Occam on Field-Programmable Gate Arrays -
Fast Prototyping of Parallel Embedded Systems

This paper was presented at PDPTA’2000 (Las Vegas, June 2000) and was published in:

R.M.A. Peel & B.M. Cook

“Occam on Field Programmable Gate Arrays - Fast Prototyping of Parallel Embedded Systems” in
H.R. Arabnia, editor, “the Proceedings of the International Conference on Parallel and Distributed Pro-
cessing Techniques and Applications (PDPTA’2000)”

Occam on Field Programmable Gate Arrays -
Fast Prototyping of Parallel Embedded Systems

Roger M.A. Peel
School of Electronic Engineering,
Information Technology and Mathematics,
University of Surrey,
Guildford, Surrey GU2 7XH, UK

Barry M. Cook
Department of Computer Science,
Keele University,
Keele, Staffordshire ST5 5BG, UK

Abstract - This paper presents experiences obtained while using a compiler that translates designs written in a parallel programming language into a form suitable for controlling a Field Programmable Gate Array. Although the initial motivation for the project was to develop tools which would assist the implementation of a distributed multi-microprocessor architecture, results have shown that the technique is also well-suited to the fast prototyping of small, parallel, embedded systems.

Keywords: FPGA, parallel programming, occam, logic synthesis, embedded systems

1. Introduction

Users of occam and the INMOS transputer have been familiar with the benefits obtained when a design is divided into small parallel sections that synchronise by message passing and that use the PARBEGIN/PAREND mechanism for forking and merging the control flow.

This paper shows that similar benefits may be obtained in the design of parallel hardware. The advent of Field-Programmable Gate Arrays (FPGAs) allows a circuit to be constructed by compiling a design description and uploading the resulting binary configuration file into a universal logic device. Following an initial design which takes into account the usual rules of deadlock freedom, liveness, fairness and so on, the initial source code may be exercised using conventional software techniques. At this point, extra debugging code and tracing outputs may be used to confirm the operation of the algorithm; later on, such code may be too complicated (or slow) to co-exist with the hardware circuits. Only when the logic has been demonstrated to be correct should it be programmed into the target hardware.

The major benefits of these hardware compilation techniques may be seen in applications where communications and pipelined data flows are present. In most cases, however, communications may easily be introduced into designs that appear initially not to need them.

The prices paid for process parallelism and inter-process communication, in a software-only or a uniprocessor hardware environment, are the costs of scheduling the processes, a slightly greater memory usage and additional overheads as operands are copied from place to place. In FPGAs, however, the individual processes of a parallel program run genuinely in parallel, so there is no scheduling. In addition, communications may be overlapped with computation, memories may be placed anywhere in the circuit and run independently, and circuit optimisations may be used to eliminate many of the overheads seen in software implementations.

The parallel language that forms the basis of our experiments is occam. This is particularly suitable because it allows programmers to specify parallelism and communication in their designs - and allows us to extend them into hardware. Its basis in Hoare’s Communicating Sequential Processes paradigm (CSP) [1] ensures that it provides a reliable and easily-understood model of these features.

The next section of this paper discusses the compiler tools that have been developed, and the additions to the language and omissions in its implementation that have been made. The remainder of the paper outlines two applications that exemplify many of the benefits that the compilation of a high-level parallel language to low-level FPGA circuits makes available.
2. The Occam Compiler Tools

Our compiler is written in the Java programming language. It is capable of parsing two languages - primary source files must be provided in a subset of occam 2.1 (with a few occam 3 features added) and library modules must be specified in the Altera / Intel PLDSHELL logic specification language. The compiler stores representations of these formats internally, and then combines them to generate outputs in PLDSHELL, ABEL, IPLS, VHDL, EDIF and various dialects of occam. The occam output stream may be compiled by existing occam 2.0 compilers, since all of the casts that occam 2.1 derives from untyped literals are made explicit. The compiler also performs replicator unrolling and various other tasks that have been used experimentally during development of the output logic. The output that is compatible with existing compilers provides a verification route for our hardware design methodology similar to that provided by circuit simulation for conventional approaches using ABEL, VHDL and other hardware descriptions. External hardware modules, since they are interfaced to the remainder of the design along channels, may easily be emulated by occam processes during this software verification process. Our experience shows that properly-verified occam programs almost always execute correctly first time when they are loaded into hardware.

2.1. Language Features for Hardware Compilation

It is our intention, eventually, to be able to compile the whole of the occam family of languages - be that occam 2, occam 2.1, occam 3 or other appropriate enhancements. Some features of all these variants are less useful in a hardware context than others, so the order of implementation is not that which a software target might dictate. For instance, it is impracticable to incorporate floating point arithmetic on currently-affordable FPGAs, so none has been programmed yet. Likewise, arrays consume hardware resources at an alarming rate, with elements accessed sequentially using expensive combining logic, so they are also low on our priority list. On the other hand, SEQ, PAR, ALT, IF, WHILE, channels, PORTs, placement, basic integer arithmetic and so on are largely complete. Replicators for SEQ, PAR, ALT and IF are also provided.

One aspect of hardware compilation that varies radically from a conventional processor target is that every bit of every variable is precious when targeting FPGAs that are only capable of maintaining their circuit state in a few thousand flip-flops. Thus, every declaration in our language may specify exactly the correct integer field width necessary for the application. Integers need not store just unsigned values, nor conventional signed values centred around zero, but any range declared for that data type. Hence, the following are all valid type declarations:

- DATA TYPE BIT IS 0..1 :
- DATA TYPE UINT8 IS 0..255 :
- DATA TYPE INT4 IS -8..7 :
- DATA TYPE special IS 5..20 :

Arithmetic logic is generated for each instance of every operation. Thus, there is no single Arithmetic and Logic Unit (ALU) within the generated circuit. This ensures that there is no shared data bus that would restrict the parallelism of the circuit. Of course, it is possible to program such an ALU as part of a processor design should the programmer wish - but then the sequential nature of operand and result routing along the data bus becomes their responsibility. Another experimental feature of this compiler is its exceedingly careful tracking of the ranges of values that a particular variable may hold following expression evaluation. If the maximum or minimum value could ever breach the range of values that can be stored in the format specified, the compiler will reject the code. This forces the programmer explicitly to handle situations where a conventional occam compiler would generate code that halts upon detecting an overflow condition at run time. Many of these concepts were discussed in [2].

It became necessary at an early stage to extend occam to provide bit extraction operations. These enable individual bits, or ranges of bits, to be fed to separate groups of logic circuits without prior bit shifting and masking operations.

2.2. The Hardware Generation Strategy

The hardware designs created by our compiler follow many of the design approaches originally described by Ian Page in [3] and subsequently reported to WoTUG in [4] and [5].
Both Page’s Handel and our occam compilers generate one-hot logic [6] which passes a single ‘enable’ signal along a chain of flip-flops to indicate which part of the logic is active at any instant. Sequential processes are thus handled directly. Parallel processes fork by setting up several chains of active flip-flops at once, and these flows synchronise at the end of the PAR process by only passing an enable signal to the following logic when all of the converging threads deliver their enable signals. Special, more economical, versions of these synchronisation circuits have been developed to handle PAR constructs with just two or three threads. Beyond this number, a general-purpose circuit with a lower performance is required.

Input and output actions (interprocess communications) take place only when the control flows for both the sending and receiving processes become ready, thus providing the necessary synchronisation. Alternation is a rather more complicated instance of the same form of synchronisation. Since each end of a channel may be used by different processes, or different parts of the same process, in each communication, the logic for determining readiness and for selecting the correct words for transmission and reception can become messy. Simple alternation processes may again be optimised, making building blocks such as two-input multiplexers particularly efficient.

One circumstance in which Ian Page’s hardware generation strategy differs markedly from ours is in the predictability of timing. Ian states throughout his reports on Handel that each expression evaluation and assignment takes one clock cycle. This implies that all of the logic for each expression is compiled into one combinational tree, and thus that the clock speed for the whole design is determined by the slowest tree (and hence expression evaluation) throughout the whole program. This seems unnecessarily restrictive, especially in a highly parallel program, although the simple statement of the timing behaviour has its attractions. In contrast, every stage of our expression evaluation - even each stage of individual arithmetic operations - is clocked through flip-flops at the system clock rate. Therefore, each expression may be evaluated at its own pace, and the system clock may be run much faster than in Page’s scheme.

Just about our only timing assertion is that SKIPs and simple assignments currently take one clock cycle to execute, and that the former will not be optimised away altogether. It is thus possible to set up a process that counts time at the system clock rate, or provides output waveforms at a predictable rate or duty cycle. Even this guarantee will be abandoned once we implement TIMER channels and fine-granularity timing mechanisms; SKIPs may then be optimised away completely, and the elements of sequential processes may be executed in parallel provided that all of their dependencies remain consistent. All parallel processes proceed separately, synchronising when necessary. One special case is the parallel execution of predictable code where the execution time of each branch may be pre-computed (for example, parallel initialisations of variables to constant values or to existing variables); in this circumstance, there is no need to re-synchronise when the parallel events terminate, and the overhead of the PAR constructor is nil.

Another benefit of allowing all of our logic to run at the fastest clock speed possible is that it places fewer restrictions on external modules that may be connected to our occam-generated hardware. These modules may be designed manually and integrated into occam programs as separate parallel processes, after which they are accessed along occam channels in the usual manner. Some building blocks were considered too demanding to build directly in occam, and have been constructed in this more conventional manner. It is thus possible to link in modules that communicate using the 10 or 20 Mbps INMOS OS-Link and 100 Mbps IEEE 1355 link protocols. Subsequent experience shows that the careful use of double-buffering and lots of parallelism allows quite demanding timing constraints to be met directly from occam - see section 3.1 for an example. Provided that appropriate metastability precautions are taken at their channel interfaces, it is not even necessary to clock all of these modules, or indeed all of the parallel occam processes, at the same rates. On the other hand, our preliminary designs for occam channel circuits that join processes running at differing clock rates suggest that the performance loss will be substantial compared with the case when the processes share a common clock.
3. Example Designs

Occam appears to be an excellent vehicle for the design of embedded system logic - either the whole of a circuit or just the glue logic that joins other integrated circuit devices together.

A previous paper [7] presented occam implementations of a simple benchmark program - Peter Welch’s commstime - and elements of a future multiprocessor. This section gives some idea of the simplicity of designing components and circuits using the occam programming language through two further examples. Together, they show that the high-level language approach has real benefits.

3.1. A Character-based Video Display

Originally conceived as a mechanism for monitoring and debugging parallel programs running on FPGAs, this application has grown to provide an interesting testbed for many of the more demanding operations of the compilation and execution environment itself.

The general requirement is to be able to generate the video output for a character-based VDU screen using just part of the resources of a small FPGA and a minimum of external components. This has been achieved. Bits may be output continuously and without jitter from successive lines of an Occam program, thus building up the displayed image at rates of over 50 MHz, using the slowest Xilinx 4010XL devices. This is achieved whilst using a VGA connector buffered from the FPGA by a few resistors, and just one 32 kbyte static RAM memory that stores both the displayed characters and the font lookup table that is used to form them on the screen.

The overall circuit generates a 24×80-character display suitable for viewing on a standard VGA monitor. Each character is formed from 24 rows of 12 pixels each, using a font converted from the XFree86 X-Window System. The font occupies 16 kbytes of the SRAM, and 2 kbytes of the remainder is used for the display memory. A dot clock of 50 MHz thus yields a line rate of 41 kHz and a frame rate of 65 Hz. The memory is read at a peak rate of 12.6 MWords/sec during the display of those parts of lines containing characters.

![Figure 1](image1.png)

**Figure 1 - the processes of the VDU display**

Figure 1 shows the basic software design for the VDU display. It is primarily a pipeline, with the leftmost process generating line specifications, the next process converting line numbers into streams of pixels, and the final process outputting these pixels in a jitter-free manner. Since only one process may have write access to the address bus of the memory device, the writing of characters into the display memory must be performed by the line-generator process. Because time is tight when generating the bits for individual characters, the updating of the 2 kbyte display memory may only be performed during line and frame blanking intervals - which is more than sufficient.

```
INT8 buf1, buf2:
WHILE TRUE
SEQ
PAR
buf2.to.pixel ? buf1
SEQ
horiz, vert, pix.out := buf2[0], buf2[1], buf2[7]
pix.out := buf2[6]
pix.out := buf2[5]
pix.out := buf2[4]
pix.out := buf2[3]
pix.out := buf2[2]
PAR
buf2.to.pixel ? buf2
SEQ
horiz, vert, pix.out := buf1[0], buf1[1], buf1[7]
pix.out := buf1[6]
pix.out := buf1[5]
pix.out := buf1[4]
pix.out := buf1[3]
pix.out := buf1[2]
```

**Figure 2 - the pixel-generator process**

In order to achieve a jitter-free output from the pixel-generator process, several requirements must be met. This process acts as a shift register, reading in six-pixel words and serialising them onto its output.
WHILE TRUE
INT8 linecmd :
INT12 linebuf :
SEQ
-- receive line specification
frame.to.line ? linebuf
SYNC
-- propagate vsync bit
linecmd[1] := linebuf[0]
-- black pixels for hsync and line blanking
linecmd[2:7] := 0
-- (also updates display memory in
-- 3 SEQs below)
-- right blanking - no hsync
SEQ ii = 0 FOR 6
line.to.buf1 ! linecmd
-- horizontal sync
linecmd[0] := 1
SEQ ii = 0 FOR 24
line.to.buf1 ! linecmd
-- left blanking - no hsync
linecmd[0] := 0
SEQ ii = 0 FOR 12
line.to.buf1 ! linecmd
SEQ -- rows containing display pixels
IF
linebuf[1] = 1 -- blanked rows
-- (also updates display memory here)
SEQ ii = 0 FOR 160
linecmd[2:7] := 0
-- blank pixel o/p
line.to.buf1 ! linecmd
TRUE
SEQ ii = 0 FOR 80
SEQ
SYNC
-- column address
memaddr[0:6] := ii
-- row address
memaddr[12:14] := 0
SKIP -- wait for memory
SYNC
memaddr[0] := 1
-- line-in-row address
-- ASCII character value
memaddr[14] := 1
-- wait for memory
SYNC
memaddr[0] := 0
-- normal pixel o/p
line.to.buf1 ! linecmd
-- normal pixel o/p
line.to.buf1 ! linecmd

Figure 3 - the line-generator process

Clearly, pauses between successive groups of six pixels would be unacceptable, so double-buffering must be used, as shown in figure 2. Provided that the six-bit input words are supplied (along channel buf2.to.pixel) in a timely manner, the hardware circuits generated from this code achieve the required characteristics, since two-input PAR processes are merged without any overheads and the SEQs and the endless WHILE loop (as well as WHILE processes with many other forms of termination conditions) are again cost-free. Simple assignments are guaranteed to take just one clock cycle, thus completing the timing conditions.

The line generator process receives messages that specify a line number, in terms of a display row number (in the range 0 to 23), a line-within row number (also in the range 0 to 23) and two bits that indicate that a line is a vertical blanking line or that it is a vertical sync line. When each line number message is received, a row of pixels is generated in six-bit units (i.e. words). If the vertical sync bit is set, it is passed through to each such word. If the vertical blanking bit is set, six black bits are sent in each case. The appropriate words in each line are annotated with a horizontal sync bit. In each of these blanked cases, a character sent from the content-generator process may be written to the display memory.

When a non-blank row of pixels is indicated, a set of six-bit pixel words is sent to the pixel shift register. Eighty pairs of these words provide the dots that build up each displayed character on the line. For each character in the row, the display memory is interrogated to find its ASCII value. This value is then used twice when accessing the font look-up table, once to read the first six bits of the character’s line and once for the second six bits. For a twelve-pixel-wide font and just an eight-pixel-wide memory, two sequential read accesses are inevitable. The output from each of these reads is sent to the pixel-generator process together with the horizontal and vertical sync bits. Since each pair of words is sent in quick succession, two simple buffers are included in this channel to ensure that the pixel shift register has words available when it needs them. Obviously, the word generation rate must exceed the consumption rate, to avoid gaps in the final display. Figure 3 shows the line generator process with the display update code omitted (to save space).
WHILE TRUE
INT12 frame :
SEQ
-- 16 lines of bottom blanking - black o/p
frame[0:1] := 2
SEQ ii = 0 FOR 16
frame.to.line ! frame
-- 2 lines of vertical sync - black o/p
frame[0:1] := 3
SEQ ii = 0 FOR 2
frame.to.line ! frame
-- 32 lines of top blanking - black o/p
frame[0:1] := 2
SEQ ii = 0 FOR 32
frame.to.line ! frame
-- 576 lines of visible characters
-- (24 rows of 24 scan lines each)
SEQ ii = 0 FOR 24
SEQ jj = 0 FOR 24
SEQ
SYNC
-- normal pixel o/p
frame[0:1] := 0
-- character row address
frame[7:11] := ii
-- line-in-row address
frame.to.line ! frame

Figure 4 - the frame-generator process

Finally, the frame-generator process produces line specifications that are fed to the line-generator process. Unlike all of the other processes, this one is not time-critical. It simply cycles once per frame, issuing line numbers and indications of the vertical sync and blanking intervals. Again, the code for this process is very easily readable, as seen in figure 4.

One version of the content-generator process simply creates a continuous stream of update bytes; these are absorbed by the line-generator process during the blanking intervals at an average rate of 3.4 Mechars/sec. A second version relays updates and screen addresses from an OS-Link.

The overall code described in this section - 301 non-blank lines of occam - compiles to just 541 flip-flops. These may be fitted into 346 Configurable Logic Blocks (CLBs) on a Xilinx 4010XL, and even the slowest such device may be clocked at 52 MHz without any intervention from User Configuration (UCF) files whatsoever. Dot clock rates of 80 MHz and above may be achieved from small Xilinx Spartan devices. The OS-Link module requires another 52 flip-flops.

3.2. A Matchbox-sized IEEE 1355 Router

Another element of a multiprocessor that may be prototyped using transputer links is the interconnecting router. Large router designs are inevitably rather wasteful to implement on FPGAs, due to the buffering and the large number of gate inputs needed at each of the cross-points or terminal nodes (depending on the architecture).

However, we have produced a simple star-connected design that uses $n^2$ internal channels to route messages amongst $n$ links. This router is similar to the software router that could be used to pass messages within an array of transputers. The code is shown in figure 5.

This design fits in a Xilinx XCS40 device, requires minimal supporting circuitry, and thus occupies a board area little larger than the footprint of the FPGA and four tightly-packaged connectors. The throughput of this circuit is entirely limited by the speed of the Data-Strobe links that we have used to exercise it.

Figure 5 - An IEEE 1355 Router
4. Conclusions

Why is the use of a parallel programming language such a good way to specify a piece of hardware?

- Most embedded designs are easily segmented. Each of these parallel processes is small, self-contained and easy to reason about. Each may be run, surrounded by a suitable test-harness, using a conventional programming environment, with the usual operating system tools available for reviewing their outputs.

- Only when the design is fully tested, by direct software execution, might it become necessary to use hardware development tools (logic analyser, oscilloscope, etc.) to complete the implementation.

- Typically, only a few of the processes in a design have severe timing requirements. Even these processes can usually be considered in isolation, unless the timing of their inter-process communications should link them tightly together. Even then, the provision of buffering can decouple these processes to some considerable extent. Designs without modularity at the process level propagate their tight timing constraints throughout.

- Many occam language features may be translated into clocked logic with no run-time execution time penalty. This allows circuits to be built up from a number of small processes in a natural manner. These circuits can run fast - running logic at clock rates of 40-80 MHz is often possible even in slow FPGAs.

- The development process for the VDU project described in this paper was complicated by concurrent revisions of the occam compiler and by necessary refinements to the optimiser to produce the performance guarantees described. Now that these are complete, the whole cycle could be performed in just an hour or two.

- Methodologies for the design of deadlock-free occam programs are well-understood, and its parallel variable usage rules prevent non-deterministic updates to memory from processes that are running in parallel. Ask a VHDL designer about deadlock and race conditions ...

5. References


