The Back End

The role of the "back end" of the compiler is to translate architecture-neutral intermediate code (e.g. "quads") into pure assembly language which can be passed on to the assembler and linker to produce an executable program.

Prior to reaching the back end, the quads may have been subjected to architecture-independent optimizations, e.g. using data flow analysis to eliminate computations of constants, eliminating unreachable code, etc. This was covered in Unit #6.

The back end must generate assembly language which correctly implements the programmer's source intent, as represented by the quads, but moreover should produce "optimal" assembly language. There are three basic tasks:

1) Instruction selection: mapping IR operations to available assembly language opcodes and addressing modes

2) Instruction scheduling: re-ordering assembly language opcodes to improve performance, especially on architectures with multiple execution units.

3) Register assignment: deciding when values (variables, temporaries) should occupy a register, and which register.

Generally, "optimal" is understood to mean speed of execution, but other criteria, such as code size or cache performance, may also be weighed.

Unfortunately, these three steps are interdependent, making the back end the most difficult part of the compiler. E.g. assigning a value to a register as opposed to a memory location changes instruction selection, which in turn changes scheduling. At the same time, reordering the instructions may change register conflicts and require a reassignment.

The characteristics of processors continue to change as processor technology advances, so the algorithms which made sense ten years ago may no longer be suitable. Back end optimization continues to be an active area of compiler research and development. These problems are almost always NP.

A further requirement of the back end is that its selection of assembly language opcodes, addressing modes, instruction ordering and register allocation must be deterministic and consistent. From a software engineering standpoint, a given version of a given compiler, compiling a given piece of source code, must create the same assembly language output each time. This is known as "reproducible build," and if the compiler uses non-deterministic algorithms which don't satisfy this, it makes management of the software build process very cumbersome.

A compiler can use a table-driven approach where it has generic algorithms for the back end, with tables specific to each target. This is the approach that gcc takes. Alternatively, a compiler might be written with custom-coded back ends for each target.

In this unit, we'll only be able to give a light overview of what is a very complicated topic. Note that some of the "back end" material, in particular how variables are declared and accessed, was already covered in the previous unit.

Instruction Selection

It is not necessarily the case that the IR is strictly more powerful or strictly less powerful than the target assembly language. Invariably, some IR operations will not be expressible in a single assembly instruction, and at the same time, some sequences of IR instructions may be coalesced into a single instruction.

There are two basic approaches to instruction selection: linear and tree-based. In the latter, the intermediate representation is placed into a tree form where the nodes represent operations and the edges represent the flow of values. The available assembly language instructions are also represented as miniature trees or "tiles" which illustrate how that particular instruction transforms values. Each of these tiles may have an associated cost metric. The process of instruction selection is then the ‘‘tree tiling’’ problem: paste the tiles over the original tree in a manner which preserves its form, using the "optimal" tiling which reduces the total cost metric.

In the linear method, the intermediate representation is a linear sequence of instructions (quads) and the assembly instructions are linear patterns. Various methods can be used to rewrite the IR into assembly through pattern matching. E.g. the "Dragon" compiler textbook discusses a theoretical way to use a parser, where the instruction patterns are akin to grammar rules, and the IR is analogous to token sequences.

Tree-based vs linear methods of instruction selection, as may be expected, each have their advantages and drawbacks. Linear methods may have difficulty observing data flow, especially if quads have been reordered during optimization, and thus are more reliant on the quad generator using certain strategies. Tree methods often have difficulty with control flow changes.

The simplest linear method is template-based. Each quad is translated directly into one or more assembly instructions, without regard to the other quads surrounding it. This method performs poorly when the assembly is richer than the quad schema, e.g. X86 assembly, because it can not recognize opportunities for combining instructions into a single more potent instruction or addressing mode.

We will discuss a template-driven method of linear instruction selection called the "peephole" method. We define a "window" which has a certain maximum size. This window spans a constantly advancing linear range of instructions (instructions might be a mix of quads, assembly or both). The algorithm can be summarized as:

```
while (there are more quads to consume)
{
        do {
                Examine the current window for pattern matches
                (If there are multiple matches, pick the least-cost)
                Perform the pattern transformation
        } while (there was a pattern which matched)
        shift more quads into the far end of the window:
                Keep window filled with at least longest-sized pattern
                Since the window has a definite size, eventually
                something will have to be pushed out of the near end.
                If this something is an un-matched quad, we have
                failed to produce an instruction selection, oops.
}
whatever is left in the window should be assembly language
```

Translated assembly language instructions roll out of the top, or near end, of the peephole window. The window size is chosen so that the longest practical sequence of quads/assembly can be handled. Making the window unduly large has no further advantage, and instead increases the translation time by requiring more searching. The peephole method can be applied individually to each basic block, or to the overall linear stream of quads. The patterns to be matched can also rewrite assembly language opcodes into more efficient opcodes.

The "shape" of the quads, i.e. the strategy used by the quad generator and the quad scheme, influences this method. If the quads are very rich, it may be necessary to expand them into simpler operations before entering the window. In order to find optimal instruction sequences on some architectures, it may be necessary to expose hidden data flow, e.g. the use of condition codes.

It should be understood that the instruction selector is not necessarily expected to perform well on arbitrary input sequences of quads. It may depend on certain properties of the quad generator, e.g. that all temporary names are assigned to just once and never re-used.

In the examples below, we will use an informal syntax for describing the pattern matching/replacement engine. It isn’t intended to be a literal implementation. In some compilers, the pattern matching is coded on an ad-hoc basis. Others, such as the GNU family of compilers, define a complicated language for modelling the target system and applying transformations. When the compiler is built for a particular target, that language is used to generate internal tables and code to perform the pattern matching.

Instruction Selection example, X86

Let's say we have the following code fragment:

```
f()
{
int i;
int ary[16];
         ary[i]++;
```

This might produce the following IR sequence, using the nomenclature previously described in Unit 5. We will make the observation that temporary names are like virtual registers. They have no assigned storage location, no explicit initialization, and no way to take their address. On the other hand, program variables have an existence in memory, and have to be loaded (either explicitly into a register, or implicitly via an addressing mode) before use. The quad code below follows the convention that all temporary names are represented by virtual registers. We assume that it has already been passed through the architecture-independent optimizer which removed the redundant address computation for a[i]:

```
         %T1=     LEA      ary{lvar}
         %T2=     MUL      i{lvar},4
         %T3=     ADD      %T1,%T2
         %T4=     LOAD     [%T3]
         %T5=     ADD      %T4,1
         STORE    %T5,[%T3]
```

The pattern recognizer could be set up to recognize something like this:

```
Recognize:
         %rega=   LEA      localvar(x)
         %regb=   MUL      localvar(y),4
         %regc=   ADD      %rega,%regb
Replace:
         movl     offset(y)(%ebp),%regb
         leal     offset(x)(%ebp,%regb,4),%regc
```

In these patterns, %rega is a wildcard, matching any virtual register, but tracking it as well, so that we can see the correspondence between the LEA into a register and the use of that register in the ADD. Likewise localvar(x) is intended as a wildcard matching any local variable value and giving it a wildcard name x. offset(x) will be replaced with the offset (within the stack frame) of the local variable which has been given the wildcard name x. This offset is clearly known to us at this time, since the entire function has been seen and the stack frame has been laid out with offsets assigned to all local variables. E.g. let's assume that i has an offset of -4 from the %ebp register, and ary has an offset of -8. The lowercase nomenclature indicates a translated assembly language instruction, vs uppercase for quads.

Considering only the pattern above, the first two quads would be shifted into the peephole but result in no matches. Then the third quad would create a match against the pattern. But in general, there will be multiple pattern matches. E.g. there should be a pattern:

```
Recognize:
         %rega=    LEA       localvar(x)
Replace
         leal      offset(x)(%ebp),%rega
```

to handle the more general case where the address of the array is needed elsewhere in an expression. This pattern will fire too early unless we modify the peephole algorithm slightly. Instead of inserting quads one at a time, we always grab groups of quads which are larger than the longest pattern. When multiple patterns match, we assign a metric or "weight" to each match which favors the longest possible match.

Another pattern to pick up the load, add, store sequence and select the incl instruction might be:

```
Recognize:
         %regb=    LOAD      [%rega]
         %regc=    ADD       %regb,1
                   STORE     %regc,[%rega]
Replace:
         incl      (%rega)
```

At this point, the following assembly instructions are in the window:

```
         movl      -4(%ebp),%T2
         leal      -8(%ebp,%T2,4),%T1
         incl      (%T1)
```

Finally, another pattern might be written which matches not quads, but partially constructed assembly code. It recognizes the addressing mode:

```
Recognize
         leal      offset(%rega,%regb,scale),%regc
         opX       (%regc)
         {%regc dead}
Replace
         opX       offset(%rega,%regb,scale)
```

Leaving us with:

```
         movl      -4(%ebp),%T2
         incl      -8(%ebp,%T2,4)
```

Thus in a CISC target which is richer than our IR schema, the number of operations has been consolidated during target code generation.

During register allocation, the virtual registers %T0, etc. would be replaced with actual registers. Note that the last pattern was predicated on "%regc dead". We defined dead and live values in Unit 6. Consider that if %regc were live after this 2-instruction pattern, it would not be proper to replace it with the single instruction, because later on the IR is depending on a valid value (the address of the array element) being in %regc. If we control the "shape" of the quads so that we would never generate a %Txx temporary that gets re-used, and the optimizer does not wind up causing it to be potentially re-used via common subexpression elimination, then we could eliminate this "live/dead" predicate.

Difficulties with 2-address targets

Most instructions in the X86 set are 2-address (there are also 1-address and 3-address instructions), meaning that the destination operand is also one of the source operands. This creates challenges in translating 3-address quads. In addition, for most operations, it is not allowed to specify both source and destination as memory operands. E.g.

```
 extern int a,b,c;
          a=b+c:

          a{global}=ADD       b{global},c{global}
```

One possible pattern sequence might be:

```
Match
          global(Z)=ADD       global(X),global(Y)
Replace
          movl      X,%regN
          add       Y,%regN
          movl      %regN,Z
```

Here %regN denotes a new virtual register which is allocated on the fly.

If the instruction selection algorithm has information about "live" values available to it (see below under register allocation) then it can make more informed decisions about 2-address issues.

```
Match
          %tempC=   ADD       %tempA,%tempB
          {%tempB live}
```

If %tempB is live after this quad, we must emit:

```
Replace
          movl      %tempB,%tempC
          addl      %tempA,%tempC
```

If we know that %tempB is dead after this quad, we can emit

```
          addl      %tempA,%tempB
```

and then alias %tempC to %tempB thereafter, so that any reference to %tempC as a source operand is satisfied by %tempB.

More trouble with X86 instructions

Some X86 instructions require the use of specific registers. Now let’s look at

```
 extern int a,b,c;
```

```
        a=b/c;

        c=DIV     a,b
```

One might think that we can select instructions for DIV in a manner analogous to ADD. But the X86 idiv instruction (idiv is for signed long division, div is for unsigned long) has a peculiar restriction. It is really a 1-address instruction. The dividend is actually a 64 bit value and is always in the register pair %edx:%eax (%edx is the most significant 32 bits). The divisor may be a register or memory operand (but not immediate!). The quotient is placed in %eax, and the remainder in %edx (there is no separate MOD instruction).

```
Match:
        global(Z)=        DIV       global(X),global(Y)
Replace:
        {kill %eax}
        movl      X,%eax
        {kill %edx}
        cltd
        idivl     Y
        movl      %eax,Z
```

The cltd instruction sign-extends the %eax register into the %edx register, and is necessary to set up properly for a signed integer 32 bit division. (If this were unsigned, we could clear edx with xorl %edx,%edx, and we would use the divl opcode instead.) An unfortunate consequence of this register-specific peculiarity is that we must violate our nice clean model of keeping all registers virtual during instruction selection. During register allocation, when the movl X,%eax is encountered, it will force whatever value happens to be in the %eax register at the time to be spilled to make the register available. Likewise, the {kill %edx} notation will instruct the register allocator to vacate the %edx register prior to the cltd instruction (which will implicitly overwrite it).

Instruction Selection example, SPARC

In the CISC architecture example above, the assembly output often contained fewer operations than the IR form. This is because, in general, the CISC instruction set is more expressive than our choice of quad schema (we've seen some annoying exceptions, however).

The RISC philosophy is much closer to that chosen for our quads, in that it is inherently 3-address with a limited number of addressing modes. Therefore we would expect that the assembly output would be similar to our quads, if not an expansion, and there would be little opportunity to coalesce sequences of quads into smaller sequences of assembly. Thus in many ways our job of instruction selection will be easier. Recall our previous example with quads:

```
        %T1=      LEA       ary
        %T2=      MUL       i,4
```

```
        %T3=    ADD     %T1,%T2
        %T4=    LOAD    [%T3]
        %T5=    ADD     %T4,1
                STORE   %T5,[%T3]
```

In the X86 instruction set, we could access memory operands directly, but in SPARC only register-to-register operations are permitted. We will assume that during architecture-neutral optimization, any opportunities for placing variables into virtual registers safely have been identified and that code has been re-written to use the virtual registers. Therefore, any references remaining to memory operands are required to be translated as actual references to those memory operands. To use the pattern-matching approach, we would need to build a pattern for each of the three operand positions:

```
Recognize
        Z=OP(O)   localvar(x),Y
Replace
        ld      [%fp+offset(x)],%regN
        Z=      OP(O)   %regN,Y
Recognize
        Z=      OP(O)   Y,localvar(x)
Replace
        ld      [%fp+offset(x)],%regN
        Z=      OP(O)   Y,%regN
Recognize
        localvar(z)=    OP(O)   X,Y
Replace
        OP(O)   X,Y,%regN
        st      %regN,[%fp+offset(z)]
Recognize
        %rega=  LEA     localvar(x)
Replace
        add     %fp,offset(x),%rega
Recognize
        %regc=  ADD     %rega,%regb
Replace
        add     %rega,%regb,%regc
Recognize
        %regb=  ADD     %rega,imm(I)
                        {Caveat: -4096<=I<4096)
Replace
        add     %rega,I,%regb
```

Here OP(O) represents any wildcard quad operation, X/Y/Z are wildcard operands which could be %T temporaries, immediate values, or local or global variables), and %regN is a new virtual register which is assigned on the fly. Applying these patterns and assuming that the offset of variable i is -84 and the offset of ary is -80:

```
        add     %fp,-80,%T1
        ld      [%fp-84],%T6
        mul     %T6,4,%T2
        add     %T1,%T2,%T3
        ld      [%T3],%T4
        add     %T4,1,%T5
```

```
        st      %T5,[%T3]
```

As discussed in Unit 7, SPARC instructions are always a single 32-bit word, and there is no room for immediate values other than small ones. 13 bits are available for an immediate value, and since the value is sign-extended to 32 bits, this limits the range of I to -4096 through +4095. To handle larger constants:

```
Match
        %regb=   OP(O)     %rega,immI
Replace
        sethi    %hi(I),%regN
        or       %regN,%lo(I),%regN
        OP(O)    %rega,%regN,%regb
```

A similar pattern needs to be in place for accessing memory operands which are global, as there is no way to specify the 32-bit address directly.

Instruction Scheduling

The objective of instruction scheduling is to change the order of instructions, where possible, to improve the overlap of operations and minimize the total time. In order to do this, the compiler constructs an internal model of the specific processor's pipeline properties, such as the number of execution units available, the execution time of the various instructions, etc. It then adjusts the timing of instruction issue to minimize the delay of an instruction which stalls because it is waiting for a result from another instruction.

We will not have time during this course to discuss instruction scheduling. The reader is referred to the Dragon textbook for an introductory presentation.

Register Allocation

The instruction selection phase can rewrite the generic IR into one in which there is a clear distinction between memory symbols and registers, because at this point it knows the target architecture and the means used to access parameters, local variables, etc. However, it is not yet practical to allocate specific registers. As we have seen, we can use virtual registers such as %reg0 and complete instruction selection.

Then, the job falls to the register allocator to make the best use of the real registers, mapping the virtual registers to real registers, and inserting additional instructions as needed. The set of physical registers is usually limited, which makes this an interesting problem.

In order to do its job, the register allocator must be able to track the "liveness" of virtual registers, i.e. could the value in that virtual register be used at a subsequent reachable point in the code prior to the virtual register being overwritten. In Unit 6, we saw algorithms for analyzing this information.

Local Register Allocation

A local register allocator works only within a single basic block, and works on an operation-by-operation basis. It maintains a "scoreboard" representing the pool of general-purpose registers. For each register, we track if the register is currently available or allocated to a specific name (virtual register). The algorithm can be summarized:

```
        for(i=1;i<=N;i++)            //opcodes numbered 1..N
        {
                consider opcode i,
                if src1 is a virtual register:
                        rx=ensure(src1)
                        rewrite src1 to use %rx
                if src2 is a virtual register:
                        ry=ensure(src2)
                        rewrite src2 to use %ry
                if (src1 is dead after operation i)
                        mark rx free
                if (src2 is dead after operation i)
                        mark ry free
                if dst is a virtual register:
                        rd=allocate(dst)
                        rewrite dst as %rd
                        mark %rd as dirty
        }
        spill all dirty registers live at end of this BB

  ensure(vr)
  {
                if a register is already allocated to vr, return that register
                r=allocate(vr)
                emit code to move vr from its backing store to r
                return r
  }

  allocate(vr)
  {
                pick an available free physical register r
                if there is none:
                        pick the "best" physical register r to spill
                        it currently contains virtual register X
                        if (r is dirty)
                                emit code to store %r to backing store for X
                mark register r as containing vr
                return r
   }
```

```
**************************************************************************
The "Scoreboard", example for X86-32
Phys Reg  Contains  C/D      Flags/properties
----------------------------------------------------------------
```

```
eax      %T1                  Clean    group_scratch,func_rv
ebx      -free-               ---      group_longterm
ecx      i{lvar}              Dirty    group_scratch
edx      z{global}            Clean    group_scratch
edi      -free-               ---      group_longterm
esi      %T2                  Dirty    group_longterm
***********************************************************************
```

Register Spills

Consider each operation in the basic block. If the number of virtual registers which are live at that point exceeds the number of physical registers in the pool, then the allocator will be forced to **spill** a register. The allocator will only spill live virtual registers. If a virtual register is dead after a particular operation, then it would be marked as free. In order to spill a virtual register which is live, its contents must be saved somewhere. Later on, when the value is needed again as a source operand, it will be fetched back into a physical register. The backing location will either be an absolute memory address for the case of global variables, or a place on the local stack frame for local and temporary variables. The register allocator is architecture-specific so it knows how to load and store these values.

It would be unusual for a temporary variable to spill. Most temps are short-lived. If the target architecture has a very limited supply of registers and is evaluating a complex expression, it is possible that this will happen. Since temporaries don't have a defined place in memory, we will need to create phantom local variable slots to hold them. We'll need to tally up the highest number of these spill slots that are in use at a given time, and reserve that many slots in our function prologue as if they were declared local variables.

Often the register allocator must deal with multiple groups of registers which serve different purposes and must be dealt with as separate allocation pools. E.g. floating point vs integer registers. The register allocator can also use liveness, or whether a particular virtual register is a temporary vs a source program name, as hints towards steering the allocation towards a particular group of registers. E.g. on the X86 architecture, short-lived values should gravitate towards the register set (%eax, %ecx, %edx) because these are caller-save registers and are expected to be lost across function calls. Registers in the (%ebx, %edi, %esi) set should be used for longer-term values as they do not need to be saved and restored after each function call. Likewise on the SPARC architecture we should use %g1..%g7 for short-term values, %l0..%l7 for long-term values, and %o0..%o7 for values which are to be used as arguments to a function call.

As a further improvement, the register allocator can track if a particular physical register is "dirty". If a value has been written into that register and then the live value in that register needs to be spilled, a write-back to backing store is needed. But if the value is clean, then an identical value already exists in backing store and no write-back is needed. The allocator may then prefer to spill clean values over dirty ones.

Because the local allocator looks only at one basic block, it does not make efficient decisions about which registers to spill. One possible metric is to look at all of the virtual registers currently in physical registers and pick the one whose use as a source operand is furthest away in the basic block.

Another serious problem of a local allocator is that it must spill all live virtual registers at the end of the basic block. Consider:

```
BB1:
          %VRx=     ADD       a,b
          BRxx      BB3
BB2:
          %VRx=     ADD       q,r
          BRxx      BB3
          ...
          ...
BB3:
          %VRz=     ADD       %VRx,%VRy
```

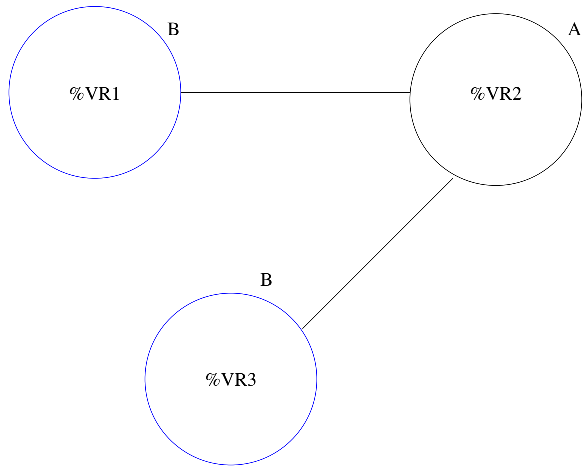
The virtual register %VRx is live on exit from both BB1 and BB2, because control flows from each to BB3 and %VRx is used in BB3 without being redefined. Upon entry to BB3, the "scoreboard" is blank, and %VRx must be fetched from its backing store into a physical register before being used. Therefore, the updated value of %VRx must have previously been placed in that memory location prior to branching out of BB1 and BB2. One might think that this can be avoided by simply assigning the same physical register to %VRx in both basic blocks, but since the local allocator works only on the local basic block, it does not have the power to do this.

Global Allocation

After performing liveness analysis on an entire function, we have a list of live ranges. DEF: A **live range** is a contiguous range of quads (or assembly language opcodes) in one basic block where a given name/virtual register is live. A given virtual register will have one or more live ranges. For example, we have the following live range data:

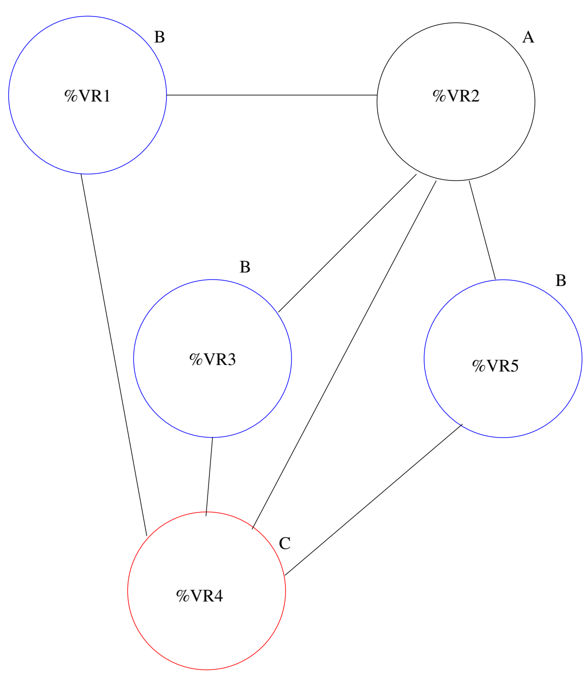
```
          %VR1:               1:1-5

          %VR2:               1:4-7
                              2:3-8

          %VR3:               1:8-10
                              2:5-6
```

Now we can construct a graph (conceptually, it doesn't have to be implemented as a graph) which represents the interference among live ranges. Each virtual register is a node, and if there is a (non-directional) link to another node, it means that their live ranges overlap at *some* point. Here is the graph for the three virtual registers above:

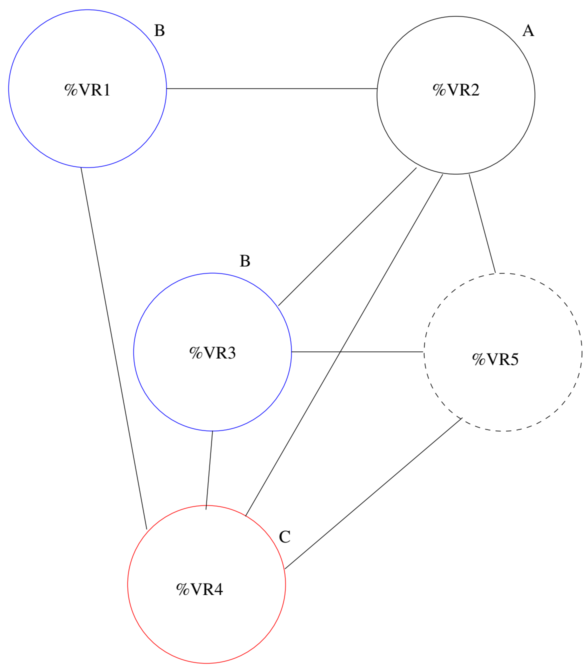


Given this interference graph, global register allocation is the classic map or graph coloring problem. If there are *n* registers available for general use (as opposed to specialized registers, or registers which are held in reserve for certain purposes), then is the interference graph colorable with just *n* colors, such that for each node, each adjacent node is a different color? If so, then all of the names under consideration can be permanently allocated to those general-purpose registers. If not, there are a variety of algorithms to pick which names go into which registers.

In the following examples, let us assume that we have three registers to work with called A, B and C. In the example above, we can color the graph with just two colors, say A and B. Now we introduce virtual registers %VR4 and %VR5 which have the interferences depicted below:

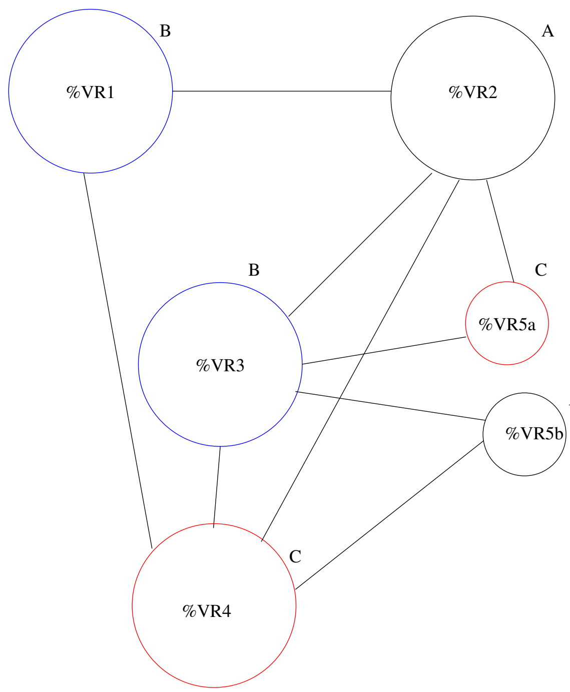


We can still color this graph with three colors, so it is possible to allocate registers without spilling. Now we introduce an interference between VR3 and 5:



We can't color this graph, so now there would be decisions to be made. First, we can make the observation that any node with fewer than *n* neighbors can always be colored, i.e. it can own a register slot without any possibility of interference. Then the remaining, uncolorable nodes need to be prioritized in some manner. One method might be to estimate how many times that name might be accessed. Another factor might be the number of interferences of that node. If the node interferes with many nodes, it might be a good candidate for "spilling" to memory, because doing so will improve the colorability of many other nodes.

Another approach is "range splitting." Let us say that one part of the live range of VR5 interferes with VR2 and 3, but not VR4, and the remaining part interferes with 3 and 4, but not 2. Then we can split VR5 into 5a and 5b:



This above graph is now colorable again using three colors and we don't need to spill. In the "a" part VR5 goes in register C and co-exists with VR2 in register A and VR3 in register B. Then later in the code, VR5 is assigned to again, but this time occupies register A.

If there isn't a natural opportunity for range splitting, then we have to introduce spills. In the above example, if VR5 didn't naturally have two live ranges with the above interference properties, we could artificially create this by spilling register C back into VR5 at the end of the "a" region of code, and then loading it back into register A prior to it being referenced again in the "b" portion of the code.

This first cut at global allocation always puts a specific virtual register into a certain physical register. In a more optimal solution, the virtual register could reside in different physical registers at different places in the code. To do this requires a mechanism for tracking the relationship between definitions and uses, for example, through SSA form as described in the Optimization unit.