# FUTURE DIRECTIONS IN COMPUTER ARCHITECTURES CURRICULA: SILICON COMPILATION FOR HARDWARE/SOFTWARE CO-DESIGN

**A. C. Downton, M. Fleury, R. P. Self and P. D. Noakes**

Department of Electronic Systems Engineering, University of Essex, UK
email: acd@essex.ac.uk

### Abstract

*Courses in Computer Architectures are increasingly becoming restricted to Electronics and Computer Engineering degrees rather than Computer Science degrees, as Computer Science and Software Engineering expands and student demand increasingly favours software rather than hardware aspects. As a result, a declining proportion of Computing students understand the relationships between software structures and their corresponding hardware implementation and performance - critical issues in embedded systems and hardware-software co-design. In an attempt to redress this balance, we propose a new top-down Computer Architectures course structure that uses silicon compilation as an abstraction to present Computer Architectures in a top-down C-software-oriented format. This approach allows first year undergraduate Computing (and Electronics) students to directly implement and modify minimal processor architectures in a C-language form, without the necessity to explicitly learn assembly language programming or a microprocessor programming environment.*

## 1    Introduction

The teaching of Computer Architectures in degree courses is at a crossroads. Demand for Computer Engineering within Computing-related degrees is declining as software abstractions are layered on top of each other obscuring the underlying hardware; Electronic Engineering attracts only a small fraction of the UCAS applicants applying to Computer Science courses. Most students are increasingly driven by vocational rather than educational goals and question the merits of understanding computer architectures in comparison with, for example, programming and database courses, which clearly provide skills that are immediately useful once they graduate. How then are we to infiltrate an understanding of computer design and performance principles into such students, at least some of whom may then be stimulated to join the rewarding but largely hidden engineering world of embedded systems and hardware/software co-design?

This paper proposes a radical approach to teaching computer architecture at foundation level based upon replacing the traditional bottom-up (gates/circuits/CPU/assembly language) lecture course with a top-down (high-level-language/CPU/ALU/logic) hands-on approach built on silicon compilation applied to platform FPGAs. This top-down approach is designed to exploit students' perception that programming is an important vocational skill while imbuing a knowledge of software engineering and core performance-computing concepts such as concurrency, parallelism, software-to-hardware mapping, data structures and efficiency, and comparative computer architectures. In addition, the approach capitalises on students' enthusiasm for learning by doing rather than listening, through experimental laboratory assignments which allow students to program not just computer applications, but also computer architecture and performance variants such as new instructions and pipelining mechanisms directly through emulation of the architecture on FPGAs. It also incidentally introduces hardware-software co-design, a key technology for the future, at first year level, and in a format which can motivate a wide range of computing students whose mainstream interests may actually be in networks, multimedia, signal processing and numerous other areas rather than ECAD.

## 2    Handel-C vs. Hardware Description Languages

Our first-level approach is based upon emulating a limited subset of the ARM processor in Handel-C using a Xilinx Virtex FPGA. Emulation software is written as a number of functions that act as wrappers encapsulating combinational and register-transfer logic operations. The emulation functions

then directly implement logic equivalent to the ARM ALU and control logic, thus retaining maximum compatibility with the C programming model while enabling a (reconfigurable) processor architecture to be built from a set of library components.

Such operations can alternatively be presented as black box logic functions in a conventional description of a computer architecture without needing to focus on timing issues, thus closing the loop with conventional computer architecture descriptions. The simple Handel-C paradigm that combinational logic generates propagation delays but no execution clock cycles, while each register store operation requires a clock edge, maps exactly from FPGAs to CPUs. The only difference is that the FPGA unrolls consecutive statements into successive logic functions, whereas the CPU recycles each statement through the same general purpose ALU, achieved by repeated calling of appropriate emulation functions in a sequence defined by the machine code ARM program stored in Select RAM on the FPGA.

This approach is not inherently new, as it has been explored in the past using languages such as VHDL or FPGA foundation tools. However, we base the course around a silicon compiler (Handel-C[1]) which is both a subset and a superset of conventional C. By using Handel-C, the course is immediately approachable to students whose other initial programming experience is likely to be based around C, C++ or Java. Handel-C removes sophisticated features of C such as floating point support and maths libraries that are irrelevant to a foundation Computer Architectures course. At the same time it enhances the language with data types that directly map to hardware (variable length integer words) and important concepts which are active elements of current architectural development both in software and hardware (concurrency, pipelining, data parallelism and channels). In contrast, courses based on VHDL or similar hardware-oriented languages typically assume significant bottom-up knowledge of logic gates, timing, signals and voltages (though this isn't strictly necessary), which makes it impossible to introduce such courses until the final year, where much of the benefit they could offer in providing an integrated view of the relationships between hardware and software has already been lost. Last but not least, the approach allows Electronic Engineers to re-assert the importance of hardware engineering to the design of software systems, a discipline which has gradually lost favour amongst computing students since the heyday of microprocessors in the early 1980's.

## 3    A first-level top-down software-oriented approach to Computer Architectures

The top-down software-oriented introduction to Computer Architecture starts from software (Handel-C) rather than hardware and assembly language. This has immediate advantages of familiarity with (and reinforcement of) a related first-level programming language, and avoidance of the pedantry imposed by most assembly languages, which is often a serious obstacle to students' progress. Furthermore, the algorithmic expression formalised through a programme description is a clearer and more precise definition of processor operations than more conventional textual/graphical descriptions. At the top level, processor architecture can be characterised through the fetch - decode - execute cycle as follows:

```
void ARMProcessor()
{
    // executing process
    while(Running){

        // sequential pipeline
        seq{
            Fetch(IStore);
            DecodeAndExecute(DStore);
        }
    }
}
```

---

[1] Handel-C outputs RTL VHDL, and so could also be regarded as a shorthand version of VHDL, making it relevant also to those with a traditional Electronic Engineering background.

where `IStore` is the area in the FPGA's memory which holds binary instructions and `DStore` is a corresponding data area. The Fetch function simply returns a 32 bit binary instruction integer from which sub-fields are extracted which access the opcode and up to 3 operands (Handel-C integer extensions specifically support these sort of operations). In a first-level course, only a subset of the full instruction set would typically be presented, restricted to basic single-clock-cyle register transfer and ALU operations, allowing the more esoteric binary instruction formats to be ignored in the interests of simplifying the Fetch function.

The Decode and Execute function then consists of a series of switch statements, each implementing the function of a particular instruction opcode. At this first level, each case statement implements the register transfers and ALU operations required of the specified instruction directly, rather than attempting to emulate the processor logic exactly. For example, the following code fragment of the Decode and Execute function implements Register Indirect Load, Store, Add and Branch instructions in a way which is very close to the register transfer language commonly used in Computer Architectures textbooks:

```
// decodes instructions and updates the register file
void DecodeAndExecute(Word* DStore)
{
  switch(Opcode){

  // load register from main memory
  case LDR :
        Registers[Rd] = DStore[Registers[Operand2 <- 4]];
        break;

  // store register to main memory
  case STR :
        DStore[Registers[Operand2 <- 4]] = Registers[Rd];
        break;

  // add registers (R3 = R1 + R2)
  case ADD :
        Registers[Rn] = Registers[Rd] + Registers[Operand2 <- 4];
        break;

  // unconditional branching
  case B :
        Registers[PC] = (unsigned 16)(0 @ Operand2);
        break;

  // branch if registers used in the last compare were equal
  case BEQ :
        if (Compare == 0){
                Registers[PC] = (unsigned 16)(0 @ Operand2);
        }
        else{
                delay;
        }
        break;
```

What is built is thus a distributed logic implementation of the processor ALU, with independent hardware for each instruction execution, rather than an optimally shared logic design. Our first-level processor implementation comprises only three pages of Handel-C programme code, so is readily assimilable in its entirety by first year students. Starting from this base, students can then undertake a variety of experimental investigations, some of which would be impossible with a conventional processor:

- Cycle-by-cycle confirmation of register transfers for specified fragments of machine code
- Measurements of total execution time in clock cycles
- Implementation of different and additional instructions, addressing modes, etc.

- Modifications to the register architecture
- Addition of architectural features, e.g. a subroutine stack frame, a barrel shifter, hardware or software interrupt mechanisms, etc.
- Examination of performance aspects of the design; e.g. by replacing `seq` with `par` in the top level fetch - decode - execute code above, instruction execution is pipelined, thereby increasing instruction throughput (though special account then needs to be taken of pipeline stalling on branch and other multi-cycle instructions, if implemented).

It is obvious that this same basic framework of Handel-C and FPGAs can equally well be used as a basis for presenting practical content for a first-level Digital Systems course. (Similar courses have been presented using FPGAs with Xilinx foundation tools in the past, but the advantage of using Handel-C is in the added coherence with corresponding Programming and Computer Architectures courses.) In this case, one would expect that the Handel-C descriptions of logic components would be developed to a much more detailed algorithmic level (identifying individual signal and clock lines), providing a suitable framework for the more detailed second-level Computer Architectures course described in the next section.

## 4    The second-level Computer Architectures course

A key feature of the first-level course outlined above is that it is conceptually complete without requiring any access to actual microprocessor hardware or software development tools (though these may be used by staff in preparing machine code fragment examples). In a second-level course however, such tools need to be introduced and integrated with the first-level environment.

Our Computer Architectures courses are based around the ARM processor architecture, and the required software tool is the ARMulator emulation package, which runs on a PC. Figure 1 shows the standard software development process for the ARMulator, and Figure 2, how this can be used to feed binary instruction data into the programme memory of the Virtex FPGA.
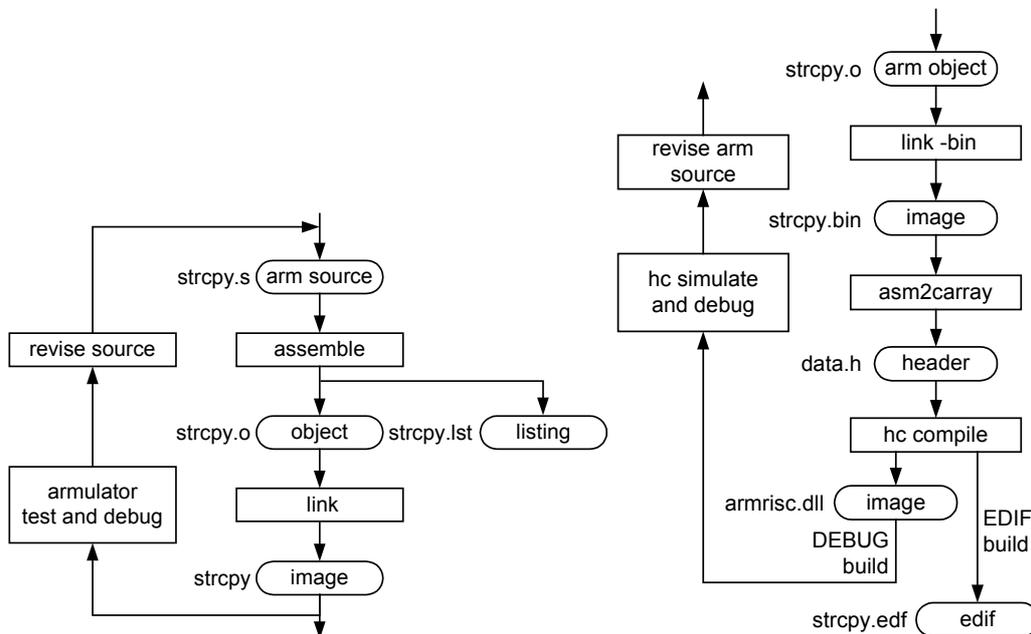


*Figure 1: ARMulator simulate and test cycle.  Figure 2: Handel-C simulate and test cycle*

Though the interface appears straightforward, it implies a considerably enhanced implementation of the Handel-C ARM emulator, since the ARMulator can generate any valid ARM binary code, but the first-level ARM emulation described in Section 3 above had very restricted functionality. For example, Table 1 and Figure 3 below show a more realistic emulation model for the ARM which:

- models ALU, control unit, registers and status/condition code bits more accurately;
- implements complexities ignored in the first-level model, such as delayed branching and branch with link in both conditional and unconditional forms
- implements a wider range of word and byte-oriented addressing modes

This implementation inevitably requires a considerably more detailed low-level Handel-C code description of the processor, as is exemplified below in the code fragment implementing the generic adder, and its utilisation by several variants of the ADD instruction. Although these algorithms are described at a signal level and make heavy use of Handel-C-specific language extensions, they will be readily understandable by students who have also taken a Digital Systems course with examples written in Handel-C.

| Signal Name | Description |
|---|---|
| Register file: | |
| ir.a | A-operand register address |
| ir.b | B-operand register address |
| ir.d | Destination register address |
| ir.im | Control flag identifying that B-operand is immediate |
| ir.imo | B-operand immediate value |
| a-, b-, r-bus | A, B, and Result busses |
| dwrite | Write enable for destination register |
| Control Unit: | |
| ir.opcode | Opcode of the data processing instruction |
| ir.type | Type of execution unit (*i.e.* data process, load and store) |
| ir.code | Identifier relating to the conditional execution of the current instruction |
| ir.s | Update status register |
| update | ALU updates status register.  Required for TST etc. |
| status | Z, N, C, V status signal outputs from ALU |

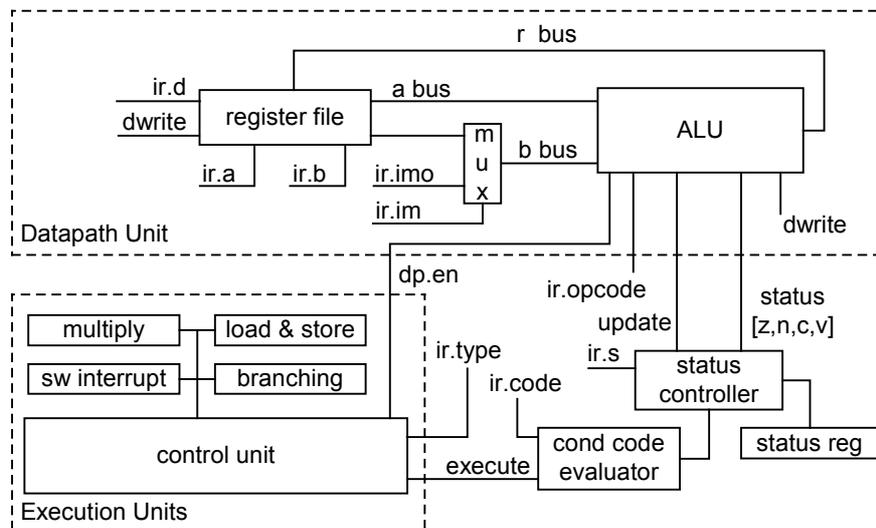*Table 1:  Signal names used in ARM Control Unit*



*Figure 3: Handel-C second-level ARM Control unit implementation*

```
    void adder(signal* ci, signal* a, signal* b, signal* s, signal* co, signal* v)
    {
     macro expr AN = width(*a) - 1;
     macro expr WR = width(*a) + 2;
     macro expr WN = WR - 1;
     macro expr CO = WN;

     signal WR aa, bb, r;

     par{
            // adder inputs
            aa = (unsigned 1)0 @ *a @ 1;
            bb = (unsigned 1)0 @ *b @ *ci;

            // adder outputs
            *s = r[WN-1:1];                     // result magnitude vector
            *co = r[CO];                        // carry out
            *v = r[CO] ^ r[WN-1] ^ (*a)[AN] ^ (*b)[AN];

            // adder
            r = aa + bb;
     }
    }

    // ADD, ADC, CMN: add, add with carry, and compare negated (updates CSPR only)
    // (Rd := Rn + Op2, Rd := Rn + Op2 + C)
    case ADD : case ADC : case CMN :
     par{
            ci = f[0] & ~f[1] ? *ici : 0;
            adder(&ci, &a, &b, &r, co, v);
     }
     break;
```

With the benefit of this flexible framework, exercises can be set to implement different aspects of processor operations, to enhance the instruction set, and to explore performance issues in depth. There is no necessity to implement the full instruction set, and by restricting to a suitable subset, key principles of computer operation can be highlighted. Conversely, there is nothing that prevents enhanced instructions (which may not be present in the actual processor's instruction set) being implemented to explore advanced architectural features. For example, the ARM emulation could be enhanced with Intel-like MMX extensions to explore SIMD concepts in media data processing for a final year specialist course. Finally, as Handel-C libraries are binary, it is possible to provide an executable model of a processor component which students are then required to re-implement, with the ability to check the performance of their own implementation against the reference library component.

## 5   Conclusions

Based upon initial material developed for an MSc/industry short course (see separate paper presentation at EEUG) we are currently developing extended examples which can be used as an adjunct to our current ARM-based first year undergraduate computer engineering programme. The top-down course structure outlined in this paper is particularly suitable where there is a need for an initial high-level course to be presented to a wide range of students (including those from a computer science background), whereas the second-level course is intended particularly for students who wish to achieve a detailed knowledge of computer hardware at the digital logic level rather than simply the register transfer level. (At Essex, the first-level course is taken by both Electronics and Computer Science students, whereas second and subsequent-level courses are presented solely to Electronics students.)