2. Načrtovanje procesorjev

2.1. Uvod, 4-bitni procesor

The on-line reconfiguration or reconstruction can be introduced at three different levels:

First: Some operations inside the standard CPU, such as multiplying and shifting, are usually realized by a special circuit. Beside the standard **arithmetic-logic unit** (ALU), we have additional data paths, which connect the ALU, multiplier and shifter parallel to the same bus. If, for example, we want to introduce a new function we would add the new circuit, which is designed via one of the hardware description languages (HDL). These are time-consuming and expensive procedures but, when implemented, the execution of such functions is much faster - in practice the logic circuit design is faster than an ALU several times used.

These principles were largely worked on in the design. The experience derived from that design was also the reason for introducing the concept of custom designed CPU.

Second: A microprogrammed machine is one in which a coherent sequence of micro-instruction is needed to execute various commands required by the machine. This technique was first introduced by Wilkes in 1950, as a structural approach to the random control logic in a computer. Microprogramming is now also used because it improves flexibility, performance, and LSI utilisation.

User **microprogrammed CPU**'s were very "fashionable" in the late 70s, while in the 80s the concept was introduced of configuring the CPU with the use of bit-slice chips.

We think the main reason that the idea did not survive was that an extremely high under-standing of CPU working philosophy was needed for the microprogramming and bit-slice design.

Third: Reconstruction of the **computer or computer system** level was introduced in the early 60s by Estrin and others. They proposed, or defined, a variable structure computer consisting of a fixed structure, a general purpose computer, an inventory of restructurable hardware (consisting of new devices or circuits), and a supervisory control unit. Many of their concepts are now being implemented in configurationable computing by the use of the PLC.

We found that the easiest way to combine all the three aspects of the reconfigurability was to introduce the in-system programmable logic circuit in order to do the design of a complete CPU or computer by the use of hardware description language (Abel-HDL). This approach is used to teach basic concepts about the structure and architecture of the computer in the first year of computer science studies. It is also used for final year students who are learning the use of programmable logic circuits. However, there are many possibilities concerning the level at which to start the (re)configuration of the computer.

The paper is divided into three parts. After discussing the concept of reprogrammability in the first part, in the second part, in Section 2.2, we propose an architecture of the CPU that can be fitted into a programmable logic circuit. Then in Section 2.3, we analyse which operations could be realised by the logic circuit; and, finally, there is a discussion of the performance of the custom designed microprocessor P4 – as we refer to our microprocessor. In the third part we give an example of the implementation of P4.

2.1.2. Designing the microprocessor P4.

Our initial version of the simple instruction set computer was taken from, which is presented as an example of a very simple model of CPU, but at the same time as a powerful one. It has a set of “primitive” instructions, which enable us to make other more “complex” computations. To this basic model were added some elementary instructions.

The common integer ALU usually has a set of arithmetic and logic functions, such as: addition, subtraction, shifting, AND, OR, negation, etc. What is crucial is the implementation of the complex arithmetical procedures, such as multiplication, and division. Beside these instructions, experienced programmers often choose the bit, slice, or byte manipulation instructions for optimization.

The microprocessor instruction set:

The microprocessor P4 is a classic von Neumann computer without I/O facilities. It has a set of instructions I, which in our case is:

```
I =
{LoaD_Accumulator_from_memory,
 LDA;
Store_Accumulator_to_memory,
 STA;
ADD_operand_to_accumulator,
 ADD;
BRAnch_unconditionally, BRA;
BRanch_on C_bit_set, BRC;
MiRRor_the_accumulator_bits,
 MRR;
SHift_Left_accumulator, SHL;
SHift_Right_accumulator, SHR}.
```

The instructions are organized as follows: four bits for operation code, and four bits for operand. The operand is taken as the address in LDA, STA, BRA, and BRC while in ADD, and shifts (SHL, SHR) are the data to either add, or to fill. There are 32 addressable memory locations, the first 16 locations being used for code, and the second 16 for data. The locations are four bit wide. The P4 offers four basic addressing modes: register, immediate, relative, and absolute.

The capital letters are used in the same way as the mnemonics for the assembly language. The instruction MRR was introduced just as an example of the specially designed instruction. For such instructions we would need about 20 assembly instructions (for an 8-bit accumulator) when programmed in the 80X86 microprocessor. The realisation of the MRR instruction will be explained later on.

For any new instructions which we would like to add to our initial set, we have to do the translation from formal description to HDL. The process is very similar to the on-line loading of the program into the RAM, which is then processed by the CPU; in the microprocessor with PLC we download the logic cell array (LCA) configuration memory. The functional circuits and data paths then process the logic of the user-defined instructions. The configuration data file, which defines the function and interconnection within a LCA, is loaded either bit-serially or byte-parallel with the configuration clock, which is (usually) slower than the system clock.

Table 2.1. Different sizes of chips in relation to the P4 performances.

chip type	MACH 111	MACH 211	MACH 5 - 512
Pins; I/O cells	44; 32	44; 32	352; 256**
Macro cells; FF	32; 32	64; 64	512; 512
No. of terms	2*70 (52 inputs)	4*64 (52 inputs)	> 8*64 (72 inputs)
CPU size in bits*	1-3	4-5	8-16

*) it is supposed that all registers have at least the dimension indicated.
**) the greatest chip which still enables parallel access to I/O data.

Table 2.2. Final statistic report for P4 from MACH211.

name	bit 4	bit 3	bit 2	bit 1	bit 0	name	bit 4	bit 3	bit 2	bit 1	bit 0
MAR	7/7*	8/8	8/8	8/8	8/8	IR	-	2/6	2/6	2/6	2/6
ACC	-	14/12	14/12	14/12	12/10	carry	8/9	3/3	3/3	1/2	"0"
IAR	5/11	7/11	7/10	7/9	7/8	Q	3/7	3/6	4/7	4/8	-

*) The first value means the number of terms, while the second is the maximal number of variables in the terms.

The configuration capabilities for serial downloading programmable logic circuits of a Vantis' MACH series are shown in Tab. 1.

Microprocessor hardware:

We begin our design with an initial configuration, and then successively enlarge it to suit the PLC being used. We suggest a simple set of instructions: the prime aim being to ensure the simplest instruction coding, and a smaller integrated circuit area.

The architecture of the P4 is simple; below are the registers we have used:

- program counter, or instruction address register, IAR,
- accumulator, ACC,
- memory address register, MAR,
- memory bi-directional data port (register), DATA,
- instruction register, IR,
- status flag register, Carrybit (Cb), and
- control register
  (Qi /internal states of the automaton,
  WE / write enable to EEPROM,
  UPMEM /enable upper part of the memory for data access,
  OE / enable access to EEPROM).

The microprocessor P4 complexity can be greater when larger programmable logic integrated components are used.

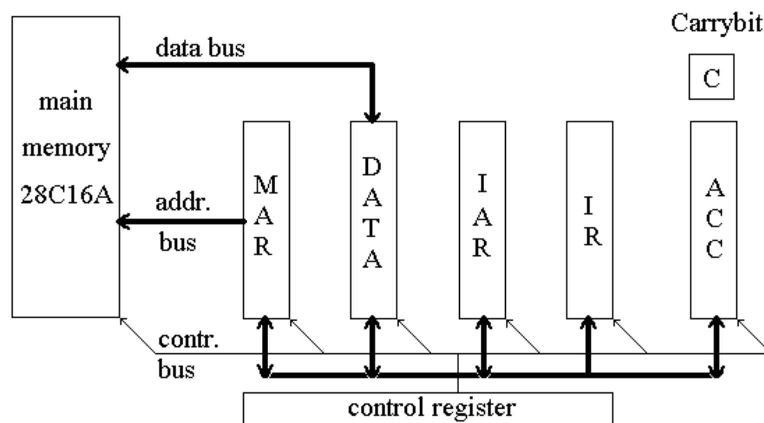


Figure. 2.1. The block scheme of a microprocessor P4. The data are a bi-directional port. The registers are 4-bit wide, except for MAR and IAR which have one bit more, because of segment memory addressing. The quantity of bits in the control register depends upon the number of states. For the main memory EEPROM is used with *read* and *write* capabilities.

```
system P4;
register ACC[Cbit, 3..0],
   MAR[4..0], IAR[4..0],
   IR[3..0], DATA[3..0];
states
"Instruction fetch states
   ADS: MAR <- IAR;
        IAR <- IAR+1; -> IFT;
   IFT: IR <- DATA; -> IFT1;
   IFT1: MAR <- IAR;
        IAR <- IAR+1; -> DEC;
"Instruction decoding state
   DEC: ?IR:
     #1 -> LDA;
     #3 -> STA;
     #2 -> ADD;
     #0 -> BRA;
     #4 -> BCS;
     #6 -> MRR;
     #5 -> SHL;
     #7 -> SHR;
"Instruction execution states
"Load to accumulator
   LDA: MAR[3..0] <- DATA;
     -> LDA1;
   LDA1:
     ACC[Cbit,3..0] <- DATA;
     -> ADS;
"Store accumulator
   STA: MAR[3..0] <- DATA;
     -> STA1;
   STA1: DATA <- ACC[3..0];
```

```
      -> ADS;
"Add value to accumulator
   ADD: ACC[3..0] <- ACC[3..0]
      + DATA[3..0]; -> ADS;
"Unconditional branch
   BRA: IAR[3..0] <- DATA;
      -> ADS;
"Branch if Carry bit set
   BCS: ?Cbit:
      #1: IAR[3..0] <- DATA;
          -> ADS;
      #0: -> ADS;
"Mirroring accumulator's bits
   MRR: ?DATA[0]:
      #1: ACC[3..0] <-
          'ACC[0..3]; -> ADS;
      #0: ACC[3..0] <-
          ACC[0..3]; -> ADS;
"Shift left
   SHL: ACC[3..0] <-
      ACC[2..0,DATA[0]; -> ADS;
"Shift right
   SHR: ACC[3..0] <-
      ACC[DATA[0],3..1];
      -> ADS;
end System;
```

Figure 2. The register transfer description of microprocessor P4. The operator "`<-`" can be simply expressed by "`:=`", while the "`->`" is indicated by the `goto` statement, and ? with `if` statement in Abel-HDL.

We can demonstrate how to design a custom oriented microprocessor through the use of the Abel-HDL. The block scheme in Fig. 2.1. is described in full detail in the Abel-HDL program list which can be obtained from. For those who are not familiar with Abel-HDL we introduce the description of register transfer of P4, which shows how a PLC should be used for the user programmable CPU logic functions. The finite state machine description gathers and edits conditions for terminal connection, register transfer, and state translation operations. The description of the machine is presented in Fig. 2.2, and in Tab. 2.2 the final statistic report is given.

2.1.3. Realisation of the user configured microprocessor P4

Traditionally, the algorithms are imple-mented by using general purpose program-mable microprocessors or coprocessors for low rate applications. We can use a special-purpose designed microprocessor if we introduce the FPGA, or any other in-system programmable logic circuits (PLC), and make our own microprocessor (P4). Furthermore, when the design is time critical, the PLC may then be a better solution. An in-system PLC can also be reconfigured in real time, and used from one application to another, i.e. from one function it can be reprogrammed to another simply by down-loading the new configuration file.

The EEPROM based in-system PLC is well suited for the functions of manipulating bit, byte, slice (subword) or word, such as:

a. set/reset a bit in a byte, or word,
b. moving or swapping bytes, or slices inside the word, such as instruction MRR,
c. coding and decoding from binary to BCD or ASCII, and vice versa,
d. extraction or rotation of a byte, or slice in the desired form, etc.
e. arithmetic and logic operations on the byte, or slice inside the word,
f. user defined processor with special purpose operations (neural networks, cellular automata machine, etc).

We will soon notice that the operators "*" and "/" are not to be used. The reason is simple: standard CPU with an additional circuit, or microprogram does the multiplication. In our case the entire PLC could be used only for realising one multiplication, which is not a realistic expectation.

And, in addition: each of the listed functions with two inputs can be (theoretically) realized by 2 levels of logic gates, which means that for different functions the execution time is the same.

If we use the Abel-HDL, then all the functions mentioned have to be made within one clock period. We also have a kind of structure - let us say an “array_of_bits” which enables us to manipulate with bits, in the way

that the microprocessor does with bytes, or words.

As an *example* of the use of data type "array_of_bits", we describe the function of the instruction MRR in the P4, which exchanges the bits 3 with 0; 2 with 1; 1 with 2, and 0 with 3. If we use the syntax of Abel-HDL we then get :

```
//comment: ACC is defined
as //set in Abel-HDL
ACC[3..0] = ACC3..ACC0; //...
equations
 ACC0:=(!D0.pin & ACC3) #
 (D0.pin & !ACC3);
 (!D0.pin & ACC2) # (D0.pin &
  !ACC2);
 (!D0.pin & ACC1) # (D0.pin &
  !ACC1);
 (!D0.pin & ACC0) # (D0.pin &
  !ACC0);
```

a. Each new instruction means a new state, and
b. a new additional conjunction is established arround the logic of the accumulator (ACC).
c. The complexity of the microprocessor depends upon the capacity of the PLC. The relation can be seen in Tab. 1.
d. We have to realize separately the adder, the incremental logic, and the carry bits' logic circuit, so that they can be properly fitted into the PLC.

Note: We found that the logic for addition in the sentence

```
ACC := ACC + DATA;
```

was extremely large. The carry bits were not taken sequentially, but were evaluated for each level separately. We avoid this problem by forcing the Abel-HDL to take it serially, and to keep the internal node of carry bits:

```
// defining a set
Carry = carry[3..0];
DATA = data[3..0];
//....
```

When we define the structure ACC we can simply write the last equation statements as:

```
ACC[3..0]:=
 (!D0.pin & ACC3[0..3]) #
 (D0.pin & !ACC[0..3];
//Very simple
```

The definition of type "array_of_bits" is identical to the description of the *set* in Abel-HDL and this is certainly not by chance. If we compare the above equation statement in Abel-HDL with C++ we find that in the first case the statement is performed concurrently, in the second (using C++) it is not. Therefore we greatly prefer Abel-HDL to C++ for modelling hardware.

2.1.4. Discussion

From the design of P4 we have acquired some experiences, which will be listed here.

```
equations
Carry = ACC & DATA + ACC &
Carry + Carry & DATA;
// in the state diagram
ACC := ACC $ DATA $ Carry;
```

Although this evaluation of the addition is time consuming, it does save a great deal of logic circuits.

e. The additional bit in the register takes another macro cell inside the PLC.
f. Some reductions of the control logic are possible, if we introduce smarter state coding.
g. A reduction in the number of pins is possible, since the I/O blocks are not needed for internal states, Qi, or the instruction register, IR.
h. We also do not always need flip-flops, as they force us to introduce the slower sequential logic, instead of the combinatorial one, which is faster. In the state diagram transition we have to add the statements:

```
equations
ACC := ACC.fb;
```

```
IAR := IAR.fb;
//etc
```

if we wish to keep the values of the registers unaltered during the state transitions of the control automaton.

2.1.5. Testing of P4

We made the experimental program and downloaded it into the EEPROM circuit.

```
START:    LDA  $1
ADDR:     ADD  #1
          STA  $0
          BCS  NEXT
          BRA  ADDR
NEXT:     ADD  #2
          SHL  #0
          STA  $3
          BRA  START
```

Figure 3. Sample program list for P4.

We also downloaded the MACH211 with the logic of P4, which we first translated from the Abel-HDL source file to the JEDEC file. The test circuit is shown in Fig. 5.

2.1.6. Implementation of P4

The concept of using the (re)configurable CPU for the realization of cellular automata machine (CAM) is very attractive, and this was why we chose it as an example of implementation. We usually find the realisation of CAM with FPGA integrated circuit, while our method is based on the in-system (re)programmable EPLD, and on the rule inside the cells (re)configured into the logic circuit. We find it best to use EPLD instead of FPGA because:

Table 2.3. Number of (128 * 128) cells in CAM, integrated circuits, and I/O pins.

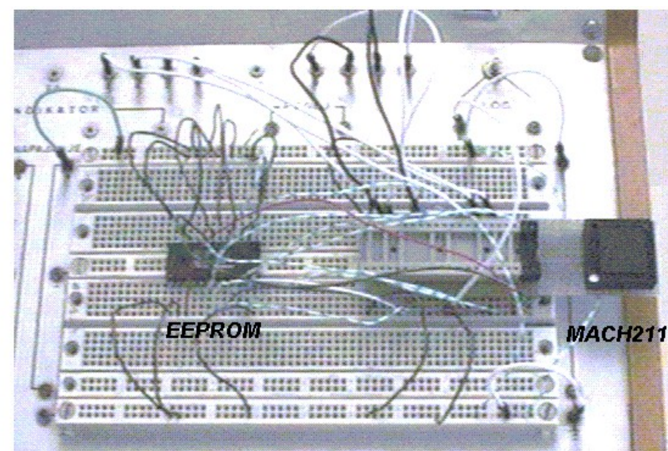
No. of cells in IC	1*1	3*3	4*4	6*6	8*8	14*14	11*23
No. of IC per CAM	16384	1821	1024	456	256	84	65
No of I/O pins	9/1	25/9	36/16	64/36	100/64	256/196	325/253

- We are not limited to 4(5) inputs from neighbouring cells to the central cell which has usually 1(2) FF. The design uses the blocks of central cells.
- We can realise any type of rule which performs its functional transformation in either one clock or a set of clock periods.

The design uses blocks of central cells with its neighbourhoods, rather than a single cell. The blocks are not overlapping. But we have first to take into account some consideration which are as follows:

a. We can look at CAM as a CPU with only one instruction, and we shall refer to such a (re)configurable processor as *a single instruction CPU*. The fact that in this case we do not use a program and data memory is very useful as it enables us to make a node with only one chip, i.e. with the CPU.

b. As the CPU enables more computation than for only one central cell with its eight neighbourhood, cells we therefore use blocks with more central cells, which means that more central cells are processed at the same time, in parallel. It depends on the number of I/O pins of the integrated circuit, how great the blocks could be. In Fig. 2.4.a. we see a single cell, while in Fig. 2.4.b. there is an example of a block of 3*3 central cells, with its 16 corresponding neighbouring cells. The cell rule can be composed either of one, or a sequential set of function transformations. If



necessary, we can also insert non-operate states.

c. Realisation of the CAM array is very simple, as we are connecting the chips in rows and columns on the board. The outputs are available from the pins of each chip separately. The synhronisation is done with the common clock connection, i.e. the signal from one clock generator is distributed to all the clock inputs.

Figure 2.5. Prototype board with the tested microprocessor P4.

Connectivity in the cell array is fixed and depends only upon the magnitude of the programmable circuit. The array of cells inside CAM is composed of:

- an array of cells inside the programmable circuit, a block, and
- an array of in-system programmable circuits, a module.

We can realise different sizes of (square or rectangle) blocks, e.g. 1*1, 3*3, 4*4, 6*6, 8*8, 14*14, 11*23 or greater. The number of chips for different realisation of the CAM array of 128*128 cells is given in Tab. 2.3.

The first column data are not within real possibilities; in the last column the expectations are slightly more realistic, it is not unusual to put 65 chips (9 sq. cm each) on a single board.

d. The CAM operations can be downloaded to all the chips in parallel or serial way, in real time. The serial strings of down-loaded data are computed by the program. We fill all chips with same data in the same time, and then fixed them by the programming signal "on". This in-system programming is very adequate as we can (re)program the CAM in real-time (in a few thousandths of seconds). By executing the sequence of clocks, the CAM array also moves sequentially, step by step, to its final position.

2.1.7. Conclusion

In the future, we can expect a rapid growth of telecommunications, and a consequent in-crease in the speed of computer networks, all of which will make possible the processing of live pictures, speech recognition, and hand-written character recognition, 3D graphics, etc. There will be many operations, for hand-ling bits, or bytes, in which either the user programmable microprocessor, or cellular automata machine (CAM), could be effectively used.

We have also to take into con-sideration the fact that many of the designing tools - i.e. program packages for the initial experimental work - are available free of charge.

We should like to thank the students who helped to make it possible for us to design the microprocessor P4 .

2. 2. Primer: MOVE mikroračunalnik

2.2.1. Uvod

Velika izbira programabilnih logičnih vezij, na primer proizvajalcev Xilinx in Altera, kot tudi množica uporabniku prijaznih programskih paketov za HDL (Hardware Description Language) kodiranje, sintezo in preizkušanje, omogočata načrtovanje in realizacijo celotnega procesorskega sistema na samo enem čipu. S HDL lahko enostavno opišemo tako podatkovno in kontrolno enoto ter periferne krmilnike, kateri omogočajo preizkus pravilnosti delovanja procesorja.

2.2.2. MOVE mikroprocesor

Ideja ukazne arhitekture MOVE mikroprocesorja je v tem, da je večina ukazov v visokih programskih jezikih prireditvenih (glej Tab. 2.4.)

Tabela 2.4. Dinamična frekvenca visoko programskih operacij.

Type of operations	Dynamic occurance [%]
assign	45 (C) ; 38 (Pascal)
loop	5; 3
if	29; 43
goto	-; 3
call	15; 12

Prireditveni izraz v jeziku C++:

A = B;

Je običajno »kombiniran« z aritmetično oziroma logično operacijo.

A = A + B;

Gornji izraz bi lahko prevedli v zbirnik procesorja RISC arhitekture takole:

»move« to register a from label A;
»move« to register b from label B;
»move« the sum of registers a and b
»move« from register a to label A;

Lahko zaključimo, da je v računalniku najpogostejša operacija »move« iz ene v drugo lokacijo (register). Velikokrat je operacija med registri povezana s premikanjem vsebine teh registrov. S tem vedenjem lahko začnemo definirati arhitekturo MOVE mikoprocesorja, katerega ukazi so vsi tipa »move«, torej ne potrebujemo operacijske kode.

Če izraz v jeziku C++:

B = A * 4 + 5;

prevedemo z Visual C++ 6.0 prevajalnikom dobimo:

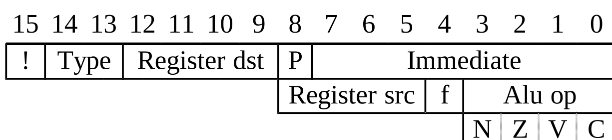
```
mov    ecx, A
lea    edx, (ecx*4 + 5)
mov    B, edx
```

Eksplicitnih operacij množenja in seštevanja ni; vsi ukazi so »move« tipa. Izračun izraza se izvede v ukazu lea (load effective address), pri čemer operaciji nista del operacijske kode, temveč del operandnega polja. Pri Move mikroprocesorju smo ALU operacijo skrili v operandno polje, kar ima še eno prednost. Navadno imajo procesorji za izračun naslova in izvrševanje dve ali več ALU, v našem primeru, pa to lahko opravimo samo z eno enoto.

2.2.3. Arhitektura MOVE mikroprocesorja

Že v prejšnjem razdelku smo namignili na RISC arhitekturo, torej moramo zagotoviti najpreprostejše kodiranje ukazov, ukaze enake velikosti in ne nazadnje čim manjšo porabo integriranega vezja.

Arhitektura MOVE mikroprocesorja je 16-bitni RISC. Sl.2.6. prikazuje ukazni format:



Slika 2.6. Format ukazov MOVE procesorja.

Posamezna polja v formatu pomenijo sledeče:

!	ob končanju ukaza se ne upošteva stanje vhoda INT;
register dst	biti IR[12:9] so naslov registra ponora, naslavljamo lahko 16 registrov;
type	00, format 1, biti IR[7:0] so takojšnji operand;
p(art)	bit pove, ali gre 8 bitna vrednost v spodnji (L, low) ali zgornji (H, high) del registra;
type	01, 10, 11, format 2:
register src	biti IR[8:5] so naslov registra izvora, naslavljamo lahko 16 registrov;
f(lag) test	pove ali spodnji 4 biti pomenijo alu operacijo (0), oziroma test zastavic in od tega odvisno izvršitev operacije. alu op(eration)artitmetično / logične operacije v ALE /iz nabora operatorjev jezika C med izvornim in ponornim registrom;
N,Z,V,C	zastavice s katerimi testiramo register stanja; N – vrednost negativna (negative) , Z – vrednost nič (zero), V – preliv (overflow), C – prenos (carry);

Operacije, ki se izvedejo odvisno od tipa, vidimo v Tab. 2.5:

Tabela 2.5. Tipi prenosov med operandi:

Tip	Opis
00	register dst ← IR[7:0] ≡ vrednost
01	register dst ← register dst op register src
10	register dst ← M(register dst op register src)
11	M(register dst) ← register src

Naslova registrov sta določena s polji: IR[12:9] za "register dst" in IR[8:5] za "register src".

Mikroprocesor ima sledeče 16-bitne sistemske registre:

MAR :	naslovni register;
MDR :	podatkovni register
IR :	ukazni register;
SR:	register stanja (N,Z,V,C);

in delovne registre:

REG0 : programski števec
REG1 : naslov vračanja iz prekinitev;
REG2 : delovni register; skladovni kazalec;
…
REG15 : delovni register; skladovni kazalec;

Registrski niz je organiziran kot 16 bitni niz registrov z dvojnim (dual-port) dostopom, dvojnimi podatkovnimi in naslovnimi linijami za branje in s po eno za pisanje.

Obnašanje mikroprogramskega avtomata procesorja MOVE podaja RTL (Register Transfer Language) diagram:

IF: IR ← M[MAR]; PC ← PC + 1;
Goto ID;
ID: **case** IR[14:13] **of**
When »00« =>
if (IR[8] = '1') **then** RegDst ← IR[7:0] & ^h00;
Else RegDst ← ^h00 & IR[7:0];
End if;
When »01« => RegDst ← RegDst op RegSrc; MAR ← PC;
If RegD = »0000« **then Goto** EXE2;
Else Goto IF;
End if;
When »10« => MAR ← RegDst op RegSrc; **Goto** WB;
When »11« => MAR ← RegDst;
MDR ← Reg Src op RegDst; **Goto** MEM;
End case;
WB: RegD ← M[MAR]; MAR ← PC;
If (regDst = 0) **then Goto** EXE2;
Else Goto IF;
End if;
MEM: M[MAR] ← MDR; MAR ← PC; **Goto** IF;
EXE2: MAR ← PC; **Goto** IF;

Na zgornjem RTL diagramu prekinitve še niso podprte, vendar bomo kljub temu podali idejo izvedbe prekinitve. Ob koncu ukaza se opazuje stanje vhoda int ter bita 15 v ukaznem registru. Če je izpolnjen pogoj za prekinitev, se namesto vsebine programskega števca v naslovni register prenese fiksni prekinitveni vektor, vsebina programskega števca pa v register 1 ter nadaljuje izvajanje, kot da se ni nič zgodilo. Programer mora poskrbeti, da je na naslovu na katerega kaže prekinitveni vektor, ukaz ki poskrbi, da se v programski števec prenese naslov prekinitvenega servisnega programa (PSP). Potrebno je tudi paziti, da imajo vsi ukazi v PSP onemogočene prekinitve, saj bi v primeru vnovične ali pa celo iste preknitve izgubili staro vsebino programskga števca. Vrnitev iz prekinitve se izvede preprosto s prenosom vsebine registra 1 v register 0 (programski števec).

2.2.4. Nabor ukazov procesorja MOVE

Ker so vsi ukazi v računalniku MOVE vrste »prenesi« (move) potem težko govorimo o naboru. Ukazi so različni le po tipu operatorja 'op', ki nastopa med REGdst in

REGsrc in "!". Torej ne potrebujemo v ukazu bite za "ukazno kodo", ki je vselej "move". Zato ne bomo pisali »move r1, r2«, ampak samo »r1, r2«! Nekaj bitov ukazne kode pa vemo, da vseeno potrebujemo, in sicer za klicaj, označevanje tipa operandov in katere operacije 'op' izvedemo med njimi.

Za primerjavo podajmo nekaj ukazov mikroprocesorja družine 80X86 in ustrezni zapis ukaza procesorja MOVE:

Tabela 2.6. Primeri preprostih ukazov zapisanih v C++..

ukaz 80X86	ukaz MOVE	tip	Opis v sintaksi jezika C
mov cl, 33	r3 ← r4;	'01'	reg3 = vred, L;
mov ax, [bx+ax]	r1 ← [r2+r1];	'10'	reg1 = M(reg2 + ;
mov [bx], dx	[r2] ← r4;	'11'	M(reg1) = reg4;
add ax, bx	r1 ← r1 + r2;	'01'	reg1 = reg1 + reg2;
neg ax	r1 ← 'r1;	'01'	reg1 = 'reg1;
sll bx, 1	r2 ← r2 + r2;	'01'	reg2 = reg2 << 1;
sub ax, cx	r1 ← r1 - r3;	'01'	reg1 = reg1 - reg3;
jmp [bx]	r7 ← r2;	'00'	goto PC ≡ reg2;
jcs [bx]	!r7 ← r2;	'00'	if C=1 then reg7 [1)] = reg2;
jsr [bx]	!r6 ← r7; r7 ← r2;	'01' '11'	reg 6 = reg7; reg 7 = reg2;
rts	r7 ← r6; [2)]	'01'	reg 7 = reg6;

Opomba:

1) Programski števec je register 7.

2) Naslov vračanja iz podprograma se shrani v reg 6, ali kak drug neuporabljen register.

Z ukazi iz Tab. 2.6. res nimamo posebnih težav. Kaj pa z ukazi kot so: int, rti, jsr itd? Tudi te lahko nadomestimo z ukazi procesorja MOVE. Kot vemo iz arhitekture procesorja MOVE, določimo poseben format ukaza, ki ga bomo predznačili z "!" in pomeni, da "ukaz opazuje pogojni bit C, oziroma ukaza ni mogoče prekiniti ob prekinitvi". Pokazalo se bo, da za en ukaz v 80X86 rabimo več ukazov procesorja MOVE, ki bodo vsi predznačeni s klicajem (!), Tab. 2.7. Pozor, če klicaj uporabimo pred ukazom, ki ima REGdst = REG7, je to pogojni skok, ki se izvede le, ko je C = 1.

Tabela 2.7. Sistemski ukazi sklada in kontrolni ukazi.

ukaz 80X86	ukaz MOVE	tip	opis
pop ax	!r1, [r4] !r4, r4+1	'10' '01'	reg1 = M(reg4 [3)]); reg4 = reg4 + 1;
push bx	!r4, r4 -1 ![r4], r2	'01' '11'	reg4 = reg4 – 1; M(reg4) = reg2;
int 0	!r0, r7 r7, [0 [4)]]	'01' '10'	reg0 = reg7; reg7 = M(reg5);
rti	[5)] r7, r0	'01'	reg 7 = reg0;

Opombe:

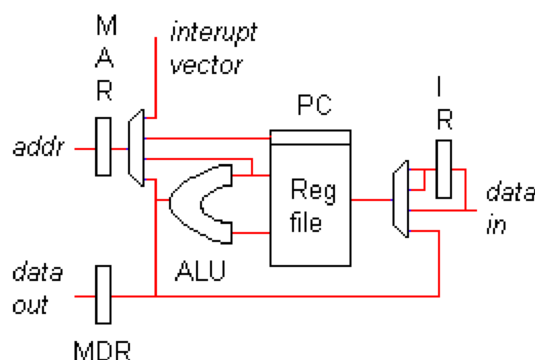
3) Registri 1, 2, 3 in 4 so lahko kazalci nesistemskega sklada.

4) Prekinitveni vektor je en sam na lokaciji "000", naslov vračanja se shrani v reg 0!

5) Tu ni klicaja, sicer se ukaz izvede kot "if C=1 then reg7 ← reg0".

2.2.5. Organizacija MOVE mikroprocesorja

Kljub preprostemu RTL diagramu, smo VHDL kodiranje izvedli na bolj strukturalen način, da bi lahko uporabili razne makroje, specifične za Spartan-II čip oziroma za Xilinx čipe v splošnem. Poleg tega smo s takim pristopom hoteli podati notranjo organizacijo mikroprocesorja in jo tako predstaviti preglednejše ter enostavnejše za vzdrževanje in nadgrajevanje. Podatkovno enoto MOVE podaja Sl.2.7.



Slika 2.7. Blok shema MOVE podatkovna enota

Podatkovno enoto krmili kontrolna enota, s katero skupaj tvorita procesorsko jedro MOVE. Preko naslovnega dekoderja smo procesorsko jedro povezali s 256 bajtov bralnega (ROM) in 128 bajtov bralno pisalnega (RAM) pomnilnika ter krmilnikom sedem segmentnega zaslona na tekoče kristale (LCD) in krmilnikom svetlečih diod (LED). Vse te komponente smo integrirali na enem samem čipu, pri čemer smo zasedi približno 40% vseh razpoložljivih resursov.
Natančnejšo analizo podaja Tab2.8:

Tabela 2.8. Povzetek porabljenih resursov

Selected Device :	2s100pq208-5
Number of Slices:	469 of 1200 39%
Number of Slice Flip Flops:	103 of 2400 4%
Number of 4 input LUTs:	712 of 2400 29%

Pregled poročil sinteze posameznih modulov pokaže, da podatkovna enota zavzema največji del logike, znotraj nje pa izstopa aritmetično logična enota. Pri realizaciji seštevalnika smo uporabili sposobnost Spartan-II čipa, da uporabi svoje štiri vhodne look up tabele (LUT) kot dvobitni polni seštevalnik oziroma odštevalnik, predmet možne optimizacije pa ostaja uporaba samo enega seštevalnika za seštevanje in odštevanje z ali brez vnaprejšnjega prenosa.

Registerski niz 16 x 16 bitov je organiziran kot dva RAM-a z ločenimi naslovi za pisanje in branje. Na prvi pogled se uporaba dveh RAM mogoče zdi potratna, vendar je potrebno upoštevati zgradbo Spartan-II čipa; vsak od CLB (Configurable logic block) je sestavljen iz dveh štiri vhodnih LUT ter dveh pomnilnih celic. LUT se lahko uporabi kot poljubna funkcija štirih spremenljivk, ali pa kot 16x1 bitni RAM. Za 16 registrov smo torej porabili 32 LUT (zaradi podvojitve), oziroma 16 CLB, kar je precej manj, kot če bi registre realizirali z pomnilnimi celicami CLB ter dvema

izhodnima 16-bitnima multipleksorjema. Primerjava obeh realizacij z dodanim programskim števcem je vidna v Tab. 2.9.

Tabela 2.9. Primerjava realizacij registerskega bloka:

Kategorija:	16 registrov	2 x RAM
# of Slices (1200):	285 (23%)	78 (6%)
# of Slice FF-s (2400):	256 (10%)	16 (0%)
# of 4 input LUTs:	359 (14%)	86 (3%)

Kontrolna enota je enostaven diagram prehajanja stanj, z zelo malo dekodiranja zaradi preprostega formata ukazov. S strukturalnim pristopom k realizaciji smo uspeli sorazmerno minimizirati porabljen prostor na čipu, vendar je hitrost postala kritična. Zaradi veliko nivojov logike, je maksimalna frekvenca padla na 25 MHz, kar napeljuje na misel, da bi morali uporabiti več registrov za hranjenje vmesnih rezultatov in se s tem približati klasični cevovodni organizaciji mikroprocesorja, kar verjetno bo naslednji razvojni korak.

2.2.6. Laboratorijsko testiranje

MOVE mirkoprocesor smo preizkusili s preprostim programom simulacije števca, na ploščici z vezjem. V vrsticah 0 do 7 je inicializacija programa, ki vsebuje izračun naslova LCD krmilnika (0x0180) ter skočnih naslovov, nato pa program preide v neskončno zanko. Znotraj neskončne zanke z vgnezdeno zanko izračunamo zakasnitev trajanja približno 1 sekunde, nato pa shranimo vrednost na naslov LCD krmilnika, kar povzroči prikaz na LCD prikazovalniku, ter jo povečamo za 1.

Programska koda:

```
0           mv ^R5, 1
1           mv _R6, $80
2           mv R5, R5 + R6
3           mv _R6, 10
4           mv _R8, 1
5           mv _R10, 8
6           mv _R11, 10
7           mv _R12, 12
8           mv _R3, $4B
9           mv ^R7, $4C
10          mv R3, R3 + R7
11          mv ^R4, 1
12          mv R4, R4 - R8
13          mv R0, R12, !Z
14          mv R3, R3 - R8
15          mv R0, R11 0100
16          mv [R5], R4
17          mv R6, R6 + R8
18          mv R0, R10
```

2.2.7. Emulacija in simulacija procesorja MOVE in zaključek

Arhitektura za procesor MOVE je napisana v jeziku VHDL. Prevedli smo jo s programskim paketom proizvajalca Insight, verzija 1.2., ki je dodan testni ploščici za vezja Xilinx Spartan-II (XC2S100-5 PQ208). Tako dobljeno logično vezje procesorja MOVE porabi približno polovico razpoložljive integrirane programabilne komponente.

Procesor MOVE je dober primer 16-bitnega procesorja, ki je primeren za enostavne vgrajene (embedded) aplikacije. Prirejen je enostavnemu programskemu jeziku, saj je njegova arhitektura posnema stavke visokega programskega jezika. preprost mehanizem prekinitve pa nam omogoča tako uporabo v realnem času kot v aplikacija z obsežno periferijo.

2.2.8. Dodatek:

Programsko kodo VHDL procesorja MOVE /nepreiskušena inačica/ bomo našli na straneh laboratorija, in sicer za procesor na naslovu lra-1.fri.uni-lj.si/move/move.vhd in za aritmetično-logično enoto na lra-1.fri.uni-lj.si/move/alu.vhd.