Universität Dortmund
Lehrstuhl Informatik XII
44221 Dortmund

Low Power Code Generation for a RISC Processor by Register Pipelining

Stefan Steinke, Rüdiger Schwarz,
Lars Wehmeyer, Peter Marwedel

Technical Report # 754

26.3.2001

Low Power Code Generation for a RISC Processor by Register Pipelining*

Stefan Steinke, Ruediger Schwarz, Lars Wehmeyer, Peter Marwedel

{steinke | schwarz | wehmeyer | marwedel}@ls12.cs.uni-dortmund.de

University of Dortmund, Dept. of Computer Science

Otto-Hahn-Strasse 16

44221 Dortmund, Germany

Abstract

This paper presents the implementation of the compiler technique register pipelining with respect to energy optimization and its comparison against performance optimization. Generally, programs optimized for performance are also energy optimized. An exception to this rule is shown where the use of register pipelining improves the energy consumption by 17% while bringing down performance by 8.8%.

Therefore, a detailed consideration of energy consumption within the processor and the memories is necessary.

1 Introduction

An increasing number of applications like digital audio players, PDAs or digital cameras are controlled by embedded systems. The requirements for these systems often include hard real-time constraints. Especially for portable devices, e.g. mobile phones, energy consumption is crucial as battery life is an important technical feature. For other applications, heat dissipation has to be minimized due to intolerable cooling costs. Finally, reducing the chip temperature can extend the lifetime of electronic systems. All these characteristics can be improved by reducing the energy consumption.

To reduce energy consumption, many technical improvements concerning the architecture of electronic systems are being investigated. Most of them are from the area of hardware design where e.g. clock gating reduces the number of active gates or bus encoding reduces the number of toggled bus lines. But beside changes in hardware design, software design issues are another promising approach. The advantage of software in general is that it can be developed and changed at a late stage in the development. Furthermore, software changes are generally less expensive and can be delivered as an update.

The reduction of energy consumption by generating software consuming less energy has been a research issue for a couple of years [14]. In addition to the optimization for code size and performance, reducing energy consumption is a new optimization goal for compilers. This means that the processor and other system components should dissipate

*This work has been supported by Agilent Technologies, USA

minimal energy during the execution of the generated program. Since we are dealing with compilers, only those aspects that can be influenced using software are considered.
During program execution, memory accesses consume a large amount of energy. Especially in systems using RISC processors and off-chip memory, it is worthwhile to investigate methods of reducing the number of memory accesses. This is also being done to improve execution speed as memory accesses are usually slow compared to processor speed. But even without saving clock cycles, it is advantageous to reduce memory accesses since overall energy consumption is reduced.
Register pipelining, a known optimization technique in compilers, eliminates memory load accesses in loops by temporarily storing the data in unused registers. This is especially profitable whenever arrays are concerned. Following a detailed analysis of the array dependencies, additional statements are usually inserted. Due to this, a program may have longer execution time, but still consume less energy than before. An example program will be given in this work.

2 Related Work

An overview of hardware and software based methods for reducing the energy consumption of electronic systems can be found in [8]. Some of these can be integrated into the compiler and thus reduce energy consumption without any hardware changes. The compiler optimization technique register pipelining [1,2,9] belongs to this class of software methods and deals with memory accesses. It reduces the number of memory accesses for arrays and ensures better usage of the memory hierarchy.
In order to be able to generate energy optimized code, the compiler needs detailed information about the energy consumption of instructions. However, hardware vendors of general purpose processors do not publish detailed data or an instruction-level power model of the processor in most cases. Therefore it is necessary to perform measurements concerning power consumption of all instructions of the individual processor. This was first done by Tiwari, who published detailed information on the energy consumption of a 486DX2 processor, a RISC and an embedded processor [13,14,15,16]. Later measurements were also taken for an ARM processor by Sinevriotis [11].
Tiwari et al. [14] proposed a power model based on their measurements of the 486DX2. This has been extended in our approach to include energy used by the program and data memory. Especially for the ARM7TDMI processor which was chosen for experiments in our work, the energy used for an instruction fetch from memory as well as data loads and stores have to be taken into consideration. Several experiments have shown that the amount of energy consumed by the memory can be twice the amount of energy consumed by the processor itself.
The work of Panda et al. [10] describes the behavior of the memory and optimizations for memory accesses in detail. The use of scratch-pad memories is also part of their research.

3 Algorithm of Register Pipelining

3.1 Principle

During compilation, a compiler replaces the variables of the programming language with virtual registers and later, during the register allocation phase, with physical registers. Due to the limited number of processor registers, some virtual registers have to be temporarily stored in memory. This is called spilling.
Arrays are usually stored in memory and the elements are accessed with load/store instructions. Register pipelining is a technique which reduces these memory accesses. It replaces data load operations for array elements retrieved in a previous loop pass by storing the data temporarily in unused processor registers whenever this is possible. This is done in loops where such improvements can be very profitable.

The main principle is shown in the C program given in Figure 1, which is transformed into the program in Figure 2 using register pipelining. The number of memory accesses for array *a* is reduced from 240 to 120, since the value of *a[i-1]* is temporarily stored in register *R*.

```
for (i = 1; i < 120; i++) {
        a[i] = a[i-1] + 3;
}
```

Figure 1: *Original source code*

3.2 Algorithm

In this work, register pipelining is performed after register allocation. This stage was chosen because the exact number of available registers can easily be determined by analyzing the instructions together with their processor registers. This is very important, since a possible spill, which could be caused by a wrong decision during register pipelining, would cost much more than the whole optimization potentially saves.

The transformation of the original assembly program is performed in the following steps and repeated from step 5 as long as beneficial optimizations can be found:

3.2.1 Step 1. Loop detection

Loops like the *for* statement in Figure 1 have to be detected. This is done by a depth first search algorithm on the internal control flow graph. A control flow graph is a representation of the instructions as nodes and their control dependencies as edges.

The result of the depth first search algorithm is a classification of the edges. Each back edge represents an existing loop. Starting with this analysis, the whole loop and all involved edges can be determined.

3.2.2 Step 2. Search for induction variables and their limiting values

Induction variables like *i* in Figure 1 are variables whose values depend on the number of executed loop iterations. In this step, the induction variables are determined along with the step width as well as their lower and upper bounds which are 1 and 120 in the above example. A higher percentage of loops can be processed by applying algebraic transformations. In the subsequent step, possible different induction variables in the index functions are transformed into one main variable if possible.

3.2.3 Step 3. Search for load and store operations

The load and store operations have to be determined in order to replace the loads with register accesses later on. For each load or store memory access concerning an array, the respective index function is computed which is based on the induction variables at loop entry.

```
R = a[0];
for ( i = 1; i < 120; i++ ) {
        R = R + 3;
        a[i] = R;
}
```

Figure 2: *Source code after register pipelining*

3.2.4 Step 4. Array data flow analysis

The array data flow analysis used here is based on a method developed by Duesterwald [3]. The dependencies between different memory accesses in the loop are evaluated.

3.2.5 Step 5. Calculation of free registers

The number of available registers for register pipelining has to be determined in order to avoid possible spill code.
In this step, inhomogeneous register files have to be considered. For example, the considered processor ARM7TDMI provides 8 lower registers in the 16-bit instruction set, which can be used with all instructions, and 5 higher registers, which can only be addressed by a limited number of instructions.
This is taken into consideration in order to allow different transformations depending on the type of available registers.

3.2.6 Step 6. Optimization exploration and selection

In general, a number of transformations have to be explored. The choice of the most efficient one depends on:

- control flow,
- life time of the registers,
- number and the type of additionally usable registers,
- positions of load and stores of different array elements.

After exploring different kinds of transformations and determining the number of free lower and higher registers, the benefit of all transformations is computed and the most efficient one is applied.
Included in the process are some standard optimizations like copy propagation, which are executed after the actual transformation to further improve code quality.
Several runs of register pipelining are performed to check if additional optimizations are possible.

4 Basics of Energy Consumption

4.1 Physical Basis

For measuring the energy consumption and optimizing the compiler, it is necessary to know the physical basis for the dissipation of energy. In an electronic system, the processor and the memory, which are the focus of this research work, are usually manufactured using CMOS technology. In CMOS, three sources of dissipation have to be distinguished [12]: The switching power, the short circuit power and the leakage power. In active CMOS circuits, switching power accounts for 70 to 90% of the total power dissipation and depends on the toggle rate of the gates.
In order to produce low power code, the compiler's target is mainly to reduce the switching activity and thus switching power.
For the decision process inside the compiler, a power model reflecting the physical processes was developed.

4.2 Power Model

It seems obvious for our approach to choose processor instructions as the base for reducing switching activity. The compiler can calculate the costs either before code selection or later in the code generation process. Also, a sequence of instructions can be considered, e.g. during register pipelining.
It has to be taken into consideration that no detailed circuit description or VHDL model can usually be obtained for generally available processors. Thus, the electrical measurement of the current consumed by the processor should be used to obtain a satisfactory decision base for the compiler.

As mentioned before, our work is based on the power model developed by Tiwari [13,14,15], which distinguishes between base costs and inter-instruction effects. Base costs consist of the measured current during execution of a single instruction in a loop and an approximate amount added for pipeline stalls and cache misses. The change of circuit state for two different consecutive instructions and resource constraints are summed up in the inter-instruction effects.

Tiwari's research work shows that the base costs constitute the major part of the total amount of energy consumed in executing a sequence of instructions. This is fortunate, since the decisions in the compiler are mostly concerned with the choice of an isolated instruction without knowledge about the complete instruction sequence.

The power model we used includes the power dissipation of the processor P_{proc} and that of the memory P_{mem}. The latter aspect which was not considered by Tiwari et al. is necessary because the measurements for the ultra low power processor ARM7TDMI have shown considerably higher energy dissipation in the memory than in the processor itself. Since the number of memory accesses can be influenced by the compiler, the energy consumption of memory accesses was integrated into the power model:

$$P_{total} = \sum_{executed\ instr} (P_{proc} + P_{mem})$$

It has to be mentioned here that although termed "power model", in the electrical terminology it would have to be "energy model". The computation of energy includes the execution time as well as the power used by a particular instruction.

4.3 Experimental Values

As a base for the experiments we used the encc compiler [4], which integrates the LANCE ANSI C frontend [6] and a retargetable backend for the ARM processor. Different cost functions for time, energy, power and code size are available. The power model described above is integrated into the compiler and into a trace analyzer, which computes the total amount of energy dissipated during execution of the program under observation.

Instruction	Cycles	Current (mA)
MOV Rd,Hs	1 + n	43.8
ADD Rd,#imm	1 + n	44.7
BGE	3 + n	43.0
CMP Rd,Hs	1 + n	41.3
LDR Rd,[SP.#imm]	3 + n + m	48.8
STR Rd,[SP,#imm]	2 + n + m	57.7

Table 1: *Measured current of processor (base cost)*

n = number of additional cycles for instruction fetch from program memory
m = number of additional cycles for data load and store from data memory

Beside the cost function there are two databases for energy. The first database contains the values of the processor energy of each instruction and the second database contains the energy consumption for accesses to different memory types in the system. Detailed measurements show the margin of error of the power model to be 1.7%.

For our measurements, we chose the ARM7TDMI processor which is common in present embedded systems especially for low power applications. This processor is a member of the family designed by Advanced RISC Machines Ltd. and

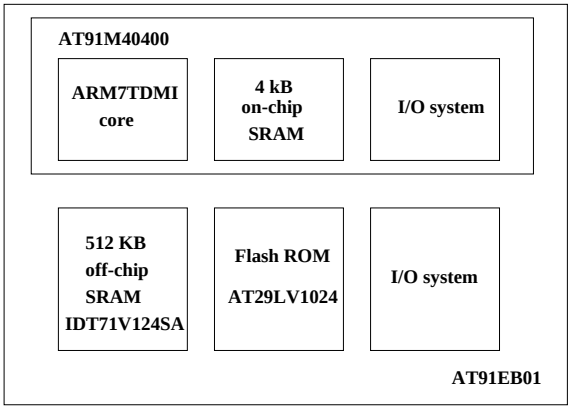

Figure 3: Evaluation hardware

is manufactured and licensed by a number of major processor vendors. The ARM7TDMI is based on the ARM7 core which is often combined with on-chip memory and I/O logic. It also includes a so called THUMB Instruction Set which is recommended by the vendor for power conscious applications.

The experiments were performed using the ATMEL AT91M40400 processor, which features an additional 4 KB static on-chip RAM, but no cache, and the evaluation board ATMEL AT91EB01. It includes the processor, 512 KB static off-chip RAM (IDT71V124SA), Flash ROM (AT29LV1024) and I/O components, which are all shown in Figure 3. For the evaluation of register pipelining, a database for the power model is required to compute the energy values for the instruction sequences. Therefore, a series of measurements was performed to measure the energy consumption of the processor instructions of the 16-bit THUMB Instruction Set using a digital amperemeter Cosinus Escort 95 with a 4-3/4-digit display and an error rate of 0.08 percent plus 5 digits.

In Table 1, some of the essential measurements are shown for the 16-bit THUMB Instruction Set. Sinevriotis [11] took measurements only for the 32-bit ARM instruction set. The load and store instructions need 20% more current than a move instruction between registers and additional current for the memory access. This difference is exploited by the optimization register pipelining. To compare the energy, this has to be multiplied with the number of cycles, which depends on the memory type used.

In a second phase, the energy consumption by the off-chip 512 KB SRAM memory was measured with different word widths (cf. Table 2). The amount of current needed for a data read (48.8mA + 98.4mA) or write compared with a

Mode	Bit width	Add. cycles	Current (mA)
Instruction Fetch	16	1	77.2
Data Read	32	3	98.4
Data Write	32	3	101.0

Table 2: *Measured current of 512 KB off-chip RAM*

move instruction (43.8mA) is twice the amount of the processor current itself. For the on-chip memory, the current is part of the measurements of the processor current. On average, an additional current of 3 mA was measured for accesses to this type of memory. Based on these values, the amount of energy necessary for an access to off-chip memory was computed and integrated into the database within the compiler.

Register pipelining replaces load instructions by a pipeline of move instructions. Based on the measurements, Table 3

shows the number of moves which are equivalent in energy consumption to a 32-bit load instruction depending on the memory type:

$$n = \frac{P_{Load} * Cycle_{Load}}{P_{Move} * Cycle_{Move}}$$

The compiler should only replace load instructions with a register pipeline of move instructions up to this amount.

Program memory	Data memory	Equivalent number of move instructions
off-chip	off-chip	3.5
off-chip	on-chip	1.6
on-chip	off-chip	16.2
on-chip	on-chip	3.4

Table 3: *Number of move instructions equivalent to one load instruction*

5 Results

5.1 Benchmarks

The register pipelining optimization is helpful in loops with accesses to array elements that were already used in previous iterations. Experiments were conducted with different benchmarks meeting this criterion. To show the efficiency of the technique, the following benchmarks were used:

The benchmark *biquad_N_sections* is typical for the DSP domain and part of the DSPStone benchmark.

The benchmark *lattice* is a filter application.

	free registers		# data memory accesses (bytes)	
Register Pipelining	No	Yes	No	Yes
biquad_N_sections	5	2	1196	1100
lattice	3	0	43020	31620
Hydro fragment	7	6	1632	1272
Tri-diagonal elimination	7	6	1472	1152
Equation of state fragment	4	0	4048	3000
First sum	8	7	1184	864
First difference	7	6	1224	872

Table 4: *free registers and # of data memory accesses*

In the work of Callahan et al. [1] the *Livermore* benchmark suite [7] was used. These benchmarks are well suited for benchmarking loops. For our experiments the following kernels were selected: kernel 1 (*hydro fragment*), kernel 5

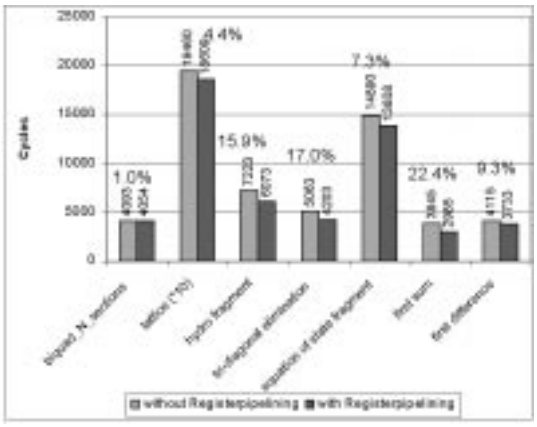

Figure 4: # of cycles (program in off-chip RAM, data in off-chip RAM)

(*tri-diagonal elimination*), kernel 7 (*equation of state fragment*), kernel 11 (*first sum*) and kernel 12 (*first difference*). The ARM7TDMI processor is typically used for processing data of type integer as it features no hardware floating point unit. Thus, the data type double was replaced by integer.

Table 4 shows how the number of free registers changes after application of register pipelining. A total number of 8 "low" registers are available in the THUMB instruction set and up to 5 "high" registers can be used with a limited number of instructions. Register pipelining leads to an increased use of the registers and thus reduces the number of necessary data memory accesses (cf. Table 4).

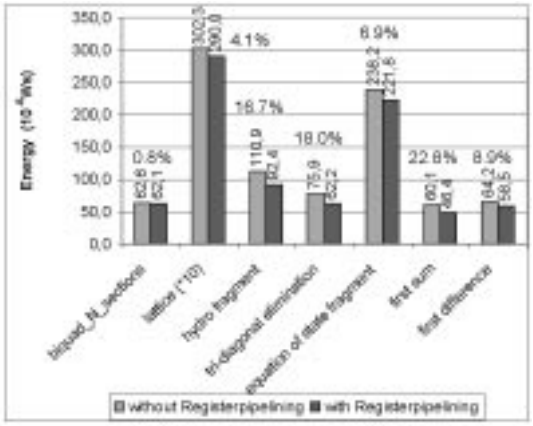

Figure 5: Energy (program in off-chip RAM, data in off-chip RAM)

The results regarding number of cycles are shown in Figure 4. Depending on the number of load statements which are replaced during register pipelining, the improvement varies between 1% and 22% with an average of 11%. Additionally, some of the index calculations necessary for array accesses are removed.

The effect on energy consumption is similar. It varies between 0% and 22% with an average of 11% (cf. Figure 5).

To show the dependency on the architecture of the system (esp. concerning the memory hierarchy), the program was then loaded into the fast and energy efficient internal 4 KB on-chip RAM. Even though this may not be possible with larger programs, it is common to use a fast, low power ROM for storing the program code in embedded systems. In this

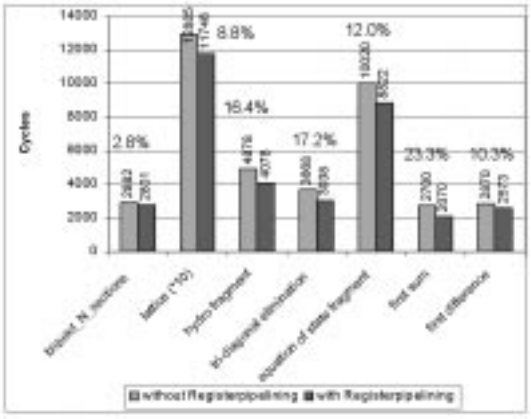

Figure 6: # of cycles (program in off-chip RAM, data in on-chip RAM)

configuration the data memory accesses are more expensive than the accesses required for instruction fetching, both regarding the number of cycles and energy, further enhancing the positive effect of the register pipelining technique.

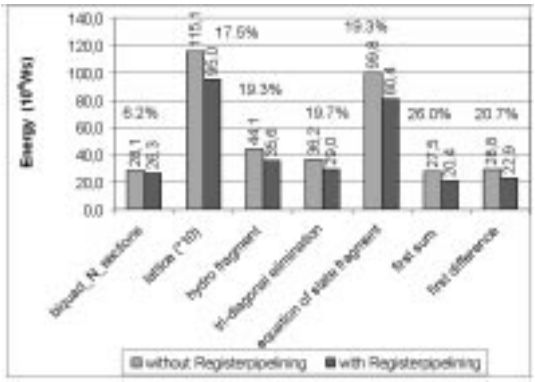

Figure 7: Energy (program in off-chip RAM, data in on-chip RAM)

Assuming the last-mentioned configuration, the improvement varies between 3% and 23% with an average of 13% (cf. Figure 6). These better results compared to Figures 4 and 5 are due to the efficient access to the on-chip RAM. The improvement concerning energy consumption is also increased and varies between 6% and 26% with an average of 18% (cf. Figure 7).

5.2 Performance versus Energy Optimization

In most cases the application of register pipelining improves both the performance and the energy consumption. However, there are exceptions to the rule. Figure 8 shows an example where the application of register pipelining is not profitable for performance but for energy (cf. Table 5). Whereas a performance optimization prefers the energy expensive load instructions, the energy optimization uses more move instructions causing more cycles but in total less energy dissipation. The optimization technique register pipelining offers this alternative and depending on the chosen optimization the best code is generated.

```
int a[100+7];

int main(void) {
  int i,b,*c;
  b = 0;
  for (i = 0; i < 100+7; i++) {
    a[i] = i;
  }
  c = a;
  for (i = 0; i < 100; i++) {
    b += *c;
    b += *(c+7);
    c += 1;
  }
  return (b);
}
```

Figure 8: time vs. energy example

	without register pipelining	with register pipelining	difference
# of cycles	1958	2130	+8.8%
# of exec. instructions	795	1330	+67%
Data memory access	796 Bytes	492 Bytes	-38%
Energy consumpt.	19.33 10^{-6}J	16.02 10^{-6}J	-17%

Table 5: *performance, size, energy of time vs. energy example*

6 Conclusion

The increasing role of embedded systems and the included software demands that optimizations of energy consumption have to be studied from the perspective of software, too.

The presented work implements register pipelining and compares the energy against the performance optimization. It is shown that for some program parts optimization for performance and optimization for energy yield different results. Therefore a detailed consideration of energy consumption is necessary. A power model was presented and the benchmarks show the dependency of results on the processor and the memory hierarchy.

7 References

[1] Callahan, D. and Carr, S. and Kennedy, K., "Improving Register Allocation for Subscripted Variables", *Proc. of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 20-22,1990

[2] Carr, S., "Memory-Hierarchy Management", Rice University, Thesis, CRPC-TR92222-S, 1992

[3] Duesterwald, E. and Gupta R. and Soffa M. L., "A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations", *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque*, NM, 1993

[4] Encc compiler, University of Dortmund, Dept. of Computer Science, http://LS12-www.cs.uni-dortmund.de/∼steinke, 1999

[5] Kandemir, M. And Vijakrishnan N., Irwin M.J. and Ye, W.,"Influence of Compiler Optimizations on System Power", *Proc. Of the 37th Design Automation Conference*, Los Angeles, CA, 2000

[6] LANCE Compilation Environment User's Guide, University of Dortmund, Dept. of Computer Science,http://LS12-www.cs.uni-dortmund.de/∼leupers, 1999

[7] Livermore Benchmarks: http://scicomp.ewha.ac.kr/netlib/ benchmark/livermorec

[8] Macii, E. and Pedram, M. and Somenzi, F.,"High-Level Power Modeling, Estimation, and Optimization",IEEE, *Trans. on CAD of ICs and Systems*, November 1998

[9] Muchnick, S. S., "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers, San Francisco, California, 1997

[10] Panda, P. R. and N. D. Dutt, N. D. and Nicolau, A., "Memory Issues in Embedded Systems-On-Chip", Kluwer Academic Publishers, 1999

[11] Sinevriotis, G. and Stouraitis, T., "Power Analysis of the ARM 7 Embedded Microprocessor", *Proc. 9th Int. Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Oct. 6-8 1999

[12] synopsys, "Power Products Reference Manual", v 3.5, Document Order Number: 23837-000 BA, 1996

[13] Tiwari, V. And Lee, M., "Power Analysis of a 32-bit Embedded Microcontroller", *VLSI Design Journal*, 1998(7)

[14] Tiwari, V., and Malik, S. And Wolfe, A., "Compilation Techniques for Low Energy: An Overview", *Proc. of the 1994 IEEE Symposium on Low Power Electronics*, San Diego, CA, 1994

[15] Tiwari, V., and Malik, S. And Wolfe, A., "Power Analysis of Embedded Software: A First Step towards Software Power Minimization", *IEEE, Trans. On VLSI Systems*, December, 1994

[16] Tiwari, V. and Malik, S. And Wolfe, A., "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing Systems*, August, 1996