TECHNICAL MANUAL

RCSL: 51-VB754
Author: Allan Giese
Edited: February 1970

RC 4005

TECHNICAL DESCRIPTION

VOLUME 1

A/S REGNECENTRALEN
Falkoneralle 1
2000 Copenhagen F

CONTENTS:

A/S REGNECENTRALEN
SCANDINAVIAN INFORMATION PROCESSING SYSTEMS

RCSL NO: 44-D6
TYPE : Report
AUTHOR : Allan Giese
EDITION : October 1968

RC 4000, HARDWARE ORGANIZATION

Abstract

This paper gives a short description of the hardware organization of the central processor and the input/output system. The maintenance problem is discussed and diagnostic programs are presented. The problems around the selection and use of high - speed integrated circuits are considered together with mounting and wiring techniques.

CONTENTS

1. CPU ORGANIZATION

1.1. Design Considerations

Experience from the design of the GIER computer showed that the control of a computer from a microprogram store containing the microprogram implied a number of advantages such as: flexibility, simple structure, good reliability, and moderate component cost. The disadvantage of such a scheme is that the repetition rate for control signals is limited to the cycle time of the microprogram store. In our case, however, this is not a severe obstacle, since the main factor which influences the instruction execution time is the cycle time for the core store and not that of the microprogram store. The cycle time for the core store is 1500 nanoseconds for read-restore and 2500 nanoseconds for read-modify-write, whereas the cycle time for the microprogram is only 500 nanoseconds. It was therefore decided to control the RC 4000 computer by means of a microprogram.

The microprogram store is organized like a general store having a maximum capacity of 1024 words, each of 100 bits. The store is constructed as a decoding network implemented entirely by means of integrated circuits. For maintenance purposes, the microprogram can be controlled to be executed step by step.

The repetition rate of 500 nanoseconds implies that all the microoperations specified in one microstep are active in this period. Certain of the logical operations, derived from the microoperations, must even within this time interval follow a strictly timed format in order for the computer to execute its instructions. The period of 500 nanoseconds is consequently subdivided into 10 intervals by pulses, having a pulse width of 100 nanoseconds and delayed 50 nanoseconds with respect to each other. The circuit which accomplishes this is a feedback shift register controlled by an oscillator.

1.2. Register Structure

The registers of the RC 4000 are grouped around a common bus line system, a configuration which has the advantage that all the registers can transfer data directly to one another. The block diagram, in the appendix, shows not only the data paths between the registers of the Arithmetic Unit, but also the paths to the Core Store Unit, the Interrupt Unit, and the Input/Output Unit. The direction of flow is indicated by means of arrows, and the bit numbers are written on the data paths.

Core Store Data Register;STdata(0:27):
Data communication between the Core Store Unit and the Arithmetic Unit takes place via STdata. The register is logically divided into three groups.

STdata(O:23)	Specifies a 24-bit dataword.
STdata(24:26)	Specifies the protection bits.
STdata(27)	Specifies the parity bit. (Not shown on the block diagram)

Core Store Address Register STaddr(6:22):
This register is able to address the maximum core store configuration of 128 K words. STaddr may therefore have fewer bits in actual installations.

Working Registers W[0](0:23), W[1](0:23), W[2](0:23), W[3](0:23):
The four working registers, each of 24 bits, can be specified as the result register. Three of the registers (W[1], W[2], and W[3]), also function as index registers. The current index register is specified by the instruction format. Since the working registers also act as the first four locations of the core store, it is possible to execute instructions stored in these registers. Like the rest of the storage words each register is supplied with its own protection bits (PB).

Protection Bits PB[0](0:2), PB[1](0:2), PB[2](0:2), PB[3](0:2):

The four 3-bit registers determine together with the protection register, PR, whether the corresponding working register is protected or not. For example, W[0] is protected if PR(PB[0]) equals one, otherwise W[0] is unprotected.

Protection Register PR(0:7):

This 8-bit register specifies the protection status for the eight possible values of the protection bits. PR(0) is permanently equal to one.

Instruction Counter IC(5:22):

The instruction counter contains the word address of the instruction to be executed. IC is normally increased by one after execution of an instruction, but jump instructions insert explicitly the jump address in IC. A decoding of IC detects when the storage capacity is exceeded.

Sequential Counter SC(11:23):

The 13-bit sequential counter is used to determine the number of iterations in, for example the multiply, divide, shift, and normalize instructions. For each iteration, SC is either increased or decreased by one.

In floating-point operations, SC is also used for temporary storage of the resultant exponent.

Funtion Register FR(0:11):

When a new instruction is fetched from storage, the function part of the instruction, i.e. the twelve left-most bits of the instruction, is assigned to the function register. FR is divided into the following five subfields:

FR(0:5)	Specifies 64 basic instructions.
FR(6,7)	Specifies one of the working registers as result register.
FR(8)	Indirect addressing.
FR(9)	Relative addressing.
FR(10,11)	Indexed addressing.

Exception Register,EX(21:23):

An exceptional outcome of an arithmetic instruction or an input/output instruction has the effect of setting the two bits EX(22) and EX(23). Bit EX(21) specifies the precision (significance mode) for floating-point operations. The contents of EX can also be altered by means of the Exception Load instruction.

Storage Buffer,SB(0:23):

When a new instruction is fetched from storage, the address part (the displacement) of the instruction is assigned to the 12 right-most bits of SB. This displacement is then extended to a 24-bit signed integer. The address modifications take then place in SB in order to obtain the effective address for the data word. In consequence of, that the address for the data word is generated in SB, we have also established an address path between the core store address register (STaddr) and SB. Moreover, the 24-bit data word, which is read from the core store, is transferred to SB just as a data word from the Arithmetic Unit to the core store also passes through SB. It is therefore no coincidence that the name for this register is storage buffer.

Either the contents of SB or its complement may be employed as input data for the ADDER circuitry. This makes it possible for the ADDER to perform addition and subtraction.

Protect Key,PK(0:2):

Every time a word is read from storage, the protection bits are inserted in PK, and reversely, PK determines the protection bits to be stored.

Storage Buffer Extension,SE(0:13):

SE forms together with SB a 38-bit register which is used in floating-point operations to store the mantissa. The two extra bits SE(12,13) play an important role in the rounding calculations.

Either the contents of SE or its complement may be employed as input data for the extended ADDER circuitry.

Adder Circuitry,ADDER(-1:23):

The adder is a parallel adder utilizing extensive carry look-ahead technique. 25 bits are added in typically 120 nanoseconds and 39 bits in 140 nanoseconds.

A-Register,AR(-1:23):

AR constitutes together with SB the two data parts to the ADDER circuitry. Bit AR(-1) detects if an overflow situation occurs.

A-Register Extension,AE(0:13):

The combined 38-bit register ARconAE is used in a manner very similar to that of SBconSE.

B-Register BR(O:23):

BR is mainly used by multiply, divide, and double length instruction, whereas other instructions only use the register to store temporary results.

If an instruction requires two successive storage words, the address of the second word is held in BR while the first word is fetched from core store. An address path is therefore provided from BR to STaddr.

B-Register Extension BE(O:11):

The BR register is extended in order to be able to execute the floating-point division instruction.

Display and Manual Control of Bus.

This unit consists of the two registers, DR(-1:23) and DP(O:2) which control BUS(-1:23) and PBUS(O:2), respectively.

Interrupt Register IR(O:23):

Each bit in IR is connected to an external or an internal device, which sets the bit in accordance with some specified condition. The left-most bits are assigned highest priority.

Interrupt Mask IM(O:23):

IM determines whether a given interrupt request should be honoured or not. IM(O) is permanently equal to one.

Input/Output Unit.

Each input/output device has in principle a 24-bit buffer register plus the two status bits Disconnected and Busy.

2. INPUT/OUTPUT

2.1. Introduction

Experiences have shown that many programs do not use the central processor very effective, because a great percentage of the total computer time is spent, waiting for the input/output devices to complete their operations. This fact has resulted in the development of monitor systems capable of executing several programs in parallel. The advantage of such a scheme is best illustrated by considering two programs A and B. When program A arrives to an input/output instruction, only a short message is sent to the selected device and the monitor will then transfer control to program B. Control is first handed over to program A again, when the external media has terminated. Hence, the central processor does not have to wait for the relatively slow devices. A scheme like this, obviously, requires that each device has its own controller, which can initiate and operate the device independent of the program.

The RC 4000 has two data channels for communication between the central processor and the peripheral devices; one for low-speed devices and one for high-speed devices.

2.2. Low-Speed Data Channel

Slow, character-oriented devices like input/output typewriters, paper tape punches, and paper tape readers are connected to a single low-speed data channel, which communicates directly with the internal working registers. Each device has a separate buffer register of 24 bits, which transmits or receives one character a time to or from the external data medium.

The data channel consists of an input/output bus, with 24 bits for transfer of data to or from device buffers and 6 bits for channel control information. Transfers of data between working register and device buffers take place one at a time under program control. Transfers between buffer registers and external data media, however, are controlled independently by the devices, so that several such transfers can occur simultaneously.

2.3. High-Speed Data Channel

Input/output devices such as magnetic drum stores, magnetic disc stores, and magnetic tape stations, which transmit large volumes of data at high speeds, are connected to a single high-speed data channel. This channel provides input/output directly to or from the internal store on a cycle-stealing basis. Program execution and input/output operations occur simultaneously.

Block transfers can take place on several devices at once. A multiplexer switches rapidly among the devices, connecting them whenever they are ready to transfer a complete data word to or from the store.

The method of solving the multi-access competition for core store cycles influences the kind and number of peripheral transfers which may be simultaneously executed. If more than one device is ready for transfer of data, the one having the greatest transfer rate should be first served, since this device will issue its next request before the other. This strategy is adopted for the RC 4000 computer and it is implemented by means of a priority system.

The capacity of the channel is 500,000 words per second.

2.4. Input/Output Control

There are four basic input/output commands: read, write, control, and sense. A read command initiates the input of a character to a buffer register. The write and control commands transfer the contents of a working register to a buffer register and initiate output and control operations respectively. A sense command requests a device to transfer a status word from its buffer to a working register. The exception register indicates whether the status word is available; this is the case only after the termination of an operation. The status word contains the last input or output character as well as information about parity errors and other exceptional events.

The high-speed devices use the low-speed channel to transfer commands and addresses of buffer areas in the store. All devices are thus controlled by the standard input/output instruction.

3. DIAGNOSTIC PROGRAMS FOR TESTING OF HARDWARE

3.1. Introduction

Experiences have shown that the use of microelectronic circuits causes maintainability problems, because errors occur so seldom that it is difficult for the service engineers to have sufficient training in error location. Thus, the new systems coupled with older maintenance techniques, will result in machine break-downs for long periods, although of lower frequency. The result is the apparent paradox of more reliable systems being less maintainable. Diagnostic programs have therefore been developed for the RC 4000 computer system.

Many of these programs do not only detect failures, but are also able to isolate them to specific modules. The programs fall into two categories; the first group comprises testprograms for the central processor, whereas the programs which check the peripheral devices constitute the second group. A major difference between the two types of programs is that the testprograms for the peripheral devices are constructed under the assumption that the central processor itself works correctly, but this, of course, can not be taken for granted for programs belonging to category one. The diagnostic programs for the central processor are therefore often referred to as self-check programs.

3.2. Diagnostic Programs for the Central Processor

The self-check programs are oriented towards the checking of hardware failures in microcommands, decoding networks, arithmetic registers, etc. An O.K. message is typed on the operator's typewriter if the outcome of these tests shows that the central processor works correctly. In the case where an error is found, information concerning its nature is typed out whenever possible, but it should be noted that some errors have the effect of destroing the programs completely and thereby prevent further communication between the operator and the self-check programs. Under such circumstances, the operator must turn to the simple diagnostic programs, which can be executed manually from the technical panel.

The self-check programs are usually controlled by the monitor system which then takes care that they are executed at regular intervals. A special version of the program is also provided which can be loaded by means of the autoload key.

Special programs for testing the core store are also available. As an example, one of the programs will generate and write into the core store a worst-case bit pattern after which the program starts to read and check the store word by word. In the event of an error, the erroneous word and its location are indicated.

3.3. Diagnostic Programs for the Peripheral Devices

A diagnostic program is constructed for each type of peripheral device. The prime objective for this assembly of programs is to reduce the tedious task of manual error detection and location to a minimum. The programs are controlled from the operator's typewriter in a conversational mode.

The programs check for each device the four input/output commands (sense, control, read, write), the status bits, the local/remote status, plus an additional number of tests dependent on the peripheral device.

If a peripheral device does not complete its operation within a prescribed time, a watch dog timer comes into force and terminates the operation. This action is indicated by a status bit.

3.4. Automatic Error Indication

The computer system is constantly supervised for gross failures even when the diagnostic programs are not running. This is made possible by additional hardware, and concerns the following:

Core Store:	check for parity error.
Read-Only Store:	check for parity error.
Power Supply:	check for current and voltage levels.
Temperature:	check of the temperature in the cabinets.
Blower Assembly:	check for that the fans are running.

4. TECHNOLOGY

4.1. Integrated Circuits

The use of integrated circuits (ICs) has greatly influenced the design and performance of todays computers. These circuits offer advantages such as high reliability, high-speed switching delay time, and a wide operating temperature range. This clearly indicates that ICs were going to be used as the basic logic circuits for the RC 4000 computer.

The circuits, used in RC 4000, are the Transistor-Transistor-Logic integrated circuits (TTL) as supplied by Texas Instruments in Series 74. The characteristics are: noise margins minimum 0.4 volt, typical 1 volt; propagation delay time typical 13 nanoseconds; fan-out 10 for standard gates, 30 for power gates; low impedance at both logic levels with specified good capacity driving capability; only one supply voltage of +5 volts; O to 70 degrees centigrade operating range; voltage swing minimum 2 volts; power dissipation typically 10 mW per gate; 14- or 16-lead plastic dual-inline package. Integrated circuits from the compatible Series 74H, characterized by a propagation delay time of only 6 nanoseconds, have been employed where speed is of paramount importance, as for example in the adder circuitry.

The Series 74 comprise a number of different circuits from which we have chosen: NAND-gates having 2, 4, and 8 inputs, the 4-input power NAND-gate, plus the 2-wide 2-input AND -OR-INVERT gate, and the 4-wide 2-input AND-OR-INVERT gate. Two of the Series bistable elements are employed, namely the J-K flip-flop and the D-type flip-flop both of which are edge-triggered. A 4-bit binary adder is also used.

Special circuit functions, for example line drivers, level converters etc., which are not available as ICs have been constructed from discrete components.

4.2. Printed Circuit Boards

The ICs are mounted on dual-sided printed circuit boards, 11x14.7 cm, by soldering technique. The boards are provided with a 41-pin connector and test points for almost all output signals. The latter facilitates the error location on the board.

A total number of approximately 65 different board types are used for the implementation of the entire computer system. Several considerations have influenced the partitioning of the logic to the various board types. A small number of boards have the advantage of minimizing the spare part problem;

but on the other hand, too few board types imply that an excessive number of boards are necessary to construct even a simple unit. Another problem is that the physical distance between the sending and the receiving IC increases with the total number of boards used in the system, and this increased wire length may lead to severe noise problems and signal delays.

4.3. Wiring Technique

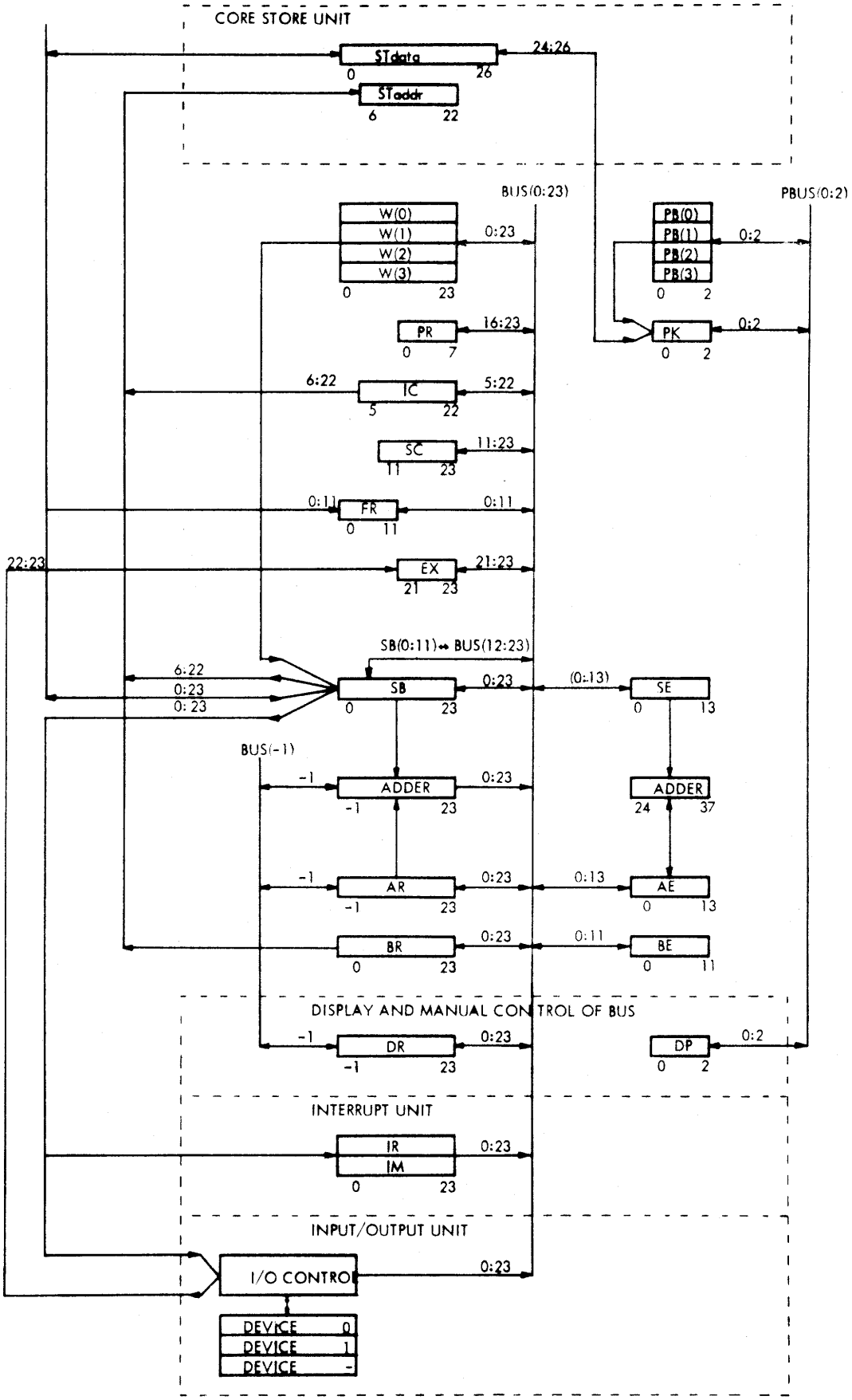
The interconnection between the boards is done by means of ordinary backplane wiring. It was soon realized that the number of connections for the central processor (500 boards) would exceed 10,000 clearly indicating that some sort of automatic data processing had to be done. We, therefore, decided to make a programming system, which covers the following two principal areas:

1) From the logic diagrams and a list over the number of used boards, supplied by the designer, the program will analyze all interconnections between the boards, and then assign a position number to each board in the rack. The program cannot produce an optimal solution because of the almost infinite number of possible combinations; but the solutions have proved to be satisfactory.

2) After the boards have been positioned, the program produces the wiring lists. These lists contain information about the addresses (i.e. the position- and pin-number of the terminal) to be interconnected and the necessary wire-length.

The programming system also incorporates very useful design and checking facilities.

APPENDIX



RC 4000

V 10690

REGISTER STRUCTURE

RCSL: 51-VB644
Author: Allan Giese
Edited: December 1969

TECHNICAL CONTROL PANEL

AND

OPERATOR CONTROL PANEL

FOR

THE RC 4000 COMPUTER

A/S REGNECENTRALEN
Falkoneralle 1
2000 Copenhagen F

CONTENTS:

This paper serves as an introduction for manual control of the RC 4000 Computer. The control is exercised by means of keys and pushbuttons on either the Operator Control Panel or on the Technical Control Panel. Visual indications of the machine status and the contents of registers are also given.

1. OPERATOR CONTROL PANEL

CPU POWER ON indicator	Green light indicates that all power supplies for the central processor function correctly.
EMERGENCY BREAK	When this knob is pulled, all power supplies are instantaneously disconnected.
OPERATOR CONTROL ON/OFF switch	With this key-operated switch in position OFF, the pushbuttons on the operator's panel are disabled. With the key in position ON and if at the same time the Technical Control Panel is disabled (i.e. the MODE SELECTOR switch on the Technical Control Panel is in position NORMAL), then the three pushbuttons RESET, START, and AUTOLOAD will function normally.
OPERATOR MODE indicator	Green light indicates that the OPERATOR CONTROL and the MODE SELECTOR keys are in positions ON and NORMAL, respectively. Moreover, the power supplies for the central processor must work satisfactorily.

RESET pushbutton

Pressing this button interrupts the program in progress. Thereafter, the return address is stored in storage word 10, and the computer is set in Monitor Mode with the interrupt system disabled. The computer remains in this reset state indicated by red light in the indicator RESET MODE, until the operator presses either the START pushbutton or the AUTOLOAD pushbutton.

The program will not enter the reset situation if an interrupt request is being served and at the same time the interrupt start address erroneously points at a non-existing core store location. This situation is indicated by no light in the CPU RUNNING indicator.

START pushbutton

When this pushbutton is pressed, the instruction counter is set to an address kept in storage word 14, after which the fetch cycle is initiated. The START pushbutton has no effect, unless the machine is in the reset state.

AUTOLOAD pushbutton

The autoload facility is used to initiate a boothstrapping routine that loads a program into the internal store from device number 0 (normally the paper tape reader). The AUTOLOAD pushbutton has no effect, unless the machine is in the reset state.

CPU RUNNING indicator

Green light indicates that the Technical Control Panel is switched off (MODE SELECTOR key in position NORMAL) and that the computer has executed a fetch cycle, i.e. an instruction has been fetched to be executed, within the last 200 milliseconds.

RESET MODE indicator	Red light indicates that the computer is in reset state waiting for the operator to depress either the START or the AUTOLOAD pushbutton
CPU PARITY ERROR indicator	Red light indicates that the computer has been stopped due to a parity failure in core store or microprogram store. The two types of errors can be distinguished by indicators on the Technical Control Panel.

2. TECHNICAL CONTROL PANEL

This panel is located behind the left-hand front cover of the central processor and it is mainly intended for control of the RC 4000 Computer during maintenance periods.

The pushbuttons on the panel are all inoperative when the MODE SELECTOR switch is in position NORMAL (Normal Mode), but turning the key to TECHNICAL position (Technical Mode) makes it possible for the operator to control and inspect the status of the computer.

Normally, the microprogram is executed and the computer is said to be in Running Mode. The opposite to Running Mode is called Stopped Mode in which case the microprogram is stopped.

Besides the Normal Mode and Running Mode (and their corresponding opposite modes) we do have two more modes, which are also opposite to one another, namely Computer Mode and Manual Mode. In Computer Mode, the next micro address is generated by the microprogram; in Manual Mode, the next micro address may be inserted manually by means of pushbuttons.

A complete list for the modes is therefore:

Normal Mode	Technical Mode
Running Mode	Stopped Mode
Computer Mode	Manual Mode

2.1. Control Panel Controls and Indicators

The following list gives a detailed description of the controls and indicators on the Technical Control Panel. If nothing else is explicitly stated, an indicator lamp lights to denote the presence of a binary one.

MODE SELECTOR NORMAL/TECHNICAL switch

With this key-operated switch in position NORMAL, all pushbuttons on the Technical Control Panel are disabled, and the computer is automatically operating in the following modes:

1. Normal Mode;
2. Running Mode; and
3. Computer Mode.

It is only possible to remove the key from the lock when the key is in position NORMAL.

When the switch is turned to TECHNICAL position, all pushbuttons function normally, and the computer operates in Technical Mode. The Operator Console is disabled in this mode.

CONTINUE pushbutton

Pressing this button sets the computer in Running Mode; i.e. the microprogram is executed.

SINGLE INSTRUCTION pushbutton

When this pushbutton is pressed, the computer stops after an instruction has been fetched (fetched cycle) but before it is executed (execute cycle). The contents of the function register (FR), the storage buffer (SB), and the instruction counter (IC) equal:

FR: operation byte.
SB: address byte extended to a 24-bit integer.
IC: address of the next consecutive instruction.
For jump and skip instructions the next instruction is not necessarily the consecutive instruction.

If SINGLE INSTRUCTION is pressed repeatedly, the computer steps through the program one instruction at a time.

If the microprogram is executing an interrupt request and if the start address of the interruption program erroneously points at a non-existing core store loaction, the microprogram will not be stopped by depressing the SINGLE INSTRUCTION pushbutton.

SINGLE MICRO INSTRUCTION pushbutton	When this pushbutton is pressed, the computer stops after each micro instruction and Stopped Mode is set. If SINGLE MICRO INSTRUCTION is pressed repeatedly, the computer steps through the microprogram one micro instruction at a time, so that the contents of registers can be observed in each step. Note that a step by step execution of the microprogram will not result in an instruction exception (IR(0)= 1) in two cases, namely: 1. the core store address exceeds the core store capacity; and 2. an unprotected program jumps to a protected location.
MAR COMPUTER CONTROLLED pushbutton	Pressing this pushbutton stops the computer when the current micro instruction is completed and sets the computer in Stopped Mode and Computer Mode.
MAR MANUAL CONTROLLED pushbutton	Pressing this pushbutton sets the computer in Manual Mode and in Stopped Mode.
JUMP SELECTOR REGISTER indicators	Indicate the contents of the 30-bit JS register.
MICRO COMMAND REGISTER indicators	Indicate the contents of the 69-bit MC register.
MICRO ADDRESS REGISTER pushbuttons and indicators	Indicate the contents of the 10-bit micro address to be executed. The micro address can be manually loaded by means of the pushbuttons below the indicators when the computer is in Stopped Mode and Manual Mode.
NORMAL MODE indicator	The indicator lights when the computer is in Normal Mode. The light is out in Technical Mode.

RUNNING MODE indicator	The indicator lights when the computer is in Running Mode. The light is out in Stopped Mode.
MANUAL MODE indicator	The indicator lights when the computer is in Manual Mode. The light is out in Computer Mode.
MONITOR MODE indicator	The indicator lights when the computer is in Monitor Mode. The light is out in Task Mode.
MPS CONTROL ON pushbutton	When this pushbutton is pressed, every word read from the microprogram store is checked against parity error. Switching the Mode Selector key to Normal position causes the MPS control to be set.
MPS CONTROL OFF pushbutton	When this pushbutton is pressed, no parity check is performed on the microprogram store. The button has also the effect of erasing a possible MPS error.
MPS CONTROL indicator	The indicator lights when the microprogram is checked against parity error.
MPS ERROR indicator	In the event of a parity error caused by the microprogram store, the computer will stop at once and the indicator will light.
CORE STORE CONTROL ON pushbutton	When this pushbutton is pressed, every word read from the core store is checked against parity error. Switching the Mode Selector key to Normal position causes the Core Store Control to be set.
CORE STORE CONTROL OFF pushbutton	When this pushbutton is pressed, no parity check is performed on the core store. The button has also the effect of erasing a possible core store parity error.

CORE STORE CONTROL indicator

The indicator lights when the core store is checked against parity error.

CORE STORE ERROR indicator

In the event of a parity error caused by the core store, the computer will stop at once and the indicator will light.

REGISTER DISPLAY indicator

This light signifies that the REGISTER DISPLAY field functions normally, and it is turned on when the computer is set in Stopped Mode.

Now the REGISTER DISPLAY indicator lights on the Control Panel can be used to display and alter the contents of all the registers in the arithmetic unit. The contents of a register is displayed in binary form by means of an array of indicators. Each indicator is controlled by two pushbuttons, one for setting the bit to 1, and one for resetting to 0. Besides this, two pushbuttons are used for collective set and reset of the whole register. The name of the register to be displayed is indicated in the top row of this field, and the function of the pushbuttons below is to connect one of the below mentioned registers to the display.

W0	pushbutton	W(0)(0:23)conPB(0)(0:2)
W1	pushbutton	W(1)(0:23)conPB(1)(0:2)
W2	pushbutton	W(2)(0:23)conPB(2)(0:2)
W3	pushbutton	W(3)(0:23)conPB(3)(0:2)
IC	pushbutton	IC(5:22)
SC	pushbutton	SC(11:23)
FR	pushbutton	FR(0:11)
SB	pushbutton	SB(0:23)conPK(0:2)
SE	pushbutton	SE(0:13)
AR	pushbutton	AR(-1:23)
AE	pushbutton	AE(0:13)
BR	pushbutton	BR(0:23)

BE	pushbutton	BE(0:11)	
PR	pushbutton	PR(0:7)	x)
EX	pushbutton	EX(21:23)	
IR	pushbutton	IR(0:23)	; display only
IM	pushbutton	IM(0:23)	; display only

x) PR(0:7) corresponds to the bit position numbers 16 to 23 in the display.

2.2. Alter and Display Registers

1. Set the computer into Stopped Mode, which is done by depressing any of the following pushbuttons:

   SINGLE INSTRUCTION,
   SINGLE MICRO INSTRUCTION,
   MAR MANUAL CONTROLLED, or
   MAR COMPUTER CONTROLLED.

2. Now the REGISTER DISPLAY lights and the chosen register is displayed, and its contents can be altered.

2.3. Execute Program Instruction by Instruction

A program may be examined in details by executing it instruction by instruction in the following manner:

1. Depress SINGLE INSTRUCTION.

2. Insert the program start address in IC.

3. Depress MAR MANUAL CONTROLLED and insert in the Micro Address Register x=4 and y=0.

4. Depress MAR COMPUTER CONTROLLED.

5. Depress SINGLE INSTRUCTION, which causes the first instruction to be fetched and loaded into FR and SB. After the first instruction has been fetched, each time the SINGLE INSTRUCTION pushbutton is operated, the previously fetched instruction is executed and the next instruction is fetched.

2.4. Execute the Microprogram Micro Instruction by Micro Instruction

The microprogram may be examined in details by executing it micro instruction by micro instruction in the following manner:

1. Depress SINGLE INSTRUCTION.

2. Depress MAR MANUAL CONTROLLED and insert the wanted start address in the Micro Address Register.

3. Depress MAR COMPUTER CONTROLLED.

4. For each time the SINGLE MICRO INSTRUCTION pushbutton is operated, one micro instruction is executed.

2.5. Read a Word from Storage

Any 27-bit word can be read out from storage as follows:

1. Depress SINGLE INSTRUCTION and insert in SB the wanted byte address.

2. Depress MAR MANUAL CONTROLLED and insert in the Micro Address Register x=1 and y=0.

3. Depress SINGLE MICRO INSTRUCTION and the 27-bit word is loaded into SB and PK from where it can be displayed.

4. Before returning to the normal mode of operation, depress MAR COMPUTER CONTROLLED.

2.6. Insert or Modify a Word in Storage

Any 27-bit word in storage can be altered as follows:

1. Depress SINGLE INSTRUCTION and insert in SB the wanted byte address.

2. Depress MAR MANUAL CONTROLLED and insert in the Micro Address Register x=0 and y=30.

3. Depress SINGLE MICRO INSTRUCTION and the selected 27-bit word is read into SB and PK. SB and PK should now be modified to the word we want to store.

4. Insert in the Micro Address Register x=16 and y=8.

5. Depress SINGLE MICRO INSTRUCTION and the contents of SB and PK are stored.

6. Before returning to the normal mode of operation, depress MAR COMPUTER CONTROLLED.

2.7. Insert a Word in IM Register

1. Depress SINGLE INSTRUCTION and insert in SB the word to be loaded into IM.

2. Depress MAR MANUAL CONTROLLED and insert in the Micro Address Register x=12 and y=12.

3. Depress SINGLE MICRO INSTRUCTION. The contents of SB are inserted into IM; remember, IM(0) is always 1.

4. Before returning to the Normal Mode of operation, depress MAR COMPUTER CONTROLLED.

2.8. Clear Selected Bits in IR register

1. Depress SINGLE INSTRUCTION and insert a 1 in the bits of SB, which corresponds to the bits to be cleared in IR.

2. Depress MAR MANUAL CONTROLLED and insert in the Micro Address Register x=12 and y=13.

3. Depress SINGLE MICRO INSTRUCTION and the selected bits of IR are cleared.

4. Before returning to the Normal Mode of operation, depress MAR COMPUTER CONTROLLED.

RCSL51: VB263
Author: Allan Giese
Edited: September 1968

THE MICROPROGRAM ALGORITHMS FOR THE RC 4000 COMPUTER

ABSTRACT.

This paper presents the algorithms which control the execution of the instruction set. A survey of the data formats and the register structure is also given.

A/S REGNECENTRALEN
Falkoneralle 1,
Copenhagen, F.

PREFACE

This paper presents for every instruction a detailed algorithm which corresponds to the actual microprogram. The purpose of this is threefold:

- It enables the programmer to determine the outcome of an instruction for all possible data combinations.

- It shows the arithmetic and logic formulae used in the implementation of the instructions. A verification of the formulae is not included.

- It serves as a guide for the understanding of the microprogram.

Before proceeding to the algorithms in Section 4, we should like to clarify the different data formats and the register structure. The data formats are found in Section 1, and Section 2 describes the register structure together with an indication of the tasks of the registers. Section 3 shows the instruction set and the numeric codes assigned to these instructions.

CONTENTS

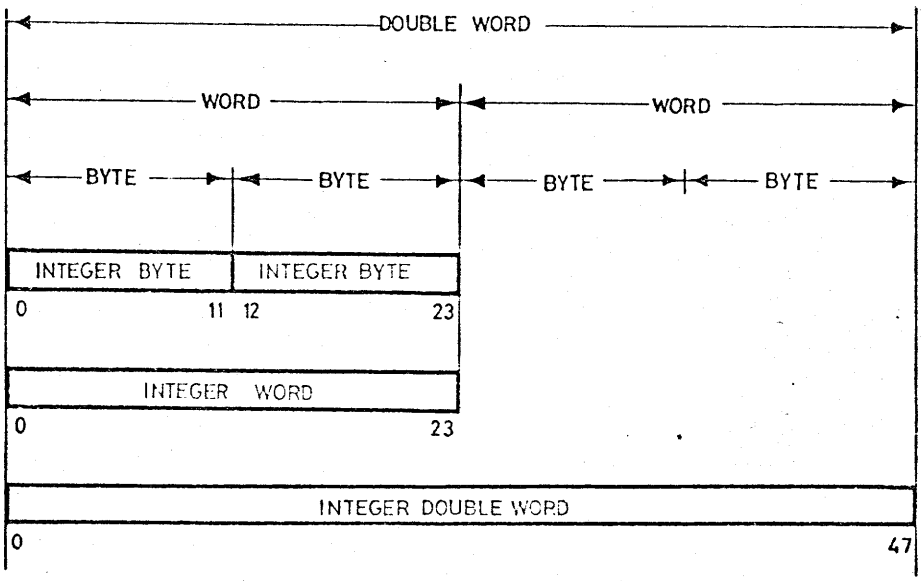
1. DATA FORMATS.

1.1. Fixed-Point Numbers.

The RC 4000 accepts three formats for fixed-point numbers:

1) 12-bit integers (bytes)
2) 24-bit integers (words)
3) 48-bit integers (double words).

They are stored as shown in the following figure:



1.2. Floating-Point Numbers.

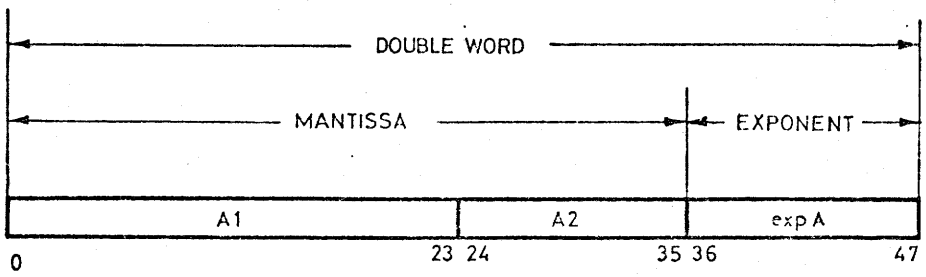
A floating-point number

$$X = A \times 2 \uparrow expA$$

consists of two portions, the mantissa A and the exponent expA. The mantissa is always in normalized form, which implies that A and expA are uniquely determined except when X equals zero. In this case it is perfectly clear that A must be zero, wheras expA may be any number. However, we have defined that expA for X = 0 should be the smallest conceivable exponent, that is in 2's complement notation 100...0. Using this notation, the following rule is always satisfied

$$X + 0 = X.$$

In the RC 4000, floating-point numbers are represented by double words of 48 bits and the mantissa is divided into two fields, A1 and A2, as seen below.



If an instruction operates on two floating-point numbers $X = A \times 2\uparrow expA$ and $Y = B \times 2\uparrow expB$, it is assumed that the numbers are stored as follows:

W[pre]: A1 ST[address-2]: B1
W[fr]: A2, expA ST[address] : B2, expB

2. REGISTER STRUCTURE.

The registers of the RC 4000 are grouped around a common bus line system - a configuration which has the advantage that all the registers can transfer data directly to one another. The figure, in the Appendix, shows not only the data paths between the registers of the Arithmetic Unit, but also the paths to the Core Store Unit, the Interrupt Unit, and the Input/-Output Unit. The direction of flow is indicated by means of arrows, and the bit numbers are written on the data paths.

Core Store Data Register: STdata(0:27).

Data communication between the Core Store Unit and the Arithmetic Unit takes place via STdata. The register is logically divided into three groups.

STdata(0:23)	Specifies a 24-bit dataword.
STdata(24:26)	Specifies the protection bits.
STdata(27)	Specifies the parity bit (Not shown in the Appendix).

Core Store Address Register; STaddr(6:22).

This register is able to address the maximum core store configuration of 128 K words. STaddr may therefore have fewer bits in actual installations.

Working Registers; W[0](0:23), W[1](0:23), W[2](0:23), W[3](0:23).

The four working registers, each of 24 bits, can be specified as the result register. Three of the registers (W[1], W[2], and W[3]), also function as index registers. The current index register is specified by the instruction format. Since the working registers also act as the first four locations of the core store, it is possible to execute instructions stored in these registers. Like the rest of the storage words each register is supplied with its own protection bits (PB).

Protection Bits; PB[0](0:2), PB[1](0:2), PB[2](0:2), PB[3](0:2).

The four 3-bit registers determine together with the protection register whether the corresponding working register is protected or not. For example, W[0] is protected if PR(PB[0]) equals one, otherwise W[0] is unprotected.

Protection Register; PR(0:7).

This 8-bit register specifies the protection status for the eight possible values of the protection bits. PR(0) is permanently equal to one.

Instruction Counter; IC(5:22).

The instruction counter contains the word address of the instruction to be executed. IC is normally increased by one after execution of an instruction, but jump instructions insert explicitly the jump address in IC. A decoding of IC detects when the storage capacity is exceeded.

If the operator executes one instruction at a time by using the SINGLE INSTRUCTION pushbutton, he should be aware that IC shows the address of the instruction, which is going to be executed when he next time depresses the SINGLE INSTRUCTION button.

Sequential Counter; SC(11:23).

The 13-bit sequential counter is used to determine the number of iterations in, for example the multiply, divide, shift, and normalize instructions. For each iteration, SC is either increased or decreased by one.

In floating-point operations, SC is also used for temporary storage of the resultant exponent.

Function Register; FR(0:11).

When a new instruction is fetched from storage, the function part of the instruction, i.e. the twelve left-most bits of the instruction, is assigned to the function register. FR is divided into the following five subfields:

FR(0:5)	Specifies 64 basic instructions.
FR(6,7)	Specifies one of the working registers as result register.
FR(8)	Indirect addressing.
FR(9)	Relative addressing.
FR(10,11)	Indexed addressing.

Exception Register; EX(21:23).

An exceptional outcome of an arithmetic instruction or an input/output instruction has the effect of setting the two bits EX(22) and EX(23). Bit EX(21) specifies the significance mode for floating-point operations. The contents of EX can also be altered by means of the Exception Load instruction.

Storage Buffer; SB(0:23).

When a new instruction is fetched from storage, the address part (the displacement) of the instruction is assigned to the 12 right-most bits of SB. This displacement is then extended to a 24-bit signed integer. The address modifications take then place in SB in order to obtain the effective address for the data word. In consequence of, that the address for the data word is generated in SB, we have also established an address path between the core store address register (STaddr) and SB. Moreover, the 24-

bit data word, which is read from the core store, is transferred to SB just as a data word from the Arithmetic Unit to the core store also passes through SB. It is therefore no coincidence that the name for this register is storage buffer.

During the initial phase of an input/output instruction, the device address of 18 bits and the command code of 6 bits are held in SB, from which the address and the code are transmitted to the Input/Output Unit. Information from the Input/Output Unit, on the other hand, does not go via SB, but has its own direct entrance to the bus system.

Either the contents of SB or its complement may be employed as input data for the ADDER circuitry. This makes it possible for the ADDER to perform addition and subtraction.

Protect Key; PK(0:2).

Every time a word is read from storage, the protection bits are inserted in PK, and reversely, PK determines the protection bits to be stored.

Storage Buffer Extension; SE(0:13).

SE forms together with SB a 38-bit register which is used in floating-point operations to store the mantissa. The two extra bits SE(12,13) play an important role in the rounding calculations.

Either the contents of SE or its complement may be employed as input data for the extended ADDER circuitry.

Adder Circuitry; ADDER(-1:37).

The adder is a parallel adder utilizing extensive carry look-ahead technique. 25 bits are added in typically 120 nanoseconds and 39 bits in 140 nanoseconds.

A-Register; AR(-1:23).

AR constitutes together with SB the two data parts to the ADDER circuitry. Bit AR(-1) detects if an owerflow situation occurs.

A-Register Extension; AE(0:13).

The combined 38-bit register ARconAE is used in a manner very similar to that of SBconSE.

B-Register; BR(0:23).

BR is mainly used by multiply, divide, and double length instruction, whereas other instructions only use the register to store temporary results.

If an instruction requires two successive storage words, the address of the second word is held in BR while the first word is fetched from core store. An address path is therefore provided from BR to STaddr.

B-Register Extension; BE(0:11).

The BR register is extended in order to be able to execute the floating-point division instruction.

Display and Manual Control of Bus.

This unit consists of the two registers, DR(-1:23) and DP(0:2) which control BUS(-1:23) and PBUS(0:2), respectively.

Interrupt Register; IR(0:23).

Each bit in IR is connected to an external or an internal device, which sets the bit in accordance with some specified condition. The leftmost bits are assigned highest priority.

Interrupt Mask; IM(0:23).

IM determines whether a given interrupt request should be honoured or not. IM(0) is permanently equal to one.

Input/Output Unit.

Each input/output device has in principle a 24-bit buffer register plus the two status bits Disconnected and Busy.

3. INSTRUCTION SET.

This list gives the total instruction set. The numeric code and the mnemonic code are added to each instruction.

Address Handling

9	AM	Modify Next Address
11	AL	Load Address
33	AC	Load Address Complemented

Register Transfer

3	HL	Load Half Register
26	HS	Store Half Register
20	RL	Load Register
23	RS	Store Register
25	RX	Exchange Register and Store
54	DL	Load Double Register
55	DS	Store Double Register

Integer Byte Arithmetic

19	BZ	Load Byte with Zeroes
2	BL	Load Integer Byte
18	BA	Add Integer Byte
17	BS	Subtract Integer Byte

Integer Word Arithmetic

7	WA	Add Integer Word
8	WS	Subtract Integer Word
10	WM	Multiply Integer Word
24	WD	Divide Integer Word

Integer Double Word Arithmetic

56	AA	Add Integer Double Word
57	SS	Subtract Integer Double Word

Arithmetic Conversion

32	CI	Convert Integer to Floating
53	CF	Convert Floating to Integer

Floating-Point Arithmetic

48	FA	Add Floating
49	FS	Subtract Floating
50	FM	Multiply Floating
52	FD	Divide Floating

Logical Operations

4	LA	Logical And
5	LO	Logical Or
6	LX	Logical Exclusive Or

Shift Operations

36	AS	Shift Single Arithmetically
37	AD	Shift Double Arithmetically
38	LS	Shift Single Logically
39	LD	Shift Double Logically
34	NS	Normalize Single
35	ND	Normalize Double

Sequencing

13	JL	Jump with Register Link
40	SH	Skip if Register High
41	SL	Skip if Register Low
42	SE	Skip if Register Equal
43	SN	Skip if Register Not Equal
44	SO	Skip if Register Bits One
45	SZ	Skip if Register Bits Zero
46	SX	Skip if No Exceptions
21	SP	Skip if No Protection

Monitor Control

15	JE	Jump with Interrupt Enable
14	JD	Jump with Interrupt Disable
47	IC	Clear Interrupt Bits
31	IS	Store Interrupt Register
12	ML	Load Mask Register
30	MS	Store Mask Register
16	XL	Load Exception Register
27	XS	Store Exception Register
28	PL	Load Protection Register
29	PS	Store Protection Register
22	KL	Load Protection Key
51	KS	Store Protection Key
1	IO	Input/Output
0	AW	Autoload Word

4. DEFINITION OF INSTRUCTIONS.

4.1. Notation.

The Hargol language is used throughout this section for describing the instruction logic. Each quantity, used in the following explanation, must therefore be properly defined by a declaration. These declarations, which are listed below, are grouped for ease of interpretation.

```
Core Store Unit:
  integer word limit;
  register array ST[4:word limit](0:26);
  register STdata(0:26), STaddr(6:22);
  comment For simplicity, the parity bit is omitted;

Arithmetic Unit:
  register array W[0:3](0:23), PB[0:3](0:2);
  register
    PR(0:7), PK(0:2), IC(5:22), SC(11:23), FR(0:11), EX(21:23),
    SB(0:23), SE(0:13), AR(-1:23), AE(0:13), BR(0:23), BE(0:11);

Interrupt Unit:
  register IR(0:23), IM(0:23);

Input/Output Unit:
  integer Selected Device, maximum number of devices;
  register array
    Device Buffer[0:maximum number of devices](0:23),
    Disconnected[0:maximum number of devices](0:0),
    Busy[0:maximum number of devices](0:0);

Abbreviations:
  register set
    SBa(-1:23) = SB(0,0:23), BRa(-1:23) = BR(0,0:23),
    SF(0:37) = SB(0:23)conSE(0:13), SFa(-1:37) = SF(0,0:37),
    AF(-1:37) = AR(-1:23)conAE(0:13),
    BF(0:35) = BR(0:23)conBE(0:11), BFa(-1:35) = BF(0,0:35),
```

```
    IRa(-1:23) = IR(0,0:23),
    IMa(-1:23) = IM(0,0:23),
    ICaddr(6:22) = IC(6:22),
    SBaddr(6:22) = SB(6:22),
    BRaddr(6:22) = BR(6:22);
  register array set
    Wa[0:3](-1:23) = W[0:3](0,0:23);
  integer procedure fr;
    fr:= FR(6,7);
  integer procedure pre;
    pre:= if FR(6,7) = 0 then 3 else FR(6,7)-1;
  integer procedure index;
    index:= FR(10,11);
```

Control Unit:

```
  register
    MMode(0:0), ITRenable(0:0), HA(23:23), SUM(-1:23), Carry(0:0),
    Main Power Key ON(0:0),
    Reset(0:0), Start(0:0), Autoload(0:0),
    Single Instruction(0:0), Continue(0:0);
  comb net PROTECT(0:0);
    begin PR(0):= 1; PROTECT:= PR(PK) end;
  switch operation:= Modify Next Address, ---, Autoload Word;
```

4.2. Control Panel Functions.

Reset System: Power Shutdown:

```
  PK:= 0; MMode:= PROTECT;
  comment MMode:= 1, since PR(0) = 1. Monitor Mode is set in order to secure
    the writing into the core store;
  SB:= 23; ashr SF; comment SBaddr:= 10;
  wait until Accept; STaddr:= SBaddr;
  SBconPK:= ST[STaddr];
  SB:= 5ext0con1Ccon0;
  ST[STaddr]:= SBconPK; comment Store IC;
  wait until Main Power Key ON ∧ -,Reset;
```

```
Power Startup:
  PK:= 0; MMode:= PROTECT;
  FR:= 0; ITRenable:= FR(5); comment ITRenable:= 0;

After Reset:
  if Autoload then
    begin
      SB:= 0; IC:= 0; goto Autoload Word;
      comment This is equivalent to an execution of the instruction AW 0,
        stored in location 0;
    end;
  if Start then
    begin
      SB:= 12 ∨ 2; comment SB:= 14;
      wait until Accept; STaddr:= SBaddr;
      SBconPK:= ST[STaddr]; IC:= SB(5:22);
      comment IC:= address of start key program;
      goto Next Instruction
    end;
  goto After Reset;
```

4.3. Interruption Service.

```
Interruption Service:
  SB:= 12; BR:= 23; lshr BF; comment BRaddr:= 10;
  wait until Accept; STaddr:= SBaddr;
  PK:= 0; MMode:= PROTECT; FR:= 0; ITRenable:= FR(5);
  AR:= 6extOconICcon0; comment IC:= address of interruption program;
  SBconPK:= ST[STaddr]; IC:= SB(5:22);
  wait until Accept; STaddr:= BRaddr; SBconPK:= ST[STaddr];
  SB:= AR(0:23); ST[STaddr]:= SBconPK;
  comment ST[10](0:23):= address of interrupted program;
  SC:= 6; SC:= SC+2; SB:= 12extOconSC(12:23);
  wait until Accept; STaddr:= SBaddr; SBconPK:= ST[STaddr];
    begin
      integer ITRno;
      for ITRno:= 0, ITRno+1 while (IR(ITRno) ∧ IM(ITRno)) = 0 do;
      IR(ITRno):= 0; SB:= ITRno×2;
```

```
      comment Clear the interruption call having the highest priority;
    end;
  ST[STaddr]:= SBconPK; comment ST[8](0:23):= interrupt number;
```

4.4. Instruction Fetch Cycle.

```
Next Instruction:
  if or(IR ∧ IM) ∧ ITRenable then goto Interruption Service;
  wait until Accept; STaddr:= ICaddr;
  AR:= 6ext0conICcon0; comment Save relative address;
  if -,(Main Power Key ON ∧ -,Reset) then goto Reset System;
  if IC > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr]; FR:= ST[STaddr](0:11);
  SB(0:11):= 12extSB(12); IC:= IC+1;
  if -,FR(8) then AR:= Wa[index]; comment Speed up mechanism for indexing;
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  MMode:= PROTECT;
  if Single Instruction then wait until Continue;

Address Modifications:
  if FR(8) = 1 ∧ FR(10,11) = 0 ∨ FR(8) = 0 ∧ FR(10,11) ≠ 0 then
    begin comment Relative address or indexing;
      SB:= AR+SBa
    end;
  if FR(8) = 1 ∧ FR(10,11) ≠ 0 then
    begin comment Relative address and indexing;
      SB:= AR+SBa; AR:= Wa[index]; SB:= AR+SBa
    end;
  if FR(9) = 1 then
    begin comment Indirect address;
      wait until Accept; STaddr:= SBaddr;
      if SB(0:22) > word limit then goto Instruction Exception;
      SBconPK:= ST[STaddr]
    end;
  comment SB contains the effective address and IC points at the next in-
    struction;
  goto operation [FR(0:5)];
```

4.5. Instruction Exceptions.

Instruction Exception:

```
IR(0):= 1; goto Interruption Service;

procedure Test Integer;
  begin
    if SUM(-1) ≠ SUM(0) then begin EX(22):= 1; IR(1):= 1 end;
    if Carry(0) then EX(23):= 1
  end;

procedure Test Shift;
  begin
    if AR(0) ≠ AR(1) then begin EX(22):= 1; IR(1):= 1 end;
  end;

procedure Test Exp;
  begin
    if SC(11) ≠ SC(12) then begin EX(22):= 1; IR(2):= 1 end;
  end;
```

4.6. Instruction Execution.

Modify Next Address:

Use the effective address as an increment to the displacement in the next instruction. The operation changes only the effective address of the next instruction whose D field remains unchanged.

```
begin
  AR:= SBa;
  comment The effective address of the am instruction is saved in AR;
  wait until Accept; STaddr:= ICaddr;
  BR:= 5extOconICconO; comment Save relative address;
  if IC > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr]; FR:= ST[STaddr](0:11);
  SB(0:11):= 12extSB(12); IC:= IC+1;
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  MMode:= PROTECT;
  if Single Instruction then wait until Continue;
```

```
    SB:= AR+SBa;
    if FR(8) = 0 ∧ FR(10,11) ≠ 0 then AR:= Wa[index];
    if FR(8) = 1 then AR:= BRa;
    goto Address Modifications
  end am 9;
```

Load Address:

Load the W register with the effective address.

```
  begin W[fr]:= SB; goto Next Instruction end al 11;
```

Load Address Complemented:

Load the W register with the two's complement of the effective address. Complementation of the maximum negative number -2↑23 gives the result -2↑23 and produces an overflow.

```
  begin
    AR:= 0; EX(22,23):= 0; SUM:= AR-SBa; W[fr]:= SUM(0:23);
    Test Integer; goto Next Instruction
  end ac 33;
```

Load Half Register:

Insert the storage byte addressed in the right-most 12 bits of the W register without changing the left-most 12 bits. The storage byte remains unchanged.

```
  begin
    wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    W[fr](12:23):= if HA(23) then SB(12:23) else SB(0:11);
    goto Next Instruction
  end hl 3;
```

Store Half Register:

Store the right-most 12 bits of the W register in the storage byte addressed. The register remains unchanged.

```
  begin
    AR:= Wa[fr];
    wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
    if SB(0:22) > word limit then goto Instruction Exception;
```

```
    SBconPK:= ST[STaddr];
    if -,(MMode ∨ -,PROTECT) then goto Instruction Exception
    if HA(23) then SB(12:23):= AR(12:23) else SB(0:11):= AR(12:23);
    ST[STaddr]:= SBconPK;
    goto Next Instruction
  end hs 26;
```

Load Register:

Load the W register with the storage word addressed. The storage word remains unchanged.

```
  begin
    wait until Accept; STaddr:= SBaddr;
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    W[fr]:= SB; goto Next Instruction
  end rl 20;
```

Store Register:

Store the W register in the storage word addressed. The register remains unchanged.

```
  begin
    AR:= Wa[fr];
    wait until Accept; STaddr:= SBaddr;
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
    SB:= AR(0:23); ST[STaddr]:= SBconPK;
    goto Next Instruction
  end rs 23;
```

Exchange Register and Store:

The W register is stored in the storage word addressed and the previous contents of the storage word is loaded into the register.

```
  begin
    AR:= Wa[fr];
    wait until Accept; STaddr:= SBaddr;
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr]; W[fr]:= SB;
```

```
    if -,(MMode ∨ -,PROTECT) then
      begin W[fr]:= AR(0:23); goto Instruction Exception end;
    SB:= AR(0:23); ST[STaddr]:= SBconPK;
    goto Next Instruction
  end rx 25;
```

Load Double Register:

Load the register pair W and Wpre with the storage double word addressed. The storage double word remains unchanged.

```
  begin
    AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
    wait until Accept; STaddr:= SBaddr;
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr]; W[fr]:= SB;
    wait until Accept; STaddr:= BRaddr;
    SBconPK:= ST[STaddr];
    W[pre]:= SB; goto Next Instruction
  end dl 54;
```

Store Double Register:

Store the register pair W and Wpre in the storage double word addressed. The register pair remains unchanged.

```
  begin
    AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
    wait until Accept; STaddr:= SBaddr;
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
    SB:= W[fr]; ST[STaddr]:= SBconPK;
    SB:= BR; AR:= Wa[pre];
    wait until Accept; STaddr:= SBaddr;
    SBconPK:= ST[STaddr];
    if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
    SB:= AR(0:23); ST[STaddr]:= SBconPK;
    goto Next Instruction
  end ds 55;
```

Load Byte with Zeroes:

Insert the storage byte addressed in the right-most 12 bits of the W register and extend it towards the extreme left with zeroes. The storage byte remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if HA(23) then begin SB(0:11):= 0; W[fr]:= SB end
    else W[fr]:= 12ext0conSB(0:11);
  goto Next Instruction
end bz 19;
```

Load Integer Byte:

Insert the storage byte addressed in the right-most 12 bits of the W register and extend the sign bit towards the extreme left. The storage byte remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if HA(23) then SB(0:11):= 12extSB(12) else SB:= 12extSB(0)conSB(0:11);
  W[fr]:= SB; goto Next Instruction
end bl 2;
```

Add Integer Byte:

The storage byte addressed is extended towards the left to 24 bits and added to the W register. The sum is placed in the register. The storage byte remains unchanged.

```
begin
  EX(22,23):= 0; wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if HA(23) then SB(0:11):= 12extSB(12) else SB:= 12extSB(0)conSB(0:11);
  SUM:= AR+SBa; W[fr]:= SUM(0:23);
  Test Integer; goto Next Instruction
end ba 18;
```

Subtract Integer Byte:

The storage byte addressed is extended towards the left to 24 bits and subtracted from the W register. The difference is placed in the register. the storage byte remains unchanged.

```
begin
  EX(22,23):= 0; wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if HA(23) then SB(0:11):= 12extSB(12) else SB:= 12extSB(0)conSB(0:11);
  SUM:= AR-SBa; W[fr]:= SUM(0:23);
  Test Integer; goto Next Instruction
end bs 17;
```

Add Integer Word:

The storage word addressed is added to the W register, and the sum is placed in the register. The storage word remains unchanged.

```
begin
  EX(22,23):= 0; wait until Accept; STaddr:= SBaddr;
  AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  SUM:= AR+SBa; W[fr]:= SUM(0:23);
  Test Integer; goto Next Instruction
end wa 7;
```

Subtract Integer Word:

The storage word addressed is subtracted from the W register, and the difference is placed in the register. The storage word remains unchanged.

```
begin
  EX(22,23):= 0; wait until Accept; STaddr:= SBaddr;
  AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  SUM:= AR-SBa; W[fr]:= SUM(0:23);
  Test Integer; goto Next Instruction
end ws 8;
```

Multiply Integer Word:

The W register is multiplied by the storage word addressed. The 48-bit signed product is placed in the register pair Wpre and W. Overflow cannot occur.

```
begin
  comment The algorithm starts by setting up the multiplicand in SB and
    the multiplier in BR. SC determines the number of iterations. The
    final 48-bit result appears, after termination of the procedure, in
    the concatenated register ARconBR;
  SC:= 23; wait until Accept; STaddr:= SBaddr; BR:= W[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr]; AR:= 0;
  for SC:= SC-1 step -1 until 0 do
    if BR(23) = 1 then AR:= (AR+SBa)/2 else ashr ARconBR;
  if BR(23) = 1 then AR:= AR-SBa; ashr ARconBR;
  W[fr]:= BR; W[pre]:= AR(0:23); goto Next Instruction
end wm 10;
```

Divide Integer Word:

The register pair Wpre and W is divided by the storage word addressed. The 24-bit signed quotient is placed in the W register, while the 24-bit signed remainder is placed in the preceeding register Wpre. A non-zero remainder has the same sign as the dividend. An overflow is registered if the divisor is zero or if the quotient exceeds 24 bits. In this case the dividend remains unchanged in the working registers.

```
begin
  comment The algorithm starts by setting up the divisor in SB and the
    dividend in ARconBR. SC determines the number of iterations;
  register WDsub(0:0);
  EX(22,23):= 0; SC:= 23;
  wait until Accept; STaddr:= SBaddr; BR:= W[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr]; AR:= Wa[pre]; BE:= Wa[pre](0:11);
  comment The quotient digits are now calculated by means of the non-re-
    storing method. The very first bit is calculated by its complemented
    value;
  WDsub:= if AR(-1) = SB(0) then 1 else 0; lshl ARconBR;
```

```
for SC:= SC step -1 until 0 do
  begin
    AR:= if WDsub = 1 then AR-SBa else AR+SBa; BR(23):= WDsub;
    WDsub:= if AR(-1) = SB(0) then 1 else 0; lshl ARconBR
  end;
comment If BR(23) was equal to one instead of zero, then
  AR(23)conBR(0:23) equals the 25-bit quotient belonging to the re-
  mainder stored in AR(-1:22). The quotient exceeds the register ca-
  pacity if AR(23) = BR(0);
if AR(23) = BR(0) then
  begin
    SB(0:11):= -2↑11; AR:= SBa; Test Shift; goto Next Instruction
  end;
ashr AF;
comment The remainder is now corrected so that
                    1) The remainder and the dividend have equal sign.
                    2) -divisor < remainder < divisor.
  The sign bit for the dividend is BE(0);
if -,AR(-1) ∧ -,SB(0) ∧ -,BE(0) then goto EXIT1;
if -,AR(-1) ∧ -,SB(0) ∧ BE(0) then goto if AR ≠ 0 then EXIT2 else EXIT1;
if -,AR(-1) ∧ SB(0) ∧ -,BE(0) then goto EXIT1;
if -,AR(-1) ∧ SB(0) ∧ -,BE(0) then goto if AR ≠ 0 then EXIT3 else EXIT1;
if AR(-1) ∧ -,SB(0) ∧ -,BE(0) then goto EXIT3;
if AR(-1) ∧ -,SB(0) ∧ BE(0) then
  begin
    SB:= AR+SBa; if SB ≠ 0 then goto EXIT1;
    W[pre]:= 0; W[fr]:= BR; goto Next Instruction
  end;
if AR(-1) ∧ SB(0) ∧ -,BE(0) then goto EXIT2;
if AR(-1) ∧ SB(0) ∧ BE(0) then
  begin
    SB:= AR-SBa; if SB ≠ 0 then goto EXIT1;
    AR:= BR+1; AR:= AR+1;
    if AR(-1) ≠ AR(0) then
      begin comment The quotient exceeds the register capacity;
        ashr AF; Test Shift; goto Next Instruction
      end;
    W[pre]:= 0; W[fr]:= AR(0:23); goto Next Instruction
  end;
EXIT1: W[pre]:= AR(0:23); W[fr]:= BR+1; goto Next Instruction;
EXIT2: W[pre]:= AR-SBa; AR:= BR+1; W[fr]:= AR+1; goto Next Instruction;
```

```
    EXIT3: W[pre]:= AR+SBa; W[fr]:= BR; goto Next Instruction
  end wd 24;
```

Add Integer Double Word:

The storage double word addressed is added to the register pair Wpre and W as an integer double word,nd the sum is placed in the register pair. The storage double word remains unchanged.

```
  begin
    EX(22,23):= 0; AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
    wait until Accept; STaddr:= SBaddr; AR:= Wa[fr];
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    SUM:= AR+SBa; W[fr]:= SUM(0:23);
    wait until Accept; STaddr:= BRaddr;
    AR:= Wa[pre];
    SBconPK:= ST[STaddr];
    SUM:= if Carry(0) then AR+SBa+1 else AR+SBa; W[pre]:= SUM(0:23);
    Test Integer; goto Next Instruction
  end aa 56;
```

Subtract Integer Double Word:

The storage double word addressed is subtracted from the register pair Wpre and W as an integer double word, and the difference is placed in the register pair. The storage double word remains unchanged.

```
  begin
    EX(22,23):= 0; AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
    wait until Accept; STaddr:= SBaddr; AR:= Wa[fr];
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    SUM:= AR-SBa; W[fr]:= SUM(0:23);
    wait until AcceptSB; STaddr:= BRaddr;
    AR:= Wa[pre];
    SBconPK:= ST[STaddr];
    SUM:= if Carry(0) then AR-SBa else AR-SBa-1; W[pre]:= SUM(0:23);
    Test Integer; goto Next Instruction
  end ss 57;
```

Convert Integer to Floating:

Convert the contents of the W register, interpreted as an integer multiplied by 2↑(effective address) to a floating-point number. The result is placed in the register pair Wpre and W.

```
begin
  comment The exponent belonging to the floating-point number,
    expR = 23 - number of shifts + effective address;
  EX(22,23):= 0; AR:= Wa[fr]; W[pre]:= W[fr];
  W[fr](0:11):= 0; AE:= 0; SC:= 23;
  if AR = 0 then begin W[fr](12:23):= -2↑11; goto Next Instruction end;
  if AR(0) = AR(1) then
    begin
      for SC:= SC, SC-1 while AR(0) = AR(1) do lshl AF; W[pre]:= AR(0:23)
    end;
  comment The mantissa is normalized and SC = 23 - number of shifts;
  if SB ≠ 0 then
    begin AR:= 13extOconSC(12:23); SC:= AR+SBa; Test Exp end;
  W[fr](12:23):= SC(12:23); goto Next Instruction
end ci 32;
```

Convert Floating to Integer:

Convert the contents of the register pair Wpre and W, interpreted as a floating-point number multiplied by 2↑(effective address) to an integer number. The result is placed in the W register. Wpre remains unchanged. An overflow is registered if the integer number exceeds 24 bits.

```
begin
  comment The exponent belonging to a floating-point number where the
    location of the binary point is moved 23 places to the right is
    expA-23. The algorithm starts by evaluating the number of right
    shifts, which equals 23 - effective address - expA;
  EX(22,23):= 0; AR:= 23; if SB ≠ 0 then AR:= AR-SBa;
  SB:= 12extW[fr](12)conW[fr](12:23); SB:= AR-SBa; SC:= SB(11:23);
  comment SB and SC equals the number of right shifts;
```

```
if SB = 0 then
  begin
    BE:= W[fr](0:11); AR:= Wa[pre];
    if BE(0) = 0 then
      begin comment No rounding is required;
        W[fr]:= AR(0:23); goto Next Instruction
      end;
    AR:= AR+1; comment Rounding;
    if AR(-1) = AR(0) then
      begin W[fr]:= AR(0:23); goto Next Instruction end;
      else goto INTEGER OVERFLOW
  end;
AR:= Wa[pre];
if SB < 64 ∧ SB(0) = 0 then
  begin comment 0 < right shifts < 64;
    SC:= SC-1; for SC:= SC-1 step -1 until 0 do ashr AF;
    AR:= (AR+1)/2; W[fr]:= AR(0:23); goto Next Instruction
  end;
if SB < 64 ∧ SB(0) = 1 then
  begin comment right shifts < 0;
    AE:= W[fr](0:11)con0con0;
    if AF = 0 then
      begin W[fr]:= AR(0:23); goto Next Instruction end;
      else goto INTEGER OVERFLOW
  end;
if SB ≥ 64 ∧ SB(0) = 0 then
  begin comment right shifts ≥ 64;
    W[fr]:= 0; goto Next Instruction
  end;
INTEGER OVERFLOW:
  SB(0:11):= -2↑11; AR:= SBa; Test Shift
end cf 53;
```

Add Floating:

The storage double word addressed is added to the register pair Wpre and W as a floating-point number, and the sum is placed in the register pair. The storage double word remains unchanged.

Subtract Floating:

Same as Add Floating except that the difference is calculated instead of the sum.

```
begin
  comment Set up for comparison of expA and expB, and store the exponent
    difference in SC;
  AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
  wait until Accept; STaddr:= SBaddr;
  AR:= 13extW[fr](12)conW[fr](12:23); EX(22,23):= 0;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  SE:= SB(0:11)con0con0; SB(0:11):= 12extSB(12);
  W[fr](12:23):= SB(12:23);
  SC:= AR-SBa;
  wait until Accept; STaddr:= BRaddr; AE:= W[fr](0:11)con0con0;
  SBconPK:= ST[STaddr];
  comment  W[pre]: A1              W[fr]: A2, expB
              SB: B1                   SE: B2
              AR: expA                 AE: A2
              SC: expA - expB
    The algorithm proceeds now in five ramifications depending on the
    value of the exponent difference;
  if SC ≥ 38 then
    begin comment expA >> expB, i.e. R = X ± Y = A×2↑expA;
      W[fr](0:11):= if EX(21) = 0 then AE(0:11) else AE(0:9,9,9);
      W[fr](12:23):= AR(12:23); goto Next Instruction
    end;
  if SC > 0 ∧ SC < 38 then
    begin comment expA > expB, i.e. R = X±Y = (A±(expA-expB) ashr B)×2↑expA;
      W[fr](12:23):= AR(12:23); AR:= Wa[pre];
      for SC:= SC-1 step -1 until 0 do ashr SF; goto ADD SUB;
    end;
  if SC = 0 then
    begin comment expA = expB, i.e. R = X±Y = (A±B)×2↑expB;
      AR:= Wa[pre]; goto ADD SUB
    end;
```

```
if SC > -38 ∧ SC < 0 then
  begin comment expA < expB, i.e. R = X±Y = ((-(expA-expB)) ashr A±B)×2↑expB;
    AR:= Wa[pre]; for SC:= SC+1 step 1 until 0 do ashr AF;
    goto ADD SUB
  end;
if SC ≤ -38 then
  begin comment expA << expB, i.e. R = X±Y = ±B×2↑expB;
    if Subtract Floating then begin AF:= 0; goto ADD SUB end;
    W[pre]:= SB; AE:= SE(0:11)con0con0;
    W[fr](0:11):= if EX(21) = 0 then AE(0:11) else AE(0:9,9,9);
    goto Next Instruction
  end;

ADD SUB:
  if Add Floating then AF:= AF+SFa;
  if Subtract Floating then AF:= AF-SFa;
  SC:= W[fr](12,12:23);
  comment After addition or subtraction of the two mantissae A and B, the
    resultant mantissa R shall first be normalized and then rounded. The
    contents of the registers are
                        AF: R,          W[fr](12:23): expR,
                                                SC: expR;

NORMALIZATION:
  if AR(-1) ≠ AR(0) ∧ AF(35) = 0 then
    begin comment Mantissa overflow and no rounding required;
      ashr AF; SC:= SC+1; goto EXIT
    end;
  if AR(-1) ≠ AR(0) ∧ AF(35) = 1 then
    begin comment Mantissa overflow and rounding required;
      AF:= (AF+4)/2; SC:= SC+1; goto ROUNDING
    end;
  if AR(-1) = AR(0) ∧ AR(0) ≠ AR(1) ∧ AF(36) = 0 then
    begin comment Mantissa normalized and no rounding required;
      goto EXIT
    end;
```

```
if AR(-1) = AR(0) ∧ AR(0) ≠ AR(1) ∧ AF(36) = 1 then
  begin comment Mantissa normalized and rounding required;
    AF:= AF+4; AF(36,37):= 0; goto ROUNDING
  end;
if AR(-1) = AR(0) ∧ AR(0) = AR(1) ∧ AF(37) = 0 then
  begin comment Mantissa unnormalized or zero, but no rounding required;
    if AF ≠ 0 then
      begin
        for SC:= SC,SC-1 while AR(0) = AR(1) do lshl AF;
        goto EXIT
      end
    else
      begin
        SC:= SC-1; lshl AF;
        W[fr](0:11):= 0; W[fr](12:23):= 2↑11; W[pre]:= AR(0:23);
        goto Next Instruction
      end;
  end;
if AR(-1) = AR(0) ∧ AR(0) = AR(1) ∧ AF(37) = 1 then
  begin comment Mantissa unnormalized. Rounding is only required if one
          shift is sufficient to normalize the mantissa;
    if AR(1) ≠ AR(2) then
      begin comment Shift only once;
        lshl AF; SC:= SC-1; AF:= AF+4; AF(36,37):= 0; goto ROUNDING
      end;
    else
      begin for SC:= SC,SC-1 while AR(0) = AR(1) do lshl AF; goto EXIT end;
  end;

ROUNDING:
  begin comment The mantissa lies in one of the intervals
          1/2 < R ≤ 1  or  -1 < R ≤ -1/2
          SC: expR          AF: R;
  end;
if AR(-1) = AR(0) ∧ AR(0) ≠ AR(1) then goto EXIT;
if AR(-1) = AR(0) ∧ AR(0) = AR(1) then
  begin SC:= SC-1; lshl AF; goto EXIT end;
if AR(-1) ≠ AR(0) then
  begin SC:= SC+1; ashr AF; goto EXIT end;
```

```
EXIT:
  W[fr](0:11):= if EX(21) = 0 then AE(0:11) else AE(0:9,9,9);
  W[fr](12:23):= SC(12:23); Test Exp; W[pre]:= AR(0:23);
  goto Next Instruction
end fa 48, fs 49;
```

Multiply Floating:

The register pair Wpre and W is multiplied by the storage double word addressed as a floating-point number, and the product is placed in the register pair. The storage double word remains unchanged.

```
begin
  comment The result, R, of the multiplication equals Y×X = B×A×2↑(expB+expA).
    The algorithm starts by calculating expB+expA and by setting up the man-
    tissa A and B in BF and SF, respectively. SC determines the number of
    iterations;
  SC:= 35; BE:= W[fr](0:11);
  AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
  wait until Accept; STaddr:= SBaddr;
  AR:= 13extW[fr](12)conW[fr](12:23); EX(22,23):= 0;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  SE:= SB(0:11)con0con0; SB(0:11):= 12extSB(12);
  W[fr]:= AR+SBa; comment W[fr] = expB+expA;
  AF:= 0; wait until Accept; STaddr:= BRaddr; BR:= W[pre];
  SBconPK:= ST[STaddr]; comment SF = B, BF = A, and AF = 0;
  for SC:= SC-1 step -1 until 0 do
    begin if BE(11) = 1 then AF:= (AF+SFa)/2 else ashr AF; lshr BF end;
  if BE(11) = 1 then AF:= AF-SFa;
  SC:= W[fr](11:23); goto NORMALIZATION; comment See Add Floating;
end fm 50;
```

Divide Floating:

The register pair Wpre and W is divided by the storage word addressed as a floating-point number, and the quotient is placed in the register pair. The storage double word remains unchanged.

```
begin
  comment The result, R, of the division equals X/Y = A/B×2↑(expA-expB).
    The algorithm starts by calculating expA - expB and by setting up the
    mantissa A and B in AF and SF, respectively. The quotient is calcu-
    lated by means of the non-restoring method;
  register FDsub(0:0);
  AR:= -2; BR:= if SB ≠ 0 then AR+SBa else 6;
  wait until Accept; STaddr:= SBaddr;
  AR:= 13extW[fr](12)conW[fr](12:23); EX(22,23):= 0;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  SE:= SB(0:11)con0con0; SB(0:11):= 12extSB(12); SB:= AR-SBa; AR:= 35;
  SC:= AR+SBa; comment SC = expA-expB+35;
  AR:= Wa[pre]; wait until Accept; STaddr:= SBaddr;
  AE:= W[fr](0:11)con0con0;
  SBconPK:= ST[STaddr];
  comment SF:= B and AF:= A;

  if AR = 0 then
    begin
      SC(11):= 0; SC(12:23):= -2↑11;
      if SB ≠ 0 then begin comment Zero result; W[fr]:= 12ext0conSC(12:23) end;
                else begin comment Irrelevant result; Test Exp end;
      goto Next Instruction
    end;
  SC:= SC+1; FDsub:= if AF(-1) = SF(0) then 1 else 0;
  comment FDsub = 1 when the signs of the divisor and the dividend are alike;
  if FDsub = 1 then AF:= (AF-SFa)×2 else AF:= (AF+SFa)×2;
  BF:= if FDsub = 1 then 0 else o 7777 7777 7777;
  if SB = 0 then
    begin
      comment Irrelevant result; SC(11):= 0; SC(12:23):= -2↑11;
      Test Exp; goto Next Instruction
    end;
  FDsub:= if AF(-1) = SF(0) then 1 else 0;
```

```
    for SC:= SC, SC-1 while BR(0) = BR(1) do
      begin
        comment This iteration proceeds until a normalized quotient is ob-
          tained. If the quotient, Q, is in the interval -1 ≤ Q < - 1/2 or
          1/2 ≤ Q < 1, the number of iterations equal 36 and SC = expA-expB.
          If -2 ≤ Q < -1 or 1 ≤ Q < 2, the number of iterations equal 35
          and SC = expA - expB + 1. If Q = -1/2, the number of iterations
          equal 37 and SC = expA-expB-1;

        lshl BF; BF(35):= FDsub; if FDsub = 1 then AF:= AF-SFa else AF:= AF+SFa;
        FDsub:= if AF(-1) = SB(0) then 1 else 0; lshl AF
      end;
    AE:= BEcon0con0;
    if FDsub = 1 then
      begin
        AR:= BRa; AF:= AF+4; AF(36,37):= 0; goto ROUNDING; comment See Add
                                                              Floating;
      end;
    W[fr]:= if EX(21) then AE(0:9,9,9)conSC(12:23) else AE(0:11)conSC(12:23);
    Test Exp; W[pre]:= BR; goto Next Instruction
  end fd 52;
```

Logical And:

The W register is combined with the storage word addressed by a logical And operation. The result is placed in the register. The storage word remains unchanged.

```
  begin
    wait until Accept; STaddr:= SBaddr; AR:= Wa[fr];
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    W[fr]:= AR(0:23) ∧ SB; goto Next Instruction
  end la 4;
```

Logical Or:

The W register is combined with the storage word addressed by a logical Or operation. The result is placed in the register. The storage word remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr; AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  W[fr]:= AR(0:23) ∧ SB; goto Next Instruction
end lo 5;
```

Logical Exclusive Or:

The W register is combined with the storage word addressed by a logical Exclusive Or operation, and the result is placed in the register. The storage word remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr; AR:= Wa[fr];
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  comment The Exclusive Or operation is carried out by means of the log-
    ical operators: And, Or, and Negation. The algorithm used is
    a exor b:= (a ∨ b) ∧ (-,a ∨ -,b);
  BR:= SB ∨ W[fr];          comment Supposing SB = b and W = a then
                                    BR = a ∨ b and AR = a;
  SB:= AR(0:23) ∧ SB;       comment SB = a ∧ b;
  AR:= BRa;                 comment AR = a ∨ b;
  W[fr]:= AR(0:23) ∧ -,SB; comment  W = (a ∨ b) ∧ -,(a ∧ b);
  goto Next Instruction
end lx 6;
```

Shift Single Arithmetically:

Shift the contents of the W register the number of places specified by the effective address in SB. If SB is negative, then shift right with sign extension in the upper bits; otherwise shift left with zero extension in the lower bits. Overflow is tested for each single shift.

```
begin
  EX(22,23):= 0;
  if SB = 0 then goto Next Instruction;
```

```
    SC:= SB(11:23); AR:= Wa[fr];
    if SB(0) = 0 then
      begin comment address > 0;
        BR:= 0; if SB ≥ 64 then SC:= 35;
        for SC:= SC-1 step -1 until 0 do
          begin Test Shift; lshl ARconBR end;
        W[fr]:= AR(0:23);
      end;
    if SB(0) = 1 then
      begin comment address < 0;
        if SB ≤ -65 ∧ AR(-1) = 0 then W[fr]:= 0;
        if SB ≤ -65 ∧ AR(-1) = 1 then W[fr]:= -1;
        if SB > -65 then
          begin
            for SC:= SC+1 step 1 until 0 do ashr ARconBR; W[fr]:= AR(0:23)
          end;
      end;
    goto Next Instruction
  end as 36;
```

Shift Double Arithmetically:

Shift the contents of the register pair Wpre and W the number of places specified by the effective address in SB. If SB is negative, then shift right with sign extension in the upper bits; otherwise shift left with zero extension in the lowest bits. Overflow is tested for each single shift.

```
  begin
    EX(22,23):= 0;
    if SB = 0 then goto Next Instruction;
    SC:= SB(11:23); AR:= Wa[pre]; BR:= W[fr];
    if SB(0) = 0 then
      begin comment address > 0;
        if SB ≥ 64 then SC:= 48;
        for SC:= SC-1 step -1 until 0 do
          begin Test Shift; lshl ARconBR end;
        W[fr]:= BR; W[pre]:= AR(0:23)
      end;
```

```
  if SB(0) = 1 then
    begin comment address < 0;
      if SB ≤ -65 ∧ AR(-1) = 0 then W[fr]:= W[pre]:= 0;
      if SB ≤ -65 ∧ AR(-1) = 1 then W[fr]:= W[pre]:= -1;
      if SB > -65 then
        begin
          for SC:= SC+1 step 1 until 0 do ashr ARconBR;
          W[fr]:= BR; W[pre]:= AR(0:23)
        end;
    end;
  goto Next Instruction
end ad 37;
```

Shift Single Logically:

Shift the contents of the W register the number of places specified by the effective address in SB. If SB is negative, then shift right with zero extension in the upper bits; otherwise shift left with zero extension in the lower bits. Overflow is not indicated.

```
begin
  if SB = 0 then goto Next Instruction;
  SC:= SB(11:23); AR:= Wa[fr];
  if SB(0) = 0 then
    begin comment address > 0;
      BR:= 0; if SB ≥64 then SC:= 35;
      for SC:= SC-1 step -1 until 0 do lshl ARconBR;
    end;
  if SB(0) = 1 then
    begin comment address < 0;
      if SB ≤ -65 then W[fr]:= 0;
      if SB > -65 then
        begin
          for SC:= SC+1 step 1 until 0 do lshr ARconBR; W[fr]:= AR(0:23)
        end;
    end;
  goto Next Instruction
end ls 38;
```

Shift Double Logically:

Shift the contents of the register pair Wpre and W the number of places specified by the effective address in SB. If SB is negative, then shift right with zero extension in the upper bits; otherwise shift left with zero extension in the lowest bits. Overflow is not indicated.

```
begin
  if SB = 0 then goto Next Instruction;
  SC:= SB(11:23); AR:= Wa[pre]; BR:= W[fr];
  if SB(0) = 0 then
    begin comment address > 0;
      if SB ≥ 64 then SC:= 48;
      for SC:= SC-1 step -1 until 0 do lshl ARconBR;
      W[fr]:= BR; W[pre]:= AR
    end;
  if SB(0) = 1 then
    begin comment address < 0;
      if SB ≤ -65 then W[fr]:= W[pre]:= 0;
      if SB > -65 then
        begin
          for SC:= SC+1 step 1 until 0 do lshr ARconBR;
          W[fr]:= BR; W[pre]:= AR(0:23)
        end;
    end;
  goto Next Instruction
end ld 39;
```

Normalize Single:

Shift the contents of the W register left with zero extension until bit 0 is different from bit 1. The number of shifts performed is stored as a negative integer in the storage byte addressed. If W = 0, the number of shifts is set to -2↑11.

```
begin
  AR:= Wa[fr]; BR:= 0; SC:= 0;
  if AR(0) = AR(1) ∧ AR ≠ 0 then
    begin
      for SC:= SC, SC-1 while AR(0) = AR(1) do lshl ARconBR;
      comment Exponent = SC; W[fr]:= AR;
      wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
```

```
        if SB(0:22) > word limit then goto Instruction Exception;
        SBconPK:= ST[STaddr];
        if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
        if HA(23) then SB(12:23):= SC(12:23) else SB(0:11):= SC(12:23);
        ST[STaddr]:= SBconPK; goto Next Instruction
      end;
    if AR = 0 then
      begin
        AR(-1:11):= 0; AR(12:23):= -2↑11; comment Exponent = -2↑11;
      end;
    if AR(0) ≠ AR(1) then begin AR:= 0; comment Exponent = 0 end;
    wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
    if SB(0:22) > word limit then goto Instruction Exception;
    SBconPK:= ST[STaddr];
    if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
    if HA(23) then SB(12:23):= AR(12:23) else SB(0:11):= AR(12:23);
    ST[STaddr]:= SBconPK; goto Next Instruction
  end ns 34;
```

Normalize Double:

Shift the contents of the register pair Wpre and W left with zero extension until bit 0 is different from bit 1. The number of shifts performed is stored as a negative integer in the storage byte addressed. If W = 0, the number of shifts is set to -2↑11.

```
  begin
    AR:= Wa[pre]; BR:= W[fr]; SC:= 0;
    if AR(0) = AR(1) ∧ AR ≠ 0 then
      begin
  UNNORMALIZED REGISTER:
        for SC:= SC, SC-1 while AR(0) = AR(1) do lshl ARconBR;
        comment Exponent = SC; W[fr]:= BR; W[pre]:= AR;
        wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
        if SB(0:22) > word limit then goto Instruction Exception;
        SBconPK:= ST[STaddr];
        if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
        if HA(23) then SB(12:23):= SC(12:23) else SB(0:11):= SC(12:23);
        ST[STaddr]:= SBconPK; goto Next Instruction;
      end;
```

```
    if AR(0) ≠ AR(1) then
      begin
        AR:= 0; comment Exponent = 0;
  NORMALIZED REGISTER:
        wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
        if SB(0:22) > word limit then goto Instruction Exception;
        SBconPK:= ST[STaddr];
        if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
        if HA(23) then SB(12:23):= AR(12:23) else SB(0:11):= AR(12:23);
        ST[STaddr]:= SBconPK; goto Next Instruction;
      end;
    if AR = 0 then
      begin
        AR:= Wa[fr]; BR:= Wa[fr];
        if AR ≠ 0 then begin AR:= 0; goto UNNORMALIZED REGISTER end;
          else begin AR(-1:11):= 0; AR(12:23):= -2↑11; goto NORMALIZED REGISTER end
      end;
  end nd 35;
```

Jump with Register Link:

If the W field ≠ 0, the instruction counter is stored in the W register. Following this, a jump is made to the effective address.

```
  begin
    comment Let the jump be from location r to location a. This is, for
      example, executed by the following instruction
                                        r: jl a;
  Jump:
    AR:= SBa; wait until Accept; STaddr:= SBaddr;
    comment Fetch instruction in location a;
    BR:= 5ext0conICcon0; comment IC = r+1;
    if SB(0:22) > word limit then goto Instruction Exception;
    if -,(Main Power Key ON ∧ -,Reset) then
      begin comment Set IC = r, i.e. the jump is not executed;
        AR:= 6ext0conICcon0; SB:= -2; IC:= AR+SBa;
        goto Reset System
      end;
```

```
SBconPK:= ST[STaddr]; IC:= AR(5:22); comment IC:= a;
if -,(MMode ∨ -,PROTECT) then
  begin comment Set IC = r+1;
    IC:= BR(5:22); goto Instruction Exception
  end;
if FR(6,7) ≠ 0 then begin comment Store link; W[fr]:= BR end;
if or (IR ∧ IM) ∧ ITRenable then
  begin comment IC = a; goto Interruption Service end;
FR:= SB(0:11); SB(0:11):= 12extSB(12); IC:= IC+1;
if -,FR(8) then AR:= Wa[index]; MMode:= PROTECT;
if Single Instruction then wait until Continue;
goto Address Modifications
end jl 13;
```

Skip if Register High:

Compare the W register and the effective address as signed integers. If the register is greater than the address, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= Wa[fr]; AR:= AR-SBa;
  if AR > 0 then IC:= IC+1; goto Next Instruction
end sh 40;
```

Skip if Register Low:

Compare the W register and the effective address as signed integers. If the register is less than the address, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= Wa[fr]; AR:= AR-SBa;
  if AR(-1) then IC:= IC+1; goto Next Instruction
end sl 41;
```

Skip if Register Equal:

Compare the W register and the effective address as signed integers. If the register equals the address, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= Wa; AR:= AR-SBa;
  if AR = 0 then IC:= IC+1; goto Next Instruction
end se 42;
```

Skip if Register Not Equal:

Compare the W register and the effective address as signed integers. If the register is unequal to the address, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= Wa; AR:= AR-SBa;
  if AR ≠ 0 then IC:= IC+1; goto Next Instruction
end sn 43;
```

Skip if Register Bits One:

Use the effective address as a mask to test selected bits in the W register. If all the selected bits are one, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= SBa; SB:= W[fr]; AR:= AR ∧ ¬SBa;
  if AR = 0 then IC:= IC+1; goto Next Instruction
end so 44;
```

Skip if Register Bits Zero:

Use the effective address as a mask to test selected bits in the W register. If all the selected bits are zero, then skip the following instruction. The register remains unchanged.

```
begin
  AR:= Wa; AR:= AR ∧ SBa;
  if AR = 0 then IC:= IC+1; goto Next Instruction
end sz 45;
```

Skip if No Exceptions:

Use the right-most three bits of the effective address as a mask to test the exception register. If the selected exception bits are zero, then skip the following instruction. The exception register remains unchanged.

```
begin
  AR:= 22extOconEX; AR:= AR ∧ SBa;
  if AR = 0 then IC:= IC+1; goto Next Instruction
end sx 46;
```

Skip if No Protection:

Use the protection key of the storage word addressed to select a bit in the protection register. If the selected bit is zero, then skip the following instruction.

```
begin
  wait until Accept; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,PROTECT then IC:= IC+1; goto Next Instruction
end sp 21;
```

Jump with Interrupt Enabled:

Same as Jump with Register Link, except that the interruption system is enabled first. This is a privileged instruction.

```
begin
  ITRenable:= 1; if -,MMode then goto Instruction Exception;
  goto Jump; comment See Jump with Register Link;
end je 15;
```

Jump with Interrupt Disabled:

Same as Jump with Register Link, except that the interruption system is disabled first. This is a privileged instruction.

```
begin
  ITRenable:= 0; if -,MMode then goto Instruction Exception;
  goto Jump; comment See Jump with Register Link;
end jd 14;
```

Clear Interrupt Bits:

Use the effective address as a mask to clear selected interruption signals. This is a privileged instruction.

```
begin
  if -,MMode then goto Instruction Exception;
  IR:= IR ∧ -,SB; goto Next Instruction
end ic 47;
```

Store Interrupt Register:

Store the interrupt register in the storage word addressed. The interrupt register remains unchanged.

```
begin
  AR:= IRa;
  wait until Accept SB; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  SB:= AR(0:23); ST[STaddr]:= SBconPK;
  goto Next Instruction
end is 31;
```

Load Mask Register:

Insert the storage word addressed in the interrupt mask register. Bit 0 of the mask register is permanently equal to one. This is a privileged instruction.

```
begin
  wait until Accept; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,MMode then goto Instruction Exception;
  IM(1:23):= SB(1:23); goto Next Instruction
end ml 12;
```

Store Mask Register:

Store the interrupt mask register in the storage word addressed. The mask register remains unchanged.

```
begin
  AR:= IMa;
  wait until Accept SB; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  SB:= AR(0:23); ST[STaddr]:= SBconPK;
  goto Next Instruction
end ms 30;
```

Load Exception Register:

Insert the right-most three bits of the storage byte addressed into the exception register. The storage byte remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  EX:= if HA(23) then SB(21:23) else SB(9:11);
  goto Next Instruction
end xl 16;
```

Store Exception Register:

Store the exception register in the right-most three bits of the storage byte addressed. The left-most nine bits of the storage byte are set to zero. The exception register remains unchanged.

```
begin
  AR:= 21ext0conEX;
  wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  if HA(23) then SB(12:23):= AR(12:23) else SB(0:11):= AR(12:23);
  ST[STaddr]:= SBconPK; goto Next Instruction
end xs 27;
```

Load Protection Register:

Insert the right-most seven bits of the storage byte addressed into the protection register. Bit 0 of the protection register is permanently equal to one. The storage byte remains unchanged. This is a privileged instruction.

```
begin
  wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,MMode then goto Instruction Exception;
  PR(1:7):= if HA(23) then SB(17:23) else SB(5:11);
  goto Next Instruction
end pl 28;
```

Store Protection Register:

Store the protection register in the right-most eight bits of the storage byte addressed. The left-most four bits of the storage byte are set to zero. The protection register remains unchanged:

```
begin
  AR:= 17extOconPR;
  wait until Accept; STaddr:= SBaddr; HA(23):= SB(23);
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,(MMode ∨ -,PROTECT) then goto Instruction Exception;
  if HA(23) then SB(12:23):= AR(12:23) else SB(0:11):= AR(12:23);
  ST[STaddr]:= SBconPK; goto Next Instruction
end ps 29;
```

Load Protection Key:

Load the right-most three bits of the W register with the protection key of the storage word addressed. The left-most twenty-one bits of the W register are set to zero. The protection key of the storage word remains unchanged.

```
begin
  wait until Accept; STaddr:= SBaddr;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  W[fr]:= 21extOconPK; goto Next Instruction
end kl 22;
```

Store Protection Key:

Store the right-most three bits of the W register into the protection key of the storage word addressed. The register remains unchanged. This is a privileged instruction.

```
begin
  wait until Accept; STaddr:= SB;
  if SB(0:22) > word limit then goto Instruction Exception;
  SBconPK:= ST[STaddr];
  if -,MMode then goto Instruction Exception;
  PK:= W[fr](21:23); ST[STaddr]:= SBconPK;
  goto Next Instruction
end ks 51;
```

Input/Output:

An input/output operation is initiated if the selected device is available. If the device is busy or disconnected, the operation is rejected. This is indicated in the exception register. The write command causes an immediate transfer of the W register to the selected device buffer, followed by an output operation to the external data medium. The control command is identical to the write command, the only exception being that the output operation is replaced by a control operation. The read command directs the device to start a transfer of the next character from the external data medium into its buffer register. Finally, the sense command is a request to the device to transfer the contents of its buffer register to the W register.

```
begin
  if -,MMode then goto Instruction Exception;
  EX(22,23):= 0; Selected Device:= SB(0:17);
  if Disconnected[Selected Device] then EX(22):= 1;
  if Busy[Selected Device] ∧ -,Disconnected[Selected Device] then EX(23):= 1;
  if EX(22,23) ≠ 0 then goto Next Instruction;
  BR:= SB;
  if BR(23) then
    begin comment Write or Control command;
      SB:= W[fr]; Device Buffer[Selected Device]:= SB;
      goto Next Instruction
    end;
  if BR(22) then
    begin comment Read command end;
    else
    begin comment Sense command; W[fr]:= Device Buffer[Selected Device] end;
  goto Next Instruction
end io 1;
```

Autoload Word:

Four 6-bit characters from device number 0 are loaded into the storage word addressed and the protection key of the storage word is set to zero. This is a privileged instruction. The computer is set in the reset state if the loading device is disconnected or if any of the status bits are set during input.

```
begin
  comment Save load address in the registers, SE and BE. Set SC to 4,
    namely the number of characters to be read;
```

```
  SE:= SB(0:11)con0con0; SB(0:11):= SB(12:23); BE:= SB(0:11);
  AF:= 0; SC:= 2; SC:= SC+1; SC:= SC+1;
Next Character:
  SB:= 2; comment Read command for device 0;
Start Input:
  if -,MMode then goto Instruction Exception;
  EX(22,23):= 0; Selected Device:= SB(0:17);
  if Disconnected[Selected Device] then EX(22):= 1;
  if Busy[Selected Device] ∧ -,Disconnected[Selected Device] then EX(23):= 1;
  if EX(22,23) ≠ 0 then
    begin
      BR:= 21ext0conEX;
      goto if BR(22) then Reset System else Start Input;
    end;
  BR:= SB; 2 lshl AF;
Read:
  if BR(22) = 1 then
    begin comment The read command is accepted, for which reason we set
            up the sense command;
      SB:= 0; goto Start Input
    end;
Sense:
  2 lshl AF; SB:= Device Buffer[Selected Device];
  if SB(0) = 1 then
    begin comment End of buffer. The contents of the Device Buffer is a
          pseudo character;
      BR:= o 0000 0060;
Shift:
      if BR(23) = 0 then begin lshr ARconBR; goto Shift end;
      2 lshr ARconBR; goto Next Character
    end;
  AR:= AR ∨ SB(0,0:23); SC:= SC-1; SB(12:23):= 0;
  if SB ≠ 0 then
    begin comment In this case, we have parity error or end of tape;
      goto Reset System
    end;
  if SC ≠ 0 then goto Next Character;
```

```
comment Store the four 6-bit characters contained in AR in the word
  addressed by SE and BE;
SB(0:11):= BE; SB(12:23):= SB(0:11); SB(0:11):= SE(0:11);
wait until Accept SB; STaddr:= SBaddr;
if SB > word limit then goto Instruction Exception;
SBconPK:= ST[STaddr];
if -,MMode ∨ -,PROTECT then
  begin SB:= AR(0:23); PK:= 0 end;
ST[STaddr]:= SBconPK; goto Next Instruction
end aw 0;
```

RCSL: 51-VB698
Author: Allan Giese
Edited: November 1969

THE MICROPROGRAM

FOR

THE RC 4000 COMPUTER

A/S REGNECENTRALEN
Falkoneralle 1
2000 Copenhagen F

CONTENTS:

1. DESIGN AND OPERATION OF THE MICROPROGRAM STORE.

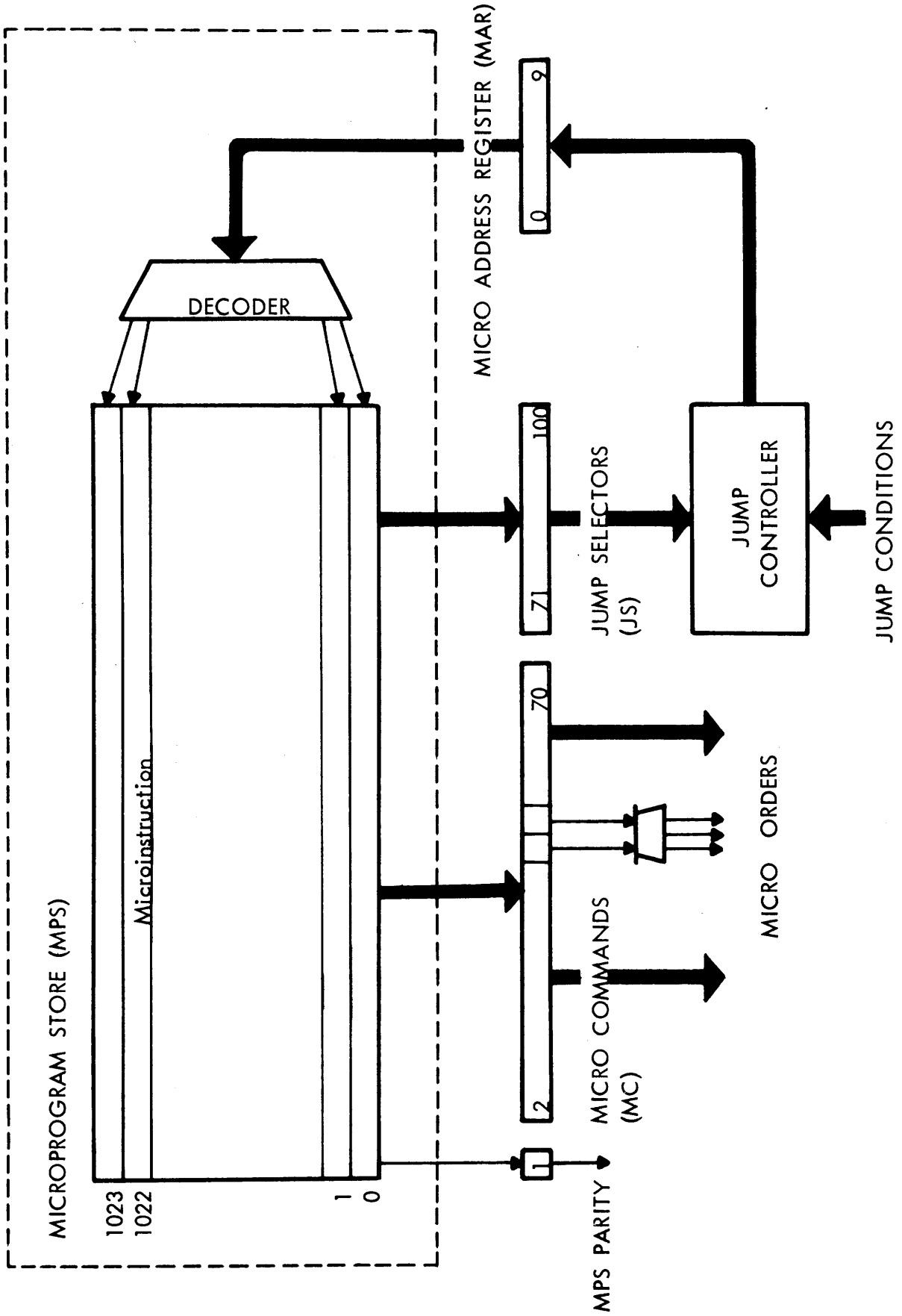
The two design approaches to the control logic of a computer are either to build a tailor-made logic box for each instruction, or to employ a microprogram. For medium size computers with a rather complex instruction set, the number of boxes is rather high and, as they differ from one another, the design and maintenance problem is enlarged. The microprogram, with its orderly structure, is therefore a more feasible approach for the medium scale RC 4000 computer. Also the economical aspects are in favour of a microprogram.

The microprogram store (MPS) we have employed is organized like a general store having a maximum capacity of 1024 words, each of 100 bits. The store is designed as a decoding network implemented entirely by means of integrated circuits. Only 472 of the possible 1024 words are used by the microprogram, and the not used words will give an all zero result if they were selected.

Figure 1.1 reflects the manner in which the MPS operates. For each cycle, a word is read out of the MPS and strobed into the three registers: MPS Parity, Micro Command, and Jump Selector where it is held for the duration of the cycle. The cycle time or the repetition rate for the MPS is 500 nanoseconds. The MC register supplies the computer with the proper control signals, and the JS register is used to address the next MPS word. The next address is controlled not only by the JS, but, what is more essential, also by data dependent conditions generated in the processor; for example, an overflow indication, an outcome of a decoding and so forth. The Micro Address Register (MAR) is then set equal to Next Address and a new MPS word is selected, and with that a new cycle is initiated.

The above description is correct as long as Running Mode is 1. If Running Mode becomes 0 (Stopped Mode = 1), no Next Address is written into MAR and the micro commands from 11 to 69 are cleared to zero. Hence, MAR, JS, MC(2:10), and MPS Parity remain unaltered.

For mnemonic reasons, MPS is logically divided into 32 sections, and the p'th word in the n'th section is identified as xnyp (xy-number). Hence, the first five bits of MAR represent the section number, and the last five bits the number within the section.



RC4000

VB 698

Figure 1.1.

MICROPROGRAM CONTROL

2. JUMP SELECTORS AND JUMP CONDITIONS.

2.1 Generation of Next Address.

The next microinstruction to be selected from the MPS is a function of the current jump selectors and the values of the jump conditions.

The TABLE OF JUMP CONDITIONS in Appendix B shows the jump conditions arranged in a matrix where the ten columns correspond to the ten bits of MAR. Empty positions signify not used jump conditions. The row to be selected in each column is controlled by a group of 3 jump selectors as indicated in the rectangular boxes below the jump conditions. As an example, let us suppose the jump selectors 71,76,85,91,95, and 97 are active corresponding to the next address

$$AR(-1) = AR(0),1,0,0,1,\ \ 0,0,0,SB \neq 0,0$$

This is a four-way branch based on two independent conditions. The xy addresses are calculated in the following table

AR(-1) = AR(0)	SB ≠ 0	Next Address
0	0	x9y0
0	1	x9y2
1	0	x25y0
1	1	x25y2

The just explained evaluation of next address may be overruled by sending the signal Fixed Address to the jump controller, and in this case MAR is explicitly set to x31y31.

2.2 Explanation of Jump Conditions.

The simple jump conditions whose interpretation is evident will not be found is this list.

Accept	is a signal generated in the Store Controller. It is 1 when the core store is accessible for data transfer to the central processor. Accept is 0 if the High-Speed Data Channel occupies the core store.
Autoload	becomes 1 for a period of 500 nanoseconds for each time the AUTOLOAD pushbutton on the Operator Console is depressed.
Carry(0)	is the carry generated in position 0 and supplied to position -1 of the adder circuitry.
FDsub	is controlled by the micro command Test FD Sign. The floating-point adder is set to subtraction as long as FDsub = 1.
HA(23)	When the microprogram executes one of the micro instructions, Read Instruction, Read Data (Double), or Read Split (Double), and the command is accepted by the Store Controller, then HA(23) is always set equal to SB(23), i.e. the least significant bit of the byte address. In other words, HA(23) is 1 for odd and 0 for even byte addresses.
Itr	is a signal from the Interruption Unit telling the microprogram to switch to the Interruption Service routine. Itr equals 1 when ITRenable $\wedge$ (ors (IR $\wedge$ IM)) $\neq$ 0.
Main Power Key ON $\wedge$ -,Reset	Main Power Key ON is 1 when the key is in position ON, otherwise 0. Reset is 1 as long as the RESET pushbutton on the Operator Console is depressed.

MMode — is an abbreviation for Monitor Mode.

MMode ∨ -,PROTECT — see MMode and PROTECT.

Modif — is a decoding of the address modification bits relative, indirect, and index. The equation is:

$$Modif := FR(8:11) \neq 0.$$

PROTECT — is a decoding network whose value depends on PR and PK. The equation is:

$$PROTECT := PR(PK) = 1.$$

Round — is used in connection with floating-point arithmetic in order to determine the correct rounding procedure.
The equation of the decoding is:

$$\begin{aligned} Round := {} & AF(-1) \neq AF(0) \wedge AF(35) \\ & \vee AF(-1) = AF(0) \wedge AF(0) \neq AF(1) \wedge AF(36) \\ & \vee AF(-1) = AF(0) \wedge AF(0) = AF(1) \wedge AF(37) \end{aligned}$$

and the lines in the equation characterize the mantissa as overflow, normalized, and underflow, taken in line order.

3. INTRODUCTION TO MICRO COMMANDS AND MICRO ORDERS.

One way of controlling the computer is to associate a given control function, from now on called a micro order, to each micro command. Hence a micro order is active if the corresponding MC bit is a 1, and any two micro orders may be active at the same time. For a computer as complex as the RC 4000 computer, it would require about 50 per cent more micro commands than the 69 supplied by the MC register. The additional micro orders are generated by grouping two or more MC bits into a field and then decode this field. For example, the three bits MC(51,52,53) (confer Appendix A) generates 7 micro orders, but by doing so we have restricted ourselves to use one and only one of the micro orders simultaneously. The groups should therefore be formed in such a way that this restriction does not influence the instruction execution time.

Appendix A is a list of the micro orders. The vast majority of the micro orders are transfer or other simple functions which are selfexplanatory, but a few of them are more complex, however, and they are explained in Section 4.

4. EXPLANATION OF MICRO ORDERS.

We have here collected the micro orders whose function are not straightforward. The corresponding micro commands are logic diagrams are also given.

4.1 IO Phase A, IO Phase B, IO Timing, and BUS(0:23):= IO Data.

MC(2,3,4), MC(47:50) = 7
(ARU026:028), (LCI003, 011:012, 020,026)

The first three signals control the peripheral devices and their controllers via the Low-Speed Data Channel. The fourth micro order transfers the contents of the Low-Speed Data Channel to the arithmetic bus. Details about their mode of operation are found in the chapter concerning input/output.

4.2 Add, Sub, AddE, SubE, Carry 24, Carry 36, and Carry 38.

MC(5:10)
(ARU098:100)

The adder circuitry has a width of 40 bits, and it is constructed to perform a number of different arithmetic operations on two operands. The first of the operands is always ARconAE, while the second operand depends on the active micro orders.

We have on basis of the micro commands MC(5:10) and the three bistables WDsub, FDadd, and FDsub constructed the micro orders listed below. WDsub is controlled by Test WD Sign (MC(67:70) = 9) while FDadd and FDsub are controlled by Test FD Sign (MC(67:70) = 8).

Add:= -,MC(6) ∧ -,WDsub ∧ -,FDsub
Sub:= MC(5) ∨ MC(6) ∨ WDsub ∨ FDsub
AddE:= -,MC(7) ∧ -,FDsub
SubE:= -,MC(8) ∧ -,FDadd
Carry 24:= MC(9) ∨ WDsub
Carry 36:= MC(10)
Carry 38:= -,AddE

Add and Sub control the 26 most significant input bits to the adder from the second operand whereas AddE and SubE control the 14 least significant bits, see Figure 4.1.

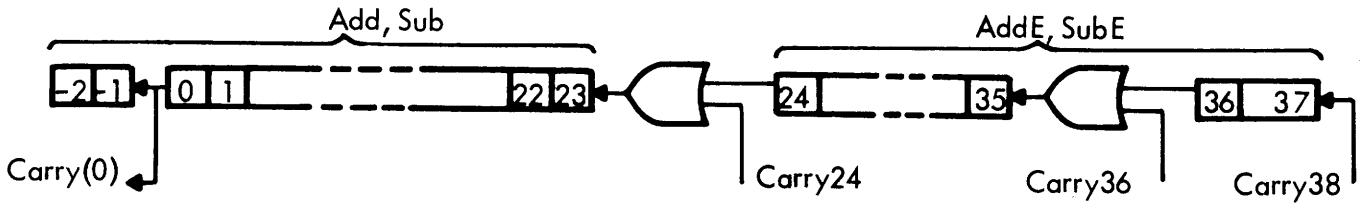


Figure 4 . 1 . Adder Configuration

The input from the second operand can now be expressed in terms of the micro orders as follows:

TABLE 4.1

Add	Sub	Second Operand(-2:23)
0	0	Impossible
0	1	-,SB(0,0,0:23)
1	0	SB(0,0,0:23)
1	1	26ext0

AddE	SubE	Second Operand(24:37)
0	0	14ext1
0	1	-,SE
1	0	SE
1	1	14ext0

The first operand is defined as

First Operand(-2:23) = AR(-1,-1:23) and

First Operand(24:37) = AE.

Table 4.2 shows the relationship between micro commands, micro orders, and second operand for a number of instructive cases. The table is only correct provided WDsub = FDadd = FDsub = 0.

TABLE 4.2

MC(5)	MC(6)	MC(7)	MC(8)	MC(9)	MC(10)	Add	Sub	AddE	SubE	Carry 24	Carry 36	Carry 38	Second Operand(-2:37)
0	0	0	0	0	0	1	0	1	1	0	0	0	+SB(0,0,0:23)con14ext0
0	0	0	0	1	0	1	0	1	1	1	0	0	+SB(0,0,0:23)con14ext0 + 2**14
0	0	0	1	0	0	1	0	1	0	0	0	0	+SF(0,0,0:37)
1	0	0	0	0	0	1	1	1	1	0	0	0	+0
1	0	0	0	0	1	1	1	1	1	0	1	0	+4
1	0	0	0	1	0	1	1	1	1	1	0	0	+2**14
0	1	0	0	1	0	0	1	1	1	1	0	0	-SB(0,0,0:23)con14ext0
0	1	0	0	0	0	0	1	1	1	0	0	0	-SB(0,0,0:23)con14ext0 - 2**14
0	1	1	0	0	0	0	1	0	1	0	0	1	-SF(0,0,0:37)

The output from the adder is called SUM and the following equation holds

SUM(-2:37):= AF(-1,-1:37) + Second Operand(-2:37)

A shorthand notation for describing the state of the adder circuitry is (b meaning binary):

Adder:= b<Add><Sub><AddE><SubE> <Carry 24><Carry 36><Carry 38>

An example; if no micro commands are specified in a microinstruction, the adder will be set up for integer addition, i.e.

Adder:= b1011 000.

The output from the adder is normally strobed into the receiving register in the middle of a microprogram cycle so the decoding network of the receiving register has time enough to stabilize and thereby control the generation of Next Address before it is read into MAR. In order to achieve this, it is necessary to set up the adder controls 1 microinstruction before the adder output is written into the receiving register. A typical example of a sequence of adder micro orders is found in Appendix B microprogram page 21 (x17y5, x1y15, x0y25, x1y25, and x1y27).

4.3 BUS(0:23):= AND(0:23).

MC(13)
(ARU098)

Logical And operations are also incorporated in the adder, and we have

AND(0:23):= AR(0:23) ∧ Second Operand(0:23).

The value of Second Operand depends of the state of the adder.

4.4 Shifts.

MC(54:56)
(ARU080:083)

The 6 micro orders:

lshl AF
lshl ARconBR
ashr AF
ashr ARconBR
lshr AF
lshr ARconBR

formed by the 3 micro commands MC(54:56) are meaningful only when used in conjunction with the adder because the adder is used as a shifter. For normal shifts, the adder controls should be

Adder:= b 1111 000, i.e. SUM(-2:37):= AF(-1,-1:37) + 0.

Roughly spoken, left shifts are performed by taking the double SUM and right shifts by taking the half SUM.

The clock pulse for storing the shifted data arrives late in the microprogram cycle, approximately at Time 350, and there is therefore no need to set up the adder for shifts in the preceding microinstruction.

The exact definitions of the micro orders are:

```
procedure lshl AF;
  begin ‡ AF(-1:37):= SUM(0:37)con0 end;

procedure lshl ARconBR;
  begin ‡ AR(-1:23)conBR(0:23):= SUM(0:23)conBR(0:23)con0 end;

procedure ashr AF;
  begin ‡ AF(-1:37):= SUM(-2:36) end;

procedure ashr ARconBR;
  begin
    ‡ AR(-1:23):= SUM(-2:22);
    ‡ BR(0,1:23):= SUM(23)conBR(0:22);
  end
```

```
procedure lshr AF;
  begin ‡ AF(-1,0,1:37):= 0con0conSUM(0:36) end;

procedure lshr ARconBR;
  begin
    ‡ AR(-1,0,1:23):= 0con0conSUM(0:22);
    ‡ BR(0,1:23):= SUM(23)conBR(0:22);
  end
```

4.5 Test WD Sign and Divide Integer.

MC(47:50) = 11, MC(67:70) = 9
(ARU080, 098:099)

Integer division is carried out by the non-restoring method. By this method, one binary quotient bit q is determined in each iteration of a recursive process. The recursive equation for the k'th iteration generates a new partial remainder X(k+1), as a function of the present partial remainder X(k) and the divisor D. The dividend X(0) is the remainder before the first iteration. The recursive equation includes either a subtraction or an addition depending on the relative signs of X(k) and D. If N denotes the word length, then the final N+1 bit quotient can be shown to be equal to -,q(-1), q(0),..., q(N-2),1.

The relationships are:

$$\text{sign } X(k) = \text{sign } D \Rightarrow q(k-1) = 1,\ X(k+1) = 2*X(k) - D*2**N$$
$$\text{sign } X(k) \neq \text{sign } D \Rightarrow q(k-1) = 0,\ X(k+1) = 2*X(k) + D*2**N$$

The purpose of Test WD Sign and Divide Integer is to control the adder and shift mechanisms so that each iteration is completed within 1 microprogram cycle. The exact definitions of the two micro orders are:

```
procedure Test WD Sign;
  begin WDsub:= if Running Mode = 1
                  then begin if SUM(-1) = SB(0) then 1 else 0 end
                  else WDsub
  end;
```

```
procedure Divide Integer;
  begin
    ǂ AR(-1:23):= SUM(0:23)conBR(0);
    ǂ BR(0:23):= BR(1:22)conWDsubcon0;
  end
```

The clock pulses for the two orders arrive at exactly the same time as the clock pulses for the shift orders, i.e. at approximately Time 350. Running Mode controls Test WD Sign, and this dependence makes it possible to manually execute the division microinstruction by microinstruction.

Let us illustrate an integer division by dividing 27 by 4 using 4-bit words; confer also the microprogram Appendix B, microprogram page 16. Before the iteration starts the contents of SB are set to 4 and ARconBR to 27.

TABLE 4.3

VB698

Dividend = 27, Divisor = 4, N = 4

SB = b 0100, -SB = b 1100

xy-number	Micro Orders		AR(-1:3)	BR(0:3)	WDsub	SUM(-1:3)
	comment Initial conditions		0 0 0 0 1	1 0 1 1	0	
x20y20	Adder:= b1111 000; Test WD Sign; lshl ARconBR	X(0) 2*X(0)	0 0 0 0 1 0 0 0 1 1	1 0 1 1| 0 1 1|0	 1=q(-1)	00001 11111
x6y19	Divide Integer; Test WD Sign	2*X(1)	1 1 1 1 0	1 1|1 0	0=q(0)	00010
x6y19	Divide Integer; Test WD Sign	2*X(2)	0 0 1 0 1	1|1 0 0	1=q(1)	00001
x6y17	Divide Integer; Test WD Sign	2*X(3)	0 0 0 1 1	|1 0 1 0	1=q(2)	11111
x0y15	Divide Integer	2*X(4)	1 1 1 1|1	0 1 1 0	0	
x24y6	Adder:= b1111 0000; ashr AF	X(4)	1 1 1 1 1|	0 1 1 0		11111

Remainder = AR(-1:3) = -1

Quotient = BR(0:2)con1 = 7

Result 27/4 = 7 with the remainder -1

4.6 Test FD Sign and Divide Floating.

MC(47:50) = 8, MC(67:70) = 8
(ARU080,098:099)

Floating division is also carried out by the non-restoring method; confer 4.5. The iterations stop when a normalized quotient is obtained. The exact definitions of the two micro orders are:

```
procedure Test FD Sign;
  begin FDsub:= if Running Mode = 1
                then begin if SUM(-1) = SB(0) then 1 else 0 end
                else FDsub;
        FDadd:= if Running Mode = 1
                then begin if SUM(-1) ≠ SB(0) then 1 else 0 end
                else FDadd
  end;

procedure Divide Floating;
  begin
    ↓ AF(-1:37):= SUM(0:37)con0;
    ↓ BF(0:35):= BF(1:35)conFDsub;
  end;
```

The clock pulses for the two orders arrive at exactly the same time as the clock pulses for the shift order, i.e. at approximately Time 350.

Running Mode controls Test FD Sign, and this dependence makes it possible to manually execute the division microinstruction by microinstruction.

Likewise integer division, we shall illustrate a floating-point division, or to be more accurate, the division of two mantissae by an example. Table 4.3 shows the example, and the associated microprogram is found in Appendix B, microprogram page 28.

TABLE 4.4

VB698

Dividend = 5/8, Divisor = 6/8, N = 4

SF = b0.110, -SF = b1.010

xy-number	Micro Orders		AF(-1:3)	BF(0:3)	FDsub	FDadd	SUM(-1:3)
	comment Initial conditions	X(0)	00.101	----	0	0	
x6y1	Adder:= b 1111 000		00.101	----	0	0	00101
	Test FD Sign		00.101	----	1=q(-1)	0	11111
x8y13	Divide Floating	2*X(1)	11.110	---1	0	0	
x9y6	Adder:= b 1111 000; BF:= 0		11.110	0000			11110
	Test FD Sign		11.110	0000	0=q(0)	1	00100
x8y7	Divide Floating; Test FD Sign	2*X(2)	01.000	0000	1=q(1)	0	00010
x8y7	Divide Floating; Test FD Sign	2*X(3)	00.100	0001	1=q(2)	0	11110
x8y7	Divide Floating; Test FD Sign	2*X(4)	11.100	0011	0=q(3)	1	00010
x8y7	Divide Floating; Test FD Sign	2*X(5)	00.100	0110	1=q(4)	0	

The normalized quotient equals 0.1101... and rounded to 3 binary digits 0.111 or 7/8

4.7 Test Integer.

MC(47:50) = 1
(ARU101:102)

Overflow and Carry are examined by Test Integer, and the results are stored in the proper registers.

```
procedure Test Integer;
  begin
    if (SUM(-1) ≠ SUM(0)) ∧ ITRenable ∧ IM(1) then Itr:= 1;
      comment Interrupt request;
    if SUM(-1) ≠ SUM(0) then begin EX(22):= 1; IR(1):= 1 end;
    if Carry(0) then EX(23):= 1
  end;
```

4.8 Test Shift.

MC(47:50) = 4
(ARU101:102)

A left shift shall produce an overflow if bit(0) differs from bit(1) before the register is shifted.

```
procedure Test Shift;
  begin
    if (AR(0) ≠ AR(1)) ∧ ITRenable ∧ IM(1) then Itr:= 1;
      comment Interrupt request;
    if AR(0) ≠ AR(1) then begin EX(22):= 1; IR(1):= 1 end;
  end;
```

4.9 Test Exp.

MC(47:50) = 2
(ARU101:102)

Exponent overflow in floating-point operations is detected by this micro order.

```
procedure Test Exp;
  begin
    if (SC(11) ≠ SC(12)) ∧ ITRenable ∧ IM(2) then Itr:= 1;
      comment Interrupt request;
    if SC(11) ≠ SC(12) then begin EX(22):= 1; IR(2):= 1 end;
  end;
```

4.10 Test IO.

MC(47:50) = 3
(ARU101:102)

The status bits of the selected peripheral device are transferred via the input/output bus to the EX register when Test IO is activated.

```
procedure Test IO;
  begin
    if Disconnected then EX(22):= 1;
    if Busy ∧ -,Disconnected then EX(23):= 1
  end;
```

4.11 ITRenable:= FR(5).

MC(51:53) = 1
(ARU102)

If the memory element ITRenable is 1, an interrupt signal causes the running program to switch to the Interruption Routine; if, on the contrary, ITRenable = 0, the running program proceeds.

It should be observed that the function codes for jump enable and jump disable are 15 (FR(5) = 1) and 14 (FR(5) = 0), respectively.

The disable mode is set in the Interruption Routine by applying all zeroes to the FR register before the micro order is executed. When the micro order is then executed, ITRenable is, of course, reset to 0, but, moreover, the asynchronous set signal to the D-type memory element Itr is set to 1. The set signal could have the value 0 due to the micro orders Test Integer, Test Shift, and Test Exp. It should be remembered that a D-type element remains in the 1-state as long as the set signal is 0.

4.12 Read Instruction, Read Data, Read Split, Split Write, and Double.

MC(61:65)
(STC001:022)

The micro orders to be explained in this section are all related to the data transfer from central processor to core store and vice versa. They are very important but also rather complex micro orders since the data transfer may take as long as up to 5 microprogram cycles (2.5 microseconds).

An easily readable introduction is found in the paper CORE STORE CONTROLLER FOR THE RC 4000 COMPUTER, Section 2 (Design Considerations), and the reader should consult this paper before he continues. In contrast, Section 4 in the said paper provides a minute description. For most purposes, included the reading of the microprogram, an adequate explanation is expressed in the following procedures.

```
Instruction Exception:
wait 20;
comment The micro address is x31y31, due to Fixed Address, see Appendix C
  p.2;
x31y31: IR(0):= 1; SB:= 12;
x4y6: BR:= 23; Read Data;
etc.

procedure Read Instruction;
  begin
Time 0:
    if Accept ∧ (-,Itr ∨ FR(0:5) = 9) then
      begin
        comment Accept = 1 when the High-Speed Data Channel does not access
          the core store; wait 165;
Time 165:
        STaddr:= ICaddr; wait 65;
Time 230:
        HA(23):= SB(23);
        Fixed Address:= IC > word limit; wait 510;
Time 740:
        SBconPK:= ST(STaddr); FR:= ST(STaddr)(0:11); wait 240;
Time 980:
        if STaddr > 3 ∧ Core Store Parity Control
          then Core Store Parity Error:= -,odd ST(STaddr);
        if Fixed Address then goto Instruction Exception;
        Fixed Address:= -,(MMode ∨ -,PROTECT); wait 500;
Time 1480:
        if Fixed Address then goto Instruction Exception; wait 20;
Time 1500:
        if Core Store Parity Error then stop Microprogram
    end;
  end Read Instruction;
```

```
procedure Read Data or Read Data Double;
  begin
Time 0:
    if Accept then
      begin
        wait 165;
Time 165:
        STaddr:= if Double then BRaddr else SBaddr; wait 65;
Time 230:
        HA(23):= SB(23);
        Fixed Address:= if Double = 0 then SB(0:22) > word limit; wait 510;
        comment Read Data Double is only used in the microprogram when it is
          known beforehand that BR(0:22) does not exceed the capacity of the
          core store;
Time 740:
        SBconPK:= ST(STaddr); wait 240;
Time 980:
        if STaddr > 3 ∧ Core Store Parity Control
          then Core Store Parity Error:= -,odd ST(STaddr);
        if Fixed Address then goto Instruction Exception; wait 520;
Time 1500:
        if Core Store Parity Error then stop Microprogram;
      end;
  end Read Data or Read Data Double;

procedure Read Split or Read Split Double;
  begin
Time 0:
    if Accept then
      begin
        wait 165;
Time 165:
        STaddr:= if Double then BRaddr else SBaddr; wait 65;
Time 230:
        HA(23):= SB(23);
        Fixed Address:= if Double = 0 then SB(0:22) > word limit; wait 510;
        comment Read Split Double is only used in the microprogram when it is
          known beforehand that BR(0:22) does not exceed the capacity of the
          core store;
```

```
Time 740:
        SBconPK:= ST(STaddr); wait 240;
Time 980:
        if STaddr > 3 ∧ Core Store Parity Control
          then Core Store Parity Error:= -,odd ST(STaddr);
        if Fixed Address then goto Instruction Exception;
        SBenable:= MMode ∨ -,PROTECT; wait 520;
Time 1500:
        if Core Store Parity Error then stop Microprogram;
      end;
  end Read Split or Read Split Double;


procedure Split Write;
  begin
Time 1500:
    wait 165;
Time 1665:
    STdata:= SBconPK; wait 315;
    comment If STaddr < 4, one of the W registers is selected and in this
      case the arithmetic bus system is used as data carrier. The micro-
      program must therefore never specify any bus transfers from 1500 to
      2000;
Time 1980:
    SBenable:= 1; wait 520;
Time 2500:
  end Split Write;
```

5. FLOWCHART EXPLANATION.

The entire microprogram is presented in Appendix B in a flowchart version, and this chapter serves as a guidance for the notations and abbreviations used in the flowchart description.

The normal execution of an instruction falls into two steps. In the first step, an instruction is fetched from core store (or W registers), the effective address is calculated and the instruction number is decoded. These activities, which are common to all instructions are collected in the FETCH CYCLE, p. 1 of the flowcharts. From the FETCH CYCLE, the microprogram continues to step two, which is a collection of 64 subprograms corresponding to each of the 64 possible instructions. The subprogram to be chosen is determined by the instruction number. All these subprograms are as a whole referred to as the EXECUTE CYCLE, pp. 4-32 of the flowcharts. After execution of the EXECUTE CYCLE, the microprogram returns to the FETCH CYCLE, ready to fetch the next instruction.

Our first description of the FETCH CYCLE is incomplete as regards Interrupt and Reset. Interrupt signals are interrogated in the FETCH CYCLE and an interrupt causes a jump to the INTERRUPTION SERVICE, p.2 of the flowcharts. A reset condition conducts the microprogram to halt execution until either a Start or Autoload signal is applied by the operator. The necessary microinstruction, for execution of this part of the instruction logic, are collected under the title RESET, START, AND AUTOLOAD, p. 3 of the flowcharts.

A sharp distinction between EXECUTE and FETCH cycles is not always possible, since the last microinstruction, for some instructions, and the first microinstruction of the FETCH CYCLE are merged together in order to reduce the instruction execution time.

Each microinstruction is depicted by a rectangle, within which the activities for that particular microinstruction are written. On top of the rectangle, we have to the left the identification number of the microinstruction (xy-number), and to the right we may have any kind of information which can clarify the description. Also for ease of interpretation, the same microinstruction may be duplicated. The microinstruction are linked together with

straight lines where arrows indicate the direction of flow. Whenever a branch occurs, the straight lines are broken and the relevant jump conditions are inserted.

Let us illustrate the rules by inspecting the flowchart p. 14. The microinstruction to follow x3y4 (or x3y6) depends upon the two jump conditions MMode and Accept, and of course of the unconditional jump conditions 0 and 1, but they are of no interest in this connection. Suppose MMode = 1 and Accept = 0 then we follow the line labelled 10 and reach the microinstruction x20y6. As long as Accept remains 0, the microprogram continues to execute x20y6 otherwise the microprogram proceeds to x20y7.

From x20y2 leads an arrow to microinstruction x4y6, but in this case the microinstruction is not represented by a rectangle, but by a pentagon. The pentagon symbol is an off-page connector, saying that information about x4y6 is found on page 2, INTERRUPTION SERVICE. An xy-number written in continuation of an arrow is an on-page connector indicating that the microinstruction having this xy-number is specified on the same page. The arrow from x24y5 is an example of this kind.

A rectangle with no xy-number denotes actually 64 rectangles, namely the rectangles that constistute the first microinstruction of the EXECUTE CYCLE for each of the 64 possible instructions. These 64 microaddresses are listed in XY-NUMBERS FOR START OF EXECUTE CYCLE in Appendix B.

The following abbreviations are used in the flowcharts:

FR(0:5)	denotes the jump conditions FR(2),FR(3),FR(0),FR(1),FR(4),FR(5)
FR(0:5) ∧-,Modif	denotes the jump conditions FR(2)∧-,Modif,FR(3)∧-,Modif,FR(0)∧-,Modif, FR(1)∧-,Modif,FR(4)∧-,Modif,FR(5)∧-,Modif

ar,sb,br, be,sc	are auxiliary variables for the registers AR,SB, BR,BE, and SC. They have no resemblance in hardware and are used only to express parallelism. An example, suppose the initial value of SC is 2 and that the microprogram is going to execute the microinstruction x6y14 on p. 10 of the flowcharts. The microinstruction x6y14 will then be executed 3 times before x6y12 is executed.

```
Initial value:  SC = 2
        x6y14:  sc:= 2; SC:= 2-1; ...
        x6y14:  sc:= 1; SC:= 1-1; ...
        x6y14:  sc:= 0; SC:= 0-1; ...
        x6y12:  comment SC = -1; ...
```

RI	Read Instruction
RD	Read Data
RS	Read Split
SW	Split Write

A P P E N D I X A

MICRO ORDER LIST.

This is a complete list of micro orders arranged after increased micro command number. A number in the Reference column denotes where more information about the function of the micro order can be obtained.

		Reference
MC(2)	IO Phase A	4.1
MC(3)	IO Phase B	4.1
MC(4)	IO Timing	4.1
MC(5,6)		
0 0	Add:= 1; Sub:= 0	4.2
0 1	Add:= 0; Sub:= 1	4.2
1 0	Add:= 1; Sub:= 1	4.2
1 1	Add:= 0; Sub:= 1	4.2
MC(7,8)		
0 0	AddE:= 1; SubE:= 1	4.2
0 1	AddE:= 1; SubE:= 0	4.2
1 0	AddE:= 0; SubE:= 1	4.2
1 1	AddE:= 0; SubE:= 0	4.2
MC(9)	Carry 24:= 1	4.2
MC(10)	Carry 36:= 1	4.2
MC(11)	BUS(-1:23):= SUM(-1:23)	
MC(12)	AF(-1:37):= if -,MC(10) then SUM(-1:37) else SUM(-1:35)con0con0	
MC(13)	BUS(0:23):= AND(0:23)	4.3
MC(14)	Not used	
MC(15,16)		
0 0	No operation	
0 1	BUS(0:11):= 12extW(fr)(12)	
1 0	BUS(0:23):= W(fr)	
1 1	BUS(0:23):= 12extW(fr)(12)conW(fr)(12:23)	

Reference

```
MC(17)      W(fr)(0:11):= BUS(0:11)
MC(18)      W(fr)(12:23):= BUS(12:23)
MC(19)      BUS(0:23):= W(pre)
MC(20)      W(pre):= BUS(0:23)

MC(21)      BUS(0:23):= if index ≠ 0
                        then W(index) else 0

MC(22)      BUS(0:23):= 5ext0conIC(5:22)con0
MC(23)      IC:= BUS(5:22)
MC(24)      BUS(0:23):= 12ext0conSC(12:23)
MC(25)      SC:= BUS(11:23)
MC(26)      SC:= SC-1
MC(27)      SC:= SC+1
MC(28)      BUS(0:23):= SB
MC(29)      SB(12:23):= BUS(12:23)
MC(30)      BUS(0:23):= 12ext0conSB(0:11)
MC(31)      BUS(0:23):= SE(0:11)con12ext0
MC(32)      SE(0:13):= BUS(0:11)con0con0;
            SB(0:11):= 12extSB(12)

MC(33)      BUS(-1:23):= AR(-1:23)

MC(34)      AR(-1:23):= if MC(11)
                        then BUS(-1:23)
                        else BUS(0,0:23)

MC(35)      if -,FR(8) then AR(-1:23):= BUS(0,0:23)

MC(36)      BUS(0:23):= if EX(21) = 0
                        then AE(0:11)con12ext0
                        else AE(0:9,9,9)con12ext0

MC(37)      AE(0:13):= BUS(0:11)con0con0
MC(38)      BUS(0:23):= BR
MC(39)      BR:= BUS(0:23)
MC(40)      IR(0):= 1
MC(41)      MMode:= PROTECT
```

VB698

		Reference
MC(42,43,44,45)		
0 0 0 0	No operation	
0 0 0 1	BUS(0:23):= -2	
0 0 1 0	BUS(0:23):= 12	
0 0 1 1	BUS(0:11):= 0; BUS(12:23):= -2048	
0 1 0 0	BUS(0:23):= 1	
0 1 0 1	BUS(0:23):= -1	
0 1 1 0	BUS(0:23):= 48	
0 1 1 1	Not used	
1 0 0 0	BUS(0:23):= 6	
1 0 0 1	Not used	
1 0 1 0	BUS(0:23):= 23	
1 0 1 1	Not used	
1 1 0 0	BUS(0:23):= 35	
1 1 0 1	Not used	
1 1 1 0	BUS(0:23):= 2	
1 1 1 1	Not used	
MC(46)	EX(22,23):= 0	
MC(47,48,49,50)		
0 0 0 0	No operation	
0 0 0 1	Test Integer	4.7
0 0 1 0	Test Exp	4.9
0 0 1 1	Test IO	4.10
0 1 0 0	Test Shift	4.8
0 1 0 1	EX(21:23):= BUS(21:23)	
0 1 1 0	BUS(0:23):= 21ext0conEX	
0 1 1 1	BUS(0:23):= IO Data	4.1
1 0 0 0	Divide Floating	4.6
1 0 0 1	BUS(0:23):= BEcon12ext0	
1 0 1 0	BE:= BUS(0:11)	
1 0 1 1	Divide Integer	4.5
1 1 0 0	FR:= BUS(0:11)	
1 1 0 1	BUS(0:23):= 16ext0conPR	
1 1 1 0	BUS(0:23):= 18ext0conITRno(18:22)con0; IR(ITRno):= 0	
1 1 1 1	Not used	

		Reference
MC(51,52,53)		
0 0 0	No operation	
0 0 1	ITRenable:= FR(5)	4.11
0 1 0	IC:= IC+1	
0 1 1	if AR > 0 then IC:= IC+1	
1 0 0	if AR(-1) = 1 then IC:= IC+1	
1 0 1	if AR = 0 then IC:= IC+1	
1 1 0	if AR ≠ 0 then IC:= IC+1	
1 1 1	if -,PROTECT then IC:= IC+1	
MC(54,55,56)		
0 0 0	No operation	
0 0 1	Not applicable	
0 1 0	lshl AF	4.4
0 1 1	lshl ARconBR	4.4
1 0 0	ashr AF	4.4
1 0 1	ashr ARconBR	4.4
1 1 0	lshr AF	4.4
1 1 1	lshr ARconBR	4.4
MC(57)	lshr BF	
MC(58)	BUS(-1:23):= AR(-1:22)con0	
MC(59)	Not used	
MC(60)	Not used	
MC(61)	Read Instruction	4.12
MC(62)	Read Data	4.12
MC(63)	Read Split	4.12
MC(64)	Split Write	4.12
MC(65)	Double	4.12
MC(66)	Not used	

MC(67,68,69,70)					Reference
0	0	0	0	No operation	
0	0	0	1	SB(0:11):= BUS(0:11)	
0	0	1	0	SB(0:11):= BUS(12:23)	
0	0	1	1	SB(0:11):= 12extSB(0)	
0	1	0	0	SB(0:11):= 12extSB(12)	
0	1	0	1	ashr SF	
0	1	1	0	PK:= BUS(21:23)	
0	1	1	1	BUS(0:23):= 21ext0conPK	
1	0	0	0	Test FD Sign	4.6
1	0	0	1	Test WD Sign	4.5
1	0	1	0	IM(1:23):= SB(1:23)	
1	0	1	1	BUS(0:23):= IM	
1	1	0	0	Not used	
1	1	0	1	BUS(0:23):= IR	
1	1	1	0	PR(1:7):= BUS(17:23)	
1	1	1	1	IR:= IR ∧ -,SB	

A P P E N D I X B

CONTENTS:

TABLE OF INSTRUCTIONS

Decimal Value	Mnemonic Code	Binary Value 0 1 2 3 4 5	Instruction
0	aw	0 0 0 0 0 0	autoload word
1	io	0 0 0 0 0 1	input/output
2	bl	0 0 0 0 1 0	load integer byte
3	hl	0 0 0 0 1 1	load half register
4	la	0 0 0 1 0 0	logical and
5	lo	0 0 0 1 0 1	logical or
6	lx	0 0 0 1 1 0	logical exclusive or
7	wa	0 0 0 1 1 1	add integer word
8	ws	0 0 1 0 0 0	subtract integer word
9	am	0 0 1 0 0 1	modify next address
10	wm	0 0 1 0 1 0	multiply integer words
11	al	0 0 1 0 1 1	load address
12	ml	0 0 1 1 0 0	load mask register
13	jl	0 0 1 1 0 1	jump with register link
14	jd	0 0 1 1 1 0	jump with interrupt disable
15	je	0 0 1 1 1 1	jump with interrupt enable
16	xl	0 1 0 0 0 0	load exception register
17	bs	0 1 0 0 0 1	subtract integer byte
18	ba	0 1 0 0 1 0	add integer byte
19	bz	0 1 0 0 1 1	load byte with zeroes
20	rl	0 1 0 1 0 0	load register
21	sp	0 1 0 1 0 1	skip if no protection
22	kl	0 1 0 1 1 0	load protection key
23	rs	0 1 0 1 1 1	store register
24	wd	0 1 1 0 0 0	divide integer word
25	rx	0 1 1 0 0 1	exchange register and store
26	hs	0 1 1 0 1 0	store half register
27	xs	0 1 1 0 1 1	store exception register
28	pl	0 1 1 1 0 0	load protection register
29	ps	0 1 1 1 0 1	store protection register
30	ms	0 1 1 1 1 0	store mask register
31	is	0 1 1 1 1 1	store interrupt register
32	ci	1 0 0 0 0 0	convert integer to floating
33	ac	1 0 0 0 0 1	load address complemented
34	ns	1 0 0 0 1 0	normalize single
35	nd	1 0 0 0 1 1	normalize double
36	as	1 0 0 1 0 0	shift single arithmetically
37	ad	1 0 0 1 0 1	shift double arithmetically
38	ls	1 0 0 1 1 0	shift single logically
39	ld	1 0 0 1 1 1	shift double logically
40	sh	1 0 1 0 0 0	skip if register high
41	sl	1 0 1 0 0 1	skip if register low
42	se	1 0 1 0 1 0	skip if register equal
43	sn	1 0 1 0 1 1	skip if register not equal
44	so	1 0 1 1 0 0	skip if register bits one
45	sz	1 0 1 1 0 1	skip if register bits zero
46	sx	1 0 1 1 1 0	skip if no exceptions
47	ic	1 0 1 1 1 1	clear interrupt bits
48	fa	1 1 0 0 0 0	add floating
49	fs	1 1 0 0 0 1	subtract floating
50	fm	1 1 0 0 1 0	multiply floating
51	ks	1 1 0 0 1 1	store protection key
52	fd	1 1 0 1 0 0	divide floating
53	cf	1 1 0 1 0 1	convert floating to integer
54	dl	1 1 0 1 1 0	load double register
55	ds	1 1 0 1 1 1	store double register
56	aa	1 1 1 0 0 0	add integer double word
57	ss	1 1 1 0 0 1	subtract integer double word
58-63			not used

INDEX FOR XY-NUMBERS AND FLOWCHART PAGE NUMBER

	x0	x1	x2	x3	x4	x6	x8	x9	x12	x16	x17	x20	x24	x28	x31
y0	29	4	5	5	1	28	1	4	10	1		27	1	3	3
y1	1	4	5	5	1	28	1	4	11,15	1	28	14	1	3	5,7
y2	29	4	9	14	3	21	1	3	10	18	29	14	6	10	14
y3	30	4	9	14	18	21	1	31	11	7	28	14	6	30	2,7,31
y4	6	4	15	14	21	26	3	13	10	3	29	3	3	3	6
y5	12	4	15	17	21	26	28	23	11	30	21	3	14	14	5
y6	6	4	6,8	14	2	22,26,27	28	28	10	7	13	14	16	10	6
y7	28	4,6	7	27	2	22,26,27	28	23	11	17,18	21	14	14	14	30
y8	6	4	16	5	14	10	8	30	5	2,7,31	8	10	3	10	6
y9	29	4	16	5	9	11	8		8	30	30	11	3	11	30
y10	6	4	7	7	19	10	1	9	11	7	8	19	17	11	6
y11	30	4	7	2	19	11	1	9		30	30	19	17	30	30
y12	6	4	7	7	19	10	17,18	7	5	26	19	19	18	19	6
y13	29	4	15	17,18	19	11	28	7	8	30	19	19	17,18	19	31
y14	6	7	7	7	16,21	10	17,18	25	11	7	4	21	17,18	11	6
y15	16	21	16	13	16,18,21	11	16	24	29	17,18	13	21	17	3	30,31
y16	20	10	13	13	2	21	25	22,26,27	25	15	5,6,8	29	24	24	3
y17	31	11,15,24,25	22	27	2	16	25	23	25	20	19	14	24	24	5
y18	8	11	13	13	1	21	1	22,26,27	12	15	3	23	19	1	31
y19	29,31	18	31	2	21	16	1	23	21	7	18	23	19	1	30
y20	12	10	13	13	26	26	9	18	12	7	20	16	9	24	6
y21	12	23	26	2	19	19	9	18	20	7	12	24	9	24	5
y22	12	1	13	8	26	26	2	31	12	24,25	28	23	24	10	6
y23	21	26	31	31	21	27	2	31	20	24	30	23	24	23	
y24	22,26	27	19	32	14	10	25	23	25	15	22	23	25	25	6
y25	21	21	19		4	11	8		2	7	22	23	3	13	30
y26	22	21	19	32	12	10	1	23	12	15	22	23	18	30	6
y27	8	21	19		12	11	1	2	12	7,31	22	23	18	13	30
y28	26	19	32	32	12	2	12	23	12	19	30	29	18	4	6
y29	26	19	28	28	12	2	20	8	20	19	30	29	17,18	13	
y30	7	8	32	32	14	10	12	23	12	19	31	23	18	11	6
y31	7	8	31		20	11	20	8	20	19	30	23	18	13	2

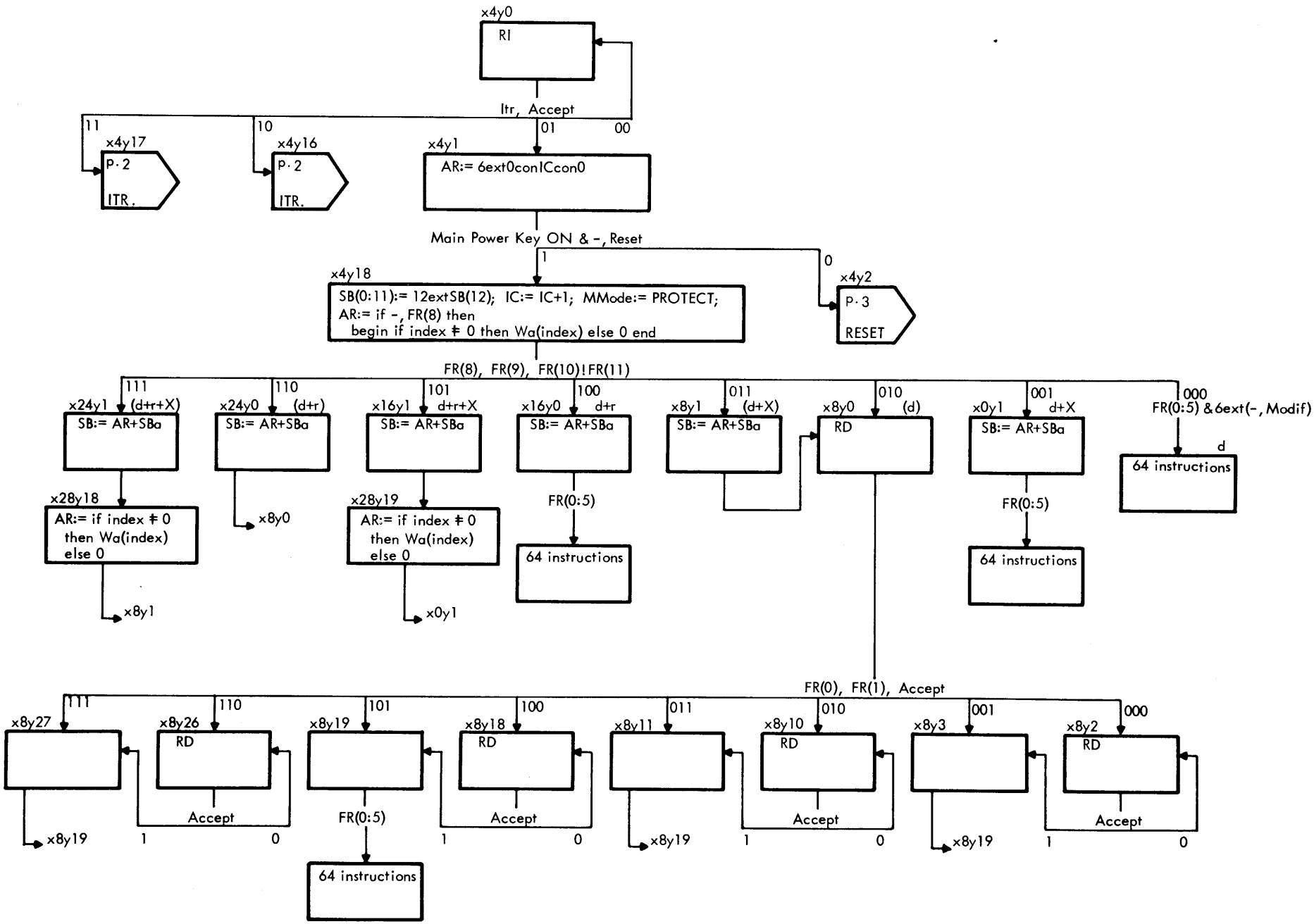
Example: Microinstruction in location x8y14 is found on page 17 and 18.

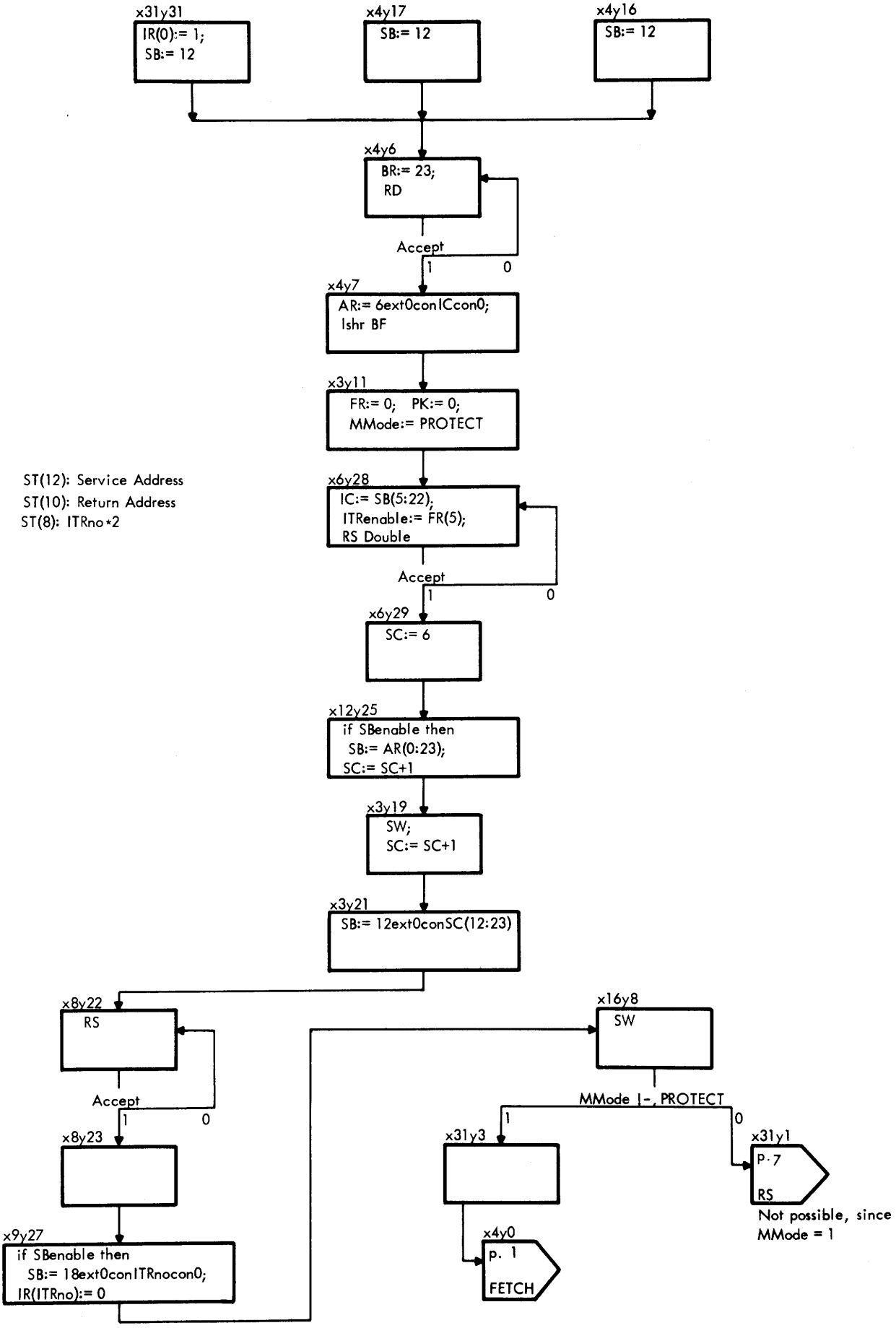
XY-NUMBERS FOR START OF EXECUTE CYCLE

Decimal Value	Mnemonic Code	xy-number	Decimal Value	Mnemonic Code	xy-number
0	aw	x0y0	32	ci	x0y16
1	io	x0y2	33	ac	x0y18
2	bl	x0y4	34	ns	x0y20
3	hl	x0y6	35	nd	x0y22
4	la	x1y0	36	as	x1y16
5	lo	x1y2	37	ad	x1y18
6	lx	x1y4	38	ls	x1y20
7	wa	x1y6	39	ld	x1y22
8	ws	x2y0	40	sh	x2y16
9	am	x2y2	41	sl	x2y18
10	wm	x2y4	42	se	x2y20
11	al	x2y6	43	sn	x2y22
12	ml	x3y0	44	so	x3y16
13	jl	x3y2	45	sz	x3y18
14	jd	x3y4	46	sx	x3y20
15	je	x3y6	47	ic	x3y22
16	xl	x0y8	48	fa	x0y24
17	bs	x0y10	49	fs	x0y26
18	ba	x0y12	50	fm	x0y28
19	bz	x0y14	51	ks	x0y30
20	rl	x1y8	52	fd	x1y24
21	sp	x1y10	53	cf	x1y26
22	kl	x1y12	54	dl	x1y28
23	rs	x1y14	55	ds	x1y30
24	wd	x2y8	56	aa	x2y24
25	rx	x2y10	57	ss	x2y26
26	hs	x2y12	58		x2y28
27	xs	x2y14	59		x2y30
28	pl	x3y8	60		x3y24
29	ps	x3y10	61		x3y26
30	ms	x3y12	62		x3y28
31	is	x3y14	63		x3y30

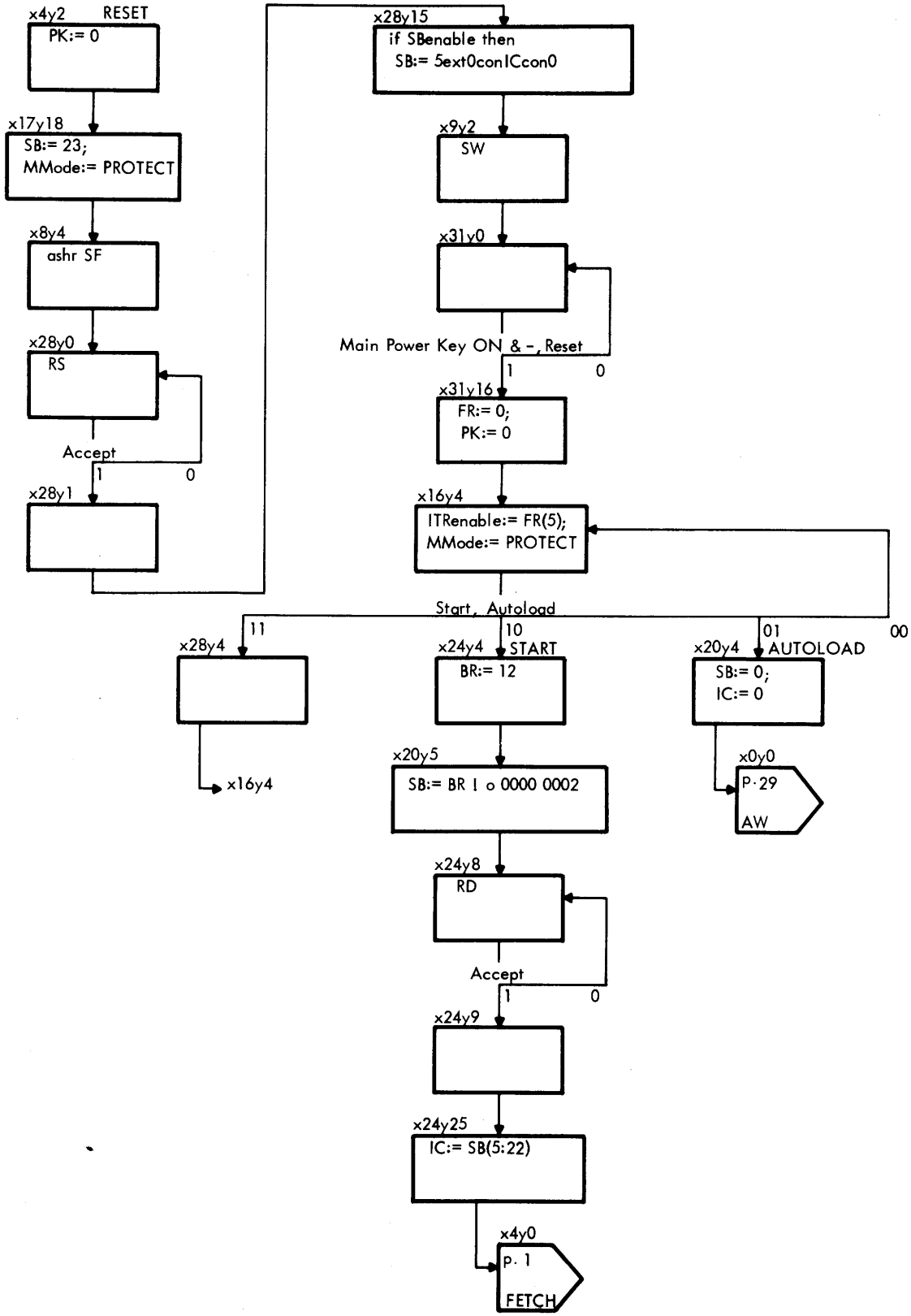
JUMP CONDITIONS

x					y				
16	8	4	2	1	16	8	4	2	1
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
73	76	79	82	85	88	91	94	97	100
SB $\leq$ -65	SB $\geq$ 64		FR(2) & -, Modif	FR(3) & -, Modif	FR(0) & -, Modif	FR(1) & -, Modif	FR(4) & -, Modif	FR(5) & -, Modif	AF $\neq$ 0
72	75	78	81	84	87	90	93	96	99
FR(8)	FR(9)		BE(11)	FDsub	FR(0)	AR $\neq$ 0	SC(11)	SC $\neq$ 0	FR(10) ! FR(11)
72 73	75 76	78 79	81 82	84 85	87 88	90 91	93 94	96 97	99 100
AR(-1) = AR(0)	AR(1) = AR(2)	AR(0) = AR(1)	FR(2)	FR(3)	Itr	FR(1)	FR(4)	FR(5)	Accept
71	74	77	80	83	86	89	92	95	98
Carry(0)	BR(1) = BR(2)		BE(10)	EX(22, 23) $\neq$ 0	HA(23)	BR(23)	SB(0)	SB $\neq$ 0	BE(0)
71 73	74 76	77 79	80 82	83 85	86 88	89 91	92 94	95 97	98 100
		FR(6) ! FR(7)			AR(-1)	BR(22)	MMode	MMode ! -, PROTECT	Round
71 72	74 75	77 78	80 81	83 84	86 87	89 90	92 93	95 96	98 99
	Start	Autoload			Main Power Key ON & -, Reset	SC > -38 & SC < 38			
71 72 73	74 75 76	77 78 79	80 81 82	83 84 85	86 87 88	89 90 91	92 93 94	95 96 97	98 99 100
MAR(0)	MAR(1)	MAR(2)	MAR(3)	MAR(4)	MAR(5)	MAR(6)	MAR(7)	MAR(8)	MAR(9)

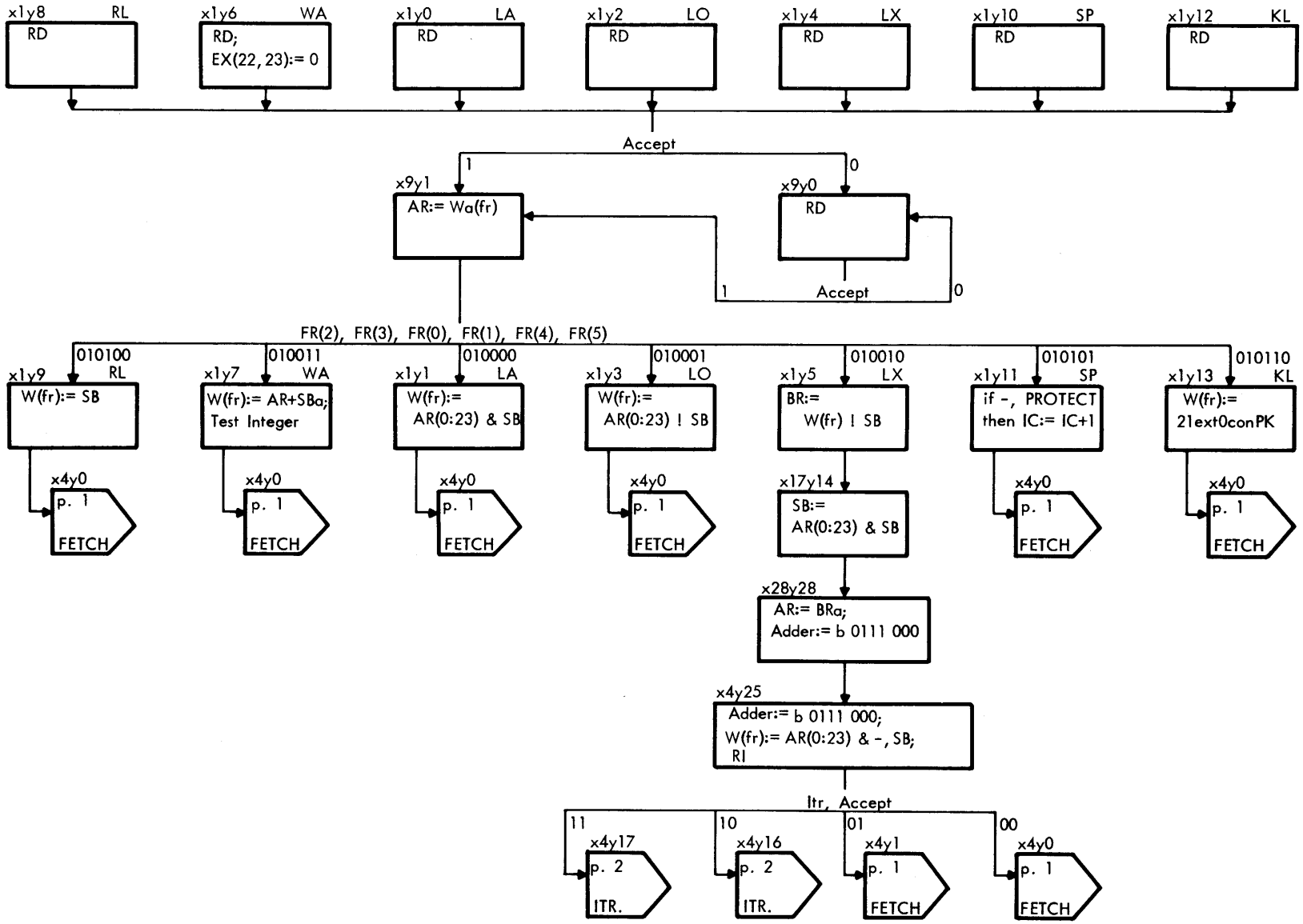

x4y0
RI
Itr, Accept
11
10
01
00
x4y17
P·2
ITR.
x4y16
P·2
ITR.
x4y1
AR:= 6ext0con1Ccon0
Main Power Key ON & -, Reset
1
0
x4y18
SB(0:11):= 12extSB(12); IC:= IC+1; MMode:= PROTECT;
AR:= if -, FR(8) then
begin if index ≠ 0 then Wa(index) else 0 end
x4y2
P·3
RESET
FR(8), FR(9), FR(10)!FR(11)
111 (d+r+X)
110 (d+r)
101 d+r+X
100 d+r
011 (d+X)
010 (d)
001 d+X
000
FR(0:5) &6ext(-, Modif)
x24y1
SB:= AR+SBa
x24y0
SB:= AR+SBa
x16y1
SB:= AR+SBa
x16y0
SB:= AR+SBa
x8y1
SB:= AR+SBa
x8y0
RD
x0y1
SB:= AR+SBa
d
64 instructions
x28y18
AR:= if index ≠ 0
then Wa(index)
else 0
x8y0
x28y19
AR:= if index ≠ 0
then Wa(index)
else 0
FR(0:5)
64 instructions
FR(0:5)
64 instructions
x8y1
x0y1
FR(0), FR(1), Accept
111
110
101
100
011
010
001
000
x8y27
x8y26
RD
x8y19
x8y18
RD
x8y11
x8y10
RD
x8y3
x8y2
RD
Accept
FR(0:5)
Accept
Accept
Accept
x8y19
1
0
64 instructions
x8y19
1
0
x8y19
1
0
x8y19
1
0

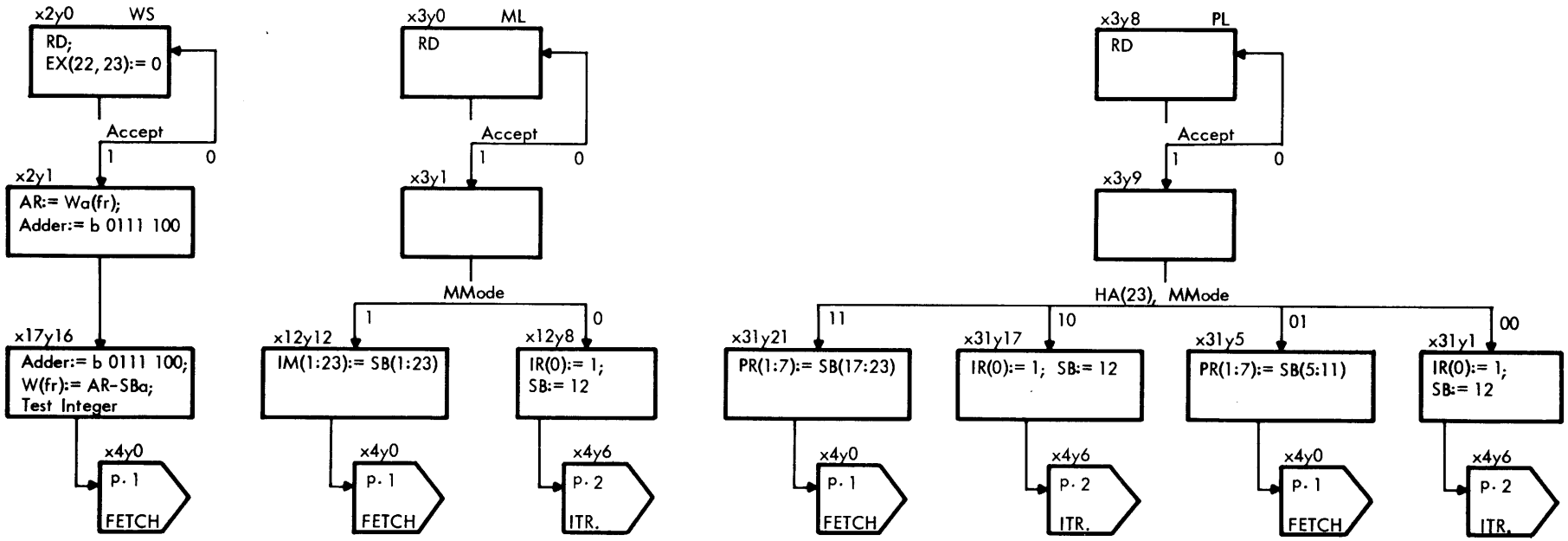


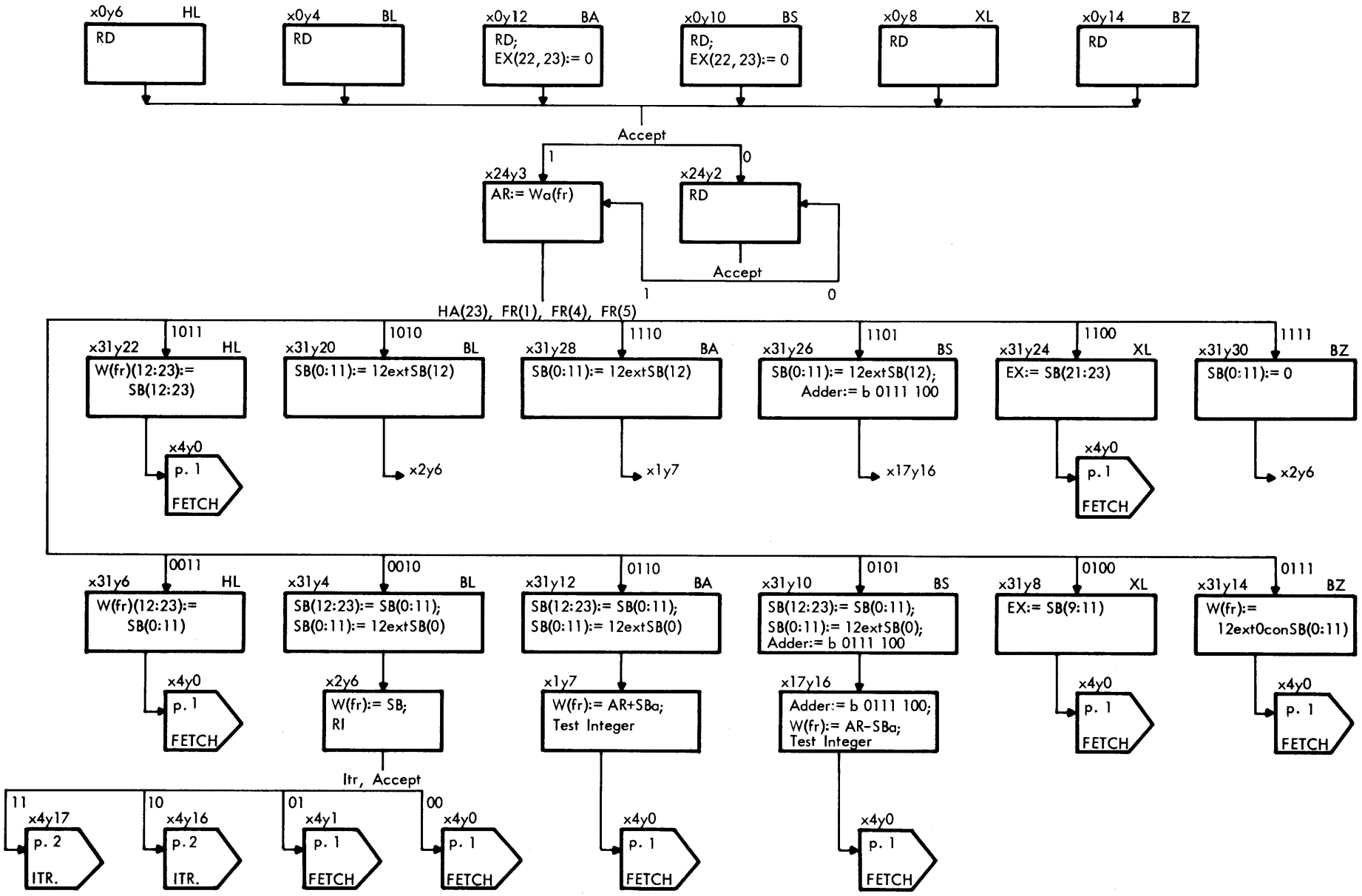
220768AG 101069HA 261169BRJ 261169AG

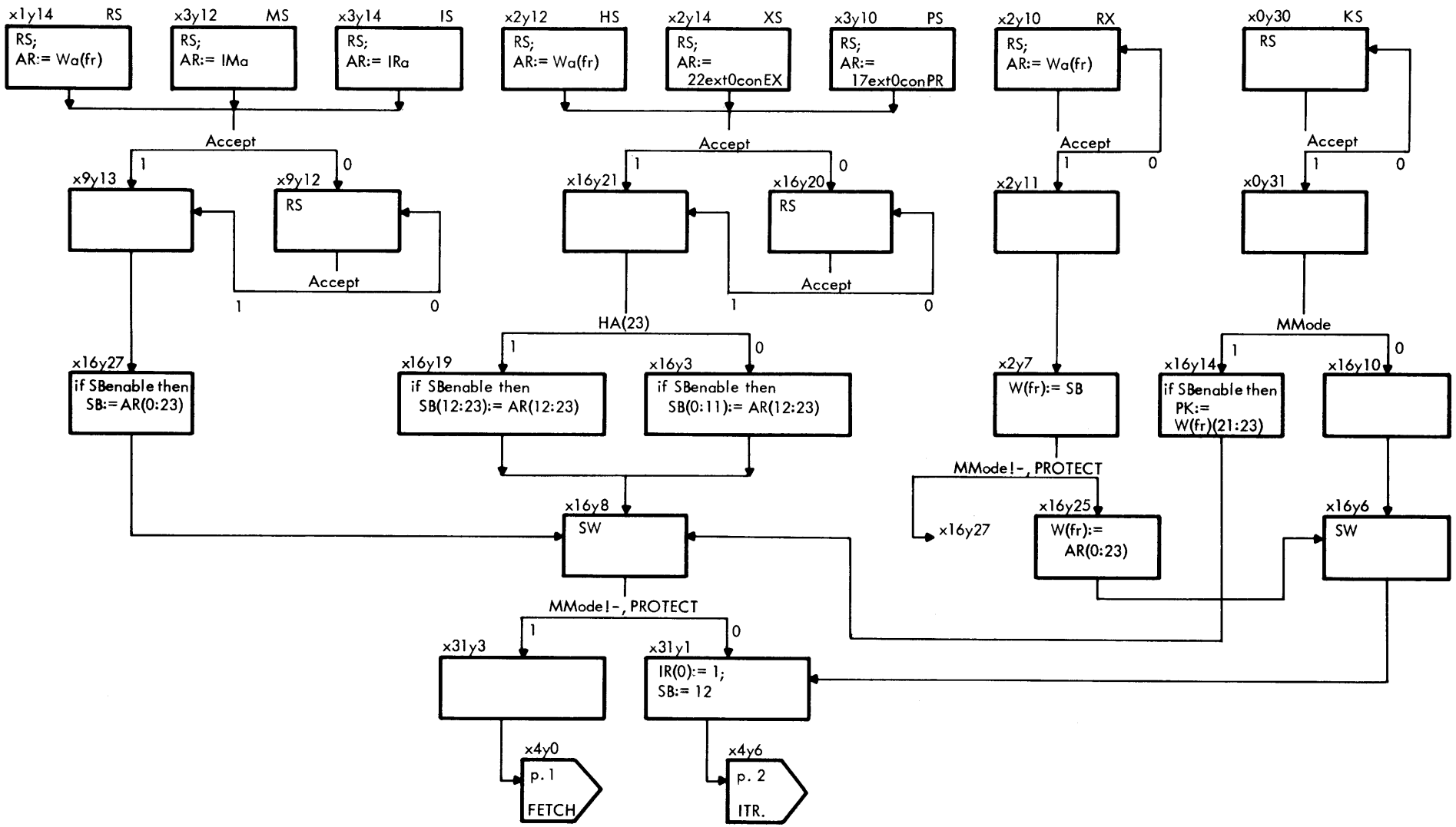


220768AG 101069HA 261169BRJ 261169AG

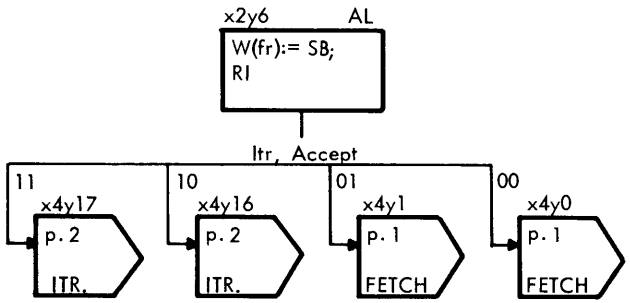

x1y8 RL
RD
x1y6 WA
RD;
EX(22, 23):= 0
x1y0 LA
RD
x1y2 LO
RD
x1y4 LX
RD
x1y10 SP
RD
x1y12 KL
RD
Accept
1
0
x9y1
AR:= Wa(fr)
x9y0
RD
1
Accept
0
FR(2), FR(3), FR(0), FR(1), FR(4), FR(5)
010100
x1y9 RL
W(fr):= SB
010011
x1y7 WA
W(fr):= AR+SBa;
Test Integer
010000
x1y1 LA
W(fr):=
AR(0:23) & SB
010001
x1y3 LO
W(fr):=
AR(0:23) ! SB
010010
x1y5 LX
BR:=
W(fr) ! SB
010101
x1y11 SP
if -, PROTECT
then IC:= IC+1
010110
x1y13 KL
W(fr):=
21ext0conPK
x4y0
p. 1
FETCH
x17y14
SB:=
AR(0:23) & SB
x28y28
AR:= BRa;
Adder:= b 0111 000
x4y25
Adder:= b 0111 000;
W(fr):= AR(0:23) & -, SB;
RI
Itr, Accept
11
x4y17
p. 2
ITR.
10
x4y16
p. 2
ITR.
01
x4y1
p. 1
FETCH
00
x4y0
p. 1
FETCH

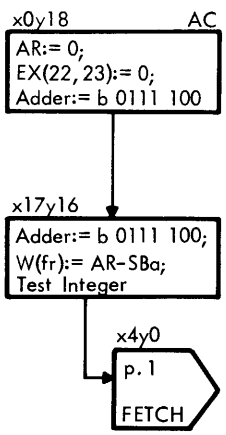

x2y0 WS
RD;
EX(22, 23):= 0
Accept
1
0
x2y1
AR:= Wa(fr);
Adder:= b 0111 100
x17y16
Adder:= b 0111 100;
W(fr):= AR-SBa;
Test Integer
x4y0
P. 1
FETCH
x3y0 ML
RD
Accept
1
0
x3y1
MMode
1
0
x12y12
IM(1:23):= SB(1:23)
x4y0
P. 1
FETCH
x12y8
IR(0):= 1;
SB:= 12
x4y6
P. 2
ITR.
x3y8 PL
RD
Accept
1
0
x3y9
HA(23), MMode
11
10
01
00
x31y21
PR(1:7):= SB(17:23)
x4y0
P. 1
FETCH
x31y17
IR(0):= 1; SB:= 12
x4y6
P. 2
ITR.
x31y5
PR(1:7):= SB(5:11)
x4y0
P. 1
FETCH
x31y1
IR(0):= 1;
SB:= 12
x4y6
P. 2
ITR.

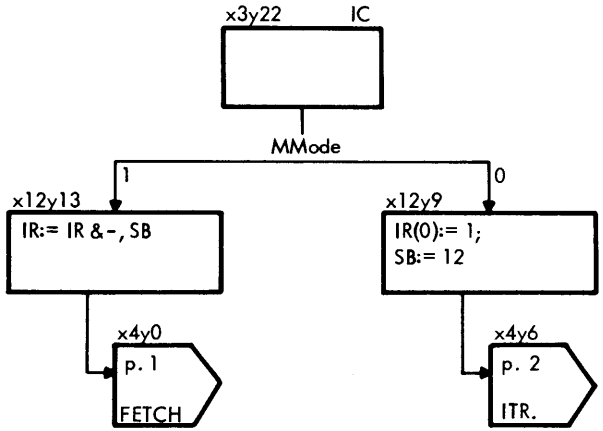

x0y6 HL
RD
x0y4 BL
RD
x0y12 BA
RD;
EX(22, 23):= 0
x0y10 BS
RD;
EX(22, 23):= 0
x0y8 XL
RD
x0y14 BZ
RD
Accept
1
0
x24y3
AR:= Wa(fr)
x24y2
RD
Accept
1
0
HA(23), FR(1), FR(4), FR(5)
1011
x31y22 HL
W(fr)(12:23):=
SB(12:23)
x4y0
p. 1
FETCH
1010
x31y20 BL
SB(0:11):= 12extSB(12)
x2y6
1110
x31y28 BA
SB(0:11):= 12extSB(12)
x1y7
1101
x31y26 BS
SB(0:11):= 12extSB(12);
Adder:= b 0111 100
x17y16
1100
x31y24 XL
EX:= SB(21:23)
x4y0
p. 1
FETCH
1111
x31y30 BZ
SB(0:11):= 0
x2y6
0011
x31y6 HL
W(fr)(12:23):=
SB(0:11)
x4y0
p. 1
FETCH
0010
x31y4 BL
SB(12:23):= SB(0:11);
SB(0:11):= 12extSB(0)
x2y6
W(fr):= SB;
RI
Itr, Accept
11
x4y17
p. 2
ITR.
10
x4y16
p. 2
ITR.
01
x4y1
p. 1
FETCH
00
x4y0
p. 1
FETCH
0110
x31y12 BA
SB(12:23):= SB(0:11);
SB(0:11):= 12extSB(0)
x1y7
W(fr):= AR+SBa;
Test Integer
x4y0
p. 1
FETCH
0101
x31y10 BS
SB(12:23):= SB(0:11);
SB(0:11):= 12extSB(0);
Adder:= b 0111 100
x17y16
Adder:= b 0111 100;
W(fr):= AR-SBa;
Test Integer
x4y0
p. 1
FETCH
0100
x31y8 XL
EX:= SB(9:11)
x4y0
p. 1
FETCH
0111
x31y14 BZ
W(fr):=
12ext0conSB(0:11)
x4y0
p. 1
FETCH

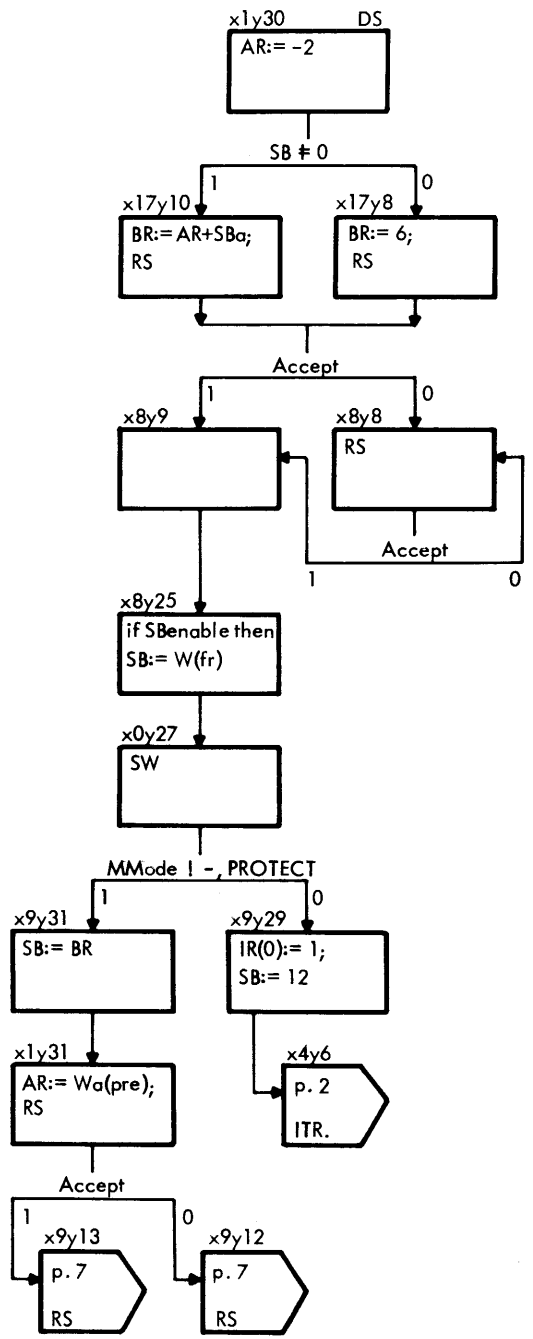

x1y14 RS
RS;
AR:= Wa(fr)
x3y12 MS
RS;
AR:= IMa
x3y14 IS
RS;
AR:= IRa
x2y12 HS
RS;
AR:= Wa(fr)
x2y14 XS
RS;
AR:=
22ext0conEX
x3y10 PS
RS;
AR:=
17ext0conPR
x2y10 RX
RS;
AR:= Wa(fr)
x0y30 KS
RS
Accept
1
0
x9y13
x9y12
RS
x16y21
x16y20
RS
x2y11
x0y31
HA(23)
MMode
x16y27
if SBenable then
SB:=AR(0:23)
x16y19
if SBenable then
SB(12:23):= AR(12:23)
x16y3
if SBenable then
SB(0:11):= AR(12:23)
x2y7
W(fr):= SB
x16y14
if SBenable then
PK:=
W(fr)(21:23)
x16y10
MMode!-, PROTECT
x16y8
SW
x16y25
W(fr):=
AR(0:23)
x16y6
SW
x31y3
x31y1
IR(0):= 1;
SB:= 12
x4y0
p. 1
FETCH
x4y6
p. 2
ITR.

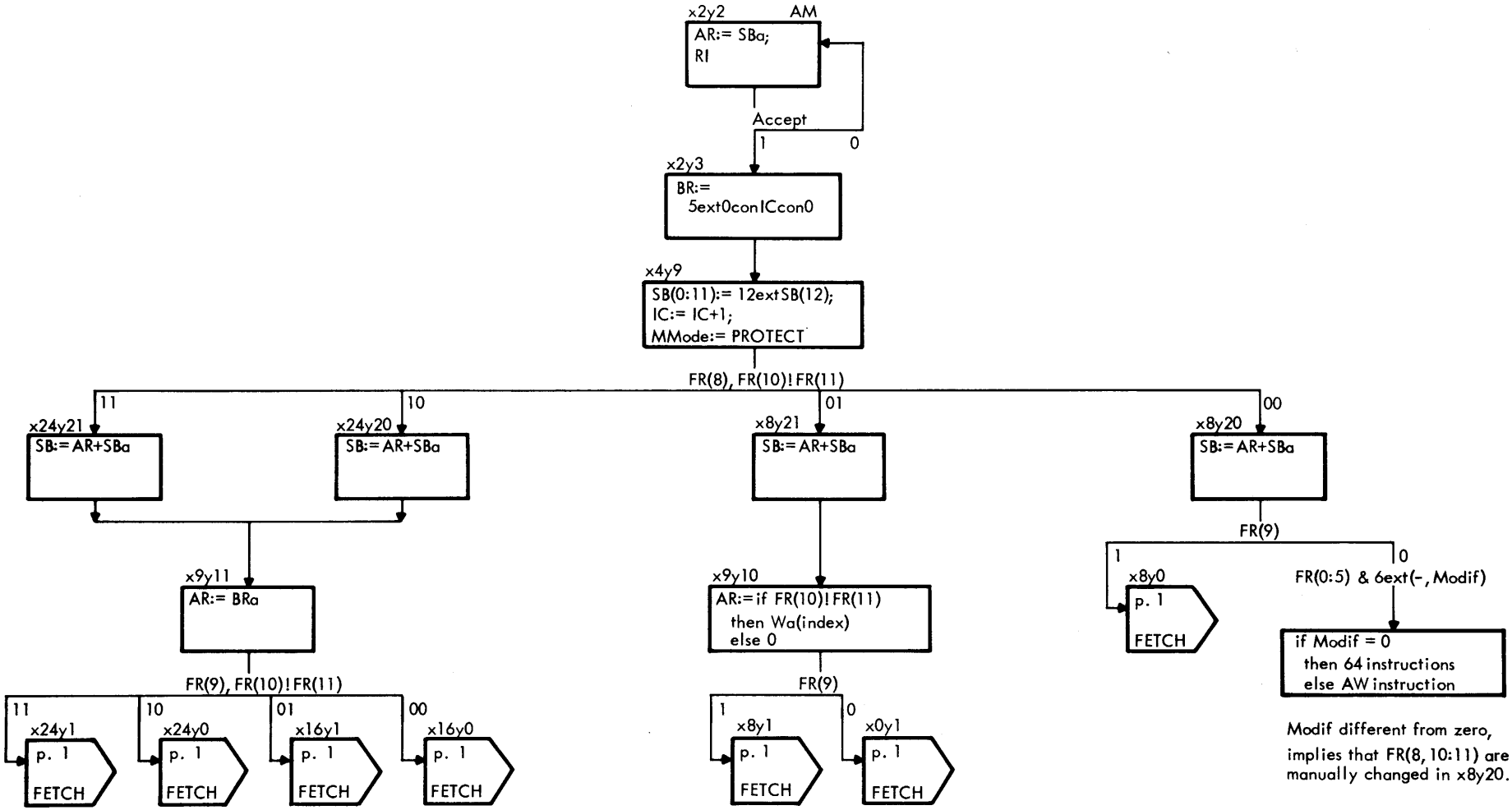
220768AG 221069BRJ 261169BRJ 261169AG

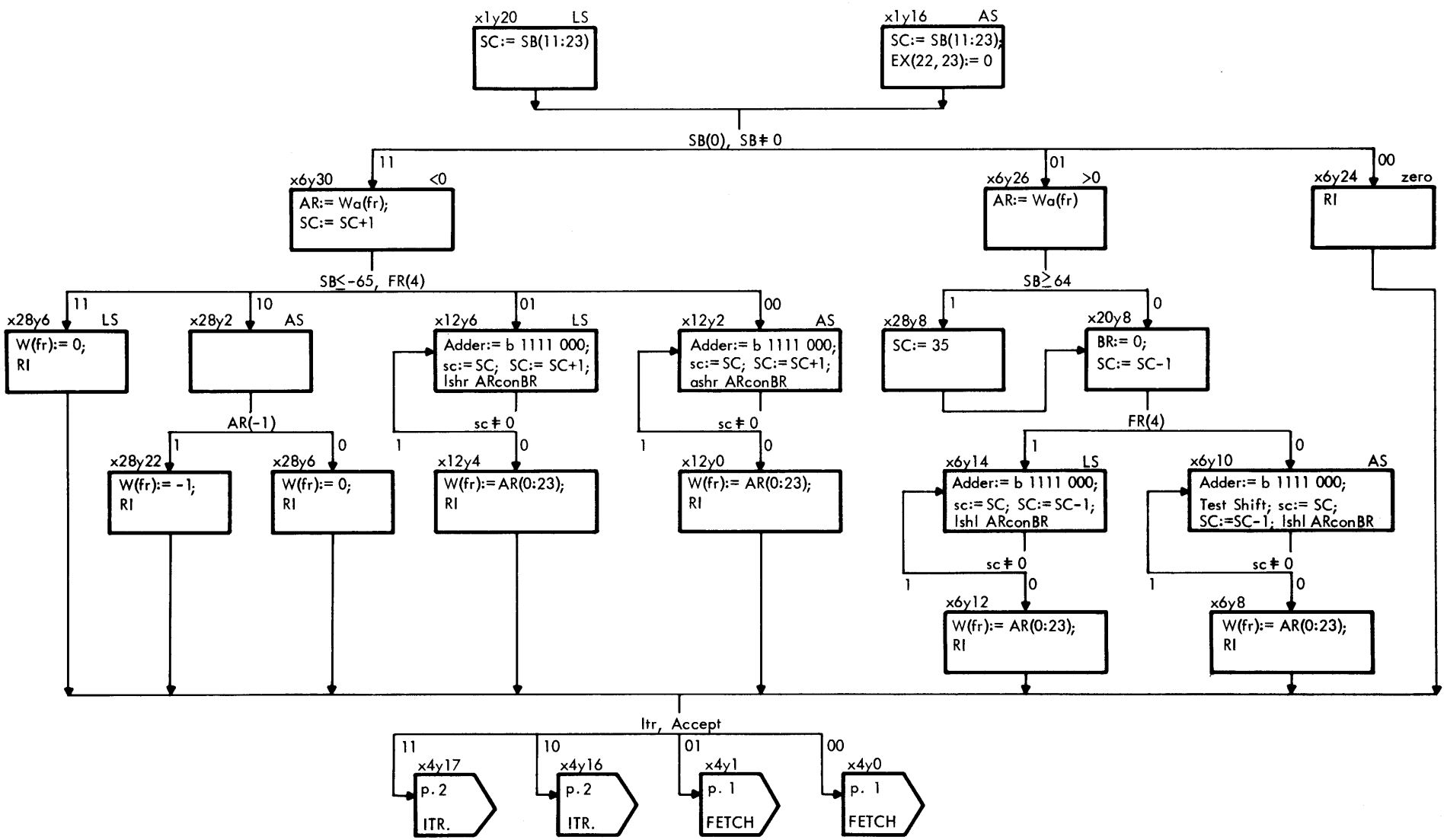


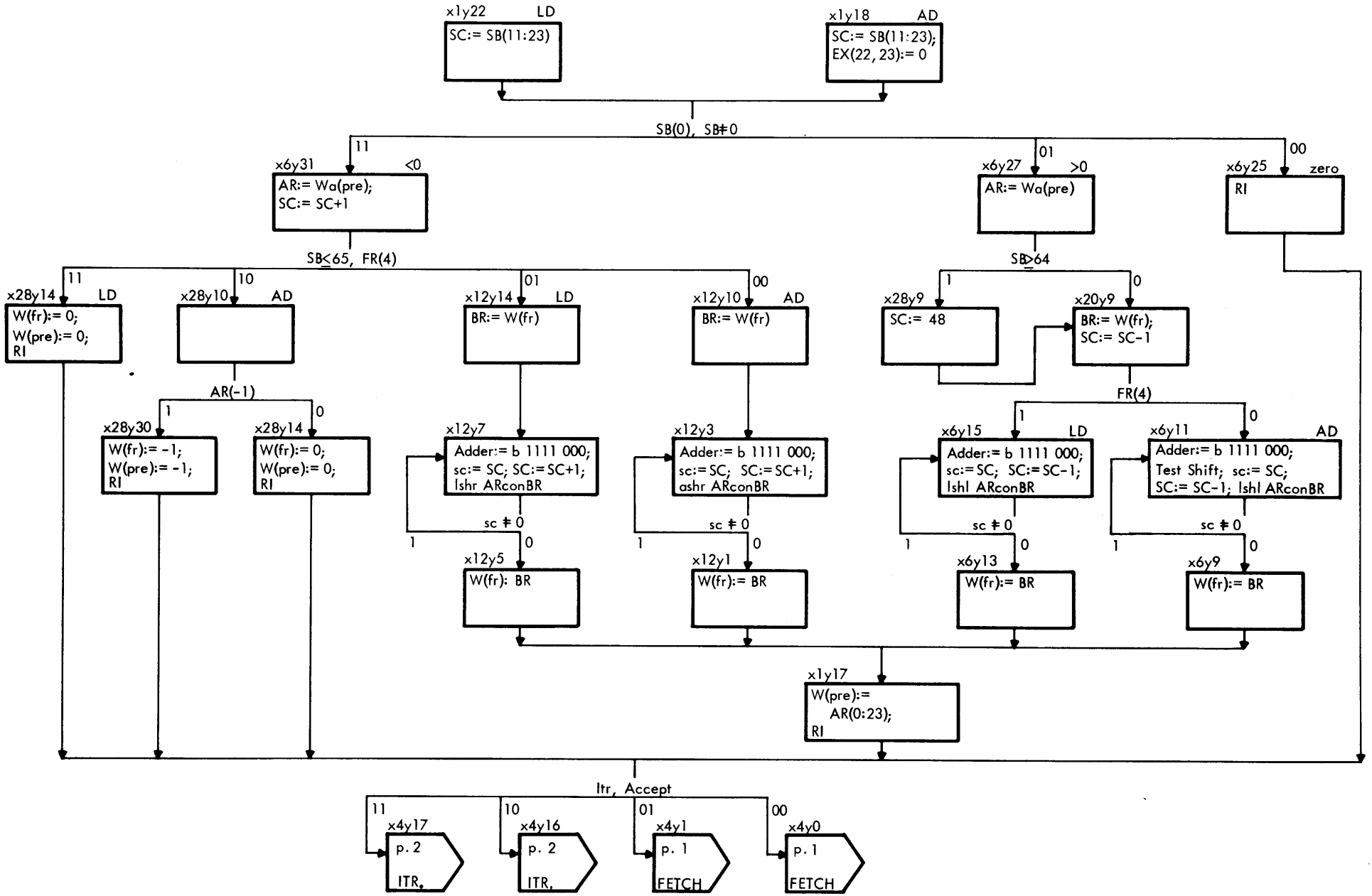


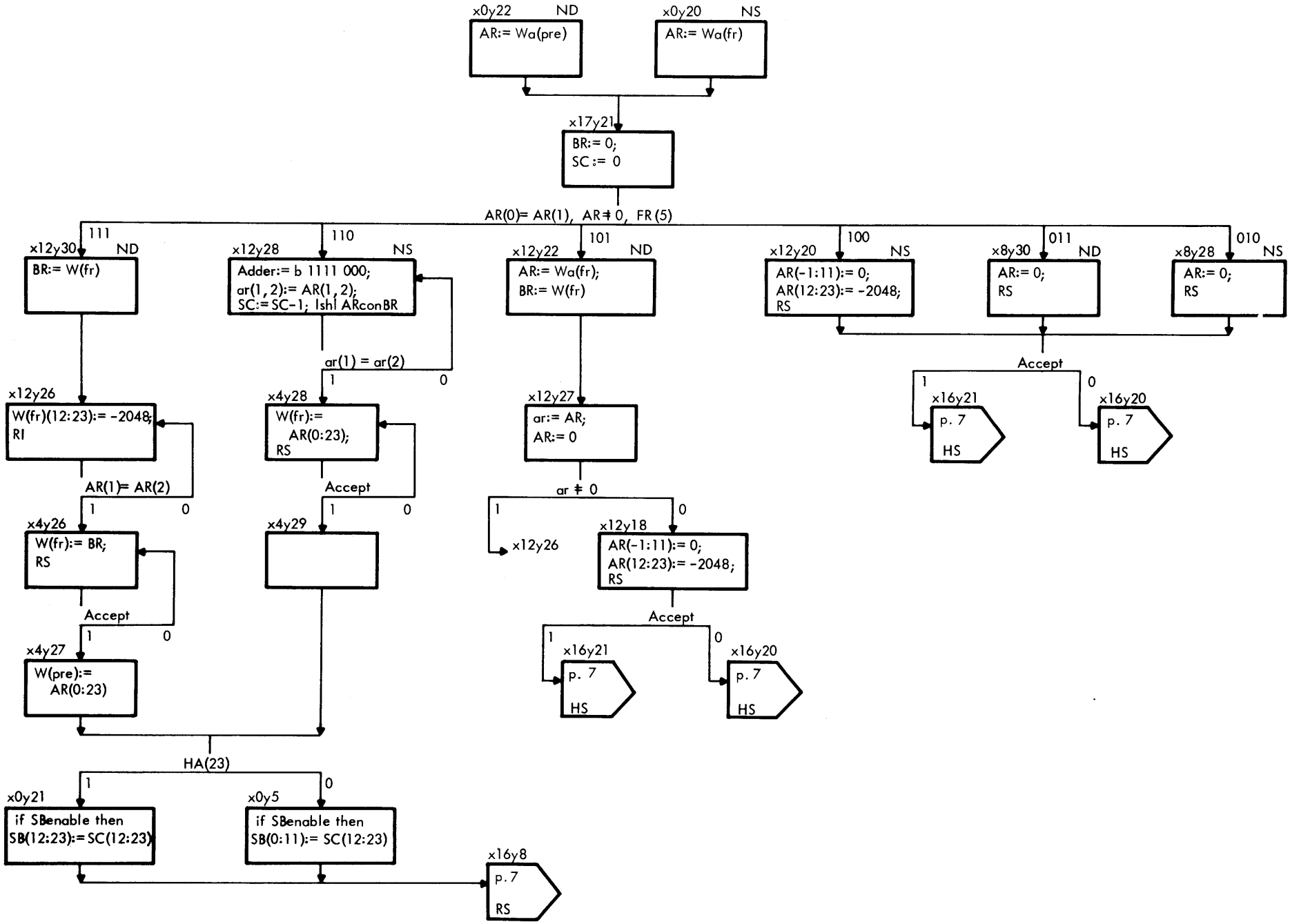


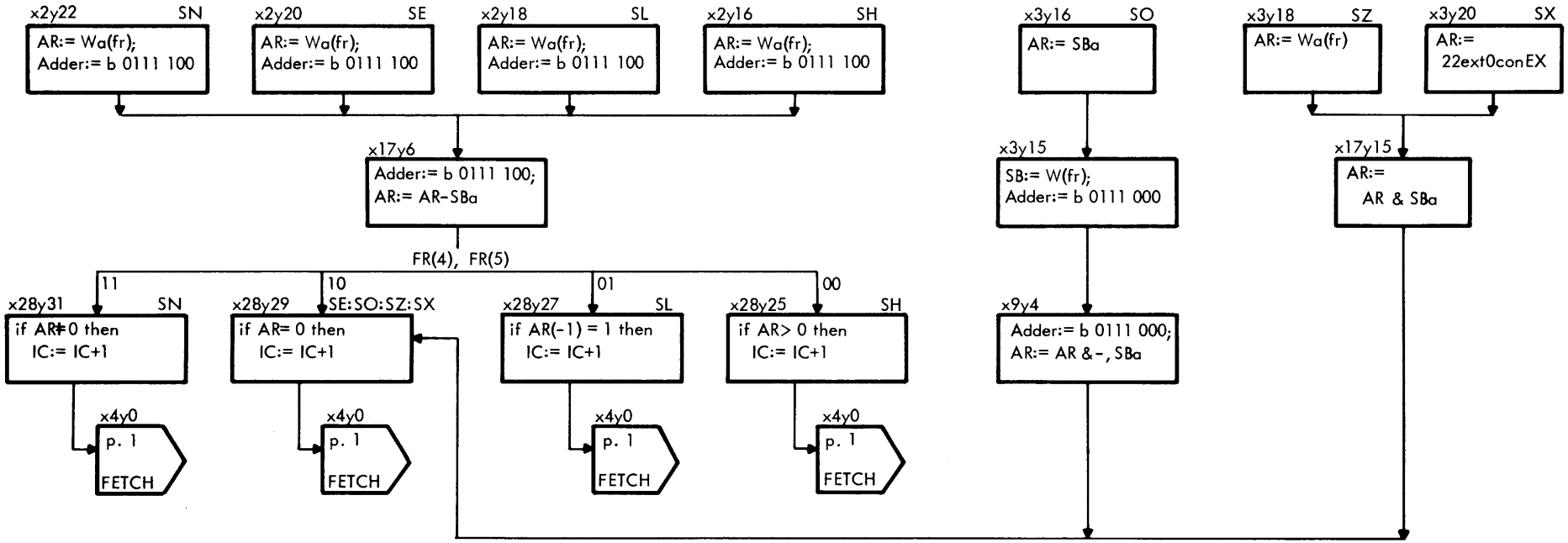


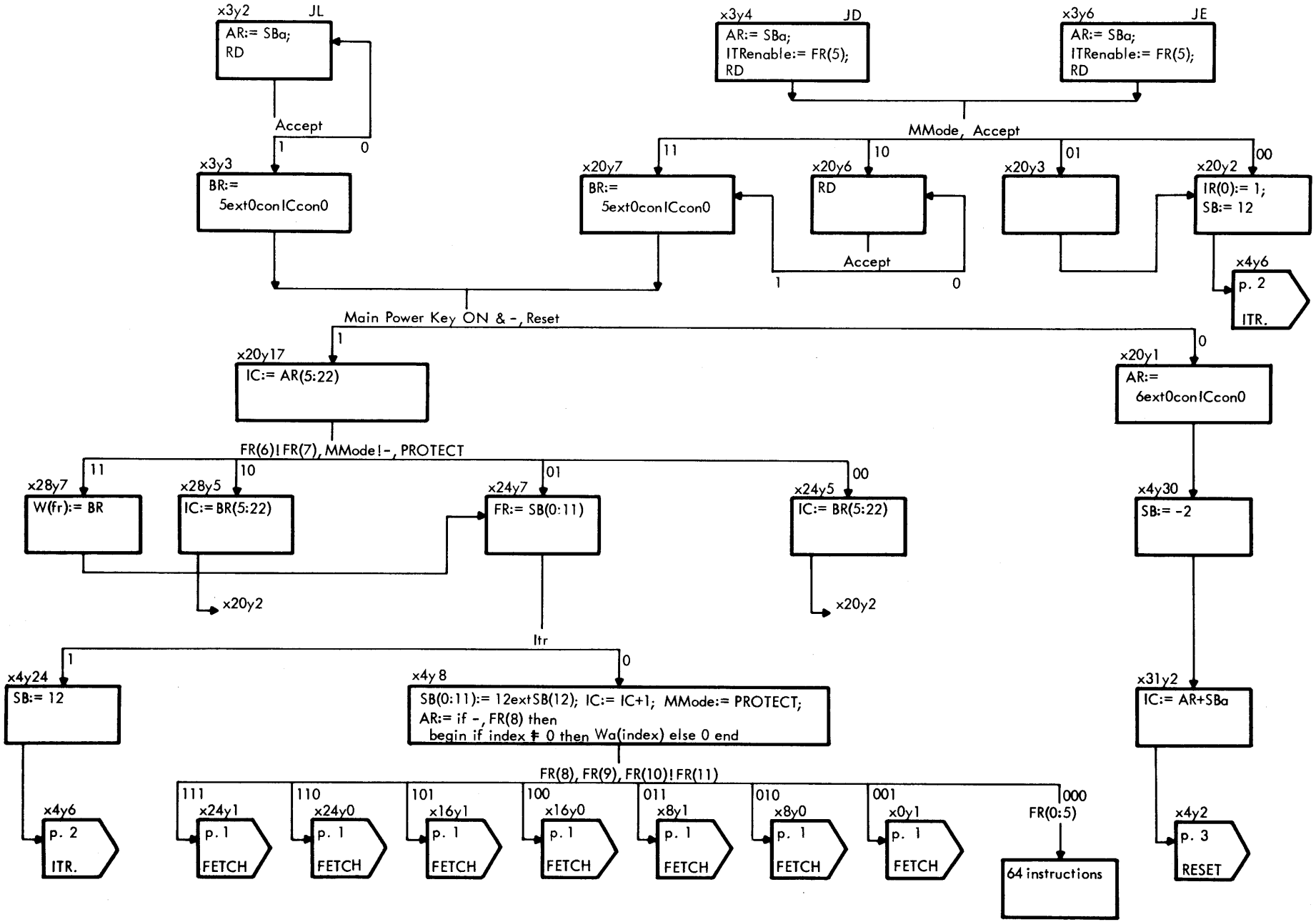

x2y2 AM
AR:= SBa;
RI
Accept
1
0
x2y3
BR:=
5ext0con1Ccon0
x4y9
SB(0:11):= 12extSB(12);
IC:= IC+1;
MMode:= PROTECT
FR(8), FR(10)!FR(11)
11
10
01
00
x24y21
SB:=AR+SBa
x24y20
SB:=AR+SBa
x8y21
SB:=AR+SBa
x8y20
SB:=AR+SBa
x9y11
AR:= BRa
x9y10
AR:=if FR(10)!FR(11)
then Wa(index)
else 0
FR(9)
1
0
x8y0
p. 1
FETCH
FR(0:5) & 6ext(-, Modif)
if Modif = 0
then 64 instructions
else AW instruction
FR(9), FR(10)!FR(11)
11
10
01
00
x24y1
x24y0
x16y1
x16y0
p. 1
FETCH
FR(9)
1
0
x8y1
x0y1
p. 1
FETCH
Modif different from zero,
implies that FR(8, 10:11) are
manually changed in x8y20.

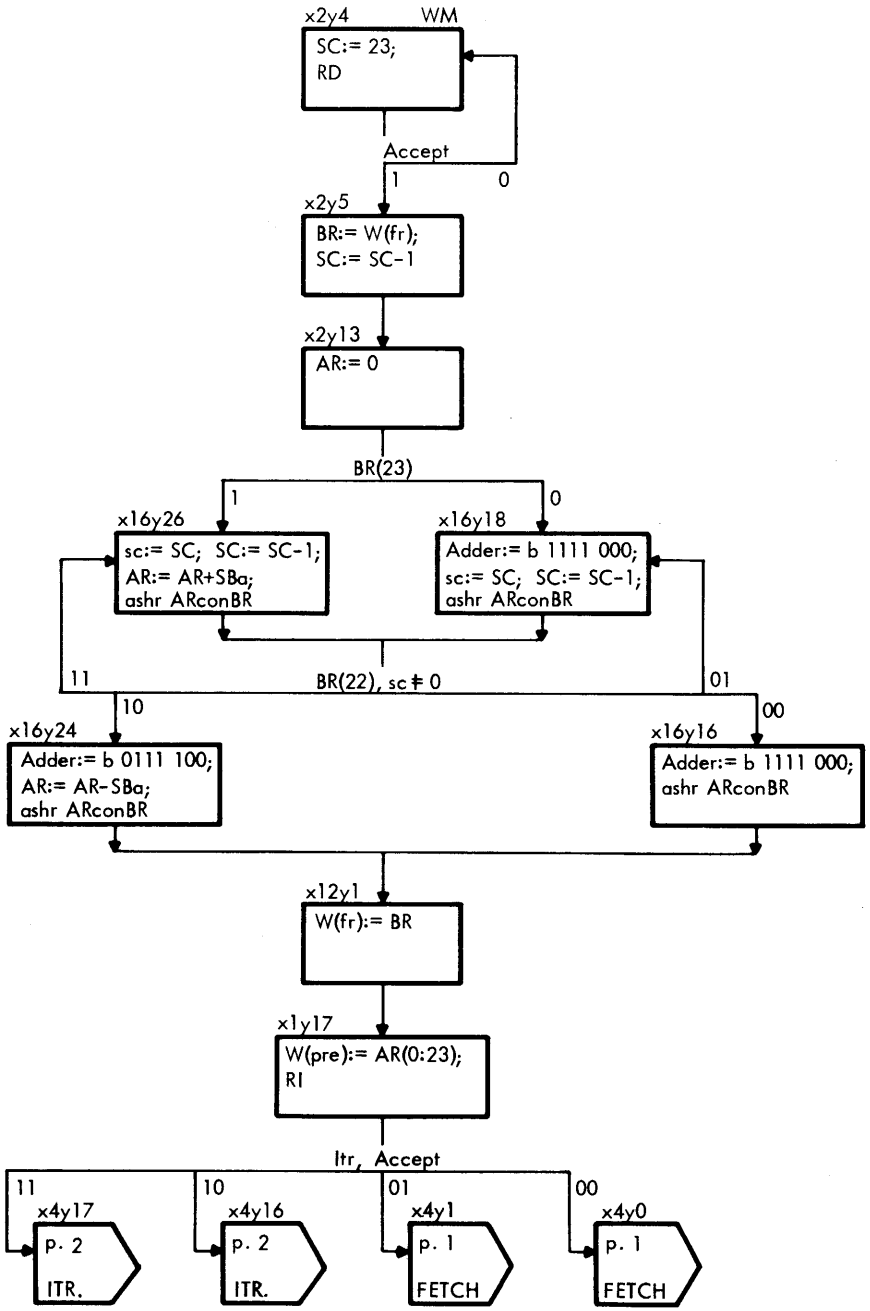

x1y20 LS
SC:= SB(11:23)
x1y16 AS
SC:= SB(11:23);
EX(22, 23):= 0
SB(0), SB≠ 0
11
x6y30 <0
AR:= Wa(fr);
SC:= SC+1
01
x6y26 >0
AR:= Wa(fr)
00
x6y24 zero
RI
SB≤-65, FR(4)
SB≥64
x28y6 LS
W(fr):= 0;
RI
x28y2 AS
x12y6 LS
Adder:= b 1111 000;
sc:=SC; SC:= SC+1;
lshr ARconBR
x12y2 AS
Adder:= b 1111 000;
sc:=SC; SC:=SC+1;
ashr ARconBR
x28y8
SC:= 35
x20y8
BR:= 0;
SC:= SC-1
AR(-1)
sc≠ 0
FR(4)
x28y22
W(fr):= -1;
RI
x28y6
W(fr):= 0;
RI
x12y4
W(fr):=AR(0:23);
RI
x12y0
W(fr):= AR(0:23);
RI
x6y14 LS
Adder:= b 1111 000;
sc:=SC; SC:=SC-1;
lshl ARconBR
x6y10 AS
Adder:= b 1111 000;
Test Shift; sc:= SC;
SC:=SC-1; lshl ARconBR
x6y12
W(fr):= AR(0:23);
RI
x6y8
W(fr):= AR(0:23);
RI
Itr, Accept
x4y17
p. 2
ITR.
x4y16
p. 2
ITR.
x4y1
p. 1
FETCH
x4y0
p. 1
FETCH

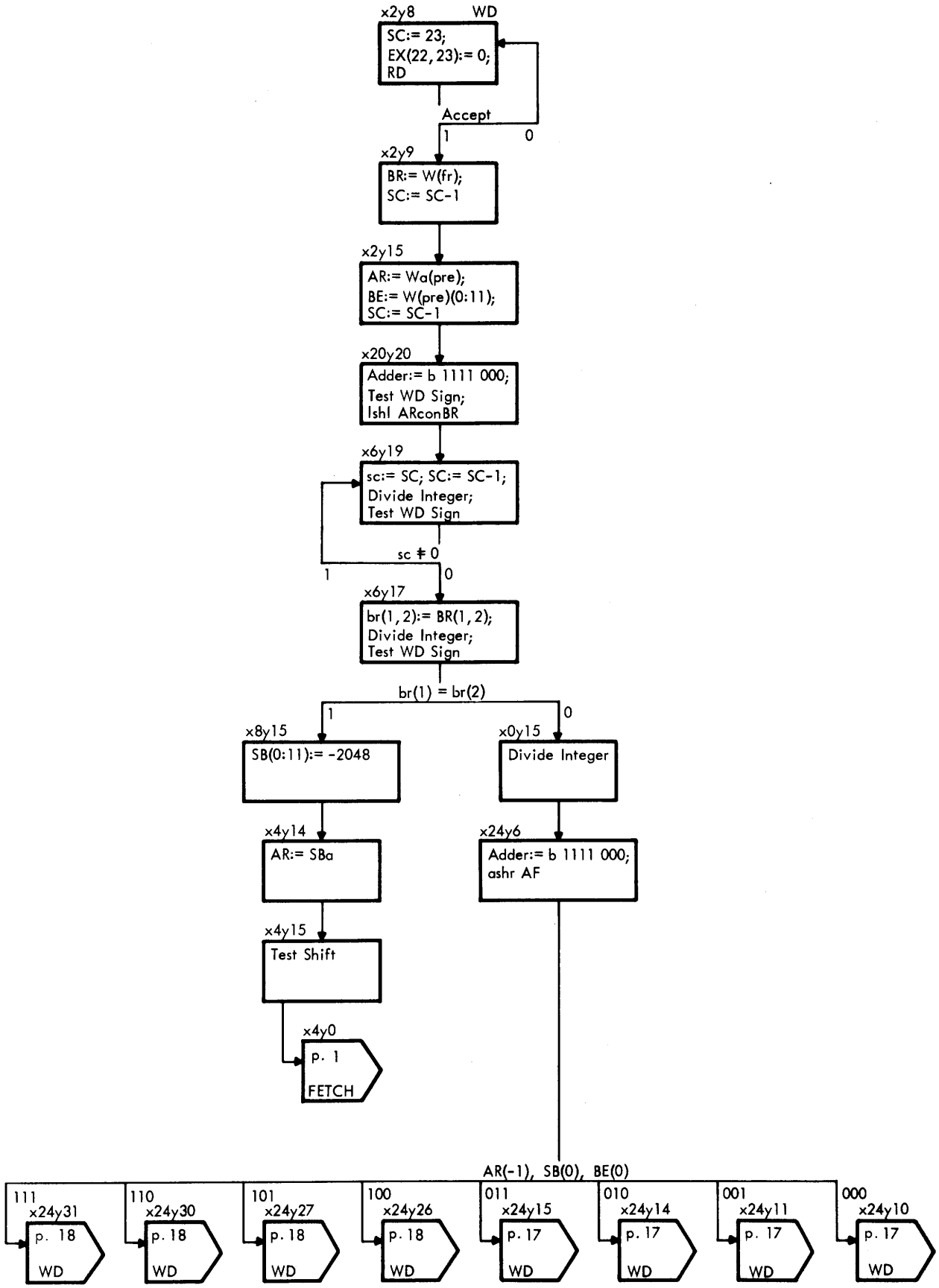

x1y22 LD
SC:= SB(11:23)
x1y18 AD
SC:= SB(11:23);
EX(22, 23):= 0
SB(0), SB≠0
11
x6y31 <0
AR:= Wa(pre);
SC:= SC+1
01
x6y27 >0
AR:= Wa(pre)
00
x6y25 zero
RI
SB≤65, FR(4)
SB≥64
11
x28y14 LD
W(fr):= 0;
W(pre):= 0;
RI
10
x28y10 AD
01
x12y14 LD
BR:= W(fr)
00
x12y10 AD
BR:= W(fr)
1
x28y9
SC:= 48
0
x20y9
BR:= W(fr);
SC:= SC-1
AR(-1)
1
x28y30
W(fr):= -1;
W(pre):= -1;
RI
0
x28y14
W(fr):= 0;
W(pre):= 0;
RI
x12y7
Adder:= b 1111 000;
sc:= SC; SC:= SC+1;
lshr ARconBR
sc ≠ 0
1
0
x12y5
W(fr): BR
x12y3
Adder:= b 1111 000;
sc:= SC; SC:= SC+1;
ashr ARconBR
sc ≠ 0
1
0
x12y1
W(fr):= BR
FR(4)
1
x6y15 LD
Adder:= b 1111 000;
sc:= SC; SC:= SC-1;
lshl ARconBR
sc ≠ 0
1
0
x6y13
W(fr):= BR
0
x6y11 AD
Adder:= b 1111 000;
Test Shift; sc:= SC;
SC:= SC-1; lshl ARconBR
sc ≠ 0
1
0
x6y9
W(fr):= BR
x1y17
W(pre):=
AR(0:23);
RI
Itr, Accept
11
x4y17
p. 2
ITR.
10
x4y16
p. 2
ITR.
01
x4y1
p. 1
FETCH
00
x4y0
p. 1
FETCH


x0y22 ND
AR:= Wa(pre)
x0y20 NS
AR:= Wa(fr)
x17y21
BR:= 0;
SC:= 0
AR(0)= AR(1), AR≠ 0, FR(5)
111
x12y30 ND
BR:= W(fr)
110
x12y28 NS
Adder:= b 1111 000;
ar(1,2):= AR(1,2);
SC:=SC-1; lshl ARconBR
101
x12y22 ND
AR:= Wa(fr);
BR:= W(fr)
100
x12y20 NS
AR(-1:11):= 0;
AR(12:23):= -2048;
RS
011
x8y30 ND
AR:= 0;
RS
010
x8y28 NS
AR:= 0;
RS
Accept
1
x16y21
p. 7
HS
0
x16y20
p. 7
HS
ar(1) = ar(2)
1
0
x12y26
W(fr)(12:23):= -2048;
RI
x4y28
W(fr):=
AR(0:23);
RS
x12y27
ar:= AR;
AR:= 0
AR(1)= AR(2)
1
0
Accept
1
0
ar ≠ 0
1
0
x4y26
W(fr):= BR;
RS
x4y29
x12y26
x12y18
AR(-1:11):= 0;
AR(12:23):= -2048;
RS
Accept
1
0
x4y27
W(pre):=
AR(0:23)
x16y21
p. 7
HS
x16y20
p. 7
HS
HA(23)
1
0
x0y21
if SBenable then
SB(12:23):= SC(12:23)
x0y5
if SBenable then
SB(0:11):= SC(12:23)
x16y8
p. 7
RS

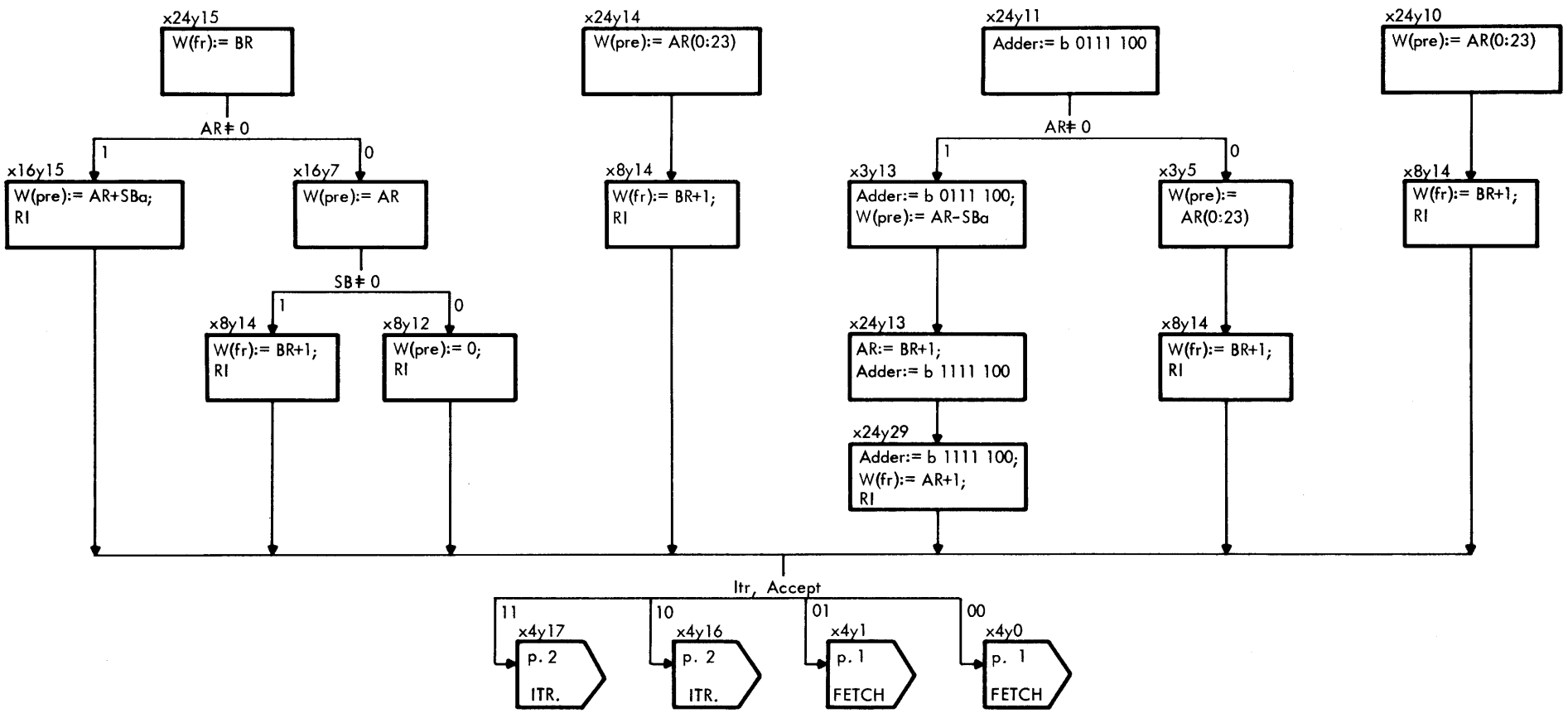

x2y22 SN
AR:= Wa(fr);
Adder:= b 0111 100
x2y20 SE
AR:= Wa(fr);
Adder:= b 0111 100
x2y18 SL
AR:= Wa(fr);
Adder:= b 0111 100
x2y16 SH
AR:= Wa(fr);
Adder:= b 0111 100
x3y16 SO
AR:= SBa
x3y18 SZ
AR:= Wa(fr)
x3y20 SX
AR:= 22ext0conEX
x17y6
Adder:= b 0111 100;
AR:= AR-SBa
x3y15
SB:= W(fr);
Adder:= b 0111 000
x17y15
AR:= AR & SBa
FR(4), FR(5)
11
10
01
00
x28y31 SN
if AR≠0 then IC:= IC+1
x28y29 SE:SO:SZ:SX
if AR= 0 then IC:= IC+1
x28y27 SL
if AR(-1) = 1 then IC:= IC+1
x28y25 SH
if AR> 0 then IC:= IC+1
x9y4
Adder:= b 0111 000;
AR:= AR &-, SBa
x4y0 p. 1 FETCH
x4y0 p. 1 FETCH
x4y0 p. 1 FETCH
x4y0 p. 1 FETCH

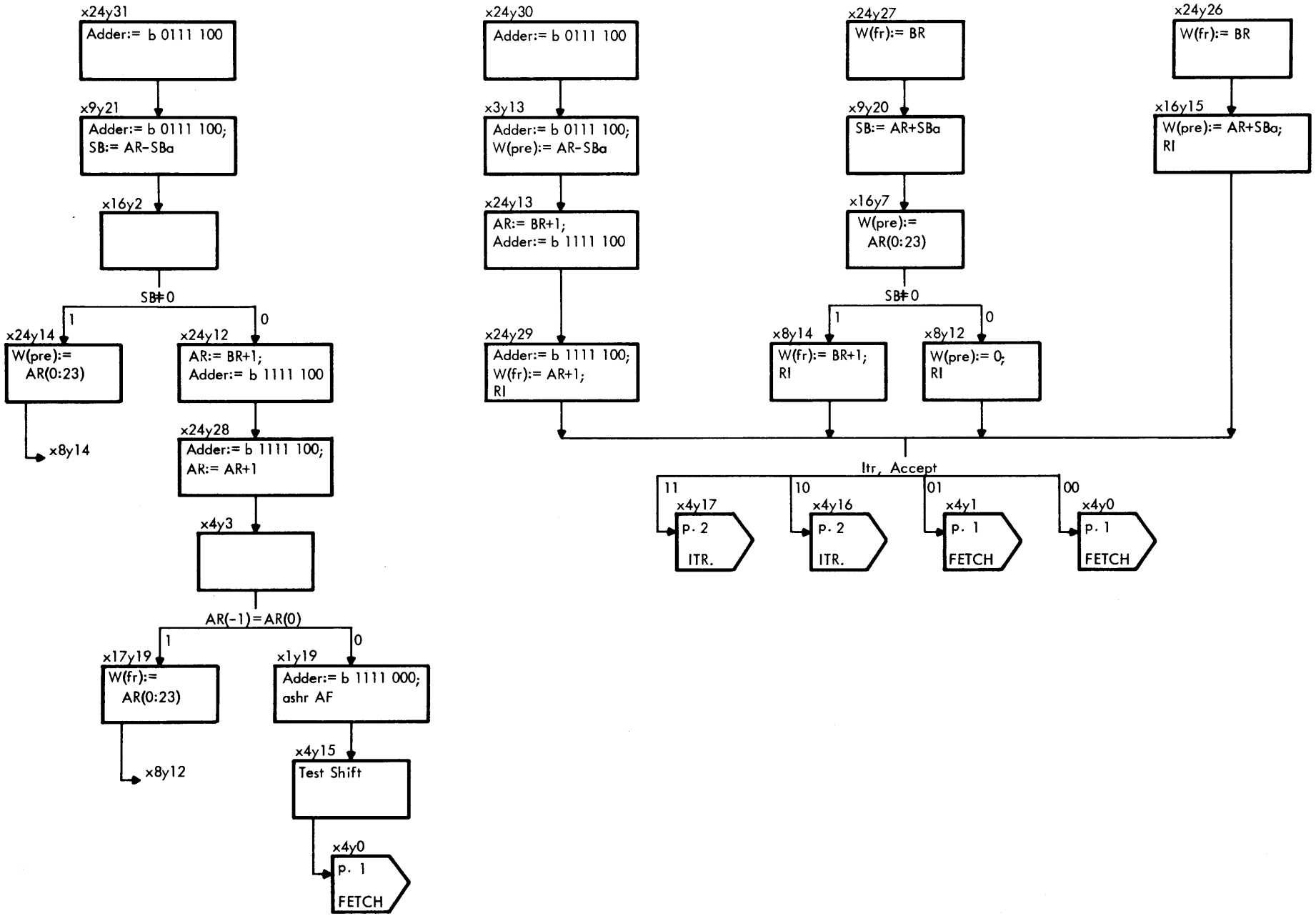

x3y2 JL
AR:= SBa;
RD
Accept
1
0
x3y3
BR:=
5ext0conICcon0
x3y4 JD
AR:= SBa;
ITRenable:= FR(5);
RD
x3y6 JE
AR:= SBa;
ITRenable:= FR(5);
RD
MMode, Accept
11
10
01
00
x20y7
BR:=
5ext0conICcon0
x20y6
RD
Accept
1
0
x20y3
x20y2
IR(0):= 1;
SB:= 12
x4y6
p. 2
ITR.
Main Power Key ON & -, Reset
1
0
x20y17
IC:= AR(5:22)
x20y1
AR:=
6ext0conICcon0
FR(6)!FR(7), MMode!-, PROTECT
11
10
01
00
x28y7
W(fr):= BR
x28y5
IC:= BR(5:22)
x24y7
FR:= SB(0:11)
x24y5
IC:= BR(5:22)
x4y30
SB:= -2
x20y2
x20y2
Itr
1
0
x4y24
SB:= 12
x4y8
SB(0:11):= 12extSB(12); IC:= IC+1; MMode:= PROTECT;
AR:= if -, FR(8) then
begin if index ≠ 0 then Wa(index) else 0 end
x31y2
IC:= AR+SBa
FR(8), FR(9), FR(10)!FR(11)
111
110
101
100
011
010
001
000
FR(0:5)
x4y6
p. 2
ITR.
x24y1
x24y0
x16y1
x16y0
x8y1
x8y0
x0y1
p. 1
FETCH
64 instructions
x4y2
p. 3
RESET

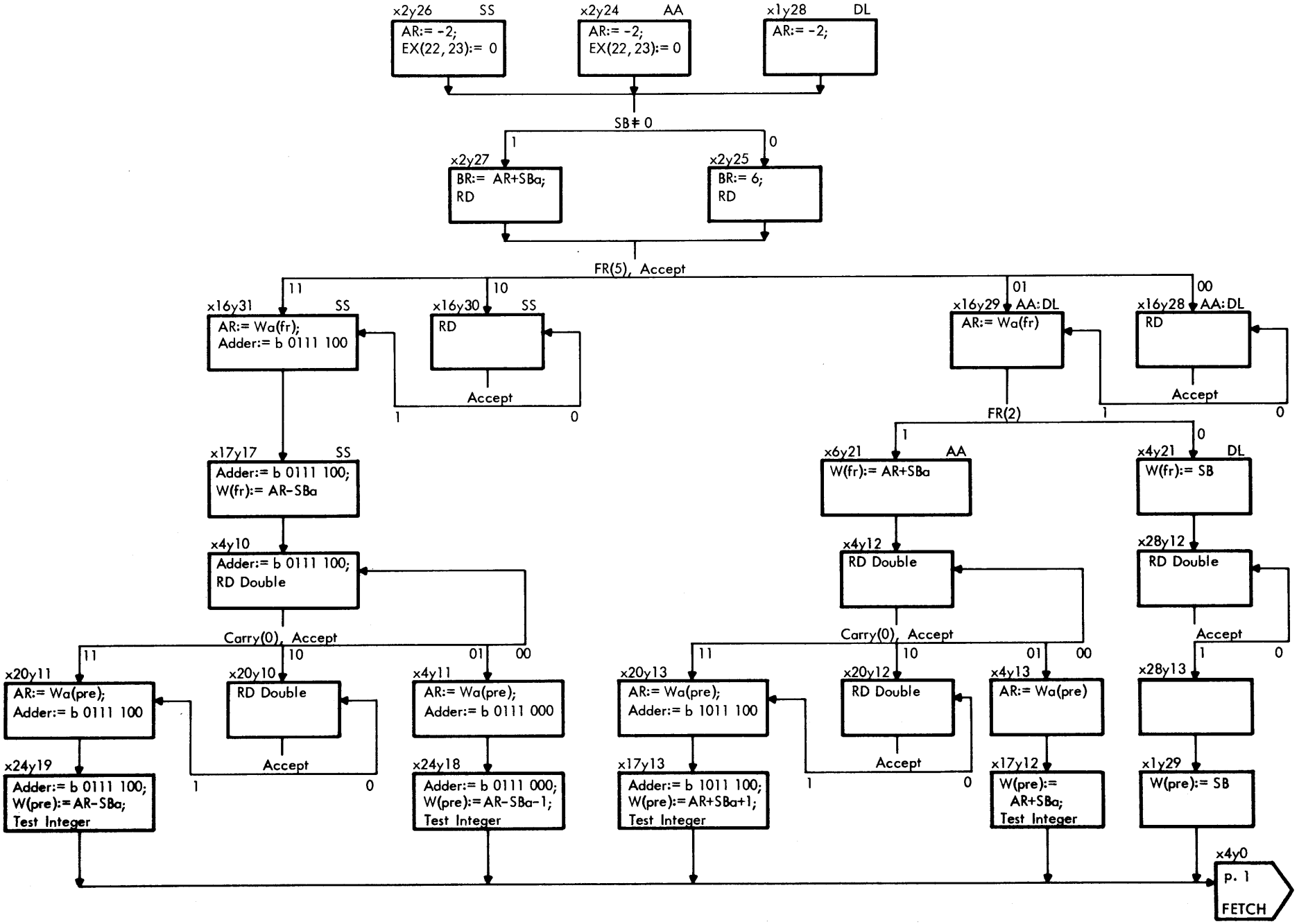

x2y4
WM
SC:= 23;
RD
Accept
1
0
x2y5
BR:= W(fr);
SC:= SC-1
x2y13
AR:= 0
BR(23)
1
0
x16y26
sc:= SC; SC:= SC-1;
AR:= AR+SBa;
ashr ARconBR
x16y18
Adder:= b 1111 000;
sc:= SC; SC:= SC-1;
ashr ARconBR
11
BR(22), sc ≠ 0
01
10
00
x16y24
Adder:= b 0111 100;
AR:= AR-SBa;
ashr ARconBR
x16y16
Adder:= b 1111 000;
ashr ARconBR
x12y1
W(fr):= BR
x1y17
W(pre):= AR(0:23);
RI
Itr, Accept
11
10
01
00
x4y17
p. 2
ITR.
x4y16
p. 2
ITR.
x4y1
p. 1
FETCH
x4y0
p. 1
FETCH

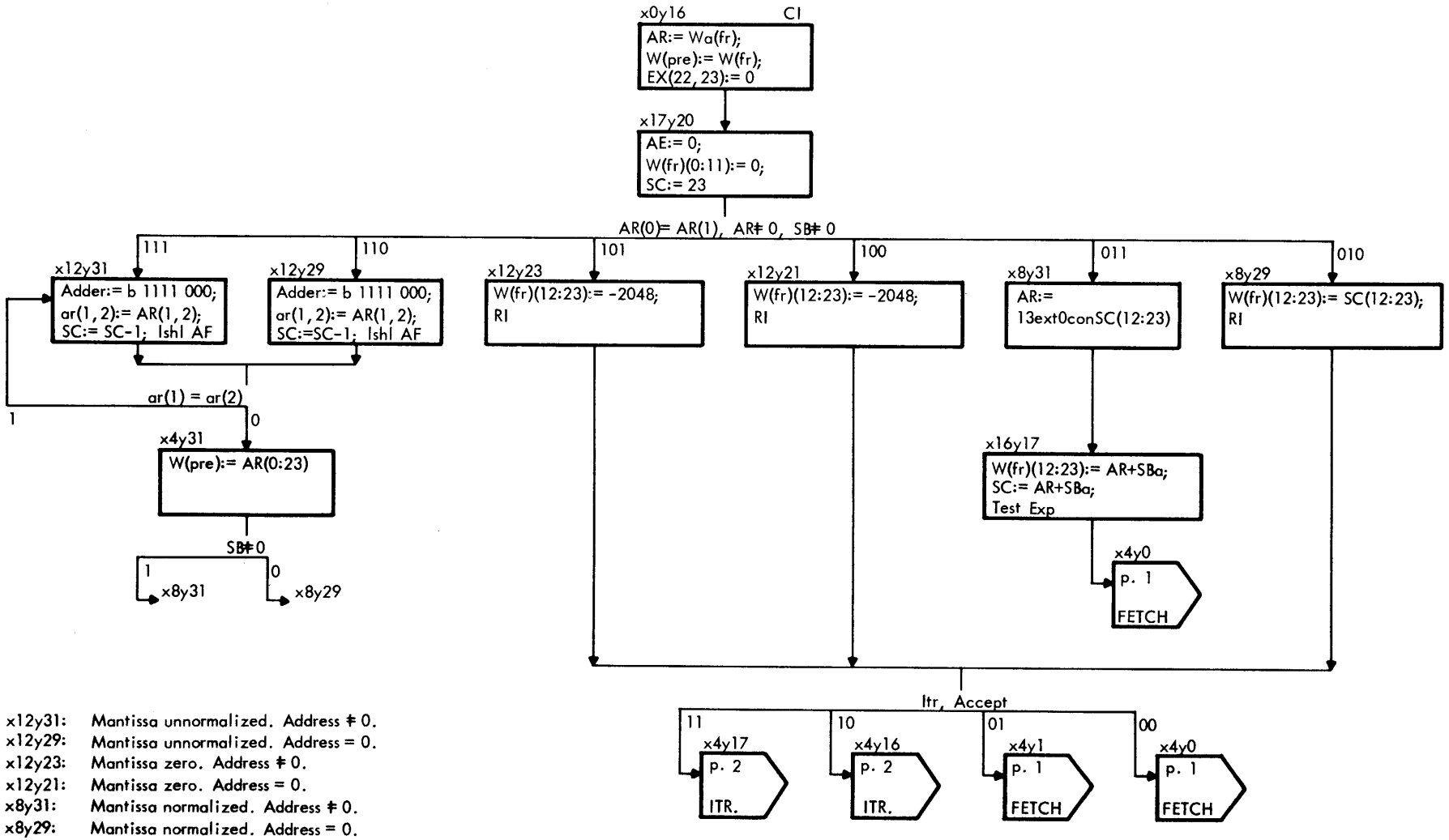


220768AG 221069HA 261169BRJ 261169AG

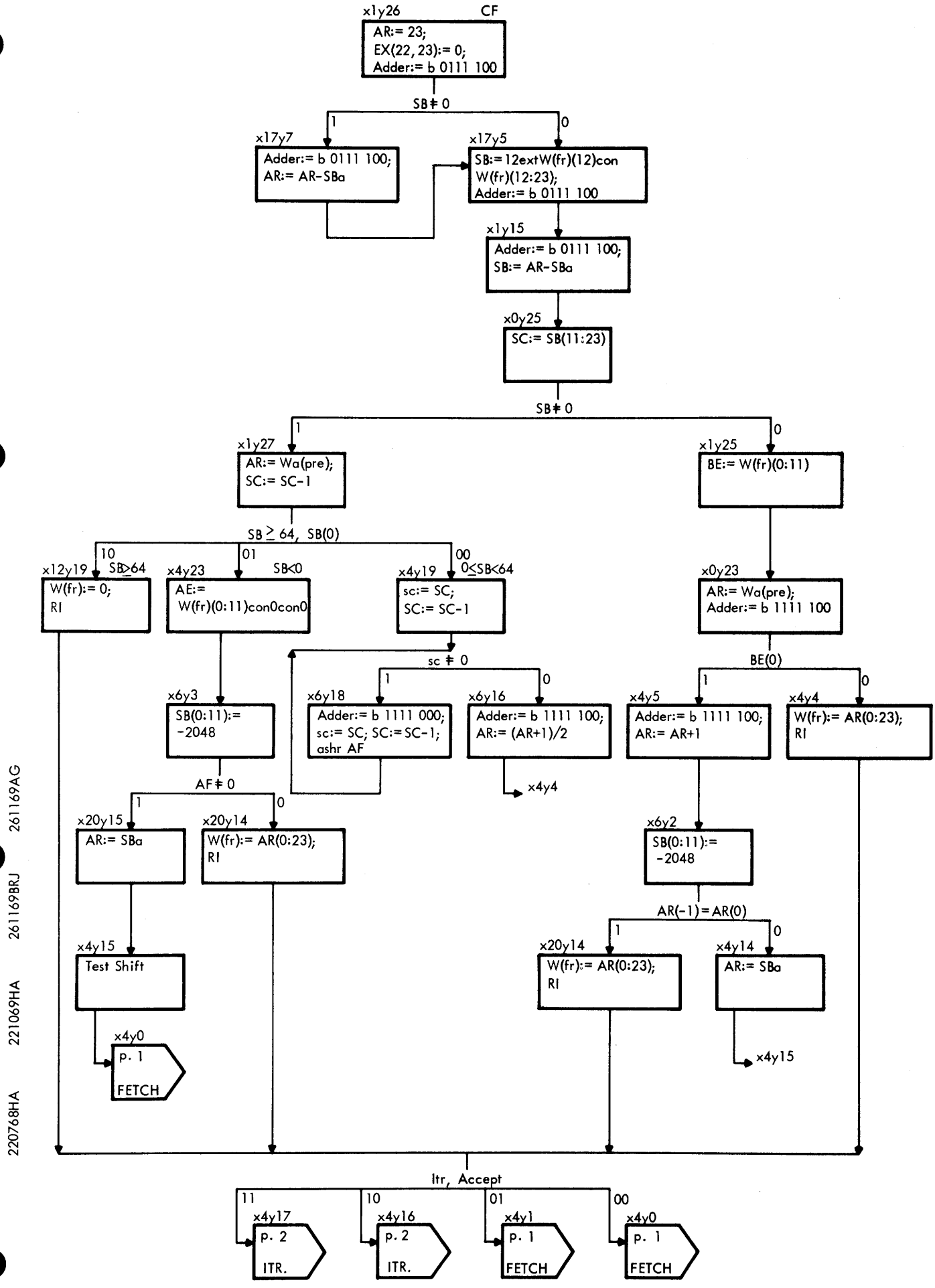

x24y15
W(fr):= BR
AR≠ 0
1
0
x16y15
W(pre):= AR+SBa; RI
x16y7
W(pre):= AR
SB≠ 0
1
0
x8y14
W(fr):= BR+1; RI
x8y12
W(pre):= 0; RI
x24y14
W(pre):= AR(0:23)
x8y14
W(fr):= BR+1; RI
x24y11
Adder:= b 0111 100
AR≠ 0
1
0
x3y13
Adder:= b 0111 100; W(pre):= AR-SBa
x3y5
W(pre):= AR(0:23)
x24y13
AR:= BR+1; Adder:= b 1111 100
x8y14
W(fr):= BR+1; RI
x24y29
Adder:= b 1111 100; W(fr):= AR+1; RI
x24y10
W(pre):= AR(0:23)
x8y14
W(fr):= BR+1; RI
Itr, Accept
11
x4y17
p. 2
ITR.
10
x4y16
p. 2
ITR.
01
x4y1
p. 1
FETCH
00
x4y0
p. 1
FETCH

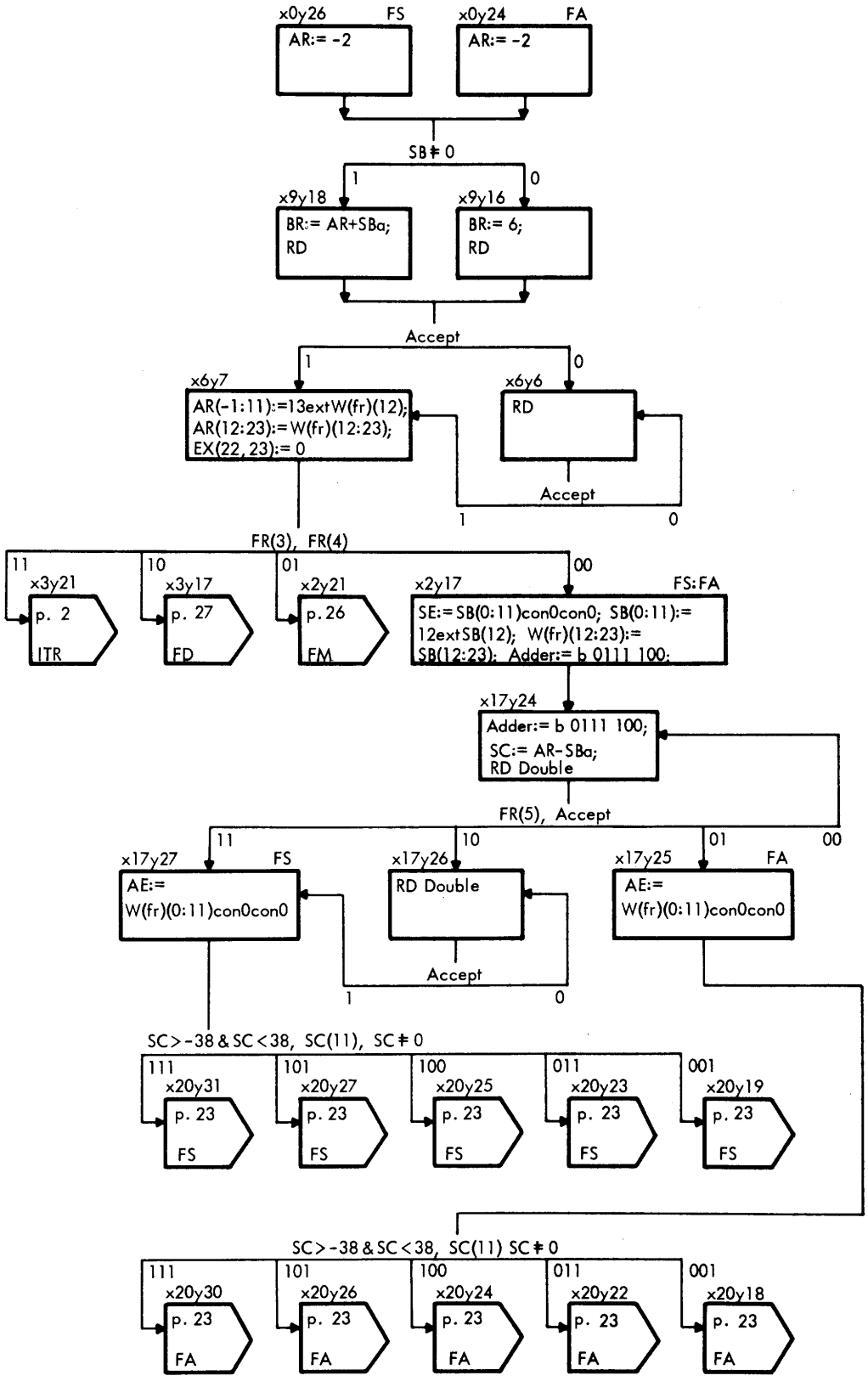

x24y31
Adder:= b 0111 100
x9y21
Adder:= b 0111 100;
SB:= AR-SBa
x16y2
SB≠0
1
0
x24y14
W(pre):=
AR(0:23)
x8y14
x24y12
AR:= BR+1;
Adder:= b 1111 100
x24y28
Adder:= b 1111 100;
AR:= AR+1
x4y3
AR(-1)=AR(0)
1
0
x17y19
W(fr):=
AR(0:23)
x8y12
x1y19
Adder:= b 1111 000;
ashr AF
x4y15
Test Shift
x4y0
p. 1
FETCH
x24y30
Adder:= b 0111 100
x3y13
Adder:= b 0111 100;
W(pre):= AR-SBa
x24y13
AR:= BR+1;
Adder:= b 1111 100
x24y29
Adder:= b 1111 100;
W(fr):= AR+1;
RI
x24y27
W(fr):= BR
x9y20
SB:= AR+SBa
x16y7
W(pre):=
AR(0:23)
SB≠0
1
0
x8y14
W(fr):= BR+1;
RI
x8y12
W(pre):= 0;
RI
x24y26
W(fr):= BR
x16y15
W(pre):= AR+SBa;
RI
Itr, Accept
11
10
01
00
x4y17
p. 2
ITR.
x4y16
p. 2
ITR.
x4y1
p. 1
FETCH
x4y0
p. 1
FETCH


x2y26 SS
AR:= -2;
EX(22, 23):= 0
x2y24 AA
AR:= -2;
EX(22, 23):= 0
x1y28 DL
AR:= -2;
SB≠ 0
1
0
x2y27
BR:= AR+SBa;
RD
x2y25
BR:= 6;
RD
FR(5), Accept
11
10
01
00
x16y31 SS
AR:= Wa(fr);
Adder:= b 0111 100
x16y30 SS
RD
Accept
1
0
x16y29 AA:DL
AR:= Wa(fr)
x16y28 AA:DL
RD
Accept
1
0
FR(2)
1
0
x17y17 SS
Adder:= b 0111 100;
W(fr):= AR-SBa
x6y21 AA
W(fr):= AR+SBa
x4y21 DL
W(fr):= SB
x4y10
Adder:= b 0111 100;
RD Double
x4y12
RD Double
x28y12
RD Double
Carry(0), Accept
11
10
01
00
Accept
1
0
x20y11
AR:= Wa(pre);
Adder:= b 0111 100
x20y10
RD Double
Accept
1
0
x4y11
AR:= Wa(pre);
Adder:= b 0111 000
x20y13
AR:= Wa(pre);
Adder:= b 1011 100
x20y12
RD Double
Accept
1
0
x4y13
AR:= Wa(pre)
x28y13
x24y19
Adder:= b 0111 100;
W(pre):= AR-SBa;
Test Integer
x24y18
Adder:= b 0111 000;
W(pre):= AR-SBa-1;
Test Integer
x17y13
Adder:= b 1011 100;
W(pre):= AR+SBa+1;
Test Integer
x17y12
W(pre):=
AR+SBa;
Test Integer
x1y29
W(pre):= SB
x4y0
p. 1
FETCH



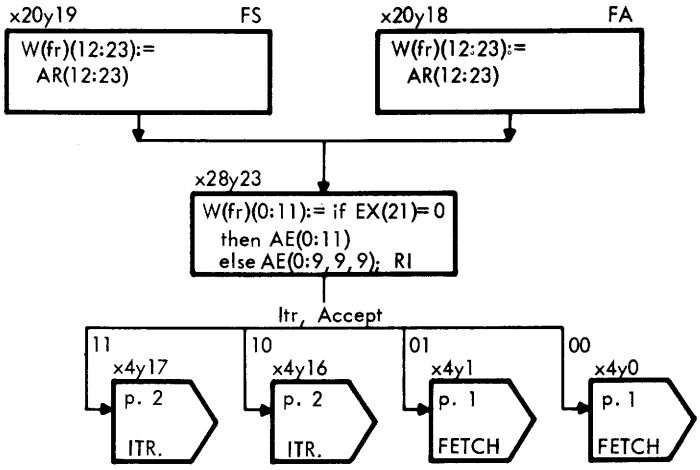
x12y31: Mantissa unnormalized. Address ≠ 0.
x12y29: Mantissa unnormalized. Address = 0.
x12y23: Mantissa zero. Address ≠ 0.
x12y21: Mantissa zero. Address = 0.
x8y31: Mantissa normalized. Address ≠ 0.
x8y29: Mantissa normalized. Address = 0.


x1y26 CF
AR:= 23;
EX(22,23):= 0;
Adder:= b 0111 100
SB≠ 0
1
0
x17y7
Adder:= b 0111 100;
AR:= AR-SBa
x17y5
SB:= 12extW(fr)(12)con
W(fr)(12:23);
Adder:= b 0111 100
x1y15
Adder:= b 0111 100;
SB:= AR-SBa
x0y25
SC:= SB(11:23)
SB≠ 0
1
0
x1y27
AR:= Wa(pre);
SC:= SC-1
x1y25
BE:= W(fr)(0:11)
SB ≥ 64, SB(0)
10
01
00
x12y19 SB≥64
W(fr):= 0;
RI
x4y23 SB<0
AE:=
W(fr)(0:11)con0con0
x4y19 0≤SB<64
sc:= SC;
SC:= SC-1
x0y23
AR:= Wa(pre);
Adder:= b 1111 100
sc ≠ 0
1
0
BE(0)
1
0
x6y3
SB(0:11):=
-2048
x6y18
Adder:= b 1111 000;
sc:= SC; SC:= SC-1;
ashr AF
x6y16
Adder:= b 1111 100;
AR:= (AR+1)/2
x4y5
Adder:= b 1111 100;
AR:= AR+1
x4y4
W(fr):= AR(0:23);
RI
x4y4
AF≠ 0
1
0
x20y15
AR:= SBa
x20y14
W(fr):= AR(0:23);
RI
x6y2
SB(0:11):=
-2048
AR(-1)=AR(0)
1
0
x4y15
Test Shift
x20y14
W(fr):= AR(0:23);
RI
x4y14
AR:= SBa
x4y15
x4y0
p. 1
FETCH
Itr, Accept
11
10
01
00
x4y17
p. 2
ITR.
x4y16
p. 2
ITR.
x4y1
p. 1
FETCH
x4y0
p. 1
FETCH
261169AG
261169BRJ
221069HA
220768HA

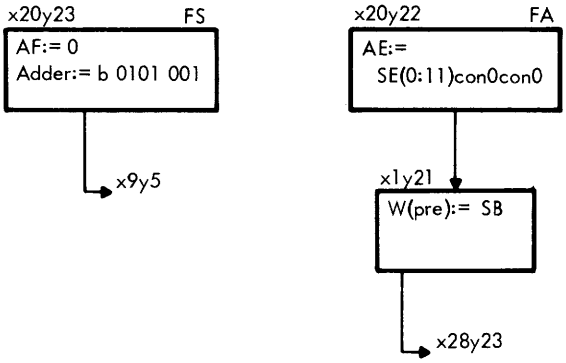


220768AG 221069HA 261169BRJ 261169AG

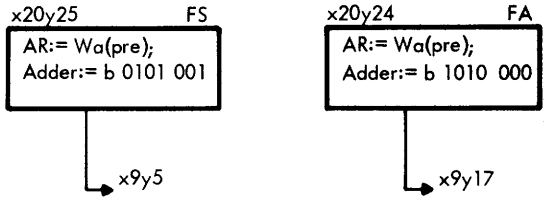
SC ≥ 38



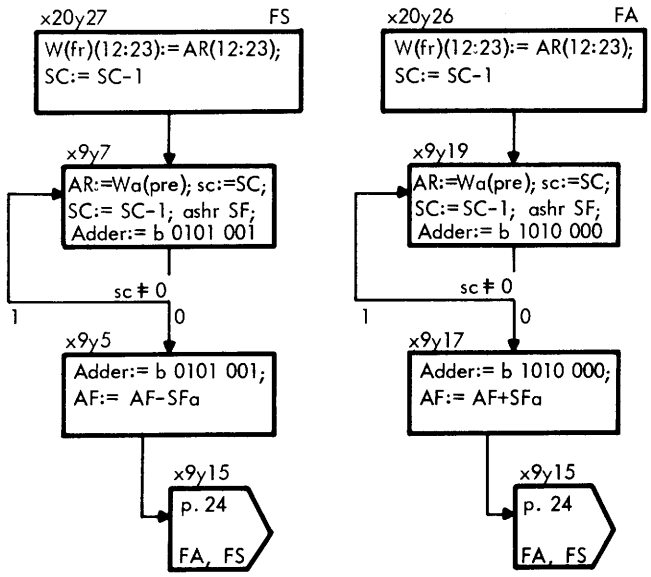
SC ≤ -38



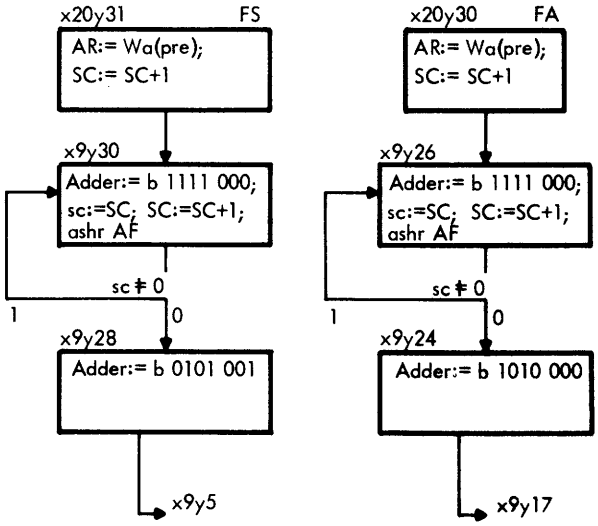
SC = 0



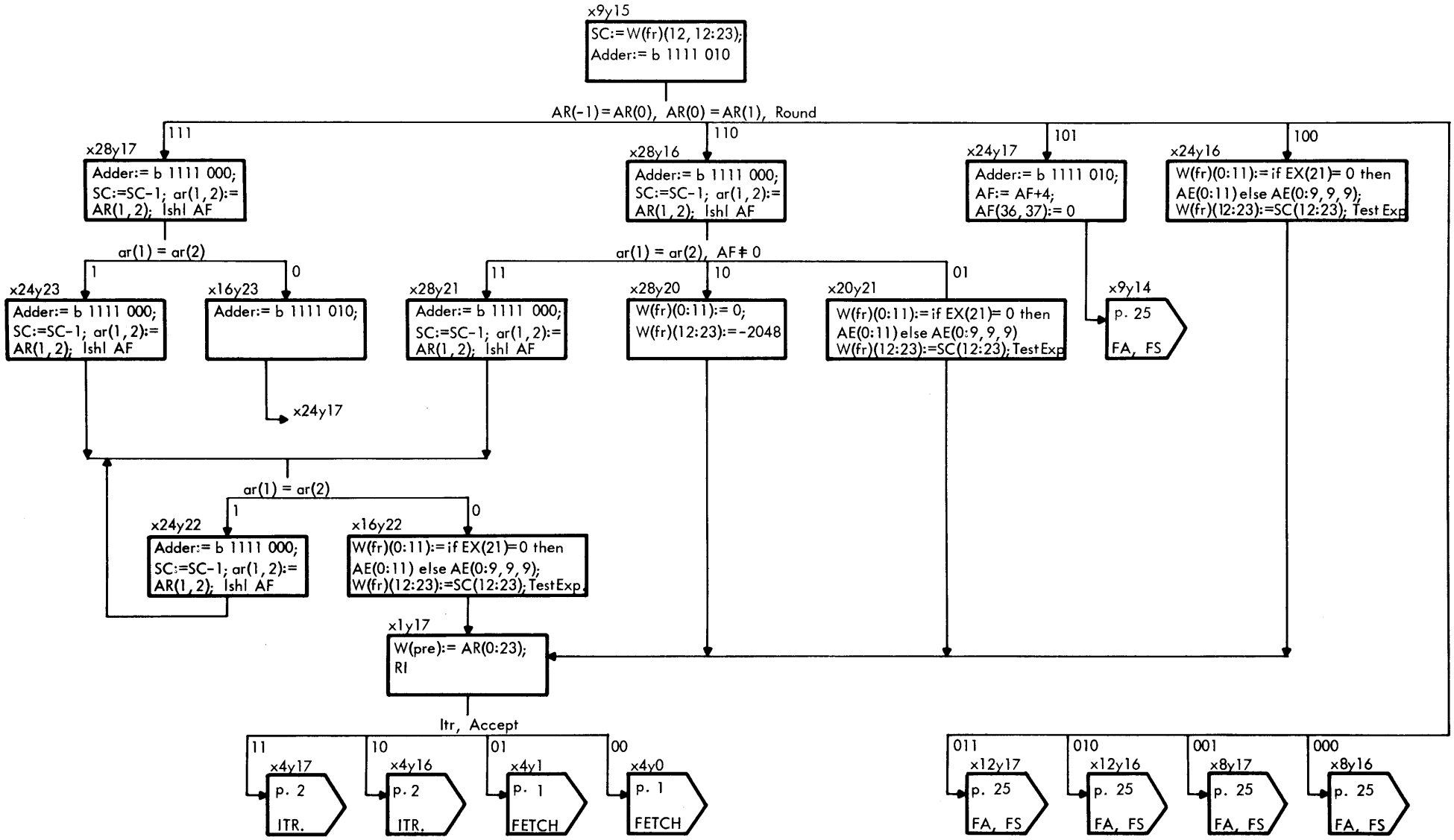
0 < SC < 38



-38 < SC < 0

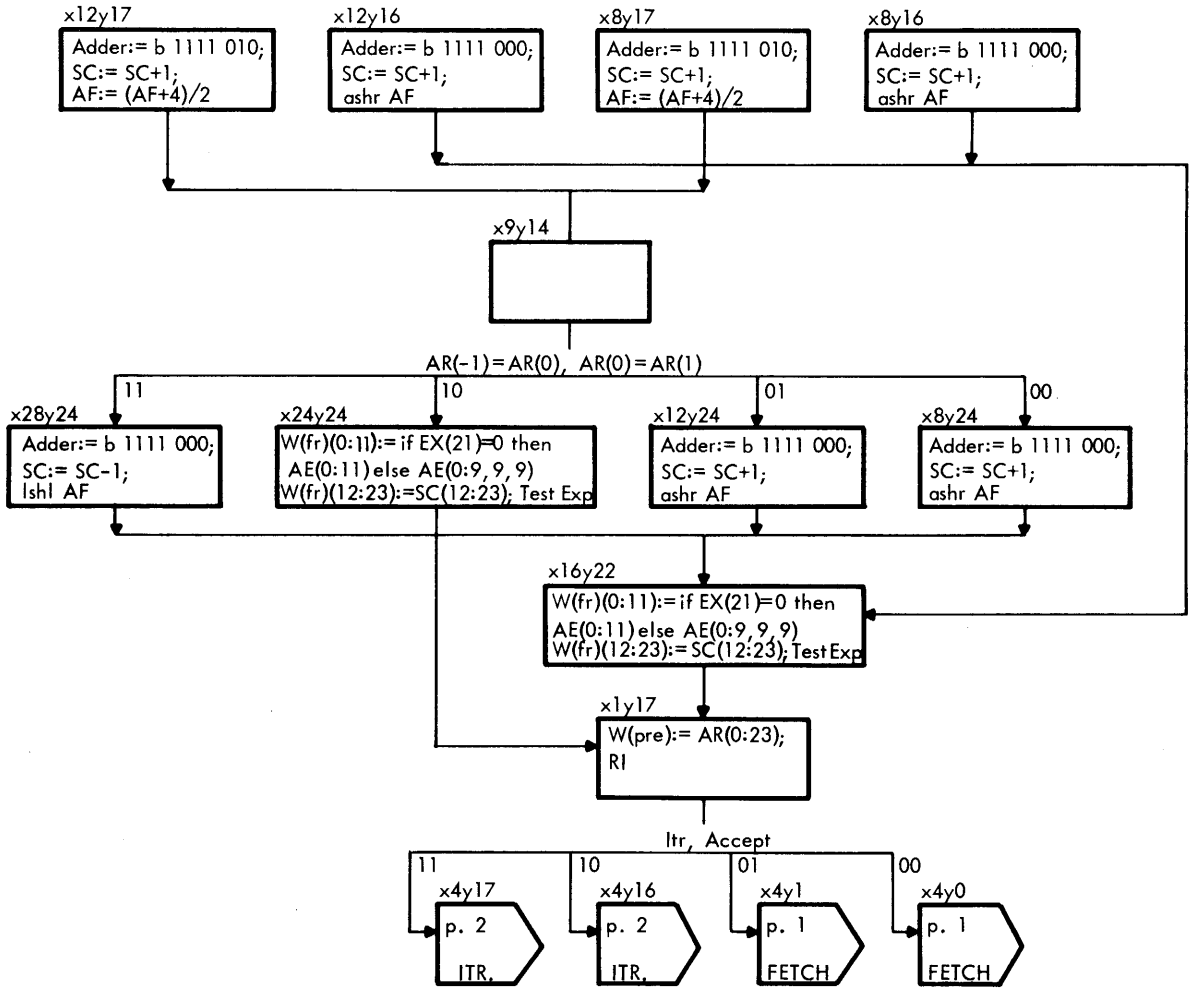


261169AG
261169BRJ
221069HA
220768AG



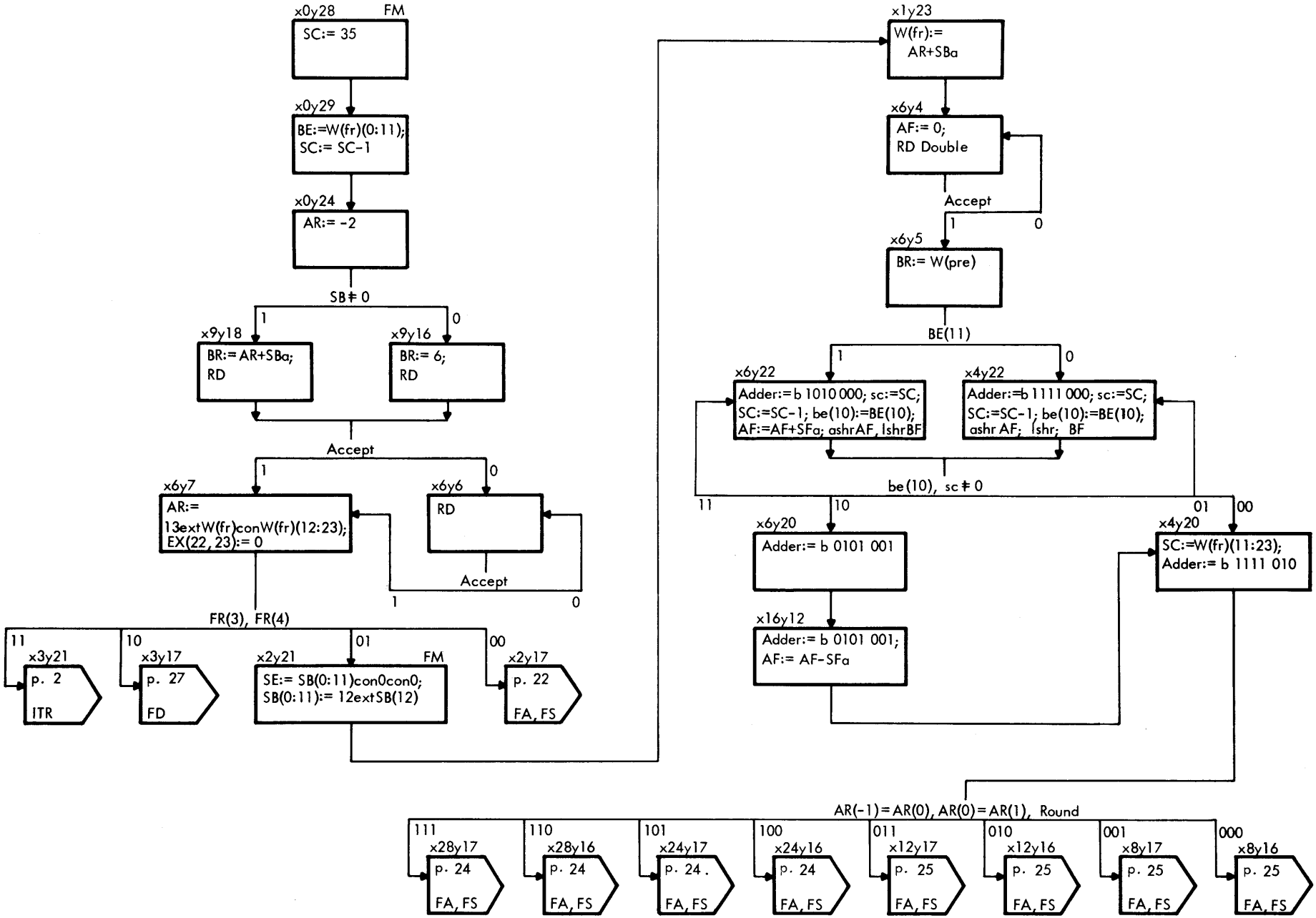
x28y17: Mantissa unnormalized. Perhaps rounding.
x28y16: Mantissa unnormalized or zero. No rounding.
x24y17: Mantissa normalized. Rounding.
x24y16: Mantissa normalized. No rounding.

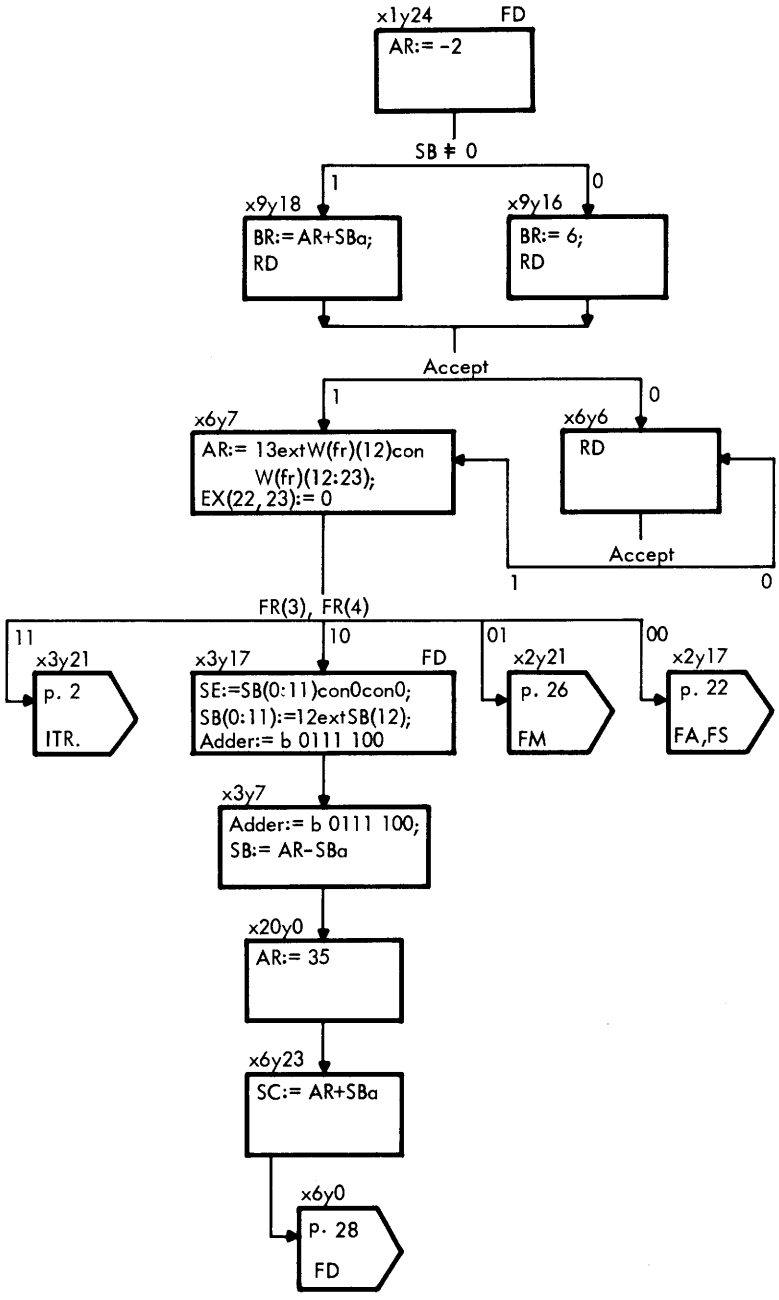
x12y17: Mantissa overflow. Rounding.
x12y16: Mantissa overflow. No rounding.
x8y17: Mantissa overflow. Rounding.
x8y16: Mantissa overflow. No rounding.



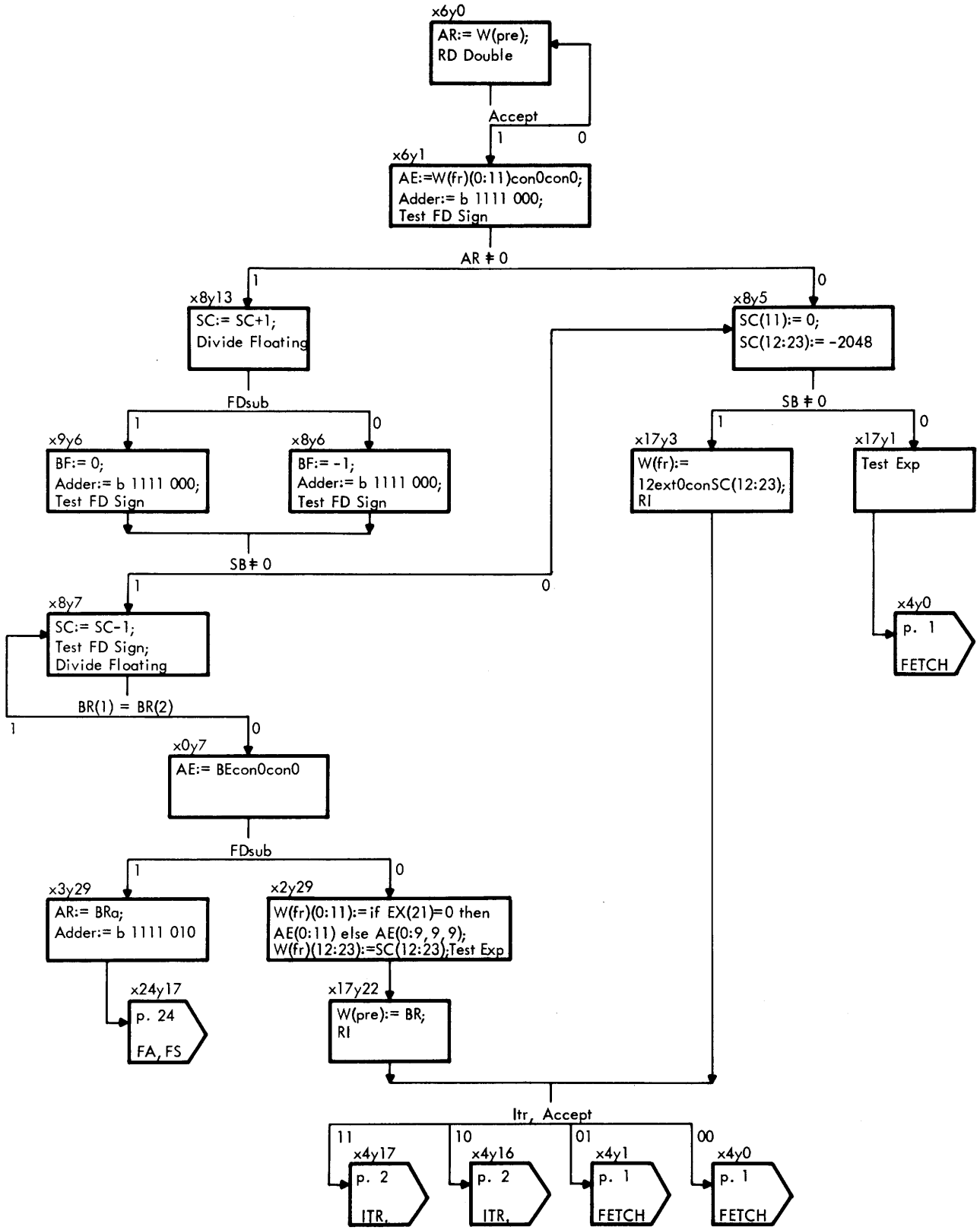
x12y17: Mantissa overflow. Rounding.
x12y16: Mantissa overflow. No rounding.
x8y17: Mantissa overflow. Rounding.
x8y16: Mantissa overflow. No rounding.

x28y24: Mantissa unnormalized.
x24y24: Mantissa normalized.
x12y24: Mantissa overflow.
x8y24: Mantissa overflow.

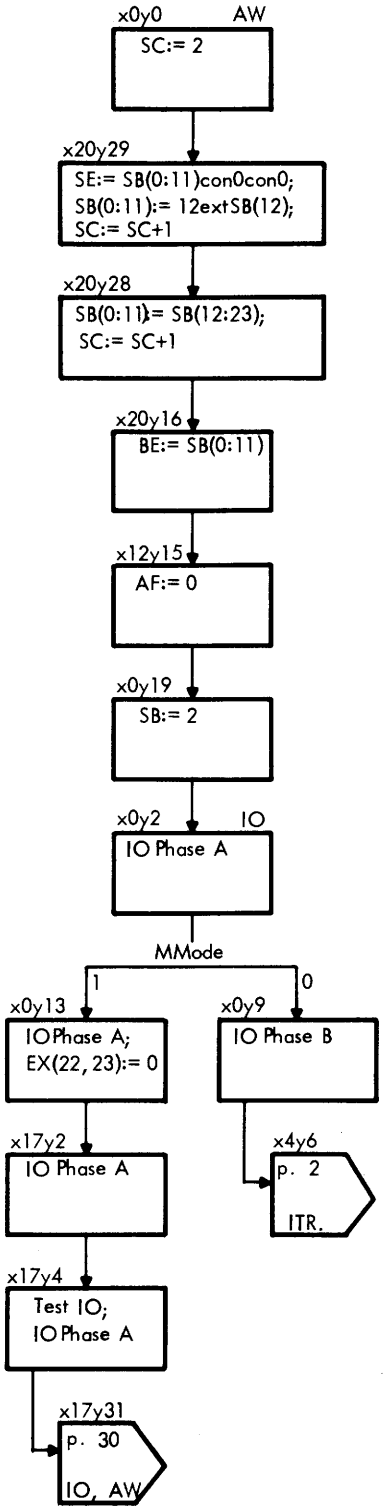

x0y28 FM
SC:= 35
x0y29
BE:=W(fr)(0:11);
SC:= SC-1
x0y24
AR:= -2
SB≠ 0
1
0
x9y18
BR:= AR+SBa;
RD
x9y16
BR:= 6;
RD
Accept
1
0
x6y7
AR:=
13extW(fr)conW(fr)(12:23);
EX(22, 23):= 0
x6y6
RD
Accept
1
0
FR(3), FR(4)
11
10
01
00
x3y21
p. 2
ITR
x3y17
p. 27
FD
x2y21
FM
SE:= SB(0:11)con0con0;
SB(0:11):= 12extSB(12)
x2y17
p. 22
FA, FS
x1y23
W(fr):=
AR+SBa
x6y4
AF:= 0;
RD Double
Accept
1
0
x6y5
BR:= W(pre)
BE(11)
1
0
x6y22
Adder:= b 1010 000; sc:=SC;
SC:=SC-1; be(10):=BE(10);
AF:=AF+SFa; ashrAF, lshrBF
x4y22
Adder:=b 1111 000; sc:=SC;
SC:=SC-1; be(10):=BE(10);
ashr AF; lshr; BF
be(10), sc≠ 0
11
10
01
00
x6y20
Adder:= b 0101 001
x16y12
Adder:= b 0101 001;
AF:= AF-SFa
x4y20
SC:=W(fr)(11:23);
Adder:= b 1111 010
AR(-1)=AR(0), AR(0)=AR(1), Round
111
110
101
100
011
010
001
000
x28y17
p. 24
FA, FS
x28y16
p. 24
FA, FS
x24y17
p. 24
FA, FS
x24y16
p. 24
FA, FS
x12y17
p. 25
FA, FS
x12y16
p. 25
FA, FS
x8y17
p. 25
FA, FS
x8y16
p. 25
FA, FS



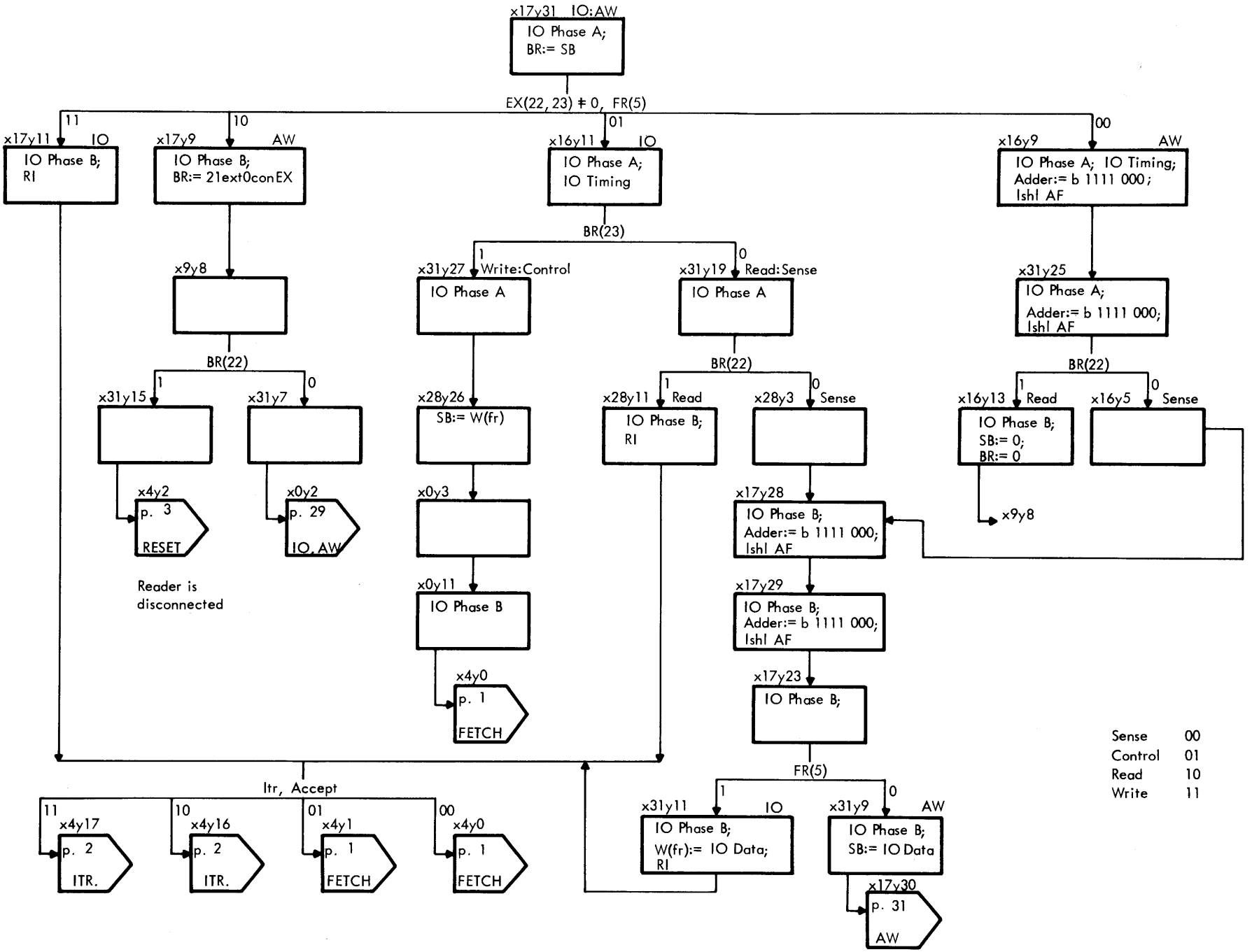
220768AG 221069BRJ 261169BRJ 261169AG

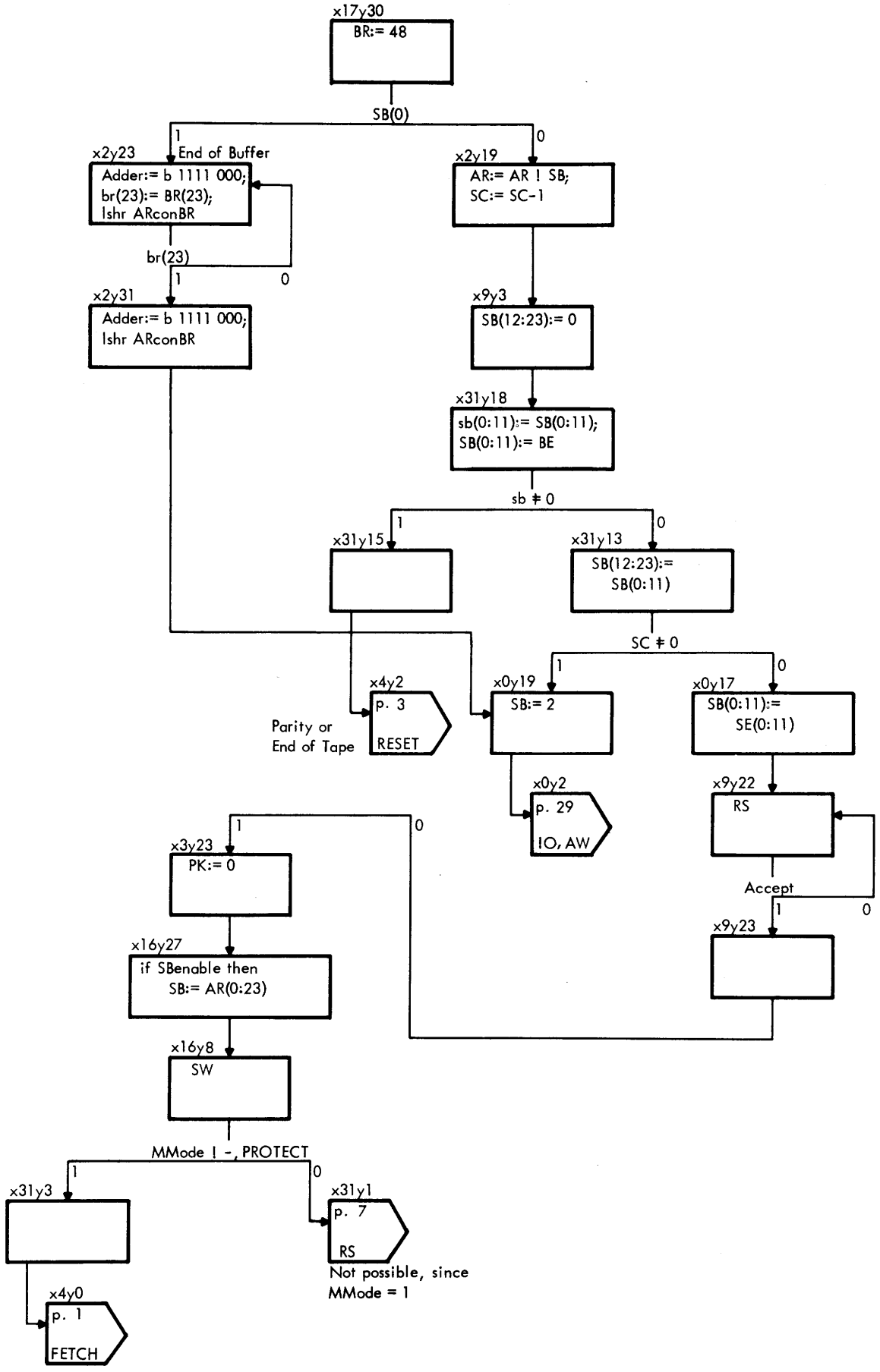


220768AG 221069BRJ 261169BRJ 261169AG



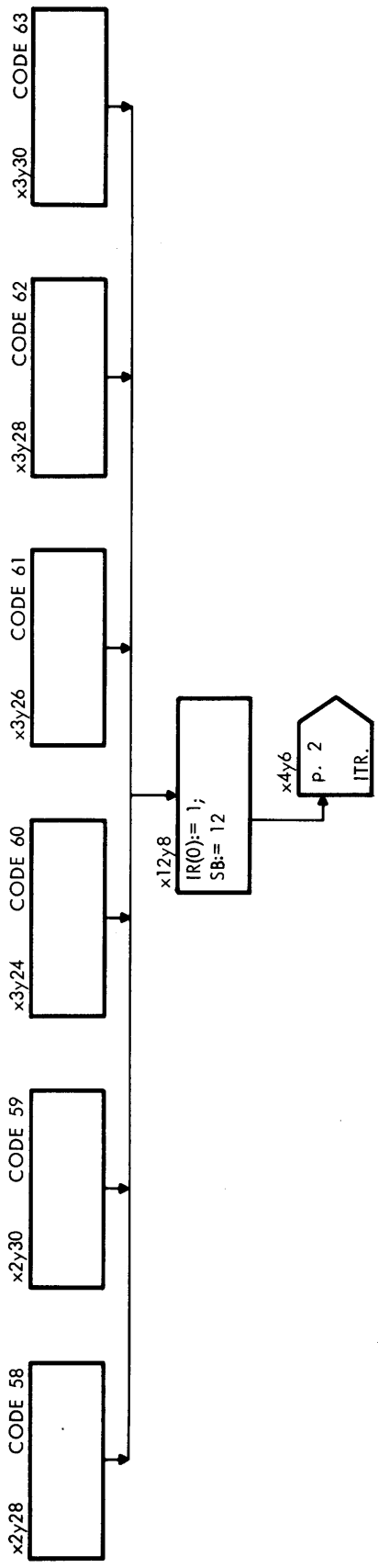
220768AG 101069HA 261169BRJ 261169AG


x17y31 IO:AW
IO Phase A;
BR:= SB
EX(22,23) ≠ 0, FR(5)
11
10
01
00
x17y11 IO
IO Phase B;
RI
x17y9 AW
IO Phase B;
BR:= 21ext0conEX
x16y11 IO
IO Phase A;
IO Timing
x16y9 AW
IO Phase A; IO Timing;
Adder:= b 1111 000;
lshl AF
BR(23)
1 Write:Control
0 Read:Sense
x9y8
x31y27
IO Phase A
x31y19
IO Phase A
x31y25
IO Phase A;
Adder:= b 1111 000;
lshl AF
BR(22)
BR(22)
BR(22)
x31y15
x31y7
x28y26
SB:= W(fr)
x28y11 Read
IO Phase B;
RI
x28y3 Sense
x16y13 Read
IO Phase B;
SB:= 0;
BR:= 0
x16y5 Sense
x4y2
p. 3
RESET
Reader is
disconnected
x0y2
p. 29
IO, AW
x0y3
x0y11
IO Phase B
x4y0
p. 1
FETCH
x17y28
IO Phase B;
Adder:= b 1111 000;
lshl AF
x9y8
x17y29
IO Phase B;
Adder:= b 1111 000;
lshl AF
x17y23
IO Phase B;
Itr, Accept
FR(5)
11
10
01
00
x4y17
p. 2
ITR.
x4y16
p. 2
ITR.
x4y1
p. 1
FETCH
x4y0
p. 1
FETCH
x31y11 IO
IO Phase B;
W(fr):= IO Data;
RI
x31y9 AW
IO Phase B;
SB:= IO Data
x17y30
p. 31
AW
Sense 00
Control 01
Read 10
Write 11



261169AG 261169BRJ 101069HA 220768AG

220768AG 101069HA 261169BRJ 261169AG



RCSL-51: VB306
Author: Allan Giese
Edited: January 1969

BIT PATTERN IN THE MICROPROGRAM STORE FOR

THE RC 4000 COMPUTER

A/S REGNECENTRALEN
Falkoneralle 1
Copenhagen F.

The Microprogram Store (MPS) has a capacity of 1024 words each of 100 bits. For mnemonic reasons, MPS is logically divided into 32 sections, and the p'th word in the n'th section is identified as xnyp (xy-number). Each bit in a word is identified by a position number in the range from 1 to 100. The bits having the position numbers 2 to 70 constitute the micro command field, whereas the jump selector field occupies the bit numbers from 70 to 100. Bit 1 specifies the parity bit of the word.

Seventeen of the thirty-two sections are not used by the microprogram, and the value zero is stored in all these words, for which reason they are omitted from the table to follow. The contents of each word in the remaining sections are given in the table by means of the position numbers. For example,

x0y10: 1, 46,62, 73,76,97,98,

states that the bit positions 1,46,62,73,76,97, and 98 have the logical value 1 and that the unspecified 93 bits have the logical value 0. It appears from the table that the contents of 8 words (x3y25, x3y27, x3y31, x9y9, x9y25, x12y11, x31y23, and x31y29) are 0 and these words are not employed by the microprogram. Hence, the microprogram is implemented by $1024 - 17 \times 32 - 8 = 472$ words.

It follows that the parity for each word which is not used by the microprogram is even, since every bit has the logical value zero. The words belonging to the microprogram, on the other hand, are arranged to have odd parity, and this is achieved by means of the parity bit.

x0y0:	1,	25,42,43,44,	73,79,88,91,94,100,
x0y1:	1,	11,29,70,	80,83,87,88,89,92,95,
x0y2:		2,	91,92,93,100,
x0y3:			91,97,100,
x0y4:		62,	73,76,97,98,
x0y5:	1,	24,69,	73,91,
x0y6:		62,	73,76,97,98,
x0y7:	1,	37,47,50,	82,84,85,88,91,94,100,
x0y8:		62,	73,76,97,98,
x0y9:	1,	3,29,40,44,70,	79,94,97,
x0y10:	1,	46,62,	73,76,97,98,
x0y11:	1,	3,	79,
x0y12:	1,	46,62,	73,76,97,98,
x0y13:		2,46,	73,85,97,
x0y14:		62,	73,76,97,98,
x0y15:		47,49,50,	73,76,94,97,
x0y16:	1,	15,20,34,46,	73,85,88,94,
x0y17:		31,70,	76,85,88,94,97,
x0y18:		6,9,34,46,	73,85,88,
x0y19:	1,	29,42,43,44,70,	97,
x0y20:		15,34,	73,85,88,94,100,
x0y21:	1,	24,29,	73,91,
x0y22:		19,34,	73,85,88,94,100,
x0y23:	1,	5,9,19,34,	79,94,98,100,
x0y24:		34,45,	76,85,88,95,97,
x0y25:	1,	25,28,	85,88,91,95,97,100,
x0y26:		34,45,	76,85,88,95,97,
x0y27:		64,	76,85,88,91,94,95,96,100,
x0y28:		25,42,43,	88,91,94,100,
x0y29:	1,	15,26,47,49,	88,91,
x0y30:	1,	63,	88,91,94,97,98,
x0y31:			73,91,92,93,97,

x1y0:	1,	62,	76,85,98,
x1y1:	1,	13,17,18,	79,
x1y2:	1,	62,	76,85,98,
x1y3:		17,18,28,33,	79,
x1y4:	1,	62,	76,85,98,
x1y5:	1,	15,28,39,	73,85,91,94,97,
x1y6:		46,62,	76,85,98,
x1y7:		11,17,18,50,	79,
x1y8:	1	62,	76,85,98,
x1y9:	1,	17,18,28,	79,
x1y10:	1,	62,	76,85,98,
x1y11:	1,	51,52,53,	79,
x1y12:	1,	62,	76,85,98,
x1y13:	1,	17,18,68,69,70,	79,
x1y14:	1,	15,34,63,	76,85,91,94,98,
x1y15:	1,	6,9,11,29,70,	88,91,100,
x1y16:		25,28,46,	79,82,88,91,92,94,95,97,
x1y17:	1,	20,33,61,	79,86,98,
x1y18:	1,	25,28,46,	79,82,88,91,92,94,95,97,100,
x1y19:		5,54,	79,91,94,97,100,
x1y20:	1,	25,28,	79,82,88,91,92,94,95,97,
x1y21:		20,28,	73,76,79,88,94,97,100,
x1y22:		25,28	79,82,88,91,92,94,95,97,100,
x1y23:	1,	11,17,18,	79,82,94,
x1y24:		34,45,	76,85,88,95,97,
x1y25:		15,47,49,	88,94,97,100,
x1y26:	1,	6,9,34,42,44,46,	73,85,94,95,97,100,
x1y27:	1,	19,26,34,	75,79,88,92,94,97,100,
x1y28:	1,	34,45,	82,88,91,95,97,100,
x1y29:		20,28,	79,
x1y30:		34,45,	73,85,91,95,97,
x1y31:	1,	19,34,63,	76,85,91,94,98,

```
x2y0:    1,    46,62,                  82,98,
x2y1:          6,9,15,34,              73,85,88,
x2y2:    1,    28,34,61,               82,97,98,
x2y3:          22,39,                  79,91,100,
x2y4:          25,42,44,62,            82,94,98,
x2y5:          15,26,39,               82,91,94,100,
x2y6:          17,18,28,61,            79,86,98,
x2y7:          17,18,28,               73,88,91,95,96,100,
x2y8:    1,    25,42,44,46,62,         82,91,98,
x2y9:    1,    15,26,39,               82,91,94,97,100,
x2y10:         15,34,63,               82,91,97,98,
x2y11:   1,                            82,94,97,100,
x2y12:         15,34,63,               73,88,94,98,
x2y13:   1,    34,                     73,88,89,91,97,
x2y14:   1,    34,48,49,63,            73,88,94,98,
x2y15:         19,26,34,47,49,         73,79,88,94,
x2y16:   1,    6,9,15,34,              73,85,94,97,
x2y17:         6,9,18,28,32,           73,85,88,91,
x2y18:   1,    6,9,15,34,              73,85,94,97,
x2y19:   1,    26,28,33,34,            76,85,97,100,
x2y20:   1,    6,9,15,34,              73,85,94,97,
x2y21:         28,32,                  85,88,94,97,100,
x2y22:   1,    6,9,15,34,              73,85,94,97,
x2y23:         5,54,55,56,             82,88,89,91,94,97,100,
x2y24:         34,45,46,               82,88,91,95,97,100,
x2y25:         39,42,62,               73,88,91,94,95,98,
x2y26:         34,45,46,               82,88,91,95,97,100,
x2y27:         11,39,62,               73,88,91,94,95,98,
x2y28:                                 76,79,91,
x2y29:   1,    17,18,24,36,49,         73,85,88,94,97,
x2y30:                                 76,79,91,
x2y31:         5,54,55,56,             88,97,100,
```

x3y0:	1,	62,	82,85,98,
x3y1:			76,79,91,92,93,
x3y2:		28,34,62,	82,85,97,98,
x3y3:	1,	22,39,	73,79,86,87,88,100,
x3y4:	1,	28,34,53,62,	73,79,92,93,97,98,
x3y5:	1,	20,33,	76,91,94,97,
x3y6:	1,	28,34,53,62,	73,79,92,93,97,98,
x3y7:		6,9,11,29,70,	73,79,
x3y8:		62,	82,85,91,98,
x3y9:	1,		73,76,79,82,85,86,88,92,93,100,
x3y10:		34,47,48,50,63,	73,88,94,98,
x3y11:	1,	41,47,48,68,69,	79,82,88,91,94,
x3y12:	1,	34,63,67,69,70,	76,85,91,94,98,
x3y13:		6,9,11,20,	73,76,91,94,100,
x3y14:	1,	34,63,67,68,70,	76,85,91,94,98,
x3y15:		6,15,29,70,	76,85,94,
x3y16:	1,	28,34,	82,85,91,94,97,100,
x3y17:		6,9,28,32,	82,85,94,97,100,
x3y18:	1,	15,34,	73,85,91,94,97,100,
x3y19:		27,64,	82,85,88,94,100,
x3y20:		34,48,49,	73,85,91,94,97,100,
x3y21:		24,29,70,	76,88,94,97,
x3y22:	1,		76,79,91,92,93,100,
x3y23:		68,69,	73,88,91,97,100,
x3y24:			76,79,91,
x3y25:			
x3y26:			76,79,91,
x3y27:			
x3y28:			76,79,91,
x3y29:	1,	5,10,34,38,	73,76,88,100,
x3y30:			76,79,91,
x3y31:			

```
x4y0:    1,   61,                    79,86,98,
x4y1:         22,34,                 79,86,87,88,97,
x4y2:    1,   68,69,                 73,85,88,97,
x4y3:                                71,85,88,97,100,
x4y4:         17,18,33,61,           79,86,98,
x4y5:         5,9,11,34,             79,82,97,
x4y6:    1,   39,42,44,62,           79,94,97,98,
x4y7:    1,   22,34,57,              82,85,91,97,100,
x4y8:         21,35,41,52,68,        72,73,75,76,81,84,87,90,93,96,99,100,
x4y9:    1,   41,52,68,              72,73,76,88,94,99,100,
x4y10:   1,   6,9,62,65,             71,73,79,91,97,98,
x4y11:        6,19,34,               73,76,88,97,
x4y12:   1,   62,65,                 71,73,79,91,94,98,
x4y13:   1,   19,34,                 73,85,91,94,
x4y14:        28,34,                 79,91,94,97,100,
x4y15:   1,   48,                    79,
x4y16:   1,   29,44,70,              79,94,97,
x4y17:   1,   29,44,70,              79,94,97,
x4y18:        21,35,41,52,68,        72,73,75,76,81,84,87,90,93,96,99,100,
x4y19:   1,   26,                    79,82,88,96,97,
x4y20:   1,   5,10,15,25,            71,76,77,88,98,99,
x4y21:   1,   17,18,28,              73,76,79,91,94,
x4y22:        5,26,54,57,            79,80,82,88,94,96,97,
x4y23:   1,   15,37,                 79,82,97,100,
x4y24:   1,   29,44,70,              79,94,97,
x4y25:   1,   6,13,17,18,61,         79,86,98,
x4y26:        17,18,38,63,           79,88,91,97,98,
x4y27:   1,   20,33,                 86,88,94,100,
x4y28:        17,18,33,63,           79,88,91,94,98,
x4y29:   1,                          86,88,94,100,
x4y30:        29,45,70,              73,76,79,82,85,97,
x4y31:        20,33,                 76,88,91,94,95,97,100,
```

```
x6y0:          19,34,62,65,          79,82,98,
x6y1:          5,15,37,67,           76,90,91,94,100,
x6y2:    1,    44,45,69,             71,79,91,94,97,
x6y3:          44,45,69,             73,79,91,94,97,99,
x6y4:    1,    34,37,62,65,          79,82,94,98,
x6y5:    1,    19,39,                79,81,82,88,94,97,
x6y6:    1,    62,                   79,82,94,97,98,
x6y7:          15,16,34,46,          82,83,88,92,100,
x6y8:          17,18,33,61,          79,86,98,
x6y9:    1,    17,18,38,             85,88,100,
x6y10:   1,    5,26,48,55,56,        79,82,91,96,97,
x6y11:         5,26,48,55,56,        79,82,91,96,97,100,
x6y12:         17,18,33,61,          79,86,98,
x6y13:   1,    17,18,38,             85,88,100,
x6y14:   1,    5,26,55,56,           79,82,91,94,96,97,
x6y15:         5,26,55,56,           79,82,91,94,96,97,100,
x6y16:         5,9,54,               79,94,
x6y17:         47,49,50,67,70,       74,76,91,94,97,100,
x6y18:   1,    5,26,54,              79,82,88,96,97,
x6y19:   1,    26,47,49,50,67,70,    79,82,88,96,97,100,
x6y20:         6,7,                  73,91,94,
x6y21:   1,    11,17,18,             79,91,94,
x6y22:         8,26,54,57,           79,80,82,88,94,96,97,
x6y23:   1,    11,25,                79,82,
x6y24:   1,    61,                   79,86,98,
x6y25:   1,    61,                   79,86,98,
x6y26:   1,    15,34,                73,75,79,91,
x6y27:         19,34,                73,75,79,91,100,
x6y28:         23,28,53,63,65,       79,82,88,91,94,98,
x6y29:         25,42,                76,79,88,91,100,
x6y30:   1,    15,27,34,             72,76,79,92,97,
x6y31:         19,27,34,             72,76,79,91,92,97,
```

```
x8y0:       62,                     76,87,88,89,97,98,
x8y1:   1,  11,29,70,               76,
x8y2:   1,  62,                     76,97,98,
x8y3:   1,                          76,88,97,100,
x8y4:       68,70,                  73,76,79,
x8y5:   1,  25,44,45,               73,85,95,97,100,
x8y6:   1,  5,39,43,45,47,49,67,    76,94,95,97,100,
x8y7:   1,  26,47,67,               74,76,94,97,100,
x8y8:   1,  63,                     76,91,98,
x8y9:   1,                          76,88,91,100,
x8y10:      62,                     76,91,97,98,
x8y11:  1,                          76,88,97,100,
x8y12:      20,61,                  79,86,98,
x8y13:      27,47,                  76,84,85,94,97,
x8y14:  1,  17,18,38,43,61,         79,86,98,
x8y15:      44,45,69,               79,91,94,97,
x8y16:      5,27,54,                73,88,94,97,
x8y17:      5,10,27,54,             76,85,91,94,97,
x8y18:      62,                     76,88,97,98,
x8y19:                              80,83,87,88,89,92,95,
x8y20:      11,29,70,               75,76,81,84,87,90,93,96,
x8y21:      11,29,70,               76,85,91,97,
x8y22:  1,  63,                     76,88,94,97,98,
x8y23:  1,                          76,85,88,91,97,100,
x8y24:      5,27,54,                73,88,94,97,
x8y25:      15,29,70,               88,91,97,100,
x8y26:  1,  62,                     76,88,91,97,98,
x8y27:  1,                          76,88,97,100,
x8y28:  1,  34,63,                  73,88,94,98,
x8y29:  1,  18,24,61,               79,86,98,
x8y30:  1,  34,63,                  73,88,94,98,
x8y31:      24,34,                  73,88,100,
```

```
x9y0:    1,   62,                    76,85,98,
x9y1:    1,   15,34,                 80,83,87,88,89,92,95,100,
x9y2:    1,   64,                    73,76,79,82,85,
x9y3:    1,   29,                    73,76,79,82,85,88,97,
x9y4:    1,   6,13,34,               73,76,79,88,91,94,100,
x9y5:         6,7,12,                76,85,91,94,97,100,
x9y6:    1,   5,39,47,49,67,         76,94,95,97,100,
x9y7:         6,7,19,26,34,68,70,    76,85,94,96,97,100,
x9y8:    1,                          73,76,79,82,85,89,90,94,97,100,
x9y9:
x9y10:        21,34,                 75,76,100,
x9y11:        34,38,                 73,75,76,99,100,
x9y12:   1,   63,                    76,85,91,94,98,
x9y13:                               73,88,91,97,100,
x9y14:                               71,76,77,88,91,
x9y15:        5,10,15,16,25,         71,76,77,88,98,99,
x9y16:   1,   39,42,62,              79,82,94,97,98,
x9y17:   1,   8,12,                  76,85,91,94,97,100,
x9y18:   1,   11,39,62,              79,82,94,97,98,
x9y19:   1,   8,19,26,34,68,70,      76,85,88,96,97,100,
x9y20:        11,29,70,              73,94,97,100,
x9y21:        6,9,11,29,70,          73,97,
x9y22:        63,                    76,85,88,94,97,98,
x9y23:   1,                          82,85,88,94,97,100,
x9y24:        8,                     76,85,88,100,
x9y25:
x9y26:        5,27,54,               76,85,88,91,96,97,
x9y27:        29,47,48,49,70,        73,91,
x9y28:   1,   6,7,                   76,85,94,100,
x9y29:        29,40,44,70,           79,94,97,
x9y30:   1,   5,27,54,               76,85,88,91,94,96,97,
x9y31:        29,38,70,              85,88,91,94,97,100,
```

```
x12y0:        17,18,33,61,          79,86,98,
x12y1:   1,   17,18,38,             85,88,100,
x12y2:   1,   5,27,54,56,           76,79,96,97,
x12y3:        5,27,54,56,           76,79,96,97,100,
x12y4:        17,18,33,61,          79,86,98,
x12y5:   1,   17,18,38,             85,88,100,
x12y6:   1,   5,27,54,55,56,        76,79,94,96,97,
x12y7:        5,27,54,55,56,        76,79,94,96,97,100,
x12y8:        29,40,44,70,          79,94,97,
x12y9:        29,40,44,70,          79,94,97,
x12y10:  1,   15,39,                76,79,97,100,
x12y11:
x12y12:       67,69,                79,
x12y13:       67,68,69,70,          79,
x12y14:       15,39,                76,79,94,97,100,
x12y15:       34,37,                88,97,100,
x12y16:       5,27,54,              73,88,94,97,
x12y17:       5,10,27,54,           76,85,91,94,97,
x12y18:  1,   34,44,45,63,          73,88,94,98,
x12y19:  1,   17,18,61,             79,86,98,
x12y20:  1,   34,44,45,63,          73,88,94,98,
x12y21:       18,44,45,61,          79,86,98,
x12y22:       15,34,39,             76,79,88,91,97,100,
x12y23:       18,44,45,61,          79,86,98,
x12y24:       5,27,54,              73,88,94,97,
x12y25:       27,29,33,70,          82,85,88,97,100,
x12y26:       5,26,55,56,           74,79,88,91,97,
x12y27:       34,                   76,79,88,90,91,97,
x12y28:       5,26,55,56,           74,79,88,91,94,
x12y29:  1,   5,26,55,              74,79,88,91,94,97,100,
x12y30:       15,39,                76,79,88,91,97,
x12y31:  1,   5,26,55,              74,79,88,91,94,97,100,
```

```
x16y0:   1,   11,29,70,                80,83,87,88,89,92,95,
x16y1:        11,29,70,                73,76,79,88,97,100,
x16y2:   1,                            73,76,91,94,95,97,
x16y3:   1,   33,69,                   73,91,
x16y4:   1,   41,53,                   73,74,75,76,77,78,79,94,
x16y5:                                 73,85,88,91,94,
x16y6:        64,                      73,76,79,82,85,100,
x16y7:        20,33,                   76,91,94,95,97,
x16y8:        64,                      73,76,79,82,85,95,96,100,
x16y9:   1,   2,4,5,55,                73,76,79,82,85,88,91,100,
x16y10:                                73,94,97,
x16y11:  1,   2,4,                     73,76,79,82,85,88,89,91,97,100,
x16y12:  1,   6,7,12,                  79,88,94,
x16y13:       3,29,39,70,              76,85,91,
x16y14:       15,68,69,                73,91,
x16y15:  1,   11,20,61,                79,86,98,
x16y16:  1,   5,54,56,                 76,79,100,
x16y17:       11,18,25,49,             79,
x16y18:  1,   5,26,54,56,              73,88,89,90,96,97,
x16y19:  1,   29,33,                   73,91,
x16y20:       63,                      73,88,94,98,
x16y21:                                73,86,88,97,100,
x16y22:  1,   17,18,24,36,49,          85,88,100,
x16y23:  1,   5,10,                    73,76,88,100,
x16y24:       6,9,54,56,               76,79,100,
x16y25:  1,   17,18,33,                73,94,97,
x16y26:       26,54,56,                73,88,89,90,96,97,
x16y27:       29,33,70,                73,91,
x16y28:  1,   62,                      73,88,91,94,98,
x16y29:       15,34,                   79,80,88,94,100,
x16y30:       62,                      73,88,91,94,97,98,
x16y31:  1,   6,9,15,34,               73,85,88,100,
```

x17y0:			
x17y1:	1,	49,	79,
x17y2:	1,	2,	73,85,94,
x17y3:		17,18,24,61,	79,86,98,
x17y4:	1,	2,49,50,	73,85,88,91,94,97,100,
x17y5:		6,9,15,16,29,70,	85,91,94,97,100,
x17y6:	1,	6,9,11,34,	73,76,79,88,91,92,95,100,
x17y7:	1,	6,9,11,34,	73,85,94,100,
x17y8:	1,	39,42,63,	76,91,98,
x17y9:		3,39,48,49,	76,85,91,
x17y10:	1,	11,39,63,	76,91,98,
x17y11:		3,61,	79,86,98,
x17y12:	1,	11,20,50,	79,
x17y13:		9,11,20,50,	79,
x17y14:		13,29,70,	73,76,79,88,91,94,
x17y15:		13,34,	73,76,79,88,91,94,100,
x17y16:		6,9,11,17,18,50,	79,
x17y17:	1,	6,9,11,17,18,	79,91,97,
x17y18:		29,41,42,44,70,	76,94,
x17y19:	1,	17,18,33,	76,91,94,
x17y20:	1,	17,25,37,42,44,	76,77,88,90,91,94,95,97,100,
x17y21:		25,39,	76,77,88,90,91,94,95,
x17y22:	1,	20,38,61,	79,86,98,
x17y23:		3,	73,76,79,82,85,91,95,100,
x17y24:	1,	6,9,11,25,62,65,	73,85,88,91,95,98,
x17y25:	1,	15,37,	73,79,88,89,90,91,93,94,96,97,
x17y26:	1,	62,65,	73,85,88,91,97,98,
x17y27:		15,37,	73,79,88,89,90,91,93,94,96,97,100,
x17y28:		3,5,55,	73,85,88,91,94,100,
x17y29:		3,5,55,	73,85,88,94,97,100,
x17y30:		39,43,44,	82,88,92,94,97,100,
x17y31:		2,28,39,	73,83,85,91,95,100,

x20y0:		34,42,43,	79,82,88,94,97,100,
x20y1:		22,34,	79,88,91,94,97,
x20y2:		29,40,44,70,	79,94,97,
x20y3:			73,79,97,
x20y4:		23,29,70,	
x20y5:		29,38,42,43,44,70,	73,76,91,
x20y6:	1,	62,	73,79,94,97,98,
x20y7:	1,	22,39,	73,79,86,87,88,100,
x20y8:		26,39,	79,82,91,92,97,
x20y9:		15,26,39,	79,82,91,92,97,100,
x20y10:		62,65,	73,79,91,97,98,
x20y11:		6,9,19,34,	73,76,88,97,100,
x20y12:		62,65,	73,79,91,94,98,
x20y13:	1,	9,19,34,	73,85,91,94,100,
x20y14:		17,18,33,61,	79,86,98,
x20y15:		28,34,	79,91,94,97,100,
x20y16:		28,47,49,	76,79,91,94,97,100,
x20y17:	X	23,33, 34, 55	73,76,77,78,94,95,96,100, ← 5/5-69 CORRECT
x20y18:		18,33,	73,76,79,88,94,97,100,
x20y19:		18,33,	73,76,79,88,94,97,100,
x20y20:	1,	5,55,56,67,70,	79,82,88,97,100,
x20y21:	1,	17,18,24,36,49,	85,88,100,
x20y22:	1,	31,37,	85,88,94,100,
x20y23:	1,	6,7,34,37,	76,85,94,100,
x20y24:		8,19,34,	76,85,88,100,
x20y25:	1,	6,7,19,34,	76,85,94,100,
x20y26:	1,	18,26,33,	76,85,88,97,100,
x20y27:	1,	18,26,33,	76,85,94,97,100,
x20y28:	1,	27,28,69,	73,79,88,
x20y29:	1,	27,28,32,	73,79,88,91,94,
x20y30:	1,	19,27,34,	76,85,88,91,97,
x20y31:		19,27,34,	76,85,88,91,94,97,

```
x24y0:   1,   11,29,70,                  76,
x24y1:   1,   11,29,70,                  73,76,79,88,97,
x24y2:        62,                        73,76,97,98,
x24y3:   1,   15,34,                     73,76,79,82,85,86,88,89,92,95,
x24y4:   1,   39,44,                     73,79,94,100,
x24y5:        23,38,                     73,79,97,
x24y6:   1,   5,54,                      73,76,86,87,91,92,94,97,98,100,
x24y7:   1,   28,47,48,                  79,86,91,
x24y8:        62,                        73,76,91,98,
x24y9:                                   73,76,88,91,100,
x24y10:  1,   20,33,                     76,91,94,97,
x24y11:  1,   6,9,                       82,85,90,91,94,100,
x24y12:  1,   5,9,34,38,43,              73,76,88,91,94,
x24y13:       5,9,34,38,43,              73,76,88,91,94,100,
x24y14:  1,   20,33,                     76,91,94,97,
x24y15:       17,18,38,                  73,90,91,94,97,100,
x24y16:  1,   17,18,24,36,49,            85,88,100,
x24y17:  1,   5,10,12,                   76,85,91,94,97,
x24y18:       6,11,20,50,                79,
x24y19:  1,   6,9,11,20,50,              79,
x24y20:  1,   11,29,70,                  76,85,91,97,100,
x24y21:  1,   11,29,70,                  76,85,91,97,100,
x24y22:  1,   5,26,55,                   73,74,88,94,97,
x24y23:  1,   5,26,55,                   73,74,88,94,97,
x24y24:  1,   17,18,24,36,49,            85,88,100,
x24y25:       23,28,                     79,
x24y26:  1,   17,18,38,                  73,91,94,97,100,
x24y27:       17,18,38,                  76,85,88,94,
x24y28:       5,9,11,34,                 79,97,100,
x24y29:       5,9,11,17,18,61,           79,86,98,
x24y30:       6,9,                       82,85,91,94,100,
x24y31:       6,9,                       76,85,88,94,100,
```

```
x28y0:          63,                     73,76,79,98,
x28y1:                                  73,76,79,91,94,97,100,
x28y2:                                  73,76,79,86,87,94,97,
x28y3:                                  73,85,88,91,94,
x28y4:    1,                            73,94,
x28y5:          23,38,                  73,79,97,
x28y6:    1,    17,18,61,               79,86,98,
x28y7:    1,    17,18,38,               73,76,94,97,100,
x28y8:    1,    25,42,43,               73,79,91,
x28y9:          25,43,44,               73,79,91,100,
x28y10:   1,                            73,76,79,86,87,91,94,97,
x28y11:         3,61,                   79,86,98,
x28y12:   1,    62,65,                  73,76,79,91,94,98,
x28y13:                                 85,88,91,94,100,
x28y14:         17,18,20,61,            79,86,98,
x28y15:   1,    22,29,70,               76,85,97,
x28y16:         5,26,55,                73,74,79,88,94,99,
x28y17:         5,26,55,                73,74,88,94,97,100,
x28y18:   1,    21,34,                  76,100,
x28y19:         21,34,                  100,
x28y20:         17,18,44,45,            85,88,100,
x28y21:   1,    5,26,55,                73,74,88,94,97,
x28y22:   1,    17,18,43,45,61,         79,86,98,
x28y23:   1,    17,36,61,               79,86,98,
x28y24:         5,26,55,                73,88,94,97,
x28y25:         52,53,                  79,
x28y26:         15,29,70,               97,100,
x28y27:   1,    51,                     79,
x28y28:         6,34,38,                79,88,91,100,
x28y29:         51,53,                  79,
x28y30:         17,18,20,43,45,61,      79,86,98,
x28y31:         51,52,                  79,
```

x31y0:	1,		73,76,79,82,85,86,87,88,
x31y1:		29,40,44,70,	79,94,97,
x31y2:	1,	11,23,	79,97,
x31y3:			79,
x31y4:		29,30,69,70,	82,94,97,
x31y5:		30,67,68,69,	79,
x31y6:		18,30,	79,
x31y7:			97,
x31y8:	1,	30,48,50,	79,
x31y9:	1,	3,29,48,49,50,70,	73,85,88,91,94,97,
x31y10:		6,9,29,30,69,70,	73,85,88,
x31y11:	1,	3,17,18,48,49,50,61,	79,86,98,
x31y12:	1,	29,30,69,70,	85,94,97,100,
x31y13:	1,	29,30,	88,96,97,100,
x31y14:	1,	17,18,30,	79,
x31y15:	1,		79,97,
x31y16:	1,	47,48,68,69,	73,94,
x31y17:		29,40,44,70,	79,94,97,
x31y18:		47,50,70,	73,76,79,82,85,91,94,95,97,100,
x31y19:	1,	2,	73,76,79,89,90,97,100,
x31y20:	1,	68,	82,94,97,
x31y21:		28,67,68,69,	79,
x31y22:		18,28,	79,
x31y23:			
x31y24:	1,	28,48,50,	79,
x31y25:	1,	2,5,55,	73,89,90,94,100,
x31y26:	1,	6,9,68,	73,85,88,
x31y27:		2,	73,76,79,88,91,97,
x31y28:		68,	85,94,97,100,
x31y29:			
x31y30:	1,	70,	82,94,97,
x31y31:		29,40,44,70,	79,94,97,

CORRECTION TO RCSL: 51-VB306

BIT PATTERN IN THE MICROPROGRAM STORE FOR THE RC 4000 COMPUTER

Page 13:

x17y1: 1, 49, 79,

should be

x17y1: 17,18,20,43,45,49, 79,

Page 14:

x20y17: 1, 23,33, 73,76,77,78,94,95,96,100,

should be

x20y17: 23,34,58, 73,76,77,78,94,95,96,100,

RCSL: 51-VB357
Author: Allan Giese
Edited: March 1969

CORE STORE CONTROLLER FOR

THE RC 4000 COMPUTER

A/S REGNECENTRALEN
Falkoneralle 1
Copenhagen, F.

CONTENTS:

1. CORE STORE CHARACTERISTICS.

The first 65,536 words of core store are housed within the main frame of the central processor and this size represents the maximum capacity for a fully developed core store unit. Another core store unit, placed in a separate cabinet, may be added to the system, thus giving a total capacity of 131,072 words. Each unit has the following characteristics:

Manufacture:	AMPEX
Type:	RG 1200
Capacity:	4,096 words - 65,536 words
Word Length:	28 bits
Cycle Time:	1200 nanoseconds (Read-Restore or Clear-Write) 1350 nanoseconds (Read-Modify-Write)
Access Time:	450 nanoseconds
Ambient Temperature:	0 degree C to +50 degrees C
Logic Levels:	Logical 1 = 2.2 volts to 5.4 volts Logical 0 = 0 volt to 0.8 volt
Input Termination:	All input lines are terminated with 100 ohms.
Output Driver:	All output lines are capable of driving a termination of 100 ohms.
Operational Modes:	Read-Restore Clear-Write Read-Modify-Write

Registers:

STaddr(6:22):	The address register has up to 16 bits.
STdata(0:27):	The input/output register has a data width of 28 bits. STdata(27) is the parity bit.

Control Signals:

SOC: Start Output Cycle.
A signal on this line initiates a storage operation. In this operation, the store is required to make available on the data output lines, the word in the store which is associated with the address provided.

SIC: Start Input Cycle.
A signal on this line initiates a storage operation in which the store is required to accept the data presented on the data input lines and store the accepted data in the location associated with the address provided.

RMW: Read Modify Write.
A logical 1 level on this line permits a SOC to initiate the read portion of a read-write operation. After the processor has modified the output data, a SIC will initiate the write portion of the cycle to store the new data at the address read.
A logical 0 level on this line permits read-restore and clear-write operations to be initiated by the SOC and the SIC, respectively.

Power Protection: This feature is designed to safeguard the contents of the store in the event that the AC input power is interrupted or the internal DC power exceeds predetermined limits.

Temperature Protection: The power will be shut down if the temperature rises above the operation temperature.

More detailed information can be found in the manuals published by AMPEX.

2. DESIGN CONSIDERATIONS.

Transfer of data between the core store and the registers of the Arithmetic Unit could have been controlled by the microprogram in a step by step fashion. This simple method, however, leads to a solution where the speed of the core store would have been drastically reduced, because, in this case, the time interval between successive commands would have been 500 nanoseconds; viz. the repetition rate of the microprogram store.

We have therefore aimed at a solution where the microprogram only initiates the data transfer while the actual control is left to a hardware device, called the Store Controller (STC). In order to optimize the overall performance, the STC can interrupt the running microprogram and guide it to the start address (x31y31) for the Instruction Exception routine. x31y31 is inserted in the micro address register (MAR) when the signal Fixed Address becomes one.

Input/output devices such as magnetic drum stores, magnetic disc stores, and magnetic tape stations, which transmit large volumes of data at high speeds, are connected to the High-Speed Data Channel (HDC). This channel provides input/output directly to and from the internal core store on a cycle-stealing basis. The data transfer between the store and the data channel is controlled by the STC whereas communication between the channel and the peripheral devices is supervised by the HDC logic.

The following seven commands control the operation mode of the STC:

(1)	MC(61);	Read Instruction. The program initiates the STC.
(2)	MC(62);	Read Data. The program initiates the STC.
(3)	MC(63);	Read Split. The program initiates the STC. Must be followed by (4).
(4)	MC(64);	Split Write. Must be preceded by (3).
(5)	MC(65);	Double. Specifies the address to be taken from BR.
(6)	HDC Call;	The HDC initiates the STC.
(7)	HDC Read Call;	Controls direction of data flow.

The first five of the above-mentioned commands are microprogram controlled, whereas the two last commands are reserved for the HDC.

Read Instruction:

This command is used when the microprogram, after completion of an instruction, wants to fetch the next instruction. Since the address of this new instruction is determined by IC, a special address path is provided from IC to STaddr. After the address has been staticised in STaddr, the STC generates a SOC signal and 450 nanoseconds afterwards, the next instruction is read out from the core store and strobed into STdata. The contents of STdata(0:26) are then transferred to the registers SB and PK. In addition, the left-most 12 bits of STdata are also read into FR in order not to lose speed. The information of FR is namely used by the microprogram in the immediately following microinstruction.

The STC also takes care that the programmer does not attempt to execute a protected instruction when the previous instruction was unprotected. In this case of protection violation, the microprogram is forced to location x31y31, i.e. the Instruction Exception routine.

If the interrupt request signal (Itr) is 1, the Read Instruction command is ignored, - the only exception being the Modify Next Address instruction (AM). This exception is due to the fact that this instruction and the subsequent instruction are inseparable.

Execution time is 1500 nanoseconds.

Read Data:

A Read Data command initiates a store cycle where the selected address is taken from SB. This is in accordance with the fact that the address part of an instruction is placed in SB when the Read Instruction is accomplished. After the access time has elapsed, STdata(0:26) is transferred to SB and PK.

A Read Data Double command is similar to the Read Data command, except that the address is taken from the BR register.

Execution time is 1500 nanoseconds.

Read Split:

A Read Split command controls the first half cycle of a read-modify-write operation, and it must always be followed by a Split Write command, which completes the second half cycle. The address is determined by SB and the selected data are transferred to SB and PK.

In every read-modify-write operation the STC must test whether the dataword is protected or not. This is done by using PK as an index to select a bit within the PR register, and this bit determines then the protection status. More precisely, the dataword is protected if and only if, the variable PROTECT:= PR(PK) equals one. An attempt to violate the protection system is

detected if the expression

PROTECT ∨ -,MMode

becomes one, in which case SB and PK are blocked. This means that the micro commands which open for the data flow from the Arithmetic Bus to the two registers are overruled. By this method, new data are only applied to SB and PK if the programmer does not violate the protection rules.

A Read Split Double command is similar to the Read Split command, except that the address is taken from the BR register.

Execution time is 1500 nanoseconds.

Split Write:

Split Write initiates the writing of data from SB and PK plus the generated parity bit (odd parity) into STdata, from which it is written into the core store location.

Execution time is 1000 nanoseconds.

HDC Call:

This command is a request from the High-Speed Data Channel to the core store for a memory cycle, and the request is honoured as soon as the current core store cycle has come to an end. Data are transferred from the core store to the HDC if the signal HDC Read Call is true (referred to the time where the memory cycle is started) and in reverse direction if the signal is false. The protection bits are not altered in this mode of operation.

Execution times are 1500 nanoseconds (HDC Read Call = 1) and 2000 nanoseconds (HDC Read Call = 0).

Parity Control and Generation:

Parity Control and Parity Generation for core store words are automatically carried out by the STC. In the event of an error the running program is instantaneously stopped and control is handed over to the operator.

Address exceeds the Word Limit for the Core Store:

If an instruction in a program refers to a non-existing core store location, it is detected by the STC which then inhibits all core store calls. This situation is also brought to the programmers attention by an interruption of the running program. The interruption is a consequence of that the STC sets Fixed Address to one.

If the HDC refers to locations outside the store, the SOC and SIC control signals are not suppressed because of lack of time, but the parity check circuitry is then disabled. It has not been found necessary that the STC should send a warning message, since the HDC addresses are normally defined by the monitor system.

Address < 4:

The working registers are addressable as the first four words of the store. An instruction can therefore specify a correct address which is not a core store address. In this case, the contents of the selected W register and its protection bits act as the 27 left-most bits of STdata. Parity check is in this connection irrelevent.

A HDC request with an address equals to 0,1,2, or 3 has the same effect as if the address was outside the core store limit.

3. IMPLEMENTATION.

[Dwg. No. 11199] Block Diagram
[Dwg. No. 11200:11204] Timing Diagrams

The STC logic is built up around a Priority System and two counters, called TB and TBhdc.

3.1. Priority System.

[STC - 001]

The HDC will effect a transfer in the next storage cycle provided the HDC Call does not occur later than one of the micro commands for the core store. The maximum delay between a HDC Call and the start of the corresponding memory cycle is therefore 2.0 microseconds, namely the longest time which is necessary for an already started read-modify-write operation to complete.

The Priority System is physically implemented by the two bistable elements STHdc and STcpu. STHdc = 1 and STcpu = 0 when the HDC occupies the store and the reverse is true when the CPU has access (STcpu is first set to one, 500 nanoseconds after the CPU has gained access). Both bits are forced to zero during the startup procedure (CPU Power OK = 0).

```
begin
  register STHdc(0:0), STcpu(0:0);
Time 0:
  ⫲ STHdc:= HDC Call ∧ -,TB(0) ∧ -,STcpu ∧ -,STHdc ∨ CPU Power OK ∧ STHdc
  ⫲ STcpu:= TB(0) ∧ -,STcpu ∨ CPU Power OK ∧ STcpu;
  wait 240;
Time 240:
  if TBhdc(3) = 1 then STHdc:= 0;
  if TB(5) = 1 then STcpu:= 0;
  wait 260; goto Time 0
end;
```

3.2. The Control Element, HDC Read.

[STC - 001]

HDC Read may change except when a HDC memory cycle is in progress.

```
begin
  register HDC Read(0:0);
Time 475:
  if TBhdc(0,1) = b00 then ⫲ HDC Read:= HDC Read Call;
  if TBhdc(0,1) = b01 then ⫲ HDC Read:= HDC Read Call ∧ HDC Read;
  if TBhdc(1) = b1 then ⫲ HDC Read:= HDC Read;
  wait 500; goto Time 475
end;
```

3.3. The Control Element, STCruns.

[STC - 001]

The bistable STCruns is 1 as long as a core store cycle is in progress. In this state, repeatedly requests for store accesses, either from the microprogram or the HDC, are rejected. When the running cycle comes to an end, the STCruns is reset and the store is ready to commence anew.

During the startup procedure, the STCruns is forced to attain the 0 value.

```
begin
  register STCruns(0:0);
Time 0:
  ⫲ STCruns:= TB(0) ∨ TBhdc(0) ∨ STCruns ∧ -,TB(5) ∧ -,TBhdc(3);
  wait 500; if CPU Power OK = 0 then STCruns:= 0; goto Time 0
end;
```

3.4. Time Base Counter, TB.

[STC - 001:002]

The 6-bit time base counter, TB(0:5), controls the STC timing if the store cycle is initiated by a micro command. TB counts in three different sequences depending on the following parameters:

(1) Read Instruction or Read Data. Address $\leq$ word limit.

(2) Read Split followed by Split Write. Address $\leq$ word limit.

(3) Read Instruction, Read Data, or Read Split. Address $>$ word limit.

The proper timing for the three modes of operation appears from the Timing Diagrams.

The start condition for TB(0:5) is

TB(0:5) = b000 100.

The Split Write command, which activates the write phase of the read-modify-write cycle, has only effect when TB(3) = 0.

Let us assure ourselves that the counter arrives to the start condition during the startup procedure. TB(3) is set to 1 because Power OK = 0, and TB(0,4):= b00 as a consequence of that the micro commands MC(61:64) are all zero. TB(1,2,5) will then, after a few clock periods, attain the value 0; and the start situation is obtained.

3.5. Time Base Counter, TBhdc.

[STC - 001:002]

The 4-bit time base counter, TBhdc(0:3) controls the STC timing if the store cycle is initiated by a HDC Call. TBhdc counts in two different sequences depending on the following parameters:

(1) HDC Call. HDC Read = 1. (Read from core store)

(2) HDC Call. HDC Read = 0. (Write into core store)

The proper timing for the two modes of operation appears from the Timing Diagrams.

The start condition for TBhdc(0:3) is

TBhdc(0:3) = b0000.

Let us assure ourselves that the counter arrives to the start condition during the startup procedure. This is easily seen if we first consider TBhdc(0,1) which have the state transitions

10 -> 11 -> 01 -> 00.

TBhdc(1) = 0 implies that TBhdc(2) is reset and this, on the other hand, implies that also TBhdc(3) is reset, - and the counter is in the start state.

4. PROGRAM DESCRIPTION.

```
CORE STORE CONTROLLER:
begin
  register
    SThdc(0:0), STcpu(0:0), HDC Read(0:0), STCruns,
    TB(0:5), TBhdc(0:3),
    AddrError(0:0), AddrST(0:0), HA(21:23), MCB(61:61), MCB(62:63),
    ST1, ST1data(0:27), ST1addr(7:22), ST2data(0:27), ST2addr(7:22),
    HDCdata(0:26);
  register set
    HDCaddr(7:22) = HDCdata(7:22);
  integer word limit;
  register array
    ST1[4:65535](0:27), ST2[0:word limit-65536](0:27);
  Boolean SOC1, SIC1, RMW1, SOC2, SIC2, RMW2;

  comment Address transfer from CPU to Core Store;
  comb net GiSTfICaddr(0:0);
    begin
      comment This signal sets up the path from ICaddr to STaddr as a conse-
        quence of a Read Instruction command provided the running program is
        not interrupted. The AM instruction has the function code 9;
      GiSTfICaddr:= -,SThdc ∧ -,STCruns ∧ (MC(61) ∧ -,Itr ∨ MC(61) ∧ FR(0:5) = 9)
    end;
```

```
comb net GiSTfSBaddr(0:0);
  begin
    comment This signal sets up the path from SBaddr to STaddr as a conse-
      quence of a Read Data or Read Split command;
    GiSTfSBaddr:= -,SThdc ∧ -,STCruns ∧ -,MC(65) ∧ (MC(62) ∨ MC(63))
  end;
comb net GiSTfBRaddr(0:0);
  begin
    comment This signal sets up the path from BRaddr to STaddr as a conse-
      quence of a Read Data Double or Read Split Double command;
    GiSTfBRaddr:= -,SThdc ∧ -,STCruns ∧ MC(65) ∧ (MC(62) ∨ MC(63))
  end;
comb net GiSTfHDCaddr(0:0);
  begin
    comment This signal sets up the path from HDCaddr to STaddr as a conse-
      quence of a HDC Call;
    GiSTfHDCaddr:= SThdc
  end;

comment Data transfer from Core Store to STBUS in CPU;
comb net GiSTBUSfST1(0:0);
  begin GiSTBUSfST1:= AddrST ∧ ST1 ∧ (TB(1) ∨ TB(2) ∨ TBhdc(0) ∨ TBhdc(1)) end;
comb net GiSTBUSfST2(0:0);
  begin GiSTBUSfST2:= AddrST ∧ -,ST1 ∧ (TB(1) ∨ TB(2) ∨ TBhdc(0) ∨ TBhdc(1)) end;

comment Data transfer from Working Registers and PB to STBUS;
comb net GiSTBUSfW0(0:0);
  begin GiSTBUSfW0:= -,AddrST ∧ TB(1) ∧ HA(21,22) = 0 end;
comb net GiSTBUSfW1(0:0);
  begin GiSTBUSfW1:= -,AddrST ∧ TB(1) ∧ HA(21,22) = 1 end;
comb net GiSTBUSfW2(0:0);
  begin GiSTBUSfW2:= -,AddrST ∧ TB(1) ∧ HA(21,22) = 2 end;
comb net GiSTBUSfW3(0:0);
  begin GiSTBUSfW3:= -,AddrST ∧ TB(1) ∧ HA(21,22) = 3 end;

comment Data transfer from SB and PK to STBUS;
comb net GiSTBUSfSB(0:0);
  begin GiSTBUSfSB:= AddrST ∧ -,TB(2) ∧ -,TB(3) end;
```

```
comment Data transfer from HDC to STBUS;
comb net GiSTBUSfHDC(0:0);
  begin GiSTBUSfHDC:= TBhdc(2) end;

comment Data transfer from SB and PK to Working Registers and PB;
comb net STCGiBUSfSB(0:0);
  begin STCGiBUSfSB:= -,AddrST ∧ -,TB(3) ∧ MC(64) end;
comb net STCGiW0fBUS(0:0);
  begin STCGiW0fBUS:= STCGiBUSfSB ∧ HA(21,22) = 0 end;
comb net STCGiW1fBUS(0:0);
  begin STCGiW1fBUS:= STCGiBUSfSB ∧ HA(21,22) = 1 end;
comb net STCGiW2fBUS(0:0);
  begin STCGiW2fBUS:= STCGiBUSfSB ∧ HA(21,22) = 2 end;
comb net STCGiW3fBUS(0:0);
  begin STCGiW3fBUS:= STCGiBUSfSB ∧ HA(21,22) = 3 end;

comment Control Signals;
comb net Fixed Address(0:0);
  begin
    Fixed Address:= AddrError ∧ TB(0) ∨ PROTECT ∧ -,MMode ∧ MCB(61) ∧ TB(2)
  end;
comb net Check Parity(0:0);
  begin Check Parity:= AddrST ∧ (TB(2) ∨ TBhdc(0) ∧ TBhdc(1)) end;
comb net EnableSB(0:0);
  begin
    comment This combined  variable prevents, for zero values, that the con-
      tents of SB and PK can be changed. This is implemented, simply by sup-
      pressing the appropriate clock pulses. The expression becomes zero if
      the programmer attempts to change a protected word by means of an un-
      protected instruction;
    EnableSB:= -,PROTECT ∨ MMode ∨ TB(3);
  end;
```

```
sequence Time Base Counter TB;
begin
Time 0:
  ⊀ TB(1):= -,AddrError ∧ TB(0);
  ⊀ TB(3):= TB(4) ∨ TB(3) ∧ -,(TB(2) ∧ MCB(63));
  wait 190;
Time 190:
  ⊀ TB(0):= -,TB(0) ∧ (GiSTfICaddr ∨ GiSTfSBaddr ∨ GiSTfBRaddr);
  ⊀ TB(2):= TB(1);
  ⊀ TB(4):= MC(64) ∧ -,TB(3);
  ⊀ TB(5):= AddrError ∧ TB(0) ∨ TB(2) ∧ TB(3) ∨ TB(4);
  wait 310; goto Time 190
end;


sequence Time Base Counter TBhdc;
begin
  wait 190;
Time 190:
  ⊀ TBhdc(0):= SThdc ∧ -,STCruns ∨ TBhdc(0) ∧ -,TBhdc(1);
  ⊀ TBhdc(1):= TBhdc(0);
  ⊀ TBhdc(3):= -,TBhdc(0) ∧ TBhdc(2) ∨ TBhdc(0) ∧ TBhdc(1) ∧ -,TBhdc(2);
  wait 285;
Time 475:
  ⊀ TBhdc(2):= TBhdc(1) ∧ -,HDC Read;
  wait 215; goto Time 190
end;
```

```
sequence STC CONTROLLED BY MICROPROGRAM;
begin
Time 0:
  wait until STCruns = 0;
  wait 100; RMW1:= RMW2:= -,SThdc ∧ MC(63); wait 65;
Time 165:
  if GiSTfICaddr ∧ -,IC(6) ∧ IC > 3 ∧ IC ≤ word limit ∧ CPU Power OK then
    begin ST1addr:= IC(7:22); SOC1:= 1 end;
  if GiSTfICaddr ∧ IC(6) ∧ IC > 3 ∧ IC ≤ word limit ∧ CPU Power OK then
    begin ST2addr:= IC(7:22); SOC2:= 1 end;
  if GiSTfSBaddr ∧ -,SB(6) ∧ SB(0:22) > 3 ∧ SB(0:22) ≤ word limit ∧ CPU Power OK then
    begin ST1addr:= SB(7:22); SOC1:= 1 end;
  if GiSTfSBaddr ∧ SB(6) ∧ SB(0:22) > 3 ∧ SB(0:22) ≤ word limit ∧ CPU Power OK then
    begin ST2addr:= SB(7:22); SOC2:= 1 end;
  if GiSTfBRaddr ∧ -,BR(6) ∧ BR(0:22) > 3 then
    begin ST1addr:= BR(7:22); SOC1:= 1 end;
  if GiSTfBRaddr ∧ BR(6) ∧ BR(0:22) > 3 then
    begin ST2addr:= BR(7:22); SOC2:= 1 end;
  wait until TB(0) = 1; wait 40;
  comment Store original input parameters for later use;
Time 230:
  MCB(61,63):= MC(61,63); HA(23):= SB(23);
  if GiSTfICaddr then
    begin
      ST1:= -,IC(6); AddrError:= IC > word limit;
      AddrST:= IC > 3 ∧ IC ≤ word limit; HA(21,22):= IC(21,22)
    end;
  if GiSTfSBaddr then
    begin
      ST1:= -,SB(6); AddrError:= SB(0:22) > word limit;
      AddrST:= SB(0:22) > 3 ∧ SB(0:22) ≤ word limit; HA(21,22):= SB(21,22)
    end;
  if GiSTfBRaddr then
    begin ST1:= -,BR(6); AddrError:= 0; AddrST:= BR(6:22) > 3 end;
  wait 200;
Time 430:
  SOC1:= SOC2:= 0; wait 160;
```

```
Time 590:
  if Fixed Address = 1 then
    begin
      comment This condition is true for AddrError = 1. The microprogram is then
        conducted to the Instruction Exception routine;
      MAR:= 1031; goto EXIT
    end; wait 25;
Time 615:
  if AddrST then
    begin if ST1 then ST1data:= ST1[ST1addr] else ST2data:= ST2[ST2addr] end;
  wait 125;
Time 740:
  SBconPK:= 27extGiSTBUSfST1 ∧ ST1data(0:26)
          ∨ 27extGiSTBUSfST2 ∧ ST2data(0:26)
          ∨ 27extGiSTBUSfW0 ∧ W[0]conPB[0]
          ∨ 27extGiSTBUSfW1 ∧ W[1]conPB[1]
          ∨ 27extGiSTBUSfW2 ∧ W[2]conPB[2]
          ∨ 27extGiSTBUSfW3 ∧ W[3]conPB[3];
  if MCB(61) then
    begin
      comment Read Instruction command;
      FR:= 12extGiSTBUSfST1 ∧ ST1data(0:11)
         ∨ 12extGiSTBUSfST2 ∧ ST2data(0:11)
         ∨ 12extGiSTBUSfW0 ∧ W[0](0:11)
         ∨ 12extGiSTBUSfW1 ∧ W[1](0:11)
         ∨ 12extGiSTBUSfW2 ∧ W[2](0:11)
         ∨ 12extGiSTBUSfW3 ∧ W[3](0:11)
    end;
  wait 240;
Time 980:
  if Check Parity ∧ Core Store Parity Control then
    Core Store Parity Error:= if ST1 then -,odd ST1data else -,odd ST2data;
  wait 110;
Time 1090:
  if Fixed Address = 1 then
    begin
      comment This condition is true if the programmer attempts to execute a
        protected instruction provided the previous instruction was unprotected;
      MAR:= 1031; wait 410; goto Select Sequence
    end;
  wait 410;
```

```
Time 1500:
  if TB(3) = 1 then
    begin
      comment The command which initialized the STC was not a Read Split com-
        mand, and the STC will therefore finish at time 1500;
      goto Select Sequence
    end;
Time 1500:
  wait until MC(64);
  comment The new data to be stored are already staticised in SB or PK; wait 165;
Time 1665:
  if GiSTBUSfSB ∧ ST1 ∧ MC(64) then
    begin ST1data(0:26):= SBconPK; ST1data(27):= -,oddSBconPK; SIC1:= 1 end;
  if GiSTBUSfSB ∧ ST2 ∧ MC(64) then
    begin ST2data(0:26):= SBconPK; ST2data(27):= -,oddSBconPK; SIC2:= 1 end;
  wait 90;
Time 1755:
  if STCGiBUSfSB ∧ STCGiW0fBUS then W[0]conPB[0]:= SBconPK;
  if STCGiBUSfSB ∧ STCGiW1fBUS then W[1]conPB[1]:= SBconPK;
  if STCGiBUSfSB ∧ STCGiW2fBUS then W[2]conPB[2]:= SBconPK;
  if STCGiBUSfSB ∧ STCGiW3fBUS then W[3]conPB[3]:= SBconPK;
  wait 175;
Time 1930:
  SIC1:= SIC2:= 0; wait 570; goto Select Sequence
end STC CONTROLLED BY MICROPROGRAM;

sequence STC CONTROLLED BY HDC;
begin
Time 0:
  wait until STCruns = 0; wait 115; RMW1:= RMW2:= STHdc ∧ -,HDC Read; wait 50;
Time 165:
  if GiSTfHDCaddr ∧ -,HDCaddr(6) ∧ -,STCruns then
    begin ST1addr:= HDCaddr(7:22); SOC1:= 1 end;
  if GiSTfHDCaddr ∧ HDCaddr(6) ∧ -,STCruns then
    begin ST2addr:= HDCaddr(7:22); SOC2:= 1 end;
  wait until TB(0) = 1; wait 40;
  comment Store original input parameters for later use;
Time 230:
  ST1:= -,HDCaddr(6); AddrST:= HDCaddr > 3 ∧ HDCaddr ≤ word limit; wait 200;
```

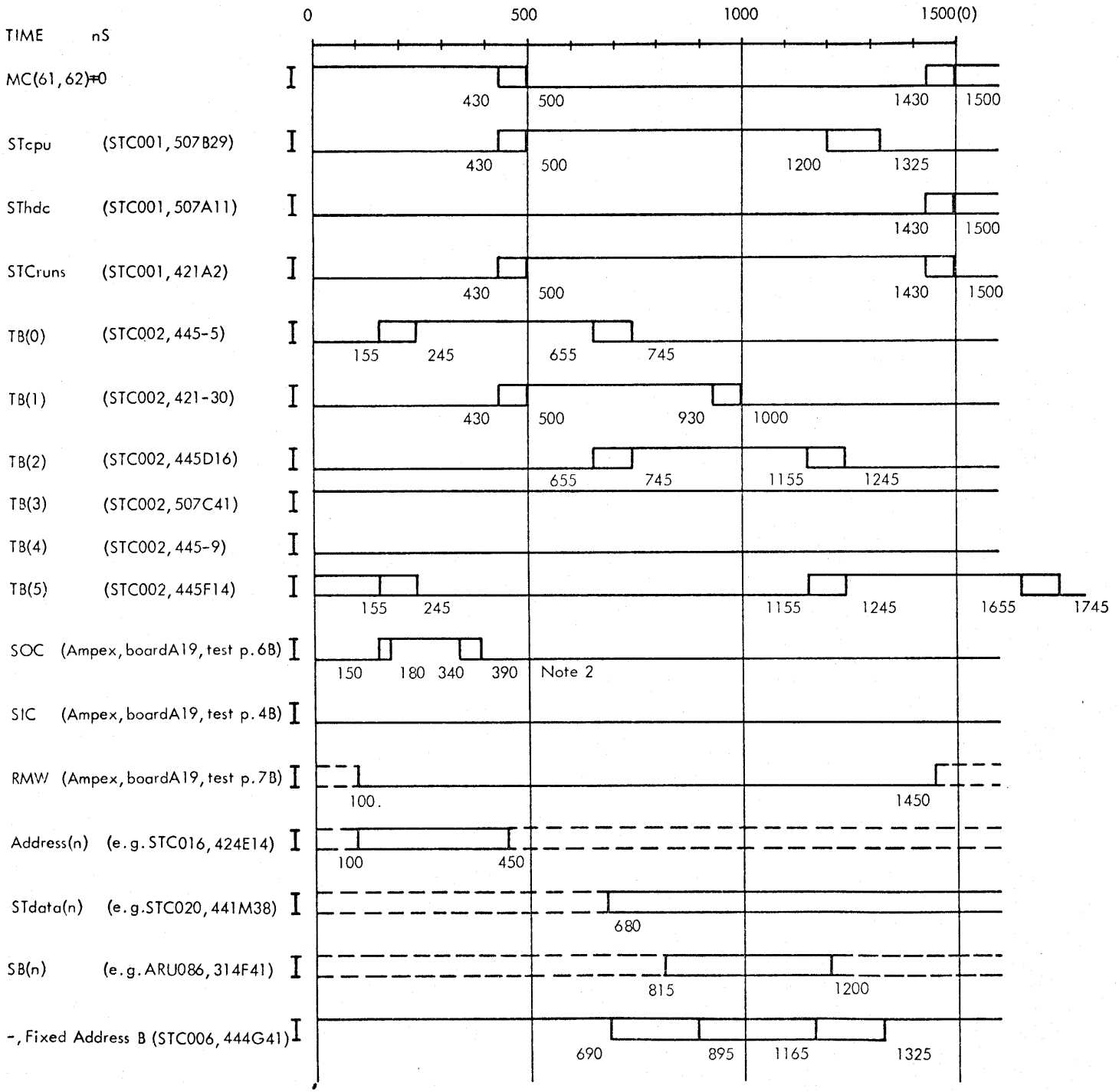
```
Time 430:
  SOC1:= SOC2:= 0; wait 185;
Time 615:
  if ST1 then ST1data:= ST1[ST1addr] else ST2data:= ST2[ST2addr]; wait 100;
Time 715:
  HDCdata(24:26):= 3extGiSTBUSfST1 ∧ ST1data(24:26)
                 ∨ 3extGiSTBUSfST2 ∧ ST2data(24:26);
  comment The 24 bits of data are available on STBUS, ready to be transferred
    to a peripheral device;
  wait 265;
Time 980:
  if Check Parity ∧ Core Store Parity Control then
    Core Store Parity Error:= if ST1 then -,odd ST1data else -,odd ST2data;
  wait 405;
Time 1385:
  if TBhdc(2) = 0 then
    begin
      comment The read cycle will finish at time 1500;
      wait 115; goto Select Sequence
    end;
  if GiSTBUSfHDC ∧ ST1 then
    begin ST1data(0:26):= HDCdata; ST1data(27):= -,odd HDCdata; SIC1:= 1 end;
  if GiSTBUSfHDC ∧ ST2 then
    begin ST2data(0:26):= HDCdata; ST2data(27):= -,odd HDCdata; SIC2:= 1 end;
  wait 250; SIC1:= SIC:= 2:= 0; wait 365; goto Select Sequence
end STC CONTROLLED BY HDC;

  start Time Base Counter TB; start Time Base Counter TBhdc;
Select Sequence:
  if HDC Call con MC(61:63) = 0 then
    begin wait 500; goto Select Sequence end;
  if HDC Call then start STC CONTROLLED BY HDC
              else start STC CONTROLLED BY MICROPROGRAM

end CORE STORE CONTROLLER;
```

TB AND CORE STORE TIMING FOR READ INSTRUCTION AND READ DATA (DOUBLE)
6 < ADDRESS < WORD LIMIT
Timing Chart

130369AG 010670MOGK 240670 AL 240670 AG



Note 2: Pulse width is 200 ± 10nS

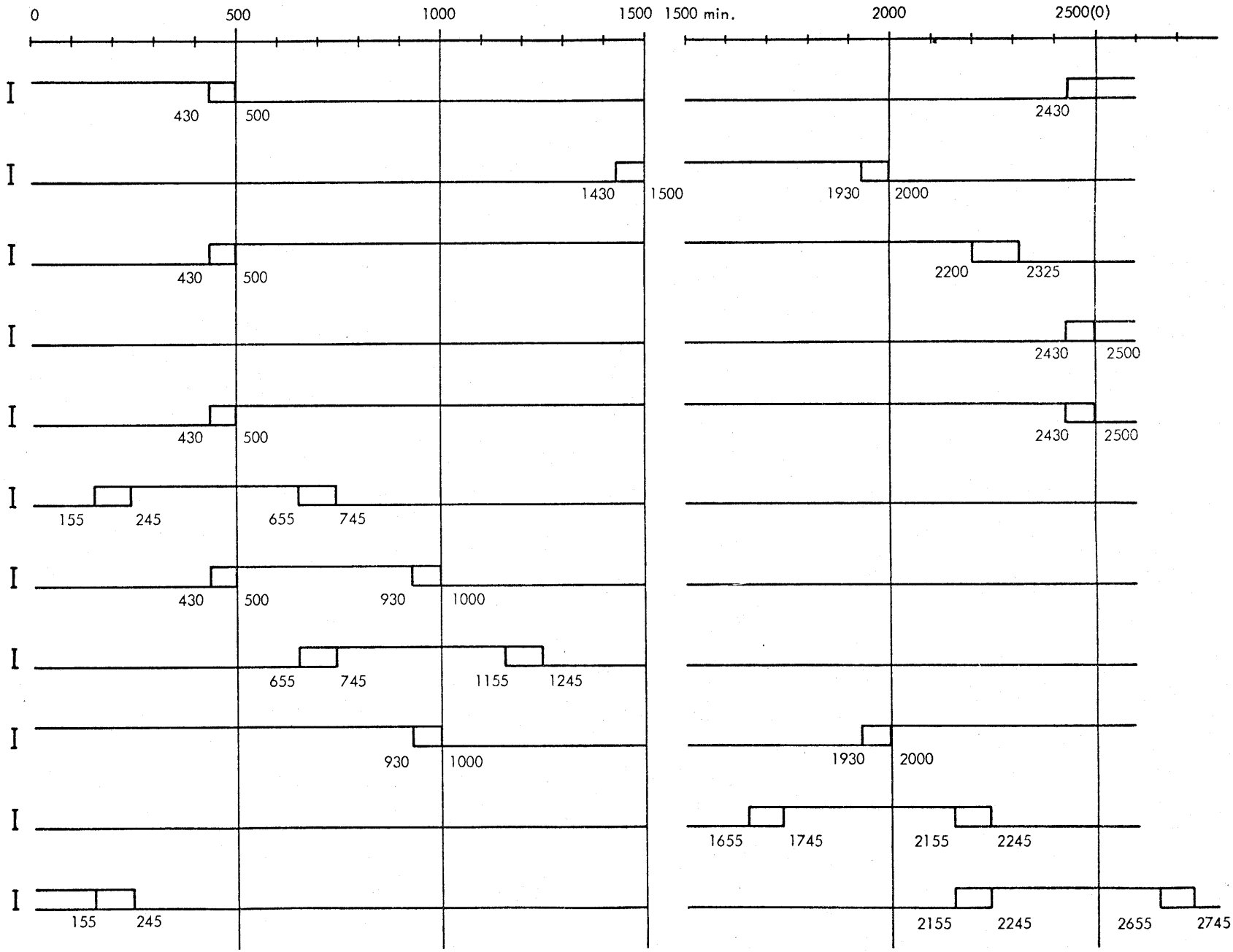
RC4000

V11201

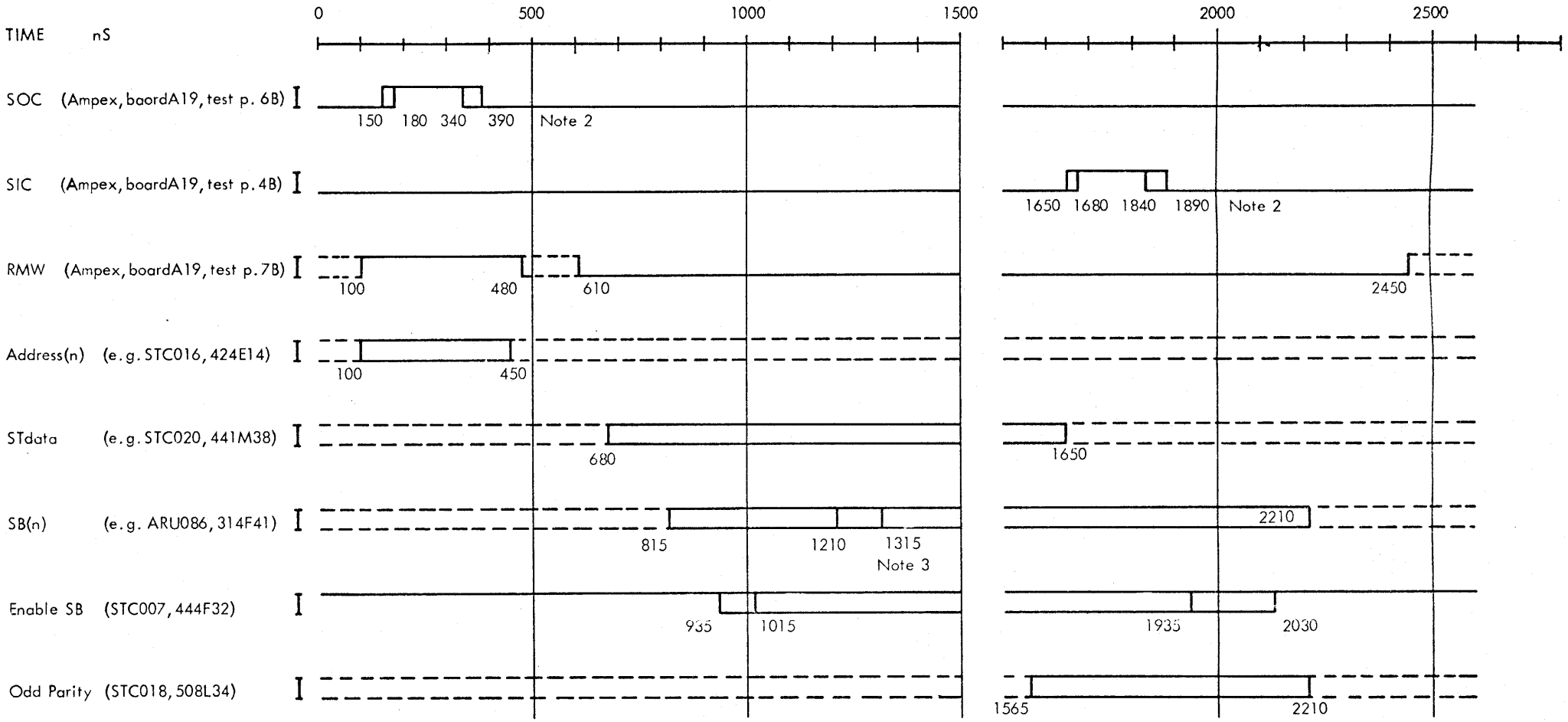
TB AND CORE STORE TIMING FOR READ SPLIT - SPLIT WRITE

6 < ADDRESS ≤ WORD LIMIT

Timing Chart



TB AND CORE STORE TIMING FOR READ SPLIT - SPLIT WRITE
6 < ADDRESS < WORD LIMIT
Timing Chart



Note 1: SOC1:(STC005, 428M40); SIC1:(STC005, 424M40); RMW1:(STC005, 347M40). If the signals are measured here, remember to subtract the cable delay which is approx. 6nS/meter.

Note 2: Pulse width is $200^{+}_{-}10$nS

Note 3: New data are strobed into SB

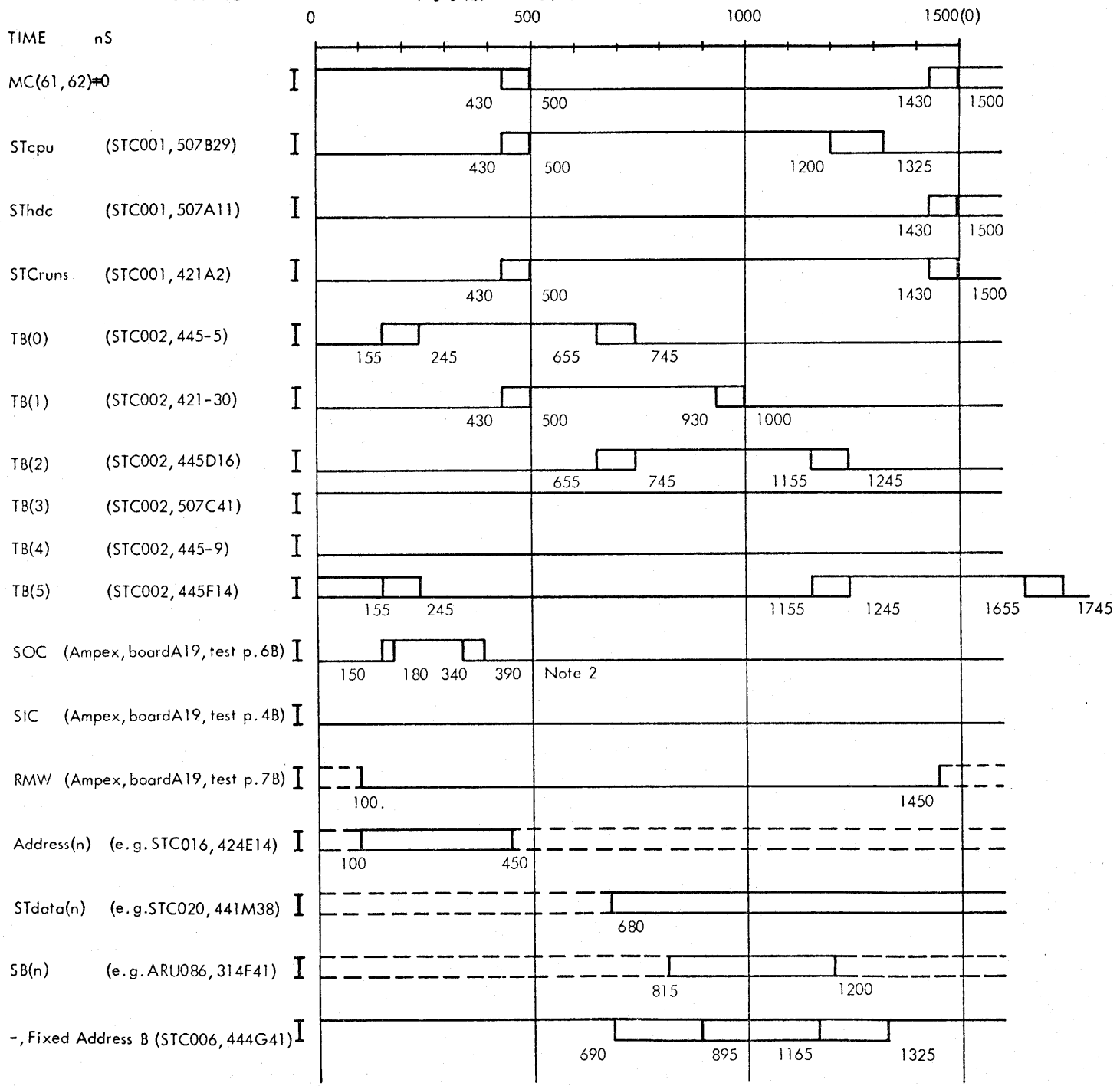
V11200 RC4000

TB AND CORE STORE TIMING FOR READ INSTRUCTION AND READ DATA (DOUBLE)

6 < ADDRESS < WORD LIMIT

Timing Chart

130369AG 010670MOGK 240670 AL 240670 AG



Note 2: Pulse width is 200 ± 10nS

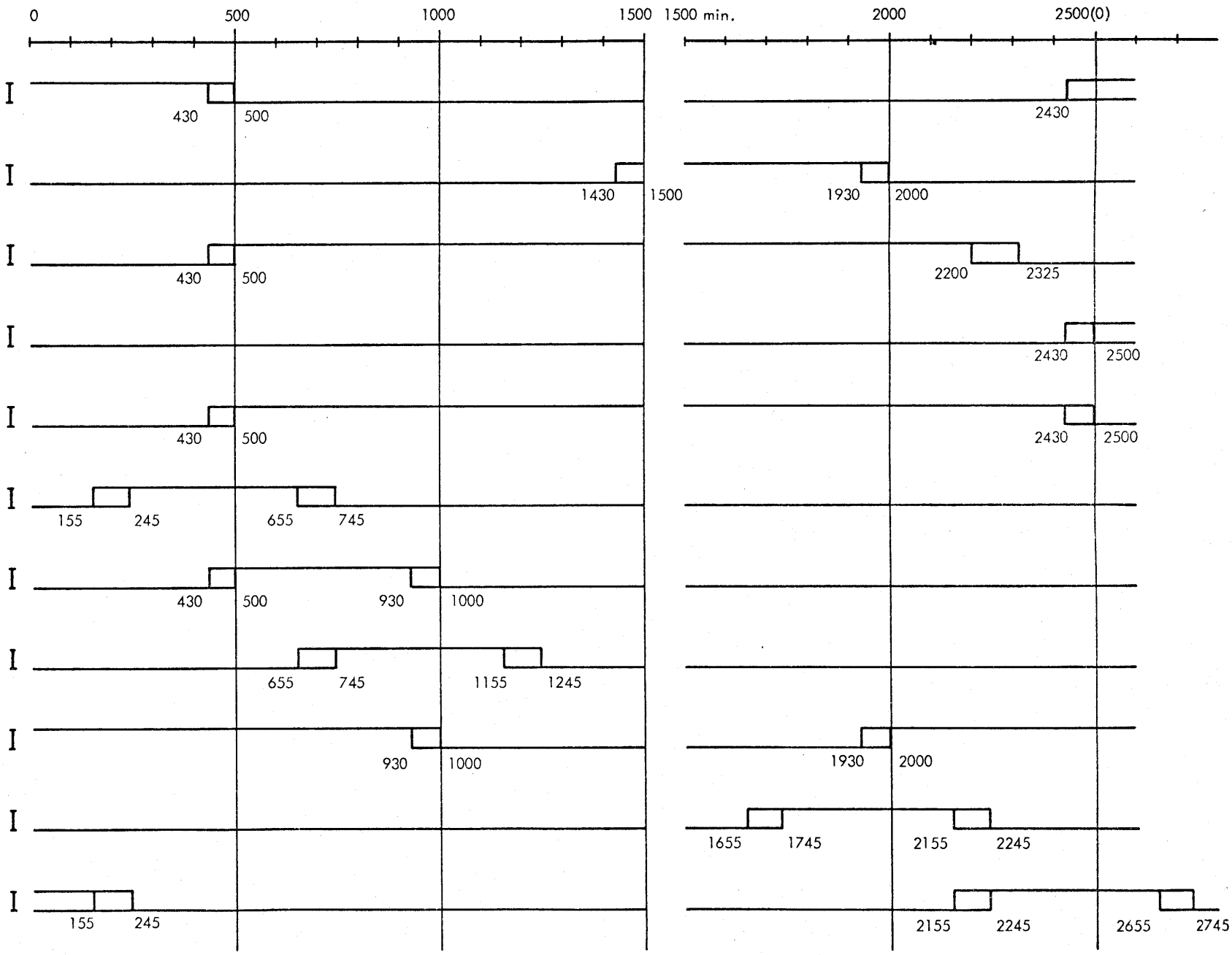
RC4000

V11201

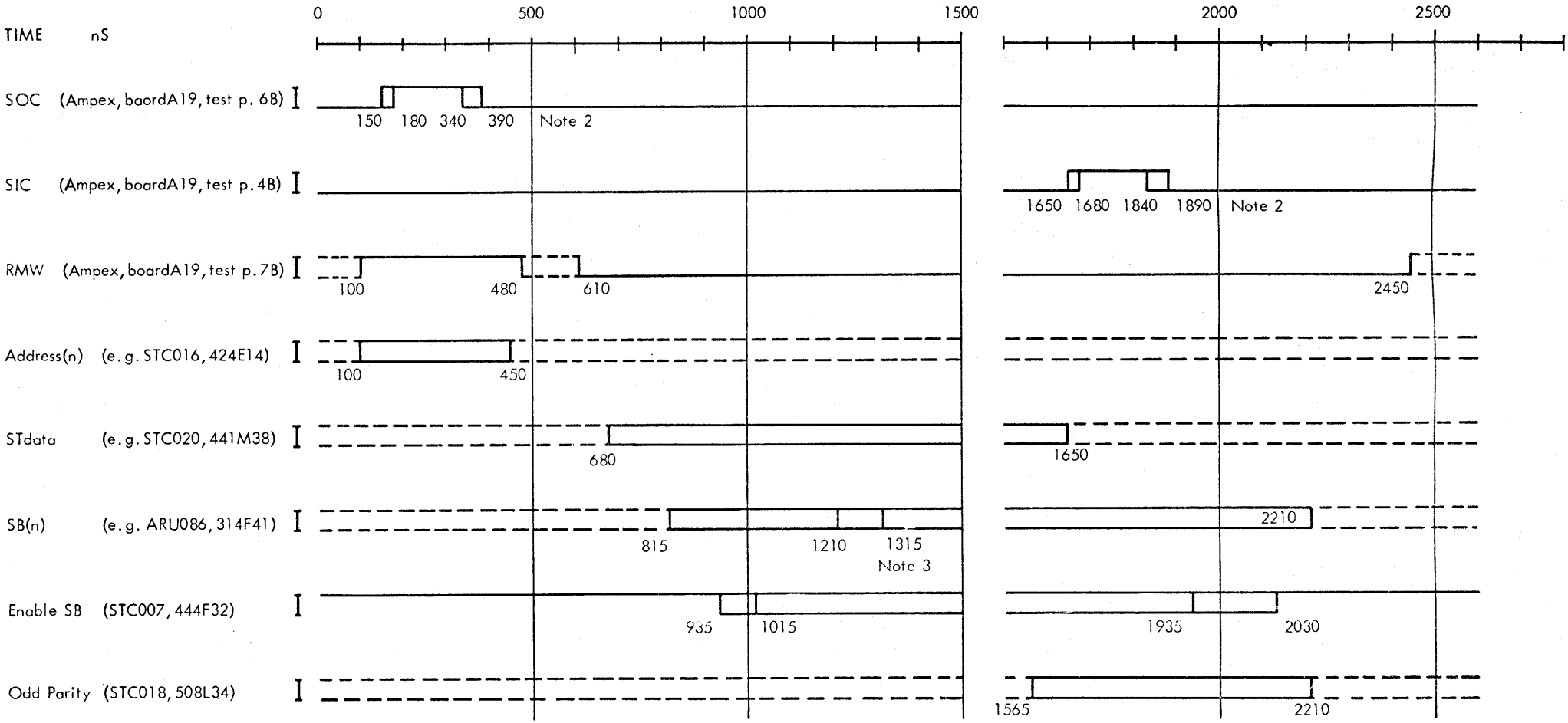
TB AND CORE STORE TIMING FOR READ SPLIT - SPLIT WRITE

6 < ADDRESS < WORD LIMIT

Timing Chart



TB AND CORE STORE TIMING FOR READ SPLIT - SPLIT WRITE
6 < ADDRESS < WORD LIMIT
Timing Chart



Note 1: SOC1:(STC005, 428M40); SIC1:(STC005, 424M40); RMW1:(STC005, 347M40). If the signals are measured here, remember to subtract the cable delay which is approx. 6nS/meter.

Note 2: Pulse width is $200^{+}_{-}10$nS

Note 3: New data are strobed into SB

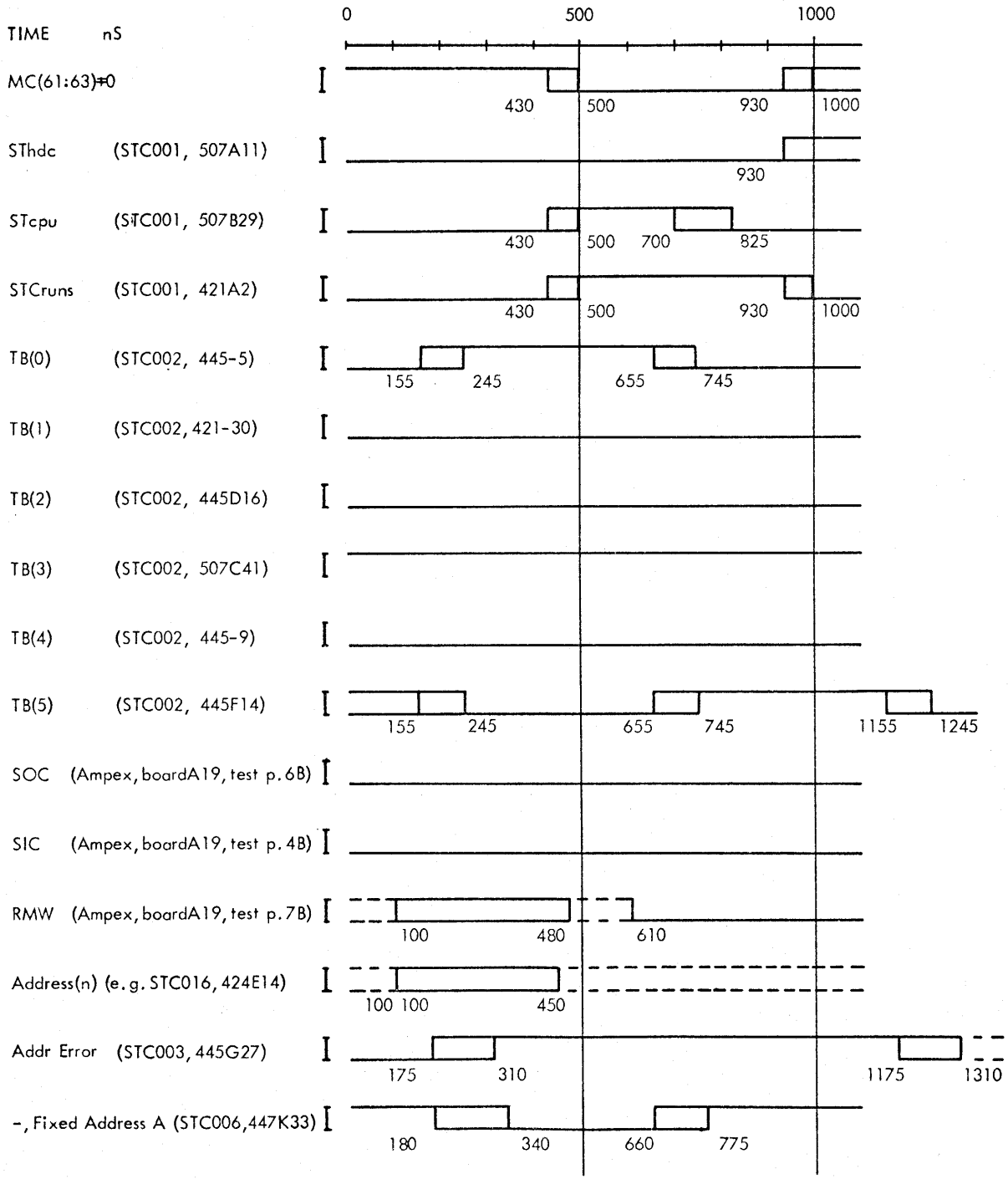
130369AG 010670MOGK 240670 dl 240670 AG

RC4000

TB AND CORE STORE TIMING FOR READ INSTRUCTION, READ DATA, AND READ SPLIT ADDRESS > WORD LIMIT

Timing Chart

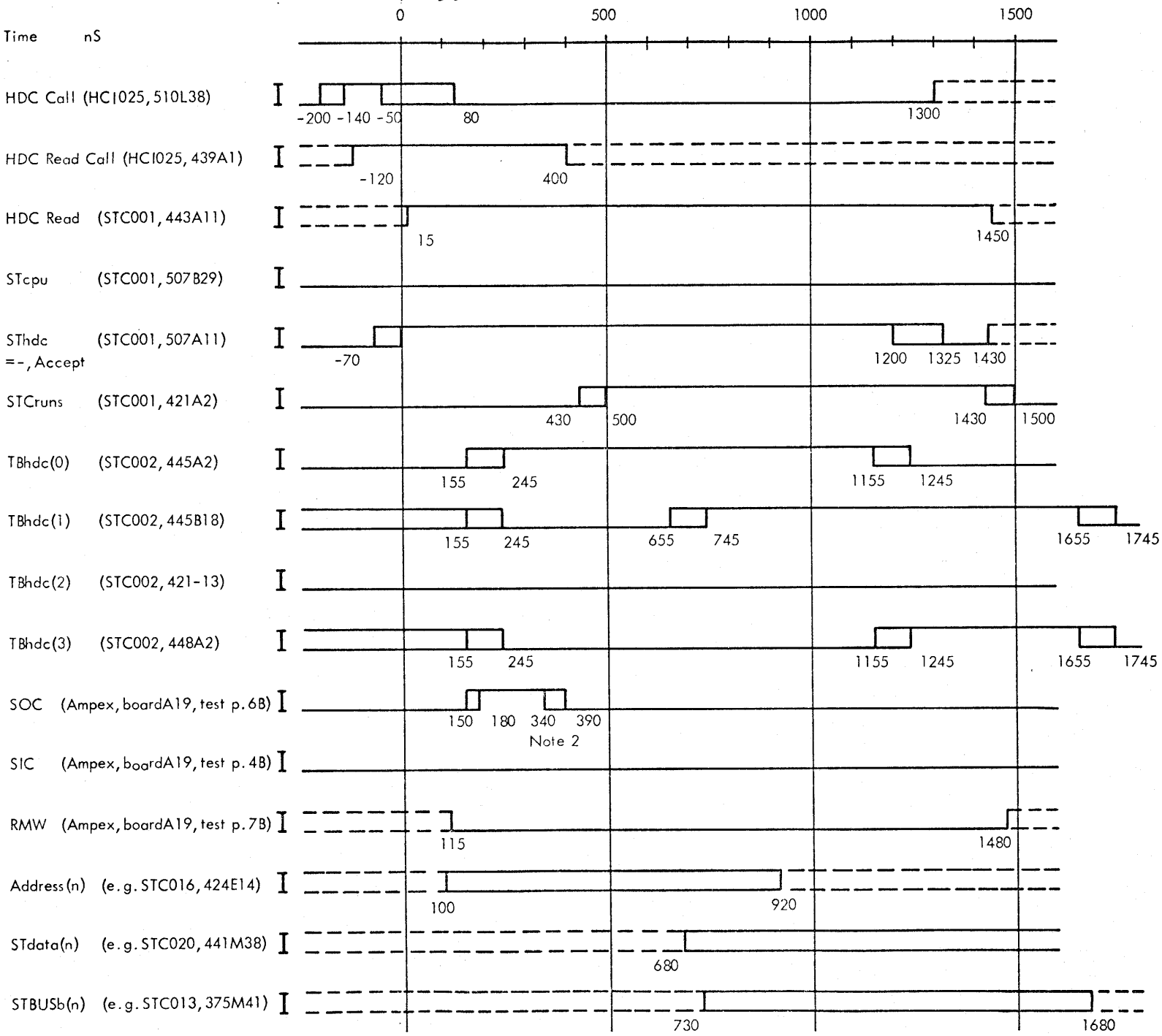
V11203



130369AG 010670MOGK 240670 ad 240670 AG

RC4000 V11204 Timing Chart

TBhdc AND CORE STORE TIMING FOR HDC READ



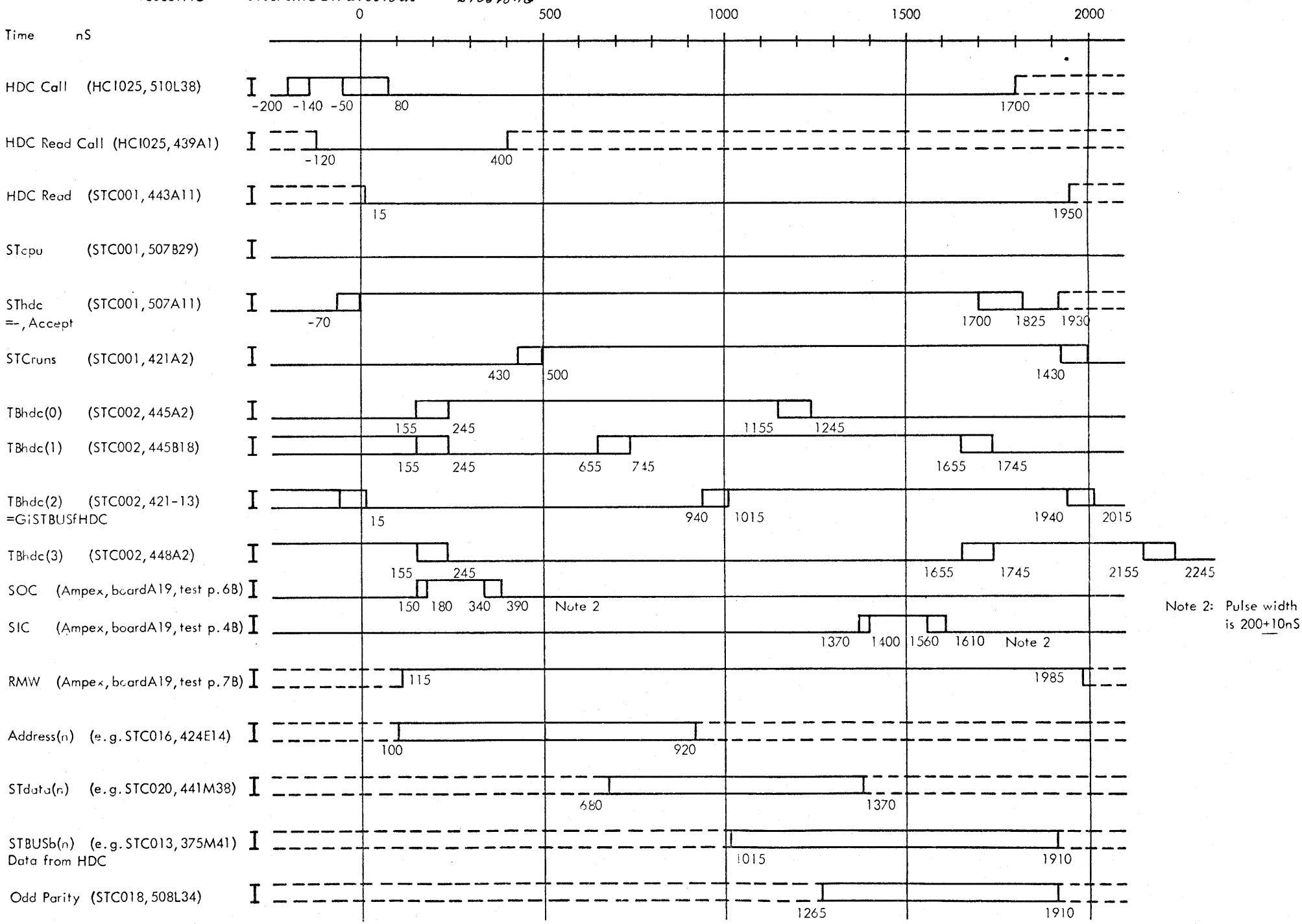
Note 2: Pulse width is 200± 10nS

130369AG 010670MOGK 240670 AL 240670 AG

RC4000 V11205

TBhdc AND CORE STORE TIMING FOR HDC WRITE

Timing Chart



Note 2: Pulse width is 200±10nS

RCSL: 51-VB720
Author: P. E. Pedersen
Edited: January 1970

CONTROL UNIT

FOR

THE RC 4000 COMPUTER

A/S REGNECENTRALEN
Falkoneralle 1
2000 Copenhagen F

CONTENTS:

PREFACE

The Control Unit consists of a Micro Program Store, MPS, and a Micro Program Controller, MPC.

This chapter starts with a comprehensive description of the microprogram stores, including realization, performance, and external control logic. Section 4 gives an examination of the clock pulse generator, the task of which is to generate the necessary timing pulses and to descripe all timing adjustments.

In section 5 we discuss what happens during the period of start-up and shutdown, and how the power supply voltages are supervised. The computer can operate in different modes, and the meaning of these modes is explained in the subsequent section. The final section is devoted to a program which gives a concise description of the Control Unit, interpreted in its entirety.

1. MICROPROGRAM STORE

1.1. Introduction

The RC 4000 Computer is controlled by a microprogram residing in the Read-Only Store (MPS). MPS is organized like a general store having a capacity of 1024 words, each of 100 bits. The cycle time is 500 nanoseconds and the access time is approximately 300 nanoseconds. Control of the computer is accomplished in the following way (confer Fig. 1): The micro address register (MAR) selects the word to be read, and 300 nanoseconds later, the selected word is ready to be read into the output register. This 100-bit register consists actually of two registers, namely the Micro Command Register (MC), and the Jump Selector Register (JS). MC is a 70-bit register, and its bit pattern determines the micro operations to be executed. In consequence of odd parity for MPS words, one of the 70 MC bits is used as a parity bit. The 30 bits of JS select 10 out of 80 possible branch conditions, which form the next address to be read into MAR. This process is repeated every 500 nanoseconds, hence the computer is said to operate in a synchronous mode having a clock frequency of 2 Mc. The abovementioned is also expressed in this program:

```
sequence MICROPROGRAM STORE;
begin
  register MAR(0:9), MC(1:70), JS(71:100);
  register array MPS[0:1023](1:100);
Time 0: Next Word:
  MCconJS:= MPS[MAR]; wait 90;
Time 90:
  MAR:= next address;
  comment The following 110 nanoseconds are used to decode the address and to
    generate the output;
  wait 410; goto Next Word
end MICROPROGRAM STORE;
```

1.2. Word Selection

(MPC 002:011)
(MPS 001:003)

The next microaddress is evaluated and read into the MAR register at maximum 116 nanoseconds after the JS register has been set up.

The word selection includes a full binary decoding of the nine most significant bits in the MAR register. As MAR(9) is not implied in the decoding, two word addresses, 2N and 2N+1 where $0 \leq N \leq 511$, will select the very same double-word 2N.

The first level in the decoding includes three 3-bit binary decoding circuits concerning MAR(0,1,5), MAR(2,3,4), and MAR(6,7,8). Outputs from a group are 8 lines having logical zero on the 7 not-selected outputs and a logical one on the selected output line.

The next decoding level decodes the group outputs into a maximum of $8 \times 8 \times 8 = 512$ double-words. The logical values of the output signals are one for all the not-selected double-word outputs and zero for the selected double-word output line.

1.3. Data Detection

(MPS 004:016, 049:130)

The data detection circuit consists of 100 NAND-gates, one multi input NAND gate for each of the 100-bit positions in MPS. The size of the NAND-gate at output n is given by the total amount of one-bits in position n in all the stored words.

Due to the fact that MAR(9) is not implied in the word selection, it has to be included in the data detecting. This is done by means of a 2-input multiplexer, steared by MAR(9) so that all one-bits in even address words have ac-

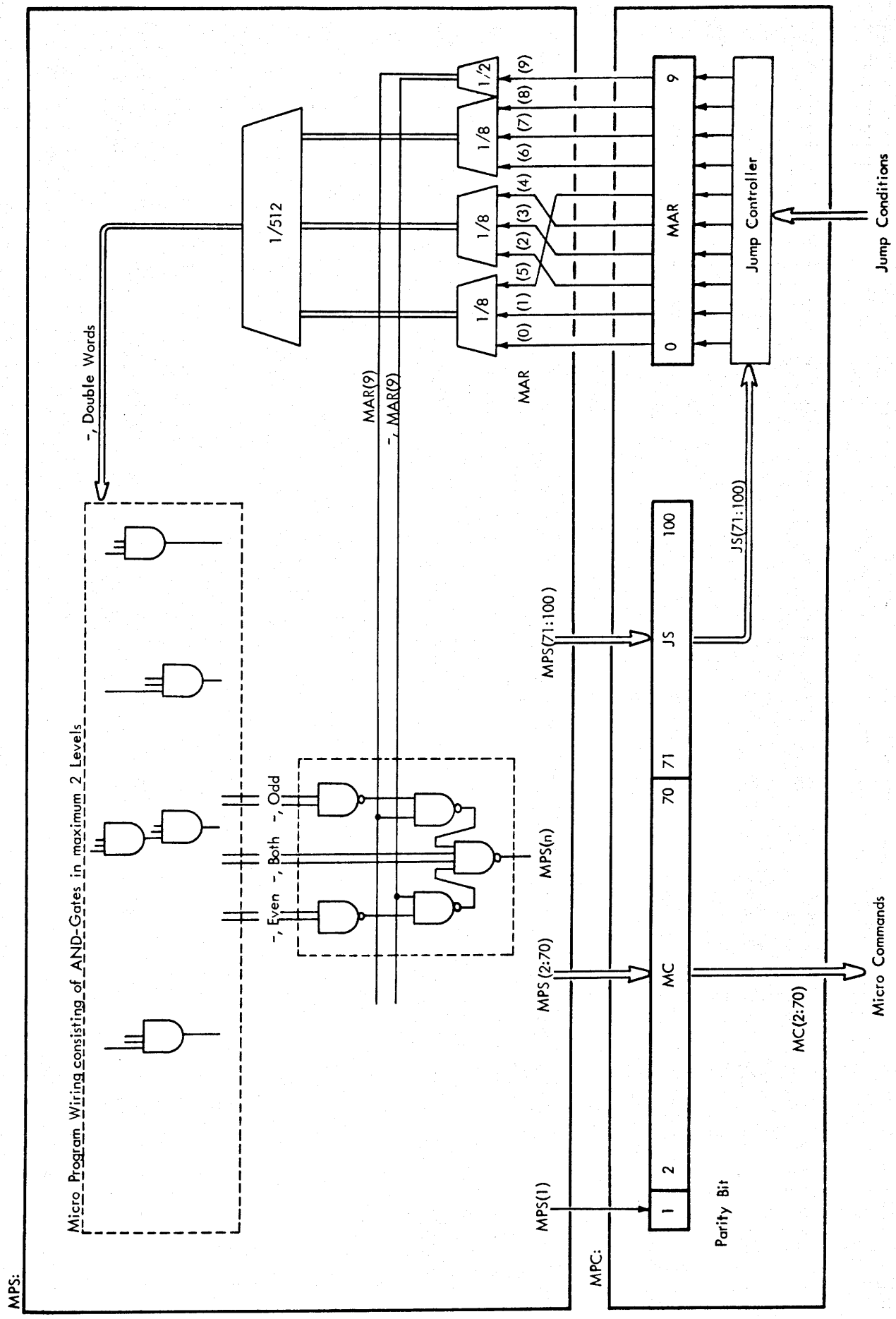


Fig. 1 Chematic Diagram of the Control Unit

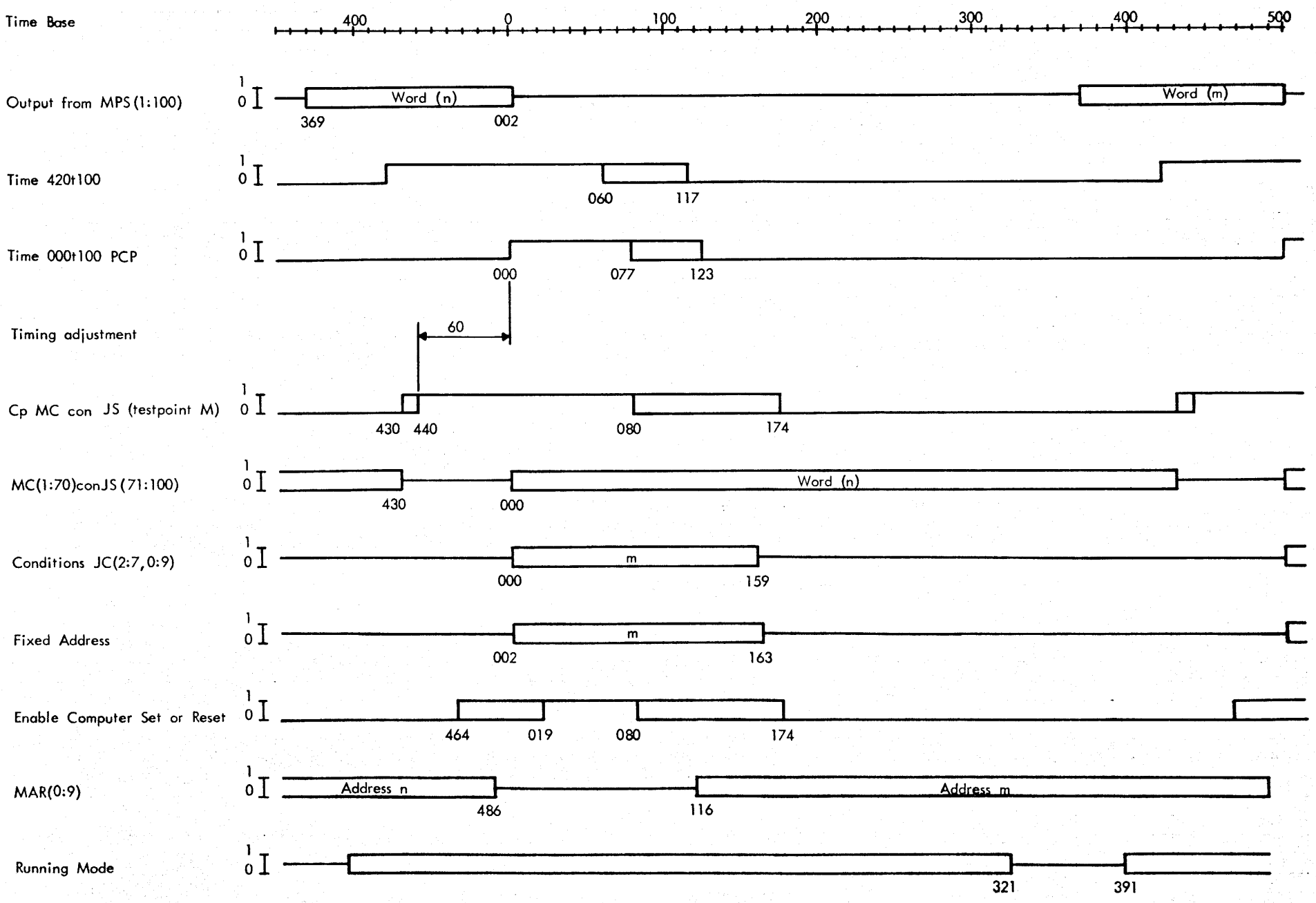


Fig. 2 Timing Diagram for MAR(0:9)

Fig. 3 Jump Conditions

JS(n+2) JS(n+1) JS(n)	MAR(0)	MAR(1)	MAR(2)	MAR(3)	MAR(4)	MAR(5)	MAR(6)	MAR(7)	MAR(8)	MAR(9)
	n=71 JS(71:73)	n=74 JS(74:76)	n=77 JS(77:79)	n=80 JS(80:82)	n=83 JS(83:85)	n=86 JS(86:88)	n=89 JS(89:91)	n=92 JS(92:94)	n=95 JS(95:97)	n=98 JS(98:100)
0 0 0	JC(0,0) 0	JC(0,1) 0	JC(0,2) 0	JC(0,3) 0	JC(0,4) 0	JC(0,5) 0	JC(0,6) 0	JC(0,7) 0	JC(0,8) 0	JC(0,9) 0
0 0 1	JC(1,0) 1	JC(1,1) 1	JC(1,2) 1	JC(1,3) 1	JC(1,4) 1	JC(1,5) 1	JC(1,6) 1	JC(1,7) 1	JC(1,8) 1	JC(1,9) 1
0 1 0	JC(2,0) SB<-65	JC(2,1) SB>64	JC(2,2) not used	JC(2,3) FR(2) & -, Modif	JC(2,4) FR(3) & -, Modif	JC(2,5) FR(0) & -, Modif	JC(2,6) FR(1) & -, Modif	JC(2,7) FR(4) & -, Modif	JC(2,8) FR(5) & -, Modif	JC(2,9) AF ≠ 0
0 1 1	JC(3,0) FR(8)	JC(3,1) FR(9)	JC(3,2) not used	JC(3,3) BE(11)	JC(3,4) FDsub	JC(3,5) FR(0)	JC(3,6) AR ≠ 0	JC(3,7) SC(11)	JC(3,8) SC ≠ 0	JC(3,9) FR(10) ! FR(11)
1 0 0	JC(4,0) AR(-1) = AR(0)	JC(4,1) AR(1) = AR(2)	JC(4,2) AR(0) = AR(1)	JC(4,3) FR(2)	JC(4,4) FR(3)	JC(4,5) Itr	JC(4,6) FR(1)	JC(4,7) FR(4)	JC(4,8) FR(5)	JC(4,9) Accept
1 0 1	JC(5,0) Carry(0)	JC(5,1) BR(1) = BR(2)	JC(5,2) not used	JC(5,3) BE(10)	JC(5,4) EX(22, 23) ≠ 0	JC(5,5) HA(23)	JC(5,6) BR(23)	JC(5,7) SB(0)	JC(5,8) SB ≠ 0	JC(5,9) BE(0)
1 1 0	JC(6,0) not used	JC(6,1) not used	JC(6,2) FR(6) ! FR(7)	JC(6,3) not used	JC(6,4) not used	JC(6,5) AR(-1)	JC(6,6) BR(22)	JC(6,7) MMode	JC(6,8) MMode ! -, PROTECT	JC(6,9) Round
1 1 1	JC(7,0) not used	JC(7,1) Start	JC(7,2) Autoload	JC(7,3) not used	JC(7,4) not used	JC(7,5) Main Power Key ON & -, Reset	JC(7,6) SC > -38 & SC < 38	JC(7,7) not used	JC(7,8) not used	JC(7,9) not used

cess to the NAND-gates when MAR(9) is logical zero, and all the one-bits in odd address words have access to the NAND-gates when MAR(9) is logical one.

The fact that many of the one-bits in the stored microprogram are stored both on a word address 2N and on the following word address 2N+1 has implied that the 2-way multiplex can be made as a 3-way multiplex, where the extra way in serves to let all one-bits stored both in the even and in the following odd words have access to the NAND-gate independently of the value of MAR(9).

In short, the one-bits in the microprogram are stored as >>even-bits<<, >>odd-bits<<, or >>both-bits<<. Zero-bits have no effect at all.

Each of the input ways to the descripted NAND-gates includes 2 terminals, a and b.

By using 16-input AND-gates in two levels, it is possible to store up to 512 >>even-bits<<, 512 >odd-bits<<, and 512 >>both-bits<< on one bit position in the microprogram.

2. EVALUATION OF NEXT MICRO ADDRESS

(MPC 002:011)	MAR
(MPC 019:021)	JS
(MPC 027)	Control of MAR

The Micro Address Register (MAR) contains the address of the selected word in MPS. This address can, in principle, be controlled in three different ways: computer controlled, manual controlled, or power controlled.

2.1. MAR Computer Controlled

In this mode of operation the address of the next step in the microprogram will depend on JS and the jump conditions (JC). JS is divided into 10 groups, each of 3 bits, in such a way that the value of group p controls the p'th bit in MAR. The relation between MAR, JS, and JC is seen in fig. 3. For example, MAR(1) equals JC(4,1) provided JS(74:76)= 4.

Moreover the address 1023 (x31y31) has an exceptional position, because it can be generated explecitly in spite of the contents of JS. This address is controlled by means of the signal Fixed Address as explained in the chapter about the Store Controller.

The computer controlled address calculation is described in details in the following sequence:

```
sequence Evaluate Comp Address;
begin
  Boolean array JC(0:7,0:9), a(0:9); comment a is a temporary variable;
  integer n,p;
  for p:= 0 step 1 until 9 do
```

```
  begin
    n:= 71+3*p;
    a[p]:=   JS(n:n+2)= 0 ∧ JC(0,p)
           ∨ JS(n:n+2)= 1 ∧ JC(1,p)
           ∨ JS(n:n+2)= 2 ∧ JC(2,p)
           ∨ JS(n:n+2)= 3 ∧ JC(3,p)
           ∨ JS(n:n+2)= 4 ∧ JC(4,p)
           ∨ JS(n:n+2)= 5 ∧ JC(5,p)
           ∨ JS(n:n+2)= 6 ∧ JC(6,p)
           ∨ JS(n:n+2)= 7 ∧ JC(7,p)
           ∨ Fixed Address
  end;
  begin
  comb net Comp Address (0:9)=
    a(0)cona(1)cona(2)cona(3)cona(4)cona(5)cona(6)cona(7)cona(8)cona(9)
  end;
end Evaluate Comp Address;
```

2.2. MAR Manual Controlled

In Technical Mode it is possible to change the control of MAR from computer controlled to manual controlled by depressing the pushbutton MAR MANUAL CONTROLLED. Hereafter MAR can be set manually by the aid of 20 buttons mounted on the ROS DISPLAY field.

```
sequence Evaluate Manual Address;
begin
  register
    MANUAL SET Make(0:9), MANUAL SET Break(0:9),
    MANUAL RESET Make(0:9), MANUAL RESET Break(0:9);
  comb net
    MANUAL SET(0:9) = MANUAL SET Make(0:9) ∧ MANUAL RESET Break(0:9),
    MANUAL RESET(0:9) = MANUAL RESET Make(0:9) ∧ MANUAL SET Break(0:9);
  comment These declarations define the 20 outputs from the push-buttons.
    It follows that SET and RESET cannot both be 1;
  if Manual Mode ∧ Stopped Mode ∧ -,GiMARf992 ∧ -,GiMARf31 then
    MAR(0:9):= MANUAL SET(0:9) ∨ -,MANUAL RESET(0:9)
end Evaluate Manual Address;
```

2.3. MAR Power Controlled

The two micro addresses 992 (x31y0) and 31 (x0y31) can at any time be read into MAR, no matter how MAR is otherwise controlled.

MAR is set to 992 during the periods of power shut-down and start-up under supervision of the signal -,GiMARf992. See Section 5.

The facility of setting MAR to 31 is not employed.

A summing up of the abovementioned three categories of addresses gives the following total expression for the next address:

```
if -,Power OK then MAR:= 992
  else
  begin
    if Manual Mode ∧ Stopped Mode then MAR:= Manual Address;
    if Manual Mode ∧ Running Mode then MAR:= MAR;
    if Computer Mode ∧ Stopped Mode then MAR:= MAR;
    if Computer Mode ∧ Running Mode then MAR:= Comp Address
  end;
```

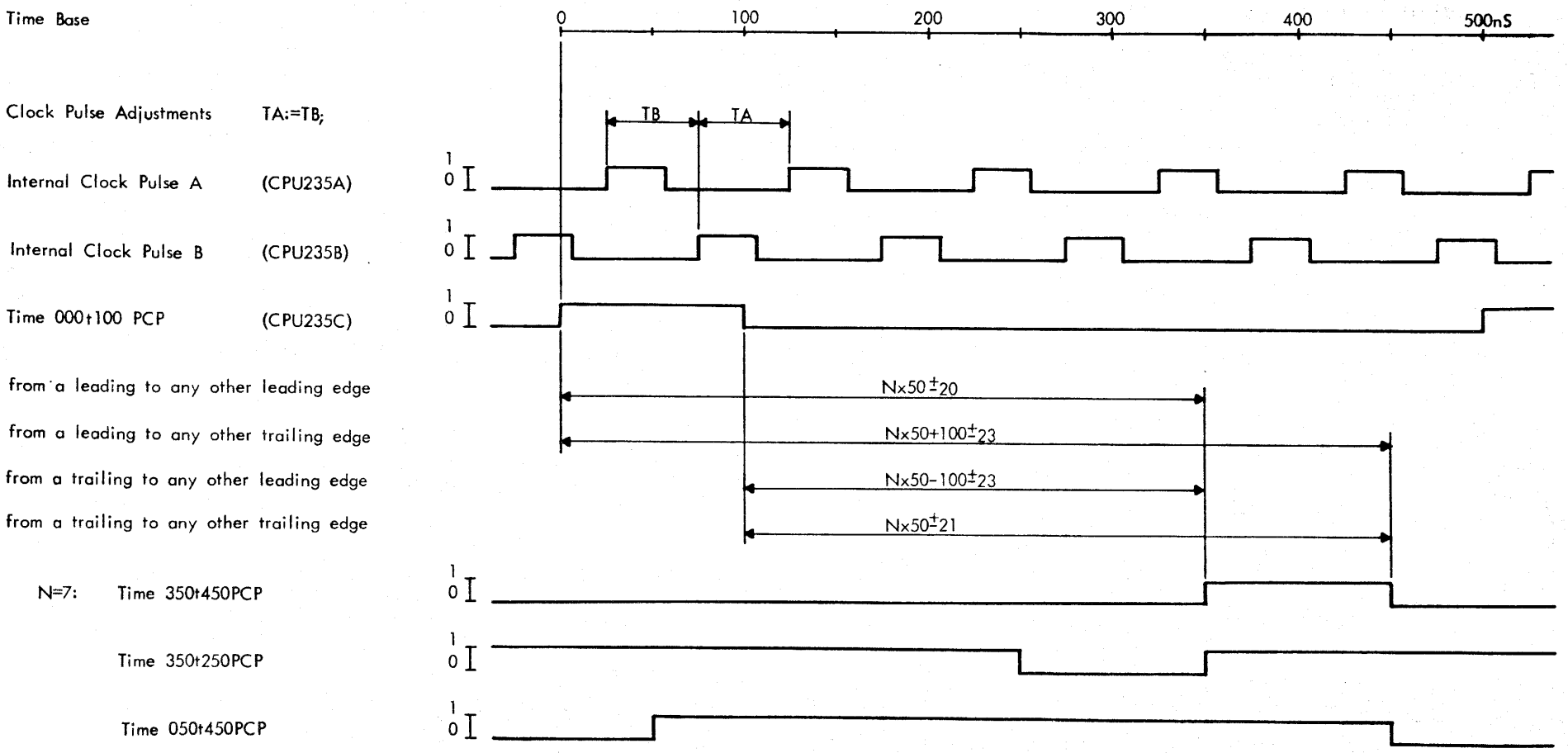


Fig. 4 Timing Diagram for Primary Clock Pulses

3. JUMP SELECTOR AND MICRO COMMAND REGISTER

(MPC 019:21) JS register
(MPC 012:018) MC register
(MPC 026)

The output from the selected word in MPS is normally read into MC and JS by means of the strobe pulses CpMC and CpJS.

Jump Selector Register.
The contents of JS changes every 500 nanoseconds in accordance with the selected word in MPS.

```
sequence JUMP SELECTORS;
begin
Time 0;
    JS:= MPS(MAR)(71:100);
    wait 500; goto Time 0
end JUMP SELECTORS;
```

Micro Command Register.
The contents of MC change normally every 500 nanoseconds in accordance with the selected word in MPS. In the case where the microprogram is stopped, MC is cleared to zero. Since this is not permissible for some micro commands, MC(1:10) is only excited when the microprogram runs.

```
sequence MICRO COMMANDS;
begin
Time 0:
  MC:= if Running Mode then MPS(MAR)(1:70) else MC(1:10)con0;
  wait 350;
Time 350:
  comment Running Mode can only change its value at time 350;
  wait 150; goto Time 0
end MICRO COMMANDS;
```

4. CLOCK PULSE TIMINGS

4.1 Primary Clock Pulse Generator

(MPC 001)

It has been mentioned that the repetition rate for the microprogram store is 500 nanoseconds, which implies that all the micro operations specified in one microstep are active in this period. Certain of the logical operations, derived from the micro operations, must even within this time interval follow a strictly timed format in order for the computer to execute its instructions. The period of 500 nanoseconds is therefore subdivided into 10 intervals delayed 50 nanoseconds with respect to each other as seen from Fig. 4. The circuit which accomplishes this is the primary clock pulse generator (PCP).

The PCP consists, in principle, of a 10-bit shift register with feed-back to the first bit only, while the remaining bits are excited by the contents of the previous bit. Registers with this property are known in the literature as feed-back shift registers. The feed-back to the first bit is determined by two constraints:

1) The contents of PCP equal 001100000 (or any of the other nine bitpatterns derived by means of cyclic shifts) after a maximum of nine clock periods no matter the original contents of PCP, i.e. PCP is self-generating and error correcting.
2) PCP operates as a cyclic register after the generating of the bitpattern mentioned in 1).

A more precise definition of PCP is:

```
sequence PRIMARY CLOCK PULSES;
begin register PCP(0:9);
SHIFT:
  PCP(0):= PCP(0:7)= 0 ∨ PCP(0) ∧ -,PCP(1);
  PCP(1:9):= PCP(0:8);
  wait 50; goto SHIFT
end PRIMARY CLOCK PULSES;
```

Thus it follows that PCP(n) will assume the logical value 1 in 100 nanoseconds and 0 in 400 nanoseconds. The reverse is naturally the case for the complemented output -,PCP(n).

PCP is in the actual design constructed of R-S elements, where R and S are excited from a two-phase 10 Mc oscillator.

All output signals from the Primary Clock Pulse generator have descriptors indicating the time interval in which they are logical 1. The time descriptors indicate an ideal timing scale which cannot be realized, but may be defined as a timing scale located exactly in the middle of two timing scales, one of them based upon the very fastest of the fast output pulses, and the other based upon the slowest of the slow output pulses. Compared with this ideal timing scale, the tolerances of all PCP signals are within $\pm$12 nanoseconds. Due to the fact that this ideal timing scale is a teoretical one, we have to define a time reference that can be measured and used as the basic time reference for the central processor.

As explained, all timing signals in the entire computer are derived from the PCP, which makes it desirable to use one of its outputs as a time reference, therefore the time 0 is defined to be the time where PCP(0) changes from 0 to 1, or more specific when the test point for PCP(0) passes the 1.5 V level, which is the threshold voltage of Texas Series 74. PCP(0) is identified as Time000t100PCP (the interval in which it is 1) and -,PCP(0) as Time100t000PCP. 000 is used instead of 500 because indication of time is modul 500.

As a consequence of using one of the PCP pulses as timing reference, the tolerances of all the other pulses must be expressed by referring to the selected PCP pulse, and therefore the tolerances of all other PCP pulses have to incorporate the tolerances of that particulary PCP pulse that is defined to indicate the timing reference. See fig. 4.

The correct tolerances of all the output signals from PCP can be obtained by adjusting the length and the phase of the output pulses from the 10 Mc oscillator (CC402). The first and preliminary adjustment of the 10 Mc oscillator

will give two 50-nanosecond output pulses with a phase shift of approximately 180 degrees between their leading edges. Then, the exact phase adjustment is carried out in the following way:

On circuit BF402 the internal two phase clock signals are accessible on testpoints A and B. The time difference TB from the leading edge of the clock pulse on testpoint A to the leading edge of the clock pulse on testpoint B must be exactly equal to the time difference TA from the leading edge of the clock pulse on testpoint B to the leading edge of the clock pulse on testpoint A. The correct time difference is 50 nanoseconds and can be obtained by adjusting the phase-potentiometer on circuit CC402. (On the assumption that the two output pulses from circuit CC402 are still about 50 nanoseconds, the PCP adjustment is completed, otherwise the two output pulses must be re-adjusted and then the exact phase adjustment must be repeated).

4.2. Timing Adjustment of MC and JS

(MPC 026, 012:022)

Normally, the proper timing of register MC and JS can be obtained by adjusting the leading edge of signal Time420t100.

The signals CpMC and CpJS are adjusted so that all bits of MC and JS are ready at time 500 (= 000). This time is, by definition, when the signal Time000t100PCP changes from 0 to 1. Since the maximum delay in a J-K element is 50 nanoseconds, it implies that the clock pulse, having the greatest delay, arrives not later than the time 450. As temperature and voltage variations may affect the delay, the adjustment should be carried out under normal running condition and the time of arrival of the clock pulse having the greatest delay should therefore be set to 440.

In case of replacement of circuit card involved in the abovementioned logic diagrams, it is necessary to perform a complete adjustment procedure: With the computer in Running Mode, the time difference is measured between the trailing edges of the signals -,CpMC(1:70) and -,CpJS(71:100) on (CPU106E) and (CPU106D). If the time difference is greater than 3 nanoseconds, the

12AC401 circuit card in (CPU106) must be replaced by another circuit card having closer tolerances. Then all the 10 signals, CpMC, and CpJS, measured on testpoint M on the BG403 are examined. If the time difference is greater than 10 nanoseconds between the fastest and the latest of the leading edges, the most outstanding circuit cards must be replaced until the desired close tolerances are obtained. Hereafter the potentiometer A on the delay circuit in POS CPU037 is adjusted so that the leading edge of the latest CpMC, or CpJS, appears exactly at time 440.

Hereby the time adjustment of MC(1:70) and JS(71:100) is finished, and the contents of the registers can now be guaranteed from time 000 to 430.

5. POWER START-UP AND SHUT-DOWN

To be defined later as the circuits for Start/Stop are not made out yet.

6. CONTROL MODES

For maintenance purposes, we have designed the Control Unit to operate in different modes. These modes can be controlled from either the OPERATOR CONTROL PANEL, OCP, or the TECHNICAL CONTROL PANEL, TCP, as described in an introductory manner in the section DESCRIPTION OF INDICATORS AND CONTROL SWITCHES. We shall now elaborate this description and, at the same time, examine the hardware realization. Before doing so it is worth noticing the different modes and their interdependence.

Operator Mode
Normal Mode = -,Technical Mode
Running Mode = -,Stopped Mode
Computer Mode = -,Manual Mode

6.1. Reset, Start, and Autoload

(MPC 033:034)

Reset, Start, and Autoload are three bistable elements controlled by the pushbuttons of the same name and by the OPERATOR CONTROL key.

Reset enters the jump condition

JC(7,5) = Main Power Key ON $\wedge$ -,Reset $\wedge$ Core Store Power OK

and JC(7,5) equals 0 as long as the pushbutton is pressed. The jump condition is interrogated once in every instruction cycle and if it becomes 0, the microprogram will execute the reset program and wait until START or AUTOLOAD is activated.

A 500-nanosecond Start signal equivalent to JC(7,1) is generated once every time we press the button START. The reason for this short signal is that it prevents a restart if the microprogram should once more go to the reset situation. This could actually happen if an autoload instruction (aw) is used in the beginning of the start program. The start signal is implemented by means of an auxiliary bistable called Enable Start Autoload as seen from the program description below.

```
          begin
AGAIN:      wait until START Break ∧ AUTOLOAD Break;
            Enable Start Autoload:= 1;
            wait until START Make;
            wait until Time 320; wait 30;
Time 350:   Start:= Enable Start Autoload ∧ START Make; wait 30;
Time 380:   Enable Start Autoload:= 0; wait 470;
Time 850:   Start:= 0;
            goto AGAIN
          end
```

A 500-nanosecond Autoload signal equivalent to JC(7,2) is generated once every time we press the button AUTOLOAD in exactly the same way as the Start signal.

6.2. Running Mode

(MPC 029:030)

As already seen from the previous description, Running Mode is one of the most essential variables in the Control Unit. The characteristic of Running Mode is that in this mode the computer executes the microprogram, whereas in Stopped Mode (= -,Running Mode) no new micro commands are executed and no new micro address is strobed into MAR. Notice that MC(1:10) may be different from 0 in Stopped Mode. In Stopped Mode, however, we can manually set up a new micro address or alter the contents of the registers in the arithmetic unit.

The Running Mode element operates as follows:

A. Running Mode is set to 0 if

- (A1) the button MAR MANUAL CONTROLLED is activated.
- (A2) the button MAR COMPUTER CONTROLLED is activated.
- (A3) a parity error in the microprogram store occurs.
- (A4) a parity error in the core store occurs.

B. Running Mode is set to 1 if

(B1) the mode selector key MODE SELECTOR NORMAL/TECHNICAL is switched to the NORMAL position. This signifies that the control unit operates in Normal Mode.

(B2) the button CONTINUE is activated in technical mode. This is achieved by a 500-nanosecond signal on the J side of Running Mode.

C. SINGLE MICRO INSTRUCTION.

When SINGLE MICRO INSTRUCTION is activated, the J side of the JK bistable Running Mode is set to 1 for a period of 500 nanoseconds, whereas the K side remains 1 as long as the pushbutton is activated. Therefore, in case Running Mode is initially 1, it is cleared to 0, and in the other case where Running Mode is initially 0 it will become 1 for a period of 500 nanoseconds and thus execute one microinstruction.

D. SINGLE INSTRUCTION.

As in case C, the J side is set to 1 for 500 nanoseconds but the K side is only 1 provided the pushbutton SINGLE INSTRUCTION is depressed and the microprogram executes an instruction where the jump condition JS(71:73)= 3. This jump condition is inserted once in every instruction cycle, and this makes it possible to execute the microprogram instruction by instruction.

The two cases C and D are spelled out in the program description to follow.

The 500-nanosecond signal on the J side of Running Mode is generated via two auxiliary variables called Running Start(1,2), which operate as a single shot generator (monostable) when one of the buttons CONTINUE, SINGLE MICRO INSTRUCTION, or SINGLE INSTRUCTION is activated.

```
begin
  if SINGLE MICRO INSTRUCTION then
    begin
        Single Micro Instruction:= 1; wait 200;
Time 50:  Running Start(1,2):= b10; wait 300;
```

```
Time 350: Running Mode:= if Running Start = 2 then -,Running Mode else 0;
            wait 200;
Time 550: Running Start(1,2):= 1conRunning Start(1); wait 300;
Time 850: Running Mode:= if Running Start = 2 then -,Running Mode else 0;
          comment Running Mode= 0;
    end;

  if SINGLE INSTRUCTION then
    begin
          Single Instruction:= 1; wait 200;
Time  50: Running Start(1,2):= b10; wait 300;
Time 350: if (Running Start= 2)∧(JS(71:73)=3)
                                then Running Mode:= -,Running Mode;
          if (Running Start= 2)∧ -,(JS(71:73)=3) then Running Mode:= 1;
          if -,(Running Start= 2)∧(JS(71:73)=3) then Running Mode:= 0;
          comment JS(71:73) equals 3 once in every instruction cycle;
          wait 200;
Time 550: Running Start(1,2):= 1conRunning Start(1); wait 300;
Time 850: if (Running Start= 2)∧(JS(71:73)=3)
                                then Running Mode:= -,Running Mode;
          if (Running Start= 2)∧ -,(JS(71:73)=3) then Running Mode:= 1;
          if -,(Running Start= 2)∧(JS(71:73)=3) then Running Mode:= 0;
          wait 200; if Running Mode = 1 then goto Time 550;
          comment Running Mode:= 0;
    end
end
```

6.3. Core Store and MPS Parity Control

(MPC 032) Core Store Parity Control

(MPC 022:025,031) MPS Parity Control

Both the core store and the read-only store are under normal conditions checked for parity errors. By this means it is possible to detect an error as soon as it occurs and thereby diminish the time which is necessary for locat-

ing the faulty circuit. A parity failure stops the microprogram and the red indicator MACHINE ERROR will light. Two indicators, mounted on the TECHNICAL CONTROL PANEL, make it possible to distinguish between Core Store parity errors and Micro Program Parity errors.

Core Store Parity Control.
The core store is initiated by the micro commands Read Instruction, Read Data, and Read Split. On receipt of one of these commands, the store will read the data from the selected word into the combined register

SBconPROTECTconPARITY

no later than 1348 nanoseconds after the leading edge of the command pulse. The formation of the logical parity expression

oddSBconPROTECTconPARITY

lasts 225 nanoseconds, and the bistable element Core Store Parity Error is set in accordance with this result at time 1665 (= 165).

In the event of an error, a restart is only possible after the Core Store Parity Error flip-flop is cleared. This is done by switching to Technical Mode and depressing the pushbutton CORE STORE PARITY CONTROL OFF.

A special program is provided which cancels all parity errors in the entire store.

MPS Parity Control.
The microcommand store is subjected to parity control when the following two requirements are met:

1) MPS Parity Control = 1
2) Running Mode = 1

The first condition is obvious. The second condition arises from the fact that MC(11:70) is set to zero when the computer is not in Running Mode.

In the event of an error, a restart is only possible after the MPS Parity Control flip-flop is cleared. This is done by switching to Technical Mode and depressing the pushbutton MPS PARITY CONTROL OFF.

6.4. Control of ARU Display

(MPC 035)

The ARU DISPLAY field serves not only to display the contents of a register but also to alter this contents by the aid of manually operated pushbuttons. The ARU DISPLAY functions when the three signals, A, B, and C, are continuously generated from a 3-bit counter. The counter starts itself as soon as the microprogram stops (Running Mode = 0) provided the MODE SELECTOR key-switch is in TECHNICAL position and none of the pushbuttons on the TECHNICAL CONTROL field are activated. The counter stops either because the abovementioned key-switch is turned to NORMAL position or because one of the pushbuttons are depressed. Fig. 5 shows the A, B, and C, and their use in the arithmetic unit.

6.5. Normal Mode

(MPC 029)

The control unit operates nearly always in Normal Mode and the operator should only switch over to Technical Mode for maintenance purposes, and naturally in the event of errors. In this connection we would like to point out that although it is possible to debug a program by stepping through instruction by instruction, this approach is very time consuming, and we would instead advise the programmer to use the standard debugging programs.

The normal status of the main variables appears from the following statement.

```
if Normal Mode ∧ Power OK ∧ -,(Core Store Parity Error ∨ MPS Parity Error)
  then
    begin
      Core Store Parity Control:= ROS Parity Control:= 1;
      Running Mode:= 1;
      Single Micro Instruction:= Single Instruction:= 0;
      Running Start(1,2):= 0;
      Manual Mode:= 0;
      ARU Display Running:= 0
    end
```

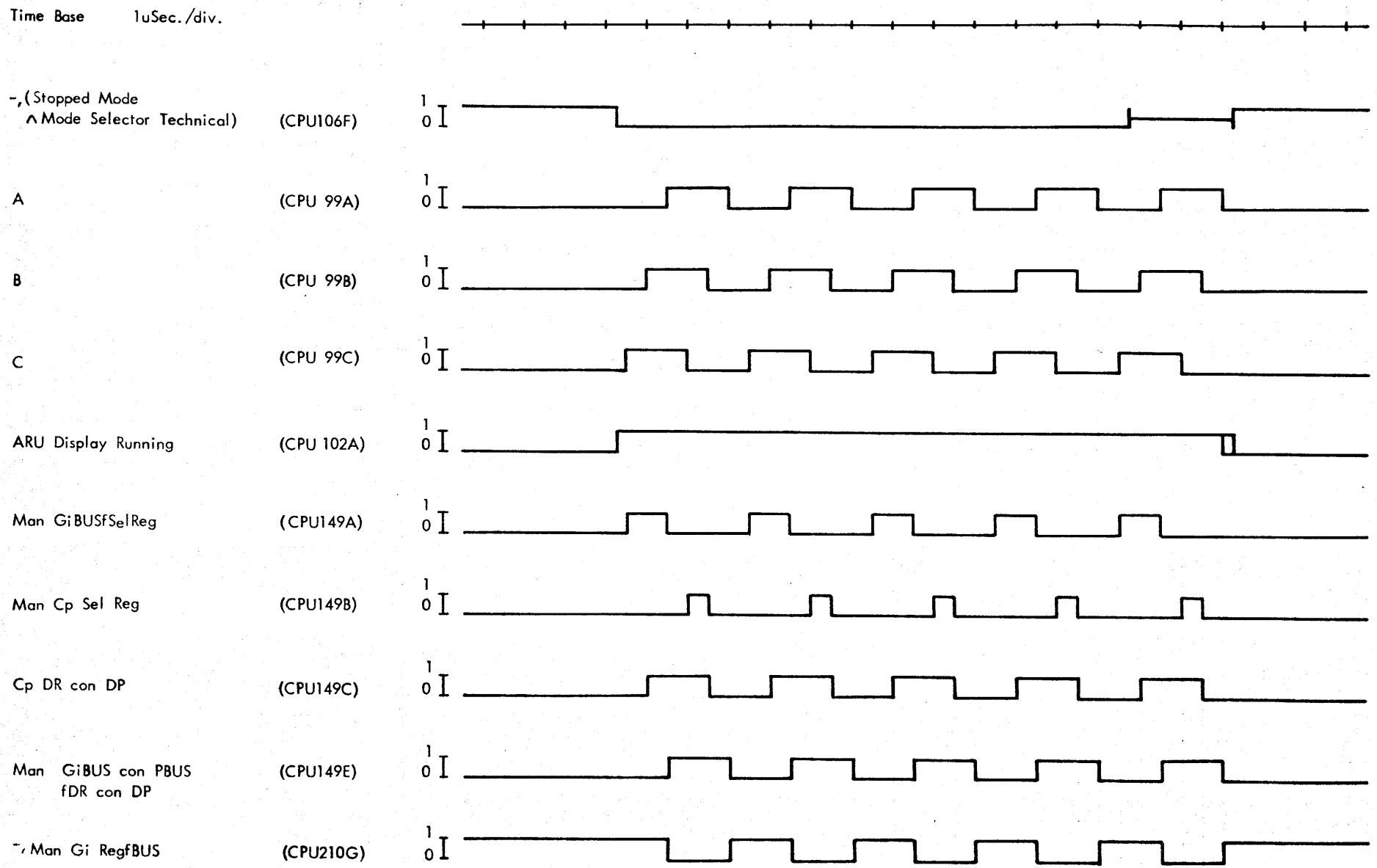


Fig. 5 Timing Diagram for ARU Display

7. PROGRAM DESCRIPTION OF THE CONTROL UNIT

(MPC 002:011)	MAR
(MPC 012:021)	MC(1:70), JS(71:100)
(MPC 027)	GiMARf992
(MPC 033:034)	Operator Mode, Reset, Start, Autoload, Enable Start Autoload
(MPC 028)	Manual Mode
(MPC 030)	Single Micro Instruction, Single Instruction, Running Start
(MPC 029)	Running Mode, Technical Mode
(MPC 032)	Core Store Parity Control, Core Store Parity Error
(MPC 022:025,031)	MPS Parity Control, MPS Parity Error, Enable Parity Control
(MPC 035)	ARU Display Running

The last section of this chapter is devoted to a program, which gives a thorough and concise description of how the Control Unit functions. The program is divided into subprograms in order to facilitate the understanding. The names of the subprograms are:

MICROPROGRAM STORE
SUPERVISION OF POWER
SUPERVISION OF START AND AUTOLOAD
CONTROL LOGIC
CORE STORE PARITY CONTROL
MPS PARITY CONTROL
ARU DISPLAY CONTROL
DISPLAY OF OPERATOR CONTROL PANEL
DISPLAY OF TECHNICAL CONTROL PANEL

This table shows in which subprograms a bistable element participates and at which time the value of the element may change. When bistables are controlled by asynchronous signals, they are denoted by A in this list.

	MICROPROGRAM STORE	SUPERVISION OF POWER	SUPERVISION OF START AND AUTOLOAD	CONTROL LOGIC	CORE STORE PARITY CONTROL	MPS PARITY CONTROL	ARU DISPLAY CONTROL
ARU Display Running							430 000
Autoload		A		350			
Core Store Parity Control					000		
Core Store Parity Error					030 165		
Enable Start Autoload			A	380			
JS(71:100)	000						
Manual Mode				350			
MAR	090	A					
GiMARf992	280	A					
MC(1:70)	000						
Operator Mode				350			
Reset				350 380			
MPS Parity	000						
MPS Parity Control						000	
MPS Parity Error						030 345	
Running Mode				350			
Running Start (1,2)				050			
Single Instruction				350			
Single Micro Instruction				350			
Start		A		350			
Technical Mode				350			

```
CONTROL UNIT:
begin
Boolean Power OK;
comment The signal Power OK is governed by the power control circuit;

register
  MAR(0:9),
  MC(1:70), JS(71:100),
  GiMARf992(0:0),
  Operator Mode(0:0),
  Reset(0:0), Start(0:0), Autoload(0:0), Enable Start Autoload(0:0),
  Manual Mode(0:0),
  Single Micro Instruction(0:0), Single Instruction(0:0),
                                        Running Start(1:2),
  Running Mode(0:0), Technical Mode(0:0),
  Core Store Parity Control(0:0), Core Store Parity Error(0:0),
  MPS Parity Control(0:0), MPS Parity Error(0:0),
  ARU Display Running(0:0);

comb net
  Stopped Mode(0:0) = -,Running Mode,
  Normal Mode(0:0) = -,Technical Mode,
  Computer Mode(0:0) = -,Manual Mode;
comment The two variables on each side of the equality sign represent the
  true and the complemented output from a J-K element;

comment The variables, in the next three declarations, serve to calculate
  Comp Address. JC are the jump conditions, a are auxiliary variables,
  and Fixed Address is a signal generated from the Store Controller;
Boolean array JC(0:7,0:9), a(0:9);
Boolean Fixed Address;
comb net
  Comb Address(0:9) = a(0)cona(1)cona(2)cona(3)cona(4)cona(5)cona(6)con
                      a(7)cona(8)cona(9);
```

```
comment The following declaration defines the pushbuttons and the keys on the
  OPERATOR CONTROL PANEL and the TECHNICAL CONTROL PANEL.
register
  OPERATOR CONTROL ON Make(0:0), OPERATOR CONTROL ON Break(0:0),
  RESET Make(0:0), RESET Break(0:0),
  START Make(0:0), START Break(0:0),
  AUTOLOAD Make(0:0), AUTOLOAD Break(0:0),

  MODE SELECTOR NORMAL Make(0:0), MODE SELECTOR TECHNICAL Make(0:0),
  CONTINUE Make(0:0), CONTINUE Break(0:0),
  SINGLE INSTRUCTION Make(0:0), SINGLE INSTRUCTION Break(0:0),
  SINGLE MICRO INSTRUCTION Make(0:0), SINGLE MICRO INSTRUCTION Break(0:0),

  MAR COMPUTER CONTROLLED Make(0:0), MAR COMPUTER CONTROLLED Break(0:0),
  MAR MANUAL CONTROLLED Make(0:0), MAR MANUAL CONTROLLED Break(0:0),

  MANUAL SET Make(0:9), MANUAL SET Break(0:9),
  MANUAL RESET Make(0:9), MANUAL RESET Break(0:9),

  CORE STORE PARITY CONTROL ON Make(0:0),
                              CORE STORE PARITY CONTROL ON Break(0:0),
  CORE STORE PARITY CONTROL OFF Make(0:0),
                             CORE STORE PARITY CONTROL OFF Break(0:0),
  MPS PARITY CONTROL ON Make(0:0), MPS PARITY CONTROL ON Break(0:0),
  MPS PARITY CONTROL OFF Make(0:0), MPS PARITY CONTROL OFF Break(0:0),

  TECHNICAL CONTROL Field Connected Make(0:0);

comment The declaration shows the switching logic, that is the logic imple-
  mented by means of internal connections between the Make and the Break parts
  of the switches;
comb net
  MANUAL SET(0:9) = MANUAL SET Make(0:9) ∧ MANUAL RESET Break(0:9),
  MANUAL RESET(0:9) = MANUAL RESET Make(0:9) ∧ MANUAL SET Break(0:9),
```

```
MAR COMPUTER CONTROLLED(0:0) = MAR COMPUTER CONTROLLED Make,
MAR MANUAL CONTROLLED(0:0) = MAR MANUAL CONTROLLED Make
                           ∧ MAR COMPUTER CONTROLLED Break,

SINGLE MICRO INSTRUCTION(0:0) = SINGLE MICRO INSTRUCTION Make
                              ∧ MAR COMPUTER CONTROLLED Break
                              ∧ MAR MANUAL CONTROLLED Break
SINGLE INSTRUCTION(0:0) = SINGLE INSTRUCTION Make
                        ∧ MAR COMPUTER CONTROLLED Break
                        ∧ MAR MANUAL CONTROLLED Break
                        ∧ SINGLE MICRO INSTRUCTION Break,
CONTINUE(0:0) = CONTINUE Make
              ∧ MAR COMPUTER CONTROLLED Break
              ∧ MAR MANUAL CONTROLLED Break
              ∧ SINGLE MICRO INSTRUCTION Break
              ∧ SINGLE INSTRUCTION Break,
ALL BUTTONS BREAK(0:0) = MAR COMPUTER CONTROLLED Break
                       ∧ MAR MANUAL CONTROLLED Break
                       ∧ SINGLE MICRO INSTRUCTION Break
                       ∧ SINGLE INSTRUCTION Break
                       ∧ CONTINUE Break,
MODE SELECTOR TECHNICAL(0:0) = MODE SELECTOR TECHNICAL Make
                             ∧ MAR COMPUTER CONTROLLED Break
                             ∧ MAR MANUAL CONTROLLED Break
                             ∧ SINGLE MICRO INSTRUCTION Break
                             ∧ SINGLE INSTRUCTION Break
                             ∧ CONTINUE Break,
MODE SELECTOR NORMAL(0:0) = MODE SELECTOR NORMAL Make,

CORE STORE CONTROL ON(0:0) = CORE STORE CONTROL ON Make
                           ∧ CORE STORE CONTROL OFF Break,
CORE STORE CONTROL OFF(0:0) = CORE STORE CONTROL OFF Make
                            ∧ CORE STORE CONTROL ON Break,
MPS PARITY CONTROL ON(0:0) = MPS PARITY CONTROL ON Make
                           ∧ MPS PARITY CONTROL OFF Break;
MPS PARITY CONTROL OFF(0:0) = MPS PARITY CONTROL OFF Make
                            ∧ MPS PARITY CONTROL ON Break;
```

```
sequence MICROPROGRAM STORE;
comment The function of the microprogram store appears from this sequence;
begin
  register array MPS(0:1023)(1:100);
  comment The program stored in MPS is defined in Chapter 8;
Time 0:
  MCconJS:= if Running Mode then MPS(MAR)
              else MPS(MAR)(1:10)con60ext0conMPS(MAR)(71:100);
  begin comment Evaluate Comp Address;
    integer n,p; wait 30;
Time 30:
    for p:= 0 step 1 until 9 do
    begin
      n:= 71+3×p;
      a(p):= JS(n:n+2)= 0 ∧ JC(0,p) ∨ JS(n:n+2)= 1 ∧ JC(1,p)
           ∨ JS(n:n+2)= 2 ∧ JC(2,p) ∨ JS(n:n+2)= 3 ∧ JC(3,p)
           ∨ JS(n:n+2)= 4 ∧ JC(4,p) ∨ JS(n:n+2)= 5 ∧ JC(5,p)
           ∨ JS(n:n+2)= 6 ∧ JC(6,p) ∨ JS(n:n+2)= 7 ∧ JC(7,p)
           ∨ Fixed Address
    end;
    comment Comp Address ≔ a(0)cona(1)con ... cona(9);
  end; wait 60;
Time 90:
  if Computer Mode ∧ Running Mode ∧ -,GiMARf992 then ↯ MAR:= Comp Address;
  wait 190;
Time 280:
  if Power OK then ↯ GiMARf992:= 0;
  comment Running Mode may change at time 350; wait 170;
Time 450:
  if Manual Mode ∧ -,Running Mode ∧ -,GiMARf992 then
  ↯ MAR:= MANUAL SET(0:9)
        ∨ -,MANUAL RESET(0:9) ∧ MAR;
end MICROPROGRAM STORE;
```

```
sequence SUPERVISION OF POWER;
comment This sequence controls the asynchronous signal Power OK;
begin
AGAIN:
  wait until -,Power OK;
  Core Store Parity Error:= MPS Parity Error:= 0;
  Start:= Autoload:= 0;
  MAR992:= 1;
  if MAR992 then MAR:= 992;
  wait until Power OK; goto AGAIN
end SUPERVISION OF POWER;
```

```
sequence SUPERVISION OF START AND AUTOLOAD;
comment This sequence controls the switches START and AUTOLOAD;
begin
AGAIN:
  wait until START Break ∧ AUTOLOAD Break;
  Enable Start Autoload:= 1;
  wait until -,(START Break ∧ AUTOLOAD Break); goto AGAIN
end SUPERVISION OF START AND AUTOLOAD;
```

```
sequence CONTROL LOGIC;
comment This sequence describes the different modes of operation. All
  operations are synchronous. The asynchronous operations are found in the
  sequences SUPERVISION OF POWER and SUPERVISION OF START AND AUTOLOAD;
begin
  register Running Start(1:2);
  comment The J-side of the J-K bistable Running Mode is 1 if
    Running Start = 2;
  wait 50;
Time 50:
‡ Running Start(1):= (Technical Mode ∧ -,ARU Display Running
              ∧ (Single Micro Instruction ∨ Single Instruction ∨ CONTINUE))
                                                    ∧ -,Running Start(1)
            ∨ -,(ALL BUTTONS BREAK ∨ -,Technical Mode) ∧ Running Start(1)

‡ Running Start(2):= Running Start(1);
  wait 300;
Time 350:
‡ Operator Mode:= (OPERATOR CONTROL ON Make ∧ Normal Mode) ∧ -,Operator Mode
            ∨ -,(OPERATOR CONTROL OFF Make ∨ -,Normal Mode) ∧ Operator Mode;

  if Operator Mode then
  ‡ Reset:= (RESET Make ∧ Operator Mode) ∧ -,Reset
              ∨ -,(RESET Break) ∧ Reset
    else ‡ Reset:= 0;

if Power OK then
‡ Start:= (START Make ∧ Operator Mode ∧ Enable Start Autoload) ∧ -,Start
            ∨ -,(-,Enable Start Autoload) ∧ Start;

if Power OK then
‡ Autoload:= (AUTOLOAD Make ∧ Operator Mode ∧ Enable Start Autoload)
                                                    ∧ -,Autoload
            ∨ -,(-,Enable Start Autoload) ∧ Autoload;
```

```
Time 350:
‡ Manual Mode:= (MAR MANUAL CONTROLLED ∧ Technical Mode ∧ -,Running Mode)
                                                    ∧ -,Manual Mode
          ∨ -,(MAR COMPUTER CONTROLLED ∧ -,Running Mode ∨ -,Power OK
                                        ∨ -,Technical Mode) ∧ Manual Mode;

‡ Single Micro Instruction:= (SINGLE MICRO INSTRUCTION ∧ Technical Mode)
                                                ∧ -,Single Micro Instruction
          ∨ -,(ALL BUTTONS BREAK ∨ -,Technical Mode)
                                ∧ Single Micro Instruction;

‡ Single Instruction:= (SINGLE INSTRUCTION ∧ Technical Mode)
                                            ∧ -,Single Instruction
          ∨ -,(-,Running Mode ∧ (-,Technical Mode ∨ ALL BUTTONS BREAK)
             ∨ -,Technical Mode ∨ Single Micro Instruction ∨ CONTINUE)
                                                ∧ Single Instruction;

‡ Normal Mode:= (-,TECHNICAL CONTROL Field Connected ∨ MODE SELECTOR NORMAL)
                                                    ∧ -,Normal Mode
          ∨ -,(MODE SELECTOR TECHNICAL) ∧ Normal Mode;

‡ Running Mode:= (-,Core Store Parity Error ∧ -,MPS Parity Error
             ∧ (Normal Mode ∨ -,Power OK
             ∨ Running Start(1) ∧ -,Running Start(2))) ∧ -,Running Mode
          ∨ -,(Technical Mode
             ∧ (MAR MANUAL CONTROLLED ∨ MAR COMPUTER CONTROLLED)
             ∨ Core Store Parity Error
             ∨ MPS Parity Control ∧ Enable Parity Control
             ∧ -,oddMCconJSconMPS Parity
             ∨ Single Instruction ∧ -,JS(71) ∧ JS(72) ∧ JS(73)
             ∨ Single Micro Instruction) ∧ Running Mode
  wait 30;
Time 380:
  if -,Operator Mode then ‡ Reset:= 0;

  if (-,START Break ∨ -,AUTOLOAD Break) ∧ (Start ∨ Autoload) then
  ‡ Enable Start Autoload:= 0;
  wait 120; goto Time 0
end CONTROL LOGIC;
```

```
sequence CORE STORE PARITY CONTROL;
comment The sequence describes the necessary logic for checking the core
  store. TB(3) is a flip-flop in the Store Controller (STC). Addr ST = 1
  signifies that the address is a valid core store address. The selected
  core store word is read into the combined register SBconPROTECTconPARITY;
begin
  register SB(0:23), PROTECT(24:24), PARITY(25:25), TB(3:3), AddrST(0:0);
Time 0:
X Core Store Parity Control:= ((-,Technical Mode ∨ CORE STORE PARITY CONTROL ON)
                                        ∧ Power OK) ∧ -,Core Store Parity Control
          ∨ -,(Technical Mode ∧ CORE STORE PARITY CONTROL OFF ∨ -,Power OK)
                                                  ∧ Core Store Parity Control;
  wait 30;
Time 30:
  if Power OK then
  X Core Store Parity Error:= Core Store Parity Control
                              ∧ Core Store Parity Error;
  wait 135;
Time 165:
X Core Store Parity Error:= Core Store Parity Control
          ∧ -,oddSBconPROTECTconPARITY ∧ TB(3) ∧ Addr ST;
  wait 335; goto Time 0
end CORE STORE PARITY CONTROL;
```

```
sequence MPS PARITY CONTROL;
comment The sequence describes the necessary logic for checking the micro-
  program store;
begin
Time 0:
‡ MPS Parity Control:= ((-,Technical Mode ∨ MPS PARITY CONTROL ON)
                                    ∧ Power OK) ∧ -,MPS Parity Control
          ∨ -,(Technical Mode ∧ MPS PARITY CONTROL OFF ∨ -,Power OK)
                                                  ∧ MPS Parity Control;
  wait 30;
Time 30:
  if Power OK then
  ‡ MPS Parity Error:= MPS Parity Control ∧ MPS Parity Error;
  wait 315;
Time 345:
‡ MPS Parity Error:= MPS Parity Control ∧ even MConJS ∧ Running Mode
           ∨ MPS Parity Error;
  wait 155;
  goto Time 0;
end MPS PARITY CONTROL;

sequence ARU DISPLAY CONTROL;
comment The ARU DISPLAY functions when the three signals Enable Register
  Selection, Register To BUS, and BUS To Register are continuously gen-
  erated from the counter;
begin
  register A(0:0), B(0:0); comment Two elements of the counter;
  comb net
    ARU Display Start(0:0) = MODE SELECTOR TECHNICAL ∧ -,Running Mode,
    ARU Display Stop(0:0) = -,MODE SELECTOR TECHNICAL,

    Enable Register Selection(0:0) = A ∧ B ∧ ARU Display Running
                                    ∧ time 030t430,
    Register To BUS(0:0) = ARU Display Running,
    BUS To Register(0:0) = -,A ∧ B ∧ ARU Display Running ∧ time 030t430;
```

```
COUNTER:
  wait 430;
Time 430:
✱ A:= (B ∨ ARU Display Running) ∧ -,A;
✱ B:= AexorB;
✱ ARU Display Running:= (-,A ∧ -,B ∧ ARU Display Start)
                                  ∧ -,ARU Display Running
            ∨ -,(ARU Display Stop ∧ A ∧ B) ∧ ARU Display Running;
  wait 500; goto Time 430
end ARU DISPLAY CONTROL;

sequence DISPLAY OF OPERATOR CONTROL PANEL;
comment This sequence defines the indicators on the OPERATOR CONTROL PANEL.
  An indicator lamp lights when the corresponding comb net variable has the
  value 1;
begin
  Boolean Core Store Power OK, BLR Power OK, BLS Power OK, BLT Power OK;
  comment These Boolean variables equal 1 if the power supplies in the
    respective units function correctly;

  Boolean MR Control; integer n;
Microprogram stopped:
  MR Control:= 0;
  wait until Normal Mode ∧ JS(86:88) = 2 ∧ -,Core Store Parity Error
                                  ∧ -,MPS Parity Error;
Microprogram running:
  for n:= 0 step 500 until 40000 do
    begin
      comment The MICROPROGRAM RUNNING lamp lights 40 microseconds after
        the above Boolean expression is satisfied. This condition
        JS(86:88) = 2 appears once in every instruction cycle;
      MR Control:= 1; wait 500;
      if Normal Mode ∧ JS(86:88) = 2 ∧ -,Core Store Parity Error
                                  ∧ -,MPS Parity Error then
        goto Microprogram Running
    end
  goto Microprogram stopped;
```

```
begin
  comb net
    SYSTEM POWER(0:0) = Power OK ∧ Core Store Power OK ∧ BLR Power OK
                        ∧ BLS Power OK ∧ BLT Power OK,

    MICROPROGRAM RUNNING(0:0) = MR Control,
    MACHINE ERROR(0:0) = Core Store Parity Error ∨ MPS Parity Error,

    RESET(0:0) = Operator Mode ∧ Power OK,
    START(0:0) = Operator Mode ∧ Power OK,
    AUTOLOAD(0:0) = Operator Mode ∧ Power OK;
  end
end DISPLAY OF OPERATOR CONTROL PANEL;

sequence DISPLAY OF TECHNICAL CONTROL PANEL;
comment This sequence defines the indicator on the TECHNICAL CONTROL PANEL.
  An indicator lamp lights when the corresponding comb net variable has the
  value 1;
begin
  comb net
    MICRO COMMAND REGISTER(1:70) = MC(1:70),
    JUMP SELECTOR REGISTER(71:100) = JS(71:100),

    MICRO ADDRESS REGISTER(0:9) = MAR(0:9),
    NEXT ADDRESS(0:9) = Comp address(0:9),

    CORE STORE PARITY ERROR(0:0) = Core Store Parity Error,
    CORE STORE PARITY CONTROL(0:0) = Core Store Parity Control,
    CORE STORE POWER OK(0:0) = Core Store Power OK,

    MPS PARITY BIT(0:0) = MPS Parity,
    MPS PARITY ERROR(0:0) = MPS Parity Error,
    MPS PARITY CONTROL(0:0) = MPS Parity Control,

    RUNNING MODE(0:0) = Running Mode,
    NORMAL MODE(0:0) = Normal Mode,
    MANUAL MODE(0:0) = Manual Mode,
    ARU DISPLAY ON(0:0) = ARU Display Running;
end DISPLAY OF TECHNICAL CONTROL PANEL;
```

```
comment All quantities in the control unit are herewith declared;

start MICROPROGRAM STORE; start SUPERVISION OF POWER;
start SUPERVISION OF START AND AUTOLOAD; start CONTROL LOGIC;
start CORE STORE PARITY CONTROL; start MPS PARITY CONTROL;
start DISPLAY OF OPERATOR CONTROL PANEL;
comment The external controlled quantities, for example the pushbuttons, the
  signals Power OK and Fixed Address, and the microprogram store MPS, are not
  initiated in this program and further information about them must be ob-
  tained in their respective chapters;

end CONTROL UNIT;
```

A/S REGNECENTRALEN

SCANDINAVIAN INFORMATION PROCESSING SYSTEMS

HEADQUARTERS: FALKONER ALLÉ 1 . DK-2000 COPENHAGEN F . DENMARK
TELEPHONE: (01) 105366 . TELEX: 6282 RCHQ DK . CABLES: REGNECENTRALEN