Techniques to Improve Performance Beyond Pipelining: Superpipelining, Superscalar, and VLIW

Jean-Luc Gaudiot, Jung-Yup Kang, Won Woo Ro
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>1</td>
</tr>
<tr>
<td>0.1 Introduction - Overview</td>
<td>1</td>
</tr>
<tr>
<td>0.2 Superpipelining</td>
<td>3</td>
</tr>
<tr>
<td>0.2.1 Concepts and Benefits</td>
<td>3</td>
</tr>
<tr>
<td>0.2.2 Design Challenges</td>
<td>4</td>
</tr>
<tr>
<td>0.2.3 Example Architectures</td>
<td>6</td>
</tr>
<tr>
<td>0.3 Superscalar Architectures</td>
<td>10</td>
</tr>
<tr>
<td>0.3.1 Concept and Benefits</td>
<td>11</td>
</tr>
<tr>
<td>0.3.2 Design Challenges</td>
<td>17</td>
</tr>
<tr>
<td>0.3.3 Example Architectures</td>
<td>18</td>
</tr>
<tr>
<td>0.4 VLIW (Very Long Instruction Word)</td>
<td>22</td>
</tr>
<tr>
<td>0.4.1 Concept and Benefits</td>
<td>22</td>
</tr>
<tr>
<td>0.4.2 Design Challenges</td>
<td>29</td>
</tr>
<tr>
<td>0.4.3 Example Architectures</td>
<td>30</td>
</tr>
<tr>
<td>0.5 Conclusion- Impact on Modern Microprocessors</td>
<td>33</td>
</tr>
</tbody>
</table>
Superpipelining, Superscalar and VLIW are techniques developed to improve the performance beyond what mere pipelining can offer.

Superpipelining improves the performance by decomposing the long latency stages (such as memory access stages) of a pipeline into several shorter stages, thereby possibly increasing the number of instructions running in parallel at each cycle.

On the other hand, Superscalar and VLIW approaches improve the performance by issuing multiple instructions per cycle. Superscalar dynamically issues multiple instructions and VLIW statically issues multiple instructions at each cycle. These techniques are pioneers (and now the basis) of modern computer architecture designs.

In this chapter, we describe and investigate examples of these techniques and examine how they have affected the designs of modern microprocessors.

0.1 Introduction - Overview

In general, the performance of a microprocessor is expressed by equation (1):

\[ \text{Execution Time} = IC \times CPI \times \text{clock_cycle_time} \] (1)

In this equation, \( IC \) corresponds to the number of instructions in the program at hand, while \( CPI \) represents the average number of clock cycles per instruction. \( \text{clock_cycle_time} \) represents the duration of a clock cycle.

In a basic pipeline architecture, only one instruction can be issued at each cycle (detailed information about pipelined architecture is given in \([24, 25, 26]\)). Overcoming this limitation has been the subject of much work in the past: more specifically, it has been the goal of computer architects to reduce each variable of equation (1). For one thing, \( CPI \) can be improved by designing more advanced or-
ganizations and architectures and also by modifying the instruction set architecture (such as providing instructions which may require fewer cycles). *clock_cycle_time* can be improved by better process technology, advanced hardware implementation techniques, and also by more sophisticated architectural organizations. Finally, *IC* can be improved by implementing advanced compiler and instruction set architectures.

The three main techniques mentioned earlier (superpipelining, superscalar, and VLIW) are such improvements to the architecture. Indeed, superpipelining improves the performance by dividing the long latency stages (such as the memory access stages) of a pipeline into several shorter stages, thereby reducing the clock rate of the pipeline. (in other words, superpipelining has the effect of reducing *clock_cycle_time* in equation (1)). Superscalar, on the other hand, improves the performance by dynamically issuing multiple instructions per cycle (this means that superscalar reduces *CPI*). VLIW (Very Long Instruction Word) improves the performance by utilizing the compiler to combine several instructions into one very long instruction and executing it. (VLIW reduces *IC*). Table 1 highlights the differences among the three techniques.

For the last two decades, each of these techniques and a variety of combinations have had a significant effect on the performance of modern processors. Therefore, in this chapter, we first introduce the concepts and benefits of each technique and

<table>
<thead>
<tr>
<th>APPROACH &amp; Effects</th>
<th>Superpipelining</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effects on (clock cycle time) of equation (1)</td>
<td>Dividing the long latency stages of a pipeline into several shorter stages</td>
<td>Dynamically issuing multiple instructions per cycle</td>
<td>Utilizing the compiler to combine several instructions into one very long instruction and executing it</td>
</tr>
<tr>
<td>Instruction issue rate</td>
<td>1</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>Instruction issue style</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>Difficulty of Design</td>
<td>Relatively easy compared to the other two</td>
<td>Complex design issues Run-time tracking</td>
<td>Complex design issues Compiler support</td>
</tr>
<tr>
<td>Keywords</td>
<td>Higher MHz, latch delay, clock skew</td>
<td>Dynamic scheduling, Out-of-Order execution, Register renaming, Scoreboarding</td>
<td>Compiler support, static scheduling, loop unrolling, trace scheduling</td>
</tr>
</tbody>
</table>
then examine examples of each kind.

0.2 Superpipelining

Superpipelining simply allows a processor to improve its performance by running the pipeline at a higher clock rate. The higher clock rate is achieved by identifying the pipeline stages that determine the clock cycle (in other words, the most time consuming stages) and decomposing them into smaller stages.

Superpipelining is the simplest technique of the three. It does not require additional hardware (such as functional units and fetch units) that a superscalar architecture does. It also does not require the complex controls (no multiple issue control or need to keep track of the instructions issued). Finally, it should be noted that superpipelining does not need the advanced compiler technologies required by the VLIW model.

0.2.1 Concepts and Benefits

The concept of superpipelining [5, 15, 16] is best explained by an illustration. Imagine the pipeline shown in Figure 1 (the simple pipeline of the DLX architecture [25, 24]). In this figure, each stage is drawn with a white box (with the corresponding name inside) while the gray skinny sticks represent the latches between the stages. The length of the double arrows represent the length of time required by each stage for completion of its task. From the Figure 1, either the IF (Instruction Fetch) stage or the MEM (Data memory access) stage determine the clock rate because they are the longest time consuming stages of the pipeline (assume for the moment that the IF and MEM stages are nearly twice as long as the other stages).

We can increase the performance of the pipeline in Figure 1 by dividing the IF and the MEM (Data memory access) stages into two halves (and placing latches in
between the new stages as in Figure 2. This allows the \textit{clock_cycle_time} of equation (1) to be reduced (by half). This is the key point about superpipelining. (Organizing a pipeline with stages that have approximately the equal amount of execution time.) This reduced clock rate improves the performance. This is because the clock cycle for the pipeline is determined by the longest stage and as in Figure 3, if the pipeline has few long stages and the rest are nominal compared to the longer ones, there are wasted time when executing the shorter stages. Thus, if there are hundreds and thousands of instructions in a program, the aggregated wasted time will be significant (this wasted time is denoted with dotted lines in Figure 3-(a)). On the other hand, if the pipeline was superpipelined with discrete design principles as it was in Figure 2, most of the stages will be balanced and the wasted time will be minimal as in Figure 3(b).

0.2.2 Design Challenges

At least superficially, superpipelining could appear easy to attain: simply divide the most time consuming stages into small units (and continue this process until all the stages are infinitely small). However, as we know, there is a limit to the efficiency (and potential of such a subdivision). Indeed, it can be observed from Figure 4 that a pipeline stage can be meaningfully pipelined only if the stage between two latches executes in more time ($A$ in Figure 4) than one latch ($B$ in Figure 4). Hwang reports that in general, the time delay of a latch is about one or two order of magnitude shorter compared to the delay of the longest pipeline stage. Otherwise, we just spend more time on additional latches for longer pipelines. Second, after some point, it may simply be infeasible to continue dividing a specific pipeline stage. Third, as there are more pipeline stages and thus more latches to control, the clock skew problem becomes significant which would in turn directly affect the time allotted to each stage to perform its execution.

Kunkel shows the analysis of different latch designs and the effects of latch overhead in pipeline as well as data dependencies and clock skew overhead.
He reports that if pipeline stages get extremely short, it becomes necessary to pad some delay to obtain performance improvement. He also says that eight to ten gate levels per pipeline stage lead to optimal overall performance.

Superpipelining also brings a number of major side effects. As the number of stages is increased, so is the number of forwarding paths (stages) and the delay (stall cycles) caused by branches and loads. The combination of the above two factors has strong adverse affects on the performance of the processor since it increases the CPI (therefore, techniques to improve CPI, such as hardware-based pipeline scheduling, branch prediction, multiple issue, and compiler-assisted parallelism exploiting techniques have been developed). These topics will be discussed in the context of superscalar and VLIW. However, it should be noted that when the above techniques are considered, superpipelining can improve CPI as well.)
0.2.3 Example Architectures

Three processor architectures can show possible implementations of superpipelining. In those examples, we focus on how superpipelining has helped improve the performance of each architecture and show the design techniques to exploit the benefits of using superpipelining.

We first show an early version of a superpipelined architecture (MIPS R4000) and then move on to more modern and advanced models. The modern processors are not only superpipelined but also based on a combination of superpipelining and superscalar architecture.

MIPS R4000

The MIPS R4000 \cite{4,10} is a superpipelined version of the MIPS R3000 which implemented only a 5-stage pipeline whereas the MIPS R4000 had 8 stages. The MIPS R4000 operates at double the speed (100 MHz) of the MIPS R3000 \cite{6}. This was achieved by advances in the process technology and improved circuit design techniques \cite{4}.

One of the main goals of the architects of the MIPS R4000 was to improve the performance over that of MIPS R3000. The choice was between superpipelining and superscalar (and actually VLIW as well) \cite{4}. Superpipelining was ultimately selected because: 1) less logic is required compared to superscalar, 2) the control logic is much simpler (no multiple issues and no need to keep track of the instructions issued and of the flow), 3) 1) and 2) result in a faster cycle time, short design and test time, and 4) no need for new compiler development.

The MIPS R4000 has a branch delay of three cycles and a load delay of two cycles. The MIPS R3000 had IF (Instruction Fetch), RF (Register Read), ALU (Add, logical, shift computation), DC (Data cache Access and tag check), and WB
0.2. **SUPERPIPELINING**

(Register File write) stages. The eight stages of R4000 are listed below:

- IF - Instruction Fetch, First Half
- IS - Instruction Fetch, Second Half
- RF - Register Fetch
- EX - Execution Instruction Fetch
- DF - Data Fetch, First Half
- DS - Data Fetch, Second Half
- TC - Tag Check
- WB - Write Back

**The ARM11 Processor**

ARM cores are famous for their simple and cost-effective design. However, ARM cores have also evolved and show superpipelining characteristics in their architectures and have architectural features to hide the possible long pipeline stalls. Amongst of the ARM cores, the ARM11 processor is based on the ARM architecture v6. The ARM11 (specifically, the ARM1136JF-S processor [3]) is a high-performance and low-power processor which is equipped with eight stage pipelining. The core consists of two fetch stages, one decode stage, one issue stage, and four stages for the integer pipeline. The eight stages of the ARM11 core are depicted in Figure[5].

Below is described the function of each of the eight stages:

- Fe1 - Instruction Fetch, First Half, Branch Prediction
- Fe2 - Instruction Fetch, Second Half, Branch Prediction
- De - Instruction Decode
- Iss - Register read and instruction issue
- Sh - Shifter stage
- ALU - Main integer operation calculation
- Sat - Pipeline stage to enable saturation of integer results
Figure 5: Eight Pipe Stages of ARM11 Core

- WBex - Write Back of data from the multiply or main execution pipelines
- MAC1 - First stage of the multiply-accumulate pipeline
- MAC2 - Second stage of the multiply-accumulate pipeline
- MAC3 - Third stage of the multiply-accumulate pipeline
- ADD - Address generation stage
- DC1 - First stage of Data Cache access
- DC2 - Second stage of Data Cache access
- WBIs - Write back of data from the Load Store Unit

The ARM11 core uses both dynamic (using branch target address cache and branch history) and static (using the direction of the branches) predictions to hide the deep penalty caused by having increased the length of the pipeline. The instruction prefetch unit of the ARM11 core buffers up to three instructions in its FIFO in order to [3]: 1) detect branch instructions before they enter the integer units, 2) dynamically predict those branches (to be taken), and 3) provide branch folding of the predicted branches if possible. By using this prefetch unit, the ARM11 core reduces the cycle time of the branch instructions and improves the performance. Previous ARM processors without the prefetch unit suffered from a one cycle delay when a branch was not taken and three or more cycles for the taken branches (since the target address was known only at the end of the execute stage).
0.2. SUPERPIPELINING

The Intel NetBurst Microarchitecture

In this section, we discuss the Intel NetBurst microarchitecture (again, we focus on the superpipelining feature of the design). It is a deeply pipelined design (called hyper-pipelined technology) and it can, not only run at high clock rates (up to 10 GHz [14]), but it also allows different part of the processor run at different clock rates. For instance, the most frequently-executed instruction in common cases (such as cache hit) are decoded efficiently and executed with short latencies [14] therefore, improving the overall performance. The pipeline is also equipped with branch penalty hiding techniques such as speculation, buffering, superscalar, and out-of-order execution. The Pentium 4 processor and the Xeon processor are based on the Intel NetBurst microarchitecture.

However, in some modern complex processor designs (such as the NetBurst architecture), the number of cycles through which an instruction must go cannot be specifically determined because of the concept of “out-of-order execution.” However, we can categorize the different parts of the pipeline through which an instruction must go. Thus, let us examine how these different parts of the NetBurst pipeline are correlated.

The NetBurst pipeline consists of three main parts and the pipeline architecture of Netburst is depicted in Figure 6. Those three main parts are:

- the in-order issue front end
- the out-of-order superscalar execution core
- the in-order retirement unit

In the NetBurst pipeline, the in-order issue front end feeds instructions to the out-or-order execution core. It fetches and decodes instructions. Instructions are decoded and translated into micro-ops (\(\mu\)ops). The main responsibility of the front-end is to feed these micro-ops to the execution core in program order. It also utilizes the trace cache to store the program execution flow and it possesses a branch prediction unit. The execution core executes the micro-ops in an out-of-order fashion. The retirement unit retires the micro-ops in program order. In the next section, superscalar and out-of-order execution, which are another important architecture design feature of modern microprocessors (such as Netburst architecture) will be discussed in detail.

The comparison of the features of the pipeline of the example superpipelining architectures are shown in Table 2.
0.3 Superscalar Architectures

One of the main design issues for today’s high-performance microprocessors is exploiting a high level of Instruction Level Parallelism (ILP). The ILP is a primary way to improve the parallelism of a sequential code. The basic two strategies for exploiting Instruction Level Parallelism are *multiple instruction issue* and *out-of-order execution*. Those two techniques were invented to execute as many instructions as possible in a given clock cycle. In addition, sophisticated branch prediction schemes and speculative execution techniques were proposed to overcome the limited amount of parallelism available within a basic block. Both techniques have successfully provided many opportunities to discover more parallelism across the control flow limit [29].

Traditionally, superscalar and Very Long Instruction Word (VLIW) architectures were developed as the main microprocessor models for exploiting ILP. Although both architectures target multiple instruction issue and out-of-order execution, they...
have adopted fundamentally different approaches. The significant differences lie in
the way to schedule the instructions. Superscalar processors are mainly character-
ized with dynamic scheduling which uses hardware to schedule the instructions at
run-time. On the contrary, VLIW architectures depend on static scheduling which
schedules the instructions at compile time in a static way [11]. We will discuss the
design features and some example architectures of superscalar and VLIW in this
and the following section. First, the basic concept and design issues of superscalar
architectures will be presented in detail.

### 0.3.1 Concept and Benefits

Superscalar architectures are strongly based on previous pipelined RISC processors.
The most significant advantage of RISC processors is the inherent simplicity of the
instruction set architecture. The early pipelined RISC processors were designed to
execute one instruction for one pipeline stage. The major innovation of the super-
scalar architecture over the traditional pipelined processor is the multiple instruction
issue advantage through dynamic instruction scheduling. In those approaches, each
pipeline stage can handle multiple instructions simultaneously. Although the early
superscalar architectures were designed to issue multiple instructions in an in-order
fashion, this would consequently require out-of-order issue to maximize the effect of
the multiple instruction issue feature. Together with the dynamic scheduling and
enough backward compatibility, the multiple issue characteristic becomes the main
advantage of the current superscalar processors.

The basic working mechanism of dynamic scheduling is quite simple. The in-
structions are simply fetched in the instruction sequence. They are then stored in

---

**Table 2: Comparison of MIPS R4000, ARM11 Core, and Intel NetBurst**

<table>
<thead>
<tr>
<th>Number of Superpipeline stages</th>
<th>MIPS R4000</th>
<th>ARM11 Core</th>
<th>Intel NetBurst</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>3 distinguished groups of pipe stages (Front end, Execution unit, and Retirement unit. Each with different number of pipestages)</td>
</tr>
<tr>
<td>Pipeline Special Features</td>
<td>Advanced process technology and improved circuit design techniques (compared to 5-stage MIPS R3000)</td>
<td>Advanced branch predictions, prefetching, and buffering</td>
<td>Hyper-pipelined technology (Pipe stages have different clock frequencies), speculation, buffering, superscalar, and out-of-order execution</td>
</tr>
</tbody>
</table>
a pool of instructions (also referred to as the *scheduling window*, *instruction window* or *instruction buffer*). Just as in the basic RISC architecture, the instructions are fetched sequentially following the program counter (PC). However, once the instructions are decoded, the dependencies among the instructions are analyzed. Then, the processor can deduce which instructions inside the pool can be executed without causing data hazards. Concurrent execution or out-of-order execution is made possible by hardware-based run-time dependency checking.

**A Basic Model of Superscalar Pipeline**

Figure 7 shows a typical model of a basic five-stage pipelined processor. In the single instruction issue RISC processor model, each stage handles only one instruction per cycle. In addition, the execution of the instructions should be made in-order, following the fetching sequence of the original binary code. On the contrary, each of the processing stage is able to handle a number of instructions in the superscalar pipeline model. For example, the IF (Instruction Fetch) stage can fetch multiple instructions from the instruction cache and the EX (Execution) stage can execute multiple instructions simultaneously. This multiple instruction issue characteristic inherently means out-of-order execution of the instruction sequence. It has been developed to reduce the execution time of a fixed size of code by exploiting fine-grain parallelism as low as at the instruction level.

```plaintext
IF  ID  EX  WB  CT
Fetch  Decode  Execute  Write-Back  Commit
```

Figure 7: Five-stage pipelined processor

However, executing multiple instructions simultaneously means the sequential characteristic of the original code cannot be guaranteed. This brings out highly critical problems since the current programming languages are strongly based on a sequential programming model, and the binary code is correctly executed only if its sequential characteristic is respected. An inappropriate execution order between
instructions could cause data hazards due to the register naming dependencies. Figure 8 depicts possible data hazards due to an out-of-order execution.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Data dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2, 0(r7) -- (a)</td>
<td>RAW: Read After Write</td>
</tr>
<tr>
<td>add r4, r2, r3 -- (b)</td>
<td>WAR: Write After Read</td>
</tr>
<tr>
<td>add r3, r4, r5 -- (c)</td>
<td>WAW: Write After Write</td>
</tr>
<tr>
<td>sub r9, r3, r4 -- (d)</td>
<td></td>
</tr>
<tr>
<td>sub r9, r2, r7 -- (e)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Data dependencies in a sequential code

The instruction stream on the left side of Figure 8 shows an example of a sequential instruction stream. In the previous pipelined RISC processor model, those instructions are fetched, executed, and committed in program order. This means that none of the instructions can be executed before any instruction ahead of it is committed. However, in the superscalar model, the processor aims at executing in a cycle as many instructions as possible through multiple instruction issue and out-of-order execution (within the allowed bandwidth).

Now, can the five instructions in Figure 8 be executed simultaneously? Obviously, executing those five instructions in parallel without considering the program order does not produce correct results. Indeed, there are some data dependencies.

For example, instructions (b) and (e) cannot be executed until instruction (a) completes, since both (b) and (e) need to read r2; there exists a RAW (Read After Write) data dependency. On the other hand, a WAR (Write After Read) dependency exists between the instruction (b) and (c) due to r3. Also, a WAW (Write After Write) dependency can be found between instructions (d) and (e). In the superscalar architecture model, the data dependencies which can cause data hazards are detected during the instruction decode stage. Based on those data dependencies, the instruction execution order is decided dynamically. It will be explained shortly in the next part. The dynamic scheduling of the Superscalar model is strongly based on the early Tomasulo [31] or Scoreboard algorithms [30].
Dynamic Scheduling of Superscalar

Superscalar architectures achieve dynamic instruction scheduling by using a scheduling window. In dynamic scheduling, the instructions are still fetched sequentially in program order. However, those instructions are decoded and stored in a scheduling window of a processor execution core. After decoding the instructions, the processor core obtains the dependency information between the instructions and can also identify the instructions which are ready for execution (ready instructions mean the instructions of which the source registers have been calculated). Indeed, there must be no data hazard in issuing the ready instructions.

In essence, the execution of an instruction $i$ is decided by the previous instruction $k$ which produces the value for the source registers of the instruction $i$. This means that the superscalar processors can execute the instructions as soon as the input values are available (also, the corresponding functional units should be available). The processor dynamically achieves parallel execution of the instructions by performing a run-time dependency check and waking up the following instructions.

However, at the same time, it also implies that every instruction should broadcast the register name and value across the issue window to wake up and select any ready instruction. This means that a superscalar architecture needs a complex wake-up and selection logic inside of the scheduling window. It should be also noted that a superscalar architecture is expected to deliver high degrees of Instruction-Level Parallelism, together with a large issue width which consequently requires more complex hardwired instruction scheduling logic. This will cause a significant overhead for future superscalar processors.

Although superscalar architectures require complex issue logic and scheduling mechanisms, they have been the standard general-purpose processor during the last decade. This is mostly due to the fact that this model possesses an inherent backward compatibility. This adaptability to legacy code and the ability to continue using conventional compilers make superscalar architectures quite attractive. That is the main reason why all major processor vendors have focused on developing superscalar-styled ILP processors since the 1990s.

Register Renaming

Register renaming is a technique which increases the chance to uncover independent instructions by removing the false dependencies. Those false dependencies are caused by using the same register name. More specifically, the register renaming can eliminate the WAW (write after write) and WAR (write after read) dependencies among the instructions. Figure 9 shows one such example. The four instructions
on the right side of Figure 9 show the original instruction stream produced by the compiler. Through register r2, there exists a WAW dependency between instructions (a) and (c) and a WAR dependency between instructions (b) and (c). Those dependencies limit the parallel execution (or out-of-order execution) of the instructions. Clearly, the instruction (c) cannot be executed before instruction (b) reads r2. However, this data dependency can be removed when the register r2 in instruction (c) and (d) is given another register name. The four instructions on the right side show the instruction stream with register renaming.

\[
\begin{array}{l}
\text{add } r2, r3, r4 \quad -- \ (a) \\
\text{add } r4, r2, r3 \quad -- \ (b) \\
\text{sub } r2, r6, r7 \quad -- \ (c) \\
\text{mul } r9, r2, r8 \quad -- \ (d)
\end{array}
\quad < \text{after renaming} >
\begin{array}{l}
\text{add } r2, r3, r4 \quad -- \ (a) \\
\text{add } r4, r2, r3 \quad -- \ (b) \\
\text{sub } r2', r6, r7 \quad -- \ (c) \\
\text{mul } r9, r2', r8 \quad -- \ (d)
\end{array}
\]

Figure 9: Register renaming example

Now, instruction (c) can be issued and executed, regardless of the execution of instruction (a), since it writes into register r2'. The register renaming scheme was originally developed as an compiling technique for an ILP processor such as VLIW. However, the superscalar architecture also uses a register renaming scheme for the same purpose. Its implementation is a purely hardware-based dynamic method. There are two ways we can implement the register renaming scheme in the superscalar model. The first method explicitly provides a larger number of physical registers than the logical registers. Then, at run-time, the processor can map any available physical register to the logical register. This technique requires a register mapping table. The second method uses a reorder buffer (ROB). In this approach, each entry of reorder buffer works as a physical register.
Reorder Buffer

A Reorder buffer (ROB) provides two main functions in modern superscalar processors. The first task is to guarantee the in-order completion of a sequential instruction stream. Especially, the in-order completion is crucial for correct speculative execution and the implementation of precise interrupts [34]. As seen in Figure 7, a basic superscalar processor has two stages for instruction completion: the Write Back (WB) stage and the Commit (CT) stage. The WB stage allows the later instructions to use the result of the instruction. However, it does not allow the instruction to write the value into the register file or memory. The instruction still has to wait inside the ROB until it retires from the CT stage. The retire process in the CT stage strictly follows the program order (in-order completion). This in-order completion guarantees that the program can return to the precise state in any point of the program execution. The precise state means the sequentiality of the program execution is guaranteed. This is extremely important for interrupt or miss-speculation cases.

The ROB is implemented as a circular FIFO and retires the instructions following the program order; each entry of the ROB is assigned at the decoding stage and must hold the original program order. During the commit stage, the instructions at the head of the ROB are scanned and an instruction is committed if all the previous instructions are committed (multiple instructions within the commit-width can also be committed during the same cycle).

The second function of the ROB is to achieve register renaming through the slots of the ROB. When the processor uses an ROB, an available entry in the ROB is assigned to an instruction at the decoding stage. Each entry of the ROB contains three fields which indicate the instruction type, the destination register, and the result. The destination register field indicate the logical name of the register and the result field holds the value for the register. It also has a single bit ready indicator. When the ready bit indicates the value is ready, the later instructions which need the values can simply read the value; otherwise the later instruction should be left to wait inside a scheduling window until the value is ready.

Branch Prediction

A superscalar architecture is capable of delivering instructions beyond the current basic block by predicting the future control path. Therefore, excellent branch prediction is essential if we are to have a large pool of instructions to choose from. More particularly, the current trend in superscalar design is to achieve large issue-width with a large scheduling window. To satisfy those demands, the number of in-flight instructions (in other words, the instruction inside the processor in a clock cycle) should be large enough to find the ready instructions to issue or execute. By using
efficient branch prediction and speculation techniques, fetching many instructions from the correct control path is possible.

The simplest way to predict branch is static branch prediction; it means the branch instructions are either always taken or always not-taken. However, this static method does not accurately predict the branch outcome. Therefore, current superscalar processors mainly depend on dynamic branch prediction which utilizes the previous history to decide the behavior of future branches. In current superscalar designs, many dynamic branch prediction techniques have been proposed and developed.

A basic form of branch prediction consists in using the local history of a branch instruction. The Branch History Table (BHT) is accessed by indexing a certain number of lower bits of the PC of the branch instruction, where the table provides the history of the branch behavior. In addition, other prediction schemes also consider the global branch history together with the local history. Those combined schemes significantly enhance the branch prediction success rate. Such schemes include two-level adaptive branch predictions \[36\] and branch correlation \[23\]. The interested reader is referred to these publications for details on the working mechanism and design features of those branch schemes.

Besides predicting the branch direction (either taken or not-taken), there had been one more prediction on the target address. It is possibly achieved by using a Branch Target Buffer to provide the target address prediction. This operation can determine the expected target address at the instruction fetch stage; therefore, the BTB is accessed during the IF stage and the next PC can be obtained for the very next cycle \[34\].

### 0.3.2 Design Challenges

In a conventional superscalar design, wake up and select operation cannot be pipelined nor be implemented as a one-cycle operation \[22\]. Therefore, as far as the size of an issue window is concerned, more instructions within an issue window mean more communication overhead (wire delay) for the wake up and select operations in a clock cycle time. In fact, wires tend to scale poorly compared to semiconductor devices, and the amount of state that can be reached in a single clock period eventually ceases to grow \[1\]. Consequently, there is a scaling problem with regard to the issue window size of the superscalar design. The goals of a larger instruction window and of a faster clock become directly antagonistic in the issue window design of future superscalar processors \[1\]. This phenomenon will be even more noticeable as higher degrees of parallelism are needed. It is well recognized that the centralized scheduling logic would be the most critical part of a processor and would limit the
Another important issue in processor design is how to solve the problem caused by the dramatically growing speed-gap between the processor and main memory. An obvious solution would be the use of a large scheduling window: searching across a wider range of instructions can offer more opportunities to uncover independent instructions which can hide long access latencies. However, since a large instruction window will cause a lower clock rate, other approaches for processor design must be investigated. As an example, the partitioning of a single scheduling window can reduce the complexity of the centralized design as well as the size of each component \[7\] \[18\]. Indeed, a higher clock rate can be achieved since the decentralized model reduces the length of the wire for communication. This microarchitecture technique is called clustering.

The last design challenge of the dynamic scheduling is the considerable amount of power consumption. According to Gowan et al. \[9\], 18% of total power consumption of the Alpha 21264 processor is used by the instruction issue unit. A large amount of power is dissipated by the wake up and the selection logic. This is mainly due to the global communication of the centralized scheduling unit. As an alternative design, the clustered architectures can reduce the power consumption of the centralized instruction issue unit. Separating a single scheduling window into multiple structures can reduce the power consumption of the aggressive scheduling unit of a superscalar architecture.

### 0.3.3 Example Architectures

We consider two microprocessor models as being good representatives of superscalar architectures: the Alpha 21264 and the MIPS R10000.

#### Alpha 21264

The Alpha 21264 (EV6) processor (Figure 10) was introduced in 1998 with a 0.35\(\mu\)m CMOS process technology operating at 600MHz frequency. It is the first Alpha processor designed with an out-of-order issue strategy.

The Alpha 21264 has a 4 way issue bandwidth. However, the execution stage supports 6-way issue in a cycle with four integer functional units and two floating-point functional units. It is also designed to support 80 in-flight instructions in a processor. Alpha 21264 has only two floating-point units and a single physical register file. Both floating-point units are fully pipelined.

The Alpha 21264 processor is implemented with a 64-KB, two-way set-associative
L1 data cache. The earlier model of the processor (Alpha 21164) had only a 8-KB data cache which resulted in many more cache misses than the more modern 21264 processor. It now has a huge L1 data cache. As a result, the access latency to the data cache requires two clock cycles. However, two concurrent accesses to the level 1 data cache are supported.

Figure 11 shows a detailed description of the Alpha 21264 pipeline. In its basic inception, an instructions is required to go through all seven stages until it can write back the results. Since the 64KB data cache needs two clock cycles to deliver the data, load/store pipeline operations require two more clock cycles than normal operations.

One distinct feature of the Alpha 21264 is in that the fetch stage uses line and way prediction for the next instruction to fetch. It is possibly achieved by using the next line predictor and set predictor. The purpose of those predictors are similar to that of the Branch Target Buffer: it provides the address for the next instruction to fetch. As for the control flow prediction, the processor still has a
branch prediction mechanism: a hybrid branch predictor is implemented. One more interesting feature of the processor is the load hit/miss prediction which helps to schedule load instructions.

The EV7 architecture (Alpha 21364), the fourth generation of the Alpha processor, inherited many characteristics from the Alpha 21264 processor. Indeed, the Alpha 21364 is designed to be the System-On-Chip version of the Alpha 21264. It uses the EV6 as the core processing element and adds peripheral features such as integrated second level cache, memory controller, and network interface.

**MIPS R10000**

The MIPS R10000 is based on the previous MIPS processors. The previous MIPS processors were mainly single issue pipelined RISC processors. The MIPS R10000 is the first out-of-order issue superscalar processor which is implemented for the MIPS IV ISA. The MIPS R1000 processor fetches four instructions from
the instruction cache in every cycle and decodes those instructions. Any branch is immediately predicted upon detection.

The first implementation of the MIPS IV ISA architecture is the 4-issue superscalar processor (R8000). The R8000 is able to execute two ALU and two memory operations per cycle. However, the R8000 processor does not have a dynamic scheduling capability. The most distinguishable advantage of the R10000 (Figure 12) is the out-of-order execution feature which is enabled by the dynamic scheduling.

![MIPS R10000 Block Diagram](image)

Four instructions can be fetched and decoded at each clock cycle. Those instructions are stored in one of the three instruction queues after the decoding stage. Each queue has 16 entries for instructions. Those issue queues can also read operands from the register file. Once the register dependencies are resolved and functional units are available, the instructions can be executed in one of the five functional units. Normal integer operations take one clock cycle while load store instructions require two clock cycles in the execution stage. The floating-point operations need three clock cycles for their execution.
Both level 1 caches are 32KB in size and two-way interleaved. The core processor chip also controls a large external level 2 cache. The R10000 is implemented with a 64-bit split-transaction system bus used to communicate with the outside of the chip. Up to four R10000 chips can be integrated using this bus architecture.

As a summary of the two example architectures, Table 3 shows a comparison between the Alpha 21264 processor and the MIPS R10000 processor.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Alpha 21264</th>
<th>MIPS R1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Width</td>
<td>4 way</td>
<td>4 way</td>
</tr>
<tr>
<td>Integer Unit</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Address Unit</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Floating Point Units</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>L1 I-Cache</td>
<td>64KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>64KB</td>
<td>32KB</td>
</tr>
<tr>
<td>Integer Instruction Queue</td>
<td>20 entries</td>
<td>16 entries</td>
</tr>
<tr>
<td>Floating Point Instruction Queue</td>
<td>15 entries</td>
<td>16 entries</td>
</tr>
<tr>
<td>Integer Register</td>
<td>80</td>
<td>64</td>
</tr>
<tr>
<td>Floating Point Register</td>
<td>72</td>
<td>64</td>
</tr>
<tr>
<td>System Bus Width</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>Technology</td>
<td>0.35 µm</td>
<td>0.35 µm</td>
</tr>
</tbody>
</table>

0.4 VLIW (Very Long Instruction Word)

VLIW has been developed to exploit Instruction-Level Parallelism by using a long instruction word which contains multiple fixed number of operations. Those operations can be fetched, decoded, issued, and executed at the same time without causing any data or control hazards. Therefore, all operations within a single VLIW instruction must be absolutely independent.

0.4.1 Concept and Benefits

In a VLIW architecture, parallel execution of multiple instructions is made possible by issuing a long instruction word. A single long instruction word is designed to achieve simultaneous execution of a fixed number of multiple operations. Those operations must be independent of each other to avoid possible data hazards. Indeed, several independent instructions are integrated inside a very long instruction word.
The VLIW instruction is wide enough to allow the concurrent operation of multiple functional units. Its size normally ranges from 64 to 128 bits, and even up to 1024 bits. Figure 13 shows a typical format of VLIW instructions. Many bits on the long instruction enable a single instruction word to sufficiently control the several functional units directly and independently in every cycle.

Since it is the long instruction word which delivers the potential ILP to the hardware, a VLIW processor can be designed with a simpler hardware compared to an equivalent superscalar processor: it need not include the special units the run-time dependency check and instruction scheduling. The block diagram of a simple VLIW processor is shown in Figure 14.

A VLIW architecture is, by essence, meant to activate multiple functional units at the same time. Therefore, the VLIW compiler should uncover independent operations to be executed in parallel. This means that the compiler must perform a detailed analysis on the data-flow and control-flow at compile time (which is when the potential ILP is fixed). When the compiler cannot find enough instructions to fill in the width of the long instruction, it simply inserts a NOP.

When the interaction between the compiler and the hardware for instruction scheduling is considered, the fundamental differences between the VLIW and the superscalar design approaches can be explained as shown in Figure 15: in superscalar architectures, most of the scheduling is handled by hardware at run-time. However, in a VLIW architecture, the scheduling is decided by the compiler in a static way. Indeed, an intelligent compiler is the key component which ultimately decides the performance of a VLIW architecture.

Since the ILP within a basic block is quite limited, the VLIW architecture needs to examine more instructions to find more ILP. It is possibly achieved by looking at the instruction stream beyond the control-flow limits. For that purpose, several techniques such as loop unrolling and trace scheduling have been introduced in the VLIW design techniques. Those techniques will be explained shortly.
Compared to the dynamic scheduling of superscalar architectures, static scheduling does not require a complex scheduling logic. In addition, VLIW can uncover more parallelism by searching over a wider range of static code. Also, it is quite beneficial to know the source code structure to find parallelism in the VLIW architecture. However, several limitations such as long compilation time, not enough compatibility, and code explosion make VLIW architectures difficult to use in practice [20].

In conclusion, we can say that VLIW architectures do not have the hardware complexity of current superscalar architectures. However, they impose a much heavier burden on the compiler than superscalar architectures. This means that there exists a clear trade-off between the two approaches in the current high-performance microprocessor design. In the following part, more detailed techniques for the VLIW architecture is explained.

**Loop Unrolling**

Loop Unrolling [11] is one basic technique to find more Instruction Level Parallelism out of the sequential instruction stream. It is based on the idea of expanding the
multiple instances of a simple loop-body. Then, a wider range of instructions is at our disposal since we can have more instructions from the multiple iterations. It is most beneficial to unroll loops when the loop body does not include any complex control flow.

As seen in Figure 16, the original loop body has a sequence of seven instructions. Therefore, without loop unrolling, the compiler can only pick from a pool of seven instructions to uncover parallelism (assuming there is no branch prediction). Loop unrolling will maximize the size of instruction window across the control flow limits by expanding the loop body. The loop body can be expanded by unrolling the several loop iterations. For example, in Figure 16, the next four iterations of the loop can be integrated as the single loop of the second diagram. Now, the basic
Figure 16: An example of loop unrolling

Although we can have more instructions after loop unrolling, there are name dependencies between iterations. To reduce the false dependencies between instructions (since they may use the same register names), register renaming is imperatively before the final instruction stream can be obtained. However, some instructions might have loop-carried dependencies, which occur that later iterations of the loop body depend on the computation results of the previous iteration. Therefore, those dependencies must be carefully respected during register renaming.
Software Pipelining

Software pipelining is a technique which enables the later iterations of a loop to start before the previous iterations have finished the loop. Therefore, the later iterations of a loop can be executed, pipelined with the previous iterations. Of course, the loop unrolling technique also achieves the same scheduling as software pipelining, which is merely a subset of loop unrolling. However, in loop unrolling, a simple form of instruction pool is obtained from the multiple instances of the same loop body. The compiler can then schedule each of the instructions in turn. On the contrary, in software pipelining, the parallelism is obtained between the different parts of the same loop body by executing later iterations a couple of clock cycles later.

The main advantage of software pipelining over a simple loop unrolling technique is the following; loop unrolling requires overhead for each instance of an unrolled loop, whereas software pipelining only imposes start-up and wind-down overhead. For example, assume in a simple example that there are 100 iterations and that the loop body contains 4 instructions for the normal computation and 2 instructions for its control flow. If we unroll the loop over 4 iterations, those result in 16+2 instructions for an extended loop (the loop which includes 4 iterations in this example). Just assume that four iterations of an instruction can be executed in parallel, and the parallel execution of four instructions takes one clock cycle. Therefore, in the loop unrolling case, one unrolled loop execution requires 4 cycles for the computation and 2 cycles for the control instructions. Since the unrolled loop needs to be iterated 25 times, the total required cycle time is 150 cycles. However, we first can eliminate the control flow instructions in the software pipelining. Then, the first iteration starts alone and the second iteration starts when the second instruction of the first iteration starts. Both operations can be executed in parallel. Since execution can support up to four instructions in parallel, the fifth iteration can start just after the first iteration finishes (recall that one loop execution includes the execution of four instructions). In this particular case, the 100th iteration starts at 100 cycles and it requires 4 more clock cycles to finish. Therefore, 104 cycles are required for the overall execution.

Software pipelining sometimes provides an easier approach to parallelize a loop with loop-carried dependencies. In the previous example, there are no loop-carried dependencies. Therefore, each iteration of the original loop can be executed in parallel. However, if there existed loop-carried dependencies, those would have to be carefully taken into account. Figure 17 shows one example of software pipelining which includes a loop carried dependency; the second instruction $\text{Add } R4, R5, R2$ requires the updated value of R5 in the previous loop iteration. Therefore, simple loop unrolling should consider this dependency to schedule the instruction. As seen in the Figure 17, the software pipelined code can yield some added ILP. As seen in the...
example, software pipelining can be an easier approach to schedule parallel execution of a loop body with loop-carried dependencies; however, if the distance between the instructions for loop-carried dependencies is too high, software pipelining does not provide much of a benefit: the later iteration cannot be executed until the previous iteration finishes and provides the results.

![Software Pipelining Diagram](image)

Figure 17: An example of software pipelining

**Trace Scheduling**

Loop unrolling and software pipelining is quite beneficial to extend the range of the instruction window for a simple loop body. However, when the loop contains very complex control flow, it cannot be easily unrolled. The *Trace Scheduling* [8] technique has been developed to find more ILP across the basic block boundary even with a complex control flow. *Trace* is considered a large extended basic block
0.4. VLIW (VERY LONG INSTRUCTION WORD)

considering the control path which is frequently taken\cite{2}. This technique considers the branch instructions as jump instructions inside a trace, and it schedules the instructions inside the trace as a large basic block.

However, to guarantee the correct computation of the other path execution, Trace Scheduling requires some compensating code to be inserted for the case when the program flows out of the trace. Trace scheduling depends on the frequent occurrence of the Trace path. Since the VLIW compiler decides traces in a static way, the traces may not be sufficiently well predicted. Therefore, if the predicted path execution does not happen frequently, the compensation code overhead can be higher than the advantage given by trace scheduling.

The above three techniques can be used to increase the amount of parallelism by extending the instruction pool. All three approaches strongly depend on an appropriate and accurate control flow prediction; using them is beneficial only when the behavior of branches is predictable by the compiler. To cope more aggressively with dynamic events inside the control flow, ISA supported techniques such as conditional instructions and predication have been introduced in addition to the traditional VLIW concept. We will cover those techniques in the example architecture section.

0.4.2 Design Challenges

The largest challenge for VLIW architecture is the code explosion problem. Code explosion is caused by the need for the compiler to insert NOPs in the object code when the available parallelism of the code is not sufficient to fill each VLIW instruction. These NOPs may waste considerable amount of bits in the VLIW code.

Also, a large number of registers are required in the VLIW architecture due to transformations such as loop unrolling, register renaming, and trace scheduling. In addition, a large data transport capability between functional units and register file is required. Since the VLIW instructions are, by definition, very long, they will require a high bandwidth between the instruction cache and the fetch unit.

Although static scheduling has several advantages over dynamic scheduling, pure VLIW has not enjoyed a successful commercial implementation. The main weakness of the architectures has to do with backward compatibility; conventional compiler or binary must be considerably modified in the VLIW architectures. In fact, superscalar architectures have a huge advantage due to the compatibility which is one reason why they have been chosen by almost all processor vendors\cite{32}.

Indeed, the VLIW instruction should be adjusted to the hardware configuration. Therefore, the compiler should be notified the hardware characteristic in
advance. It consequently means that there is not even compatibility between different VLIW processors.

0.4.3 Example Architectures

The IA-64 ISA and Itanium Processor

The Intel Itanium \[28\] is the first commercial processor which implements the IA-64 ISA \[12\]. The IA-64 ISA is a 64-bit instruction set architecture implementing EPIC (Explicitly Parallel Instruction Computing) \[27\], a design which is a joint project of Intel and HP. Although the Itanium processor is not a pure VLIW processor, many features of Itanium resemble that of a VLIW architecture.

First of all, a 128-bit wide instruction bundle is used for the long instruction word. Indeed, a single bundle holds three instructions. Each base instruction requires 41-bit wide, and 123 bits on a bundle are used for those three instruction. The remaining 5 bits can be used as the template\[28\]. Those template bits are used to deliver the execution information of the bundle. The template bits are very useful to avoid empty operations (NOPs), and allow higher flexibility in the exploitation of parallelism. Figure 18 shows the format of a three-instruction bundle.

![Figure 18: Format of an instruction bundle](image)

The IA-64 provides 128 65-bit general registers with 65 bits for each register and 128 82-bit floating-point registers. It also has 64 1-bit registers designed to indicate the result of conditional expression instructions. In addition, eight 64-bit
branch registers are implemented for function call linkage and return. This ISA also provides space for up to 128 64-bit special-purpose application registers.

Two distinct features of the IA-64 are predication and load speculation. Originally, the predication requires the instructions for predicated execution. For example, the instruction itself detects a conditional variable to decide whether the instruction can be executed or not. In other words, the instruction is executed only when its predication is true. However, in the IA-64 architecture, all the branch paths are eagerly executed and the instructions inside those paths are committed only when the predication is true. Under the predication, the execution of each instruction needs to be accompanied by a dedicated predicate register among 64 predicated registers. Then, the instructions can only be committed when the specified predicate register is confirmed as \textit{true}.

\begin{verbatim}
Source Code
if (a = b)
a = 0;
else
b = 0;
\end{verbatim}

\begin{verbatim}
Normal Code without Predication
beq r1, r2, L1
J   L2
L1:
mov r1, 0
J   L3
L2:
mov r2, 0
L3:
\end{verbatim}

\begin{verbatim}
Predicated Code
CMPEQ r1, r2, p1/p2   // check (a == b) sets p1, p2
[p1] MOV r1, 0        // if (p1 == true) r1 = 0
[p2] MOV r2, 0        // if (p2 == true) r2 = 0
\end{verbatim}

Figure 19: Predication Example

Figure 19 shows a simple example of the predicated execution. The \textit{CMPEQ} instruction sets the value of the predication registers based on the comparing operation. Then, the later predicated instructions are committed according to the con-
The predication in IA-64 means both paths of a branch instruction must execute in parallel. After finding the branch result, only the correct path execution is committed and can affect the program execution. Indeed, it is very similar to the multi-path execution or eager execution [33].

The other technique called speculative load execution of the IA-64 enables to move load instructions across the control flow boundary. Speculative loading, indeed, has been proposed to reduce the access latencies of load instructions. For speculative loading, a load instruction is converted into two other instructions: the speculative load instruction which is moved before a branch instruction and the checking instruction which stays in the original location to check the proper execution of the program flow. An example of the speculative load execution is shown in Figure [20]. The lw.s instruction is a speculative load instruction which is moved across a control flow boundary, and the chk.s instruction is the checking instruction in the original location. The checking instruction confirms the validity of the speculative execution.

The first IA-64 processor, Itanium, is implemented with a six-wide and 10-stage pipeline. It is designed to operate at a frequency of 800 MHz. It has four integer units, four multimedia units, two load/store units, three branch units, two extended-precision floating-point units, and two additional single-precision floating-point units. Two bundles are fetched in every clock cycle. Therefore, the machine can fetch six instructions at each clock cycle.
The three architecture styles just discussed have significantly impacted the design of modern processors, each to their own degree. While superscalar architectures have dominated the commercial field, superpipelining techniques have had a strong influence on the bottom design of processors, a number of VLIW compiler techniques have made their imprint on the design of compilers for conventional architectures.

Superpipelining could appear to be the simplest technique to implement. However, it is inherently tied to low-level implementation issues and is limited by the amount of hardware partitioning which can be attained.

VLIW has the advantage of design simplicity as it pushes much of the complexity to the compiler stage. However, code explosion, requirements for high bandwidth between the instruction caches and the fetch units, and lack of backward compatibility has reduced the potential of this model.

Superscalar has known the most success of all three techniques and has indeed satisfied the users’ ever increasing hunger for more computing power. However, it is intrinsically restricted by two basic considerations. First, the ability to uncover sufficient ILP (Instruction-Level Parallelism) is seriously questionable for wide superscalar processors. Second, with larger scale of integration, clock skew and signal propagation delays become a major consideration (a pulse may take dozens of clock cycles to propagate across a chip). This may, in many cases, reduce the effective clock frequency which can be achieved.

Both these problems have been somewhat successfully attacked by a number of hardware and software techniques as we have described. Future models [17] which exploit Thread-Level Parallelism (TLP), include Simultaneous MultiThreading (SMT) and Chip MultiProcessor (CMP). They hold much promise for the design of future architectures.
Bibliography


