Trends in programmable instruction-set processor architectures 1

A *computer* is a system which processes data according to a specified algorithm. It contains one or more programmable (in the sense that the user can specify its operation) digital processors, also called *central processing units* (CPUs), memory for storing data and instructions, and input and output devices[1]. The science which researches the design of those systems is called *computer architecture*. Although the *architecture* of a computer system is usually defined as a specification of its functional appearance [5], i.e. its external behavior, the science of computer architecture has to take into account the characteristics of the implementation and realization of computer systems, the operating system and even the methods available for generating efficient code. The latter is necessary because human specified algorithms have to be translated (*compiled*) into code which is 'understandable' by the system. With the advances of VLSI technology one or more CPUs can be realized on a single chip; these single chip processors are called *microprocessors*.

To appreciate the importance of multi-media processor architectures, it is necessary to understand the trends in computer architecture and the developments which led to those trends. The trends are largely influenced by the aim of achieving a higher performance, or a better performance-cost ratio. The performance of a computer is dependent on the complete mapping trajectory, from application to operations performed by the data path. Therefore, to understand the architecture developments, we will look at compiler and VLSI developments as well. This chapter discusses past developments and the most important trends.

To begin with, section 1.1 looks at the role of computer architecture in the mapping trajectory of applications to hardware. It introduces some terminology and shows that architectures can be considered at different semantic levels. Section 1.2 gives an overview of the performance increase of computer systems during the latest years. The primary developments contributing to the performance increase are described in the following sections: VLSI developments are detailed in section 1.3, architecture developments in sections 1.4, and compiler developments in section 1.7. Meanwhile, section 1.5 presents a classification model for

[1]Analog processors can also be programmable, and could fit into this definition of a computer. However, we will restrict ourselves to digital processors.

computer architectures, based on the described architecture developments, while the relation between architectures and applications is discussed in section 1.6. Finally section 1.8 summarizes this chapter and evaluates several aspects of the current trends in computer architecture.

1.1 Bridging the semantic gap

As stated in the introduction of this chapter, a computer contains one or more CPUs. A CPU is capable of performing operations on data. These data must reside in the addressable memory of the computer in order to be (directly) operable by the CPU. Data may also be stored on input/output (I/O) storage devices or come from the external world, using a keyboard, or sensors via I/O interfaces. In the latter cases data have to be brought into memory first (e.g. by an intelligent I/O interface unit), or alternatively, the address spaces of these I/O devices must be mapped onto the memory address space.

The operations which transform the data have to be specified in a language which is directly 'understandable' by the processor. This language, $L_{architecture}$, is also called the *machine language*. A program written in this language is composed of instructions taken from the instruction set, or I_set, of the processor. I_set is defined as the set of all instructions which can be executed by the processor. Each instruction can specify one or more operations. An instruction can be considered as the most elementary or atomic unit of execution.

A computer can now be defined as a system CS, executing a program written in $L_{architecture}$ and residing in program memory, which transforms data as a consequence of executing instructions of this program. Formally, we can view CS as the following transformation function:

$$CS : (L_{architecture}, Dmem, ID) \rightarrow (Dmem, OD) \qquad (1.1)$$

where $Dmem$ represents the computer data memory, and ID and OD the memory mapped input and output devices respectively[2].

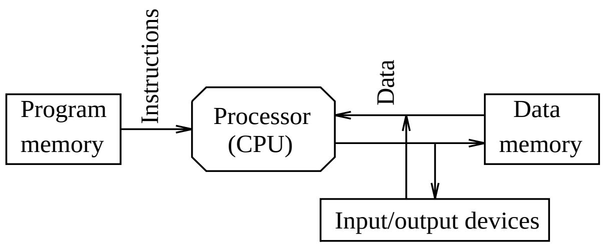

Figure 1.1: Basic computer system.

Figure 1.1 pictures a simplified computer system consisting of four main components: the CPU, the program memory, the data memory, and the input/output devices. We directly

[2]The data in $Dmem$ is considered *state* of the computer system; it is part of the state of the active processes. This state is not included in the *processor state*, however.

recognize three data streams: incoming program instructions, incoming data and outgoing data. One of the well known performance enhancements is to have separate physical streams for at least instructions and data (at the first level of the memory hierarchy: the cache), resulting in a *harvard architecture*. Low performance systems may combine these streams and contain a shared (first level) memory for data and programs.

In general $L_{architecture}$ is not equal to the programming language of the user. Mostly, the user specifies his application in a language having a much higher semantic level, a so-called *high level language*, or HLL. The higher semantic level of an HLL allows the user to specify his application in fewer lines of code and in, for the user, a much more readable format. On the other hand, the primitive hardware operations supported by the hardware data path, generally have a lower semantic level than $L_{architecture}$. We therefore distinguish three programming domains, each with their corresponding languages:

1. The *application domain* with language $L_{application}$, into which the programmer specifies the application algorithm. With increasing semantic level we mention as examples: assembler, Fortran, C, Lisp, Algol-68, C++, HPF (high performance Fortran), and Miranda.

2. The *architecture domain* with language $L_{architecture}$. At this level the CPU is considered as a box containing *internal processor state* PS, like the values of the internal registers, and executing instructions from I_set. The execution of an instruction can transform state PS, or result in input or output of data. Formally:

$$CPU : (I\_set, PS, Idata) \rightarrow (PS, Odata) \tag{1.2}$$

   where $Idata$ is the incoming and $Odata$ the outgoing data domain. Note the difference between the definition of CS and CPU. The former executes programs while the latter executes instructions. The *architecture* of a CPU describes its internal state PS and all its instructions $Ins \in I\_set$, where an individual Ins can also be interpreted as performing the following transformation:

$$Ins : (PS, Idata) \rightarrow (PS, Odata) \tag{1.3}$$

3. The *data path domain* of the CPU with language $L_{datapath}$. Examples of operations possibly supported by the CPU data path are: logic operations, integer operations, floating point operations, multi-media operations, and data moves between registers or latches. Sometimes these operations are called *micro-operations*.

Example 1.1 illustrates the above three programming domains by showing how a typical instruction from $L_{application}$ can be translated into the two other languages.

Example 1.1 *Three different programming domains.*

The addition of two numbers can be specified in the three different programming domains as follows (comments start with two slashes '//'):

$L_{application}$:

```
A := B + C
```

$L_{architecture}$:

```
LD  r1,M(&B)      // load register r1 (r2) with data from
LD  r2,M(&C)      // memory location at address B (C)
ADD r1,r1,r2      // perform the actual addition
ST  r1,M(&A)      // store result at memory address A
```

$L_{datapath}$:

```
&B  → MAR             // put address in memory address register
MDR → r1              // load data from memory data register
&C  → MAR
MDR → r2
r1  → ALU_input-1     // transport operands to the adder
r2  → ALU_input-2
ALU_output := ALU_input-1 + ALU_input-2
ALU_output → r1
r1  → MDR             // store result at
&A  → MAR             // memory address A
```

The data path domain has the highest level of detail; it is closest to the actual hardware. Note that in reality many more actions have to take place at the data path level; e.g., not shown are the sequencing of the program counter, the checking for interrupts, and the actions necessary for instruction fetch.

History has shown many solutions for bridging the semantic gap between $L_{application}$ and $L_{datapath}$. Figure 1.2 shows four of those solutions for fixed semantic levels of application and data path. The semantic level of $L_{architecture}$ differs for each solution. Extreme cases are the direct execution architectures [31], where $L_{architecture}$ is almost equal to $L_{application}$, and the microcoded architectures, i.e., architectures intended to be programmed in microcode by the user, where $L_{architecture}$ is almost equal to $L_{datapath}$. In between are the more realistic *complex instruction set computer* (CISC) and *reduced instruction set computer* (RISC) architectures. They are described next.

In the seventies and the early eighties many architectures were of the CISC type. They had a large and complex instruction set, resulting in an $L_{architecture}$ with a high semantic level. Many operations and operand specifications used in HLLs could be directly (one-to-one) mapped onto corresponding instructions and addressing modes supported by the architecture. Examples are `CALL` instructions, which perform a complete stackframe setup, and complex addressing modes for array and structured data access. Instruction formats could be very long and complex; e.g., a typical CISC, the VAX-11/780, supported instructions with a size

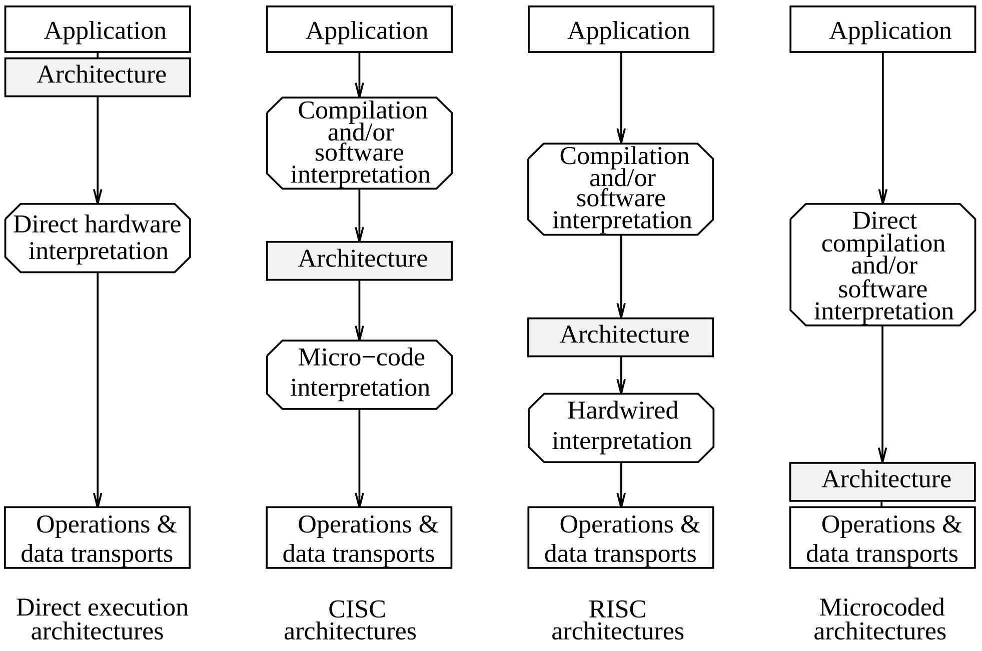
Application
Architecture
Direct hardware interpretation
Operations & data transports
Direct execution architectures
Application
Compilation and/or software interpretation
Architecture
Micro−code interpretation
Operations & data transports
CISC architectures
Application
Compilation and/or software interpretation
Architecture
Hardwired interpretation
Operations & data transports
RISC architectures
Application
Direct compilation and/or software interpretation
Architecture
Operations & data transports
Microcoded architectures

Figure 1.2: Bridging the semantic gap.

ranging from 16 to 456 bits [15]. Most instructions supported memory-to-memory addressing modes, or were stack based; it was considered difficult to use internal registers efficiently.

To support this high semantic level, CISC processors applied one or two levels of interpretation of instructions using so-called *microcode* and *nanocode* [46]. Each (complex) instruction from $L_{architecture}$ translates into many micro or nano-instructions from $L_{datapath}$, and therefore consumes many clock cycles. Micro and nano-instruction storage was considered cheap and fast in those days, and easily extensible. This led several researchers to the idea of further raising the level of $L_{architecture}$ and make it almost equal to $L_{application}$. This research culminated into Pascal processors [22], for example, which could directly execute an intermediate language, called P-code, which had a semantic level not far below the level of Pascal. Another example is the NOVIX-4000 processor which directly executes instructions from the Forth language. Many of these approaches never left the academic environment, mainly because they were inflexible with respect to the support of different HLLs.

Some CISC processors allowed the user to load his own microcode; they had a so called *writable control store* (WCS). This feature had two applications:

1. Supporting different architectures by changing the instruction set and therefore $L_{architecture}$.
2. Compiling a program directly into microcode. This usage corresponds to the microcoded architectures as shown in figure 1.2.

Both applications did not become very popular for several reasons:

- Micro instructions had a very complex format, supporting a high level of concurrency and having a low semantic level.
- Writing correct microcode was error prone. One had to know peculiar details of the processor in order to write correct code.
- At that time the compiler technology was not capable of exploiting the advantages of translating HLL code directly into a language with a very low semantic level.
- When programs did not fit into the WCS one had to use expensive overlay techniques. Timesharing resulted also in much time spent swapping the WCS.
- Exceptions caused problems. What to do if an operand does not reside in main memory (page fault exception in a virtual memory system)? Support of this exception requires that all micro state is accessible and restorable.

In the early eighties RISCs were introduced. They lower the semantic level of the architecture by reducing the number of different instructions; especially complex instructions and addressing modes are not supported by RISCs. This facilitates pipelining of the instruction set and allows for the replacement of the micro and nanocode interpretation of CISCs by the more efficient hardwired interpretation.

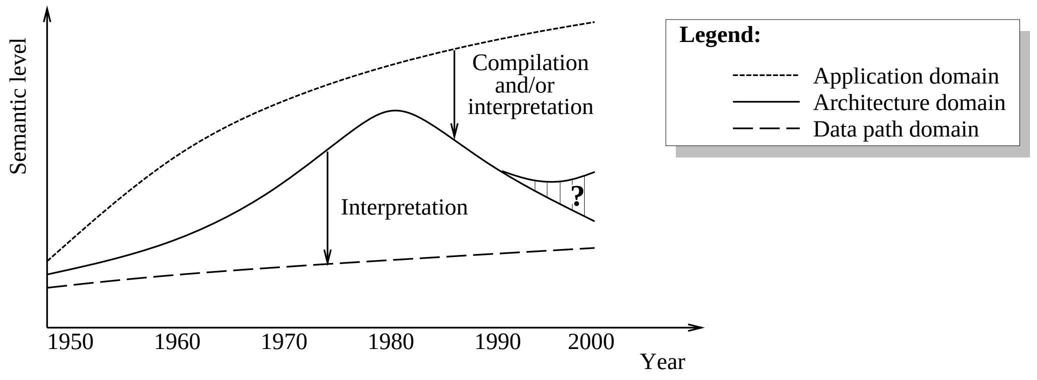

Figure 1.3: Increasing role of the compiler.

As will be clear from the foregoing discussion, the semantic level of the architecture domain did change in the course of history. The same holds for the application and data path domains. Figure 1.3 indicates these changes.

The application level gradually increased, and is still increasing. For example, many programmers are making a move from imperative to object oriented programming. C++, an extension of the imperative C language to which object oriented features have been added, will become one of the major languages of choice[3].

[3]The C++ descendant Java may even become more popular, because of its processor platform independence and its use within world-wide-web (WWW) applications [10, 33]. Java can be viewed as 'C++ minus C'; it is a pure object-oriented language.

The hardware level also increased. For example, while in the past floating point (FP) support was a coprocessor option, it is now common for every microprocessor to have this support built-in. This is strange in the sense that for many applications the FP operations contribute little to the total processing time. However, (1) marketing requires built-in FP support, (2) current VLSI technology offers the necessary transistor densities to integrate FP support together with the other CPU functionality on a single chip, and (3) it is relatively easy to exploit concurrency between integer and FP operations because they hardly have any conflicting resource demands. The question remains, does adding FP support, instead of using the available VLSI area for other purposes, lead to the best cost-performance ratio? In the future we will experience further raising of the hardware level. We observe this even now through the inclusion of multi-media support operations [24]; Intel has already set a standard within the PC world with its MMX multi-media instruction set.

The most remarkable feature about figure 1.3 is the development of the architectural level, and the corresponding changing application-architecture gap which has to be bridged by software, using a compiler or interpreter. While in the fifties and sixties this gap rapidly increased as a consequence of HLL development, it decreased in the seventies through the usage of CISCs, and started to increase again with the introduction of RISCs. The question remains, what will happen to the architectural level in the coming years? It may increase again (like in Java processors [45] which directly execute Java byte-code) or decrease almost to the data path level, as will be the case in the proposed transport triggered architectures. The next sections discuss why the architectural level did change and how it may change during the coming years. This will be done from a performance perspective, because architecture developments are strongly performance driven.

1.2 Performance of computer systems

When a computer user talks about performance, what is meant is the real time it takes to accomplish a certain task or application; this time is also called *elapsed* or *wall clock time*. The elapsed time includes (1) the *user time*, the time the system executes instructions specified by the application (i.e., when the system executes in *user mode*), (2) the *system time*, needed to handle operating system calls as requested by the application (e.g. I/O requests), and (3) the time for swapping and executing other processes. The latter is a consequence of time-slicing, which is used by multi-tasking operating systems. For now we are mainly concerned with decreasing the user time.

In order to estimate the performance of a computer, standardized benchmarks like Dhrystone, or better, a set of benchmarks, like SPECint and SPECfp, are used. The performance of several well known general purpose microprocessors is listed in table 1.1[4]. Shown are the year of introduction, the processor clock frequency at that time, the ratings for SPECint92 and

[4]Data listed in tables 1.1 and 1.2 have been taken from various sources like the Newsnet and several WWW-sites.

Processor	Year	Freq. (*MHz*)	SPECint92	SPECfp92	Issue rate	Pipelining[a]
Intel 8086	1978	5	∼0.2	∼0.1	1	-
Intel 286	1982	6	∼1.0	∼0.5	1	-
Intel 386	1986	16	3.1	1.6	1	-
Intel 486	1989	25	15	∼7	1	5
Intel Pentium P5	1993	66	67.4	63.6	2	5
Intel Pentium P54C	1994	100	100	80.6	2	5
Intel Pentium P55C	1995	155	∼155	∼125	2	5
Intel Pentium Pro	1995	150	245	220	3	14
M68040	1989	25	21	15	1	3
Sparc micro	1992	50	26	21	1	5
Sparc super	1992	40	53	65	3	4
Sparc super2	1995	90	135	147	3	4
Sparc ultra	1995	167	275	305	4	9
Mips 3000	1989	33	18	19	1	5
Mips 4000	1992	100	59	61	1	8
Mips 4600	1994	150	110	83	1	5
Mips 10000	1996	200	∼300	∼600	5	5
HP 7000	1990	66	48	75	1	5
HP 7100	1992	99	80	151	2	5
HP 7200	1994	140	∼150	∼250	2	5
HP 8000	1996	180	>400	>600	4	7
Alpha 21064	1992	200	133	200	2	7
Alpha 21164	1994	300	330	500	4	7
Alpha 21264	1997	∼500	∼1100	∼1900	4	7
MPC 601a	1993	50	40	60	3	4
MPC 601b	1994	100	105	125	3	4
MPC 604	1994	100	160	165	4	4
MPC 620	1995	130	225	300	5	4

[a]Pipelining degree of integer operations.

Table 1.1: Performance improvements.

SPECfp92[5], the maximum issue rate, and the pipelining degree of integer operations[6]. The *issue rate* is the number of instructions which can be issued (started) per cycle; the *pipelining degree* is equal to the number of cycles needed to execute an instruction; they are discussed in section 1.4. SPECfp92 measures floating point performance, while SPECint92 figures are based on integer type work-loads. All benchmark figures are based on the required user time. The figures in this table with a prefix '∼' are estimated (e.g. by using ratings from similar CPUs at other frequencies). Note that these figures do not only depend on the CPU internals, but on external (non-CPU) factors as well, such as the quality of the compiler, and the amount of external (first or higher) level caching. They therefore have to be viewed as performance indicators and not as absolute, fixed performance values.

When we fit the SPECrate figures into a curve (using least-squares curve fitting), as shown in

[5]Currently, processors are benchmarked with the SPEC95 benchmark suite; depending on the architecture and compiler, SPEC95 numbers are about a factor of 35 lower.

[6]The figures count all integer pipelining stages, including instruction fetch, decode and write-back.

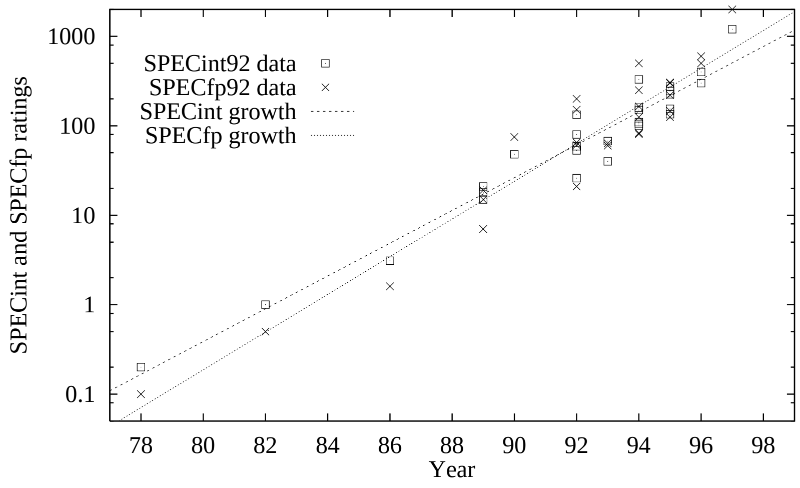

Figure 1.4: Microprocessor SPEC92 ratings.

figure 1.4, we observe an annual performance improvement of more than 50% for SPECint, and more than 60% for SPECfp. How is such a tremendous increase possible? In order to answer this question we have to look at the factors determining the user time of an application. This time is calculated from

$$t_{user} = N_{instr} \times \mathit{CPI} \times t_{cycle} \tag{1.4}$$

where N_{instr} is the number of instructions executed in user mode (also called the *dynamic* number of instructions), *CPI* is the average number of cycles per instruction, and t_{cycle} is the cycle time. In order to increase the performance we have to decrease the factors contributing to the user time: N_{instr}, *CPI* and/or t_{cycle}. There are three main developments which influence these factors:

1. The improvement of VLSI technology, decreasing t_{cycle}, and increasing the available number of transistors per chip.
2. Architecture developments like pipelining instructions and offering instruction level parallelism, influencing t_{cycle}, N_{instr} and *CPI*. The last two columns of table 1.1, which list the issue rate and pipelining degree, give an indication of these developments.
3. Compiler developments, especially the exploitation of instruction level parallelism, which influences N_{instr} and *CPI* as well.

The above developments are strongly related. VLSI improvements offers the possibility to put much more hardware on a single chip, allowing the implementation of multiple *function units* (FUs) on one chip. However, these units only make sense when this potential hardware performance can be exploited by compilers. Compilers need therefore to be extended with so-called *code schedulers*, which look for independent operations which can be executed in parallel. The sections 1.3 to 1.7 detail these developments.

1.3 VLSI developments

There has been a great technology improvement since the first electronic (tube) computer system, the ENIAC (electronic numerical integrator and calculator), came into life in the year 1946 [27]. The ENIAC, occupying a floor area of 140 m^2, could perform a multiplication of two 10-digit numbers in 2 ms. Currently this can easily be done in less than 10 ns, a speed increase of more than 5 orders of magnitude, or an average of about 30% per year. This increase is largely due to improvements of electronic switching devices, from tubes in the fifties, transistors in the sixties, to large scale integration of transistors starting in the seventies. The latter is especially important.

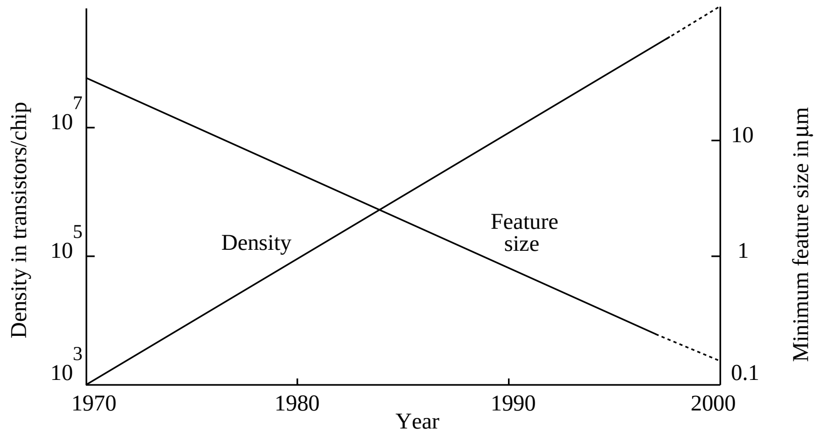

Figure 1.5: MOS/CMOS integration density and minimum feature size.

Figure 1.5 indicates the developments of integration density and minimal feature size during the last decades. The CMOS feature sizes scaled down far below the 1 μm level; it decreases about 20% per year. As a consequence, and also because chips get larger, the number of transistors per chip, $\#Trans$, shows an annual increase of more than 50%. For example, the integration level of dynamic RAM memory chips is currently (1997) well beyond 100 million transistors per chip, and quadruples each three years. Their number of transistors can be approximated by:

$$\#Trans \approx 2^{(year-1956)\times 2/3} \tag{1.5}$$

For CPUs this number is roughly one order lower; in 1997 it is in the order of ten million. Higher densities generally result in faster circuit realizations. This becomes clear when we look at the achievable cycle time, t_{cycle}, which is determined by the critical timing path of a circuit, and roughly estimated by:

$$t_{cycle} \approx t_{gate} \times \#gate\_levels + wiring\_delay + pad\_delay \tag{1.6}$$

Because the main circuitry of a processor, including first level cache memory, fits onto a single chip, the *pad_delay* contribution can be avoided[7]. Depending on how all dimensions of mask-layers scale with the minimal feature size *mfs*, the switching time of a gate, t_{gate}, reduces at least linearly with *mfs* [58]. Internal wiring delay forms still a minor contribution; however, its importance will likely increase in future designs [11]. With pipelining one can reduce the effective number of gate levels, $\#gate\_levels$. This is an architectural decision; we will delay its discussion until section 1.4 where the architectural developments are described.

Table 1.2 lists several realization characteristics of different microprocessor designs, at their year of introduction. Unknown values are marked '-'. From this table the following can be observed. State of the art processors will be implemented in a 0.35 μm technology with four or more metal layers used for interconnections. The area slightly increases; large chips have an area of about 3 cm^2. The number of transistors is in the ten million range. Note that the indicated transistor counts are largely determined by the total size of the internal caches; e.g., the DEC Alpha 21164 processor chip contains first level instruction and data caches of each 8 $kbytes$, and a second level shared cache of 96 $kbytes$. They require (probably) more than six million transistors, leaving around three million for the other parts of the processor.

The table also lists the power dissipation when operating at the indicated frequency. With switching frequencies beyond 100 *MHz* the power dissipation becomes very high; this issue becomes especially important for battery operated computers, and is therefore a major future research and design topic.

Two questions remain: how will VLSI development continue, and how will this affect computer architecture and design? The following remarks can be made:

- The feature size will certainly decrease further; however, below 0.1 μm electrons happen to tunnel (a quantum dynamics effect) through the isolation between on-chip wires.
- Chip area will slowly increase; its increase strongly depends on yield characteristics of foundries.
- Increased power dissipation forces the use of lower supply voltages (3.3 *V* gets accepted and VLSI processes with lower voltages become available), and perhaps heat controlled clock frequency.

It is yet unclear what will happen when physical limits have been reached. Alternatives to the CMOS technology, like the use of GaAs technology (although its integration density is far below CMOS), development of junction switches, or even the use of optical switching techniques, are being considered. Research is being performed in these areas.

In the next section we will look at architecture developments. The computer architect has to answer the question, what to do with the opportunities offered by VLSI; i.e., how to exploit those millions of transistors?

[7]An important exception forms the HP-PA processors of Hewlett-Packard, which still use off-chip first level caching. Its main advantage is that the first level cache can be much larger and therefore the cache miss-rate and *CPI* penalty much lower.

Even HP is going to change this; HP announced the PA-8500 processor, a 0.25 μm, 120 million transistor processor with 1.5 Mbyte on-chip cache.

Processor	**Year**	**Tech.**[a]	**Area** (mm^2)	**#Trans.** (k)	**Voltage** V	**Diss.**[b] (W)	**Freq.** (*MHz*)	**Cache**[c] (*kbyte*)
Intel 8086	1978	1 / -	-	29	5.0	-	5	0
Intel 286	1982	2 / -	-	134	5.0	-	6	0
Intel 386	1986	2 / -	-	275	5.0	∼1	16	0
Intel 486	1989	3 / -	99	1200	5.0	∼4	25	8
Intel Pentium P5	1993	3 / 0.8	296	3100	5.0	16	66	16
Intel Pentium P54C	1994	4 / 0.6	163	3100	5.0	16	100	16
Intel Pentium Pro	1995	4 / 0.6	306	5500	3.1	29	150	16[d]
M68040	1989	2 / -	126	1200	5.0	6	25	8
Sparc micro	1992	2 / 0.8	225	800	5.0	4	50	6
Sparc super	1992	3 / 0.7	256	3100	5.3	14	60	36
Sparc super2	1995	3 / 0.6	299	3100	-	16	90	36
Sparc ultra	1995	4 / 0.5	315	3800	-	∼30	95	32
Mips 4000	1992	3 / 1.0	213	1100	-	-	100	16
Mips 4200 (SGI)	1993	3[e] / 0.64	76	1300	3.3	1.8	80	24
Mips 4400	1993	3 / 0.6	186	2300	3.3	15	150	32
Mips 4600	1994	3 / 0.64	77	1850	3.3	4.6	150	32
Mips 10000	1996	4 / 0.5	298	5900	3.3	∼30	200	64
HP 7000	1990	- / 1.0	196	580	5.0	-	66	0
HP 7100	1992	3 / 0.8	202	850	5.0	23	99	0
HP 7200	1994	3 / 0.55	210	1260	4.4	30	140	0
HP 8000	1996	4 / 0.5	345	3900	-	>40	180	0
Alpha 21064	1992	3 / 0.75	234	1680	3.3	30	200	16
Alpha 21164	1994	4 / 0.5	299	9300	3.3	∼45	300	112[f]
Alpha 21264	1997	6 / 0.35	302	15200	2.0	60	500	128
MPC 601a	1993	4 / 0.65	121	2800	3.6	9.1	80	32
MPC 601b	1994	5 / 0.5	74	2800	3.3	5.6	100	32
MPC 604	1994	4 / 0.5	196	3600	3.3	13	100	32
MPC 620	1995	4 / 0.5	311	6900	3.3	30	130	64

[a]Two technology numbers are specified: the number of metal layers, and the minimal transistor gate length, measured in μm.

[b]Peak power dissipation.

[c]Total size of on-chip instruction and data caches.

[d]A second level cache of 256 kbyte is integrated as separate die on a multi-chip module.

[e]Uses 2 layers of polysilicon.

[f]Includes 96 kbyte second level on-chip cache.

Table 1.2: Technology improvements in microprocessor design.

1.4 Overview of architecture developments

This section gives an overview of the main architectural developments, taking the CISC approach of the seventies as point of departure, and the evolution of the VLSI technology as a driving force. The computer architect has to maximize the performance, or performance-cost ratio, through the optimal exploitation of VLSI capabilities, under the constraint of a usable interface to the compiler writer. The latter is important since it does not make sense to offer functionality which cannot efficiently be exploited by the compiler[8]. In order to improve the performance the processor can be changed such that all three terms of equation 1.4 reduce. To this purpose, the computer architect has three techniques at his disposal:

- **(Super)-pipelining:** pipelining reduces *CPI*; Superpipelining reduces t_{cycle}.
- **Powerful instructions:** performing more work per instruction, essentially making instructions more complex. This may reduce N_{instr}.
- **Multiple instruction issue:** issuing more instructions per cycle, possibly reducing the effective *CPI* below one.

Note that, while the second technique changes the architecture (because the instruction set changes), the other two techniques may be implemented such that their effect is invisible at the architectural level; in that case these techniques only change the processor *organization*. In line with the remarks made at the beginning of this chapter about the science of computer architecture, we will treat all three techniques as being applied by the computer architect. Below, sections 1.4.1 to 1.4.3 are each devoted to one of the above three techniques illustrating the resulting type of architecture. Section 1.4.4 summarizes the application of above techniques by giving instruction pipelining diagrams of several of the discussed architectures.

1.4.1 Pipelining and superpipelining

In order to execute an instruction from $L_{architecture}$ several steps occur: fetching the instruction from memory, decode it, get the required operands, execute the specified operation, and finally, write back the result of the operation. Together these steps form the well known *Von Neumann cycle*; it is called a cycle because these steps are repeated for every instruction.

In the seventies, CISC architectures were not pipelined; i.e., they performed these steps sequentially. In principle this would lead to a very long cycle time. In practice, these steps were microcoded, each micro instruction taking one clock cycle. As a result, the cycle time was kept short, but each instruction from $L_{architecture}$ took many cycles to execute, resulting in a large *CPI*.

The throughput of instructions increases, and *CPI* therefore decreases, if we are able to overlap or pipeline the execution of instructions. However, this requires a streamlined instruction set; this means that each instruction can be split into the same number of stages, whereby each stage takes about the same time and uses different hardware. This was not the case with the instruction set of CISCs. Instructions differed strongly in number of bits, the number and the type of operands, the used addressing modes, and the execution time of the required operations. As a result RISCs evolved; they have a reduced instruction set and support a very limited number of addressing modes, such that instructions fit well in a simple pipelining scheme. The pipeline stages correspond more or less to the micro-instructions of a CISC. In principle, RISCs can issue one instruction each cycle, giving a theoretical *CPI* of

[8]In the embedded system world manual writing of machine (assembly) code is still common practice; however, in the long run, when applications become larger and reuse of code is important, usage of HLLs becomes mandatory and a compilation trajectory will be required.

one. Figure 1.6 shows how instructions are executed in a possible pipelined fashion, assuming that operands are located within registers[9]. Although each instruction takes five clock cycles, every cycle a new instruction is started. In practice, due to hazards in the pipeline (see [25]), the *CPI* will be somewhere between one and two.

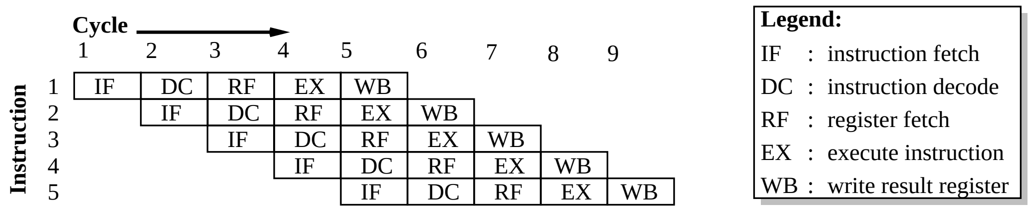

Figure 1.6: Pipelined execution of instructions.

The architect can go one step further and pipeline the actions of a micro-instruction, that means the stages of a RISC, as well. This is called *superpipelining*. Using superpipelining one reduces the number of gate levels, $\#gate\_levels$, in the critical path. In principle this number can be reduced to one, however extra latches cause time and area overhead, and also clock and data skew can raise this minimum (see e.g. [37] for a discussion about optimal pipelining in supercomputers, and [42] for a general discussion about the limitations of pipelining). There is some disagreement about when to call a microprocessor superpipelined. Some people call it that way if at least one of the traditional RISC pipeline stages is split into two or more smaller stages. Others restrict the word superpipelining to those processors which at least split the traditional ALU or execution stage. We will use the first convention. This means that one or more of the critical stages, like instruction fetch, data fetch or the execution stage is split. Consequently branch, load or ALU operations experience a longer delay[10]. For example, a two stage ALU results in one *delay slot* for ALU operations. Optimal hardware utilization requires these delay slots to be filled with independent operations. If two succeeding ALU operations are dependent on each other a *pipeline stall* should occur[11]; if this dependence is not detected by hardware and no stall cycle is inserted an incorrect result occurs.

As compared with pipelining, superpipelining is an extension of the pipelining concept; it just means a higher degree of pipelining as compared to the one used within RISC processors. However, while the result of RISC pipelining is usually interpreted as reducing the *CPI* close to one, superpipelining decreases t_{cycle}, and in fact leads to an increase of *CPI*. The latter is a consequence of data hazards which cannot be resolved by the compiler. It usually means that

[9]Note that most RISC pipelines look somewhat different; they combine instruction decode and register fetch, and often have a separate stage for data memory access.

[10]A program does not have to experience this delay if the scheduler could speculate on the outcome of these operations. This is possible for branches by using a branch target buffer, and also for loads (see [17]). For most ALU operations this would require an 'Oracle' to predict the outcome. However, see [40] for some exciting results on 'value prediction'.

[11]In some cases internal ALU bypassing can avoid this stall. For example, imagine a 32-bit adder split into two stages where the first stage adds the least significant 16 bits and the second stage the remaining bits. The result of both stages can be forwarded to their previous ones; this avoids stalls between two dependent additions.

the application contains insufficient amounts of parallelism to fill all delay slots effectively.

1.4.2 Powerful instructions

Besides aiming at a higher clock frequency by increasing the pipelining degree the architect can also reduce the number of instructions by adding more powerful instructions to the processor's instruction set. Two techniques can be applied:

1. **MD-technique:** multiple sets of data operands per operation.
   One operation is applied to multiple sets of data operands; that means, to more than the usual one or two input operands required for monadic and dyadic operations; e.g., two vectors (of equal length) can be added by specifying one operation only.
2. **MO-technique:** multiple operations per instruction.
   The execution of one instruction results in multiple basic operations being performed. Basic operations are simple monadic or dyadic operations like, integer add, subtract and multiply, logic operations like bitwise or, control operations and data move operations.

The first technique results in *data parallel* architectures, the second in *operation parallel* architectures. CISC architectures already applied both techniques, although to a limited extent. For example, a function call instruction could change the contents of several registers (like the stack and the frame pointer registers, and the program counter) and also could store several register values into a newly created stack frame. Another well known example is the string or character move instruction, which specifies the copying of a large block of data from one memory area to another. These complex CISC instructions caused several problems when applying performance enhancement methods. First, as explained already in the section 1.4.1, they did not fit into a regular pipelining scheme. Second, these instructions contributed to the existence of different, and difficult to decode, instruction formats. Finally, the use of complex instructions inhibits many compiler optimizations; e.g., some of the operations performed by a complex instruction may not be needed for common cases, or could be placed outside loops (loop invariant code motion), or replaced by more efficient basic operations. So, applying both techniques by adding complex instructions is not the right direction to proceed. However, as described below, there are better ways to apply these two techniques.

MD-technique
Vector and *SIMD* (*single instruction multiple data*) processors both exploit the use of multiple data operands per specified operation. Although their architectures are both of the data parallel type, and both support vector operations, they implement the data parallelism differently: vector processors execute a vector operation by applying this operation to a (linear) vector of data elements *sequentially in time*; SIMD processors apply the operation concurrently to all the data elements. Figure 1.7 illustrates both types of data parallel execution. It shows how instructions are (ideally) executed on a vector processor with K FUs, or equivalently, on an SIMD processor with K processor nodes. On the vector processor each instruction uses (usually) only one FU, and can have a (very) long execution time; the next instruction can be

issued if the required resources (like FUs) are available, even while the previous instructions are still executing (on different FUs). An SIMD processor executes instructions one at a time; each instruction may require all the available nodes. Both architectures have their pros and cons. The advantages of vector processors are:

- FUs can be specialized for different operation classes, as long as the mix of FUs corresponds with the average program usage. SIMD nodes each have to implement all required functionality.
- Loads and stores of vectors are easily implemented using vector or interleaved memory. While one FU does a load or store, other units operate on loaded or to be stored data. A memory access operation in an SIMD processor requires an enormous bandwidth because all units need data simultaneously. Every node needs a separate load-store path to its local (private) memory.
- Vector processors can work on shorter vectors, without becoming very inefficient. It is even possible to compress sparse vectors to get a good efficiency [25]. SIMD systems will leave (many) nodes idle, when the number of data elements to operate on is (far) less than the number nodes.

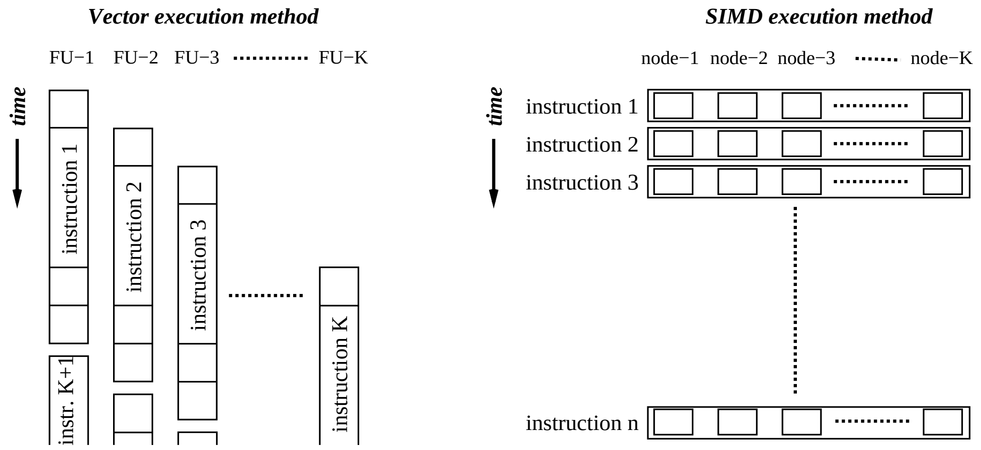

Figure 1.7: Two styles of data parallel operation: vector and SIMD execution.

FUs of a vector processor are tightly connected using a central file of vector registers to which all FUs have access. Besides this register file, FUs are also *directly* connected to each other to allow *chaining*, of operations; i.e., FU results can be *forwarded* directly to other FUs. This high connectivity is necessary because the producer and the consumer of data values could be any FU. Existing vector processors are typically heavily superpipelined, and their cycle time is therefore very short (in the $1 \sim 4$ *ns* range), but they have a limited number of FUs. They are used in the high end market, where a lot of general scientific, vector oriented calculations are performed.

On the other hand, SIMD nodes require less interconnectivity than the FUs in a vector processor. Usually SIMD nodes are often connected in a mesh or hypercube structure, where each node has four or $^2log(K)$ connections to neighbor nodes respectively. SIMD processors exploit the data locality which results from the natural data decomposition of many applications on meshes and hypercubes. They may contain hundreds or thousands of nodes. For example, the connection machines CM-1 and CM-2 of Thinking Machines [26] could contain up to 64k nodes. SIMD processors are often used for image processing applications where each node does (simple) pixel oriented operations.

MO-technique

The other technique, multiple operations per instruction, is exploited by *VLIW* (*very long instruction word*) processors. In contrast to CISC instructions, which are vertically encoded, VLIW instructions are horizontally encoded. Each instruction contains O fields, where O is the number of operations which can be executed concurrently. Each field corresponds to a specific FU. An example VLIW instruction is shown in figure 1.8. For this instruction function unit one will execute operation `sub r8,r5,3`; similarly for the other FUs. Only basic (RISC like) operations are allowed. The compiler has to make sure that all operations within one instruction are independent.

Field:	FU−1	FU−2	FU−3	FU−4	FU−5
Instruction:	`sub r8,r5,3`	`and r1,r5,12`	`mul r6,r5,r2`	`ld r3,0(r5)`	`bnez r5,13`

Figure 1.8: Example instruction for a VLIW with $O = 5$.

Compared to CISCs the advantages of VLIWs are: (1) instructions have fixed fields and are therefore easier to decode, (2) instructions still fit in a pipelining scheme, resulting in a low *CPI*, and (3), perhaps the most important, compilers can exploit the offered concurrency.

The disadvantage is a much larger code size, and consequently the need for a high instruction bandwidth. The use of internal caching and large external memories solves these problems to a large extent. VLIWs are also not object code (machine code) compatible with one of the mainstream architectures used today; we will come back to this issue later on.

VLIWs have much in common with SIMDs; both architectures accept a single instruction stream, and each instruction specifies many operations. VLIWs differ from SIMDs in the following aspects:

1. In principle, VLIWs can implement any mixture of FUs.
2. VLIW instructions allow the specification of different types of operations within a single instruction.
3. VLIWs can exploit *fine grain* parallelism; i.e. parallelism which exists between small code segments, even as small as a single operation.
4. To exploit fine grain parallelism VLIWs require, just like vector processors, a tight interconnection network with a large communication bandwidth between FUs. Typically the

FUs use a shared register file, with many register ports, to communicate.

5. VLIW instructions are large, while SIMD instructions can be compact.

Although the latter two characteristics constrain the scalability of VLIWs, the former three characteristics make them especially suitable in the area of application specific processor design.

The good characteristics of the MO-technique and the MD-technique can be combined; this occurs for example when VLIW operations may operate on either single 64-bit words or on eight bytes concurrently. Image processing (with pixel oriented operations) particularly profits from this approach.

1.4.3 Multiple instruction issue

Instead of making instructions more powerful, the architect can also decide to start multiple instructions per cycle. The following problem arises: who guarantees that issued instructions are executed in such a way as to preserve application semantics? There are three solutions to this problem:

1. The user specifies a number of instruction streams which can be issued concurrently. Communication (and therefore dependences) between streams is explicitly specified. MIMD (*multiple instruction multiple data*) computers are based on this solution. There exists an enormous amount of literature on MIMD computer design, and MIMD (parallel) computing (e.g., start with [3, 30]). It is a very active research area.

2. A sequential instruction stream[12] is fed to the processor. However, the processor has capabilities to look ahead in the stream in order to detect multiple instructions which it can issue concurrently. Processors having this capability are called *superscalar*. An excellent book on the architecture of these processors is written by Johnson [34]; the reader may also study chapter four of [25]. Superscalar processors are used in most top-of-the-line PCs and workstations; their success is greatly due to their binary compatibility with previous sequential architectures.

3. The compiler compiles the program into a dataflow representation, which means that communication between instructions is explicitly specified by the instructions themselves. This alternative allows in principle all instructions to be issued at once! *Dataflow processors* use this approach [6, 12, 14, 23, 47, 56, 59].

   The general structure of a dataflow processor is shown in figure 1.9. Instructions with their operand values are sent to the FUs (which may have some buffering capacity in the form of one or more reservation stations). When execution of an instruction finishes its result is forwarded, as a so-called *token*, to all consumer instructions. These tokens are stored in the token store, and wait there until they are matched with other tokens

[12]Sequential instruction stream means that the semantics of the program relies on the fact that instructions are executed in the specified, sequential, order. This holds for all so-called imperative programs where instructions cause state changes (through assignments) of variables which have to be processed in the specified order.

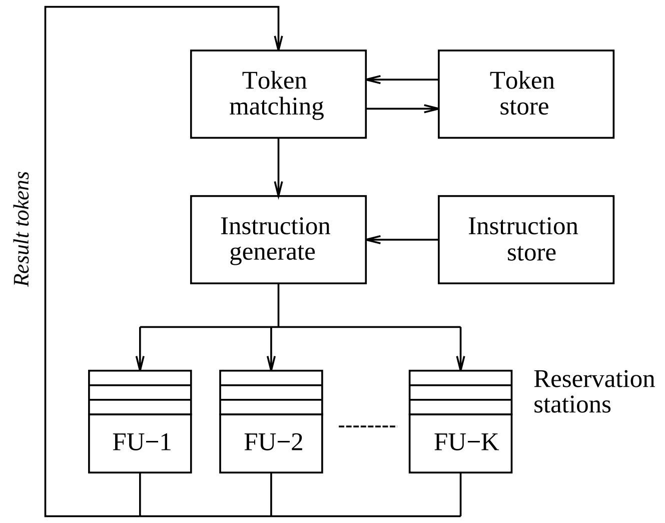

Figure 1.9: Structure of a dataflow processor.

to be consumed by the same instruction (usually an instruction is dyadic and needs two tokens). After being matched, the tokens are sent to the instruction generate unit which combines the tokens with the corresponding instruction and generates an instruction token for the FUs.

In principle dataflow processors can exploit enormous amounts of parallelism. However, so far these processors did not become successful; their run-time overhead, especially the token-matching, turned out to be too expensive and time consuming.

The difference between these three solutions is demonstrated in example 1.2.

Just like dataflow computers, MIMD computers could in principle also rely on compiler techniques to decompose a program. However to do this efficiently, without user assistance, turns out to be very hard. In contrast to dataflow computers, MIMD computers support only limited connectivity and/or limited bandwidth between processing nodes.

Example 1.2 *Three approaches to multiple instruction issue.*

In this example we illustrate the difference between MIMD, superscalar and dataflow processing of a small program fragment. Consider the following HLL code:

```
a := b + 15;
c := 3.14 * d;
e := c / f;
```

This code will be translated by the compiler to a data dependence graph (DDG), as shown in figure 1.10. This graph shows the real dependences between all instructions; we also added the required load and store operations. Clearly the graph contains two disjunct subgraphs which may be executed concurrently. The DDG can be translated to code for

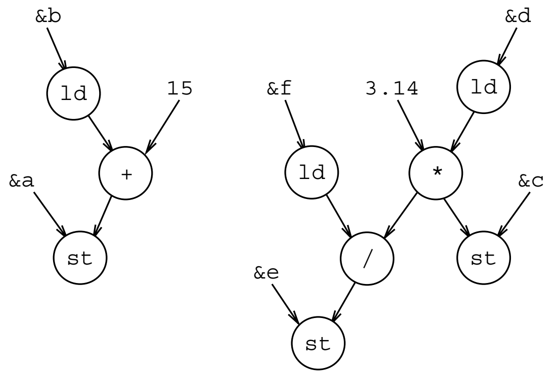

Figure 1.10: Data dependence graph example; `&a` denotes the address of `a`.

either a sequential or a dataflow processor as shown next (`M(&b)` indicates a memory access at address `&b`):

Instr.	Sequential code	Dataflow code
I1	`ld   r1,M(&b)`	`ld M(&b ) -> I2`
I2	`addi r1,r1,15`	`addi 15   -> I3`
I3	`st   r1,M(&a)`	`st M(&a)`
I4	`ld   r1,M(&d)`	`ld M(&d)  -> I5`
I5	`muli r1,r1,3.14`	`muli 3.14 -> I6,I8`
I6	`st   r1,M(&c)`	`st M(&c)`
I7	`ld   r2,M(&f)`	`ld M(&f)  -> I8`
I8	`div  r1,r1,r2`	`div       -> I9`
I9	`st   r1,M(&e)`	`st M(&e)`

How is this code executed on the different systems? Let us first consider an MIMD computer with two (sequential executing) processor nodes and shared memory. On this system instructions I1-I3 (of the sequential code) may be mapped to one processor node and instructions I4-I9 to the second node. These two parts can execute concurrently without further run-time checks; each node uses separate registers (so there are no register conflicts). Of course this is a simplified example; in practice communication is needed between code partitions, using either explicit communication primitives or rely on implicit communication through shared memory locations. This way synchronization between processor nodes is enforced by the program.

A superscalar processor will also execute the sequential code but it will try to execute multiple instructions at the same time. Note however, that the sequential code contains register dependences. For example, instruction I2 depends on the result of I1; this is a *true* dependence. There are also many so-called *name* dependences which do not exist in the DDG; they are caused by the chosen register allocation. For example, I4 and I1 both write to register `r1`; these writes have to occur in the right order. An advanced superscalar processor will detect these dependences, enforce true dependences and remove name dependences by renaming register locations at run-time; e.g., different (internal) destination registers will be chosen for instructions I1 and I4 when they execute concurrently. Actually what happens in a superscalar is the 'reverse engineering' of the DDG from the sequential code (within a limited instruction window).

Dataflow code is a direct representation of the DDG. This code does not require register allocation. Instead, results of operations are sent (as so-called *tokens* directly to those operations which need the results. For example, the result of instruction I5 is sent to both I6 and I8 (note that store instructions do not produce result tokens). Consequently the hardware does not have to enforce synchronization and undo name dependences. Conceptually all instructions can be issued at once. Of course, execution has to wait until the right tokens (containing the source operands) and function units are available.

1.4.4 Instruction pipeline overview

Within the former three subsections a variety of architectures were introduced. It is interesting and instructive to compare the instruction pipeline of a number of these architectures. Let us assume the following five actions needed to execute one instruction:

1. Instruction fetch (IF).
2. Instruction decode (DC).
3. Register fetch (RF); get the source operand values.
4. Execute its operation (EX).
5. Write-back (WB); write the result into the desired destination(s).

These actions are executed in a pipelined fashion, except for CISC processors. Based on these actions, figure 1.11 summarizes the instruction pipeline of six different discussed architectures.

RISC processors usually combine the decode and register fetch into one pipeline stage. The VLIW, superscalar and dataflow processors are assumed to contain K single cycle FUs. The VLIW fetches and decodes one instruction, containing K operations, per cycle.

The superscalar processor fetches K instructions at a time and decodes them in parallel. A separate *issue* stage has been added for resolving all the inter-instruction dependences. Instruction may be several cycles in the RF-stage waiting for their source to become available. Similarly, in the EX-stage it may have to wait for a free FU. As a result execution and completion may occur out-of-order. In order to guarantee the right (sequential) semantics of the program the write-back has to occur in-order; this is solved by leaving the results a number of cycles in the so-called reorder buffer (ROB). Results leave the ROB and are written back into the register file in the right order.

The superpipelined processor has split up its time critical IF and EX stages into S substages; each stage now requires less time.

The dataflow pipeline does not contain IF and DE stages. This does not mean that instructions are not read from memory, and that no decoding has to be done. However, at least

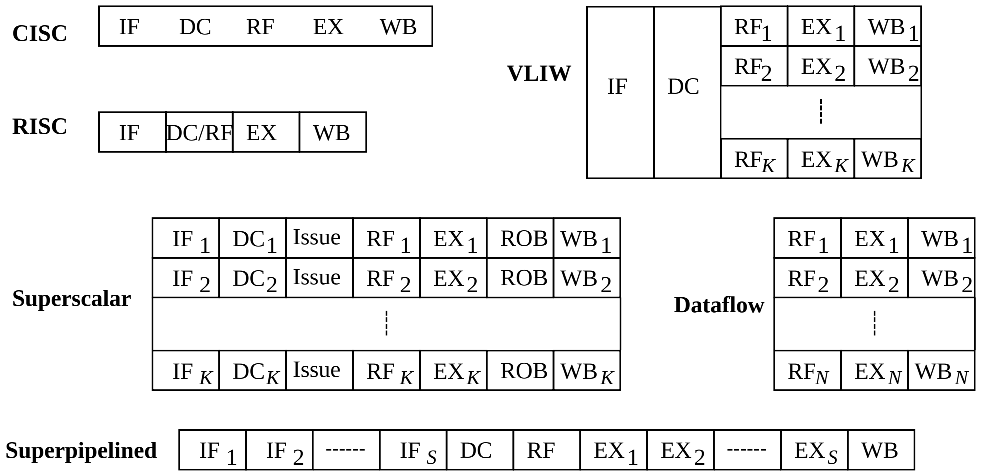
CISC
IF DC RF EX WB
RISC
IF DC/RF EX WB
VLIW
IF DC RF1 EX1 WB1 RF2 EX2 WB2 RFK EXK WBK
Superscalar
IF1 DC1 Issue RF1 EX1 ROB WB1
IF2 DC2 Issue RF2 EX2 ROB WB2
IFK DCK Issue RFK EXK ROB WBK
Dataflow
RF1 EX1 WB1
RF2 EX2 WB2
RFN EXN WBN
Superpipelined
IF1 IF2 ------ IFS DC RF EX1 EX2 ------ EXS WB

Figure 1.11: Instruction pipeline diagrams of different architectures.

conceptually, all N instructions are already fetched and decoded at the start of the program execution. Decoding is done at compile-time, in the sense that data dependences between instructions are made explicit in the code. Therefore, all program instructions can be issued at once. Note that this does not imply that all instructions execute directly; data dependences may inhibit execution temporarily. Consequently the pipelined execution may stall during the RF-stage when operand values are not ready yet. Furthermore, out of N instructions only K may execute simultaneously, because of the limit on the number of available FUs.

1.5 Architecture design space

Based on the different techniques available to the computer architect to enhance performance, as described in section 1.4, we are now able to present the resulting architectures into a four dimensional architecture design space, as illustrated in figure 1.12[13]. Each architecture can be specified as a 4-tuple (I, O, D, S), where I is the issue rate (instructions per cycle), O the number of (basic monadic or dyadic) operations specified per instruction, D the number of operands or operand pairs to which the operation is applied, and S is the *superpipelining degree*. The latter is introduced by Jouppi [35], and defined as

$$S(\text{architecture}) = \sum_{\forall Op \in I\_set} f(Op) \times lt(Op) \qquad (1.7)$$

where $f(Op)$ is the relative frequency with which operation Op occurs in a representative mix of applications, and $lt(Op)$ is the latency of operation Op; $lt(Op)$ indicates the minimal number of cycles after which operations, dependent on Op, have to be scheduled in order

[13]As noted earlier, some techniques only change the organization and not the architecture; perhaps we should have called this space the processor design space.

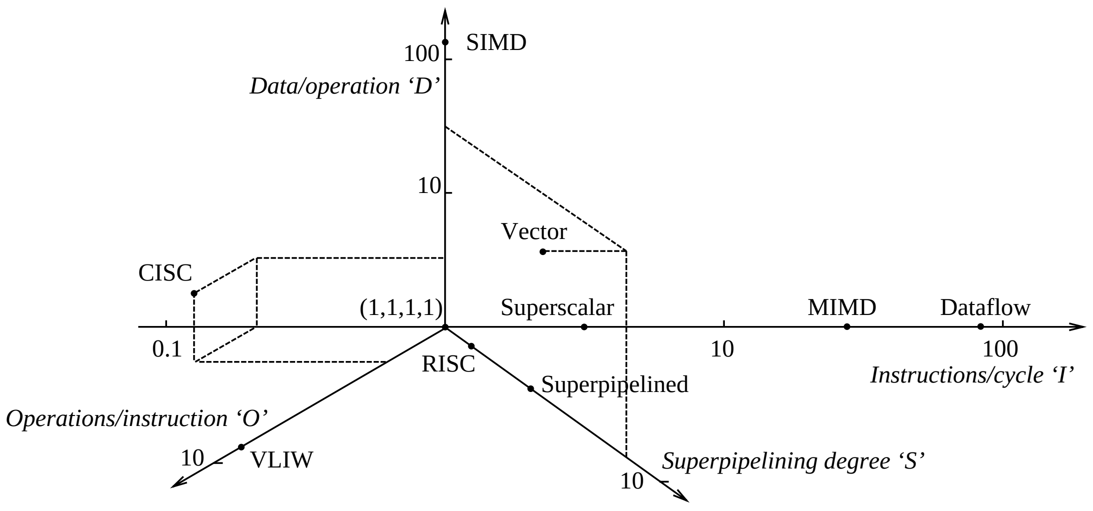

Figure 1.12: Four dimensional representation of the architecture design space.

not to cause pipeline stalls (in case the architecture supports *dependence locking*), or cause semantic incorrect results (in case the architecture does not lock on dependences). lt is related to the number of delay slots d of an operation by:

$$lt(Op) = 1 + d(Op) \tag{1.8}$$

Delay slots are invisible at the architectural level if dependence locking is used; in that case filling them with independent operations is semantically not required. However, filling them results in higher resource utilization and therefore better performance; this is part of the compiler's job. The latency of non-pipelined operations is defined as one; *S(CISC)* is therefore one. The superpipelining degree indicates how many delay slots have to be filled (on average) in order to keep the processor busy.

RISC architectures are very close to the center (1,1,1,1) of the architectural design space. RISCs have the potential of issuing one instruction per cycle ($I = 1$), where each instruction specifies one operation ($O = 1$), each operation applies to a single or single pair of operands ($D = 1$), and the superpipelining degree is slightly larger than one ($S \approx 1$).

Typical values of (I, O, D, S) for the discussed architectures are found in table 1.3. K indicates the number of FUs or processor nodes available. This table also indicates the amount of parallelism $Mpar$ offered by the architecture; this number corresponds to the average number of operations in progress, at least when the hardware is kept busy, and is defined by:

$$Mpar = I \times O \times D \times S \tag{1.9}$$

A few remarks apply to this table:

- The presented values are only meant to give the reader a feeling for the parallelism of a certain architecture; i.e., large deviations exist and several numbers may increase in

Architecture	K	I	O	D	S	$Mpar$
CISC	1	0.2	1.2	1.1	1	0.26
RISC	1	1	1	1	1.2	1.2
VLIW	10	1	10	1	1.2	12
Superscalar	6	4	1	1	1.2	4.8
Superpipelined	1	1	1	1	3	3
Vector	8	0.1	1	64	5	32
SIMD	128	1	1	128	1.2	154
MIMD	32	32	1	1	1.2	38
Dataflow	10	10	1	1	1.2	12

Table 1.3: Typical values of K, the number of FUs or processor nodes, and (I, O, D, S) for different architectures.

future.

- Although dataflow architectures issue in principle all instructions right at the start of a program (apart from function call instantiations and loop iterations), the number of active instructions is limited by the number of FUs.
- Vector architectures basically issue one instruction per cycle. However, similar to CISCs, their average *CPI* is much larger than one because of the limited number of FUs.
- As mentioned earlier, CISCs can be considered as having a long cycle time. In that case we consider them as having an issue rate of one, and a superpipelining degree of less than one. In either case they are considered *subpipelined*.

An additional remark has to be made about the presented model, because it can be confused with the well know classification model of Flynn [20] which is based on two orthogonal parameters: the number of instruction streams I (single or multiple) and the number of separate data streams D (single or multiple), giving four architecture classes (from SISD to MIMD)[14]. The two parameters of Flynn's model could be confused with the instruction issue and data/operation parameters, but they are quite different. For example, a superscalar is a single instruction stream architecture, but can issue multiple instructions per cycle. As another example, a VLIW specifies only one operand pair per operation, but would be classified as an SIMD architecture in Flynn's model. The classification presented here is more in line with new architecture developments.

We have shown four architectural design techniques to achieve a high $Mpar$. These four techniques are orthogonal, so they can be combined to create hybrid architectures. Offering a high performance translates into designing architectures with a high degree of parallelism $Mpar$. Two questions remain: given a certain area of silicon, what is the best (I, O, D, S)-tuple, and can the offered parallelism be exploited? The answers to these questions is highly

[14]However, one still does not agree whether an MISD architecture is an anomaly of the model, or corresponds to a systolic pipelined type of architecture.

application domain dependent; different domains offer different amounts of parallelism and require different architectures. In the next section we will consider those domains, and look into the available parallelism.

1.6 Fitting architecture to application

Despite the rapidly increasing performance of computing systems, the user continuously raises his performance demands. These demands result from changes in application requirements. Several developments cause these requirements to increase:

1. *More functionality*. Applications require more functionality; e.g. today most applications incorporate advanced, but performance consuming, visual user interfaces.
2. *Larger data sets*. Programs are applied to larger data sets, to get better accuracy, or better correspondence with physical reality.
3. *New application domains* emerge. Examples of these domains are multi-media, virtual-reality, neural networks, expert systems, genetic algorithm based applications, computational simulation, and highly optimizing compilers.
4. *Real-time requirements*. Several applications have performance requirements dictated by real-time constraints; one may think of image and signal processing, and control applications.

As seen in section 1.5, the computer architect can increase the degree of parallelism $Mpar$ of an architecture in order to meet these requirements. However, a parallelism of $Mpar$ does not guarantee a speedup of $Mpar$. Achievable speedup is largely application dependent. It is therefore important to differentiate between the following application domains:

- **Scalar domain:** this is the general purpose computing domain where non-numeric applications, like compilers, text formatters, and symbolic programs, dominate. Typically, programs in the scalar domain use many pointers, allocate much heap area, and spend a lot of time in the operating system. These programs hardly use floating point operations; most operations are integer based.
- **Vector domain:** scientific, highly numeric applications fit into this domain. Programs in the vector domain contain many operations on large vectors and matrices. A typical operation is the dot-product on double precision floating point vectors.
- **Application specific** or **embedded domain:** for example, multi-media applications, requiring lots of signal and/or image processing, fit into this domain. These applications often allow a natural data decomposition onto one or higher dimensional networks of processing elements. Operations are often integer based, but can be floating point as well. The required precision is largely application dependent, but is quite often much lower than 32-bits per data value.

Computers supporting these three domains are called *general purpose*, *super* or *vector*, and *application specific* computers respectively. On average, applications from the scalar domain are expected to contain less exploitable parallelism as compared to applications from the vector or application specific domain.

Application	**Domain**	**Compiler model**			
		Limited	**Real**	**Oracle-a**	**Oracle-b**
Dhrystone	scalar	1.74	3.85	4.8	74.1
Cpp	scalar	1.28	3.65	10.6	40.1
Compress	scalar	1.30	2.65	6.2	22.1
Linpack	vector	2.26	4.19	9.9	92.6
Livermore	vector	1.77	3.55	6.0	9.7
Livermore, Kernel 1	vector	2.66	7.91	520.0	527.0
Mpeg_play	appl. spec.	1.85	3.25	14.4	32.7
Mpeg_play, DCT	appl. spec.	2.47	6.12	60.9	60.9
Mccd	appl. spec.	2.06	3.18	17.4	45.0
Mccd, GVI	appl. spec.	3.39	3.39	111.0	111.0

Table 1.4: Available parallelism in different applications.

Table 1.4 gives the amount of parallelism available for different application benchmarks, taken from the discussed application domains. Most applications, except for the Mccd application which generates contours of objects in an image, are well known. Of some applications the results of an individual function, contained within the application, is shown: the 'Kernel 1' of the Livermore loops, the DCT (discrete cosine transformation) function of the Mpeg_play program, and the GVI (gray value interpolation) function of the Mccd application. The listed numbers are measured with the compiler and the trace analysis tools of the MOVE framework [28]. All measurements assume sufficient hardware resources. The amount of parallelism is determined for four compiler models. The Limited model exploits only parallelism within a basic block, where a basic block is a piece of code having one entry and one exit point. Usually basic blocks contain only a few instructions, so the amount of available parallelism is low. The 'Real' column shows what current state of the art research compilers achieve; the used compiler can schedule regions [8] which may contain tens of basic blocks; more parallelism can be found and exploited within these regions. The final two columns show the *Oracle* models; they list the amount of parallelism found, when the only limiting factor is the existence of real data dependences between instructions of a program trace. These models assume that branches are always predicted correctly, and that memory addresses of load and store operations are precisely known. All models assume sufficient hardware resources. The Oracle-a model assumes that parallelism exploitation is restricted to single functions, while the Oracle-b model even allows parallelization over function call boundaries.

Looking at the figures, we clearly recognize the different capabilities of the listed compiler models. The figures of the Oracle models are added to see the limits of available parallelism (parallelism can be higher only, if we transform or rewrite the source program). In practice we may not expect results which go far beyond the values listed for the real compiler. The available parallelism within the Livermore kernels is, on average, surprisingly low. This is caused by the fact that a number of these 24 kernels is difficult to parallelize. Note that individual functions, like the first 'Kernel' of Livermore loops, the DCT routine of the Mpeg_play program, and the GVI function of Mccd show higher amounts of available parallelism. This behavior is typical. It means that achieving a reasonable speedup for complete programs is usually much harder than for individual routines; even applications from the vector and application specific domain, as a whole, may contain less parallelism than expected.

Considering the exploitable parallelism, we distinguish two types:

- **Operational parallelism:** the parallelism between different operations of a single-threaded program[15].
- **Data parallelism:** the parallelism which exists when one or more operations can be applied to many data elements in parallel.

To illustrate the difference, look at the next program fragment, which applies a function `f` to all elements of vector `b` and assigns the result to vector `a`:

```
for i:=1 to n do a[i] := f(b[i]) od
```

If vectors `a` and `b` do not overlap, and function `f` does not contain state (it is a pure function in the mathematical sense), all iterations of this loop can be executed in parallel. This is called data (or vector) parallelism, because the same operation (the application of `f`) can be applied to all data elements concurrently. The program fragment can then be changed into a multi-threaded one, by changing the `for`-statement into a `forall`-statement. Long vectors give rise to large amounts of exploitable data parallelism; many execution units can be kept busy without to much communication between units.

The compilation of function `f` results in a number of basic operations. Between these operations there is usually still a limited amount of parallelism left, which can be exploited. This parallelism is of the operational type. Exploitation requires a high connectivity between the execution units because of the large number of dependences between those basic operations.

In general it can be said that all programs contain at least limited amounts of operational parallelism, while data parallelism shows up primarily in the scientific and application specific domains.

Single instruction stream computers can be characterized according to the type of parallelism to which their processor is oriented. We differentiate:

- **Instruction level parallel processors,** or ILP processors. They support the exploitation of operational parallelism. Processors in this class either contain multiple execution or function units which are usually differentiated to support different types of operations,

[15]A thread is a sequence of instructions with a single locus of control; i.e., when executing a single threaded program only one program counter is required which points to the currently active (executed) instruction. Multi-threaded programs have multiple active instructions (multiple control loci) each requiring a separate program counter. Usually within a single thread there is still parallelism available of the operational type.

or they apply the superpipelining technique; that is, I, O or S are greater than one, D is usually one. The following processors belong to this class: superscalars, VLIWs, superpipelined and dataflow processors, and processors using a Transport Triggered Architecture (TTA) [11].

TTAs are comparable to VLIWs, with the difference that the compiler has access to all the internal data transports. Instead of programming operations (which internally trigger all kinds of data transports), for a TTA the data transports are programmed. These data transports trigger the operations. Letting the compiler (or assembly code writer) control these transports gives many new code optimization possibilities. E.g., many superfluous data transports, occurring in standard VLIWs and superscalar processors can be avoided. This also reduces the power consuption.

- **Data level parallel processors,** or DLP processors. These processors primarily support data parallelism. They have the common property that D is in the range from tens to thousands, while I and O are usually one. Examples are SIMD and vector processors; for the latter $S > 1$ as well.

As mentioned, operational parallelism is limited, but found everywhere; therefore techniques exploiting this type of parallelism will always increase performance. In addition, ILP processors profit from data parallelism as well; multiple simultaneously active operations means multiple operands are operated upon concurrently. In that sense ILP processors are theoretically more powerful than DLP processors.

Application specific processors (ASPs) can, depending on the supported application, exploit both types of parallelism. Their power lies in the fact that they eliminate unnecessary features like, virtual memory, high precision integer or floating point support, and cache coherency protocols. This reduces their complexity, and allows them to support higher $Mpar$ values, or the same $Mpar$ value at lower costs.

Although single instruction stream computers can become quite powerful, especially when they exploit both types of parallelism, they have their limitations; e.g., when the control flow of a program is strongly data dependent. In that case *multiple instruction stream computers* (like MIMD systems) may be the solution to the high power demands. They contain many nodes which are connected through a high performance communication network. These nodes may also exploit operation or data parallelism. The technique of using multiple instruction streams can be applied orthogonal to the architecture of a single node. Most multi-media processors exploit ILP to achieve sufficient performance; therefore we will focus our discussion on the ILP-class.

1.7 Compiler developments

Exploiting parallelism has its price: the parallelism has to be detected and exploited efficiently by the architecture under consideration. Let us look at which translation and interpretation steps have to be taken in order to execute a program written in a HLL:

1. Frontend compilation. Lexical analysis and parsing of the program, performing optimizations (often optimizations are made after the next step), and compilation to basic operations. During this step *alias analysis* is also performed; it decides if variables can be allocated to internal registers [2, 60].
2. Determine dependences and produce data and control dependence graphs, or alternatively, a data flow graph which includes control dependences [59].
3. Partitioning the graph (for multiple instruction stream computers). Partitions have to be made such that available nodes are kept optimally busy; in practice this often means that inter-node communication has to be minimized, that is, the number of interconnections between partitions must be minimized.
4. Bind partitions to nodes (for multiple instruction stream computers).
5. Bind operands to locations. For example, non-aliased variables and temporaries are bound to local registers for a certain life-time in order to reduce off-chip data traffic; this is called *register allocation*.
6. Bind operations to time slots, that is, decide when to execute operations and data transports. This is usually called *scheduling*.
7. Bind operations to FUs. When there are multiple function units of the same type, a binding choice has to be made.
8. Bind transports to buses. Performing operations requires the transport of many data values; buses must be allocated to these transports.
9. Execute the operations and perform the transports.

The order of these steps is not necessarily the same as presented above. It is clear that the first step is performed by the compiler, and the final step by the hardware. Intermediate steps perform partitioning, scheduling and binding, which should result in optimal utilization of available hardware resources. Who performs these intermediate steps? In particular, given a *sequential program*, that is a program which does not convey any explicit information regarding parallelism (e.g. a program specified in C or Fortran), who is responsible for detection and efficient exploitation of the parallelism within this program? There are three possibilities:

1. The programmer. Detection is done at application *coding-time*. The programmer himself partitions the program into different instruction streams, or threads, and assigns those streams to processor nodes of an MIMD computer. Also within an instruction stream, the programmer can specify data parallel operations (like vector operations) and data decompositions.
2. The compiler. Detection is done at *compile-time*. The compiler splits up the program into multiple instruction streams and/or detects and exploits intra instruction stream parallelism.
3. The hardware detects during *run-time* the available parallelism between individual operations and binds them to the available hardware resources.

One extreme solution is to let the hardware do everything. This is easy for both the compiler writer and for the programmer. However, the hardware gets rather complex, even if it has to dynamically detect and exploit only small amounts of parallelism[16]. The other extreme solution is to let the programmer be responsible for detection and exploitation of all available parallelism. However, the programmer easily makes mistakes when he has to split up an application into several threads; he has to deal with indeterminism and explicit synchronization between threads. These factors give a huge boost to current compiler research.

As will be clear from the former discussion, three compiler fields can be distinguished:

1. Compiling for MIMD. The compiler has to split a sequential program into multiple threads in such a way as to achieve acceptable speedup, when run on an MIMD computer.
2. Compiling for DLP processors. Transform a sequential program into a data parallel one, suitable for a vector or an SIMD processor, or even for an MIMD computer, when using the single program multiple data (SPMD) model.
3. Compiling for ILP processors, in particular for VLIWs, superscalar and superpipelined processors. In this case the compiler has to search for independent operations which are scheduled within a single instruction (for VLIWs) or in successive instructions (for superpipelined and superscalar processors), in order to fill the pipelines and keeping FUs busy.

The current state-of-the-art in compiler technology is such that compilers for MIMD computers and DLP processors need assistance from the programmer in order to be successful; e.g., annotations to an otherwise sequential program, like the specification of data decompositions, have to be added. Another approach is to use non-imperative languages (i.e. languages which do not have a sequential program semantics) for program specification, like functional languages or logic languages. These approaches suffer from two disadvantages, which will not make them popular on a short term: (1) they deviate from accepted programming standards and methodologies, and (2) the semantic gap to existing architectures is high and difficult to bridge efficiently by current compilers. In the long run these approaches may become successful though.

The third area, compiling for ILP processors, is getting mature. This is most important for Superscalars, VLIWs, and TTAs, and we will further concentrate on it. Here, the job of the compiler is to reduce the dynamic code size (N_{instr}) as far as possible. Code compaction techniques were already researched in the seventies, where they were applied to reduce the static size of microcode. However, as noted by Fisher [19], as soon as the scheduling scope surpasses basic blocks, compaction of dynamic code size is not equivalent any more to static size compaction. New techniques were invented (e.g., trace scheduling, speculative code motion, and software pipelining), and successfully applied, to allow the movement of operations from one basic block to another, in case such a movement reduces the dynamic code size. Of course, as mentioned in the former section, the compiler can only achieve good results if

[16]Therefore hardware will hardly be able to split up a program into multiple threads; this would require a global view of the program.

the application provides enough exploitable parallelism. Table 1.4 showed that not all available parallelism is exploitable by single instruction stream computers, unless at the price of excessive code duplication and corresponding increase of instruction bandwidth; there still is a large gap between the amount of parallelism offered by the application and the amount exploitable by the compiler and the hardware.

Despite the successful compilation techniques for ILP processors, there are reasons to offer hardware support for detection and exploitation of parallelism in ILP processors. In the nineties many ILP processors resulted as an outgrowth of sequential architectures, which did not support parallel execution. For binary compatibility reasons with former generations these ILP processors are mostly of the superscalar type[17]. Superscalars offer hardware support for many of the listed steps necessary to execute a program. Another reason for offering hardware support is that some information necessary for the efficient exploitation of hardware is run-time dependent, like:

- Memory disambiguation information. The compiler sometimes has to assume dependences between memory access operations which do not exist. As a result some code motions, like changing the order of load and store operations, are inhibited [36, 38].
- Branch directions. Although the compiler does a reasonable job in predicting branch directions (in particular when profiling information is used), hardware support may be needed in order to further reduce branch penalties (especially for scalar code, which shows fewer predictable branches). A branch target buffer may be implemented in order to cache successor instruction addresses of branch instructions. Furthermore, moving operations across branches can be supported by techniques like boosting [52], guarded operations, and a non-trapping set of instructions (see e.g. [18, 41]).
- Operation latencies which may be indeterministic. For example, the compiler often is unable to predict the latency of a load operation, because it does not know (for sure) whether the datum is in the data cache; if not, a cache miss occurs. The hardware may reduce the miss penalty by scheduling other operations, independent from the previous load.

Based on the given steps for executing a HLL program, and inspired by an article of Rau and Fisher [50], figure 1.13 shows a division between the responsibilities of the compiler and the hardware for ILP processors. Note that steps three and four are left out (see beginning of this section); they are not needed for ILP processors. The figure illustrates how this division is made for different architectures: *superscalar, dataflow, multi-threaded, independence architectures, VLIWs,* and *TTAs.*

Superscalar architectures shift most responsibilities to the hardware, while VLIWs and TTAs shift most responsibilities to the compiler. Although superscalars are able to run unmodified object code of former generation sequential architectures, they certainly need compiler assistance to execute programs efficiently. Hardware complexity limits superscalars to a very

[17]Many microprocessor manufacturers had experienced the trouble and expense in going from CISC to a binary incompatible RISC architecture; it was not attractive to go through this experience again before capitalizing the investment made in the RISC architecture.

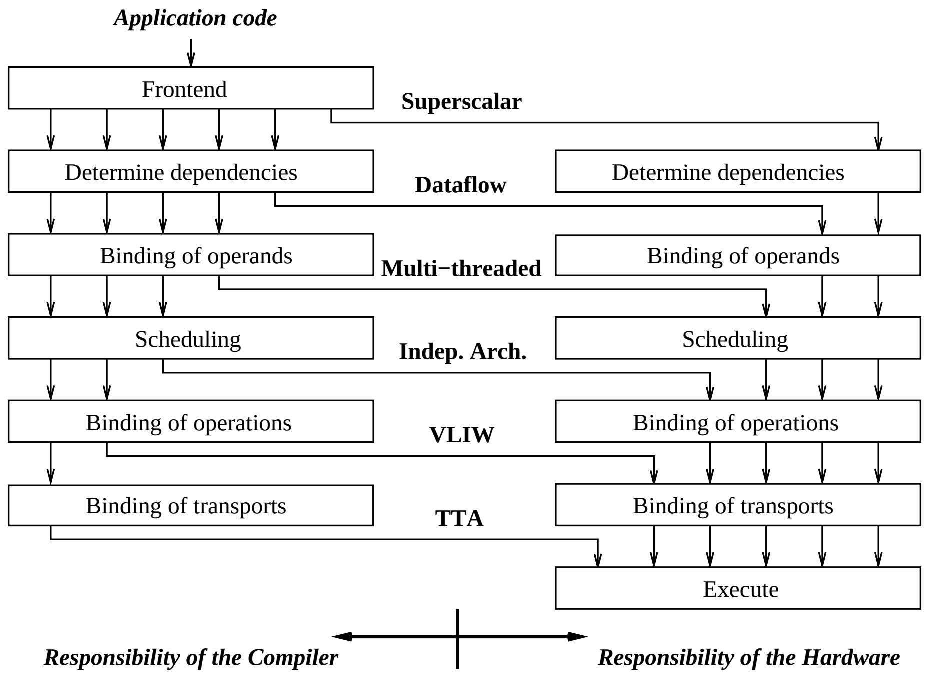

Figure 1.13: The division of responsibilities between hardware and compiler.

small run-time scheduling window (see e.g. [34]); therefore the compiler has to schedule code in such a way that operations within the scheduling window are independent from each other as much as possible.

Dataflow architectures were described in section 1.4.3; they let the compiler perform the frontend compilation and the data dependence analysis; other steps are performed by hardware. Dependences are made explicit in the object code, that is, each operation specifies its successor operations to which the result value(s) of the operation must be sent. These values are sent by so-called *tokens*; they contain both the value and the successor specification. When tokens are received they have to be matched at run-time with instructions and other tokens (most instructions require multiple tokens). This matching turns out to be very expensive and time consuming.

Multi-threaded architectures can be considered as architectures which do the scheduling at run-time, but, in contrast to dataflow architectures, rely on the compiler for binding of operands to locations (this avoids token matching). A further difference with dataflow and superscalar architectures concerns the scheduling granularity. For the latter architectures this granularity is one instruction; for multi-threaded ones it is a thread. Although multi-threaded architectures support *fine grain* execution in the form of small threads, the compiler should try to enlarge threads (and therefore the scheduling granularity) in order to avoid the still existing thread switching overhead. Typically, this overhead is in the order of 10 cycles. A number of multi-threaded architectures have been proposed. These architectures support multiple threads by offering hardware support for efficient thread synchronization and thread switching. Just like superscalar and dataflow architectures, they can tolerate long, possibly indeterministic, latency operations, like data loads, by switching quickly to another runnable thread of execution. Some of these architectures are presented as descendents from dataflow architectures [12, 13, 43, 48]. They tend to avoid the dynamic token matching. Others are inspired by concurrent object oriented computation, like the J-machine [44] and Sparcle [1].

More recently several research groups have proposed the use of multi-processor architectures running multiple threads extracted from a single (sequential) program, with the aim to exceed the amount of instruction level parallelism exploitable from a single thread [16, 29, 39, 53, 57]. Each thread is run on a separate processor, presumably a superscalar processor; this way they combine both the exploitation of fine grain (at the instruction level) and more coarse grain (thread level) parallelism.

Independence architectures require that the compiler has to specify independent operations whose operands are available in a given cycle; in other words, the compiler performs the scheduling. The hardware performs the binding of operations and transports to hardware resources. An example is the Horizon architecture as presented in [54]; each operation encodes a number H which specifies the next H concurrent operations. The hardware can execute these H (or less) operations concurrently, without having to test if they are dependent or whether their operands are ready. The hardware still has to allocate (bind) these operations to FUs.

VLIW architectures are like independence architectures, except that the compiler has to do the binding of operations to FUs as well.

Figure 1.13 also shows that TTAs depend even more on compilers than VLIWs, because the compiler has to bind transports to buses, and schedule them explicitly. The question is, whether the compiler is able to perform this binding and scheduling efficiently? This is part of the research performed within the MOVE project. The results of this compiler effort are encouraging [32].

1.8 Summary and evaluation of trends

In this chapter the architecture of a processor is introduced as an intermediate abstraction level between the hardware data path and the application level. It fills the gap between these two levels, and therefore reduces the job of the compiler. Architectures (and their corresponding processors) have been viewed from different perspectives:

- The parallelism ($Mpar$) offered. This resulted in a classification based on four parameters: the issue rate I, the number of operations per instruction O, the number of data elements per operation D, and the superpipelining degree S.
- The supported application domain(s): scalar, vector or application specific (embedded) domain.
- The compiler view. We analyzed what architectures do at run-time and what actions remain to be done at compile-time.

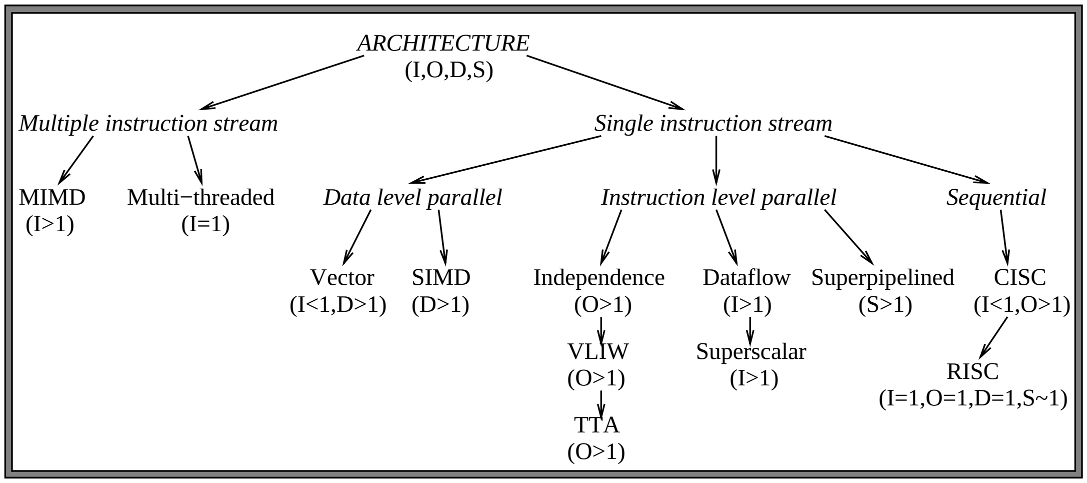

Figure 1.14: Relationship between architectures.

- The type of parallelism supported: operational or data parallelism, resulting in ILP and DLP processors.
- The number of supported instruction streams; we distinguished single and multiple instruction stream computers.

As a result many different architectures were discussed. Figure 1.14 illustrates the relationship between all discussed architectures, taking the different perspectives into account. With each architecture its characteristic form of parallelism supported is indicated; e.g., a superpipelined architecture has $S > 1$. Note that the multi-threaded architecture in its basic appearance only supports single issue; only one thread is scheduled at a time. Of course these architectures are good candidates to be extended to large issue rates.

The role of the computer architect is to develop the right architecture for a certain application domain, such that the cost-performance ratio is minimized, and performance requirements are met. Several techniques to increase performance exist. The use of these techniques has to match with (1) available VLSI capabilities, (2) application domain, and (3) progress in compiler technology. Besides the right match at the right time, the architect is faced with other problems, related to Amdahl's law [4], and with design complexity or binary incompatibility problems. These issues are discussed in the following subsections.

1.8.1 The right match

It is important to introduce the right architecture at the right time; this is illustrated by figure 1.15. Drawn are the circuit density development of microprocessor chips (in transistors/chip), and rough estimates of the required number of transistors for different architectures (horizontal lines)[18]. As shown, a 32-bit RISC core (with only integer support) did fit onto a single chip in the early eighties, a VLIW or superscalar in the early nineties. A single chip implementation is important because it avoids heavy inter-chip communication. For example, RISCs can only achieve a low *CPI* if they support enough operand transport capacity, e.g. two reads and one write per cycle. If these transports involve off-chip communication, it becomes too slow and becomes too expensive to perform them concurrently (it would require many external connections). Also the instruction and data bandwidth have to be sufficient. Most RISCs therefore have on-chip caches. The same holds for VLIWs and superscalars for which it is necessary to keep the inter-FU communication on-chip as well. As a counter example we mention two (formerly) commercial VLIWs: Multiflow TRACE and CYDRA-5 [7, 55]. They were not successful. Their (VLIW) concept was splendid, but it required many off-chip data communication channels. They suffered from being too early on the market; their concept did not have the right match with available VLSI technology.

Something similar might have been the case with dataflow processors which were extensively researched in the eighties[19]. Their run-time overhead turned out to be too expensive[20]. Strangely enough superscalars are nowadays a great success while they even have to perform more run-time checking, although within a *limited* instruction window; they require run-time dependence checking while dataflow processors do not need this (see figure 1.13). Actually, the internals of a dynamic scheduling superscalar can be viewed as a dataflow processing engine [51]. Perhaps current technology densities allow a revival of dataflow processing.

Besides a right match between architecture and VLSI there also has to be a right match between architecture and compiler technology. We have mentioned in this respect the WCS architectures, introduced in the seventies. Compilers could not exploit the features of those architectures at that time. Similar arguments hold for MIMD computers. They lack sufficient inter-node communication support, because nodes do not fit on a single chip, and the required compiler technology (and software environment) is not yet available. This may change during the next decade.

Traditionally, computer architecture was identified with instruction set design. The quality of an instruction set was judged by characteristics like consistency, orthogonality, propriety, and generality (see [9, 21] for an explanation), without concern for implementation, realization, and compiler applicability aspects. However, to avoid the above mentioned pitfalls the architect is required to have knowledge of the complete mapping trajectory of the

[18]The size of the internal cache largely influences this amount, but the estimations are still useful for this discussion.

[19]Some dataflow processors were commercially exploited, like the NEC PD7281 microprocessor which implemented so-called static dataflow. It did not support token coloring needed for dynamic procedure calls and loops with run-time dependent iteration counts.

[20]Other reasons for dataflow processors not being successful have to do with object and source code compatibility. It will be clear that dataflow processors require a completely different binary instruction format. Also the HLLs meant to be run on these processors are quite different. They are *functional languages* which have a so-called *single assignment* semantics; i.e., a variable can only be bound to a value once; this binding is active in the whole program. As noted in section 1.7 these type of languages are not (yet) popular. A combination of dataflow with imperative languages may have more success.

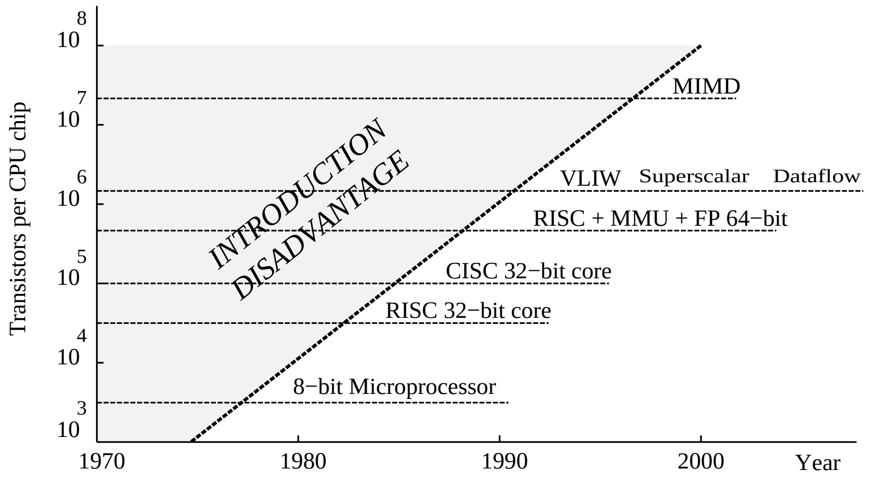

Figure 1.15: Number of transistors required to fit on a single chip.

application to the hardware at the data path level.

1.8.2 Amdahl's law for architects

People who practice the parallelization of code experience Amdahl's law which states that the speedup achieved when parallelizing an application using N_p processors is limited by:

$$Speedup \equiv \frac{\text{serial processing time}}{\text{parallel processing time}} \leq \frac{1}{f_{par}/N_p + 1 - f_{par}} \tag{1.10}$$

where f_{par} is the fraction of code which could be parallelized; the serial fraction, $1 - f_{par}$, cannot be parallelized. The architect runs into problems which are related to this law. The following examples illustrate these type of problems:

1. It is relative easy to duplicate the arithmetic capabilities of a processor. However, in doing this, load and store operations may become the performance bottleneck. Supporting multiple concurrent loads and stores is far more complex and expensive.
2. Increasing the degree of parallelism $Mpar$ may uncover other bottlenecks, like the inability of the compiler to disambiguate all load and store operations, or the control bottleneck (i.e., not being able to predict all branches correctly). This limits the reduction of the critical path of operations, and therefore limits speedup.
3. Making a microprocessor twice as fast does not mean that the computer system shows a speedup of two. As soon as data goes off-chip the architect may experience that off-chip traffic (like memory and I/O traffic) does not speedup so easily. Although off-chip bandwidth can be bought (by increasing the number of pins, and increasing the clock rate), external data access latencies quickly become the next bottleneck.

Many other examples could be added. In general all parts of the 'execution chain' have to be strengthened in order to be successful.

1.8.3 Design complexity and binary incompatibility

Another problem has to do with the increasing design complexity. As indicated in figure 1.3, there are tendencies to increase again the architectural level. Also the hardware level increases in future. These tendencies lead to an increase of the design complexity. The capability to put former external functionality, like second level cache control, and multiprocessor cache consistency support, on-chip enforces this effect. As a result the microprocessor design-to-market time increases. Given the rapid developments in VLSI technology, the manufacturer may end up with an advanced design realized in old technology.

A major reason why processors get complex, especially in the general purpose application domain, was already mentioned in section 1.7: manufacturers favor complex superscalars above less complex VLIWs. There are two reasons for this:

- **Binary compatibility.** The new VLIW architectures are incompatible with existing architectures used within the general purpose domain. The latter all assume an instruction stream for which (1) each instruction specifies a single (RISC style) operation only, and (2) FU latencies are not visible at the architecture level[21]. Being binary compatible (compatibility at the object code level) is a major selling argument.
- **Extensibility.** Adding FUs to a VLIW changes its architecture; it is therefore difficult to support a range of identical architectures with different degrees of parallelism.

However, as was clear from figure 1.13, superscalars need hardware for checking dependences and performing scheduling and binding of operations and operands. VLIWs on the other hand (and TTAs even more) put much complexity into the compiler[22], in order to obtain less complex hardware. Until now VLIWs are not used in the general purpose application domain. Especially the extensibility problem is severe. Users (and therefore processor manufacturers) are willing to change to another architecture only infrequently; i.e., not for every new processor generation (which means every two or three years). This problem can be circumvented however. It is possible to make VLIWs with many FUs upward and downward compatible with a VLIW with fewer, or even one FU. Upward compatibility requires that object code generated for a single FU VLIW also runs on a K-FU VLIW. This does not look hard to support; a simple solution is to let the code specify for how many FUs it is generated, and only use this specified number of FUs. Downwards compatibility looks harder. For example, parallel code like:

```
r1 := r2 + r3;    r2 := r1 + r3
```

[21]There are often a few exceptions to these two rules; e.g., SIMD style operations like 'string-copy' may be supported, and branch delays may be visible.

[22]Superscalars still need to use the same compiler techniques as for VLIWs in order to optimally exploit the available hardware.

behaves differently when these two operations are executed sequentially. The use of scheduling restrictions may solve this problem.

Different FU latencies can also be supported within a range of architectural compatible processors. For example, as proposed in [49], the FU latencies for which the code was compiled could be made explicit in the code. If the actual (hardware) FU latency is larger than the one for which the code was compiled, the processor has to lock; if it is shorter, the result is available in time. Application of these methods may introduce VLIWs into the world of general purpose computing.

Another reason why VLIWs were not effective in the scalar domain was the difficulty in keeping many FUs busy; a small number of FUs is usually sufficient. If the number of FUs is low, the overhead of superscalars is still manageable. However, just like almost every microprocessor nowadays contains floating point support which is hardly used by scalar applications, future processors may include more FUs, even if they are only useful for a limited number of applications.

References

1. Agarwal, A. *et al.*, “Sparcle: an evolutionary processor design for large-scale multiprocessors,” *IEEE Micro*, pp. 48–61, June 1993.
2. Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science, Reading, Massachusetts: Addison-Wesley Publishing Company, 1985.
3. Almasi, G. S. and Gottlieb, A., *Highly Parallel Computing (2nd ed.)*. The Benjamin/Cummings Publ. Company Inc., 1994.
4. Amdahl, G. M., “Validity of the single processor approach to achieving large scale capabilities,” in *Proceedings of the AFIPS Spring Joint Computer Conference*, (Atlantic City, New Jersey), Apr. 1967.
5. Amdahl, G. M. *et al.*, “Architecture of the IBM/360,” *IMB Journal of Research and Development*, vol. 8, pp. 87–101, Apr. 1964.
6. Arvind and Culler, D., “Dataflow Architectures,” *Annual Reviews in Computer Science*, vol. 1, pp. 225–253, 1986.
7. Beck, G. R., Yen, D. W. L., and Anderson, T. L., “The Cydra 5 Minisupercomputer: Architecture and Implementation,” *The Journal of Supercomputing*, vol. 7, pp. 143–180, May 1993.
8. Bernstein, D. and Rodeh, M., “Global instruction scheduling for superscalar machines,” in *Proceedings of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.
9. Blaauw, G. A. and Brooks Jr., F. P., *Computer Architecture; Concepts and Evolution*. Addison-Wesley, 1997.
10. Campione, M. and Walrath, K., *The Java Tutorial, Object-Oriented Programming for the Internet*. Addison-Wesley, 1996.
11. Corporaal, H., *Microprocessor Architectures; from VLIW to TTA*. John Wiley, ISBN 0-471-97157-X, 1998.
12. Culler, D. E., *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, MIT, computer science lab., report number: 446, 1990.
13. Culler, D. E. *et al.*, “Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine,” in *Proceedings of ASPLOS-IV*, (Santa Clara, California), pp. 164–175, Apr. 1991.
14. Dennis, J., “Dataflow supercomputers,” *IEEE computer*, vol. 13, pp. 48–56, Nov. 1980.
15. Digital Equipment Corporation, *VAX-11/780 Architecture Handbook*, 1979.

16. Dubey, P. K., O'Brien, K., O'Brien, K., and Barton, C., "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading," in *International Conference on Parallel Architectures and Compilation Techniques*, pp. 109–121, 1995.
17. Eickemeyer, R. J. and Vassiliadis, S., "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37, pp. 547–564, July 1993.
18. Ertl, M. A. and Krall, A., "Exception Handling vs. Speculative Execution," Tech. Rep. TR 1851-1992-10, Technische Universität Wien, Institut fu ̈r Computersprachen, July 1992.
19. Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.
20. Flynn, M. J., "Very high speed computing systems," *Proceedings of the IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.
21. Goor, A. J. v. d., *Computer Architecture and Design*. Addison-Wesley, 1989.
22. Goor, A. J. v. d. *et al.*, *Van Pascal tot Chips*. Department of Electr. Eng. TUD, 1982.
23. Gurd, J., "The Manchester Dataflow Machine," *Future Generation Computer Systems*, vol. 1, pp. 201–212, June 1985.
24. Guttag, K., Gove, R. J., and Aken, J. R. V., "A Single-Chip Multiprocessor for Multimedia: The MVP," *IEEE Computer Graphics & Applications*, pp. 54–64, Nov. 1991.
25. Hennessy, J. L. and Patterson, D. A., *Computer Architecture, a Quantitative Approach, Second Edition*. Morgan Kaufmann publishers, 1996.
26. Hillis, D., *The Connection Machine*. ACM Distinguished Dissertation, MIT Press, 1985.
27. Hollingdale, S. H., *High speed computing; methods and applications*. The English University Press, 1959.
28. Hoogerbrugge, J., *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, ISBN 90-9009002-9, Feb. 1996.
29. Hordijk, J. and Corporaal, H., "The Potential of Exploiting Coarse-Grain Task Parallelism from Sequential Programs," in *HPCN Europe '97, The International Conference and Exhibition on High-Performance Computing and Networking*, (Vienna, Austria), Apr. 1997.
30. Hwang, K., *Advance Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.
31. Itano, K., "PASDEC: A Pascal Interactive Direct-Execution Computer," in *Proceedings of the international workshop on high-level language computer architecture*, (Fort Lauderdale, Florida), pp. 161–169, 1982.
32. Janssen, J., *Compilation Strategies for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, 2001.
33. "Java^{TM} - Programming for the Internet." Website: http://java.sun.com/, 1996.
34. Johnson, W. M., *Superscalar Microprocessor Design*. Prentice Hall, 1991.
35. Jouppi, N. P. and Wall, D. W., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.
36. Karkowski, I. and Corporaal, H., "Overcoming the Limitations of the Traditional Loop Parallelization," in *HPCN Europe '97, The International Conference and Exhibition on High-Performance Computing and Networking*, (Vienna, Austria), Apr. 1997.
37. Kunkel, S. R. and Smith, J. E., "Optimal Pipelining in Supercomputers," in *ISCA-13*, (Tokyo, Japan), pp. 404–414, June 1986.

38. Larus, J. R., “Parallelism in Numeric and Symbolic Programs,” in *Proceedings of International Workshop on Compilers for Parallel Computers*, pp. 157–170, December 1990.
39. Li, Z., Tsai, J., Wang, X., Yew, P., and Zheng, B., “Compiler Techniques for Concurrent Multithreading with Hardware Speculation Support,” in *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.
40. Lipasti, M. H. and Shen, J. P., “Exceeding the Dataflow Limit via Value Prediction,” in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (Paris, France), pp. 226–237, Dec. 1996.
41. Mahlke, S. A. *et al.*, “Sentinel scheduling for VLIW and Superscalar Processors,” in *Proceedings of ASPLOS-V*, (Boston), pp. 238–247, Sept. 1992.
42. Meng, T. H., *Synchronization Design for Digital Systems*. Kluwer Academic Publishers, 1991.
43. Nikhil, R. S., Papadopoulos, G. M., and Arvind, “*T: a Killer Micro for a Brave New World,” Tech. Rep. Computation Structures Group Memo 325, MIT, Jan. 1991.
44. Noakes, M. D., Wallach, D. A., and Dally, W. J., “The J-Machine Multicomputer: An Architectural Evaluation,” in *ISCA-20 proceedings*, pp. 224–235, May 1993.
45. O’Connor, J. M. and Tremblay, M., “PicoJava-I: The Java Virtual Machine in Hardware,” *IEEE Micro*, vol. 17, no. 2, pp. 45–53, 1997.
46. Organick, E. I. and Hinds, J. A., *Interpreting Machines: Architecture and Programming of the B1700/B1800 Series*. Operating and Programming systems series, North-Holland, 1978.
47. Papadopoulos, G., *Implementation of a General-Purpose Dataflow Multiprocessor*. PhD thesis, MIT, computer science lab., 1988.
48. Papadopoulos, G. M. and Traub, K. R., “Multithreading: A Revisionist View of Dataflow Architectures,” in *18th International Symposium on Computer Architecture*, (Toronto), May 1991.
49. Rau, B. R., “Dynamically Scheduled VLIW Processors,” in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, (Austin, Texas), pp. 80–92, Dec. 1993.
50. Rau, B. R. and Fisher, J. A., “Instruction-Level Parallel Processing: History, Overview and Perspective,” *The Journal of Supercomputing*, vol. 7, pp. 9–50, May 1993.
51. Simone, M. *et al.*, “Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor,” in *The 22nd Annual International Symposium on Computer Architecture*, pp. 151–162, June 1995.
52. Smith, M. D., Horowitz, M., and Lam, M. S., “Efficient superscalar performance through boosting,” in *Proceedings of ASPLOS-V*, (Boston), pp. 248–261, Sept. 1992.
53. Sohi, G. S., Breach, S. E., and Vijaykumar, T., “Multiscalar Processors,” in *ISCA’22 proceedings*, (Santa Margherita Ligure, Italy), pp. 414–425, June 1995.
54. Thistle, M. R. and Smith, B. J., “A processor architecture for Horizon,” *Proceedings of Supercomputing*, pp. 35–41, Nov. 1988.
55. “Trace: Technical summary.” Multiflow Computer Inc., June 1987.
56. Treleaven, P., “Future Computers: Logic, Data Flow, ..., Control Flow?,” *IEEE Computer*, vol. 17, pp. 47–57, Mar. 1984.
57. Tsai, J.-Y. and Yew, P.-C., “The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation,” Tech. Rep. TR 96-037, Univ. of Minnesota, Department of Computer Science, 1996.
58. Uyemura, J. P., *Fundamentals of MOS Digital Integrated Circuits*. Addison-Wesley, 1988.
59. Veen, A. H., *The misconstrued semicolon: Reconciling imperative languages and dataflow machines*. PhD thesis, Center for Mathematics and Computer Science, Amsterdam, 1986.

60. Wilhelm, R. and Maurer, D., *Compiler Design*. Addison-Wesley, 1995.