Rapid Embedded Hardware/Software System Generation

Jorgen Peddersen[†‡], Seng Lin Shee[†‡], Andhi Janapsatya[†], Sri Parameswaran[†‡]
†School of Computer Science and Engineering, The University of New South Wales
Sydney, NSW 2052, Australia
‡National Information and Communications Technology Australia (NICTA)[*]
Sydney, NSW 2052, Australia
{jorgenp,senglin,andhij,sridevan}@cse.unsw.edu.au

Abstract

This paper presents an RTL generation scheme for a SimpleScalar / PISA Instruction set architecture with system calls to implement C programs. The scheme utilizes ASIPmeister, a processor generation tool. The RTL generated is available for download. The second part of the paper shows a method of reducing the PISA instruction set and generating a processor for a given application. This reduction and generation can be performed within an hour, making this one of the fastest methods of generating an application specific processor. For five benchmark applications, we show that on average, processor size can be reduced by 30% , energy consumed reduced by 24%, and performance improved by 24%.

1. Introduction

Humans are increasingly reliant on Embedded Systems. Embedded systems are seen today in application specific equipment such as telephones, PDAs, cars, cameras etc. Embedded systems differ from general purpose computing machinery since a single application or a class of applications are repeatedly executed. Thus, processing units can be customized without compromising functionality.

The heart of an embedded system is usually implemented using either general purpose processors, ASICs or a combination of both. General Purpose Processors (GPPs) are programmable, but consume more power than ASICs. Reduced time to market, and minimized risk are factors which favor the use of GPPs in embedded systems. ASICs, on the other hand, cost a great deal to design and are non-programmable, making upgradability an impossible dream. However, ASICs have reduced power consumption and are smaller than GPPs.

Recently a new entrant called the Application Specific Instruction Set Processor (ASIP) has taken center stage as an alternative contender for implementing functionality in embedded systems. These are processors with specialized instructions, selected co-processors, and parameterized caches applicable only to a particular program or class of programs. An ASIP will execute an application for which it was designed with great efficiency, though they are capable of executing any other program (usually with greatly reduced efficiency). ASIPs are programable, quick to design and consume less power than GPPs (though more than ASICs). ASIPs in particular are suited for utilization in embedded systems where customization allows increased performance, yet reduces power consumption by not having unnecessary functional units. Programmability allows the ability to upgrade, and reduces software design time. Tools such as ASIPmeister [1], Tensilica [2], ARCtangent [3], Jazz [4], Nios [5] and SP5-flex [6] allow rapid creation of ASIPs.

The advent of tools to create Application Specific Instruction Set Processors has greatly enhanced the ability to reduce design turn–around time. Despite several efforts to the contrary [7, 8, 9, 10, 11], customization still remains an art rather than a well understood science. In addition, customization of an ASIP can take a significant amount of time.

[*]National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

In this paper, we look at a simple method of customization to speed up design turn–around time of the processor. We start with a well configured processor with a rich set of instructions. After compilation of the application and its associated libraries, we remove unwanted instructions and re-synthesize the processor. The resulting processor is now capable of running the same application, in a smaller processor with reduced power consumption, and higher speed (due to reduced clock width). The initial processor we created was based upon the SimpleScalar / PISA [12] instruction set. The selection of this particular instruction set enabled an opulent set of tools to be available.

Structure of the paper is as follows. Section 2 gives a summary of existing works on application specific processors; Section 3 provides our methodology for processor generation; Section 4 describes the SimpleScalar processor we have designed; Section 5 defines the minimization process we apply to the instruction set; Section 6 describes the experimental procedure and present the results; and Section 7 will conclude the paper.

2. Related Work

Early research for ASIPs focused on instruction set customizations to satisfy the constraints on embedded system designs. [13] describes instruction set synthesis for an application in a parameterized, pipelined microarchitecture. Complex instructions which cannot be accommodated within the clock constraint have to be designed as a multi–cycle instruction. [14] proposed a methodology to generate the later as well as single–cycle instructions for DSP applications.

With the demand for shorter design turnaround times, many commercial and research organizations have provided base processor cores, so that less modifications have to be made on the design to achieve the particular performance requirements. This has led to the emergence of reconfigurable and extensible processors. Xtensa [2], Jazz [4], PEAS-III [1], ARCtangent [3], Nios [5] and SP5-flex [6] are examples of processor template based approaches which build ASIPs around base processors.

Xtensa [2] is a configurable and scalable RISC core. It provides both 24–bit and 16–bit instructions to freely mix at a fine granularity. The base processor supports 80 base instructions of the Xtensa ISA with a 5–stage pipeline. New functional units and extensible instructions can be added using the Tensilica Instruction Extension(TIE) Language. Synthesizable code can be obtained together with the software chain tools for various architectures implemented with Xtensa.

The Jazz Processor [4] permits the modelling and simulation of a system consisting of multiple processors, memories and peripherals. Data width, number of registers, depth of hardware task queue, and addition of custom functionality can be added. It has a base ISA which supports addition of extensible instructions to further optimize the core for specific applications. The Jazz DSP Processor has a 2–

stage instruction pipeline, single cycle execution units and supports interrupts with different priority levels. Users are able to select between 16–bit or 32–bit data paths. It also has a broad selection of optional 16–bit or 32–bit DSP execution units which are fully tested and ready to be included in the design. However, Jazz is suitable only for VLIW and DSP architectures. The Jazz DSP System can be configured to handle memories, and on-chip or off-chip bus interfaces clocked at either the same speed or double the speed of the processor [15].

PEAS-III [1, 16] is able to capture a target processors' specification using a GUI. Estimation of area, delay and power consumption of the target processor can be obtained in the architectural design phase. A Micro-operation level simulation model and RT level description for logic synthesis can be generated along with software chain tools. It provides support for several architecture types and a library of configurable components. The core produced follows the Harvard style memory architecture. Several JPEG encoder designs were achieved and evaluated within a short span in [17] by using the PEAS-III approach.

In [18], an existing processor instruction set and architecture can be customized without designing and creating a new processor. This is related to the platform based approach for architectural exploration. There is a need to broaden the architectural space being explored. The issues which need to be addressed for ASIP design exploration are discussed in [19].

One branch of design space exploration for ASIPs is for rapid architectural exploration with added extensible instructions. This has been explored in [7] through the use of the Xtensa processor from Tensilica [2].

Atomic operations within an extensible instruction operation can be duplicated [8] via various cut enumeration and mapping, thus potentially achieving a higher speedup. Extensible instructions are generated via a compilation method. This work was performed by extending the Altera Nios [5] processor which managed to show encouraging speedups of 2.75X on average.

Searching for the best extensible instruction is vital to shorten the design time of modern ASIPs. In [9], a matching algorithm is employed to match the traced program with a set of predefined extensible instructions which have been highly optimized while meeting performance and power constraints. [10] took another step ahead by generating the extensible instruction directly from the application code. An estimation method is used to meet the area and latency constraints to avoid synthesizing which will slow down the exploration process. Both of these works were presented on the Xtensa [2] platform by extending the available instruction sets.

While estimation provides fast exploration of the design space, it is vital that area, power and clock frequency be obtained to justify the decision to select a certain architectural configuration. Recently, [11] used a synthesis-driven design exploration flow for rapid investigation of the different configuration processor architectures. The EXPRESSION ADL [20] was used to generate the design tool chains (i.e. compiler, assembler and simulator). A functional abstraction approach is used to facilitate the generation of HDL code via a *HDL generator*. Thus, chip area, clock frequency and power consumption can be determined from the result of the synthesis of the HDL code. Consistencies among the software tool chain and HDL code can be maintained for a wide range of pipelined architectures because the framework tools, hardware model, compiler and simulator are generated from the same ADL specification. The framework is able to add support for additional pipeline paths, interlocking, stalling, flushing and multi-cycle operations [21]. Modification of the pipeline features and ISA can be made by just making changes to the ADL specification. Architectural features such as VLIW and Superscalar have been implemented on the DLX [22] architecture in this work. However, the occurrence of data dependencies from previous instructions have not been addressed other than stalling of the pipeline [23].

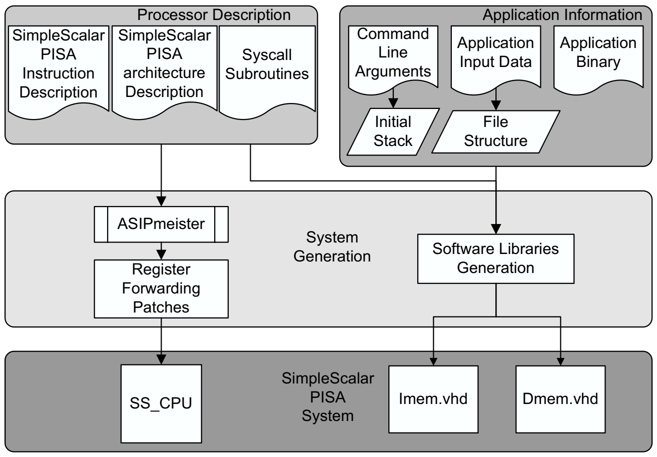


Figure 1: Methodology to construct the SS processor

A large portion of previous research on ASIPs has been focussed on completely custom instruction sets through extending the work on base processors. We have developed a framework that provides total control of the implementation and configuration of the *base processor*, providing opportunities for further design exploration not only by extending instructions, but also by reducing the instruction set to improve performance of the system.

Our research exploits the flexibility of ASIPmeister [1, 16] to include and exclude any subset of the instruction set. Instead of augmenting to the base processor, we can remove redundant instructions from the processor to improve performance in terms of area overhead, power dissipation and latency. Instruction sets can be chosen to closely fit the application being run. Our processor implements the Portable Instruction Set Architecture (PISA) which is closely linked to the SimpleScalar [12] architecture. This ISA is chosen to take advantage of the tool set already available as part of the SimpleScalar framework. Our work can be further improved to include extended instructions as well, providing extra functionality via addition to the existing instruction set.

The contributions of our work are:

1. a methodology for rapid design of a configurable microprocessor core

2. a full SimpleScalar architecture (integer) processor core which is synthesizable into SOC or onto an FPGA for prototyping

3. a novel approach to generate a processor with various subsets of instructions in contrast with other approaches of just extending the base core processor

3. Methodology

The methodology to construct a processor is shown in Figure 1. The system generation step consists of hardware and software library generation. Rapid generation of the processor is achieved using ASIPmeister as it allows targeting to any processor description and instructions can be added and removed at will.

3.1 Hardware Generation

ASIPmeister produces an HDL model for the given instruction set description. We have added register forwarding to the ASIPmeister HDL model to eliminate the need for multiple stalls during execution due to data hazards.

The first step in generation of the hardware is to create a description of the processor to be generated which is suitable for ASIPmeister. The pipeline stages and hardware resources (e.g. register file, divider etc.) must be defined. Instructions to be included in the processor should also be defined at this time. Several pipeline registers need to be added here to allow register forwarding to occur. These include result, destination and source select registers.

Once the processor has been described and hardware resources are defined, the description is entered into ASIPmeister. In this design step, the micro operations performed by each stage of the pipeline are entered to describe the processor. Some register forwarding information needs to be provided by the designer here. Destination registers must be updated to indicate which registers are being updated by each stage of the pipeline. Results for each stage must be collated in registers for ease of forwarding these results to the executing instruction. Signals are also used to indicate which source inputs require forwarding to take place. This is required so operands such as immediate data do not cause register forwarding or data stalls to occur.

After the processor has been input into ASIPmeister and generated, some alterations must be made to complete register forwarding. A parameterized program that generates patch files for ASIPmeister output has been created to perform these alterations. Several key datapath and control issues for register forwarding are fixed by these patches and a forwarding unit is generated that resides in the register decode stage.

The forwarding unit has access to the current instruction and the source and destination registers which were added to the processor description. Using this data, the forwarding unit determines which stage will provide each of the source operands for the execute stage. Some data hazards cannot be overcome by register forwarding such as an instruction using the result of a preceding memory instruction. Stalls are automatically inserted into the pipeline by the forwarding unit when it detects this dependency. The forwarding unit can also detect conflict in the resources being used. As the pipeline has two memory stages, but the data memory has only one access port, a resource hazard may result. The forwarding unit detects these structural hazards and stalls the pipeline so that they will not occur.

Once these alterations have been made, the HDL description of the CPU of the processor is complete. Additional hardware can now be added to the design such as cache and memory mapped I/O to complete the hardware specification of the processor.

3.2 Software Generation

The software generation stage of the design involves the generation of instruction and data memories to be interfaced with the processor and the addition of software subroutines needed to interface to hardware and possibly service interrupts. This may also mean creation of a file system and structure for systems where an operating system is not present. Other additions here may be the creation of a boot loader to initialize the memories and stack used in the system.

This process ultimately generates the memory models used by the design including all the initial memory maps. It must therefore be performed for each application to be executed on the system. The process for creating the memory maps will be the same for each processor and can easily be automated.

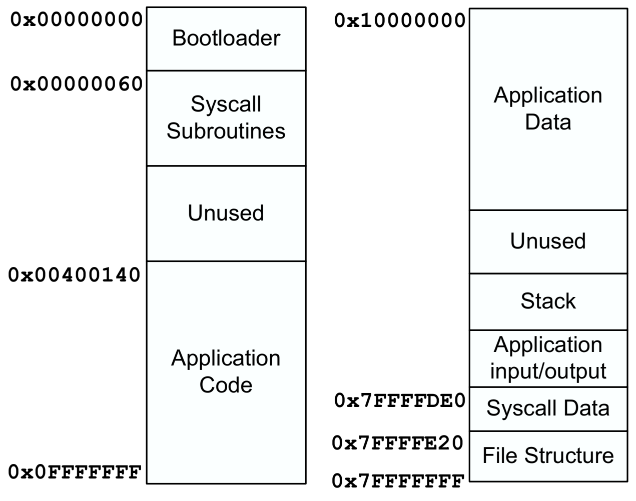
4. SimpleScalar Implementation

To demonstrate the system we have made a hardware implementation of the PISA instruction set executing on the SimpleScalar processor. All the integer instructions and registers in the processor have been defined and implemented using the method described above. Specific implementation issues for creating the SimpleScalar processor are described in this section. We have made this implementation available to download from http://www.cse.unsw.edu.au/~esl/rapid.

4.1 SimpleScalar Processor

The SimpleScalar processor has 64-bit wide instructions and contains a 32x32-bit register file. Additional registers are used for specific instructions, such as HI/LO registers used by multiply and divide type instructions.

The SimpleScalar simulator[24] allows simulation of single cycle



(a) Instruction Memory (b) Data Memory

Figure 2: Memory Map

processor and multiple issue out-of-order execution versions of the processor. However, to demonstrate the methodology, a middle ground has been set for the processor by implementing a six-stage pipelined version of the processor. The stages are as follows:

- IF - Instruction Fetch
- ID - Instruction Decode and Read Register File
- EXE - Execute Instruction in ALU
- MEM1 - Perform First Memory Access (for load/store instructions)
- MEM2 - Perform Second Memory Access (for double word load/store instructions)
- WB - Write Register File

One interesting instruction in SimpleScalar is the syscall instruction. This instruction can be used to perform a wide variety of functions such as file I/O, heap manipulation etc. The SimpleScalar simulator simply uses the syscalls provided by the operating system the simulator is running on to implement these calls. However, this is not an option for a hardware implementation.

The hardware processor therefore treats this instruction like a software interrupt and jumps to a known area of instruction memory where the syscalls are implemented in assembly. One additional register is added to the design to hold the return location of the syscall. An additional instruction has been added to the instruction set to return from syscalls after completion. This is the only new instruction we are required to add to the processor.

4.2 SimpleScalar Software Implementation

SimpleScalar needs several software additions to correctly initialize the processor to execute compiled code. A set of tools has been created to automate creation of the initial memory contents required by each application compiled for the system. The input to these tools are the application binary, command line arguments and information about the files the application will use.

The memory structure for the instruction and data memories created are shown in Figure 2. Known memory locations are included on the diagram at boundary areas of the memory map. The addresses of unknown length boundaries are not shown in the figure as the size will depend on the application executed by the processor and its input data. Each of the areas shown in the figure are further described below.

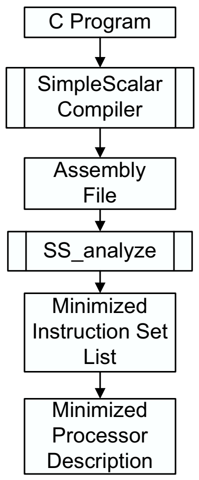

C Program
SimpleScalar Compiler
Assembly File
SS_analyze
Minimized Instruction Set List
Minimized Processor Description

Figure 3: Minimizing instruction set

4.2.1 *Boot Loader*

The boot loader is located at the start of memory and is where execution starts. This code initializes the stack pointer to point to the top of the stack and then jumps to the application code. Once the application completes, it returns to this area to indicate that execution is finished.

4.2.2 *Syscall Subroutines*

As mentioned earlier, SimpleScalar relies on the syscall instruction to perform file I/O and other important system operations. A set of assembly instruction subroutines has been created to provide the syscalls to the processor. The syscalls that have been implemented for our processor are exit, open, close, read, write, lseek, fstat, ioctl, dup2, brk and gettimeofday which provide enough functionality for most programs. These syscalls use the information that has been pre-loaded into data memory to perform their required actions for the application.

4.2.3 *Application Code*

The SimpleScalar binary's entry point is located at 0x400140 in the instruction memory. This is where the machine code of the application resides.

4.2.4 *Application Data*

The start of data memory is used for the application's data and the heap. Global data and constant strings are initialized at this location as specified by the application binary. This area of memory will grow down into the unused space below as the heap grows.

4.2.5 *Stack*

The stack for the system is located in memory so that it can grow up into the unused space as the stack fills. The bottom of the stack is placed as close to the end of the memory as possible after adding the other required data memory segments.

The stack for the SimpleScalar processor is initialized with the command line arguments of the executing application was called with (known as argc and argv in C). This is done to emulate the equivalent stack initialization that the SimpleScalar simulator performs on startup.

To generate initialization data for this section, the command line for the application is needed by the data memory generator. The initial size and contents of the stack and location of the initial stack pointer can be calculated using this information.

4.2.6 *Application Input/Output*

An area of memory is reserved here for file accesses performed by the application. The data for input files used by the system is stored for later reading by the application. Space is also allocated for writing to output files. Included in this area is space for the stdin, stdout and stderr streams for applications which use these for their I/O purposes.

To generate the layout for this section, the data memory generator needs to know the amount of size to allocate to each file and needs access to the input files to copy their data into the data memory. The size of this section is determined by the number and size of the files required by the application. The size is minimized as much as possible to leave more room for the stack.

4.2.7 *Syscall Data*

The syscall data and file structure sections of memory store the information about the system and files that the syscalls need to operate properly. Information for the gettimeofday syscall is retrieved from here and the mapping from file descriptors to pointers into the file table in the file structure is contatined here.

4.2.8 *File Structure*

The file structure section contains a table showing the allocation of files into the Application I/O space. Each file in the structure consists of a filename and three pointers into the I/O area. The pointers indicate the start of the file, the end of the file and the current seek pointer in the file. These pointers are used and updated by the file I/O syscalls to open and close files as well as perform sequential reads and writes.

5. Application Specific Processor Generation

Many applications do not use the full range of instructions available in general purpose processor instruction sets (e.g. the floating point instructions or the 'div' instruction). If a processor is being designed for one of these applications, then the hardware dedicated to decoding and executing the instruction can be removed to potentially increase performance and decrease area and power costs of the processor. By analyzing the application, you can create a minimized processor just for that application by turning off the instructions you no longer need.

We have performed this task for our applications running on the SimpleScalar processor. The methodology of this process is shown in Figure 3. The application is first compiled to an object file by the processor compilation tools. The resulting file is then analyzed to determine which instructions are present. Instructions not present in the compiled application can then be removed from the processor description and the processor is created again through the hardware generation tool. Additionally, the removed instructions are analyzed to see if any hardware resources are no longer required. If all the instructions that access a resource (e.g. the divider) are removed, the resource can obviously be removed from the system without harm.

Another option to allow reduction in size and power of the design is to replace large hardware resources with software subroutines that perform the same function. This technique is best used when you have a large resource that is used very infrequently so the speed loss incurred will not be too high. To perform this task, the instructions that access the resource are turned into syscall-like jumps to known code locations. To demonstrate this, we have written subroutines for the division instruction to replace the divider in applications where the 'div' instructions are only used by infrequent conversion functions such as printf() and scanf().

The process of reducing instructions and replacing large infrequently used hardware components can reduce the processor size quite substantially, especially if entire design resources are removed. This technique is not possible with other ASIP design solutions that use a base processor instruction set that cannot be pruned. This allows rapid processor generation targeted at the application the processor will execute.

6. Experimental Setup

The experimental setup consisted of the system-on-chip architecture as the Device Under Test (DUT) connected to instruction and

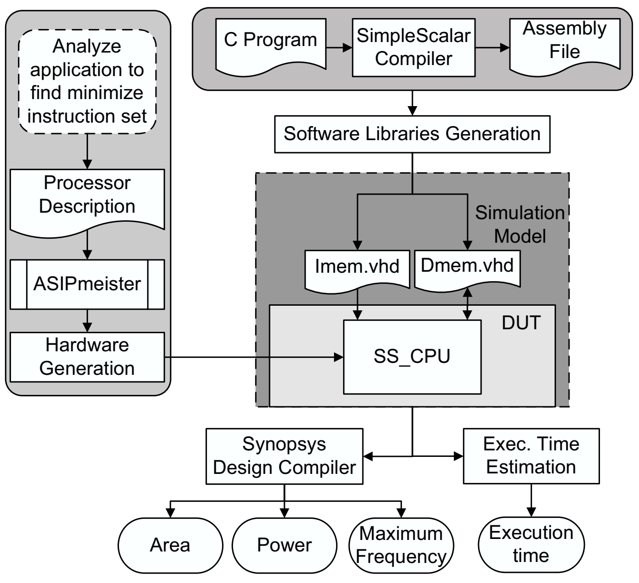


Figure 4: Experimental Setup

Application	Details of the application
adpcmenc	adpcm file encoder
adpcmdec	adpcm file decoder
pegwitkey	pgp key generation
pegwitenc	pgp encryption
pegwitdec	pgp decryption

Table 1: Mediabench Benchmark Applications used in experiment.

data memory models. The memory models are generated by compiling C programs into SimpleScalar binaries then translating them into VHDL models. The setup is shown in Figure 4.

Benchmark applications used in our test are taken from Mediabench [25]. The specific application used for testing are listed in Table 1. Three processors were rapidly generated for use in the experiment, they are listed in Table 2. Configuration A is a processor with all SimpleScalar/PISA instructions; configuration B is a processor with minimized SimpleScalar/PISA instructions based on the adpcm application; and configuration C is a processor with minimized instruction set based on the pegwit application.

Components within the DUT are synthesizable HDL model of our SimpleScalar processor generated through the methodology outlined above.

The SOC architectures described above are used for the purpose of this paper to evaluating the area, power, and performance of the DUT. Other customizations, such as existence of a memory hierarchy, multiple data paths, changes to pipeline depth, instruction issue width, etc. can be rapidly generated for evaluating different SOC architectures.

Synopsys Design Compiler [26] and a 90nm standard cell library were used to evaluate the DUT area, power, and maximum operating frequency. To evaluate the execution time of each application, we estimate the number of cycles needed to run on the processor. A custom tool was created to calculate the total number of instructions executed in the application plus any additional stalls that will occur during the execution. This total execution cycle figure provide an accurate measurement of the processor performance without the need to perform lengthy RTL simulations.

6.1 Analysis of Results

Area and power figures are measurements of the on-chip components only and do not include the external memory. Results of these measurements are shown in Table 3. Column 1 in Table 3 shows the application executed, column 2 shows the processor configuration of the DUT, column 3 provides the area measurement, column 4 gives the percentage area reduction of the minimized instruction set processor compared to a full instruction set processor, column 5 gives the energy measurement, column 6 gives percentage energy reduction when comparing the different processor configuration.

Column 7 shows the total number of estimated execution cycles of the application. Column 8 gives the maximum frequency based on the longest pipeline delay, and column 9 gives the percentage speedup of the minimized instruction set processor compared to the full instruction set processor.

The graph in Figure 5(a) shows that the minimized instruction set processors have an average reduction of 30% of the area compared to the full instruction set processor. Figure 5(b) shows a minimized instruction set architecture can achieve a speedup of 25% compared to the full instruction set processor. In Figure 5(c), the energy reduction of the minimized instruction set processor compared to the full instruction set processor is shown. On average a 24.5% energy reduction is achieved.

Config.	Specification
A	SS_CPU with full instruction set.
B	SS_CPU with minimized instruction set for adpcm application.
C	SS_CPU with minimized instruction set for pegwit application.

Table 2: Different configuration of the SOC architecture.

7. Conclusions

This paper presented a novel methodology for rapid processor generation. Included in this process is a method to tailor the processor to specific applications by reducing the instruction set to the minimum required to execute the application. We have implemented the SimpleScalar/PISA processor as a six-stage pipelined processor and included libraries for syscalls and a file structure in the data memory. Performance figures have been calculated for this processor and minimized versions of the processor for particular applications, which show a marked improvement with the processor reduction technique. As the final processor is in HDL it provides a useful tool for testing other additions to the processor. The SimpleScalar hardware implementation we created and the software tools for generating memories for the device have been made available to download from http://www.cse.unsw.edu.au/˜esl/rapid.

8. References

[1] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: an ASIP design environment," in *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, (Austin, TX, USA), pp. 430–436, 2000.

[2] "Xtensa Processor." Tensilica Inc. (http://www.tensilica.com).

[3] "ARCtangent." ARC International (http://www.arc.com).

[4] "Jazz DSP." Improv Inc. (http://www.improvsys.com).

[5] "Altera Nios Processor." Altera Corp. (http://www.altera.com).

[6] "SP-5flex." 3DSP Corp. (http://www.3dsp.com).

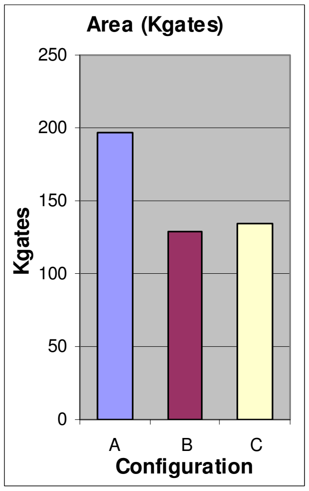
[7] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration & instruction selection for an ASIP: A case study," in *DATE'03*, (Messe Munich, Germany), pp. 802–807, IEEE Computer Society, Los Alamitos, California, 2003.

[8] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, (Monterey, California, USA), pp. 183–189, ACM Press, New York, NY, USA, 2004.

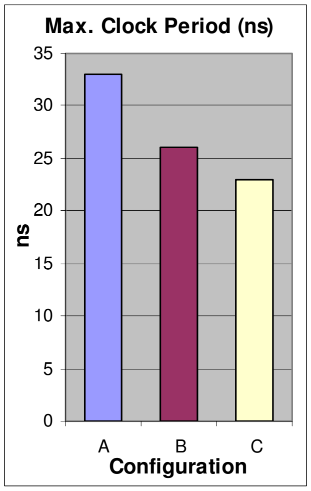
[9] N. Cheung, S. Parameswaran, J. Henkel, and J. Chan, "MINCE: Matching instructions using combinational equivalence for extensible processor," in *DATE'04*, vol. 2, (CNIT La Dfense, Paris, France), pp. 1020–1025, IEEE Computer Society, Los Alamitos, California, 2004.

Application	Configuration	Area (gates)	% area reduction	Energy (mJ)	% Energy reduction	Clock Cycle	Frequency (MHz)	% Frequency improvement
adpcmenc	A	196218		53.23		9560870	30.3	
	B	128843	34.3%	41.94	21.17	9565930	38.4	21.2%
adpcmdec	A	196218		100.74		18092079	30.3	
	B	128843	34.3%	79.37	21.20	18094103	38.4	21.2%
pegwitkey	A	196218		92.75		16654773	30.3	
	C	133924	31.7%	64.64	30.30	16655279	43.4	30.3%
pegwitenc	A	196218		217.48		39052206	30.3	
	C	133924	31.7%	123.75	30.29	39059037	43.4	30.3%
pegwitdec	A	196218		123.75		22222020	30.3	
	C	133924	31.7%	86.25	30.28	22228345	43.4	30.3%

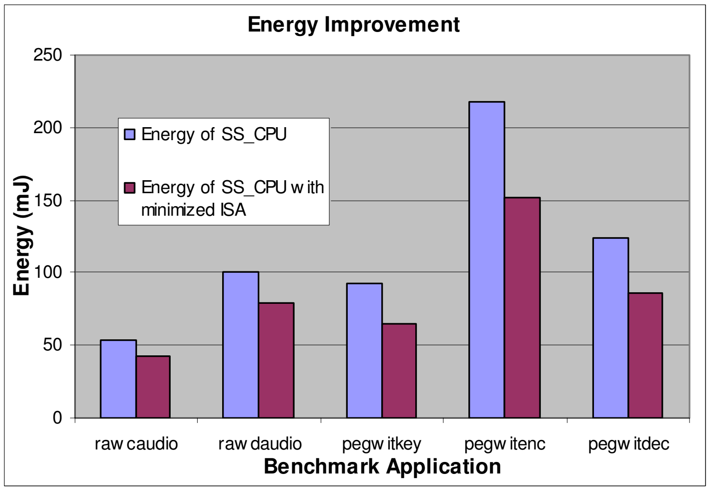
Table 3: Table of results.



(a) Area comparison graph.



(b) Maximum Clock Period.



(c) Energy Improvement.

Figure 5: Experimental results showing area, performance, and energy improvement.

[10] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 216–228, 2004.

[11] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven exploration of pipelined embedded processors," in *VLSID'04*, pp. 921–926, 2004.

[12] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[13] I.-J. Huang and A. M. Despain, "Synthesis of application specific instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 663 – 675, 1995.

[14] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung, "Synthesis of application specific instructions for embedded dsp software," *IEEE Transactions on Computer*, vol. 48, no. 6, pp. 603–614, 1999.

[15] "Application to silicon: Understanding the improv methodology," white paper, Improv Systems Inc., June 2001.

[16] "ASIP Meister." ASIP Meister (http://www.eda-meister.org/asip-meister).

[17] S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai, "Rapid prototyping of JPEG encoder using the ASIP development system: PEAS-III," in *ICASSP*, vol. 2, pp. 485–488, 2003.

[18] K. Küçükçakar, "An ASIP design methodology for embedded systems," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, (Rome, Italy), pp. 17–21, 1999.

[19] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *Proceedings of the International Conference on VLSI Design*, (Bangalore, India), pp. 76–81, 2001.

[20] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *DATE'99*, (Munich, Germany), pp. 485–490, 1999.

[21] P. Mishra, A. Kejariwal, and N. Dutt, "Rapid exploration of pipelined processors through automatic generation of synthesizable rtl models," in *Workshop of Rapid System Prototyping (RSP)*, 2003.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd ed., 2003.

[23] A. Kejariwal, P. Mishra, J. Astrom, and N. Dutt, "HDLGen: Architecture description language driven HDL generation for pipelined processors," technical report, Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA, February 2003.

[24] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," user manual, SimpleScalar LLC, 1997.

[25] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, (Research Triangle Park, NC USA), pp. 330 – 335, 1997.

[26] "Synoposys Design Compiler." Synopsys Design Compiler (http://www.synopsys.com).