社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＳＬＤＭ－112　(25)
2003／11／28

自己ハザードによるCISCパイプラインプロセッサのメモリアクセスステージ低減手法

渡辺 貴宏　宮崎 範子　伊藤 和人

埼玉大学工学部電気電子システム工学科
〒338-8570 埼玉県さいたま市桜区下大久保255
E-mail: {watanabe, miyazaki, kazuhito}@elc.ees.saitama-u.ac.jp

あらまし　CISC型プロセッサでは1つの命令が多数のマイクロ操作を実行できるので，プログラムコード長を低減し，プログラムメモリ容量を節約できる長所がある．その一方で，多数のマイクロ操作制御に多数の命令実行ステージが必要であり，パイプライン化した際にパイプラインレジスタ等に要するコストが増大する傾向がある．本論文では，自己ハザードによってCISCパイプラインプロセッサのメモリアクセスステージを低減することでパイプライン段数を削減し，プロセッサのコストを削減する手法を提案する．

キーワード　パイプラインプロセッサ，CISC，メモリアクセス，自己ハザード

Reducing Memory Access Stage of Pipelined CISC Processor by Self-Hazard

Takahiro Watanabe　Noriko Miyazaki　Kazuhito Ito

Department of Electrical and Electronic Systems, Saitama University
Shimookubo 255, Sakura-ku, Saitama, Saitama 338-8570, Japan
E-mail: {watanabe, miyazaki, kazuhito}@elc.ees.saitama-u.ac.jp

Abstract　One of the advantages of CISC type processors is code efficiency since each instruction can contain many micro operations. On the other hand, the relatively large number of instruction execution stages of CISC processor may lead to higher cost if the processor is pipelined. In this paper a method to reduce the number of pipeline stages for CISC processors is presented where stages dedicated to memory access are eliminated by self-hazard technique.

Keyword　Pipeline Processor, CISC, Memory Access, Self-Hazard

1. Introduction

With the advances in EDA tools for designing and synthesizing processors and in generating technique for a custom processor [1-3] and for software development system for the processor [1,4,5], it is becoming practical to design a processor which is specific to a particular application.

Code efficiency of a processor is defined as the number of program memory bits necessary to describe a particular processing. For a processor of high code efficiency, the necessary number of memory bits for programs can be small. One of the advantages of CISC (complex instruction set computer) type processors is code efficiency since each instruction can contain many micro operations. In order to implement a same task, a CISC processor may need less number of instructions than a RISC (reduced instruction set computer) processor [6,7]. The chip area for program memory is proportional to the number of instructions, i.e. static code size. Therefore, using a CISC processor can lead to an implementation whose total cost of a processor and necessary memory is cheaper than using a RISC processor.

While program memory size is minimized, the relatively large number of micro operations in each instruction of a CISC processor may result in more hardware cost if the processor is designed so that instructions are executed in pipelined manner.

Pipelining is one of the techniques to increase instruction execution efficiency of processors [6]. In a pipelined processor, execution of instruction is divided into some stages (pipeline stages) and different stages of more than one instruction are executed in parallel. In order to transfer control signals and intermediate operational data from one pipeline stage to another, pipeline registers are inserted between consecutive pipeline stages.

In CISC type processor, the necessary number of pipeline stages, in other words, the pipeline length of the instruction execution may vary from instruction to instruction. The processor must have the pipeline of the length equal to the longest instruction execution among all the instructions. The length of the CISC pipeline tends to become large since each instruction must control many micro operations. Such a long pipeline may result in higher product cost because a large number of pipeline registers must be used to implement pipelined processor and the cost for registers has large impact on the total cost of processor. Hence it is important to minimize pipeline length without sacrificing code efficiency.

Memory accessing is one of the reasons to increase the number of pipeline stages of CISC processors. Memory access usually requires many micro operation steps, such as setting memory address, wait for the data to become available, and transferring the data to some register. These micro operation steps must be done in sequence and a pipeline stage must be assigned to each step to achieve it. Therefore, reducing pipeline stages assigned to memory access can result in shorter pipeline if an instruction containing memory access is the bottle neck of minimizing the pipeline length.

In this paper a method to reduce the number of pipeline stages for CISC processors is presented where stages dedicated to memory accesses are eliminated by self-hazard technique. Note that the technique described in this paper is not to speed up the program execution but to reduce the processor cost without degrading program execution speed.

2. Processor Model

2.1. Memory Access

The memory address register (MAR) is a special purpose register to store memory address from which the memory content is read or to which data is written (Fig. 1). To access the memory, the memory address must be stored in MAR first. The exception is the memory read access for instruction fetch. Only in this case the content of program counter (PC) is directly supplied to the memory as a memory address. To realize this, a 2-to-1 multiplexor (MUX) is used to select the actual memory address from PC and MAR. The memory data register (MDR) is another special purpose register for memory data. When read memory, the memory content addressed by either PC or MAR is stored in MDR. When write memory, the data to be written to memory is first stored in MDR and then written to the memory address specified by MAR.

The memory access time (including the delay for the multiplexor for memory address) is assumed to be less than a clock cycle.

2.2. Data-path

The data-path of the processor is illustrated in Fig. 2. The processor has an arithmetic-logic unit (ALU). Its output is connected to an accumulator (AC). Thus operation result of the ALU is always stored in AC. When this data must be stored for more than one clock cycles, it should be transferred to a general purpose register. The left and right inputs to the ALU are selected (e.g. by multiplexors) from any of the special purpose registers MDR, MAR, PC, AC, the general purpose registers, and some special constants such as 0 and 1. These inputs are stored in special registers L_IN and R_IN for ALU input so that ALU operation fully utilizes a clock period.

Since PC is usually incremented each time an instruction is fetched, PC has its own circuitry to increment its value rather than using the ALU for this job.

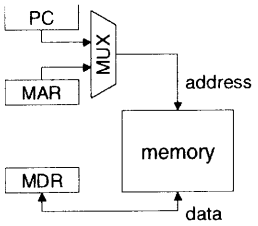

Fig. 1 Registers for memory access.

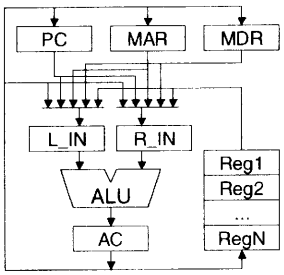

Fig. 2 Data-path of processor model.

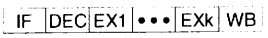

Fig. 3 Pipeline stages.

2.3. Pipeline Stages

The execution of an instruction is divided into several pipeline stages as shown in Fig. 3. The first stage is for the instruction fetch (IF). The memory read access is performed with PC selected as the memory address source. The data read from memory is stored in MDR at the end of IF stage. The second stage is for decoding the fetched instruction (DEC). The data-path control signals for the next stage are decoded by using the content of MDR and stored in pipeline registers at the end of DEC stage. In this stage, the sources of ALU operation are decoded and the requested data are transferred into ALU input registered. At the same time the content of MDR is transferred to a special instruction register (IR) to be used later to decode control signals for later stages. IF and DEC stages are common to all instructions. Following the DEC stage are execution stages denoted as EX1, EX2, and so on. In Fig. 3, the final execution stage is marked as EXk. The operations in each stage depend on the instruction. Furthermore, the number of execution stages also depends on the instruction in the case of CISC processors. The write back (WB) stage is added as the last pipeline stage if the result of the last ALU operation, which is stored in AC, must be stored in a general purpose register or any of the special purpose registers.

2.4. Pipelined Control Structure

To execute an instruction, control signals for the data-path are obtained by decoding the instruction word and supplied

to the appropriate data-path components in the appropriate instruction execution stage. In pipelined processors, different stages of several instructions are executed simultaneously. Hence more than one instruction must be decoded for different pipeline stages. To achieve this, instruction register IR is not a single register but consists of copies of registers IR1, IR2, and so on, configured as a shift register. Fig. 4 illustrates the pipeline for control signals. The registers IR1, IR2, ..., IRk are connected so that these registers form a shift register. In DEC stage, the content of MDR is fed into the decode circuit DEC1 which generates control signals for EX1 stage. The control signals are stored in control register CR1 and supplied to data-path components. The content of MDR is also transferred to IR1. At the same time, the content of IR1, which is the instruction code of the previous instruction, is fed into the decode circuit DEC2 which generates control signals for EX2 stage. EXk is the final execution stage. The content of IRk is decoded by the circuit DECWB to generate control signals for WB stage, if WB stage follows EXk stage.

The register transfer language (RTL) description of pipelined control structure is shown in Fig. 5. The micro operation 'mem_address=PC' in IF stage means the multiplexor is controlled so that PC is selected as the memory address source in IF stage. PC is incremented in IF stage and data output from the memory is transferred to MDR at the end of IF stage.

3. Pipeline Hazards

In this section, pipeline hazards [6] are briefly reviewed.

In pipelined processors, execution of instructions is distorted by several reasons. This is called pipeline hazard [6]. When a pipeline hazard occurs, some pipeline stages cannot perform any operation and just wait for the reason of the hazard is resolved.

3.1. Structural Hazard

This type of hazard is caused when two or more instructions request an identical hardware resource at the same time. The possible sources of structural hazard are ALU for operations and memory for reading and writing data and fetching instruction. The earliest instruction is given the priority to use the hardware resource. Other instructions wait until this hardware resource becomes available.

The pipeline hazard is divided into three categories.

1. structural hazard
2. data hazard
3. branch hazard

3.2. Data Hazard

This type of hazard is caused when there exist data dependencies between instructions. For example, suppose the following two instructions are executed in sequence as shown in Fig. 6(a).

I1: ADD Reg1, Reg2, Reg3 (Reg1 ← Reg2+Reg3)
I2: SUB Reg5, Reg1, Reg4 (Reg5 ← Reg1-Reg4)

The instruction I1 adds the data of Reg2 and Reg3, and stores the sum into Reg1. The instruction I2 subtracts the data of Reg4 from Reg1 and stores the result into Reg5. When these instructions are executed correctly, the data of Reg5 is (Reg2)+(Reg3)-(Reg4). For the instruction I2 to work correctly, the data of Reg1 must be (Reg2)+(Reg3) by the moment Reg1 is referred for subtraction. If the addition of (Reg2)+(Reg3) and writing back the result to Reg1 are

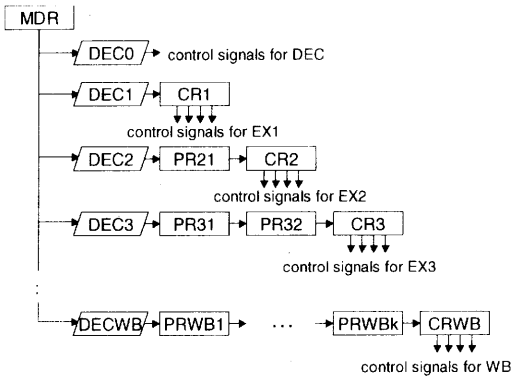

Fig. 4 Pipeline for control signals.

Stage	Micro operations
IF:	mem_address=PC, PC ← INC(PC), MDR ← mem_data
DEC:	CR1 ← DEC1(MDR), PR21 ← DEC2(MDR), PR31 ← DEC3(MDR), ...
EX1:	CR2 ← PR21, PR32 ← PR31, PR42 ← PR41, ...
EX2:	CR3 ← PR32, PR43 ← PR42, PR53 ← PR52, ...
	⋮
EXk:	CRWB ← PRWBk
WB:	

Fig. 5 RTL description for pipeline control.

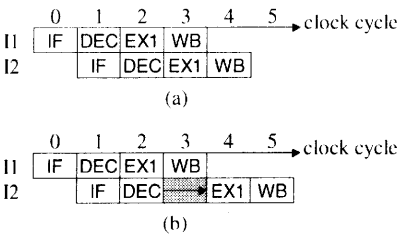

Fig. 6 Data hazard. (a) The situation where data dependency is not satisfied. (b) Data hazard is caused to satisfy the data dependency.

not in time for the subtraction, the execution of this subtraction must be postponed as illustrated in Fig. 6(b) until Reg1 gets the correct data. This is called *read after write* (RAW) hazard. RAW hazard occurs when an instruction wants to read data before the data is written by the previous instruction.

Write after read (WAR) hazard is caused in the situation where an instruction tries to write data before the data is read by the previous instruction. In this case the instruction to write data must wait until the date is read by the previous instruction so that the data is not overwritten before it is referred.

There exist two other situations: *read after read* (RAR) and *write after write* (WAW). However these will not cause data hazard.

3.3. Branch Hazard

This type of hazard is caused by executing a branch instruction. For example, assume that a branch instruction at address j is executed. A branching is to transfer the branch destination address to PC. Usually this is done in execution stage of the pipeline. Once PC is loaded the new address, instructions are fetched from this address. This means that the instructions at addresses j+1, j+2, and so on, which are already fetched, decoded, and possibly partly executed, must be discarded of flushed when the branch occurs. Hence the idle stages are introduced in the pipeline by the branch and it needs some time to refill the pipeline with the instructions of the branch destination.

4. Self-Hazard Technique for Memory Access Stage Reduction

4.1. Instruction with Memory Access

The number of pipeline stages of a processor is determined as the largest number of pipeline stages among the instructions of the processor. Therefore the pipeline for control signals shown in Fig. 4 must contain as many stages as to control the instruction which uses the most pipeline stages. The number of instruction registers, the circuits for decoding control signals, and pipeline registers for the decoded control signals depend on the number of pipeline stages. The longer the pipeline is, the more instruction registers, decode circuits, and pipeline registers are required.

Since the instructions of CISC processors control many micro operations for high code efficiency, the number of pipeline stages tends to be large. The cost of pipeline registers generally has large impact on the total cost of pipeline for control signals. Therefore, to minimize the cost of CISC processor, it is effective to reduce the cost of pipeline for control signals by reducing the number of pipeline stages. However, just reducing pipeline stages may sacrifice the code efficiency and result in increasing the cost of program memory. Consequently, it is important to reduce the number of pipeline stages without degrading the code efficiency.

Stage	Micro operations
IF:	mem_address=PC, PC ←INC(PC), MDR ← mem_data
DEC:	
EX1:	MAR ← Reg2
EX2:	mem_address=MAR, RdMem=1, MDR ← mem_data
EX3:	L_IN ← AC, R_IN ← MDR
EX4:	AC ← L_IN + R_IN
WB:	Reg1 ← AC

Fig. 7 Pipeline stages for instruction 'ADD Reg1, AC, (Reg2)'.

Stage	Micro operations
IF:	mem_address=PC, PC ←INC(PC), MDR ← mem_data
DEC:	
EX1:	MAR ← Reg2
EX2:	mem_address=MAR, RdMem=1, MDR ← mem_data, L_IN ← AC, R_IN ← MDR
EX3:	AC ← L_IN + R_IN
WB:	Reg1 ← AC

Fig. 8 Modified pipeline stages for instruction 'ADD Reg1, AC, (Reg2)'.

One of the differences of RISC and CISC instruction sets is that both memory access and arithmetic/logic operations are executed for some instructions in the case of CISC. For example, in CISC processors, it is possible to have a complex instruction which reads data from memory, adds the data with the content of a general purpose register, and then stores the result to memory. While including memory access in addition to arithmetic/logic operations within an instruction contributes to increase code efficiency, this makes the instruction pipeline longer since memory access generally requires many micro operations and hence many pipeline stages to control the memory access.

In this paper, we pay attention to pipeline stages for memory access and propose a technique called *self-hazard* to reduce the pipeline stages for memory access without sacrificing code efficiency.

4.2. Self-Hazard for Memory Read Access

Fig. 7 shows an example of the pipeline stages for the instruction 'ADD Reg1, AC, (Reg2)', which reads the memory data at the address specified by Reg2, adds the data with AC, and then stores the result to Reg1. Based on the instruction decode (Micro operations for instruction decode are omitted from the figure. For detail, refer Fig. 5.), the content of Reg2 is transferred to MAR in EX1 stage. In EX2 stage, MAR is used as the memory address and memory read control signal RdMem is enabled. Thus the memory is accessed for read in EX2 stage and the data output from the memory is loaded onto MDR at the end of EX2 stage. In EX3 stage the content of MDR, which is now the memory data at the address of Reg2, is transferred to the right input

ALU register R_IN. It is added into AC in Ex4 stage. Then the addition result is written back to Reg1 in stage WB.

The memory access for read and the data transfer are separated into two stages of EX2 and EX3 to ensure that the memory data is read from memory and stored in MDR, and the ALU uses the data read from memory. This can be interpreted as the ALU right input register waits for its operation data to be available by the memory access for read. In other words, there is a data dependency on MDR between the memory access for read and the transfer to R_IN. This is the read after write (RAW) situation for MDR. It is possible to make the transfer to R_IN (read for MDR) wait until MDR being loaded with the data (write for MDR) by using RAW hazard mechanism. Usually RAW hazard is caused for data dependency between two consecutive instructions. However in the case of the data dependency on MDR between memory access for read and the transfer to R_IN, the data dependency exists within an instruction. Hence the term 'self-hazard' is introduced.

By using self-hazard for the instruction 'ADD Reg1, AC, (Reg2)', pipeline stages for its execution is modified as shown in Fig. 8. In EX2 stage, the first three micro operations are executed similarly as the original pipeline. The fourth micro operation, R_IN ← MDR, is postponed since self-hazard is caused for RAW condition on MDR. The self-hazard makes EX2 stage repeated in the next clock cycle. In the second clock cycle of EX2 stage, the micro operation R_IN ← MDR is executed since MDR is already loaded with the memory data. The addition is performed by using R_IN and its result is written back to Reg1 in the following stage WB. The EX3 stage in the original pipeline is merged into EX2 stage as the second clock cycle.

4.3. Self-Hazard for Memory Write Access

Fig. 9 shows an example of the pipeline stages for the instruction 'ADD (Reg2), AC, Reg1', which adds the data of Reg1 with AC and then stores the result to the memory at the address specified by Reg2. Based on the instruction decode (micro operations for instruction decode are again omitted from the figure), the addition is executed in EX1 stage. In EX2 stage, the content of AC is transferred to MDR and the content of Reg2 is transferred to MAR. MAR is used as the memory address and memory write control signal WrMem is enabled in EX3 stage to perform memory write operation. WB stage does not exist since no data is needed to be written to registers.

Memory access for write is executed by using two stages of EX2 and EX3. That is, MAR and MDR are set in EX2 stage and then memory write control signal WrMem is asserted in EX3 stage. Although it is not explicitly indicated in the micro operations, MDR is referred for read when the memory is accessed for write. Therefore, there exists read after write (RAW) situation for MDR since it is loaded with the addition result and the data is then used in memory access for write. The memory access for write must wait until MDR is loaded with the correct data. This is achieved by RAW hazard for MDR.

Stage	Micro operations
IF:	mem_address=PC, PC ← INC(PC), MDR ← mem_data
DEC:	L_IN ← AC, R_IN ← Reg1
EX1:	AC ← L_IN+R_IN
EX2:	MAR ← Reg2, MDR ← AC
EX3:	mem_address=MAR, WrMem=1

Fig. 9 Pipeline stages for instruction 'ADD (Reg2), AC, Reg1'.

Stage	Micro operations
IF:	mem_address=PC, PC ← INC(PC), MDR ← mem_data
DEC:	L_IN ← AC, R_IN ← Reg1
EX1:	AC ← L_IN+R_IN
EX2:	MAR ← Reg2, MDR ← AC mem_address=MAR, WrMem=1

Fig. 10 Modified pipeline stages for instruction 'ADD (Reg2), AC, Reg1'.

Table 1 Specification of the processor

Data-path registers (8 bits)	PC MAR, MDR L_IN, R_IN AC Reg1 ~ Reg4

By using self-hazard for the instruction 'ADD (Reg2), AC, Reg1', pipeline stages for its execution is modified as shown in Fig. 10. In EX2 stage, the micro operations MAR ← Reg2 and MDR ← AC are executed similarly as the original pipeline. The memory access for write (enabling WrMem signal) is postponed since self-hazard is caused for RAW condition on MDR. Similar to the self-hazard for memory read access in the previous section, EX2 stage is repeated in the next clock cycle by self-hazard. In the second clock cycle of EX2 stage the control signal WrMem is enabled to perform memory access for write since MDR is already loaded with the data to be written to memory. In this way, the EX3 stage in the original pipeline is merged into EX2 stage as the second clock cycle.

It must be noted that although the number of pipeline stages is reduced by self-hazard, code efficiency is not sacrificed since the micro operations executed in instructions are not altered. In addition, self-hazard does not change the necessary number of clock cycles to execute instructions. It neither increases or decreases clock cycles. It only reduced the number of pipeline stages of instructions with memory access.

5. Experimental Results

To show the effectiveness of the proposed self-hazard technique, an example base CISC processor was designed. Then self-hazard technique was applied to the processor and the performances of these two processors were compared.

The specification of the base CISC processors is summarized in Table 1. The bit length of all the registers used in data-path is assumed to be 8. The number of general purpose registers is four in the experiments.

The base processor implements 27 instructions, including data transfers among registers, arithmetic/logic operations on registers, conditional and non-conditional branches, and data transfer/operation with memory access. The length of the instruction pipeline is 7, consisting of the stages of IF, DEC, EX1, EX2, EX3, EX4, and WB. This is determined by the instruction 'ADD Reg1, AC, (Reg2)' (as shown in Fig. 7) which performs memory access for read in addition to arithmetic (add) operation.

The base processor was described in RTL level by using VHDL. The description was tested by using a VHDL simulator to confirm that the instruction pipeline works correctly for all the types of pipeline hazard. The processor circuit was synthesized by using SYNOPSYS Design Compiler with the target process of 0.5μm, 2-metal layer CMOS technology. CPU time for synthesis is less than 5 minutes on 750MHz SPARC workstation for each of the base processor and the processor with self-hazard technique. The synthesis result is summarized in Table 2. This table shows the length of the instruction pipeline in the first row. Shown in the following 6 rows are the number of gates for pipeline registers, common hazard detection circuit, self-hazard detection circuit, data-path registers, ALU, and the number of gates for decoders for control circuits. In this table the maximum operation clock frequency reported by the synthesis tool is also indicated in the bottom row. On the base processor, the critical path exists in hazard detection and control signal generation part.

By applying self-hazard technique to the base processor, the length of the instruction pipeline is reduced from 7 to 6 because the necessary number of pipeline stages of the longest instruction can be reduced. This results in reduction of the number of IR registers, decode circuits, and pipeline registers for control signals shown in Fig. 4. The effect of this reduction overcomes the additional cost of self-hazard detection. The processor with self-hazard was also synthesized and the result is summarized in Table 2. The number of gates for data-path registers and ALU is not changed. The number of gates for the pipeline for control signals shown in Fig. 4 is reduced. Hence the total number of gates is reduced to about 74% from the base processor.

On the processor with self-hazard, since the number of pipeline stages is reduced, hazard detection and control signal generation become simpler than the base processor and the critical path is reduced. Now the critical path exists in the ALU.

This result is obtained for the case that only one pipeline stage is reduced. If a more complex instruction with more than one memory access is included in the instruction set, the pipeline stage reduction by the proposed self-hazard technique can be applied for each memory access. Hence larger improvement is possible for a processor with such a complex instruction set.

Table 2 Comparison of base and proposed processors

	Base processor	Processor with self-hazard
Pipeline stages	7	6
Pipeline registers	1800	1080
Hazard detect	2047	1208
Self-hazard detect	-	121
Data-path registers	546	
ALU	620	
Decoders	601	601
Total [gates]	5614 (100%)	4176 (73.9%)
Clock Freq. [MHz]	68.2	77.8

6. Conclusions

In this paper, self-hazard technique was proposed to reduce the memory access stages in the instruction pipeline of pipelined processors. In CISC processors, instructions containing memory access may determine the required number of pipeline stages of the processor. In that case, by using self-hazard technique memory access stages can be reduced and it results in minimizing the number of pipeline stages and therefore the cost of the processor.

Support for forwarding and interrupt remains as future work.

References

[1] B. Shackleford, et al., "Satsuki: An Integrated Processor Synthesis and Compiler Generation System," IEICE Trans. Inf. & Syst., vol. E79-D, No. 10, pp. 1373-1381, 1996.

[2] M. Itoh, et al., "Processor Generation Method for Pipelined Processors in Consideration with Pipeline Hazards," IPSJ Journal, vol. 41, No. 4, pp. 851-862, 2000.

[3] T. Sasaki, et al. "Rapid Prototyping of Complex Instructions for Embedded Processors using PEAS-III," Proc. SASIMI 2000, pp. 61-66, 2000.

[4] N. Ishiura, T. Watanabe, and M. Yamaguchi, "A Code Generation Method for Datapath Oriented Application Specific Processor Design," Proc. SASIMI 2000, pp. 71-78, 2000.

[5] O. Wahlen, et al., "Instruction Scheduler Generation for Retargetable compilation," IEEE Design & Test, vol. 20, No. 1, pp. 34-41, 2003.

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

[7] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall Inc., 1991.