12

Instruction Level Parallelism

What is computer architecture?

(a) We define *computer architecture* as the arrangement by which the various system building blocks—processors, functional units, main memory, cache, data paths, and so on—are interconnected and inter-operated to achieve desired *system performance*.

(b) Processors make up the most important part of a computer system. Therefore, in addition to (a), *processor design* also constitutes a central and very important element of computer architecture. Various functional elements of a processor must be designed, interconnected and inter-operated to achieve desired *processor performance*.

Example 12.1 Performance bottleneck in a system

In Fig. 12.1 we see the schematic diagram of a simple computer system consisting of four processors, a large shared main memory, and a processor-memory bus.

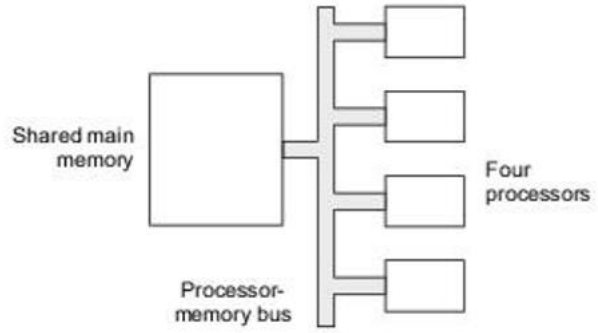

Fig. 12.1 A simple shared memory multiprocessor system

For the three subsystems, we assume the following performance figures:

(i) Each of the four processors can perform double precision floating point operations at the rate of 500 million per second, i.e. 500 MFLOPs.

(ii) The shared main memory can read/write data at the aggregate rate of 1000 million 32-bit words per second.

(iii) The processor-memory bus has the capability of transferring 500 million 32-bit words per second to/from main memory.

This system exhibits a performance mismatch between the processors, main memory, and the processor-memory bus. The data transfer rates supported by the main memory and the shared processor-memory bus do not meet the aggregate requirements of the four processors in the system.

The system architect must pay careful attention to all such potential mismatches in system design. Otherwise, the sustained performance which the system can deliver can only equal the performance of the slowest part of the system—i.e. the bottleneck.

While this is a simple example, it illustrates the key challenge facing system designers. It is clear that, in the above system, if *processor performance* is improved by, say, 20%, we may not see a matching improvement in *system performance*, because the performance bottleneck in the system is the relatively slower processor-memory bus. In this particular case, a better investment for increased system performance could be (a) faster processor-memory bus, and (b) improved cache memory with each processor, i.e. one with better hit rate—which reduces contention for the processor-memory bus.

12.2 BASIC DESIGN ISSUES

Instruction pipeline and cache memory (or multi-level cache memories) hide the memory access latencies of instruction execution. With multiple functional units within the processor, *superscalar* instruction execution rates—greater than one per processor clock cycle—can be targeted, using multiple issue pipeline architecture. The aim is that the enormous processing power made possible by VLSI technology must be utilized to the full, ideally with each functional unit producing a result in every clock cycle. For this, the processor must also have data paths of requisite bandwidth—within the processor, to the memory and I/O subsystems, and to other processors in a multiprocessor system.

With a single processor chip today containing a billion (10^9) or more transistors, system design is not possible in the absence of a target application. For example, is a processor being designed for intensive scientific number-crunching, a commercial server, or for desktop applications?

One key design choice which appears in such contexts is the following.

Should the primary design emphasis be on:

(a) exploiting fully the parallelism present in a single instruction stream, or

(b) supporting multiple instruction streams on the processor in multi-core and/or multi-threading mode?

This design choice is also related to the depth of the instruction pipeline. In general, designs which aim to maximize the exploitation of instruction level parallelism need deeper pipelines; up to a point, such designs may support higher clock rates. But, beyond a point, deeper pipelines do not necessarily provide higher net throughput, while power consumption rises rapidly with clock rate, as we shall also discuss in Chapter 13.

Let us examine the trade-off involved in this context in a simplified way:

$$\text{total chip area} = \text{number of cores} \times \text{chip area per core}$$

or

$$\text{total transistor count} = \text{number of cores} \times \text{transistor count per core}$$

Within a processor, a set of instructions are in various stages of execution at a given time—within the pipeline stages, functional units, operation buffers, reservation stations, and so on. Recall that functional units themselves may also be internally pipelined. Therefore machine instructions are not in general executed in the order in which they are stored in memory, and all instructions under execution must be seen as 'work in progress'.

As we shall see, to maintain the work flow of instructions within the processor, a superscalar processor makes use of *branch prediction*—i.e. the result of a conditional branch instruction is predicted even before the instruction executes—so that instructions from the predicted branch can continue to be processed, without causing pipeline stalls. The strategy works provided fairly good branch prediction accuracy is maintained.

But we shall assume that instructions are *committed* in order. Here *committing* an instruction means that the instruction is no longer 'under execution'—the processor state and program state reflect the completion of all operations specified in the instruction.

Thus we assume that, at any time, the set of committed instructions correspond with the program order of instructions and the conditional branches actually taken. Any hardware exceptions generated within the processor must reflect the processor and program state resulting from instructions which have already committed.

PROBLEM DEFINITION

Let us now focus our attention on the execution of machine instructions from a single sequential stream. The instructions are stored in main memory in program order, from where they must be fetched into the processor, decoded, executed, and then committed in program order. In this context, we must address the problem of detecting and exploiting the parallelism which is implicit within the instruction stream.

We need a prototype instruction for our processor. We assume that the processor has a *load-store* type of instruction set, which means that all arithmetic and logical operations are carried out on operands which are present in programmable registers. Operands are transferred between main memory and registers by *load* and *store* instructions only.

We assume a three-address instruction format, as seen on most RISC processors, so that a typical instruction for arithmetic or logical operation has the format:

opcode operand-1 operand-2 result

Data transfer instructions have only two operands—source and destination registers; *load* and *store* instructions to/from main memory specify one operand in the form of a memory address, using an available addressing mode. Effective address for *load* and *store* is calculated at the time of instruction execution.

Conditional branch instructions need to be treated as a special category, since each such branch presents two possible continuations of the instruction stream. Branch decision is made only when the instruction executes; at that time, if instructions from the branch-not-taken are in the pipeline, they must be *flushed*. But pipeline flushes are costly in terms of lost processor clock cycles. The payoff of branch prediction lies in the fact that correctly predicted branches allow the detection of parallelism to stretch across two or more basic blocks of the program, without pipeline stalls. It is for this reason that branch prediction becomes an essential technique in exploiting instruction level parallelism.

Limits to detecting and exploiting instruction level parallelism are imposed by *dependences* between instructions. After all, if N instructions are completely independent of each other, they can be executed in parallel on N functional units—if N functional units are available—and they may even be executed in arbitrary order.

But in fact dependences amongst instructions are a central and essential part of program logic. A dependence specifies that instruction I_k must wait for instruction I_j to complete. Within the instruction pipeline, such a dependence may create a *hazard* or *stall*—i.e. lost processor clock cycles while I_k waits for I_j to complete.

Data Dependences

Assume that instruction I_k follows instruction I_j in the program. *Data dependence* between I_j and I_k means that both access a common operand. For the present discussion, let us assume that the common operand of I_j and I_k is in a programmable register. Since each instruction either reads or writes an operand value, accesses by I_j and I_k to the common register can occur in one of four possible ways:

Read by I_k after read by I_j
Read by I_k after write by I_j
Write by I_k after read by I_j
Write by I_k after write by I_j

Data Dependences

Assume that instruction I_k follows instruction I_j in the program. *Data dependence* between I_j and I_k means that both access a common operand. For the present discussion, let us assume that the common operand of I_j and I_k is in a programmable register. Since each instruction either reads or writes an operand value, accesses by I_j and I_k to the common register can occur in one of four possible ways:

Read by I_k after read by I_j
Read by I_k after write by I_j
Write by I_k after read by I_j
Write by I_k after write by I_j

Of these, the first pattern of register access does not in fact create a dependence, since the two instructions can read the common value of the operand in any order.

The other three patterns of operand access do create dependences amongst instructions. Based on the underlined words shown above, these are known as *read after write* (RAW) dependence, *write after read* (WAR) dependence, and *write after write* (WAW) dependence, respectively.

Sometimes we need to show dependences between instructions using graphical notation. We shall use small circles to represent instructions, and double line arrows between two circles to denote dependences. The instruction at the head of the arrow is dependent on the instruction at the tail; if necessary, the type of dependence between instructions may be shown by appropriate notation next to the arrow. A missing arrow between two instructions will mean explicit absence of dependence.

Single line arrows will be used between instructions when we wish to denote program order without any implied dependence or absence of dependence.

Figure 12.2 illustrates this notation.

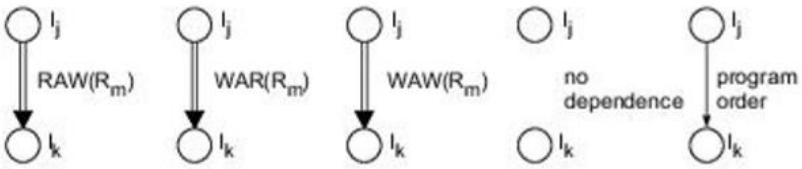

Fig. 12.2 Dependences shown in graphical notation (R_m indicates register)

When dependences between multiple instructions are thus depicted, the result is a *directed graph* of dependences. A *node* in the graph represents an instruction, while a *directed edge* between two nodes represents a dependence.

Often dependences are thus depicted in a *basic block* of instructions— i.e. a sequence of instructions with entry only at the first instruction, and exit only at the last instruction of the sequence. In such cases, the graph of dependences becomes a *directed acyclic graph*, and the dependences define a *partial order* amongst the instructions.

Part (a) of Fig. 12.3 shows a *basic block* of six instructions, denoted I_1 through I_6 in program order. Entry to the basic block may be from one of multiple points within the program; continuation after the basic block would be at one of several points, depending on the outcome of conditional branch instruction at the end of the block.

Part (b) of the figure shows a possible pattern of dependences as they may exist amongst these six instructions. For simplicity, we have not shown the type of each dependence, e.g. RAW(R_3), etc. In the partial order, we see that several pairs of instructions—such as (I_1, I_2) and (I_3, I_4)—are not related by any dependence. Therefore, amongst each of these pairs, the instructions may be executed in any order, or in parallel.

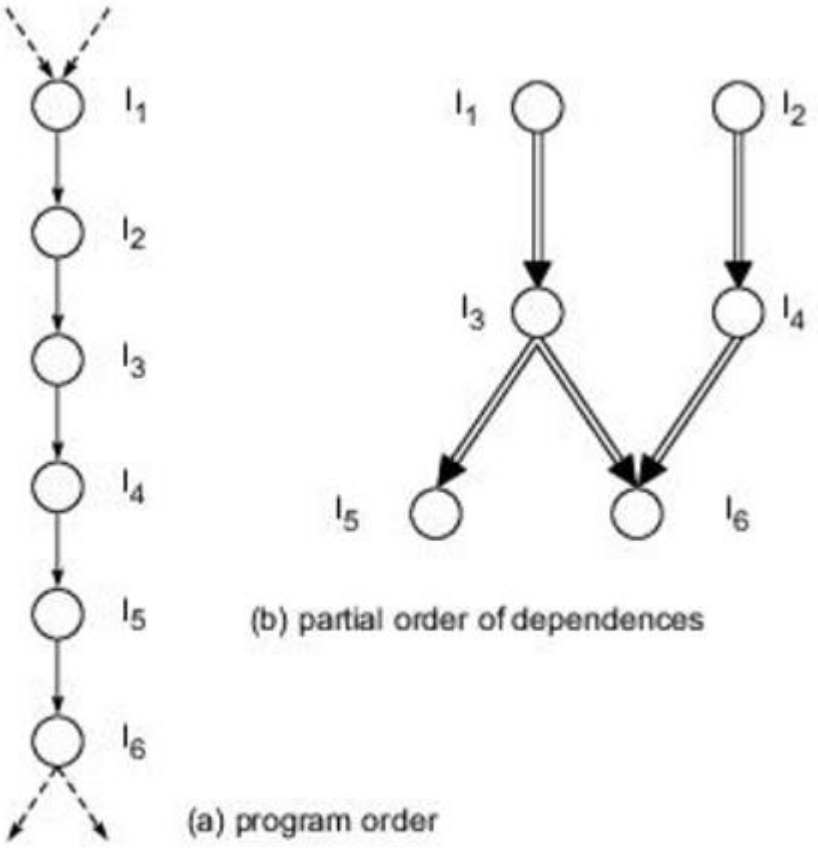

Fig. 12.3 A basic block of six instructions

Consider once again the pattern of dependences shown in Fig. 12.3(b). If the processor is capable of completing two (or more) instructions per clock cycle, and if no pipeline stalls are caused by the dependences shown, then clearly the six instructions can be completed in three consecutive processor clock cycles. Instruction latency, from fetch to commit stage, will of course depend on the depth of the pipeline.

Control Dependences In typical application programs, *basic blocks* tend to be small in length, since about 15% to 20% instructions in programs are branch and jump instructions, with indirect jumps and *returns* from procedure calls also included in the latter category. Because of typically small sizes of basic blocks in programs, the amount of instruction level parallelism which can be exploited in a single basic block is limited.

Assume that instruction I_j is a conditional branch and that, whether another instruction I_k executes or not depends on the outcome of the conditional branch instruction I_j. In such a case, we say that there is a *control dependence* of instruction I_k on instruction I_j.

Let us assume that a processor has instruction pipeline of depth eight, and that the designers target superscalar performance of four instructions completed in every clock cycle. Assuming no pipeline stalls, the number of instructions in the processor at any one time—in its various pipeline stages and functional units—would be $4 \times 8 = 32$.

If 15% to 20% of these instructions are branches and jumps, then the execution of subsequent instructions within the processor would be held up pending the resolution of conditional branches, procedure returns, and so on—causing frequent pipeline stalls.

This simple calculation shows the potential adverse impact of conditional branches on the performance of a superscalar processor. The key question here is: How can the processor designer mitigate the adverse impact of such *control dependences* in a program?

Answer: Using some form of *branch and jump prediction*—i.e. predicting early and correctly (most of the time) the results of conditional branches, indirect jumps, and procedure returns. The aim is that, for every correct prediction made, there should be no lost processor clock cycles due to the conditional branch, indirect jump, or procedure return. For every mis-prediction made, there would be the cost of flushing the pipeline of instructions from the wrong continuation after the conditional branch or jump.

Example 12.2 Impact of successful branch prediction

Assume that we have attained 93% accuracy in branch prediction in a processor with eight pipeline stages. Assume also that the mis-prediction penalty is 4 processor clock cycles to flush the instruction pipeline. What is the performance gain from such a branch prediction strategy?

Recall that the expected cost of a random variable X is given by $\Sigma x_i p_i$, where x_i are possible values of X, and p_i are the respective probabilities. In our case, the probability of a correct branch is 0.93, and the corresponding cost is zero; the probability of a wrong branch is 0.07, and the corresponding cost is 2. Thus the expected cost of a conditional branch instruction is $0.07 \times 4 = 0.28$ clock cycle i.e. much less than one clock cycle.

As a primitive form of branch prediction, the processor designer could assume that a conditional branch is always taken, and continue processing the instructions which follow at the target address. Let us assume that this simple strategy works 80% of the time; then the expected cost of a conditional branch is $0.2 \times 4 = 0.8$ clock cycles.

Suppose that not even this primitive form of branch prediction is used. Then the pipeline must stall until the result of every branch condition, and the target address of every indirect jump and procedure return, is known; only then can the processor proceed with the correct continuation within the program. If we assume that in this case the pipeline stalls over half the total number of stages, then the number of lost clock cycles is 4 for every conditional branch, indirect jump and procedure return instruction.

Considering that 15% to 20% of the instructions in a program are branches and jumps, the difference in cost between 0.28 clock cycle and 4 clock cycles per branch instruction is huge, underlining the importance of branch prediction in a superscalar processor.

Example 12.3 Resource dependence

Consider a simple pipelined processor with only one floating point multiplier, which is not internally pipelined and takes three processor clock cycles for each multiplication. Assume that several independent floating point multiply instructions follow each other in the instruction stream in a single basic block under execution.

Clearly, while the processor is executing these multiply instructions, it cannot for that duration get even one instruction completed in every clock cycle. Therefore pipeline stalls are inevitable, caused by the absence of sufficient floating point multiply capability within the processor. In fact, for the duration of these consecutive multiply operations, the processor will only complete one instruction in every three clock cycles.

We have assumed the instructions to be independent of each other, and in a single basic block—i.e. there are no conditional branches within the sequence. Thus there is no data dependence or control dependence amongst these instructions. What we have here is *resource dependence*, i.e. all the instructions depend on the resource which has not been provided to the extent it is needed for the given workload on the processor.

We can say that there is an imbalance in this processor between the floating point capability provided and the workload which is placed on it. Such imbalances in system resources usually have adverse performance impact. Recall that Example 12.1 above and the related discussion illustrated this same point in another context.

A *resource dependence* which results in a pipeline stall can arise for access to any processor resource—functional unit, data path, register bank, and so on[2]. We can certainly say that such resource dependences will arise if hardware resources provided on the processor do not match the needs of the executing program.

Now that we have seen the various types of dependences which can occur between instructions in an executing program, the problem of detecting and exploiting instruction level parallelism can finally be stated in the following manner:

Problem Definition Design a superscalar processor to detect and exploit the maximum degree of parallelism available in the instruction stream—i.e. execute the instructions in the smallest possible number of processor clock cycles—by handling correctly the data dependences, control dependences and resource dependences within the instruction stream.

12.4 MODEL OF A TYPICAL PROCESSOR

We assume a processor with *load-store* instruction set architecture and a set of programmable registers as seen by the assembly language programmer or the code generator of a compiler. Whether these registers are bifurcated into separate sets of integer and floating point registers is not important for us at present, nor is the exact number of these registers.

To support parallel access to instructions and data at the level of the fastest cache, we assume that L1 cache is divided into instruction cache and data cache, and that this split L1 cache supports single cycle access for instructions as well as data. Some processors may have an *instruction buffer* in place of L1 instruction cache; for the purposes of this section, however, the difference between them is not important.

The first three pipeline stages on our prototype processor are *fetch*, *decode* and *issue*.

Following these are the various functional units of the processor, which include integer unit(s), floating point unit(s), load/store unit(s), and other units as may be needed for a specific design—as we shall see when we discuss specific design techniques.

Let us assume that our superscalar processor is designed for k instruction issues in every processor clock cycle. Clearly then the *fetch*, *decode* and *issue* pipeline stages, as well as the other elements of the processor, must all be designed to process k instructions in every clock cycle.

On multiple issue pipelines, *issue* stage is usually separated from *decode* stage. One reason for thus increasing a pipeline stage is that it allows the processor to be driven by a faster clock. *Decode* stage must be seen as preparation for instruction *issue* which—by definition—can occur only if the relevant functional unit in the processor is in a state in which it can accept one more operation for execution. As a result of the *issue*, the operation is handed over to the functional unit for execution.

The process of issuing instructions to functional units also involves *instruction scheduling* [3]. For example, if instruction I_j cannot be issued because the required functional unit is not free, then it may still be possible to issue the next instruction I_{j+1}—provided that no dependence between the two prohibits issuing instruction I_{j+1}.

When instruction scheduling is specified by the compiler in the machine code it generates, we refer to it as *static scheduling*. In theory, static scheduling should free up the processor hardware from the complexities of instruction scheduling; in practice, though, things do not quite turn out that way, as we shall see in the next section.

If the processor control logic schedules instruction *on the fly*—taking into account inter-instruction dependences as well as the state of the functional units—we refer to it as *dynamic scheduling*. Much of the rest of this chapter is devoted to various aspects and techniques of dynamic scheduling. Of course the basic aim in both types of scheduling—static as well as dynamic—is to maximize the instruction level parallelism which is exploited in the executing sequence of instructions.

As we have seen, at one time multiple instructions are in various stages of execution within the processor. But *processor state* and *program state* need to be maintained which are consistent with the program order of completed instructions. This is important from the point of view of preserving the semantics of the program.

Therefore, even with multiple instructions executing in parallel, the processor must arrange the results of completed instructions so that their sequence reflects program order. One way to achieve this is by using a *reorder buffer*, shown in Fig.12.4, which allows instructions to be *committed* in program order, even if they execute in a different order; we shall discuss this point in some more detail in Section 12.7.

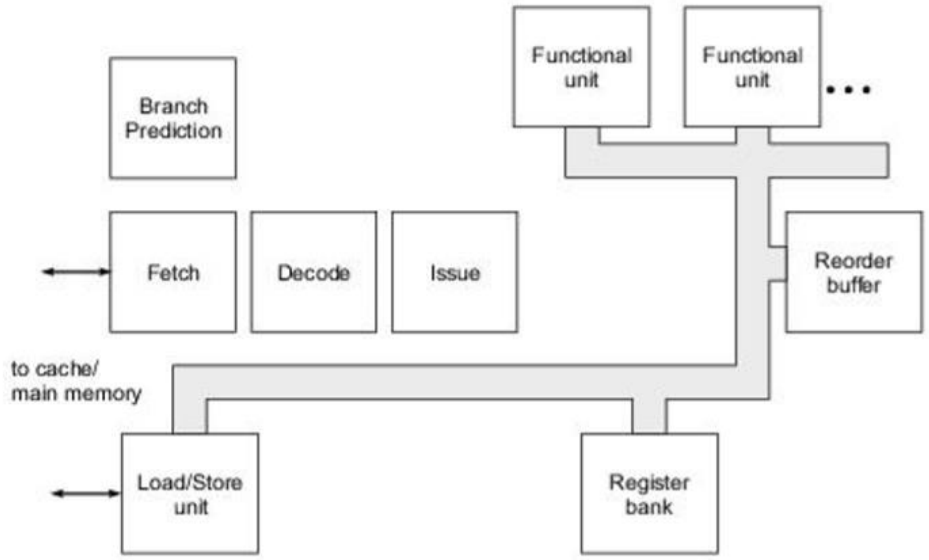

Fig. 12.4 Processor design with reorder buffer

If instructions are executed on the basis of predicted branches, before the actual branch outcome is available, we say that the processor performs *speculative execution*. In such cases, the reorder buffer will need to be cleared—wholly or partly—if the actual branch result indicates that speculation has occurred on the basis of a mis-prediction.

Functional units in the processor may themselves be internally pipelined; they may also be provided with *reservation stations*, which accept operations issued by the *issue* stage of the instruction pipeline. A functional unit performs an operation when the required operands for it are available in the reservation station. For the purposes of our discussion, memory *load-store unit(s)* may also be treated as functional units, which perform their functions with respect to the cache/memory subsystem.

Figure 12.5 shows a processor design in which functional units are provided with *reservation stations*. Such designs usually also make use of *operand forwarding* over a *common data bus* (CDB), with *tags* to identify the source of data on the bus. Such a design also implies *register renaming*, which resolves RAW and WAW dependences.

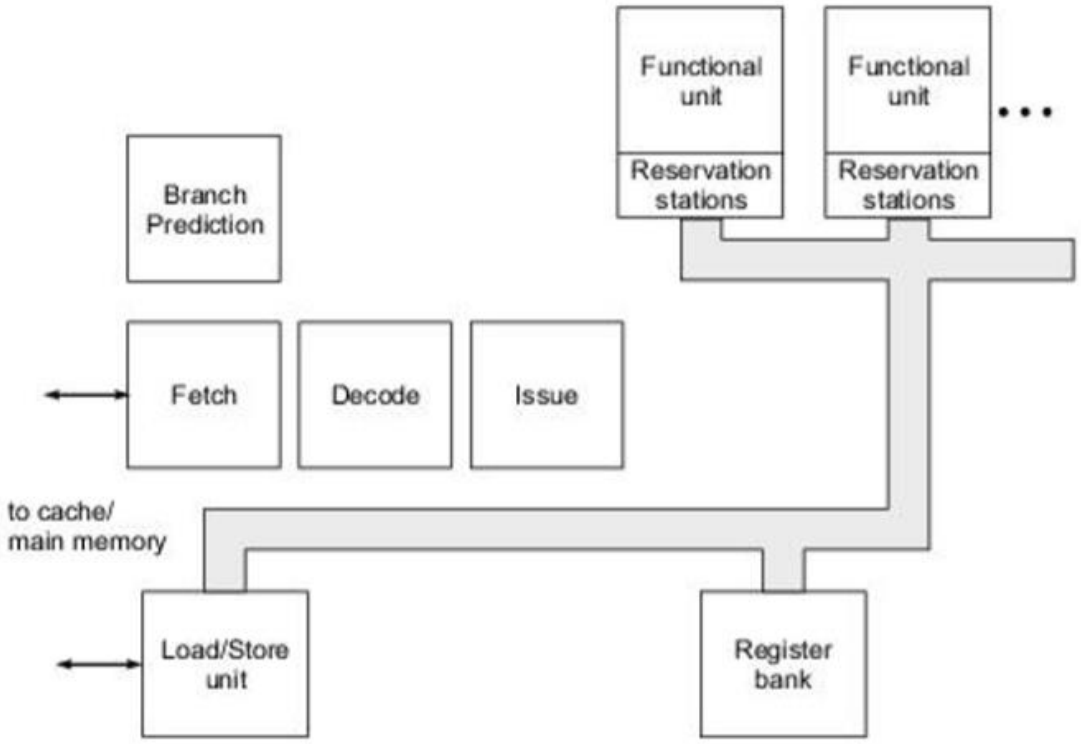

Fig. 12.5 Processor design with reservation stations on functional units

Data paths connecting the various elements within the processor must be provided so that no *resource dependences*—and consequent pipeline stalls—are created for want of a data path. If k instructions are to be completed in every processor clock cycle, the data paths within the processor must support the required data transfers in each clock cycle.

The processor designer must select the architectural components to be included in the processor—for example a *reorder buffer* of a particular type, a specific method of *operand forwarding*, a specific method of *branch prediction*, and so on. The designer must also specify fully the algorithms which will govern the working of the selected architectural components. These algorithms are very similar to the algorithms we write in higher level programming languages, and are written using similar languages. These algorithms specify the control logic that would be needed for the processor, which would be finally realized in the form of appropriate digital logic circuits.

Given the complexity of modern systems, the task of translating algorithmic descriptions of processor functions into digital logic circuits can only be carried out using very sophisticated VLSI design software. Such software offers a wide range of functionality; *simulation* software is used to verify the correctness of the selected algorithm; *logical design* software translates the algorithm into a digital circuit; *physical design* software translates the logical circuit design into a physical circuit which can be built using VLSI, while ***design verification*** software verifies that the physical design does not violate any constraints of the underlying circuit fabrication technology.

All the architectural elements and control logic which is being described in this chapter can thus be translated into a physical design and then realized in VLSI. This is how processors and other digital systems are designed and built today. For our purposes in this chapter, however, it is not necessary to go into the details of how the required circuits and control logic are to be realized in VLSI.

We take the view that the architect decides *what* is to be designed, and then the circuit designer designs and realizes the circuit accordingly. In other words, our subject matter is restricted to the functions of the architect, and does not extend to circuit design—i.e. to the question of *how* a particular function is to be realized in VLSI. We assume that any required control logic which can be clearly specified can be implemented.

12.5 COMPILER-DETECTED INSTRUCTION LEVEL PARALLELISM

In the process of translating a sequential source program into machine language, the compiler performs extensive syntactic and semantic analysis of the source program. Therefore computer scientists have considered carefully the question of whether the compiler can uncover the instruction level parallelism which is implicit in the program. As we shall see, there are several ways in which the compiler can contribute to the exploitation of implicit instruction level parallelism.

One relatively simple technique which the compiler can employ is known as *loop unrolling*, by which independent instructions from multiple successive iterations of a loop can be made to execute in parallel.

Unrolling means that the body of the loop is repeated n times for n successive values of the control variable—so that one iteration of the transformed loop performs the work of n iterations of the original loop.

Example 12.4 Loop unrolling

Consider the following body of a loop in a user program, where all the variables except the loop control variable i are assumed to be floating point:

```
for i = 0 to 58 do
    c[i] = a[i]*b[i] - p*d[i];
```

Now suppose that machine code is generated by the compiler as though the original program had been written as:

```
for j = 0 to 52 step 4 do
{
   c[j] = a[j]*b[j] - p*d[j];
   c[j+1] = a[j+1]*b[j+1] - p*d[j+1];
   c[j+2] = a[j+2]*b[j+2] - p*d[j+2];
   c[j+3] = a[j+3]*b[j+3] - p*d[j+3];
}
c[56] = a[56]*b[56] - p*d[56];
c[57] = a[57]*b[57] - p*d[57];
c[58] = a[58]*b[58] - p*d[58];
```

Note carefully the values of loop variable `j` in the transformed loop.

The reader may verify, without too much difficulty, that the two program fragments are equivalent, in the sense that they perform the same computation. Of course the compiler does not transform one source program into another—it simply produces machine code corresponding to the second version, with the *unrolled* loop.

In the unrolled program fragment, the loop contains four independent instances of the original loop body—indeed this is the meaning of *loop unrolling*. Suppose machine code corresponding to the second program fragment is executing on a processor. Then clearly—if the processor has sufficient floating point arithmetic resources—instructions from the four loop iterations can be in progress in parallel on the various functional units.

In the simple example above, the loop control variable in the original program goes from 0 to 58—i.e. its initial and final values are both known at compile time. If, on the other hand, the loop control values are not known at compile time, the compiler must generate code to calculate at run-time the control values for the unrolled loop.

Note that loop unrolling by the compiler does not in itself involve the detection of instruction level parallelism. But loop unrolling makes it possible for the compiler or the processor hardware to exploit a greater degree of instruction level parallelism. In Example 12.4, since the basic block making up the loop body becomes longer, it becomes possible for the compiler or processor to find a greater degree of parallelism amongst the instructions across the unrolled loop iterations.

Can the compiler also do the additional work of actually scheduling machine instructions on the hardware resources available on the processor? Or must this scheduling be necessarily performed *on the fly* by the processor control logic?

When the compiler schedules machine instructions for execution on the processor, the form of scheduling is known as *static scheduling*. As against this, instruction scheduling carried out by the processor hardware *on the fly* is known as *dynamic scheduling*, which has been introduced in Chapter 6 and will be discussed further later in this chapter.

If the compiler is to schedule machine instructions, then it must perform the required dependence analysis amongst instructions. This is certainly possible, since the compiler has access to full semantic information obtained from the original source program.

Example 12.5 Dependence across loop iterations

Consider the following loop in a source program, which appears similar to the loop seen in the previous example, but has a crucial new dependence built into it:

```
for i = 0 to 58 do
    c[i] = a[i]*b[i] - p*c[i-1];
```

Now the value calculated in the i^{th} iteration of the loop makes use of the value `c[i-1]` calculated in the previous iteration. This does not mean that the modified loop cannot be unrolled, but only that extra care should be taken to account for the dependence.

Dependences amongst references to simple variables, or amongst array elements whose index values are known at compile time (as in the two examples seen above), can be analyzed relatively easily at compile time.

But when pointers are used to refer to locations in memory, or when array index values are known only at run-time, then clearly dependence analysis is not possible at compile time. Therefore processor hardware must provide support at run-time for *alias analysis*—i.e. based on the respective effective addresses, to determine whether two memory accesses for read or write operations refer to the same location.

Recall that conventional machine instructions specify one operation each—e.g. *load*, *add*, *multiply*, and so on. As opposed to this, multi-operation instructions would require a larger number of bits to encode. Therefore processors with this type of instruction word are said to have *very long instruction word* (VLIW).

A little further refinement of this concept brings us to the so-called *explicitly parallel instruction computer* (EPIC). The EPIC instruction format can be more flexible than the fixed format of multi-operation VLIW instruction; for example, it may allow the compiler to encode explicitly dependences between operations.

The aim behind VLIW and EPIC processor architecture is to assign to the compiler primary responsibility for the parallel exploitation of plentiful hardware resources of the processor. In theory, this would simplify the processor hardware, allowing for increased aggregate processor throughput. Thus this approach would, in theory, provide a third alternative to the RISC and CISC styles of processor architecture.

12.6 OPERAND FORWARDING

We know that a superscalar processor offers opportunities for the detection and exploitation of instruction level parallelism—i.e. potential parallelism which is present within a single instruction stream. Exploitation of such parallelism is enhanced by providing multiple functional units and by other techniques that we shall study. True data dependences between instructions must of course be respected, since they reflect program logic. On the other hand, two independent instructions can be executed in parallel—or even out of sequence—if that results in better utilization of processor clock cycles.

We now know that pipeline *flushes* caused by conditional branch, indirect jump, and procedure return instructions lead to degradation in performance, and therefore attempts must be made to minimize them; similarly pipeline *stalls* caused by data dependences and cache misses also have adverse impact on processor performance.

Therefore the strategy should be to minimize the number of pipeline stalls and flushes encountered while executing an instruction stream. In other words, we must minimize wasted processor clock cycles within the pipeline and also, if possible, within the various functional units of the processor.

In this section, we take a look at a basic technique known as *operand forwarding*, which helps in reducing the impact of true data dependences in the instruction stream. Consider the following simple sequence of two instructions in a running program:

```
ADD       R1, R2, R3
SHIFTR    #4, R3, R4
```

The result of the ADD instruction is stored in destination register R3, and then shifted right by four bits in the second instruction, with the shifted value being placed in R4. Thus, there is a simple RAW *dependence* between the two instructions—the output of the first is required as input operand of the second.

In terms of our notation, this RAW dependence appears as shown in Fig. 12.6, in the form of a graph with two nodes and one edge.

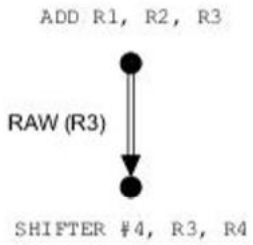

Fig. 12.6 RAW dependence between two instructions

In a pipelined processor, ideally the second instruction should be executed one stage—and therefore one clock cycle—behind the first. However, the difficulty here is that it takes one clock cycle to transfer ALU output to destination register R3, and then another clock cycle to transfer the contents of register R3 to ALU input for the right shift. Thus a total of two clock cycles are needed to bring the result of the first instruction where it is needed for the second instruction. Therefore, as things stand, the second instruction above cannot be executed just one clock cycle behind the first.

This sequence of data transfers has been illustrated in Fig. 12.7 (a). In clock cycle T_k, ALU output is transferred to R3 over an internal data path. In the next clock cycle T_{k+1}, the content of R3 is transferred to ALU input for the right shift. When carried out in this order, clearly the two data transfer operations take two clock cycles.

But note that the required two transfers of data can be achieved in only one clock cycle if ALU output is sent to both R3 and ALU input in the same clock cycle—as illustrated in Fig. 12.7 (b). In general, if X is to be copied to Y, and in the next clock cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.

If this is done in the above sequence of instructions, the second instruction can be just one clock cycle behind the first, which is a basic requirement of an instruction pipeline.

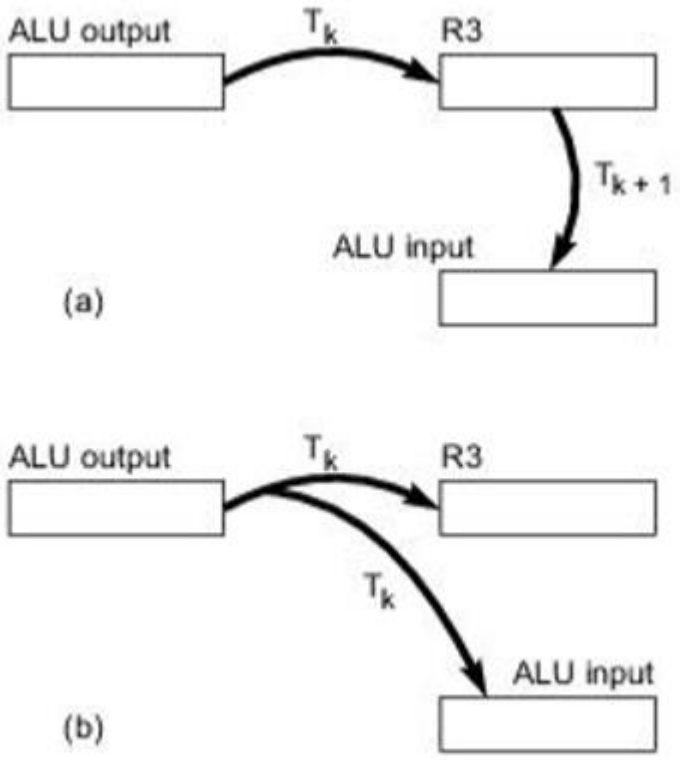

Fig. 12.7 Two data transfers (a) in sequence and (b) in parallel

In technical terms, this type of an operation within a processor is known as *operand forwarding*. Basically this means that, instead of performing two or more data transfers from a common source one after the other, we perform them in parallel. This can be seen as parallelism at the level of elementary data transfer operations within the processor. To achieve this aim, the processor hardware must be designed to detect and exploit *on the fly* all such opportunities for saving clock cycles. We shall see later in this chapter one simple and elegant technique for achieving this aim.

The benefits of such a technique are easy to see. The wait within a functional unit for its operand becomes shorter because, as soon as it is available, the operand is sent in one clock cycle, over the common data bus, to every destination where it is needed. We saw in the above example that thereby the common data bus remained occupied for one clock cycle rather than two clock cycles. Since this bus itself is a key hardware resource, its better utilization in this way certainly contributes to better processor performance.

The above reasoning applies even if there is an intervening instruction between ADD and SHIFTR. Consider the following sequence of instructions:

```
ADD       R1, R2, R3
SUB       R5, R6, R7
SHIFTR    #4, R3, R4
```

SHIFTR must be executed after ADD, in view of the RAW dependence. But there is no such dependence between SUB and any of the other two instructions, which means that SUB can be executed in program order, or before ADD, or after SHIFTR.

If SUB is executed in program order, then even without operand forwarding between ADD and SHIFTR, no processor clock cycle is lost, since SHIFTR does not directly follow ADD. But now suppose SUB is executed either before ADD, or after SHIFTR. In both these cases, SHIFTR directly follows ADD, and therefore operand forwarding proves useful in saving a processor cycle, as we have seen above.

Figure 12.8 shows the dependence graph of these three instructions. Since there is only one dependence in this instance amongst the three instructions, the graph in the figure has three nodes and only one edge.

But *why* should SUB be executed in any order other than program order?

The answer can only be this: to achieve better utilization of processor clock cycles. For example, if for some reason ADD cannot be executed in a given clock cycle, then the processor may well decide to execute SUB before it.

Therefore the processor must make *on the fly* decisions such as

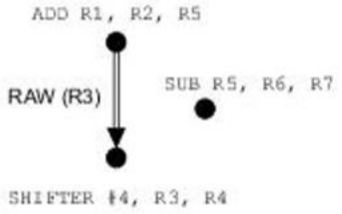

Fig. 12.8 Dependence graph of three instructions

(i) transferring ALU output in parallel to both R3 and ALU input, and/or
(ii) out of order execution of the instruction SUB.

This implies that the control logic of the processor must detect any such possibilities and generate the required control signals. This is in fact what is needed to implement *dynamic scheduling* of machine instructions within the processor.

Of course, to achieve performance speed-up through dynamic scheduling, considerable complexity must be added to processor control logic—but that is a price which must be paid for exploiting instruction level parallelism in the sequence of executing instructions; the complexity in achieving superscalar performance would of course be greater.

Machine instructions of a typical processor can be classified into *data transfer* instructions, *arithmetic and logic* instructions, *comparison* instructions, *transfer of control* instructions, and other miscellaneous instructions.

Of these, only the second group of instructions—i.e. arithmetic and logic instructions—actually alter the values of their operands. The other groups of instructions involve only transfers of data within the processor, between the processor and main memory, or between the processor and an I/O adapter.

Arithmetic and logic instructions are basically functions to be computed—either unary functions of the form $y = f(x)$, or binary functions of the form $y = f(x_1,x_2)$. And these computations are carried out by functional units such as *arithmetic and logic unit* (ALU), *floating point unit* (FPU), and so on. But even to get any computations done by these functional units, we need (i) transfer of operands to the inputs of functional units, and (ii) transfer of results back to registers or to the reorder buffer.

From the above arguments, it should be clear that data transfers make up a large proportion of the work of any processor. The need to fully utilize available hardware resources forces designers to pay close attention to the data transfers required not only for a single executing instruction, but also across multiple instructions. In this context, operand forwarding can be seen as a potentially powerful technique to reduce the number of clock cycles spent in carrying out the required data transfers within the processor.

In Fig. 12.4 and Fig. 12.5, we have not shown details of the data paths connecting the various elements within the processor. This is intentional, because the nature and number of data paths, their widths, their access mechanisms, *et cetera*, must be designed to be consistent with (i) the various hardware resources provided within the processor, and (ii) the target performance of the processor. Details of the data paths cannot be pinned down at an early stage, when the rest of the design is not yet completed.

We have discussed earlier a basic point related to any system performance: *there should be no performance bottlenecks in the system.* Clearly therefore the system of data paths provided within the processor should also not become a performance limiting element. A multiple issue processor targets $k > 1$ instruction issues per processor clock cycle. Hence the demands made on each of the elements of the processor—including cache memories, functional units, and internal data paths—would be k times greater.

12.7 REORDER BUFFER

The *reorder buffer* as a processor element was introduced and discussed briefly in Section 12.4. Since instructions execute in parallel on multiple functional units, the reorder buffer serves the function of bringing completed instructions back into an order which is consistent with program order. Note that instructions may *complete* in an order which is not related to program order, but must be *committed* in program order.

At any time, *program state* and *processor state* are defined in terms of instructions which have been committed—i.e. their results are reflected in appropriate registers and/or memory locations. The concepts of program state and processor state are important in supporting context switches and in providing precise exceptions.

Entries in the reorder buffer are completed instructions, which are queued in program order. However, since instructions do not necessarily complete in program order, we also need a flag with each reorder buffer entry to indicate whether the instruction in that position has completed.

Figure 12.9 shows a reorder buffer of size eight. Four fields are shown with each entry in the reorder buffer—instruction identifier, value computed, program-specified destination of the value computed, and a flag indicating whether the instruction has completed (i.e. the computed value is available).

In Fig. 12.9, the head of queue of instructions is shown at the top, arbitrarily labeled as instr[i]. This is the instruction which would be committed next—if it has completed execution. When this instruction commits, its result value is copied to its destination, and the instruction is then removed from the reorder buffer. The next instruction to be issued in the *issue* stage of the instruction pipeline then joins the reorder buffer at its tail.

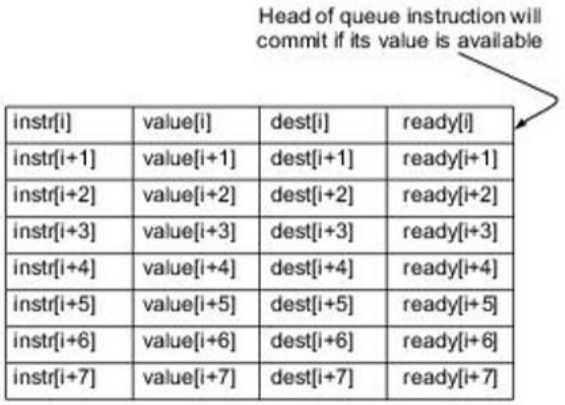

Fig. 12.9 Entries in a reorder buffer of size eight

If the instruction at the head of the queue has not completed, and the reorder buffer is full, then further issue of instructions is held up—i.e. the pipeline stalls—because there is no free space in the reorder buffer for one more entry.

The result value of any other instruction lower down in the reorder buffer, say value[i+k], can also be used as an input operand for a subsequent operation—provided of course that the instruction has completed and therefore its result value is available, as indicated by the corresponding flag ready[i+k]. In this sense, we see that the technique of *operand forwarding* can be combined with the concept of the reorder buffer.

It should be noted here that operands at the input latches of functional units, as well as values stored in the reorder buffer on behalf of completed but uncommitted instructions, are simply 'work in progress'. These values are not reflected in the state of the program or the processor, as needed for a context switch or for exception handling.

We now take a brief look at how the use of reorder buffer addresses the various types of dependences in the program.

(i) Data Dependences A RAW dependence—i.e. true data dependence—will hold up the execution of the dependent instruction if the result value required as its input operand is not available. As suggested above, operand forwarding can be added to this scheme to speed up the supply of the needed input operand as soon as its value has been computed.

WAR and WAW dependences—i.e. anti-dependence and output dependence, respectively—also hold up the execution of the dependent instruction and create a possible pipeline stall. We shall see below that the technique of *register renaming* is needed to avoid the adverse impact of these two types of dependences.

(ii) Control Dependences Suppose the instruction(s) in the reorder buffer belong to a branch in the program which should not have been taken—i.e. there has been a mis-predicted branch. Clearly then the reorder buffer should be flushed along with other elements of the pipeline. Therefore the performance impact of control dependences in the running program is determined by the accuracy of branch prediction technique employed. The reorder buffer plays no direct role in the handling of control dependences.

(iii) Resource Dependences If an instruction needs a functional unit to execute, but the unit is not free, then the instruction must wait for the unit to become free—clearly no technique in the world can change that. In such cases, the processor designer can aim to achieve at least this: if a subsequent instruction needs to use another functional unit which is free, then the subsequent instruction can be executed out of order.

However, the reorder buffer queues and commits instructions in program order. In this sense, therefore, the technique of using a reorder buffer does not address explicitly the resource dependences existing within the instruction stream; with multiple functional units, the processor can still achieve out of order completion of instructions.

In essence, the conceptually simple technique of reorder buffer ensures that if instructions as programmed can be carried out in parallel—i.e. if there are no dependences amongst them—then they are carried out in parallel. But nothing clever is attempted in this technique to resolve dependences. Instruction issue and commit are in program order; program state and processor state are correctly preserved.

We shall now discuss a clever technique which alleviates the adverse performance effect of WAR and WAW dependences amongst instructions.

REGISTER RENAMING

Traditional compilers allocate registers to program variables in such a way as to reduce the main memory accesses required in the running program. In programming language C, in fact, the programmer can even pass a hint to the compiler that a variable be maintained in a processor register.

Traditional compilers and assembly language programmers work with a fairly small number of programmable registers. The number of programmable registers provided on a processor is determined by either

(i) the need to maintain backward instruction compatibility with other members of the processor family, or

(ii) the need to achieve reasonably compact instruction encoding in binary. With sixteen programmable registers, for example, four bits are needed for each register specified in a machine instruction.

Amongst the instructions in various stages of execution within the processor, there would be occurrences of RAW, WAR and WAW dependences on programmable registers. As we have seen, RAW is true data dependence—since a value written by one instruction is used as an input operand by another. But a WAR or WAW dependence can be avoided if we have more registers to work with. We can simply remove such a dependence by getting the two instructions in question to use two different registers.

But we must also assume that the *instruction set architecture* (ISA) of the processor is fixed—i.e. we cannot change it to allow access to a larger number of programmable registers. Rather, our aim here is to explore techniques to detect and exploit instruction level parallelism using a given instruction set architecture.

Therefore the only way to make a larger number of registers available to instructions under execution within the processor is to make the additional registers *invisible* to machine language instructions. Instructions *under execution* would use these additional registers, even if instructions making up the machine language program stored in memory cannot refer to them.

Let us suppose that we have several such additional registers available, to which machine instructions of the running program cannot make any direct reference. Of course these machine instructions do refer to programmable registers in the processor—and thereby create the WAR and WAW dependences which we are now trying to remove.

For example, let us say that the instruction:

```
FADD      R1, R2, R5
```

is followed by the instruction:

```
FSUB      R3, R4, R5
```

Both these instructions are writing to register R5, creating thereby a WAW dependence—i.e. output dependence—on register R5. Clearly, any subsequent instruction should read the value written into R5 by FSUB, and not the value written by FADD. Figure 12.10 shows this dependence in graphical notation.

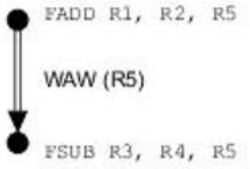

Fig. 12.10 WAW dependence

With additional registers available for use as these instructions execute, we have a simple technique to remove this output dependence.

Let FSUB write its output value to a register other than R5, and let us call that other register X. Then the instructions which use the value generated by FSUB will refer to X, while the instructions which use the value generated by FADD will continue to refer to R5. Now, since FADD and FSUB are writing to two different registers, the output dependence or WAW between them has been removed![5]

When FSUB *commits*, then the value in R5 should be updated by the value in X—i.e. the value computed by FSUB. Then the physical register X, which is not a program visible register, can be freed up for use in another such situation.

Note that here we have *mapped*—or *renamed*—R5 to X, for the purpose of storing the result of FSUB, and thereby removed the WAW dependence from the instruction stream. A pipeline stall will now not be created due to the WAW dependence.

In general, let us assume that instruction I_j writes a value into register R_k. At the time of instruction issue, we map this programmable register R_k onto a program invisible register X_m, so that when instruction I_j executes, the result is written into X_m rather than R_k. In this program invisible register X_m, the result value is available to any other instruction which is truly data dependent on I_j— i.e. which has RAW dependence on I_j.

If any instruction other than I_j is also writing into R_k, then that instance of R_k will be mapped into some other program invisible register X_n. This renaming resolves the WAW dependence between the two instructions involving R_k. When instruction I_j commits, the value in X_m is copied back into R_k, and the program invisible register X_m is freed up for reuse by another instruction.

A similar argument applies if I_j is reading the value in R_k, and a subsequent instruction is writing into R_k—i.e. there is a WAR dependence between them.

The technique outlined, which can resolve WAR and WAW dependences, is known as *register renaming*. Both these dependences are caused by a subsequent instruction writing into a register being used by a previous instruction. Such dependences do not reflect program logic, but rather the use of a limited number of registers.

Let us now consider a simple example of WAR dependence, i.e. of anti-dependence. The case of WAW dependence would be very similar.

Example 12.6 Register renaming and WAR dependence

Assume that the instructions:

```
FADD    R6, R7, R2
FADD    R2, R3, R5
```

are followed later in the program by the instruction:

```
FSUB    R1, R3, R2
```

The first FADD instruction is writing a value into R2, which the second FADD instruction is using, i.e. there is true data dependence between these two instructions. Let us assume that, when the first FADD instruction executes, R2 is mapped into program invisible register X_m.

The latter FSUB instruction is writing another value into R2. Clearly, the second FADD (and other intervening instructions before FSUB) should see the value in R2 which is written by the first FADD—and not the value written by FSUB. Figure 12.11 shows these two dependences in graphical notation.

Fig. 12.11 RAW and WAR dependences

With register renaming, it is a simple matter to resolve the WAR anti-dependence between the second FADD and FSUB.

As mentioned, let X_m be the program invisible register to which R2 has been mapped when the first FADD executes. This is then the remapped register to which the second FADD refers for its first data operand.

Let FSUB write its output to a program invisible register other than X_m, which we denote by X_n. Instructions which use the value written by FSUB refer to X_n, while instructions which use the value written by the first FADD refer to X_m.

The WAR dependence between the second FADD and FSUB is removed; but the RAW dependence between the two FADD instructions is respected via X_m.

When the first FADD commits, the value in X_m is transferred to R2 and program invisible register X_m is freed up; likewise, later when FSUB commits, the value in X_n is transferred to R2 and program invisible register X_m is freed up.

12.9 TOMASULO'S ALGORITHM

In the IBM 360 family of computer systems of 1960s and 1970s, model 360/91 was developed as a high performance system for scientific and engineering applications, which involve intensive floating point computations. The processor in this system was designed with multiple floating point units, and it made use of an innovative algorithm for the efficient use of these units. The algorithm was based on operand forwarding over a common data bus, with tags to identify sources of data values sent over the bus.

The algorithm has since become known as *Tomasulo's algorithm*, after the name of its chief designer[6]; what we now understand as *register renaming* was also an implicit part of the original algorithm.

Recall that, for register renaming, we need a set of program invisible registers to which programmable registers are re-mapped. Tomasulo's algorithm requires these program invisible registers to be provided with reservation stations of functional units.

Let us assume that the functional units are internally pipelined, and can complete one operation in every clock cycle. Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values. Note that the exact depth of this functional unit pipeline does not concern us for the present.

Figure 12.12 shows such a functional unit connected to the common data bus, with three reservation stations provided on it.

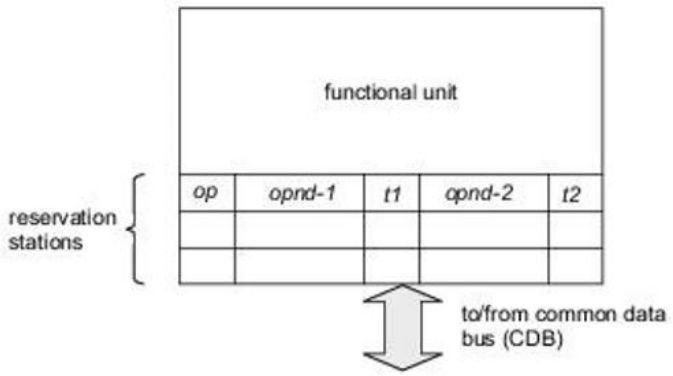

Fig. 12.12 Reservation stations provided with a functional unit

The various fields making up a typical reservation station are as follows:

op	operation to be carried out by the functional unit
opnd-1 & *opnd-2*	two operand values needed for the operation
t1 & *t2*	two source tags associated with the operands

When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle.

At the time of instruction issue, the reservation station is filled out with the operation code (*op*). If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station.

However, if the operand value is not available at the time of issue, the corresponding source tag (*t1* and/or *t2*) is copied into the reservation station. The source tag identifies the source of the required operand. As soon as the required operand value is available at its source—which would be typically the output of a functional unit—the data value is forwarded over the common data bus, along with the source tag. This value is copied into all the reservation station operand slots which have the matching tag.

Thus operand forwarding is achieved here with the use of tags. All the destinations which require a data value receive it in the same clock cycle over the common data bus, by matching their stored operand tags with the source tag sent out over the bus.

Example 12.7 Tomasulo's algorithm and RAW dependence

Assume that instruction I1 is to write its result into R4, and that two subsequent instructions I2 and I3 are to read—i.e. make use of—that result value. Thus instructions I2 and I3 are truly data dependent (RAW dependent) on instruction I1. See Fig. 12.13.

Assume that the value in R4 is not available when I2 and I3 are issued; the reason could be, for example, that one of the operands needed for I1 is itself not available. Thus we assume that I1 has not even started executing when I2 and I3 are issued.

When I2 and I3 are issued, they are parked in the reservation stations of the appropriate functional units. Since the required result value from I1 is not available, these reservation station entries of I2 and I3 get source tag corresponding to the output of I1—i.e. output of the functional unit which is performing the operation of I1.

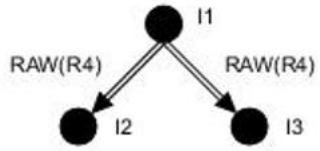

Fig. 12.13 Example of RAW dependences

When the result of I1 becomes available at its functional unit, it is sent over the common data bus along with the tag value of its source—i.e. output of functional unit.

At this point, programmable register R4 as well as the reservation stations assigned to I2 and I3 have the matching source tag—since they are waiting for the same result value, which is being computed by I1.

When the tag sent over the common data bus matches the tag in any destination, the data value on the bus is copied from the bus into the destination. The copy occurs at the same time into all the destinations which require that data value. Thus R4 as well as the two reservation stations holding I2 and I3 receive the required data value, which has been computed by I1, at the same time over the common data bus.

Thus, through the use of source tags and the common data bus, in one clock cycle, three destination registers receive the value produced by I1—programmable register R4, and the operand registers in the reservation stations assigned to I2 and I3

Let us assume that, at this point, the second operands of I2 and I3 are already available within their corresponding reservation stations. Then the operations corresponding to I2 and I3 can begin in parallel as soon as the result of I1 becomes available—since we have assumed here that I2 and I3 execute on two separate functional units.

It may be noted from Example 12.7 that, in effect, programmable registers become renamed to operand registers within reservation stations, which are program invisible. As we have seen in the previous section, such renaming also resolves anti-dependences and output dependences, since the target register of the dependent instruction is renamed in these cases to a different program invisible register.

Example 12.8 Combination of RAW and WAR dependence

Let us now consider a combination of RAW and WAR dependences.

Assume that instruction I1 is to write its result into R4, a subsequent instructions I2 is to read that result value, and a latter subsequent instruction I3 is then to write *its* result into R4. Thus instruction I2 is truly data dependent (RAW dependent) on instruction I1, but I3 is anti-dependent (WAR dependent) on I2. See Fig. 12.14.

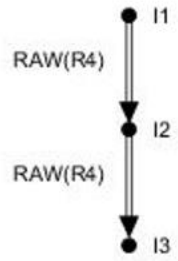

Fig. 12.14 Example of RAW & WAR dependences

As in the previous example, and keeping in mind similar possibilities, let us assume once again that the output of I1 is not available when I2 and I3 are issued; thus R4 has the source tag value corresponding to the output of I1.

When I2 is issued, it is parked in the reservation station of the appropriate functional unit. Since the required result value from I1 is not available, the reservation station entry of I2 also gets the source tag corresponding to the output of I1—i.e. the same source tag value which has been assigned to register R4, since they are both awaiting the same result.

The question now is: Can I3 be issued even before I1 completes and I2 starts execution?

The answer is that, with register renaming—carried out here using source tags—I3 can be issued even before I2 starts execution.

Recall that instruction I2 is RAW dependent on I1, and therefore it has the correct source tag for the output of I1. I2 will receive its required input operand as soon as that is available, when that value would also be copied into R4 over the common data bus. This is exactly what we observed in the previous example.

But suppose I3 is issued even before the output of I1 is available. Now R4 should receive the output of I3 rather than the output of I1. This is simply because, in register R4, the output of I1 is programmed to be overwritten by the output of I3.

Thus, when I3 is issued, R4 will receive the source tag value corresponding to the output of I3—i.e. the functional unit which performs the operation of I3. Its previous source tag value corresponding to the output of I1 will be overwritten.

When the output of I1 (finally) becomes available, it goes to the input of I2, but not to register R4, since this register's source tag now refers to I3. When the output of I3 becomes available, it goes correctly to R4 because of the matching source tag.

For simplicity of discussion, we have not tracked here the outputs of I2 and I3. But the student can verify easily that the two data transfers described above are consistent with the specified sequence of three instructions and the specified dependences.

Example 12.9 Scheduling across multiple iterations

Consider now the original iterative program loop discussed in Example 12.4.

Let us assume that, without any unrolling by the compiler, this loop executes on a processor which provides branch prediction and implements Tomasulo's algorithm. If instructions from successive loop iterations are available in the processor at one time—because of successful branch prediction(s)—and if floating point units are available, then instructions from successive iterations can execute at one time, in parallel.

But if instructions from multiple iterations are thus executing in parallel within the processor—at one time—then the net effect of these hardware techniques in the processor is the same as that of an unrolled loop. In other words, the processor hardware achieves *on the fly* what otherwise would require unrolling assistance from the compiler!

Even the dependence shown in Example 12.5 across successive loop iterations is handled in a natural way by branch prediction and Tomasulo's algorithm. Basically this dependence across loop iterations becomes RAW dependence between instructions, and is handled in a natural way by source tags and operand forwarding.

This example brings our clearly how a particular method of exploiting parallelism—*loop unrolling*, in this case—can be implemented either by the compiler or, equivalently, by clever hardware techniques employed within the processor.

Example 12.10 Calculation of processor clock cycles

Let us consider the number of clock cycles it takes to execute the following sequence of machine instructions. We shall count clock cycles starting from the last clock cycle of instruction 1, so that the answer is independent of the depth of instruction pipeline.

```
1    LOAD     mem-a, R4
2    FSUB     R7, R4, R4
3    STORE    mem-a, R4
4    FADD     R4, R3, R7
5    STORE    mem-b, R7
```

We shall assume that (a) one instruction is issued per clock cycle, (b) floating point operations take two clock cycles each to execute, and (c) memory operations take one clock cycle each when there is L1 cache hit.

If we add the number of clock cycles needed for each instruction, we get the total as 1+2+1+2+1 = 7. However, if no operand forwarding is provided, the RAW dependences on registers R4 and R7 will cost three additional clock cycles (recall Fig. 12.7), for a total of 10 clock cycles for the given sequence of instructions.

With operand forwarding—which is built into Tomasulo's algorithm—one clock cycle is saved on account of each RAW dependence—i.e. between (i) instructions 1 and 2, (ii) instructions 2 and 3, and (ii) instructions 4 and 5.

Thus the total number of clock cycles required, counting from the last clock cycle of instruction 1, is 7. With the assumptions as made here, there is no further scope to schedule these instructions in parallel.

In Tomasulo's algorithm, use of the common data bus and operand forwarding based on source tags results in *decentralized control* of the multiple instructions in execution. In the 1960s and 1970s, Control Data Corporation developed supercomputers CDC 6600 and CDC 7600 with a *centralized* technique to exploit instruction level parallelism.

In these supercomputers, the processor had a centralized *scoreboard* which maintained the status of functional units and executing instructions (see Chapter 6). Based on this status, processor control logic governed the issue and execution of instructions. One part of the scoreboard maintained the status of every instruction under execution, while another part maintained the status of every functional unit. The scoreboard itself was updated at every clock cycle of the processor, as execution progressed.

12.10 BRANCH PREDICTION

The importance of branch prediction for multiple issue processor performance has already been discussed in Section 12.3. About 15% to 20% of instructions in a typical program are branch and jump instructions, including procedure returns. Therefore—if hardware resources are to be fully utilized in a superscalar processor—the processor must start working on instructions beyond a branch, even before the branch instruction itself has completed. This is only possible through some form of branch prediction.

What can be the logical basis for branch prediction? To understand this, we consider first the reasoning which is involved if one wishes to predict the result of a tossed coin.

Note 12.3 Predicting the outcome of a tossed coin

Can one predict the result of a single coin toss?

If we have prior knowledge—gained somehow—that the coin is unbiased, then the answer is a clear NO, in the sense that both possible outcomes *head* and *tail* are equally probable. The only possible prediction one can make in this case is that the coin will come up either *head* or *tail*—i.e. a prediction which is of no practical value!

But how can we come to have prior knowledge that a coin is unbiased? Logically, the only knowledge we can have about a coin is obtained through observations of outcomes in successive tosses. Therefore, the more realistic situation we must address is that we have no prior knowledge about the coin being either unbiased or biased. Having received a coin, any inference we make about it—i.e. whether it is biased or not—can only be on the basis of observations of outcomes of successive tosses of the coin.

In such a situation of no prior knowledge, assume that a coin comes up *head* in its first two tosses. Then simple conditional probability theory predicts that the third toss of the coin has a higher probability of coming up *head* than of coming up *tail*.

This is a straightforward example of Bayesian reasoning using conditional probabilities, named after Rev. Thomas Bayes [1702–1761]. French Mathematician Laplace [1749–1827] later addressed this and related questions and derived a formula to calculate the respective conditional probabilities[7].

Like tossed coins, outcomes of conditional branches in computer programs also have *yes* and *no* answers— i.e. a branch is either taken or not taken. But outcomes of conditional branches are in fact biased—because there is strong correlation between (a) successive branches taken at the same conditional branch instruction in a program, and (b) branches taken at two different conditional branch instructions in the same program.

This is how programs behave, i.e. such correlation is an essential property of real-life programs. And such correlation provides the logical basis for branch prediction. The issue for processor designers is how to discover and utilize this correlation *on the fly*—without incurring prohibitive overhead in the process.

A basic branch prediction technique uses a so-called *two-bit predictor*. A two-bit counter is maintained for every conditional branch instruction in the program. The two-bit counter has four possible states; these four states and the possible transitions between these states are shown in Fig. 12.15.

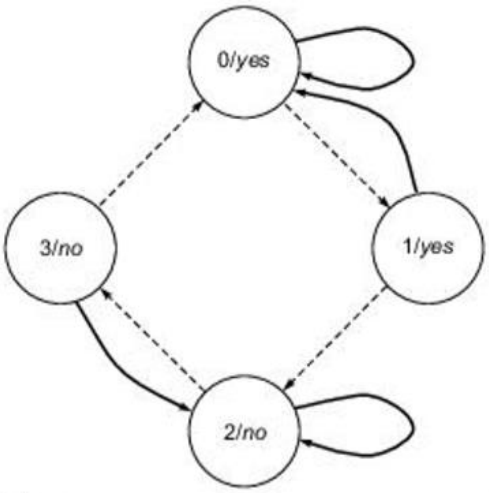

Fig. 12.15 State transition diagram of 2-bit branch predictor[8]

When the counter state is 0 or 1, the respective branch is predicted as *taken*; when the counter state is 2 or 3, the branch is predicted as *not taken*. When the conditional branch instruction is executed and the actual branch outcome is known, the state of the respective two-bit counter is changed as shown in the figure using solid and broken line arrows.

When two successive predictions come out wrong, the prediction is changed from *branch taken* to *branch not taken*, and *vice versa*. In Fig. 12.14, state transitions made on mis-predictions are shown using broken line arrows, while solid line arrows show state transitions made on predictions which come out right.

This scheme uses a two-bit counter for every conditional branch, and there are many conditional branches in the program. Overall, therefore, this branch prediction logic needs a few kilobytes or more of fast memory. One possible organization for this branch prediction memory is in the form of an array which is indexed by low order bits of the instruction address. If twelve low order bits are used to define the array index, for example, then the number of entries in the array is 4096[9].

In some programs, whether a conditional branch is taken or not taken correlates better with other conditional branches in the program—rather than with the earlier history of outcomes of the same conditional branch. Accordingly, *correlated predictors* can be designed, which generate a branch prediction based on whether other conditional branches in the program were taken or not taken.

Branch prediction based on the earlier history of the same branch is known as *local prediction*, while prediction based on the history of other branches in the program is known as *global prediction*. A *tournament predictor* uses (i) a global predictor, (ii) a local predictor, and (iii) a *selector* which selects one of the two predictors for prediction at a given branch instruction. The selector uses a two-bit counter per conditional branch—as in Fig. 12.14—to choose between the global and local predictors for the branch. Two successive mis-predictions cause a switch from the local predictor to the global predictor, and *vice versa*, the aim is to infer which predictor works better for the particular branch.

As we discussed in Section 12.3, under any branch prediction scheme, a mis-predicted branch means that subsequent instructions must be flushed from the pipeline. It should of course be noted here that the actual result of a conditional branch instruction—as against its predicted result—is only known when the instruction completes execution.

Speculative Execution Instructions executed on the basis of a predicted branch, before the actual branch result is known, are said to involve *speculative execution.*

If a branch prediction turns out to be correct, the corresponding speculatively executed instructions must be committed. If the prediction turns out to be wrong, the effects of corresponding speculative operations carried out within the processor must be cleaned up, and instructions from another branch of the program must instead be executed.

12.11 LIMITATIONS IN EXPLOITING INSTRUCTION LEVEL PARALLELISM

There is no such thing as free lunch!—an American proverb.

Technology is about trade-offs—and therefore it will come as no surprise to the student to learn that there are practical limits on the amount of instruction level parallelism which can be exploited in a single executing instruction stream. In this section we shall try to identify in broad terms some of the main limiting factors[10].

Consider as an example a multiple issue processor which targets four instruction issues per clock cycle, and has eight stages in the instruction pipeline. Clearly, in this processor at one time as many as thirty two instructions may be in different stages of fetch, decode, issue, execute, write result, commit, and so on—and each stage in the processor must handle four instructions in every clock cycle.

Assuming that 15% of the executing instructions are branches and jumps, the processor would handle at one time four to five such instructions—i.e. multiple predicted branches would be executing at one time.

Similarly, multiple loads and stores would be in progress at one time. Also, dynamic scheduling would require a fairly large instruction window, to maintain the issue rate at the targeted four instructions per clock cycle.

Consider the instruction window. Instructions in the window must be checked for dependences, to support out of order issue. This requires associative memory and its control logic, which means an overhead in chip area and power consumption; such overhead would increase with window size. Similarly, any form of checking amongst executing instructions—e.g. checking addresses of main memory references, for alias analysis—would involve overhead which increases with issue multiplicity k. In turn, such increased overhead in aggressive pursuit of instruction level parallelism would adversely impact processor clock speed which is achievable, for a given VLSI technology.

The increased overhead also necessitates a larger number of stages in the instruction pipeline, so as to limit the total delay per stage and thereby achieve faster clock cycle; but a longer pipeline results in higher cost of flushing the pipeline. Thus the aggregate performance impact of increased overhead finally places limits on what is achievable in practice with aggressively superscalar, VLIW and EPIC architecture.[11]

Basically, the increased overhead required within the processor implies that:

(i) To support higher multiplicity of instruction issue, the amount of control logic required in the processor increases disproportionately, and

(ii) For higher throughput, the processor must also operate at a high clock rate.

But these two design goals are often at odds, for technical reasons of circuit design, and also because there are practical limits on the amount of power the chip can dissipate.

Power consumption of a chip is roughly proportional to $N \times f$, where N is the number of devices on the chip, and f is the clock rate. The number of devices on the chip is largely determined by the fabrication technology being used, and power consumption must be held within the limits of the heat dissipation possible.

12.12 THREAD LEVEL PARALLELISM

We have already seen that dependences amongst machine instructions limit the amount of instruction level parallelism which is available to be exploited within the processor. The dependences may be true data dependences (RAW), control dependences introduced by conditional branch instructions, or resource dependences[13].

One way to reduce the burden of dependences is to combine—with hardware support within the processor—instructions from multiple independent threads of execution. Such hardware support for multi-threading would provide the processor with a pool of instructions, in various stages of execution, which have a relatively smaller number of dependences amongst them, since the threads are independent of one another.

Let us consider once again the processor with instruction pipeline of depth eight, and with targeted superscalar performance of four instructions completed in every clock cycle (see Section 12.11). Now suppose that these instructions come from four independent threads of execution. Then, on average, the number of instructions in the processor at any one time from one thread would be $4 \times 8/4 = 8$.

With the threads being independent of one another, there is a smaller total number of data dependences amongst the instructions in the processor. Further, with control dependences also being separated into four threads, less aggressive branch prediction is needed.

Another major benefit of such hardware-supported multi-threading is that pipeline stalls are very effectively utilized. If one thread runs into a pipeline stall—for access to main memory, say—then another thread makes use of the corresponding processor clock cycles, which would otherwise be wasted. Thus hardware support for multi-threading becomes an important latency hiding technique.

To provide support for multi-threading, the processor must be designed to switch between threads—either on the occurrence of a pipeline stall, or in a round robin manner. As in the case of the operating system switching between running processes, in this case the hardware context of a thread within the processor must be preserved.

But in this case what exactly is the meaning of the *context of a thread*?

Basically, thread context includes the full set of registers (programmable registers and those used in register renaming), PC, stack pointer, relevant memory map information, protection bits, interrupt control bits, etc. For N-way multi-threading support, the processor must store at one time the thread contexts of N executing threads. When the processor switches, say, from thread A to thread B, control logic ensures that execution of subsequent instruction(s) occurs with reference to the context of thread B.

Note that thread contexts need not be saved and later restored. As long as the processor preserves within itself multiple thread contexts, all that is required is that the processor be able to switch between thread contexts from one clock cycle to the next.

As we saw in the previous section, there are limits on the amount of instruction level parallelism which can be extracted from a single stream of executing instructions—i.e. a single thread. But, with steady advances in VLSI technology, the aggregate amount of functionality that can be built into a single chip has been growing steadily.

Therefore hardware support for multi-threading—as well as the provision of multiple processor cores on a single chip—can both be seen as natural consequences of the steady advances in VLSI technology. Both these developments address the needs of important segments of modern computer applications and workloads.

Depending on the specific strategy adopted for switching between threads, hardware support for multi-threading may be classified as one of the following:

(i) *Coarse-grain multi-threading* refers to switching between threads only on the occurrence of a major pipeline stall—which may be caused by, say, access to main memory, with latencies of the order of a hundred processor clock cycles.

(ii) *Fine-grain multi-threading* refers to switching between threads on the occurrence of any pipeline stall, which may be caused by, say, L1 cache miss. But this term would also apply to designs in which processor clock cycles are regularly being shared amongst executing threads, even in the absence of a pipeline stall.

(iii) *Simultaneous multi-threading* refers to machine instructions from two (or more) threads being issued in parallel in each processor clock cycle. This would correspond to a multiple-issue processor where the multiple instructions issued in a clock cycle come from an equal number of independent execution threads.

With increasing power of VLSI technology, the development of multi-core *systems-on-a-chip* (SoCs) was also inevitable, since there are practical limits to the number of threads a single processor core can support. Each core on the Sun UltraSparc T2, for example, supports eight-way fine-grain multi-threading, and the chip has eight such cores. Multi-core chips promise higher net processing performance per watt of power consumption.

Systems-on-a-chip are examples of fascinating design trade-offs and the technical issues which have been discussed in this chapter. Of course, we have discussed here only the basic design issues and techniques. For any actual task of processor design, it is necessary to make many design choices and trade-offs, validate the design using simulations, and then finally complete the design in detail to the level of logic circuits.

Over the last couple of decades, enormous advances have taken place in various areas of computer technology; these advances have had a major impact on processor and system design. In the next chapter, we shall discuss in some detail these advances and their impact on processor and system design. We shall also study in brief several commercial products, as case studies in how actual processors and systems are designed.