Design of a RISC Microcontroller Core in 48 Hours

D. Šulík [1(2)] M. Vasilko[1] D. Ďuračková[2] P. Fuchs[2]

[1]Microelectronics Systems Research Group
School of Design, Engineering & Computing, Bournemouth University
Fern Barrow, Poole, Dorset BH12 5BB
United Kingdom

E-mail: M.Vasilko@computer.org

[2]Faculty of Electrical Engineering & Information Technology
Slovak University of Technology
Ilkovicova 3, 812 19 Bratislava
Slovakia

Abstract

In this paper we present a design case study using Handel-C—a recently developed programming language for compilation of high-level programs directly into FPGA hardware. The design is an 8-bit RISC microcontroller core with 33 instructions, prescaler and a programmable timer. Handel-C was used throughout the entire design and debugging flow. The RISC microcontroller design was implemented in on the XESS XS40 FPGA board with Xilinx XC4010XL FPGA . The overall design, including debugging, testing and the FPGA implementation was completed in less than 48 man-hours.

1 INTRODUCTION

Increasing performance and gate capacity of recent FPGA devices permits complex logic systems to be implemented on a single programmable device. Such a growing complexity demands design approaches, which can cope with designs containing hundreds of thousands of logic gates, memories, high-speed interfaces, and other high-performance components. One category of such design approaches are design methodologies based on languages derived from traditional programming languages such as C, Pascal, Java or others. These allow designers to use the familiar language syntax to develop hardware systems at high level.

In this paper we present a design case study of a RISC microcontroller core, designed using a hardware compilation design flow. A C-like language called Handel-C is used for the hardware design. The goal of this work was to evaluate the feasibility of using Handel-C for rapid design and prototyping of microprocessors.

In the following Section 2 we outline the architecture of the RISC microcontroller implemented in this design case study. Section 3 details the implementation methodology, including some real examples of Handel-C code used in the design and the overall project details. The paper concludes with a summary in Section 4.

2 DESIGN ARCHITECTURE

A simplified architecture of the RISC microcontroller core implemented in this case study is shown in Fig. 1. The core uses Harvard architecture, where program memory and data

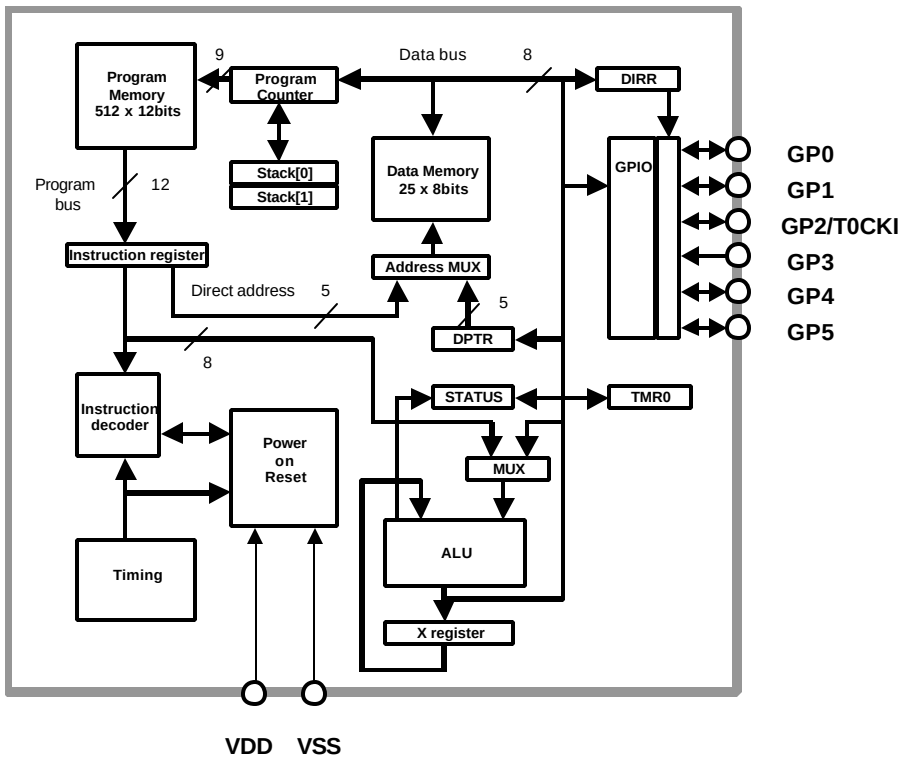

Figure 1. RISC Microcontroller Core Architecture

memory are accessed using separate dedicated busses. When compared to von Neuman processor architectures, the Harward architecture improves the bus bandwidth as in von Neuman architectures both program and data memory is being accessed through a shared bus.

The RISC processor core provides an 8-bit ALU with a working register X. The ALU supports simple arithmetic operations, including addition, subtraction, shift and Boolean logic operations. Register X is an 8-bit working register used by ALU operations. The ALU provides also number of flags to indicate various conditions after performing the arithmetic operations. The supported flags include *Carry*, *Digit Carry*, and *Zero.*

An 8-bit timer (TMR0) and an 8-bit prescaler module was provided as a part of the RISC microcontroller architecture. This is a fully programmable unit, which can work with either internal or external clock signals.

In order to avoid development of dedicated compilation tools for this RISC microcontroller, the instruction set was designed to be compatible with the instruction set of one popular microcontroller family. 33 instructions are provided, implementing the variety of arithmetic, logic and branching operations. All instructions are executed in a single-cycle, except for program branching instructions, which take two cycles. Separating program and data memory allows instructions to be sized differently than the 8-bit wide data word. Instruction operation codes (opcodes) are 12-bits wide.

The RISC microcontroller core can use both direct and indirect addressing to access its register files and the data memory. All special function registers including the program counter are mapped into the data memory. An illustration of the core's program and data memory map is shown in Fig. 2.

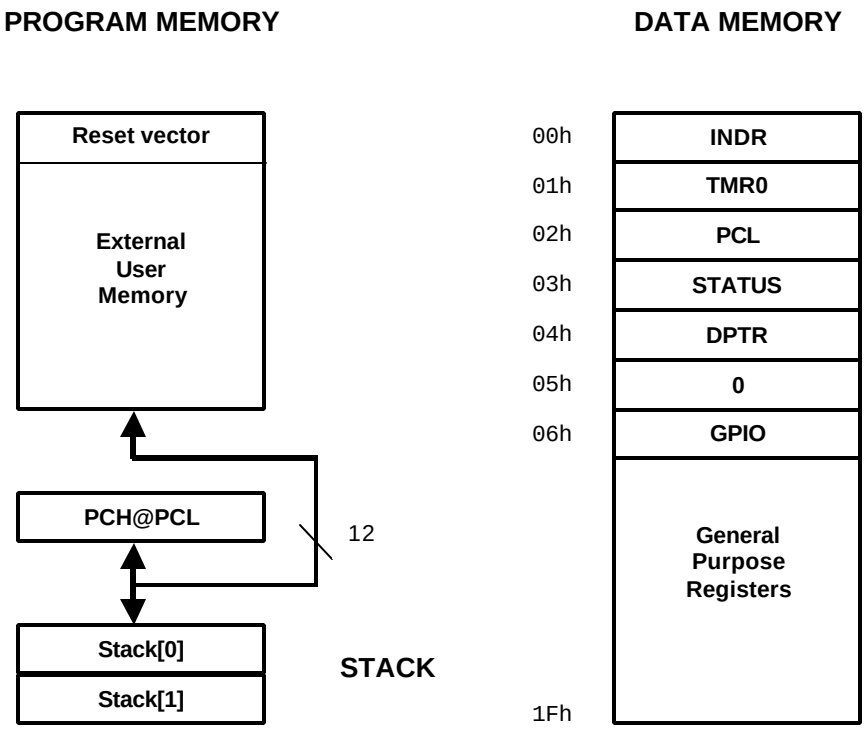

Figure 2. RISC Microcontroller Core Memory Map

2.1 INSTRUCTION FLOW DESCRIPTION

It takes four system clock cycles to execute one complete instruction cycle. For example, for a 12MHz clock frequency, a single-cycle instruction will take 333ns while two-cycle instruction will take 666ns.

The instruction cycle consists of four Q-cycles (Fig. 3). The instruction fetch takes one Q-cycle, while decode and execute take another instruction cycle each. If an instruction results in a change of the program counter (e.g. JUMP instruction), then two cycles are required to complete the instruction cycle.

The fetch cycle begins with a program counter (PC) incrementing its value by one. The

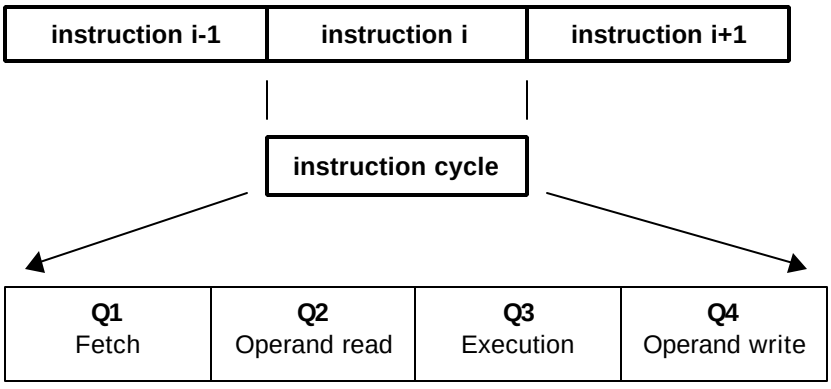

Figure 3. Instruction flow

instruction from a program memory is latched into the Instruction Register (IR) in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

2.1 SUMMARY OF FEATURES

The implemented RISC microcontroller core architecture provides the following main features:

- 33 single-word instructions (12-bit wide)
- instruction-level compatibility with one of the popular microcontroller families
- 8-bit wide data path
- external program memory of 512 x 12 bits
- several special registers + 25 Bytes of data RAM
- two-level deep hardware stack
- direct, indirect and relative addressing modes for data and instructions
- 8-bit real time clock/counter (TMR0) with 8-bit programmable prescaler
- emulation for a SLEEP mode

3 DESIGN METHODOLOGY AND IMPLEMENTATION

The entire RISC microcontroller was designed using the Handel-C language [1]. Handel-C is a special programming language designed to enable compilation of C-like programs into synchronous digital hardware. Although the task could have been accomplished using traditional Hardware Description Languages (e.g. VHDL or Verilog), we have opted for Handel-C as we aimed to evaluate its rapid design capabilities and its suitability for the design of soft intellectual property cores.

3.1 HANDEL-C DESIGN FLOW

A typical design flow using the Handel-C based hardware compilation is shown in Fig. 4. An input design is written using the Handel-C language. Handel-C provides new language constructs, which allow description of parallel digital hardware. A rich set of macro libraries is provided to simplify the routine tasks.

First the design is coded using the Handel-C language. The functionality of the design can be verified using a Handel-C simulator called HSIM. A Handel-C compiler is used to produce the netlist suitable for the simulation with HSIM. Once satisfactory simulation results have been achieved the Handel-C compiler can be used again to generate the FPGA gate-level netlist. This is performed using fast hardware compilation process after which a result summary is provided for the user. The netlist formats created by the Handel-C compiler (version 2.1) are either Xilinx XNF or EDIF.

Target technology-specific placement and routing tools are used next to map the gate-level netlist generated by the Handel-C tools into the targeted FPGA device. In our implementation we have used Xilinx M2.1i tools, as the targeted FPGA was Xilinx XC4010XL [2]. The FPGA placement & routing tools will produce a final layout of the FPGA design. From this layout, the FPGA tools can also generate an FPGA configuration bitstream, which will be used to program the target FPGA device.

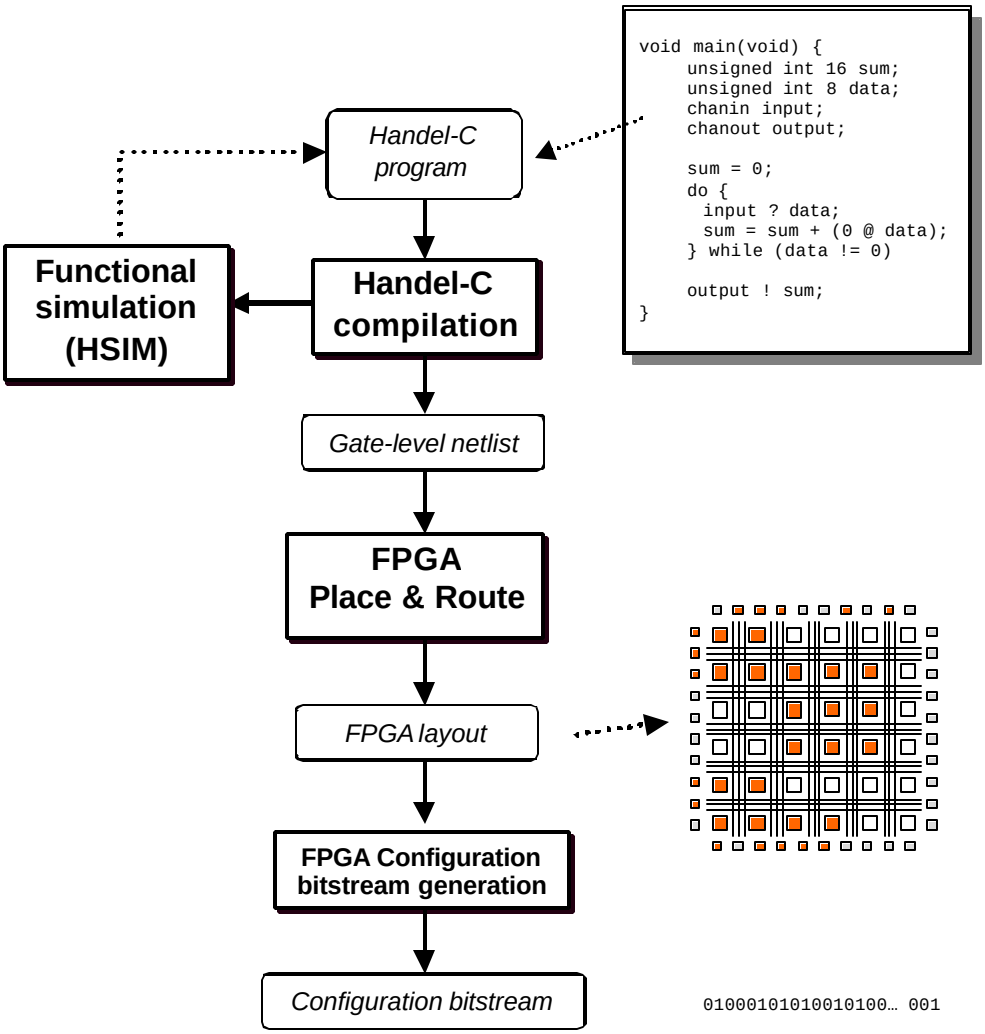

Figure 4. Handel-C hardware compilation design flow

3.2 HANDEL-C PROCESSOR IMPLEMENTATION

The entire RISC microcontroller design was implemented on XESS Corporation XS40 FPGA board, v1.3 [3]. On-board Xilinx XC4010XL FPGA and a separate 8-bit x 32kBytes RAM were used in the implementation.

The following sections describe the implementation of the selected parts of the microcontroller core architecture in the Handel-C language.

3.2.1 Instruction Cycle

Figure 3 indicates that each instruction requires 4 Q-cycles. In our design we have used dedicated Handel-C routines to deal with each instruction phase:

Q1 **Instruction fetch.** During this phase the instruction is read from the program memory (held in the on-board RAM external to the FPGA) and loaded into the instruction register IR. On-board RAM is only 8-bit wide, while the instruction word width is 12 bits. Therefore 2 clock cycles are required to read the entire 12-bit instruction.

The Handel-C macro **`fetch()`** implements the fetch instruction cycle phase:

```
macro proc fetch()
{
      // lower part of the instruction
      IRL = (includeNOP == 0 ? PROGRAM[PCH @ PCL @ 0] : 0);
      par {
            IRH = ( includeNOP == 0 ?
                  PROGRAM[PCH @ PCL @ 1]<-4 : 0);
            if (noincPCL) delay;
            else par {
                  PCL = PCL + 1;
                  PCH = (((PCH @ PCL) + 1)[8:8] == 1 ? 1 : 0);
            }
            noincPCL = 0;
      }
}
```

Note the use of special Handel-C constructs in the above code:

- **`par`** construct forces parallel execution the statements within the `{}` block
- `@` operator indicates bit concatenation
- **`word<-N`** operator returns least significant bits from the **`word`**

Q2 **Operand read.** Depending on the instruction code the operand will be read either from the data memory or from a special function register. The microcontroller core also selects between direct and indirect addressing modes. If register INDIR is the source register (`IRL[4:0]==0`), the 5-bit address stored in register DPTR is used to indirectly access the operand stored in a register. The operand is loaded in the temporary register TMP. If the operand address is greater than 6, data memory space will be accessed instead.

```
macro proc operandREAD()
{
      switch( (IRL[4:0]==0) ? DPTR<-5 : IRL[4:0] )
      {
            // read operand from one of the registers
            case 0: TMP = 0; break;
            case 1: TMP = TMR0; break;
            case 2: TMP = PCL; break;
            case 3: TMP = STATUS; break;
            case 4: TMP = DPTR; break;
            case 5: TMP = 0; break;
            case 6: TMP = rdGPIO(); break;

            // read operand from the data memory
            default:
                  TMP = DATARAM[(IRL[4:0]==0) ?
                              ((DPTR<-5)-7) : (IRL[4:0]-7)];
                  break;
      }
}
```

Q3 **Execution.** During this phase the TMP register will be used as a source and destination of the operation performed by the instruction. If the instruction affects any flags in the status register, these have to be evaluated in parallel with instruction execution. For example, to evaluate the state of flag Zb in the status register efficiently, the following shared Handel-C expression was used:

```
shared expr zero(TMP,X)
```

Another example is the addition instruction `ADDXF`, where four processes are executed in parallel in one clock cycle:

```
...
case 7:   // 00 0111  = instruction ADDXF s,d decoded
      par {
            TMP = addxf(TMP,X);
            // flags Zb,Cb,DCb
            Cb = carry(TMP,X);
            DCb = dcarry(TMP,X);
            Zb = zero(TMP,X);
      }
      break;
...
```

Special coding was required for instructions affecting program control flow (e.g JUMP, CALL) and those few instructions requiring two instruction cycles. Statement `addxf(TMP, X)` evaluates sum of universal X register and contents of TMP register.

Q4 **Operand write.** The result of the operation performed in phase Q3 stored in register TMP is written to the destination location. This process is similar to that of operand write described in paragraph Q1 above.

3.2.2 Memory

In the presented RISC microcontroller, the memory is separated into program and data memory (Fig. 2). Stack is also provided as a part of the microcontroller architecture. This section outlines the implementation of memory components using Handel-C.

Program Memory. The program memory in this design was implemented using the XS40 on-board RAM, which is external to the FPGA. Handel-C allows definition of external memory interfaces using the `ram` keyword with `offchip = 1` option:

```
ram unsigned int 8 PROGRAM[1024] with {
    offchip = 1,
    westart = 1,
    welength = 1,
    data = {"P10", "P80", "P81", "P35", "P38", "P39", "P40", P41"},
    addr = {"P57","P59", "P84","P83", "P82", "P79", "P78",
            "P5", P4", "P3"},
    we = {"P62"},
    oe = {"P61"},
    cs = {"P65"}
};
```

Other options supplied in the `ram` construct above, define the actual connections between the FPGA pins and the memory data, address and control signals.

Data Memory. The data memory contains number of special function registers and the general-purpose registers (Fig. 2). In our design, this was implemented using a register file inside the FPGA in order to allow easy access to the registers. In total, 25 registers were

implemented in the data memory. A trivial Handel-C declaration was used to create the data memory:

```
ram unsigned int DATARAM[25].
```

Stack. A 9-bit wide 2 levels deep LIFO stack was implemented. A CALL instruction will push the current stack value stored in stack[0] into stack[1], and then push the incremented current program counter value into the stack[0]. In our Handel-C implementation, the stack operation is coded for the CALL and RETLX instructions as follows:

```
...
case 0:    //10 00 = RETLX xx
      par
      {
            //flags none
            PCL = Stack[0]<-8;
            PCH = Stack[0][8];
            Stack[0] = Stack[1];
            TMP = IRL;
      }
      break;

case 1:    //10 01 = CALL xx
      par {
            //flags none
            PCL = IRL;
            PCH = 0;
            Stack[0] = PCH@PCL;
            Stack[1] = Stack[0];
            TMP = X;
      }
      break;
...
```

3.2.3 I/O ports

Input/output ports of the RISC microcontroller core can be easily implemented in Handel-C. All off-chip interfaces other than RAMs are declared with the interface keyword. This keyword specifies the type of interface required and the type values associated with the objects arriving from the interface. In our Handel-C implementation all ports were defined as follows:

```
interface bus_ts() IOPORTx(GPIO[x], DIRR[x]==0);
```

where x is the port index. This type of an interface allows Handel-C to perform bi-directional communication via external pins, where `DIRR[x]==0` is the condition when the pin will be driven by the value stored in the `GPIO[x]` register.

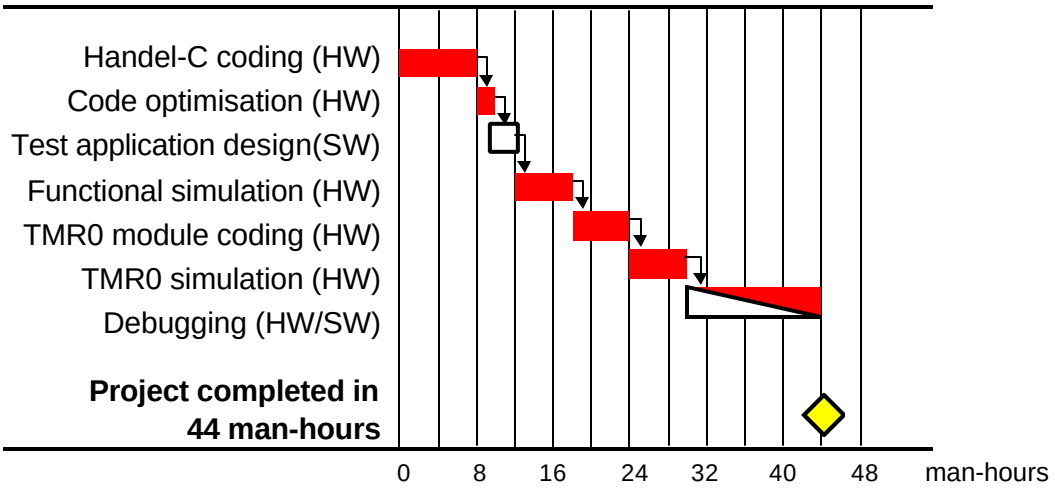

Figure 5. Project timescale.

3.3 TESTING METHODOLOGY

Functional simulation using the HSIM simulator was used to verify the operation of the RISC microcontroller core in the early stages of the design (as is indicated in Fig. 4). Small testbenches were written to test specific architectural components.

In order to generate the FPGA design layout and the configuration bitstream the result from the hardware compilation was processed by the Xilinx Alliance M2.1i. In the final stage, the XS40 FPGA board was loaded with the design configuration. Once the first working hardware was implemented, the design was tested using small programs written using the assembly language of the RISC microcontroller. Numerous programs were developed for this purpose, starting from simple control routines for the LED display, pulse-width modulator, to a terminal-based game, which fully exercises the microcontroller features.

3.4 PROJECT TIMESCALE

The entire project, including design coding and simulation using the Handel-C tools, development of software test benches, FPGA implementation, debugging and hardware prototyping took less than 48 hours. The actual project timescale, as recorded during the project, is shown in Fig. 5.

Coding of the presented RISC microcontroller took about 10 man-hours. This has included code optimisation, during which the code structure was improved and coding redundancies were removed. After the coding was completed, a simple assembly language program was developed for testing of all microcontroller instructions, which was initially used during for testing in a functional simulation with HSIM.

After the basic core functionality was validated, an implementation of the timer module (TMR0) was developed. This task took about 6 man-hours and involved the investigation of the most suitable method of external clock synchronisation and management of exceptions. Functional simulation of the timer module and functional debugging of the entire RISC microcontroller was performed in the following step.

Table 1. RISC microcontroller implementation statistics

Design size	338 CLBs
Size of the design gate-level netlist (as reported by the Handel-C compiler)	884 gates 125 inverters 202 latches 11 others
Xilinx XC4010XL device usability	84%
Max. clock frequency (-1 speed grade, design obtained using a "push-button" design flow without any user-specified optimisation)	11.88 MHz
Handel-C compilation time for simulation *(Handel-C program ® HSIM netlist, Pentium II 120MHz PC)*	8 seconds
Handel-C compilation time for hardware generation *(Handel-C program ® XNF gate-level netlist, Pentium II 120MHz PC, performed up to 7 times during the project)*	5 minutes

As indicated in Section 3.1, in the following steps the Handel-C design was compiled into a gate-level netlist and mapped onto the target FPGA using the standard Xilinx tools. The final design was debugged in real-time hardware in order to rectify any outstanding problems.

Table 1 summarises the overall implementation statistics from this project.

With Handel-C compiler it took only few seconds to generate the netlist suitable for simulation using the HSIM tool. XNF[1] gate-level netlist was generated with the Handel-C compiler on a low-spec PC within only few minutes.

The clock frequency possible for our design is about 30% higher than that of comparable off-the-self microcontroller devices. It should be noted, however, that the FPGA device used in our implementation is based on a more advanced technology than that of compared microcontrollers.

4 CONCLUSIONS

We have demonstrated a RISC microcontroller core design case study using the Handel-C hardware compilation design flow. While the coding of the microcontroller design took only 12 man-hours, the entire design has completed and tested in a real FPGA hardware in less than 48 man-hours.

The design was implemented in the Xilinx XC4010XL FPGA on the XESS XS40 board. For the presented design case study, the Handel-C design methodology has proven invaluable in providing a very rapid FPGA design and prototyping path.

ACKNOWLEDGEMENTS

The authors wish to thank to their colleagues, Darrell Gibson and Steve Holloway for their help and assistance during this project. The support of Embedded Solutions Ltd for parts of this work is gratefully acknowledged.

[1] Xilinx Netlist Format

REFERENCES

1. Embedded Solutions Ltd, “Handel-C Language Reference Manual “, version 2.1, available from `http://www.embedded-solutions.ltd.uk/`
2. Xilinx, Inc., “The Programmable Logic Databook”, 1998
3. XESS Corp., “XS40, XSP Board V1.3 XS40, XSP Board V1.3 User Manual”, available from `http://www.xess.com/`