易符真经

禅与计算机
硅谷奇遇
第一章 单晶系统的挑战
第二章 P24 CPU 的设计
第三章 P24 组合器兼编译的设计与建造
第四章 Forth 系统(OS) 设计和建造
第五章 P24 仿真器的设计和建造
庖丁解牛

禅与计算机

农历新年前，跌了一跤，把右手腕跌伤了，上了石膏。打计算机用左手，还没有问题，但没有右手，中文就没法写了。杂文写作只好暂缓。

正好老友崔德高自台来访，晤谈甚欢，两人商议一个计划，用计算机来校刊坛经。因此把坛经各种版本，再找出来仔细研究。又上了网络的佛经网站，下载了三版的坛经，敦煌抄本、明藏本及流行本。整理校刊，乐在其中。

禅宗是佛教传入中国后，在中国文化的土壤生根，开出的灿烂花朵。在禅宗以前，佛教是外国宗教，译经五千部，都不得要领。佛教的发展也是大起大落，皇帝信了佛就造寺度僧。或是归了孔教，便毁寺焚经，佛教就消声匿迹。但到了禅宗六祖惠能，一介樵夫，目不识丁，反而把佛陀精义和中国文化结合，创立了中国本土化的禅宗佛教。佛家的思想及禅修的制度，在民间广传深化。从此之后，皇帝老子，再怎样灭佛毁经，也动不了佛教的根本了。

禅宗的广传，坛经是个重要的因素。六祖不识字，不会写书。坛经是他的弟子，将他最重要的几篇演讲，纪录下来，汇集成书。弟子带了坛经出去传教说法，使得禅宗发扬光大，成为中国佛教的主流。并影响到日本，韩国及东南亚各地。这样一本重要的著作，在我们独尊儒术的教育系统中，完全被忽略，是很可惜的事。

六祖揭橥的是完全的人本主义。坛经又称为摩诃般若波罗密经，是六祖经典中最重要的一篇讲章，就是用摩诃般若波罗密经为题的。在梵文中摩诃是「大」，般若是「智慧」，波罗密是「到彼岸」。这个「大智慧到彼岸」是怎么个说法呢？

就这一个「大」字，六祖花了十五分钟来讲。六祖说人的心量广大，犹如虚空、能含万法。就是说人的头脑思想，没有限制范围。这个头脑用得对，就可以顿悟、可以得到智慧。可以解脱今生现世所有的问题痛苦，而可以抵达一个光明世界即是彼岸。所以佛是在每一个人自己的心中。在自身以外找不到真正的佛。现世的困难，今生的问题就是众生。系缚和解脱，只是一念之间。所以迷则佛是众生，悟则众生是佛，至于经书文字，都不是要紧的东西。六祖和他的大师兄神秀争五祖的衣钵传承时，两人呈上的偈语，最能突显六祖的革命精神。

神秀说：

『身如菩提树，心如明镜台，时时勤擦拭，勿使染尘埃。』

惠能说：

『菩提本无树，明镜亦非台，本来无一物，何处惹尘埃。』

好个『本来无一物』，一语扫尽西来佛学，创建东土禅宗。这个『本来无一物』和蓬勃发展的硅谷有什么关系？区区在下，局促在这间门窗破落的硅斋斗室中，能有多少见地，是硅谷中多少聪明人没有看到的？

我在想，假如六祖有幸或是不幸，来到硅谷。看到半导体工业的飞跃进步，看到计算机的日新月异，看到因特网的蓬勃发展，他会怎么想？他会对我们日煎夜熬、孜孜不息的人，有什么样的逆耳忠言？

我想六祖会带一支大棒子来，在我们头上，猛力一击。大喝一声：『本来无一物』，然后哈哈大笑，扬长而去。

计算机的渊源可以往前推很久，如英国 Babbage 的 Analytic Engine，甚至可以推到易经、八卦。但真正用电子技术建造的计算机，还是得由 1942 年的 ENIAC 算起。ENIAC 的设计技术，是相当粗陋的。但 Von Neuman 立即看出它的缺失，而由理论上确立了现代计算机的架构，和发展软件的方法。Von Neuman 的设计，主导了五十年来计算机的发展。我们看到本世纪以来计算机虽不断的进步，但没有超出 Von Neuman 的范围。

计算机的硬件进步是跟着半导体走的。IC 的技术使计算机的容量，每十八个月增加一倍。他的速度每三年增加一倍，价钱也差不多每三年降一半。因为价钱降低，使计算机普及到人手一机的程度。计算机的速度增快，容量增大，是不是它的功能就以超指数的速度增加呢？这可不必。一般来说，虽计算机容量增加，但它每个时候，只能做一件工作。计算机的线路越多，固然能做的工作样数增多，但在做一件工作的时候，其它线路都在空转。所以计算机越大，其中每个线路组件的实际工作效率就减少了。Intel 的 Pentium 越做越大，越做越快。但它最有效的工作是烧电发热。真正做工的效率是很差的！所以 Intel 的芯片叫做「差 86」（X86）。

计算机软件的发展，说起来比硬件更差。硬件的进步是按着 Moore 定律走『硬件容量每十八个月增加一倍』。软件的发展却是跟着 Parkinson 定律走『软件一定会占满所有可用的记忆空间』。不管计算机的记忆容量如何的增加，总是跟不上软件的无限需求。

从软件开发的工具上来看，其进展更是可怜。FORTRAN 是 1957 年问世的。COBOL，LISP 是 1959、ALGOL 是 1960、APL 是 1961、BASIC 是 1965、PASCAL 是 1971、C 和 PROLOG 是 1972。C++到 1984 才出来。JAVA 是 1990。这些计算机语言虽然各有其特性，也各有专擅市场，但基本上是换汤不换药。写作软件的方法，没有什么大的进步。

由 OS 上看，以前计算机市场是 IBM 独占，七家小公司陪衬。各家公司做自己的 OS，各不兼容。等到微电脑起步，微软又挟 DOS 的影响力，以 Windows 横扫群雄。当其锋者无不披弥。Windows 越做越大，若不是 Intel 的 Pentium 容量加大，速度加快，Windows 根本跑不动。Parkinson 定律沾了 Moore 定律的光，才让 Intel 和微软独大。

UNIX 和 ARPANET 都是 1968 年发展出来的。几十年来 Internet 都是局限在政府机关和学校研究室里面使用。直到个人计算机普及到每人桌上都有一台时，Mosaic 被 Netscape 买下，Internet 才突然冒出头来。使用者群不断地扩大，把所有的 PC 都卷进了网络里去，挡都挡不住，这也使得计算机的应用进入了崭新的旅程，但从计算机软硬件的系统技术层面来看，Internet 也没有什么极端突破的进展。

计算机的发展，和当年的佛教传入中国之后的热闹情形十分相似。先是从西域来的僧人，将少量的佛经带入中国，掀起了学佛热潮。于是有玄奘去西天求经，法显由海路前往。翻译了五千卷的佛经，可谓盛况空前了！

现在去 Computer Literacy 看看，整个书店从地面堆到天花板，数不尽的书。林林总总，

涵盖千百种计算机专题。就是皓首穷经，也是读不完的。这还只是现在出版的书。去 Stanford 大学的图书馆，Berkeley 加大的图书馆，有更多的文献资料，怎叫人不废卷兴叹呢？

问题是，计算机真的是那么麻烦？整栋房子的书还讲不清楚吗？其实，计算机是很简单的，再复杂的计算机里面也只是几百条指令，也只能做这几百件简单的事情。麻烦事都是人自己找的。把这几百条指令串连起来，串连得多了，因排列组合的不同、而生出无限的变化。人们再挖空心思，想出不同的工具，来帮助他们做更多、更复杂的排列组合。弄到最后，没有人真正知道计算机在干什么事情。聪明的人，认为发展出高阶的计算机语言，完整的操作系统，就可以帮助人更方便的使用计算机。但却不知道这些计算机语言和操作系统的复杂难度，超过个人理解的能力。所以反而成为人和计算机沟通的障碍。

最聪明的人是 Bill Gates。他发展出 BASIC，DOS，Windows，用很便宜的价钱卖给每一个人。每个人都觉得好用，囫囵吞下他布下的钓饵。然后乖乖地被他笼住半辈子，成为他的奴隶。他叫人按鼠标左键两下，大家都乖乖地按左键两下，任他摆怖。

大家学计算机，都是在学怎样做计算机奴才。计算机叫人怎么做，大家便怎么做。学计算机语言，学 OS，学 Internet 无不如此。

只是计算机原是简单的东西，是要用来帮人做事的，是做人的奴隶的。但是却被有心人利用，作为奴役人的工具。Bill Gates 就是以后新世界的专制统治者的雏型。他掌握了计算机的语言及 OS，就掌握了世界。

我们有没有方法跳出这些聪明人的掌握，叫计算机做我们自己要做的事，而且不受别人统治呢？

六祖在诠释禅宗精义时，最重要的观点、乃是人自己就有佛性。每个人自己就有智慧，能解释万物的。他彻悟之后说：『何期自性本自清净，何期自性本不生灭，何期自性本自俱出，何期自性本不动摇，何期自性能生万法。』又说：『如是诸法，在自性中，如天常清，日月常明。为浮云盖覆，上明下暗。忽遇风吹云散，上下俱明，万象皆现』。

计算机也是这样的。我们要相信计算机是简单的、是给人用的，也是人可以理解掌控的。只是被人加上了一层层复杂的软件，叫你不能自由地来利用发挥计算机的功能。去除这层障碍，我们就可以直接向计算机下达指令，做我们自己要做的事。我们不要学做计算机的奴才，要学怎样做计算机的主人。

六祖又说：『智如日，慧如月，智慧常明。于外着境，被妄念浮云盖覆，自性不得明朗。若遇善知识、闻真正法，自除迷妄。内外明彻，于自性中万法皆现。见性之人，亦复如是。』

谁是这位善知识，能教我们怎样从计算机的枷锁中解脱出来?什么是真正法，能让我们直接指挥计算机做我们心想的事？广东人过年的一个祝语是『心想事成』。有什么好的办法，能叫我们心里想到的事，就能自然做成功？

这位善知识，不在 Standford 大学，也不在 Berkeley 加大。这个真正法，你在 Computer Literacy 也找不到。这位善知识是谁，近在眼前，远在天边。这个真正法，早在三十年前就已经是公开的秘密。

究竟如何，且听下回分解。

正是：

『心地含诸种，普雨悉皆萌，顿悟花情已，菩提果自成。』

(六祖．临终偈语)

硅谷奇遇

武侠小说是中国文化的一部份。半数的中国人，尤其是男的，多少都会一段时间沉迷在武侠小说中。虽然大家都知道、武侠小说中的人物和功夫都是虚构的，但还是兴致勃勃的一本本地读下去，乐此不疲。

大部分武侠小说的主轴都差不多，男女主角皆去深山古寺中拜师学艺。或是经过名师调教，或是不经意地获得奇经秘籍，修练成功。然后下山、闯荡江湖，行侠仗义。以下的情节，或是恩怨相报，或是男女情仇，就看作者的想象能力了。

现代最著名的武侠作家是金庸。在他的经典之作射雕英雄传中，人物众多、情节曲折。但其中心却是围绕着一部『九阴真经』。谁得了这本经，练就了其中一些功夫，就可称霸武林了。为了这本九阴真经，大家拼得你死我活，看得读者们眼花瞭乱。

现在已经都是二十一世纪了，人也已到月球上逛一趟回来了。原子弹、雷射、飞机、无线电、火箭、计算机、和网络等等，大家都也见识过了。九阴真经这种神话式小说故事，骗骗人吧，谁拿来当真呢？

这里，我却要告诉您一个，真真实实『易符真经』的故事。

许多人都有中年危机，我是在三十五岁的时候，遭遇到中年危机。我念的是化学，研究的是分子光谱学。但在台湾做了多年的研究之后发现，我们的基本问题是，需要有一流的仪器才能做一流的研究。但一流的仪器是买不到的。在买来的仪器上所能做的研究，都是早已过时的题目。一流的仪器是要自己做的。

拆开人家的仪器，里面都是一大堆的 IC，一大堆的线路，也看到一大堆的微电脑芯片。我的中年危机就是这个问题。我以后是永远买人家的仪器，永远跟在人家的后面跑呢？还是弄清楚人家是怎么搞出这些东西来的，然后可以去做个更好的？

化学的步调很慢，十年以后再回来还是可以赶得上。但是电机、电子、和计算机的进步太快了，不努力专心追赶就注定要落后的。因此就撇下一切，再度飘洋过海来到硅谷，看看电子与计算机究竟是怎么回事。

长话短说。我进了洛克希德火箭太空公司的研究部。在非破坏性检验室，协助建造高能 X 光的透视系统，检查固体火箭燃料柱中的问题。美国海军用的火箭，小到麻雀火箭大到三叉戟，都是用我们的系统来检查的。

起先、我们的透视系统中的数字影像处理部分是自己做的。我则是负责用 8080 微处理机，来控制搬运火箭弹体的系统。后来我看到商用的数字影像处理系统开发出来，价钱比我们自己做的便宜很多。所以花了很多力气，说服老板采用现成的影像处理机。我也义不容辞的负责写影像处理的程序。系统价格突然减少了很多，海军和公司都很高兴。

有一天。海军通知我们，希望我们系统中不要使用磁盘片，要把整套系统全都放在 ROM 内存中，避免在火箭生产时、可能遇到的污染及干扰问题。大家都傻了眼，这么大的 FORTRAN 程序要缩到几十 K 的 ROM 是不可能的事。

我的建议是使用符式【Forth】语言，其实我也没有用过这种计算机语言，但几年来密切注意微电脑的发展，知道这个语言特别精简，也容易移植到 ROM 中操作。

老板在别无选择之际，勉强同意试用 Forth。我们花了五千美金，从 Forth Inc 买回了一张八吋的磁盘片，老板看了这张磁盘问说：『这张磁盘，真的值得五千块钱吗？』我说没有别的办法，也只好估且一试了。但是一试之下，奇迹发生了。一边看手册，一边试着写程序，控制影像处理机。我在两个星期内，把以前六个月做的 FORTRAN 程序所有的功能，完全转建到新的 Forth 系统上。原来的 FORTRAN 只是个应用的编译程序，有新的应用进来，系统总是要重新整理开发。而 Forth 系统则好像是活的，新的工作很容易编进现存的系统中，应付不同的需求只在举手之间。

虽然我学会了用 Forth 写程序，但是我对 Forth 系统还是不十分了解。 Forth, Inc. 把所有的系统程序都放在那一张磁盘上，我全部印出来看，但就是看不懂，说明书也讲不清楚。这时候听说旧金山弯区有一个符式协会【Forth Interest Group, FIG】 经常开会。打听出来他们每个月开会的时间地点，就去参加了。起先去的时候，他们讲的东西我完全听不懂。去久了，认识一些人，慢慢问慢慢听，才知道他们在推广他们自己开发出来的 figForth，跟我用的 polyForth 是不一样的。

公司里的影像系统是装在 PDP-11 的迷你计算机上。我们还有一套 Nova 迷你计算机，我就想怎么把 Forth 转移到 Nova 计算机上面去。有一次 FIG 开会时，有一个人带了一些 Forth 数据来，有一份是关于 Nova 上的 Forth 系统，我就买了回来研究。翻来覆去读了很久，最后决定用这本 Forth 和 figForth 做为蓝本，在 Nova 上写一个 figForth 的系统。figForth 比 polyForth 简单得多，而且资料完全，还有人可以帮忙回答问题，所以比较容易建立。

公司里的工作因为用了 Forth 变得很容易对付，剩下来的时间就全力放在 Nova 计算机上面，先把 PDP-11 的程序改写为 Nova 的程序，然后再一步一步地除错。在除错的过程中，所有没有看清楚的问题、都慢慢地搞清楚了。等到这 Forth 系统真正动起来，我对这 Forth 系统，也就算是真正的搞通了。一个多月密集工作之后，终于看到 Nova 计算机送出了『OK』讯息，真是喜不自胜。当晚回到家就跟太太说：『我现在已经从应用软件工程师，晋升到系统软件工程师了！』

第一次把计算机系统带动起来，激动的心情是很难用语言来表达的。每一行指令都是自己写的，每一个指令的动作、和他的目的都是自己规定的。这时候的计算机是真属于我了，我的思想和计算机已融合为一，再也没有界限了。这个经验在人生中只有三件事可以相比：一是母亲生下一个孩子，二是基督徒的重生得救，三是禅宗里的顿悟。

在我顿悟之后，晚上都无法入睡。眼前看见的尽是一条条的指令在飞舞，排列成群，井然有序。后来想到，非得把这些思想倒出来，不然头脑就不得清静。所以就把 PDP-11 figForth 摊了开来，按着逻辑顺序，把每个指令的功用、和它在系统里的角色，分章分节，一段一段地解说。 Forth 系统就是这样构筑起来的。就这样完成了 System Guide to figForth 一书 。

我印了五十册，带到 FIG 的月会，起先我还不知道怎么卖，有人看我带了纸箱来，不客气的掀开来，问那是什么东西？一看那是 Forth 的说明书就马上付钱买了，别人看见了也凑向前来，不一会儿，五十本全都卖完了。后来的人只好问说：『你下个月还会回来吧？』

那时 FIG 中流行三个笑话：第一是 Forth 不是个计算机语言，是个宗教，因为搞 Forth 的人都是狂热分子，精神都有些失常。第二是 Forth 是 WOL，是唯写的计算机语言【Write-Only Language】 也就是只能写、不能读的语言，这是从 ROM 对比来的。第三是写 Forth 语言的人都是用筷子吃饭。因为 FIG 开理事会时，领导人 Bill Ragsdal 都是从中国餐馆叫了盒装饭菜来请大家吃。

Forth 的程序简洁而精致，堆栈上的工作数据是不用名字的。前后文又常分散很远，所以不容易读和了解。所以大家戏称他是『唯写语言』。但自从有了 System Guide to figForth 一行一行地揭开 Forth 的神秘面纱之后，Forth 不再是那么地神秘难解了。自此之后，这里面第二个笑话也不再有人提起了。

不久之后，我就在 FIG 见到了发明 Forth 的摩尔先生【Chuck Moore】。他是德州人，个子很高大，头秃了大半。他还喜欢穿着马靴，带着德州牛仔的大帽子，名片上印的公司名子叫做『计算机牛仔【Computer Cowboy】』。他为人很和气，脸上永远是带着笑容，讲话也是从容不迫。大家都围着他提问题，他也都认真考虑才说话，从不轻率提意见。他在 70 年代初去洛杉矶创立了 Forth Inc. 这公司，但那时他已经脱离这公司了，他把赚得的钱在旧金山湾区附近的 Woodside 小镇山上买了一栋房子定居下来。他搬来湾区的主要原因，就是要进行 Forth 芯片的设计。

他认为 Forth 并不是一种计算机语言，而是一个最好的计算机硬件设计。但 Forth 公开了十年，还没有 IC 设计的人来设计 Forth 芯片。他觉得责无旁贷，所以要专心来做真正能执行 Forth 指令的 CPU。过了不久芯片真的做出来了，这便是 NC4000。Forth 芯片的出世，我们搞 Forth 的人都十分兴奋。大家也跟着发展 NC4000 的应用。

摩尔先生自己设计了一块电路板，给大家做 NC4000 的开发工作。他也帮助我们做我们自己的电路板，并提供给我们一套 cmForth 系统，可以在板子上直接用 Forth 语言发展应用程序。这套 cmForth 是摩尔先生钻研 Forth 二十余年的结晶，短小精悍，十分精致，全部的程序只有十页长。我上山去他家请回这套 cmForth 时的感受，就像摩西上西乃山、接受上帝颁发的十诫一样。回来后，我仔细研究，详加批注，发表在『空谷足音【Footsteps In An Empty Valley】』一书中。

NC4000 的所有权后来辗转落到 Harris Semiconductor 公司。Harris 公司据此 开发出 RTX2000 系列的 CPU。不幸的是，Harris 公司营运不利，而将 CMOS 有关的部门全部解散，RTX2000 的计算机也就没有继续发展下去。但美国太空总署通过了 RTX2010 使用在太空计划中的认证。一直到今天(公元 2001 年)，许多太空及卫星计划，还都继续使用 RTX2010 做控制系统的计算机芯片。

cmForth 就像是射雕英雄传中的九阴真经。了解这套系统就可以将 Forth 转到任何计算

机系统中，对计算机的资源做最有效的利用。我有幸得此真传，受益匪浅。以后碰到任何问题，也就都能挥洒自如，无往而不利。

摩尔先生后来又设计了一套 32 位的 CPU，称做 ShBoom。 其所有权后来由 Patriot Scientific 取得，这公司藉此发展出了 PSC1000 号计算机，成了市面上唯一能自由购得的 JAVA 芯片。

1990 年初，我们在硅谷 FIG 讨论 Forth 以后的发展趋势。在 FIG 的十几个年头中，发展了 figForth, F83 和 FPC 等系统，免费给大家使用。我觉得这些系统越做越大，使用的人也就越来越难全盘了解，也就很难有效使用它了。因此提出了易符【eForth】系统的构想，希望能提出一个简单实用的 Forth 系统，让没有 Forth 经验的人也能很快的上手，并能据此发展自己的系统和应用。经过长时间的讨论之后，我们决定采用了 Bill Muench 的模式，发展了 PC 上第一套 8086 eForth。以后，陆续有二十多套 eForth 发展出来，应用在许多不同的 CPU 上面。

eForth 系统的特色，就是将一部计算机的必要功能，提炼出 31 个核心基本指令。对于任何一个 CPU，我们只要重写这 31 个 eForth 基本指令，整个 eForth 系统就可以在这一个 CPU 上活起来。这『e』的系统就是指容易【easy】，也是指嵌入式系统【embedded-system】的意思。

1992 年，MOSIS 计划中的一批人搬到了硅谷成立了 Orbit 公司，专门做实验芯片的代工制造。用他们最小号的芯片做一次只收 1500 美金。我就去找摩尔先生做新的 Forth 芯片，请他设计，我出钱制造。这就是 MuP21 计算机。

MuP21 是二十位的计算机，它的指令集很简单，只有二十五个指令，可以用五个位来界定。这 25 个指令摩尔先生是依硬件设计的方便而决定的，并不考虑其后续软件开发的影响。但我们把这指令集与 eForth 基本指令集相互比较，大部分的指令是相同的。这说明了软件设计与硬件设计，有其共同的需求，这也是我们所说的 Forth 系统，在计算机软硬件上的一贯性。

当今晶圆代工业与 IC 设计工具的发展，已使有本事的人，都可设计自己的单晶系统，开发自己的应用。公元两千年以来 IC 业的最大挑战之一，就是系统芯片【System-On-a-Chip,SOC】，在硅谷及全世界各个科技场所，SOC 也都是最热门的话题。但大家对 SOC 的讨论都还只是在『芯片【Chip】』上打转，却没有看到 SOC 的关键在『系统【System】)，也就是应该用一个计算机系统的角度，来整合数以百万闸的系统芯片。

这样，eForth 就可以大展所长了。

因为 eForth 是一个非常精简的系统，只有大约 6Kbyte 的程序代码，所以很容易被植入 SOC 系统中。这具有 eForth 的系统芯片，事实上就是一部内建的小计算机，只要 CPU 让这小计算机里的 eForth 先动起来，其余再复杂的工作，就容易处理了。在这样的芯片系统中，所有 CPU 所可以到达的周边装置，就都可以加以规划、测试。而整套 SOC 应用系统的程序发展，也都得以经由这 eForth 来进行。

这个观点得到了在台湾许多朋友的认同，并愿意认真地加以实现。于是大家合作成立了易码公司，要把这观念做出实际的芯片来应用。

于是，我们先把 MuP21 的指令集，在某个 Xilinx FPGA【Field Programing Gate Arrays】的发展工具上验证出来，并把 eForth 系统移植上去，这样就发展出了 P16 和 P24 两套系统。P24 系统并得到加州理工学院太空辐射实验室的采用，做为太阳能辐射研究卫星上，计算机装置的核心架构。接着，我们又发展出支持更多指令的 eM24 系统，做为 eForth 的一个更高阶的应用平台。

在易码公司中，郭宏达负责芯片的的设计制造，陈昌江负责 eM 系列芯片的开发，金城负责各种技术的研究规划，锺永源则擘画这一芯片系统技术的经营方针。这样，在经过半年之后，锺永源认为该是以更慎重的态度，来认真推广这 SOC 技术的时候了，于是就成立易符集团，把 SOC 系统技术的开发，与易码的芯片制造分开。他并全力地投入这易符集团的经营运作。

一天，锺永源问陈昌江说，要认真学 Forth 应该如何拜师学艺。我听陈昌江这一提，想到事态严重，不能等闲视之。经过一夜长考，就和陈昌江商定开班授徒，并请锺永源、赖溱君、黎锡壁、洪文昌和林大彰等五位参加。为这次讲习，准备的教材如下：

1. 硅斋禅思
2. 空谷足音
3. CPU 的设计
4. Assembler 与 Compiler 制作
5. OS 的设计制作
6. Simulator 的设计制作

并将此命名为『易符真经』。总结我二十年来研究与使用 Forth 的经验。 正是：

『为学日益，为道日损，损之又损，以致于无。为无为，而无不为。』

（老子．道德经）

第一章 单晶系统的挑战

1-1 摩尔定律

摩尔定律说芯片的密度每十八个月增加一倍，这是在人类历史上从来没有的现像。五十年来这个增加的趋势一直没有改变。当芯片中的闸数不断地增加时，量变引起质变。我们如果以十年为一期，硅芯片的发展主导了计算机工业进步：

1950 晶体管时代， SSI 芯片密度 1-10。
1960 集成线路时代， MSI 芯片密度 10-1K。
1970 微电脑时代， 芯片密度 1K-100K。
1980 个人计算机时代， 芯片密度 100K-1M。
1990 网络时代， 芯片密度 1M-10M。
2000 单晶系统时代， 芯片密度 10M-100M。

在 50 年代，晶体管的广泛使用取代了真空管，而使第二代计算机取代了真空管的第一代计算机。在 60 年代，中密度集成线路 MSI 的使用产生了迷你计算机，而使计算机侵入了工厂和实验室。在 70 年代，高密度集成线路 LSI 的使用产生了微电脑和微处理器，植入了各式各样的消费性产品中，如同计算器，电子表，数字游乐器等。在 80 年代，微电脑进步成为个人计算机，每个人的办工桌上都有一台，完全改变了人们的工作和生活习惯。在 90 年代，个人计算机能够容纳图标接口和网络时，就引爆了国际网络。

在这个新世纪的开始，什么产品会是以后十年的主流？我不会准确预测未来的变化，但是我可以确信，这些产品必定是用单晶系统为核心做成的。

单晶系统【System-On-a-Chip, SOC】的定义不是十分明确的，每个人都有他自巳的定义和说法。在集成电路技术一开始的时侯，大家就是在设法将整个系统做在一个芯片上。早年的放大器、振荡器、正反器、逻辑闸等，都是合理的单晶系统。后来我们见到更复杂的计数器、缓存器、算术逻辑分片器【ALU Bit-Slice】。以后出现了微电脑和微处理器。微电脑严格的说并不是单晶系统，因为它必需和内存和许多外围装置配合才能使用。但是微处理器就是很完整的单晶系统了。例如 8051 整合了 CPU、RAM、ROM、及许多常用的外围装置，一个线路板上，往往只有它一颗大芯片。所有工作都由它完成，是一个很好的单芯片系统的定义。

早年像 8051 的微处理机有很多只读型记忆 ROM，读写型的记忆 RAM 是很少的，只是用来存一些变量数据，以供给执行程序使用。以后的 SOC 单晶系统需要很多的 RAM 记忆，因此可以随时加载新程序，更方便地解决实时面对的问题。这样的系统是智能型的系统，有学习成长的功能，而不是被限制在只读记忆中预先植入的程序。廿一世纪智能型的单芯片系统和愚笨型的单芯片系统的差别就是在其中读写型记忆的数量。

1-2 智愚之界

智慧是什么？由字面的定义，智能是思考【Thinking】、学习【Learning】、和认知

【Understanding】的能力。思考是一个抽象化的过程，将许多简单的对象及观念组合起来变成一个新的观念。学习是将抽象的对象和观念记住的行动。抽象化的过程不断进展到一层一层更高的境界，垒积成为智识，技术和科学。认知是反向的动作，是将抽象的观念一层一层地展开成低层的对象及观念，直到我们熟悉而能使用的层次。

从芯片的设计和制造工作来看，单芯片系统是逻辑线路抽象化最高的层次。但对实际使用这些系统的人来看，单芯片系统还是在很原始很低的层次。两者在学习和认知间的鸿沟，是靠着软件来超越的。

我们人类借着在大脑里非常复杂的神经系统，能够做到全方位【Holistic】的思考和学习的工作。人的专长是做抽象化的工作，这是人脑智慧【Human Intelligence】。相对的来看，计算机的长处就是认知，因为它有极其可靠的记忆和非常快速的逻辑运算速度。所以，计算机可以说是做认知工作最好的工具，它将层层堆积高层次的观念展开并且快速执行，这是机械智能【Machine Intelligence】。人脑智慧和机械智慧是相辅相成的。人脑智慧和机械智慧最理想的结合，就是用人脑来做思考、创新、和设计的工作，而让计算机做认知、展开、和执行的工作。

而人工智能【Artificial Intelligence】的理想，乃是要计算机同时做思考和认知的工作，这是很困难的事，因为我们自已还不完全了解我们的大脑如何工作。如果我们能分辨人脑智能和机械智能，就可以在现阶段找到使用计算机最好的办法。

我们这一代的人最幸福了，因为我们有机会整合数百万闸的线路到单芯片上去。这是前人梦昧以求但又是不能想象的。我们应该怎样做才能把我们理想的设计，实现在单芯片系统上呢？

现在我们正是站在这条分割智愚的界在线。微电脑和微处理机正在由愚笨型转变成为智能型。但是，若是仅仅增加读写内存的容量并不自然地产生智慧。机械智能的一个重要的因素是计算机里的操作系统。只是一般的操作系统都是庞然大物，无法植入容量有限的单芯片系统里。符式【Forth】是一个短小精悍的实时操作系统，最适合植入单芯片系统，作为支持智能型微电脑发展及使用的平台。

将符式植入单芯片系统，还带进来许多实际的好处，帮助单芯片系统的规划、设计、测试、侦错、以及后阶段的软件开发。

单芯片系统的测试，对芯片制造厂是非常头痛的大问题。现在使用的技术，如 JTAG 扫瞄及植入自测 BIST 设备，对复杂的单芯片系统要增加大量的额外线路、测试的时间、和复盖率，也都是很大的负荷。如果我们先将符式植入单芯片，只要 CPU 核心和少量的内存没有问题，符式系统一起动，就可以很方便测验所有和 CPU 联结的装置。硬件测试的问题化解成为软件的问题，就容易解决了。有了好的操作系统，CPU 就是一个装备完全的芯片测试器，不必再外加昂贵复杂的测试设备。

有了一个符式操作系统在单芯片上，硬件侦错和软件侦错的工作都变得简单了。符式操作系统包含了一百多个模块或指令，这些指令可以直接从键盘输入而实时执行，做测试的工

作。使用者还可以定义新的指令，做更复杂的工作。这是符式中的直译器【Interpreter】和编译器【Compiler】，利用这两个工具，使用者可以很方便地在交谈式的环境中，测试硬件和软件，很快的找出问题的征结并修正错误的地方。

有了一个符式操作系统在单芯片上，应用的软件程序也可以直接在单芯片系统上开发。应用软件可以分解成几个层次：在最底层的是输入输出装置的驱动程序【Device Drivers】，上一层是这些驱动程序的组合和时序的安排。这些模块建造完成后，最高层的应用程序只是一个无穷回路，依序呼叫这些模块。只要单芯片系统上有足够的读写记忆，软件开发的工作就可以直接在单芯片系统上进行，不必借重外挂的仿真器【Simulator/Emulator】。直接在单芯片系统上进行软件开发的工作还有一个好处，就是可以有准确的时序。这对开发实时的应用程序，是十分便利的。

1-3 发展单晶的契机

单芯片系统是芯片工业未来的趋势。单芯片系统是令人兴奋的技术。但是这个趋势这个技术，对我们个人，一些手无寸铁负债累累的工程师，有什么实质的意义呢？

设计芯片谈何容易！制造芯片更是需要雄厚的财力和广泛的资源，是庞大的企业公司如IBM、Intel、AMD 的范畴。但是近十年来，芯片工业中产生了极大的，基础性的变化，给我们个人带来无限的新机会。

芯片工业由传统的垂直整合的模式转变成水平分工的模式，是这十年中最大的变化。垂直整合的公司，从市场分析，设计，布线，模拟，光罩，晶元处理，到封装测试，每样都得自巳来做，所费不赀。现在电路自动设计【Electronic Design Automation, EDA】已形成一个独立的工业，供应各种设计和模拟的工具。高阶的电路设计语言如 Verilog 和 VHDL 更使芯片设计从芯片制程的枷锁中解放出来。芯片代工业的兴起，也带动了一大群的无厂芯片公司【Fabless IC Manufacturers】欣欣向荣的发展。芯片工业不再是大公司垄断的局面。

第三个重要的新发展是可规划芯片【Programmable Logic Devices】的成熟。可规划芯片有好些类型，如可规划数组【Programmable Array Logic, PAL】、可规划逻辑【Programmable Logic Device, PLD】、和现场规划闸阵【Field Programmable Gate Array, FPGA】等。目前的现场规划闸阵中，闸数已达到几百万闸，足够做许多单芯片系统的设计和仿真的工作了。

由这一系列的发展来看，单芯片系统的设计工作，是已经全盘平民化的了。个人和大公司有完全相同的机会来追求各自的理想和市场。如果你有一个好的设计，在一台个人计算机上，你就可以用 Verilog 或 VHDL 来描述你的线路。现成的仿真器可以帮助你仿真及验证你的线路。线路设计在个人计算机上立即可以进行综合【Synthesis】的工作，产生的网表【Net List】可以烧录到现场规划闸阵中，进行实际的测试和验证。一旦在现场规划闸阵上能成功地证明电路设计的正确性，就可以将网表送给芯片代工厂，制造你自巳的单芯片。

这个单芯片系统的发展流程，在目前可能还有一些枝节的问题，不是一蹴可成的。但这流程是一定可以做到的，这些枝节的问题，都是可以用有限的金钱和时间来解决的。

单芯片硬件的设计制造，不再是困难的瓶颈了。但是单芯片只是执行软件程序的平台。

软件和硬件紧密无隙的结合，方才构成了完整的单芯片系统。要紧密无隙地整合硬件和软件，必需要有一个有效且能植入单芯片的操作系统。符式操作系统，是最理想的选择。

1-4 符式语言

符式是查尔斯.摩尔【Charles Moore】先生在 1960 后期发明的计算机语言和操作系统。他有感于当时人机接口的复杂性，超出了个人掌控能力。因此他致力简化人机接口，剔除了所有不必要的工具和语法架构，创造出一套简单而完整的语言系统。经由这个语言系统，使用者可以直接向计算机下达指令，执行他需要做的工作。程序设计师也可以很方便地编写高阶程序，解决应用问题。

1-4-1 语言和文字

符式是一个很特殊的计算机语言，它有点像英文，但是没有英文复杂的文法。它的基本单元是一组指令，类似英文字。一序列的指令可以顺序输入计算机，但指令之间必须用空格键分开。计算机接受了这一序列的指令后，就依顺序一个指令一个指令连续执行。执行完毕，计算机就再等你输入下一序列指令。因此，符式的指令，我们习惯上也称之为字【Word】。

符式的指令集是可以延伸的，即是使用者可以自由地加入新指令。一序列的现有指令，可以定出一个新的名字，而建造成为一个新指令。这个新指令定义完成后，就能和现有的指令一样，可以随时输入计算机去执行，或是用来建造新的指令。这个指令集可以延伸的机构，是和自然语言相同的。人们经常创造新字，来更有效地代表新生事物和新观念。字 Words 是语言的基础，不允许创这新字的语言，是无法有效地为人类使用的。

用符式解决问题，就是建造新指令来取代现有指令的序列。这是和自然语言一样的。新的观念用新的字来表达，但是新的观念必须要用现有的观念和事物组合起来才能说明清楚。新发明的字，就把现有的事物和观念抽象化，使得以后的思考和理解，提升到一个较高的层次。抽象化是智慧和思想的重要表征。所以符式是用计算机仿真大脑智能最有效的工具。

以下我们将完整，详细，且严格地说明符式语言的结构。

1-4-2 符式的指令

符式语言是一组指令的集合。每个指令有它独特的名字，也有它在计算机上执行的程序。指令可以单独输入计算机，个别执行。也可以集合成为一知个表列输入，计算机即能顺序执行表列内每个指令。在表列里，指令之间必须以空格键分开。

符式的指令有两种，基本指令和延伸指令。基本指令是符式计算机的机器码。如果负载系统的计算机不是符式计算机，而是符式的虚拟机时，基本指令就必须用负载计算机的机器码组合而成。延伸指令内含一个基本指令和其它延伸指令的表列。执行一个延伸指令，就是顺序执行它内含表列里每个指令。

符式语言有许多不同的版本，各种版本中基本指令和延伸指令的选择和分界都不相同。易符是最简单的一种符式语言。我们以下的讨论，就只专注于易符系统。

在详细讨论基本指令和延伸指令前，我们必须先说明符式使用的堆栈。符式语言使用两个堆栈:返回堆栈和参数堆栈。堆栈是一种无地址，后入先出的内存。它像是一迭盘子，存

新盘子时是堆在最上面，取用时取出的也是最上面最后存上去的那只盘子。

返回堆栈是给子程序用的。呼叫子程序时将返回的地址堆上返回堆栈。在执行返回 Ret 指令时，就将返回堆栈最上面的地址取回，跳到这个地址就可以去继续执行那个子程序以后的指令。一般计算机都有子程序呼叫和返回的指令，也都有返回堆栈的机置。绝大多数的程序设计师，都熟悉返回堆栈和它的使用法。

参数堆栈是给符式指令间传递参数用的。子程序是非常重要的软件构造，是程序模块化的主要工具。将参数传递给子程序，使子程序的功能和使用范围更是大幅增加。传统的计算机语言用参数表列的方式向子程序输入并输出参数，因而必须用十分复杂的编译器来处理原程序。编译器中最困难的工作就是如何将子程序需要的参数送进子程序，并将子程序产生的结果交还给主程序。

在执行一个符式指令前，它所需要的参数必须按照它取用的顺序放上参数堆栈。指令执行完毕，它输出的结果也都堆上了参数堆栈，给后续的指令使用。因此，符式的算术和逻辑指令，都是使用后算符【Postfix, Reverse Polish Notation, RPN】的方式，处理数值数据。例如做 2+3 的工作时，必须输入这样的指令:

2 3 +

在叙述一个指令的功能时，我们就必须准确地标示出这个指令的输入及输出参数，和它们在参数堆栈上正确的顺序。在写符式程序定义新指令时，注解文字都是圈在圆括号里的，所以在这里说明参数堆栈的使用顺序，也放在圆括号里。输入的参数放在双－号的左边，输出的参数放在双－号的右边。最右边的参数在堆栈最上面。

1-4-3 易符的基本指令集

以下是易符中的基本指令: .

名称 堆栈参数 功能

AND (n1 n2 -- n3) 逻辑与

OR (n1 n2 -- n3) 逻辑或

XOR (n1 n2 -- n3) 逻辑亘斥或

COM (n1 -- n2) 逻辑反

UM+ (n1 n2 -- n1+n2 c) n1 加 n2，保留溢位 c

DUP (n -- n n) 复制 n

DROP (n --) 弃去 n

SWAP (n1 n2 -- n2 n1) 交换 n1 及 n2

OVER (n1 n2 -- n1 n2 n1) 复制 n1

>R (n --) 将 n 除去，堆上返回堆栈

R> (-- n) 取回返回堆栈上的参数

R@ (-- n) 复制返回堆栈上的参数

@ (a -- n) 读取记忆地址 a 的内容 n

! (n a --) 将 n 存入地址 a

C@ (a -- b) 读取记忆地址 a 的八位内容 b

C! (b a --) 将八并元值 b 存入地址 a

doLIT (-- n) 将此指令后的数值放上参数堆栈

BRANCH (--) 跳到此指令后的地址

ZBRANCH(n --) 若 n 不是 0，跳到此指令后的地址

0< (n -- 0|-1) 若 n<0，改之为-1；否则改之为 0

doNEXT (--) 若返回堆栈上参数是 0，结束回路

否则将其减 1，跳到此指令后地址

EXECUTE (a --) 执行在地址 a 的指令

EXIT (a --) 结束现在执行的指令，跳到返回堆最上面的地址

这样的一个基本指令集，看起来小得不够用。但是事实证明，将这个指令集延伸，可以建造整个符式操作系统，也可以解决任何困难的问题。

1-4-4 易符的延伸指令

易符系统中大部份的指令都是延伸指令。延伸指令里面都是一个表列，表列里是一连串的子程序呼叫指令 CALL，呼叫基本指令和其它延伸指令。表列最后都是一个返回指令 RET 或 EXIT。所有的指令都存在内存内，可以随时呼叫执行。指令储存的记忆区，称为字典。

目前的易符系统里有大约 180 个指令。这些指令我们以后讨论易符的系统时会详细说明。在这个指令集中，大部份的指令都是为了建造符式的操作系统而设计的。一般写应用程序的指令，大概是 70 个。学习使用符式语言，最要紧的事就是要充分掌握这 70 个常用指令。后面我们有一章符式的学习课程，会用许多例题来说明如何用这些指令来写程序，解决应用问题。

以下是最常用的易符延伸指令：

+ (n1 n2 -- n3) n3=n1+n2。

- (n1 n2 -- n3) n3=n1-n2。

* (n1 n2 -- n3) n3=n1*n2。

/ (n1 n2 -- 商) n1/n2 保留商。

MOD (n1 n2 -- 余数) n1/n2 保留余数。

*/ (n1 n2 n3 -- 商) n1*n2 / n3，保留商。 比例计算。

MAX (n1 n2 -- n3) 保留 n1 和 n2 间的较大值。

MIN (n1 n2 -- n3) 保留 n1 和 n2 间的较小值。

ABS (n1 -- n2) 绝对值。

NEGATE (n1 -- n2) 负值。

< (n1 n2 -- n3) 若 n1

= (n1 n2 -- n3) 若 n1=n2，则 n3=-1，否则 n3=0。

EMIT (c --) 将 ASCII 码 c 送到终端机上显示。

KEY (-- c) 等使用者在终端机上按键，并将键入的 ASCII 码放上堆栈。

HERE (-- a) 词典边界的备用记忆地址。

PAD (-- a) 字符串暂存区的地址。

CMOVE (a1 a2 n --) 从 a1 搬 n 个数值到 a2 区。

FILL (a n1 2 --) 将数值 n2 填入 a 起的 n1 字码区。

. (n --) 将 n 转换成字符串显示在终端机上。

U. (n --) 将正数 n 显示在终端机上。

.R (n1 n2 --) 将 n1 显示在 n2 字段中，靠右对齐。

U.R (n1 n2 --) 将正数 n1 显示在 n2 字段中，靠右对齐。

? (a --) 取出地址 a 中数值，显示在终端机上。

SPACE (--) 输出一个空格。

SPACES (n --) 输出 n 个空格。

CR (--) 换行。

CHARS (n c --) 输出 n 个 ASCII 码 c 所对应的字符。

TYPE (a n --) 从地址 a 取 n 个 ASCII 码字符输出。

, (n --) 逗点指令，将 n 编入字典顶端备用的内存中。

ALLOT (n --) 将字典顶端留下 n 个字码的备用空间。

; (--) 分号，结束一个新指令。

DUMP (a n --) 将内存由地址 a 开始的 n 个字码显示到终端机上。

.S (--) 显示堆栈上所有的数据到终端机上。

WORDS (--) 显示字典中所有指令名称到终端机上。

OK (--) 解读由 READ 下载存在 PAD 的文字。

SEND (a n --) 将从地址 a 开始长度为 u 的数据，上传到主计算机中储存。

: (--) 冒号，定义一个延伸指令。用下一字符串做为新指令的名称。

冒号指令：是符式里最重要的指令。它使我们能够自由地定义新的延伸指令。它的使用方法很简单：

： 新指令名称 现有指令表列 ；

新指令名称和现有指令表列中所有的指令间都必须用空格键或换行键分开。指令间都必须用空格键或换行键分开，是符式最基本也是最简单的第一个文法规则。

1-4-5 分支结构和回路结构

符式是结构化的计算机语言。它允许我们在延伸指令里建造分支结构和回路结构，使得在执行这些指令时，程序可以选择不同的路径，或是重复一段相同的路径。所有的结构都是只有一个进口和一个出口，因此结构可以连续排列，也可以重复套装。但是结构不能交叉重迭。

以下就是易符系统中提供的结构指令集和它们能够建造的结构。这些指令只能用在延伸指令里建造规定的结构。结构可以连续排列，也可以重复套装。但是不能交叉重迭就是符式的第二个文法规则。

(f --) IF ... THEN

若 f≠0，IF 即执行后续的指令。否则跳到 THEN 执行以后的指令。

(f --) IF ... ELSE ... THEN

若 f≠0，IF 即执行后续的指令到 ELSE 为止，然后执行 THEN 以后的指令。若 f=0，IF 即忽略后续的指令跳到 ELSE，执行 ELSE 以后的指令。

BEGIN ... AGAIN

重复执行 BEGIN 及 AGAIN 中间的指令。

BEGIN ... (f --) UNTIL

执行 BEGIN 及 UNTIL 中间的指令，UNTIL 检验堆栈上的数值 f。若 f=0，则跳到 BEGIN 重复回路。若 f≠0，则跳出回路，继续执行 UNTIL 后的指令。

BEGIN ... (f --) WHILE ... REPAT

执行 BEGIN 及 WHILE 凡中的指令，WHILE 检验堆栈上的数值 f。若 f≠0，即执行 WHILE 到 REPEAT 中间的指令，然后跳回 BEGIN，继续此回路。若 f=0，WHILE 即跳出回路，接着执行 REPEAT 以后的指令。

(n --) FOR ... NEXT

FOR 将回路指针 n 从数据堆栈取出，放到返回堆栈上，接着进入回路，执行 FOR 与 NEXT 间的指令。NEXT 检验回路指标，若指标是 0，就跳出回路，执行 NEXT 后的指令。若指标不是 0，就将其减 1，然后跳回 FOR 处重复执行此回路中的指令。

1-4-6 字符串指令

许多符式指令要处理文字字符串。文字字符串是紧跟在指令后面的。大部份时侯，字符串是以空格键为分界的。但有些指令用特殊的字符为分界键。这类指令常用的是：

VARIABLE (--) 用下一字符串建造变量指令。执行该指令时将其地址放上堆栈。

CONSTANT (n --) 用下一字符串建造常数指令。执行该指令时，会将 n 放上堆栈。

CREATE (--) 用下一字符串建造矩阵指令，矩阵大小用 ALLOT 来界定。

' (-- a) 单引号指令，寻找下一字符串所代表的指令，找到后将其执行码栏地址留在堆栈上。

." (--) 在冒号指令里编入一段文字。执行时将文字显示在终端机上。字符串以”号结束。

((--) 略过括号内的文字，在原始程序中，作为批注之用。字符串以)号结束。

\ (--) 略过所有已输入的文字序列。

FORGET (--) 搜寻下一个的指令，将该指令及其后所有指令清除。

SEE (--) 反编码指令。用来检查已定义指令的内容。

READ (--) 从终端机直接下载档案文字。当接收到 cntl Z 时结束下载。

VARIABLE ，CONSTANT， CREATE 和冒号指令：都在字典里加入新指令。新指令

的名称就是紧跟着指令的字符串。在执行时，变量指令和矩阵指令都是把变量或矩阵的起始地址放上堆栈。常数指令则是将常数值放上堆栈。

文字字符串必须紧跟在字符串指令的后面，这是符式的第三个也是最后的文法规则。

1-5 易符系统

易符系统 eForth 是 1990 年在硅谷符式协会【Silicon Valley Forth Interest Group】发展出来专为植入型微处理机使用的符式系统，Bill Muech 提供了基本的设计。它包含了一组 31 个原始指令，和 190 个延伸指令，可以很容易地植入任何微电脑或微处理机中。十多年来，易符被移植到三十多种不同的计算机系统上。

同时在 1990 初年，我和摩尔先生合作发展 MuP21 型微电脑。摩尔先生基于硬件设计的考虑，规划了 26 个机器码指令。这套机器码，和 Bill Muech 纯粹由软件需求定出的 31 个原始指令非常相似。现在的易符系统，就是 eForth 和 MuP21 的综合，专门为单芯片设计规划的 CPU 核心和完整的语言及操作系统。

易符系统的硬件设计，有一系列的计算机核心，包括了 16 位 P16、24 位的 P24、和 32 位的 P32。它们都能执行 MuP21 的指令集，可以执行兼容的程序。它们的操作系统是兼容的，不同的是它们的数值数据，有从 16 到 32 位不同的宽度。这个系列的硬件和各自的易符操作系统，都在 Xilinx 的 VCX1000E 现场规划闸阵上验证成功的。

就硬件观点来看，符式是一个虚拟主机【Virtual Machine】。它有一个返回堆栈【Return Stack】储存子程序的返回地址，和另一个数值堆栈【Data Stack】存放子程序间交换的参数。所有的算术及逻辑运算都是在数值堆栈上完成，因而避免了使用大量缓存器的问题。使用数值堆栈最大的好处就是呼叫子程序时不必提供参数表【Parameter List】，只要提供子程序的地址就够了。

此易符虚拟机，其实只需要有下列一组原始指令【Primitive Instructions】集：

算术指令 + - * / MOD

逻辑指令 AND OR XOR COM

记忆指令 @ ! C@ C!

堆栈指令 DUP DROP SWAP OVER ROT

结构指令 IF ELSE THEN BEGIN UNTIL WHILE REPEAT

DO LOOP FOR NEXT

定义指令 : ; CONSTANT VARIABLE

这些原始指令是用承载符式虚拟机的计算机机器码组合成的。但这原始指令集不够建造符式系统，更不足以建造使用者需要的应用程序。符式因此有延伸指令【Extension Instructions】的设计。延伸指令是一个包含原始指令和其它延伸指令的表列【List】。执行一个延伸指令就是顺序执行表列中每一个指令。定义延伸指令的方法是：

： ＜延伸指令名称＞ ＜指令表列＞ ；

新造的延伸指令又可以用来建造更高层的延伸指令。这样层层延伸，就可以解决任何可

以用计算机解决的问题。

符式和人工智能常用的 Lisp 计算机语言很类似，都是用表列来解决问题。但是符式的结构和语法远比 Lisp 简单，执行也更方便快速，所以适合植入单芯片系统。

1-6 易符真经

对中国人来说，经是一个很严肃的字，在佛教的文献中，只有佛祖说的话才能称为经。中国人写的书，只有六祖的坛经被尊为经。六祖本是一介樵夫，目不识丁，但以

『菩提本无树，明镜亦非台，本来无一物，何处染尘埃。』

一偈得到五祖的激赏，和禅宗的衣砵传承。六祖整合了佛家精义和儒家的传统，使得禅宗成为中国佛教的主流，千年不堕。六祖的弟子收集了他几篇最重要的讲章，编为坛经，流布中国及东亚。

六祖及禅宗的基本观点是人人都有佛性，一悟便至佛地。涅盘成佛，只在一念之间。外在的教条，仪式和经书文字，都不是要紧的事。从初祖达摩到五祖弘忍，都是口传心授，不立文字。六祖看到佛教和中国都已经为禅宗准备成熟，才让坛经成书，嘱咐弟子们秉书传教，使得南宗顿教，在东亚大陆遍地开花。

符式因其语法简洁，系统精要短小，大家都喻之为计算机语言中的禅宗。也因为它简洁精要，程序往往艰深难懂。早年符式的传播，主要也是靠口传心授，一个人传给一个人，慢慢地流传出来的。符式的程序艰深，甚至写程序的人，过一阵子也读不懂自巳写的程序。圈内人戏称符式是唯写语言，只能写不能读的。后来符式协会【Forth Interest Group】成立，我拿了它出版的 figForth 程序，一行一行地详加注释，证明了符式程序是可以理解，能写能读的。

发明符式的摩尔先生，很早就看到符式不仅是计算机语言，也不只是操作系统，而且是一种最精简的 CPU 设计。他等了十年，没见到搞计算机设计的人利用这个架构来设计计算机。他认为计算机设计也不是什么大了不起的事业，就自己卷起袖子来做。1984 年做出 16 位的 NC4000 微电脑，现在许多太空总署的计昼，还是在用这一型计算机。1988 年他又做成功 32 位的 ShBoom 计算机，现在给 Patroit Scientific 公司当做 JAVA 计算机在卖 。

摩尔先生对 CPU 设计的热心执着，对搞符式的人也有很大的启发，就是设计计算机不是难事。许多人也跟着他走上这条路。我也在 1990 件开始和他合作开发 MuP21 计算机。MuP21 是一个非常有趣的设计，核心是一个 20 位的 CPU，只有 25 个指令，所以每个指令只占 5 位。十分简单，但也十分有效。摩尔先生连全套芯片设计，布线及仿真的 CAD 软件工具都是一个人自巳做的，令人叹为观止。但是这套工具并未发展成熟。MuP21 虽然做了出来。良率不高。无法进入量产。

为了 MUP21。摩尔先生特别写了一套编码器 Assembler 及相关的侦错测试软件。还有驱动 MuP21 中视频系统的程序。我也将易符操作系统搬了上去。这样就形成了完整的符式系统。硬件和软件全都是我们自巳开发出来的。走过这段崎岖的道路。我也学到了计算机的全貌。以后几年中。就一直试着用各种 CAD 设计工具重写 MuP21 的核心。重复检验我对

计算机系统的认识和掌控。

当 FPGA 的闸数够多的时侯。我便开始将 MuP21 的 CPU 核心转上去。实际用硬件来证明这个 CPU 和易符操作系统是正确的。当时 Xilinx 的 FPGA 还很小。但我已能够将一个 16 位 P16 计算机塞进 XC4005XL 号 FPGA 里去。许多搞符式的朋友见到了。都觉这是一个不可多得的机会。可以在将来的单芯片系统的市场上。争得一席之地。这样才成立了易符智慧科技公司。来发展及推广以符式及易符为基础的单芯片系统。

多年来。符式和易符都是公开流传的软件系统。易符公司也不认为它可以独占持续发展出来的技术。而且单芯片系统的市场非常广大。需要很多公司和个人投入合作奋斗。才能切入市场。因此必须公开核心技术。欢迎合作伙伴来壮大声势。开拓领域。另外公司同人也亟需训练教材。所以我就将全部易符系统整理出来。包括了以下的题目:

计算机 CPU 的设计和制造

编码器 Assembler 的设计和写作

直译器 Interpreter 和编译器的写作

操作系统 Operating System 的设计和写作

仿真器 Simulator 的设计和写作

所有的讨论。都是基于完整的原程序。除了计算机 CPU 设计是用 VHDL 写成。必需在 Xilinx 的 Foundation 2。1i 平台上综合 Synthesis 外。其它的程序都用 PC 上的 Win32Forth 符式语言撰写的。可以在 PC 上编译测试。

这样的一本讲义。涵盖层面之广。讨论题材之深。都是空前的。因此我将之命名为易符真经。将它称之为经。对佛祖和孔老夫子是有所冒渎。但从六祖坛经成书的过程来看。这样的突破和创新。冒称为经还是勉强说得过去的。

虽然我是执笔人。书里的字和所有的程序都是我写的。但是其中的精义。却是摩尔先生和许多无私的符式工作者。卅多年来研究开发累积的结果。我用中文来写这本书。是切望我们中国人在廿一世纪的开始。能大步地向单芯片系统快速迈进。赶上美日欧。不要以为赶上老二老三。就沾沾自喜。我们要走的路还很长。没有好的工具和好的技术。一步一步走实在是很辛苦。符式和易符。是十分犀利的宝剑。可以帮我们摆除美日的科技枷锁。早日脱颖而出。

3.7 易符智慧科技公司

易符智慧科技公司是今年年初在新竹成立的公司，它的目标就是推广易符系统在单芯片系统上的应用。它的智慧财产包括了：

P 系列的 CPU 核心设计

易符操作系统

易符系统内软件开发工具

单芯片系统整合的技术

易符智能科技公司因此提供了制造单芯片系统的平台，它寻求的是具有发展专业产品的

伙伴，合仍开发适合生产单芯片的产品。芯片一旦进入量产，数量必定十分庞大，因此产品必然是属于消费型的电子商品。P 系列的 CPU 核心，因为构造简单而且运算速度快，特别适合掌上型省电的应用。目前正在开发的项目有：

蓝芽通讯控制器

中文电子书主控器

掌上电子游戏

易符智能科技公司欢迎电子工业上游的芯片制造设计和下游的消费产品业者来共亨这便利有效的单芯片系统的发展技术，并且切望在我们合力的推动下，能使台湾的单芯片工业能在短时间内主导世界的单芯片市场。

1-8 有无之际

老子在道德经里有一段话，发人省思:

『挺埴以为器，当其无，有器之用。凿户牖以为室，当其无，有室之用。故有之以为利，无之以为用。』

做单芯片系统，也要掌握这个无字诀。单芯片系统，只是一个平台，必须尽量留下空间，让使用者发挥他的创造力与想象力。因此在有无之际，方能收利用之效。

第二章 P24 CPU 的设计

2-1 基础篇

硅谷是一个迷人的地方，60 年代的迷你计算机【Mini computer】出现后，我们就舍弃了大计算机。大部分的计算工作都可以在实验室做了，何必去计算机中心排队？70 年代微电脑【Micro computer】来了，自己可以装自己的微电脑，干嘛还要花大把钞票去买迷你计算机？80 年代的 PC 上到每个人的办公桌上，引爆了 90 年代的网络。计算机的普及给了使用者自由，可以处理个人和工作上所有的数据。但我们也用我们的灵魂，去交换微软的 Windows 窗口和 Intel 的 CPU。从 IBM 得到的自由，却又被 Wintel 的枷锁取代了。

在 IC 芯片的容量增加到百万闸以上，而且芯片的制造已经不被美、日大厂垄断时，新一代计算机的曙光已破晓。我们可以有真正的自由，来设计、制造我们自己的计算机，及来探索、解决我们自己问题的最好途径。现在通用的计算机，其结构和其设计有其历史的渊源及限制，并没有达到理想的层面。现在我们有客观的环境，允许我们自己设计自己的计算机，我们就必须仔细地检讨这些结构和设计的缺失。去做更小、更快、更省电、更好的计算机。

设计计算机谈何容易！你若如此想，如此信，那就只好永远做 Intel 和微软的奴隶。乖乖的掏腰包买下一代的 Pentium，灌下一代的 Windows 吧!其实计算机设计不是难事，因为计算机本身就是简单的东西。我这篇文章就是第一课，希望能给你足够的数据，让你自己可以开始设计你的「梦想计算机」。

2-1-1 数字逻辑

计算机的原理是数字逻辑，计算机的制造是 IC 线路。这两条路的结合，是凸显在 NAND 闸上，NAND 闸的符号和其逻辑操作是： 图 01 NAND 闸逻辑

数字逻辑中，1 代表正，0 代表反。NAND 有 A 和 B 两个输入，一个输出 C。只有 A 和 B 都是正时，C 才为反。这个 NAND 闸可以用来合成任何的逻辑闸，例如 Inverter (NOT)、AND、和 OR： 图 02 NOT、AND、和 OR 闸逻辑

所有的逻辑操作都以 AND、OR、NOT 来表示的，因此，所有的逻辑线路都可以用 NAND 来合成。逻辑线路一般都是描述持续稳定的逻辑状态，不随时间变化的。这是开放式逻辑，输入讯号经过线路的作用，发展变化，决定了输出的讯号。

当输出讯号反馈【Feedback】而用来做输入的讯号时，现象就更有趣了。例如只用一个 NAND 闸而将其输出信号连到一个输入点，就成为： 图 03 反馈逻辑

当 A＝0 时，C＝1。但当 A＝1 时，C 就会震荡，在零与一间往复跳动。跳动的频率则可以在反馈线路上控制。此即震荡器，是驱动计算机操作的重要机构。如果用两个 NAND 闸串联来做反馈时，又有一个不同的现象。图 04 记忆逻辑

在这个线路里，如果 B＝0,那 C＝1。如果 B＝1 而且 A＝0,那 C＝0。如果 A 和 B 都是 1，那 C 就保持原形不变【No Change】。这是一个 Flip-Flop，是一个基本的记忆单元。逻辑线路（Random Logic），震荡器和 Flip-Flop 的 记忆单位，是构筑计算机或任何 IC 的基本组件。

2-1-2 杜林机（TURING ENGINE）

Alan Turing 在二次大战期间确立了计算机的理论基础。在逻辑线路，震荡器及 Flip-Flop 之外，引进了程序的观念。下面是一个最简单的 1 位计算机或杜林机。 图 05 杜林机

磁带上有一系列 0 和 1 的信息，用 0 和 1 的顺序来代表控制的程序。任何一个时间，逻辑的线路读取磁带上的 0 或 1 讯号。经过处理之后产生一些行动。这些行动，包括了：将新的讯号写进磁带现在的位置、开动磁带向前移动一格、或向后移动一格等等。根据磁带上的数据，杜林机可以进行很复杂的运算工作。但每次只处理一个位。杜林证明，这简单的数字机器，可以模仿任何复杂的数字机器，解决它们所能解的任何逻辑和数学问题。

2-1-3 计算单元

把杜林机推广到现代的计算机设计，我们可以归纳出一个计算机处理计算单元的结构。一具计算机可含有许多这种结构，这个单元里有一组缓存器【Registers】，其宽度等于数据字码的宽度。下图显示机器字码宽度为 16 位的一个中央处理器【CPU】与内存【Memory】之关系: 图 06 CPU

逻辑线路由内存中读取一笔数据作为指令，指令经过逻辑线路的译码，处理输入的数据，产生许多不同的输出数据。逻辑线路经过 Mux 选取其中一笔数据，在下一个时钟讯号（Clk）上升的时间，将这笔数据写进缓存器中。缓存器的输出可以送回，作为逻辑线路的输入，以进行下一次时钟讯号上升时的计算动作。

Mux 其实可看成逻辑线路的一部份。将他分出来，可以使逻辑线路设计简单化。因为很多相同的线路，可以用模块化的方式设计，得到最简单的线路，可以用在不同场合。

2-2 廿四位计算机 P24 的设计

一个简单快速的廿四位计算机，可以比通常的八位计算机更简单。这个计算机使用两个堆栈和四个缓存器，即可完成全部的功能。它的架构如下图所示：图 07 P24 CPU

部份的功能如下： P 缓存器： 内容为下一指令的地址。 I 缓存器： 内容为目前执行的指令。 T 缓存器： 算数逻辑（ALU）运算的核心，所有数据的处理都经过此缓存器。 A 缓存器： 内容为读写内存时，数据的地址。返回堆栈： 呼叫子程序时，P 缓存器中地址的堆置处。数据堆栈： 内容为 T 缓存器中数据的堆置处。 ALU 运算器： 取 T 缓存器及 S 堆栈最上层的数据，经过运算后，将结果存回 T 缓存器。

这计算机能执行一组精简但功能全备的指令。指令有两种格式，控制程序流程的指令，属 A 类，有这样的格式：

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 0 c c c c c a a a a a a a a a a a a a a a a

其它指令，属 B 类，格式如下：

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 0 c c c c c 0 c c c c c 0 c c c c c 0 c c c c c

每指令用六位代表，故廿四位的字码可容纳四个指令。

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 指令一 指令二 指令三 指令四

指令 代 码 功 能 DROP 011 111 将 S 堆栈顶层弹出存入 T 缓存器。 DUP 011 010 将 T 缓存器的内容压入 S 堆栈。 LDA 011 001 将 T 缓存器的内容压入 S 堆栈，并将 A 的内容存入 T。 STA 011 101 将 T 缓存器的内容存入 A，并将 S 堆栈顶层弹出存入 T。 PUSH 011 100 将 T 缓存器的内容压入 R 堆栈，并将 S 堆栈顶层弹出存入 T。 POP 011 000 将 T 缓存器的内容压入 S 堆栈，并将 R 堆栈顶层弹出存入 T。 NOP 011 110 无变化。 COM 010 000 将 T 缓存器的内容位取反转值（1's Complement）。 SHR 010 010 将 T 缓存器的内容右移一位，最高位补 0。 SHL 010 001 将 T 缓存器的内容左移一位，最低位补 0。 ADD 010 111 将 S 堆栈顶层弹出，加入 T 缓存器。 AND 010 101 将 S 堆栈顶层弹出，与 T 缓存器的内容做 AND 动作后存回 T。 XOR 010 100 将 S 堆栈顶层弹出，与 T 缓存器的内容做 XOR 动作后存回 T。 MUL 010 011 单步乘法，重复 24 次后，T-A 缓存器的内容即乘积。 DIV 010 110 单步除法，被除数置 T-A 缓存器，重复 24 次，即得商及余数。 ST 001 111 将 T 存到 A 所指的记忆地址，并将 S 堆栈的顶层弹出存入 T。 STP 001 101 做 ST 动作后，将 A 自动加 1。 LD 001 011 将 T 压入 S 堆栈后，并将 A 所指记忆地址的内容存入 T。 LDP 001 001 做 LD 动作后，将 A 自动加 1。 LIT 001 010 将 T 压入 S 堆栈后，并将此指令下一字码(24 位)取出存入 T。

A 类的指令含有一组 18 位的地址，若要跳到新的地址，将此 18 位取代 P 缓存器中右边的 18 位，即是新地址。

指令 代 码 功 能 JMP 000 000 跳到指令中所指定地址。 BZ 000 010 若 T＝0，跳到指令中所指定地址。 BNC 000 011 若 Carry 在上次算数运算中未被设为 1，跳到指令所指定地址。 CALL 000 100 呼叫子程序，将 P 中地址压入 R 堆栈，并跳到所指定地址。 RET 000 001 将 R 堆栈顶层弹出存入 P 缓存器 (子程序的回归指令)。

因跳跃指令中是 18 位的地址，所以，此计算机的内存乃是以 256K 字码为单位所分成的许多区块。程序只能在相同的 256K 字码区块之内跳跃。如果要在 24 位地址的 16M 字码全程范围中跳跃，则可先用 LDI 指令将地址存入 T，再用 PUSH 指令将此地址压入 R 堆栈，然后再执行 RET，即可跳至内存中的任何地址。

2-3 计算机的设计与制造

前面有了一台计算机基本的架构和指令集。依照这样的规格，去实际设计计算机的线路，并且自己制造出来，这在以前是不可能以个人财力和资源来达成的。但此时此地，这已经不是渺不可及的梦想了。进行计算机线路设计，有许多不同的方法，设计好了的线路要去制造实际的计算机，也有更多不同的途径。在现在的计算机辅助设计【CAD】技术进步的情况看，我们只要将线路用 VHDL 或 Verlog 等高阶设计语言写明白，就可以找到专业的线路设计工程师，将其转换成 IC 设计图。这些 IC 设计图就可以送到晶元代工厂制成芯片。

生产芯片可以完全借重代工，但生产大量芯片还是十分昂贵的过程。做少量研究实验，就可以用 FPGA【Field Programing Gate Array】。将 IC 设计灌入或烧录到 FPGA 中，马上可

以测试验证，看设计是否正确可用。若有错误，可以在 VHDL 或 Verilog 中进行修改，然后再下载测试。

上述的 P24 计算机设计，若是用 VHDL 来表述，可参考本文后面的 VHDL 原程序文件 p24e.vhd 之内容。此程序乃是用 Xilinx 的 Foundation i3.1 设计系统写成并经过实证的。特附录于后供大家参考研究。

2-3-1 芯片脚位设定

VHDL 程序中 60 行开始设定 P24 芯片的脚位。 holdosc0 此脚须为高位，才能开动外部的振荡器。此是外接线路的特性，一般不须要。 clk 主振荡器，在 FPGA 上我采用 8MHz，实际若用 Xilinx Vertex 1000E 可达 20MHz。 arst 重设讯号，低电位时系统重设，由 O 地址开动。 uart_in RS232 输入埠。 uart_out RS232 输出埠。 interrupt 5 条中断输入线。 icode 6 条讯号线显示目前执行的指令。 data 24 条讯号线显示目前数据汇排状态。 address 11 条讯号线显示目前地址汇排状态。

此设计目前已在 Xilinx 的 Virtex1000E 型 FPGA 上验证成功，此芯片内含有甚多的 RAM 结构，因此不须要外挂 RAM 或 ROM 芯片。data 和 address 两汇排只是用来观测芯片内部的动作。实际制作芯片时，此二总线要按需要接适合的内存。

程序中 74 行是选用内部 RAM 的方式。P24RAM 选用 2048 个 24 位的 RAM 来存放程序及数据。

2-3-2 内部讯号的设定：

程序 113 行开始是内部讯号的设定。 as_stack 数据堆栈，16 层，25 位宽。 ar_stack 返回堆栈，16 层，25 位宽。 sp、spl 资料堆栈指针。 rp、rpl 返回堆栈指针。 at T 缓存器。 s 数据堆栈输出端口。 r 返回堆栈输出端口。 i 指令缓存器。 p 指令地址缓存器。 sum 加法器输出埠。 a 地址缓存器。 t_in T 缓存器输入端口。 s_in 数据堆栈输入端口。 r_in 返回堆栈输入端口。 a_in A 缓存器输入端口。 p_in P 缓存器输入端口。 t_sel、p_sel、a_sel、r_sel 相关输入端口 Mux 的选择讯号。 spush、spopp 数据堆栈工作讯号。 rpush、rpopp 返回堆栈工作讯号。 tload、aload、pload、iload 相关缓存器输入锁定讯号。 add_sel 地址输出端口选择讯号。 reset 重设信号。 write 内存写入讯号。 uload RS232 输出输入锁定讯号。 addr 内存地址总线。 data_out、data_in 内存数据输入输出总线。 clk0 内存使用时钟脉冲。

2-3-3 常数设定

程序 133 到 195 行是一些常数的设定，将数字讯号用常数定名，以后在设计程序中引述，比较容易看得清楚。

133-157 行 机器码的定名。160-174 行 T 缓存器输入可能讯号来源的定名，大多数 P24 处理的数据都必须经过 T 缓存器，所以 T 缓存器输入的选择是比较麻烦的。 177-181 行 A 缓存器输入可能讯号来源的定名。 182-185 行 R 堆栈输入可能讯号来源的定名。 188-191 行 P 缓存器输入可能讯号来源的定名。 192-195 行 输出地址端口选择讯号的定名。

2-3-4 同步讯号的设定（Concurrent Assignments）

198-271 行 是同步讯号的设定 201-207 行 简单讯号线的交换设定。 210-219 行 使用 Virtex Block RAM 的设定。 221-233 行 icode 将 code 输出到接脚。sum 是 T 和 S 之和的线路。 226-239 行 t_in 是 T 缓存器前 Mux 的输出端口。它有 13 种不同的可能讯号来源。 242-246 行 I 缓存器分出四个机器码到 code 在线。 249-252 行 A 缓存器的输入讯号设定。 252-255 行 返回堆栈的输入讯号设定。 257-260 行 P 缓存器的输入讯号设定。 262-263 行 接到内存地址线的选择设定。 266-271 行 z 是当 T=O 时变成高位。

2-3-5 顺序讯号的设定（Sequential Assignments）

272-303 行 是 decode 程序中顺序讯号的设定方式。在此顺序中所有 p24 的指令应该产生的控制讯号，都在这一段程序中规定。当所有缓存器及堆栈中的数据稳定后，即产生这些控制讯号。当下一个主振荡器的脉冲到达时，新的讯号即被锁入合适的缓存器及堆栈中。如此周而复始，p24 即按着规定的程序执行机器码。 276-291 行 是各个控制讯号的初设值。 292-303 行 是执行 Slot 0 工作的控制讯号，有中断讯号时，即依中断埠的状态，呼叫在内存 1-31 位置的子程序，处理中断情况。若无中断讯号，即由内存中加载下一指令，在 Slot1-4 中连续执行。 305-433 行 是执行机器码时所须产生的控制讯号。在下一主振荡器脉冲到达时，锁定所须的数据。 436-488 行 这里是执行机器码的 fsm 机制（Finite State Machine）。 438-450 行 是重设时各缓存器及堆栈的状态。 452-488 行 则是当主振荡器脉冲到达，如果按机器码选定的控制讯号未锁住到缓存器及堆栈的讯号。

2-3-6 eForth 的程序

eFoth 的程序是由 Meta24.f 产生的机器码表，此表存在 log.f 档案中。此档中的机器码必须抄进 (Foundation 3.1 i 中叫出的 Core Generator 建造的) p24ram.mif 档案中。如此，当 Foundation 综合 p24e.vhd 文件时，即会将 eForth 程序加入综合文件 p24.bit 中。P24.bit 在 Programming 的操作项目中，即可下载到 Virtex1000E 芯片中，下载完毕 p24 即开动执行 eForth 的程序。使用者即可由终端机下达指令给 p24 执行。

第三章 P24 组合器兼编译的设计与建造

我们有了这样的一台计算机，他的中央处理器是 P24 附加了一些内存，并可以连接一个 RS232 的终端机，这样就是一台完整的计算机，就可以做计算机该做的事了！对不对？这样是差不多了，只是计算机一开动就要执行一些指令，或是程序。这些指令和程序怎么能进入计算机去让他执行呢？我们先复习一下这个 CPU 的指令，他有一组指令是控制程序执行的顺序的。其中以下几个：

指令 代 码 功 能 JUMP 000 000 跳到指令中所指的地址。 BZ 000 010 若 T＝0，跳到指令中指定的地址。 BNC 000 011 若 Carry 在上次算数运算中未被设为 1，跳到指令中指定地址。 CALL 000 100 呼叫子程序，将 P 中地址压入 R 堆栈，并跳到指定地址。

这些长指令，每个都占 24 位。其中前 6 位是指令码，后面的 18 位是将要跳跃去之目的地址。其它的指令都只有 6 位。所以在一个 24 位的字码中，可以填充 4 个指令。算数运算的指令有： COM、SHL、SHR、AND、XOR、ADD、MUL、DIV 堆栈和缓存器的指令：PUSH、POP、LDA、STA、DUP、DROP、NOP 内存存取的指令： LD、ST、LDP、STP、LIT、LDRP、STRP 结束子程序的指令： RET

程序设计师的任务，就是把这些指令，在内存里做适当的安排。完成许多不同的程序，可以做使用者需要做的各种工作。在没有其它的工具可以使用的情形下，有本事的人可以直接在内存里填入机器码。在 P24 的计算机里，你可以从位置 0 开始写入机器码。计算机一开动，就从 0 位置读第一个字码。这字码中可以有一到四个机器码。CPU 执行完了这几个指令后，就由下一个位置读一个字码，执行完了，会再读下一字码执行。如此继续，直到工作做完或计算机停止。

这是最原始的组合器【Assembler】，我把他叫做零阶组合器【Zero Order Assembler】。写程序的设计师，直接用机器码来建造程序。这个工作，在设计建造一台新的计算机时，可能需要用到，因为没有其它现成的组合器可用。

假如你嫌用机器码来写程序太麻烦，我们可以自己来写一个组合器程序当作工具。这个工具可以把你用汇编语言写的程序，翻译成为计算机可以执行的机器码。现在市售的各种型号的 CPU，都有厂商供给之组合器。让使用的人用一台 PC，就可以用汇编语言来写作程序。大多数的 CPU 厂家，甚至都可以提供 C 语言的编译器【Compiler】。让你用高阶的 C 语言来写作程序。

我们这台自己设计的 P24 计算机，当然没有厂家提供组合器或是编译器。但，如果没有人提供组合器或编译器，我们就不能为 P24 编写甚么程序了吗？组合器和编译器真是那么神秘的东西，若没有提供我们就不能生存吗？

3-1 建立组合器

其实，组合器和编译器也不是很困难的工作，在此我们就讨论怎样设计和建造 P24 的组合器和编译器。你学会了自己如何写 P24 的组合器和编译器，就可以为任何 CPU 写组合器和编译器，不需要在仰赖微软或其它公司提供的工具了。

3-1-1 组合器兼编译之功能

这套我们自行开发的组合器，可以组合最佳化的机器码，也可以编译 Forth 语言的高阶程序，综合了组合器和编译器的功能和优点。在组合机器码时，它尽可能将许多 6 位的机器码编入一个 24 位的字码中，每个字码可以容纳四个机器码。

编译 Forth 语言的一个高阶指令，大部分都被编成以 call 指令来呼叫子程序的方式。因而，在组合程序里可直接加入高阶指令，同样在高阶程序中，也可很容易编入组合的机器码。这样程序员可以写出最好有效的程序，一方面充分发挥机器码的效用，一方面仍可保持高级语言的结构。

3-1-2 组合器相关档案

这一套组合器是用 Win32Forth 系统写成的。Win32Forth 是一套免费公开的 Forth 系统，你可以从 www.forth.org 网页下载。在该网页，点选 Resources 下的 Forth Compliers 项目，再点选其中 Windows 系统之 Win32For V4.2 的 Self-installing Binary 项目，就可以免费获得目前最新版的 Win32Forth。下载后，执行它，即可自动安装到你的 PC 上。

这里所提供的 P24 组合器包括了下列几个档案： META24.F 组合器设定程序总文件 OK24.F 组合器程序代码 KERN24.F FORTH 的系统核心指令集 EF24.F FORTH 操作系统指令集

进入 Win32Forth 后，执行 fload meta24.f，看到「ok」即已将 P24 的 eForth 系统建造完成。可以移植到 P24 计算机系统中启动 P24。这章中，主要将讨论如何建造组合器及编译器的工作。Forth 的系统和功能则在下一章中详述。META24.F 程序文件它会先做一些准备工作，然后它加载 OK24.F，建成组合器。再用该组合器加载 KERN24.F 建成 P24 的 eForth 系统内码。

Win32Forth 是一套专门为 Windows 环境所设计之 32 位的 Forth 系统。我们现在把它用来做 24 位之 P24 计算机的组合器，在 Windows 下，组合成一套完整的 eForth 系统内码，可以直接下载到 P24 计算机上使用。 META24.F 的主要功能就是在 Windows 的环境下，建造一个方便的环境，用以组合 P24 的机器码和高阶指令，进而建造 Forth 系统。它最重要的工作，就是在 Windows 中预留一个 32KByte 的 IC 记忆区，我们用以填入 24 位的 P24 机器码。

3-1-3 机器码的组合原理

为了方便存取机器码数据，我们用 32 位（4 Bytes）储存一笔 24 位的 P24 的字码数据，所以在 32K Bytes 的 RAM 区域中，我们可以组合 8K 字码的 P24 指令（程序）。由 P24 之地址去索取其中字码的指令是 RAM@，将 24 位的数据存入 P24 某一地址的指令是 RAM!。这两个指令是此组合器的基本指令。

在组合 P24 程序后，我们可以用 SHOW 的指令来检视组合的结果。整套 P24 Forth 内的机器码，可以用 SHOWRAM 及 BRAM 两指令显示出来。所显示的结果经检视无误后，就可用来烧 PROM，或下载到 P24 计算机中去执行。

这些指令都用 RAM@和 RAM！定义好后，META24.F 就载入 OK24.F，将 P24 的组合

器建造好。组合器造建好之后，就接着载入 KERN24.F。KERN24.F 中所包含的是 P24 的 Forth 低阶指令集。这些指令都是用 P24 的机器码所编写定义成的。这些指令集把 P24 扩充为一个完整的 Forth 虚拟机，可以执行 Forth 高阶指令。而将 Forth 虚拟机，再度扩充后就成为一个完整的一个 Forth 操作系统了。这些 Forth 高阶指令的原程序都在 EF24.F 檔中。

P24 组合器的程序在 OK24.F 檔中。这个组合器主要的工作，就是要能将一到四个 P24 的机器码，编入一个 24 位的 P24 字码中。碰到跳跃的指令如：JUMP、CALL、BZ、BNC 时，一个 24 位的字码只能塞一个指令。如果已经组合的机器码，尚未塞满一个字码，就要以 NOP 先补满，而将跳跃指令塞到下一个字码中。

为了要方便塞四个机器码到一个字码中，我们用 H 指标指到下一个空的字码、用 Hw 指标指到现在在建造的字码、而用 Hi 指到在目前建造的字码中，以便在适当位置塞进新的机器码。所有 6 位的机器码组合工作都是用 INST 指令来定义的。

3-1-4 组合器工具指令集

这一组指令是为了设定组合器的工作环境而准备，使组合器方便建造汇编语言的程序。此处简单介绍这些工具指令。

名 称 说 明 ASM24 组合指令辞典，此辞典收集所有组合指令及所建的系统指令。 SIM24 仿真器指令辞典，此辞典收集所有仿真器建造所需的指令。 .HEAD 在定义新指令时，印出指令的名称及地址。 RAM 32KB 的记忆区，用以存放 2K 字码的 P24 机器码及高阶程序。 RAM@ 由 P24 地址取其中 24 位字码数据。 RAM! 将 24 位字码数据存入 P24 记忆区的地址内。 RESET 清除 RAM 区所有数据。以便重新编造 P24 程序代码。 SHOW 由 P24 地址印出相连的 128 个字码内容。 SHOWRAM 印出全部 P24 记忆区中所有字码的内容。 BRAM 以特定格式印出记忆区中 P24 程序代码的全部内容，印出的结果可以直接贴入 P24.VHD 檔中做 FPGA 试验。

此外尚有，forth'、forthDUP、forthDROP、forthOVER、forthSWAP、forth@、forth!、forthAND、forth+、forth-、forthWORD、forth.(、forthCOUNT 等指令，用以保证在 P24 系统建成后，还可使用原来 Forth 系统中的 '、DUP、DROP、OVER …等等这些基本指令。

3-1-5 P24 组合器（ASSEMBLER）

P24 的组合器程序代码在 OK24.F 檔中。这个组合器是一个具结构化和最佳化的组合器，可以组合 P24 的机器码。它充分发挥 P24 的特性，能在一个 24 位的字码中塞入 4 个机器码。它也能很方便地以结构化方式编译子程序，以建造 Forth 语言的高阶程序。

因为 P24 高阶程序是以子程序方式编译的，所以高阶程序和低阶的机器码可以任意混编。因而程序员可以自由发挥 P24 的优点，写出最佳效率的程序。这个 P24 组合器，因而具备了高级语言编译的功能。 P24 组合器有 H、HW、Hi、Bi 等四个指标，用来决定新的机器码或数据应存入 P24 RAM 记忆区中哪个字码及其中之位置。另有其它相关指令，如下表：

名 称 说 明 H 指到 RAM 中下一个可以储存机器码或数据的 24 位字码。Hw 指到现

在正组合机器码的 24 位字码。 Hi 指到现在正组合机器码的 24 位字码中接着可使用的 6 字节。 Bi 指到现在正在组合数据的 24 位字码中接着可使用的 8 字节。 ORG 设定 H 的指针，由此位置开始组合新程序。 #, 将一笔 24 位字码数据存入 H 所指的位置，同时将 H 指针值加 1。 ,W 将一个 6 位的机器码编入 Hw 所指的字码中，放在下一个可以使用的 6 字节位置。 ,I 组合一个 6 位机器码，若现在处理的字码已装满 4 个机器码，则将新机器码填入下一个字码中。 ,B 将一个 8 位的数据填入目前处理的字码中，若此字码已三个装满 8 字节的数据，则将数据填入下一个字码中。 INST 这是组合器中最重要的指令，它是用来建造所有 6 位机器码的指令。

3-1-5 六位机器码

所有 6 位机器码都是用 INST 这个指令建造的，例如 NOP 就是这样定义的：

79E79E INST NOP

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0

NOP 的机器码是十六进制数 1E（八进制数 36），十六进制数 79E79E 是一个 24 位的字码，其中包含了四个 1E 的 6 字节。

在组合器执行到 FORTH 低阶程序中的一个 NOP 指令的时候，NOP 指令会藉由 Hw 及 Hi 知道，现在应该是在那个字码中之适当位置，即将 1E 填入那位置的一个字节内。这机制的关键在于，NOP 会自动采用 Hi 选择适合的 6 位面罩，在 24 位信息 79E79E 里，选取其中一个对应的机器码 1E，填入目前正处理的字码中，其编码工作即算大功告成了。

依此简单原理，几乎所有 6 位的机器码指令都用可藉 Forth 指令 INST，如此照样定义出来：

字 码 Forth 指令 汇编语言名称 041041 INST Ret 249249 INST Ldp 2CB2CB INST Ld 34D34D INST Stp 3CF3CF INST St 410410 INST Com 451451 INST Shl 492492 INST Shr 4D34D3 INST Mul 514514 INST Xor 555555 INST And 596596 INST Div 5D75D7 INST Add 618618 INST Pop 659659 INST Lda 69A69A INST Dup 71C71C INST Push 75D75D INST Sta 7DF7DF INST Drop

在 6 位的机器码中，lit 很特别。它除了将 0A 的指令码填入现在正在处理的字码中，还会将随带的一个 24 位数值，组合到下一个字码中。将来当 CPU 执行到 lit 的时候，会自动将下一个字码的数值取出放在堆栈上。

3-1-6 跳跃指令

跳跃指令有 JUMP、CALL、BZ、及 BNC 四个。这四个跳跃指令都是 24 位的。除了前面 6 位的指令码之外，还有后面 18 位的地址，因此每个指令都会占掉整整一个 24 位的字码。当组合器碰到一个跳跃指令时，必须先结束正在处理的指令字码。若尚未填满四个 6 位的指令码，必须用 NOP 补满，才能在下一个字码中填入跳跃指令。

Forth 指令 anew，就是这样一个能将现用字码以 NOP 补满的指令。如果目前正在处理

的字码尚未填满四个 6 位的指令码，anew 即将余下的空间都用 NOP 补满。以便开始一个新字码的处理。若该字码已填满了四个机器码，则不必做任何事。

JMP 是类似 INST 的组合指令。它是用来建造所有此类的跳跃指令，例如：

字 码 Forth 指令 汇编语言名称 000000 JMP JUMP 100000 JMP CALL 080000 JMP BZ 0C0000 JMP BNC

UNTIL 和 -UNTIL 分别为 BZ 和 BNC 的同义指令。在 Forth 语法中，我们并不用 label 来标定程序中机器码的地址，藉以组合成一个 24 位的跳跃指令。而是用结构化的指令，来建造高阶的回路和分岔，像是： IF …… THEN IF …… ELSE …… THEN BEGIN …… AGAIN BEGIN …… UNTIL BEGIN …… WHILE …… REPEAT

-IF、-UNTIL、及 -WHILE 的功能与 IF、UNTIL、及 WHILE 相同。只是条件不是 T=0，而是 Carry≠0。这结构化的指令都是由 JUMP、BZ、BNC 来定义的，如下： : begin anew H @ ; : if begin 0 bz ; : -if begin 0 bnc ; : skip begin 0 jump ; : then dup >R >R begin 3FFFF AND R> RAM@ XOR R> RAM! ; : else skip SWAP then ; : while if SWAP ; : -while -if SWAP ; : repeat jump then ; : again jump ;

3-2 高阶指令的编译

在这个 Forth 系统中，所有系统指令都可以用呼叫子程序的方式被编译在新定义的系统指令里，所以一个高阶指令就是以一序列呼叫子程序的机器码所定义的程序。一个简单的例子： :: : TOKEN $,n] -;'

此为定义 : ，冒号（COLON），这指令的原始程序代码，执行冒号指令时，它就会执行下面这一系列呼叫子程序的机器码： CALL TOKEN CALL $,n CALL] CALL EXIT

在这个组合器里，所有的指令都用 CODE 或 :: 来定义的。一般在定义 Forth 系统时，CODE 是用来定义一个低阶程序，用机器码组合成的指令。而 :: 是用来定义一个用 Forth 指令组合的高阶指令。但在这组合器中，机器码和高阶指令可以混编。所以 CODE 和 :: 是完全一样的，它们的定义如下：: CODE makeHead begin .head CONSTANT DOES> @ call ; : :: CODE ;

这 CODE 指令，会先在 p24 的 RAM 区域里，建造一个新指令的头部，其中包括了连接区（Link Field）及名称区（Name Field）。名称区之后就是执行码区（Code Field）。CODE 会将执行码区的地址，存在它自己的参数区内（这是 CONSTANT 所做的工作）。当这个新指令在被组合器执行的时候（被用于别的程序中），它就从参数区中将执行码区的地址取出来给 CALL，编成一个副程呼叫形式的机器码。

我们在以后的系统原始程序，EF24.F 档案中，将看到许多系统指令都是用 :: 定义的。在 :: 定义后，凡遇到高阶指令时，一律都编译成为呼叫子程序的 24 位机器码。若遇到机器码指令时，就将该机器码尽可能地塞进可放四个机器码的一个字码里去。

:: 会先结束前面定义的一个指令，组合一个 RET 机器码。

-;' 亦会结束一个高阶指令。但若程序的最后是一个用 CALL 呼叫高阶指令的机器码，

就不必组合一个 RET 了，而可将该呼叫子程序的 CALL XXX 指令变成 JUMP XXX。这样可节省一个字码，又少了不必要的 CALL 与 RET 进出返回堆栈之动作。

3-3 其它组合器指令

另外还有几个组合器指令，可帮助或简化组合程序的写作： LIT，即 ldi，组合一个紧随的即用数值（Immeadiate Literal） $LIT 则组合一个字符串【String Literal】。

一个字符串是将每三个 8 位数据，压缩在一个 24 位的字码中，并且将剩余部份补为 0。字符串的第一个 8 字节内是字符串的长度，例如 6 个字母的「NUMBER」字符串，组合成的结果是：

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 0 1

6 'N' 'U'

字码 0 1 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 1

'M' 'B' 'E'

字码 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

'R' 0 0

每个指令头部的名称栏，就是用这样的方式所组合的。 : for push begin ; : next doNEXT jump ;

另外，此处所列的这两个指令，是建造 FOR …. NEXT 回路之程序结构所需要的指令。

总而言之，这是一个简单但极为有效的组合器，并且兼具高级语言的编译功能。虽然这个组合器是为了 P24 所设计的，但是，它的原理可以广泛运用于任何的 CPU，或特殊的指令集中。有了这样的组合器，我们可以为任何新设计的计算机写作程序。

当然，复杂的计算机有复杂的指令集。要组合复杂的指令集，组合器也一定会变得复杂。可是计算机不一定要复杂才好。复杂的计算机用的闸数大大增多，耗电量也加大，总的效果不见得会好。RISC 计算机的设计，证明了它可以比 CISC 计算机更有效。但是目前的 RISC 计算机也越变越复杂了，背弃了原来设计的初衷。P24 的结构，说明了我们如何可用最少的闸来制造出简单快速的计算机。简单的计算机需要的软件工具也应该是简单易用。这个组合器就是一个最好的例子。

第四章 FORTH 系统(OS)的设计和建造

这里，我们就来看看，如何用 P24 的组合器，来建造 Forth 系统的译码器和编码器，以及一些常用的侦错除错工具。这样就将成为一个完整的操作系统。

我们有了个 P24 的计算机，我们又有了组合器，可用来写程序，使用这台计算机。要写甚么程序，是看我们要用计算机来做甚么事情所决定的。

每人要做的事情并不一样，我无法事先知道，也不能样样都帮你做。但我可做一件事，就是在 P24 上装一个操作系统。有了操作系统，若再附上高级语言的编译器，任何人经过稍加训练，便可以用这台计算机写自己的程序。解决自己的问题了，岂不皆大欢喜？

Forth 就是这样的一个东西，它在计算机里面，能够接受你的指挥。它是一套指令集，你可以在键盘上输入一些指令的名称，它就会去依序执行。执行完了再等你输入新的指令。而这就是一个 Forth 的译码器【Interpreter】。

Forth 的指令，好像文字词汇一样，很容易记，容易使用。但也包括了一些符号，还是需要学习和认识的。一套指令集不管有多少指令，总是不够用的，因为我事先并不知道你要用它来做甚么特定的事情。但是，Forth 系统除了译码器之外，还有一个编码器（Compiler）。使用编码器，你就可以自由地用现有的指令，来构筑你所需要的新指令。构筑新指令的文字文件，就是 Forth 的程序。把 Forth 的程序加载计算机，就是把文字用编码器编成新的指令集。而新的指令集，就可以很方便地解决你面对的相关问题。

这里，我们就来看看，如何用 P24 的组合器，来建造 Forth 系统的译码器和编码器，以及一些常用的侦错除错工具。这样就将成为一个完整的操作系统。P24 加上这样的一个 Forth 操作系统，就可以实际用来解决许多的应用问题了。

Forth 的操作系统，架构很简单，就如下面的易符系统芯片图解所示。P24 开机后即执行 COLD。COLD 送出开机讯号就进入 QUIT 回路。QUIT 接受输入字符串，用 EVAL 来处理该字符串。在译码时 EVAL，依变量 'EVAL 内容，将字符串交给 $INTERPET 处理，编码时则将字符串交给 $COMPILE 处理。图中提到的指令在下面略作说明，以后再将整个系统的指令分节解说。 （）图 08 易符系统芯片

COLD 开机启动。 QUIT 接受输入的文字序列，处理其中字符串，是个无限回路。 QUERY 接受输入可长达 80 字的文字序列。 EVAL 有限回路，在文字序列中分出一个以空白隔开的字符串，并解读该字符串。 'EVAL 变量，内容为 $INTERPRET 或 $COMPILE，据以决定如何处理 EVAL 所分出的字符串。 $INTERPRET 译码器。在字典中查所给的字符串，若是找到即视为指令来执行。若找不到，则试着转换为数码。若亦非数码，就交给 ERROR 处理。 $COMPILE 编码器。在字典中查所给的字符串，若是被宣告为「立即」属性之指令，会立即被执行。若是普通指令则将其编入字典中。若非指令，即试着转换为数码，并以数码结构编入字典，若非数码，就交给 ERROR 处理。 NAME? 查字典是否有与字符串相符的指令。 NUMBER? 取变量 BASE 内容为基数，将字符串转换成数值。 EXECUTE 执行查

到的指令。 COMPILE 将查到的指令地址编成 CALL 码，编入字典。 LITERAL 将转换成的数值以数码结构编入字典。 ERROR 印出无法处理的字符串，并跳回 ABORT 再启动。

4-1 Forth 系统的核心（KERNEL）

Forth 系统的核心是一套指令集，提供系统所需基本的一切功能。这些功能都需要交给 P24 去执行。P24 本身已具备了相当大的一部份功能，但是不足以支持整个 Forth 系统的运作。此核心里的指令，都是用 P24 的汇编语言来建造的。有了这一套指令集的支持，操作系统中其它较高阶又复杂的功能（指令），都可以用这套核心指令来撰述。

Forth 系统的核心指令都在 KERN24.F 档案中，现在我们就依照档案中指令定义的顺序，来看看这些核心指令的功能，和它们被建立以后的使用方法。

4-1-1 系统变量

在译码器和编码器运作的时候，许多数据要存入内存的特殊位置，留下记录，或者要从内存中取出来参考，这一套相关的系统变量是： HLD 将数值转换成 ASCII 字符串时，系统所用暂存字符串的指针。 SPAN EXPECT 指令所接受到字符串的长度。 >IN 输入字符串中下一个译码器要处理位置的指针。 #TIB 译码器所处理字符串的长度。 'TIB 变量，内容为终端机输入文字的暂存区。 BASE 变量，内容为数值转换时所用的基数。 CONTEXT 查字典时的搜寻指针，最先要查之指令名称栏。 CP 字典上空白区开始的地址。 LAST 字典上最后一指令的名称栏 'EVAL 变量，内容为 $INTERPRET 或 $COMPILE 的地址，用以决定系统是译码或编码。 'ABORT 变量，内容为 QUIT 的地址，发生错误时要执行的指令。 TEXT 文字解压后储存的暂存区域。 tmp 编码缓存器。

4-1-2 宏指令

P24 的机器码大部分都是 Forth 的基本指令。但也有一些较复杂的 Forth 指令，需用几个机器码组合而成。这种指令，一般可用子程序的方式来建造。但是在 P24 计算机中，一个 24 位的字码可以塞入四个机器码，所以如果一个 Forth 指令可以直接用四个或更少的机器码来完成，就比用子程序来建造要节省记忆空间。并且运算速度也可以加快许多。这种 Forth 指令在我们组合器中，就是以宏的方式来呈现。宏指令直接可将机器码组合起来，避免了子程序呼叫所需进出返回堆栈的负担。： EXIT ret ; 子程序返回。： EXECUTE push ret ； 执行目前堆栈上的地址。： ! sta st ； 将数据存入内存的指定地址中。： @ sta ld ； 从内存的指定地址中取出数据。： R> pop ； 将返回堆栈上的数弹出，压入堆栈。： R@ pop dup push ； 将返回堆栈上的数复制，压入堆栈。： >R push； 将堆栈上的数弹出，压入返回堆栈。：SWAP push sta pop lda ； 交换堆栈最上二数值的位置。：OVER push dupp sta pop lda ； 复制堆栈上之第二数值。： 2DROP drop drop； 弹出并弃除堆栈上的两个数值。： + add； 弹出两个数值，相加后放回堆栈。： NOT com； 将堆栈上的数变成相反值。： NEGATE com 1 ldi add ； 将堆栈上的数变成负值。： 1- -1 ldi add； 将堆栈上的数值减 1。： 1+ 1 ldi add； 将堆栈上的数值加 1 。： BL 32 ldi ； 将空格码压入堆栈。： +! sta ld add st； 将数值加入内存的指定地址中。。： - com add 1 ldi add； 弹出两个数值，相减后放回堆栈。

有些宏指令，如 OVER，需要 5 个机器码，比用子程序好像还要多占一点空间，似乎并不很适当。但是，如果它们在编码中，前后都是机器码，或者是其它宏指令所产生的机器码，它们即可有效地合并，连续塞入机器码，而不必像组合子程序呼叫指令时，要先把未完成的字码用 NOP 补满，因而会节省空间，也可加快执行速度。

使用宏指令，整个系统会组合得更精简快速，更能充分发挥 P24 计算机的功能。唯一的缺点是宏指令只能在 P24 的组合器内工作。在实际操作时，译码器和编码器都无法直接利用宏指令。因此我们必须在操作系统中，另外加上一套与宏指令相对应的子程序指令，专供译码器和编码器使用。这些补强译码器及编码器的指令都定义在 OK24.F 档案中。

4-1-3 组合指令集

下面这一组指令都是 P24 的子程序指令，其内部的功能都是用 P24 机器码组合成的。我们为了要 P24 的运算速度能再加强到最快的速度，所有子程序可能用机器码组合的都尽量如此建造起来，并都收集在 KERN24.F 档案中。这些指令包括了： O< (n -- f) 若 n 小于 0，则 f=-1，否则 f=0 OR (n n -- n) 或 UM+ (n1 n2 -- sum carry) 算出 n1 加 n2 的和及其溢位号。 ?DUP (n -- n n | 0) 若 n 为 0 则复制一份。 ROT (n1 n2 n3 -- n2 n3 n2) 将堆栈最上三数旋转。 2DUP (n1 n2 -- n1 n2 n1 n2) 复制堆栈最上二数。 DNEGATE (d -- d) 将堆栈上双数改变数值符号。 ABS (n -- |n|) 将堆栈上数值变为绝对值。 = (n1 n2 -- f) 若 n1=n2，f=-1；否则 f=0。2! (d a --) 将双数 d 存入 a 地址。2@ (a -- d) 从 a 地址取出双数 d。COUNT (a -- a+1 n) 从 a 地址取出 n，并将 a 加 1。 B> (b a -- b+1 a) 将地址 b 处未压缩数据字码取出，移入 a 的最低 8 位，a 中数据向左移八位。这是压缩字符串到 24 位字码的基本指令。>B (a b -- a+1 b+3 cnt) 将 a 处的 24 位字码解出三个 8 字节放到 b+0、b+1、b+2 位置，b 加 3 可以继续解压。a 中最高的 8 位，则作为 COUNT，它可能是 a 中压缩字符串的长度。

4-1-4 终端机指令

EMIT (c --) 将 ASCII 码 c 弹出，到终端机上显示。 KEY (-- c) 等使用者在终端机上按键，并将键入的 ASCII 码 c 放到堆栈上。

P24 的终端机接口是一个有效又很简单的设计。在执行 SHR 指令时，T 缓存器中的数据都向右移一位，此时 T(0) 位即被挤出。但我们将 T(0)送入一个正反器，而这个正反器的输出直接就连到一个输出脚位上，此即 P24 的终端机输出端口。

当执行 SHR 指令时，我们将终端机输入脚位的状态锁入 T(24) ，这是 T 缓存器的溢位【Carry】，此即终端机的输入端口了。终端机的输入和输出，即全由 EMIT 及 KEY 来操控。EMIT 及 KEY 用 50us 及 100us 两个指令来延迟到 9600 Baud 的半位及 1 位的时序。

4-2 系统常用指令

4-2-1 比较指令：

< (n1 n2 -- f) 若 n1< 用双数 d 除以 n，得余数及商。 q ） r – n MOD（ d M 用 n1 除以 n2，仅留商。 ） q -- n2 （ n1 用 n1 除以 n2，仅留余数。 MOD（ n1 用 n1 除以 n2，得余数及商。 24 次（六个字码）即完成了除法。 -u 放在堆栈 S 上，重复执行 DIV 存器中，将

在 P24 有单步除法指令 DIV，ud 放在 T-A 暂 堆栈上。由此将衍生出下列其它除法指令。除以单自然数 u。产生余数 r 及商 q，保留在 这是除法的基本指令，它将一个双自然数 ud u MOD（ ud UM 4-2-2 除法指令 这些指令看似简单，但很不容易直接用简单的几个机器码来组合，因此保留它们用高阶的型式来定义。则 f="-1；否则 f=0。" 若 n1<="n3<>

4-2-3 乘数指令：

UM *（ n1 n2 -- d） 将自然数 n1 乘以 n2，得双数积。这是乘法的基本指令，其它乘法指令皆从它衍生出。P24 有单步乘法机器码 MUL。将 T 清作 O，n1 放在 S 内，n2 放在 A 内，重复执行 MUL24 次，即可在 T-A 中得到 48 位的积。

其它的乘除指令是： * （ n1 n2 -- prod） 乘。 M * （ n1 n2 -- d） 双乘，算出 48 位的双数积。 */ （ n1 n2 n3 -- q） 算出 n1*n2/n3（计算中维持 48 位）。 */MOD（ n1 n2 n3 -- r q ）算出 n1*n2/n3 最后的余数及商。

4-2-4 记忆指令：

HERE （ -- a） 变量，内容为词典边界的最高地址。 PAD （ -- a） 变量，内容为字符串暂存区的地址。 TIB （ -- a） 变量，内容为终端机暂存区的地址。 @EXECUTE（ a -- ）由 a 中取出执行码栏地址，然后跳到该处去执行。 CMOVE （ a1 a2 n -- ） 从 a1 搬 n 个 24 位字码到 a2 区。FILL （ a n c -- ） 将 24 位数值 c 填入 a 起的 n 字码区。>CHAR （ c -- c ）将不能印出的字符都改为底线符号_，如此为防止打印机在纸张上印出乱码。

4-2-5 压缩字符串及解压指令：

PACK$（ b n a -- a） 将在 b 地址的 n 个字符压缩到 a 区，每三个 8 位字符压缩到一个 24 位的字码中，最后补 0。 UNPACK$（ a b -- b） 将在 a 地址中的一个压缩后的字符串解压，并放在 b 区。

例如 7 个字符的字符串「UNPACK$」，未压缩时是由 8 个 24 位之字码组成，压缩后成为 3 个字码，内容如下：

Bit 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

字码 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 0 1 1 1 0

7 'U' 'N'

字码 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 1

'P' 'A' 'C'

字码 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0

'K' '$' 0

字符串压缩后不但减少了记忆存储的空间，也显著地加速了 Forth 系统查字典的速度。在比对字符串时，可一次比较 3 个字符。

4-2-6 数值印出指令：

p24 的数值都是以 24 位方式存在内存及缓存器中的，只有在将数字印出时才须要将其转成为 ASCII 字符串，每字符 8 位。转换时用 BASE 里所存数值作转换的基数。所以可以

很方便地把数值随时转成以十进制，十六进制，或其它基数的字符串形式印出。

数字打印的指令常用的有以下几个：. （ n-- ） 将 n 转换成字符串显示在终端机上。U. （ u-- ） 将自然数 u 显示在终端机上。 .R （ n1 n2 -- ）将 n1 显示在 n2 字段中，靠右对齐。U.R （ u n -- ） 将自然数 u 显示在 n 字段中，靠右对齐。？（ a-- ） 将 a 地址中的数值取出，显示在终端机上。 HEX （ -- ） 将 BASE 基数改为 16，以后数字转换即十六进制。DECIMAL （ -- ） 将 BASE 基数改为 10，以后数字转换即十进制。

为要完成以上数值打印的功能，eForth 须要有一系列的支持指令。使用者也可以自行利用这些指令，设计出更适合特殊不同场合应用的打印指令。 str（ n -- b u ） 将数值 n 转换成字符串储存在 b 区，字符串长度是 u。 DIGIT（ u -- c ） 将一位数的值 u 转换成其对应的 ASCII 数码 c。 EXTRACT（ n1 base -- n2 c ）以基数 BASE 去除 n1，余数转成一位的数码 c，留下商数 n2，以便继续数值转换之程序。 <#（ -- ） 准备转换，设定 PAD 存所得字符串，以 HLD 为指针。 HOLD（ c -- ） 将一位数码 c 写到 HLD 所指的位置，并将 HLD 减 1，准备接受下一数码。# （ u1 -- u2 ） 由 u1 中转出一位数码写到 PAD 中，商是 u2。#S（ u --o ）将 u 转出所有数码字符串写到 PAD 中，最后商是 0。 SIGN（ n -- ） 若 n 小于零，则在数码字符串前加负号。 #>（ u1 -- b u2 ）丢掉 u1，留下数码字符串的地址 b 及其长度 u2。

4-2-7 数值输入指令：

由终端机输入的数字都是 ASCII 数码字符串，须要转换成对应的 24 位数值才能放在堆栈上运用，或是存在内存中。将数码字符串转换成数值的指令是: NUMBER?（ a -- n -1 | a 0 ） 将在地址 a 的数码字符串转换成 24 位的数值 n 及-1 的旗号。若转换不成功，(字符串中有非数字码的符号)，则保留地址 a 并留下 0 的旗号。 DIGITS（ c base -- u f ） 将一位数码字符 c 按基数 BASE 转为数值 u 及 -1，若 c 不是数码，则保留 c 并留下 0 的旗号。

4-2-8 其它打印指令：

除了打印数值字符串外，eForth 也提供一套相当完整的指令，让使用者很方便地在终端机上显示各样文字数据，排列成整齐美观的文件，这套指令是： SPACE （ -- ） 输出一个空格。 SPACES（ n -- ） 输出 n 个空格。 CR （ -- ） 换行。 CHARS （ n c -- ）输出 n 个 ASCII 码 c 所对应的字符。 TYPE （ a u -- ）从地址 a 取 u 个 ASCII 码字符输出。 Do$ （ -- a ） 将存编码在程序中的字符串解压缩到 TEXT 区，并将 TEXT 区的地址 a 放上堆栈。 $"| （ -- a ） 将编码在此指令后的字符串解压，存到 a 区。 ."| （ -- ） 将编码在此指令后的字符串解压，并印出。

4-3 译码器相关指令

4-3-1 解读指令：

解读指令是 eForth 系统由输入的文字序列中，分解出个别字符串的机制。一般文字序列中有许多字符串，以空格或某些不能印出的控制字符（ ASCII 码 0 到 31 之间 ）来分割的。字符串分割出来后，才交给译码器【Interpreter】处理。 PARSE （ c -- b u ） 在文字输入区（ TIB，Text Input Buffer ）中取出下一个以字符 c 来分割的字符串。找到此字符串后，将

其所在地址 b 及长度 u 留在堆栈上。 (parse) （ b u c -- b u delta ） 在从 b 开始长度为 u 的文字序列中取出一个用字符 c 所分割的字符串。找到字符串后，将字符串长度 delta 取代 c。如果 c 码是 32（空格），则还会将 b 区中起头的空格都略过。此时的 b，将是剩下起头非空格后的字符串地址。 TOKEN （ -- a ） 在文字输入区取出用空格字码分割的下一个字符串。字符串存在 a+1，其长度存在 a。 WORD （ c -- a ）在文字输入区用字符 c 取出下一个字符串。字符串存在 a+1，其长度存在 a。

4-3-2 查字典指令：

eForth 的指令，都是存在内存中，一个特定叫做字典的区域。每个指令中都有三栏：连接栏、名称栏和执行码栏。连接栏为一字码长度，其中含一个地址，指到上一个指令的名称栏。名称栏的长度不定，存有指令的名称及其长度，以压缩字符串的形式储存。执行码栏内，则存有可执行的机器码及相关数据。图 09 易符字典结构

这样所有的指令在字典里就都连接成一长串的列表。若连接栏中存的是 0，表示列表的尽头。译码器可在这个字典里，依输入的名称寻找到要执行的指令。查字典时遇到连接栏里是 0，就知道已查到了字典的尽头，都还没找到所要的指令，表示名称错误或尚未定义。而查字典的最关键指令就是 FIND。 FIND （ a va -- xt na | a o ）指令名称字符串在地址 a，va 为搜寻起点的指标，其内容为最后定义而被纳入字典之指令的名称栏地址。由 va 开始查字典，若找到的指令其名称与所给之字符串相同，留下该指令执行码的地址 xt 及名称栏的地址 na。若查完字典仍找不到，则保留原字符串地址 a，并给一个 0 的旗号表失败。 NAME? （ a -- xt na | a o ）将系统变量 CONTEXT 作为字典搜寻起点的指针，从其中可取出字典最后定义的名称栏地址，与所给在地址 a 的名称字符串相比，以搜查字典中该指令的执行码地址 xt 及名称栏的地址 na。 SAME? （ al a2 u -- a1 a2 f ）比较在 a1 区和在 a2 区两个长度为 u 的字符串，若两个字符串完全相同，则 f=0，若有任何不等的字码，f≠0。 NAME> （ na -- xa ）由指令的名称栏地址 na 换算出执行码栏的地址 xa。

4-3-3 输入文字序列的指令：

KEY 和 EMIT 是终端机输入输出的基本指令。由此二指令衍生出许多高阶的输出输入指令，支持 eForth 系统的人机界面。 QUERY（--）从终端机接受最多达 80 个字符的字符串，并将此字符串存入终端机输入区（TIB）。 ACCEPT (b u1 -- b u2) 从终端机接受 u1 个字符的字符串，并将其存入 b 区。若使用者接收到 u1 字符前按了 RET 键，所实际接受到的字符数是 u2。 EXPECT (b u --) 从终端机接受 u 个字符，存至 b 区，实际所接受到字符串的长度则存于 SPAN 中。 ^H (bot eot cur -- bot eot cur) 执行退格清除键【Backspace】的功能。若 cur=bot 则三参数保持不变，若 cur>bot，则将 cur 减 1，同时输出三个 ASCII 字码 Backspace、Space、Backspace，以清除终端机上才输入的字符。 TAP (bot eot cur – bot eot cur) 接受一个字符 c，存入记忆区，并将 cur 加 1。 KTAP (bot eot cur – bot eot cur) 接受一个字符 c，存入记忆并处理 CR、BS、及其它 ASCII 控制符号。

4-3-4 侦错指令

当 eForth 执行一连串指令，若在半途发生错误，必须立即反应，停止后续的动作，将系统重设到正常的执行状态，等候使用者输入新的指令。最关键的侦错指令即是 ABORT。ABORT (--)停止现在执行的指令，跳到 QUIT 重新开动 INTERPRETER 译码器等候输入下一轮的新指令。 ERROR (a --)a 指到目前所执行指令的名称字符串，将此字符串印出并附加「?」字符，再经由 ABORT 跳回 QUIT。 abort" (f --)若 f≠0，则印出编码在此指令后的字符串，并由 ABORT 跳回 QUIT。若 f=0，则执行后续的指令。

4-3-5 译码器

译码器是 eForth 操作系统的核心。它由终端机接受使用者键入的文字序列，将其中的字符串分解出来，然后依序处理这些字符串。有些字符串是指令，在字典中查到后即去执行。有些字符串是数字，即转换成数值放到堆栈上。全部文字序列如此处理完毕，就等下一轮再接受新的文字序列来处理，如此周而复始，直到关机。执行指令或处理字符串时若有错误，则重新开动译码器。 QUIT (--)译码器。设定终端机输入区后进入回路，接受文字序列并处理序列中的指令。 EVAL (--)本身也是一个回路，在已经输入的文字序列中分解字符串。然后将字符串交给 'EVAL 中存的指令处理。当文字序列处理完毕，则由 .OK 指令印出「OK」字样。解碼时，'EVAL 中存的是 $INTERPRET。编码时，则 'EVAL 中是 $COMPILE。 .OK (--)在解碼操作时，印出「OK」字样，显示一组文字序列已经处理完。若在编码操作时，则不印，并继续编码工作。[(--)将 $INTERPRET 指令的执行码栏地址存入 'EVAL 变量中，如此系统即由译码器来处理指令字符串。 $INTERPRET (a --)根据存在地址 a 的名称字符串查字典，若找到与名称相符的指令，即执行该指令。若字典中没有此指令，则将字符串当作数目字转换成一个数值，留在堆栈上。若字符串不是数目字，则交由 ERROR 处理此错误情况。

4-4 编码器(COMPILER)相关指令

4-4-1 编码器指令

eForth 中的译码器和编码器是一对孪生兄弟，他们用相同的 QUIT 机制，接受终端机输入的文字序列。用相同的解读指令 PARSE 分解出指令字符串，再由指令字符串去查字典。不同的是当译码器找到了指令时即去执行，而编码器找到指令时，不会直接执行，而是将指令的执行码栏地址，编入字典边界顶端所正在定义的新指令中。新指令建造完成后，在被点名执行时，编入的执行码就会被依序执行了。

在 EVAL 的回路中有一段是 'EVAL @EXECUTE,即将 'EVAL 变量中所存的地址叫出来执行。译码器工作时，'EVAL 中所存的是 $INTERPRET 指令的执行码栏地址，若是编码器在工作，'EVAL 中存的则是 $COMPILE 的执行码栏地址。改变 'EVAL 中所存的地址，就会让 eForth 或做编码的工作，或做译码的工作。

$COMPILE (a --)

根据 a 区中的名称字符串查字典。若找到与名称字符串相符的指令，即将此指令的执行码地址，编入字典边界顶端正在定义的新指令中。但假若此指令名称栏中最高位($800000)

不是零，这个指令即属立即执行【Immediate】类，这类指令会随即被执行而不被编码。若字符串不是一个指令，则将它当作数值转换，而将数值以 Literal 形式编入新指令。若不是指令，也不是数字，则跳到 ERROR，处理此错误情况。

LITERAL(n --) 是数码指令，在新指令中编入 LIT 机器码，然后接着编入数码。当新指令执行到 LIT 时，会将其后编入的数码取出放在堆栈上。这是 eForth 程序中产生数值，并将它放上堆栈的唯一方法。

编码器最核心的工作是由 $COMPILE 完成的。所做编码的工作，即是在字典上建造新的指令。这是相当复杂的工作，须要一系列下述编码器的辅助指令，才能圆满达成完整的编码工作。 $,n (a --)在字典边界的顶端建造一个新指令的连接栏和名称栏。建名称栏时，将在地址 a 的名称字符串压缩后塞入名称栏中。?UNIQNE (a – a) 用地址 a 的字符串查字典，看其中是否已有相同名称的指令。若有，则印出「Redef」的警告讯号。 $," (--)在新指令的定义中编入一个字符串，此字符串是以 ” 号结束的。 LITERAL (n --)在新指令中编入一个数码指令 LIT，然后编入数码 n，将来在程序执行到此指令时，n 会被取出放在堆栈上。 COMPILE (--)将程序中此指令后面所编入的指令取出，编入现在定义的新指令中。 [COMPILE] (--)此属立即执行指令。它会寻找下一字符串文字所代表的指令，并将其地址以 CALL 的指令形式，编入字典边界顶端正定义的新指令中。， (n --)此逗点指令，乃是将堆栈上的一个数值 n 编入字典边界顶端正在定义的新指令中。ALLOT (n --)将字典指针 CP 增加 n，给正定义的新指令留下 n 个字码的空间。 ' (-- a)此单引号指令，寻找下一字符串所代表的指令，找到后将其执行码栏地址留在堆栈。找不到与名称相符的指令，则会显示错误信号。 OVERT (--)编造新指令成功后，将这新指令的名称栏加入字典的串连链列表中。以后查字典，即可在此串连链中找到这个新指令。： (--)开始定义一个新指令。用下一字符串做为新指令的名称，建造一个连接栏及名称栏。； (--)结束一个新指令的定义。在此新指令的机器码最后，追加编入一个 RET 的机器码，并用 OVERT 把这新指令加入字典的串连链。] (--)将 $COMPILE 的执行码栏地址存入 'EVAL 中。以后即让此编码器处理所输入文字序列中的指令字符串。

4-4-2 结构化编码指令

eForth 的指令中执行码栏中，所编入的是 P24 的机器码。但在高阶定义的指令中，大部份都是以机器码 CALL 来呼叫子程序，所以一般皆称这样的结构作子程序表列【List of Subroutine Calls】，或子程序串链【Subroutine Threaded】。这种结构很容易插入 P24 的机器码，和子程序的呼叫码一起混编。所以在一个定义中的程序，可以是高阶的编码器和低阶的组合器同时作用所产生最佳化的结果。

eForth 是严格的结构化语言，备有一套完整的结构化编码指令。我们可以在新指令的定义中，建造各种分叉及回路的程序结构，使得程序的写作及侦错都比较容易。它能建造的程序结构及相关的结构化指令，如下：分叉结构： (f --) IF …… THEN 若 f≠0，IF 即执行后续的指令。若 f=0，即忽略后续的指令，跳到 THEN，接着执行 THEN 以后的指令。 (f --)

IF …… ELSE …… THEN 若 f≠0，IF 即执行后续的指令到 ELSE 为止，然后执行 THEN 以后的指令。若 f=0，IF 即忽略后续的指令，跳到 ELSE，执行 ELSE 以后的指令，并且也继续执行 THEN 以后的指令。无条件回路： BEGIN …… AGAIN 进入此回路后，重复执行 BEGIN 及 AGAIN 中间全部的指令。有条件回路： BEGIN …… (f --) UNTIL 进入此回路后，执行完 BEGIN 及 UNTIL 中间全部的指令，UNTIL 会检验堆栈上的数值 f。若 f=0，则跳到 BEGIN 重复回路。若 f≠0，则跳出回路，继续执行 UNTIL 后的指令。 BEGIN …… (f --) WHILE …… REPAT 进入此回路后，执行完 BEGIN 及 WHILE 中所有指令，WHILE 会检验堆栈上的数值 f。若 f≠0，即执行 WHILE 到 REPEAT 中间的指令，然后跳回 BEGIN，继续此回路。若 f=0，WHILE 即忽略后续指令跳出回路，接着执行 REPEAT 以后的指令。有限回路： (n --) FOR …… NEXT FOR 将回路指针 n 从数据堆栈取出，放到返回堆栈上，接着进入回路，执行 FOR 与 NEXT 间的指令。NEXT 会检验回路指标，若指标是 0，就跳出回路，执行 NEXT 后的指令。若指标不是 0，就将其减 1，然后跳回 FOR 处重复执行此回路中的指令。

在编造的新指令中，我们还可以加入一些其它的结构，例如前述用 LITERAL 建造的数码结构，乃是由机器码 LIT 加上后面编入的一个数值，所形成的简单结构。字符串的结构则较为复杂，是由一个处理字符串的指令码，附加后面编入的一个压缩字符串所形成。LITERAL (n --)在新指令中建造一个数码结构。由 LIT 机器码加上数值码 n 组成。执行时，LIT 即将该数码取出放上堆栈。." (--)在新指令中建造一个字符串结构，执行时将此字符串印出。字符串结构是以．"| 指令开始，加上压缩的字符串。 $" (-- a)在新指令中建造一个字符串结构。执行时将此压缩字符串解开，放在 TEXT 区。并将 TEXT 的地址 a 放上堆栈。ABORT" (f --)在新指令的定义中，建造一个印出字符串的指令结构。在执行到该指令时，若堆栈上的数值 f≠0，即印出字符串并跳到 ABORT 重新起动系统。

4-4-3 其它词类指令的建造指令

eForth 中最常用的指令是用冒号：来建造的，其执行码栏内都是 CALL 或其它机器码。这类指令通称冒号指令，是 Forth 的基础。除了冒号指令外，eForth 还提供几种不同词类的建造指令，用来构筑各种不同词类的指令。

eForth 支持的词类除冒号外，尚有常数、变量、矩阵、机器码及其它词类构筑等等。CODE (--)建造机器码定义的低阶指令。在此 eForth 系统中，与冒号指令相同。不同的地方，仅在 CODE 指令执行后，系统还会留在译码器中，而不是进入专门建造冒号指令所用的编码器中。VARIABLE (--) 建造一个变量指令，其初始值设为 0。执行该变量指令时，会将它的地址放上堆栈，以便后续读取或写入数值的动作。 CONSTANT (n --) 建造一个常数指令，取堆栈上的 n 为其值。执行该常数指令时，会将此 n 放上堆栈。 CREATE (--)建造一个矩阵指令，执行该矩阵指令时会将矩阵起始的地址放上堆栈。至于矩阵区域的大小，要用 ALLOT 来界定。

DOES 是一个特别的指令，被用来以下列的方式建造一个新词类的构筑指令：：名称

CREATE …… DOES …… ;

这里的「名称」是新词类构筑指令的名字。这个构筑指令是用来定义许多属此同一词类的新指令。这些新指令所用嵌入数据的编码方式乃用 CREATE 和 DOES 间的指令来规定。这些新指令在执行时，嵌入数据的地址会先放上堆栈，然后接着执行在 DOES 和；之间的指令。换言之，CREATE 和 DOES 中间的指令是这词类指令的编码器，而在 DOES 与；间的指令是这词类指令的译码器。

用 CREATE …… DOES 的结构，能让使用者随心所欲地自行规定或创造其它新的词类，这是使用 Forth 撰写程序的最高境界。可以将很复杂的专业解法，经由词类指令得到最简单有效的表达。余下的一些指令是： .((--)印出括号内的文字。这在原始程序文件中很有用，在程序加载过程中，可用以显示指令集的编码进程。 ((--)忽视括号内的文字，在原始程序中，作为批注之用。 \ (--)忽视在 TIB 中已输入的文字序列。 IMMEDIATE (--) 在新定义指令的名称栏内、设定第一字码的最高位($800000) 为 1，使此指令变为「立即执行」的指令。就算在编码时，该指令会直接被执行，而是不被编入。

4-5 工具指令

4-5-1 程序发展工具指令

eForth 中有了译码器和编码器，就是一个很完整的操作系统了。但要用它来发展应用的软件，还不是很方便的。我们再加上几个工具指令，帮助做侦错及系统检查的工作。这样就形成了一个很有效的程序发展环境，可以很快速地写作，并且修改一些复杂的应用程序。DUMP (a u --) 将内存由地址 a 开始的 u 个字码显示到终端机上。用这个指令可以检查全部内存中所有的数据和执行码。若输入输出装置也都挂在记忆汇排上，它也可以用来检查所有的输入输出装置的状态。 .S (--)将堆栈上所有的数据显示到终端机上。Forth 处理的数据大都经过堆栈。用 .S 来检查堆栈上的数据，很快可以知道程序的进行是否正确。 SEE (--) 反编码指令。将其后的字符串当作一个指令名称去查字典，若有相符的指令，将其执行码栏中当初所定义的内容显示在终端机上。遇到 CALL 机器码，会将所呼叫的指令名称印出。这 SEE 指令可用来检查刚定义的指令，是否正确地依所预期方式编码成功。 WORDS (--) 将字典中所有指令的名称显示到终端机上。检查档案中的指令集是否都成功地被编码加载系统，或可了解所完成的进度。 READ (--)从终端机直接下载档案文字到 p24，存在 PAD 开始的记忆区。当接收到 ESC 控制字符时就结束下载。 OK (--)解读由 READ 下载存在 PAD 的文字序列。依序执行译码或编码的动作，建立、设定、或执行不同的应用系统。 SEND (a u --)将从地址 a 开始长度为 u 的数据，经由终端机，从 p24 上传到主计算机中储存。若包括了整个系统含字典的结构，上传的数据即可直接烧入 EPROM 建成嵌入式系统。 FORGET (--)在字典搜寻与下一字符串名称相符的指令。找到后将该指令及其后所有定义的指令，从字典的串连链上切除。这时字典即恢复到该指令未被编入前的状态。这个清除字典的指令，在反复试验一段应用程序时十分有用，不需浪费 p24 的记忆空间。 DIAGNOSE (--)这一个指令会执行到大部份的机器码，用以检验所有的机器码是否都被 p24 系统正确地执行，因而

能确认 CPU 是正常工作了。

4-5-2 组合器指令

我们在前述，组合器的设计和制造章节中，已看到如何在 PC 上设计组合器，用来组合 P24 的整个系统。当这个系统在 P24 上动起来之后，我们的编码器能用 CALL 的形式编组新的指令。要充分发挥 P24 计算机的功能，我们在译码器和编码器之外，还需要一个在 P24 上跑的组合器。因为在编码器上，我原已经建好了许多构筑结构化程序的指令。所以现在只要加上能够把四个机器码塞入一个字码的工具，就可拥有一个完整的组合器了。为了做这件事，我们将 OK24.F 中一些指令搬进 P24 系统： mask (-- mask)此为系统放置四组面罩之矩阵。选择面罩之一，可用以将一个 6 位的机器码塞入目前编码的 24 位字码中。 ALIGN (--) 准备将机器码编进下一个空的 24 位字码中。 ,W (n --)将数据堆栈上的 n，编入正在编码的 24 位字码里。 ,I (n --)将 6 位的机器码 n 塞入现在编码的 24 位字码中，它会依序编四个机器码，然后会将新码编入下一个 24 位字码中。 Anew (--)若目前编码的 24 位字码还没有装完，补满 NOP 以便开始编下一 24 位的字码。 DoInst (--)将程序中此指令后面原编入的机器码取出，编入现在编码的 24 位字码中。这个指令是组合器的核心。

所有在 P24 的组合器用的机器码指令、都用小写的名字，这样不会和常用的 Forth 指令相混淆了。这一套集合指令是： ret ldp ld stp st com shl shr mul xor and div add pop lda dup push sta nop drop

4-5-3 宏指令

为了建造最精简又快速的 P24 系统，在 KERNEL 中我们定了许多宏指令，这些指令只在建造 P24 系统的 PC 上有用，在建好的系统用的是 P24 机器码，就没有用了。为了在 P24 也能使用这些指令，我们必须另建一套给译码器和编码器使用。这些指令是： EXIT EXECUTE ! @ R> R@ >R SWAP OVER 2DROP + NOT NEGATE 1- 1+ BL +!

4-5-4 启动指令

eForth 开机启动的指令是 COLD。开机时 P24 由地址 0 开始执行机器码。这段机器码会先将系统变量由 ROM 区抄到 RAM 区，然后就跳入 COLD 指令。COLD 做的事也很简单，先执行 DIAGNOSE 确定 CPU 工作正常，然后印出「P24 V2.02」告诉使用人 CPU 已开始工作，然后进入 QUIT 等候使用人由终端机上输入指令文字。

4-6 Forth 与 LISP 的比较

这样做下来，就是一个完整的操作系统了。虽然我花了很多篇辐，将所有的指令一一解说，但你可看出，一个操作系统并不如大家想象的那么困难、麻烦、或是深奥不可测度。

Forth 和 LISP 有很多类似的地方。它们的基本假设都是：任何计算机能做的事，都可以用表列【LIST】的方式来表达。所以所有的程序都是一系列的表列。在 eForth 中，这些表列主要即最有效率的一系列 CALL 机器码。

LISP 麻烦的地方在于它是从上到下【Top Down】的写法。没有预先定义好的指令也可以放在表列中，到执行时才看该指令是否定义好。而且还要用许多括号来界定表列的范围。

Forth 则是由下到上【Bottom Up】的写法，所有指令在用时必须先被定义好。这样的编码工作就很容易又有效率，产生的系统也精简快速。

在这几十页有限的叙述中，当然也不易将整个系统作很详密的解说。但当你了解到整个系统的架构后，不清楚的地方，可以研究所附之原始程序代码，慢慢地体会。由于原程序代码只有这么大，你终究会完全掌控这个 p24 系统的，需要的只是一份执着和耐心。

正是：『我要在你的命令中自乐，这命令素来是我爱的。我看万事尽都有限，惟有你的命令、极其宽广。』注：这里的命令就是指令。（圣经．诗篇第一百十九篇）

第五章 P24 仿真器的设计与建造

我们自己设计制造一套 P24 计算机的仿真器，忠实地仿真 CPU 在每一时钟周期上的状况。因而可以在 PC 上非常准确地模拟 CPU、以致整个操作系统的运作。

在自行设计 P24 计算机的过程中，我使用 Xilinx 公司的 Vertex VCX1000E FPGA。该公司提供了一套芯片设计的系统 Foundation 3.12,允许使用者以线路图、VHDL、或 Verilog 的不同方式来设计。并附有相当完整的辅助工具，可以做布线、模拟、及测试的相关工作。但是，其仿真器则主要还是展示产生的信号波形，而信号的选择及设定都不是很方便，因此做验证及测试的工作，相当麻烦并且费时。

这类仿真器对 CPU 的低阶验证，在闸门信号、和机器码仿真上相当有用。但对比较复杂的程序逻辑，验证就很花时间。为了要节省验证的工作时间，我们自己设计制造一套 P24 计算机的仿真器，忠实地仿真 CPU 在每一时钟周期上的状况。因而可以在 PC 上非常准确地模拟 CPU、以致整个操作系统的运作。这个仿真器大大缩简了 CPU 和操作系统的设计、制作的时间和工作量。而成了整个计划真正成功的关键工具。

5-1 P24 的时效仿真器（CYCLE-BASED SIMULATOR）

P24 是一个同步的逻辑线路。用单独的一个主时钟，提供单一的同步控制信号。所有缓存器和堆栈，都是利用时钟信号上升边缘，将新的数据锁住。

一旦新的数据被锁住，计算的工作和控制信号即经过闸门、MUX、及逻辑线路来动作，或产生新的数据。新的数据在下一个时钟信号的上升边缘又被锁住。如此周而复始，P24 即可正常运作，完成一个程序中规定的所有工作。

这个仿真器基本的架构就如下图所示，主要的部份包含了两个大的记忆区：FROM 区和 TO 区。FROM 区里面储存的，是目前所有缓存器和堆栈中的数据。而 TO 区里面，则储存在本周期内各单元操作计算所产生的结果。 图 10 仿真器工作模式

计算及控制逻辑线路，拿 FROM 区中缓存器及堆栈的数据来计算与处理，将结果写入 TO 区所对应缓存器及堆栈的地址。当时钟信号上升边缘发生时，仿真器即将 TO 区中的所有数据抄入 FROM 区所对应的地址，这样就完成了一个时钟周期所需要做的工作。而 P24 的缓存器有： P Program Counter, 程序计数器，存下一个指令的地址。 I Intruction Register，指令缓存器。存现在要执行的指令字码 I1 现在执行字码中第一个机器码 I2 现在执行字码中第二个机器码 I3 现在执行字码中第三个机器码 I4 现在执行字码中第四个机器码 T ALU 缓存器，所有数据计算结果存放于此。 A 地址缓存器，记忆区读写地址存于此。 RP Return Stack 的地址指针。 SP Data Stack 的地址指针。 RSTACK Return Stack 返回堆栈。 SSTACK Data Stack 数据堆栈。 CLOCK 是仿真器中的一个变量，其中的数值控制时钟的周期。每经一个时钟周期，CLOCK 中变量值自动加 1，（+1）。P24 计算机时钟和指令执行的顺序是：图 11 仿真器工作时序

CLOCK 的最低 3 位是数 Slot 的顺序，由 0 到 4。CLOCK 到 4 时被改为 7,在下一时钟

信号到达时就进位到第四位 Bit 3. 所以 CLOCK 的值除 8，就是执行的字码总数。

5-2 仿真器的基本动作

这仿真器是用 Win32Forth 写作成的工具。以下是一组相关的 Forth 指令，指挥仿真器的基本动作。 CYCLE 模拟时钟上升边缘的动作，即是将 TO 区的数据抄入 FROM 区。 NEXT 将 CLOCK 的最低 3 位改为 7，这样在下一个周期时，会由下一字码读取一个新的字码去执行。如此就跳过了目前字码中余下的机器码。 CONTINUE 读取程序中下一个字码到 I 缓存器中，将四个机器码读出，并分别写入 I1,I2,I3 及 I4 缓存器中。 RPUSH 把一字码压进返回堆栈中。 SPUSH 把一字码压进数据堆栈中。 RPOPP 将返回堆栈最上一字码弹出。 SPOPP 将数据堆栈最上一字码弹出。 EXECUTE 依照机器码的数值去执行 CPU 要做的动作。仿真器执行机器码时，是将缓存器及堆栈中的数据由 FROM 区中读出，加以适当的处理后，存入相当的 TO 区地址中。每个机器码的工作都可以类似如此模拟。GET 由 Windows 的键盘等下一个按键，将得到的 8 位数据交给 Forth OS 中的 KEY 指令 PUT 将一个 8 位数据送到 Windows 的显示屏幕上展示。仿真 Forth OS 中 EMIT 指令的动作。

机器码各自的动作在此不再详述，唯有以下两个机器码、是为了要模拟 Forth OS 另外加的： P24 机器码中最麻烦的指令是 MUL 及 DIV。它们的动作特别加以详细说明，如下：

MUL 步进乘法。先测试 A(0)位是否为 1,若是 1 时，将数据堆栈最上层的数值（SSTACK 取得）与 T 中数值相加。其结果向右移一位存入 T。其结果的最低位则与 A 中的数值一并向右移一位存入 A 中。若 A(0)位是 0 时，将 T 向右移一位存入 T 中，T(0）则和 A 一并向右移一位存入 A 中。如此就完成了乘法的一步。此机器码重复执行 24 次，即将 SSTACK 最上层数值与 A 相乘。48 位的乘积则留在 T 与 A 缓存器中。

DIV 步进除法。将 SSTACK 最上层的数值与 T 相加，若 Carry 位为 1, 即将和数向左移一位存入 T 中。T(0)位则以 A(23)补足。A 也向左移一位，Carry 则填入 A(0)。若 SSTACK 加 T 的 Carry 是 0 则丢掉和数，只将 T 左移一位。而 T(0)用 A(23)补足，A 亦左移一位，A(0)用 0 补足。这个机器码若重复执行 25 次，即可将 T 与 A 中的 48 位数值被 SSTACK 最上层数值的负值除完。商数在 A 中，余数在 T 中。

5-3 仿真器的使用

使用仿真器之前，先要在 Win32Forth 中用 P24 的组合器将 P24 程序组合完成，存在 RAM 区中。然后再加载 P24 仿真器。正确的操作程序是： 1） 在 Windows 下，启动 Win32Forth 2） 在 Win32Forth 中输入 FLOAD META24 (载入 META24.F) 3） 在 Win32Forth 中输入 FLOAD SIM24 （载入 SIM24.F）

然后即可由键盘输入以下指令操作仿真器：HELP 打印出仿真器的操作指令及功能。S 打印出 P24 所有缓存器及堆栈的内容。 C 执行一周期的指令。 D 打印出下面将要执行的 8 个字码。 addr M 打印出由所指定 addr 开始的 128 个字码。 addr P 设定程序开始执行的地址为 addr。 addr G 开始执行程序，当执行到 addr 时停止。 RUN 开始执行程序，每按一键执行一步，并且展示所有的缓存器和堆栈。若按了 ESC 才跳出回路，回到仿真器系统，

等下一个指令。

要仿真执行全套 Forth 操作系统程序，可键入下列指令： 3000 G

3000 这不是在 p24 Forth 系统中有效地址范围，0 到 2047，之内的地址。正常仿真器的执行，不会经过这个地址，所以仿真器即在无限的回路中接受 Forth 指令。仿真器的功能和实际 P24 计算机的功能完全相同。键入上述指令后，Win32Forth 的窗口中即展示： P24 Forth V1.02

按 CR 键，FORTH 即回应 OK。

此仿真器会十分忠实地在 PC 上反映 P24 计算机的实际动作。我利用这个仿真器，帮助了 P24 计算机的实际设计工作，尤其是我借着它将 P24 Forth 的操作系统全盘测试验证。这是十分强有力的一套计算机硬件及软件设计的辅助工具，使用者可以此用仿真器发展及测试实际的各种应用程序，不必等到硬件做成功后、才来发展与测试其上的相关软件程序。

庖丁解牛

庖丁解牛是庄子里一则很有意思的故事。这位大厨师宰牛的技术到了极致，一把刀用了十九年，宰了数千条牛，但刀刃锋利如故。牛虽然大，骨骼坚挺，但他能见到骨与骨之间的缝隙。骨与骨有间、而刀刃无厚，所以宰牛之时，以无厚而入有间，恢恢呼其游刃必有余地也。

我们在做软件的设计制造时，总是在想用最新最好的工具。但这些工具随着计算机内存的增加而日益庞大。而且因为里面加装了越来越多、无关紧要的功能，而越跑越慢。我们设计的线路也越来越庞大，而需要更大、更快的计算机，也要更新、更复杂的软件工具。每隔一段时间就必须换刀了，因原来的工具不管用了。 Forth 是一种精悍短小的软件工具，所以十分容易移植到任何计算机系统中去。这是一把刀刃无厚、十分锋利的刀。我用它搞了二十多年的软件，发现什么问题都很容易解决。现在用它来看计算机设计的问题，觉得也是很容易解决的。

短短不到一百页的易符真经里，用实际的程序将计算机整体的设计和制造做了完整的介绍。学的人可以很大胆的投入 SOC 系统的开发。我引用了庄子庖丁解牛的故事来勉励我的徒弟们：

『良庖岁更刀，割也。族庖月更刀，折也。今臣之刀十九年矣，所解数千牛矣，而刀刃若新于硎。彼节者有间，而刀刃者无厚，以无厚入有间，恢恢乎其于游刃必有余地矣。是以十九年而刀刃若新发于硎。』这本易符真经是一把无厚的刀。我用它做什么工作都是无往不利，希望每个人领了这把刀去，都能登峰造极，为中国人在计算机界另创出一片新天地。现在是廿一世纪了，还有这样神妙的事。你不相信？来试试看吧！

正是：『十年磨一剑，霜刃未曾试。今日把示君，谁有不平事？』（唐．贾岛．剑客）