Lecture 4

CISC architecture provided a rich set of instructions and addressing modes but it made the job of the compiler harder when it came to generate efficient machine code. The RISC architecture simplified this problem.

Register Allocation

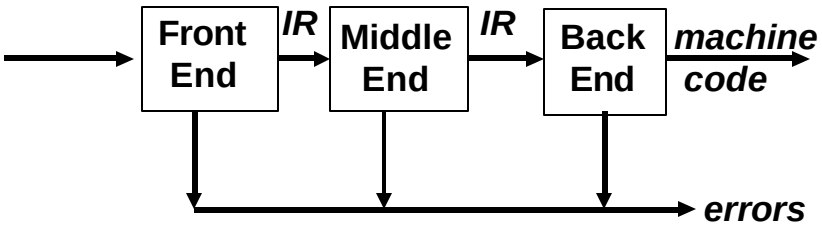
Registers in a CPU play an important role for providing high speed access to operands. Memory access is an order of magnitude slower than register access. The back end attempts to have each operand value in a register when it is used. However, the back end has to manage a limited set of resources when it comes to the register file. The number of registers is small and some registers are pre-allocated for specialize used, e.g., program counter, and thus are not available for use to the back end. Optimal register allocation is *NP-Complete.*

Instruction Scheduling

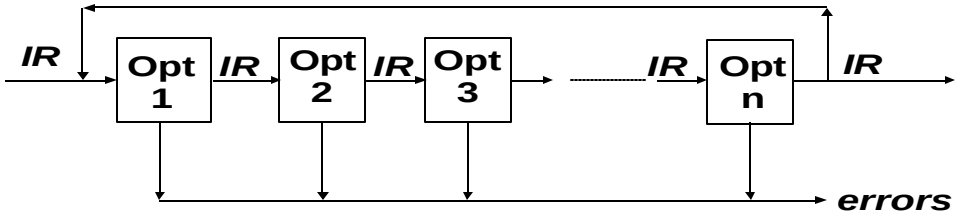
Modern processors have multiple functional units. The back end needs to schedule instructions to avoid hardware stalls and interlocks. The generated code should use all functional units productively. Optimal scheduling is *NP-Complete* in nearly all cases.

Three-pass Compiler

There is yet another opportunity to produce efficient translation: most modern compilers contain three stages. An intermediate stage is used for code improvement or optimization. The topology of a three-pass compiler is shown in the following figure:



The middle end analyzes IR and rewrites (or *transforms*) IR. Its primary goal is to reduce running time of the compiled code. This may also improve space usage, power consumption, etc. The middle end is generally termed the "Optimizer". Modern optimizers are structured as a series of passes:



Typical transformations performed by the optimizer are:

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Role of Run-time System

The executable code typically runs as a process in an Operating System Environment. The application will need a number of resources from the OS. For example, dynamic memory allocation and input output. The process may spawn more processes of threads. If the underline architecture has multiple processors, the application may want to use them. Processes communicate with other and may share resources. Compilers need to have an intimate knowledge of the runtime system to make effective use of the runtime environment and machine resources. The issues in this context are:

- Memory management
- Allocate and de-allocate memory
- Garbage collection
- Run-time type checking
- Error/exception processing
- Interface to OS – I/O
- Support for parallelism
- Parallel threads
- Communication and synchronization