Lecture 15
Register allocation vs instruction scheduling, reverse engineering

Allocation vs. scheduling

We have seen why register allocation is a useful compilation phase: when done well, it can make the best use of available registers and hence reduce the number of spills to memory.

Unfortunately, by maximising the utilisation of architectural registers, register allocation makes instruction scheduling significantly more difficult.

Allocation vs. scheduling

```
*x := *a;
*y := *b;
```

lexing, parsing, translation →

```
LDR v36,v32
STR v36,v33
LDR v37,v34
STR v37,v35
```

compilation ↓

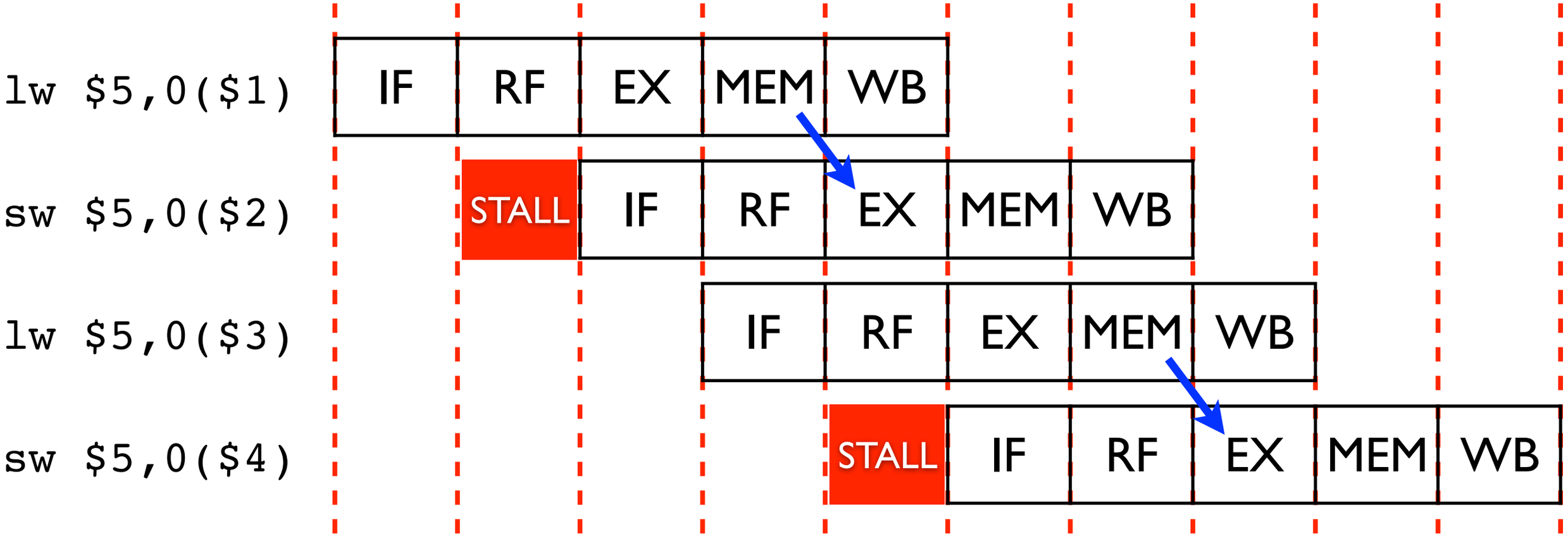
register allocation ↓

```
lw $5,0($1)
sw $5,0($2)
lw $5,0($3)
sw $5,0($4)
```

← code generation

```
LDR v5,v1
STR v5,v2
LDR v5,v3
STR v5,v4
```

Allocation vs. scheduling

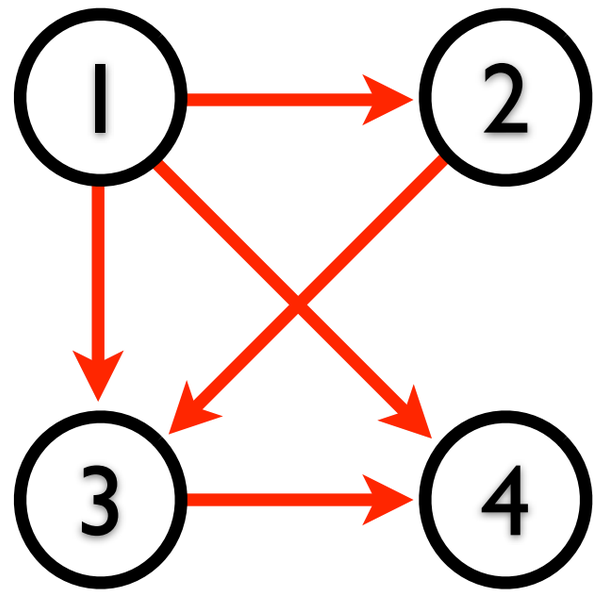



```
lw $5,0($1)
sw $5,0($2)
lw $5,0($3)
sw $5,0($4)
```

This schedule of instructions produces two pipeline stalls (or requires two NOPs).

Allocation vs. scheduling

Can we reorder them
to avoid stalls?




```
1 lw $5,0($1)
2 sw $5,0($2)
3 lw $5,0($3)
4 sw $5,0($4)
```

1, 2, 3, 4

No: this is the *only*
correct schedule for
these instructions.

Allocation vs. scheduling

We might have done better if register $5 wasn't so heavily used.

If only our register allocation had been less aggressive!

Allocation vs. scheduling

```
*x := *a;
*y := *b;
```

lexing, parsing, translation →

```
LDR v36,v32
STR v36,v33
LDR v37,v34
STR v37,v35
```

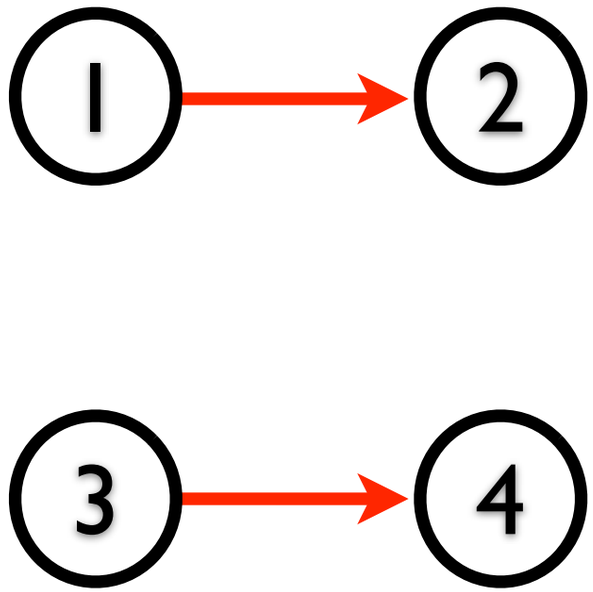
register allocation ↓

```
LDR v5,v1
STR v5,v2
LDR v6,v3
STR v6,v4
```

← code generation

```
lw $5,0($1)
sw $5,0($2)
lw $6,0($3)
sw $6,0($4)
```

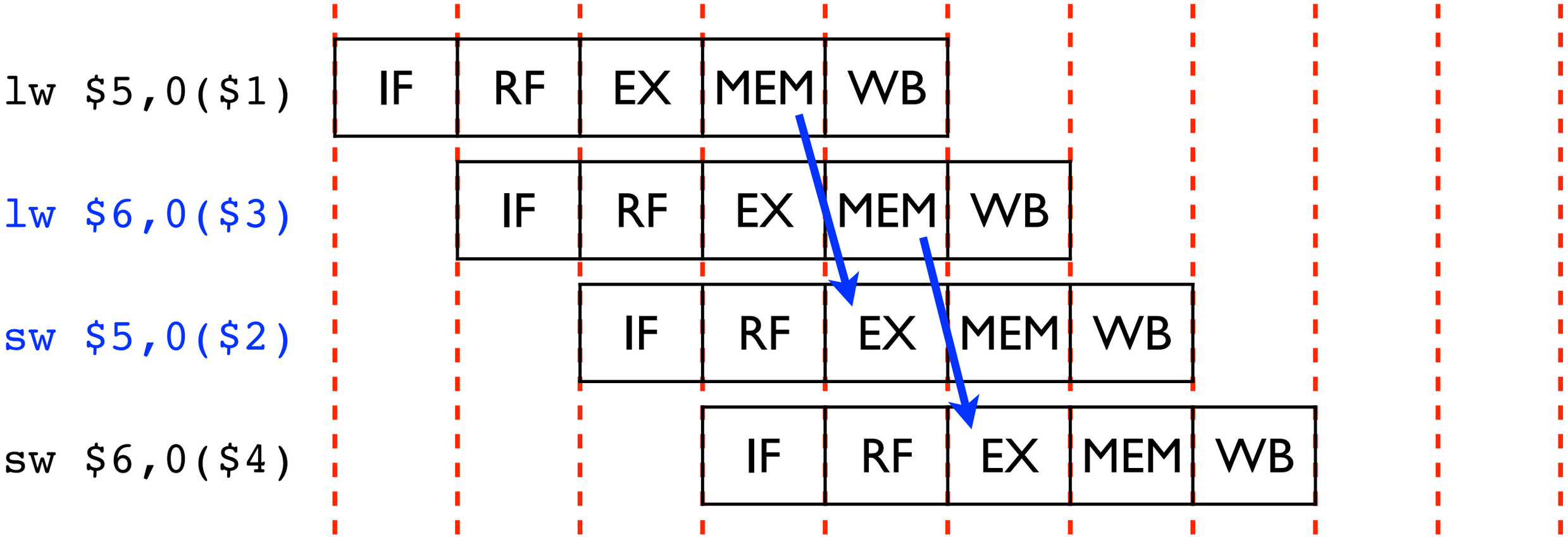
Allocation vs. scheduling




```
1 lw $5,0($1)
2 sw $5,0($2)
3 lw $6,0($3)
4 sw $6,0($4)
```

1, 2, 3, 4
1, 3, 2, 4
3, 1, 2, 4
1, 3, 4, 2
3, 1, 4, 2
3, 4, 1, 2

Allocation vs. scheduling



This schedule of the new instructions produces no stalls.

```
lw $5,0($1)
lw $6,0($3)
sw $5,0($2)
sw $6,0($4)
```

Allocation vs. scheduling

There is clearly antagonism between register allocation and instruction scheduling: one reduces spills by using *fewer* registers, but the other can better reduce stalls when *more* registers are used.

This is related to the *phase-order problem* discussed earlier in the course, in which we would like to defer optimisation decisions until we know how they will affect later phases in the compiler.

It's not clear how best to resolve the problem.

Allocation vs. scheduling

One option is to try to allocate architectural registers cyclically rather than re-using them at the earliest opportunity.

It is this eager re-use of registers that causes stalls, so if we can avoid it — and still not spill any virtual registers to memory — we will have a better chance of producing an efficient program.

Allocation vs. scheduling

In practise this means that, when doing register allocation by colouring for a basic block, we should

- satisfy all of the important constraints as usual (i.e. clash graph, preference graph),
- see how many spare architectural registers we still have left over, and then
- for each unallocated virtual register, try to choose an architectural register distinct from all others allocated in the same basic block.

Allocation vs. scheduling

So, if we are less zealous about reusing registers, this should hopefully result in a better instruction schedule while not incurring any extra spills.

In general, however, it is rather difficult to predict exactly how our allocation and scheduling phases will interact, and this particular solution is quite ad hoc.

Some (fairly old) research (e.g. CRAIG system in 1995, Touati's PhD thesis in 2002) has improved the situation.

Allocation vs. scheduling

The same problem also shows up in dynamic scheduling done by hardware.

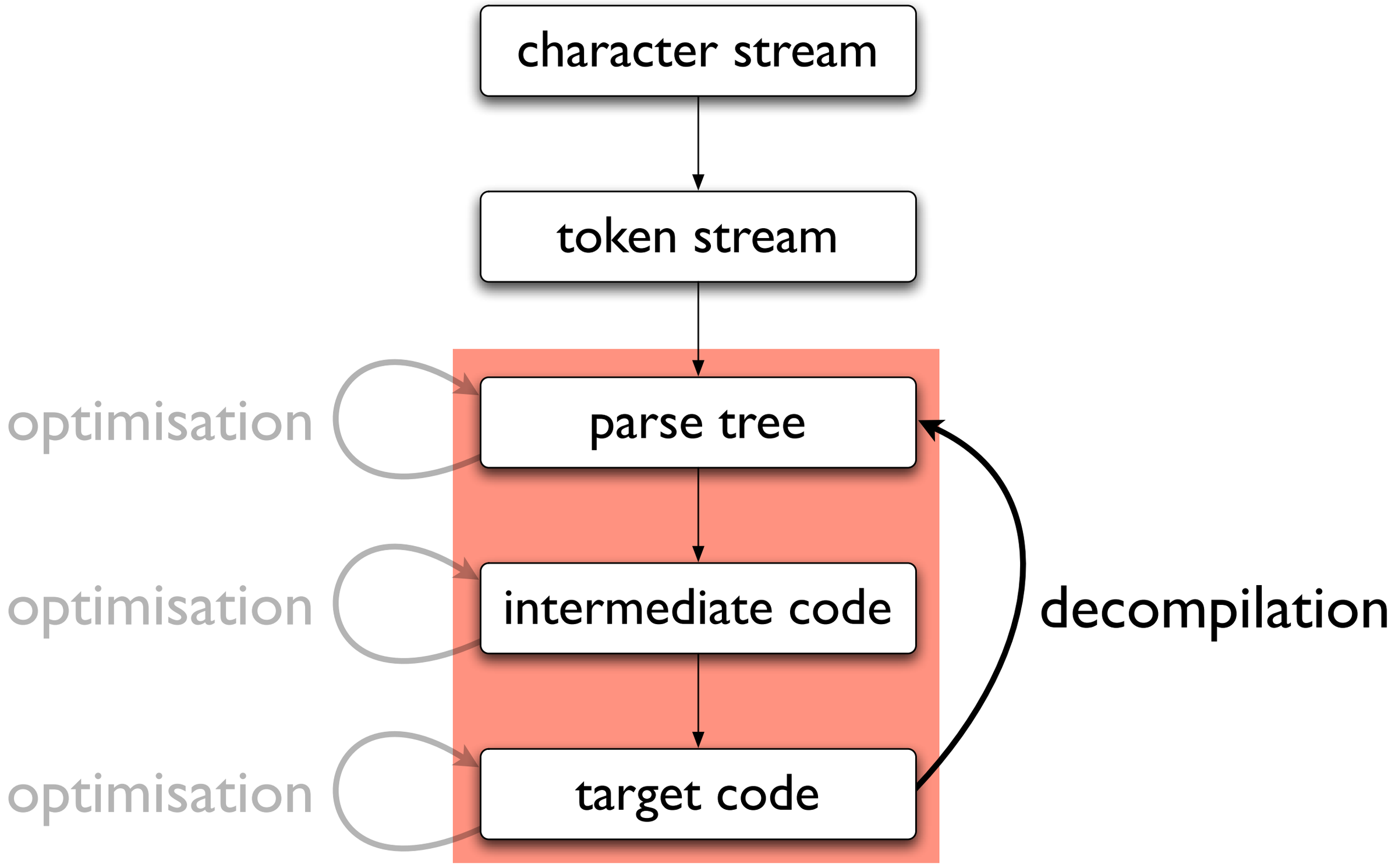
Executable x86 code, for example, has lots of register reuse because of the small number of architectural registers available.

Modern machines cope by actually having more registers than advertised; it does dynamic recolouring using this larger register set, which then enables more effective scheduling.

Part D

Decompilation and reverse engineering

Decompilation



Motivation

The job of an optimising compiler is to turn human-readable source code into efficient, executable target code.

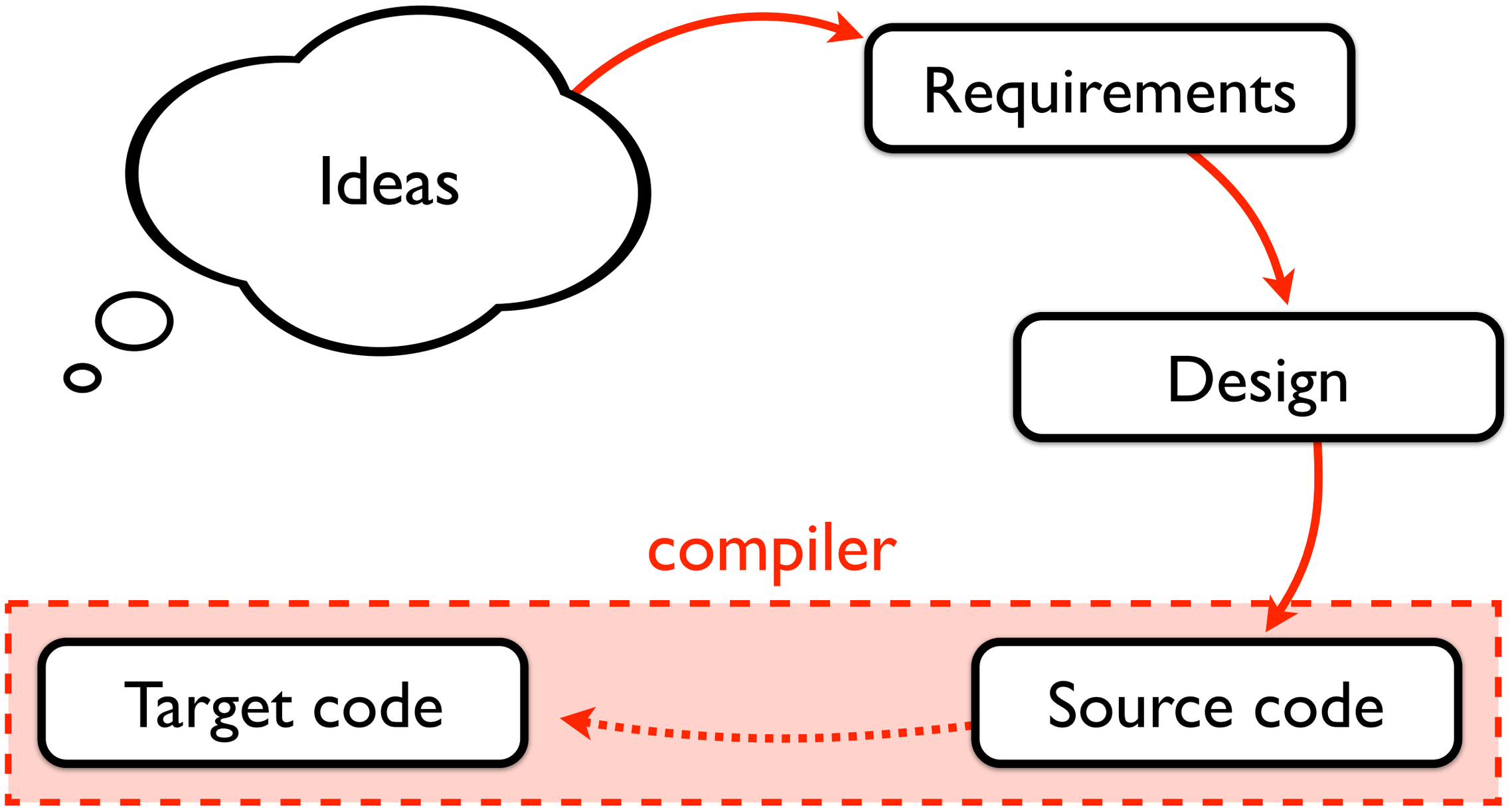
Although executable code is useful, software is most valuable in source code form, where it can be easily read and modified.

The source code corresponding to an executable is not always available — it may be lost, missing or secret — so we might want to use *decompilation* to recover it.

Reverse engineering

In general terms, engineering is a process which decreases the level of abstraction of some system.

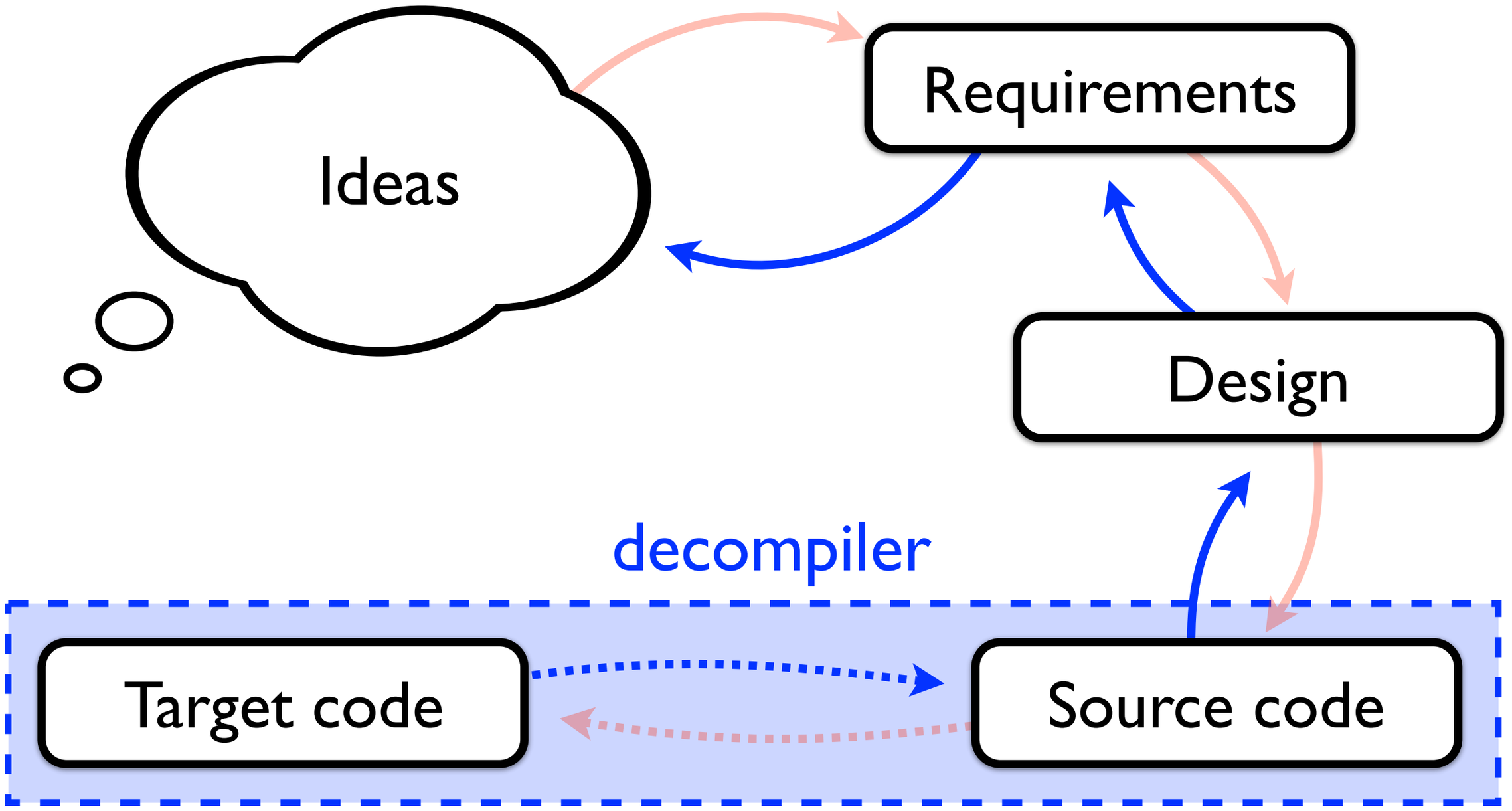
Reverse engineering



Reverse engineering

In contrast, *reverse engineering* is the process of *increasing* the level of abstraction of some system, making it less suitable for implementation but more suitable for comprehension and modification.

Reverse engineering



Legality and ethics

It is quite feasible to decompile and otherwise reverse-engineer most software.

So if reverse-engineering software is technologically possible, is there any e*thical* barrier to doing it?

In particular, when is it *legal* to do so?

Legality and ethics

Companies and individuals responsible for creating software generally consider source code to be their confidential intellectual property; they will not make it available, and they do not want you to reconstruct it.

(There are some well-known exceptions.)

Usually this desire is expressed via an end-user license agreement, either as part of a shrink-wrapped software package or as an agreement to be made at installation time ("click-wrap").

Legality and ethics

However, the European Union Software Directive of 1991 (91/250/EC) says:

Article 4 Restricted Acts

Subject to the provisions of Articles 5 and 6, the exclusive rights of the rightholder within the meaning of Article 2, shall include the right to do or to authorize:

(a) the permanent or temporary reproduction of a computer program by any means and in any form, in part or in whole. Insofar as loading, displaying, running, transmission or storage of the computer program necessitate such reproduction, such acts shall be subject to authorization by the rightholder;

(b) the translation, adaptation, arrangement and any other alteration of a computer program and the reproduction of the results thereof, without prejudice to the rights of the person who alters the program;

(c) any form of distribution to the public, including the rental, of the original computer program or of copies thereof. The first sale in the Community of a copy of a program by the rightholder or with his consent shall exhaust the distribution right within the Community of that copy, with the exception of the right to control further rental of the program or a copy thereof.

Article 5 Exceptions to the restricted acts

1. In the absence of specific contractual provisions, the acts referred to in Article 4 (a) and (b) shall not require authorization by the rightholder where they are necessary for the use of the computer program by the lawful acquirer in accordance with its intended purpose, including for error correction.

2. The making of a back-up copy by a person having a right to use the computer program may not be prevented by contract insofar as it is necessary for that use.

3. The person having a right to use a copy of a computer program shall be entitled, without the authorization of the rightholder, to observe, study or test the functioning of the program in order to determine the ideas and principles which underlie any element of the program if he does so while performing any of the acts of loading, displaying, running, transmitting or storing the program which he is entitled to do.

Article 6 Decompilation

1. The authorization of the rightholder shall not be required where reproduction of the code and translation of its form within the meaning of Article 4 (a) and (b) are indispensable to obtain the information necessary to achieve the interoperability of an independently created computer program with other programs, provided that the following conditions are met:

(a) these acts are performed by the licensee or by another person having a right to use a copy of a program, or on their behalf by a person authorized to to so;

(b) the information necessary to achieve interoperability has not previously been readily available to the persons referred to in subparagraph (a); and (c) these acts are confined to the parts of the original program which are necessary to achieve interoperability.

2. The provisions of paragraph 1 shall not permit the information obtained through its application:

(a) to be used for goals other than to achieve the interoperability of the independently created computer program;

(b) to be given to others, except when necessary for the interoperability of the independently created computer program; or (c) to be used for the development, production or marketing of a computer program substantially similar in its expression, or for any other act which infringes copyright.

Legality and ethics

"The authorization of the rightholder shall not be required where [...] translation [of a program is] necessary to achieve the interoperability of [that program] with other programs, provided [...] these acts are performed by [a] person having a right to use a copy of the program"

Legality and ethics

The more recent European Union Copyright Directive of 2001 (2001/29/EC, aka "EUCD") is the EU's implementation of the 1996 WIPO Copyright Treaty.

It is again concerned with the ownership rights of technological IP, but Recital 50 states that:
"[this] legal protection does not affect the specific provisions [of the EUSD]. In particular, it should not apply to [...] computer programs [and shouldn't] prevent [...] the use of any means of circumventing a technological measure [allowed by the EUSD]."

Legality and ethics

And the USA has its own implementation of the WIPO Copyright Treaty: the Digital Millennium Copyright Act of 1998 (DMCA), which contains a similar exception for reverse engineering:

"This exception permits circumvention [...] for the sole purpose of identifying and analyzing elements of the program necessary to achieve interoperability with other programs, to the extent that such acts are permitted under copyright law."

Legality and ethics

Predictably enough, the interaction between the EUSD, EUCD and DMCA is complex and unclear, particularly at the increasingly-blurred interfaces between geographical jurisdictions (cf. Dmitry Sklyarov), and between software and other forms of technology (cf. Jon Johansen).

Get a lawyer.

Clean room design

Despite the complexity of legislation, it is possible to do useful reverse-engineering without breaking the law.

In 1982, Compaq produced the first fully IBM-compatible personal computer by using *clean room design* (aka "Chinese wall technique") to reverse-engineer the proprietary IBM BIOS.

This technique is effective in legally circumventing copyrights and trade secrets, although not patents.

Summary

- Register allocation makes scheduling harder by creating extra dependencies between instructions
- Less aggressive register allocation may be desirable
- Some processors allocate and schedule dynamically
- Reverse engineering is used to extract source code and specifications from executable code
- Existing copyright legislation may permit limited reverse engineering for interoperability purposes