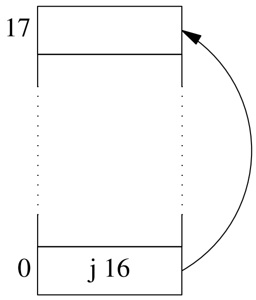
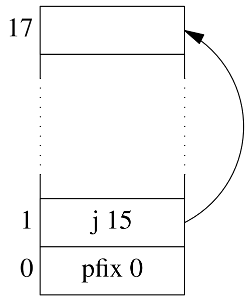
A new assembler for transputers Michael Haardt Rütscherstraße 155/1703 52072 Aachen Germany michael@cantor.informatik.rwth-aachen.de The special architecture of transputers and the requirements of fast compilers require to leave traditional implementation techniques. This document describes the design and implementation of an assembler and linker for transputers in the programming environment. The need for a new assembler There are a number of development tools for high level languages. Unfortunately, most of these tools are expensive and sources are not available. So I decided to start from the beginning, writing first a link device driver for Linux, My user space driver has, in the mean time, been replaced by a kernel space driver, written by Christoph Niemann. then porting INMOS iserver and 3L afserver and developing the programming tools: assembler, compiler and related utilities. From the beginning one goal was to use freely available software tools to cut down development time by not reinventing wheels, following the ‘‘Modular Software Tools’’ idea which made as powerful as it is today. Assemblers can usually be divided into two classes: Simple source to binary assemblers for writing small programs. Complicated macro assemblers which produce linkable object modules. The first class is unsuitable for serious use. The second class still often misses reasonably usable macro support for writing code by hand. A macro processor costs some CPU time even when it is not needed due to the scanning of the text for predefined macros. This, and the wish not to learn yet another macro language leaded to the first design decision: Usage of a seperated standard macro processor Apart from the rather simple C preprocessor, the only standard macro processor is m4, of which various free implementation are available. It uses inside-out macro expansion and offers enough features for serious use when assembler code has to be written manually. Among its features are recursive macros, expression evaluation, string functions, conditional expansion, file functions and much more. Profiling a typical compiler run shows that I/O needs significant time. [Thompson90] Omitting the assembler pass and generating object modules directly by the compiler saves much time, especially if the object format has a simple structure. However, for compiler development this approach can not be taken, so the assembler also has to be fast in generating object code. Traditional processors make it easy to determine all offsets in an object module in one pass, resolving references in a second pass to get the executable binary. This technique fails on transputers, because all operands are stored only with the nesseccary width to encode them, using a granularity of 4 bits. The result is usually short code but it leads to the uncommon design of a n-pass algorithm, which iterates over the object modules until all operand encoding lengths are determined. This leads to the second design decision: Using tokenised assembler language as object format. The object module format is a binary stream of tagged records. They contain fixed code from operatorless instructions, tags for (pseudo) mnemonics with symbol expressions in postfix notation as operands. The assembler The assembler has only the most neccessary features and a syntax which is efficient to translate into object code. One line of source code may be empty, contain a comment, a label or a (pseudo) mnemonic. Mnemonics have to be indented by at least one blank character, labels must not be indented. The assembler uses the mnemonics as defined in [Inmos88] with two extensions: ‘ldpi/lend’ can have an operand, which will be translated to ‘ldc relative-operand’ and ‘ldpi/lend’, this avoiding an explicit difference of two symbols. It makes assembler code easier to generate and to read. The other extension is to support instructions which INMOS didn’t document officially, but which are frequently used to implement fast context switches. Symbols which are local to the module are not declared, they are marked by a heading ‘@’ character. Symbols are typeless pairs of a symbol name and an associated signed number big enough to hold any address constant. No differences between label symbols, equation symbols or anything else are made. This avoids restrictions which are common with other assemblers. I believe that at this very low programming level, as less restrictions as possible should be made. An example is the POSIX requirement about a user not being able to disrupt a library routine that is dependent on another library routine. Compare the stub implementations of open: _open:

jmp __open with

equ _open,__open The second approach is better, because it causes no overhead at runtime. It is a natural consequence of having typeless symbols, whereas the first one is enforced by some traditional assemblers. The pseudo mnemonics allow alignment, binding expressions to symbols, defining data constants, specifying the processor type to warn about instructions not available on certain chips, and adding addresses to the relocation table. Expressions are usual arithmetical expressions, parsed using recursive descent functions. Constants can be numbers or strings in C notation, because the cost of these features are low and converting octal or hexadecimal numbers and strings to decimal numbers is very slow when done using a macro processor. Due to the chosen design, the assembler is a one pass assembler. It checks source code syntax and translates line by line into object format. Operatorless mnemonics and those with constant operators, like ‘adc 1’ are translated into fixed code with operators already converted to ‘nfix/pfix’ sequences. Apart from the ‘equ’ mnemonic which binds an expression to a symbol, parsing is reduced to scanning. Not declaring any symbols means that the error check for used but not defined symbols is moved to the linker. The advantage is to avoid a symbol table, which was considered to be worth the disadvantage of not having an error message telling the exact location in the source code. This simple structure makes the assembler very fast, so I/O becomes the dominant part in speed. This allowed a very comfortable and fast development of the code generator for a C compiler. The linker The linker first takes the specified object modules and links them with required modules from libraries to a loader input file. The first of three linkage passes determines which symbols are exported by a module (labels and equ symbols) and which symbols are imported (operands of mnemonics). This is not as straight forward as it could be, because it is needs to be allowed that object modules can export symbols of the same name as symbols in the library which leads to the concerned library module not being used. Although this is legal, the linker outputs a warning in such cases because errors resulting from doing so by accident are hard to find. The second pass uses an iterative algorithm to determine all needed modules, which are concatenated in a third pass. During concatenation, symbols local to a module will be encoded to be unique global symbols. The encoding is still visible to the loading pass, so they don’t appear in the symbol table. The libraries are simple ar archives of object modules, they don't need to be in a particular order; cycles are allowed. This makes utilities to build a total order and sort the library archive unneccessary. The loss in performance is not significant as other linkers show as well and building libraries gets very simple this way. Free implementations of ar are available. If symbols can not be resolved, the linker produces error messages telling which module needs a symbol which is not defined anywhere. The output is still a mixture of tokenized assembler instructions and operatorless code, using tagged records as file format. No symbol or relocation tables are built at all, which would make further processing before loading to a binary a very easy task. The linker output is a binary of all linked object modules. If absolute addressing is used, the resulting binary is not relocatible and has to be executed at the specified address, otherwise it can be load anywhere in memory with the address having the same byte selector as the load start address if 'align' pseudo instructions are used. Unlike other linkers, loading the input module to a binary is done using an iterative algorithm. In the first pass, it is assumed that all instructions needing an operand can be coded in one byte. The next passes check if the allocated length is sufficient and allocate more if needed. The algorithm terminates when no changes occured during one pass; it has to terminate because there is an upper bound on how much memory an operand can take. Finally, a last pass is done to actually create the binary, no output is done in the iteration steps. This algorithm will also produce the smallest binary with the given order of object modules, which is not neccessarily the case if starting on allocating maximum storage and then iterating to compress it. It is interesting, that with bigger binaries the increase in needed passes is very low. For example, for both a 27k and a 170k output file, 12 passes are needed, whereas files of far less than one 1k need 4 or 6 passes. [Inmos88] leaves out one case which may happen during this iteration on how to encode jumps and calls: Sometimes it is neccessary to code a jump with a leading 0 as operand, because without the 0 the distance can not be reached exactly:



Distance is 16, so use pfix to encode it.



Distance is now only 15, but pfix is needed.

Allowing that operands can take less space than in a previous iteration here can cause the algorithm to get in an infinite loop because the length toggles between increasing and decreasing. This got reported to happen to the assembler from [Gerlach91], which implements the code generation only based on the neccessary length to encode numbers. It is important that a pass has to find out how much space is needed for operands, but that it does not update the symbol table on the fly. The subsequent pass will perform that, otherwise the symbol table gets temporarily inconsistent. Placing computation of symbols to the linker means that expressions of symbols located in different modules are possible. That makes it very easy to create binaries usable to boot a transputer over a link, which are needed with the INMOS iserver. The disassembler During the testing phase of the assembler and linker, a disassembler was needed. First a non-symbolic disassembler was written to check correctness of code generation, later symbolic capabilities were added. To allow decoding **pfix** and **ldc** instruction to be decoded as one **ldc** instruction with the correct operand, a queue is used. Decoded instructions are queued and possibly reduced with previously queued ones to form more complex instructions. The disassembler has two modes of operation. In the first mode it creates a table of found jump, call and **ldpi** targets as well as unreachable code locations. It can use an already existing table as input to improve the heuristics used, letting it iterate over its own output may improve the results significantly. Using the unreachable code locations, it tries to seperate code from data. The more usual operation mode is to optionally load a symbol table file and output an assembler listing with symbols inserted where appropiate. Labels flush the output queue, so the following will be decoded right:

pfix 0xa label

ldc 0x1 Like the assembler, the disassembler has the extension to allow an operand to **lend** and **ldpi** which makes reading assembler code much more pleasant than expressions of symbols and loading the resulting constant before the loop/load instruction. Further, some undocumented instructions which are widely known are also disassembled correctly. Discussion Implementing the above approach has shown to be straight forward, and the resulting software has been used to successfully build lcc, which showed the expected performance win by omitting the assembler pass. However, there are still a few points where improvements and further research could be done: The fact that the number of needed iterations to calculate the prefix sequence lengths increases very little with bigger binaries is interesting. I believe that this happens because of locality, but not evidence of that has been shown yet. By calculating those prefix sequences at assembler time which are independent of inter-module relations and module position, that locality could be improved, which would speed up calculating prefix lengths because of less needed passes and less calculations per iteration. The costs are a more complicated assembler and a more complicated code generator in compilers. Optionally it could be done by an external program and depending on the needed CPU time, only be applied to library modules. Nothing is known about strategies (either mathematical or heuristical) and possible performance and size improvements of reordering object modules before calculating prefix lengths with the target of decreasing these lengths because of smaller distances for relative address related instructions. References Ken Thomp-

son, ‘‘A New C Compiler’’, p. 41–51, Proc. of the Summer 1990 UKUUG Conf., London, July 1990. A reprinted version is available from ftp server research.att.com via netlib ftp. INMOS Limited, ‘‘Transputer instruction set, a compiler writer’s guide’’, p. 8–9, Prentice Hall, 1988, ISBN 0-13-929100-8. Uwe Gerlach, ‘‘Das Transputerbuch’’, Markt&Technik, 1991, ISBN 3-87791-019-X.