# Application of ASM++ methodology on the design of a DSP processor

S. de Pablo, S. Cáceres, J.A. Cebrián
University of Valladolid
E.T.S.I.I., Paseo del Cauce, s/n
47011 Valladolid (Spain)
sanpab@eis.uva.es

M. Berrocal
eZono AG
Winzerlaerstrasse 2
07745 Jena (Germany)
manuel@ezono.com

## Abstract

*This article presents the application of a graphical methodology used to develop a Digital Signal Processor designed for FPGA. The instruction set and main features of this processor are introduced. Then, a modified Algorithmic State Machine methodology, named ASM++, is applied to fully describe the processor implementation. This processor has been simulated and physically tested on Xilinx Spartan-3 devices, achieving 37.5~75 MIPS and up to 150 MOPS running at 75 MHz.*

## 1. Introduction

Most intellectual property (IP) modules are designed as synchronous digital circuits using a standard hardware description language (HDL), usually VHDL or Verilog. Designers usually prefer a text-based tool to describe their circuits because editing and managing texts is easier than dealing with the arrangement of schematics. Compared to schematic entry, productivity is increased, mostly when parametrical modules are required.

To assist designers in their daily job, several visual tools have been developed to facilitate the circuit behavior description and understanding, namely Finite State Machines (FSM) and Algorithmic State Machine (ASM) [1], [3]. However, these tools are limited in their scope, so they are applied only on small state machines and circuits.

This paper presents several modifications of standard ASM diagrams with the aim of applying this methodology to design real-life circuits, document them and ease their supervision [8]. As an example, this methodology has been successfully applied in the design of an FPGA based DSP processor.

## 2. The DSPuva18 processor

The DSPuva18 processor is based on the former DSPuva16 [6], a Digital Signal Processor developed for Power Electronic applications [2]. These are the main features of this new processor and their improvements:

- Its computational instructions are executed using two clock cycles, rather than four [6], thanks to the use of an FPGA hardwired multiplier.
- Its control instructions (call, ret, jp, …) are usually executed in one clock cycle.
- It has adaptive conditional jumps and returns: it introduces one or two wait states to leave previous operations to finish.
- The program length can be up to 64K instructions.
- It can execute from 16 to 128 nested subroutines.
- The data memory is up to 64K words, with fast direct and indirect access (two clock cycles).
- It has direct access to 256 ports/devices.
- The instruction set, as shown in table 1, has been designed around 17 basic instructions, but most of these instructions lead to more possibilities.
- It has access to immediate constants in program code to ease filter implementation.
- An implicit access to last port used, with write back capability, has been introduced to speed up filters. It allows up to four operations per instruction.
- The range of fixed-point registers and values can be selected at instantiation time between ±1, ±2, ±4 and ±8. This feature eases in-circuit debugging.

As can be seen, some of these features are common with other processors, but other ones are new. The basic instruction set of this processor is shown below.

**Table 1. DSPuva18 basic instruction set.**

| OpCode | Mnemonic | Function |
|---|---|---|
| 0000 dddd dddd dddd | call <destination-address> | Jump to a subroutine. |
| 0001 dddd dddd dddd | goto <destination-address> | Unconditional jump. |
| 0010 0fff dddd dddd | jp*FLAG* <relative-jump> | Conditional jump. |
| 0010 1fff •••• •••• | ret*FLAG* | Conditional return. |
| 0011 kkkk kkkk kkkk | imm $K_{12}$ | Prepare a constant. |
| 0100 kkkk kkkk nnnn | rN = port($K_8$) | Read from a direct port. |
| 0101 kkkk kkkk nnnn | port($K_8$) = rN | Write to a direct port. |
| 0110 •••• bbbb nnnn | rN = mem({rB,$K_{16}$}) | Read from memory. |

| | | |
|---|---|---|
| 0111 •••• bbbb nnnn | mem({rB,$K_{16}$}) = rN | Write to memory. |
| 1000 sfff bbbb nnnn | if*FLAG* rN = [–]{rB,$K_{16}$} | Conditional assignment. |
| 1001 xxxx bbbb nnnn | rN = *fx*({rN,*LP},{rB,$K_{16}$}) | Extra functions. |
| 1010 nnnn bbbb aaaa | rN = {rA,*LP} + {rB,$K_{16}$} | Addition. |
| 1011 nnnn bbbb aaaa | rN = {rA,*LP} – {rB,$K_{16}$} | Subtraction. |
| 1100 nnnn bbbb aaaa | rN = {rA,*LP} * {rB,$K_{16}$} | Multiply two values. |
| 1101 nnnn bbbb aaaa | rN = – {rA,*LP} * {rB,$K_{16}$} | Multiply and change sign. |
| 1110 nnnn bbbb aaaa | rN += {rA,*LP} * {rB,$K_{16}$} | Positive accumulation. |
| 1111 nnnn bbbb aaaa | rN –= {rA,*LP} * {rB,$K_{16}$} | Negative accumulation. |

This basic instruction set is extended as seen on tables 2 and 3. Additionally, most instructions allow the use of a register ('rB') or a 16-bit constant ('$K_{16}$'), easing constant coefficient filter implementation. This constant is built using four bits of the current instruction and twelve bits of the previously executed 'imm' instruction.

At the same time, a completely new feature has been added: when 'r0' is addressed as register 'rA', the last port used ('*LP') is read, the read value is used instead of r0's value, and then it is written back to the same port. As seen later, this feature speed up the implementation of large filters, requiring just one instruction per tap.

The control instructions of this processor are easy to understand. First of all, 'call' and 'goto' execute an absolute jump to a 4K to 64K address in one clock cycle. As long as only twelve bits are available to give the destination address, its value is multiplied by 1, 2, 4, 8 or 16, depending on the processor model, thus allowing larger programs. Consequently, all subroutines must be aligned to a reachable address, but the assembler can do it easily using the '#align' directive.

Conditional jumps and returns are a bit different (see the eight available conditions on table 2, that shows conditional assignments): they execute their task, but they wait one clock cycle for arithmetic and logic operations to finish, and two clock cycles for multiplications. This way, the use of interleaving 'nop' instructions is avoided. When unconditional 'jp' or 'ret' is used, it is executed in one clock cycle.

The access to external data is fast and flexible. The processor can address up to 256 direct ports, usually related to physical devices or small memories, maybe shared with other FPGA processors. When large amounts of data must be used, the processor implements a dedicated interface enabling the use of synchronous FPGA memories like Xilinx BlockRAM or Altera M4K and M-RAM. It can address up to 64K words per page, and different pages may be selected using a page-register controlled through a port. All these accesses are executed using two clock cycles.

This processor can conditionally load a register with a constant or the value of another register (see table 2), and it also implements more functions as shown in table 3. Right and left shifts are a bit different than expected because most used shifts are the shortest ones, thus using shifts by 7, 3, 2 and 1 rather than 8, 4, 2 and 1 it is on average better. The 'max' and 'min' instructions are also useful, particularly "rN = abs(rN)" is recognized by the assembler and replaced by "rN = max(rN,–rN)". All these instructions use two clock cycles for their execution, like additions and subtractions; their results are immediately available in the following instruction.

The four multiplying instructions, with optional positive or negative accumulation, are executed using only two clock cycles, but the result cannot be used as an operand, except for accumulation, at the following instruction. If required, a one clock 'nop' (an assembler macro replaced by "jp <next-address>") must be added.

**Table 2. Conditional assignments of DSPuva18.**

| OpCode | Mnemonic | Function |
|---|---|---|
| 1000 0000 bbbb nnnn | rN = {rB,$K_{16}$} | Load a register. |
| 1000 0001 bbbb nnnn | ifV rN = {rB,$K_{16}$} | Load if oVerflow. |
| 1000 0010 bbbb nnnn | ifEQ rN = {rB,$K_{16}$} | Load if EQual to 0. |
| 1000 0011 bbbb nnnn | ifNE rN = {rB,$K_{16}$} | Load if Not Equal to 0. |
| 1000 0100 bbbb nnnn | ifGT rN = {rB,$K_{16}$} | Load if Greater Than 0. |
| 1000 0101 bbbb nnnn | ifGE rN = {rB,$K_{16}$} | Load if Greater or Equal. |
| 1000 0110 bbbb nnnn | ifLE rN = {rB,$K_{16}$} | Load if Less or Equal. |
| 1000 0111 bbbb nnnn | ifLT rN = {rB,$K_{16}$} | Load if Less Than 0. |
| 1000 1000 bbbb nnnn | rN = –{rB,$K_{16}$} | Load changing sign. |
| 1000 1001 bbbb nnnn | ifV rN = –{rB,$K_{16}$} | Load if oVerflow. |
| 1000 1010 bbbb nnnn | ifEQ rN = –{rB,$K_{16}$} | Load if EQual to 0. |
| 1000 1011 bbbb nnnn | ifNE rN = –{rB,$K_{16}$} | Load if Not Equal to 0. |
| 1000 1100 bbbb nnnn | ifGT rN = –{rB,$K_{16}$} | Load if Greater Than 0. |
| 1000 1101 bbbb nnnn | ifGE rN = –{rB,$K_{16}$} | Load if Greater or Equal. |
| 1000 1110 bbbb nnnn | ifLE rN = –{rB,$K_{16}$} | Load if Less or Equal. |
| 1000 1111 bbbb nnnn | ifLT rN = –{rB,$K_{16}$} | Load if Less Than 0. |

**Table 3. Extra instructions of DSPuva18.**

| OpCode | Mnemonic | Function |
|---|---|---|
| 1001 0000 bbbb nnnn | rN = rB >> 7 | Right shift seven bits. |
| 1001 0100 bbbb nnnn | rN = rB >> 3 | Right shift three bits. |
| 1001 1000 bbbb nnnn | rN = rB >> 2 | Right shift two bits. |
| 1001 1100 bbbb nnnn | rN = rB >> 1 | Right shift one bit. |
| 1001 0001 bbbb nnnn | rN = rB << 7 | Left shift seven bits. |
| 1001 0101 bbbb nnnn | rN = rB << 3 | Left shift three bits. |
| 1001 1001 bbbb nnnn | rN = rB << 2 | Left shift two bits. |
| 1001 1101 bbbb nnnn | rN = reverse rB | Reverse all bits. |
| 1001 0010 bbbb nnnn | rN = {rN,*LP} and {rB,$K_{16}$} | Logic AND. |
| 1001 0110 bbbb nnnn | rN = {rN,*LP} or {rB,$K_{16}$} | Logic OR. |
| 1001 1010 bbbb nnnn | rN = {rN,*LP} xor {rB,$K_{16}$} | Logic XOR. |

| | | |
|---|---|---|
| 1001 1110 bbbb nnnn | rN = not rB | Logic NOT. |
| 1001 0011 bbbb nnnn | rN = min ({rN,*LP},{rB,$K_{16}$}) | Minimum of two values. |
| 1001 0111 bbbb nnnn | rN = max({rN,*LP},{rB,$K_{16}$}) | Maximum of two values. |
| 1001 1011 bbbb nnnn | rN = min({rN,*LP},–{rB,$K_{16}$}) | Minimum changing sign. |
| 1001 1111 bbbb nnnn | rN = max({rN,*LP},–{rB,$K_{16}$}) | Maximum changing sign. |

A program example that implements an infinite impulse response filter (IIR) is shown below. Most instructions of this filter execute up to four operations: a read from last used port (through '*LP'), a write back of the read value to the same port (so it reads an old sample or output from a FIFO and returns it to the same FIFO for the next filter update), a fixed-point 18x18 product and a positive 32-bit accumulation. This means 37.5 MIPS and 150 MOPS running at 75 MHz.

```
/*
   Demonstration program of DSPuva18 for FPGAworld'2007
   2007/08/27   Santiago de Pablo (sanpab@eis.uva.es)
*/

#model    E              // Programs up to 64K instructions
#range    8              // DSP values between +-8.0
#include  "uva18std.h"   // Several definitions

// IIR filter implementation:
//    Input X values are available at port 200.
//    Output Y values are written at port 201.
//    Old X values are stored in a small FIFO at port 202.
//    Old Y values are stored in a small FIFO at port 203.

   #define IN_X        200
   #define OUT_Y       201
   #define FIFO_X      202
   #define FIFO_Y      203

   #define YC1         0.9345
   // Define also YC2...YC4 and XC0...XC5 constants.

0x0000:                // Programs begins here after reset
        call InitFilter   // Prepare the filter
Loop: call UpdateFilter   // 14 + 2x(NX + NY) clks
        jp Loop           // Infinite loop (2 MSPS at 70 MHz)

#align
InitFilter:
   // First reset FIFO_X and FIFO_Y (not done here)
   // Then load dummy values as old samples
   r1 = 0.0
   port(FIFO_Y) = r1   // Load four values on FIFO_Y:
   port(FIFO_Y) = r1   //   they are y4, y3, y2 & y1.
   port(FIFO_Y) = r1
   port(FIFO_Y) = r1
   port(FIFO_X) = r1   // Load five values on FIFO_X:
   port(FIFO_X) = r1   //   they are x5, x4, x3, x2 & x1.
   port(FIFO_X) = r1
   port(FIFO_X) = r1
   port(FIFO_X) = r1
   ret
```

```
#align
UpdateFilter:
   r2 = port(FIFO_Y)   // Read y4 value (and loose it later)
   r1 =         r2 * YC4 // … and multiply y4 by its coefficient
   r1 = r1 + *LP * YC3 // Get y3 and multiply it by its coefficient
   r1 = r1 + *LP * YC2 // Get y2 and multiply it by its coefficient
   r1 = r1 + *LP * YC1 // Get y1 and multiply it by its coefficient
   r2 = port(FIFO_X)   // Read x5 value (and loose it later)
   r1 = r1 +   r2 * XC5 // … and multiply x5 by its coefficient
   r1 = r1 + *LP * XC4 // Get x4 and multiply it by its coefficient
   r1 = r1 + *LP * XC3 // Get x3 and multiply it by its coefficient
   r1 = r1 + *LP * XC2 // Get x2 and multiply it by its coefficient
   r1 = r1 + *LP * XC1 // Get x1 and multiply it by its coefficient
   r2 = port(IN_X)     // Get a new x0 value (from an A/D?)
   r1 = r1 +   r2 * XC0 // … and multiply x0 by its coefficient
   port(FIFO_X) = r2   // Put x0 value on its FIFO for later use
   port(FIFO_Y) = r1   // Put y0 value on its FIFO for later use
   port(OUT_Y) = r1    // Output of the IIR filter (to a D/A?)
   ret                 // Finish
```

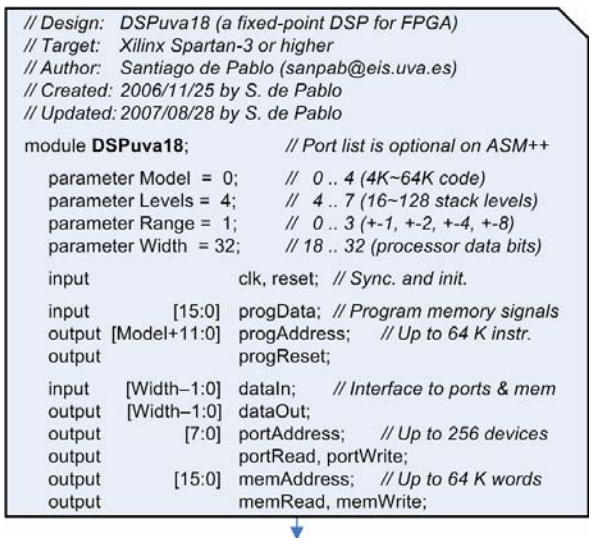
## 3. ASM++ diagram of DSPuva18

The design of this processor has been entirely done using ASM++ diagrams. These diagrams, proposed at [8] and described further here, are an extension of Algorithmic State Machines [1], [3], a methodology used forty years ago for the development of microprocessors. As can be seen with this example, the ASM++ diagrams are now fully capable of describing whole IP modules.

This diagram and the manually generated equivalent code use Verilog 2001, but VHDL may be used instead. An ASM++ compiler that accept standard Verilog and VHDL languages for input and output is in progress.

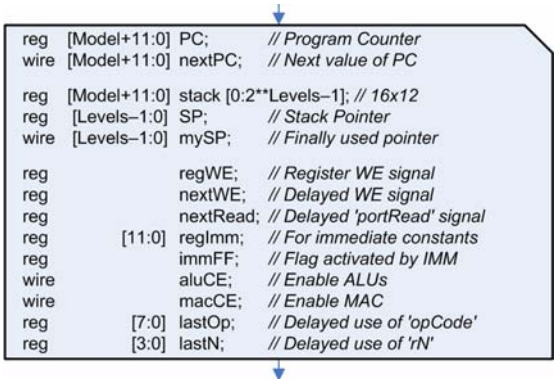
The first ASM++ box of this design, as seen below on Fig. 1, is a "code box", able to introduce Verilog or VHDL code. It is used in this case to describe the processor interface.

**Figure 1. Design header using Verilog.**



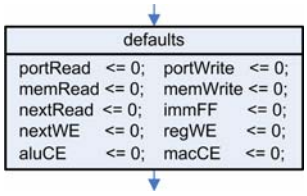
Afterwards, a second code box specifies several internal signals. As long as this box has global meaning, other signals would be and will be declared later.

**Figure 2. Declaration of several signals.**



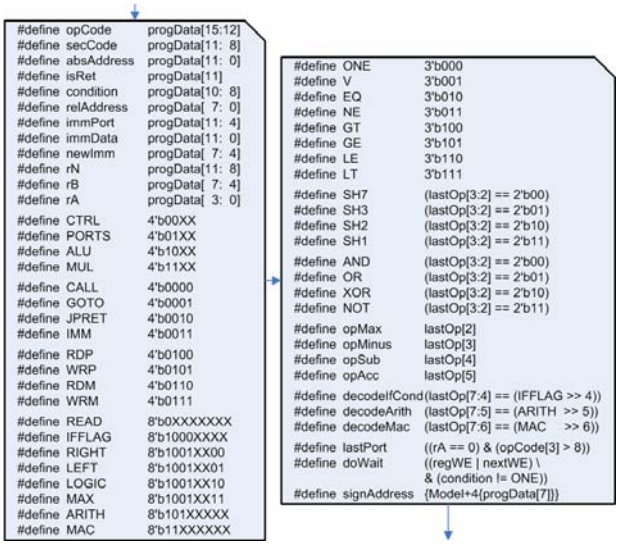

The third box introduces a first difference between ASM++ and the pure code. It specifies global defaults for synchronous and asynchronous internal signals and outputs. If the user does not assign anything to a synchronous signal in a state the default behavior is to *keep* its last value; for an asynchronous signal the compiler must implement a *don't care* logic value. Designer can easily change this default behavior using this box.

**Figure 3. Default values of signals and outputs.**



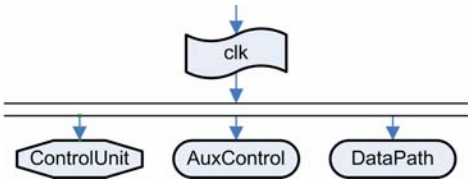

The following two code boxes are a combinational instruction decoder implemented using a C-like "#define" compiler directive. Other directives are also available to include files and other purposes.

**Figure 4. Instruction decoder.**




After all these definitions, a box is used to specify the synchronism of this circuit. In this case there is a unique clock signal, named 'clk', but several clocks may be used instead. Then, three branches are initiated: the first one is a state machine named "ControlUnit"; the second one contains several synchronous and asynchronous components that assist at any time to the previous state machine; the last one is the data path of this processor, also described as an independent thread. Any dependence between branches may be implemented using the name of the state of each thread. This example shows how easily ASM++ diagrams may describe multi-clocked or multi-threaded circuits.

**Figure 5. Parallel circuits description.**



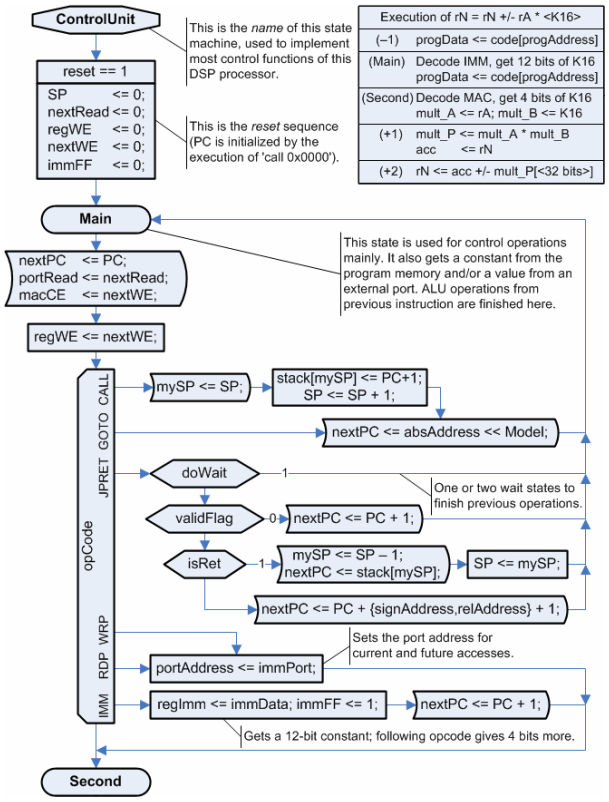

The first branch, which state variable is named 'ControlUnit' as seen on Fig. 6, begins with an asynchronous reset sequence controlled by the active high 'reset' signal. This box increases the ASM possibilities: standard diagrams cannot describe properly reset sequences.

Then, a first state named 'Main', which begins with an oval "state box", executes several overlapped operations from the previous instruction and decodes the current instruction. For 'call', 'goto', 'jp' and 'ret' instructions only one clock is needed, so the next state is 'Main' again; other instructions require a 'Second' state.

Figure 6 shows more ASM++ features:

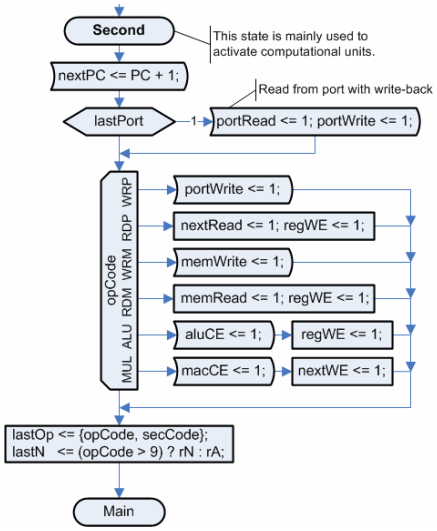
- Synchronous operations, those that are executed when the current clock cycle finishes, like "SP <= SP + 1", are described using a rectangular box anywhere. This is a difference with traditional ASM diagrams, where only unconditional operations use these boxes at the beginning of any clock cycle.
- Asynchronous operations, executed all through the current clock cycle, like "nextPC <= PC + 1", use a box with bent sides. This is a nice feature, that shows the difference in the behavior between synchronous and asynchronous signals. When Verilog language is used, the equal operator ('=') may also be used for asynchronous assertions.
- Conditions are expressed in the same way than standard ASM diagrams, but also multiple output decisions are included.
- The use of VHDL/Verilog expressions allows an easy implementation of complex functions, like a register file or a returning address stack, that need vector notation.

**Figure 6. Processor control unit (I).**




The following state named 'Second', seen at Fig. 7, executes all computational instructions after receiving operands from the previous clock cycle. Actually, this state just activates all the required control signals, because data path and external devices do the real job.

**Figure 7. Processor control unit (II).**

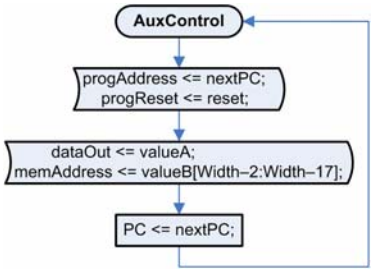



Readers are kindly invited to translate this state machine to HDL code[1], either using VHDL or Verilog. Then, the relationship between ASM++ and HDL arises, and the advantages of using a graphical tool to design and/or document complex circuits also becomes clear.

To complete control tasks a second thread is more than convenient (see Fig. 8). Several operations must be done during or at the end of *all* clock cycles. Writing these operations in the previous thread is at least uncomfortable and prone to mistakes. Real life circuits require the possibility of writing parallel threads, but standard ASM diagrams cannot do it.

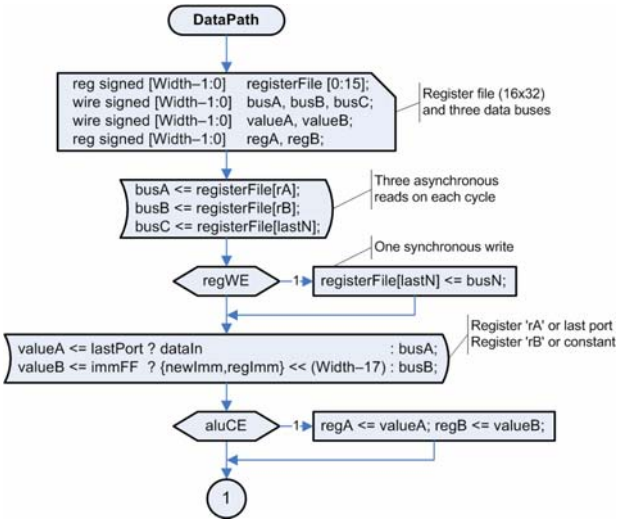
A second detail of Fig. 8 is that, from the point of view of the 'PC' signal, this is a *state-less* state machine: it needs no state at all because it has just one state. Additionally, the only reference to a clock here is the rectangular box used for 'PC'; in absence of it, this could be a *clock-less* thread, a pure-combinational circuit properly described using ASM++ diagrams.

**Figure 8. Processor control unit (III).**




Following figures, from 9 to 13, implement the data path of this processor. First of all, a register file keeps the 32-bit values of r0 to r15 registers. Its design is based on two dual-ported distributed memories, allowing up to four asynchronous reads and one synchronous write on every clock cycle; only three reads are actually needed. During the state 'Second', if 'aluCE' signal is asserted, two operands are stored at register 'regA' and 'regB' for their operation during the following 'Main' state.

**Figure 9. Processor data path (I).**

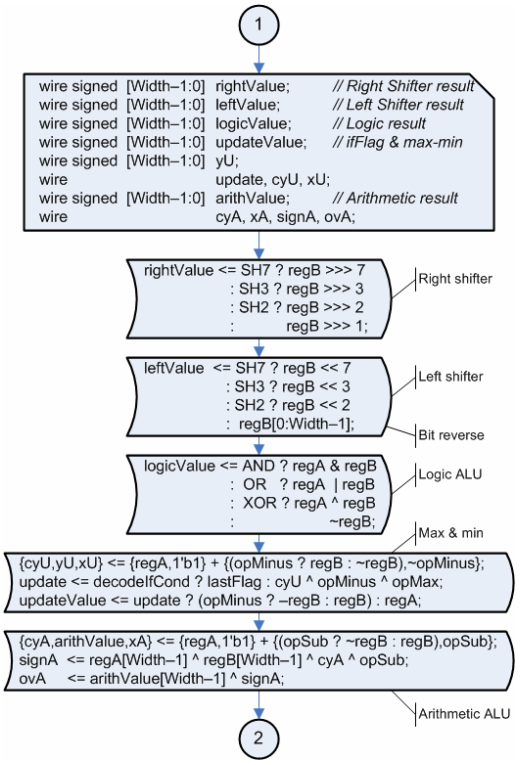



After operand selection, several computational units calculate different results throughout the clock period: a

[1] During the translation process, at least two processes or always blocks are needed, one of them for all *clk-dependent* synchronous operations and the other one, unconnected from the former, for the asynchronous operations. ASM++ diagrams join both worlds.

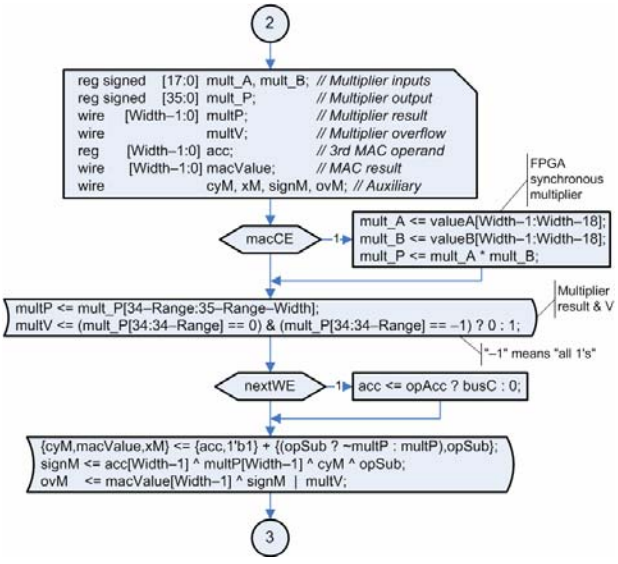
right or left shifted value, a logic or arithmetic result [4], [7], and an update value used for conditional assignments and maximum and minimum evaluation.

**Figure 10. Processor data path (II).**




The core of this processor, a fixed-point 18x18 multiplier with 32-bit result, is described below in such a way that most synthesis tools infer a wired synchronous multiplier: it registers two operands during one clock cycle and gives the product of them at the end of the following cycle. This segmentation stage introduces a one clock latency, so a 'nop' or any dummy instruction must be used before retrieving the product result.

**Figure 11. Processor data path (III).**




When all partial results are available, they are multiplexed in order to store the final value in the register file and to update flags. In these diagrams, it is not important if a signal like 'busN' has been used *before* its declaration (see Figs. 9 and 12).

**Figure 12. Processor data path (IV).**

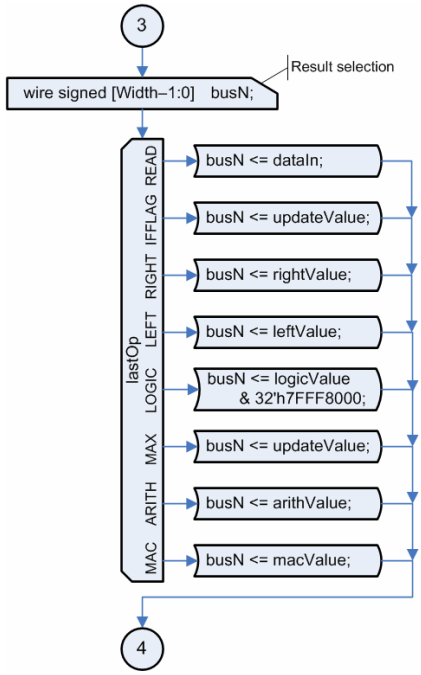
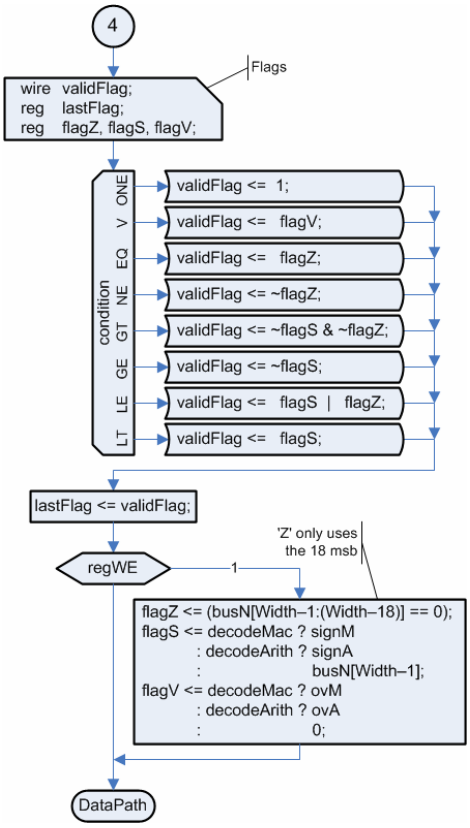



**Figure 13. Processor data path (V).**



## 4. Conclusions

This article has presented a small and easy to understand digital signal processor developed using Verilog and ASM++ diagrams for FPGA. Throughout this paper, the capabilities of ASM++ for the development and documentation of IP modules has arisen. Additionally, supervision of complex designs would be ease when using this methodology. Compared with classic HDL description, the learning curve of ASM++ is shorter and the possibility of mixing synchronous and asynchronous signals is also a great advantage.

The proposed DSP processor executes all its instructions in one or two clock cycles, achieving up to 150 MOPS at 75 MHz on Xilinx Spartan3 devices. It introduces several new features: a variable code length between 4K and 64K, a variable range at implementation time between ±1 and ±8 for numerical values, a transparent access to constants and a built-in read with write back capability to speed up filter implementation. This processor is currently been used in power electronics applications.

## 5. Acknowledgments

The authors would like to acknowledge the partial financial support of *eZono AG* at Jena, Germany, *ISEND SA* at Boecillo, Valladolid, Spain, and the regional government, *Junta de Castilla y León*, under grants VA004B06 and VA021B06.

## References

[1] C.R. Clare, *Designing Logic Using State Machines*, McGraw-Hill, 1973. Referenced by [5].

[2] epYme workgroup, online at *http://www.dte.eis.uva.es/epYme*, last updated on August 2007.

[3] D.D. Gajski, *Principles of Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1997.

[4] J. Gray, "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip", *DesignCon'2001*, online at *http://www.fpgacpu.org/gr/index.html*, 2001.

[5] S. Leibson, "The NMOS II Hybrid Microprocessor: Fusing silicon, ceramic, and aluminium with rubber baby buggy bumpers", online at *http://www.hp9825.com/html/hybrid_microprocessor.html*, revised on August 2007.

[6] S. de Pablo et al., "A soft fixed-point Digital Signal Processor applied in Power Electronics", *FPGAworld Conference 2005*, Stockholm, Sweden, 2005.

[7] S. de Pablo et al., "A very simple 8-bit RISC processor for FPGA", *FPGAworld Conference 2006*, Stockholm, Sweden, 2006.

[8] S. de Pablo et al., "A proposal for ASM++ diagrams", *10th Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2007)*, Kraków, Poland, 2007.