Code generation for superscalar RISC-processors

What are RISC and CISC?

- CISC:
  (Complex Instruction Set Computers)

  Example:

  mem(r1+r2) = mem(r1+r2)*mem(r3+disp)

- RISC:
  (Reduced Instruction Set Computers)

  Example:

```
loadf   fp1 = mem(r1+r2)     # load first fl.p. number
loadf   fp2 = mem(r3+disp)   # load the next
multf   fp3 = fp1,fp2        # multiplication of fl.p.n
storef  mem(r1+r2) = fp3     # store result
```


Characteristics of RISC processors

- Instructions perform primitive operations
  (simply load, store or register operation)
- All instructions with memory references either load into a register, or store contents from a register
  (load-store architectures)
- Often several sets of registers
  integer registers, floating-point number registers, shadow registers

Moreover

- All instructions are of the same length
  (quick decoding)
- No implicitly set conditional registers
  (To be set explicitly, tested by branch instructions)

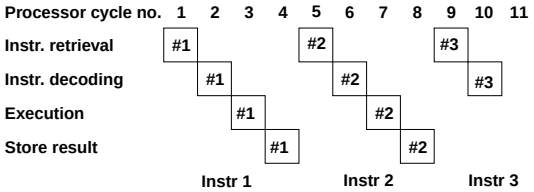
Advantages

- The compiler has direct access to and can manipulate performance-improving code features
- Code generation “is simplified” as there are fewer primitives to choose from
- Fewer instructions - smaller chip area
  (the area can be used to make the remaining instructions run faster)

Processors with and without pipelining

- Traditional processor without pipelining

  One instruction takes 4 processor cycles, i.e. 0.25 instructions/cycle

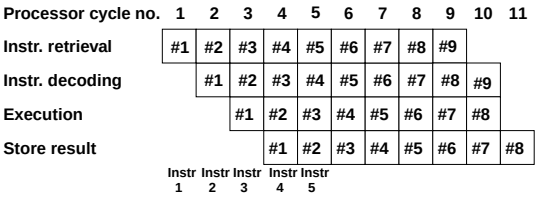


-
-
-
-
-

Processor with simple pipelining

An instruction takes 1 cycle on average with pipeline
i.e. 1 instruction/cycle

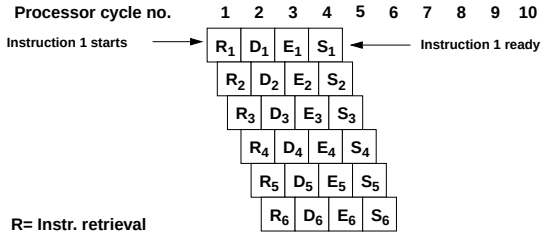
This pipeline achieves 4-way parallelism



Processor with super-pipelining

A new instruction can begin before the previous one is finished.

Thus you manage on average 3 instr/cycle when the pipeline is full.



R= Instr. retrieval

D= Instr. decoding

E= Execution

S= Store result

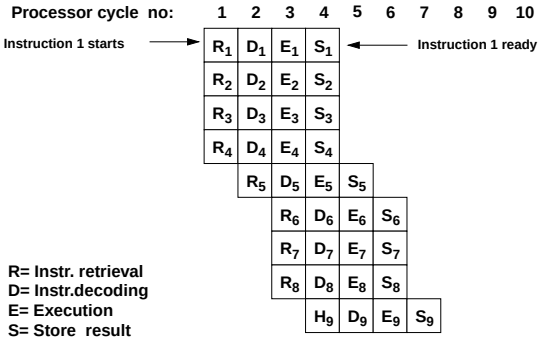
Superscalar processors

A superscalar processor has several function units that can work in parallel and which can load more than 1 instruction per cycle.

The word superscalar comes from the fact that the processor executes more than 1 instruction per cycle.

The diagram below shows how a maximum of 4 units can work in parallel, which in theory means they work 4 times faster.

The type of parallelism used depends on the type of instruction and dependencies between instructions.



R= Instr. retrieval
D= Instr.decoding
E= Execution
S= Store result

A parallel pipeline

(Weiss & Smith, figure 1.11)

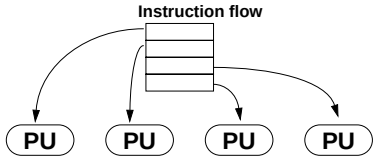
A superscalar pipeline

(Weiss & Smith, figure 1.12)

Comparison between superscalar processors and VLIW processors

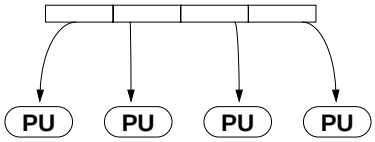
Superscalar

with multiple loading of instructions (multi-issue):



Several processor units are loaded simultaneously with different instructions

VLIW (Very Long Instruction Word):



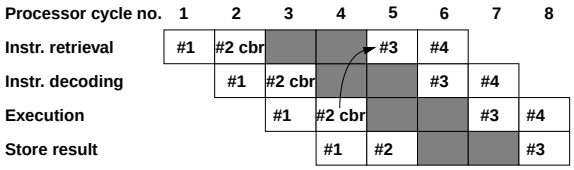
Several processor units are loaded simultaneously by different operations in the same instruction.
(E.g. The multiflow machine, 1024 bits, 28 operations)
(E.g. some specialized graphic processors.)

Problems using branch instructions on simple pipelined processors

Branch instructions force the pipeline to restart and thus reduce performance.

The diagram below shows execution of a branch (cbr = conditional branch) to instruction #3, which makes the pipeline restart.

The grey area indicates lost performance. Only 4 instructions start in 6 cycles instead of the maximum of 6.

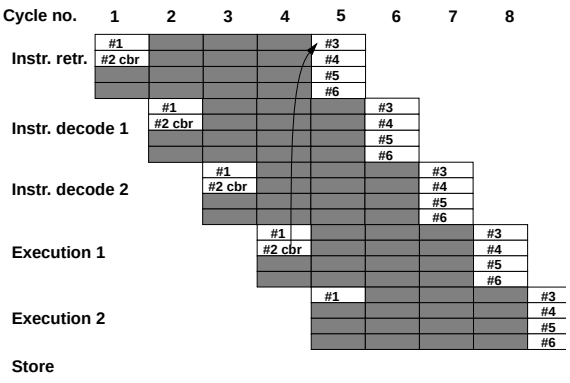


Branch effects on performance for deeply pipelined superscalar processors

Branch-instructions force the pipeline to restart and thus reduce performance.

The diagram below shows execution of a branch (cbr = conditional branch) to instruction #3, which makes the pipeline restart.

The grey area indicates lost performance. Only 6 instructions start during 5 cycles instead of a maximum of 20.



Various problems the compiler must deal with on superscalar RISC-processors

- Efficient global register allocation.

  This is becoming increasingly important as memory access takes longer compared with executing instructions with new fast processors.

- Instruction scheduling.

  Major performance gains can be made by proper scheduling (= timetabling) of instructions on function units working in parallel.

- Branch prediction

  To see in advance the direction a branch takes so that code can be optimized so that a particular case leads to a full pipeline.

- Loop unfolding

  Reduce the number of branches so the pipeline is filled better.

- "Software pipelining"
  Overlap: Load, Execute, Save in a Loop

"Software pipelining"

(Weiss & Smith, figure 1.1 6)

for i := l to n

get values;

compute;

store;

end for

---

get values 1 get values 2 get values 3

compute 1 compute 2 In

store 1 parallel

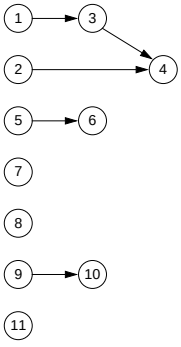
The code-scheduling problem

- Schedule the instructions in such an order that parallel function units are used to the greatest possible degree.
- Input:
  - Instructions to be scheduled
  - A data dependency graph
  - A processor architecture
  - Register allocation has been performed
- Output:
  - A scheduling of instructions which minimises delays

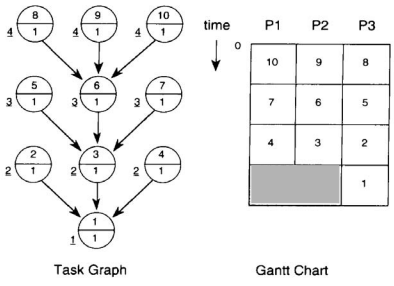
Code to be scheduled:

```
(1)     ld    [%sp + 0x64], %g1
(2)     ld    [%sp + 0x68], %l1
(3)     add   %l4, %g1, %l2
(4)     add   %l2, %l1, %o1
(5)     sethi %hi(0x2000), %l7
(6)     or    %l7, 0x240, %l7        ! 0x2240
(7)     clr   %o0
(8)     mov   0x5, %o2
(9)     sethi %hi(0x80000000), %o3
(10)    or    %o3, 0x2, %o3          ! -0x7ffffffe
(11)    mov   %l6, %o4
```

Dependency graph for this code:

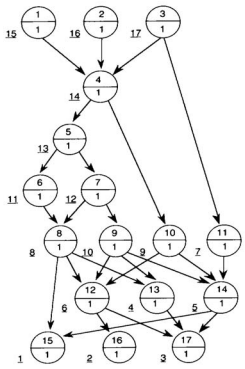


Example: Highest-Level-First algorithm used on a tree-structured task graph.

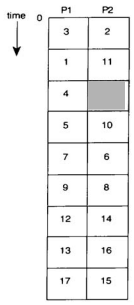


- The level of a task node is the maximal number of nodes that are passed on the way to the final node, itself included.
- The algorithm:
  - **The level of each node is used as priority.**
  - **When a processor/function unit is free, assign the unexecuted task which has highest priority and which is ready to be executed.**

Example of the Highest-level-first algorithm applied to an arbitrary task graph for 2 processors



Task Graph



Gantt Chart

Global Register Allocation

A global register allocator decides the content of the limited set of processor registers during execution.

It tries to use the registers so that the number of memory references is minimised over an area which covers up to one procedure body.

“adds the largest single improvement”,
20%-30% improvement, sometimes factor of 1-2

Important for other optimisations which create many temporaries.

If these have to be stored and retrieved from memory, these optimisations can even increase execution time.

When ?

Register allocation is normally performed at the end of global optimisation, when the final structure of the code and all potential use of registers is known.

It is performed on abstract machine code where you have access to an unlimited number of registers or some other intermediary form of program.

The code is divided into sequential blocks (basic blocks) with accompanying control flow graph.

Basic concepts

A *variable* in register allocation can be a program variable, one temporarily generated by the compiler or a constant.

A variable is *defined* at a point in the program if the variable is given a value.

A variable is *used* at a point in the program if its value is referenced in an expression.

A variable is *alive* at a point if it is referenced there or at some following point which has not been preceded by any redefinition.

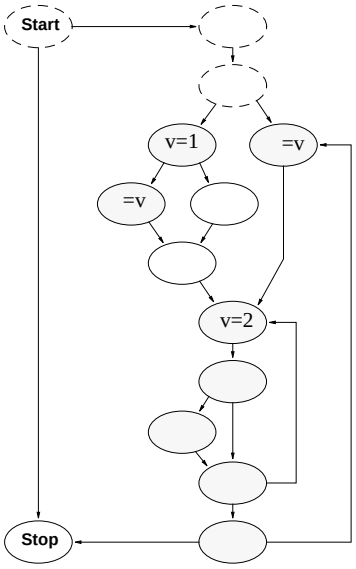
A variable is *reaching* at a point if an (arbitrary) definition, or usage (because a variable can be used before it is defined) reaches the point.

Live range

A variable's *live range* is the area in the code (set of all basic blocks) where the variable is both alive and reaching. This range does not need to be consecutive.

(Procedure calls are treated specially depending on the linking convention)

Examples of various live ranges



Interference graph

Each connected component in the live range is a *"proper" live range* for the variable. Each of these can be assigned a separate register. Certain algorithms do not make this distinction (explicitly).

The live ranges of two variables *interfere* if their intersection is not empty.

Each live range builds a node in the *interference graph* (or *conflict graph),* and if two live ranges interfere, an edge is drawn between the nodes.

Two adjacent nodes in the graph can not be assigned the same register.

Colouring

Register allocation can be compared with the classic colouring problem. That is, to find a way of colouring - with a maximum of *k* colours - the interference graph which does not assign the same colour to two adjacent nodes.

k = the number of registers. On a RISC-machine there are, for example, 16 or 32 general registers. Certain methods use some registers for other tasks. e.g., for spill code.

The chromatic number $\gamma(G)$ of a graph *G* is the smallest number of colours needed to colour the graph.

Determining whether a graph is colourable using *k* colours is NP-complete.

In other words, it is unmanageable always to find an optimal solution.

Colouring (continued)

- We have thus two problems:

1. How can we colour in a good, quick way?

This is needed in order to perform global (at procedure level) register allocation in a reasonable time.

2. What do we do if more than *k* colours are needed?

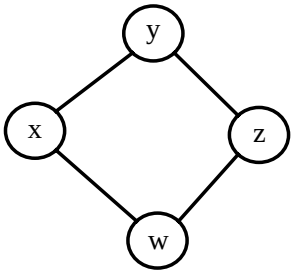
That is, if there are not enough registers.

Chaitin's Algorithm (1981)

- Performs colouring of an interference graph
- 1 register per colour

Example of an interference graph:

2.



Conflict with instruction scheduling.

If register allocation is performed before, false dependencies arise with re-use of registers. This limits possibilities of moving code.

If scheduling is performed first, the live range will be larger and therefore allocation will be more difficult with more spill code.

Furthermore the exact register assignment is needed in some cases by the scheduler.

This can be solved by joining these two steps together.

Register allocation using hierarchical, cyclic interval graphs

Interference graphs have some weaknesses:

- Imprecise information on how and when live ranges interfere.
- No special consideration is taken of loop variables' live ranges (except when calculating priority).

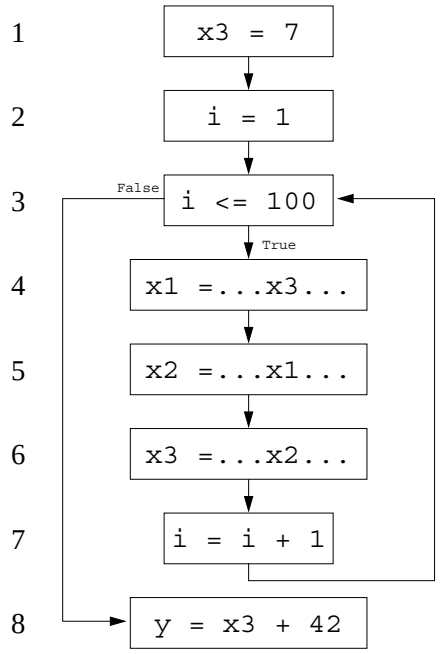
In a cyclic interval graph:

- The time relationships between the live ranges are explicit.
- Live ranges are represented for a variable whose live range crosses iteration limits by cyclic intervals.

Example

```
x3 = 7
for i = 1 to 100 {
    x1 = ... x3 ...
    x2 = ... x1 ...
    x3 = ... x2 ...
}
y = x3 + 42
```


Example, continued.



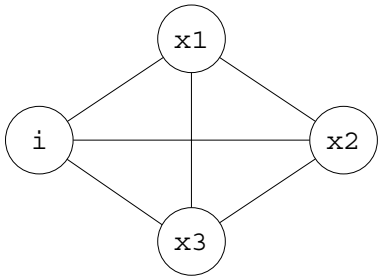
Example (continued)

$lr_{\mathtt{i}} = \{2, 3, 4, 5, 6, 7\}$

$lr_{\mathtt{x1}} = \{4, 5\}$

$lr_{\mathtt{x2}} = \{5, 6\}$

$lr_{\mathtt{x3}} = \{1, 2, 3, 4, 6, 7, 8\}$



Live intervals for loops

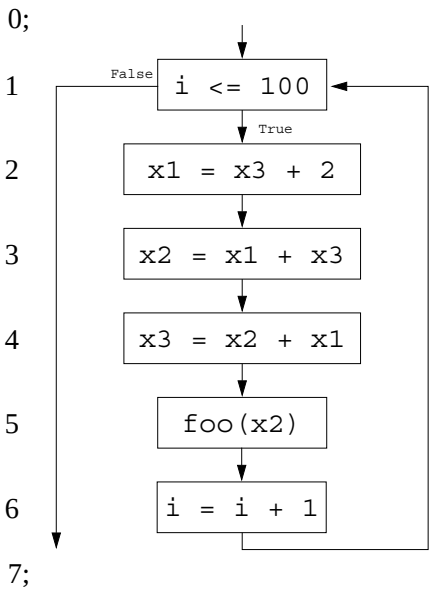
We examine single loops, inner loops first.

To achieve a compact notation:

- Intervals for loop variables which do not cross the iteration limit are included precisely once.
- Intervals which cross the iteration limit are represented as an interval pair, *cyclic interval*: $([0, t'), [t, t_{\text{end}}])$

Cyclic interval graph

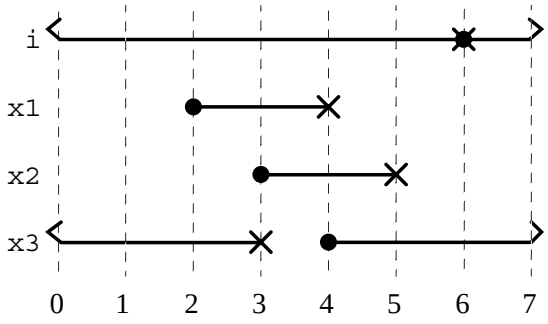
Somewhat modified example:



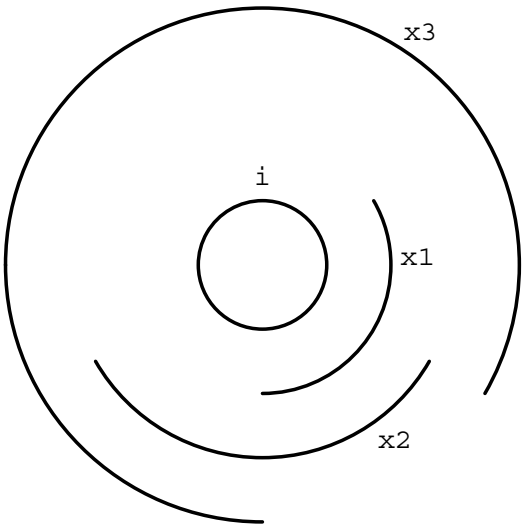
Cyclic interval graph (continued)

The following are used for the example:

`i`: ([0, 6), [6, 7])
`x1`: [2, 4)
`x2`: [3, 5)
`x3`: ([0, 3), [4, 7])



Circular edge graph



Tests

The circular interval method has been compared with the methods which are used in three advanced C-compilers (highest level of optimisation) for:

- IBM RS6000
- Sun Sparc (SunOS 4.1.1)
- MIPS

Very good results, often by a factor of 2 to 3 fewer load/store instructions, or even better.

The only one using "chameleon intervals". Other approaches involve costly register waste.