ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO DE TELECOMUNICACIÓN

Título del proyecto:

SPARC instruction set extension for GNSS SW receivers on LEON2FT processor

Hugo Arguiñáriz Bridonneau

Tutor: Javier Navallas

Pamplona, 20 de Junio de 2014

(This page was intentionally left blank)

Table of Contents

Table of figures

1- INTRODUCTION

Project context

This project was developed as part of a 6 months internship at the European Space Agency (ESA). The ESA has several research centers all around Europe, but most of its technical development is done at the European Space Research and Technology Centre (ESTEC) in Noordwijk, the Netherlands. The whole project took place at ESTEC as an intern at the TEC-ED department.

Figure 1: Aerial view of ESTEC (left) and ESA locations (right)

This project will be used as the final research project for the Telecommunication Engineering studies at the Public University of Navarre (UPNA), Spain. This could be considered equivalent to a degree thesis in other educational systems.

Introducing the Galileo GNSS

To understand the usefulness of this project we must know what the European Galileo project is.

A Global Navigation Satellite System (GNSS) is any network of geostationary satellites that are used to provide worldwide geo-spatial positioning on the Earth surface. Two examples are GPS and Galileo.

Most people are familiar with the American GPS system. Until very recently GPS was the only widely available satellite navigation system. The USA military grants civilian availability worldwide and it is being used for any imaginable application.

However GPS has several drawbacks. GPS is entirely controlled by the USA Department of Defense (DoD). That means that the DoD could, in case of conflict, intentionally degrade or interrupt GPS in any region of the world.

For most of the history of GPS, civilian availability was very limited. Although the DoD granted worldwide civilian access, the civil signal was voluntarily degraded and accuracy was limited to 100m, while the American military had access to undistorted GPS with precision up to 20m.

This double system of a different civilian and military access, called selective availability, was disabled in 2000. However it shows how the access to GPS can be controlled and restricted by policy makers according to political or military interest, thus making GPS unreliable for foreign governments.

That is why some countries have been trying to develop their own independent system. This would prevent external control and avoid dependence on the USA system. Several countries have developed systems on a regional level (covering the whole country territories, but without worldwide coverage). Some examples of countries with such systems already in operation are China, Japan, India and France.

Those are systems with only a regional coverage. However in the last decade several systems with worldwide coverage have been developed. The 4 worldwide satellite navigation systems to be operational in the near future are:

- GPS (USA, already in operation)
- GLONASS (Russian Federation, already in operation)
- COMPASS (China, not fully operational yet)
- GALILEO (Europe, to be fully operational by 2020).

The Galileo project is being developed by the European Union and the ESA to serve as an autonomous alternative to the American GPS. Its goal is to provide a high precision, reliable and global satellite positioning systems upon which all European nations can rely, independent from other similar systems controlled by foreign powers (USA, Russia or China).

A very distinctive feature of Galileo will be that it is not a military project, and it will be controlled by the European Union directly. This makes it better for civilian applications.

One of the main focuses of the Galileo project will be search and rescue operations. Although GPS can already be used for this role, Galileo was specifically optimized for search and rescue from the beginning, making it much more precise in situations like natural disasters. The Galileo receivers themselves will have a distress signal that can be activated by the user in emergency, immediately alerting the Rescue Co-ordination Centre and giving rescue teams an extremely precise position of the person.

The specification of Galileo states an accuracy better to 1 m, which is much higher than any other similar system (at the time of writing, GPS can only reach 7.8 m in accuracy). This high accuracy makes Galileo suitable for civilian applications not possible with current GPS systems, like automatic air traffic control.

The first Galileo satellites were launched for test purposes in 2011, and the system is planned to be fully operational by 2020.

This project is not directly linked to the Galileo project, but is applicable to any GNSS system, including GPS. Although the particular details of each different GNSS system vary, the basic principles this project deals with are common to all systems.

The main goal of the project is simply to optimize the reception of GNSS signals. This makes receptors consume less time and resources in the process of getting a GNSS signal.

During the development of this project I was indeed in contact with people who were working on the Galileo project, mainly in order to get ideas and feedback. Because it is system independent, it can be used on GPS receivers while Galileo is still not operational.

Brief GNSS overview

To understand this project we must briefly see how GNSS receptors work. GNSS systems are extremely complex so only a few relevant aspects will be mentioned. For a full explanation of GNSS receivers please refer to more complete sources.

To determine its location the receiver (mobile phone or any other GNSS device) must receive signals from several satellites at the same time (at least 4, although usually many more are received). Receivers are passive components that don't send any data, they just listen to incoming signals.

The receiver must know exactly which of all the satellites orbiting the Earth he is receiving from. By knowing which satellites are visible, and the exact location of each satellite, it can calculate its own position.

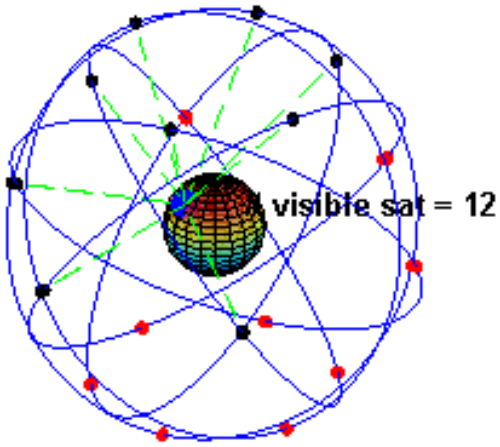

Figure 2: Example of GNSS receptor receiving broadcast signals from 12 satellites at the same time

All satellites broadcast in the exact same frequency, so there must be a way by which the receiver determines which satellites the signal came from. In order to do this each satellite has its own unique code. The signal is mixed with this unique code before broadcasting it.

Those codes are always the same and unique for each satellite, so when we get a signal we know which specific satellite it came from, as long as we are able to decipher which code it carries. In the case of GPS this unique code is called C/A code, or spreading code.

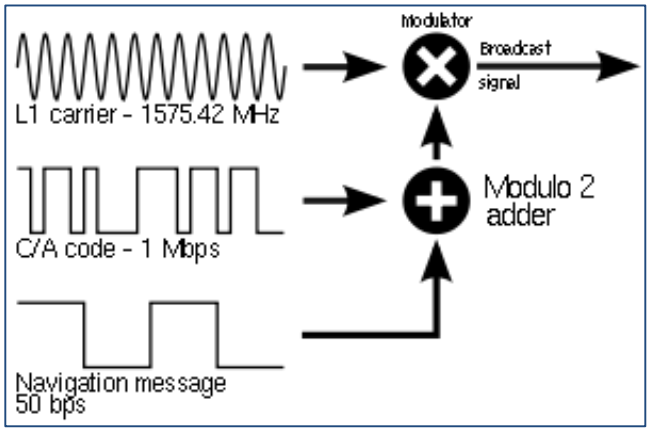

Figure 3: Simplified modulation scheme of a GPS broadcast signal with C/A code.

When the signal reaches the receiver, the code is extracted and compared with a list of all known codes. The codes corresponding with the correct satellites will then be detected.

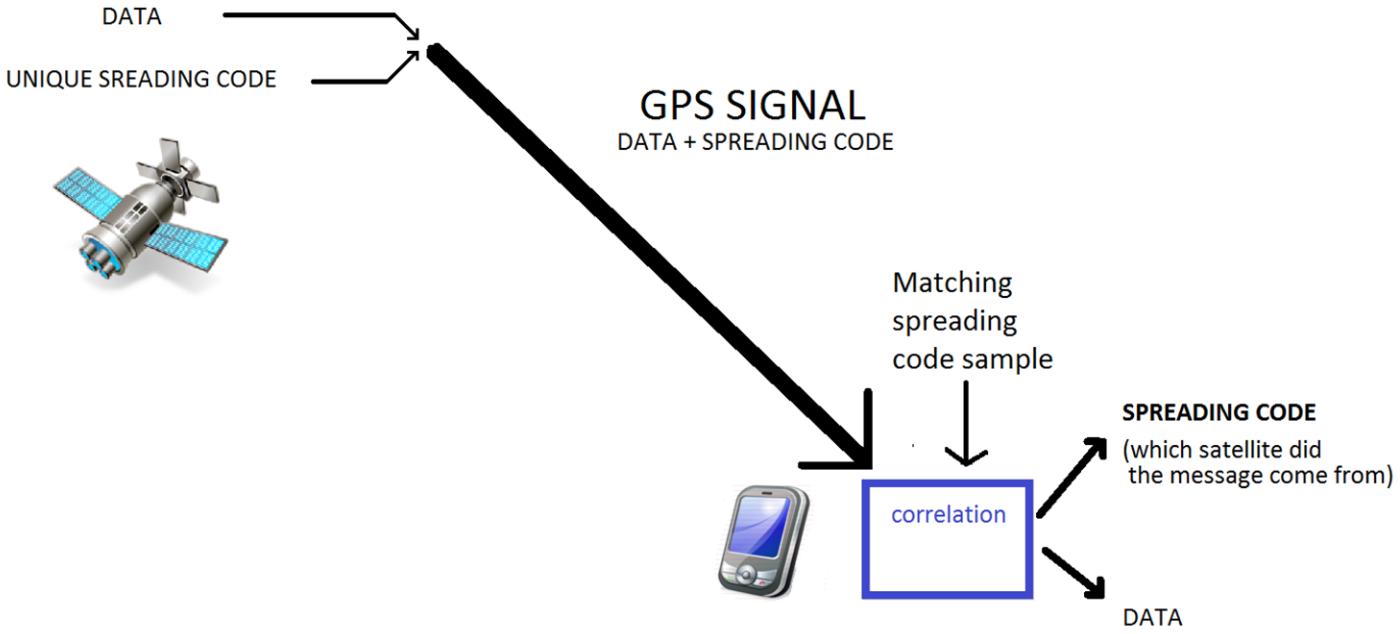

Figure 4: Schematic showing where the correlation of spreading code and signal takes place

This comparison is done via a correlation. The signal is continuously correlated with sample codes. When the code doesn't match the result of the correlation will almost be zero. However when we get the right code there will be a peak in the output that can be detected.

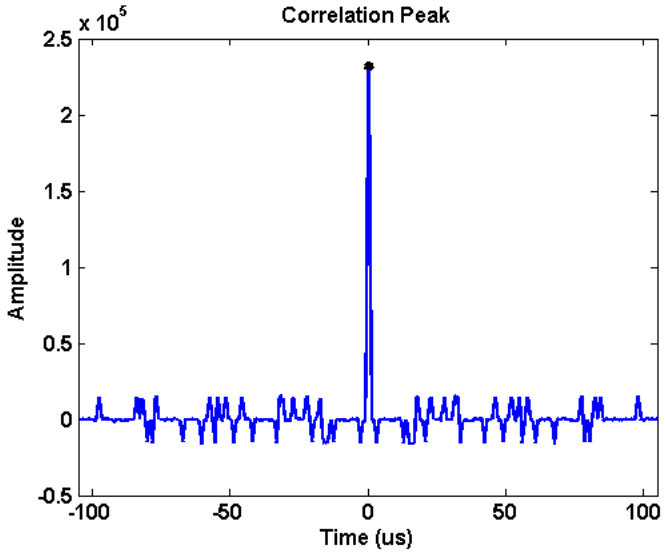

Figure 5: Correlation peak example in GNSS receiver

Correlation is a simple operation consisting in only a sumatory of multiplications:

$$Z(t) = \sum x(i) * c(i-t)$$

Being x the signal and c the sample code.

The correlation is then just a continuous multiplication and accumulation of samples.

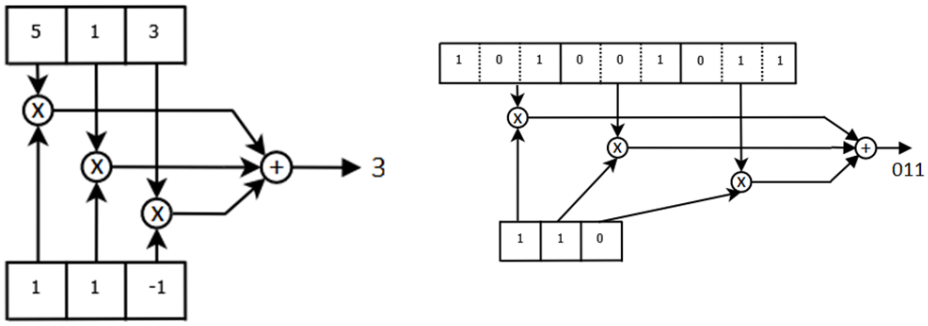

Figure 6: Example of 3 sample multiply and accumulate

However in GNSS receivers it must be calculated continuously at a very high speed. A typical receiver has more than 8 channel, with 3-5 correlation units per channel, which means that not only one, but many correlations are taking place in parallel all the time.

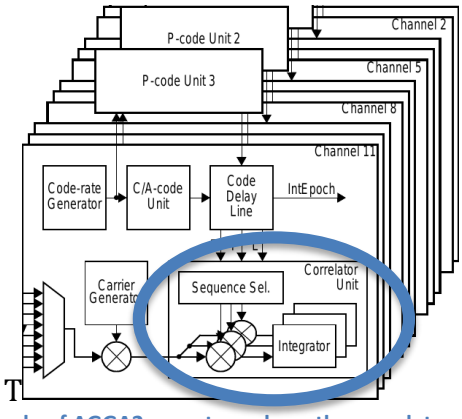

Figure 7: Example of AGGA2 receptor, where the correlator unit is shown.

Even such a simple operation made so fast and so often can become a serious bottleneck if not optimized properly. If better and faster ways to perform correlations in the receiver could be found performance could be greatly increased.

Because correlations can become such a bottleneck in receiver design, **the purpose of this project was to find a way to optimize correlation to make it faster**.

Ways to improve correlation performance

1- ***Software optimization:*** If the correlation is done by software the algorithms could in theory be improved. Some really clever optimizations have already been developed in the past.

2- ***Specialized hardware optimization:*** In the same way that computers have hardware dedicated to graphics or sound, designing circuits that deal with only one task in an optimized way is always the solution which offers the highest performance, and it will always beat software solutions in raw speed. It would be possible to design correlation hardware for this specific task (Application Specific Integrated Circuit or ASIC). However this is the least flexible option and modern system try to avoid it. Once the device is functioning a software update is pretty straightforward on the existing device, but a hardware update would require to throw away the device and get a new one altogether.

3- ***Processor optimization:*** This falls in between the two previous solutions and it's the scope of this project. With this approach the correlation is performed by software. It is the underlying hardware that is optimized to deal with specific operations in an optimized way, but how that hardware is used depends on the software programmer, thus retaining flexibility.

This project will take the last approach, modifying the processor to deal faster with correlation instructions.

The core of the Project: CMUL operation

The way this project tries to optimize correlation for the GNSS receiver is introducing new custom instructions into the processor. The software programmer can then use this fast instructions for optimized code.

It is important to note that correlations can already be done with existing instructions in any processor family. But those instructions are general purpose and thus not optimized. This project offers new alternative instructions that are less general but optimized (they do less, but they are good at doing it).

The instructions will be called Cumulative Multiplication (CMUL) instructions, and are basically a hybrid between multiply instructions and accumulate instructions. Thus instead of first multiplying samples and then adding the results, CMUL instructions do it all at once.

A single instructions can also multiply and accumulate several samples in parallel, thus optimizing the process even more. A more detailed explanation of how CMUL instructions work and why they improve performance will be discussed in later sections.

This project will take advantage of the fact that most existing instructions work on 32 bit registers, but samples for GNSS signal correlations are usually between 1 and 3 bits wide only. Thus most of those 32 bits are wasted.

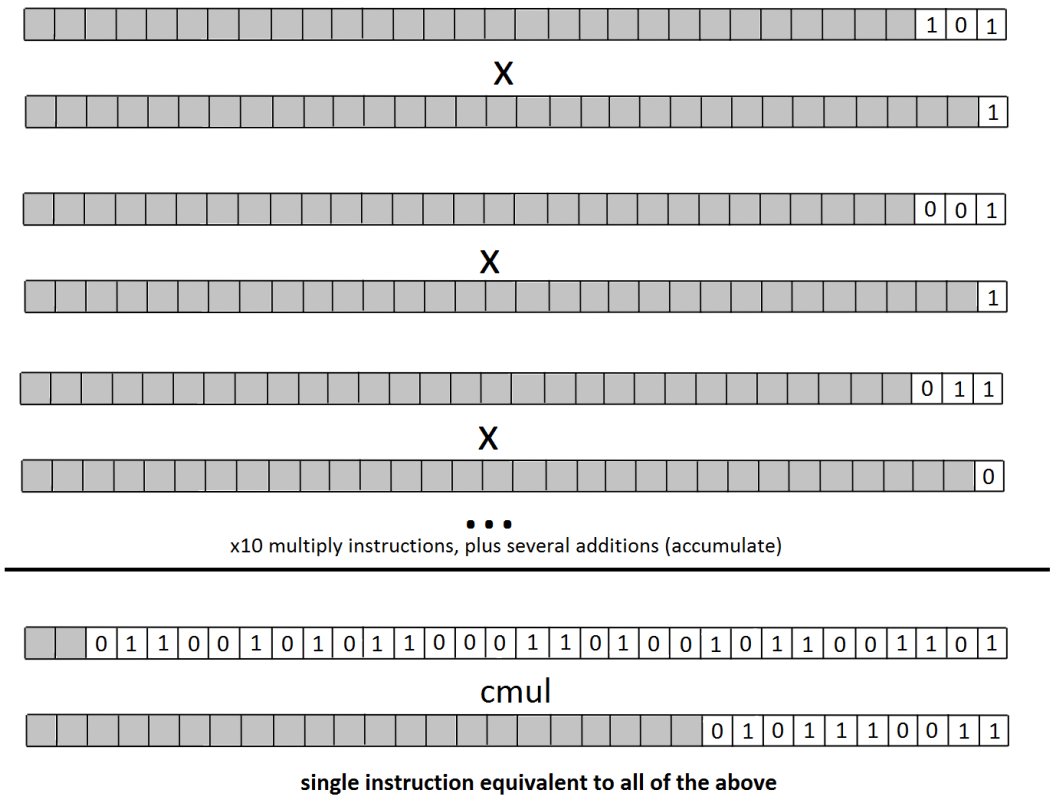

Figure 8: Classical multiply and accumulate using several instructions vs a single cmul operation.

Introducing the LEON processor

LEON is a 32 bit RISC processor based in the SPARC-V8 architecture and instruction set. It was originally developed at ESTEC at the TEC-ED department (the same department this project was collaborating with). Development of the LEON 1 started in 1997, and the most current version is the LEON 4.

Originally it was developed by engineer Giri Gaisler, working at ESTEC. However after the release of the first version Gaisler left ESA to focus on the development of LEON and founded Gaisler Research, the company that currently owns and distributes the LEON processor. The LEON is distributed as self-contained VHDL code. That way the user can burn the processor into any FPGA or printed circuit of their choice.

Most LEON versions are distributed for free by Gaisler Research. That has made it a very popular processor for universities and other research centers.

ESA owns a propietary version of LEON, called the LEON2FT (FT = Fault Tolerant). This is a Single Event Upset (SEU) fault tolerant version of LEON2. All flip flops are protected by triple redundancy, and additional parity checks protect critical areas of the processor. That ensures that the processor wont be affected by external radiation in space. This version is licensed by ESA.

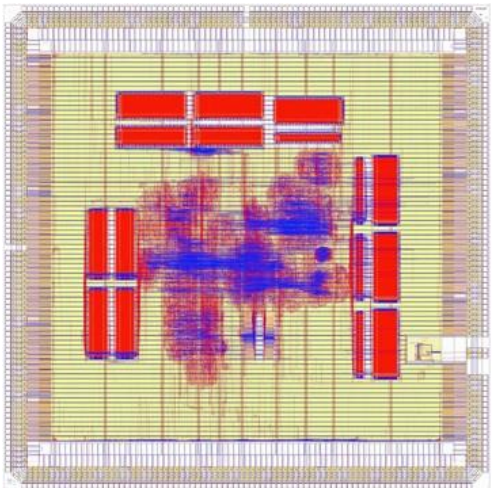

Figure 6: LEON2FT IP core.

This Project will modify a LEON2FT processor to implement the new instructions. There are more recent versions of the processor like LEON3FT or LEON4, but those have more complex structures and would be more complex to modify. Because the purpose of this project is not getting a releasable version of the modified processor, but to test an idea, it was decided to go for the simplest choice. If the idea proofs to be worthy, further research could be done to implement it on more advanced processors.

VHDL concepts overview

Because this project deals mainly with modifying the LEON2FT VHDL code, and the concept of VHDL is not very known for people not using it, this short overview will explain the concept so that the rest of the project is easier to understand.

VHDL is just a Hardware Description Language. The way a hardware designer codes VHDL may seem similar to the way a programmer uses C++ or Python. However instead of compiled into machine code, VHDL will be synthesized into a netlist that can be used to manufacture real hardware.

This is what confuses most people not familiar with VHDL. During the project we will be talking about the processors source code, but we must keep in mind that this code refers to a real hardware architecture and not to a software program.

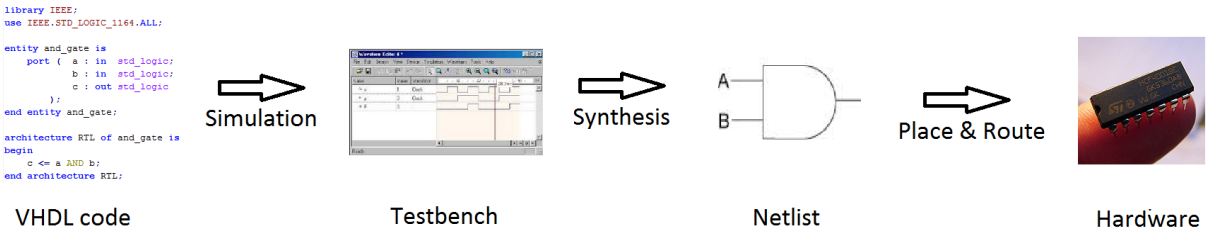

Figure 9: VHDL design phases

This may seem like a complex way to design hardware, but in projects involving millions of logic gates this is the only scalable method.

The LEON processor is fully described as a group of vhdl files. Those files are then linked together and synthesized into a netlist. That netlist can be used to program the processor into an FPGA or to manufacture an ASIC by printing the corresponding hardware components into a circuit board.

This project will modify some of those VHDL files to implement the changes. When that VHDL code is used to manufacture the processor circuit, the added CMUL instructions will be available to anyone using the chip.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity declaration, with I/O ports
entity mux is
    port (  A, B : in  std_logic;
            sel  : in  std_logic;
            Y    : out std_logic
         );
end entity mux;

-- Behavioural implementation
architecture RTL of mux is
begin
    process(A, B, sel)
    begin
        if (sel = '1') then
            Y <= B;
        else
            Y <= A;
    end process;
end architecture RTL;
```

Figure 10: Example of a very simple 2 input multiplexer in VHDL

2- PROJECT SUMMARY

In order to make the document easier to read this overview will summarize the sections that will follow.

This project was born to test the idea of optimizing performance by adding new instructions to a processor. This has been done in the past, but never for this kind of applications.

Implementing this idea into a real world scenario would involve a big team of experienced developers, a big budget and more time than a simple 6 months internship offers. Because this would be a big commitment, this project will help decide if it is worth the effort.

This document will present all the steps from specification to final testing.

1. The first few sections will focus on technical specifications, constraints and explaining CMUL instructions in a more detailed and technical manner.

2. Implementation into the real processor VHDL code. Even the final processor is hardware, it is defined by VHDL code which will later be translated into hardware.

3. Adding instructions to the compiler and assembler. Once the processor is finished, it will be able to handle new instructions. However when programming software for that processor we must tell the assembler and compiler about the existence of those instructions and when to use them. This step was more complex than expected, so a complete tutorial on how to modify the GNU compiling toolchain and compile it will be provided step by step.

4. When both the processor and the assembler are ready some results can be tested to see if there was an improvement. The end of the project was spent proving that the modifications on the processor didn't cause any trouble and did not interfere with normal operation.

3- BACKGROUND

A general purpose microprocessor (GPUP) offers a set of standard arithmetic instructions with a fixed data width (e.g. 32 bit). In many applications, no optimal use can be made of these instructions, because the algorithm predominantly uses specific operations or a reduced data width. For example, correlation with coarse quantisation (1-3 bit) is used in spread-spectrum applications, e.g. navigation receivers, software defined radio (SDR). Such operations can always be emulated on a GPUP, but this often implies a huge overhead in terms of performance and power consumption, and present space GPUP often do not have enough spare performance for SDR.

The classical way out is to develop dedicated hardware processing blocks to be implemented on FPGA or ASIC devices, loosely coupled with a GPUP which is almost always necessary to control the processing. This concept is adequate for applications with a large and continuous demand for a specific type of processing. But it may not be very power- and resource-efficient for use cases with intermittent processing requirements.

To serve these use cases, lower level processing 'blocks' are more adequate. Elementary, semi-specialized instructions can be used flexibly for various types of application, and they allow an improvement of the power- and area-efficiency compared to SW emulation on a GPUP.

The SPARC instruction set has a number of instruction-opcodes which are not used by the specification. These can be assigned to custom instructions to be developed. These custom instructions can significantly enhance the performance for certain applications, while re-using, as much as possible, the existing integer unit data path, e.g. registers, load/store, hence minimizing the area and power overhead.

Similar concepts have been used on commercial microprocessors, such as the Pentium MMX extensions, introduced by Intel in the 90's, or MD5/Montgomery operations for crypto applications implemented on Sparc T4 [1].

Instruction set extension for LEON, are also found in literature [2] [3]. This two references can be used as an example and guideline.

4- OBJECTIVES

Development of custom instruction set extensions for SPARC general purpose microprocessors. The extensions shall be developed as a plugin or patch to the existing LEON2FT integer unit. The focus is on coarsely quantized correlation operations required for GNSS receivers. The extensions shall be verified in simulation and synthesis and a small demo application be developed.

5- ORIGINAL SPECIFICATION

The following specification uses the usual SPARC rs1/rs2/rd register naming convention. The spreading code (= one operand in rs1) is quantized in 1 bit corresponding to the values [-1, 1]. The signal (= other operand in rs2) can be quantized in various ways:

- CMUL2: 1-bit quantization like for the spreading code
- CMUL3: 2-bit 3 levels (also called 1.5 bit), corresponding to the values [-1, 0, 1]
- CMUL4: 2-bit code with values [-3, -1, 1, 3]
- CMUL7: 3-bit code with values [-7, -5, -3, -1, 1, 3, 5, 7]

Since the number of samples in rs2 varies (32 with 1-bit quantization, 16 with 2-bit, 10 with 3-bit), while the number of samples in rs1 is always 32, it could be convenient (if enough opcodes are available) to have several variants of the instructions, for example:

- CMUL3U and CMUL4U correlating rs2 with the upper part (bits 31:16) of rs1
- CMUL3L and CMUL4L correlating rs2 with the lower part (bits 15:0) of rs1
- CMUL7U, CMUL7M, CMUL7L correlating rs2 with the upper (29:20), mid (19:10) and lower (9:0) part of rs1

The correlation operation consists of the following operations:

- Vectorial multiplication rs1 * rs2. With 1-bit signal quantization, 32 samples are multiplied in parallel, with 2-bit 16 samples, and with 3-bit 10 samples. Due to the 1-bit spreading code, the multiplication is reduced to multiply by 1 or -1 (sign reversal).
- Adding up the 10, 16 or 32 partial sums.
- Adding the result to the accumulator rd.

The new arithmetics, should not significantly degrade the timing performance of the existing LEON logic. Single cycle operation is the goal and can probably be achieved with an accumulator limited to 32-bit. If the new logic introduces a critical path, pipelining over 2 cycles may be considered.

An implementation with a 64-bit result, involving the Y register, in analogy to the Sparc UMUL/SMUL instructions may be considered, but probably leads to timing problems. ICC flags should be updated to indicate overflow of the accumulator.

6- CONSTRAINTS

1. The new Sparc processor must be compatible with SPARC V8 specifications [4]. All SPARC V8 instructions remain implemented.
2. The LEON2 specific instructions smac and umac are not suppressed to maintain compatibility with programs using them.
3. The flow of the original instructions is not altered whatsoever. The pipeline remains unchanged, and the original instructions remain untouched. The only difference is the existence of new instructions.
4. The new instructions will be implemented in the processor itself. The goal of the project is not to implement new capabilities using a dedicated coprocessor.
5. The performance of the new instructions should be checked first at synthesizable RTL and also at a higher software level.
6. We use the fact that in the SPARC V8 some binary instructions opcodes are not valid. If executed they will generate an illegal instruction trap. The new instructions will be implemented using those unused encodings. We expect that a correct program will never use those illegal instructions to perform system calls, but instead use the supported way of performing system calls (ticc instructions). Incorrect programs that do not follow this standard may not work on the new processor.
7. The binutils assembler will be modified to recognize the new instructions. We do not intend to write a compiler who takes our instructions into account. Optimized techniques of doing so do exist, but this is considered a separate and ambitious project.

7- IMPLEMENTATION

7.1- Mapping the new instructions onto available opcodes

The SPARC V8 architecture defines 6 categories of instructions. Any new instructions must be implemented in the correct category, which for the CMUL instructions is the arithmetic instructions set. This is due not only to consistency, but it avoids adding overhead in the decode stage.

For this category of opcodes, there is 9 unimplemented gaps in the basic V8 specification. However LEON processors have their own additional instructions (smac, umac), so there is only 7 available opcodes for the CMUL expansion.
All 7 will be used for the CMUL instructions detailed in the project specification.

		op3 [5:4]			
		0	1	2	3
op3 [3:0]	0	ADD	ADDcc	TADDcc	WRASR† WRY‡
	1	AND	ANDcc	TSUBcc	WRPSR
	2	OR	ORcc	TADDccTV	WRWIM
	3	XOR	XORcc	TSUBccTV	WRTBR
	4	SUB	SUBcc	MULScc	FPop1 **See Table F-5**
	5	ANDN	ANDNcc	SLL	FPop2 **See Table F-6**
	6	ORN	ORNcc	SRL	CPop1
	7	XNOR	XNORcc	SRA	CPop2
	8	ADDX	ADDXcc	RDASR* RDY** STBAR***	JMPL
	9	CMUL3U	CMUL3L	RDPSR	RETT
	A	UMUL	UMULcc	RDWIM	Ticc **See Table F-7**
	B	SMUL	SMULcc	RDTBR	FLUSH
	C	SUBX	SUBXcc	CMUL2	SAVE
	D	CMUL4U	CMUL4L	CMUL7U	RESTORE
	E	UDIV	UDIVcc	CMUL7M	SMAC
	F	SDIV	SDIVcc	CMUL7L	UMAC

Figure 11: Table of extended arithmetic and logic opcodes

7.2- The new instructions

Although not stated in the original specification, the CMUL instructions are actually 3 operand instructions. Every CMUL operation needs not only the two input operands, but an extra input with the previous value of the accumulator. After each operation this accumulator will be updated for use by future instructions or for reading.

An extra register will be used to store the value of this accumulator. Sparc V8 provides a series of Ancillary State Registers (ASR) for this purpose. ASR 16 till 31 are available for implementation specific uses, and one of them will be used as an accumulator.

LEON2FT already uses certain ASR for different purposes. In particular register asr16, asr18 and asr24 to asr31 are already taken. For the CMUL expansion the register asr20 will be used, although this choice is completely arbitrary.

The implementation of the instructions in inspired in the LEON smac/umac instructions, which also use additional accumulator registers (%y and %asr18).

The asr18 accumulator can be written using the WRASR instruction. This can be used to reset the accumulator to zero. It can be also read using the RDASR instruction.

Note that because rs1 is assumed to always contain +/- 1 levels, the partial multiplications that appear in the following examples are actually just a sign change operation. A loockup table can be used instead of actual 32 bits multiplications, which would be overkill for this simple case.

Operations on 1 bit codes

Assembly syntax

CMUL2 rs1,rs2,rd

Operation

prod[31:0] = rs1[31]*rs2[31] + rs1[30]*rs2[30] + rs1[29]*rs2[29] + rs1[28]*rs2[28] + rs1[27]*rs2[27] + rs1[26]*rs2[26] + rs1[25]*rs2[25] + rs1[24]*rs2[24] + rs1[23]*rs2[23] + rs1[22]*rs2[22] + rs1[21]*rs2[21] + rs1[20]*rs2[20] + rs1[19]*rs2[19] + rs1[19]*rs2[19] + rs1[18]*rs2[18] + rs1[17]*rs2[17] + rs1[16]*rs2[16] + rs1[15]*rs2[15] + rs1[14]*rs2[14] + rs1[13]*rs2[13] + rs1[12]*rs2[12] + rs1[11]*rs2[11] + rs1[10]*rs2[10] + rs1[9]*rs2[9] + rs1[8]*rs2[8] + rs1[7]*rs2[7] + rs1[6]*rs2[6] + rs1[5]*rs2[5] + rs1[4]*rs2[4] + rs1[3]*rs2[3] + rs1[2]*rs2[2] + rs1[1]*rs2[1] + rs1[0]*rs2[0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

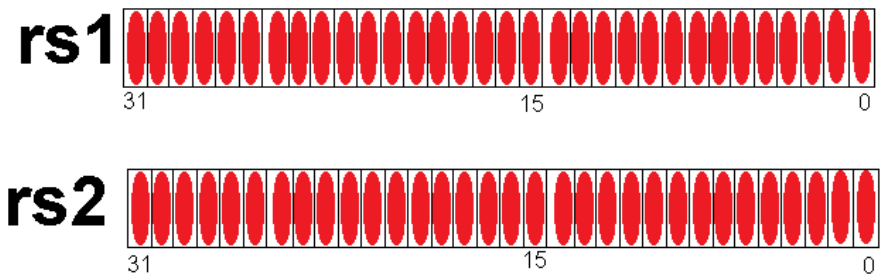

Figure 12: CMUL2 operation registers

Operations on 2 bit codes

Assembly syntax

CMUL3L	rs1,rs2,rd
CMUL4L	rs1,rs2,rd
CMUL3U	rs1,rs2,rd
CMUL4U	rs1,rs2,rd

Operation (for the CMUL3L and CMUL4L versions)

prod[31:0] = rs1[15]*rs2[31:30] + rs1[14]*rs2[29:28] + rs1[13]*rs2[27:26] + rs1[12]*rs2[25:24] + rs1[11]*rs2[23:22] + rs1[10]*rs2[21:20] + rs1[9]*rs2[19:18] + rs1[8]*rs2[17:16] + rs1[7]*rs2[15:14] + rs1[6]*rs2[13:12] + rs1[5]*rs2[11:10] + rs1[4]*rs2[9:8] + rs1[3]*rs2[7:6] + rs1[2]*rs2[5:4] + rs1[1]*rs2[3:2] + rs1[0]*rs2[1:0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

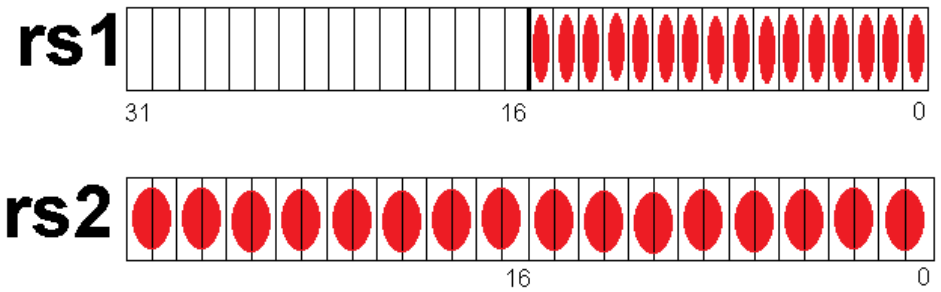

Figure 13: CMUL3L/4L operation registers

Operation (for the CMUL3U and CMUL4U versions)

prod[31:0] = rs1[31]*rs2[31:30] + rs1[30]*rs2[29:28] + rs1[29]*rs2[27:26] + rs1[28]*rs2[25:24] + rs1[27]*rs2[23:22] + rs1[26]*rs2[21:20] + rs1[25]*rs2[19:18] + rs1[24]*rs2[17:16] + rs1[23]*rs2[15:14] + rs1[22]*rs2[13:12] + rs1[21]*rs2[11:10] + rs1[20]*rs2[9:8] + rs1[19]*rs2[7:6] + rs1[18]*rs2[5:4] + rs1[17]*rs2[3:2] + rs1[16]*rs2[1:0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

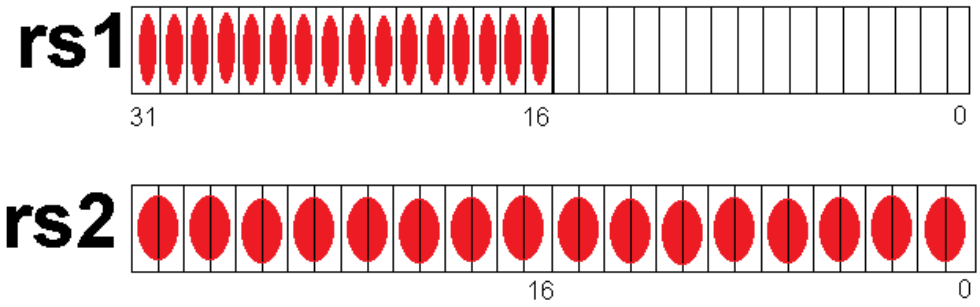

Figure 14: CMUL3U/4U operation registers

Operations on 3 bit codes

Assembly syntax

```
CMUL7L      rs1,rs2,rd
CMUL7M      rs1,rs2,rd
CMUL7U      rs1,rs2,rd
```

Operation (for CMUL7L)

prod[31:0] = rs1[9]*rs2[29:27] + rs1[8]*rs2[26:24] + rs1[7]*rs2[23:21] + rs1[6]*rs2[20:18] + rs1[5]*rs2[17:15] + rs1[4]*rs2[14:12] + rs1[3]*rs2[11:9] + rs1[2]*rs2[8:6] + rs1[1]*rs2[5:3] + rs1[0]*rs2[2:0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

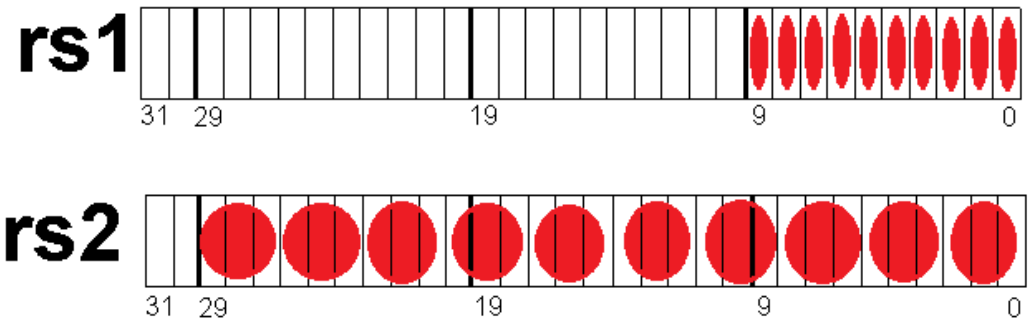

Figure 15: CMUL7Loperation registers

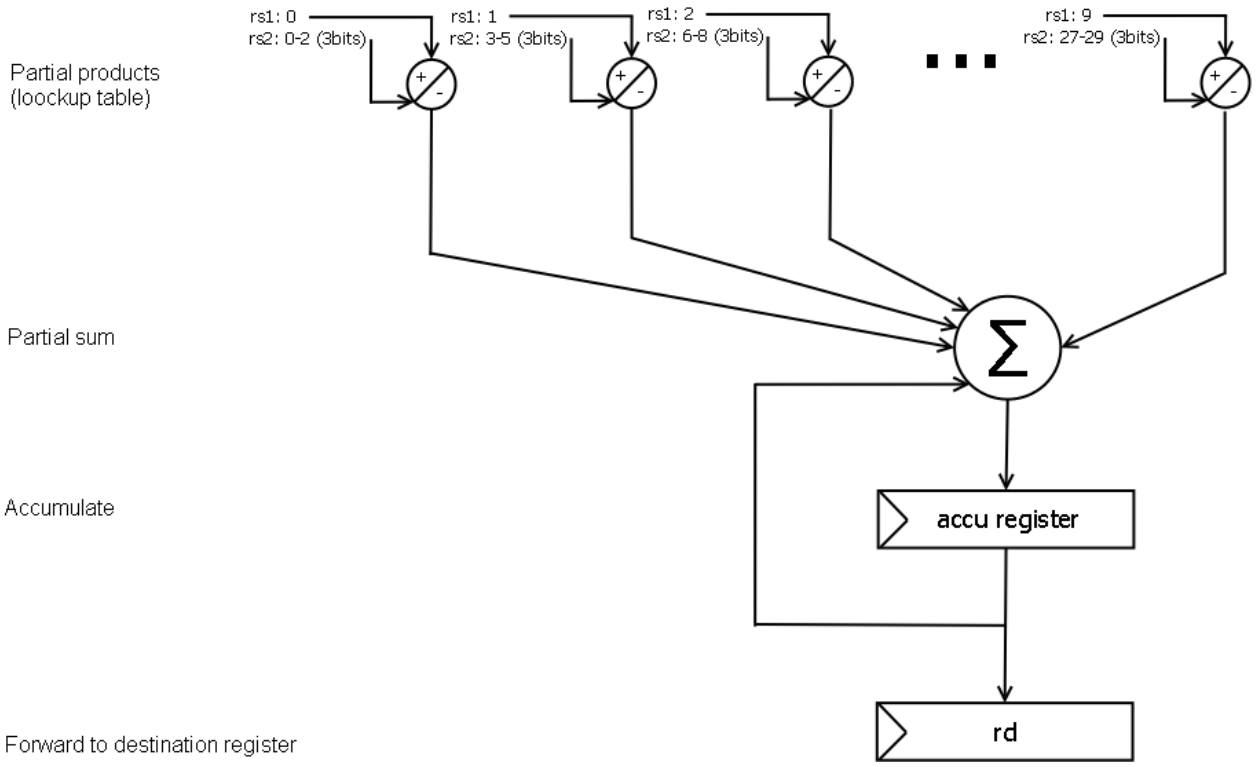

Figure 16: CMUL7L operation flow example

Operation (for CMUL7M)

prod[31:0] = rs1[19]*rs2[29:27] + rs1[18]*rs2[26:24] + rs1[17]*rs2[23:21] + rs1[16]*rs2[20:18] + rs1[15]*rs2[17:15] + rs1[14]*rs2[14:12] + rs1[13]*rs2[11:9] + rs1[12]*rs2[8:6] + rs1[11]*rs2[5:3] + rs1[10]*rs2[2:0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

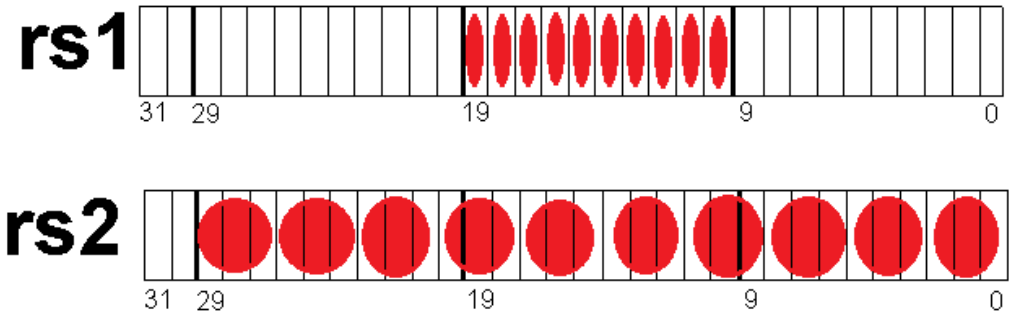

Figure 17: CMUL7M operation registers

Operation (for CMUL7U)

prod[31:0] = rs1[29]*rs2[29:27] + rs1[28]*rs2[26:24] + rs1[27]*rs2[23:21] + rs1[26]*rs2[20:18] + rs1[25]*rs2[17:15] + rs1[24]*rs2[14:12] + rs1[23]*rs2[11:9] + rs1[22]*rs2[8:6] + rs1[21]*rs2[5:3] + rs1[20]*rs2[2:0]

result[63:0] = prod[31:0] + %asr20[31:0]

%asr20[31:0] = result [31:0]

rd = result [31:0]

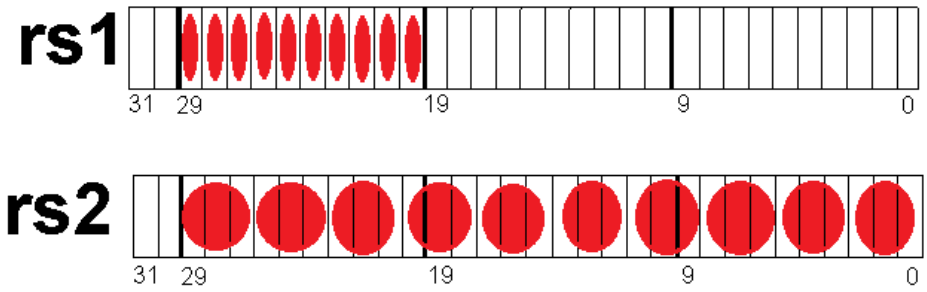

Figure 18: CMUL7U operation registers

7.3- Modifying the pipeline

The main task for this project has been modifying the pipeline to integrate the new CMUL block without interfering with the rest of the pipeline.

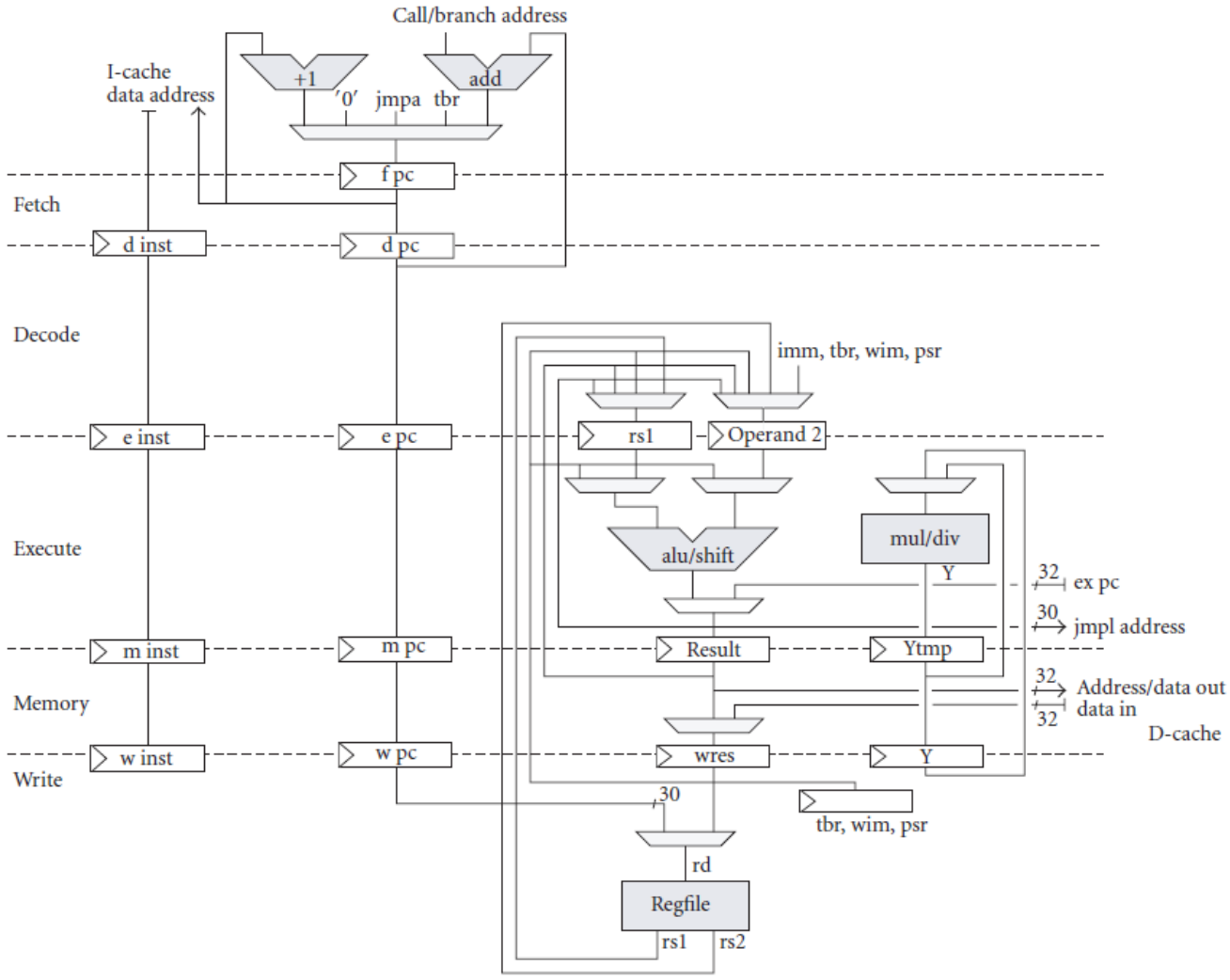

Figure 19: Unmodified pipeline of SPARC V8 processor

Special care must be taken with data and structural hazards. Part of the pre-existing hazard handling architecture can be utilized (because the result of the CMUL operation is propagated through the pipeline as any other result, which can use the built in mechanism regardless of what operation originally provided that result).

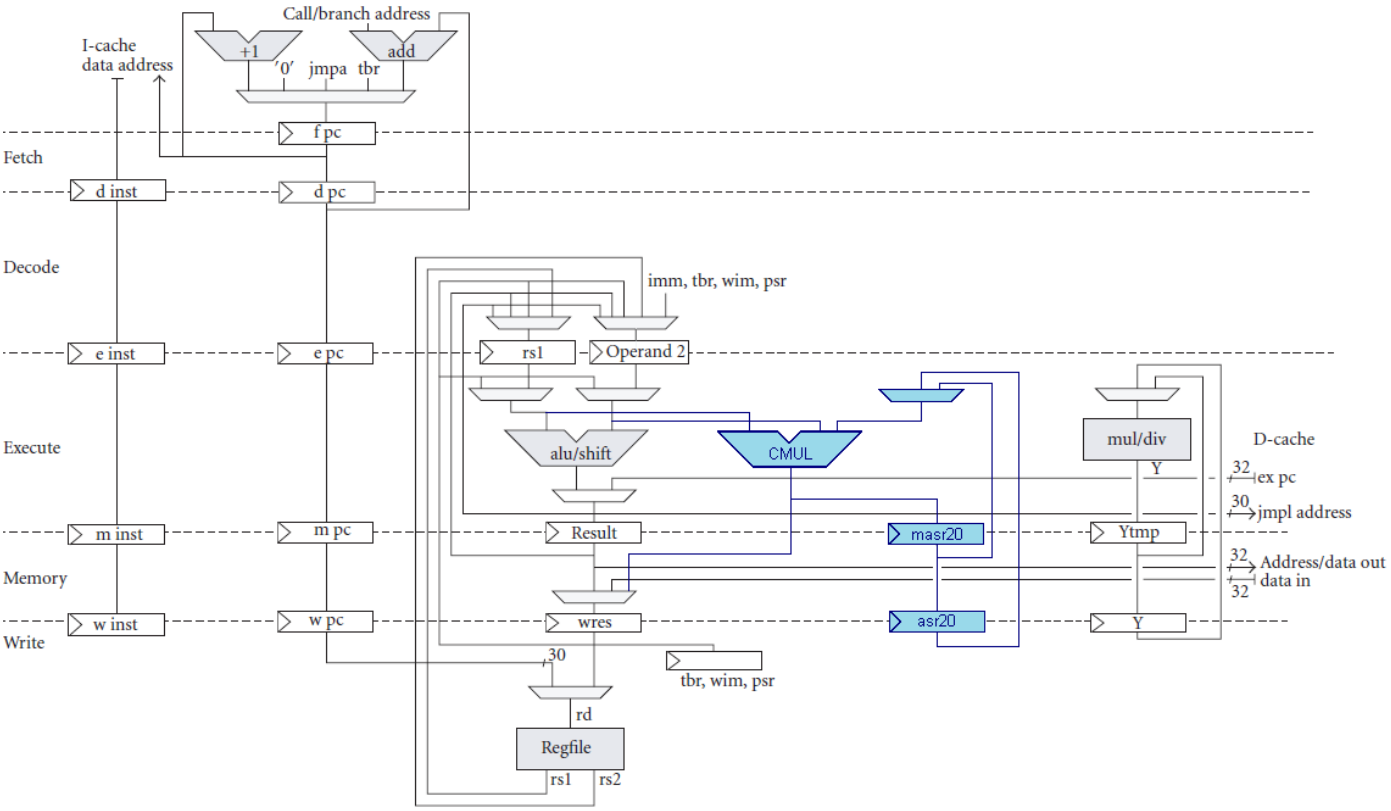

Figure 10: CMUL extension of the pipeline

However data protection (and forwarding) for the %asr20 accumulator register must be taken care of. For this the Graz University CIS extension can be a good resource, as some of its instructions use the %asr18 register in a similar manner. Their implementation can't be reused 100%, but is a good starting point.

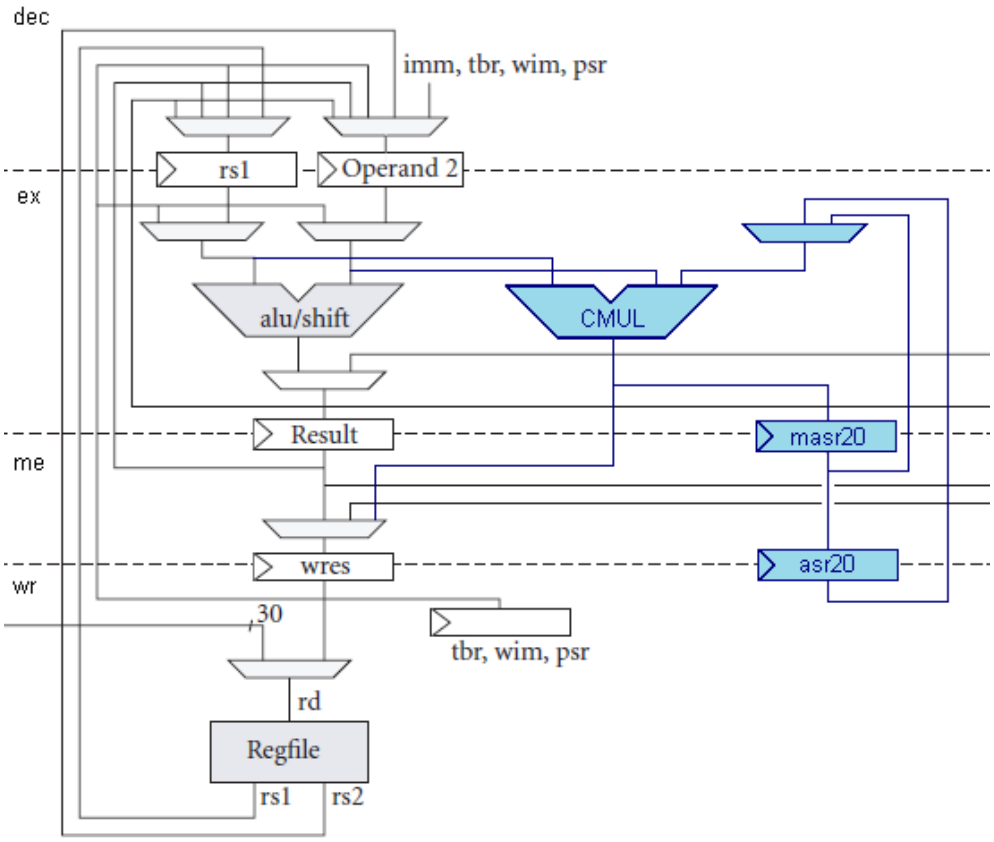

Figure 11: Zoom on the CMUL modifications to the pipeline

The original idea didn't include two intermediate asr20 registers (memory and writing) but only one. However it became apparent that both are need to allow bypassing of the accumulator in some cases, as shown by the multiplexer in the upper right hand of figure 11.

Also note that the output of the CMUL block is not inserted into the "result" register during the execute stage (which was the original idea) but during the memory stage. It was also suggested to forward it directly to the write stage, but this would not profit of the hazard protection mechanism built into the pipeline.

The CMUL executes in only one clock cycle. However the SPARC architecture can only store into two processor registers at the same time, not enough for the CMUL instruction. That means that if the next instruction after the CMUL operation operates on the same registers a one cycle stalling could be introduced into the pipeline by the hazard protection mechanism. The CMUL modification has not taken care of this, as it is expected that it is already taken care of by the original pipeline implementation. This assumption should be tested during debugging..

8- VHDL files modification

8.1- New VHDL files added to LEON

cmul.vhdl

This is the CMUL block.

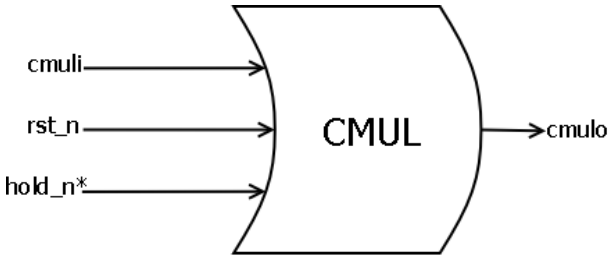

Figure 1: Cmul block schematic

The hold_n signal is not used because all CMUL operations are 1 cycle only. If multicycle operations are implemented this signal will be used together with start and hold signals in the processors pipeline.

Cmuli is of custom type cmul_in_type and consist of the following records:

```
cmuli.op1    : std_logic_vector  (31 downto 0);   -- operand 1
cmuli.op2    : std_logic_vector  (31 downto 0);   -- operand 2
cmuli.opsel  : std_logic_vector  (2  downto 0);   -- select which CMUL instruction
cmuli.asr20  : std_logic_vector  (31 downto 0);   -- 32 bits accumulator in the asr20 register
```

Cmulo is of custom type cmul_out_type and consist of the following records:

```
result  : std_logic_vector  (31 downto 0);   -- 32 bits result
icc     : std_logic_vector  (3   downto 0);   -- Update ICC if there is overflow
```

All the instructions are implemented in this block, which is then included inside the pipeline in iu.vhd which gives the appropriate inputs and expects the correct outputs from the CMUL block.

The current cmul instructions are so simple that the operations are hardcoded inside the block (in a similar way to a loockup table, without doing any real arithmetic math). Other implementations can be tested without having to change any code outside of cmul.vhd.

A self –contained vhdl testbench was used to test the correct functionality of the isolated block. Obviously this didn't test for timing constraints after synthesis, but made sure the logical implementation was correct.

The timing should be tested not in isolation, but once the CMUL block is part of the whole pipeline, due to the influence of external factors. The internal functioning of the block is so simple no timing constraints are likely to happen.

cmul_config.vhdl

At the moment the configuration options for the CMUL extension are very limited. However the configuration file has been integrated from the beginning to make future expansions easier.

Right now the only configuration option is the cmul_config flag, which can be set to none to disable the CMUL expansion all together (then the Leon will act as a normal Leon2FT processor).

Other declarations in the configuration package are constants used by the CMUL block and shouldn' t be changed by the user, but they allow customization for the designer.

Specifically, the quantization encoding constants can be changed here.

8.2- OLD LEON VHDL FILES THAT NEED TO BE MODIFIED

sparc.vhdl

This file includes the all the instruction definitions for the processor.

They are defined as constants which correspond to the op3 field of every opcode (in case of arithmetic instructions of format 3, like the CMUL instructions).

The hex encoding of each instruction is taken from the table shown in figure 1.

```
CMUL2     = "101100"   (Hexadecimal 2C, op3[5:4] = 2,  op3[3:0] =  C)
CMUL3U    = "001001"   (Hexadecimal 09, op3[5:4] = 0,  op3[3:0] =  9)
CMUL3L    = "011001"   (Hexadecimal 19, op3[5:4] = 1,  op3[3:0] =  9)
CMUL4U    = "001101"   (Hexadecimal 0D, op3[5:4] = 0,  op3[3:0] =  D)
CMUL4L    = "011101"   (Hexadecimal 1D, op3[5:4] = 1,  op3[3:0] =  D)
CMUL7U    = "101101"   (Hexadecimal 2D, op3[5:4] = 2,  op3[3:0] =  D)
CMUL7M    = "101110"   (Hexadecimal 2E, op3[5:4] = 2,  op3[3:0] =  E)
CMUL7L    = "101111"   (Hexadecimal 2F, op3[5:4] = 2,  op3[3:0] =  F)
```

iu.vhdl

The pipeline is modified here. New signals are added (cmul inputs and output signals, plus write_accumulator signals to control the writing to the asr20 accumulator register).

iface.vhdl

Two new data types, cmul_in_type and cmul_out_type, have been added to the interface list so that they can be recognized by other files in the project. These are the inputs and outputs of the cmul block.

Makefile

The new vhdl files must be added and linked correctly during the make process.

9- HOW TO ADD NEW OPCODES INTO THE BINUTILS ASSEMBLER

Once the latest binutils package has been downloaded, this are the files that MAY have to be modified to add new opcodes into the Sparc architecture specification.

Once modified, binutils can be built and compiled to have a new toolset that recognizes the new opcode instructions.

9.1- FILES THAT HAVE TO BE MODIFIED

binutils/opcodes/sparc-opc.c

This is the most important file to be modified. It contains the opcodes definition for all the instructions.

All opcode definition must follow one of the existing formats (these formats are defined in binutils/include/opcode/sparc.h, but unless we are adding new formats of our own we shouldn't modify that file). The basic SparcV8 specification has 3 different formats.

Arithmetic expressions like the ones in this project use format 3 and are defined in the following manner:

```
{
        "opcode_name",
        F3(2, opcode_address (hex),  immediate_field_flag),
        F3(~2, ~opcode_address (hex), ~immediate_field_flag),
         "op1_address, op2 _address, dest_address", 0, v8
},
```

If a single opcode can be used with different addressing method for its operands, it must be declared for each case.

For example the declaration of the Leon instruction UMUL accepts three different addressing modes:

```
{ "umul",       F3(2, 0x0a, 0), F3(~2, ~0x0a, ~0)|ASI(~0),        "1,2,d", 0, v8 },
{ "umul",       F3(2, 0x0a, 1), F3(~2, ~0x0a, ~1),          "1,i,d", 0, v8 },
{ "umul",       F3(2, 0x0a, 1), F3(~2, ~0x0a, ~1),          "i,1,d", 0, v8 },
```

This is the most frequent declaration for arithmetic instructions, which usually only have those addressing methods for the operands:

```
"1,2,d"
"1,i,d"
"i,2,d"
```

Where 1 means rs1, 2 means rs2, i means immediate address of 13 bits, and d means rd. Other more complex instructions have fancier addressing modes.

Note that the flag immediate field flag is set to 1 when one of the operand registers is addressed with an immediate constant.

Also note that ASI indicates the ASI (Address Space Identifier) field of a format 3 instruction (see SparcV8 specification page 10 for information about this register, although understanding it is not really needed just to modify the opcodes).

IMPORTANT: All declarations of the same opcode must be defined sequentially. If we declare them out of order it will build and compile with no problem, but it may crash later during runtime without any clue of what's going on.

For example:

```
{ "umul",     F3(2, 0x0a, 0), F3(~2, ~0x0a, ~0)|ASI(~0),     "1,2,d", 0, v8 },
{ "smul",     F3(2, 0x0b, 0), F3(~2, ~0x0b, ~0)|ASI(~0),     "1,2,d", 0, v8 },
{ "umul",     F3(2, 0x0a, 1), F3(~2, ~0x0a, ~1),          "1,i,d", 0, v8 },
{ "smul",     F3(2, 0x0b, 1), F3(~2, ~0x0b, ~1),          "1,i,d", 0, v8 },
```

The previous declaration will give no warning whatsoever, but the programs created with such binutils could have unexpected behaviors. This should be avoided at all cost. Instead we should do:

```
{ "umul",     F3(2, 0x0a, 0), F3(~2, ~0x0a, ~0)|ASI(~0),     "1,2,d", 0, v8 },
{ "umul",     F3(2, 0x0a, 1), F3(~2, ~0x0a, ~1),          "1,i,d", 0, v8 },
{ "smul",     F3(2, 0x0b, 0), F3(~2, ~0x0b, ~0)|ASI(~0),     "1,2,d", 0, v8 },
{ "smul",     F3(2, 0x0b, 1), F3(~2, ~0x0b, ~1),          "1,i,d", 0, v8 },
```

binutils/include/opcode/sparc.h

This file defines the different formats of instructions.

The basic Sparc implementation has three formats. Unless we are adding an additional format we should not modify this file.

However it is useful to look at it in order to understand some of the more esoteric opcode declarations in binutils/opcodes/sparc-opc.c

gas/config/tc-sparc.c
gas/config/tc-sparc.h

Macros and type definitions for the whole Sparc architecture.

Complex projects like the Graz University CIS extension may have to slightly modify this files, but for simple projects that don't add new types they shouldn't be changed.

opcodes/sparc-dis.c

Sparc disasembler.

This doesn't need to be modified if binutils is being modified for test purposes only.

ld/configure.tgt

According to the comments in the file itself:

```
# This is the linker target specific file.
# This is invoked by the autoconf generated configure script.
# Putting it in a separate shell file lets us skip running autoconf when modifying target specific
# information.

# This file switches on the shell variable ${targ}, and sets the following shell variables:
#  targ_emul               name of linker emulation to use
#  targ_extra_emuls        additional linker emulations to provide
#  targ_extra_libpath      additional linker emulations using LIB_PATH
#  targ_extra_ofiles       additional objects needed by the emulation
#  NATIVE_LIB_DIRS  library directories to search on this host (if we are a native or sysrooted linker)
```

Only the variables used by sparc must be modified. The modified binutils patch from Graz university is again a great reference for this.
http://www.iaik.tugraz.at/content/research/vlsi/archive/isec/downloads/

Basically sparc*-*-elf, which is just targ_emul=elf32_sparc by default, must be modified to an if statement that checks if we are using a leon processor.

ld/makefile.in

esparcleon.o must be added to the makefile.

9.2- OTHER FILES THAT MUST BE ADDED TO THE BINUTILS DIRECTORY

The following two files are specific to the Sparc Leon architecture. They will not be included in a default binutils download. Other Sparc architectures (like Lynx) are suported by the basic binutils, but not Leon.

The best solution would be to download the binutils patch from the Graz University LEON CIS project. That patch has this Leon files modified for their own CIS project on a Leon2 board, and they can easily be modified for any Leon processor.

http://www.iaik.tugraz.at/content/research/vlsi/archive/isec/downloads/

ld/scripttempl/sparcleon.sc

Linker script of sparcleon.

ld/emulparams/sparcleon.sh

This short script is copied directly from the CIS project.

In the CMUL project case the only change made was substituting the format from elf32-sparc to elf-sparc. This is not very important, but it will have to be the same as the TARGET we use when building binutils.

10- BUILDING THE BINUTILS AND GCC TOOLCHAIN

10.1 - HOW TO BUILD BINUTILS FROM SCRATCH

First binutils has to be downloaded to some directory. Let's call it $WORKDIR. Assuming we have the version x.yy of binutils:

1. The binutils sourcecode will be unziped into $WORKDIR/binutils-x.yy
2. The binutils will be modified to incorporate new opcodes as described in previous sections
3. Binutils will be built into $WORKDIR/objdir as explained in this section

Assuming the binutils have already been downloaded and modified into $WORKDIR/binutils-x.yymod:

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
## BUILDING BINUTILS

# Some variables export to make life easier
export binutils_version=2.14
export WORKDIR=/home/hugo/leonbuild
export BINDIR=$WORKDIR/binutils-$binutils_version

# Create objdir where we are going to build everything
cd $WORKDIR
rmdir -r objdir
mkdir objdir
export OBJDIR=$WORKDIR/objdir

# Run configure script, make and install binutils for the target architecture (sparc-elf)
cd $BINDIR
./configure --prefix=$OBJDIR --target=sparc-elf
make
make install
```

10.2 - HOW TO BUILD GCC FROM SCRATCH

Although the tests for the new instructions are going to be done in assembly language only, most of the test scripts are c scripts with embedded assembly code.

That's why we also need a compatible gcc compiler. In theory it should be possible to download a precompiled crosscompiler for leon sparc and link it with our custom binutils, but it does not seem to work properly.

That's why the simplest solution is to download the gcc sourcecode and build it ourselves with a sparc target architecture and linking it with our custom binutils from the beginning.

Note that we are not modifying the gcc compiler and we are not making it understand the new instructions. We are building gcc from its source code exactly as downloaded from the gnu project website. The only difference is that we use different binutils when building it, so it won't complain when he encounters the new instructions.

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
## BUILDING GCC

export gcc_version=4.8.2
export WORKDIR=/home/hugo/leonbuild
export GCCDIR=$WORKDIR/gcc-$gcc_version
cd $GCCDIR

##
# Downloading MPFR, GMP and MPC (all needed to build GCC)
# This is  usually done by running the ./contrib/download_prerequisites script from $GCCDIR
# But the script provided by the default gcc download has two problems:
# 1-  The version variables $MPFR, $GMP and $MPC  are usually not up to date
# 2-  The download links point to ftp/gnu, but are blocked by the ESA firewall
#      We have to use other mirror links.

# This script has to be modified with the instructions provided next
# Or alternatively the instructions can be run directly in the terminal
# This will download MPFR, GMP and MPC and create 3 folders in the $GCCDIR where all other
# sources are
# Failing to download this files properly will produce very obscure build messages

MPFR=mpfr-3.1.2
GMP=gmp-5.1.3
MPC=mpc-1.0.1

wget http://www.mirrorservice.org/sites/ftp.gnu.org/gnu/mpfr/$MPFR.tar.bz2 || exit 1
tar xjf $MPFR.tar.bz2 || exit 1
ln -sf $MPFR mpfr || exit 1

wget http://www.mirrorservice.org/sites/ftp.gnu.org/gnu/gmp/$GMP.tar.bz2 || exit 1
tar xjf $GMP.tar.bz2  || exit 1
ln -sf $GMP gmp || exit 1

wget http://www.mirrorservice.org/sites/ftp.gnu.org/gnu/mpc/$MPC.tar.gz || exit 1
tar xzf $MPC.tar.gz || exit 1
ln -sf $MPC mpc || exit 1

rm $MPFR.tar.bz2 $GMP.tar.bz2 $MPC.tar.gz || exit 1
```

```
## No stack overflow checking
## Because sparc-as has suspicious code and the stack protector will prevent us from building gcc
export CFLAGS="-fno-stack-protector"


## VERY VERY VERY IMPORTANT
# The gcc configure script should NEVER be run directly from the gcc directory.
# Doing ./configure from the gcc directory will give no warning
# But this is not supported by the GNU project
# And will cause many problems when running make.
# Instead the script should be run from another folder we will call objdir
# For simplicity this will be the objdir where we previously built the sparc binutils.

export OBJDIR=$WORKDIR/objdir
cd $OBJDIR

$GCCDIR/configure --target=sparc-elf --prefix=$OBJDIR  --enable-languages=c --with-build-time-
tools=$OBJDIR/sparc-elf/bin  --disable-bootstrap --disable-libssp

## CONFIGURATION OPTIONS EXPLAINED

## -- target
## could be sparc-elf, sparc-v8-elf, or sparc32-elf, as long it is the same used for binutils

## -- with-build-time-tools
## Tell it where to find the modified binutils tools we compiled previously
## If the binutils were correctly built this folder should contain at least the elf-sparc version of:
## as, ld, ar, nm, ranlib, strip, objdump

## --disable-bootstrap
## By default gcc is built with a 3 steps bootstrap which uses itself as a compiler to compile itself.
## This is supossed to work also for crosscompilers
## But it doesn't. So we just disable it.

## disable-libssp
## disable the gcc  runtime stack smashing protector
## the sparc-as and other sparc binutils tools have many suspicious stack operations
## If we don't disable libssp
## Will cause the building of gcc to abort when it tests for possible buffers overflows

make
make install

## Now we should have gcc in objdir/bin and in objdir/sparc-elf/bin with all the other binutils
```

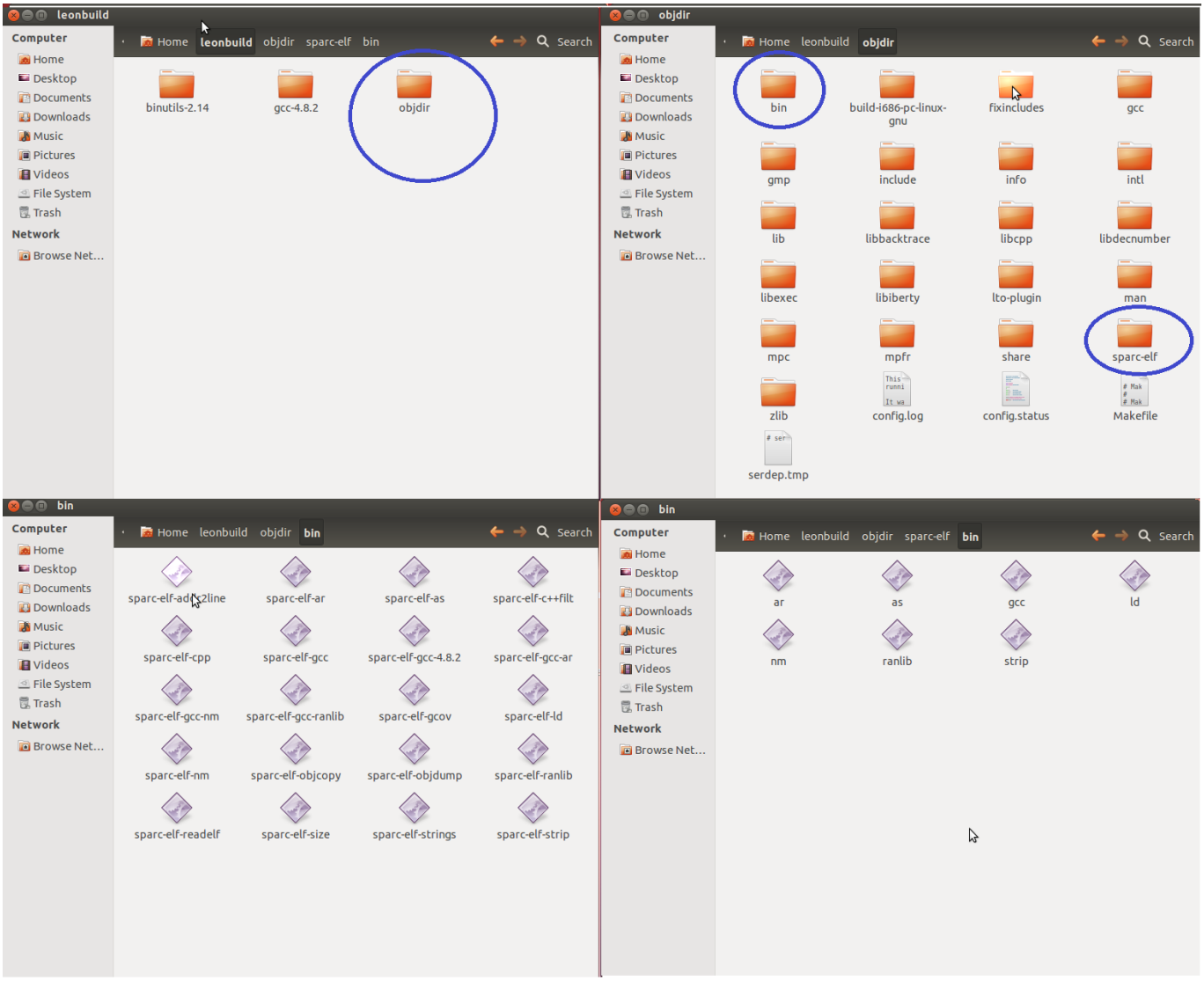

Figure 20: Example of successfully built binutils and gcc

11- RESULTS

The increase is size of the vhdl files is not considered important, but is discussed here for the sake of completeness. Obviously factors like the effects of the modification on area and efficiency are considered much more critical.

The resulting VHDL code after adding the modifications is not much bigger than the original LEON. The following table shows all changes in number of code lines:

Vhdl block	Original	CMUL extension	
iu.vhdl	3669	4075	376
iface.vhdl	301	312	11
sparc.vhdl	1104	1115	10
makefile	240	250	10
cmul.vhdl	-	256	256
cmul_config.vhdl	-	75	75

Figure 21: Comparison of vhdl files before and after modification

The total increase in lines was only 739, which is negligible compared to the total size of the vhdl project. The only file with a significant increase in size was the iu.vhdl due to the complexity of integrating the instructions into the pipeline. This corresponds to a 10,25% increase in the number of lines in the iu.vhdl file.

However the total size of the vhdl files is 2,6 MB in both cases, which shows how negligible this small increase in code is for the whole process.

Note that the original code hasn't been changed despite temptations to do so, in order to keep this comparison more realistic. The original code lacks comments and should be rewritten in a cleaner style in order to make it understandable by mere mortals.

The new code has not followed this principle of unreadability and includes such things as comments and relevant variable names.

The modified processor has been tested with the testbench for the original LEON2FT to make sure the changes do not interfere with the operation of the basic LEON.

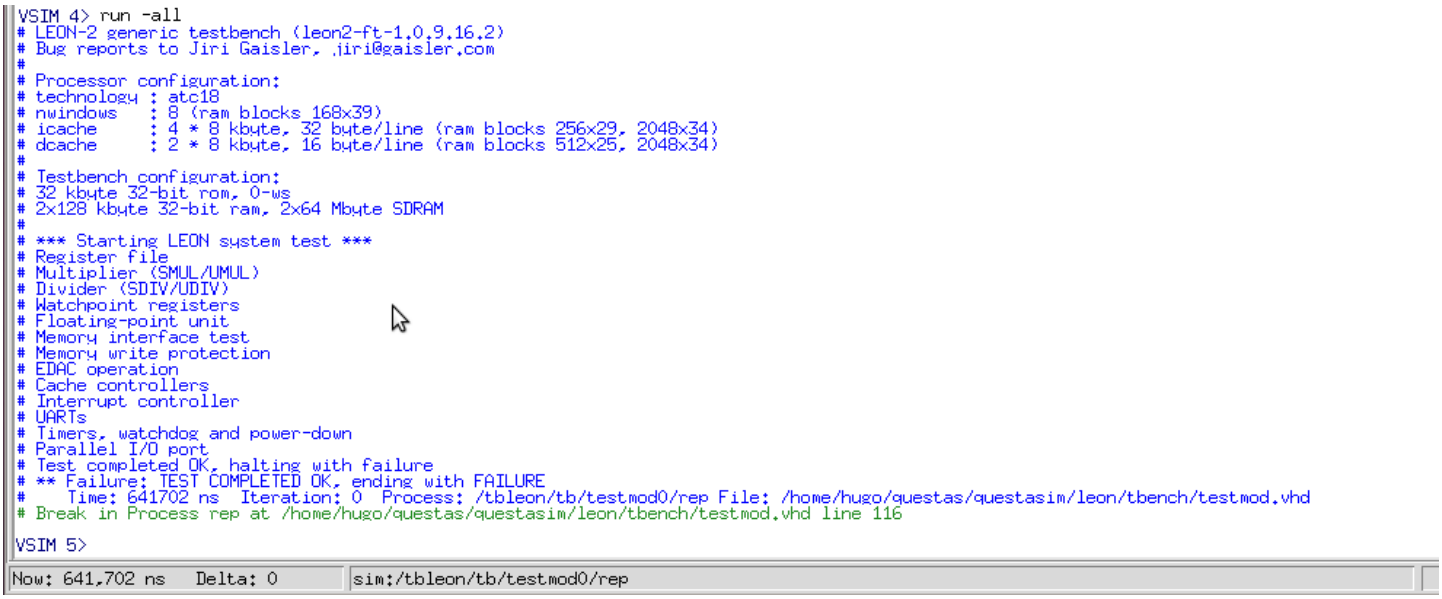

Figure 22: Simulation to ensure that the modifications don't interfere with proper operation.

Synthesis tools were also used to perform an analysis. The target device was a Spartan 6 FPGA. The results showed that device utilization, area and power consumption were almost not altered by the modifications.

This are the results of the synthesis on a Spartan 6 FPGA using Xilinx ISE:

Slice Logic Utilization:			
Number of Slice Registers:	125 out of	18,224	1%
Number used as Flip Flops:	125		
Number used as Latches:	0		
Number used as Latch-thrus:	0		
Number used as AND/OR logics:	0		
Number of Slice LUTs:	227 out of	9,112	2%
Number used as logic:	225 out of	9,112	2%
Number using O6 output only:	213		
Number using O5 output only:	0		
Number using O5 and O6:	12		
Number used as ROM:	0		
Number used as Memory:	0 out of	2,176	0%
Number used exclusively as route-thrus:	2		
Number with same-slice register load:	2		
Number with same-slice carry load:	0		
Number with other load:	0		
Slice Logic Distribution:			
Number of occupied Slices:	89 out of	2,278	3%

Number of MUXCYs used:	12 out of	4,556	1%
Number of LUT Flip Flop pairs used:	235		
Number with an unused Flip Flop:	112 out of	235	47%
Number with an unused LUT:	8 out of	235	3%
Number of fully used LUT-FF pairs:	115 out of	235	48%
N of slice register lost to control set restrictions:	0 out of	18,224	0%

IO Utilization:

Number of bonded IOBs:	188 out of	232	81%

Specific Feature Utilization:

Number of RAMB16BWERs:	0 out of	32	0%
Number of RAMB8BWERs:	0 out of	64	0%
Number of BUFIO2/BUFIO2_2CLKs:	0 out of	32	0%
Number of BUFIO2FB/BUFIO2FB_2CLKs:	0 out of	32	0%
Number of BUFG/BUFGMUXs:	3 out of	16	18%
Number used as BUFGs:	3		
Number used as BUFGMUX:	0		
Number of DCM/DCM_CLKGENs:	0 out of	4	0%
Number of ILOGIC2/ISERDES2s:	0 out of	248	0%
Number of IODELAY2/IODRP2/IODRP2_MCBs:	0 out of	248	0%
Number of OLOGIC2/OSERDES2s:	0 out of	248	0%
Number of BSCANs:	0 out of	4	0%
Number of BUFHs:	0 out of	128	0%
Number of BUFPLLs:	0 out of	8	0%
Number of BUFPLL_MCBs:	0 out of	4	0%
Number of DSP48A1s:	0 out of	32	0%
Number of ICAPs:	0 out of	1	0%
Number of MCBs:	0 out of	2	0%
Number of PCILOGICSEs:	0 out of	2	0%
Number of PLL_ADVs:	0 out of	2	0%
Number of PMVs:	0 out of	1	0%
Number of STARTUPs:	0 out of	1	0%
Number of SUSPEND_SYNCs:	0 out of	1	0%

Peak Memory Usage: 193 MB

The test scripts for the original LEON2FT were also run showing that everything was working fine.

12- FURTHER DEVELOPEMENT

The main goal of the project was to study the feasibility of the idea. This was not only proved, but a vhdl prototype was fully implemented.

The results showed that it is possible to add new instructions with minimum impact on the processor. The size of the processor was not affected very much, making the idea of adding custom instructions quite reasonable.

It was tested and proved that the modifications with the processor don't interfere with the processor functioning normally. The new CMUL instructions were superficially tested, but a deeper testing would be needed to ensure 100% proper functioning under any circumstance. Such a thorough testing was beyond the resources and timeframe of this project.

Now that the binutils chain has been rebuilt to recognize the new instruction further testing would be required. Due to the lack of testing on the new implementation, bugs are likely to occur.
Several special cases should be tested:

1. Single CMUL operations handle wrong inputs in the appropriate manner.
2. Consecutive CMUL operations don't cause data hazards.
3. CMUL instructions just before or after other instructions on the same operands/destination registers don't cause data hazards.

The two last cases are the most critical ones, because the biggest point of failure are data dependencies and data bypassing. This has been implemented for CMUL instructions and the %asr20 accumulator register inside the iu. However due to the complex nature of the pipeline (and chaotic vhdl implementation of the LEON2FT processor) errors are likely to sneak in.

Due to all the above factors, the CMUL expansion should be subject to merciless testing and debugging processes.

After the implementation is 100% bug free, comparative analysis on efficiency could be carried out.

Reference documents

[1] Oracle, "Sparc T4 Architecture," 2011. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/sparc-architecture-2011-1728132.pdf.

[2] P. A. a. F. P.Guironnet de Massas, "On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration," 2007. [Online]. Available: http://www.hindawi.com/journals/vlsi/2007/028686/abs/.

[3] Tu Graz, "Performance Evaluation of Instruction Set Extensions for Long Integer," 2007. [Online]. Available: http://www.iaik.tugraz.at/content/research/vlsi/archive/isec/leon2-cis/.

[4] SPARC, "The Sparc V8 specification," 1992. [Online]. Available: http://www.sparc.com/standards/V8.pdf.