Embedded Microcontrollers and FPGAs Soft-cores

Daniel Francisco Gómez Prado

Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, USA

ABSTRACT: The FPGA's soft-cores main idea is to provide designers with the flexibility of creating a perfect fit in terms of processor(s)[†], peripherals and memory interfaces for embedded applications, this perfect fit usually, but not always, can imply a tradeoff with performance and cost. This paper presents a comparison in speed, power, flexibility and cost between a microcontroller and its soft-core version. For this, an HDL synthesizable soft-core of an 8 bit microcontroller capable of executing the same assembler code of a middle range Microchip PIC, as the 16F84, is performed. The delays introduced by the FPGA interconnect is considered after synthesis by performing post placement & routing simulations.

I. INTRODUCTION

Since 1980, when Intel designed the 8051, an 8-bit microcontroller, embedded systems have used microcontrollers as a core part of their system. The applications in which they have been used came from automotive, industrial control, office automation and communications, to name a few; in general any application that needs fast time to market, lower total system cost and low-risk product development have been designed using a microcontroller; thus promoting the use of microcontrollers everywhere.

This market was particularly understood by Microchip Company which upon foundation in 1989 released an 8 bit one time programmable (OTP) and a reprogrammable (Flash) microcontroller based on a modified Harvard RISC (Reduced Instruction Set Computing) architecture. This simple architecture combined with a reprogrammable capability provided embedded designers with even faster time to market and lower cost systems; which in turn made Microchip grew in the market share of 8 bit microcontrollers (based on worldwide unit shipments) from the 20th place in 1990 to the number one in 2002 [3].

Even though Microchip core architecture has remained unchanged, Microchip now offers more than 180 PIC devices [6] featuring numerous on-chip peripherals to fit best the needs of the huge spectrum of embedded applications in which they are used.

The customization of the instruction set for a given application has resulted in literally hundreds of microcontrollers available today, not just from Microchip but from different companies; each one with a different set of peripherals, memories, interfaces, and performance characteristics. Being one of the biggest challenges faced by embedded designers the selection of a processor that fits best their application requirements; although, designers usually end up either buying more processor than they need to get the right mix of peripherals and interfaces, or settling for a less than ideal solution to keep costs down.

This wide variety of microcontroller peripherals and memories have been understood lately by major FPGAs companies such as Xilinx, Altera and Actel which have developed softcores to embed microcontrollers into their FPGAs, microBlaze [10], Nios [2] and core8051 [1] respectively. Xilinx and Altera provide different kind of configurations, implementing softcore microcontroller from 4 bits to 32 bits, each with different add ons inside the FPGA to handle a variety of peripherals as USBs, UARTs, LCDs, Ethernet, DMA, etc.

II. PREVIOUS WORK AND OVERVIEW

Even though the design of Microchip devices in FPGAs have been successfully done in [4][5][8] and [9], none of them really fully implemented the same architecture. For example the configuration bits on the

[†] One or more microprocessors can be embedded into the same FPGA

The research reported in this paper has been supported in part by the National Science Foundation, contract No. CCR-0204146.

TRIS registers that sets the data direction on the ports of the microcontroller does not work as specified in [7] in none of the designs previously mentioned.

In our microcontroller softcore, referred from now on as UMASScore, the TRIS registers are implemented and the bidirectional ports works as in [7]. The power save mode, implemented via the SLEEP instruction, adds a control signal that stacks the phase clock of the UMASScore. This keeps all registers with the same value and disables the arithmetic unit, thus eliminating the dynamic power dissipation. The interruptions (IRQ) are implemented by a process that reads every clock cycle one of the bits of the input ports configured as IRQ. This process can set a flag that will tell the fetch unit to save the PC on the stack and to jump to a known ROM location in which the IRQs are handled. Additionally, to allow the UMASScore embedded microcontroller interacts with the rest of the FPGA, two instructions are added to the original instruction set: EXTWR and EXTRD. These instructions define an extension module for the UMASScore, so more functionality can be added to the design without changing its primary specification. The concept of expansion module has been successfully used to customize the instruction set provided by NIOS II [2] in which the expanded operations are added using multiplexes as part of the ALU instruction set. In our design the expansion module is going to use the concept of program active memory as presented in [13] so the expansion module will be accessed as a memory block rather than as ALU operation.

The main features in which UMASScore differ from the PIC16F84 and other previous implementations are summarized in the table I.

TABLE I

Feature	Microchip 16F84A	If in any [4][5][8][9]	UMASScore
Oscillator	Several oscillator options	Direct clock input	Direct clock input
Clocking	4 phased clock	Varies (1-4 phased clock)	4 phased clock
Reset	Active low MRST and a power-up circuit	Low MRST and high Reset	Low MRST, no power-up circuit
Sleep	Sleep instruction and circuitry	None	Sleep instruction
Tri-stable Ports	Bidirectional ports programmed by the TRIS register	None	Bidirectional ports as in the 16F84
Watchdog timer	WDT circuit	Done in [8]	WDT circuit
Timer0	Free running or external source	Free running	Free running
Interruption	Multiple and programmable IRQ, with priorities and configurable to rising or falling edge.	Dedicated pin for IRQ	Multiple and programmable IRQ
Extended Inst Set	None	Done in [9], also [1][2][10]	PAM oriented [13]

This project uses Xilinx ISE 6.3i webpack and Xilinx ModelSim II v5.8c software to synthesize, place, route and simulate the VHDL design; and a spartan3 device as the target architecture. The MPASM programming language and its compiler MPLAB IDE v6.62 from Microchip is used to write the microcontroller programs. In [8] and [9] a program to convert from HEX to VHDL has been done, so the assembler code written for the PIC16F84 can be compiled, and the hexadecimal file obtained can be translated into a VHDL format that contains the binary instructions to add to the UMASScore ROM module to simulate and test the design.

It is worth mentioning that the two extended operations EXTWR and EXTRD are not part of the MPASM code and not compatible with the MPLAB IDE software; therefore these two instructions are tested by adding their binary code manually into the program ROM module.

III. DESIGN HIERARCHY

The design of the UMASScore is done hierarchically as in [9]; by breaking the design into five modules: the Rom module, the Ram module, the ALU module, the CPU module and the expansion module.

A. The ROM module

This module implements the program memory of the UMASScore. It contains the binary code with the instructions to be executed. The ROM module takes as input the 13 bits of the program counter PC, and gives as output the 14 bits needed to encode a single instruction in the PIC16F84. See Fig. 1.

To test the instruction set, conditional and unconditional jumps and the proper function of the bidirectional ports, a test program in assembler MPASM language [7] is downloaded into the UMASScore ROM module.

With the 13 bits PC this module can store up to 8K instructions each one of 14 bits width, but this does not mean that a 8Kx14 bits memory is implemented on the design, the number of registers inferred on the module are as many as the number of instructions to be executed; the previous program for example will infer a 65 registers of 14 bits each.

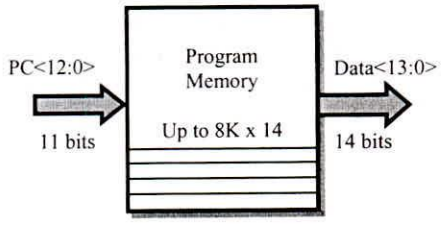

Fig. 1. The block diagram of the ROM module.

B. The RAM module

From the 14 bits instruction retrieved from the ROM module, the 7 least significant bits <6:0> are used to address the RAM memory. This memory of 8 bits width stores the 128 general purpose registers of the UMASScore, from addresses 00h to 7Fh. Three signals control the operation of the RAM: the clock, the general enable and the write enable; and they are used to read the data at the beginning of an instruction execution and to store the result in the desired address. See Fig. 2.

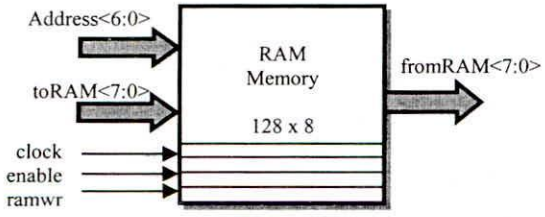

Fig. 2. The block diagram of the RAM module.

The RAM module is written in VHDL according to the coding style suggested in [11] [12] so the synthesizer is able to infer that the block RAM available in the Spartan3 device is going to be used. Respecting the coding style is important otherwise the synthesizer is not able to infer that the device block memory can be used for this module and the memory would be implemented using LUTs.

C. The ALU module:

The UMASScore has a very simple ALU capable of doing arithmetic, logic, shift and bitwise operations as in [7]. It takes as inputs two 8 bits operands, A and B, and 4 bits operation selection; and it produces an 8 bits result with a carry out. There is an additional output used to indicate that the result of the ALU module is zero. See table II.

TABLE II

Opcode	Operation	Description
0000	Addition	A + B
0001	Substraction	A – B
0010	AND	A AND B
0011	OR	A OR B
0100	XOR	A XOR B
0101	Complement	NOT A
0110	Shift right	{Cin, A[7:1]} , Cout <= A[0]
0111	Shif left	{A[6:0], Cin} , Cout <= A[7]
1000	Swap	{A[3:0], A[7:4]}
1001	Clear bit	(Not BitMask) AND A
1010	Set bit	BitMask OR A
1011	Compare bit with 0	If (BitMask AND A) = 0 => Zout = 1
1100	Compare bit with 1	If (BitMask AND A) /= 0 => Zout = 1
1101	Not defined	Propagate A
1110	Not defined	Propagate A
1111	Not defined	Propagate A

In the case of bitwise operations, the 3 more significant bits of operand B are decoded into a BitMask which select the bit of operand A in which the operation takes place.

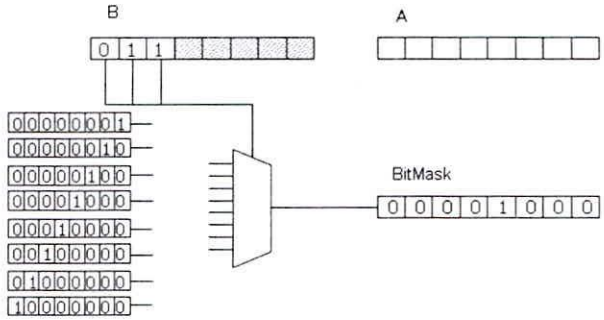

Fig.3. Bitmask selection

The operation to perform in the ALU is determined on the CPU module when decoding the instruction to execute. Once the ALU operation is determined the operands for the ALU A and B are selected.

D. The CPU module

This is the top level hierarchy of the UMASScore and it implements the program flow. Therefore the instruction decode, the specific purpose registers, the data and address buses, the multiplexers and the decoders are implemented here.

The instruction cycle of the UMASScore is divided into a four phased clock that is used to provide synchronization between the different execution steps of an operation. These synchronizations are scheduled in the table III.

TABLE III

Q1		Q2		Q3	Q4
Decode instruction register and determine the ALU op	Update W	Update Op A	ALU	Update Wnext	Write RAM
Read RAM and Special Purpose Registers, read Ports.	Update RegF	Update Op B		Update ToRAM	Write Special Registers
Increment PC if not stalled		Retrieve next instruction		Stall if actual Instruction requires it	Update the Instruction register

The operation of the CPU module can be divided into three functions: The program counter control, the instruction decode and the datapath control.

1) The Program counter control

Upon reset or at start the program counter is loaded with the address 1FFFh and a NOP instruction is forced

into the UMASScore; thus the instruction in 1FFFh is not executed. With this address in the PC the next instruction to be fetched is in 0000h so in the next 4 phased cycle the instruction in 0000h has been stored in the instruction register; and at the beginning of phase Q1, as shown below, the instruction is ready to be decoded and executed.

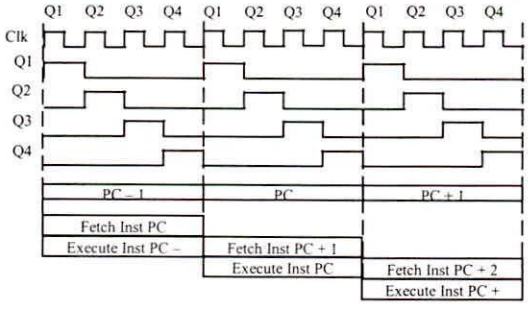

Fig. 4. The instruction fetch

The control of the PC also determines if the instruction in execution produces a conditional or an unconditional branch.

For conditional branches, instructions BTFSC and BTFSS, the zero flag from the ALU module is checked and if the condition is satisfied the next instruction is replaced by the NOP instruction. In this two instructions the PC continues is normal count, only the next instruction to be executed is changed with the NOP instruction if needed. For example in the following instructions if the RAM address 0x16 has the value 58h:

```
20  DECFSZ    0x16,1    //NoJump (57 h)
21  ANDLW     0x6C      //W=20 h
22  BTFSS     0x16,1    //Jump
23  IORLW     0x9F      //W=20 h
```

The instruction DECFSZ will decrement by one that value and store it, as the result is 57h different than 00h the next instruction ANDLW is executed, BTFSS check if the last bit in the address 0x16 is 1, as that value is 57h, the condition is satisfied and the next instruction IORLW is changed by a NOP instruction. This is shown in the next table 3:

TABLE III

Fetch 21	Fetch 22	Fetch 23	Fetch 24
PC = 20	PC = 21	PC = 22	PC = 23
DECFSZ	ANDLW	BTFSS	NOP

For unconditional branches, instructions CALL, GOTO, RETLW and RETURM, the next instruction is replaced by a NOP and the PC is loaded with the destination address. Actually the PC is loaded with the destination address – 1; so the fetch unit, that retrieves the instruction PC + 1, retrieves the instruction in destination address while the NOP instruction is still being executed. For example in the following instructions:

```
49  CALL      subroutine      //Jump
50  BSF       STATUS,RP0      //<0x03>=
----------
subroutine:
384 MOVLW     0x04            //W=04 h
385 MOVWF     0x20            //<0x20>=04 h
386 inner_loop
387 DECFSZ    0x20,1          //<0x20>=03 h
388 GOTO      inner_loop
389 RETLW     0x77            //W=77 h
```

The instruction CALL forces a NOP instruction to be executed instead of BSF and in the meantime it loads the PC with the address 383, which is used for the fetching unit to retrieve the instruction MOLW, instructions 384 and 385 write into the RAM address 0x20 the value 04h. The instruction DECFSZ decrements the value in RAM 0x20 by 1, and checks if it is 00h, as it is not the next instruction GOTO is executed. The GOTO instruction loads in the PC 385, and forces a NOP meanwhile the fetch unit retrieves the instruction from 385, DECFSZ. This process goes on until the value stored in the RAM 0x20 is 01h, a decrement here will produce 00h which will force a NOP instruction to replace the GOTO instruction. The instruction RETLW instruction will be executed then loading the PC with the address 49 and forcing a NOP instruction, this will make the fetch unit to retrieve the instruction BSF from the address 50. This is shown in the table IV.

TABLE IV

Fetch 50	Fetch 384	Fetch 385	Fetch 386	Fetch 387	Fetch 388	Fetch 386
PC 49	PC 50	PC 384	PC 385	PC 386	PC 387	PC 388
CALL	NOP	MOVLW	MOVWF	DECFSZ	GOTO	NOP
			0x20=04h	0x20=03h		

Fetch 387	Fetch 388	Fetch 386	Fetch 387	Fetch 388	Fetch 386	Fetch 387
PC 386	PC 387	PC 388	PC 386	PC 387	PC 388	PC 386
DECFZ	GOTO	NOP	DECFSZ	GOTO	NOP	DECFSZ
0x20=2h			0x20=1h			0x20=0h

Fetch 388	Fetch 389	Fetch 50	Fetch 51
PC = 387	PC = 388	PC = 389	PC = 50
NOP	RETLW	NOP	BSF

2) The Instruction decode

The instruction set supported by the UMASScore is described detailed in [7], and its summary is shown below

In the table V, C and Z denote the carry and zero status bits modified by the ALU module; the File register F represents any position in the RAM, the register W is an internal working register not directly addressable that accumulates the last computation of the ALU, and

K is any constant value passed together with the instruction.
The 14 bits of binary code retrieved from the ROM module has the following formats:

TABLE V

NEMONTECNIC CODE	INSTRUCTION	BINARY CODE	FLAG
Byte oriented operations			
ADDWF F,d	W + F	00 0111 dfff ffff	C, Z
ANDWF F,d	W AND F	00 0101 dfff ffff	Z
CLRF F	Clean register F	00 0001 1fff ffff	Z
CLRW	Clean register W	00 0001 0xxx xxxx	Z
COMF F,d	Complement F	00 1001 dfff ffff	Z
DECF F,d	Decrease F by 1	00 0011 dfff ffff	Z
DECFSZ F,d	Decrease F by 1, skip if 0	00 1011 dfff ffff	
INCF F,d	Increase F by 1	00 1010 dfff ffff	Z
INCFSZ F,d	Increase F by 1, skip if 0	00 1111 dfff ffff	
IORWF F,d	W OR F	00 0100 dfff ffff	Z
MOVF F,d	Move F	00 1000 dfff ffff	Z
MOVWF F,d	Move W to F	00 0000 1fff ffff	
NOP	No operation	00 0000 0000 0000	
RLF F,d	Shift F to the left through C	00 1101 dfff ffff	C
RRF F,d	Shift F to the right through C	00 1100 dfff ffff	C
SUBWF F,d	F – W	00 0010 dfff ffff	C, Z
SWAPF F,d	Swap nibbles in F	00 1110 dfff ffff	
XORWF F,d	W XOR F	00 0110 dfff ffff	Z
Bit oriented operations			
BCF F,b	Clean bit b of F	01 00bb bfff ffff	
BSF F,b	Set bit b of F	01 01bb bfff ffff	
BTFSC F,b	Check bit b of F, skip if 0	01 10bb bfff ffff	
BTFSS F,b	Check bit b of F, skip if 1	01 11bb bfff ffff	
Literal or Control operations			
ADDLW K	K + W => W	11 111x kkkk kkkk	C, Z
ANDLW K	K AND W	11 1001 kkkk kkkk	Z
CALL ExtendeK	Call to subroutine	10 0kkk kkkk kkkk	
CLRWDT	*No implemented (ignored)*	00 0000 0110 0100	
GOTO ExtendeK	Go to ExtendeK	10 1kkk kkkk kkkk	
IORLW K	K OR W	11 1000 kkkk kkkk	Z
MOVLW K	K => W	11 00xx kkkk kkkk	
RETFIE	*No implemented (Ignored)*	00 0000 0000 1001	
RETLW K	Return from subroutine with K => W	11 011x kkkk kkkk	
RETURM	Return from subroutine	00 0000 0000 1000	
SLEEP	*No implemented (ignored)*	00 0000 0110 0011	
SUBLW W, K	K – W => W	11 110x kkkk kkkk	C, Z
XORLW W, K	K XOR W	11 1010 kkkk kkkk	Z
Extended operations			
EXTWR F	Write to address in [F] value W	11 0100 1fff ffff	
EXTRD F	Read from address in [F] and save in W	11 0101 1fff ffff	

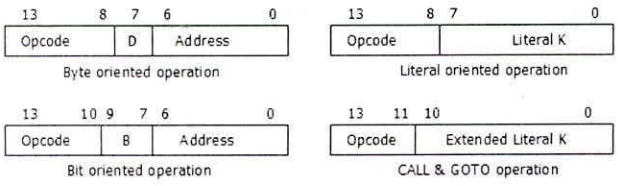

Fig. 5. The instruction format.

The more significant bits are compared and the instruction to be executes is determined. Each instruction correspond to a state in a finite state machine and depending on which state is reached the control signal of the datapath will vary.

3) *The Datapath control*

The control of the datapath can be further divided in different processes, each one synchronized to a phase of the 4 phased clock.

a) *The Working register and the File register:*

As mentioned before the file register represents any position in the RAM and the working register is an internal register use to accumulate the output of the ALU. When we start the execution of a byte or bit oriented instructions, phase Q1 of the 4 phased clock, the working register, W, needs to be updated with the value accumulated in the previous instruction and held in Wnext. And similarly the memory needs to be accessed to update the value of the file register.

As the first 12 bytes of the memory correspond to special registers implemented on the same CPU module and not in the RAM, the file register will be updated with the values from the special registers when the address is less or equal to 0Ch and with the values from the RAM when the address is from 0Dh to 7Fh. See figure 6.

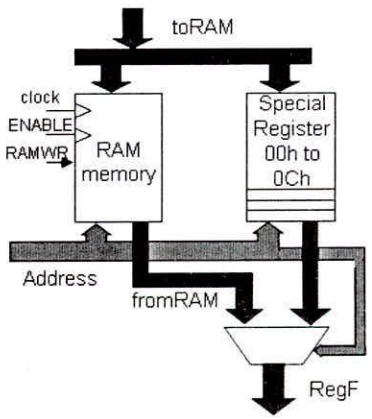

Fig. 6. The working and file register.

At the end of the execution of byte or bit operation, phase Q4, the file register might need to be saved. If so the RAM is written with the value coming from the ALU module or from the file register.

These two operations of reading and writing the RAM are controlled by the signals ENABLE and

RAMWR. When ENABLE is set to 1 and there is a rising edge of the clock, the RAM outputs the value indexed by the address bus; and, when ENABLE is 1, RAMWR is 1 and there is a rising edge of the clock, the data placed in the bus toRAM is stored.

If we need to read and write on the phases Q1 and Q4, then the signal ENABLE is set to 1 during the phases Q3 and Q4, and the signal RAMWR is set to 1 during Q3. For example, if we need to write a value in memory, we set ENABLE and RAMWR both to 1 during phase Q3, and in the next rising edge of the clock the RAM will be written. This is because phase Q4 will start after the rising edge of the clock as it is derived from the main clock and therefore some delay is associated to them, so after RAMWR and ENABLE are set in Q3 the rising edge of the clock will produce the memory to store the value, so it will look as the memory is being written in the beginning of the phase Q4.

b) The Input/Output Ports:

The UMASScore interacts with the external device through the bidirectional ports A and B. These ports are addressed as part of the special registers at addresses 0x05 and 0x06 respectively, and the direction of each bit is set in the registers TRISA and TRISB at addresses 0x05 and 0x06. The UMASScore differentiates which register is being accessed TRISA or PORTA by checking the status of the bit RP0, fifth bit of the register STATUS in address 0x03. If RP0 is set to 1 then the registers TRIS are accessed, otherwise the registers PORT are accessed. This is important because modifying the bits on the register TRIS the respective bits of the register PORT are configured; thus a value of 1 in the register TRIS configures the same bit in its PORT as input, and a value of 0 as output.

c) The ALU operation:

The instruction to be executed has been fetched and decoded in the last phase Q4 of the previous instruction. With the instruction to execute determined, the operation to perform in the ALU module can be determined in phase Q1 meanwhile the working register and the file register are being updated; and in phase Q2 the operands for the ALU are selected according to the operation to perform. See figure7.

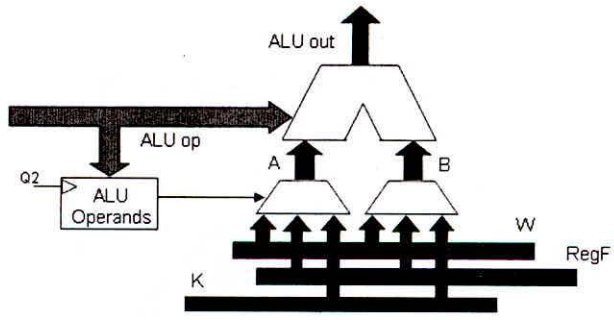

Fig. 7. The ALU unit.

The final architecture implemented in the CPU module is shown below. See figure8.

E. The Expansion module

The two instructions EXTWR and EXTRD added to the instruction set allow to indirect address an expanded memory. This memory is thought as an active memory, so any additional functionality can be added inside the FPGA to the UMASScore.

For both instructions the register F is used to indirectly address an expanded memory, that is, the value store in F is used as address; and the register W is used to connect the expanded data bus. This expanded memory can map up to 256 bytes and computation can take place in this memory as described in the PAM architecture paper [13].

To test this idea a simple fixed point 4 bit multiplier is implemented. The following assembler lines are used to test the proper operation of the expanded instructions.

```
57  MOVWF   0x31    //0x31=C5 h
58  EXTWR   0x31    //0xC5= W[7:4]xW[3:0]
59  CLRW            //W=00 h
60  EXTRD   0x31    //W=3Ch
61  MOVWF   0x31    //0x31=3Ch
```

Assuming the working register has a value of C5h, the instruction MOVWF loads that value on the register 0x31 of the RAM; then the execution of EXTWR address the extended memory with the content of the register file 0x31, and sends the value of W=C5h. As this memory is an active memory, before storing the value some computation takes place, and in our example this computation is the multiplication of W[7:4] with W[3:0]. So the value stored in the extended memory 0xC5 is actually C×5 = 3Ch, and this is the value that is retrieved and send to the working register with the instruction EXTRD.

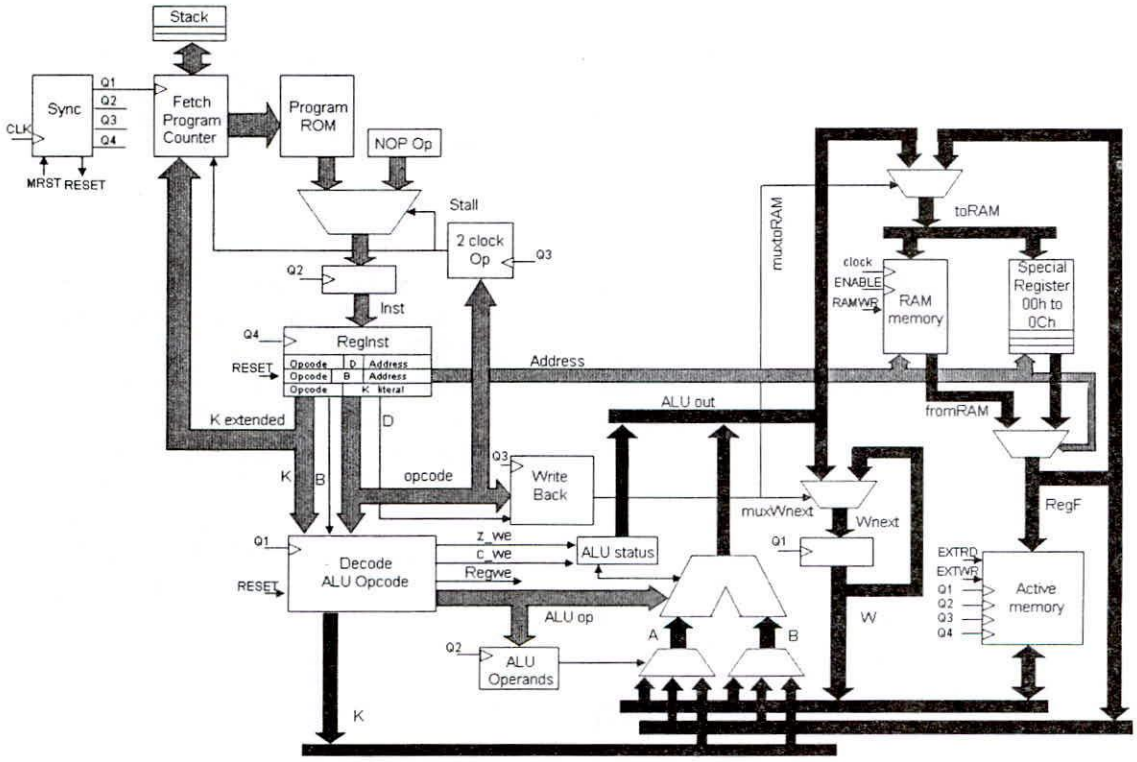

Fig. 8 the UMASScore schematic

The interface of this module is shown below; and in general this module can be used as an extended port, inside the FPGA, to communicate data with some other processes in the FPGA. See figure9.

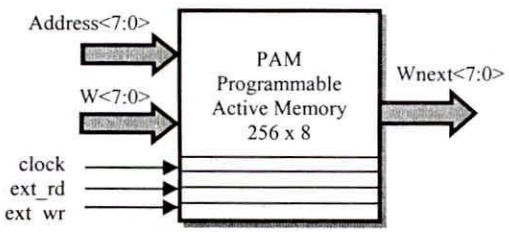

Fig. 9. The block diagram of the PAM module.

The VHDL description of the Expansion module is given in annex V.

IV IMPLEMENTATION AND TIMING ANALYSIS

After writing the VHDL code that implement the UMASScore, we compile it and synthesize it for the Spartan3 device xc3s200[‡]. At this point, before placement and routing, functional simulations are done to correct misbehaviors and incorrect specifications in the code. The most important problems faced at this stage were:

- Inferring the 128 bit registers as a block RAM in the device: The coding style used produced LUT implementation of the memory, so the XST manual [11] was used to infer the use of block memory. The final type of memory coded was a single-port read first memory, so an active enable signal reads the memory.
- Initializing and synchronizing the program counter: Though in idea simple, making the program counter start at 1FFF and fetching the next instruction forced a modification in the time the next instruction was being fetched.
- When executing conditional branches the jump was never taken: This problem was due to for conditional branches the zero flag has be set, and we were taking the decision of the branch at phase Q3 looking at the zero flag on the status register that is written at phase Q4. This problem was solved by looking at the zero output of the ALU.
- When executing unconditional branches there was a mismatch between the program counter and the executing instruction: This problem was introduced by the modification done to the fetch unit, this was solved by setting the PC to the previous address of the destination, this is address – 1. With this

[‡] The reason for choosing this device, as explained later, is the big amount of I/O pins that it provides, 173.

modification the fetching unit was able to force a NOP at address – 1, while decoding the instruction stored in address.

- Resetting all the registers: As the clock unit is stop at reset and forced to phase Q1, resetting the registers at phases Q2, Q3 and Q4 with the MRST signal was impossible. To overcome this problem an internal RESET signal was created, this signal propagates once with a NOP instruction when the MRST signal is released.

After solving these problems the functional description of the UMASScore was correct, and the process of simulating the design after placement and routing began. The principal problems encountered at this point were:

- The program counter was stall in zero and the phased clock was stack in Q1: This problem was solved by looking at the synthesis report. The longest path found in synthesis was from MRST to an internal node; and we were holding the MRST signal for less than one clock cycle, so we increased the hold time of MRST to 4 clock cycles which is the time taken to execute one instruction.
- I/O pin limitation on the FPGA: Doing simulation after placement and routing is simulating from netlist, at this level all the signals are encapsulated in a black box and the design is observable only from its input/output pins. At this point checking the correctness of the UMASScore from its outputs only was not helpful, so we extracted internal signals to the output to observe the execution of the instruction. The latter produced an increase of I/O pins, from 18 pins (1 bit clock, 1 bit MRST, 8 bits PORTA and 8 bits PORTB) to 220 pins for complete visibility of the control and intermediate results on the datapath. This increase on I/O pin requirements to verify the correct behavior of the UMASScore leads us to change the initial Spartan2 device xc2s50 to the actual Spartan3 xc3s200. As the xc3s200 has 173 I/O pins, only the most important signals where extracted from the UMASScore as outputs, some other signals where derived from these outputs in the testbench given in annex VI.
- False triggering of control signals: Once the placed and routed signals were observable during simulation, there were a lot of mismatches between the expected results and the observed results. Most of these mismatches were due to control signals triggered out of their scheduled phase. Looking in detail at the simulation we found that the clock signals where unbalanced, and between phase Q1 and Q2 there was a middle ground where no phase was active. This was activating incomplete specified control signals that were corrected, and some small reschedule of non critical path operations were done so the four phased clock were as balanced as possible.

- With all the bugs fixed the program shown in the ROM module is tested in the UMASScore, and its simulation after placement and routing is shown in the following 6 figures. Some glitches can still be observed on the ALU unit as this unit is not synchronized with the clock, that is, the ALU is asynchronous and its outputs are affected by any change in its inputs. The glitches in the ALU unit do not produce any misbehavior as the correct result is already computed when the synchronous part of the UMASScore requires it, phase Q3.

V. CAD Reports

To get accurate reports on area, power and speed the extracted signals used in simulation are commented. Thus the UMASScore is implemented without the additional logic and I/O pins used for verification purposes. Its interface is shown in the figure below. See figure 10.

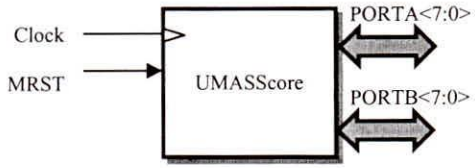

Fig. 10. The Block diagram of the UMASScore.

From the different steps of the CAD design process, only the most important results of the generated report files are shown here.

A. From Synthesis

At this step is important to notice that the memories for the RAM and PAM module where inferred correctly.

```
Synthesizing Unit <PicPAM>.
  Found 256x8-bit single-port block RAM for signal <PAM>.
-----------------------------------------------------------------------
| mode          | read-first                                |          |
| aspect ratio  | 256-word x 8-bit                          |          |
| clock         | connected to signal <clock>               | rise     |
| enable        | connected to signal <ENABLE>              | high     |
| write enable  | connected to signal <PAMWR>               | high     |
| address       | connected to signal <PAMAddr>             |          |
| data in       | connected to signal <Store>               |          |
| data out      | connected to signal <EXT_Dout>            |          |
| ram_style     | Auto                                      |          |
-----------------------------------------------------------------------
  Found 4x4-bit multiplier for signal <Store>.
```

```
Synthesizing Unit <PicRAM>.
  Found 128x8-bit single-port block RAM for signal <RAM>.
-----------------------------------------------------------
| mode          | read-first                      |       |
| aspect ratio  | 128-word x 8-bit                |       |
| clock         | connected to signal <clock>     | rise  |
| enable        | connected to signal <Enable>    | high  |
| write enable  | connected to signal <RAMWR>     | high  |
| address       | connected to signal <RamAddr>   |       |
| data in       | connected to signal <DataIn>    |       |
| data out      | connected to signal <DataOut>   |       |
| ram_style     | Auto                            |       |
-----------------------------------------------------------
```

The adders in the ALU module where inferred as well as the decoder for the bitwise operations

```
Synthesizing Unit <PicALU>.
    Found 8-bit adder for signal <$n0000> created at line 61.
    Found 8-bit adder for signal <$n0006> created at line 61.
    Found 8-bit adder carry out for signal <$n0032> created at
          line 61.
    Found 8-bit xor2 for signal <$n0048> created at line 78.
    Found 1-of-8 decoder for signal <BitMask>.
```

And the finite state machines for the instruction set, the adder of the program counter and multiplexer controls where implemented.

```
Synthesizing Unit <piccpu>.
    Using one-hot encoding for signal <opcode>.
    Using one-hot encoding for signal <StallOpcode>.
    Found 13-bit adder for signal <AddrPreFetch>.
    Found 74 1-bit 2-to-1 multiplexers.
    inferred 349 D-type flip-flop(s).
    inferred   4 Adder/Subtracter(s).
    inferred   3 Comparator(s).
    inferred  87 Multiplexer(s).
```

The big amount of flip flops inferred is to implement the registers of the control and data signals in the CPU module. After synthesis the total amount of resources used by the UMASScore microcontroller are:

```
Design Statistics
# IOs                              : 18
Macro Statistics :
# RAM                              : 2
#      128x8-bit single-port block RAM: 1
#      256x8-bit single-port block RAM: 1
# Registers                        : 110
#      1-bit register              : 82
#      11-bit register             : 1
#      13-bit register             : 8
#      14-bit register             : 2
#      3-bit register              : 1
#      4-bit register              : 1
#      5-bit register              : 1
#      8-bit register              : 14
# Multiplexers                     : 40
#      13-bit 8-to-1 multiplexer   : 1
#      2-to-1 multiplexer          : 39
# Decoders                         : 1
#      1-of-8 decoder              : 1
# Adders/Subtractors               : 5
#      11-bit subtractor           : 1
#      13-bit adder                : 1
#      8-bit adder                 : 2
#      8-bit adder carry out       : 1
# Multipliers                      : 1
#      4x4-bit multiplier          : 1
# Comparators                      : 3
#      4-bit comparator greater    : 1
#      8-bit comparator greatequal : 1
#      8-bit comparator less       : 1
# Xors                             : 2
#      1-bit xor3                  : 2
```

And these resources for the Spartan3 xc3s2000 represent a device utilization of:

```
Device utilization summary:
---------------------------

Selected Device : 3s200ft256-5

 Number of Slices:                      568  out of   1920    29%
 Number of Slice Flip Flops:            373  out of   3840     9%
 Number of 4 input LUTs:               1015  out of   3840    26%
 Number of bonded IOBs:                  17  out of    173     9%
 Number of BRAMs:                         2  out of     12    16%
 Number of MULT18X18s:                    1  out of     12     8%
 Number of GCLKs:                         1  out of      8    12%
```

It is important to notice that the device utilization varies when:

- Not all the instructions are implemented: When a instruction is not used the corresponding state on the FSM becomes not reachable, for example if the CALL instruction is never used not only the state is drop but also the circuit controlled by the GOTO state is minimize on the synthesis process. This prunning reduces the number of slices used in the device.
- The number of instructions increase or decrease: We've seen that the UMASScore is capable of storing up to 8K instructions, each of 14 bits. As the ROM memory is implemented with LUTs reducing or increasing the number of instructions well result in more or less device utilization.

B. From Placement and Routing

As from synthesis we got the device utilization, the the most important information after placement and routing is to generate the timing analyzer report. This is shown below:

```
--------------------------------------------------------------------------------
Release 6.3.03i - Timing Analyzer G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

Physical constraint file:
C:\Xilinx\My_Designs\PiccpuPR\piccpu.pcf
Device,speed:              xc3s200,-5
================================================================================
Timing constraint: Default period analysis for net "Q4"
 26766 items analyzed, 0 timing errors detected. (0 setup
errors, 0 hold errors)
 Minimum period is  14.337ns.
--------------------------------------------------------------------------------
Delay:                  14.337ns (data path - clock path skew)
  Source:               RegInst_0 (FF)
  Destination:          BufferPortA_1 (FF)
  Data Path Delay:      13.820ns (Levels of Logic = 6)
  Clock Path Skew:      -0.517ns
  Source Clock:         Q4 rising
  Destination Clock:    Q4 rising
  Clock Uncertainty:    0.000ns

  Data Path: RegInst_0 to BufferPortA_1
    Delay type          Delay(ns)  Logical Resource(s)
    ------------------------------  -------------------
    Tcko                  0.626    RegInst_0
    net (fanout=8)        2.353    RegInst<0>
    Tilo                  0.529    Ker28112_SW0
    net (fanout=1)        0.343    Ker28112_SW0/O
    Tilo                  0.529    Ker28112
    net (fanout=3)        0.690    N28114
    Tilo                  0.529    Ker2952630
    net (fanout=16)       1.574    CHOICE4025
    Tilo                  0.529    RamAddr<2>1
    net (fanout=11)       1.515    RamAddr<2>
```

```
    Tilo                  0.529   Ker296221
    net (fanout=16)       1.697   N29624
    Tilo                  0.529   _n0669
    net (fanout=1)        1.324   _n0669
    Tceck                 0.524   BufferPortA_1
    ----------------------------  ----------------------------
    Total                13.820ns (31.3% logic, 68.7% route)
--------------------------------------------------------------------------------
All constraints were met.


Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 28727 paths, 0 nets, and 4375 connections

Design statistics:
   Minimum period:  14.337ns (Maximum frequency:  69.750MHz)
   Minimum input required time before clock:  15.571ns
```

From this report the maximum frequency of the UMASScore is 69.75 MHz, though we were only able to run the post placement and routing simulation at a frequency of 62.5 MHz (clock period of 16 ns). At higher frequencies some registers started getting unknown values, though the simulation at the end of phase Q4 was still valid.

This timing analysis is more accurate than the one obtained in synthesis where a maximum frequency of 106 MHz was found.

```
Timing Summary:       (From synthesis, before P&R)
---------------
Speed Grade: -5
   Minimum period: 9.346ns (Maximum Frequency: 106.998MHz)
   Minimum input arrival time before clock: 13.189ns
```

Another important result is that the critical path is given by phase Q4 with a minimum period of 14.337 ns; and the next critical path comes from phase Q1 with a minimum period of 9.670 ns. This is important as we thought at the beginning of the design that the critical path will be given by the ALU computation in phases Q2 or Q3. There are two reasons for these:

- The paths on phase Q4 and Q1 have to determine lots of signals, i.e decode the instruction and select the control signals to write to memory. This means that the clock of these two phases Q4 and Q1 have more load than Q3 and Q2, as shown in the following report:

```
------------------------------------------------------------------
Clock Signal                    | Clock buffer(FF name)   | Load  |
--------------------------------+-------------------------+-------|
Q3:Q                            | NONE                    | 8     |
Q1:Q                            | NONE                    | 164   |
Q2:Q                            | NONE                    | 32    |
Q4:Q                            | NONE                    | 149   |
EXT_WR:Q                        | NONE                    | 8     |
PAM_ENABLE(PAM__n00011:O)       | NONE(*)(PAM_PAMAddr_7)  | 8     |
clock                           | BUFGP                   | 6     |
------------------------------------------------------------------
```

- The ALU implemented on our design is asynchronous, so it is always computing the selected operation when the values at its inputs change. And this means that the computation of the right value can be splitted between the phase Q2 and the moment on phase Q3 where its output is needed.

C. From Power Analyzer

To be able to run the power analyzer, files with extension NCD and VCD for our design where needed. The VCD files where obtained by checking the option to write an output VCD file on the post P&R simulation, and the NCD files where obtained from the Map and Place & Route property menu.

Setting the power analyzer to a confidence level name reasonable, the average power consumption of the UMASScore is 28mWatts[§]. This result is shown in the following fragment of the power report file:

```
--------------------------------------------------------------------
Release 6.3.03i - XPower SoftwareVersion:G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.
Design:       piccpu
Preferences:  C:\Xilinx\My_Designs\PiccpuPR\piccpu.pcf
VCD File:     C:\Xilinx\My_Designs\PiccpuPR\piccpu.vcd
Part:         3s200ft256-5
Data version: ADVANCED,v1.0,11-03-03

Power summary:                          I(mA)     P(mW)
--------------------------------------------------------------------
Total estimated power consumption:                   28
                                        ---
                      Vccint 1.20V:       3         3
                      Vccaux 2.50V:      10        25
                      Vcco25 2.50V:       0         0
                                        ---
          Quiescent Vccaux  2.50V:      10        25

Thermal summary:
--------------------------------------------------------------------
   Estimated junction temperature:                   26C
                  Ambient temp:  25C
                     Case temp:  26C
```

VI. COMPARISON BETWEEN THE UMASSCORE AND THE PIC16F84 MICROCHIP MICROCONTROLLER

The UMASScore device is compared against the PIC16F84 and PIC16F877[**] in frequency, power dissipation, program memory, flexibility and price. Even though initially we had the objective of comparing area, at this point we realize that it really does not make much sense to compare the number of slides or gate count used on the FPGA against the PIC device; so we take into consideration the area or percentage of utilization to reduce the price of the FPGA. It can be argued that this is an artificial price and it does not correspond to market, but at least this

[§] This power dissipation is really small and it is probably due to the low toggle of signals for the tested program, it will be of interest to see how this power varies if the I/O ports are used frequently.

[**] The PIC16F877 is similar to the PIC16F84 and support the same instruction set. It has a bigger program memory and some specialize registers for UART communication.

approach gives us an estimate on how much resources are left on the FPGA for continuing adding functionality. In the following table the values shown for the PIC16F84 and PIC16F877 are taken from their datasheet [7]; and the values shown for the UMASScore are taken from the report analysis.

TABLE V

Comparison Table	XC3S200-4VQ100C UMASScore	PIC16F84 Microchip	PIC16F877 Microchip
Max Frequency	69.75Mhz	10Mhz	20Mhz
Power dissipation	28mWatts	800mWatts	1Watt
Memory ROM	8K instructions	1K instructions	8K instructions
Instruction customization	PAM memory	None	None
Percentage of utilization	29%	--	--
Device Price	13.45$	4.39$	5.11$
Utilization Price	3.9$[††]	4.39$	5.11$

This table shows that the UMASScore device achieves a speed up of 6.9X against the PIC16F84 for a similar price[5]. The UMASScore has some degree of flexibility that none of the PIC devices have; as a customize hardware can be added inside the FPGA to perform specialized functions that are not provided on the instruction set of the microcontrollers.

VII. CONCLUSIONS

The more advanced VLSI process of the FPGA technology plays a key role on speed and power. This gives an advantage to the soft-core version over the real microcontroller. The price, per percentage of silicon used, is also cheaper for the soft-core version. In this case, the FPGA soft-core version of the microcontroller outperforms the microcontroller in speed by a factor of 6.9, and in power by a factor of 28 for roughly the same price.

The design of the soft-core has been verified by executing the whole instruction set with post placement and routing simulations. From the synthesis process we have observed that the softcore implementation of a microcontroller saves space when there are unused resources, as these unused resources are found to be unreachable or never used for the synthesizer and they are ripped out by the optimization tool. Looking at the report files generated from placement and routing we have seen that, for the UMASScore, the critical path lies on the decoding of the instruction to execute and not on the arithmetic operation to be performed as we initially thought.

Some modifications were done to the initial UMASScore code in order to work properly after placement and routing, as interconnect delays and loads were not taken into account on the first behavioral simulation. The changes done were merely on retiming certain operations or fully specifying multiplexers and decoders to avoid glitches on control signals. When proper simulation of the post place and route model was achieved, the cad processes were redone for the UMASScore, commenting all the signals that were placed as outputs during the verification analysis.
The verification of the UMASScore let us understood the complexity of this process and its important, as most time of the design process was devoted to the verification process. This also shows the importance of the JTAG port for testing as if the design is placed on a printed board we will not be able to extract our internal signals to verify its internal execution as we did it here. The process of verification gave us a practical example of Rent's rule, in which the design without the predefine port interface exploded the I/O pin requirements from 18 pins to 220 pins.

From here we can see that the gain in speed and power of the soft-core version comes with a more complex design process, and longer times of verification. This detriment can be neglected if we use IP soft-cores for the microcontrollers, but of course at an additional cost. If the FPGAs manufactures release their soft-core IPs, the market of embedded systems will drift to FPGA devices.

ACKNOWLEDGMENTS

I would like to thanks professors Guy Gogniat and Rusell Tessier for their valuable comments and corrections.

REFERENCES

[1] Actel Inc, *the core8051*, http://www.actel.com.
[2] Altera Corporation, Nios II Device, http://www.altera.com/products/ip/processors/nios2/
[3] Gartner Dataquest rankings, *2002 Microcontroller Market Share and Unit* Shipments, http://www3.gartner.com/

[††] This price can be misleading as the 29% of device utilization correspond to a program of 389 lines, a more fare comparison will synthesize the FPGA with the ROM fully utilized, 1K or 8K instructions, and use that as percentage as utilization. Though this 30% of device utilization while be valid up to 2K instructions if the remaining block memories of the device are used to implement the ROM.

[4] J. Zafra, *VHDL implementation of the microcontrollers PIC-16/17*, University of Sevilla, Jun 2000.

[5] J. Clayton, *the PIC16F84 in Verilog*, http://www.opencores.org/projects.cgi/web/risc16f84/

[6] Microchip Company, *Microchip Technology Jumps to Number One in Worldwide 8-bit Microcontroller Shipments*, http://www.microchip.com/stellent/, press release Jul 2004.

[7] Microchip Company, *PIC16F8X Datasheet*, http://www.microchip.com

[8] S. Morioka, *VHDL implementation of the PIC16F84 in FPGA*, Transistor Gijutsu Magazine, Dec 1999, http://www02.so-net.ne.jp/~morioka/cqpic.htm,

[9] T. Coonan, *The synthethic PIC*, 1999, http://www.mindspring.com/~tcoonan/synthpic.html.

[10] Xilinx Inc, *Processor central*, http://www.xilinx.com/products/design_resources/

[11] Xilinx Inc, *XST manual, http*://www.xilinx.com/

[12] Xilinx Inc, *Application notes: xapp463 and xapp464 – Spartan3 FPGA family*, http://www.xilinx.com/, Jul 2003.

[13] P. Bertin, D. Rocin and J. Vuillemin, *Programmable Active Memories: a Performance Assesment*, Digital Equipment Corporation Paris Research Lab, 1993