tanulmányok 91/1979

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MIKROPROCESSZOROK ÉS MIKROPROCESSZOR-BÁZISU RENDSZEREK ARCHITEKTURÁJA ÉS STRUKTÚRÁJA.

T é l i i s k o l a

Szentendre, 1978. február 13-18.

*

ARCHITEKTURE AND STRUCTURE OF MICROPROCESSORS AND MICROPROCESSOR-BASED SYSTEMS.

W i n t e r S c h o o l

*

Előadások

Proceeding

Доклады

Архитектура и структура микропроцессоров и микропроцессорные системы

З и м н я я Ш к о л а

A kiadásért felelős:

DR VÁMOS TIBOR

Szerkesztette:

DÁVID GÁBOR

ISBN 963 311 078 5

ISSN 0324-2951

Készült a KSH Nemzetközi Számitástechnikai
Oktató és Tájékoztató Központ Reprográfiai Üzemében

79/093

A konferenciát a "Számitástechnika tudományos kérdései" c. többoldalu akadémia együttmüködés keretében rendezte a 10. MCS.

Конференция была проведена в рамках многостороннего сотрудничества академий социалистических стран по проблеме "Научные вопросы вычислительной техники"

PM.10.

Conference was held in the frame of the multilateral cooperation of the academies of sciences of the socialist countries on Computer Sciences, WG. 10.

TARTALOMJEGYZÉK

SESSION 1: ARCHITECTURE

Сессия 1: Архитектура

THIS MAKES ME THINK ...
(UNDER THE PRETEXT OF A WELCOMING ADDRESS)

F. VAJDA

Central Research Institute for Physics of the Hungarian Academy of Sciences, Budapest, Hungary

Having the unusual opportunity of opening this winter school, please allow me to introduce a few thoughts of mine regarding its topics.

The *main purpose* of our winter school is to facilitate the international flow of information among the participating socialist countries in the field of

- microprogramming and
- microprocessing (in other words: microprocessor applications.)

Our scope of interest includes in particular

- theory
- research
- design
- development
- application

relating to

- microprogramming
- microprocessor systems
- distributed computing and multimicroprocessor system
- integrated hardware-firmware-software design.

The *main components* of up-to-date system design are:

- hardware
- software
- formware
- brainware (We are particularly strong in this field!)

We are to have a separate section of the school devoted to application aids (developing tools and methods); because there is no *system development* without a *development system*. We have to change today's situation when microprocessor programming is like printing before Gutenberg.

First of all a few definitions are shown to have a "*common vocabulary*".

A *microprocessor* is the central processing unit of a small computer implemented on one or a few LSI chips.
The computer based on a microprocessor is a *microcomputer*.

Microprogramming is the design of the control function of a processing system as a sequence of control signals which are organized into words (called *microinstructions*) and stored in a *control memory*.

A system is *microprogrammed* if this control function is implemented in this way and *microprogrammable* if the control function can be changed by the user.

An *emulator* is a microprogrammed system which copies another system.

Considering *components' development*, Texas Instruments VLSI (Very Large Scale Integration) schedule is given in Fig.1.

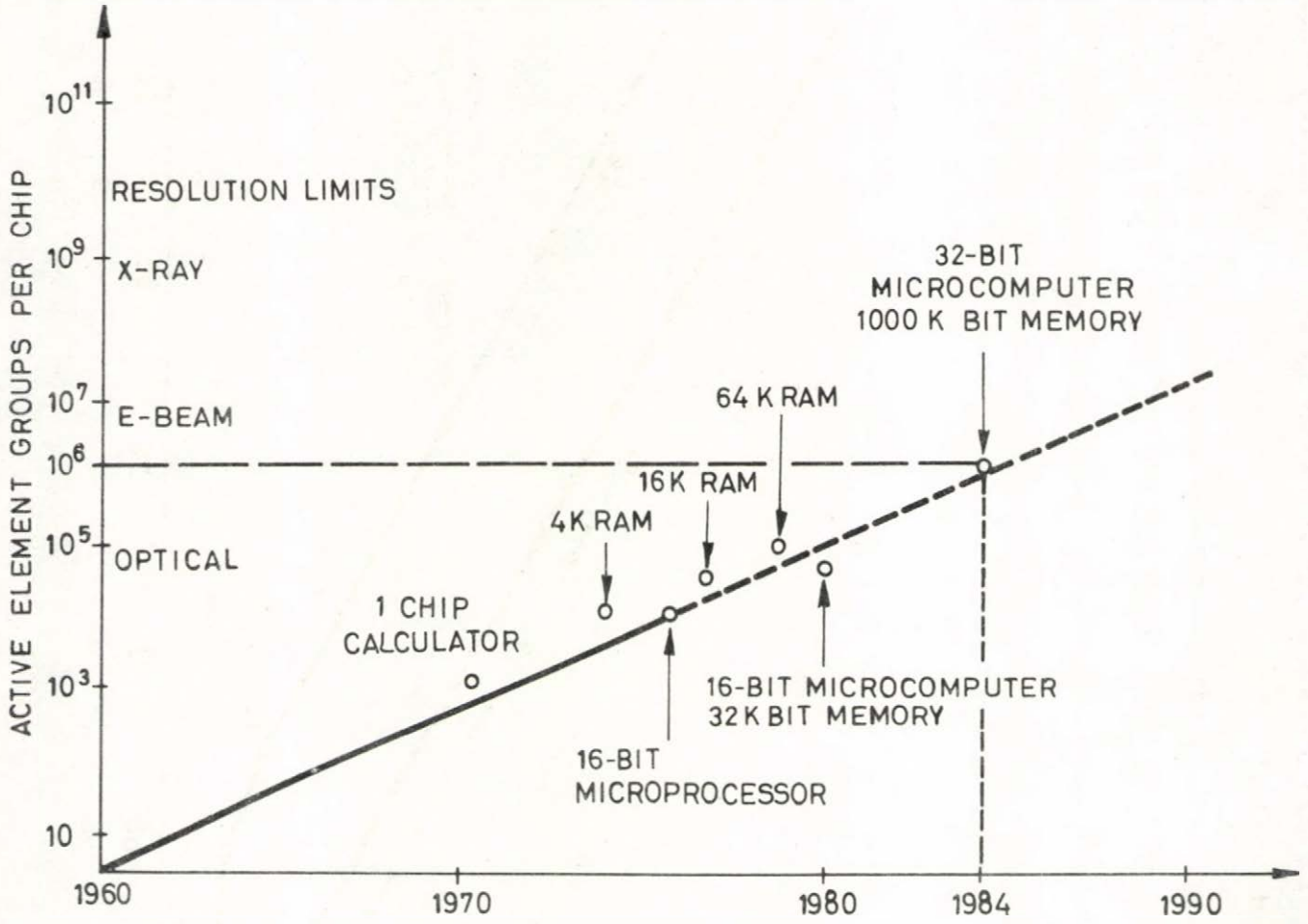


Fig. 1.

Texas Instruments VLSI schedule.

Regarding VLSI *"mathematics"*, the general rules are as follows (see Fig.2.)

- *Number of elements* doubles every year (The so called "Moore-Nayce rule")
- *Chip size* increases slowly with time; therefore, *element size* must decrease.
- *Chip power dissipation* must remain constant; therefore, *dissipation per element* must be halved every year.

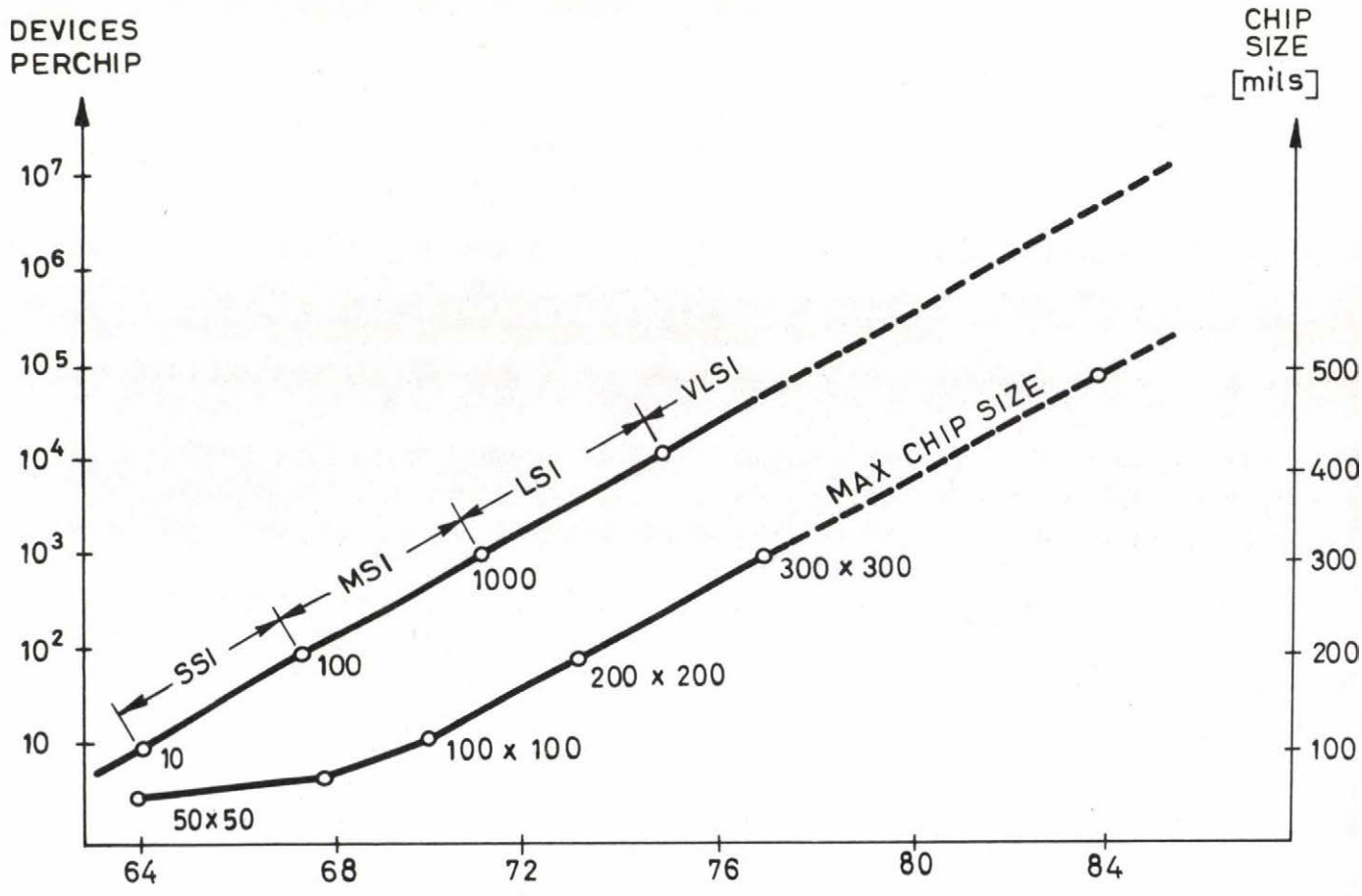


Fig. 2.

VLSI "mathematics".

Considering now the two main topics i.e. microprocessors and microprogramming and the *interaction between them* we find that the greatest impact of microprocessors on microprogramming so far is in the implementation if microprogrammed systems with bit-sliced microprocessors.

In the field of *bit-sliced processor applications* the most important areas of interest are:

- *Emulation* of existing minicomputers (e.g. NOVA, PDP-11, Intel 8080, Interdata 7/16 etc.)
  Difficulties in implementation:
  - Complex bus Structure
  - I/O systems

- High speed *controllers* for
  - process control systems
  - video displays
  - communication etc.

  A microprogrammed system without arithmetic capability oriented toward control applications is called a *micro-controller*. An LSI sequencer together with a microprogram memory and some other elements may do the job alone.

- Special-purpose *computing elements*. The current and potential applications are as follows.
  - Signal processing, e.g. filtering, transformation, statistical analysis etc.
  - Simulation, e.g. direct execution of simulation languages.
  - Video processing.

There is a new tendency in microprogrammable universal computers- the so-called *flexible architecture* (e.g. Nanodata Q-1 with many independent buses which can be configurated in order to meet particular needs). This tendency leads to a new category and brought to life the *universal host* as a research and development tool. We shall refer to its application as *emulative support*.

The *applications of a universal host* computer *by the manufacturer* can be

- in the *development of new microprocessors.*

The manufacturer could easily experiment with different architectures, design instruction sets, and emulate existing machines. It is a flexible hardware and software tool.

- Developing *new I/O chips* to be used with microprocessors.

In many cases the latest chips are even more complex than the processors with which they work. These chips themselves may be microprogrammed. Therefore the universal host could be a tool in the design of such chips - as programmable interfaces, single-chip peripheral controllers - and could support their microprogramming.

- Developing *System Software* for microprocessors.

In this case the system software could be written using the universal host and would not depend on the availability or performance of the processor itself.

Application of the universal host computer *by the user:*

Advantages are the same as for the manufactures, among other things these are experimentation with different architectures and instruction sets, rapid development of software and emulation of existing machines. The user of microprogrammable processors can develop microprograms and essential system software on the universal host. It can be used as a *single development tool* for many different processors and can *benchmark* different devices. It is very useful for the development and design of *custom LSI*.

There is also a new offspring of merging hardware and software, the *SSS (Solid State Software)*. This is a new methodology for developing reliable and efficient application programs for microprocessors. It means low cost, off-the-shelf operational

and support software in ROMS. Operational subroutines are eqvivalent to LSl building blocks. They reduce the programmers burden, requiring only linking modules. Today's microprocessor architectures contain an important feature that has a strong impact on this design strategy. This feature is an external address stack which is managed in hardware, via a stack pointer register and the CALL and RETURN instructions within the processor's instruction set. Use of the stack for program results in a lot of work being performed at a reasonably low price.

Finally, dealing only very shortly with the many possible consequences of *using microprocessors*, we refer to Fig.3 which covers a wide range of overlapping areas affecting *individuals and society*.

I am very pleased to velcome you here in Szentendre and wish you a succesful and pleasant time both on my behalf and on the behalf of the organizing committee.

REFERENCES

1. M. Layer: Microprocessors, side effects and society. Microprocessors, 1/1977/ pp. 305-308.

2. M.A. Boden: Social implications of intelligent machines The Radio and Eelctronic Engineer, 47/1977/ pp.393-399.

3. T.P. Hughes at al.: LSI software. Microcomputers'77 Conf. Record /Oklame City, 6-8 Ápril 1977/ IEEE New York, 1977. pp. 46-53.

4. H. Schmid : Monolitic Procesisng Overview /Privat communications/.

ВЛИЯНИЕ ВЫСОКОРАЗВИТОЙ МИКРОЭЛЕКТРОННОЙ СХЕМНОЙ ТЕХНОЛОГИИ НА СОЗДАНИЕ МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Г. Сальцманн

НП Роботрон, научно-исследовательский центр

Прогрессивное развитие микроэлектронной схемной технологии в связи с новыми потребностями и привычками пользователя существенно может влиять на будущие поколения ЭВМ. Использование больших интегральных схем /БИС/ требует модульной организации функций системы и приводит к новым принципам проектирования. При этом многопроцессорные и многомашинные системы на базе микропроцессоров могут играть большую роль. Настоящий уровень характеризуется концепцией и опробованием ряда новых решений, в результате которых стандартное решение до сих пор не могло быть получено. На основе еще решаемых вопросов, включая вопрос обеспечения совместимости с существующей вычислительной техникой в эксплуатации, краткосрочных принципиальных изменений в области универсально используемых ЭВМ средней и высокой производительностью не ожидается.

1. Введение

С возможностью использования микроэлектронной схемой технологии, позволяющей очень высокую степень интеграции и низкие затраты на элемент, за минувшие годы начал процесс, который в будущем, быть может, рассматривается как качественно новый этап в области реализации и использования техничес-

ких средств ЭВМ. Как показывается, микроэлектроника не только оказывает воздействие на дальнейший научно-технический прогресс в народном хозяйстве, но она также будет влиять на общественное развитие.

Одно направление этого процесса уже ясно вырисовывается. Все в большем объеме находят применение мини ЭВМ и технические средства ЭВМ в совершенно новых сферах, как например, в устройствах децентрализованного управления и обработки для множества устройств и оборудований, включая потребительские товары.

Менее очевидными, однако, являются возможные воздействия на традиционные классы универсальных малых и больших ЭВМ. Считается, что использование больших интегральных схем в этой сфере также может приобрести большое значение. Для рассмотрения возможных направлений развития вычислительной техники мы считаем такую исходную точку особенно пригодной, причем другие коэффициенты влияния не остаются без внимания.

2. О развитии схемной технологии

Настоящее состояние развития БИСов по п-МОП-технологии характеризуется степенью интеграции 10.000 ÷ 20.000 транзисторов/чип. Это позволяет, что в одну схему могут быть включены 8-разрядный микропроцессор с высокой производительностью и тактовой частотой более 2 Мгц., память 8 Кбит или 16 Кбит или даже микро-ЭВМ с центральным процессором, входные и выходные каналы и некоторая память для 8-разрядной обработки. Международно предполагается, что теперешнее развитие постоянного повышения степени интеграции за счет совершенствования технологий и повышения скорости переключения, уменьшения потребления мощности и стоимости будут продолжаться. При этом большую роль будет играть переход к новым методам структурирования. Используемый в настоящее время метод фотолитографии позволяет изготовление структурных элементов шириной прибли-

зительно 1 мкм. Это является границей, установленной длиной световых волн. С применением электроннолучевой литографии расчитывают получить не менее десятикратного меньшего значения. На основе более новых прогнозов /2/ для развития степени интеграции выделяются следующие этапы:

Сроки	Степень интеграции транзисторы/чип	Параметры схемы
1976/77	$1 \div 2 \times 10^4$	8-разрядный центральный процессор или 8-разрядная микро-ЭВМ на одном чипе
1979	10^5	16-разрядный центральный процессор с памятью и входными и выходными каналами
1984	10^6	32-разрядный центральный процессор с памятью 64 Кбит.

На основе процесса бывшего развития микроэлектроники нет повода для игнорирования таких прогнозов. Уже теперь имеются в распоряжении 16-разрядные микропроцессоры или 16-разрядные ЭВМ на одном чипе; получено сообщение о появлении на рынке дальнейших процессоров /4/, /5/. Целью является достижение производительности процессора, отвечающей производительности типа **PDP 11** серии 40. Здесь поднимается принципиальный вопрос о том, как использовать эти технологические возможности для средней и высокой техники и для универсальных машин.

3. Некоторые вопросы проектирования ЭВМ на базе БИСов

Специфика БИСов требует некоторых новых предпосылок для

проектирования процессоров, тесно связанных с требованиями пользователя к новым системам. Существенным влиянием на стратегию проектирования и создание новых структур и архитектур являются следующие аспекты:

1. С одной стороны, современная полупроводниковая техника позволяет высокую и все больше повышающуюся степень интеграции. С другой стороны, число внешних присоединений, т.е., число вводов сигналов в интегральную систему из-за технологических причин очень ограничивается, в настоящее время - на 40 ÷ 70. В связи с этим, только те функциональные единицы /блоки/ интегрируемы, которые доступны в кодированном виде по малому числу сигнальных проводов.

Эти условия требуют модульной организации ЭВМ, разделенной, по возможности, на законченные субкомплексы. Такими функциональными комплексами, например, являются процессоры /с системной программой/, модули памяти или логические блоки интерфейса, коммуникация которых должна проводиться с помощью соединительной системы. Таким образом, специфика техники БИС приводит к созданию многопроцессорных систем с возможностью лучшей деконцентрации функций системы. С точки зрения эксплуатации, этим даны предпосылки улучшения прозрачности системы и достижения более высокой гибкости при приспособлении /адаптации/ к задачам, а также расширяемости за счет переоснащения.

2. С увеличивающейся степенью интеграции повышается комплектность /сложность/ схемы с тенденцией специализации. Повышаются основные затраты на проектирование и разработку схемы и уменьшается количество изделий на единицу устройств. В экстремальном случае все основные функции ЭВМ могут быть размещены на одном чипе. Поскольку речь не идет об использовании схем крупных серий, выводом из этого является задача оптимизации с целью достижения соответствующей степени универсальности набора схем. На основе малого, по возможности, числа типов схем следует обеспечить экономичное производство схем в больших количествах.

Это противоречие между тенденцией специализации и требованиями универсальности также приводит к модульной организации функций системы с названными в п. 1 последствиями, причем различные функции могут быть реализованы соответствующим программированием аппаратных модулей, т.е. процессоров.

3. Экономичное производство БИСов /массовое производство функциональных элементов в одном процессе/ уже сегодня обеспечивает малые затраты на одну логическую функцию.

Путем лучшего освоения технологии и повышения плотности функций эти затраты все больше уменьшаются. Из этого вытекает экономическая необходимость использования БИСы, изготовляемые с помощью таких технологических процессов или технологий, которые требуют малого количества производственных шагов /операций и позволяющих высокую плотность упаковки. Одновременно изменяются пропорции затрат между компонентами вычислительной системы и этим - действительные до сих пор принципы проектирования. В связи с тем, что имеется достаточная дешевая мощность ЭВМ, ее максимальное пользование все же не стоит на переднем плане. Однако необходимым является обеспечивать наилучшее использование остальных ресурсов системы /накопителей, периферий/ и уменьшить затраты на разработку системных программ. Эти аспекты также способствуют созданию многопроцессорных структур за счет использования относительно дорогих ресурсов системы некоторыми процессорами.

Схемная база, требующая меньших затрат, является предпосылкой все большей работы с резервом, а также уменьшить объем и степень сложности машинно-ориентированного матобеспечения. Возможности для этого вытекают из уменьшения организационных затрат внутри системы, из децентрализации и из перевода функций системы, исполнение которых до сих пор преимущественно блоки /модули/ с программируемой памятью. Это, однако, требует организации многопроцессорного режима на основе новых программных средств.

С точки зрения эксплуатации, экономия схемотехники на базе БИСов приводит к улучшению соотношения производительность/цена и открывает возможные пути к системам, допускающим отказы, а также к лучшему обслуживанию ЭВМ, например, за счет широкого использования средств коммуникации и использования языков программирования более высокого уровня.

Этим мы констатируем развитие условий, взаимодействующих на изменение архитектуры и структуры ЭВМ. Наряду с эксплуатационными /пользовательскими/ требованиями значительное влияние, преимущественно, имеют технологические возможности в связи с экономическими факторами.

4. Некоторые возможности использования многопроцессорных и многомашинных структур на базе микропроцессоров

4.1. Повышение производительности и ее приспособление в организации параллельной работы

Организация параллельной работы некоторых процессоров позволяет, при наличии соответствующей схемной базы и данной границы затрат, повысить производительность системы по сравнению с однопроцессорной системой, или за счет соответствующей конфигурации приспособить ее к требованиям пользователя.

Целесообразной является классификация по функциям и задачам /см. рис. 1/, хотя здесь не во всяком случае достигается ясное ограничение.

В случае функционально-разделенных систем проводится специализация функций системы на некоторых процессорах, сохраняя режим последовательной обработки команд. В связи с тем, что в настоящее время программы разрабатываются, преимущественно, для последовательной обработки, такие системы могут использоваться универсально. Например, может быть проведено разделение работ между процессорами для управления, обработки, вво-

да/вывода, технического обслуживания и диагностики.

В случае разделенных на задачи систем организуется параллельная /одновременная/ обработка последовательного параллелизованного алгоритма или программы на некоторых процессорах с целью повышения пропускной способности системы. Универсальное использование таких структур связано с автоматическим разделением программы на параллельно обрабатываемые секции на этапе компиляции. Для специальных случаев применения /например, в области управления технологическими процессами/ параллельная работа организуется программистом или структура системы прямо выводится из алгоритма решения одного класса проблем /например, для задач вычислительной математики/.

На практике по функциям и по задачам не редко проводится в смешанном виде.

4.2. Системы допускающие отказы

Многомикропроцессорные структуры также могут являться основой реализации систем, допускающих отказы, необходимые в случае высоких требований к надежности. Эти структуры с "мягким поведением отказа" представляют собой экономическую альтернативу к двойным и тройным системам. Соответствующая модульная организация системных функций, аппаратные и программные средства и упорядочивание приоритета в связи с эффективными мерами диагностики позволяют в случае неисправности сохранение работоспособности системы, в необходимом случае также с пониженной производительностью.

4.3. Децентрализованная обработка

Для различных случаев применения, например, для терминалов или комплексов цифрового управления, типичными являются относительно автономные подсистемы, связанные с ведущей ЭВМ. С целью упрощения организации системы, ограничения затрат на

аппаратуру /линии/ передачи информации и повышения надежности системы возможным является переводить некоторые процессы обработки на подсистемы. Экономической основой для этого также являются дешевые микропроцессоры. В связи с тем, что управляемые ведущей ЭВМ интеллектуальные подсистемы только "слабо" соединены друг с другом, такой комплекс может считаться многомашынной системой.

4.4. Состояние в настоящем

Использование многопроцессорных систем и соединение малых и больших ЭВМ для создания многомашинных комплексов в области электронной обработки данных не являются новостью.

В международных масштабах, однако, за минувшие годы в возрастающей степени были спроектированы и опробованы системы на базе микропроцессоров во многих вариантах. Они, преимущественно, служат для исследования принципов решения и частичных проблем; коммерческого значения, однако, эти системы до сих пор не могли приобрести.

Несмотря на то, что определенные элементы уже используются, как например,

- шина, работающая с разделением времени как соединительная система;
- коммуникация через общую память для быстродействующего обмена данными;
- относящиеся к процессорам накопители для разгрузки шины и сокращения времени ожидания ВМ;
- принцип "мастер-слейв" /Master-Slave/ для проведения обмена данными,

стандартные решения для многопроцессорных систем до сих пор не были получены.

В случае системы "слабого или нежесткого" соединения ситуация немного другая. На рынке появились первые изделия,

предназначенные для режима "мастер-слейв" /5/.

При оценке данной ситуации следует учесть, что широкое введение новых архитектур ЭВМ для универсального использования также предполагает обеспечение совместимости с существующим поколением /генерацией/, по крайней мере на уровне прикладных программ. В связи с этим следует отметить, что теоретических основ, вспомогательных средств и эмпирических данных для проектирования многопроцессорных систем, в особенности для определенных целей использования, еще мало. Это касается почти всех частичных проблем, как

- проектирования и оптимизации архитектуры и структуры;
- моделирования, описания системы и имитации;
- модульной организации, специализации функций на функционально разделенные системы и конфигурации интерфейса;
- сети связи;
- операционных систем;
- разделения программы на параллельно обрабатываемые секции; языков программирования, поддерживающих параллельную обработку;
- динамического структурирования, самодиагностики и реконфигурации;
- использования аппаратных средств для уменьшения объема и сложности машинно-ориентированного матобеспечения;
- прямой интерпретации языков более высокого уровня;
- обеспечения совместимости;
- оценки производительности и эффективности;
- создания наборов БИС для многопроцессорного и многомашинного режима.

Эта ситуация требует интуитивного и широкого ориентированного на проблему подхода. Для того, чтобы полностью использовать все возможности высокоразвитой микроэлектронной схемой технологии для перспективной вычислительной техники, надо еще решать ряд научно-исследовательских задач.

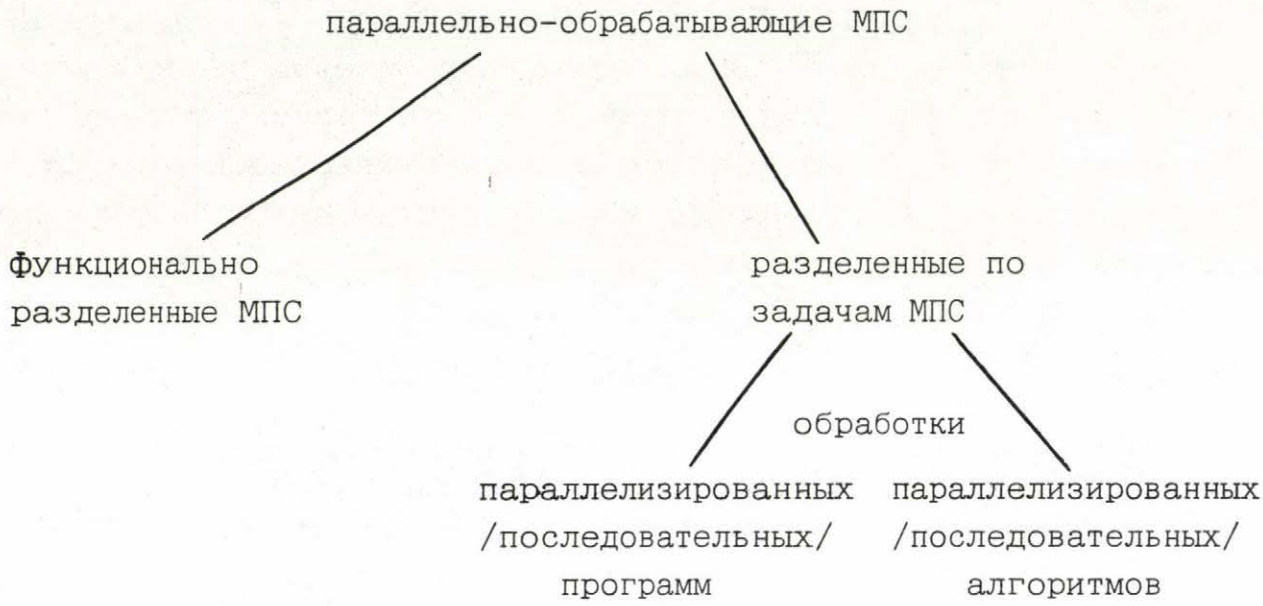


Рис. 1.: Классификация многопроцессорных систем /МПС/

ЛИТЕРАТУРА:

/1/ Grimes, J.D.: Microprocessors and Microcomputers. Selected reprints from Computer. IEEE Computer Society 1977.

/2/ IBM 370 on a Chip in 1984. Electronics Weekly, London /1977/ Nr. 893, p. 2.

/3/ Köhler, E.: Entwicklungsbestimmende Faktoren für die Halbleitertechnik und Mikroelektronik nach 1980. Sozialistische Rationalisierung in der Elektrotechnik/Elektronik Berlin 6 /1977/ 7, p. 176-178.

/4/ Osborne, A.: 16-bit-Mikroprozessoren erobern den Minicomputermarkt. Elektronik, München, 26 /1977/ 10, p. 16.

/5/ Klasche, G.: Notizen aus den USA. Elektronik, München, 26 /1977/ 9, p. 81-88.

/6/ Köhler, R.: Computer der 70er Jahre. Elektronik, München, 24 /1975/ 1, p. 75-79.

/7/ Enslow, P.H.: Multiprocessors and Parallel Processing. John Wiley and Sons, New York, 1974.

/8/ Reyling, G.: Performance and Control of Multiple Microprocessor Systems. Computer Design, Winchester Ma, 13 /1974/ 3, p. 81-86.

/9/ Weissberger, A.J.: Analysis of Multiple Microprocessor System and Architectures. Computer Design Winchester Ma, 16 /1977/ 6, p. 151-163.

/10/ Turn, R.: Computers in the 1980s-Trends in Hardware Technology. Information Processing 74, North-Holland Publishing Company, 1974 p. 137-140.

/11/ Merkel, G.: Zur Architektur elektronischer Rechenanlagen. Beiträge zur Informationsverarbeitung, Schriftenreihe Informationsverarbeitung BSB B.G. Teubner Verlagsgesellschaft, Leipzig, 1977, p. 77-89.

MULTILEVEL MEMORY STRUCTURE FOR THE CONTROL OF MICROPROGRAMMED MACHINES

J.C. REMESAR

Cuba

ABSTRACT

This paper proposes a multilevel the organization of the control memory in microprogrammed computers. The fundamental achievement is the flexibility obtained in programming a system with such architecture. User's language development is done via micro-programming. Only a simple assembler could allow the programmer to employ high and low-level instructions in his work. This architecture presents good characteristics for special-purpose microprocessor-based system design.

INTRODUCTION

Microprocessor revolution has played a decisive role in the growth of software/hardware costs ratio. Because of this people are paying more attention to those problems traditionally solved by software, which could be simplified by hardware.

Microprocessor-based systems are very often used for special--purpose equipment design, but when programs to be developed for them present some complexity, the low price components achievement could be lost because of software development costs. There are two alternatives:

- to code large programs in machine language
- to develop basic software means as assemblers, macro-assemblers, compilers, stc.

The architecture presented allows the solving of user-oriented language creation via firmware.

SOME FUNDAMENTAL TOPICS IM MICROPROGRAMMING

The general principles upos which Wilkes based his work (1), and the development of them until today (2, 3, 4) might be summarised as - follows, using fig 1:

- it is a processor whose control is governed by a set of lines called the "Control bus"
- it is a control memory (CM), loaded with a binary number at each of its words in such a way that every one corresponds to a certain microinstruction (MI) code of the processor
- the microinstruction set of any processor is determined during its design, and the instruction set of the computer is developed by programming the CM. On this rests the flexibility of microprogrammed machines.

To recognize the end of each microprogram (MP), and consequently the end of each machine instruction, a determined code (a MI which represents the end of MP) can be used, or a bit might be taken along the CM. The CM area designated for this organization purpose is shown in fig.2. Depending on certain parameters such as the mean number of MI per MP, word length, etc, one of these two extreme methods could present more befinits in CM word, with the exception of a bit in the second one, are to be sent throught the control bus to produce an MI at the processor.

An important feature of some microprogrammed machines' organization is to provide a way to use certain common MP sections as subroutines. This implies the existence of a "jump" MI code. The domain of these jumps is generally restricted to a sub-set of the addressable field of the CM, for instance marking them

out within CM page limits, limiting the magnitude of the jump to a fixed maximum value, etc.

Using a unique field of the CM word for the MI executable by the processor and for the jumps (also for the end of MP when it is necessary) implies hardware complexity and speed loss which could be saved by assigning separated fields for the codes to be sent to the processor and the codes for CM address selection, as is show in fig 3.

This approach is used in the INTEL 3001/2 microprogrammable microprocessor family (5). In this series, the following must be specified in the CM word:

- in the first field: the code to be processed by the Control-Memory Unit to generate the address of the CM word which would be read at the next cycle
- in the second field: the code of the MI tobe executed by the - central processor.

MULTILEVEL CONTROL MEMORY ORGANIZATION

The main idea expressed in this paper consists of the creation of - two or more levels of "addressing fields" at the CM.

This would allow the elaboration of powerful instructions (as macroinstructions) on the basis of the ones from the preceeding - level without basic software development.

More detailed comprehension of the preceeding could be reached-assuming the following:

- let us "cut" the CM into two independent blocks at the field boundary.

At the 0^{th} level are the processor's MI codes.
At the 1^{st} level are the memory Control Unit codes, which represent addresses of the 0^{th} level MI, or some information to elaborate it, taking into account certain signals.

- by adding new memory blocks as new levels the 2^{nd}, 3^{rd}, and so on, levels may be created. Always, the contents of the N^{th} level-represent the address of a $(N-1)^{th}$ level location.
- a program at the N^{th} level is started by addressing its beginning location from the $(N+1)^{th}$ level, and finishes when the "end of program" code is reached.

A three-level CM architecture is shown in fig 5. On the instruction bus the user's instructions, which might be from any level are present. When it is recognized (for instance) by the MCU_2, the initial address of the 2^{nd} level program which performs this instruction is generated on this unit, and the process is started. Each content of the CM_2 locations provokes the generation of an initial program address at CM_1. Then, a program at the 1^{st} level is started and, consequently, the corresponding one at the 0^{th} level. The process is finished when "end of program" code is detected at the highest level.

ADVANTAGES OF THE MULTILEVEL CM ORGANIZATION

The benefits of such an architecture can be summarised as follows:

- the user's instruction set could be composed of higher and lower level ones, and they may ne sequenced without any rules.
- at the lowest levels high speed memory chips must be placed keeping cheaper slower memories for the highest ones. At this level the memory speed has less influence on the overall system speed. Also, this speed could be practically dependent only on the lowest level (0^{th} and 1^{st}) memory

chips by employing pipe-line methods -- with only a few additional components.

- after designing a system, only a simple assembler is needed
- currently available components permit compact system implementation - using multilevel control memory.

REFERENCES

1/ Wilkes, M.V.: "The best way to design an automatic calculating machine"
Report of Manchester University Computer Inaugural Conference July 1951

2/ Sell, J.V.: "Microprogramming in an integrated hardware/software system" Computer Design Jan 1975

3/ Casaglia, G.: "Nanoprogramming vs microprogramming"
COMPUTER Jan 1976

4/ Baigley J.D.: "Microprogrammable virtual machines
COMPUTER Feb 1976

5/ Intel Corporation: Intel Data Catalog 1975

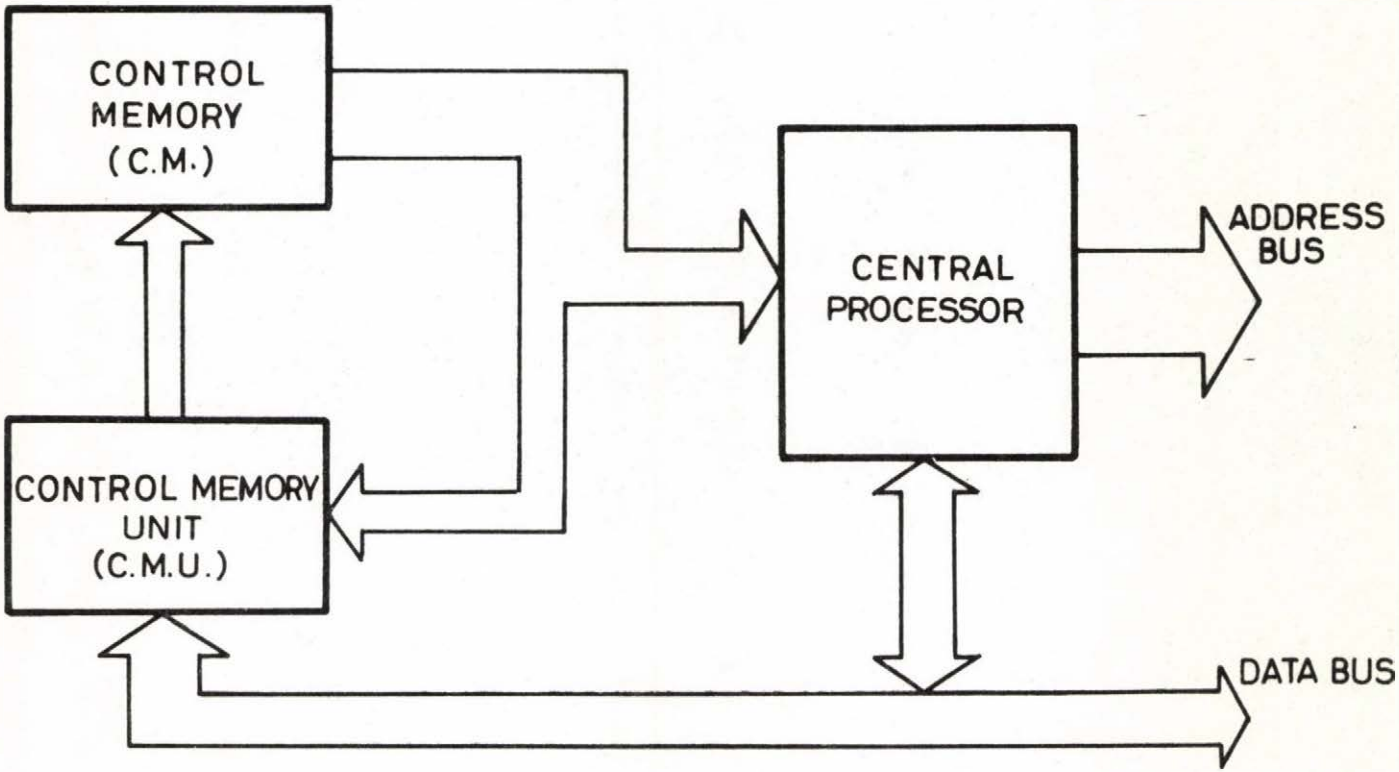


Fig.1. MICROPROGRAMMED CONTROL ORGANIZATION.

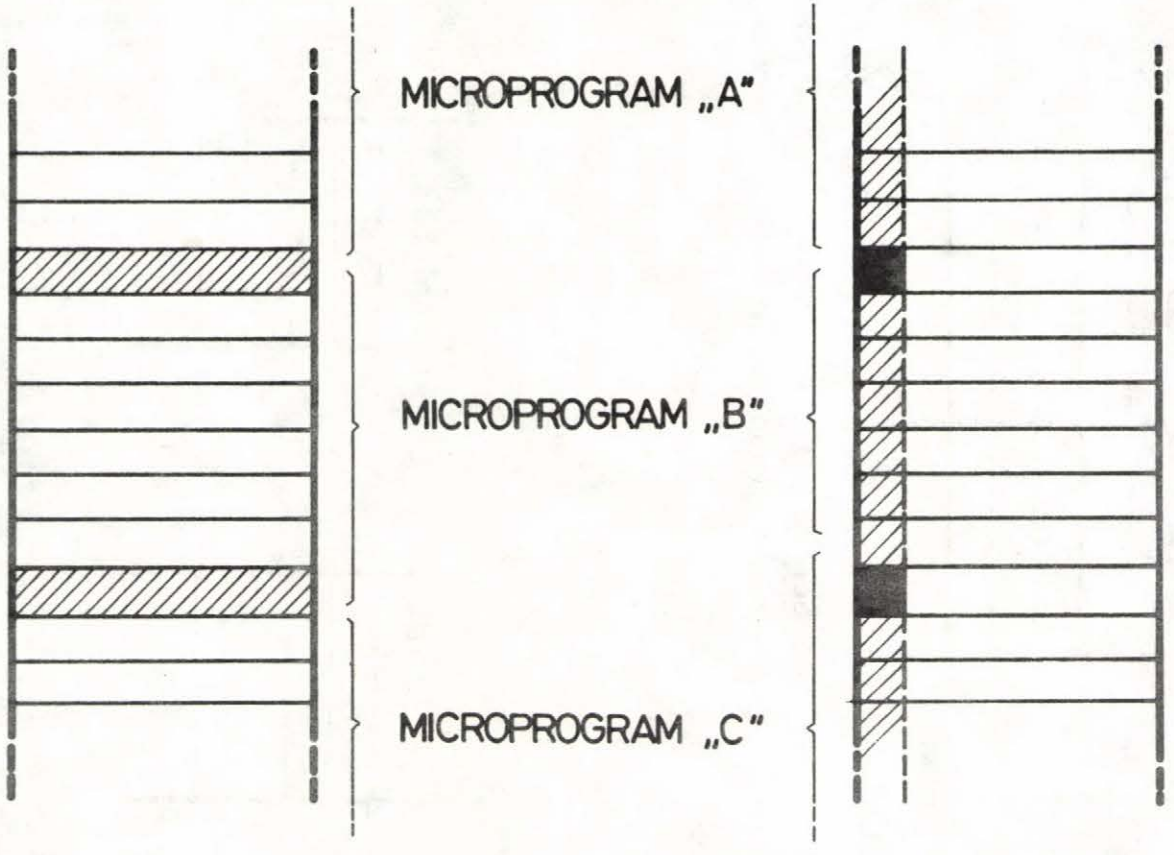


Fig. 2.

The CMU area occupied by the "end of program" code is shown, for the two extreme methods: designing a code along a CM word (left) and assigning a bit along the CM (right).

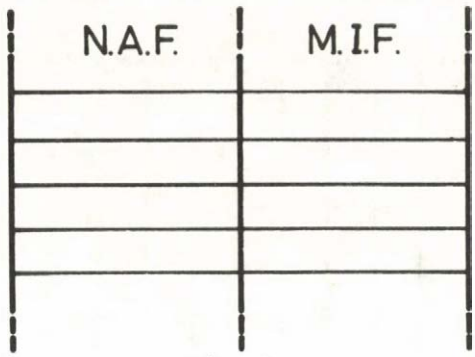


Fig. 3.

The next address field (N.A.F) and the Microinstruction field could be written at the same CM word forming a microinstruction

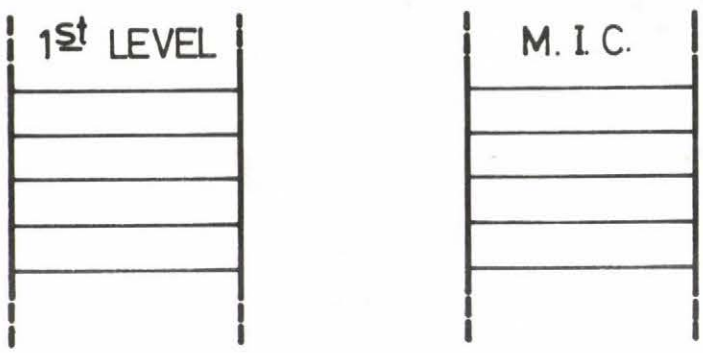


Fig. 4.

The fields had been separated and the 1^{st} level is formed. At the 0^{th} level are the microinstruction codes.

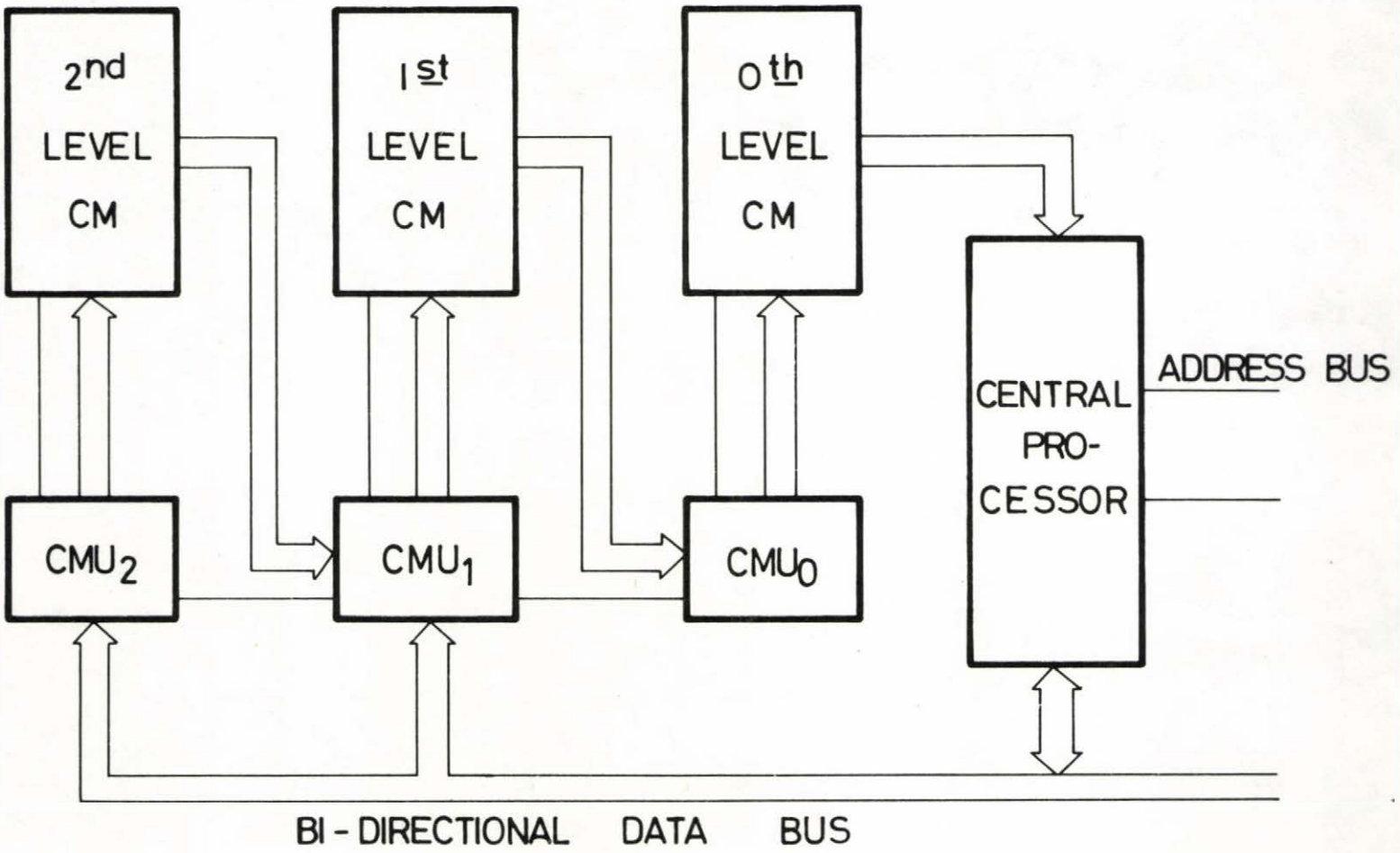


Fig. 5.

Three-level control memory organization

СИНТЕЗ СТРУКТУР МИКРОПРОЦЕССОРОВ И МИКРОПРОЦЕССОРНЫХ СИСТЕМ

Башлаков Е.П., Кратко М.И.

Резюме

В докладе предлагается один из возможных подходов к построению формализованной теории синтеза структур микропроцессоров и микропроцессорных систем. Приводятся оценки сложности коммутатора связи и числа микропроцессоров для проблемно-ориентированных микропроцессорных систем.

СИНТЕЗ СТРУКТУР МИКРОПРОЦЕССОРОВ И МИКРОПРОЦЕССОРНЫХ СИСТЕМ

Башлаков Е.П., Кратко М.И.

В докладе предлагается теория формализованного синтеза структур микропроцессоров и микропроцессорных систем для различных областей применения.

Для проблемно-ориентированных микропроцессорных систем /микропроцессоров/ устанавливается верхняя оценка сложности коммутатора связей между микропроцессорами /регистрами/ и нижняя оценка числа микропроцессоров /регистров/.

Используя аппарат булевых матриц и теории графов, предлагаемая теория позволяет получить для проблемно-ориентированных микропроцессорных систем /микропроцессоров/ удобные для практического использования методы оптимизации связей в микропроцессорных системах /микропроцессорах/ и минимизации числа микропроцессоров /регистров/.

Введение

Элементная база вычислительных машин быстро приближается к естественному пределу физических возможностей. Поэтому дальнейшее повышение производительности вычислительных машин и систем возможно за счет совершенствования их структуры и архитектуры. Современные вычислительные машины - сложные микропроцессорные системы с универсальными связями между микропроцессорами.

Большая вычислительная мощность, универсальность, гибкость, высокая живучесть, низкая стоимость вычислений, расширение контингента пользователей таких вычислительных машин оправдывают условия, направленные на развитие микропроцессорных систем.

Одной из проблем, которые здесь возникают, является формализация выбора средств и методики проектирования сложных микропроцессорных систем.

Микропроцессорную систему можно представить как совокупность микропроцессоров и средств связи, позволяющих объединить этот набор микропроцессоров в единую вычислительную систему.

Современный микропроцессор-процессор, выполненный на одном кристалле.

С точки зрения надежности каждый кристалл должен иметь небольшое количество выводов. Поэтому основные трудности в создании сложных микропроцессорных систем смещающихся в область обеспечения эффективной связи между микропроцессорами.

Процесс переработки информации в микропроцессорной системе /микропроцессоре/ можно представить как композицию двух автоматов управляющего и операционного, в которой выходные сигналы управляемого автомата совпадают с входными сигналами операционного автомата, а его входными сигналами являются выходные сигналы операционного автомата /рис. 1/.

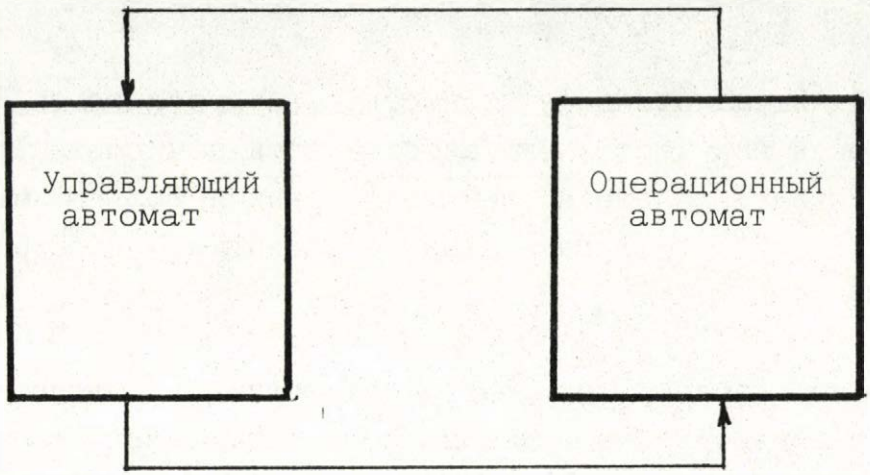


Рис. 1.

Конкретные области применения предъявляют различные требования к быстродействию, гибкости, надежности, живучести, и другим характеристикам системы. Поэтому актуален вопрос о максимальной эффективности использования в системе.

Для математической формулировки задач коммутации между микропроцессорами и максимальной эффективностью использования микропроцессоров в системе введем ряд понятий и определений.

§ 1. Основные понятия и определения

Определение 1.1.

К о м м у т а т о р о м, точнее n -коммутатором, называется устройство K, имеющее входных и n выходных каналов, занумерованных числами от 1 до n, такое, что для каждого взаимно однозначного отображения $\phi = \begin{pmatrix} 1, 2, \ldots, n \\ i_1, i_2, \ldots i_n \end{pmatrix}$ множества $\{1, 2, \ldots, n\}$ на себя существует состояние q_ϕ пребывания в котором коммутатор K имеет соединенными первой входной канал с i_1 - М выходным каналом, второй входной канал с i_2 - N выходным каналом и т.д. /говорят, что в состоянии q_ϕ коммутатор реализует отображение ϕ /.

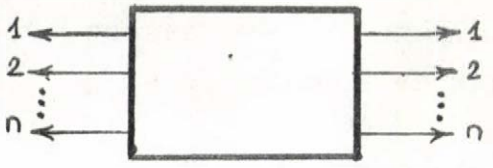


Рис. 2.

Определение 1.2.

Коммутаторная сеть определяется следующим образом.

1. Каждый отдельно взятый коммутатор есть коммутаторная сеть. Все входные каналы этого коммутатора являются входными каналами данной сети, все выходные каналы коммутатора - ее выходными каналами.

2. Пусть $\mathcal{N}$ - коммутаторная сеть, x - некоторый ее входной канал и z - некоторый ее выходной канал. Результат присоединения канала x к каналу z также является коммутаторной сетью. Ее входными каналами являются все входные каналы сети $\mathcal{N}$, за исключением канала x, выходными - все выходные каналы сети $\mathcal{N}$, за исключением канала z.

3. Пусть $\mathcal{N}$ и $\mathcal{Y}$ - коммутаторные сети, x - некоторый входной канал сети $\mathcal{N}$ и z - некоторый выходной канал сети $\mathcal{Z}$. Результат присоединения канала x к каналу z также является коммутаторной сетью. Ее входными каналами являются все входные каналы сети $\mathcal{N}$, за исключением канала x, и все входные каналы сети $\mathcal{Y}$, выходными - все выходные каналы сети $\mathcal{N}$ и все выходные каналы сети $\mathcal{Y}$ за исключением канала z.

4. Не существует никаких других коммутаторных сетей, кроме тех, которые могут быть получены конечным числом применений правил, изложенных в п. 1-3.

Определение 1.3.

Состоянием коммунальной сети назовем совокупность состояний всех составляющих ее коммутаторов.

Пусть в коммутаторной сети $\mathfrak{N}$ выходной канал z_1 коммутатора K_1 присоединен ко входному каналу x_2 коммутатора K_2. Пусть коммутатор K_1 в состоянии q_1 имеет соединенным входной канал x_1 с выходным каналом z_1. Пусть коммутатор K_2 в состоянии q_2 имеет соединенным входной канал x_2 с выходным каналом z_2. Тогда, если в сети $\mathfrak{N}$ коммутатор K_1 находится в состоянии q_1, а K_2 - в q_2, канал x_1 считывается соединенным с каналом z_2. Далее отношение соединенности распространяется по транзитивности: Если α соединен с β и β соединен с γ, то α соединен с γ. Таким образом, для каждого состояния сети можно установить, какие входные каналы сети соединены с какими выходными каналами, когда сеть пребывает в данном состоянии.

Определение 1.4.

Сетью связи S микропроцессорной системы называется коммутаторная сеть с занумерованными коммутаторами /числами от 1 до N / такая, что:

1. В точности один входной канал каждого коммутатора является входным каналом сети. Будем нумеровать эти входные каналы теми же числами, что и соответстующие им коммутаторы.

2. В точности один выходной канал каждого коммутатора является выходным каналом сети. Будем также нумеровать эти выходные каналы теми же числами, что и соответствующие им коммутаторы.

3. Для каждого взаимно однозначного отображения $\phi = \begin{pmatrix} 1,2,\ldots,N \\ i_1,i_2,\ldots,i_N \end{pmatrix}$ множества $\{1,2,\ldots,N\}$ на себя существует состояние сети

q_φ , пребывая в котором сеть S имеет соединенным свой первый входной канал с i_1-М выходным каналом, второй входной канал - с i_2-М выходным каналом, и т.д. Говорят, что в состоянии q_ψ сеть S реализует отображение Ψ .

Пусть P_i - микропроцессор с одним входом и одним выходом.

Определение 1.5.

Микропроцессорной системой B , построенной на основе микропроцессора P_i и коммутаторной сети микропроцессорной системы S , является результат присоединения к каждому коммутатору сети S одного микропроцессора, причем это присоединение выполнено так, что выходной канал коммутатора, который является выходным каналом сети S , присоединяется к входу микропроцессора, а входной канал, являющийся входным каналом сети S , - к выходу микропроцессора $P_i (i=\overline{1,N})$.

Представим процесс переработки информации в микропроцессорной системе в виде композиции из двух автоматов - управляющего A и операционного [1-3] .

Функционирование схемы из двух автоматов определяется функциями переходов и выходов управляющего автомата A .

Определение 1.6.

Управляющим автоматом A назовем конечный абстрактный автомат Мура $A = \langle \mathcal{X}, \mathcal{Y}_A, \mathcal{A}, a_0, a^*, \delta_A, \mu_A \rangle$ $\mathcal{X}$ - входной, а $\mathcal{Y}_A$ - выходной алфавиты, $\mathcal{A}$ - множество состояний, a_0 и a^* - начальное и заключительные состояния, δ_A /функция переходов/ - отображение $\mathcal{A} \times \mathcal{X}$ в $\mathcal{A}$, μ_A /функция выходов/ - отображение $\mathcal{A}$ в $\mathcal{Y}_A$.

Определение 1.7.

Последовательность $P^{*}=(a_{o}=a_{i_1}, a_{i_2}, \ldots a_{i_n}=a^{*})$ назовем путем в автомате A .

Та или иная последовательность выходных сигналов, выдаваемая управляющим автоматом, вызывает последовательное изменение состояний микропроцессорной системы.

Определение 1.8.

Состоянием микропроцессорной системы назовем различные наборы состояний составляющих ее элементов.

Управляющий автомат A получает от микропроцессорной системы B выходные сигналы $x (x \in X)$, которые представляют собою кортежи $<c_1, c_2, \ldots, c_n>$ значений логических условий, определенных соответствующим образом на микропроцессорной системе, характеризующих результаты проводимых вычислений в процессе переработки информации. Входные сигналы микропроцессорной системы отождествляются с некоторыми отображениями /преобразованиями/ множества состояний этой системы в себе. Проверяя на каждом такте работы требуемые логические условия, микропроцессорная система выдает соответствующий выходной сигнал, зависящий лишь от состояния микропроцессорной системы, но не от сигнала на входе.

Таким образом, микропроцессорная система является автоматом Мура с бесконечным числом состояний, в которой может быть обеспечен одновременный обмен информацией между всеми микропроцессорами, ведущийся каждый по отдельному тракту связи с целью избежать конфликтных ситуаций, связанных со смешиванием информации.

Возникают задачи:

1./ Какое минимальное количество входов и выходов должны иметь коммутаторы, чтобы из них можно было построить сеть микропроцессорной системы, состоящую из N микропроцессоров и как эту сеть строить;

2./ Какое минимальное количество микропроцессоров должна иметь микропроцессорная система для решения задач в конкретных областях применения.

§ 2. Постановка задачи

Приведем математическую формулировку задачи коммутации связей в микропроцессорной системе на языке теории графов.

Определение 2.1.

Граф G, имеющий N вершин /занумерованных числами от 1 до N /, назовем информационным, если для каждой N - перестановки $\Psi=(i_1,i_2,\ldots,i_N)$ в нем можно выделить N путей таких, что первый путь начинается в вершине номер 1 и оканчивается в вершине номера i_1, второй - начинается в вершине номера 2 и оканчивается в вершине номера i_2, и т.д., причем никакие два различных пути не имеют ни одного общего ребра.

Определение 2.2.

Степенью графа G назовем максимальную степень его вершины /число ребер, сходящихся в этой вершине/.

Задача 1. Требуется построить N - вершинный информационный граф минимальной степени.

Для формальной постановки задачи II зададим процесс переработки информации в микропроцессоре P_i в виде схемы из

двух автоматов $\mathcal{A}_i$ и B_i $(i=\overline{1,N})$.

Каждая такая "схема" определяет некоторое, вообще говоря, частичное преобразование j_{Ai} на операционном автомате B_i . /Операционный автомат B_i - многорегистровый автомат с периодически-определенными преобразованиями [1,2] /.

Пусть $\beta=\{P_1,P_2,\ldots,P_N\}$ - множество микропроцессоров в микропроцессорной системе B , а $\mathcal{F}_B=\{j_{A1}, j_{A2},\ldots, j_{AN}\}$ - множество преобразований, осуществляемых на множестве β выходными сигналами управляющего автомата A .

Сложность микропроцессора $\rho(P_i)$ определим формулой:

$$\rho(P_i) = \rho(j_{Ai}) + \rho(B_i) + \rho(\phi_i) \qquad /2.1/, \text{ где}$$

$\rho(j_{Ai})$- сложность схемы, реализующих преобразование j_{Ai} , $\rho(B_i)$ - сложность операционного автомата B_i , а $\rho(\varphi_i)$- сложность коммутатора связи φ_i микропроцессора P_i .

Из определения /1.5/ следует, что конструктивными элементами микропроцессорной системы B являются микропроцессоры и коммутаторная сеть S , связывающая микропроцессоры в единую систему.

Отсюда искомая оценочная функция /функционал/ сложности автомата B может быть представлен в виде:

$$F(B) = \sum_{i=1}^{N} \rho(P_i) \qquad /2.2/$$

Используя в качестве критерия оптимальности минимальное значение этого функционала, получаем, что задача синтеза и минимизации числа микропроцессоров микропроцессорной системы сводится к отыскиванию такой технической реализации автомата B при заданном функционировании схемы из двух автоматов, чтобы:

$$\min_{(P_{i\kappa})\in F(B)} F(B) = \sum_{\kappa=1}^{m} \rho(\tilde{P}_{i\kappa}) \qquad /2.3/$$

и

$$\min_{\tilde{P}_{i\kappa}\in F^*(B)} F^*(B) = \sum_{\kappa=1}^{m} \tilde{P}_{i\kappa} \qquad /2.4/$$

Учитывая то обстоятельство, что конструктивными элементами операционного автомата B_i являются регистры и комбинационные схемы, реализующие заданные на этих регистрах преобразования, формулы /2.1 - 2.4/ для синтеза многопроцессорных автоматов будут иметь аналогичный вид при соответствующей интерпретации слагаемых.

§ 3. Построение сети связи микропроцессорной системы

С целью решения задачи 1, определим функцию $\Phi(N)$ следующим образом: $\Phi(N)$ равно минимуму степеней информационных графов, имеющих ровно N вершин.

Теорема 3.1. Справедлива следующая нижняя оценка:

$$S(N) \geq C_1 \frac{\log N}{\log \log N}$$

Доказательство. Пусть граф G имеет N вершин и максимальная степень его равна $\alpha (\alpha > 1)$. Очевидно, что в окрестности радиуса $\frac{\log_\alpha N}{2}$ любой его вершины находится не более, чем $\frac{N}{2}$ вершин. Следовательно, можно задать по крайней мере $\frac{N}{2}$ пар номеров вершин $\langle x_i, y_i \rangle$ $(1 \leq i \leq \frac{N}{2})$ таких, что все x_i - различные числа, и все y_i - различные, и для каждого x_i вершина номера x_i в графе G находится на расстоянии не меньше, чем $\frac{N \log_\alpha N}{4}$ от вершины номера y_i. Чтобы в графе G можно было провести все пути, соответствующие этим парам, т.е. такие, что i -ый путь начинается в вершине

номера i и оканчивается в вершине номера y_i, и никакие два различные пути не имеют ни одного общего ребра, число ребер в графе G должно быть не меньше $\frac{N \log_\alpha N}{4}$. Фактически граф G имеет число ребер N_α. Следовательно, чтобы граф G был информационным, должно выполняться неравенство.

$$N_\alpha \geq \frac{N \log_\alpha N}{4}$$

или $\alpha \geq 1/4 \log_\alpha N$. Решение уравнения $x = \log n$, как известно, асимптотически равно $\frac{\log n}{\log \log n}$. Значит для того, чтобы выполнялось приведенное выше неравенство, степень графа G должна быть по порядку $\geq \frac{\log N}{\log \log N}$. Теорема доказана. Теперь покажем, что существуют информационные графы степени $\leq 4 \frac{\log_2 N}{\log_2 \log_2 N}$. Воспользуемся следующим классом графов. Чтобы упростить его описание положим, что число N является степенью двойки и расположим все вершины на плоскости в точках с целочисленными координатами в виде прямоугольника ширины N и высоты $\log_2 N$. Каждый ряд содержит N вершин и таких рядов имеется $\log_2 N$, так что всего граф содержит $N \log_2 N$ вершин. Ребра проводятся только между вершинами двух соседних рядов, причем в зависимости от того, какая пара рядов соединяется, ребра проводятся по разному. Первый /сверху/ ряд соединяется со вторым так, что ребра проводятся между вершинами первого и второго ряда, имеющими одинаковые номера по модулю $\frac{N}{2}$. Чтобы соединить второй ряд с третьим надо разбить их оба на части - $\{1,2,\ldots,\frac{N}{2}\}$ и $\{\frac{N}{2}+1, \frac{N}{2}+2,\ldots,N\}$, и в каждой части отдельно соединить вершины, имеющие одинаковые номера по модулю $\frac{N}{4}$. Чтобы соединить ряды κ и $(\kappa+1)$ надо каждую часть, на которую был разбит ряд κ при соединении его с рядом $(\kappa-1)$, разбить пополам, так же разбить ряд $(\kappa+1)$ и соединить отдельно в каждой из частей вершины, имеющие одинаковые номера по модулю числа в два раза меньшего, чем для рядов $(\kappa-1)$ и κ.

Тот же класс графов может быть задан индуктивно следующим образом. Граф ранга 1 это граф, изображаемый на рис. 3.

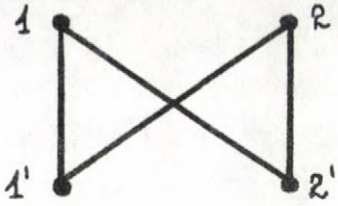


Рис. 3.

У него четыре вершины, две из них, называемые верхними, это вершины 1 и 2, нижние - вершины 1′ и 2′. Граф ранга 2 строится из двух графов ранга 1 следующим образом.

Два графа ранга 1 рисуются рядом, и непосредственно над каждой верхней вершиной обоих графов рисуется еще по одной вершине.

Каждая верхняя вершина графа ранга соединяется ребрами с вершиной, находящейся непосредственно над ней и с вершиной, находящейся непосредственно над вершиной того же номера во втором графе ранга 1.

Полученный граф изображен на рис. 4.

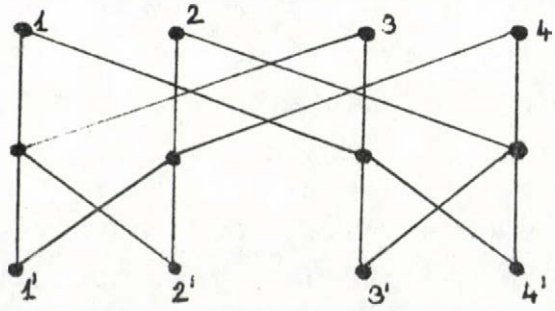


Рис. 4.

Верхние вершины в нем нумеруем числами 1, 2, 3 и 4.

Чтобы получить граф ранга n поступают аналогично. Берут два графа ранга $(n-1)$. Над каждой верхней вершиной каждого из этих графов рисуют еще одну вершину. Соединяют каждую верхнюю вершину графа ранга $(n-1)$, с непосредственно находящейся над ней вершиной и с вершиной, непосредственно находящейся над верхней вершиной того же номера в другом графе ранга $(n-1)$, как показано на рис. 5.

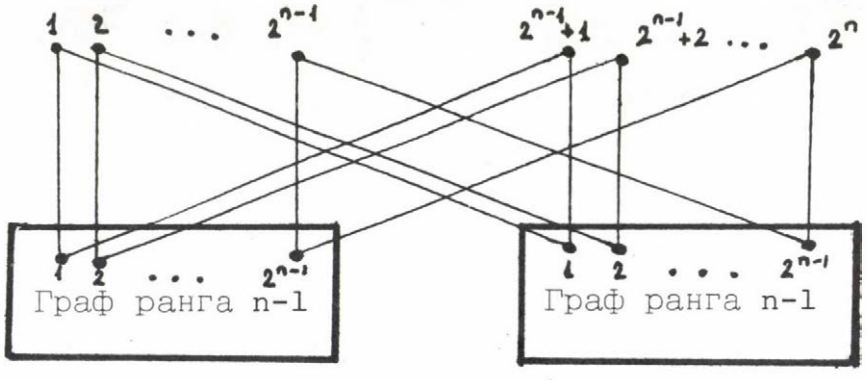


Рис. 5.

Нумеруем верхние вершины нового ранга n числами от 1 до 2^n.

Пусть $N=2^n$. Для нас будут важными следующие свойства введенных графов.

1. Степень каждой вершины, кроме вершин, расположенных в верхнем и в нижнем ряду, равна четырем. Степень вершин, расположенных в верхнем и нижнем ряду, равна двум.

2. Любая полоса высоты τ в графе ранга n, при условии, что τ намного меньше n /т.е. подграф на вершинах, которые находятся в рядах с номерами $\{i, i+1, \ldots, i+\tau\}$ $(1 \leq i \leq n-\tau)$, распадаются на $2^{n-\tau}$ несвязных частей, являющихся графами ранга τ /.

Если к нижнему ряду вершин графа ранга n "приклеить" зеркальное отображение его, то хорошо известно, что в получен-

ном графе G_1 для любой N-перестановки $\phi=\{i_1,i_2,\ldots,i_N\}$ можно выделить N путей, соединяющих вершины верхнего ряда с вершинами нижнего ряда так, что первый путь соединяет первую вершину с i_1-ой, второй - вторую вершину с i_2-ой и т.д., и никакие два разные пути не имеют ни одного общего ребра. Поэтому, если попарно склеить соответствующие вершины верхнего и нижнего рядов графа G_1, то для этого множества вершин /назовем их основными/ полученный граф G_2 является информационным. Это значит, что для любой N-перестановки $\phi=(i_1,i_2,\ldots,i_N)$ в графе G_2 можно выделить N путей так, что первый путь начинается в первой основной вершине и оканчивается в i_1-ой основной вершине, второй начинается во второй основной и оканчивается в i_2-ой основной вершине и т.д., и никакие два различные пути не имеют ни одного общего ребра. Разумеется эти пути проходят не только через основные, но и через вспомогательные вершины.

Чтобы превратить граф G_2 просто в информационный граф с вершинами, надо разбить все его вершины на N групп так, чтобы в каждой группе была одна основная, некоторое число вспомогательных вершин, и "стянуть" такую группу в одну вершину.

Наш граф G_2 имеет $2N\log_2 N$ вершин, следовательно, каждая группа содержит $2\log_2 N$ вершин, и если при разбиении вершин на группы не заботиться, какие вспомогательные вершины попадают в одну группу, то полученный информационный граф будет иметь степень $8\log_2 N$. Если же стараться выделить группы так, чтобы каждая группа вершин была графом ранга $\log_2 n$, то степень "стянутой" вершины может быть уменьшена до $4\frac{\log_2 N}{\log_2 \log N}$. Действительно, разделим граф G_2 на полосы высотой $(\log_2 n-\log_2\log_2 n)$. Каждая полоса, согласно отмеченному выше свойству 2/, распадается на $\frac{N\log_2 \log_2 N}{\log_2 N}$ частей ширины $\frac{\log N}{\log \log N}$. Каждая часть соединяется с остальной частью графа только ребрами, исходящими из верхнего и нижнего рядов. Поэтому, если стянуть такой подграф в одну вершину,

то ее степень будет равна /см. свойство 1/ $4\frac{\log_2 N}{\log_2 \log_2 N}$. Всех частей, на которые распадается граф G_2 , имеется

$$2\frac{N\log_2 \log_2 N}{\log_2 N}\left(\frac{\log_2 N}{\log_2 \log_2 N - \log_2 \log_2 \log_2 N}\right) < 4N \quad ,$$

т.е., если стянуть все такие части к N основным вершинам, то получим /при больших N / информационный граф, имеющий степень по порядку равную $C_2 \frac{\log N}{\log \log N}$, где N - число его вершин. Таким образом, описанная нами конструкция информационных графов позволяет сформулировать следующую теорему.

Теорема 3.2. С точностью до порядка /мультипликативной константы C /

$$\Phi(N) = C\frac{\log N}{\log \log N}$$

§ 4. Синтез структур микропроцессорных систем

С целью решения задачи II, рассмотрим схему из автоматов и B . Функционирование "схемы" - процесс выполнения некоторого алгоритма, а преобразование f_A - функция, вычисляемая этим алгоритмом на множестве B .

Через $f_{\mu_A(a)}$ обозначим преобразование, вызываемое в автомате B выходным сигналом $\mu_A(a)$ автомата A в состоянии a . Пусть преобразование $f_{\mu_A(a)}$ осуществляется в автомате B в момент перехода автомата A из одного состояния в другое.

Через $f^{(j)}_{\mu_A(a)}$ обозначим функцию, вычисляемую на множестве микропроцессором P_j , и представим ее в виде кортежа:

$$\langle P_1, P_2, \ldots, P_N, f^{(j)}_{\mu_A(a)}, P_j \rangle \qquad /4.1/$$

Полагаем, что для выполнения преобразования $\tilde{i}^{(j)}\mu_A(a)$ информация с микропроцессоров $(P_1,P_2,\ldots,P_N)$ последовательно пересылается на микропроцессор P_j, а память микропроцессоров неограничена.

Значение функций $\tilde{i}^{(j)}\mu_A(a)$ на множесте B существенно зависит не от всех аргументов $(P_1,P_2,\ldots,P_N)$. Поэтому функцию $\tilde{i}^{(j)}\mu_A(a)$ представим в виде:

$$\langle \tilde{P}_1,\tilde{P}_2,\ldots,\tilde{P}_N,\ \tilde{i}^{(j)}\mu_A(a),\ P_j\rangle \qquad /4.2/, \text{ где}$$

$$\tilde{P}_i = \begin{cases} P_i, & \text{если значение функции } \tilde{i}^{(j)}\mu_A(a) \\ & \text{существенно зависит от } P_i, \\ 0 & \text{- в противном случае.} \end{cases}$$

Каждому преобразованию $\tilde{i}\,\mu_A(a)$ автомата B будет соответствовать кортеж вида /4.2/. Это преобразование вызывается в автомате B выходным сигналом $\mu_A(a)$ автомата A. Поставим в соответствие каждому выходному сигналу $\mu_A(a)$ кортеж вида:

$$\langle \mu_A^{(1)},\mu_A^{(2)},\ldots,\mu_A^{(n)},\ \tilde{i}\,\mu_A(a),\ j\rangle \qquad , \text{ где}$$

$$\mu_A^{(i)} = \begin{cases} 1, \text{ если } P_i \neq 0 & \text{в выражении /4.3/,} \\ 0 \text{ - в противоположном случае,} \end{cases}$$

j - номер микропроцессора P_j, на котором осуществляется преобразование $\tilde{i}\,\mu_A(a)$.

Обозначим множество кортежей вида /4.3/ автомата A через β_μ.

Определение 4.1.

Автомат $\mathcal{A}_B = \langle \mathfrak{X}, \beta_\mu, \mathfrak{A}, a_o, a^*\delta_A, \mu_{A_B} \rangle$ назовем B - автоматом.

$\mathcal{A}_B$ - автомат содержит информацию о тех и только тех микропроцессорах автомата B, которые необходимы только и только для выполнения преобразования $\hat{\mu}_A(a)$ в состоянии a. В этом же состоянии некоторые микропроцессоры могут хранить информацию для выполнения преобразований в следующих за состоянием a состояниях. С целью выявления таких микропроцессоров введем определение.

Определение 4.2.

Путем P_j занятости микропроцессора P_j в $\mathcal{A}_B$ -автомате назовем такой максимальный путь $P_j=(a_{\kappa 1}, a_{\kappa 2}, \ldots, a_{\kappa t})$, который является отрезком некоторого пути P^*, и такой, что:

$$\mu^{(n+i)}_{A_B}(a_{\kappa 1}) = j, \quad \text{а} \quad \mu^{(j)}_{A_B}(a_{\kappa_t}) = 1,$$

и для каждого a_{κ_i} $(1<i<t)$ $\mu^{(n+2)}_{A_B}(a_{\kappa_i}) \neq j$. Состояние $a_{\kappa 1}$ назовем начальным состоянием пути p_j, а состояние a_{κ_t} - конечным.

Иными словами, путь P_j занятости микропроцессора P_j - максимальный путь, на котором состояниями микропроцессора являются только значения результата преобразования $\hat{\mu}_{A_B}(a_{\kappa_1})$, осуществляемого в начальном состоянии a_{κ_1}, а которое обязательно должно быть аргументами преобразования в конечном состоянии a_{κ_t} пути p_j.

Используя автомат $\mathcal{A}_B$ и определение /4.2/ с помощью методов теории графов можно построить новый $\mathcal{A}_C$ - автомат: $\mathcal{A}_C = \langle \mathfrak{X}, \beta_C, 1, a_o, a^*, \delta_A, \mu_{A_C} \rangle$, в котором через β_C обозначено множество кортежей, определяемых следующим образом:

$$\langle \mu^{(1)}_{A_C}, \mu^{(2)}_{A_C}, \ldots, \mu^{(n)}_{A_C}, \hat{\mu}_A(a), j \rangle \qquad /4.4/$$

где

$$\mu_{A_c}^{(i)} = \begin{cases} 1, \text{ если микропроцессор } P_i \text{ содержит информацию для выполнения преобразования } {}^{i}\mu_{A}(a) \text{ или преобразований, выполненных в следующих за состоянием } a \text{ состояниях } a \text{ ;} \\ 0 - \text{в противном случае.} \end{cases}$$

Пусть задан $\mathcal{A}_C$ - автомат микропроцессорной системы B . Через T_i - обозначим подмножество состояний автомата A , на котором используется микропроцессор P_i - в микропроцессорной системе B .

Определение 4.3.

Графом $Q=(\beta,\Gamma)$ микропроцессорной системы B назовем конечный неориентированный граф с множеством вершин $\beta = \{P_1, P_2, \ldots, P_N\}$, в котором вершины P_j и P_i смежны тогда и только тогда, когда выполняется условие $T_j \cap T_i \neq \phi$.

Определение 4.4.

Отмеченным графом $Q_B=(\beta, \mathcal{F}y, F)$ микропроцессорной системы B назовем такой граф $Q = (\beta,\Gamma)$, в котором каждой вершине P_i приписано множество функций, вычисляемых микропроцессором P_i .

Определение 4.5.

Отождествление двух несложных вершин графа Q /графа Q_B/ назовем элементарным гомоморфизмом ε .

Определение 4.6.

Последовательность элементарных гомоморфизмов $\phi=(\varepsilon_1\ \varepsilon_2 \ldots\ \varepsilon_\kappa)$ графа Q /графа Q_B / назовем гомоморфизмом ϕ графа Q /графа Q_B /.

Определение 4.7.

Гомоморфизм ϕ графа Q /графа Q_B / назовем полным порядка n, если его образ $\phi(Q)$ ((Q_B)) является полным графом на n вершинах.

Наименьший порядок всех полных гомоморфизмов графа Q/графа Q_B / обозначим $\lambda(Q)(\lambda(Q_B))$. В теории графа известна следующая теорема.

Теорема 4.8. Для любого графа Q и его элементарного гомоморфизма ε

$$\lambda(Q) \leq \lambda(\varepsilon Q) \leq 1 + \lambda(Q).$$

Следствие. Для любого гомоморфизма ϕ графа Q

$$\lambda(Q) \leq \lambda(Q).$$

Пусть задан граф Q. Рассмотрим полный гомоморфизм графа Q на граф Q' с порядком $\lambda(Q)$. Отображение определяется разбиением множества вершин β графа Q на $\lambda(Q)$ непересекающихся подмножествах $B_K(K=\overline{1,X(Q)})$ попарно несмежных вершин, т.е.

$$\beta = \bigcup_{K=1}^{\lambda(Q)} B_K \; ; \; B_i \cap B_j = \phi(i \neq j) \qquad /4.5/$$

Условие /4.5/ в силу определения /4.3/ равносильно выполнению условия:

$$T_B = \bigcup_{K=1}^{\lambda(Q)} T_K \; ; \; T_i \; T_j = \phi(i \neq j).$$

Поэтому каждой вершине графа Q может быть сопоставлен микропроцессор в новом автомате B'. Обозначим через -число микропроцессоров в автомате B'.

Таким образом, описанная нами конструкция позволяет сформулировать следующую теорему.

Теорема 4.9. Для микропроцессорной системы B, реализующей функцию f_A, справедлива следующая нижняя оценка:

$$\tau \geq \lambda(Q)$$

Граница $\tau = \lambda(Q)$ достигается при минимизации графа Q. Как правило, микропроцессорная система задается графом Q_B.

Таким образом, теорема 4.9. сводит проблему синтеза микропроцессорной системы B к оптимизации графа Q_B, а проблему минимизации числа микропроцессоров в микропроцессорной системе к минимизации числа вершин графа Q.

Методы оптимизации таких графов известны.

Заключение

Описанный выше подход к построению формальной теории сложных микропроцессорных систем по сути свел эту важнейшую и сложнейшую проблему современной вычислительной техники к решению следующих задач.

1. Выбору минимального числа микропроцессоров микропроцессорной системы.
2. Построению сети связи микропроцессорной системы с выбранным числом микропроцессоров.
3. Синтезу и минимизации управляющего автомата известными методами [4].

Используя аппарат булевых матриц и теории графов, описанный подход позволяет создать системы с перестраиваемой структурой связей, что упрощает переход от одного варианта системы к другой путем их перепрограммирования с помощью современных ЭВМ для проблемно-ориентированных микропроцессорных систем.

В заключение авторы отмечают, что настоящая работа выполнена как развитие идей академика В.М. Глушкова о создании вычислительных систем будущего.

ЛИТЕРАТУРА

1. В.М. Глушков: О применении абстрактной теории автоматов для минимизации микропрограмм, "Известия АН СССР". "Техническая кибернетика", № 1, М., 1964.

2. В.М. Глушков: Теория автоматов и вопросы проектирования структур цифровых машин, журн. "Кибернетика", № 1, К., 1965.

3. В.М. Глушков: Теория автоматов и формальные преобразования микропрограмм, журн. "Кибернетика", № 5, К., 1965.

4. В.М. Глушков: Синтез цифровых автоматов, Физматгиз, М., 1962.

СТЫКОВКА ЭВМ В МНОГОМАШИННЫХ СИСТЕМАХ НА БАЗЕ МИКРО-ЭВМ

В. Хенцлер

НП Роботрон НИЦ, г. Дрезден, ГДР

Появление больших интегральных схем сегодня создает возможность разработки многомашинных систем с экономически допускаемыми затратами и применения их в широких масштабах в народном хозяйстве. При этом затраты на стыковку и ее производительность сильно зависят от выбираемой стратегии совместного включения. Одновременно вид стыковки решающим образом влияет на поведение и надежность всей многомашинной системы.

Основные функциональные единицы микро-ЭВМ, смотря со стороны стыковки, изображены на рис. 1. Ядро микро-ЭВМ состоит из центрального процессорного устройства /ЦПУ/ и из собственной памяти /ПЗУ и ОЗУ/. Коммуникация с внешней средой происходит с помощью приведенных четырех типов устройств /БИС-ов/ для ввода/вывода. Коммуникация ЦПУ с устройствами ввода/вывода и памятью, т.е. внутри микро-ЭВМ, производится через системную магистраль, которая состоит из адресной шины, шины данных и шины управления.

Приведенные на рис. 1 устройства ввода/вывода дальше рассматриваются подробнее относительно их применяемости для стыковок в многомашинных системах.

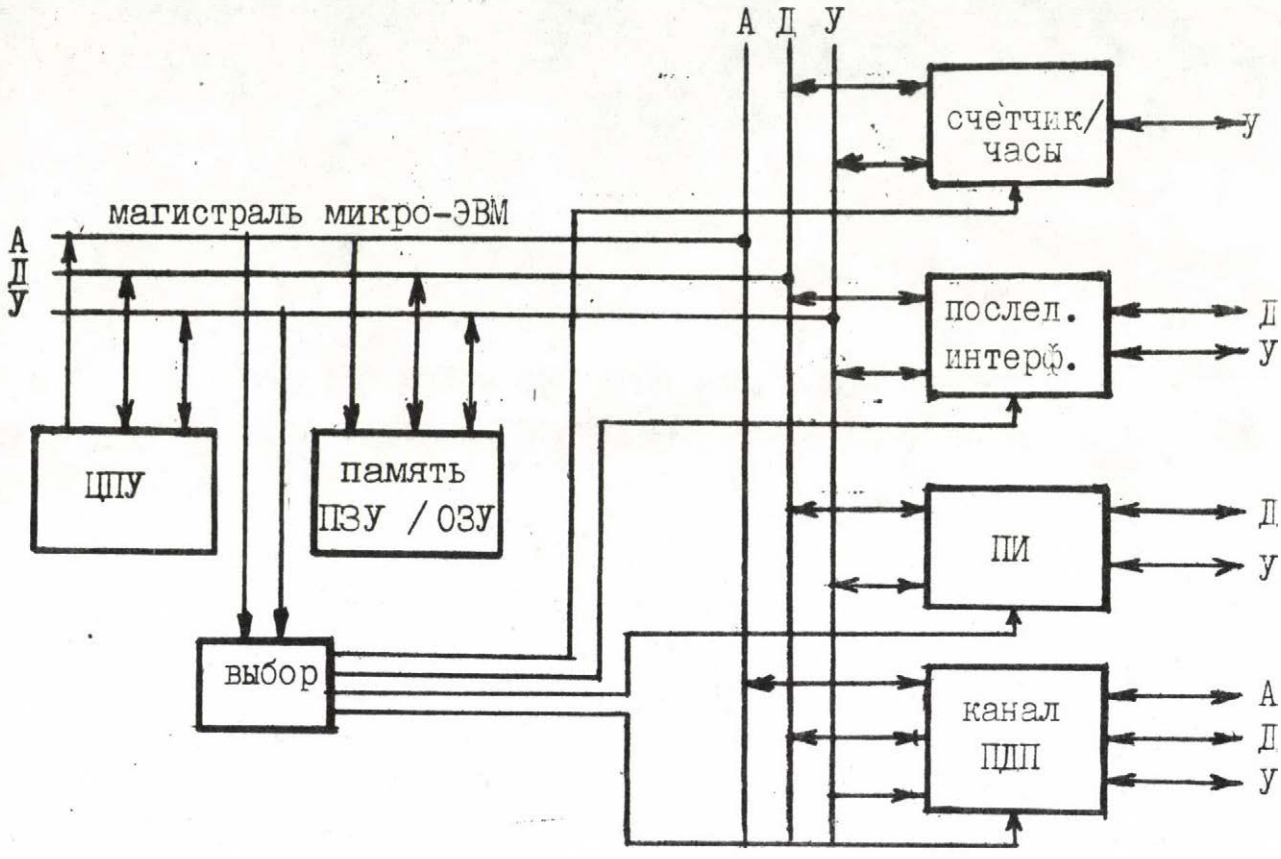


Рис. 1. Блок-схема микро-ЭВМ

1. Устройство счетчика/часов

Самое простое устройство - это устройство счетчика/часов, которое может выдавать к ЦПУ импульсы /прерывания/ в определенных интервалах времени и считывать определенные внешние события /импульсы/. Длительность временного интервала между импульсами и начальное состояние счетчика можно программно задавать и также можно опрашивать текущее состояние счетчика.

Устройство счетчика/часов непосредственно не применимо для стыковки с микро-ЭВМ, так как через него не проходит никакого прямого обмена данными. Несмотря на это, его применяют в многомашинных системах для обеспечения времен реакций и обслуживания.

2. Последовательный интерфейс

Этот интерфейс является традиционным интерфейсом для стыковки ЭВМ между собой, так, например, в вычислительных сетях для телепередачи данных. В области микро-ЭВМ последовательный интерфейс получает особенное значение для подключения медленных внешних устройств.

Для стыковки незначительное количество соединительных проводов последовательного интерфейса особенно выгодно, если между ЭВМ имеются относительно большие расстояния. Это, например, имеет место в децентрализированных многомашинных системах на базе микро-ЭВМ для контроля и управления производственными процессами. Из-за малых затрат на соединения /кабеля/ здесь имеется возможность, существенно повысить надежность линии передачи посредством многократного резервирования, причем вся вычислительная система остается еще экономически выгодной.

За счет передачи незначительного количества информаций во времени первые два вида стыковки годятся только условно для сопряжения ЭВМ в многомашинных системах с большим потоком информаций. Этому требованию в большей степени удовлетворяют следующие два вида стыковки.

3. Параллельный интерфейс

Для параллельного интерфейса характерно, что шина данных подключается полностью параллельно к Общей шине стыковки /ОШС/, а не поразрядно, как при последовательных интерфейсах. Это ведет к значительному увеличению скорости обмена данными, но и сопровождается повышенными затратами на аппаратные средства. Такая стыковка называется и периферийной стыковкой, что обусловливается тем, что устройства ввода/вывода параллельного интерфейса /ПИ/ со стороны микро-ЭВМ обслуживаются как внешние устройства, т.е. ЦПУ должно обращаться к ним с операцией ввода/вывода по определенным адресам. Устройство параллельного интерфейса, при этом, находится в определенном состоянии /го-

тов или занят/; если оно готово, - оно должно быть в состоянии временно запоминать полученные от ЦПУ данные и потом самостоятельно обмениваться с устройством ПИ принимающей ЭВМ сигналы готовности передачи/приема и, наконец, передать данные. Мощность такой периферийной параллельной стыковки в значительной мере зависит от следующих факторов:

1. С какой скоростью микро-ЭВМ может выдавать данные устройству ПИ /т.е. какие временные затраты потребуются, чтобы пересчитать счетчики адреса и длины массива данных и чтобы организовать некоторое условие окончания обмена/.

2. Содержатся ли в устройстве ПИ специальные аппаратные средства для управления самостоятельным обменом данными.

3. Сообщает ли устройство ПИ ЦПУ об окончании передачи или ЦПУ приходится анализировать признак окончания передачи программным путем. То же касается и сигнала окончания приема данных.

Основной элемент ПИ представляет собой регистр, выходы которого подключаются через усилитель с третьим высокоомным состоянием к шине стыковки. В зависимости от вида управления обменом данными к этой схеме надо еще добавить логику для управления обменом данными и логику прерывания. Современная схемотехника выдвигала для этого уже удобные БИС-ы для ввода/вывода с внешними устройствами, но они до сих пор еще не пригодны для нужд стыковки микро-ЭВМ между собой.

Основное звено ПИ представлено на рис. 2. Блок-схема многомашинной системы со стыковкой через БИС-ы устройства ПИ изображена на рис. 3. Здесь устройства ввода/вывода /ПИ/ всех ЭВМ соединяются через двухнаправленную Общую шину стыковки /ОШС/ и шину управления стыковкой. При этом ОШС, кроме шины данных, содержит и линии управления обменом данными. Управление и распределение ОШС принадлежит одной ЭВМ, которая станет, таким образом, управляющей. В зависимости от места этой ЭВМ в многомашинной системе, т.е. от места в информационном пото-

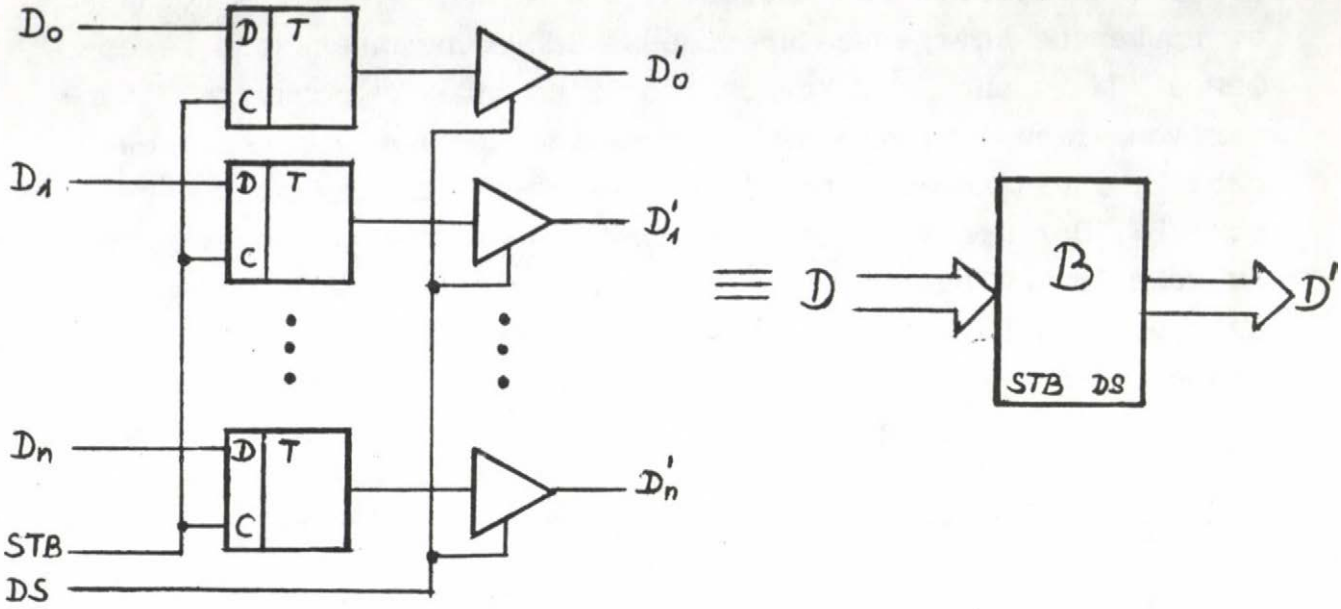


Рис. 2. Регистр/усилитель с высокоомным выходом /В/.

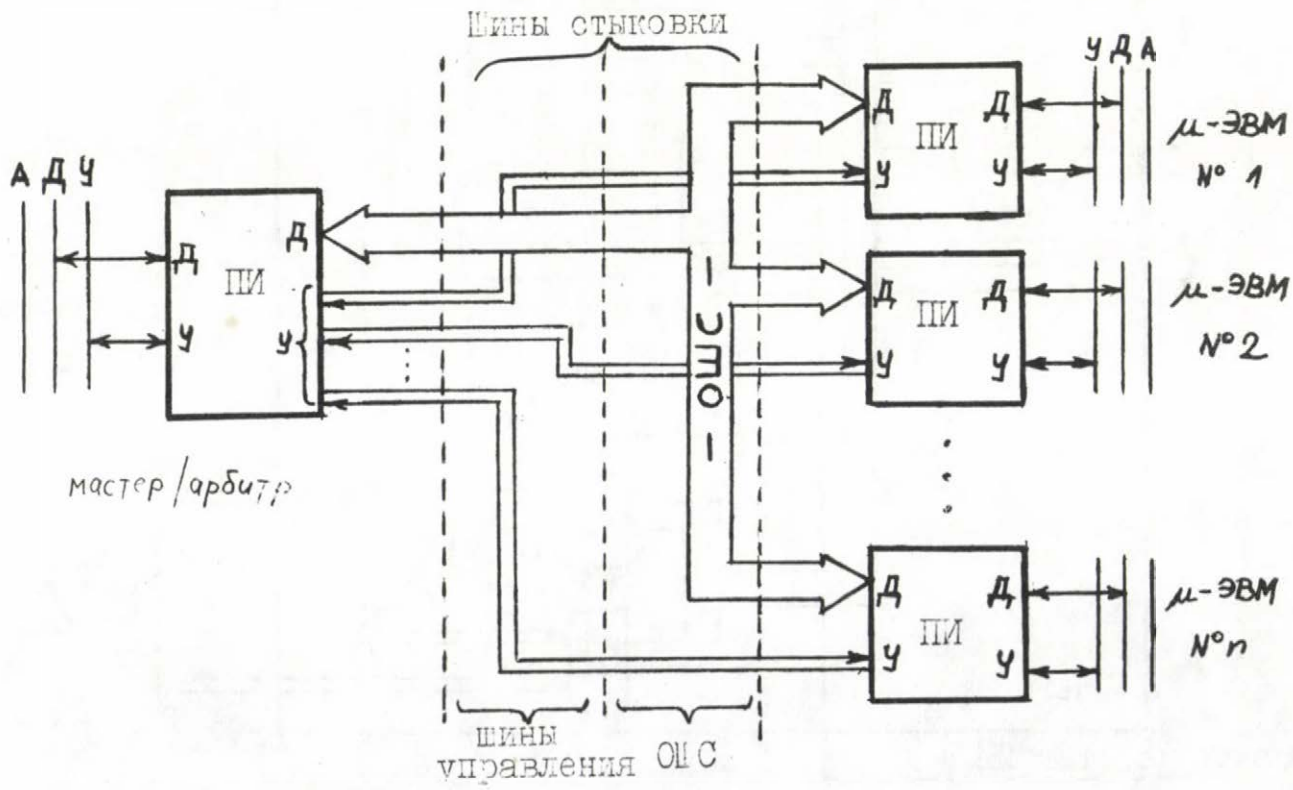


Рис. 3. Блок-схема стыковки микро-ЭВМ через устройства ПИ.

ке, она называется или мастером, или арбитром. Рис. 4 показывает примерное внутреннее построение БИС-а параллельного интерфейса /ПИ из рис. 3/. Устройство ПИ содержит 4 регистра, включая усилители с третьим высокоомным состоянием соответственно рис. 2, управляемые логикой управления ПИ. Два из этих регистров /$В_1$, $В_2$/ предназначены для двухнаправленного обмена данными между шиной данных микро-ЭВМ и шиной данных ОШС. Остальные два регистра служат для приема /$В_4$/ и выдачи /$В_3$/ сигналов запроса и ответа для эксплуатации ОШС. Логика управления ПИ содержит, кроме устройства управления, регистры состояния и режи-

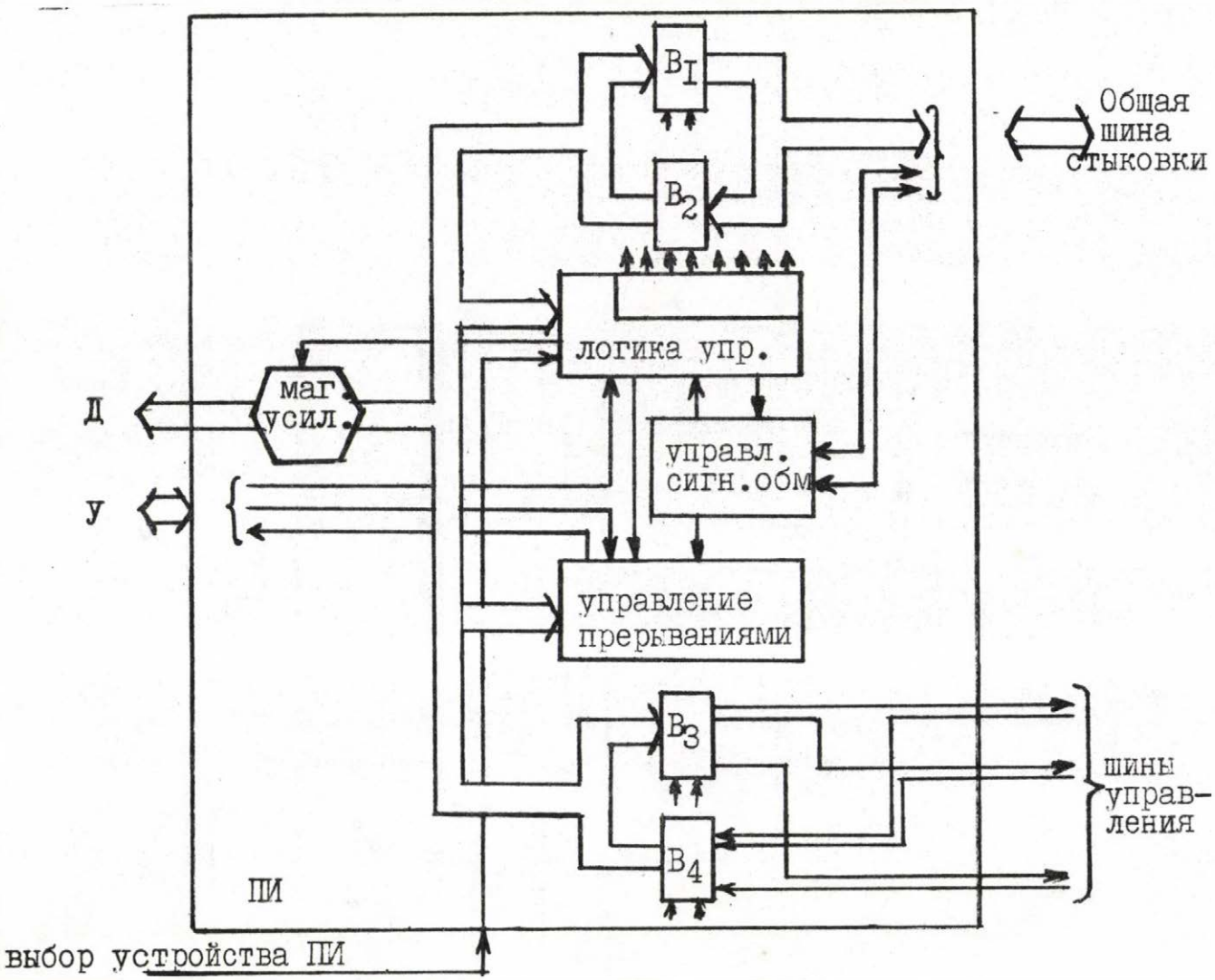


Рис. 4. Блок-схема устройства /БИС-а/ параллельного интерфейса ПИ.

мов работ. Выдача и ввод данных осуществляется с помощью сигналов "готов" и "строб" линии управления ОШС посредством логики управления обменом данными. Логика прерывания обеспечит генерирование требуемых сообщений об окончании выаода или запросов ввода для ЦПУ через шину управления системой магистрали микро-ЭВМ.

В более простом варианте параллельного интерфейса /только применяя аппаратные средства из рис. 2 без логики управления/ выше приведенные сигналы управления должны быть анализированы и генерированы программным путем, в результате чего значительно уменьшается скорость обмена данными.

4. Прямой доступ к памяти

Все пока описанные варианты стыковки не достигали теоретически возможную скорость обмена данными, максимальная величина которой определяется временем выработки данных из устройств памяти. Но выигрыш в скорости передачи при прямом доступе к памяти /ПДП/ связан с повышенными затратами на аппаратные средства. Обычно принцип прямого доступа к памяти реализуется подключением всей системой магистрали микро-ЭВМ /адресной шины, шины данных, шины управления/ к ОШС /смотри рис. 5/. Кроме схем канала ПДП, необходимы еще регистры для выдачи запроса и для получения сигнала предоставления ОШС как и при стыковке через параллельный интерфейс ПИ. В зависимости от выбираемого режима работы регистры обмена данными /B_1, B_2/ и логика управления обменом данными схмы ПИ /рис. 4/ могут применяться или не применяться. Блок-схема канала ПДП из рис. 5 изображена на рис. 7 и 8 и для двух вариантов прямого доступа к памяти. Рис. 6 показывает основной элемент канала ПДП, представляющий собой двухнаправленный управляемый вентиль.

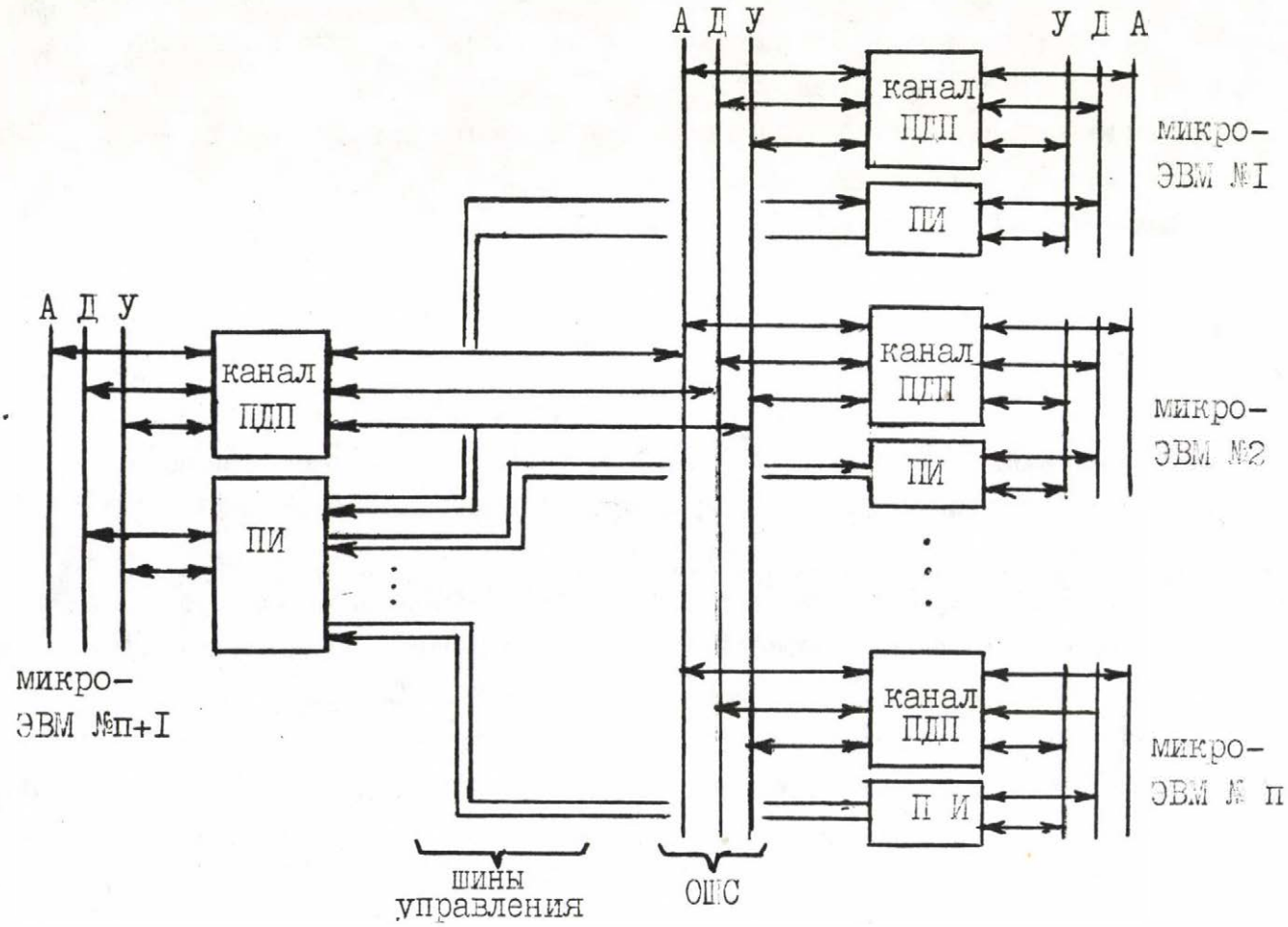


Рис. 5. Стыковка микро-ЭВМ через прямой доступ к памяти.

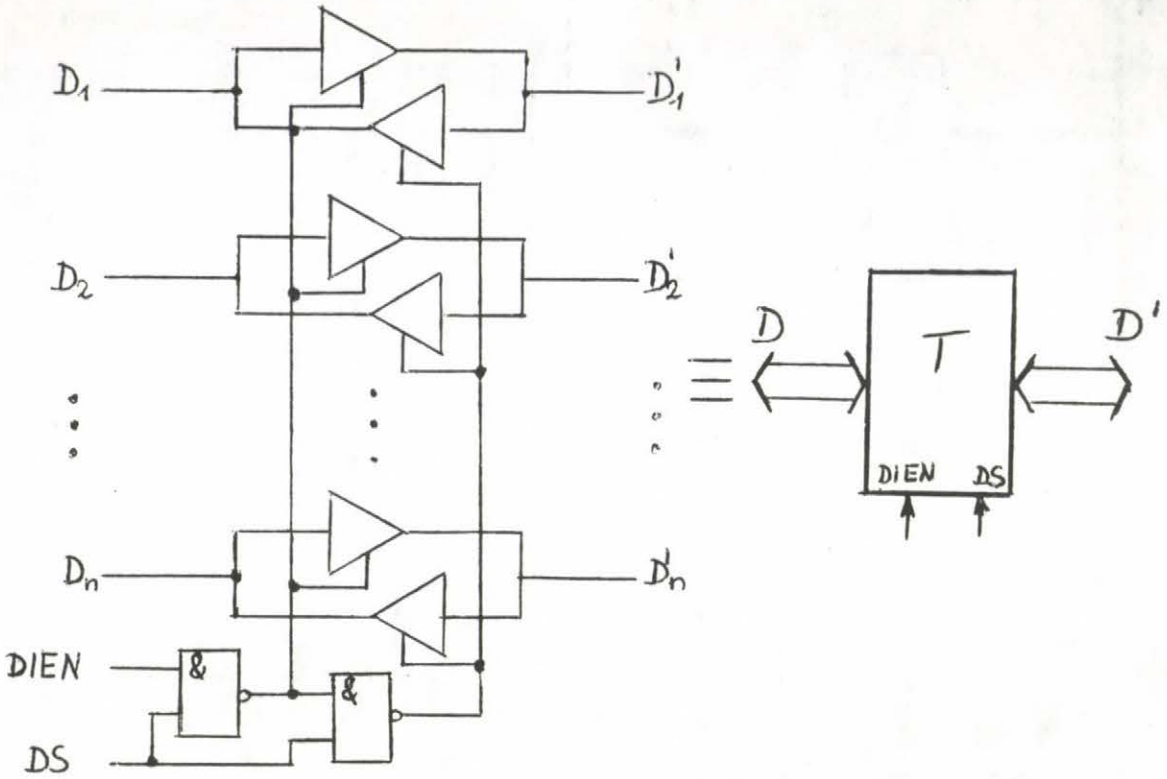


Рис. 6. Основной элемент канала ПДП.

На рис. 7 указано более простое, но и более медленное, исполнение канала ПДП, при котором системная магистраль одной ЭВМ полностью подключается к системной магистрали другой ЭВМ. Для этого необходимо обеспечить, чтобы до подключения ЦПУ к одной микро-ЭВМ оно отключалось бы от своей системной магистрали и перешло бы в пассивное состояние. Другое ЦПУ получит тогда управление обоих системных магистралей и программно управляет обмен данными между обемими системами. Так как в данном случае стыковки требуется еще больший объем организационных затрат максимальная скорость обмена данными не достигается. Скорость более близкая к максимальной достигается с применением дополнительной схемы - т.е. специальное устройство управления /УУ/ канала ПДП /см. рис. 8/.

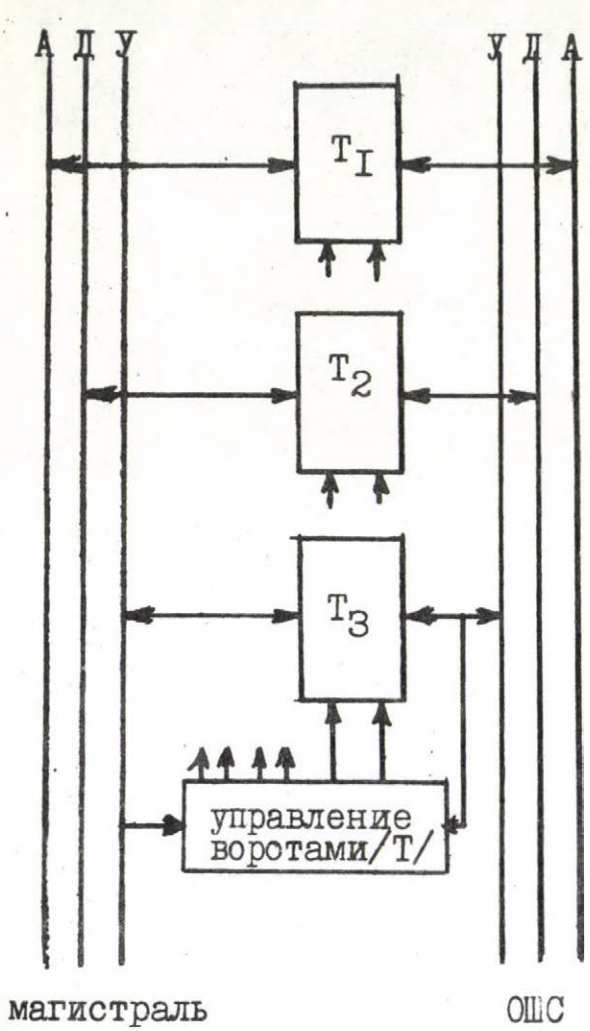


Рис. 7. Канал ПДП, управляемый программно от ЦПУ.

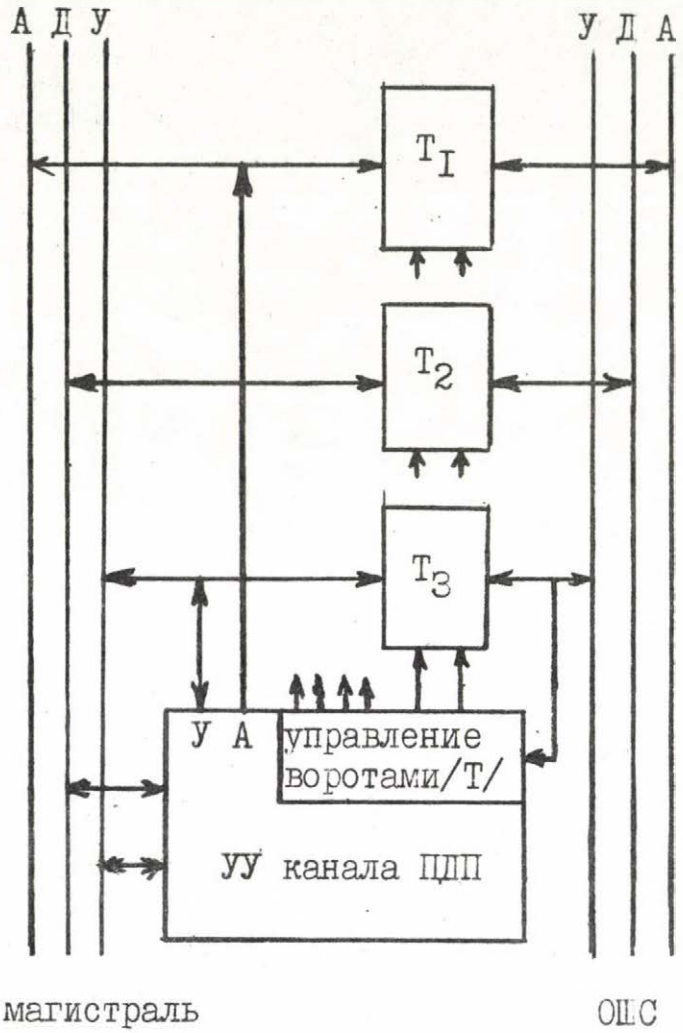


Рис. 8. Канал ПДП, управляемый от УУ канала ПДП.

Это УУ канала ПДП содержит один или несколько счетчиков длины массива, один или несколько счетчиков адресов, логику для управления сигналами шины управления и логику управления, содержащую также регистры режимов работ и состояния /см. рис. 9/.

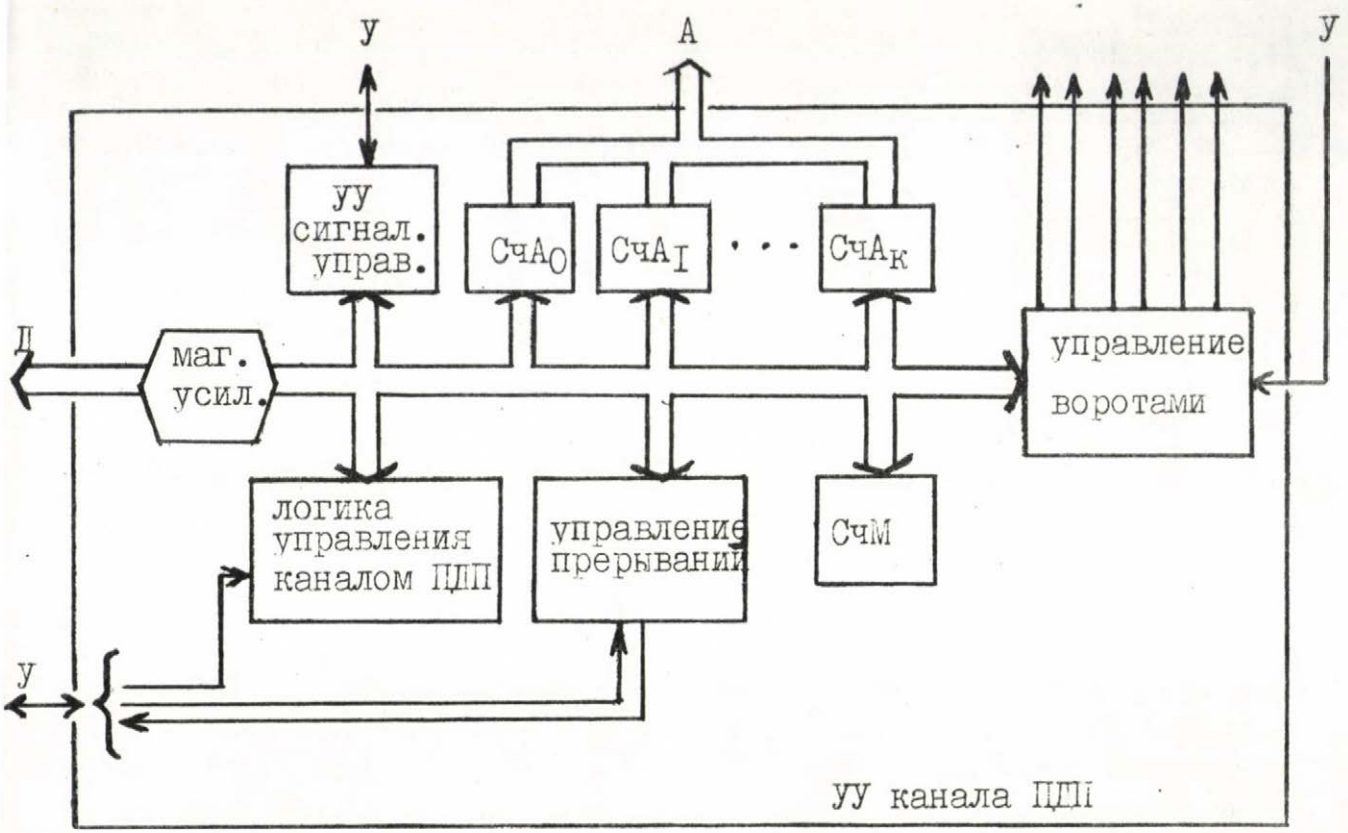


Рис. 9. Устройство управления канала ПДП
СчА - счетчик адресов
СчМ - счетчик длины массива данных.

Оценка производительности

Приведенные в пунктах 2÷4 варианты стыковки отличаются основным образом тем, в какой ширине они подключают системную магистраль микро-ЭВМ к Общей шине стыковки. Эти варианты стыковки, как видно из таблицы 1, охватывают очень широкий спектр производительности обмена данными.

Таблица 1.

Вид стыковки	Режим обмена данными	Скорость обмена в кбайт/сек
Последовательный интерфейс	асинхронный	2
	синхронный	6
Параллельный интерфейс	программно управляемый	10
	аппаратно управляемый	40
Канал прямого доступа к памяти	программно управляемый	120
	управляемый от УУ канала ПДП	625 /1250/

Приведенные в табл. 1 значения для скорости обмена данными являются максимально достигаемыми с микро-ЭВМ шириной 8 бит в н-МОП-технологии с частотой такта примерно в 2,5 МГц. Последнее значение в табл. 1 в скобках указывает на пределы скорости обмена данными за счет устройства памяти. Это значение 1,25 Мбайт/сек достижимо только с применением быстродействующих схем ОЗУ, имеющих время обращения примерно в 60÷100 нсек.

ЛИТЕРАТУРА:

1. H. Belim, A. Sauer:

Methods of data exchange between microcomputers

Microcomputer architectures -
3d EUROMICRO symposium on microprocessing and microprogramming - October 3-6, 1977, Amsterdam

Preprints, pp. 16-22;

2. V.A. Titus, P.R. Rony, D.G. Larsen:

Microcomputer interfacing: accumulator I/O versus memory I/O

Computer Design, June 1976, pp. 114-116.

LOGIC-BASED DESCRIPTION OF MICROCOMPUTERS

G. DÁVID, S. KERESZTÉLY, I. LOSONCZI, A. SÁRKÖZY
Computer and Automation Research Institute of HAS
Budapest

1. INTRODUCTION

In this conference we want to describe in detail our project on the automazitaion of microprogramming and programming of microcomputers. Our approach to the automatization is based on mathematical logic, using a set of symbolic statements and logical deductions associated with them, see [1].

The classical first-order predicate logic is cumbersome for formalizing the description, hence we developed a new mathematical logic, the *Structure Logic* SL [2], in which one can describe structures and instructions as mappings of structures onto structures. A mathematical logic-based automatization consists of the description of the hardware (architecture) and a mechanical theorem proving technique by which the system will generate programs. Accordingly we have devided this introduction to our project into two parts: the description of microcomputers will be given in this paper and the microprogram-synthesis will be presented in the session on programming aspects [3].

Our goal is the description of microcomputers. For this purpose we have constructed a language. What is required of such a language?

- It is to have correct semantics, which in our case is ensured, as our language is based on Vienna Definition Language and Mathematical Logic.

- It should be suitable for the description of microcomputers, so that it should ease structured 'Micro' Programming. It is to be simple, based on a few basic notions and symbols, such as (, ; .).

- The language is to be near to the user's way of thinking, so that he can define his problem without giving an algorithm for it.

To reach our general we shall speak of the following:

- Structures
- Contents of structures
- Operations on structures
- Types
- Instructions

2. *STRUCTURES*

First we give a strict description of the *Structure Logic* SL Language SLL in the Bachus-Naur Form.

Concetenation is denoted by ".", sets are bracketed by { }, and "selector" represents a symbol standing on a place of the selector, hence it is implicity declared as selector.

```
letters::=A|...|Z; digits::=0|...|9;
externals: constant|reference|formulae|instruction|
           selector|structure|
           bit(n)|<|>|[|]|(|)|:|,|;|logical connectives;
logical connectives::=unuary-logical-connectives|
                      binary-logical-connectives;
unuary-logical-connectives::=¬ ;
binary-logical-connectives::=→,V,Λ;
alphanumeric::=digit|letter|letter.alphanumeric|
              digit. alphanumeric;
list::=letter|letter.alphanumeric;
set { };
concatenation .;
selector::=sel|"selector";
selector-domain::=[lower-bound:upper-bound];
lower-bound::="selector" ;
upper-bound::="selector";
constant-declaration::=constant list-of-symbols;
selector-expression::=["selector"]|["selector"].
                      selector-expression ;
basic-type::={bit(n),n≥1}|instruction|selector;
type-declarator::=structure ;
type-name::=symbol ;
type-declaration::=type-declarator.
                  .type-name<{"selector":struc-type-ref}>;
struc-type-ref::=type-ref|struc-ref ;
type-ref::=basic type|type-name|type-name.
          .selector-expression|type-expression ;
type-expression::=type-name<{"selector":type-reference}>;
structure-reference::=structure-name|
                      struc-name.selector expression|
                      struc-expression;
structure-declarator::=type-reference|reference(type-name);
```

```
struc-declaration::=struc-declarator.list-of-structure-names;
structure-name::=symbol
structure-expression::=structure-name.
                      .<{"selector":structure-reference}>;
function-declaration::=type-reference.function.function symbol.
                      .specification;identities-declaration,body;
function-symbol::=function-name.(list-of-formal-parameters)
function-name::=symbol;
formal-parameters::=symbol;
specification::= declaration
identities-declaration::=identities list-of-identities;
identity::=term.=.term;
term::=symbol|function name(list -of-terms);
body::=begin.local declaration;list-of-structure assignments.end;
local-declaration::={declaration};
structure-assignment::=left-part.:=right-part;
left-part::=structure-reference|function-name;
right-part::=structure-reference|term;
predicate-declaration::=predicate.structure-expression;
statement::=structure expression.binary-logical-connective.
           .structure-expression|unary-logical-connective.
           .struc-expression;
declaration-part::={declarations};
declaration::=type-declaration|structure-declaration|
              constant-declaration;
description::=formulae{statements};
problem-specification::=statement;
Abbreviations:sel,inst,struc,ref;
```

What does this description cover?

By structures we mean objects and hierarchical interconnections between them. Instructions are mappings of structures:

$$i: s_1 \rightarrow s_2.$$

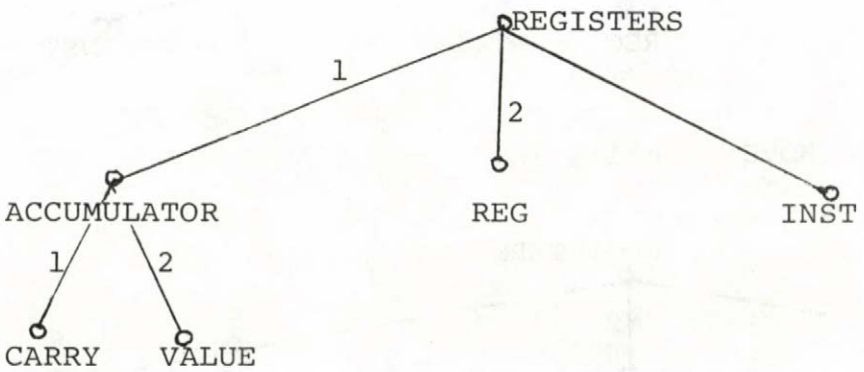
To describe the hierarchy of structures we use selectors. The selectors of one structure must be different. Instead of declarations the selectors are allocated by < > signs in the following form:

structure-name <{ $selector_i$: $substructure_i$ }>

For instance:

REGISTERS <1:ACCUMULATOR,2:REG,3:INST>
ACCUMULATOR <1:CARRY,2:VALUE>

The meaning of this can be illustrated as follows:



We can refer to the substructures as well:

e.g. REGISTERS[1] =ACCUMULATOR
REGISTERS[1] [2] = VALUE

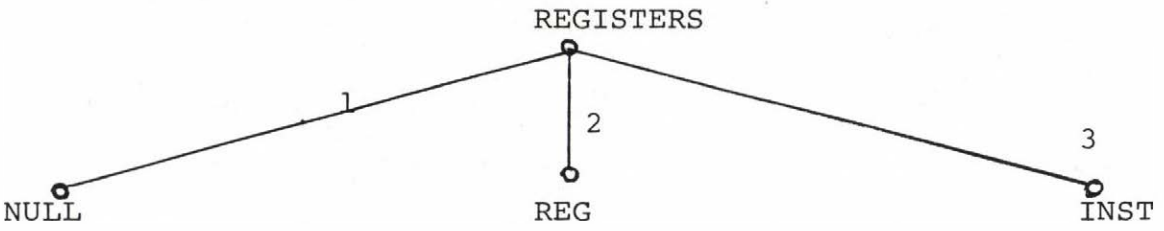
A structure is called homogeneous if it has substructures of the same type. In this case we can condense our notion:

structure-name<[1:n]:substructures>

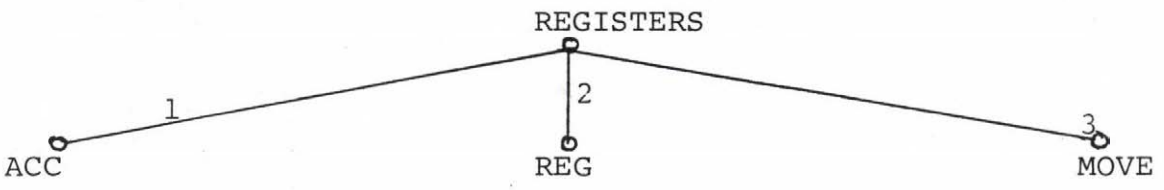
where [1:n] stands for a set of selectors.

Following the VDL notation we shall now present the operations on structures. If we write S<sel:S^*> then the structure S has a substructure S^*, that is if S has had a substructure connected to S by the selector sel before, then it is changed by S^*, if not, a new selector sel is made and S^* is put on it.

Referring to the above-mentioned example REGISTERS<1:NULL> results in the following:



Again REGISTERS <3:MOVE> results in:



As can be seen only those-selectors which are changed need to be written. So REGISTERS < > would mean no change at all.

3. TYPES

Speaking of homogeneous structures it was mentioned earlier that each structure has a type. The folloiwng rule applies: only structures of the same type can be substituted for each other. In this way instructions with no meaning are eliminated; for instance to move data from MEMORY to the CARRY bit, when one would not know which part of the data is to be stored and which part is to be dropped.

We have three basic types, such as:

- bit(n)
- selector
- instruction

Every structure has to have declared what type it belongs to.

Examples:

```
bit(1)CARRY;
bit(4)REG,VALUE;
selector j;
instruction MOVE,EXCH;
```

New, structured types can be defined from these basic types:

type-name<{ selectors:sub-types}>;

Example:

```
REGIST-TYPE<1:bit(5),2:REG-TYPE,3:instruction>;
REG-TYPE<[1:8]:bit(4)>;
```

In this way "new standard types" which also can be used for declaration are obtained.

As has been shown, homogeneous types are treated in a similar way to homogeneous structures. In a declaration we allow the use of structures, the type of which has already been defined. This can be important if we want to emphasize that two different structures have the same substructure.

For instance:

If we have a machine structure with a 4-bit accumulator and eight 4-bit index registers

```
structure MACH <1:ACC,2:REG-TYPE,3:s;>
bit(4) ACC;
REG-TYPE<[1:8] bit(4)>
reference(MACH) MACH 1, MACH 2; or
'MACH' MACH 1, MACH 2;
reference(instruction)s;
```

respectively means that MACH 1 and MACH 2 will always execute the same instructions.

Thus we have shown how to declare compound structures using the compound types already defined.

To summarize the last paragraph there is no restriction as to the sequence of declarations, but it is important that by the end there should be no structure left without a declaration of type.

4. CONSTANTS, FUNCTIONS

Any identifier having been already declared can be included in the constant list. Constants are not evaluated, so if we use the symbol 7 for a constant it does not mean that the value of it is seven. On the contrary, we can give values to the constants and in this way the value of 7 can be, form example, three. The list of constants is as follows:

```
constant VALUE,5,REG,
         ACC/<1:12>/, 7/3/;
```

This means that the substructure connected to the structure ACC by the selector 1 has the value 12 and the symbol 7 has the value 3.

Similarly, functions are not evaluated either, unless all of their parameters are constants with defined values. To declare a function, its type, name and parameter list must be given. A function transforms structures of its parameter list into a structure having the function's type. The list of identities and the body of the function belong also to the declaration of functions (see the syntax).

It should be noted that in a logic-based description language the various symbols used are not evaluated - they do not represent values unless it is explicitly stated. The description of a machine will be handled symbolically and the list-of-identities should be treated as symbolic equations. This means that (in the following example) if somewhere

```
P<i:x>
```

stands, it will be equivalent to

```
P<i:RAL(RAL(RAL(RAL(RAL(x)))))>
```

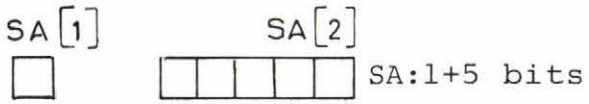
or P<j:RAL(1)> is equivalent to P<j:2>.

Let us look at an example of function declaration. This function defines the instruction Rotate Left by One.

```
bit(5) function RAL(X); bit(5) X;
       identities RAL(0) = 0,
                  RAL(1) = 2,
                  RAL(RAL(RAL(RAL(RAL(X))))) =X;
begin
      structure α <1:bit(1),2:bit(5)>,δ<1:bit(5),2:bit(1)>;
      ref(α) SA;ref(δ)SD;equivalence (SA,SD);
      SD[1]:=X; SD[2]:=SA[1];
      RAL:=SA[2];
end;
```

The"*bit(5) function* RAL(X); *bit(5)*X;" declares RAL as a five-bit operation in both input X and output RAL(X). The *"identities"* part states that five successive "RAL"-s are equivalent with a no-operation NOP and also gives facts about the operation on the objects 0 and 1.

The body part starts with declaration of local structures: SA, SD. They are equivalenced, so they refer to the



same object but with different internal structures depending on the point of view.

SD[1]:= X states that the higher 5 bits contain the variable to be rotated. SD[2]:=SA[1] means that the lowest bit will be filled with the highest one. RAL:=SA[2] gives us the required result.

There is a possibility of declaring references to different elements as equivalent ones. This equivalence may be considered as a function, where the body and the list-of-identities are

empty. We declare equivalent structures by

equivalence (list-of-structures);

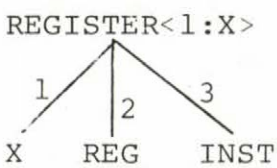
5. DESCRIPTION OF INSTRUCITONS

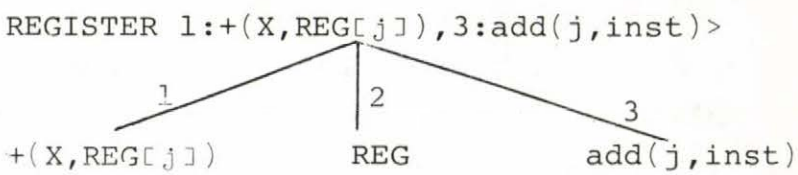
Having structures, types and functions we can describe instructions by giving the state of structures before and after them.

E.g. REGISTERS<1:X>→REGISTERS<1:+(X,REG[j]),3:add(j,inst)>

or the equivalent form: REGISTERS→
→REGISTERS<1:+(REGISTERS[1],REG[j]),3:add(j, inst)>

This statement can be interpreted as follows: the instruction add(j) can be executed in an arbitrary state of the machine described by REGISTERS <1:X> and after the execution of add j the state will be changed to REGISTERS<1:+(X,REG[j]), 3:add(j,inst)>





Syntactically the set of statements describing all the instructions should be declared by

formulae {statements}.

To make it clear we have to stress the difference between the instruction declaration and instruction description.

In the instruction declaration we declare some of the function symbols as instructions and in the parameter-list we write the appropriate types of parameters:

instruction move (*selector, instruction*)

For technical reasons we include one parameter of *instruction type*.

move(j,s)

where s has been declared elsewhere as an instruction-symbol. This parameter will be used in automatic program generation.

The description of the instructions is nothing else but the description of the changes in the relevant structure, effected by the execution of the instrucitons in question.

For the sake of simplicity we introduce a declaration

predicate structure-expression;

e.g. *predicate* MC <1:A,2:R,3:M,4:SRC,5:P,6:s> ;

for those structure-names which will be used in the instruction--description part. Here we can associate substructures having already defined types with a structure-name in order to ease the notation used in the description-part.

Let us examine out requirements to see whether they are satisfied or not.

- The semantics of this language are correct, as Structure Logic is based upon many-sorted logic which is a mathematical apparatus and completed with selectors.
- By means of structures it ensures the possibility of structured programming.
- It is simple: there are only 3 basic types and the signs [<:>] /;, used.

- The language makes automatic microprogramming possible without the need for giving an algorithm, but this question is outside the scope of this paper.

6. AN EXAMPLE

Having discussed the Structure Logic Language, let us look at the description of the microprocomputer MCS-4 as a paradigm. Naturally we have to keep in sight only those parts of the machine which are modified or referred to by instructions.

The MCS-4 has a 4-bit accumulator and a carry flag. An SRC register of one byte (8 bits) belongs to it.

CY	ACC	SRC
Carry	Accumulator	SRC

0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15

The sixteen 4-bit long index registers belongs to the Central Processor Unit as well. These can be used in the form of eight 1-byte long words too.

Regarding the memory, at the moment we are interested only in RAMs and not ROMs. The RAM consists of 256 4-bit long words from among which we can choose by means of the contents of the SRC register.

There are 16 input-output ports with 4-bit long words.
These can be addressed by the four lower bits of the SRC register.

The description begins with the declarative part, where we describe the structure of the machine, that is the structure of the hardware.

```
structure α<[O:15]:bit(4)>,β<[O:7]bit(8)>,
          γ<[O:255]:bit(4)>,δ<1:bit(1),2:bit(4)>,
          ε<[O:2]bit(4)>,
¶<1:ref(α),2:ref(α),3:ref(γ),4:bit(8),5:ref(α),6:instruction>
bit(1) CY/O/,C/1/;     bit(5)AC;
bit(4) K,ACO/O/,KO/7/,9/9/;
bit(8) SRC,const;
ref(α)R,P;ref(β)RR;    ref(γ)M;   ref(δ)A;
ref(¶)MC;ref(ε)SR;
equivalence (R,RR),(SRC,SR);(A,AC);
predicate MC<1:A,2:R,3:M.4:SRC,5:P,6:S>;
instruction S,NOP(inst),FIM(sel,bit(8),inst),
            SRC(sel,inst),LDM(bit(4),inst),
            LDR(sel,inst),XCH(sel,inst),
            INC(sel,inst),ADD(sel,inst),
            SUB(sel,inst),WRM(inst),
            RDM(inst),SBM(inst),ADM(inst),
            WRR(inst),RDR(inst),CLB(inst),
            CLC(inst),IAC(inst),CMC(inst),
            CMA(inst),RAL(inst),RAR(inst),
            TCC(inst),DAC(inst),
            TCS(inst),STC(inst).
bit(5) function ral(X);bit(5)X;
       identities ral(O)=O, ral(1)= 2,
                  ral(ral(ral(ral(ral(X))))) = X;
       begin
            structureα<1:bit(1),2:bit(5)>,δ<1:bit(5),2:bit(1)>;
            ref(α)SA,ref(δ)SD; equivalence(SA,SD);
            SD[1]:=X; SD[2]:=SA[1],
            ral:=SA[2];
      end;
```

```
bit(5) function rar(X); bit(5)X;
       identities rar(0)=0, rar(2)=1,
                  rar(rar(rar(rar(rar(X))))) =X;
       begin
            structureα<1:bit(1),2:bit(5)>,δ<1:bit(5),2:bit(1)>
            ref(α)SA,ref(δ)SD; equivalence(SA,SD);
            SA[2]:=X; SA[1]:=SA[2];
            rar:=SD[1];
       end;
bit(4) function inc(X);bit(4)X;
       begin
            structure α<1:bit(1),2:bit(4)>
            bit(5)S,ref(α)SA; equivalence(S,SA);
            S:=X+1;
            inc:=SA[2];
       end;
```

Having defined the hardware we can now turn to the instructions. The trick of the description is that the formulae include the instructions, although in the machine the instructions do not exist. This makes program generation possible for us.

```
formulae MC→MC<6:NOP(s)>
         MC→MC<2:RR<i:const>,6:FIM(i,const,s)>
         MC→MC<4:RR[i],6:SRC(i,s)>
         MC→MC<1:<2:K>,6:LDM(K,s)>
         MC→MC<1:<2:R[i]>,6:LDR(i,s)>
         MC→MC<1:<2:R[i]>,2:<i:A[2]>,6:XCH(i,s)>
         MC→MC<2:R<i:inc(R[i]) ,6:INC(i,s)>
         MC→MC<1:R[i]+A[2]+A[1],6:ADD(i,s)>
         MC→MC<1:A[2]-R[i]-A[1],6:SUB(i,s)>
```

```
MC→MC<3:<SRC:A[2]>,6:WRM(s)>
MC→MC<1:<2:M[SRC]>,6:RDM(s)>
MC→MC<1:A[2]+M[SRC]+A[1],6:ADM(s)>
MC→MC<1:A[2]-M[SRC]-A[1],6:SBM(s)>
MC→MC<5:<SR[2]:A[2] ,6:WRR(s)>
MC→MC<1:<2:P[SR[2]]>,6:RDR(s)>
MC→MC<1:<1:CY,2:ACO>,6:CLB(s)>
MC→MC<1:<1:CY>,6:CLC(s)>
MC→MC<1:A[2]+1,5:IAC(s)>
MC→MC<1:<1:A[1]+1>,6:CMC(s)>
MC→MC<1:<1:C>,6:STC(s)>
MC→MC<1:<2:KO-A[2],6:CMA(s)>
MC→MC<1:rar(AC),6:RAR(s)>
MC→MC<1:ral(AC),6:RAL(s)>
MC→MC<1:A[2]-1,6:DAC(s)>
MC→MC<1:9+A[1],6:TCS(s)>
MC→MC<1:ACO+A[1],6:TCC(s)>.
```

Now, what sort of instructions do we have?

NOP	- does not effect anything
FIM i,const	- loads the constant "const" (8-bit long) into the i-th register pair
SCR i	- loads the contents of the i-th register pair into the SRC register
LDM K	- loads the constant "K" (4-bit long) into the accumulator
LDR i	- loads the contents of the i-th register into the accumulator
XCH i	- exchanges the contents of the i-th register and the accumulator
INC i	- increments the contents of the i-th register by one. In case of overflow the i-th register is set to zero
ADD i	- adds the contents of the i-th register to the contents of the accumulator
SUB i	- subtracts the contents of the i-th register from the contents of the accumulator

WRM - writes the contents of the accumulator into the memory register previously selected by by the SRC

RDM - writes the contents of the memory register previously selected by the SRC into the accumulator

SBM - subtracts the contents of the memory register previously selected by the SRC from the contents of the accumulator

ADM - adds the contents of the memory register previously selected by the SRC to the contents of the accumulator onto the output part previously selected by the four lower bits of the SRC register

RDR - reads from the input part previously selected by the four lower bits of the SRC register into the accumulator

CLB - clears the carry bit and the accumulator

CLC - clears the carry bit

IAC - increments the contents of the accumulator by 1, an overflow sets the carry bit

CMC - complements the contents of the carry bit

CMA - complements the contents of the accumulator

RAL - rotates the contents of the accumulator and the carry bit left by one

RAR - rotates the contents of the accumulator and the carry bit right by one

TCC - the accumulator is cleared. The least significant position of the accumulator is set to the value of the carry bit. The carry bit is set to 0.

DAC - the contents of the accumulator is decremented by one. A borrow sets the carry bit to one.

TCS - the accumulator is set to 9 if the carry is 0 and to 10 if the carry bit is 1. The carry bit is set to 0.

STC - the carry bit is set to one.

REFERENCES

[1] Chin-Liang Chang, Richard Char-Tung Lee: Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973. New York

[2] G. Dávid, S. Keresztély, A. Sárközy: Program Synthesis by theorem proving in Structure Logic SL, II. Hung. Comp. Sci. Conference, 1977., part 1, pp. 291-310

[3] G. Dávid, S. Keresztély, I. Losonczi, A. Sárközy: Microprogram synthesis (in this issue)

CONSIDERATIONS FOR IMPLEMENTING A MICRO-BASED MINICOMPUTER

S. EBERGÉNYI, L. LEVELKI, G. MESSING, M. SZALAY

Central Research Institute for Physics

Budapest

INTRODUCTION

Advanced LSI technologies seem to have revolutionized computer science and the associated technology. The rapid development in the field of microprocessors, processor slices and highly integrated accessories provided a great variety of different micro-circuits.

In connection with two concrete implementations of micro-based minicomputers we intend to describe some criteria for selecting the processor and defining the structure and architecture of the computer.

BASIC CONSIDERATIONS

There exists extensive literature dealing with microprocessors; they are handled from different aspects such as semiconductor technology, circuit density, speed, architecture, electrical features and organization. All these technical parameters are very important in starting a new development, but first of all one has to consider less technical aspects like the purpose of the computer, the time available for the development and whether large or small production series are intended, to list but a few.

Our Institute has been producing for some years the now the minicomputer TPA-i, which is a twelve bit single address machine common on a software level with the PDP 8. Numerous small, as well as comparatively large, systems have been built based on the TPA-i for industrial as well as for laboratory purposes. A quite large set of peripheral controllers is available for conventional peripherals such as disks, lineprinter, tape, etc. as well as for CAMAC [2]. Software such as OS-i, RTS, etc. is utilized.

Our aim has been to modernize this SSI-MSI system considerably by maintaining complete software compatibility.

In our case, probably the most important factor has been the safeguarding of existing measuring technique and computer investments. The importance lies not only in the reduction of system development time (as some of the necessary devices already exist) but in saving old equipment by using it in new, higher quality systems. At least a part of the existing devices can be used even by systems with higher specification demands. The word devices as used here applies not only to hardware devices since the appropriate software represents the major part of the investment.

Our intention was to build a new microprocessor based minicomputer with an improved price/performance factor which would be compatible - at least on software level - with the TPA-i. For the CPU a twelve bit monolithic CMOS microprocessor chip was chosen. Although the microprocessor does perform the basic instruction set of the PDP-8 for a 4K memory, its timing and control signals differ from those used by the TPA-i.

There were two different conceptions for the implementation. The first (the TPA-L) involved a general purpose computer which would be especially suitable for business applications; the second solution (the TPA-LC) was principally concerned with real-time applications. In the following we will briefly describe both of these.

The principal concerns were software compatibility with the TPA-i; availability of existing TPA-i peripheral interfaces and, as far possible, of TPA-i mechanics.

The block-diagram of the TPA-L is shown in Fig. 1.
The bus system of the TPA-L is divided into two parts, viz. the µBUS and the so-called ADT and PDT bus. The latter has its origin in the TPA-i. (PDT is the acronym for Programmed Data Transfer bus, ADT is Autonomous Data Transfer bus.) The two buses are connected by the Bus Converter Unit (BCU), which enables data transfer and control from the CPU to the units placed along the ADT and PDT bus.

The µBUS, which consists of data and control signals of the CPU, is terminated by the Bus Terminator Card (BTC) containing a µBUS display for maintenance purposes and connectors for a possible µBUS extension. The slots between CPU and BTC cards are used for connecting memory and new type interface cards. The Control Panel Interface (CPI) takes care of the implementation of the keyboard functions. The Control Panel (CP) is connected directly to the CPI card and contains the switches and display necessary for the manual control of the computer.

The two CPU cards contain, besides the CPU, the memory extension and time-share control electronics and the power-up restart facility. One of the CPU cards contains a real-time clock as well.

The whole computer is mounted in the original 19" rack of the TPA-i, where only the front-panel has to be changed.

The advantages of this structure can be recognized immediately. The development time was very short (less then one year). A cost factor of 0.4 to 0.25, depending on the system size, could be reached. The whole software package and all interface cards of the TPA-i are immediately available. In this regared no new development is necessary.

THE TPA-LC-A MICROCOMPUTER FOR REAL-TIME APPLICATIONS

Real time computer systems established by our Institute are interfaced by CAMAC to industrial or laboratory equipment. The CAMAC system has proved to be extremely powerful and admirably suited to this purpose. One of the main ideas of the TPA-LC was to benefit from the high density of modern LSI circuits, i.e. to build a microcomputer in CAMAC mechanics and eliminate therewith expensive TPA-i mechanics, cables, connectors and driver - receiver circuitry. A second consideration was to create a modular system and to allow more processors along the same bus, and a parallel processing capacibility. None of this, of course, should affect the software compatibility with the TPA-i.

The block-diagram of the TPA-LC is shown in *Fig.2*. The TPA-LC is a three module wide CAMAC unit (the area enclosed by the dashed line). The card in the middle is the CPU with the memory, extension controller. The CPU controls an internal bus which carries the same signals as the µBUS mentioned earlier. To this internal µBUS is connected the Keyboard Controller (KBC), a TTY interface and the CAMAC interface card. There are there connectors on the front panel of the unit: one for the TTY, one for the µBUS extension, one for a small touch board (RECORD) with the necessary switches and display.

CAMAC untis, TPA-LC interface cards and, if required, other processors can be connected to any of the slots along the Dataway.
The physical lines of the Dataway carry two sets of signals, either the CAMAC Dataway signals according to the CAMAC standard EUR 4100 or the signal set of the TPA-LC external bus, the IMBUS. The Dataway signal B informs the units connected to the Dataway whether the signals are to be taken as CAMAC signals (B=1) or IMBUS signals (B=0). CPU activity takes place on the internal bus. If the TPA-LC wants to reach any of the units connected to the Dataway it asks for the Dataway and waits until signals (Bus Busy, Bus Bargain In and Bus Bargain Out) do not allow Dataway access.

If the Dataway access becomes free the TPA-LC starts its IMBUS or CAMAC cycle - whichever is intended.

The number of interruptable processors (TPA-LC-s) on the Dataway can be up to three. Interprocessor communication may take place via common scratchpad memory, and/or via the vectored interrupt facility in the MPO (Multiprocessor Option) unit of the TPA-LC, by which processors may interrupt each other by a vector stored in the vector register of the MPO.

Versatility and the modular concept of the TPA-LC show special advantages in real-time applications.

CONCLUSIONS

Two practical approaches for micro-based minicomputer implementation have been demonstrated. Both solutions enable the safeguarding of existing investments. The use of the TPA-L development concept shortened the development time considerably. The multiprocessor capability of the TPA-LC is very advantages for real--time applications; however, the development of new-type IMBUS compatible peripheral controllers must be taken into account.

REFERENCES

1. TPA-i Computer Manual, KFKI Budapest
2. CAMAC - A Modular Instrumentation System for Data Handling, EUR 4100, 1972
3. G. Messing; A Distributing LAM Grader Unit - A Way to Distribute Intelligence in the CAMAC Crate.

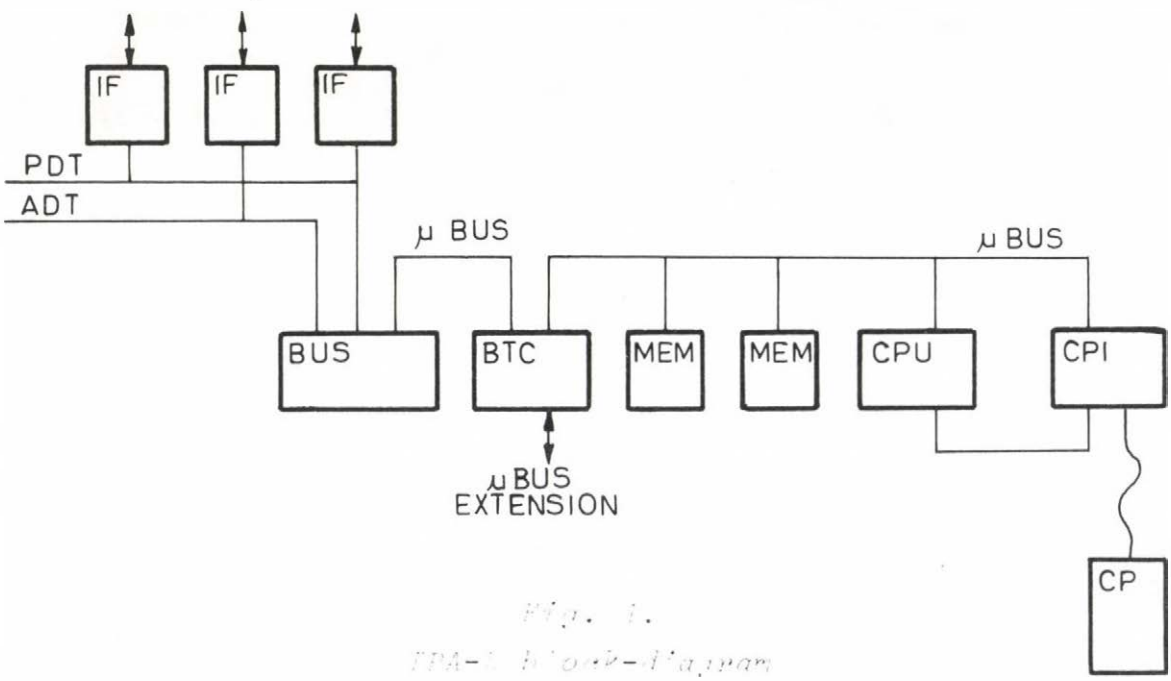


Fig. 1.
IPA-S block-diagram

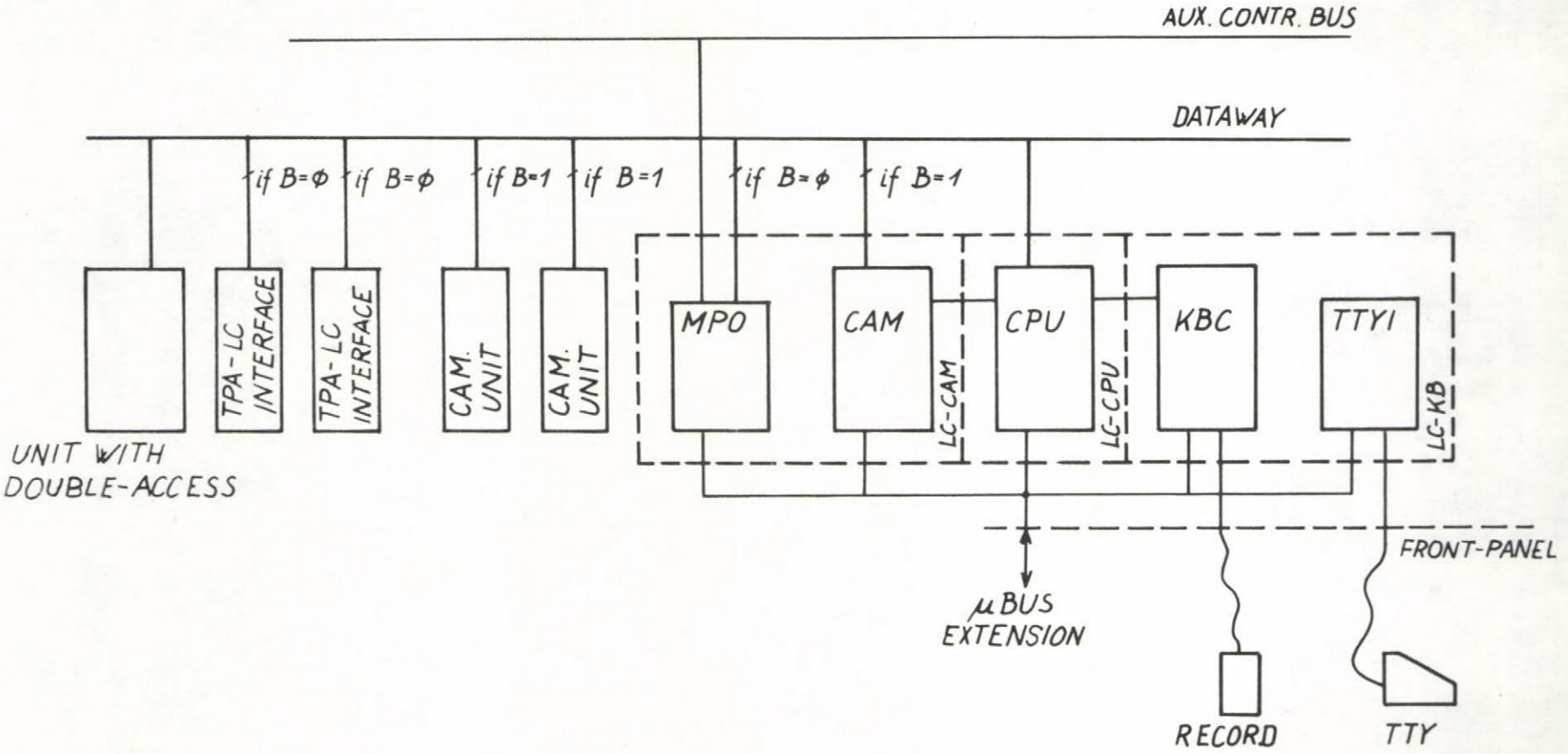


Fig.2 TPA-LC block-diagram

A SURVEY OF TEHCNIQUES FOR TESTING MICROPROCESSORS

P. KERNTOPF

Institute of Computer Science,
Warsaw

INTRODUCTION

Testing microprocessors is a relatively new and challenging problem. The high complexity of microprocessors and nonaccessibility of internal paths of signals make the conventional methods of testing LSI circuits as well as methods developed for testing large digital computers inadequate for solving this problem.

Techniques for testing microproprocessors are needed by both microprocessor manufacturers and users but the needs of these two groups differ greatly. Testing microprocessors by manufacturers requires exhaustive checking of every product for all possible types of failure. The users developing a microprocessor system are interested in checking performed during the normal work. Moreover, only those operations that are really used should be checked (e.g. some of the instructions may not be used in an application program).

In this paper a brief review of methods for detecting microprocessor malfunctions is presented from the user's point of view.

The obvious technique for detecting microprocessor malfunctioning is to introduce multiple-voting redundancy. However, this well-known approach is quite costly as it requires the replication of microprocessors, memories and voters. In addition,

to use this approach effectively one must first solve some new problems which arise [1].

In the paper we discuss some recently considered approaches different from the multiple-voting one. Firstly, a natural approach using a set of diagnostic programs that checks the units of the microprocessor system is described and performance parameters for a commercially available microprocessor are given. Then, three recently proposed concepts for testing microprocessor systems through observing some selected parameters characterizing the application program behaviour and detecting unexpected or invalid events are presented. An example of the simple possibility of detecting the cumulative effect of a failure on the program behaviour is the detecting of unused operation codes and forbidded memory access requests [2]. However, such testing is not very sensitive. We shall describe methods based on choosing state, sequencing and execution time as the parameters to be observed.

TEST PROGRAMS

A natural idea for testing a microprocessor in the system is executing a set of special test programs on the microprocessor itself. These programs are run during periods when no application programs are executed. We shall consider a typical diagnostic system described in [3]. The system consists of resident diagnostic programs (permanently in the microprocessor system), non-resident diagnostic programs (loaded into the memory) and the diagnostic supervisor (the collection of programs controlling the execution of the diagnostic programs).

The diagnostic supervisor initiates the execution of resident and non-resident programs. It has three states: program state, idle state and machine check state. The application programs of the microprocessor system are executed in the program state. The diagnostic supervisor has no control over the execution of the programs in this state. If there are no application programs

waiting to be executed, then the supervisor changes to the idle state. The detection of a fault in the idle state results in a transition to machine check state. The faulty units are located by the diagnostic programs executed under the control of the supervisor in the machine check state.

The approach presented has one essential disadvantage: it requires that a substantial part of the tested microprocessor operates correctly. Moreover, some faults occuring during execution of application programs might remain undetected while the others are detected after a delay which sometimes may be too long in terms of safety of the system. The testing using test programs is also costly if we take into account the additional equipment and memory space.

Let us examine the performance figures of the diagnostic system described above [3]. Resident diagnostic programs for a commercially available 8-bit microprocessor with a basic instruction time of 2μsec and 8K bytes of RAM memory use 1K bytes of storage and their execution time is about 1 minute. Nonresident programs for the above two units use about 4K bytes of storage and their execution time is about 5 minutes.

STATE VERIFICATION

Now the method originally developed for testing control units with no direct and exhaustive observation points [4,5] will be presented. In this method a system is divided into two parts: the control part and the operative part.

The control part is viewed as a Moore automaton. Let Q denote the set of control states and let a set of commands C_i be associated with every state $Q_i \in Q$, where C_i is the set of commands generated by Q_i which activate some operations in the operative part.

The operative part is considered as a set of independent functional units U_k. The subset of commands which is sent to U_k from the control part is denoted by C^k. Let C_i^k be an operation activated by the state Q_i and performed by the unit U_k. Observing a state of the control part can only be performed through the operative part. Two states Q_i and Q_j are distinguishable through a funcitonal unit U_k if $C_i^k \neq C_j^k$.

Testing of the system consists in developing tests by choosing appropriate paths in application programs and appropriate data allowing the checking of the operation of the control part. This is done by exhaustive analysis of application programs to be performed by the tested system. When constructing tests we look for the distinguishability of every state from all the other states, the ability to check all commands which are generated by every state through each of the functional units and which pass through every transition for all possible input value allowing this transition.

In the microprocessor-based system the control part is assumed to be in the form of the flowchart of the application program; a control state corresponds to an instruction of an application program, and the operative part is made up of the microprocessor, the memories, the peripheral devices, etc. An observability means of a unit is an instruction of an application program which is observable during a transfer between the unit and another unit.

To perform the tests a working memory space is needed for storing information about the test progress. The disadvantage of the method is that this space may be very large for complex systems.

SEQUENCING VERIFICATION

A method for developing reliable software has been presented in [6]. The method is based on a graph-theoretic model of software systems which characterizes the system by its set of allowable execution sequences. The method was applied in [7] to testing microprocessor systems during their normal functioning.

In this method an application program is divided into modules. At the entry to each module the sequence of the fixed number of previously executed modules is checked. Faults can be detected by comparing the actual sequence with known allowable sequences. To use this method it is necessary to develop an algorithm for structuring application programs so that every execution sequence generated under the presence of a fault is not among allowable execution sequences of the system.

Known allowable sequences can be stored as relations between modules (for example, execution of module A can follow execution of B or C,A can be interrupted for B but not for C, etc). In addition to extra memory space some hardware is needed for performing the checking operation.

EXECUTION TIME VERIFICATION

The time-based method [8,9,10] is based on

(1) using time as the unique integration parameter for detecting malfunctions, and

(2) the observation that in existing microprocessors it is possible to insert some instructions into a program in such a way that they will not interfere with the execution of the program.

The method compares the previously known execution time of a program with the actual time.

It is possible to insert into a program generators sending signals every time the processor passes through these generators

during execution of the program. The generators can be used to denote the start and the end of measuring time. Let us consider a sequence of instructions in a program and let A represent the beginning and B represent the end of executing the sequence. Let us assume that a timer in a testing unit is initialized at the point A and that the testing unit "knows" what is the correct time interval for executing the sequence. Let us also assume that at the point B the signal for the end of executing the sequence is sent to the testing unit. If

(1) The testing unit receives the signal for the end of executing the sequence before the beginning of the time interval,

or (2) the expected time (i.e. the end of the time interval) has passed, but the testing unit has not yet received the signal for the end of executing the sequence,

it will indicate that something has gone wrong.

CONCLUSIONS

The recently proposed methods of testing microprocessors during their normal functioning consists of detecting deviations from the a priori known behaviour of application programs. They differ substantially in the parameters which are observed for detecting malfunctions. In all these methods preliminary analysis of application programs is necessary to prepare information about allowable program behaviour and all require extra equipment. It is difficult to compare the methods without sufficient experience in applying them. Nevertheless the time-based method seems to provide a reduction in the amount of information that must be observed to detect malfunctions and to organize the observations. This method is also very sensitive because it deals with integer numbers (i.e. execution times expressed in clock cycles). The major advantage of these methods is that they are capable of detecting not only the effects of hardware failures but also the effects of incorrect synchronizing of processes within the system as well as error made by the system designer

(i.e. mistakes in design or implementation).

REFERENCES

1. Wakerly J. F., "Microcomputer reliability improvement using triple-modular redundancy", Proceedings of the IEEE, vol.64, No. 6, June 1976, pp. 889-895.

2. Anceau F., "Sécuarité dans les systémes complexes: Application aux autocommutateurs téléphoniques", Internal Report, ENSIMAG, Grenoble, December 1976, 15 pages.

3. Srini V. P., "Fault diagnosis of microprocessor systems", Computer, vol. 10, No. 1, January 1977. pp- 60-65.

4. Robach C., Saucier G., "Dynamic testing of control units", to appear in IEEE Trans. on Computers.

5. Robach C., "Microprocessor testing", Internal Report, ENSIMAG, Grenoble, 1977, 26 pages.

6. Kane J. R., Yau S. S., "Concurrent software fault detection", IEEE Trans. on Software, vol. SE-1, No. 1, March 1975.

7. Ayza J., Figueras J., "Partial on-line surveillance of microcomputer-based systems", Internal Report, ETSII, Technical University of Barcelona, Ref. no. 7605-01.

8. Marczyński R. W., Kerntopf P., Anceau F., Courtois B., "A method for detection of microcomputer malfunctions". Institute of Computer Science, Polish Academy of Sciences, Warsaw 1978, Report No. 308.

9. Kerntopf P., Marczyński R. W., "Time-based method of detecting malfunctions in programmed digital devices", to be presented at International Conference on Fault-Tolerant Systems and Diagnostics, Gdańsk, Poland September 22-24, 1978.

10. Kerntopf P., Marczyński R.W., Michalski A., "Microcomputer fault detecting using the time-based method", to be presented at EUROMICRO Symposium, Munich, October 17-19, 1978.

МЕТОДЫ И СРЕДСТВА ПРОЕКТИРОВАНИЯ ЦИФРОВЫХ УСТРОЙСТВ, ВЫПОЛНЕННЫХ НА БАЗЕ МИКРОПРОЦЕССОРОВ

Эрени И.

Венгерская Академия Наук
Центральный Институт Физических Исследований, Будапешт

Введение

Появление микропроцессорных БИС создало возможность для построения таких систем и решения таких задач, которые раньше могли быть осуществлены громоздкими логическими схемами или мини ЭВМ. Но с применением микропроцессоров компактность систем и устройств значительно возросла при существенном снижении затрат на производство. Это в свою очередь способствовало и введению ряда новых свойств и показателей в новые системы. Снижение затрат на производство цифровых устройств на базе микропроцессоров, дешевизна БИС и возможность приобретения микропроцессоров в большом количестве - все это толкает к их широкому применениею в новых разработках. Но они требуют особого подхода к решению задач от разработчика. В первое время - во время первой генерации микропроцессоров - успехи новых разработок часто были сломаны большим временем, необходимым для создания систем, и, конечно, дороговизной разработок. Естественно, что микропроцессоры - это новое семейство "логических элементов" несравненно более сложное, чем любое из предыдущих, и которое требует новые средства, а также методы разработок, и наладочных работ.

Во всяком случае к инженеру-разработчику предъявляются следующие требования: знакомство с проектированием аппаратурной части, логическая схема; практика в составлении программ

для /микро/ ЭВМ; и системный подход к работе по созданию оборудования.

В будущем разработки аппаратной части все большее сводятся к формированиям различных конфигураций из БИС таких, как центральные элементы, элементы памяти и периферийные элементы. Но основной проблемой остается и дальше создание программного обеспечения и совместная разработка комплекса аппаратной части и управляющей программы.

В прошлом работы по созданию систем с микропроцессорами были разделены на два основных направления: разработку основных характеристик и схемных частей аппаратуры, а также на разработку и наладку программ. Такое разделение привело к значительным трудностям в согласовании работ и к большим запозданиям в выпуске прототипного образца. Важнейшим шагом, облегчающим применение микропроцессоров, было создание средств, дающих возможность объединить, а также запараллелить проектирование аппаратуры и составление программ во всех фазах разработочного цикла.

Подход к применению микропроцессоров

По вопросу о подходе к применению микропроцессоров можно различить два крайне отличных друг от друга взгляда. По первому микропроцессор применяется в комплексе схемных элементов в виде микро ЭВМ. Под микро ЭВМ здесь понимается система, собранная на одной или нескольких печатных платах и содержащая кроме микропроцессора /МП/ и запоминающее устройство /ЗУ/, и блоки ввода/вывода. Такую микро ЭВМ можно заказать и встроить в разработанную систему, и можно составить программу для обслуживания системы.

По второму подходу микропроцессор применяется в виде семейства БИС элементов, встраиваемых в разрабатываемую систему по усмотрению инженера-проектировщика. Исторически этот вто-

рой подход более сходен с разработкой аппаратурной части, а первый с применением мини ЭВМ.

В чем же появляются отличительные черты этих двух подходов, и когда целесообразно следовать по одному и при каких условиях по другому? Попробуем ответить на эти вопросы.

Микро ЭВМ имеют следующие основные свойства:

- инженер-разработчик получает их в готовом, работоспособном виде;
- ЗУ, как правило, составлено из элементов ЗУ произвольной выборки, и для ввода программы необходимы вводные периферийные устройства;
- устройство разработки /микро ЭВМ/ входит в продукцию;
- проверка работоспособности микро ЭВМ не зависит от применения; осуществляется подобно тестам-программам ЭВМ;
- их применение более целесообразно в продукции с малым количественным выпуском.

Семейство БИС элементов имеет следующие особенности:

- инженер-проектировщик получает БИС в виде элементов; разработка схемы является его задачей;
- разрабатываемая схема обычно работает на базе программы, внесенной в постоянное ЗУ; обычно нет периферийных устройств для ввода программы;
- средство разработки и наладки абсолютно отделено от продукции, поскольку готовый продукт не должен иметь функции, необходимые в разработочной стадии;
- в качестве периферийных устройств обычно применяются интерфейсы БИС-члены семейства микропроцессора;
- проверка работоспособности сводится к проверке правильности всех функций готового продукта;
- микропроцессорные семейства, как правило, применяются в большом количестве;

- из применения постоянных ЗУ вытекает требование к минимальному объему памяти, необходимому для хранения программы, - то есть программа должна быть составлена по возможности более компактно.

Спектр средств разработки и наладки цифровых устройств, выполненных на базе микропроцессоров

Естественно, что подобно тому, как микропроцессоры появились благодаря развитию вычислительной техники /в том числе мини ЭВМ/, а также благодаря развитию логических схем и технологии производства интегральных схем, так и средства разработки и наладки микропроцессорных устройств рождены из области методики создания и наладки аппаратуры дискретного действия, а также из области вычислительной техники. На рис. 1. показан примерный спектр существующих вспомогательных аппаратных и программных средств разработки.

Слева спектр открыт осциллографами и счетчиками импульсов, которые служат исключительно для помощи исследования работы аппаратурной части. Справа начинается системами на базе сетей ЭВМ и их программного обеспечения, используемых только для разработки и составления программной части.

Устройства левой части спектра годны только для проверки аппаратурной части без окончательной программы, в которой редко можно выделить хорошо заметные на осциллографе циклы, часто появляющиеся и однозначные импульсы, особенно, если пробег программы не безошибочен. Средства в правой части спектра не способствуют обнаружению неисправностей и ошибок в аппаратной части, не дают возможности для проверки работы разработанной системы в реальном масштабе времени. К тому же стоимость разработок на сетях ЭВМ очень велика.

Исследуя данный спектр можно заключить, что в середине его имеются те средства, которые и по стоимости, и по создан-

ным ими возможностям больше всего удовлетворяют требованиям разработчиков. С их помощью хорошо осуществляется проверка и наладка разрабатываемого устройства, работоспособность аппаратурной части, а также удобно составляется программа работы. Особенно выгодно применяются системы разработки и наладки с моделирующим микропроцессор модулем в случае, когда применяется семейство микропроцессорных элементов, не микро ЭВМ. Во многих случаях хорошо используется и система на базе мини ЭВМ.

В дальнейшем рассмотрим требования, поставленные к системе разработки и проектирования на базе мини ЭВМ с использованием моделирования управляющей памяти и системной шины. Разберем примерную функциональную схему и ход разработки с помощью этой системы, а также пример универсальной системы проектирования и наладки цифровых устройств на базе микропроцессоров.

Система проектирования и наладки на базе мини ЭВМ

В том случае, если доступна мини ЭВМ для разработчиков оборудований на базе микропроцессоров, легко можно построить гибридную систему некоторого рода моделирования работы оборудования на базе этой ЭВМ. В ряде случаев использование оперативного ЗУ ЭВМ для программы микропроцессора дает уже значительную выгоду для проверочных работ. Добавляя небольшой блок стыковки микропроцессорного оборудования к ЭВМ, успешно можно решать эту задачу так, что путем использования всех услуг мини ЭВМ и ее периферийных устройств инженер-проектировщик и программист получают отличную возможность исследования и вмешательства в работу микропроцессорной системы. При этом сохраняется и возможность прослежения за работой микропроцессора и его программы в реальном масштабе времени /если ЗУ мини ЭВМ достаточно быстродействующая/ или в близком к режиму реального масштаба времени.

В фазе составления программ микропроцессоров можно использовать основное программное обеспечение мини ЭВМ. Так ре-

дакторная программа мини ЭВМ помогает составлению исходной программы. Ассемблеры /трансляторы/ ЭВМ в ряде случаев легко поддаются адаптации к командному набору микропроцессора, что дает возможность создания кросс-ассемблера для выбранного типа микропроцессора. Часто составление такого кросс-ассемблера потребует только замены таблицы команд ЭВМ на таблицу команд микропроцессора.

Важнейшими функциями системы проектирования и наладки в фазе наладочных работ являются следующие:

- возможность считывания и записи в память микропроцессора /в данном случае она совпадает с ЗУ ЭВМ/;
- возможность исполнения программы по командам;
- доступ ко внутренним регистрам микропроцессора /считывание и запись содержания в них/;
- возможность остановки программы в любой заранее определенной точке программы.

Так, работа с подобной системой на мини ЭВМ проектирует следующим образом. После трансляции программы необходимо соединить мини ЭВМ и систему микропроцессора без памяти. Под управлением программы "моделирования" в память мини ЭВМ вводится программа микропроцессора и, при желании оператора, задаются исходные данные /граничные условия/ в необходимой ячейке памяти, а также регистрах микропроцессора. Потом оператор задает условия остановки программы и также адрес запуска. Перед отправлением программы микропроцессора мини ЭВМ заставляет выполнение "вспомогательной программы" заполнения внутренних регистров микропроцессора. Затем программа из ЗУ ЭВМ считывает по командам и исполняется микропроцессором до появления остановки. В момент остановки ЭВМ также заставляет выполнение "вспомогательной программы", которая считывает и записывает в ЗУ ЭВМ содержание внутренних регистров. Интересующие оператора данные при этом отображаются на периферийных устройствах мини ЭВМ.

Аппаратурная часть может быть проверена и налажена прогоном специальных тестовых программ на микропроцессоре и использованием при этом осциллографа,а также вспомогательных пультов отображения логического состояния важнейших шин разрабатываемого оборудования.

На рис. 2. приведена блочная схема такой системы - системы "ассемулятора" -, созданного в ЦИФИ на базе мини ЭВМ типа ТРА-1 для наладочных работ оборудований, выполненных на основе ряда различных типов микропроцессоров. В систему входят блоки для программирования различных типов программируемых и перепрограммируемых постоянных ЗУ.

Универсальные системы проектирования и наладки цифровых автоматов, выполненных на базе микропроцессоров

Естественно возникает идея: применить микропроцессоры для построения систем проектирования и наладки. Микропроцессор тоже является центральным блоком вычислительной аппаратуры, может брать на себя функции, которые были перечислены в связи с применением мини ЭВМ для таких задач. Более того, целесообразно создать специальный модуль внутри системы проектирования и наладки для моделирования микропроцессора, находящегося в разрабатываемом устройстве. Этот модуль, заменяя все функции микропроцессора в проверяемом устройстве, дает еще возможность для вмешательства оператора в ход проектирования программы. Но если моделировать микропроцессор, то почему не моделировать аналогичным типом микропроцессора в модуле моделирования, конечно, добавляя еще и схему для создания возможности вмешательства. Такой принцип "моделирования в схеме", по сути дела, является моделированием всех операций микропроцессора в его физической позиции. Для достижения этого микропроцессор удаляется из цоколя в разрабатываемом оборудовании на время проверочных работ и заменяется модулем моделирования.

К системе проектирования и наладки для реализации вмешательства оператора предъявляются следующие требования:

- обнаружение начала исполнения новой команды;
- возможность остановки программы в определенной точке;
- осуществелние считывания и записи в ЗУ;
- доступ к содержанию внутренних регистров микропроцессора;
- желательна еще возможность занесения различных состояний проверяемой системы в накопитель событий, откуда после остановки можно анализировать предшествующие события.

В фазе разработки программы система проектирования и наладки работает как микро ЭВМ, имеет систему математического обеспечения: редакторную программу, трансляторы, различные программы для работы с блоками данных /файлами/ и т.д.

В фазе разработки, проверки и наладки аппаратурной и программных частей, а также при их совместной проверке и наладке применяются модули: моделирования, остановки, накопителя событий и программы, обеспечивающей нормальный ход проверочных и наладочных работ.

В качестве примера работы с такой системой приводится система **UMDS-KFKI**, разработанная в ЦИФИ. При ее создании были учтены следующие важнейшие требования:

- возможность параллельной разработки аппаратурной и программной частей;
- возможность как испытания разрабатываемого устройства в целом, так и постепенное построение системы из уже проверенных блоков;
- универсальность с точки зрения исполнения системы разработки и наладки для многих типов микропроцессоров, отобранных для разрабатываемого устройства;
- удобство работы с системой.

Блочная схема системы проектирования и наладки показана на рис. 3. Система построена вокруг единой системной шины. К этой шине подсоединяется центральный модуль управления /ЦМУ/ с микропроцессором типа Zilog Z-80 большой вычислительной мощности. В схему этого модуля входит также блок передачи/приема информации последовательного кода, который является интерфейсом для алфавитно-цифрового дисплея. Операционная система и наладочные программы занесены в постоянное ЗУ. Наряду с постоянным ЗУ система имеет и ЗУ произвольной выборки, максимальная емкость которого 64 Кбайта. Все системные программы, кроме ОС и наладочных программ, используются из этого ЗУ. Для хранения программ и данных служит накопитель на гибких магнитных дисках. Поскольку емкость дискового накопителя практически неограничена, то и сложные системные программы можно использовать. Но кроме системных программ, на дисках можно хранить и файлы данных и программы пользователя.

По желанию к системе можно подключить и различные периферийные устройства: перфоленточные устройства ввода/вывода, строкопечатающее устройтсво и т.п.

Все эти здесь упомянутые блоки создают мини ЭВМ, с помощью которой можно очень удобно разрабатывать программы.

Для наладки аппаратурных и/или программных частей служат следующие модули:

а/ Модуль модулирования микропроцессора разрабатываемого устройства. Этот модуль подключается кабелем к устройству и заменяет в нем микропроцессор. При такой замене модуль берет на себя все функции заменяемого микропроцессора. Схема модуля дает возможность исполнения функций устройства в реальном масштабе времени так, чтобы в любой точке пробега программы можно было остановить работу микропроцессора, анализировать и, при желании, изменить его внутреннее состояние. Схема модуля также содержит мультиплексоры, с по-

мощью которых в целях хранения во время исполнения разрабатываемой программы можно выбрать либо память разрабатываемого устройства, либо /если ее еще нет/ память системы проектирования и наладки.

б/ Модуль обнаружения условий остановки, записи в накопитель событий и выдачи синхронного импульса для осциллографа вырабатывает свои сигналы по заранее запрограммируемым условиям. Условия задаются оператором в виде логической комбинации контрольных сигналов микропроцессора с содержанием адресного слова и с содержанием слова данных, а также некоторых аппаратных сигналов. Сигнал остановки передается модулю моделирования, который при этом останавливает свою работу, ожидая следующую команду от оператора. Сигнал записи в накопитель событий воспринимается модулем накопителя событий.

в/ Модель накопителя событий служит для сбора информации о состоянии шин и контрольных сигналов микропроцессора в разрабатываемом устройстве во время проектирования программы в режиме реального времени. Эта информация служит для анализа процессоров в устройстве без остановки и нарушения хода его действия. После остановки по считанным из этого накопителя данным можно узнать о правильности работы или о появлении нарушения нормального хода работы устройства. Микропроцессорное событие характеризуется 40 двоичными разрядами /важнейшие контрольные сигналы, содержание адресной шины и шины данных/. В накопитель вмещается до 256 слов событий.

По желанию, можно подключить к модулю моделирования блок параллельных/последовательных интерфейсов и создать возможность испытания периферийных устройств с программами в разрабатываемой системе до их окончательного построения.

В систему входят и модули для записи данных в самые различные типы программированных полупроводниковых ЗУ, а также для программирования БИС-матриц логических элементов.

Основное математическое обеспечение системы состоит из:

- операционной системы;
- редакторной программы;
- трансляторов: макро-ассемблеры и кросс-ассемблеры для различных типов микропроцессоров;
- набор программ для обслуживания различных действий с файлами данных на дисках;
- набора программ для обслуживания модулей программирования ППЗУ и матриц логических элементов;
- набора наладочных программ.

Эти последние имеют функции, вызываемые оператором с помощью печатной машинки:

- ввод программ и данных с дискового накопителя в ЗУ системы;
- считывание и запись содержания регистров микропроцессора в модуле моделирования;
- запрограммирование модуля обнаружения условий;
- считывание из накопителя событий;
- считывание и изменение содержания любой ячейки ЗУ;
- отправление программы в разрабатываемом оборудовании.

На рис. 4. показан ход разработочных и наладочных работ в случае применения **UMDS-KFKI**. Видно, что после определения структуры оборудования параллельно можно вести разработку и проверку программы и аппаратурных частей. Потом наступает фаза совместных испытаний и наладки.

Создание возможности параллельных разработок и ускорением, облегчением наладки и испытаний **UMDS-KFKI** сокращает и денежные, и временные затраты на разработки цифровых устройств, выполненных на базе микропроцессоров и больших интегральных схем.

ЛИТЕРАТУРА

1. Csákány A., Vajda F.: Mikroszámitógépek
Müszaki Könyvkiadó, Budapest, 1975.

2. L. Krummel, G. Schultz:
Advances in Microcomputer Development Systems
Computer, February, 1977.

3. W. Davidow:
The Coming Merger of Hardware and Software Design
Electronics, May 29, 1975.

4. R.D. Catterton, G.S. Casilli:
'Universal' development system is aim of master-slave processors
Electronics, September 16, 1976.

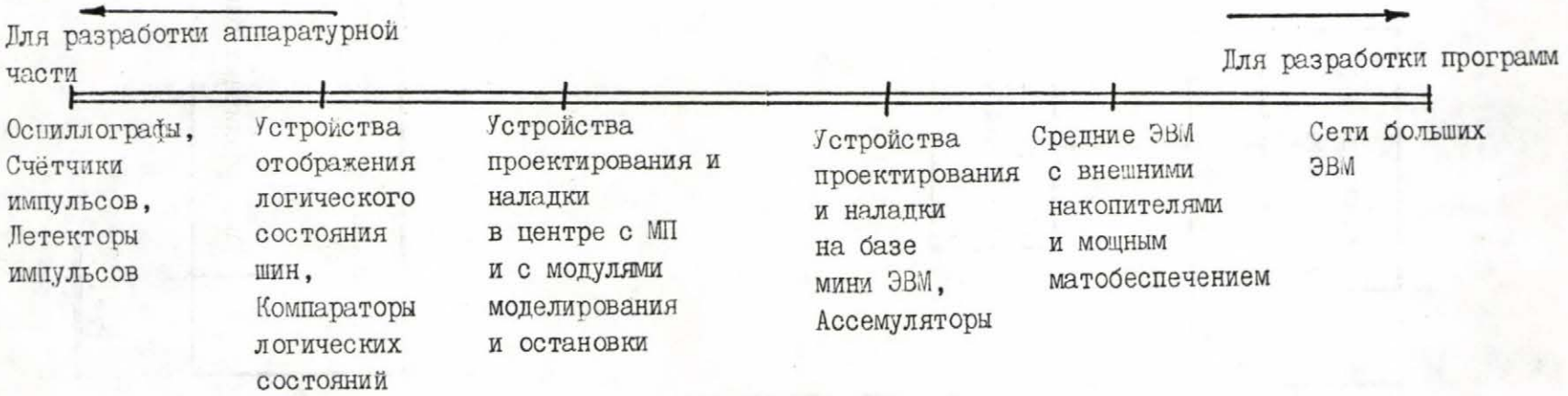

Для разработки аппаратурной части
Для разработки программ
Осциллографы, Счётчики импульсов, Летекторы импульсов
Устройства отображения логического состояния шин, Компараторы логических состояний
Устройства проектирования и наладки в центре с МП и с модулями моделирования и остановки
Устройства проектирования и наладки на базе мини ЭВМ, Ассемуляторы
Средние ЭВМ с внешними накопителями и мощным матобеспечением
Сети больших ЭВМ

Рис. I.

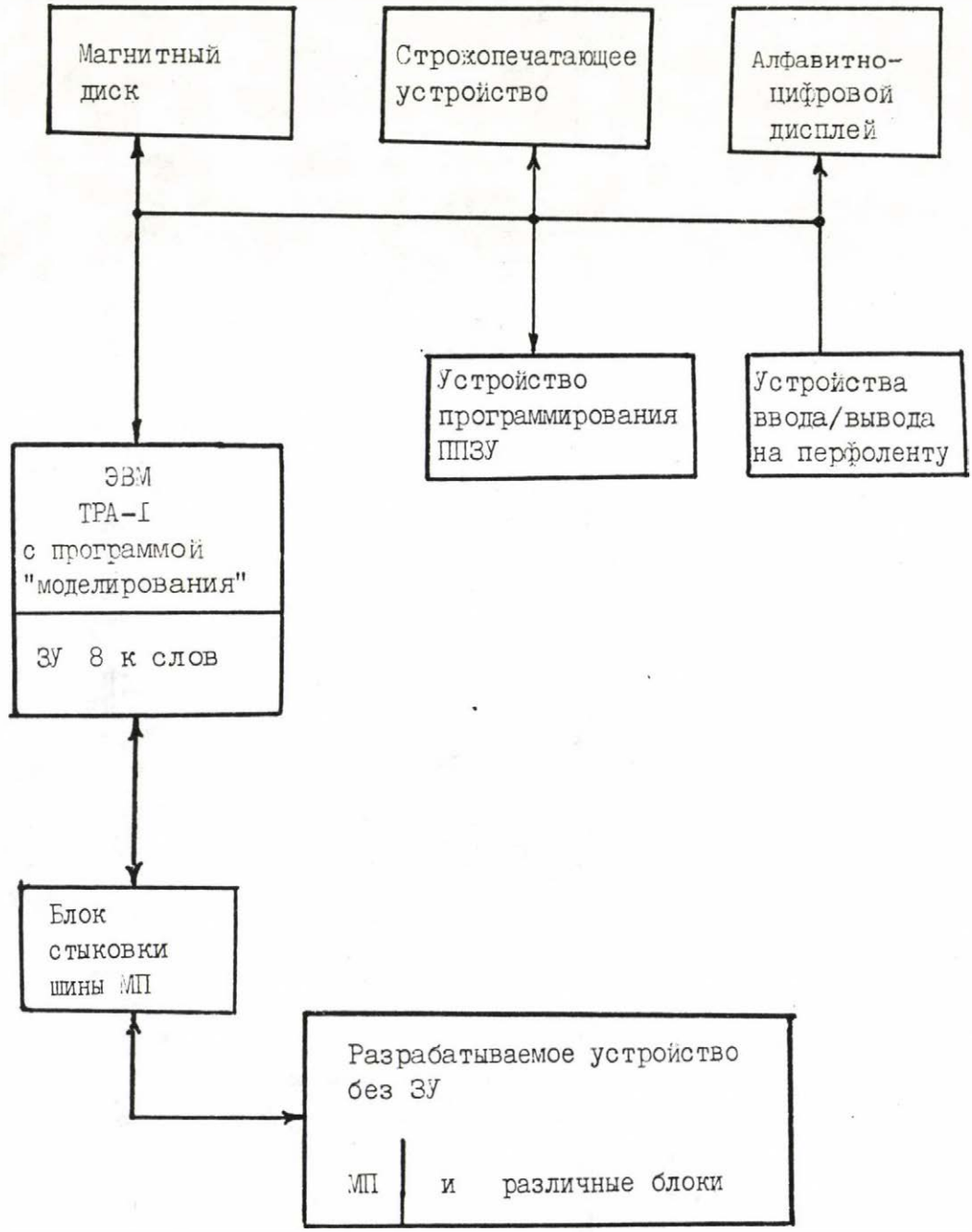

Магнитный диск
Строкопечатающее устройство
Алфавитно-цифровой дисплей
Устройство программирования ППЗУ
Устройства ввода/вывода на перфоленту
ЭВМ ТРА-1 с программой "моделирования"
ЗУ 8 к слов
Блок стыковки шины МП
Разрабатываемое устройство без ЗУ
МП
и различные блоки

Рис. 2.

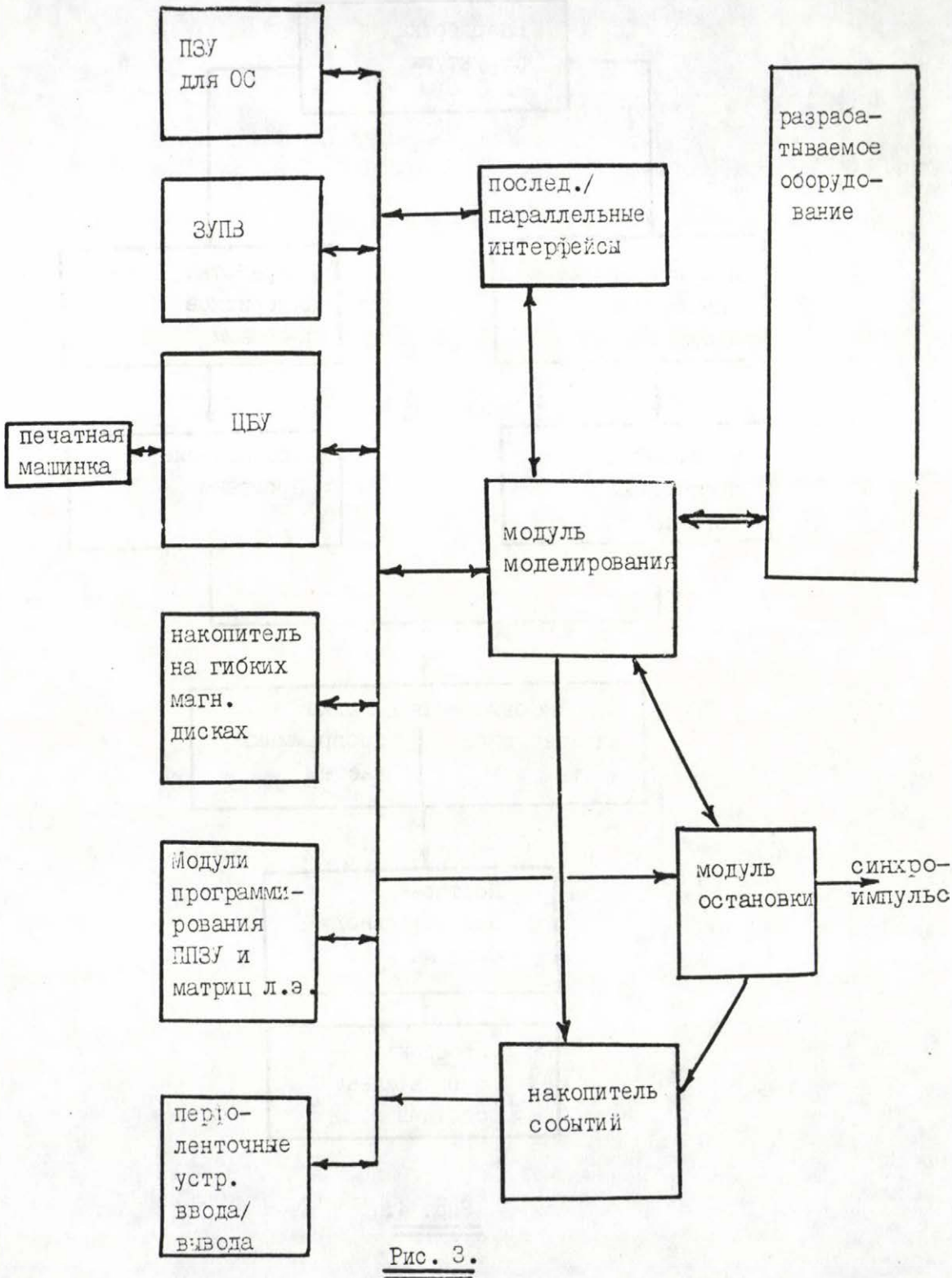

ПЗУ для ОС
ЗУПВ
ЦБУ
печатная машинка
накопитель на гибких магн. дисках
Модули программирования ППЗУ и матриц л.э.
перфоленточные устр. ввода/вывода
послед./ параллельные интерфейсы
модуль моделирования
разрабатываемое оборудование
модуль остановки
синхроимпульс
накопитель событий

Рис. 3.

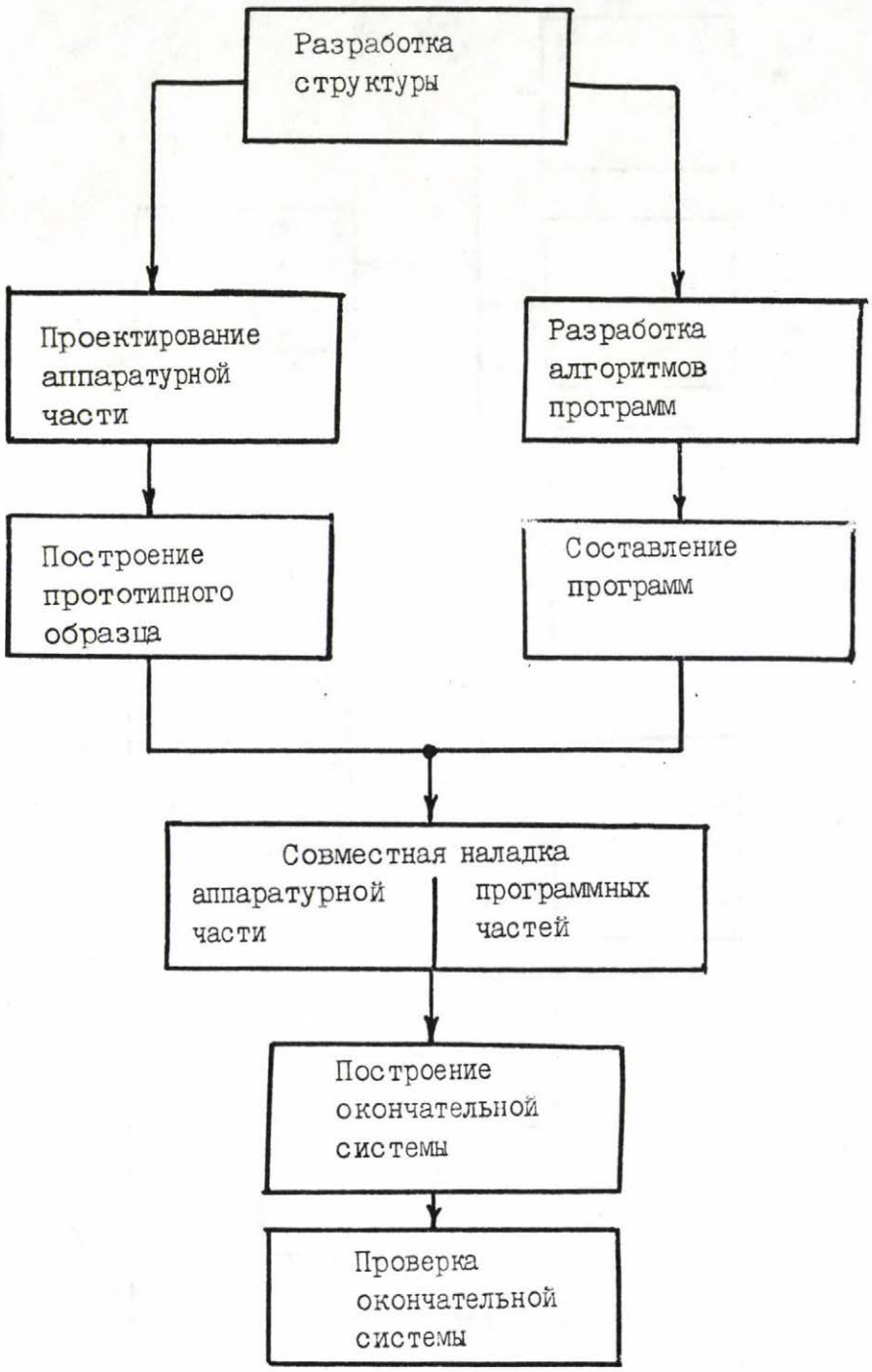

Разработка структуры
Проектирование аппаратурной части
Разработка алгоритмов программ
Построение прототипного образца
Составление программ
Совместная наладка
аппаратурной части
программных частей
Построение окончательной системы
Проверка окончательной системы

Рис. 4.

Session 2: System Development Systems

Сессия 2: Разработка систем

MICROPROGRAM DEVELOPING SYSTEM FOR EMULATION PURPOSES BASED ON A PDP-8E

G. AMBROZY *J. MISKOLCZI*
Central Research Institute for Physics

Budapest, Hungary

This paper describes a realized solution for some practical problems of microprogrammable microprocessor system applications, i.e. generation and verification of microprograms. The solution is especially suitable for emulation applications, e.g. simulation of the central processing unit of computers.

Automatic microprogram generating, optimizing and verifying high level program were not used for microprogram writing. As a matter of fact the procedure itself and the hardware structure form a round whole, therefore neither the known micro-assemblers /CROMIS [8], , Signetics Micro Assembler [9] / nor in-circuit emulators /as Intel MDS ICE-30 or the one described in [10] / were applicable.

The program package described syntactically checks the symbolic source microprograms, makes it possible to edit the source programs, assembles them, and produces binary object programs. The hardware-software system presented here is oriented towards the Intel 3000 microprocessor family but is applicable to any other type of microprogrammable microprocessor.

The method for the symbolic writing of microprograms is presented; the assembler and auxiliary programs for generating and correcting microprograms are then described; and finally, de-

scriptions of the development system and the program verification are given.

1. SYMBOLIC WRITING OF MICROPROGRAMS

1.1. Structure of a microinstruction

The program package presented is suitable for processing microprograms with horizontal-vertical type instructions with a maximum length of 72 bits. As an example, a possible microinstruction is shown in Fig. 1; it is an instruction for the Intel Series 3000.

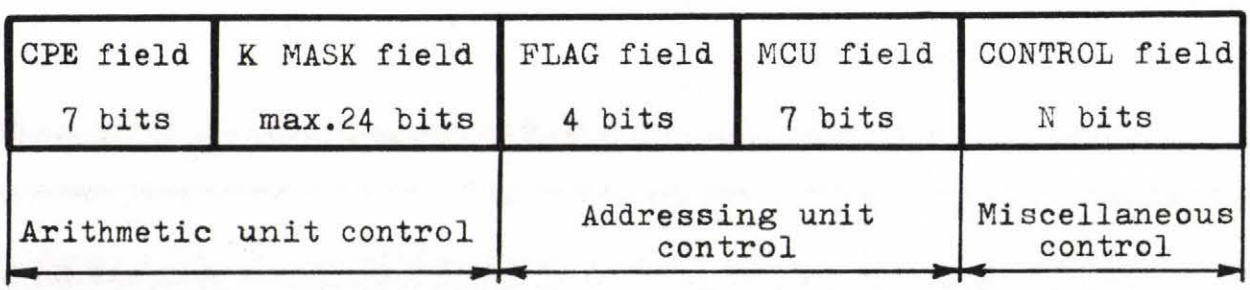


Figure 1: Structure of a microinstruction

The role of the microinstruction is as follows:

- Control of the arithmetic unit

  Two fields serve this purpose: the first field controls the actions of the central processing elements /CPE/ and the K field contains a constant. The CPE field is a microfunction bus determinig the operation to be performed and the operands. The K constant may be an input or the mask of an input. The first field is vertical, the second is horizontal.

- Control of the microprogram sequencing unit

  The next microinstruction address is determined by the MCU /Microprogram Control Unit/ controlled by the MCU field.

This field contains a variable length operating code and an address part. The MCU makes it possible to fetch the next microinstruction from the special environment of the currently executed instruction /except in one case/. The MCU handles the current values of the carry output and input of the CPE array; it can store the values in two flag flip-flops. This function is controlled by the Flag Control field. Both fields of the MCU control may be considered as vertical instruction field.

- Miscellaneous control

This field controls the auxiliary circuits of the system, e.g. input/output interface, scratch-pad register block, etc. It may consist of horizontal and vertical parts.

1.2. Philosophy of symbolic writing of microprograms

There are different points of view concerning the method used for microprogram writing.

- Microprograms should be written in an easily understandable and well structured manner.
- The computer used for the generation of microprograms and the one used for microprogram verification should be the same.
- The hardware and software support should be based on known means.

2. ASSEMBLER

With the above philosophy in mind, the minicomputer was chosen; the definition of the microinstructions is made so that the assembler of the minicomputer is able to assemble microprograms - the only modification was to change its symbol table.

Naturally the other programs of the minicomputer /e.g.EDIT, PIP, etc./ are available as well.

Since the assembler interprets the microinstruction fields one by one, a memory place of 12 bits is assigned to each field of the microinstruction. As the lengths of the fields differ and are often shorter than 12 bits with the result that the fields do not completely fill their memory places, assembling must be followed by an arrangement of binary information; by so doing useful information is able to find its way to the place of non-valuable bits.

Specification of microinstructions is made by symbolic names field by field.

Arithmetic unit fields: 2 or 3 memory places are assigned to these fields.

- Specification of CPE field is performed by one or two symbolic names. One symbolic name is used in the case of the instructions concerning the two special registers only /AC and T/; e.g. CIAA means making AC complement. If two symbolic names are used the first one is the operator code, the second is the operand /e.g. ILR R5/.
- Specification of K mask field may be made by symbols or octal numbers. The maximum length of the field is 24 bits; if the length exceeds 12, two memory places are assigned to this field.

Microprogram Control Unit fields: two memory places are assigned to these fields. The first field has a length of 7 bits and may be specified by a mnemonic /which defines the jump-instruction type/ and by a label /which points to the address or address environment of the next microinstruction/. The other field /the Flag Control field/ may be specified by two mnemonics:

- the first determines the flag bit input
- the second controls the carry readout and flag bit generation.

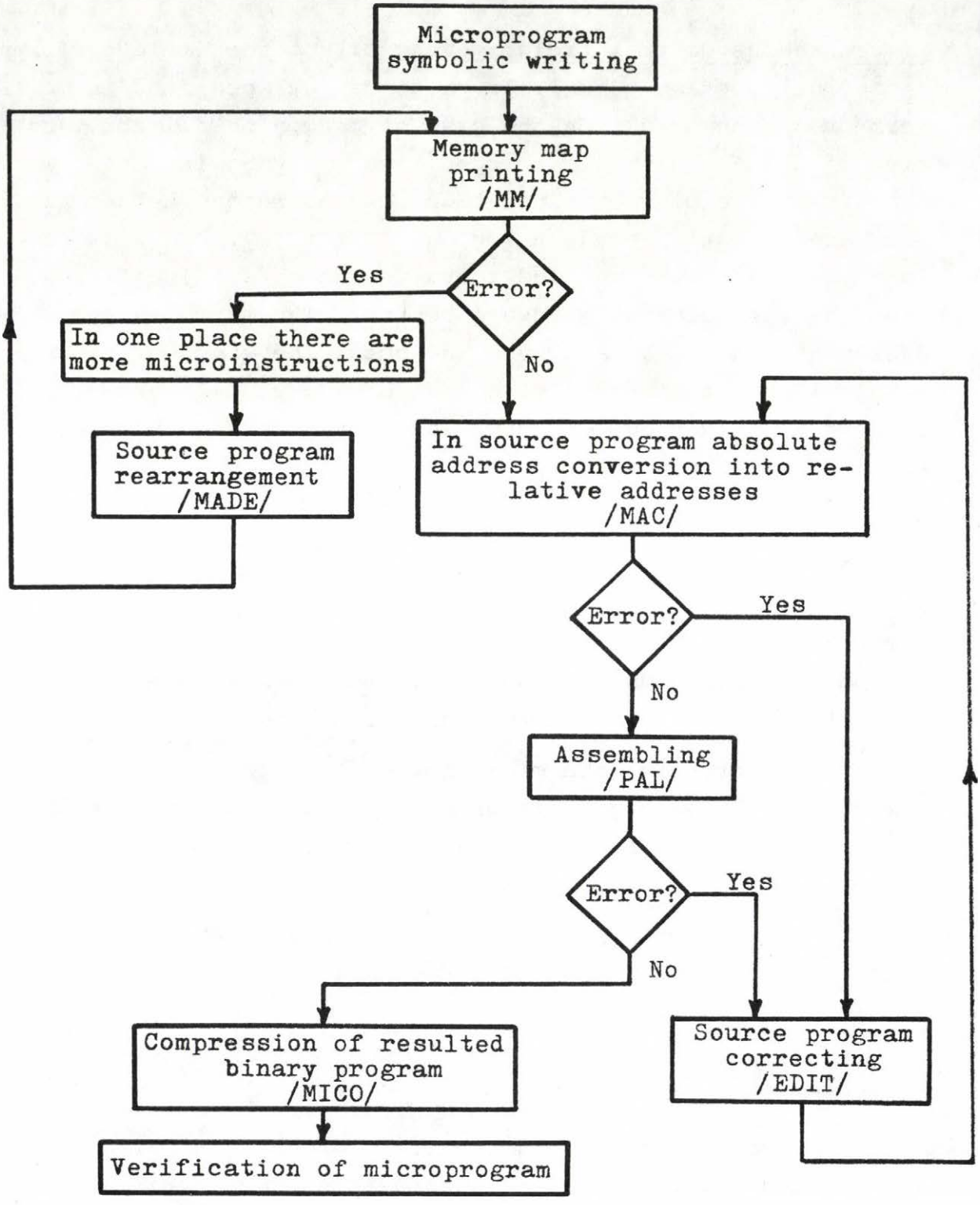
Miscellaneous control field: every bit of the field performs a control function. The field value can be specified in octal form or by mnemonics. In the case of mnemonics all the used bits of the field should be specified by symbolic names. The names may have plus or minus signs depending on whether the hardware operation needs a logical "Ø" or "1". The bits not specified will be assembled to values which are indifferent from the viewpoint of hardware control. The mnemonic specification has the advantage over the octal one /though it may result in the definition of a max. of 12 symbolic names/ because the ingenious choice of the names gives a precise and understandable description of the action performed by the instruction.

The advantageous features of the assembler created in this way are:

- The assembler itself may be easily and quickly obtained from the minicomputer assembler. Thus an assembler specially adapted to the given microprogramming task can be produced.
- System programs of the minicomputer help the process of microprogram generation. For example, the editing and correcting of source programs can be done in the usual way by the EDIT program of the minicomputer.
- Storing the source program files and the assembled binary files on disc, and updating and loading the programs are both simple and fast.

3. AUXILIARY PROGRAMS

Here we present the programs which are partly of help in writing and editing source microprograms and are partly needed to transform the source programs for the minicomputer assembler. The linkage of the programs, and the procedure for producing binary microprograms are shown in Fig. 2.



Generation of microprograms.

Fig. 2.

3.1. Memory Map program

When writing microprograms the programmer should always bear in mind the empty and the already used addresses of the microprogram memory because /as mentioned above/ the Intel 3001 MCU circuit is not able to address any specific place in the memory.

The Memory Map program helps the programmer with information about the used places by making a map of the microprogram memory. The map of two lineprinter pages is printed in an arrangement of 32x16 cells of the 512 memory places addressable by the MCU. A cell contains the name of the microinstruction and under the name the next microinstruction address environment /i.e. the row or column number/. The microinstruction name may consist of a maximum of 4 characters, the next address of 3 characters. On the map the unused cells are empty.

Arrangement and information of the map correspond to the MCU features /necessitating row or column specification for the next address calculation/. The Memory Map program can produce the map on the basis of two sources, viz.

- source microprograms,
- binary information.

The program parts of the microprogram written in submodules can be processed by the Memory Map program; at the end of processing the binary form of the map can be output if desired. Having written a new microprogram module the MM program prints the new, updated map from the new source microprogram module and from the previously punched binary information. In this way the map can always be enlarged and it follows the growth of the microprogram. Thus the MM program gives great support for writing further microprogram modules.

On the one hand, during program writing the unused memory spaces for new microprograms are in front of the programmer in clear format and on the other hand the MM program processing the new source program prints those memory addresses /by their row and column number/ which were incorrectly used, i.e. by more than one microinstruction.

3.2. Microprogram Address Editor /MADE/

During microprogram writing the requirement for the transfer of microprogram parts or the entire program in the microprogram memory occurs frequently. This task is facilitated by the MADE program, which processes the labels /i.e. the address of the microinstruction, and the statements specifying the next microprogram address; and helps their modification; the other parts of the microprograms remain unchanged.

The MADE program searches for those parts of the microprogram which are related to the addresses; these statements are typed and the program waits for the operator's command which may be:

- to leave the address unchanged
- to change the statement to another one
- to alter the address by a given value /displacement/.

The last comman can be a single one for the entire microprogram. In this case the MADE program checks whether the converted addresses overflow the memory address range; if so an error message is sent. Overflow may result from incorrectly specifying the displacement.

3.3. Microprogram Address Converter /MAC/

The MAC program has to convert the absolute addresses of the MCU instructions into relative ones. This conversion is necessary for the Assembler. The conversion means relative address calculation with a modulus depending on the jump instruction type.

The MAC program tests the microinstructions syntactically; it checks whether all for the microinstruction fields are specified, and if not: an error message.

In addition to these tasks the MAC makes a page editing on the source microprogram. It aims at avoiding the case when a form feed cuts a microinstruction in two when printing after assembly. Thus a listing is produced which is easy to manage.

3.4. Microprogram Compressor /MICO/

As has already been mentioned a memory place of 12 bits is assigned to each microinstruction field, thus after having been assembled there are fields which do not entirely fill their memory places. The necessary compression is made by the MICO program, which discards worthless information. The constants determining the length of the microinstruction fields can be found in the MICO program and can be modified if the structure of the fields is changed.

4. THE DEVELOPMENT SYSTEM

The Development System presented here was in fact used for the verification of the microprograms for an emulated small computer built from the Intel Series 3000. This microprogram developing and verifying system contains the emulated small computer - "microcomputer" - and a controlling minicomputer. The tasks of this latter were as follows:

- to store the macro- and microprograms of the emulated small computer,
- to preset the inner registers of the microprocessor to the initial values and then to reaout these registers after the running of the microprogram,
- to carry out other control and checking tasks.

The motives for such an arrangement of the system were:

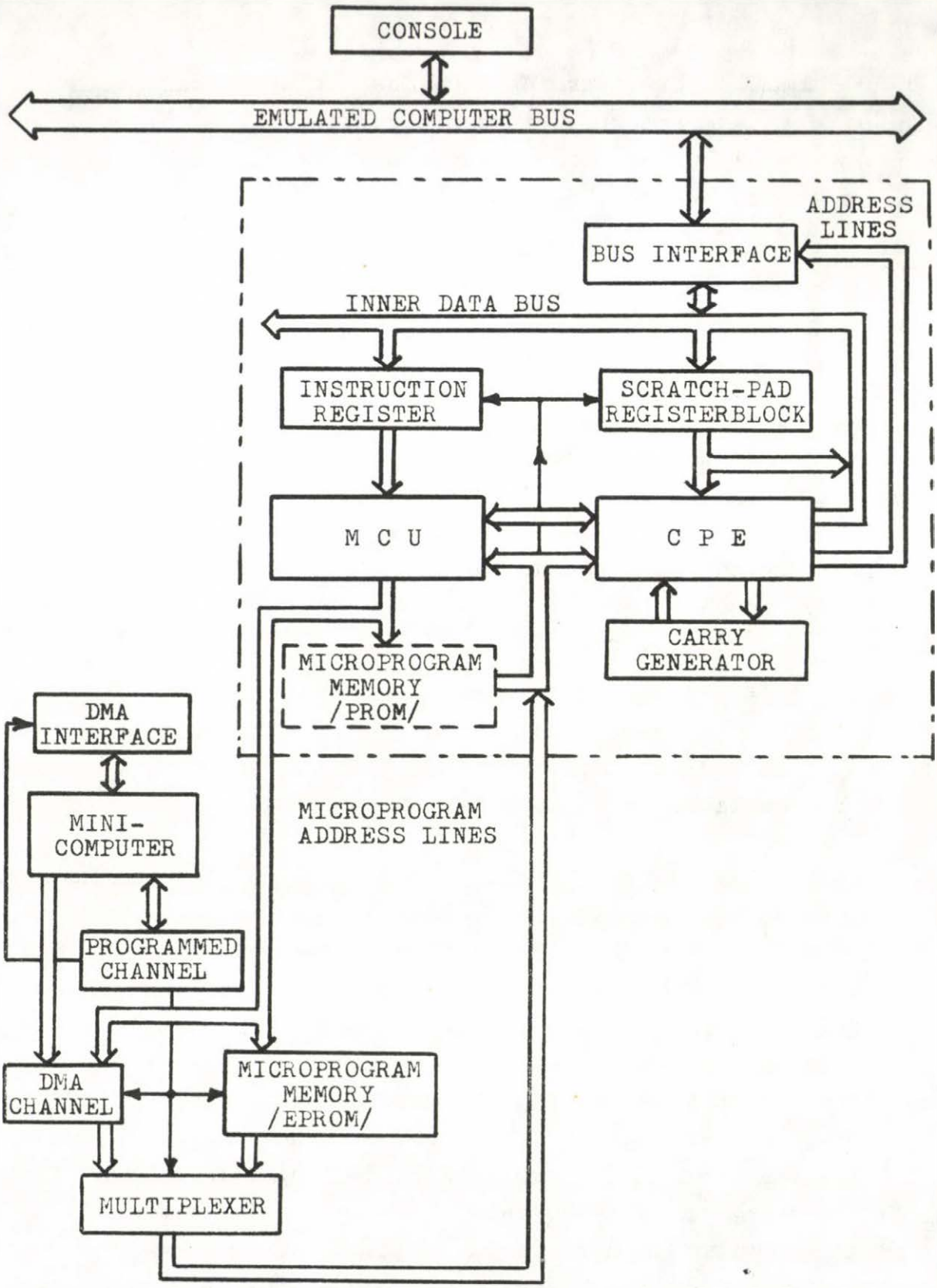
- A minicomputer with relatively many utility programs was available.
- The utility programs are necessary for elaborating micro-programs which need a great deal of mork.
- Editing, assembling and verification of microprograms and simultaneously checking the emulated computer hardware can be done on the same computer /minicomputer/.

The Development System hardware is essentially an "intelligent" microprogram memory consisting of

- The controlling minicomputer with its peripherals and disc.
- The direct memory access interface allowing the emulated computer to read macroprogram instructions and operands as well as microinstructions executing the macroprogram from the memory of the controlling minicomputer. The macroprogram memory is used by the microcomputer as if it were its own memory /the direct memory access channel of the Development System is connected to the microcomputer input-output bus/.
- The monitor program controlling the whole system. This program runs simultaneously with the macro- and microprograms.

4.1. Hardware structure of the system

The block diagram of the hardware components of the Development System is shown in Fig. 3. The part enclosed within the dotted line is the microcomputer. When this works independently it contains the PROM microprogram memory shown by the broken line. During development and verification the microcomputer gets the microprogram instructions either from the minicomputer memory or from the EPROM memory. In this latter case there is no essential difference in the timing relations of the independent or verified running of the microprograms.



Hardware block diagram of the Development System

Figure 3.

Input-output of the macroinstructions and operands and the input of the microinstructions are carried out by direct memory access. During controlled program running the system is commanded through the programmed channel interface.

4.2. Monitor program

Functions of the Monitor program are as follows:

- Macroprogram listing within the given address range and its checking and changing at desired addresses. When listing, the program types the contents of memory places between given beginning and final addresses; when checking, the contents are typed address by address. If changing is necessary the new content can be entered after the old one in the same line. The program then rewrites the corrected content of the current address.
- Loading macroprograms from paper tape and outputting macroprograms onto paper tape. For punching, both the start and the final address should be specified. The Information format of the emulated small computer /the absolute loading format/ is used.
- Microprogram listing, checking and changing. In listing mode the program writes a heading according to the microinstruction fields and the contents of microprogram words in octal form are typed between the specified beginning and final addresses. For checking single addresses the program types the actual values of the microinstruction fields of a specified address. During changing, an entire microinstruction or parts of it can be rewritten. The program then types the corrected microinstruction again. This possibility is useful when microprograms are being verified and some little correction needs to be made quickly since this avoids the need to correct and assemble the source program, which is a relatively long procedure.

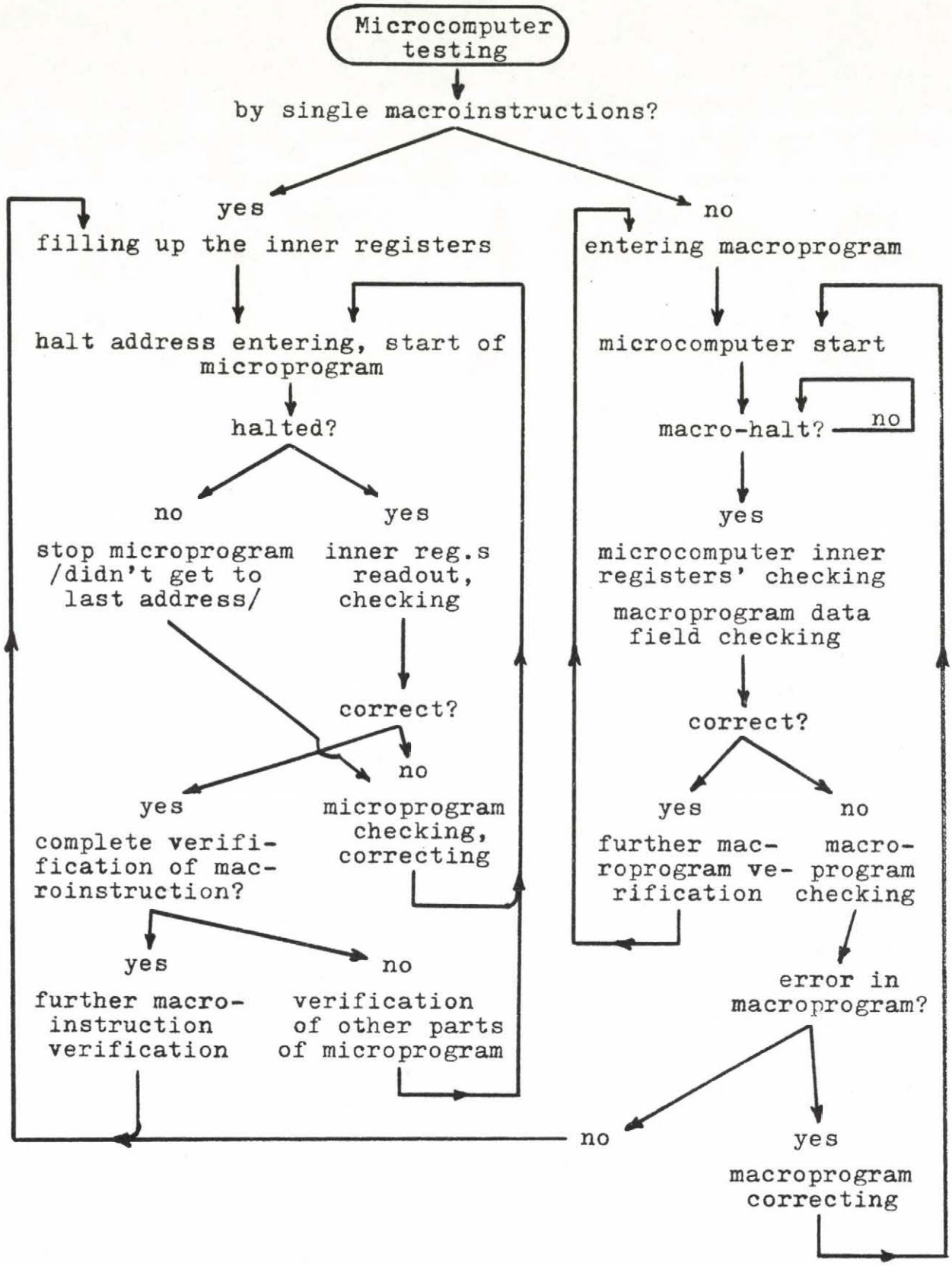
- Filling up the inner registers of the emulated small computer /i.e. the registers of the microcomputer/. The inner registers of the microprocessor are part of the inner registers of the microcomputer because the latter has a scratchpad register block as well . The filling up is carried out in two steps. First the operator specifies the register names and their contents by means of the console peripheral /contents of the unnamed registers remain unchanged/; the contents are stored in a table and do not change during microprogram running. This allows the restarting of the microprogram many times with the same initial conditions /register contents/. In the second step /which is started by the monitor program/ the microprogram built into the monitor program fills up the inner registers before the start of the microprogram to be run.

- Starting of the microprogram to be verified. After the operators command the monitor program waits for the last executed microinstruction address. At this address a microinstruction bit will be changed which makes the microprogram stop. The control then goes back to the monitor program which begins to read out the contents of the microcomputer registers.

- Readout of the inner registers of the microcomputer. This action - similar to the filling up the registers - is performed by a microprogram built into the monitor program. The contents are read into the memory of the minicomputer. Thus the operator is able to display and verify the contents of the inner registers of the emulated small computer.

5. VERIFICATION OF THE MICROCOMPUTER

Essentially verification is carried out by execution of single macroinstructions and complete macroprograms. The verification procedure is shown on the flowchart in Fig. 4. The system executes the macroprogram either by steps or continuously.



Procedure of testing the minicomputer

Figure 4.

During verification - when the system stops, the data fields of macroprogram and the inner registers of the microcomputer may always be checked. Should there be any errors, micro-program correction can be made by the monitor program on the level of the object program. For a comparatively large amount of correction it is advisable to carry over the modifications into the source program and to generate a new object program by means of the program package described above.

6. REFERENCES

[1] Barnes,D.H.,Wear, L.L.: Instruction tracking via micro-programming, Sigmicro '74

[2] Bouricius,W.G.: Procedure for testing microprograms, Sigmicro '74

[3] Davies,P.M.: Readings in microprogramming, IBM System Journal 1972.

[4] Intel Series 3000 Reference Manual Intel Corp. 1976.

[5] Miskolczi,J.: Microprogramming and microprogram languages, Automation 1976/12 /in Hungarian/

[6] Miskolczi,J.: Microprogrammable microprocessors, Measurement and Automation 1977.No.1. /in Hungarian/

[7] Vickery,Ch.: Software aids for microprogram development, Sigmicro '74

[8] CROMIS Cross Microprogramming System Reference Specification, Intel Corp. 1975.

[9] Signetics Micro Assembler Reference Manual, Signetics Corp. 1977.

[10] Mick,J.R.Schopmeyer,R.: MOS Support microprocessor teams with bitslice prototypes for easier microprogram debugging, Electronics 77/9

Session 3: Software

Сессия 3: Математическое

PROBLEMS IN MICROPROGRAMMING

G. DÁVID

Computer and Automation Inst.
of the Hungarian Academy of
Sciences, Budapest, Hungary

We want to give in this introductory paper a short overview of the history and the perspectives of microprogramming. The word "microprogramming" is meant here in both the sense of "programmed logic" and of "microcomputer programming". At the end of this paper we shall see the validity of this interpretation. The first chapter is concerned with microprogramming as "programmed logic", the second with "microcomputer-programming", while in the third chapter both will be discussed.

1. MICROPROGRAMMING

The concept of microprogamming was introduced as early as 1951 by Wilkes (9). Wilkes defined an intermediate level between the machine code and the hardwired components, recognizing the fact that a "few" basic logical operations in appropriate sequences can interpret the machine codes. This part was called by a "control section" and the sequences of operations by "microprograms". As it was originally published this supplied a computer--design method and this revolutionary concept was applied in research and development since the fifties and in practice since the sixties by the widely used, microprogrammed thirdgeneration computers.

The advantages in computer design can be summarized as follows:

- it is a tool with which to adapt hardware to software and application requirements

- it provides variable instruction-sets
- microprogramming is a systematical approach to designing the control section of a machine
- it provides a tool for emulation in applications as well as during the system development
- it eases maintenance

The disadvantages are also listed in the literature:

- unified notions were not used in microprogramming: a wide range of microprogrammable (even user-microprogrammable) computers were applied without basic common notions
- the architecture-description languages generally reflect special architectural philosophy
- the parallelism - and, consequently the synchronization - led to very complex control structures
- no hardware tools were provided to check the correctness of microprograms.

We want to refer here to most of the paper (especially those which are listed in the references at the end of this paper), giving practical solutions for some of these problems and discussing the pros and cons.

Microprogramming languages were developed to assist microprogram-design, and hardware-description languages and to give the semantics of these microprogramming languages. The users found themselves in the same situation as application programmers with the same problems but at a deeper and sharper level. Differences and identities between the styles of the work of a microprogrammer and of a programmer (in general) can be illustrated as

- there were two sets of algoritgmic languages (one set for each) but these had no language in common,
- when realized, a microprogram must be correct - a program may by incorrect,
- due to the fact that microprogramming is an adaptable system-architecture component, microprograms reflected both the

system-architecture and the application dield, but a wide variety of these fields existed.

These problems were internal (laboratory-level) ones in the industry but with the advent of user-microprogrammable machines (HP 2100 MX) these problems entered the application-field.

2. PROGRAMMING MICROMACHINES

In this decade microprocessors, and the components around them with which are built either a microcomputer or a microprocessor--system have led us to flourishing area in the application field. We do not want to summarize either the new applications or the new products, but to discuss them from a programming standpoint.

Microprogramming and micromachine programming have one feature in common and this is ROM or PROM-programming: the program produced is placed (burned) into a ROM or PROM and no further change is available in the case of a ROM, while a limited number of rewriting possibilities exist in the case of PROM-programs. Due to this fact the classical model of the programs and data in the same memory cannot be applied further, because these should be divided into two parts: control-section for programs and mostly registers, RAM's to store data.

New and important problems arose from the rapidly growing application fields and the variety of components in the micromachines. Some of these problems are

- during micromachine design the formerly applied design-technologies cannot be used, because the hardware and software and the "supervisors" should be designed together - hence a unified technology would be needed: the systemdesing technology.

- during micromachine development two kinds of developing tools are necessary: one for the hardware and one for the software. Generally these are based on different tools; hardware-development tools are based on compatible components, software--developing tools are based on incompatible computers. Consequently the programs produced by crossassemblers or cross--compilers (even if they were carefully tested) are not directly those which were required, because the requirements and the programs are expressed on another computer by a (possibly high-level) programming language. Hence it follows that system-developing tools are needed to develop both programs and microcomputers. The industry recognized this problem at an early stage.
- It would be another variation of the same theme to discuss the question of whether we really need isolated system-design and system-developing tools or not: it is possible either to integrate or to realize them in the same way.
- in micromachine applications the application-field requires a special architecture to be tailored. Here by the word "architecture" we mean both hardware and software. A small change in these requirements would disturb the special architecture and a tremendous amount of man-power would be wasted. To avoid this role of modular and structured programming principles is more important here.

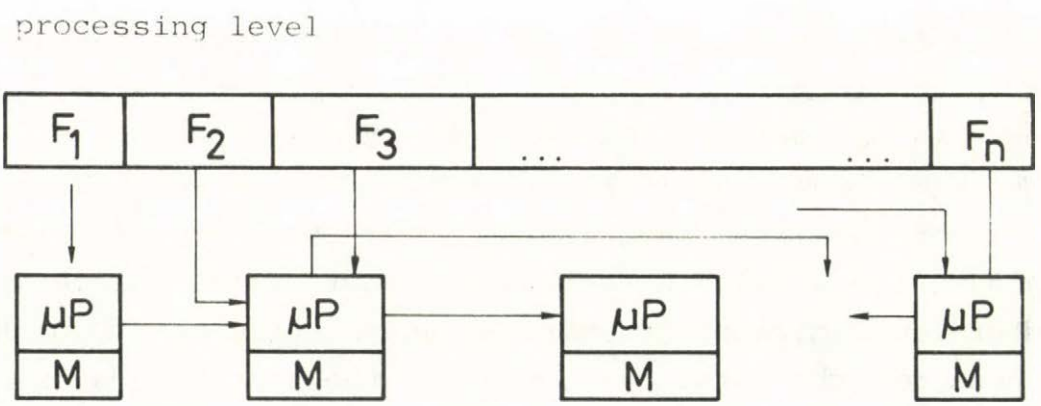
In order to solve these problem, the tools now available are

- higher-level languages (both macro- and algorithmic languages)
- verification-oriented languages
- program-developing kits
- modular design and developing tools
- structured (micro) programming technology

3. MICROPROCESSOR-BASED MICROPROGRAMMING

In recent years the applications of microprocessors as components of larger computer systems have been of growing importance. The main problems in this issue are merged from the problems of both microprogramming and microprocessor-programming, and on the other side the advantages of these two lines are summed up. Most of the open problems can be derived from this application-area, hence it - as a paradigm - reflects the nature of questions to be solved.

To illustrate this let us assume that a microprogram-memory word is given divided into different fields: F_1, F_2,...,F_n



field-interpreter microprocessors

which are interpreted by an appropriate set of (possibly different) microprocessors with their own memories M. Bit-sliced microprocessors are designed to realize these sorts of central processings units or processing elements. From a programming point of view we can see that the programming of the field--interpreter microprocessor leads us to those problems which are listed in Section 2 and programming this processing unit or element leads us to the problems in Section 1.

Some of these are reduced to simpler ones or resolved such as the problems of parallelism and synchronization on the level of microprocessors but not on the processing-level.

A new problem is that the architecture of microprocessors more or less determines the programming-level architecture. The number of bits in a register, the number of shared synchronization components etc. are such features of microprocessors, so that sometimes in order to build up a processing unit there is only one possible solution, and to design a significantly different architecture for other purposes is almost impossible. These statements are not valid at the processing level because the "micro-orders" (i.e. the contents of the fields) can be different in order to reflect the problems in applications.

Due to this fact the reconfiguration principle (i.e. the dynamic changes in the interrelation between the microprocessors in question) can be realized. Consequently the microprograms on the processing level should be written in such a way that either they are (more or less) invariant against these reconfigurations, or they have the available control structure as a "parameter". The higher-level microprogramming language approach supports the methods for the first case, but for the second today's programming principles are not enough.

In this conference we shall see some results in this direction. What we really need are microprogram and microprocessor-programming design and development tools integrated with the other components as a system design and development tool.

REFERENCES

1. C.Boon: Computer Design. Infotech State of the Art Report, 1974.

2. P.M.Davies: Readings in microprogramming. IBM Syst. Journal vol 11 nol pp. 16-40 (1972).

3. M.J.Flynn, M.D.MacLaren: Microprogramming revisited, Proc ACM National Meeting, pp. 457-464 (1967).

4. M.J.Flynn, R.F.Rosin: Microprogramming; and introduction and a viewpoint. Transaction on Computers vol C-20 no 7 pp. 727-731 (1971).

5. V.M.Gluskov: Automata theory and formal microprogram transformation. Cybernetics vol 1 no 5 pp. 1-9 (1965).

6. S.S.Husson: Microprogramming: principles and practices. Prentice-Hall Inc., New York

7. R.F. Rosin: Contemporary concepts of microprogramming and emulation Comp.Surveys vol 1, pp. 197-212 (1969).

8. R.F. Rosin: Contemporary concepts of microprogramming and emulation. Infotech State of the Art Report 2. Giant Computers (Boon C. /Ed), (1971).

9. M.V.Wilkes: The best way to desing an automatic calculating machine. Manchester University Computer Inaugural Conf. p.16. (1951).

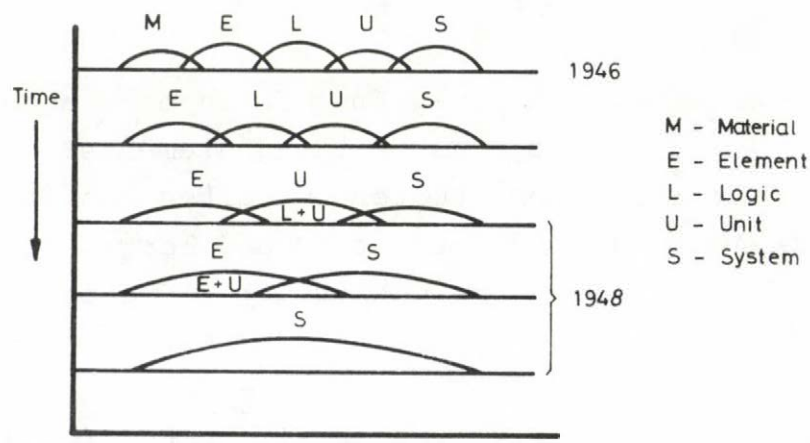
MICROPROGRAM STRUCTURES OF MICROPROCESSORS

R. W. MARCZYŃSKI

Institute of Computer Science of PAS
Poland

INTRODUCTION

The progress in the development of digital devices is characterized by a permanent increase in the complexity of elements and at the same time a decrease in the variety of raw materials used to build computers. Figure 1 shows this phenomenon in time.



Today, computers are built up from small SSI and MSI elements, as for example the CRAY 1, computer to one-chip VLSI microcomputers as the INTEL 8748.

EUS	EU	ES	S
Big Computer	Bit-Slice Micropr.	Microprocessors	One Chip Micropr.
CRAY-1 Vector Processor Four Elements /Fairchild Motorola/ 138 * 10^6 /sec	Fairchild 9405, 9408 Texas Inst. SPB0400 Intel 3002, 3001 Monolithic Memories 6701,6710 Signetics 8X300,8X02 ADM 2901,2909 Motorola 10800,10801	Z-8000 Intel 8086 Motorola MACS /Advanced Computer System/ PDP 11/45	Mostek 3870 Intel 8748 Z-8

Table I

In Table I the actual trends in these developments are presented. In the two right-hand columns devices with fixed instruction sets and fixed structures are shown. These devices are built using unipolar technology. The two left-hand columns contain devices whose structures can be defined by constructors. They are built using bipolar technology.

The only approach which gives full freedom to a constructor uses gates and flip-flops as building elements. The bit-slice architecture gives a constructor a limited possibility of choosing the length of word, the type and length of instructions, an instruction set and some possibility of choosing the computer structure.

The history of computer evolution allows the formulation of the following conclusion:
We look for modular solutions from hardware through firmware to software.

During the sixties there were two trends in this approach.
One of them has its origin in the theoretical works which were oriented on universal logic modules. It means that such a module can realize all logical functions for a fixed number of variables.
The second trend has its origin in homogeneous cellular elements.

Unfortunately, these two trends have not found any repercussion in the production of IC elements. The analysis of a production trend of integrated digital elements shows that the production types have their origin in the structural divisions of computer structures. A classification of IC products would be very useful. The classification should be carried out using the implementation mechanisms in computers. Bit-slice microprocessors are elements of this kind. These modules provide an opportunity for the realization of computers by assembling bit-slice elements.

The basic assumptions for developing bit-slice elements were:

1) Vertical partitioning of computer structures
2) Using microprogrammed structures

The first assumption is obvious in parallel computers where vertical partitioning was used to implement boards in the old technology. It provides an opportunity to chose the required length of words in the designed machine and is also more suitable for IC technology. It leads to elements with a small number of pins and moderate power dissipation.

The second assumption, however, introduces some standard structures which help manufacturers to define modules as well as providing an opportunity for defining the designer's own instruction set. It also gives the designer a very handy methodology for designing the control unit in a processor, and due to this standard structure helps the integrated circuit designer in the implementation of the more powerful elements.

Advantages of microprogrammed structures are:

- flexibility
- intrinsic design methodology.

A disadvantage is lower resulting speed.

The microprogrammed implementation requires a very fast microprogram control memory to obtain the necessary speed, especially if we use horizontal microprogramming. The microprogram control memory should be several times faster than the main memory. If this condition is not satisfied many advantages disappear. The speed factor of the control memory is not so critical in solutions with vertical microprogramming.

MICROPROGRAMMED PROCESSOR STRUCTURE

The principal microprogrammed processor structure is shown in Figure 2.

The processor consists of an arithmetic logical unit - ALU, a microprogrammed control unit - MCU, a main memory - M, an instruction register - an IR, an instruction counter - IC, an I/O Interface, a Clock and a Bus.

The processor activity is bounded by data and instructions coming through the BUS and switches which are controlled by a control signal generated by the emitter of the MCU and a series of clock pulses (not shown in the figure).

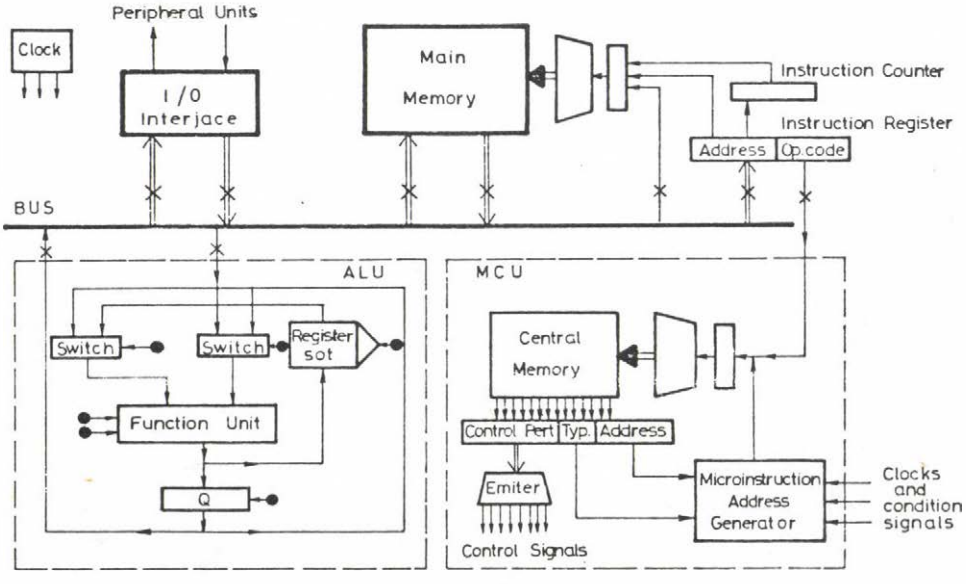


Fig. 2.

A microprogrammed processor

There are two levels of microprogrammed processor activity--instruction or macro-level and microinstruction or micro-level. The macro-level has two phases:

- phase 1 - fetch of instruction from memory
- phase 2 - execution of instruction

Activities on the macro-level are controlled by microprograms. Phase 2 of macro-level realization on the micro-level starts by decoding the operation part of the instruction contained in the IR. The decoded operation part serves as the first address of the microprogram stored in the control store. Next, the first microinstruction is transferred to the microinstruction register. The control part of the microinstruction is used to generate the control signals which control execution of the prescribed micro-operations. Then the address of the next microinstruction is generated and the next microinstruction is executed. This procedure is repeated. The last microinstruction of the microprogram causes the fetching of the next instruction of the program at the macro-level. A new instruction is executed in the same manner on the micro-level.

The method of generating the next address of the microinstruction is a necessary factor in the description of an MCU element. The ALU can be consideres as a set of storage elements (registers), a set of function elements and paths between them. The ALU activity consists of a series of elementary register transfers. The function elements can be independent of storage elements, i.e. they are special elements for processing only, or dependent, i.e. they are linked with definite storage elements.

The ALU realizes a register transfer as an elementary micro-operation. The number of possible simultaneous transfers depends on the ALU construction.
Every transfer and its function is controlled by control signals from the MCU and the clock. Operation on one storage element can be considered as a transfer operation on itself. To characterize the ALU structure it is enough to give a set of all admissible register transfer and their functions. A simple ALU structure

is presented in Figure 2.

The other parts of the microprogrammed processor operate like typical units in a conventional processor.

CLASSIFICATION OF BIT-SLICE MICROPROCESSORS

Bit-slice microprocessors are produced by many manufactures but at present generally accepted standards do not exist. The produced elements have many differences.

Processing elements

The Classification concept for ALU-s is based on the fact that in microprogrammed machines the processing operations are sets of microoperations. A Microoperation is a simple register transfer operation during which some functions are performed. The activity of microprogrammed ALU-s can be described as:

$$D := S^{I} \rho_i S^{II}$$

where D is a place of destination, S^{I}, S^{II} are sources and ρ_i is a function which is executed during the transfer.

Each transfer implements a part of a microinstruction. The description of all possible combinations of these transfers gives us the operational and structural possibilities of a bit-slice arithmetical unit.

The structure of an ALU is characterized by a set of destinations and sources and their properties. The various structures of an ALU contained in bit-slice microprocessors have been identified by Aspinall [2]. A destination place can be a register inside an ALU or output pins. Sources can be registers inside an ALU and/or input pins.

An ALU structure is defined by a set of sources and destinations. In additional tables possible transfers between sources, the set of destinations and the set of possible operations are described.

The Advanced Micro Devices 2901 shown in Fig.3. is an example of a bit-slice ALU [3].

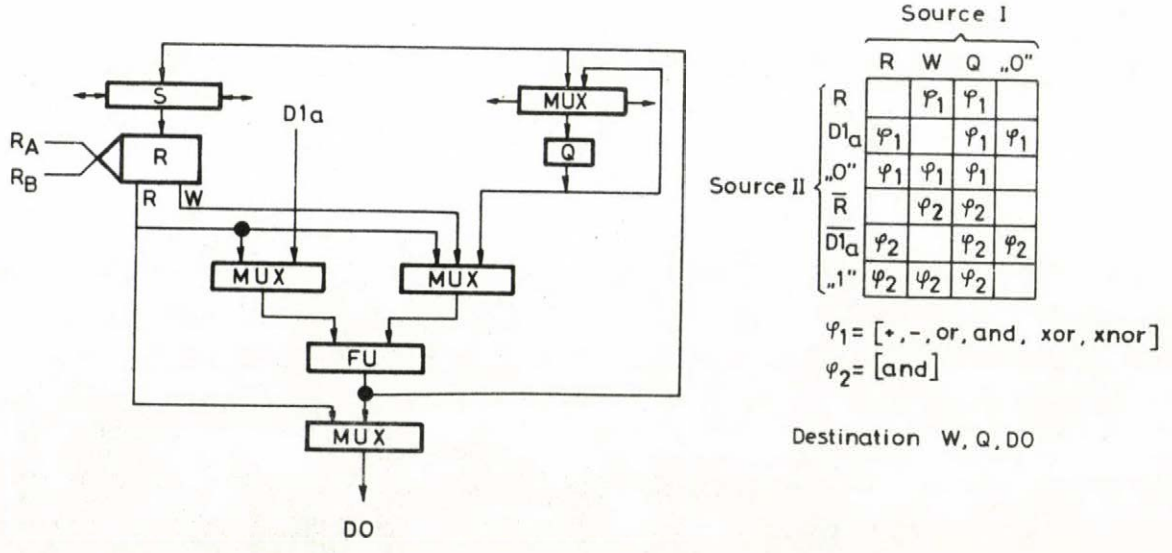


Fig. 3.

The data structure of the AMD 2901 bit-slice device (see Figure 3) has seven extra registers in the FU feedback path. The register Q has a shift facility and two multiplexed inputs. The register array has the interesting feature of being a double read register.

The device allows two registers in the array to be read at the same time (R and W) by means of two separate register addresses (RA and RB). Data may only be written into the W-register. The data may also be shifted, left or right, before being loaded into the register.

In Table 2 the classificaiton of current bit-slice ALU devices is shown.

Bit-slice processor elements	Structure	Slice width /bits/
Fairchild 9405	DI; FU.R; DO	4
Intel 3002	DI; FU,R,Q; DO	2
Texas SBP0400	DI; FU,R,P; DO	4
Monolitic Memories 6704	DI; FU,R/R,Q; DO	4
AMD 2901	DI; FU,R/R,Q; DO	4
Motorola 10800	D_aI; R/R,Q; D_bI/O	

where

FU - Function unit
R,P - Register sets
Q - Register
DI - Data input
DO - Data output
R/R - Double acces register set
I/O - Switched Input/Output

Table 2.

Microprgrammed control unit

Microprogrammed control unit structures can be classified with respect to the mechanism of the selection of words in the control store. This is the next address generation. The chosen solution to this problem is reflected in the hardware structure of the control store address generator. The structure of the generator determines the general characteristics of the control unit.

Being more precise, this classification will be made with regard to the sources of the next microinstruction address.

The primary or basic address sources are a microinstruction counter and an address in the current microinstruction. The secondary or auxiliary address sources are a stock and an auxiliary random access memory. These four elements define sixten possible solutions.

Primary source		*Secondary source*		*Name*
Counter	*Address*	*Stack*	*Aux.Memory*	
X				C
	X			A
X	X			CA
X	X	X		CAS
X	X		X	CAM
X	X	X	X	CASM

Table 3

Table 3 contains the most representative solutions. The two first rejected cases are the "stack" and the "auxiliary memory" alone cases, which do not fulfil the condition that a basic source must be present in a solution. As far as the other cases absent in the table are concerned their structures and properties are restricted versions of the cases presented in the table.

The function for generating the address of the next microinstruction can be written as follows:

$$A = F(C,AS,M,t,c)$$

where F is a function of the MCU, C,A are primary generators, S,M are secondary generators, t is the type or operation mode of microinstruction and c means conditions.
The type of microinstruction control unit structure is characterized by primary and secondary generators. The t and c are usually are given in the form of a table in which all control details are presented.

This classification of MCU structures with the use of a table is sufficient for the description of a microprogrammed control unit.

A more detailed description of MCU features is given in [4,5]. In table 4 the most important features of the MCU are given in a condensed form.

STRUCTURE Type	next address		others features				
	linear	random	branch	loop	multi branch	modular	dynamic
C	+	-	possible	difficult	-	-	-
A	+	+	possible	possible	-	-	-
CA	+	+	+	+	-	-	-
CAS	+	+	+	+	+	+	-
CA M	+	+	+	+	+	possible	+
CASM	+	+	+	+	+	+	+

Table 4

We assume that all MCU-s have some condition possibility dependent on a result of the executed microinstruciton.

In the Table 5 a classification of current bit-slice MCU's is presented.

Type of bit-slice MCU	Structure	Stack depth
Intel 3001	A	-
Texas 74161	C	-
Texas 74S482	CAS	1
Monolithic Memories 6710	CAS	4
Signetices 8X02	CAS	4
AMD 2902	CAS	4
Fairchild 9408	CAS	4

Table 5

Most of the produced bit-slice MCU's have a CAS structure which provides an opportunity for modular microprogramming, but small stack depth limits this feature. The Intel 3001 has an A structure only with some limitation of word length. This structure is very fast, but it is difficult to write microprograms for it.

CONCLUSION

The Classification of bit-slice microprocessors presented in this paper had two approaches. One is based on the identification of already produced structures and their classification [5]. The second approach is based on an anticipation of future structures.

Both approaches are very useful and shown how to define and classify LSI modules and blocks in order to stimulate manufacturers.

REFERENCES

1. C.J. Walter, A.B. Walter: The Significance of the Next Generation Fourth Generation Computers. Prentice-Hall pp. 11-29.
2. D. Aspinall: Microprogrammable microprocessors Tutorial paper at CERN School of Computing (1976).
3. M. Edwards, E. Dagless: LSI microprogrammable microprocessors. Vol. 1, No. 7, October 1977, pp. 407-414.
4. R. Marczyński, M. Tudruj: Une Approche de Structuration des Unites de Commande Microprogrammes, IRIA Research Report No. 47, January, 1974.
5. R. Marczyński, M. Tudruj: Microprogrammed Control Units: towards modularity in microprogramming 2nd Symposium on Microarchitecture (1976), pp. 173-181.

MODULAR MICROPROGRAMMING APPROACH IN MICROPROCESSORS

M. TUDRUJ
Institute of Computer Science
of the Polish Academy of Science
Warsaw, Poland

INTRODUCTION

One of the most important elements of a microprogrammed control unit is the control store address generator, which prepares address for consecutive microinstructions to be read from the control store. The features of this generator are crucial if complicated control schemes have to be embodied in microprogramms. The growth of complication at the software level and the need for more and more sophisticated computer instruction sets put strong demands on the microinstruction sequencing aspect of microprogramming, which became the commonly used means for solving software/hardware articulation problems.

The achievements of LSI technology gave important support for the development of microprogrammed control unit structures. The advent of microprocessors showed at first rather little interest in applying microprogramming, which was the consequence of the simplicity of the early microprocessor designs. Microprogramming was then used only in two microprocessors of the bit-slice type, namely Intel 3000 and MMI 6700 series. The growth of interest came after 1975, when technology permitted the manufacture of more developed designs. Since then microprogramming became a commonly used technique in bit-slice microprocessors, and even recently, it has proved to be useful in 16-bit monolithic designs such as CP 1600 or PAGE microprocessors.

The bit-slice microprocessor designers could use all the microprogramming experience which has been accumulated since the early years of microprogramming. So now most bit-slice microprogrammed control units have facilities which permit quite complex control structure in microprograms, including the structures which lead to modularity in microprogramming.

Microprogram modularity concept

The idea of modularity in microprogramming was implied by three trends which appeared in the development of microprogrammed control units:

- the minimization of control store volume by applying the subroutine technique to microprograms,
- the need for standard control constructs provided in the control units, enabling simple implementation of flowchartable microprogram logic,
- the attepmts to implement complex control structures in microprograms through the application of structured programming techniques.

When the modular approach is applied, a microprogram is in fact a sequence of conditional or unconditional references to the set of standard microcode modules. It does not exclude, however, some straight line microinstruction sequences in microprogramms. The exemplary modular microprogram structure is shown in figure 1.

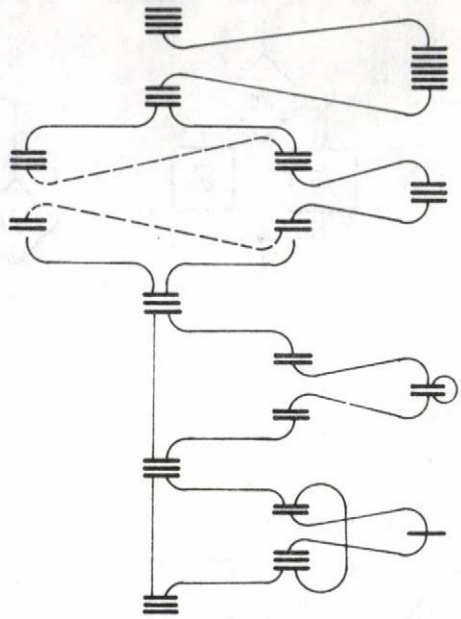


Fig. 1.

A microcode module is a sequence of microconstructions which has the following features:

a/ it has distinguished entry and exit points,
b/ it can be entered an executed independently of the context by which it has been referenced,
c/ the return to the calling context is executed automatically,
d/ it can contain conditional or unconditional references to other microcode modules.

The basic primitives for modularity are call and return operations added to the control store addressing functions. These operations should be combined with branch operations upon the result of computer status testing.

The following control constructs are essential for the modular structure of microprograms, figure 2. The microprogrammed control unit oriented on modularity should enable their straightforward implementation.

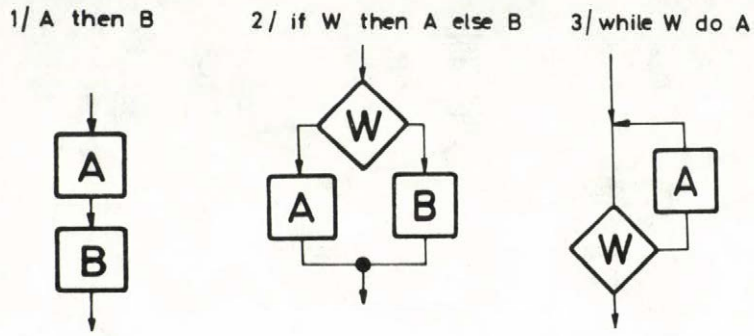


Fig. 2.

In these constructs A and B mean microcode modules, which can be straight line sequences of microinstructions or can be microcode structures defined recursively from the three constructs shown above.

To make the modularity more efficient, the "case" construct can be added to the constructs depicted above.

The modularity in microprograms should be differentiated from the commonly used subroutine technique in microprograms. A microcode subroutine has almost all microcode module features except, that a subroutine does not have to be common for many microprogram sections and be independent of the calling context. Hence it is allowable for a microcode subroutine to have a particular address embedded in its return microinstruction. For example, it can be necessary to enable an implementation of one of the basic control constructs stated above. This is not allowed for microcode module return microinstructions, which cannot be a priori particularized to any address.

A further requirement that could be imposed on microcode modules is that they have no explicit addresses in their bodies at all. This requirement seems to be too excessive for the current state of microprogramming engineering development, so it will not be enforced in this paper. However the control store address generator described in the final part of this paper has some

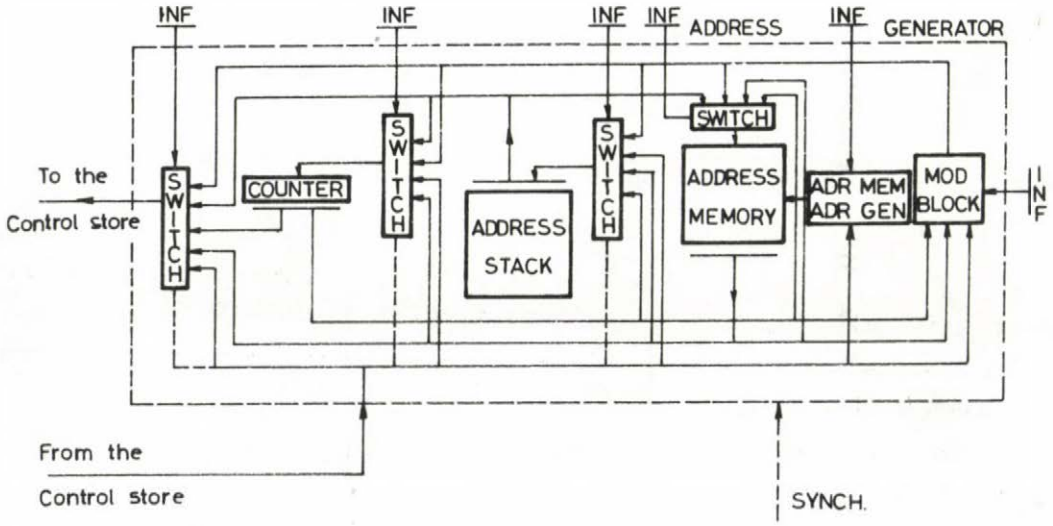
facilities for an implementation of this feature.

Hardware structures.

The classification of the control store generator structures given in [6] shows that only those structures which contain the address stack or the address memory can implement modularity in microprograms. These structures are:

1/ "follow counter / address + stack" - CAS type,
2/ "follow counter / address + memory" - CAM type,
3/ "follow counter / address + stack + memory" - CASM type.

The third of the structures which is the most evolved one is shown in figure 3.



There are 3 elements in this structure which store the control store addresses:

- the counter, which is used for accessing the consecutive locations in the control store; it is used mainly for addressing microinstructions inside the microcode modules,

- the stack, which is used for storing return addresses from microcode modules; it can be also used for branch addresses,
- the auxiliary address memory, which is the additional source of addresses; in modular environment the memory can store sequences of references to the modules held in the control store.

The review of bit-slice microprogram control unit structures contained in [8] shows that most of these structures are of CAS type or CASM, if some additional registers used for intermediate storage of addresses can be treated as a substitute for a memory.

The table given below shows bit-slice microprocessor control units which have stacks for microprogram addresses and gives their stack depths and numbers of control store addressing functions.

Table I

Microprocessor type	Stack depth	Number of cs addressing functions
MM 6710	1	8
AM 2909	4	12
SIGNETICS 8 x 02	4	8
TEXAS 74S 482	4	64
FAIRCHILD 9408	4	13
MOTOROLA 10801	4 (expandable)	16

The depth of stacks contained in these control units is limited to 4 and only in the M 10801 unit is a control provided to enable the expansion of the stack. Numbers of addressing functions vary from 8 to 64 in the Texas 74S482 unit, which has the full adder inside, especially for control store addressing purposes.

The control store addressing functions of the above units can be classified in one of the following categories:

a/ basic functions, which are unconditional and are sets of such primitives as microprogram counter incrementation, push or pop of a stack, register contents transfer or the like.

b/ higher level functions, which are conditional and are composed of basic functions; they are wired inside the control unit chip and hence are ready for a microprogrammer.

The AM 2909, SN 74 S 482 bit-slice control units have only basic addressing functions which have to be combined in more complicated conditional ones using additional circuitry outside these units. The remaining control units of Table I have addressing functions of both types, which are up and need not be combined by any outside unit circuitry.
The analysis of the addressing functions of the control units mentioned above showed that all these units can implement an evolved microcode subroutine technique rather than a modular technique in the sense assumed in this paper.

Let us now discuss these problems in more detail, taking as an example the AM 2909 microprogram control unit [9]. The global scheme of the unit is shown in figure 4.

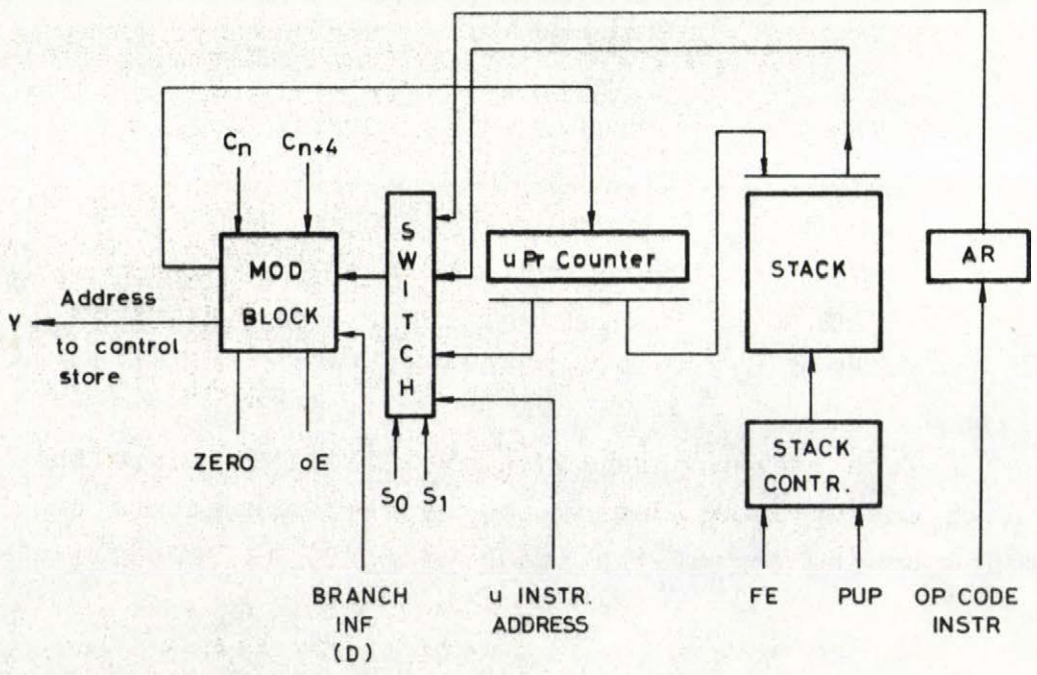


Fig. 4.

There are different control signals applied in the scheme:

S_o, S_1	select the address source for the modification block
FE	activates and de-activates the stack
PUP	controls POP or PUSH operation on the stack
ZERO	forces the address to the zero state or enables OR - modification with BRANCH INFORMATION
OE	opens the address output from the modification block to the control store

The values for the control signals must be prepared outside the AM 2909 unit in the additional logic. The signals are in general a combination of addressing control bits of the current microinstruction and the AM 2901 CPU state test result: There are 12 addressing functions in AM 2909, which are controlled by S_1, S_o, FE, PUP signals. They are shown below:

S_1	S_o	FE	PUP	SYMB	FUNCTION
0	0	0	0	POP	Pop, y←µPC, µPC←µPC + 1
0	0	0	1	PUSH	Push, y←µPC, µPC←µPC + 1
0	0	1	X	STEP	y←µPC, µPC←µPC + 1
0	1	0	0	POP AR	Pop, y←AR, µPC←AR + 1
0	1	0	1	CALL AR	Push, y←AR, µPC←AR + 1
0	1	1	X	JMP AR	y←AR, µPC←AR + 1
1	0	0	0	RTS	y←STKO, µPC←STKO + 1, Pop
1	0	0	1	RTS1	y←STKO, Push , µPC←STK1 +
1	0	1	X	LOOP	y←STKO, µPC←STKO + 1
1	1	0	0	RTSD	y←D, Pop, µPC←D + 1
1	1	0	1	CALL D	y←D, Push, µPC←D + 1
1	1	1	X	JMP D	y←D, µPC←D + 1

where µPC is the microprogram counter; STKO, STK1 are the top and next to top locations of the stack; Push enters the µPC contents to the top of the stack; for Y, AR, D see fig.4.

A *then* B

1. CALL D (D=A_1)
2. CALL D (D=B_1)
3.

where A = A_1 x x x
. . .
A_n RTS

if W *then* A *else* B

1. AR=D (D=A_1), STEP
2. *if* W *then* CALL AR (AR=A_1)
   else CALL D (D=B_1)
3.

B = B_1 x x x
. . .
B_m RTS

while W *do* C

I. 1. JMP D (D=C_1)
2.
where C: C_1 PUSH
. . .
C_k *if* W *then* LOOP
else RTS D (D=2)

II. 1. CALL D (D=C_1)
2.
where C: C_1 x x x
. . .
C_k *if* W *then* JMP D (D=C_1)
else RTS

III. 1. AR←D, CALL D (D=C_1)
2.
where C: C_1 x x x
. . .
C_k *if* W *then* JMP AR
else RTS

It can be seen that there is no problem with implementing "then" and "if-then-else" structures in this unit with standard microcode modules A, B.
However it is not so in the case of the "while-do" structure. Three possible solutions of this structure are shown above. Some of them are very close to the modularity concept, but they do not fulfil all the modularity requirements. In the first solution C is a microcode subroutine, which ends with a conditional loop/return microinstruction. The return address is embedded in this microinstruction, hence the solution cannot be used in the modular environment. In the second solution C can be treated only as a special kind of microcode module, which ends with jump to C_1/return microinstruction. The C_1 address is contained in jump to C_1/return microinstruction. C can be used only in looping control structures and so it is not a standard module common for any microprogram. The third solution is like the second one, but the explicit address has been eliminated from the jump/return microinstruction, thanks to the use of the AR register as the source of the jump address. However in this solution also C cannot be a standard module, because it does not have a standard return microinstruction at the end.

A modularity oriented control store address generator.

In this part of the paper an outline of a control store address generator unit will be presented which enables straightforward and efficient implementation of the modularity in microprograms. The structure for this unit has been proposed in [6].

In this unit it is assumed that each microcode module has a standard structure and therefore is a sequence of microinstructions ended by a microinstruction with the unique return code for all modules. Microprograms in this unit make sequences of references to the set of standard modules stored in the control store.

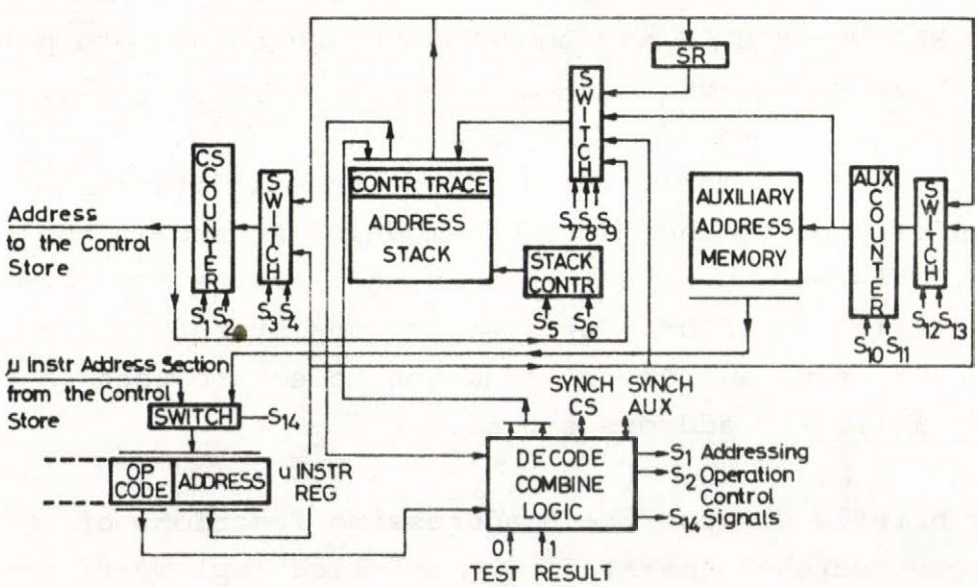
There are 8 control store sequencing function codes utilized by a microprogrammer which can be interpreted differently in the unit to provide 11 effective sequencing primitives. This is a practical minimum set of functions which should be treated as a frame for a real control unit.

These 8 sequencing functions can be subdivided into four main categories:

1. Ordinary sequensing functions - STEP, JUMP
2. Parameter function - STORE,
3. Module calling functions - CALL CS, CALL AUX, BRANCH, LOOP-BRANCH,
4. Module returning function - RETURN.

Ordinary sequencing functions ensure primarily all the addressing inside microcode modules. The remaining 3 types of functions provide sequencing at the microcode modules level.

This control store address generator is of the "follow counter/ address + stack + memory" type. The scheme of this generator is shown in the figure 5.



There are four devices in this structure used to store addressing information: the control store, the address counter, the address stack and the auxiliary address memory. The microinstruction word contains the control field and the addressing field. The addressing field contains a code of the sequencing operation to be performed by the address generator and, depending on this code, the next microinstruction address.

The stack is used to store return addresses and some parameters of module references. The stack word has the TRACE field containing the address and the CONTROL field, defined for every address introduced to the stack.

When a module is called, the return address for the TRACE field and the appropriate code for the CONTROL field are prepared and stored at the top of the stack.
This CONTROL code depends mainly on the control constructs in which the module will be used. It is a function of the operation code in the microinstruction addressing field, the test result and the current contents of the CONTROL field at the top of the stack.

The CONTROL field at the top of the stack is always examined before the RETURN, BRANCH and LOOP-BRANCH operations and parameterizes their execution.

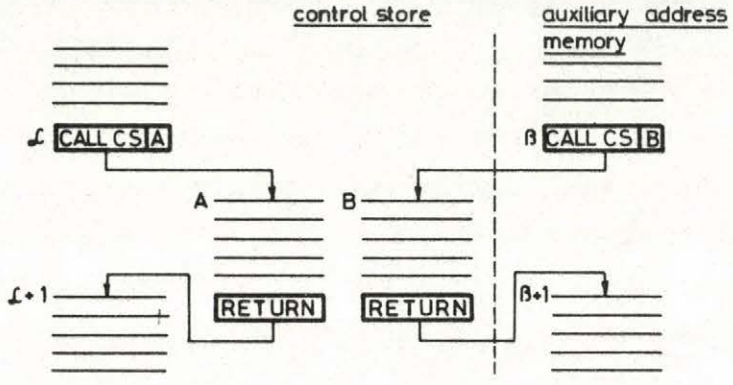
The auxiliary address memory is used to store additional modules built of references to the microcode modules stored in the control store. This memory word format is the same as the format of the addressing field of microinstructions stored in the control store. The same addressing function codes are used in both control and auxiliary address stores.

Let us now briefly discuss the 8 addressing functions of this control store address generator. The detailed implementation of these functions is explained in Tables 2 and 3.

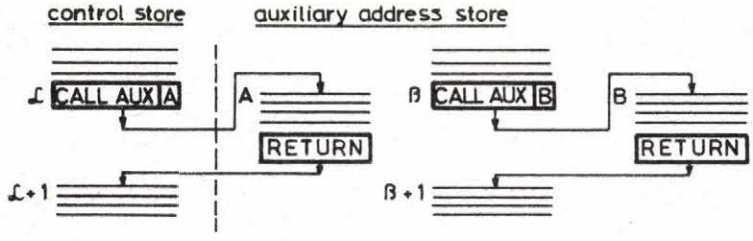
STEP and JUMP are commonly used functions of the microprogram counter incrementation, and the unconditional jump to an address contained in the current microinstruction.

STORE is a function which performs a push on the address stack and enters the address of the current microinstruction to the top of this stack. With this address special control information is stored in the stack which indicates that it is branch operation. When STORE is performed in incrementation of the microprogram counter is also performed.
The flow - diagrams of module calling functions are shown in figure 5.

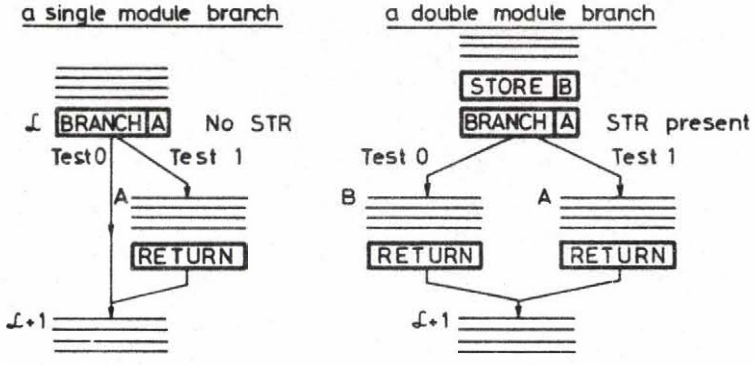
CALL CS



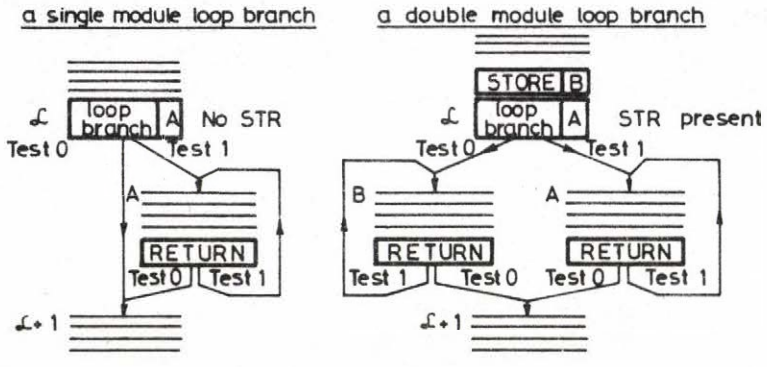
CALL AUX



BRANCH



LOOP - - BRANCH



CALL CS and CALL AUX are unconditional calls of modules stored in the control store or in the auxiliary address store respectively. Both operations can be is from the control store and from the auxiliary store. The return from a module is to the store from which the call was fetched. The return address is stored in the stack, with corresponding control information.

The BRANCH function code can be interpreted in two ways: as the single module branch and the double module branch. The single module branch is in fact the conditional call of a module stored in the control store at an address placed in the current microinstruction.

The double module branch is the branch between two modules, stored at the arbitrary addresses in the control store. One of these addresses is placed in the current microinstruction, the other is taken from the stack.

The double module branch is performed only if this alternative address has been prepared previously by the STORE operation and is now at the top of the stack. This is detected through a test of the CONTROL field of the top of the stack. When the BRANCH operation is executed, the return address from the module and the CONTROL code are stored in the stack, which will make the RETURN operation unconditional. If the BRANCH code was fetched from the auxiliary address memory the return is made to this memory.

The LOOP-BRANCH function code is used when modules are to be executed in loops upon a condition.

This code can be interpreted as the single module loop-branch or the double module loop-branch. The first one is the conditional execution of a single loop containing a module stored in the control store. The beginning address of the module is given in the current microinstruction. The double module loop-branch is the branch between two loops, each with a module inside. The double module loop-branch is performed only if the address of the alternative module has been prepared before in the stack by the STORE operation. When the LOOP-BRANCH operation is performed,

two addresses are stored in the stack together with corresponding codes for the CONTROL fields. One of them is the return address from the module, the other is the address of the beginning of the module effectively executed. The codes in the CONTROL field will cause the return from the module to be executed as a branch between these two addresses. If the LOOP-BRANCH code has been obtained from the auxiliary address memory the return is made to this memory.

There is only one RETURN function code used in microcode modules. This RETURN code can be the unconditional return to the address, which is the TRACE field content at the top of the stack increased by 1, or the conditional return, which is in fact the branch between two returns to addressed stored in the two uppermost locations in the stack. The condition for this branch is the test result. When the RETURN code is encountered in the module, the CONTROL field at the top of the stack, and depending on it also the test result, determines the way in which the return is performed.

This parameterization of the RETURN operation permits three effective return functions using only one code in the microinstruction. The same RETURN code is used for modules stored in the control and auxiliary address stores. The store to which the effective return will be performed is indicated by the CONTROL code stored in the stack together with the return address.

Using the sequencing functions of this unit the basic control constructs can be implemented as shown below.

A than B

1. CALL A_1
2. CALL B_1
3. ...

where A: A_1 x x x
⋮
A_n RETURN

if W than A else B

1. STORE B_1
2. BRANCH A_1
3. ...

and where B: B_1 x x x
⋮
B_m RETURN

while W do A

1. LOOP-BRANCH A_1
2. ...

if W_1 *than (while* W_2 *do* A)
else (while W_3 *do* B)

1. STORE B_1
2. LOOP-BRANCH A_1
3. ...

It can be checked that these basic structures can be embedded in each another using a set of really standard microcode modules.

Conclusions.

An approach to modularity in microprogramming has been presented in this paper. The basic microprogram modularity features have been discussed. Then the implementation of these features in microprogram control units of bit-slice microprocessors are analysed, taking as an example the AM 2909 control unit.

Next a solution of a control store address generator has been presented, which enables efficient modularity in microprograms. There are two features of this generator which should be emphasized: the extensive use of the address stack and the use of the auxiliary address memory. In this solution control store addresses are stored in the stack, associated with control information.

Table I I

μinsts. Address	Addressing Operation Code	Address in μins.	Top of the Stack before Operation CONTROL		TRACE	Test	Next Address Generated X[1]	Stack Oper.	Top of the Stack after Operation CONTROL		TRACE	Remarks
α	STEP	-	*[2]	*	*	*	α+1	-[3]	**	**[4]	**	STEP is used only in the contr.store
α	JUMP	A	*	*	*	*	A	-	**	**		
α	CALL CS	A	*	*	*	*	A	PUSH	Δ[5]	UNCD	α	
α	STORE	A	*	*	*	*	α+1	PUSH	Δ	STR	A	
α	BRANCH	A	*	$\overline{\text{STR}}$[6]	*	0	α+1	-	**	**	**	single module branch
						1	A	PUSH	Δ	UNCD	α	
			*	STR	B	0	B	CLEAR TOP	Δ	UNCD	α	double module branch
						1	A	CLEAR TOP	Δ	UNCD	α	
α	LOOP-BRANCH	A	*	$\overline{\text{STR}}$	*	0	α+1	-	**	**	**	single module loop-branch
						1	**	PUSH	Δ	UNCD	α	
							A	PUSH	Δ	CD	A	
			*	STR	B	0	**	CLEAR[7] TOP	Δ	UNCD	α	double module loop-branch
							B	PUSH	Δ	CD	B	
						1	**	CLEAR TOP	Δ	UNCD	α	
							A	PUSH	Δ	CD	A	
α	CALL AUX	A	*	*	*	*	A	PUSH	Δ	UNCD	α	
α	RETURN		For RETURN see Table III.									

1) for CALL CS, BRANCH, LOOP-BRANCH: X is for the control store; for CALL AUX: X is for the auxiliary address store; for STEP, STORE, JUMP: X is for this store from which the operation code has been fetched.
2) * means "does not affect the decoding of the operation code".
3) - means "does not appear".
4) ** means "no change".
5) Δ denotes 0 id the operation code has been fetched from control store; Δ denotes 1 if the operation code has been fetched from the auxiliary address store.
6) $\overline{\text{STR}}$ means all codes except STR.
7) Before the CLEAR TOP operation the TRACE field from the top of the stack is saved in the SR register.

Table III

μinstr. address	Addressing Operation Code	Address in μins.	Top of the Stack before Operation CONTROL	TRACE	Test	Stack Oper. I.	Next Address Generated X	Stack Oper.II.	Remarks
			Δ[1]	UNCD β	*[2]	-	β+1	POP	unconditional
			Δ	CD β	0	POP	β+1	POP	
α	RETURN	-[3]	*	CD β	1	-	β[4]	-	conditional

1) Δ means 0 or 1; if Δ = 0 then X is prepared for the control store,
if Δ = 1 then X is prepared for the auxiliary address store.

2) *means "does not affect the operation code decoding".
3) - means "does not appear".
4) the address is for the control store

This information parameterizes the next sequencing operations performed on these addresses, and so the number of the addressing operation codes can be diminished. These codes, however enable truly modular structures in microprograms to be implemented. The use of the auxiliary address memory permits dynamic restructuring of microprograms built of modules stored in the control store by changing the contents of this auxiliary memory. Also, the hierarchy feature in modularity is possible with the use of this memory. The control store address generator described above, enriched with some additional functions such as microinstruction address modifications, can be used as a modular microprogramming sequencing unit for bit-slice microprocessor architecture.

BIBLIOGRAPHY

1. Husson, S.S., "Microprogramming: Principles and Practices", Prentice Hall, 1970.

2. Marczyński, R.W., Tudruj, M.S., "O pewnej koncepcji mikroprogramowanego układu sterujacego", Projektowaniw maszyn i systemów cyfrowych, Materiały z Sympozjum, PWN, s. 285-296, 1972.

3. Noguez, C.L.M., "A standardized Microprogram Sequencing Control with a Push Down Storage", 5-th Workshop on Microprogramming" , pp. 66-70, Illinois, Sept. 1972.

4. Marczyński, R.W., Tudruj, M.S. "Une approche de structuration des unités de commande microprogrammeés", Rapport IRIA No. 47, Janv. 1974.

5. Jones, L.H. "Instructions sequencing in microprogrammed computers", AFIPS 1975, Proceedings, pp. 91-98.

6. Marczyński R.W., Tudruj, M.S., "Microprogrammed control units - towards modularity in microprogramming", EUROMICRO Symposium on Micro Architecture Proceedings, pp. 173-182, Venice, 1976.

7. Aspinall, D., "Microprogrammable microprocessors", CERN School of Computing Papers, 1976.

8. M. Edwards, E. Dagless, "LSI microprogrammable microprocessors", Microprocessors, pp. 407-414, No. 7, 1977.

9. Am 2901, Am 2909 Technical Data, Advanced Micro Devices Inc. Publication.

10. BIT-SLICE, A SYBEX Seminar Book, SYBEX 1976.

MICROPROGRAM SYNTHESIS

G. DÁVID, S. KERESZTÉLY, I. LOSONCZI, A. SÁRKÖZY

MTA SZTAKI

1. INTRODUCTION

In this paper the logical base for program synthesis is discussed. In our previous papers [2], [3] a detailed definition of Structure Logic SL was given by which microcomputers and the user's problems can be described. Here we want to explain microprogram synthesis, which is a mechanized theorem proving technique, where the user's problem is to be proved on the base of the computer's description. The notations introduced in [3] will be used, not only in the case of SL but in the examples concerning to first-order logic as well, because SL contains first-order logic. To make our approach better understandable we start from program verification and the resolution principle in lower-order logics: 0-order (propositional) and first-order (predicate) logic, giving examples to illustrate mechanical theorem proving. In section 6 will be described how the system generates microprograms in SL.

2. PROGRAM ANALYSIS: VERIFICATION

An instruction is considered as a mapping of structures $i:s_1 \to s_2$. If there are two instructions executed in sequence this can be interpreted as the execution of one single instruction. If $inst_1:s_1 \to s_2$ and $inst_2:s_2 \to s_3$ are two instructions, then $inst = inst_2 \circ inst_1:s_1 \to s_3$ will be the instruction, which results the same as the sequence of the former ones. It is to be noted that $inst_2 \circ inst_1$ means $inst_1$ will be executed first.

For isntance if we take the instructions XCHi and LDRi of the microcomputer MCS-4:

MC<1:<2:R[i]>,2:<i:A[2]>,6:XCH(i,s)>
MC<1:<2:R[i]>,6:LDR(i,s)>

then

MC<1:<2:R[i]>,6:XCH(i,LDR(i,s))> shows the effect of the execution of LDRi and XCHi in this order which coincides with the effect of one single instruction LDRi, of course. In opposite order the effect can be described as follows:

MC<2:<i:A[2]>,6:LDR(i,XCH(i,s))>.

In a similar way a program may be understood as one instruction mapping the input data onto the output data. In fact a program is a sequence of transformations of the structure representing the machine. If we describe the structure of the machine at each step of a program from the beginning until the very end of it this will show us in which state the machine is after the instruction HALT. This way a program can be verified.

Let us take again the microcomputer MCS-4. We want to verify that the program

LDR1
CLC
ADD2
XCH3

will add the contents of the first and second registers and store it in the third one.
The input data puts the machine into the following state:

MC<2:<1:X,2:Y>, 6:s

First we executed the instruction *LDR1*, which results the next state of the machine:

MC<1:<2:X>,2:<1:X,2:Y>,6:LDR(1,s)>

We have to call attention upon the fact that on the 6th selector of the structure representing the machine appeared the instruction having been executed.

After the instruction *CLC:*

MC<1:<1:CY,2:X>,2:<1:X,2:Y>,6:CLC(LDR(1,s))>

On the 6th selector we can read from the right to the left the part of the program already executed.

The program follows with *ADD2:*

MC<1:X+Y,2:<1:X,2:Y>,6:ADD(2,CLC(LDR(1,s)))>

The last instruction is *XCH3:*

MC<1:<2:R[3]>,2:<1:X,2:Y,3:X+Y>,6:XCH(3,ADD(2,CLC(LDR(1,s))))>

And now our program can be found on the 6th selector.
It is to be read from the right to the left and it *has been proved* that in the output data the content of the third register is the sum of the first and second one.

3. RESOLUTION PRINCIPLE

What does the task of program generation mean? The input and the output data are known. The instruction is a variable, that is what we look for. So in order to generate a program all the possible ways of mapping the input data onto the output data must be found. From another point of view program synthesis is the proof of the final statement the output data.

This can be mechanized by means of *the resolution principle*.

In *propositional logic* the rule "modus ponens" is known, which is the pattern of logical reasoning. That is the following: There are permisses and a conclusion. The conclusion always must be true under the condition that the premisses are true. In the case of "modus ponens" the premisses have the form $P, P \rightarrow Q$ and the conclusion is Q. This is usually written as:

$$\frac{P \qquad P \rightarrow Q}{Q} \qquad \text{or} \qquad \frac{P \qquad \sim P \vee Q}{Q}$$

Clearly $P \rightarrow Q$ is equivalent with $\sim P \vee Q$. Extending the above rule we gain the *resolution principle of the propositional logic*, which falls within logical reasoning too:

$$\frac{L \vee C_1 \qquad \sim L \vee C_2}{C_1 \vee C_2}$$

that is if these permisses are true

then this conclusion is also true.

From now on it is supposed that our premisses are so called *clauses*, that is disjunctions of literals. So the set of permisses can be interpreted as an expression in conjunctive normal form: conjunction of expressions, which are disjunction of literals.

In this case $C_1 \vee C_2$ is called the *resolvent of the clauses* $C_1 \vee L$ and $C_2 \vee L$ and the following theorem holds.

Theorem: Given two caluses C_1 and C_2, a resolvent of C_1 and C_2 is a logical consequence of them.

Therefore a proof of S, and expression in conjunctive normal form is a deduction of the clause $\square$ which is always true.

Definition: If C is a clause, then the deduction of C from S is a finite sequence $C_1, C_2, \ldots, C_k$ of *clauses*, such that C_i is a resolvent of clauses preceding C_i and $C_k = C$.
This way theorem proving can be mechanized. So in propositional logic a program will be such:

input data $\rightarrow \ldots \rightarrow$ output data, where every state is the logical consequence of the one preceding it, i.e. the sequence of clauses describing the states of the microcomputer during the execution of a sequental program is a deduction of the output state, by a set of instruction-description clauses and by an input state used as the given set S of clauses.

In *first order logic* (predicate logic) there are variables, so we cannot always decide whether one expression is the negation of another or not.
For example $P(x)$ and $\sim P(y)$.

Therefore we introduce the notion of *substitution*.

Definition: A *substitution* Θ is a finite set of the form $\{t_1/v_1, \ldots, t_n/v_n\}$ where every v_i is a variable, every t_i is a term different from v_i and there are different variables after the stroke symbol. If E is an expression, then $E\Theta$ is another expression obtained from E by replacing simultaneously each occurence of the variable v_i, $1 \leq i \leq n$ in E by the term t_i. Composition of substitutions is defined as $\Theta = \sigma \cdot \lambda$ for substitutions Θ, σ, λ, if $E\Theta = (E\sigma)\lambda$.

Clearly $E \rightarrow E\theta$, that is every time when E is true $E\theta$ is true, too. (Each model of E is also a model of $E\theta$.)

Definition: A substitution θ is called a *unifier* for a set $\{E_1,\ldots,E_k\}$ if and only if $E_1\theta = \ldots = E_k\theta$. A unifier σ for a set $\{E_1,\ldots,E_k\}$ of expressions is a *most general unifier* if and only if for each inifier θ for the set there is a substitution λ such that $\theta = \sigma\cdot\lambda$.

Definition: If two or more literals of a clause C have a most general unifier σ, then $C\sigma$ is called a *factor of C*.

Definition: $(C_1\sigma - L_1\sigma) \vee (C_2\sigma - L_2\sigma)$ is a *binary resolvent* of C_1 and C_2 if $L_1\sigma = \sim L_2\sigma$.

A *resolvent* of C_1 and C_2 is a binary resolvent of C_1 or a factor of it and C_2 or a factor of it.

So the form of logical consequence in first order logic will be the following:

$$\left.\begin{array}{l} C_1 \vee L_1 \rightarrow C_1\sigma \vee L_1\sigma \\ C_2 \vee L_2 \rightarrow C_2\sigma \vee L_2\sigma \end{array}\right\} \quad C_1\sigma \vee C_2\sigma$$

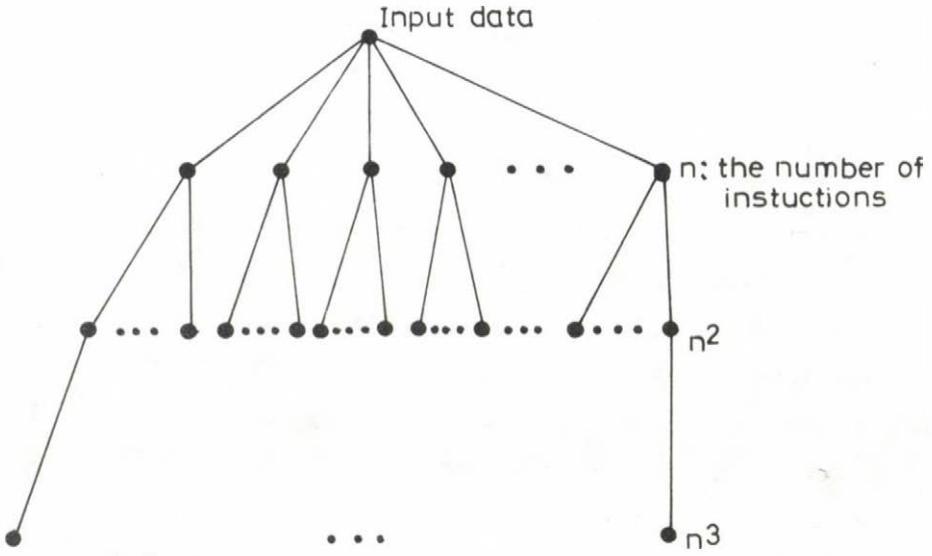
where $L_1\sigma = \sim L_2\sigma$

So the program in first order logic will have a similar form, as in the proportional one:

input data $\rightarrow\ldots\ldots\rightarrow$ output data

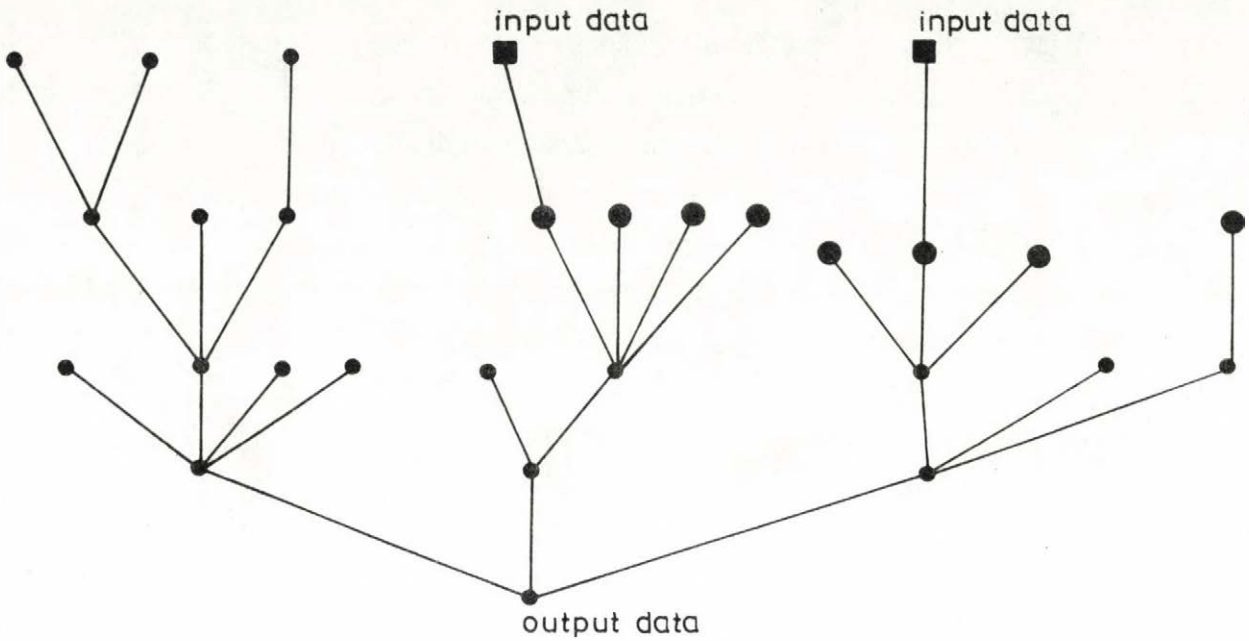
4. PROGRAM SYNTHESIS

The *method of program synthesis* is the following: Given a set of instructions of a certain machine written in the form of sentences PQ and the input and output data. Starting from the input data we make every possible resolution. After each resolution it is to be checked, whether we have got the requested result or not.But this way every resolution can be executed. That means, by the time the good program is ready, we have resolved loads of times redundantly. This can be shown on a tree.



Generating a program consisting of k instructions, $\sum_{i=1}^{k} n^i$ resolutions must be executed.

If we start from the output conditions, then those resolutions corresponding to instructions which are impossible cannot be carried out. The senseless instructions which does not change the state of the machine will not be carried out either. In this case our tree will look like this:



Each route will come to an end sooner or later, it is to be checked only whether some of these end points agree with the input data or put it properly the sentence at this point can be substituted with the input data. Even now in the case of long programs there will be plenty of different routes. How to decide which way to go to reach the input data as soon as possible this is the question of strategy. The problem of finding the best strategy or at least a suitable one is still open.

To make a resolution two expressions of the form $\sim PvQ(P \rightarrow Q)$ and $\sim Q$ are needed. One of them is the expression representing the instruction. The other - if going backwards - consists of the output data and a new predicate ANSWER having only one selector with a substructure of the type instruction, which is a variable at the beginning. After each substitution in the predicate ANSWER the part of the program already generated can be read from right to left.

Let us see an example:

Given a machine with a four-bit accumulator, 3 four-bit index registers and the instructions

LOAD1, ADD2, STORE3. M<O:A,1:R1;2:R2;3:R3,4:s>

The input data is:

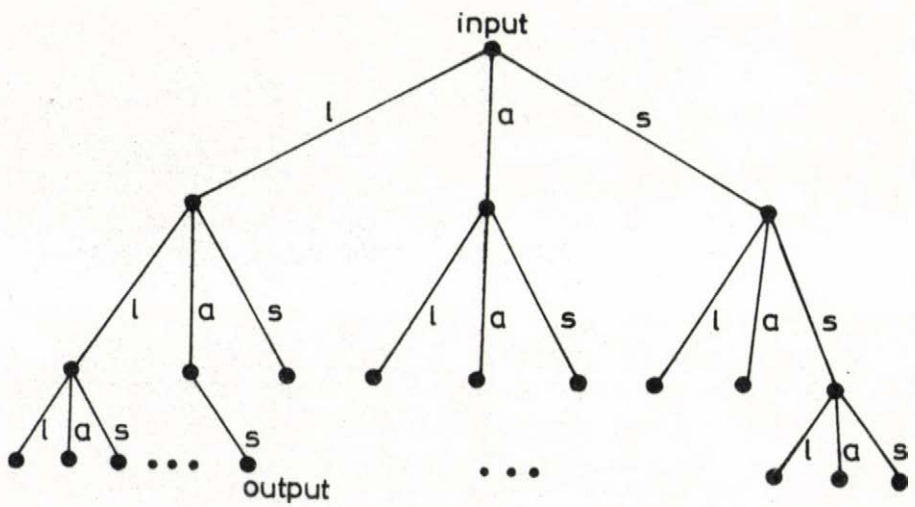
M<O:ACC,1:X,2:Y,3:Z,4:d>

The output condition is:

~M<3:X+Y>vANSWER<1:s>

We want to generate a program similar to the one verified in section 2.

First we generated the program fowards and there had to be made 27 resolutions, while the right program was found:

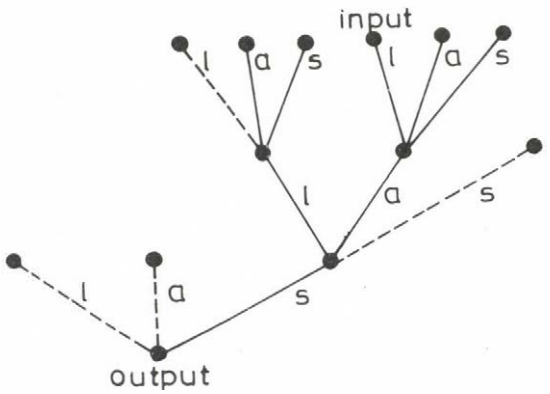


(ℓ:LOAD1,a:ADD2,s:STORES)

Working backwards there is no reason to resolve with the instructions not changing the substructure on the third selector.

~M<3:X+Y> v ANSWER<1:s>
~MvM<3:A,4:STORE3(s)>

I.~M<0:X+Y>vANSWER<1:STORE3(s)>



At the second step execution of the instruction STORE3 will make no change.
Choosing LOAD1.

I. ~M<0:X+Y>vANSWER<1:STORE3(s)>
~MvM<0:R1,4:LOAD1(s)>

II. ~M<1:X+Y>vANSWER<1:STORE3(LOAD1(s))>
To execute again LOAD1 will make no change.
So either
~MvM<3:A,4:STORE3(s)>

III. ~M<0:X+Y,1:X+Y>vANSWER<1:STORE3(LOAD1(STORE3(s)))>
or
~MvM<0:A+R2,4:ADD2(s)>

III. ~M<O:X,1:X+Y,2:Y>vANSWER<1:STORE3(LOAD1(ADD2(s)))>
is possible.
Choosing ADD2 as a second step.
~MvM<O:A+R2,4:ADD2(s)>

II. ~M<O:X,2:Y>vANSWER<1:STORE3(ADD2(s))>
Now the possibilities are:
a./ ~MvM<O:R1,4:LOAD1(s)>

~M<1:X,2:Y>vANSWER<1:STORE3(ADD2(LOAD1(s)))>
b./ ~MvM<O:A+R2,4:ADD2(s)>

~M<O:X-Y,2:Y>vANSWER<1:STORE3(ADD2(LOAD1(s)))>
c./ ~MvM<3:A,4:STORE(s)>

~M<O:X,2:Y,3:X>vANSWER<1:STORE3(ADD2(STORE3(s)))>

Observing the results of these resolutions there is only one sentence among them, which can be resolved with the input data:

~M<1:X,2:Y>vANSWER<1:STORE3(ADD2(LOAD1(s)))>
M<O:ACC,1:X,2:Y,3:Z,4:d>
ANSWER<1:STORE3(ADD2(LOAD1(d)))>

As it is seen only 8 resolutions were made to generate the program LOAD1
ADD2
STORE3.

5. MICROPROGRAM SYNTHESIS IN SL

The above description of the machine is good for our purpose, but there is a problem: Fancy a machine with (for instance) 16 working registers and with a memory consisting of only 256 registers, then certain instructions will be represented 16 and 256 times respectively. Indeed these instructions have the same form, so a common notation could emphasize their common characteristics. That is why Structure Logic SL was entered instead of first order logic. Structure Logic boils down not only notation, but meaning as well; by means of selector structures of the same type can be unified. See syntax of SL in [3].

Now it is to be discussed how to use resolution principle in Structure Logic. As there are variables in SL the same problem arises as in predicate logic how to decide whether an expression is the negation of another or not, so we need substitution again.

Definition: A substition δ is a finite set of the form $\{t_1/v_1,\ldots,t_n/v_n;\ inst/s;\ z_1/s_1,\ldots,z_k/s_k\}$ where every v_i is a variable, s an instruction variable, s_j a selector variable and every t_i is a term of the same type as v_i, inst is an instruction term, z_j is a selector term and there are different variables after the stroke symbol. If P is a formula $P\sigma$ is another formula obtained form P by replacing simultaneously each occurence of the variable v_i, $1 \leq i \leq n$ in P by the term t_i, s by inst and s_j by z_j. Each model of P will be a model of $P\sigma$ too, that is $Mod(P) \subseteq Mod(P\sigma)$, so $P \rightarrow P\sigma$, for it is easy to prove that for formulae F_1 and F_2 $F_1 \rightarrow F_2$ if and only if for their models $Mod(F_1) = MOD(F_2)$. The essence of substitution is that variables can be replaced by variables or terms. Clearly this allows identification of variables as well.

Having substitution the resolvent can be defined similarly as in first order logic:

P is a resolvent of clauses P_1', P_2' if $P_1' = P_1 v B_1$

$P_2' = P_2 v B_2$

and there are substitutions σ_1, σ_2 for which $\sim B_1\sigma_1 = B_2\sigma_2$
$P = P_1\sigma_1 v P_2\sigma_2$.

Here again the logical consequence holds.

$$P_1 v B_1 \rightarrow P_1\sigma_1 v B_1\sigma_1$$

$$P_2 v B_2 \rightarrow P_2\sigma_2 v B_2\sigma_2$$

$$\Rightarrow P_1\sigma_1 v P_2\sigma_2$$

Hence our program gained by resolution will have again the form:

input data →.....→ output data.

That means in Structure Logic too a program can be generated by proving the output data from the input data and certain instructions using the same mechanized theorem proving system as before.

6. PROGRAM WITH BRANCHES

This way long linear programs can be generated. But what if there are branches, that is conditional jumps in the program? In this case we have more than one sets of output data. To describe an instruction jump it is to be known where the certain instructions are in the program and what the condition of the jump is. For this purpose new substructures are entered into the description of the machine, which make at the same time using the predicate ANSWER unneccesary. These are the flags (F), a structure calles conditions (COND), where the conditions of the branches are registered, a program counter PC and a program

memory PM containing the instructions already generated. So before the beginning of program generation the program memory is empty. Naturally relative addresses are used for the isnstructions, as program generation starts from the end of the program and at that point it is not known how many instructions will the program consist of.

What now happens is: from the negation of the output data the negation of the input data is proved, which is equivalent with the proof of the output form the input data by certain instructions. The proof is constructive again, so at the end the program can be read in the program memory together with the relative addresses.

Let us see now what an instruction jump look like:

~P<PC:PC-C-1,M<PC-C:JCN(i,T(X),PC+1)>vP<F<i:T(X)>
COND<j:T(X)>>vP<F<i:C(X)>,COND<j:C(X)>,PC:PC-C>,

where $T(X_i)$ shows that the i-th flag is equal to 1 and $C(X_i)$ shows that the i-th flag is equal to 0.

How can two branches be put together by means of an instruction jump?

Let us suppese there are two branches represented by the following two clauses:

Cl1:~P<COND<K:T(sgn f)>,PC:PC-A,M<PC-A+1:U_1,...,PC:U_A
Cl2:~P<COND<K:C(sgn f)>,PC:PC-B,M<PC-B+1:v_1,...,PC:v_B

First Cl1 is resolved with the instruction jump. The following substitution is to be executed:

σ_1 = { 1/j,sgn f/X,PC-A/PC,M<PC-A+1:U_1,...,PC:U_A>/M}

~P<F<i:sgn f>,PC:PC-A-C-1,M<PC-A-C:JCN(i,T(sgnf),PC-A+1),
,PC-A+1:U_1,...,PC:U_A>>v

vP<F<i:C(sgn f)>,COND<K:C(sgn f)>,PC:PC-A-C,M<PC-A+1:U_1,...,PC:U_A>

Now Cℓ2 can be resolved by the result of the previous resolution with the instruciton jump.

$$\sigma_2 = \{ \text{PC-A}_{/PC},\ B_{/C},\ M<\text{PC-A-B+1:...,PC:...}>_{/M} \}$$

The result of this will give us the point of the program before the branch.

~P<F<i: sgn f>,PC:PC-A-B-1,M<PC-A-B:JCN(i,T(sgn f),PC-A+1),
PC-A-B+1:v_1,...,PC-A:v_B,PC-A+1:U_1,...,PC:U_A>>

It can be proved, that resolving in the opposite order - namely first Cℓ2 with the instruction jump and then with Cℓ1 would produce the same result.

In [3] the hardware of the machine MCS-4 is described, but is is to be completed with the new elements, which is the result of the new viewpoint. The new description is

predicate MC<1:A,2:R,3:M,4:SCR,5:P,6:F,7:C,8:PC,9:PM>
equivalance (A[1],F[1])
bit (8) PC
structure η <[0:255]:*bit*(8)>, ξ<[0:2]:*bit*(1)>
μ<[0:63]:*bit*(1)>
ref(η) PM; *ref* (ξ)F; *ref* (μ) C

And now some subtructions which could not have been described before.

MC<8:PC-1,9:<PC:JIN(i)>>→MC<8:RR[i]>

MC<8:PC-1,9:<PC:FIN(i)>>→MC<2:RR<i:PN[RR[0]]>>

MC<8:PC-1,9:<PC:JUN(K)>>→MC<8:K >

~MC <8:PC-C-1,9:<PC:JNC(i,T(X),C)>>v

vMC<6:<i:T(X)>,7:<j:T(X)>>vMC<6:<i:C(X)>,7:<j:C(X),8:PC-C>

~MC<8:PC-C-1,9:<PC-C:JNC(i,C(X),C >>v

vMC<6:<i:C(X)>,7: j:C(X)>>vMC<6:<i:T(X)>,7:j:T(X)>8:PC-C>

Here is an example what for instance the instruction exchange looks like in the new description.

~MC<8:PC-1,9:<PC:XCH(i)>>vMC<1:R[i],2:<i:A>>

Let us see an example for program generation with this new method. When generating a program automatically every possible resolution will be executed, but by means of strategy most of tehse cases can be eliminated. The reasoning given in the following example shows, how a strategy would work. For the sake of simplicity let us take a program similar to the one generated before. Given a machine with a four bit accumulator, 2 four bit index registers, a flag which is true if the content of the accumulator is zero and the instructions LOADi, ADDi, STOREi, JCN(1,C).

This is the machine:

M<1:A,2:R,3:F,4:C,5:PC,6:PM>

bit(4)A, *bit*(1)F, *bit*(8)PC,

α<[1:2]:*bit*(4)>,β<[0:63]:*bit*(8)>,γ<[1:32]:*bit*(1)>

ref(α)R, *ref*(β)PM, *ref*(γ)C,

We want to write a program which puts the sum of the two registers into the first one if this sum is zero and into the second one if the sum is not zero.

So the input data is:

M<2:<1:X,2:Y>>

The output data is:

M<2:<1:X+Y>,4:T(Z(X+Y))>v
vM<2:<2:X+Y>,4:C(Z(X+Y))>

As it was told above: from the negation of the output the negation of the input will be proved.
The end of one branch is

~M<2:<1:X+Y>,4:T(Z(X+Y))>

As the result depends upon the state of the flag, a "jump" must be in the program before the flag changes. Nevertheless there is no use of a "jump" at the very end of a program. There is only one instruction satisfying these requirements.

~M<5:PC-1,6:<PC:STORE(1)>> vM<2:<1:A>>

Cℓ1:~M<1:X+Y,4:T(Z(X+Y)),5:PC-1,6:<PC:STORE(1)>>

The end of the other branch is:

~M<2:<2:X+Y>,4:C(Z(X+Y))>

The above reasoning can be repeated:

~M<5:PC-1,6:<PC:STORE(2)>>vM<2:<2:A>>

Cℓ2: M<1:X+Y,4:C(Z(X+Y)),5:PC-1,6:<PC:STORE(2)>>

According to our train of ideas now the instruction "jump" is needed:

~M<5:PC-C-1,6:<PC-C:JCN($T(\alpha_i)$,C)>>vM<3:$T(\alpha_i)$,4:$T(\alpha_i)$>v
vM<3:$C(\alpha_i)$,4:$C(\alpha_i)$,5:PC-C>

Resolving with C&1:

~M<1:X+Y,3:Z(X+Y),5:PC-C-2,6:<PC-C-1:JCN(T(Z(X+Y)),C),
PC:STORE(1)>>v
vM<1:X+Y,3:C(Z(X+Y)),4:C(Z(X+Y)),5:PC-C-1,6:<PC-1:STORE(1)>>

and the resolvent with C&2:

~M<1:X+Y,3:Z(X+Y),5:PC-3,6:<PC-2:JCN(T(Z(X+Y)),C),
,PC-1:STORE(2),PC:STORE(1)>>

An instruction STORE would not do any good at this moment as we do not know the contents of the registers. An instruction LOAD would just annihilate what was done before. There is only one branch, that shows there is no need of jump.

So the next instruction is:

~M<5:PC-1,6:<PC:ADD(i)>>vM<1:A+R[i],3:Z(A+R[i])>

~M<1:X,2:<i:Y>,5:PC-4,6:<PC-3:ADD(i),PC-2:JCN(1,C),
PC-1:STORE(2),PC:STORE(1)>>.

Different resolutions could be made now, but looking at the input data we choose the following:

~M<5:PC-1,6:<PC:STORE(j)>vM<2:<j:A>>

where j is not equal with the i used above.

~M<1:X,2:<i:Y,j:X>,5:PC-5,6:<PC-4: LOAD(j),PC-3:ADD(i),
PC-2:JCN(1,C),PC-1:STORE(2),PC:STORE(1)>>.

Choosing j = 1 and i = 2 we got the negation of the input data, while in the program memory the generated program can be read with relative addresses.

REFERENCES

[1] Ching-Liang Chang, Richard Char-Tung Lee:
Symbolic Logic and Mechanical Theorem Proving
Academic Press, 1973. New York

[2] G. Dávid, S. Keresztély, A. Sárközy:
Program synthesis by theorem proving in Structure Logic SL

II. Hung. Comp. Sci. Conference
1977., part/1. pp. 291-310

[3] G. Dávid, S. Keresztély, I. Losonczi, A. Sárközy:
Logic-based description of microcomputers (in this issue)

СИМУЛЯЦИЯ ПРИБОРО-ТЕХНИЧЕСКИХ ФУНКЦИЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ НА БАЗЕ МИКРОПРОЦЕССОРА В РЕАЛЬНОМ МАСШТАБЕ ВРЕМЕНИ

B. Hackler
Technische Universität Dresden
Sektion Informationsverarbeitung

1. Введение

Для подготовки введения микро-ЭВМ на базе микропроцессора ИНТЕЛ 8080 в Техническом Университете Дрездена разрабатывалась система программирования, которая предназначена для вычислительных машин типа ряд. Она состоит из макроассемблера и симулятора. Так как программы микро-ЭВМ обычно записываются в постоянной памяти, прежде всего надо их тщательно проверить. Поэтому была разработана система симуляции для отладки программ микро-ЭВМ, которая может работать вместе с ассемблером, а также отдельно.

2. Основания для симуляции вычислительных машин

Чтобы исполнить симуляцию, надо моделировать реальный объект. Симуляцией называются эксперименты с моделью для исследования свойства и поведения реального объекта. Симуляция - это изображение опередеденных интересующих качеств, признаки и поведения одной системы с помощью другой системы. Если идет речь о дискретной симуляции, тогда модель трансформирует-

ся в систему программ, которая обрабатывается на цифровой вычислительной машине. При дискретной симуляции модель рассматривается в дискретных точках времени $\tau_н$(н=0,..., м), $\tau_н \in [\tau_0, \tau_м]$. Разница $a = \tau_{н+1} - \tau_н$ называется величиной шага. Дискретная симуляция - это шаговый алгоритмический процесс. Он начинается процессом инициализации,при котором входной переменной и каждому параметру состояния присваивается начальное значение. Циклические шаги симуляции состоят из отображения входных переменных в выходные переменные следующего состояния, причем между двумя состояниями может наступить взаимодействие между системой и окружающей средой. Цели симуляции вычислительных машин и протекающих в них процессов состоят в исследовании с помощью теории массового обслуживания и в моделировании аппаратных функций вычислительных систем для отладки программ.

С развитием микро-ЭВМ значение моделирования аппаратных функций ЭВМ увеличивается. В целях управления микро-ЭВМ обычно вставляются в приборы или установки. Поэтому у них вообще нет внешних периферийных устройств, необходимых для составления программ. Кроме того, программы хранятся в постоянной памяти. Тщательный тест логической правильности такой программы очень важный, потому что последующие изменения в программе являются всегда дорогими. Для разработки программ микро-ЭВМ используется благоприятно, так называемая, система отладки. Это - вычислительная система с необходимыми периферийными устройствами, у которой функция центрального устройства выполняется микропроцессором того же самого типа.

Если такая система не имееется в распоряжении, тогда приходится разрабатывать программы на ЭВМ другого типа. В этом случае надо использовать для разработки программы **Cross-Software**, которые обрабатываются на вычислительной машине другого типа. Система симуляции основана на модели реальной вычислительной системы, в которой моделируются важные аппаратные функции для выполнения программы.

3. Моделирование аппаратных функций вычислительной системы

К конфигурации вычислительной системы принадлежат центральное устройство, единицы основной памяти, периферийные устройства, устройство перерыва и шина, которая содержит канал управления, канал передачи данных и адресную шину. Типичной для микро-ЭВМ является система стандартных блоков. Такая конфигурация вычислительной системы структурирована нагладно. Для моделирования аппаратных функций вычислительной системы благоприятно использовать как исходную точку для структуры модели структуру реального объекта. Такие качества системы, которые в модели должны содержаться, надо распределять на элементы модели.

Систему устройства реальной вычислительной машины можно понимать как множество операторов. Оператор действует на операндах. Все операнды, которые поступают в вычислительную систему, являются состояниями. Данные соответствуют состоянию регистров, счетчиков или ячеек памяти. Состояния являются, например, тоже в положении триггеров. Каждый элемент модели соответствует оператору или группе операторов с мало измененным или постоянным операционным положением. Необходимое для модели подмножество операндов реальной системы изображается как множество входных и выходных данных и у элементов системы модели. Функция поведения f /и, у/ элементов модели реализует моделируемое операционное положение реальной системы устройства.

Тоже благоприятно определять структуру с точки зрения позднейшего превращения модели в программную систему. Должна быть обеспечена возможность, чтобы каждый элемент модели превращать в программный модуль, в подпрограмму или в программную секцию дискретной системы симуляции.

Поэтому уже при выборе структуры модели надо принимать во внимание: какие возможности дает выбор ЭВМ, на которой

симулятор обрабатывается, выбор языка программирования и какие ограничения необходимо накладывать. Прежде всего, надо принимать во внимание емкость основной памяти, скорость выполнения операций ЭВМ и помощь выбранного языка программирования. С этой точки зрения происходило соединение единиц основной памяти и периферийных устройств для каждого одного элемента модели. Шина микро-ЭВМ разделяется на канал управления, канал передачи данных и адресную шину. Эти другие каналы моделируются различно. Функция канала управления реализуется с помощью отдельного элемента модели. Его главная задача состоит в использовании управляющих сигналов, которые посылаются от других элементов модели, и в передаче управления следующему элементу.

Так как модель надо перевести в программную систему, при поделировании отказались от функции канала передачи данных. Данные хранятся в общих доступных областях. Функции поведения элементов гарантируют, что только эти данные будут востребованы, которые и оператор в реальном объекте востребовал бы. Подобно решаются и задачи адресной шины. Данные и адреса являются входными и выходными величинами почти всех элементов системы модели. Соотношения между элементами модели определяется с помощью функции поведения γ /и, у, х, е, ф/ модели с

и пространство входного состояния,
у пространство выходного состояния,
х пространство состояния системы,
/е, ф/ описание поведения системы,
е множество последовательных соотношений,
ф множество соотношений результатов.

Существенным для вида соотношений между элементами является расположение элементов в иерархических уровнях. Существуют связи только между элементами отдельных уровней.

4. Моделирование среды вычислительной системы

С помощью уже описанного симулятора можно обрабатывать программы, которые составляют вместе с аппаратной системой ЭВМ относительно замкнутую систему.

Влияния процесса в ЭВМ на среду или влияния процесса в среде на процесс в ЭВМ не могут симулироваться таким симулятором. На основе свойств микропроцессоров определяются главные случаи применения микро-ЭВМ в области управления процессов. На микро-ЭВМ, которая используется вычислительной машиной управления, существует процесс, который общается с управляемым основным процессом с помощью входных и выходных информаций.

Поэтому существует модель симулятора, которым симулируется взаимодействие между управляющим процессом и основным процессом. Для этого необходимо моделировать части среды и сообщающие процессы среды. Моделирование двух или многих параллельных процессов в одной программной системе, кооторая обрабатывается на цифровой вычислительной машине удобное последовательное изображение. Как общий эффект должна быть обеспечена видимость параллельной работы процессов. Непрерывное течение времени необходимо заменить конечной последовательностью дискретных точек времени $\tau_н$ с $н = 1, 2, \ldots, к$. В интервале $(\tau_{н+1} - \tau_н)$ процессы являются неизменными и независимыми друг от друга. Состояния процессов актуализируются по истечению каждого интервала времени.

Так как эти моделируемые процессы производят изменения состояния сообщающих процессов только в дискретные точки времени, применяется метод точки времени события. То есть, что точки времени, где производятся изменения состояния, являются последовадельностью дискретных точек, в которых можно актуализировать состояния процесса.

4.1 Моделирование системы

Пересечениями между ЭВМ и средой установились периферийные регистры и вход в модуль перерыва. Периферийные регистры - это такие регистры, которые существуют в периферийных устройствах и хранят входные информации в буферах.

К близкой среде ЭВМ принадлежат периферийные устройства, которые связывают ЭВМ с основной системой. Задача периферийных устройств состоит в преобразовании представления информаций или связи многих информаций с одной или многими новыми информациями. Моделирование близкой среды ЭВМ состоит поэтому в моделировании периферийных устройств, которые являются одновременно элементами системы. Элементы построены в симуляторе в одинаковой форме и получают потребителем свои специфические качества с помощью ввода параметров. Такие свойства:

- адреса обложенных периферийных регистров и каналов перерыва;
- поведение вычислительного устройства во время преобразования информаций периферийным устройством;
- поведение периферийного устройства во время и после окончания работы.

Специальное периферийное устройство является в этом смысле тоже отдельным каналом перерыва. Его действие состоит в извещение о перерыве. Начало и конец действия совпадают.

4.2 Моделирование основных процессов

Процессы, которые протекают в среде ЭВМ и управляются вычислительной машиной, называются основными процессами. Процессы,которые протекают в близкой к ЭВМ среде,называются периферийными процессами. Моделируемый периферийный процесс протекает на элементе моделируемой системы среды, на моделируемом периферийном устройстве. Влияния связанных с ним основных процессов могут состоять в подготовке информаций к вводу и в виде запаздывания во времени. В общем процессы в среде ЭВМ находятся под влиянием случайных функций. В ряде случаев, однако,

достаточно моделировать такие процессы детерминированными процессами. В подобном случае интервал времени работы периферийных устройств, которые работают в большинстве случаев циклически, остается постоянным для всех преобразований информации.

Если для эксперимента симуляции существенно, что интервал времени работы прерываний в определенных границах, тогда можно процесс моделировать как стационарно стохастический процесс. При помощи генератора случайных чисел время подготовки информаций становится переменным.

Для моделирования процессов в среде ЭВМ нужно внести в систему потребителем симулятора следующие параметры:

- подготовка данных ввода;
- продолжительность времени преобразования информации;
- интервал времени, на протяжении которого продолжительность времени преобразований информации должна быть изменена.

4.3 Моделирование взаимодействий между основным процессом и управляющим устройством

Взаимодействия между двумя процессами P1 и P2 существуют, когда состояния процесса P2 зависимы от особых прежних состояний процесса P1 и потом особые состояния процесса P2 причиняют опять изменения состояний процесса P1. Между управляющим и основным процессом вычислительной системы существуют взаимодействия, когда точка времени подготовки данных в периферийных регистрах или точках времени требований перерыва зависима от состояния других периферийных регистров в прежнюю точку времени или от прежних требований перерыва.

Моделируемый основной процесс состоит из ряда частей процесса, которые принадлежат моделируемым периферийным устройством.Они являются в общем асинхронными друг к другу. В моделе изображаются асинхронные изменения состояния этих параллельных процессов на одну единственную временную последова-

тельностью. При выявлении определенных изменений состояния процесса определенные обратные связи начинают действовать на управляющий процесс. Влияние состояний управляющего процесса на периферийные процессы нельзя реализовать с помощью таблицы времени потому, что моделируемый управляющий процесс получает управление моделью в каждом интервале времени один раз и проводит изменение состояния управляемого процесса.

Описание симулируемых взаимодействий между управляющим и основным процессом надо передать симулятору от пользователя. Четкое описание дается с помощью сообщения множества преобразований информаций, которые надо проводить до того, как множество периферийных регистров и каналов перерыва годны к употреблению.

5. Возможности применения системы симуляции

При построении симулятора придавалось большое значение возможности сопряжения системы с проблемами пользователя. Кроме помощи в отладке, например, протоколирование машинных операций или групп машинных операций и вывода любых областей памяти, можно определить точки остановки в программе и вести протокол о счетчике времени. Можно давать любое разделение на память с прямым доступом и на постоянную память, которое принимается во внимание при выполнениях программы. Периферийными устройствами микро-ЭВМ могут являться любые периферийные устройства ЭВМ, на которой симулятор обрабатывается. Посредством возможности моделирования пользователем среды ЭВМ и процессов в ней в реальном масштабе времени даны дальнейшие возможности применения симулятора.

Session 4: Applications

Сессия 4: Применения

К ВОПРОСУ ОПТИМИЗАЦИИ РАЗДЕЛЕННЫХ ПО ЗАДАЧАМ МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Р. Шульце

НП Роботрон, НИЦ г. Дрезден

Развитие современной вычислительной техники в возрастающей мере характеризуется предоставлением интегральных схем высокой степени интеграции. В этом смысле происходит приспособление стандартных средств вычислительных систем к требованиям пользователей, например, ЭВМ техники связи, научно-технические специальные ЭВМ и т.д. Имея ввиду, что пользователь за последние годы на основании компактности предлагаемых ЭВМ был принужден пойти на уступки относительно конфигурации, он теперь имеет существенную возможность целеустремленно влиять на структуру системы и видоизменить ее в соответствии со своими специфическими желаниями. Это также касается конфигурации памяти, каналов ввода/вывода, специальных процессов и устройств связи.

Но эта тенденция требует высокой гибкости при разработке ЭВМ. Этим неизбежно надо применять новые способы разработки и использовать методы или результаты смежных специальностей. Сюда относятся, например, математика /особенно дискретная математика/, алгебра, а также физика.

2. Классификация структур вычислительных машин

Грубая классификация вычислительных структур отличает последовательные от параллельных структур многопроцессорных систем. Подразделение параллельных структур в разделение по задачам и функционально разделенные структуры выражают, что параллелизация относится или к потоку управления, или к потоку

данных. Для полноты надо еще привести подобные структуры аналоговых ЭВМ, как типичные примеры традиционных параллельных ЭВМ.

Внешним признаком параллельно работающих ЭВМ является их децентрализованный интеллект. Функционирование таких ЭВМ достаточно точно описывается для специальных процессов реализуемым алгоритмом или реализуемой программой.

Многообразие многопроцессорной системы обусловлено оформлением систем шин, организацией обмена данными, изменчивостью длины слова с помощью **slice** - процессоров и т.д. В литературе можно наблюдать крайние тенденции, например, в системах **Hypercube** II и III, которые представляют эффективность многопроцессорных систем в линейной зависимости от степени децентрализации или, иначе говоря, от количества установленных в системе процессоров. Несмотря на то, что микропроцессоры сегодня уже нашли применение во многих областях науки, техники и сбыта, кажется, что об их взаимодействии в многопроцессорных системах еще нет законченной теории. На основании уже названного многобразия конечно возникает вопрос, может ли вообще существовать такая теория?

В настоящее время превалирует мнение, что с растущим количеством совместно включенных микропроцессоров время выполнения программ принципиально сокращается. Основываясь на подобных проблемах, данным докладом хочется доказать, что в многопроцессорных системах для выполняемой программы или алгоритма существует оптимальное количество установленных мультипроцессоров. Отклонения от этого оптимального количества связаны с увеличением времени выполнения. Для параллелизированных последовательных алгоритмов оптимальное количество определяется характеристикой алгоритма, а для параллелизированных последовательных программ - степенью параллелизации и конечностью определенных системных параметров, как поясняется далее. Данный подход пытается доказать на примерах, что для определнной выполняемой программы или алгоритма существует некоторое определенное количество процессоров, которое является оптимальным.

1. - Наблюдается, что между количеством микропроцессоров в многопроцессорной системе и выполняемым ими параллелизированным алгоритмом стоит оптимальная взаимосвязь.

2. - Имеются формальные высказывания о том, что параллелизированием последовательных алгоритмов можно увеличивать количество вычислительных операций, в связи с чем возникает возможность одновременно выполнять большее количество операций и, тем самым, достигается сокращение общего времени обработки.

3. - В настоящее время еще невозможно доказать эти положения общедействительными и поэтому они демонстрируются на примере быстрого преобразования Фурье /БПФ/ /1, 3/.

Для быстрого преобразования Фурье действительно следующее изображение.

$$X(j) = \sum_{k=0}^{N-1} A(k)\exp\left(\frac{2\Pi i}{N} jk\right), \qquad /1/$$

$$i = \sqrt{-1}, \quad j = 0,1,\ldots, N-1$$

В нем $X(j)$ означает оригинал и $A(k)$ означает изображение.

Пусть N является

$$N = \prod_{i=1}^{m} r_i = r^m \quad \forall r_i = r \qquad /2/$$

количеством опор, полученных произведением из m, в данном случае идентичных множителей $r_i = r$. Аргументы j и k удовлетворяют описывающим уравнениям 2.1 и 2.2.

$$j = \sum_{\mu=0}^{m(r)-1} j_\mu \rho_\mu(r), \quad k = \sum_{\nu=0}^{m(r)-1} k_\nu \rho_\nu(r) \qquad /2.1/2/$$

где ρ нулевое при r равно 1 или нуль, меньше или равно j_μ максимум равное $r = -1$ соответственно это и для k_ν.

Для инженерных применений $\rho_{\mu\nu}(r)$ получается в общем как экспоненциальная функция.

$$\rho_{\ell}(r) = r^{\ell}, \quad \ell = \mu,\nu \qquad /2.3/$$

Под этими условиями быстрое преобразование Фурье можно изображать как многомерное функциональное преобразование.

$$X(j_o,\ldots,j_{m-1}) = \sum_{(k_o)} \ldots \sum_{(k_{m-1})} A(k_o,\ldots,k_{m-1}) \prod_{\nu=o}^{m(r)-1} \exp\left(\frac{2\pi i}{N}\, jk_\nu r^\nu\right) \qquad /3/$$

Количество T выполняемых операций для расчета оригинала X приведен в уравнении 3.1.

$$T = \lambda mr = \lambda\, \ell nN \frac{r}{\ell nr} \qquad /3.1/$$

Количество операций является интересным для последовательных режимов обработки потому, что оно должно быть минимальным. Очевидно **N** и **r** определяют значение T. Так как **N** определен внешними условиями, минимизацию T можно осуществить только с помощью базиса **r**.

$$\frac{\partial}{\partial r} T(N,r) = \lambda\, \ell nN \frac{\ell nr - 1}{(\ell nr)^2} = 0 \Bigg|_{r=\ell} \qquad /3.2/$$

Так как **r** может быть только целое положительное число, пусть принимается **r = 2**. Это является и основой последовательных цифровых вычислительных систем. И так оказывается, что быстрое преобразование Фурье предназначено для решения на цифровых ЭВМ.

Теперь рассмотрим возможность параллелизации быстрого преобразования Фурье. Сначала производится превращение быстрого преобразования Фурье, изображенного как многомерное функциональное преобразование. Превращение здесь касается произведения экспоненциальных функций.

$$\exp\left(\frac{2\pi i}{N}\, j\, k_\nu\, r^\nu\right)=\exp\left(\frac{2\pi i}{N}\, k_\nu \sum_{\mu=o}^{m-\nu-1} j_\nu r^{\mu+\nu}\right) \qquad /3.3/$$

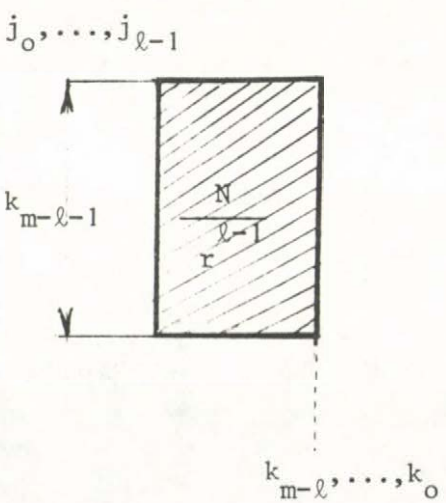
$$\exp\left(\frac{2\pi i}{N}\, k_\nu \sum_{\mu=m-\nu}^{m-1} j_\mu r^{\mu+\nu}\right) = 1 \;\forall\; j_\mu, \quad (r^{\mu+\nu}/N) \in R^+$$

С помощью этого превращения изображение переводится последовательной заменой на m уровнях в оригинал.

$$^{\ell+1}A(j_o,\ldots,j_\ell,k_{m-\ell-2},\ldots,k_o) = \qquad /4/$$

$$= \sum_{k_{m-\ell-1}=o} {}^{\ell}A(j_o,\ldots,j_{\ell-1},k_{m-\ell-1},\ldots,k_o)\exp\left(\frac{2\,i}{N}\, k_{m-\ell-1} r^{m-\ell-1} \sum_{\mu=o}^{\ell} j_\mu r^\mu\right)$$

Этот перевод можно наглядно объяснить с помощью рисунка 1.



остаток для решения $\rho_\ell(r,N)$

$$K(r,N) = \sum_{\ell=1}^{m} \rho_\ell(r,N) = (N-1)\,\frac{r}{r-1}$$

Рис. 1.: Изображение остатка для решения при последовательном переводе.

После перевода изображения с уровня ℓ на уровень $\ell+1$, остается некоторый остаток для решения $\rho(r,N)$.

Параллелизация быстрого преобразования Фурье состоит в предоставлении слагаемых из уравнения /4/. С параллелизацией связаны два требования.

1. Количество $T/N, r/$ выполняемых операций должно быть малым.
2. Перевод изображения в оригинал должен осуществляться, по возможности, только на немногих уровнениях.

Поскольку N, как принималось, определяется внешними условиями, базис чисел быстрого преобразования Фурье должен выполнить требования вариации r. Оба требования находятся в противоречии друг к другу, как видно из рисунка 2.

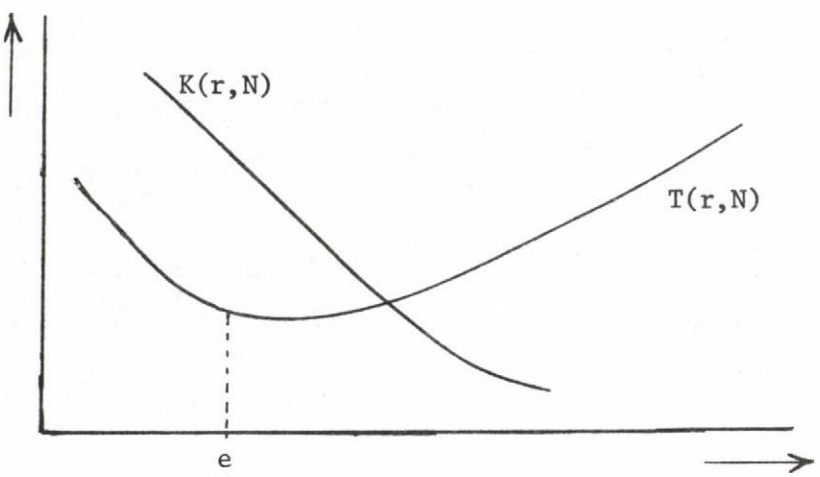


Рис. 2.: Изображение функции $T(r,N)$ и $K(r,N)$.

Определение оптимального r производится через минимум произведения количества выполняемых операций и количества уровней решения относительно r.

$$F(r,N) = T(r,N)K(r,N) = N_o \frac{r^2}{(r-1)\ell_n r}, \qquad N_o = \lambda N \ell_n N$$

Частное производное $F(r,N)$ на r достигает минимум при $r = 4,4$.

$$\frac{\partial}{\partial r} F(r,N) = N_o \frac{r((r-2)\ell_n r - (r-1))}{(r-1)^2(\ell_n r)^2} = 0 \Bigg|_{r \approx 4,4}$$

Эта картина представлена на рис 3.

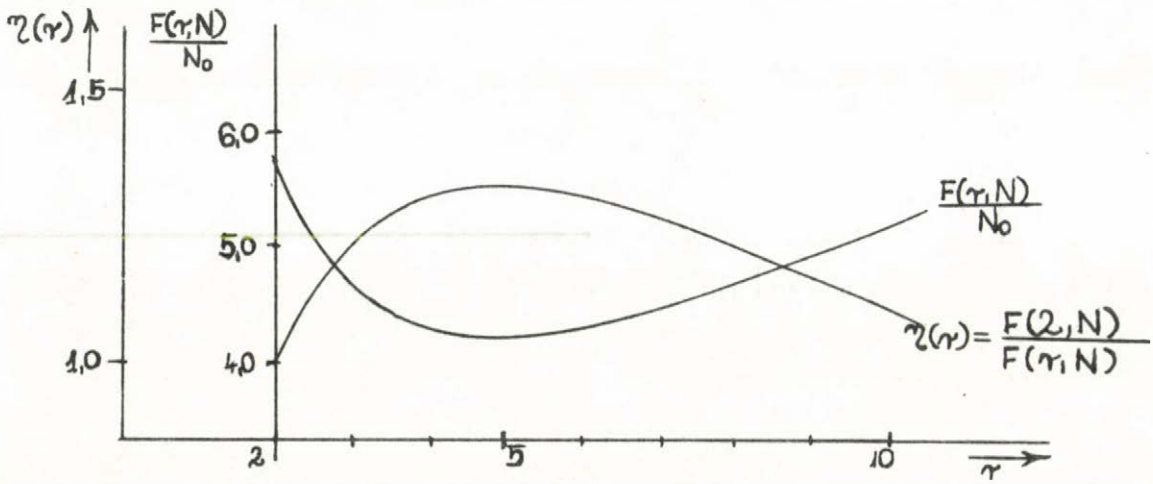


Рис. 3.: Изображение функции $F(r,N)/N_o$ и функции качества $\eta(r)$

Функция качества указывает, в какой мере с увеличением базиса осуществляется сокращение времени выполнения.

Рис. 3 еще раз наглядно показывает, что параллелизация последовательного алгоритма - в данном случае алгоритма быстрого преобразования Фурье - повышает количество вычислительных операций, а также открывает возможность одновременно выполнять большое количество операций. Таким образом, в конечном итоге еще получается увеличение пропускной способности.

Пусть для реализации параллелизированного быстрого преобразования Фурье служит мультипроцессорная система типа "Master-Slave". Выбором $r = 4$ определяется количество "Slave" - процессоров, равное 4.

Вычисление слагаемых происходит параллельно автономными процессорами P_0 по P_3 . В начале вычисления "Master" - процессор P передает всем процессорам через общую шину B изображение уровня представляющееся частично модифицированным изображением, которое должно быть последовательно переведено в оригинал X. Каждому "Slave" - процессору следует индекс = 0,1,2,3. Каждый "Slave" - процессор производит частичную актуализацию переданного изображения.

После вычисления слагаемых "Slave" - процессорами "Master" - процессор осуществляет их сложение и заменяет в сумме позицию через j_ℓ . Таким образом получается изображение уровня $\ell+1$, которое по приведенному алгоритму последовательно переводится в оригинал X. Перевод закончен, когда последний аргумент актуализирован. После этого оригинал записывается в главную память. Новая пара значений вводится и в "Master" - процессор.

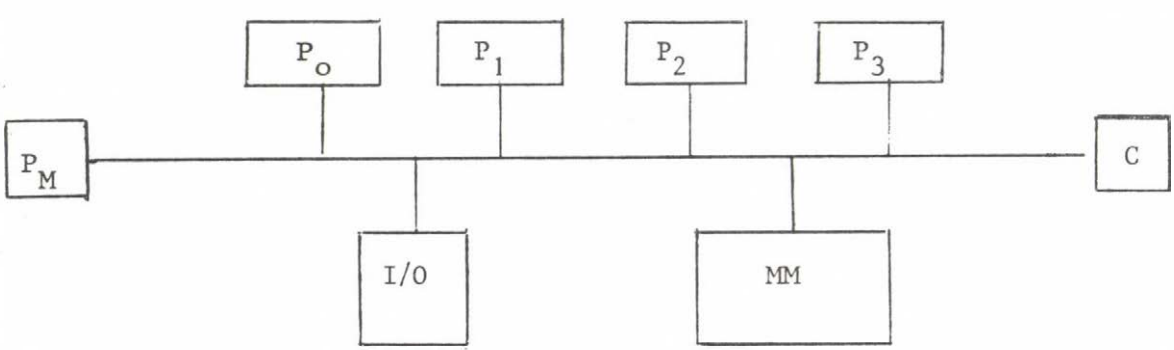


Рис. 4.: Многопроцессорная система для вычисления параллелизированного Быстрого преобразования Фурье.

В следующем обсуждается вопрос выбора оптимального количества процессоров в мультипроцессорной системе для обработки параллелизированных последовательных программ.

Пусть имеется последовательная программа R, разлагаемая в произвольное множество программных отрезков.

$$R = \{ {}^{1}P, \ldots, {}^{j}P, \ldots, {}^{\rho}P \}, \qquad 1 \leq j \leq \rho$$

Отрезок P программы состоит из множества операторов.

$${}^{j}P = \{ {}^{i}f \} = \{ {}^{j}f_1, \ldots, {}^{j}f_i, \ldots, {}^{j}f_{\kappa_j} \}, \qquad 1 \leq i \leq \kappa_j$$

Под оператором здесь понимается сопоставитель в форме арифметической команды.

$${}^{j}f_i = \{ {}^{j}\mathcal{A}_i, {}^{j}\mathcal{Z}_{\mu \neq i}, {}^{\ell \neq j}\mathcal{R} \}$$

Оператор выполянется, применяя множество ${}^{j}\mathcal{A}_i$ первичных входных переменных, множество ${}^{j}\mathcal{Z}_{\mu \neq i}$ промежуточных переменных /как элементы множества результатов соседних операторов внутри ${}^{j}P$ / и множеств ${}^{\ell \neq j}\mathcal{R}$ результатов соседних отрезков программ.

Побочными условиями являются:

1. Как различные операторы внутри отрезка программ ${}^{j}P$, так и соседние отрезки программы ${}^{\mu}P$ внутри P, имеют общие входные переменные.

$$\bigcap_{i=1}^{\kappa_j} {}^{j}\mathcal{A}_i \neq 0 \qquad \bigcap_{j=1}^{\rho} \left(\bigcup_{i=1}^{\kappa_j} \mathcal{A}_i \right) \neq \emptyset$$

2. Результаты соседних отрезков программы ${}^{j}P$ могут быть входными переменными отрезка ${}^{\ell \neq j}P$ программы .

$$\left(\bigcup_{j=1}^{\rho} \left(\bigcup_{i=1}^{\kappa_j} {}^{j}\mathcal{A}_i \right) \right) \cap \bigcup_{\substack{\ell=1 \\ \ell \neq j}}^{\rho} {}^{\ell}\mathcal{R} \neq \emptyset$$

Приняв во внимание заранее оговоренные условия, произвольно сепарируемая программа точно тогда параллелизируется, если выполняются следующие условия:

1. Промежуточные переменные действительны только для прикрепленного к ним отрезка программы, т.е. имеется автономия.

$$\bigcap_{j=1}^{\rho}\left(\bigcup_{i=1}^{\kappa_j} \mathcal{L}_{j_i}\right) \neq \emptyset$$

2. Промежуточные переменные отрезка программы j не тождественны с множеством переменных программы R.

$$\left(\bigcup_{i=1}^{\kappa_j} \mathcal{L}_{j_i}\right) \cap \left(\bigcup_{j=1}^{\rho}\left(\bigcup_{i=1}^{\kappa_j} \mathcal{X}_{j_i}\right)\right) = \emptyset$$

3. Результаты отрезков программ действительны для своих отрезков программ и не являются тождественными. Ramamorthy

$$\bigcap_{j=1}^{\rho} \mathcal{R}_j = \emptyset$$

На этих предпосылках основывается тоже алгоритм параллелизации.

На симпозиуме "Euromicro" - 1976 г. Kober /Siemens - A6/ в интересном докладе /2/ представил относительную оценку времени последовательной и параллельной обработки программы. На основе этой оценки в следующем хочется показать, что для МПС существует оптимальное, т.е. не эффективно расширяемое количество процессоров для обработки параллелизированных последовательных программ.

Пусть, следуя Коберу, время последовательной обработки программы $T_{seq.}$ задается как произведение количества час -

тичных программ ρ на среднее время обработки T_{av} для одной частичной программы,

и время параллельной обработки программы $T_{par.}$ в приведенной форме:

$$T_{par.} = T_{\rho} + \mu T_{CM} + \frac{\rho}{\mu\alpha} T_{av}$$

где μ = количество процессоров в МПС

α = средняя степень загрузки одного процессора в автономной фазе

T_{ρ} = длительность фазы управления /представление управляющих информаций/

T_{CM} = время обмена данными для одного процессора.

В выражении для $T_{par.}$ характерно, что растущее число процессоров потенциально влияет на уменьшение времени обработки программ, но в то же время и повышается длина фазы обмена данными системы.

Для наглядного пояснения отношений пусть принимается МПС, у которой

$\mu = 3$ процессора.

В этой системе должна обрабатываться программа состоящая из

$\rho = 9$ отрезок программ.

Один отрезок программы в среднем состоит из 35 алгебраических операций и 3 стандартных функций. Длина слова - 24 бита. Операции осуществляются в арифметике с плавающей запятой. Продолжительность фаз управления обмена данными предполагается в 10 мсек. Степень загрузки одного процессора - 80%, $T_{av} = 250$ мс.

Отношение R между $T_{par.}$ и $T_{seq.}$ выражает уменьшение времени обработки программ при параллелизации последовательной программы.

На основании преобладания $\frac{k}{\rho a} \approx 0{,}42$ над $\frac{T_\rho}{\rho\, T_{av}} \approx 4{,}5 \cdot 10^{-3}$ и $\frac{T_{CM}}{k\, T_{av}} \approx 1{,}4 \cdot 10^{-2}$, $k = \rho/\mu$ для R действительно следующее приближение:

$$R \approx \frac{1}{\mu \cdot \alpha(\mu)} \quad , \quad \mu = \frac{\rho}{k} \; .$$

В этом приближении для R далее учитывается дополнительная зависимость степени загрузки α от количества процессоров. Естественно ожидается, что при увеличении количества процессоров на основании ситуации конфликтов или состояний ожидания в системе убывает степень загрузки α . Но это приведет к увеличению от R , что качественно представлено на рис. 5.

Названные состояния ожидания могут быть обусловлены системными или программными причинами, вызванными, например, неподходящей параллелизацией программы. Без ограничения общедействительности в следующим предполагается, что программа R была оптимально разделена на программные отрезки. Отклонения от этого подтверждают следующие высказывания.

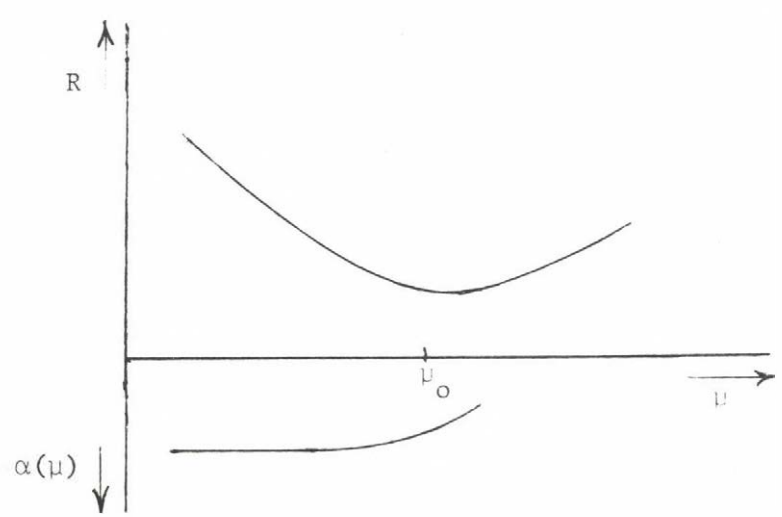


Рис. 5.: Зависимость производительности процессоров от общего количества процессоров в системе.

С предполагаемой степенью згрузки α уже для $\mu = 2$ получается значительное уменьшение времени обработки программ. Эта черта затеряется при дальнейшем увеличении μ . Для μ меньше μ_o - где μ_o является критическим количеством процессоров - степень загрузки существует независимо от α. Зависимость для μ больше μ становится наличным за счет α .

Обобщая, ход $\alpha(\mu)$ имеет следующие причины:

- образование очереди во время обслуживания ресурсов;
- конечная пропускная способность каналов;
- возможное появление аварий в управлении, приводящих к блокировке всей системы;
- гетерогенное распределение приоритетов.

Заключение

Цены на микропроцессоры на мировом рынке снижаются. В данном докладе хотелось показать, что несмотря на это нет смысла превышать критическое количество установленных микропроцессоров в мультипроцессорной системе.

Причинами для этого являются:

- системные ограничения
- т.е. способность организации операционной системы прямо зависит от количества организуемых ресурсов, которые неизбежно должны быть в состоянии ожидания, если они представляют собой сверхкритическое количество.
- алгоритмические или программные ограничения
  имеется ввиду, что эффективность обработки зависит от возможностей степени параллелизации.

Общая теория названных ограничений еще не существует. Подходы к такой теории возникают скорее всего из первого ограничения, чем из второго, которое более глобальное. Кроме этих предположений тоже на основании надежности существуют сомне-

ния в большом количестве установленных процессоров в мультипроцессорной системе. Здесь имеется ввиду, что не принимая особые структурные мероприятия, надежность уменьшается при увеличении количества оборудования.

Конкретные высказывания о воросе количественной оптимизации многопроцессорных систем могут быть только результатом взаимодействия теоретических и экспериментальных исследований, что существенно зависит от степени зрелости промышленности изготовления интегральных схем. Здесь хотелось только показать и наглядно пояснить этот вопрос.

Цель настоящего доклада состояла в том, чтобы указать на данную проблему и наглядно обрисовать ее.

ЛИТЕРАТУРА

/1/ Cooley, J.W., Tukey, J.W.:

An algorithm for the machine calculation of complex Fourier Series
Mathematics of Computation, 19 /1965/, 297-301.

/2/ Kober, R.:

A fast communication processor for the SMS multiprocessor system
Euromicro, 1976.

/3/ Krien, R.:

Die numerische Berechnung der zweidimensionalen diskreten Fouriertransformation mit Hilfe der Schnellen Fourier-transformation
Nachrichtentechnik Elektronik, 24 /1974/ 1, 22-26.

ЯДРО ОПЕРАЦИОННОЙ СИСТЕМЫ В МУЛЬТИМИКРО-ПРОЦЕССОРНОЙ СИСТЕМЕ

Хротко Габор

Будапешт

Введение

Отличительными чертами мультипроцессорных систем /МПС/ являются следующие:

- содержат два и больше процессоров со сравнимой производительностью;
- процессоры совместимо используют оперативную память, каналы и внешние устройства;
- МПС управляется общей операционной системой /ОС/, обеспечивая взаимодействие процессоров и их программ на уровне задания, шага задания и процессоров.

Для МПС, построенных на микропроцессорах, относительно высокая стоимость внешних устройств по сравнению со стоимостью процессоров требует

- обеспечения возможности параллельной работы одновременно всех внешних устройств.

Это требование можно удовлетворить построением автономного канала /мультиплексного или селекторного/ для каждого внешнего устройства.

Основными задачами ядра ОС в МПС /рис. 1/ является организация взаимодействия процессоров и распределение ресурсов между ними. Ядро является узким местом, т.к., будучи центральным ресурсом системы, его услугами пользуются все остальные ресурсы. Это приводит к необходимости свести до минимума те функции, которые сосредоточены в ядре.

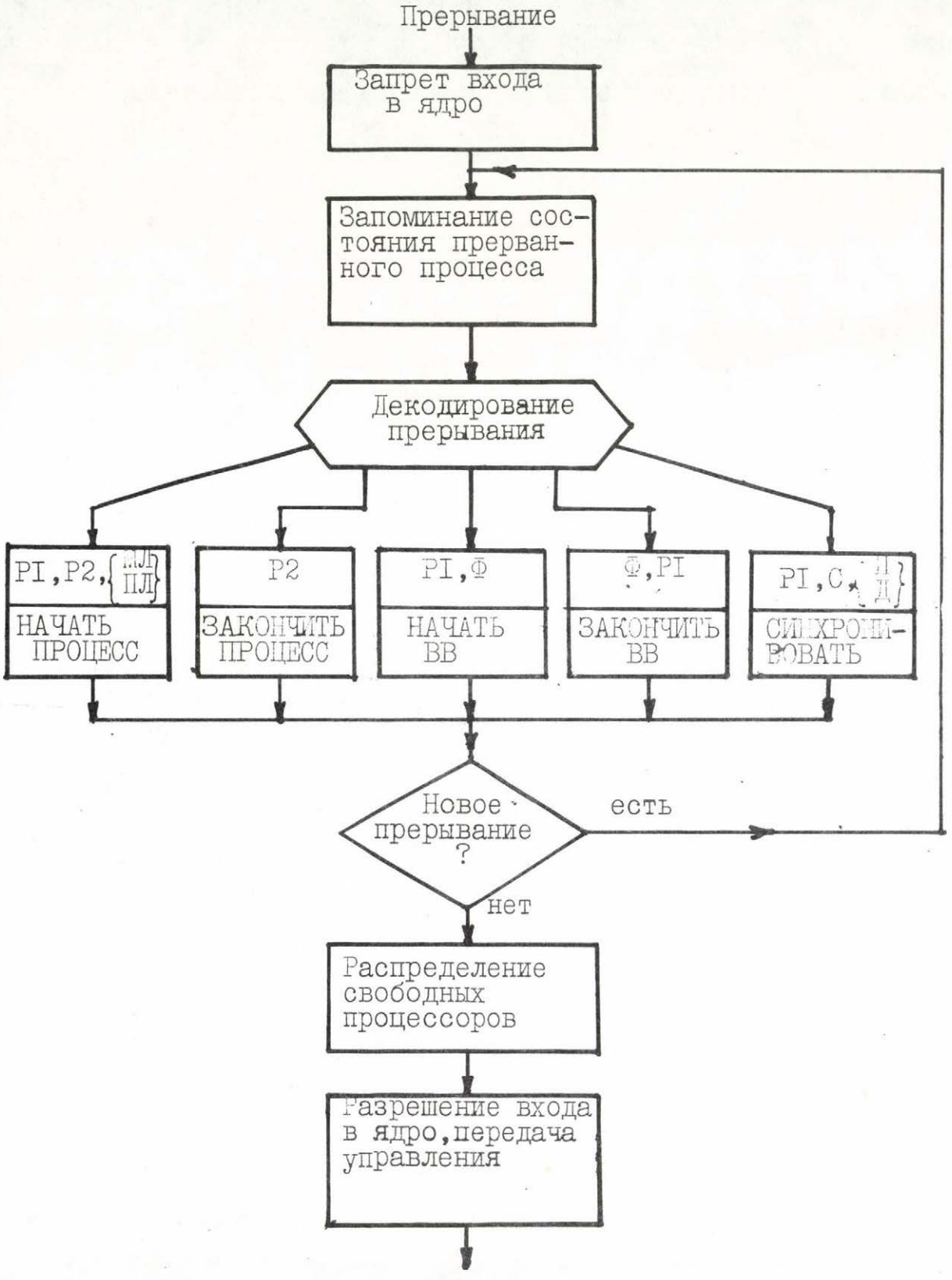


Рис.I. Структура процедур ядра

Структура процессов

Под процессом понимаем последовательность действий при выполнении команд некоторой программы или подпрограммы /формальное определение процесса смотри в 1 /. При решении заданий пользователей в МПС процессы образуют древовидную структуру /рис. 2/.

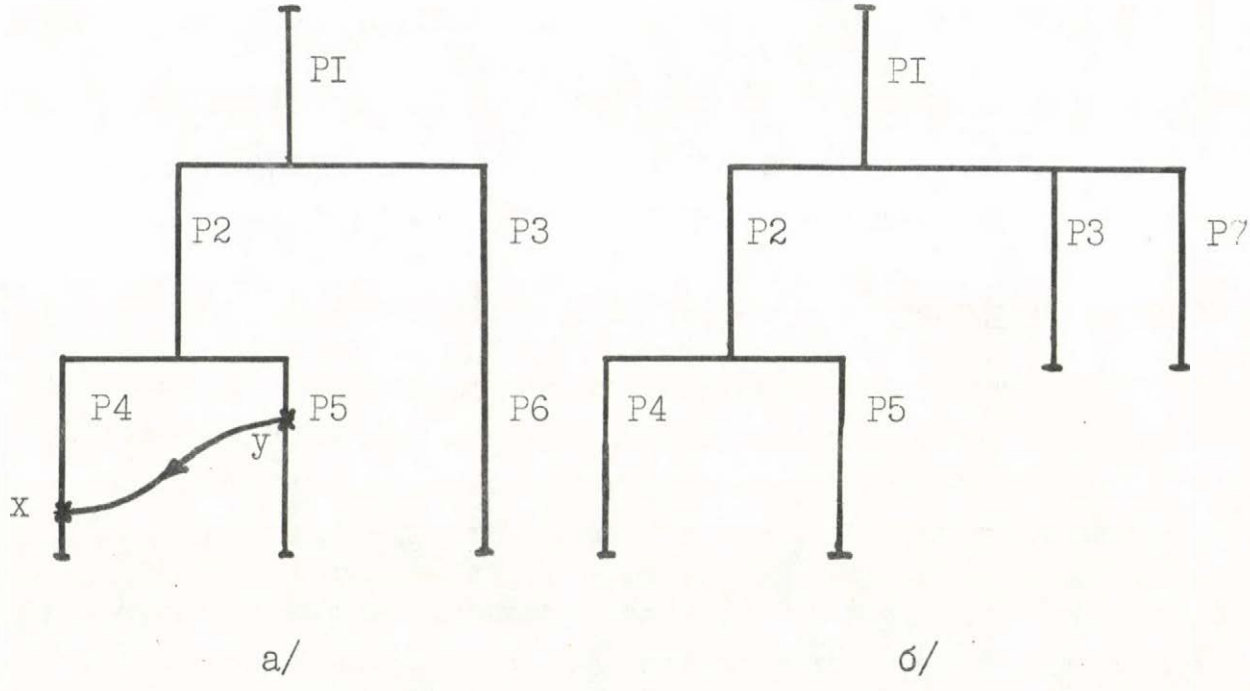


Рис. 2. Структура процессов задания.

В такой структуре возможны два вида связей между процессами:

а/ старший - младший, б/ параллельный.

Для связи "старший - младший" /напр., P3 и P6 на рис. 2.а/ характерно, что старший процесс, создав младший, блокируется до окончания действия младшего. По этой причине только нижние, свободные ветви /процессы/ дерева могут быть активными, т.е. выполняться на центральных или канальных процессорах.

Младшие процессоры, исходящие от общего старшего, называются параллельными. Из них одни создается старшим, а остальные генерируют друг друга. Параллельные процессы действуют независимо друг от друга, кроме случая синхронизации. Например, на рис. 2.а синхронизация между P4 и P5 означает, что процесс P4 блокируется перед командой "x" , пока процесс P5 не выполнил команду "y" .

Построение дерева динамически меняется, поскольку в ходе выполнения некоторые процессы заканчиваются, а другие создаются /сравни рис. 2.а и 2.б, показывающие два последовательных момента в жизни данного задания/. В МПС в каждый момент времени присутствует столько таких древовидных структур, сколько заданий пользователей обрабатывается в системе.

Состояние процессов

В круг ресурсов системы, кроме физических /процессор, память, внешние устройства/, включаем и программные ресурсы, т.е. общедоступные подпрограммы, например, модули ОС. Пользователи тоже имеют право создать общедоступные подпрограммы.

Имея в виду вышеописанное обстоятельство, для процессов определим три взаимоисключающих состояния: активное, готовое и блокированное. Процесс активный , если выполняется на процессоре /центральном или канальном/. Готовый процесс ждет освобождения нужного физического ресурса. Процесс находится в блокированном состоянии в следующих случаях: а/ имеет один или несколько младших, б/ хочет создать младший или параллельный процесс, но тот занят, в/ ждет синхронизацию от параллельного процесса.

Построение ядра /рис. 1/

Связь между активными процессами и ядром обеспечивается через традиционную систему прерываний. Число возможных прерываний большое, но процедурой "декодирование прерываний" они сводятся к пяти основным запросам /А-Д/, которые обрабатывают-

ся соответствующими процедурами ядра:

А - НАЧАТЬ ПРОЦЕСС - процесс Р1 хочет создать процесс Р2, который будет младший /МЛ/ или параллельный /ПЛ/.

Б - ЗАКОНЧИТЬ ПРОЦЕСС - процесс Р2 заканчивается, необходимо уничтожить его взаимосвязи с другими процессами и освободить его ресурсы.

В - НАЧАТЬ ВВ - процесс Р1 хочет начать ввод-вывод на внешнем устройстве У.

Г - ЗАКОНЧИТЬ ВВ - процесс Р2 хочет закончить ввод-вывод на внешнем устройстве У.

Д - СИНХРОНИЗИРОВАТЬ - процесс Р1 хочет производить операцию "проверить семафор" - операция П /С/ или операцию "увеличить значение семафора" - операция Д /С/. Этот запрос может получить аппаратную реализацию и в этом случае не входит в ядро. В то же время можно представить себе более сложные функции синхронизации, которые требуют программную реализацию в ядре [2] .

В дальнейшем сосредоточим внимание на процедурах А-Г. Процедуры "запрет входа в ядро", "запоминание состояния прерванного процесса", "декодирование прерывания", "разрешение входа" и "передача управления" реализуются в соответствии с возможностями данной МПС. Процедура "распределение свободных процессов" содержит выбор из списка процессов, готовых занять процессор, и алгоритмически не отличается от процедуры "ЗАКОНЧИТЬ ПРОЦЕСС".

Структура данных ядра

Данные, описывающие процессы и ресурсы образуют три списка: список физических ресурсов /СФ/, список программных ресурсов /СП/ и список процессов /СР/. СФ и СП являются линейными

списками, создаются и заполняются при генерировании Супервизора. Список СП является ценным списком и заполняется ОС /ядром/ в ходе решения заданий пользователей. Ниже показано содержание одного элемента из каждого списка.

СФ - список физических ресурсов

1 - имя /тип/ ресурса /$Ф_i$/
2 - имя процесса, занимающего ресурс /$Р_j$/
3 - указатель на список в готовых процессах /СГ / $Ф_i$//
4 - замок ресурса /З/$Ф_i$//
5 - число свободных ресурсов типа Ф

Ф1	
Ф2	
.	
.	
$Ф_i$	1 2 3 4 5
.	
.	

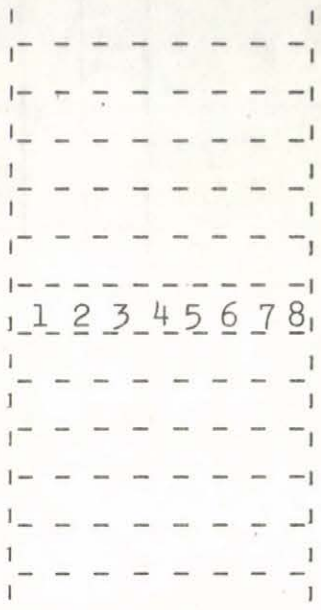
СП - список программных ресурсов

1 - имя модуля /$П_i$/
2 - имя процесса, образованного из данного модуля /$Р_j$/
3 - указатель на список блокированных процессов /СБ/$П_i$//
4 - ключ модуля /К/$П_i$//
5 - число процессов, образованных из данного модуля $П_i$

П1	
П2	
.	
.	
$П_i$	1 2 3 4 5
.	

СП - список процессов

1 - имя процесса /P_i/
2 - имя модуля из СП
3 - имя старшего /P_i/СТ//
4 - имя младшего /P_i/МЛ//
    или имя ресурса /P_i/Ф//
5 - указатель на параллельный
    процесс
6 - указатель в списке СГ или
    СБ /готовый или блокированный/
7 - ключ процесса /К/P_i//
8 - описание процесса: приоритет,
    адрес, имя задания, програм-
    мный счетчик



ЗАМЕЧАНИЯ:

1/ Если элемент списка СФ описывает несколько одинаковых ресурсов, то ячейка 2 данного элемента не используется.

2/ Если элемент списка СП списывает модуль, из которого можно создать не один, а больше процессов, то ячейка 2 данного элемента не используется.

3/ Элемент списка СП называется документом процесса. Документ заполняется при создании процесса и стирается при его окончании.

Изображение взаимосвязей в списках

Список СР отражает взаимосвязи /старший-младший, параллельный/ и состояние /активный, готовый, блокированный/ процессов системы. На рис. 3 показан СР и в нем изображен пример взаимосвязи процессов с рисунка 2.а . На рис. 4 показаны списки СФ, СП и СР и изображены взаимосвязи между процессами и ресурсами /физическими и программными/.

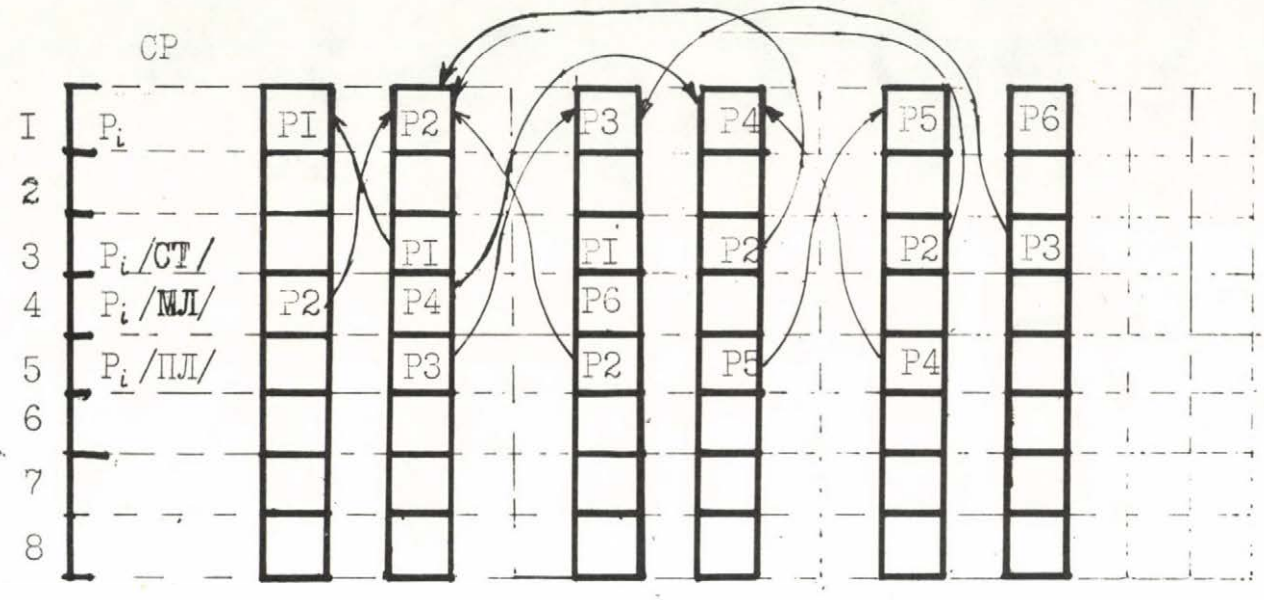


Рис.3. Изображение взаимосвязей процессов в списке СР

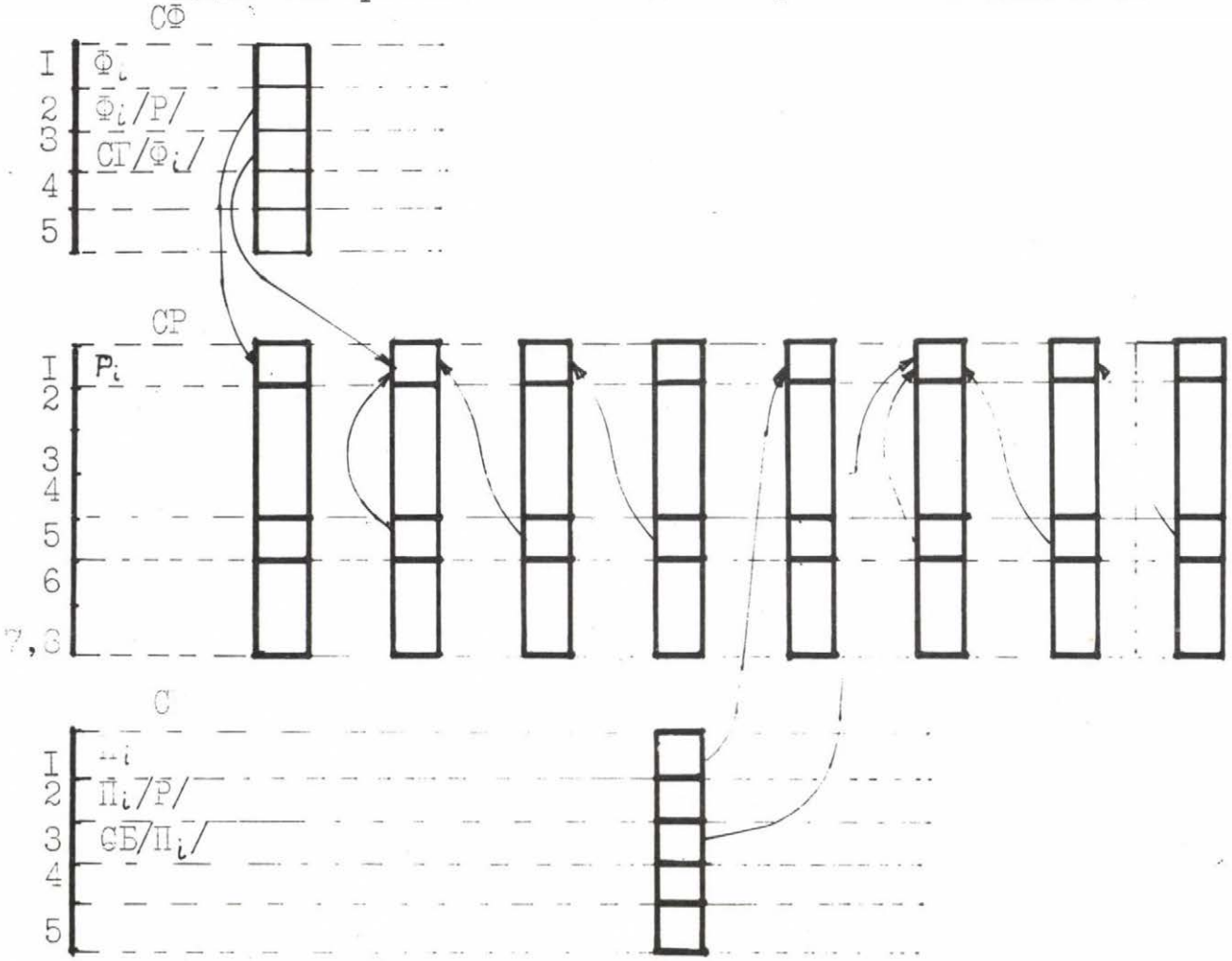


Рис.4. Изображение взаимосвязей процессов и ресурсов в списках СФ, СР и С.

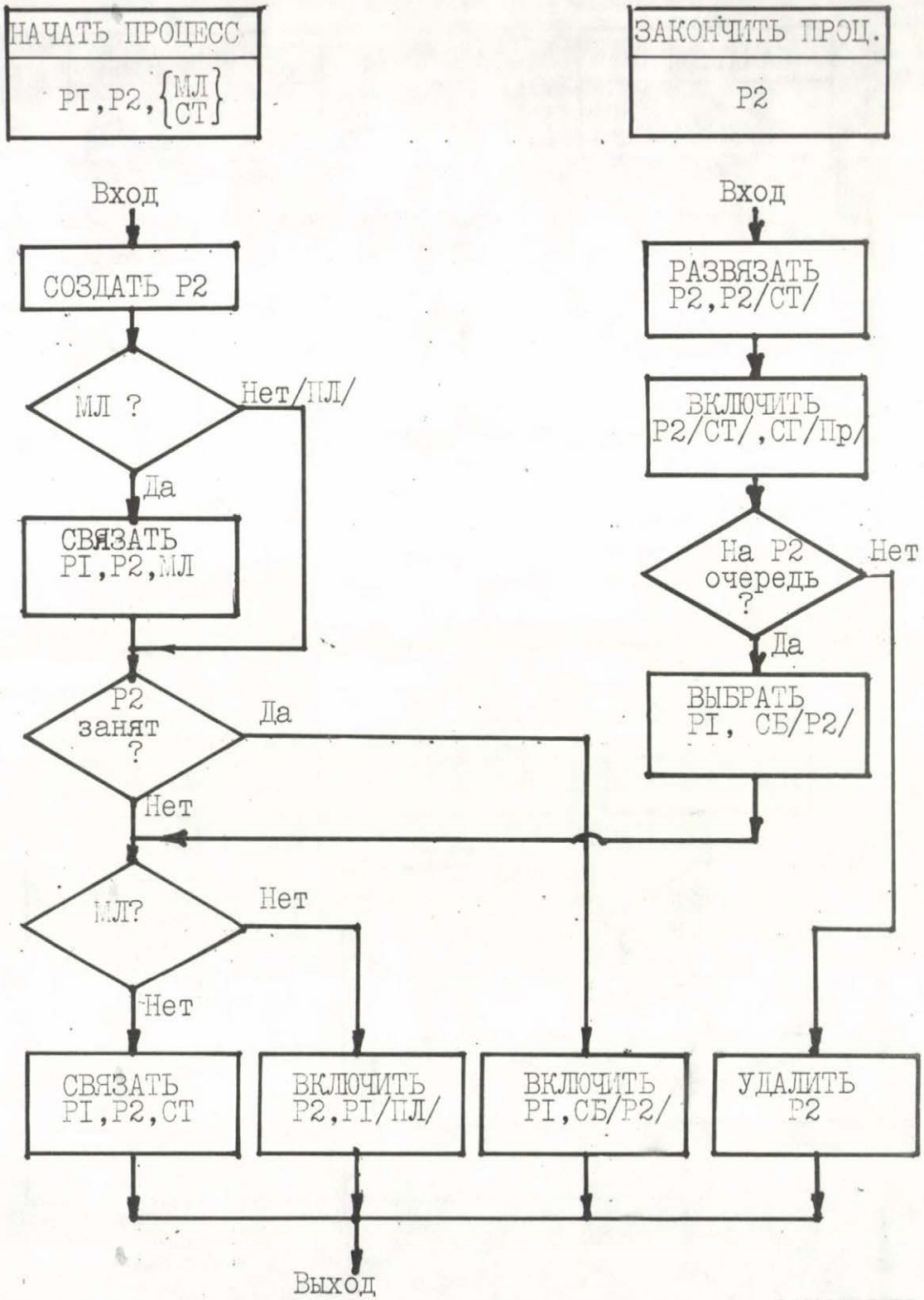


Рис.5 Блок схема алгоритмов НАЧАТЬ ПРОЦЕСС и ЗАКОНЧИТЬ ПРОЦЕСС

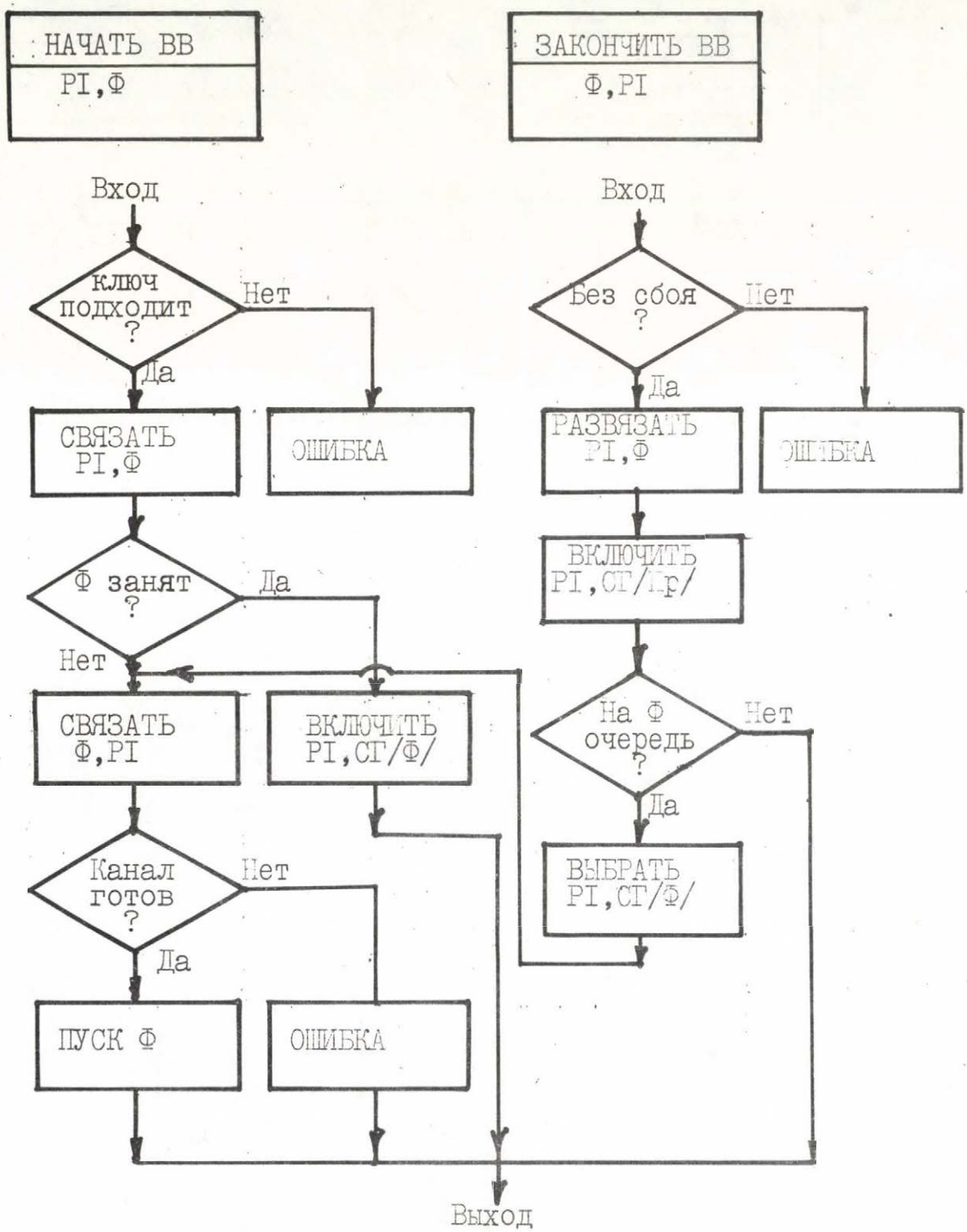


Рис. 6. Блок-схема процедур НАЧАТЬ ВВ и ЗАКОНЧИТЬ ВВ

Блок-схема некоторых процедур ядра

На рис. 5 и 6. показаны четыре алгоритма ядра:

НАЧАТЬ ПРОЦЕСС, ЗАКОНЧИТЬ ПРОЦЕСС, НАЧАТЬ ВВ, ЗАКОНЧИТЬ ВВ.

Объяснения некоторых выражений /макрокоманд/:

СОЗДАТЬ - создать документ для указанного процесса.
УДАЛИТЬ - удалить документ указанного процесса.
СВЯЗАТЬ - отметить указанную связь /МЛ, СТ, ПЛ или Ф/ между двумя процессами /процессом и ресурсом/ в документе первого процесса.
РАЗВЯЗАТЬ - взаимосвязь между процессами /процессом и ресурсом/ стереть в документе обоих.
ВКЛЮЧИТЬ - включить процесс в указанный список /СГ, СБ или параллельный/.
ИСКЛЮЧИТЬ - исключить процесс из указанного списка.
ВЫБРАТЬ - выбрать следующий процесс из указанного списка.
ОШИБКА - создать процесс анализа ошибки.

Выводы

1/ Описанное ядро ОС реализует

- сложные взаимосвязи между процессами,
- синхронизацию между параллельными процессами,
- динамическое распределение физических ресурсов, осуществляя "взаимное исключение" в состязании процессов за ресурсы.

2/ Списковая структура данных ядра с применением ассоциативного метода организации памяти позволяет осуществить эффективный поиск требуемых данных.

3/ Процедуры ядра состоят из небольшого числа простых макрокоманд манипуляции списками. Это качество дает основание для микропрограммной реализации процедур.

ЛИТЕРАТУРА

1/ Horning J.J., Randell B.: Process structuring.
Computing Surveys 5, 1 /March, 1973/, pp. 5-30.

2/ Presser L.: Multiprogramming Coordination.
Computing Surveys 7, 1 /March, 1975/, pp. 21-44.

CASSETTE MEMEORY FOR DATA STORAGE CONTROLLED BY MICROPROCESSOR

J. MOUDRÝ
L. JAKUŠ

Institute of Physics, Czechoslovak Academy of Sciences
Prague, Czechoslovakia

SUMMARY

The use of the microprocessor (MP) Intel 8080 as a control element for a cassette memory (CM) unit type PK-1, is described in the paper. The MP controls all the mechanical functions of the CM unit as well as the input and the output of data.

1. THE PURPOSE OF THE DEVICE

Devices using magnetic media as means of data storage have become more and more common in recent years. They have some better properties than paper tape (PT) devices, e.g. possibility of repetitive storage, higher speed and capacity, more reliable and less noisy work and they need less intervention by an operator during their work. On the other hand, because of a constant speed, the magnetic media need the data to be transferred synchronously while the data to be stored may be generated asynchronously. That is why we have decided to build up an interface enabling the storing/reading of digital information in/form a magnetic cassette in an asynchronous mode of work. The interface will enable the CM to be connected directly to any device normaly working with a PT reader type FS 1500 (ZPA, Czechoslovakia) or a PT puncher type PE 1500 (Facit, Sweden).

2. TECHNICAL DATA OF THE ELEMENTS

The CM unit type PK-1 (Meramat, Poland) is used as a cassette tape drive. The electronics for motor drive control and for writing and reading data are part of the unit. All the control and data signals are in TTL levels. The dual gap (read-after--write) head enables the written data to be checked during writing.

-Control signals to the PK-1:
SELECT (motors powered)
FORWARD
REVERSE
FAST (search tape speed)
REWIND
WRITE
READ

-Control signals from PK-1:
READY (cassette in, motors running)
EOT (end of tape)
WRITE ENABLE
SIDE A/B (not used in our case)

The recording meets the ECMA-34/ISO-3407 standards. It is phase encoded and there are two tracks on the tape. The information is divided into blocks, 32-2048 bits each. The nominal interblock gap is 20 mm.

The Intel 8080 MP operates on 8 bit words with an average instruction time of 8 µs approximately. 1024 words PROM and 768 words RAM are used of possible 64000 words memory.

The program for controlling the MP is stored in PROM, while RAM is used for variables and as a cassette buffer (2blocks).

The data is transferred from/to a cassette via an Intel 8251 Communication Interface working in synchronous mode. The external data go through an Intel 8255 Peripheral Interface. The other signals go via Intel 8212 I/O Ports.

3. SPECIAL ELECTRONICS

Phase encoding is a synchronous recording method, in which "zeros" and "ones" are defined by opposite flux transitions. An additional flux transition must be placed between two identical bits, to establish the proper coding.

The serial data of 8251 are level coded. So, special circuits must be built up between the 8251 and the read/write head to convert the serial data of the 8251 to phase encoded mode or vice versa.

The EOT signal from the CM unit is a pulse (10 ms) approximately 50 cm before the end of the tape. For the sake of simplicity of the program, this signal is only checked for at the end of a block, i.e. each 1 s. Instead of EOT an EOTMEM latch signal is used by the MP. The EOTMEM is set by the trailing edge of the EOT and is cleared on reading by the MP.

If the information read is not in order (e.g. due to impurities on the tape), it is desirable to return the tape and to read or - in a case of recording - to write the block once more. This is made possible by the checking of an EOB (end of block) signal. The EOB will be generated after not finding at least 3 bits of serial information on some 2 mm of the tape (the tape is erased in the interblock gaps), and it will be cleared on reading it by the MP.

4. DRIVE CONTROL

Almost all the functions of the CM unit (all the same CM being on-line or off-line) are controlled by the MP. Switching on/off and loading/unloading the cassette has to be carried out manually.

After the device has been switched on, a program waiting for the loading of the cassette is run in the MP. Once this

is done the MP begins to control all the activity of the device. Unloading the cassette is only possible when the motors have been switched off, which would happen if no command comes within 5 s.

The MP accepts the external commands (CM on-line) "read" (read a word) or "write" (write a word). Other commands are generated by an operator's intervention. He handles the following elements:

MANUAL/EXTERNAL switch
WRITE TM push-button
FIND FILE push-button
FILE NO XX two decades selector

The role of the MANUAL/EXTERNAL switch is to control which commands will be accepted by an MP. At the same time, the EXTERNAL position blocks the unloading of the cassette.

The other control elements enable required data on the tape to be found.

The capacity of one side of the cassette is approximately 250000 words and it will generally be composed of more than one file of information. The files are supposed to be numbered from 1 to 99. They are separated one from the other by a special block, called a tape mark (TM).

When using a CM instead of a PT device we note that the device to which the CM is connected is not intended to have the ability to close the information file by a TM; the WRITE TM push-button allows us to do it manually.

When reading data from a PT, it is assumed that the punched tape is so loaded that the data to be read is what is required. The beginning of a proper file on a cassette is set up by pushing the FIND FILE push-button after setting a proper file number by the FILE NO XX selector. Then the MP reads the XX value and starts moving the tape fast in the corresponding direction. The TM-s are counted during the motion, and the motion is stopped

at the proper information file.

Finding the file during the fast motion of a tape is not a simple task. The speed of the tape varies in range 1 to 15 when accelerating from normal to fast speed and the phase decoding electronics of a CM unit will not work well. The normal method of data decoding would not be reliable enough and we have decided to try another method of TM detection which seems to be more promising.

The fact is that the number of flux transitions of phase encoded records is smaller at a TM than at any other block. So, when looking for a TM, only the number of flux transitions need be counted in the interval of 1 - 4 ms. If this number is equal to zero, the read head is just in the interblock gap; then the number of all the actual transitions previously detected corresponds to the previously read block. If this number is equal to the number of transitions in a TM block record, then the last block detected had to be TM.

We note that the TM detection time interval must be smaller than half of the interblock time at the maximum speed of tape. Then it would be possible to detect no flux transitions during the interblock reading at least once. At the same time, the TM detection time must be greater than one bit time at a normal speed, so as to find some actual transitions during the block reading.

During operation the MP checks for a proper timing of all the signals and for legality of commands. This is a necessity to some extent, but also more reliable operation of a CM device may be ensured in this way. If an error occurs, the tape motion is stopped and the error code is shown on a simple display. After correcting the error or fault source, the CM device may resume its work.

ACKLOWLEDGEMENT

We would like to thank our colleaquaes V. Kohl and M. Záruba for their valuable notes in discussions.

LITERATURE

Cassette Tape Drive PK-1, product description. MERAMAT, Warsaw.

8080 Microcomputer Systems User's Manual, July 1975. INTEL CORP., Santa Clara.

ECMA-34, 1971.

ISO-3407, 1976.

Fotoelektrický snímač děrné pásky FS 1501, FS 751, návod. ZPA Košíře, Praha.

FACIT PE 1500 Tape Punch, manual. FACIT, Solna.