Advanced Processor Design Using Hardware Description Language AIDL

Takayuki Morimoto† Kazushi Saito† Hiroshi Nakamura*
Taisuke Boku† Kisaburo Nakazawa‡

†Institute of Information Sciences and Electronics, University of Tsukuba
*Research Center for Advanced Science and Technology, University of Tokyo
‡Department of Computer Science, University of Electro-Communications

Abstract— In order to design advanced processors in a short time, designers must simulate their designs and reflect the results to the designs at the very early stages. However, conventional hardware description languages (HDLs) do not have enough ability to describe designs easily and accurately at these stages. Then, we have proposed a new hardware description language AIDL.

In this paper, in order to evaluate the effectiveness of AIDL, we describe and compare three processors in AIDL and VHDL descriptions.

I. Introduction

As processors get larger and achieve higher performance, their designs become complicated. Recently, HDLs are usually used in processor designs, for example VHDL[1][2], Verilog-HDL[3], SFL[4], and so on. Using HDLs, designers can reuse what have ever been designed, verify and simulate their designs by using tools, and synthesize lower level designs.

Our purpose is shortening the required time for designing advanced processors which are best suited for design purpose. For this purpose, designers are required to select the best design at the very early stages. In order to satisfy this requirement, at the very early design stages, designers must design several implementations which are rough but completed, evaluate the effectiveness of the designs by simulations, and reflect the results into the designs. Here, the term of implementation represents how to control the executions of instructions.

At the very early design stage, only the basic architecture and its implementation should be considered and described. We call this level of design as "architecture and implementation level". At this level, designers have to consider only instruction set architecture and control of the instructions. Here, they do not have to consider detailed data path structures such as connections of functional units and detailed timing relations such as delays of circuits. Even in this early design stage, there are wide variety of architectures and implementations to be investigated.

Conventional HDLs have two major problems for designing advanced processors at the architecture and implementation level. First, designers must consider detailed data path structures such as connections between functional units. At the architecture and implementation level, this consideration is usually out of the scope. Second, designers should describe detailed timing relations even in synchronous behavior. For example, clock should be explicitly described, and designers have to pay attention to the timings of signal changes. However, such timing relations are too detailed at this design level. Because of these drawbacks, the initial design and the following modifications get difficult and take longer time. For example, in VHDL and Verilog-HDL, designers need to consider both of them. SFL also suffers from the first problem.

Then, we have proposed a new hardware description language AIDL[5] suited for design at the architecture and implementation level and design approach using AIDL. In AIDL, designers do not need to consider detailed data path structures. Moreover, actions are synchronized with an implicit clock in AIDL. Design flow using AIDL is shown in Fig.1. There is no logic synthesis system for AIDL. However, that disadvantage is solved by using a translator from AIDL into other HDLs which can be synthesized (Fig.1).

This paper is organized as follows: Section II describes the characteristics of AIDL. To evaluate the effectiveness of AIDL, we have designed three processors in AIDL and VHDL and compared them. This experiment is described in Section III. Conclusion and future works are mentioned in Section IV.

II. Hardware Description Language AIDL

To attain out objective, it is important that an HDL should satisfy two requirements:

- Designers can describe various instruction set architectures and hardware architecture simply and accurately.
- Designers can modify the description easily.

In order to satisfy these requirements, AIDL is designed so that it can describe simply and accurately timing relations such as sequentiality and concurrency and cause/effect relations between pipeline stages.

A. Timing Relation

AIDL has two characteristics for describing timing relations simply and accurately.

First, the concept of timing relations is based on Interval Temporal Logic[6]. AIDL is defined on discrete time sequences and its behavior is defined on time interval. Therefore, designers need not to consider explicitly

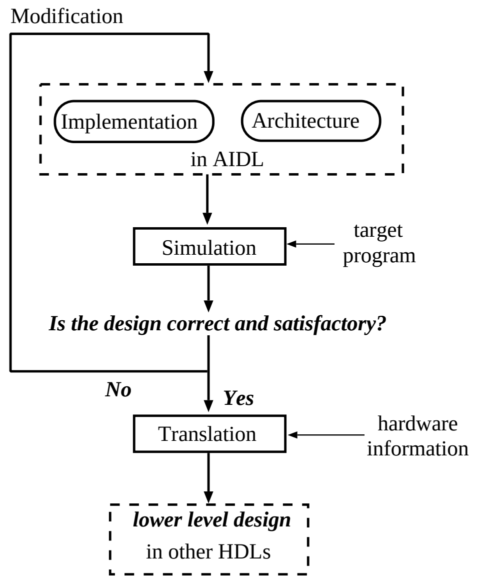


Fig. 1. Design flow in AIDL

detailed delay or timing relations in AIDL. Because we focus on advanced processor design at the very early stages, synchronous model is sufficient.

Second, sequentiality or concurrency of behavior is clearly defined in AIDL. Each behavior is described as a part of *stage* in AIDL. *Stage* usually correspond to stage of instruction pipeline at the architecture and implementation level. Sequentiality and concurrency between *stages* or within a *stage* are defined as follows. Fig.2 shows the syntax of *stage* definition. A *stage* is activated whenever its "*activating condition*" is satisfied unless the same *stage* is under execution. If there are other *stages* whose "*activating conditions*" are satisfied, those *stages* are also activated and executed in parallel. The operator of "//" expresses sequentiality. "A // B" represents that the "B" is executed after the execution of "A" has been completed. Therefore, "flag-set parts" and "data-transfer parts" in the same *stage* are executed sequentially.

A "flag-set part" is defined on a time interval of length δ. Because δ is a constant which is common throughout all the *stages*, the end of executed flag-set parts are synchronized even if they belong to different *stages*. A "data-transfer part" is defined on a time interval whose length is "*execution-time*". The "*execution-time*" should be a positive integer. Statements in a "flag-set part" or in a "data-transfer part" are executed in parallel.

A data assignment to a variable is expressed by "<-" operator. The statement of "A <- B" represents the value of "B" at the beginning of the interval is assigned into "A" at the ending time of the interval. An example is shown in Section II.C.

B. Cause/Effect Relation

AIDL has two kinds of variables, that is, flag variables and register variables. Flag variables are introduced to specify cause/effect relations between *stages*. Value of each flag variable is either TRUE or FALSE. Each flag variable has a value of priority. When different values are simultaneously assigned to a flag variable, the prior one is assigned to it. Intuitively, this variable represents a latch in control logic. By evaluating the values of flag variables in the activating conditions, cause/effect relations between *stages* are specified. An example is shown in Section II.C.

```
stage <stage name>(activating condition){
  <flag-set part> //
  block(execution-time){ data-transfer part } //
  <flag-set part> //
  block(execution-time){ data-transfer part } //
                  ⋮
}
```

Fig. 2. Syntax of *stage* definition

Register variable has a value of multiple bits. Intuitively, this variable represents a register in data path part. This variable can be assigned in only "data-transfer parts".

In AIDL, both variables are global and can be referred to or assigned from wherever in the descriptions. This nature is beneficial for controlling cause/effect relations between *stages* which behave independently.

C. Example

Here, the characteristics mentioned in Sections II.A and II.B are explained by using a simple example. In Fig.3, (A) is a part of instruction pipeline description in AIDL and (B) represents how (A) is executed. The first underlined part of Fig.3-(A) represents that "decode" *stage* is activated if the flag variable "decode_start" is "TRUE" and "datahazard" is not "TRUE". The second underlined part represents that "execute" *stage* is activated if "execute_start" is "TRUE". The register variable "counter" is a four-bit register which is used to arise pipeline stalls.

Suppose that "decode" *stage* is active on the time interval between clock **t** and (**t+1**) (illustrated as ♯1 in Fig.3-(B)) because "decode_start" is "TRUE" and "datahazard" is "FALSE" at clock **t**. Next, the flag-set part (line 3 in Fig.3-(A)) is executed in a time interval δ. The flag variable "decode_start" is assigned to "FALSE" at (**t**+δ). Then the data-transfer part between line 5 and line 10 in the block is executed on the time interval between clock (**t**+δ) and (**t+1**). The statement at line 5 in Fig.3-(A) represents the cause/effect relation between "decode" and "execute". By this statement, "execute_start" is set to "TRUE" at clock (**t+1**).

At clock (**t+1**), both "decode" and "execute" *stages* are activated. The part of `(counter<3> == 'b1)` (line 8) represents the condition that the third bit of "counter" is equal to "1". Because the least significant bit of "counter" is 1, the statement at line 9 is executed and "datahazard" is set to "TRUE" at clock (**t+2**).

At clock (**t+2**), "decode" *stage* is not activated because its activating condition is not satisfied. However, "execute" *stage* is activated. Then, "datahazard" is set to "FALSE" at clock (**t+3**) because the condition represented at line 15 is satisfied. Therefore, at the next clock (clock (**t+3**)), "decode" *stage* is activated again (♯2 in Fig.3-(B)).

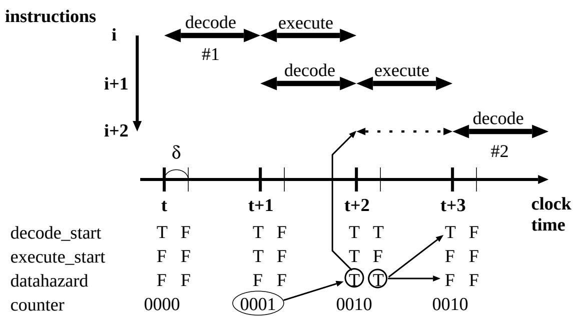
In this way, timing relations are described simply in AIDL. Cause/effect relations are also expressed simply and accurately by using flag variables.

III. Experiment

In order to evaluate the effectiveness of AIDL and our design flow, we have designed three processors in both

```
stage decode(decode_start == TRUE
             && datahazard != TRUE){
  decode_start <- FALSE;//
  block(1){
    execute_start <- TRUE;
    decode_start <- TRUE;
    counter<0:3> <- counter<0:3> + 'b0001;
    if (counter<3> == 'b1) then
      datahazard <- TRUE;
    endif;
}}
stage execute(execute_start == TRUE){
  execute_start <- FALSE;//
  block(1){
    if (datahazard == TRUE) then
      datahazard <- FALSE;
      decode_start <- TRUE;
    endif;
}}
```

(A): Instruction pipeline in AIDL


instructions
i
i+1
i+2
decode
execute
#1
decode
execute
decode
#2
δ
t
t+1
t+2
t+3
clock time
decode_start T F T F T T T F
execute_start F F T F T F F F
datahazard F F F F T T F F
counter 0000 0001 0010 0010

(B): Executing the description of (A)

Fig. 3. Example of *stage* control

AIDL and VHDL. Here, we just designed at the architecture and implementation level, because, in our design approach using AIDL, a description at the lower level such as RTL is generated by using a translator which accepts AIDL description and information on data path structure (Fig.1). Two designers design the target processors in AIDL or VHDL separately. They have some experiences to design other processors by each language. We compared the required time for the designs with each other. Here, design time includes the time for describing, debugging, and simulating the designs at the architecture and implementation level. We have already developed a prototype of translator which generates VHDL description. The translator can translate a simple AIDL description as shown in Fig.3 into VHDL in less than one second. However, since the routine for taking the input hardware information into account is not yet completed, the output of the translator is a description at the architecture and implementation level currently. Therefore, we do not compare two languages on synthesis level.

A. Design Target

Three processors are called "basic pipeline", "data forwarding", and "out-of-order completion". Instruction set architecture of these processors is based on PA-RISC 1.1[7]. Not all the instructions but only 23 instructions are implemented in these processors. The implemented instructions are listed in Table I.

All the processors have data cache memory (capacity : 1 KB, line size : 32 B, writeback, write allocate). As for instruction, all the processors do not have instruction cache. It always takes one cycle to fetch an instruction.

The simulator of AIDL has been already developed. By using the simulator, we confirm that each description is grammatically correct and satisfies the given specification.

TABLE I
INSTRUCTIONS

Type of data	Type of operation	Instructions
integer	load/store	LDW, LDWX, LDWS, STW, STWS
	arithmetic	ADDIL, SH2ADD, OR, LDIL, LDO
	branch	COMBF, COMBT, COMIBF, ADDIBT, ADDIBF
	others	HALT
floating-point	load/store	FLDDS, FLDDX, FSTDS
	arithmetic	FADD, FCPY, FDIV, FMPY

A.1 Basic Pipeline

Fig.4 shows structures of instruction pipeline. When a read after write (RAW) hazard occurs, executions of the following instructions are stalled until the dependency is resolved. Branch instructions have delayed branch feature as is in PA-RISC 1.1 architecture. The number of delayed slot is one.

A.2 Data Forwarding

"Data forwarding" is a processor where a data forwarding mechanism is added into "basic pipeline". In this improvement, the structure of instruction pipeline is not changed.

A.3 Out-of-order Completion

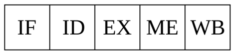
In order to improve performance, we modify instruction pipeline of "data forwarding" processor. Writeback stage is changed into four stages, "WB", "FW1", "FW2", or "FW3", depending on the instruction type. Memory stages of "FMPY" and "FDIV" are eliminated. Fig.5 shows the modified structures of instruction pipeline. This processor realizes out-of-order completion and concurrent writes into registers at different writeback stages by these improvements.

B. Result and Comparison

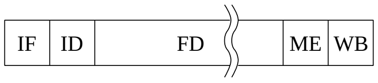
Table II shows the time for designing processors in AIDL and VHDL. In this experiment, we focus on the ratio of the design time in AIDL and in VHDL. Note that real processors have more instructions and more complex mechanisms such as trap handler, TLB, and so on. Then the difference of the absolute design time gets larger.

In Table II, the difference in "basic pipeline" between AIDL and VHDL is quite large. It takes about 4.6 times to design using VHDL as long as using AIDL. There are two reasons for this result. One reason is that since assignments in process shown in Fig.6 have a sequential ordering in VHDL, the designer has to consider the ordering of assignments very carefully. The other reason is that VHDL requires the designer to be very careful of how to control pipeline stages of multiple cycles. The consideration of detailed timings, such as a delay for an assignment under multiple execution cycles, is required.

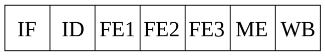
The required time for improving "basic pipeline" to "data forwarding" using VHDL is about 1.3 times as long as that using AIDL. In improving "data forwarding" to



(A): Instruction pipeline (except for "FMPY" and "FDIV")



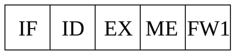
(B): Instruction pipeline of "FDIV"



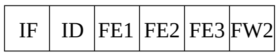
(C): Instruction pipeline of "FMPY"

- IF : instruction fetch
- ID : instruction decode, check of a data dependency, condition check and calculation of a target address for "COMBF", "COMBT", and "COMIBF", read of source registers
- EX : arithmetic calculation except for floating-point multiply and divide, address calculation for a load/store instruction, condition check and calculation of a target address for "ADDIBF" and "ADDIBT"
- FE1, FE2, FE3 : floating-point multiply
- FD : floating-point divide
- ME : memory operand access
- WB : write a result into general purpose or floating-point register files

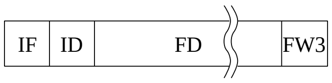
Fig. 4. Structures of instruction pipeline



(A): Instruction pipeline of floating-point load, "FCPY", and "FADD"



(B): Instruction pipeline of "FMPY"



(C): Instruction pipeline of "FDIV"

Fig. 5. Modified structures of instruction pipeline

"out-of-order completion", the required time using VHDL is about 1.7 times as long as that using AIDL. The ratio of time for improving "data forwarding" to "out-of-order completion" is greater than that for improving "basic pipeline" to "data forwarding". There are two reasons for this difference. One reason is that modifications of timings or controls of pipeline are required in only the latter improvement from "data forwarding" to "out-of-order completion". The other reason is that it is more difficult to modify timings or controls of pipeline using VHDL than that using AIDL. The total time of designing "out-of-order completion" by using VHDL (156 hours) is about 2.4 times as long as by using AIDL (66 hours).

We investigate only two improvements in our experiment. However, in real processor design, designers are required to implement more instructions, trap handler, TLB, and so on. In such cases, the difference between the required time in AIDL and in VHDL gets larger. More difficult implementations such as superscalar, VLIW, or a branch prediction will make the difference much larger. Then, the difference of required time may become several months. Therefore, AIDL is more suitable to explore the design which is best suited for the design purpose from possible instruction set architectures and hardware architectures than VHDL.

TABLE II
Result of design time

Design	Time [hours]	
	VHDL	AIDL
basic pipeline	83	18
basic pipeline → data forwarding	24	19
data forwarding → out-of-order completion	49	29

```
pipeline : process(clk)
begin
  if (clk'event and clk = 1) then
    if (activating condition of stage) then
      ----- stage actions -----
    end if;
    if (activating condition of stage) then
      ----- stage actions -----
    end if;
        ⋮
  end if;
end process pipeline;
```

Fig. 6. Pipelined behavior in VHDL

IV. Conclusion and Future Works

We described three processors both in AIDL and VHDL and measured the time required for description. It takes about 1.3 to 4.6 times in VHDL as long as in AIDL. The main reason is that VHDL is less flexible for describing timing relations. In real designs, designers have to investigate much more complex designs than in our experiment. Therefore, AIDL is helpful for designing advanced processors.

The simulator and prototype of translator has been already developed. We have successfully evaluated the description and estimated its performance. The translator can translate a simple AIDL description into VHDL. Using the translator, designs in AIDL can be followed by lower level design. Therefore, the entire design time is reduced if AIDL is utilized as the architecture and implementation level hardware description language.

As a future work, we will describe more complicated processors to evaluate the effectiveness of AIDL. Another future work is improving the translator so that it can generate RTL descriptions of high quality.

References

[1] R.Lipsett, C.Schaefer and C.Ussery, *VHDL: Hardware Description and Design,* Kluwer Academic Publishers, 1989.

[2] H.Juan, N.Holmes, S.Bakshi, D.Gajski, "Top-Down Modeling of RISC Processor in VHDL," *Technical Report 92-96, Dept. of Information and Computer Science, University of California, Irvine*, 1992.

[3] E.Sternheim, R.Singh, and Y.Trivedi, *Digital Design with Verilog HDL,* Automata Publishing Company, 1990.

[4] Y.Nakamura, "An Integrated Logic Design Environment Based on Behavioral Description," *IEEE Trans. on CAD*, CAD-6, No.3, pp322-336, 1987.

[5] T.Morimoto, K.Yamazaki, H.Nakamura, T.Boku, K.Nakazawa, "Superscalar Processor Design with Hardware Description Language AIDL," *Proc. of APCHDL'94*, pp51-58, 1994 .

[6] B.Moszkowski, "A Temporal Logic for Multi-Level Reasoning about Hardware," *Proc. of CHDL'83*, 1983.

[7] Hewlett Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual (Third Edition),* 1994.