

THIS PAPER PRESENTS A UNIVERSAL RECONFIGURABLE SOPC PLATFORM BASED ON A COMBINATION OF THE ATMEL AT94K FPSLIC DEVICE AND AN EXTERNAL MEMORY. THE PRESENTED PLATFORM INCREASES THE POWER OF THE FPSLIC DEVICE BOTH BY EXTENDING THE INTERNAL ADDRESS SPACE THROUGH AN INTRODUCTION OF A VIRTUAL PROGRAM MEMORY AND BY PROVIDING A TRANSPARENT INFRASTRUCTURE FOR FPGA RECONFIGURATION. THE PLATFORM IS DEMONSTRATED ON TWO SIMPLE DESIGNS THAT DEMONSTRATE BOTH ASPECTS.

Reconfigurable System on a Programmable Chip Platform

By: M. Daneš, P. Honzík, J. Kadlec, R. Matoušek, Z. Pohl of the Institute of Information Theory and Automation, Dept. of Computer Science & Eng. and Centre of Applied Cybernetics*

Introduction

Field-programmable gate arrays are configurable VLSI devices that can implement various logic functions. Classical SRAM-based FPGA chips introduced in 1984 were designed to be configured only once at the beginning of their operation and to enable a designer to improve the functionality after a device was shipped to the end user.

Now, almost two decades later, the current FPGA technology includes the concept of reconfigurability. The reconfigurability has to be looked at from two levels. One level is the actual dynamic reconfigurability of a device, while the other is a support built into the CAD tools supplied for the device. At present, devices that support limited or full version of limited reconfigurability are available, for example, the recently introduced devices from Xilinx (Spartan2, VirtexII) or Atmel (AT40K, AT94K), but it has not been incorporated into the supplied design tools [3]. The reason for the latter fact is the rather complicated methodology that has to be implemented in the tools [6].

This paper describes a universal reconfigurable SOPC platform built around the Atmel AT94K FPSLIC1 device. The platform enables a software programmer to take advantage of a possible hardware implementation of a user function while not being directly involved in the hardware design process. This is made possible by providing a universal transparent hardware-software interface and a library of pre-compiled configuration bitstreams that are selected by the programmer according to the required hardware functions.

The platform implements an extended virtual memory space that increases the standard 64KB address space that is accessible by the FPGA and AVR processor inside AT94K. The extended memory space can be used for AVR programs, data, and FPGA configuration bitstreams, which increases the power of the AT94K device [5]. Similar architectures can be found in [1], [2].

The paper is structured in the following way: Section 2 describes basic features of the AT94K device, Section 3 presents basic features of the presented platform, Section 4 shows the use of the platform on two sample applications, and Section 5 concludes the paper.

AT94K FPSLIC

The Atmel AT94K device (see [11] and Figure 1) consists of an AT40K FPGA [12], an 8-bit AVR microcontroller, a hardware multiplier, two UARTs, a two-wire serial port (TWS), three counters, a watchdog timer, and a 36KB SRAM memory that is partitioned to a 20-32KB AVR program space and a 4-16KB AVR data space. Prototyping boards based on this device are available from Atmel (ATSTK94, ATSTK594).

The AVR microcontroller can be programmed both in assembler or in C compiled, for example, by the Imagecraft C compiler or the free AVR gcc compiler.

The FPGA can be programmed by bitstreams generated by the Atmel-supplied Figaro place & route tool in combination with common design synthesis tools (Mentor Graphics Leonardo Spectrum, Synplify Synplify, etc.) [8], [9].

The AT94K device can implement both pure AVR or pure AT40K designs or their combination. The standard AVR-FPGA data interface consists of 8 data inputs, 8

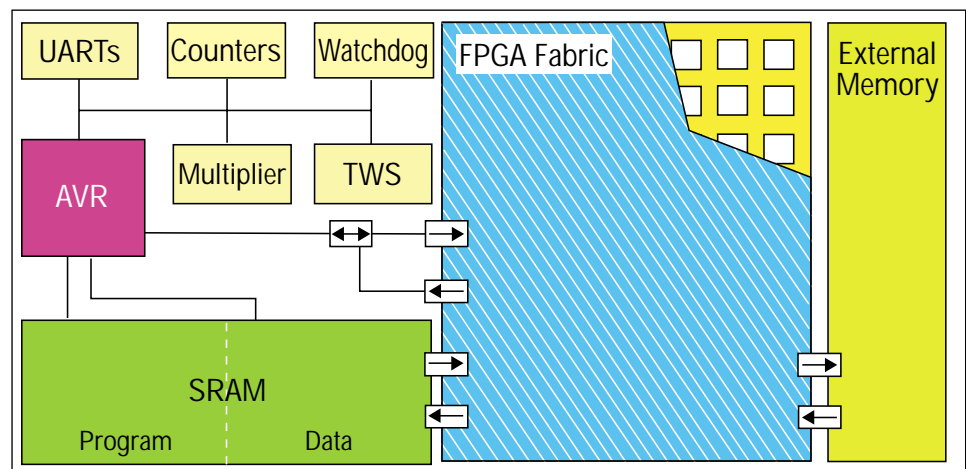
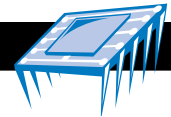


Figure 1: AT94K with an external memory. Arrows denote the direction of the data ports.

* This work has been partially supported by the EU IST programme under No. IST-2001-34016 and by the Grant Agency of the Czech Republic under No. 102/04/2137.

¹Field-Programmable System Level Integrated Circuit



data outputs, 16 FPGA select lines generated by the AVR, and 16 AVR interrupt lines generated by the FPGA. The FPGA can access both the AVR program area and AVR data area of the built-in SRAM. In addition, the AVR can reconfigure the FPGA using Mode 4 bitstreams and four FPGA configuration registers (FPGAX, FPGAY, FPGAZ, FPGAD) [10], [11].

AT94K alone is suitable for simple tasks that do not deal with large amount of data and that do not require high data throughput. Fortunately the AT94K target is supported by available C compilers, so programming the AT94K is a matter of routine C programming and FPGA design.

Its limits are mainly the size of its internal memory and the rather small size of the built-in AT40K FPGA. The internal memory is 36KB, out of which max. 32KB is available for program code and max. 16KB for user data. In the best configuration the built-in FPGA contains 50K equivalent gates. Also, the available C compilers do not provide any sophisticated interface to access the FPGA fabric.

Extending the AT94K Power

To increase the capabilities of AT94K it is necessary to increase mainly the real size of the program memory by implementing a virtual memory and the size of the data memory that stores FPGA bitstreams. Notice that a common size of a simple compiled AVR code is about 10KB; a common size of a moderate size bitstream in Mode 4 [10] is well over 100KB.

Both problems can be solved by using an external data memory. The proposed platform (Figure 2) uses an external FLASH memory to store fixed program and bitstream data that are copied at different times to specific locations of the internal SRAM memory. The FPGA is divided to a static part that implements an access to the FPGA reconfiguration data and the AVR program code, and to a dynamic part that contains user-defined designs; the designs can be reconfigured at run time.

Data Sharing between AVR and FPGA

Before the AT94K FPGA can be used as an AVR coprocessor, it is necessary to define data exchange schemes between AVR and FPGA. There are two possible schemes: to use registers implemented in the FPGA, or to use the internal SRAM.

The width of the FPGA registers is limited by the 8-bit AVR-FPGA data bus. The advantage of this scheme is its simplicity, a major drawback is its slow transfer rate: each AVR load or store instruction requires 2 clock cycles.

The SRAM exchange scheme is faster, since the FPGA can implement a DMA controller that can access the memory directly without an AVR support in one clock

cycle, which saves at least 4 clock cycles per operation (when processing one 8-bit data value at a time). Its disadvantage is the necessity to implement a data sharing or protection mechanism; in its simplest form it may be represented by a convention that a given memory area must not be modified by the AVR until the FPGA has finished the calculation.

Virtual Program Memory

The virtual program memory is often implemented using program overlays or memory paging. Since AVR is intended for simple tasks, we think that program overlays without memory protection are sufficient. The following text considers 2KB long overlays, since this size seems adequate for AVR application programs. The data memory can be extended using a similar concept.

The implementation uses the fact that the program memory can be accessed as data by the FPGA by setting bit 36 in the FPSLIC system control register [11]. The implementation consists of two parts: a software overlay support contained in the AVR BIOS, and a hardware DMA controller implemented in the FPGA that transfers overlays from an external memory to a given location in the program SRAM. The DMA controller consists of a context register that determines the starting address (its MSbits) of the overlay in the external memory and of a 12-bit counter that generates the LS address bits of the overlay address. A write to the context register initiates a data transfer. When the counter overflows, it means that the transfer has been completed and the controller generates an interrupt.

Since a sequence of locations as accessed by the AVR in the program memory maps to different clusters of locations in the data memory, it is necessary to translate the compiled overlay code. It is convenient to perform this translation at compile time, since the resulting data can be just streamed from the external memory to the internal SRAM memory by the DMA controller each time an overlay is required and no run-time address translation is necessary.

The overlay service is accessed through a BIOS function call with a parameter that identifies the required overlay. This parameter is translated and stored to the context register. The BIOS function ensures that on completion the memory contains a valid overlay, a jump to its starting address is then performed in the application program.

Programming Overlays in GCC

The overlay concept requires that a common C program is transformed to several functions with a common start function, all contained in one overlay that is brought into the AVR program memory by the AVR BIOS on demand. The start function is similar to the main function in C. From the start function another sub-functions are called that are not visible from the AVR BIOS. The AVR BIOS can download a particular overlay to the memory,

and give control to the start function in the actual overlay. The function for changing overlays is a loop that checks the ID of the actual overlay in the memory and compares it with that of the required overlay. When a different overlay is requested, it must be downloaded from the external memory (in our case, an external flash memory is used to store overlays). The overlay transfer process is carried out by the FPGA logic; when a user program calls a particular overlay, the call initiates an FPGA-controlled DMA transfer. On completion the FPGA generates an interrupt.

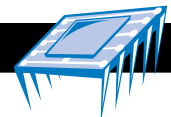
The AVR BIOS does not write into the program space or call any function from the overlay memory. The AVR BIOS knows only the overlay starting address; an overlay is executed by a function call to that address. Each overlay needs to communicate with the AVR BIOS. To do this a service table is located at a fixed address in the AVR BIOS. The service table describes calling addresses of services that are provided by the AVR BIOS. The services include registering an interrupt, or starting a timer. This organization ensures the independence of overlays; the only implementation-dependent part of an overlay is a header file with the definition of the service table.

FPGA Run-Time Reconfiguration

Run-time reconfiguration enables the designer to increase the functional density of the FPGA coprocessor, in other words, to download functions to the FPGA as they are needed. AT94K supports four configuration modes, out of which the simplest reconfiguration mode is Mode 4 (simplest = principally does not require an external hardware). Mode 4 reconfiguration is based on four configuration registers serviced by the AVR [10], [11].

The most convenient implementation of the extended bitstream memory uses one FPGA register for context (bitstream) selection and another for data passing. The static part of the FPGA implements an address register that consists of the context register (MS bits) and a counter. On writing to the context register the counter is reset. Each time the data register is read by the AVR the counter increments. When the top address specified in the bitstream header is reached, i.e. when the reconfiguration of the dynamic part is finished, the FPGA interrupts the AVR operation.

When not considering different execution time due to reconfiguration, the reconfiguration process is transparent to the application software. The access to the FPGA coprocessor is implemented as a special BIOS function, whose parameters are the required operation and its operands passed either as direct values in the case of the register transfer, or as a starting address of their location and their number in the case of the SRAM transfer. When BIOS detects a request for an operation different than the one currently present in the FPGA, it calls a function that reconfigures the FPGA.



This function first translates the requested operation to the context (address) of the corresponding bitstream and writes it to the context register. Then it sequentially reads 4-tuples of values and writes them to the four FPGA configuration registers (X, Y, Z, D) until the FPGA generates an interrupt.

Generation of Reconfiguration Bitstreams

The runtime reconfiguration of the AT94K FPGA fabric requires partial bitstreams. Such bitstreams reconfigure only a part of the chip while the rest is not affected in its operation. The Figaro design implementation tool provided by Atmel is meant to generate bitstreams that are not intended for partial reconfiguration of the FPGA. A special implementation procedure must be used to obtain such bitstreams [3]. The idea is to get several complete bitstreams with all different coprocessor contexts that contain the same placement and routing of the identical static part.

The Figaro tool works with a system of libraries. Any design component can be implemented as a macro and stored in a named library. The top-level design may contain instances of such components as black boxes (i.e. without a description of their content). Figaro will then search project libraries for components that fit the instance interfaces.

This feature is used in the described approach to implement and store different contexts of the reconfigurable coprocessor in libraries with different names. Since all contexts are implemented, the top-level design can be opened in a new project that includes only one of the context libraries. A complete bitstream is generated by performing all implementation steps.

The next complete bitstream with a different coprocessor context must contain the same placement and routing of the identical static part. This is achieved by

opening the same project under a different name with the placement and routing locked. Before the project is re-opened, the context library must be changed to the next coprocessor context. The Figaro tool then detects that the coprocessor has changed and it reimplements only the coprocessor without any modifications in the locked static part. This procedure is repeated until all coprocessor contexts have been generated.

To obtain partial bitstreams the complete bitstreams obtained in the previous step must be compared using the Figaro bitstream compression tool. The tool generates incremental changes that must be performed to switch from the configuration given by the base bitstream to the configuration given by the new bitstream. To be able to use the partial bitstreams for changing the coprocessor configuration all possible coprocessor context swap combinations must be generated. A direct approach leads to $n!$ combinations (each to each), where n is the number of contexts. A significant reduction of combinations is obtained by introducing a common reference coprocessor configuration, such as an empty contents or the most frequently used function; this approach decreases the number of necessary bitstreams to $2n$.

Sample Applications

The described platform (Figure 2) is presented on two simple applications: the first shows the use of overlays, the second demonstrates a transparent run-time reconfiguration of the FPGA triggered by the application program.

Program Overlays

Program overlays are user-defined functions compiled as a standard C code with a specific starting address, here denoted as *START_PROGRAM* (Figure 3). The overlays are implemented as data transfers from the external memory to specific locations in the program

memory. The *START_PROGRAM* value has to be set in accordance with the starting location used by the FPGA DMA controller. Once the overlay has been loaded, the application code performs a jump to the *START_PROGRAM* location (appl is a pointer to a function). When the overlay finishes, the control is returned to the main application, which can load and execute another overlay.

The *LoadOvly* function implemented in BIOS translates the overlay ID to its context in the external memory (its MS address bits) and stores it to the *VIRTUAL_MEM_CTRL* context register (Figure 2). This write initiates a DMA operation (implemented in the FPGA) that transfers a 2KB block of data from the external memory to the AVR program SRAM. Once the transfer is completed the FPGA generates an interrupt.

```
if (LoadOvly(ovlyID) != TRUE)
    PrintStr("OVERLAY LOAD FAILED\r\n");
else {
    appl = START_PROGRAM;
    appl();
}
```

Figure 3: An overlay use in an application program.

```
len = LoadData(&oper, data);
i=0;
while (i<(len/6)) {
    CallFPGA(oper,
        data[6*i], data[6*i+1], data[6*i+2],
        data[6*i+3], data[6*i+4], data[6*i+5],
        &results[3*i], &results[3*i+1],
        &results[3*i+2]);
    i++;
}
```

Figure 4: An FPGA call in an application program.

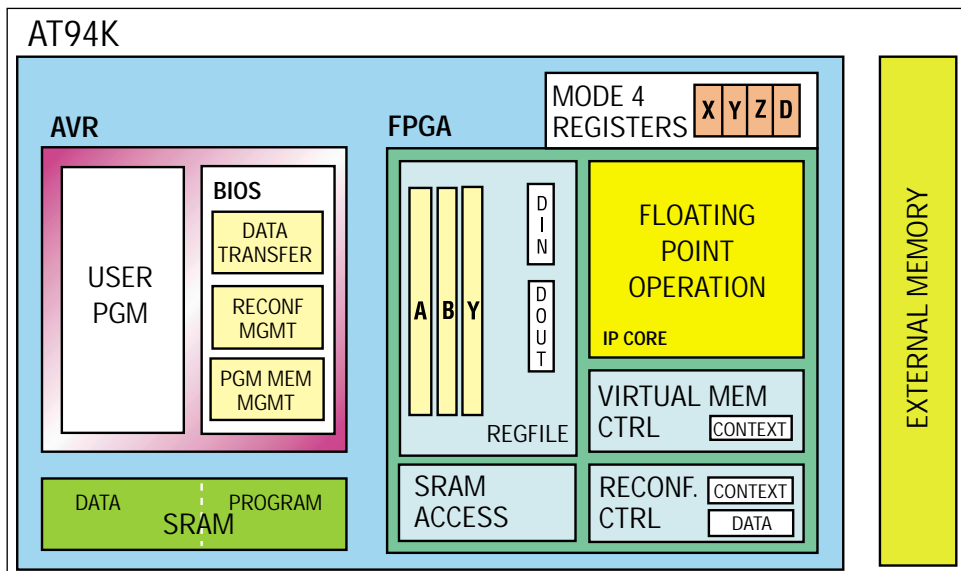


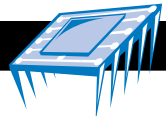
Figure 2: The AT94K-based reconfiguration platform.

Reconfigurable Coprocessor

The reconfigurable coprocessor implements different operations executed in the FPGA. The coprocessor can be accessed by an application program executed by the AVR microcontroller.

In the described application the coprocessor implements basic floating-point operations - ADD, MUL, DIV, SORT - in a 24-bit precision (1-bit sign, 6-bit exponent, 17-bit mantissa). As the AT40K40 FPGA is too small to contain all the operations at once, the coprocessor is reconfigured prior to calculation when necessary.

The performed task is simple: the AVR gets a data block from the serial port, then it uses the FPGA floating-point coprocessor to calculate results for these values.



The design consists of four main parts (Figure 2):

1. the floating-point coprocessor implemented in the FPGA, denoted as *IP CORE*,
2. the data management, i.e. the data transfer part of the AVR BIOS and the *REGFILE* and *SRAM ACCESS* blocks in the FPGA,
3. the virtual memory management, implemented in the *PGM MEM MGMT* part of the BIOS and the *VIRTUAL MEM CTRL* block in the FPGA,
4. and the reconfiguration controller, shown in BIOS as *RECONF MGMT* and in the FPGA as *RECONF CTRL*.

A sample application program built upon the infrastructure is shown in Figure 4. The code first gets a data block, then it calls the FPGA to process the data. The first item in the block determines which operation should be performed, this is stored in the *oper* parameter.

The FPGA infrastructure is transparent to the programmer. The only place at which the AVR-FPGA interaction can be noticed is the function call *CallFPGA*. Each operand in the example is encoded using 24 bits; the amount of parameters passed to the function is determined by the 8-bit architecture of the internal AVR-FPGA data bus. An additional parameter called *oper* parameter is passed to the AVR BIOS to determine if it

is necessary to reconfigure the FPGA. The BIOS then transfers the data to the coprocessor and gets the results.

Conclusion

This paper has presented a hardware platform that makes use of dynamic reconfiguration to increase a computational potential of a simple microcontroller. The platform uses an external memory to emulate a bigger program and data space, and to store reconfiguration bitstreams needed to program an FPGA-based accelerator connected to the microcontroller. Two sample applications have demonstrated the transparency of this approach in terms of application programming.

The most important advantage of the presented approach is the possibility to offer an unlimited number of different operations accelerated in the FPGA to an application programmer without his concern for digital circuit design.

Acknowledgements

The authors would like to gratefully acknowledge the support for this work provided by Atmel Hellas and Atmel Nantes.

References

- [1] Collahan, T. J., Hauser, J. R., Wawrzynek, J.: The Garp architecture and C compiler. In IEEE Computer, vol. 33, iss. 4, 2000, pp. 62–66.
- [2] Horta, E. L., Lockwood, J. W., Taylor, D. E., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In Proceedings of the Design Automation Conference, 2002, pp. 343–348.
- [3] Matousek, R., Danek, M., Pohl, Z., Kadlec: Dynamic runtime partial reconfiguration in FPGA. In ECMS2003, Liberec, 2003. pp. 294–297.
- [4] Matousek, R., Pohl, Z., Danek, M., Kadlec, J.: Dynamic reconfiguration of FPGAs. In 10th International Workshop on Systems, Signals and Image Processing (IWSSIP'03), 2003, pp. 288–291.
- [5] Wirthlin, M. J., Hutchings, B. L.: Improving functional density using run-time circuit reconfiguration. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 6, iss. 2, 2002, pp. 247–256.
- [6] X. Zhang, Kam W. Ng: A review of high-level synthesis for dynamically reconfigurable FPGAs. In: Microprocessors and Microsystems, No. 24 (2000), pp. 199–211.
- [7] FPLIC User's Guide and Tutorials, Revision 5 (Compatible with SystemDesigner 2.1), shipped with the ATSTK94 starter kit. Atmel, 2001.
- [8] FPLIC System Designer 3.0 User Guide. Atmel, 2003.
- [9] Atmel FPGA Integrated Development System. Atmel, 2001.
- [10] FPLIC on-chip Partial Reconfiguration of the Embedded AT40K FPGA. Atmel, 2002.
- [11] AT94K Series Field Programmable System Level Integrated Circuit. Atmel, 2002.
- [12] AT40K 5K to 50K Gates Coprocessor FPGA with FreeRAM. Atmel, 2002.

IDE, Compilers, ICD, Simulators, Programmers...

Emulators

8051 **PIC** **AVR**
80196 XEMICS
MSP-430 SENSORY

Phyton 718.259.3191
www.phyton.com

Integrated Development Tools for Embedded Microcontrollers

Atmel: T89C51RB2/RC2/RD2, T89C5111/5112/5115, T89C51AC2, T89C51CC01/02/03, TS8xC51U2, AT8xLV51/52/55, AT87F51/52/55, AT89C1051/2051/2051x2/4051, AT89C51/52/55/55WD, T89C51B2/IC2, AT87F51RC, TS8xC51RA2, AT90S, ATtiny and others...

Philips: LPC760/761/762/764/767/768/769, LPC932/9xx, 80C31X2/51X2/52X2/54X2/58X2, 89C51RA2/RB2/RC2/RD2, 8xC660/662/664/668, P87C552/554, 87C652/654 and others...

Intel: 80C31/32/51/52, 87C51FA/FB/FC, 87C51RA/RB/RC, 8xC196KC/KD, 8xC196MC/MD/MH, 8xC196CB, 8xC196NT and others...

Winbond: W77C32/58; W77E58, W77L32, W77LE58, W78C54, W78C58, W78E516B, W78E51B, W78E52B/54/58, W78IE54, W78L51/52/54, W78LE51/52/54/58, W78LE516/532, W78LE52, W78LE54, W78C51D, others...

Dallas Semiconductor: DS87C310/320/520/530 and others...

SST: 89C59, 89F54, 89F58 and others...

Microchip: The entire PIC12, PIC16, PIC17, and PIC18 families including PIC16F627/628, PIC17C756, PIC16F877A, PIC16F818/819, PIC18F452, PIC18F458, PIC18F6620, PIC18F6720, PIC18F8620, PIC18F8720, PIC18F4320 and others...

Texas Instruments: The entire MSP430 family, TAS1020A, TUSB3220

Xemics: XE88LC01/05, XE88LC02, XE88LC06/06A

Sensory: RSC4xx

Hi-End Features @ Affordable Prices