Micro Programming

Herman Spanjersberg

Micro Programming

Herman Spanjersberg
h.a.spanjersberg@planet.nl

Abstract: In the 1970s a need arose to perform special arithmetic operations on minicomputers much more quickly than had been possible in the past. This paper tells the story of why micro programming was needed for special arithmetic operations on mini computers in the 1970s and how it was implemented. The paper tells how the laboratory in which the first experiment took place had a PDP-9 minicomputer from Digital Equipment Corporation and how the author, with several colleagues, after attending a course for the technical service of a PDP-9 given by a specialist from Digital Equipment, knew exactly which signals flew through the machine at any time. The paper describes how by having 'programmable' control memory they were able to make changes in the execution of instructions.

Keywords: Micro programming, minicomputer,

1. Introduction

In the late seventies of the last century there came a need for special arithmetic operations with a (mini)computer. One of the reasons was that researchers wanted to collect and adapt measurement data within a few microseconds: in some cases the result of the operations had to be fed back to the process to control the process or the character of the research project demanded a rapid insight in the converted data. For example there was a research project in an ophthalmologic laboratory where one needed an instant fast Fourier transform of the measured signals. With the normal processors this was impossible, or better formulated: it took too long time with normal 'general purpose' computers. Based on an article of Professor Wilkes[1] we started an experiment on a standard minicomputer with the theme *'microprogramming*'. To understand that technique at first a short description is given of processor[2] architecture.

2. CPU architecture

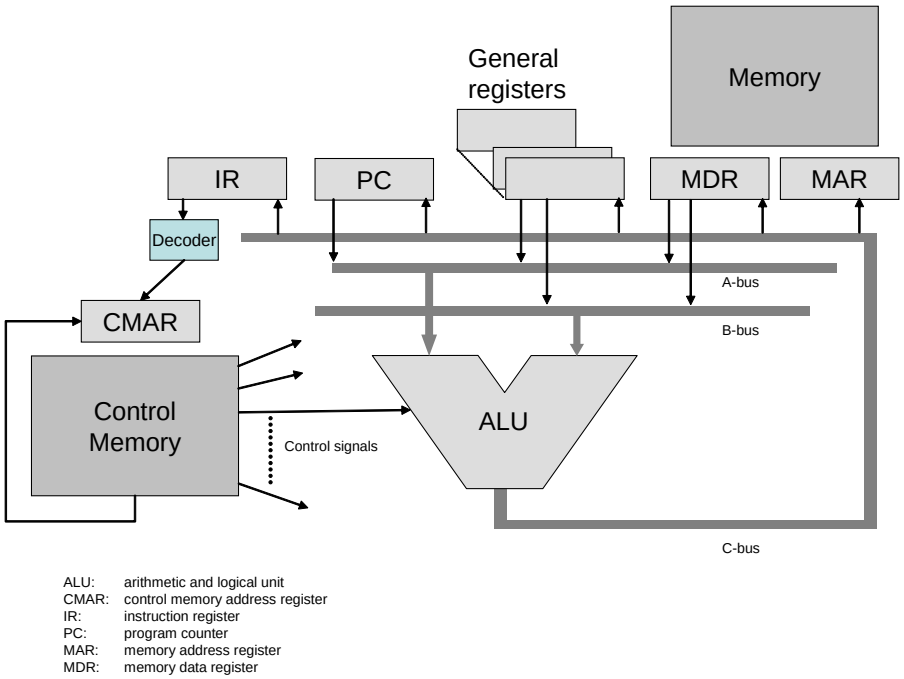
In fig. 1 a general architecture of a processor is sketched. The heart of the processor is the arithmetic and logical unit (ALU) to perform the arithmetic and logical operations. The ALU is connected with three busses: one at each entrance (A and B) and one at

[1] Sir Maurice Wilkes was Director of the Cambridge Computer Laboratory throughout the whole development of stored program computers starting with EDSAC; inventor of labels, macros and microprogramming.

[2] In those days indicated as the central processing unit (CPU).

the exit (C). The busses form together with the ALU a transport channel for data between several registers (at one combination of control signals the ALU transmits only the data from one entrance to the exit). Some registers are implicit in use, such as the program counter (PC) and the instruction register (IR) while other registers have to be addressed explicitly in the instruction.

Figure 1: CPU outline



By placing an extra register at the entrance of the ALU one can delete one bus and by adding a further extra register at the second entrance even one bus suffices. It will be clear that the speed of instruction execution in that case proportionally decreases.

The whole process of instruction execution is controlled by signals coming from the control memory (CM).

3. Instruction Architecture

To fully understand the working of a traditional computer we look not only at the hardware, but also at the instruction format. Fig. 2 represents a general format.

Figure 2: Instruction format

The first part of the instruction indicates what operation has to be executed. The other parts indicate in which memory places the operands are found and where the result has to be placed. The three addresses are not always specified. To understand the ratio for that one has to consider that in most architectures for reasons of execution speed the length of an instruction (number of bits) is equal to the length of the words in main memory: in that case only one memory cycle is needed to fetch an instruction. But the larger the main memory is, the longer is the address. At the other hand one wants to have a large number of different operations, so ideally the operation code is long. These wishes are conflicting so that the designer has to seek for an optimum.

To begin with one can reduce the number of addresses in the instruction by always placing the result in source address 2. The price one has to pay for that reduction is that one of the two original operands is lost. A further reduction can be reached when by definition one operand is the content of a register (the number of registers is always far less than the number of addresses in the main memory so one needs but a small number of bits to address a register). That requires sometimes an extra operation to place the operand in that register with as a consequence that the program execution slows down. In some computers even one special register was implicitly indicated to contain one operand and afterwards the result.

In the first minicomputers that came on the market in the late sixties (like the PDP-8 from Digital Equipment Corporation) the instruction had one address and the other operand was supposed to be in a special register (called the accumulator), while the result was placed back into the accumulator. Later on there came computers on the market, like the PDP-11 from the same company, that had a complex operand specification: two address parts, each part divided into a register field and a mode field. The register field indicated a register that was supposed to contain a memory address. The mode field specified the manner in which that address had to be used (e.g. direct or indirect).

To keep it simple we will use in the following part of this paper an instruction type with one memory address that points to one operand and an implicit specified register containing the other operand and receiving the result.

4. Instruction execution

Most instruction cycles can be divided into three phases:
The instruction fetch phase
The data fetch phase[3] and
The execution phase.

Each phase can in its turn be divided into several steps. Each step corresponds with one word from the CM: the microinstruction.
The microinstruction can in general be described with figure 3.

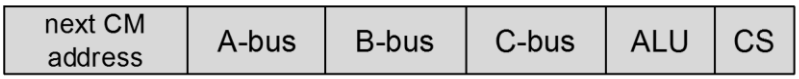


Figure 3: General outline of a microinstruction

Each CM word contains the address of the next microinstruction to read. Further there are three fields that control the output and the input of the several registers on the buses. The next field defines the function the ALU has to do. The last field specifies a number of independent control signals (like a command to read main memory). A quite simple architecture required a word length of roughly 80 bits.
The last microinstruction of each execution phase points to the first one in the fetch phase for the next instruction. The fetch phase is the same for every instruction and is found in a fixed place in CM. The program counter (PC) contains the address of the next instruction to be executed. That instruction itself is still unknown until it is fetched from the main memory. So the content of the PC register is fed via the ALU to the memory address register (MAR) and thereafter a read command is given to the memory. As a result of the read operation the word in that place – the instruction to be executed – comes out in the memory data register (MDR) and is further transported to the instruction register (IR). In between the content of the PC has been incremented with one so that the PC already points to the next instruction. The first part of the IR, indicating the wanted operation, is connected to a decoder that gives an address of the control memory. That address is put into the CMAR and by starting a read operation for the CM the first step of the new phase is started.
Knowing now what operation has to be executed the question arises on which operands it has to be done. Finding the operands is closely related to the design[4] of the instruction. As indicated earlier we take the most simple manner in this paper: an address with an extra bit indicating that the address is direct or indirect[5]. The address part of the IR is now transported to the MAR and a memory read operation is started to fetch the specified operand. If the '*indirect bit*' in the instruction is 1 an extra

[3] Only the instructions influencing the sequence of instruction execution, such as jump instructions, don't have a data fetch phase in general.
[4] In some architectures, like the PDP-11 family from Digital Equipment, there are several addressing modes that require some arithmetic to determine the place of the operands.
[5] With direct addressing the address points to the operand, while indirect addressing means that the address points to a location which contains the address of the operand.

memory read cycle has to be executed because the address part of the instruction does not point to the operand itself but to the place where the address of the operand can be found.

After the operand has been fetched from main memory the content of MDR is offered to the one input gate of the ALU and the content of the accumulator to the other gate. At the same time a command is given to the ALU to perform an arithmetic or logical operation and the output is fed back to the accumulator.

5. The idea of microprogramming

In most minicomputers being on the market in the late seventies the CM was constructed as a read only memory (ROM). That was already a step forwards in relation to older machines where the control part was a specially designed complex of counters, gates, etc. – the design with a CM gave a clear architecture as we saw in the preceding paragraphs. But the combination of the need to perform special operations with this well designed architecture gave the idea to replace the CM by another, a variable one if that was possible.

6. The First Experiment

The laboratory in which the first experiment took place had a minicomputer of the type PDP-9 from Digital Equipment Corporation. This machine still was composed of lumped components: transistors, resistors, etc. With several colleagues we followed a course for technical service of that machine, given by a specialist from Digital Equipment so we knew after the course exactly which signals flew through the machine at any time.

Our first aim was to make the machine run as delivered, but now with a new CM. To run the original programs the content of that CM had to be the same as the original one. Because of the wish to make it variable we took a matrix board[6] (fig. 4) with enough rows and columns to let the machine run its original instruction packet.

[6] As was used to program analog computers or, in older days, Hollerith machines.

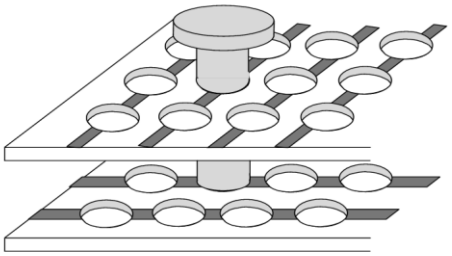


Figure 4: Schematic detail of a matrix board

We were lucky in so far that the original ROM consisted of separate magnetic cores, so we could use the original electronic circuits that select a row as the result of a handed over bit combination (indicating the CM 'address').

Having a 'programmable' CM we were able to make changes in the execution of instructions. For example we could implement 'incremental indirect addressing' whereby the address of the operand is automatically incremented each time the operand is used. This way of addressing saves an instruction and thus quickens the handling of data in a list: supposing this data handling requires a number of instructions, the whole list can be processed by repeating these instructions again and again with the same instructions and the same addresses as illustrated in Figure 5.

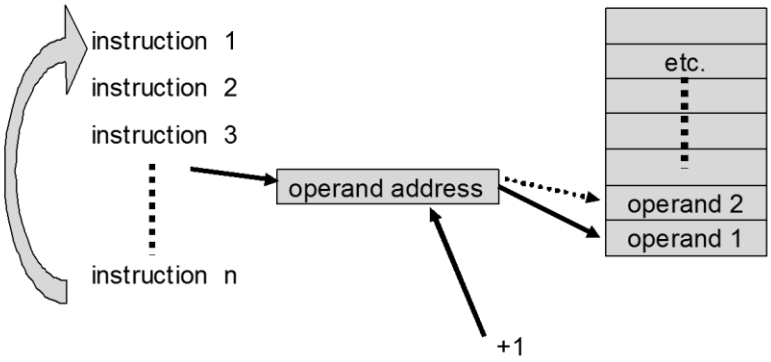


Figure 5: Incremental indirect addressing

7. Extensions

Having a changeable ROM the idea arose to make the content more flexible and programmable by using a random access memory (RAM).

This idea was compelling but we had to realize that by replacing the whole CM with a different content the original instruction execution was destroyed. So it was decided only to add a new part to the CM with RAM. That however presented several problems. At first there was a slight technical problem: the electric signals that came from the ROM didn't agree with the electric power levels of the signals that came from standard RAM devices. That problem could simply be tackled. But a severe problem was filling the CRAM[7]: how and with what data?

To fill the CRAM we found a solution in regarding the CRAM as a special output device: with I/O-instructions the CRAM could be filled from the general memory via the I/O-bus. The only obstacle was that the word length of the CRAM didn't agree with the width of the data path on the I/O-bus. But that could simply be avoided by using a number of data transfers for one word in the CRAM. We decided always to fill the CRAM with a block transfer because we saw no need to replace single words of the RAM. Filling the CRAM was now comparable with writing a disk drive.

8. Micro-Assembler

Now the problem still remained on how to define the content of the CRAM. In a brain storming session the idea arose to look how the general memory was filled with instructions. At a high level we used programming languages like FORTRAN, Algol etc. But at a lower level the manufacturer designed an assembler language that facilitated writing programs – e.g. the drivers for the peripherals – in symbolic instructions (e.g. the instruction to add two numbers was indicated with the acronym ADD), with symbolic addresses. The assembler program converted the symbolic instructions into rows of bits, so we designed a micro-assembler. Now we could program new instructions and even test them before implementing them.

9. Commercial products

At about the time we got the micro-assembler, two manufacturers of minicomputers launched a new model of their minicomputer family: Digital Equipment Corporation put the PDP-11/60 on the market and Hewlett Packard a model of the HP-2100A. Both had all the benefits we already discovered.

As far as I know these were the only commercial available machines with microprogramming facilities and they were at the same time the last scions on the family tree of small minicomputers, because microprocessors and PCs soon came along.

[7] To avoid long descriptions we use the acronym CRAM for the control memory with RAM.