Report Number: WUCSE-2005-25

2005-05-30

Processor Generator v1.3 (PG13)

Eduard V. Kotysh and Patrick Crowley

This project presents a novel automated framework for microprocessor instruction set exploration that allows users to extend a basic MIPS ISA with new multimedia instructions (including custom vector instructions, a la AltiVec and MMX/SSE). The infrastructure provides users with an extension language that automatically incorporates extensions into a synthesizable processor pipeline model and an executable instruction set simulator. We implement popular AltiVec and MMX extensions using this framework and present experimental results that show significant performance gains of customized microprocessor.

2005-25

Processor Generator v1.3 (PG13)

Authors: Kotysh, Eduard V.; Crowley, Patrick

May 30, 2005

Abstract: This project presents a novel automated framework for microprocessor instruction set exploration that allows users to extend a basic MIPS ISA with new multimedia instructions (including custom vector instructions, a la AltiVec and MMX/SSE). The infrastructure provides users with an extension language that automatically incorporates extensions into a synthesizable processor pipeline model and an executable instruction set simulator. We implement popular AltiVec and MMX extensions using this framework and present experimental results that show significant performance gains of customized microprocessor.

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

PROCESSOR GENERATOR v1.3 (PG13)

by

Eduard V. Kotysh B.S. Applied Science

Prepared under the direction of Dr. Patrick Crowley

A project presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

PROCESSOR GENERATOR v1.3 (PG13)

by Eduard V. Kotysh

ADVISOR: Dr. Patrick Crowley

May, 2005

Saint Louis, Missouri

This project presents a novel automated framework for microprocessor instruction set exploration that allows users to extend a basic MIPS ISA with new multimedia instructions (including custom vector instructions, a la AltiVec and MMX/SSE). The infrastructure provides users with an extension language that automatically incorporates extensions into a synthesizable processor pipeline model and an executable instruction set simulator. We implement popular AltiVec and MMX extensions using this framework and present experimental results that show significant performance gains of customized microprocessor.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Custom microprocessors have gained large popularity in the industry, as well as in academia. A small amount of custom logic can make a large improvement in performance, but has historically required a costly custom processor design. Not only does it take a considerate amount of money to develop the desired functionality, but it is also a difficult and timely process to perform.

The solution is to provide a friendly customization environment that allows easy modification of an existing microprocessor to suit the desired requirements. For this purpose, companies like Tensilica Inc. [4] have developed instruction extension languages to aid in the tailoring process and achieve desired functionality. Their tools provide an automated and easy way to implement custom extensions; however, the tools are usually expensive and require learning of a new language.

I have developed an open-source infrastructure for microprocessor customization that allows users to extend a basic MIPS ISA with new instructions, including custom vector instructions, a la AltiVec or MMX/SSE [1][2]. The infrastructure contains a synthesizable MIPS IV-compatible [3] processor written in VHDL and an automated extensibility framework written in JAVA. The framework provides users with a low-level VHDL-like extension language that automatically incorporates new extensions into a synthesizable processor pipeline model and an executable simulator. This project addresses all the issues with current extension frameworks: it is a free open-source project; it is completely automated (integrated interpreter, generator, assembler and simulator); and its language is based on VHDL, so computer architecture developers don't have to learn a new extension language.

This infrastructure is an experimental project, developed as a learning tool for graduate and undergraduate level computer engineers, as well as professional microprocessor developers. Basic knowledge of VHDL and computer architecture is required to be able to define new extensions and take advantage of available components to exploit parallelism.

This report is organized as follows: section 2 describes the high-level design of PG13 infrastructure explaining the function of each component within the framework; section 3 goes into implementation details of how each component works; section 4 presents synthesis results of the Spartacus base microprocessor with two embedded extensions; section 5 describes the testing techniques employed and the results observed during testing; section 6 draws conclusions from the developed system; finally, section 7 contains references and people who made PG13 possible.

2. HIGH-LEVEL DESIGN

Figure 1 captures the high level view of PG13 infrastructure. Components placed inside the processor generator are the integrated framework pieces stitched together for automation. User can invoke those parts by simply pressing a button on the GUI interface.

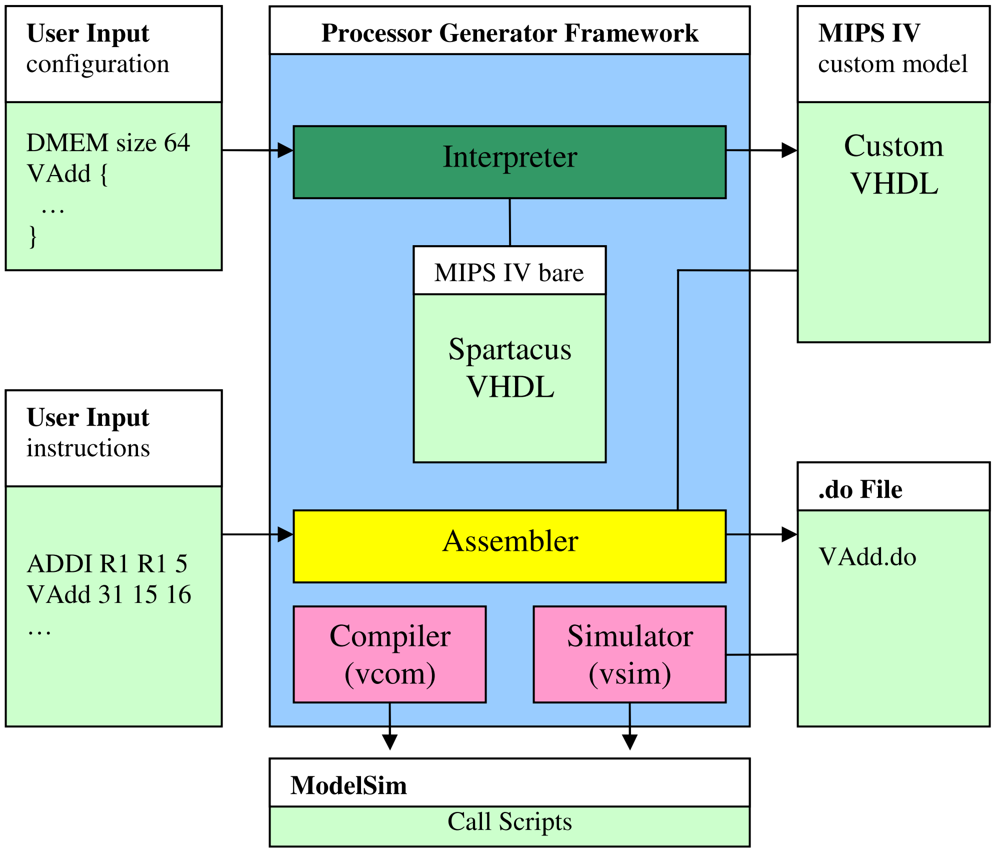


Figure 1: PG13 Infrastructure

Entire framework consists of 5 major components: Base MIPS IV processor, Interpreter, Assembler, VHDL Compiler batch, and Simulator batch.

Base MIPS IV Processor

For this project, I have developed a synthesizable model of a single-threaded, 5-stage pipeline MIPS IV microprocessor in VHDL. It contains:

- 35 MIPS-compatible instructions [3]
- 2 reconfigurable Register Files (32-bit and 64-bit)
- 3 ALUs (32-bit, 64-bit, and vector ALU)
- Reconfigurable instruction memory
- Reconfigurable dual-ported data memory

Interpreter

Interpreter is the most complex part of the framework. On a high level, it serves 2 primary functions:

1) Takes user-defined configuration parameters (such as Register File size, clock period, etc.) and modifies the base processor according to given specifications.
2) Takes the new extension created by the user, parses it, and translates it to corresponding VHDL blocks and adds these blocks to the processor.

Assembler
I wrote a custom assembler for the Spartacus processor that converts human-readable MIPS assembly into binary code for the microprocessor's Instruction Register (IR). It then incorporates the produced binary into a .do simulation testbench. It's important to note that my assembler can also translate the instructions of the new extensions that just have been created, since it has access to the produced custom VHDL processor.

Compiler
The integrated compiler batch script can call the ModelSim compiler (vcom) with preferred arguments, such as synthesis check, explicit conflict resolution, 93' syntax support and others. The compilation batch is spawned in a new process and involves all generated VHDL files that are compiled with proper dependencies. This step has to be done before simulation is started; otherwise, the architecture in simulation will be outdated.

Simulator
Integrated simulator batch script calls ModelSim simulator (vsim) with provided .do testbench and automatically configured architecture. It spawns a separate process that starts ModelSim simulation and runs for as long as specified by the user. All the signals, components and waves are available for viewing and debugging, as in regular ModelSim simulation mode.

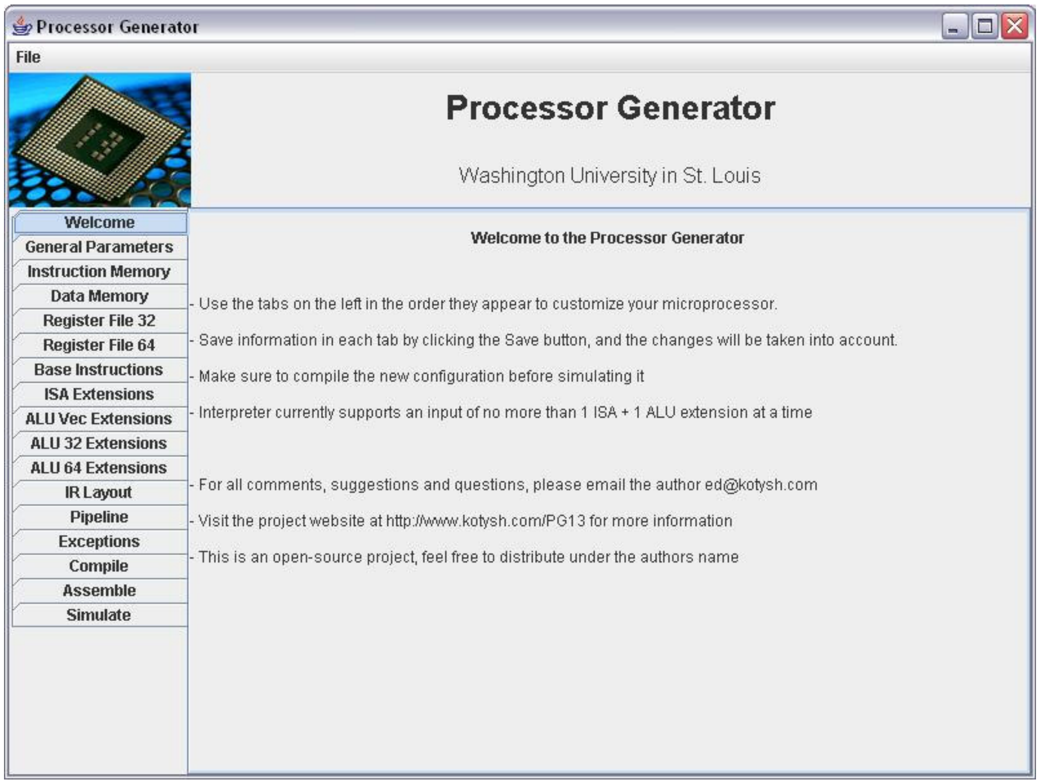
Data Flow
The following chain of events will take place in the process of generating a new custom processor using PG13 infrastructure:

1. The PG13 framework GUI is started by the user
2. User sets desired configuration parameters on the microprocessor
3. User inputs the implementation of the new extension
4. Interpreter parses user input and builds the new VHDL model of the processor
5. Compiler verifies correctness of the new VHDL model
6. User writes desired MIPS assembly instructions
7. Assembler converts them into binary representation and builds a .do testbench
8. Simulator takes the .do testbench and the user-specified running time and creates a simulation model for the new microprocessor
9. Compiler verifies the synthesis compatibility of the new processor

3. IMPLEMENTATION

At the heart of the infrastructure lies the GUI framework that provides an easy, interactive way to access, use and modify the processor components in an automated fashion.

Written entirely in JAVA swing, it runs on any platform as a jar executable.



Welcome to the Processor Generator

- Use the tabs on the left in the order they appear to customize your microprocessor.
- Save information in each tab by clicking the Save button, and the changes will be taken into account.
- Make sure to compile the new configuration before simulating it
- Interpreter currently supports an input of no more than 1 ISA + 1 ALU extension at a time
- For all comments, suggestions and questions, please email the author ed@kotysh.com
- Visit the project website at http://www.kotysh.com/PG13 for more information
- This is an open-source project, feel free to distribute under the authors name

Figure 2: Framework Interface

As can be seen in Figure 2, the entire framework is subdivided into step-by-step configurations (called tabs) to ensure a safe and orderly flow of events that is intuitive to the user. Each tab panel serves a unique function and passes on its results using shared memory down the chain of events.
Following is the description of each tab panel and its implementation:

General Parameters
This section lets you personalize the processor by doing the following:
- Naming the top-level entity
- Setting the clock period (i.e. 20ns)
- Setting the edge of the clock (rising/falling)
- Adding a reset signal to the controller
- Specifying what processor components should be added to the build, such as additional registers, vector ALU, etc.

When the Save button is clicked, all the parameter configurations that user specified are encoded into a file and passed along to the Interpreter. For general parameters, such as clock period, the interpreter creates a copy of the old VHDL configuration, seeks into the appropriate place to make a change and writes the new parameters. In order to include only the desired processor components, the Interpreter creates only requested component mappings and declarations at the top-level entity.

Instruction/Data Memory
Current instruction and data memories are implemented as double arrays of size word (32 bit).

For instruction memory, the user can change the size of the memory (how many blocks) but cannot change the size of the block, for the Instruction Register is based on a 32-bit architecture and the microprocessor is currently single-threaded; therefore, no parallelism can be achieved from fetching two instructions at once. The future work section addresses this thought in more detail, leaving it for the upcoming groups to develop multi-threading support for this project.

For data memory, the user is allowed to change the size of the memory, as well as the size of the block in memory providing maximum flexibility to achieve speedup. Data memory is made dual-ported for optimal performance of the vector memory operations.

Reset signal is optional for simulation, but strongly recommended for a synthesizable model to avoid massive creation of feedback MUXes by the synthesizer.

Register Files
I have implemented two register files (32 and 64-bit) to support MMX extensions using 64-bit MMX registers. Both register files are dual-ported.

User can specify the size of each register file, which will determine the number of register mappings generated within the regFile entity.

Base Instructions
In this tab panel, user can check or uncheck the base MIPS instructions he or she wants implemented in the microprocessor. This allows saving resources and utilizing only the required functionality at the base level. Currently, 35 base MIPS instructions are implemented.

ISA Extensions
This section lets the user create custom instruction extensions, such as AltiVec, 3DNOW! and MMX extensions [1][2]. The user writes the implementation of the extension he wants, expressing what he wants this instruction to do at every stage of the pipeline. The implementation of each stage is written in a VHDL- or Verilog-like syntax, which is easy and familiar to computer architecture developers. For more information on the syntax of the language and how to write custom extensions in general, refer to the PG13 manual.
After the implementation is entered, the parser checks for syntax errors and interpreter translates and incorporates the implementation into the microprocessor. When interpreter is done, the user can view the produced VHDL and interpretation parameters for correctness check or debugging purposes.

ALU Extensions

In order to exploit maximum parallelism, I provide the ability to create custom ALU operations that can be later used in the implementation of an ISA extension. This becomes very beneficial for the implementation of custom vector instructions, where several adds or multiplies can be performed in one cycle. Consider the following example of a custom vector add instruction:

vecadd8_mmx (memLoc dest, mmxReg r1, mmxReg r9)

This instruction vector adds 64-bit MMX registers in 32-bit chunks and stores the results in memory:

dest[0] = r1(63:32) + r9(63:32)
dest[1] = r1(31:0) + r9(31:0)

dest[2] = r2(63:32) + r10(63:32)
dest[3] = r2(31:0) + r10(31:0)

dest[14] = r8(63:32) + r16(63:32)
dest[15] = r8(31:0) + r16(31:0)

To execute this instruction in N clock cycles (where N is the length of the vector, in this case 8) we will need to perform two 32-bit adds in one cycle. ALU Extension environment lets you create a custom operation for the vector ALU that will do just that.

IR Layouts
Currently, there are two fixed Instruction Register layouts that are MIPS-compatible, used for regular operations and special operations.

SPECIAL	Rs	Rt	Rd	0	OPCODE
000000	00000	00000	00000	00000	000000
31 26	25 21	20 16	15 11	10 6	5 0

Figure 3: IR: OP Layout

SPECIAL	Rs	Rt	Immediate/Offset
000000	00000	00000	0000 0000 0000 0000
31 26	25 21	20 16	15 0

Figure 4: IR: SP Layout

Figure 3 depicts the OP (OPCODE-active field) layout for instructions that do not operate on immediates or offsets. SP (SPECIAL-active field) layout shown in Figure 4 is used for Store, Load, Branch and immediate addressing instructions.

Considering the timeframe of this project, these layouts were made predetermined and cannot be altered by user, for MIPS backwards compatibility will be ruined by changing any of the IR fields. However, the user must still carefully consider these layouts before creating a custom extension to not exceed the maximum number of operands and fit data within the available range of the operand bits; hence, this tab panel was included.

Pipeline
This section allows the user to create new debug signals and set them to monitor each stage of the pipeline. First, user provides the definition of the signal, which is added to the processor implementation and the .do testbench for visual simulation. Then, user wires the created debug signal in any stage of the pipeline to any other signal, be it register file output or memory address. The wiring information is also converted into VHDL and added to the microprocessor by the interpreter.

Exceptions
In this tab, the user has quick access to the list of trap codes and exceptions that might occur during the simulation for convenience.

Compiler
This tab allows users to choose the options to supply to the vcom compiler script for microprocessor compilation. Below are currently available options:

- Check for compliance with synthesis rules
- Resolve resolution conflicts in favor of explicit functions
- Enable support for VHDL 1076-1993
- Print the source line with error messages

After the options have been selected, the compiler spawns a separate command line process, where it sets the ModelSim environmental variables, navigates to the vhdl directory and calls the vcom compiler with appropriate arguments on all of the VHDL files generated. This step must be performed before simulation is called on the generated files.

Assembler
In this window, the user specifies the output .do file and writes assembly instructions in MIPS format that he/she wants the processor to execute during the simulation. Assembler then runs a syntax check on the written assembly and, if passed, translates assembly into binary representation, encapsulates it into the .do testbench and writes everything out to the user-specified file.

Simulator
It is important to note that before proceeding with simulation, the user is required to execute the compilation step to not get out of synch and have a valid .do testbench to run the simulation on.

In this section, the user first chooses the .do testbench file to run the simulation on and then specifies the amount of time to run the simulation. After this information is set, the simulator will once again set the ModelSim environmental variables, navigate to the proper directory and call the vsim simulator with appropriate arguments on the top-level entity of the generated microprocessor.

Spartacus Processor
I have developed a MIPS IV-compatible microprocessor based on classic 5-stage pipeline architecture (Fetch, Decode, Execute, Memory, Writeback).

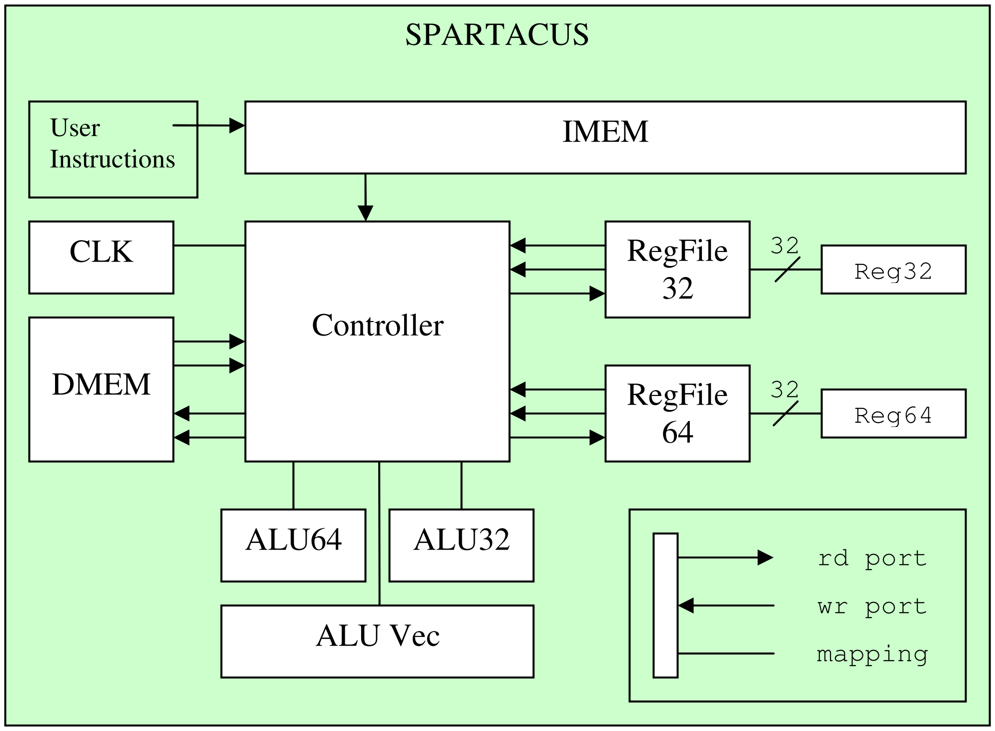


Figure 5: Spartacus Base Processor

Components
As seen from Figure 5, it contains controller (the top-level entity) and 10 components: clock, instruction memory, data memory, 2 register files, 32-bit register, 64-bit register, and 3 ALUs. Controller reads instructions from instruction memory via a single read port (32 bits/cycle). Data memory is dual-ported, and therefore, the controller can write data via two write ports (two 32-bit/cycle) and read data via two

read ports (two 32-bit/cycle). Controller has maps to 3 ALUs (32-bit, 64-bit and a vector ALU) and a clock. It's also connected to two register files via two read ports and one write port each. Each register file deploys 32 maps to the register component. Clock is symmetrical and has an adjustable period. A separate clocked process is used in the controller to latch signals instead of creating registers out of them. The default state machine design is used, where one process is dedicated solely to assigning the current state on every rising clock edge and another process is based on sensitivity list and assigns the next state based on the current state it's in.

Base Instruction Set
The base of the Spartacus core contains 35 instructions shown in Figure 6.

Function	Opcode	Description
OP_ADD	100000	add
OP_ADDU	100001	add unsigned
SP_ADDI	001000	add immediate
SP_ADDIU	001001	add immediate unsigned
OP_AND	100100	and
SP_ANDI	001100	and immediate
SP_BEQ	000100	branch on equal
SP_BGTZ	000111	branch on >0
SP_BLEZ	000110	branch on <=0
SP_BNE	000101	branch on not equal
OP_DADD	101100	64bit add
OP_DADDU	101101	64bit add unsigned
SP_DADDI	011000	64bit add immediate
SP_DADDIU	011001	64bit add imm unsigned
OP_DDIV	011110	64bit divide
OP_DDIVU	011111	64bit divide unsigned
OP_DIV	011010	divide
OP_DIVU	011011	divide unsigned
OP_DMULT	011100	64bit multiply
OP_DMULTU	011101	64bit multiply unsigned
OP_DSLL	111000	64bit shift left logical fixed
OP_DSLLV	010100	64bit shift left logical variable
OP_DSRA	111011	64bit shift right arith fixed
OP_DSRAV	010111	64bit shift right arith variable
OP_DSRL	111010	64bit shift right logic fixed
OP_DSRLV	010110	64bit shift right logic variable
OP_DSUB	101110	64bit subtract
OP_DSUBU	101111	64bit subtract unsigned
SP_LW	100011	load word
SP_SW	101011	store word
OP_MFHI	010000	move from 32bit HI register
OP_MFLO	010010	move from 32bit LO register
OP_MFDHI	010001	move from 64bit HI register
OP_MFDLO	010011	move from 64bit LO register

Figure 6: Base Instruction Set

This set of instructions contains standard logic, arithmetic, branch and memory operations, with support for 32-bit and 64-bit architecture as well.

Implemented Extensions
The following extensions have been implemented and tested using PG13 framework:

Function	**Description**	**Type**
Vecadd32_mem	vector adds 32-bit values in memory	AltiVec
Vecadd64_mmx	vector adds values of mmx registers	Mixed
Vecadd16_hybrid	vector adds values of mmx registers in 16-bit chunks and stores the results into memory	Mixed
Vecld	vector indexed load	AltiVec
Vecmax	vector maximum extraction	AltiVec
Maskmove	streaming store using byte mask	MMX/3DNOW!
PavgB	packed averaging of bytes of mmx regsiters	MMX/3DNOW!

Figure 7: Implemented Extensions

The functionality of this entire set of instructions can be added to the base MIPS processor in less than 20 minutes. Most of these (as indicated in parenthesis) were taken from AltiVec or MMX manual and implemented to show how easy it is to create non-standard extensions using this framework. In addition, they provide good examples to study for new users of PG13.

Interpreter
Interpreter is the central part of the PG13 framework, what actually does most of the work and performs translation from user-given implementation into VHDL code on the microprocessor.

Keywords
The interpreter employs a translation language that converts user's definition of the custom extension to VHDL code for the microprocessor. The language employs special keywords to simplify the declarations for the user. These keywords can be used to declare particular steps to perform at each pipeline stage, to specify a vector extension and vector length, or simply to decrement/increment a certain signal. For full list of special keywords and their functionalities, please refer to the PG13 manual documentation.

Two-stage Architecture
Interpreter works in two stages. In the first stage, it will analyze the declaration of user's extension and identify all the operands and its types, the type of IR layout to use, and find and reserve a free opcode for that instruction. In the second stage, the interpreter will analyze the implementation scope of the instruction. It will set the loop variables if any (only for vector instructions), go through each pipeline declaration stage, perform translation of signal names to standardize with the processor and deploy pipeline code on the microprocessor in VHDL, taking care of the sensitivity list and code dependencies in microprocessor VHDL.

Example
In order to better understand how the interpreter works and what it does, consider the following simple example and all will become clear:

Suppose a user would like to create a vector-add operation in memory that adds memory locations starting at “dest” and “s1” in a vector fashion with a vector length of “s2”.

Figure 8 shows the correct user implementation of this instruction:

```
operation vecadd32_mem (memLoc dest, memLoc s1, imm32 s2)
  loop s2
  {
      Decode
        dmemory_address1 <= x"000000" & "000" & dest;
        dmemory_address2 <= x"000000" & "000" & s1;

      Execute
        alu32_operation <= OP_ADD;
        alu32_source1 <= dmemory_data_out1;
        alu32_source2 <= dmemory_data_out2;

      Memory
        dmemory_write_en1 <= '1';
        dmemory_address1 <= x"000000" & "000" & dest;
        dmemory_data_in1 <= alu32_result;

      Writeback
        decrement dest 1;
        decrement s1 1;
   }
```

Figure 8: Sample Vector-add Extension

The first stage of interpreter will parse out the name of the operation (vecadd32_mem), reserve a free and valid opcode for this instruction by looking in the database of currently used opcodes in the microprocessor, identify the source and destination operands (dest, s1, s2) and their types (memory location, memory location, and 32-bit immediate) and choose the proper IR layout for this instruction (SP layout because of the immediate, refer to Figures 3 and 4). Depending on the layout, different source/destination signal names will have to be substituted in order to suit VHDL implementation at the microprocessor level.

The second stage will work on the scope of the instruction implementation. Interpreter will see that this is a vector instruction by parsing in the loop keyword and know that s2 is the length of the vector (number of times to perform the implementation scope) and that inside the brackets is the implementation of the vector instruction. At this point, the interpreter produces the parameter file, shown in Figure 9, that can be later viewed by the user to trace what happened for educational, debugging or simply curiosity reasons.

```
func:name=vecadd32_mem;
dest:name=instr_rs_out;
dest:type=memLoc;
s1:name=instr_rt_out;
s1:type=memLoc;
s2:name=instr_immoff_out;
s2:type=imm32;
dest:width=32;
s1:width=32;
s2:width=32;
func:code=111111;
func:type=SP;
loop_en <= '1';
loop_cnt <= instr_immoff_out;
```

Figure 9: Extension Parameters

Next, the interpreter will analyze each pipeline stage implementation, add necessary signals to the sensitivity list, perform translation of names “dest”, “s1”, “s2” and others to valid signal names, interpret keyword functions like “decrement” into valid registered vhdl code and so on. After all the formalities have been translated, the produced VHDL implementation is added to the microprocessor files in appropriate places (i.e. the controller, ALU, etc.)

Similarly, the ALU extensions are translated by the interpreter, and then users can use them in the declaration of their ISA extension to exploit more parallelism.

Limitations
PG13 framework allows users to implement any kind of extensions, as long as they can be implemented considering the current microprocessor resources and architecture. In other words, the users are limited to the use of signals and components available in the Spartacus MIPS IV processor. However, the processor core was specifically designed to accommodate all currently known MIPS, MMX and AltiVec extensions and written for easy scalability, and therefore, experienced users are encouraged to become developers and extend the processor implementation as the needs grow.

Assembler
The goal of the Assembler is to take human-readable MIPS assembly instructions and convert them into binary representations that Spartacus understands during simulation. Hence, it produces the binary, encapsulates it into the .do simulation testbench and passes the file to the simulator.

Basically, the Assembler needs to know two things before it can produce the binary for the instruction: the opcode of the instruction and the IR layout it uses. After that, it knows exactly how much to padd each field and where in the 32-bit binary string to place each operand. The opcode for instruction is looked up by the Assembler in the opcode database file, which already contains the new ISA extensions, which lets the user create a new extension and immediately after write assembly for it. The IR layout information is passed to the Assembler by the Interpreter. After these two things are known, the Assembler runs a syntax check on the given code and notifies the user of the instruction errors including errors in the number of operands supplied, the type of the operands, unknown operations and etc. If syntax check passes on all of the instructions, the Assembler constructs the 32-bit binary representation of each instruction, deposits them to the instruction memory signal one by one in the .do testbench, and passes the .do file to the simulator.

Executing Scripts
These scripts are essentially child processes that are spawned by my framework calling external executables provided by ModelSim with user-defined parameters. I chose to use two of those executables: vcom and vsim that are described below. Environmental setup is performed (PATH setting) before each batch execution.

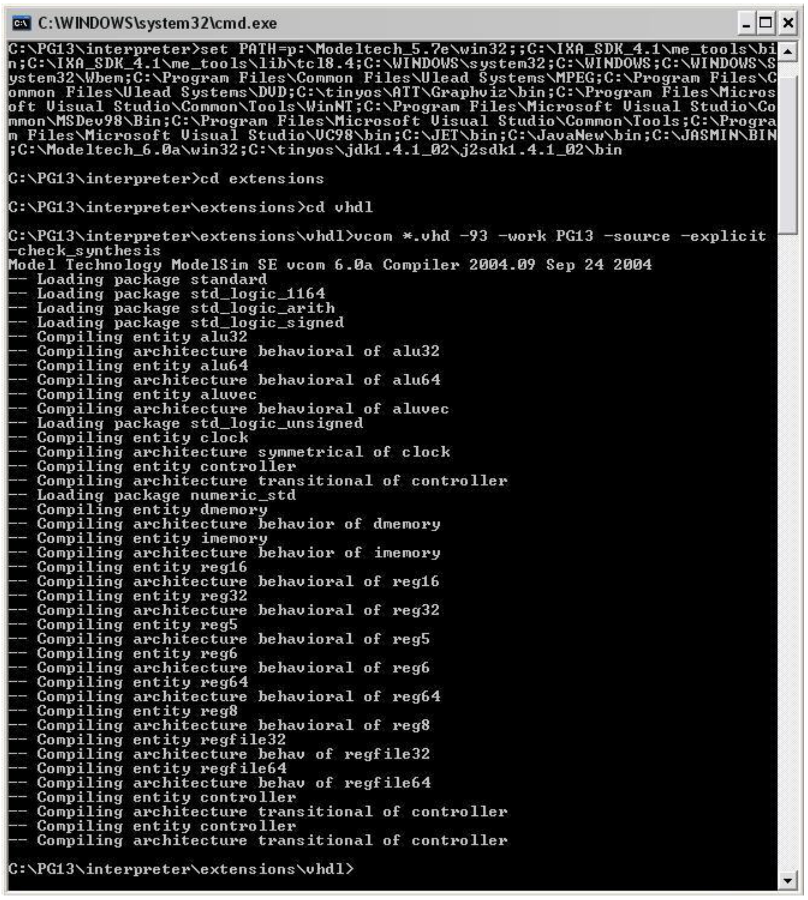
Compilation Batch
There are several compilation batch files that are called depending on which arguments and options were selected by the user in the GUI framework environment. For example, if user selected synthesis check and explicit conflict resolution options, the following batch will be executed:

```
::Set ModelSim 5.7e Environment varibles
set LM_LICENSE_FILE=regNumber@licenseServer
set MGLS_LICENSE_SERVER=regNumber@licenseServer
set PATH=p:\Modeltech_5.7e\win32;%PATH%

:: Fire Up ModelSim 5.7e
cd extensions
cd vhdl
vcom *.vhd -93 -work PG13 -source -explicit -check_synthesis
```

Figure 10: Compilation Batch

The first three lines set the environment and license for the ModelSim software; therefore, the user is required to have ModelSim installed on his/her workstation, or perform compilation and simulation steps outside the framework. Next, the batch navigates out of the framework directory to the VHDL directory and calls vcom compiler to run over all VHDL files in the directory specifying the top-level entity and options provided by the user.



```
C:\PG13\interpreter>set PATH=p:\Modeltech_5.7e\win32;;C:\IXA_SDK_4.1\me_tools\bi
n;C:\IXA_SDK_4.1\me_tools\lib\tcl8.4;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\S
ystem32\Wbem;C:\Program Files\Common Files\Ulead Systems\MPEG;C:\Program Files\C
ommon Files\Ulead Systems\DVD;C:\tinyos\ATT\Graphviz\bin;C:\Program Files\Micros
oft Visual Studio\Common\Tools\WinNT;C:\Program Files\Microsoft Visual Studio\Co
mmon\MSDev98\Bin;C:\Program Files\Microsoft Visual Studio\Common\Tools;C:\Progra
m Files\Microsoft Visual Studio\VC98\bin;C:\JET\bin;C:\JavaNew\bin;C:\JASMIN\BIN
;C:\Modeltech_6.0a\win32;C:\tinyos\jdk1.4.1_02\j2sdk1.4.1_02\bin

C:\PG13\interpreter>cd extensions

C:\PG13\interpreter\extensions>cd vhdl

C:\PG13\interpreter\extensions\vhdl>vcom *.vhd -93 -work PG13 -source -explicit
-check_synthesis
Model Technology ModelSim SE vcom 6.0a Compiler 2004.09 Sep 24 2004
-- Loading package standard
-- Loading package std_logic_1164
-- Loading package std_logic_arith
-- Loading package std_logic_signed
-- Compiling entity alu32
-- Compiling architecture behavioral of alu32
-- Compiling entity alu64
-- Compiling architecture behavioral of alu64
-- Compiling entity aluvec
-- Compiling architecture behavioral of aluvec
-- Loading package std_logic_unsigned
-- Compiling entity clock
-- Compiling architecture symmetrical of clock
-- Compiling entity controller
-- Compiling architecture transitional of controller
-- Loading package numeric_std
-- Compiling entity dmemory
-- Compiling architecture behavior of dmemory
-- Compiling entity imemory
-- Compiling architecture behavior of imemory
-- Compiling entity reg16
-- Compiling architecture behavioral of reg16
-- Compiling entity reg32
-- Compiling architecture behavioral of reg32
-- Compiling entity reg5
-- Compiling architecture behavioral of reg5
-- Compiling entity reg6
-- Compiling architecture behavioral of reg6
-- Compiling entity reg64
-- Compiling architecture behavioral of reg64
-- Compiling entity reg8
-- Compiling architecture behavioral of reg8
-- Compiling entity regfile32
-- Compiling architecture behav of regfile32
-- Compiling entity regfile64
-- Compiling architecture behav of regfile64
-- Compiling entity controller
-- Compiling architecture transitional of controller
-- Compiling entity controller
-- Compiling architecture transitional of controller

C:\PG13\interpreter\extensions\vhdl>
```

Figure 11: Compilation Batch Execution

The entire batch call is depicted in Figure 11. This process lets users see if there were errors encountered during the compilation.

Simulation Batch
A very similar process is adopted in this batch. The vsim application is called with the assembled .do testbench file on a given processor architecture. Vsim will spawn its own child process, in which ModelSim application is loaded, opening the signals, structures, and waves of the microprocessor and running the simulation steps outlined in the .do file. The user has access to all debugging windows and resources as in normal ModelSim simulation.

4. SYNTHESIS

Synthesis has been performed using: Synplicity Compiler and Xilinx Technology Mapper version 7.3. The target platform of choice was Xilinx Virtex-E part XCV2000E package FG1156, at speed -6 [5]. This platform has the following characteristics: 43,200 Logic cells, 80x120 CLB array, 2.5M system gates, and 655,360 block RAM bits.

The synthesized design of PG13 microprocessor utilizes:
16866.72 LUTs (78%), 9/16 Global clock buffers (56%), 1/56 Block RAMs (1%), and 9056 Register bits (42%). Considering this is not optimized, it leaves a good amount of room for implementations of the new custom extensions by users.

```
Starting Points with Worst Slack
****************************************************
                               Starting
Instance                       Reference       Time      Slack
                               Clock
----------------------------------------------------------------
instr_special_out[0]           controller|clk  0.449     -8.886
instr_special_out_fast[3]      controller|clk  0.449     -5.895
instr_special_out_fast[5]      controller|clk  0.449     -5.895
instr_special_out_fast[1]      controller|clk  0.449     -5.726
instr_special_out_fast[2]      controller|clk  0.449     -5.726
instr_opcode_out[4]            controller|clk  0.449     -5.455
instr_opcode_out[5]            controller|clk  0.449     -5.378
instr_opcode_out[2]            controller|clk  0.449     -5.329
instr_opcode_out[3]            controller|clk  0.449     -5.329
current_state[1]               controller|clk  0.449     -5.280
================================================================
Ending Points with Worst Slack
****************************************************
```

Figure 12: Worst Slack Report

As can be seen from generated report shown in Figure 12, the path with the worst slack happens at signal that captures the special opcode field of the instruction. Most of the 35 implemented instructions use the special IR layout and hence the special field of the instruction is passed to other entities a lot (i.e. ALU32, ALU64, ALU vector) and used within controller in several stages of the pipeline to determine the operations needed to perform on that instruction.

With the requested frequency of 500 MHz and inferred worst slack of -8.886, the performance was estimated at around 92MHz, leaving plenty of performance room for new customized instructions, as can be seen from Figure 13.

```
Performance Summary
*******************
Worst slack in design: -8.886
                                      Requested    Estimated    Estimated
Starting Clock                        Frequency    Frequency    Period       Slack
------------------------------------------------------------------------------------
controller|clk                        500.0 MHz    98.6 MHz     10.146       -8.146
current_state_inferred_clock[4]       500.0 MHz    389.9 MHz     2.565       -0.565
System                                500.0 MHz    91.9 MHz     10.886       -8.886
====================================================================================
```

Figure 13: Performance Report

5. TESTING

First and foremost, I have tested each individual instruction (35 total) on the base of the Spartacus processor. I made sure each of them performed in the correct amount of time and with valid results. I have then tested multiple instructions within each section (memory accesses – stores and loads, logical operations, arithmetic operations). The final test included a set of small programs that combined cross-section instructions like register file loads and memory stores.

Each framework component was tested separately as the development of the whole infrastructure progressed. Similar to the bottom-up technique, this insured that the higher level components will function correctly because of the tested lower-level infrastructure that it depends on. Once all components were in place, the automation process was deployed that stitched the components' functionalities together into an automated fashion. Then, the automation technique was tested by creating very simple instruction extensions using the framework.

After being able to create simple instructions, I tried to implement an AltiVec vector add extension using the framework, incorporating the interpreter, compiler, assembler and simulator altogether. The next step was testing custom ALU operations and making sure they are available for use in ISA extension editor right after creation. This was a difficult step because the entire implementation and various shared configuration files must be updated before the build process of the new extension begins. After this was successful, I have experimented with several MMX and other AltiVec extensions and left them in the framework to serve as examples for new users and developers. During this testing phase, I have timed the execution of extensions like vecadd8_mmx (described in section 3.1.6) that employ custom ALU operations to exploit parallelism and made sure the performance was as expected.

Consider an example where the user would like to add two mmx registers in 32-bit data chunks and store the results to memory for a vector length of 8. To exploit parallelism, the user can use PG13 to first create a custom vector ALU operation that performes two 32-bit adds in one clock cycle (Figure 14).

```
operation VADD32 (gen64 dest, gen64 s1, gen64 s2)
   {
      dest(31 downto 0) <= s1(31 downto 0) + s2(31 downto 0);
      dest(63 downto 32) <= s1(63 downto 32) + s2(63 downto 32);
   }
```

Figure 14: Custom ALU Extension

Now that we can perform two 32-bit adds in parallel, we can include this functionality in the definition of our ISA Extension (Figure 15).

```
operation vecadd8_mmx (memLoc dest, mmxReg s1, mmxReg s2)
  loop 8
  {
      Decode

      Execute
           aluVec_operation <= OP_VADD32;
           aluVec_source1 <= regfile64_data_out1;
           aluVec_source2 <= regfile64_data_out2;

      Memory
           dmemory_write_en1 <= '1';
           dmemory_write_en2 <= '1';
           dmemory_address1 <= x"000000" & "000" & dest;
           dmemory_address2 <= x"000000" & "000" & dest+1;
           dmemory_data_in1 <= aluVec_result(63 downto 32);
           dmemory_data_in2 <= aluVec_result(31 downto 0);

      Writeback
           decrement dest 2;
           decrement s1 1;
           decrement s2 1;
   }
```

Figure 15: Custom Vecadd8_mmx Extension

As can be seen from Figure 15, we're employing the VADD32 ALU operation we just created to perform the parallel addition of 32-bits in mmx registers. Then, in Memory stage, we write both results into memory and go to the next set of registers and memory locations in the writeback stage.

The assembly for the entire operation becomes what's depicted in Figure 16. We preload the mmx registers R1-R8 with values 13, 14, …, 20. Then we preload mmx registers R9-R16 with values 333, 334, …, 340; and then, we execute our custom extension that vector adds two mmx registers in 32-bit chunks and saves the results to two 32-bit memory locations, doing so for the vector length of 8.

```
SP_DADDI R1 R1 13
SP_DADDI R2 R2 14
SP_DADDI R3 R3 15
SP_DADDI R4 R4 16
SP_DADDI R5 R5 17
SP_DADDI R6 R6 18
SP_DADDI R7 R7 19
SP_DADDI R8 R8 20
SP_DADDI R9 R9 333
SP_DADDI R10 R10 334
SP_DADDI R11 R11 335
SP_DADDI R12 R12 336
SP_DADDI R13 R13 337
SP_DADDI R14 R14 338
SP_DADDI R15 R15 339
SP_DADDI R16 R16 340
OP_vecadd8_mmx 16 R8 R16
```

Figure 16: Assembly for Customized Execution

As can be seen from the simulation capture (Figure 17) after the preloading of the mmx registers, the execution of the custom extension and getting the result takes 3200ns.

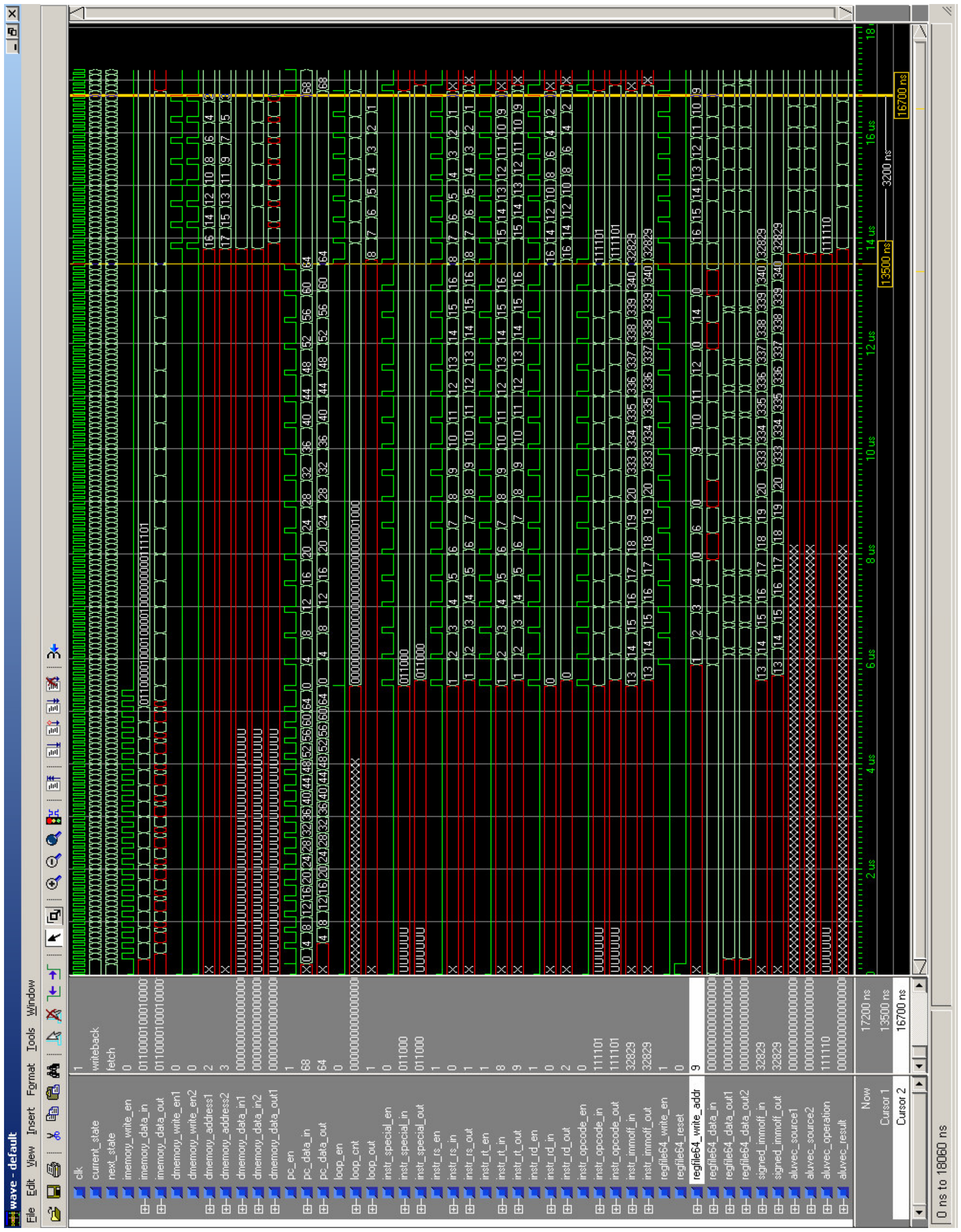


Figure 17: Performance with the vector extension

Now let’s compare what would the performance be if we didn’t design the parallel extension. The assembly code becomes what’s shown in Figure 18.

```
SP_DADDI R1 R1 13
SP_DADDI R2 R2 14
SP_DADDI R3 R3 15
SP_DADDI R4 R4 16
SP_DADDI R5 R5 17
SP_DADDI R6 R6 18
SP_DADDI R7 R7 19
SP_DADDI R8 R8 20
SP_DADDI R9 R9 333
SP_DADDI R10 R10 334
SP_DADDI R11 R11 335
SP_DADDI R12 R12 336
SP_DADDI R13 R13 337
SP_DADDI R14 R14 338
SP_DADDI R15 R15 339
SP_DADDI R16 R16 340
OP_DADD R1 R1 R9
OP_DADD R2 R2 R10
OP_DADD R3 R3 R11
OP_DADD R4 R4 R12
OP_DADD R5 R5 R13
OP_DADD R6 R6 R14
OP_DADD R7 R7 R15
OP_DADD R8 R8 R16
SP_MVDU R1 R1 0
SP_MVDL R2 R1 0
SP_MVDU R3 R2 0
SP_MVDL R4 R2 0
SP_MVDU R5 R3 0
SP_MVDL R6 R3 0
SP_MVDU R7 R4 0
SP_MVDL R8 R4 0
SP_MVDU R9 R5 0
SP_MVDL R10 R5 0
SP_MVDU R11 R6 0
SP_MVDL R12 R6 0
SP_MVDU R13 R7 0
SP_MVDL R14 R7 0
SP_MVDU R15 R8 0
SP_MVDL R16 R8 0
SP_SW 1 R1 0
SP_SW 2 R2 0
SP_SW 3 R3 0
SP_SW 4 R4 0
SP_SW 5 R5 0
SP_SW 6 R6 0
SP_SW 7 R7 0
SP_SW 8 R8 0
SP_SW 9 R9 0
SP_SW 11 R11 0
SP_SW 12 R12 0
SP_SW 13 R13 0
SP_SW 14 R14 0
SP_SW 15 R15 0
SP_SW 16 R16 0
```

Figure 18: Assembly Without Custom Extension

As expected, now that we don't have the vector instruction, in order to add the values in mmx registers, we would use the regular 64-bit adder instruction DADD and then move higher 32 bits and lower 32 bits into regular 32-bit registers, the values of which will then be stored in memory.

This imitational behaviour simulated for 15900ns (Figure 19) to achieve the same result as with the custom vector extension.

This gives us almost a 5x speedup with that particular custom extension. The even greater parallelisms can be exploited in custom vector extensions that perform data operation on smaller data chunks, such as bytes instead of 32-bit words.

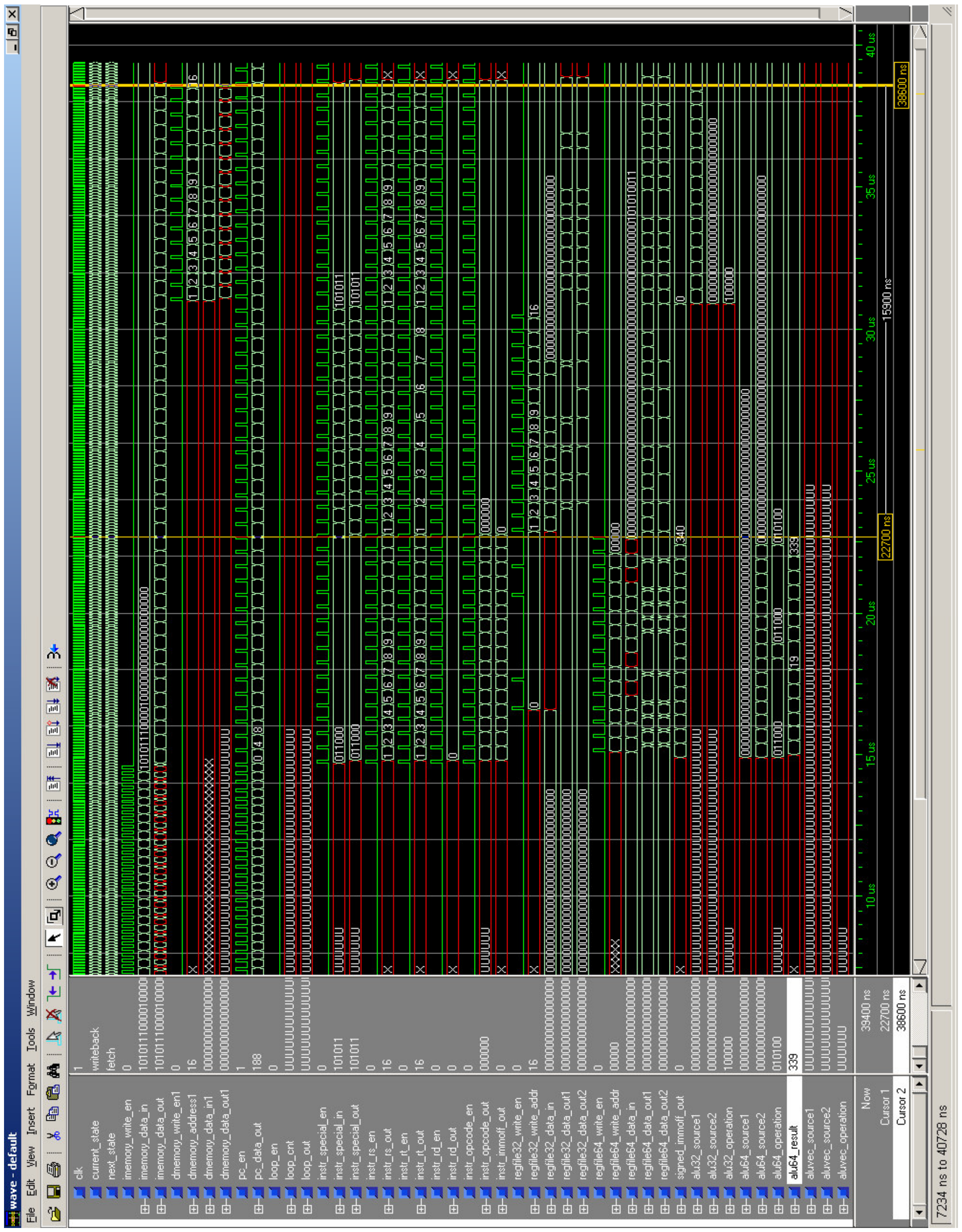


Figure 19: Performance without the vector extension

6. CONCLUSION / FUTURE WORK

I have successfully developed an infrastructure which allows users to develop custom ISA and ALU extensions in an easy environment. By providing an automated framework where users can go from specification to simulation in less than 5 minutes, this project gives maximum flexibility and functionality, saving time to dig into multiple VHDL entities or learning a new extension language. The design lets users exploit parallelism by defining their own ALU operations and employing them later in custom ISA extensions.

As much success as this project has been, there was only a fixed amount of features I could implement in a given time frame. I have tried to concentrate on the most important features that had to be implemented first, and on scalability of the design overall to let future groups add functionality with ease. Below is the main list of features I wanted, but didn't have time to implement that should be considered by future groups working on this project:

- Multithreading (currently, Spartacus is single-threaded)
- Reconfigurable block size for Instruction memory (once multithreading is done)
- A C compiler developed for this infrastructure
- More tools to specifically exploit parallelism in extensions

7. ADDITIONAL RESOURCES

For more information on the project, please visit http://www.kotysh.com/PG13.

Detailed documentation is written on the use of the entire framework and can be downloaded from the project website.

This is an open-source project, and executables, as well as source code can be downloaded from the project website and distributed under Author's name.

8. REFERENCES / THANKS

I would like to thank my advisor, Patrick Crowley, and a colleague, Kristian Georgiev, for helping with the design and implementation decisions and for making this project possible.

References

[1] AltiVec Technology Programming Interface Manual. Rev.0 1999.
AltiVec.org *The altivec information source* http://www.altivec.org

[2] AMD Extensions to the 3DNOW! and MMX Instruction Sets Manual

[3] MIPS IV Instruction Set Manual. Rev. 3.2 1995

[4] Tensilica Inc. TIE *Tensilica Instruction Extension Language*
http://www.tensilica.com

[5] Xilinx Inc. *Online specifications source* Virtex-E parts http://www.xilinx.com/