Dynamic Microcontroller in an XC4000 FPGA

December 1994

Application Note BY KEN CHAPMAN

Summary

This Application Note describes how to build a microcontroller with dynamic bus size for implementing complex state machines and processing functions either as part of a system, or for use during development and test.

Xilinx Family

XC4000 and derivatives

Demonstrates

X-BLOX™ module generator

Using RAM and PROM

Table of Contents

Features

- Dynamic bus width — 1 to ***n*** bits
- 16 Data Registers
- 16 I/O Ports
- Flexible instruction set
  - Add and Subtract
  - Logical OR, AND, and XOR
  - Load, In, Out
  - Jump group, shift and rotate sets
- Program ROM — Dynamic depth from 16 to 256 instructions
- Typically >3 MIPS performance
- Unique architecture for highly compact design in XC4000 device

Overview

Microcontrollers are common in many digital systems. The relatively low cost of these complex devices makes them ideal for certain applications. The decision to include a microcontroller in a design is often very clear because it transforms the design effort from a logic design into more of a software design.

Xilinx FPGA devices offer similar flexibility for all the other logic functions required in such systems. These would include special high performance circuits, or signal conditioning for the microcontroller.

With the ever increasing size and reductions in cost of FPGA devices, it is now possible to implement a complete system on one device. The microcontroller and associated software can be replaced by a complex state machine dedicated to the function. However, such state machines are often difficult to develop. Consequently, a microcontroller usually remains a discrete device, unless board space is at a premium.

A microcontroller is often used for diagnostics and test functions in a system. Small programs are easy to write, and very flexible.

This application note offers an alternative to discrete microcontrollers by providing a microcontroller macro for an XC4000. This microcontroller macro may be used for board test and diagnostics, regardless of the function the device will perform after reprogramming. It is also useful in systems where the control logic is too complex for hardware logic, but almost too simple for software. Some applications requiring high security such as data encryptors may also incorporate this macro.

The macro, named 'PSM', is a programmable state machine. The macro's name conveys its potential use. Although full featured, the macro is limited by the amount of FPGA device that the designer is willing to convert to program ROM.

A good efficient instruction set and the ability to avoid the constraints of a fixed bus width make programs

compact. During diagnostics and test, the entire device is available to the user for this function, and the size of the program ROM is not an issue. In a system application, the code complexity is the deciding factor.

Demand for a Compact Architecture

When developing the PSM macro, silicon efficiency was the primary focus. High performance circuits will *always* be implemented as dedicated circuits. Hence, all design decisions for this macro favor optimizations for minimum area (low CLB count), with processing speed a second priority.

The major design task was to define the functionality of the microcontroller. This is defined by:

- The bus width
- The instruction set

Each affects silicon efficiency in two ways—the size of the processing core, and the size of the program (and the corresponding program ROM) to carry out the sequence of operations.

The data bus determines the width of all data paths and processing elements such as the ALU and data registers. However, a suitably wide data bus simplifies the program code. For example, the addition of two 16-bit values is only one instruction in a 16-bit microprocessor, but is two instructions in an 8-bit version. Clearly this decision also effects the overall system performance.

The complexity and the range of available instructions impacts the amount of logic required for the data paths and processing elements. Too small a range of instructions—or the inability to manipulate data effectively—results in long programs, with poor system performance, that require large program ROMs. Suitable instructions lead to efficient programs.

Consequently, the ability to choose the precise data bus width required for a system, and the availability of a highly efficient set of instructions result in a usable microcontroller macro.

Exploiting FPGA Features

This microcontroller design exploits various XC4000 FPGA features including:

- ***Arithmetic carry logic*** — ALU Add and Subtract functions, program counter
- ***ROM*** — Program memory
- ***RAM*** — Data registers

These features, and the ultimate flexibility of an FPGA, offer some significant advantages.

Traditional microprocessors and microcontrollers hold their program code in standard EPROMs. This approach results in variable length instructions and the added complexity of op-code and operand fetch cycles. Furthermore, the data bus is a shared resource for the manipulation of program code and the processing of data.

The FPGA architecture permits the program code and the data bus to be separated. This allows the data bus to be a different width from that required for the instruction codes. In fact, it permits the data bus to be the width most suitable for the application. Furthermore, the FPGA can implement any width of ROM to accommodate the program instructions. Having a ROM wider than the normal eight bits enables the instruction and operands to be defined in a single access for compact and fast system performance.

Practical Aspects of Implementation

The instruction set and its encoding provides the key to the processor architecture. However, a few basic decisions must be made.

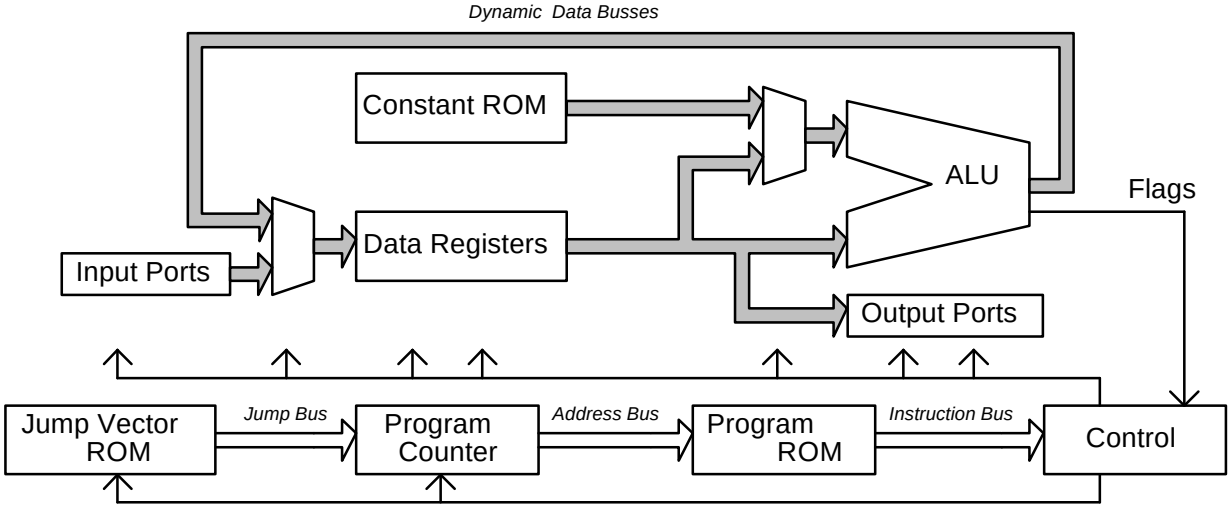

Figure 1. Dynamic microcontroller architecture.

X-BLOX™ provides a design entry method where the data path bus widths can be changed, and for the corresponding synthesized logic. X-BLOX is the preferred method for the design of this macro.

The XC4000 CLB RAM feature provides the ideal solution for building the data registers in the microcontroller. This makes the 16 registers extremely space efficient because a CLB contains two 16x1 RAMs.

Many processors include an accumulator in their structure. This has the advantage of implied instructions, which remove the need for two operands per instruction. Unfortunately it also results in a high percentage of instructions which simply move values into and out of the accumulator. It is important to keep the program code small in an FPGA, and hence all operations directly access the registers.

Program code is efficient when all the bits of the encoded instructions and operands are used. The ability to express the instruction and all the operands in a single access also leads to simple control circuits for the processor. All instructions are limited to a single access by making the program ROM the necessary width—knowing that there is a space advantage of shallow-but-wide ROMs over deep-but-narrow ROMs in the XC4000 FPGA.

Instructions and Encoding

Some factors are already determined by the previously stated architectural decisions. Others are defined by the actual functionality. Four main instruction types emerge for data processing, and one for program flow control, considering the number of registers and the access required by each instruction. These instruction types are shown in Table 1.

In most cases, a resultant value needs to be stored. Although it is possible to specify a third location, the additional operand information adds too much extra logic. Hence, the result will generally be placed back into Register A.

With 16 registers to access, four bits of encoded instruction are need to specify each register access. A total of eight bits are therefore dedicated to operand specification in a Type 1 instruction.

Table 1. Function Types

Type	Function
Type 1	Function of Register A with Register B
Type 2	Function of Register A with a constant value
Type 3	Function of Register A with I/O port access
Type 4	Data manipulation of Register A
Type 5	Program flow control and flag testing

For Type 3 instructions, Register A is again specified by four bits. Another easy architectural decision is the number of I/O ports. Sixteen ports can be specified by the same four bits used to access register B in a Type 1 instruction.

The specified constant in Type 2 instructions is a problem. Constants relate to the data processing, and hence are as wide as the data bus. Normally a fetch cycle is used to access the next memory location for the bits required to define the constant. In this macro, where dynamic bus width is desirable, a fetch cycle would place an upper limit on the bus width, and waste memory bits for smaller bus sizes.

The solution is to permit 16 pointers (defined by four bits) to a ROM. This ROM holds up to 16 constants of the same width as the data bus. A program therefore consists of:

- A *main* instruction memory of fixed width, and
- A *separate* constant memory of data bus width.

All Type 1, 2 and 3 instruction operands are therefore defined by only eight bits.

The actual operations need to be encoded. The bits required to encode the operations are directly related to the number of instructions. Too few instructions result in long programs, and too many result in an overly-large processor.

A minimum instruction set provides the largest number of functions with the least amount of instruction overlap—e.g. comparison can be done with a subtract instruction. This instruction set is organized into the following instruction types:

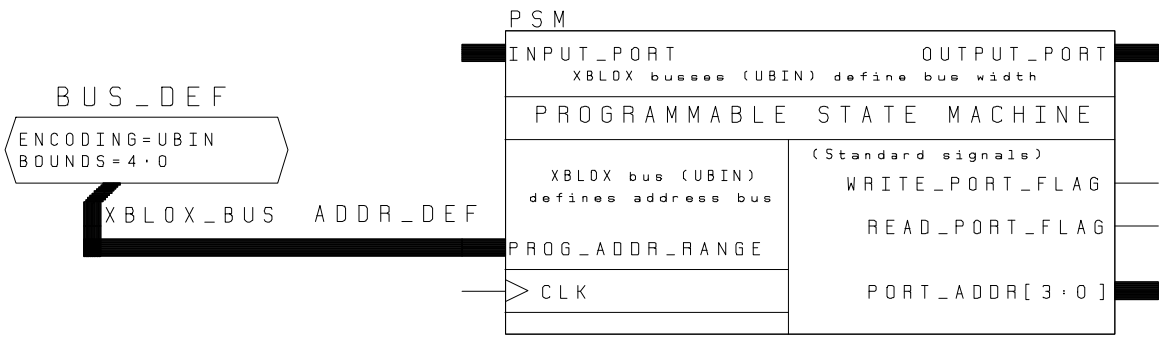

Figure 2. Defining the address bus range on the PSM macro symbol.

- ***Type 1***—Load, Add, Subtract, AND, OR, XOR
- ***Type 2***—Load, Add, Subtract, AND, OR, XOR
- ***Type 3***—Input, Output
- ***Type 4***—Shift group, Rotate group
- ***Type 5***—Jump group

This results in 17 basic instructions, although those of Type 4 and 5 require several variations. It would appear that five bits are required to encode the full range of instructions. However the encoding of Type 4 and 5 instructions can use some of the eight operand bits from the other instruction types which allows just four bits to encode the operation.

Type 4 instructions only require access to one register. The remaining four operand bits are available to define the shift or rotate process required. Shift and rotate are then encoded by one instruction code reducing the basic instruction count to 16. Instructions can now be represented by a total of 12 bits.

The remaining challenge is to implement the Type 5 instruction within the same 12 bits. The four bits of the operation code already define this as a jump instruction, leaving the eight operand bits.

It is possible to use eight bits to specify a relative jump of -128 to +127. This is sufficient for small programs, but would not leave any bits to encode the condition for the jump. Reducing the number of bits allocated to relative addressing would be very limiting. Absolute addressing presents the same problem as defining a constant did for Type 3 instructions. However the same solution is applicable, and hence four bits are used as a pointer to a small memory containing up to 16 jump vectors. The jump memory need only be wide enough to support the size of the program.

The testing of flags defines the remaining four bits on a Type 5 instruction. ZERO and CARRY flags provide suitable flow control to the user.

The actual encoding of all instructions keeps the logic to a minimum. Controlling the data flow and processing directly with the status of bits reduces size and increases performance. The complete instruction encoding follows.

Instruction Quick Reference

function	code(hex)	function	code(hex)
ADD sx,sy	Dxy	SR0 sx	6xE
ADD sx,c	5xc	SR1 sx	6xF
SUB sx,sy	Cxy	SRX sx	6xA
SUB sx,c	4xc	SRA sx	6x8
OR sx,sy	8xy	RR sx	6xC
OR sx,c	0xc	SL0 sx	6x6
AND sx,sy	9xy	SL1 sx	6x7
AND sx,c	1xc	SLX sx	6x4
XOR sx,sy	Axy	SLA sx	6x0
XOR sx,c	2xc	RL sx	6x2
LD sx,sy	Bxy	JP j	70j
LD sx,c	3xc	JP Z,j	73j
		JP C,j	7Cj
IN sx,p	Exp	JP NZ,j	72j
OUT sx,p	Fxp	JP NC,j	78j
		JP GT,j	7Aj
		JP LT,j	7Bj

Type 1 and 2 instructions — Arithmetic and Load Functions

```
1 1       CODE
Op_code     |  1 0 9 8  |  7 6 5 4  |  3 2 1 0 |
-------     -------------------------------------
ADD sx,sy      1 1 0 1     x x x x     y y y y
SUB sx,sy      1 1 0 0     x x x x     y y y y
OR  sx,sy      1 0 0 0     x x x x     y y y y
AND sx,sy      1 0 0 1     x x x x     y y y y
XOR sx,sy      1 0 1 0     x x x x     y y y y
LD sx,sy       1 0 1 1     x x x x     y y y y
ADD sx,c       0 1 0 1     x x x x     c c c c
SUB sx,c       0 1 0 0     x x x x     c c c c
OR  sx,c       0 0 0 0     x x x x     c c c c
AND sx,c       0 0 0 1     x x x x     c c c c
XOR sx,c       0 0 1 0     x x x x     c c c c
LD sx,c        0 0 1 1     x x x x     c c c c
               | | | |
               | | | |       Select operation
               | | | |       ----------------
               | | | |__  1 - Add
               | | | |    0 - Subtract
               | | | |__                  0 0 - OR
               | | |____}-------------->  0 1 - AND
               | |                       1 0 - XOR
               | |                       1 1 - LOAD
               | |
               | |______  0 - Logical or load operation
               |        1 - Arithmetic operation
               |
               |________  0 - 'sx' and constant
                        1 - 'sx' and 'sy'
```

Notes:

- 'c' is a 4 bit pointer (cccc) to a constant table.
- 'sx' is any one of 16 registers represented by 4 bits (xxxx).
- 'sy' is any one of 16 registers represented by 4 bits (yyyy).
- The result of operation is placed into 'sx'.
- All commands effect ZERO and CARRY flags except LOAD.
- **ADD** and **SUB** commands will include the value of the carry flag in the calculation.

Type 3 Instructions — Ports

```
1 1       CODE
Op_code     |  1 0 9 8  |  7 6 5 4  |  3 2 1 0 |
-------     -------------------------------------
IN sx,p        1 1 1 0     x x x x     p p p p
OUT sx,p       1 1 1 1     x x x x     p p p p
```

Notes:

- 'sx' is any one of 16 registers represented by 4 bits (xxxx).
- 'p' is a 4 bit port address (pppp).
- flags : no effect.

Type 4 Instructions — Shift and Rotate group

```
1 1       CODE
Op_code     |  1 0 9 8  |  7 6 5 4  |  3 2 1 0 |
-------     -------------------------------------
SR0 sx         0 1 1 0     x x x x     1 1 1 0
SR1 sx         0 1 1 0     x x x x     1 1 1 1
SRX sx         0 1 1 0     x x x x     1 0 1 X
SRA sx         0 1 1 0     x x x x     1 0 0 X
RR sx          0 1 1 0     x x x x     1 1 0 X
SL0 sx         0 1 1 0     x x x x     0 1 1 0
SL1 sx         0 1 1 0     x x x x     0 1 1 1
SLX sx         0 1 1 0     x x x x     0 1 0 X
SLA sx         0 1 1 0     x x x x     0 0 0 X
RL sx          0 1 1 0     x x x x     0 0 1 X
                                       | | | |
                direction 0 - left  ___| | | |
                          1 - right      | | |
                                         | | |
         select bit to move in           | | |
                                         | | |
        0 0 - carry flag                 | | |
        0 1 - msb                  _____| | |
        1 0 - LSB                  _______| |
        1 1 - Forced value                 |
                                           |
       Forced value of bit to shift in_____|
```

Notes :

- 'sx' is any one of 16 registers represented by 4 bits (xxxx).
- ZERO and CARRY flags may be effected.
- Functions:
  - **SR0** — *shift right zero*, forcing 0 into MSB, carry takes value from LSB.
  - **SR1** — *shift right one*, forcing 1 into MSB, carry takes value from LSB.
  - **SRX** — *shift right extended*, MSB copied into MSB, carry takes value from LSB.
  - **SRA** — *shift right arithmetic*, carry moved into MSB, carry takes value from LSB.
  - **RR** — *rotate right*, LSB moved into MSB, carry takes value from LSB.
  - **SL0** — *shift left zero*, forcing 0 into LSB, carry takes value from MSB.
  - **SL1** — *shift left one*, forcing 1 into LSB, carry takes value from MSB.
  - **SLX** — *shift left extended*, LSB copied into LSB, carry takes value from MSB.
  - **SLA** — *shift left arithmetic*, carry moved into LSB, carry takes value from MSB.
  - **RL** — *rotate left*, MSB moved into LSB, carry takes value from MSB.

Type 5 Instructions —- Jump group

```
1 1       CODE
Op_code     |  1 0 9 8  |  7 6 5 4  |  3 2 1 0 |
-------     -------------------------------------

JP j           0 1 1 1     0 X 0 X     j j j j
JP Z,j         0 1 1 1     0 X 1 1     j j j j
JP C,j         0 1 1 1     1 1 0 X     j j j j
JP NZ,j        0 1 1 1     0 X 1 0     j j j j
JP NC,j        0 1 1 1     1 0 0 X     j j j j
JP GT,j        0 1 1 1     1 0 1 0     j j j j
JP LT,j        0 1 1 1     1 0 1 1     j j j j
                           | | | |
                           | | | |_zero flag status
                           | | |_look at zero flag
                           | |_carry flag status
                           |_look at carry flag
```

Notes:

- 'j' is a 4 bit pointer (jjjj) to a jump vector table.
- Conditional jumps -
  - **Z** — Jump if ZERO flag set
  - **NZ** — Jump if NOT ZERO
  - **C** — Jump if CARRY flag set
  - **NC** — Jump if NO CARRY
  - **GT** — Jump if GREATER THAN
  - **LT** — Jump if LESS THAN
  - **GT** and **LT** apply after a 'SUB sx, ?' such that the test is applied to 'sx'. i.e. sx < ?
- flags : no effect.

Programming Example

The following is an example of a program written to multiply two 4-bit numbers and provide an 8-bit result. Based on the resulting 8-bit product, it is more efficient to implement an 8-bit data bus. The schematic design for this function is shown in Figure 6 on page 12. The design is intended for the XC4000 demonstration board (containing a single 84-pin PLCC socket for an XC4003PC84C or XC4005PC84C device).

```
;
;Program for 4 bit Multiply on Demo Board
;
START:  LD  s3,04     ;4 bits to multiply
        XOR s2,s2     ;clear s2
        IN  s0,0      ;read switches
        LD  s1,s0     ;
        AND s1,F0     ;isolate high nibble
LOOP:   SR0 s0        ;test bit of low nibble
        JP NC,NO_ADD  ;bit was zero
        OR  s1,s1     ;clear carry flag
        ADD s2,s1     ;accumulate result
NO_ADD: SRA s2        ;shift result
        SUB s3,01     ;
        JP NZ,LOOP    ;test if all 4 bits used
        OUT s2,1      ;display output
        JP START      ;repeat
```

Figure 3. Multiply program written in PSMBLE.

The macro connects to the rest of the circuit in such a way that port connections define the data bus width to be synthesized by X-BLOX. The program address

range must be set by attaching a BUS_DEF symbol to the PROG_ADDR_RANGE bus input on the macro and adjusting the 'BOUNDS=' parameter on the BUS_DEF symbol as shown in Figure 2. This parameter may be adjusted later if the program turns out to be smaller or larger than expected.

The design is then processed using the XACT™ 5.0 FPGA development system. The results is three ROM template files for the user's program code.

- **PROGRAM.MEM** contains the main instructions.
- **CONSTANT.MEM** defines any data constants required.
- **JUMP.MEM** contains the vectors for any jump instructions.

The names of these files may be changed by redefining the 'FILE=' attribute on the X-BLOX PROM symbols within the macro. The internal details of the PSM macro, including the X-BLOX data paths and ROMs, are shown in Figure 7 and Figure 8.

An assembler, called 'PSMBLE', is written in QBASIC and is included with the demonstration designs (see ***PSMBLE Assembler for PSM*** for details on using it). The program generates the required three data files from assembly code, simplifying the task of creating the constant pointers and jump vectors.

When the MEM files are ready, the XACT tools are used again to process the design including the ROM data definitions.

Size and Performance

Both size and performance of a dynamic macro are difficult to evaluate, but here are some guidelines.

The control logic is very simple because of the instruction encoding. In fact only about ten CLBs carry out the instruction decoding and implement the control state machine. Although the absolute performance depends on the maximum clock frequency, the state machine dictates the number of clock cycles (t-states) required to perform each instruction:

- All instructions *excluding* JUMP group require six cycles.
- JUMP group (condition true or false) require one cycle.

The size of the program counter and JUMP ROM depend on the size of the program in the PROGRAM ROM. However, at one CLB per address bit, no more than eight CLBs are ever used for these combined elements.

The instruction format minimizes the size of programs, and hence the size of the PROGRAM ROM. The ROM has a fixed width of 12 bits, but the depth is defined by the PROG_ADDR_RANGE on the PSM macro.

The ROMs are built from function generators in the XC4000 CLBs. Fundamentally, 16 or 32 addressable locations are available. Larger memories are formed by combining CLBs. When the PSM macro is only a portion of the overall design, the user will want to keep the program relatively short in order to minimize the number of CLBs used for program storage. However, when the whole device is turned into a microcontroller for test purposes, then all CLBs are available to hold much longer and possibly less efficient programs.

The values shown in Table 2 indicate the number of CLBs required for programs of a given depth.

It is possible to adjust the DEPTH value in the PROGRAM.MEM file to further minimize the number of CLBs. For example, if only 135 program instructions are required, then setting DEPTH=135 reduces the

```
Compiler Report for program 'mult4.psm'.

addr code   label  instruction   cross-ref      comment

 00                ;
 00                ;Program for 4 bit Multiply on Demo
 00                ;
 00  330    START: LD  s3,0      ;0 -> '04'
 01  A22           XOR s2,s2
 02  E00           IN  s0,0
 03  B10           LD  s1,s0
 04  111           AND s1,1      ;1 -> 'F0'
 05  60E     LOOP: SR0 s0
 06  780           JP NC,0       ;0 -> 'NO_ADD'
 07  811           OR  s1,s1
 08  D21           ADD s2,s1
 09  628   NO_ADD: SRA s2
 0A  432           SUB s3,2      ;2 -> '01'
 0B  721           JP NZ,1       ;1 -> 'LOOP'
 0C  F21           OUT s2,1
 0D  702           JP 2          ;2 -> 'START'
```

Figure 4. Compiler report from PSMBLE showing jump vector and constant pointer assignments.

number of CLBs from 126 to only 73 CLBs—even though address bus is still 7:0.

The dynamic data paths have the largest effect on size and performance. The design maps very well into the architecture using no more than five CLBs per bit, including the constant ROM and the RAM based registers.

Table 2. Design Size as a Function of Address Range.

Program Size	Program Address Range	CLB count
16	3:0	6
32	4:0	12
64	5:0	30
128	6:0	60
256	7:0	126

Performance of this macro was a secondary consideration. The primary focus was on minimum CLB count. However, preliminary results indicate that the combined effect of instruction encoding, pipelined design, and X-BLOX implementation produces two to three times the performance of a typical 8-bit microcontroller.

The macro operates at up to 23 MHz in an XC4000-5 device. In most designs, however, the clock frequency is much lower. Under typical test applications, performance is usually of little consideration. In these applications, the macro can be clocked with the internal 8 MHz (nominal) clock source.

PSMBLE Assembler for PSM

This section describes the PSMBLE assembler for the PSM macro described earlier.

PSMBLE.BAS is written for QBASIC on the PC, and is supplied in original uncompiled format to allow modifications by the user. This provides a way to complement any changes made to the standard PSM macro.

Though careful effort makes this program easy to use, it has not received any official quality testing. Please help to improve this program by reporting any problems encountered.

What does it do?

The program can be executed from within QBASIC, or by invoking QBASIC with

```
qbasic /run psmble
```

```
Syntax table
------------

[ ] means optional
[ ... ] means option may be repeated
{ a | b } means that one of the enclosed must be specified
[a-z] means in the range specified.
::=  means 'is defined by'.

note : upper and lower case are always acceptable

each line should take the format:-

    program_line ::=    [ label : ] [ instruction ] [ ; comment ]

where

 label = lab_char [ lab_char... ]

 lab_char = { [A-Z] | [0-9] | _ }

 instruction = { arith | logical | port | shift | jump }

 arith = { ADD | SUB } reg_spec , second_operand

 logical = { OR | AND | XOR | LD } reg_spec , second_operand

 port = { IN | OUT } reg_spec , hex_char

 shift = { SR0 | SR1 | SRX | SRA | RR | SL0 | SL1 | SLX | SLA | RL } reg_spec

 reg_spec =  S hex_char

 jump = JP [ { C | NC | Z | NZ | GT | LT } , ] label

 second_op = { reg_spec | constant }

 constant = hex_char [ hex_char... ]

 hex_char = { [0-9] | [A-F] }

 comment =  [ any characters ]
```

Figure 5. Syntax table.

The program asks for the name of your assembly code file, and then processes it.

It takes only a few seconds to carry out the single pass process, followed by another few seconds resolving the jump addresses.

PSM requires that constants and jump addresses be separated from the main program code. It also ensures that no more than a maximum of 16 different constants or jump vectors are specified.

The program produces three files called:

- `program.dat`
- `constant.dat`
- `jump.dat`

These files contain the data needed for the corresponding MEM files used by X-BLOX in the macro schematic. It is a simple task to paste this data into each MEM file following the word 'DATA', and recompile the design.

Helpful Files

During the assembly process, PSMBLE creates several files to aid program development, debugging and verification:

`compile.log`

A complete listing of the compiled program with address and instruction codes. This file is a complete reconstruction of the original file, and consequently can be used to verify the assembly process.

`constant.tab`

Lists all the constants specified in the program against the pointer value (0 to F hex) to which they have been assigned.

`jump.tab`

Lists all the labels used in JUMP instructions against the vector number (0 to F hex) to which they have been assigned.

`label.tab`

Lists every label specified and its address. The program has a limit of 100 labels, but only 16 can actually be referenced in jump instructions.

`jumpaddr.tab`

Lists how the jump vectors and labels are resolved to form addresses used in the **`jump.mem`** file.

`format.prg`

This file is a formatted copy of the original program and may be adopted as a replacement for the original source file. It also acts as a verification of how PSMBLE interpreted the assembly program.

How to Write a Program for PSM

All the instructions are described in detail earlier in the application note. Complete syntax tables are provided in Figure 5.

List of Instructions

In this list of all instructions, '2B7' is used as a constant, 'ken' is used as a label, and '5' is used as a port number.

```
Arithmetic                Shift and Rotate
  ----------                ----------------

  ADD s1,s2                   SR0 s1
  ADD s1,2B7                  SR1 s1
  SUB s1,s2                   SRX s1
  SUB s1,2B7                  SRA s1
                              RR  s1
  Logical
  -------                     SL0 s1
                              SL1 s1
                              SLX s1
  OR  s1,s2                   SLA s1
  OR  s1,2B7                  RL  s1
  AND s1,s2
  AND s1,2B7                   Jump
  XOR s1,s2                    ----
  XOR s1,2B7
  LD  s1,s2                   JP ken
  LD  s1,2B7                  JP Z,ken
                              JP C,ken
   Port                       JP NZ,ken
   ----                       JP NC,ken
                              JP GT,ken
  IN  s1,5                    JP LT,ken
  OUT s1,5
```

Case sensitivity

Upper and lower cases are accepted. The assembler converts all characters to upper case.

Tabs and Spaces

Tabs and spaces can be used freely to format the program. They are removed during processing.

Constants

Constants are interpreted in hexadecimal only, and hence only characters **`0-9`** and **`A-F`** are valid. The designer must ensure that the data bus width setting for the PSM macro is large enough to support the constants specified in the assembly program.

Registers

The use of a register in an instruction is indicated by the letter '**s**' before the single hexadecimal character 0 to F representing which of the 16 registers is to be used. Most instructions expect the first operand to be a register, but the second operand is assumed to be a constant if '**s**' is not used.

Labels

Labels can use any alpha-numeric combination. Spaces are removed, but the underscore ('_') character can be used as a separator. There is no fundamental

limit to the length of labels, but labels longer than 15 characters make the **compile.log** file untidy.

Jumps

Jumps must be performed using labels. For each label used in a jump instruction, a corresponding label must appear in the program.

Comments

Any characters specified after a semicolon (';') until the end of the line are assumed to be a comment and are ignored. Comments are retained in the **compile.log** file. Any character can be used in a comment, but control characters inserted by some text editors may give unexpected results.

Interesting Ideas and Examples

Following are a few ideas that may help in the use of PSM and this assembler. If you have any more ideas, please send them in.

Labels do not have to be on the same line as an instruction

As seen in the earlier example, labels do not have to be on the same line as an instruction. By placing them on a line with a comment introducing a procedure, programs become very readable.

Example:

```
mult_by_8 : ;multiply the value in S3 by 8
                  SL0 s3
                  SL0 s3
                  SL0 s3
```

which seems to make much more sense than

```
mult_by_8 : SL0 s3 ;multiply the value in S3 by 8
            SL0 s3 ;using a shift to multiply by 2
            SL0 s3 ;three times.
```

Avoid multiple labels at one address

Multiple labels can be defined to a single address location. Although PSMBLE can process them, referencing different labels in jump instructions causes unnecessary jump pointers to be assigned. A review of the **jumpaddr.tab** indicates duplicate addresses.

Take care of Carry flag

ADD, **SUB**, **SRA** and **SLA** all use the carry flag during data processing. If you do not wish the carry flag to have an effect, there are several options:

1. Shift instructions are very flexible, and where possible, you should force a '1' or '0' into the register instead of the carry flag. For example, use '**SL0 s4**' instead of '**SLA s4**' if you definitely want to force a zero into the LSB.
2. Perform any logical function (AND, OR, XOR) before the carry flag operation. All logical functions have the effect of clearing the carry flag; hence by ordering instructions carefully, the desired effect is achieved without wasting instructions.
3. The carry flag can be cleared by using a logical OR of any register with itself. This step preserves data, but may also affects the zero flag, which may or may not be useful. For example, '**OR s4,s4**' clears the carry flag.

Obtaining more constants

If your program uses more than 16 constants, there are several tricks to obtain more.

First, avoid using zero, '**0**', as a constant by clearing any register with the XOR instruction. For example, to effectively load register **s2** with zero, execute:

```
XOR s2,s2
```

It may also be possible to form the constant you need from those you already have and hold it in an unused register. Look at various kinds of instructions to make the value required. The following are some examples of values created from the constants 3 and 5:

Assume

```
LD s3,3
LD s5,5
```

then the following operations creates these new values

```
XOR s3,s3     -> 0
AND s5,s3     -> 1
SUB s5,s3     -> 2
SL0 s3        -> 6
OR  s5,s3     -> 7
ADD s5,s3     -> 8
SL0 s5        -> A
SL1 s5        -> B
```

Finally, use any unused input ports to read an external ROM containing further constants the same way as the internal constant ROM.

PSM does not support a CALL and RETURN system

A manual approach to call and returns functions is possible, but it adds instructions to a program. Decide whether duplicating the subroutine in straight code is smaller than the effect of making the subroutine call.

The suggested method only requires four instructions per call, but also uses up some jump vectors. Remember, the PSM only permits 16 jump vectors in total.

The concept is to load a register before making the 'call' such that the return can be made logically. Sometimes unique data passed to the sub-routine can also be used to indicate the point of return.

Example:

```
call_from_A:  LD sF, 01    ;return flag
              JP sub_routine
return_to_A:  ;continue the program

call_from_B:  LD sF, 02    ;return flag
              JP sub_routine
              return_to_B: ;continue the program

call_from_C:  LD sF, 03    ;return flag
              JP sub_routine
return_to_C:  ;continue the program
sub_routine:  ;instructions to perform sub routine
              SUB SF,01
              JP Z, return_to_A
              SUB SF,01
              JP Z, return_to_B
              JP Z, return_to_C
```

External hardware interacting with PSM

PSM is an imbedded micro-controller, and all the signals are available to be connected to other logic. This means that other 'external' processes can be triggered by the PSM instructions without actually using 'IN' and 'OUT' instructions.

Example:

A program is assembled and a particular process is only activated by a jump to address 34. Clearly, this address will then appear on the 'CURRENT_ADDR' bus. Other hardware can be controlled to operate or stop by decoding Address 34 on 'CURRENT_ADDR'. This technique reduces the number of instructions required and improves performance.

Conclusions

This application note introduces a novel microprocessor macro which can be used in two obvious ways:

- As an imbedded processor in a complex design.
- To convert an FPGA into a microcontroller during production test or field diagnostics.

This application note also demonstrates ways to exploit the architectural features of an XC4000 FPGA. X-BLOX synthesis provides a logical schematic and a simple method of accessing the density and performance of the device.

Finally, Xilinx FPGAs offer total flexibility. This macro may provide a basis for your own custom processor design. The instructions can be adapted to meet your unique system requirements.

Using the PSM Design Files

This design is available on the **Programmable Logic Breakthrough '95** CD-ROM. This section describes what software is required to run the design and the steps involved. Also, please read through the ***Limitations and Restrictions*** section.

Software Requirements

The following software is required to process this design:

- VIEWdraw or VIEWdraw-LCA schematic editor. This software is required in order to make modifications to the schematics.
- Xilinx XACT 5.0 FPGA development system, including the PPR place and route program and the X-BLOX module generator.
- The QBASIC BASIC interpreter, available with MS-DOS, is required to run the PSMBLE assembler.

Using the Design on Your System

1. Create a new directory called **PSM** on your hard disk.
2. Copy the files and sub-directories from the /MISCAPPS/MICROCNT/DESIGNS directory on the Programmable Logic Breakthrough '95 CD-ROM into your **PSM** directory.
3. Edit the **VIEWDRAW.INI** file. Make sure that the VIEWlogic® design library pointers are set appropriately for your machine. You will find the library pointers near the end of the file.

Limitations and Restrictions

WARNING: THIS IS AN UNTESTED DESIGN.

Xilinx, Inc. does not make any representation or warranty regarding this design or any item based on this design. Xilinx disclaims all express and implied warranties, including but not limited to the implied fitness of this design for a particular purpose and freedom from infringement. Without limiting the generality of the foregoing, Xilinx does not make any warranty of any kind that any item developed based on this design, or any portion of it, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country. It is the responsibility of the user to seek licenses for such intellectual property rights were applicable. Xilinx shall not be liable for any damages arising out of or in connection with the use of the design including liability for lost profit, business interruption, or any other damages whatsoever.

Design Support and Feedback

This application note may undergo future revisions and additions. If you would like to be updated with new versions of this application note, or if you have questions, comments, or suggestions please send an E-mail to

apps@xilinx.com

or a FAX addressed to "PSM Application Note Developers" sent to

1+(408) 879-4442.

IMPORTANT: Please be sure to include which version of the application note you are using. The version number is in the lower right-hand corner of page 1.

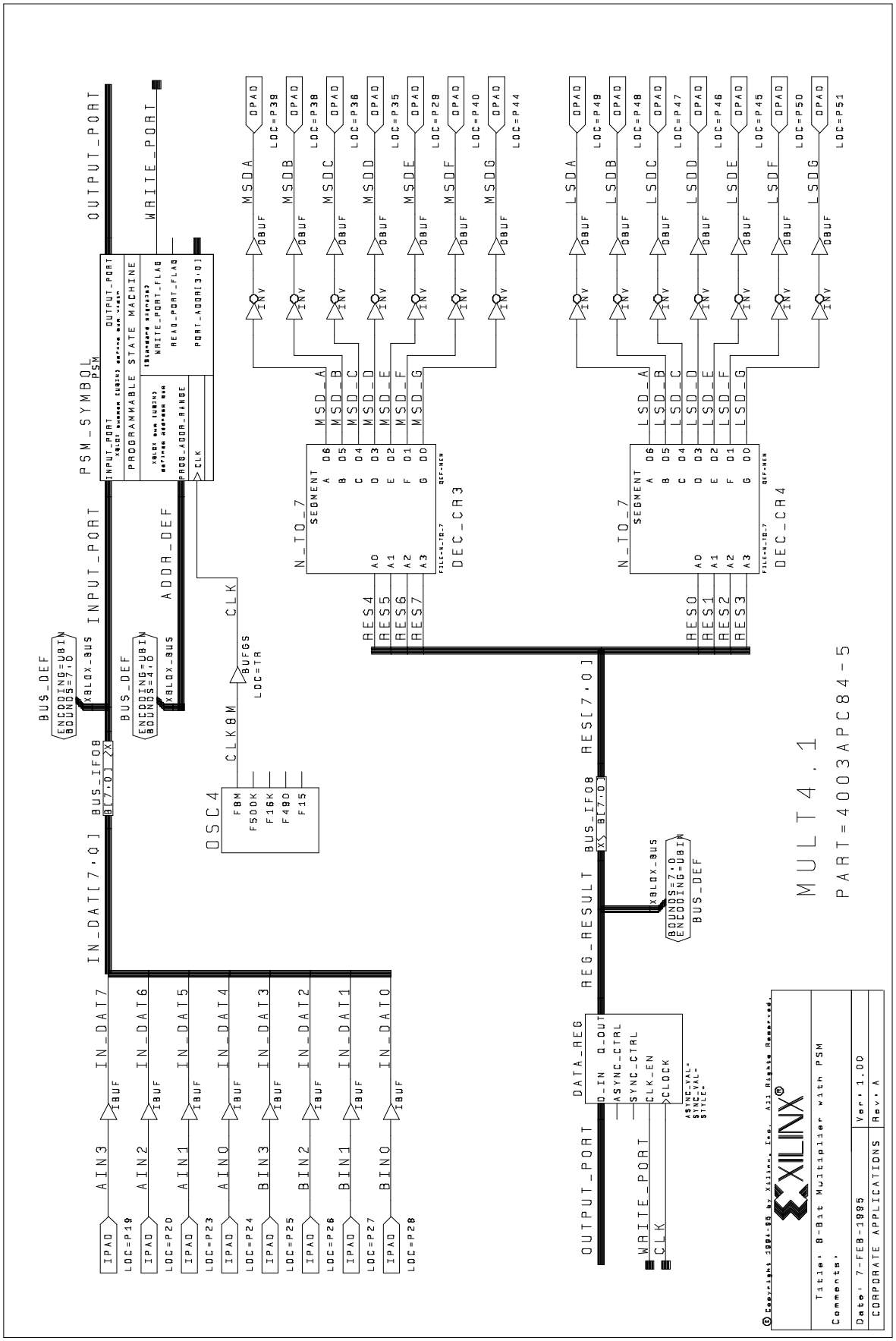

Figure 6. Four-bit multiplier design using PSM macro.

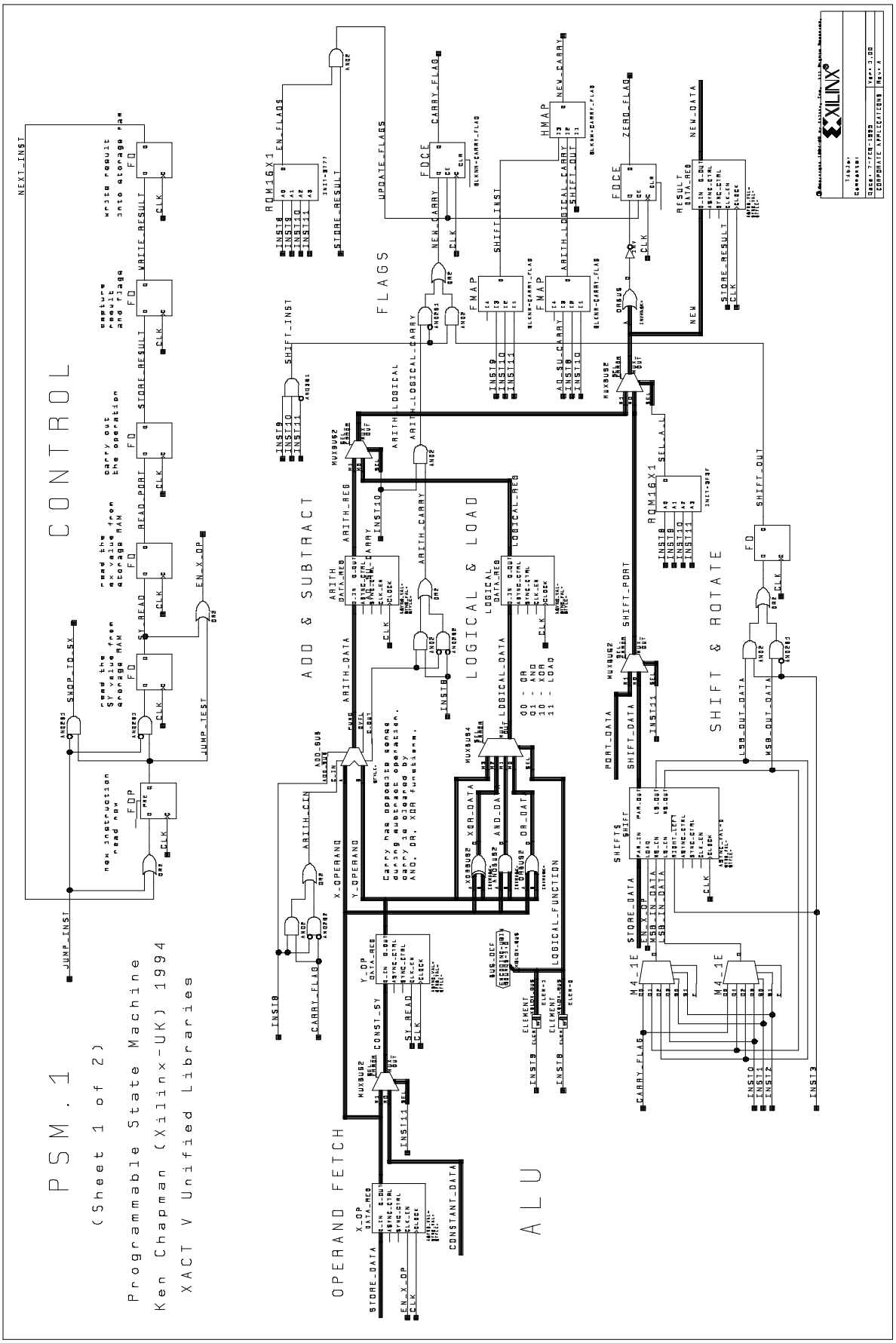

Figure 7. The internal details of the PSM macro. Many portions of the design use X-BLOX.

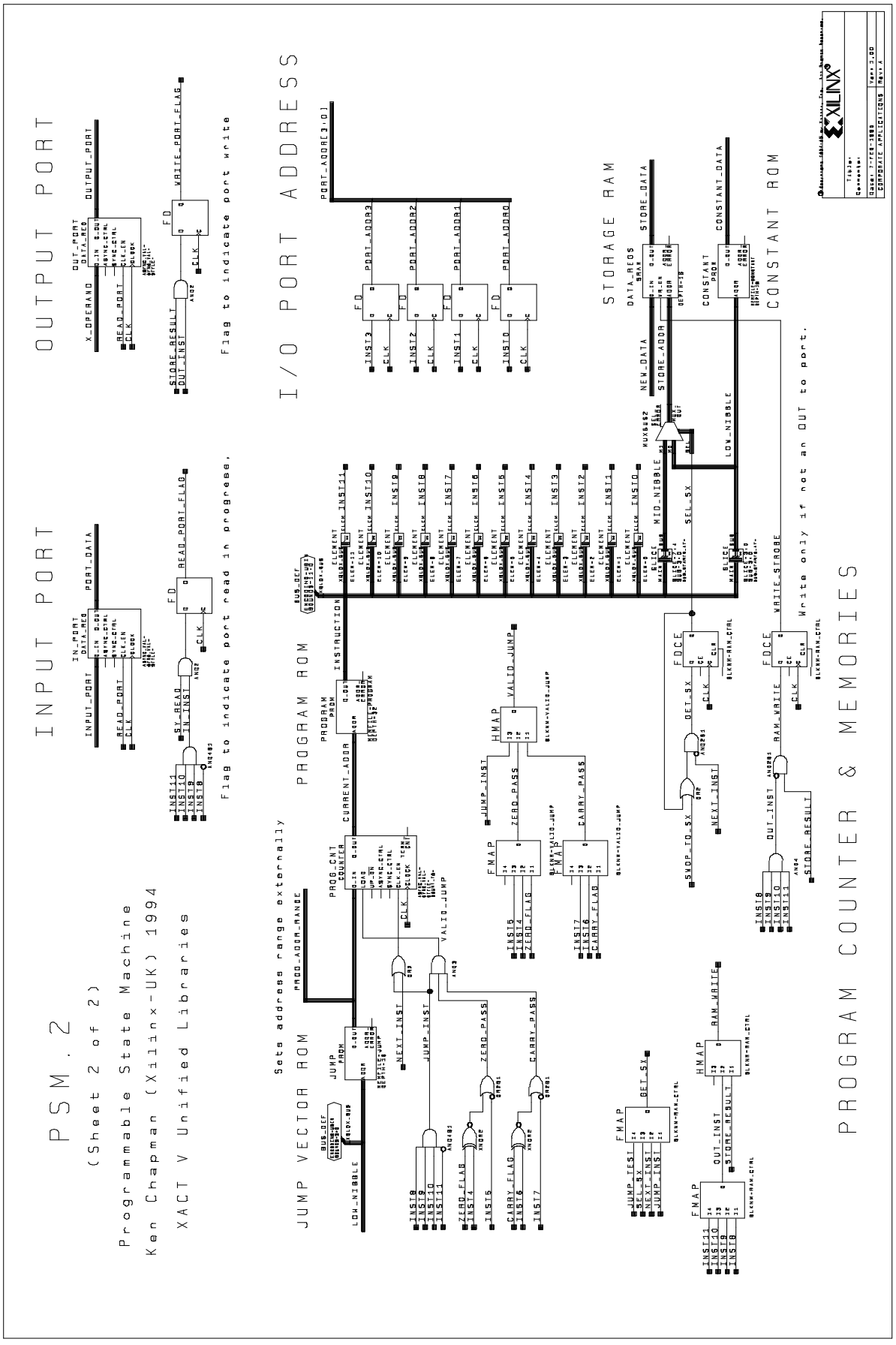

Figure 8. More internal details of the PSM macro. Many portions of the design use X-BLOX.