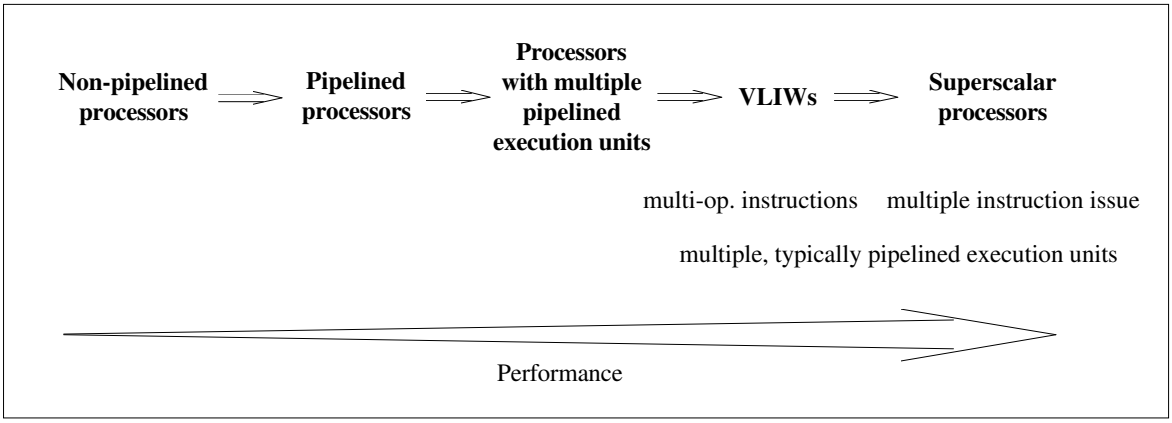
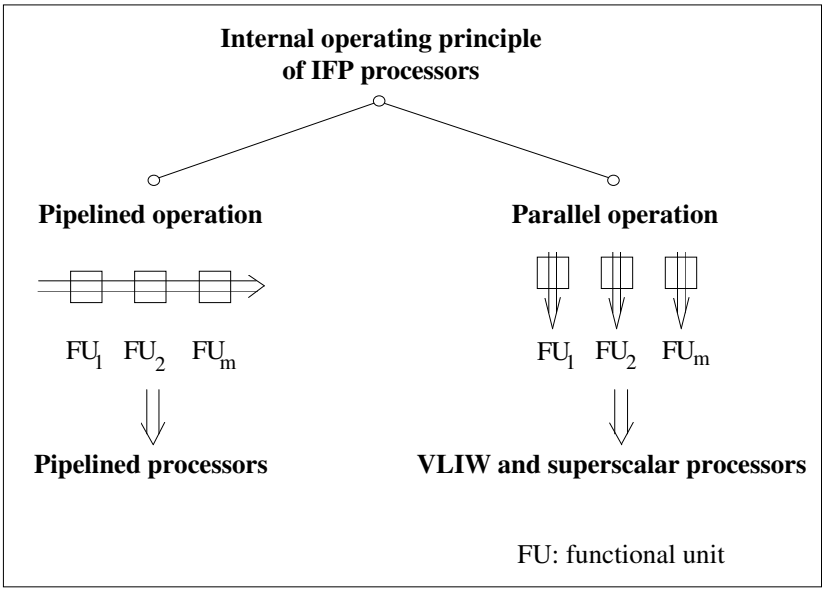
The IFP achitectures

The main line of development of von Neumann processors with reference to parallel instruction processing



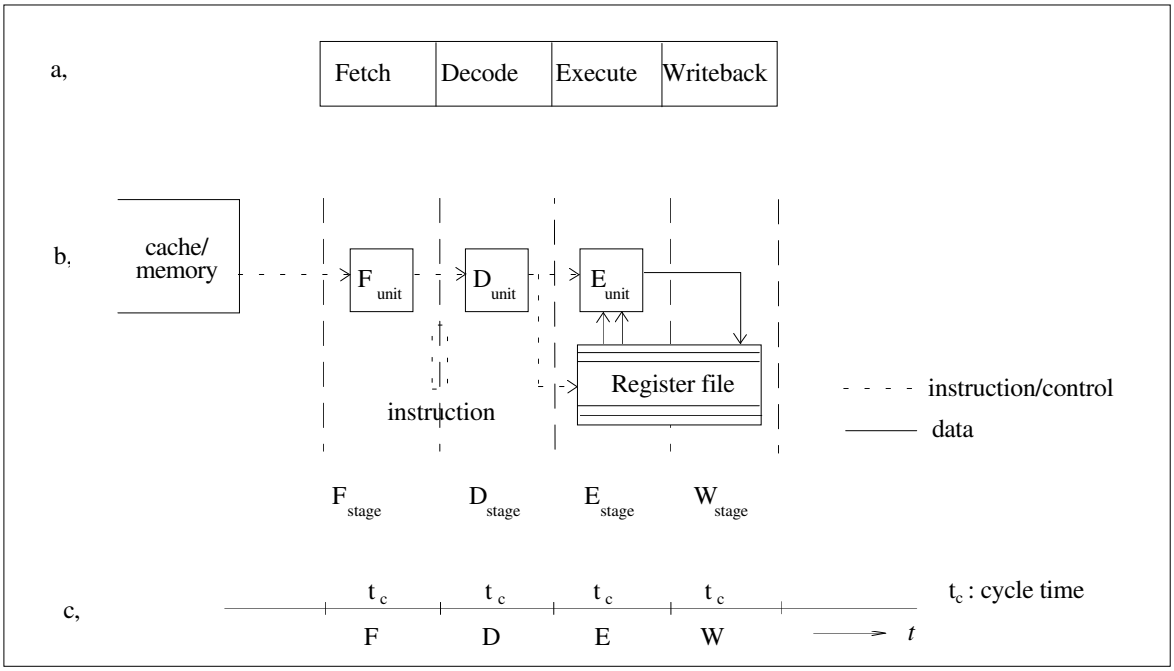
Overview of IFP processors



At pipelined operation instruction processing is subdivided into a number of successive subtasks, such as instruction fetch (F), decode, (D), execute (E) and writeback of the result (W)

At parallel operation a number of functional units (FU), each of them capable of parallel operation on data fetched from a set of registers. Now, multiple FUs execute instructions and write back results into the register file in parallel.

Pipeline stages



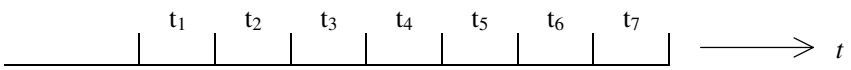
a: Subtasks of instruction execution for integer register-register operations

b: Basic structure of an integer pipeline

c: Timing of the execution

All the pipeline stages operate like an assembly line, with synchron timing of the operation. At the beginning of every pipeline cycle each stage accepts a new input and at the same time it delivers its output to the next stage, except the last one, which writes the result into the specified destination

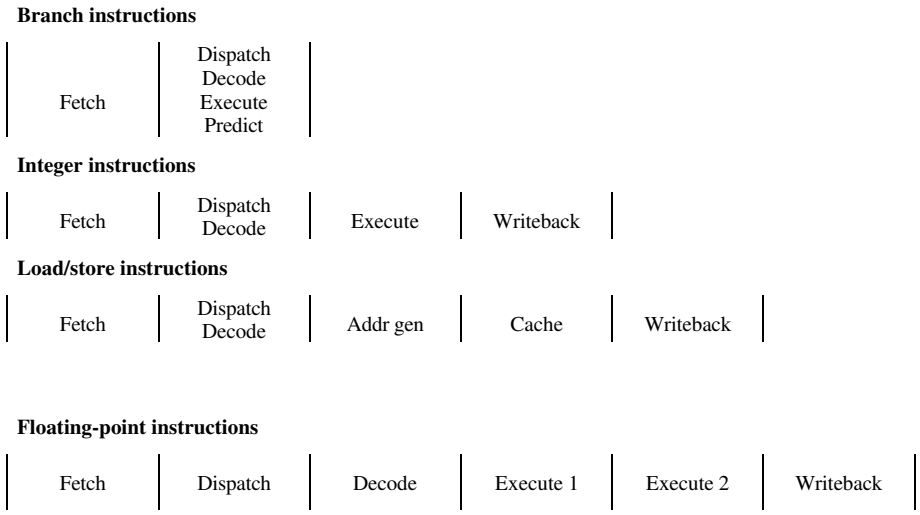
Parallel execution of instructions in a traditional pipelined processor



instr. 1	F_1	D_1	E_1	W_1			
instr. 2		F_2	D_2	E_2	W_2		
instr. 3			F_3	D_3	E_3	W_3	
instr. 4				F_4	D_4	E_4	W_4
instr. 5					F_5	D_5	E_5
instr. 6						D_6	D_6
instr. 7							F_7

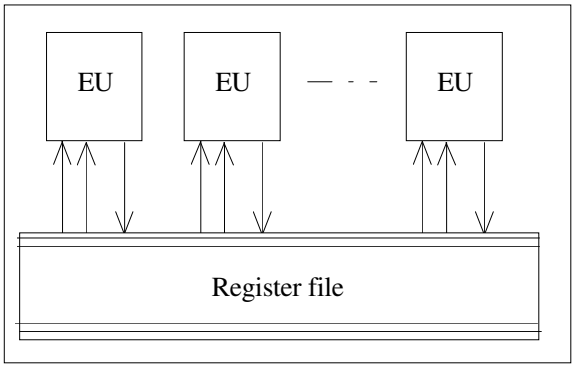
It is a significant feature of pipelined execution that *in each cycle a new instruction* can enter the pipeline. Evidently, if an instruction has completed the last phase of its execution, it retires. Thus, as many instructions can be executed in parallel, as there are pipeline stages, that is four in our example.

Number and kind of pipeline stages in the pipelined execution units of the PowerPC 601



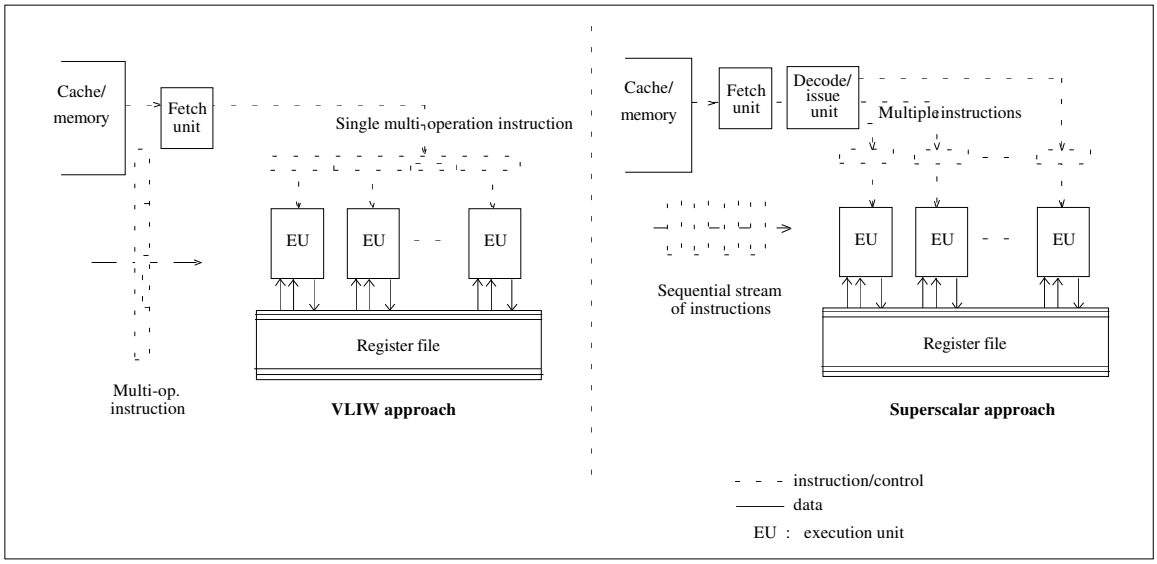
Pipelining offers higher performance at a cost of increased hardware complexity. Accordingly, pipelined processors were first employed in very expensive super- and vector processors, around 1970. Subsequently, they became the standard implementation of mainframes and, in the 80s, of microprocessors. At present, pipelined execution units (such as fixed- or floating point execution units) are commonly used as building blocks of advanced processors, like VLIW or superscalar ones. For example, the PowerPC 601 has three pipelined execution units, each one for fixed-point, floating-point- and branch execution. The number of required pipeline stages depends, first of all, on the kind of the operation to be performed. Usually, two to ten stages are used. For instance, the pipelined execution units of the PowerPC 601 use two to six stages.

Common basic structure of conventional VLIW and superscalar processors



A common feature of VLIW and superscalar processors is that both are made up of a number of execution units (EU), each of them capable of parallel operation on data fetched from a set of registers. Now, multiple EUs execute instructions and write back results into the register file in parallel.

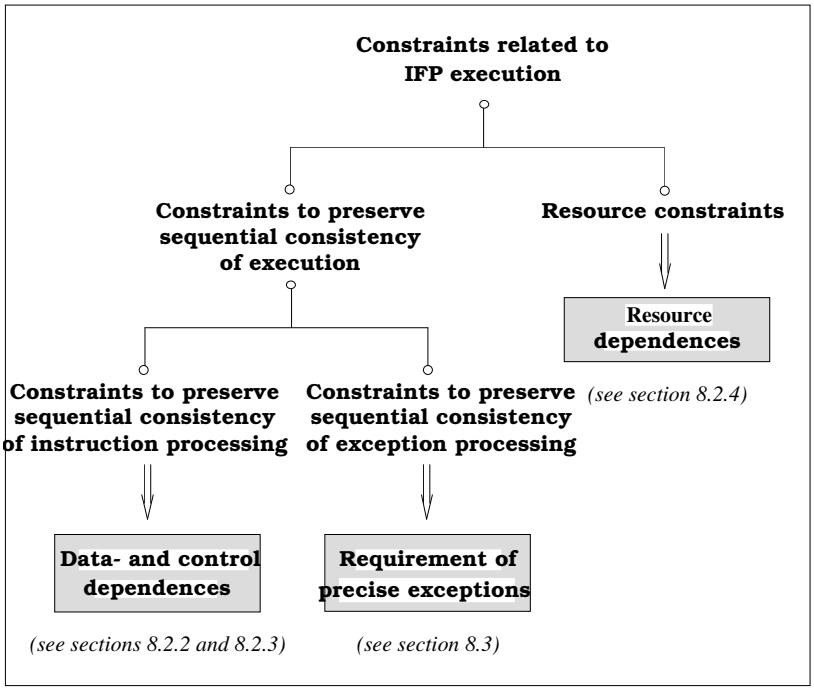
Distinction between VLIW and superscalar processors



The *VLIW approach* obviously needs *very long instruction words* in order to specify what each EU should do. The necessary length of a VLIW instruction is n-times the length of a traditional RISC instruction word length, if n EUs are presumed. Thus, for VLIW processors usually embodying of the order of ten execution units, the required word lengths are of the order of hundreds of bits. For instance, the TRACE VLIW family can execute 7 to 28 instructions in parallel, depending on the number of modules incorporated (1 - 4), and has word lengths of 256 - 1024 bits (Colwell 1987).

On the other hand, *superscalar processors* receive a sequential stream of conventional instructions. It is then the task of the Decode and issue unit to issue, in each cycle*, multiple instructions* for the multiple EUs. Current superscalar processors issue usually two (for instance DEC Alpha 21064, Intel Pentium), three (e.g. i960, PowerPC 6O1) four (as the IBM RISC/6000 or PowerPC 604) or seldom more (e.g. 6 in case of the Power2) instructions.

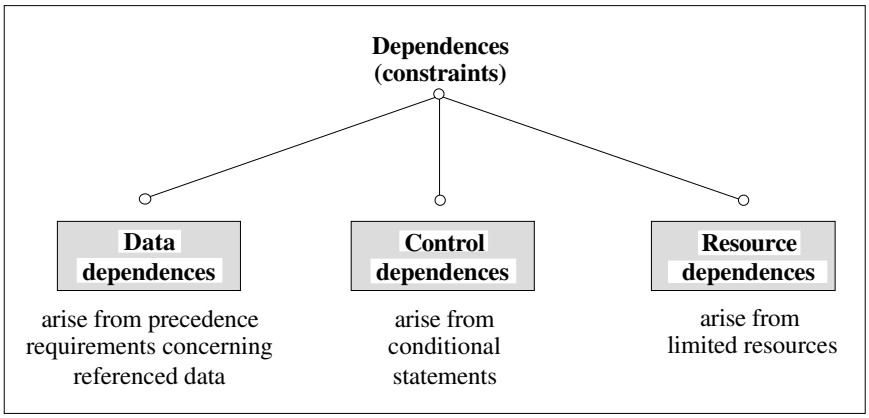
Constraints related to IFP execution



The *sequential consistency of the execution* is needed to guarantee the same results by parallel execution as would be produced during a traditional sequential run.

- The *resource constraints,* termed **resource dependences**, might also impose restrictions on instruction issue and processing. As an example, consider a processor having two fixed-point units and a single floating-point unit, all pipelined and capable of accepting instructions in each machine cycle. Then, evidently, not more than two fixed-point instructions and a single floating-point instruction can be issued in each cycle. Then, evidently, not more than two fixed-point instructions and a single floating-point instruction can be issued in each cycle.

Kinds of dependences

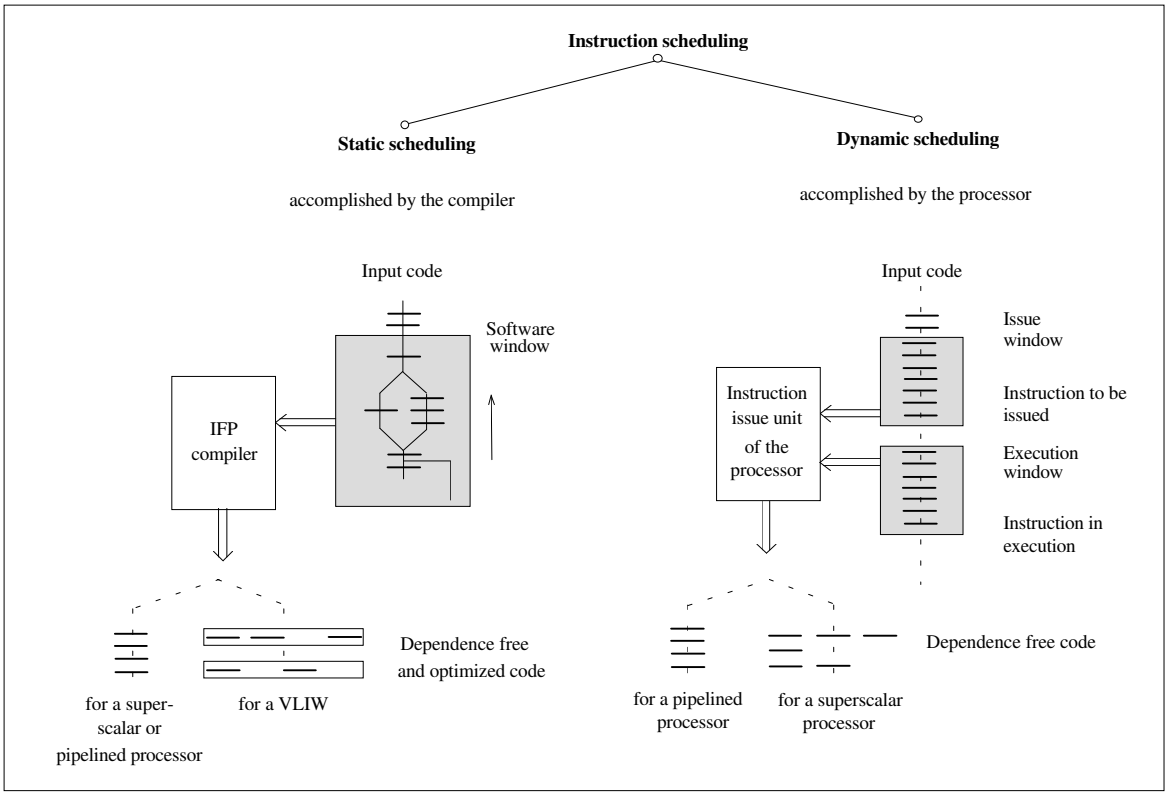


Data dependences are precedence requirements between instructions, needed to keep data, which are referenced by the instructions, consistent with those of a sequential execution. For instance, a precedence requirement is that a data value has to be produced sooner than it can be used.

Conditional instructions, such as conditional branches, also impose restrictions upon the sequence of instructions, which are referred to as **control dependences**. For example, in case of a conditional branch, the choice of which succeeding path will be taken, depends obviously, on the result of the evaluation of the referred condition.

The **resource dependences**, arises from the limited availability of hardware resources.

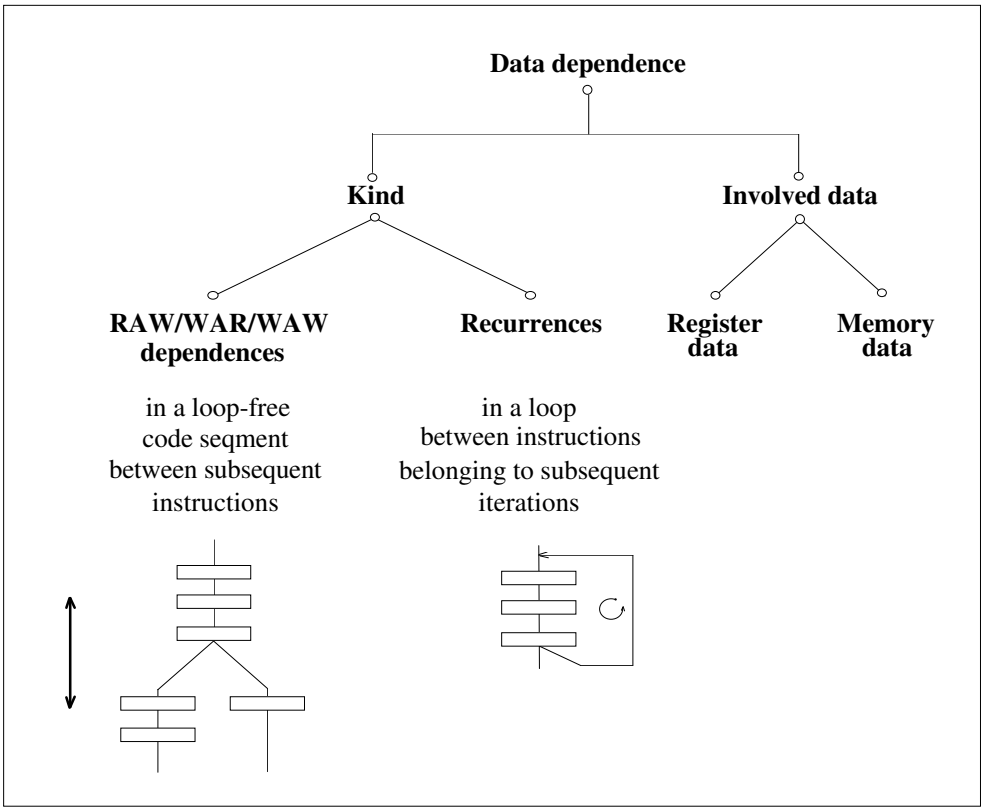
Basic approaches to instruction scheduling



During the *static scheduling* the compiler reorganises a code in order to avoid dependences, but beyond that, *it also* performs a *code optimisation* to expose as much parallelism to the architecture as possible.

In contrast, *dynamic scheduling* is performed by the architecture. For this reason, the processor maintains, typically, two gliding windows: the issue and execution window.

The main aspects concerning data dependences

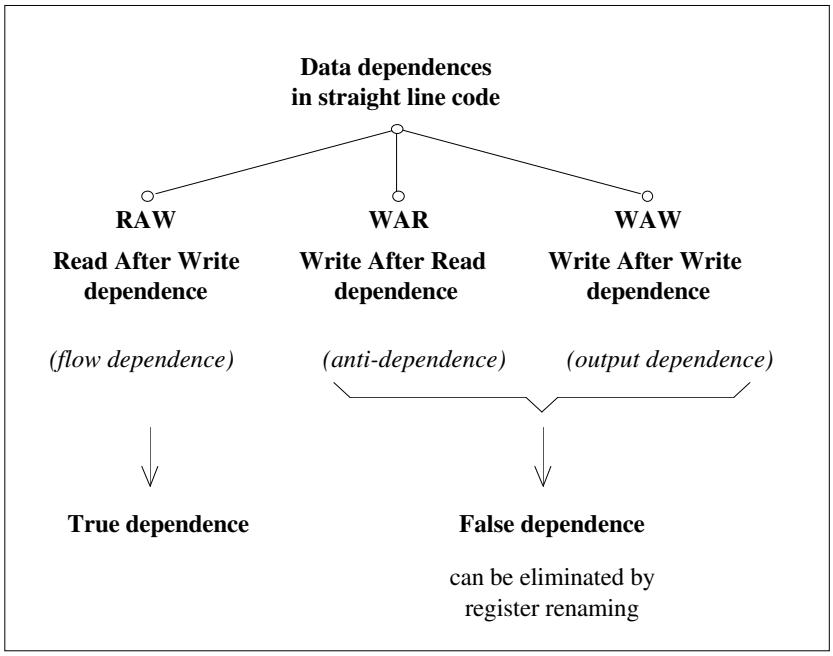


RAW-dependence (Read after Write dependence or flow dependence)

WAR-dependence (Write After Read dependence, anti-dependence)

WAW-dependence (Write After Write dependence, output dependence)

Terms relating to data dependences occurring in "loop-free code"

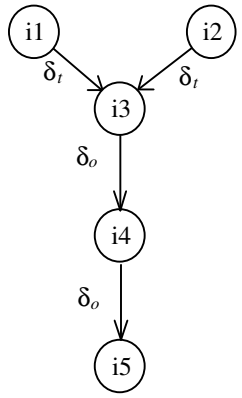


In general, an instruction i_l is said to be **RAW-dependent** on another instruction i_k, if i_l references as a source, a data value which will be produced by i_k. RAW-dependences are also called *flow dependences*. They are, at the same time, *true dependences*, since they cannot be abandoned. This is in sharp contrast to WAR- and WAW-dependences which can be circumvented by proper methods.

Data dependence graphs

Let us now consider a straight line code sequence, such as shown in Figure 5.2.7. Evidently, all the dependences among the instructions can be represented by a directed graph, as indicated in the same figure.

```
i1 : loadr1, a;
i2: loadr2, b;
i3: add r3, r1, r2;
i4: mul r1, r2, r4;
i5: div  r1, r2, r4;
```



Instruction interpretation: $r3 \leftarrow (r1) + (r2)$ etc.

Figure 5.2.7: A straight line assembly code sequence and its corresponding DDG

The graph has a node for each instruction considered. *Directed arcs* indicate *dependences* (precedence relations) between instruction pairs. In additional, as a clarification, we labelled each arc ###*t*, ###*a* or ###*o* for denoting true (RAW), anti (WAR) or output (WAW) dependence, respectively.

Control dependences

Conditional branches are also a major cause to impede the parallel execution of instructions. For instance, in the code sequence

```
sub r1, r2, r3;
jz    zproc;
mul r4, r1, r1;
    :
    :
zproc: load r1, x:
    :
```

the actual path of execution depends on the outcome of the subtraction. This means that the instructions following a conditional branch, are dependent on it. In a similar way, all conditional control instructions, such as conditional branches, calls, skips etc., cause dependences on the subsequent instructions which are referred to as **control dependences.**

How *frequent* are conditional branches?

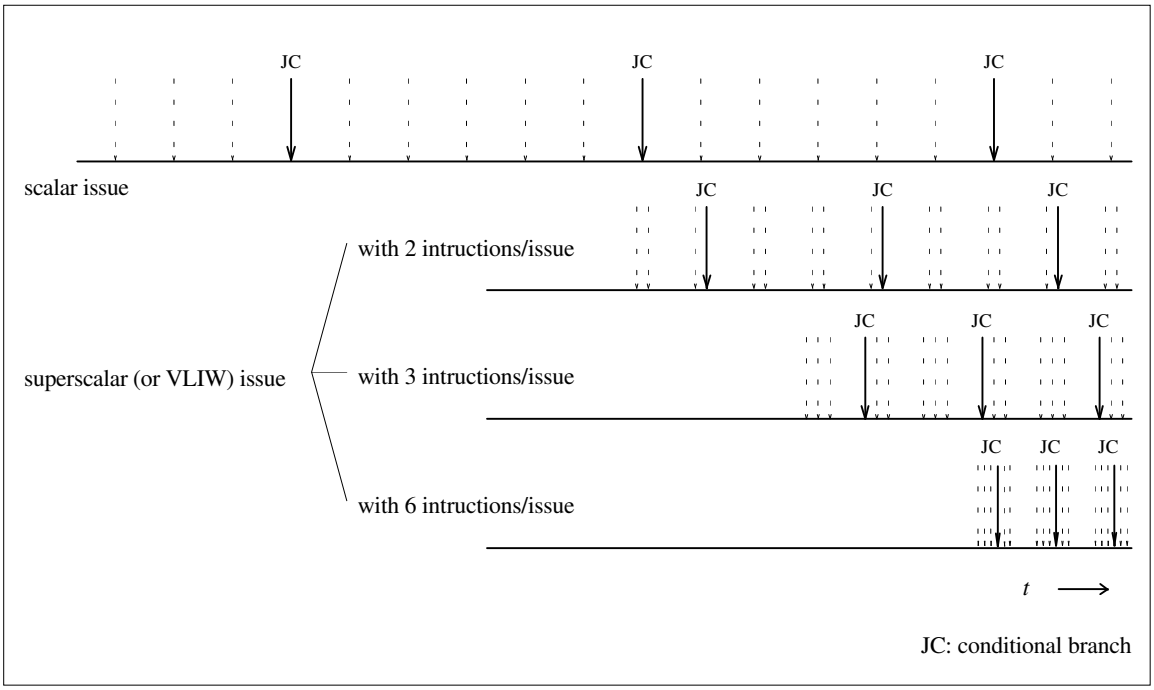
This is a point which deserves special attention in order to see how seriously conditional branches impede performance.

	Branch ratio ###		Uncond. total branch ratio ###		Uncond. branch ratio ###	
	gen. purp.	scientific	gen. purp.	scientific	gen. purp.	scientific
Yeh, Pat (1992) M88100	24	5	78	82	20	4
Stephens & al. (1991) RS/6000	22	11	83	85	18	9
Lee, Smith (1984) IBM/370	29	10	73	73	21	8
PDP-11-70	39		46		18	
CDC 6400		8		53		4

Published branch statistics

The compiled data indicate that *general purpose code* have a very high percentage of *branches,* which can amount to *20 - 30 %.* By contrast, in *scientific floating point programs* we can expect much fewer branches, the probable frequency is as low *as 5 - 10 %.* The ratio of unconditional branches and the total number of branches seems to be quite stable in different programs, remaining within the range of 75 - 85 %. As a consequence, the expected frequency of unconditional branches in general purpose code is about 20 %, whereas in scientific programs it is merely 5 -10 %. This means that as far as IFP processors are concerned, control dependences constitute a more serious hurdle in speeding up a general purpose code than a scientific one. A further consequence is that in considering conditional branches, it makes more sense to distinguish between general purpose and scientific code than discussing average values.

Impact of increasing the instruction issue rate on the frequency of conditional



Frequent conditional branches, however, impose a *heavy performance constraint* on IFP architectures. This is due to the fact that IFP processors can boost performance mainly by executing more and more instructions in parallel. Increased parallel execution, on the other hand, forces multiple instruction issue, in order to feed multiple execution units properly. However, the more instructions will be issued, on the average, in each cycle, the higher the probability is to encounter a conditional branch, that is a control dependence, during an instruction issue.

Control dependence graph

Like data dependences, control dependences can also be represented by directed graphs.Although, there are several slightly differing interpretations and designations, basically, in order to represent control dependences, nodes with two successors are introduced. The outgoing arcs now represent the true (T) and the false (F) path, and are usually labelled accordingly, with T and F.

Nodes with only one outgoing arc, are interpreted either as operational instructions or as a sequence of conditional branch free operating instructions (straight-line code). The usual designation of directed graphs expressing control dependences is **Control Dependence Graph (CDG).**

```
i0:   r1 = op1;
i1:   r2 = op2;
i2:   r3 = op3;
i3:   if (r2 > r1)
i4:     { if (r3 > r1)
i5:         r4 = r3;
i6:       else r4 = r1}
i7:     else r4 = r2;
i8:         r5 = r4 * r4
```

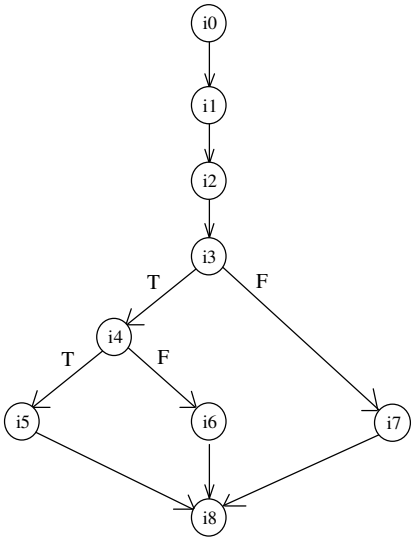


Figure 5.2.11: An example for a CDG

Resource dependences

An instruction is **resource dependent** on any previously issued instruction if it requires a hardware resource, in most cases an execution unit, which is in use by a previously issued instruction. If, for instance, there is only a single multiplication unit available, then in the code sequence

```
mul r1, r2, r3:
mul r4, r2, r5;
```

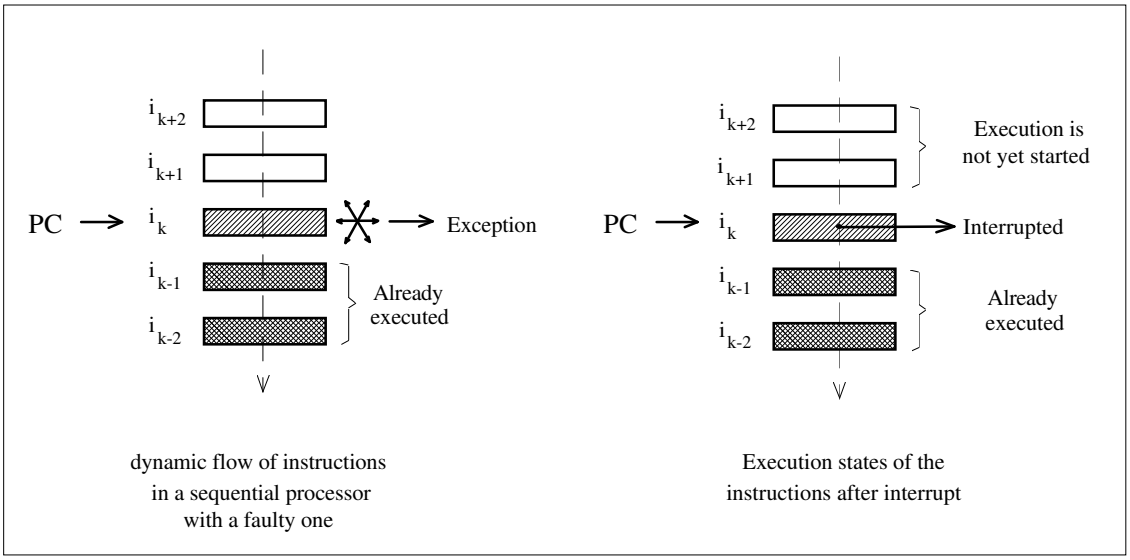
the second multiplication instruction is resource dependent on the first and can not be executed in parallel with the first one. By providing a large enough number of high performance execution units resource dependences usually do not reduce seriously the performance of up-to-date IFP architectures.

Processor type	Year of appearance	Number of execution units
Motorola 88110	1991	8
DEC Alpha 21064	1992	2
Intel Pentium	1993	3
PowerPC 601	1993	3
PowerPC 604	1994	5

Number of execution units provided by recent superscalar processors

Exception handling of a sequential processor

If a particular instruction causes an exception during its execution, and as a consequence, some time later the execution will be interrupted in order to process the exception, the state of an IFP processor by its interruption, must be the same as a sequential processor would have. Practically, this requirement can be satisfied by an IFP processor by delaying the acceptance of an exception until all instructions preceding the faulty one have been executed, and by preventing all subsequent ones from execution.



How much speed-up potential has functional parallelism ?

The next table summarises a number of the available results concerning the potential speed-up of programs when both, true data and control dependences restrict parallel instruction execution. The published data are divided into two groups: general purpose and scientific programs. Compilers, operating systems and non numeric application programs, are regarded as **general purpose programs** whereas **scientific programs** are dedicated ones containing massive floating point computations usually on vectors, or matrices.

Paper	Benchmark	Speed-up for general programs		Speed-up for scientific programs	
		range	average	range	average
Tjaden, Flynn 1970	31 Library programs			1.2 - 3.2	**1.9**
Kuck & at. 1972	20 FORTRAN programs			1.2 - 17	**### 4**
Riseman, Foster 1972	7 FORTRAN/ assembly programs	1.2 - 3.0	**1.8**	1.4 - 1.6	**1.6**
Jouppy 1989	8 Modula-2 programs	1.6 - 2.2	**1.9**	2.4 - 3.3	**2.8**
Lam, Wilson 1992	6 SPECmarks + 4 others	1.5 - 2.8	**2.1**	2 - 293	

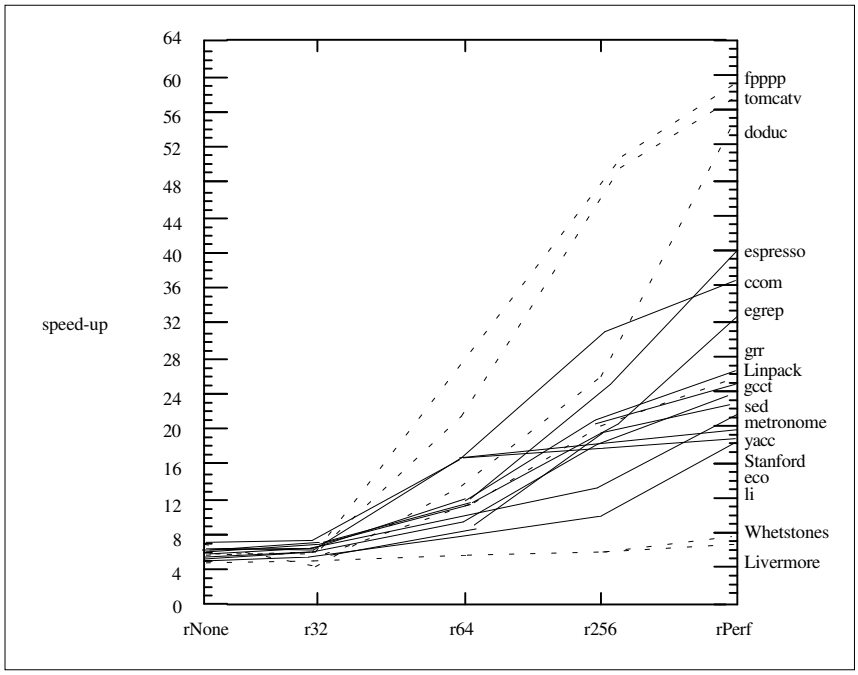
The figures show that, subject to the assumptions made, *general purpose programs* which embody predominantly irregular parallelism, have only a *very low amount* of *available parallelism*. Most investigations have revealed an astonishingly low potential speed-up in the range of *1.2 - 3.0* with an *average* value of *about 2*. The corresponding speed-up values for scientific programs are a little higher being in the range of 1.2 -17 with an average of about 2 - 4.

Spead-up: handling of control depenences

Paper	Benchmark	Speed-up for general purpose programs		Speed-up for scientific programs	
		range	average	range	average
Riseman, Foster 1972	7 FORTRAN/ assembly programs	8 - 100	**42**	30 - 120	**75**
Nicolaus, Fisher 1984	22 numerical programs			3 - 988	**12**
Kumar 1988	4 FORTRAN programs			475 - 3500	**1839**
Butler & et 1991	9 SPEC benchmarks	38 - 200	**170**	17 - 1165	**509**
Wall 1991	6SPECbenchmarks, 13 others	16 -41 7 - 37	**28** **24**	57 - 60 6 - 27	**59** **18**
Austin, Sohi 1992	SPEC benchmarks	13 - 942	**288**	51 - 33749	**6188**
Lam, Wilson 1992	6SPEC benchmarks, 4 others	174 - 3283 47 - 265	**1400** **229**	844 - 188 470	**64500**

These data indicate that *control dependences* are the *real obstacle* in utilising instruction level parallelism. If we were able to make perfect oracles, and could assume the availability of sufficient resources of hardware, we could expect speed-up values of the order of 10 to 100, in some cases even higher. In practice, of course, no oracles can be made, and these are no unlimited resources available.

Spead-up: effect of register renaming I.



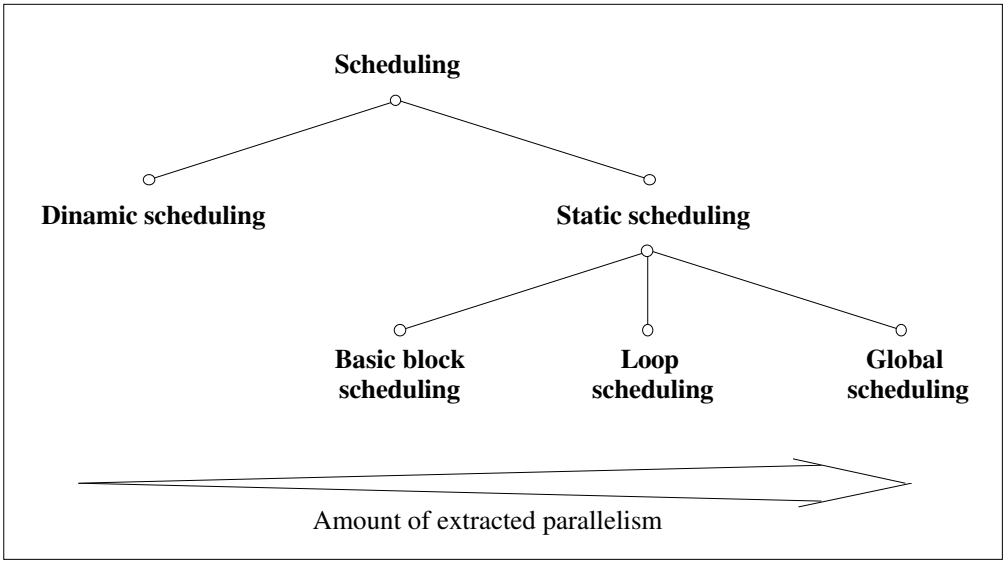
The achievable speed-up figures assuming either no elimination, imperfect elimination using 32, 64 or 256 renaming registers, or a perfect elimination assuming an unlimited number of available registers. The results show that *false dependence elimination* has, for the majority of benchmarks, a *considerable impact* on increasing speed-up. Therefore, an advanced IFP execution is expected to remove false dependences in one or another form. However, in order to achieve higher values of gain, at least 64 renaming registers appear to be required.

Spead-up: effect of register renaming II.

The characteristics of the test programs used in transparens I.

	Lines	Dynamic instructions	Remarks
Livermore	268	22294030	Livermore loops 1-14
Whetstones	462	24479634	Floating-point
Linpack	814	174883597	Linear algebra (3)
Stanford	1019	20759516	Hennessy's suite (5)
sed	1751	1447717	Stream editor
egrep	844	13910586	File search
yacc	1856	30948883	Compiler-compiler
metronome	4287	70235508	Timing verifier
grr	5883	142980475	PCB router
eco	2721	26702439	Recursive tree comparison
ccom	10142	18456797	C compiler front end
gcc1	83000	22745232	pass 1 of GNU C compiler
espresso	12000	135317102	boolean function minimizer
li	7000	1247190509	Lisp interpreter
fpppp	2600	244124171	quantum chemistry
doduc	5200	284697827	hidrocode simulation
tomcatv	180	1986257545	mesh generation

Parallelism extraction

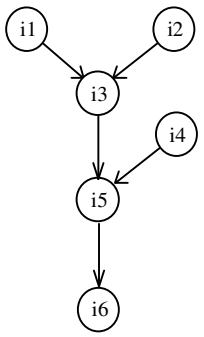


The *dynamic scheduling* comprises only a dependence check as well as the issue of independent instructions. On the other hand, *static scheduling* includes both dependence check as well as code optimisation. Therefore, we can expect static scheduling to be more effective than dynamic scheduling, as far as parallelism extraction is concerned.

However, static scheduling and, consequently, static parallelism extraction may be performed at three different levels: at basic block-, loop- and global level. The associated scheduling and the embodied parallelism extraction techniques are termed accordingly, as *basic block-* (or local), *loop-* and *global techniques*. These techniques have an increasing performance in the listed order.

An example of basic block and its length

```
i1:   calc:   add r3,r1,r2;
i2:             sub r4,r1,r2;
i3:             mul r5,r3,r4;
i4:             mul r7,r6,r6;
i5:             sub r8,r7,r5;
i6:             jn   negproc;
```



An assembly level basic block and its graphical representation by a DDG

	Program	Basic Block Length	
		General purpose programs	Scientific programs
others	awk	7.8	
	ccom	7.5	
	irsim	7.7	
	latex	10.4	
specmarks	eqntott	4.4	
	espresso	7.0	
	qcc (cc1)	8.9	
	matrix 3000		21.0
	spice 2g6		14.1
	tomcatv		59.8
	average	**7.8**	**31.6**

Basic block lengths of 6 SPEC benchmark and 4 other programs, measured on a MIPS 3000

The loop level and the global level extraction

The next possible level of parallelism extraction is *loop level extraction*. Here, instructions belonging to consecutive iterations of a loop can usually be executed in parallel resulting in considerable speed-up. However, recurrences may, to a certain extent, impede achievable speed-up.

It is widely accepted that loops are a significant source of parallelism, first of all in scientific programs. Therefore, it is apparent, that highly parallel ILP architectures, such as VLIW processors, should extract parallelism at least at the loop level. Accordingly, a large number of techniques have been developed to perform scheduling at this level.

The most effective way to extract parallelism is to do it at the highest possible level, using *global scheduling techniques*. Here, parallelism is searched for and extracted beyond basic blocks and simple loops, in such constructs as compound program sections involving loops as well as conditional control constructs. Evidently, in case of a chunk of code larger than a basic block, there are much more possibilities for parallelisation. However, the price to pay is increased complexity.