International Journal of Engineering & Technology
CENTRE OF PROFESSIONAL RESEARCH PUBLICATIONS
ISSN 2049-3444
International Journal of Engineering and Technology Volume 1 No. 3, December, 2011

Performance Measures of Superscalar Processor

K.A.Parthasarathy
Global Institute of Engineering and Technology
Vellore, Tamil Nadu, India

ABSTRACT

In this paper the author describes about superscalar processor and its architecture. A superscalar architecture is one in which several instructions can be initiated simultaneously and executed independently. pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment. Superscalar architectures include all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage. They have the ability to initiate multiple instructions during the same clock cycle. Superscalar processing is the latest in a long series of innovations aimed at producing ever-faster microprocessors. By exploiting instruction-level parallelism, superscalar processors are capable of executing more than one instruction in a clock cycle. This paper discusses the microarchitecture of superscalar processors. We begin with a discussion of the general problem solved by superscalar processors: converting an ostensibly sequential program into a more parallel one. The principles underlying this process, and the constraints that must be met, are discussed. The paper then provides a description of the specific implementation techniques used in the important phases of superscalar processing. The major phases include: i) instruction fetching and conditional branch processing, ii) the determination of data dependences involving register values, iii) the initiation, or issuing, of instructions for parallel execution, iv) the communication of data values through memory via loads and stores, and v) committing the process state in correct order so that precise interrupts can be supported. Examples of recent superscalar microprocessors, the MIPS R10000, the DEC 21164, and the AMD K5 are used to illustrate a variety of superscalar methods. The goal of a superscalar microprocessor is to execute multiple instructions per cycle. Instruction-level parallelism (ILP) available in programs can be exploited to realize this goal. Unfortunately, this potential parallelism will never be utilized if the instructions are not delivered for decoding and execution at a sufficient rate. A high performance fetching mechanism is required.

Key words: *Superscalar, ILP, Pipelining, RISC*

1. INTRODUCTION

Super pipelining is based on dividing the stages of a pipeline into sub stages and thus increasing the number of instructions which are supported by the pipeline at a given moment. By dividing each stage into two, the clock cycle period will be reduced to the half,; hence, at the maximum capacity, the pipeline produces a result every 2 s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. A solution to further improve speed is the superscalar architecture. Superscalar processing, the ability to initiate multiple instructions during the same clock cycle, is the latest in a long series of architectural innovations aimed at producing ever faster microprocessors. Introduced at the beginning of this decade, superscalar microprocessors are now being designed and produced by all the microprocessor vendors for high-end products. Although viewed by many as an extension of the Reduced Instruction Set Computer (RISC) movement of the 1980s, superscalar implementations are in fact heading toward increasing complexity. And superscalar methods have been applied to a spectrum of instruction sets, ranging from the DEC Alpha, the "newest" RISC instruction set, to the decidedly non-RISC Intel x86 instruction set.

A typical superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analyzed for data dependences, and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as *dynamic instruction scheduling*. Upon completion, instruction results are re-sequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as instruction level parallelism (ILP).

The Superscalar Digital Signal Processor is a 32-bit Superscalar processor with a RISC style instruction set. It is pipelined and has a fetch bandhwidth of four instructions per cycle. The architecture of the SDSP is shown in fig.1. It is divided into three basic units: the

International Journal of Engineering & Technology
CENTRE OF PROFESSIONAL RESEARCH PUBLICATIONS
ISSN 2049-3444
International Journal of Engineering and Technology Volume 1 No. 3, December, 2011

Instruction Unit (IU), the Scheduling Unit (SU) and the Execution Unit (EU).

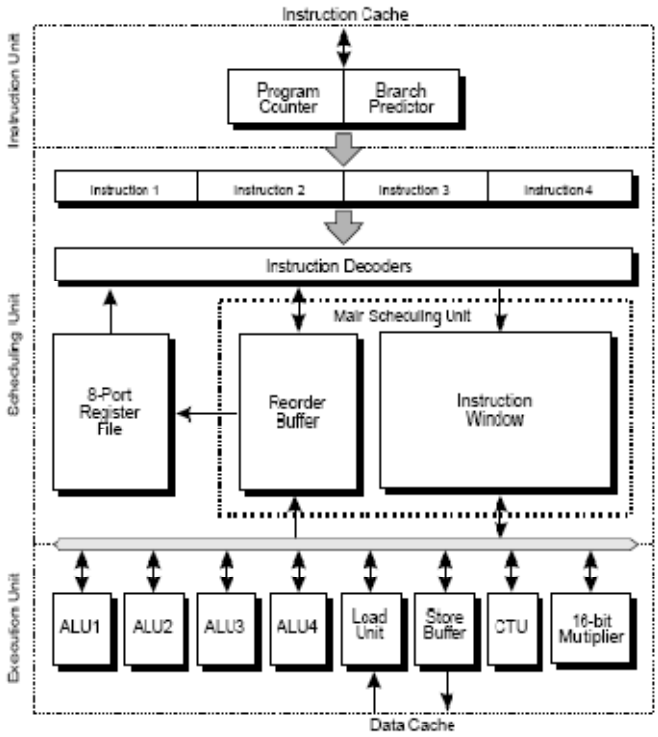

Figure 1: SDSP Architectural Organization

2. THE INSTRUCTION PROCESSING MODEL

Because hardware and software evolve, it is rare for a processor architect to start with a clean slate; most processor designs inherit a legacy from their predecessors. Modern superscalar processors are no different. A major component of this legacy is *binary compatibility*, the ability to execute a machine program written for an earlier generation processor. When the very first computers were developed, each had its own instruction set that reflected specific hardware constraints and design decisions at the time of the instruction set's development. Then, software was developed for each instruction set. It did not take long, however, until it became apparent that there were significant advantages to designing instruction sets that were compatible with previous generations and with different models of the same generation. For a number of very practical reasons, the instruction set architecture, or binary machine language level, was chosen as the level for maintaining software compatibility.

The sequencing model inherent in instruction sets and program binaries, the *sequential execution model*, closely resembles the way processors were implemented many years ago. In the sequential execution model, a program counter is used to fetch a single instruction from memory. The instruction is then executed -- in the process, it may load or store data to main memory and operate on registers held in the processor. Upon completing the execution of the instruction, the processor uses an incremented program counter to fetch the next instruction, with sequential instruction processing occasionally being redirected by a conditional branch or jump. Should the execution of the program need to be interrupted and restarted later, for example in case of a page fault or other exception condition, the state of the machine needs to be captured. The sequential execution model has led naturally to the concept of a *precise state*. At the time of an interrupt, a precise state of the machine (architecturally-visible registers and memory) is the state that would be present if the sequential execution model was strictly followed and processing was stopped precisely at the interrupted instruction. Restart could then be implemented by simply resuming instruction processing with the interrupted instruction. Today, a computer designer is usually faced with maintaining binary compatibility, i.e., maintaining instruction set compatibility *and* a sequential execution model (which typically implies precise interrupts1). For high performance, however, superscalar processor implementations deviate radically from sequential execution -- much has to be done in parallel. As a result, the program binary nowadays should viewed as a specification of *what* has to be done, not *how* it is done in reality. A modern superscalar microprocessor takes the sequential specification as embodied in the program binary and removes much of the non-essential sequentiality to turn the program into a parallel, higher-performance version, yet the processor retains the outward *appearance* of sequential execution.

3. INSTRUCTION EXECUTION CYCLE

The execution of a single machine instruction can be divided into a sequence of individual operations called the *instruction execution cycle*. Before executing, a program is loaded into memory. The *instruction pointer* contains the address of the next instruction. The *instruction queue* holds a group of instructions about to be executed. Executing a machine instruction requires three basic steps: *fetch, decode,* and *execute.* Two more steps are required when the instruction uses a memory operand: *fetch operand* and *store output operand*. Each of the steps is described as follows:

• ***Fetch:*** The control unit fetches the instruction from the instruction queue and increments the instruction pointer (IP). The instruction pointer is also known as the *program counter*.

• ***Decode:*** The control unit decodes the instruction's function to determine what the instruction will do. The instruction's input operands are passed to the arithmetic logic unit (ALU), and signals are sent to the ALU indicating the operation to be performed.

• ***Fetch operands:*** If the instruction uses an input operand located in memory, the control unit uses a *read* operation to retrieve the operand and copy it into internal registers. Internal registers are not visible to user programs.

• ***Execute:*** The ALU executes the instruction using the named registers and internal registers as operands and sends the output to named registers and/or memory. The ALU updates status flags providing information about the processor state.

• ***Store output operand:*** If the output operand is in memory, the control unit uses a write operation to store the data.

The sequence of steps can be expressed neatly in pseudocode:

- loop
- fetch next instruction
- advance the instruction pointer (IP)
- decode the instruction
- if memory operand needed, read value from memory
- execute the instruction
- if result is memory operand, write result to memory
- continue loop

4. SUPERSCALAR ARCHITECTURE

A superscalar, or multi-core processor has two or more execution pipelines, making it possible for two instructions to be in the execution stage at the same time. To better understand why a superscalar processor would be useful, let's consider the preceding pipelined example, in which we assumed that the execution stage (S4) required a single clock cycle. That was an overly simplistic approach. What would happen if stage S4 required two clock cycles? Then a bottleneck would occur, shown in Figure 2. Instruction I-2 cannot enter stage S4 until I-2 has completed the stage, so I-2 has to wait one more cycle before entering stage S4. As more instructions enter the pipeline, wasted cycles occur (shaded in gray). In general, for k stages (where one stage requires two cycles), n instructions require (k < 2n < 1) cycles to process.

Stages

exe

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3	I-3	I-2	I-1			
4		I-3	I-2	I-1		
5			I-3	I-1		
6				I-2	I-1	
7				I-2		I-1
8				I-3	I-2	
9				I-3		I-2
10					I-3	
11						I-3

Figure 2: Pipelined Execution Using a Single Pipeline

A superscalar processor permits multiple instructions to be in the execution stage at the same time. For *n* pipelines, *n* instructions can execute during the same clock cycle. The Intel Pentium, with two pipelines, was the first superscalar processor in the IA-32 family. The Pentium Pro processor was the first to use three pipelines.

Figure 3 shows an execution scheme with two pipelines in a six-stage pipeline. We assume stage S4 requires two cycles. Odd-numbered instructions enter the *u-pipeline* and even-numbered instructions enter the *v-pipeline*. Wasted cycles are removed, so *n* instructions can be executed in (*k* +*n*) cycles, where *k* indicates the number of stages.

Stages

S4

Cycles	S1	S2	S3	u	v	S5	S6
1	I-1						
2	I-2	I-1					
3	I-3	I-2	I-1				
4	I-4	I-3	I-2	I-1			
5		I-4	I-3	I-1	I-2		
6			I-4	I-3	I-2	I-1	
7				I-3	I-4	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

Figure 3: Superscalar 6-Stage Pipelined Processor

5. INSTRUCTION FETCHING AND BRANCH PREDICTION

The instruction fetch phase of superscalar processing supplies instructions to the rest of the processing pipeline. An *instruction cache*, which is a small memory containing recently-used instructions [52], is used in almost all current processors, superscalar or not, to reduce the latency and increase the bandwidth of the instruction fetch process. An instruction cache is organized into *blocks* or *lines* containing several consecutive instructions; the program counter is used to search the cache contents associatively to determine if the instruction being addressed is present in one of the cache lines. If so, there is a cache *hit*; if not, there is a *miss* and the line containing the instruction is brought in from main memory to be placed in the cache.

For a superscalar implementation to sustain the execution of multiple instructions per cycle, the fetch phase must be able to fetch multiple instructions per cycle from the cache memory. To support this high instruction fetch bandwidth, it has become almost mandatory to separate the instruction cache from the data cache (although the PowerPC 601 provides an interesting counter example). The number of instructions fetched per cycle should at least match the peak instruction decode and execution rate and is usually somewhat higher. The extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer

than the maximum number of instructions can be fetched. For example, if a branch instruction transfers control to an instruction in the middle of a cache line, then only the remaining portion of the cache line contains useful instructions. Some of the superscalar processors have taken special steps to allow wider fetches in this case, for example by fetching from two adjacent cache lines simultaneously. Using this buffer, the fetch mechanism can build up a "stockpile" to carry the processor through periods when instruction fetching is stalled or restricted. The default instruction fetching method is to increment the program counter by the number of instructions fetched, and use the incremented program counter to fetch the next block of instructions. In case of branch instructions which redirect the flow of control, however, the fetch mechanism must be redirected to fetch instructions from the branch target. Because of delays that can occur during the process of this redirection, the handling of branch instructions is critical to good performance in superscalar processors. Processing of conditional branch instructions can be broken down into the following parts:

1) recognizing that an instruction is a conditional branch,
2) determining the branch outcome (taken or not-taken),
3) computing the branch target,
4) transferring control by redirecting instruction fetch (in the case of a taken branch).

Specific techniques are useful for handling each of the above.

6. HANDLING MEMORY OPERATIONS

Memory operations need special treatment in superscalar processors. In most modern RISC instruction sets, only explicit load and store instructions access memory, and we shall use these instructions in our discussion of memory operations (although the concepts are easily applied to register-storage and storage-storage instruction sets). To reduce the latency of memory operations, memory hierarchies are used. The expectation is that most data requests will be serviced by data cache memories residing at the lower levels of the hierarchy. Virtually all processors today contain a data cache, and it is rapidly becoming commonplace to have multiple levels of data caching, e.g. a small fast cache at the *primary* level and a somewhat slower, but larger, *secondary* cache. Most microprocessors integrate the primary cache on the same chip as the processor; notable exceptions are some of processors developed by HP and the high-end IBM POWER series.

Unlike ALU instructions, for which it is possible to identify during the decode phase the register operands that will be accessed, it is not possible to identify the memory locations that will be accessed by load and store instructions until after the issue phase. The determination of the memory location that will be accessed requires an *address calculation*, usually an integer addition. Accordingly, load and store instructions are issued to an execute phase where address calculation is performed. After address calculation, an *address translation* may be required to generate a physical address. A cache of translation descriptors of recently accessed pages, the *translation lookaside buffer (TLB)*, is used to speed up this address translation process. Once a valid memory address has been obtained, the load or store operation can be submitted to the memory. It should be noted that although this suggests address translation and memory access are done in series, in many superscalar implementations these two operations are overlapped -- the initial cache access is done in parallel with address translation and the translated address is then used to compare with cache tag(s) to determine if there was hit.

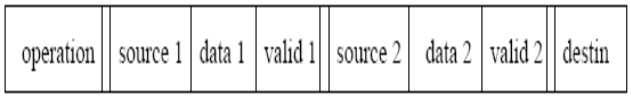

Fig 4: A typical reservation station

There are entries for the *operation* to be performed and places for each *source* designator, its *data* value, and a *valid* bit indicating that the data value is present in the reservation station. As an instruction completes and produces result data, a comparison of the instruction's destination designator is made with the source designators in the reservation station to see if instruction in the station is waiting for the data; if so, the data is written into the value field, and the corresponding valid bit is set. When all the source operands are valid, the instruction in the reservation station is ready to issue and begin execution. At the time it begins execution, the instruction takes its destination designator along, to be used later for its reservation station comparisons.

7. CONCLUSION

Both performance and compatibility have driven the development of superscalar processors. They implement a sequential execution model although the actual execution of a program is far from sequential. After being fetched, the sequential instruction stream is torn apart with only true dependences holding the instructions together. Instructions are executed in parallel with a minimum of constraints. Meanwhile enough information concerning the original sequential order is retained so that the instruction stream can conceptually be squeezed back together again should there be need for a precise interrupt. A number of studies have been done to determine the performance of superscalar methods. Because the hardware and software assumptions and benchmarks vary widely, so do the potential speedups -- ranging from close to 1 for some program/hardware combinations to many 100s for others. In any event, it is clear that there is something to be gained, and every microprocessor

manufacturer has embraced superscalar methods for high-end products.

REFERENCES

[1] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger, ''Cray X-MP: The Birth of a Supercomputer,'' *IEEE Computer*, vol. 22, pp. 45-54, January 1989.

[2] C. G. Bell, ''Multis: a New Class of Multiprocessor Computers,'' *Science*, vol. 228, pp. 462-467, April 1985.

[3] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith, ''The IBM System/360 Model 91: Storage System,'' *IBM Journal*, vol. 11, pp. 54-68, January 1967.

[4] Werner Bucholz, ed., *Planning a Computer System*. New York: McGraw-Hill, 1962. [5] B. Case, ''Intel Reveals Pentium Implementation Details,'' *Microprocessor Report*, pp. 9-13, Mar. 29, 1993.

[5] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, ''A VLIW Architecture for a Trace Scheduling Compiler,'' *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August 1988.

[6] T. M. Conte, P. M. Mills, K. N. Menezes, and B. A. Patel, ''Optimization of Instruction Fetch Mechanisms for High Issue Rates,'' *Proc. 22nd Annual International Symposium on Computer Architecture*, pp. 333-344, June 1995.

[7] K. Diefendorff and M. Allen, ''Organization of the Motorola 88110 Superscalar RISC Microprocessor,'' *IEEE Micro*, vol. 12, April 1992.

[8] P. K. Dubey and M. J. Flynn, ''Optimal Pipelining,'' *Journal of Parallel and Distributed Computing*, vol. 8, pp. 10-19, 1990.

[9] L. Gwennap, ''PPC 604 Powers Past Pentium,'' *Microprocessor Report*, pp. 5-8, Apr. 18, 1994.

[10] L. Gwennap, ''MIPS R10000 Uses Decoupled Architecture,'' *Microprocessor Report*, pp. 18-22, Oct. 24, 1994.

[11] E. Hao, P.-Y. Chang, and Y. N. Patt, ''The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited,'' *Proc. 27th Int. Symp. on Microarchitecture*, pp. 228-232, December 1994.

[12] S. Wallace and N. Bagherzadeh. Instruction fetching mechanisms for superscalar microprocessors. In *Euro-Par '96*, August 1996.

[13] D.A.Gilbert and J.D.Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Async '97*, pages 2–11. IEEE Computer Society Press, Apr. 1997.

[14] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronousMIPS R3000microprocessor. In *AdvancedResearch in VLSI*, pages 164–181, Sept. 1997.