S5: The Architecture and Development Flow of a Software Configurable Processor

Jeffrey M. Arnold
Stretch, Inc.
Mountain View, CA
jarnold@stretchinc.com

Abstract

A software configurable processor (SCP) is a hybrid device that couples a conventional processor datapath with programmable logic to allow application programs to dynamically customize the instruction set. SCP architectures can offer significant performance gains by exploiting data parallelism, operator specialization and deep pipelines. The S5000 is a family of high performance software configurable processors for embedded applications. The S5000 consists of a conventional 32-bit RISC processor coupled with a programmable Instruction Set Extension Fabric (ISEF). To develop an application for the S5 the programmer identifies critical sections to be accelerated, writes one or more extension instructions as functions in a variant of the C programming language, and accesses those functions from the application program. Performance gains of more than an order of magnitude over the unaccelerated processor can be achieved.

Index Terms— Reconfigurable Architectures, Software Configurable Processor, Instruction Set Extension, Embedded Computing

1. Introduction

Computer designers have long sought ways to customize the architecture of a computer to suite the needs of particular problem domains. Estrin first described the concept of combining a fixed instruction set architecture with application specific computational elements in 1960 [1]. Unfortunately, the library of substructures offered a limited set of additional functionality, while the need to match program behavior against library elements was beyond the ability of contemporary compilers. Changing the configuration often required the use of a soldering iron.

The desire to tailor computer architecture to specific application domains led to the development of writeable control stores in the 60s and 70s [2]. These machines allowed the creation of large instruction sets by microcoding control structures and processor dataflow around a fixed set of datapath elements.

More recently we have seen the development of "configurable processors"[3]. Configurable processing combines elements from both traditional hardware and software development approaches by incorporating customized and application-specific compute resources into the processor's architecture. These compute resources become additional functional engines or accelerators that are accessible to the designer through custom instructions. Configurable processors offer significant performance gains by exploiting data parallelism through wide paths to memory; operator specialization such as bit width optimization, constant folding and partial evaluation; and temporal parallelism through the use of deep pipelines.

Since the advent of the FPGA, researchers have proposed ways to use programmable logic not only to add application specific functionality to a processor, but to change that functionality dynamically from application to application, and even within a single application [4,5,6].

1.1. Related Work

In the PRISC architecture Razdan and Smith propose adding a programmable function unit (PFU) to the core of a RISC microprocessor[6]. The PFU logically sits adjacent to the microprocessor's ALU and executes combinational functions of arguments provided by the host's register file.

Wittig and Chow extended the PRISC concept in OneChip[7]. Like PRISC, the OneChip PFU receives operands from the processor's register file, but unlike PRISC the PFU has flip flops and can implement instructions with arbitrary latency.

The data bandwidth to the PFU in both PRISC and OneChip was limited to the width of the processor's register file. Ye et. al. proposed Chimaera in which a shadow register is used to cache the contents of up to 9 of the MIPS processor's 32-bit registers[8]. The shadow register is presented as the only operand to the PFU instruction, with the 32-bit result written through the shadow register to the register file. The set of registers to be shadowed is fixed at compile time, making a PFU instruction non-reentrant. A PFU instruction is a collection of MIPS

instructions automatically extracted from the pre-compiled application program.

Cong et. al. added instruction extension capability to the Nios soft processor available for Altera FPGAs[9]. Their approach is similar to Chimaera in that communication between the Nios core and the extension unit uses a shadow register to transfer two 32-bit operands and one 32-bit result.

Becker and Thomas proposed the use of FIFOs coupled with dedicated data movement instructions to facilitate passing operands to extension instructions[10]. Unfortunately, FIFO operands can only be read once; any reuse requires storing the operand values within the reconfigurable array.

Garp and NAPA address the bandwidth issue by using a more loosely coupled coprocessor model in which the reconfigurable units may initiate autonomous access to the processor's memory hierarchy[11,12].

Borgatti et. al. propose a structure that combines FPGA logic and a fixed instruction set processor to support 3 different programming models:

- Instruction set extension;
- Bus mapped coprocessor;
- Flexible I/O processing.

In the instruction set extension model the operands and results come from the fixed processor's register file, similar to PRISC and OneChip. The FPGA fabric and the processor are synchronized by stretching the processor's clock to accommodate the slower FPGA[13].

1.2. Overview

The S5000 is a family of high performance software configurable processors for embedded applications. The S5000 consists of a conventional 32-bit RISC processor coupled with a programmable Instruction Set Extension Fabric (ISEF) able to contain multiple application specific instructions. Extension Instructions (EIs) follow load/store semantics through a very high bandwidth path to memory. Arguments to extension instructions are provided from a 32 entry by 128-bit Wide Register (WR) file. Each EI may read up to three 128-bit operands and write up to two 128-bit results. A rich set of dedicated load and store instructions are provided to move data between the WR and the 128-bit wide cache and memory subsystem. The ISEF supports deep pipelining by allowing extension instructions to be pipelined up to 27 cycles.

In addition to the load/store model, a group of extension instructions may also define arbitrary state variables to be held in registers within the ISEF. State values may be read and modified by any EI in the group, thereby reducing WR traffic.

To ease application development a single, consistent programming model and development environment is provided. Extension instructions are captured using a dialect of C augmented with data types and operators for arbitrary bit width data. The compiler can target the developer's workstation, the instruction set simulator, and the S5000 platform. A rich development and debugging environment completes the tool suite.

In this paper we present the architecture and application development flow of the S5000 embedded processor. Section 2 explores the architecture of the system and the S5 Engine. Section 3 describes the application development flow and the compiler. Section 4 presents some performance data, and Section 5 concludes the discussion.

2. Architecture

2.1. Platform Architecture

The S5000 is a family of platform processors designed to support a wide range of embedded applications. Figure 1 shows a block diagram of one member of the family, the S5610. The heart of the S5000 is the S5 Engine, described in detail in Section 2.2. The S5 Engine consists of a Tensilica Xtensa T1050 32-bit RISC processor running at a clock rate of up to 300MHz coupled with an Instruction Set Extension Fabric (ISEF) running at a clock rate of up to 150MHz.

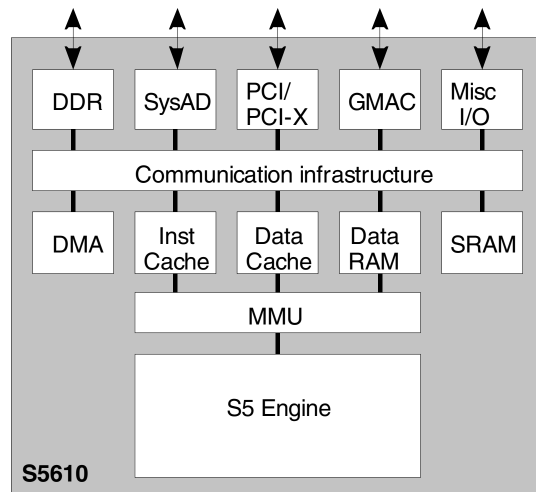

Figure 1. S5610 Block Diagram

The Memory Management Unit (MMU) provides address translation and memory protection mechanisms between the S5 engine and the caches, memory subsystems, and I/O. The Communication Infrastructure is a hierarchical collection of busses designed to deliver bandwidth as required to each of the platform's subsystems. At the interface to the caches and the data RAM it is capable of sustaining a transfer rate of 128 bits per cycle at the processor clock rate of 300 MHz, or 4.8GB/sec.

The Instruction and Data Caches are each 32KB in depth, with a line size of 16 bytes. The Data RAM block is a 32 KB dual port SRAM, with on port

dedicated to single cycle, non-cached access from the processor. The SRAM block is 256KB.

The S5000 platform contains a rich assortment of I/O interfaces, including:

- 1 DDR400 controller capable of addressing up to 3GB of external SDRAM;
- 1 SysAD interface for communication with an external processor;
- 1 PCI-X interface;
- Up to four gigabit Ethernet MACs;
- 2 TDM/HDLC ports;
- 2 UARTs;
- 1 port each of SPI, I2C, JTAG and a programmable serial interface port;
- A DMA engine that supports up to 24 unidirectional channels.

2.2. S5 Engine Architecture

As shown in Figure 2 the S5 engine consists of five major blocks: the instruction fetch and decode unit, the load/store unit, the floating point unit, the integer unit, and the extension unit. The core processor is a Tensilica Xtensa T1050 configurable RISC processor[14]. The fetch and decode unit is responsible for fetching instructions from the ICache and dispatching control to the appropriate functional unit. The load/store unit moves data between the memory subsystem and the register files associated with the integer, floating point and extension units. The integer unit is a conventional 32-bit RISC datapath. The floating point unit implements IEEE single precision floating point arithmetic. The extension unit consists of the Wide Register (WR) file and the programmable Instruction Set Extension Fabric (ISEF).

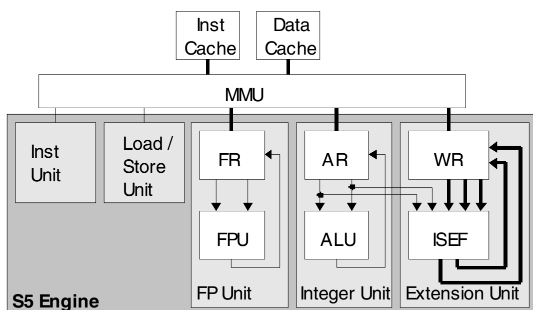

Figure 2. S5 Engine Block Diagram

Most of the S5 engine (e.g. the Xtensa processor, the MMU and memory subsystems) runs at a maximum clock frequency of 300MHz. The ISEF has a maximum clock frequency of 150MHz. This discrepancy is resolved through a programmable clock divider: the ISEF clock is divided from the processor clock by a ratio of 1, 2, 3, 4, 6, or 9. The most typical ratio is 1:3 (300MHz Xtensa, 100MHz ISEF).

The current generation S5000 is implemented in 0.13u CMOS technology. The ISEF is a full custom layout, memory structures are generated, and the rest of the S5 is implemented in standard cells.

2.2.1. Integer and Floating Point Units

The Integer Unit is a 32-bit RISC engine with a five stage instruction pipeline, zero overhead loop construct, and a 16x16 bit multiply and MAC function. Both 16 and 24-bit instruction formats are supported. The AR is a 64-entry 32-bit fully bypassed register file.

The Floating Point unit supports IEEE 754 compatible single precision arithmetic. The FR is a 16 entry single precision register file.

2.2.2. Extension Unit

The WR is a fully bypassed 32 entry by 128 bit register file with four read ports and 3 write ports. Three of the read ports and two write ports are available to the ISEF, while the remaining ports are used by load and store operations.

The load/store unit implements both the standard data movement instructions for the AR and FR register files, as well as a set of dedicated operations to support the WR. The WR load and store instructions transfer 1, 2, 4, 8 or 16 bytes with either immediate offsets or register offsets with optional post-increment. In addition, there are instructions for managing circular buffers, bit reversed loads and stores for FFT like access patterns, loads and stores of 1 to 16 bytes from arbitrarily aligned byte streams, and WR move and rotate.

2.2.3. Instruction Set Extension Fabric

The Instruction Set Extension Fabric consists of a plane of arithmetic/logic elements and a plane of multiplier elements embedded and interlinked in a programmable, hierarchical routing fabric. The arithmetic/logic plane consists of an array of 4-bit ALUs which may be cascaded through a fast carry circuit to form up to 64-bit ALUs. Each 4-bit ALU may also implement up to four 3-input logic functions, and includes four register bits. These registers may be used to implement EI state variables or pipeline resources. A conditional ALU, or CALU, mode is also supported in which the third input selects between two independent ALU functions, allowing expressions of the form:

$$Y = C \ ? \ (A \ op_1 \ B) \ : \ (A \ op_2 \ B)$$

to be implemented in a single ALU. CALU mode is used for a variety of optimizations, including implementing multiplexers and adder/subtractors.

The multiplier plane consists of an array of 4x8 bit multipliers which may be cascaded to form up to 32x32 bit multiplies. The multiplier block also includes programmable registers which may be used to pipeline the multiplication. If a block is not used to perform multiplication the registers may be used as general pipeline resources.

All of the instructions in any application program, hardwired as well as extension, share a

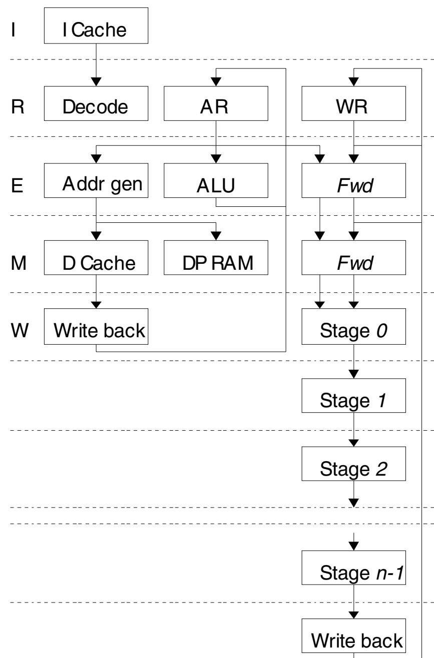

Figure 3. S5 Pipeline

common opcode space. Extension instructions are aggregated into groups by the application developer. A group of EIs is compiled and mapped together to form a single ISEF configuration. Aggregating related instructions in this way allows the compiler to share ISEF resources among multiple, related instructions. The EIs within a configuration are further subdivided into "use/def" classes, where a use/def class identifies the pipeline stages in which WR and state registers are read and written. A configuration table in the instruction fetch/decode unit contains information about the EI group that is resident in the ISEF. This table identifies the opcode and use/def class of each EI, and is consulted whenever an EI opcode is fetched from the instruction stream. If a use/def conflict would occur, (for example, if the fetched instruction needs to read a WR before a previously issued instruction has written it) the instruction issue is stalled until the conflict is resolved.

2.2.4. Dynamic Reconfiguration

The ISEF supports dynamic reconfiguration, either as a result of user or compiler provided directives, or automatically on demand. If a fetched opcode corresponds to an EI that is not resident in the ISEF, an instruction fault is raised. The operating system will then save the contents of any internal state registers, find the EI group containing the missing instruction and initiate an ISEF reconfiguration before resuming the application program. Reconfiguration time is a function of the location of the configuration data and the performance of the memory system, but typically takes less than 100 microseconds.

2.2.5. Processor Pipeline

The pipeline structure of the S5 engine is shown in Figure 3. The Xtensa processor uses a standard five stage pipeline:

- I stage: fetch from instruction cache;
- R stage: decode instruction, read operands from register file;
- E stage: execute instruction or compute memory address;
- M stage: initiate memory transaction
- W stage: write back to register file, or field exception.

An instruction is not committed until the end of the M stage; that is, if an exception or interrupt occurs, instructions in the pipeline that have not completed the M stage will be aborted and restarted later. Those past the M stage will be allowed to complete.

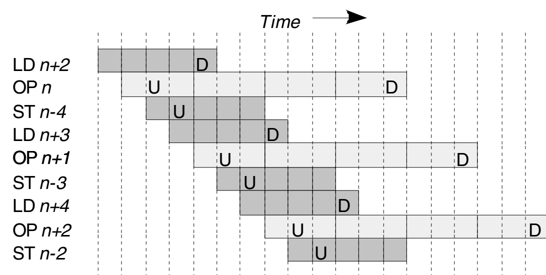

Figure 4. Pipeline Schedule

Extension instructions follow the same basic pipeline: operands are fetched from the WR and/or AR in the R stage. The operands are forwarded through the E and M stages before being presented to the ISEF in the W stage, after the instruction has committed. The current implementation imposes a limit of 27 processor clock cycles on the number of pipeline stages within the ISEF (31 total from R stage to write back). When issuing an ISEF instruction, if the clock ratio is not 1:1 the instruction unit may have to stall to synchronize the ISEF pipeline with the processor clock.

The pipeline integrity is guaranteed through hardware interlocks: if there is a potential register conflict, the pipeline is stalled in the I stage.

Although EIs have variable pipeline depths, the pipeline behavior of any given EI is precisely known to the compiler, which can use this information to effectively perform register allocation and schedule standard and extension instructions together. Figure 4 shows a pipeline schedule of an unrolled inner loop in steady state. The loop body performs one WR load, one EI operation, and one store from the WR. The EI takes 12 processor clocks, running at a clock ratio of 1:3. The load is fetching for two iterations ahead of the EI while the store is four iterations behind The "U" and "D" indicate the register use and def cycles of each instruction. Note that the S5 engine is able to issue a new instruction every cycle,

with no stalls, despite the fact the ISEF is running at one third the clock rate of the Xtensa.

3. Development Flow

3.1. Overview

Application development for the S5000 typically starts with a new or existing application program running on a sequential platform. This code is profiled and analyzed to identify hot spots: those small portions of the code, typically inner loops, that account for most of the execution time. The developer then captures these loops as extension instructions in a dialect of C. The modified application is then compiled and debugged on the developer's workstation using the native mode of the compiler, an emulation package and a standard debugger. Once functional correctness is attained, the code may be compiled for the cycle accurate instruction set simulator, profiled, and tuned for performance. Finally, the application is compiled and linked for the S5000 hardware platform.

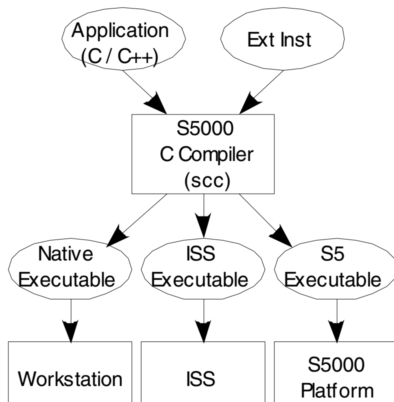

Figure 5. Compiler Targets

The targets of the compiler are shown in Figure 5. The native executable runs on an x86 workstation (Windows or Linux). The ISS executable runs on the cycle accurate instruction set simulator, itself running on the developer's workstation. The S5 executable may be downloaded and run on an S5000 development board, or the customer's own system.

3.2. EI Language and Compiler

The language used to capture extension instructions is a variant of the C language augmented with constructs for specifying arbitrary bit width data types, operators for extracting and concatenating fields of bits, and a syntax for specifying the instruction header, including symbolic opcode names. Extension instructions are written as functions in this language, with either one or several EIs in a single function. An example of the one instruction syntax is shown in Figure 6. This example shows an EI to convert a sequence of pixels from RGB color space to YCbCr space. In RGB space the pixels are 24 bits, 8 bits per color; in YCbCr space the pixels are 16 bits, 8 bits of luminance and 8 bits of chrominance.

```
1  SE_FUNC void RGB2YCbCr (WR A, WR *B) {
2    int i;
3    se_sint<8> Rd[4], Gn[4], Bl[4];
4    se_sint<8> Y[4], Cb[4], Cr[4];
5    for (i = 0; i < 4; i++) {
6      Rd[i] = A(24*i+23, 24*i+16);
7      Gn[i] = A(24*i+15, 24*i+8);
8      Bl[i] = A(24*i+7, 24*i);
9      Y[i] = (77*Rd[i]+150*Gn[i]+29*Bl[i]) >> 8;
10     Cb[i] = (32768-43*Rd[i]-85*Gn[i]+128*Bl[i]) >> 9);
11     Cr[i] = (32768+128*Rd[i]-107*Gn[i]-21*Bl[i]) >> 9);
12   }
13   *B = (Y[3],Cr[3]+Cr[2],Y[2],Cb[3]+Cb[2],
14         Y[1],Cr[1]+Cr[0],Y[0],Cb[1]+Cb[0]);
15 }
```

Figure 6. Example Extension Instruction

The keyword `SE_FUNC` in line 1 identifies this as an EI header. The function name, `RGB2YCbCr`, is the symbolic name of the opcode; the numeric opcode will be determined by the compiler. The EI takes one operand, `A` of type `WR`, and produces one result, `B` of type `WR`. The `WR` type is 128 bits in length, and tells the compiler to values must be passed through the Wide Register file. Local variables are declared next (lines 2-4). The type specifier "`se_sint<8>`" indicates a signed integer object of length 8 bits.

Since the latency of an EI must be known to the compiler, indefinite loops are not supported. However, the compiler will fully unroll any loop with a compile time constant tick count, such as the loop in line 5. Lines 6 through 8 unpack 96 bits of the operand into the RGB components of 4 pixels using parenthesis notation to extract ranges of bits. Since the loop will be unrolled, the unpacking has no time or area overhead. The YCbCr values are then computed using standard C notation in lines 9-11. Lines 13 and 14 assemble the 4 resulting pixels using parenthesis concatenation notation, and assign the result to the B register.

The compiled instruction has 84 operators: 28 multiplies by constant (the multiply by 128 becomes a left shift), 36 adds or subtracts, and 20 shifts by constant. The pipeline depth is three ISEF cycles at 100MHz, for a total latency of 12 processor cycles at a 1:3 clock ratio. The steady state throughput is 4 pixels every three processor cycles, or 1.2GB/sec in and 800MB/sec out.

A state variable is one whose value persists across invocations of one or more EIs. If state variables are required, they are declared as static objects outside the scope of the extension instruction definition. Any EI in the group may read or write the value of any state; the instruction unit guarantees sequential consistency by interlocking references to state variables. Using state variables to hold the overlapping chrominance values, the example of Figure 6 could be extended to process 5 pixels, contain 105 operators, and processes 1.5GB/sec of RGB data.

```
1   /* Declare WRF variables */
2   WR A, B;
3
4   for (...) {
5     /* Load 4 RGB pixels (12 bytes) */
6     WRGET0(&A, 12);
7     /* Convert */
8     RGB2YCbCr(A, &B);
9     /* Store 4 YcbCr pixels (8 bytes) */
10    WRPUT0(B, 8);
11  }
```

Figure 7. Calling an Extension Instruction

Figure 7 shows the inner loop that calls the color conversion EI. Line 2 shows the declaration of two local variables of type `WR`. These are used to pass the data to and from the EI. The `WRGET0` call at line 6 moves 12 bytes of unaligned data to the WR pointed to by &A. Similarly, the `WRPUT0` call at line 10 moves 8 bytes of data from the `WR B`. The base addresses for the GET and PUT operations must be initialized prior to the loop. The load/store unit optimizes the data transfers by moving and buffering aligned blocks of 16 bytes. The actual call to the EI is shown on line 8.

Figure 8 shows the internal flow of the compiler when targeting the S5000 hardware. SCG performs the initial compilation of the extension instruction, and produces three primary outputs:

- Instruction header and latency information, including register use and def, for the Xtensa compiler, xt-xcc;
- Use/def class information for the instruction unit;
- Structural netlist of C operators to be mapped to the ISEF.

SCG performs a number of optimizations on the extension instruction, including:

- Constant propagation;
- Loop unrolling: loops with constant iteration count are completely unrolled;
- Tree height leveling: long sequences of operators are aggregated into balanced trees where possible;
- Bit width optimization;
- Operator specialization: a number of specialization techniques are applied, including converting some multiply by constant into shift and add;
- Resource sharing among operators of different instructions.

The user may specify a target clock rate and ISEF clock ratio. Once the extension instructions are compiled, SCG estimates the timing behavior, determines the pipeline depth and schedule. The instruction latency and use/def information for the WR and state registers are passed to the Xtensa compiler and to the configuration generator.

If the application will run in a multi-processing environment, or if reconfiguration on demand is required, SCG will synthesize additional extension instructions to save and restore EI state variables. These instructions may then be called by the OS to effect a context switch.

Since the logic resources of the ISEF are finite, it is possible to write a group of extension instructions that exceeds the capacity. To help the programmer, SCG provides an estimate of the resources required to implement the EI group, and will exit if the capacity will likely be exceeded.

The Xtensa compiler, xt-xcc, compiles the application C/C++ with references to the extension instructions. Since the EIs follow the same semantics with respect to the WR and the pipeline, the compiler is able to use the instruction header and timing data provided by SCG to perform register allocation and optimal scheduling of the entire instruction stream.

The back end of the compiler resembles a conventional FPGA tool flow. The map stage performs module generation from the set of operators provided by SCG. This step may involve additional operator specialization and merging.

The place stage assigns locations to each of the modules generated by map using a "timing aware" algorithm.

The router performs detailed, timing driven routing on the placed netlist.

Retime is a pipeline retimer that moves registers to balance every pipeline stage delay.

The bit stream stage performs several clean up chores, including the creation of the ISEF and instruction unit configuration files. It also performs a

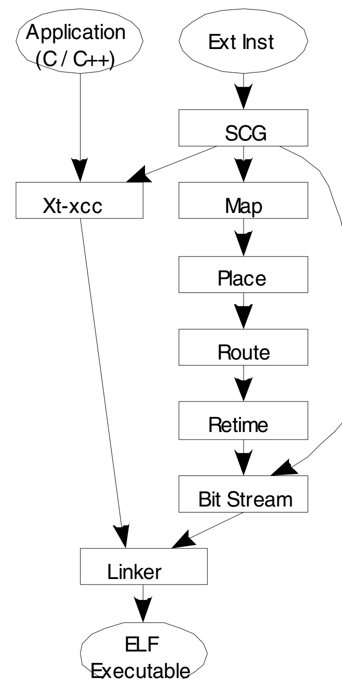

Figure 8. Compiler Flow

number of integrity checks on the bit stream to ensure proper and safe mapping.

The linker packages the components of the application together into a single executable file. This file includes a directory of the ISEF configurations that is used by the OS/runtime system to locate instruction groups for dynamic reconfiguration.

3.3. Other Tools

In addition to the compiler described above, the S5000 software package includes several other tools and libraries. The instruction set simulator (ISS) provides a cycle accurate model of the S5 engine and memory subsystems for detailed performance analysis and tuning. When targeting the ISS the compiler produces a cycle accurate model of the extension instructions that is linked into the ISS at run time.

A common debugger provides access to both the ISS and the physical hardware. A profiler and analysis tool are available for performance analysis and tuning. An integrated development environment (IDE) combines all of the tools together with an editor into a single graphical environment.

A simple OS/runtime provides support for the peripheral devices, memory management, application loading and configuration management.

4. Performance

To measure the effectiveness of the software configurable processor approach we report the performance of the S5000 on the EEMBC Telemark benchmark suite[15]. The Telemark suite consists of five programs representing different telecommunications tasks, each with three different data sets:

- Autcor: Calculation of a finite length 16-bit fixed point autocorrelation function. The computation is dominated by multiplies.
- Conven: A ½ rate convolutional encoder for forward error correction. The computation is dominated by bitwise AND and XOR, byte shift and 8- and 16-bit unsigned arithmetic.
- Fbital: Bit allocation for a DSL modem using discrete multi-tone modulation. The computation is dominated by arithmetic and comparisons.
- FFT: A 256 point 16-bit complex finite Fourier transform.
- Viterbi: Viterbi decoder algorithm for forward error correction. The data packet consists of 344 6-bit values; the computation is dominated by bitwise logic and add-compare-select operations.

EEMBC requires two sets of measurements: "out-of-box", or unoptimized compilation of the reference C code; and "full fury", or optimized to the greatest extent possible for the architecture. The out-of-box code is compiled with the Tensilica compiler, xt-xcc, using –O2 optimization. Since the EEMBC benchmarks are intended to compare processor architectures and not memory systems, the EEMBC test harness runs the application code on each test data set many times to minimize the effect of cache misses on the performance measurement. The results produced by the program under test are compared against a reference file to ensure correctness. The performance results are reported in terms of the number of algorithm iterations per million clock cycles.

Table 1. EEMBC Telecom Benchmark Results

Benchmark	OOB	Opt	Speedup
Autcor data 1	1164	30267	26
Autcor data 2	9.4	1229	131
Autcor data 3	9.9	1234	125
Conven data 1	10.5	36984	3522
Conven data 2	12.3	36984	3007
Conven data 3	14.7	36984	2516
Fbital data 2	2.8	811	290
Fbital data 3	33.3	5049	152
Fbital data 6	4.1	789	192
Fixed pt FFT	17.6	899.4	51
Viterbi	4.8	574.4	120

The EEMBC certified results reported in Table 1 are simulated on the cycle accurate instruction set simulator; laboratory measurements on the actual silicon confirm the results. The processor clock rate is 300MHz, with a 1:3 ISEF clock ratio. The performance of the FFT and Viterbi benchmarks is independent of the data set, so only one line appears for each.

As can be seen, the speedups when optimized using extension instructions range from 26 to more than 3500.

The autcor implementation contains two instructions which share 64 16x8 bit multiples, 8 8-bit incrementers, 64 adders ranging is size from 24 to 27 bits, and 8 dynamic shifts specialized for shift distances of 1 to 8 for a total of 144 operators. In addition, there are 8 24-bit state variables used as accumulators. In the table, the speedup for data set 1 is clearly an outlier. This is due to the small size of the test data set (16 points, compared to 1024 for data set 2).

The conven benchmark implements two 5th degree polynomials, and includes 10 ANDs and 8 XORs per bit, times 128 bits, for 128-way data parallelism with 2304 bit operations. Fourteen bits of state are used to hold polynomial coefficients and the previous code. It should be noted that the EEMBC reference code is written for clarity, not performance. The inner loop does a compare and branch for each coefficient on each bit of data. Careful recoding should improve the performance of the reference code by more than an order of magnitude.

The fbital implementation contains 3 instructions and 1 16 bit state variable. Two of the instructions share 80 adders ranging in size from 4 to 16 bits, 48 8-bit comparisons, and 32 9x7 multiplies, for 128 operations, plus 48 static shifts.

The FFT implementation performs four butterfly operations in a single instruction, reading three 128-bit operands and producing two 128-bit results. The ISEF contains 6 instructions with 16 16x16 multiplies, 8 32-bit add/subtracts, and 16 16-bit add/subtracts, for 40 operations, plus logic for steering data.

The Viterbi decoder likewise is straightforward. It has 2 instructions containing 70 adds and subtracts, plus 32 14-bit comparisons. There are also 64 state variables.

5. Conclusion

We have described a software configurable processor architecture capable of achieving one to two orders of magnitude performance improvement over unoptimized code on the base RISC processor. There are three sources of performance gain:

- Data parallelism, with extension instructions reading up to three 128-bit operands and producing up to two 128-bit results;
- Temporal parallelism, with deep pipelines extending up to 27 processor clock cycles;;
- Instruction specialization, such as bit width optimization, partial evaluation and resource sharing.

A single, consistent programming model and tools set is provided to ease the development process and to help achieve significant performance gain.

Future software configurable architectures may add coarser grain parallelism such as instruction level parallelism (ILP) and task level parallelism (multi-processing).

6. Acknowledgements

The S5 architecture is the product of the efforts of many people. I would especially like to thank the rest of the Stretch Architecture team: Gary Banta, Ricardo Gonzalez, Scott Johnson, Charle' Rupp, Albert Wang, and Mark Williams.

7. References

[1] G. Estrin, "Organization of Computer Systems: The Fixed-Plus Variable Structure Computer," *Proc. Western Joint Computer Conf.,* Am. Inst. Electrical Engineers, New York, 1960, pp. 33-40.

[2] C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering*, Digital Press, 1978.

[3] R.E. Gonzalez, "Xtensa: A Configurable and Extensible Processor", IEEE Micro, 20:2, March/April 2000, pp. 60-70.

[4] P.M. Athanas, and H.F. Silverman, "Processor Reconfiguration through Instruction Set Metamorphosis: Architecture and Compiler", *Computer*, Vol. 26, No. 3, Mar. 1993, pp. 11-18.

[5] D.A. Buell, J.M. Arnold and W.J. Kleinfelder, Splash 2 FPGAs in a Custom Computing Machine, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 171-176.

[6] R. Razdan, and M.J. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", International Symposium on Microarchitecture, ACM Press, San Jose, CA, 1994, pp. 172-180.

[7] R.D. Wittig, and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", IEEE Symposium on FPGAs for Custom Computing Machines, CS Press, Napa, CA, 1996, pp. 126-135.

[8] Z.A. Ye, A. Moshovos, S. Hauck, and P. Bannerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Function Unit", International Symposium on Computer Architecture, 2000, pp. 225-235.

[9] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", International Symposium on Field-Programmable Gate Arrays, ACM Press, Monterey, CA, 2004, pp. 183-189.

[10] J. Becker, and A. Thomas, "Scalable Processor Instruction Set Extension," *IEEE Design & Test of Computers,* Vol. 22, No. 2, Mar/Apr 2005, pp. 136-148.

[11] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture", IEEE Symposium on FPGAs for Custom Computing Machines, CS Press, Napa, CA, 1998, pp. 28-37.

[12] J. Hauser, and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", IEEE Symposium on FPGAs for Custom Computing Machines, CS Press, Napa, CA, 1997, pp. 12-21.

[13] M. Borgatti, F. Lertora, B. Foret, and L. Cali, "A Reconfigurable System Featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customizable I/O", IEEE Journal of Solid State Circuits, 38:3, March 2003, pp. 521-529.

[14] Tensilica, Inc., http://www.tensilica.com.

[15] Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.