EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Design of an embedded microprocessor for array intensive tasks

Wijffels, A.

Award date:
1996

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

7410 EB610

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

Design of an embedded microprocessor for array intensive tasks.

ing. A. Wijffels

Coach : dr. ir. A.C. Verschueren
: ir. L.C. Benschop
Supervisor : prof. ir. M.P.J. Stevens
Period : July 1995-April 1996

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses

Preface

In 1985 I got my Bachelor's degree in Electronics and became an employee at the Eindhoven University of Technology. Three years later I decided to register as a student again and in my spare time started a study at the university to achieve my Master's degree. This report is the final part of that study.

During my years of study at the university I had the pleasure to study with my colleague and roommate Wido Kruijtzer. We have spent a lot of time together. In our social live we spent time together playing tennis, biking in the Belgium Ardennes and trying to win the first price of the Eindhoven University of Technology car rally. I thank him a lot for his support and company during those years.
I like to thank my boss and coach prof. ir. M.P.J. Stevens, who gave me the opportunity to realise this study. Also I like to thank him for this project which I enjoyed a lot. I like to thank my other coaches dr. ir. A.C Verschueren and ir. L.C. Benschop for their support on this work.
I like to thank all of my colleagues who supported me with my study and in particular our secretary Rian van Gaalen for her cheerfulness. Together with her and Frank Volf we had good times baking the 'oliebollen' for the 'end of the year celebration'. Also I like to thank Frank for his support of the computer management which gave me more time for my study. I still need his good support during the car rallies.
Furthermore, like to thank my wife Tineke for all the support she gave me during my study and especially for her care during the last few weeks in which I had to finish this report.
Last but not least I like to thank my parents for their support during all my years of education. I know this achievement makes them very proud.

Abstract

For his PhD thesis ir. L.C. Benschop has defined a VLSI circuit that can compress and decompress data in hardware at a speed of 100 Megabit per second. The functionality of the circuit has been analysed and for all functions the optimum implementation technique has been selected. The resulting architecture consists of several parts including an embedded microprocessor. This report explains the design of this microprocessor.
The processor to be designed must perform the intermediate rate, high complexity tasks of a lossless data compressor and decompressor. More specifically it must generate the optimum Huffman code from statistics of the data. Further it must encode and decode descriptions of these Huffman codes. These descriptions are included with the compressed data. Finally it must program the code into the Huffman encoding and decoding hardware. These algorithms use simple operations (like addition) and no multiplication or division. They do frequently access arrays in a random order. Therefore array indexing must be very efficient. Speed requirements are high for this embedded processor (target is 50 MHz).
The proposed processor has a Harvard architecture. This means that the program memory and the data memory are separate. Instructions and data can be accessed at the same time. The program memory is a ROM and the data memory is a RAM. A small part of the data address space will be mapped to I/O devices.
The step by step design path is shown according to the method as shown by Patterson and Hennessy in their book 'Computer organisation and design, the hardware/software interface'. First the datapaths are defined necessary to execute the several addressing modes of the binary operations. A stack pointer and the datapaths for the (conditional) branch instructions are added and finally a datapath is added to implement the interrupt feature.
To increase the throughput of the microprocessor pipelines are implemented. As the pipeline became too long the design specification was changed and the microprocessor was given a load-store architecture.
To decrease data hazards data-forwarding has been added.
The final design is modelled and simulated with IDaSS, an interactive design and simulation environment for synchronous digital circuits.
To know whether the timing requirements are achieved the design can be translated to VHDL. With a silicon compiler the design can then be implemented in silicon and a more accurate timing analyses can be done.

Table of contents.

Preface ... ii
Abstract ... iii

1 Introduction ... 1
1.1 Overview of the VLSI design project ... 1
1.2 Compression methods ... 2
1.3 The architecture ... 2

2 The microprocessor specification ... 5
2.1 Introduction ... 5
2.2 Overview of the architecture ... 6
2.3 Memory and I/O interfacing ... 8
2.4 The instruction set ... 9
2.5 Application-specific instructions ... 15
2.6 Interrupts ... 15
2.7 Variations ... 15
2.8 Modifications and supplements to the original concept ... 16
2.8.1 Changing to a load-store architecture ... 16
2.8.2 Changing the suggested encoding of the instructions ... 16
2.8.3 The displacement in case of a conditional branch instruction ... 20
2.8.4 The bidirectional databus changed to two separate busses ... 20

3 Designing an architecture for the embedded processor ... 21
3.1 Constructing a datapath for the program counter ... 21
3.2 Constructing the part of the datapath for the binary operations ... 21
3.2.1 Register to register binary operations ... 22
3.2.2 Direct addressing binary operations ... 23
3.2.3 Indexed addressing binary operations ... 24
3.2.4 Immediate addressing binary operations ... 25
3.2.5 Putting the parts together for the binary operations ... 26
3.3 Adding the part of the datapath for the unary operations ... 28
3.4 Adding the part of the datapath for al the jump operations ... 29
3.5 Constructing the part of the datapath for the other operations ... 31
3.5.1 Constructing a datapath for the fill instruction ... 31
3.5.2 Constructing a datapath for the rti instruction ... 33

4 Adding pipelines to the design ... 37
4.1 Chopping the datapath into pipe stages ... 37
4.2 Data hazards ... 44
4.3 Branch hazards ... 46

5 Modelling and simulation ... 51
5.1 The microprocessor environment ... 52
5.2 The fetch stage ... 53
5.2.1 The logic block namuxc ... 53
5.2.2 The logic blocks amux and imux ... 54
5.2.3 The finite state machine controller intctrl ... 55

5.3 The decode stage 59
5.3.1 The logic block decctrl 60
5.3.2 The logic block wbctrl 61
5.3.3 The logic block fbctrl 62
5.3.4 The logic block stall 63
5.3.5 The finite state machine controller idctrl 65
5.4 The execute stage 66
5.4.1 The logic block fbctrl 66
5.4.2 The logic block exctrl 68
5.5 The write-back stage 70
5.5.1 The logic block flgen 70
5.5.2 The logic block brlogic 72
5.6 Test verification 73
5.6.1 Testing of the instruction set 73
5.6.2 Testing of the interrupt feature 73
5.7 Critical path analyses 73

6 Conclusions and recommendations 81

Bibliographies 83

Appendix
A : The IDaSS models of the microprocessor 85
B : The assembler code for testing the instruction set 169
C : The assembler code of a realistic program 173

1 Introduction.

Modern software for personal computers asks for a lot of disk space. Three years ago one would buy a new PC with a harddisk of 100 MegaByte. That was sufficient at the time but nowadays 10 times that amount is needed. The explosive growth of software and information data makes it necessary to store data more efficiently. Software tools like 'DoubleSpace' have been introduced to compress data when written to hard disk and decompress when read from the disk.
Because of the growth of information data, the transport of this data increases also. The user of a communication line is charged for the amount of bits which he sends over the line. To decrease the amount of data to be transmitted and to gain speed the data can best be compressed before it will be transmitted and decompressed at the receiving side.

These techniques are not new and are already used on a large scale. But most of the time it is done by software, which makes it relatively slow.
For his PhD thesis ir. L.C. Benschop has defined a VLSI circuit that can compress and decompress data in hardware at a speed of 100 Megabit per second.

The VLSI circuit that must be designed has the following features:
- Optimised for lossless data compression of typical computer files, like English text and computer programs.
- A speed potential of 100 Megabit per second or more (rate of uncompressed data).
- A compression ratio comparable to that of widely used software methods (2.9 bits/byte).

Apart from these features there is a requirement that the design is sufficiently generic, so that it can be adapted to specific requirements.

The functionality of the circuit has been analysed and for all functions the optimum implementation technique has been selected. The resulting architecture consists of several parts including an embedded microprocessor. All functions have been mapped to the architecture.

1.1 Overview of the VLSI design project.

The VLSI circuit operates as a peripheral within a normal computer system. It is usable with a wide variety of computer bus architectures, employing a data bus of 8, 16 or 32 bits wide. It employs two DMA channels, one to receive the input data and one to write the output data. The device is programmable by the host system through I/O ports. It sends interrupts to the CPU in the host system whenever CPU attention is needed. CPU attention is needed for the allocation of new memory buffers for the DMA process, but not for steps of the compression process itself. The device can either be used for compression or for decompression, but it can only perform one of the tasks at a time.

1.2 Compression methods.

The device uses the LZH (Lempel Ziv+Huffman) method, which is a combination of two methods, sliding window coding (a variation of LZ77) and Huffman coding. The popular PC programs PKZIP, LHA and ARJ are known to use this method. Its compression ratio is considerably worse than that of the most advanced software methods (arithmetic coding and a method of context modelling), but better than either Huffman coding or Lempel-Ziv coding perform alone. At the moment, very high speed implementation of methods that perform better than LZH seems infeasible.
The format of the compressed data is compatible with that of the data produced by the LHA program.
True dynamic Huffman coding has been considered, but its algorithmic complexity was found too high for the required speed. Static Huffman coding is used. This means that blocks of data are collected and for each block the Huffman code is computed from the frequency statistics.

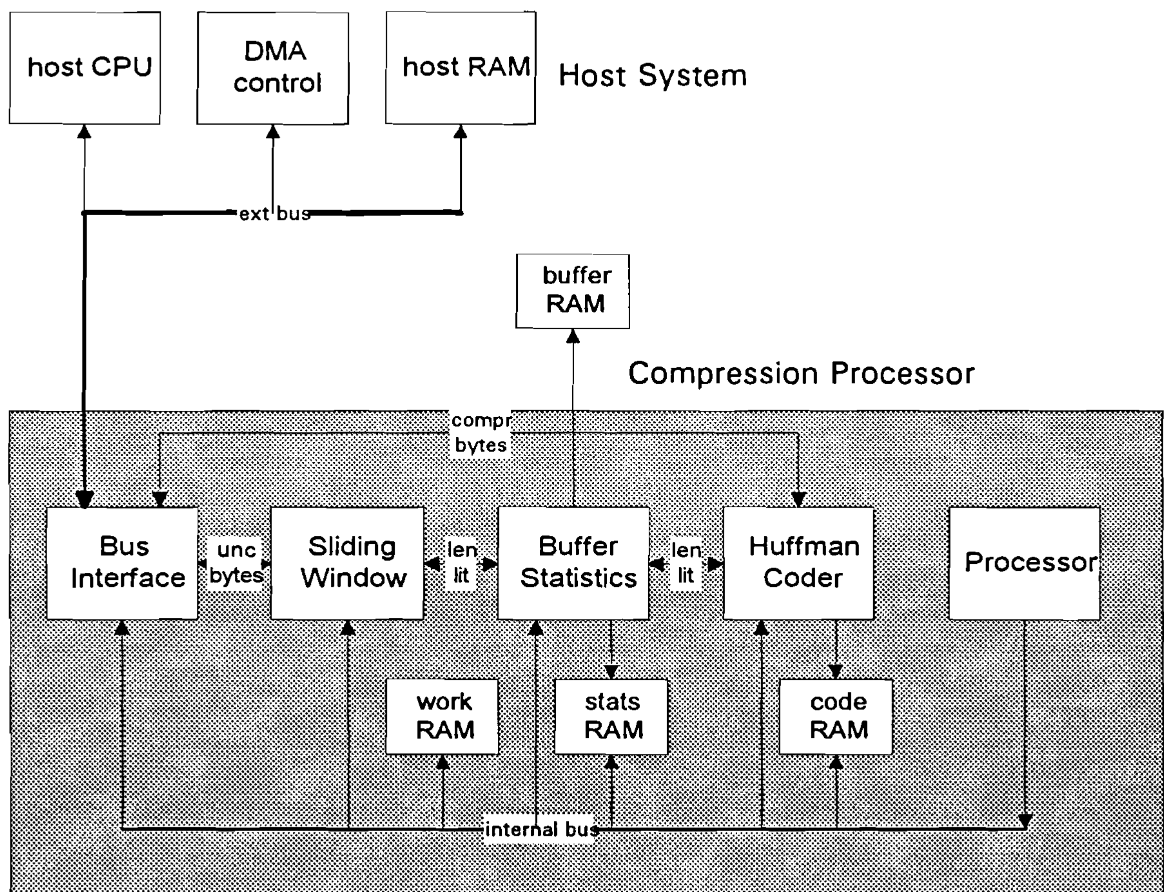


figure 1.1. The main modules of the VLSI circuit for data compression and decompression.

1.3 The architecture.

The device contains the following main modules as can be seen in figure 1.1:

- Bus interface
  It reads the input data and writes the output data through two DMA channels of the host computer. Further it communicates with the host computer through I/O registers and interrupts.
- Sliding window coder
  It performs compression or decompression using sliding window coding.
- Buffering and statistics module
  It stores blocks of lengths, literals and addresses into a separate external RAM and retrieves them later. When storing the data, it compiles frequency statistics. When the device is decompressing, the modules pass the data, using the external memory as a FIFO (first in/first out).
- Huffman coder
  It performs Huffman coding or decoding of lengths, literals and addresses, using code descriptions stored in a RAM. It also enables the internal processor to send and receive bit fields in the compressed data stream.
- Internal processor
  It controls all other main modules, it performs the Huffman tree computation and it sends and receives Huffman tree descriptions in the compressed data. Note that Huffman tree computation can be performed efficiently in software.

Sliding window coding, Huffman tree computation and Huffman coding are the stages of a high-level pipeline. They are performed concurrently on different data blocks.

There is an internal processor bus to which all main modules are connected. The other main modules have I/O ports through which they are controlled and give status. The modules can also generate internal interrupts. The internal processor is the only bus master on the internal processor bus. Only the bus interface module is connected to the host system bus.

This report contains the design of the embedded microprocessor. Chapter 2 describes the specifications of the microprocessor and the main parts it must contain. It will have a RISC architecture (reduced instruction set computing) and the instruction set and encoding is given. Chapter 3 describes the design of the datapaths that connect the main parts of the microprocessor, necessary to execute the instructions. To gain speed the microprocessor is given a pipelined architecture. The modifications to the datapath necessary to implement these pipelines are shown in chapter 4. A high-level design tool is used to implement the design. This is shown in chapter 5. It also contains an analyses of the critical paths. The final chapter gives the conclusions and recommendations.

2 The microprocessor specifications.

This chapter contains the specification of the embedded microprocessor as proposed by ir. L.C. Benschop. Section 2.8 contains some modifications on the original specifications.

2.1 Introduction.

Many ASICs perform tasks whose functionality is best implemented in software. They therefore contain a programmed embedded processor. This processor is integrated into the ASIC and it is not accessible from outside.

In a complex system we can distinguish several types of tasks.

- High rate tasks that cannot be performed at the required speed by a programmable computer. Dedicated hardware is needed to perform these tasks. The dedicated hardware performs a single or very few different operations at high speed. Sometimes many operations are performed in parallel.
- Intermediate rate tasks. These tasks can be performed by a programmed computer. If they are repetitions of a single operation they are usually not implemented that way as a complex device is tied up nearly all the time by a simple operation. If these tasks are very complex, they are implemented by a computer. The computer often has to have special capabilities, for example with digital signal processing.
- Low rate tasks. Any embedded processor has enough computing power to handle these. These tasks would usually require certain I/O facilities and an interrupt capability of the processor.

The processor to be designed must perform the intermediate rate, high complexity tasks of a lossless data compressor and decompressor. More specifically it must generate the optimum Huffman code from statistics of the data. Further it must encode and decode descriptions of these Huffman codes. These descriptions are included with the compressed data. Finally it must program the code into the Huffman encoding and decoding hardware. These algorithms use simple operations (like addition) and no multiplication or division. They do frequently access arrays in a random order. Therefore array indexing must be very efficient. Speed requirements are high for an embedded processor.

As the designed processor is the only processor in the device, it will also handle the low rate tasks. We decided not to use a second processor for the low rate tasks for the following reasons.

- The task performed by the device is not real-time. It is therefore acceptable that the intermediate rate tasks are interrupted for a short time to perform low rate tasks.

- The processor already needs a fairly extensive I/O capability to perform the intermediate rate tasks.

This section is a specification for a processor that meets the following requirements.

- The word size and the size of the addressable memory are adapted to the tasks that must be performed.
- The instruction set is adapted to the tasks that must be performed. Multiplication and division are left out whereas some unusual instructions are included.
- The processor must run at a very high speed.
- Addressing modes are optimised for random access of arrays.
- The instruction set is general enough to be used for typical low-rate control tasks.
- The processor interfaces to the rest of the system.
- It must be implemented in IDaSS and/or VHDL so it can be integrated into the rest of the design.

2.2 Overview of the architecture.

The proposed processor has a Harvard architecture. This means that the program memory and the data memory are separate. Instructions and data can be accessed at the same time. The program memory is a ROM and the data memory is a RAM. A small part of the data address space will be mapped to I/O devices.

Operations of up to 16 bit quantities are required, hence the data registers and memory locations are 16 bits wide. The proposed instruction encoding requires an instruction width of 16 bits as well. Program addresses are limited to 12 bits, but the only basis for this limitation is the encoding of the jump and jsr instructions. Extension to 13 or even 16 bits should be possible, if necessary. Data addresses are 16 bits, but they could be reduced, e.g. to 12 bits if that is sufficient. In the proposed application 12 bits would be sufficient. As a consequence the base registers and stack pointer could all be reduced to 12 bits.

The processor would have the following registers.

- index base registers
  There are up to 16 index base registers. They are added to an index register to obtain the address in indexed addressing mode. These registers are typically set to the start address of an array and remain constant during the execution of an algorithm. The index registers are used as the array index and vary far more frequently.
- direct base registers
  There are up to 2 direct base registers. They are added to a 6-bit direct address to obtain the address in direct addressing mode. These registers are typically set to a memory area for various variables and the I/O address area respectively.

The direct addressing mode is then used to address either one of 64 variables or one of 64 I/O ports.

- general purpose registers
  There are up to 8 general purpose registers. Arithmetic and logic operations always have a general purpose register as the destination. One of the source operands is the same register as the destination and the other source operand is either a general purpose register, a memory operand, or an immediate value. General purpose registers also serve as index registers in indexed addressing mode. If the word 'register' is used in this paper, its default meaning is one of the general purpose registers.

- program counter
  The program counter addresses an address in ROM where the next instruction will be loaded from.

- stack pointer
  The stack pointer addresses the top of stack in RAM. The stack will grow downward as registers or return addresses are pushed on it. The stack is used by the push/pop instructions, the jsr/rts instructions and by interrupts.

- status flags
  The status flags indicate the result of the last compare instruction, whether the result of the last instruction was zero or whether the result of the last instruction was all bits set.

The execution of an instruction takes the following steps.

- Fetch the instruction from ROM at the program counter address.
- Decode the instruction.
- Get the values of the register operand(s).
- Compute the address of the memory operand and address the memory.
- Get the value of the memory operand.
- Compute the result.
- Increment the program counter or assign a new value to it (jump).
- Store the result in a register if appropriate.
- Store the appropriate flag values.
- Check for interrupt, if it occurs save program counter and status flags, reload program counter with start address interrupt routine .

Many of these steps can be combined into one cycle by using combinatorial logic and pipelining.

The following operations will each take a clock cycle.

- Accessing the RAM for either reading or writing.
- Actually storing values in registers.
- Accessing the ROM for reading an instruction.

Each instruction accesses only one RAM location (either reading or writing), reads only one ROM location and modifies at most one register, not counting stack pointer, program counter or status bits.
By careful design it should be theoretically possible to finish each instruction in exactly one cycle. A practical pipelined processor will have certain inter-instruction dependencies that require at least one cycle between them.

2.3 Memory and I/O interfacing.

The processor has the following connections to the rest of the system.

- data
  This is a 16 bit wide bi-directional bus. It carries all data from and to the RAM and the I/O devices.
- address
  This is a 12 bit wide bus driven by the processor. It carries the address of the RAM location or the I/O device that is accessed.
- read
  This is one line driven by the processor. If set high, the RAM (or the addressed I/O port) must place a value on the bus at the next clock edge.
- write
  This is one line driven by the processor. If set high, the RAM (or the addressed I/O port) must accept the value on the data bus at the next clock edge.
- interrupt
  This is an input line for the processor. If it is high at the active clock edge, the processor interrupts its normal program flow.

The ROM is part of the processor itself and the connections to it are not described here.

The external interface is similar to a traditional processor bus, except that it is a synchronous design in which all bus cycles take exactly one clock cycle. If an address is placed on the address bus along with the read and write signals, the addressed data (either read from RAM or written to RAM) appears on the data bus during the next clock cycle and by that time the next address may already appear.

An address decoding circuit will decode the address into the signals necessary to select the various pieces of RAM and I/O devices. This decoder is primarily a combinatorial circuit. In the designed system a 4 kiloword address range is expected, divided into the following regions.

- 2 kilowords of RAM that is only accessible by this processor.

- About 1 kiloword of RAM that is swappable between the processor and the Huffman encoder/decoder. There are two 'banks' of this RAM, one being exclusively accessible by the Huffman coder and the other being exclusively accessible by the processor. While the Huffman coder uses one of the memory banks, the processor prepares the other bank for the next set of codes. The banks are swapped by setting one input to the address decoding circuit.

- About 0.5 kilowords of RAM that is swappable between the processor and the statistics collecting module.
- A 16 byte 'parameter RAM', which is shared between the processor and the bus interface of the device.
- Various control, status and other ports of the other modules. These are the I/O addresses, preferably mapped to one consecutive range. It is expected that less than 64 addresses are needed.

2.4 The instruction set.

The proposed processor must have the following instructions. The mnemonics of similar instructions on widely used processors are used.

add Add, binary operation.

and Logical bitwise and, binary operation.

cmp Compare, destination is not changed, only flags indicate whether destination was above, below or equal to the source operand, using unsigned binary arithmetic.

cpl Complement register (unary operation).

djnz (optional) Decrement a register and jump if it was not zero.

ja Jump if flags indicate Above.

jae Jump if flags indicate Above or Equal.

jb Jump if flags indicate Below.

jbe Jump if flags indicate Below or Equal.

je Jump if flags indicate Equal.

jmp Jump to ROM address.

jne Jump if flags indicate Not Equal.

jns Jump if All-set flag is not set.

jnz Jump if Zero flag is not set.

js Jump if All-set flag is set.

jsr Jump to subroutine at ROM address, push old program counter on stack.

jz Jump if Zero flag is set.

mhi Move immediate data to high 8 bits of destination register.

mov Move source operand to destination operand. Both can be registers, the source can be a memory or immediate operand, in which case the destination is a register. The destination can be a memory operand, in which case the source must be a register. If either of the operands is a non-general purpose register (base register, stack pointer), the other operand must be a general purpose register. As opposed to other instructions, mov occurs in many different instruction formats.

or Logical bitwise or (binary operation).

pop Pop a register from the stack (unary operation).

push Push a register onto the stack (unary operation).

rti Return from interrupt.

rts Return from subroutine.

set Set register to all ones (unary operation).

shl8 Shift a general purpose register left 8 places (unary operation).

shr Shift a general purpose register right (unary operation). There is no shl, because an add of the register to itself already does this.

shr8 Shift a general purpose register right 8 places (unary operation).

sub Subtract source from destination (binary operand).

xor Logical bitwise xor (binary operation).

Next to these instructions there are three more application specific instructions: sel2, sel3 and the fill operation. They will be discussed in section 2.5.

The instructions fall into several categories.

- Binary operations
  These are add, sub, and, or, xor, cmp, mov and mhi. There are 8 binary operations. mhi only occurs with immediate addressing. With the other addressing modes there are two different forms of mov, one to move the source operand to the destination register (load) and one to move the destination register to memory at the address of the source operand (store). With register to register operations, only seven binary operations are possible.

- Unary operations
  These are cpl, pop, push, set, shl, shr and shr8. These operations have one register as an operand. Up to 16 different unary operations can be specified. push and pop access memory and modify the stack pointer.

- Jump operations
  These are djnz, ja, jae, jb, jbe, je, jmp, jne, jns, jnz, js, jsr and jz. jmp and jsr have a 12 bit address that can be any ROM location. The other jump operations specify an 8-bit signed displacement with respect to the current program counter. At this stage of the specification it is not important whether or not the program counter was already incremented before the displacement is added to it.

- Other operations
  These are rti, rts and other application specific instructions.

The binary operations have the following possible addressing modes.

- register to register
  The source operand and the destination operand are both registers.

- direct addressing
  The destination operand is a register and the source operand is in memory at an address that is the sum of a direct base register and a 6-bit address specified in the instruction. There are two directly addressable areas of 64 locations each. One of the areas will be used to store certain variables and the other one will be the I/O area.

- indexed addressing
  The destination operand is a register and the source operand is in memory at an address that is the sum of an index base register and a general purpose register (used as index register). This is the primary mode to access arrays.

- immediate addressing
  The destination operand is a register and the source operand is an 8-bit value. The high bits of this value are 0. mov with an immediate operand sets the high bits of the destination to zero. mhi sets the high bits of the destination register to the immediate value and leaves the low bits unchanged.
  The proper way to set r1 to the value 1234h is therefore

  ```
  mov r1,34h
  mhi r1,12h
  ```

  The logical operations with immediate addressing are often needed to select certain flag bits in I/O operations. Therefore we will limit the width of I/O ports with flag bits to 8 bits.

A possible encoding for the instruction set with each instruction fitting in 16 bits is shown in table 2.1. The fields have the following meaning.

addr12 12-bit address in jmp or jsr instruction.

bd 1-bit direct base register specifier.

basei 4-bit index base register specifier.

bop 3-bit specifier for binary operation.

cond 4-bit specifier for jump condition.

direct6 6-bit direct address.

disp 8-bit signed displacement in jump conditions.

imm8 8-bit immediate value.

jop 1-bit specifier to select jmp to jsr.

regd 3-bit specifier for destination register.

regi 3-bit specifier for index register.

regs 3-bit specifier for source register.

uop 4-bit specifier for unary operation.

It is not required (maybe not desirable) to use the exact same encoding, but this one shows that it is possible to encode all instructions in 16 bits.

Table 2.2 contains the encoding of the several fields of the instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	---basei---				--regi--			--bop---			--regd--			Binary operation, indexed
0	0	1	bd	-----direct6-----						--bop---			--regd--			Binary operation, direct
0	1	-----------imm8-------								--bop---			--regd--			Binary operation, immediate
1	0	0	jop	------------paddr----------------												JMP, JSR
1	0	1	0	------------disp8------								---cond----				Conditional jump
1	0	1	1	------------disp8------								x	--regd--			DJNZ
1	1	0	0	x	x	x	x	x	----uop----				--regd--			Unary operation
1	1	0	1	x	x	x	--regs--			--bop---			--regd--			Binary operation, register
1	1	1	---basei---				--regi--			0	0	0	--regs--			FILL
1	1	1	---basei---				0	0	0	0	0	1	--regs--			Mov to base register
1	1	1	x	x	x	x	0	bd	1	0	0	1	--regs--			Mov to direct base reg
1	1	1	x	x	x	x	1	0	0	0	0	1	--regs--			Mov to SP
1	1	1	x	x	x	x	x	x	x	1	1	0	x	x	x	RTS
1	1	1	x	x	x	x	x	x	x	1	1	1	x	x	x	RTI

table 2.1. One way to encode the instructions.

Binary operations (bop field)

0 0 0	Mov
0 0 1	Mov (reverse direction), Mhi with immediate mode
0 1 0	Add
0 1 1	Sub
1 0 0	And
1 0 1	Or
1 1 0	Xor
1 1 1	Cmp

Unary operations (uop field)

0 0 0 0	Pop
0 0 0 1	Push
0 0 1 0	Cpl
0 0 1 1	Set
0 1 0 0	(reserved)
0 1 0 1	Shr
0 1 1 0	Shl8
0 1 1 1	Shr8
1 0 0 0	Sel2
1 0 0 1	Sel3

Jump operations (jop field)

0	Jmp
1	Jsr

Jump conditions (cond field)

0 0 0 0	Jbe (below or equal)	\
0 0 0 1	Ja (above)	|
0 0 1 0	Jb (below)	| Last compare
0 0 1 1	Jae (above or equal)	|
0 1 0 0	Je (equal)	|
0 1 0 1	Jne (not equal)	/
0 1 1 0	Jz (zero)	\ Zero flag
0 1 1 1	Jnz (not zero)	/
1 0 0 0	Js (all set)	\ All-set flag
1 0 0 1	Jns (not all set)	/

table 2.2. Encoding of the bop. uop, jop and cond sub fields.

2.5 Application-specific instructions.

For the specific application of Huffman tree computation the following special instructions are desired.

fill Fill a memory region with a certain value. The value is in the source register. The memory region is specified by an index base and an index register. The source register is stored in memory at the address that is the sum of the index base register and the index register. Then the index register is decremented by one. If the index register is decremented below zero (it then contains all 1 bits), the instruction terminates, else the instruction does not increment the program counter and repeats itself.
This way a region could be filled one location per clock cycle.

sel2 Select second field (unary operation). Shift the destination register right four places and `and' with 15.

sel3 Select third field (unary operation). Shift the destination register right eight places and `and' with 31.

2.6 Interrupts.

When the microprocessor is reset it starts execution at ROM address 0. When an interrupt occurs, the program counter and status bits are saved and execution proceeds at ROM address 1. The rti instruction restores the program counter and status bits that were saved by the interrupt. Only one interrupt is planned. The interrupt handler would poll the various possible interrupt sources to see which event caused it. Between an interrupt and an rti, the interrupt is disabled.

As program addresses are 12 bit and there are four status bits (two for compare result, an all-set flag and a zero flag), it would be possible to combine them in one 16-bit word and push that onto the stack.

Alternatively one can use a special-purpose register to save the program counter and status bits.

2.7 Variations.

It would be possible to simplify the processor considerably if the following modifications could be carried out.
Whether these variations are acceptable is highly dependent on the program that is run on the processor. It is probably only known after the software is written.

The following variations are proposed.

- Leave out the stack pointer, the push and pop instructions and the jsr/rts instructions. For the interrupt facility use a special-purpose register to save the program counter and the status.

- Hardwire all or some base registers to fixed locations. These registers can than be replaced by ROM cells, or even a combinatorial circuit. For the direct base registers, this is almost certainly possible.

2.8 Modifications and supplements to the original concept.

This section describes the modifications made to the original concept during the course of the design.

2.8.1 Changing to a load-store architecture.

All binary operations can also be performed on data which comes directly from the Data Memory. When implementing pipelines in the design it became clear that there were two extra pipeline stages necessary to make this possible. These extra pipeline stages would cause some extra stalling of the throughput of the pipeline. Therefore the decision was made to change the concept and to give the processor a load-store architecture. This means that no operations can directly be done on data coming from or going to the Data Memory.
If an operation must be performed on data from the Data Memory, this data must first be transferred to one of the General Purpose Registers. Then the operation can be performed on the data after which it will be written back to one of the General Purpose Registers. Then this result can be written back to the Data Memory.

2.8.2 Changing the suggested encoding of the instructions.

In the assignment suggestions were made for the instruction encoding. After a closer study some changes have been made.

The suggested encoding of the instructions for "direct binary operations" and for "move to direct base register" were:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

 0  0  1 bd -----direct6----- --bop--- --regd--  Binary operation, direct

 1  1  1  x  x  x  x  0 bd  1  0  0  1 --regs--  Mov to direct base reg
```

As bit 12 of the "mov to direct base reg" is a don't care it would be more logical to use that same bit for assigning which direct base register is to be used.

The encoding then changes to:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	bd	-----direct6-----						--bop---			--regd--			Binary operation, direct
1	1	1	bd	x	x	x	0	0	1	0	0	1	--regs--			Mov to direct base reg

Another change has been made to the encoding of the "Mov to ... " instructions. With the aforementioned change the encoding would have been:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	---basei---				0	0	0	0	0	1	--regs--			Mov to base register
1	1	1	bd	x	x	x	0	0	1	0	0	1	--regs--			Mov to direct base reg
1	1	1	x	x	x	x	1	0	0	0	0	1	--regs--			Mov to SP

Less hardware for instruction decoding can be used if bits 6 to 8 each have a one in a different place. This changes the encoding into:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	---basei---				0	0	1	0	0	1	--regs--			Mov to base register
1	1	1	bd	x	x	x	0	1	0	0	0	1	--regs--			Mov to direct base reg
1	1	1	x	x	x	x	1	0	0	0	0	1	--regs--			Mov to SP

It is necessary to have an instruction which does nothing, the so called NOP instruction. The best value for this instruction is all ones or all zeroes. In the suggested encoding the easiest way is to choose for all ones. This means that the bit assignment for the RTI instruction must be altered. It can be changed into:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	x	x	x	x	x	x	x	1	0	0	x	x	x	RTI

It will still remain having only one bit different (bit 4) with the RTS instruction which it is closely related to.
Next to the newly introduced NOP instruction there is also need for an instruction to tell the microprocessor to start an interrupt routine. This will be called the INT instruction. The NOP and INT instruction can be encoded as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	x	x	x	x	x	x	x	0	1	0	x	x	x	INT
1	1	1	x	x	x	x	x	x	x	1	1	1	x	x	x	NOP

The complete encoding as will be implemented is shown in table 2.3.

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0  0  0 ---basei--- --regi-- --bop--- --regd-- Binary operation, indexed
 0  0  1 bd -----direct6----- --bop--- --regd-- Binary operation, direct
 0  1 ------------imm8------- --bop--- --regd-- Binary operation, immediate
 1  0  0 jop-------------paddr----------------- JMP. JSR
 1  0  1  0 ------------disp8------ ---cond---- Conditional jump
 1  0  1  1 ------------disp8------  x --regd-- DJNZ
 1  1  0  0  x  x  x  x  x ----uop---- --regd-- Unary operation
 1  1  0  1  x  x  x --regs-- --bop--- --regd-- Binary operation, register
 1  1  1 ---basei--- --regi--  0  0  0 --regs-- FILL
 1  1  1 ---basei---  0  0  1  0  0  1 --regs-- Mov to base register
 1  1  1 bd  x  x  x  0  1  0  0  0  1 --regs-- Mov to direct base reg
 1  1  1  x  x  x  x  1  0  0  0  0  1 --regs-- Mov to SP
 1  1  1  x  x  x  x  x  x  x  0  1  0  x  x  x INT
 1  1  1  x  x  x  x  x  x  x  1  0  0  x  x  x RTI
 1  1  1  x  x  x  x  x  x  x  1  1  0  x  x  x RTS
 1  1  1  x  x  x  x  x  x  x  1  1  1  x  x  x NOP
```

table 2.3. The encoding of the instructions as they will be implemented.

Next to the encoding of the instructions are some modifications made to the encoding of the unary operations. The suggested encoding was:

Unary operations (uop field)	
0 0 0 0	POP
0 0 0 1	PUSH
0 0 1 0	CPL
0 0 1 1	SET
0 1 0 0	(reserved)
0 1 0 1	SHR
0 1 1 0	SHL8
0 1 1 1	SHR8
1 0 0 0	SEL2
1 0 0 1	SEL3

From these instructions only the POP and the PUSH operation are not using the ALU. The binary operations consist of 8 different ALU-operations. The unary operations also have 8 different ALU-operations. By changing the first bit of the unary operations needing the ALU into a one, it is easier to generate the control signals for the ALU. Binary operations will start with a zero and unary operations will start with a one.
The encoding also has one encoding free (reserved). This will be used for the 'decrement' operation which is necessary in case of an DJNZ instruction.
The new encoding will now be:

Unary operations (uop field)	
0 0 0 0	POP
0 0 0 1	PUSH
1 0 1 0	CPL
1 0 1 1	SET
1 1 0 0	DEC
1 1 0 1	SHR
1 1 1 0	SHL8
1 1 1 1	SHR8
1 0 0 0	SEL2
1 0 0 1	SEL3

As the dec operation is added for the djnz instruction this unary operation can now also be used as an extra instruction.
The binary operations have two possible MOV operations: one to transfer data from one of the General Purpose Registers to the Data Memory and one to transfer data the other way. To make it more clear which is which:

bop field		
0 0 0	MOV	is used for data transfer from memory to a register (load)
0 0 1	MOV	is used for data transfer from a register to memory (store)

One encoding which was failing from the suggested encodings was the flag assignment. The assignment will be:

flag bit 0 :	all zero
flag bit 1 :	all set
flag bit 2 :	smaller
flag bit 3 :	greater

The 'smaller' and 'greater' flags will only be changed by a CMP instruction (compare). No flag will change by the instructions JMP, JSR, RTS, NOP, INT and one of the conditional jump instructions, except for the DJNZ instruction which is able to change the flags.
In case of an interrupt the return address will be saved on the stack together with the flags. On a 'return from interrupt' all flags will be restored from memory.

2.8.3 The displacement in case of a conditional branch instruction.

In section 2.4 the specification left open whether or not the program counter was incremented before the displacement is added to it in case of a conditional branch instruction. In the final design in case of a conditional branch instruction the displacement is added to the incremented program counter contents.

2.8.4 The bidirectional databus changed to two separate busses.

In the specification the external databus of the embedded microprocessor is a bidirectional databus. The disadvantage is that the bus can not write to the Data Memory one clock cycle after it has done a read operation from that Data Memory as both operations then need the databus at the same time.
To prevent these unnecessary stalls and extra tri-state buffers the databus is separated into a data-input-bus and a data-output-bus.

3 Designing an architecture for the embedded processor.

This chapter will describe the steps which were taken in designing this microprocessor. During the course of the design the decision was made to give the processor a load-store architecture. This means that no operations can directly be done on data coming from the Data Memory. To show why this decision was made the original idea will first be worked out. When appropriate I will change to the load-store concept.

The processor will contain the following major parts:
- an Instruction Memory (4Kx16 ROM)
- a Data Memory (4Kx16 RAM)
- 8 General Purpose Registers (GPR)
- 16 Index Base Registers (IBR)
- 2 Direct Base Registers (DBR)
- 1 Stack Pointer Register (SPR)

The first step is to construct a datapath to interconnect all these parts. This is done according to the design concept presented by Patterson and Hennessy in [Pat94]. A reasonable way to start a datapath design is to examine the major components required to execute each type of instruction. By looking at which datapath elements each instruction needs the sections of the datapath will be built up from these elements for each instruction type. When the datapath elements are shown initially, the control signals will be shown. To prevent the figures to be blurred by all the control signals they will be left out until the final datapath will be presented.

The instructions fall into several categories:
- binary operations
- unary operations
- jump operations
- other operations

3.1 Constructing a datapath for the program counter.

A 12 bit register can be used as the Program Counter (PC). The program counter addresses the instructions, which are stored in the Instruction Memory (Read Only Memory). The program counter can be incremented by an adder to be able to address the next instruction. The datapath for this part is shown in figure 3.1.

3.2 Constructing the part of the datapath for the binary operations.

The binary operations are *add*, *sub*, *and*, *or*, *xor*, *cmp* and *mov*. These operations, except for *mov*, can best be performed by an ALU. As can be seen in table 2 the *mov* operation has two different encodings and will be explained for each addressing mode.

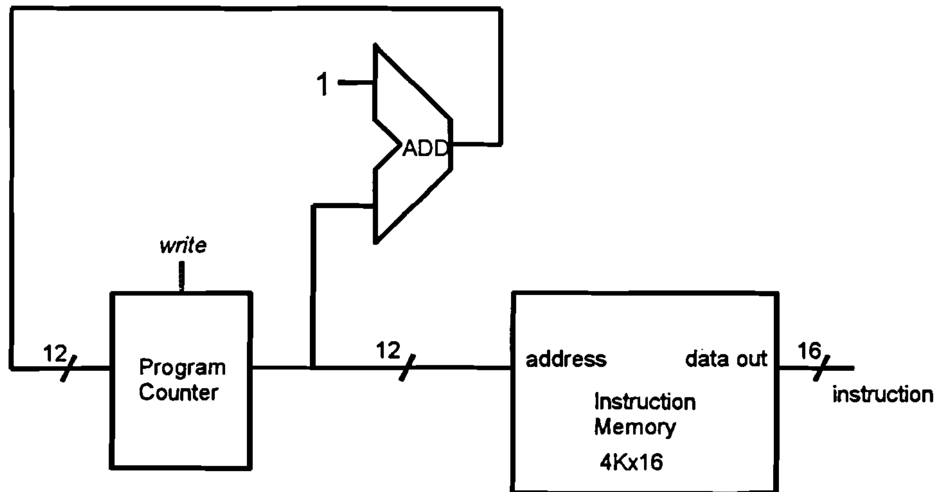


figure 3.1. A part of the datapath used for fetching instructions and incrementing the program counter.

The binary operations can be divided into the following addressing modes:

- register to register
- direct addressing
- indexed addressing
- immediate addressing

3.2.1 Register to register binary operations.

The two operands both come from a General Purpose Register. The ALU performs an operation on the data. The result of the operation is always written back to source one of the source registers. This results in a datapath as shown in figure 3.2.
In case of the *mov* operation, the ALU functions as a multiplexor and only passes the data. The data can only be copied (*mov*) from source2 to source1 (destination). For this mov operation the encoding 000 will be used. The *mov* encoding 001 will not be used with this addressing mode.

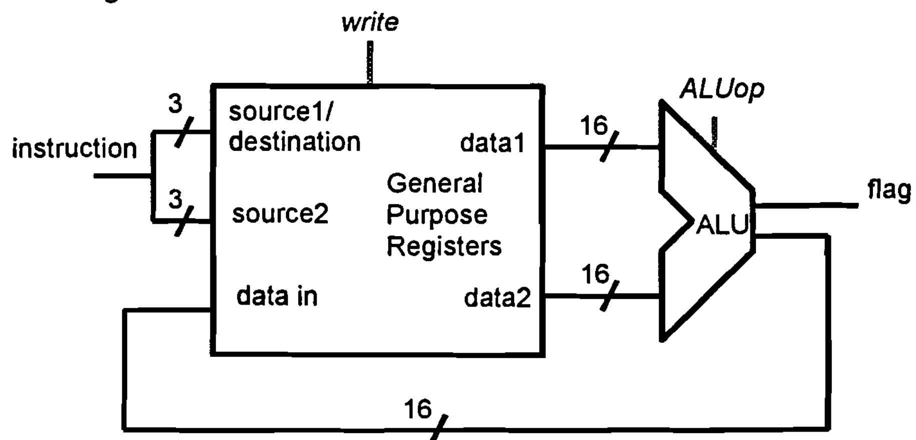


figure 3.2. The datapath for register to register binary operations.

3.2.2 Direct addressing binary operations.

These operations must also be handled by an ALU. With direct addressing one operand comes from one of the General Purpose Registers and the second operand for the ALU comes from a memory location. The memory is addressed by the sum of the contents of a selected Direct Base Register and a six bit number specified in the instruction. This six bit number is complemented with 6 zeroes to form a 12 bit number. This number can be added by the output from the selected Direct Base Register by means of a dedicated address adder.
The *mov* operation must function in both directions, i.e. from a General Purpose Register to a memory location and vice versa. In that case the ALU will have to function as a multiplexor. For data transfer from a General Purpose Register to the Data Memory the encoding for the *mov* operation is 001. For data transfer from the Data Memory to a General Purpose Register the encoding for the *mov* operation is 000.
The result of all other operations must always be written back to the selected General Purpose Register which was also functioning as a source.
The Direct Base Registers can be written to by one of the General Purpose Registers. Only the least significant 12 bit of the dataword will be used as these registers are only 12 bit wide. The datapath is shown in figure 3.3.

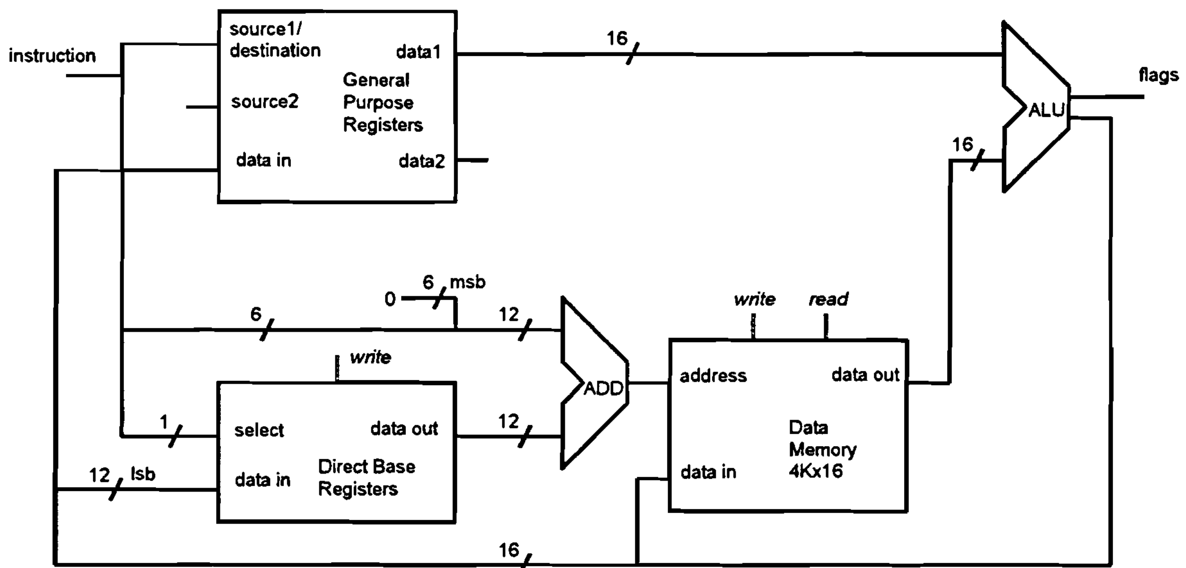


figure 3.3. The datapath for direct addressing binary operations.

The only way here to write data into a Data Memory location is a mov operation from a General Purpose Register to the Data Memory. There are no other sources that can write to the Data Memory. In figure 3 the data from the General Purpose Register is first passed through the ALU and then written into memory. It is very simple to shorten the path by connecting the output of the General Purpose Registers directly to the Data Memory. This modification is shown in figure 3.4.
The transfer of Memory data to a General Purpose Register will still go through the ALU. Writing data into one of the Direct Base Registers doesn't have to go through the ALU either. The only source for the Direct Base Registers is the data1 output of the General Purpose Registers. So a direct path can be made of only the least significant 12 bit.

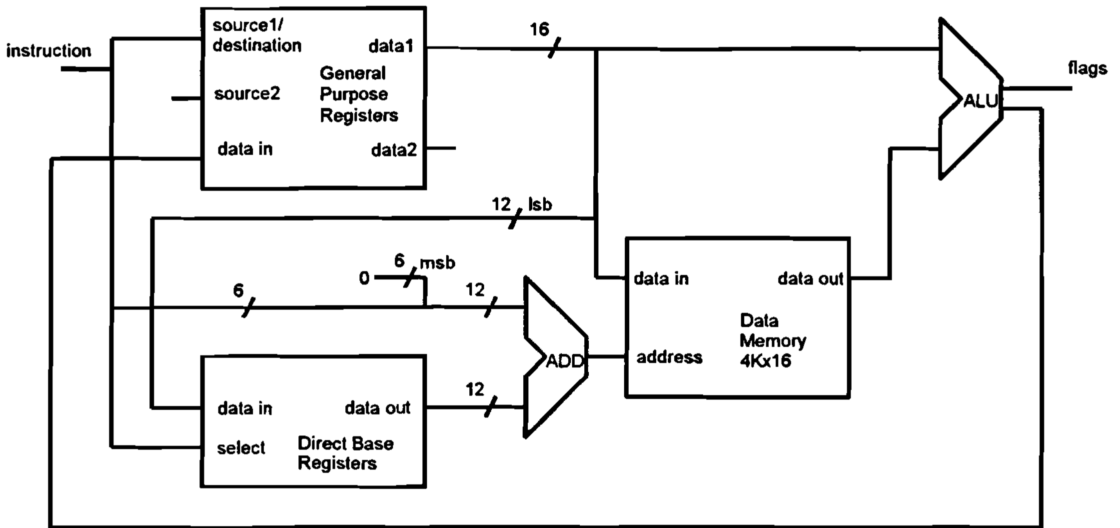


figure 3.4. The modified datapath for a register to memory data move operation.

3.2.3 Indexed addressing binary operations.

With indexed addressing the memory address is the sum of the contents of a General Purpose Register (used as an index register) and the contents of a selected Index Base Register. From the contents of the General Purpose Register only the 12 least significant bits are used as the address is only 12 bit wide.
Again the *mov* operation must function in both directions, i.e. from a General Purpose Register to a memory location and vice versa. For data transfer from a General Purpose Register to the Data Memory the encoding for the *mov* operation is 001. For data transfer from the Data Memory to a General Purpose Register the encoding for the *mov* operation is 000. The result of all other operations is always written back to the selected General Purpose Register.
Like with direct addressing binary operations, data transfers from a General Purpose Register to the Data Memory don't have to pass through the ALU as well as data transfers from a General Purpose Register to one of the sixteen Index Base Registers.

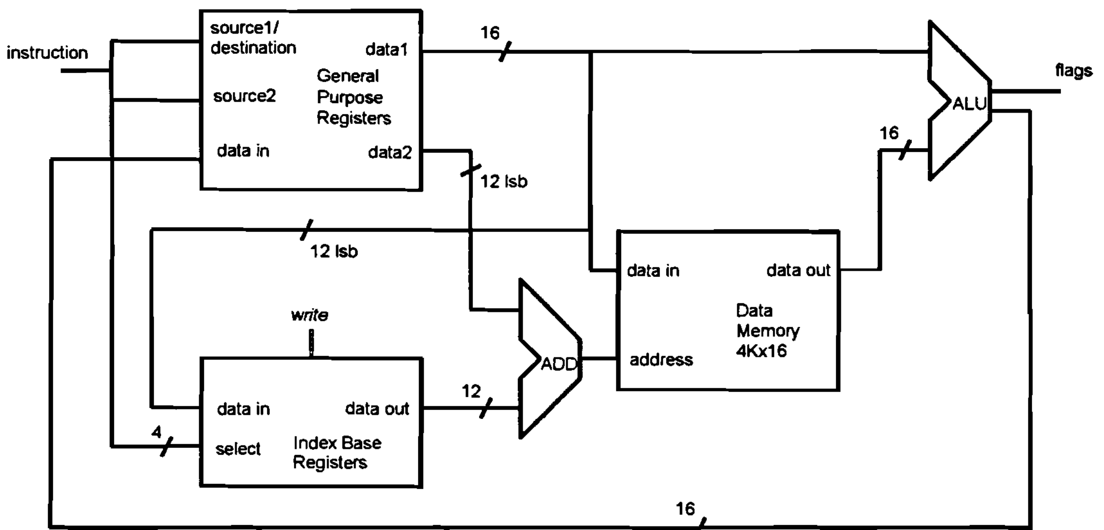


figure 3.5. The datapath for indexed addressing binary operations.

From the data written to a selected Index Base Register only the least significant 12 bits are used as the Index Base Registers are only 12 bit wide.
The resulting datapath for indexed addressing is shown in figure 3.5.

3.2.4 Immediate addressing binary operations.

With immediate addressing one operand is the contents of a General Purpose Register. The lower 8 bits of the other operand is part of the instruction and the upper 8 bits are made zero. The result of the binary operation is written back to the selected General Purpose Register.
In case of a *mov* operation with encoding 000, the 8-bit value from the instruction (bits 6 to 13) will be used as the least significant bits of the result. The 8 most significant bits are made zero. This result will be written to the destination register. The other *mov* operation with encoding 001 will here be called an *mhi* instruction (move high).
In case of an *mhi* instruction the 8-bit value from the instruction will replace the upper 8 bits of the source/destination register and leaves the lower bits unchanged.
With the register-to-register, indexed and direct addressing modes the *mov* operation with encoding 000 pass the data through the lower part of the ALU. The mov operation with encoding 001 has so far not been going through the ALU. To keep the ALU as simple as possible it would be convenient to have all *mov* and *mhi* operations pass through the lower input port of the ALU. A possible implementation for the datapath of this addressing mode is given in figure 3.6.

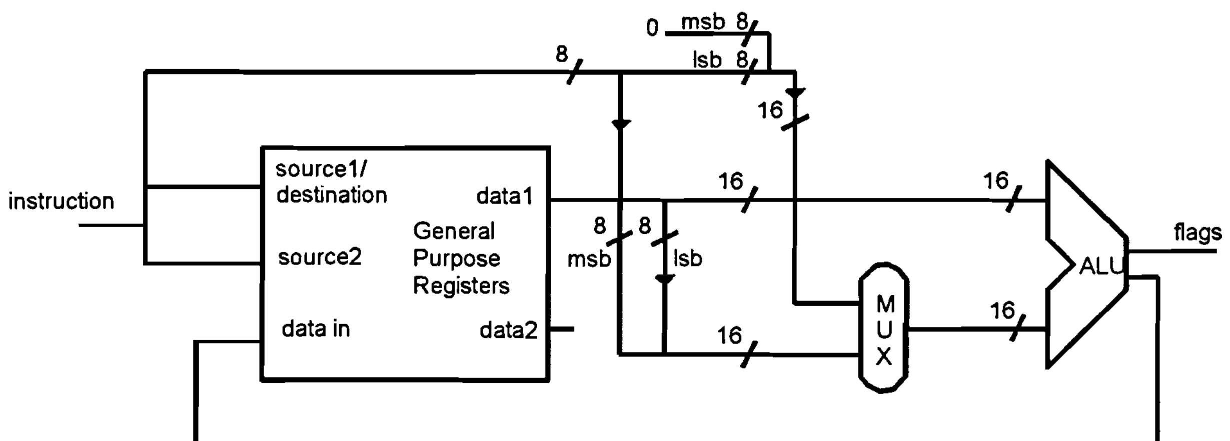


figure 3.6. The datapath for immediate addressing binary operations. Both mov and mhi operations pass through the lower input port of the ALU.

Another possible solution is to have the *mov* operation pass through the lower input port of the ALU and the *mhi* operation pass through the higher part of the ALU. The multiplexor will merely move to the other input of the ALU and the same amount of hardware is needed. This is shown in figure 3.7.
It is now quite clear that hardware can be saved. The lower 8 bits of the dataword for the upper part of the ALU must always pass unchanged. If the 16-bit dataword is split into two 8-bit parts then only the upper 8 bit need to be multiplexed.
The multiplexor now only needs to be half the size. This is shown in figure 3.8.

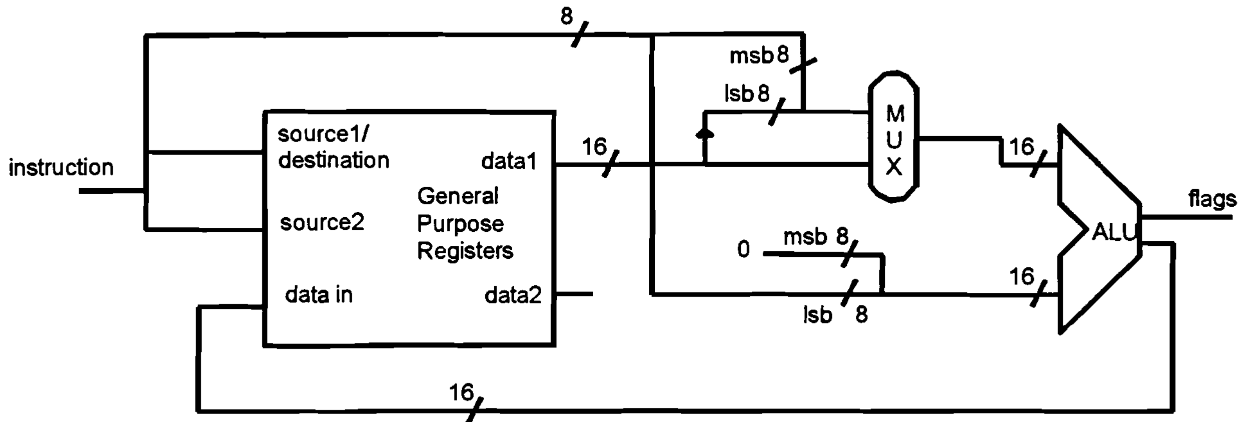


figure 3.7. The mhi operation now passes the data through the upper input port of the ALU.

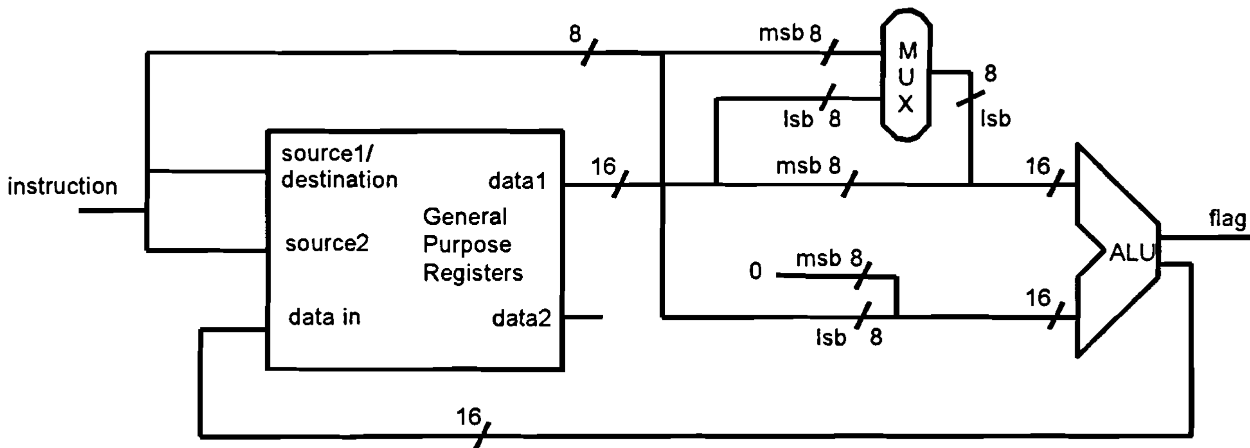


figure 3.8. The databus for the upper input port of the ALU can be split into two parts of 8 bit.

3.2.5 Putting the parts together for the binary operations.

All the parts of the datapath as we have seen so far can be combined into one schematic. The direct addressing mode and the indexed addressing mode both share the same adder to calculate the effective memory address. A multiplexor for both inputs of the adder will make the switching between the two modes possible.
Another multiplexor is needed to select the data for the lower ALU input. It must be possible to select between the immediate number from the immediate addressing mode, the output of the source 2 register in case of a register-to-register addressing mode and between the output of the Data Memory in case of a direct or indexed addressing mode. Each multiplexor will have its own control signals. They will not be shown here yet.
The combined datapaths are shown in figure 3.9.

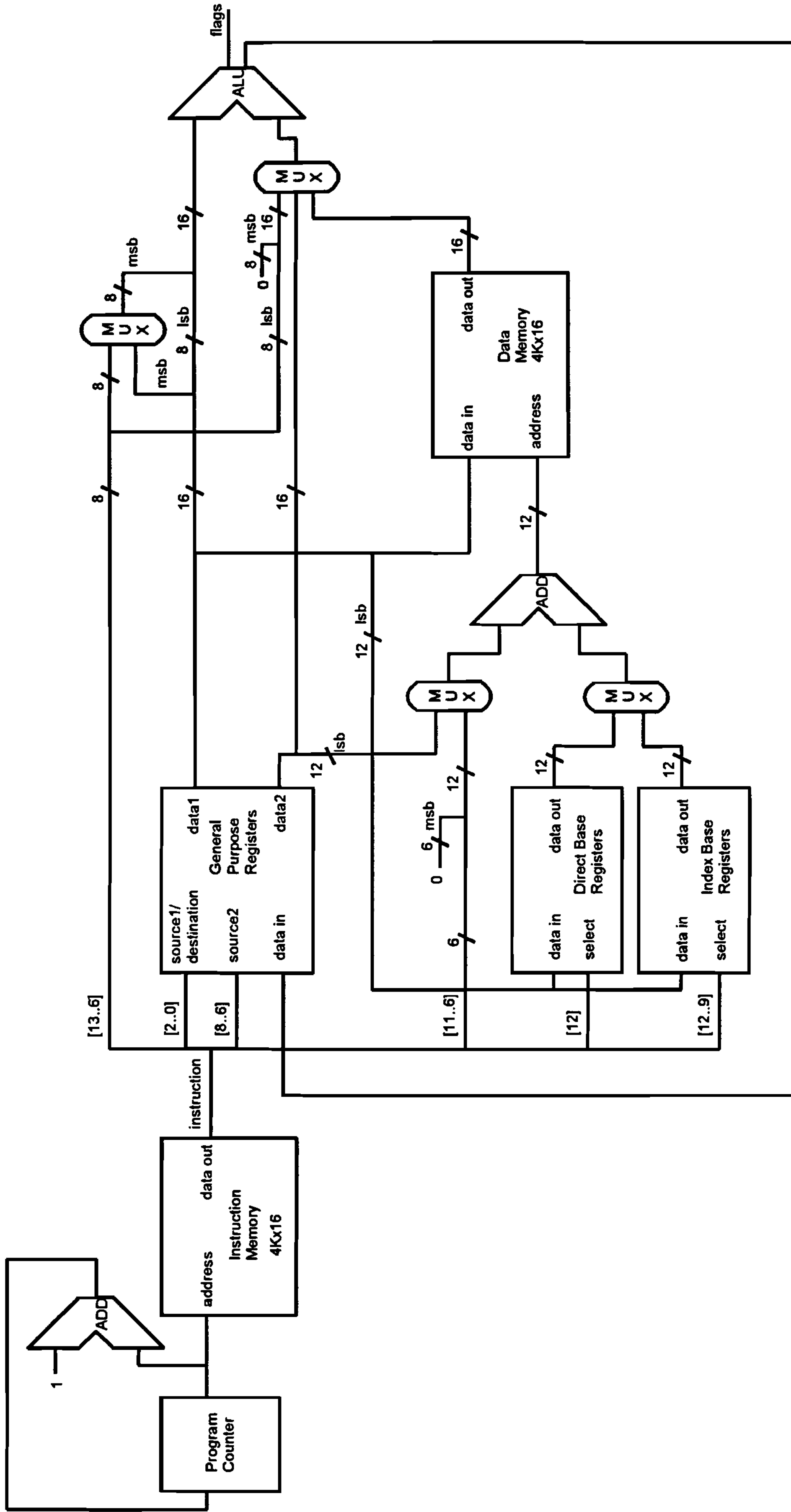


figure 3.9. Combining the parts of the datapath for the binary operations.

3.3 Adding the part of the datapath for the unary operations.

The Unary operations are *pop*, *push*, *cpl*, *set*, *shr*, *shl8*, *shr8*, *sel2* and *sel3*.
Except for the *pop* and *push* operations all unary operations can be performed by the ALU. So there must be a path from the General Purpose Registers through the ALU and back. As this is already implemented for the register-to-register binary operations, no further path modifications have to be made.
The *pop* and *push* operations need a stackpointer. A stackpointer needs to be loaded, incremented and decremented. The contents of the stackpointer addresses a memory location of the Data Memory. As the address adder already addresses the Data Memory another multiplexor is necessary here. This is shown in figure 3.10.

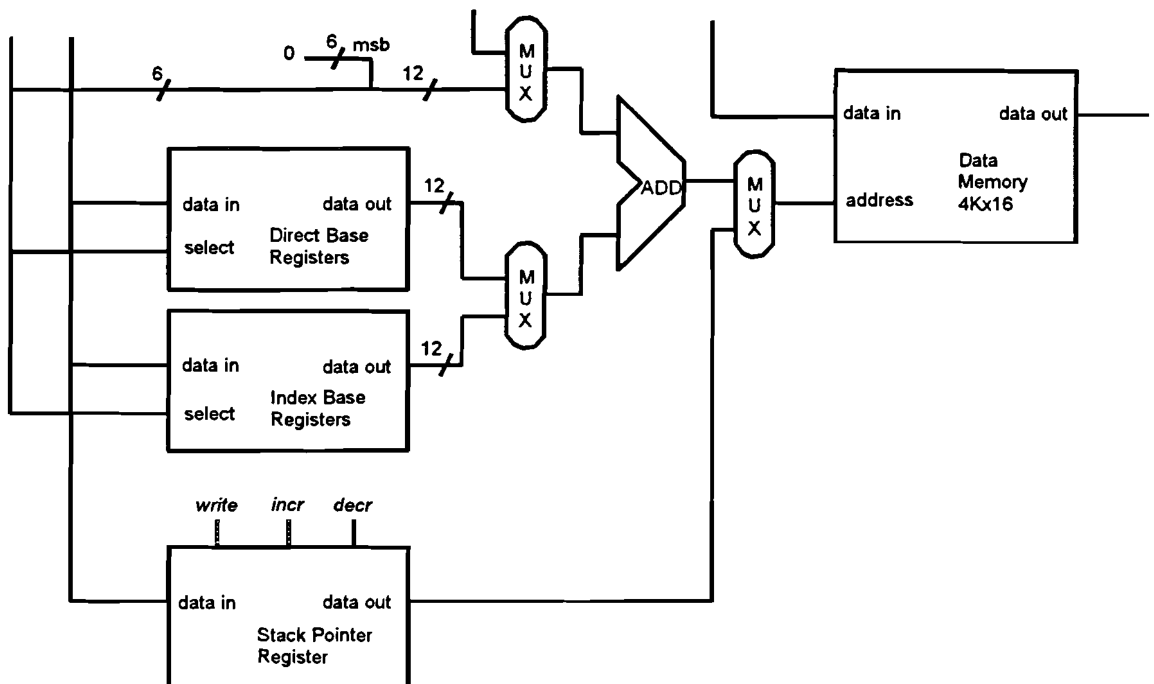


figure 3.10. Adding the Stack Pointer Register to the datapath.

In this configuration it is quite clear that when the Data Memory is addressed by the Stack Pointer Register, the adder is not used. It can be used to increment or decrement the contents of the Stack Pointer Register. This makes the Stack Pointer Register more simple and changes it into an ordinary register.
The stack pointer points to the top of stack which is an unused memory location. As the stack pointer points to the next free location, the address pointed to by the stack pointer must be used to address the Data Memory in case of a *push* instruction. At the same time the address can be decremented by the address adder and be directed back to the input of the Stack Pointer Register. At the next clock the decremented data will be clocked into the Stack Pointer Register which will now point again to the next free location.
In case of a *pop* instruction the contents plus 1 must be used to address the Data Memory. This value can be extracted from the address adder after it has incremented the contents of the Stack Pointer Register. This value will also be written back to the Stack Pointer Register.

Another multiplexor is needed to switch between data to be written into the Stack Pointer Register by a 'Mov to Stack Pointer Register' instruction and an updating of the register after a decrementation or incrementation.
The modification is shown in figure 3.11.

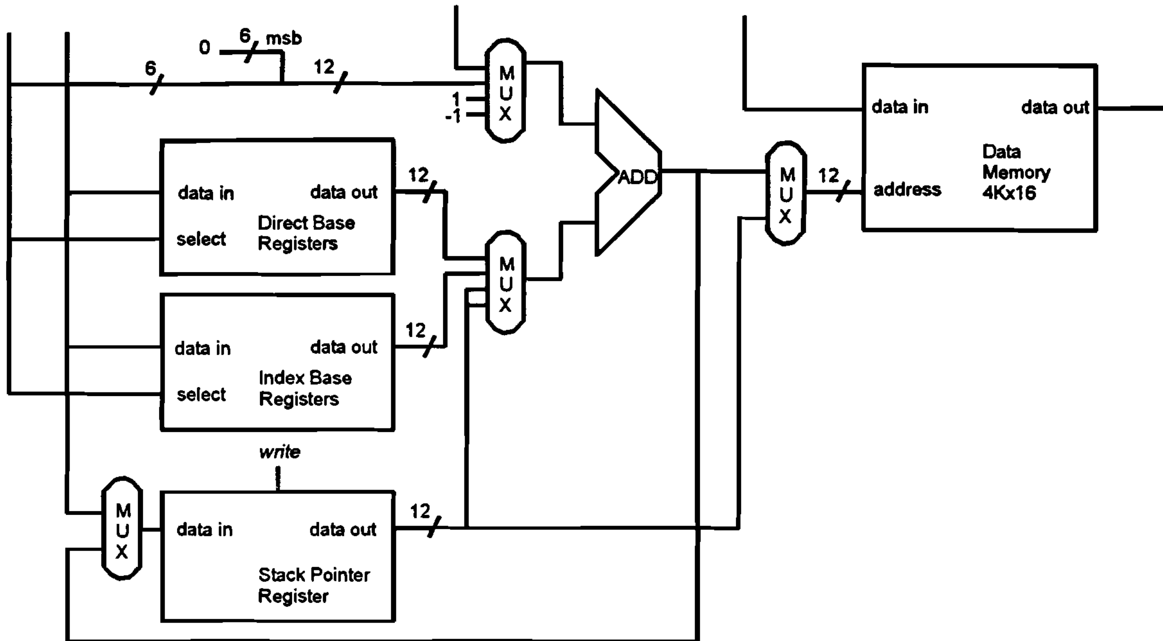


figure 11. Using the address adder to increment and decrement the Stack Pointer Register.

3.4 Adding the part of the datapath for all the jump operations.

Another set of important operations are the jump operations. The jump operations are *djnz*, *ja*, *jae*, *jb*, *jbe*, *je*, *jmp*, *jne*, *jno*, *jnz*, *jz* and *jsr*. Except for *jmp* and *jsr*, all jump operations are 8-bit signed displacements with respect to the current program counter and depend on flags set by an ALU operation. They are called the conditional jump operations or branch operations.
The flag register will contain the following flags:

flag bit:	
0	all zero
1	all set
2	smaller
3	greater

Combinational logic will decide whether to branch or not depending on the type of branch instruction.
An extra adder is needed to calculate the address to be jumped to. The displacement which is part of the instruction is an 8-bit two's complement signed binary number. For example the number -3 is represented by the 8-bit signed binary number 11111101. This way the calculation of the branch address can easily be done by adding the displacement to the current address.
As the current address is 12 bit wide the 8-bit signed number must be extended to 12 bit. This can be done by copying the most significant bit 4 times.

If for example the current program address is 3E6h and the displacement is 2Bh address places forwards (43 decimal) the new address will be:

displacement (8 bit):		0010	1011	2Bh	
displacement extended:	0000	0010	1011	2Bh	43
current program address:	0011	1110	0110	3E6h	998
	------------------------- +				------- +
summation:	0100	0001	0001	411h	1041

If the jump must be 2Bh address places backwards the displacement will be the two's complement of 2Bh:

the number 2Bh:	0010	1011
two's complement of 2Bh (-2Bh):	1101	0101

The new address will be:

displacement (8 bit):		1101	0101	- 2Bh	
displacement extended:	1111	1101	0101	- 2Bh	- 43
current program address:	0011	1110	0110	3E6h	998
	------------------------- +				------- +
summation:	0011	1011	1011	3BBh	955

As the displacement is only 8 bits wide the displacement can only reach from -128 to +127 places from the current program counter. The programmer should be aware of that limitation. If any branches must be taken to program locations further away the programmer must use the *jmp* instruction.

The jmp and jsr instructions hold a 12-bit address where to jump to and can be any ROM location. The extra modifications necessary for the *jsr* operation will be added later.

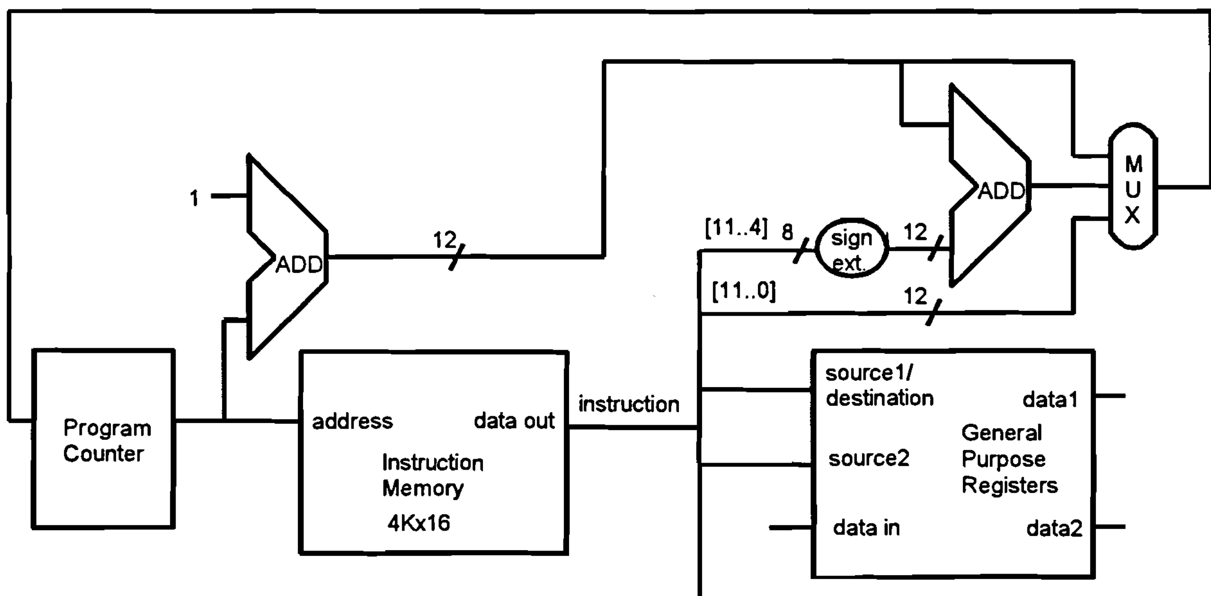


figure 3.12. The modifications for the jump operations.

To switch between the incremented next address, the branch address and the jump address another multiplexor is needed. Figure 3.12 shows the modification for the jump operations.

The *jsr* instruction (jump subroutine) is a bit more complicated. Together with the *rts* instruction (return from subroutine) it enables the programmer to use a specific part of the program more than once. In case of a jump to a subroutine the address following the *jsr* instruction must be saved on the stack which is part of the Data Memory.
To save this address on the stack a path must exist from the 'address incrementing adder' to the Data Memory. To be able to restore the address from the stack to the Program Counter Register a path must exist from the Data Memory to the Program Counter Register.
During a *jsr* instruction the contents of the Stack Pointer Register must be decremented by 1 just like with a *push* instruction. During an *rts* instruction the contents of the Stack Pointer Register must be incremented by 1 just like with a *pop* instruction.
The additions to the datapath for all the jump and branch instructions are shown in figure 3.13.

3.5 Constructing the part of the datapath for the other operations.

There are only two instructions left to be implemented. These instructions are *fill* and *rti* (return from interrupt).

3.5.1 Constructing a datapath for the *fill* instruction.

The fill instruction must fill a specified memory region with a value from one of the General Purpose Registers. This value is stored in memory at the address that is the sum of the contents of an Index Base Register and the contents of another General Purpose Register which functions as an index register. This register is decremented by one and the operation repeats itself until the index register decrements below zero. The program counter will not be incremented until the fill operation terminates.
There is already a path from the General Purpose Registers and the Index Base Registers for the data to and address for the Data Memory. What needs to be added is the decrementation of the index register.
During the fill operation the adder for calculating the jump address is not used. This adder can be used to decrement the contents of the index register. There must be two extra multiplexors added to the input of the adder to select between the address calculation and the decrementation of the index register. There must be an extra multiplexor added to the data input of the General Purpose Registers to select between the write back data from the decremented index register and the output data of the ALU. This is shown in figure 3.14. Mind that the adder is only 12 bits wide. As only the 12 least significant bits of the data from the index register are used for calculating the Data Memory address it is not necessary to increase the width of the adder.
Figure 3.15 shows the datapaths so far.

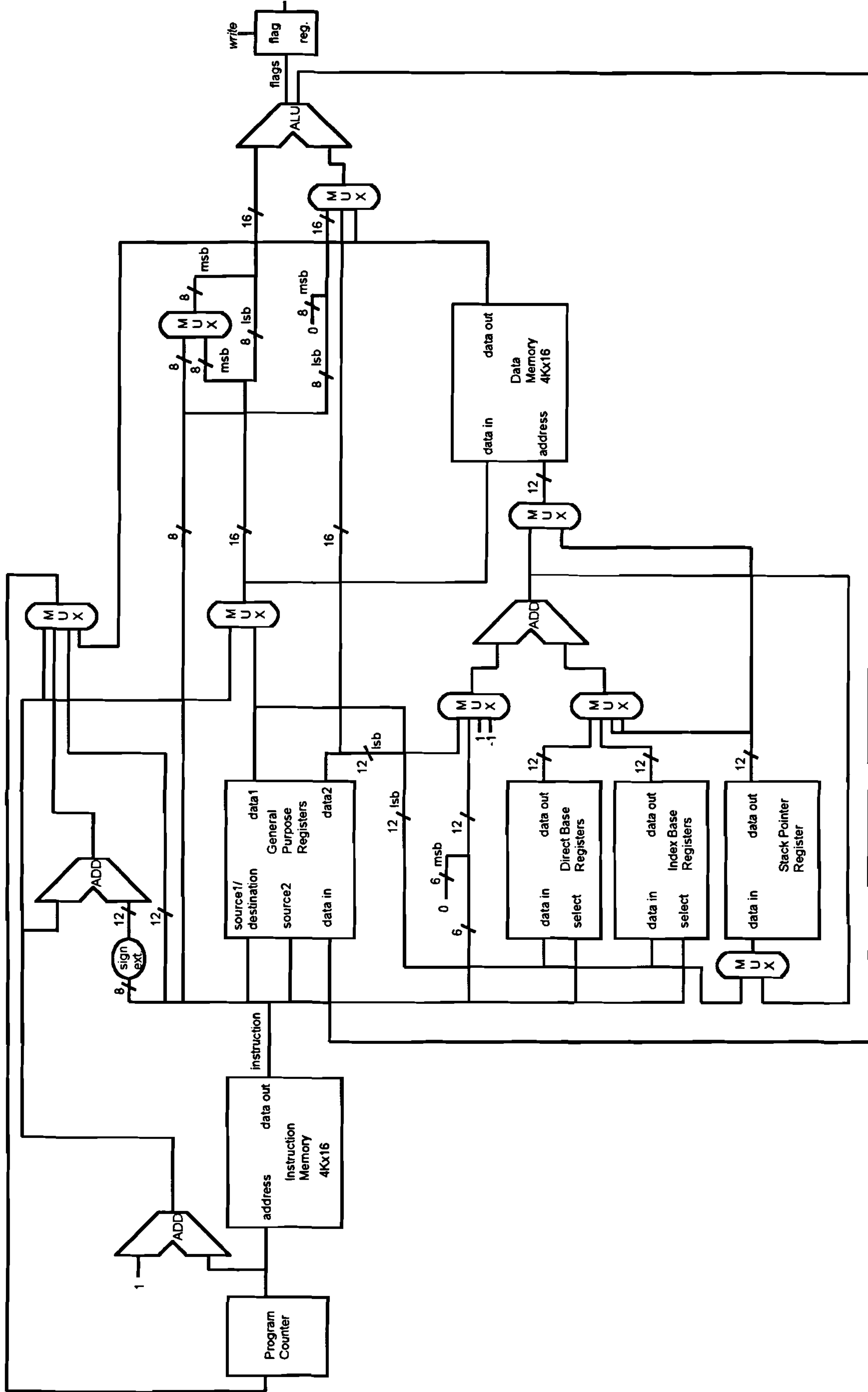


figure 3.13. Adding datapath parts for the jump and branch operations.

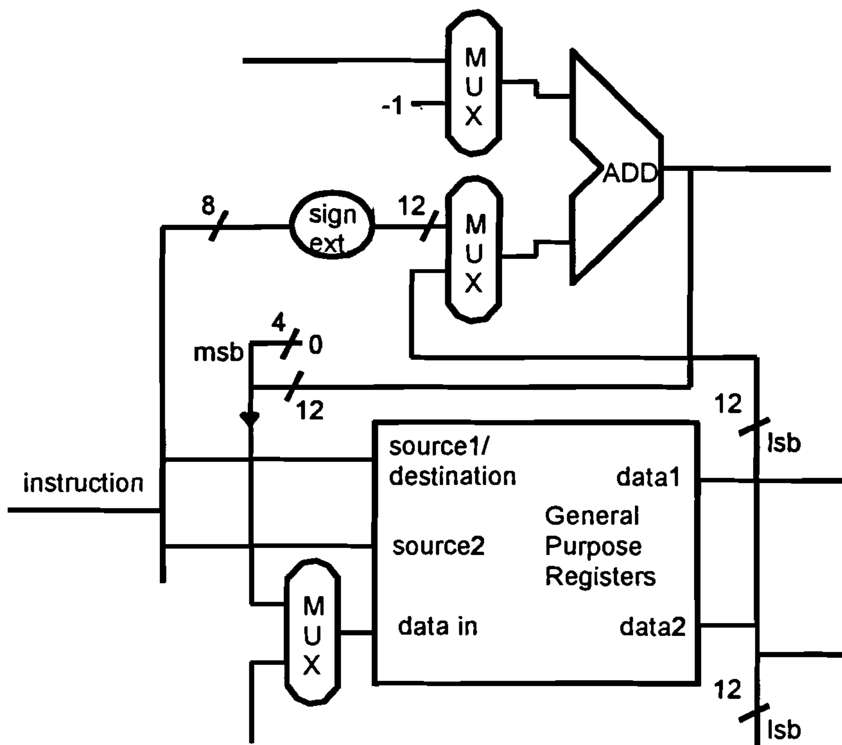


figure 3.14. Using the jump address adder for the index register decrementation.

3.5.2 Constructing a datapath for the rti instruction.

Most microprocessors have a control signal via which external logic can demand the microprocessor's attention. This processor must also be fitted with such a feature which is called an interrupt. If the interrupt signal is activated the processor must postpone its current program and start an interrupt routine. After it has finished this temporary activity it must continue where it left off. To remember where it left off the program counter must be saved. Next to saving the program counter the flags also need to be saved. When finishing the interrupt routine the *rti* (return from interrupt) instruction must restore the program counter and the status flags.
When the interrupt signal is activated the processor will have to finish the last instruction it was executing after which it will have to save the next instruction address and the flags on the stack. It will then load the Program Counter Register with value 1 to proceed with the interrupt routine. To save the Program Counter Register contents there must be a path from this register to the Data Memory. As program addresses are 12 bit and there are 4 flag bits, they can be combined and saved together on the stack. So there must also be a path from the flag register back to the Data Memory. Both paths need to be build with multiplexors.
When the interrupt routine has finished the return address and flags must be restored from the stack. So there must also be a path from the Data Memory to the Program Counter Register. But that is no problem as it has already been implemented for the *rts* instruction. There must also be a path from the Data Memory to the flag register as the flags have to be restored as well. Another multiplexor can be used to implement that path.
These adaptations are shown in figure 3.16.

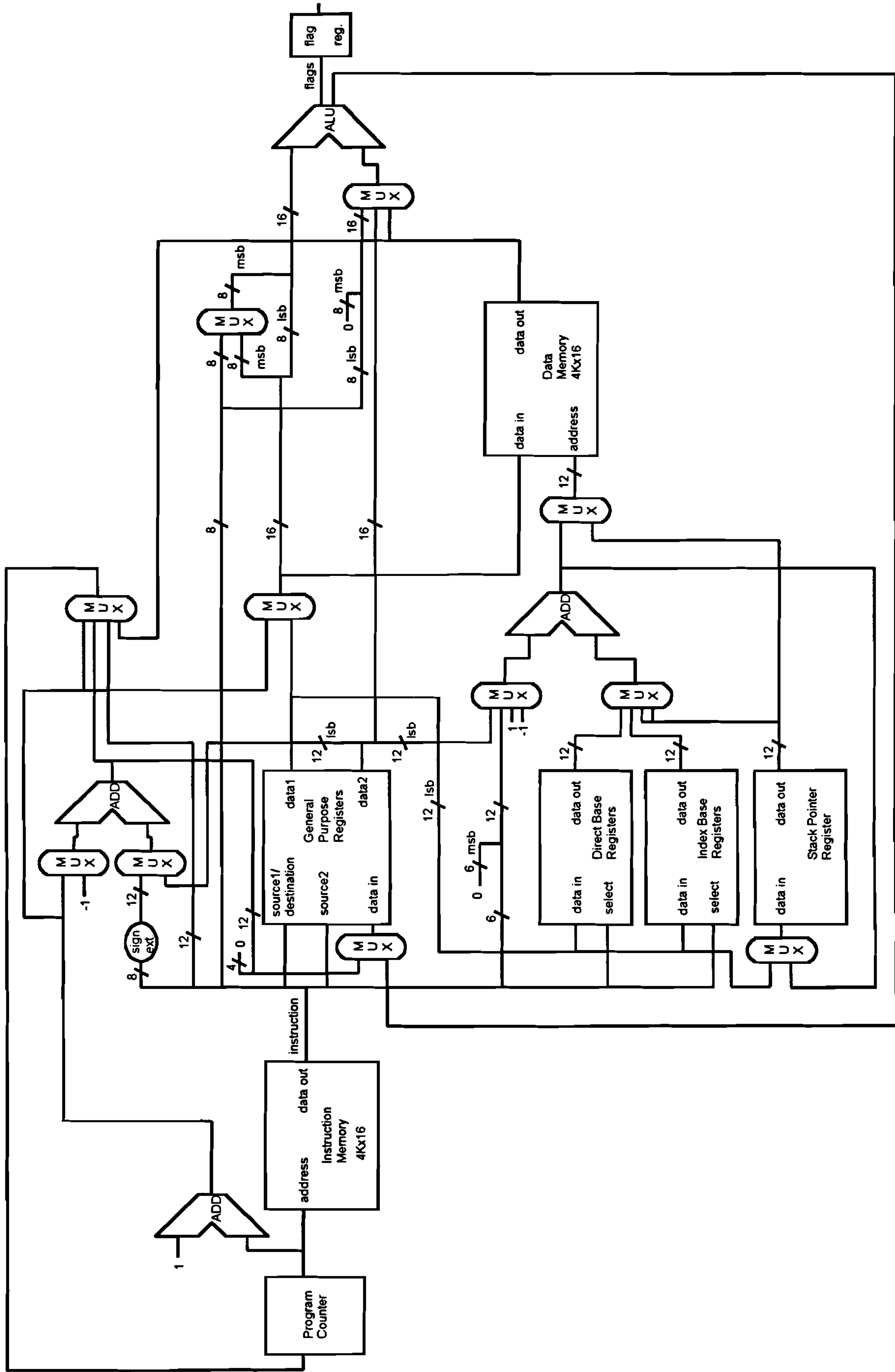


figure 3.15. Using the branch address adder also for the fill operation.

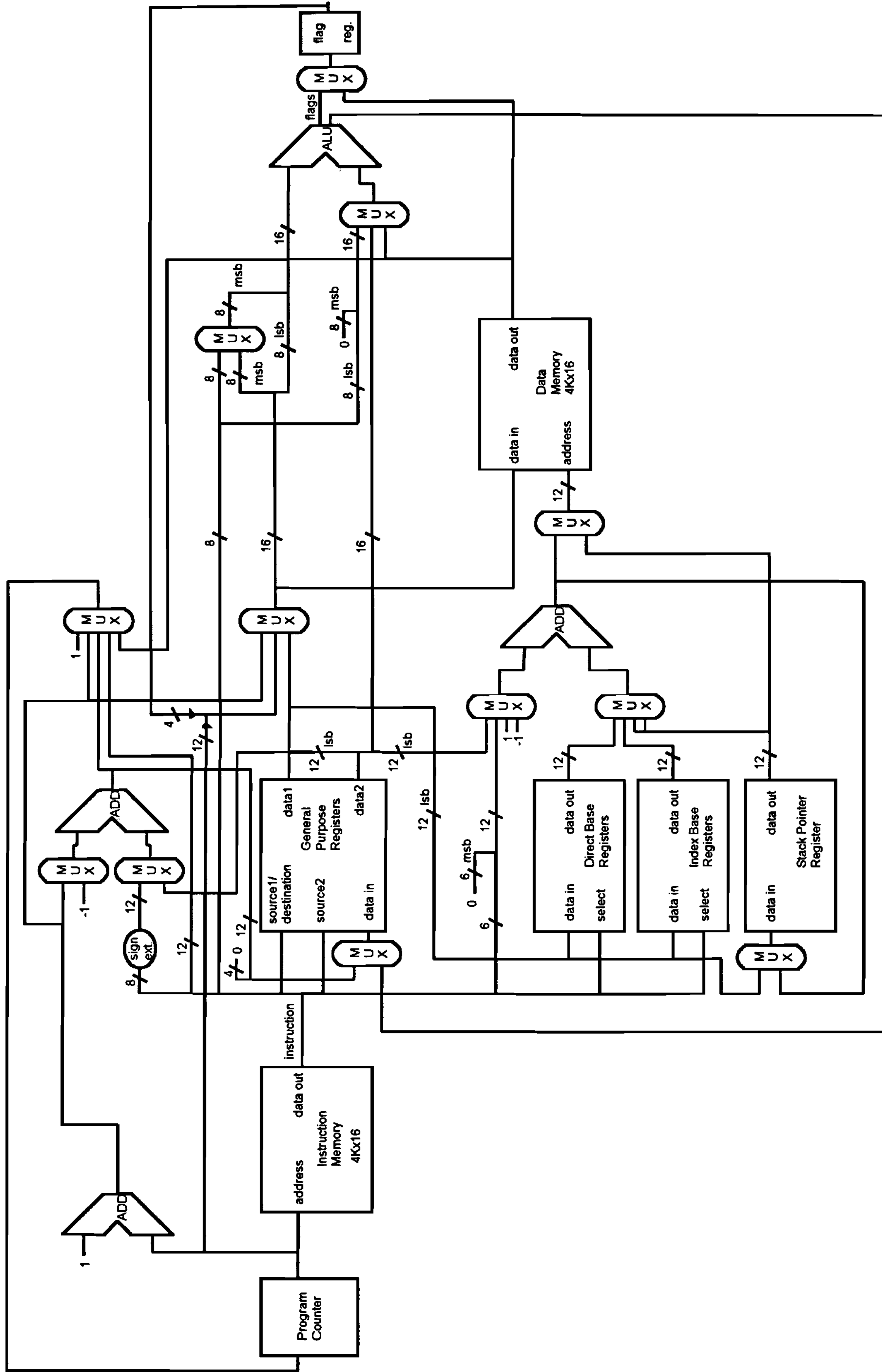


figure 3.16. Adding the path for the interrupt feature.

4 Adding pipelines to the design.

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is the key to making processors fast.
A pipeline is like an assembly line: in both, one step completes one piece of the whole job. On a well balanced assembly line a product exits the line in the time it takes to perform one of the many steps. Note that the assembly line does not reduce the *time* it takes to complete an individual product; it increases the number of products being built simultaneously and thus the *rate* at which products are started and completed.

As in an assembly line, the work to be done in a pipeline for an instruction is broken into small pieces, each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is called a pipe stage or a pipe segment. The stages are juxtaposed to form a pipe: instructions enter at one end, are processed through the stages, and exit at the other end. Once again, pipelining does not reduce the time it takes to complete an individual instruction. It increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.

The time required to move an instruction one step down the pipeline is ideally one clock cycle. The length of a clock cycle is determined by the time required for the slowest pipe stage, because all stages must proceed at the same rate. It is important to balance the length of each stage. Otherwise, there will be idle time during a stage.

4.1 Chopping the datapath into pipe stages.

In a pipeline each pipeline stage is separated by a pipeline register. All clocked elements of the datapath can equally be classified as pipeline registers.
Of course the program counter is a clocked register. The register for the flags is also a clocked register. The General Purpose Register, Direct Base Register, Index Base Register and Stack Pointer Register can be asynchronously read and will not be regarded as pipeline registers. Writing to these registers on the other hand, will be synchronous which means that writing to these registers will be clocked.
The Instruction Memory (ROM) and the Data Memory (RAM) will also be clocked modules and will form a part of a pipeline register.

Next we want to know what are the slowest parts of the design. The ALU is such a 'slow' part. So a pipeline stage needs to be right in front and right behind the ALU. Other 'slow' parts are the memory modules in spite of the fact that they are clocked. So they also need to have a pipeline stage in front and behind them.

Information which will be necessary in a further stage of the pipeline must be there together with the rest of the instruction. So next to shifting of the instruction through the pipeline all other necessary information must be shifted as well.
The ALU is nearly at the end of the pipeline. In case of a jump operation the result of the ALU flags determine whether there will be a jump in the program or not. So the jump address must be shifted through the whole pipeline together with the instruction.

In the design without the pipelines, the instruction holds the address of one of the General Purpose Registers. This register functions as a source and as the destination where the ALU result must be written back to. When pipelines are implemented the selected write-back register address must be stored until needed. So this information must be piped as well. This brings another change into the design. When data must be written back to one of the General Purpose Registers this stage of the pipeline is already processing another instruction and other register addresses may be selected. So the General Purpose Register need to have separate inputs for selecting the registers for reading and for writing. Therefore the source/destination input needs to be split into a separate source input and a separate destination input. Figure 4.1 shows the design with the pipeline registers added. There are six pipeline stages.

Each pipeline register holds several parts of the datapath. The width of each pipeline register depends of the number of lines of each datapath part. In some of the pipeline stages this width can be decreased by moving some parts of the datapath. The multiplexors, which are situated just before the ALU, can be moved to the stage where the General Purpose Registers are situated.
As the ALU can only write to one of the General Purpose Registers, which behave synchronously for writing, it is possible to skip the last pipeline stage for the ALU output and the destination address.
These modifications lead to the design of figure 4.2.

Because this microprocessor is an embedded processor it will run just one specific program which will be placed in the ROM. As this program is already known, it is clear that it will contain a lot of loops. This means that the program will be using a lot of conditional jump operations. In the design layout of this processor so far, all conditional branch instructions must ripple through all pipeline stages. At the last stage of the pipeline the decision is made whether the branch has to be taken or not. All instructions which come directly behind the conditional branch instruction and are already in the pipeline, must be flushed in case of a branch. This means a lot of waste of time. For that reason the decision was made to change the concept of the processor and give it a load-store architecture. This means that there are no other operations than move operations to and from the Data Memory. The output of the Data Memory must first be written to one of the General Purpose Registers before it can go to the ALU for any arithmetic processing.

With this modification two pipeline stages become obsolete but the pipeline stage behind the ALU will be added again. This is necessary to prevent instructions passing each other. The output of the Data Memory was first going through the ALU and from there on back to the General Purpose Registers. As the memory data is no longer going through the ALU there is now a need for an extra multiplexor to switch between data from the ALU and the Data Memory. As the ALU and the Data Memory can never be used at the same time, and as they are in the same pipeline stage, they can not be writing back to the General Purpose Registers at the same time.
No other major changes have to be made. The design changes to that of figure 4.3.
Each stage can be given a name. The stage between the Program Counter Register and the ROM will be called the Pre-fetch stage. The stage right behind the ROM will be called the Fetch stage. The next stage will be called the Decode stage as all instruction decoding will be done here. The stage containing the ALU will be called the Execute stage. The last stage will be called the Write-back stage.

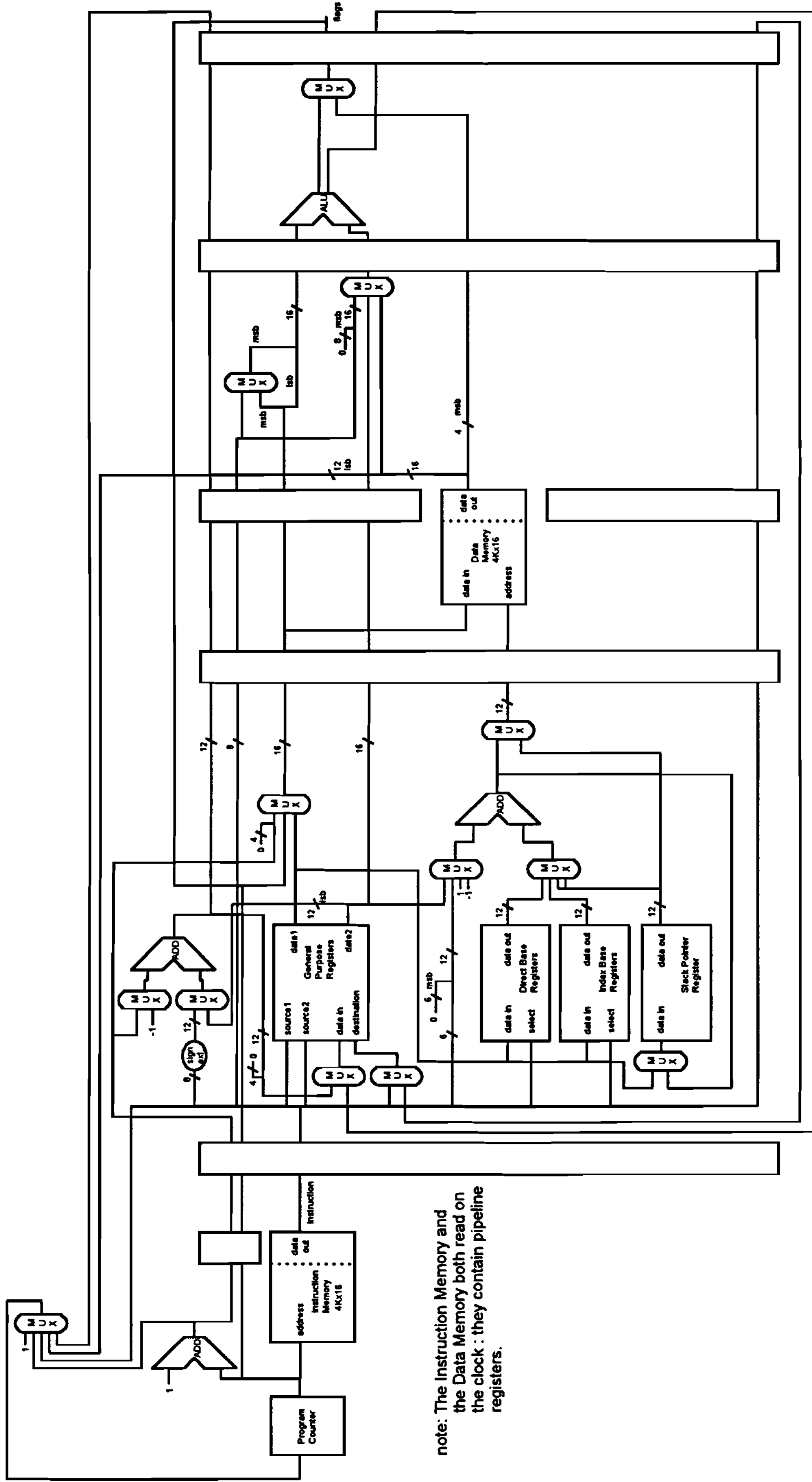


figure 4.1. Pipelines added to the design.

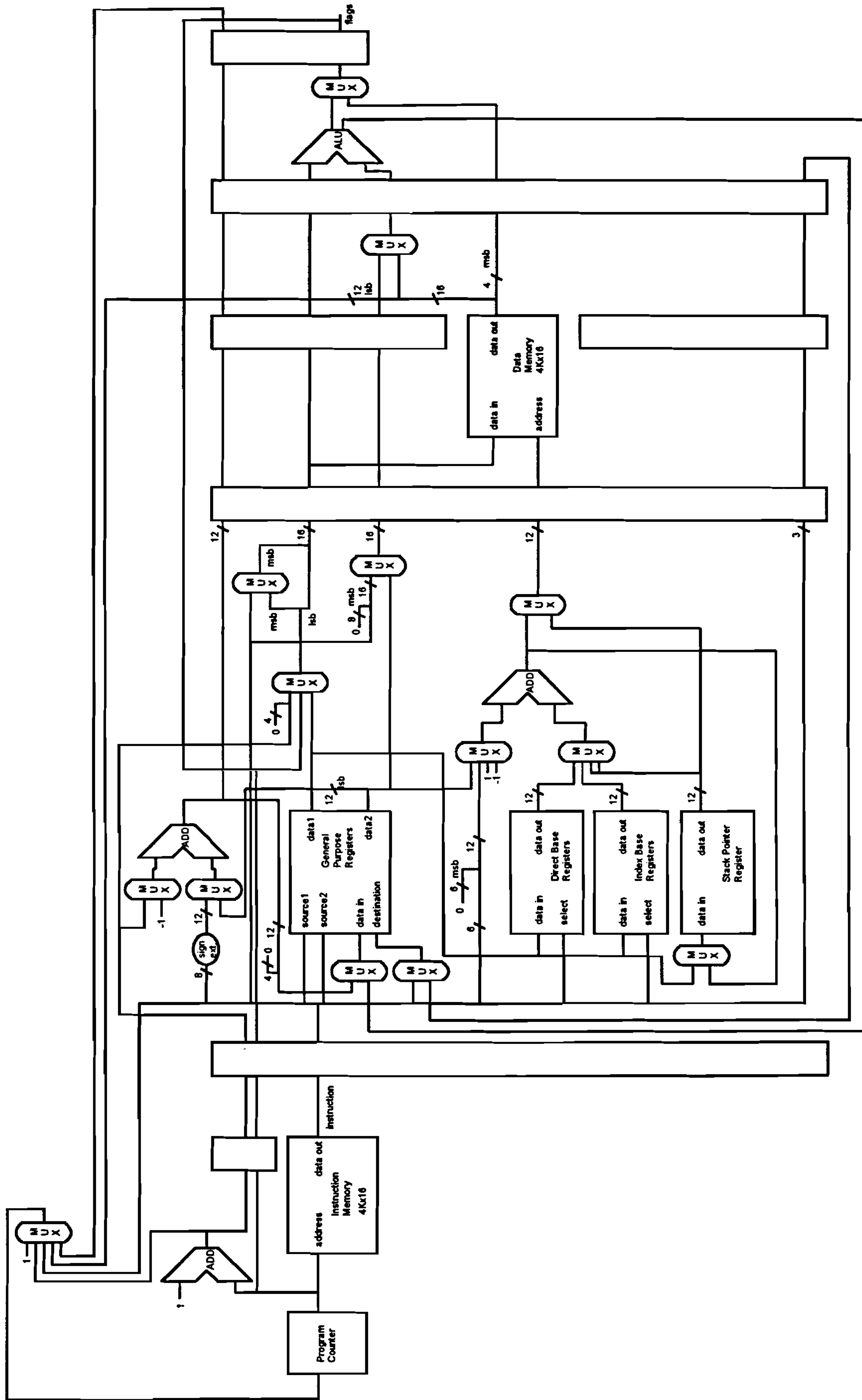


figure 4.2. Moving the multiplexors more to the front and removing the pipeline stage behind the ALU.

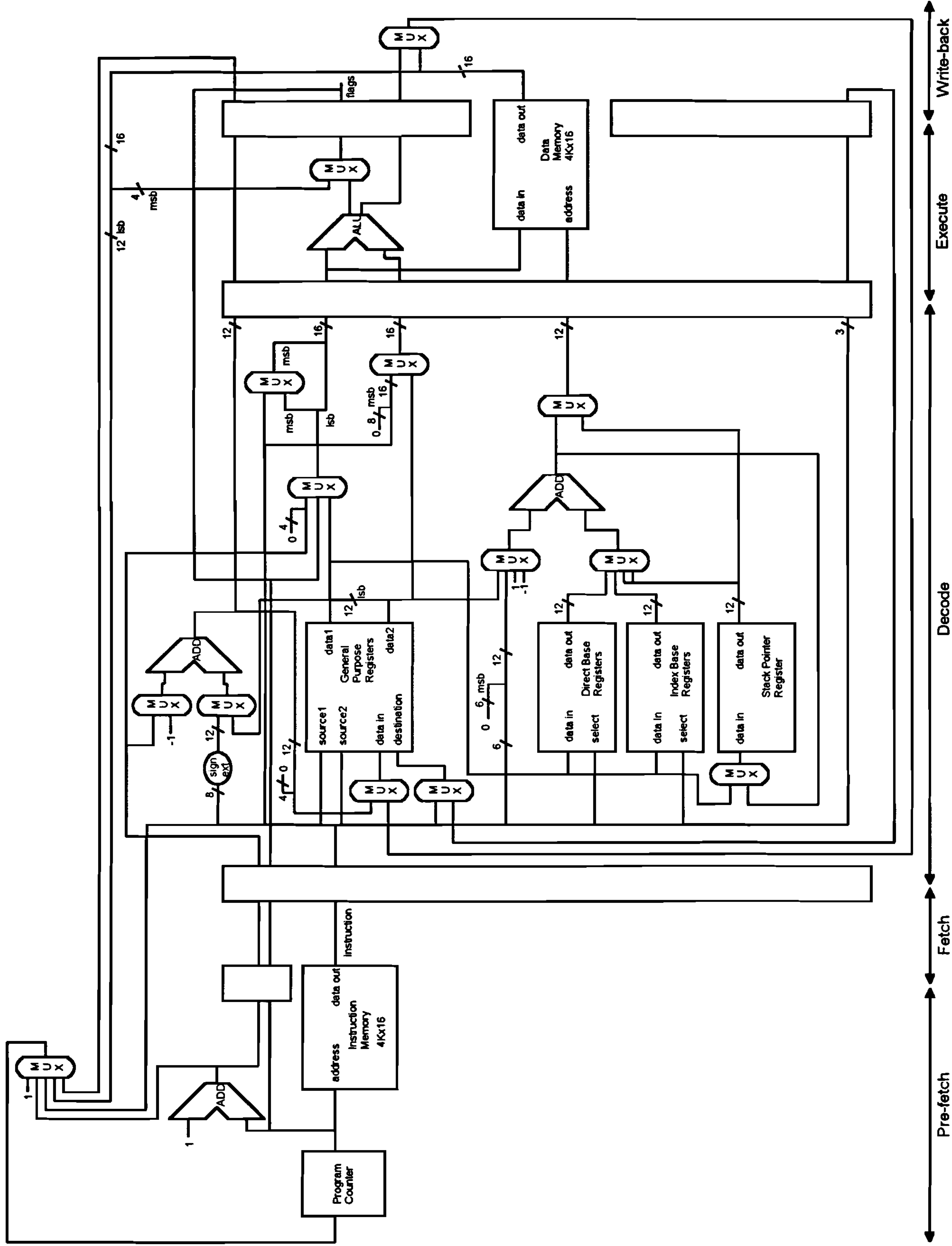


figure 4.3. Changing the design into a load-store machine.

In the design layout before we added the pipeline registers there was a flag register with its own control signal to keep the flags in case they shouldn't change. As this flag register is now part of a pipeline register it will always be written to on each clock. To freeze the flags on instructions that do not allow the flags to change, there must be a path from the output of the flag register back to the input of the flag register.

Before we switched to a load-store architecture all data that could change the flags was going through the ALU. The ALU could then create all 4 flags. But now that the data from the Data Memory is not passing the ALU any more a slight change has to be made.
The generating of the 'below' and 'above' flags must still be done by the ALU. But the 'all set' and 'all zero' flags must be generated from the data coming from the ALU output **or** from the memory output, depending on the type of instruction.
Therefore the decision is made to move the flag generating to the Write-back stage. This flag generator will also restore the flags in case of a 'return from interrupt'.
Because the flags are still depending on the Data Memory output the new flags are only known at the end of the Write-back stage.
Figure 4.4 shows the changes in the flag generating.

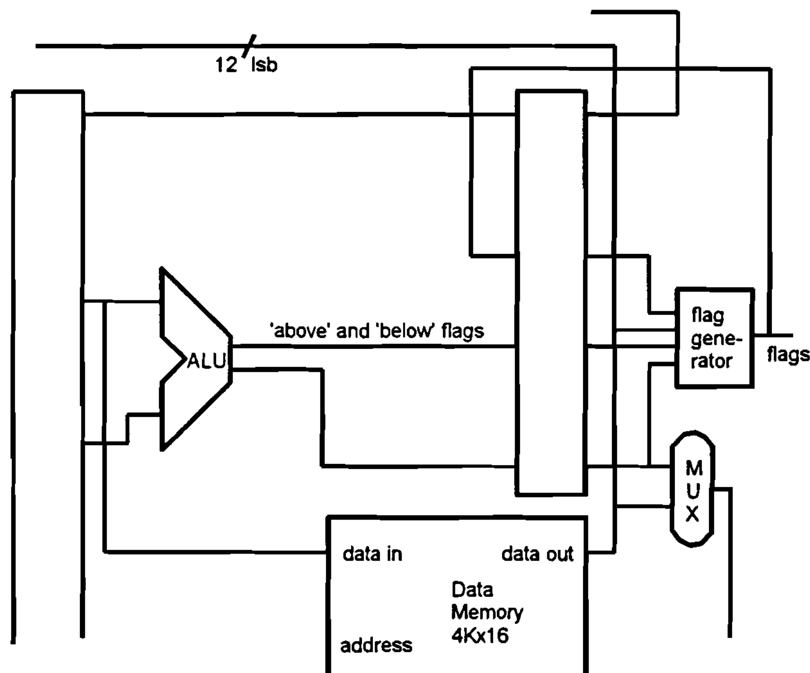


figure 4.4. To assure proper functioning of the flags a feed-back line is needed.

Another problem that occurs is that in case of an interrupt the flags are copied to the stack while there might still be instructions further on in the pipeline that can change the flags. So the copying of the flags must be postponed until the Execute stage. This is yet not a good solution as this will create a critical path. As the flag values can also depend on the output of the Data Memory the flags will only be stable at the end of the clock cycle. But when those flags will also serve as inputs for the Data Memory the setup time will be not be made. So the flags which can be saved must first go through a register. That makes it exactly the output of the feed-back line implemented in figure 4.4. In that case no instruction that can alter the flags may precede the *int* instruction. This can be assured by preceding the int instruction by a *nop* instruction. This change is shown in figure 4.5.

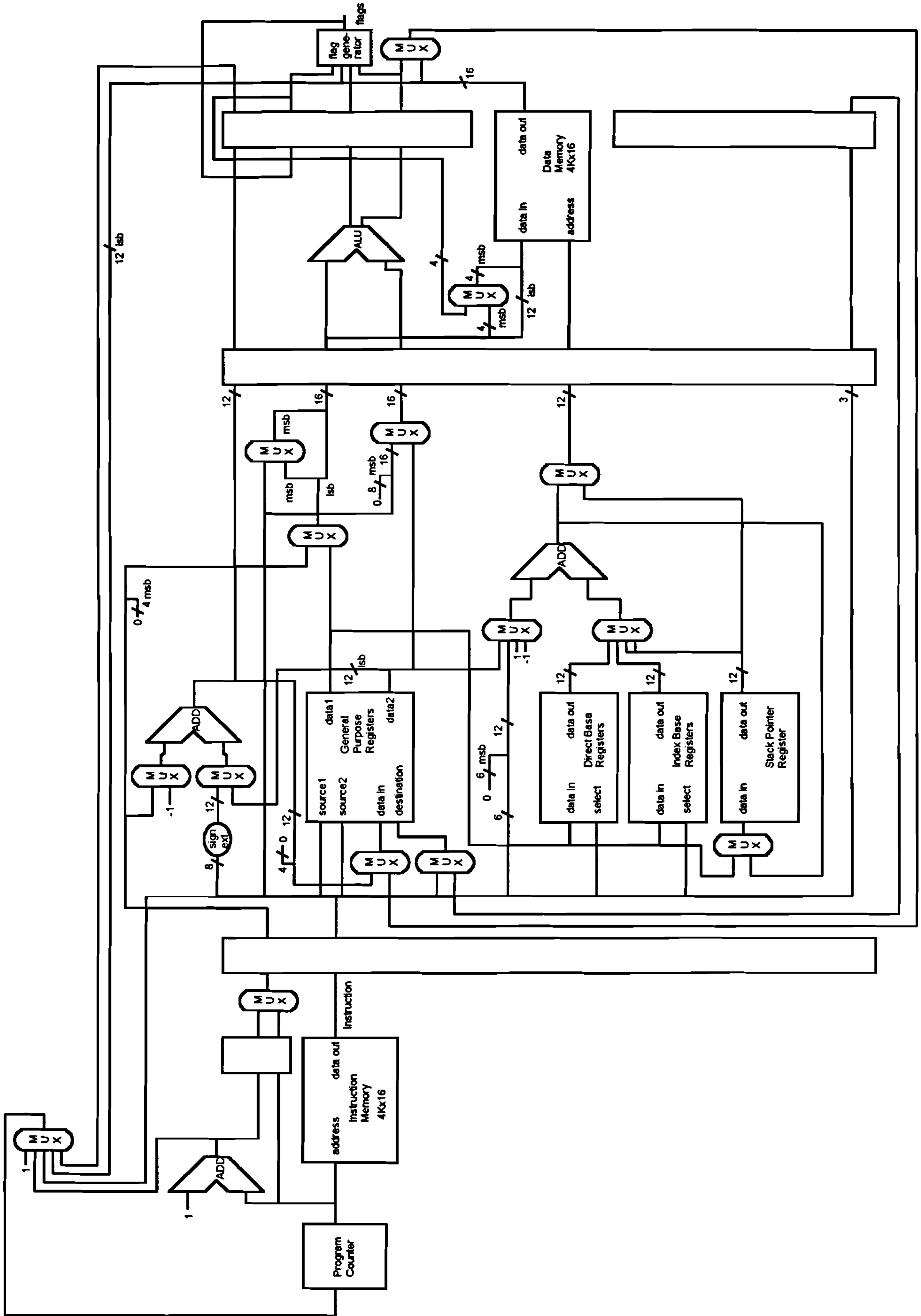


figure 4.5. *The copying of the flags to the stack must be postponed until the Execute stage.*

The path from the fetch stage to the Decode stage to pass the 'next instruction address', in case of an jsr instruction, can also be used for passing the address of the next instruction in case of an interrupt. This change is also shown in figure 4.5.

4.2 Data Hazards.

Pipelining is an implementation technique in which multiple instructions are overlapping in their execution. But what happens if the instructions in the pipe have dependencies. If the result of one instruction is going to be used by the following instruction, it has to wait for its execution until that result is available. Such dependencies are called data hazards. To resolve the problem of data hazards it is important to know where the dependencies can occur.

The most common data dependency is that of an instruction that must write its ALU-result back to one of the General Purpose Registers and the following instruction(s) using that result. For example the loading of a Direct Base Register with an immediate value.
First, one of the General Purpose Registers needs to be loaded with the lower 8 bits of the data word by means of an immediate move. The execution of moving the higher 8 bits needs to wait until the result of the first immediate move is written back into the General Purpose Register. The execution of moving the result of the whole 16 bit word to the selected Direct Base Register must be stalled until the result of the second immediate move has been written back to the General Purpose Register.

Another type of data dependency is that of an instruction needing data from the Data Memory. It has to wait until the data from the Data Memory has been written into one of the General Purpose Registers. In these cases forwarding may resolve some stalling.

Forwarding can best be done by adding two multiplexors at the inputs of the ALU. Data from the Write-back stage can then be directed back as inputs for the ALU.
As the General Purpose Registers act as a pipeline stage when they are written to, it can be convenient to forward the data input of the General Purpose Registers to the outputs if they have the same register number. An exception must be made here in case of the fill operation where this forwarding is an unwanted situation.

A disadvantage of this solution is that the critical path becomes too long. As the output from the Data memory is very slow it can't be directly routed back to the input of the ALU which is also a time consuming part. So the data from the Data Memory can only be forwarded to the decode-stage of the pipeline. When data is fetched from the Data Memory and the following instruction is an operation on that data stalling one clock cycle is inevitable. Forwarding the data from the ALU output can still be done to both decode and execute stages. This is shown in figure 4.6.

In case of a fill instruction the pipeline has to be stalled as long as there are write-back operations further on in the pipeline, because it wants to write to the General Purpose Registers itself. Forwarding can not solve this problem.

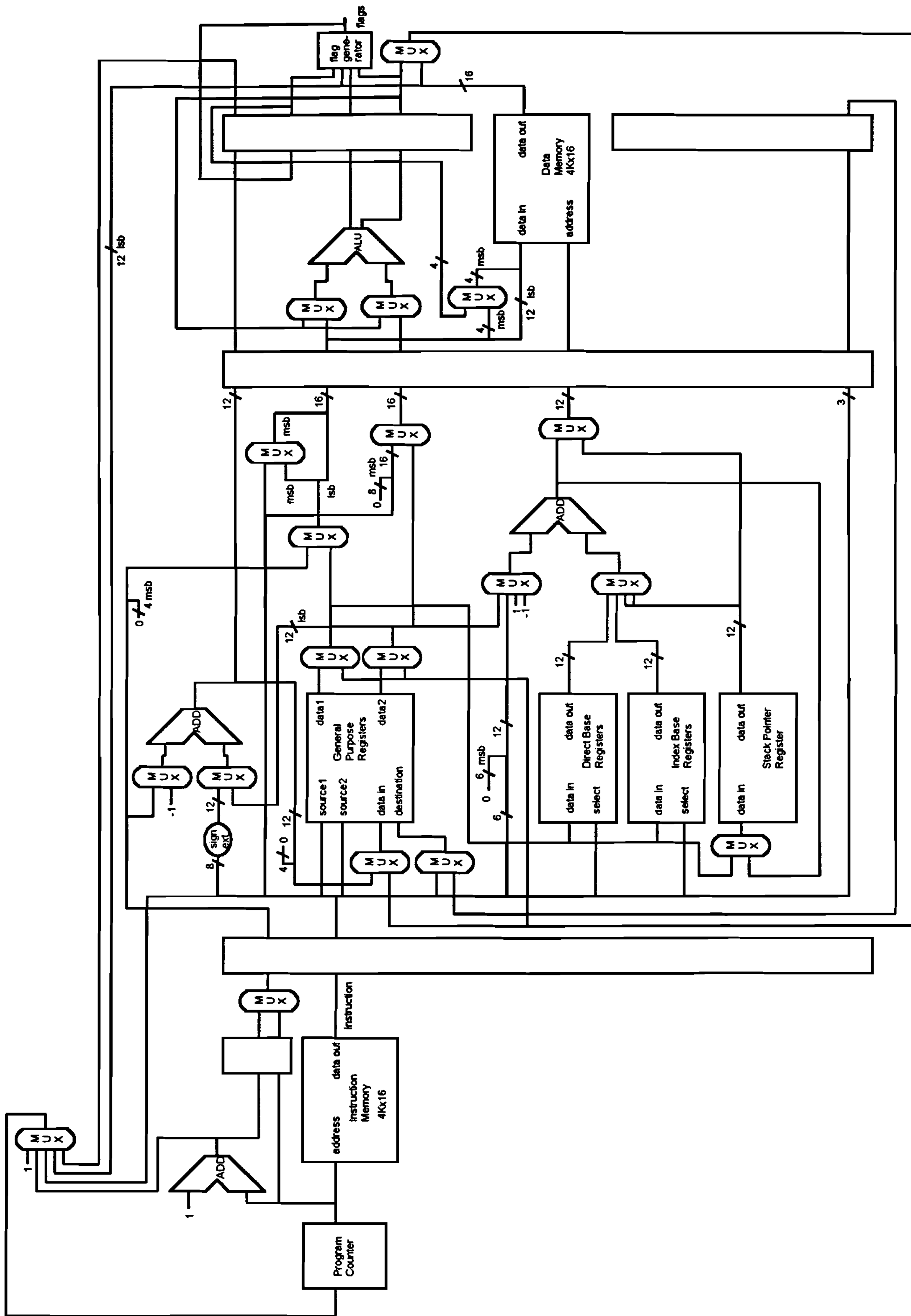


figure 4.6. Most data hazards can be avoided by forwarding data.

In case the Execute stage contains an instruction which alters the data for one of the General Purpose Registers which is also used in the Decode stage for an indexed operation, the pipeline has to be stalled until that data is available.

Now that the forwarding paths are implemented let's take a closer look at the loading of an immediate value in one of the General Purpose Registers. If a 16-bit value must be loaded into one of the registers it must be done by two immediate move operations. In this example we will only concentrate on the execution of these two instructions.
While the first instruction is decoded in the Decode stage the next one is fetched from the Read Only Memory. The Decode stage will prepare the data for the lower path of the ALU. This data will contain the 8-bit immediate value as the lower byte and 8 zeroes for the higher byte. On the clock this data will be shifted to the Execute stage. In this stage it will only pass the ALU without any arithmetic operations and is ready to be shifted to the Write-back stage. But at the same time the next immediate move operation is a *mhi* instruction. This instruction needs the lower 8 bits of the dataword which is still in the Execute stage. As there is no path back from that stage to the Decode stage a stall will be necessary. When the data from the first immediate move operation is shifted from the Execute stage to the Write-back stage the stall will be cancelled and the second move operation can proceed.
This stall can be prevented by a simple change of the datapath. In figure 4.2 we moved the multiplexors which were just in front of the ALU more to the front to save width of the pipeline registers. If we shift the two multiplexors that are responsible for the immediate move operations back to the Execute stage the stall will be prevented. This is shown in figure 4.7.
Now let us take a look again at the loading of a 16-bit immediate value in one of the General Purpose Registers. The first move operation will do nothing in the Decode stage. The 8-bit immediate value will be merely passed to the Execute stage. When it arrives in that state the 8-bit immediate value will be provided by 8 zeroes and on the clock shifted to the Write-back stage. In the meanwhile the second move operation was decoded in the Decode stage. As this stage doesn't do any data manipulations but only passes the immediate value to the Execute stage, no stall will be necessary. When the second move operation arrives at the Execute stage, the result of the first move operation is needed. As that data is now present in the Write-back stage it can be forwarded to the Execute stage. The complete 16-bit immediate value can now be formed and on the next clock be shifted to the Write-back stage.

4.3 Branch Hazards

Next to the data hazards are the branch hazards. An instruction must be fetched every clock cycle to sustain the pipeline, yet in this design the decision about whether to branch doesn't occur until the Write-back pipeline stage. One solution is to stall until the decision whether to branch or not is taken. The drawback is that many times a conditional branch will decide against branching, and the work that would have been accomplished fetching and decoding the following instructions is exactly what will need to happen anyway.
A common improvement over stalling when a branch instruction is in the pipeline is to assume that the branch will not be taken and so will continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.

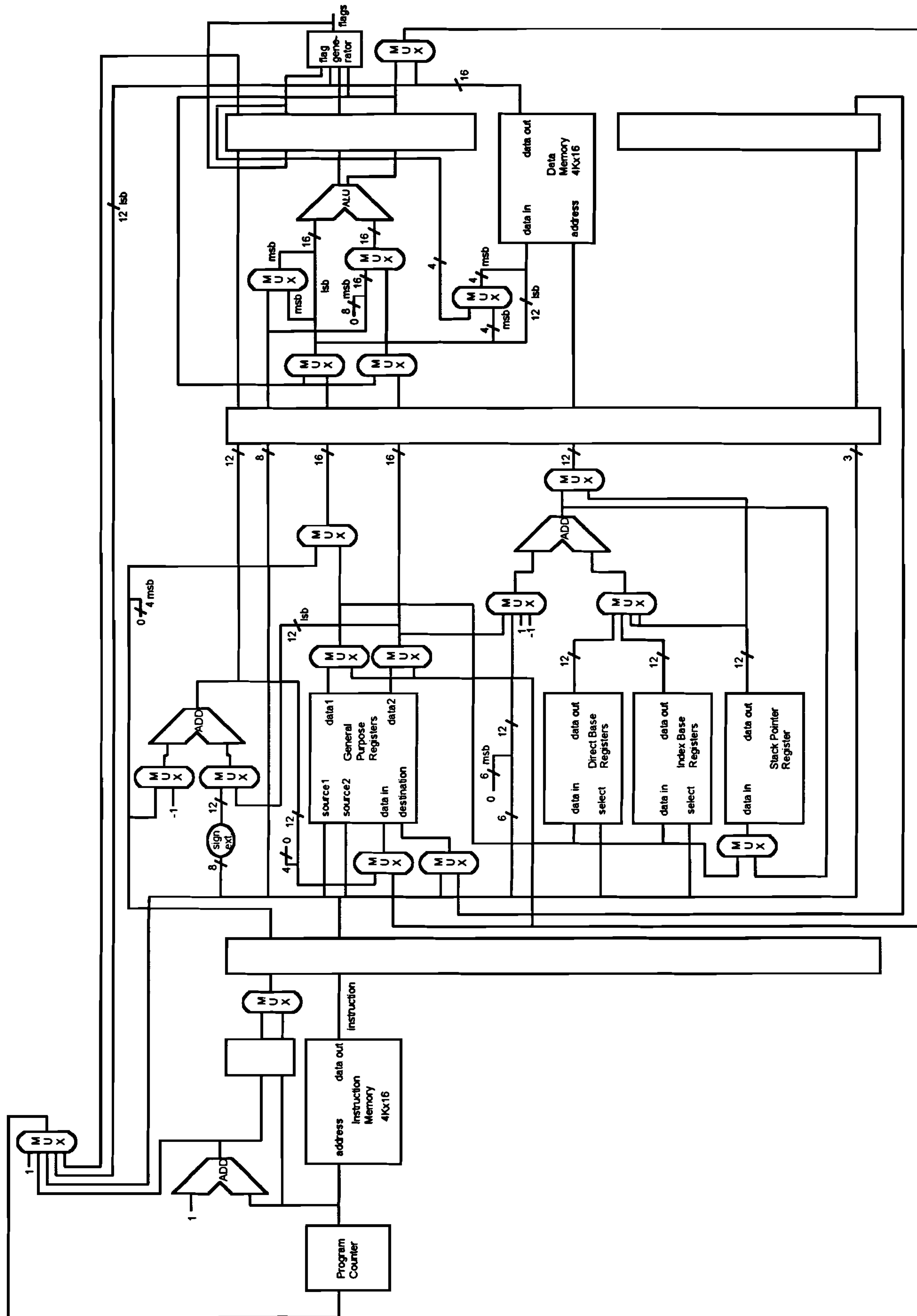


figure 4.7. Preventing a stall when loading a 16-bit immediate value.

To discard instructions in the pipeline the pipeline registers need extra control signals to flush their contents.
Just flushing the pipeline is not always the correct solution. All write operations to the registers and the Data Memory by instructions following the branch instruction must be stalled until the branch decision is clear. It is not possible to undo these write operations. Instructions like *jsr*, *rts*, *rti*, *pop* and *push* alter the contents of the Stack Pointer Register. The *fill* instruction alters the contents of one of the General Purpose Registers and moving data from a General Purpose Register to one of the Direct Base Registers, Index Base Registers or Stack Pointer Register is also done in the decode stage.
Except for the complexity of the control logic no other architectural changes have to be made.

Stalling the pipeline is assumed when an instruction is decoded in the Decode stage and not all the data for the execution is available, or the instruction wants to alter register contents which it can't undo in case of a branch. The instruction will be kept in the Decode stage until no further stalling is necessary. In case of a stall everything in front of the Execute stage will stall its operations. This means that no new instruction will be fetched and the program counter will not be incremented.

A small problem occurs with a stall as the Read Only Memory is clocked and its output can not be 'held'. When no stall occurs, the Read Only Memory will generate a new instruction while the Program Counter Register generates the next address for the following instruction. On a stall the Program Counter Register will hold its current value. But the Read Only Memory can not hold its current output and a new input address is already generated. The solution is to either save the former input address of the memory or save the output of the memory. This can be done with an extra register and a multiplexor to switch between the old and the new data. Both possibilities are shown in figure 4.8. The final design is shown as figure 4.9.

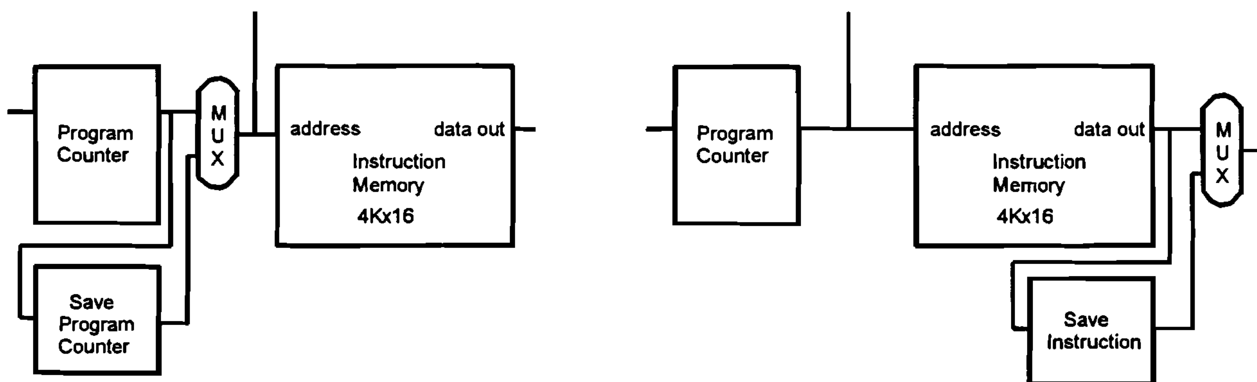


figure 4.8. Saving the input or output of the Read Only Memory in case of a stall.

Saving the former instruction address has a disadvantage for timing reasons. The address setup time for the Read Only Memory is relatively long. If a stall occurs, which takes a relatively long time to detect, the multiplexor in front of the memory has to switch and the address for the memory may not be stable in time. The setup time for a normal register is much shorter so switching the output data is preferred above switching the input address.

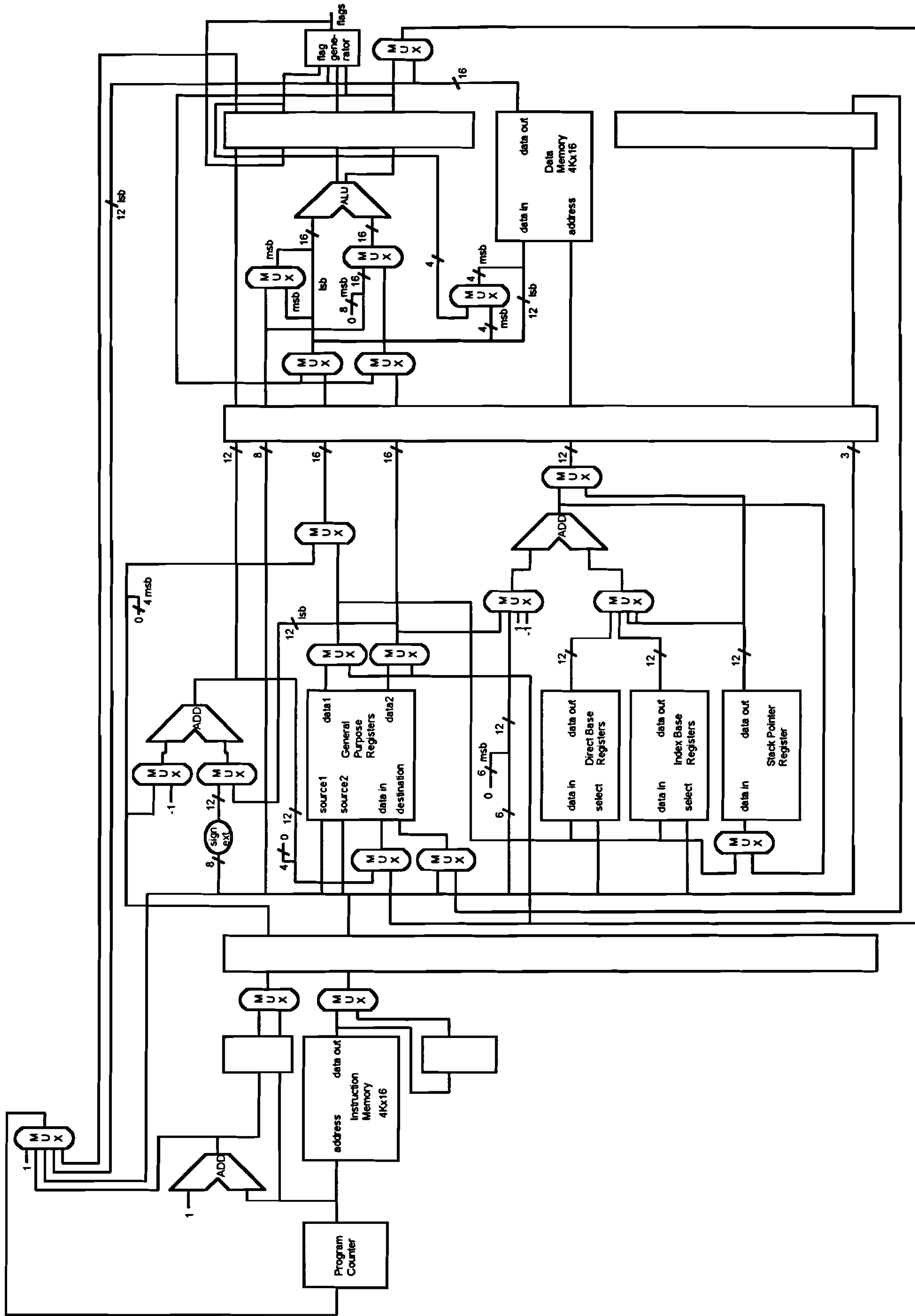


figure 4.9. The final design of the microprocessor.

As mentioned before, stalling will prevent the program counter from being updated.
If there is a branch taken the program counter must be updated so it can not be kept in a 'hold' position as in case of a stall. So all pipeline stages must continue their operations.
In case a conditional branch instruction decides the branch to be taken it takes two clock cycles to have the Fetch stage produce the next valid instruction for the Decode stage.
As the Fetch stage continues all its operations it will still produce unwanted instructions during those two clock cycles. The Decode stage will receive instructions which were not meant to be processed. This can be solved by filling the instruction register (the input register for the Decode stage) with two consecutive *nop* instructions. The *nop* instruction tells the processor to just do nothing in that particular stage the instruction is in. Filling the instruction register with *nop* instructions can be done by pre-setting the register.
As the number of *nop* instructions to be inserted can differ for the several instructions which can cause a branch hazard, it is necessary to use a finite state machine controller to control the loading of the *nop* instructions into the instruction register of the Decode stage. In the next chapter, which will describe the implementation of the microprocessor, this finite state machine will be explained in more detail.

The *rts* and *rti* instructions are very alike. The only difference is that the flags are restored as well in case of an *rti* instruction. Can the call for the interrupt routine be as simple as the call for a normal subroutine? The answer is yes. The only difference is that the call for the interrupt routine must also store the flag bits together with the return address. In case of an interrupt a special instruction can be forced into the instruction register so each pipe stage will know what to do. This will be called the *int* instruction.

In case of a *jsr* instruction it is obvious to save the return address of the next instruction following the *jsr* instruction. An interrupt can also be considered as a jump to a subroutine (the interrupt routine). But what will the return address be? With 4 instructions already in the pipeline it is not trivial what address that will be.
If there are no branch instructions in the pipeline the current program counter contents can be saved as the return address.
But in case there is a branch instruction in the pipeline, the pipeline needs to be drained to decide if the branch needs to be taken. As soon as there is no branch instruction left in the pipeline the instruction for the interrupt can be started.
Instructions still in the pipeline can be finished normally if no branches are taken. The address of the last instruction in the pipeline, incremented by one, must then be saved on the stack as the return address. If a branch must be taken the branch address must be saved on stack and the instructions following the branch instruction must be discarded.
If the pipeline contains an *rts* instruction when an interrupt occurs, the pipeline has to be drained to resolve the return address from memory. This address can then be regarded as the return address.
In the next chapter this will be explained in more detail.

5 Modelling and simulation.

Verifying the design can be done by simulation. There are several tools for simulating digital circuits. Each tool has its own particular way of modelling the parts of the design. A hardware description language many people use today is VHDL. But VHDL is only a description language. One needs a good compiler and an environment to check the results. If a test is run and an error has been detected in the design, the simulation application must be stopped and the editor must be started to edit the design. Then the design must be recompiled, the simulator can be started again and another test can be run.

This design has been modelled with IDaSS. IDaSS is an interactive design and simulation environment for synchronous digital circuits. The advantage of IDaSS is that the models can be changed during the tests without quitting the simulator. Each modification is instantaneous. If a function is changed during a test, the output of the corresponding function immediately adapts according to the change.

This chapter will describe the way the design is implemented in IDaSS. The design is divided into several smaller parts. These parts are the pipeline stages as have been defined in chapter 4. They are shown again in figure 5.1. The pipeline stages Pre-fetch and Fetch will be combined into one part named Fetch.

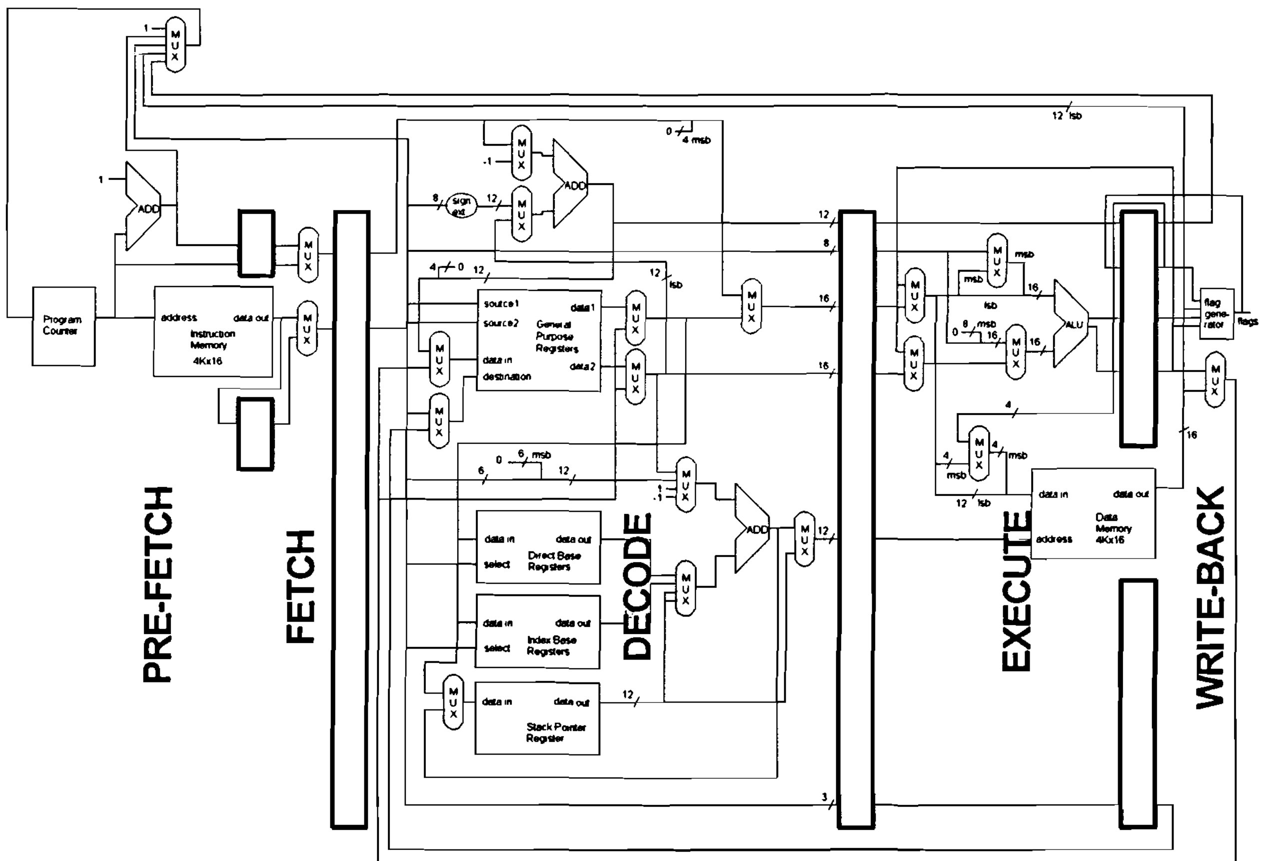


figure 5.1. The design op the microprocessor devided into 5 stages.

As the datapaths for the microprocessor have already been explained in chapter 4, this chapter will concentrate on the control logic for the parts of that datapath.

Each pipeline stage is drawn in IDaSS with a layout corresponding to that of the datapath layout as shown in figure 4.9. Appendix A contains the details of the IDaSS models.

5.1 The microprocessor environment.

The microprocessor will be part of chip for data compression as is explained in chapter 1. In our design the Data Memory was part of the datapath but it is actual an external part of the microprocessor. The microprocessor will be fitted with input and output busses and signals, which will be called the addressbus and the databus, and the control signals necessary to control these busses. This is shown at the right side of figure 5.2. The left side shows the interrupt input. As the Read Only Memory is also a complex and large part of the microprocessor it is also drawn as an external part though it is part of the microprocessor.

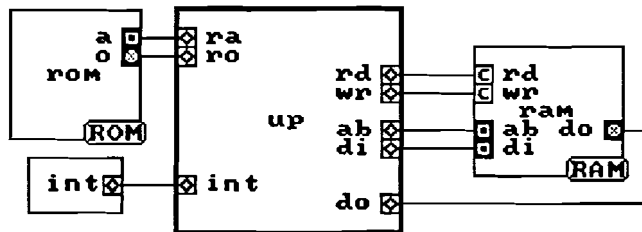


figure 5.2. The microprocessor must communicate with the external Data Memory.

The read (rd) and write (wr) signals control the external databus. Mind however that the Data Memory is not all Random Access Memory (RAM) but also contains memory mapped I/O. Further more it contains separate busses for data to and for data from that memory.

The microprocessor is split into 4 parts as shown in figure 5.2. These parts represent the pipeline stages as defined in chapter 4.

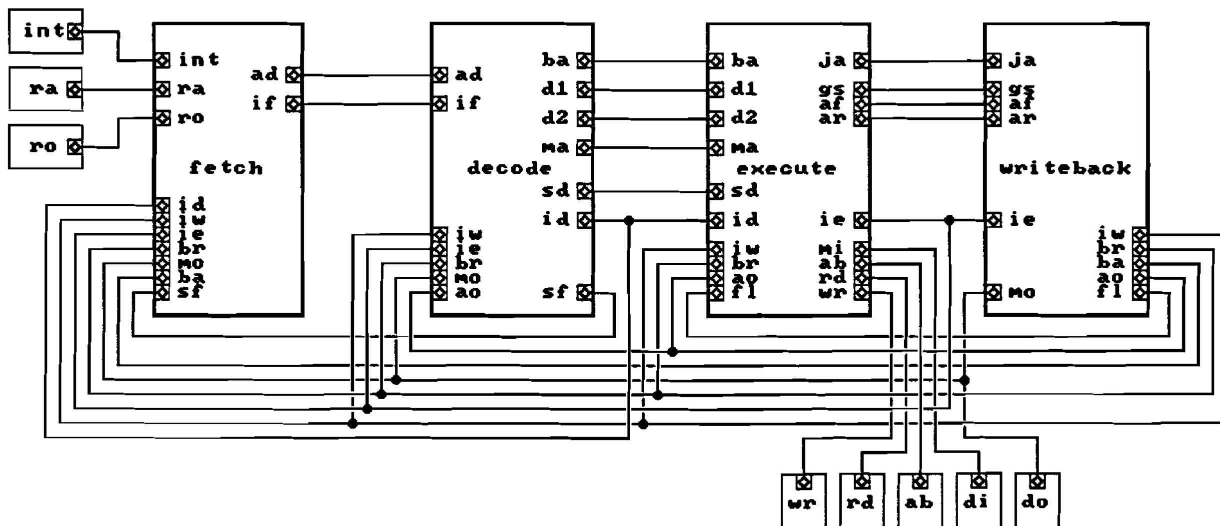


figure 5.3. The microprocessor can be split into smaller parts.

5.2 The Fetch stage.

This stage actually includes the Pre-fetch stage and the fetch stage. This stage holds the Program Counter and the datapaths for generating the instructions. The Read Only Memory is part of this stage but is placed externally.
This stage drawn in IDaSS is shown in figure 5.4.

Figure 5.4. The fetch stage drawn in IDaSS.

The registers *nia* (next instruction address), *pcs* (program counter save), and *si* (save instruction) form the pipeline which separates the actual Pre-fetch stage from the actual Fetch stage.

This stage needs control signals for the multiplexors. The multiplexor *namux* (next address) is controlled by *namuxc*. The multiplexors *amux* (address) and *imux* (instruction) are controlled by *intctrl* which is a finite state machine controller that takes care of starting the interrupt call at the right time. The external interrupt signal will first be synchronised by a register. This register is called *ireg*. The register *sic* controls the register *si*.

5.2.1 The logic block *namuxc*.

The multiplexor *namux* switches between the several busses which can generate the 'next instruction address' for the program counter.

The controller *namuxc* controls the multiplexor *namux*. This multiplexor must switch between 5 different inputs. In the IDaSS drawing the multiplexor only has 4 inputs but one of the 5 inputs is a constant value. This value is generated internally.
These 5 values are coded with 3 bits as is shown in table 5.1.

function	control value	select input	
sela4	000	a4	load incremented program counter
sela1	001	a1	load jump address
sela3	010	a3	load return address
sela2	011	a2	load branch address
selint	1xx	value 001h	load interrupt start address 001h

table 5.1. The control signals for namux.

The control signal *C* is formed by the functions as shown in figure 5.5. The signal *id* is the instruction which is active in the Decode stage. The signal *iw* is the instruction which is active in the Write-back stage. The signal *br* is 1 in case a conditional branch instruction in the Write-back stage decides that a branch must be taken.

```
---------------------V---------------------
"_jmp is 1 in case id is a jsr or jmp instruction"
_jmp := ((id from: 13 to: 15) = %100).
"_ret is 1 if iw is a rts or rti instruction"
_ret := ((iw from: 13 to: 15) = %111) Λ (iw at: 5) Λ ((iw at: 3)not).
"_int is 1 in case id is an interrupt call"
_int := ((id from: 13 to: 15) = %111) Λ ((id from: 3 to: 5)=%010).

"note: execution of rts or rti inserts 4 nops in the decode stage."
"So _jmp and _ret can not be active at the same time"
"_int can not be active together with a branch, _jmp or _rts"

c := _int,(br V _ret),(br V _jmp)
---------------------^---------------------
```

figure 5.5. The model for namuxc.

5.2.2 The logic blocks *amux* and *imux*.

The Logic block *amux* is an ordinary multiplexor. It switches between the two inputs *nia* and *pcs* as is shown in table 5.2. Switching is done by the finite state machine controller *intctrl*.

function	
nia	select input nia (next instruction address)
pcs	select input pcs (program counter save)

table 5.2. The functions of multiplexor amux.

The logic block imux is also a multiplexor. The control however is somewhat more complex. The finite state machine controller *intctrl* switches between three functions. One of these functions is the generating of the *nop* instruction. Another is the generating of the *int* instruction.

The third function is named *normal* and is an ordinary multiplexor function which switches between the two inputs *rom* and *si*. Switching between these two inputs is controlled by the input signal *sic*. This is shown in table 5.3.

function	sic signal	output value
int	-	value FFD7h
nop	-	value FFFFh
normal	0	from input rom
	1	from input si

table 5.3. The outputs of logic block imux.

5.2.3 The finite state machine controller *intctrl.*

This part of the microprocessor is probably the most complex part. In case of an interrupt it takes care of saving the right return address on the stack and activates the interrupt routine at the proper time.
This microprocessor does not allow nested interrupts. If an interrupt routine is started the interrupt signal will be masked until the interrupt routine has finished.
The interrupt routine would poll the various possible interrupt sources to see which event caused it. Reading the status register of the interrupt source must deactivate the external interrupt signal. After finishing the interrupt routine a new interrupt routine can be started if the external interrupt signal is active again.

As we have seen in paragraph 4.1 (figure 4.5) the flags must come from the flag register of the Write-back stage. Between the interrupt instruction and the last instruction of the interrupted program, a *nop* instruction must be inserted to assure that the proper flags are saved on the stack.

The most simple case for starting the interrupt routine is when there are no address altering instructions in the pipeline (conditional branch instructions, *rti*, *rts*, *jmp* and *jsr*) and the stall signal (*sf*) is not active. As the finite state machine is implemented as a Moore-machine during the first clock cycle when the synchronised interrupt signal is active, the interrupt signal will only prepare the finite state machine to jump to another state. During that clock cycle the microprocessor will continue its uninterrupted operation. On the next clock the instruction which was prepared by the Fetch stage will be clocked into the instruction register of the Decode stage. At the same time the *pcs* (program counter save) register will hold the address of the next instruction that is fetched from the Read Only Memory. So if we hold that address, discard the instruction that resides at that address (which at that time comes from the instruction memory or the *si* register) and present a *nop* and an *int* instruction sequentially the interrupt routine call can be started. Together with the interrupt instruction the return address coming from the *pcs* register will be passed to be saved on stack.

But what happens if there are address altering instructions in the pipeline or a stall occurs at the time the internal interrupt signal is activated? Each of these possibilities wil be examined.

1. As explained above when no stalls occur and/or there are no address altering instructions present in the pipeline the *pcs* register must be held and a *nop* instruction must be inserted. Then the *int* instruction will be inserted together with the contents of the *pcs* register which is the return address that has to be saved on the stack.

2. If a stall occurs the interrupt has to be stalled as well until the stall is lifted. If no further address altering instructions are present in the pipeline the interrupt call can be initiated as in situation 1.

3. If the branch signal (*br*) is active a conditional branch must be taken. All instructions in the pipeline are discarded and need no further investigation. The address to be branched to will be present in the *pcs* register after 2 clock cycles. That address must be saved as the return address of the interrupt routine. During those two clock cycles *nop* instructions can be inserted. This is not really necessary because the controller of the Decode stage will reset the instruction register during those clock cycles as we will see later when the function of that controller is explained.

4. In case there are conditional branch instructions present in the Decode stage and/or the execute stage, the pipeline has to be drained until the last conditional branch instruction is present at the Write-back stage. During that time the *pcs* register must keep its contents. If the branch must be taken situation 3 occurs again. If no branch occurs situation 1 will arise.

5. If the Write-back stage contains a *jmp* or *jsr* instruction the Fetch stage has already fetched the new instruction from the Instruction Memory. The *pcs* register will hold the address of the new instruction. The Decode stage and Execute stage will contain *nop* instructions due to the presence of the *jmp* or *jsr* instruction in the Write-back stage so it is not necessary to check for the instructions in the rest of the pipeline. Situation 1 can be applied here again.

6. If the Write-Back stage contains an *rts* instruction the address coming from the Data Memory must be used as the return address for the interrupt routine. Two *nop*s have to be inserted before the address is present in the *pcs* register. Then the *int* instruction can be inserted. The Decode stage and Execute stage will contain *nop* instructions due to the presence of the *rts* instruction in the Write-back stage.

7. If the Execute stage contains an *rts* instruction situation 6 occurs again except that an extra *nop* must be inserted before the *int* instruction can be inserted. The Decode stage will contain a *nop* instruction due to the presence of the *rts* instruction in the Execute stage.

8. If the Execute stage contains a *jmp* instruction (and the branch signal is not active) or *jsr* instruction the Program Counter Register is already holding the address of the next program instruction. While inserting a *nop* instruction the *pcs* register must be loaded with the contents of the Program Counter Register after which the *int* instruction can be inserted in the next clock cycle.

9. If the Decode stage contains an *rts* instruction it is possible that the Execute stage contains a conditional branch instruction. This will however result in a stall while the conditional branch instruction moves on to the Write-back stage. If the condition decides against branching the *rts* instruction can proceed its execution. After one clock cycle the *rts* instruction has moved on to the Execute stage after which situation 7 occurs again.

10. If the Decode stage contains a *jsr* instruction it is possible that the Execute stage contains a conditional branch instruction. This will however result in a stall while the conditional branch instruction moves on to the Write-back stage. If the condition decides against branching the *jsr* instruction can proceed its execution. After one clock cycle the *jsr* instruction has moved on to the Execute stage after which situation 8 occurs again.

11. If the Decode stage contains a *jmp* instruction it is possible that the Execute stage contains a conditional branch instruction. This will not result in a stall! The *jmp* instruction will proceed its normal behaviour and can be overruled if the condition decides to branch when arrived at the Write-back stage. In that case situation 3 occurs again. If the condition decides against branching the jump address has already arrived at the Program Counter Register. The *pcs* register must load the jump address on the next clock. This jump address will be saved as the return address.

All these situations must be controlled by the finite state machine controller *intctrl*. The state description diagram is presented as figure 5.6. For each of the states the functions of the two multiplexors *imux* and *amux* and the register *pcs* are shown in table 5.4. The IDaSS model is to large to show here. It can be found in Appendix A.

State 11 is entered when the *int* instruction has been inserted into the pipeline. It will not continue to state 1 until the *rti* instruction has arrived at the Write-back stage. This prevents nesting of interrupt routines.

If there are no address altering instructions in the Decode stage, Execute stage and/or Write-back stage the finite state machine will proceed to state 9 on an interrupt. When arrived at that state the Decode stage will have received a new instruction which still needs to be processed. This instruction can again be an address altering instruction. If not, the finite state machine will proceed to state 3 where it will insert the instruction *int* to start the interrupt routine.

figure 5.6. The state transition diagram of intctrl.

state	amux	imux	pcs
1	nia	normal	load
2	pcs	nop	load
3	pcs	int	load
4	pcs	nop	hold
5	pcs	nop	load
6	pcs	nop	load
7	pcs	nop	hold
8	pcs	nop	hold
9	pcs	nop	hold
10	pcs	nop	load
11	nia	normal	load

table 5.4. The functions of amux, imux and pcs in each state of intctrl.

5.3 The Decode stage.

The Decode stage holds all the Data Registers. These are the 8 General Purpose Registers (GPR), the 16 Index Base Registers (IBR), the 2 Direct Base Registers (DBR) and the Stack Pointer Register (SPR). As the latter 3 type of registers are never used at the same time they can be clustered into one big registerfile of 19 registers. They are modelled as the logic block *regs.* The address selection of these registers is done by a logic block named *regsel*. The pipeline register is formed by the registers *adreg* and *idreg*. The instruction addresses are passed by *adreg*. The instructions that are generated by the Fetch stage are clocked into the instruction register *idreg* which is controlled by a finite state machine controller *idctrl.*
The Decode stage drawn in IDaSS is shown in figure 5.7. At the top of the drawing are 4 logic blocks which control this stage. The logic block *decctrl* controls most parts of this stage, such as the multiplexors. The logic block *wbctrl* generates the write signal for the General Purpose Registers. The logic block *fbctrl* controls the feedback multiplexors of this stage. The logic block *stall* generates the stall signal.

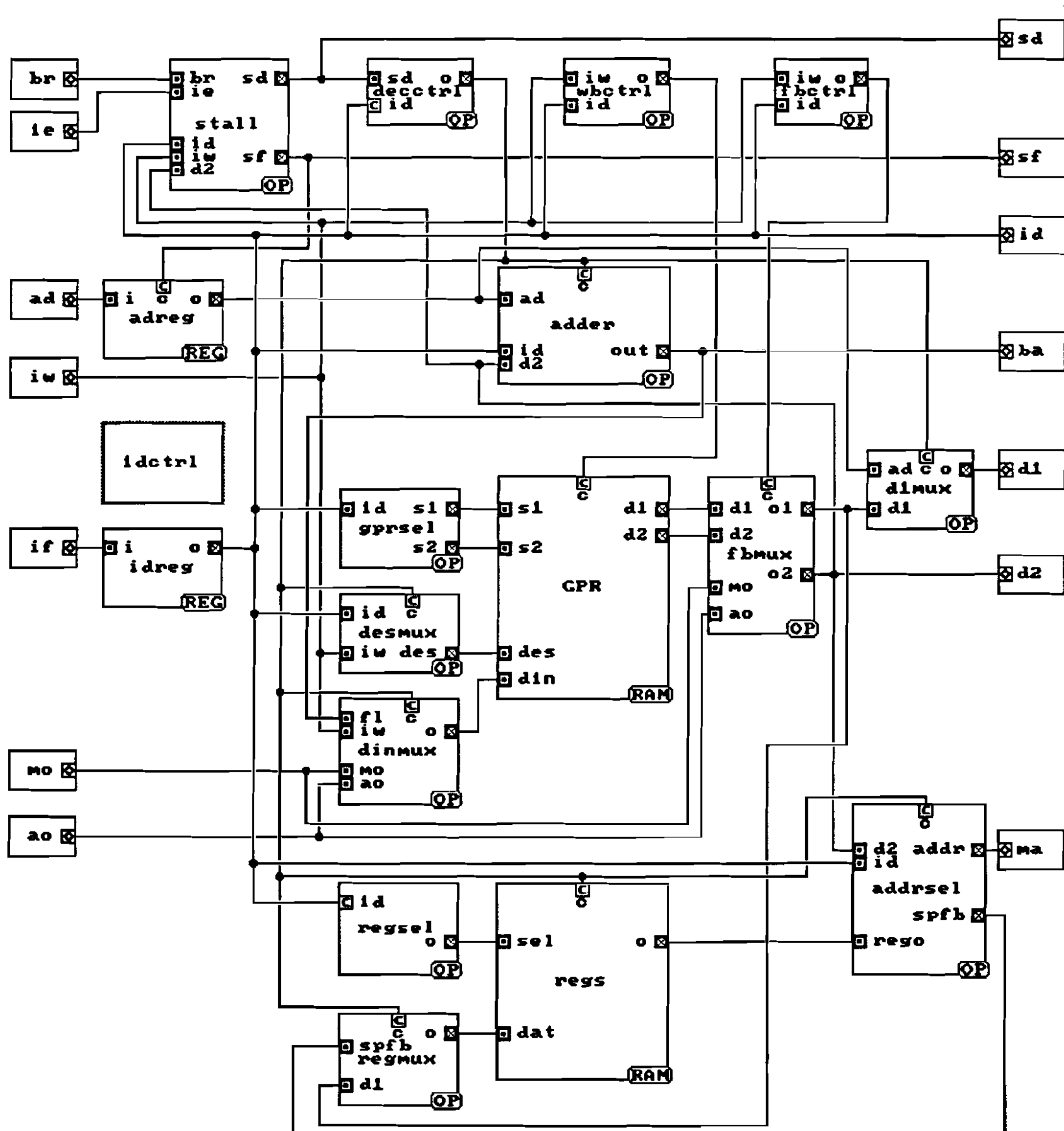


figure 5.7. The IDaSS drawing of the Decode stage.

5.3.1 The logic block *decctrl.*

This block is actually the instruction decoder. It switches the functions of several logic blocks. Those logic blocks are shown in tables 5.5 to 5.10.

function	control value	
seld1	0	select input d1
selad	1	select input ad

table 5.5. The functions of multiplexor ***d1mux.***

function	control value	
seliw	0	select bits 0..2 of iw as input
selid	1	select bits 6..8 of id as input

table 5.6. The functions of multiplexor ***desmux.***

function	control value	
seld1	0	select input d1
selspfb	1	select input selspfb

table 5.7. The functions of multiplexor ***regmux.***

Multiplexor *dinmux* is somewhat more complex. It has 3 input busses to be switched to which means that it needs 2 control lines. One of those control lines comes from the controller *decctrl.* It switches between the output of the adder (at the top) in case of a *fill* instruction and the output of the Write-back stage. The latter however can be the output of the ALU or the output of the Data Memory. An extra control line is needed to select between those two inputs. This control line is not fitted as one signal but as the instruction bus iw. A function internal of *dinmux* determines which input must be selected. This internal signal will be called *am* (ALU or memory). If it is zero, data from the ALU will be selected. If it is one, data from the Data Memory is selected.

function	control value	am value	
alumem	0	0	select input ao
		1	select input mo
adder	1		select input fl

table 5.8. The functions of multiplexor ***dinmux.***

The logic block named *adder* at the top of figure 5.10 contains the adder which calculates the branch address. The branch address is calculated by adding the displacement to the next instruction address. The displacement is formed by bits 4 to 11 of the instruction word id extended to 12 bits. This adder is also used to decrement the contents of one of the General Purpose Registers in case of a *fill* instruction. The controller *decctrl* will switch between those two functions.

function	control value	
add	0	the output will be the branch address
decr	1	the output will be the decremented input d2.

table 5.9. The functions of logic block ***adder.***

The last block controlled by *decctrl* is the logic block *addrsel*. This block contains the adder to calculate the Data Memory address. The adder is also used to decrement or increment the contents of the Stack Pointer Register.

function	control value	
direct	00	add bits 6..11 of id extended with zeroes to input regout
indexed	01	add bits 0..11 of d2 to input regout
pop	10	both outputs have value regout + 1
push	11	addr := regout, spfb := regout - 1

table 5.10. The functions of logic block ***addrsel.***

Table 5.11 shows the several types of instructions and the functions of the logic blocks for each of those types. The *wrreg* signal is the write signal for the register file *regs*. Next to the instruction input *id* the logic block *decctrl* has another input. This is input *sd*. This input will be 1 if a stall or a branch occurs. In those cases the Decode stage must switch to a passive state. This is done by making all the control signals zero.

instruction word	type of instruction	addrsel	regmux	dinmux	desmux	wrreg	d1mux	adder
%000XXXXXXXXXXXXX	indexed binary operations	01	0	0	0	0	0	0
%001XXXXXXXXXXXXX	direct binary operations	00	0	0	0	0	0	0
%01XXXXXXXXXXXXXX	immediate binary operations	00	0	0	0	0	0	0
%1000XXXXXXXXXXXX	jump instruction	00	0	0	0	0	0	0
%1001XXXXXXXXXXXX	jump subroutine	11	1	0	0	1	1	0
%1010XXXXXXXXXXXX	conditional jump	00	0	0	0	0	0	0
%1011XXXXXXXXXXXX	the djnz instruction	00	0	0	0	0	0	0
%1100XXXXX0XX0XXX	the pop instruction	10	1	0	0	1	0	0
%1100XXXXX0XX1XXX	the push instruction	11	1	0	0	1	0	0
%1100XXXXX1XXXXXX	other unary operations	00	0	0	0	0	0	0
%1101XXXXXXXXXXXX	reg. to reg. binary operations	00	0	0	0	0	0	0
%111XXXXXXX000XXX	the fill instruction	01	0	1	1	0	0	1
%111XXXXXXX001XXX	move to IBR, DBR or SPR	00	0	0	0	1	0	0
%111XXXXXXX010XXX	jump to interrupt routine	11	1	0	0	1	1	0
%111XXXXXXX1X0XXX	return from int./subroutine	10	1	0	0	1	0	0
%111XXXXXXX111XXX	the nop instruction	00	0	0	0	0	0	0

table 5.11. The logic block decctrl controls the functions of six other logic blocks.

5.3.2 The logic block *wbctrl*.

This logic block controls the 'write'-signal of the General Purpose Registers. The *fill* instruction is the only instruction in the Decode stage that can write to these registers. All other instructions that can modify the contents of one of the General Purpose Registers will only do so when they are in the Write-back stage of the pipeline.

These instructions are:

- The binary operation *mov*, which moves data from the Data Memory to a General Purpose Register. This can be either an indexed or direct binary operation.
- Any other binary operation except for the *cmp* operation.
- All unary operations except for the *push* operation.
- The djnz instruction.

These conditions are modelled with IDaSS and shown in figure 5.8.

```
---------------------V---------------------
"write back to a GPR"

_fillid := ((id from: 13 to: 15) = %111) Λ ((id from: 3 to: 5) = %000). "the fill instruction"
_binwriw := (((iw from: 14 to: 15) = %00) Λ ((iw at: 3) = %0)) V "direct or indexed mov operation"
          ((((iw from: 14 to: 15) = %01) V ((iw from: 12 to: 15) = %1101)) Λ "other binary operations"
          ((iw from: 3 to: 5) ~= %111)). "except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) Λ (((iw from: 3 to: 6) = %0001) not). "all unary except push"
_djnziw := ((iw from: 12 to: 15) = %1011). "the djnz instruction"

wr := _fillid V _binwriw V _unwriw V _djnziw
---------------------^---------------------
```

figure 5.8. The IDaSS model of wbctrl.

5.3.3 The logic block *fbctrl.*

This logic block controls the two feed-back multiplexors of the Decode stage. The logic block *fbmux* contains those multiplexors and table 5.12 shows the coding of the control signals. Each of the two multiplexors switches between the output of the General Purpose Register, the output of the ALU or the output of the Data Memory. The outputs are named *o1* and *o2*.

function	control value	output o1	output o2
normal	X00	data from GPR	data from GPR
fbd1alu	001	data from ALU	data from GPR
fbd1mem	101	data from Data Memory	data from GPR
fbd2alu	010	data from GPR	data from ALU
fbd2mem	110	data from GPR	data from Data Memory
fbbothalu	011	data from ALU	data from ALU
fbbothmem	111	data from Data Memory	data from Data Memory

table 5.12. The coding of the control signal C for logic block fbmux.

Of course these feed-backs will only be necessary in case the Write-back stage contains data to be written in one of the General Purpose Registers and the instruction currently in the Decode stage needing that data. In case of a feed-back the data comes from either the ALU or from the Data Memory. They can not produce their data at the same time.
The instructions at the Write-back stage that can write data to one of the General Purpose Registers are already discussed at section 5.3.2. Figure 5.9 shows the IDaSS model of *fbctrl.*

```
--------------------V--------------------
"feedback control for the decode stage"
"fbo1 = (iws1 = ids1) and RGWiw  : feedback data of source1"
"fbo2 = (iws1 = ids2) and RGWiw  : feedback data of source2"

"_mem is 1 in case of a memory to register move and in case of a pop"
_mem := (((iw from: 14 to: 15) = %00) Λ ((iw from: 3 to: 5) = %000)) V
        (((iw from: 12 to: 15) = %1100) Λ ((iw from: 3 to: 6) = %0000)).

"write back to a GPR"
_binwriw := (((iw from: 14 to: 15) = %00) Λ ((iw at: 3) = %0)) V  "direct and indexed binary mov from mem"
            (((iw from: 14 to: 15) = %01) Λ ((iw from: 3 to: 5)~=%111)) V  "imm binary ops except cmp"
            (((iw from: 12 to: 15) = %1101) Λ ((iw from: 3 to: 5)~=%111)). "reg-reg binary ops except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) Λ (((iw from: 3 to: 6) = %0001) not). "unary ops except a push"
_djnziw := ((iw from: 12 to: 15)=%1011).  "the djnz instruction"

"_RGWiw is 1 if the instruction in iw writes back to one of the GPRs"
_RGWiw := _binwriw V _unwriw V _djnziw.

c := _mem,
   (((iw from: 0 to: 2) = (id from: 6 to: 8)) Λ _RGWiw)  "fbo2"

   (((iw from: 0 to: 2) = (id from: 0 to: 2)) Λ _RGWiw)  "fbo1"
--------------------^--------------------
```

figure 5.9. The IDaSS model of fbctrl.

5.3.4 The logic block *stall.*

This block is the most complex part of the Decode stage. It will produce the stall signals for the microprocessor. In case an instruction arrives at the Decode stage that can not start its operation because it needs data that will not be available until the next clock cycle, it will have to stall the fetching of the next instruction. The instruction in the Decode stage must stay passive until it can start its operation. The instructions at the Execute and Write-back stages must continue their operations. There are several reasons why a stall can occur:

- As the Data Memory is a relatively slow device, the data from that memory can not be fed-back to the Execute stage. This means that any instruction that loads data from the Data Memory and that is immediately followed by an instruction needing that data must be stalled.
- The *fill* instruction will write back data to one of the General Purpose Registers on each clock cycle. It must stall its operation if there are instructions in the Execute and/or Decode stage that want to write back their results to one of the General Purpose Registers even if different registers are used.
- If there is a *mov* instruction to one of the Index Base Registers, Direct Base Registers or to the Stack Pointer Register and the data to be written to that particular register is going to be produced by the preceding instruction the *mov* instruction has to be stalled until that data is produced by the Write-back stage one clock cycle later.
- An indexed operation needs the contents of one of the General Purpose registers for the calculation of the memory address. If the contents of that particular register is going to be changed by the instruction preceding the indexed operation then a stall is inevitable.

- If there is a conditional instruction in the Write-back stage and the condition decides for branching all other instructions in the pipeline have to be flushed. Therefore no alterations to register contents may be made by the instructions directly following the conditional branch instruction. Instructions that alter register contents in the decode stage are: *fill* (General Purpose Register), *pop*, *push*, *jsr*, *rts* and *rti* (Stack Pointer Register) and moves to one of the Direct Base Registers, Index Base registers or Stack Pointer Registers.

There are different conditions for stalling the Decode stage and for stalling the Fetch stage. Both pipeline stages will have to stall on the conditions mentioned above. But the Fetch stage also has to stall when the Decode stage contains the fill instruction until the contents of the selected General Purpose Register has been decremented to zero.

On a stall all control signals of the Decode and Fetch stage will get a passive value which results in a situation where no modifications are being made. For the Fetch stage it will amongst other things result in not incrementing the program counter. The stall signal *std* will force the instruction register of the Execute stage to load a nop instruction during the next clock cycle.
Flushing the Decode stage in case of a branch will have the same effect for the control signals as in case of a stall. So these signals can be combined into one signal. The Fetch stage on the other hand must continue its operation in case of a branch and may not respond to the stall signal when the branch signal is active.
As the IDaSS model is too long to print here only a summary of the IDaSS model will be given here as figure 5.10. The complete model can be found in Appendix A.

```
----------------------V----------------------
"stf is the stall signal for the Fetch stage"
"std is the stall signal for the Decode stage"

"id is the instruction in the Decode stage"
"ie is the instruction in the Execute stage"

"_idmemrd is 1 when data will go from memory to a register"
"_samereg1 is 1 if the write-back register of ie is also used in id as register1"
"_samereg2 is 1 if the write-back register of ie is also used in id as register2"
"_regop is 1 in case an instruction uses one of the General Purpose Registers"
"_regwrie is 1 if ie will write-back to a register"
"_regwriw is 1 if iw will write-back to a register"
"_indid is 1 if id is an indexed operation"
"_fillid is 1 if id is a fill instruction"
"_popid is 1 if id is a pop instruction"
"_pushid is 1 if id is a push instruction"
"_returnid is 1 if id is an rti or rts instruction"
"_jsrid is 1 if id is a jump subroutine instruction"
"_movrid is 1 if id is a move to IBR/DBR/SPR instruction"
"_condbrie is 1 if ie is a conditional branch instruction"
"_std to reuse the equation of std for stf"
```

```
_std := (_idmemrd  Λ _regop Λ (_samereg1 V _samereg2)) V
        (_fillid Λ (_regwrie V _regwriw)) V
        (_fillid Λ _condbrie) V
        (_popid Λ _condbrie) V
        (_pushid Λ _condbrie) V
        (_returnid Λ _condbrie) V
        (_jsrid Λ _condbrie) V
        (_movrid Λ _condbrie) V
        (_movrid Λ _regwrie Λ _samereg1)  V
        (_indid Λ _regwrie Λ _samereg2).

std := _std V br.
stf := (_std V (_fillid Λ ((d2 = 0)not))) Λ ((br)not)
-----------------Λ------------------
```

figure 5.10. Part of the IDaSS model of stall.

5.3.5 The finite state machine controller *idctrl.*

The last block of the Decode stage that hasn't got any attention so far is the finite state machine controller *idctrl*. This finite state machine controls the functions of the instruction register *idreg*. This register can be hold, preset or loaded. In case of a stall it must be hold.
In case of some instructions *nop* instructions need to be inserted to guarantee proper operation of the pipeline. For instance in case a jmp instruction is decoded in the Decode stage it takes two clock cycles to select the new program address and fetch the new instruction from the Instruction Memory. As the Fetch stage can not flush any instructions the instruction register *id* will have to do that. During the two clock cycles after the jmp instruction has left the decode stage the instruction register *id* must be preset to produce the *nop* instruction. The two instructions produced by the Fetch stage will automatically be rejected.
The instructions that need this feature are the instructions *jmp*, *jsr*, *rts*, *int* and *rti*. The instructions *rts* and *rti* need four *nop* instructions inserted. The other three instructions need only two *nop* instructions inserted. Two *nop* instructions need also be inserted in case a conditional branch is taken.
If there is a *jmp* instruction in the Decode stage and a conditional branch instruction in the Execute stage the finite state machine controller will start producing two sequential *nop* instructions. When the conditional branch instruction arrives at the Write-back stage one clock cycle later and the condition decides that the branch must be taken the pipeline must be flushed and the *jmp* instruction must be discarded. The finite state machine controller has only produced one nop instruction at that time. It must be stopped generating the second nop instruction but on the other hand for the branch to be taken it must again start to generate two sequential *nop* instructions.
In case the function of the instruction register must be changed it must be done before the next clock. Therefore this finite state machine must be implemented as a Mealy machine.
The state transition diagram is shown in figure 5.11. With each transition the function for the instruction register *idreg* is given.

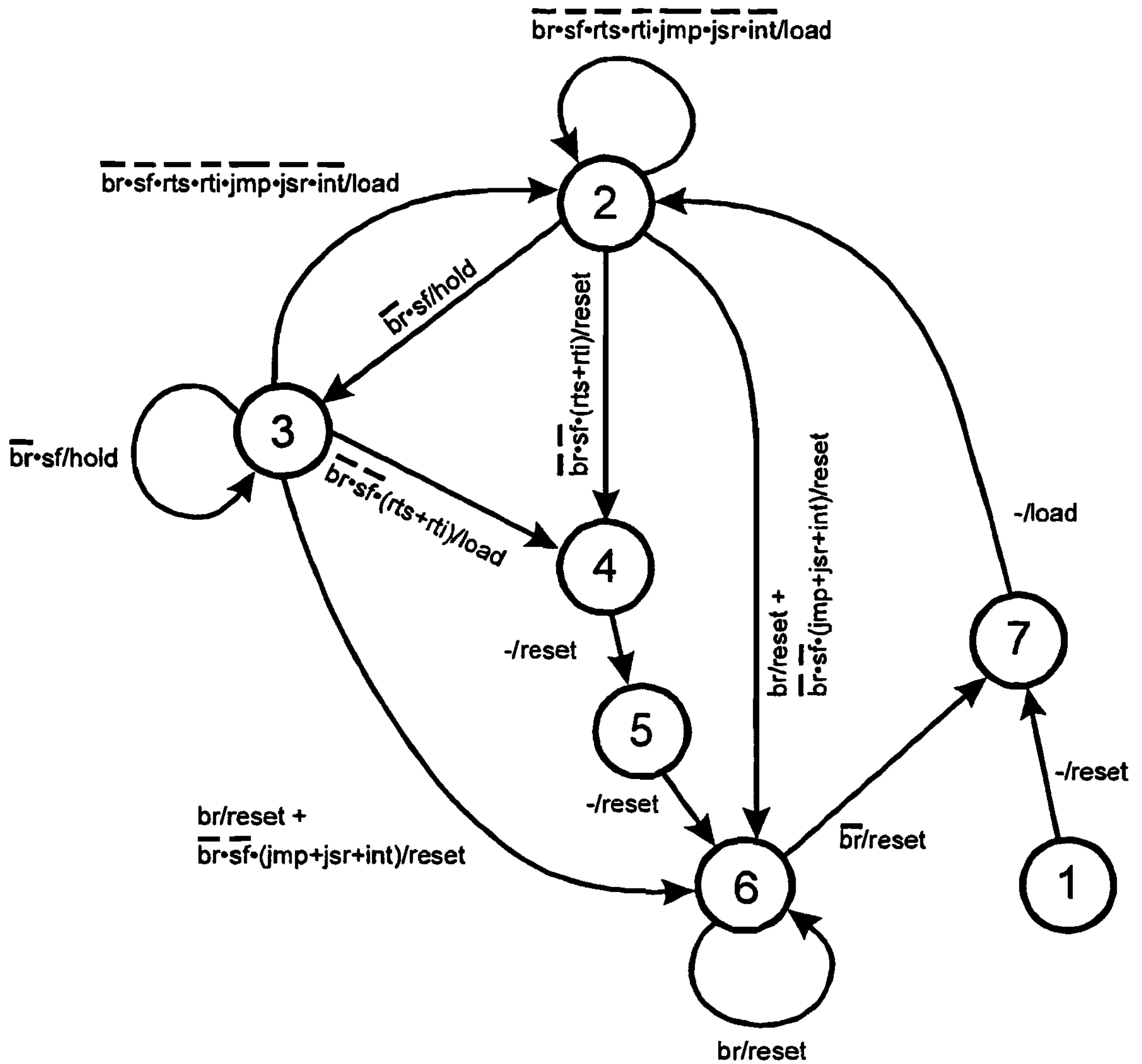


figure 5.11. The state transition diagram of idctrl.

5.4 The Execute stage.

The Execute stage holds the ALU and it prepares the data for the Data Memory. As the Data Memory is a synchronous device it will be part of the pipeline register between the Execute stage and the Write-back stage. The parts of the Execute stage are controlled by two logic blocks. The feed-back multiplexor is controlled by *fbctrl*. The other parts are controlled by *exctrl*. The latter also generates the read and write signals for the Data Memory. The IDaSS drawing of the Execute stage is shown in figure 5.12.

5.4.1 The logic block *fbctrl*.

The logic block *fbctrl* controls the two feed-back multiplexors which are modelled as logic block *fbmux*. As the Data Memory is a relatively slow device it will not be fed-back to the Execute stage. The only data that can be fed-back is the data from the ALU once it has passed the pipeline register at the output of the ALU. If the instruction in the Execute stage needs the data which is going to be written back by the Write-back stage it simply switches the multiplexors to that data.

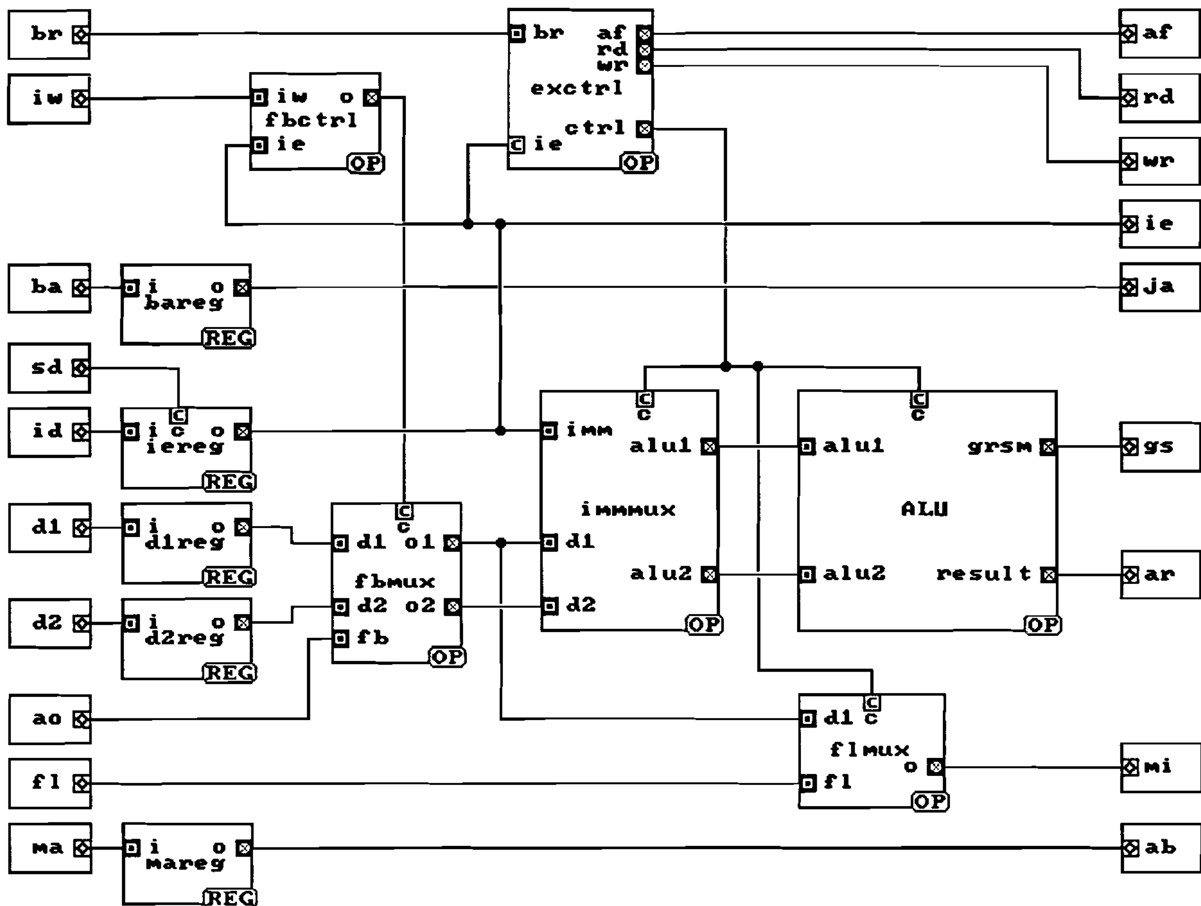


figure 5.12. The IDaSS drawing of the Execute stage.

The two multiplexor outputs are o1 and o2. They can be switched between data from the Decode stage and the feed-back data. Table 5.12 shows the control signals for the logic block *fbmux*.

function	control value	output o1	output o2
normal	00	data from input d1	data from input d2
fbdata1	01	data from ALU feed-back	data from input d2
fbdata2	10	data from input d1	data from ALU feed-back
fbboth	11	data from ALU feed-back	data from ALU feed-back

table 5.12. The coding of the control signal C of the logic block fbmux.

The simplest way to detect if a feed-back is necessary is to check whether the instruction in the Write-back stage is going to write back to one of the General Purpose Registers and then check if that specific register is also used by the instruction in the Execute stage. The simplest way is to check if the bits with which the registers of the General Purpose Registers are selected (bits 0..2 and bits 6..8) have a match with the write-back selection address of the instruction in the Write-back stage (bits 0..2) independent of the instruction in the Execute stage.
There is one exception to this solution. No feed-back may be applied if the instruction in the Execute stage is a *jsr* or *int* instruction. The data from the Decode stage must pass output *o1* the multiplexor as it must be written to the Data Memory. This data is the return address which must be written on the stack.

The IDaSS model of *fbctrl* is shown in figure 5.13.

```
---------------------V---------------------
"feedback control"
"fbo1 = (iws1 = ies1) and RGWiw and (ie <> jsr) and (ie <> int)"
"fbo2 = (iws1 = ies2) and RGWiw"

"write back to a GPR only from the ALU output"
_binwriw := (((iw from: 14 to: 15) = %01) /\ ((iw from: 3 to: 5)~=%111)) V "imm binary ops except cmp"
            (((iw from: 12 to: 15) = %1101) /\ ((iw from: 3 to: 5)~=%111)). "reg-reg binary ops except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) /\ ((iw from: 3 to: 6) ~= %0001). "all unary ops except a push"
_djnziw := ((iw from: 12 to: 15)=%1011). "the djnz instruction"

"_RGWiw is 1 if the instruction in iw writes its ALU output back to one of the GPRs"
_RGWiw := _binwriw V _unwriw V _djnziw.

ctrl := (((iw from: 0 to: 2) = (ie from: 6 to: 8)) /\ _RGWiw)          "fbo2"
    ,
    (((iw from: 0 to: 2) = (ie from: 0 to: 2)) /\ _RGWiw /\)          "fbo1"
    ((ie from: 12 to: 15) ~= %1001) /\)                               "not in case of a jsr instruction"
    ((((ie from: 13 to: 15) = %111) /\ ((ie from: 3 to: 5) = %010)) not)) "not in case of an int instruction"
---------------------^---------------------
```

figure 5.13. The IDaSS model of logic block fbctrl.

5.4.2 The logic block *exctrl.*

This block is actually the instruction decoder. It switches the functions of the logic blocks *flmux*, *immmux* and *ALU* and generates the control signals for the Data Memory. The logic blocks are shown in tables 5.13 to 5.15.

The logic block *flmux* always passes the lower 12 bits of input d1. The upper 4 bits can come from input *d1* or from input *fl*.

function	control value	
normal	0	select upper 4 bits from input d1
save	1	select upper 4 bits from input fl

table 5.13. The functions of multiplexor flmux.

The logic block immmux is the multiplexor that switches between normal data feed-through or data modifications concerning immediate data operations. This logic block contains two multiplexors that are switched by two control signals. These signals are coded as shown in table 5.14.

function	control value	
immlow	00	alu2 its lower part is imm value extended with zeroes
immhigh	01	alu1 its upper part is imm value, lower part that of d1
opimm	10	alu1 gets value of d1, alu2 same as immlow
opnormal	11	alu1 gets value of d1, alu2 gets value of d2

table 5.14. The coding of the control signal C of logic block immmux.

The logic block *ALU* can perform 16 different arithmetic operations. These operations are already discussed in section 2. Some adaptations have been made to the encoding of the unary operations. These are discussed in section 2.8. Table 5.15 shows the encoding of the functions the *ALU* must be able to perform.

function	control value	
mov	0000	the output gets the value of input alu2
mhi	0001	the output gets the value of input alu1
add	0010	alu1 + alu2
sub	0011	alu1 - alu2
and	0100	a bitwise and operation on both inputs
or	0101	a bitwise or operation on both inputs
xor	0110	a bitwise xor operation on both inputs
cmp	0111	the value of alu1 compared to that of alu2
sel2	1000	select the second nibble of alu1 (bits 4..7)
sel3	1001	select the third nibble of alu1 (bits 8..12, note: 5 bits)
cpl	1010	a bitwise invert of the bits of alu1
set	1011	the output bits all get value 1
dec	1100	alu1 - 1
shr	1101	shift all bits of alu1 right one bit, a zero is shifted in
shl8	1110	shift all bits of alu1 left 8 bits, zeroes are shifted in
shr8	1111	shift all bits of alu1 right 8 bits, zeroes are shifted in

table 5.15. The coding of control signal C of logic block ALU.

To keep the ALU as simple as possible the add and subtract functions are done with the same adder. Subtraction is done by adding the two's-complement value of the alu2 input. Therefore it is necessary to implement the adder as a 17-bit adder. The result will be the 16 most significant bits. Both ALU inputs will form the 16 most significant bits of the adder. On addition the least significant bit of both inputs will be zero. On subtraction the least significant bit of both inputs will be one.

The logic block *ALU* also generates two flags. They form the output *grsm* (greater/ smaller). The input alu1 will always be compared with input alu2. The coding of the signal grsm is shown in table 5.16.

grsm	
00	alu1 = alu2
10	alu1 > alu2
01	alu1 < alu2

table 5.16. The coding of the signal grsm.

Table 5.17 shows the several types of instructions and the functions of the logic blocks for each of those types. The *readmem* and *writemem* signals are the 'read' and 'write' control signals for the Data Memory. The *altflag* signal is used in the Write-back stage and denotes that the 'all set'-flag and 'zero'-flag may be altered by that instruction. Next to the instruction input *ie* the logic block ex*ctrl* has another input. This is input *br*. This input will be 1 if a branch occurs.

In those cases the Execute stage must switch to a passive state. This is done by making all the control signals zero, except the read signal of the Data Memory which will be one.

instruction word	type of instruction	flmux	aluop3	aluop 2..0	altflag	immux	writemem	readmem
%000XXXXXXXXXXXXX	indexed binary mov operation	0	0	a)	1	11	b)	c)
%001XXXXXXXXXXXXX	direct binary mov operation	0	0	a)	1	11	b)	c)
%01XXXXXXXXXXXXXX	immediate binary operations	0	0	a)	1	d)	0	1
%1000XXXXXXXXXXXX	jump instruction	0	0	a)	0	11	0	1
%1001XXXXXXXXXXXX	jump subroutine	0	0	a)	0	11	1	0
%1010XXXXXXXXXXXX	conditional jump	0	0	a)	0	11	0	1
%1011XXXXXXXXXXXX	the djnz instruction	0	1	100	1	11	0	1
%1100XXXXX0XX0XXX	the pop instruction	0	1	a)	1	11	0	1
%1100XXXXX0XX1XXX	the push instruction	0	1	a)	1	11	1	0
%1100XXXXX1XXXXXX	other unary operations	0	1	a)	1	11	0	1
%1101XXXXXXXXXXXX	reg. to reg. binary operations	0	0	a)	1	11	0	1
%111XXXXXXX000XXX	the fill instruction	0	0	a)	1	11	1	0
%111XXXXXXX001XXX	move to IBR, DBR or SPR	0	0	a)	1	11	0	1
%111XXXXXXX010XXX	jump to interrupt routine	1	0	a)	0	11	1	0
%111XXXXXXX1X0XXX	return from int./subroutine	0	0	a)	0	11	0	1
%111XXXXXXX111XXX	the nop instruction	0	0	000	0	11	0	1

note a) : bits 5..3 of instruction word ie.
note b) : bit 3 of instruction word ie.
note c) : inverted bit 3 of instruction word ie.
note d) : ((ie from: 4 to: 5)~=%00),((ie from: 3 to: 5)=%001)

table 5.17. The logic block exctrl controls the functions of 3 other logic blocks.

5.5 The Write-back stage.

The Write-back stage is the last stage in the pipeline and contains only a few logic blocks. Logic block *flgen* generates the flags. Logic block *flctrl* controls that block.
At this pipeline stage the decision for a conditional branch is made. This is done by logic block *brlogic*. It generates the *br* signal. The IDaSS model is shown as figure 5.14.

5.5.1 The logic block flgen.

This logic block determines the flags from the ALU and memory outputs. There are 4 flag bits:

- flag bit 0: all zero
- flag bit 1: all set
- flag bit 2: smaller
- flag bit 3: greater

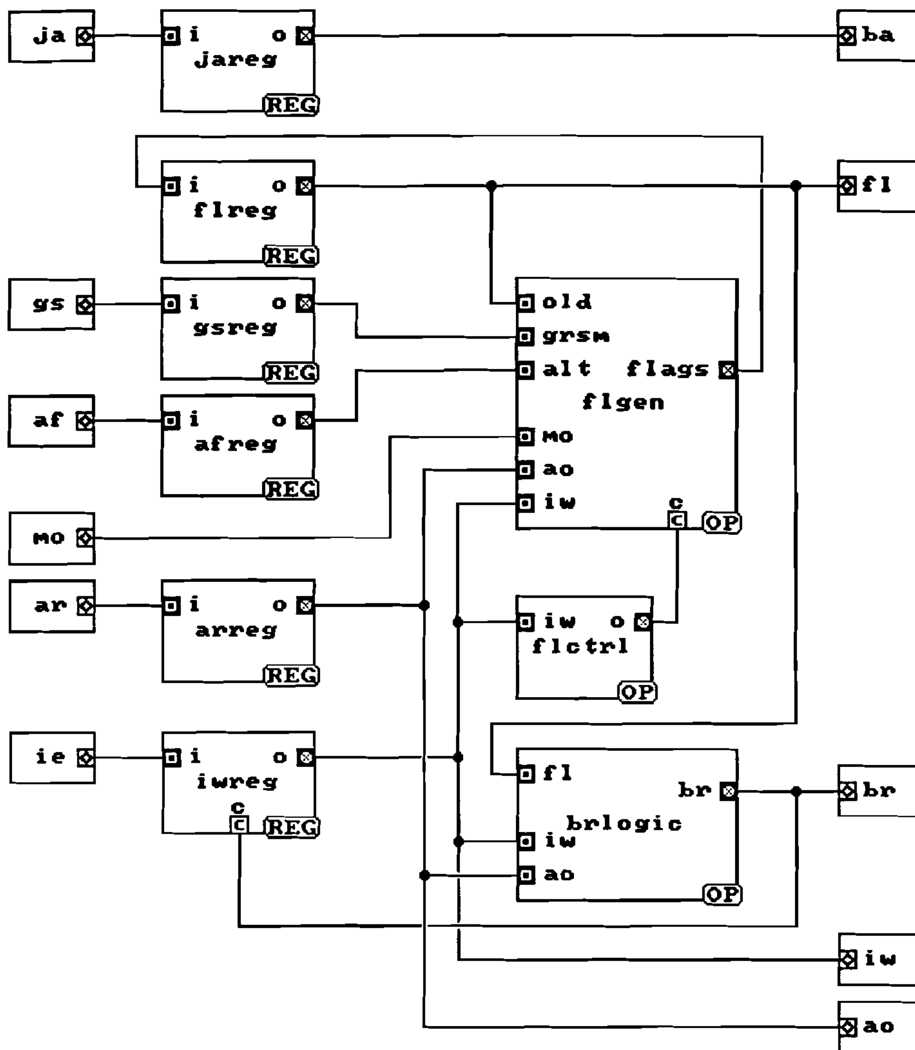


figure 5.14. The IDaSS model of the Write-back stage.

function	control value	
normal	00	generate the new flags if necessary
readmem	01	the 'all zero' and 'all set' flags can change on memory data
restore	10	restore the flags from the stack

table 5.18. The functions of the logic block flgen.

The most simple function is the restore of data from the stack. If an interrupt occurs the *int* instruction will be inserted and will save the flags on the stack. On an *rti* instruction the flags must be restored back into the flag register *flreg*.

The 'smaller' and 'greater' flags will only be changed by a compare instruction. The new values for these flags will be determined by the ALU in the Execute stage. These flags will be piped to the Write-back stage by register *gsreg* (greater,smaller).

None of the flags will change by the instructions *jmp*, *jsr*, *rts*, *nop*, *int* and one of the conditional branch instructions, except for the *djnz* instruction. The Execute stage generates a control signal that indicates whether the flags may be changed or not. This *af* signal (alter flags) is piped to the Write-back stage by the register *afreg*.

Most flag changes will originate from ALU operations. In case of a load from memory operation, the data read from memory can also change the 'all set' and the 'all zero' flags. As there are two data sources that can change these flags for each source there is a separate function.

The function 'readmem' will be used in case data is loaded from the Data Memory. The 'greater' and 'smaller' flags will not change in this case as they can only be changed by the cmp instruction.

The function 'normal' will be used in all other cases. The 'all set' and 'all zero' flags will be determined by the ALU output which will be piped from the Execute stage to the Write-back stage by the register *arreg* (ALU result).
If the *af* signal is active the flags may be changed by the flag generator *flgen*. Otherwise the old flags from the flag register *flreg* must be kept and fed-back to the flag register.

Note: The actual flags are the ones that are already stored in the flag register. They were stored there by the preceding instruction. At the time the flags are needed, which is in case of a conditional branch instruction arrives at the Write-back stage, the flags will be stable and can be used for the branch conditions.

5.5.2 The logic block *brlogic*.

The logic block brlogic generates the *br* signal (branch). If this signal is active it indicates that all instructions in the pipeline must be flushed and that the branch address must be loaded by the Program Counter Register. The branch signal can only be activated by conditional branch instructions. Table 5.19 shows these instructions and the flag conditions on which the branch signal must be activated. The djnz instruction is also a conditional branch instruction but does not check on the flags. If the ALU result is not zero it will decrement. One could say that it then should check on the 'all zero' flag. But as the flags first need to be stored in the flag register they are only up to date one clock cycle later than the instruction that caused the flag change. So the *djnz* instruction can not check on the flags. It will have to check directly on the ALU output register.

instruction	encoding	flag conditions	
jbe	0000	(flag 3) not	below or equal
ja	0001	flag 3	above
jb	0010	flag 2	below
jae	0011	(flag 2)not	above or equal
je	0100	(flag 2)not and (flag 3)not	equal
jne	0101	flag 2 or flag 3	not equal
jz	0110	flag 0	zero
jnz	0111	(flag 0)not	not zero
js	1000	flag 1	set
jns	1001	(flag 1)not	not set

table 5.19. The conditional branch instructions and the flags they check on.

5.6 Test verification.

To test the microprocessor model a set of instructions must be run to check if each type of instruction has the proper functioning. This program must be loaded into the Read Only Memory as a hex-file. This file contains the binary codes of all the instructions of the test program.
To generate such a file ir. L.C. Benschop has also created an assembler program. With this assembler the instructions can be edited in a text file as mnemonics. The assembler checks the syntax of the text file and if correct produces the binary codes of the hex-file.

5.6.1 Testing of the instruction set.

Appendix B contains the text file with the instructions as tested. This program is only meant to test all instructions and special sequences of instructions. Appendix C contains a more realistic program. This program looks a lot like the final program to be run by the microprocessor when embedded into the data compressor chip.
At each clock cycle an instruction should be processed. But stalls and address altering instructions can diminish the instruction throughput. To check the throughput of the microprocessor a counter has been temporarily added that counts the number of *nop* instructions that passes through the Write-back stage. Another counter has been added to count the number of stalls that occurred running the program.
The result is shown in table 5.20.

number of clock cycles	number of *nop* instructions	number of stalls
17576	7991 (45%)	2412 (14%)

table 5.20. Some test results of an ordinary program.

The stalls are caused by data hazards and by the *fill* instruction. Each stall causes a *nop* instruction in the Write-back stage. Next to the stalls the *nop* instructions are basically caused by the *jmp*, *jsr*, *rts* and the conditional branch instructions that actually cause a branch.

5.6.2 Testing of the interrupt feature.

In section 5.2.3 the finite state machine controller that takes care of the interrupt mechanism has been explained. The test program of Appendix B has been used to test all possible cases the finite state machine might react to. All those cases are numbered and are shown in table 5.21. The table also contains the instruction addresses where the situations of the test occur.
If the comment line is indented it means that the processor has moved up to the next clock cycle.

5.7 Critical Path analyses.

The goal of this microprocessor is to make it fast. As long as there hasn't been a complete 'place and route' of this model on chip level the actual timing is unknown.

Each logic block modelled has its own particular signal transition delays as proposed by the simulator. These delays are bases on an inverter delay of 2 nano seconds which is somewhat unrealistic for a 0.8 nano seconds ASIC process. Some of these delays were too long and were changed to a somewhat more realistic value. These delays can also be found in Appendix A.
According to these delays the critical path of the microprocessor has been examined.

number	address	situation during the occurrence of the interrupt
1	047h	- stall
2	05Ch	- branch taken
3	06Eh	- iwrts
4	060h	- iejmp (no branch)
5	06Fh	- iejsr
6	06Eh	- ierts
7	060h	- idjmp and no iebr
8	06Fh	- idjsr (no branch)
9	06Eh	- idrts
10		- idjmp and iebr
11	0A8h	- branch taken
12	0A8h	- branch not taken
13		- no (stall, br, iwrts, iejmp, iejsr, ierts, iebr, idjmp, idjsr, idrts, idbr)
14	088h	- stall
15	0ABh	- idjmp
16	06Fh	- idjsr
17	06Dh	- idrts
18		- idbr
19	05Bh	- branch taken
20	05Bh	- branch not taken
21	008h	- no (stall, idjmp, idjsr, idrts, idbr)
22		- idbr or iebr and no idjmp
23	088h	- stall
24	097h	- branch taken
25	088h	- idrts
26	08Dh	- idjsr
27	05Fh	- idjmp and not iebr
28		- idjmp and iebr
29	0A7h	- branch taken
30	0A7h	- branch not taken
31	09Ch	- no (stall, branch, iebr, idjmp, idjsr, idrts, idbr)
32		- idbr or iebr and no idjmp
33	096h	- branch taken
34	09Bh	- no iebr
35		- iebr
36	085h	- branch taken
37	084h	- branch not taken

table 5.21. The several different situations during the occurrence of an interrupt.

Figures 5.15, 5.16, 5.17 and 5.19 show the IDaSS models of the four pipeline stages.

The input names of the logic blocks have been replaced by the transition delays and are given in nano seconds. If a logic block has two outputs the transition delay for each output is given separated by a comma. In that case the left delay is for the top output and the delay behind the comma is for the lower output.
On a clock all registers (except for the register files) refresh their output values after 10 nano seconds. The signal changes ripple through the rest of each pipeline stage. The 'worst case' timing of each signal is also shown in the figures.

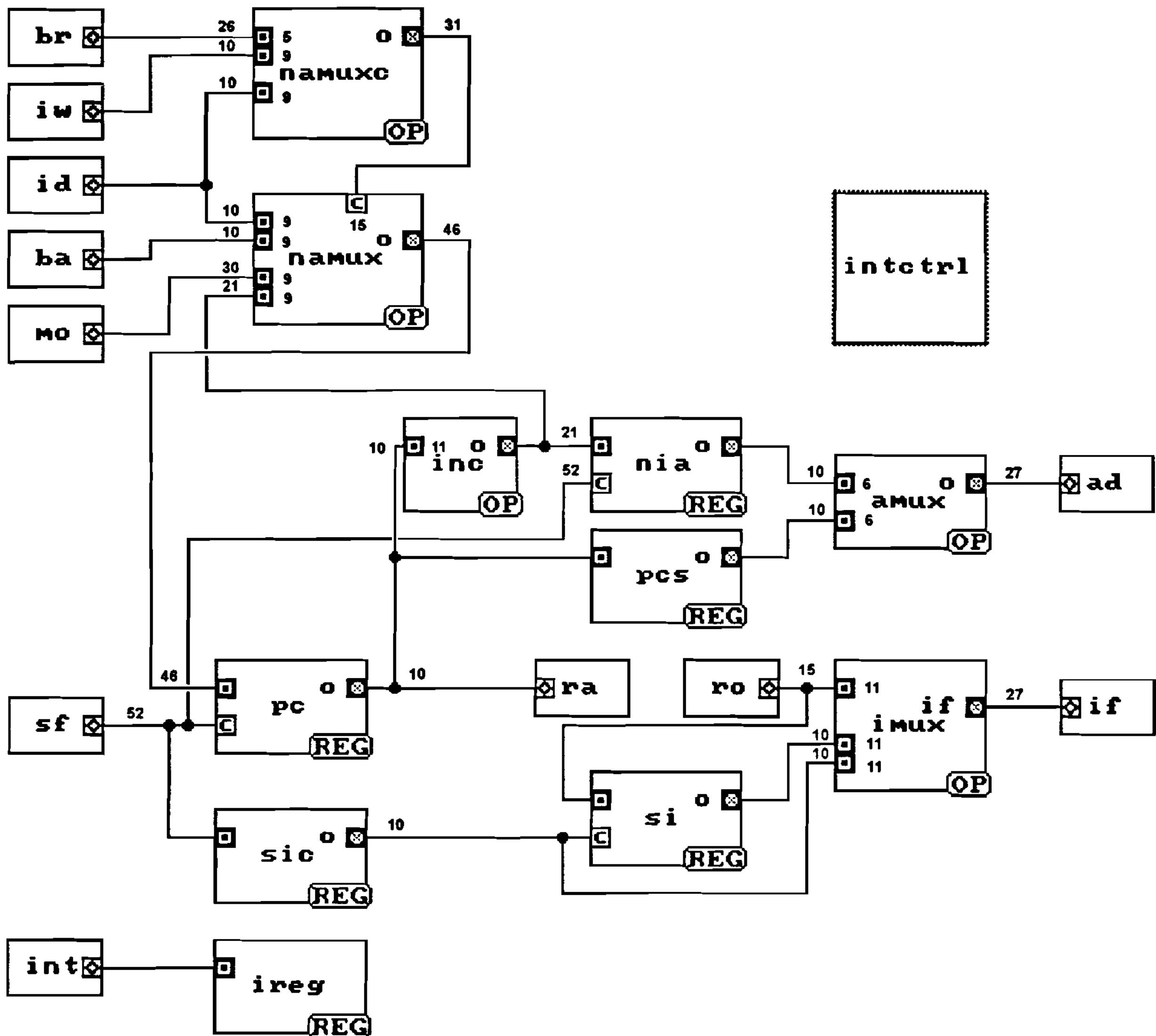


figure 5.15. The critical path analyses of the Fetch stage.

According to these delays and signal changes the critical path can be examined. The signals need to be stable at a specific time before the next clock. This is called the 'setup time' for the clocked device. The setup time for the ordinary pipeline registers is 12 nano seconds. If a signal changes 30 nano seconds before the next clock it means there is 18 nano seconds of 'spare time' before the setup time has been reached. The smallest of all these 'spare times' determines how much faster the clock can be made. The setup time for the register files is 16 nano seconds for the data and 12 nano seconds for the address. The setup time for the finite state machines is 35 nano seconds.

For the simulation a clock frequency of 100 nano seconds was used.

For each of the pipeline stages the critical path has been examined. The 'fastest' stage is the Write-back stage. The branch signal appears 26 nano seconds at the latest after the clock. The last possible signal change appears 39 nano seconds after the clock. This leaves 61 nano seconds of 'spare time'. This is shown in figure 5.19.

figure 5.16. The critical path analyses of the Decode stage.

In the Decode stage the latest signal change of signals for the pipeline registers appears at 59 nano seconds after the clock. The setup time is 12 nano seconds. This leaves 29 nano seconds of 'spare time' for those registers.
The setup time of the destination input of the register files is 12 nano seconds. The latest signal change for those inputs is 57 nano seconds after the clock.

The setup time for the data inputs of the register files is 16 nano seconds. The latest signal change for those inputs is 65 nano seconds after the clock. This leaves a 'spare time' of 19 nano seconds for the register files.
The setup time of the finite state machine is 35 nano seconds. The finite state machine *idctrl* uses the contents of *idreg* and the signals *sf* and *br*. The last that possibly changes its value is sf at 52 nano seconds after the clock. This leaves a 'spare time' of 13 nano seconds. So the overall 'spare time' for the Decode stage is 13 nano seconds.

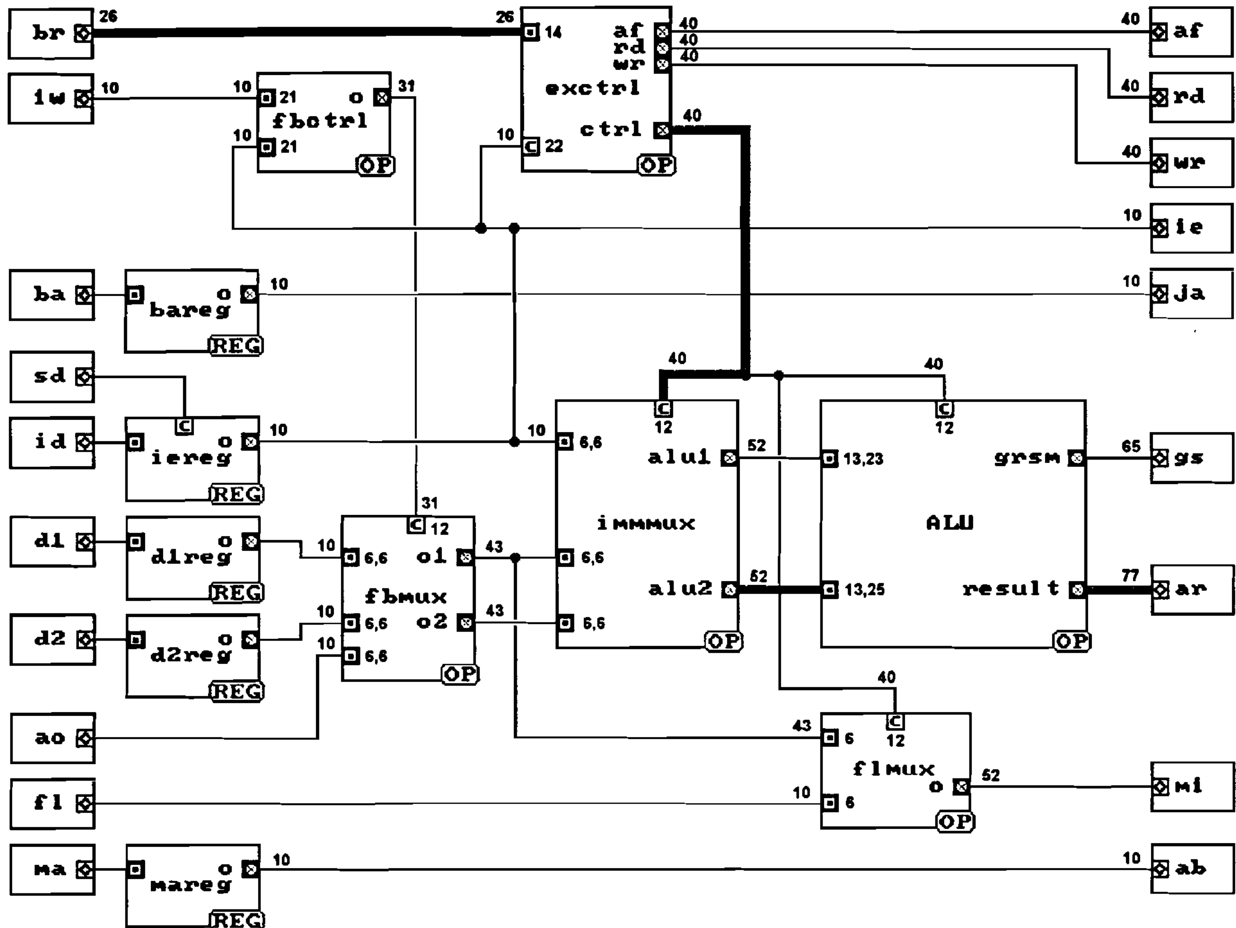


figure 5.17. The critical path analyses of the Execute stage.

In the Execute stage the latest signal change of signals for the pipeline registers appears at 77 nano seconds after the clock. The setup time is 12 nano seconds. This leaves 11 nano seconds of 'spare time' for those registers.
The path that causes this delay is not realistic. It is marked in figure 5.17 with a thick line. If a branch occurs the *ALU* will switch to the *mov* operation which means it will pass the input data without any arithmetic calculations. It takes 9 nano seconds to pass the data through the *ALU* and it will be even faster than the output grsm which always has a delay of 13 nano seconds. The output grsm will then produce its output 65 nano seconds after the clock. This leaves a 'spare time' of 23 nano seconds.
If the branch signal is not active the critical path changes to that of figure 5.18. This path is far more realistic. The last signal now changes 74 nano seconds after the clock. With a setup time of 12 nano seconds this leaves 14 nano seconds of 'spare time'.
The data setup time for the external Data Memory has been modelled as 30 nano seconds. The last signal change appears 52 nano seconds after the clock. This leaves 18 nano seconds of 'spare time' for the Data memory.

As this is also not a realistic path a more realistic path would lead to a last signal change at 49 nano seconds after the clock. This is mostly caused by the feedback multiplexors of this stage.
The overall 'spare time' of the Execute stage is 14 nano seconds.

figure 5.18. A more realistic critical path for the Execute stage.

The Fetch stage holds the Program Counter Register. The data for this register will be stable 46 nano seconds after the clock. This stage also holds the Instruction Memory. The address setup time is modelled as 30 nano seconds. As this address comes directly from the Program Counter Register it has plenty of time to get stable for the next clock. The output of the memory module is stable 15 nano seconds after the clock.
The finite state machine *intctrl* is implemented as a Moore-machine. The multiplexors *amux* and *imux* which are controlled by this finite state machine will switch their functions 15 nano seconds after the clock.
The finite state machine depends on the values of the instructions from the Decode stage (*id*), the Execute stage (*ie*) and from the Write back stage (*iw*). As they all come directly from a register they all appear 10 nano seconds after the clock. Next to those instructions the finite state machine also depends on the branch signal *br* (26 nano seconds after the clock) and the stall signal *sf* (52 nano seconds after the clock).

As we have seen before with the finite state machine of the Decode stage the finite state machine has a setup time of 35 nano seconds. This means the 'spare time' here is 13 nano seconds.
The overall 'spare time' of the Fetch stage would be 13 nano seconds.

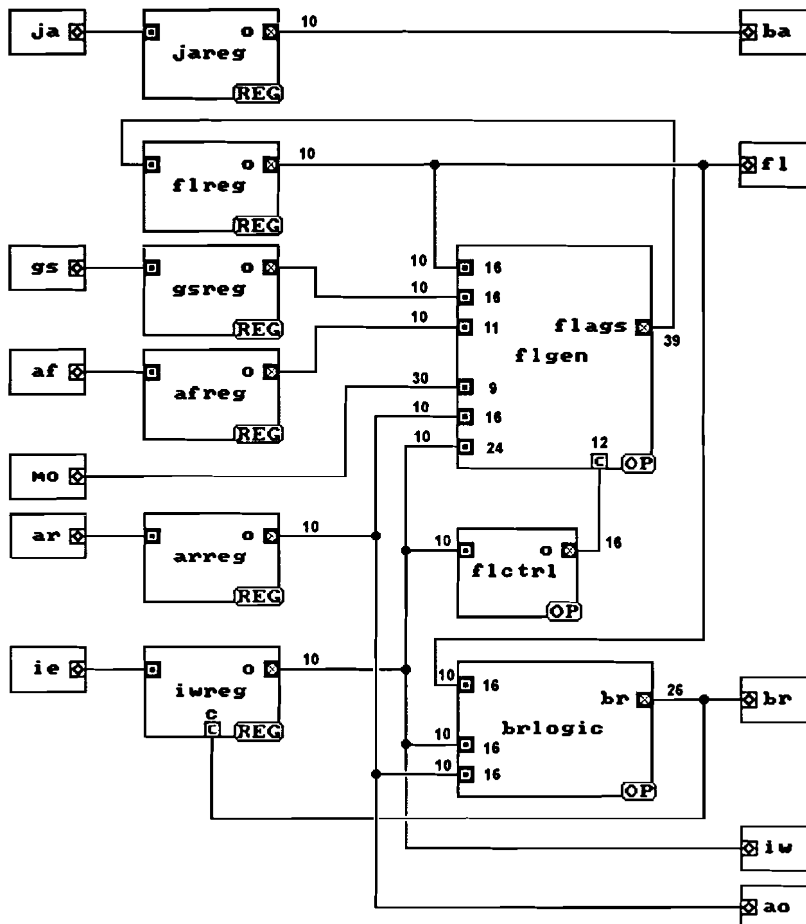


figure 5.19. The critical path of the Write-back stage.

The overall 'spare time' of all stages would be 13 nano seconds. This is caused by both the finite state machines of the Fetch stage and the Decode stage. The Execute stage follows very closely with a 'spare time' of only 14 nano seconds.
From these delays can be concluded that the pipeline stages are well balanced.

6 Conclusions and recommendations.

It is not easy designing a fast architecture for a microprocessor. The necessary steps to be taken are not very well documented. The best book that can be used when one hasn't got any experience in designing a microprocessor is [Pat94] though the example given is quite simple. When a more complex instruction set is used other microprocessor architectures need to be studied.
What was missing from any book I have seen, that explains the design of a microprocessor is the way the interrupt mechanism is implemented in the architecture. This was the hardest part to be implemented.
The memory modules used in this design are synchronous which means they become part of the pipeline registers. This makes the design somewhat more complex. In spite of being clocked the data coming from the memories still have a relatively long delay. Therefore this data can not be fed back into other pipeline stages to have those stages perform time consuming (arithmetic) operations on them.

The design can now be converted to VHDL and this can then be used by a silicon compiler to generate a placement and routing on chip level. The results from that 'place and route' will give more accurate timing information. With this information another critical path analyses can be done and modifications might be necessary.

A modification can probably be made to the finite state machine of the Fetch stage. If turned into a Mealy-machine it can probably become a little bit less complex. Or if the 'next address incrementer' *inc* will be provided with an extra function *pass* it might be possible to lose the register *pcs* and use the register *nia* instead.
As both finite state machines from the Fetch stage and from the Decode stage influence the data of the instruction register of the Decode stage it might be possible to combine both finite state machines into one new one.
The feedback controller *fbctrl* of the Execute stage determines a great deal of the delay of that stage. As the signal generated by this controller only depends on the instructions of the Execute stage and the Write-back stage it must be possible to generate this signal earlier in the Decode stage. Note however that the logic becomes more complex as the branch signal *br* and the stall signal *sd* can change the instructions of the pipeline stages on the next clock.
The stall signal sf determines the critical path of the Decode stage. This critical path is also determined by the feedback controller. Here it will be even harder to generate the control signals for the feedback multiplexor in an earlier stage.

Bibliographies

[FLY95] Flynn, Michael J.
Computer Architecture: Pipelined and parallel processor design.
London: Jones and Bartlett, 1995

[HEU92] Heudin, J.C. and C. Panetto
RISC Architectures.
London: Chapman & Hall, 1992
translated from French: Les Architectures RISC.
Paris: Dunod Editeur, 1990

[KOG81] Kogge, Peter M.
The architecture of pipelined computers.
Washington: Hemisphere, 1981

[PAT90] Patterson, David A. and John L. Hennessy
Computer Architecture: A Quantitative Approach (second edition).
San Fransisco: Morgan Kaufmann, 1990

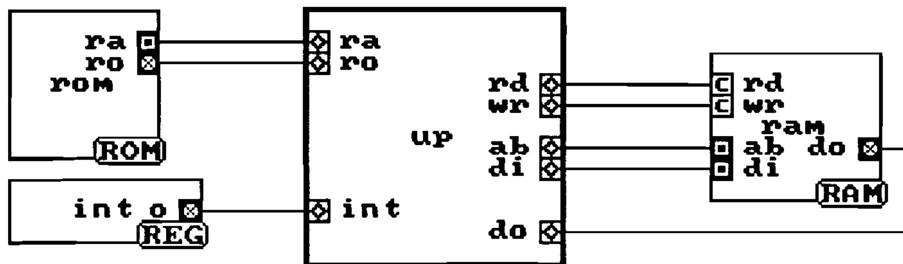
[PAT94] Patterson, David A. and John L. Hennessy
Computer Architecture: The Hardware/Software interface.
San Mateo: Morgan Kaufmann, 1994

[VER90] Verschueren, A.C.
IDaSS for ULSI (IDaSS manual).
Section of Digital Information Systems, Faculty of Electrical Engineering,
Eindhoven University of Technology, March 1990

Appendix A.

The IDaSS models of the microprocessor

Part 1. The direct environment of the embedded microprocessor.



The top level of the embedded microprocessor consists of the unit **up** and the instruction memory **rom**. The instruction memory is placed external though it is part of the microprocessor. When translating the microprocessor to VHDL this memory need not be translated.
The data memory **ram** is modelled here as one block but actually consists of several separated banks of memory and memory mapped I/O. This is not part of the microprocessor but is added for testing purposes.
A register **int** is added to simulate the interrupt signal. This register is only added for testing purposes.

<u>Blocks</u>

int : register
Purpose: interrupt generation; This register can simulate an interrupt signal.
The meaning of the **int** bit is shown in the following table:

int	Meaning
'0'	no interrupt
'1'	interrupt

Inputs: none.
Outputs: 'o', 1 bit.
Reset value: 0

ram : RAM
Purpose: data memory; This RAM will hold the data to be processed.
Inputs: 'ab', 12 bits; RAM address.
'di', 16 bits; data input.
'rd', 1 bit; the control connector for the read signal.
'wr', 1 bit; the control connector for the write signal.
Outputs: 'ro', 16 bits; data output.

Size: 4096 x 16 bits
Technology: IDaSS default technology.

Control specification for control input 'rd':

%1 read

System-defined timing for control input 'rd':
Bus to command delay: 6n sec.

Control specification for control input 'wr':

%1 write

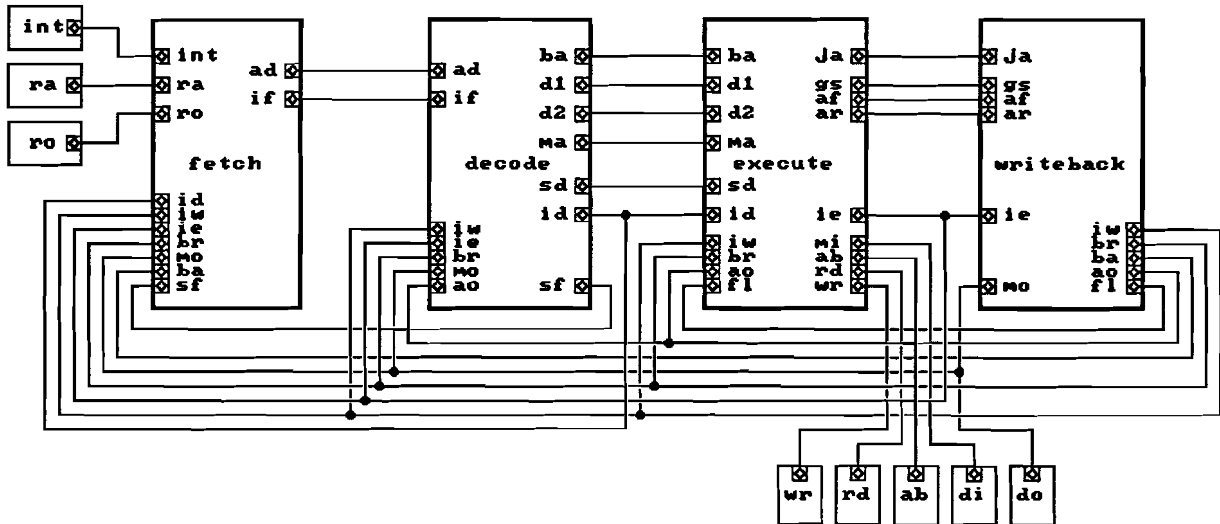
System-defined timing for control input 'wr':
Bus to command delay: 6n sec.

Delays:
Addr. to output delay (async): 25n sec.
Clock to (addr.) output delay: 20n sec.
Port address setup time: 20n sec.
Port command setup time: 18n sec.
Clock to (fixed) output delay: 15n sec.
Write data setup time: 30n sec.

rom : ROM
Purpose: program ROM; It contains the instructions that must be executed. It's default function is 'read'.
Inputs: 'ra', 12 bits; ROM address.
Outputs: 'ro', 16 bits; ROM output.
Size: 4096 x 16 bits
Technology: ASA synchronous read only memory
Delays:
Addr. to output delay (async): 35n sec.
Clock to (addr.) output delay: 15n sec.
Port address setup time: 30n sec.
Port command setup time: 20n sec.

up : schematic
Purpose: the actual microprocessor. For a more detailed description see part 2.

Part 2. Description of the schematic up.



Connectors

name	I/O	bits	description
ab	O	12	address bus; generated by the execute block.
di	O	16	data in; data to be stored in the RAM; generated by the execute block.
do	I	16	data out; data loaded from the RAM; generated by the external RAM.
int	I	1	interrupt signal.
ra	O	12	ROM address; generated by the fetch block.
rd	O	1	the read signal for the external RAM; generated by the execute block.
ro	I	16	ROM output; the instruction read from the ROM; generated by the ROM.
wr	O	1	the write signal for the external RAM; generated by the execute block.

Blocks

decode : schematic
Purpose : The pipeline stage named decode. This stage receives the instruction from the fetch stage and decodes the instruction.

execute : schematic
Purpose : The pipeline stage named execute. This stage operates on the data generated by the decode stage. The result is written to the external RAM or piped to the writeback-stage.

fetch : schematic
Purpose : The pipeline stage named fetch. This stage generates the next instruction address and receives the instruction from the ROM. This instruction is piped to the decode stage. This pipeline stage also controls the interrupt mechanism.

writeback : schematic
Purpose : The pipeline stage named writeback. This stage writes the result from the execute stage back to the registers of the decode stage. The flags are modified and the branch signal is generated by this stage.

Part 2.1. The description of the schematic fetch.

Connectors

name	I/O	bits	description
ad	O	12	the address of the next instruction.
ba	I	12	branch address; the address to jump to in case of a conditional branch.
br	I	1	the branch signal.
id	I	16	the instruction present at the decode stage.
ie	I	16	the instruction present in the execute stage.
if	O	16	the instruction generated by the fetch stage.
int	I	1	the interrupt signal.
iw	I	16	the instruction present in the writeback stage.
mo	I	16	memory out; the output of the RAM.
ra	O	12	ROM address
ro	I	16	ROM output; this is the instruction fetched from the ROM.
sf	I	1	the stall signal for the fetch stage.

Blocks

amux : operator

Purpose: address multiplexor; This multiplexor switches between the 'next instruction address register' (nia) and the 'program counter save register' (pcs). The output of this multiplexor will be piped to the decode stage. It can be saved on stack or used to calculate the branch address. The functionality is shown in the following table. Switching between the two functions is controlled by the finite state machine controller **intctrl**.

function	
nia	select input nia (next instruction address)
pcs	select input pcs (program counter save)

Inputs: 'nia', 12 bits; next instruction address.
'pcs', 12 bits; program counter save.
Outputs: 'o', 12 bits; output of multiplexor.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined output multiplexor delays:
For output 'o': 4n sec

Text for function nia:

```
"select nia-register output"

o := nia
```

Text for function pcs:

```
"select pcsave-register"

o := pcs
```

imux : operator

Purpose: instruction multiplexor; This multiplexor has three functions which are controlled by the finite state machine controller **intctrl.** One of these functions is the generating of the *nop* instruction. Another is the generating of the *int* instruction. The third function is named *normal* and is an ordinary multiplexor function which switches between the two inputs 'rom' and 'si'. Switching between these two inputs is controlled by the input signal 'sic'. This is shown in the following table.

function	sic signal	output value
int	-	value FFD7h
nop	-	value FFFFh
normal	0	from input rom
	1	from input si

Inputs: 'rom', 16 bits; instruction from ROM.
'si', 16 bits; saved instruction.
'sic', 1 bit; control signal to select between input 'rom' and input 'si'.
Outputs: 'if', 16 bits; instruction generated by the fetch stage.

IDaSS description:

System-defined timing for output 'if':
Data transfer delay: 2n sec

System-defined output multiplexor delays:
For output 'if': 4n sec

Text for function 'int':

```
"call interrupt routine"

if := %1111111111010111
```

Text for function 'nop':

```
"insert a nop instruction"

if := %1111111111111111
```

Text for function 'normal':

```
"no interrupt call"

if := sic if0: rom
          if1: si
```

Internal time delays for function 'normal':
From 'rom' to 'if': 5n sec
From 'si' to 'if': 5n sec
From 'sic' to 'if': 5n sec

inc : operator
Purpose: incrementor; This operator increments its input value.
Inputs: 'i', 12 bits.
Outputs: 'o', 12 bits.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'increment':

```
"increment the input"

o := i + 1
```

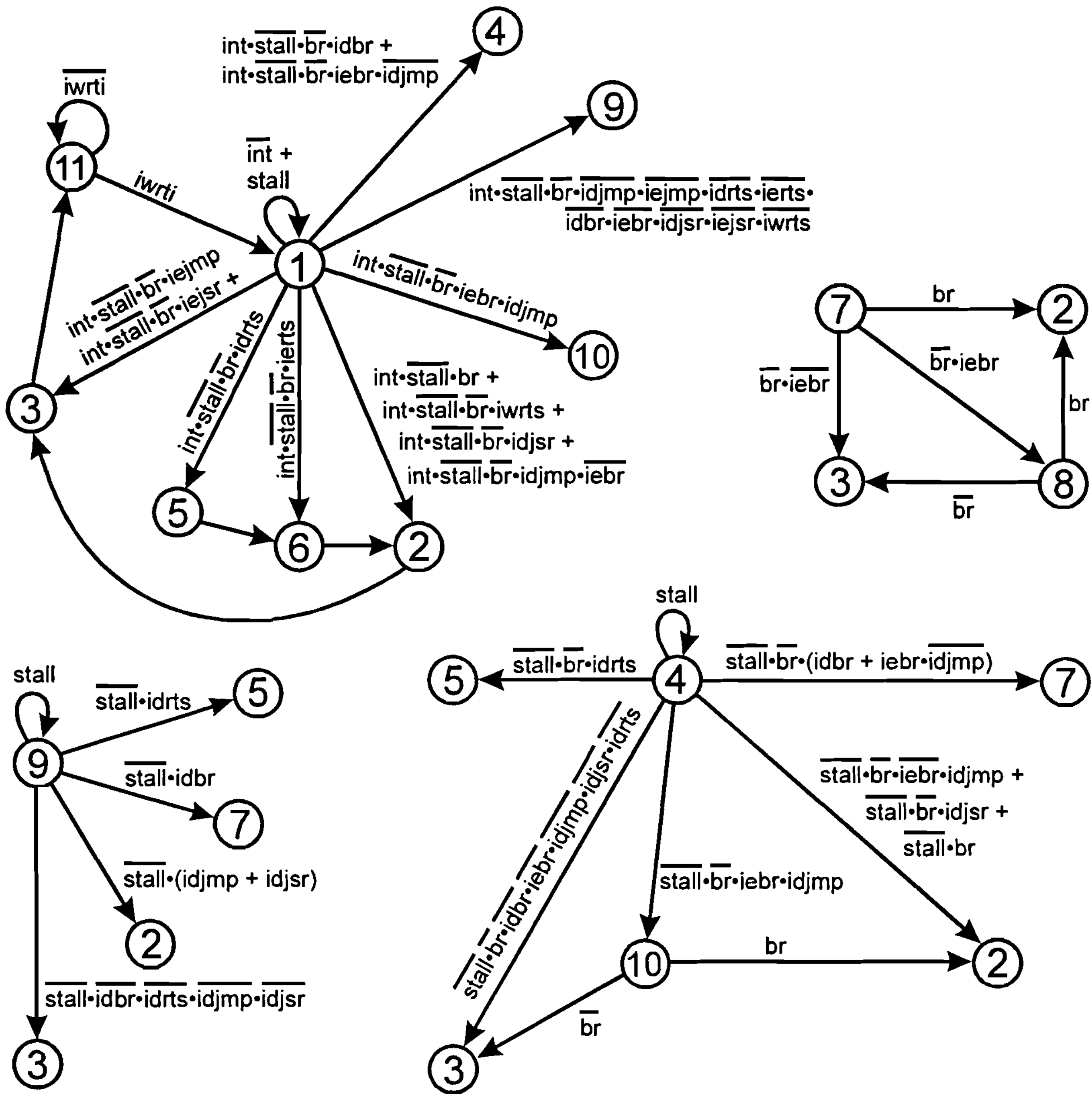
Internal time delays for function 'increment':
From 'i' to 'o': 11n sec

intctrl : state machine controller

Purpose: interrupt controller; In case of an interrupt it takes care of saving the right return address on the stack and activates the interrupt routine at the proper time.
This finite state machine controller controls the logic blocks **amux**, **imux** and **pcs**. The controller consists of 11 states. The function of the controlled blocks is shown for each of the states in the following table.

state	amux	imux	pcs
1	nia	normal	load
2	pcs	nop	load
3	pcs	int	load
4	pcs	nop	hold
5	pcs	nop	load
6	pcs	nop	load
7	pcs	nop	hold
8	pcs	nop	hold
9	pcs	nop	hold
10	pcs	nop	load
11	nia	normal	load

In the following figure the state transition diagram is shown. The condition 'idjmp' means that there is a *jmp* instruction in the decode stage (ie: execute stage; iw: writeback stage). The condition 'idbr' means that there is a conditional branch instruction in the decode stage.



A short description of each state:

State 1:
This is the initial state where it will remain as long as there is no interrupt signal. If an interrupt signal appears at the time of a stall, no state change will occur until the stall signal is passive again.

State 2:
State 2 inserts a *nop* instruction to enable the right flags to be saved on the stack.

State 3:
In this state the *int* instruction is generated.

State 4:
This stage will be entered if the decode stage contains a conditional branch instruction or the execute stage contains a conditional branch instruction without a *jmp* instruction in the decode stage. The **pcs** register holds the return address in case the branch is not taken.

State 5:
If the decode stage contains an *rts* instruction the pipe must be drained and *nop*s are inserted to retrieve the return address from the stack. This address will be saved on the stack again as the return address of the interrupt routine.

State 6:
If the execute stage contains an *rts* instruction (the decode stage will always contain a *nop* instruction in this case) the pipe must be drained and *nop*s are inserted to retrieve the return address from the stack. This address will be saved on the stack again as the return address of the interrupt routine.

State 7:
If a branch occurs the branch address must be saved as the return address and it takes another 'wait state' (state 2, a *nop* instruction is inserted) to let the program counter load the return address. If no branch occurs and there is no conditional branch instruction in the execute stage the interrupt instruction can be started (state 3). But if the execute stage does hold a conditional branch instruction it must first be evaluated (state 8).

State 8:
If a branch occurs the branch address must be saved as the return address and it takes another 'wait state' (state 2, a *nop* instruction is inserted) to let the program counter load the return address. If no branch occurs and there is no conditional branch instruction in the execute stage the interrupt instruction can be started (state 3).

State 9:
This state is entered if there are no instructions in the decode, execute and writeback stage that can change the program counter contents. At this state the decode stage is loaded with an instruction which was already fetched from the instruction memory. This instruction must be examined to see if it can change the 'normal' program flow (can change the program counter contents: conditional branches, jumps, etc.).

State 10:
This state is entered if the decode stage contains a jmp instruction and the execute stage contains a conditional branch instruction. At this state the conditional branch instruction has moved to the writeback stage and it will be clear if the branch is to be taken (state 2 is next state) or not (state 3 is next state).

State 11:
As this microprocessor does not allow for nested interrupts, this state will not be left until the *rti* instruction is executed.

The IDaSS description:

This state machine controller has 11 states.

No stack is available for 'subroutine' calls.
This controller is enabled following system reset.

This state machine controller has no connectors.

System-defined timing for this state machine controller:
Clock to state delay: 15n sec
State/test to command delay: 10n sec
Test to clock setup time: 25n sec

Text for state number 1 (reset state):

```
"state1 is the initial state"

state1:
 imux normal;
 pcs load;
 amux nia;
 [ sf "is there a stall"
 | %1 -> state1
 | %0
  [ ireg "is there an interrupt"
  | %0 -> state1
  | %1
   [ br "is there an active branch"
   | %1 -> state2
   | %0
    [ _idbr := ((id from: 13 to: 15)=%101).
      _idbr "is there a branch instruction in id"
    | %1 -> state4
    | %0
     [ _iejump := ((ie from: 13 to: 15)=%100).
       _iejump "does ie contain a jsr or jmp instruction"
     | %1 -> state3
     | %0
      [ _idjsr := ((id from: 12 to: 15)=%1001).
        _idjsr "does id contain a jsr instruction"
      | %1 -> state2
      | %0
       [ _iwrts := (((iw from: 13 to: 15)=%111) Λ
                   ((iw from: 3 to: 5)=%110)).
         _iwrts "does iw contain an rts instruction"
       | %1 -> state2
       | %0
        [ _ierts := (((ie from: 13 to: 15)=%111) Λ
                    ((ie from: 3 to: 5)=%110)).
          _ierts "does ie contain an rts instruction"
        | %1 -> state6
        | %0
         [ _idrts := (((id from: 13 to: 15)=%111) Λ
                     ((id from: 3 to: 5)=%110)).
```

```
            _idrts "does id contain an rts instruction"
           | %1 -> state5
           | %0
            [ _iebr := ((ie from: 13 to: 15)=%101).
              _iebr "is there a branch instruction in ie"
            | %1
              [ _idjmp := ((id from: 12 to: 15)=%1000).
                _idjmp "does id contain a jmp instruction"
              | %1 -> state10
              | %0 -> state4
              ]
            | %0
              [ _idjmp := ((id from: 12 to: 15)=%1000).
                _idjmp "does id contain a jmp instruction"
              | %1 -> state2
              | %0 -> state9
              ]
            ]
           ]
          ]
         ]
        ]
       ]
      ]
     ]
    ]
   ]
```

Internal time delays for this state:
From block 'id' to '_idbr' in test #4: 6n sec
From block 'ie' to '_iejump' in test #5: 6n sec
From block 'id' to '_idjsr' in test #6: 6n sec
From block 'iw' to '_iwrts' in test #7: 9n sec
From block 'ie' to '_ierts' in test #8: 9n sec
From block 'id' to '_idrts' in test #9: 9n sec
From block 'ie' to '_iebr' in test #10: 6n sec
From block 'id' to '_idjmp' in test #11: 6n sec
From block 'id' to '_idjmp' in test #12: 6n sec

Text for state number 2:

```
"This is state 2"

state2:
 imux nop;
 pcs load;
 amux pcs;
 -> state3
```

Text for state number 3:

```
"this is state 3"

state3:
 imux int;
 pcs load;
 amux pcs;
 -> state11
```

Text for state number 4:

```
"This is state 4"

state4:
 imux nop;
 pcs hold;
 amux pcs;
 [ sf "is there a stall"
 | %1 -> state4
 | %0
   [ br "is there a branch to be taken"
   | %1 -> state2
   | %0
     [ _idbr := ((id from: 13 to: 15)=%101).
       _idbr "is there a branch instruction in id"
     | %1 -> state7
     | %0
       [ _idjsr := ((id from: 12 to: 15)=%1001).
         _idjsr "does id contain a jsr instruction"
       | %1 -> state2
       | %0
         [ _idrts := (((id from: 13 to: 15)=%111) Λ
                     ((id from: 3 to: 5)=%110)).
           _idrts "does id contain an rts instruction"
         | %1 -> state5
         | %0
           [ _iebr := ((ie from: 13 to: 15)=%101).
             _iebr "is there a branch instruction in ie"
           | %1
             [_idjmp := ((id from: 12 to: 15)=%1000).
              _idjmp "does id contain a jmp instruction"
             | %1 -> state10
             | %0 -> state7
             ]
           | %0
             [_idjmp := ((id from: 12 to: 15)=%1000).
              _idjmp "does id contain a jmp instruction"
             | %1 -> state2
             | %0 -> state3
             ]
```

```
          ]
        ]
      ]
    ]
  ]
]
```

Internal time delays for this state:
From block 'id' to '_idbr' in test #3: 6n sec
From block 'id' to '_idjsr' in test #4: 6n sec
From block 'id' to '_idrts' in test #5: 9n sec
From block 'ie' to '_iebr' in test #6: 6n sec
From block 'id' to '_idjmp' in test #7: 6n sec
From block 'id' to '_idjmp' in test #8: 6n sec

Text for state number 5:

```
"This is state 5"

state5:
 imux nop;
 pcs load;
 amux pcs;
 -> state6
```

Text for state number 6:

```
"This is state 6"

state6:
 imux nop;
 pcs load;
 amux pcs;
 -> state2
```

Text for state number 7:

```
"This is state 7"

state7:
 imux nop;
 pcs hold;
 amux pcs;
 [ br "is there a branch"
 | %1 -> state2
 | %0
   [ _iebr := ((ie from: 13 to: 15)=%101).
     _iebr "does ie contain a conditional branch instruction"
   | %1 -> state8
   | %0 -> state3
   ]
]
```

Internal time delays for this state:
From block 'ie' to '_iebr' in test #2: 6n sec

Text for state number 8:

```
"This is state 8"

state8:
 imux nop;
 pcs hold;
 amux pcs;
 [ br "is there a branch"
 | %1 -> state2
 | %0 -> state10
 ]
```

Text for state number 9:

```
"This is state 9"

state9:
 imux nop;
 pcs hold;
 amux pcs;
 [ sf "is there a stall"
 | %1 -> state9
 | %0
   [ _idjump := ((id from: 13 to: 15)=%100).
     _idjump "does id contain a jmp or jsr instruction"
   | %1 -> state2
   | %0
     [ _idbr := ((id from: 13 to: 15)=%101).
       _idbr "does id contain a conditional branch instruction"
     | %1 -> state7
     | %0
       [ _idrts := (((id from: 13 to: 15)=%111) Λ
                   ((id from: 3 to: 5)=%110)).
         _idrts "does id contain an rts instruction"
       | %1 -> state5
       | %0 -> state3
       ]
     ]
   ]
 ]
```

Internal time delays for this state:
From block 'id' to '_idjump' in test #2: 6n sec
From block 'id' to '_idbr' in test #3: 6n sec
From block 'id' to '_idrts' in test #4: 9n sec

Text for state number 10:

```
"This is state 10"

state10:
 imux nop;
 pcs load;
 amux pcs;
 [ br "is there a branch"
 | %1 -> state2
 | %0 -> state3
 ]
```

Text for state number 11:

```
"This is state 11, wait until the interrupt"
"service routine has ended."

state11:
 imux normal;
 pcs load;
 amux nia;
 [ _iwrti := ((iw from: 13 to: 15)=%111) Λ
          ((iw from: 3 to: 5)=%100).
   _iwrti "is the interrupt routine finished"
 | %0 -> state11
 | %1 -> state1
 ]
```

Internal time delays for this state:
 From block 'iw' to '_iwrti' in test #1: 9n sec

ireg : register

Purpose: interrupt register; This register first synchronises the external interrupt. The value of the register contents is read by the state machine controller **intctrl**.

Inputs: 'i', 1 bit.
Outputs: none.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for this register:
 Clock to data delay: 8n sec
 Clock to status delay: 7n sec
 Data setup time: 12n sec
 Command setup time: 7n sec

namux : operator

Purpose: next address multiplexor; A 5-input multiplexor which is controlled by **namuxc**. One of the inputs is generated internally. The following table shows the functions of this block.

function	control value	select input	
sela4	000	a4	load incremented program counter
sela1	001	a1	load jump address
sela3	010	a3	load return address
sela2	011	a2	load branch address
selint	1xx	value 001h	load interrupt start address 001h

Inputs: 'a1', 16 bits, only 12 bits are used; the jump address.
'a2', 12 bits; the branch address.
'a3', 16 bits, only 12 bits are used; the return address.
'a4', 12 bits; the incremented program counter.
'c', 3 bits; the control input connector.

Outputs 'o', 12 bits.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o': 7n sec

Control specification:

```
"select an input of the multiplexor"

%000 sela4.
%001 sela1.
%010 sela3.
%011 sela2.
%100 selint
```

System-defined timing for the control input 'c':
Bus to command delay: 6n sec

Text for function 'sela1':

```
"select input a1: the jump address"

o := a1 from: 0 to: 11
```

Text for function 'sela2':

```
"select input a2: branch address"

o := a2
```

Text for function 'sela3':

```
"select input a3: return address"

o := a3 from: 0 to: 11
```

Text for function 'sela4':

```
"select input a4: normal incrementation"

o := a4
```

Text for function 'selint':

```
"generate the interrupt routine start address (001h)"

o := 1
```

namuxc : operator

Purpose: next address multiplexor controller; This operator controls the multiplexor **namux**.

Inputs: 'br', 1 bit; the branch signal.
'id', 16 bits; the instruction in the decode stage.
'iw', 16 bits; the instruction in the writeback stage.

Outputs: 'o', 3 bits;

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'doit':

```
"this operator controls the next program counter"
"address multiplexer."
"c(000) = a4 : normal incrementation"
"c(001) = a1 : load jump address"
"c(010) = a3 : load return address"
"c(011) = a2 : load branch address"
"c(100)      : load interrupt start address (001h)"

"_jmp is 1 in case id is a jsr or jmp instruction"
_jmp := ((id from: 13 to: 15) = %100).
"_ret is 1 if iw is a rts or rti instruction"
_ret := ((iw from: 13 to: 15) = %111) Λ (iw at: 5) Λ ((iw at: 3)not).
"_int is 1 in case id is an interrupt call"
_int := ((id from: 13 to: 15) = %111) Λ ((id from: 3 to: 5)=%010).
"note: execution of rts or rti inserts 4 nops in the decode stage."
"So _jmp and _ret can not be active at the same time"
"_int can not be active together with a branch, _jmp or _rts"
```

```
o := _int,(br V _ret),(br V _jmp)
```

Internal time delays for function 'doit':
 From 'id' to '_jmp': 6n sec
 From 'iw' to '_ret': 12n sec
 From 'id' to '_int': 9n sec
 From '_jmp' to 'o': 3n sec
 From '_ret' to 'o': 3n sec
 From 'br' to 'o': 3n sec

nia : register

Purpose: next instruction address; As the instruction memory is a synchronous ROM the 'next instruction address' that accompanies the instruction must be piped. This is done by this register.

Inputs: 'i', 12 bits;
'c', 1 bit; the control connector
Outputs: 'o', 12 bits;

IDaSS description:

Default function: 'load'
Reset value: 0

Control specification:

```
"hold when stalled"

%0 load.
%1 hold
```

System-defined timing for control input 'c':
 Bus to command delay: 6n sec

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined timing for this register:
 Clock to data delay: 8n sec
 Clock to status delay: 7n sec
 Data setup time: 12n sec
 Command setup time: 7n sec

pc : register

Purpose: program counter register.

Inputs: 'i', 12 bits.
'c', 1 bit; the control connector.
Outputs: 'o', 12 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

Control specification:

```
"hold when stalled"

%0 load.
%1 hold
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

pcs : register
Purpose: program counter save register; In some cases, when an interrupt occurs, it is necessary to save the contents of the program counter. It will then be saved in this register. This register is controlled by the state machine controller **intctrl**.

Input: 'i', 12 bits.
Outputs: 'o', 12 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

si : register

Purpose: save instruction; In case of a stall the instruction memory can not 'hold' its current output. To prevent the data to be lost on the next clock the data is saved in this register. After a stall the multiplexor **imux** selects the output of this register to generate the next instruction. This register is controlled by **sic**.

Inputs: 'I', 16 bits.
'c', 1 bit; the control connector.
Outputs: 'o', 16 bits.

IDaSS description:

Default function: 'hold'
Reset value: 65535 (0FFFFh)

Control specification:

```
"this signal controls the 'save instruction' register"

%0 load.
%1 hold
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

sic : register

Purpose: save instruction controller; In case of a stall the output of the instruction memory must be saved. During the first clockcycle of a stall, the 'hold' control signal must be postponed one clock cycle to enable the **si** register to load the output of the instruction memory.

Inputs: 'I', 1 bit.
Outputs: 'o', 1 bit.

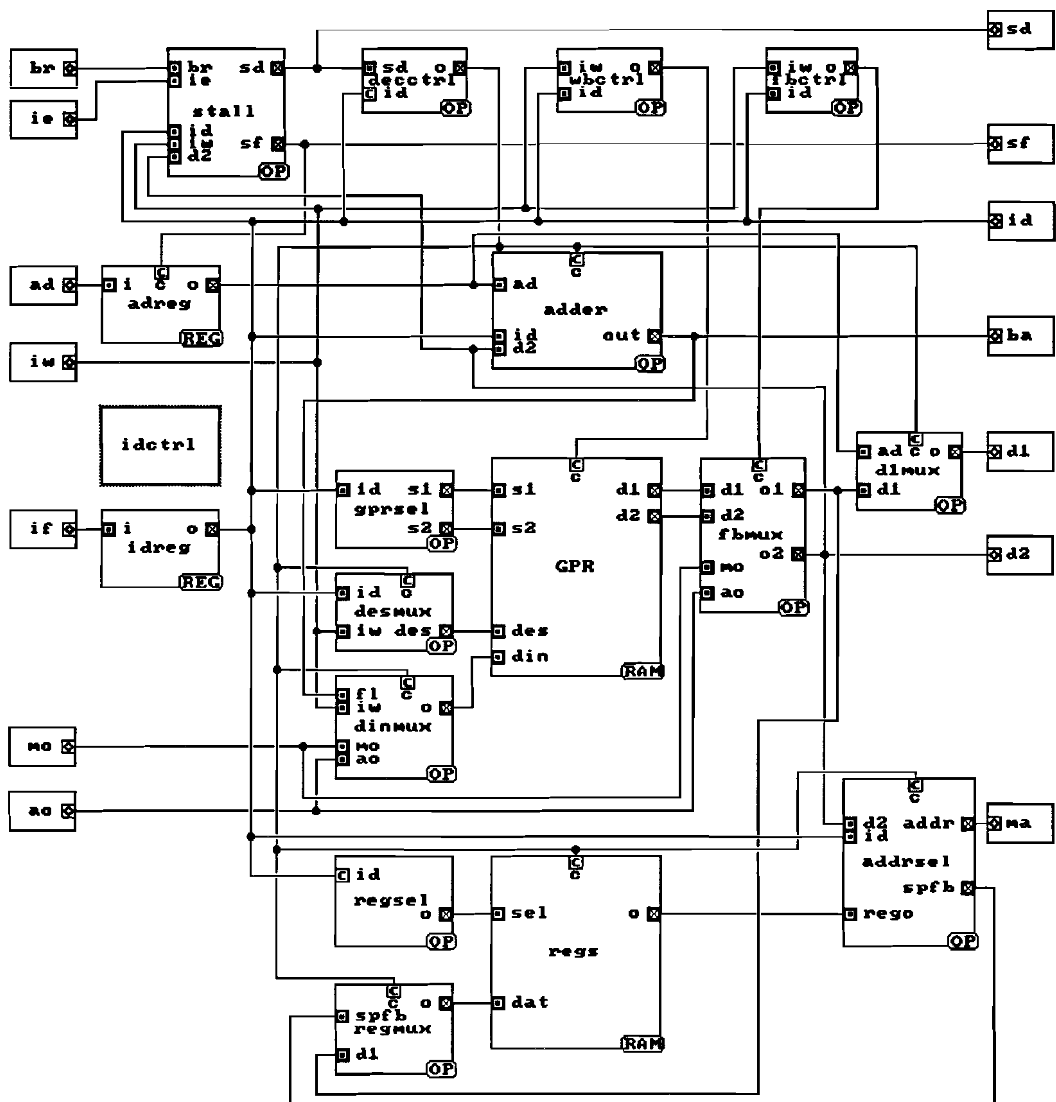
IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

Part 2.2. The description of the schematic decode.



Connectors

name	I/O	bits	description
ad	I	12	next instruction address.
ao	I	16	ALU output.
ba	O	12	branch address; used in case of a conditional branch.
br	I	1	the branch signal.
d1	O	16	data1.
d2	O	16	data2.
id	O	16	instruction in the decode stage.
ie	I	16	instruction in the execute stage.
if	I	16	instruction in the fetch stage.
iw	I	16	instruction in the writeback stage.
ma	O	12	memory address of the data memory.
mo	I	16	memory output of the data memory.
sd	O	1	the stall signal for the decode stage and execute stage.
sf	O	1	the stall signal for the fetch stage.

Blocks

adder : operator

Purpose: This block contains the adder which calculates the branch address. The branch address is calculated by adding the displacement to the next instruction address. The displacement is formed by bits 4 to 11 of the instruction word id extended to 12 bits. This adder is also used to decrement the contents of one of the General Purpose Registers in case of a *fill* instruction. The operator **decctrl** controls the switching between these two functions. The functions are shown in the following table.

function	control value	
add	0	the output will be the branch address
decr	1	the output will be the decremented input d2.

Inputs: 'ad', 12 bits; the address of the next instruction.
'c', 8 bits, only bit 0 is used; the control connector.
'd2', 16 bits; data2.
'id', 16 bits; the instruction in the decode stage, containing the displacement.
Outputs: 'out', 12 bits;

IDaSS description:

Control specification:

```
"functions of the adder"
(0)
%0 "add address with offset"
 add.
%1 "decrement register data2"
 decr
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'out':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'out': 4n sec

Text for function 'add':

```
"Add jump offset to current program address"

out := ad +
     ((id at: 11),
      (id at: 11),
      (id at: 11),
      (id at: 11),
      (id from: 4 to: 11))
```

Internal time delays for function 'add':
From 'ad' to 'out': 11n sec
From 'id' to 'out': 11n sec

Text for function 'decr':

```
"decrement dat1 for the fill operation"

out := (d2 from: 0 to: 11) + (12 ones)
```

Internal time delays for function 'decr':
From 'd2' to 'out': 11n sec

addrsel : operator
Purpose: address selection; This operator contains the adder to calculate the effective address of the data memory. The adder is also used to decrement or increment the contents of the Stack Pointer Register. This operator has 4 functions which are shown in the following table. Switching between the 4 functions is controlled by **decctrl**.

function	control value	
direct	00	add bits 6..11 of id extended with zeroes to input regout
indexed	01	add bits 0..11 of d2 to input regout
pop	10	both outputs have value regout + 1
push	11	addr := regout, spfb := regout - 1

Inputs: 'c', 8 bits, only bits 6 and 7 are used; the control connector.
'd2', 16 bits; data2.
'id', 16 bits; the instruction in the decode stage.
'rego', 12 bits; the output of the registers DBR/IBR/SPR.
Outputs: 'addr', 12 bits; the effective address of the data memory.
'spfb', 12 bits; the stack pointer feedback value.

IDaSS description:

Control specification:

```
"select address adder function"
"direct,indexed, pop, push"
(7,6)
%00 "direct address"
 direct.
%01 "indexed address"
 indexed.
%10 "pop address"
 pop.
%11 "push address"
 push
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'addr':
Data transfer delay: 2n sec

System-defined timing for output 'spfb':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'addr': 4n sec
For output 'spfb': 4n sec

Text for function 'direct':

```
"determine direct address"

addr := rego + (6 zeroes, (id from: 6 to: 11)).
spfb := 0
```

Internal time delays for function 'direct':
From 'id' to 'addr': 11n sec
From 'rego' to 'addr': 11n sec

Text for function 'indexed':

```
"determine indexed address"

addr := rego + (d2 from: 0 to: 11).
spfb := 0
```

Internal time delays for function 'indexed':
From 'd2' to 'addr': 11n sec
From 'rego' to 'addr': 11n sec

Text for function 'pop':

```
"determine address for pop operation"
"the stack pointer must be incremented"

addr := rego + 1.
spfb := rego + 1
```

Internal time delays for function 'pop':
From 'rego' to 'addr': 11n sec
From 'rego' to 'spfb': 11n sec

Text for function 'push':

```
"address for a push operation"
"the stack pointer must be decremented"

addr := rego.
spfb := rego + (12 ones)
```

Internal time delays for function 'push':
From 'rego' to 'spfb': 11n sec

adreg : register

Purpose: address register; Part of the pipeline register between the fetch stage and the decode stage. This register holds the address of the next instruction. This register is controlled by the stall signal for the fetch stage, generated by logic block **stall**.

Inputs: 'c', 1 bit; the control connector.
'i', 12 bits.
Outputs: 'o', 12 bits.

IDaSS description:

Default function: 'load'
Reset value: 4095 (0FFFh)

Control specification:

```
"on a stall hold this register"

%0 load.
%1 hold
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

d1mux : operator

Purpose: multiplexor for d1; A plain multiplexor in datapath d1 that switches between data from the General Purpose Registers (output d1) and data from the address register **adreg**. Logic block decctrl controls the switching between the two inputs. This is shown in the following table.

function	control value	
seld1	0	select input d1
selad	1	select input ad

Inputs: 'ad', 12 bits; address to be saved on stack
'c', 8 bits, only bit 1 is used; the control connector.
'd1', 16 bits; the data from the General Purpose Register.
Outputs: 'o', 16 bits; output of the multiplexor.

IDaSS description:

Control specification:

```
(1)
%0 "select d1"
 seld1.
%1 "select ad"
 selad
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o': 4n sec

Text for function 'selad':

```
"feed through the instruction address"

o := (4 zeroes) , ad
```

Text for function 'seld1':

```
"feed through of data1"

o := d1
```

decctrl : operator
Purpose: decode stage controller; This is the controller for most parts of the decode stage. The following table shows the several types of instructions and the functions of the logic blocks for each of those types. The 'wrreg' signal is the write signal for the register file **regs**. Next to the instruction input 'id' the logic block **decctrl** has another input. This is input 'sd'. This input will be 1 if a stall or a branch occurs. In those cases the Decode stage must switch to a passive state. This is done by making all the control signals zero.

instruction word	type of instruction	addrsel	regmux	dinmux	desmux	wrreg	d1mux	adder
%000XXXXXXXXXXXXX	indexed binary operations	01	0	0	0	0	0	0
%001XXXXXXXXXXXXX	direct binary operations	00	0	0	0	0	0	0
%01XXXXXXXXXXXXXX	immediate binary operations	00	0	0	0	0	0	0
%1000XXXXXXXXXXXX	jump instruction	00	0	0	0	0	0	0
%1001XXXXXXXXXXXX	jump subroutine	11	1	0	0	1	1	0
%1010XXXXXXXXXXXX	conditional jump	00	0	0	0	0	0	0
%1011XXXXXXXXXXXX	the djnz instruction	00	0	0	0	0	0	0
%1100XXXXX0XX0XXX	the pop instruction	10	1	0	0	1	0	0
%1100XXXXX0XX1XXX	the push instruction	11	1	0	0	1	0	0
%1100XXXXX1XXXXXX	other unary operations	00	0	0	0	0	0	0
%1101XXXXXXXXXXXX	reg. to reg. binary operations	00	0	0	0	0	0	0
%111XXXXXXX000XXX	the fill instruction	01	0	1	1	0	0	1
%111XXXXXXX001XXX	move to IBR, DBR or SPR	00	0	0	0	1	0	0
%111XXXXXXX010XXX	jump to interrupt routine	11	1	0	0	1	1	0
%111XXXXXXX1X0XXX	return from int./subroutine	10	1	0	0	1	0	0
%111XXXXXXX111XXX	the nop instruction	00	0	0	0	0	0	0

Inputs: 'id', 16 bits; instruction of the decode stage, configured as a control connector.
'sd', 1 bit; stall of the decode stage.
Outputs: 'o', 8 bits;

IDaSS description:

Control specification:

```
"Determine the control signals per instruction type"
%000XXXXXXXXXXXXX "indexed binary operation"
 ind.
%001XXXXXXXXXXXXX "direct binary operation"
 dir.
%01XXXXXXXXXXXXXX "immediate binary operation"
 imm.
%1000XXXXXXXXXXXX "jump"
 jmp.
%1001XXXXXXXXXXXX "jump subroutine"
 jsr.
%1010XXXXXXXXXXXX "conditional jump"
 jc.
%1011XXXXXXXXXXXX "djnz"
 djnz.
%1100XXXXX0XX0XXX "pop"
 pop.
%1100XXXXX0XX1XXX "push"
 push.
```

```
%1100XXXXX1XXXXXX "other unary operations"
 unop.
%1101XXXXXXXXXXXX "register to tegister"
 reg.
%111XXXXXXX000XXX "fill operation"
 fill.
%111XXXXXXX001XXX "move to IBR/DBR/SPR"
 regwr.
%111XXXXXXX1X0XXX "return from subr/int"
 return.
%111XXXXXXX111XXX "NOP operation"
 nop.
%111XXXXXXX010XXX "jump to interrupt routine"
 int.
```

System-defined timing for control input 'id':
 Bus to command delay: 6n sec

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined output multiplexer delays:
 For output 'o': 7n sec

Text for function 'dir':

```
"Direct Binary operations"

o  :=   1 zeroes, "address adder function"
        1 zeroes, " is direct"
        1 zeroes, "data for IBR/DBR/SPR"
        1 zeroes, "data to GPR, one if fill"
        1 zeroes, "select a GPR, one if fill"
        1 zeroes, "write IBR/DBR/SPR"
        1 zeroes, "data1 mux"
        1 zeroes  "adder select, one if fill"
```

Text for function 'djnz':

```
"The djnz operation"

o  :=   1 zeroes, "address adder function"
        1 zeroes, " is direct"
        1 zeroes, "data for IBR/DBR/SPR"
        1 zeroes, "data to GPR, one if fill"
        1 zeroes, "select a GPR, one if fill"
        1 zeroes, "write IBR/DBR/SPR"
        1 zeroes, "data1 mux"
        1 zeroes  "adder select, one if fill"
```

Text for function 'fill':

```
"The fill operation"

o  := sd if0: 1 zeroes, "address adder function"
              1 ones,   " is indexed"
              1 zeroes, "data for IBR/DBR/SPR"
              1 ones,   "data to GPR, one if fill"
              1 ones,   "select a GPR, one if fill"
              1 zeroes, "write IBR/DBR/SPR"
              1 zeroes, "data1 mux"
              1 ones    "adder select, one if fill"
         if1: 8 zeroes "stall"
```

Internal time delays for function 'fill':
From 'sd' to 'o': 5n sec

Text for function 'imm':

```
"Nothing special for immediate operations"

o  :=    1 zeroes, "address adder function"
         1 zeroes, " is direct"
         1 zeroes, "data for IBR/DBR/SPR"
         1 zeroes, "data to GPR, one if fill"
         1 zeroes, "select a GPR, one if fill"
         1 zeroes, "write IBR/DBR/SPR"
         1 zeroes, "data1 mux"
         1 zeroes  "adder select, one if fill"
```

Text for function 'ind':

```
"Indexed Binary operations"

o  :=    1 zeroes, "address adder function"
         1 ones,   " is indexed"
         1 zeroes, "data for IBR/DBR/SPR"
         1 zeroes, "data to GPR, one if fill"
         1 zeroes, "select a GPR, one if fill"
         1 zeroes, "write IBR/DBR/SPR"
         1 zeroes, "data1 mux"
         1 zeroes  "adder select, one if fill"
```

Text for function 'int':

```
"The jump interrupt subroutine operation"

o    := sd if0: 1 ones,   "address adder function"
                1 ones,   " is push"
                1 ones,   "data for IBR/DBR/SPR"
                1 zeroes, "data to GPR, one if fill"
                1 zeroes, "select a GPR, one if fill"
                1 ones,   "write IBR/DBR/SPR"
                1 ones,   "data1 mux"
                1 zeroes  "adder select, one if fill"
           if1: 8 zeroes "stall"
```

Internal time delays for function 'int':
From 'sd' to 'o': 5n sec

Text for function 'jc':

```
"The jump conditional operation"

o    :=    1 zeroes, "address adder function"
           1 zeroes, " is direct"
           1 zeroes, "data for IBR/DBR/SPR"
           1 zeroes, "data to GPR, one if fill"
           1 zeroes, "select a GPR, one if fill"
           1 zeroes, "write IBR/DBR/SPR"
           1 zeroes, "data1 mux"
           1 zeroes  "adder select, one if fill"
```

Text for function 'jmp':

```
"Nothing special for the jump operation"

o    :=    1 zeroes, "address adder function"
           1 zeroes, " is direct"
           1 zeroes, "data for IBR/DBR/SPR"
           1 zeroes, "data to GPR, one if fill"
           1 zeroes, "select a GPR, one if fill"
           1 zeroes, "write IBR/DBR/SPR"
           1 zeroes, "data1 mux"
           1 zeroes  "adder select, one if fill"
```

Text for function 'jsr':

```
"The jump subroutine operation"

o  :=  sd if0: 1 ones,   "address adder function"
               1 ones,   " is push"
               1 ones,   "data for IBR/DBR/SPR"
               1 zeroes, "data to GPR, one if fill"
               1 zeroes, "select a GPR, one if fill"
               1 ones,   "write IBR/DBR/SPR"
               1 ones,   "data1 mux"
               1 zeroes  "adder select, one if fill"
          if1: 8 zeroes  "stall"
```

Internal time delays for function 'jsr':
From 'sd' to 'o': 5n sec

Text for function 'nop':

```
"The NOP operation"

o  :=    1 zeroes, "address adder function"
         1 zeroes, " is direct"
         1 zeroes, "data for IBR/DBR/SPR"
         1 zeroes, "data to GPR, one if fill"
         1 zeroes, "select a GPR, one if fill"
         1 zeroes, "write IBR/DBR/SPR"
         1 zeroes, "data1 mux"
         1 zeroes  "adder select, one if fill"
```

Text for function 'pop':

```
"The pop operation"

o  :=  sd if0: 1 ones,   "address adder function"
               1 zeroes, " is pop"
               1 ones,   "data for IBR/DBR/SPR"
               1 zeroes, "data to GPR, one if fill"
               1 zeroes, "select a GPR, one if fill"
               1 ones,   "write IBR/DBR/SPR"
               1 zeroes, "data1 mux"
               1 zeroes  "adder select, one if fill"
          if1: 8 zeroes  "stall"
```

Internal time delays for function 'pop':
From 'sd' to 'o': 5n sec

Text for function 'push':

```
"The push operation"

o  :=  sd if0: 1 ones,   "address adder function"
               1 ones,   " is push"
               1 ones,   "data for IBR/DBR/SPR"
               1 zeroes, "data to GPR, one if fill"
               1 zeroes, "select a GPR, one if fill"
               1 ones,   "write IBR/DBR/SPR"
               1 zeroes, "data1 mux"
               1 zeroes  "adder select, one if fill"
           if1: 8 zeroes "stall"
```

Internal time delays for function 'push':
From 'sd' to 'o': 5n sec

Text for function 'reg':

```
"Register Binary operations"

o  :=    1 zeroes, "address adder function"
         1 zeroes, " is direct"
         1 zeroes, "data for IBR/DBR/SPR"
         1 zeroes, "data to GPR, one if fill"
         1 zeroes, "select a GPR, one if fill"
         1 zeroes, "write IBR/DBR/SPR"
         1 zeroes, "data1 mux"
         1 zeroes  "adder select, one if fill"
```

Text for function 'regwr':

```
"move to IBR/DBR/SPR"

o  :=  sd if0: 1 zeroes, "address adder function"
               1 zeroes, " is direct"
               1 zeroes, "data for IBR/DBR/SPR"
               1 zeroes, "data to GPR, one if fill"
               1 zeroes, "select a GPR, one if fill"
               1 ones,   "write IBR/DBR/SPR"
               1 zeroes, "data1 mux"
               1 zeroes  "adder select, one if fill"
           if1: 8 zeroes "stall"
```

Internal time delays for function 'regwr':
From 'sd' to 'o': 5n sec

Text for function 'return':

```
"Return from subroutine/interrupt"

o  :=  sd if0: 1 ones,   "address adder function"
               1 zeroes, " is pop"
               1 ones,   "data for IBR/DBR/SPR"
               1 zeroes, "data to GPR, one if fill"
               1 zeroes, "select a GPR, one if fill"
               1 ones,   "write IBR/DBR/SPR"
               1 zeroes, "data1 mux"
               1 zeroes  "adder select, one if fill"
          if1: 8 zeroes  "stall"
```

Internal time delays for function 'return':
From 'sd' to 'o': 5n sec

Text for function 'unop':

```
"The unary operations without pop and push"

o  :=   1 zeroes, "address adder function"
        1 zeroes, " is direct"
        1 zeroes, "data for IBR/DBR/SPR"
        1 zeroes, "data to GPR, one if fill"
        1 zeroes, "select a GPR, one if fill"
        1 zeroes, "write IBR/DBR/SPR"
        1 zeroes, "data1 mux"
        1 zeroes  "adder select, one if fill"
```

desmux : operator

Purpose: destination multiplexor; Writing data to one of the registers of the General Purpose Register (**GPR**) is mostly done by the writeback stage. In case of a *fill* instruction the addressing is done by the instruction in the decode stage. Selection between these two is controlled by **decctrl**. The following table shows the functions of this block.

function	control value	
seliw	0	select bits 0..2 of iw as input
selid	1	select bits 6..8 of id as input

Inputs: 'c', 8 bits, only bit 3 is used; this is the control connector.
'id', 16 bits; the instruction in the decode stage.
'iw', 16 bits; the instruction in the writeback stage.

Outputs: 'des', 3 bits; the destination register.

IDaSS description:

Control specification:

```
"select the register destination"
(3)
%0 "data from the write back stage"
 old.
%1 "selection in case of the fill operation"
 new
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for this output:
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'des': 4n sec

Text for function 'new':

```
"register selection with fill operation"

des := id from: 6 to: 8
```

Text for function 'old':

```
"data from the write back stage"

des := iw from: 0 to: 2
```

dinmux : operator

Purpose: data input multiplexor; It has 3 input busses to be switched to which means that it needs 2 control lines. One of those control lines comes from the controller **decctrl**. It switches between the output of the **adder** (at the top) in case of a *fill* instruction and the output of the Write-back stage. The latter however can be the output of the ALU or the output of the Data Memory. An extra control line is needed to select between those two inputs. This control line is not fitted as one signal but as the instruction bus 'iw'. A function internal of **dinmux** determines which input must be selected. This internal signal will be called '_am' (ALU or memory). If it is zero, data from the ALU will be selected. If it is one, data from the Data Memory is selected. The functions are shown in the following table.

function	control value	_am value	
alumem	0	0	select input ao
		1	select input mo
adder	1		select input fl

Inputs: 'ao', 16 bits; ALU output.
'c', 8 bits, only bit 4 is used; this is the control connector.
'fl', 12 bits; the output of the adder in case of a *fill* instruction.
'mo', 16 bits; data memory output.
'iw', 16 bits; the instruction in the writeback stage.
Outputs: 'o', 16 bits; the actual dat to be written to one of the **GPR** registers.

IDaSS description:

Control specification:

```
"write back data select"
(4)
%0 "write back data from ALU or mem"
 alumem.
%1 "write back data from Adderin case of a fill"
 adder
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o': 4n sec

Text for function 'adder':

```
"feed through data from adder"

o := 4 zeroes, fl
```

Text for function 'alumem':

```
"feed through data from alu or mem"
"_am is 1 in case of a memory to register move and in case of a pop"

_am := (((iw from: 14 to: 15)=%00) Λ ((iw at: 3)not)) V
    (((iw from: 12 to: 15)=%1100) Λ ((iw from: 3 to: 6)=%0000)).

o := _am
   if0: ao  "alu output"
   if1: mo  "memory output"
```

Internal time delays for function 'alumem':
From 'iw' to '_am': 12n sec
From '_am' to 'o': 5n sec
From 'ao' to 'o': 5n sec
From 'mo' to 'o': 5n sec

fbctrl : operator
Purpose: feedback control; This logic block controls the feedback multiplexor **fbmux**.

Inputs: 'id', 16 bits; the instruction in the decode stage.
'iw', 16 bits; the instruction in the writeback stage.
Outputs: 'o', 3 bits;

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'doit':

```
"feedback control for the decode stage"
"fb1 = (iws1 = ids1) and RGWiw"
"fb2 = (iws1 = ids2) and RGWiw"

"_mem is 1 in case of a memory to register move and in case of a pop"

_mem := (((iw from: 14 to: 15) = %00) Λ ((iw from: 3 to: 5) = %000)) V
     (((iw from: 12 to: 15) = %1100) Λ ((iw from: 3 to: 6) = %0000)).

"write back to a GPR"
_binwriw := (((iw from: 14 to: 15) = %00) Λ ((iw at: 3) = %0)) V
       (((iw from: 14 to: 15) = %01) Λ ((iw from: 3 to: 5)~=%111)) V "except cmp"
       (((iw from: 12 to: 15) = %1101) Λ ((iw from: 3 to: 5)~=%111)). "except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) Λ
      (((iw from: 3 to: 6) = %0001) not). "all except a push"
_djnziw := ((iw from: 12 to: 15)=%1011).

_RGWiw := _binwriw V _unwriw V _djnziw.

o := _mem,
   (((iw from: 0 to: 2) = (id from: 6 to: 8)) Λ _RGWiw) "fb2"
   ,
   (((iw from: 0 to: 2) = (id from: 0 to: 2)) Λ _RGWiw) "fb1"
```

Internal time delays for function 'doit':
From 'iw' to '_mem': 9n sec (user-changed)
From 'iw' to '_binwriw': 9n sec (user-changed)
From 'iw' to '_unwriw': 7n sec (user-changed)
From 'iw' to '_djnziw': 4n sec (user-changed)
From '_binwriw' to '_rgwiw': 3n sec (user-changed)
From '_djnziw' to '_rgwiw': 3n sec
From '_unwriw' to '_rgwiw': 3n sec (user-changed)
From '_rgwiw' to 'o': 6n sec (user-changed)
From 'id' to 'o': 8n sec (user-changed)
From 'iw' to 'o': 8n sec (user-changed)

fbmux : operator

Purpose: feedback multiplexors; This block contains two multiplexors. Each of the two multiplexors switches between the output of the General Purpose Register, the output of the ALU and the output of the Data Memory. This block is controlled by **fbctrl**. The functions of **fbmux** are shown in the following table.

function	control value	output o1	output o2
normal	X00	data from GPR	data from GPR
fbd1alu	001	data from ALU	data from GPR
fbd1mem	101	data from Data Memory	data from GPR
fbd2alu	010	data from GPR	data from ALU
fnd2mem	110	data from GPR	data from Data Memory
fbbothalu	011	data from ALU	data from ALU
fbbothmem	111	data from Data Memory	data from Data Memory

Inputs: 'ao', 16 bits; the output of the ALU.
'c', 3 bits; the control connector.
'd1', 16 bits; data input 1.
'd2', 16 bits; data input 2.
'mo', 16 bits; the output of the data memory.

Outputs: 'o1', 16 bits; data output 1.
'o2', 16 bits; data output 2.

IDaSS description:

Control specification:

```
%X00 "normal feedthrough"
normal.
%001 "d1 feedback from alu"
fbd1alu.
%101 "d1 feedback from mem"
fbd1mem.
%010 "d2 feedback from alu"
fbd2alu.
%110 "d2 feedback from mem"
fbd2mem.
%011 "feedback both from alu"
fbbothalu.
%111 "feedback both from mem"
fbbothmem
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o1':
Data transfer delay: 2n sec

System-defined timing for output 'o2':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o1': 4n sec
For output 'o2': 4n sec

Text for function 'fbbothalu':

```
"feedback both from alu"

o1 := ao.
o2 := ao
```

Text for function 'fbbothmem':

```
"feedback both from mem"

o1 := mo.
o2 := mo
```

Text for function 'fbd1alu':

```
"feedback d1 from alu"

o1 := ao.
o2 := d2
```

Text for function 'fbd1mem':

```
"feedback d1 from mem"

o1 := mo.
o2 := d2
```

Text for function 'fbd2alu':

```
"feedback d2 from alu"

o1 := d1.
o2 := ao
```

Text for function 'fbd2mem':

```
"feedback d2 from mem"

o1 := d1.
o2 := mo
```

Text for function 'normal':

```
"normal feed through"

o1 := d1.
o2 := d2
```

GPR : RAM
Purpose: General Purpose Registers; Thsi is a register file of 8 registers, each 16 bits wide.

Inputs: 'c', 1 bit; this is actually the write signal for the selected register.
'des', 3 bits; the destination selection.
'din', 16 bits; the input data for the selected register (des).
's1', 3 bits; address select for output 1.
's2', 3 bits; address select for output 2.
Outputs: 'd1', 16 bits; output 1.
'd2', 16 bits; output 2.

IDaSS description:

This RAM uses the 'ASA 6-ported register file' technology.
It contains 8 words of 16 bits each.
There is no contents file attached.
The contents are 0 after system reset.

Control specification:

```
"write register"

%1 write
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'd1':
Data transfer delay: 2n sec

System-defined timing for output 'd2':
Data transfer delay: 2n sec

The functions of the (non-control) connectors are as follows:

Two Read-only ports:
These port's technology is called 'ASA asynchronous read only port'.
Cycle and mode settings are as follows:
This port reads asynchronously.
System-defined timing for these read port:s
Addr. to output delay (async): 12n sec

One Write-only port:
This port's technology is called 'ASA synchronous write only port'.
Cycle and mode settings are as follows:
A write cycle takes 1 clock.
The input must be valid when the write cycle is started.
The memory cells are updated in clock cycle 1.
The default writing command is 'nowrite'.

System-defined timing for this write port:
Clock to (addr.) output delay: 20n sec
Port address setup time: 12n sec
Port command setup time: 12n sec
Clock to (fixed) output delay: 15n sec
Write data setup time: 16n sec (user-changed)

gprsel : operator
Purpose: General Purpose Register select; Filter the addresses from the instruction in the decode stage to select the two registers from GPR. This block is only necessary in IDaSS. In the actual circuit the input ports of the GPR can be directly connected to the appropriate bits of the instruction in the decode stage.

Inputs: 'id', 16 bits; the instruction in the decode stage.
Outputs: 's1', 3 bits; address of the first port of the GPR.
's2', 3 bits; address of the second port of the GPR.

IDaSS description:

System-defined timing for output 's1':
Data transfer delay: 2n sec

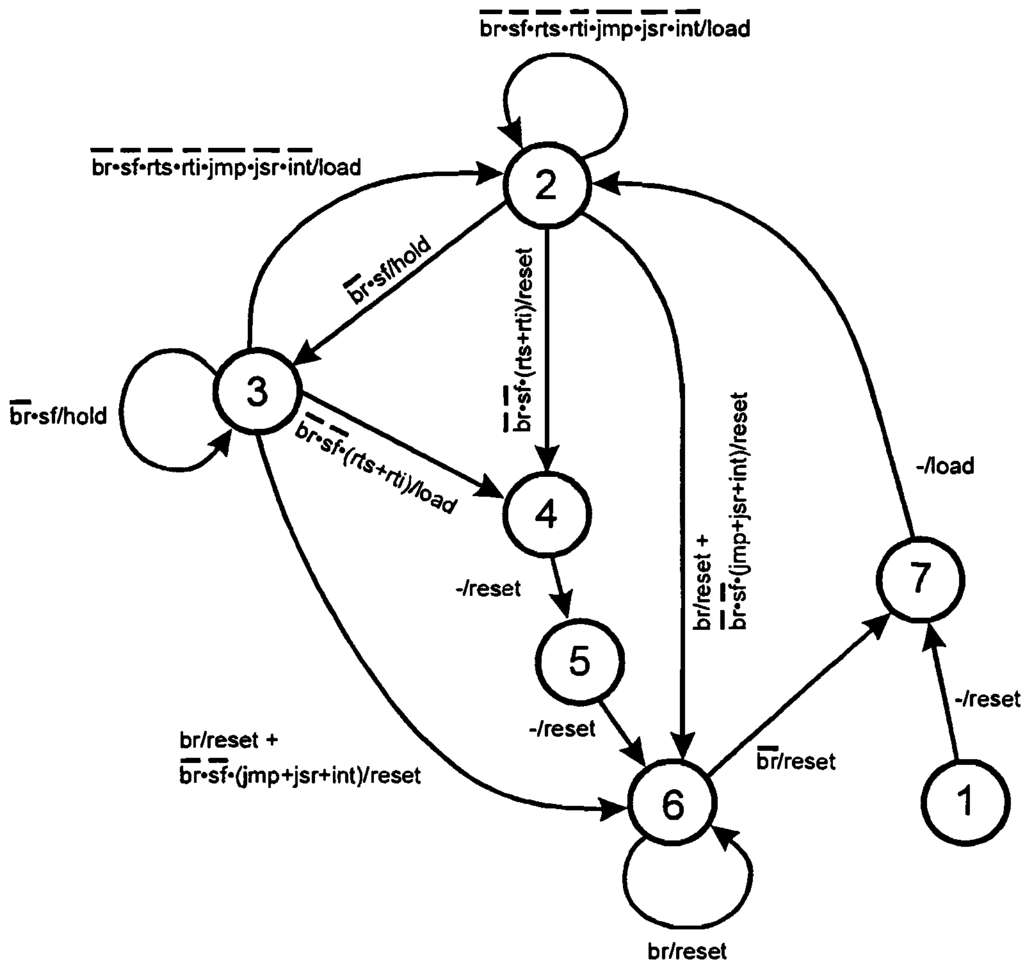
System-defined timing for output 's2':
Data transfer delay: 2n sec

Text for function 'select':

```
"select the registers from the instruction"

s1 := id from: 0 to: 2.
s2 := id from: 6 to: 8
```

idctrl : state machine controller
Purpose: In some cases the instruction in the instruction register of the decode stage must not be loaded with a new instruction on the next clock. In case of a stall the instruction must be held. In case of some instructions *nop* instructions need to be inserted to guarantee proper operation of the pipeline. This is done by resetting the instruction register. In case the function of the instruction register **idreg** (load, hold, reset) must change, it must be done before the next clock. Therefor this finite state machine must be implemented as a Mealy Machine. The following figure shows the state transition diagram. The next function of the instruction register **idreg** is shown for each transition.



IDaSS description:

This state machine controller has 7 states.
No stack is available for 'subroutine' calls.
This controller is enabled following system reset.

This state machine controller has no connectors.

System-defined timing for this state machine controller:
Clock to state delay: 15n sec
State/test to command delay: 10n sec
Test to clock setup time: 25n sec

Text for state number 1 (reset state):

```
"on system reset make the first instruction a nop"
reset:
 idreg reset;
 -> nop1
```

Text for state number 2:

```
"the state in which no nops are inserted"
nonop:
 [ br
 | %0 "no branch must be taken"
  [ sf
  | %0 "no stall"
   [ idreg
   | %100XXXXXXXXXXXXX
     "jmp or jsr must insert 2 nops"
     idreg reset; -> nop2
   | %111XXXXXXX1X0XXX
     "rts or rti must insert 4 nops"
     idreg reset; -> nop4
   | %111XXXXXXX010XXX
     "int instruction"
     idreg reset; -> nop2
   | %0XXXXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   | %101XXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   | %110XXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   | %111XXXXXXX000XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   | %111XXXXXXX0X1XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   | %111XXXXXXX1X1XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
   ]
  | %1 "a stall must be given"
   idreg hold; -> stall
  ]
 | %1 "a branch must be taken"
  idreg reset; -> nop2
 ]
```

Text for state number 3:

```
"this state will stall the pipelineregister"
stall:
 [ br "branch overrules the stall"
 | %0 "no branch"
   [ sf
   | %1 "keep stalling"
    idreg hold; -> stall
   | %0 "return from stall"
    [ idreg
    | %100XXXXXXXXXXXXX
     "jmp or jsr must insert 2 nops"
     idreg reset; -> nop2
    | %111XXXXXXX1X0XXX
     "rts or rti must insert 4 nops"
     idreg reset; -> nop4
    | %111XXXXXXX010XXX
     "int instruction"
     idreg reset; -> nop2
    | %0XXXXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    | %101XXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    | %110XXXXXXXXXXXXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    | %111XXXXXXX000XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    | %111XXXXXXX0X1XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    | %111XXXXXXX1X1XXX
     "part of others which don't need a nop"
     idreg load; -> nonop
    ]
   ]
 | %1 "branch"
  idreg reset; -> nop2
 ]
```

Text for state number 4:

```
"insert 4 nops"
nop4:
 idreg reset;
 -> nop3
```

Text for state number 5:

```
"insert 3 nops"
nop3:
 idreg reset;
 -> nop2
```

Text for state number 6:

```
"insert 2 nops"
nop2:
 [ br "branch overrules the jmp function"
 | %1 "a branch must be taken"
   idreg reset;
   -> nop2
 | %0 "continue operation"
   idreg reset;
   -> nop1
 ]
```

Text for state number 7:

```
"insert 1 nop"
nop1:
 idreg load;
 -> nonop
```

idreg : register

Purpose: instruction register for the decode stage; This register will hold the instruction to be processed by the decode stage. It receives its contents from the fetch stage. In some cases it is necessary to hold the contents or to reset the register. Control is done by the finite state machine controller **idctrl**.

Inputs: 'i', 16 bits;
Outputs: 'o', 16 bits;

IDaSS description:

Default function: 'load'
Reset value: 65535 (0FFFFh)
Reset command value: 65535 (0FFFFh)

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined timing for this register:
 Clock to data delay: 8n sec
 Clock to status delay: 7n sec
 Data setup time: 12n sec
 Command setup time: 7n sec

regmux : operator

Purpose: register file data input multiplexor; This multiplexor switches between the data from the data1 output of the General Purpose Registers and the stack pointer feed back from addrsel. The following table shows the functions of this block.

function	control value	
seld1	0	select input d1
selspfb	1	select input selspfb

Inputs: 'c', 8 bits, only bit 5 is used; the control connector.
'd1', 16 bits; the data1 output of the General Purpose Registers.
'spfb', 12 bits; stack pointer feedback.
Outputs: 'o', 12 bits;

IDaSS description:

Control specification:

```
"data selection for the IBR/DBR/SPR"
(5)
%0 "data from GPR"
 data1.
%1 "data from adder"
 feedback
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o': 4n sec

Text for function 'data1':

```
"write data from a GPR to a register"

o := d1 from: 0 to: 11
```

Text for function 'feedback':

```
"feedback for the stack pointer"

o := spfb
```

regs : RAM

Purpose: This is a register file of 19 registers, each 12 bits wide. The first 16 registers are the Index Base Registers (IBR). The next 2 registers are the Direct Base Registers (DBR). The last register is the Stack Pointer Register (SPR). They are clustered together for silicon compilation reasons (a register file takes less space than separate registers).

Inputs: 'c', 8 bits, only bit 2 is used; the control connector.
'dat', 12 bits; the data input.
'sel', 5 bits; the address input of the registers.
Outputs: 'o', 12 bits.

IDaSS description:

This RAM uses the 'ASA 6-ported register file' technology.
It contains 19 words of 12 bits each.
There is no contents file attached.
The contents are 4095 (0FFFh) after system reset.

Control specification:

```
"write register"
(2)
%1 write
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

The functions of the (non-control) connectors are as follows:

Combined read/write port:
This port's technology is called 'ASA async read/sync write port'.
Cycle and mode settings are as follows:
This port reads asynchronously.
A write cycle takes 1 clock.
The input must be valid when the write cycle is started.
The memory cells are updated in clock cycle 1.
The default writing command is 'nowrite'.

System-defined timing for this read/write port:
Addr. to output delay (async): 12n sec
Clock to (addr.) output delay: 20n sec
Port address setup time: 12n sec
Port command setup time: 12n sec
Clock to (fixed) output delay: 15n sec
Write data setup time: 16n sec (user-changed)

regsel : operator

Purpose: register selection; This block selects the appropriate register from the register file **regs**. The selection depends on the type of instruction.

Inputs: 'id', 16 bits; instruction in the decode stage, configured as a control connector.
Outputs: 'o', 5 bits; the address of the selected register.

IDaSS description:

Control specification:

```
"determine from the instruction which"
"of the IBR, DBR or SPR must be selected"

%000XXXXXXXXXXXXX "indexed binary operation"
 indexed.
%001XXXXXXXXXXXXX "direct binary operation"
 direct.
%01XXXXXXXXXXXXXX "immediate bin operation"
 indexed.
%100XXXXXXXXXXXXX "jmp and jsr"
 stack.
%101XXXXXXXXXXXXX "conditional jump"
 indexed.
%110XXXXXXXXXXXXX "pop and push and others"
 stack.
%111XXXXXXX000XXX "fill operation"
 indexed.
%111XXXX001001XXX "move to IBR"
 indexed.
%111XXXX010001XXX "move to DBR"
 direct.
%111XXXX100001XXX "move to SPR"
 stack.
%111XXXXXXX1X0XXX "rts and rti"
 stack.
%111XXXXXXX010XXX "int"
 stack.
%111XXXXXXX111XXX "nop"
 indexed
```

System-defined timing for control input 'id':
 Bus to command delay: 6n sec

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined output multiplexer delays:
 For output 'o': 4n sec

Text for function 'direct':

```
"select a direct register"

o := (1 ones, 3 zeroes, (id at: 12))
```

Text for function 'indexed':

```
"select an index register"

o := (1 zeroes),(id from: 9 to: 12)
```

Text for function 'stack':

```
"select the stack pointer register"

o := 10010b
```

stall : operator

Purpose: stall signal generator; This block will produce the stall signals for the microprocessor. In case an instruction arrives at the Decode stage that can not start its operation because it needs data that will not be available until the next clock cycle, it will have to stall the fetching of the next instruction. The instruction in the Decode stage must stay passive until it can start its operation. The instructions at the Execute and Write-back stages must continue their operations. The output signal 'sf' takes care of stalling the fetch stage. The output signal 'sd' takes care of stalling the decode stage.

Inputs: 'br', 1 bit; the branch signal.
'd2', 16 bits; the data 2 output of the General Purpose Registers.
'id', 16 bits; the instruction in the decode stage.
'ie', 16 bits; the instruction in the execute stage.
'iw', 16 bits; the instruction in the writeback stage.

Outputs: 'sd', 1 bit; the stall signal for the decode stage.
'sf', 1 bit; the stall signal for the fetch stage.

IDaSS description:

System-defined timing for output 'sd':
Data transfer delay: 2n sec

System-defined timing for output 'sf':
Data transfer delay: 2n sec

Text for function 'stall':

```
"sf is the stall signal for the fetch stage"
"sd is the stall signal for the decode stage"

"_idmemrd is 1 when data will go from memory to a register"
_iememrd := (((ie from: 14 to: 15)=%00) Λ ((ie from: 3 to: 5)=%000)) V
        (((ie from: 12 to: 15)=%1100) Λ ((ie from: 3 to: 6)=%0000)).
"_samereg1 is 1 if the write-back register of ie is also used in id as register1"
```

```
_samereg1 := ((ie from: 0 to: 2)=(id from: 0 to: 2)).
"_samereg2 is 1 if the write-back register of ie is also used in id as register2"
_samereg2 := ((ie from: 0 to: 2)=(id from: 6 to: 8)).
"_regop is 1 in case an instruction uses one of the General Purpose Registers"
_regop := ((((id from: 14 to: 15)=%00) Λ ((id from: 3 to: 5)=%001)) V
        (((id from: 14 to: 15)=%01) Λ (((id from: 3 to: 5)=%000)not))) V
        (((id from: 12 to: 15)=%1011) V
        ((id from: 12 to: 15)=%1101)) V
        ((((id from: 12 to: 15)=%1100) Λ (((id from: 3 to: 6)=%0000)not)) V
        (((id from: 13 to: 15)=%111) Λ ((id from: 4 to: 5)=%00))).
"_regwrie is 1 if ie will write-back to a register"
_regwrie := (((ie from: 14 to: 15)=%00) Λ ((ie from: 3 to: 5)=%000)) V
         (((ie from: 14 to: 15)=%01) Λ ((ie from: 3 to: 5)~=%111)) V
         ((ie from: 12 to: 15)=%1011) V
         (((ie from: 12 to: 15)=%1100) Λ ((ie from: 3 to: 6)~=%0001)) V
         (((ie from: 12 to: 15)=%1101) Λ ((ie from: 3 to: 5)~=%111)).
"_regwriw is 1 if iw will write-back to a register"
_regwriw := (((iw from: 14 to: 15)=%00) Λ ((iw from: 3 to: 5)=%000)) V
         (((iw from: 14 to: 15)=%01) Λ ((iw from: 3 to: 5)~=%111)) V  "except cmp"
         ((iw from: 12 to: 15)=%1011) V
         (((iw from: 12 to: 15)=%1100) Λ ((iw from: 3 to: 6)~=%0001)) V
         (((iw from: 12 to: 15)=%1101) Λ ((iw from: 3 to: 5)~=%111)).   "except cmp"
"_indid is 1 if id is an indexed operation"
_indid := ((id from: 13 to: 15)=%000).
"_fillid is 1 if id is a fill instruction"
_fillid := ((id from: 13 to: 15)=%111) Λ ((id from: 3 to: 5)=%000).
"_popid is 1 if id is a pop instruction"
_popid := ((id from: 12 to: 15)=%1100) Λ ((id from: 3 to: 6)=%0000).
"_pushid is 1 if id is a push instruction"
_pushid := ((id from: 12 to: 15)=%1100) Λ ((id from: 3 to: 6)=%0001).
"_return is 1 if id is an rti or rts instruction"
_return := (((id from: 13 to: 15)=%111) Λ (id at: 5)) Λ ((id at: 3)not).
"_jsr is 1 if id is a jump subroutine instruction"
_jsr := ((id from: 12 to: 15)=%1001).
"_movrid is 1 if id is a move to IBR/DBR/SPR instruction"
_movrid := ((id from: 13 to: 15)=%111) Λ ((id from: 3 to: 5)=%001).
"_condbrie is 1 if ie is a conditional branch instruction"
_condbrie := ((ie from: 13 to: 15)=%101).
"_std to reuse the equation of std for stf"
_std := (_iememrd  Λ _regop Λ (_samereg1 V _samereg2)) V
      (_fillid Λ (_regwrie V _regwriw)) V
      (_fillid Λ _condbrie) V
      (_popid Λ _condbrie) V
      (_pushid Λ _condbrie) V
      (_return Λ _condbrie) V
      (_jsr Λ _condbrie) V
      (_movrid Λ _condbrie) V
      (_movrid Λ _regwrie Λ _samereg1)  V
      (_indid Λ _regwrie Λ _samereg2).
```

```
sd := _std V br.
sf := (_std V (_fillid Λ ((d2 = 0)not))) Λ ((br)not)
```

Internal time delays for function 'stall':
From 'ie' to '_iememrd': 9n sec (user-changed)
From 'id' to '_samereg1': 6n sec
From 'ie' to '_samereg1': 6n sec
From 'id' to '_samereg2': 6n sec
From 'ie' to '_samereg2': 6n sec
From 'id' to '_regop': 12n sec (user-changed)
From 'ie' to '_regwrie': 12n sec (user-changed)
From 'iw' to '_regwriw': 12n sec (user-changed)
From 'id' to '_indid': 4n sec (user-changed)
From 'id' to '_fillid': 6n sec (user-changed)
From 'id' to '_popid': 6n sec (user-changed)
From 'id' to '_pushid': 6n sec (user-changed)
From 'id' to '_return': 6n sec (user-changed)
From 'id' to '_jsr': 4n sec (user-changed)
From 'id' to '_movrid': 6n sec (user-changed)
From 'ie' to '_condbrie': 4n sec (user-changed)
From '_condbrie' to '_std': 4n sec (user-changed)
From '_fillid' to '_std': 4n sec (user-changed)
From '_iememrd' to '_std': 4n sec (user-changed)
From '_indid' to '_std': 4n sec (user-changed)
From '_jsr' to '_std': 4n sec (user-changed)
From '_movrid' to '_std': 4n sec (user-changed)
From '_popid' to '_std': 4n sec (user-changed)
From '_pushid' to '_std': 4n sec (user-changed)
From '_regop' to '_std': 4n sec (user-changed)
From '_regwrie' to '_std': 4n sec (user-changed)
From '_regwriw' to '_std': 4n sec (user-changed)
From '_return' to '_std': 4n sec (user-changed)
From '_samereg1' to '_std': 4n sec (user-changed)
From '_samereg2' to '_std': 4n sec (user-changed)
From '_std' to 'sd': 1n sec (user-changed)
From 'br' to 'sd': 3n sec
From '_fillid' to 'sf': 4n sec (user-changed)
From '_std' to 'sf': 4n sec (user-changed)
From 'br' to 'sf': 6n sec (user-changed)
From 'd2' to 'sf': 8n sec (user-changed)

wbctrl : operator

Purpose: writeback control; This logic block controls the 'write'-signal of the General Purpose Registers. The *fill* instruction is the only instruction in the Decode stage that can write to these registers. All other instructions that can modify the contents of one of the General Purpose Registers will only do so when they are in the Write-back stage of the pipeline.

Inputs: 'id', 16 bits; the instruction in the decode stage.
'iw', 16 bits; the instruction in the writeback stage.
Outputs: 'o', 1 bit; the write signal of the General Purpose Registers.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'writeback':

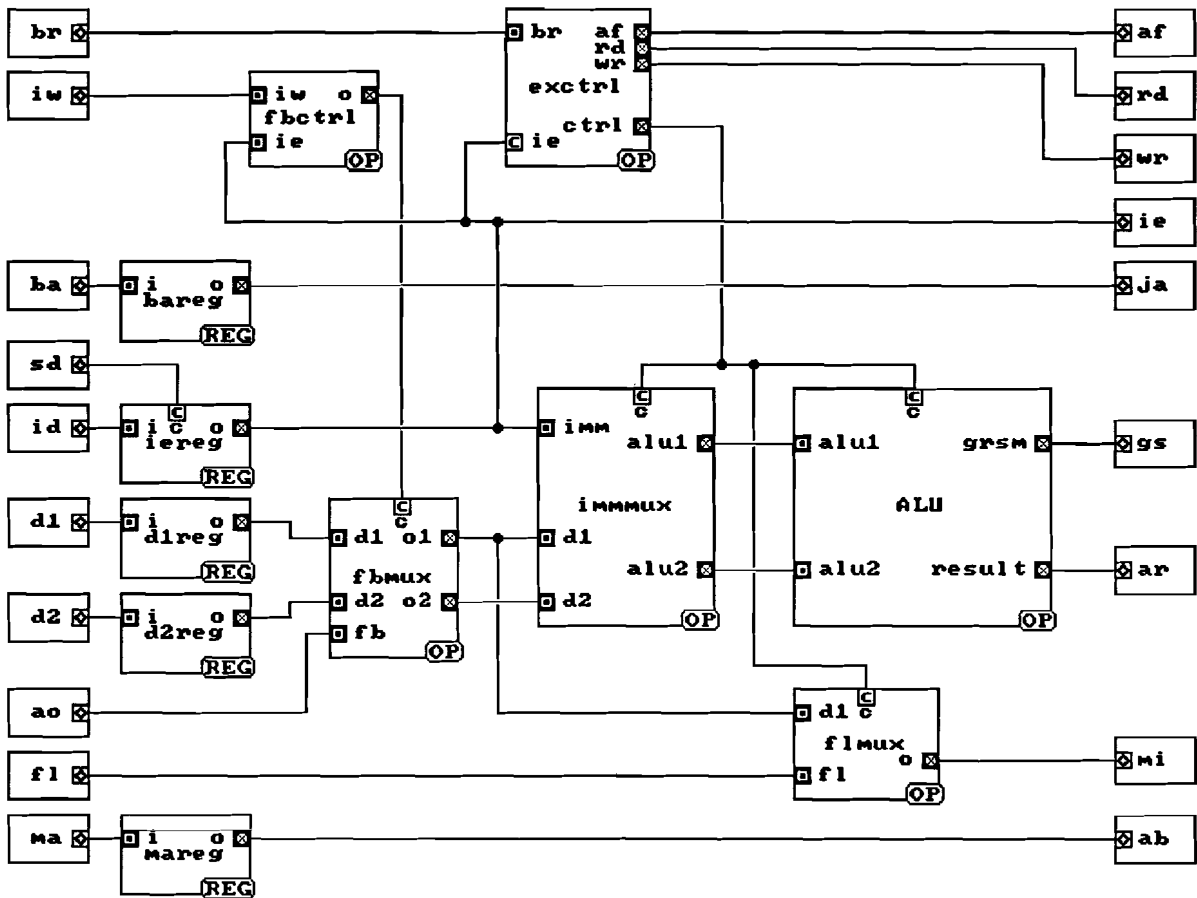
```
"write back to a GPR"

_fillid := ((id from: 13 to: 15) = %111) Λ
        ((id from: 3 to: 5) = %000).
_binwriw := (((iw from: 14 to: 15) = %00) Λ ((iw at: 3) = %0)) V
         ((((iw from: 14 to: 15) = %01) V ((iw from: 12 to: 15) = %1101)) Λ
          ((iw from: 3 to: 5) ~= %111)). "except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) Λ
        (((iw from: 3 to: 6) = %0001) not).
_djnziw := ((iw from: 12 to: 15) = %1011).

o := _fillid V _binwriw V _unwriw V _djnziw
```

Internal time delays for function 'writeback':
From 'id' to '_fillid': 6n sec (user-changed)
From 'iw' to '_binwriw': 10n sec (user-changed)
From 'iw' to '_unwriw': 6n sec (user-changed)
From 'iw' to '_djnziw': 4n sec (user-changed)
From '_binwriw' to 'o': 3n sec (user-changed)
From '_djnziw' to 'o': 3n sec
From '_fillid' to 'o': 3n sec (user-changed)
From '_unwriw' to 'o': 3n sec (user-changed)

Part 2.3. The description of the schematic execute.



Connectors

name	I/O	bits	description
ab	O	12	address bus of data memory.
af	O	1	alternate flags.
ao	I	16	feed back of ALU output.
ar	O	16	ALU result.
ba	I	12	branch address.
br	I	1	branch signal.
d1	I	16	data 1 bus.
d2	I	16	data 2 bus.
fl	I	4	flags to be saved on stack.
gs	O	2	'greater' and 'smaller' flags.
id	I	16	instruction currently in the decode stage.
ie	O	16	instruction currently in the execute stage.
iw	I	16	instruction currently in the writeback stage.
ja	O	12	branch address.
ma	I	12	data memory address.
mi	O	16	data memory input data.
rd	O	1	read signal for data memory.
sd	I	1	stall signal.
wr	O	1	write signal for data memory.

Blocks

ALU : operator

Purpose: Arithmetic Logic Unit; This block can perform 16 different arithmetic operations. The following table shows the encoding of the functions of this block.

function	control value	
mov	0000	the output gets the value of input alu2
mhi	0001	the output gets the value of input alu1
add	0010	alu1 + alu2
sub	0011	alu1 - alu2
and	0100	a bitwise and operation on both inputs
or	0101	a bitwise or operation on both inputs
xor	0110	a bitwise xor operation on both inputs
cmp	0111	the value of alu1 compared to that of alu2
sel2	1000	select the second nibble of alu1 (bits 4..7)
sel3	1001	select the third nibble of alu1 (bits 8..12, note: 5 bits)
cpl	1010	a bitwise invert of the bits of alu1
set	1011	the output bits all get value 1
dec	1100	alu1 - 1
shr	1101	shift all bits of alu1 right one bit, a zero is shifted in
shl8	1110	shift all bits of alu1 left 8 bits, zeroes are shifted in
shr8	1111	shift all bits of alu1 right 8 bits, zeroes are shifted in

This block also generates two flags. They form the output 'grsm' (greater and smaller). The input 'alu1' will always be compared with input 'alu2'. The coding of the signal 'grsm' is shown in the following tabel.

grsm	
00	alu1 = alu2
10	alu1 > alu2
01	alu1 < alu2

Inputs: 'alu1', 16 bits; input 1 of the ALU.
'alu2', 16 bits; input 2 of the ALU.
'c', 7 bits, only bits 2,3,4 and 5 are used; the control connector.

Outputs: 'grsm', 2 bits; the flags 'greater' and 'smaller'.
'result', 16 bits; the result of the arithmetic operation.

IDaSS description:

Control specification:

```
(5,4,3,2)
%0000 mov.
%0001 mhi.
%0010 add.
%0011 sub.
%0100 and.
%0101 or.
```

```
%0110 xor.
%0111 cmp.
%1000 sel2.
%1001 sel3.
%1010 cpl.
%1011 set.
%1100 dec.
%1101 shr.
%1110 shl8.
%1111 shr8.
```

System-defined timing for this input 'c':
 Bus to command delay: 6n sec

System-defined timing for output 'grsm':
 Data transfer delay: 2n sec

System-defined timing for output 'result':
 Data transfer delay: 2n sec

System-defined output multiplexer delays:
 For output 'result': 7n sec

Text for function 'add':

```
"Basic add operation"

result := (alu1, 1 zeroes) + (alu2, 1 zeroes)
      from: 1 to: 16.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'add':
 From 'alu1' to 'result': 14n sec
 From 'alu2' to 'result': 14n sec
 From 'alu1' to 'grsm': 11n sec
 From 'alu2' to 'grsm': 11n sec

Text for function 'and':

```
"Basic and operation"

result := alu1 Λ alu2.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'and':
 From 'alu1' to 'result': 3n sec
 From 'alu2' to 'result': 3n sec
 From 'alu1' to 'grsm': 11n sec
 From 'alu2' to 'grsm': 11n sec

Text for function 'cmp':

```
"Basic compare operation"

result := 0.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'cmp':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'cpl':

```
"Unary complement operation"

result := alu1 not.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'cpl':
From 'alu1' to 'result': 2n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'dec':

```
"Basic decrement operation"

result := (alu1, 1 zeroes) + (16 ones, 1 zeroes)
      from: 1 to: 16.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'dec':
From 'alu1' to 'result': 14n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'mhi':

```
"move data from the input ALU1 to the output"

result := alu1.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'mhi':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'mov':

```
"move data from input ALU1 to output"

result := alu2.
```

```
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'mov':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'or':

```
"Basic or operation"

result := alu1 V alu2.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'or':
From 'alu1' to 'result': 3n sec
From 'alu2' to 'result': 3n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'sel2':

```
"Unary select second nible operation"

_tmp := alu1 shr: 4.
result := _tmp Λ %0000000000001111.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'sel2':
From '_tmp' to 'result': 3n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'sel3':

```
"Unary select third nible operation"

_tmp := alu1 shr: 8.
result := _tmp Λ %0000000000011111.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'sel3':
From '_tmp' to 'result': 3n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'set':

```
"Unary set operation"

result := %1111111111111111.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'set':

From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'shl8':

```
"Unary shift left over 8 bits operation"

result := alu1 shl: 8.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'shl8':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'shr':

```
"Unary shift right operation"

result := alu1 shr: 1.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'shr':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'shr8':

```
"unary shift right over 8 bits operation"

result := alu1 shr: 8.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'shr8':
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'sub':

```
"Basic subtract operation"

result := (alu1, 1 ones) + (alu2 not, 1 ones)
        from: 1 to: 16.
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'sub':
From 'alu1' to 'result': 14n sec
From 'alu2' to 'result': 16n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

Text for function 'xor':

```
"Basic exlusive or operation"

result := alu1 >< alu2.
```

```
grsm := (alu1 > alu2),(alu1 < alu2)
```

Internal time delays for function 'xor':
From 'alu1' to 'result': 4n sec
From 'alu2' to 'result': 4n sec
From 'alu1' to 'grsm': 11n sec
From 'alu2' to 'grsm': 11n sec

bareg : register
Purpose: branch address register; This is the pipeline register for the branch address.

Inputs: 'i', 12 bits;
Outputs: 'o', 12 bits;

IDaSS description:

Default function: 'load'
Reset value: 4095 (0FFFh)
System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

d1reg : register
Purpose: data 1 register; This is the pipeline register for the data 1 bus.

Inputs: 'i', 16 bits.
Outputs: 'o', 16 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

d2reg : register
Purpose: data 2 register; This is the pipeline register for the data 2 bus.

Inputs: 'i', 16 bits.
Outputs: 'o', 16 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

exctrl : operator
Purpose: execute stage controller; This block is the instruction decoder for the execute stage. It controls the blocks **ALU**, **flmux** and **immmux** and also generates the control signals for the data memory. The following table shows the several types of instructions and the functions of the logic blocks for each of those types. The *readmem* and *writemem* signals are the 'read' and 'write' control signals for the Data Memory. The *altflag* signal is used in the Write-back stage and denotes that the flags may be altered by that instruction.
Next to the instruction input 'ie' this logic block has another input. This is input 'br'. This input will be 1 if a branch occurs. In those cases the Execute stage must switch to a passive state. This is done by making all the control signals zero, except the read signal of the Data Memory which will be one.

instruction word	type of instruction	flmux	aluop3	aluop2..0	altflag	immmux	writemem	readmem
%000XXXXXXXXXXXXX	indexed binary mov operation	0	0	a)	1	11	b)	c)
%001XXXXXXXXXXXXX	direct binary mov operation	0	0	a)	1	11	b)	c)
%01XXXXXXXXXXXXXX	immediate binary operations	0	0	a)	1	d)	0	1
%1000XXXXXXXXXXXX	jump instruction	0	0	a)	0	11	0	1
%1001XXXXXXXXXXXX	jump subroutine	0	0	a)	0	11	1	0
%1010XXXXXXXXXXXX	conditional jump	0	0	a)	0	11	0	1
%1011XXXXXXXXXXXX	the djnz instruction	0	1	100	1	11	0	1
%1100XXXXX0XX0XXX	the pop instruction	0	1	a)	1	11	0	1
%1100XXXXX0XX1XXX	the push instruction	0	1	a)	1	11	1	0
%1100XXXXX1XXXXXX	other unary operations	0	1	a)	1	11	0	1
%1101XXXXXXXXXXXX	reg. to reg. binary operations	0	0	a)	1	11	0	1
%111XXXXXXX000XXX	the fill instruction	0	0	a)	1	11	1	0
%111XXXXXXX001XXX	move to IBR, DBR or SPR	0	0	a)	1	11	0	1
%111XXXXXXX010XXX	jump to interrupt routine	1	0	a)	0	11	1	0
%111XXXXXXX1X0XXX	return from int./subroutine	0	0	a)	0	11	0	1
%111XXXXXXX111XXX	the nop instruction	0	0	000	0	11	0	1

note a) : bits 5..3 of instruction word ie.
note b) : bit 3 of instruction word ie.
note c) : inverted bit 3 of instruction word ie.
note d) : ((ie from: 4 to: 5)~=%00),((ie from: 3 to: 5)=%001)

Inputs: 'br', 1 bit; the branch signal.
'ie', 16 bits; instruction in the execute stage, configured as a control connector.
Outputs: 'af', 1 bit; a control signal that denotes that the flags may be changed.
'ctrl', 7 bits; the control word for the execute stage.
'rd', 1 bit; the 'read' signal for the data memory.
'wr', 1 bit; the 'write' signal for the data memory.

IDaSS description:

Control specification:

```
"decode the instruction into control signals"
%000XXXXXXXXXXXXX "Indexed move"
 ind.
%001XXXXXXXXXXXXX "Direct move"
 dir.
%01XXXXXXXXXXXXXX "Immediate operation"
 imm.
%1000XXXXXXXXXXXX "Jump unconditional"
 jmp.
```

```
%1001XXXXXXXXXXXX "Jump subroutine"
jsr.
%1010XXXXXXXXXXXX "Jump conditional"
jc.
%1011XXXXXXXXXXXX "Decrement and jump on not zero"
djnz.
%1100XXXXX0XX0XXX "Pop operation"
pop.
%1100XXXXX0XX1XXX "Push operation"
push.
%1100XXXXX1XXXXXX "Other unary operation"
unop.
%1101XXXXXXXXXXXX "Register to register"
reg.
%111XXXXXXX000XXX "Fill operation"
fill.
%111XXXXXXX001XXX "Move to IBR/DBR/SPR"
movreg.
%111XXXXXXX1X0XXX "Return from (interrupt)subroutine"
return.
%111XXXXXXX111XXX "No operation"
nop.
%111XXXXXXX010XXX "Interrupt"
int
```

System-defined timing for output ‘af’:
Data transfer delay: 2n sec

System-defined timing for output ‘ctrl’:
Data transfer delay: 2n sec

System-defined timing for control input ‘ie’:
Bus to command delay: 6n sec

System-defined timing for output ‘rd’:
Data transfer delay: 2n sec

System-defined timing for output ‘wr’:
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'af': 4n sec
For output 'ctrl': 7n sec
For output 'rd': 4n sec
For output 'wr': 4n sec

Text for function 'dir':

```
"Direct move operation"

ctrl := br if0: 1 zeroes,              "mem mux: normal"
                1 zeroes,              "ALU binary op"
                (ie from: 3 to: 5),    "ALU operation"
                2 ones                 "normal route"
           if1: 7 zeroes.

af := (br)not.                         "if br=0: flags may alter"
                                       "if br=1: flags may not alter"

wr := br if0: (ie at: 3)               "mem write"
         if1: 1 zeroes.

rd := br if0: (ie at: 3)not            "mem read"
         if1: 1 ones
```

Internal time delays for function 'dir':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec
From 'br' to 'wr': 5n sec
From 'ie' to 'wr': 5n sec
From 'br' to 'rd': 5n sec
From 'ie' to 'rd': 7n sec

Text for function 'djnz':

```
"Decrement and jump if not zero"

ctrl := br if0: 1 zeroes,              "mem mux: normal"
                1 ones,                "ALU unary op"
                1 ones, 2 zeroes,      "ALU operation"
                2 ones                 "normal route"
           if1: 7 zeroes.

af := (br)not.                         "if br=0: flags may alter"
                                       "if br=1: flags may not alter"

wr := 1 zeroes.                        "no mem write"

rd := 1 ones                           "mem read"
```

Internal time delays for function 'djnz':
From 'br' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec

Text for function 'fill':

```
"Fill operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := (br)not.                          "if br=0: flags may alter"
                                        "if br=1: flags may not alter"

wr := (br)not.                          "mem write"

rd := br                                "mem read"
```

Internal time delays for function 'fill':
 From 'br' to 'ctrl': 5n sec
 From 'ie' to 'ctrl': 5n sec
 From 'br' to 'af': 2n sec
 From 'br' to 'wr': 2n sec

Text for function 'imm':

```
"Immediate operation"

ctrl := br if0:  1 zeroes,                        "mem mux: normal"
                 1 zeroes,                        "ALU binary op"
                 (ie from: 3 to: 5),              "ALU operation"
                 (((ie from: 4 to: 5) = %00)not), "dataroute1"
                 ((ie from: 3 to: 5) = %001)      "dataroute0"
           if1:  7 zeroes.

af := (br)not.                                    "if br=0: flags may alter"
                                                  "if br=1: flags may not alter"

wr := 1 zeroes.                                   "no mem write"

rd := 1 ones                                      "mem read"
```

Internal time delays for function 'imm':
 From 'br' to 'ctrl': 5n sec
 From 'ie' to 'ctrl': 13n sec
 From 'br' to 'af': 2n sec

Text for function 'ind':

```
"Indexed move operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := (br)not.                          "if br=0: flags may alter"
                                        "if br=1: flags may not alter"

wr := br if0: (ie at: 3)                "mem write"
         if1: 1 zeroes.

rd := br if0: (ie at: 3)not             "mem read"
         if1: 1 ones
```

Internal time delays for function 'ind':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec
From 'br' to 'wr': 5n sec
From 'ie' to 'wr': 5n sec
From 'br' to 'rd': 5n sec
From 'ie' to 'rd': 7n sec

Text for function 'int':

```
"Jump interrupt subroutine operation"

ctrl := br if0:  1 ones,                "mem mux: save"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := 1 zeroes.                         "flags may not alter"

wr := (br)not.                          "mem write"

rd := br                                "no mem read"
```

Internal time delays for function 'int':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'wr': 2n sec

Text for function 'jc':

```
"Conditional jump operation"

ctrl := br if0: 1 zeroes,          "mem mux: normal"
             1 zeroes,             "ALU binary op"
             (ie from: 3 to: 5), "ALU operation"
             2 ones                "normal route"
         if1: 7 zeroes.

af := 1 zeroes.            "no flags may alter"

wr := 1 zeroes.            "no mem write"

rd := 1 ones               "mem read"
```

Internal time delays for function 'jc':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec

Text for function 'jmp':

```
"Jump operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
            if1: 7 zeroes.

af := 1 zeroes.                         "no flags may alter"

wr := 1 zeroes.                         "no mem write"

rd := 1 ones                            "mem read"
```

Internal time delays for function 'jmp':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec

Text for function 'jsr':

```
"Jump subroutine operation"

ctrl := br if0: 1 zeroes,          "mem mux: normal"
             1 zeroes,             "ALU binary op"
             (ie from: 3 to: 5),   "ALU operation"
             2 ones                "normal route"
         if1: 7 zeroes.

af := 1 zeroes.    "no flags may alter"
```

```
wr := (br)not.          "mem write"

rd := br          "no mem read"
```

Internal time delays for function 'jsr':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'wr': 2n sec

Text for function 'movreg':

```
"Move to IBR/DBR/SPR register"

ctrl := br if0:  1 zeroes,             "mem mux: normal"
                 1 zeroes,             "ALU binary op"
                 (ie from: 3 to: 5),   "ALU operation"
                 2 ones                "normal route"
           if1:  7 zeroes.

af := (br)not.                         "flags may alter"

wr := 1 zeroes.                        "no mem write"

rd := 1 ones                           "mem read"
```

Internal time delays for function 'movreg':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec

Text for function 'nop':

```
"No operation: nothing may change"

ctrl :=  1 zeroes,    "mem mux: normal"
         1 zeroes,    "ALU binary op"
         3 zeroes,    "ALU operation"
         2 ones.      "normal route"

af :=    1 zeroes.    "no flags may alter"

wr :=    1 zeroes.    "no mem write"

rd :=    1 ones       "mem read"
```

Text for function 'pop':

```
"Pop operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 ones,                "ALU unary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := (br)not.                          "flags may alter"

wr := 1 zeroes.                         "no mem write"

rd := 1 ones                            "mem read"
```

Internal time delays for function 'pop':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec

Text for function 'push':

```
"Push operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 ones,                "ALU unary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := (br)not.                          "flags may alter"

wr := (br)not.                          "mem write"

rd := br                                "no mem read"
```

Internal time delays for function 'push':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec
From 'br' to 'wr': 2n sec

Text for function 'reg':

```
"Register to register binary operation"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.
```

```
af := (br)not.                          "flags may alter"

wr := 1 zeroes.                         "no mem write"

rd := 1 ones                            "mem read"
```

Internal time delays for function 'reg':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec

Text for function 'return':

```
"Return from (interrupt)subroutine"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 zeroes,              "ALU binary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := 1 zeroes.                         "no flags may alter"

wr := 1 zeroes.                         "no mem write"

rd := 1 ones                            "mem read"
```

Internal time delays for function 'return':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec

Text for function 'unop':

```
"Other unary operations"

ctrl := br if0:  1 zeroes,              "mem mux: normal"
                 1 ones,                "ALU unary op"
                 (ie from: 3 to: 5),    "ALU operation"
                 2 ones                 "normal route"
           if1:  7 zeroes.

af := (br)not.                          "flags may alter"

wr := 1 zeroes.                         "no mem write"

rd := 1 ones                            "mem read"
```

Internal time delays for function 'unop':
From 'br' to 'ctrl': 5n sec
From 'ie' to 'ctrl': 5n sec
From 'br' to 'af': 2n sec

fbctrl : operator

Purpose: feedback control; This block controls the two feed-back multiplexors which are modelled as logic block **fbmux**. As the Data Memory is a relatively slow device it will not be fed-back to the Execute stage. The only data that can be fed-back is the data from the ALU once it has passed the pipeline register at the output of the ALU. If the instruction in the Execute stage needs the data which is going to be written back by the Write-back stage it simply switches the multiplexors to that data.

Inputs: 'ie', 16 bits; the instruction in the execute stage.
'iw', 16 bits; the instruction in the writeback stage.

Outputs: 'o', 2 bits.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'doit':

```
"feedback control"
"fb3 = (iws1 = ies1) and RGWiw and "
"     (ie <> jsr) and (ie <> int)"
"fb4 = (iws1 = ies2) and RGWiw"

"write back to a GPR"
_binwriw := (((iw from: 14 to: 15) = %01) Λ ((iw from: 3 to: 5)~=%111)) V "except cmp"
        (((iw from: 12 to: 15) = %1101) Λ ((iw from: 3 to: 5)~=%111)). "except cmp"
_unwriw := ((iw from: 12 to: 15) = %1100) Λ
        ((iw from: 3 to: 6) ~= %0001).  "all except a push"
_djnziw := ((iw from: 12 to: 15)=%1011).

_RGWiw := _binwriw V _unwriw V _djnziw.

o :=    (((iw from: 0 to: 2) = (ie from: 6 to: 8)) Λ _RGWiw)
     ,
     (((iw from: 0 to: 2) = (ie from: 0 to: 2)) Λ _RGWiw Λ
      ((ie from: 12 to: 15) ~= %1001) Λ
      ((((ie from: 13 to: 15) = %111) Λ ((ie from: 3 to: 5) = %010)) not))
```

Internal time delays for function 'doit':
From 'iw' to '_binwriw': 8n sec (user-changed)
From 'iw' to '_unwriw': 6n sec (user-changed)
From 'iw' to '_djnziw': 4n sec (user-changed)
From '_binwriw' to '_rgwiw': 3n sec (user-changed)
From '_djnziw' to '_rgwiw': 3n sec
From '_unwriw' to '_rgwiw': 3n sec (user-changed)
From '_rgwiw' to 'o': 8n sec (user-changed)
From 'ie' to 'o': 8n sec (user-changed)
From 'iw' to 'o': 8n sec (user-changed)

fbmux : operator

Purpose: feedback multiplexor; This multiplexor switches between data coming from the decode stage and data forwarded by the writeback stage. This multiplexor is controlled by **fbctrl**. The following table shows the functions of this block.

function	control value	output o1	output o2
normal	00	data from input d1	data from input d2
fbdata1	01	data from ALU feed-back	data from input d2
fbdata2	10	data from input d1	data from ALU feed-back
fbboth	11	data from ALU feed-back	data from ALU feed-back

Inputs: 'c', 2 bits; the control connector.
'd1', 16 bits; the data 1 bus.
'd2', 16 bits; the data 2 bus.
'fb', 16 bits; feedback from the piped ALU output.
Outputs: 'o1', 16 bits; output of the multiplexor in the path of the data 1 bus.
'o2', 16 bits; output of the multiplexor in the path of the data 2 bus.

IDaSS description:

Control specification:

```
%00 normal.
%01 fbdata1.
%10 fbdata2.
%11 fbboth
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'o1':
Data transfer delay: 2n sec

System-defined timing for output 'o2':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o1': 4n sec
For output 'o2': 4n sec

Text for function 'fbboth':

```
"feedback for both data1 and data2"

o1 := fb.
o2 := fb
```

Text for function 'fbdata1':

```
"feedback for data1 only"

o1 := fb.
o2 := d2
```

Text for function 'fbdata2':

```
"feedback for data2 only"

o1 := d1.
o2 := fb
```

Text for function 'normal':

```
"normal operation"

o1 := d1.
o2 := d2
```

flmux : operator

Purpose: flag multiplexor; This block always passes the lower 12 bits of input d1. The upper 4 bits can come from input d1 or from input fl. The latter will be selected if the instruction is *int*. Switching is controlled by **exctrl**. The following table shows the functions of this block.

function	control value	
normal	0	select upper 4 bits from input d1
save	1	select upper 4 bits from input fl

Inputs: 'c', 7 bits, only bit 6 is used; the control connector.
'd1', 16 bits; the data 1 bus.
'fl', 4 bits; the flags from the writeback stage.
Outputs: 'o', 16 bits; the data to be written to the data memory.

Control specification:

```
(6)
%0 normal.
%1 save
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for this output:
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'o': 4n sec

Text for function 'normal':

```
"When data will be written to memory"

o := d1
```

Text for function 'save':

```
"When an address must be saved"

o := fl,(d1 from: 0 to: 11)
```

iereg : register

Purpose: This register will hold the instruction currently in the execute stage. It can be 'reset' by the stall signal 'sd'.

Inputs: 'c', 1 bits; the control connector.
'i', 16 bits;
Outputs: 'o', 16 bits;

IDaSS description:

Default function: 'load'
Reset value: 65535 (0FFFFh)
Reset command value: 65535 (0FFFFh)

Control specification:

```
"on stall load a NOP instruction"

%0 load.
%1 setto:$ffff
```

System-defined timing for this control input:
Bus to command delay: 6n sec

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

immmux : operator

Purpose: immediate operations multiplexor; This block is the multiplexor that switches between normal data feed-through or data modifications concerning immediate data operations. This logic block contains two multiplexors that are switched by two control signals from **exctrl**. The functions of this block are shown in the following table.

function	control value	
immlow	00	alu2 its lower part is imm value extended with zeroes
immhigh	01	alu1 its upper part is imm value, lower part that of d1
opimm	10	alu1 gets value of d1, alu2 same as immlow
opnormal	11	alu1 gets value of d1, alu2 gets value of d2

Inputs: 'c', 7 bits, only bits 0 and 1 are used; the control connector.
'd1', 16 bits; the data 1 bus.
'd2', 16 bits; the data 2 bus.
'imm', 16 bits, only bits 6 .. 13 are used; the immediate value from the instruction.
Outputs: 'alu1', 16 bits; the first operand for the ALU.
'alu2', 16 bits; the second operand for the ALU.

IDaSS description:

Control specification:

```
(1,0)
%00 immlow.
%01 immhigh.
%10 opimm.
%11 opnormal
```

System-defined timing for output 'alu1':
Data transfer delay: 2n sec

System-defined timing for output 'alu2':
Data transfer delay: 2n sec

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined output multiplexer delays:
For output 'alu1': 4n sec
For output 'alu2': 4n sec

Text for function 'immhigh':

```
"data routing in case of an "
"immediate move high operation"

alu1 := (imm from: 6 to: 13), (d1 from: 0 to:7).
alu2 := d2
```

Text for function 'immlow':

```
"data routing in case of an  "
"immediate move low operation"

alu1 := d1.
alu2 := 8 zeroes, (imm from: 6 to: 13)
```

Text for function 'opimm':

```
"data routing in case of an  "
"immediate operation (no move)"

alu1 := d1.
alu2 := 8 zeroes, (imm from: 6 to: 13)
```

Text for function 'opnormal':

```
"route in case of a normal operation"

alu1 := d1.
alu2 := d2
```

mareg : register

Purpose: memory address register; This register is a pipeline register for the data memory address.

Inputs: 'i', 12 bits.
Outputs: 'o', 12 bits.

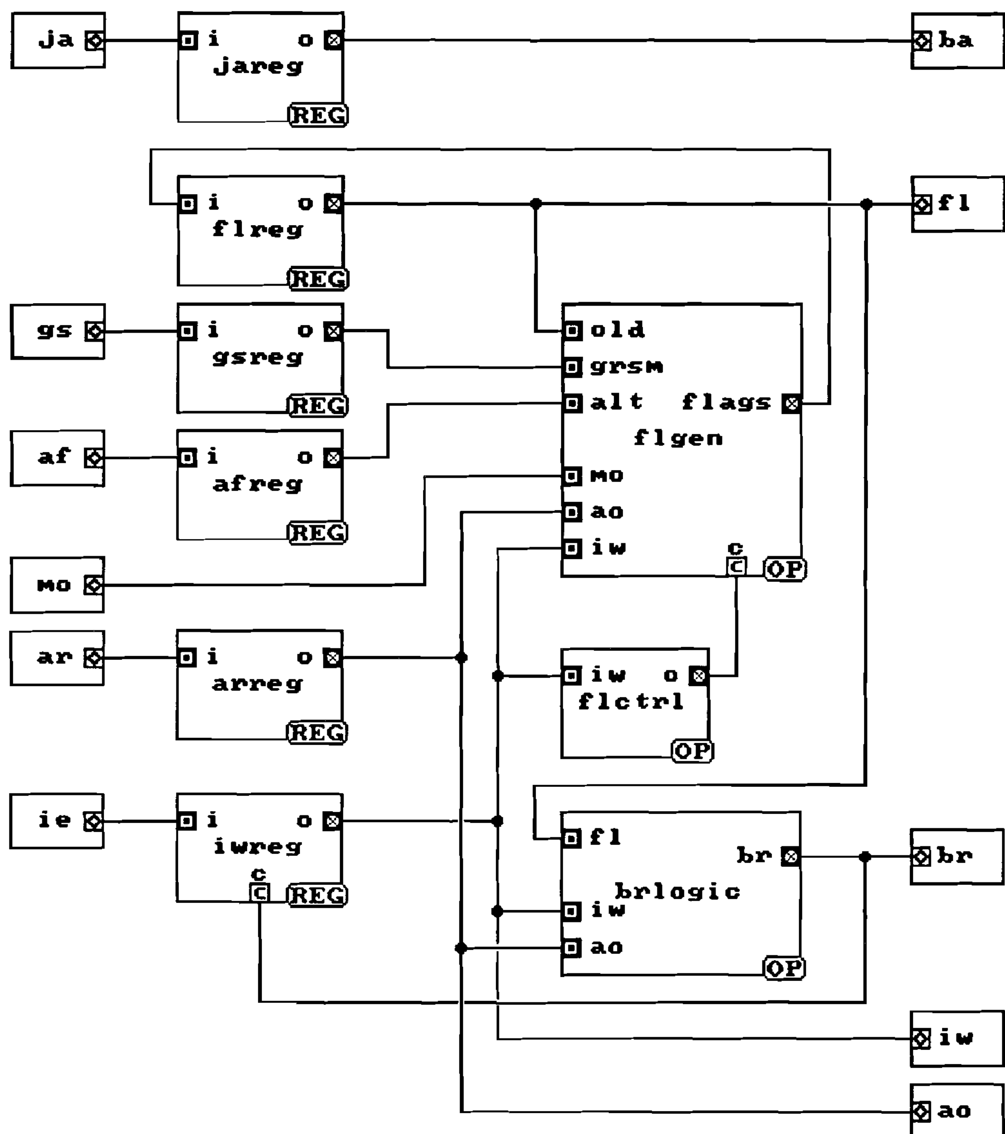
IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

Part 2.4. The description of the schematic writeback.



Connectors

name	I/O	bits	description
af	I	1	if active this signal permits the flags to change.
ao	O	16	the piped output of the ALU.
ar	I	16	the direct output of the ALU.
ba	O	12	the piped branch address.
br	O	1	the branch signal.
fl	O	4	the flags.
gs	I	2	the 'greater' and 'smaller' flags.
ie	I	16	the instruction currently in the execute stage.
iw	O	16	the instruction currently in the writeback stage.
ja	I	12	the branch address.
mo	I	16	the output of the data memory.

Blocks

afreg : register
Purpose: alternate flags register; This is the pipeline register for the signal that permits logic block **flgen** to change the value of the flags.

Inputs: 'i', 1 bit.
Outputs: 'o', 1 bit.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

arreg : register
Purpose: ALU result register; This is the pipeline register for the ALU result.

Inputs: 'i', 16 bits.
Outputs: 'o', 16 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

brlogic : operator

Purpose: branch logic; This block generates the 'br' signal (branch). If this signal is active it indicates that all instructions in the pipeline must be flushed and that the branch address must be loaded by the Program Counter Register. The branch signal can only be activated by conditional branch instructions. The following table shows these instructions and the flag conditions on which the branch signal must be activated.

instruction	encoding	flag conditions	
jbe	0000	(flag 3) not	below or equal
ja	0001	flag 3	above
jb	0010	flag 2	below
jae	0011	(flag 2)not	above or equal
je	0100	(flag 2)not and (flag 3)not	equal
jne	0101	flag 2 or flag 3	not equal
jz	0110	flag 0	zero
jnz	0111	(flag 0)not	not zero
js	1000	flag 1	set
jns	1001	(flag 1)not	not set

Inputs: 'ao', 16 bits; the piped output of the ALU.
'fl', 4 bits; the flags.
'iw', 16 bits; the instruction in the writeback stage.
Outputs: 'br', 1 bit; the branch signal.

IDaSS description:

System-defined timing for output 'br':
Data transfer delay: 2n sec

Text for function 'branch':

```
"The output indicates a branch to be taken"

_br := ((iw from: 12 to: 15) = %1010).
_djnz := ((iw from: 12 to: 15) = %1011).
_alunotzero := (ao~=$0000).

br := (_br Λ ((((iw from: 0 to: 3) = %0000) Λ ((fl at: 3)not)) V  "below or equal"
        (((iw from: 0 to: 3) = %0001) Λ (fl at: 3)) V      "above"
        (((iw from: 0 to: 3) = %0010) Λ (fl at: 2)) V      "below"
        (((iw from: 0 to: 3) = %0011) Λ ((fl at: 2)not))  V "above or equal"
        (((iw from: 0 to: 3) = %0100) Λ ((fl from: 2 to: 3) = %00)) V "equal"
        (((iw from: 0 to: 3) = %0101) Λ (((fl from: 2 to: 3) = %00)not)) V "not equal"
        (((iw from: 0 to: 3) = %0110) Λ (fl at: 0)) V       "zero"
        (((iw from: 0 to: 3) = %0111) Λ ((fl at: 0)not)) V   "not zero"
        (((iw from: 0 to: 3) = %1000) Λ (fl at: 1)) V       "all set"
        (((iw from: 0 to: 3) = %1001) Λ ((fl at: 1)not))     "not all set"
        )
    ) V
```

```
(_djnz Λ _alunotzero)
```

Internal time delays for function 'branch':
From 'iw' to '_br': 6n sec
From 'iw' to '_djnz': 6n sec
From 'ao' to '_alunotzero': 6n sec (user-changed)
From '_alunotzero' to 'br': 8n sec (user-changed)
From '_br' to 'br': 8n sec (user-changed)
From '_djnz' to 'br': 8n sec (user-changed)
From 'fl' to 'br': 14n sec (user-changed)
From 'iw' to 'br': 14n sec (user-changed)

flctrl : operator

Purpose: flag control; This block generates the control signals for the logic block **flgen**. The encoding of the output is shown in the following table.

instruction word	type of instruction	output
%00XXXXXXXXXX0XXX	read from data memory	01
%111XXXXXXX1X0XXX	restore data from the stack	10
all others		00

Inputs: 'iw', 16 bits; the instruction in the writeback stage.
Outputs: 'o', 2 bits; the control signal for logic block **flgen**.

IDaSS description:

System-defined timing for output 'o':
Data transfer delay: 2n sec

Text for function 'doit':

```
"Output 01 when: iw=00XXXXXXXXXX0XXX"
"for memread."
"Output 10 when: iw=111XXXXXXX1X0XXX"
"for restore."
"Output 00 for all others."

_restore := (((iw from: 13 to: 15)=%111) Λ ((iw at: 5) Λ ((iw at: 3)not))).
_memread := (((iw from: 14 to: 15)=%00) Λ ((iw at: 3)not)).
o := _restore,_memread
```

Internal time delays for function 'doit':
From 'iw' to '_restore': 4n sec (user-changed)
From 'iw' to '_memread': 4n sec (user-changed)

flgen : operator

Purpose: flag generator; This logic block determines the flags from the ALU and memory outputs. There are 4 flag bits:

flag bit 0: all zero
flag bit 1: all set
flag bit 2: smaller
flag bit 3: greater

This functions of this block are shown in the following table.

function	control value	
normal	00	generate the new flags if necessary
readmem	01	the 'all zero' and 'all set' flags can change on memory data
restore	10	restore the flags from the stack

Inputs: 'alt', 1 bit; if this signal is active the flags are allowed to change.
'ao', 16 bits; the piped output of the ALU.
'c', 2 bits; the control connector.
'grsm', 2 bits; the 'greater' and 'smaller' flags.
'iw', 16 bits; the instruction in the writeback stage.
'mo', 16 bits; the output of the data memory.
'old', 4 bits; the flags generated by the former instruction.

Outputs: 'flags', 4 bits; the new flags.

IDaSS description.

Control specification:

```
%00 "feed through of flags"
normal.
%01 "determine flags from memread"
readmem.
%10 "restore flags from stack"
restore
```

System-defined timing for control input 'c':
Bus to command delay: 6n sec

System-defined timing for output 'flags':
Data transfer delay: 2n sec

System-defined output multiplexer delays:
For output 'flags': 4n sec

Text for function 'normal':

```
"just route the flags from the alu to the output"

_allones := (ao = $ffff).
_allzeroes := (ao = $0000).
_compare := (((iw from: 14 to: 15)=%01) Λ ((iw from: 3 to: 5)=%111)) V
        (((iw from: 12 to: 15)=%1101) Λ ((iw from: 3 to: 5)=%111)).
_grsm := _compare if0: (old from: 2 to: 3)
           if1: grsm.

flags := alt if0: old
        if1: _grsm,_allones,_allzeroes
```

Internal time delays for function 'normal':
From 'ao' to '_allones': 4n sec (user-changed)
From 'ao' to '_allzeroes': 4n sec (user-changed)
From 'iw' to '_compare': 8n sec (user-changed)
From '_compare' to '_grsm': 5n sec
From 'grsm' to '_grsm': 5n sec
From 'old' to '_grsm': 5n sec
From '_allones' to 'flags': 5n sec
From '_allzeroes' to 'flags': 5n sec
From '_grsm' to 'flags': 5n sec
From 'alt' to 'flags': 5n sec
From 'old' to 'flags': 5n sec

Text for function 'readmem':

```
"check for zero or all set in case"
"of a memory read"

_allones := (mo = $ffff).
_allzeroes := (mo = $0000).

flags := (old from: 2 to: 3),_allones,_allzeroes
```

Internal time delays for function 'readmem':
From 'mo' to '_allones': 3n sec (user-changed)
From 'mo' to '_allzeroes': 3n sec (user-changed)

Text for function 'restore':

```
"copy the flags from memdata in"
"case of a return from interrupt"

flags := mo from: 12 to:15
```

flreg : register
Purpose: flag register; This is the pipeline register for the flags.

Inputs: 'i', 4 bits.
Outputs: 'o', 4 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

gsreg : register
Purpose: greater and smaller flags register; This register pipes the 'greater' and 'smaller' flags, generated by the ALU, from the execute stage to the writeback stage.

Inputs: 'i', 2 bits.
Outputs: 'o', 2 bits.

IDaSS description:

Default function: 'load'
Reset value: 0

System-defined timing for output 'o':
Data transfer delay: 2n sec

System-defined timing for this register:
Clock to data delay: 8n sec
Clock to status delay: 7n sec
Data setup time: 12n sec
Command setup time: 7n sec

iwreg : register
Purpose: This register holds the instruction currently in the writeback stage.

Inputs: 'i', 16 bits.
Outputs: 'o', 16 bits.

IDaSS description:

Default function: 'load'
Reset value: 65535 (0FFFFh)

Control specification:

```
"flush register in case of a branch"

%0 load.
%1 setto: $ffff
```

System-defined timing for control input 'c':
 Bus to command delay: 6n sec

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined timing for this register:
 Clock to data delay: 8n sec
 Clock to status delay: 7n sec
 Data setup time: 12n sec
 Command setup time: 7n sec

jareg : register
Purpose: This is the pipeline register for the branch address.

Inputs: 'i', 12 bits.
Outputs: 'o', 12 bits.

IDaSS description:

Default function: 'load'
Reset value: 4095 (0FFFh)

System-defined timing for output 'o':
 Data transfer delay: 2n sec

System-defined timing for this register:
 Clock to data delay: 8n sec
 Clock to status delay: 7n sec
 Data setup time: 12n sec
 Command setup time: 7n sec

Appendix B.

The assembler code for testing the instruction set.

```
0000: 8006          startup jmp     start
0001: 80AC          intstrt jmp     introut
0002:               ;
0002: 4054          testjsr inc     r4              ; this is some code to
0003: D17C                  cmp     r4,r5           ; test the jumping from
0004: D085                  mov     r5,r2           ; address 0000 to label
0005: E030                  rts                     ; start.
0006: 4000          start   clr     r0              ; r0 := 0000h
0007: 43C8                  mhi     r0,0fh          ; r0 := 0f00h
0008: 4441                  mov     r1,11h          ; r1 := 0011h
0009: 4882                  mov     r2,22h          ; r2 := 0022h
000A: 488A                  mhi     r2,22h          ; r2 := 2222h
000B: 4CC34CCB              movd    r3,3333h        ; r3 := 3333h
000D: D0C4                  mov     r4,r3           ; r4 := 3333h
000E: D085                  mov     r5,r2           ; r5 := 2222h
000F: D142                  mov     r2,r5           ; r2 := 2222h
0010: D046                  mov     r6,r1           ; r6 := 0011h
0011: D186                  mov     r6,r6           ; r6 := 0011h
0012: D007                  mov     r7,r0           ; r7 := 0f00h
0013: E04F                  mov     b0,r7           ; b0 := f00h
0014: E24F                  mov     b1,r7           ; b1 := f00h
0015: E24E                  mov     b1,r6           ; b1 := 011h
0016: E44D                  mov     b2,r5           ; b2 := 222h
0017: E08D                  mov     d0,r5           ; d0 := 222h
0018: F08C                  mov     d1,r4           ; d1 := 333h
0019: E10B                  mov     sp,r3           ; sp := 333h
001A: E10A                  mov     sp,r2           ; sp := 222h
001B: 40004208              movd    r0,0800h        ; r0 := 0800h
001D: 40014249              movd    r1,0900h        ; r1 := 0900h
001F: E088                  mov     d0,r0           ; d0 := 800h
0020: F089                  mov     d1,r1           ; d1 := 900h
0021: 7F8243CA              movd    r2,0ffeh        ; r2 := 0ffeh
0023: E10A                  mov     sp,r2           ; sp := ffeh
0024: 2409                  mov     d0[10h],r1      ; mem[810h] := 0900h
0025: 304A                  mov     d1[01h],r2      ; mem[901h] := 0ffeh
0026: 2402                  mov     r2,d0[10h]      ; r2 := mem[810h]=0900h
0027: 244A                  mov     d0[11h],r2      ; mem[811h] := 0900h
0028: 0009                  mov     b0[r0],r1       ; mem[700h] := 0900h
0029: 0248                  mov     b1[r1],r0       ; mem[911h] := 0800h
002A: 0007                  mov     r7,b0[r0]       ; r7 := mem[700h]=0900h
002B: 004F                  mov     b0[r1],r7       ; mem[800h] := 0900h
002C: 0046                  mov     r6,b0[r1]       ; r6 := mem[800h]=0900h
002D: 0045                  mov     r5,b0[r1]       ; r5 := mem[800h]=0900h
002E:                 ; now test the other binary operations
002E: D010                  add     r0,r0           ; r0 := 1000h
002F: D1D5                  add     r5,r7           ; r5 := 1200h
0030: D157                  add     r7,r5           ; r7 := 1b00h
0031: 4057                  inc     r7              ; r7 := 1b01h
0032: 4057                  add     r7,1            ; r7 := 1b02h
0033: 43E7                  and     r7,0fh          ; r7 := 0002h
0034: D0A5                  and     r5,r2           ; r5 := 0000h
0035: D13B                  cmp     r3,r4           ; equal
0036: D03B                  cmp     r3,r0           ; greater
0037: D0F8                  cmp     r0,r3           ; smaller
0038: 407F                  cmp     r7,01h          ; greater
0039: 40BF                  cmp     r7,02h          ; equal
003A: 40FF                  cmp     r7,03h          ; smaller
003B: 7C2F                  or      r7,0f0h         ; r7 := 00f2h
003C: D02B                  or      r3,r0           ; r3 := 3333h
003D: D033                  xor     r3,r0           ; r3 := 2333h
003E: 7FF4                  xor     r4,0ffh         ; r4 := 33cch
003F: D11F                  sub     r7,r4           ; r7 := cd26h
0040: 405E                  dec     r6              ; r6 := 08ffh
```

```
0041:                   ;testing of the fill instruction
0041: 4004404C                 movd    r4,0100h         ; r4 := 0100h
0043: 4443444B                 movd    r3,1111h         ; r3 := 1111h
0045: 43C5                     mov     r5,0fh           ; r5 := 000fh
0046: E84C                     mov     b4,r4            ; b4 := 0100h
0047: E943                     fill    b4[r5],r3        ; mem[0100h..010fh] := 1111h
0048:                   ;testing of the unary operations
0048: C00B                     push    r3
0049: C00C                     push    r4               ;switch r3 and r4
004A: C003                     pop     r3               ; r3 := 0100h
004B: C004                     pop     r4               ; r4 := 1111h
004C: C054                     cpl     r4               ; r4 := eeeeh
004D: C05D                     set     r5               ; r5 := ffffh
004E: 405D                     dec     r5               ; r5 := fffeh
004F: C06D                     shr     r5               ; r5 := 7fffh
0050: C075                     shl8    r5               ; r5 := ff00h
0051: C07D                     shr8    r5               ; r5 := 00ffh
0052: 4D04448C                 movd    r4,1234h         ; r4 := 1234h
0054: 77057F8D                 movd    r5,0fedch        ; r5 := fedch
0056: C044                     sel2    r4               ; r4 := 0003h
0057: C04D                     sel3    r5               ; r5 := 001eh
0058:                   ;testing of the conditional branch instructions.
0058: 4100                     mov     r0,4
0059: 4101                     mov     r1,4
005A: 4102                     mov     r2,4
005B: 4058              label1 dec     r0
005C: AFE7                     jnz     label1
005D: 4100                     mov     r0,4
005E: A026              label2 jz      label3
005F: 4058                     dec     r0
0060: 805E                     jmp     label2
0061: 4059              label3 dec     r1
0062: AFE9                     jns     label3
0063: 4101                     mov     r1,4
0064: A028              label4 js      label5
0065: 4059                     dec     r1
0066: 8064                     jmp     label4
0067: 4050              label5 inc     r0
0068: 4051                     inc     r1
0069: BFD2                     djnz    r2,label5
006A: A046                     jz      label7
006B: 41C0              label6 mov     r0,7
006C: 4101                     mov     r1,4
006D: 41C2                     mov     r2,7
006E: E030                     rts
006F: 9002              label7 jsr     testjsr
0070: 4051              label8 inc     r1
0071: D039                     cmp     r1,r0
0072: AFD0                     jbe     label8
0073: 4051                     inc     r1
0074: 4050              label9 inc     r0
0075: D039                     cmp     r1,r0
0076: AFD3                     jae     label9
0077: 4059              lab10  dec     r1
0078: D0B9                     cmp     r1,r2
0079: AFD1                     ja      lab10
007A: 4051              lab11  inc     r1
007B: D039                     cmp     r1,r0
007C: AFD2                     jb      lab11
007D: 4059              lab12  dec     r1
007E: D0B9                     cmp     r1,r2
007F: AFD5                     jne     lab12
0080: D039              lab13  cmp     r1,r0
0081: A024                     je      lab14
0082: 4051                     inc     r1
0083: 8080                     jmp     lab13
0084: 906B              lab14  jsr     label6
0085: 9087                     jsr     lab15
0086: 808A                     jmp     lab16
0087: 4059              lab15  dec     r1
```

```
0088: AFE7                    jnz     lab15
0089: E030                    rts
008A: 908C           lab16    jsr     lab17
008B: 8090                    jmp     lab19
008C: 405A           lab17    dec     r2
008D: A016                    jz      lab18
008E: 908C                    jsr     lab17
008F: E030           lab18    rts
0090: 4000           lab19    clr     r0
0091: 4001                    clr     r1
0092: D078                    cmp     r0,r1
0093: A045                    jne     lab20
0094: A035                    jne     lab20
0095: A025                    jne     lab20
0096: A014                    je      lab20
0097: 4001                    clr     r1
0098: 4051           lab20    inc     r1
0099: 4039                    cmp     r1,0
009A: A014                    je      lab21
009B: A004                    je      lab21
009C: 43C5           lab21    mov     r5,0fh
009D: 4051                    inc     r1
009E: 4000                    clr     r0
009F: 4038                    cmp     r0,0
00A0: A045                    jne     lab22
00A1: D078                    cmp     r0,r1
00A2: E943                    fill    b4[r5],r3
00A3: 4000                    clr     r0
00A4: 40C1                    mov     r1,3
00A5: 4050           lab22    inc     r0
00A6: D078                    cmp     r0,r1
00A7: A011                    ja      lab23
00A8: 80A5                    jmp     lab22
00A9: 4040           lab23    mov     r0,1
00AA: 4041                    mov     r1,1
00AB: 8006           endlab   jmp     start
00AC:                ;
00AC: 4007           introut  clr     r7
00AD: EE4F                    mov     b7,r7
00AE: 407F                    cmp     r7,1
00AF: E020                    rti
00B0:

SYMBOL TABLE
    ENDLAB P 00ab   INTROUT P 00ac   INTSTRT P 0001     LAB10 P 0077
     LAB11 P 007a     LAB12 P 007d     LAB13 P 0080     LAB14 P 0084
     LAB15 P 0087     LAB16 P 008a     LAB17 P 008c     LAB18 P 008f
     LAB19 P 0090     LAB20 P 0098     LAB21 P 009c     LAB22 P 00a5
     LAB23 P 00a9    LABEL1 P 005b    LABEL2 P 005e    LABEL3 P 0061
    LABEL4 P 0064    LABEL5 P 0067    LABEL6 P 006b    LABEL7 P 006f
    LABEL8 P 0070    LABEL9 P 0074     START P 0006   STARTUP P 0000
   TESTJSR P 0002
```

Appendix C.

The assembler code of a realistic program.

This program was written by ir. L.C. Benschop to calculate the length/literals of a block of data.

```
0000:
0000:                   ; Memory areas. dorg 0
0000:                   directpg        ds 64           ;Direct page area.
0040:                   stack           ds 16
0050:                   staktop         ds 0
0050:                   binheads        ds 32
0070:                   binheads1       ds 16
0080:                   bintails        ds 32
00A0:                   nclass          ds 32
00C0:                   binlist         ds 512
02C0:                   binlist1        ds 512
0600:                   lengths         equ 0600h
0000:
0800:                   freqs           equ 0800h
0C00:                                   dorg 0c00h
0C00:                   sfreqs          ds 1024
0000:
0000: 4000              initialize      clr r0
0001: E088                              mov d0,r0       ;Initialize direct memory base
register.
0002: 60014289                          movd r1,0a80h
0004: F089                              mov d1,r1       ;Initialize I/O base register.
0005: 4003420B                          movd r3,freqs
0007: E04B                              mov b0,r3
0008: 4003418B                          movd r3,lengths
000A: E24B                              mov b1,r3
000B: 40004088                          movd r0,512
000D: 9014                              jsr freqtolen
000E: 40004088                          movd r0,512
0010: 90D9                              jsr lentocode
0011: 8FFF                              jmp 4095
0012:
0012:                   ;freqtolen, compute code word lengths from frequency
distribution.
0012:                   ;r0 Number of symbols.
0012:                   ;b0 Address where frequencies reside.
0012:                   ;b1 Address where lengths come.
0012:                   ;
0012:
0012:                   ; belongs to pass 1 of bin sort.
0012: 0289              iszero          mov b1[r2],r1
0013: 803C                              jmp pas1cont            ;Set length to zero for
zero-freq
0014:
0014: D007              freqtolen       mov r7,r0
0015:                   ; Set the base registers.
0015: 5403400B                          movd r3,binheads
0017: E44B                              mov b2,r3
0018: 7003400B                          movd r3,binlist
001A: E64B                              mov b3,r3
001B: 6003400B                          movd r3,bintails
001D: E84B                              mov b4,r3
001E: 5C03400B                          movd r3,binheads1
0020: EA4B                              mov b5,r3
0021: 7003408B                          movd r3,binlist1
0023: EC4B                              mov b6,r3
0024: 4003430B                          movd r3,sfreqs
0026: EE4B                              mov b7,r3
0027: 6803400B                          movd r3,nclass
```

```
0029: F04B                        mov b8,r3
002A: C05E                        set r6
002B:             ; Prepare for pass 1 of the bin sort.
002B: 43C2                        mov r2,15
002C: E486                        fill b2[r2],r6         ;Set heads array.
002D: 4001                        clr r1
002E: 4002                        clr r2
002F: 4004                        clr r4
0030: 0083        pas1loop        mov r3,b0[r2]
0031: AE06                        jz iszero
0032: 4054                        inc r4
0033: 43E3                        and r3,15
0034: 04C5                        mov r5,b2[r3]
0035: A029                        jns pas1els
0036: 04CA                        mov b2[r3],r2
0037: 803A                        jmp pas1endi
0038: 08C5        pas1els         mov r5,b4[r3]
0039: 074A                        mov b3[r5],r2
003A: 08CA        pas1endi        mov b4[r3],r2
003B: 068E                        mov b3[r2],r6
003C: 4052        pas1cont        inc r2
003D: D1FA                        cmp r2,r7
003E: AF15                        jne pas1loop
003F:             ; Prepare for pass 2 of the bin sort.
003F: 43C2                        mov r2,15
0040: EA86                        fill b5[r2],r6         ;Set binheads1 array.
0041: 43C1                        mov r1,15
0042: 0442        pas2loop        mov r2,b2[r1]
0043: A0C8                        js pas2next
0044: 0083        pas2inn         mov r3,b0[r2]
0045: C043                        sel2 r3
0046: 0AC5                        mov r5,b5[r3]
0047: A029                        jns pas2else
0048: 0ACA                        mov b5[r3],r2
0049: 804C                        jmp pas2endi
004A: 08C5        pas2else        mov r5,b4[r3]
004B: 0D4A                        mov b6[r5],r2
004C: 08CA        pas2endi        mov b4[r3],r2
004D: 0C8E                        mov b6[r2],r6
004E: 0682                        mov r2,b3[r2]
004F: AF49                        jns pas2inn
0050: 4059        pas2next        dec r1
0051: AF09                        jns pas2loop
0052:             ; Prepare for pass 3 of the bin sort.
0052: 47C2                        mov r2,31
0053: E486                        fill b2[r2],r6         ;Set binheads array.
0054: 43C1                        mov r1,15
0055: 0A42        pas3loop        mov r2,b5[r1]
0056: A0C8                        js pas3next
0057: 0083        pas3inn         mov r3,b0[r2]
0058: C04B                        sel3 r3
0059: 04C5                        mov r5,b2[r3]
005A: A029                        jns pas3else
005B: 04CA                        mov b2[r3],r2
005C: 805F                        jmp pas3endi
005D: 08C5        pas3else        mov r5,b4[r3]
005E: 074A                        mov b3[r5],r2
005F: 08CA        pas3endi        mov b4[r3],r2
0060: 068E                        mov b3[r2],r6
0061: 0C82                        mov r2,b6[r2]
0062: AF49                        jns pas3inn
0063: 4059        pas3next        dec r1
0064: AF09                        jns pas3loop
0065:             ; Bin sort done. Place sorted frequencies in sfreqs array.
0065: 4002                        clr r2
0066: 47C1                        mov r1,31
0067: 0443        wsortloop       mov r3,b2[r1]
0068: A058                        js wsortnext
0069: 00C5        wsortinn        mov r5,b0[r3]
006A: 0E8D                        mov b7[r2],r5
```

```
006B: 4052                        inc r2
006C: 06C3                        mov r3,b3[r3]
006D: AFB9                        jns wsortinn
006E: 4059      wsortnext         dec r1
006F: AF79                        jns wsortloop
0070:           ; Now ready for the Lu/Chen Algorithm
0070:           ; Contraction Stage.
0070:           ; i=r1 lp=r2 m=r3
0070:           ; tot=r4 aux=r5
0070: 4041                        mov r1,1
0071: D102                        mov r2,r4
0072: 40DA                        sub r2,3
0073: 4003                        clr r3
0074: D106                        mov r6,r4
0075: 405E                        dec r6
0076: 0F85                        mov r5,b7[r6]
0077: 405E                        dec r6
0078: 0F86                        mov r6,b7[r6]
0079: D195                        add r5,r6
007A: 0E86      contrloop         mov r6,b7[r2]
007B: D1BD                        cmp r5,r6
007C: A0A2                        jb contrelse
007D: 0E80                        mov r0,b7[r2]
007E: D086                        mov r6,r2
007F: D116                        add r6,r4
0080: D05E                        sub r6,r1
0081: 409E                        sub r6,2
0082: 0F88                        mov b7[r6],r0
0083: 4053                        inc r3
0084: 405A                        dec r2
0085: A1E8                        js contrend
0086: 807A                        jmp contrloop
0087: D086      contrelse         mov r6,r2
0088: D116                        add r6,r4
0089: D05E                        sub r6,r1
008A: 409E                        sub r6,2
008B: 0F8D                        mov b7[r6],r5
008C: 0C4B                        mov b6[r1],r3
008D: 4051                        inc r1
008E: D106                        mov r6,r4
008F: 409E                        sub r6,2
0090: D1B9                        cmp r1,r6
0091: A124                        je contrend
0092: D106                        mov r6,r4
0093: D05E                        sub r6,r1
0094: 405E                        dec r6
0095: D1BA                        cmp r2,r6
0096: A065                        jne cont2else
0097: 0E85                        mov r5,b7[r2]
0098: D196                        shl r6
0099: 0F86                        mov r6,b7[r6]
009A: D195                        add r5,r6
009B: 405A                        dec r2
009C: 807A                        jmp contrloop
009D: D196      cont2else         shl r6
009E: 0F85                        mov r5,b7[r6]
009F: 405E                        dec r6
00A0: 0F86                        mov r6,b7[r6]
00A1: D195                        add r5,r6
00A2: 405B                        dec r3
00A3: 807A                        jmp contrloop
00A4: D105      contrend          mov r5,r4
00A5: 409D                        sub r5,2
00A6: 80AA                        jmp cont2next
00A7: 0C4B      cont2loop         mov b6[r1],r3
00A8: 405B                        dec r3
00A9: 4051                        inc r1
00AA: D179      cont2next         cmp r1,r5
00AB: AFB2                        jb cont2loop
00AC:           ; Next the expansion stage.
```

```
00AC:                   ; mcl=r2
00AC: 4042                              mov r2,1
00AD: 108A                              mov b8[r2],r2
00AE: 4052                              inc r2
00AF: 108A                              mov b8[r2],r2
00B0: 4083                              mov r3,2
00B1: D101                              mov r1,r4
00B2: 40D9                              sub r1,3
00B3: A146                              jz expend
00B4: 1085              exploop         mov r5,b8[r2]
00B5: 0C46                              mov r6,b6[r1]
00B6: D17E                              cmp r6,r5
00B7: A063                              jae expelse
00B8: 1086                              mov r6,b8[r2]
00B9: 405E                              dec r6
00BA: 108E                              mov b8[r2],r6
00BB: 4052                              inc r2
00BC: 108B                              mov b8[r2],r3
00BD: 80C6                              jmp expendi
00BE: D085              expelse         mov r5,r2
00BF: 405D                              dec r5
00C0: 1146                              mov r6,b8[r5]
00C1: 405E                              dec r6
00C2: 114E                              mov b8[r5],r6
00C3: 1086                              mov r6,b8[r2]
00C4: 4096                              add r6,2
00C5: 108E                              mov b8[r2],r6
00C6: 4059              expendi         dec r1
00C7: AEC7                              jnz exploop
00C8:                   ;Finally generate the code word lengths.
00C8: 4043              expend          mov r3,1
00C9: 10C5                              mov r5,b8[r3]
00CA: 47C1                              mov r1,31
00CB: 0442              lgenloop        mov r2,b2[r1]
00CC: A098                              js lgencont
00CD: 403D              lgeninn         cmp r5,0
00CE: A035                              jne lgenendwh
00CF: 4053              lgenwhile       inc r3
00D0: 10C5                              mov r5,b8[r3]
00D1: AFD6                              jz lgenwhile
00D2: 028B              lgenendwh       mov b1[r2],r3
00D3: 405D                              dec r5
00D4: 0682                              mov r2,b3[r2]
00D5: AF79                              jns lgeninn
00D6: 4059              lgencont        dec r1
00D7: AF39                              jns lgenloop
00D8: E030                              rts
00D9:
00D9:                   ;Lentocode Compute canonical Huffman codes from lengths.
00D9:                   ;b1 points to array of code word lengths, b7 points to area
00D9:                   ;where code words are stored, r0 is number of code words.
00D9: D007              lentocode       mov r7,r0
00DA: C05E                              set r6
00DB:                   ; First Pass: collect the code word lengths in each in their own
bin.
00DB: 4442                              mov r2,17
00DC: E486                              fill b2[r2],r6
00DD: 4002                              clr r2
00DE: 0283              ltc1loop        mov r3,b1[r2]
00DF: A086                              jz ltc1cont
00E0: 04C4                              mov r4,b2[r3]
00E1: A029                              jns ltc1else
00E2: 04CA                              mov b2[r3],r2
00E3: 80E6                              jmp ltc1endi
00E4: 08C4              ltc1else        mov r4,b4[r3]
00E5: 070A                              mov b3[r4],r2
00E6: 08CA              ltc1endi        mov b4[r3],r2
00E7: 068E                              mov b3[r2],r6
00E8: 4052              ltc1cont        inc r2
00E9: D1FA                              cmp r2,r7
```

```
00EA: AF35                              jne ltc1loop
00EB:                  ; Second Pass: Assign code words to increasing lengths.
00EB: 4004600C                          movd r4,8000h
00ED: 4005                              clr r5
00EE: 4042                              mov r2,1
00EF: 4401                              mov r1,16
00F0: 0483             ltc2loop         mov r3,b2[r2]
00F1: A048                              js ltc2cont
00F2: 0ECD             ltc2inn          mov b7[r3],r5
00F3: D115                              add r5,r4
00F4: 06C3                              mov r3,b3[r3]
00F5: AFC9                              jns ltc2inn
00F6: C06C             ltc2cont         shr r4
00F7: 4052                              inc r2
00F8: 4059                              dec r1
00F9: AF67                              jnz ltc2loop
00FA: E030                              rts
00FB:

SYMBOL TABLE
  BINHEADS D 0050 BINHEADS1 D 0070   BINLIST D 00c0  BINLIST1 D 02c0
  BINTAILS D 0080 CONT2ELSE P 009d CONT2LOOP P 00a7 CONT2NEXT P 00aa
 CONTRELSE P 0087  CONTREND P 00a4 CONTRLOOP P 007a  DIRECTPG D 0000
   EXPELSE P 00be    EXPEND P 00c8   EXPENDI P 00c6   EXPLOOP P 00b4
     FREQS C 0800 FREQTOLEN P 0014INITIALIZE P 0000    ISZERO P 0012
   LENGTHS C 0600 LENTOCODE P 00d9  LGENCONT P 00d6 LGENENDWH P 00d2
   LGENINN P 00cd  LGENLOOP P 00cb LGENWHILE P 00cf  LTC1CONT P 00e8
  LTC1ELSE P 00e4  LTC1ENDI P 00e6  LTC1LOOP P 00de  LTC2CONT P 00f6
   LTC2INN P 00f2  LTC2LOOP P 00f0    NCLASS D 00a0  PAS1CONT P 003c
   PAS1ELS P 0038  PAS1ENDI P 003a  PAS1LOOP P 0030  PAS2ELSE P 004a
  PAS2ENDI P 004c   PAS2INN P 0044  PAS2LOOP P 0042  PAS2NEXT P 0050
  PAS3ELSE P 005d  PAS3ENDI P 005f   PAS3INN P 0057  PAS3LOOP P 0055
  PAS3NEXT P 0063    SFREQS D 0c00     STACK D 0040    STAKTOP D 0050
  WSORTINN P 0069 WSORTLOOP P 0067 WSORTNEXT P 006e
```