UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

The Minimal Pipeline Architecture

Martti Forsell

Report A-1995-3

ACM C.1.2
UDK 681.3.02
ISSN 0789-7316
ISBN 951-708-361-0

The Minimal Pipeline Architecture

Martti J. Forsell

Department of Computer Science, University of Joensuu, PB-111, SF-80101 Joensuu, Finland

Phone + 358 - 73 - 1511
Fax + 358 - 73 - 151 3290
Email mforsell@cs.joensuu.fi

September 11, 1995

Abstract

Pipelining is used in almost all recent processor architectures to increase the performance. It is, however, difficult to achieve the theoretical speedup of pipelining, because code dependencies cause delays in execution. Processor designers must use complex techniques like forwarding, register renaming and branch prediction to reduce the loss of performance due to this problem.

In this paper we introduce abstract Minimal Pipeline Architecture (MPA) featuring special two level pipelining. MPA offers interesting advantages over architectures using deeper pipelines: Pipeline delays as well as register file bandwidth problems are absent in MPA. Also the utilization of functional units is better in MPA than in basic RISC machines. A preliminary performance evaluation of MPA is given.

Keywords: Computer architectures, pipelining, VLIW

INTRODUCTION

Pipelining is very popular technique in increasing the performance of processors. This is because ideal pipelining gives the speedup of the number of pipeline stages with only marginal hardware effort. In practice it is, however, difficult to achieve the speedup of ideal pipelining, because data dependencies and branches cause delays in execution [Kogge81, Hennessy90, Flynn95].

Processor designers must use complex techniques like forwarding, register renaming and branch prediction to reduce the loss of performance due to this problem [Tomasulo67, Smith84, Johnson89].

This paper introduces four new techniques for minimizing the performance penalty of pipelining—general forwarding, simple instruction set, uncoded instructions and fast branching. They both reduce the depth of deep pipelines and eliminate unnecessary pipelining. These techniques are applied to a basic VLIW core to design a new abstract architecture featuring two level pipelining. We call the obtained architecture as Minimal Pipeline Architecture (MPA).

MPA offers interesting advantages over architectures using deeper pipelines: Pipeline delays as well as register file bandwidth problems are absent in MPA. Also the utilization of functional units is

better in MPA than in basic pipelined RISC machines.

The paper gives also performance, code size and functional unit utilization comparisons between two instances of MPA and basic pipelined and superscalar architectures.

DEPENDENCIES AND EXECUTION MODELS

There are two basic models for exploiting instruction level parallelism—*pipelined* and *superscalar* execution model.

In the pipelined model [Kogge81, Flynn95] the execution of instructions is divided into parts called pipeline stages, such that the different parts of consecutive instructions can be executed simultaneously.

In the superscalar model [Johnson89, Hennessy90] multiple instructions are executed in multiple functional units simultaneously. The pure superscalar execution model is, however, very rare. It is usually combined with the pipelined model such that the execution of instructions over functional units is pipelined.

The execution of instructions in both models can be divided into two clearly separate parts—instruction fetch (IF) and instruction execute (IE)—which are always committed sequentially. Furthermore, the IE part is usually divided into three or more subparts.

Let us denote the number of these subparts increased by one (representing the effect of IF part) with S. In the pipelined execution model S clearly equals the number of pipeline stages.

Non-pipelined execution

An execution model is *non-pipelined* if no explicit division of instruction execution into parts is made, i.e., the latency of instruction equals the length of clock cycle.

Dependencies do not cause delays in the scalar non-pipelined execution model, because instructions are executed in order and no overlapping happens. This model is, however, rather inefficient—the ideal S-level pipelined execution model performs S times better than this model.

Pipelined execution

Let us assume that we are dealing with the S-level pipelined execution model without any specific knowledge about the IE part of the execution.

It is quite obvious that data dependencies in the code cause delays in execution if IE parts of dependent instructions overlap (see Figure 1).

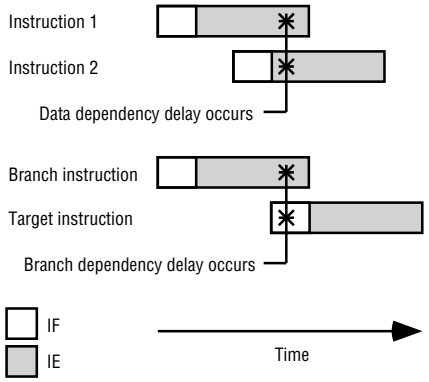

Figure 1. Situations where data and branch dependency delays occur.

Let l_X be the latency of the part X. Now we can say, that no data dependency delay exists if following condition holds:

$$l_{IE} \leq l_{IF} \qquad (1)$$

Similarly, no delay is caused by a branch instruction if the IF part of the branch target instruction does not overlap with the IE part of the branch instruction:

$$l_{IE(\text{Branch instruction})} \leq 0 \qquad (2)$$

These two conditions suggest that dependency delays can be avoided with two

level pipelining where the latency of IE part of branch instructions is zero.

Superscalar execution

In the superscalar model data and branch dependencies cause similar delays as in the pipelined model.

Some advanced superscalar architectures use internal buffers, which allow out of order execution of decoded instructions [Johnson89]. This leads to good exploitation of available instruction level parallelism, because execution is now limited only by actual data dependencies.

Unfortunately fetching, decoding and buffering of decoded instructions take some time causing a lot of branch delays. The performance degradation due to these delays can be partially eliminated by branch prediction and speculative execution [Smith84, Johnson89].

A much simpler alternative to a superscalar processor with out of order execution and branch prediction is a *Very Long Instruction Word* (VLIW) processor [Fisher83]. A VLIW processor consists of multiple functional units that operate in parallel controlled by a single sequencer. Advanced compilers are used to parallelize the program before the execution in a VLIW processor [Fisher81, Chang91].

We selected VLIW-style architecture as our choice of superscalar model design, because it offers good performance with very simple hardware.

IMPLEMENTATION TECHNIQUES

This section describes the four new key implementation techniques we used in designing MPA—general forwarding, simple instruction set, uncoded instruction format and fast branching.

General forwarding

The pipeline depth of four or more stages is very usual in current RISC machines. This causes a lot of data dependency delays, because the left side of equation 1 becomes at least three times greater than the right side.

An old hardware technique called *forwarding* eliminates the data dependency delays presuming the actual operation is executed in a single fixed stage of the pipeline [Hennessy90] (see Figure 2).

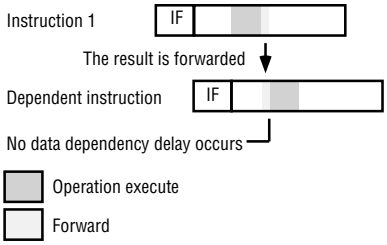

Figure 2. Forwarding eliminates data dependency delays.

We get a novel technique called *general forwarding* by applying the idea of forwarding to the selection of all operands: The results of functional units including the contents of registers are passed to the input multiplexers of functional units. The correct operands are selected by appropriate bits in the operation code of an instruction.

The use of general forwarding requires simultaneous access to all registers, which requires quite complex wiring with ordinary multiported register files.

Simultaneous access to all registers can be implemented by a novel register file organization called *distributed register file*, in which single registers are treated as separate functional units that are connected to each other by a cross-bar switch-like forwarding mechanism (see Figure 3).

In a processor using general forwarding the separate operand select stage is unnecessary, because the forwarding machinery now

selects operands between the operations of consecutive stages.

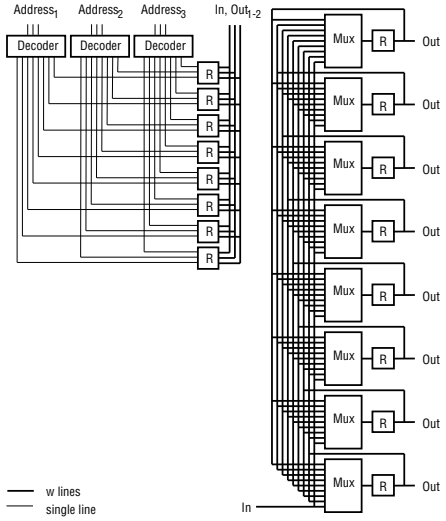

Figure 3. A conventional register file with two read ports and one write port (left). A distributed register file with one external input and eight outputs (right).

Simple instruction set

The instruction sets of basic RISC processors do not support execution in two level pipeline properly, because most instructions require sequential involvement of more than just one operational unit (see Figure 4).

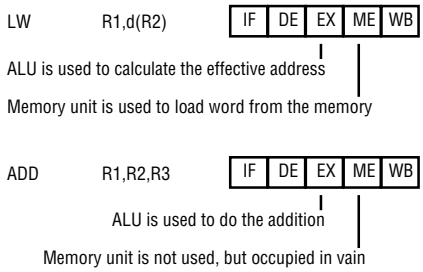

Figure 4. Some instructions occupy many functional units in DLX pipeline [Hennessy90].

The problem disappears by using a *simple instruction set*: Instructions involving the use of multiple functional units are divided into multiple instructions occupying exactly one functional unit and using always absolute or indirect addressing.

A processor using simple instruction set potentially offers better utilization of functional units than a conventional RISC processor using typical instruction set. This is because one has more accurate control of functional units with the simple instruction set. There are for example no situations where an instruction occupies a functional unit in vain (see the ADD instruction in figure 4).

Uncoded instruction format

Instruction coding is used in almost all current processors to reduce the size of program code. This does not come for free, because usually the processor must decode instructions in a separate pipeline stage before they can be executed.

This stage can be eliminated by using an *uncoded instruction format*, in which instructions are left completely uncoded: The operation code of an instruction is fetched directly into the operation register and separate bitfields in the code control directly the operation of different parts of the processor.

The negative consequence of using uncoded instruction format is that according to our tests the width of instruction word increases by the factor of four to ten.

Fast branching

Equation 2 reveals that the main reason for control delays is the long latency of branch instructions.

Fast branching is a novel technique in which general forwarding is applied also to the operation of sequencer: The result bit of the compare unit controls forwarding mechanism, which selects the next instruction memory fetch address from the

next instruction address calculated by sequencer adder and absolute target address in operation code. All this happens before the next instruction fetch is started.

Fast branching eliminates branch delays presuming the operation is executed in stage two. Otherwise one or more delay slots are required between compare instructions and branch instructions (see Figure 5).

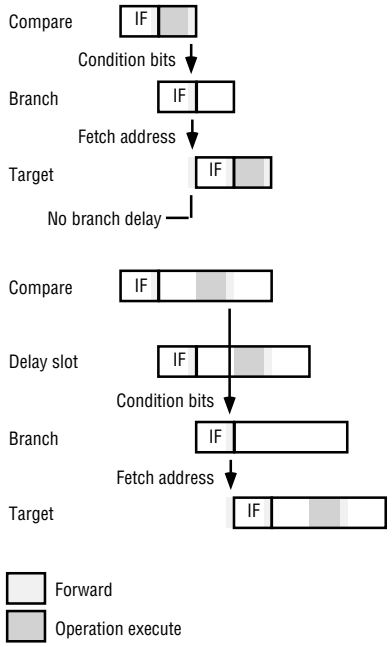

Figure 5. Fast branching requires the operation is executed in stage two otherwise delay slots are added.

The use of fast branching has two disadvantages: The program loader must calculate branch addresses while loading programs into the memory and the use of absolute addressing increases the size of the program slightly.

ARCHITECTURE

In order to outline an abstract architecture with two level pipelining and fast branching, we apply the four described implementation techniques to the basic VLIW core [Fisher83]. We call the obtained architecture as the *Minimal Pipeline Architecture* (MPA) [Forsell94].

The main parts of a MPA processor are *a* arithmetic and logic units (ALU_0-ALU_{a-1}), arithmetic unit for condition codes (AIC), *m* memory units (M_0-M_{m-1}), *r* registers (R_0-R_{r-1}) and sequencer (S). Parts are connected to each other by a large cross-bar switch, which acts as general forwarding mechanism (see Figure 6).

MPA uses VLIW-style instruction format, where a single instruction consists of two word operands and several subinstructions that are assigned to appropriate functional units. The types of subinstructions are arithmetic and logical subinstructions (A), compare subinstructions (AIC), load and store subinstructions (M), write back subinstructions (R), and control subinstructions (S) (see Figure 7). In order to avoid separate decoding stage in MPA pipeline, instructions are uncoded and operands are always directly forwarded to appropriate functional units.

The instruction set of MPA is derived from DLX [Hennessy90] such that MPA subinstructions use only one functional unit. DLX instructions involving sequential use of many functional units are divided into separate MPA subinstructions.

MPA processor has a single addressing mode called *forwarded absolute addressing*. The address of a memory reference is always an absolute value, which is selected through the cross-bar from ALU operation result, memory unit result, data word operands or register contents. The more complicated addressing modes must be built from several sequential instructions: The address must first be calculated using ALU and then the result must be forwarded to memory unit.

MPA Pipeline

The pipeline of MPA consists of two stages: Instruction Fetch (IF) and Execute (EX).

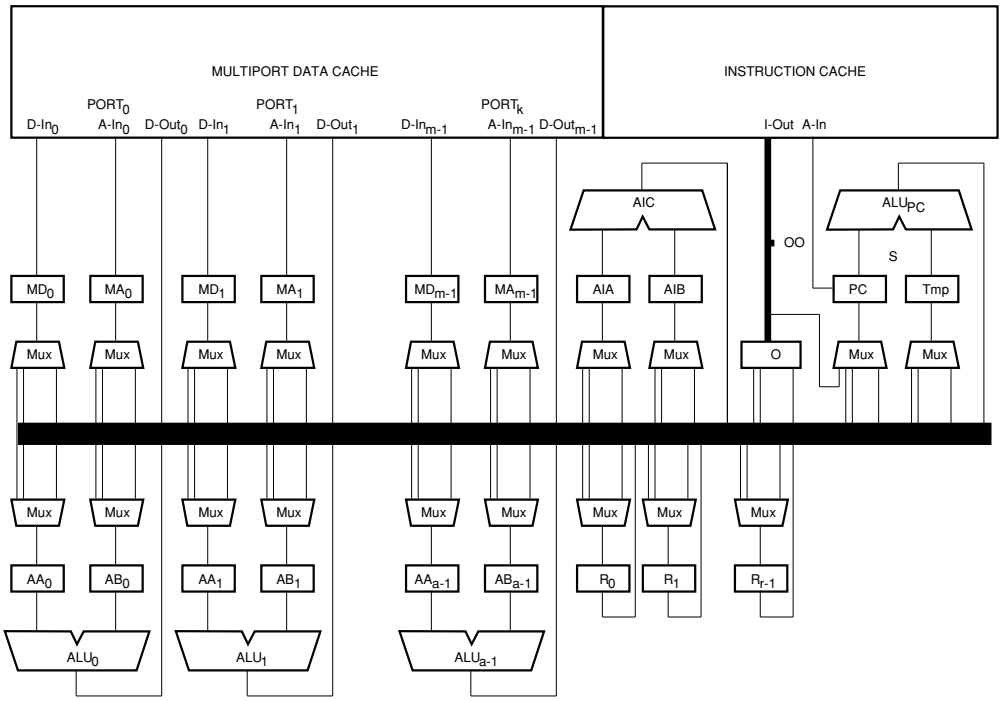

Figure 6. A block diagram of a MPA processor.

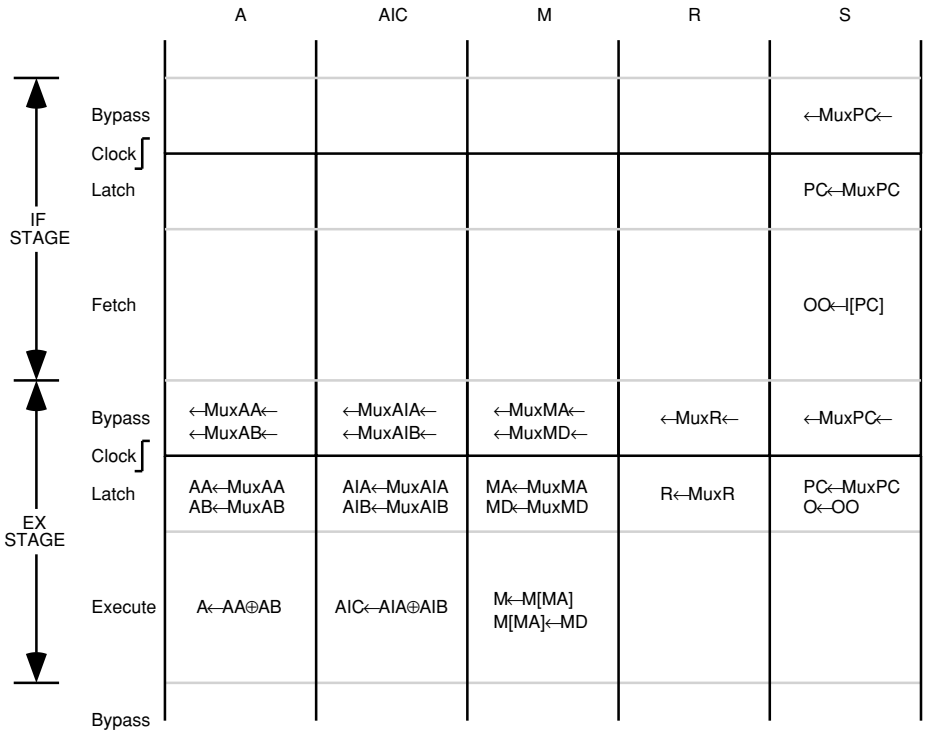

Figure 7. The operations for different types of subinstructions of M5 processor. The indexes are not shown.

Both IF and EX stages are divided into three phases: Bypass Phase (BP), Latch Delay Phase (LD) and Execution Phase (EX) (see Figure 7).

The BP and LD phases of the IF stage are controlled by the previous instruction, which will select the correct value for PC according to operation code and condition codes. During the EX phase of the IF stage the processor fetches an operation code of an instruction from the instruction cache. This value on the instruction bus (OO) is used to control the beginning of instruction execution before the operation code is actually written into operation register (O) at the LD phase of the EX stage.

During the BP phase of the EX stage the operands of an instruction are selected and forwarded to operand registers of appropriate functional units. During the LD phase of the EX stage the values provided by the forwarding mechanism are written into operand registers of the functional units. The write back and control transfer operations are completed with this phase.

During the EX phase of the EX stage the functional units except the register units and sequencer commit their operations: The ALU executes an ALU subinstruction, the AIC unit executes a compare subinstruction, the memory unit executes a memory reference subinstruction. By the end of execution phase the results of the executed operations are ready for the bypassing phase of the next instruction.

EVALUATION

We evaluated the performance, code size and utilization of functional units of four processors—DLX, superDLX, M5 and M11 (see Figure 8).

The first two were included for comparison purposes as representatives of basic pipelined and superscalar processors.

	DLX	sDLX	M5	M11
Opcode length	32	128	320	512
Instr scheduling	pl	ss	VLIW	VLIW
Pipeline stages	5	5	2	2
Functional units	5	13	5	11
ALU	1	2+2+2	1	4
AIC (CMP)	1	1	1	1
MU	1	4	1	4
REG	1	1	1	1
SEQ	1	1	1	1

Figure 8. Evaluated processors.

DLX is an experimental load/store RISC architecture featuring basic five level pipeline [Hennessy90]. SuperDLX is an experimental superscalar processor derived from DLX architecture [Moura93] featuring out of order execution and out of order completion of instructions and branch prediction. M5 and M11 are instances of MPA containing five and eleven functional units respectively [Forsell94].

The numbers are chosen such that DLX and M5 have approximately the same amount of processing resources. Similarly superDLX and M11 have the same amount of resources except superDLX has two additional shift units.

Simulation methods

The evaluation was made by simulating the execution of five hand compiled toy benchmarks in the processors (see Figure 9).

PROG	NOTES
block	A program that moves a block of words to other location in memory
fib	A program that calculates the fibonacci value
sieve	A program that calculates prime numbers using the sieve of Eratosthenes
sort	A program that sorts a table of words using recursive quicksort algorithm
trans	A program that returns the transpose of a matrix of words

Figure 9 The benchmark programs.

Toy benchmarks were chosen, because we wanted to use hand compilation to make sure we were measuring the performance of architectures not compilers. The benchmarks were run assuming ideal memory hierarchy, i.e., there are no cache misses or the cycle time of main memory is smaller than the cycle time of the processor.

Results

We measured the execution time in clock cycles, program code size in bytes and utilization of functional units. The results of simulations are shown in Figure 10, Figure 11 and Figure 12 respectively.

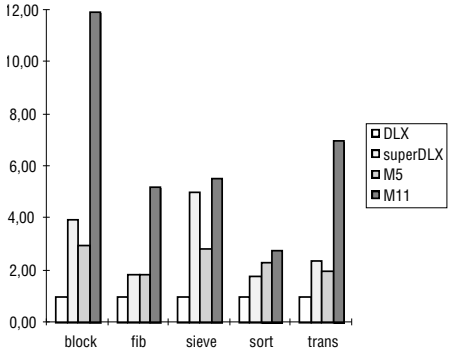

Figure 10 The relative performance of processors.

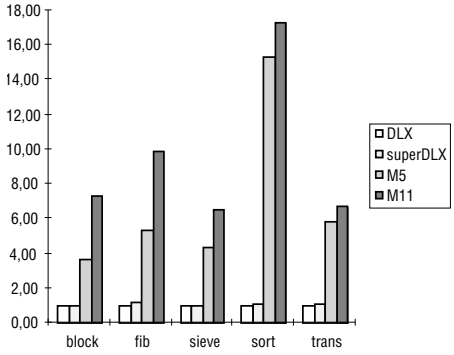

Figure 11 The relative size of programs of processors.

According to our execution time simulations M5 ran benchmarks 235% faster than DLX and only 14.9% slower than superDLX. The speedups ranged from 1.9 to 3.0. M11 ran benchmarks 581% faster than DLX and 211% faster than superDLX. The speedups ranged from 2.8 to 12.

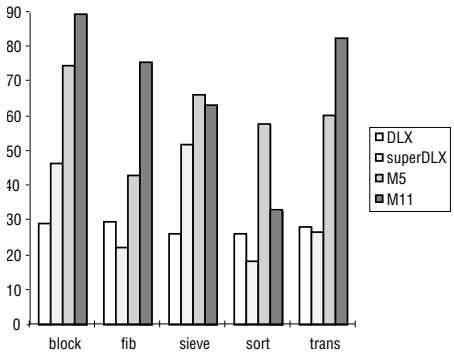

Figure 12 The utilization of functional units of processors in percents.

The size measurements of the simulated benchmark programs reveal that M5 programs were 4.9 times longer than DLX programs and M11 programs were 6.5 times longer than DLX programs. These average code expansion factors were calculated using code expansion of only *block*, *sieve* and *trans* benchmarks, because certain space consuming hand optimization methods were used to compile *fib* and *sort* benchmarks.

The average utilization of functional units in M5 was 60.5% that is 2.2 times better than in DLX. The average utilization of functional units in M11 was 69.0% which is 2.1 times better than in superDLX. Register units were not taken into account while calculating these numbers, because they just save data instead of actually changing it.

Discussion

M5, a five unit implementation of MPA, performed over two times better than DLX, a basic pipelined RISC processor, that has the same number of functional units than M5. The better performance comes from better exploitation of available instruction level parallelism, which is due to over two times better utilization of functional units and absence of pipeline hazards.

M11, an eleven unit implementation of MPA, performed almost six times better than DLX and over two times better than superDLX, a superscalar processor featuring out of order execution and branch prediction, and more functional units than M11. This is quite surprising, because in theory superscalar and VLIW processors should perform equally well. We believe that the difference in performance is an evidence that the static compile time scheduling in a VLIW processor performs better than the dynamic run-time scheduling in a superscalar processor.

Unfortunately the performance increase of MPA processors does not come for free: The code size of MPA processors expands by the factor of five to over six compared to the code size of DLX. This is due to the uncoded instruction format, absolute addressing and multiple word operands in the operation code.

The functional unit utilizations of these benchmarks in MPA processors are higher than one might expect. This is caused by certain hand optimization techniques that decrease execution time by adding subinstructions to suitable locations in the code.

CONCLUSIONS

We have described techniques to eliminate pipeline delays without losing the performance advantage of pipelining. We applied these techniques to a basic VLIW architecture. As a result we get Minimal Pipeline Architecture (MPA) using pipeline that has only two stages.

We evaluated two MPA processors and two reference processors by simulations. The results are quite favorable to MPA:

MPA processors perform over two times better than basic pipelined or superscalar processors with the same number of functional units. The better performance comes from better exploitation of available instruction level parallelism, which is due to over two times better utilization of functional units and absence of pipeline hazards. Unfortunately the code size expands by the factor of five to six.

The obtained speedups imply that MPA certainly provides an alternative to basic three or more stage pipelined architetures in terms of raw performance. The feasibility of MPA processors would be an interesting topic for further investigation.

REFERENCES

[Chang91]
P. Chang, S. Mahlke, W. Chen, N. Warter and W. Hwu, IMPACT: An architectural Framework for Multiple-Instruction-Issue Processors, Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991, Toronto, 266-275.

[Fisher81]
J. Fisher, Trace Scheduling: A technique for global microcode compaction, IEEE Transactions on Computers C-30, (1981) 478-490.

[Fisher83]
J. Fisher, Very Long Instruction Word Architectures and ELI-512, Proceedings of the 10th Annual International Symposium on Computer Architecture, June 1983, 140-150.

[Flynn95]
M. Flynn, Computer Architecture—Pipelined and Parallel Processor Design, Jones and Bartlett Publishers, 1995.

[Forsell94]
M. Forsell, Design and Analysis of Some Chip-level Parallel Architectures, Licentiate thesis, Department of Computer Science, University of Joensuu, Joensuu, 1994.

[Hennessy90]
J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[Johnson89]
W. M. Johnson, Super-Scalar Processor Design, Technical Report No. CSL-TR-89-383, Stanford University, 1989.

[Kogge81]
P. M. Kogge, The Architecture of Pipelined Computers, Hemisphere Publishing Corporation, USA, 1981.

[Moura93]
C. Moura, SuperDLX - A Generic Superscalar Simulator, ACAPS Technical Memo 64, McGill University, Montreal, Canada, 1993.

[Nicolau84]
A. Nicolau and J. Fisher, Measuring the parallelism available for very long instruction word architectures, IEEE Transactions on Computers C-33, 11 (1984), 968-976.

[Smith84]
A. Smith and J. Lee, Branch Prediction Strategies and Branch Target Buffer Design, Computer 17, 1 (January 1984), 6-22.

[Tomasulo67]
R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, IBM Journal of Research and Development 11, 1 (December 1967), 25-33.