Control and Cybernetics

vol. **39** (2010) No. 2

Concurrent operation of processors in the bit-byte CPU of a PLC*

by

Mirosław Chmiel and Edward Hrynkiewicz

Institute of Electronics, Silesian University of Technology
Akademicka 16, 44-100 Gliwice, Poland
e-mail: {Miroslaw.Chmiel, Edward.Hrynkiewicz}polsl.pl

Abstract: The paper presents some selected hardware solutions for the PLC dual processor bit-byte CPUs, which are oriented at optimised data exchange between the CPU processors. The optimisation aims at maximum utilisation of capabilities of the two-processor architecture of the CPU. The key point is preserving high speed of instruction processing by the bit-processor, and high functionality of the byte-processor. The structure should enable the processors to work in concurrent mode as far as it is possible, and minimise the situations, when one processor has to wait for the other.

Keywords: programmable logic controller, central processing unit, bit-byte structure of CPU, scan time, throughput time, concurrent operation.

1. Introduction

One of the basic parameters that determine the performance of Programmable Logic Controllers (PLCs) is the time needed to execute one thousand instructions (Michel, 1990). If the value of this parameter is low, a possible range of PLC applications is wider. This is why design and development of a unit that would enable execution of a control program during an extremely short time is becoming a very important task. Owing to its constructional features, such a unit should not only cover all the possible functional requirements but also make possible taking maximum benefits from the provided features. When attempting to propose a cost-effective solution that is fast enough and assures assumed time limits, one should consider a bit-byte structure of a controller CPU. Thanks to their special operational features such structures enable achieving satisfactory results in the case of both binary signal processing, owing to the inclusion of a dedicated bit processor, and handling the analogue signals, which is carried

*Submitted: October 2005; Accepted: October 2009.

out by a standard, cheap microcontroller. Such a structure includes two separate components: a bit unit and a byte (word) unit. Therefore, the control program language has to be subdivided into two parts. Such an approach is justified by the fact that special features of instructions for the two processors are different. The bit processor executes every instruction during a single clock cycle, whereas for the byte processor every single instruction is equivalent to a procedure that must be developed in an assembler or high-level programming languages like C/C++, Pascal or other. Within the confines of this paper the authors intended to focus their attention on analysing the possibility of concurrent work of both processors to get a minimum throughput time and scan time of the central processing unit.

2. A program example

Let us consider an example, allowing for the observation of the capabilities of the bit-byte central processing unit in a PLC. A simple belt conveyor with a remote controlled pneumatic cylinder is considered. Objects that are delivered by the conveyor after reaching its endpoint are pushed from the belt.

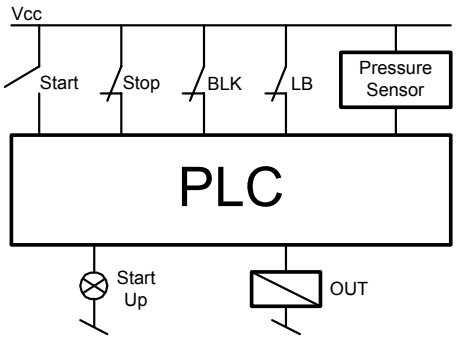

Figure 1. An example of a circuit signal connection

The control program consists of fragments that are executed by the bit processor and the byte processor. Some actions require co-operation of both processors. There are four digital inputs connected to the controller (Fig. 1). Two of them are connected to push-buttons and the other two are connected to sensors:

- *Start* – depressing this button starts the process. The process begins from a *Start-up* procedure. The *Start-up* procedure is completed after the *Start-up* signal goes up,
- *Stop* – depressing this button stops the entire process immediately and turns off the output,
- *BLK* – blockade. If this signal is activated, the system is out of operation,
- *LB* – *Light Barrier*. When activated, the output must be turned off.

The analogue input delivers information about pressure in the object installation.

There are also two binary outputs used:

- *Start-up* – is active for 5 seconds after depressing the *Start* button (when *BLK* is active and the installation pressure is higher than the minimal value),
- *Output* – a signal that controls the output. This signal is activated after five seconds of depressing the *Start* button and if the *BLK* signal is active as well as the satisfactory pressure is present.

The control algorithm, written in the LD graphical language for Simatic S7-300 (Berger, 2001a) is presented in Fig. 2. Apart from symbols of switches and coils, two block components are used. A comparator is used for the current pressure comparison with a reference value. A time unit is used to delay switching on the *Output* signal.

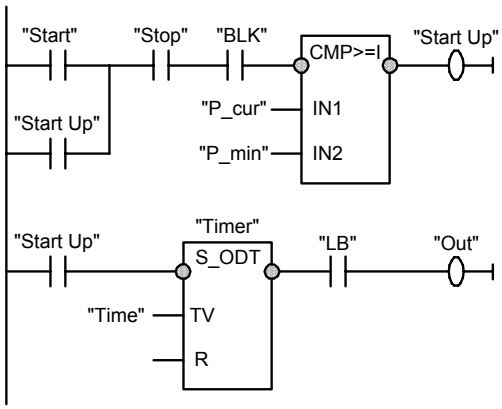

Figure 2. The ladder diagram (LD) program

In Fig. 2 all places are marked where passing of logical conditions between processors is required. As it seen, the logic condition based on signals *Start*, *Stop*, *BLK* and *Start-up* activates the comparison operation. The comparison result drives the *Start-up* signal. When the *Start-up* signal is activated, it triggers time counting in the timer block. The timer output in conjunction with the optic sensor signal determines the *Out* signal value. There are four different places, where information is passed between processors in the CPU. Each block component on the Ladder Diagram is triggered from logic signals. On the other hand, each block drives logic signals. This simple example gives an overview of logic state transfer in the bit-byte PLC CPU between two processors. Another problem concerns the control program instruction ordering and instruction fetching and passing between the processors as well as operation synchronization.

3. The single processor CPU

The simplest PLC CPUs are designed with the use of a single general-purpose microprocessor or microcontroller. The following approach allows for obtaining a simple and inexpensive solution with a greatly improved performance. The performance is limited by the lack of a bit operation unit. The operation on bit variables requires performing several instructions. In fact, most of them are used to extract bit variables from a byte or a word in order to perform the operation. Finally, some additional operations are required in order to store the result in an appropriate place. Bit procedures executed by a byte (or a word) processor require much more time than a similar operation performed on entire words or bytes.

Let us consider a byte and a bit operation performed with the use of the 8051 microcontroller (Gałka, 1995).

The byte operation adds binary contents of two cells with symbolic addresses MB0 and MB2 and places the result in the cell MB4. All memory cells are located in external memory (Fig. 3).

Program for MCS'51	Description	Cycles
MOV DPTR,#MB0	MB0 → DPTR	2
MOVX A,@DPTR	(DPTR) → A	2
MOV TMP,A	A → TMP	1
MOV DPTR,#MB2	MB2 → DPTR	2
MOVX A,@DPTR	(DPTR) → A	2
ADD A,TMP	A + TMP → A	1
MOV DPTR,#MB4	MB4 → DPTR	2
MOVX @DPTR,A	A → (DPTR)	2
		14

Figure 3. Byte operation example in the MCS-51 assembler

The second operation is AND operation performed on two bits located in M0.0 and M2.0. The result should be stored in M4.0 (Fig. 4).

As it was presented, an operation on bits, even though the variable size is reduced, requires executing more instructions than the typical byte operation.

In Fig. 5 a control program for the belt conveyor, written with the use of Instruction List language is shown. The presented program is based on instructions and semantics from Siemens (Berger, 2001b). An important fact is that a logic instruction takes only one argument. This makes them very simple in use and increases readability of the program. There are also combined instructions that perform bit and byte operations, like value comparison, which is combined with logical AND operations.

Program for MCS'51	Description	CLK
MOV DPTR,#MB0	MB0 → DPTR	2
MOVX A,@DPTR	(DPTR) → A	2
MOV TMP,A	A → TMP	1
MOV DPTR,#MB2	MB2 → DPTR	2
MOVX A,@DPTR	(DPTR) → A	2
ANL A,TMP	A **AND** TMP → A	1
ANL A,#0Eh	A **AND** #0E → A	1
MOV TMP,A	A → TMP	1
MOV DPTR,#MB4	MB4 → DPTR	2
MOVX A,@DPTR	(DPTR) → A	2
ORL A,TMP	A **OR** TMP → A	1
MOVX @DPTR,A	A → (DPTR)	2
		19

Figure 4. Bit logic operation in the MCS-51 assembler

```
1.  TH  "Start"      ;Bit Instructions
2.  O   "Start-up"
3.  A   "Stop"
4.  A   "BLK"

5.  L   "P_cur"      ;Byte Instructions
6.  L   "P_min"
7.  A>=I             ;Byte-Bit Instr.
8.  =   "Start-up"   ;Bit Instruction

9.  TH  "Start-up"   ;Bit-Byte Instr.
10. L   "Time"       ;Byte Instruction
11. SD  "Timer"      ;Byte-Bit Instr.

12. TH  "Timer"
13. A   "LB"         ;Bit Instructions
14. =   "Out"
```

Figure 5. Program 1 written in the Instruction List

A comparison takes two arguments that are represented by numbers while the comparison result is represented by a bit variable that stores the logic true - 1 and false - 0. The instruction that triggers the timer function (SD) is executed on the basis of the *Start-up* variable. When the execution of the SD instruction succeeds, an initial value of a given time interval is loaded to a memory cell and the timer starts counting.

As it was presented in the above example, there are two precisely tailored groups of instructions. One of them operates on numerical variables. A typical microprocessor or microcontroller can process such variables. Bit instructions belong to the second group. Microprocessors or microcontrollers are not optimised to perform operations on such variables.

In that case the implementation of a processing unit that is able to process bit variables much faster than a typical microprocessor is quite natural. It is also important that such a solution should be inexpensive and simple. In order to overcome the limitations of a general-purpose microprocessor, a specific bit processor must be designed. It must be optimised for the fastest execution of bit instructions. There is also a need to design additional circuitry that will allow to synchronise the computation process in both processors and solve conflicts in access to common resources.

4. Bit-byte dual-processor CPUs

A basic design of a bit-byte CPU has a common instruction memory. Each processor fetches an instruction, as it is needed. In this mode the processors operate in serial way, waiting for each other until the instruction execution is completed.

By introducing dual-processor architecture some performance improvement is expected even with single instruction stream. Bit operations are time consuming for a general-purpose processor or microcontroller. An additional processor, designed for bit operation greatly reduces the time and program overhead for bit operation because bit operation can be executed very quickly (Chmiel and Hrynkiewicz, 1999, 2001).

In simple dual-processor architecture the instruction set is similar to that presented in Fig. 5. The bit processor executes bit instructions while the byte processor remains idle. The byte processor executes byte instructions while the bit processor is waiting for an instruction. Complex bit-byte instructions are expanded into instructions for the bit and byte processor in an appropriate order.

This basic dual processor architecture can use a bit-processor as a kind of co-processor for specific operations (Aramaki et al., 1997). When both processor units are equally privileged, then additional arbitration circuitry is required. The circuit initially decodes an instruction and directs it to the proper processing unit.

In Fig. 6 a dual-processor architecture (Getko, 1983) is presented where the instruction decoder selects a processing unit for the execution of the current instruction. When instruction processing is completed, the active processor increments the instruction counter and the cycle starts over. The instruction decoder is usually a part of the bit processor. The instructions processing is deterministic and controlled by a common instruction decoder for both processors. A common instruction memory forces a serial way of the control program execu-

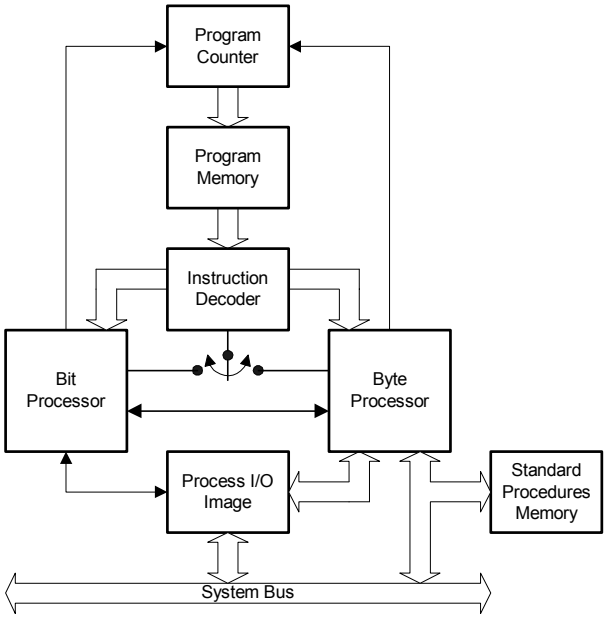

Figure 6. Dual processor CPU with the common program memory

tion. The byte processor is typically equipped with a local standard procedures memory. In this memory common procedures for a complex control instruction are stored. Such functions as PID or FUZZY controllers, timers, counters, advanced arithmetic functions, network and remote communication servicing etc, are usually implemented as standard procedures. Such a solution utilizes high speed of the bit processor operation, however it has to execute the control program in a fully serial way because the system is equipped only with one control program memory, one process I/O image and one system bus. In the presented solution the standard procedures memory is used. The procedures stored in this memory could be executed by the byte processor concurrently with the bit processor operation. On the basis of the above observation an idea of equipping both processors with their own program memories may be carried out. The appropriate part of the program is placed in bit and byte processors program memories.

Such an approach was proposed by Donandt (1989). The computation process is controlled by the byte processor that transfers tasks to the bit processor. The bit processor is an autonomous calculation unit independent from the master processor. The approach presented allows for concurrent operation of both processors. The byte processor (master), after calculation initialisation of the bit processor, is able to resume its operation and continue program execution instead of waiting for the task completion in the bit processor. There are some

architectural limitations that reduce performance and concurrency. Only the master processor can access the process image memory. This limits a parallel program execution.

The presented examples of bit-byte CPUs deliver guidelines and requests for a new architecture design. A dual-processor CPU should be equipped with a fast bit processor. This processor can be designed as custom hardware, for example with the use of programmable logic components. The bit processor is responsible for instruction fetching. The byte processor is equipped with a local standard procedures memory that contains implementations of all byte instructions that this processor is expected to perform. Each instruction for the byte processor is a subprogram that is executed as a request from the bit processor. In order to avoid conflicts during data memory access each processor uses its local data memory.

The described CPU architecture is shown in Fig. 7 (Chmiel, 2004). A control program is stored in the main program memory. The program contains instructions for the bit processor. The byte processor instructions are represented by entry points addresses to appropriate subprograms. The byte processor, after execution of a instruction completion transfers its ready state to the bit processor. Then the bit processor fetches the next instruction and passes it to the execution to the appropriate processor.

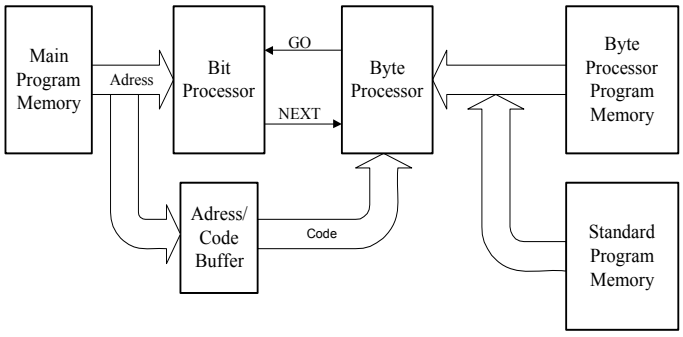

Figure 7. Bit-byte CPU with a master bit processor

A flow diagram for the program execution is shown in Fig. 8. In the presented circuit the processors are able to execute single instructions or a group of instructions for a given processor concurrently. Such a possibility decreases the time of control program execution.

Let us return to the previously considered example and consider, step by step, the program execution process by the CPU from Fig. 7, which operates according to the flow diagram presented in Fig. 8.

The control program shown in Fig. 9 consists of different types of instructions:

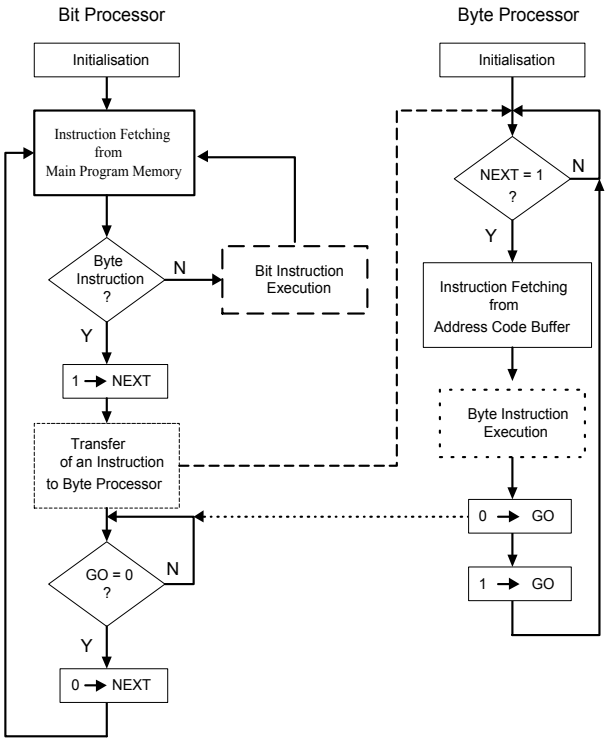

Figure 8. Program execution in the bit-byte CPU

- Bit instructions stored in the main program memory, executed usually in one clock cycle;
- Byte instruction that requires byte processor program memory access;
- Complex instructions that require processor co-operation with the result of operation depending on both processor calculations being executed in the proper order.

Byte instructions can have the following format:

- Simple without arguments – an instruction consists of an operation code that points to a program memory place where standard procedure can be found (A>=I);
- Simple with a single argument – an instruction modified by the compiler. As the result of modification the special procedure is placed in byte processor memory (L "Time");
- Complex instructions that require co-operation of the bit-processor. An instruction of this type issues appropriate actions in both bit and byte processors (SD "Timer").

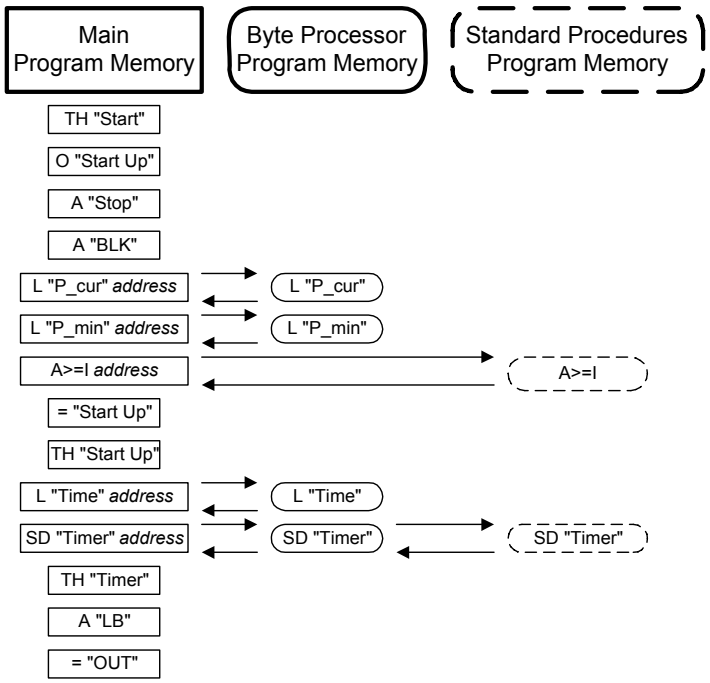

Figure 9. Instruction allocation - case 1

All bit instructions are executed very quickly (each instruction in a single clock cycle). A byte instruction must be passed to the byte processor. The signal NEXT is activated (Fig. 8). Bit processor enters the wait state until the signal GO is activated.

It can be suggested that three successive byte instructions can be initiated once by the bit processor. This approach allows for further shortening of the instruction execution time. The same optimisation can be applied to the next two-byte instructions. The obtained program is shown in Fig. 10. The proposed modification results in shortening of the main program. The execution time of the byte instruction group is also reduced, as they are stored in the byte processor memory now. The overall execution time of the program loop is also reduced.

Additionally, it can be noticed that the first two byte instructions are independent of the following bit instructions. As those instructions are independent of each other, they can be executed concurrently with these bit instructions. The following situation is considered in Fig. 11. In this way the program execution time has been further reduced by concurrent operation of bit and byte processors.

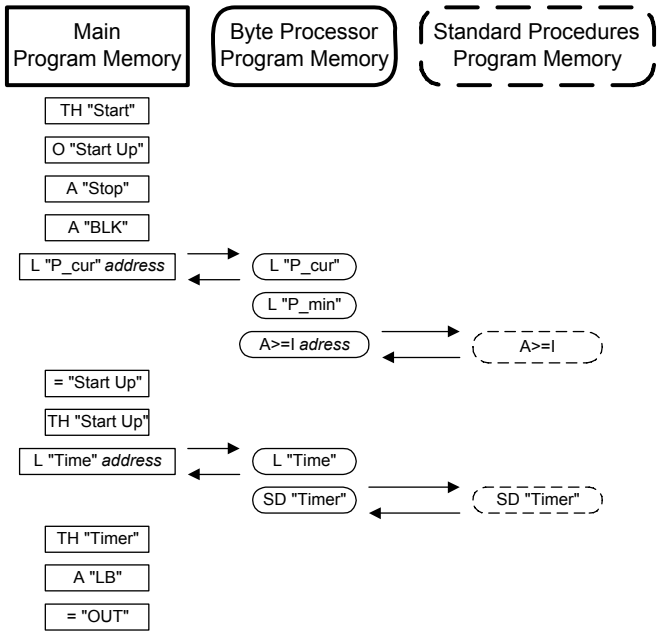

Figure 10. Instruction allocation - case 2

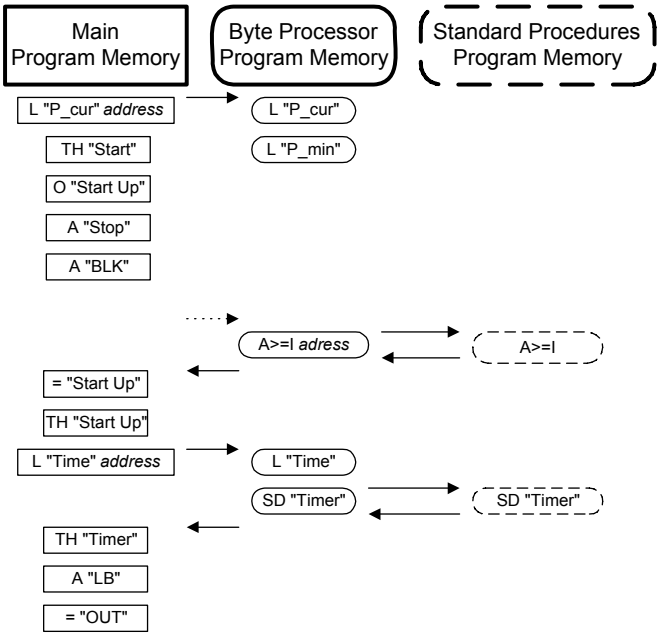

Figure 11. Instruction allocation - case 3

5. Inter processor data transfer

In a single processor unit a condition flip-flop is used for calculating a logic function that determines the execution of further operations. The two-processor CPU is equipped with only one condition flip-flop too. This flip-flop is also required, but it is available for both processors and is used by a particular processor as needed. The CPU built in such a manner enables only the execution of the serial program.

A single condition flip-flop limits the parallel operation of the CPU. In the case of the considered construction the bit and byte processors must be able to process and exchange logic conditions with each other without interrupting the operation of the other processor. This requires implementation of two logic condition flip-flops, one for each processor.

This allows for the execution of the program fragments that require a logic function execution in each processor without influencing one another.

Let the condition flip-flops be called F_B and F_b for the byte and bit processor, respectively. The information stored in flip-flops must be transferred to the appropriate processor. It can be done by transferring a value from F_B to F_b and also from F_b to F_B. This data transfer does not require additional hardware overhead.

There is a serious limitation in the described construction. When a processor wants to access the condition flip-flop of the other processor, the requestor has to wait until access is granted.

Such problems are not very important for a serial program execution, but in the case of concurrent program execution become an important object of interest. In order to reduce the number of synchronisation wait states, a modified construction of condition registers can be proposed, introducing specific buffered condition flip-flops that are available for both processors.

There are two additional condition flip-flops that store a copy of the main condition flip-flops of the appropriate processor. Each of the processors transferring the condition flip-flop to the buffer flip-flop is able to continue program execution until the next condition flip-flop update. New information can be written to the buffer flip-flop only if the previous content was read out by other processor. When buffer flip-flop is empty (does not contain valid condition bit), condition bit can be transferred and the processor can immediately resume its operation. This requires an appropriate compiler able to insert synchronisation instructions into the compiled code while the program designer is concentrated on problem solving.

A schematic diagram of the proposed construction is presented in Fig. 12. Condition buffer flip-flops are called F_{Bb} (transfer from the byte to the bit processor) and F_{bB} (transfer from the bit to the byte processor).

Let us return to the previously considered program presented, in Fig. 5. In this new situation the program can be split into two independent parts. The first part generates the signal *Start-up*, while second part creates the signal *Out*

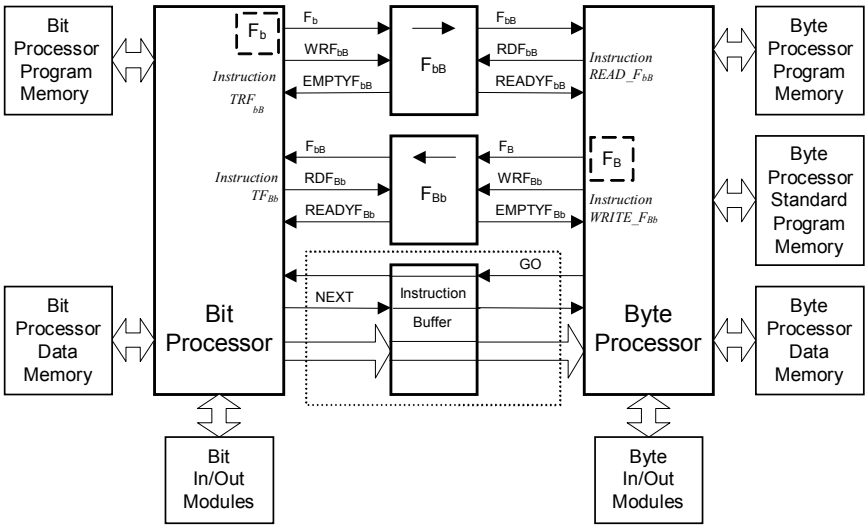

Figure 12. Block diagram of the parallel CPU structure

based on *Start-up* signal (see Fig. 3).

The first part of the program starts with four instructions for the bit processor that use the flip-flop F_b. Instructions number five and six are executed by the byte processor. A critical part of the program is the instruction number seven. Logical-AND is calculated on the result of comparison operation executed by the byte processor and the logic result worked out by the bit processor. Finally, a logic condition is obtained for the *Start-up* output.

The operation of both processors is required in the presented part. The bit processor executes its part of program and finally performs logic-AND with contents of the flip-flop F_{Bb}. The byte processor performs load and comparison operations. The result of comparison operation is transferred to the flip-flop F_B and F_{Bb}.

A program listing for the dual-processor CPU is presented in Fig. 13. There are two additional instructions that are responsible for transferring the comparison result to F_{Bb} and logic-AND operation with the content of this flip-flop.

The dual-processor CPU introduces little overhead in control program. In the presented program fragment two additional instructions appear. On the other hand, this overhead allows parallel and concurrent operation. First four instructions for the bit processor can be executed with the next two instructions for the byte processor in a parallel way.

The synchronisation point is located in the 9th instruction. At this instruction the bit processor will wait for the byte processor until the comparison operation is completed and the result is transferred to the flip-flop F_{Bb}.

```
1.  TH    "Start"     ;Bit Processor
2.  O     "Start-up"
3.  A     "Stop"
4.  A     "BLK"

5.  L     "P_cur"     ;Byte Processor
6.  L     "P_min"
7.  >=I               ;Both Processors
8.  Write FBb         ;FB to FBb

9.  A     FBb         ; Fb AND FBb
10. =     "Start-up"
11. =     FbB         ;Fb to FbB

12. Read  FbB         ;test FBb
13. L     "Time"
14. SD    "Timer"

15. TH    "Timer"
16. A     "LB"
17. =     "Out"
```

Figure 13. Program 2 in the Instruction List

It can be noticed that the total execution time has been reduced. The execution time is mainly determined by the byte processor, which executes four instructions (two time load, compare and transfer instructions). The bit processor executes its instruction almost in the background of the byte processor operation. Its operation does not extend the program loop execution time. The presented benefits are lost in the serial-cyclic program execution. In the serial-cyclic way of operation only one processor executes an instruction while the other processor waits for its completion.

Continuing with the exemplary program, it can be noticed that the start of time counting is only possible when the *Start-up* output is active. The bit processor controls this output. The state of the *Start-up* output must be transferred to the flip-flop F_{bB}. On the basis of the state of F_{bB} the byte processor is able to trigger the timer operation. In the program, two additional instructions are inserted that allow for the transfer and receive the *Start-up* bit value between the processors (Fig. 13).

There is one additional point of the inter-processor data transfer. The bit processor requires the status of the timer/counter while the byte processor performs timer/counter procedures. Using the programming mechanism presented above it is required to place two additional instructions for exchanging logic conditions among the processors.

This can be avoided by a specific hardware extension dedicated for timers/counters, which are implemented as additional registers of an external buffer that

stores timer flags. This buffer is updated by the byte processor (which services all timers/counters) and can be read out by the bit processor. It allows to read out the state of the timer by the bit processor without any need of logic condition exchange. The timer register is mapped in the input space of the bit processor. This allows for avoiding program overhead and additional synchronisation point for timer operation (Chmiel, Ciążyński and Nowara, 1995).

The modified timer construction changes operation of the TH Timer instruction that is now executed only by the bit processor. It tests the state of the appropriate timer register bit. The block diagram of timer implementation is shown in Fig. 14.

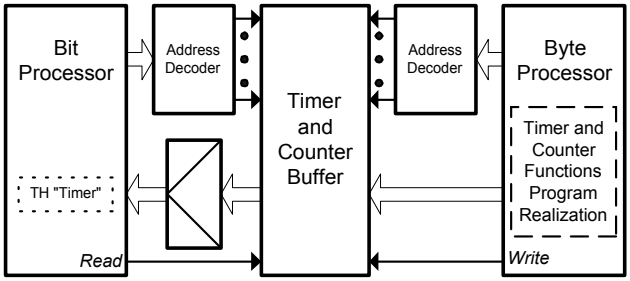

Figure 14. Realisation of the Timer and the Counter

After all modification in the hardware structure of the controller, the program consists of seventeen instructions, with four of them transferring a logic condition and synchronising operation of the processors. It is important that all those instructions belong to the basic set of instructions and their execution is relatively short. In Fig. 15 the program is presented that is split into two columns for the bit and byte processor with additional transfer and synchronisation instructions. The discussed case requires transfer of conditions in both directions between the processors.

Bit Processor		**Byte Processor**	
TH	"Start"		
O	"Start-up"	L	"P_cur"
A	"Stop"	L	"P_min"
A	"BLK"	>=I	
A	**F_{Bb}** ◄———	**Write**	**F_{Bb}**
=	"Start-up"		
=	**F_{bB}** ———►	**Read**	**F_{bB}**
TH	**"Timer"**	L	"Time"
A	"LB"	**SD**	**"Timer"**
=	"Out"		

Figure 15. Program 3 in the Instruction List

It can be noticed that the 5th instruction for the bit processor requires reading of the logic condition, worked out by the byte processor. The synchronisation point should be placed for the instruction A F_{Bb}, which requires an updated value of pressure comparison.

The observable result of such arrangement operations on the bit variables are hidden under operation on the byte variables. The execution time is not the sum of all instruction execution times, since byte instructions are executed concurrently with bit instructions.

By applying the same mechanisms to other instructions a concurrent operation effect can be achieved. A processor waits for the counterpart unit when the synchronisation data exchange point is achieved. It can be noticed that the synchronisation mechanism does not require transferring of an instruction from the bit processor to the byte processor. It means that the program is also free from additional operations connected with instruction transfer. Additional instructions are only inserted for synchronisation of logic condition exchange. The total program execution time is equal to the maximum value of the byte processor program execution time and bit processor execution time. But usually, byte instructions are executed longer than bit instructions, so the execution time is mainly dominated by operation of the byte processor.

6. Remarks on further development of the Bit-Byte CPU of PLC

Finally, a CPU can be constructed of two almost independent processors.

The unit is equipped with three memory banks for control programs: the program memory for the bit-processor, the program memory for the byte-processor and the memory unit for standard procedures. As it was assumed, both processors can operate independently of each other. They are able to execute instructions from the respective memory units and to access input/output modules simultaneously (Chmiel, Ciążyński and Hrynkiewicz, 1995).

In Fig. 12 the block diagram of the bit-byte CPU is shown. The part responsible for instruction passing to the byte processor was removed, while each processor possesses its own program memory. Processors are synchronised by the state of condition flip-flops F_{Bb} and F_{bB} (empty/full).

Instructions to the processors are delivered from their local memories, so that they do not have to wait until an instruction is delivered from the common program memory. The processors enter the wait state only when they attempt to read the empty condition register or to overwrite an unread condition in a register. This allows for an effective concurrent operation of both processors. Some performance reduction can be expected in the two following cases:

1. One of the processors does not complete the awaited operation and the other one has to wait for the result ($READYF_{Bb} = 0$ or $READYF_{bB} = 0$).

2. The condition bit was not read-out by the other processor and a new one is waiting for writing to condition flip-flop ($\mathrm{EMPTYF_{Bb}} = 0$ or $\mathrm{EMPTYF_{bB}} = 0$).

In order to avoid many wait states the program should be written and compiled in such a way that the load of both processors is equally distributed in the operation time. Further optimisation can be achieved by increasing the number of flip-flops that pass a logic condition between the processors. This leads directly to the implementation of a common memory with two access ports (dual port RAM) in which the states of condition flip-flops for both processors are stored. In that case the problem of cell assignment to given tasks or functions appears.

The assignment problem can be solved in two ways:

1. Condition flip-flops are constantly assigned to a given program instruction or to smaller fragments of the program that can be called tasks, automatically by the compiler, e.g. the comparison instruction of the byte processor sets a selected flip-flop, a counter sets another flip-flop, etc. Basing on that concept each instruction that ends with logic result possesses its condition flip-flop. A given flip-flop is modified only when the same instruction is encountered again in the program stream. This solution is limited by small flexibility and frequent access cycles to the common memory, in which logic conditions are stored.

2. The second possibility is to charge the user with cell assignment. A similar situation can be observed in the Modicon 984 controller programming (Modicon, 1990). Block instruction outputs may be assigned to memory cell but this is not necessary. If block output is not assigned, its output value is not stored. Connecting blocks together propagates operation result to following blocks.

In the considered case cells are used instead of using condition flip-flops. The operation results are exchanged through the common memory available for both processors. States of cells that allow for conditional program execution and synchronisation between processors are stored in this area.

Therefore, it is possible to overcome the problem of idle waiting of one processor until result is ready from the second one. Having many cells instead of waiting for a given flag the processor can continue the control program execution. The given cell will be tested in the next scan.

It can be noticed that during a control program loop execution by one processor the second one (especially a bit processor) may change the state of its condition flip-flop (cell) a few times. This causes that the parts of the control program, executed by the first processor utilise different states of the condition flip-flop (cell). The rule of the serial program execution is violated. Both processors operate asynchronously with respect to each other. They observe the cells area (like the I/O space) in order to perform actions that can be executed in response to changing cells.

Preserving serial program execution by each processor causes that conditions are generated in the same order. This allows for designing a specific hardware with an extremely fast access to cells. The solution is based on a set of D flip-flops.

As the set of flip-flops two registers may be implemented – one for each direction of condition transfer. These registers are connected to one processor with write access while the other one has read access to them (Fig. 16). The position that is currently written to, is pointed by a counter that is incremented by the processor after write operation. After reading data from the register the read pointer is also incremented. The presented registers form queues (something like a FIFO register) that hold and pass results in the order of their appearance. There is a possible danger of a queue overlap in the case when one processor is executing tasks much faster then the other one and generates a large number of synchronisation flags (Chmiel and Hrynkiewicz, 2004).

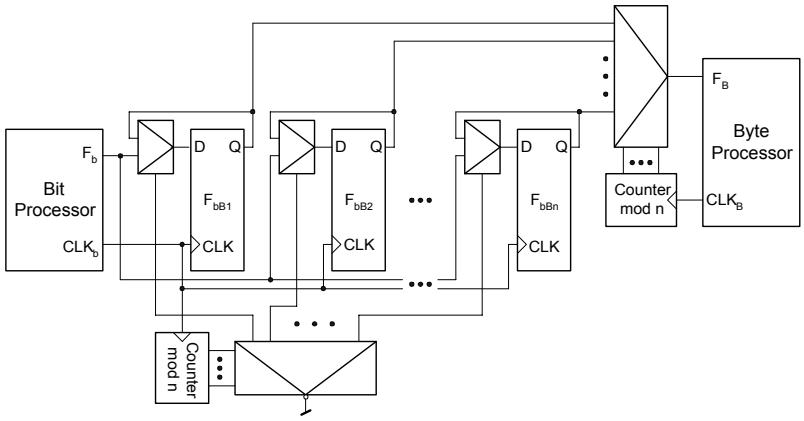

Figure 16. Block diagram of the two-processor CPU with a common memory unit

The presented architecture is extremely fast as accessing to information takes no more than one clock cycle for bit processor and port write operation for byte processor. This solution is, unfortunately more expensive and complicated than the use of a RAM. It is necessary to realize that in the case of a RAM used for condition storing, an access time to the memory is longer than to the flip-flop.

It can be noticed that the presented solution allows for unconstrained operation of both processors while the processors do not have to wait for condition passing. In the case of disproportion of program execution time in both processors, some flags are updated and examined more frequently than the others (Chmiel, Hrynkiewicz and Milik, 2005; Chmiel and Hrynkiewicz, 2005).

Can the presented execution method lead to improper operation of the entire CPU? In general, it cannot disturb the operation of the controller. In the

presented structure common information is processed like any other information delivered from sensors, actuators and so on. The counterpart processor constitutes a set of flags that are being examined and based on their states proper actions are triggered. This approach has a shorter response time, since both processors can operate with a maximum operation speed instead of waiting for calculation completion.

The introduced improvement to a PLC structure and programming allows for reducing the PLC throughput time. The processors can respond much faster to alarms and exception signals. Continuation of execution of the control program instead of waiting until the entire task in counterpart processor is completed allows for much faster operation.

7. Conclusions

Studies on the optimised information exchange between the processors of the bit-byte CPU of the PLC have shown great capabilities and many possible applications of this architecture.

As one can see from the above considerations, the proposed PLC structure or - to be more precise – organisation of information exchange between both processors of a PLC central unit allows for fast execution of control programs consisting of a bit command and/or word commands. Two modes of CPU operations were considered. The basic mode – called the dependent operation mode – brings worse timings than the independent operation mode. It is obvious, taking into account that both processors wait for the results of the operation executed by each other. The paper shows that a fully concurrent work of both CPU processors with exchange of condition flags is possible.

In the Table 1 a comparison of control program execution times for ignition burners in a steel plant furnace for different controllers and the controller built

Table 1. Comparison of a few PLCs

PLC	Number of bit instructions	Number of byte instructions	Execution time [ms]
S5-100U	1030	140	9.0
S5-115U	1030	140	1.9
S7-222	780	50	2.8
S7-224	780	50	2.9
S7-313	1000	115	2.7
S7-315-2DP	1000	115	1.5
Modicon A984	*The equivalent control program in LD representation*		8.0
80C320 – serial mode	1050	280	1.7
80C320 – parallel mode	850	140	1.1

on the basis of the above considerations is presented (Chmiel, 2004). As it can be seen from Table 1 (rows for 80C320), execution times for the designed controller are relatively small.

References

Aramaki, N., Shimokawa, Y., Kuno, S., Saitoh, T. and Hashimoto, H. (1997) A new Architecture for High-performance Programmable Logic Controller. *Proceedings of the IECON'97 23rd International Conference on Industrial Electronics. Control and Instrumentation*, IEEE **1**, New York, USA, 187-190.

Berger, H. (2001a) *Automating with STEP 7 in LAD and FBD – SIMATIC S7-300/400 Programmable Controllers*. Siemens AG.

Berger, H. (2001b) *Automating with STEP 7 in STL and SCL – SIMATIC S7-300/400 Programmable Controllers*. Siemens AG.

Chmiel, M. (2004) *Improvement of Data Exchange Between the Processors of the Bit-Byte CPU of a PLC*. PhD Thesis. Gliwice, Poland.

Chmiel, M., Ciążyński, W. and Hrynkiewicz, E. (1995) An Overview of Process Data Access and Program Execution Methods Applied in PLCs. *PDS'95: International Conference Programmable Devices and Systems*. Gliwice, Poland 9-10.11.

Chmiel, M., Ciążyński, W. and Nowara, A. (1995) Timers and Counters Applied in PLCs. *PDS'95: International Conference Programmable Devices and Systems*. Gliwice, Poland 9-10.11.

Chmiel, M. and Hrynkiewicz, E. (1999) Parallel Bit-Byte CPU structures of Programmable Logic Controllers. *International Workshop on ECMS*. Liberec, Czech Republic, 67-71.

Chmiel, M. and Hrynkiewicz, E. (2001) Remarks on Parallel Bit-Byte CPU structures of Programmable Logic Controllers. *International Workshop on DESDes*. Przytok, Poland, 147-152.

Chmiel, M. and Hrynkiewicz, E. (2004) Concurrent Operation of the Processors in Bit-Byte CPU of Industrial PLC. *International Conference on PDS*. Kraków, Poland, November 18-19, 15-20.

Chmiel, M. and Hrynkiewicz, E. (2005) Remarks on Parallel Bit-Byte CPU structures of Programmable Logic Controllers. In: M.A. Adamski, A. Karatkevich, M. Węgrzyn, eds., *Design of Embedded Control Systems*. Section V, Springer Science + Business Media, Inc., 231-242.

Chmiel, M., Hrynkiewicz, E. and Milik, A. (2005) Concurrent operation of the processors in Bit-Byte CPU of a PLC. *Preprints of the IFAC World Congress*. Prague, Czech Republic, July 3-8.

Donandt, J. (1989) Improving response time of Programmable Logic Controllers by use of a Boolean coprocessor. In: *VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks*, IEEE Computer Society Press, Washington, DC, 167–169.

Gałka, P. (1995) *Podstawy programowania mikrokontrolerów 8051 (Fundamentals of 8051 microcontroller programming; in Polish).* Mikom, Warszawa.

Getko, Z. (1983) *Programowalne systemy sterowania binarnego PLC (Programmable systems of binary control, PLC; in Polish).* Elektronizacja, WKiŁ, Warszawa.

Michel, G. (1990) *Programmable Logic Controllers, Architecture and Applications.* John Wiley & Sons, West Sussex, England.

Modicon (1990) *Modicon 984 Programmable Controller – System Manual.* AEG Modicon.