On writing a lcc back end for transputers

Michael Haardt
michael@cantor.informatik.rwth-aachen.de

Rütscherstraße 155/1703
52072 Aachen
Germany

ABSTRACT

This document explains how special parts of the back end for INMOS transputers work and gives reasons for certain design decisions. It further explains some differences between transputers and more common processors. It is assumed that the reader is familiar with the documentation included in the lcc distribution.

1. Introduction

1.1. The transputer architecture

Many modern processors have a few random accessably registers, of which a few are special purpose registers like stack pointer, frame pointer, instruction counter and segment base registers. The key word is ‘‘random accessably’’, which is where the transputer is different. Apart from extensions for fine granular parellel processing, which are not explained here because the developed lcc back end is intended for sequential programming, the transputer architecture looks as following:

FAReg	AReg	IPtr
FBReg	BReg	WPtr
FCReg	CReg	

AReg, BReg and CReg form an operand stack. Values can be put on that stack (pushing CReg out), AReg and BReg can be swapped, and values can be stored (moving BReg and CReg up, leaving the new contents of CReg undefined).

WPtr is the workspace pointer (stack pointer). Note that there is no frame pointer. That makes all local accesses WPtr relative, like on the Z80 and unlike the 8086. Usually this is no restriction, with one exception, which is explained at a later time.

IPtr is the instruction pointer (program counter). Not that there is no data segment pointer, because there is absolutely no MMU at all. This has serious impact on the usability of transputers for workstations, although most probably it is fine for embedded systems. If parallel programming is used, a hidden parameter for static areas has to be passed to all functions, for sequential programming static data is addressed IPtr relative.

This makes transputer workstations easy for attacks at sensitive data. Probably this is why none exist from commercial vendors. Crashing transputer machines is easy.

Having said this, the back end was nevertheless designed for generating sequential code, either for standalone applications or in a UNIX® like environment.

There is not much more to say about the architecture. There are no uncommon addressing modes available and there is no flag register. The T80x series has an additional FPU, which is in fact a seperate processing unit coupled to the main processor, although located on the same chip. (FAReg, FBReg and FCReg)

1.2. Impacts on the back end

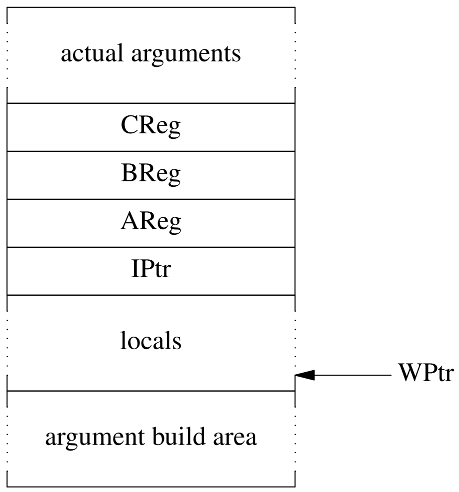
Transputers support byte and word data quantities in hardware, a word being 32 bit for 32 bit transputers. There are no instructions to deal with 16 bit quantities. The first versions of the back end defined short ints to be 32 bits, too, which is not a violation to ANSI C. It has shown though, that many code assumes that short ints are 16 bit wide. Of course that is unportable, but very common, so 16 bit accesses are simulated using a temporary variable and memory move instructions. If the value of the address is known at compile time, then a word access with value correction is used.

Together with the assumptions made in lcc, these are the properties of data types:

type	size	alignment	constants direct loadable
char	8 bit	8 bit	yes
short	16 bit	16 bit	yes
int	32 bit	32 bit	yes
long	32 bit	32 bit	yes
float	32 bit	32 bit	no
double	64 bit	32 bit	no
struct	-	32 bit	no

lcc assumes that pointers are unsigned in comparisons, because most processors indeed have their address space starting at 0. Transputers however start at the most negative address. The conclusion is that the value of a `NULL` pointer is `MinInt`. It would take some efforts to change the frontend to generate signed comparisons, changing the value of a `NULL` pointer is easy though. Therefore the back end is currently broken for 16 bit transputers with only 64k address space. On 32 bit transputers this restriction is meaningless.

The transputer can pass values to function calls in registers which are dumped to memory on executing the call. In case the values are not suitable for this very fast way to store them to the parameter area, they are written value by value, and a jump is used with changing WPtr using extra instructions. The resulting working space looks as following:



2. Generating code for expressions

In order to generate good code for transputers, the intermediate code trees need to be modified to use specialized instructions if possible. Further, making good use of the register stack instead of using temporary variables increases the efficiency of the generated code a lot. Using specialized instructions can change the number of needed registers for an expression and it is a very local operation on the tree, therefore it is done at the time of attributing the nodes.

3. Attributing the nodes

Choosing which back end specific attributes are needed showed to a hard part of development, often changes were made. The way lcc stores the intermediate code is well suited to transputers. For certain opcodes, the transputer needs the arguments to be evaluated different than they are generated, simply swapping the children of a node solves this problem very elegant. On register machines, loading constants to registers and holding them there is efficient, this is usually impossible on the transputer. Therefore, constants are always reloaded. Addresses are dealt with similar. This is archieved by simply setting an attribute which flags to visit a node no matter if it was visited earlier. For WPtr relative addresses, values can be load in one instruction instead of loading a pointer and using an indirect load. Using extended opcodes solved this problem together with manipulating the tree by removing obsolete nodes. The front end specifies to release memory for a tree using special memory allocation functions. This means that no care has to be taken with removing nodes from the tree. Other similar optimisations involve using instructions to add constants, compare with constants, using word index instructions instead of calculating the word index in software and using load-and-add or load-and-multiply instructions.

Atributes for unsigned remainder (and division) nodes and floating point operators are set. It should have become clear, that a node has to be attributed before its children are attributed, because attributing can change the children.

In contrary to the VAX code generator, no linearisation is done. This is because no assignment of node contents to registers is done. The generated code just computes values in a way that the register stack holds the operands in the right order for operators to be applied to them. Therefore, no `spill()` function is used, spilling registers is part of reordering the tree and a decision local to a subtree, not to a part of the linearised nodes.

3.1. Determining the register stack depth

Before decisions about modifying trees can be made, a function determining the cost in terms of needed registers has to be developed. Instead of using a notation similar to occam a notation more close to C is used (with float designating all floating point data types). The algorithm to determine the number of needed registers on the register stack taken from [Inmos88] is as following:

depth(constant)	=	1
depth(variable)	=	1
depth(call())	=	infinite
depth(e1 op e2)	=	if (depth(e1)>depth(e2)) return (depth(e1))
		if (depth(e1)<depth(e2)) return (depth(e2))
		return (depth(e1)+1)

This algorithm is easy to understand and looks elegant and complete at first glance. Unfortunately, it was found that for lcc a more detailed algorithm has to be used:

$$
\begin{aligned}
\mathrm{depth}(\mathrm{constant}_{\mathrm{float}}) &= 0\\
\mathrm{depth}(\mathrm{constant}) &= 1\\
\mathrm{depth}(\mathrm{variable}) &= 1\\
\mathrm{depth}(e1_{\mathrm{unsigned}} \,\%\, e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1_{\mathrm{unsigned}} \,/\, e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1_{\mathrm{unsigned}} > e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1_{\mathrm{unsigned}} \geq e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1_{\mathrm{unsigned}} < e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1_{\mathrm{unsigned}} \leq e2_{\mathrm{unsigned}}) &= 3\\
\mathrm{depth}(e1 = e2_{\mathrm{short}}) &= 3\\
\mathrm{depth}(*e_{\&\mathrm{short}}) &= 3
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{depth}(*e_{\&\text{aligned short}}) &= 2\\
\mathrm{depth}(\mathrm{call}()) &= 3\\
\mathrm{depth}((\mathrm{int})e_{char}) &= 1+\mathrm{depth}(e_{char})\\
\mathrm{depth}((\mathrm{int})e_{short}) &= 1+\mathrm{depth}(e_{short})\\
\mathrm{depth}((\mathrm{int})e_{float}) &= 1+\mathrm{depth}(e_{float})\\
\mathrm{depth}((\mathrm{float})e_{int}) &= 1+\mathrm{depth}(e_{int})\\
\mathrm{depth}(e1_{float}\ \mathrm{op}\ e2_{float}) &= \text{if } (\mathrm{depth}(e1_{float})<\mathrm{depth}(e2_{float}))\ \text{return } (\mathrm{depth}(e2_{float}))\\
&\quad\ \text{return } (\mathrm{depth}(e1_{float}))\\
\mathrm{depth}(e1\ \mathrm{op}\ e2) &= \text{if } (\mathrm{depth}(e1)>\mathrm{depth}(e2))\ \text{return } (\mathrm{depth}(e1))\\
&\quad\ \text{if } \mathrm{depth}(e1)<\mathrm{depth}(e2)\ \text{return } (\mathrm{depth}(e2))\\
&\quad\ \text{return } (\mathrm{depth}(e1)+1)
\end{aligned}
$$

The extension are casts and taking care of the different data types. It can only be suspected why the unsigned operators were omitted from the original algorithm. The depth function for the floating point stack from [Inmos88] has to be extended in a similar way to take care of nodes in the expression tree which only affect the integer register stack. For transputers without a FPU all floating point operations have integer depth 3, because they are emulated by calls to subroutines, which work on emulated floating point registers. This was done because it was easy to implement that way and nobody would use a transputer without fpu for more than casual fp operations.

3.2. Generating code

Generating code basically just means to determine the order of visiting nodes. [Inmos88] includes a short function which is a direct result of the `depth()` function. Having said this, it is obvious that this function is lacking support for unsigned remainder and division instructions, which itself is a direct result of the `depth()` function presented in the last section.

Code generation is done in various steps:

- initializing basic node attributes, optimising small sub trees to transputer specific node operation codes and determining the amount of needed temporary storage for each node.
- reordering the trees according to their depths.
- determining if arguments to functions can be passed using the register stack.
- assigning needed temporary storage to nodes. This has to be done as last step, because the node sequences are determined now.

Rather than visiting children in a different order depending on the execution order determined by the depths of the subtrees when emitting the code, `reordertree()` swaps children after determining the order. This allows to find out if a node is the first node which is code generated for, in which case it can assume to have the result of the last tree still on top of the register stack and storing/reloading is not neccessary. This looks like a minor optimisation, but all function call are roots of trees, so assigning the result of a function call would always be done using a temporary variable else. From looking at the concept of the intermediate code this is unexpected and should be documented more clear in the lcc documentation.

4. Emitting code

The source code to actually emit already generated code takes almost half of the back end. That may surprise, but there is a reason: The back end can generate either assembler source or object code. The first is used to see what is generated and when inline assembler is used, the latter is used else. Omitting the assembler pass results in a significant speed up.

Two ways to emit code required a clean interface, which in theory should make extensions to generate code for other assemblers easy. However, transputers differ from other processors here as well, because there are various ways to implement the features of the variable length of instructions to load values or to cope with the missing frame pointer.

Inline assembler is not implemented as straight forward as it seems it could on first glance. Loading pointers to local variables which may not be word aligned could be expressed either using a new mnemonic, but that would make parsing the assembler code neccessary, which is why another way was chosen. The code is written as if the variable would be word aligned, if it is not, an add-constant instruction will be silently inserted by the compiler right after loading the pointer.

The back end can emit code to check the working space pointer each time before it will get decreased. If the checking function finds that the decrease will cause the stack to grow into the heap, it will print the message "stack fault" and abort the program using the ANSI C function `abort()`.

Although not discussed in [Inmos88], loading 32 bit constants values can be speeded up by only falling back to the traditional way of using `ldc`, if the same result can not be obtained by combinations of `mint`, `ldnlp`, `adc` and `not`. This can result in substantial performance improvements on loading 32 bit constants with big absolute values. That may sound like a big win, but in real code most loaded constants have a very small absolute value, which was the reason for the transputer instruction set design. Not using `ldc` is an optimisation which can improve a rare worst case a lot. Since it was very easy to implement this, I decided to do so, despite the tiny win.

As previously mentioned, transputers have no frame pointer. While this is usually no problem, there is one situation where it is needed badly: `alloca()`. This function allocates memory in the stack frame, which has the two advantages of being very fast and avoiding memory leaks, because the memory is deallocated as soon as the function returns. Because of the need for a frame pointer, this function is neither part of ANSI C or POSIX. Much free software uses it though. For that reason, an emulation was implemented, which patches the return address of the function to a routine that frees the allocated memory, which is allocated in the regular heap. During the process, it showed that the lcc front end does not allow `asm()` to be used as transparently as one could wish, so passing the address of the stack begin to `alloca()` was not possible. Instead, the function which emits code was changed to pass this as additionally parameter to `_alloca()`.

5. ANSI C issues

ANSI C defines some aspects of C to be implementation dependent. Making assumptions about these aspects results in unportable code. However, many machines show similar behaviour, so an uncommon compiler or machine will give sudden surprises with what was believed to be portable and working C code. Here are some **wrong** assumptions, which were found while using lcc with the above back end. In all cases the code was working on at least one other architecture before:

- The evaluation order in expressions is either always left to right or always right to left.
- The result of shifting negative integer variables is the same as dividing by two.
- Overflow or underflow during expression evaluation will cause the values to wrap around their range and be silently ignored. In particular:
- Negating the most negative number has the most negative number as result.
- Enumeration variables have the same size as integer variables and therefore are able to store any integer value.

6. Discussion

What could be improved in the current compiler? One major problem is missing support for 16 bit transputers. That would require a change in the lcc front end though, because lcc makes the obviously unportable assumption that the address space is unsigned. Further some more advanced inline assembler support could help at times. Some aggressive optimisations on assignments of boolean expressions to variables could be done. They are rare, but the current code for that is quite inefficient. Unused workspace slots of a stack frame resulting from functions which have less than three arguments are not used for variables. There is no support for floating point/integer unit scheduling. The front end could make better use of temporary variables and should have a better cost evaluation for common subexpression elimination. No tail recursion elimination is done. Although structures of the size of a machine word could be passed in registers, this is not done. lcc does not use inline functions.

Most optimisations are very processor specific and don't help much with average code. In those rare cases, a compiler specifically designed for a transputer will generate dramatically better code. It is questionable if that is worth the efforts. Usually those rare cases can be optimised further in assembler. If a small amount of code dominates the overall performance, it will have to be optimised for a given combination of compiler and architecture anyway. Code written for architectures with many registers will not result in too satisfying run time results, whereas code where a register machine can not make good use of its registers will run very efficient on transputers. It is clearly visible that transputers were not designed for C.