement number which is added to the contents of the PC, which have been updated to point to the opcode of the next instruction.

Examples:

```
BRL DEST
PER ENTRY
```

[17] ABSOLUTE INDIRECT-(a)

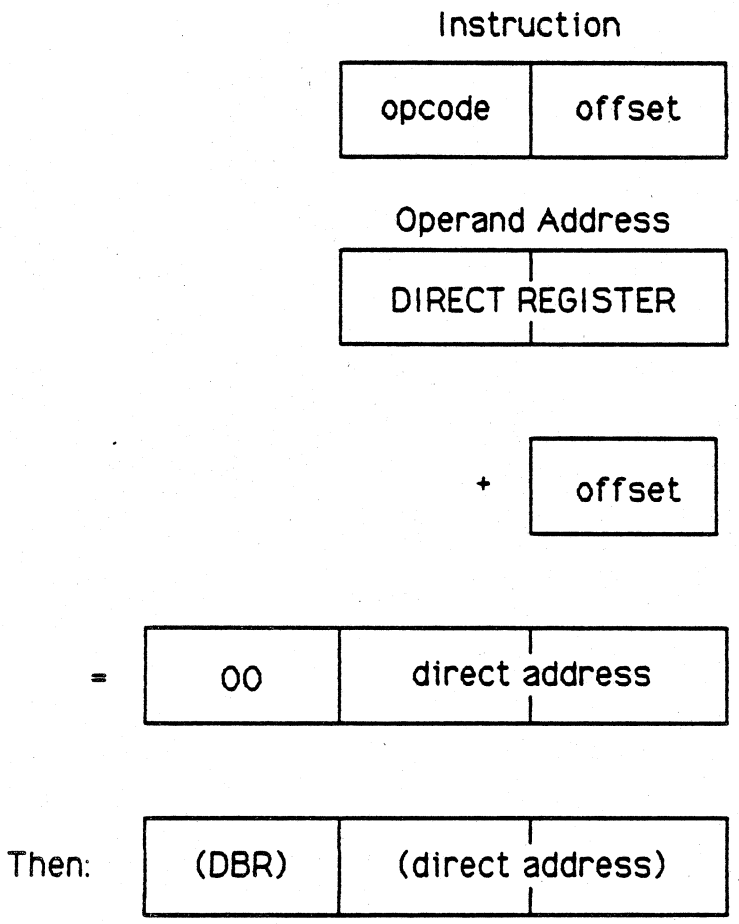


Mode 17 really constitutes two addressing modes, one used only by the instruction JMP (a) and one used only by the instruction JML (a). In the former case, the contents of the indirectly addressed location contain two bytes, and the PBR value remains unchanged. In the latter case, the indirectly addressed location contains all three bytes of the destination address.

Examples:

```
JMP (LOC)
JML (|LOC)
```

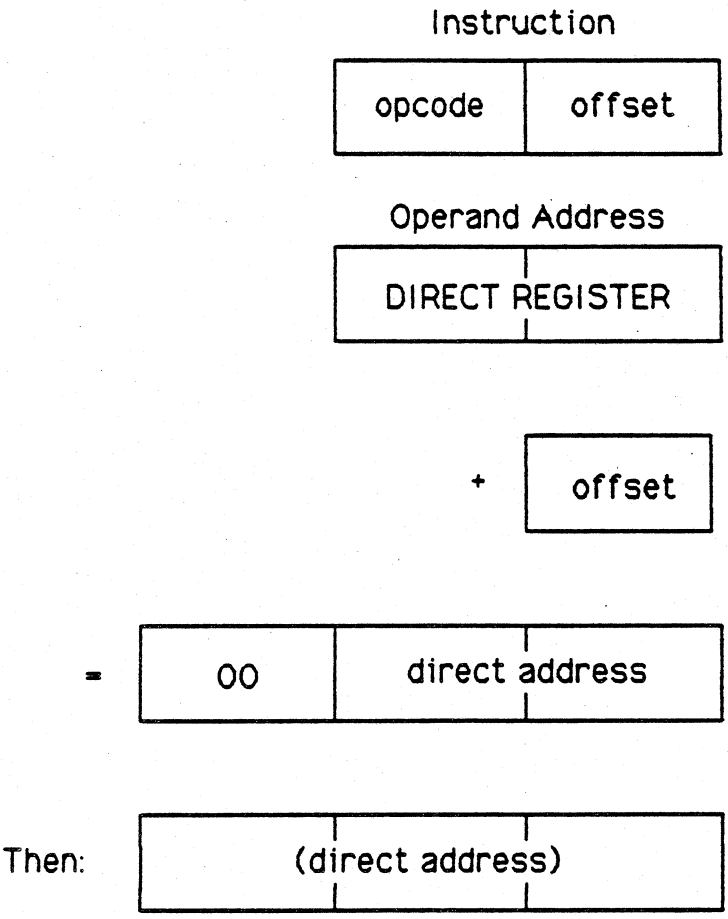
[18] DIRECT INDIRECT-(d)



Examples:

```
LDA (LOC)
LDA (<LOC)
```

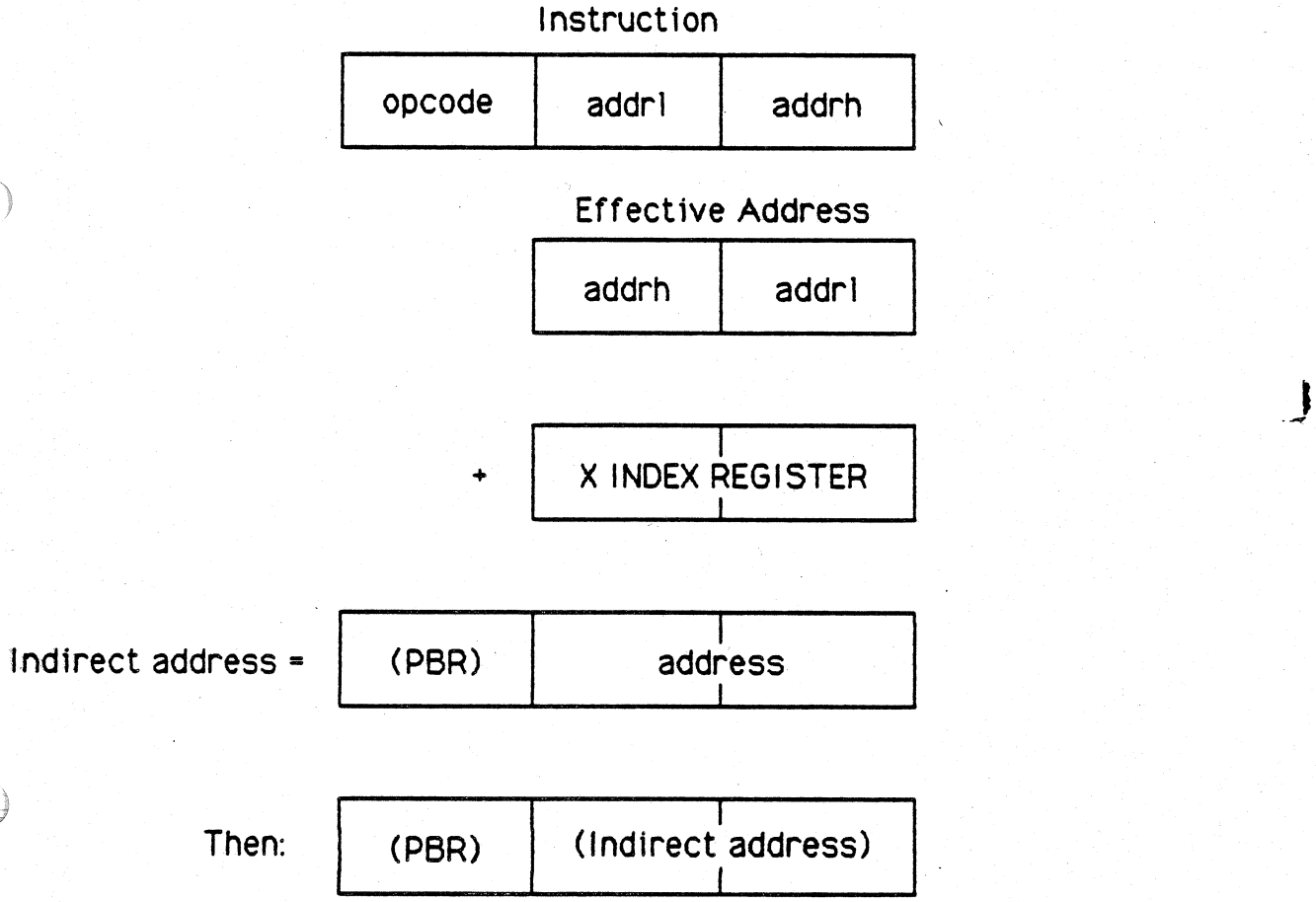
[19] DIRECT INDIRECT LONG-[d]



Examples:

```
LDA [LOC]
LDA [<LOC]
```

[20] ABSOLUTE INDEXED INDIRECT-(a,x)



Examples:

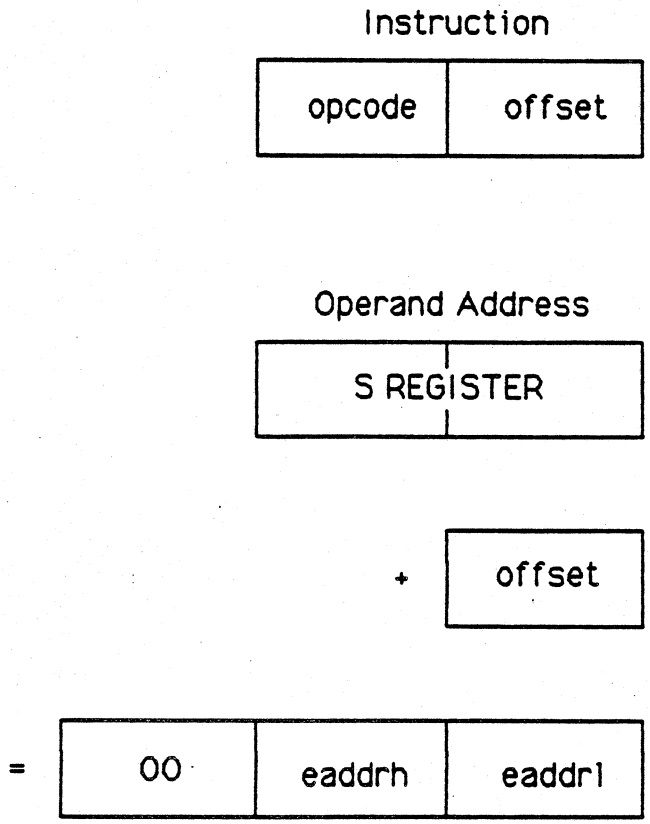
```
JMP (DEST,X)
JSR (IDEST,X)
```

[21] STACK-s

Stack addressing actually refers to a number of distinct addressing modes, each of which uses the stack somehow.

Instruction(s)	Operation
Hardware interrupts IRQ, NMI, ABORT, RES	Interrupts push PBR, PCH, PCL, and P onto the stack
Software interrupts BRK, COP	These interrupts push PBR, PCH, PCL, and P onto the stack
RTI	Pulls P, PCL, PCH, and PBR from the stack
RTS	Pulls PCL and PCH from the stack
RTL	Pulls PCL, PCH, and PBR from the stack
Register push and pull instructions	Push register contents onto the stack or pull top of stack element(s) into register
PEI	Push a word of direct (zero) page onto the stack
PEA	Push third and second bytes of instruction onto the stack. This is really a "push immediate" instruction.
PER	Push onto the stack the value obtained by adding the PC to the contents of bytes 3 and 2 of the instruction

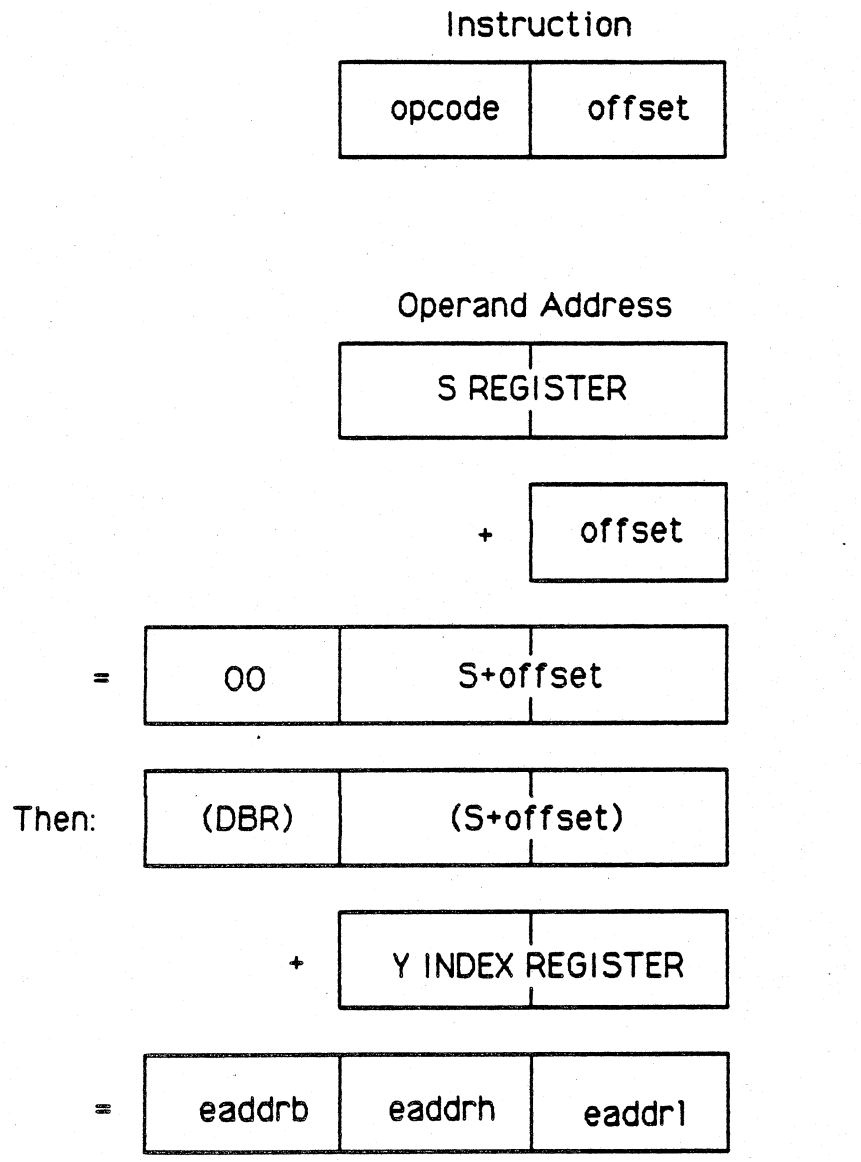
[22] STACK RELATIVE-d,s



Examples:

```
EOR $10,S
LDA <LOC,S
```

[23] STACK RELATIVE INDIRECT INDEXED-(d,s),y

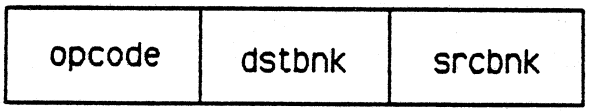


Examples:

```
ORA (LOC,S),Y
ORA (<LOC,S),Y
```

[24] BLOCK MOVE-xyc

Instruction

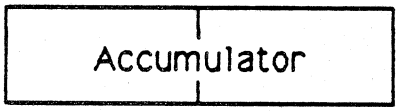


dstbnk -> DBR

Source:

Destination:

Count:



SUMMARY OF TIME AND STORAGE FOR EACH ADDRESSING MODE

Address Mode	Time (cycles)		Program Memory (by)	
	6502	65C816	6502	65C816
1. Immediate	2	2 (3)	2	2 (3)
2. Absolute	4 (5)	4 (3,5)	3	3
3. Absolute Long	-	5 (3)	-	4
4. Direct	3 (5)	3 (3,4,5)	2	2
5. Accumulator	2	2	1	1
6. Implied	2	2	1	1
7. Direct Indirect Indexed	5 (1)	5 (1,3,4)	2	2
8. Direct Indirect Indexed Long	-	6 (3,4)	-	2
9. Direct Indexed Indirect	6	6 (3,4)	2	2
10. Direct Indexed with X	4 (5)	4 (3,4,5)	2	2
11. Direct Indexed with Y	4	4 (3,4)	2	2
12. Absolute Indexed with X	4 (1,5)	4 (1,3,5)	3	3
13. Absolute Long Indexed X	-	5 (3)	-	4
14. Absolute Indexed with Y	4 (1)	4 (1,3)	3	3
15. Relative	2 (1,2)	2 (2)	2	2
16. Relative Long	-	3 (2)	-	3
17. Absolute Indirect	5	5	3	3
18. Direct Indirect	-	5 (3,4)	-	2
19. Direct Indirect Long	-	6 (3,4)	-	2
20. Absolute Indexed Indirect	-	6	-	3
21. Stack	3-7	3-8	1-3	1-4
22. Stack Relative	-	4 (3)	-	2
23. Stack Rel Indirect Indexed	-	7 (3)	-	2
24. Block Move	-	7	-	3

1. Page boundary: add 1 cycle if page boundary is crossed when forming address
2. Branch taken: add 1 cycle if branch is taken
3. m=0 or x=0, 16-bit operation: add 1 cycle, add 1 byte for immediate
4. DL≠0: add 1 cycle
5. Read-Modity-Write: add 2 cycles for m=1 and 3 cycles for m=0

RECOMMENDED 65816 ASSEMBLY LANGUAGE STANDARDS

- Refer to Western Design Center Data Sheet, pp. 17-18
- Byte selection of one or two bytes in immediate operands:

#$01020304	04 (one byte)	0304 (two bytes)
#<$01020304	04	0304
#>$01020304	03	0203
#^ $01020304	02	0102

- Forcing a specific address mode in an ambiguous situation:

< expression	forces direct (1 byte) addressing
| expression	forces absolute (2 byte) addressing
> expression	forces long absolute (3 byte) addressing

- When the mode is not forced and the context is ambiguous, the assembler will assume absolute (2 byte) addressing
- Long indirect addresses are indicated by square brackets—e.g. [d],y
- For MVN and MVP, the first operand is the source bank, and the second operand is the destination bank (contrary to the order of the object code bytes)
- There is a list of acceptable address mode formats on WDC p. 18
- There are several recommended mnemonic aliases. In each case, the leftmost mnemonic is considered standard:

BCC = BLT
BCS = BGE
CMP = CMA
DEC A = DEA
INC A = INA
JSL = JSR
JML = JMP
TCD = TAD
TCS = TAS
TDC = TDA
TSC = TSA
XBA = SWA

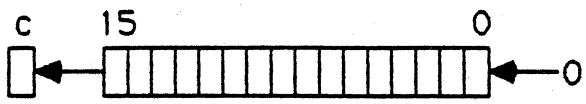
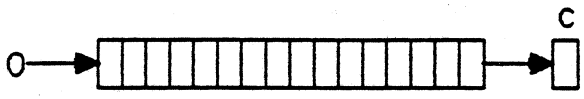
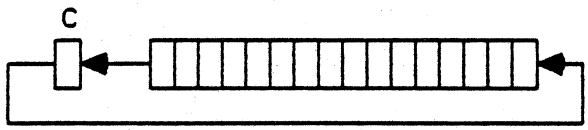
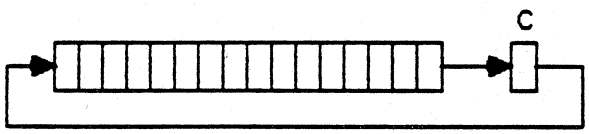
BIG EIGHT DATA MANIPULATION INSTRUCTIONS

INSTRUCTION CLASS	MNEM-ONIC	FUNCTION	FUNCTION
DATA MOVEMENT	LDA STA	W –> A A –> W	LoaD Accumulator STore Accumulator
ARITHMETIC	ADC SBC	A+W+c –> A A-W-~c –> A	ADd with Carry SuBtract with Carry
ARITHMETIC COMPARISON	CMP	A - W (set CCs)	CoMPare
LOGICAL	AND EOR ORA	A&W –> A A XOR W –> A A | W –> A	Logical AND Exclusive OR Logical OR (accumulator)

Each of these instructions has the following 15 address modes, except STA, which omits the immediate mode (#). This yields 119 distinct opcodes.

#	Immediate
a	Absolute
al	Absolute Long
d	Direct
(d),y	Direct Indirect Indexed
[d],y	Direct Indirect Indexed Long
(d,x)	Direct Indexed Indirect
d,x	Direct Indexed with X
a,x	Absolute Indexed with X
al,x	Absolute Long Indexed with X
a,y	Absolute Indexed with Y
(d)	Direct Indirect
[d]	Direct Indirect Long
d,s	Stack Relative
(d,s),y	Stack Relative Indirect Indexed

SHIFTS AND ROTATES

ASL		Arithmetic Shift Left
LSR		Logical Shift Right
ROL		ROtate Left
ROR		ROtate Right

Each of these instructions has the following 5 address modes, yielding 20 distinct opcodes.

A	Accumulator
d	Direct
d,x	Direct Indexed with X
a	Absolute
a,x	Absolute Indexed with X

INCREMENT AND DECREMENT ACCUMULATOR

INC	INCrement accumulator
DEC	DECrement accumulator

Each of these instructions has the following 5 addressing modes, yielding 10 distinct opcodes.

a	Absolute
d	Direct
A	Accumulator
d,x	Direct Indexed with X
a,x	Absolute Indexed with X

INDEX REGISTER MANIPULATION

INSTRUCTION CLASS	MNEM-ONIC	FUNCTION	FUNCTION	ADDRESS MODES
DATA MOVEMENT	LDX	W –> X	LoaD X	# a d d,y a,y
	LDY	W –> Y	LoaD Y	# a d d,x a,x
	STX	X –> W	STore X	a d d,y
	STY	Y –> W	STore Y	a d d,x
ARITHMETIC COMPARISON	CPX	X - W (set CC)	ComPare X	# a d
	CPY	Y - W (set CC)	ComPare Y	# a d
INCREMENT/ DECREMENT	DEX	X - 1 –> X	DEcrement X	i
	DEY	Y - 1 –> Y	DEcrement Y	i
	INX	X + 1 –> X	INcrement X	i
	INY	Y + 1 –> Y	INcrement Y	i

These instructions, in combination with their addressing modes, yield 26 distinct opcodes.

BIT TEST-BIT

Function: 1) A & W
2) W bit 15 -> n
3) W bit 14 -> v
4) z set according to result of A & W

Note: When immediate address mode is used, n and v are not set as shown above.

Address modes: # a d d,x a,x

STORE ZERO-STZ

Function: Store zero at the addressed location.

Address modes: a d d,x a,x

REGISTER PUSHES/PULLS AND REGISTER-REGISTER TRANSFERS

REGISTER PUSHES AND PULLS

Register	Push (cycles)	Pull (cycles)
A	PHA (4)	PLA (5)
X	PHX (4)	PLX (5)
Y	PHY (4)	PLY (5)
DB	PHB (3)	PLB (4)
S	-	-
D	PHD (4)	PLD (5)
PB	PHK (3)	-
PC	-	-
P	PHP (3)	PLP (4)

REGISTER-REGISTER TRANSFERS

Source Register	Destination Register				
	A	X	Y	S	D
A	XBA*	TAX	TAY	TCS (TAS)	TCD (TAD)
X	TXA	-	TXY	TXS	-
Y	TYA	TYX	-	-	-
S	TSC (TSA)	TSX	-	-	-
D	TDC (TDA)	-	-	-	-

All register-register transfers take 2 cycles, except XBA, which takes 3 cycles..

* Swaps the upper and lower bytes of the accumulator

SPECIAL PUSH INSTRUCTIONS: PEA, PEI, PER

PEA–Push Effective Absolute Address onto Stack
(Should be called "Push Immediate Data onto Stack")

opcode	datal	datah

1) Push third byte of instruction (datah) onto stack
2) Push second byte of instruction (datal) onto stack

Time: 5 cycles

PEI–Push Effective Indirect Address onto Stack
(Should be called "Push Zero Page Word onto Stack")

opcode	offset

1) eaddr = (D Register) + offset
2) push contents of location eaddr+1 onto stack
3) push contents of location eaddr onto stack

Time: 6 cycles (DL=0)
7 cycles (DL≠0)

PER–Push Effective Program Counter Relative Address onto Stack

opcode	offsetl	offseth

1) Let offset = offseth concatenated with offsetl
2) value = (Program Counter) + offset
3) push high order byte of value onto stack
4) push low order byte of value onto stack

Time: 6 cycles

Mnemonic	Function	Function
TRB	Test and Reset Bit	~A&W –> W; set z on result of A&W
TSB	Test and Set Bit	A|W –> W; set z on result of A&W
Note: the Memory Lock (ML) signal is active during the entire read-modify-write phase of these instructions.		

Addressing modes: d a

Example:

```
TRB LOC
TSB <LOC
```

STATUS REGISTER MANIPULATION INSTRUCTIONS

INDIVIDUAL BIT SETS AND CLEARS

	Carry	Decimal	IRQ Disable	Overflow
Set	SEC	SED	SEI	-
Clear	CLC	CLD	CLI	CLV
All of these instructions take 2 cycles.				

SET AND RESET STATUS BITS INSTRUCTIONS

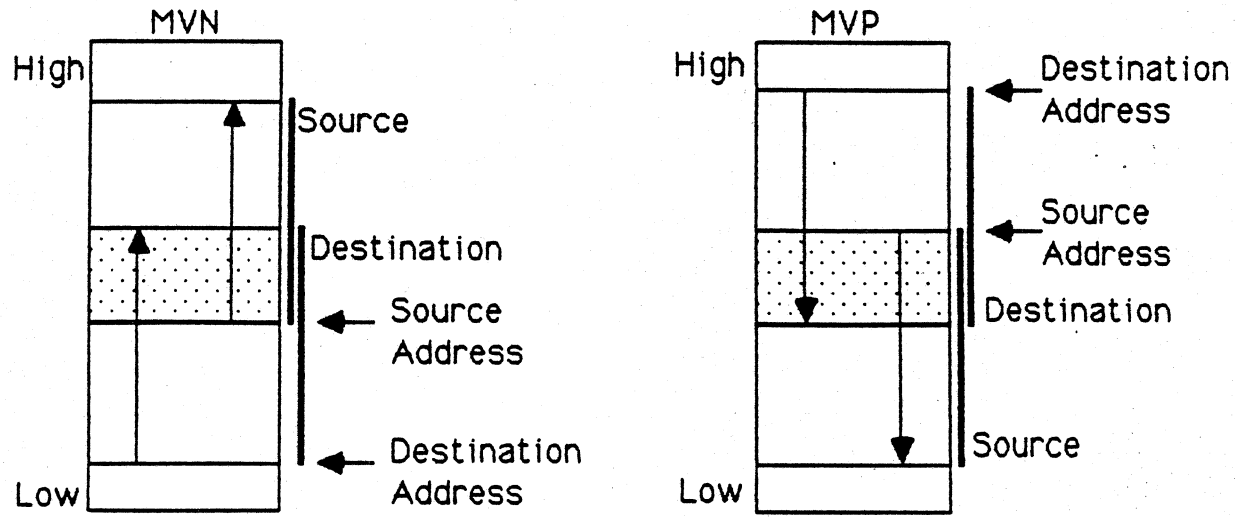
Mnemonic	Function	Function
REP	Reset processor status bits	P&¬B->P
SEP	Set processor status bits	PvB->P
These instructions take 3 cycles.		

XCE–EXCHANGE CARRY AND STATUS BITS

This instruction is used to switch the processor between native mode and emulation mode. To switch into native mode, set c to 0 and execute XCE. To switch to emulation mode, set c to 1 and execute XCE. Whenever e is set to 1, certain registers and status bits are set to the indicated states:

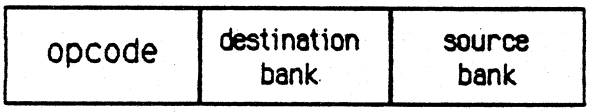
XH=0 YH=0 SH=1 m=1 x=1

BLOCK MOVES: MVN AND MVP



Arrows indicate order in which bytes are copied

Instruction format



Instruction setup

X Register—Low order 16 bits of source address
Y Register—Low order 16 bits of destination address
C Register—Number of bytes to move less 1

Restrictions

Neither source nor destination block may straddle bank boundary
Maximum block size: 65,536 bytes

Side effects

Previous contents of DBR are replaced by destination bank number

Timing: 7 cycles per byte

Approximate time to move 64Kby in //-16: 164 ms

BRANCH INSTRUCTIONS

Mnemonic	Function	Condition Code
BCC	Branch Carry Clear	c=0
BCS	Branch Carry Set	c=1
BEQ	Branch EQual (zero)	z=1
BMI	Branch MInus	n=1
BNE	Branch Not Equal (zero)	z=0
BPL	Branch PLus	n=0
BVC	Branch oVerflow Clear	v=0
BVS	Branch oVerflow Set	v=1
BRA	BRanch Always	(unconditional)
BRL	BRanch (always) Long	(unconditional)

Timing:
2 cycles–branch not taken
3 cycles–branch taken (most cases)
4 cycles–in emulation mode (e=1), branch taken across page boundary

All instructions except BRL consist of an opcode followed by a one-byte offset that represents a two's complement number in the range -128...127. If the branch is taken, this value is added to the Program Counter, which by this time, points to the opcode of the next sequential instruction.

BRL consists of the opcode followed by a two-byte offset that represents a two's complement number in the range -32,768...32767. As for the other instructions, this value is added to the Program Counter.

JUMP, SUBROUTINE JUMP, AND RETURN INSTRUCTIONS

Mnemonic	Function	Time (cycles)	Size (bytes)
JMP a	JuMP	3	3
JMP al	JuMP	4	4
JMP (a)	JuMP	5	3
JML (a)	JuMp Long	6	3
JMP (a,x)	JuMP	6	3
JSR a	Jump to SubRoutine	6	3
JSL al	Jump to SubRoutine Long	8	4
JSR (a,x)	Jump to SubRoutine	6	3
RTS	ReTurn from Subroutine	6	1
RTL	ReTurn from subroutine Long	6	1

Note:
JSR a and JSR (a,x) push PC onto the stack.
RTS should be used to return from JSR a and JSR (a,x).
JSL al pushes PB and PC onto the stack.
RTL should be used to return from JSL al.

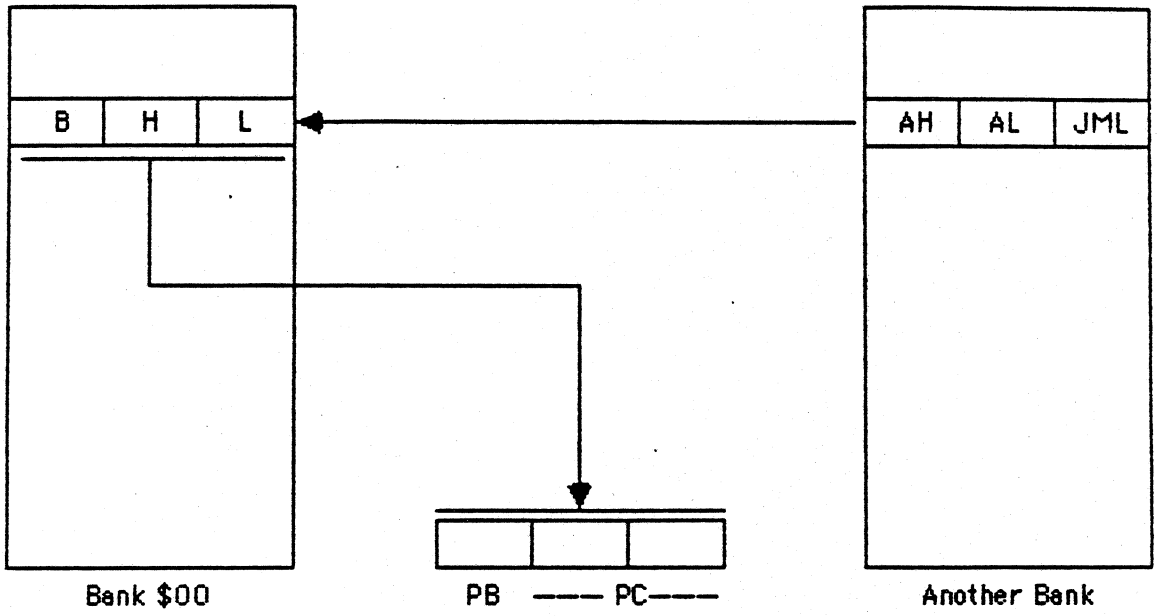


Figure 1. Effective Address Formation for JML (a)

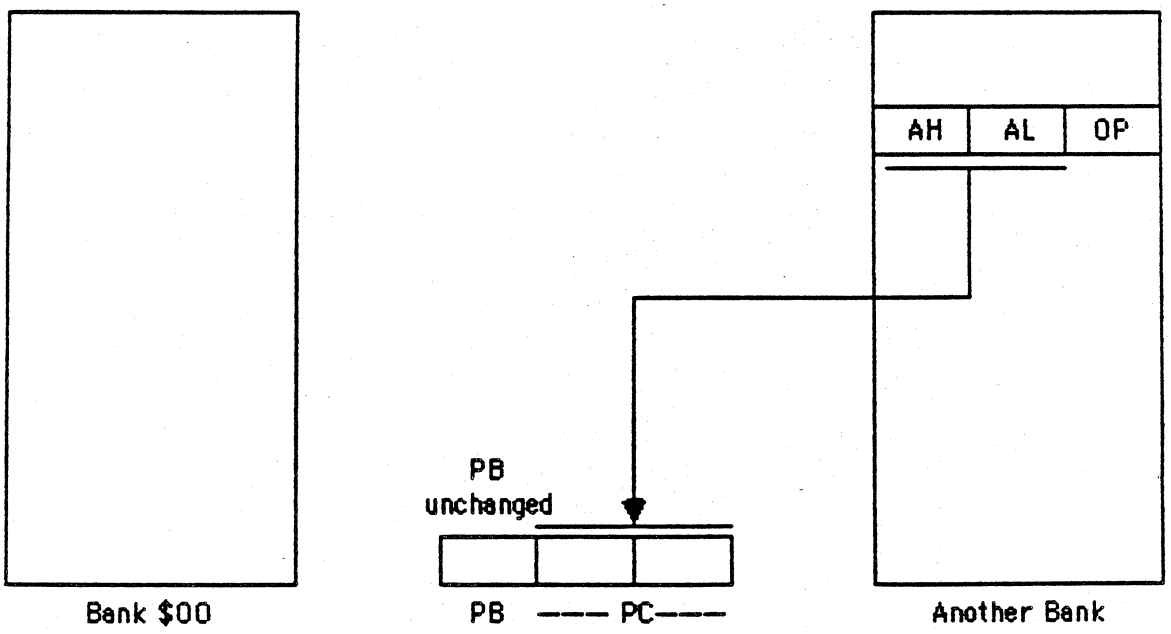


Figure 2. Effective Address Formation for JMP a and JSR a

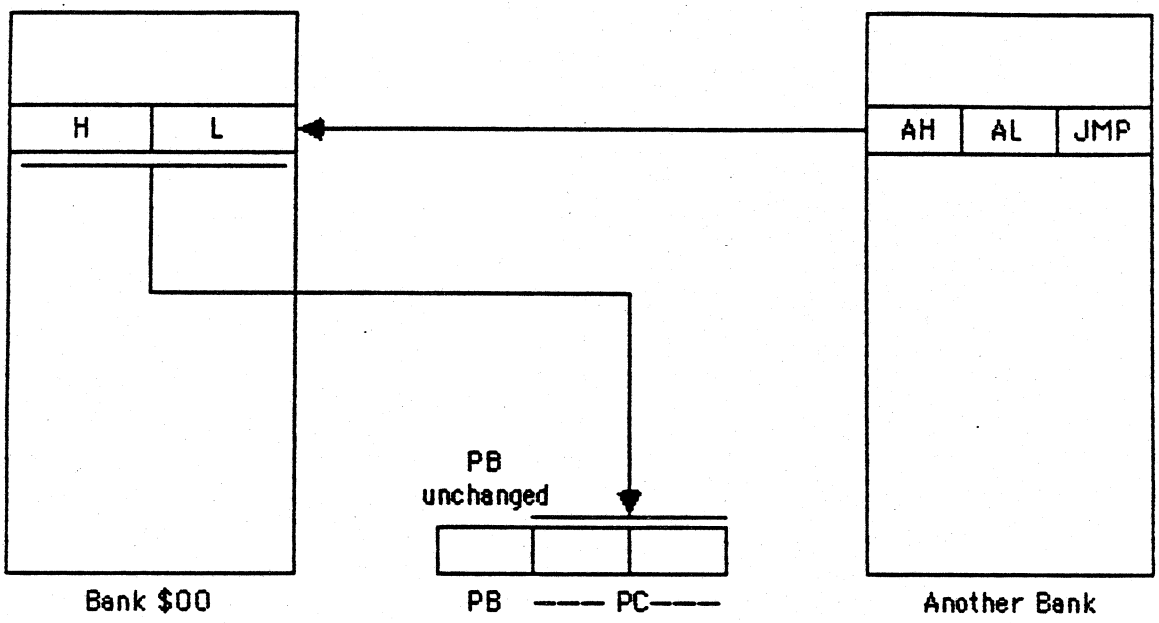


Figure 3. Effective Address Formation for JMP (a)

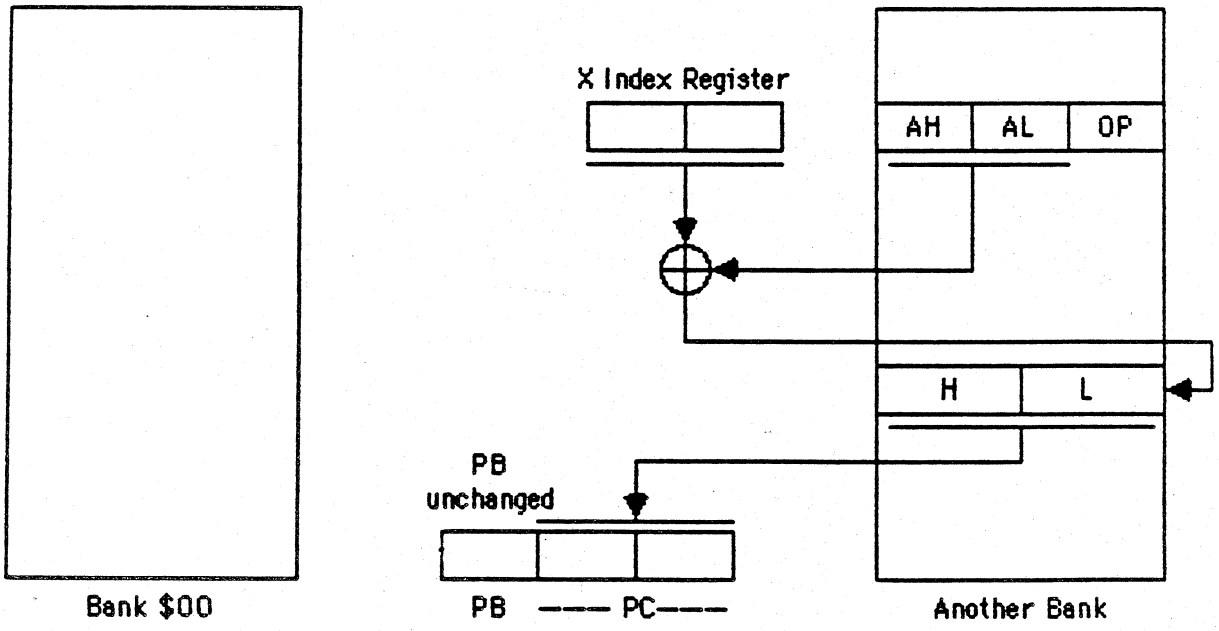


Figure 4. Effective Address Formation for JMP (a,x) and JSR (a,x)

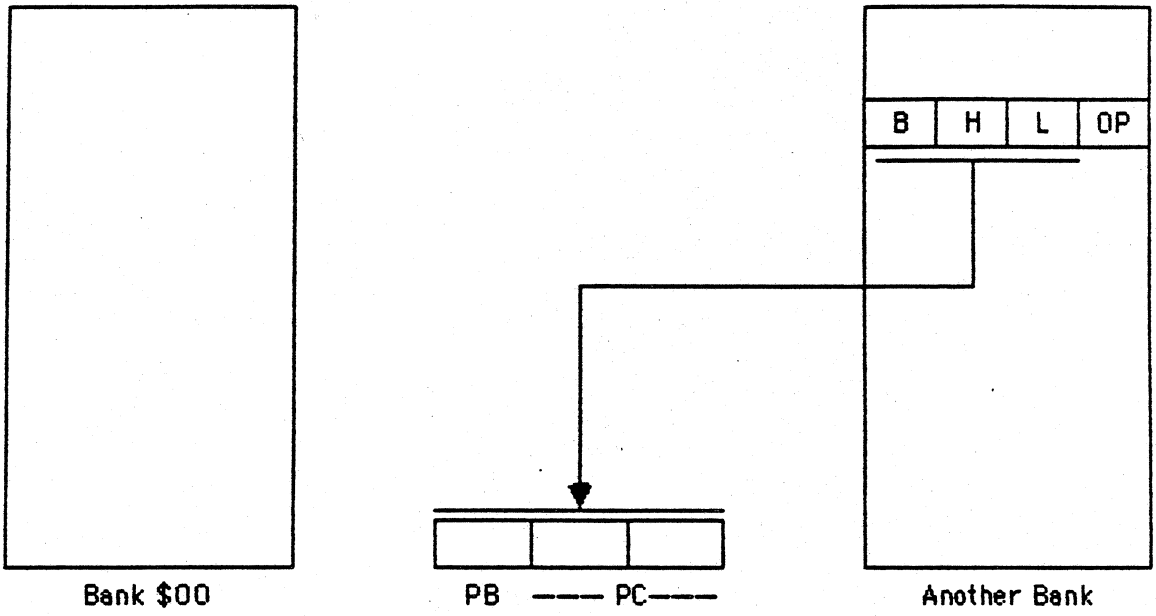
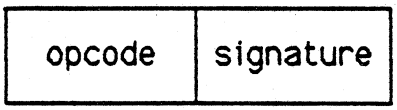


Figure 5. Effective Address Formation for JMP al and JSL al

BRK and COP are software interrupts. The instruction format for both is:



Each instruction initiates a standard interrupt sequence with the value of the signature byte placed on the data bus during cycle 2. Each software interrupt instruction has its own interrupt vector at the top of bank $00.

The state information saved on the stack, in the order that bytes are pushed, consists of PBR, PCH, PCL, and P.

The P register is modified so that d=0 and i=1.

Time: 8 cycles

RTI–RETURN FROM INTERRUPT

1) Pull P from stack
2) Pull PCL from stack
3) Pull PCH from stack
4) Pull PBR from stack (only in native mode)

Time: 7 cycles

STP–STOP THE PROCESSOR

Stop the processor clock to reduce power consumption.
Requires a RESET interrupt to continue execution.

WAI–WAIT FOR INTERRPUT

Wait for interrupt in a state that reduces power consumption and minimizes interrupt latency.
RDY line held low until interrupt occurs.

NOP–NO OPERATION

Don't do anything, but take 1 byte and 2 cycles not to do it.

WDM–WILLIAM D. MENSCH RESERVED OPCODE

RESERVED FOR FUTURE EXPANSION OF OPCODE SET.

BASIC INSTRUCTION TIMING

[1] Instruction execution times are expressed in machine cycles.

[2] The duration of a cycle depends on several factors:
- System clock speed (e.g. 1MHz vs high speed)
- Operations that cause cycle stretching (e.g. references to $CXXX space)

[3] Phoenix Apple // simulation mode:
- ~ 1µs/cycle
- ~ 1 MHz cycle rate

[4] Phoenix high speed mode:
- ~ 385 ns/cycle
- ~ 2.6 MHz cycle rate

[5] QUICK AND DIRTY ESTIMATION OF INSTRUCTION EXECUTION TIME
- Count 1 cycle for each byte in the instruction
- Count 1 cycle for each byte of data fetched from memory
- Count 1 cycle for each byte of data written to memory
- For any addressing mode involving the direct register, count 1 cycle if DL≠0
- For most stack operations, add a cycle (used to adjust value of stack pointer)
- NOTE THE MINIMUM EXECUTION TIME OF 2 CYCLES
- For exact timing, use the charts in the GTE and WDC data sheets

CYCLE-BY-CYCLE TIMING–(d),y

Cycle	VDA	VPA	Address Bus	Data Bus	Comment
1	1	1	PBR,PC	opcode	Fetch opcode
2	0	1	PBR,PC+1	DO	Fetch direct offset (DO)
2a [2]	0	0	PBR,PC+1	IO	Computer D+DO
3	1	0	0,D+DO	AAL	Get LSB of indirect addr.
4	1	0	0,D+DO+1	AAH	Get MSB of indirect addr.
4a [4]	0	0	0,D+DO+1	IO	Add index to indirect addr.
5	1	0	DBR,AA+Y	Data Low	Fetch LSB of data
5a [1]	1	0	DBR,AA+Y+1	Data High	Fetch MSB of data

[1] Cycle executed only for 16-bit operand fetch.
[2] Cycle executed only for DL=0.
[4] Cycle executed for indexing across page boundary or x=0.

NOTE DETAILED TIMING SPECIFICTIONS:

- WDC Data Sheet, pages 13, 15-16, 19, especially footnotes, page 16
- GTE Data Sheet, pages 13-16, expecially footnotes, page 13

NOTE WELL:

- ALL OF THE TIMING SPECIFICATIONS IN THE DATA SHEETS ARE INCOMPLETE OR INCORRECT IN SOME WAY. THE MOST CORRECT SPECIFICATION SEEMS TO BE THAT ON PAGES 15-16 IN THE WDC DATA SHEET.

POSITION INDEPENDENT JUMPS

Unconditional Jumps

The absolute jump (JMP a) is obsolete because there is a long relative branch (BRL rl) that can be used to pass control unconditionally to any location in the current program bank.

The advantage of using BRL is that it is "position independent," that is, it still works properly even if the program is moved.

BRL occupies 3 bytes and takes 3 cycles, just like JMP a.

Conditional Jumps

Short conditional jumps simply use the conditional branch instructions, e.g.

```
        BEQ BLAH
```

For more distant targets, reverse the branch condition and follow the reversed branch with a BRL to the destination. To get the same effect as above with a distant branch target, use:

```
        BNE AROUND
        BRL BLAH
AROUND
```

POSITION INDEPENDENT SUBROUTINE CALL

We can synthesize a position independent subroutine call to any location in the current program bank. Consider the call-return sequence:

```
        JSR SUBR
RETURN

        •  •  •

SUBR    RTS
```

The position independent version is as follows:

```
        PER RETURN-1-*   ;push return address-1
        PER SUBR-1-*     ;push address of subroutine-1
        RTS              ;jump to the subroutine
RETURN

        •  •  •

SUBR    RTS
```

PERFORMANCE COMPARISON

Version	Size (bytes)	Time (cycles)
Standard	4	12
Position Independent	8	24

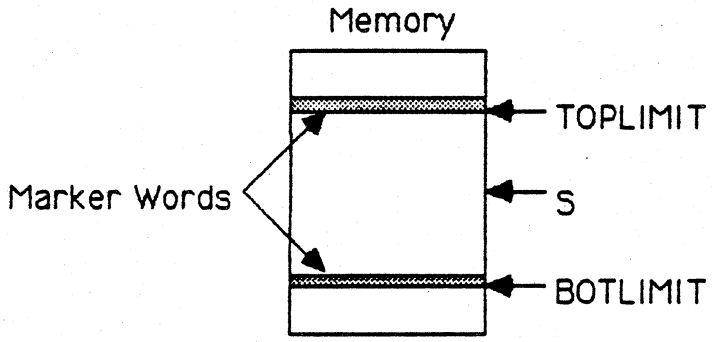
"POSITION INDEPENDENT" DATA REFERENCES

- Unnecessary to change code when data area is moved from one memory location to another.
- Only a pointer to the base of the data area needs to be updated.
- Technique used in machine code produced by most high level languages.

```
DATAPTR     DS   4          ;pointer (on zero page) to data area
XOFFSET     EQU  0          ;offset of X in data area
YOFFSET     EQU  2          ;offset of Y in data area
ZOFFSET     EQU  4          ;offset of Z in data area

;set up data area pointer in DATAAREA
;typically done by the code that handles a procedure call

            LDA #PTRLOW
            STA DATAPTR
            LDA #PTRHIGH
            STA DATAPTR+2

;add X to Y and put result in Z (16-bit integer addition)
;use direct indirect indexed long addressing

            LDY #XOFFSET
            LDA [DATAPTR],Y   ;get X
            CLC
            LDY #YOFFSET
            ADC [DATAPTR],Y   ;add Y
            LDY #ZOFFSET
            STA [DATAPTR],Z   ;store in Z
```

STACK OVERFLOW/UNDERFLOW DETECTION

- Stack overflow/underflow detection must be done by software since there are no hardware facilities to detect these conditions.
- Checking on every stack operation is too expensive.
- Therefore, periodically check the value of S and integrity of stack end markers.




```
PATTERN   EQU  $ACAC

INITIALIZE LDA  #PATTERN       ;initialize pattern in marker words
           STA  TOPLIMIT
           STA  BOTLIMIT
           LDX  TOPLIMIT-1     ;initialize stack pointer
           TXS
            •          •          •
CHECK      LDA  #PATTERN       ;check top and bottom marker words
           CMP  TOPLIMIT       ;for integrity
           BNE  STACKUNDERFLOW
           CMP  BOTLIMIT
           BNE  STACKOVERFLOW
           TSC                 ;check stack pointer between TOPLIMIT-1
           CMP  #TOPLIMIT      ;and BOTLIMIT+2 inclusive
           BCS  STACKUNDERFLOW
           CMP  #BOTLIMIT+2
           BCC  STACKOVERFLOW
;high probability that stack is OK if we get to this point
```

PUSH AND PULL THREE-BYTE POINTERS

PUSH A THREE-BYTE POINTER FROM ZERO PAGE ONTO THE STACK

```
LOC  DS   3          ;assume a zero page location

     PEI  LOC+1      ;push two high order bytes
     PHB             ;dummy push of DBR to decrement S by 1
     LDA  LOC        ;get two low order bytes
     STA  1,S        ;store two low order bytes on stack
```

Note: Weird as it may be, the middle byte of LOC is saved twice.

Time: 18 cycles (19 cycles if DL≠0)
Space: 7 bytes

PULL A THREE-BYTE POINTER FROM THE STACK INTO A ZERO PAGE LOCATION

```
LOC  DS   3          ;assume a zero page location

     PHB             ;push DBR in order to add 1 more byte to stack
     PLA             ;pull dummy byte and low order byte
     XBA             ;put low order byte in low order side of A
     STA  LOC        ;low order byte now in right place
     PLA             ;get two high order bytes
     STA  LOC+1      ;store at two high order bytes of LOC
```

Time: 24 cycles (26 cycles if DL≠0)
Space: 8 bytes

PUSH AND PULL FOUR-BYTE POINTERS

PUSH A FOUR-BYTE POINTER FROM ZERO PAGE ONTO THE STACK

```
LOC  DS   4          ;assume a zero page location

     PEI  LOC+2      ;push two high order bytes
     PEI  LOC        ;push two low order bytes
```

Time: 12 cycles (14 cycles if DL≠0)
Space: 4 bytes

PULL A FOUR-BYTE POINTER FROM THE STACK INTO A ZERO PAGE LOCATION

```
LOC  DS   4          ;assume a zero page location

     PLA             ;pull two low order bytes
     STA  LOC        ;store them
     PLA             ;pull two high order bytes
     STA  LOC+2      ;store them, too
```

Time: 18 cycles (20 cycles if DL≠0)
Space: 6 bytes