# Register Allocation

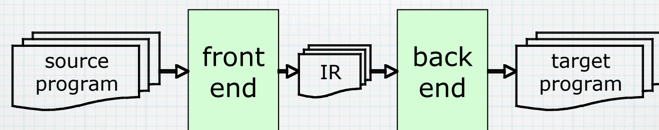
15-745 Optimizing Compilers
Spring 2006

Peter Lee

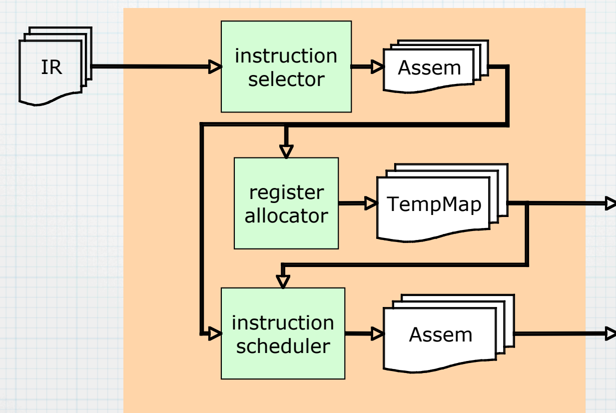
# Announcements

* Read Ch.16 (register allocation)
* Task 2 due today
* Task 3 available **next week**
  * read: efficient path profiling

# Compiler structure




# Back-end structure



## Sample instruction selector run

```
    .text
        .align 4
    .globl l1main
    l1main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %ebx
        pushl   %esi
        movl    $2, t7
        movl    t7, t11
        imull   t7, t11
        movl    t11, t10
        imull   $37, t10
        movl    t10, t8
        movl    t7, t17
        addl    $1, t17
        movl    $33, %eax
        cltd
        idivl   t17
        movl    %eax, t15
        movl    t7, t14
        addl    t15, t14
        movl    t8, t13
        imull   t14, t13
        movl    t13, t8
        movl    $78, t20
        negl    t20
        movl    t8, t19
        subl    t20, t19
        movl    t19, %eax
        popl    %esi
        popl    %ebx
        popl    %edi
        movl    %ebp, %esp
        popl    %ebp
        ret
```

standard prelude

result value in %eax

standard postlude

## Assem representation

```
…
movl    $2, t7
movl    t7, t11
imull   t7, t11
movl    t11, t10
imull   $37, t10
movl    t10, t8
movl    t7, t17
addl    $1, t17
movl    $33, t1
cltd
idivl   t17
…
```

represented as

```
…
OPER("movl\t$2, `d0",    [],     [t7])
MOVE("movl\t`s0, `d0",   [t7],   [t11])
OPER("imull\t`s0, `d0",  [t7],   [t11])
MOVE("movl\t`s0, `d0",   [t11],  [t10])
OPER("imull\t$37, `d0",  [],     [t10])
MOVE("movl\t`s0, `d0",   [t10],  [t8])
MOVE("movl\t`s0, `d0",   [t7],   [t17])
OPER("addl\t$1, `d0",    [],     [t17])
OPER("movl\t$33, `d0",   [],     [%eax])
OPER("cltd",             [%eax], [%edx])
OPER("idivl `s0", [t17], [%eax,%edx])
…
```

Assume
`tempMap: temp -> string`
How do we emit assembly code?

## Assem in ML

```
datatype instr =
          OPER of {assem : string,
                   dst : temp list,
                   src : temp list}

        | MOVE of {assem : string,
                   dst : temp list,
                   src : temp list}

        | DIRECTIVE of {assem : string}

        | COMMENT of string
```

## Assem in Java

```
abstract public class Instruction { … }

public class OperInstruction extends Instruction {
    public OperInstruction(String a, List d, List s)
      { … }
  …
}

public class MoveInstruction extends Instruction {
    public MoveInstruction(List d, List s)
      { … }
  …
}

…
```

# Register allocator’s view

```
…
OPER(“movl\t$2, `d0”,    [],     [t7])
MOVE(“movl\t`s0, `d0”,   [t7],   [t11])
OPER(“imull\t`s0, `d0”, [t7],   [t11])
MOVE(“movl\t`s0, `d0”,   [t11], [t10])
OPER(“imull\t$37, `d0”, [],     [t10])
MOVE(“movl\t`s0, `d0”,   [t10], [t8])
MOVE(“movl\t`s0, `d0”,   [t7],   [t17])
OPER(“addl\t$1, `d0”,    [],     [t17])
OPER(“movl\t$33, `d0”,   [],     [%eax])
OPER(“cltd”,                    [%eax],[%edx])
OPER(“idivl `s0”,    [t17], [%eax,%edx])
…
```

```
…
t7    <- ⊗()
t11   := t7
t11   <- ⊗(t7)
t10   := t11
t10   <- ⊗()
t8    := t10
t17   := t7
t17   <- ⊗()
%eax <- ⊗()
%edx <- ⊗(%eax)
%eax,%edx <- ⊗(t17)
…
```

*Register allocator’s view*

# Register allocator’s job

- Assign each temp to a machine register
- If that fails (due to a shortage of registers), rewrite the code so that it can succeed, and then try again

```
…
t7    <- ⊗()
t11   := t7
t11   <- ⊗(t7)
t10   := t11
t10   <- ⊗()
t8    := t10
t17   := t7
t17   <- ⊗()
%eax <- ⊗()
%edx <- ⊗(%eax)
%eax,%edx <- ⊗(t17)
…
```

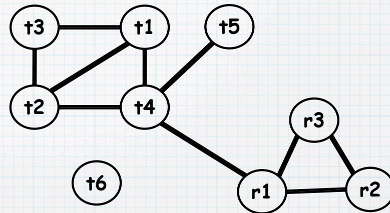
```
…
t7  : %ebx
t8  : %ecx
t10 : %eax
t11 : %eax
t17 : %esi
…
```

*The TempMap*

# A graph-coloring problem

*A small example:*

```
t1 <- ⊗()
t2 <- ⊗()
t3 <- ⊗(t1,t2)
t4 <- ⊗(t1,t3)
t5 <- ⊗(t1,t2)
t6 <- ⊗(t4,t5)
```



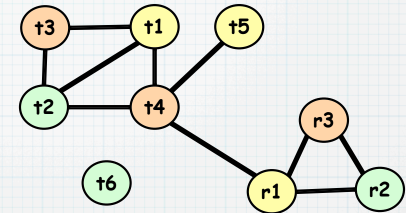

*Assume 3 machine registers, {r1,r2,r3}.*

*Assume t4 may not be in r1.*

*Then we have the **interference graph***

# A graph-coloring problem

```
r1 <- ⊗()
r2 <- ⊗()
r3 <- ⊗(r1,r2)
r3 <- ⊗(r1,r3)
r1 <- ⊗(r1,r2)
r2 <- ⊗(r3,r1)
```




*Assume 3 machine registers, {r1,r2,r3}.*

*Assume t4 may not be in r1.*

*Then we have the **interference graph***

# Steps in register allocation

- **Determine** what temps are **candidates** for register allocation
- Construct the **interference graph**
- **Allocate** registers by coloring the graph with K colors (where K is the number of assignable registers), so that no adjacent nodes have the same color
- **Assign** each temp to the register corresponding to its color

# History

- For early architectures, register allocation was a theoretical curiosity with negligible practical importance
- **Cocke** in 1971 proposed register allocation as a graph-coloring problem
- **Chaitin** was the first to implement this idea, for the IBM 370 PL/1 compiler, in 1981
- In 1982, Chaitin's allocator was used for the landmark **PL.8 compiler** for the IBM 801 RISC system

# History, cont'd

- The RISC revolution inspired many people to think about register allocation
- Motivated by the MIPS architecture, **Chow and Hennessy** developed priority-based coloring in 1984
- Today, one of the most popular algorithms is due to **Briggs**, in 1992

"…since I was a mathematician, the register allocation kept getting simpler and faster as I understood better what was required. I preferred to base algorithms on a simple, clean idea that was intellectually understandable rather than write complicated *ad hoc* computer code…

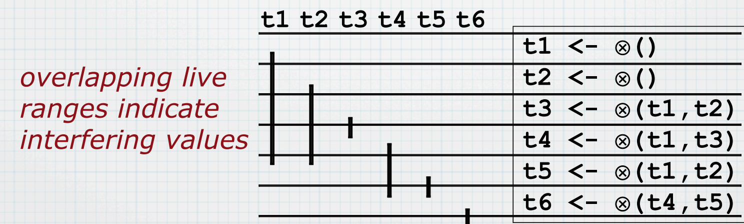
So I regard the success of this approach, which has been the basis for much future work, as a triumph of the power of a simple mathematical idea over ad hoc hacking. Yes, the real world is messy and complicated, but one should try to base algorithms on clean, comprehensible mathematical ideas and only complicate them when absolutely necessary. In fact, certain instructions were omitted from the 801 architecture because they would have unduly complicated register allocation…"

- G. Chaitin, 2004

* Today, register allocation is arguably the single most important optimization
  * memory accesses are expensive
    * even with caches and on CISC machines
  * when it doesn't work well, the performance impact is noticeable

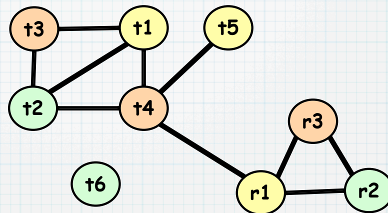
# Live range

* A **live range** for a temp t is a node containing a def of t plus all other nodes for which that def is live

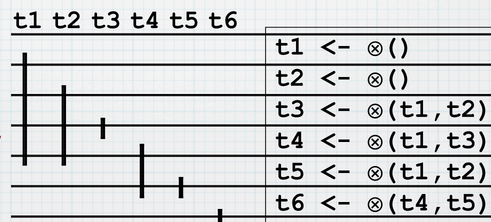



# Live-in sets and the IG

```
{}
{}
{t1}
{t1,t2}
{t1,t2,t3}
{t1,t2,t4}
{t4,t5}
{t6}
```




*overlapping live ranges indicate interfering values*




# Making the IG

* Liveness analysis provides the basic information needed to build the interference graph
* The graph nodes represent the temps plus all of the machine registers
  * each machine register interferes with all other machine registers

# Subtlety #1: MOVE instructions

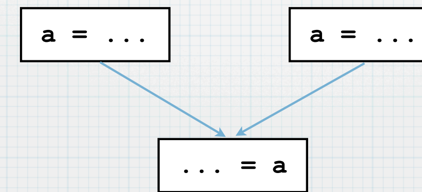
* During graph construction, **MOVE** instructions should be treated specially

```
t := s
 …
x <- ⊗(…s…)
 …
y <- ⊗(…t…)
```

*Note that s and t don't really interfere*

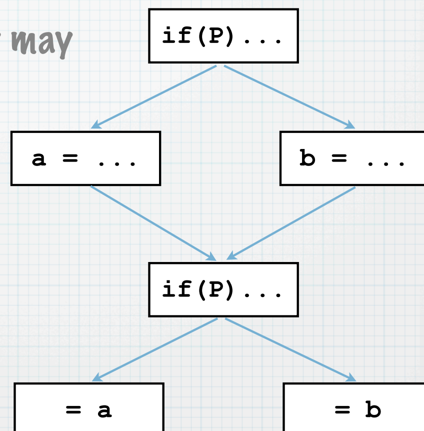
# Subtlety #2: Merging live ranges

* If there are overlapping live ranges for the same temp, they must be merged and considered a single live range
* This requires reaching definitions in addition to liveness analysis




# Subtlety #3: Splitting live ranges

* Two live ranges may may have **unremarkable interferences**




* We'll address these subtleties next time...

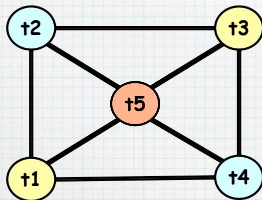
# Graph coloring

- Once we have an interference graph, we can attempt register allocation by searching for a K-coloring
- This is an NP-complete problem (for K>2)
- But a linear-time simplification algorithm (by **Kempe**, 1879) tends to work well in practice

# Kempe's observation

- Given a graph **G** that contains a node **n** with degree less than **K**, the graph is **K**-colorable iff **G** with **n** removed is **K**-colorable
  - This is called the "degree<K" rule
- So, let's try iteratively removing nodes with degree<K
- If all nodes are removed, then **G** is definitely K-colorable

# Doesn't always work...

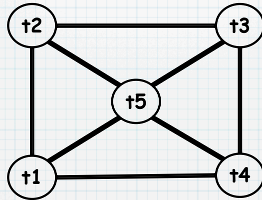



This graph is 3-colorable, but has no nodes with degree < 3

# Kempe's algorithm

- First, iteratively remove degree<K nodes, pushing each onto a stack
- If all get removed, then pop each node and rebuild the graph, coloring as we go
- If we get stuck (i.e., no degree<K nodes), then remove any node and continue

# Try it out...




# Failure

* It is possible (even probable) that Kempe's algorithm will fail for some programs
* In that case we must rewrite the code and try again
* The rewriting involves inserting **spill code**
  * for each node for which we fail to color, rewrite so that its value is fetched from memory prior to each use, and stored to memory after each def

# Spill-code generation

* The effect of spill-code generation is to turn long live ranges into lots of small ones
* This introduces many new temps
* Hence, register allocation must **start over from scratch** whenever spill code is generated

# Chaitin's allocator

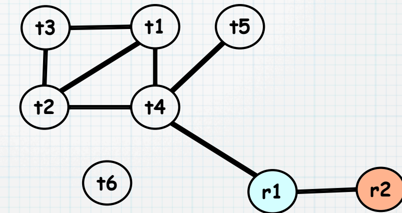
* **Build**: construct the interference graph
* **Simplify**: node removal, a la Kempe
* **Spill**: if necessary, remove a degree≥K node, marking it as a **potential spill**
* **Select**: rebuild the graph, coloring as we go
  * if a potential spill can't be colored, mark it as an **actual spill** and continue
* **Start over**: if there are actual spills, generate spill code and then start over

## Subtlety #4: Choosing potential spills

- When choosing a node to be a potential spill, we want to minimize its performance impact
- Can attempt to compute a spill cost for each temp
  - by estimating performance cost
  - or by using actual profile information
- More on this later...

## Simple example

```
t1 <- ⊗()
t2 <- ⊗()
t3 <- ⊗(t1,t2)
t4 <- ⊗(t1,t3)
t5 <- ⊗(t1,t2)
t6 <- ⊗(t4,t5)
```

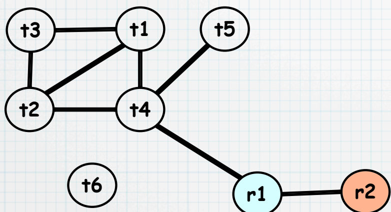



*Assume 2 machine registers, {r1,r2}.*

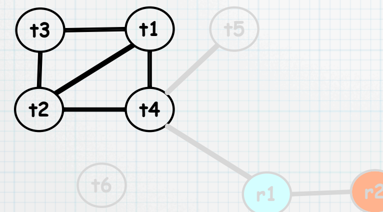
*Assume t4 may not be in r1.*

*Then we have the **interference graph***

## Simplification steps...

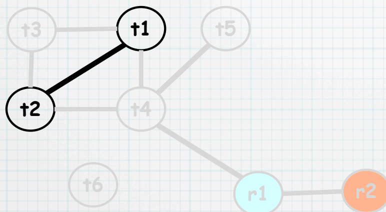



## More simplification...



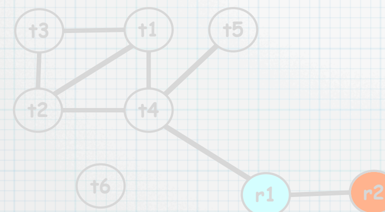

| t6 |
|---|
| t5 |
| r2 |
| r1 |

# Choosing potential spills



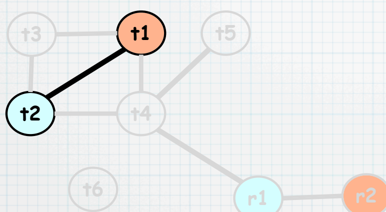

| t6 | |
|---|---|
| t5 | |
| r2 | |
| r1 | |
| t3 | *ps* |
| t4 | *ps* |

# Completing the simplification



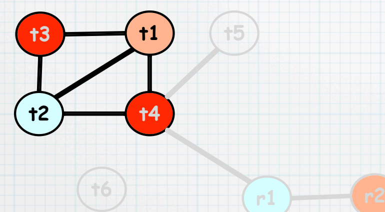

| t6 | |
|---|---|
| t5 | |
| r2 | |
| r1 | |
| t3 | *ps* |
| t4 | *ps* |
| t1 | |
| t2 | |

# Rebuilding...



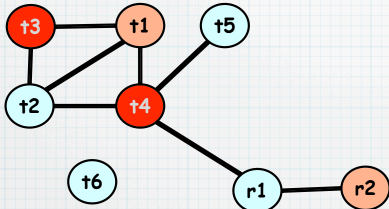

| t6 | |
|---|---|
| t5 | |
| r2 | |
| r1 | |
| t3 | *ps* |
| t4 | *ps* |

# Actual spills

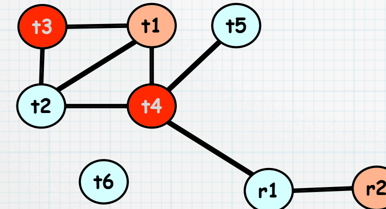



| t6 |
|---|
| t5 |
| r2 |
| r1 |

# Rebuild complete!



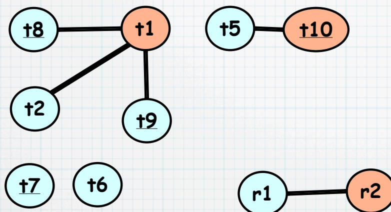

# Spill code generation




```
t1 <- ⊗()
t2 <- ⊗()
t7 <- ⊗(t1,t2)
t3  := t7
t8  := t3
t9 <- ⊗(t1,t8)
t4  := t9
t5 <- ⊗(t1,t2)
t10  := t4
t6 <- ⊗(t10,t5)
```

Live ranges for t3 and t4 have been chopped up

# Start over!




```
r2 <- ⊗()
r1 <- ⊗()
r1 <- ⊗(r2,r1)
slot0  := r1
r1  := slot0
r1 <- ⊗(r2,r1)
slot1  := r1
r1 <- ⊗(r2,r1)
r2  := slot1
r1 <- ⊗(r2,r1)
```

spilled values are loaded just before used

# Subtlety #5: Allocating spill slots

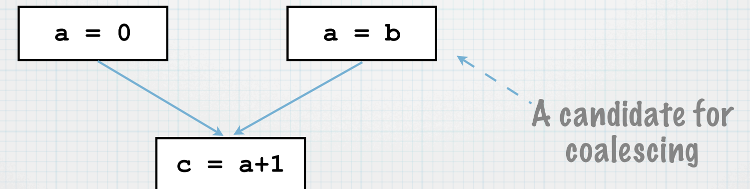
* Spill slots may or may not interfere
* Hence, they can (and should) be allocated just like registers are

# Coalescing

* Compilers generate many temp-temp copies
* If the program contains a copy of the form
  * t = u
* we try to replace, globally, t by u so that we get
  * u = u
* which can then be eliminated

# Coalescing is not copy propagation

* Copy propagation and dead-code elimination can't eliminate all unnecessary copy instructions



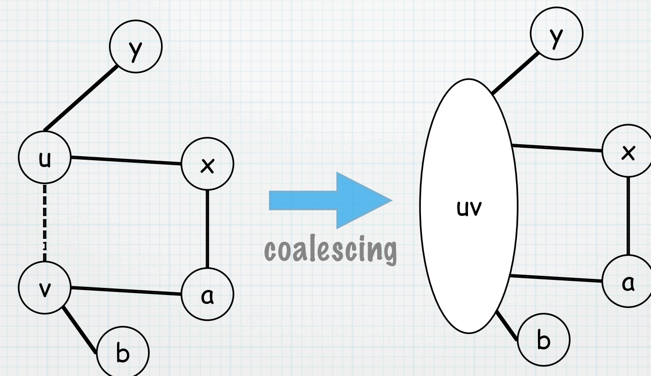

# A common coalescing situation

```
l1main:
        pushl   %ebp
        movl    %esp, %ebp
        t1 := %edi
        t2 := %ebx
        t3 := %esi
        subl    $n, %esp
…
        %esi := t3
        %ebx := t2
        %edi := t1
        movl    %ebp, %esp
        popl    %ebp
        ret
```

The compiler uses "boilerplate" copies to save/restore callee-save registers.

The expectation is that these save/restore operations will be eliminated via coalescing, whenever possible

# Subtlety #6: Coalescing is bad (sometimes)




Which interference graph is 2-colorable?

# Next time...

- Subtleties 1-6
- Register allocation and SSA form