Implementation of KRoC on Analog Devices’ "SHARC" DSP


Mechatronic Research Centre Twente and Control Laboratory,
Department of Electrical Engineering, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands.
Email: swz@rt.el.utwente.nl

Abstract: This paper summarises the experiences gained at the Control Laboratory of the University of Twente in porting the Kent Retargetable occam Compiler - KroC - to the Analog Devices’ ADSP21060 SHARC Digital Signal Processor.

The choice of porting the KRoC to the DSP processor was in our view both a challenge and an absolute necessity because DSP processors are an important ingredient in modern day control systems. Currently, our implementation contains the most important occam primitives such as channel communication, PAR, ALT, and most of the integer arithmetic. Furthermore, a basic kernel was realised, providing channel-communication based scheduling only. This porting process, using quite straightforward modifications of the SPARC KRoC-translator, was done within six weeks. A representative benchmark was constructed, showing that the 33Mhz SHARC-KRoC implementation is 40% faster than the the 25Mhz T800 using the INMOS D7205 Toolset.

1. Introduction

Prospective control system design will inevitably be influenced by present day micro-processor developments. These developments are in brief: availability of (Digital Signal) Processors with built-in memory and equipped with external communication channels and a general trend in the development of micro processors with combined DRAM+CPU together with on-chip integrated fibre optic link drivers. All this is supported by the new HIC [3] standard The architecture of future control systems will be complemented by a (fibre optic) network chip interconnecting a large variety of different processors i.e. I/O, DSP or number crunching processors. The question remains: how to program such a heterogeneous system? In our laboratory the use of CASE tools is encouraged [4] in order to create a parallel design that can be transformed into parallel processes most easily described as occam processes.

Although the system of the future should, in our view, be programmed in a parallel object oriented way; in present day terms we are quite satisfied with the utilisation of the CSP based occam language for such a system. This is the reason that we have taken up the task to implement occam on the 'SHARC'-processor, as manufactured by Analog Devices, using the Kent Retargetable
2. Kent Retargetable occam Compiler (KRoC)

KRoC is a software package that provides a way of compiling and running occam programs on micro processors other than the for occam programs usual transputer architecture. The KRoC software package contains a translator and a kernel.

The KRoC translator translates transputer assembly code generated by an occam compiler into an assembly file for the target microprocessor. The translated code is assembled and linked with the KRoC kernel. The KRoC kernel provides some of the transputer’s hardware functionality which is normally not available on regular microprocessors. KRoC is currently available for the SUN-SPARC and DEC-ALPHA microprocessor architectures, running the UNIX operating system.

The ADSP-2106x EZ-LAB evaluation kit [5][6][7] is used for testing and debugging of the KRoC implementation. This kit contains a SHARC-processor evaluation board that may be installed in a 16-bit ISA bus personal computer. The evaluation board is equipped with expansion connectors that allow for additional memory and/or SHARC processors. The software that is provided includes a C-compiler, assembler, linker, emulator (a debugger is not yet available, but the emulator allows some basic debugging), and various other development tools. There is an input/output library (C-lib, including scanf, printf) provided which make it easy to read and write to and from the host PC keyboard and screen.

3. ADSP21060 Processor architecture

The ADSP21060 is a Super Harvard Architecture Risc Processor (SHARC). This means that it has separate address and data buses for both program and data memory. The 4 Mbit on-chip dual-ported SRAM is divided in two blocks of 2Mbit: one for program code and one for data. The SHARC’s architecture has quite a few similarities with the transputer family such as its on-chip programmable timer functions with interrupt facilities and its support of six 4-bit link ports, and a DMA controller. The link ports can operate independently and simultaneously, with a maximum data throughput of 240 Mbytes per second. Link port data can be directly read by the core processor or DMA transferred to on-chip memory. Each port has its own double-buffered input and output registers. Clock/acknowledge handshaking controls link port transfers. Another interesting feature is that the unified address space of the SHARC allows direct interprocessor accesses of the internal memory of the SHARC. This could give interesting options for reducing the burden of process-placement among multiple SHARC-processors [9].
4. KRoC implementation on SHARC

The SHARC processor does not support the addressing of single bytes in memory. If a pointer to a memory location is incremented by one, the pointer points to the next (32-bits) word, and not to the next byte. The transputer has the ability to store an array of bytes (or Boolean) in single byte locations, but stores simple variables of these types in words. The word addressing behaviour of the SHARC processor caused some implementation problems. For this reason the array of Bytes has not been implemented. Suggestions are given however for a proposed solution. It is not clear whether this solution will hold under all circumstances though. It prevents for example, a simple implementation of the transputers SB and LB instructions (load and store byte).

```
INT i: -- integer (occupies 4 bytes)
BYTE b: -- byte (occupies 4 bytes)
```

These variable declarations occupy in occam 12 bytes of memory. The compiler translates this declaration in the assembly instruction AJW-3, which adjusts the transputers workspace pointer by 12 bytes. KRoC translates AJW-3 to SHARC assembly as:

```
m0=-3;
modify(i0,m0);
```

The register i0 (which is used as the transputers workspace pointer) is adjusted by -3. So the i0 registers points 3 words lower, and not the supposed 12 bytes. This is however not a problem as long we do not use the array of 4 bytes. Examine for example the occam statements:

```
i := 2
b := 'A'
```

In the transputers assembly language looks this like:

```
ldc 2 -- Load constant 2
stl 2 -- Store in location 2
ldc 65 -- Load constant 65 ('A')
stl 1 -- Store in location 1
```

The transputer multiplies the operands of these assembly instruction by 4 to obtain the real memory locations of the variables that are loaded and stored.

KRoC translates this into:

```
r0=2; -- Load constant 2
dm(2,i0)=r0; -- Store in location 2
r0=65; -- Load constant 65 ('A')
dm(1,i0)=r0; -- Store in location 1
```

Note that the operands are not multiplied by 4, because the smallest memory location of the SHARC processor is already 4 bytes.

Problems are coming up with a statement like:
\[ a[2] := 'B' \]

The OCCAM compiler generates:

\begin{verbatim}
  ldc  66 -- Load constant 66 ('B')
  ldlp 0 -- Load local pointer to begin of array at location 0
  adc  2 -- Add constant 2 to the start address of the array
  sb  -- Store byte
\end{verbatim}

A straightforward translation into SHARC assembly would look like:

\begin{verbatim}
  r0=66; -- Load constant 66 ('B')
  r1=i0; -- r1 points to begin of array at location 0
  r15=2; -- Add 2 to the start address of the array
  r1=r1+r15; -- to obtain target address
  r1=r1; -- Load calculated address into temporary pointer
  dm(0,i1)=r0; -- Store byte
\end{verbatim}

This translation is incorrect, because by the addition of two, a wrong target address is calculated. Actually, the integer i at memory location 2 is overwritten. To solve this memory addressing problem, it would be necessary to pack groups of 4 bytes into words. This is however difficult to implement, because KRoC has to determine from the transputer assembly source when an array of Bytes or Boolean is used. It may be necessary to scan for array addressing characteristic phrases of instructions in the transputer source. This is complicated because the generated assembly code is not always the same. A correct translation of the above sample code would be something like:

\begin{verbatim}
  r0=66; -- Load constant 66 ('B')
  r0=ashift r0 by 16; -- Shift to the location of byte 2
  r15=dm(0,i0); -- Read the word that contains the targeted byte
  r14=0xff00ffff; -- Load mask for accessing byte 2
  r15=r15 and r14; -- Reset the bits of byte 2 in the word
  r15=r15 or r0; -- Put byte in the word
  dm(0,i0)=r15; -- Restore the updated word
\end{verbatim}

This means that quite a bit of address calculation and byte-within-word locating has to be performed at every memory BYTE (or INT16) access. It is clear that this will inevitably result in performance loss, besides it is not clear if this solution holds under all circumstances, reason why it has not yet been implemented.

A few problems that had to be overcome in building a SHARC KRoC translator, are described in the following paragraphs.

### 4.1 the separate data and program memory

The SHARC processor has separate address and data buses for both program and data memory. This means that the 4 Mbit on-chip dual-ported SRAM is divided into two 2 Mbit blocks: one for program code, and one for data. Two data address generators (DAGs) provide memory addresses transferring data between memory and registers. Dual DAGs enable the processor output simultaneous addresses for two operand reads or writes. DAG1 supplies addresses to data
memory, DAG2 supplies addresses for program memory. Each DAG keeps track of up to eight address pointer registers, eight modify registers, eight length registers and eight base registers. A pointer used for indirect addressing can be modified by a value in a specified modify register, either before (pre-modify) or after (post-modify) the access. The length and base registers may be associated with each pointer to perform automatic modulo addressing for circular data buffers. The DAG registers are listed in the Table 1.

<table>
<thead>
<tr>
<th></th>
<th>DAG1 (Data)</th>
<th>DAG2 (Program)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer</td>
<td>i0..i7</td>
<td>i8..i15</td>
</tr>
<tr>
<td>Modify</td>
<td>m0..m7</td>
<td>m8..m15</td>
</tr>
<tr>
<td>Length</td>
<td>l0..l7</td>
<td>l8..l15</td>
</tr>
<tr>
<td>Base</td>
<td>b0..b7</td>
<td>b8..b15</td>
</tr>
</tbody>
</table>

Table 1. The pointer register of the SHARC

Because of the special character of the length and base pointer registers they are currently not used in the KRoC implementation. The KRoC implementation uses the program memory for program code and the data register for data, although it may happen that constants appear in the program area.

The mapping of the SHARC registers onto KRoC is illustrated in Table 2

<table>
<thead>
<tr>
<th>SHARC</th>
<th>Transputer</th>
<th>SHARC</th>
<th>Transputer</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>A register</td>
<td>m0</td>
<td>to change pointers</td>
</tr>
<tr>
<td>r1</td>
<td>B register</td>
<td>m5</td>
<td>zero offset pointer</td>
</tr>
<tr>
<td>r2</td>
<td>res. for SHARC run time library</td>
<td>m6</td>
<td>value one</td>
</tr>
<tr>
<td>r3</td>
<td>C register</td>
<td>m13</td>
<td>zero offset prog mem</td>
</tr>
<tr>
<td>r10</td>
<td>used by SHARC kernel</td>
<td>i0</td>
<td>Workspace pointer</td>
</tr>
<tr>
<td>r11</td>
<td>used by SHARC kernel</td>
<td>i1</td>
<td>complex transp. instr.</td>
</tr>
<tr>
<td>r12</td>
<td>used by SHARC kernel</td>
<td>i2</td>
<td>Proc queue front pointer</td>
</tr>
<tr>
<td>r13</td>
<td>Status register</td>
<td>i3</td>
<td>Proc queue back pointer</td>
</tr>
<tr>
<td>r14</td>
<td>MostNeg(0x80000000)</td>
<td>i9</td>
<td>Link-pointer</td>
</tr>
<tr>
<td>r15</td>
<td>used by SHARC kernel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Use of transputer registers in SHARC applications

The use of some of these registers is illustrated in the following examples.

```
ajw -3 (adjust workspace pointer met -3)
```

translates to SHARC as:

```
m0=-3;
modify(i0,m0);
```

Transputer instruction:

```
ldl 0
```

translates to SHARC as:

```
r0=dm(m5,i0);
```
Similar but for program memory:

\[ r_{15} = pm(m_{13}, i_{9}) \]

The SHARC processor further has 16 general-purpose data registers numbered from 0 to 15. These registers are prefixed with an 'f' when used in floating-point computations and with an 'r' in integer computations. The following assembly instructions for example, use the same registers:

\[
\begin{align*}
    f_0 &= f_1 \ast f_2; & \text{floating point multiply} \\
    r_0 &= r_1 \ast r_2; & \text{integer multiply}
\end{align*}
\]

Logic and arithmetic operations are only possible on the general purpose registers. Reading and writing to and from memory is possible with instructions like:

\[
\begin{align*}
    r_1 &= dm(i_0, m_0); & \text{read data memory} \\
    r_2 &= dm(m_7, i_5); & \text{read data memory}
\end{align*}
\]

This instruction loads register \( r_1 \) with the contents from the data memory location whose address is retrieved from register \( i_0 \). After this operation register \( i_0 \) is modified with the value in register \( m_0 \).

This instructions loads register \( r_2 \) with the contents from the data memory location whose address is retrieved from register \( i_5 \) modified with \( m_7 \). In this case the value of register \( i_5 \) is not modified. The following instructions perform similar operations on the program memory.

\[
\begin{align*}
    pm(i_8, m_9) &= r_3; & \text{write program memory} \\
    pm(m_{10}, i_{10}) &= r_3; & \text{write program memory}
\end{align*}
\]

Besides the DAG and general-purpose registers the SHARC processor has a number of special purpose registers for controlling processor mode, interrupts, timer and I/O. The DAG and general-purpose register files can be swapped with an alternate register file, by controlling some bits in the processor status register.

### 4.2 Label renaming

The SHARC assembler does not accept relative forward and backward labels that are used by the SPARC assembler. This type of label is used in the implementation of some complicated transputer instructions which require jump instructions. This is not really a problem, but it does implicate that the independent parts of KRoC had to be re-written. To demonstrate this the SPARC and the SHARC implementation of the STARTP instruction are listed below.

**SPARC implementation (with relative labels)**

\[
\begin{align*}
    \text{call} &\ 1f \\
    \text{add} &\ %11, %o7, %11 \\
    \text{l: inc} &\ 36, %11 \\
    \text{st} &\ %11, [%12-4] \\
    \text{cmp} &\ %14, %14 \\
    \text{bne, a} &\ 1f \\
    \text{st} &\ %12, [%15-8]
\end{align*}
\]
A function has been implemented that automatically generates a unique number every time a label is required.

4.3 Address calculations
Address calculations using label references, are not allowed in the SHARC assembler, therefore label references have to be assigned to registers before the final address can be calculated. The implemented solution is illustrated below.

The transputer assembler generates instructions like:

```
ldc L5 - L6
```

A straight forward SHARC translation is:

```
r0=LL5-LL6;       (The labels are renamed for SHARC syntax)
```

The SHARC assembler however refuses to accept this statement or similar statements that include address calculations. As a workaround the following solution is implemented:

```
r0=LL5;
r15=LL6;
r0=r0-r15;
```

Unfortunately this takes three instructions instead of one.

4.4 Assembly syntax of the SHARC processor
The SHARC assembler uses an unusual assembler syntax which looks more like statements from a higher programming language. The SPARC processor uses a more typical assembly language in the format:
Since the target independent part of KRoC relies highly on the usual assembler syntax mentioned above, it was necessary to rewrite parts of KRoC that were supposed to be target-processor independent.

### 4.5 Constant declaration

Constant declarations in an occam program are translated by the occam compiler in .BYTE assembler directives in the same address space (program memory) as the generated program code. Variable declarations are however declared on the transputers’ workspace, which is stored in the data memory block. Because the translated occam code makes no difference between accessing constants and variables; both of these data types, which resides in different memory blocks are accessed by the same DAG.

Although the SHARC processor allows this memory addressing strategy, this is not a recommend programming style, and may cause problems under certain circumstances. For example the SHARC processor emulation software that is included in the EZ-Lab evaluation kit does not support programs which uses the same DAG for addressing program memory and data memory. To solve this it may be necessary to move constant declarations to the data memory.

<table>
<thead>
<tr>
<th>Basic arithmetic operations:</th>
<th>ADC, ADD, DIFF, DIV, MUL, REM, SUB, SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical operations:</td>
<td>AND, OR, XOR</td>
</tr>
<tr>
<td>Address calculation:</td>
<td>BSUB, LDPI, WSUB</td>
</tr>
<tr>
<td>Load/Store instructions:</td>
<td>LDC, LDL, LDLP, LDNL, LDNLP, STL, STNL</td>
</tr>
<tr>
<td>General instructions:</td>
<td>AJW, CSUB0, DUP, EQC, MINT, REV</td>
</tr>
<tr>
<td>Program flow control:</td>
<td>CALL, CJ, J, LEND, RET</td>
</tr>
<tr>
<td>Alt instructions:</td>
<td>ALT, ALTEND, ALTWT, DISC, ENBC,</td>
</tr>
<tr>
<td>Process control:</td>
<td>STARTP, ENDP</td>
</tr>
<tr>
<td>Channel communication:</td>
<td>IN, OUT, OUTBYTE, OUTWORD</td>
</tr>
</tbody>
</table>

The error checking functionality of the transputer (set error flag on overflow), that is provided on some instructions, has not been implemented as yet.

| Table 3. List of transputer instructions that have been implemented so far |

### 4.6 Implemented instructions

Currently the most important occam primitives such as channel communication, PAR, ALT, as well as most of the integer arithmetic have been implemented and tested for the SHARC. Instructions that have not yet been implemented are: TIMER functions, floating point arithmetic and link communication. Also, some thoughts have to be given to aspects of multiprocessor applications, using link communication and placed PAR. A summary of the implementation is given in Table 3.
5. SHARC kernel

A SHARC kernel is realised that performs basic scheduling functions based on channel communication. Link-I/O is another kernel task, but hasn’t been implemented for the SHARC so far. Apart from link communication, one could also think of exploiting the shared memory facilities of the SHARC, as integral part of the kernel. The kernel of the original KRoC version for the SPARC architecture makes use of two UNIX processes. The child process executes the translated occam program, while the parent process waits for user input. This is to prevent deadlock when one of the occam processes is waiting for user input by calling an input function, because KRoC provides no time slice driven scheduling mechanism.

The two UNIX processes communicate with each other through pipes and signals. When the parent process reads a character from the terminal, it sends a signal to the child process and writes the character into a pipe. The child process responds on the signal by reading the character from the pipe, and puts it in the keyboard channel of the occam program. Because there is no UNIX like operating system on the SHARC processor, the UNIX specific parts were removed from the KRoC kernel. Because of this there is currently no keyboard channel available. It is however possible to read input by calling an input routine from the C-library like `getchar` or `scanf`. A disadvantage however is that a program will be suspended during the time it is waiting for input in the C-library routine, since it can not be descheduled at that point (no time-slicing).

The fact that no time slicing is available does give some general problems, for example when using the ALT-construct in a polling loop: if the ALT has a TRUE - SKIP entry, the process will never be de-scheduled in common cases such as in this sample process:

```occam
Boolean stop;
BYTE b;
CHAN OF BYTE c;
SEQ
  stop := FALSE
  WHILE stop=FALSE
    ALT
      c ? b
      STOP := TRUE
      TRUE & SKIP
  TRUE & SKIP
```

Whenever this process is active, it can never be descheduled because the TRUE & SKIP will always be favoured.

6. Performance

The SUN-SPARC implementation of KRoC was used as a basis for porting the SHARC version. In order to get an idea of performance, a comparative benchmark program was run on the
following processors.

- SHARC-KRoC on a 33 MHz ADSP21060 SHARC processor
- the original SPARC-KRoC on a 70 MHz micro-SPARC-II processor
- the original INMOS occam-D7205A compiler on a 20 MHz T4 and a T8
- Michael Poole’s PC-occam compiler on a 90 MHz Pentium.

The occam program below, a home-brew benchmark program for comparing the different implementations, was designed to give a measure of the extra overhead caused by running multiple parallel processes on a single processor. This program also demonstrates the current capabilities of the SHARC-KRoC implementation.

```occam
PROC bench.main (CHAN OF BYTE key, screen, error)

PROC calculate (CHAN OF INT out, VAL INT k)
    INT a, b:
    [100]INT c:
    SEQ
    SEQ i = 0 FOR 100
    SEQ
        a := 10000
        b := 5000
        c[i] := a * b
        c[i] := a / b
        c[i] := a + b
        c[i] := a - b
        out ! k

PROC collect ([25]CHAN OF INT in)
    SEQ i = 0 FOR 25
    INT x:
    ALT j = 0 FOR 25
    in[j] ? x
    SKIP
    :

SEQ i = 0 FOR 1000
[25]CHAN OF INT ch:
PAR
    PAR j = 0 FOR 25
    calculate(ch[j], j)
    collect(ch)
```
7. Results

The results of the benchmark program running on different processors is given in Table 4.

<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARC-KRoC on a 70 MHz micro-SPARC-II processor</td>
<td>5.0 seconds</td>
</tr>
<tr>
<td>Michael Poole’s PC-occam compiler on a 90 MHz Pentium.</td>
<td>7.7 seconds</td>
</tr>
<tr>
<td>SHARC-KRoC on a 33 MHz ADSP21060 SHARC processor</td>
<td>14.0 seconds</td>
</tr>
<tr>
<td>INMOS occam-D7205A compiler on a 25 MHz T8</td>
<td>24.3 seconds</td>
</tr>
<tr>
<td>INMOS occam-D7205A compiler on a 20 MHz T414</td>
<td>29.2 seconds</td>
</tr>
</tbody>
</table>

Table 4. Resulting execution times of the benchmark program

8. Conclusions

A SHARC implementation of KRoC has been partially realised in a reasonably short period of time. It proves that this port is an undertaking that can be performed by non compiler experts with good results. Although not all functions have been implemented, we got ourselves quite ahead on the learning curve of such a port that we feel confident that the more difficult tasks like the implementation of the timer and the external link communication is now within reach. The availability of occam on a large variety of processors will turn a new page in the realisation of control systems.

References


[3] IEEE Draft Std. P1355, *Standard for Heterogeneous InterConnect (HIC) for Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction*


[7] Analog Devices, *C Tools Manual*, provides information about the C-compiler, including calls from assembler from C and contains information about the run-time library among other things it explains which registers are used for run-time libraries.
