EB 168

Definition and High Level description of the C-processor

Master Thesis of:
Willem Jan Withagen
Departement of Electrical Engineering,
Digital Systems groep.
Eindhoven University of Technology.
November 1987 — December 1988.

Abstract

In this master thesis is reported on the structured design of a processor which is targetted to execute high level programming languages with more efficiency. The thesis encompasses:

- The definition of an instruction set.
- The developement of a software model that will executed the defined instruction set.
- The developement of a high level decomposition for a choosen architecture. The modules in this high level description are programmed with HHDL. (Silvar-Lisco)

The results of this project give an indication of the problems to be solved in near future before a working system can be created.
It contains two high level models which will execute the defined instruction set. Each of the models is written for a specific task. The software model will execute complete programs to test the performance of the system and its instruction set. The decomposition model will be used as a testing vehicle for future decompositions and possibly gate implementations.

Acknowledgements.

A place like a masters thesis report is a too a small place to thank the friends who have helped me during my (long) time at the university.

None the less would I like to say to every one:

Thank, Thank, Thank for having patience with me!
or **Dank, Dank, Dank voor al het geduld!**

The following is in dutch, since the quotations are typically dutch.
Natuurlijk heeft mijn favoriete schrijver Martin Toonder zijn stripfiguren voor deze gelegenheden wel wat passelijks laten zeggen.
Spreuken die we in het dagelijkse leven ook gebruiken:

- Als je begrijpt wat ik bedoel.... .
- Ik wist niet dat ik het in mij had.
- Had ik maar beter geluisterd.

Maar ook wat diepzinniger:

- Bedrijfsleven.
  'Recht is is kroms dat verbogen is,' hernam Super na een korte pauze. 'En daar heb ik verstand van, als zakenman zijnde.'
- Communicatie.
  'Uw antwoord klinkt als een klepel in een gebarsten klok van grote schoonheid,' sprak Ping Po.
- Voor Frank.
  Wanneer men zich in een voertuig zonder bodem bevindt moet men zeer hard lopen wanneer het rijden gaat, en dat deed heer Bommel dan ook.
- Voor de Prof. (onderwijs)
  'Als het anders niet is,' sprak heer Bommel. 'Mijn goede vader heeft mij tweeënzeventig meter boeken nagelaten. Volg me maar naar de bibliotheek, heer Trot. Er bestaan maar weining dingen waar iemand van mijn stand geen weet van heeft.'

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The history of the project.

As part of a project to get a better and more systematic grip on the design aspects of digital systems, a number of "test" design projects have been started at the Digital Systems group of the Department of Electrical Engineering of the Eindhoven University of Technology. The aim is to retrieve common design steps and methods in order to automate, or at least partly automate, these design steps.
The studies are focused on the higher levels of the designs, and do not bother about the last steps of finishing the design, the production of layout and masks. There are two reasons for these limitations. The restriction placed on the projects make them more or less technology independent. At a high level of the design, functional blocks are created which have very little relation yet with their physical implementation.
Current state-of-the-art design systems have speeded up the design process enormously: Time consuming items as drawing, documenting, and testing are already, or will in the near future, be handled in large amount by software systems. What remains for the human designer are the more creative aspects of the design: Developing the specifications and architecture of the circuit. Once the circuit has been broken down into pieces small enough to be handled by software, most of the designers work is done. The software programs can be items such as: State diagram translators, RAM/ROM generators, PLA generators, or even a "silicon compiler".
The design project described in this report is a follow up on the graduation work of F. Budzelaar [Bud88], coached by Prof. M.P.J. Stevens. This thesis involves the design of a microprocessor targeted to execution of high level languages, and in more detail the language "C"[Ker88].
My "inheritance" consisted of a master's thesis which gave a clear analysis of the language items to be implemented in the C-processor, a proposal for the first level of design and the decomposition of the first level into subblocks. Next to this contained the thesis discussions on: Processor design in general, a first global definition of the instructionset, communication between the modules, and low level machine architecture.

1.2 The aspects added in this thesis

The assignment for this thesis was to take the work of [Bud88] and check his analysis of the design for flaws (if there where any). And from there on continue with the design process starting at the highest level possible. In this thesis are described:

- A minor change in the functionality of the onboard register set. This results in a redraw the highest level of design.
- The bitwise description of the instructionset. The instructionset had to be fully defined before any work of greater detail could be done.
- A software model of the "C"-processor. Due to the complexity of designing the processor a distinction was made in executing mere code, and the functioning of the designed hardware. As a consequence of this divide and conquer methode, the software model would give information about functionality of the given instructionset, usefulness of the first step of decomposition, possible guidelines for future decompositions.
- Another advantage to consider was the possibility to forgo the timing relations and concentrate on the data path and data manipulation. A final profit was constituted by the test files. To test the software model, small code files had to be made, and these code files can also be used during designing and testing of the hardware realization.
- A high level description in HHDL (Silvar Lisco) of the decomposition was an obvious next step following the software model. On this level the software functions and procedures are translated into separate hardware modules. Each of these modules has its interfaces with the "calling" modules and/or a controller. The objective at this stage of implementation is to acquire a HHDL model that will execute the same "C-code" as was executed by the software model. No efforts are made to convert the serial program flow of the software model into parallel execution in hardware. Note that this does not imply that parallelism will not be possible or wanted. But at the current level this is not really considered a valid item.

Chapter 2

Minor changes to the registerset and the first level decomposition

2.1 "C"-processor language background

The "C"-processor is designed to execute the language "C" more efficiently. To show some of the properties of this concept a short introduction of the background principles is made before the "real" work is started.

2.1.1 A possible memory layout.

In fig 2.1 a possible subdivision is given. In this are clearly distinguishable:

- The Code space.
- The Stack space
- The Heap
- Unallocated memory.

It is obvious that this subdivision is not the only one possible. But it will serve the purpose of explaining some of the used principles and ideas.
The Code space can be enlarged, in which case the routines comprising the program can be seen. It is possible to look upon the main global body of a program as the first and entrance routine. It is possible for routines to contain a section with constants. These can be either user defined constants, or possibly construct a jump table. It has however to be fixed data.
When the execution of a routine is started, the routine will create a stackframe. In this frame (fig. 2.2) are located:

- Parameters

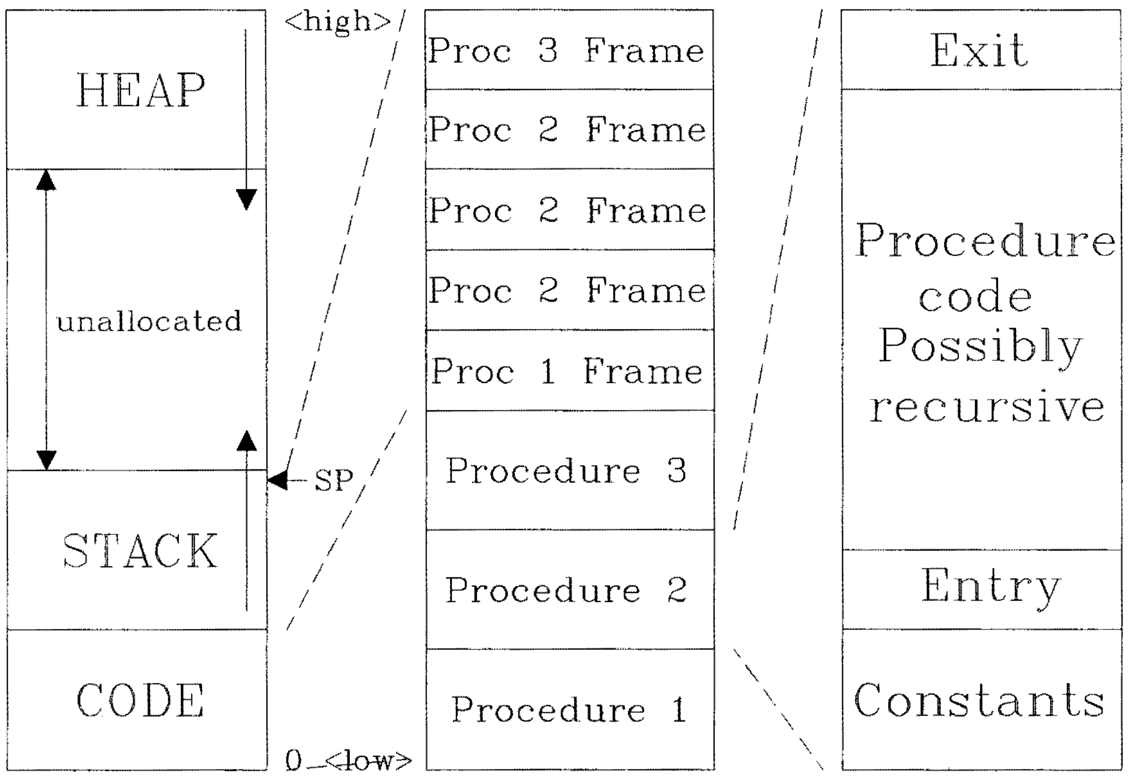

Figure 2.1: A possible memory layout used for reference.

- The status of the previous routine to be saved. (Here: The return address, and the old frame pointer.)
- Local variables.
- The dynamically used part of the stack, containing <TOS>.

The main routine of the program can be considered as a special type of routine. In the frame of this routine could also the global variables be declared. If a routine calls itself recursively, it will create a new frame for every new entrance of the routine.
Once a routine terminates it returns the Stack space occupied to the unallocated space. (Or the calling routine will do so.)
If we were to consider frames created by the programming language **PASCAL**, the above would hold. And the allocation of the parameter and local variable space would be simpler, since this could always be done in the function call of the routine to execute. It is however beyond the scope of this thesis to go too much into detail. For more details of **PASCAL** program construction we refer to [Pem82].
Although these considerations would give enough reason to call the design a stack-processor design, the name "C"-processor will be kept for historical reason.

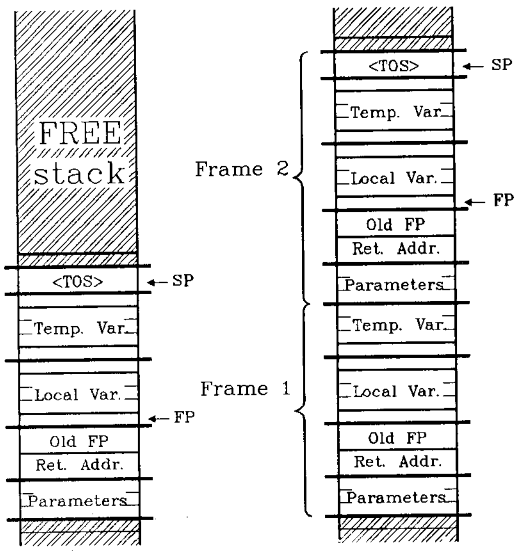

Figure 2.2: A frame on stack.

2.1.2 A possible speedup of the architecture.

In our design of the "C"-processor shall the concept of the stack frames (As shown in the above paragraph) be used as the key item of the design philosophy. The instructionset will contain operations which will handle data in the stackframe with optimal speed and with flexibility. The hardware design will facilitate these operations and will make the stack frame fast accessible.
To fulfil the first requirement the instructionset contains a set of P-instructions. (derived from the P(ascal)4 instructions), which are in all kinds of types and machines available)
To fulfil the second requirement the hardware will be equipped with a dedicated stack system. This system will manipulate a high speed onboard stack window. This window will always contain the last activated stackframe. As a consequence of the above assumption all operands to the P-instructions are within the system. And thus low access times can be guaranteed. The programmer could view the system as drawn in figure 2.3.

2.2 Changes in the registerset.

Although the design in [Bud88] complies with the above assumptions a few changes will be made. Most of these changes are not in the system, but in the functionallity of the designed parts.
The initial design has a TOS (Top of Stack) and a TOPS (Top of Parameter Stack). Both point to the same location, with one exception: When the parameters are loaded on the stack, the stackpointer points in the old frame, while the parameter stackpointer increments with every parameter loaded.
Since it is considered useful to be able to use the stack as a save location for temporary

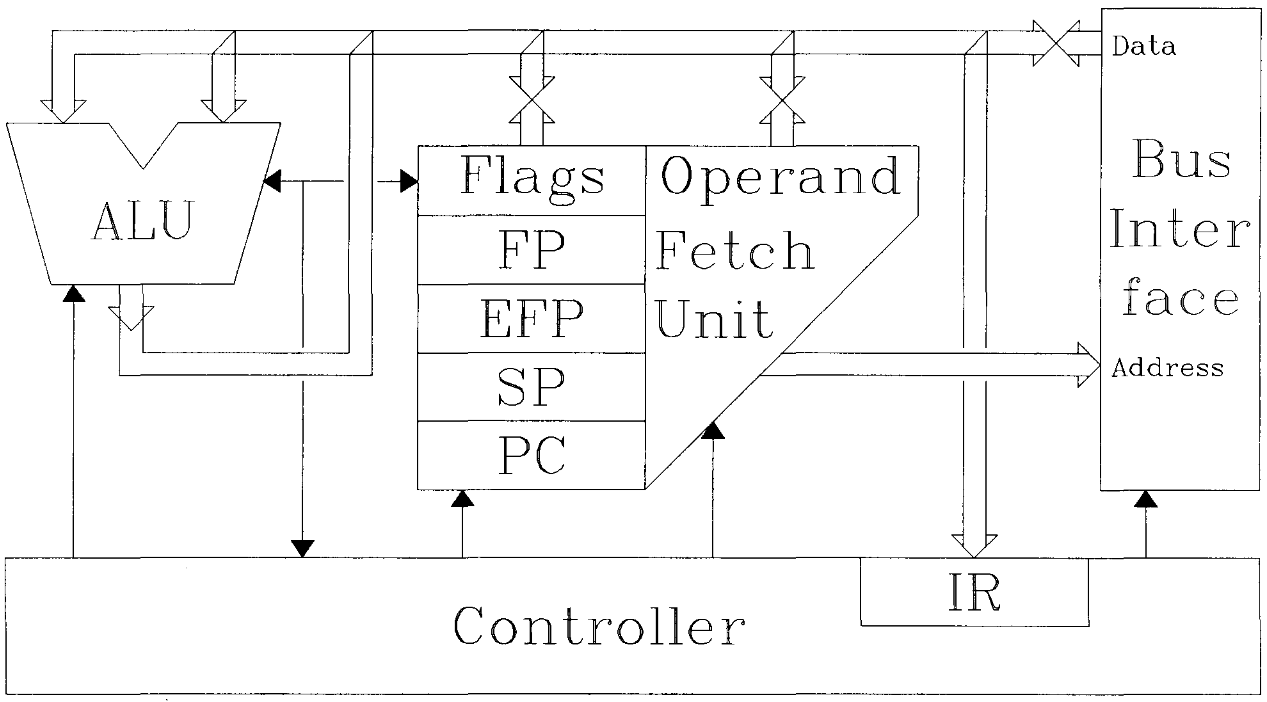

Figure 2.3: The processor for programmers.

results, and as a consequence of this a chance is made to the functionality of TOS and TOPS.
< TOS >will be a pointer that will always point to the Top of Stack (in future it will be called "SP"), TOPS will point in the current frame to the return address. A negative offset will give access to the parameters, a positive offset will return a value from the local variable space. When examining compiled **PASCAL** programs it is very often found that one of the general registers serves as a frame pointer. This is also due to the nature of the language **PASCAL**. The language "C" has the same implications and concerns at this point.
From the same analysis one can deduct that storing large quantities of data on the stack, will defeat the purpose of a cache for the stack. Due to these large quantities, too many reads and writes to main memory will occur and to processor will be slowed down. For these large quantities of data a special frame pointer is added (EFP = External Frame Pointer). This register can be used as an offset register in all addressing modes. But it is the programmers responsibility to maintain a correct value in this pointer. The pointer is not modified by calls or returns. It only changes on a process swap or if the programmer writes into the register.

The consequences of this change are major with respect to the instructionset, but since this was not yet rigidly defined no work is lost.

2.3 A new level 1 decomposition

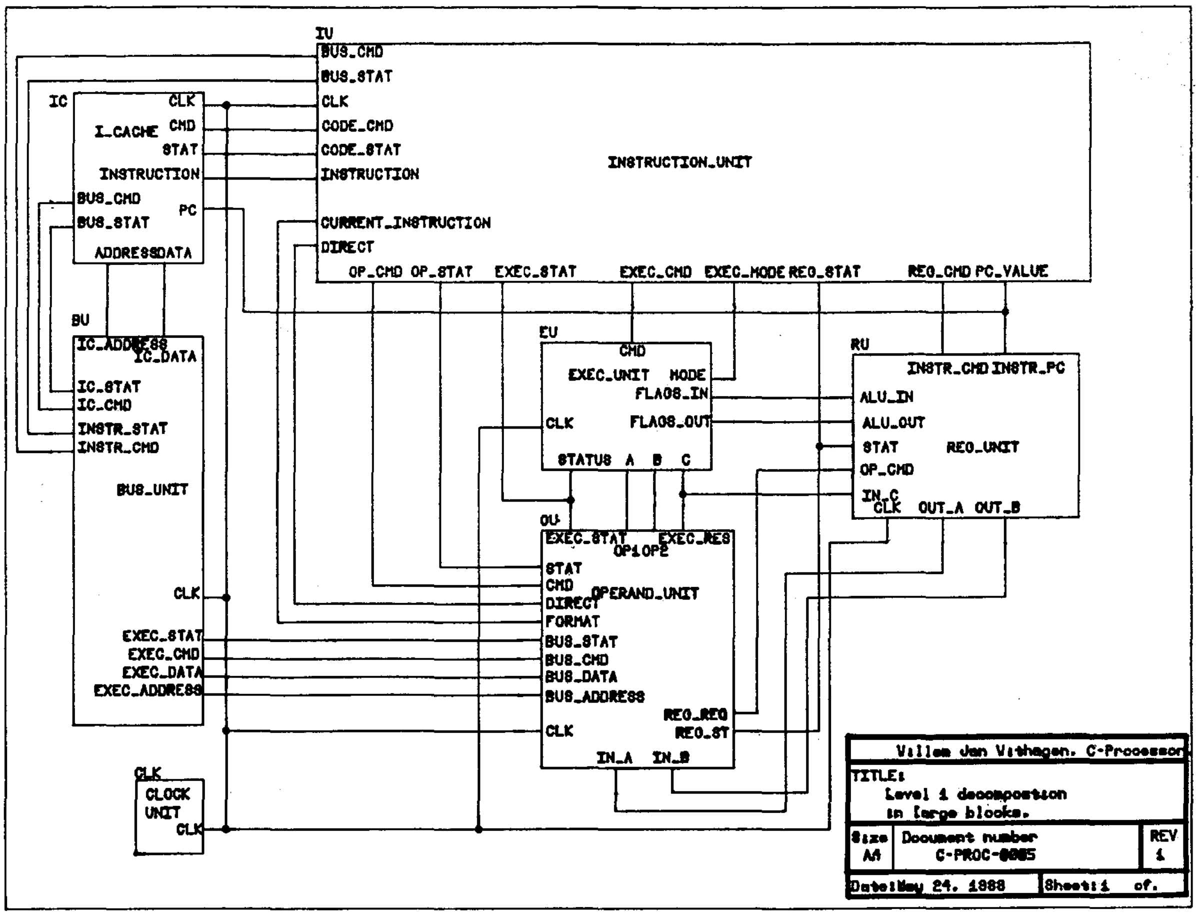

Figure 2.4: A first decomposition in global blocks.

The impacts on the design however are very modest. None the less is a slightly new architecture created. This architecture will make better use of the FP and EFP, see fig. 2.4 If one compares this architecture with the design from [Bud88] on page 41, it is clear that almost all submodules are still there, the names have changed a little. By the new design the data paths and command paths are given in greater detail and have a completely different structure, as are the connections.
For instance:

- Since instructions with two operands will be created, a double data path is used to connect all registers and the operand unit. This gives a possibility to calculate the addresses of both operands at the same time, given that other criteria allow for this parallelism.

- The program counter can be used for references during offset calculations, hence the path to the operand_unit.
- But as one can deduct from the design, the paths to and from the execution unit are all uni-directional. The A- and B-bus have six sources, the C-bus or the "result bus" has one master and six slaves. The flags register has two connections to the execution unit, one for either direction. Through these connections the values of the flags will be transported. (to and from the ALU)
- All modules (on this level) are centrally controlled by the instruction-unit. Every module interfaces with the instruction-unit through a set of handshake signals. These signals contain at least on of the following signals or busses:
  - A command bus, with a NOP command as one of the commands. Other commands can be either a plain start command or more complex commands which specify the action to perform.
  - A result or status bus. This bus has at least the following values: READY, BUSY. They indicate whether the unit is currently active. Other included commands can be request for the controller. If for instance extra data is required, this can be asked with a status like MORE or NEW_DATA.
  - An exception to the above scheme is the `regsiter_unit` This unit can also be controlled directly from the `operand_unit`. Although this is clearly an evasion from the "central controller" idea, it prevents extensive amounts of communication between the `instruction_unit` and the `regsiter_unit`.

2.3.1 A means of synchronizing controller(s) and/or slaves.

Since on forehand very little is know about timing and sequence relations, a tight (but flexible) mechanism is needed to keep all modules in synchronization. It would be possible to solve some of these problems, using a central clock, but this would force us to give timing estimates for the modules. In stead of the clocked solution, a choice is made for a completely asynchonous communication. In this way no information concerning timing relations will be left out. In future steps, however, it can prove possible to remove certain constrains in the operations. These decisions should then be made at the appropriate level and not on the current level, in which case they could stress the design to comply with some ad hoc rules.
The principal way of handshaking will be:

1) The slave unit is READY, and the command is NOP.
2) The controller issues a command (COMMAND), and the slave responds with setting it's status to BUSY.

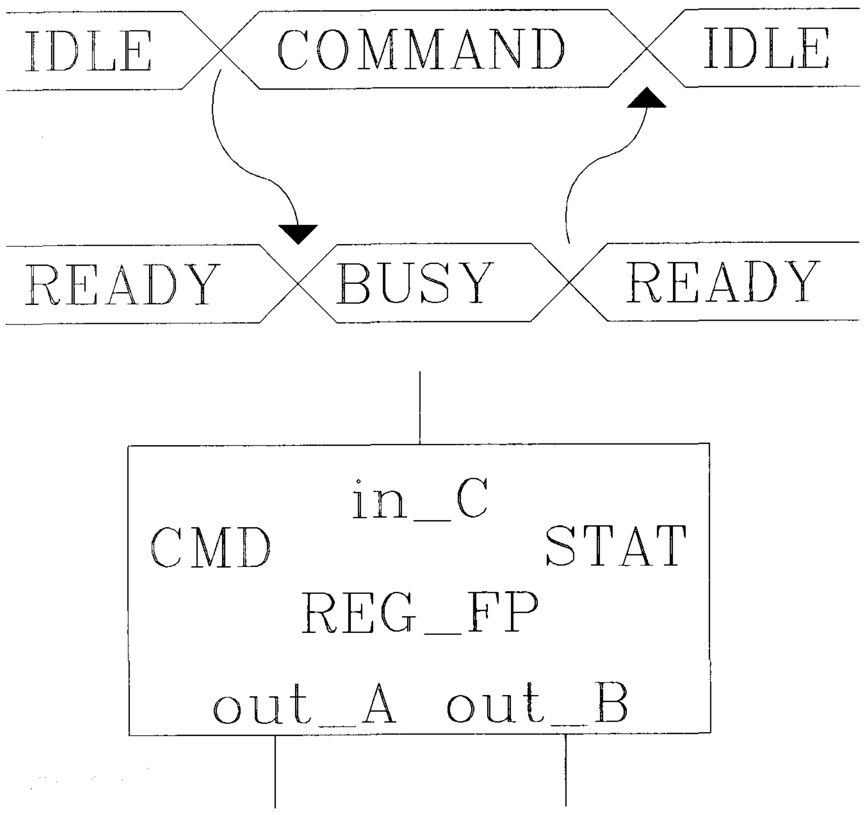

Figure 2.5: The basic handshake cycle.

3) The slave carries out the requested task. While in the meantime other modules could be scheduled, either in parallel or upon request of the slave unit currently observed.

4) Once the slave is done with its assignment, it clears the status to READY.

5) The controller acknowledges the READY by resetting the command to NOP.

Compare with figure 2.5.
If the slave module has a "request" for the controller, then the communication could be extended as in figure 2.6:

3a) The status changes to REQUEST. The slave could continue its operations, but it is not yet allowed to use the requested item.

3b) The controller responds with a NOP on the controlbus. This should be interpreted by the slave as: "The controller is currently not observing the statusbus of this unit".

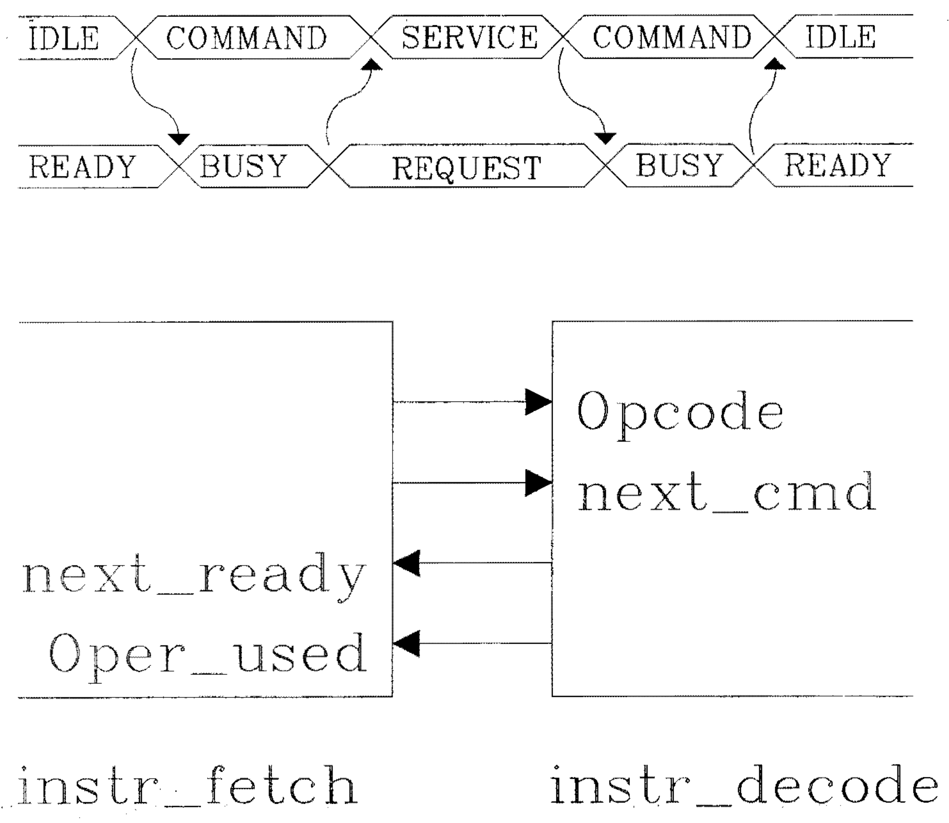

Figure 2.6: An extende handshake cycle.

3c) Once the controller is able to fulfil the request issued by the slave, the controller will change the controlbus to COMMAND to alert the slave that the operation asked for has been completed.

3d) The slave unit will acknowledge the completion of the request by setting its status back to BUSY.

Naturally there are many possible variations on this theme. The two examples above are considered the most elementary ones used in this design. They are, if so needed, adapted for each situation. But on the high level description a strong attempt is made to exercise a complete handshake mechanism on all interfaces.

2.4 What makes the design testable?

If testability needs to be considered on this level of the design it would be possible to state:“ I'm using a scan-design, so the design will be testable by definition.” However, simple observations show that on this level every module can be connected directly to I/O-pins of the chip by switching other modules into a transparent state. In this test state

there will be no changes in the flow of data. Or in other words: no datapath will have to change its direction of flow for the testing.
Switching a module into a transparent state will require a multiplexer and a demultiplexer per datapath, and in the overall design a small controller to perform the testing.
The problem with this approach is that there is no guarantee that the chip will be fully testable. Scan-design however will in its crude form guarantee that all modules are testable. The systems buses are the one exemption from this. Even in the current scan-designs one has to test busses with a different strategy.
The valid question to ask now is: "Which of the two options will be the cheaper one." The answer to this question at the moment is of very little concern, since it was already stated that scan-design is a possibility that is almost always possible, and for the other option the circuit needs no changes at this level of implementation.

2.5 What will the next step(s) be?

The decomposition has still left us five large items:

- The bus controller, a circuit which has to arbitrate memory requests from the operand unit, and the instruction cache unit. It also has to manipulate the data to be able to write BYTEs, WORDs and QUADs. The complexity of this will be fairly low, the next step will result in a definition of the datapath, registers and controller within the module.

- The instruction cache unit, the unit can have several types of implementation. However, all of these are on a rather primitive level. Hence the next decomposition step for this block will probably be port level.

- The instruction unit, this block is the major controller of the processor. It will consist of a micro programmed controller, or a state machine, or any combination of the two. The design can be straight forward.

- The execution unit, in this unit the actual data manipulation will take place. It will contain all logic to do the needed arithmetic and logic operations, most of these can be done by a "simple" combinatorial network. Multiply, divide, modulo are candidates for serialisation, since a 32/32 divide ain't nothing to program just straight into logic.

- As a consequence of this the execution block will contain a large and complex logic part, next to a simple controller with again a large logic network.

- The operand unit, this unit has to prepare the operands to go along with an instruction. The operands will be used by the execution unit. All different types of addressing modes are implemented in this module. The stack cache could also be fitted into this module. This module has to implement several subprocesses and will be fairly complex. This module will have to perform the actual fetching of the operands, whether they are in main core or in the stack cache. It has to translate operand sizes to their appropriate formats. It has to convert from signed to unsigned and vice versa.

Before a next decomposition step is taken, it is very desirable to have the instructionset of the "C"-processor available in full detail. It is obvious that the instructions used, and the operand formats possible will have a great impact on the future decompositions. The next chapter will contain a short recapture of the analysis from [Bud88] and then an enumeration of the used instructions and operands will be given.
After this it will be possible to make a new decomposition for the major blocks in the current decomposition, and for this is it advisable to make a complete analyses of the structure and the requirements of the modules that will be sliced up. For this purpose a software model will be created. This model should supply detailed information on the data paths used and manipulations done with this data.

Chapter 3

The definition of an Instructionset

For the "C"-processor to function, it is necessary to have an set of instructions which can be executed. (How very trivial!) The definition of this instructionset is however not a trivial matter.

3.1 The method of constructing the instructionset.

As a guideline for the decisions to be made during the selection of instructions and the subsequent assignment of codes to the instructions, a collection of objectives is generated. The source of these objectives is in the first place the famous big thumb, and secondly there are some indications for selection in the literature. [Das84]

3.1.1 First term objectives.

As guideline for the design of the instructionset, the following criteria were taken into account:

- Instructions are aligned at either BYTEs (8 bits), WORDs (16 bits) or QUADs. (The term QUAD stands for 32 bits.) A good choice seems to be alignment on WORD basis. Which gives the basic opcode a size of 16 bits.
- The possible operand sizes included in the instruction are BYTE, WORD and QUAD. If an operand is of BYTE size then it might be desirable not to waste any space due to alignments. Not from a next operand, not from a next instruction.

  This would mean that for instructions with BYTE operands the opcodefield is limited to 8 bits.
- Two operand instructions. Most of the arithmetic and some other instructions have two operands, which are both to be included in the instruction.
- If an instruction contains a QUAD immediate operand, then this operand will be aligned on a QUAD boundary. The two lowest address bits are zero. If the alignment

leaves holes in the code space, they will be treated as NOPs, it is up to the execution-unit to detect these items in time to ignore them completely.

The coding of the instructions is chosen in such a way that the decoding and/or the interpretation can be done in a straightforward way. Straightforward here means in as little time as possible.

The following addressing modes are candidates for implementation. With these items one has to keep in mind that the processor under design has to be of semi-RISC architecture, and this prohibits the use of complex addressing modes such as base-indexed, indirect-indexed, etc. The reason for this is that in most processors a lot of registers are used in these modes to get a high performance. Since our processor does not stand out in the number of onboard registers these modes are taboo. The emphasis should be on the dedicated stack in the system

The only exemptions from this are the indirect modes, which need one extra cycle to obtain the final result. And the linked-list mode which will access to get next operands, and these might cause extra accesses given their format.

- Register.

  SP, <TOS>, FP, EFP, Flags, PC.
  Although the previous arguments favor a rigid and orthogonal instructionset, not all registers need to be possible in all instructions. (i.e. Flags and PC)
  For the <TOS> register could using the register mode mean: implementation of autoincrement and autodecrement: POP and PUSH.

- Immediate.

  Data size: BYTE, WORD, QUAD.
  The operand is contained in the instruction.

- Direct.

  Address size: WORD, QUAD.
  (BYTE is possible but gives page *ZERO* addressing. Which does not seem very important in this design.)
  The address of the operand is included in the instruction.

- Relative.

  Offset size: BYTE, WORD, QUAD.
  The address of the operand is found through the addition of the contents of a register and an offset contained in the instruction. The registers used with this mode are mostly SP, FP, EFP, <TOS>. The PC could be used for a, in code included, table of constants.

- Indirect.

Address sizes: **Word**, **Quad**.
(For **Byte** indirect addressing can be said exactly the same as with direct addressing.)
The field in the instruction gives the address where the operands address can be found.

- Relative Indirect.

  Offset: **Byte**, **Word**, **Quad**.
  The offset in the instruction and the indicated register create the address where the address of the operand can be found.
  The registers for this mode are again: SP, FP, EFP, <TOS>.

- Next to the previous types of "normal" operands is the linked-list addressing. This kind of addressing is specifically targeted towards high level programming languages.
- To get the value of a variable which is defined on a higher lex-level in PASCAL, a series of static links has to be processed before the required value can be obtained.
- In "C" often variables are referenced with more than one level of indirection. Since the types and repetitions of the indirections are known at compile-time is it possible to create a list of actions to be taken.
- This method does not have a fixed format. It consists of a repeated list of addressing operands with offsets. The offset of a item in the chain is calculated by part of the chain already processed.
- Offsets specified in the instructions are ,where used, of several sizes. **Byte** and **Word** are valid candidates for this. **Quad** would be comfortable if large structures are used on the External Frame. (Large means bigger than 64K)

Literature indicates that a considerable amount of processing power is spent on the transportation of data from one location to the next. (approx. 50%)
Next to these instructions are the arithmetic-instructions (approx. 35%) of which the add instruction is by far out the most used instruction.
Even more difficult is the fact that in most cases of immediate data the size of the data is very small. The instructions ADD 1, LOAD 0, TESTBIT 1 are very common examples of this type.

3.1.2 First term problems.

Given the above statements, some of the problems already shine on the horizon.

- If all possibilities are simply coded in the instruction, without regards to compactification, the size of an instruction might well be larger than 32-bits.

- When however all operand formats and opcodes are merged into one large bit field, then the decoding of a instruction will require a large block of logic to decode the exact meaning of the opcode and the operand formats.
- When a two-operand instruction has operands of different sizes, the smaller operand has to be converted to the size of the larger one. This extension can be done by either sign-extension or by logic- extension. Some information regarding the types has to be added in the instruction.

3.2 Inventory of operations and functions.

The previous section contains an enumeration of possible operands and their addressing methods. To be able to define an instructionset one needs a list of operations on these operands. Furthermore needed are other instructions. A list is given below which is extracted from: [Bud88], 68000 user manual, am29000 user manual and the P4-code instructions. Note that this list is not the full and complete union of the four mentioned instructionsets. The report [Bud88] and the P4-code were used as prime sources, the other two are "checked" to see if there where any omissions.

3.2.1 Listing of all possible instructions.

This is a list with:

- A description of the instruction,
- A mnemonic and
- The number of possible operands.

NOTE:

> *This is not an listing of the instructions in assembler format. It is mere jested as a inventory list for the instructions possibly implemented.*
> *If and when the instruction are going to be used, then a definition of the assembler foramt would be apropriate.*

End NOTE

The instructions are grouped in appropriate classes:

- Arithmetic, boolean and logic operations.
  Table 3.1
  - Boolean operations can have only two result values. The result is either 1 or 0, (**true** or **false**.) A non boolean value is always considered **true** if it is not 0.

Source	Description	Mnem.	#operands
B	Add.	ADD	2
B	Add with Carry.	ADDC	2
B	Boolean AND.	BAND	2
B	Logical AND.	LAND	2
B	Boolean OR.	BOR	2
B	Logical OR.	LOR	2
B	Logic XOR.	XOR	2
M	Check bounds.	CHK	3
B	Compare.	CMP	2
B	Divide Unsigned.	DIVU	2
B	Divide Signed.	DIVS	2
A	Extract part of item.	EXTR	2
B	Modulo Unsigned.	MODU	2
B	Modulo Signed.	MODS	2
B	Multiply Unsigned	MULU	2
B	Multiply Signed	MULS	2
P	Boolean Not.	BNOT	1
P	Logic Not.	LNOT	1
B	Negate	NEG	1
B	Move	MOVE	2
B	Shift left.	SHL	2
B	Shift right, arithmetic Signed.	SARS	2
B	Shift right, arithmetic Unsigned.	SARU	2
B	Subtract.	SUB	2
B	Subtract, with Borrow.	SUBB	2
B	Select operand	SEL	3

Table 3.1: Suggestions for Arithmetic and Logic operations

Source	Description	Mnem.	#operands
B	Call "high level" Function.	CALLF	2
B	Call Subroutine	CALL	1
B	Conditional Jump.	JMPC	2
B	Jump, Goto.	JMP	1
B	Return from "high level" Function.	RETF	1
B	Return from Interrupt.	IRET	0
B	Return from Subroutine	RET	0
B	Swap context.	SWAP	2
M	Test, Decrement and Jump.	TDJ	2
M	Trap.	TRAP	1

Table 3.2: Flow control operations

Source	Description	Mnem.	#operands
B	Call "high level" Function.	CALLF	2
B	Clear Instruction Cache	CIC	0
B	Enable interrupt.	ENAINT	1
B	Disable interrupt	DISINT	1
A	Halt.	HALT	0
A	Lock.	LOCK	0
M	Reset external system.	RESET	0
B	Update Main Memory with data cache	UPDM	0

Table 3.3: Processor control operations

  - Logic operations are operations which are performed bitwise on all bits of the operands.

- Flow-control instructions.
  Table 3.2

- Processor-control instructions.
  Table 3.3

The indications with the instructions, A = am29000, B = [Bud88], M = Motorola 68000, P = P4-code, give an indication where the instruction is first found during the generation of this list. It does not suggest as to which processor knows which instructions.
This gives 42 different types of opcodes to be implemented.
It is also possible to classify the instructions by the way they handle their operands. In the following paragraphs different types of instructions will be introduced.

3.2.2 P-type instructions.

These instructions emphasize the stack oriented character of the processor. On a "regular" processor they are usually very slow and cumbersome. Here this is not the case due to the typical implementation of the stack-cache, or stack-window.
The instructions are targeted at:

- Data manipulation on the stack, in the stack-window.
- Use autoincrement and autodecrement modes with the stackpointer, when used as <TOS> operand.
- Are short and compact. (8 or 16 bits)
- Can contain short immediate data.
- Some can take one operand from the local frame using the FP.

In the following description the notation given below is used:

- <TOS> the QUAD pointed at by the stackpointer SP.
- <TOS – 1> is the QUAD just below the stackpointer.
- <TOS – 2> is the QUAD just 2 below the stackpointer.

Instructions with implied operands.

These instructions have complete implicit addressing, no operands what so ever are present in the instruction.

1 Operand instructions.

Plnot.

$<TOS>:=$ lnot $<TOS>$

Logic bitwise complement the top of stack.

Pbnot.

$<TOS>:=$ bnot $<TOS>$

Boolean complement the top of stack.

Pneg.

$<TOS>:= - <TOS>$

2's-complement the top of stack.

Pdup.

$<TOS+1>:= <TOS>$

$;SP:= SP + 1$

Create a copy of $<TOS>$on the stack.

2 operand instructions.

All of the following instructions have the same description except for the ALU-action to be performed:

'code' = Padd, Paddc, Pland, Pband, Plor, Pbor Pxor, Pdivu,Pdivs, Pmodu, Pmods, Pmultu, Pmults, Psub, Psubb, Pshl, Pshrs, Pshru.

Where the sequence of action is given by:

$<TOS-1>:= <TOS-1>$ 'code' $<TOS>$

$;SP:= SP - 1$

In plain English:
The two top elements of the stack are added and deleted from the stack. The result will be placed back on the stack. This is shown in figure 3.1.

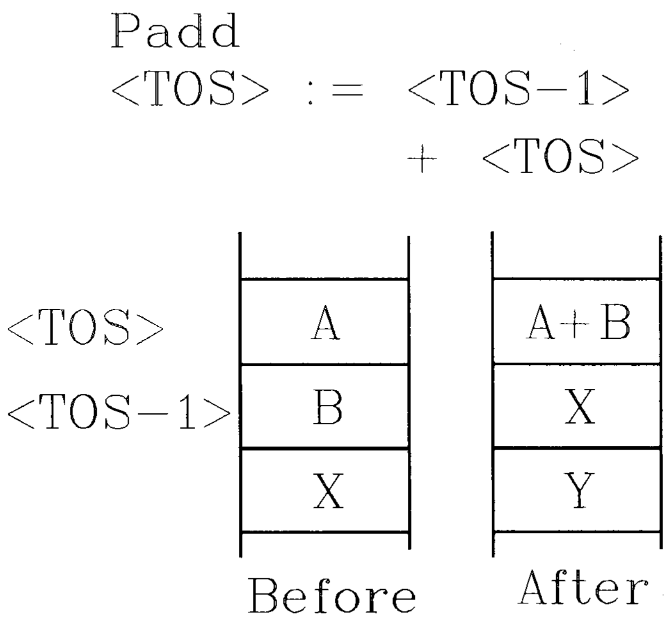

Figure 3.1: An example of a two operand P-code

At first it may seem strange to use $< \mathsf{TOS} - 1 >$ as the first operand for the instruction, since $< \mathsf{TOS} >$ is ready available. But it is the design target to execute instructions like this in one cycle, and hence both have to be fetched in parallel. And if one considers that statements of the construction "A := A 'code' B " are very common, then the used sequence becomes even more obvious.
In the remaining text of this thesis it will be assumed, unless stated otherwise, that in constructions like the ones discussed here **A** represents operand 1, and **B** represents operand 2.
Pcomp.

Flags:= $< \mathsf{TOS} - 1 >$ - $< \mathsf{TOS} >$

;SP:= SP - 2

Set the flags according the result of a subtraction of the top 2 elements. The elements are removed from the stack.
Pswap.

$< \mathsf{TOS} >$:= $< \mathsf{TOS} - 1 >$

'parallel'

$< \mathsf{TOS} - 1 >$:= $< \mathsf{TOS} >$
Swap the top two elements in the stack.

3 Operand instructions.

Pcheck.

Flags or <TRAP> := $<\mathsf{TOS}-2> <= <\mathsf{TOS}> <= <\mathsf{TOS}-1>$

(possibly $<\mathsf{TOS}-2> := <\mathsf{TOS}>$)

;SP:= SP- 3 (-2???)

Check the bounds of the top against the two given bounds on the stack. It is either possible to generate a TRAP if the check fails. It is also possible to set flag to indicate a failure.

Psel.

$<\mathsf{TOS}-2> := (<\mathsf{TOS}-2>?\ <\mathsf{TOS}-1>:\ <\mathsf{TOS}>)$

;SP:= SP- 2

Replace the top of stack with either $<\mathsf{TOS}-1>$ or $<\mathsf{TOS}>$ dependant on the value of the selector in $<\mathsf{TOS}-2>$.

Remarks on P-instructions with implicit operands

All instructions described above have no specified operands, and their mnemonics can be used as is. It is also clear that the majority of the instructions will have two operands, only a few will have 1 operand. Only 2 instructions require three operands, of which the Check instruction also requires two aritmetic actions.

Instructions with explicit operands.

In most cases where immediate data is used, the size of the data is small. One of the most common actions is the addition of one, increment, or a subtraction of one, decrement. Another class of instructions with parameters is shifts and rotates. The explicit operands indicate the number of places to be shifted. Also possible is an extract with two operands: They indicate the place of the first bit to be extracted and the length of the string to be extracted.
The size of the operand is made 8 bits in arithmetic and logic operations. This gives a possibility to act on values which are actually 8 bits, characters for instance. For the shifts and the extract an operand size of 5 bits will generate all possible index addresses.

Arithmetic and logic instructions

'code' = Paddi, Paddci, Plandi, Pbandi, Plori, Pbori, Pxori, Pdivui, Pdivsi, Pmodui, Pmodsi, Pmultui, Pmultsi, Ppushi, Psubi, Psubbi, Pcompi.

Where the sequence of action is given by:

<TOS>:= <TOS>'code' <extended immediate>

Next to the 7-bits immediate data included, the data BYTE also contains a bitflag to indicate whether the immediate data is to be logic or sign extended. It is also possible to see the data as an 8 bits signed integer which is always sign extended. The ultimate effect will be the same.

The immediate data will be the second operand during calculations of this type of instruction.

Extract.

Pextract.

$< TOS$[0..length]> := $< TOS$[start..(start+length)]>

The bitstring [start..(start+length)] replaces the < TOS > right justified. (if length = 0 exactly **one** bit is extracted!) Since this extraction needs 2 * 5 bits to identify all extractions possible, 2 bits from the instruction code are used. This gives 4 Pextract opcodes, they all extract a part of the < TOS > value, but dependant on the start value another version of the extract instruction will be used.

Instructions with a relative reference to FP.

The instructions with a reference to the FP use the location referred to in the same way as the short P-instructions use < TOS >. As a consequence of this 2-operand P-instructions are the only instructions that are able to use this mode. These instructions were:

Padd, Paddc, Pland, Pband, Plor, Pbor, Pxor, Pdivu, Pdivs, Pmodu, Pmods, Pmultu, Pmults, Ppush, Psub, Psubb, Pcomp.

The index is an offset to the FP and thus are they in "C" or PASCAL references to the local variable space and current parameters from the routine under execution. The size of the index is 8 bits, in signed notation. This makes it possible to address 127 locations in the local frame, and 126 locations in the parameterlist. (The maximum negative number is 128, but the locations -1 and -2 are references to the old FP and return address.)

NOTE:

One has to realize that references made with the FP *as base register are always to* QUAD *units. Since bits are considered valuable in P-instructions the units of displacement also reference* QUADs. *This in contrast with F(ull)-operand instructions where the addressing is done in* BYTE *units.*

End NOTE

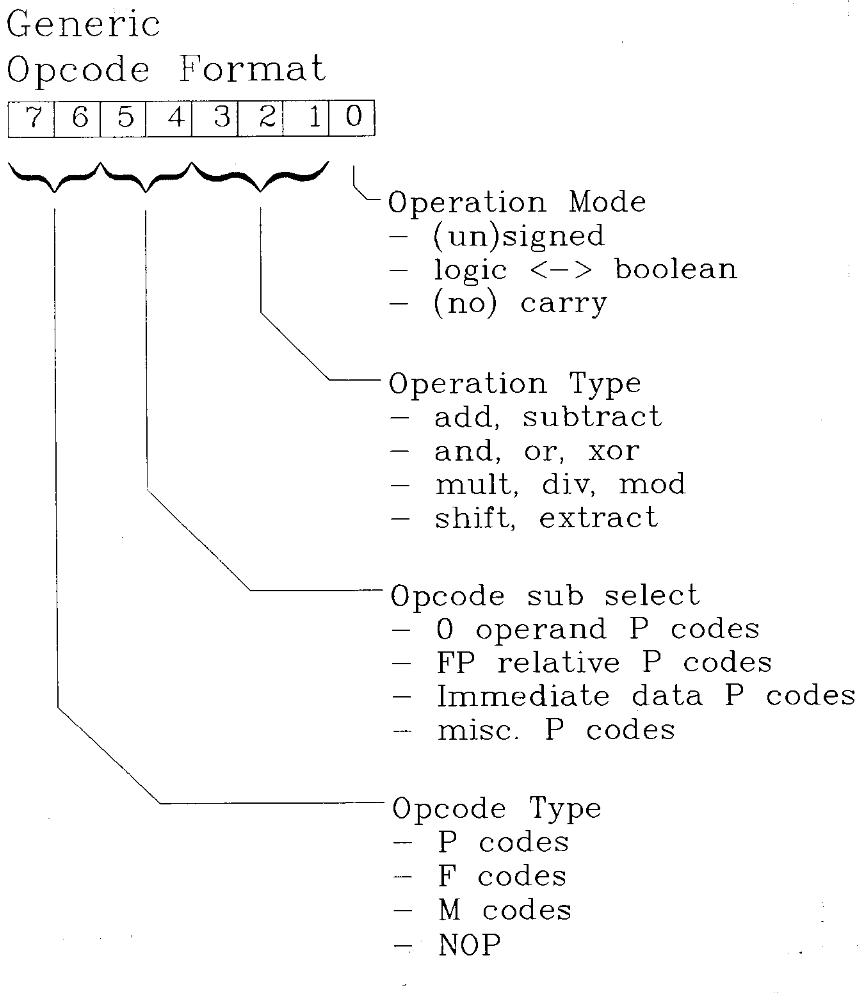

Figure 3.2: The bit assignment for P-opcodes

Bit-allocations for the instructions.

There are all in all there 64 instructions in the P-instructionset. This takes 6 bits from the opcode field. If instructions can be aligned on BYTE addresses and opcodes are 8 bits wide then with the remaining 2 bits the indication must be given that the instruction to

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

P-code instructions

P-opcode	P-opcode	P-opcode	P-opcode

P-code instructions with data field

Pdata-opcode	Offs. or Imm.	...	...
...	Pdata-opcode	Offs. or Imm.	...
...	...	Pdata-opcode	Offs. or Imm.

Table 3.4: Several P-opcodes merged into one QUAD.

be used in this opcode is part of the P-instructionset. In figure 3.2 are the bit assignments for the modes depicted, in table 3.4 is show how P-codes are merged into QUADs.

3.2.3 F-instructions.

Are P-instructions targeted towards the optimal use of the onboard stack-cache. The F(ull)-instructionset forms an orthogonal set of instructions with various methods of addressing, referencing and indexing. And this is true for almost all instructions mentioned in the inventory of operations and instructions. The exceptions are those instructions where the operands are of a specific type.
For every operand in an instruction are all possible addressing modes possible, whether this is immediate, indirect or indirect relative.
Relative instructions can use either SP, FP, EFP or < TOS > (and/or PC) as the base register.
The registers usable in the register mode are: SP, FP, EFP, PC, < TOS >and Flags. < TOS >, **which is not just a short notation** for SP[0], gives an automatic update of the stackpointer SP. If the < TOS > operand is used as a operand which is read than the item is popped from the stack. If < TOS > is used as base value in an offset calculation then the base value is left on stack, it has to be removed explicitely. Is the result of the instruction placed in < TOS > then the result is pushed on the stack.
The exception to this rule is a move to < TOS >. In this case there is no operation involved using the first operand, This is the mere destination. The action is corrected and the SP is incremented before the value is loaded on the stack. The old < TOS > value is ,as a consequence of this, kept as < TOS − 1 >, the new value becomes < TOS >.

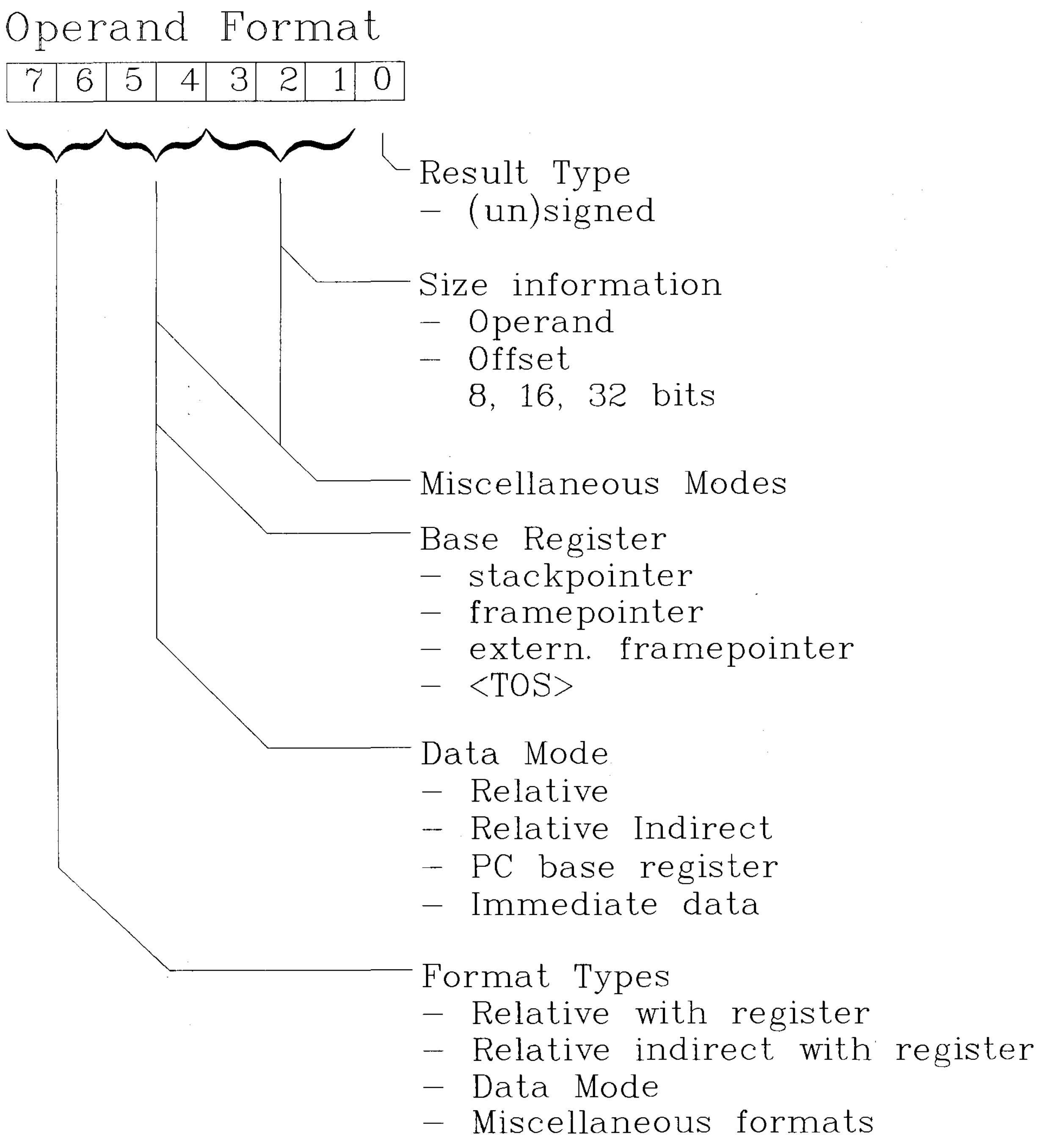

Figure 3.3: The bit assignment for F-operands

Opcodes in the F-instructionset.

All F-instructions, arithmetic and logic, can be combined with all variants of the the F-operands.
The used arithmetic and logic instructions are:

> Fadd, Faddc, Fband, Fland, Fbor, Flor, Fxor, Fcheck, Fcomp, Fdivu, Fdivs, Fmodu, Fmods, Fbnot, Flnot, Fmove, Fshl, Fshrs, Fshrs, Fsub, Fsubb, Fselect.

If one visually depicts the bit assignment of an opcode BYTE, it looks like figure 3.3.

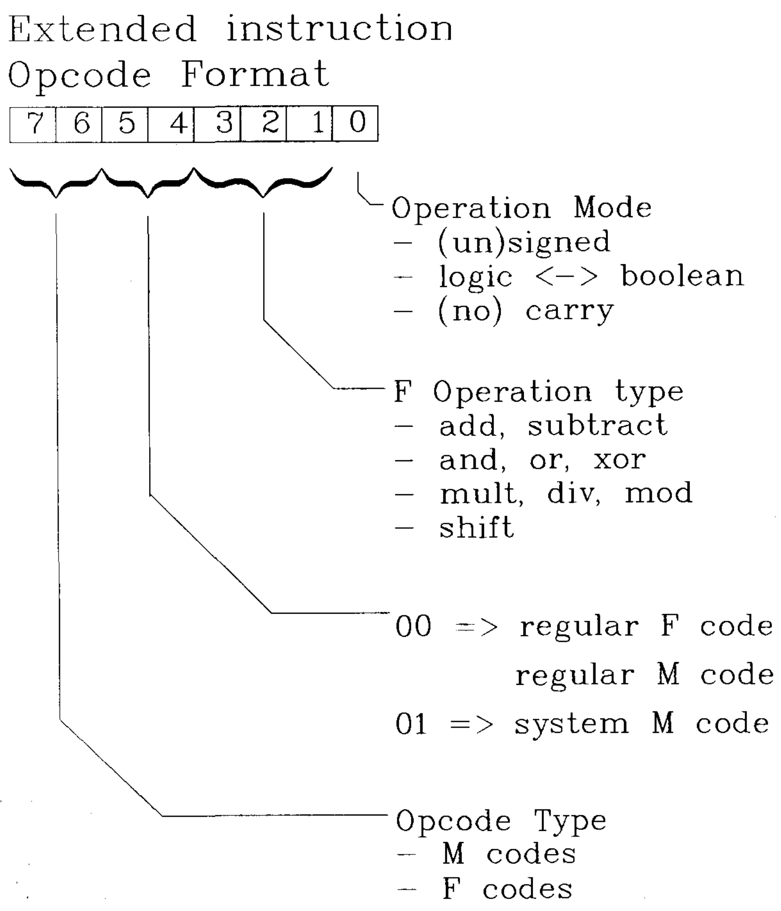

Figure 3.4: The bit assignment for F-opcodes

F-operands.

The F-instructions are characterized by their large amount of operand types possible. These operands, F-operands, are the key item for the orthogonality of this part of the instructionset.

The following section gives a complete list with all possible operands. First are listed all partial components. Next are listed all combinations of the partial components.

Addressing modes.

Immediate.
Register.
Direct.
Relative.
Relative Indirect.

Result Types.

Signed.
Unsigned.

Sizes.

8 bits or BYTE.
16 bits or WORD.
32 bits or QUAD.

Combinations of all items.

Explanation of the used notation.

"N#IM":{s|u}M has the following meaning: an immediate value **IM** is stored in the code-stream, the size of the information is N bits. (N = 8, 16, 32). This value has to be extended to M bits. (signed or unsigned). The resulting value is then used for any operation indicated.

NOTE:

Every operation uses a 32-bit intermediate value, this according K&R. All operations are converted to 32 bits according to the type of the operand. The resulting value will be truncated in the highest bits to fit into the format of the destination.

End NOTE

[*]	is the notation that stands for the contents of memory location *.
reg[*]	is the memory location addressed by the addition of the register contents and *.
=	Items marked with = are equivalent with another possible format, and are therefor not used.
*	Items marked with * are not used to keep the number of possibilities with 8 bits.

```
{ Immediate }
"im8":S8                "im8":S16               "im8":S32
"im8":U8 =              "im8":U16               "im8":U32

"im16":S8 =             "im16":S16              "im16":S32
"im16":U8 =             "im16":U16 =            "im16":U32

"im32":S8 =             "im32":S16 =            "im32":S32
"im32":U8 =             "im32":U16 =            "im32":U32
{ Register }
{         The registers are only usable as a QUAD quantity.
          The exception is <TOS> but than this is not a real
          register.  <TOS> can use all possible formats, but the
          results on the stack are always aligned on QUAD addresses.
}
SP:S8 *                 SP:S16 *                SP:S32 *
SP:U8 *                 SP:U16 *                SP:U32

FP:S8 *                 FP:S16 *                FP:S32 *
FP:U8 *                 FP:U16 *                FP:U32

EFP:S8 *                EFP:S16 *               EFP:S32 *
EFP:U8 *                EFP:U16 *               EFP:U32

PC:S8 *                 PC:S16 *                PC:S32 *
PC:U8 *                 PC:U16 *                PC:U32
{ with auto-stack }
<TOS>:S8                <TOS>:S16               <TOS>:S32
<TOS>:U8                <TOS>:U16               <TOS>:U32

<flags>:S8 *            <flags>:S16 *           <flags>:S32 *
<flags>:U8 *            <flags>:U16 *           <flags>:U32
{ Direct }
["addr8"]:S8            ["addr8"]:S16           ["addr8"]:S32
["addr8"]:U8            ["addr8"]:U16           ["addr8"]:U32

["addr16"]:S8           ["addr16"]:S16          ["addr16"]:S32
["addr16"]:U8           ["addr16"]:U16          ["addr16"]:U32

["addr32"]:S8           ["addr32"]:S16          ["addr32"]:S32
["addr32"]:U8           ["addr32"]:U16          ["addr32"]:U32
{ Indirect }
[["addr8"]]:S8          [["addr8"]]:S16         [["addr8"]]:S32
[["addr8"]]:U8          [["addr8"]]:U16         [["addr8"]]:U32

[["addr16"]]:S8         [["addr16"]]:S16        [["addr16"]]:S32
[["addr16"]]:U8         [["addr16"]]:U16        [["addr16"]]:U32

[["addr32"]]:S8         [["addr32"]]:S16        [["addr32"]]:S32
[["addr32"]]:U8         [["addr32"]]:U16        [["addr32"]]:U32
{ Relative }
SP["off8"]:S8           SP["off8"]:S16          SP["off8"]:S32
SP["off8"]:U8           SP["off8"]:U16          SP["off8"]:U32

SP["off16"]:S8          SP["off16"]:S16         SP["off16"]:S32
SP["off16"]:U8          SP["off16"]:U16         SP["off16"]:U32

SP["off32"]:S8          SP["off32"]:S16         SP["off32"]:S32
```

SP["off32"]:U8	SP["off32"]:U16	SP["off32"]:U32
FP["off8"]:S8	FP["off8"]:S16	FP["off8"]:S32
FP["off8"]:U8	FP["off8"]:U16	FP["off8"]:U32
FP["off16"]:S8	FP["off16"]:S16	FP["off16"]:S32
FP["off16"]:U8	FP["off16"]:U16	FP["off16"]:U32
FP["off32"]:S8	FP["off32"]:S16	FP["off32"]:S32
FP["off32"]:U8	FP["off32"]:U16	FP["off32"]:U32
EFP["off8"]:S8	EFP["off8"]:S16	EFP["off8"]:S32
EFP["off8"]:U8	EFP["off8"]:U16	EFP["off8"]:U32
EFP["off16"]:S8	EFP["off16"]:S16	EFP["off16"]:S32
EFP["off16"]:U8	EFP["off16"]:U16	EFP["off16"]:U32
EFP["off32"]:S8	EFP["off32"]:S16	EFP["off32"]:S32
EFP["off32"]:U8	EFP["off32"]:U16	EFP["off32"]:U32
PC["off8"]:S8	PC["off8"]:S16	PC["off8"]:S32
PC["off8"]:U8	PC["off8"]:U16	PC["off8"]:U32
PC["off16"]:S8	PC["off16"]:S16	PC["off16"]:S32
PC["off16"]:U8	PC["off16"]:U16	PC["off16"]:U32
PC["off32"]:S8	PC["off32"]:S16	PC["off32"]:S32
PC["off32"]:U8	PC["off32"]:U16	PC["off32"]:U32
<TOS>["off8"]:S8	<TOS>["off8"]:S16	<TOS>["off8"]:S32
<TOS>["off8"]:U8	<TOS>["off8"]:U16	<TOS>["off8"]:U32
<TOS>["off16"]:S8	<TOS>["off16"]:S16	<TOS>["off16"]:S32
<TOS>["off16"]:U8	<TOS>["off16"]:U16	<TOS>["off16"]:U32
<TOS>["off32"]:S8	<TOS>["off32"]:S16	<TOS>["off32"]:S32
<TOS>["off32"]:U8	<TOS>["off32"]:U16	<TOS>["off32"]:U32
{ Relative Indirect }		
[SP["off8"]]:S8	[SP["off8"]]:S16	[SP["off8"]]:S32
[SP["off8"]]:U8	[SP["off8"]]:U16	[SP["off8"]]:U32
[SP["off16"]]:S8	[SP["off16"]]:S16	[SP["off16"]]:S32
[SP["off16"]]:U8	[SP["off16"]]:U16	[SP["off16"]]:U32
[SP["off32"]]:S8	[SP["off32"]]:S16	[SP["off32"]]:S32
[SP["off32"]]:U8	[SP["off32"]]:U16	[SP["off32"]]:U32
[FP["off8"]]:S8	[FP["off8"]]:S16	[FP["off8"]]:S32
[FP["off8"]]:U8	[FP["off8"]]:U16	[FP["off8"]]:U32
[FP["off16"]]:S8	[FP["off16"]]:S16	[FP["off16"]]:S32
[FP["off16"]]:U8	[FP["off16"]]:U16	[FP["off16"]]:U32
[FP["off32"]]:S8	[FP["off32"]]:S16	[FP["off32"]]:S32
[FP["off32"]]:U8	[FP["off32"]]:U16	[FP["off32"]]:U32
[EFP["off8"]]:S8	[EFP["off8"]]:S16	[EFP["off8"]]:S32
[EFP["off8"]]:U8	[EFP["off8"]]:U16	[EFP["off8"]]:U32
[EFP["off16"]]:S8	[EFP["off16"]]:S16	[EFP["off16"]]:S32
[EFP["off16"]]:U8	[EFP["off16"]]:U16	[EFP["off16"]]:U32
[EFP["off32"]]:S8	[EFP["off32"]]:S16	[EFP["off32"]]:S32
[EFP["off32"]]:U8	[EFP["off32"]]:U16	[EFP["off32"]]:U32
[<TOS>["off8"]]:S8	[<TOS>["off8"]]:S16	[<TOS>["off8"]]:S32

```
[<TOS>["off8"]]:U8 [<TOS>["off8"]]:U16  [<TOS>["off8"]]:U32
[<TOS>["off16"]]:S8[<TOS>["off16"]]:S16 [<TOS>["off16"]]:S32
[<TOS>["off16"]]:U8[<TOS>["off16"]]:U16 [<TOS>["off16"]]:U32
[<TOS>["off32"]]:S8[<TOS>["off32"]]:S16 [<TOS>["off32"]]:S32
[<TOS>["off32"]]:U8[<TOS>["off32"]]:U16 [<TOS>["off32"]]:U32
```

For an exact definition of the values assigned to the operand codes, see the appendix with the definition of the instructionset. In this appendix the formats are described as encoded bitfields and a full enumeration of the values and their functionality is given. The figure 3.4 gives however an idea of the bit assignments used for operand formats. In the table 3.5 are examples given of the packing of an F-instruction into the code stream. (This table can be found at the end of the chapter.)

3.2.4 M(ixed)-instructions.

A processor needs more instructions types than the instructions explained in detail in the previous sections. This section of the instructionset description covers those instructions that control the flow of the program, maintain the processor status, etc.
The instructions in this collection do not created a neatly organised subset. Every item has its specific operands and operand formats. The only operand format that used more general, is the F-operand. This operand has all the possible forms as given in the section on F-operands.

FLOW-CONTROL instructions.

- CALL "high level" function with "C" mechanism.

  Instruction format: **Mcallc** "8U-operand","F-operand"

  The "8U-operand" is a subset of the F-operand, namely all formats which have an 8 bit unsigned integer as a result. (i.e. "im8":U8 or < TOS >:U8). The integer indicates the number of QUADs to be reserved for extra parameter variables on entrance of the function. This space can be used for return values. The local variable space, which is only known inside the function, is created by adding a value to SP. If the value of the second operand is outside its range, this will not result in a system error. One cannot rely on the resulting actions to be reproducible.

  The "F-operand" gives the address of the procedure or function to be called.

  First the size for the parameter area is added to the SP.

  Next is the programcounter pushed on the stack.

  Then the frame pointer is pushed on the stack, and once it is pushed, FP is updated to point to <TOS>. This leaves the framepointer pointing to the first local variable, if needed.

The code in the called routine is now free to reserve any extra space for local variables by adding a value to the SP.

- CALL "high level" function with **PASCAL** mechanism.

  Instruction format: **Mcallp** "F-operand", "8U-operand"

  The "F-operand" gives the address of the procedure or function to be called.

  The "8U-operand" is a subset of the F-operand, namely all formats which have a 8 bit unsigned integer as a result. (i.e. "im8":U8 or < TOS >:U8). The integer indicates the number of QUADs to be reserved for extra **local stack** variables on entrance of the function. This space can be used for the local variable space, which is only known inside the function.

  First is the programcounter pushed on stack.

  Second is the frame pointer pushed on the stack.

  Third is the SP is pushed onto the stack to keep the return address. Parallel with this is the framepointer updated to point to the first entry in the new local variable. Thus under normal operation will FP[0] indicate the first local variable. The last action is the addition of the "8U-operand" to the stackpointer.

- CALL subroutine.

  Instruction format: **Mcall** "F-operand"

  This type of call has a strong likeness with the previous calls. The only difference is that no stack-space is allocated for parameter or local variables and thus the FP pointer needs no changes. It is left pointing to the current frame of local variables. Local variables are only created explicitly if the SP is incremented.

- JUMP, GOTO.

  Instruction format: **Mjmp** "F-operand"

  The F-operand gives the new address for the programcounter to be loaded with.

- Conditional JUMP, GOTO.

  Instruction format: **Mjmpc** "condition", "F-operand"

  The flow-control is directed to the address of the F- operand when the given condition shows **TRUE**. Otherwise the instruction following the **jmpc** instruction is executed.

  The following conditions can be used:

- Zero.
- Non zero.
- True.
- False.
- Equal.
- Not equal.
- less.
- greater.
- less or equal.
- greater or equal.
- Sign (negative).
- No Sign (positive).

It is not said that all of the above conditions are represented by different flag settings.

- RETURN from "high level" function.

Instruction format: **Mretf** "8U-operand"

This return can be used for returning from a call in a **PASCAL** routine and from a call in a "C" routine.

First the 8-bits operand releases the indicated number of QUADs for the local variables.

Secondly the FP is restored to the value it had before calling the "high level" function.

Finally program execution is resumed on the address that is found on <TOS>.

These actions leave the stackpointer pointing to the top element in the list with parameters. If these parameters have to be ignored, they have to be removed explicitly by subtraction of the number of nparameters from the stackpointer.

- RETURN from subroutine.

Instruction format: **Mret**.

Execution proceeds to the address given in <TOS>. The FP remains unchanged.

- RETURN from interrupt.

Instruction format: **Mreti**.

The execution of an interrupt routine is completed. And the processor returns to the task it was executing when it was interrupted. The stored processorstatus is retrieved from the stack.

- SWAP context.

  Instruction format: **Mswap** "32-address A", "32-address B"

  The complete processorstatus is stored in a processorstatus frame given by the absolute address. The new processorstatus is loaded from address B.

- DECRement, TEST and JUMP.

  Instruction format: **Mdtj** "F-operand A", "F-operand B"

  De variable addressed by operand A is decremented. If the result of the operation is not zero, control is transferred to the address given by operand B.

- TRAP.

  Instruction format: **Mtrap** "im8" instruction.

  Processorstatus is saved on the stack and the interrupt service routine of interrupt "im8" is executed.

Processor handling.

- Clear Instruction Cache.

  Instruction format: **Mcic**. Is a valid short instruction.

  All valid tags from the instruction cache are cleared.

- Update DATA memory.

  Instruction format: **Mupm**. Is a valid short instruction.

  All entry's in the stack-window are flushed to main memory.

- Enable Interrupts.

  Instruction format: **Menaint** "F-operand"

  The F-operand contains a mask. All interrupt levels with a 1 in the mask are enabled.

- Disable Interrupts.

  Instruction format: **Mdisint** "F-operand"

  The F-operand contains a mask. All interrupt levels with a 1 in the mask are disabled.

- Reset

Instruction format: **Mreset**. Is a valid short instruction.

The external reset line is asserted by the processor. The internal status of the processor is maintained. Execution continues with the next instruction.

- Halt.

  Instruction format: **Mhalt**. Is a valid short instruction.

  The processor ceases to execute the next instruction. All data-, address- and control-lines are tri-stated and the HALT line is asserted. The only exit from this state is an external reset or an interrupt on a level that is not masked.

- Lock.

  Instruction format: **Mlock "count8"**

  Count8 specifies the number of instructions(or system cycles or clock cycles. This is not yet decided on at this time.) the system busses are locked by the processor. Interrupt handling will also be postponed for this period of time.

For a full description of all instructions and their assigned opcodes is referred to appendix with the instructionset description. Here the complete instructionset is listed with all possible operand values, including the various bitfield assignments.

3.3 Remarks.

Although it was indicated that a more complex type of linked-list operands could prove very useful, this type of operands is currently not implemented.
Reasons for this could be the following:

- Definition of the instructionset consumed far more time than expected, and the exact definition of a linked list operand type requires lots of "bit fiddling".
- The linked list operand could be implemented in full extend. This means that all combinations of operands could be possible, and as a consequence of this far to many combinations would be available. However once information is available on indexing methods used or wanted, a more appropriate set of operands could be created.

The currently introduced instructionset has however properties which are not directly obvious at first. For instance the following question (and answer) could be discussed:
Has the "C"-processor no PC-relative jump?

No! Not in the direct sense that there is an opcode or an operand for this action.

Yes! The instruction **Fadd** PC, **'operand2'** will give a relative branch with the offset given by the value of operand2.

Table 3.5: Layout of 1-operand FM instruction in the code stream

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

FM-instruction with 1 Operand

FM-opcode	Operl Format	...	Byte Data

With 1 Operand of Word Size

FM-opcode	Operl Format	...	...
...		Word Data	

With 1 Operand of Quad Size

FM-opcode	Operl Format	...	...
Word Data			

If, however, a relative conditional jump needs to be taken (or a relative call), then the matter becomes a little more complex. A construction for this would be to create the new address in $< \mathsf{TOS} >$. And than take the conditional jump with the value in $< \mathsf{TOS} >$ as the new address.
It has to be admitted that the last construction is not a very elegant one. But the lacking of this type of operand has become clear only after writing the Software model, and after the implementation of the HHDL models going with the operand unit. If this omission would be considered serious enough, then it could be implemented relatively fast. (As long as the design is in the High Level design phase) There are still opcodes and operand formats available. The appropriated codes for these formats have to be inserted in the large case statements in the operand fetchers. (in the operand unit)

Table 3.6: Layout of 2-operand FM instruction in the code stream

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2 Operand FM-instructions

Without Operand Data

FM-opcode	Operand 1 Format	Operand 2 Format	...

With Operand 1 = Byte, Operand 2 = Byte.

FM-opcode	Operand 1 Format	Operand 2 Format	Operand 1 Byte
...	...	...	Operand 2 Byte

With no Operand 1 data, Operand 2 = Byte.

FM-opcode	Operand 1 Format	Operand 2 Format	Operand 2 Byte

With Operand 1 = Word, Operand 2 = Byte.

FM-opcode	Operand 1 Format	Operand 2 Format	Operand 2 Byte
...		Operand 1 Word	

With Operand 1 = Word, Operand 2 = Quad.

FM-opcode	Operand 1 Format	Operand 2 Format	...
...		Operand 1 Word	
Operand 2 Quad			

With Operand 1 = Quad, Operand 2 = Quad.

FM-opcode	Operand 1 Format	Operand 2 Format	...
Operand 1 Quad			
Operand 2 Quad			

Chapter 4

A first model description of the "C"-processor.

Given the instruction set, defined in the previous chapter, and the first decomposition model, it is possible create a software model. With this software processor model is it possible to:

- Test the designed instruction set and test its functionality. This will give an indication whether the chosen instruction set is applicable for its purpose or not.
- Since this processor is designed to have a data cache of a very special type. The software model would give the possibility of testing the effects of different types of implementation.

Next to these "verification" possibilities the software model also gives information which can be of use in future steps of the design process.

- The software model will give a complete description of the datapath in the processor:
- All operations and manipulations with the variables are clearly visible in the code.
- The parameters communicated between routines give information on variables that are shared between only two routines.
- Global variables are an indication of variables that are used in all routines in the processor and are thus candidates for registers with a separate bus.
- The software model will be able to supply verification material for test runs of the high level implementation of the processor. The software model is currently also able to supply the values that are communicated between the modules. This makes it possible to compare the software modules with the hardware modules on a lower level.

- And as a aid in further designs on the next level, the software design could give guidelines for choosing the decomposition of modules in smaller ones.

4.1 What will the software model not do?

The software model will not, or is not able to:

- Give timing information. None of the actions of the software modules are related to any timing what so ever. As a consequence of this, is it also not possible to detect all action that can be executed in parallel.
- Perform I/O on its boundary, as were it the real chip. Core is available as a simple array of integers, which is directly used if and when a memory value is needed. This also holds for some instructions that control I/O pins: Strong examples of these are Reset and Lock.
- Another "disadvantage" of the software model is the use of the operations available in the programming language. Multiply, modulo, etc are done using the functions given by the programming language. These will have to be coded out on the real chip. The emulation will (or should) however give the correct results of the calculations.

4.2 The modules in the "C"-processor.

The software processor consists of several routines, read modules, that implement the processor as a whole. These modules are described in the following part. The modules that are of little importance are described only in short detail, others are discussed more thoroughly.
The list is generated by following the generated source files, these are listed in full in Appendix 'Software- listing' (Which is available upon request). Some of these source files contain more than one module.

- `ALU`
  This module performs all operations on the user data. If the processor needs to add two operands, is this the part where the operation is done. Operations can be done in different modes: carry versus nocarry, signed ⇔ unsigned, boolean ⇔ logic. All operands, regardless the operations required, are considered signed integer. The result of the operation is dependant on the selected mode.
  Operations done during internal actions in the processor, like the addition of an offset, are done in special dedicated subcircuits. These subcircuits are hidden in statements like: SP+=4; (This increments the stackpointer.)
- `CODE_EXE`
  The "C"-processor runs code ad infinitum, or until the power goes down. This

consists in software of a simple but never ending loop. Since, however, that would be a waste of computer time during simulation, the software loop is controlled with some termination criteria. In the loop is the first elementary instruction decoding done. Instructions are separated in full quad instructions that start on quad boundary, and byte instructions, like the P-instructions, that can start on a byte boundary.
If this is the case, then every P-code quad is disassembled by `P_exec` and the resulting P-instructions are executed in sequence.

- `MEM_ACCESS`
  For the processor to operate useful, it needs to read data and opcodes from memory, after processing some data it has to write the result back to the memory. The routine **`get_mem`** and **`put_mem`** are written to execute the actions. Parameters indicate the address to be read or written, the size of the operand, and in case of a write also the value to be written.
  Tests are made for reading, or writing on not QUAD or WORD boundaries. For QUADs is this on multiples of four, WORDs on multiples of two. If an action is on an off boundary, a warning is given and the result could be correct, but usually this is not the case.
  The processor is designed to have instruction cache and data or stack cache. If they where to be implemented, they should be implemented in these routines. But as one can deduct for this text: '**Currently no caches are used!**' The interfaces to the routines: **`get_stack_cache`**, **`put_stack_cache`**, **`get_instruction_cache`**, however are available as empty routines. Currently they transfer all requests directly to **`get_mem`** and **`put_mem`**.

- `FLAGS`
  The results of arithmetic an logic operations are stored in a status register, more commonly called "the flags". The processor will use these flags as separate bits, or as just another integer, depending on the instruction format. The routines: **`flags_to_int`** and **`int_to_flags`** convert flagbits to integers and vice versa. Most probably these conversions will be very little more than straight wires in hardware. But they are needed for the correctness of the program.

- `LOADER`
  The model needs food, code, to run on. This code has to be loaded into main core before execution can start. The loader routine loads this code from a file on disk. The format for the code file is described in appendix "Using the "C"-processor Software model". This format is designed specific for this assignment.

- `STORE_RESULT`
  Once the instruction is completed, the calculated result has to be stored. The place to store the result is implied by the source of the first operand. The routine **`store_result`** takes the calculated value as parameter and stores it at the destination which is set by the routine that acquired the first operand (**`get_operand_1`**). It also

takes into account the size and format of the destination. This matches the size of the first operand read.

- `LIST_AUX`
  The processor has a processor status consisting of a number of registers, the flags, the current opcode and the stack. These items are displayed during execution, to be able to follow the processing flow in the processor. On chip are these routines not needed.

- `GET_OPERAND`
  Most instructions refer to one or two operands to perform an action on. The two routines **`get_operand_1`** and **`get_operand_2`** are very much alike. They both return the value of the operand so desired. **`Get_operand_1`**, however, also set the destination for the result. It also has to take into account whether the instruction is an instruction with one or two operands. (if it is a one operand instruction than a 16-bit offset could be stored in the instruction)
  Both routines take the operand description from the current instruction and decode it into the appropriate fields. From that the resulting operand is fetched. Implied conversions and sign-extensions are done on the fly. Both routines basically consist out of large case statements, in which every type of operand is obtained.

- `INTERP`
  This gigantic case statement selects on of the 255 possible opcodes to be executed. This is in fact the main controller that executes one instruction. It arranges for the operands to be fetched, the operation to be executed, and the result to be stored.

- `GET_OFFSET`
  Most types of operands need an offset in their calculation of the operand. Or if the operand consists of direct data, the offset equals the operand. The routines `get_offset_1` and `get_offset_2` read these offsets directly from the instruction stream. Offsets have 3 possible sizes: 8, 16 and 32 bits. But they are all considered to be signed. And the 8 an 16 bits offset are sign-extended to a 32 bits value before they are returned.

- `C_INT`
  This routine contains the main routine that is required in "C". It calls:
  - The loader.
  - The interpreter.

4.3 Abstractions from the software model.

From the now created software model should information be extracted that will guide future decisions. This paragraph will indicate what kind of data type resources are used, and what operations work on those resources. This information can be (and was) extracted from a simple thing like a cross-reference.

4.3.1 Global variables in the C-processor.

The "C"-processor has the following global variables:

- registers:
  - PC
    - used in:
      - `code_exe`, `get_offset_1`, `_2`, `get_operand_1`, `_2`, `interp`, `store_result`.
  - SP
    - used in:
      - `get_operand_1`, `_2`, `interp`, `store_result`.
  - FP
    - used in:
      - `get_operand_1`, `_2`, `interp`, `store_result`.
  - EFP
    - used in:
      - `get_operand_1`, `_2`, `interp`, `store_result`.
  - Flags
    - used in:
      - `alu`, `flags`, `get_operand_1`, `_2`, `interp`, `store_result`.

- current_instruction:
  - used in:
    - `code_exec`, `get_offset_1`, `_2`, `get_operand_1`, `_2`.

- destination indicators:
  - `destination_adr`
    - used in:
      - `get_operand_1`, `interp`, `store_result`.
  - `destination_reg`
    - used in:
      - `get_operand_1`, `interp`, `store_result`.

destination_size
  used in:
    get_operand_1, interp,
    store_result.

- communication between get_offset_1 and get_offset_2
  second_quad
    used in:
      get_offset_1, _2, interp.
  second_quad_read
    used in:
      get_offset_1, _2, interp.
  instruction_byte_free
    used in:
      get_offset_1, _2, interp.

4.3.2 Local operations in modules on variables.

The above variables are used in different modules and in different operations. Further information can be found in reviewing the operations done on what variable in which module. This will give information on what operations to implement with each of the resources.
The Flags are only assigned, read and tested. They are not operated on, neither arithmetically nor logical.

Current_instruction is assigned in code_exec. And the only othero peration on it is BYTE extraction in get_offset_1, _2 and get_operand_1, _2.

The destination items are reset in code_exec, set in get_operand_1 and read in store_result. Destination_reg is also read in inter to check whether the destination is <TOS>. In this case the SP should be updated.

Instruction_byte_free is reset in interp, eventually set in get_offset_1 and read in get_offset_2.

In get_offset_1 possible a second quad is read and if this extra quad was only needed for a word, second_quad_read indicates that second_quad contains a valid value. This can be used in get_offset_2.

4.4 Remarks

Certain instructions and/or operations are unsuited for implementation in the software model. As a consequence it is possible that in future steps these items will be forgotten. For this reason and for reason of completeness are these instructions mentioned here:

- **RETI**, is modeled but not tested.
- **CIC**
- **UPM**
- **ENAINT**

  Interrupts are modeled by just one possible interrupt and accompanying flag.
- **DISINT**
- **RESET**
- **HALTE**
- **LOCK**

For completeness it is mentioned here that the linked list operand type was not modeled in this program.

The model is tested with numerous small code parts, which are listed in an appendix with code tests. This appendix is not included, since this would not be of any information to the reader of this report. If however so desired the information can be made available.
In this appendix with information on the usage f the Software model are also found:

- The used format for the code-loader and
- The commandline format of the program.

Chapter 5

A high level implementation in HHDL

In the previous chapter the "C"-processor is described by means of the programming language "C", which had certain advantages to it. However one of the disadvantages was the lacking of timing or scheduling information and the fact that the execution of the modules was performed completely sequential. These disadvantages can be overcome by implementing the "C"-processor in HHDL (Silvar-Lisco, Hardware Description Language). This language offers the capability to describe several subprocesses, which will be (virtually) executed in parallel. The language is also able to put timing related information in the external communication actions of the processes, with respect to the expected operation delays.
Next to the above properties, the development system has the means to make a decomposition of large elements into smaller parts, and at a bottom level a functional part can be modeled by a logic circuit. From these functional decompositions, and further decompositions still to be made, the ultimate steps will lead to an implementation on silicon, which this will hopefully render a real and working "C"-processor.

First to obtain a working high level description, an inventory is made of the subprocesses, with their interfaces, to implement. Since there exists a "C"-routine counterpart of almost all modules, there will be no extra formal description of the function of the subprocess. The functions not described in the available "C"-routines are the controller parts. And thus these parts are the parts that might require some extra attention. The problems are to be expected in relating the functioning of the serial flow control in the serial program execution to the now parallel execution of the same actions but then controlled by a specific controller.

5.1 The parts to implement on level 1.

A first level of decomposition follows from:

a) The original decomposition in [Bud88] into high level units. This decomposition only lists the channels of interactions of the modules through their data paths. Controller information is not available other then through thoroughly examining the given implementation of the high level units.
b) The module structure of the "C"-model. The routines in the "C"-program give indications of which modules (read functions and/or routines) interact with one another. It also gives indications of what parameters are passed between the modules.

From this moment on will be assumed that a module will be controlled by modules on a "higher" level. With this higher level is referred to the calling sequence as they can be found in the "C"-interpreter. A compromise between using the exact signals and using all general busses is made. This to prevent too much iterating between adjusting the high level decomposition signals with its data paths and control lines and the implementation of the high level modules. (Also known as the 'Yo-Yo' effect.) The tradeoffs in this matter are made on an *ad-hoc* basis.
The figure with the level 1 decomposition (figure 5.1) shows the following used modules:

- Module: `REGISTERS`

  Interface:

  `IN_C` input integer
  For loading one of the **`registers`**.

  `OP_CMD` input record
  `INSTR_CMD` input record
  Gives the next action to perform. The commands can be given by either thez `instruction unit` or the `operand unit`.

  `STAT` output record
  Gives the result of the last action done.

  `OUT_A`,
  `OUT_B` tri-state integer
  The registers values given on command. It is possible to have different values on the outputs.

  `INSTR_PC` output record
  The `instruction unit` has through this bus the current value of the program counter ready and available for instruction (pre)fetches.

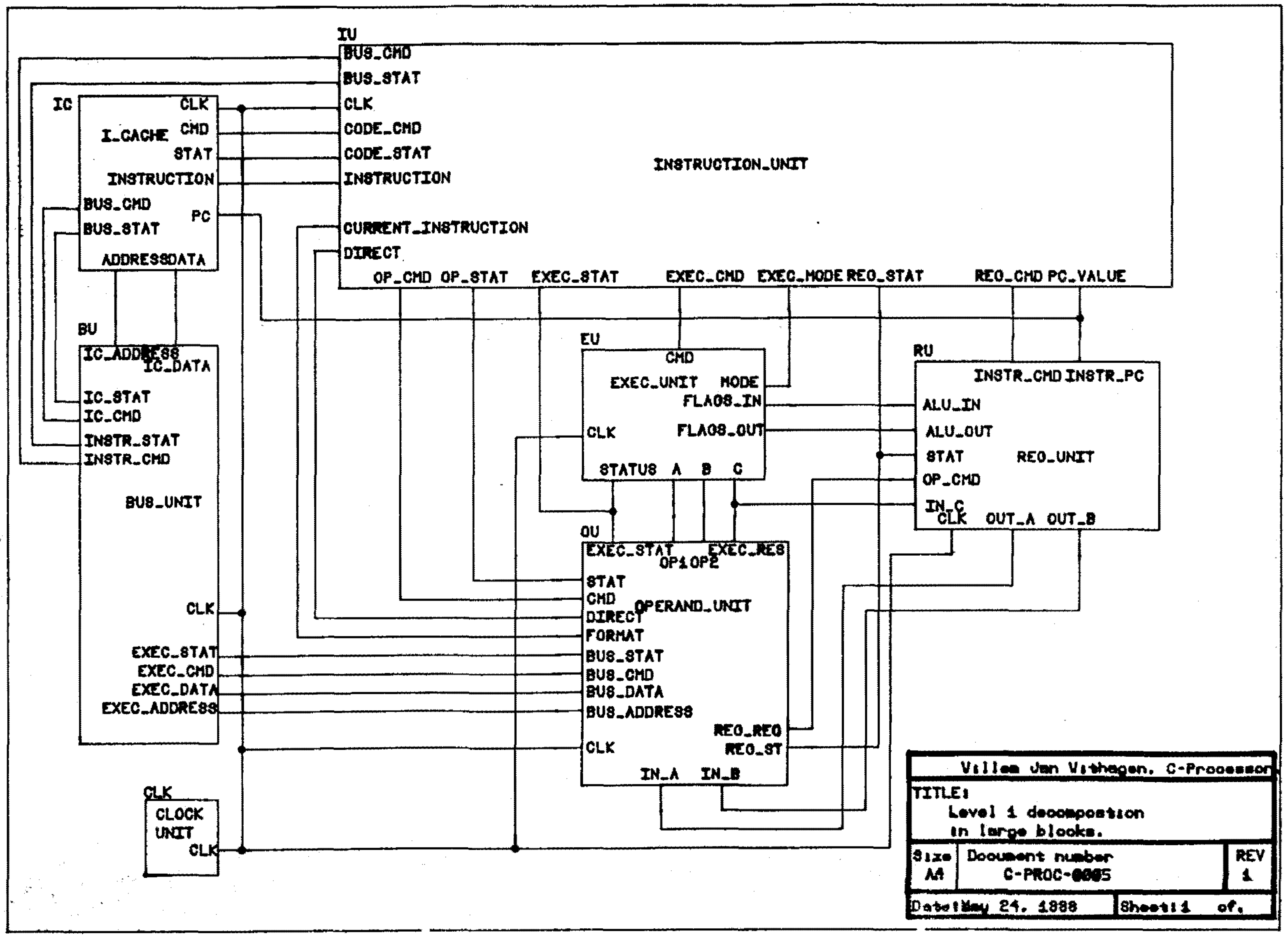

Figure 5.1: Level 1 decomposition in smaller blocks.

ALU_IN	input	record

The flags to be saved, created by operations in the ALU, by previous instructions. Not all instructions modify the Flags.

ALU_OUT	output	record

Previous saved Flags are always direct available to the ALU.

Memory:

The registers FP, EFP, SP, PC, Flags, TEMP_SAVE.

Operations:

Load the registers.
Write the registers to the buses.
Write variants of the SP to the buses.
Increment or decrement the SP.
Increment the PC.
Set or Reset specific Flags.

Comments:

Due to the totally distinct actions for the FP, EFP and the TEMP_SAVE versus the SP, PC and/or Flags, a new decomposition for the register module will be made.

- Module: EXEC_UNIT

Interface:

A,		
B	input	integer

The values of the operands to be acted on.

C	output	integer

The resulting value after calculation is done.

CMD	input	record

Schedules the activity of the module.

MODE	input	record

Gives the format in which the action is performed.

STATUS	output	record

Gives the result of the last action done.

FLAGS_OUT	output	record

Direct available value (status) of the result of the execution's Flags.

FLAGS_IN	input	record

The **Flags** input, with the result from previous executions, for use during the current execution.

- Module: OPERAND_UNIT

Interface:

OP1,		
OP2	output	integer

The values of the operands to be acted on.

EXEC_RES	input	integer

The resulting value after calculation is done.

CMD	input	record

Schedules the activity of the module.

MODE	input	record

Indicates the different types of operands to be fetched. This is dependant of the instruction type under execution.

STATUS	output	record

Gives the results of the last action done.

FORMAT	input	integer

Gives the type of F-operands to be fetched. This is a part of the current instruction, and contains the operand formats, and possibly a 1-byte datafield.

DIRECT	input	integer

During the fetching of the operands extra quads of the instructionstream will be passed through this channel.

`BUS_CMD` output record
Commands for the `bus-unit` in case the `operand unit` needs to fetch data from external memory.

`BUS_STAT` input record
The status result from a command issued to the `bus-unit`.

`BUS_ADDRESS` input integer
The address on which the `bus-unit` needs to put or get data.

`BUS_DATA` bidirectional integer
The data to be transported to and from the `bus-unit`.

- Module: `BUS_UNIT`

Interface:

`IC_CMD` input record
Requests for new data from the `instruction cache` are given through this path.

`IC_STAT` output record
Resulting status on commands from `instruction cache`.

`IC_ADDRESS` input integer
The address to fetch the instructions from.

`IC_DATA` output integer
The new instruction read.

`INSTR_CMD` input record
General commands given by the global control unit `instruction_unit`.

`INSTR_STAT` output record
Status returned to the `instruction_unit`.

`EXEC_CMD` input record
Requests for reading or writing of data into main memory.

EXEC_STAT output record
Status to be returned to the operand_unit.

EXEC_ADDRESS input integer
Address of the memory location to be read or written.

EXEC_DATA bidirectional integer
The data to be read or written. This will be data which is flushed out of the cache, reread into the cache. Or those items that are outside the cache window.

- Module: INSTRUCTION_UNIT

Interface:

BUS_CMD output record
Commands for the Bus_unit.

BUS_STAT input record
Status of the Bus_unit.

CODE_CMD output record
Commands for the Instruction_cache.

CODE_STAT input record
Status of the Instruction_cache.

INSTR_ADR output integer
The address to fetch the next instruction QUAD from.

INSTRUCTION input integer
The result of an instruction fetch by the instruction cache.

CURRENT_INSTRUCTION
output integer
The instruction quad that is currently being executed by the instruction_unit.

DIRECT output integer
The value of an extra quad, read by the instruction_unit.

OP_CDM	output	record

Scheduling commands for the operand_unit.

OP_STAT	input	record

Status of the operand_unit.

EXEC_CMD	output	record

Scheduling commands for the execution_unit.

EXEC_STAT	input	record

Resulting status of the execution_unit. It might also contain things like flag information.

EXEC_MODE	output	record

Mode information which types the action to be executed. It contains the operations (add, ... , mult) and the type of the operation. (signed $\Leftrightarrow$ unsigned, logic $\Leftrightarrow$ boolean, carry $\Leftrightarrow$ no_carry)

FLAGS	input	record

Status of the flags_unit. Contains information concerning the values of the flags.

REG_CMD	output	record

Commands for the register_unit.

REG_STAT	input	record

Status results of the register_unit.

PC_VALUE	input	integer

Direct input from the programcounter. Usually this will be the address of the instruction under execution.

5.2 A next decomposition for the Registermodule.

The register module contains six distinct items. These six items will be the next level of decomposition for the register module. Thus the register module will consist of:

- A stackpointer.
- A framepointer.

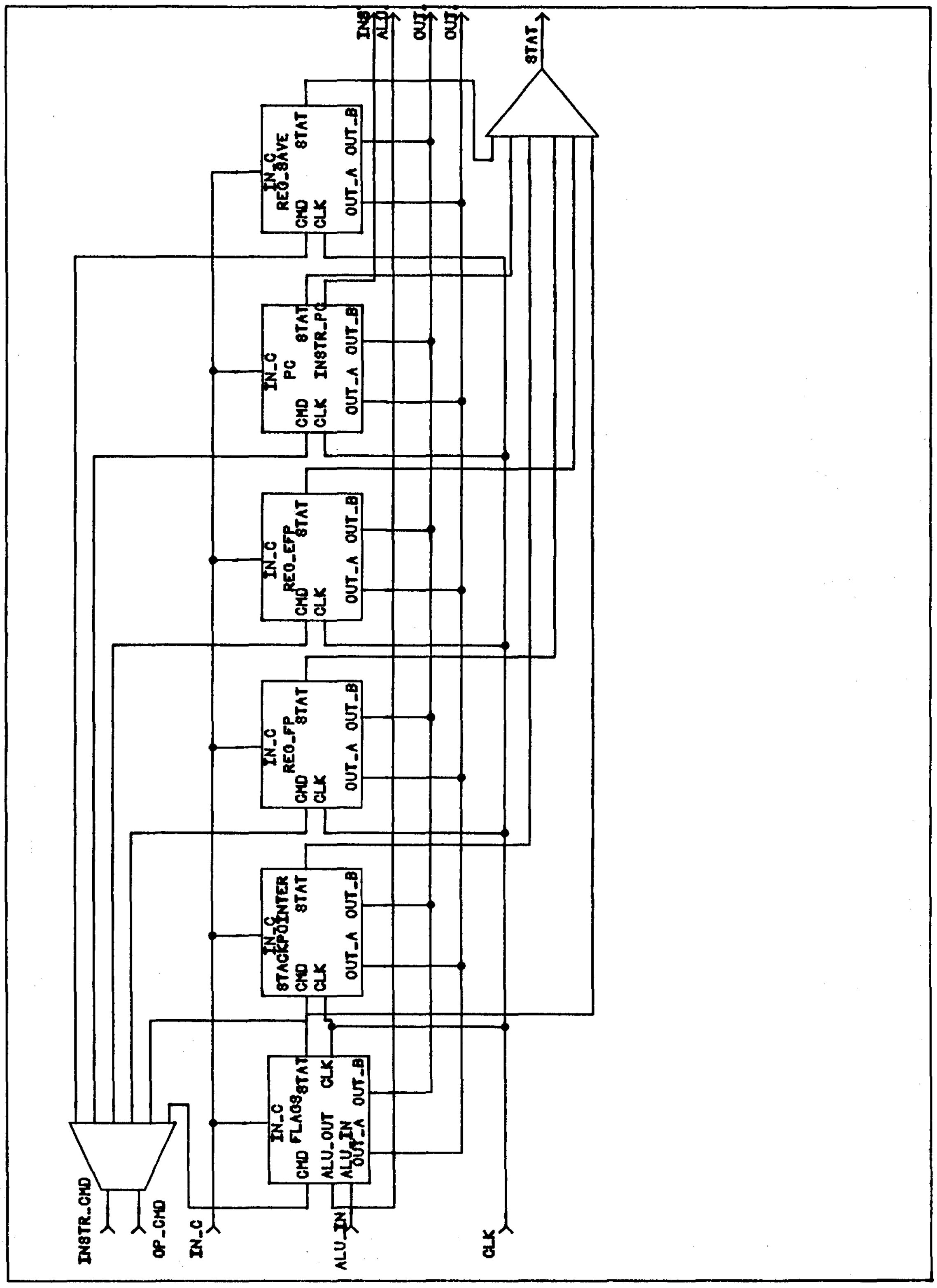

Figure 5.2: The register module decomposition.

- An external framepointer.
- The program counter.
- A flags register.
- A temporary save location.

Next to these primary modules, two additional modules are implemented. They will translate the control and status signals of the individual registers into the control and status signals available on the exterior the registers module. All commands are mapped into one integer bus, as are the status results. Although the given high level implementation is equipped with a complete handshake interface ,it will be the intention to execute this module in one clockcycle.
Handshaking is of the very simple type:
COMMAND $\Rightarrow$ BUSY $\Rightarrow$ READY $\Rightarrow$ IDLE
Hence the synchronization might not be needed in future. On this level of description is however no clock used as reference and thus is it a prerequisite of every module to report its completion status. In lower level descriptions this might be changed as soon as a clocking system is introduced.

The following sections will describe the implemented registers and "translators" in more detail. The decomposition of the registers module and writing the implementation of the subcomponents was used as a first exercise with the Silvar-Lisco tools.

5.2.1 The stackpointer.

Making an inventory of the operations possible on the SP and of the values to be assigned to the A and/or B-bus can be done using the cross-reference of the SP in all the "C"-modules.

(The items listed give the values to be written on the buses and the internal operation to be performed on the SP.)

A bus	B bus	Operation.
SP	X	SP-= 4
X	SP	SP-= 4
SP-4	SP	SP-= 4
SP	X	—
SP+4	X	SP+= 4
SP-4	SP	SP-= 8
SP	SP-4	—
SP-4	X	—
SP-8	SP	—
SP	SP-4	SP-= 8

Next to the above "complex" operations, will the following basics be implemented: NOP, load, increment, decrement, add value from the C-bus.

The targeted stackpointer module is one which will be able to perform all three given action(s) in the same transition (write to A-, B-bus and execute internal action). This will make it possible to activate some processes in parallel during the fetching of the operands. This is however not yet investigated at the current level of implementation, but there is very little reason to believe that a one-cycle implementation is not possible.

- Module: SP

  Interface:

`IN_C`	input	integer

  loading the SP.

`CMD`	input	record

  Gives the next action to perform.

`STAT`	output	record

  Gives the result of the last action done.

`OUT_A`, `OUT_B`	tri-state	integer

  SP value, with possible adjustments, given on command.

  `Memory:`
  The stackpointer

5.2.2 The framepointer, external framepointer andtemporary save location.

All of these modules are of a very simple type. The only actions are: NOP, load, write to the A and/or B-busses. Moreover: All are identical, except for the values they will contain during operation.

There are no transformations of the contents needed in these registers. The pointer registers are used by the operand module, where they act as base registers during some of the indexing operations. The temporary save location is primarily used for storing a new PC-value before it can be loaded, or the second value used during swap. Either during a value swap of values on the stack, or during a swap of processes.

It is obvious that these registers are no more then a mere block of Flip-Flops and tri-state busdrivers. In future developments they will easily comply with the one- cycle requirement of the register's module.

- Module: FP (or EFP, TEMP_SAVE)

  Interface:

  IN_C input integer

  For loading the register.

  CMD input record

  Gives the next action to perform.

  STAT output record.

  Gives the result of the last action done.

  OUT_A,
  OUT_B tri-state integer.

  The register value written on the bus.

  Memory:

  Framepointer
  External framepointer
  Temporary save location

5.2.3 The program counter.

The processor uses the programcounter to address the code to be executed. Next to this is it also possible to use the programcounter for reference purposes. Be it inline constants, or other fixed data items. And furthermore is it possible to load the program counter with new values to influence the flow of the program. To be able to access the program counter, without disturbing the calculation of operands, the register is equipped with a direct output bus which will always contain a the latest loaded, and hopefully the correct, address for fetching the newest code.

- Module: PC

  Interface:

  IN_C input integer

  For loading the PC.

`CMD`	input	record

Gives the next action to perform.

`STAT`	output	record

Gives the result of the last action done.

`INSTR_PC`	output	integer

direct available value of the current PC.

`OUT_A`, `OUT_B`	tri-state	integer

The register value written on the bus on command.

`Memory:`

The program counter.

5.2.4 The Flagsregisters.

During execution of arithmetic, logic and boolean instructions a new result is calculated. The result of the calculation can have certain properties which are kept in the **Flags**register. The processor in itself has also status items which are used and/or modified by the user. This status flags are also kept in the **Flags**register.
Currently are the following properties maintained:

Result flags:

1. Zero.
2. Carry.
3. Overflow.
4. Sign.

Status flags:

5. Interrupts enable.
6. User mode.

The **Flags** module has the following connections:

- Module: Flags

  Interface:

IN_C	input	integer
For direct loading the Flags.		
CMD	input	record
Gives the next action to perform.		
STAT	output	record
Gives the result of the last action done.		
ALU_IN	input	record
Result input directly from the EXEC_UNIT where the flags are calculated.		
ALU_OUT	output	record
Direct available value of the current Flags.		
OUT_A, OUT_B	tri-state	integer
The value of the Flags given on command.		

  Operations:
  Load the flags from the C-bus.
  Load the flags from the ALU.
  Write the contents of the flags to the buses.
  Set or reset specific flags.

5.2.5 Command and status translators.

The HHDL is familiar with the concept of a translator. But since the translators in the register module are slightly different, they are implemented as regular modules with ZERO delay. If the translators have to be kept, in a lower level description, this can be changed, and then the translators will become physical parts of the circuit. Be it combinatorial parts.

It would be most convenient to describe the command and status "busses" as regular PASCAL records. But the current version of the HELIX-software has problems assigning to record structures, na furthermore has HHDL no support for a record of record type structure definition. The busses will be, as a consequence of this, of integer type. The commands are bitwise encoded in the integer. The exact coding and encoding can be found in the module that defines the command and status busses for the registers. For completeness it is noted that only those bits of the "integer connection" that are used will actually be created on chip. For an enumeration of the assignments possible is referred to the HHDL-listing of the translators.

5.3 Decomposition of the operand unit.

The software description of the "C"-processor gives a clear indication that the `operand_unit` is not a very simple unit. Even in the software model the unit is already divided into several functional parts that cooperate. Therefore is the `operand_unit` in de HHDL also expanded into several subcomponents, on the first hardware description level. The components, with their interfaces, are listed below and are in accordance with their software model counterparts. Some of these interfaces will connect directly to the interface on the higher level, some others will connect to the subcomponents of the `operand_unit`.

The purpose of the operand unit is to prepare the operands in the instruction, as indicated by the operand formats. After they are fetched, they are operated on by the execution unit. And finally the result from the execution unit is again presented to the operand unit. This unit has setup the destination for the result, in the previous stage. This could be an address in memory, but registers are also valid targets.

The operand unit does all of the work if the instruction is a F(ull) or a M(ixed) instruction. The instruction QUAD is passed to the operand unit as a whole and the operand formats are completely decoded in the operand unit. If the operand unit needs more QUADs from the instruction stream it can ask the instruction unit for more QUADs.
Is the instruction however a P-instruction then the fetching of the operands is assisted by the instruction unit. The registers addressed implicitly by the instruction, possibly the SP or the FP, are then controlled by the instruction unit. If the operand requires either an offset or immediate data (both of 8 bit size which is included in the P instruction), then the instruction unit puts this byte in the lower byte field of the `current_instruction` bus. Through this channel the extra value can be retrieved by the `get_offset_2` unit for operand 2.
During the construction of operand 1 the `get_operand_1` module will also create the new destination for the result, if any is needed. The `store_result` unit shall then be used to store the result, when commanded to do so.

5.3.1 The cache window.

The cache window has to manipulate the dedicated memory projected in this module. The objective of this special cache is to have the last used frame ready and available, and this all at the highest speed possible. Currently the real contents of this module is still under study. And the actual implementation is in this version no more than just a gateway to

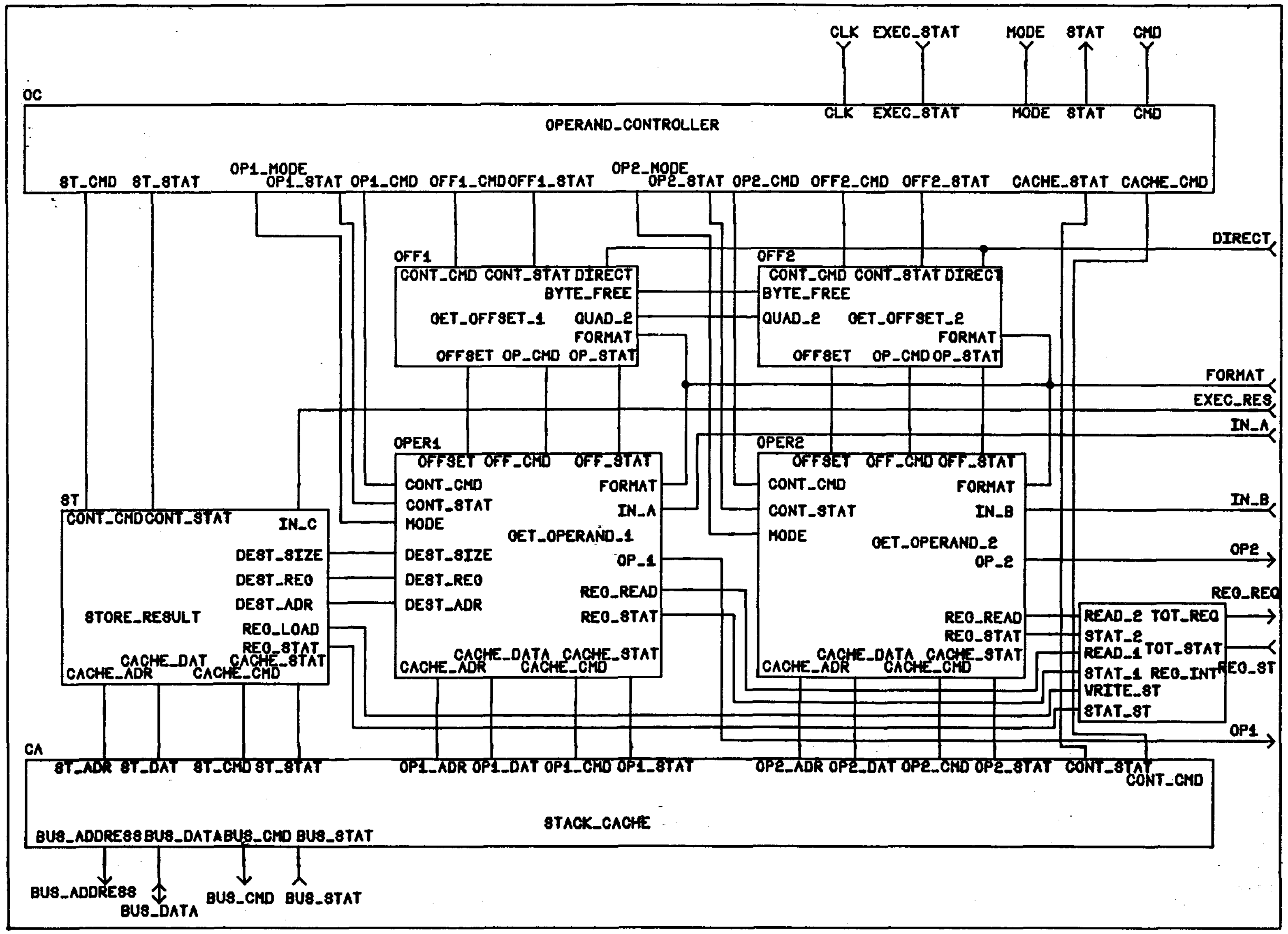

Figure 5.3: The decomposition of the operand unit.

the bus unit. Items read or written are traded with the bus unit, without any further manipulation other then putting words and byte at the proper place in a QUAD.

One essential function the unit does perform is arbitrating the sequence of accesses to the memory. It is very likely that this will not be needed in future. At least not for the accesses to the cache window. The cache window then has to be implemented as a three ported memory:

- Two for reading, (at random addresses)
- One for writing. (at possibly one of the read addresses)

The cache window could be implemented as a straight "of the shelf" linear cache. This is probably the simplest implementation, but also the next to poorest in performance. (The poorest would be no cache at all.) One of the more advanced methods would be to implement the cache as a real sliding window on the stack. This could make the top elements be very fast addressable, with a further possibility to address elements at random in the current stack window. The cache would than be "wrapped around" once the window grows out of the cache. One step further would be a cache like the previous one but it is supervised by a "storage scheduler". This scheduler process would take care of preparing the cache for future changes of the stack, either up or down. Not yet saved data QUADs in the cache, which are liable to be used next if the stack grows, should be saved first. Values that are needed in the near future when the stack shrinks should be loaded first. The author is aware of the fact that the last suggestion is simple one to write down, but it is a lot harder to design such a device.

Once the cache is implemented as a memory module it should also be possible to flush the data which was stored in it. For this purpose only a special instruction is created, and this is represented in the modules by a specific command. This commands is passed onto the `stack_cache`, through the operand controller.

- Module: `STACK_CACHE.`

  Interface:
  (* = ST, OP1, OP2)

  `*_CMD` input record
  Through these lines will the cache be informed about requests for data. It is possible that the command will also contain information on the size of the requested operand.

  `*_STAT` output record
  These lines will report the current status for a requested action.

  `*_ADR` input integer
  The address on which the data request has to act. This address need not be aligned on a quad.

`*_DATA` output (for OP1, OP2)
input (for ST)
data to be transported during the request.

`BUS_CMD` output record
Through these lines will the cache be informed about requests for data. It is possible that the command will also contain information on the size of the requested operand.

`BUS_STAT` input record
These lines will report the current status for a requested action.

`BUS_ADR` output integer.
The address on which the data request has to act. This address need not be aligned on a quad.

`BUS_DATA` bidirectional integer
The data to be transported to the main memory during a request.

`CONT_STAT` output record
`CONT_CMD` output record
The interface with the `operand_controller`.

5.3.2 The operand fetchers.

Both operand fetchers are much very alike. The two differences are:

1. The source of the operand formats for F-operands. Which varies only in the byte of the instruction word.
2. The destination for the result is set by the `get_operand_1` unit.

First the common elements of their structure are enlightened, then the calculation of the destination parameters is discussed.
Fetching an operand can be subdivided into two classes:

- Fetching F-operands
- Others.

To start with the more simple case (seen from the operand fetcher), the others. This class contains the fetching of the operands for the different P-instructions and some of the

M-instructions. Through the mode lines is indicated what type of instruction currently is executed. The control of the registers in this case is handled by the instruction controller on the highest level. The A- and B-bus will contain the required addresses.

Get_operand_1 is able to acquire the first operand with this address. It also has to create the destination for the result from this address.

Get_operand_2 can be an 8-bit immediate value, or could need an offset to be added to the framepointer. This value is extracted from the instruction by the instruction controller and passed on through the offset fetcher to **get_operand_2**. This value is either used as operand #2 or is used to construct the effective address for operand #2. Which is then fetched from the stack.

Is the instruction an instruction with F-operands? Then all the work is done in the operand-unit, except the fetching of a new QUAD from the instruction stream. Dependant on the instruction type one or two operands are required. This property of the instruction is passed onto operand controller from the instruction unit. The operand controller passes this information onto the operand fetchers. The operand fetchers decode their format byte which contains the operand format applicable to them. If offsets are needed, **get_operand_1** is allowed to execute first, then if needed **get_operand_2** fetches it offset. Currently all this is implemented in a complete sequential order, this could (or should) however be changed in the future.

The decoding of the format is done by extraction of the fields in the operand descriptor. Then the actual execution of the operand fetch is done with several large case statements in which all possible variants of the formats are coded. The interfacing with either the stack cache or the registers is done autonomously. On completion the operand controller is signaled and the module returns to an idle state.

The creation of the destination.

The operand fetcher for the first operand has to perform an extra task when compared to the **get_operand_2** module. This task however is usually a simple one. When the operand is an F-operand the destination is exactly alike the source operand. Thus if the source was a register then the destination will be that same register. Was the source a location in memory, be it main memory or the stack window, then the destination is that same location in memory.

It is more complicated with the P-operands. Here has to be taken into account whether the stack is pushed or popped. This information is however passed onto the operand fetchers, by specific operand commands. **Get_operand_1** then selects the new stack address in one of the branches of a case statement. The case selector picks the type of operation and the destination address.

Again another case are the M(ixed)-instructions. With these instructions more manipulation of the data is involved. When the normal destination options are not sufficient, the modules are manipulated "manually" by the instruction unit. For this some extra operand commands are introduced.

- Module: GET_OPERAND_1.

- Module: GET_OPERAND_2.

  Interface:

CONT_CMD	input	record
MODE	input	record
CONT_STAT	output	record

  Through these channels the operand_controller will steer the `get_operand` unit.

IN_A(B)	input	integer

  If the operand is one of the registers, or a register is needed during the calculation of the operand, this input will supply the register's data.

OP1(2)	output	integer

  Once the operand is calculated, it is passed on to the `execution_unit` through this path.

FORMAT	input	integer

  This will contain the part of the instruction which contain the operand format.

CACHE_ADR	output	integer
CACHE_DATA	input	integer
CACHE_CMD	output	record
CACHE_STAT	input	record

  This is the interface with the `cache_unit`. This channel will be used to fetch the first (second) operand, if it is to be found in the cache window or the main memory.

OFF_CMD	output	record
OFF_STAT	input	record
OFFSET	input	integer

  Above three lines are the interface to the unit that will get the offset. On conclusion of the action OFFSET will contain the resulting value.

REG_READ	output	record

REG_STAT	input	record

The two lines that will able the get_operand unit to fetch the register values needed during the calculation of the operand.

For the get_operand_1 unit only.

DEST_SIZE	output	record
DEST_REG	output	record
DEST_ADR	output	integer

These lines are only available on the **get_operand_1** unit. They indicate the place where the result of the instruction should be stored. This is a direct interface to the **store_result** unit.

5.3.3 The offset calculation units.

When using F-operands, and also with certain P-instructions, offset or immediate data is used during the construction of an operand. Separate units are created for calculation of the offset or immediate value. Both the **get_operand_1** and the **get_operand_2** unit have their own offset fetcher. Though both modules are more or less alike they differ enough to justify the creation of two distinct modules.

Get_offset_1 uses information from the instruction controller that indicates whether the instruction is an one or a two operand instruction. This information indicates the place in the instruction for fetching a offset of word size. Furthermore it sets flags which inform the **get_offset_2** module of the used parts of the current instruction stream.
With the flags constructs **get_offset_2** the offset for the operand 2 fetcher. After completion the module involved informs the operand fetcher and the operand controller of its current status.

The value returned to the operand fetcher is completely processed. The resulting value is a signed 32 bit integer, sign extended and data extracted according the information passed on by the **get_operand_1(2)** unit.

- Module: GET_OFFSET_1.

- Module: GET_OFFSET_2.

  Interface:

CONT_CMD	input	record
CONT_STAT	input	record

For direct control by the operand_controller

FORMAT	input	integer

The part of the instruction that contains the operand formats, and a possible 1 byte size offset.

DIRECT	input	integer

During the construction of the operands extra data from the instructionstream could be needed. This action will be executed by the **instruction_unit**, and the result will be placed on the DIRECT lines.

BYTE_FREE		integer
QUAD_2		integer

The above two lines are a means of communicating the status of the used instructionstream. They are outputs on **get_offset_1** and input on **get_offset_2**.

OFF_CMD	input	record
OFF_STAT	output	record
OFFSET	output	integer

Above three lines are the interface to the **get_operand** unit that requires the offset. On conclusion of the action OFFSET will contain the result.

5.3.4 The store result module.

Once the operations on the operands are done, the "calculated" result has to be stored at the indicated destination. The destination is either implied by the instruction: P-instructions. Or the destination equals the source of first operand. The store module however does not calculate this address itself, it is obtained from the unit that fetches the first operand.

The result has either to be stored in memory or in a registers. Using the local frame as destination would place the result in the cache window. But it could also be place in the global frame or in the external frame as a consequence of the reference to the external framepointer, in which case it would be placed in "real" core. In these last two cases the store unit propagates the result to the stack cache unit to be stored, be it in the cache window or the main memory.
Is however the destination a register than the result is placed into the registers via the C-bus. The only action taken by the store unit is the handshaking with the register, under control of the operand unit controller.

- Module: STORE_RESULT.

Interface: CACHE_ADR	output	integer
CACHE_DATA	output	integer
CACHE_CMD	output	record
CACHE_STAT	input	record

This is the interface with the **cache_unit**. This channel will be used to write the result of the operation, if it is to be placed in the cache window or the main memory.

DEST_REG	input	integer
DEST_ADR	input	integer

These lines are only available from the **get_operand_1** unit. They indicate the place where the result of the instruction should be stored.

IN_C	input	integer

The result from the execution unit that is to be stored.

5.3.5 The Operand unit controller.

The operand controller is actually quite a simple model. This could of course change in future when the sequence of execution is going to be more complex.
All the operand controller has to do is: Pass parameters to the controlled modules, schedule the activation of the subordinate modules, wait for their completion, and return the status of the operand unit as a whole.
Once the modules are activated, a request could be made for more QUADs from the instructionstream. This request is passed on to a higher level, which will indicate when the request has been serviced. This indication of completion is passed on to the requesting party, which then continues its operations, until finished.
The interface of the module is:

- Module: OPERAND_CONTROLLER.

Interface:
For all other modules, described above, in this subcomponent there are the following signals. These signals are also described above.

*_CMD	output	record
*_STAT	input	record

To the two **get_operand** modules:

OP1(2)_MODE	output	record

Next to that are the interface signals to a higher level (the instruction unit and de execution unit):

EXEC_STAT	input	record

Information about the completion of the **execution_unit**. Currently not used, but could be needed when in future the operand unit is allowed to store a result once the execution unit has completed its actions.

STAT	output	record
CMD	input	record
MODE	input	record

To control the **operand_unit** controller.

5.3.6 The interface to the registers module.

The operand fetchers, and the **store_result** unit, require the contents of the registers, for certain instructions. As explained in a previous paragraph the commands for the registers are of a rather complex structure, so a translator would be appropriate.
The **reg_int(erface)** translator accepts a request for one register, and transforms it into a command on the register commands lines. Once the status of the commanded register is returned, this status is passed back to only those modules that are currently addressing this register.
One advantage of the requests made by the **get_operand** units, and the **store_result** unit, is the fact that the type of the issued commands are the simple types. They are either write requests for the busses, or a direct load from the C-bus, nothing more.

The connections to this module are:

- Module: REG_INT.

  Interface to the operand modules:

READ_1(2)	input	record
STAT_1(2)	output	record

  And to the **store_result** module.

WRITE_ST	input	record
STAT_ST	output	record

  The above three control sets have different naming to suggest their connection. The main difference lies technically in the fact that the READ_* lines indicate the register to be written on what bus. The WRITE_ST indicates into which register the result has to be written.

The interface to the register module:

`TOT_REQ`	output	record
`TOT_STAT`	input	record

The packed signal lines to indicate one or more requests to the registers, and to obtain the results of the issued commands.

5.4 The execution unit.

The execution unit consists of more then what usually is denominated as ALU. In the current case this certainly would not be complete. Functionally the execution unit comprises an ALU, a shift and extract unit and a multiplexer.
The ALU is used for the arithmetical and logic operations, ranging from a simple add to a 32 by 32 signed division.
The shift and extract unit is used to manipulate a data word by shifting it signed or unsigned, left or right, or by extracting a partial field and adjusting it to the least significant bit.
The multiplexer gates one of the operands to the result bus.

Currently all three partial modules are programmed in one HHDL module, and are not split into smaller sub blocks. This could be a next step in decomposition. The commands issued to the execution unit activate the subparts. For the ALU is extra information regarding the function to perform passed on the mode lines. The other two units are directly controlled through the commands issued.

During arithmetical and logic operations the flags output is updated to represent the status of the current value on the result bus. When however a new result is obtained the previous flags are not kept unless saved in the flags register in the register module.

The current software implementation of the execution unit does not give any further details regarding the kind of hardware implementation to use. All instructions are executed by more or less standard **PASCAL** statements, for shifts the HHDL routines are used. Due to the complete handshaking of all modules there are no time limits placed on the execution time of this unit. However this will not be the case when the design is in next steps of design. Then more profound algorithms have to be implemented to prevent a bottleneck in the execution unit. This is of course more important, time wise, for multiply, divide and modulo operations than it will be for add or subtract.
The interface definition of the execution unit:

- Module: `EXEC_UNIT`.

  Interface to the operand module:

A, B	input	record
C	output	record

Interface to the FLAGS register in the register_module.

FLAGS_OUT	output	record
FLAGS_IN	input	record

And the complete set handshake and mode signals.

CMD	input	record
MODE	input	record
STAT	output	record

The above three control sets have different naming to suggest their connection. The main difference lies technically in the fact that the READ_* lines indicate the register to be written on what bus. The WRITE_ST indicates into which register the result has to be written.

The interface to the register_module:

TOT_REQ	output	record
TOT_STAT	input	record

The packed signal lines to indicate one or more requests to the registers, and to obtain the results of the issued commands.

5.5 The instruction cache.

The instruction cache resembles the stack cache in the current state of implementation. It is nothing more than an empty funnel through which instructions are passed from the bus unit to the instruction unit.
The instruction cache will service a request from the instruction unit for another QUAD in the instruction stream. If this QUAD is not already available in the onboard memory of the cache, a request will be issued to the bus unit. Which will then fetch the requested QUAD from main memory.
For this cache a linear cache is probably a very good candidate. Memory accesses for instruction fetches usually categorize within the principles of locality. The only problem will be the determination of a optimal size. Not to big since this will be a waist of silicon, but not to small or otherwise the number of misses will be too large.
A more elaborated cache method however could be devised. Since execution sequence of code can be predicted fairly simple, most likely the next instruction, a (small) prefetch queue would be an enhancement with very low costs in terms of silicon. (if compared with the bulk of a linear cache)

- Module: I_CACHE.

Interface to the bus unit:

BUS_CMD	output	record
BUS_STAT	input	record
ADDRESS	output	integer
DATA	input	integer

Interface to the instruction unit:

CODE_CMD	input	record
CODE_STAT	output	record
INSTR_ADR	output	integer
INSTRUCTION	output	integer

The set of signals to handshake the data exchange with, the lines to indicate the **Quad** with, and the lines to retrieve the **Quad** through.

And the complete set handshake and mode signals.

CMD	input	record
MODE	input	record
STAT	output	record

5.6 The bus unit.

The bus unit is the unit that takes care of data transfers to and from the main memory. The two units that will require this service are: The instruction cache, for fetching instructions. And the stack cache in the operand module. The stack cache will use the memory as the bottom end of the stack. Every thing that "falls out of the bottom" of the stack will be written into main memory. And when the stack decreases the previously saved data will be reloaded into the stack.

The side which is not deployed in this thesis is the I/O interface to the external world. This part of this module has to interface with the rest of the system in which the "C"-processor will be used. No thought was given to a possible implementation of this part of the module. The advantage of this is that no dependencies for this are build in the code. On the other hard are there no restrictions made which would be required for an certain interface. There are however some considerations taken into account, which are no more that general. The instruction set contains an instruction that will activate the external reset pin (is found on the Motorola series 680XX). An instruction is available to control mutual exclusion of the external system buses. A Halt instruction will put the processor into a sleeping status, refraining from any activity on the buses. These instructions are, when issued, at the moment absorbed in the bus unit, and they have no effect what so ever.

As usual the bus unit interfaces with its master controller and the modules requiring service through complete handshake channels. paths are available along which the addresses and the data are passed. The data connection between operand module and bus unit is a bidirectional one, the one with the instruction cache is unidirectional.

The general controller, the instruction unit, is the main controller of the bus unit. In general this will mean that the bus unit is started once, and will only be stopped to actuate commands like LOCK, RESET or HALT. Other internal reasons for stopping bus accesses are at the moment not programmed.
The interface definition:

- Module: `BUS_UNIT`.

Interface to the operand module:

`EXEC_CMD`	input	record
`EXEC_STAT`	output	record
`EXEC_ADDRESS`	input	integer
`EXEC_DATA`	bidirect	integer

Interface to the instruction cache:

`IC_CMD`	input	record
`IC_STAT`	output	record
`IC_ADDRESS`	input	integer
`IC_DATA`	output	integer

Interface to the FLAGS register in the register module:

`FLAGS_OUT`	output	record
`FLAGS_IN`	input	record

And the complete set handshake and mode signals:

`CMD`	input	record
`MODE`	input	record
`STAT`	output	record

5.7 The instruction unit.

When we take a look at the top level design of the "C"-processor, can visualize the instruction unit as a spider in a web. All of the other modules are controlled by the instruction unit. And this clearly states the purpose of the instruction unit: To pull the strings and be the general manager of data and control flow. All subunits have a control interface with the instruction unit:

- A command line with one or more COMMANDs and STOP.
- A status line with at least BUSY and READY.

The only unit that is controlled in no other way than the above is the bus unit. The bus unit takes the actual action commands from either the instruction cache or the stack cache from the operand unit. All other units have an interface to the instruction unit that will also indicate required modes of operation.

The instruction cache receives requests for instructions, from the instruction controller, located at the indicated addresses. On a miss of the cache the value will be fetched from main memory. An additional command to be issued is the CLEAR command. This command follows from the Mcic instruction.

The register unit is directly controlled during the P-instructions and some of the M-instructions. During F-instructions the register unit is controlled by the operand unit only.

The execution unit receives the mode of operation from the instruction unit, This mode will set the function of operation to be done, and what the type of the operation is. I.e. operation = add, type = signed. Currently this only holds for ALU operations, the shift and multiplex operations/commands are indicated through the command line. The operand unit receives the type of operands it has to fetch through its command line. Once activated with a certain command, it will autonomously fetch the operands. If the instructions are P-instructions then the register unit is controlled directly by the instruction unit, and the correct registers values are already available on the A-bus and B-bus. Certain operands require more QUADs from the instruction stream. In this case the status of the operand unit will indicate a request which has to be serviced by the instruction unit. This communication is again performed by means of a fully handshaked data exchange.

The software model makes a clear distinction between the part of the instruction unit that fetches the instructions and the part that decodes and executes the instruction/opcode. The first one, the `INSTR_FETCH`, gets the next instruction to be executed, and does a preliminary decode of the instruction. This determines whether the instruction QUAD contains one or more P-instructions or one other instruction. In the later case the opcode part of the instruction is passed onto the instruction decoder, and the remaining part of the instruction is put on the `current_instruction` lines. In case of P-instructions the P-opcode is written to the instruction decoder and the next byte is put on the LS Byte of the

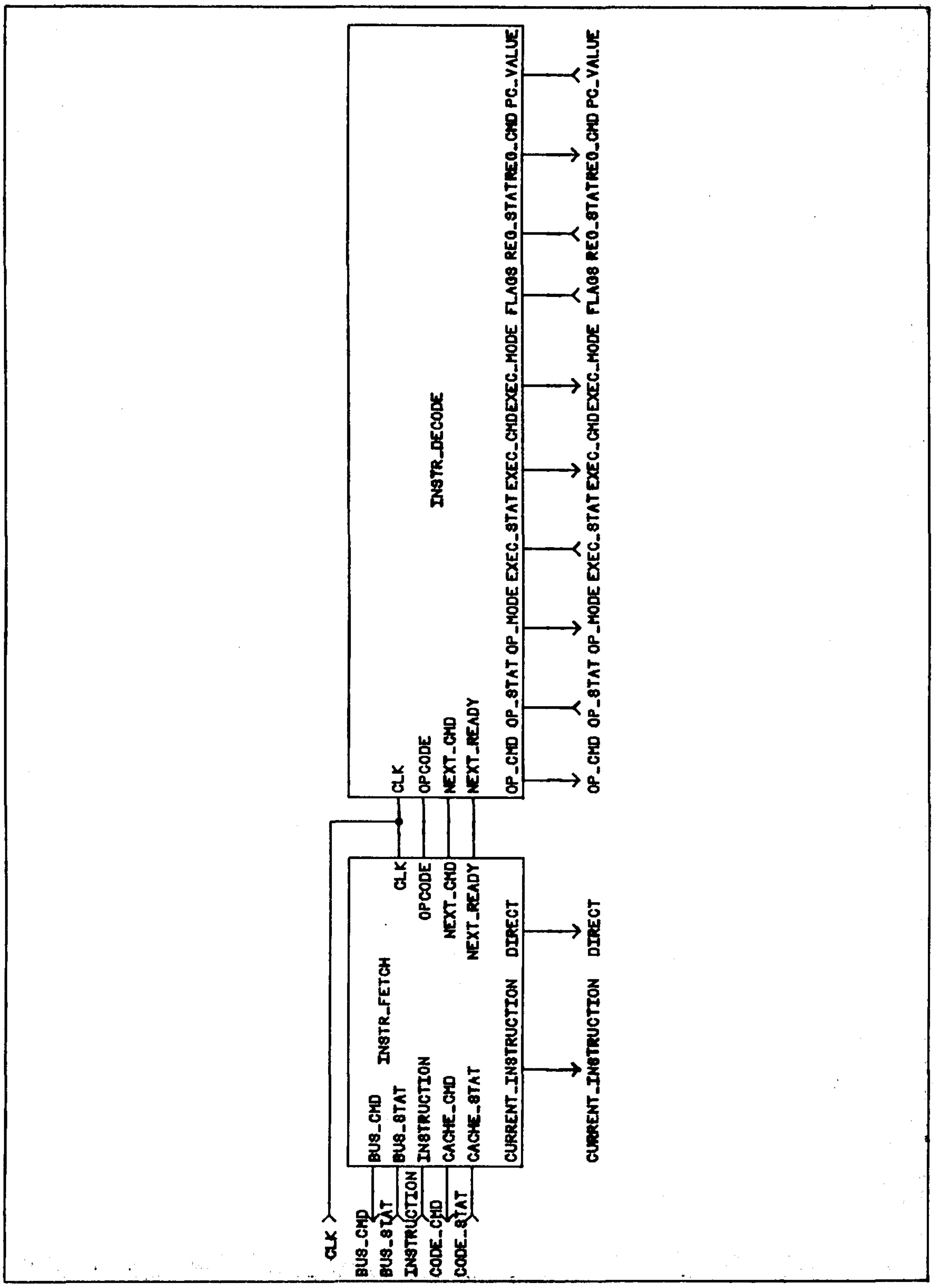

Figure 5.4: The Instruction decoder decomposition.

`current_instruction` lines as a possible operand in Immediate of FP relative instructions. This way all opcodes are passed on to the decoder. If a byte in the P-instruction QUAD is used as an operand than this is reported to the instruction fetcher through the signal `operand_used`. In these cases the instruction fetcher will skip the byte used as operand. The instruction fetcher will also test for NOP fillers. These are not passed onto the decoder, but the next opcode from the next QUAD is fetched.

The transport of the opcodes from the instruction fetcher to the decoder occurs, as in all other cases, under control of the standard handshake cycles. If the instruction has any actions to undertake on the bus unit, than this is communicated to the instruction fetcher as a request on the `next_ready` status lines, again with the appropriate handshaking.
During the execution a F(ull) or a M(ixed) instruction the operand unit might require extra instruction QUADs to calculate the operands. If this is the case, a request is issued through the operand status line, `op_stat`, to the instruction decoder which passes it on to the fetcher. The fetcher in its turn gets the next instruction QUAD from the cache and puts it on the direct lines. It furthermore reports the termination of the request to the instruction decoder. And this module will pass this information on to the operand unit. Where it will be passed on to the requesting operand submodule.

Next the two sub modules of the `instruction_unit` are discussed in more detail.

5.7.1 The instruction fetcher.

The instruction fetcher is the simpler one of the two modules in the instruction unit. It prepares the instruction QUADs for execution by the instruction decoder. For this purpose are the instructions classified into two sets: P-instructions and others. If the QUAD contains P-instructions the QUAD is subdivided into four BYTEs, which are each in sequence handed over to the decoder. The other instructions have the opcode stored in the MS BYTE. During the execution of these P-instruction only this MS BYTE is passed on to the decoder.
During the execution of a P-instruction an extra BYTE can be required for an immediate operand or an FP relative addressing. In these cases the BYTE, used as data or offset, will be skipped and not used for execution. And the next BYTE to the data is transferred to the decoder.
The instruction fetcher fetches the instruction QUAD from the address that is held in the programcounter in the register module. A special output port has been created on the register module, solely for this purpose. The PC register is updated by commands from the decoder which is instructed to do so by the instruction fetcher. When the instruction QUAD contains one or more P-instructions then the PC is incremented only at the execution of the first P-BYTE. The other instructions (F(ull) and M(ixed)) require an increment with every executed instruction.
This module also forces the reset of the micro-processor as a whole. Upon reset the fetch unit forces a Mjmp on the opcode lines, and a reset address on the `current_instruction` lines. Thus the processor is forced to start execution at a known address. Currently no

specific actions are undertaken to reset all submodules. Of this is taken care by the HHDL language, and is at this level of decomposition not of any interest.

- Module: `INSTR_FETCH`.

  Interface to the `instruction cache`:

CACHE_CMD	output	record
CACHE_STAT	input	record
INSTR_ADR	output	integer
INSTRUCTION	input	integer

  Interface to the `bus unit`.

BUS_CMD	output	record
BUS_STAT	input	record

  Interface to the `instruction decoder`.

NEXT_CMD	output	record
NEXT_READY	input	record
OPCODE	output	integer
OPERAND_USED	input	boolean

  Data lines to the operand unit. they are not guarded by and handshake or ready signals. Handshaking is coordinated by the `instruction decoder`.

CURRENT_INSTRUCTION	output	integer
DIRECT	output	integer

  Address line to obtain the current programcounter. Is also not guarded by any handshake or ready signals.

PC_VALUE	input	integer

5.7.2 The instruction decoder.

The instruction decoder in itself is actually a very simple module. This view however is blurred by the enormous amount of instructions that all require a different sequence of operation.
In short: the module consists of only two case statements. In the first case statement is checked whether the programcounter should be updated (incremented) or not. The second case statement decodes the different instructions and schedules the appropriate submodules to execute their part of instruction.
The decoding of the instructions can be subdivided into eight large catagories, seven of them are executed very simply, the other needs (a lot of) extra attention.

The four simple classes are:

1. The regular two operand P-instructions.
2. The regular immediate operand P-instructions.
3. The regular FP relative operand P-instructions.
4. The regular two operand F(ull)-instructions.

   With 1-4 :

   Here regular denotes the use of the arithmetic and logic instructions, of which the shift and extract P- and extract F-instructions are excluded. The mode and type of ALU functions can be extracted directly from the opcode, and as a consequence of this all opcodes in one class are executed with the same piece of code. Every class has only one operand command to indicate the type of operands used.

5. The next class of instructions are the shift instructions (P- and F-instructions). They obtain their operand(s) in the same way as the previous 4 classes of opcodes. But the execution unit mode directly is set through the command line and not through the mode lines.
6. The next class is the set with one operand instructions(P- and F-instructions). They are all separately decoded and executed, each instruction has its own ALU mode and type. Every opcode has its own little piece of code for execution.
7. This leaves a small class of P-instructions that move data to and from < TOS >. (Ppushi, Ppushfp, Ppopfp, Pswap, Pdup), and Pextract0(123) Every one of these instructions has a specific operand command code, which sets the operand unit into the appropriate mode for the fetching and storing the operands.
8. The last class is the one with all other, not yet included instructions. These are mainly M-instructions, but also a few P- and F-instructions. Each of them has its own block of execution code, which are all distinct. For some of the opcodes are specific operand commands created. Some opcodes use operand commands that also used by the previous seven classes.

What all classes have in common, at the moment, is the way the execute scheduling of the submodules. A module is started by activating its command lines, then the decoder waits for the completion of the handshake sequence and deactivates the command line. Only then the next unit is executed. As usual with absolute rules there is one exception to the rule:

When a P-instruction is executed the register module is only then deactivated after completion of the operand unit.

This is due to the implementation of the register unit and the register module is directly controlled by the decoder. When the register module is deactivated the A- and B-bus are tri- stated, but the values previously on the bus are needed by the operand unit for the calculation of the operands.

- Module: INSTR_DECODE.

Interface to the **instruction fetch** input record index] NEXT_CMD

NEXT_READY	output	record
OPCODE	input	integer
OPERAND_USED	output	boolean

Interface to the **operand unit**.

OP_CMD	output	record
OP_STAT	input	record
OP_MODE	output	record

Interface to the **execution unit**.

EXEC_CMD	output	record
EXEC_STAT	input	record
EXEC_MODE	output	record

Interface to the **register unit**.

REG_CMD	output	record
REG_STAT	input	record

Status from the **Flags** register.

FLAGS	input	record

They are not guarded by and handshake or ready signals. Handshaking is achieved through the REG_CMD and REG_STAT lines.

5.7.3 Conclusions and/or remarks.

After a long elaboration on the different modules currently implemented, a few remarks and a short 'pre'conlusion seem appropriate.
As was already predicted in [Bud88], are the modules at the top level of very different complexity. The register unit is almost ready to be implemented in ports (and transistors). Whilst the caches on the other hand are not dealt with at all.
A general valid conclusion at this stage can however be that the current set with modules gives a good and solid basis to tackle the scheduling problem with. Currently everything is still executed in sequence. Note that with some of the modules already some possible

parallel actions are suggested.
Some of the problems are treated with a large overkill. I think at the forced complete handshake with all modules, and most data busses. In the real model this will certainly not be the case. Another case of overkill could be considered the two instruction busses to the `operand_unit`. (`current_instruction` and `direct`. This could prove to be overkill but the current way of implementation does not lead to any restrictions due to not enough "bus power" for fast execution. When in a next study is shown that only one is needed, then parts only have to be left out, which is simpler then to put it back in, in all previous stages.

>>>> REMARKS <<<<

As with the software model, are in this model some of the functions not completely (or completely not) defined. The most important fact and items of this imcompleteness are:

- The linked list operands. Since there was already no equivalent of those in either the instruction set. As was there also not in the software model.
- The instructions that manipulate the external bus actions: Lock, Halt, Reset, are more or less implemented. They are transported to the module that will be able to deal with the appropriate commands. But nothing is executed as part of the instructions.
- The instructionset contained two three operand instructions. They are currently totally disregarded in the design. More important is the fact that it will be very difficult to implement three operand instructions in the current architecture as one cycle actions. This would mean that the instructions be either removed from the instructionset, or that their definitions are changed to fit the current architecture.
- Defined in the instructions set are different high level Calls defined. The HHDL instruction decoder only knows the Mcallc instruction. The Mcallp instruction is not much work but currently it is not yet included in the code.

Chapter 6

Simulation and testing of the HHDL models

In the previous chapter is described how the "C"-processor is decomposed into several functional modules. A short description of each of all generated modules is given. It gives details of their functionality and of their external I/O definitions. It goes beyond saying that these modules need to be tested.

There are several reason for testing, but the two main reasons are (to my opinion):

1) Verify the correct functioning of the behaviour descriptions in HHDL. More crudely: Check whether the modules do what they are supposed to do.
2) Generate simulation results that can be used to verify future implementations of the same module. If the module has more steps of decomposition, or even a direct gate realization, the newly created model can be verified against the high level model.

The Silvar-Lisco package has several ways of simulating and testing the created models. The ones used in this thesis are:

- Direct simulation of the model through the HELIX simulator.
- And stimulation and observation of the model and a model driver.

In the first case the described model is compiled and linked with the simulator support package. This gives a simulator representation of the model. Pins on the model can be stimulated and observed. If the model in fact consists of several interconnected submodels, then the interconnections can be stimulated and observed as well. In this way all possible input stimuli can be assigned to test wether the model performs according the specifications. If this is not the case, then the model(s) has(ve) to be changed.

The second case, with the model driver, has all the characteristics of the first option. But it has an additional circuit, or symbol, with it. This driversymbol a matching pin for every pin on the symbol to test. If the testsymbol has an outputpin XXXX, then the drivermodel has an inputpin XXXX, and they are connected. The model driver is a regular model in the way that it is described in HHDL, but its purpose is to help with the testing of the module, and it usually mirrors the actions of the model to test.

Example.

- To test a module which interfaces to a master module (controller) and several child modules (slaves). In the first system all modules which have communication with the testmodule have to be expressed in simulation commands. This requires an exact knowledge of the timing information, and this for all modules with random interleaving. This is not a trivial thing to write, especially since the simulator control language is not very powerful.
- In the second case however, is it possible to create one or more subprocesses which will each mimic a child module in communication and data. The controller module could also be implemented in the drivermodel, in this case the whole system executes without interference from the operator. Tests could be build in to check for correct operation of the testmodule, and a GO/NO GO test would be the outcome. Less futuristic is the system where the role of controller is 'played' by the operator, which in his turn checks whether all child modules are activated correctly, and with the appropriate requirements.

It goes beyond saying that the type of testing to select, is dependant of the kind of module to test. And that it is certainly dependant on the complexity in communications and the complexity of the internal functioning of the model. The more complex the system, the harder it is to generate and observer all appropriate signals. (And this is not only because not all signals fit onto one display!)

As a guide rule one could say that a system with one channel to control and 2-3 channels to observe (but only observe) is the maximum that can be efficiently simulated without the assistance of a driver model. Are several channels in need of responses to their changes of state, then it would be advisable to test each of the channels in turn. The other channels will be interfaced by the driver model and will need (very) little attention.

This chapter contains only information on the ways of testing and simulating the implemented modules. No detailed simulator information is given. For this should be referred to either a separate appendix to this thesis, or directly to the used sources which are on the Apollo systems in the group.

6.1 Testing of the registers and the Register Unit.

Form the above section can be concluded that the register models are of such a low complexity that they can be tested with a simple simulation session. This is the approach chosen in this case. A simulator is created with the Silvar-Lisco package, a simulator run control file is created. And the results are evaluated.

This all sounds to good to be true, there is only one flaw to the specific software version. The HHDL models do not allow certain constructions of types, which are allowed in PASCAL, as types for the I/O of the models. Because of this some of these types are translated back and forth to strict integers. The exact assignments of bits in the integers can be obscure, and thus require some fiddling with large decimal integers. For this reason only, the simulator run control file interactsd directly with the register control lines. While the values should be assigning values to the global control busses instead.

While testing the `REGISTER unit` the model created for the uni-directional tri-state buses also needs to be tested. The model for this is simple and so is the testing for the correct functioning of the `BUSCHECK model`.

6.2 Models tested with a driver model.

It has already been stated that driver models can be very useful for the testing of more complex models. In fact are all models decribed in this thesis of such complexity that they justify the use of model drivers to exercise the created models.

If a model is tested with a driver then the driver will be used to facilitate the interfacing with the handshake channels on the test module (i.e. the module to test). The driver will contain several subprocesses which will run in parallel. And when the test module wants to communicate with the appropriately "connected" module, the driver will comply with the required handshake and service a possible data request.

The control signals of the test module are not activated by the driver. This is done through a simulation control file, which will write command data on inputs and will activate certain control lines. It the test module reports status information to the master (in this case the simulation control file), then it can proceed with the next step if the required conditions are true.

The modules tested in this way are:

- Stack.
- `get_offset_1`
- `get_offset_2`
- `get_operand_1`
- `get_operand_2`
- `store_result`
- `bus_unit`

- `instruction cache`
- `instr_fetch`
- `instr_decode`

The next step is to assemble the low level modules into their higher level counterparts and to simulate these. This has not yet been done. But it is very obvious that this will be done in exactly the same environement as it is done for the smaller submodules. The `Operand Unit` and the `Instruction Unit` will require far more interfaces for the submodules they control. But in general overview are all drivers very much alike.
The last step should be the assembly of the whole of the "C"-processor. Once all submodules are tested and are found to behave according expectations, the global module can be tested. The first and simplest test will be run all the different instructions on the simulator model. Every instruction is run in a seperate batch file. And the results should be thoroughly checked for errors and/or incomplete actions. The Software testfiles can also be used for this purpose although the bigger ones will require a large amount of processing and will produce a lot of data.

6.3 The results of the simulations.

The actual results of the simulations are not included in this report. This would serve no purpose what so ever. The amount of data is vast (3-4 megabytes), and would be only of intrest to those that will continue to study on this topic. The result are of course available on the system, and can be reviewed there.

Since the amount of data is large, the essential thing to do during the creation of the models and simulators to give thought to the way the results should be represented. If no thoughts are given to this detail at the current implementation level then the extraction of the valuable information will be very tedious. It is for this reason that (most) all modules have an abundant quantity of statments that report the status of the module. As a consequence of this, one does not need to view all signals in the system to trace the flow of execution. If there is a flaw in the design several signals could be traced to track the error down.

Once the current modules are divided into smaller components it will be unavoidable to use the signal values to trace the activity of the system. But up to this point holds the former statement too: "It's not just infomation that counts, it's valuable information that what we're looking for."

Chapter 7

Conclusions

This chapter will contain some specific conclusions regarding the projects undertaken. The conclusions drawn from this thesis should (or could) be used as guidelines for future steps in the design of the "C"-processor. One large (and major part) of the results is not in this document. Largely due to the vast amount of data generated, is chosen for a master thesis with as few listings as possible. If not included listings are needed they are available (upon request) on the Apollo systems of the Digital Systems group.

7.1 What has been done?

The instruction set has been defined.

The designed instructionset is actually divided into two instruction sets. A P-instruction set, which is targeted to the specific needs and capabilities of the "C"-processor. And a FM-instruction set which has a more general purpose implication for programming with this instruction set.
Effort is made with the P-instructions to have these executed a the highest possible speed. All operands are already onboard chip before the execution starts. The instruction are of a RISC type. The processor should be able to execute P-instructions within one cycle. (with the possible exception of multiply, divide and modulo operations).
The FM-instructionset is in the same sense a little more restricted. It does not contain instructions which are abundant with extra memory references. The maximum number of references per operand is two. One to get the offset of the operand, a second one is needed if the operand was given by an indirect format. This gives, all in all, a maximum of four references within one instruction. (two are in the code space in sequence from the instruction QUAD, two are in the data space at random)

A software model has been written.

This software model will execute the instructionset that is defined in this thesis. The software model was initially created as a preliminary version to the HHDL descriptions. Its purpose would be to supply a correct model which was easy to write and did not have any timing information related to it.
In the end it has become more that just a simple stepup to the HHDL version. The software model was very useful for gaining insight in the required data structures and the dataflow in the design. Next to this it can serve other purposes in the near future.
If the structural design of the processor resembles the modularity of the software model, the software model will be able to give information on the internal values in its components. The values in the software model will then relate to the values calculated in the hardware.

A sequential High Level model description is written.

the relevant information extracted is from the software model. (with simple tools like a cross-referencer, ..) This information is used to construct several of the data structures and is also used to find the relevant data paths in the design. It will also indicate the type of operations that will operate on a data structure. The decomposition of the topmost level into smaller components follows the exact decomposition in the software model. Not all constructs used in the software model are however allowed in HHDL, so where appropriate a conversion is made. Attempts are made to keep the difference as small as possible.
The models in HHDL use the, in chapter 2, described handshake for all communication actions. The flow of control follows the general control flow of the software model. This results in a completely sequential execution of the internal modules. At every point in time is only one module active. Through the handshake signals is the activity transferred back and forth from module to module. This again has the advantage that only one aspect in the design has influence on the results of the simulations. It is my opinion that the steps in the design should be made with as few varying parameters, per step, as possible. This will make the analysis of (faulty) results less complex.

A PL/0 compiler is written and implemented.

As a test case is a small compiler developed. The PL/0 compiler shows that a compiler can be made with the current set of instruction . The compiler will also be helpful to write small test programs. The limited syntax of the compiler is a severe limitation. It will not be possible to write large and complex programs, but smaller functional tests can be written in PL/0. An advantage of the limited syntax is the short learning period. The syntax is a very small subset of the language **PASCAL** and thus should not cause problems for any one.

7.2 What is the current status of the project?

The instruction set.

The instruction set has an operational status. Programs can be executed using the given instruction set. Space in the code set is still available for future additions and updates of the instruction set.

The software model.

The software model is able to execute programs in "C"-code. With this execution is it possible to trace the internal operations in the modules of the software model. Since most of these models have a hardware counterpart, it is possible to verify the hardware actions against the actions undertaken in the software model.

The HHDL description.

The High Level design of the "C"-processor is described in HHDL. Almost all modules are tested for correct behaviour. All submodules in the generated model are however executed in a completely sequential fashion. This means that one a module is started by its controller, the controller will wait for the submodule to terminate before starting another submodule. It is obvious that this will not be the case in the final design. For the time being however is this method a very good approximation of the functioning of the system.

The PL/0 compiler.

The **PL/0** compiler generates "C"-code. With this it is proven that the "C"-code can be used by a compiler to generate code. It is however not said that the code is optimal or representative for other high level language compilers. And as a consequence of this no conclusions on the completeness of the instruction set can be me. Non the less is every effort made to make the instruction set complete.

7.3 Future steps to take.

Extensively test the software model.

There are several good reasons for this.

1) The software model is not tested to its limits by it current designer, and bugs are still around. The problem with these bugs is that they could influence the hardware model, since the hardware model was derived from the software model. The sooner the bugs in the software model are found, the sooner can they be corrected in the hardware models. The penultimate in this would be to find a bug in the software model will the chip is already in the processing stadium.

2) Tests generated for the software model, can also be used as tests for the hardware model. And along with goes the fact that almost all responses in both systems should be alike. This making the initial testing simple. A recommendation would then be to generate a set of small test programs and use these as benchmarks in the initial tests of all future steps in the design.

3) One of the "C"-processor essential characteristics are the stackwindow and the instruction cache. To specify the properties of these parts with motivation, information is needed on the effects of size, algorithms, etc. used in those caches. To obtain this information simulations of large programs need to be run. From these runs can statistics be derived on the performance of the caches. These simulations will require less computing power if executed on a high level. Furthermore is it "very" simple to write a cache algorithm in a programming language like "C", and give statistical information from the required reads, writes, and misses.

Developing high level languages generating "C"-code.

In the previous section is already emphasized that large quantities of code need to be executed. It will be very cumbersome to write all of this code in the devised code format. In [Bud88] are even more profound reasons mentioned in favor of high level language compilers. For this purpose "C"-code generation compilers should be developed. Currently are available in source text:

- Small "C"compiler. [Hen84]

  This compiler accepts a subset of the language "C"which is substantial. It will certainly be complex enough for the required purposes. The compiler is able to compile itself, and this should be a nice task to use as test case.

  The current code generator generates 8086 code, which has very little resemblance with the "C"-code.

- P4 **PASCAL** compiler.

  This compiler originates from the university where **PASCAL** originated. It is a full implementation of the language (with some small extensions for a specific system) and the generated code is P4 code. P4 code is very much alike the P-code used in current instruction set.

Test the HHDL models to its full extends.

Was with the software model indicated that it still requires testing. Not all HHDL models are tested yet, and so is of course the complete assembly of the modules. The first objective will be to test all modules in their separate environments.
Once all modules function according their specifications and/or requirements, the building blockss can be put together. And the whole of the design is ready of testing. The vast majority of tests will check all communication channels. And once all are found in working order, the system can start the execution of single instructions and small programs.
No sooner than this can be concluded whether the current design is a successful design or that another approach should be taken.

Implement some of the submodules already on a lower level.

Some parts in the available design are on forehand already in for extra attention. The caches are not in the standard set of design items. One cycle 32x32 bit multipliers and dividers are also a challenging task. In a lot of modules 32 bit additions are found. The performance of the processor is in large detail dependant on the performance of these components.
Another very essential factor is the space required to implement certain items. From this point of view valid questions would be: How large are certain components, with certain (adjustable) parameters? And as final question how much silicon is already taken up by caches, multipliers, and other large items?
If these components cannot fulfil certain requirements, either the performance will be lower, or the design has to be redone to adjust the surrounding modules. This can be prevented by starting research on these specific components on lower levels or even on a gate level implementation. From this research should follow information which will guide some of the future design steps. It should provide answers to questions as: "Can a combinatorial multiplier be used, or should a 32x4 multiplier be used with an extra delay in execution?".

These results are not essential for the design on the current level of description, but are essential to make motivated decisions. (**It does not guarantee that correct decisions are made, but only that they are motivated.**)

7.4 The ultimate question.

Is the current design feasible?

The answer to this question should of course be YES! Otherwise 1 year of work would be wasted.

Aside from the above answer is it the authors opinion that from the current stage of

development the system can be developed to a real system. There are however a few catches to this:
It could prove that the chip is too complex to be manufactured with the current technology. The above studies should give an indication for this.
It could also prove that the execution speed is lower than expected. This could be due to a more cycles per instruction problem. Or it could also be a technology problem.

But still is it my opinion that from the current design a correct functioning lower level design can be made.

Appendix A

Bibliography

[Bud88] Frank Budzelaar. *The structured design of a processor for the language C.* Master's thesis, Eindhoven Universsity of Technology, The Netherlands, may 1988.

[Das84] Subrata Dasgupta. *The design and description of computer architectures.* Wiley-Interscience,, 1984. ISBN 0-471-89616-0.

[Hen84] James Hendrix. *The small-C handbook.* S.l. : Reston, 1984.

[Ker88] Brian W. Kernighan. *The C programming language.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[Pem82] S. Pemberton. *P4 Compiler and assembler/interpreter.* Halsted Press, 1982.

[Wir76] Niklaus Wirth. *Algorithms + data structures = Programs.* Prentice-Hall, 1976.

Appendix B

Enumeration of the "C"-processor instructions.

The following "C"-source files descibes the exact use of the bits in the instructions. It was used to determine which of the formats would be most convenient to implement the instruction stream in.
As a consequence one will find three instruction streams in this listing: A BYTE oriented, a WORD oriented, a QUAD oriented. And although the QUAD stream seems to lead to a little waist of instruction space, it is the more useful of the three.

```
/*
   The definitions of several types used for the
   generation of the "C"-processor instructionset.

   first made 28-4-88. WJW.

*/

typedef
struct {         /* Description of byte type parts  */
          unsigned :8;
      }byte;

typedef
union  {         /* Description of word type parts  */
          unsigned :16;
          struct { byte wb[2];
                 } wb;
      }word;

typedef
union  {         /* Description of quad type parts  */
          unsigned :32;
          struct { word qw[2];
                 }qw;
          struct { byte qb[4];
                 }qb;
       }quad;

union  core{            /* the basic elements of storage
                         */
          byte m8 [ 65536 ];
          word m16[ 32768 ];
          quad m32[ 16384 ];
           } mem;


typedef
struct {        /* Description of a Full operand   */
       unsigned  mode :2;
       union{
          struct{
             union{
                unsigned base_reg  :2;
                unsigned data_for  :2;
                  }source_spec;
             unsigned operand_size :3;
                }regular;
          unsigned   mix :5;
            }operand_descr;
       unsigned    signed :1;
       }F_operand_select;

#define F_OP_signed       1  /* bit 0  : for signed operands          */
#define F_OP_unsigned     0  /* bit 0  : for unsigned operands        */
#define F_OP_off8_siz8    0  /* bit 1-3: size of the operand          */
#define F_OP_off8_siz16   1  /*          and size of the offset       */
#define F_OP_off8_siz32   2
#define F_OP_off16_siz8   3
#define F_OP_off16_siz16  4
#define F_OP_off16_siz32  5
#define F_OP_off32_siz8   6
#define F_OP_off32_siz16  7

#define F_OP_base_sp      0    /* bit 4,5: use the stackpointer as base  */
#define F_OP_base_fp      1    /* bit 4,5: use the framepointer as base  */
#define F_OP_base_epf     2    /* bit 4,5: use the extern FP as base     */
#define F_OP_base_TOS     3    /* bit 4,5: use the Top of Stack as base  */
```

```
#define F_OP_data_direct  0    /* bit 4,5: use the direct data          */
#define F_OP_data_indir   1    /* bit 4,5: use the indirect data        */
#define F_OP_data_relPC   2    /* bit 4,5: use the PC as base register  */
#define F_OP_data_immed   3    /* bit 4,5: use the immediate data       */

#define F_OP_mode_rel     0    /* bit 6,7: RELATIVE                     */
#define F_OP_mode_rel_ind 1    /* bit 6,7: RELATIVE INDIRECT            */
#define F_OP_mode_data    2    /* bit 6,7: DATA                         */
#define F_OP_mode_misc    3    /* bit 6,7: MISCELLANEOUS with 32:32 op's */

#define F_OP_mix_sp32         0 /* register sp                           */
#define F_OP_mix_rel_sp32     1 /* relative sp[off32]:32                 */
#define F_OP_mix_relin_sp32   2 /* relative [sp[off32]]:32               */
#define F_OP_mix_flags32      3 /* FLAGS                                 */
#define F_OP_mix_imm32        4 /* immediate:32                          */
#define F_OP_mix_fp32         8 /* register fp                           */
#define F_OP_mix_rel_fp32     9 /* relative fp[off32]:32                 */
#define F_OP_mix_relin_fp32  10 /* relative [fp[off32]]:32               */
#define F_OP_mix_direct32    12 /* direct [addr32]                       */
#define F_OP_mix_efp32       16 /* register fp                           */
#define F_OP_mix_rel_efp32   17 /* relative fp[off32]:32                 */
#define F_OP_mix_relin_efp32 18 /* relative [efp[off32]]:32              */
#define F_OP_mix_indir32     20 /* indirect [[addr32]]                   */
#define F_OP_mix_pc32        24 /* register PC                           */
#define F_OP_mix_rel_TOS32   25 /* realtive <TOS>[off32]:32              */
#define F_OP_mix_relin_TOS32 26 /* relative [<TOS>[off32]]:32            */
#define F_OP_mix_TOS8        27 /* register <TOS>:8                      */
#define F_OP_mix_TOS16       27 /* register <TOS>:16                     */
#define F_OP_mix_TOS32       27 /* register <TOS>:32                     */
#define F_OP_mix_rel_pc32    27 /* relative pc[off32]:32                 */

typedef
struct {         /* condition codes */
       unsigned :8;
       }conditions;

typedef
struct{          /* the operand types possible in P Instruction's
                  */
      unsigned :2;
      }P_operand_type;
#define P_0_op    0   /* 0 operand instruction */
#define P_i_op    1   /* immediate data        */
#define P_fp_op   2   /* FP reference for data */
```

```
#define P_div_op  3   /* diverse operation sel */

typedef
struct{          /* the actions performed by the alu
                  */
      unsigned :3;
      }ALU_operation;
#define ALU_add   0   /* addition    w/o carry  */
#define ALU_sub   1   /* subtraction w/o borrow */
#define ALU_and   2   /* AND         bool+logic */
#define ALU_or    3   /* OR          bool+logic */
#define ALU_xor   4   /* XOR         logic      */
#define ALU_mult  5   /* multiply    (un)signed */
#define ALU_mod   6   /* modulo      (un)signed */
#define ALU_div   7   /* divide      (un)signed */

typedef
union{           /* the mode of the operation to perform.
                  */
     unsigned          carry:1;
     unsigned          signed:1;
     unsigned          boolean:1;
     }ALU_mode;

#define ALU_nocarry  0 /* operation without carry */
#define ALU_carry    1 /* operation with carry    */
#define ALU_unsigned 0 /* unsigned operation      */
#define ALU_signed   1 /* signed operation        */
#define Alu_logic    0 /* logic operation         */
#define ALU_boolean  1 /* boolean operation       */

typedef
struct {         /* The P_code instruction descriptor   */
       P_operand_type   P_operand;   /* type of operand */
       ALU_operation    alu_code ;   /* alu function sel*/
       ALU_mode         P_mode   ;   /* c|nc, u|s, b|l  */
       }P_opcode;

typedef
struct {         /* The P_code instruction */
       unsigned         P_select :2; /* P_indicator */
       /* must be 00b */
       P_opcode         P_function;
       }P_instruction;
```

```
typedef
struct {        /* A P_code instruction with a short data field
                 */
       unsigned         P_select :2;
       /* must be 00b */
       P_opcode         P_function;
       byte             P_data;
       }PD_instruction;


typedef
struct {        /* The F_code instruction selector */
       unsigned :6;
       }F_opcode;

typedef
struct {        /* The F_code instruction */
       unsigned         F_select :2;
       /* must be 10b */
       F_opcode         F_function;
       }F_instruction;

typedef
struct {        /* The F_code instruction with 1 operand
                 */
        unsigned         F_select :2;
        /* must be 10b */
        F_opcode         F_function;
        F_operand_select F_op;
        union{
             word        F_data_16;
             struct{
                   byte  F_data_8;
                   byte  empty;
                   }short_data;
             }data;
       }F_1op_instruction;

typedef
struct {        /* The F_code instruction with 2 operands
                 */
        unsigned         F_select :2;
        /* must be 10b */
        F_opcode         F_function;
        F_operand_select F_op[2];
        union{
             byte        F_data_8;
             byte        empty;
             }short_data;
       }F_2op_instruction;


typedef
struct {         /* The M_code instruction selector */
       unsigned :6;
       }M_opcode;


typedef
struct {         /* The M_code instruction */
       unsigned          M_select :2;
       /* must be 01b */
       M_opcode          M_function;
       }M_instruction;

typedef
struct {         /* The M_code instruction without operands
                  */
       unsigned          M_select :2;
       /* must be 01b */
       M_opcode          M_function;
       byte              empty[3];
       }M_0op_instruction;

typedef
struct {         /* The M_code instruction with 1 operand
                  */
       unsigned          M_select :2;
       /* must be 01b */
       M_opcode          M_function;
       union{
            F_operand_select M_op;
            byte             M_data_8;
            }field_1;
       union{
             word        F_data_16;
             struct{
                   byte  F_data_8;
                   byte  empty;
                   }short_data;
            }data;
      }M_1op_instruction;

typedef
struct {         /* The M_code instruction with 2 operands
                  */
      unsigned          M_select :2;
      /* must be 01b */
      M_opcode          M_function;
      union{
           F_operand_select  M_op[2];
           struct{
                  conditions C;
                  F_operand_select M_op
                  }jump_field;
            }operands;
      }M_2op_instruction;

typedef
struct {         /* The set of NOP's                */
          unsigned :6;
       }N_opcode;

typedef
struct {         /* The N_code instruction */
          unsigned N_select :2;
           /* must be 11b */
          N_opcode N_function;
      }N_instruction;

typedef
union  {         /* just the fields with the opcodes
                    are merged not the operands      */
     P_instruction P;
     F_instruction F;
     M_instruction M;
     N_instruction Nop;
     struct{
            unsigned opcode_select :2;
            /* 00b P_code
               10b F_code
               01b M_code
               11b NOP
             */
            union{          /* the active item depends
```

```
                                  the opcode_select */
                      P_opcode P_field;
                      F_opcode F_field;
                      M_opcode M_field;
                      N_opcode Nop;
                    }op_field;
              }split_fields;
        }opcode;

typedef
union  {         /* The description of an 8 bits
                    instruction stream
                  */
          opcode           instruction;
          F_operand_select operand;
          byte             data_8; /* normal 8 bits data     */
          byte             word_h; /* high part of a word    */
          byte             word_l; /* low part of a word     */
          byte             quad_3; /* byte 3 of a quad, msb */
          byte             quad_2; /* byte 2 of a quad,      */
          byte             quad_1; /* byte 1 of a quad,      */
          byte             quad_0; /* byte 0 of a quad, lsb */
       }instruction_8;


typedef
union  {         /* The description of an 16 bits
                    instruction stream
                  */
          /* data information */
          byte             data_8[2]; /* 2 databytes in 1 word   */
          word             data_16;   /* normal 16 bits of data */
          word             quad_h;    /* the msb word of quad    */
          word             quad_l;    /* the lsb word of quad    */

          /* instruction formats */

          /* with P_instructions */
          P_instruction    P_i2[2];   /* 2 executable P_instr.   */
          PD_instruction   PD_i1;     /* P_instr with data       */
          struct{
             P_instruction P_i1;      /* 1 executable P_instr.   */
             N_instruction Nop;       /* and a NOP instr.        */
               }P_i1;
```

```
   /* with F_instructions */
struct{                        /* the first part of
                                  F_instr. contains opcode
                                  and first operand */
   F_instruction    F_i;
   F_operand_select F_op1;     /* first F_operand.      */
      }F_i_begin;
union {                        /* the possible second part
                                  of a F_instruction contains
                                  second operand and data */
   word             F_data_16; /* second field is data    */
   struct{
      F_operand_select F_op2;  /* second F_operand.       */
      union{                   /* and a data field        */
          byte        F_data_8;/* with data               */
          byte        empty;   /* or is empty             */
           }F_i_end_data;
         }F_i_operand;
      }F_i_end;

   /* with M_instructions */
struct{                        /* first part contains
                                  opcode and possible
                                  the (first) operand     */
   M_instruction       M_i;
   union{                      /* place for operand or
                                  data                    */
       conditions      c;      /* conditions for tests    */
       F_operand_select F_op1; /* first operand           */
       byte            M_data_8;/* data field             */
       byte            empty;  /* not used                */
        }M_i_field2;
      }M_i_begin;
union {                        /* second part contains the
                                  second operand and or
                                  data ( 8 or 16 )        */
   word                data_16;
   struct{
      F_operand_select F_op2;  /* second operand          */
      union{                   /* possible data           */
          byte        data_8;
          byte        empty;
           }data_field;
         }split;
      }M_i_end;
```

```
       }instruction_16;



typedef
union  {          /* The description of an 32 bits
                     instruction stream
                   */
          /* data information */
          byte            data_8[4]; /* 4 databytes in 1 quad  */
          word            data_16[2];/* 2 words in a quad      */
          quad            data_32;   /* the normal quad        */

          /* instruction formats */

          /* with P_instructions */
       struct{
          P_instruction    P_i1;     /* first field alway P_inst*/
          union{
            N_instruction  Nop234[3]; /* rest is nop's          */
            union{
              struct{
                byte    data_8;      /* data from instr 1       */
                union{
                  N_instruction Nop34[2];
                                     /* only one PD_instruction */
                  struct{
                     P_instruction P_i3;
                                     /* 1PD and 1 P_instruction */
                     union{
                       N_instruction Nop4;
                                     /* last field is NOP       */
                       P_instruction P_i4;
                                     /* 1PD and 2 P_instructions */
                       byte         data_8;
                                     /* data for instr on field 3
                                        2 PD instructions
                                      */
                         }field_4;
                       }active_34;
                     }field_34;
                    }data_2;
              struct{
                P_instruction P_2i;  /* 2nd P_instruction       */
                union{
```

```
            N_instruction Nop34[2];
                              /* only 2 P_instructions   */
            union{
              struct{
                byte          data_8;
                              /* 1 P and 1 PD_instruction*/
                union{
                  P_instruction P_i4;
                              /* 1 P, 1 PD, 1P_instr.    */
                  N_instruction Nop4;
                              /* 1 P, 1 PD, 1 NOP        */
                     }field_4;
                    }data_3;
              struct{
                P_instruction   P_i3;
                              /* uptil now 3 P_instr.    */
                union{
                  N_instruction Nop4;
                              /* last field is NOP       */
                  P_instruction P_i4;
                              /* 4 P_instructions        */
                  byte          data_8;
                              /* data for instr on field 3
                                 2 P and 1 instructions
                               */
                     }field_4;
                    }instr_3;
                 }active_34;
              }field_34;
              }instr2;
           }active_234;
         }field_234;
       }P_i;

    /* with F_instructions */

   /* 1 quad can contain either 1 2-operand F-instruction
       with some data (8 bits).
       Or 1 1-operand instruction with data
     */
 F_1op_instruction  F_1i;
 F_2op_instruction  F_2i;

    /* with M_instructions */
 M_0op_instruction  M_0i;
```

```
        M_1op_instruction  M_1i;
        M_2op_instruction  M_2i;

        }instruction_32;
```

The following list simply enumerates all possible instruction opcodes with their nummeric value. They fit in the above patterns but, it is rather hrad to derive an opcode from that.

```
/*  file: instr_def.def

    All instructions that are valid for execution by
    the C-processor are in this file.
    They are subdivided in the following types of instruction.
    -   P-instructions: short and fast P-code instructions,
                        with implied stack addressing.
    -   M-instructions: Those instructions that are not in the
                        two classes above.
    -   F-instructions: With a Full addressing possibility for
                        almost all instructions.
    -   NOP-instruction: Fill for quad's with Pcodes

     P-instructions:
     operations with implied addresses
     <TOS>, <TOS-1>, <TOS-2>.

     And thus have no operands.
*/
#define Padd         0 /* add                 */
#define Paddc        1 /* add with carry      */
#define Psub         2 /* sub                 */
#define Psubb        3 /* sub with borrow     */
#define Pland        4 /* logic and           */
#define Pband        5 /* boolean and         */
#define Plor         6 /* logic or            */
#define Pbor         7 /* boolean or          */
#define Pxor         8 /* logic eXclusive OR  */
#define Pdup         9 /* duplicate stack top */
#define Pmultu      10 /* unsigned multiply   */
#define Pmults      11 /* signed multiply     */
#define Pmodu       12 /* unsigned modulo     */
#define Pmods       13 /* signed modulo       */
#define Pdivu       14 /* unsigned modulo     */
#define Pdivs       15 /* signed modulo       */
/*
     operations with 1 immediate operand.
*/
#define Paddi       16 /* add                 */
```

```
#define Paddci      17 /* add with carry         */
#define Psubi       18 /* sub                    */
#define Psubbi      19 /* sub with borrow        */
#define Plandi      20 /* logic and              */
#define Pbandi      21 /* boolean and            */
#define Plori       22 /* logic or               */
#define Pbori       23 /* boolean or             */
#define Pxori       24 /* logic eXclusive OR     */
#define Ppushi      25 /* push immediate data    */
#define Pmultui     26 /* unsigned multiply      */
#define Pmultsi     27 /* signed multiply        */
#define Pmodui      28 /* unsigned modulo        */
#define Pmodsi      29 /* signed modulo          */
#define Pdivui      30 /* unsigned modulo        */
#define Pdivsi      31 /* signed modulo          */
/*
     operations with 1 operand relative to the
     FP. This gives a local variable or a parameter.
*/
#define PaddFP      32 /* add                    */
#define PaddcFP     33 /* add with carry         */
#define PsubFP      34 /* sub                    */
#define PsubbFP     35 /* sub with borrow        */
#define PlandFP     36 /* logic and              */
#define PbandFP     37 /* boolean and            */
#define PlorFP      38 /* logic or               */
#define PborFP      39 /* boolean or             */
#define PxorFP      40 /* logic eXclusive OR     */
#define PpushFP     41 /* push variable or param. */
#define PmultuFP    42 /* unsigned multiply      */
#define PmultsFP    43 /* signed multiply        */
#define PmoduFP     44 /* unsigned modulo        */
#define PmodsFP     45 /* signed modulo          */
#define PdivuFP     46 /* unsigned modulo        */
#define PdivsFP     47 /* signed modulo          */
/*
     operations of diverse types.
     this is a weird collection of P_codes,
     and there is as a consequence of that
     very little logic in it's structure.
*/
#define Plnot       48 /* logic not                  */
#define Pbnot       49 /* boolean not                */
#define Pneg        50 /* negate                     */
#define Pcomp       51 /* compare                    */
```

```
#define Pextract0   52 /* extract a subrange          */
#define Pextract1   53 /* of bits from <TOS>.         */
#define Pextract2   54 /* but this needs 10 bits.     */
#define Pextract3   55 /* so 8 data and 2 instr. bits */
#define Pshl        56 /* shift left                  */
#define Pswap       57 /* exchange <TOS>, <TOS-1>     */
#define Pcompi      58 /* compare with immediate data */
#define PpopFP      59 /* Pop into FP reference       */
#define Pshru       60 /* unsigned shift right        */
#define Pshrs       61 /* signed shift right          */
#define Pcheck      62 /* check bounds                */
#define Psel        63 /* select one of two operands  */

/*
     M-operation instructions
     Mixed instructions.

*/
#define Mret         64 /* return from subroutine      */
#define Mcall        65 /* call subroutine             */
#define Mcallc       66 /* call function or procedure */
                        /* use C calling convention    */
#define Mjmpc        67 /* jump on condition           */
#define Mreti        72 /* return from interrupt       */
#define Mjmp         73 /* jump always                 */
#define Mswap        74 /* swap process descriptors    */
#define Mcallp       75 /* call function or procedure */
                        /* Use PASCAL calling convent.*/
#define Mcic         80 /* clear instruction cache     */
#define Mretf        81 /* return from function or pr.*/
#define Mtdj         82 /* test, decrement and jump    */
#define Mupm         88 /* update memory               */
#define Mtrap        89 /* take trap nnn               */
#define Mreset       96 /* activate reset line         */
#define Menaint      97 /* enable interrupts           */
#define Mhalt       104 /* halt processor              */
#define Mdisint     105 /* disable interrupts          */
#define Mlock       113 /* lock systembuses            */

/*
     F-operation instructions
     Operations with full addressing modes for both
     operands.

*/
```

```
#define Fadd        128 /* add with 2 full operands   */
#define Faddc       129 /* add carry 2 operands       */
#define Fsub        130 /* subtract                   */
#define Fsubb       131 /* subtract with borrow       */
#define Fland       132 /* logic AND                  */
#define Fband       133 /* boolean AND                */
#define Flor        134 /* logic OR                   */
#define Fbor        135 /* boolean OR                 */
#define Fxor        136 /* XOR                        */
#define Fmove       137 /* move data                  */
#define Fmultu      138 /* unsigned multiply          */
#define Fmults      139 /* signed multiply            */
#define Fmodu       140 /* unsigned modulo            */
#define Fmods       141 /* signed modulo              */
#define Fdivu       142 /* unsigned divide            */
#define Fdivs       143 /* signed divide              */
#define Fcheck      146 /* check bounds               */
#define Fselect     148 /* select one out of two      */
#define Flnot       152 /* logic inverse              */
#define Fcomp       162 /* compare                    */
#define Fbnot       168 /* boolean inverse            */
#define Fneg        178 /* two's complement           */
#define Fshl        184 /* shift left                 */
#define Fshrs       188 /* signed shift right         */
#define Fshru       189 /* unsigned shift right       */


#define NOP         192 /* no action instruction      */

/* end file: instr_def.def */
```

B.1 Operand formats and their values.

In this appendix is described the format of the operands. Listed are the possible operands and their appropriate value in decimal and binary notation.

The following notational conventions are used:

[...] Indicates a memory reference.

reg Gives the name of a register used for reference or as source or destination.

XX[...] Indicates a memory reference with the register XX used as baseregister.

offnn Is a value of described with nn bits, These nn bits are stored in the codestream. If the result however is to be used as an offset, the value in the codestream is sign extended to 32 bits. As sign indicator is the most significant bit used of the offset stored in the instructionstream. For example an offset off8 with stream value 129(dec) will result in an 32 bit value of -127.

:nnX Indicates the size and sign of the operand the fetch. If the operand is not 32 bits in size, then it has to be extended before it can be used in the ALU. The sign extension flag X indicates whether the extension is signed of unsigned. The field nn indicates the memory size of the operand to be fetched.

nn#immediate:mmX The operand is immediate data. The size of storage in the codestream is indicated by the field nn. The size and sign for the conversion are given by mmX.

dec	binary	description
	relative operands	
0	00000000	sp [off8]:8 u
1	00000001	sp [off8]:8 s
2	00000010	sp [off8]:16u
3	00000011	sp [off8]:16s
4	00000100	sp [off8]:32u
5	00000101	sp [off8]:32s
6	00000110	sp [off16]:8 u
7	00000111	sp [off16]:8 s
8	00001000	sp [off16]:16u
9	00001001	sp [off16]:16s
10	00001010	sp [off16]:32u
11	00001011	sp [off16]:32s
12	00001100	sp [off32]:8 u
13	00001101	sp [off32]:8 s
14	00001110	sp [off32]:16u
15	00001111	sp [off32]:16s
16	00010000	fp [off8]:8 u
17	00010001	fp [off8]:8 s
18	00010010	fp [off8]:16u
19	00010011	fp [off8]:16s
20	00010100	fp [off8]:32u
21	00010101	fp [off8]:32s
22	00010110	fp [off16]:8 u
23	00010111	fp [off16]:8 s
24	00011000	fp [off16]:16u

25	00011001	fp [off16]:16s
26	00011010	fp [off16]:32u
27	00011011	fp [off16]:32s
28	00011100	fp [off32]:8 u
29	00011101	fp [off32]:8 s
30	00011110	fp [off32]:16u
31	00011111	fp [off32]:16s
32	00100000	efp[off8]:8 u
33	00100001	efp[off8]:8 s
34	00100010	efp[off8]:16u
35	00100011	efp[off8]:16s
36	00100100	efp[off8]:32u
37	00100101	efp[off8]:32s
38	00100110	efp[off16]:8 u
39	00100111	efp[off16]:8 s
40	00101000	efp[off16]:16u
41	00101001	efp[off16]:16s
42	00101010	efp[off16]:32u
43	00101011	efp[off16]:32s
44	00101100	efp[off32]:8 u
45	00101101	efp[off32]:8 s
46	00101110	efp[off32]:16u
47	00101111	efp[off32]:16s
48	00110000	TOS[off8]:8 u
49	00110001	TOS[off8]:8 s
50	00110010	TOS[off8]:16u
51	00110011	TOS[off8]:16s
52	00110100	TOS[off8]:32u
53	00110101	TOS[off8]:32s
54	00110110	TOS[off16]:8 u
55	00110111	TOS[off16]:8 s
56	00111000	TOS[off16]:16u
57	00111001	TOS[off16]:16s
58	00111010	TOS[off16]:32u
59	00111011	TOS[off16]:32s
60	00111100	TOS[off32]:8 u
61	00111101	TOS[off32]:8 s
62	00111110	TOS[off32]:16u
63	00111111	TOS[off32]:16s

relative in direct operands

64	01000000	[sp[off8]]:8 u
65	01000001	[sp[off8]]:8 s
66	01000010	[sp[off8]]:16u

67	01000011	[sp[off8]]:16s
68	01000100	[sp[off8]]:32u
69	01000101	[sp[off8]]:32s
70	01000110	[sp[off16]]:8 u
71	01000111	[sp[off16]]:8 s
72	01001000	[sp[off16]]:16u
73	01001001	[sp[off16]]:16s
74	01001010	[sp[off16]]:32u
75	01001011	[sp[off16]]:32s
76	01001100	[sp[off32]]:8 u
77	01001101	[sp[off32]]:8 s
78	01001110	[sp[off32]]:16u
79	01001111	[sp[off32]]:16s
80	01010000	[fp[off8]]:8 u
81	01010001	[fp[off8]]:8 s
82	01010010	[fp[off8]]:16u
83	01010011	[fp[off8]]:16s
84	01010100	[fp[off8]]:32u
85	01010101	[fp[off8]]:32s
86	01010110	[fp[off16]]:8 u
87	01010111	[fp[off16]]:8 s
88	01011000	[fp[off16]]:16u
89	01011001	[fp[off16]]:16s
90	01011010	[fp[off16]]:32u
91	01011011	[fp[off16]]:32s
92	01011100	[fp[off32]]:8 u
93	01011101	[fp[off32]]:8 s
94	01011110	[fp[off32]]:16u
95	01011111	[fp[off32]]:16s
96	01100000	[efp[off8]]:8 u
97	01100001	[efp[off8]]:8 s
98	01100010	[efp[off8]]:16u
99	01100011	[efp[off8]]:16s
100	01100100	[efp[off8]]:32u
101	01100101	[efp[off8]]:32s
102	01100110	[efp[off16]]:8 u
103	01100111	[efp[off16]]:8 s
104	01101000	[efp[off16]]:16u
105	01101001	[efp[off16]]:16s
106	01101010	[efp[off16]]:32u
107	01101011	[efp[off16]]:32s
108	01101100	[efp[off32]]:8 u
109	01101101	[efp[off32]]:8 s
110	01101110	[efp[off32]]:16u
111	01101111	[efp[off32]]:16s

112	01110000	[TOS[off8]]:8 u
113	01110001	[TOS[off8]]:8 s
114	01110010	[TOS[off8]]:16u
115	01110011	[TOS[off8]]:16s
116	01110100	[TOS[off8]]:32u
117	01110101	[TOS[off8]]:32s
118	01110110	[TOS[off16]]:8 u
119	01110111	[TOS[off16]]:8 s
120	01111000	[TOS[off16]]:16u
121	01111001	[TOS[off16]]:16s
122	01111010	[TOS[off16]]:32u
123	01111011	[TOS[off16]]:32s
124	01111100	[TOS[off32]]:8 u
125	01111101	[TOS[off32]]:8 s
126	01111110	[TOS[off32]]:16u
127	01111111	[TOS[off32]]:16s

Operands with direct data, or relative to PC

128	10000000	[off8]:8 u
129	10000001	[off8]:8 s
130	10000010	[off8]:16u
131	10000011	[off8]:16s
132	10000100	[off8]:32u
133	10000101	[off8]:32s
134	10000110	[off16]:8 u
135	10000111	[off16]:8 s
136	10001000	[off16]:16u
137	10001001	[off16]:16s
138	10001010	[off16]:32u
139	10001011	[off16]:32s
140	10001100	[off32]:8 u
141	10001101	[off32]:8 s
142	10001110	[off32]:16u
143	10001111	[off32]:16s
144	10010000	[[off8]]:8 u
145	10010001	[[off8]]:8 s
146	10010010	[[off8]]:16u
147	10010011	[[off8]]:16s
148	10010100	[[off8]]:32u
149	10010101	[[off8]]:32s
150	10010110	[[off16]]:8 u
151	10010111	[[off16]]:8 s
152	10011000	[[off16]]:16u

153	10011001	[[off16]]:16s
154	10011010	[[off16]]:32u
155	10011011	[[off16]]:32s
156	10011100	[[off32]]:8 u
157	10011101	[[off32]]:8 s
158	10011110	[[off32]]:16u
159	10011111	[[off32]]:16s
160	10100000	pc[off8]:8 u
161	10100001	pc[off8]:8 s
162	10100010	pc[off8]:16u
163	10100011	pc[off8]:16s
164	10100100	pc[off8]:32u
165	10100101	pc[off8]:32s
166	10100110	pc[off16]:8 u
167	10100111	pc[off16]:8 s
168	10101000	pc[off16]:16u
169	10101001	pc[off16]:16s
170	10101010	pc[off16]:32u
171	10101011	pc[off16]:32s
172	10101100	pc[off32]:8 u
173	10101101	pc[off32]:8 s
174	10101110	pc[off32]:16u
175	10101111	pc[off32]:16s
176	10110000	immediate:8 :8 u
177	10110001	immediate:8 :8 s
178	10110010	immediate:8 :16u
179	10110011	immediate:8 :16s
180	10110100	immediate:8 :32u
181	10110101	immediate:8 :32s
182	10110110	immediate:16:8 u
183	10110111	immediate:16:8 s
184	10111000	immediate:16:16u
185	10111001	immediate:16:16s
186	10111010	immediate:16:32u
187	10111011	immediate:16:32s
188	10111100	immediate:32:8 u
189	10111101	immediate:32:8 s
190	10111110	immediate:32:16u
191	10111111	immediate:32:16s

miscellaneous: register and 32:32 operands

192	11000000	sp
193	11000001	sp
194	11000010	sp[off32]:32u

195	11000011	sp[off32]:32s
196	11000100	[sp[off32]:32]u
197	11000101	[sp[off32]:32]s
198	11000110	flags
199	11000111	flags
200	11001000	immediate:32:32u
201	11001001	immediate:32:32s
202	11001010	
203	11001011	
204	11001100	
205	11001101	
206	11001110	
207	11001111	
208	11010000	fp
209	11010001	fp
210	11010010	fp[off32]:32u
211	11010011	fp[off32]:32s
212	11010100	[fp[off32]]:32u
213	11010101	[fp[off32]]:32u
214	11010110	
215	11010111	
216	11011000	[off32]:32u
217	11011001	[off32]:32s
218	11011010	
219	11011011	
220	11011100	
221	11011101	
222	11011110	
223	11011111	
224	11100000	efp
225	11100001	efp
226	11100010	efp[off32]:32u
227	11100011	efp[off32]:32s
228	11100100	[efp[off32]]:32u
229	11100101	[efp[off32]]:32u
230	11100110	[[off32]]:32u
231	11100111	[[off32]]:32s
232	11101000	
233	11101001	
234	11101010	
235	11101011	
236	11101100	
237	11101101	
238	11101110	
239	11101111	

240	11110000	pc
241	11110001	pc
242	11110010	TOS[off32]:32u
243	11110011	TOS[off32]:32s
244	11110100	[TOS[off32]]:32u
245	11110101	[TOS[off32]]:32s
246	11110110	TOS:8
247	11110111	TOS:8
248	11111000	TOS:16
249	11111001	TOS:16
250	11111010	TOS:32
251	11111011	TOS:32
252	11111100	
253	11111101	
254	11111110	
255	11111111	

Appendix C

A PL/0 compiler for the C-processor

It is already mentioned several times before: ' It would be very convenient if a compiler was available.' When the "C"-processor, under design, should become a real product it is inevitable that compilers should be available. One could read [Bud88] on this philosophy. For the purpose of testing the created instructionset, and perhaps for testing the effects of different cache- mechanisms, a very small and simple but usable compiler is written. The **PL/0** compiler listed in Chapter 5 of **Datastructures + Algorithms = Programs** by N. Wirth [Wir76] is used as a skeleton, the lexical scanner and language parser were retained but the code generation part was rewritten to fit the needs and capabilities of the "C"-processor.
In this appendix the code generation be discussed will, the lexical scanner and the parser are thoroughly treated in the book by Wirth, and thus only a few words are spent on these topics.

C.1 The lexical scanner.

The lexical scanner is the working horse of a compiler. It collects character strings which together make a language item. It then translates these items into tokens, where tokens can be things like BEGIN-symbol, END-symbol, IDENTIFIER, NUMBER, etc. With these tokens are sometimes attributes connected. For instance the token NUMBER has an attribute VALUE which hold the numeric value of the token. An IDENTIFIER has an attribute which holds the NAME of the identifier, a integer which indicates the level of declaration, a address which gives the offset in the current frame.

In our specific case the scanner recognizes the reserved words of the language:

- **BEGIN**
- **CALL**
- **CONST**
- **DO**

- **END**
- **IF**
- **ODD**
- **PROCEDURE**
- **THEN**
- **VAR**
- **WHILE**

The following items are also recognized:

- identifiers, maximum length is 10. An identifier can only contain alphabetic characters and digits, The first character has to be a character. (Identifiers are case insensitive.)
- Numbers in decimal notation. The size is restricted to the system on which the compiler is implemented.
- + for addition.
- - for subtraction or as unary negation operator.
- * for multiplication.
- / for integer division.
- (and) for priority in expressions.
- := is the assignment operator
- = for tests on equal.
- , separator during constant or variable definition.
- . program terminator.
- ! for tests on not equal.
- < for tests on less.
- > for tests on greater.
- # for tests on less or equal.
- $ for tests on greater or equal.
- ; as separator for statements.

The above mentioned items are read from the input file, symbol by symbol and handed to the requester: the parser.

C.2 The PL/0 parser.

The **PL/0** parser of the recursive descend type. This makes it very easy to implement a structure of routines which recognizes the language definition. The code generation has to be fitted into this structure of routines. This is also relatively simple since the parser clearly indicates which language element is being translated, and when the programmer has a clear idea of what to do with each of the statements, the two match up very well.

A language which is to be compiled with this method has to comply with certain restrictions in its grammar and its context sensitivity. For specific information on this topic is referred to [Wir76, paragraph 5.1 to 5.4], or to most other textbooks on compilers for more general information on recursive descend compiling.
The main language items recognized by the parser are:

- Constant declarations.
- Variable declarations.
- Procedure declarations.
- Block statement.
- Assignment statement.
- If statement.
- While statement.
- Procedure call statement.
- Expressions consisting of:
  - terms
  - factors
  - conditions.

Once these items are recognized by the parser, the last thing the compiler has to do, is to generate the code which is needed for the execution of this language element.

C.3 The code generation.

The original **PL/0**compiler is equipped with a code generator which selects its instructions from a small subset of the P-codes. All of the P-codes in the literature can be classified as STACK oriented instructionsets. This code generator has to be replaced by one that generates instructions chosen for the "C"-code instructionset.
The code generation is done in two steps. In the first step is code generated which maps to the "C"-code on a very simple way. But the code generator ignores the fact that some codewords can contain more that just one instruction. and that for different sizes of operands, different versions of instructions are available. This also makes it very simple to generate the addresses that go with the jumps, or to fill them in later on during the compilation. For this reason also, is the whole memory image of generated code kept in memory. The advantage to this is that jump addresses can directly be filled in once the jump addressis known, and thus no labels are needed for updating in a second pass.

The disadvantage is that the maximum amount of generated code is limited to the space reserved for code in the compiler. The amount of code however that can be generated is fairly large, and the compiler is simple. These two would validate the assumption that the memory space will be more that adequate for almost all programs to be written in **PL/0**.

C.3.1 The instructions generated.

The following instructions are used in the first step of the code generation. With them is the "C"-code instruction sequence that is generated in the second pass. Some of the instructions are already "C"-code instructions, and hence they need no translation, they will only be packed into quads when that is possible.

Direct usable "C"-code instructions:

• **Pneg**	{ negate <TOS> }
• **Padd**	{ add <TOS> + <TOS>-1 }
• **Psub**	{ sub }
• **Pmult**	{ multiply }
• **Pdiv**	{ divide }
• **Mret**	{ return from subroutine }
• **Mretf**	{ return from function or procedure }
• **callf**	{ call a function or procedure }
• **calr**	{ call routine without frame }
• **jmp**	

PL/0-Instructions that need to be translated.

• lit { load a constant }

Dependant on the size of the constant the use "C"-code is:

Ppushi <constant>
If the constant is in the interval -128 <= <constant> <= 127.

Fmove <TOS>, 32:<constant16>
In all other cases.

• jpc { jump on condition of a test }

Before it is possible to make a jump, conditions to jump on have to be set. This is done with an extra instruction. Possible conditions to jump on are:

```
{ test in jumps codes }
jmp_test =(
        t_equ        { test for equality }
        ,t_neq       { test for unequality }
,t_lss          { test for less }
,t_gtr          { test for greater }
,t_leq          { for less or equal}
,t_geq          { for greater or equal }
,t_odd          { is TOS odd or even }
);
```

If the expression is a comparision then the conditions are set by doing a Pcomp instruction. This instruction subtracts < TOS > and < TOS − 1 >, does not leave the result of the subtraction on the stack, but does affect the flags. The conditions are specific tests on the ZERO and CARRY flag.

For the test on **ODD** the < TOS > is anded with 1 and a jump on ZERO is taken, as the representative of the even condition.

As with the unconditional jump, the addresses for the jumps are fixed on 16 bit addresses. Since some of the jumps are forward jumps. Not all addresses can be resolved at first. For this purpose a new pass through the intermediate code is needed. There are several other solutions for this. The one currently used was the fasted to program, and requires very little code.

Thus the instruction is either:

Pcomp
Mjmpc <condition>, <address16>

or for the **ODD** condition:

Plandi 1
Mjmpc Z, <address16>

- lod { load a variable on the stack }

This code is used as the generic instruction for loading the value of a variable on the stack. Since **PL/0**is a language with a specific scope for every variable, different variables are to be found in different stack frames. This in contrast with the "C"-language which has only the global and local variables which it can access. The globals are in the "outermost" frame, whilest the local variables are located in the current stackframe.

Currently is this exactly what is supported by the current version of the **PL/0** compiler. As a consequence of this it is not allowed to nest procedures. Although the compiler syntax and parser are propely equipped for handeling nested procedures, the code generation part is not yet capable of generating code to go with the syntax. The problem lies in finding the address of intermediate frames to address. For this some extra code has to be generated, which will link all frames together. A excellent example of the straightforward implementation of

these frames can be found in: **PASCAL IMPLEMENTATION, The P4 compiler** by Pemberton and Daniels. [Pem82]
So for the time being only two levels of variables are accessible: Global and local variables.
Global variables are recognized by a indicated difference in lex-level of -1. The address going along with it is the offset in the data space. For hardwiring the address, the base address of the data space has to be added. The instruction used is:

Fmove <TOS>, [<address16>]:32

Local variable are in the current frame, and can thus be addressed through the frame pointer. Local variables can be identified by a difference in lex-level of 0. The address indicates the offset to the FP, if FP is situated at the bottom of the frame. This is the case after a regular procedure call. The instruction used is:

PpushFP <frameoffset>

With the last instruction is of course, implied that the offset of the address is found within an offset of 127 quads from the frame pointer. In the current **PL/0**should this not cause any problems, since the creation of a 127 items in the current frame requires 127 separate declarations of variables. If, however, arrays are implemented this could case some problems. But for variables of this size the External frame pointer was added to the processor.

- sto

The remarks made for the **lod** generic instruction also hold for the **sto** instruction.
For global references is the instruction used:

Fmove [<address16>]:32, <TOS>

For local references:

PpopFP <frameoffset>

- int { increment the stackpointer }

This instruction is a sort of leftover from the original **PL/0**-code version. It is currently not used however. And there thus is there no direct translation for it.

- opr { not yet defined instructions }

This instruction was used as a generic operation instruction. The parameters indicated what type of operation was to be performed. Currently it is used as an instruction to test some of the extensions to be made to the instruction set. If the expansion is in accordance with the requirements a new enumerated field is added to the above instructions. Currently there are no such extensions.

C.3.2 CAUTION with addressing of items.

The addressing in the "C"-processor is dependant on the item to be accessed. If the item is on the stack and is referenced through either SP or FP than the offset quantity is QUADs. However, if the item is not referenced thru either SP or FP than all can be addressed with byte offsets.
Since in the **PL/0**- compiler all items are off QUAD size, this means that hardwired addresses for either jumps or references in the global frame have to be byte addresses, of which the lower 2 bits are always zero. The addresses are then obtained by multiplying the QUAD address by 4. References on the stack are not multiplied by 4, since all references are expressed in QUADs.

C.4 Remarks about extensions and improvements to the PL/0-compiler.

The current compiler is no more or less than the skeleton given in [Wir76]. Improvements can be made in either optimizing some (or most) of the code generation, or by extending the language with extra features. Of both will be given an example below. But be it said that there are more than just these.

C.4.1 Peephole code optimisation.

Consider the following expression:

C := A + B

(Where A, B and C are found in the local frame)
Currently the code for this would be:

```
PpushFP<offset_A>   2 bytes
PpushFP<offset_B>   2 bytes
Padd                1 byte
Ppop    <offset_C>  2 bytes
```

Which requires 7 bytes and (probably) 4 cycles.

With use of the instruction **PaddFP** the code stream would be:

```
PpushFP<offset_A>   2 bytes
PaddFP <offset_B>   2 bytes
Ppop    <offset_C>  2 bytes
```

Which requires 6 bytes and (probably) 3 cycles.

This is only a small example of what can be done by adding more instructions to the possible output of the **PL/0**-compiler. It also has to effect that more of the instructions of the processor are being used, and hence would the testing of possible cache algoritms become more realistic.

C.4.2 Extending the language.

Currently the language has procedures which are called without parameters and return no result value. Various programming methods however are based on recursive programming, for which the parameters and return values are more or less essential.
The parameters can be bypassed by devoting global names to the input parameters, and than the first this to do in the called procedure is to copy the global values into local variables. This frees the global parameters/variables, and they can be reused.
The return value could also be handled this way, but the modification to the compiler to implement functions will be very modest.
The object definition has to be expanded with an entry for a function, and so has the reserved word list. The parser has to be enriched with an if-then part that tests for a FUNCTION token. Once this is found the processing continues as if a procedure was found.
The main difference are in:

- Calling the function: An extra field is left open on the stack before the function block is called.
- When an assignment to the function name is made, the address generated with the variable name will not be in the local frame but to this extra created space.

The result of all this is that after the function call the < TOS > value will contain the function result.

C.4.3 Other changes.

Other possible changes to the compiler are given by the questions acompanying in [Wir76, chapter 5]. Unfortunately the answers are not in the book.
More demanding "improvements" are to be made to the conversion from the intermediate code to the "C"-code. Currently this is done in a disorganized fashion. Cleaning up would give the compiler a better performance and a nice implementation.

Making changes to this compiler will be a good experience for those that are going to tackle the bigger problems for this processor. Rewriting the code generators for the **P4** compiler, or for **the Small-**"C"compiler

Appendix D

The UNIX features used during the development of the system

The original SL2010 package is developed to run under the Apollo operating system Aegis. This system is more or less like Unix some of its features are better, some of them are worse.
One feature which is certainly better on the Unix system is the writing of script files. For this are man more tools available than there are under Aegis. For this purpose and for the convenience of Unix are all used commands "transferred" to the Unix environment.
Transporting the tools encompasses nothing more then executing the SL2010 commands from an Aegis script. An Aegis shell is characterized by the command `#!/com/sh` on the first line. Scripts of this kind can be called by either Aegis or Unix shells, commandline parameters are passed correctly.
Using the above features, script file are set up that will automate the compilation and linking of the generated modules. As an additional bonus to this are the compiler and linker in the SL2010 package (remote) runable from sites which do not have a licence for the package.

D.1 Using the m4 package.

The programming language HHDL is a PASCAL based language. Most structures are exact copies of their counterparts in PASCAL. The extensions to PASCAL are however large and complex. Most of these have to do with mapping variables to connections in the hardware design. Or they are used to convert the sequential nature of PASCAL into a semi parallel model, fit to execute a hardware model.
Due to restrictions in the used HHDL compiler are there restrictions placed on the usage of the scheduling statements. They are allowed in only two places:

- In the main body of the component.
- And in the asynchronous sections with the component.

It is however not allowed to use these statements in procedures of functions.
This causes a great deal of problems since it is very customary to put often used parts of code in small subroutines. In HHDL is this not always possible.

To prevent the code from growing immensely the m4 macro expander is used to place this code inline. The places to put this code is indicated by "macro calls". As a consequence of this the resulting code file will still be quite large. But the amount of code is derived in a different fashion. Possible errors in "macro sub routines" are easier to correct, since they need fixing in only one place. Not every place of the same code has to be changed by hand

A second instance where the use of the macro expander became very useful was the "fix" for the "feature" in the WAITFOR statement. The initial interpretation was that this statement would wait for the required conditions to become true. If they were already true then the statement would be skipped. It turned out that the statement would first wait for the line to have a transition, and only then was the condition checked. One possible way to correct the risen problem was to go through all modules and fix the code with an IF-statement to test the condition first.
With the m4 expander another approach was taken:
A macro TESTWAITFOR was created, and this new macro was used at all instances where the required test was needed.

A third example lies in the communication and the handshaking that goes with it. Currently many tests are of the shape: test for ready, give command, wait for busy, wait for ready, deactivate command. It is possible to write the full code every time this is needed. It would however be more convenient to put it in a subroutine. Alas is this not possible, due to the restrictions. So for most of these instances is a macro written which will do all handling of the handshaking. The parameters of the macro indicate the condition and the lines to check.

D.2 Using the SCCS system.

When creating a large and complex system it is usually a big problem to keep track of all changes made. Which changes are made very, when and how.
Unix has a set of tools which will do most of the above mentioned tasks. The SCCS system (SOURCE CODE CONTROL SYSTEM) creates an archive in which it keeps all the source files. The system keeps information on the changes made to all source files.

- Who made them?
- When were they made?
- What is changed?

Next to that is it possible to use some of this information in the source itself, which will indicate version numbers etc. will running the programs.

Another facility that comes with the SCCS system is the management. If somebody (Whom is authorized to do so) wants to change the source, he or she issues a request which is only honored when nobody else is currently using the file for a change action.
The system makes it possible to share source without to much trouble of keeping the correct set of programs and files without denying people the strict right to make changes. And thus if this system is used with the required tools, some problems can be prevented.

In the current design are only the HHDL sources in the SCCS file system. It will be the plan to use the SCCS system also for the software model, the testfiles. And perhaps it would also be wise to use it for the compilers and other tools which are going to be written.

D.3 The used files and directories.

A design directory is created in which the SL2010 package will work and put it required files. This directory is also used as root for all other subdirectories in the design.
For every module at the top level is a subdirectory created in which all files belonging to this part of the design are stored.

The following extensions are used:

.hdl A HHDL source file

.hd4 The macro extension of the matching .hdl file. This file is created by running the .hdl version through the UNIX m4 package. A file needs m4 expansion if it has the string `(*M4*)` on the first line of the original .hdl file.

.lis The output listing of the HHDL compilation of the matching .hdl file.

_drv.hdl The model is tested with the aid of a driver. The driver could also have the macro expanded version.

_drv.hd4 Driver m4 version.

.simlink Used for batch controller linking of the module. It contains all interactive commands to make a model for testing.

.bat Input control file used for redirection during simulation of a module. It will usually contain the commands to start the simulation, and to terminate the execution. Assignments, etc. are made in the accompanying .scl file.

.log The redirected output of the simulation of a module. This will be the regular screen output.

.scl The simulation control file which is read by the HELIX simulator, once it starts up.

.bsc The HELIX simulator produces an output file with entered HELIX commands which should be reuseable. Unfortunately are there some bugs, and is the file more or less useless.

.trc The Helix simulator has an option to switch on tracing. The resulting output can be found in this file.

.fcl The output of the simulation can be printed nicely formatted on paper. For this is a control file needed. The program FORMT does however contain several bugs, and the control language is not easy to use.

.fmt The output of the FORMT program is written to this file. It is a ready printable version.

The directories used for the modules are:

bus: The bus interface.

exec: The execution unit.

icache: The instruction cache.

instr: The instruction unit. With instruction fetcher and instruction decoder.

operand: The operand unit. It contains offset fetchers, operand calculators, store result, stack window cache, the controller and a “translator”.

package: All defined interfaces have their signal values defined in packages. Some of the general definitions can be found in the files:

def.hdl
gen.hdl

registers: The register module comprises several modules. All registers are include, as are the command and status translators

total: All modules put together should render a working system. most of the commands to make and/or initially test the system as a whole can be found here.

Other used directories are:

com: For developing the designs several script files are developed. (nothing really fancy)

dbase: Of compilations and linking tries a log is kept. The intention is to create a database from it. The database should keep version of tested and changed files in line with one another.

def: Files which are often includes, or contain standard remarks are found here.

hplt: Plotter output is kept here. .plt for plot files, .plt.opt for optimized plotfiles.

macro: Standard macro texts ready for usage in other modules. They can be include with the m4 processor.

sccs: The Unix SourCe Control System is used to keep so line in the created source files. Currently are only the HHDL files entered in this system. But the simulator control files should be entered too. Once this is done, it will be possible to keep record of simulator and the used HHDL versions and the used simulator versions.