1.1. Tracks

A track is a place holder for a register through the assembly code. Let us look at the following example.

```
return a + b & c - d;      1. temp0 = a + b
                           2. temp1 = c - d
                           3. temp2 = temp0 & temp1
                           4. return temp2
```

(a) C code (b) Middle code

The following middle code is generated.

Let us assume that **a**, **b**, **c**, **d**, and **e** are placed on the current activation record of a regular function with offsets 6, 8, 10, and 12, and the **bp** and **di** registers are the regular frame pointer and the variadic frame pointer.

The following assembly code is generated, with the tracks **track0**, **track1**, and **track2** as place holders for the registers. Each track represents a register, but we do not yet know which one.

```
mov track0,[bp + 6]
add track0,[bp + 8]

mov track1,[bp + 10]
sub track1,[bp + 12]

and track0, track1
```

In the return sequence, we reset the variadic and regular frame pointers to the values of the calling function, which is placed at offset 4 and 2 on the activation record, and then jump to the return address, which is placed at the beginning of the activation record. Note the order between the two middle lines. We cannot swap them, since **bp** is set on the third line.

```
mov track2, [bp]
mov di, [bp + 4]
mov bp, [bp + 2]
jmp track2
```

The return value is stored in **track0**, and shall thereby be assigned the return value register. I have chosen **bx** for the return value register, since it can be used both as an operand in arithmetic operations and as an address register. So **track0** is assigned **bx** when the return instruction is added to the code.

The register allocator assigns the remaining tracks **track1** and **track2** to suitable registers. Since **track0** and **track2** do not overlap, it is quite possible to assign them the same register, such as **ax**.

```
mov bx,[bp + 6]
add bx,[bp + 8]

mov ax,[bp + 10]
sub ax,[bp + 12]

and bx, ax

mov ax, [bp]
mov di, [bp + 4]
mov bp, [bp + 2]
jmp ax
```

The **Track** class holds a track representing a known or unknown register.

Track.cs

```
using System;
using System.Collections.Generic;

namespace CCompiler {
  public class Track {
```

The first constructor takes a symbol and a potential register as parameters, and initializes the current and maximum sizes.

```
    public Track(Symbol symbol, Register? register = null) {
      Register = register;
      Assert.Error(symbol != null);
      Assert.Error(!symbol.Type.IsStructOrUnion());
      CurrentSize = m_maxSize = symbol.Type.ReturnSize();
    }
```

The second constructor takes a type and initializes the current and maximum sizes.

```
    public Track(Type type) {
      Assert.Error(type != null);
      Assert.Error(!type.IsStructOrUnion());
      Assert.Error(!type.IsArrayFunctionOrString());
      CurrentSize = m_maxSize = type.Size();
    }
```

A track has a current size and a maximal size. The current size may vary between assembly code instructions. We need the maximal size for the register allocator. It the size more than one byte, a reduced set of register is available.

```
    public int CurrentSize {get;set;}

    private int m_maxSize;
    public int MaxSize {
      get { return m_maxSize; }
      set { m_maxSize = Math.Max(m_maxSize, value); }
    }
```

The min and max index hold he indexes of the first and last assembly code instruction where the track occurs. We need them to decide whether tracks overlap.

```
    private int m_minIndex = -1, m_maxIndex = -1;
    public int Index {
      set {
        m_minIndex = (m_minIndex != -1) ? Math.Min(m_minIndex, value) :
value;
        m_maxIndex = Math.Max(m_maxIndex, value);
      }
    }
```

Although the idea is that a track does not hold a register but is rather assigned a register by the register allocator, there are some cases where the track is assigned a register from the beginning. A track can hold a specific register. In multiplication and division, the left operands must be stored in a specific register. In left or right shift, the right operands must be stored in the **cl** register. When returning a value, it must be stored in a specific register.

```
    public Register? Register {get; set;}
```

If the track is used as a pointer, a reduced set of registers are available.

```
    public bool Pointer {get; set;}
```

The **Overlaps** method returns true if the tracks overlaps. Two tracks overlap if the min and max line numbers overlap.

```
    public static bool Overlaps(Track track1, Track track2) {
      Assert.Error((track1.m_minIndex != -1) && (track1.m_maxIndex != -1));
      Assert.Error((track2.m_minIndex != -1) && (track2.m_maxIndex != -1));
      return !(((track1.m_maxIndex < track2.m_minIndex) ||
                (track2.m_maxIndex < track1.m_minIndex)));
    }
  }
}
```

1.2. Register Allocation

The architecture holds a set of registers. One particular feature of the registers is that they to some extent overlaps. For a more detailed description, see Appendix A.

Register.cs

```
namespace CCompiler {
  public enum Register {al, ah, ax, eax, rax, bl, bh, bx, ebx, rbx,
                        cl, ch, cx, ecx, rcx, dl, dh, dx, edx, rdx,
                        si, esi, rsi, di, edi, rdi,
                        sp, esp, rsp, bp, ebp, rbp};
}
```

The **RegisterAllocator** method take the total track set and total assembly list; that is, the track set for the whole function, and the assembly list for the whole function. The first task is the find out which tracks overlaps each other. We construct the graph in the following way: the tracks are the vertices of the graph and two vertices have an edge if their tracks overlaps. The register allocation is then performed as a graph coloring, two vertices connected by an edge cannot be given the same color. In our case: two tracks that overlaps cannot be assigned the same register.

Graph coloring is a NP-complete problem, which means that there is no known algorithm that works on polynomial time. To be sure that we have found the best solution, we have to exam every possible solution, which would require an unrealistic amount of time. Instead, in this book we perform a **deep search**, where we sort the vertices into a list and for each vertex try to find a register not already allocated by any of its neighbors. If we cannot find such register, we backtrack and try another combination. If we finally find a solution where each vertex is mapped to a register not mapped by any of its neighbors, we have found an optimal solution. If we do not find a solution, we report an error: shortage of registers.

RegisterAllocator.cs

```
using System.Collections.Generic;

namespace CCompiler {
  public class RegisterAllocator {
    public RegisterAllocator(ISet<Track> totalTrackSet,
                             List<AssemblyCode> assemblyCodeList) {
      Graph<Track> totalTrackGraph = new Graph<Track>(totalTrackSet);
```

First, we build the total track graph, with all track as vertices and an edge between two vertices if their tracks overlaps. Two tracks that overlap shall not be assigned the same register, or two overlapping registers.

```
      foreach (Track track1 in totalTrackSet) {
        foreach (Track track2 in totalTrackSet) {
          if (!track1.Equals(track2) && Track.Overlaps(track1, track2)) {
            totalTrackGraph.AddEdge(track1, track2);
          }
        }
      }
```

Then we split the total graph into independent subgraphs. In this way, we can start by any vertex in each of the subgraph. Otherwise, we would have to start at every vertex to be sure that we have reached all the vertexes.

```
      ISet<Graph<Track>> split = totalTrackGraph.Split();
```

When the graph has been split, we iterate through the subgraphs and perform a deep-first search on each of them.

```
      foreach (Graph<Track> trackGraph in split) {
        List<Track> trackList = new List<Track>(trackGraph.VertexSet);
        Assert.Error(DeepFirstSearch(trackList, 0, trackGraph),
                     Message.Out_of_registers);
```

If the **DeepFirstSearch** method returns true, we have a graph coloring; that is, the tracks have been assigned registers that do not overlap. If **DeepFirstSearch** return false, we have not found a graph coloring, and we report an error, that we are out of registers.

```
      ISet<Graph<Track>> split = totalTrackGraph.Split();
      foreach (Graph<Track> trackGraph in split) {
        List<Track> trackList = new List<Track>(trackGraph.VertexSet);
        Assert.Error(DeepFirstSearch(trackList, 0, trackGraph),
                     Message.Out_of_registers);
      }
```

If we have iterated through the track graphs without encountering shortage of register, each track has been assigned a register. We call **SetRegistersInCodeList** that iterates through the code and replace each track with its assigned register.

```
      SetRegistersInCodeList(assemblyCodeList);
    }

    private static void SetRegistersInCodeList(List<AssemblyCode>
                                               assemblyCodeList) {
      foreach (AssemblyCode assemblyCode in assemblyCodeList) {
        if (assemblyCode.Operator == AssemblyOperator.set_track_size) {
          Track track = (Track) assemblyCode[0];
          object operand1 = assemblyCode[1];
```

When iterating through the code list, we also need to keep track of the current size of the tracks, in order to replace the track with a register of correct size.

```
          if (operand1 is int) {
            track.CurrentSize = (int) operand1;
          }
          else {
            track.CurrentSize = ((Track) operand1).CurrentSize;
          }

          assemblyCode.Operator = AssemblyOperator.empty;
        }
        else {
          Check(assemblyCode, 0);
          Check(assemblyCode, 1);
          Check(assemblyCode, 2);
        }
      }
    }
```

The **Check** method checks for each position in the assembly code is a reference a track is replaced by a register of the correct size.

```
    private static void Check(AssemblyCode assemblyCode, int position) {
      if (assemblyCode[position] is Track) {
        Track track = (Track) assemblyCode[position];
        Assert.Error(track.Register != null);
        assemblyCode[position] =
         AssemblyCode.RegisterToSize(track.Register.Value,
track.CurrentSize);
      }
    }
```

The **DeepFirstSearch** method searches the graph in a deep-first manner. It takes the track list and the current index in that list as well as the track graph.

```
    private bool DeepFirstSearch(List<Track> trackList, int listIndex,
                                 Graph<Track> trackGraph) {
```

If the index equals the size of the track list, we return true because we have iterated through the list and found a match; that is, each track has been assigned a register and no overlapping tracks have the same register.

```
      if (listIndex == trackList.Count) {
        return true;
      }
```

If the current track has already been assigned a register, we just call **DeepFirstSearch** with the next index.

```
      Track track = trackList[listIndex];
      if (track.Register != null) {
        return DeepFirstSearch(trackList, listIndex + 1, trackGraph);
      }
```

If the current track has not been assigned a register, we look up the set of possible register and the set of neighbor vertices; that is, the set of overlapping tracks.

```
      ISet<Register> possibleSet = GetPossibleSet(track);
      ISet<Track> neighbourSet = trackGraph.GetNeighbourSet(track);
```

We iterate through the set of possible register and, for each register that does not cause an overlapping, we assign the register to the track and call **DeepFirstSearch** recursively with the next index. If the call returns true, we have found a total mapping of registers to the track, and we just keep returning true. However, if the call does return false, we just try with another of the possible registers. If none of the registers causes a match, we clear the register of the track and return false.

```
      foreach (Register possibleRegister in possibleSet) {
        if (!OverlapNeighbourSet(possibleRegister, neighbourSet)) {
          track.Register = possibleRegister;

          if (DeepFirstSearch(trackList, listIndex + 1, trackGraph)) {
            return true;
          }

          track.Register = null;
        }
      }

      track.Register = null;
      return false;
    }
```

The **OverlapNeighbourSet** method return true if the register overlaps any of its neighbors. The **RegisterOverlap** method in the **AssemblyCode** class test whether two register overlaps.

```
    private bool OverlapNeighbourSet(Register register,
                                  ISet<Track> neighbourSet) {
      foreach (Track neighbourTrack in neighbourSet) {
        if                          (AssemblyCode.RegisterOverlap(register,
neighbourTrack.Register)) {
          return true;
        }
      }

      return false;
    }
```

The **VariadicFunctionPointerRegisterSet** set holds the possible pointer registers of a variadic function while **RegularFunctionPointerRegisterSet** holds the possible pointer registers of a regular function. The **Byte1RegisterSet** set holds all registers of one byte while **Byte2RegisterSet** holds all registers of two bytes.

```
    public static ISet<Register>
      VariadicFunctionPointerRegisterSet = new HashSet<Register>() {
        AssemblyCode.RegisterToSize(Register.bp, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.si, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.di, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.bx, TypeSize.PointerSize)
      },
      RegularFunctionPointerRegisterSet =
        new HashSet<Register>(VariadicFunctionPointerRegisterSet),
      Byte1RegisterSet = new HashSet<Register>() {
        Register.al,Register.ah, Register.bl, Register.bh,
        Register.cl, Register.ch, Register.dl, Register.dh
      },
      Byte2RegisterSet = new HashSet<Register>() {
        Register.ax, Register.bx, Register.cx, Register.dx
      };
```

We remove the frame register from **RegularFunctionPointerRegisterSet** and **VariadicFunctionPointer-RegisterSet**, since we need it to point at the current activation record. We also remove the variadic register from **VariadicFunctionPointerRegisterSet**, since we need it to point at the activation record in a variadic function.

```
    static RegisterAllocator() {
      VariadicFunctionPointerRegisterSet.
        Remove(AssemblyCode.RegularFrameRegister);
      RegularFunctionPointerRegisterSet.
        Remove(AssemblyCode.RegularFrameRegister);
      RegularFunctionPointerRegisterSet.
        Remove(AssemblyCode.VariadicFrameRegister);
    }
```

The **GetPossibleSet** method returns the possible set a track, depending on whether the track holds a pointer, or the size of the track.

```
    private static ISet<Register> GetPossibleSet(Track track) {
      if (track.Pointer) {
        if (SymbolTable.CurrentFunction.Type.IsVariadic()) {
          return RegularFunctionPointerRegisterSet;
        }
        else {
          return VariadicFunctionPointerRegisterSet;
        }
      }
```

If the track does not hold a pointer wee look into its size. If the size is one, we have a larger set to choose from. There are eight non-pointer registers of size one while there are four registers of the other sizes.

```
      else if (track.MaxSize == 1) {
        return Byte1RegisterSet;
      }
```

We return the set of registers of size two, even if the size is actually larger than two. In that case, the **RegisterToSize** method in the **AssemblyCode** class will convert the register to the correct size.

```
      else {
        return Byte2RegisterSet;
      }
    }
  }
}
```

A. The Register Set

The architecture holds a set of overlapping registers.

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
al	ah		
ax			
eax			
rax			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
bl	bh		
bx			
ebx			
rbx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
cl	ch		
cx			
ecx			
rcx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
dl	ah		
dx			
edx			
rdx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
si			
esi			
rsi			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
di			
edi			
rdi			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
bp			
ebp			
rbp			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
sp			
esp			
rsp			