Roy H Braden

$10

IEEE

MICROPROCESSOR UPDATE

MANUAL

APRIL, 1978

THIRD EDITION

by Karl V. Amatneek

IEEE

uP UPDATE MANUAL

BASED ON THE 6502 uP CHIP AND THE PIEEE-77 BOARD

Third Edition

by Karl V Amatneek

April, 1978

Published by the UPDATE Committee of IEEE Philadelphia Section
Charles M Philips, Chairman
Carroll R Williams, Executive VC
Karl V Amatneek, Education VC

CONTENTS

INTRODUCTION

THIS MANUAL IS ADDRESSED TO DESIGNERS OF ELECTRONIC EQUIPMENT. ITS PURPOSE IS TO PROVIDE THE DESIGNER WITH PRACTICAL KNOW-HOW FOR USING THE uP CHIP WHILE AVOIDING COMPUTER JARGON.

PROGRAMMING OF THE uP IS VERY SIMILAR TO WIRING OF CHIPS, BUT THE TECHNIQUES AND SOME OF THE CONCEPTS APPEAR STRANGE AND FORBIDDING TO THE BEGINNER.

WE USE HARD-WIRING TO INTERCONNECT THE uP WITH OTHER CHIPS; WE USE FIRM-WIRING TO MAKE THE SYSTEM PERFORM THE NECESSARY FUNCTIONS. THE uP CHIP LOOKS UP ITS SCHEDULE OF ACTION IN THE FIRM-WIRED "PROGRAM STORE" (LIKE A WIRING LIST) AND SETS UP CONDUCTING PATHS AMONG INTERNAL AND EXTERNAL REGISTERS. THESE PATHS SERVE TO BRING IN SIGNALS FROM THE OUTSIDE WORLD, MANIPULATE THEM IN MAGNITUDE AND IN TIME, AND SEND THEM OUT AGAIN TO CONTROL THE OUTSIDE WORLD.

THE LANGUAGE USED IS ENGINEERING. THE CONCEPTS USED ARE ENGINEERING. THE RESULTS ARE ENGINEERING.

YOU CAN STUDY THIS MANUAL FROM THE BEGINNING AS YOU WOULD ANY OTHER BOOK. BUT IT IS MORE FUN TO STUDY IT WITH THE PIEEE-77 BOARD ON HAND. IN THAT CASE S T A R T W I T H CHAPTER II, THE COOKBOOK, AND BEGIN USING THE BOARD TO GENERATE SIGNALS AND OBSERVE THEM IN LEDs, ON A SCOPE OR WITH A LOUDSPEAKER. AS YOU RUN INTO UNFAMILIAR TERMS, TURN TO THE INDEX/GLOSSARY IN THE BACK OF THE MANUAL AND LOOK UP DEFINITIONS; FOR FURTHER INFORMATION GO TO THE PAGE REFERENCES. ANY TIME, COME BACK TO THE PROGRAMS AND LEARN BY DOING.

KVA

FOREWORD TO THIRD EDITION

Second edition of this Manual was published in Yugoslavia for Informatica-77. This, the third, edition has a number of additions devloped during the year while lectruing to IEEE groups around the country and to industry groups both her and in Canada. The review of uPIEEE-77 Bench Programming Workshop was first presented in Yugoslavia and in Canada.

During the year two interesting developments have taken place -- Commodore, the parent company of MOS Technology, has announced the cheap PET computer; and the 6502 is now also being manufactured by Rockwell International.

James and DiCamillo, our two members who are making the PIEEE-77 board under the Datac name, have started publishing "the datac 1000 users' group", and a number of items from it are being reprinted in this Manual.

KVA

HAND PROGRAMMING OF MICROPROCESSOR-BASED INSTRUMENTS AND CONTROL SYSTEMS

When a circuit designer first hears about microprocessors, he reacts with surprise and disbelief: why must he buy $20,000 worth of "microprocessor development" equipment before he can use the $20 chip? The fact is, with modern hand programming techniques -- unless his project is too large -- he doesn't have to buy any equipment to speak of. Large numbers of engineers are doing microprocessor programming by hand. The advent of cheap single-board systems with hexadecimal keyboard has made hand programming especially attractive.

This is a review of the uPIEEE-77 Workshop on Bench Programming of Microprocessors. Since it was the first such workshop, the Proceedings has become a rich resource book of bench programming techniques. The numbered references are to papers and pages in the Proceedings.

(1) DEFINITION OF BENCH PROGRAMMING

"Bench programming" and "hand programming" are more or less interchangeable terms. Strictly speaking, *hand* programming means writing and entering programs by hand in machine code via a switch register. While the term is also used more loosely, *bench* programming means using inexpensive lab equipment to improve upon strict hand programming.

(2) ALTERNATIVES TO BENCH PROGRAMMING

The two main alternatives to bench programming are (a) purchase of a commercial "uP development system", and (b) rental of a time-share terminal connected to some company's powerful program for developing a uP (microprocessor) program. Both these alternatives are expensive. [63-6, 64-5]

(a) A *development system* usually consists of a microcomputer with a control panel for running it conveniently. Commercial development systems cost 5 to 20 thousand dollars. While a development system may save time in writing a program, it also

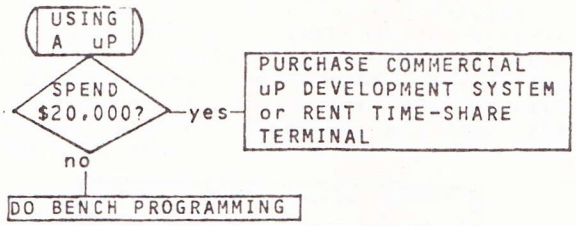


Fig. 1. Main reason for bench programming.

takes time to learn to use it and to learn to use it efficiently; it takes time to service and trouble-shoot it. And, obviously, while the development system is down, no programming can be done. While a development system may be advertised for both program and hardware development, its hardware testing facilities are often inadequate [71].

(b) The *time-share terminal*, of course, is not connected to the uP itself, and so it can not be used to locate wiring errors or to discover places where signals arrive at inappropriate times.

(3) ADVANTAGES OF BENCH PROGRAMMING

Bench programming goes hand in hand with single-board systems: programs can be entered and debugged without the expense or bulk of a teletypewriter. Thus, work and learning can proceed in the evenings and over weekends at home, or even while commuting.

Bench programming is obviously *but not necessarily* time-consuming; when combined with some lab equipment, built as the need indicates, or with inexpensive commercial equipment (complementing single-board computers), it can result in a saving of time. [62-1, 63-2, 63-6]

Bench programming forces one to gain a more intimate knowledge of the uP and its capabilities. In any case, knowledge of hand programming is required for patching (repairing) programs while debugging. Finally, bench programming leaves one free to use any uP, whereas development systems may freeze one into a particular uP chip.

(4) DISADVANTAGES OF BENCH PROGRAMMING

In preparing programs by hand, as compared to computer-aided preparation of programs, there is more chance of making an error; moreover, modification of programs requires laborious rewriting so as to clear up inserts. To quote Peatman, hand programming may be practical, but it is "tedious".[Peatman] Stratagems and equipment for counteracting these difficulties are offered in many papers in the *Proceedings*.

It must be noted here that there are also commercial and emotional objections to hand programming. We inherited the uP chip from computer art, and computer people automatically teach us to use mnemonics, assembly language and the assembler, as if that was the only way to make the uP work. They also want us to use their "high level" languages such as Basic, which is quite unsuitable for uControllers [12-3].

Computer programmers, who often don't really understand control hardware, find it "unnatural" to consider any other way of programming but the "missionary" way which they themselves use to program computers. They consider bench programming obsolete and obviously uneconomical, and proponents of bench programming as frauds [personal communication]. To them it seems we are going back to the ice age.[63-5]

Knowing this attitude of computer programmers, manufacturers of uP chips produce development systems. To a uP chip manufacturer the great virtue of a development system lies in the fact that this $5-20K investment will make a user hesitate to switch to a chip by a different manufacturer that would again require a similar investment. Manufacturers of more versatile development systems, who do not themselves make uP chips, simply hate to lose potential sales to bench programming.

And then, of course, there are the ambitious managers who should know better, and the young engineers who don't know any better: they like to show off by doing things in an "elegant", "sophisticated" or "modern" way regardless of cost. Of course, much technical satisfaction is derived from punching the typewriter and seeing things happen.

(5) PRACTITIONERS OF BENCH PROGRAMMING

Despite this commercial and emotional opposition, there are many small and large practitioners of bench programming. Among them are groups in Western Electric, RCA and Essex Wire, for instance. All of them have successful products on the market that were designed in bench programming fashion.

One 5-million dollar company exists on bench programming alone.[21, 22] And the very first bench programming conference, uPIEEE-77, attracted 95 practitioners of bench programming from large and small companies.

(6) uP BOARDS WITH HEX KEYBOARDS

Bench programming has been practiced ever since uPs were first invented 6 years ago. The outstanding advocate and practitioner is the Pro-Log Corp.[21, 22] Bench programming took a big step forward with the proliferation of single-board systems equipped with a hex keyboard for entering programs in machine code. Single-board computers made by MOS Technology, Motorola, National Semiconductor, RCA, as well as many smaller companies, all have the hex keyboard.

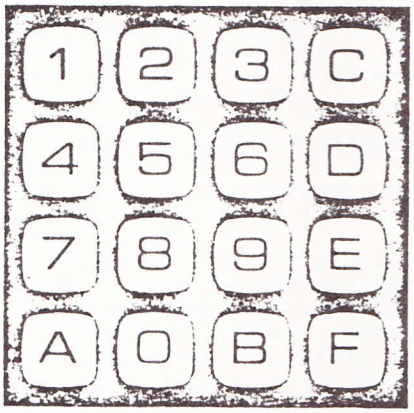


Fig. 2. A hex keyboard.

Programs to control the real world can be developed on paper starting with a high-level statement of the task to be accomplished, and then making it more and more detailed.[12] The program is finally translated into hex code and hand-keyed into temporary program store (memory chip) on the board.

For storage overnight, the temporary program may be stored either in a BRAM (Battery-supported memory) [63, 65-3], or in an inexpensive music cassette.[63-6]

After a temporary program has been proved out, it may be *burned* into a UVROM (Ultra-Violet Erasable Read-Only-Memory). A UVROM burner may be constructed by the user, or else commercial UVROM burners [22-7] may be purchased at various prices. Some inexpensive UVROM burner cards will automatically burn in the program directly from the temporary program store, without further hand keying.

The PIEEE-77 board developed for the IEEE [Amateek] and the OSI-300 board of Ohio Scientific Instruments, have a binary

switch register for hand programming. Intersil has a single-board computer that is equipped with a keyboard where each key is labelled with an instruction.[33]

All of these boards also have single-step and other circuitry to assist program debugging and hardware trouble-shooting. Some have cassette storage and retrieval programs. Some have sockets wired for UVROMs plus unwired sockets for application requirements. The PIEEE-77 board has a speaker for audio monitoring and two LEDs for visual monitoring or trouble shooting.

(7) ASSIST EQUIPMENT AND PROGRAMS FOR VARIOUS STAGES OF BENCH PROGRAMMING

Bench "programming" encompasses (a) writing the program in near-natural language and detailing it to just-short of machine language, (b) one-by-one coding it into machine language and entering the code into temporary program store, (c) debugging the program so that it works and so that it does what it is supposed to do, (d) if any of the hardware in newly designed, trouble-shooting it for signal mistiming and for errors in wiring.

Work on these 4 stages may be speeded up, and errors may be reduced, by using hardware and software assistance.

(a) Programming proper. For writing and developing the program on paper, some preprinted form is useful.[21-5, 43-7] Some progress has been made toward providing assist software for this phase.[12-10]

(b) Coding and entering the program into temporary program store. Not only is coding of the program time-consuming, it is also very error-prone; and coding errors are difficult to locate. These troubles could be greatly reduced by a program, running on the uC itself, which would translate a program written in engineering symbols (from a $60 typewriter keyboard [44]) directly into machine code, and store it in temporary program store. [43]

(c) Program debugging. Program debugging can be done with the assistance of hardware (circuits for stopping the uP after each program step so that registers may be examined and modified without the uP);[61] or it can be done in software -- using the uP with a special program such as "breaking" the program at suspicious points and examining and modifying contents of registers.[62]

Program development can also be done by connecting the uP board (the "target") to a separate uController (the "host") with a special program for exercising the target uP. The host may be a commercial uComputer, unrelated to the target uP [64, 65], or it may be similar to the target uP.[63]

(d) Trouble-shooting of new hardware. If the controller will contain new circuits, there will be signal timing and miswiring problems that will have to be located and corrected. Program debugging schemes also have provisions for hardware trouble-shooting.[61 to 65] The readouts may be digital, or there may be provision for triggering a lab scope.[62]

Regardless of what debugging scheme is used, much debugging and trouble-shooting time may be saved by snatching a sequence of the flow of addresses and data on their buses while the uP is running at full speed. This stored information can then be searced thoroughly and methodically for program, wiring and timing errors. Such data analyzing equipment may be built up on a card [71] or it may be purchased for a few hundred dollars.[72]

(8) TECHNIQUES OF BENCH PROGRAMMING

If we choose to do our programs at the bench, how do we go about it?

Bench programming is not a single, rigid method; rather, it is a search for techniques that will produce reliable programs cheaper and quicker. With that in mind, the following is a likely procedure at the present time; we will discuss it item for item.

1 Write the program
2 Choose the uP and translate the program into machine code
3 Test run the board with substitute I/O
4 Develop the real system

1 WRITE THE PROGRAM

Designer writes a uP-independent sequence of labelled steps -- the program -- that will satisfy the requirements:

1A Ascertain performance requirements

User provides a set of requirements to which the instrument is to perform. Note that at this point no uP need be specified.

1B Choose a high-level ("systems") language

The program is written in natural language so that the user can understand it. However, it is written in a special, short, stylized, easy-to-learn format, so that it may eventually be expanded easily into uP steps. One such language is TLC.[12]

Say, the instrument is to play a tune -- a sequence of square waves of stated duration *dn* and stated period *pn* such as in this list: *d1 p1, d2 p2, d3 p3,* etc. [43] The program could read as follows:

```
loop
    do FETCH_DURATION from list
    do FETCH_PERIOD from list
    do PLAY_NOTE on speaker
    do INCREMENT_NOTE_POINTER to point at
         next note
pool
```

The four capitalized statements are *labels* to be spelled out in detail later. The underlines _ are connectors to join the words into a single label. "pool" is "loop" spelled backwards; these are eventually to be spelled out in the machine code of the uP.

1C Interaction between user and designer

If the control instrument is to be multi-mode (multi-purpose), the user will need one or more keys with which to choose the desired mode. If the number of modes is large, or if the user will also have to enter numerical values, a special keyboard program may be useful, such as AOK (Applications-Oriented Keyboard).[42, 11-3]

Since the program uses spoken language, the designer and user interact to assure that, on the one hand, the indicated requirements are in fact true, and that, on the other hand, the proposed sequence of steps will indeed satisfy these requirements.

For instance, in the above example, the user may bring up a thought that had not occurred to him before: let the tune be repeated if desired. The program is then modified to take care of this new requirement:

```
loop
        set NOTE_POINTER=START
    loop
        do FETCH_DURATION
        if DURATION=00,
          if REPEAT_ENABLE=ON
            exit this loop
        do FETCH_PERIOD
        do PLAY_NOTE
        do INCREMENT_NOTE_POINTER
    pool
pool
```

Now, when the DURATION of the next note is 00, and if the repeat is desired (REPEAT_ENABLE has been turned ON), the program will exit from the inner loop and jump back to fetch the first note by setting NOTE_POINTER to START value.

1D Detail the program

The designer then spells out in detail how the labels will be executed (short of getting involved in uP mechanics).

Here the modern viewpoint is that, first and foremost, programs must be easy to understand, easy to debug and easy to modify. To achieve this, 3 buzz-words are currently in vogue: Top-down sequence, "structured" instructions [11]; and modular subdivisions.[12-2, 21]

Top-down sequence means disciplining yourself to work on developing the program in an orderly fashion, starting with the statement of requirements (top) down to machine language, without jumping back and forth.

Structured instructions means using only a limited number of chosen instruction sequences (structures). Structured programming prohibits multiple entrances and exits, and it prohibits the jump instruction with absolute address. Moreover, only 5 structures are permitted: linear; if-then-else do a relative jump; do while; loop; and subroutine.

The reason for this delimiting of the freedom of choice lies in the fact that most computer programs have been difficult to comprehend. (Of course, a structured program is more difficult to write.[11-5])

Modular subdivisions means programs subdivided into portions not more than one typewritten page each, each page (module) performing a "do this" function, or a set of such functions. The purpose of this approach is to make each module comprehensible to the eye at a glance.

1E Decide on interfaces to the real world

As the designer spells out each step of the program into more and more detailed actions, he makes his choices of what interfaces (from TTL voltages to the real world) he will need to bring the program steps to practical realization.

2 CHOOSE THE uP AND TRANSLATE THE PROGRAM INTO MACHINE CODE

When the user has approved the final program, the designer can choose a uP (if there is a choice) by translating samples of the program into the machine language of each uP.[41, 43-3,4] He can base his choice on the difficulty in translating; on the number of bytes required; and on the final speed of execution. These three parameters are likely to be different from uP to uP. If the three parameters are not important to the application, any uP can do the job.

Having chosen the uP, the approved program is translated line by line into machine code.

3 TEST RUN THE BOARD WITH DUMMY I/O

At this point all the paper work is put to the test against the merciless logic of the uP. It is nearly impossible to write a program that will run on the first try, but using the TLC language to develop the program is a significant step forward.

3A Obtain board and make I/O dummies

Having chosen the uP, the designer obtains a ready-made uP board (which can later be redesigned for a production run if necessary). He then breadboards input and output substitutes (dummies) so that he can run and test the program conveniently.

3B Enter program into temporary program store, and debug it

At this point the machine language program is copied by hand into uC memory and run with dummy inputs and outputs. Any of the schemes in (7) above may be used for further debugging and program development.

3C Burn-in UVROM and exercise the board

When the program is apparently bug-free, it may be burned into a UVROM (Ultra-Violet-erasable Read-Only Memory), and attention is turned to making the program reliable: an "exerciser" program may be written to permutate through all possibilities of input and output combinations and to catch any that don't work.[63-2]

4 DEVELOP THE REAL SYSTEM

When the program is deemed ready, the interfaces and the real, full system are connected up and run. As at previous stages, trouble-shooting and final program changes will be necessary. When the system is running smoothly, and no further trouble is expected, the final program is burned into the UVROM again, and the uController is then deemed completed.

(9) SUMMARY

A uP may be incorporated into a control instrument without spending large sums on purchasing special equipment. The method is known as bench programming. The program is first developed on paper using natural language; the user and designer cooperate in verifying it. The designer then chooses a microprocessor and codes the program into its machine language. The single-board microcomputer that will become the control instrument is first used for debugging the program.

BIBLIOGRAPHY

Reference numbers correspond to the numbering of papers in Proceedings uPIEEE-77, *IEEE Catalog No. EHO 125-5, $20.*

11 *Lance A Leventhal*, Can structured programming help the bench programmer?

12 *Tony Karp*, TLC -- a new systems language.

21 *Matt Biewer*, The engineering design approach to microprocessors.

22 *Edwin Lee*, Design and document microprocessor systems for easy maintenance.

31 *Gregory Zick, Jerry Vanaken*, Comparison of 16-bit microprocessor architectures.

32 *Paul S Mitzen*, Microcoding an MSI chip controller.

33 *Gopal Ramachandran*, Development of microinterpreter for Intercept Jr.

41 *Russ Walter*, Universal assembly language -- a quicker way to understand microprocessors.

42 *Erich A Pfeiffer*, Applications-Oriented Keyboard languages for small microprocessor systems.

43 *Karl V Amatneek*, No-language programming.

44 *John Prenis*, A keyboard for an engineering language programming system.

45 *John Buffington*, E/L, a universal assembly language notation.

51 *Robert S Chen, Rajeev Sangal*, A design to share memory among microprocessors.

52 *R L Krutz*, Parallel programmed logic elements.

61 *Bernard Carey, Michael Varanka*, Control box for programming, debugging and trouble shooting.

62 *Dwight B Sawin III, Thomas P Hughes*, Real-time microprocessor software debugging techniques.

63 *Thomas Y Chen*, Development of simple function test card for the RCA Studio II, a microprocessor-based video game.

64 *Norman Rosenfeld*, Development of microprocessors and microprocessor-based systems.

65 *Tony Karp*, A low-cost, machine-independent system for microprocessor hardware and software.

71 *William M Goble*, Two hardware circuits + microprocessor = quick trouble-shooting.

72 *Gerald F Muething*, Low-cost logic analysis.

Amatneek, *Karl V Amatneek*, IEEE Microprocessor UPDATE Manual, June 1977. Publ by Committee on Professional UPDATE, Philadelphia IEEE, Univ of Penna Moore School, Philadelphia, PA 19104, $10.

Peatman, *John B Peatman*, Microcomputer-Based Design. Pub 1977 by McGraw-Hill, NYC.

THE GENERAL IDEA OF A uP: A BIRD'S EYE VIEW

1 WHAT IS A uP?

The uP chip replaces a portion of the computer known as the CPU, Central Processing Unit. The CPU is not an independent unit. To be sure, it itself performs logic and arithmetic manipulation of signals, but it also manipulates other units to route signals in from the outside and out again. It does all this one thing at a time, but at megahertz rate. It has been compared to a one-armed paper hanger.

It can do nothing alone. At the very least it needs a program store; i.e. a ROM chip for storing the sequence of instructions that it is to execute; and an I/O (Input/Output) chip that will receive and transmit signals between the outside world and the uP chip.

2 WHAT IS ALL THE EXCITEMENT ABOUT?

As compared to conventional logic chips, these are the advantages of uP systems:

- Because the uP is an LSI and because it has universal application, it is bargain-priced.

- Because it is a single chip, it takes less space.

- Because it is a single chip, and there is market pressure to design intelligent assist chips, the parts count is less. (Single-chip microComputers such as the 8048 are already on the market.)

- Because it is programmable, errors in design, or changing customer requirements do not require a re-design of the board: only reprogramming is required.

- Off-the-shelf mass-produced uP boards may be applied to many different custom designs without any modification of hardware. Only program needs to be rewritten.

3 WHAT DOES A uP DO?

- While conventional logic chips process one bit at a time, a uP handles several bits (4, 8, 12, 16) simultaneously, depending upon the particular chip.

- A conventional chip (AND, OR) can perform only the one process for which it was made, while a single uP has as its repertoire *several dozen processes*, and it performs any one of them on demand.

- Conventional chips are hard-wired in the system -- they can execute only what they have been wired to do; while a uP will perform any desired sequence of processes *without any change of wiring*. The sequence is entered into erasable UVROM chips, and it is called a program.

+ Since it is a single chip, it increases reliability of a system in which it is used.

4 WHAT DOES A uP ACCOMPLISH?

In a system, a uP
(1) enables the input terminals (reads input data),
(2) manipulates the signal (performs arithmetic and logical manipulation) and makes procedure decisions, and
(3) turns on the output terminals (writes data).

5 WHY ARE OTHER CHIPS REQUIRED?

To be able to operate at all, a conventional uP chip requires a program store and input/output adapters. The program store instructs the uP what sequence of operations is required. The input/output adapter is required so that inputs may be multiplexed into the uP and so that microsecond duration outputs may be latched to last long enough for practical applications. A minimum working system consists of a uP chip connected on one side to a UVROM chip that contains several hundred steps of a program store; and on the other side to an I/O adapter chip, so-called, that has a number of pins (say 16) through which signals are received from and transmitted to the outside world.

6 HOW DO YOU WORK A uP SYSTEM?

Aside from connecting up the several chips to the uP chip, the main job is to write the program -- the sequence of steps that the uP is to perform. Once both wiring and programming are successfully completed, the system will do the job it was designed to do as soon as power is turned on.

HOW MUCH DO THE MAIN CHIPS COST?

In unit lots the 6502 is $25, the 2708 is $35 and the 6520 is $9.75. The price of the new computer-on-a-chip, the 8748, is $275; ROM version, 8048, $10.

IS A MINIMUM uP SYSTEM A PROPER COMPUTER?

A minimum uP system becomes a *dedicated* computer after it has been programmed. It is not a *general-purpose* computer if it does not have lots of memory chips. A general-purpose computer has to have lots of memory because you never know how big a problem it will have to handle some day.

EVERYTHING YOU EVER WANTED TO KNOW ABOUT uPs

REGISTERS

REGISTERS
- A uP system consists of registers.
- Registers inside the uP chip are lettered (A, X, Y, etc).
- Registers outside the uP are numbered. (from 0000 to FFFF).
- The sum total of these numbers is the address space.
- 16 bits will accomodate an address space of 64K (65,536) registers.

ADDRESS
- The register numbers are called addresses.
- If 64K addresses were listed in a book, the book would have FF pages of FF lines each.
- Any address consists of a page number and a line number.
- A 4-digit address consists of a 2-digit page number and a 2-digit line number.

DATA
- The numbers stored in registers are called data.
- A 4-bit register would hold a single hex digit (nybble).
- An 8-bit register holds 2 hex digits (byte).
- To hold 4 hex digits, 2 8-bit registers are needed.
- *All work of the uP is done by shuffling data in registers.*

INSTRUCTIONS

OP-CODES
- A uP can perform many ready-made functions numbered from 00 to FF.
- Each of these numbers is called an operation code or op-code.
- An op-code consists of two hex digits, thus:A1, 27, CC, etc.
- Each op-code tells the uP what operation to perform.

OPERANDS
- The operand tells the uP on what number to perform that operation.
- The number following the op-code will be called the operand.
- Usually the operand is a 2-digit or 4-digit address.
- The operation called for by the op-code is performed on the contents of that address.
- Some op-codes require only the line number or else no operand at all.
- In an op-code that requires no operand the operand is implied.

INSTRUCTION
- The op-code and operand together are called an instruction.

PROGRAM
- The written list of instructions (like a wiring list) is called a program.
- The program consists of op-codes and operands.

PROGRAM STORE
- The program is copied from a piece of paper into the program store.
- The program store nowadays is usually a UVROM.
- While a new program is tried out it is temporarily stored in RAM.
- While programs are being worked on, they are stored overnight in cassettes.

HOW THE uP WORKS

PROGRAM COUNTER
- One of the many registers in the uP is the program counter.
- The program counter sends consecutive addresses to the program store.
- In response to each address the uP receives an op-code or operand byte.
- The only byte sure to be an op-code is the first byte in the program.
- The programmer must keep careful track of subsequent bytes.

ADDRESSING
- For operating the numbered registers, the uP issues an address signal on the address bus.
- When a system is not halted, the only source of addresses is the uP.
- The address bus of the 6502 cons ists of 16 leads running in parallel from the uP to most of the other chips.

DURATION
- The address signal on the 16-bit address bus lasts one uS.
- This address duration is sufficient to pick out the required register.

DATA
- Simultaneously with the address, a data signal is put on the 8-bit data bus.
- The data bus consists of 8 leads running to all chips in parallel.
- The data signal also lasts one uS.
- This data duration is sufficient for most chips but too short for the real world.

INPUT/OUTPUT REGISTERS

OUTPUT DATA
- Output data is captured from the data bus in a latched register at the right moment.
- The right moment occurs when the proper address signal is issued.

INPUT DATA
- Input data is gated onto the data bus at the right moment.

EVERYTHING YOU EVER WANTED TO KNOW ABOUT uPs -2-

SEQUENCE OF OPERATIONS

POWER-UP CIRCUIT
- A complete uP board contains a circuit that senses the d-c line going on.

RESTART SIGNAL
- When power is turned on, the power-up circuit delivers a long restart signal to the uP.
- The restart signal starts off the uP on the programmed sequence of operations as described below.
- The restart signal is also applied to the input/output chips (6520 and 6530).
- When the I/O chips receive the restart signal they disconnect themselves from the real world(but not quite! Be careful!)

RESTART POINTER
- The restart signal makes the uP look at the restart pointer in the program store. (In the 6502 the restart pointer is at addresses FFFC&D).

FIRST LINE OF PROGRAM
- The restart pointer register contains the first line of program, i.e. the address where the very first op-code is stored.

FETCH OP-CODE
- The uP looks up the reset pointer and issues that address on the address bus.
- In response to that address the program store sends the first op-code.
- The uP decodes the op-code and figures out the number of bytes in the operand.

FETCH OPERAND
- The uP then sends out the next address to fetch the first byte of the operand.
- In the 6502 this address must contain the line number where the data is stored.
- The uP then sends out the next program address to fetch the rest of the operand.
- This address contains the page number where the data is stored.

FETCH DATA
- The uP now sends out the address furnished by the operand.
- The uP finds the data at this address.

EXECUTE OPERATION
- The uP next executes the operation.

FETCH OP-CODE
- Now the uP fetches the next op-code from the program store.
- The uP decodes the op-code and determines the number of bytes in the operand.

IN THIS WAY THE uP WORKS ITS WAY THROUGH THE WHOLE PROGRAM.

LAST LINE
- When the program proper is finished, the uP would normally read the random data in the next line treating it like an op-code.
- Interpreting resident random number as an op-code may wreak havoc with the program and with real world equipment connected to the uP system.

JUMP ROPE
- To prevent this from happening, after the last active line we insert 'jump rope' instruction.
- A jump rope instruction keeps the uP jumping in one place.

CONTINUOUS LOOP
- In a controller there is no "last line".
- In a controller the uP may run in a continuous loop monitoring the equipment

CAPABILITIES OF THE uP CHIP:

INSTRUCTION SET AND ARCHITECTURE

Let us review the operations that a uP is capable of performing. We will use the 6502 as an example. We can order it to do the following kinds of operations:

(1) A&hh, i.e. AND the Accumulator with a number hh.
(2) X+1, i.e. increment register X.
(3) A=X, i.e. copy into the Accumulator the 8-bit data (signal) in register X.
(4) if 0,J+9, i.e. if the result of the last preceding calculating operation was zero, then skip (jump, branch) the next 9 program bytes.

The above examples all take place inside the uP, but there are also similar operations that involve external registers such as those for sensing and controlling the outside world as well as those for the "peripherals" -- displays, keyboards, memory.

The computer people historically don't understand such a simple approach. Because they do not deal in signals but rather in voluminous calculations and texts, they need the help of a language with mnemonics (from the Greek, probably meaning difficult to remember) or Fortran, Basic, APL, high-level languages, low-level languages, applications oriented languages, machine independent languages, etc.

As circuit designers, the idea of computing and manipulating a signal by simply writing down the desired step is appealing to us -- no solder, no wire, no chips, no rejects, no real-estate. Just write down what has to be done in pencil on paper, look up the op-code for the particular uP and punch the op-code into RAM registers! Instead of connecting up 8-bit registers to an 8-bit full adder, we write "to the number (signal) in the Accumulator ADD the number (signal) in the X-register" or simply A+X. If you wish to call the plus-sign a language, you call this engineering language.

ENGINEERING LANGUAGE

True, some new symbols will have to be added to the traditional ones. For instance, in a uP we are able to shift the whole register one bit to the left or to the right into a one-bit register known as Carry. The left (<--) and right (-->) arrows are obvious choices for this purpose. If the programming is done on a typewriter, we can pencil in the arrows, or we can get used to symbols such as SL and SR for "Shift left" and "Shift Right".

We have to become accustomed to some new logic symbols as well because the + sign is preempted for addition. We can use "&" for AND, "V" for inclusive OR and "¥" for exclusive OR.

No symbols are available for jumping to another point in the program sequence, and for that the capital J seems good -- since it is not otherwise used.

LIST OF ENGINEERING SYMBOLS

In the back of this manual there is a list of symbols that have been used and found practical. Most of them are obvious—some require getting used to.

SYMBOLS UNIVERSAL FOR ALL uPs

If such symbols are used, then every uP may be programmed by using the identical symbols. This doesn't appear to be saying much, but in view of the multitude of difficult assembly languages -- a different one for each uP -- this is quite an advantage. These symbols will be used throughout this manual.

I INSTRUCTION SET:

(A) OPERATIONS WHICH MANIPULATE THE SIGNAL

In addition to AND, OR, EXOR, NOT, the uP can perform dozens of other operations. It does them one register (8 bits) at a time. (Some uPs have 4-bit, 12-bit and 16-bit registers).

As an example, the full instruction set of the 6502 and a table of the symbols used is shown in chapter 6.

Here is a summary of operations that manipulate the signal.

(1) LOGIC: &, V, ⊻ (and, or, exor).

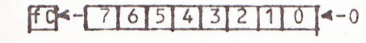
(2) ARITHMETIC: +, -, =, ↓, ↑.

↓ is PUSH, and it means the same thing as "=", namely "into first register copy contents of second register". Here the first register is an automatic "stack" register where the address is decremented after every operation.

↑ is POP, and it is the reverse of PUSH. The address is automatically INcremented each time.

(3) MULTIPLY AND DIVIDE BY POWERS OF TWO: ◄-, -►.

◄- is SHIFT LEFT. The contents of every bit are shifted into its neighbor on the left. Bit 7 goes into carry flag fC. Bit 0 receives a zero. The resulting num-

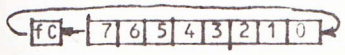


ber is 2x the original number. Think about it.

-► is SHIFT RIGHT. The result is ½ the original number.

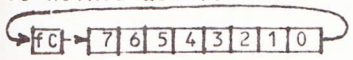
(4) SEPARATE INDIVIDUAL BITS: ↻, ⊆.

↻ is ROTATE LEFT.



Every bit is brought into the carry flag one at a time. There it can be examined using an IF instruction, or it may be modified before returning it back to the register.

⊆ is ROTATE RIGHT.



II ARCHITECTURE:

PROGRAM COUNTER -- THE ORDER IN WHICH THE uP READS THROUGH THE INSTRUCTIONS

The program counter register, pC, points at the next address in the program store. Normally the program counter is incremented automatically as each instruction is read by the uP. If the result of an operation is not the normal one, then the program counter JUMPS to another part of the program.

There are 7 occasions on which the program counter jumps instead of incrementing:

(1) IF-DECISION: If the result of an operation is not the expected one, there is a small jump (branch) to an adjacent part of the program written to handle that situation, and to slip right back into the same program.

(2) SUBROUTINE: If the next step in the program is written out in detail elsewhere -- to be executed and then to jump back to the normal order -- then the program counter jumps to this SUBROUTINE and jumps back when finished.

(3) PROGRAM BREAK: During trouble shooting (DEBUGGING) of a program, it is convenient to have a short BREAK sequence to help in analyzing the problem. Its starting address must be written into the BREAK pointer registers (FFFE&F in the 6502). The BREAK instruction (00 in the 6502) is temporarily inserted -- like a probe -- at strategic points in the program. It is removed when trouble-shooting is over.

(4) HARDWARE REQUIREMENT: When a slow piece of outside equipment which is sensed or controlled by the uP board is ready to communicate, it sends a signal to the INTERRUPT REQUEST pin, IRQ/, of the uP chip (#4 on the 6502). The uP looks up the starting address of the appropriate piece of program in the IRQ/ POINTER registers (same as BREAK in the 6502), and the program counter jumps to point at that address.

(5) UNCONDITIONAL JUMP (frowned upon): for patching in a new piece of program for which no room is available at that point in the program.

(6) INTERRUPT IMMEDIATELY (NMI): When supply voltage falls low enough to fear a power failure, in the last few milliseconds the uP can still DUMP important registers into a non-volatile memory (if such is indeed on the

board). If such a low-line sensor is installed on the board, it signals the NMI pin of the uP (#6 of the 6502), and the uP makes the program counter jump to point at the address given in the NMI, Non-Masked-Interrupt, pointer registers (FFFA&B in the 6502). At that point in the program, an appropriate sequence has to be written to implement the DUMP.

(C) RESTART If, while running the program, the operator wishes to start from the top, he HALTS the uP and touches the RESTART switch, which sets the program counter to point at the top line of the program, as shown in the RESET POINTER registers (FFFC&D in the 6502).

Obviously all the above pointers (sometimes called VECTORS) have to be written into the program before the real business of the program starts. This housekeeping chore is part of what is known as INITIALIZING.

III INSTRUCTION SET:
(B) INSTRUCTIONS THAT MAKE THE PROGRAM COUNTER JUMP

To sum up, the program counter jumps on the following 5 instructions:

(B) IF result of previous operation is X, then J+hh. These instructions are described in section VI below.

(B) Memorize program counter reading and jump to subroutine ppll; later J:ret S. ppll are page and line number where the subroutine is located.

(B) Memorize program counter reading and jump to breakroutine (jumps to pointer FFFE&F); later J:ret B.

(B) J:ppll or J:(ppll*). ppll* means that address and the next one.

(B) J+1. This is an "idle" instruction that increments the program counter without doing anything useful. It may be used as a short time delay (2 cycles).

The program counter is also made to jump by grounding one of three uP pins:

(B) It jumps to pointer FFFA&B when pin 6 (NMI/) is grounded. It is returned by program instruction J:ret I.

(B) It jumps to pointer FFFC&D when pin 40 (IRQ/) is grounded.

(B) It jumps to pointer FFFE&F when pin 4 (IRQ/) is grounded. It is returned by program instruction J:ret I.

To repeat, the program counter jumps on the following conditions:

IF-CONDITION
SUBROUTINE
BREAK INSTRUCTION
JUMP INSTRUCTION
IDLING INSTRUCTION
6 Grounding of pin 6 (NMI/).
7 Grounding of pin 40 (IRQ/).
8 Grounding of pin 4 (IRQ/).

IV ARCHITECTURE:
FLAGS -- A SCRIBBLE PAD OF INTERIM RESULTS

So that the uP can evaluate the results of an operation and make decisions for further processing, the significant results of the operation must be jotted down somewhere. A set of one-bit latches, FLAGS, has been provided in the uP for this purpose. The flags are set or reset automatically while certain operations are being performed.

The following flags in the 6502 are typical:

(0) fC, Carry flag. When Bit 7 overflows,..... fC=1

(1) fZ, Zero flag. When the result of an operation leaves zero in the register,
..... fZ=1

(6) fV, oVerflow flag. When bit 6 overflows,..... fV=1

(7) fN, Negative flag. When the result of an operation makes bit 7=1,..... fN=1

There are also 16 trial instructions that will set flags without going thru the actual computation. For example,
fl:A-hh
will set flags as if the A-hh instruction had been carried out, yet the the contents of the Accumulator will not change.

The two trial instructions
fl:A&(Zll), and
fl:A&(ppll)
will in addition set fV=bit 6 and fN=bit 7.

V INSTRUCTION SET:
(C) INSTRUCTIONS WHICH MANIPULATE THE FLAGS

In section IV the flags were set and cleared to correspond to certain results of register-manipulating instructions.

Two of these flags, fC and fV, may be manipulated by the program; i.e., fC=0, 1, and fV=0.

Finally, there is a one-bit latch that is not affected by any operation, but stays whichever way it is set by the program:

(3) fD, Decimal mode flag. While fD=1, binary calculations are done in the uP in decimal numbers; i.e., binary-coded-decimal rather than HEX.

Another flag which is set and reset by uP operations is the Interrupt request disable flag, fI. It is automatically set during hardware RESET and INTERRUPT operations to prevent another interruption before the present one is over. It may be set and reset by program instructions fI=1 and fI=0.

(2) fI, INTERRUPT DISABLE flag. While fI=1, interrupt requests, IRQ\ at pin 4 are ignored.

When a subroutine, a break or an interrupt takes place, the uP automatically stores all the flags in the STACK and returns them at the end.

None of the flags, or internal registers, are accessible for data display and inspection. However, there is an operation, rF↓, that will copy all the flags into one register of THE STACK (which is located on page 1), and any external register, of course, may be displayed. The numbers in parentheses above are the bit numbers of the flag register, thus:

Bit	Flag	Function
(7)	N	Negative
(6)	V	oVerflow
(5)	-	Reserved for future use
(4)	B	Break
(3)	D	Decimal
(2)	I	Interrupt & disable
(1)	Z	Zero
(0)	C	Carry

Fig 12.1 Flag register

VI INSTRUCTION SET:
(D) DECISION INSTRUCTIONS

In the 6502 instruction set there are 8 IF-decision instructions. Each has the format "if X,J±hh". It means "if condition X exists, jump hh bytes forward or back".

Condition X is the state of one of the flags, such as "if fC=1..." In the course of running a program, when the uP comes to an IF instruction, it checks the state of the specified flag. If the flag is in the state specified, the jump takes place. If not, the program counter takes it to the next op-code.

Remember that by the time the operand hh has been read by the uP, its program counter is already pointing to the next op-code. Therefore any jumps forward or back have to be counted from there. This has to be done in signed hex! This is tricky -- see hex chart.

Here is the set of IF-instructions:

(1) if result was zero: if =0,J±hh.
(2) if result was not zero: if ≠0,J±hh.
(3) if result was zero or signed positive, i.e. between 00 and 7F: if POS,J±hh.
(4) if result was signed negative, i.e. between FF and 80: if NEG,J±hh.
(5) if fC=0,J±hh.
(6) if fC=1,J±hh.
(7) if fV=0,J±hh.
(8) if fV=1,J± hh.

VII INSTRUCTION SET:
(E) ADDRESSING MODES

Addressing is a simple concept. The "normal", missionary way of stating an address is to write the page and line number, for instance: A=(pp ll). This is known as ABSOLUTE addressing. However, there are occasions when this is unnecessarily cumbersome. There are 6 shortcuts for such occasions.

(1) INDEXED INSTRUCTIONS

If there is a list of addresses whose contents are needed sequentially, there ought to be a simple way of writing just the first address and letting the uP do the drudgery of going through all the addresses. There are special instructions that do just that. These instructions are said to be written in the INDEXED MODE. There are five such in the 6502 instruction set.

(2) SAVING ADDRESS BYTES WITH ZERO-PAGE INSTRUCTIONS

This type of instruction permits writing a foreshortened address pertaining to page zero only. In these instructions only the line number has to be written, thus: A=(Zll).

(3) ACCESSING LISTS OF ADDRESSES WITH INDEXED INSTRUCTIONS

Of the five indexed instructions in the 6502 three are straight-forward. A starting address is written down, and then the number in the INDEX register (X or Y) is added to this address. As the index number is incremented, new sequential addresses are created automatically.

There are three such automatically-addressing instructions:
A=(Zll+X)
A=(ppll+X)
A=(ppll+Y).

The following is an example of how they would be used in a program:

```
X=00                      loop[hh]
A=(ppll+X) <------------- ↑ fetch NUMBER
J:sub READ.ACCUM          | READ.ACCUM
X+1                       |
fl:X-hh                   |
if /0,J-5 >---------------| pool
```

The loop jumps around hh times, and each time the next register in sequence is read into the Accumulator, starting with zero.

In this example loop[hh] means "go around the loop hh times." "pool" is loop spelled backward; it indicates the return point of a loop.

(5) INDEXED INSTRUCTION WITH CHOICE OF LISTS

On occasion the ultimate user of the uP board will have a choice of several functions to perform (i.e. several lists of addresses to choose from). If the program is written into a ROM, then the user can not modify the program line that gives the starting address of a list. To accommodate this situation there is a type of instruction that looks up the starting address of the list in page zero of RAM. This address is chosen and inserted by the user into the zero-page address. The uP looks that up and proceeds as with any indexed instruction. Here is an example of this type of instruction: A=((Zℓℓ*)+Y). Since an address requires two bytes -- page and line -- two zero-page registers (Zℓℓ and Zℓℓ+1) are used up for stating the first address. This is expressed as Zℓℓ*.

The following program uses this instruction:

```
Y=00
A=((Zℓℓ*)+Y)
JSUB PROCESS.ACCUMULATOR
X=0
LOOP X=hh
BNE /0,J-5
```

This instruction is used in the music program in section 24.

Because the uP first looks up the ("indirect") address on page zero and then steps through ("indexes") the list starting at that address, the name INDIRECT INDEXED has been given to this type of instruction.

(6) PICKING UP SCATTERED ADDRESSES WITH INDEXED INSTRUCTIONS

Sometimes the addresses that have to be looked up in sequence are not contiguous -- such as the addresses of registers in different I/O adapters. For such occasions these scattered addresses may be listed in a single list on page zero, and there is a type of instruction that will scan ("index") through this list and then ("indirectly") look up the required registers. Accordingly, this type of instruction is called INDEXED INDIRECT.

An example of this type of instruction is A=((Zℓℓ+X*))

Here is the way it would be used in a program:

```
Y=00
A=((Zℓℓ+X*))
JSUB PROCESS.ACCUMULATOR
X=1
X=1
LOOP X=X-2h
BNE /0,J-6
```

The first time around, since X=0, the first required address will be looked up in two consecutive zero-page registers (automatically): first the required line number, then the page number. The uP then goes to this address and reads the contents into the Accumulator. The contents are processed in the subroutine PROCESS. THE ACCUMULATOR, and the index X is incremented twice so as to skip over the zero-page register that contains the previous page number. Next the index X is tested to see whether all the registers on the list have been read out. Since each instruction gobbles up two zero-page registers, the test number has to be twice the required number. /2h

(6) STACK INSTRUCTIONS

Stack instructions store and retrieve data in some particular part of memory without requiring an operand, i.e. address.

In the 6502 stack instructions automatically address page 1. In the PIEEE-77 board addressing page 1 locates page 0.

Each next data is tossed on top of the stack, or pulled from top of stack. Such a system is known as LIFO -- Last In, First Out.

The address of each next register is provided automatically by a counter known as stack pointer, pS. The commonly used symbol for storing data in the stack is a down arrow ↓. It is often referred to as PUSH. Thus, A↓ may be read as "push Accumulator on stack". To read stack data into the Accumulator, an up-arrow is used, ↑. A↑ is read as "pop the Accumulator from the Stack".

In the 6502 the only other register that can be pushed or popped is the Flag register, rF.

The stack is also used by subroutine, break instructions and by hardware interrupts. Since these instructions temporarily suspend the orderly operation of the program, the uP automatically stores the program step at which it was interrupted and the contents of Accumulator and Flag register at that time, so that it knows where to return when the interruption has been taken care of.

STEP-BY-STEP DESCRIPTION OF THE OPERATION

POWER TURNED ON

As the power supply voltage rises toward +5 volts there is a circuit on board (POWER-ON) that shorts out the RESET pin of the uP to ground for a moment and then releases it. Release of the RESET pin puts the uP into starting sequence of 5 uS. At the end of that time the uP reads registers FFFC,D and jumps to the program line that the programmer has left there. It then proceeds to execute the program. Note that no start-up buttons of any kind are required. The uP starts working the moment the board is turned on.

Let us observe power-up operation of a uP with the program starting out as follows:

```
FFFC* -- 10 00  J:0010
0012* A9 F0 --  A=F0
0013* 8D 0▮ 6L  (6L0▮)=A  set pins A0-3
0015* AD 0▮ 6E  A=(6E0▮)  read input
```

When power switch is turned on, power-on circuit holds down RESET line for a moment.

CLK CYC	ADRS BUS	DATA BUS	INTERNAL ACTION	EXTERNAL ACTION
1	RESET line is released; five cycles pass for internal start-up of uP.			
6	FFFC	10, line number	saves 10	
7	FFFD	00, page number	saves 00	
8	0010	A9, op-code	saves A9	
9			decodes A9	
	0011	F0, constant	saves F0	
10			A=F0	
	0012	8D, op-code	saves 8D	
11			decodes 8D	
	0013	0▮, line number	saves 0▮	
12	0014	6E, page number	saves 6E	
13	6E0▮	F0		(pp11)=A, i.e. WRITE
14	0015	AD, op-code	saves AD	
15			decodes AD	
	0016	0▮, line number	saves 0▮	
16	0017	6E, page number	saves 6E	
17	6E0▮	input signal	A=(6E0▮)	

etc.

Fig 12.2 Step-by-step power-on start-up of 6502 uP. *Note that start-up is automatic-no switches have to be activated besides turning on power. Note that on cycle 13 the address bus carries not the next sequential address but rather the address of the register being loaded.*

Program: 0020 if =0,J+3

1	0020	op-code F0	save F0
2			decode F0
	0021	offset 03	save 03
3	0022	*ignore*	0022+03
4	0025	op-code	save op-code

Fig 12.3 Step-by-step operation of an if-then-jump instruction. *Note that cycle 3 is used solely for calculating the next program line; while data do appear at this time, they are ignored.*

Figs 12.4 to 12.8 show pictorially how data are routed on the data bus for instruction fetch, read and write output, and read and write memory.

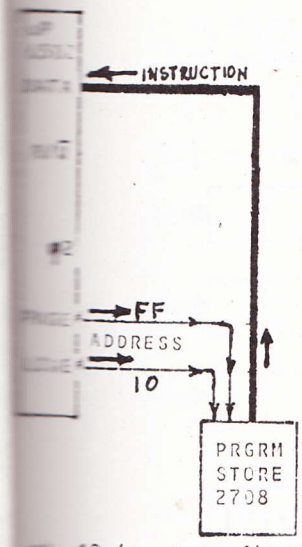


Fig 12.4 Data flow: Fetching an instruction from program store, address 1002.

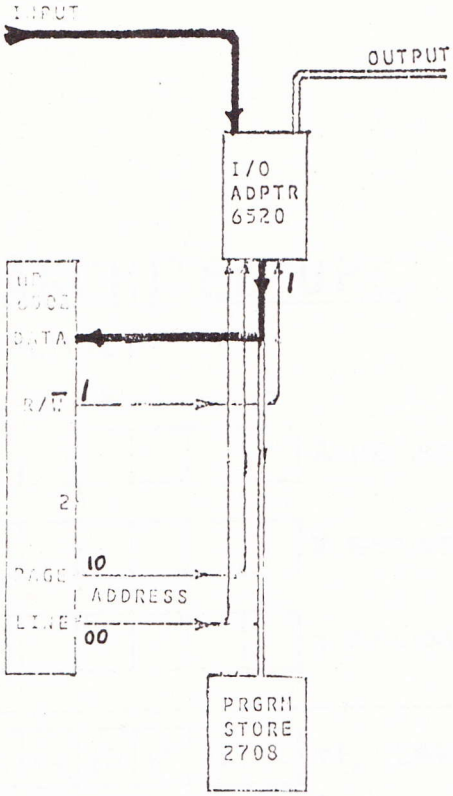


Fig 12.5 Data flow: Reading an input signal trough an I/O adapter chip. The address of the register in the adapter is 1000.

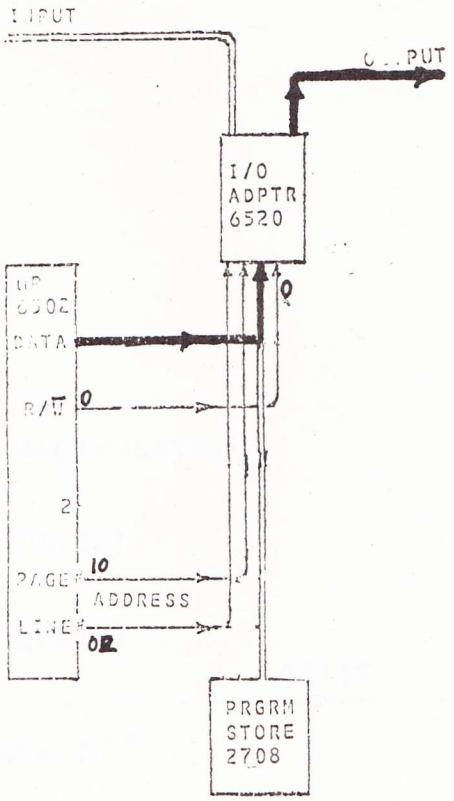


Fig 12.6 Data flow: Sending (writing) an output signal through I/O adapter chip. Note that R/W line is down.

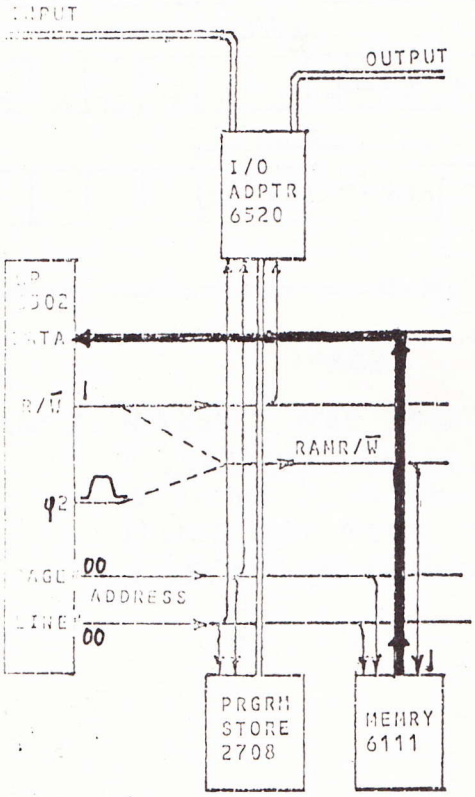


Fig 12.7 Data flow: Reading data from a register in memory. The register address is 0000. Note that a memory register is read during phase 2.

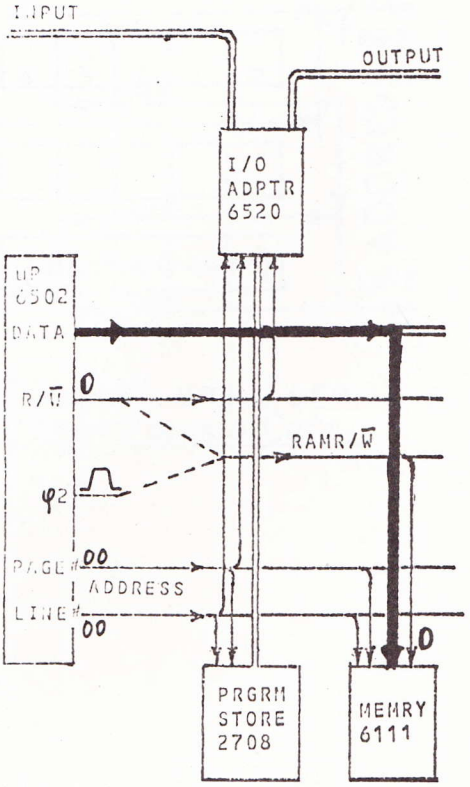
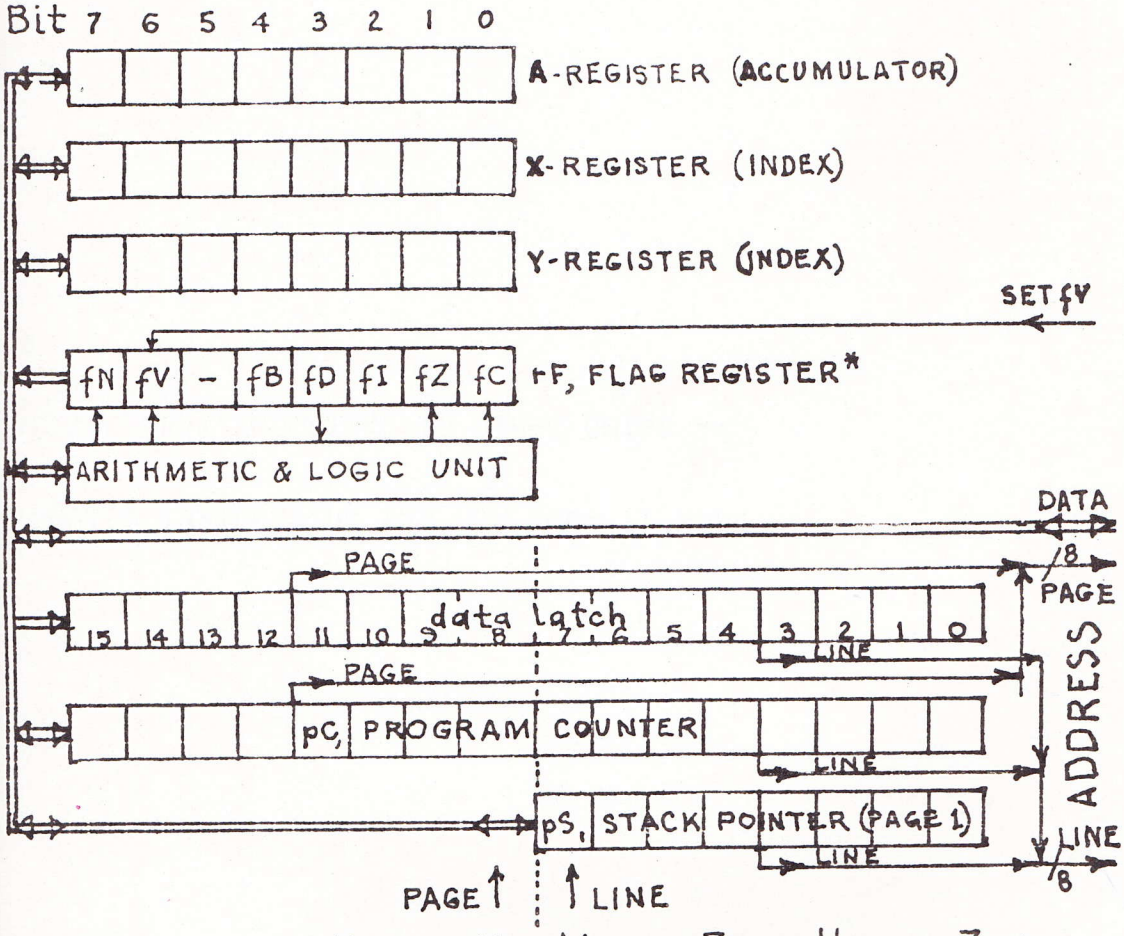


Fig 12.8 Data flow: Writing data into memory address 0000. Writing into memory also takes place during phase 2.

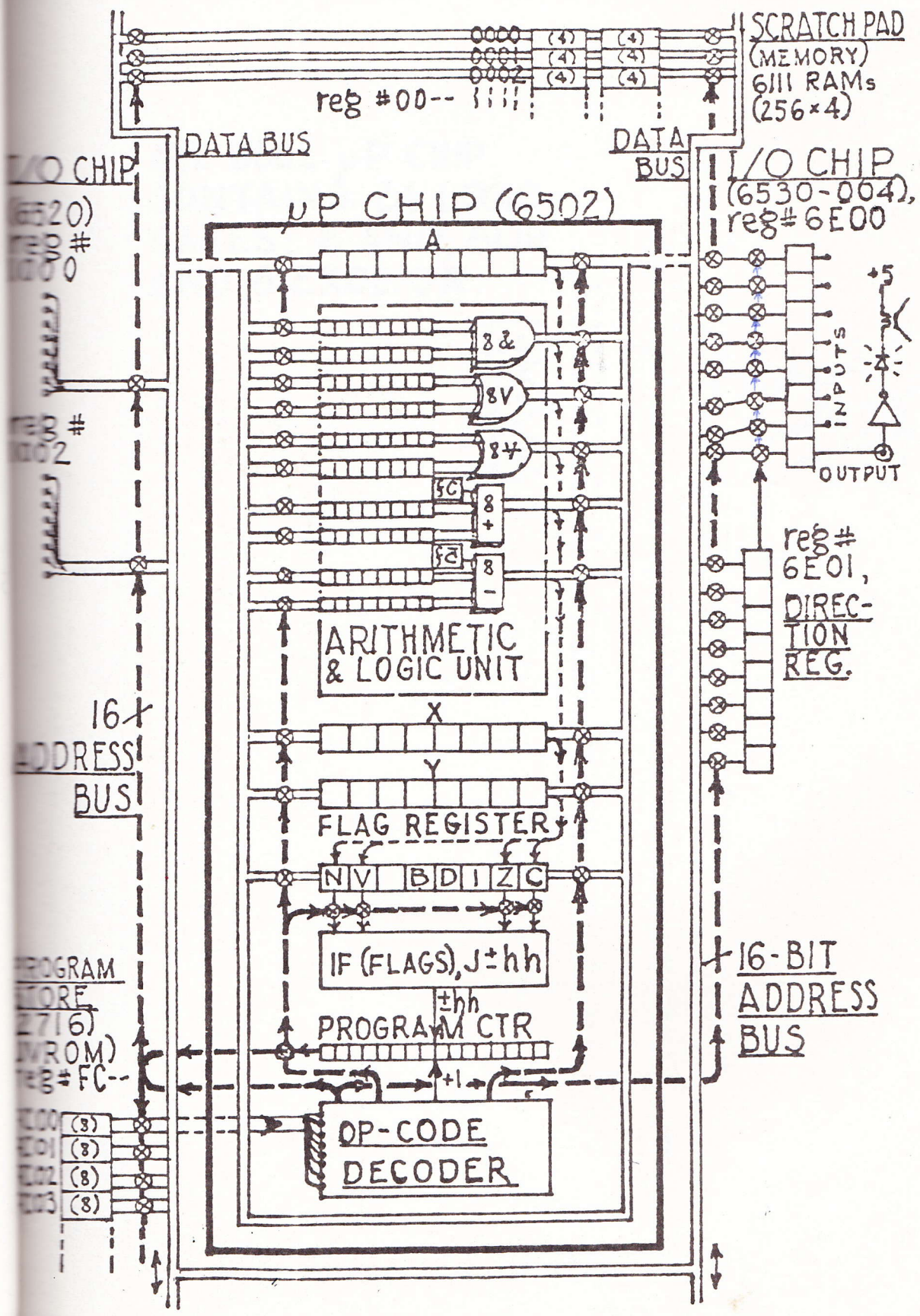
6502 ARCHITECTURE

(SIMPLIFIED)



* FLAGS f: N result was Negative Z result was Zero
V there was oVerflow C there was a Carry
B "Break" subroutine is on
D Decimal mode is on
I Interrupt is disabled

µP STRUCTURE, AND LINKS TO OTHER CHIPS



780110 KVA