In this example, it is seen that generation of a carry from the add operation will allow an out-of-sequence branch to a new location.

4.1.1 Basic Concept of Relative Addressing

If one considers that the instruction JMP required three bytes, one for OP CODE, one for new program counter low (PCL), and one for new program counter high (PCH), it is seen that jump on carry set would also require three bytes. Because most programs for control require many continual jumps or branches, the R6500 Series uses "relative" addressing for all conditional test instructions. To perform any branch, the program counter must be changed. In relative addressing, however, we add the value in the memory location following the OP CODE to the program counter. This allows us to specify a new program counter location with only two bytes, one for the OP CODE and one for the value to be added.

To illustrate this, in the following example, the branch on carry set (BCS) illustration is followed by a value of 50. If the carry is set, the new program location would be 108 + 50 = 158; in other words, it will take the branch.

Example 4.4: Illustration of "Branch on Carry Set"

Address	Data	Comments
0100	LDA	Load First Value
0101	ADL1	First Number, Low Byte
0102	ADH1	First Number, High Byte
0103	ADC	Add Second Value
0104	ADL2	Second Number, Low Byte
0105	ADH2	Second Number, High Byte
0106	BCS	Test for Carry Set. If Yes, Branch to 0158
0107	+50	
0108	STA	If Not, Store Results of Add
0109	ADL3	Result, Low Byte
010A	ADH3	Result, High Byte
0158	OP CODE	New Instruction

The 0108 represents the value of the program counter after reading the offset value. The program counter automatically increments so it can reference the next memory location on the next cycle. The add of the offset is a signed binary add as discussed in the arithmetic section. A positive branch is indicated by a 0 in bit 7 of the relative value, and a minus branch is in two's complement form and is indicated by a 1 in bit 7. The inherent capabilities of this type of notation system allow branch conditionally forward 127 bytes from the next instruction and back 128 bytes from that instruction. All branches in the R6500 series are conditional relative branches and all have the form shown above. The advantage of relative addressing is best shown in the following example:

Example 4.5: Sequencing Two Branch Instructions

Address	Data	Comments
0100	LDA	Load First Value
0101	ADL1	
0102	ADH1	
0103	ADC	Add Second Value
0104	ADL2	
0105	ADH2	
0106	BCS	Test for Carry Set. If Yes, Branch to 0158
0107	+50	
0108	BMI	Test for Minus Number. If Yes, Branch to 0095
0109	-75	
010A	STA	If Not, Store
010B	ADL3	
010C	ADH3	

In this example, the previous single-branch example was modified to also test the resulting number to see if it is negative. In sequencing two-branch instructions, this loop is two bytes shorter by use of relative branches rather than three byte branches.

4.1.2 Branch Instructions

4.1.2.1 BMI -- Branch on Result Minus

This instruction takes the conditional branch if the N bit is set (1). Branch on result minus is used to determine if the previous result was minus or bit 7 was on (1).

BMI does not affect any of the flags or any other part of the machine except the program counter and then only if the N bit is set.

The mode of addressing for BMI is Relative.

4.1.2.2 BPL -- Branch on Result Plus

This instruction is the complementary branch to branch on result minus. It is a conditional branch which takes the branch when the N bit is reset (0). BPL is used to test if the previous result bit 7 was off (0).

The instruction affects no flags or registers other than the P counter and only affects the P counter when the N bit is reset.

The addressing mode is Relative.

4.1.2.3 BCC -- Branch on Carry Clear

This instruction tests the state of the carry bit and takes a conditional branch if the carry bit is reset (0).

It affects no flags or registers other than the program counter and then only if the C flag is reset.

The addressing mode is Relative

4.1.2.4 BCS -- Branch on Carry Set

This instruction takes the conditional branch if the carry flag is set (1).

BCS does not affect any of the flags or registers except for the program counter and only then if the carry flag is set.

The addressing mode is Relative.

4.1.2.5 BEQ -- Branch on Result Zero

This instruction could also be called "Branch on Equal." It takes a conditional branch whenever the Z flag is set (1), indicating that the previous result was equal to 0.

BEQ does not affect any of the flags or registers other than the program counter and only then when the Z flag is set.

The addressing mode is Relative.

4.1.2.6 BNE -- Branch on Result Not Zero

This instruction could also be called "Branch on Not Equal." It tests the Z flag and takes the conditional branch if the Z flag is reset (0), indicating that the previous result was not zero.

BNE does not affect any of the flags or registers other than the program counter and only then if the Z flag is reset.

The addressing mode is Relative.

4.1.2.7 BVS -- Branch on Overflow Set

This instruction tests the V flag and takes the conditional branch if V is set (1)

BVS does not affect any flags or registers other than the program counter and only when the overflow flag is set.

The addressing mode is Relative.

4.1.2.8 BVC -- Branch on Overflow Clear

This instruction tests the status of the V flag and takes the conditional branch if the flag is reset (0).

BVC does not affect any of the flags or registers other than the program counter and only when the overflow flag is reset.

The addressing mode is Relative.

4.1.3 Branch Summary

To summarize, the R6500 branches have two characteristics; each of them tests the state of a flag and then either accesses the next instruction in program sequence if the flag is not in the tested state or adds the offset value to the PC value at the OP CODE of the next instruction (PC + 1) to allow the program to change operations. This gives the programmer the full ability to make decisions. By writing a sequence of branch instructions, any combination of conditions of the microprocessor may be determined and new action taken as a result of the tests.

There are four branch conditions in the R6500 Series microprocessors. These are branch on carry flag, branch on overflow flag, branch on N flag, and branch on zero flag. Each of the branches has a branch on flag set (1) or branch on flag clear (0).

4.1.4 Solution to Branch Out of Range

The branch relative instruction is unlike the jump instruction which can reach anywhere in memory, since branch relative is limited to +129 or -126 from the current program counter location. Although for many loops and many tests this is sufficient range, longer programs will occasionally find it necessary to conditionally branch to a location that is significantly further away than the branch command will directly reach. This is one of the uses of complementary branches. If a program should find it necessary to branch to a location which was significantly further away than 129, the following solution would facilitate the branch:

Example 4.6: Use of JMP to Branch Out of Range

	Address	Data	Comments
	100	LDA	Load First Value
	101	ADL1	
	102	ADH1	
	103	ADC	Add Second Value
	104	ADL2	
	105	ADH2	
	106	BCC	Branch, If No Carry, Ahead 3 (to Point 2)
	107	+3	
	108	JMP	If Carry set, Jump to Location Specified by ADH4, ADL4
	109	ADL4	
	10A	ADH4	
Point 2	10B	BMI	Check for Minus
	10C	Offset	
	10D	STA	
	10E	ADL3	If not Minus, Store Result
	10F	ADH3	

In this example, carry set is being checked. In order to accomplish this when the branch command would have to reach outside of the 128 range, the use of a complementary branch is required. Instead of doing the "branch on carry set" to the location, the "branch on carry clear" is utilized (a complementary instruction) which branches past the jump. If the complementary branch is not taken, the jump is the "branch on carry set" function.

This technique of branching past a jump with the complementary branch is a universal solution to the branch out of range problem.

Another solution is to find a like branch to the same location that is within range and, although this involves two branches to transfer control, it does save memory locations.

By use of the relative branch fewer bytes of code are required than if a conditional jump had been used. However, in large programs, the branch out of range occurs more frequently. If the user can determine that a branch will be out of range by inspection, he should apply the jump solution at the time he is writing the code. Otherwise, the

assemblers will indicate an out-of-range branch which will require recoding to use the jump solution.

4.2 TEST INSTRUCTIONS

Although most of the normal operations of the microprocessor involve setting of flags, there are specific instructions which are designed only to set flags for testing with the branch instruction.

4.2.1 CMP - Compare Memory and Accumulator

This instruction subtracts the contents of memory from the contents of the accumulator.

Its symbolic notation is A - M.

The use of the CMP affects the following flags: Z flag is set on an equal comparison, reset otherwise; the N flag is set or reset by the result bit 7, the carry flag is set when the value in memory is less than or equal to the accumulator, reset when it is greater than the accumulator. The accumulator is not affected.

It is a "Group One" instruction and therefore has as its addressing modes: Immediate; Zero Page; Zero Page, X; Absolute; Absolute, X; Absolute, Y; (Indirect, X); (Indirect), Y.

The purpose of the compare instruction is to allow the user to compare a value in memory to the accumulator without changing the value of the accumulator. An example of where this becomes extremely important is when one is receiving command instructions from an external device. In this case, an input byte may have several values. Each value can cause the program to perform a different operation. The only rapid way to determine the value of the input data is to compare the memory with a series of constants. It is fairly simple to perform "compare to constant" operations. By use of the immediate addressing mode which will be developed later, the following example compares an input to three values and branches to different locations for each:

Example 4.7: Using the CMP instruction

```
Data          Comments

LDA           Load Value
ADL           Address Low
ADH           Address High
CMP           Compare COUNT 1 to Accumulator
COUNT 1
BEQ           If Equal, Take the Branch of OFFSET 1
OFFSET 1
CMP           Compare COUNT 2 to Accumulator
COUNT 2
BEQ           If Equal, Take the Branch of OFFSET 2
OFFSET 2
CMP           Compare COUNT 3 to Accumulator
COUNT 3
BEQ           If Equal, Take the Branch of OFFSET 3
OFFSET 3
Next Inst.    Otherwise, Go to Next Instruction
              Based on Default Value (COUNT 4).
```

This example shows how to use the default option. A value was compared against three values and, if none were equal, a fourth or default value is assumed. This is a useful technique for code minimization.

The compare instruction is designed to allow a signed comparison between two values, assuming one makes appropriate use of the Z and N and C flags. In order to give maximum flexibility to the instruction, the instruction performs an effective subtract between the value in memory and the value in the accumulator. The reason it is an effective subtract is that subtraction permits the user to compare equal or less with one instruction.

The results of a compare are:

	N	C	Z	V
Accumulator < Memory	Either	Reset	Reset	Unchanged
Accumulator = Memory	Reset	Set	Set	Unchanged
Accumulator > Memory	Either	Set	Reset	Unchanged

So, to check if the accumulator is less than memory, the compare is followed by a BCC; to check if equal to it is followed by a BEQ; and to check if greater it is followed by a BEQ followed by a BCS. Greater than or equal to is checked by BCS.

4.2.2 Bit Testing

The comparison instruction is designed for cases when byte or multiple bytes of values are being compared; however, in the analysis of logic functions, it is very often necessary to determine the condition of an individual bit. One of the ways to accomplish this is with the use of the AND instruction as previously discussed. In other words, the user can load a value into the accumulator and AND it with a field that contains a one bit only in the corresponding bit poston to the bit under test. By using a Branch on Zero Flag after the AND, the status of the bit in memory is testable by this technique. However, the use of this technique involves destroying the accumulator value with the AND instruction. Therefore, searching a table looking for a single bit in a given position would necessitate the reloading of the test value (mask) after each AND instruction. In order to allow memory sampling without disturbing the accumulator, the BIT instruction is used.

4.2.2.1 BIT -- Test Bits in Memory with Accumulator

This instruction performs an AND between a memory location and the accumulator but does not store the result of the AND into the accumulator.

The symbolic notation is M $\wedge$ A.

The bit instruction affects the N flag with N being set to the value of bit 7 of the memory being tested, the V flag with V being set equal to bit 6 of the memory being tested and Z being set by the result of the AND operation between the accumulator and the memory if the result is Zero, Z is reset otherwise. It does not affect the accumulator.

The addressing modes are Zero Page and Absolute.

The BIT instruction, like the compare test, permits the examination of an individual bit without disturbing the value in the accumulator and is illustrated by the following example:

Example 4.8: Sample Program Using the BIT Test

```
Data        Comments
LDA         Load MASK into Accumulator
MASK
BIT         Test First Memory Value for Mask Bit
ADL1
ADH1
BNE         Branch if Set
+50
BIT         Test Second Memory Value for Mask Bit
ADL2
ADH2
BNE         Branch if Set
-75
etc.
```

The value "MASK" loaded into the accumulator in this example is actually a descriptive title since this byte is 8 bits only one of which is a 1. Using this byte in the AND operation inherent in the BIT test will effectively mask out all bits in the memory location under test except that bit position corresponding to the 1 residing in the accumulator. In Example 4.8, the MASK byte is AND'ed to the data found in location ADH1, ADL1 and if the bit under test is a 1, the branch will be taken; if not a 1, the second memory location will be tested with the same mask, etc.

In addition to the nondestructive feature of the bit which allows us to isolate an individual bit by use of the branch equal or branch not equal test, two modifications to the PDP-11 version of that instruction have been made in the R6500 series microprocessors. These modifications are made to permit a test of bit 7 and bit 6 of the field examined with the BIT test. This feature is particularly useful in serving polled interrupts, and especially in dealing with the R6520 Peripheral Interface Device. This device has an interrupt sense bit in bit 6 and bit 7 of the status words. It is a standard of the M6800 bus that whenever possible, bit 7 reflects the interrupt status of an I/O device. This means that under normal circumstances, an analysis of the N flag after a load or BIT instruction should indicate the status of the bit 7 on the I/O device being sampled. To facilitate this test using

the BIT instruction, bit 7 from the memory being tested is set into the N flag irrespective of the value in the accumulator. This is different from the BIT instruction in the M6800 which requires that bit 7 also be set in the accumulator to set N. The advantage to the R6520 user is that if he decides to test bit 7 in the memory, it is done directly by sampling the N bit with a Bit followed by branch minus or branch plus instruction. This means that with the R6520, I/O sampling can be accomplished at any time during the operation of instructions irrespective of the value preloaded in the accumulator.

Another feature of the BIT test is the setting of bit 6 into the V flag. As indicated previously, the V flag is normally reserved for overflow into the sign position during an add and subtract instruction. In other words, the V flag is not disturbed by normal instructions. When the BIT instruction is used, it is assumed that the user is trying to examine the memory that he is testing with the BIT instruction. In order to receive maximum value from a BIT instruction, bit 6 from the memory being tested is set into the V flag. In the case of a normal memory operation, this just means that the user should organize his memory such that both of his flags to be tested are in either bit 6 or bit 7, in which case an appropriate mask does not have to be loaded into the accumulator prior to implementing the BIT instruction. In the case of the R6520, the BIT instruction can be used for sampling interrupts, irrespective of the mask. This allows the programmer to totally interrogate both bit 6 and bit 7 of the R6520 without disturbing the accumulator. In the case of the concurrent interrupts, i.e., bit 6 and bit 7 both on, the fact that the V flag is automatically set by the BIT instruction allows the user to postpone testing for the "6th bit on" until after he has totally handled the interrupt "for bit 7 on" unless he performs an arithmetic operation subsequent to the BIT operation.

CHAPTER 5

NON-INDEXING ADDRESSING TECHNIQUES

5.0 ADDRESSING TECHNIQUES

The addressing modes of the R6500 Microcomputer System's microprocessor (CPU) family can be grouped into two major categories: indexed and non-indexed addressing. This section deals with the non-indexed mode of addressing. Before detailing the various modes available to the user, several concepts will be reviewed. The first of these is the concept of memory field. address bus and data bus. Then a brief introduction to two non-indexed addressing modes and timing will be made with the intent of preparing the reader for a discussion of program sequence and the internal activity of the microprocessor during execution of an instruction. This will be followed by a review of how one treats memory and the assorted allocation of memory space to the elements of RAM, ROM and I/O.

Subsequent to reading this section the user should have an understanding of the following fundamentals:

- Memory Field
- Address Bus
- Data Bus
- Cycle Timing
- Program Sequence
- Pipelining

With these tools in hand, the reader will be better prepared to readily comprehend the detailed definitions of the non-indexed addressing modes.

As discussed in Section 1.1, the R6500 System's microprocessor family is organized around a 16-bit address function. All locations are accessed by a 16-bit word, even though in the case of the R6503 thru 6507 and 6513 thru 6515 only 11 or 12 bits are actually utilized.

Sixteen bits of address allow access to 65,536 memory locations, each of which, in the R6500 series, consists of 8 bits of data. Figure 5.1 displays the total memory field and incorporates the concept of address bus and data bus. The memory address can be regarded as 256 pages (each page defined by the high-order byte) of 256 memory locations (bytes) per page. It will be seen in the detailed discussion of addressing that the lowest-order page, page zero, has special significance in the minimization of program code and execution time.

Much of the uniqueness of the R6500 product family has to do with how the 16-bit address is created. The simplest way to create a 16-bit address is for the programmer to indicate to the microprocessor the 16 bits necessary to access a particular operand on which the microprocessor is expected to operate. An instruction consists of 1, 2, or 3 bytes. It always takes 1 byte to specify the operation which is to be performed (OP CODE). This OP CODE is then followed by 0, 1, or 2 bytes of address depending on the specific operation involved. In the case of the simple instructions such as transfer accumulator to X, operations are performed internally and, therefore, no additional bytes are necessary. This instruction mode is known as "Implied" in the sense that the instruction contains both the OP CODE and the source and destination for the operation. This is the simplest form of addressing and applies to only a limited number of the instructions available in the R6500 family. Another form of addressing, absolute addressing, is the case when the programmer specifies directly to the microprocessor the address he wants the microprocessor to use in fetching the memory value on which the operation will occur. This form is illustrated by the example below.

Example 5.1: Using Absolute Addressing

Clock Cycle	Address Bus	Data Bus
1	0100	LDA, Absolute
2	0101	ADL
3	0102	ADH
4	ADH, ADL	Data

In this example, memory location 0100 contains the OP CODE "LDA Absolute." The next location, 0101, contains ADL which will be defined as the

"low-order byte of the address," hence address low (ADL). Location 0102 contains ADH -- the "high-order byte of the address," hence address high (ADH). At the next clock cycle, the 16 bits composed of ADH and ADL are put on the address bus with the location defined by ADH, ADL containing the data to be loaded into the accumulator. The effective address of the data is best described in Figure 5.1, where the 16-bit address (AB00 through AB15) is composed of ADH and ADL.

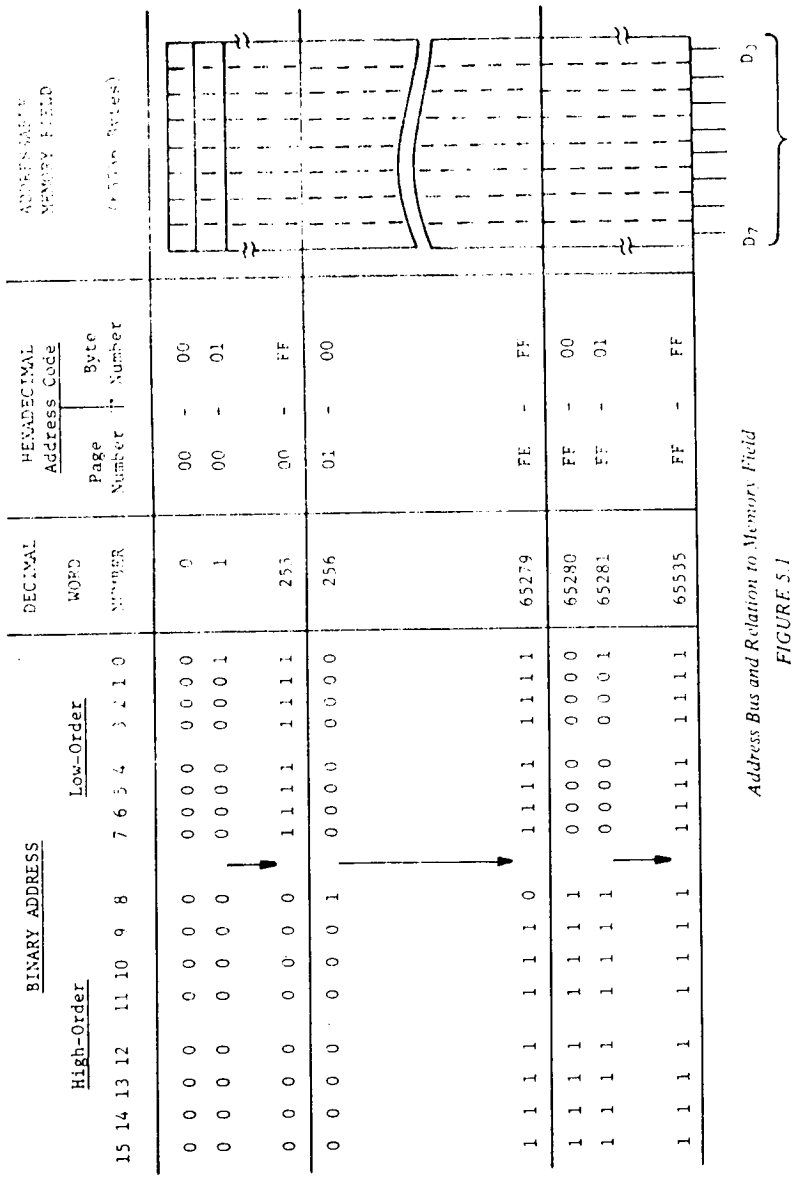
This is the normal form for an absolute memory address. The first byte of the instruction which is picked up by the program counter is the operation code. This is interpreted by the microprocessor as "Load A - Absolute." At the same time that this Load A is being interpreted by the microprocessor, the microprocessor accesses the next memory location by putting the program counter content, which was incremented as the OP CODE was fetched, on the address bus.

5.1 CONCEPTS OF PIPELINING AND PROGRAM SEQUENCE

The overlap of fetching the next memory location while interpreting the current data from memory minimizes the operation time of a normal 2- or 3-byte instruction and is referred to as "pipelining." It is this feature that allows a 2-byte instruction to take only 2 clock times and a 3-byte instruction to be interpreted in 3 clock cycles.

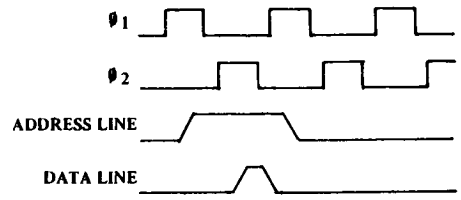
In the R6500 series microprocessors, a clock cycle is defined as one complete operation of each of the two phase clocks. Figure 5.2 is a sketch of the address and data bus timing as it relates to the system clocks.

The major point to be noted is that every clock cycle in the microprocessor is a memory cycle in which memory is either read or written. Simultaneously with the read or write of memory, an internal operation of the microprocessor is also occurring.



Address Bus and Relation to Memory Field

FIGURE 5.1



Example of Timing -- A R6500 Microcomputer System Microprocessor

FIGURE 5.2

The following example will let us analyze this effect:

Example 5.2: Demonstration of "Pipelining" Effect

Clock Cycles	External Operation	Address	Data	Internal Operation
1	Fetch OP CODE	100	ADC	Increment P-counter to 101
2	Fetch first-address half from memory	101	ADL	Increment P-counter to 102, Interpret ADC instruction
3	Fetch second address half from memory	102	ADH	Increment P-counter to 103; Hold ADL
4	Fetch operand from memory	ADH, ADL	Data	Load Data
5	Fetch next OP CODE from memory	103	STA	Increment P-counter to 104, Perform ADC operation: A + M + C
6	Fetch address from memory	104	ADL	Increment P-counter to 105, Result of Add → accumulator, Interpret STA Instruction

The above example shows the operation of an ADC, add with carry instruction, using absolute addressing. In the first cycle, the OP CODE is fetched from memory addressed by the P-counter. To implement the

look-ahead or pipeline in cycle two, the fetch of ADL address low is done simultaneously with the interpretation of the ADC absolute instruction. By the end of cycle 2, the microprocessor knows that it should access the next memory location for the address high as a result of interpretation of the absolute addressing mode.

The address low (ADL) is stored in the ALU while the address high (AD is being fetched in cycle 3.

On the fourth cycle, no internal operation is necessary while the microprocessor is putting the calculated value onto the address bus. However, during this cycle, the operand is loaded into the microprocessor.

The four cycles have all been involved with memory access for the ADC absolute instruction. The first to fetch the instruction, the second to fetch the address low, the third to fetch the address high, and the fourth to use the calculated address to fetch the operand. Because that complete the memory operations for this instruction, during the fifth cycle the microprocessor starts to fetch the next instruction from memory while it is completing the add operation from the first instruction. During the sixth cycle, the microprocessor is interpreting the new instruction fetch during cycle 5 while transferring the result of the add operation to the accumulator. This means that even though it really takes six cycles for the microprocessor to do the ADC instruction, the programmer only need concern himself with the first four cycles, as the next two are overlapped as shown.

All instructions take at least two cycles; one to fetch the OP CODE and one to interpret the OP CODE and, with few exceptions, the number of cycles that an instruction takes is equal to the number of times that memory must be addressed.

The details of how each addressing mode is overlapped are described in the individual sections, and for specific details of each cycle in various operations the user is referred to the Hardware Manual, Appendix A.

5.2 MEMORY UTILIZATION

As indicated, the 16-bit address allows the user to access greater than 65,000 separate locations. Most of the locations which will be accessed in the course of a control problem will be in program or P-counter referenced locations. A typical program will probably range from 1000 to 8000 bytes and will normally be implemented in fixed ROM or non-volatile alterable ROM.

A second type of memory will be the read-write memory in which the user keeps data such as working values, input and output data. Depending on the type of problem being addressed, this RAM usually ranges from 32 bytes to 8000 bytes, although most applications will be under 2000 bytes of RAM.

It would seem there is significant address space not used in most applications. To get the maximum benefit of the addressing space, two concepts are implemented in the R6500 series. These are the use of data addressing as I/O control, and distributed address connections for minimum control lines. The latter concept utilizes the address bus, which is basic to and therefore pervasive in any microcomputer system, as a controlling network whenever possible. An example of this is the use of the address bus in selecting devices to interface with the microprocessor.

5.2.1 I/O Control

The advantages of accessing I/O as memory are 1) the use of distributed address space allows for simple I/O control lines and 2) all of the power of the instructions is applied to I/O operations. This has the advantage of minimizing I/O hardware and allows the programmer to be innovative in the application of I/O devices in solving his problem.

All R6500 product family I/O devices contain 8-bit registers which are addressed by the microprocessor as though they were a memory byte. In the simplest case, the 8-bit register being read contains a 1's and 0's pattern which corresponds to the TTL voltage level applied to eight input pins to the I/O device.

If the register were a flip-flop register driving eight output pins with TTL levels, the storing of eight bits of data with a STA instruction into that I/O register would, in effect, be programming the flip-flop to a specific desired state. Thus, one can use the instructions with the I/O just as any other memory location.

5.2.2 Memory Allocation

Figure 5.1 displays the relationship between memory, address bus and data bus while referencing the address values in hexadecimal notation. The previous section has dealt with utilization of memory address space for not only ROM and RAM but for I/O as well. At this time, the concept of allocation of the memory field of Figure 5.1 to the elements of ROM, RAM and I/O will be considered. The allocation below satisfies most applications requirements and represents only a suggested allocation for minimization of programming code and speed.

Hexadecimal Address	Suggested Allocation of Memory
0000 - 3FFF	RAM
4000 - 7FFF	I/O
8000 - FFFF	ROM

It should be noted that the three memory block address definitions which, while not mandatory or required for proper system operation, do represent a logical assignment of space. The choice for this particular allocation will be presented in Section 9.12. In the meantime, the reader should retain the concept of the various memory blocks allocated to RAM, I/O and ROM as they are useful in the following discussion.

5.3 IMPLIED ADDRESSING

Instructions which use implied addressing are single-byte instructions. The byte contains the OP CODE which stipulates an operation internal to the microprocessor. Instructions utilizing this type of addressing include operations which clear and set bits in the P (Processor Status) register, incrementing and decrementing internal registers and transferring

contents of one internal register to another internal register. Operations of this form take two clock cycles to execute. The first cycle is the OP CODE fetch and, during this fetch, the program counter increments.

In the second cycle, the incremented P-counter is now the address of the next byte of the instruction. However, since the OP CODE totally defines the operation, the second memory fetch is worthless and any P-counter increment in the second cycle is suppressed. During the second cycle, the OP CODE is decoded with recognition of its single-byte operation.

In the third cycle, the microprocessor repeats the same address to fetch the next OP CODE. This is the second time the memory address is fetched; once as the second byte of the first instruction and second, as the correct OP CODE address for the next instruction.

A symbolic representation of a 2-cycle instruction is given below. "PC" means "Program Counter."

Example 5.3: Illustration of Implied Addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 1	New OP CODE	Ignore New OP CODE; Decode Old OP CODE
3	PC + 1	PC + 2	New OP CODE	Fetch New OP CODE; Execute Old OP CODE

Instructions which use implied addressing and require only 2 cycles include CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

Instructions utilizing implied addressing and which require more than 2 cycles are stack operations which include BRK, PHA, PHP, PLA, PLP, RTI, and RTS.

5.4 IMMEDIATE ADDRESSING

Instructions which use immediate addressing are 2-byte instructions. The first byte contains the OP CODE specifying the operation and address mode. The second byte contains a constant value defined by the programmer. It is often necessary to compare, load and/or test against certain known values. Rather than requiring the user to define and load constants into some auxiliary RAM, the microprocessor allows the user to specify constant values by the immediate addressing mode.

Example 5.4: Illustration of Immediate Addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 2	Data	Fetch Data, Decode OP CODE
3	PC + 2	PC + 3	New OP CODE	Fetch New OP CODE, Execute Old OP CODE

Immediate addressing is the simplest form of constant manipulation available to the programmer. It requires a minimum execution time in the sense that one cycle is used in loading the OP CODE and as this CODE is being interpreted, the constant is being fetched.

Instructions utilizing immediate addressing are ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, and SBC.

5.5 ABSOLUTE ADDRESSING

Instructions which use absolute addressing are 3-byte instructions. The first byte contains the OP CODE for specifying the operation and address mode. The second byte contains the low-order byte of the effective address (that address which contains the data), while the third byte contains the high-order byte of the effective address. Thus, the programmer specifies the full 16-bit address and, since any memory location can be specified, this is considered the most normal mode for addressing. Other modes may be considered special subsets of this 16-bit addressing mode.

Example 5.5: Illustration of Absolute Addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 2	ADL	Fetch ADL, Decode OP CODE
3	PC + 2	PC + 3	ADH	Fetch ADH, Hold ADL
4	ADH, ADL	PC + 3	Data	Fetch Data
5	PC + 3	PC + 4	New OP CODE	Fetch New OP CODE, Execute Old OP CODE

The basic operation of the microprocessor in an Absolute address mode is to read the OP CODE in the first cycle while finishing the previous operation. In the second cycle, the microprocessor automatically reads the first byte after the OP CODE (in this case the address low) while interpreting the operation code. At the end of this cycle, the microprocessor knows that it needs a second byte for program sequence; therefore, one more byte will be accessed using the program counter while temporarily storing the address low. This occurs during the third cycle. In the fourth cycle, the operation is one of taking the address low and address high that were read during cycles 2 and 3 to address the operand. For example, in load A, the effective address is used to fetch from memory the data which is going to be loaded in the accumulator. In the case of storing, data is transferred from the accumulator to the addressed memory.

As was illustrated in the review of pipelining, depending on the instruction, it is possible for the microprocessor to start the next instruction fetch cycle after the effective address operation and independently of how many more internal cycles it may take to complete the OP CODE. The only exception to this is the case of "Jump Absolute" in which the address low and address high that are fetched in cycle 2 and cycle 3 are used as the 16-bit address for the next OP CODE. The jump absolute therefore only requires three cycles. In all other cases, absolute addressing takes four cycles, three to fetch the full instruction including the effective address, the fourth to perform the memory transfer called for in the instruction.

Absolute Addressing always takes three bytes of program memory; one for the OP CODE, one for the address low, one for the address high.

Instructions which have absolute addressing capability include ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

5.6 ZERO PAGE ADDRESSING

Instructions which use zero page addressing are 2-byte instructions. The first byte contains the OP CODE, while the second byte contains the effective address in page zero of memory.

As seen in absolute addressing, the ability to address anywhere in the 65K memory space costs three bytes of program space, plus a minimum of four cycles to perform address operations. In order to allow the user to shorten both memory space and execution time, particularly when dealing with working registers and intermediate values, the R6500 System's microprocessor family has a special addressing mode that automatically assumes the effective address high (ADH) to be that of the lowest page of memory. In order to understand the page concept one should think of each of the various memory addresses as comprising a consecutive block of 256 locations which have an independent high-order address associated with that block. Each block is called a page. Other than for zero page and for calculating indexed addresses which will be covered in the following sections, the microprocessor pays little attention to the page concept.

The microprocessor assumes that the high-order byte of the effective address, for instructions which indicate the zero page addressing option, is all 0's (ADH = 00, hexadecimal). This allows the following sequence to occur:

Example 5.6: Illustration of Zero Page Addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 2	ADL	Fetch ADL, Decode OP CODE
3	00, ADL	PC + 2	Data	Fetch Data
4	PC + 2	PC + 3	New OP CODE	Fetch New OP CODE, Execute Old OP CODE

On the first cycle, the microprocessor puts out the program counter, reads the OP CODE and increments the program counter. On the second cycle, the microprocessor puts out the program counter, reads the effective address low, interprets the OP CODE and increments the program counter. So far, the operations are identical to those described in the absolute addressing mode. However, by the end of the second cycle, the microprocessor has decoded the fact that this is a zero page operation and on the next cycle, it outputs address 00, as the effective address high, along with the address low that it just fetched, and then either reads or writes memory at that location, depending on the OP CODE.

The advantage of zero page addressing is that it takes only two bytes, one for the OP CODE and one for the effective address low; and only three cycles, one to fetch the OP CODE, one to fetch the address low, and one to fetch the data, as opposed to absolute addressing which takes three bytes and four cycles.

In order to make most effective utilization of this concept, the user should organize his memory so that he is keeping his most frequently accessed RAM values in the memory locations between 0 and 255. If one organizes the zero page of memory properly, including moving data into these locations for longer loops, significant shortening of program code and execution time can be obtained.

The concept of zero page is so important that the R6500 assemblers have error notations which indicate when improper use of this space is made. If one's memory is organized according to the guidelines shown in Section 5.2.2, one normally will find working storage located in values from 0 to 255. This is an important aspect of the discipline known as "memory management."

Once the pattern of coding for the R650X or R651X, which considers working storage or registers in the zero page, becomes a habit, one finds that in most control applications, all of the working registers will take advantage of this programming and the associated time reduction without any special effort on the user's part.

Instructions which allow zero page addressing include ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

5.7 RELATIVE ADDRESSING

As discussed in Section 4.1, all of the branch operations in the microprocessor use the concept of relative addressing. In example 5.7, it is seen that for the case of the straightforward branch in which the branch is not taken, on the first program count cycle, the microprocessor puts out the program counter as an address, fetches the OP CODE and finishes the previous operation. During the second cycle, the program counter is put on the address bus, picking up the relative offset. Internally, the microprocessor is decoding the OP CODE to determine that it is a branch instruction.

Example 5.7: Illustration of Relative Addressing: Branch Not Taken

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Finish Previous Operation, Increment Program Counter to 101
2	0101	Offset	Fetch Offset	Interpret Instruction, Increment Program Counter to 102
3	0102	Next OP CODE	Fetch Next OP CODE	Check Flags, Increment Program Counter to 0103

This is only the second cycle of an internal operation; therefore, the microprocessor may be storing a computed value from the previous instruction at the same time it is finishing interpreting the present instruction. It is while doing the store operation that the flags in the machine get physically set; therefore, the microprocessor allows the program counter

to go one more cycle to allow itself time to determine the value of the flags. For example, if the previous instruction is ADC, the flags will not get set until the cycle in which the offset value is fetched.

During the third cycle, the microprocessor puts the incremented PC onto the address bus, fetches the next OP CODE and checks the flag in order to decide whether or not the program counter value that is going out is correct and that the branch is not going to be taken. Therefore, an additional type of pipeline, in this case fetching the next OP CODE in a branch sequence, accomplishes the implementation of a branch relative with no branch being taken. This requires two cycles. One cycle fetches the branch OP CODE and one cycle fetches the next operation, the relative offset. The second fetch is effectively ignored by virtue of the fact that the branch is not taken, so the program counter location has already been incremented and the next OP CODE has already been fetched by the microprocessor.

If in the above example it is assumed that the flag is set such that the branch is taken and the relative offset is +50, the microprocessor takes a third cycle to perform the branch operation.

Example 5.8: Illustration of Relative Addressing; Branch Positive Taken, No Crossing of Page Boundaries

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Finish Previous Operation, Increment Program Counter to 101
2	0101	+50	Fetch Offset	Interpret Instruction, Increment Program Counter to 102
3	0102	Next OP CODE	Fetch Next OP CODE	Check Flags, Add Relative to PCL, Increment Program Counter to 103
4	0152	Next OP CODE	Fetch Next OP CODE	Transfer Results to PCL, Increment Program Counter to 153

In Example 5.8, on the first cycle, a branch OP CODE is fetched while the previous operation is finished. On the second cycle, the offset is fetched while the branch instruction is being interpreted. On the third cycle, the microprocessor uses the adder to add the program count low to

the offset and also checks the flags. Because the program count for the next OP CODE in program sequence is already in the program counter and is being incremented, the microprocessor can allow the incrementation process to continue. If the value for the next instruction is indicated because the flag is not set; then the microprocessor loads the next OP CODE and the add of the program counter low to the offset value, is ignored as it was in the previous example.

If during the third cycle the flag is found to be the correct value for a branch, the OP CODE that has been fetched during this cycle is ignored. The microprocessor then updates the program counter with the results from the add operation, puts that value out on the address bus which fetches a new OP CODE.

This gives the effect of a 3-cycle branch. Thus it can be seen that in a case where the branch is not taken, the microprocessor has an effective 2-cycle branch, i.e., two memory references. In the case when the branch is taken, the branch takes three cycles as long as the relative value does not force an update to the program counter high. In other words, three cycles are required if the page boundary is not crossed (recall the discussion of the "page" concept in Section 5.0). If in the above example the branch was back from address 0102 fifty locations, as opposed to +50 locations, the following result would occur:

Example 5.9: Illustration of Relative Addressing: Branch Negative Taken, Crossing of Page Boundary

Cycle	Address Bus	Data Bus	External Operations	Internal Operations
1	0100	OP CODE	Fetch OP CODE	Finish Previous Instruction
2	0101	-50	Fetch Offset	Interpret Instruction
3	0102	Next OP CODE	Fetch Next OP CODE	Check Flags Add Relative to PCL
4	01B2	Discarded Data	Fetch Discarded Data	Store Adder in PCL and Subtract 1 from PCH
5	00B2	Next OP CODE	Fetch Next OP CODE	Put Out New PCH and Increment PC to 00B3

In this example, the adder is used to perform the arithmetic operation, and the adder can do only the eight bits of addition at a time. The minus branch crosses back over the page boundary, therefore an intermediate result is developed of 01B2 which has no intrinsic value because of the borrow which now has to be reflected into the program counter high. Since this example displays both a negative offset and the crossing of a page boundary, additional explanation is in order.

The value to which the offset will be added is 0102 (hexadecimal). The offset itself is -50 (hexadecimal).

Subtract low-order byte:

02_{HEX} = 0000 0010
50_{HEX} = 0101 0000

Take two's complement of 50:

$\overline{50}$ = 1010 1111
Add 1 = 1
-50 = 1011 0000

Add 02	0000	0010
-50	1011	0000
Carry = $\overline{0}$	1011	0010
	B	2

Up to this point, the PCH has not been affected; therefore the value on the address bus is 01B2.

The Carry = 0, indicating a borrow.

Subtract high-order byte:

01_{HEX} = 0000 0001
00_{HEX} = 0000 0000

Take two's complement of 00:

$\overline{00}_{HEX}$ = 1111 1111
Add Carry = 0
-00_{HEX} = 1111 1111

Add 01	0000	0001
-00	1111	1111
Carry = $\overline{1}$	0000	0000
	0	0

The presence of the Carry indicates no borrow, hence a positive result.

At this time, after the arithmetic operation on both bytes of the P.C., the address bus will be: 00B2.

The microprocessor does put out on the address line the intermediate results (01B2), thereby reading a location within the page it was currently working in, the value of which is ignored. It then subtracts 1, or if this was a branch forward to the next page, the microprocessor would add 1 to program counter high in this fourth cycle. In the fifth cycle, the microprocessor will recognize that it has the correct new program counter high and program counter low and is able to start a new instruction operation, thereby giving an effective length to the branch operation when a page crossing is encountered of four cycles.

We can see that it is possible to control the execution time of a branch. This is important for counting or estimating execution times of operations. For counting purposes, the following applies:

If a branch is normally not taken, assume two cycles for the branch.

If the branch is normally taken but it is not across the page boundary, assume three cycles for the branch.

If the branch is over a page boundary, then assume four cycles for the branch.

In loops which are repeated many times, one can assume some type of statistical factor between 3 and 2, or 4 and 2, depending on the probability of taking the branch versus not taking it.

It should be re-emphasized that other than for timing purposes, page boundary crossings can be ignored by the programmer.

To summarize, the relative addressing always takes two bytes, one for the OP CODE and one for the offset.

The execution time is as follows:

Branch with Not Taking the Branch	--	2 cycles
Branch When the Branch Is Taken But No Page Crossing	--	3 cycles
Branch When the Branch Is Taken with a Page Crossing	--	4 cycles

Only branch instructions have relative addressing. The branch instructions are: BCC, BEQ, BMI, BNE, BPL, BCS, BVC, BVS. For a more detailed explanation of relative offset calculations the reader is referred to Appendix H.

CHAPTER 6

INDEX REGISTERS AND INDEX ADDRESSING CONCEPTS

6.0 GENERAL CONCEPT OF INDEXING

In previous sections techniques for using the program counter to address memory locations after the operation code to develop the address for a particular operation have been discussed. Other than cases when the programmer directly changes the program memory, it can be considered that the addressing modes discussed up until now are fixed or direct addresses and each has the relative merits discussed under each individual section. However, a more powerful concept of addressing is that of computed addressing. There are basically two types of computed addressing; indexed addressing and indirect addressing.

Indexed addressing uses an address which is computed by means of modifying the address data accessed by the program counter with an internal register called an index register.

Indirect addressing uses a computed and stored address which is accessed by an indirect pointer in the programming sequence.

In the R6500 product family, both of these modes are used and combinations of them are available.

Before undertaking the more difficult concepts of indirect addressing the concept of indexed addressing will be developed.

In order to move five bytes of memory from a starting address of FIELD 1 to another set of addresses, starting with FIELD 2, the following program could be written:

Example 6.1: Moving Five Bytes of Data With Straight Line Code

LABEL	INSTRUCTION	OPERAND	COMMENTS
START	LDA	FIELD 1	Move First Value
	STA	FIELD 2	
	LDA	FIELD 1 + 1	Move Second Value
	STA	FIELD 2 + 1	
	LDA	FIELD 1 + 2	Move Third Value
	STA	FIELD 2 + 2	
	LDA	FIELD 1 + 3	Move Fourth Value
	STA	FIELD 2 + 3	
	LDA	FIELD 1 + 4	Move Fifth Value
	STA	FIELD 2 + 4	

In this example, data are fetched from the first memory location in FIELD 1, as addressed by the next one or two bytes in program memory, stored temporarily in A and then written into the first memory location in FIELD 2, also addressed by the next one or two bytes in program memory. This sequence is repeated, with only the memory addresses changing, until all the data have been transferred. This type of programming is called "straight line" programming becasue each repetitive operation is a separate group of instructions listed in sequence or straight line form in program memory. This is necessary even though the instruction OP CODES are identical for each memory transfer operation because the specific memory addresses are different and require a different code to be written into the program memory for each transfer.

It takes a total of 10 instructions to accomplish the move when it is implemented this way. It should be noted that it is not indicated whether or not FIELD 1 and FIELD 2 are Zero Page addresses or Absolute addresses.

If they were Zero Page addresses, the total number of bytes consumed in solving the problem would be two bytes for each instruction, thereby requiring 20 bytes of memory; if both FIELD 1 and FIELD 2 were Absolute memory locations, each instruction would take three bytes and this program would require 30 bytes of program storage.

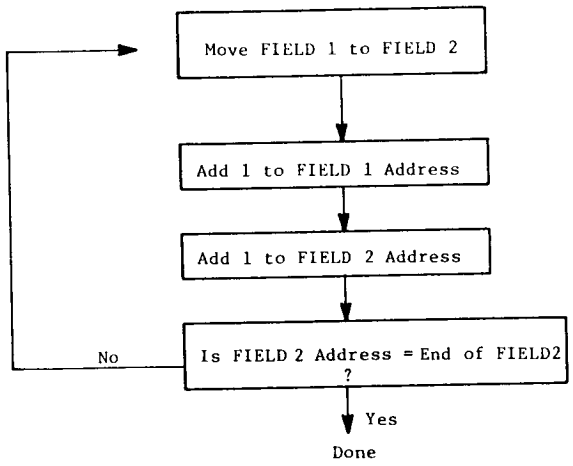
The Zero Page program would execute in three cycles per instruction or a total of 30 cycles, and the Absolute location version would execute in four cycles per instruction or a total of 40 cycles.

A new concept has been introduced in this example, that of symbolic notation rather than actual locations for the instructions.

The form in which this short program is written uses symbolic addressing in which the address of the beginning of the program has the name, "START". Symbolic representations of addresses such as "START" are referred to as labels. The addresses in the two address fields used in this example have also been given names: The first address of the first field is called FIELD 1; and the first address of the second field is called FIELD 2. Each additional address in the fields has been given a number which is referenced to the first number; for example, the address of the third byte in the first field is FIELD 1 + 2. All of these concepts are implemented to simplify the ease of writing a program because the user does not have to worry about the locations of FIELD 1 and FIELD 2 until after analyzing the memory needs of the whole program. Symbolic notation also results in a more readable program.

Translation from symbolic form instructions and addresses into actual numerical OP CODES and addresses is accomplished by a program called a symbolic assembler. Several different versions of symbolic assemblers are available for the R6500 product family. Symbolic notation will be employed throughout the remainder of this text because of its ease of understanding, and because individual byte addresses are unnecessary, although for an explanation of a particular mode the byte representation may be used.

In this example, only direct addresses were used. A program to reduce the number of bytes required to move the five values follows:



Moving Five Bytes of Data with Loop

FIGURE 6.1

Example 6.2 is a program listing that corresponds to the flow chart:

Example 6.2: Moving Five Bytes of Data With Loop

LABEL	INSTRUCTION	OPERAND	COMMENTS
INIT	CLC		
START	LDA	FIELD 1	Move Loop
OTHER	STA	FIELD 2	
	LDA	START + 1	Modify Move Values
	ADC	#1	
	STA	START + 1	
	LDA	OTHER + 1	
	ADC	#1	
	STA	OTHER + 1	
	CMP	#FIELD 2 + 5	Check for End
	BNE	START	

NOTE: For ease of reading, labels have been written in the form "FIELD 1". This is incorrect format for use in the various symbolic assemblers. "FIELD 1" must be written "FIELD1" when coding for assembler formats.

Assuming Zero Page, direct addressing, Example 6.3 is written below with one byte per line just as it would appear in program memory. This will provide a more detailed description of Example 6.2.

Example 6.3: Coded Detail of Moving Fields With Loop

LABEL	CODE NAMES	COMMENTS
INIT	CLC	Clear Carry
START	LDA	(FIELD 1)→A
	FIELD 1	
OTHER	STA	A→(FIELD 2)
	FIELD 2	
	LDA	From Address→A
	START + 1	
	ADC	A + 1→A
	1	
	STA	A→From Address
	START + 1	
	LDA	To Address→A
	OTHER + 1	
	ADC	A + 1→A
	1	
	STA	A→To Address
	OTHER + 1	
	CMP	A - ORIGINAL FIELD 2 + 5
	ORIGINAL FIELD 2 + 5	
	BNE	If not, loop to START
	START	

In this example, the program is modifying the addresses of one load instruction and one store instruction rather than writing 10 instructions to move five bytes of data and 50 instructions to move 25 bytes of data.

The address of the Load A instruction is located in memory at START +1, and the Store instruction at OTHER + 1. In order to perform this operation, the address must be modified once for each move operation until all of the data are moved.

Checking for the end of the moves is accomplished by checking the results of the address modification to determine if the address exceeds the end of the second field. When it does so, the routine is complete.

If 100 values were to be moved this program would remain 20 bytes long, whereas the solution to the first problem would require a program of 200 instructions.

The type of coding used in this example is called a "loop." Although the program loop in this case requires as many bytes as the original program, more values could be moved without increasing the length of the program. The greater the number of repetitive operations that are to be accomplished, the greater the advantage of the loop type program over straight line programming.

Important Note: The execution time required to move the five values is significantly longer using the loop program than the straight line program. In the straight line program, if a Zero Page operation is assumed, the time to perform the total move is 30 cycles. Using the loop program, the execution time to move five values is five times through the entire loop, which takes 25 cycles. Therefore the time to move five values is 125 cycles.

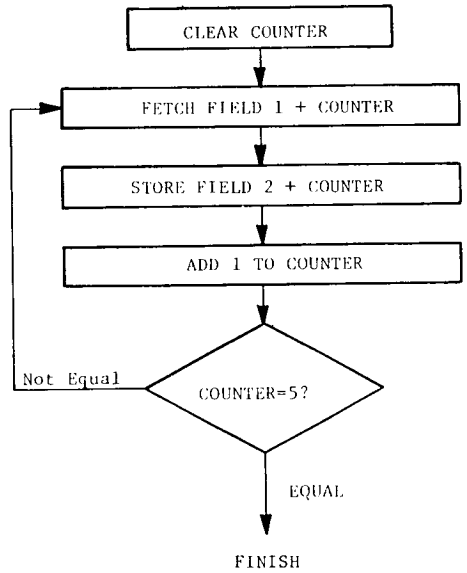
While loops have an advantage in coding space efficiency, all loops cost time. If the programmer has a problem that is extremely time-dependent, taking the loop out and going to straight line programming, even though it is extremely inefficient in terms of its utilization of memory, will often solve the timing problem.

The straight line programming technique becomes very useful in some control applications. However, it is not recommended as a standard technique, but should be used only when there are extreme timing problems. Loops will normally save a significant number of bytes but they will always take more time.

The technique used in the loop program example has two major problems:

1. The necessity to modify program memory. This should be avoided to take advantage of the ability to put programs into read-only memory with the corresponding savings in hardware costs.

2. Although this is the simplest form of computed addressing, fewer program bytes would be necessary with the more sophisticated form of program shown in the following flow chart.



Moving Five Bytes of Data with Counter
FIGURE 6.2

In the R6500 System's microprocessor family, the counter is called an index register. It is an 8-bit register which is loaded from memory and has the ability to have 1 added to it by an increment instruction (INX,INY) and can be compared directly to memory using the compare index instruction (CPX,CPY). Example 6.4 shows the program listing for the flow chart of Figure 6.2.

Example 6.4: Moving Five Bytes of Data With Index Register

BYTES	LABEL	INSTRUCTION	OPERAND	COMMENTS
2		LDX	0	Load Index With Zero
3	LOOP	LDA	FIELD 1,X	
3		STA	FIELD 2,X	
1		INX		Increment Count
2		CPX	5	Compare For End
2		BNE	LOOP	
13 for Absolute				

In this example, index register X is used as an index and as a counter. It is initialized to zero. Data are fetched from memory at the address "FIELD 1 plus the value of register X", and placed in A. The data are then written from A to memory at the address "FIELD 2 plus the value of register X." Register X is incremented by 1 and compared with five in order to determine if all five data values have been transferred. If not the program loops back to LOOP. In this example, "FIELD 1" is called the "Base Address" which is the address to which indexing is referenced.

This only takes 11 or 13 bytes, depending on whether or not the field is in Page Zero or in absolute memory. It still requires 13 or 15 cycles per byte moved, again confirming that loops are excellent for coding space but not for execution time.

It can be seen from the example that there are basically two criteria for an index register: 1) that it be a register which is easily incremented, compared, loaded, and stored; and 2) that in a single instruction one can specify both the Base Address and the index register to be used.

In a R6500-series microprocessor, the indexed instruction is symbolically represented by "OP CODE Address, X." This indicates to the symbolic assembler that an instruction OP CODE should be picked, which should specify either the absolute address modified by the content of index X register or Zero Page address modified by the content of index X register.

In performing these operations, the microprocessor fetches the instruction OP CODE as previously defined, and fetches the address, modifies the address from the memory by adding the index register to it prior to loading or storing the value of memory.

The index register is a counter. As discussed previously, one of the advantages of the flags in the microprocessor is that a value can be modified and its results tested. Assume that the last example is modified so that instead of moving the first value in FIELD 1 to the first value in FIELD 2, the last value in FIELD 1 is moved first to the last value in FIELD 2, then the next to the last value, etc. and finally the first value. With the index register preloaded with five and using a decrement instruction, the contents of the index register would end at zero after the five fields of data were transferred. The zero indicates that the number of times through the loop is correct and the loop exited by use of the zero test. The program listing for this modification is shown in Example 6.5:

Example 6.5: Moving Five Bytes of Data by Decrementing the Index Register

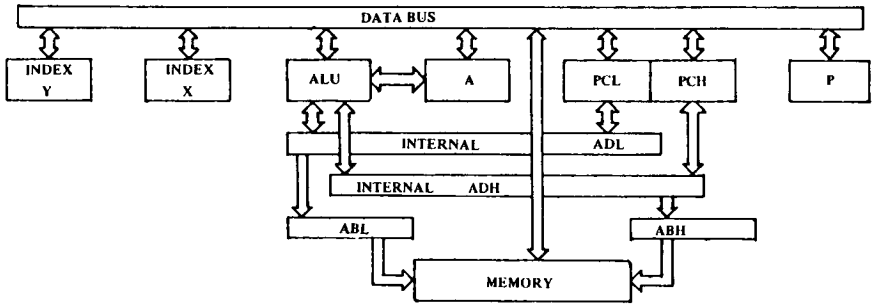
LABEL	INSTRUCTION	OPERAND
	LDX	5
LOOP	LDA	FIELD 1-1,X
	STA	FIELD 2-1,X
	DEX	
	BNE	LOOP

In this example, Index Register X is again used as an Address Counter but it will count backwards. It is initialized to five for this example. Data are fetched from memory at the address "FIELD 1-1 plus the value of Register X" and placed in A. The data are then written from A to memory at the address "FIELD 2-1 plus the value of Register X." Register X is decremented by one. If the decremented value is not zero, as determined by a Branch on Zero instruction, the program loops back to LOOP.

The loop has been decreased to 9 or 11 bytes, and the execution time per byte has been decreased from 15 cycles to 13 cycles per value

which shows the advantage of using the flag setting of the decrement index instruction.

The two index registers, X and Y, can now be added to the system block diagram as in Figure 6.3.



Partial Block Diagram of a R6500 Microcomputer System Microprocessor Including Index Registers

FIGURE 6.3

Each of the index registers is 8 bits long and is loaded and stored from memory, using techniques similar to the accumulator. Because of this ability, the registers can be considered as auxiliary channels to flow data through the microprocessor. However, their primary use is in being added to addresses fetched from memory to form a modified effective address, as described previously. Both index registers have the ability to be compared to memory (CPX,CPY) and to be incremented (INX,INY) and decremented (DEX,DEY).

Because of OP CODE limitations, X and Y have slightly different applications. X is a little more flexible because it has Zero Page operations which Y does not have with exception of LDX and STX. Aside from which modes they modify, the registers are autonomous, independent and of equal value.

6.1 ABSOLUTE INDEXED

Absolute indexed address is effective addressing with an index register added to the absolute base address. The sequences that occur for absolute indexed addressing without page crossing are as follows:

Example 6.6: Absolute Indexed; with No Page Crossing

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Increment PC to 101, Finish Previous Instruction
2	0101	BAL	Fetch BAL	Increment PC to 102, Interpret Instruction
3	0102	BAH	Fetch BAH	Increment PC to 103, Calculate BAL + X
4	BAH,BAL+X	OPERAND	Put Out Effective Address	
5	103	Next OP CODE	Fetch Next OP CODE	Finish Operations

BAL and BAH refer to the low- and high-order bytes of the base address, respectively. While the index X was used in Example 6.7, the index Y is equally applicable.

If a page is not crossed, the results of the address low + X does not cause a carry. The processor is able to pipeline the addition of the 8-bit index register to the lower byte of the base address (BAL) and not suffer any time degradation for absolute indexed addressing over straight absolute addressing. In other words, while BAH is being fetched, the add of X to BAL occurs. Both addressing modes require four cycles, with

the only difference being that X or Y must be set at a known value and the OP CODE must indicate an index X or Y.

The second possibility is that when the index register is added to the address low of the base address that the resultant address is in the next page. This is illustrated in Example 6.7.

Example 6.7: Absolute Indexed; with Page Crossing

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Finish Previous Operation Increment PC to 101
2	0101	BAL	Fetch BAL	Interpret Instruction Increment PC to 102
3	0102	BAH	Fetch BAH	Add BAL + Index Increment PC to 103
4	BAH,BAL +X	Data (Ignore)	Fetch Data (Data is ignored)	Add BAH + Carry
5	BAH+1, BAL+X	Data	Fetch Data	
6	0103	Next OP CODE	Fetch Next OP CODE	Finish Operation

The most substantial difference between the page crossing operation and no page crossing is that, during the fourth cycle, the address high and the calculated address low are put out, thereby incorrectly addressing the same page as the base address. This operation is carried on in parallel with the adding of the carry to the address high. During the fourth cycle the address high plus the carry from the adder are put on the address bus, moving the operation to the next page. Thus, there are two effects from the page crossing: the first effect is the addressing of a false address; this is similar to what happens in a branch relative during a page crossing. Secondly, the operation takes an additional cycle while the new address high is calculated. As with the branch relati

this page crossing occurs independently of programmer action and there is no penalty in memory for having crossed the page boundary. It is possible for the programmer to predict a page crossing by knowing the value of the base address and the maximum offset value in the index register. If timing is of concern, the base address can be adjusted so that the address field is always in one page.

As with absolute addressing, absolute indexed is the most general form of indexing. It is possible to do absolute indexed modified by X, and absolute indexed modified by Y. Instructions which allow absolute indexed by X are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, and STA.

The instructions which allow indexed absolute by Y are ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, and STA.

6.2 *ZERO PAGE INDEXED*

As with non-computed addressing, there is a memory use advantage to the short-cut of zero page addressing. Except in LDX and STX instructions which can be modified by Y, Zero Page is only available modified by X. If the base address plus X exceeds the value that can be stored in a single byte, no carry is generated; therefore there is no page crossing phenomenon. A wrap-around will occur within Page Zero. The following example illustrates the internal operations of Zero Page indexing.

Example 6.8: Illustration of Zero Page Indexing

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Finish Previous Operation, 0101 → P
2	0101	BAL	Fetch Base Address Low (BAL)	Interpret Instruction, 0102 → PC
3	00,BAL	Data (Discarded	Fetch Discarded Data	Add: BAL + X
4	00,BAL +X	Data	Fetch Data	
5	0102	Next OP CODE	Fetch Next OP CODE	Finish Operation

As can be seen from the example, Zero Page indexing offers no time savings over absolute indexing without page crossing. In the case of the indexed absolute, during cycle 3 the address high is being fetched at the same time as the addition of the index to address low. In the case of the Zero Page, there is no opportunity for this type of overlap; therefore, indexed Zero Page instructions take one cycle longer than non-indexed instructions.

In both Zero Page indexed and absolute indexed with a page crossing, there are incorrect addresses calculated. Provisions have been made to make certain that only a READ operation occurs during this time. Memory modifying operations such as STORE, SHIFT, ROTATE, etc. have all been delayed until the correct address is available, thereby prohibiting any possibility of writing data in an incorrect location and destroying the previous data in that location.

As has been previously stated, there is no carry out of the Zero Page operation. 00 is forced into address high under all circumstances in cycle 4. For example, if the index register containing a value of 10 is to be added to base address containing a value of F7, the following operation would occur:

Example 6.9: Demonstrating the Wrap-Around

Cycle	Address Bus	Internal Operation
3	00F7	F7 + 10
4	0007	

This indicates the wrap-around effect that occurs with Zero Page indexing with page crossing. This wrap-around does not increase the cycle time over that shown in the previous example.

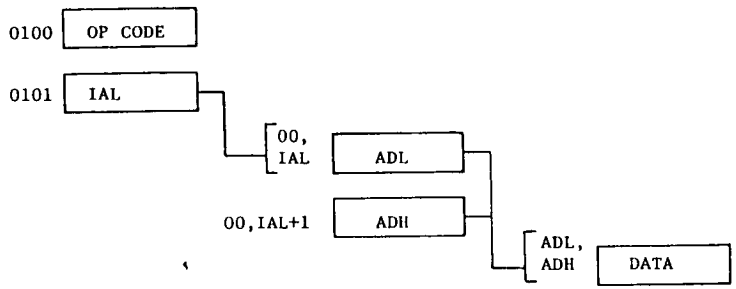
Only index X is allowed as a modifier in Zero Page. Instructions which have this feature include ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA and STY. Note that index Y is allowed in the instructions LDX and STX.

6.3 INDIRECT ADDRESSING

In solving a certain class of problems, it is sometimes necessary to have an address which is a truly computed value, not just a base address with some type of offset, but a value which is calculated or sometimes obtained as a group of addresses. In order to implement this type of indexing or addressing, the use of indirect addressing has been introduced.

In the R6500-series microprocessors, indirect operations have a special form. The basic form of the indirect addressing is that of an instruction consisting of an OP CODE followed by a Zero Page address. The microprocessor obtains the effective address by picking up from the Zero Page address the effective address of the operation. The indirect addressing operation is much the same as absolute addressing, except that indirect addressing uses a Zero Page addressing operation to access the effective address. In the case of absolute addressing, the value in the program counter is used as the address to pick up the effective address low, and one is added to the program counter which is used to pick up the effective address high. In the case of indirect addressing, the next value after the OP CODE, as addressed with the program counter, is used as a pointer to address the effective

address low in the zero page. The pointer is then incremented by one with the effective address high fetched from the next memory location. The next cycle places the effective address high (ADH) and effective address low (ADL) on the address bus to fetch the data. An illustration of this is shown in Figure 6.4.



Indirect Addressing–Pictorial Drawing
FIGURE 6.4

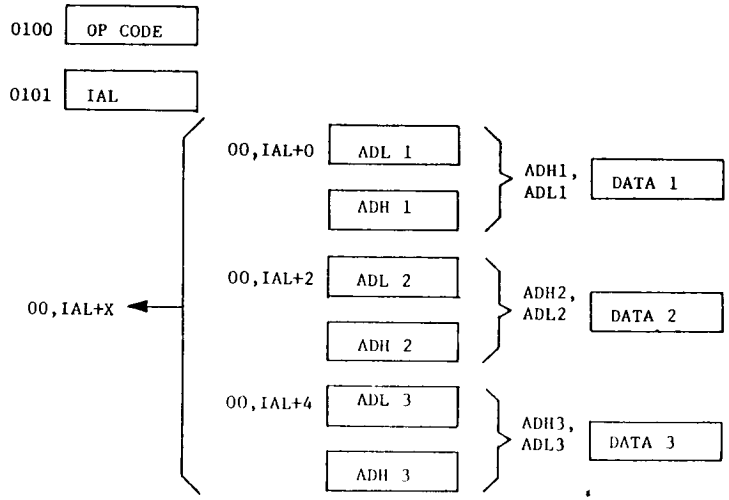
The address following the instruction is really the address of an address, or "indirect" address. The indirect address is represented by IAL in the figure.

A more detailed definition of indirect addressing is included in the appendix.

Although the R6500 System's microprocessor family has indirect operations, it has no simple indirect addressing, except for JMP instructions, such as described above. Instead, there are two modes of indirect addressing: 1) indexed indirect and 2) indirect indexed. The two modes are discussed in Sections 6.4 and 6.5, respectively.

6.4 INDEXED INDIRECT ADDRESSING

The major use of indexed indirect is in picking up data from a table or list of addresses to perform an operation. Examples where indexed indirect is applicable are found in polling I/O devices or in performing string or multiple-string operations. Indexed indirect addressing uses the index register X. Instead of performing the indirect as shown in Figure 6.4, the index register X is added to the Zero Page address, thereby allowing varying address for the indirect pointer. The operation and timing of the indexed indirect addressing is depicted in Figure 6.5.



Indexed Indirect Addressing
FIGURE 6.5