Modelling IDaSS elements in POOSL

Michielsen, R.

Award date:
1997

ICS-EB 661

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Information- and Communication Systems Group

Master's Thesis:

Modelling IDaSS elements in POOSL

By ing. R. Michielsen

Coaches	:	dr.ing. P.H.A. van der Putten
		dr.ir. J.P.M. Voeten
Supervisor	:	prof.ir. M.P.J. Stevens
Period	:	January 1997 – October 1997

Abstract

Software/Hardware Engineering (SHE) is a new object-oriented method for the co-specification and design of complex reactive hardware/software systems. SHE incorporates a framework for design activities, and a formal description language called POOSL (Parallel Object-Oriented Specification Language). Starting from informal object-oriented analysis, SHE produces rigorous system-level behaviour and architecture descriptions expressed in the POOSL language.

This thesis describes the exploration of the path from IDaSS (Interactive Design and Simulation System) towards POOSL. It addresses the modelling of IDaSS elements in POOSL on the basis of two IDaSS designs. The first IDaSS design uses one Algorithmic Level block to describe an 8048 microprocessor. The second design describes the same microprocessor by means of Register Transfer Level blocks. It will be fairly easy to convert IDaSS designs towards POOSL if we can make a general POOSL specification of these RTL blocks. This way we will obtain a very suitable environment for the co-simulation of hardware/software systems.

Acknowledgements

I am indebted to many people for their advice, assistance and contributions to my master's project. First of all I would like to thank my coaches Jeroen Voeten and Piet van der Putten for their support and confidence and for giving me the opportunity to contribute to their research project. I thank my supervisor prof. Mario Stevens for his interest and for introducing me to Piet and Jeroen.

I also wish to thank Ad Verschueren for his contributions and his help with IDaSS. In particular, I wish to thank Marc Geilen for solving the problems I had with the POOSL Simulator and for his major assistance in my master's project. The past nine months would not have been the same without the companionship of Emil Reuter and Terry Dennemans. I would like to thank both of them for their friendship and support. I am also grateful to all other members and (ex-) students of the Information- and Communication Systems Group.

Finally I would like to thank my family for their love, support and encouragement.

Contents

1. Introduction

1.1 Project overview

Software/Hardware Engineering (SHE) [vdPV97] is a new object-oriented method for the co-specification and design of complex reactive hardware/software systems. SHE incorporates a framework for design activities, and a formal description language called POOSL (Parallel Object-Oriented Specification Language). Starting from informal object-oriented analysis, SHE produces rigorous system-level behaviour and architecture descriptions expressed in the POOSL language.

This report describes the exploration of the path from IDaSS [Ver90, Ver92, Ver97] towards POOSL [Voe94, Voe95a, Voe95b]. It addresses the modelling of IDaSS elements in POOSL on the basis of two IDaSS designs. The first IDaSS design uses one Algorithmic Level block to describe an 8048 microprocessor. The main reason why this design was modelled in POOSL was to get familiar with IDaSS, POOSL and the 8048 microprocessor. The second design describes the same microprocessor by means of Register Transfer Level blocks. If we can make a general POOSL specification of these blocks, then it will be fairly easy to convert IDaSS designs towards POOSL. This way we will obtain a very suitable environment for the co-simulation of hardware/software systems.

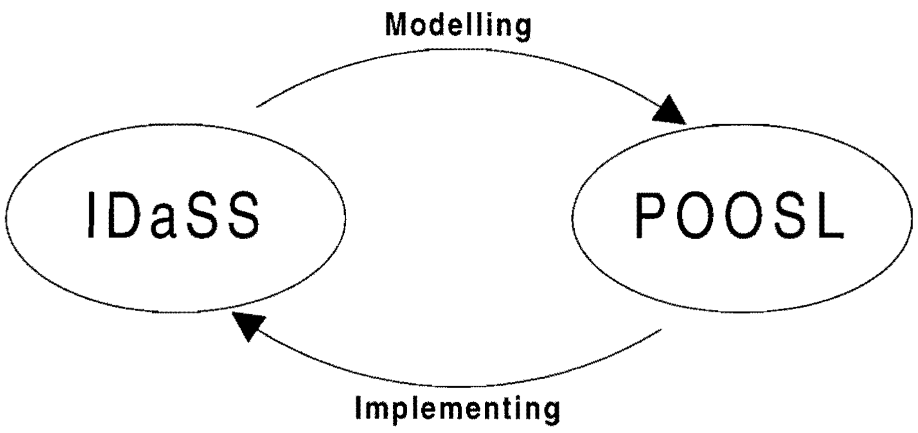


Figure 1.1: Overview of the whole project

The project described in this report is the first part of a whole project in which the paths between IDaSS and POOSL are explored (Figure 1.1 gives a brief overview of the whole project). The second part of the project (the implementation of POOSL in IDaSS) is described in [Mic97].

1.2 IDaSS

IDaSS is an Interactive Design and Simulation System for digital circuits, targeted towards VLSI and ULSI designs of complex data processing hardware [Ver90, Ver92, Ver97]. IDaSS describes a design as a hierarchy of schematics. A schematic contains elements like registers, RAM's, ROM's, combinatorial operators, Finite State Machine controllers and Algorithmic Level blocks.

IDaSS provides a direct path to hardware: the design files can be converted into a format suitable as input for silicon compilers. Synthesizable VHDL output is available, although the converter is still under construction.

Several designs were created with IDaSS. Examples include, but are not limited to, a token ring controller, several processor cores (including 'simple' processors as the Intel 8048 as well as super-scalar processors), a Texas Instruments 32010 DSP, an LCD controller, a PCM switching network and different cache-designs.

1.3 POOSL

POOSL [Voe94, Voe95a, Voe95b] is short for **P**arallel **O**bject-**O**riented **S**pecification **L**anguage. The following introductory information about POOSL was taken from [VvdPS].

POOSL is a new system-level modelling, specification and design language. The key feature of POOSL is the expressive power to model very complex dynamic (communication and functional) behaviour as well as static (architecture and topology) structure in an object-oriented fashion.

The language combines a process part with a data part. The process part is based on the key ideas of the process algebra CCS. The data part is bases upon the concepts of traditional sequential object-oriented programming languages such as Smalltalk, C++ and Eiffel.

In POOSL very complex reactive real-time behaviour is represented by a collection of asynchronous concurrent process objects that communicate synchronously or asynchronously by passing messages over static channels. To describe complex functional behaviour, POOSL supports data objects. Data objects have a sequential behaviour and communicate by synchronous message passing. They are contained in processes and they model the private (non-shared) data of these processes.

In this thesis it is assumed that the reader is familiar with the characteristics of POOSL. An in-depth explanation of the language POOSL is given in [vdPV97].

2. The 8048 microprocessor

The IDaSS 8048 designs (AL8048.DES and UP8048N.DES) are based on the 8048 industry standard 8-bit microprocessor. This section gives an overview of the architecture and instruction set of this processor. The following information was taken from [Phi94]], which describes the Philips MAB84XX family and from [Int93], which describes the Intel MCS-48 family. Figure 2.1 shows a block diagram of the 8048.

2.1 Arithmetic Section

The arithmetic section of the processor contains the basic data manipulation functions of the 8048. It can be divided into the following blocks:

- Instruction decoder: decodes the instructions and supplies control signals to several ports of the microprocessor.
- Arithmetic Logic Unit (ALU): the ALU accepts 8-bit data words from one or two sources and generates an 8-bit result under control of the instruction decoder. The ALU can perform the following functions:
  - ADD (with or without carry);
  - AND, OR, XOR;
  - increment / decrement;
  - bit complement;
  - rotate left, right;
  - swap nibbles;
  - BCD decimal adjust.
- Carry flag: if the operation performed by the ALU results in a value represented by more than 8 bits (overflow of the most significant bit), a carry flag is set in the program status word.
- Accumulator: in most operations this is the source or destination register, therefore it is the most important data register in the processor.

2.2 Program memory

The 8048 contains 1024, 2048 or 4096 8-bit words of on-chip read-only memory (ROM). Each location is directly addressable by the program counter. Three program memory locations are of special importance:

- Location 0: first instruction to be executed after the processor is reset.
- Location 3: first instruction of an external interrupt service routine.
- Location 7: first instruction of a timer/event counter interrupt service routine.

Figure 2.2 shows the program memory map.

2.3 Data memory

The 8048 contains 64, 128 or 256 bytes of random access data memory (RAM). All locations are indirectly addressable using RAM pointer registers, up to 16 designated locations are directly addressable (register banks 0 and 1). Another 16 bytes are designated to an 8-level program counter stack addressed by a 3-bit stack pointer (see Figure 2.2 for a complete map of the data memory).

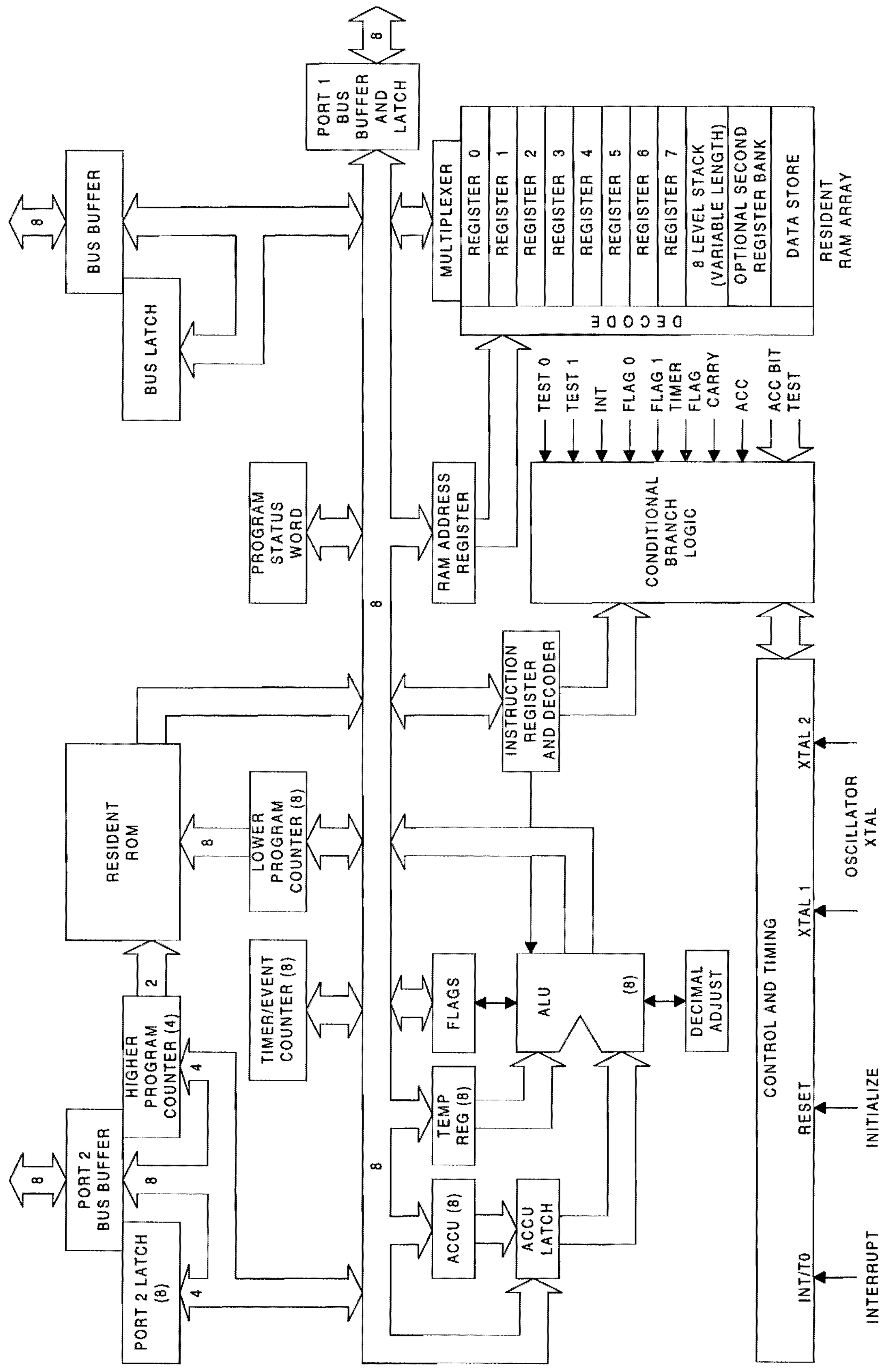


Figure 2.1: 8048 block diagram

2.3.1 Registers R0 to R7

Registers R0 to R7 are directly addressable by direct register instructions. Ease of addressing, and a minimum requirement of instruction bytes to manipulate their contents, make these locations suitable for storing frequently addressed intermediate results.

Executing the SEL RB0 (SELect Register Bank 0) instruction designates R0-R7 to data memory locations 0-7. Executing the SEL RB1 instruction designates R0-R7 to data memory locations 24-31. This second register bank may be used as an extension of the first, or it may be reserved for use during interrupt service routines leaving the first bank available for the main program.

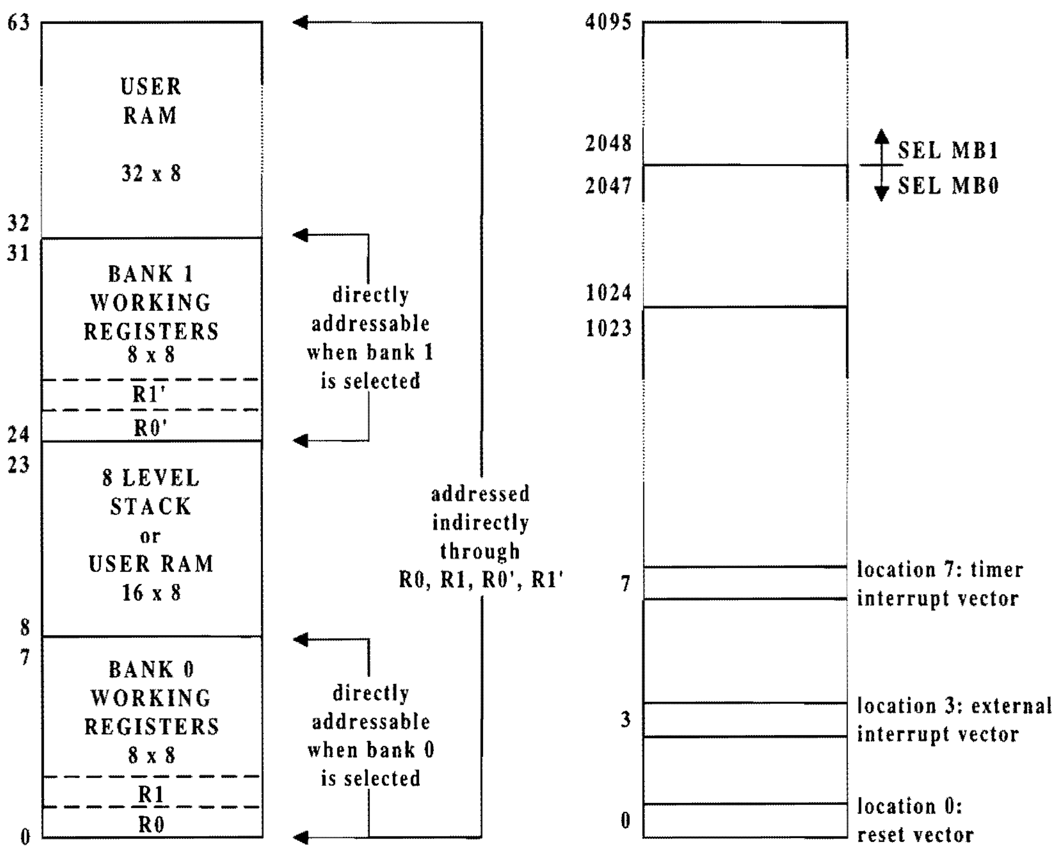

63
USER RAM 32 x 8
32
31
BANK 1 WORKING REGISTERS 8 x 8
R1'
R0'
24
23
8 LEVEL STACK or USER RAM 16 x 8
8
7
BANK 0 WORKING REGISTERS 8 x 8
R1
R0
0
directly addressable when bank 1 is selected
addressed indirectly through R0, R1, R0', R1'
directly addressable when bank 0 is selected
4095
2048
2047
SEL MB1
SEL MB0
1024
1023
7
location 7: timer interrupt vector
3
location 3: external interrupt vector
0
location 0: reset vector

Figure 2.2: Data memory map & program memory map

2.3.2 Stack

Locations 8 to 23 of the data memory are designated to an 8-level program counter stack (2 locations per level). A 3-bit stack pointer points to the next free location on the stack.

During a subroutine CALL or interrupt, the contents of the 13 bit program counter and bits 4-7 of the program status word are transferred to the stack. The stack pointer is then incremented. During a RET or RETR instruction, the 13 bit program counter is restored and the stack pointer is decremented. The RETR instruction restores the saved bits of the program status word.

2.3.3 User RAM

Data memory locations 32 to 63 are designated as user RAM and are indirectly addressable with the RAM pointers R0, R1 or R0', R1'. Unused register and stack locations are also available as user RAM and are addressed in the same way.

2.4 Program counter

The 13-bit program counter can address up to 8 K bytes of ROM. The least-significant 11 bits are auto-incrementing. The two most significant bits are used for selecting the memory bank.

2.5 Program status word

The program status word (PSW) is an 8-bit CPU register that stores information about the current status of the microprocessor. The ability to write to PSW allows for easy restoration of machine status after a power down sequence. The PSW bit definitions are as follows:

- bits 0-2: stack pointer bits;
- bit 3: not used;
- bit 4: working register bank switch bit;
- bit 5: flag 0 bit, user controlled flag;
- bit 6: auxiliary carry bit;
- bit 7: carry flag.

2.6 Conditional branch logic

The conditional branch logic within the processor enables several conditions internal and external to the processor to be tested by the users program.

2.7 Interrupt logic

The 8048 handles external and timer/event counter interrupts. The interrupt mechanism is single level; an executing interrupt service routine can not be interrupted by other interrupts. An interrupt request will only be serviced if the interrupt is enabled (both interrupt types have individual enable and disable instructions).

2.8 Timer / event counter

The 8048 contains a counter to aid the user in counting external events (counter mode) and generating accurate time delays (timer mode) without placing a burden on the processor for these functions. In both modes the counter operation is the same, the only difference being the source of the input to the counter.

2.9 Input / Output

The 8048 has 27 lines that can be used for input or output functions. These lines are grouped as 3 ports of 8 lines each, which serve as either inputs, outputs or bi-directional ports and 3 "test" inputs that can alter program sequences when tested by conditional jump instructions.

2.10 Instruction set

The 8048 instruction set consists of over 80 one and two byte instructions that execute in either one or two cycles. Double cycle instructions include all immediate instructions, and all I/O instructions. The instruction set can be divided into the following groups:

- data transfers;
- accumulator operations;
- register operations;
- flags;
- branch instructions;
- subroutines;
- timer instructions;
- control instructions;
- input / output instructions.

A summary of the MCS-48 instruction set is given in Appendix A.

3. Bounded Integer data class

IDaSS provides the designer with only one data type, namely the Bounded Integer. Bounded Integers are (unsigned) integers with a specific fixed number of bits. For an optimal 'conversion' from IDaSS to POOSL, this data class may not be missing. This section gives an overview of the Bounded Integer data class, and shows how IDaSS operators can be expressed in POOSL data methods.

A Bounded Integer data object uses two instance variables to store the relevant data for the object:

- *val*: used to store the actual value of the object;
- *width*: used to store the number of bits the object has.

Both variables belong to the primitive data class 'Integer'. A Bounded Integer with a *width* equal to zero is called a 'Bounded Integer Constant'.

IDaSS provides the following function classes with basic operators:

- standard integer arithmetic;
- logical operations;
- (un)signed comparisons;
- shifts, rotates and butterflies;
- priority encoding;
- (de-)multiplexing and merging;
- counting bits in a word;
- concatenation of words;
- bit (field) extraction;
- width manipulation and checking;
- constant generation;
- special tests (only in probes).

The complete POOSL-specification of the Bounded Integer data class can be found in Appendix B. The following paragraphs show how some of the IDaSS operators were converted into POOSL data methods.

3.1 Increment operator

The increment operator is an unary arithmetic operator. It increments the receiver (the value to the left of the operator) by adding one with wrap around.

%00010100 **inc** *gives:* %00010101
%11111111 **inc** *gives:* %00000000

IDaSS is written in Smalltalk, this makes it quite easy to convert the IDaSS operators to POOSL data methods. First of all let us take a look at the Smalltalk code of the increment operator. It is shown in Figure 3.1.

```
1. inc
2.   width = 0
3.     ifTrue:
4.       [^BoundedInteger new value: ConstError width: 0]
5.   ^BoundedInteger new
6.     value:
7.       (val < 0
8.         ifTrue:
9.           [val min: UNK]
10.        ifFalse:
11.          [(val + 1) bitAnd: (Masks at: width)])
12.    width:
13.      width!
```

Figure 3.1: Increment operator (Smalltalk code)

The Smalltalk code contains a lot more functionality then we actually need. The major part of the code is used to check for errors (checks whether the receivers value is unknown and whether the receiver is a Bounded Integer Constant). The most important lines of this operator are at the end of the specification (lines 11-13). In line 11 the current value is incremented by one (**val + 1**) and the result is checked for overflow (**bitAnd: (Masks at: width)**).

The POOSL specification of the Bounded Integer class contains no error checking. Therefore the increment-method is very compact (Figure 3.2). The comparison in line 2 is needed to set the result to zero if the current value is the same as the maximum value (**2 power(width) - 1**).

```
1. inc() : BoundedInteger
2.   if val=((2 power(width)) - 1) then
3.     val := 0
4.   else
5.     val := val + 1
6.   fi;
7.   return(self).
```

Figure 3.2: Increment operator (POOSL code)

3.2 Concatenation operator

The binary operator "," concatenates the receiver word with the word on the right hand side, returning an integer with a width that is the sum of the widths of the receiver and the right hand side. The result's most significant bits will come from the receiver and the least significant bits from the right hand side word. Neither side can be a constant.

Here is an example that shows what the concatenation operator does:

%1100 , %0011   *gives:* %11000011

In Figure 3.3 the Smalltalk code of the concatenation operator is shown.

```
1. , aBoundedInteger
2. | otherWidth otherValue newWidth |
3.   ((otherWidth := aBoundedInteger width) = 0 or: [width = 0])
4.      ifTrue:
5.         [^BoundedInteger new value: ConstError width: 0].
6.   (newWidth := width + otherWidth) > MaxWidth
7.      ifTrue:
8.         [^BoundedInteger new value: WidthOflo width: MaxWidth].
9.   ^BoundedInteger new
10.     value:
11.        (((otherValue := aBoundedInteger val) < 0 or: [val < 0])
12.           ifTrue:
13.              [(otherValue min: val) min: UNK]
14.           ifFalse:
15.              [(val bitShift: otherWidth) bitOr: otherValue])
16.     width:
17.        newWidth!
```

Figure 3.3: Concatenation operator (Smalltalk code)

In lines 3-5 the widths of the two Bounded Integers are checked. If one of them has zero width an error is returned. Lines 6-8 checks if the new width is above the maximum allowed width (called **MaxWidth**). When this is the case an error is returned. In lines 11-13 both Bounded Integers are checked on negative values[1]. Again an error is returned if one of the values is negative.

Again most code consists of error checking. Only one line contains the actual concatenation-operation (line 15). The first part of this line (**val bitShift: otherWidth**) shifts the receiver to the left, thereby introducing "otherWidth" number of zeroes in the least significant bits. The second operation in this line performs a logical or operation between the result of the first part and the value of the right hand side.

In POOSL the concatenation operator is called "concat", since POOSL does not allow special characters in method names (like a comma). If we leave out the error checking, we get the specification as shown in Figure 3.4. At the end of the POOSL code a Bounded Integer is returned with [**width + otherWidth**] number of bits. The value of the Bounded Integer is determined by multiplying the value of the receiver by $2^{\text{otherWidth}}$ and finally by adding the result of that operation to the value of Bounded Integer on the right hand side.

```
concat(aBI : BoundedInteger) : BoundedInteger
| otherVal, otherWidth : Integer |
   otherVal := aBI getVal();
   otherWidth := aBI getWidth();
   return(new(BoundedInteger) init(2 power(otherWidth) * val + otherVal,
      width + otherWidth)).
```

Figure 3.4: Concatenation operator (POOSL code)

[1] The instance variable '*val*' of a Bounded Integer can have a negative value. Negative values have a special meaning in the Smalltalk-code of IDaSS. For instance: -1 stands for three-state (TS), -2 stands for overload (OVL) and -3 stands for unknown (UNK). Other negative values are mostly used to refer to error messages.

3.3 Bit extraction operator

The bit extraction operator 'from:to:' returns a BoundedInteger extracted from the receiver bits. The extraction starts at the bit specified by the 1st parameter and ends with the bit specified by the 2nd parameter. The result has [2nd parameter - 1st parameter + 1] number of bits. Both parameters must be constants with the following bounds: 0 <= 1st parameter <= 2nd parameter < receiver width.

```
1. from: fromBoundedInteger to: toBoundedInteger
2.   | from to newWidth |
3.   from := fromBoundedInteger val.
4.   to := toBoundedInteger val.
5.   (fromBoundedInteger width > 0 or:
6.     [toBoundedInteger width > 0 or:
7.       [from < 0 or:
8.         [to < from] ] ])
9.           ifTrue:
10.            [^BoundedInteger new
11.              value: ((ConstError min: from) min: to) width: 0].
12.  (width > 0 and: [to >= width])
13.      ifTrue:
14.        [^BoundedInteger new value: ConstOflo width: 0].
15.  to > MaxWidth
16.      ifTrue:
17.        [^BoundedInteger new value: WidthOflo width: MaxWidth].
18.  newWidth := to - from + 1.
19.  ^BoundedInteger new
20.      value:
21.        (val < 0
22.          ifTrue:
23.            [val min: UNK]
24.          ifFalse:
25.            [(val bitShift: from negated) bitAnd:
26.            (Masks at: newWidth)])
27.      width:
28.        newWidth!
```

Figure 3.5: Bit extraction operator (Smalltalk code)

Here is an example of what the bit extraction operator does with a Bounded Integer:

%00001111 **from:** 2 **to:** 5*gives:* %0011.

Figure 3.5 shows what the Smalltalk code of this operator looks like. The actual bit extraction operation is executed in lines 25, 26. First of all the receiver's value is shifted to the right by the amount of bits equal to the 1st parameter. Then an *all-ones* mask is generated containing the amount of bits equal to the 2nd parameter. Finally a logical AND operation of the 'shift-result' and the mask gives the desired value.

The POOSL data method "fromTo" works more or less the same way as its IDaSS / Smalltalk counterpart (see Figure 3.6). Line 4 corresponds to the shift right operator in Smalltalk. Line 5 contains the code to determine the all-ones bitmask and to execute the logical AND function.

```
fromTo(from,to : Integer) : BoundedInteger
| anInt, newWidth : Integer |
    newWidth := to - from + 1;
    anInt := val div(2 power(from));
    anInt := anInt & (2 power(newWidth) - 1);
    return(new(BoundedInteger) init(anInt, newWidth)).
```

Figure 3.6: Bit extraction operator (POOSL code)

4. Algorithmic Level description of the 8048

An IDaSS 'Algorithmic Level' language block (AL block) allows the execution of programs at an algorithmic level (comparable to the way programs are written in normal structured languages like Pascal), while still providing means of communication with a register transfer level environment (the programs can test and control the entities placed in this environment). AL blocks are normally used as precursor for normal datapath/controller design, to figure out which algorithms and clock cycle timing must be used.

4.1 IDaSS AL 8048 design

The IDaSS design file 'AL8048.DES' contains an almost complete design of an 8048 processor, using a single Algorithmic Level block for the actual processor core. Figure 4.1 shows the schematic of the AL 8048. The block with the double lined border named '*CORE*' is the AL block. The logic outside this AL block provides the I/O ports, interrupt and timer logic, program memory and data memory.

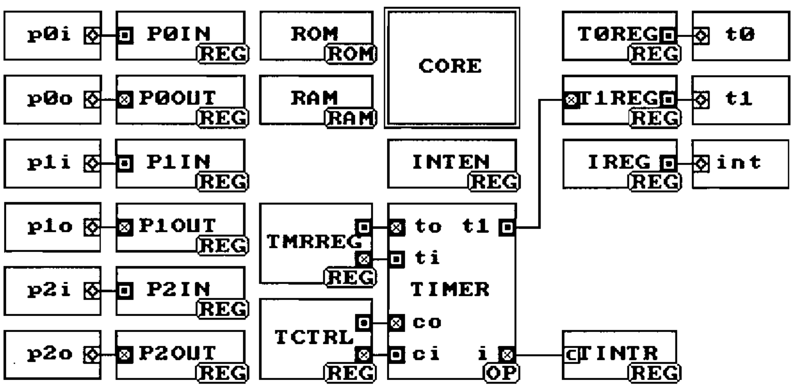


Figure 4.1: IDaSS schematic of AL 8048

Local storage within the AL block takes the form of 9 local registers. Writing to these local entities from within the routines is immediate, there is no need to wait for the clock. The AL block has the following local variables:

- *_PCLOW*, *_PCHIGH*, *_PCBUF*: program counter and buffer;
- *_ACCU*: accumulator;
- *_PSW*: program status word;
- *_F1*: flag;
- *_IR*: instruction register;
- *_TEMP8*, *_TEMP9*: temporary variables.

The AL block contains a large main routine to execute the program in the 8048 processor core and several local subroutines that perform sub-tasks. The structure of the main routine is shown in Figure 4.2. The first stage of the endless loop checks for *interrupts* and handles them if there are any. The second stage *fetches* a new instruction from the program memory and puts it in the instruction register. The two last stages actually consist of one big **case** statement that determines which instruction is in the instruction register (*decode*) and that calls the proper subroutines to *execute* the instruction.

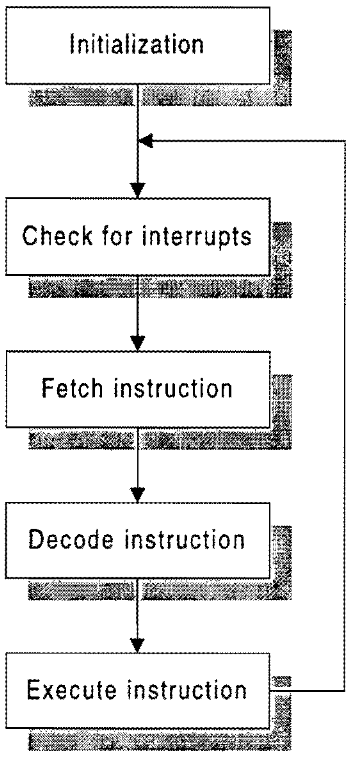


Figure 4.2: AL 8048 main routine

4.2 AL 8048 POOSL specification

Appendix C contains the complete POOSL specification of the AL 8048. The 'conversion' of the AL 8048 from IDaSS to POOSL is quite straightforward.

4.2.1 Structure of the main-method

The AL block is converted to a POOSL process object with almost exactly the same structure as the IDaSS counterpart (Figure 4.2). The only difference is that the decode-stage has been divided into several sub-stages for speed improvement.

Opcode	Process method
\$x0, \$x1	indirectInstr
\$x2	timerInstr
\$x3	varInstr
\$x4	callOrJmpInstr
\$x5	flagInstr
\$x6	branchInstr
\$x7	accuInstr
\$x8 .. \$xF	registerInstr

Table 4.1: Instruction groups for decoding

Although POOSL's **select** statement comes very close to the **case** statement in IDaSS, it has one drawback. The **select** statement is very slow if it contains too many choices. Instead of one large **case** statement, the decode-stage now contains a lot of nested

if..**then** statements. For the decode-stage 8 different groups of instructions have been formed. The groups are based on the 4 least significant bits (first hexadecimal character) of the opcode (see Table 4.1).

4.2.2 Data statements

As (almost) all IDaSS operators have been converted to POOSL data methods (see Chapter 3) it is very easy to convert the data operations in the AL block into POOSL code. Here are some examples that show how some of the IDaSS data operations have been converted to POOSL data statements:

From the local subroutine '_JumpLong':

- IDaSS: *_PCLOW* := (*_IR* **from:** 5 **to:** 7), *_TEMP8*
- POOSL: *pclow* := *ir* **fromTo**(5,7) **concat**(*temp8*).

From the local subroutine '_AddToAccu':

- IDaSS: *_TEMP9* := *_ACCU* **add:** *_TEMP8* **cin:** (*_IR* **at:** 4) Λ (*_PSW* **at:** 7)
- POOSL: *temp9* := *accu* **addCin**(*temp8*, *ir* **at**(4) **logicAND**(*psw* **at**(7))).

From the local subroutine '_FetchRegister':

- IDaSS: *_TEMP8* := *RAM* @ ((2 **copiesOf:** (*_PSW* **at:** 4)), (*_IR* **from:** 0 **to:** 2))
- POOSL: *temp8* **setVal**(*ram* **get**(new(BoundedInteger) **copiesOf**(2, *psw* **at**(4)) **concat**(*ir* **fromTo**(0,2)) **getVal**() + 1)).

4.2.3 Clock

Assignments to external registers and RAM's in an AL block are clocked. If we want the POOSL specification to do exactly the same as the IDaSS design, we need some sort of clock-mechanism. In this case a simple **delay** statement suffices. In addition, if the timer / counter is enabled then it has to be incremented each clock cycle. The process method 'waitForClock' in the AL8048 process takes care of this clock-mechanism.

4.2.4 Program memory

In IDaSS it is possible to 'program' a ROM by loading its contents from a special file. In POOSL it is not possible (yet) to fill the contents of an array with the contents of a file. This means that if we want to put a new program for the 8048 into the ROM array (program memory) we have to do this manually. Figure 4.3 shows how a (useless) program is converted from Assembly to hexadecimal data.

```
        JMP   Start

Start:  MOV   A, 5
Decr:   DEC   A
        JNZ   Decr
        JMP   Start
```

⇒

```
$00:  JMP   $0A

$0A:  MOV   A, 5
$0C:  DEC   A
$0D:  JNZ   $0C
$0F:  JMP   $0A
```

⇒

```
$00:  $04
$01:  $0A
$0A:  $23
$0B:  $05
$0C:  $07
$0D:  $96
$0E:  $0C
$0F:  $04
$10:  $0A
```

Figure 4.3: Simple 8048 program

The process method ‘initRom’ must now put this program into the ROM array (Figure 4.4 shows how this is done).

```
initRom()()
   rom put(1, $04);
   rom put(2, $0A);
   rom put(11, $23);
   rom put(12, $05);
   rom put(13, $07);
   rom put(14, $96);
   rom put(15, $0C);
   rom put(16, $04);
   rom put(17, $0A).
```

Figure 4.4: Process method 'initRom'

4.2.5 Input / Output

In the IDaSS design of the AL 8048, the logic outside the AL block provides the I/O ports, interrupt and timer logic, program memory and data memory. In the POOSL specification of the AL 8048, almost all these functions are available in the main process object ‘AL8048’. The I/O ports are specified by single process objects named ‘IO_8bits’ and ‘Input_1bit’. These separate processes are only needed to assure a continuous message flow from and towards the I/O ports.

5. Register Transfer Level description of the 8048

5.1 IDaSS RTL 8048 design

Figure 5.1 shows the IDaSS Register Transfer Level design of the 8048. Please note that this is not a complete 8048 processor as the I/O ports are missing from the design. The most important element in this design is the state machine '*CONTROL*' that is used to control the other elements of the design.

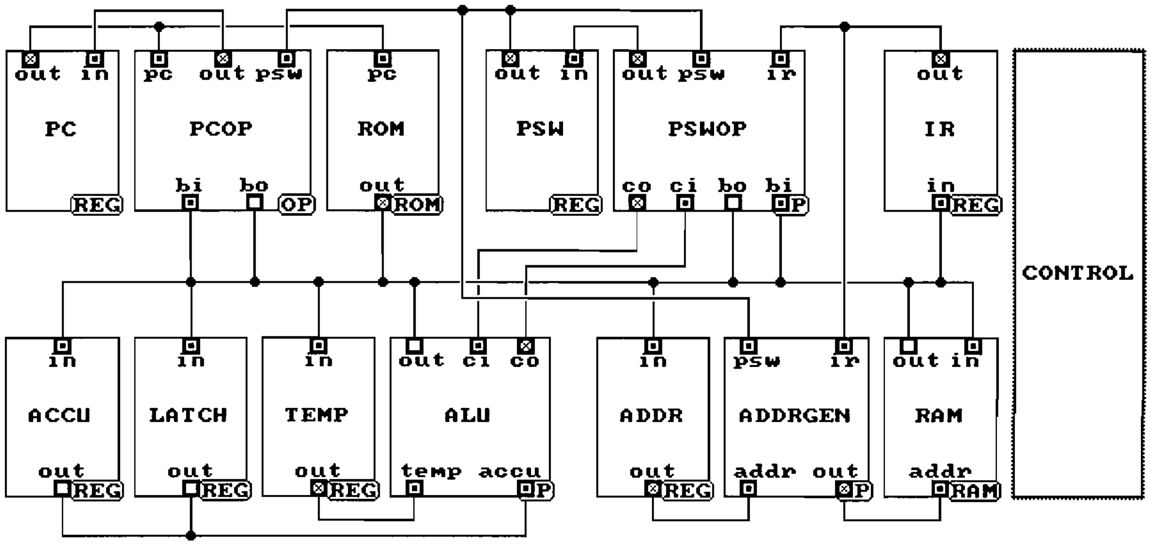


Figure 5.1: IDaSS schematic of RTL 8048

Here are the functions of the other elements in the design:

- *PC*: a 12 bits wide register used as Program Counter;
- *PCOP*: an operator that is used to increment and modify the Program Counter;
- *ROM*: a ROM for the program memory;
- *PSW*: an 8 bits wide register used as Program Status Word;
- *PSWOP*: an operator that is used to modify the Program Status Word;
- *IR*: an 8 bits wide register used as Instruction Register;
- *ACCU*: an 8 bits wide register to store the main accumulator;
- *LATCH*: an 8 bits wide register that is used when an ALU operation should not modify the accumulator;
- *TEMP*: an 8 bits wide register used to hold a possible second ALU operand;
- *ALU*: an operator that provides the main arithmetic and logic unit (ALU);
- *ADDR*: an 8 bits wide register used to temporarily store indirect addresses;
- *ADDRGEN*: an operator to generate the addresses for the working memory RAM;
- *RAM*: a RAM for the working memory.

The connections between these elements are databuses, used to transport data values from one block to another. Next to the visible busses shown in the schematic, IDaSS allows invisible test and control channels to be used to give commands or receive information from system elements.

5.2 RTL 8048 POOSL specification

The first problem we encounter when specifying the RTL 8048 in POOSL is how the visible buses in IDaSS can be represented in POOSL. We can do this by specifying a single process object for each bus, or by stating that a POOSL channel represents a bus. The second option allows the design of a schematic that is (visually) very similar to the one in IDaSS, but its functionality is a lot less powerful than the one provided by the first option.

If we use a process object for the databus it is possible to check the value of the databus during a simulation of the system (this is not possible if we use a channel to represent the bus). In addition it is possible to check if a bus is three-state (no driving outputs) or overloaded (more than one output driving it). In order to perform these checks the databus must know which outputs are connected and it must know the values of these outputs (so again this is not possible if we just use a POOSL channel to represent the bus).

Both options were tested and eventually the 'channel-option' was chosen for performance reasons. The extra functionality of the 'process object-option' is not needed to obtain a correct specification of the RTL 8048. It is only helpful during the simulation of the system. The 'channel-option' is the faster and the simpler and its visual aspects are similar to those of its IDaSS counterpart.

5.2.1 Clock

Just as in the case of the POOSL specification of the AL 8048, we require a clock mechanism for the IDaSS clock. The current model uses three phases to represent one clock step, although two phases may suffice. These different phases are necessary to assure that certain information from one element is available at a certain time for the other elements.

We need at least two phases to represent one clock step because two specific tasks in a register-object may not be performed at the same time. The first task is the execution of the current command and the calculation of the new register value. The second task is to send the new value to the register output and to wait for new commands. The reason why these tasks may not be performed at the same time will be explained in paragraph 5.2.4.

A third phase is needed if we represent an IDaSS bus by a single process object. In this third phase the contents of a databus are updated (checked for overload and three-state)[1].

Figure 5.2 shows what the '**main**' method of the process class '*Clock*' looks like. The **delay** statements are used for synchronisation to make sure that every process object that may want to receive broadcast messages will actually receive them.

[1] Although the current POOSL specification of the RTL 8048 does not use a single process object to represent an IDaSS databus, it does still use three phases. This is because the old specification did use process-objects as databuses and when the model was changed, the three phases were left intact. Lack of time has prevented me from changing the POOSL specification to a two phase model.

One channel named ‘clock’ is used to distribute the clock messages over the whole system. Another special channel in the POOSL specification of the RTL 8048 is the ‘cmd’ channel. It is mainly used by the *CONTROL* state machine to send control messages to the other elements of the system and to test the values of these elements.

```
main()()
    delay(1);
    clock !* beforeClock;
    delay(1);
    clock !* clockPulse;
    delay(1);
    clock !* afterClock;
    main()().
```

Figure 5.2: Three-phase clock mechanism

5.2.2 Input

An IDaSS input connector is used to let data values enter the block in which the input connector is placed. In the POOSL specification a process object is defined that represents a normal input connector.

The process class ‘*Input*’ has one instance variable that stores the current value (the last received value) of the input. An input-object has two communication channels. The first one (‘toCore’) connects the object to the block in which the input connector is placed (for instance a register- or an operator-object). The second channel (‘input’) can connect the input-object to one or more output-objects (this channel is the representation of an IDaSS databus).

An input-object has two tasks:
1. Receive new values from the connected output-objects.
2. Send the current value to the block-object to which the input-object is connected.

The instance methods **‘init’** and **‘main’** perform these two tasks (Figure 5.3).

```
init()()
    contents := -3;
    main()() interrupt( input ? value(contents) ).

main()()
    toCore ! value(name, contents);
    main()().
```

Figure 5.3: POOSL specification of a normal input

The initial method call ‘init’ contains the first task. The **interrupt** statement is used to make sure that the new value sent by an output-object will be received at any instant. The second task is performed in the ‘main’ method. The send statement has two parameters, the data objects *name* and *contents*. So besides the current value also a name-tag is sent: as some blocks may have more than one input connected to its input-channel (for example an operator can have several inputs), a block must be able to recognise where the current message is coming from.

In IDaSS it is also possible for a state machine to test the value present on the bus connected to an input connector. This feature is not included in the POOSL specification, mainly because it is not needed in the RTL 8048 design, but it can be easily added if needed.

Some blocks may require a special kind of input connector. Normally a RAM will contain at least one write port. A write port on a memory has an address input connector that determines which word of the memory will be written and another input connector that determines the value that will be written there. The address input is the same as the normal input that was already specified, but the data input is not. The data input can also receive 'write' and 'nowrite' commands, which define the status of the write port. This means that the POOSL specification of a normal input as shown in Figure 5.3 must be extended to the specification as shown in Figure 5.4.

```
init()()
    contents := -3;
    write := defaultWrite;
    main()() interrupt( input ? value(contents) ).

main()()
| dest, stateCmd : String |
    sel
        cmd ? writePortState(dest, stateCmd | dest=name);
        if stateCmd = "write" then
            write := true
        else
            write := false
        fi
    or
        clock ? clockPulse;
        if write then
            toCore ! value(name, contents)
        else
            toCore ! value(name, -3)
        fi;
        write := defaultWrite
    les;
    main()().
```

Figure 5.4: POOSL specification of a data input (for a write port)

Besides the instance variable 'contents', this process class (called 'WritePortInput') also contains the instance variable 'write' that stores the current status of the write port. The task of the 'init' method remains the same, but the 'main' method now contains a **select** statement with two choices. The first choice allows the reception of a new status command ('write' or 'nowrite' command). The second choice allows the reception of the message 'clockPulse'. When this message is received, the current value is sent to the connected memory-block if the write port is in write-status (after that the status of the write port is set to the default status).

Another special input port is the control-input. A control-input is used to provide very localised control for a single IDaSS element (the element it is placed in). In essence, a control-input translates the values that are present on the bus it is connected to into commands for the block it is placed in. As control-inputs are not used in the RTL

8048 design, a POOSL specification has not been made (control-inputs can be implemented in POOSL, but this may require some changes to other process classes).

5.2.3 Output

An IDaSS output connector is used to let a block place data on a bus. There are two different kinds of outputs:

1. normal (continuous) output connector,
2. three-state output connector.

For the POOSL specification the main difference between these two outputs is the fact that the three-state output can also receive commands that define the state of the output (enabled or disabled). The process class '*Output*' contains the specification of both outputs. The type of output that is needed can be defined in the instantiation parameter 'TS' (Boolean). For a three-state output, the default output-state must also be defined. This must be done by setting the instantiation parameter 'defaultDisabled' to <u>true</u> (output is disabled by default) or <u>false</u> (output is enabled by default). The initial method '**init**' is shown in Figure 5.5.

```
init()()
    contents := -3;
    newValue := 0;
    disabled := defaultDisabled;
    fullOutputName := blockName concat(":") concat(name);
    delay(1);
    if TS then
        threeState()()
    else
        continuous()()
    fi.
```

Figure 5.5: Initial method for the Output class

An output-object has four instance variables:

1. *contents*: to store the current value of the output;
2. *newValue*: a temporary variable;
3. *disabled*: holds the actual state of the output (only used for TS-output);
4. *fullOutputName*: used for identification of the output.

The delay statement in the 'init' method is there to make sure that the first broadcast messages sent by the Output-objects will be received by the Input-objects (otherwise a broadcast message might be sent by an Output-object while an Input-object is not yet listening).

The main routine for a three-state output is called '**threeState**', and for a normal output it is called '**continuous**' (this method is shown in Figure 5.6).

As you can see the only task a continuous output-object has is to receive new values from the block it is connected to (on the channel 'toCore') and broadcast this new value to all connected inputs. The newly received value is only sent to the connected inputs if it is different than the current output value (this is done to reduce the amount of channel traffic).

```
continuous()()
| dest : String |
    toCore ? value(dest, newValue | dest=name);
    if (newValue != contents) then
        output !* value(newValue);
        contents := newValue
    fi;
    continuous()().
```

Figure 5.6: Main routine of a continuous output

The program code of the ‘threeState’ method is shown in Figure 5.7. As you can see it contains a lot more functionality than the method ‘continuous’.

```
threeState()()
| dest, stateCmd : String |
    if disabled then
        sel
            toCore ? value(dest, newValue | dest=name)
        or
            cmd ? outputState(dest, stateCmd | dest=fullOutputName);
            if stateCmd = "enable" then
                disabled := false;
                contents := newValue;
                output !* value(contents)
            fi
        or
            clock ? afterClock;
            disabled := defaultDisabled;
            if disabled not() then
                contents := newValue;
                output !* value(contents)
            fi
        les
    else
        sel
            toCore ? value(dest, newValue | dest=name);
            if contents != newValue then
                contents := newValue;
                output !* value(contents)
            fi
        or
            cmd ? outputState(dest, stateCmd | dest=fullOutputName);
            if stateCmd = "disable" then
                disabled := true
            fi
        or
            clock ? afterClock;
            disabled := defaultDisabled
        les
    fi;
    threeState()().
```

Figure 5.7: Main routine of a three-state output

An **if..then** statement divides the ‘threeState’ method description into two different parts: *disabled* and *enabled*. The first part specifies the behaviour of a disabled output, and the other part specifies the behaviour of an enabled output. Both parts consist of a **select** statement with three choices:

1. receive new value;
2. receive state command ('enable' or 'disable');
3. receive clock message.

The last choice is only needed to set the three-state output to its default state when a clock message is received.

5.2.4 Register

An IDaSS register model can have one input connector and/or one (three-state) output connector. Aside from normal register operations like 'load from input' or 'hold value', it can also perform increment and decrement operations without needing external hardware. The normal value test on a register will return the current contents of the register to a state controller.

Because all the registers in the RTL 8048 design are very similar (they all have one input connector and one output connector), a cluster class named '*Register*' was created. The structure of this cluster class is shown in Figure 5.8. As one can see a Register-cluster contains three process objects: *Input*, *Register* and *Output*. The specifications of the Input- and Output-objects were examined in the previous paragraphs. In this paragraph the process class '*Register*' is described.

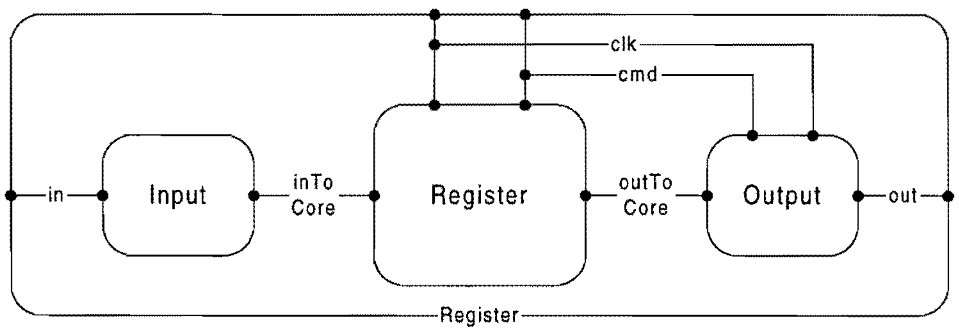


Figure 5.8: Instance Structure Diagram of a Register cluster

The cluster class 'Register' contains nine instantiation parameters that are used to initialise the three process objects:

- registerName: name of the register;
- inputName: name of the register's input;
- outputName: name of the register's output;
- width: number of bits;
- defaultCommand: default command (*hold*, *load*, *loadinc*, *loaddec*, *inc*, *dec*);
- cmdResetVal: value the register has after *reset* command;
- systemResetVal: system reset value;
- outputTS: specifies if output is continuous (*false*) or three-state (*true*);
- outputDefaultDisabled: specifies the default state of three-state output.

Figure 5.9 shows the structure of the specification of the process class *Register*. The endless loop is the specification of the main method, which contains three different phases. In the first phase the new control command for the register is determined. Commands can be received on the 'cmd' channel. These can be normal register control commands such as 'load' and 'inc', but also commands to control the output-state (enableAll, disableAll) and test commands. The control commands are received as a String and they are converted to Integers in the method '**decodeCmd**' for convenience and speed improvement in the following phases. The first phase is aborted when the message 'beforeClock' is received on the 'clk' channel.

In the second phase the new control command will be executed. If no control command was received in the first phase, then the default command will be executed. The new command will not be executed before the message 'clockPulse' is received, to make sure that the register input is stable. Command execution takes place in the process method '**cmdExecute**'.

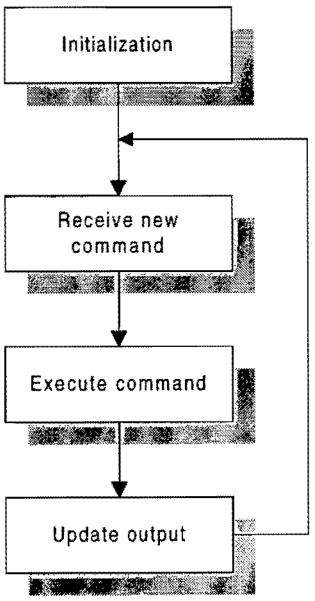


Figure 5.9: Structure of the Register specification

In the last phase of the main routine, the new value that was determined in the previous phase is being sent to the output of the register. The following example shows why the tasks performed in this phase and in the previous phase may not be executed in the same phase. In Figure 5.10 a simple design is shown, which contains two registers.

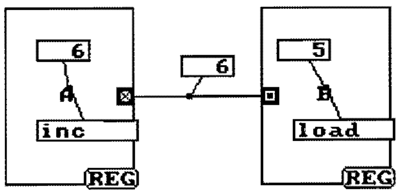


Figure 5.10: Simple IDaSS design

The register on the left is about to execute the 'inc' command (the new value of this register will be '7') and the register on the right will load the value that is currently present on its input ('6'). If the *execution*-task and the *output update*-task would take place in the same phase, then it is possible that the new value of the left register is sent to the input of the right register before this register has loaded the input value. In that case the new value of the left register will be '6' instead of '7'.

5.2.5 Operator

Operators model all asynchronous elements in a design. An operator is capable of performing any conceivable combinatorial operation. It can have multiple inputs and multiple (three-state) outputs. The inputs or outputs can have an independent number of bits as width. An operator can also have multiple 'functions', from which it can always execute only one at a time. The function that is actually being executed is determined by sending this function's name as a command to the operator.

All operators in the RTL 8048 design are different and therefore each one of them has its own process class. Although the functionalities of these classes are different, the main structure is the same. This is why only one of the operators will be discussed in this paragraph.

The RTL 8048 design contains four operator blocks: *PCOP*, *PSWOP*, *ALU* and *ADDRGEN*. This last operator was taken as an example to explain the operator process class because the *ADDRGEN*-operator only contains four functions. The operator has the following connectors:

- An input connector with name 'addr'; an 8 bits wide input from the indirect address temporary register.
- An input connector with name 'ir'; an 8 bits wide input from the instruction register (to select registers within a bank).
- An input connector with name 'psw'; an 8 bits wide input from the program status word (bank select bit).
- An output connector with name 'out'; a 6 bits wide continuous output for the generated address.

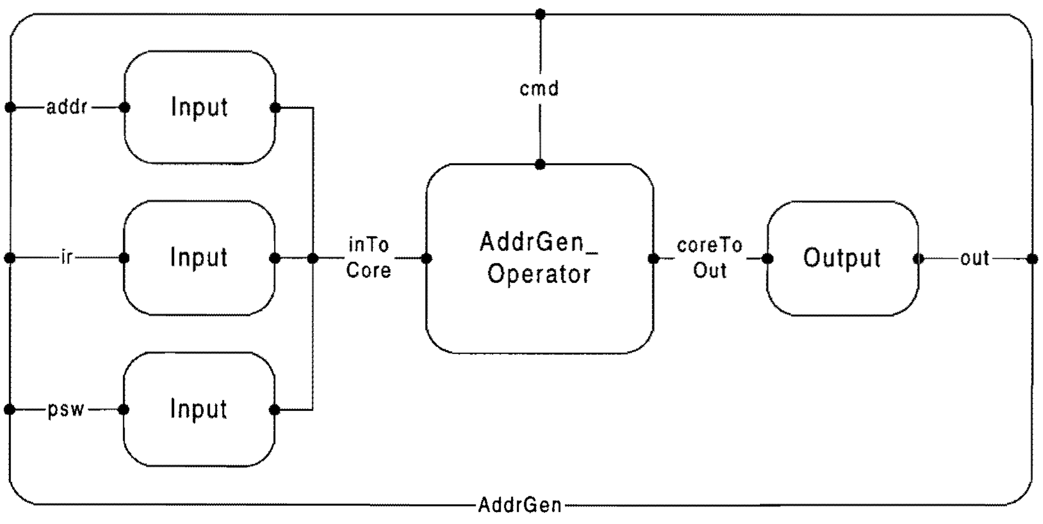


Figure 5.11: Instance Structure Diagram of AddrGen cluster

A cluster class represents the ADDRGEN-operator. The structure of the '*AddrGen*' cluster class is shown in Figure 5.11. The process class '*AddrGen_Operator*' holds the main specification of the operator. This specification is actually very simple, because the clock does not have to be taken into consideration in this block.

The only thing an operator block has to do is wait for a new input value or for a new control command and then execute the current function that calculates the new output value. This is done in the method '**main**', which is shown in Figure 5.12. The first three choices of the select statement allow the reception of new input values. The last choice of the select statement allows the reception of new commands to control which operator function will be executed.

```
main()()
| newVal : Integer; newFunc, source, dest : String |
    sel
        input ? value(source, newVal | (source="addr") & (newVal != addr getVal()));
        addr setVal(newVal)
    or
        input ? value(source, newVal | (source="ir") & (newVal != ir getVal()));
        ir setVal(newVal)
    or
        input ? value(source, newVal | (source="psw") & (newVal != psw getVal()));
        psw setVal(newVal)
    or
        cmd ? command(dest, newFunc | dest=name);
        if newFunc = function then
            main()()
        fi;
        function := newFunc
    les;
    updateOutput()();
    main()().
```

Figure 5.12: Main routine of the AddrGen operator

The method '**updateOutput**' executes the current function and determines the new output value. If the new output value is different from the old one, then the new value is sent to the connected Output-object. The execution of the four operator functions is performed by the method '**funcExecute**', which is shown in Figure 5.13. Four if..then statements check which operator function should be executed. The data statements that represent the operator functions look very much the same as the original descriptions in IDaSS. The IDaSS function-definitions for the *ADDRGEN*-operator are shown in Table 5.1. The local variable '*bank*' is used as a temporary variable for the '**reg**' function, just like in the IDaSS function definition.

Function '**addr**':	Function '**reg**':	Function '**stack**':	Function '**stinc**':
out := addr from: 0 to: 5	_bank := psw at: 4. out := 1 zeroes, _bank, _bank, (ir from: 0 to: 2)	out := 1 zeroes, (psw at: 2), (psw at: 2) not, (psw from: 0 to: 1), 1 zeroes	out := 1 zeroes, (psw at: 2), (psw at: 2) not, (psw from: 0 to: 1), 1 ones

Table 5.1: Functions of the ADDRGEN-operator

```
funcExecute()()
| bank : BoundedInteger |
    if function = "addr" then
        out := addr fromTo(0,5)
    fi;
    if function = "reg" then
        bank := psw at(4);
        out := new(BoundedInteger) zeroes(1) concat(bank) concat(bank)
            concat(ir fromTo(0,2))
    fi;
    if function = "stack" then
        out := new(BoundedInteger) zeroes(1) concat(psw at(2)) concat(psw at(2) not())
            concat(psw fromTo(0,1)) concat(new(BoundedInteger) zeroes(1))
    fi;
    if function = "stinc" then
        out := new(BoundedInteger) zeroes(1) concat(psw at(2)) concat(psw at(2) not())
            concat(psw fromTo(0,1)) concat(new(BoundedInteger) ones(1))
    fi.
```

Figure 5.13: Method ‘funcExecute’ of process class ‘AddrGen_Operator’

The complete specification of all the operator process classes is given in Appendix D (along with the rest of the RTL 8048 POOSL specification).

5.2.6 ROM

The RTL 8048 design contains one Read-Only Memory (ROM) with a single read port. This read port is made up of an address input connector that determines which word of the memory will be read and output on the three-state output connector. The reading is done asynchronously, so the output will follow the address input changes and no commands need be given to read. The only commands given to the ROM are commands to enable and disable its three-state output. The architecture of the cluster class ‘Rom’ that is shown in Figure 5.14 is very similar to that of the ‘Register’ class, but of course its functionality is totally different.

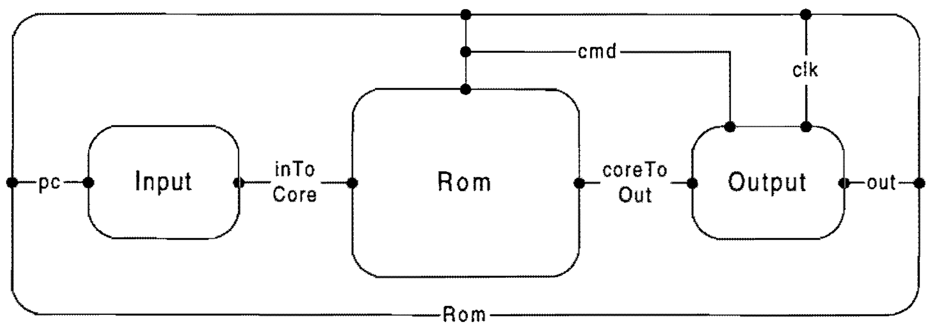


Figure 5.14: Instance Structure Diagram of Rom cluster

The ‘Rom’ cluster class uses the following instantiation parameters to initialise the process objects:

- width: number of bits per word;
- depth: number of words;
- addressInputName: name of the address input connector;
- dataOutputName: name of the data output connector;
- name: name of the element (simply ‘ROM’ in this case).

The definition of the initial method '**init**' is shown in Figure 5.15. The contents of the ROM are stored in an array that is initialised in the method '**initRom**'. This process method works the same way as the method 'initRom' of the AL 8048 that was described in paragraph 4.2.4. The contents of the ROM are stored in the instance variable '*rom*' of type Array. The other instance variables are '*inVal*', which holds the current value of the address input and '*outVal*', which does the same for the data output of the read port.

```
init()()
| fullOutputName, dest, stateCmd : String |
    initRom()();
    inVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    main()() interrupt(
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState(fullOutputName, "enable")
        else
            cmd ! outputState(fullOutputName, "disable")
        fi ).
```

Figure 5.15: Method 'init' of process class 'Rom'

The '**main**' method can be interrupted by the reception of a state command (enableAll, disableAll) for the three-state data output. The specification of the 'main' method is shown in Figure 5.16. The reading of the read port is done asynchronously, which means that as soon as the value of the address input changes the value on the data output is updated.

```
main()()
| oldOutVal, newVal : Integer; source : String |
    input ? value(source, newVal | (source=addressInputName)
        & (newVal != inVal));
    inVal := newVal;
    oldOutVal := outVal;
    if inVal < 0 then
        outVal := -3
    else
        outVal := rom get(inVal + 1)
    fi;
    if outVal != oldOutVal then
        output ! value(dataOutputName, outVal)
    fi;
    main()().
```

Figure 5.16: Method 'main' of process class 'Rom'

5.2.7 RAM

The IDaSS RAM models a random access read/write memory. The RAM in the RTL 8048 contains one read-only port and one write-only port. The read port contains an address-input connector (**ra**) and a data-output connector (**out**). The write port also has an address-input connector (**wa**) and it has a data-input connector (**in**). Reading is done asynchronously; the data-output follows the address-input directly. Writing is done synchronous with the clock.

The commands given to the ROM can be commands to enable and disable the three-state output of the read port, or commands that control the state of the write port. A 'writeAll' command will enable the writing from the data input of the write port, and a 'nowriteAll' command will disable writing.

The architecture of the Ram cluster (Figure 5.17) is a bit more complicated than the Rom cluster (Figure 5.14) due to the presence of the write-only port. The following instantiation parameters are used by the Ram cluster class:

- width: number of bits per word;
- depth: number of words;
- waInputName: name of the address-input connector of the write port;
- dataInputName: name of the data-input connector of the write port;
- raInputName: name of the address-input connector of the read port;
- dataOutputName: name of the data-output connector of the read port;
- defaultContents: the default contents of the words in the RAM (set this parameter to '-3' for an unknown value, see footnote 1 on page 11 for more information);
- name: name of the element (simply 'RAM' in this case).

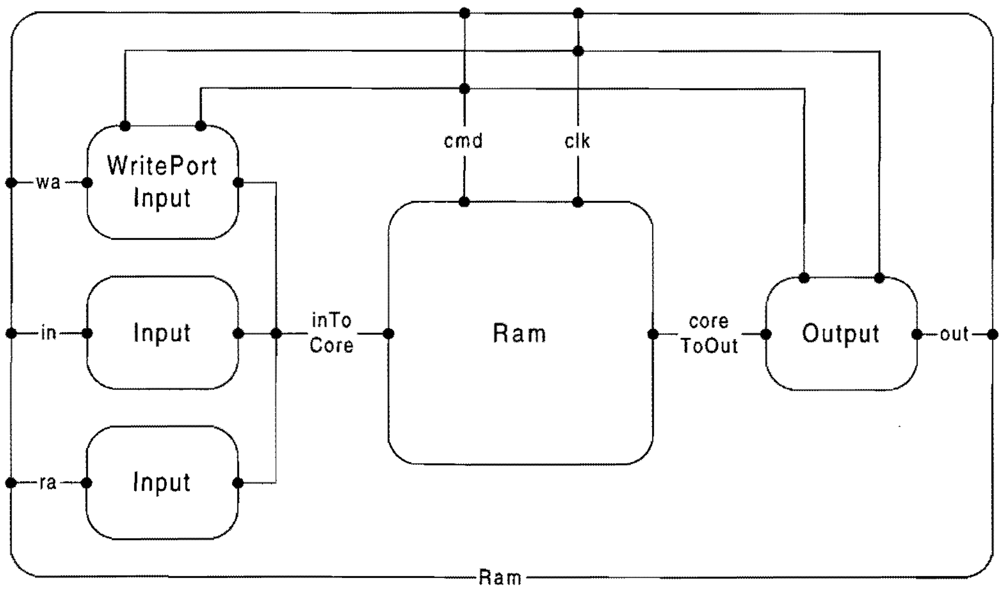


Figure 5.17: Instance Structure Diagram of Ram cluster

The specification of the process class 'Ram' is very much the same as that of the 'Rom' class. The functionality needed for the write port is the only thing that is added.

The contents of the RAM are stored in the instance variable *ram*, which is initialised in the method '**initRam**'. Normally this method will contain only the following statement: *ram* := new(Array) **size**(*depth*) **putAll**(*defaultContents*). This statement creates the array and sets all the elements of this array to the value given by the instantiation parameter *defaultContents*. Of course it is also possible to set certain elements to a value different than *defaultContents* (this can be done with the Array data method '**put**'), but normally this is only helpful for test purposes.

The only difference between the 'init' method of Figure 5.15 and the one shown in Figure 5.19 is the presence of the select statement. The first choice in this statement allows the reception of commands (enableAll, disableAll) that control the three-state data output of the read port. The second choice allows the reception of commands (writeAll, nowriteAll) that control the state of the write port (whether or not writing is allowed).

```
init()()
| fullOutputName, dest, stateCmd : String |
    initRam()();
    raVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    main()() interrupt(
        sel
            cmd ? outputState(dest, stateCmd | dest=name);
            if stateCmd = "enableAll" then
                cmd ! outputState(fullOutputName, "enable")
            else
                cmd ! outputState(fullOutputName, "disable")
            fi
        or
            cmd ? writePortState(dest, stateCmd | dest=name);
            if stateCmd = "writeAll" then
                cmd ! writePortState(waInputName, "write")
            else
                cmd ! writePortState(waInputName, "noWrite")
            fi
        les).
```

Figure 5.19: Method 'init' of process class 'Ram'

The '**main**' method of the Ram process class is shown in Figure 5.20. Here the differences with the same method of the Rom class are also very small. The first choice of the select statement takes care of the functionality for the read-only port of the RAM. The second choice does the same thing for the write port. As soon as the message *clockPulse* is received, the current write address is received on the 'input' channel. A negative value of this write address means that writing is disabled and in that case nothing will be done. When a positive address value was received, then the current value on the data input of the write port is read and placed in the *ram* array at the specified address. This new value will also be sent to the data output of the read port if the input of that port points to the same address as the address input of the write port.

```
main()()
| waVal, inVal, newVal, oldOutVal : Integer; source : String |
    sel
        input ? value(source, newVal | (source=raInputName) & (newVal != raVal));
        raVal := newVal;
        oldOutVal := outVal;
        if raVal < 0 then
            outVal := -3
        else
            outVal := ram get(raVal + 1)
        fi;
        if outVal != oldOutVal then
            output ! value(dataOutputName, outVal)
        fi
    or
        clock ? clockPulse;
        input ? value(source, waVal | source=waInputName);
        if waVal >= 0 then
            input ? value(source, inVal | source=dataInputName);
            ram put(waVal + 1, inVal);
            if ((raVal = waVal) & (outVal != inVal)) then
                outVal := inVal;
                output ! value(dataOutputName, outVal)
            fi
        fi
    les;
    main()().
```

Figure 5.20: Method 'main' of process class 'Ram'

5.2.8 State controller

The RTL 8048 design contains one state controller that controls all the other elements in the design. The state controller is specified in the process class ‘Control’. Each process method of this class represents a state of the state controller. An extra process method was added, which is only used as the initial method.

The conversion of the states of the controller to POOSL process methods can be best explained by looking at an example. Figure 5.21 shows what the IDaSS specification of the state ‘**exec3**’ looks like in the process method ‘**exec3**’. Each clock cycle the controller executes one of its states. To respect this clock cycle each process method that represents a state must wait for a message from the clock (this is the first statement in the ‘**exec3**’ method).

To control the state of a three-state output on a block, an enable/disable command can be sent to this block. For instance, if we want to enable the three-state output that is part of the read port of the ROM, we will have to do this by executing the following send statement: *cmd ! outputState(“ROM”, ”enableAll”).* As you can see the destination of the command is specified by the first parameter of this message-send statement and the command itself is specified in the second parameter.

To control the functions of a register or an operator, the message *command* has to be sent to the object that we want to control. If we want to change the function of the operator PCOP to ‘*loadlo*’, we can do this as follows: *cmd ! command(“PCOP”,* *"loadlo")*. If we want to change the contents of the register PC to '25', we can do this as follows: *cmd ! command("PC", "setto:", 25)*. Hence, controlling a register requires an extra parameter in this case ('**setto:**' is a keyword command). For other register commands (such as '**inc**' and '**load**') we can fill in any value for this parameter, because it is not used.

```
exec3:
[    IR
|    %xxx10100
     ROM enable;
     PCOP loadlo;
     PC load;
     -> exec4
|    %10010011
     PSWOP rest;
     PSW load;
|    %100x0011
     ADDRGEN stinc;
     RAM enable;
     PCOP loadhi;
     PC load;
     -> fetch
|    %11101xxx
     ROM enable;
     PCOP loadlo;
     PC load;
     -> fetch
|    %0010000x
     ADDRGEN addr;
     RAM enable;
     ACCU load;
     -> exec4
|    %00101xxx
     ADDRGEN reg;
     RAM enable;
     ACCU load;
     -> exec4
]
```

⇒

```
exec3()()
    clock ? afterClock;
    if (irVal & %11111) = %10100 then
        cmd ! outputState("ROM","enableAll");
        cmd ! command("PCOP","loadlo");
        cmd ! command("PC","load",-3);
        exec4()()
    fi;
    if irVal = %10010011 then
        cmd ! command("PSWOP","rest");
        cmd ! command("PSW","load",-3)
    fi;
    if (irVal & %11101111) = %10000011 then
        cmd ! command("ADDRGEN","stinc");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("PCOP","loadhi");
        cmd ! command("PC","load",-3);
        fetch()()
    fi;
    if (irVal & %11111000) = %11101000 then
        cmd ! outputState("ROM","enableAll");
        cmd ! command("PCOP","loadlo");
        cmd ! command("PC","load",-3);
        fetch()()
    fi;
    if (irVal & %11111110) = %00100000 then
        cmd ! command("ADDRGEN","addr")
    else
        cmd ! command("ADDRGEN","reg")
    fi;
    cmd ! outputState("RAM","enableAll");
    cmd ! command("ACCU","load",-3);
    exec4()().
```

Figure 5.21: Conversion from IDaSS state to POOSL method

Testing the value of another element in the IDaSS design is done by sending the message *test* on the channel *cmd* to the object we wish to test. For instance, if we would like to know the value of the register IR, the following send statement must be executed: *cmd ! test("IR", "normal")* followed by the conditional receive statement *cmd ? value(source, ir* | *source="IR")* which allows the reception of the current contents of the IR register.

5.2.9 The complete design

Now that all the IDaSS blocks that are used in the RTL 8048 design have been specified in POOSL, we can take a look at the complete POOSL specification of the RTL 8048. Figure 5.22 shows a screen snapshot of the complete POOSL specification (designed and simulated with the POOSL Simulator). The large amount of channels make the design look quite messy, which is the reason why scenarios were introduced into the design. Scenarios make it possible to hide certain (irrelevant) elements of the design in order to make the design easier to understand [vdPV97].

The following four scenarios were created:

1. *BasicBlocks*; this scenario contains all the basic elements of the RTL 8048 design (operators, registers, RAM, ROM and state controller).
2. *Clock*; this scenario contains the process object *Clock* and the channel *clk* that goes with this object.
3. *Command*; this scenario contains the channel *cmd*, which is used to control all the basic blocks of in design.
4. *DataBuses*; this scenario contains all the channels that represent an IDaSS bus.

In Figure 5.23 the RTL 8048 design is shown with only the *BasicBlocks* and *DataBuses* scenarios visualised. Notice that the design looks very much like the original IDaSS schematic that is shown in Figure 5.1. Fortunately the POOSL design not only looks the same as its IDaSS opponent, but it also seems to behave the same (as far as this could be tested). Certain aspects of the design and of the POOSL simulator make it very difficult to properly test and debug the complete specification of the RTL 8048. First of all the design has a lot of process objects with very heavy traffic running over the channels that connect these objects. The POOSL simulator is not fast enough (yet) to handle a design of this size. In addition to this the POOSL simulator currently lacks proper debug facilities.

The complete POOSL specification of the RTL 8048 can be found in Appendix D. This appendix also contains the Instance Structure Diagram of the complete system (which looks basically the same as the screen snapshot in Figure 5.22).

The RTL 8048 design contains only the basic elements that are available in IDaSS. The POOSL specifications of these basic elements have been discussed in the previous paragraphs, but other IDaSS blocks have been specified in POOSL as well (blocks such as FIFO, LIFO, CAM, constant generator and buffer). The POOSL specifications of these blocks are given in Appendix E (mostly without any further explanation).

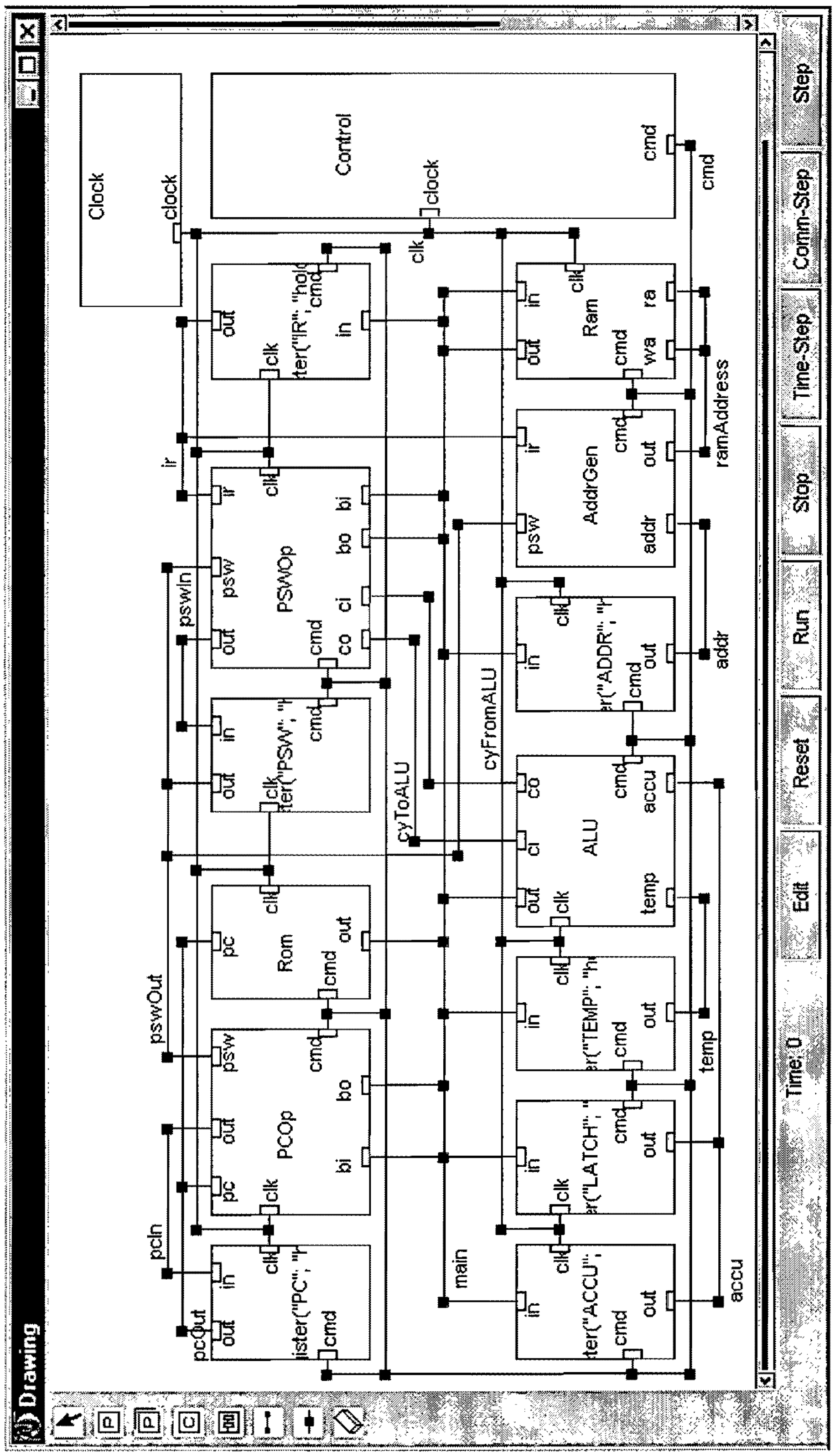


Figure 5.22: Simulator screen snapshot of RTL 8048

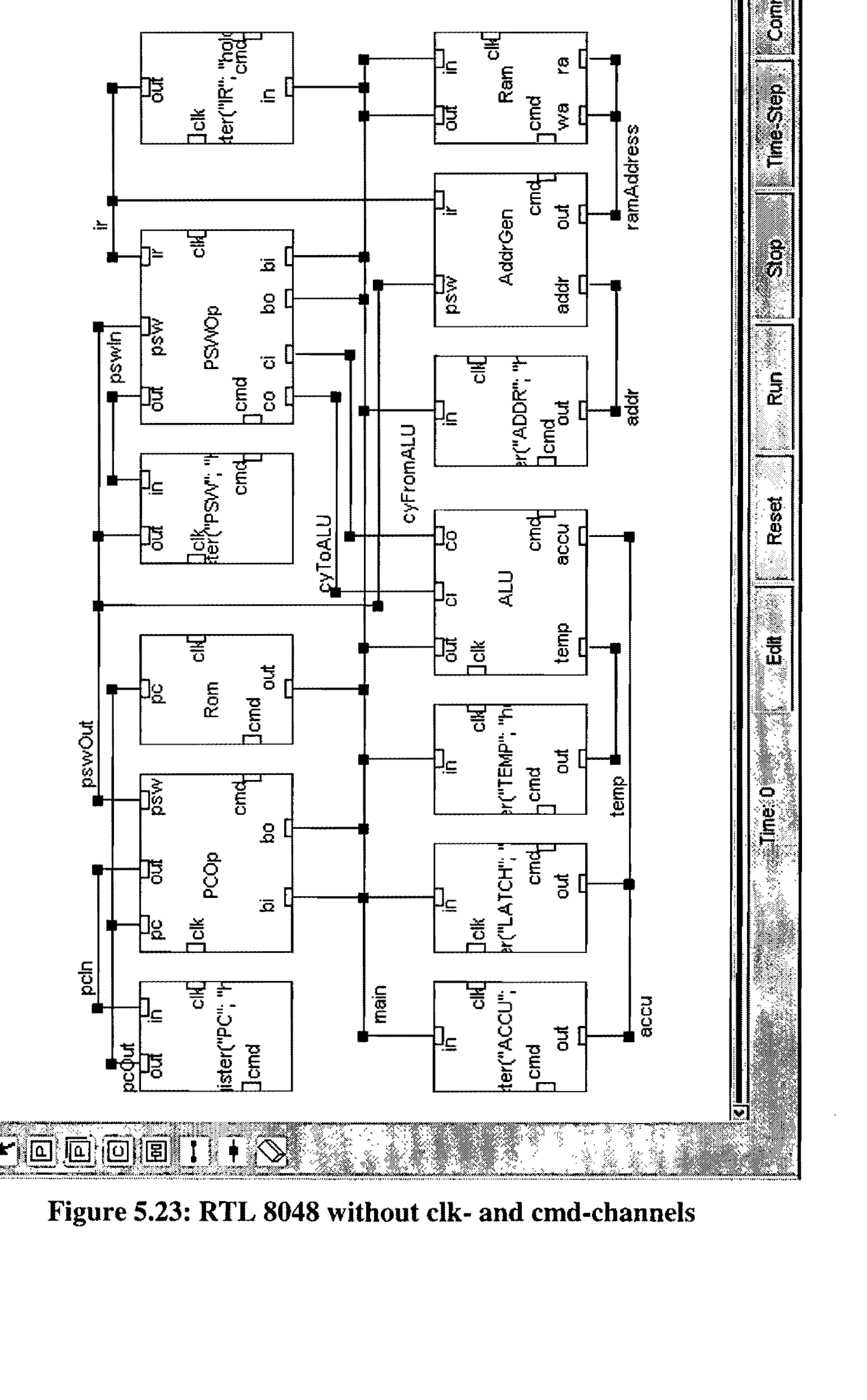


Figure 5.23: RTL 8048 without clk- and cmd-channels

6. Conclusions and recommendations

The previous chapters show that it is fairly easy to model IDaSS elements in POOSL, and to specify an IDaSS design in POOSL. The POOSL specification is obtained by mapping the IDaSS blocks onto POOSL process objects and clusters and IDaSS databuses into POOSL channels. Although small IDaSS designs already require a rather large POOSL specification, the language POOSL is very well suited for the description of digital systems.

Now that we are able to describe the IDaSS designs in POOSL, we have an entire collection of hardware system designs to our disposal. This is very practical for the co-simulation of hardware/software systems.

By modelling the most common IDaSS elements in POOSL, it is also demonstrated that it is possible to model hardware descriptions with synchronous concurrency and asynchronous communication (IDaSS) in a specification language that is based on asynchronous concurrency and synchronous communication (POOSL).

In future research concerning this project it would be wise to take a good look again at the two different clock models (2-phase model and 3-phase model). Due to a lack of time these models have not been thoroughly tested.

Only one IDaSS RTL design has been modelled in POOSL. Other designs may introduce different problems; therefore other RTL designs should be modelled in POOSL. These designs might even contain some of the 'untested' IDaSS elements from Appendix E.

IDaSS contains an export function called *Alien File Generation*. The 'alien file' generation functionality allows IDaSS to write out a design (or part of such a design) to one or more text files in a format which is not the normal IDaSS design (.DES) file format. The language to be used is not defined within IDaSS itself, but rather in a template file which contains a kind of program to convert each of the IDaSS constructs and operators into the chosen target language. This target language can be POOSL, if a proper template file is written that can produce POOSL specifications. The existence of such a template file will imply that almost all IDaSS designs can be quickly available in POOSL.

At this moment it is not very clear which POOSL statements and constructions produce a 'slow' model. Maybe it is possible to make a tool that measures the speed of these different constructions. A designer may be able to avoid certain statements in his POOSL specification that have proved to be very slow. When the 'slow' statements are known it is also possible to optimise these statements for speed in the POOSL Simulator. Speed improvements might also be obtained by making the data methods of the BoundedInteger data class primitive (implemented in Smalltalk rather than in POOSL).

The POOSL Simulator not only works as a simulator, but also as a model editor. As a simulator the POOSL Simulator suffices very well. Debug facilities must be added to make it satisfy as a model editor as well. At this moment it is impossible to properly debug data methods. Maybe it is possible to add *breakpoint* facilities to the POOSL Simulator. A single debug-window that shows different instance and local variables [*watchpoints*] can also be very practical.

The last recommendation concerns the documentation of IDaSS and POOSL. There is a user manual available for IDaSS, but this manual is not very much up-to-date, and especially for novice designers the online help system in IDaSS is not very convenient. At this moment no up-to-date documentation is available concerning the POOSL language and the POOSL Simulator. Writing a document that contains all the POOSL statements and their explanation will not take much time, and it is very useful for a novice POOSL modeller.

Appendix A: Intel MCS-48 Instruction Set

Mnemonic	Op	Description	Bytes	Cycles
Accumulator				
ADD A,R	68	Add register	1	1
ADD A,@R	60	Add data memory to A	1	1
ADD A,#data	03	Add immediate to A	2	2
ADDC A,R	78	Add register with carry	1	1
ADDC A,@R	70	Add data memory with carry	1	1
ADDC A,#data	13	Add immediate with carry	2	2
ANL A,R	58	And register to A	1	1
ANL A,@R	50	And data memory to A	1	1
ANL A,#data	53	And immediate to A	2	2
ORL A,R	48	Or register to A	1	1
ORL A,@R	40	Or data memory to A	1	1
ORL A,#data	43	Or immediate to A	2	2
XRL A,R	D8	Exclusive Or register to A	1	1
XRL A,@R	D0	Exclusive Or data memory to A	1	1
XRL A,#data	D3	Exclusive Or immediate to A	2	2
INC A	17	Increment A	1	1
DEC A	07	Decrement A	1	1
CLR A	27	Clear A	1	1
CPL A	37	Complement A	1	1
DA A	57	Decimal adjust A	1	1
SWAP A	47	Swap nibbles of A	1	1
RL A	E7	Rotate A left	1	1
RLC A	F7	Rotate A left through carry	1	1
RR A	77	Rotate A right	1	1
RRC A	67	Rotate A right through carry	1	1
Input/Output				
IN A,P	08	Input port to A	1	2
OUTL P,A	38	Output A to port	1	2
ANL P,#data	98	And immediate to port	2	2
ORL P,#data	88	Or immediate to port	2	2
INS A,BUS	08	Input BUS to A	1	2
OUTL BUS,A	02	Output A to BUS	1	2
ANL BUS,#data	98	And immediate to BUS	2	2
ORL BUS,#data	88	Or immediate to BUS	2	2
MOVD A,P	0C	Input Expander port to A	1	2
MOVD P,A	3C	Output A to Expander port	1	2
ANLD P,A	9C	And A to Expander port	1	2
ORLD P,A	8C	Or A to Expander port	1	2
Registers				
INC R	18	Increment register	1	1
INC @R	10	Increment data memory	1	1
DEC R	C8	Decrement register	1	1

Mnemonic	Op	Description	Bytes	Cycles
Branch				
JMP addr	04	Jump unconditional	2	2
JMPP @A	B3	Jump indirect	1	2
DJNZ R,addr	E8	Decrement register and jump	2	2
JC addr	F6	Jump on carry = 1	2	2
JNC addr	E6	Jump on carry = 0	2	2
JZ addr	C6	Jump on A zero	2	2
JNZ addr	96	Jump on A not zero	2	2
JT0 addr	36	Jump on T0 = 1	2	2
JNT0 addr	26	Jump on T0 = 0	2	2
JT1 addr	56	Jump on T1 = 1	2	2
JNT1 addr	46	Jump on T1 = 0	2	2
JF0 addr	B6	Jump on F0 = 1	2	2
JF1 addr	76	Jump on F1 = 1	2	2
JTF addr	16	Jump on timer flag = 1	2	2
JNI addr	86	Jump on INT = 0	2	2
JBb addr	12	Jump on Accumulator Bit	2	2
Subroutine				
CALL addr	14	Jump to subroutine	2	2
RET	83	Return	1	2
RETR	93	Return and restore status	1	2
Flags				
CLR C	97	Clear Carry	1	1
CPL C	A7	Complement Carry	1	1
CLR F0	85	Clear Flag 0	1	1
CPL F0	95	Complement Flag 0	1	1
CLR F1	A5	Clear Flag 1	1	1
CPL F1	B5	Complement Flag 1	1	1
Data Moves				
MOV A,R	F8	Move register to A	1	1
MOV A,@R	F0	Move data memory to A	1	1
MOV A,#data	23	Move immediate to A	2	2
MOV R,A	A8	Move A to register	1	1
MOV @R,A	A0	Move A to data memory	1	1
MOV R,#data	B8	Move immediate to register	2	2
MOV @R,#data	B0	Move immediate to data memory	2	2
MOV A,PSW	C7	Move PSW to A	1	1
MOV PSW,A	D7	Move A to PSW	1	1
XCH A,R	28	Exchange A and register	1	1
XCH A,@R	20	Exchange A and data memory	1	1
XCHD A,@R	30	Exchange nibble of A and register	1	1
MOVX A,@R	80	Move external data memory to A	1	2
MOVX @R,A	90	Move A to external data memory	1	2
MOVP A,@A	A3	Move to A from current page	1	2
MOVP3 A,@A	E3	Move to A from Page 3	1	2

Mnemonic	Op	Description	Bytes	Cycles
Timer/Counter				
MOV A,T	42	Read Timer/Counter	1	1
MOV T,A	62	Load Timer/Counter	1	1
STRT T	55	Start Timer	1	1
STRT CNT	45	Start Counter	1	1
STOP TCNT	65	Stop Timer/Counter	1	1
EN TCNTI	25	Enable Timer/Counter Interrupt	1	1
DIS TCNTI	35	Disable Timer/Counter Interrupt	1	1
Control				
EN I	05	Enable external Interrupt	1	1
DIS I	15	Disable external Interrupt	1	1
SEL RB0	C5	Select register bank 0	1	1
SEL RB1	D5	Select register bank 1	1	1
SEL MB0	E5	Select memory bank 0	1	1
SEL MB1	F5	Select memory bank 1	1	1
ENT0 CLK	75	Enable clock output on T0	1	1
NOP	00	No Operation	1	1

Appendix B: Bounded Integer data class

IDaSS operator	POOSL data method	Description
dec	dec	decrement value
epty	epty	even parity bit
inc	inc	increment value
isovl	not available	check if bus overloaded
ists	not available	check if bus is three-state
isunk	not available	check if value is unknown
log2	log2	number of bits to represent receiver
lsomask	lsomask	least significant one bit mask
lsone	lsone	least significant one bit position
lszmask	lszmask	least significant zero bit mask
lszero	lszero	least significant zero bit position
maj	maj	majority gate
msomask	msomask	most significant one bit mask
msone	msone	most significant one bit position
mszmask	mszmask	most significant zero bit mask
mszero	mszero	most significant zero bit position
neg	neg	two's complement negative
not	not	complement bits
onecnt	onecnt	count number of ones in word
ones	ones	generate all ones
opty	opty	odd parity bit
rev	rev	reverse all bits MSB ↔ LSB
width	not available	return number of bits in value
zerocnt	zerocnt	count number of zeroes in word
zeroes	zeroes	generate all zeroes
+	add	add
-	sub	subtract
*	umply	unsigned multiply
*+	rhsmply	right hand signed multiply
+*	lhsmply	left hand signed multiply
+*+	smply	signed multiply
Λ	logicAND	logical AND
~Λ	logicNAND	logical NAND
V	logicOR	logical OR
~V	logicNOR	logical NOR
><	logicXOR	logical XOR
<>	logicXNOR	logical XNOR
,	concat	concatenate words
=	equal	unsigned 'equal'
~=	notEqual	unsigned 'not equal'
<	unsignedLess	unsigned 'less than'
<=	unsignedLessEqual	unsigned 'less than or equal'
=<	unsignedLessEqual	unsigned 'less than or equal'

IDaSS operator	POOSL data method	Description
>	unsignedGreater	unsigned ‘greater than’
>=	unsignedGreaterEqual	unsigned ‘greater than or equal’
=>	unsignedGreaterEqual	unsigned ‘greater than or equal’
+=+	equal	signed ‘equal’
+~=+	notEqual	signed ‘not equal’
+<+	signedLess	signed ‘less than’
+<=+	signedLessEqual	signed ‘less than or equal’
+=<+	signedLessEqual	signed ‘less than or equal’
+>+	signedGreater	signed ‘greater than’
+>=+	signedGreaterEqual	signed ‘greater than or equal’
+=>+	signedGreaterEqual	signed ‘greater than or equal’
add:cin:	addCin	addition with carry in/out
at:	at	select a single bit
at:width:	atWidth	select a shifting bit field
copiesOf:	copiesOf	concatenate a word with itself
decode:	decode	1-out-of-N decoder
decode:enable:	decodeEnable	1-out-of-N decoder with enable
from:to:	fromTo	select a fixed bit field
if0:if1:	if0if1	multiplex two values
if1:if0:	if1if0	multiplex two values
merge:from:to:	mergeFromTo	shifting bit field merge
merge:mask:	mergeMask	masked merge of two words
rol:	rol	rotate left
ror:	ror	rotate right
sar:	sar	shift arithmetic right
shl:	shl	shift logical/arithmetic left
shr:	shr	shift logical right
signed:	signed	sign extend a word
sol:	sol	shift left, inserting ones
sor:	sor	shift right, inserting ones
width:	width	change width of a word
not available	getVal	Return value
not available	getWidth	Return width
not available	init	Set value and width
not available	isOne	Return ‘true’ if val = 1
not available	isZero	Return ‘true’ if val = 0
not available	printString	Used by POOSL Simulator
not available	setVal	Set value
not available	setWidth	Set width

```
data class BoundedInteger
/* superclass: (Object) */

instance variables
width: Integer; val: Integer

instance methods
add(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ====================== Add ======================  */
/*  Addition (two's complement/unsigned). If neither side is a constant,  */
/*  then their widths must be equal. If only one side is a constant, then  */
/*  it's value should be representative with the number of bits in the  */
/*  other side's word (adding constants returns a constant result).  */
/*  ==================================================  */
    if width = 0 then
        newWidth := aBI getWidth()
    else
        newWidth := width
    fi;
    returnBI := new(BoundedInteger) init(aBI getVal() + val, newWidth);
    if newWidth != 0 then
        returnBI := returnBI width(newWidth)
    fi;
    return(returnBI).

addCin(aBI,carry : BoundedInteger) : BoundedInteger
| returnBI : BoundedInteger |
/*  ============= Addition with carry in / out ================ */
/*  Return the sum of the receiver and parameter 1, with an extra bit  */
/*  indicating the carry out. The carry-in for this addition is given by  */
/*  parameter 2 (which must be either a constant with value 0 or 1 or  */
/*  a single bit variable). Unlike the 'add' method, it is NOT allowed here */
/*  to add two constant values. If one of the summed values is a  */
/*  constant, it should be in the range of representative values of the  */
/*  other value. If neither of them is a constant, they should have the  */
/*  same width.  */
/*  ==================================================*/
    if width=0 then
        returnBI:=new(BoundedInteger) init(val + aBI getVal() +
            carry getVal(), otherWidth+1)
    else
        returnBI:=new(BoundedInteger) init(val + aBI getVal() +
                carry getVal(),width+1)
    fi;
    return(returnBI).

at(pos : Integer) : BoundedInteger
| returnBI : BoundedInteger |
/*  =============== Select a single bit ================  */
/*  Return a single bit Bounded Integer containing the receivers bit  */
/*  at the position given by the parameter (integer). The parameter  */
/*  should lie in the range 0 ... (number of bits in receiver - 1).  */
/*  =============================================  */
    if (val & (2 power(pos)))=0 then
        returnBI:=new(BoundedInteger) init(0,1)
    else
        returnBI:=new(BoundedInteger) init(1,1)
    fi;
    return(returnBI).

atWidth(pos, returnWidth : Integer) : BoundedInteger
| i, anInt : Integer |
/*  =============== Select a shifting bit field ================*/
/*  Return an integer with a width given by parameter 2, with it's value  */
/*  extracted from the receiver starting at the bit position given by p1.  */
/*  The bit field specified this way should lie completely within the bit  */
/*  width of the receiver. In other words, the following bounds should  */
/*  be adhered to: 0 <= p1 < (receiver width - p2).  */
/*  ======================================================*/
    i := 1;
    anInt := val;
    while i <= pos do
        anInt := anInt div(2);
        i := i + 1
    od;
    anInt := anInt & (2 power(returnWidth) - 1);
    return(new(BoundedInteger) init(anInt, returnWidth)).

concat(aBI : BoundedInteger) : BoundedInteger
| otherVal, otherWidth : Integer |
/*  =============== Concatenation operator =============     */
/*  Concatenate the receiver word with the right hand side, returning     */
/*  an integer with a width which is the sum of the widths of the     */
/*  receiver and the right hand side. The result's most significant bits     */
/*  will come from the receiver, the least significant bits from the right     */
/*  hand side. Neither side can be a constant.     */
/*  ==============================================     */
    otherVal := aBI getVal();
    otherWidth := aBI getWidth();
    return(new(BoundedInteger) init(2 power(otherWidth) * val +
                otherVal, width + otherWidth)).

copiesOf(nrOfCopies : Integer; aBI : BoundedInteger) : BoundedInteger
| i, tempVal, tempWidth : Integer |
/*  ======== Concatenate copies of a word =========     */
/*  Return a Bounded Integer containing parameter 1 copies     */
/*  of the Bounded Integer given by parameter 2 (which     */
/*  cannot be a constant) concatenated side by side.     */
/*  ========================================     */
    tempVal := aBI getVal();
    tempWidth := aBI getWidth();
    i := 2;
    while i <= nrOfCopies do
        tempVal := tempVal * 2 power(tempWidth) + aBI getVal();
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal, nrOfCopies * tempWidth)).

dec() : BoundedInteger
/*  =============== Decrement value (subtract 1) =============     */
/*  Return a variable with the same width as the receiver, with as value the     */
/*  receiver's value minus one (with wrap around).     */
/*  The receiver cannot be a constant.     */
/*  ================================================     */
    if val = 0 then
        val := 2 power(width) - 1
    else
        val := val - 1
    fi;
    return(self).

decode(pos, outWidth : Integer) : BoundedInteger
/*  ========== 1-of-N decoder ==========     */
/*  Return a Bounded Integer with width given by     */
/*  parameter 2, with the bit number given by     */
/*  parameter 1set to %1. All other bits are %0.     */
/*  ================================     */
    return(new(BoundedInteger) init(2 power(pos), outWidth)).
```

```
decodeEnable(pos, outWidth : Integer; enableBit : BoundedInteger) : BoundedInteger
| returnBI : BoundedInteger |
/*  ========== 1-of-N decoder with enable ==========         */
/*  Return a value with width given by parameter2, with the   */
/*  bit number given by parameter 1 set to the value of       */
/*  parameter 3 (which must be a single bit Bounded Integer). */
/*  All other bits are %0.                                    */
/*  ============================================              */
    if enableBit getVal() = 0 then
        returnBI := new(BoundedInteger) init(0, outWidth)
    else
        returnBI := new(BoundedInteger) init(2 power(pos), outWidth)
    fi;
    return(returnBI).

epty() : BoundedInteger
| i, parity : Integer |
/*  ================== Even parity bit ==================           */
/*  Returns a single bit variable containing an even parity flag for the */
/*  receiver (value 1 if the number of ONEs is even).               */
/*  The receiver cannot be a constant.                              */
/*  ============================================                    */
    i := 0;
    parity := 1;
    while i < width do
        if (2 power(i) & val) != 0 then
            parity := 1 - parity
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(parity,1)).

equal(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ============ (Un)signed compare 'equal' ============      */
/*  Returns a single bit Bounded Integer, with value 1 meaning that */
/*  the test passed, value 0 meaning that the test failed. If neither */
/*  side is a constant, then their widths must be equal. If both */
/*  sides are constant, then this operator returns a constant with */
/*  values 0 or 1 for failing or passing the test.           */
/*  ============================================             */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
        newWidth := 1
    fi;
    if val = aBI getVal() then
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

fromTo(from,to : Integer) : BoundedInteger
| anInt, newWidth : Integer |
/*  =============== Bit extraction ===============             */
/*  Return a Bounded Integer extracted from receiver bits      */
/*  'from'.. 'to'. The width of the result will be ('to' - 'from' + 1). */
/*  Both parameters must be constants with as bounds the       */
/*  following: 0 <= 'from' <= 'to' < receiver width.             */
/*  =========================================                    */
    newWidth := to - from + 1;
    anInt := anInt div(2 power(from));
    anInt := anInt & (2 power(newWidth) - 1);
    return(new(BoundedInteger) init(anInt, newWidth)).

getVal() : Integer
/*  ===================================   */
/*  Returns the value of the receiver.    */
/*  Note: This method is not an operator in IDaSS.  */
/*  ===================================   */
    return(val).

getWidth() : Integer
/*  ===================================   */
/*  Returns the width of the receiver in bits.      */
/*  Note: This method is not an operator in IDaSS.  */
/*  ===================================   */
    return(width).

if0if1(zeroReturnBI, oneReturnBI : BoundedInteger) : BoundedInteger
| returnBI : BoundedInteger |
/*  ========== Multiplex two values ============= */
/*  Simulation of a two-input multiplexer. The receiver   */
/*  is used to select between the two parameters. The     */
/*  result will be parameter 1 if the receiver has value 0, */
/*  the result will be parameter 2 if the receiver has    */
/*  value 1.                                               */
/*  ======================================= */
    if val = 0 then
        returnBI := new(BoundedInteger) copiesOf(1, zeroReturnBI)
    else
        returnBI := new(BoundedInteger) copiesOf(1, oneReturnBI)
    fi;
    return(returnBI).

if1if0(oneReturnBI, zeroReturnBI : BoundedInteger) : BoundedInteger
| returnBI : BoundedInteger |
/*  ========== Multiplex two values ============= */
/*  Simulation of a two-input multiplexer. The receiver   */
/*  is used to select between the two parameters. The     */
/*  result will be parameter 1 if the receiver has value 1, */
/*  the result will be parameter 2 if the receiver has    */
/*  value 0.                                               */
/*  ======================================= */
    if val = 1 then
        returnBI := new(BoundedInteger) copiesOf(1, oneReturnBI)
    else
        returnBI := new(BoundedInteger) copiesOf(1, zeroReturnBI)
    fi;
    return(returnBI).

inc() : BoundedInteger
/*  ============ Increment value (add 1) ========== */
/*  Increment the receiver by adding one with wrap around, */
/*  the receiver cannot be a constant.                      */
/*  ========================================= */
    if val=(2 power(width) - 1) then
```

```
        val := 0
    else
        val := val + 1
    fi;
    return(self).

init(newVal, newWidth : Integer) : BoundedInteger
/*  ============ Initialize Bounded Integer =========== */
/*  Initialize both the value and the width of the BoundedInteger. */
/*  Note: This method is not an operator in IDaSS.                 */
/*  ============================================== */
    val := newVal;
    width := newWidth;
    return(self).

isOne() : Boolean
/*  ======================================      */
/*  Returns 'true' if the value of the receiver equals 1.   */
/*  Note: This method is not an operator in IDaSS.          */
/*  ======================================      */
    return(val = 1).

isZero() : Boolean
/*  ======================================      */
/*  Returns 'true' if the value of the receiver equals 0.   */
/*  Note: This method is not an operator in IDaSS.          */
/*  ======================================      */
    return(val = 0).

lhsmply(aBI : BoundedInteger) : BoundedInteger
| newVal, newWidth, tempVal : Integer |
/*  ================= Left hand signed multiply ===============  */
/*  The receiver cannot be a constant. The result's width is the total of the   */
/*  widths of receiver and right hand side value (where a constant has          */
/*  width zero). If this width is not enough to hold the result, then the excess */
/*  bits are chopped off (an overflow is ignored). The result should be         */
/*  interpreted as a signed value!                                              */
/*  ====================================================  */
    tempVal := val;
    newWidth := width + aBI getWidth();
    if width = 1 then
        if val >= 1 then
            tempVal := tempVal - 2
        fi
    else
        if val >= (2 power (width - 1)) then
            tempVal := val - (2 power(width))
        fi
    fi;
    newVal := (tempVal * (aBI getVal())) % (2 power(newWidth));
    return(new(BoundedInteger) init(newVal, newWidth)).

log2() : BoundedInteger
| i : Integer |
/*  ======= Number of bits to represent receiver =======  */
/*  Returns a constant indicating the number of bits needed  */
/*  to represent the receiver. The receiver must be a constant */
/*  itself.                                                   */
/*  ==========================================  */
```

```
    i := 1;
    while 2 power(i) <= val do
        i := i + 1
    od;
    return(new(BoundedInteger) init(i,0)).

logicAND(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer |
/*  ============== Logical AND function ==============       */
/*  If neither side is a constant, then their widths must be equal.    */
/*  If only one side is a constant, then it's value should be          */
/*  representative with the number of bits in the other side's word.  */
/*  ==============================================            */
    if width=0 then
        newWidth:=aBI getWidth()
    else
        newWidth:=width
    fi;
    return(new(BoundedInteger) init(aBI getVal() & val, newWidth)).

logicNAND(aBI : BoundedInteger) : BoundedInteger
/*  ============== Logical NAND function ==============      */
/*  If neither side is a constant, then their widths must be equal. If */
/*  only one side is a constant, then it's value should be             */
/*  representative with the number of bits in the other side's word.   */
/*  This function may not be used between constants.                   */
/*  ==============================================            */
    return(self logicAND(aBI) not()).

logicNOR(aBI : BoundedInteger) : BoundedInteger
/*  ============== Logical NOR function ==============       */
/*  If neither side is a constant, then their widths must be equal.    */
/*  If only one side is a constant, then it's value should be          */
/*  representative with the number of bits in the other side's word.   */
/*  This function may not be used between constants.                   */
/*  ==============================================            */
    return(self logicOR(aBI) not()).

logicOR(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer |
/*  ============== Logical OR function ==============        */
/*  If neither side is a constant, then their widths must be equal.    */
/*  If only one side is a constant, then it's value should be          */
/*  representative with the number of bits in the other side's word.   */
/*  ==============================================            */
    if width=0 then
        newWidth:=aBI getWidth()
    else
        newWidth:=width
    fi;
    return(new(BoundedInteger) init(val | aBI getVal() , newWidth)).

logicXNOR(aBI : BoundedInteger) : BoundedInteger
/*  ============= Logical XNOR function ============= */
/*  Cannot be used between constants. If neither side is a      */
/*  constant, then their widths must be equal. If only one side is */
/*  a constant, then it's value should be representative with the  */
/*  number of bits in the other side's word.                       */
/*  ========================================== */
    return(self logicAND(aBI) logicOR(self not() logicAND(aBI not()))).
```

```
logicXOR(aBI : BoundedInteger) : BoundedInteger
/*  ============== Logical XOR function ==============  */
/*  If neither side is a constant, then their widths must be equal. If  */
/*  only one side is a constant, then it's value should be  */
/*  representative with the number of bits in the other side's word.  */
/*  ==============================================  */
    return(self logicAND(aBI not()) logicOR(self not() logicAND(aBI))).

lsomask() : BoundedInteger
| i : Integer |
/*  ========= Least significant one bit mask =========  */
/*  Returns a variable (with the same width as the receiver)  */
/*  with a single ONE at the position of the least significant  */
/*  ONE bit in the receiver. Returns 0 if the receiver does not  */
/*  contain any ONEs. The receiver cannot be a constant.  */
/*  ========================================  */
    i := 0;
    while ((2 power(i) & val) = 0) & (i < width) do
        i := i +1
    od;
    return(new(BoundedInteger) init(2 power(i) & val, width)).

lsone() : BoundedInteger
| i : Integer |
/*  ======== Least significant one bit position =========  */
/*  Returns a variable (with the same width as the receiver)  */
/*  with as value the bit position of the least significant ONE in  */
/*  the receiver. The returned value equals the width of the  */
/*  receiver if the receiver does not contain any ONEs.  */
/*  The receiver cannot be a constant.  */
/*  ==========================================  */
    i := 0;
    while ((2 power(i) & val) = 0) & (i < width) do
        i := i +1
    od;
    return(new(BoundedInteger) init(i, width)).

lszero() : BoundedInteger
| i, tempVal : Integer |
/*  ======== Least significant zero bit position =======  */
/*  Returns a variable (with the same width as the receiver)  */
/*  with as value the bit position of the least significant ZERO  */
/*  in the receiver. The returned value equals the width of the  */
/*  receiver if the receiver does not contain any ZEROes.  */
/*  The receiver cannot be a constant.  */
/*  ========================================  */
    i := 0;
    tempVal := self not() getVal();
    while ((2 power(i) & tempVal) = 0) & (i < width) do
        i := i +1
    od;
    return(new(BoundedInteger) init(i, width)).

lszmask() : BoundedInteger
| i, tempVal : Integer |
/*  ======= Least significant zero bit mask ========  */
/*  Returns a variable (with the same width as the receiver)  */
/*  with a single ONE at the position of the least significant  */
/*   ZERO bit in the receiver. Returns 0 if the receiver does    */
/*   not contain any ZEROes.                                      */
/*   The receiver cannot be a constant.                           */
/*   ==========================================                   */
     i := 0;
     tempVal := self not() getVal();
     while ((2 power(i) & tempVal) = 0) & (i < width) do
          i := i +1
     od;
     return(new(BoundedInteger) init(2 power(i) & tempVal, width)).

maj() : BoundedInteger
| i, nrOfOnes : Integer; returnBI : BoundedInteger |
/*   =============== Majority gate =================     */
/*   Majority gate, if the receiver's width is odd, this operator     */
/*   returns a single bit variable having the same value as the       */
/*   majority of the bits in the receiver.                            */
/*   If the receiver's width is even, this operator returns a two     */
/*   bit variable with bit zero set if the number of ZEROes is        */
/*   larger than the number of ONEs, bit one set if the number        */
/*   of ONEs is larger than the number of ZEROes.                     */
/*   The receiver cannot be a constant.                               */
/*   ==========================================                       */
     i := 0;
     nrOfOnes := 0;
     while i < width do
          if (2 power(i) & val) != 0 then
               nrOfOnes := nrOfOnes + 1
          fi;
          i := i +1
     od;
     if width % 2 = 1 then
          if nrOfOnes <= (width div(2)) then
               returnBI := new(BoundedInteger) init(0,1)
          else
               returnBI := new(BoundedInteger) init(1,1)
          fi
     else
          if nrOfOnes = (width div(2)) then
               returnBI := new(BoundedInteger) init(0,2)
          else
               if nrOfOnes > (width div(2)) then
                    returnBI := new(BoundedInteger) init(2,2)
               else
                    returnBI := new(BoundedInteger) init(1,2)
               fi
          fi
     fi;
     return(returnBI).

mergeFromTo(aBI : BoundedInteger; from, to : Integer) : BoundedInteger
| i, part1, part2, part3 : Integer |
/*   ========= Shifting bitfield merge ===========     */
/*   Parameter 1 (aBI) is merged into the receiver (which      */
/*   cannot be a constant) starting at the bit indicated by    */
/*   parameter 2, ending at the bit indicated by parameter     */
/*   3 (with end wrap-around, parameter 2 may have a           */
/*   larger value than parameter 3). Parameter 1 is left-      */
/*   extended with ZEROes if too few bits are present.         */
/*   Parameter 1 may be a constant. Both parameter 2           */
/*  and 3 should be in the range 0..(receiver width#- 1).      */
/*  ==========================================                 */
    if to >= from then
        part1 := val & (2 power(from) - 1);
        part2 := (aBI getVal() & (2 power(to - from + 1) - 1)) * 2 power(from);
        part3 := val & ((2 power(width - to - 1) - 1) * 2 power(to + 1))
    else
        i := 1;
        part1 := aBI getVal();
        while i <= (width - from) do
            part1 := part1 div(2);
            i := i + 1
        od;
        part1 := part1 & (2 power(to + 1) - 1);
        part2 := val & ((2 power(from - to - 1) - 1) * (2 power(to + 1) - 1));
        part3 := (aBI getVal() & (2 power(width - from) - 1)) * 2 power(from)
    fi;
    return(new(BoundedInteger) init(part1 | part2 | part3,width)).

mergeMask(mergeBI, maskBI : BoundedInteger) : BoundedInteger
| tempVal, newWidth : Integer |
/*  ========= Masked merge operator ==========  */
/*  It returns an integer containing bits from the receiver */
/*  (where 'maskBI' bits are ZERO) merged with bits      */
/*  from 'mergeBI' (where 'maskBI' bits are ONE).        */
/*  At most two of the three values involved may be      */
/*  constants, the other(s) should have the same width.  */
/*  The result will also have this width.                */
/*  ======================================               */
    if width = 0 then
        if mergeBI getWidth() = 0 then
            newWidth := maskBI getWidth()
        else
            newWidth := mergeBI getWidth()
        fi
    else
        newWidth := width
    fi;
    tempVal := (val & (maskBI width(newWidth) not()) getVal()) |
                (mergeBI logicAND(maskBI) getVal());
    return(new(BoundedInteger) init(tempVal,width)).

msomask() : BoundedInteger
| i : Integer |
/*  ========== Most significant one bit mask ===========   */
/*  Returns a variable (with the same width as the receiver)   */
/*  with a single ONE at the position of the most significant  */
/*  ONE bit in the receiver. Returns 0 if the receiver does not */
/*  contain any ONEs. The receiver cannot be a constant.       */
/*  ==========================================                 */
    if val = 0 then
        i := 0
    else
        i := width - 1;
        while (2 power(i) & val) = 0 do
            i := i - 1
        od
    fi;
    return(new(BoundedInteger) init(2 power(i) & val, width)).
```

```
msone() : BoundedInteger
| i : Integer |
/*   ========= Most significant one bit position =========   */
/*   Returns a variable (with the same width as the receiver)   */
/*   with as value the bit position of the most significant ONE in   */
/*   the receiver. The returned value equals the width of the   */
/*   receiver if the receiver does not contain any ONEs.   */
/*   The receiver cannot be a constant.   */
/*   ===========================================   */
    if val = 0 then
        i := width
    else
        i := width - 1;
        while (2 power(i) & val) = 0 do
            i := i - 1
        od
    fi;
    return(new(BoundedInteger) init(i, width)).

mszero() : BoundedInteger
| i, tempVal : Integer |
/*   ========= Most significant zero bit position =========   */
/*   Returns a variable (with the same width as the receiver)   */
/*   with as value the bit position of the most significant ZERO   */
/*   in the receiver. The returned value equals the width of the   */
/*   receiver if the receiver does not contain any ZEROes.   */
/*   The receiver cannot be a constant.   */
/*   ===========================================   */
    tempVal := self not() getVal();
    if tempVal = 0 then
        i := width
    else
        i := width - 1;
        while (2 power(i) & tempVal) = 0 do
            i := i - 1
        od
    fi;
    return(new(BoundedInteger) init(i, width)).

mszmask() : BoundedInteger
| i, tempVal : Integer |
/*   ========= Most significant zero bit mask =========   */
/*   Returns a variable (with the same width as the receiver)   */
/*   with a single ONE at the position of the most significant   */
/*   ZERO bit in the receiver. Returns 0 if the receiver does not   */
/*   contain any ZEROes. The receiver cannot be a constant.   */
/*   ===========================================   */
    tempVal := self not() getVal();
    if tempVal = 0 then
        i := 0
    else
        i := width - 1;
        while (2 power(i) & tempVal) = 0 do
            i := i - 1
        od
    fi;
    return(new(BoundedInteger) init(2 power(i) & tempVal, width)).

neg() : BoundedInteger
| returnVal : Integer |
/*  ============ Two's complement negative ==========  */
/*  Return the two's complement negative value of the receiver.  */
/*  This value is calculated by complementing the bits and then  */
/*  adding one (as if the operators not and inc were applied).  */
/*  The receiver cannot be a constant.  */
/*  ============================================  */
    if val = 0 then
        returnVal := 0
    else
        returnVal := 2 power(width) - val
    fi;
    return(new(BoundedInteger) init(returnVal, width)).

not() : BoundedInteger
/*  ========== Complement bits (logical NOT) ==========  */
/*  Returns the one's complement negative value of the receiver */
/*  (all bits inverted), with the same width as the receiver.  */
/*  The receiver cannot be a constant.  */
/*  ============================================  */
    return(new(BoundedInteger) init(2 power(width) - 1 - val, width)).

notEqual(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ============ (Un)signed compare 'not equal' ============  */
/*  Returns a single bit Bounded Integer, with value 1 meaning that  */
/*  the test passed, value 0 meaning that the test failed. If neither  */
/*  side is a constant, then their widths must be equal. If both  */
/*  sides are constant, then this operator returns a constant with  */
/*  values 0 or 1 for failing or passing the test.  */
/*  ============================================  */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
        newWidth := 1
    fi;
    if val != aBI getVal() then
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

onecnt() : BoundedInteger
| i, nrOfOnes : Integer |
/*  ========== Count number of ONEs in word ========  */
/*  Returns a variable with the same width as the receiver,  */
/*  containing a value which gives the number of ONE bits in  */
/*  the receiver. The receiver cannot be a constant.  */
/*  ============================================  */
    i := 0;
    nrOfOnes := 0;
    while i < width do
        if (2 power(i) & val) != 0 then
            nrOfOnes := nrOfOnes + 1
        fi;
        i := i +1
    od;
    return(new(BoundedInteger) init(nrOfOnes, width)).

ones(nrOfOnes : Integer) : BoundedInteger
```

```
/*  =============== Generate all ONEs ==============   */
/*  Returns a variable with the width given by the parameter's   */
/*  value, containing all ONEs (so, the result is not really a   */
/*  'variable'...).                                              */
/*  ============================================                 */
    return(new(BoundedInteger) init(2 power(nrOfOnes) - 1, nrOfOnes)).

opty() : BoundedInteger
I i, parity : Integer I
/*  ================ Odd parity bit ==============   */
/*  Return a single bit variable containing an odd parity flag  */
/*  for the receiver (value 1 if the number of ONEs in the      */
/*  receiver is odd).The receiver cannot be a constant.         */
/*  ============================================                */
    i := 0;
    parity := 0;
    while i < width do
        if (val & 2 power(i)) != 0 then
            parity := 1 - parity
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(parity,1)).

printString() : String
    return(val asString()).

rev() : BoundedInteger
I i, tempVal : Integer I
/*  ========== Reverse all bits MSB <-> LSB ==========   */
/*  Returns a variable with the same width as the receiver,     */
/*  it's value is the receiver's value with all bits reversed back  */
/*  to front (leftmost receiver bit is rightmost result bit,    */
/*  etcetera). The receiver cannot be a constant.               */
/*  ============================================                */
    i := 0;
    tempVal := 0;
    while i < width do
        if (val & 2 power(i)) != 0 then
            tempVal := 2 power(width - i - 1) I tempVal
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal, width)).

rhsmply(aBI : BoundedInteger) : BoundedInteger
I newVal, newWidth, otherVal, otherWidth : Integer I
/*  ================ Right hand signed multiply ================   */
/*  The value at the right hand side cannot be a constant. The result's width  */
/*  is the total of the widths of the receiver and right hand side value (where  */
/*  a constant has width zero). If this width is not enough to hold the result,  */
/*  then the excess bits are chopped off (an overflow is ignored). The result  */
/*  should be interpreted as a signed value!                                   */
/*  ==================================================                          */
    otherVal := aBI getVal();
    otherWidth := aBI getWidth;
    newWidth := width + otherWidth;
    if otherWidth = 1 then
        if otherVal >= 1 then
            otherVal := otherVal - 2
        fi
    else
        if otherVal >= (2 power (otherWidth - 1)) then
            otherVal := otherVal - (2 power(otherWidth))
        fi
    fi;
    newVal := (otherVal * val) % (2 power(newWidth));
    return(new(BoundedInteger) init(newVal, newWidth)).

rol(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  ================ Rotate left ================== */
/*  Rotate the receiver left (wrap around) by the number of bits  */
/*  given by the parameter.                                       */
/*  ============================================== */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        tempVal := 2 * tempVal;
        if tempVal >= (2 power(width)) then
            tempVal := (tempVal & (2 power(width) - 1)) + 1
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal,width)).

ror(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer; returnBI : BoundedInteger |
/*  ================== Rotate right ================ */
/*  Rotate the receiver right (wrap around) by the number of bits */
/*  given by the parameter.                                       */
/*  ============================================ */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        if (tempVal & 1) = 1 then
            tempVal := ((tempVal - 1) div(2)) + (2 power(width-1))
        else
            tempVal := tempVal div(2)
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal,width)).

sar(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  ============ Shift arithmetic right ============ */
/*  Shift the receiver arithmetic right (sign preserved) by the */
/*  number of bits given by the parameter.                      */
/*  ======================================== */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        if (tempVal & 1) = 1 then
            tempVal := (tempVal - 1) div(2)
        else
            tempVal := tempVal div(2)
        fi;
        i := i + 1
    od;
    if (val & 2 power(width - 1)) != 0 then
        tempVal := tempVal | ((2 power(numberOfBits) - 1) *
                                (2 power(width - numberOfBits)))
    fi;
    return(new(BoundedInteger) init(tempVal,width)).

setVal(newVal : Integer) : BoundedInteger
/*  ============== Set receiver's value =============== */
/*  Sets the receiver's value to the value given by the parameter. */
/*  Note: This method is not an operator in IDaSS.              */
/*  ============================================== */
    val := newVal;
    return(self).

setWidth(newWidth : Integer) : BoundedInteger
/*  ============== Set receiver's width =============== */
/*  Sets the receiver's width to the value given by the parameter. */
/*  Note: This method is not an operator in IDaSS.              */
/*  ============================================== */
    width := newWidth;
    return(self).

shl(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  ============= Shift logical / arithmetic left ===============     */
/*  Shift the receiver logical/arithmetic left by the number of bits given     */
/*  by the parameter, introducing ZEROes in the least significant bit(s).  */
/*  ================================================     */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        tempVal:=2 * tempVal;
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal, width) width(width)).

shr(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  ================ Shift logical right ================== */
/*  Shift the receiver logical right by the number of bits given by       */
/*  the parameter, introducing ZEROes in the most significant bit(s). */
/*  ================================================ */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        if (tempVal & 1) = 1 then
            tempVal := (tempVal - 1) div(2)
        else
            tempVal := tempVal div(2)
        fi;
        i := i + 1
    od;
    return(new(BoundedInteger) init(tempVal,width)).

signed(newWidth : Integer) : BoundedInteger
| tempVal : Integer |
/*  ============== Sign extend a value ============ */
/*  Returns the receiver sign extended to the width given by */
/*  the parameter (which must be an integer equal to- or      */
/*  greater than the width of the receiver).                    */
/*  ========================================== */
    if (val & 2 power(width - 1)) != 0 then
        tempVal := val + ((2 power(newWidth - width) - 1) * 2 power(width))
    else
        tempVal := val
    fi;
    return(new(BoundedInteger) init(tempVal, newWidth)).

signedGreater(aBI : BoundedInteger) : BoundedInteger
| calcWidth, tempVal, otherVal : Integer; returnBI : BoundedInteger |
/*  ======== Signed compare 'greater than' ========      */
/*  Returns a BoundedInteger with width 1, value 0: false,    */
/*  value 1: true. If one side is a constant which is outside  */
/*  the range for the other side, then this is considered an   */
/*  error. Cannot be used between constants.                   */
/*  ========================================        */
    tempVal := val;
    otherVal := aBI getVal();
    if width = 0 then
        calcWidth := aBI getWidth()
    else
        calcWidth := width
    fi;
    if tempVal >= 2 power(calcWidth - 1) then
        tempVal := tempVal - 2 power(calcWidth)
    fi;
    if otherVal >= 2 power(calcWidth - 1) then
        otherVal := otherVal - 2 power(calcWidth)
    fi;
    if tempVal > otherVal then
        returnBI := new(BoundedInteger) init(1,1)
    else
        returnBI := new(BoundedInteger) init(0,1)
    fi;
    return(returnBI).

signedGreaterEqual(aBI : BoundedInteger) : BoundedInteger
| calcWidth, tempVal, otherVal : Integer; returnBI : BoundedInteger |
/*  ====== Signed compare 'greater than equal' =====      */
/*  Returns a BoundedInteger with width 1, value 0: false,    */
/*  value 1: true. If one side is a constant which is outside  */
/*  the range for the other side, then this is considered an   */
/*  error. Cannot be used between constants.                   */
/*  ========================================        */
    tempVal := val;
    otherVal := aBI getVal();
    if width = 0 then
        calcWidth := aBI getWidth()
    else
        calcWidth := width
    fi;
    if tempVal >= 2 power(calcWidth - 1) then
        tempVal := tempVal - 2 power(calcWidth)
    fi;
    if otherVal >= 2 power(calcWidth - 1) then
        otherVal := otherVal - 2 power(calcWidth)
    fi;
    if tempVal >= otherVal then
        returnBI := new(BoundedInteger) init(1,1)
    else
        returnBI := new(BoundedInteger) init(0,1)
    fi;
    return(returnBI).

signedLess(aBI : BoundedInteger) : BoundedInteger
| calcWidth, tempVal, otherVal : Integer; returnBI : BoundedInteger |
/*  =========  Signed compare 'less than'  ========    */
/*  Returns a BoundedInteger with width 1, value 0: false,    */
/*  value 1: true. If one side is a constant which is outside    */
/*  the range for the other side, then this is considered an    */
/*  error. Cannot be used between constants.    */
/*  ======================================    */
    tempVal := val;
    otherVal := aBI getVal();
    if width = 0 then
        calcWidth := aBI getWidth()
    else
        calcWidth := width
    fi;
    if tempVal >= 2 power(calcWidth - 1) then
        tempVal := tempVal - 2 power(calcWidth)
    fi;
    if otherVal >= 2 power(calcWidth - 1) then
        otherVal := otherVal - 2 power(calcWidth)
    fi;
    if tempVal < otherVal then
        returnBI := new(BoundedInteger) init(1,1)
    else
        returnBI := new(BoundedInteger) init(0,1)
    fi;
    return(returnBI).

signedLessEqual(aBI : BoundedInteger) : BoundedInteger
| calcWidth, tempVal, otherVal : Integer; returnBI : BoundedInteger |
/*  ======  Signed compare 'less than or equal'  ======    */
/*  Returns a BoundedInteger with width 1, value 0: false,    */
/*  value 1: true. If one side is a constant which is outside    */
/*  the range for the other side, then this is considered an    */
/*  error. Cannot be used between constants.    */
/*  ======================================    */
    tempVal := val;
    otherVal := aBI getVal();
    if width = 0 then
        calcWidth := aBI getWidth()
    else
        calcWidth := width
    fi;
    if tempVal >= 2 power(calcWidth - 1) then
        tempVal := tempVal - 2 power(calcWidth)
    fi;
    if otherVal >= 2 power(calcWidth - 1) then
        otherVal := otherVal - 2 power(calcWidth)
    fi;
    if tempVal <= otherVal then
        returnBI := new(BoundedInteger) init(1,1)
    else
        returnBI := new(BoundedInteger) init(0,1)
    fi;
    return(returnBI).
```

```
smply(aBI : BoundedInteger) : BoundedInteger
| newVal, newWidth, otherVal, otherWidth, tempVal : Integer |
/*  ================= Signed multiply ==================  */
/*  Neither side can be a constant. The result's width is the total of the  */
/*  widths of receiver and right hand side value. An overflow cannot        */
/*  happen here. The result should be interpreted as a signed value!        */
/*  ====================================================  */
    otherVal := aBI getVal();
    otherWidth := aBI getWidth();
    newWidth := width + otherWidth;
    tempVal := val;
    if width = 1 then
        if val >= 1 then
            tempVal := tempVal - 2
        fi
    else
        if val >= (2 power (width - 1)) then
            tempVal := val - (2 power(width))
        fi
    fi;
    if otherWidth = 1 then
        if otherVal >= 1 then
            otherVal := otherVal - 2
        fi
    else
        if otherVal >= (2 power (otherWidth - 1)) then
            otherVal := otherVal - (2 power(otherWidth))
        fi
    fi;
    newVal := (tempVal * otherVal) % (2 power(newWidth));
    return(new(BoundedInteger) init(newVal, newWidth)).

sol(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  =================== Shift ones left ==================  */
/*  Shift the receiver left by the number of bits given by the parameter,  */
/*  introducing ONEs in the least significant bit(s).                      */
/*  ===================================================  */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        tempVal:=2 * tempVal;
        i := i + 1
    od;
    tempVal := tempVal + (2 power(numberOfBits) - 1);
    return(new(BoundedInteger) init(tempVal, width) width(width)).

sor(numberOfBits : Integer) : BoundedInteger
| tempVal, i : Integer |
/*  ============== Shift ones right ===============  */
/*  Shift the receiver right by the number of bits given by the  */
/*  parameter, introducing ONEs in the most significant bit(s).  */
/*  ========================================  */
    tempVal := val;
    i := 1;
    while i <= numberOfBits do
        if (tempVal & 1) = 1 then
            tempVal := (tempVal - 1) div(2)
        else
```

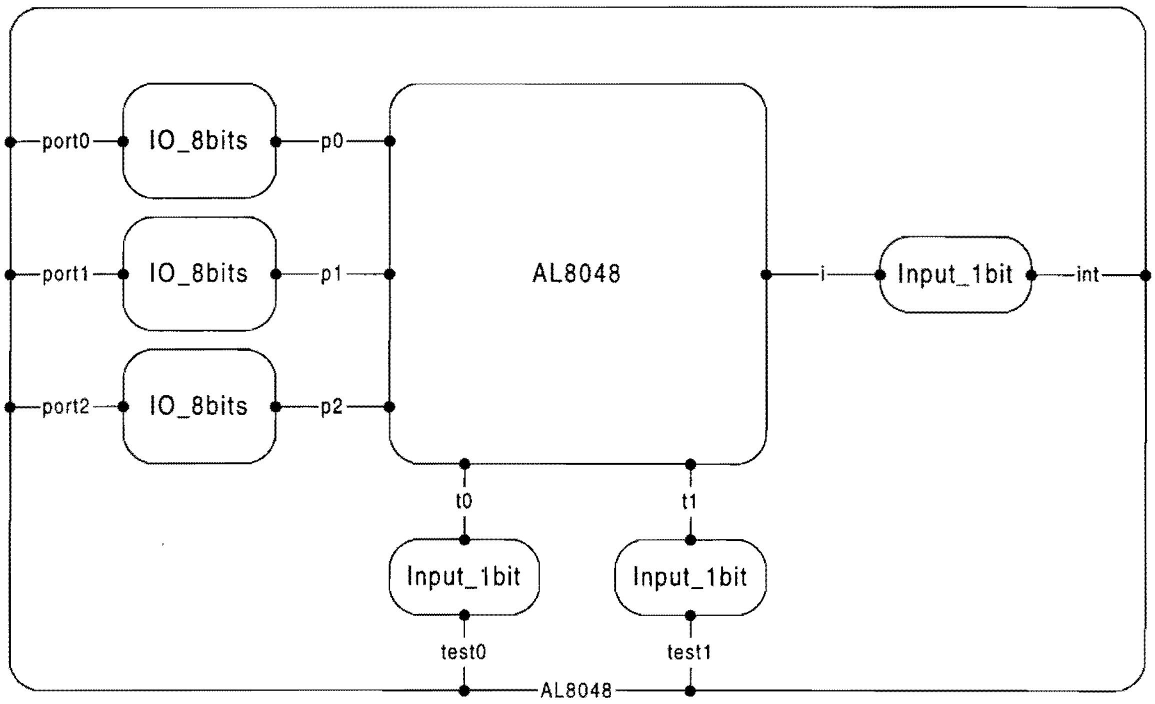
```
            tempVal := tempVal div(2)
        fi;
        i := i + 1
    od;
    tempVal := tempVal + ((2 power(numberOfBits) - 1) *
                (2 power(width - numberOfBits)));
    return(new(BoundedInteger) init(tempVal,width)).

sub(aBI : BoundedInteger) : BoundedInteger
| newVal, newWidth : Integer |
/*  ==================== Subtraction ==================== */
/*  Subtraction (two's complement/unsigned). If neither side is a        */
/*  constant, then their widths must be equal. If both sides are constant, */
/*  then the result may not be negative. If only one side is a constant,  */
/*  then it's value should be representative with the number of bits in the */
/*  other side's word.                                                    */
/*  ================================================== */
    if width = 0 then
        newWidth := aBI getWidth()
    else
        newWidth := width
    fi;
    newVal := val - (aBI getVal());
    if newVal < 0 then
        newVal := 2 power(newWidth) + newVal
    fi;
    return(new(BoundedInteger) init(newVal, newWidth)).

umply(aBI : BoundedInteger) : BoundedInteger
| newVal, newWidth : Integer |
/*  ==================== Unsigned multiply =================  */
/*  Unsigned multiply. The result's width is the total of the widths of receiver */
/*  and the right hand side value (where a constant has width zero). If this    */
/*  width is not enough to hold the result, then the excess bits are chopped    */
/*  off (an overflow is ignored). If the receiver and right hand side are both  */
/*  constants, then the result will also be a constant (no overflow is possible */
/*  in that case).                                                              */
/*  ====================================================  */
    newWidth := aBI getWidth() + width;
    newVal := aBI getVal() * val;
    if newWidth != 0 then
        newVal := newVal % (2 power(newWidth))
    fi;
    return(new(BoundedInteger) init(newVal, newWidth)).

unsignedGreater(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ========= Unsigned compare 'greater than' ===========  */
/*  Returns a single bit Bounded Integer, with value 1 meaning that  */
/*  the test passed, value 0 meaning that the test failed. If neither */
/*  side is a constant, then their widths must be equal. If both      */
/*  sides are constant, then this operator returns a constant with    */
/*  values 0 or 1 for failing or passing the test.                    */
/*  ============================================  */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
        newWidth := 1
    fi;
    if val > aBI getVal() then
```

```
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

unsignedGreaterEqual(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ====== Unsigned compare 'greater than or equal' =======  */
/*  Returns a single bit Bounded Integer, with value 1 meaning that  */
/*  the test passed, value 0 meaning that the test failed. If neither  */
/*  side is a constant, then their widths must be equal. If both  */
/*  sides are constant, then this operator returns a constant with  */
/*  values 0 or 1 for failing or passing the test.  */
/*  ==============================================  */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
        newWidth := 1
    fi;
    if val >= aBI getVal() then
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

unsignedLess(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ========== Unsigned compare 'less than' ============  */
/*  Returns a single bit Bounded Integer, with value 1 meaning that  */
/*  the test passed, value 0 meaning that the test failed. If neither  */
/*  side is a constant, then their widths must be equal. If both  */
/*  sides are constant, then this operator returns a constant with  */
/*  values 0 or 1 for failing or passing the test.  */
/*  ==============================================  */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
        newWidth := 1
    fi;
    if val < aBI getVal() then
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

unsignedLessEqual(aBI : BoundedInteger) : BoundedInteger
| newWidth : Integer; returnBI : BoundedInteger |
/*  ======== Unsigned compare 'less than or equal' =======  */
/*  Returns a single bit Bounded Integer, with value 1 meaning that  */
/*  the test passed, value 0 meaning that the test failed. If neither  */
/*  side is a constant, then their widths must be equal. If both  */
/*  sides are constant, then this operator returns a constant with  */
/*  values 0 or 1 for failing or passing the test.  */
/*  ==============================================  */
    if (width = 0) & (aBI getWidth() = 0) then
        newWidth := 0
    else
```

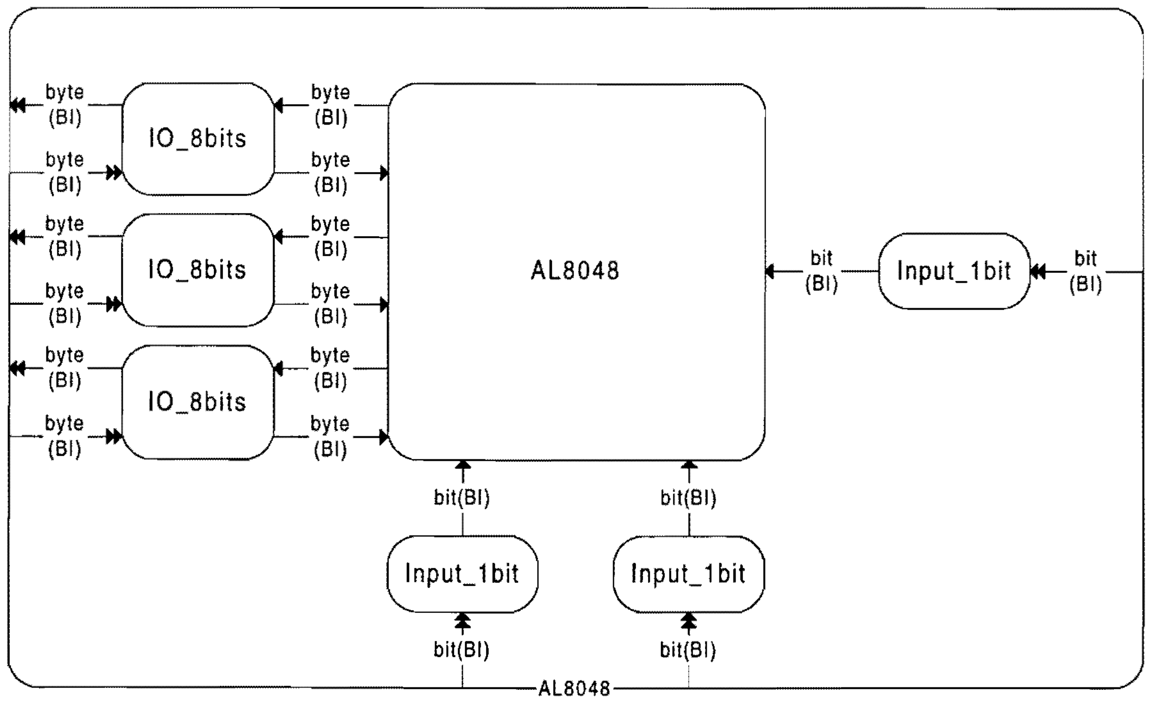
```
        newWidth := 1
    fi;
    if val <= aBI getVal() then
        returnBI := new(BoundedInteger) init(1, newWidth)
    else
        returnBI := new(BoundedInteger) init(0, newWidth)
    fi;
    return(returnBI).

width(newWidth : Integer) : BoundedInteger
| returnBI : BoundedInteger |
/*  ========== Set / change width of a value ============ */
/*  Return the receiver with a new width given by the parameter. */
/*  The receiver's bits are simply truncated from the MSB side if */
/*  too wide, ZEROes are padded left if too narrow.              */
/*  ============================================================= */
    if newWidth = 0 then
        returnBI := new(BoundedInteger) init(val, 0)
    else
        returnBI := new(BoundedInteger) init(val &
                        (2 power(newWidth) - 1), newWidth)
    fi;
    return(returnBI).

zerocnt() : BoundedInteger
| i, nrOfOnes : Integer |
/*  ======== Count number of ZEROes in word ======= */
/*  Returns a variable with the same width as the receiver,  */
/*  containing a value which gives the number of ZERO bits */
/*  in the receiver. The receiver cannot be a constant.      */
/*  ======================================================== */
    i := 0;
    nrOfOnes := 0;
    while i < width do
        if (2 power(i) & val) != 0 then
            nrOfOnes := nrOfOnes + 1
        fi;
        i := i +1
    od;
    return(new(BoundedInteger) init(width - nrOfOnes, width)).

zeroes(nrOfZeroes : Integer) : BoundedInteger
/*  ============= Generate all ZEROes =============     */
/*  Returns a variable with the width given by the parameter's   */
/*  value, containing all ZEROes (so, the result is not really a  */
/*  'variable'...).                                               */
/*  ============================================================  */
    return(new(BoundedInteger) init(0, nrOfZeroes)).
```

Appendix C: AL 8048 POOSL specification

C.1 Instance Structure Diagram



C.2 Message Flow Diagram



C.3 AL 8048 behaviour specification

AL8048\{int, test1, port2, port1, port0, test0}

C.4 Process class definitions

```
process class AL8048()
/* no superclass */

instance variables
pclow: BoundedInteger; pchigh: BoundedInteger; psw: BoundedInteger;
accu: BoundedInteger; f1: BoundedInteger; t0reg: BoundedInteger; ir: BoundedInteger;
ireg: BoundedInteger; p1reg: BoundedInteger; p0reg: BoundedInteger; tcntCmd: Boolean;
temp8: BoundedInteger; tctrl: BoundedInteger; pcbuf: BoundedInteger;
temp9: BoundedInteger; inten: BoundedInteger; t1reg: BoundedInteger; irVal: Integer;
ram: Array; p2reg: BoundedInteger; rom: Array; tintr: BoundedInteger; tmrreg: BoundedInteger

communication channels
test1, port2, test0, int, port1, port0

message interface
port2 ? byte(UNKNOWN);
test1 ? bit(UNKNOWN);
test0 ? bit(UNKNOWN);
int ? bit(UNKNOWN);
port1 ! byte(UNKNOWN);
port2 ! byte(UNKNOWN);
port0 ! byte(UNKNOWN);
port0 ? byte(UNKNOWN);
port1 ? byte(UNKNOWN)

initial method call
init()()

instance methods
accuInstr()()
| temp : Integer |
/*  =========================================   */
/*  Decode & execute (mostly) accu related instructions .   */
/*  =========================================   */
    temp := irVal & 240;
    if temp <= 96 then
        if temp <= 48 then
            if temp <= 16 then
                if temp = 0 then
                /*  %0000.0111 : DEC A   */
                    accu := accu dec()
                else
                /*  %0001.0111 : INC A   */
                    accu := accu inc()
                fi
            else
                if temp = 32 then
                /*  %0010.0111 : CLR A   */
                    accu setVal(0)
                else
                /*  %0011.0111 : CPL A   */
                    accu := accu not()
                fi
            fi
        else
            if temp <= 80 then
```

```
                if temp = 64 then
                /*   %0100.0111 : SWAP A  */
                     accu := accu rol(4)
                else
                /*   %0101.0111 : DA A */
                     if (psw at(6) getVal() = 1) | (accu fromTo(0,3)
                                                          getVal() > 9) then
                          if (psw at(7) getVal() = 1) | (accu fromTo(4,7)
                                                                   getVal() > 8) then
                                                          accu := accu add(new(
                                                                   BoundedInteger) init(102,0));
                               psw:=psw logicOR(new(
                                                                   BoundedInteger) init(128,0))
                          else
                               accu := accu add(new(
                                                                   BoundedInteger) init(6,0))
                          fi
                     else
                          if (psw at(7) getVal() = 1) | (accu fromTo(4,7)
                                                                   getVal() > 9) then
                               accu := accu add(new(
                                                                   BoundedInteger) init(96,0));
                               psw := psw logicOR(new(
                                                                   BoundedInteger) init(128,0))
                          fi
                     fi
                fi
           else
           /*   %0110.0111 : RRC A    */
                temp9 := psw at(7) concat(accu);
                accu := temp9 fromTo(1,8);
                psw := temp9 at(0) concat(psw fromTo(0,6))
           fi
      fi
 else
      if temp <= 192 then
           if temp <= 144 then
                if temp = 112 then
                /*   %0111.0111 : RR A      */
                     accu := accu ror(1)
                else
                /*   %1001.0111 : CLR C     */
                     psw := psw logicAND(new(BoundedInteger)
                                                   init(127,0))
                fi
           else
                if temp = 160 then
                /*   %1010.0111 : CPL C     */
                     psw := psw at(7) not() concat(psw fromTo(0,6))
                else
                /*   %1100.0111 : MOV A,PSW */
                     accu setVal(psw getVal())
                fi
           fi
      else
           if temp <= 224 then
                if temp = 208 then
                /*   %1101.0111 : MOV PSW,A */
                     psw setVal(accu getVal())
                else
                /*   %1110.0111 : RL A      */
```

```
                    accu := accu rol(1)
                fi
            else
            /*  %1111.0111 : RLC A     */
                temp9 := accu concat(psw at(7));
                accu := temp9 fromTo(0,7);
                psw := temp9 at(8) concat(psw fromTo(0,6))
            fi
        fi
    fi.

addToAccu()()
/*  ========================================= */
/*  Method to perform ADD(C) operations and update flags. */
/*  ========================================= */
    temp9 := accu addCin(temp8, ir at(4) logicAND(psw at(7)));
    psw := temp9 at(8) concat(accu at(4) logicXOR(temp8 at(4)
        logicXOR(temp9 at(4)))) concat(psw fromTo(0,5));
    accu := temp9 fromTo(0,7).

branchInstr()()
| temp, address : Integer ; jump : Boolean |
/*  ============================       */
/*  Decode & execute branch instructions.   */
/*  ============================       */
    temp := irVal & 240;
    if temp = 16 then
    /*  %0001.0110 : JTF addr */
        if tintr getVal() = 0 then
            waitForClock()();
            pclow inc()
        else
            tintr setVal(0);
            jumpInPage()()
        fi
    else
        int ? bit(ireg);
        test0 ? bit(t0reg);
        test1 ? bit(t1reg);
        jump := ((temp=32) & (t0reg isZero()))          /*  JNT0 addr  */
            | ((temp=48) & (t0reg isOne()))             /*  JT0 addr   */
            | ((temp=64) & (t1reg isZero()))            /*  JNT1 addr  */
            | ((temp=80) & (t1reg isOne()))             /*  JT1 addr   */
            | ((temp=112) & (f1 isOne()))               /*  JF1 addr   */
            | ((temp=128) & (ireg isZero()))            /*  JNI addr   */
            | ((temp=144) & (accu isZero() not()))      /*  JNZ addr   */
            | ((temp=176) & (psw at(5) isOne()))        /*  JF0 addr   */
            | ((temp=192) & (accu isZero()))            /*  JZ addr    */
            | ((temp=224) & (psw at(7) isZero()))       /*  JNC addr   */
            | ((temp=240) & (psw at(7) isOne()));       /*  JC addr    */
        if jump then
            jumpInPage()()
        else
            waitForClock()();
            pclow inc()
        fi
    fi.

callOrJmpInstr()()
| address : Integer |
/*  ===================================  */
/*  Decode & execute JMP and CALL instructions.  */
/*  ===================================  */
/*  %xxxx.0100 : JMP addr / CALL addr*/
    fetchImmediate()();
    if (irVal & 16) = 1 then
        /*  %xxx1.0100 : CALL addr    */
        address := psw at(2) concat(psw at(2) not()) concat(psw fromTo(0,1))
            concat(new(BoundedInteger) zeroes(1)) getVal() + 1;
        ram put(address, pclow fromTo(0,7) getVal());
        ram put(address + 1, psw fromTo(4,7) concat(pchigh at(0))
            concat (pclow fromTo(8,10)) getVal());
        psw := psw fromTo(3,7) concat(psw fromTo(0,2) inc())
    fi;
    jumpLong()().

fetchAtAccu()()
| address : Integer |
/*  ================================================  */
/*  Method to fetch data from program memory (rom) in the current  */
/*  program page indexed by the ACCU.                              */
/*  ================================================  */
    address := pchigh concat(pclow fromTo(8,10)) concat(accu) getVal() + 1;
    temp8 := rom get(address);
    waitForClock()().

fetchImmediate()()
| address : Integer |
/*  ============================================  */
/*  Method to fetch 2nd instruction byte (mostly immediate data).*/
/*  ============================================  */
    address := pchigh concat(pclow) getVal() + 1;
    temp8 setVal(rom get(address));
    waitForClock()();
    pclow inc().

fetchIndirect()()
| address : Integer |
/*  ===================================  */
/*  Method to fetch an indirectly addressed location. */
/*  ===================================  */
    address := new(BoundedInteger) copiesOf(2, psw at(4)) concat(new(BoundedInteger)
        zeroes(2)) concat(ir at(0)) getVal() + 1;
    temp8 := ram get(ram get(address) + 1).

fetchRegister()()
| address : Integer |
/*  ==========================================    */
/*  This method is used to fetch a directly addressed register.   */
/*  ==========================================    */
    address := new(BoundedInteger) copiesOf(2, psw at(4)) concat(ir fromTo(0,2))
              getVal() + 1;
    temp8 setVal(ram get(address)).

flagInstr()()
| temp : Integer |
/*  ========================================    */
/*  Decode & execute (mostly) flag related instructions .     */
/*  ========================================    */
    temp := irVal & 240;
    if temp <= 112 then
        if temp <= 64 then
```

```
            if temp <= 48 then
                if temp <= 16 then
                /*   %000x.0101 : DIS I / EN I          */
                    inten := inten fromTo(1,2) concat(ir at(4) not())
                else
                /*   %001x.0101 : DIS TCNTI / EN TCNTI    */
                    inten := inten at(2) concat(ir at(4) not())
                                                        concat(inten at(0));
                    tintr setVal(0)
                fi
            else
            /*   %0100.0101 : STRT CNT          */
                if t1reg getVal() = 1 then
                    tctrl setVal(6)
                else
                    tctrl setVal(2)
                fi;
                tintr setVal(0);
                tcntCmd := true
            fi
        else
            if temp = 80 then
            /*   %0101.0101 : STRT T   */
                tctrl setVal(1);
                tintr setVal(0);
                tcntCmd := true
            else
            /*   %0110.0101 : STOP TCNT  */
                tctrl setVal(0);
                tintr setVal(0);
                tcntCmd := true
            fi
        fi
    else
        if temp <= 160 then
            if temp <= 144 then
                if temp = 128 then
                /*   %1000.0101 : CLR F0   */
                    psw := psw logicAND(new(BoundedInteger)
                                                        init(223,0))
                else
                /*   %1001.0101 : CPL F0   */
                    psw := psw fromTo(6,7) concat(psw at(5) not())
                        concat(psw fromTo(0,4))
                fi
            else
            /*   %1010.0101 : CLR F1   */
                f1 setVal(0)
            fi
        else
            if temp <= 208 then
                if temp = 176 then
                /*   %1011.0101 : CPL F1   */
                    f1 := f1 not()
                else
                /*   %110x.0101 : SEL RBx */
                    psw := psw fromTo(5,7) concat(ir at(4))
                                                        concat(psw fromTo(0,3))
                fi
            else
            /*   %111x.0101 : SEL MBx */
```

```
                pcbuf := pcbuf at(1) concat(ir at(4))
            fi
        fi
    fi.

indirectInstr()()
I temp : Integer I
/*  ================================== */
/*  Decode & execute instructions with indirects.*/
/*  ================================== */
    temp := irVal & 240;
    if temp != 0 then
        if (irVal & 192) = 128 then
            if temp = 176 then
            /*  %1011.000x : MOV @R,#data   */
                fetchImmediate()()
            else
            /*  %1010.000x : MOV @R,A   */
                temp8 setVal(accu getVal())
            fi;
            storeIndirect()()
        else
            fetchIndirect()();
            if temp <= 80 then
                if temp <= 48 then
                    if temp <= 32 then
                        if temp = 16 then
                        /*  %0001.000x : INC @R  */
                            temp8 inc();
                            storeIndirect()()
                        else
                        /*  %0010.000x : XCH A,@R   */
                            temp9 := new(BoundedInteger)
                                                                zeroes(1) concat(temp8);
                            temp8 := accu copy();
                            storeIndirect()();
                            accu := temp9 fromTo(0,7)
                        fi
                    else
                    /*  %0011.000x : XCHD A,@R */
                        temp9 := new(BoundedInteger) zeroes(1)
                                                        concat(temp8);
                        temp8 := temp8 fromTo(4,7)
                                                        concat(accu fromTo(0,3));
                        storeIndirect()();
                        accu := accu fromTo(4,7)
                                                        concat(temp9 fromTo(0,3))
                    fi
                else
                    if temp = 64 then
                    /*  %0100.000x : ORL A,@R   */
                        accu := accu logicOR(temp8)
                    else
                    /*  %0101.000x : ANL A,@R   */
                        accu := accu logicAND(temp8)
                    fi
                fi
            else
                if temp <= 128 then
                /*  %011x.000x : ADD(C) A,@R      */
                    addToAccu()()
                    else
                        if temp = 208 then
                        /*  %1101.000x : XRL A,@R    */
                            accu := accu logicXOR(temp8)
                        else
                        /*  %1111.000x : MOV A,@R    */
                            accu setVal(temp8 getVal())
                        fi
                    fi
                fi
            fi
    fi.

init()()
/*  ====================  */
/*  Initialize instance variables.  */
/*  ====================  */
    ram := new(Array) size(256);
    initRam()();
    rom := new(Array) size(8192);
    initRom()();
    accu := new(BoundedInteger) init(0,8);
    ir := new(BoundedInteger) init(0,8);
    psw := new(BoundedInteger) init(0,8);
    f1 := new(BoundedInteger) init(0,1);
    temp8 := new(BoundedInteger) init(0,8);
    temp9 := new(BoundedInteger) init(0,9);
    pclow := new(BoundedInteger) init(0,11);
    pchigh := new(BoundedInteger) init(0,2);
    pcbuf := new(BoundedInteger) init(0,2);
    inten := new(BoundedInteger) init(0,3);
    tintr := new(BoundedInteger) init(0,1);
    t0reg := new(BoundedInteger) init(0,1);
    t1reg := new(BoundedInteger) init(0,1);
    ireg := new(BoundedInteger) init(0,1);
    p0reg := new(BoundedInteger) init(0,1);
    p1reg := new(BoundedInteger) init(0,1);
    p2reg := new(BoundedInteger) init(0,1);
    tctrl := new(BoundedInteger) init(0,6);
    tmrreg := new(BoundedInteger) init(0,8);
    tcntCmd := false;
    waitForClock()();
    main()().

initRam()()
/*  ============================================      */
/*  Initialize the "data storage" in ram. Ram contains the register      */
/*  banks, stack and general scratchpad for the core.      */
/*      $00..$07 : Register bank 0.      */
/*      $08..$17 : Stack (8 levels starting at $08 / $09).      */
/*      $18..$1F : Register bank 1.      */
/*      $20..$FF : General scratchpad memory space.      */
/*  Note: Location 1 in the ram array refers to address $00 !      */
/*  ============================================      */
    ram put(1,0).

initRom()()
/*  ========================================  */
/*  This method is used to load the "program" into rom.  */
/*  -----------------------------------------------------------------------  */
/*  Three locations in rom are of special importance :  */
/*  location 1 (address 0)                                          */
/*      Activating reset causes the first instruction to be         */
/*      fetched from this location.                                 */
/*  location 4 (address 3)                                          */
/*      An enable message to int causes a jump to the               */
/*      subroutine at this location (if interrupt is enabled).      */
/*  location 8 (address 7)                                          */
/*      A timer, counter interrupt resulting from timer counter     */
/*      overflow causes a jump to subroutine at this location       */
/*      (if enabled).                                               */
/*  Note: Location 1 in the rom array refers to address $00 !       */
/*  ============================================================    */

    /*  **********   Program : 16 By 8 Unsigned Divide   **********    */
    /*  At Entry:                                                       */
    /*      A = Lower 8 bits of destination operand                     */
    /*      R2 = Upper 8 bits of dividend                               */
    /*      R1 = Divisor in internal memory                             */
    /*  At Exit:                                                        */
    /*      A = Lower 8 bits of result                                  */
    /*      R2 = Remainder                                              */
    /*      C = Set if overflow else cleared                            */
    /*  *************************************************************** */
    /*  [30.000 (R2 I A)] / [236 (R1)] = ???                            */
    /*  *************************************************************** */
    rom put(1,4);       /*  JMP       */
    rom put(2,10);      /*  address   */
    rom put(11,35);     /*  MOV A,    */
    rom put(12,48);     /*  data      */
    rom put(13,185);    /*  MOV R1,   */
    rom put(14,236);    /*  data      */
    rom put(15,186);    /*  MOV R2,   */
    rom put(16,117);    /*  data      */
    rom put(17,42);     /*  XCH A,R2  */
    rom put(18,187);    /*  MOV R3,   */
    rom put(19,8);      /*  data      */
    rom put(20,55);     /*  CPL A     */
    rom put(21,105);    /*  ADD A,R1  */
    rom put(22,55);     /*  CPL A     */
    rom put(23,246);    /*  JC        */
    rom put(24,27);     /*  address   */
    rom put(25,167);    /*  CPL C     */
    rom put(26,4);      /*  JMP       */
    rom put(27,52);     /*  address   */
    rom put(28,105);    /*  ADD A,R1  */
    rom put(29,151);    /*  CLR C     */
    rom put(30,42);     /*  XCH A,R2  */
    rom put(31,247);    /*  RLC A     */
    rom put(32,42);     /*  XCH A,R2  */
    rom put(33,247);    /*  RLC A     */
    rom put(34,230);    /*  JNC       */
    rom put(35,40);     /*  address   */
    rom put(36,55);     /*  CPL A     */
    rom put(37,105);    /*  ADD A,R1  */
    rom put(38,55);     /*  CPL A     */
    rom put(39,4);      /*  JMP       */
    rom put(40,48);     /*  address   */
    rom put(41,55);     /*  CPL A     */
    rom put(42,105);    /*  ADD A,R1  */
    rom put(43,55);     /*  CPL A     */
    rom put(44,230);    /*  JNC       */
    rom put(45,48);     /*  address     */
    rom put(46,105);    /*  ADD A,R1    */
    rom put(47,4);      /*  JMP         */
    rom put(48,49);     /*  address     */
    rom put(49,26);     /*  INC R2      */
    rom put(50,235);    /*  DJNZ R3,    */
    rom put(51,28);     /*  address     */
    rom put(52,151);    /*  CLR C       */
    rom put(53,42);     /*  XCH A,R2    */
    rom put(54,1).

jumpInPage()()
| tempBI : BoundedInteger |
/*  ============================================    */
/*  Read 2nd byte of instruction (to which the PC is pointing now)    */
/*  and load lowest 8 PC bits with the value read. Keep within    */
/*  page of 2nd byte!    */
/*  ============================================    */
    tempBI := new(BoundedInteger) init(rom get(pchigh concat(pclow) getVal() + 1), 8);
    waitForClock()();
    pclow := pclow fromTo(8,10) concat(tempBI).

jumpLong()()
/*  ============================================  */
/*  Load the PC (low part) from the instruction bits 5..7 and the  */
/*  (already fetched) second instruction byte.  */
/*  ============================================  */
    pclow := ir fromTo(5,7) concat(temp8);
    if inten at(2) getVal() = 1 then
    /*  Interrupt in progress forces bank zero     */
        pchigh setVal(0)
    else
    /*  No interrupt in progress */
        pchigh setVal(pcbuf getVal())
    fi.

main()()
| address, temp : Integer |
/*  ============================================    */
/*  Main routine to execute a program in an 8048 processor core.    */
/*  ============================================    */

    /*  Check for and handle interrupts. */
    if inten at(2) getVal() = 0 then
        /*  Masked interrupt sources.  */
        int ? bit(ireg);
        temp := inten fromTo(0,1) logicAND(tintr concat(ireg)) getVal();
        if temp != 0 then
            /*  Basic handling, push PSW, PC. */
            address := psw at(2) concat(psw at(2) not()) concat(psw fromTo(0,1))
                concat(new(BoundedInteger) zeroes(1)) getVal() + 1;
            ram put(address, pclow fromTo(0,7) getVal());
            ram put(address + 1, psw fromTo(4,7) concat(pchigh at(0))
                concat(pclow fromTo(8,10)) getVal());
            psw := psw fromTo(3,7) concat(psw fromTo(0,2) inc());
            /*  Interrupt now in progress...  */
            inten := new(BoundedInteger) ones(1) concat(inten fromTo(0,1));
            pchigh setVal(0);
            if temp != 2 then         /*  External interrupt first.  */
                pclow setVal(3)
            else                    /*  Timer interrupt second.  */
```

```
                pclow setVal(7);
                tintr setVal(0);
            fi
        fi
    fi;

    /*  Fetch Instruction     */
    ir setVal(rom get(pchigh concat(pclow) getVal() + 1));

    /*  Increment Program Counter */
    pclow inc();

    /*  Decode and Execute     */
    irVal := ir getVal();
    temp := irVal & 15;

    if temp <= 4 then
        if temp <= 2 then
            if temp = 2 then
                timerInstr()()
            else
                indirectInstr()()
            fi
        else
            if temp = 3 then
                varInstr()()
            else
                callOrJmpInstr()()
            fi
        fi
    else
        if temp <= 6 then
            if temp = 5 then
                flagInstr()()
            else
                branchInstr()()
            fi
        else
            if temp = 7 then
                accuInstr()()
            else
                registerInstr()()
            fi
        fi
    fi;

    waitForClock()();
    if irVal != 1 then
        main()()
    fi.

registerInstr()()
| temp : Integer |
/*  ================================  */
/*  Decode & execute 'register-instructions'.     */
/*  ================================  */
    temp := irVal & 240;
    if ((irVal & 64) = 0) & (temp != 32) & (temp != 16) then
        if temp <= 128 then
            if temp <= 48 then
                if temp = 0 then
                if irVal = 8 then
                /*  %0000.1000 : INS A,BUS    */
                    waitForClock()();
                    port0 ? byte(accu)
                else
                    if irVal = 9 then
                    /*  %0000.1001 : IN A,P1    */
                        waitForClock()();
                        port1 ? byte(accu)
                    else
                    /*  %0000.1010 : IN A,P2    */
                        waitForClock()();
                        port2 ? byte(accu)
                    fi
                fi
            else
                if irVal = 57 then
                /*  %0011.1001 : OUTL P1,A    */
                    waitForClock()();
                    port1 ! byte(accu)
                else
                /*  %0011.1010 : OUTL P2,A    */
                    waitForClock()();
                    port2 ! byte(accu)
                fi
            fi
        else
            if irVal = 136 then
            /*  %1000.1000 : ORL BUS, #data  */
                fetchImmediate()();
                port0 ? byte(p0reg);
                port0 ! byte(p0reg logicOR(temp8))
            else
                if irVal = 137 then
                /*  %1000.1001 : ORL P1, #data      */
                    fetchImmediate()();
                    port1 ? byte(p1reg);
                    port1 ! byte(p1reg logicOR(temp8))
                else
                /*  %1000.1010 : ORL P2, #data      */
                    fetchImmediate()();
                    port2 ? byte(p2reg);
                    port2 ! byte(p2reg logicOR(temp8))
                fi
            fi
        fi
    else
        if temp <= 160 then
            if temp = 144 then
                if irVal = 152 then
                /*  %1001.1000 : ANL BUS, #data  */
                    fetchImmediate()();
                    port0 ? byte(p0reg);
                    port0 ! byte(p0reg logicAND(temp8))
                else
                    if irVal = 153 then
                    /*  %1001.1001 : ANL P1, #data      */
                        fetchImmediate()();
                        port1 ? byte(p1reg);
                        port1 ! byte(p1reg logicAND(temp8))
                    else
```

```
                        /*   %1001.1010 : ANL P2, #data      */
                             fetchImmediate()();
                             port2 ? byte(p2reg);
                             port2 ! byte(p2reg logicAND(temp8))
                        fi
                    fi
                else
                /*   %1010.1xxx : MOV R, A      */
                    temp8 setVal(accu getVal());
                    storeRegister()()
                fi
            else
            /*   %1011.1xxx : MOV R, #data*/
                fetchImmediate()();
                storeRegister()()
            fi
        fi
    else
        fetchRegister()();
        if temp <= 80 then
            if temp <= 32 then
                if temp = 16 then
                /*   %0001.1xxx : INC R*/
                    temp8 inc();
                    storeRegister()()
                else
                /*   %0010.1xxx : XCH A,R  */
                    temp9 := new(BoundedInteger) zeroes(1)
                                                 concat(temp8);
                    temp8 setVal(accu getVal());
                    storeRegister()();
                    accu := temp9 fromTo(0,7)
                fi
            else
                if temp = 64 then
                /*   %0100.1xxx : ORL A,R  */
                    accu := accu logicOR(temp8)
                else
                /*   %0101.1xxx : ANL A,R  */
                    accu := accu logicAND(temp8)
                fi
            fi
        else
            if temp <= 224 then
                if temp <= 112 then
                /*   %011x.1xxx : ADD(C) A,R   */
                    addToAccu()()
                else
                /*   %11x0.1xxx : DEC R / DJNZ R   */
                    temp8 dec();
                    storeRegister()();
                    if temp = 224 then
                        if temp8 getVal() = 0 then
                            pclow inc()
                        else
                            jumpInPage()()
                        fi
                    fi
                fi
            else
            /*   %1111.1xxx : MOV A,R */
```

```
                accu setVal(temp8 getVal())
            fi
        fi
    fi.

storeIndirect()()
| address : Integer |
/*  ====================================  */
/*  Method to store an indirectly addressed location.  */
/*  ====================================  */
    address := new(BoundedInteger) copiesOf(2, psw at(4))
        concat(ir fromTo(0,2)) getVal() + 1;
    ram put(ram get(address) + 1, temp8 getVal()).

storeRegister()()
| address : Integer |
/*  ==================================  */
/*  Method to store a directly addressed register.  */
/*  ==================================  */
    address := new(BoundedInteger) copiesOf(2, psw at(4)) concat(ir fromTo(0,2))
            getVal() + 1;
    ram put(address, temp8 getVal()).

timerInstr()()
| temp : Integer |
/*  ======================================  */
/*  Decode & execute (mostly) timer-related instructions.  */
/*  ======================================  */
    temp := irVal & 240;
    if (irVal & 16) = 16 then
    /*  %xxx1.0010 : JBb addr  */
        if (accu at(ir fromTo(5,7) getVal()) getVal()) = 0 then
            waitForClock()();
            pclow inc()
        else
            jumpInPage()()
        fi
    else
        if temp = 0 then
        /*  %0000.0010 : OUTL BUS,A */
            waitForClock()();
            port0 ! byte(accu)
        else
            if temp = 64 then
            /*  %0100.0010 : MOV A,T */
                accu setVal(tmrreg getVal())
            else
            /*  %0110.0010 : MOV T,A */
                tmrreg setVal(accu getVal());
                tcntCmd := true
            fi
        fi
    fi.

updateTimer()()
/*  ==================================  */
/*  Method to perform timer / counter functionality.  */
/*  ==================================  */
    if tcntCmd = false then
        if tctrl at(0) isOne() then
        /*  Timer mode.  */
```

```
            if tctrl getVal() = 63 then
            /*  Overflow of prescaler.   */
                if tmrreg getVal() = 255 then
                /*  Generate overflow interrupt. */
                    tintr setVal(1)
                fi;
                tmrreg inc();
            /*  Reset prescaler.      */
                tctrl setVal(1)
            else
            /*  Increment prescaler.      */
                tctrl setVal(tctrl getVal() + 2)
            fi
        else
            if tctrl at(1) isOne() then
            /*  Counter mode.        */
                if ((tctrl at(2) isOne()) & (t1reg isZero())) then
                    if tmrreg getVal() = 255 then
                    /*  Generate overflow interrupt. */
                        tintr setVal(1)
                    fi;
                    tmrreg inc()
                fi;
                if t1reg isOne() then
                    tctrl setVal(6)
                else
                    tctrl setVal(2)
                fi
            else
            /*  Idle state.          */
                if t1reg isOne() then
                    tctrl setVal(4)
                else
                    tctrl setVal(0)
                fi
            fi
        fi
    else
    /*  Timer / Counter is in command mode. Don't update it      */
    /*  this clock cycle.                                        */
        tcntCmd := false
    fi.

varInstr()()
| temp, address : Integer |
/*  =============================  */
/*  Decode & execute various instructions.  */
/*  =============================  */
    temp := irVal & 240;
    if (irVal & 224) = 128 then
    /*  %100x.0011 : RET /  RETR */
        psw := psw fromTo(3,7) concat(psw fromTo(0,2) dec());
        address := psw at(2) concat(psw at(2) not()) concat(psw
            fromTo(0,1)) concat(new(BoundedInteger) ones(1)) getVal() + 1;
        temp8 setVal(ram get(address));
        waitForClock()();
        pclow := temp8 fromTo(0,2) concat(new(BoundedInteger)
                        init(ram get(address-1), 8));
        pchigh := pchigh at(1) concat(temp8 at(3));
        if ir at(4) getVal() = 1 then
            psw := temp8 fromTo(4,7) concat(psw fromTo(0,3));
            inten := new(BoundedInteger) zeroes(1) concat(inten fromTo(0,1))
        fi
    else
        if (irVal & 160) = 160 then
            fetchAtAccu()();
            if temp = 160 then
            /*  %1010.0011 : MOVP A,@A */
                accu setVal(temp8 getVal())
            else
                if temp = 176 then
                /*  %1011.0011 : JMPP @A     */
                    pclow := pclow fromTo(8,10) concat(temp8)
                else
                /*  %1110.0011 : MOVP3 A,@A     */
                    accu setVal(rom get(temp8 getVal() + 1))
                fi
            fi
        else
            fetchImmediate()();
            if temp <= 64 then
                if temp <= 32 then
                    if temp <= 16 then
                    /*  %000x.0011 : ADD(C) A,#data   */
                        addToAccu()()
                    else
                    /*  %0010.0011 : MOV A,#data */
                        accu setVal(temp8 getVal())
                    fi
                else
                /*  %0100.0011 : ORL A,#data */
                    accu := accu logicOR(temp8)
                fi
            else
                if temp = 80 then
                /*  %0101.0011 : ANL A,#data  */
                    accu := accu logicAND(temp8)
                else
                /*  %1101.0011 : XRL A,#data  */
                    accu := accu logicXOR(temp8)
                fi
            fi
        fi
    fi.

waitForClock()()
/*  ====================================================    */
/*  This method is used to provide a clock which defines a machine cycle.    */
/*  ====================================================    */
    updateTimer()();
    delay(1).
```

```
process class Input_1bit()
/* no superclass */

instance variables
oldVal: BoundedInteger

communication channels
outer, inner

message interface
```

```
inner ! bit(UNKNOWN);
outer ? bit(BoundedInteger)

initial method call
init()()

instance methods
init()()
    oldVal := new(BoundedInteger) init(0,1);
    main()() interrupt(inner ! bit(oldVal)).

main()()
| newVal : BoundedInteger |
    outer ? bit(newVal | newVal getVal() != (oldVal getVal()));
    main()().
```

```
process class IO_8bits()
/* no superclass */

instance variables
oldVal: BoundedInteger

communication channels
outer, inner

message interface
outer ! byte(UNKNOWN);
inner ? byte(BoundedInteger);
inner ! byte(UNKNOWN);
outer ? byte(BoundedInteger)

initial method call
init()()

instance methods
init()()
    oldVal := new(BoundedInteger) init(0,8);
    main()() interrupt(
        sel
            inner ! byte(oldVal)
        or
            outer ! byte(oldVal)
        les).

main()()
| newVal : BoundedInteger |
    sel
        outer ? byte(newVal | newVal getVal() != oldVal getVal())
    or
        inner ? byte(newVal)
    les;
    oldVal := new(BoundedInteger) copiesOf(1, newVal);
    main()().
```

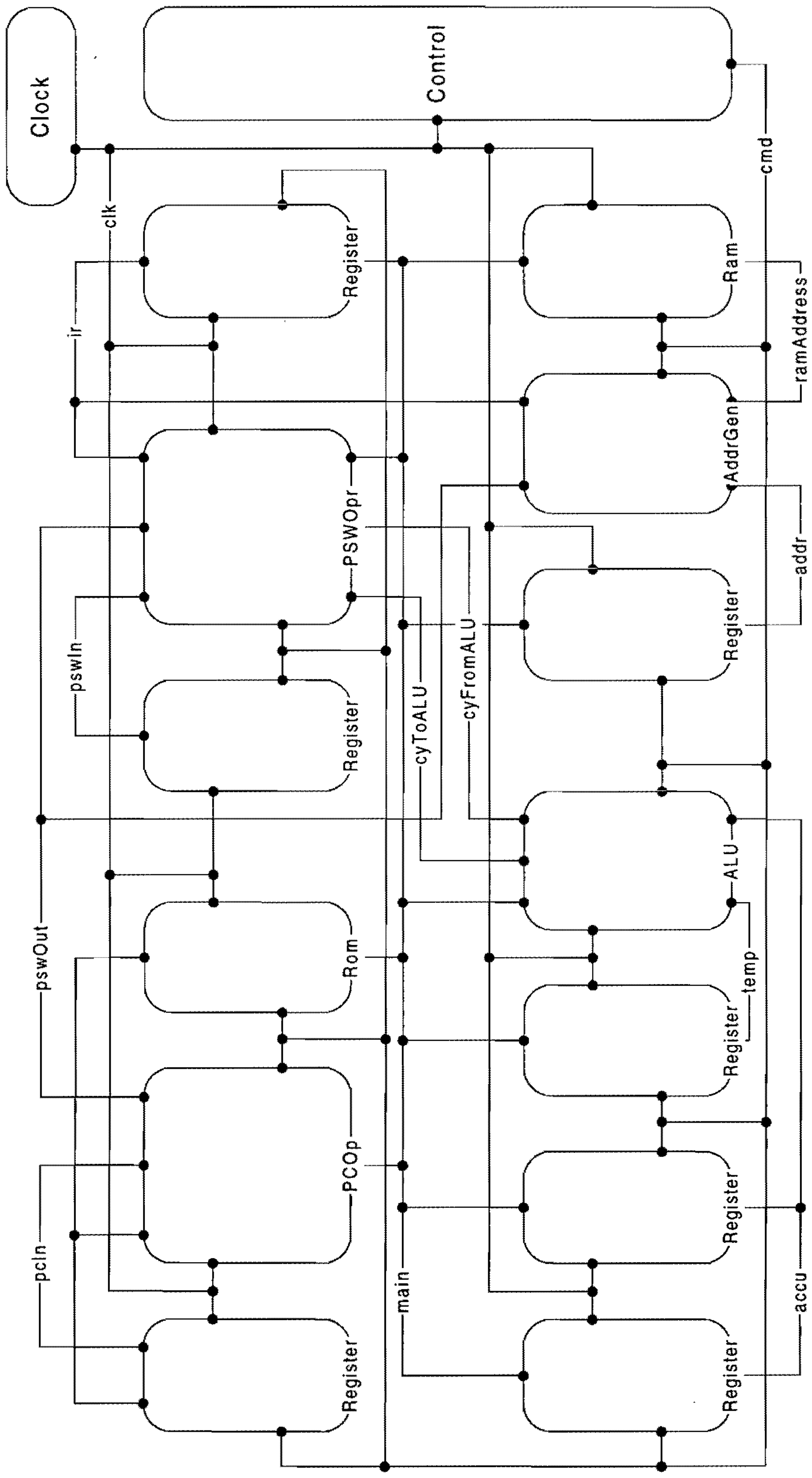
C.5 Cluster class definition

```
cluster class AL8048()

communication channels
port0, port1, port2, test0, test1, int
```

```
message interface
port2 ! byte(UNKNOWN);
port2 ? byte(BoundedInteger);
port0 ? byte(BoundedInteger);
port1 ? byte(BoundedInteger);
int ? bit(BoundedInteger);
test0 ? bit(BoundedInteger);
test1 ? bit(BoundedInteger);
port0 ! byte(UNKNOWN);
port1 ! byte(UNKNOWN)

behaviour specification
(AL8048[t1/test1, p0/port0, t0/test0, p2/port2, i/int, p1/port1] || IO_8bits[port0/outer, p0/inner] ||
IO_8bits[port1/outer, p1/inner] || IO_8bits[port2/outer, p2/inner] || Input_1bit[test0/outer,
t0/inner] || Input_1bit[test1/outer, t1/inner] || Input_1bit[int/outer, i/inner])\{p0, p1, p2, i, t0, t1}
```

Appendix D: RTL 8048 POOSL specification

D.1 Instance Structure Diagram



D.2 RTL 8048 behaviour specification

(Register(0; "out"; 0; "in"; false; "PC"; false; 12; "hold")[pcIn/in, pcOut/out] || Register(0; "out"; -3; "in"; true; "ACCU"; true; 8; "hold")[main/in, accu/out] || PCOp[pswOut/psw, main/bo, main/bi, pcOut/pc, pcIn/out] || Rom[pcOut/pc, main/out] || Register(0; "out"; 0; "in"; false; "PSW"; false; 8; "hold")[pswIn/in, pswOut/out] || PSWOp[cyFromALU/ci, cyToALU/co, main/bo, main/bi, pswOut/psw, pswIn/out] || Register(0; "out"; -3; "in"; false; "IR"; false; 8; "hold")[main/in, ir/out] || Clock[clk/clock] || Register(0; "out"; -3; "in"; true; "LATCH"; true; 8; "hold")[main/in, accu/out] || Register(0; "out"; -3; "in"; false; "TEMP"; false; 8; "hold")[main/in, temp/out] || ALU[cyFromALU/co, main/out, cyToALU/ci] || Register(0; "out"; -3; "in"; false; "ADDR"; false; 8; "hold")[main/in, addr/out] || AddrGen[ramAddress/out, pswOut/psw] || Ram[ramAddress/wa, main/in, main/out, ramAddress/ra] || Control[clk/clock])\{cmd, clk, pcOut, pcIn, pswOut, pswIn, ir, main, cyToALU, cyFromALU, accu, temp, addr, ramAddress}

D.3 Process class definitions

```
process class Clock()
/* no superclass */

instance variables

communication channels
clock

message interface
clock !* beforeClock();
clock !* clockPulse();
clock !* afterClock()

initial method call
init()()

instance methods
init()()
    delay(1);    /*  This delay is to make sure that the first broadcast-   */
                 /*  messages sent by the Output-objects will be            */
                 /*  received by the Input-objects.                         */
    main()().

main()()
    delay(1);
    clock !* beforeClock;
    delay(1);
    clock !* clockPulse;
    delay(1);
    clock !* afterClock;
    main()().
```

```
process class Rom(width: Integer; addressInputName: String; dataOutputName: String; depth:
Integer; name: String)
/* no superclass */

instance variables
outVal: Integer; rom: Array; inVal: Integer

communication channels
cmd, output, input

message interface
input ? value(String, Integer);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
```

```
output ! value(UNKNOWN, UNKNOWN)

initial method call
init()()

instance methods
init()()
| fullOutputName, dest, stateCmd : String |
    initRom()();
    inVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    main()() interrupt(
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState(fullOutputName, "enable")
        else
            cmd ! outputState(fullOutputName, "disable")
        fi ).

main()()
| oldOutVal, newVal : Integer; source : String |
    input ? value(source, newVal | (source=addressInputName) & (newVal != inVal));
    inVal := newVal;
    oldOutVal := outVal;
    if inVal < 0 then
        outVal := -3
    else
        outVal := rom get(inVal + 1)
    fi;
    if outVal != oldOutVal then
        output ! value(dataOutputName, outVal)
    fi;
    main()().

initRom()()
    rom := new(Array) size(depth) putAll(-3);

    rom put(1,35);      /*  MOV   A, 3    */
    rom put(2,3);
    rom put(3,3);       /*  ADD   A, 2    */
    rom put(4,2);
    rom put(5,7);       /*  DEC   A       */
    rom put(6,198);     /*  JZ    0       */
    rom put(7,0);
    rom put(8,4);       /*  JMP   4       */
    rom put(9,4).
```

```
process class Output(TS: Boolean; blockName: String; name: String; defaultDisabled:
Boolean)
/* no superclass */

instance variables
contents: Integer; newValue: Integer; fullOutputName: String; disabled: Boolean

communication channels
toCore, cmd, output, clock

message interface
cmd ? outputState(String, String);
toCore ? value(String, UNKNOWN);
```

```
clock ? afterClock();
output !* value(UNKNOWN)

initial method call
init()()

instance methods
continuous()()
| dest : String |
    toCore ? value(dest, newValue | dest=name);
    if (newValue != contents) then
        output !* value(newValue);
        contents := newValue
    fi;
    continuous()().

init()()
    contents := -3;
    newValue := 0;
    disabled := defaultDisabled;
    fullOutputName := blockName concat(":") concat(name);
    delay(1);
    if TS then
        threeState()()
    else
        continuous()()
    fi.

threeState()()
| dest, stateCmd : String |
    if disabled then
    /*  Output is disabled...     */
        sel
            toCore ? value(dest, newValue | dest=name)
            /*  Output is disabled, so nothing will be done with   */
            /*  the received value (newValue).                      */
        or
            cmd ? outputState(dest, stateCmd | dest=fullOutputName);
            if stateCmd = "enable" then
                disabled := false;
                contents := newValue;
                output !* value(contents)
            else
            /*  Output was already disabled, so nothing changes...  */
                skip
            fi
        or
            clock ? afterClock;
            disabled := defaultDisabled;
            if disabled not() then
                contents := newValue;
                output !* value(contents)
            fi
        les
    else
    /*  Output is enabled...     */
        sel
            toCore ? value(dest, newValue | dest=name);
            if contents != newValue then
                contents := newValue;
                output !* value(contents)
            fi
        or
            cmd ? outputState(dest, stateCmd | dest=fullOutputName);
            if stateCmd = "enable" then
            /*  Output was already enabled, so nothing changes...  */
                skip
            else
            /*  Disable the output...       */
                disabled := true
            fi
        or
            clock ? afterClock;
            disabled := defaultDisabled
        les
    fi;
    threeState()().
```

```
process class Register(width: Integer; cmdResetVal: Integer; systemResetVal: Integer; name:
String; defaultCommand: String; outputName: String)
/* no superclass */

instance variables
defCmd: Integer; command: Integer; fullOutputName: String; contents: Integer; newCmd:
String; cmdVal: Integer; semFlag: Integer

communication channels
clock, cmd, output, input

message interface
cmd ? command(String, UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? test(String, String);
input ? value(String, UNKNOWN);
clock ? afterClock();
cmd ! value(UNKNOWN, UNKNOWN);
output ! value(UNKNOWN, UNKNOWN);
clock ? clockPulse();
clock ? beforeClock()

initial method call
init()()

instance methods
decodeCmd()()
    if newCmd = "setto:" then
        command := 1
    fi;
    if newCmd = "write:" then
        command := 514
    fi;
    if newCmd = "hold" then
        command := 4
    fi;
    if newCmd = "load" then
        command := 520
    fi;
    if newCmd = "loadinc" then
        command := 528
    fi;
    if newCmd = "loaddec" then
```

```
        command := 544
    fi;
    if newCmd = "inc" then
        command := 64
    fi;
    if newCmd = "dec" then
        command := 128
    fi;
    if newCmd = "ressem" then
        command := (command | 256)
    fi;
    if newCmd = "reset" then
        command := (command | 1024)
    fi.

init()()
    contents := systemResetVal;
    fullOutputName := name concat(":") concat(outputName);
    output ! value(outputName, contents);
    newCmd := defaultCommand;
    decodeCmd()();
    defCmd := command;
    semFlag := 0;
    main()().

cmdExecute()()
| source : String |

/*  ==========================================        */
/*  The variable "command" is used in the following way:  */
/*  ----------------------------------------------------- */
/*  bit 0 :   setto: <value>                          */
/*  bit 1 :   write: <value>                          */
/*  bit 2 :   hold                                    */
/*  bit 3 :   load                                    */
/*  bit 4 :   loadinc                                 */
/*  bit 5 :   loaddec                                 */
/*  bit 6 :   inc                                     */
/*  bit 7 :   dec                                     */
/*  bit 8 :   ressem                                  */
/*  bit 9 :   setsem                                  */
/*  bit 10 :  reset                                   */
/*  bit 11 :  multiple                                */
/* ===========================================        */
    if (((command & 512) != 0) & ((command & 1024) = 0)) then
    /*  Set the semaphore flag. */
        semFlag := 1
    else
        if (command & 256) != 0 then
        /*  Reset the semaphore flag.   */
            semFlag := 0
        fi
    fi;
    if (command & 1024) != 0 then
    /*  reset    */
        contents := cmdResetVal
    else
        if (command & 2048) != 0 then
        /*  Multiple, set contents to UNK      */
            contents := -3
        else
```

```
            if (command & %11) != 0 then
            /*   setto: / write:     */
                 contents := cmdVal
            else
                 if (command & %111000) != 0 then
                 /*   load / loadinc / loaddec  */
                      input ? value(source, contents)
                 fi;
                 if (command & %10100000) != 0 then
                 /*   dec / loaddec     */
                      contents:=new(BoundedInteger) init(contents, width)
                           dec() getVal()
                 else
                      if (command & %1010000) != 0 then
                      /*   inc / loadinc */
                           contents:=new(BoundedInteger) init(contents, width)
                                inc() getVal()
                      fi
                 fi
            fi
       fi
  fi.

main()()
| oldContents : Integer; dest, stateCmd, testCmd : String |
  command := 0;
  while true do
       sel
            cmd ? command( dest, newCmd, cmdVal | dest=name);
            if (((command | 256)=256) | (newCmd="reset") | (newCmd="ressem")) then
                 decodeCmd()()
            else
                 command := (command | 2048)       /*   Multiple */
            fi
       or
            cmd ? test(dest, testCmd | dest=name);
            if testCmd = "normal" then
            /*   Normal value test.   */
                 cmd ! value(name, new(BoundedInteger) init(contents, width))
            else
            /*   Auxiliary value test.  */
                 cmd ! value(name, new(BoundedInteger) init(semFlag, 1));
                 if testCmd = "auxTestReset" then
                 /*   Reset the semaphore flag.   */
                      command := (command | 256)        /*   Reset semaphore    */
                 fi
            fi
       or
            cmd ? outputState(dest, stateCmd | dest=name);
            if stateCmd = "enableAll" then
                 cmd ! outputState(fullOutputName, "enable")
            else
                 cmd ! outputState(fullOutputName, "disable")
            fi
       les
  od   abort(clock ? beforeClock);
  /*   First stage passed, new command is now known...   */
  if command = 0 then
  /*   No command received, execute the default command.    */
       command := defCmd
  else
```

```
        if command = 256 then
        /*  Only ressem-cmd received, hold contents...  */
            command := 260
        fi
    fi;
    clock ? clockPulse;
    /*  All inputs are stable now...  */
    oldContents := contents;
    cmdExecute()();
    clock ? afterClock;
    /*  Command executed, ready to send the new value...  */
    if contents != oldContents then
        output ! value(outputName, contents)
    fi;
    main()().
```

```
process class ALU_Operator(name: String; defaultFunction: String)
/* no superclass */

instance variables
ci: BoundedInteger; co: BoundedInteger; temp: BoundedInteger; function: String; accu:
BoundedInteger; out: BoundedInteger

communication channels
cmd, output, input

message interface
input ? value(String, Integer);
cmd ? outputState(String, String);
output ! value(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? command(String, String)

initial method call
init()()

instance methods
init()()
| dest, stateCmd : String |
    accu := new(BoundedInteger) init(-3, 8);
    ci := new(BoundedInteger) init(-3, 1);
    co := new(BoundedInteger) init(-3, 1);
    out := new(BoundedInteger) init(-3, 8);
    temp := new(BoundedInteger) init(-3, 8);
    function := defaultFunction;
    main()().

updateOutput()()
| oldOutVal, oldCoVal : Integer |
    oldOutVal := out getVal();
    oldCoVal := co getVal();
    funcExecute()();
    if out getVal() != oldOutVal then
        output ! value("out", out getVal())
    fi;
    if co getVal() != oldCoVal then
        output ! value("co", co getVal())
    fi.

main()()
| newVal : Integer; newFunc, source, dest, stateCmd : String |
    sel
        input ? value(source, newVal | (source = "accu") & (newVal != accu getVal()));
        accu setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "ci") & (newVal != ci getVal()));
        ci setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "temp") & (newVal != temp getVal()));
        temp setVal(newVal);
        updateOutput()()
    or
        cmd ? command(dest, newFunc | dest=name);
        if newFunc = function then
            main()()
        fi;
        function := newFunc;
        updateOutput()()
    or
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState("ALU:out", "enable")
        else
            cmd ! outputState("ALU:out", "enable")
        fi
    les;
    main()().

funcExecute()()
| sum : BoundedInteger |
    if function = "UNK" then
        /*  Function UNK   */
        out setVal(-3);
        co setVal(-3)
    fi;
    if function = "add" then
        /*  Basic add without carry operation :  */
        sum := new(BoundedInteger) zeroes(1) concat(accu) concat(new(BoundedInteger)
            zeroes(1)) add(new(BoundedInteger) zeroes(1) concat(temp)
            concat(new(BoundedInteger) zeroes(1)));
        out := sum fromTo(1,8);
        co := sum at(9)
    fi;
    if function = "addc" then
        /*  Basic add with carry operation :  */
        sum := new(BoundedInteger) zeroes(1) concat(accu) concat(ci)
            add(new(BoundedInteger) zeroes(1) concat(temp) concat(ci));
        out := sum fromTo(1,8);
        co := sum at(9)
    fi;
    if function = "and" then
        /*  Basic logical AND operation :    */
        out := accu logicAND(temp)
    fi;
    if function = "clr" then
        /*  Basic operation to clear the ACCU :  */
        out := 0
    fi;
    if function = "cpl" then
        /*  Basic logical NOT operation :    */
        out := accu not()
    fi;
    if function = "dec" then
        /*  Basic decrement operation : */
        out := new(BoundedInteger) zeroes(1) concat(accu) concat(new(BoundedInteger)
            zeroes(1)) add(new(BoundedInteger) zeroes(1) concat(new(BoundedInteger)
            ones(8)) concat(new(BoundedInteger) zeroes(1))) fromTo(1,8)
    fi;
    if function = "inc" then
        /*  Basic increment operation :  */
        out := new(BoundedInteger) zeroes(1) concat(accu) concat(new(BoundedInteger)
            ones(1)) add(new(BoundedInteger) zeroes(9) concat(new(BoundedInteger)
            ones(1))) fromTo(1,8)
    fi;
    if function = "or" then
        /*  Basic logical OR operation : */
        out := accu logicOR(temp)
    fi;
    if function = "rl" then
        /*  Rotate the ACCU left :   */
        out := accu rol(1)
    fi;
    if function = "rlc" then
        /*  Rotate ACCU left through carry :*/
        out := accu fromTo(0,6) concat(ci);
        co := accu at(7)
    fi;
    if function = "rr" then
        /*  Rotate ACCU right : */
        out := accu ror(1)
    fi;
    if function = "rrc" then
        /*  Rotate ACCU right through carry :    */
        out := ci concat(accu fromTo(1, 7));
        co := accu at(0)
    fi;
    if function = "swap" then
        /*  Swap the nibbles in the ACCU :  */
        out := accu fromTo(0,3) concat(accu fromTo(4,7))
    fi;
    if function = "xfer" then
        /*  Place accu input on main bus :   */
        out := accu
    fi;
    if function = "xor" then
        /*  Basic logical XOR function : */
        out := temp logicXOR(accu)
    fi.
```

```
process class Input(name: String)
/* no superclass */

instance variables
contents: Integer

communication channels
toCore, input

message interface
toCore ! value(UNKNOWN, UNKNOWN);
input ? value(UNKNOWN)
```

```
initial method call
init()()

instance methods
init()()
    contents := -3;        /*  Set initial contents to UNK.  */
    main()() interrupt( input ? value(contents) ).

main()()
    toCore ! value(name, contents);
    main()().
```

```
process class WritePortInput(name: String; defaultWrite: Boolean)
/* no superclass */

instance variables
write: Boolean; contents: Integer

communication channels
toCore, cmd, clock, input

message interface
cmd ? writePortState(String, String);
clock ? clockPulse();
input ? value(UNKNOWN);
toCore ! value(UNKNOWN, UNKNOWN)

initial method call
init()()

instance methods
init()()
    contents := -3;
    write := defaultWrite;
    main()() interrupt( input ? value(contents) ).

main()()
| dest, stateCmd : String |
    sel
        cmd ? writePortState(dest, stateCmd | dest=name);
        if stateCmd = "write" then
            write := true
        else
            write := false
        fi
    or
        clock ? clockPulse;
        if write then
            toCore ! value(name, contents)
        else
            toCore ! value(name, -3)
        fi;
        write := defaultWrite
    les;
    main()().
```

```
process class Ram(width: Integer; waInputName: String; dataInputName: String;
dataOutputName: String; raInputName: String; defaultContents: Integer; depth: Integer; name:
String)
/* no superclass */
```

```
instance variables
ram: Array; outVal: Integer; raVal: Integer

communication channels
clock, cmd, output, input

message interface
cmd ! writePortState(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? writePortState(String, String);
input ? value(String, Integer);
clock ? clockPulse();
output ! value(UNKNOWN, UNKNOWN)

initial method call
init()()

instance methods
initRam()()
    ram := new(Array) size(depth) putAll(defaultContents).

init()()
| fullOutputName, dest, stateCmd : String |
    initRam()();
    raVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    main()() interrupt(
        sel
            cmd ? outputState(dest, stateCmd | dest=name);
            if stateCmd = "enableAll" then
                cmd ! outputState(fullOutputName, "enable")
            else
                cmd ! outputState(fullOutputName, "disable")
            fi
        or
            cmd ? writePortState(dest, stateCmd | dest=name);
            if stateCmd = "writeAll" then
                cmd ! writePortState(waInputName, "write")
            else
                cmd ! writePortState(waInputName, "noWrite")
            fi
        les).

main()()
| waVal, inVal, newVal, oldOutVal : Integer; source : String |
    sel
        input ? value(source, newVal | (source=raInputName) & (newVal != raVal));
        raVal := newVal;
        oldOutVal := outVal;
        if raVal < 0 then
            outVal := -3
        else
            outVal := ram get(raVal + 1)
        fi;
        if outVal != oldOutVal then
            output ! value(dataOutputName, outVal)
        fi
    or
```

```
        clock ? clockPulse;
        input ? value(source, waVal | source=waInputName);
        if waVal >= 0 then
            input ? value(source, inVal | source=dataInputName);
            ram put(waVal + 1, inVal);
            if ((raVal = waVal) & (outVal != inVal)) then
                outVal := inVal;
                output ! value(dataOutputName, outVal)
            fi
        fi
    les;
    main()().
```

```
process class AddrGen_Operator(name: String; defaultFunction: String)
/* no superclass */

instance variables
addr: BoundedInteger; ir: BoundedInteger; psw: BoundedInteger; out: BoundedInteger;
function: String

communication channels
cmd, output, input

message interface
input ? value(String, Integer);
output ! value(UNKNOWN, UNKNOWN);
cmd ? command(String, String)

initial method call
init()()

instance methods
init()()
    addr := new(BoundedInteger) init(-3, 8);
    ir := new(BoundedInteger) init(-3, 8);
    psw := new(BoundedInteger) init(-3, 8);
    out := new(BoundedInteger) init(-3, 6);
    function := defaultFunction;
    main()().

updateOutput()()
| oldOutVal : Integer |
    oldOutVal := out getVal();
    funcExecute()();
    if out getVal() != oldOutVal then
        output ! value("out", out getVal())
    fi.

main()()
| newVal : Integer; newFunc, source, dest : String |
    sel
        input ? value(source, newVal | (source="addr") & (newVal != addr getVal()));
        addr setVal(newVal)
    or
        input ? value(source, newVal | (source="ir") & (newVal != ir getVal()));
        ir setVal(newVal)
    or
        input ? value(source, newVal | (source="psw") & (newVal != psw getVal()));
        psw setVal(newVal)
    or
        cmd ? command(dest, newFunc | dest=name);
        if newFunc = function then
            main()()
        fi;
        function := newFunc
    les;
    updateOutput()();
    main()().

funcExecute()()
| bank : BoundedInteger |
    if function = "UNK" then
        /*  Function UNK  */
        out setVal(-3)
    fi;
    if function = "addr" then
        /*  Generate address out of ADDR register :     */
        out := addr fromTo(0,5)
    fi;
    if function = "reg" then
        /*  Address a register using the bank select bit in    */
        /*  the PSW and the low order IR bits :            */
        bank := psw at(4);
        out := new(BoundedInteger) zeroes(1) concat(bank) concat(bank)
            concat(ir fromTo(0,2))
    fi;
    if function = "stack" then
        /*  Generate the address for the low stack        */
        /*  byte out of PSW bits 0..2 :                 */
        out := new(BoundedInteger) zeroes(1) concat(psw at(2)) concat(psw at(2) not())
            concat(psw fromTo(0,1)) concat(new(BoundedInteger) zeroes(1))
    fi;
    if function = "stinc" then
        /*  Generate an address for the high stack        */
        /*  byte out of PSW bits 0..2 :                 */
        out := new(BoundedInteger) zeroes(1) concat(psw at(2)) concat(psw at(2) not())
            concat(psw fromTo(0,1)) concat(new(BoundedInteger) ones(1))
    fi.
```

```
process class PSW_Operator(name: String; defaultFunction: String)
/* no superclass */

instance variables
co: BoundedInteger; bo: BoundedInteger; function: String; ir: BoundedInteger; psw:
BoundedInteger; bi: BoundedInteger; out: BoundedInteger; ci: BoundedInteger

communication channels
cmd, output, input

message interface
input ? value(String, Integer);
cmd ? outputState(String, String);
output ! value(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? command(String, String)

initial method call
init()()

instance methods
init()()
| dest, stateCmd : String |
    bi := new(BoundedInteger) init(-3, 8);     /*  Input from main bus.               */
    bo := new(BoundedInteger) init(-3, 8);     /*  Output to main bus.                */
    ci := new(BoundedInteger) init(-3, 1);     /*  Carry flag input from ALU.         */
    co := new(BoundedInteger) init(-3, 1);     /*  Carry flag output to ALU.          */
    ir := new(BoundedInteger) init(-3, 8);     /*  Input from instruction register.   */
    out := new(BoundedInteger) init(-3, 8);    /*  Output to program status register. */
    psw := new(BoundedInteger) init(-3, 8);    /*  Input from program status register. */
    function := defaultFunction;
    main()().

updateOutput()()
| oldOutVal, oldCoVal, oldBoVal : Integer |
    oldOutVal := out getVal();
    oldCoVal := co getVal();
    oldBoVal := bo getVal();
    funcExecute()();
    if out getVal() != oldOutVal then
        output ! value("out", out getVal())
    fi;
    if co getVal() != oldCoVal then
        output ! value("co", co getVal())
    fi;
    if bo getVal() != oldBoVal then
        output ! value("bo", bo getVal())
    fi.

main()()
| newVal : Integer; newFunc, source, dest, stateCmd : String |
    sel
        input ? value(source, newVal | (source = "bi") & (newVal != bi getVal()));
        bi setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "ci") & (newVal != ci getVal()));
        ci setVal(newVal);
        updateOutput()();
    or
        input ? value(source, newVal | (source = "ir") & (newVal != ir getVal()));
        ir setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "psw") & (newVal != psw getVal()));
        psw setVal(newVal);
        updateOutput()()
    or
        cmd ? command(dest, newFunc | dest=name);
        if newFunc = function then
            main()()
        fi;
        function := newFunc;
        updateOutput()()
    or
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState("PSWOP:bo", "enable")
        else
            cmd ! outputState("PSWOP:bo", "disable")
        fi
    les;
    main()().
```

```
funcExecute()()
    if function = "UNK" then
        /*  Function UNK   */
        out setVal(-3);
        co setVal(-3);
        bo setVal(-3)
    fi;
    if function = "bank" then
        /*  Update the register bank select bit in the PSW    */
        /*  with the instruction bit from the IR :            */
        out := psw fromTo(5, 7) concat(ir at(4)) concat(psw fromTo(0, 3))
    fi;
    if function = "carry" then
        /*  Change carry bit in the PSW while outputting      */
        /*  the current carry bit value :                     */
        out := ci concat(psw fromTo(0,6));
        co := psw at(7)
    fi;
    if function = "clrcy" then
        /*  Clear the carry flag (bit 7) :  */
        out := new(BoundedInteger) zeroes(1) concat(psw fromTo(0,6))
    fi;
    if function = "clrf0" then
        /*  Clear the F0 flag (bit 5) :*/
        out := psw fromTo(6,7) concat(new(BoundedInteger) zeroes(1))
            concat(psw fromTo(0,4))
    fi;
    if function = "clrf1" then
        /*  Clear the F1 flag (bit 6) :*/
        out := psw at(7) concat(new(BoundedInteger) zeroes(1)) concat(psw fromTo(0,5))
    fi;
    if function = "cplcy" then
        /*  Complement the carry flag : */
        out := psw at(7) not() concat(psw fromTo(0,6))
    fi;
    if function = "cplf0" then
        /*  Complement the F0 flag :        */
        out := psw fromTo(6,7) concat(psw at(5) not()) concat(psw fromTo(0,4))
    fi;
    if function = "cplf1" then
        /*  Complement the F1 flag :        */
        out := psw at(7) concat(psw at(6) not()) concat(psw fromTo(0,5))
    fi;
    if function = "load" then
        /*  Write value on main bus into PSW :      */
        out := bi
    fi;
    if function = "page" then
        /*  Update the program page select bit in   */
        /*  the PSW with the instruction bit from   */
        /*  the IR (uses bit 3 of PSW, always 1     */
        /*  for Intel) :                            */
        out := psw fromTo(4,7) concat(ir at(4)) concat(psw fromTo(0,2))
    fi;
    if function = "pchi" then
        /*  Output the highest bits of the IR, concatenated  */
        /*  with the 'PAGE' bit in PSW (unused bit #3) onto  */
        /*  the main bus (used for CALL / JMP) :             */
        bo := new(BoundedInteger) zeroes(4) concat(psw at(3)) concat(ir fromTo(5,7))
    fi;
    if function = "psw" then
```

```
        /*  Output the PSW onto the main bus : */
        bo := psw
    fi;
    if function = "rest" then
        /*  Restore the 4 MSB's of the PSW during  */
        /*  a RET instruction :                    */
        out := bi fromTo(4,7) concat(psw fromTo(0,3))
    fi;
    if function = "stdec" then
        /*  Decrement the stackpointer bits in the    */
        /*  PSW by 1 :                                */
        out := psw fromTo(3,7) concat(psw fromTo(0,2) dec())
    fi;
    if function = "stinc" then
        /*  Increment the stackpointer bits in the    */
        /*  PSW by 1 :                                */
        out := psw fromTo(3,7) concat(psw fromTo(0,2) inc())
    fi.
```

```
process class PC_Operator(name: String; defaultFunction: String)
/* no superclass */

instance variables
pc: BoundedInteger; bo: BoundedInteger; function: String; bi: BoundedInteger; psw:
BoundedInteger; out: BoundedInteger

communication channels
cmd, output, input

message interface
input ? value(String, Integer);
cmd ? outputState(String, String);
output ! value(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? command(String, String)

initial method call
init()()

instance methods
init()()
    pc := new(BoundedInteger) init(-3, 12);
    bi := new(BoundedInteger) init(-3, 8);
    psw := new(BoundedInteger) init(-3, 8);
    out := new(BoundedInteger) init(-3, 12);
    bo := new(BoundedInteger) init(-3, 8);
    function := defaultFunction;
    main()().

updateOutput()()
| oldOutVal, oldBoVal : Integer |
    oldOutVal := out getVal();
    oldBoVal := bo getVal();
    funcExecute()();
    if out getVal() != oldOutVal then
        output ! value("out", out getVal())
    fi;
    if bo getVal() != oldBoVal then
        output ! value("bo", bo getVal())
    fi.
```

```
main()()
| newVal : Integer; newFunc, source, dest, stateCmd : String |
    sel
        input ? value(source, newVal | (source = "pc") & (newVal != pc getVal()));
        pc setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "bi") & (newVal != bi getVal()));
        bi setVal(newVal);
        updateOutput()()
    or
        input ? value(source, newVal | (source = "psw") & (newVal != psw getVal()));
        psw setVal(newVal);
        updateOutput()()
    or
        cmd ? command(dest, newFunc | dest=name);
        if newFunc = function then
            main()()
        fi;
        function := newFunc;
        updateOutput()()
    or
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState("PCOP:bo", "enable")
        else
            cmd ! outputState("PCOP:bo", "disable")
        fi
    les;
    main()().

funcExecute()()
| pcInc : BoundedInteger |
    if function = "UNK" then
        /*  Function UNK  */
        out setVal(-3);
        bo setVal(-3)
    fi;
    if function = "inc" then
        /*  Increment the PC, but only the lowest 11 bits :   */
        out := pc at(11) concat(pc fromTo(0,10) inc())
    fi;
    if function = "loadhi" then
        /*  Change most significant bits in the PC :  */
        out := bi fromTo(0,3) concat(pc fromTo(0,7))
    fi;
    if function = "loadlo" then
        /*  Change the least significant bits of the PC :  */
        out := pc fromTo(8, 11) concat(bi)
    fi;
    if function = "readhi" then
        /*  Output the most sigificant bits of the (incremented)   */
        /*  PC concatenated with the most significant bits of the  */
        /*  PSW to the main bus :                                  */
        pcInc := pc at(11) concat(pc fromTo(0,10) inc());
        bo := psw fromTo(4,7) concat(pcInc fromTo(8,11))
    fi;
    if function = "readlo" then
        /*  Output the least significant bits of the (incremented)  */
        /*  PC to the main bus :                                    */
        pcInc := pc at(11) concat(pc fromTo(0,10) inc());
        bo := pcInc fromTo(0,7)
    fi.
```

```
process class Control()
/* no superclass */

instance variables
ir: BoundedInteger; latch: BoundedInteger; psw: BoundedInteger; irVal: Integer; accu:
BoundedInteger

communication channels
cmd, clock

message interface
cmd ! command(UNKNOWN, UNKNOWN, UNKNOWN);
cmd ! command(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ! writePortState(UNKNOWN, UNKNOWN);
clock ? afterClock();
cmd ? value(String, UNKNOWN);
cmd ! test(UNKNOWN, UNKNOWN)

initial method call
init()()

instance methods
exec2()()
| source : String |
    clock ? afterClock;
    if (irVal & %11111) = %10100 then
    /*  CALL :  */
        cmd ! command("PCOP","readhi");
        cmd ! outputState("PCOP","enableAll");
        cmd ! command("ADDRGEN","stinc");
        cmd ! writePortState("RAM","writeAll");
        cmd ! command("PSWOP","stinc");
        cmd ! command("PSW","load",-3);
        exec3()()
    fi;
    if (irVal & %11101111) = %10000011 then
    /*  RET / RETR :   */
        cmd ! command("ADDRGEN","stack");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("PCOP","loadlo");
        cmd ! command("PC","load",-3);
        exec3()()
    fi;
    if (irVal & %11111110) = %00100000 then
    /*  XCH A, @Rx :        */
        cmd ! outputState("ACCU","enableAll");
        cmd ! command("ALU","xfer");
        cmd ! outputState("ALU","enableAll");
        cmd ! command("LATCH","load",-3);
        exec3()()
    fi;
    if (irVal & %11111110) = %00010000 then
    /*  INC @Rx :  */
        cmd ! command("ADDRGEN","addr");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("LATCH","load",-3);
        exec4()()
    fi;
    if (irVal & %01001110) = %01000000 then
    /*  Indirect ORL, ANL, ADD, ADDC, XRL operations :     */
        cmd ! command("ADDRGEN","addr");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("TEMP","load",-3);
        exec4()()
    else
    /*  DJNZ Rx :  */
        cmd ! outputState("LATCH","enableAll");
        cmd ! command("ALU","dec");
        cmd ! outputState("ALU","enabelAll");
        cmd ! command("ADDRGEN","reg");
        cmd ! writePortState("RAM","writeAll");
        cmd ! test("LATCH","normal");
        cmd ? value(source, latch | source="LATCH");
        if latch isOne() then
        /*  Decremented to 0 :  */
            cmd ! command("PCOP","inc");
            cmd ! command("PC","load",-3);
            fetch()()
        else
        /*  Not decremented to 0 :  */
            exec3()()
        fi
    fi.

init()()
/*  Fetch instruction and increment PC: */
    cmd ! outputState("ROM","enableAll");
    cmd ! command("IR","load",-3);
    cmd ! command("PCOP","inc");
    cmd ! command("PC","load",-3);
    exec1()().

exec3()()
/*  Third decode / execute state:     */
    clock ? afterClock;
    if (irVal & %11111) = %10100 then
    /*  CALL :  */
        cmd ! outputState("ROM","enableAll");
        cmd ! command("PCOP","loadlo");
        cmd ! command("PC","load",-3);
        exec4()()
    fi;
    if irVal = %10010011 then
    /*  RETR : */
        cmd ! command("PSWOP","rest");
        cmd ! command("PSW","load",-3)
    fi;
    if (irVal & %11101111) = %10000011 then
    /*  RET & RETR :  */
        cmd ! command("ADDRGEN","stinc");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("PCOP","loadhi");
        cmd ! command("PC","load",-3);
        fetch()()
    fi;
    if (irVal & %11111000) = %11101000 then
    /*  DJNZ Rx :  */
        cmd ! outputState("ROM","enableAll");
        cmd ! command("PCOP","loadlo");
        cmd ! command("PC","load",-3);
        fetch()()
    fi;
    if (irVal & %11111110) = %00100000 then
    /*  XCH A,@Rx :  */
        cmd ! command("ADDRGEN","addr")
    else
    /*  XCH A,Rx : */
        cmd ! command("ADDRGEN","reg")
    fi;
    cmd ! outputState("RAM","enableAll");
    cmd ! command("ACCU","load",-3);
    exec4()().

exec1()()
| jumpCondition : Boolean; temp1, temp2 : Integer; source : String |
    clock ? afterClock;
    cmd ! test("IR","normal");
    cmd ? value(source, ir | source="IR");
    irVal := ir getVal();
    temp1 := (irVal & %00001111);
    temp2 := (irVal & %11110000);

    if (irVal & %1110) = 0 then
    /*  Indirect operations :       */
        cmd ! command("ADDRGEN","reg");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("ADDR","load",-3);
        if (irVal & 10100000) = %10100000 then
        /*  Indirect MOV operations :    */
            exec4()()
        else
        /*  Indirect INC, XCH, ORL, ANL, ADD, ADDC, XRL operations :       */
            exec2()()
        fi
    fi;

    if temp1 = %0010 then
    /*  JBx :    */
        cmd ! test("ACCU","normal");
        cmd ? value(source, accu | source="ACCU");
        if accu at(ir fromTo(5,7) getVal()) isOne() then
            cmd ! outputState("ROM","enableAll");
            cmd ! command("PCOP","loadlo")
        else
            cmd ! command("PCOP","inc")
        fi;
        cmd ! command("PC","load",-3);
        fetch()()
    fi;

    if temp1= %0011 then
        if (irVal & %11101111) = %10000011 then
        /*  RET & RETR :  */
            cmd ! command("PSWOP","stdec");
            cmd ! command("PSW","load",-3);
            exec2()()
        fi;
        if irVal = %10110011 then
        /*  JMPP @A : */
```

```
            cmd ! outputState("ACCU","enableAll");
            cmd ! command("ALU","xfer");
            cmd ! outputState("ALU","enableAll");
            cmd ! command("PCOP","loadlo");
            cmd ! command("PC","load",-3);
            fetch()()
        fi;
        if irVal = %00100011 then
        /*  MOV A, #data   */
            cmd ! outputState("ROM","enableAll");
            cmd ! command("ACCU","load",-3);
            cmd ! command("PCOP","inc");
            cmd ! command("PC","load",-3);
            fetch()()
        else
        /*  Immediate ops into accu :   */
            cmd ! outputState("ROM","enableAll");
            cmd ! command("TEMP","load",-3);
            cmd ! command("PCOP","inc");
            cmd ! command("PC","load",-3);
            exec4()()
        fi
    fi;

    if temp1 = %0100 then
        if (irVal & %11111) = %100 then
        /*  JMP addr :  */
            cmd ! outputState("ROM","enableAll");
            cmd ! command("PCOP","loadlo");
            cmd ! command("PC","load",-3);
            exec4()()
        else
        /*  CALL addr :*/
            cmd ! command("PCOP","readlo");
            cmd ! outputState("PCOP","enableAll");
            cmd ! command("ADDRGEN","stack");
            cmd ! writePortState("RAM","writeAll");
            exec2()()
        fi
    fi;

    if temp1 = %0101 then
        if temp2 = %10000000 then
        /*  CLR F0 :    */
            cmd ! command("PSWOP","clrf0")
        fi;
        if temp2 = %10010000 then
        /*  CPL F0 :    */
            cmd ! command("PSWOP","cplf0")
        fi;
        if temp2 = %10100000 then
        /*  CLR F1 :    */
            cmd ! command("PSWOP","clrf1")
        fi;
        if temp2 = %10110000 then
        /*  CPL F1 :    */
            cmd ! command("PSWOP","cplf1")
        fi;
        if (irVal & %11101111) = %11000101 then
        /*  SEL RBn :  */
            cmd ! command("PSWOP","bank")
        fi;
        if (irVal & %11101111) = %11100101 then
        /*   SEL MBn :  */
             cmd ! command("PSWOP","page")
        fi;
        cmd ! command("PSW","load",-3);
        cmd ! outputState("ROM","enableAll");
        cmd ! command("IR","load",-3);
        cmd ! command("PCOP","inc");
        cmd ! command("PC","load",-3);
        exec1()()
   fi;

   if temp1 = %0110 then
        cmd ! test("ACCU","normal");
        cmd ? value(source, accu | source="ACCU");
        cmd ! test("PSW","normal");
        cmd ? value(source, psw | source="PSW");
        jumpCondition :=    ((irVal = %10110110) & (psw at(5) isOne()))       /* JF0 : */
                        |   ((irVal = %01110110) & (psw at(6) isOne()))       /* JF1 : */
                        |   ((irVal = %11000110) & (accu isZero()))           /* JZ :  */
                        |   ((irVal = %10010110) & (accu isZero() not()))     /* JNZ : */
                        |   ((irVal = %11110110) & (psw at(7) isOne()))       /* JC :  */
                        |   ((irVal = %11100110) & (psw at(7) isZero()));     /* JNC : */
        if jumpCondition then
             cmd ! outputState("ROM","enableAll");
             cmd ! command("PCOP","loadlo")
        else
             cmd ! command("PCOP","inc")
        fi;
        cmd ! command("PC","load",-3);
        fetch()()
   fi;

   if temp1 = %0111 then
        if temp2 = %11000000 then
        /*   MOV A, PSW :       */
             cmd ! command("PSWOP","psw");
             cmd ! outputState("PSWOP","enableAll");
             cmd ! command("ACCU","load",-3);
             fetch()()
        fi;
        if temp2 = %11010000 then
        /*   MOV PSW, A :       */
             cmd ! outputState("ACCU","enableAll");
             cmd ! command("ALU","xfer");
             cmd ! outputState("ALU","enableAll");
             cmd ! command("PSWOP","load");
             cmd ! command("PSW","load",-3);
             fetch()()
        fi;
        if temp2 = %10010000 then
        /*   CLR C :*/
             cmd ! command("PSWOP","clrcy");
             cmd ! command("PSW","load",-3);
             cmd ! outputState("ROM","enableAll");
             cmd ! command("IR","load",-3);
             cmd ! command("PCOP","inc");
             cmd ! command("PC","load",-3);
             exec1()()
        fi;
        if temp2 = %10100000 then
        /*  CPL C :*/
            cmd ! command("PSWOP","cplcy");
            cmd ! command("PSW","load",-3);
            cmd ! outputState("ROM","enableAll");
            cmd ! command("IR","load",-3);
            cmd ! command("PCOP","inc");
            cmd ! command("PC","load",-3);
            exec1()()
        fi;
        if temp2 = 0 then
        /*  DEC A :     */
            cmd ! command("ALU","dec")
        fi;
        if temp2 = %00010000 then
        /*  INC A : */
            cmd ! command("ALU","inc")
        fi;
        if temp2 = %00100000 then
        /*  CLR A :*/
            cmd ! command("ALU","clr")
        fi;
        if temp2 = %00110000 then
        /*  CPL A :*/
            cmd ! command("ALU","cpl")
        fi;
        if temp2 = %01000000 then
        /*  SWAP A :  */
            cmd ! command("ALU","swap")
        fi;
        if temp2 = %01110000 then
        /*  RR A :  */
            cmd ! command("ALU","rr")
        fi;
        if temp2 = %11100000 then
        /*  RL A :  */
            cmd ! command("ALU","rl")
        fi;
        if temp2 = %01100000 then
        /*  RRC A :     */
            cmd ! command("ALU","rrc");
            cmd ! command("PSWOP","carry");
            cmd ! command("PSW","load",-3)
        fi;
        if temp2 = %11110000 then
        /*  RLC A :*/
            cmd ! command("ALU","rlc");
            cmd ! command("PSWOP","carry");
            cmd ! command("PSW","load",-3)
        fi;
        /*  Unary accu-to-accu ops :    */
        cmd ! outputState("ACCU","enableAll");
        cmd ! outputState("ALU","enableAll");
        cmd ! command("ACCU","load",-3);
        fetch()()
    fi;

    if (irVal & %00001000) = %1000 then
        if (((irVal & %11001000) = %01001000) | (temp2 = %11010000)) then
        /*  ORL / ANL / ADD / ADDC / XRL  A, Rx : */
            cmd ! command("ADDRGEN","reg");
            cmd ! outputState("RAM","enableAll");
            cmd ! command("TEMP","load",-3);
            exec4()()
        fi;
        if temp2 = %00100000 then
        /*  XCH A, Rx :*/
            cmd ! outputState("ACCU","enableAll");
            cmd ! command("ALU","xfer");
            cmd ! outputState("ALU","enableAll");
            cmd ! command("LATCH","load",-3);
            exec3()()
        fi;
        if ((temp2 = %00010000) | ((irVal & %11011000) = %11001000)) then
        /*  INC / DEC / DJNZ Rx operations :   */
            cmd ! command("ADDRGEN","reg");
            cmd ! outputState("RAM","enableAll");
            cmd ! command("LATCH","load",-3);
            if temp2 = %11100000 then
            /*  DJNZ Rx :  */
                exec2()()
            else
            /*  INC / DEC Rx operations :   */
                exec4()()
            fi
        fi;
        if temp2 = %10100000 then
        /*  MOV Rx, A :     */
            cmd ! outputState("ACCU","enableAll");
            cmd ! command("ALU","xfer");
            cmd ! outputState("ALU","enableAll");
            cmd ! command("ADDRGEN","reg");
            cmd ! writePortState("RAM","writeAll");
            fetch()()
        fi;
        if temp2 = %11110000 then
        /*  MOV A, Rx :     */
            cmd ! command("ADDRGEN","reg");
            cmd ! outputState("RAM","enableAll");
            cmd ! command("ACCU","load",-3);
            fetch()()
        fi;
        if temp2 = %10110000 then
        /*  MOV Rx, #data :     */
            cmd ! outputState("ROM","enableAll");
            cmd ! command("ADDRGEN","reg");
            cmd ! writePortState("RAM","writeAll");
            cmd ! command("PCOP","inc");
            cmd ! command("PC","load",-3);
            fetch()()
        fi
    fi.

fetch()()
/*  Fetch instruction and increment PC :*/
    clock ? afterClock;
    cmd ! outputState("ROM","enableAll");
    cmd ! command("IR","load",-3);
    cmd ! command("PCOP","inc");
    cmd ! command("PC","load",-3);
    exec1()().
```

```
exec4()()
| tempVal : Integer |
    clock ? afterClock;

    if (irVal & %1111) = %100 then
    /*  JMP & CALL :  */
        cmd ! command("PSWOP","pchi");
        cmd ! outputState("PSWOP","enableAll");
        cmd ! command("PCOP","loadhi");
        cmd ! command("PC","load",-3);
        fetch()()
    fi;

    /*  Operations loading accu :   */

    if (((irVal & %10101111) = %00000011) | ((irVal & %11000000) = %01000000) |
        ((irVal & %11110000) = %11010000)) then
        tempVal := irVal & %11110000;
        if ((tempVal = 0) | (tempVal = %01100000)) then
        /*  ADD operations :    */
            cmd ! command("ALU","add");
            cmd ! command("PSWOP","carry");
            cmd ! command("PSW","load",-3)
        fi;
        if ((tempVal = %10000) | (tempVal = %01110000)) then
        /*  ADDC operations :  */
            cmd ! command("ALU","addc");
            cmd ! command("PSWOP","carry");
            cmd ! command("PSW","load",-3)
        fi;
        if tempVal = %01000000 then
        /*  ORL operations :    */
            cmd ! command("ALU","or")
        fi;
        if tempVal = %01010000 then
        /*  ANL operations :    */
            cmd ! command("ALU","and")
        fi;
        if tempVal = %11010000 then
        /*  XRL operations :    */
            cmd ! command("ALU","xor")
        fi;
        cmd ! outputState("ACCU","enableAll");
        cmd ! outputState("ALU","enableAll");
        cmd ! command("ACCU","load",-3);
        fetch()()
    fi;

    if (irVal & %11111110) = %11110000 then
    /*  MOV A, @Rx :       */
        cmd ! command("ADDRGEN","addr");
        cmd ! outputState("RAM","enableAll");
        cmd ! command("ACCU","load",-3);
        fetch()()
    fi;

    /*  Operations writing into registers :     */

    if (irVal & %11111000) = %11001000 then
    /*  DEC Rx :  */
        cmd ! outputState("LATCH","enableAll");
        cmd ! command("ADDRGEN","reg");
        cmd ! command("ALU","dec");
        cmd ! outputState("ALU","enableAll");
        cmd ! writePortState("RAM","writeAll");
        fetch()()
    fi;
    if (irVal & %11000000) = 0 then
    /*  Data coming from latch :      */
        cmd ! outputState("LATCH","enableAll")
    fi;
    if (irVal & %11111110) = %10100000 then
    /*  Data coming from accu. MOV @Rx, A :        */
        cmd ! outputState("ACCU","enableAll")
    fi;
    if (irVal & %11111110) = %10110000 then
    /*  Data coming from rom.  MOV @Rx, #data      */
        cmd ! outputState("ROM","enableAll");
        cmd ! command("PCOP","inc");
        cmd ! command("PC","load",-3);
        cmd ! command("ADDRGEN","addr");
        cmd ! writePortState("RAM","writeAll");
        fetch()()
    fi;
    if (irVal & %01001110) = 0 then
    /*  Using indirect addressing :  */
        cmd ! command("ADDRGEN","addr")
    else
    /*  Using direct addressing :     */
        cmd ! command("ADDRGEN","reg")
    fi;
    if (irVal & %11110000) = %00010000 then
    /*  INC @Rx, INC Rx operations :   */
        cmd ! command("ALU","inc")
    else
    /*  XCH A,@Rx,  XCH A,Rx,  MOV @Rx, A */
        cmd ! command("ALU","xfer")
    fi;
    cmd ! outputState("ALU","enableAll");
    /*  All operations writing ram :  */
    cmd ! writePortState("RAM","writeAll");
    fetch()().
```

D.4 Cluster class definitions

```
cluster class AddrGen()

communication channels
addr, ir, psw, out, cmd

message interface
psw ? value(UNKNOWN);
addr ? value(UNKNOWN);
out !* value(UNKNOWN);
ir ? value(UNKNOWN);
cmd ? command(String, String)

behaviour specification
(AddrGen_Operator("ADDRGEN"; "UNK")[coreToOut/output, inToCore/input] || Output(false;
"ADDRGEN"; "out"; false)\{cmd, clock}[coreToOut/toCore, out/output] ||
Input("addr")[inToCore/toCore, addr/input] || Input("ir")[inToCore/toCore, ir/input] ||
Input("psw")[inToCore/toCore, psw/input])\{inToCore, coreToOut}
```

```
cluster class Ram()

communication channels
ra, in, wa, cmd, clk, out

message interface
in ? value(UNKNOWN);
ra ? value(UNKNOWN);
cmd ! writePortState(UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
clk ? afterClock();
wa ? value(UNKNOWN);
clk ? clockPulse();
cmd ? writePortState(String, String);
out !* value(UNKNOWN)

behaviour specification
(Ram(8; "wa"; "in"; "out"; "ra"; -3; 64; "RAM")[clk/clock, coreToOut/output, inToCore/input] ||
Input("in")[inToCore/toCore, in/input] || WritePortInput("wa"; true)[inToCore/toCore, clk/clock,
wa/input] || Output(true; "RAM"; "out"; true)[clk/clock, coreToOut/toCore, out/output] ||
Input("ra")[inToCore/toCore, ra/input])\{inToCore, coreToOut}
```

```
cluster class PCOp()

communication channels
pc, psw, bi, out, bo, cmd, clk

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
clk ? afterClock();
cmd ? outputState(String, String);
cmd ? command(String, String);
pc ? value(UNKNOWN);
psw ? value(UNKNOWN);
bi ? value(UNKNOWN);
bo !* value(UNKNOWN);
out !* value(UNKNOWN)

behaviour specification
(PC_Operator("PCOP"; "UNK")[coreToOut/output, inToCore/input] || Output(false; "PCOP";
"out"; false)\{cmd, clock}[coreToOut/toCore, out/output] || Output(true; "PCOP"; "bo";
true)[clk/clock, coreToOut/toCore, bo/output] || Input("pc")[inToCore/toCore, pc/input] ||
Input("psw")[inToCore/toCore, psw/input] || Input("bi")[inToCore/toCore, bi/input])\{coreToOut,
inToCore}
```

```
cluster class PSWOp()

communication channels
psw, ir, ci, bi, cmd, clk, out, co, bo

message interface
ir ? value(UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? command(String, String);
clk ? afterClock();
psw ? value(UNKNOWN);
ci ? value(UNKNOWN);
co !* value(UNKNOWN);
```

bi ? value(UNKNOWN);
bo !* value(UNKNOWN);
out !* value(UNKNOWN)

behaviour specification
(PSW_Operator("PSWOP"; "UNK")[coreToOut/output, inToCore/input] || Output(false; "PSWOP"; "out"; false)\{cmd, clock}[coreToOut/toCore, out/output] || Output(false; "PSWOP"; "co"; false)\{cmd, clock}[coreToOut/toCore, co/output] || Output(true; "PSWOP"; "bo"; true)[clk/clock, coreToOut/toCore, bo/output] || Input("psw")[inToCore/toCore, psw/input] || Input("ir")[inToCore/toCore, ir/input] || Input("ci")[inToCore/toCore, ci/input] || Input("bi")[inToCore/toCore, bi/input])\{coreToOut, inToCore}

---

cluster class ALU()

communication channels
ci, temp, accu, co, out, cmd, clk

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
clk ? afterClock();
cmd ? outputState(String, String);
cmd ? command(String, String);
temp ? value(UNKNOWN);
accu ? value(UNKNOWN);
ci ? value(UNKNOWN);
co !* value(UNKNOWN);
out !* value(UNKNOWN)

behaviour specification
(ALU_Operator("ALU"; "UNK")[coreToOut/output, inToCore/input] || Output(false; "ALU"; "co"; false)\{cmd, clock}[coreToOut/toCore, co/output] || Output(true; "ALU"; "out"; true)[clk/clock, coreToOut/toCore, out/output] || Input("ci")[inToCore/toCore, ci/input] || Input("temp")[inToCore/toCore, temp/input] || Input("accu")[inToCore/toCore, accu/input])\{inToCore, coreToOut}

---

cluster class Rom()

communication channels
pc, out, cmd, clk

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
out !* value(UNKNOWN);
cmd ? outputState(String, String);
clk ? afterClock();
pc ? value(UNKNOWN)

behaviour specification
(Rom(8; "pc"; "out"; 1024; "ROM")[coreToOut/output, inToCore/input] || Output(true; "ROM"; "out"; true)[clk/clock, coreToOut/toCore, out/output] || Input("pc")[inToCore/toCore, pc/input])\{coreToOut, inToCore}

---

cluster class Register(cmdResetVal: Integer; outputName: String; systemResetVal: Integer; inputName: String; outputTS: Boolean; registerName: String; outputDefaultDisabled: Boolean; width: Integer; defaultCommand: String)

communication channels
in, cmd, clk, out

message interface
in ? value(UNKNOWN);

```
cmd ? command(String, UNKNOWN, UNKNOWN);
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? test(String, String);
clk ? afterClock();
clk ? clockPulse();
clk ? beforeClock();
cmd ! value(UNKNOWN, UNKNOWN);
out !* value(UNKNOWN)

behaviour specification
(Register(width; cmdResetVal; systemResetVal; registerName; defaultCommand;
outputName)[clk/clock, outToCore/output, inToCore/input] ||
Input(inputName)[inToCore/toCore, in/input] || Output(outputTS; registerName; outputName;
outputDefaultDisabled)[clk/clock, outToCore/toCore, out/output])\{inToCore, outToCore}
```

Appendix E: POOSL specification of other IDaSS elements

E.1 Buffer

```
process class Buffer(name: String; width: Integer)
/* no superclass */

instance variables
contents: Integer

communication channels
cmd, output, input

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? test(String, String);
cmd ! value(UNKNOWN, UNKNOWN);
input ? value(String, Integer);
output ! value(UNKNOWN, UNKNOWN)

initial method call
init()()

instance methods
init()()
| dest, fullOutputName, stateCmd : String |
    contents := -3;         /*  Set default contents to UNK */
    fullOutputName := name concat(":out");
    main()() interrupt(
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState(fullOutputName, "enable")
        else
            cmd ! outputState(fullOutputName, "disable")
        fi ).

main()()
| newValue : Integer; source, dest, testCmd : String |
    sel
        input ? value(source, newValue | newValue != contents);
        contents := newValue;
        output ! value("out", contents)
    or
        cmd ? test(dest, testCmd | dest=name);
        if testCmd = "normal" then
            cmd ! value(name, new(BoundedInteger) init(contents, width))
        fi
    les;
    main()().
```

E.2 CAM

```
process class CAM(width: Integer; adrOutputName: String; defaultCommand: String;
firstMatchMode: Boolean; matchMaskInputName: String; defaultMask: Integer;
cmdResetValue: Integer; depth: Integer; name: String; defaultData: Integer;
matchWriteDataInputName: String; systemResetValue: Integer; adrInputName: String;
dataOutputName: String; matchDataInputName: String)
/* no superclass */
```

```
instance variables
maxVal: Integer; matchData: Integer; matchMask: Integer; numberOfMatches: Integer;
matchSetMask: Integer; addressMask: Integer; noMatchSetMask: Integer; buffer: Array;
cmdReceived: Boolean; matchResetMaskNot: Integer; command: Integer; addressFMC:
Integer; tempWidth: Integer; noMatchResetMaskNot: Integer; dataFMC: Integer

communication channels
clock, cmd, output, input

message interface
clock ? afterClock();
cmd ? test(String, String);
input ? value(String, UNKNOWN);
cmd ? command(String, String, Integer);
cmd ! value(UNKNOWN, UNKNOWN);
output ! value(UNKNOWN, UNKNOWN);
clock ? clockPulse()

initial method call
init()()

instance methods
init()()
| i : Integer |
    buffer := new(Array) size(depth) putAll(systemResetValue);
    buffer put(3,1);
    command := 0;
    numberOfMatches := -3;    /*  Return-value at auxiliary test-request     */
    dataFMC := -3;            /*  Return-value at normal test-request        */
    cmdReceived := false;
    matchData := defaultData;
    matchMask := defaultMask;
    maxVal := 2 power(width) - 1;
    matchSetMask := 0;
    matchResetMaskNot := maxVal;
    noMatchSetMask := 0;
    noMatchResetMaskNot := maxVal;
    i := 1;
    tempWidth := 1;
    while i < depth do
        tempWidth := tempWidth + 1;   /*  This loop is used to determine the    */
        i := i * 2                    /*  width of the Bounded Integer that is  */
    od;                               /*  returned on an auxiliary test request.*/
    main()().

main()()
| dest, newCmd, testCmd : String; cmdVal : Integer |
    sel
        cmd ? command(dest, newCmd, cmdVal | dest=name);
        if newCmd get(newCmd length()) != ':' then
            if newCmd = "reset" then
                command := 32
            else
                if cmdReceived then
                /*  Multiple commands...    */
                    command := (command | 64)
                else
                    cmdReceived := true;
                    if newCmd = "match" then
                        command := (command | 1)
                fi;
                if newCmd = "wrfirst" then
                    command := (command | 2)
                fi;
                if newCmd = "wrall" then
                    command := (command | 4)
                fi;
                if newCmd = "rdaddr" then
                    command := (command | 8)
                fi;
                if newCmd = "wraddr" then
                    command := (command | 16)
                fi
            fi
        fi
    else
        if newCmd = "data:" then
            matchData := cmdVal
        fi;
        if newCmd = "mask:" then
            matchMask := cmdVal
        fi;
        if newCmd = "mset:" then
            matchSetMask := cmdVal
        fi;
        if newCmd = "mres:" then
            matchResetMaskNot := new(BoundedInteger) init(cmdVal, width)
                not() getVal()
        fi;
        if newCmd = "mdata:" then
            matchSetMask := cmdVal;
            matchResetMaskNot := cmdVal
        fi;
        if newCmd = "nmset:" then
            noMatchSetMask := cmdVal
        fi;
        if newCmd = "nmres:" then
            noMatchResetMaskNot := new(BoundedInteger) init(cmdVal, width)
                not() getVal()
        fi;
        if newCmd = "nmdata:" then
            noMatchSetMask := cmdVal;
            noMatchResetMaskNot := cmdVal
        fi
    fi
or
    cmd ? test(dest, testCmd | dest=name);
    if testCmd = "normal" then
        cmd ! value(name, new(BoundedInteger) init(dataFMC, width))
    else
        cmd ! value(name, new(BoundedInteger) init(numberOfMatches, tempWidth))
    fi
or
    clock ? clockPulse;
    numberOfMatches := -3;
    cmdExecute()();
    command := 0;
    cmdReceived := false;
    matchData := defaultData;
    matchMask := defaultMask;
    matchSetMask := 0;
```

```
        matchResetMaskNot := maxVal;
        noMatchSetMask := 0;
        noMatchResetMaskNot := maxVal;
        clock ? afterClock
    les;
    main()().

cmdExecute()()
| i, cellVal, writeVal : Integer; source : String |
    if (command & 32) != 0 then
    /*  Reset  */
        buffer putAll(cmdResetValue);
        output ! value(adrOutputName, 0);
        output ! value(dataOutputName, -3);
        if defaultCommand = "rdaddr" then
            command := 8
        else
            command := 1
        fi
    fi;
    if (command & 64) != 0 then
    /*  Multiple...  */
        numberOfMatches := -3
    else
        if (command & %00111) != 0 then
        /*  match / wrfirst / wrall     */
            input ? value(source, matchData | source=matchDataInputName);
            input ? value(source, matchMask | source=matchMaskInputName);
            input ? value(source, matchSetMask | source=matchWriteDataInputName);
            matchResetMaskNot := matchSetMask;
            i := 1;
            numberOfMatches := 0;
            dataFMC := 0;
            addressFMC := depth;
            addressMask := 0;
            while i <= depth do
                cellVal := buffer get(i);
                if (((cellVal & matchMask) = matchData) & (cellVal >= 0)) then
                /*  Note:   A cell with contents UNK will never match     */
                /*          (in IDaSS an UNK cell will match when the     */
                /*          match mask is zero. This will rarely ever     */
                /*          happen, so this wil not be a major problem.)  */
                    numberOfMatches := numberOfMatches + 1;
                    addressMask := (addressMask | (2 power(i - 1)));
                    if addressFMC = depth then
                    /*  This is the "First Matched Cell"...*/
                        dataFMC := cellVal;
                        addressFMC := i - 1;
                        if command = 2 then
                        /*  wrfirst  */
                            writeVal := ((cellVal & matchResetMaskNot)
                                | matchSetMask);
                            buffer put(i, writeVal)
                        fi
                    fi;
                    if command = 4 then
                    /*  wrall    */
                        writeVal := ((cellVal & matchResetMaskNot)
                            | matchSetMask);
                        buffer put(i, writeVal)
                    fi
                else
                    if command = 4 then
                    /*  wrall   */
                        writeVal := ((cellVal & noMatchResetMaskNot)
                            | noMatchSetMask);
                        buffer put(i, writeVal)
                    fi
                fi;
                i := i + 1
            od;
/*          if firstMatchMode then
                output ! value(adrOutputName, addressFMC)
            else
                output ! value(adrOutputName, addressMask)
            fi  */
        else
            if command = 8 then
            /*  rdaddr  */
                input ? value(source, address | source=adrInputName);
                if address<0 then
                    numberOfMatches := -3;
                    dataFMC := -3;
                    output ! value(adrOutputName, -3)
                else
                    if address >= depth then
                        numberOfMatches := 0;
                        dataFMC := 0;
                        if firstMatchMode then
                            output ! value(adrOutputName, address)
                        else
                            output ! value(adrOutputName, 0)
                        fi
                    else
                        numberOfMatches := 1;
                        dataFMC := buffer get(address + 1);
                        if firstMatchMode then
                            output ! value(adrOutputName, address)
                        else
                            output ! value(adrOutputName, 2 power(address))
                        fi
                    fi
                fi
            else
                if command = 16 then
                /*  wraddr  */
                    input ? value(source, address | source=adrInputName);
                    if address<0 then
                        numberOfMatches := -3;
                        dataFMC := -3;
                        output ! value(adrOutputName, -3)
                    else
                        if address >= depth then
                            numberOfMatches := 0;
                            dataFMC := 0;
                            if firstMatchMode then
                                output ! value(adrOutputName, address)
                            else
                                output ! value(adrOutputName, 0)
                            fi
                        else
                            numberOfMatches := 1;
```

```
                              dataFMC := buffer get(address + 1);
                              writeVal := ((cellVal & matchResetMaskNot)
                                   I matchSetMask);
                              buffer put(address+1, writeVal);
                              if firstMatchMode then
                                   output ! value(adrOutputName, address)
                              else
                                   output ! value(adrOutputName, 2 power(address))
                              fi
                         fi
                    fi
               fi
          fi
     fi;
     output ! value(dataOutputName, dataFMC)
fi.
```

E.3 *Constant Generator*

```
process class ConstGen(width: Integer; outputTS: Boolean; name: String; defaultValue:
Integer)
/* no superclass */

instance variables
fullOutputName: String; outputVal: Integer; cmdReceived: Boolean

communication channels
clock, cmd, output

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? test(String, String);
cmd ? command(String, String, Integer);
cmd ! value(UNKNOWN, UNKNOWN);
output ! value(UNKNOWN, UNKNOWN);
clock ? beforeClock()

initial method call
init()()

instance methods
init()()
I dest, stateCmd : String I
     fullOutputName := name concat(":out");
     outputVal := defaultValue;
     output ! value("out", outputVal);
     cmdReceived := false;
     if outputTS then
          main()() interrupt(
               cmd ? outputState(dest, stateCmd I dest=name);
               if stateCmd = "enableAll" then
                    cmd ! outputState(fullOutputName, "enable")
               else
                    cmd ! outputState(fullOutputName, "disable")
               fi )
     else
          main()()
     fi.

main()()
```

```
I dest, newCmd, testCmd : String; cmdValue : Integer I
    sel
        cmd ? command(dest, newCmd, cmdValue I dest=name);
        if cmdReceived then
        /*  Multiple "setto: <value>"-commands received.    */
            outputVal := -3;
            output ! value("out", outputVal)
        else
            cmdReceived := true;
            outputVal := cmdValue;
            output ! value("out", outputVal);
            if outputTS then
            /*  Make sure that the three-state ouput is enabled. */
                cmd ! outputState(fullOutputName, "enable")
            fi
        fi
    or
        cmd ? test(dest, testCmd I dest=name);
        if testCmd = "normal" then
            cmd ! value(name, new(BoundedInteger) init(outputVal, width))
        fi
    or
        clock ? beforeClock;
        if cmdReceived not() then
            outputVal := defaultValue;
            output ! value("out", outputVal)
        else
            cmdReceived := false
        fi
    les;
    main()().
```

E.4 FIFO

```
process class FIFO(width: Integer; defaultNoWrite: Boolean; dataOutputName: String;
dataInputName: String; depth: Integer; raInputName: String; name: String; defaultNoRead:
Boolean)
/* no superclass */

instance variables
command: Integer; head: Integer; buffer: Array; length: Integer; tail: Integer; tempWidth:
Integer; outVal: Integer; cmdVal: Integer; raVal: Integer

communication channels
clock, cmd, output, input

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);
cmd ? test(String, String);
input ? value(String, UNKNOWN);
cmd ? command(String, String, Integer);
cmd ! value(UNKNOWN, UNKNOWN);
input ? value(String, Integer);
output ! value(UNKNOWN, UNKNOWN);
clock ? clockPulse()

initial method call
init()()

instance methods
```

```
main()()
I dest, newCmd, source, testCmd : String; newVal, newCmdVal, oldOutVal, address : Integer I
    sel
        cmd ? command(dest, newCmd, newCmdVal I dest=name);
        if newCmd = "read" then
            command := (command I 1)
        fi;
        if newCmd = "noread" then
            command := (command I 2)
        fi;
        if newCmd = "write" then
            command := (command I 4)
        fi;
        if newCmd = "nowrite" then
            command := (command I 8)
        fi;
        if newCmd = "write:" then
            command := (command I 16);
            cmdVal := newCmdVal
        fi;
        if newCmd = "reset" then
            command := 32
        fi
    or
        input ? value(source, newVal I ((source=raInputName)
            & (newVal != raVal)));
        raVal := newVal;
        oldOutVal := outVal;
        if ((raVal < 0) I (raVal >= length)) then
            outVal := -3
        else
            address := head + raVal;
            if address > depth then
                address := address - depth
            fi;
            outVal := buffer get(address)
        fi;
        if outVal != oldOutVal then
            output ! value(dataOutputName, outVal)
        fi
    or
        cmd ? test(dest, testCmd I dest=name);
        if testCmd = "normal" then
            if length = 0 then
                cmd ! value(name, new(BoundedInteger) init(-3,width))
            else
                cmd ! value(name, new(BoundedInteger) init(buffer get(head),width))
            fi
        else
            cmd ! value(name, new(BoundedInteger) init(length, tempWidth))
        fi
    or
        clock ? clockPulse;
        cmdExecute()();
        command := 0
    les;
    main()().

incHead()()
    if head = depth then
        head := 1
    else
        head := head + 1
    fi.

init()()
| i : Integer; fullOutputName, dest, stateCmd : String |
    buffer := new(Array) size(depth);
    head := 1;
    tail := 1;
    length := 0;
    command := 0;
    cmdVal := -3;
    raVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    i := 1;
    tempWidth := 1;
    while i < depth do
        tempWidth := tempWidth + 1;   /*  This loop is used to determine the    */
        i := i * 2                    /*  width of the Bounded Integer that is  */
    od;                               /*  returned on an auxiliary test request.*/
    main()() interrupt(
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState(fullOutputName, "enable")
        else
            cmd ! outputState(fullOutputName, "disable")
        fi ).

cmdExecute()()
| source : String |
    if (command & 32) != 0 then
    /*  Reset   */
        head := 1;
        tail := 1;
        length := 0
    else
        if (((command & %011100) > 16) | ((command & %001100) > 8)
            | ((command & %000011) > 2)) then
        /*  Multiple commands requested, do nothing !   */
            skip
        else
            if (command & %11) = 0 then
                if defaultNoRead then
                    command := (command | 2)
                else
                    command := (command | 1)
                fi
            fi;
            if (command & %11100) = 0 then
                if defaultNoWrite then
                    command := (command | 8)
                else
                    command := (command | 4)
                fi
            fi;
            if (command & %1) != 0 then
            /*  Read    */
                if (command & %10100) != 0 then
                /*  Read & Write / Write:    */
                    if length = 0 then
                    /*   Buffer is empty...     */
                         skip
                    else
                         incHead()();
                         if (command & %100) != 0 then
                         /*   Write    */
                              input ? value(source, cmdVal | source=dataInputName)
                         fi;
                         buffer put(tail, cmdVal);
                         incTail()()
                    fi
               else
               /*   Read & NoWrite      */
                    if length = 0 then
                    /*   Buffer is empty, there's nothing to read ! */
                         skip
                    else
                         incHead()();
                         length := length - 1
                    fi
               fi
          else
          /*   NoRead      */
               if (((command & %10100) = 0) | (length = depth)) then
                    skip
               else
                    if (command & %100) != 0 then
                    /*   Write    */
                         input ? value(source, cmdVal | source=dataInputName)
                    fi;
                    buffer put(tail, cmdVal);
                    incTail()();
                    length := length + 1
               fi
          fi
     fi
fi.

incTail()()
     if tail = depth then
          tail := 1
     else
          tail := tail + 1
     fi.
```

E.5 LIFO

process class LIFO(width: Integer; dataOutputName: String; dataInputName: String; depth: Integer; raInputName: String; name: String)
/* no superclass */

instance variables
command: Integer; head: Integer; buffer: Array; length: Integer; cmdReceived: Boolean; tempWidth: Integer; outVal: Integer; cmdVal: Integer; raVal: Integer

communication channels
clock, cmd, output, input

message interface
cmd ! outputState(UNKNOWN, UNKNOWN);
cmd ? outputState(String, String);

```
clock ? afterClock();
cmd ? test(String, String);
input ? value(String, UNKNOWN);
cmd ? command(String, String, Integer);
cmd ! value(UNKNOWN, UNKNOWN);
output ! value(UNKNOWN, UNKNOWN);
input ? value(String, Integer);
clock ? clockPulse()

initial method call
init()()

instance methods
updateReadPort()()
| oldOutVal, address : Integer |
    oldOutVal := outVal;
    if ((raVal < 0) | (raVal >= length)) then
        outVal := -3
    else
        address := head + raVal;
        if address > depth then
            address := address - depth
        fi;
        outVal := buffer get(address)
    fi;
    if outVal != oldOutVal then
        output ! value(dataOutputName, outVal)
    fi.

init()()
| i : Integer; fullOutputName, dest, stateCmd : String |
    buffer := new(Array) size(depth);
    head := 1;
    length := 0;
    command := 0;
    cmdReceived := false;
    cmdVal := -3;
    raVal := -3;
    outVal := -3;
    fullOutputName := name concat(":") concat(dataOutputName);
    i := 1;
    tempWidth := 1;
    while i < depth do
        tempWidth := tempWidth + 1;    /*  This loop is used to determine the     */
        i := i * 2                     /*  width of the Bounded Integer that is   */
    od;                                /*  returned on an auxiliary test request. */
    main()() interrupt(
        cmd ? outputState(dest, stateCmd | dest=name);
        if stateCmd = "enableAll" then
            cmd ! outputState(fullOutputName, "enable")
        else
            cmd ! outputState(fullOutputName, "disable")
        fi ).

incHead()()
    if head = depth then
        head := 1
    else
        head := head + 1
    fi.
```

```
cmdExecute()()
| source : String; doNothing : Boolean; belowHeadVal : Integer |
    if (command & 1024) != 0 then
    /*  Reset  */
        head := 1;
        length := 0
    else
        doNothing := false;
        if (((command & 2048) != 0) | cmdReceived not()) then
        /*  Multiple or Hold, do nothing !      */
            doNothing := true
        fi;
        if (((length = 0) & ((command & %100100110) != 0))
            | ((length < 2) & ((command & %1001011000) != 0))
            | ((length = depth) & ((command & %10100001) != 0))) then
        /*  Command can't be executed with current buffer-length !  */
            doNothing := true
        fi;
        if doNothing not() then
            if (command & %1101) != 0 then
            /*  Get the value at the write port... */
                input ? value(source, cmdVal | source=dataInputName)
            else
                if (command & %1100000) != 0 then
                    cmdVal := buffer get(head)
                fi
            fi;
            if (command & %1101011110) != 0 then
                incHead()();
                length := length - 1
            fi;
            if command = 64 then
                belowHeadVal := buffer get(head)
            fi;
            if (command & %1001011000) != 0 then
                incHead()();
                length := length - 1
            fi;
            if (command & %1111101101) != 0 then
                decHead()();
                buffer put(head, cmdVal);
                length := length + 1
            fi;
            if command = 64 then
                decHead()();
                buffer put(head, belowHeadVal);
                length := length + 1
            fi
        fi
    fi.

decHead()()
    if head = 1 then
        head := depth
    else
        head := head - 1
    fi.

main()()
| dest, newCmd, source, testCmd : String; newVal, newCmdVal : Integer |
    sel
```

```
        cmd ? command(dest, newCmd, newCmdVal I dest=name);
        if newCmd = "reset" then
            command := 1024
        else
            if cmdReceived then
            /*  Multiple commands...    */
                command := (command I 2048)
            else
                cmdReceived := true;
                if newCmd = "push" then
                    command := (command I 1)
                fi;
                if newCmd = "pop" then
                    command := (command I 2)
                fi;
                if newCmd = "replace" then
                    command := (command I 4)
                fi;
                if newCmd = "poprepl" then
                    command := (command I 8)
                fi;
                if newCmd = "pop2" then
                    command := (command I 16)
                fi;
                if newCmd = "pushcopy" then
                    command := (command I 32)
                fi;
                if newCmd = "swap" then
                    command := (command I 64)
                fi;
                if newCmd = "push:" then
                    command := (command I 128);
                    cmdVal := newCmdVal
                fi;
                if newCmd = "replace:" then
                    command := (command I 256);
                    cmdVal := newCmdVal
                fi;
                if newCmd = "poprepl:" then
                    command := (command I 512);
                    cmdVal := newCmdVal
                fi
            fi
        fi
    or
        input ? value(source, newVal I ((source=raInputName)
            & (newVal != raVal)));
        raVal := newVal;
        updateReadPort()()
    or
        cmd ? test(dest, testCmd I dest=name);
        if testCmd = "normal" then
            if length = 0 then
                cmd ! value(name, new(BoundedInteger) init(-3, width))
            else
                cmd ! value(name, new(BoundedInteger) init(buffer get(head), width))
            fi
        else
            cmd ! value(name, new(BoundedInteger) init(length, tempWidth))
        fi
    or
```

```
        clock ? clockPulse;
        cmdExecute()();
        command := 0;
        cmdReceived := false;
        clock ? afterClock;
        updateReadPort()()
    les;
    main()().
```

References

[Int93] Intel. *Embedded microcontrollers and processors Vol. 1.* Rotterdam: Intel, 1993.

[Mic97] R. Michielsen. *Implementing POOSL in IDaSS.* Master's thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997.

[Phi94] Philips. *8048-based 8-bit microcontrollers.* Eindhoven: Philips, 1994.

[vdPV97] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems: The method Software/Hardware Engineering (SHE).* PhD thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997.

[Ver90] A.C. Verschueren. *IDaSS for ULSI manual.* Eindhoven: Eindhoven University of Technology, 1990.

[Ver92] A.C. Verschueren. *An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems.* PhD thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1992.

[Ver97] A.C. Verschueren. *IDaSS help system.* This is the helpfile that is integrated in the IDaSS software.

[Voe94] J.P.M. Voeten. *POOSL: A Parallel Object-Oriented Specification Language.* In: Proceedings of the Eight Workshop Computer Systems, pages 25-45, Amsterdam, The Netherlands, 1994. University of Amsterdam.

[Voe95a] J.P.M. Voeten. *POOSL: An Object-Oriented Language for the Analysis and Design of Hardware/Software Systems.* EUT Report 95-E-290, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1995.

[Voe95b] J.P.M. Voeten. *Semantics of POOSL: An Object-Oriented Language for the Analysis and Design of Hardware/Software Systems.* EUT Report 95-E-293, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, October 1995.

[VvdPS] J.P.M. Voeten, P.H.A. van der Putten, and M.P.J. Stevens. *Formal modelling of reactive hardware/software systems.* To be published in Proceedings of ProRISC/CSSP'97, Utrecht: STW, Technology Foundation.