|  |  |
| --- | --- |
|  |  |
|  | IMP Compiler Implementation Notes  Introduction  This page contains implementation notes that may be of interest to compiler buffs who have downloaded the source code to play with. If you're just using the compiler to compile and run your own IMP programs, you can probably skip this, and just read the [User Guide](https://web.archive.org/web/20050407092807/http:/imp.nb-info.co.uk/user.htm) page instead.  Overall Structure  The Intel IMP compiler is based on Peter Robertson's IMP-77 compiler, which uses three passes.   * The first pass parses the source code into an intermediate format, iCode. The first pass used in this project is based on the production compiler used for the Mouses operating system on the 32-bit Perkin Elmer (Interdata) machine. Only modest changes were made to the source code to increase portability - needed because the compiler was bootstrapped using a simplified SKIMP based compiler. All three Intel compilers (MS-DOS, Win-32 and UNIX) use the same source code for Pass 1 to simplify maintenance. Note however that Pass 1 is self-sizing to the native machine word size, so for correct results the 32-bit compiler must be compiled with itself - you can't mix 16-bit and 32-bit images. * The second pass reads the iCode and writes a proprietary intermediate object code file. The second pass is loosely based on the structure of a typical "Robertson" compiler but has significant detailed differences internally. It writes two files on output, one is a simple tagged binary object file, and the other is a diagnostic assembler source listing - a very useful tool for debugging compiler behaviour. The 16-bit and 32-bit versions of Pass 2 have over 95% commonality in source code. Both 32-bit compilers - for UNIX ELF and Windows COFF output - use exactly the same Pass 2. * The third pass fixes up the proprietary object file, which includes forward references (for jumps) and backward fix-ups (for local variable static allocations at procedure entry), and adds the trap table used by the run-time **signal** mechanism. The third pass also deals with the details of the target object file format. As a result, only some 50% of Pass3 is common between the different target environments.   The first two passes are written in IMP, and were bootstrapped using a simplified IMP compiler (SKIMP) written in C. The third pass is written in C since it needs to be portable between different targets, all of which have C compilers, but not IMP compilers.  Memory Architecture  For the 16-bit compiler, the "small" memory model is used - that is, there is a single code segment of up to 64 kilobytes, and a single data segment, also of up to 64 kilobytes, containing all the logical data segments and the stack. This has the important advantage that all pointers are 16 bits, the same as the native word size. For the 32-bit compiler the "flat" memory model is used - that is, a single 4 GB address space contains the code, data and stack areas. Again, all pointers are the same size as native words.  The compiler uses logical segments within the physical address space so that the linker can keep related items together, and so that (where available) appropriate memory protection can be applied.  The logical segments are:   * CODE - the executable code segment, generally execute only permission. * CONST - contains literal constants such as string literals, or floating -point constants, as well as fixed dope vectors. Generally, read-only permission. * DATA - contains initialised static data, mainly **own** variables, but also contains the data portion of **const** arrays. Read/Write permission. * SWITCH - contains switch tables. This is another read-only data segment, which is handled separately from the CONST segment to simplify relocation - there are no relocations in the CONST segment, whereas every word in the SWITCH segment is relocated by the offset of the corresponding CODE segment. * TRAP - contains procedure trap vectors for catching events. This segment is created by Pass 3 on the basis of procedure preamble and postamble information learned from Pass 2. The TRAP segment is read-only.   There is also some vestigial provision in the compiler for two further logical segments which are not used by the current implementation - BSS, for uninitialised static data and DISPLAY, which was used in the initial 8086 SKIMP compiler to hold the virtual display registers. Unititialised variables are actually initialised by the current Pass 2 to an unassigned pattern, and the display is held in the local stack frame.  The stack segment is allocated at run-time by the operating system. In UNIX systems, using the 32-bit flat memory model, the stack pointer is set to the top of the user address space and the stack segment is grown automatically by the operating system in response to page faults as the stack pointer moves down the store. Unfortunately, both DOS and Windows do not offer an automatic stack segment, and instead a fixed size must be requested at link time - a source of some frustration, since over-running the fixed size leads to a memory fault and abrupt termination.  Adding run-time stack probes so that violation of the requested size can be detected is on my to-do list. It will not stop the program failing, but might give a better error message!  Code Generator  The biggest difference between this Intel compiler and a classic Robertson compiler is that this compiler defers very little. That is, when it receives an iCode directive it goes ahead and plants code - in a Robertson compiler Pass 2 collects pending operators on the Pass 2 operand stack and thus defers planting code until it knows what is coming next. This makes the Intel compiler much simpler, both in stack data structure and in internal operation. The primary disadvantage in working this way is that a Robertson compiler generally knows the destination for a CPU operation before planting the code, whereas the Intel compiler can get caught out, typically by having an intermediate result in the "wrong" register.  Partly mitigating against this, the compiler has a simple look-ahead to pick up obvious improvements (such as **a = a + b** - most math instructions in the Intel architecture can act directly on a memory location) and in practice the code generated has surprisingly few wasted MOV instructions to fix an earlier poor decision.  Register Allocation  The Intel architecture has 8 registers - AX, BX, CX, DX, SI, DI, BP and SP. (In the 32-bit implementation, Intel call these EAX, EBX and so on to avoid ambiguity. In the compiler, there is no ambiguity, since the 32-bit compiler never uses the 16-bit form of the register set, and in the source code they are still referred to as AX, BX, etc. for commonality with the 16-bit compiler). Of these 8 registers, BP and SP have fixed roles (local name space base pointer, and stack pointer respectively) and are not available as general registers, leaving 6 general registers.  One of the key differences between the 16-bit and 32-bit architectures is in the combinations of registers that may be used in addressing memory. In the 16-bit architecture, only BX, SI and DI can be used as pointers to memory. The compiler therefore distinguishes between register allocations that will be used as pointers, and register allocations that will be used for math operations, and selects from the appropriate set accordingly. This incidentally leads to the classic "wrong register" problem identified earlier, where the array access **A(X+Y+Z)** will generally evaluate **X+Y+Z** in an accumulator (AX) and only then discover that it needs to move the result to a pointer register to complete the array access.  Although not strictly necessary, the 32 bit compiler maintains the same pointer register separation for compatibility, but unlike the 16 bit compiler it knows that it can use any register as a memory pointer - in the example above it will happily use the result in AX to index into the array.  Primitive support routines  IMP relies on support subroutines for many of the more complex operations, such as array access, string copy and concatenation, and certain math operations not supported by the CPU directly. In this implementation all the support routines are written in C for greater portability and maintainability. Although this means that all calls to primitive routines must use "proper" C-style stack based parameter passing rather than dedicated registers, performance seems more than adequate.  Catching events  One of the language features that took some thought was the event mechanism. When a procedure invokes a **signal**, the primitive called must walk the stack to see if any procedure (including this one) is interested in catching this event. I was very reluctant to add any overhead in the procedure entry and exit code to maintain a stack of event handlers, and therefore the run-time library needs to search for them when an event is signalled.  By tracing back the stack, the event mechanism finds a list of program counter values of all the calling locations. It needs to map these into procedures, and then determine whether each procedure has a handler for this event type, and if so, what the entry point to the handler is. The solution is to include in the run-time data area a table of structures containing:   |  |  | | --- | --- | | **Block Start** | First code address of this block | | **Block End** | Last code address of this block | | **Trap Entry Point** | Entry point to the trap handler (if any) | | **Protected Area Start** | A quirk of IMP - **on** **event** blocks only protect the code in the procedure block after the closing event **finish** | | **Event Mask** | Bitmask of events caught here | | **Abbreviated Procedure Name** | Used in the diagnostic backtrace |   A word about bootstrapping  There have been many successful IMP compilers written over the years, and when embarking on this project a port of an existing compiler seemed the obvious way forward. Unfortunately, there is a snag with this strategy - most IMP compilers were written in IMP. The original environment in which these compilers were developed was rich in IMP tools, making that a reasonable choice, but today none of these tools are readily accessible.  Nevertheless, the motivation to produce a "historically accurate" IMP compiler was strong, so some lateral thinking was required. The solution was to write a cheap and cheerful compiler for a subset of the IMP language which could be used to bootstrap a full IMP compiler. This simple compiler was based on the SKIMP compiler by D. J. Rees, but rewritten in C.  Why Skimp? I wanted something that would be quick to implement, and that would not require too much original thought. I still have the Skimp manual from my university days, which gives a pretty thorough description of how to construct a simple compiler, and more importantly I have listings of my own student work on Skimp. Of course, Skimp is written in IMP, and the whole point of the exercise is that I don't have an IMP compiler, so I have re-implemented Skimp in C - hence "C Skimp".  The next step in the strategy has been to take that SKIMP compiler (which produced 16-bit 8086 code) and use it to develop a code generator (pass 2) for an IMP-77 compiler. Because the IMP-77 compiler was being developed in the 16-bit SKIMP environment, it made sense to re-use the same tools and libraries and target the IMP-77 compiler at the same 16 bit environment.  The final stage was to take that 16-bit Intel compiler and retarget it to the 32-bit architecture. From a code generation point of view that was relatively simple; the challenge was in producing object files compatible with a quite different set of tools.  There were also a few tricks needed to hand-craft intermediate versions of the compiler that would run on a 16-bit machine but output 32 bit literals, needed for the actual bootstrap of up-compiling the compiler.  Current Status  The compiler is working in both 16-bit and 32-bit versions. The object file support for 16-bit OMF and 32-bit COFF is complete, and the ELF object file support is substantially complete.  There are still some areas that need to be tidied up. These include:   * Adding debug line number information to the ELF format - currently only MS-DOS and Win 32 targets support source code level debugging. * Adding more run-time checks, including byte assignment overflow, unassigned switch labels, and switch bound checks. * Fixing some bits that don't work yet, particularly external data import/export. * Improving the declaration of arrays with constant bounds (they are currently treated as dynamic arrays allocated at run-time, but the hooks are in there to allow them to be static, allocated at compile time). |