Pipelined Processor Synthesis from Micro-operation Level Specification

Makiko Itoh

January, 2001

①

Pipelined Processor Synthesis from Micro-operation Level Specification

Doctoral Dissertation

by

Makiko Itoh
Department of Informatics and Mathematical Science
Graduate School of Engineering Science
Osaka University

Contents

Abstract

In an embedded system area, application specific instruction set processors (ASIPs) provide better performance, lower power consumption and a smaller chip area than general purpose processors. However, the design time of ASIPs becomes longer with the growth of the design scale. Higher abstraction level processor design method is required more than a traditional register transfer level (RTL) processor design method. The processor designs at RTL require a long design time because the designer has to design datapath and controller structures while considering the assignment of registers, functional units, interconnects among them, and the organization of the finite state machine of the controller. Designing processor organization at RTL from instruction set architecture level processor specification is an error-prone and time-consuming task. In addition, the modification of the processor specification requires a long time for re-design of datapath and controller at RT level. Therefore, comparing with several design candidates for specific application in a short design time is difficult.

In this thesis, micro-operation level pipelined processor specification and a processor synthesis method from a micro-operation level processor description are proposed for the improvement of design productivity of the ASIP development. The higher abstraction level than RTL contributes to the easiness of design and design modification for ASIP design. The ease of specification and modification of processor architecture enables architectural exploration of a large design space in a short design time. The designer only specified clock based instruction behavior in micro-operation level specifications. Datapath and controller of the processor are synthesized from the behavioral description of instructions.

The design space of micro-operation level processor specification is large enough for practical straightforward pipelined processors. Exploration of larger design space enables the designer to select a more suitable architecture for the target application. The target architecture of micro-operation level processor specification includes the following features: user-defined pipeline organization in terms of the number of pipeline stages, the number of delayed branch slots and the role of each pipeline stage; clock based behavioral representation of instructions and interrupts; utilization of parameterized hardware modules; and user-defined instruction format, processor interface ports and external interrupt conditions.

Processor synthesis from the micro-operation level processor specification includes datapath synthesis and controller synthesis. The synthesis of datapath and controller allows the designer to concentrate on instruction set design and evaluate various architecture candidates in a short time. In datapath synthesis, data flow graph generation from micro-operation description, signal conflicts resolution and insertion of pipeline registers are performed. In controller synthesis, instruction decoder, pipeline control logic such as pipeline stall and pipeline flush, and external interrupt control are synthesized.

From experimental results, the effectiveness and feasibility of the proposed processor synthesis method were evaluated. Examples in experiments are a MIPS R3000 compatible processor, DLX, PEAS-I core, a simple RISC controller, and a customized MIPS R3000 processor for DSP application. The amounts of processor design time and design modification time were drastically reduced compared with that of conventional RT level manual design. Processor synthesis time was about two minutes for the processor, which has 52 instructions. The design space of practical processors was explored at an architecture level in a short design time. In the design quality of synthesized processors and manually design processors, the clock frequencies are almost the same. The area of synthesized processors is about 20% larger than that of manually designed processors. Though the area is inferior to manual design, the advantage of effective design space exploration has an impact on the total design quality. The effectiveness of the micro-operation level processor specification and processor synthesis for architectural design space exploration is confirmed.

The proposed processor synthesis method enables the designer to explore a large design space at an architectural level. By the architectural exploration of a large design space, design productivity for application specific processors is drastically improved.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Prof. Masaharu Imai, Osaka University, for introducing me to this research area and guiding this work, for providing all the facilities to carry it out, and for continuous support, help and encouragement.

I would also like to express my thanks to Prof. Ken-ichi Taniguchi and Prof. Teruo Higashino for helpful suggestions and comments in writing this thesis. I wish to express my thanks to the professors and staff of the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, for their guidance, especially the late Prof. Seishi Nishikawa, and the late Prof. Mamoru Fujii, and Prof. Masaru Sudo, Prof. Nobuki Tokura, Prof. Akihiro Hashimoto, Prof. Toru Kikuno, Prof. Hideo Miyahara, Prof. Toshinobu Kashiwabara, Prof. Toru Fujiwara, Prof. Katsuro Inoue, Prof. Kenichi Hagihara, Prof. Masayuki Murata, Prof. Toshimitsu Masuzawa, Prof. Tadahiro Kitahashi and Prof. Shinichi Tamura.

The author is extremely thankful to Prof. Yoshinori Takeuchi, Prof. Akira Kitajima from Osaka University, Prof. Jun Sato from Tsuruoka National College of Technology and Prof. Akichika Shiomi from Shizuoka University for their continuous support, help and encouragement, and many thanks to all members of the PEAS project for their kind assistance, especially, Dr. N. N. Binh, Mr. Yoshimichi Homma, Prof. Takumi Nakano, Prof. Tsutomu Kimura, Mr. Nobuyuki Hikichi from Software Research Associates, Inc., and to the members of the VLSI System Design Laboratory at Osaka University, especially, Ms. Akiko Fujii, Ms. Ranko Morimoto, Mr. Takafumi Morifuji, Mr. Norimasa Ohtsuki, Mr. Shigeaki Higaki, Mr. Shinsuke Kobayashi, Mr. Yoshiharu Watanabe, Mr. Tomohide Maeda and Mr. Naoki Morita.

The author would like to express her special thanks to Dr. Tokinori Kozawa, Mr. Michiaki Muraoka and Dr. Eiji Masuda from STARC (Semiconductor Technology Academic Research Center), Dr. Masao Hotta, Mr. Mitsuho Seki, Mr. Yoshio Takamine from Hitachi Limited, and Mr. Masanobu Mizuno from Matsushita Electric Industry Corporation.

This research was supported in part by STARC.

Chapter 1

Introduction

1.1 Background

With advancements in semiconductor technology, chip complexity, that is, the number of transistors on a silicon chip, is doubling every three years. In the near future, it is estimated that over 10 million transistor circuits will be realized on a silicon chip of only 1 cm^2. From the technology innovation, *System-on-a-Chip* (SoC) with compound functionalities integrated on a single chip tends to be widely used in electronic equipment [1]. Figure 1.1 shows a typical organization of SoC. SoC usually consists of a combination of the following components: processors such as CPU core, digital signal processor (DSP) and specific processors; ASICs such as signal processing hardware, control hardware and other specific hardware; memories like flash and DRAM; and analog circuits. LSI designs are moved from the individual design of microprocessors and application-specific integrated circuits (ASICs) to a whole system design on a chip.

In SoC design, design productivity is a key issue. The roadmap of SEMATECH [2] indicates the growing productivity gap between available transistors and those that can be designed in microprocessors. The growth of transistor density is 58% per year. On the other hand, the growth of design productivity is only 21% per year. In addition, the rapidly changing technological environment shrinks product life cycles and shortens time-to-market.

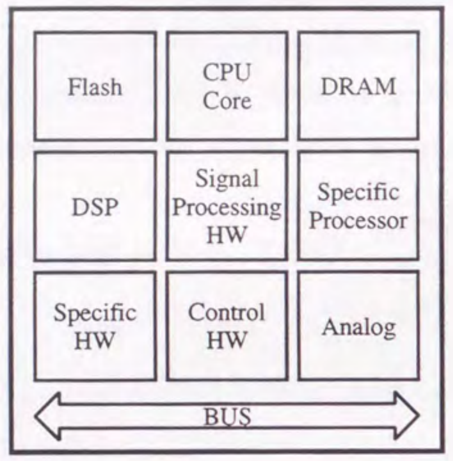


Figure 1.1: Typical Organization of System-On-a-Chip.

1.2 ASIP Development

Focusing on the design of application specific instruction set processors (ASIPs) that are integrated to the SoC, Hardware/Software co-design [3] environment with architectural design space exploration is considered to be key to design productivity improvement. For the ASIP design, it is important to explore suitable processor architecture for the target application. The HW/SW co-design environment enables the designer to design and evaluate the processor while considering target application and suggests the direction for design improvement. Using the HW/SW co-design environment, the designer is able to design and evaluate various architecture candidates at instruction set architecture level easily. As a consequence, the designer is able to choose the most suitable architecture for the target application in a short design time.

Figure 1.2 shows one HW/SW co-design framework for effective design space exploration. The designer specifies processor architecture with entry system at instruction set architecture level. Processor components such as registers, memory access units, functional units and so on are instantiated from a module library. The module library provides instances at various abstraction levels. A processor synthesizer generates a simulation model and a synthesizable model of the designed

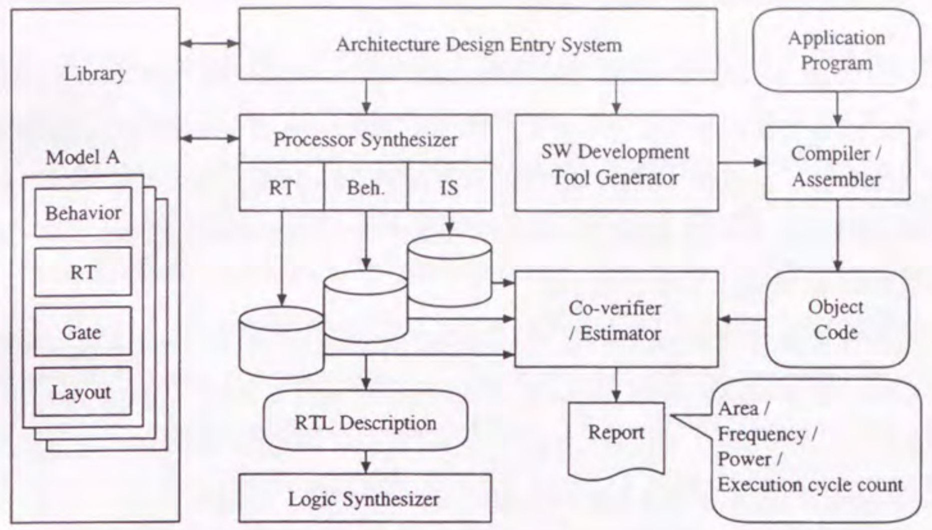


Figure 1.2: HW/SW Co-design Framework

processor. The processor synthesizer receives instances used in the processor from the database manager of the library. Software development tools such as a compiler and an assembler are also generated from the same processor description as processor synthesis. The synthesized instruction set simulation model and SW development tools enable co-verification and performance evaluation of the designed processor. A generated RT level processor model is used to estimate area, clock frequency and power consumption. Estimation and verification results suggest the direction for improvement of the processor design. Using a HW/SW co-design framework like this, exploration of large design space becomes possible because the turn-around time of the ASIP is drastically reduced.

The following techniques are required to implement a HW/SW co-design framework: instruction set level processor specification, processor synthesis and software development tool synthesis method from the same processor description, and fast estimation of designed processor. The processor synthesis method of the HW/SW co-design system often limits the design space of the system. In the recent research, several HW/SW co-design methods and processor synthesis methods are proposed, but their design space is very small in regard to pipeline organization.

1.3 Objective

The aim of this research is an investigation of a processor synthesis method for exploration of a large design space for ASIPs. Because processor synthesis methods usually limit the design space of the HW/SW co-design environment, processor synthesis methods should support various other architecture candidates of the ASIPs for architectural design exploration.

To explore a large design space, two requirements must be satisfied: a short turn-around time for evaluation of various candidates, and a large design space. Even if the design space is large enough, the design space cannot be fully explored in the restricted design time if the turn-around time for the design is too long. There is a tradeoff between the easiness of the processor specification and the design space. An appropriate abstraction level for processor specification should be considered.

1.4 Approach

Considering the tradeoff between the easiness of the processor specification and the design space of the specification language, micro-operation level processor specification for processor synthesis and a processor synthesis method for micro-operation level specification are proposed in this thesis.

Micro-operation level processor specification is based on a clock base behavioral description of instructions. With the abstraction level processor specification higher than the RT level, the design time and design modification time of the ASIPs are drastically reduced. Despite the easiness of the specification, the design space of micro-operation level processor specification enables the designer to specify practical straightforward pipelined processors. The designer can specify the pipeline organization, hardware module configuration and external interrupts. From these points, a micro-operation level is appropriate for a straightforward pipelined processor in terms of the easiness of the design and design space.

At a micro-operation level processor design, datapath structure and controller are synthesized from behavioral description of instructions and hardware module configuration. The designer is free from tedious, error-prone datapath and controller

design. Therefore the designer can design various ASIPs in a short design time.

The target processor architecture is straightforward pipelined architecture that includes basic functionality of embedded microprocessors [4, 5, 6, 7, 8, 9] such as multi-cycle operation, delayed branch and external interrupts. Micro-operation level processor specification includes: the number of pipeline stages and the number of delayed branch slots; utilization of parameterized hardware modules; user-defined instruction format, processor interface ports and external interrupt conditions; and clock-based behavioral representation of instructions and interrupts. Operations of each pipeline stage are specified by the designer with micro-operation description of instructions. The pipeline depth, role of each pipeline stage and hardware modules have an impact on clock frequency and area. The number of delayed branch slots affects code size and execution cycles. Therefore, flexibility in the processor architecture, such as the number of pipeline stages and delayed branch slot and the role of pipeline stages, and in the configuration of hardware modules, allows exploration of a large design space.

For the processor synthesis method from a micro-operation level processor specification, datapath and controller synthesis is required for user-defined pipeline organization in terms of the number of pipeline stages, the number of delayed branch slot and role of each pipeline stage. The controller synthesis includes pipeline control logic synthesis for pipeline hazards, interrupt controller synthesis, and instruction decoder synthesis. Structural hazards are caused by multi-cycle operations and resource conflicts from multiple stages. Hence generation of the hazard detection logic and pipeline interlock logic is required. To deal with the specified number of delayed branch slots, generation of branch control and pipeline flush control logic are also required.

In this thesis, to deal with user-defined pipeline organization, a flexible pipelined processor model is proposed. The model has flexibility regarding the number of pipeline stages and pipeline control logic. The model consists of datapath and controller of each pipeline stage, instruction decoder and external interrupt controller. The pipeline control mechanism using the model for pipeline interlock and pipeline flush is discussed. The processor model and pipeline control mechanism supports the processor synthesis from the micro-operation level processor specification.

Finally, the processor synthesis method based on the processor model is proposed. Processor synthesis from the micro-operation level processor specification includes datapath structure synthesis and controller synthesis. Synthesis of datapath and controller allows the designer to concentrate on instruction set design and evaluate various architecture candidates in a short time. In datapath synthesis, data flow graph generation from micro-operation description, signal conflicts resolution and insertion of pipeline registers are performed. In controller synthesis, instruction decoder, pipeline control logic such as pipeline stall and pipeline flush, and external interrupt control are synthesized.

1.5 Contribution of the Research

The effectiveness of architectural design space exploration using the proposed processor design method and synthesis method is known from the experimental results. Design and design modification time is reduced compared with the RT level processor design. Processor design space was successfully explored at an architecture level in a short design time. Processor synthesis time was about two minutes for the processor, which has 52 instructions.

In the design quality of synthesized processors and manually design processors, the clock frequencies are almost the same. The area of synthesized processors is about 20% larger than those of manually designed processors. Though the area is inferior to manual design, the advantage of effective design space exploration has an impact on the total design quality.

Consequently, the effectiveness of the micro-operation level processor specification and processor synthesis for architectural design space exploration is confirmed. By the architectural exploration of a large design space, design productivity for application specific processors is improved drastically.

1.6 Organization of the Thesis

This thesis is organized as follows.

In Chapter 2, existing HW/SW co-design environments, customizable processor cores and processor synthesis methods are reviewed. Problems of existing methods are discussed.

Chapter 3 describes micro-operation level processor specification and processor design environment PEAS-III. To determine parameters and user-definable parts of target processors, the characteristics of processor architecture are classified and evaluated in view of their impacts on performance and area on the processor.

In Chapter 4, the pipelined processor model for processor synthesis is illustrated. The model consists of datapath, an instruction decoder, a pipeline controller and an external interrupt controller. The conditions of pipeline interlock and pipeline flush are considered. The pipeline controller mechanism using these conditions is explained.

Chapter 5 is devoted to the processor synthesis method. The datapath and controller synthesis methods are described. The datapath synthesis includes data flow graph generation, signal conflicts resolution, and pipeline register insertion. The controller synthesis includes instruction decoder synthesis, pipeline control logic synthesis and interrupt controller synthesis.

In Chapter 6, the effectiveness of the method is evaluated through several experiments and architectural design space exploration is demonstrated. Examples in experiments are a MIPS R3000 compatible processor, DLX, a simple RISC controller, PEAS-I core, and a customized MIPS R3000 processor for DSP application. The amount of processor design time was drastically reduced compared with that of conventional RT level manual design in HDL. The processor design space was successfully explored at an architecture level in a short design time.

Chapter 7 presents the discussion of this thesis. The design space, design productivity and design quality of the proposed processor synthesis method are evaluated. The direction of further expansion of design space, reduction of turn-around time and improvement of the design quality are discussed.

The last chapter discusses the research results and concludes with future work.

Chapter 2

Related Work

In this chapter, related work for application specific instruction set processors (ASIPs) design is reviewed.

2.1 HW/SW co-design in early years

HW/SW co-design systems in early years are closely connected to the base processor of the system. The system adopts parameterized processor cores. However, datapath structure and pipeline organization are almost restricted. PEAS-I [10], Satsuki [11] and ARC [12] are classified to this approach. In these systems, the given configurable architecture is tuned to specific application by changing some architectural parameters such as bit width of hardware functional blocks, register file size, memory size, etc. The super set of instructions that can be executed on adopted processor architecture for the system is restricted. The system does not allow the user-defined extension instructions, so that the system cannot always fully satisfy the demand of diverse applications. User-defined instructions for extension are required to gain high performance.

2.2 Recent ASIP Development System

In the recent research, several ASIP development systems, which permit user-defined application specific instructions to be equipped with the target processor, have been proposed. These systems use their original processor description language to describe the target processor's instruction set and the hardware structure. From the processor description, a code generator, an instruction set simulator and HDL description of the target processor are generated.

ISPS [13] is a common processor description for code generation, simulation and processor synthesis in 1980's. While it incorporates a rich set of control mechanisms to describe parallelism and synchronization of processes, the synchronization mechanisms are inadequate to model pipeline operations and hazards for modern pipeline processors.

The other processor description based ASIP design systems for pipelined processors are classified into three types.

1. Adding several dedicated instructions to already designed processors. This approach includes FLEXWARE [14], Xtensa [15], Trimaran [16], CASTLE [17] and MetaCore [18].

2. Software development tool generation and performance evaluation system for its original processor specification language. This approach includes ISDL [19] based system, Expression [20] based system and LISA [21] based system.

3. RT level processor HDL description synthesis system for its original processor specification language. This approach includes MIMOLA [22] based system, nML [23] based system, AIDL [24] based system, and [25].

2.2.1 Prepared Processor based Approach

Processor descriptions in the first approach describe instruction set and portion of the datapath structure. In the approach, their pipeline organizations are fixed, so that modification of pipeline control does not allowed.

In the FLEXWARE [14], user-defined instructions can be described by the combination of generic instructions. The generic instructions are supported by instruction set simulator model of the FLEXWARE in VHDL. The designer can specify execution cycles for each instruction, but cannot specify pipeline organization. While FLEXWARE supports the retargetable code generator CodeSyn and the instruction set simulator Insulin, it doesn't support processor synthesis.

Xtensa [15] uses a customizable processor core. Xtensa permits some user-defined instructions using Tensilica Instruction Extension Language (TIE). While Xtensa supports both processor synthesis and software development tool generation, user-defined instructions must be executed in restricted cycles. The designer can only describe behavior of the instructions and the structure of "execution" stage, but he/she cannot change the number of pipeline stages and other pipeline stages.

Trimaran uses processor description language MDes [26], which describes both behavior/structure of the target processor. Trimaran allows only a restricted retargetability of the simulator to the HPL-PD [27] processor family.

CASTLE [17] specifies target processor's datapath in block diagram and generates VHDL description of a processor. The target architecture of CASTLE is VLIW. The feature of CASTLE includes: instantiation of VHDL descriptions for functional units from a module library, automatic input signal conflict resolution by selector insertion, and generation of VLIW control word for specified datapath. However, CASTLE assumes a basic VLIW architecture and cannot change pipeline stages.

MetaCore [18] is an application specific DSP development system. MetaCore prepares basic and extended instruction set, and additional user-defined instructions are permitted. Net-list level description of the datapath structure and behavioral description of instructions are described as a specification of a target processor. From these descriptions, software development tools and an HDL description of the target processor are synthesized. However, additional execution units are specified only for the "execution" stage. Additional execution units for other stages and changing the number of pipeline stages are not permitted.

2.2.2 Software Development Tool Generation Systems

Processor descriptions in the second approach describe an instruction set and structure of datapath. The designer can define pipeline structure of the target processor in terms of the number of pipeline stages and operations in each pipeline stage.

ISDL [19] [28] is one of such approach that describes an instruction set and datapath structure. In ISDL, constraints of pipeline execution are explicitly specified through illegal operation groupings. This is tedious for complex architectures like

DSPs that permit operation parallelism.

EXPRESSION [20] specifies an instruction set and datapath structure. A Pipeline description provides a mechanism to specify the order of pipeline stages. Accurate reservation tables can be generated from the description. While EXPRESSION supports cycle-accurate instruction set simulation by SIMPLESS [29], processor synthesis has not been supported.

LISA [21] [30] describes the datapath structure and operation-level description of the pipeline. LISA describes activation relationship among pipeline stages, pipeline stalls and pipeline flushes. However, LISA is used for retargetable simulators [31]. Processor synthesis has not been supported, either. Furthermore, description of pipeline control is tedious to design and to modify branch instructions and multi-cycle operations.

2.2.3 Processor Synthesis Systems

In the last approach, both behavior and datapath structure of the target processor are described. Synthesizable processor HDL descriptions are generated.

MIMOLA [22] describes behavior and structure of the target processor and generates RT level processor description. However, pipeline control is not supported since MIMOLA is micro-code based approach.

nML [23] describes behavior of instructions and datapath structure. From nML description, an instruction set simulator is generated [32]. nML is used by the retargetable code generation environment CHESS [33] to describe DSPs and ASIPs. Processor synthesis tool "Go" is also developed for nML processor description. However, nML does not directly support complex pipeline control such as pipeline interlock.

AIDL [24] specifies operations of each pipeline stage and timing relations and cause/effect relations among pipeline stages. Using AIDL, various kinds of processors can be represented including processors with out-of-order completion. However, the modification of the design is difficult for complicated architecture because the designer have to consider various kinds of dependency in the inter-instruction behavior.

Hamabe, *et al.* [25] proposed a description of clock based instruction behavior

and pipeline stage information includes the correspondence of hardware units to the stage that contains their operations. However, designers must describe instruction behaviors considering with pipeline registers. Furthermore, pipeline control is not directly described.

2.3 Problems of Existing Processor Descriptions

Existing processor development systems have some problems.

1. Existing processor development systems need both structural and behavior description of the target processor in order to generate the processor. Describing a datapath structure wastes design time. Furthermore, for design space exploration it is tedious to describe datapath structure in consideration of consistency between behavior/structural descriptions.

2. Most systems do not support specification of pipeline organization. The pipeline model of such languages is restricted. The designer cannot change the number of pipeline stages and role of each stage. Several systems support pipeline control synthesis, but explicit definition of the pipeline control is needed. Pipeline control definition is error-prone task and design of it takes long design time.

For the more effective architectural design space exploration, synthesis of datapath from behavioral description of instructions and pipeline control logic synthesis for user-defined pipeline organization are required. The ability of dealing with the user-defined pipeline organization is essential to evaluate various pipelined processor architectures. Datapath synthesis and pipeline control logic synthesis for user-defined pipeline organization and instructions can reduce the design time and design modification time drastically. Consequently, large design space for ASIPs can be explored in a short design time.

Chapter 3

PEAS-III: Processor Design Environment

This chapter describes micro-operation level processor specification and application specific instruction set processor (ASIP) design environment PEAS-III based on micro-operation level processor specification. First of all, characteristics of processor architecture are classified. Then, their impacts on performance and cost on the processor are evaluated for decision of flexibility on micro-operation level processor specification.

3.1 Characteristics of Modern Processor Architecture

Architectural characteristics of modern processors are classified into the following points:

- instruction set architecture: Instruction set architecture is an interface between software and hardware. Instruction set is influenced by many other architectural features described below.
- configuration of functional units: Performance of the functional unit affects execution time of application program. Hardware cost of the function unit affects total chip area. The functionality of the units and connectivity among them, in other words "datapath structure," restricts instruction set. The number of functional units determines how many operations are executed at the

same time.

- storage units' organization: Storage units' organization includes location of operands, the number of operands, size of register-file and memory, memory hierarchy and so on.

  The operands can be located in accumulators, special registers, general-purpose registers and memories. When operands are located in the accumulators or special registers, location of them are implicitly appointed by an instruction. Using implicit operands, the designer can reduce the instruction word length. However, load and store overhead from memory or register to accumulator or special registers makes execution time long. On the other hand, locations of operands are explicitly declared in an instruction when operands are located in a general-purpose register or in a memory.

  Furthermore, the processor architecture is classified to register-register architecture, register-memory architecture and memory-memory architecture whether operands are located in a general-purpose register or memory. Addressing modes for operands affect various fields such as instruction bit width, execution cycles, the number of address generation units and memory access units, pipeline organization and structural hazards.

  In general, register-register architecture and harvard architecture are preferred for the design of general purpose RISC processor. Complex memory architecture and memory-accumulator architecture are often preferred for data intensive digital signal processor design. For ASIP design, decision of suitable memory organization for applications is required.

- pipeline organization and pipeline hazard resolution policy: Clock frequency and pipeline hazard occurrence are influenced by pipeline organization in terms of the number of pipeline stages and role of each pipeline stage. The deep pipeline makes clock frequency high, but hardware cost of it also increases. Scheduled operations of each pipeline stage decide clock frequency of the processor. Specifying the operations of each pipeline stage also decides clock frequency, area, and condition of pipeline hazard occurrence and penalties of

them.

  The penalty of pipeline hazards increases execution time of application program. Several techniques to decrease penalty of pipeline hazards are proposed. Data forwarding, re-order buffer reduces data hazards. Delayed branch, branch prediction and non-overhead loop reduce the penalty of control hazards. Additional functional units for the division of the operations of conflicted resource resolve structural hazards. Selection of those techniques makes trade-off between performance and hardware cost.

- instruction issue and completion policy: The policies of instruction issue and completion are classified into in-order and out-of-order. Complex issue and completion mechanism make processor performance high but hardware cost becomes high, too.

- exception and interrupts: Exception and interrupt handling manner has some variations especially for architectures with out-of-order instruction completion. One of the exception mechanisms is to use history file or future file to keep original register values. Another approach is to store status of each pipeline stage in detail and let the interrupt handling routine to recover the pipeline status. The other is a technique that stops the instruction issue while it is uncertain that all the execution instructions will complete without causing an exception.

These characteristics are not orthogonal and influenced each other. The designer has to decide processor architecture in considering with these architecture characteristics and feature of target applications. To overcome the difficulty of architecture exploration, pipeline stage level processor design system is indispensable. PEAS-III is proposed as one of pipeline stage level processor design system.

For the architectural design space exploration in consideration of target application, micro-operation level processor specification and design system PEAS-III is proposed [34, 35]. PEAS-III enables the designer to do architectural design space exploration in a short design time. The designer can try various architecture candidates including following architecture variations: configuration of hardware modules, specification of application specific instructions which include multi-cycle operations, user-defined external interrupts, the number of branch delay slots, and the number of pipeline stages.

Figure 3.1 shows the organization of PEAS-III. The designer entries processor specification using GUI, "Architecture Design Entry System," and processor synthesis system generates micro-operation level simulation model and RT level processor description for logic synthesis in VHDL [36]. The designer selects resources from flexible hardware model database (FHM-DB) [37] and the processor synthesis system receives HDL descriptions of selected resources from FHM-DBMS. Estimation is also performed at each design step, architecture design phase and micro-operation specification phase. Estimation system also accesses to FHM-DBMS to get estimation results of selected resources. This thesis describes architecture level processor specification and processor synthesis.

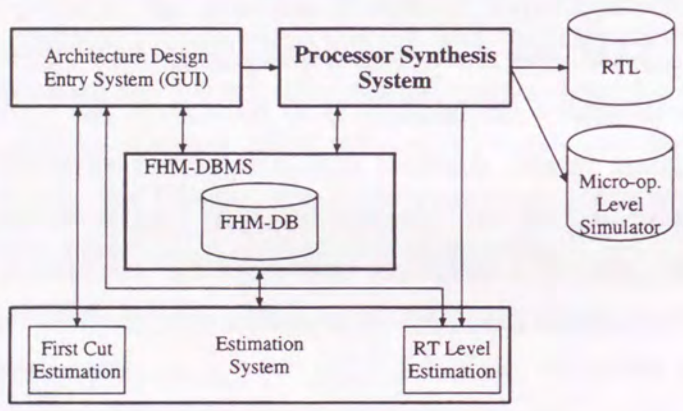

Architecture Design Entry System (GUI)
Processor Synthesis System
RTL
Micro-op. Level Simulator
FHM-DBMS
FHM-DB
First Cut Estimation
Estimation System
RT Level Estimation

Figure 3.1: PEAS-III System.

3.2 Design Methodology

Figure 3.2 shows a design flow of PEAS-III. With PEAS-III, processor is designed design step by step. Firstly, design goal and processor architecture type are set. Secondly, outline of the processor is specified. Specification in the second step includes declarations of resources, which are used in the processor, definition of instruction format and conditions of external interrupts, and definition of interface

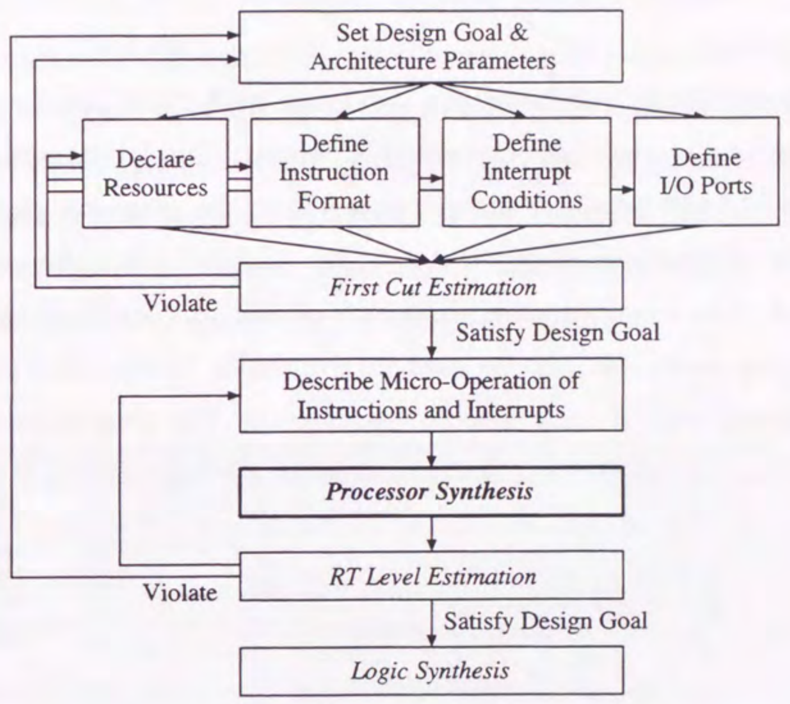


Figure 3.2: PEAS-III Design Flow.

ports. In the resource declaration, hardware modules are selected with appropriate parameters from parameterized hardware library FHM-DB. The designer can specify application specific interface between the processor core and other modules on SoC by specifying the external interrupt condition and specific processor interface ports. Then, area, clock frequency and power consumption of designed processor are estimated at the first cut estimation. When the estimation results do not satisfy the design goal, the designer changes architecture parameters, resources, instruction formats and so on to satisfy design constraint.

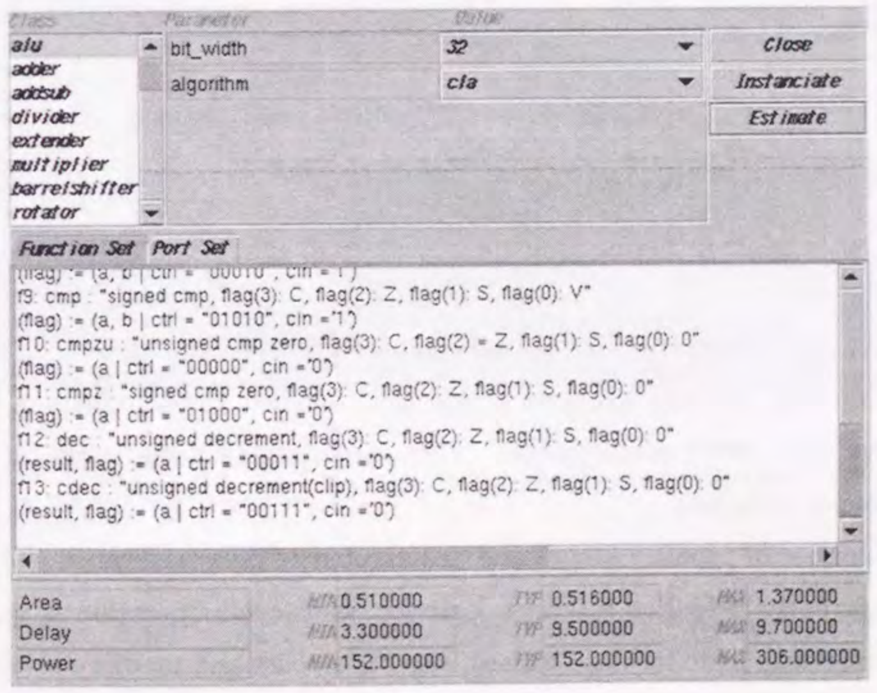
After the estimation results satisfy the design goal, clock based micro-operation description of instructions and interrupts is defined. Simulation model and synthesizable model of the processor are generated from the processor description. The functionality of the designed processor can be validated using the generated simulation model. The simulation model consists of behavior level instances in VHDL. The simulation model can also be used for evaluation of execution cycles of application programs, and for cycle based co-verification. The area, clock frequency and power consumption of the designed processor are evaluated from synthesized datapath and controller. When estimation results do not satisfy the design goal, the designer improves the processor design by re-scheduling operations of instructions to the pipeline stages or changing the number of pipeline stages. Re-scheduling may improve clock frequency and the number of pipeline stages improve area and clock frequency.

Description and modification time of micro-operation level processor specification is shorter than other existing processor description for synthesis because datapath and pipeline control logic are automatically generated. To generate datapath of designed processor, "Processor Synthesis System" inserts selectors for signal conflicts and pipeline registers for pipeline execution. The pipeline hazard detection and pipeline control logic for pipeline interlock and pipeline flush are also synthesized. The designer can concentrate on instruction set design.

3.2.1 Flexible Hardware Model

For architectural design space exploration, effective design reuse of hardware modules and frequent cut and try of them are required. For that purpose, flexible hardware model [38] is utilized. FHM is parameterized with various characteristics such as bit width, algorithm of the operation, etc., and various design instances can be generated according to the given parameter values. Since instances can be generated with various combinations of parameter values, the designer is able to evaluate many kinds of resources only by changing parameter values of FHM.

Several instances of different abstraction levels can be generated from an FHM. The processor synthesis system uses behavioral level instances to synthesis micro-operation level simulation model and gate level instances to generate RT level processor HDL description for logic synthesis. FHM provides estimation results of instances for various combinations of parameter values. The estimation results of FHMs are also used for estimation of the designed processor.



Parameter	Value
bit_width	32
algorithm	cla

```
f9: cmp : "signed cmp, flag(3): C, flag(2): Z, flag(1): S, flag(0): V"
(flag) := (a, b | ctrl = "01010", cin ='1')
f10: cmpzu : "unsigned cmp zero, flag(3): C, flag(2) = Z, flag(1): S, flag(0): 0"
(flag) := (a | ctrl = "00000", cin ='0')
f11: cmpz : "signed cmp zero, flag(3): C, flag(2): Z, flag(1): S, flag(0): 0"
(flag) := (a | ctrl = "01000", cin ='0')
f12: dec : "unsigned decrement, flag(3): C, flag(2): Z, flag(1): S, flag(0): 0"
(result, flag) := (a | ctrl = "00011", cin ='0')
f13: cdec : "unsigned decrement(clip), flag(3): C, flag(2): Z, flag(1): S, flag(0): 0"
(result, flag) := (a | ctrl = "00111", cin ='0')
```

	MIN	TYP	MAX
Area	0.510000	0.516000	1.370000
Delay	3.300000	9.500000	9.700000
Power	152.000000	152.000000	306.000000

Figure 3.3: Flexible Hardware Model Browser View.

Figure 3.3 shows an FHM browser. FHMs in FHM-DB are displayed in the left box. FHM parameters are shown in the upper-central box and the designer can select candidates of parameter values from the pull down menu on the right. Functionality of the selected FHM is shown in the central box. Estimation results of the FHM with selected parameters are shown at the bottom of the window. An FHM "alu" has a two parameters "bit_width" and "algorithm." "32" and "carry look ahead (cla)" are selected for the parameter value of "alu" respectively.

3.3 Micro-operation Level Processor Specification

The micro-operation level processor description consists of six major parts as follows:

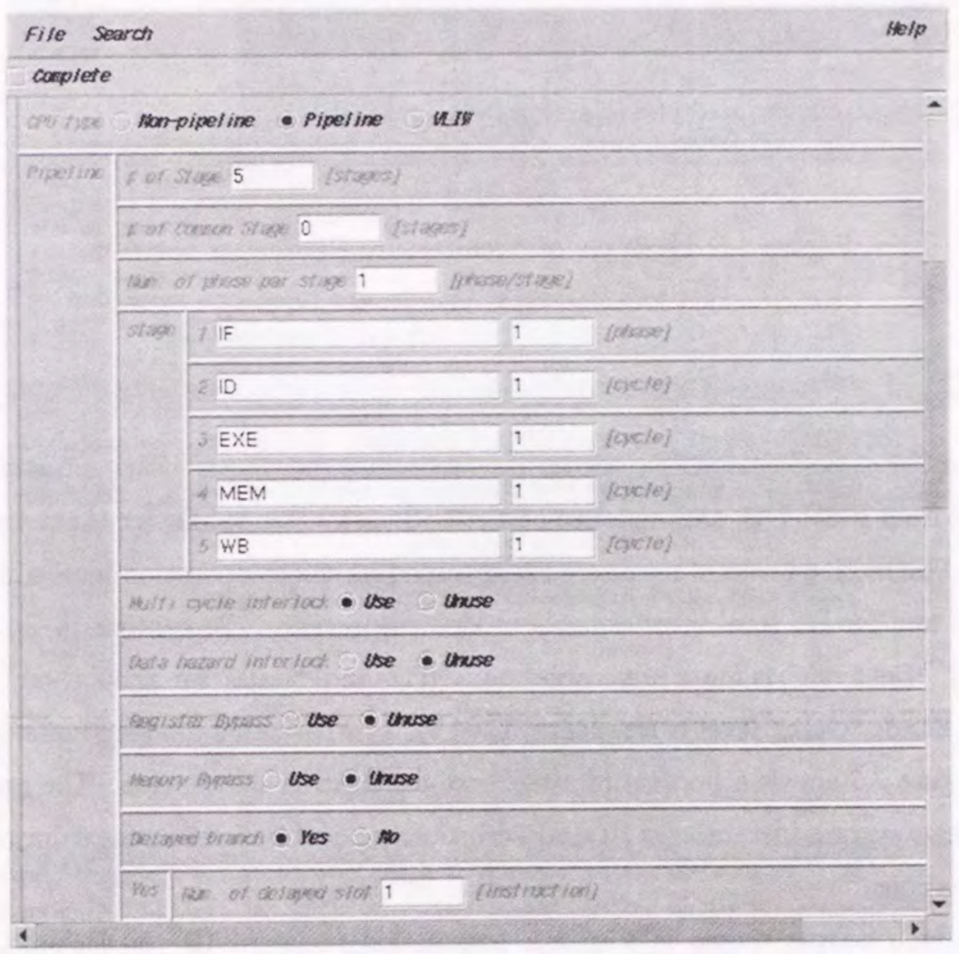
1. Design Goal and Architecture Parameter Setting
2. Resource Declarations
3. Instruction Format Definition
4. Interrupt Condition Definitions
5. Interface Definitions
6. Micro-operation Descriptions of instructions and interrupts

In this section, details of each part are described.

3.3.1 Design Goal and Architecture Parameter Setting

Figure 3.4 shows a portion of design goal and architecture parameter setting window. In this step, the designer specifies design goal of area, clock frequency, execution cycle count and power consumption. Then, architecture parameters for pipelined processors are specified.

The number of pipeline stages and the number of delayed branch slots are supported, currently. Pipeline interlock logic for multi-cycle operation is synthesized. Pipeline interlock logic for data hazard, register bypass and memory bypass are not synthesized. These parameters are prepared for future extension of PEAS-III. Figure 3.5 shows a portion of processor description, which is output from architecture



CPU type: Non-pipeline [✓ Pipeline] VLIW

Pipeline

# of Stage 5 [stages]

# of Common Stage 0 [stages]

Num. of phase per stage 1 [phase/stage]

stage			
1	IF	1	[phase]
2	ID	1	[cycle]
3	EXE	1	[cycle]
4	MEM	1	[cycle]
5	WB	1	[cycle]

Multi cycle interlock [✓ Use] Unuse

Data hazard interlock Use [✓ Unuse]

Register Bypass Use [✓ Unuse]

Memory Bypass Use [✓ Unuse]

Delayed branch [✓ Yes] No

Yes: Num. of delayed slot 1 [instruction]

Figure 3.4: Architecture Parameter Setting Window.

entry system (GUI). In the example the number of pipeline stages is five and delayed branch architecture is selected. The number of delayed branch slots is specified to '1'. It indicates that synthesized execute one succeeding instruction to the branch instruction whether branch is taken or not.

```
Abstract_level_architecture{
  Pipeline_architecture{
    Number_of_stages{"5"},
    Delayed_branch{"Yes"},
    Number_of_exec_delayed_slot{number{"1"}}}}
```

Figure 3.5: Example of Architecture Parameter Settings.

3.3.2 Resource Declarations

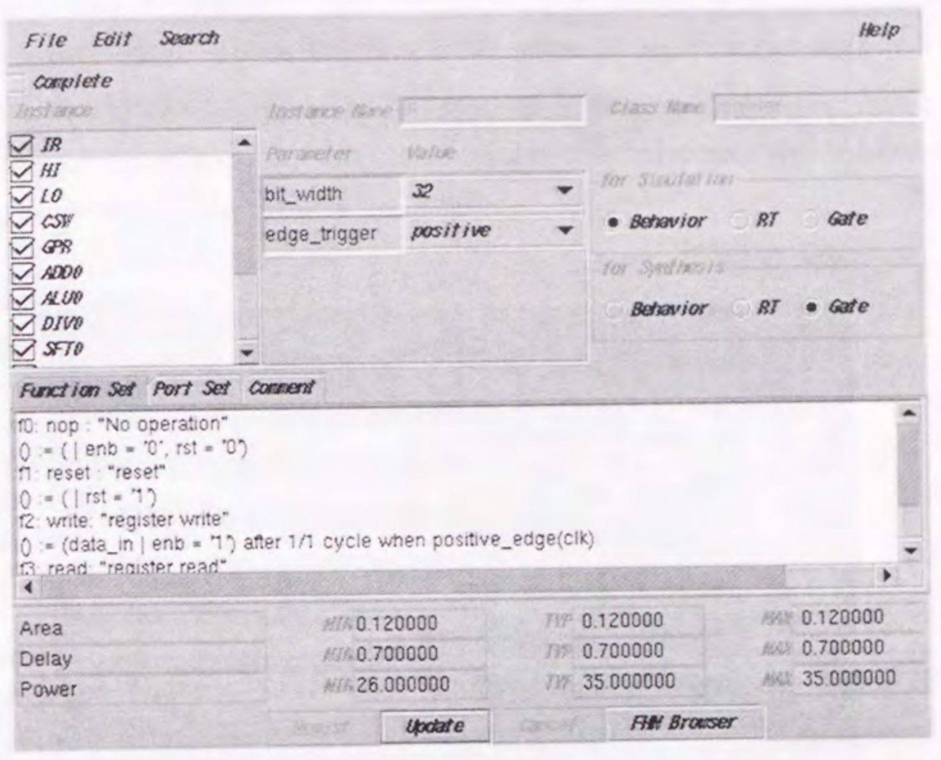
Figure 3.6 shows a resource declaration window. Flexible hardware models are selected from FHM-DB, and instance names and parameter values for them are specified. Abstraction levels of resources are specified for micro-operation level simulation model and for RT level synthesizable model, respectively. To synthesize simulation model, "Behavior" is more preferable than "RT" and "Gate" for simulation. On the other hand, "Gate" level is frequently used for synthesizable model generation.

Figure 3.7 shows a portion of a resource declaration description. The processor synthesis system instantiates HDL descriptions of declared resources from resource declarations.

In an example shown in Fig. 3.7, instruction register "IR" is declared. "IR" is a positive edge trigger type register and its bit width is "32." "Behavior" level instance is used for micro-operation level simulation model generation and "Gate" level instance is used for logic synthesizable model generation.

3.3.3 Instruction Format Definitions

Figure 3.8 shows an instruction format definition window. Bit fields, field type, field name, and binary value of it are defined for each instruction type. Field type is



Instance: ☑ IR, ☑ HI, ☑ LO, ☑ CSW, ☑ GPR, ☑ ADD0, ☑ ALU0, ☑ DIV0, ☑ SFT0

Parameter	Value
bit_width	32
edge_trigger	positive

for Simulation: ● Behavior ○ RT ○ Gate

for Synthesis: ○ Behavior ○ RT ● Gate

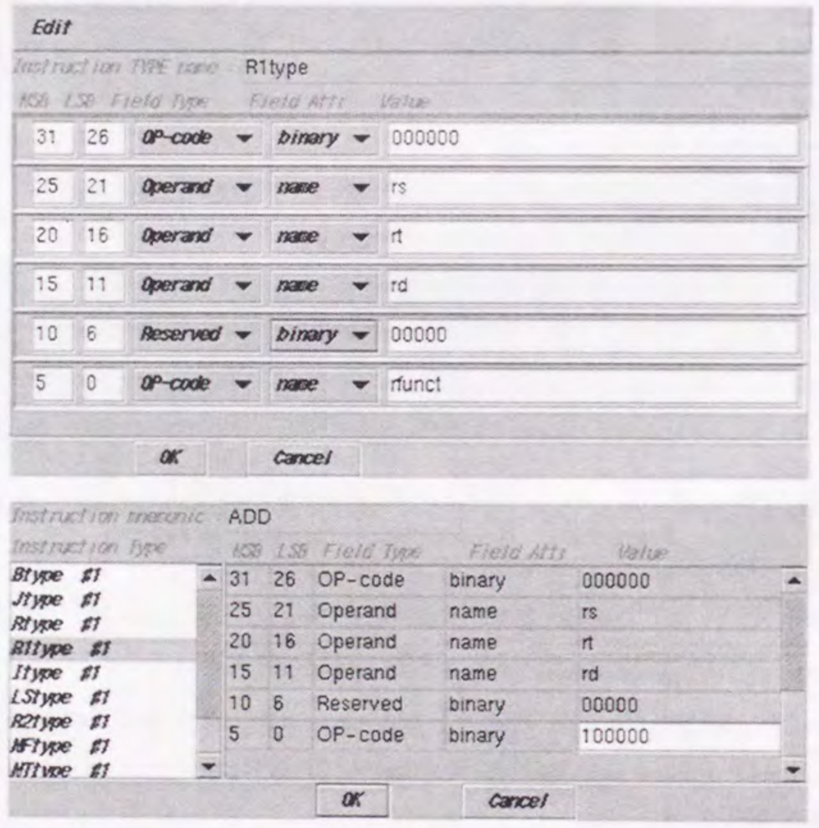
```
f0: nop : "No operation"
0 := ( | enb = '0', rst = '0')
f1: reset : "reset"
0 := ( | rst = '1')
f2: write: "register write"
0 := (data_in | enb = '1') after 1/1 cycle when positive_edge(clk)
f3: read: "register read"
```

	MIN	TYP	MAX
Area	0.120000	0.120000	0.120000
Delay	0.700000	0.700000	0.700000
Power	26.000000	35.000000	35.000000

Figure 3.6: Resource Declaration Window.

```
Resource{
  "IR"{
    class{"register"},
    classpath{""},
    parameter{
    abstraction_level{
      for_simulation{"Behavior"},
      for_synthesis{"Gate"}},
    bit_width{"32"},
    edge_trigger{"positive"}}}
```

Figure 3.7: Example of Resource Declarations.



Instruction TYPE name R1type

MSB	LSB	Field Type	Field Attr	Value
31	26	OP-code	binary	000000
25	21	Operand	name	rs
20	16	Operand	name	rt
15	11	Operand	name	rd
10	6	Reserved	binary	00000
5	0	OP-code	name	rfunct

Instruction mnemonic ADD

Instruction Type

Btype #1
Jtype #1
Rtype #1
R1type #1
Itype #1
LStype #1
R2type #1
MFtype #1
MTtype #1

MSB	LSB	Field Type	Field Attr	Value
31	26	OP-code	binary	000000
25	21	Operand	name	rs
20	16	Operand	name	rt
15	11	Operand	name	rd
10	6	Reserved	binary	00000
5	0	OP-code	binary	100000

Figure 3.8: Instruction Format Definition Window.

selected among “op-code,” “operand” and “reserved.” “op-code” means operation code and “reserved” indicates that the field is reserved for extension in the future. Operation code value is specified when the value is constant for all instructions belongs to that type, and the value for reserved field is also specified.

Then, for each instruction, instruction type is selected among defined instruction types and operation code value is decided.

```
Instruction_type{
 "R1type"{
   "OP-code"{"binary"{"000000"},width{"31","26"}},
   "Operand"{"name"{"rs"},width{"25","21"}},
   "Operand"{"name"{"rt"},width{"20","16"}},
   "Operand"{"name"{"rd"},width{"15","11"}},
   "Reserved"{"binary"{"00000"},width{"10","6"}},
   "OP-code"{"name"{"rfunct"},width{"5","0"}}}}
   :
Instruction{
 "ADD"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
   "Operand"{"name"{"rs"},width{"25","21"}},
   "Operand"{"name"{"rt"},width{"20","16"}},
   "Operand"{"name"{"rd"},width{"15","11"}},
   "Reserved"{"binary"{"00000"},width{"10","6"}},
   "OP-code"{"binary"{"100000"},width{"5","0"}}}
```

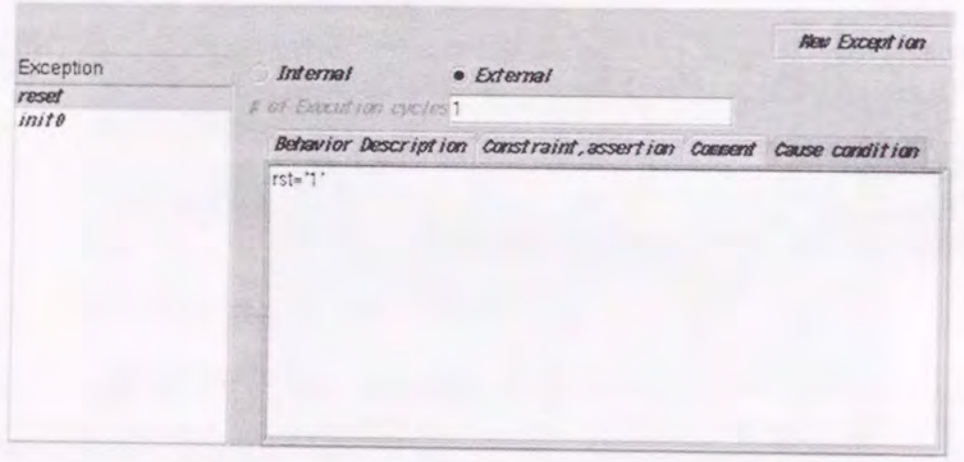
Figure 3.9: Example of Instruction Format Definitions.

In micro-operation descriptions, bit field of the instruction is referred by the field name that is defined in instruction format definition phase. Modification of instruction format which includes varying instruction bit width, re-ordering instruction fields, changing operation code and so on do not require modification of micro-operation description of instructions. When bit width, name and role of the field are not changed, there is no need to modify micro-operation description. Instruction code definition is used to generate instruction decoder, which is mentioned in Section 4.4.1 and Section 5.2.4.

In an example shown in Fig. 3.9, an instruction type “R1type” and an instruction “ADD” which belongs to “R1type” are defined. The instruction type “R1type” has six instruction fields. The range of the first field is from "31" to "26." The type of the first field is "OP-code" and its value is constant "000000." The second and the third fields indicate register address of source operands and the forth field indicates destination register address. The fifth field is reserved for future extension. The last filed is an operation code for R1type instructions. The operation code for "ADD" is "100000."

3.3.4 Interrupt Condition Definitions

Figure 3.10 shows an interrupt condition definition window. Interrupt definitions include interrupt conditions and the number of execution cycles of the interrupt. In the example of interrupt "int0." Processor receives interrupt "int0" when external input port "INT" received '1', and needs one cycle to process the interrupt "int0."



Internal [✓ External]

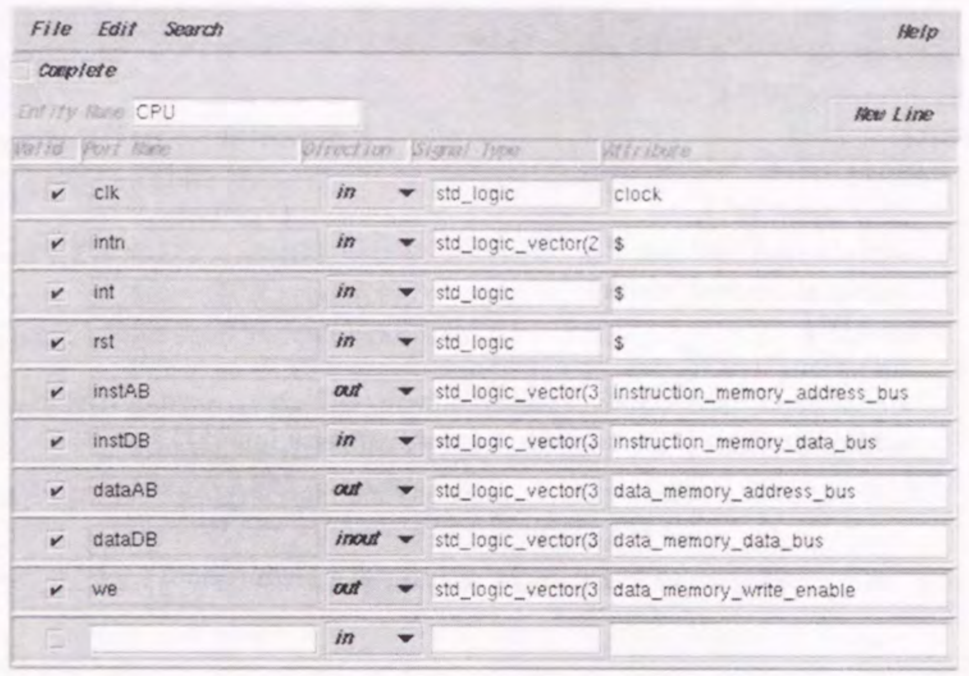
# of Execution cycles: 1

```
rst='1'
```

Figure 3.10: Interrupt Condition Definition Window.

3.3.5 Interface Definitions

Figure 3.11 shows an interface definition window. In an interface definition, an entity name, and input and output ports of target processor are defined. Port name, direction, type and attribute of processor interface ports are also defined. For the standard processor, memory interface port, clock port, reset port and external



Entity Name: CPU

Valid	Port Name	Direction	Signal Type	Attribute
✓	clk	in	std_logic	clock
✓	intn	in	std_logic_vector(2	$
✓	int	in	std_logic	$
✓	rst	in	std_logic	$
✓	instAB	out	std_logic_vector(3	instruction_memory_address_bus
✓	instDB	in	std_logic_vector(3	instruction_memory_data_bus
✓	dataAB	out	std_logic_vector(3	data_memory_address_bus
✓	dataDB	inout	std_logic_vector(3	data_memory_data_bus
✓	we	out	std_logic_vector(3	data_memory_write_enable
		in		

Figure 3.11: Interface Definition Window.

interrupt ports are usually defined. Furthermore special purpose interface port can be declared.

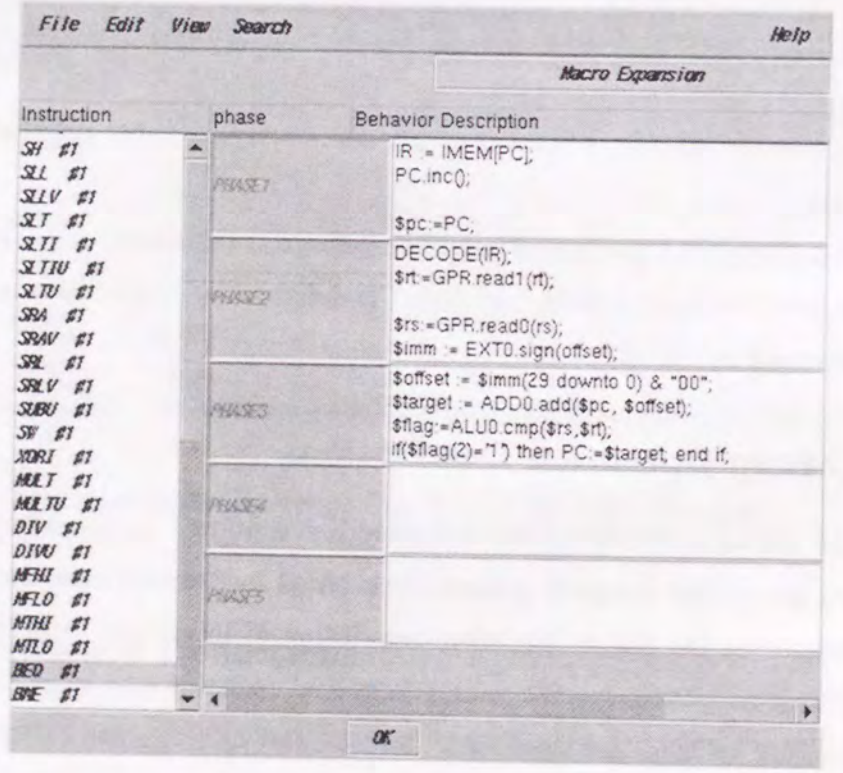
Figure 3.12 shows a portion of interface definition description. In the example, clock port "clk" of which type is "std_logic" is defined. "std_logic" is a bit type that is generally used in VHDL.

3.3.6 Micro Operation Descriptions

Figure 3.13 shows a micro-operation description window. In the micro-operation description phase, the designer defines clock based instruction behavior and interrupt behavior. In the micro-operation description of interrupts, operations of the processor such as setting specific values to special registers and jumping to the interrupt handler routine, are described. Micro-operation consists of three kinds of statements: (i) Operations which are executed by resources, e.g. arithmetic and logic operation, read, register write, (ii) Data transfers between resources, and (iii) Conditional execution of (i) and (ii).

```
Port_declaration{
  entity_name{"CPU"},
    Port{
      "clk"{
        direction{"in"},
        signal_type{"std_logic"},
        signal_attribute{"clock"}},
      "instAB"{
        direction{"out"},
        signal_type{"std_logic_vector(31 downto 0)"},
        signal_attribute{"instruction_memory_address_bus"}}}}
```

Figure 3.12: Example of Interface Definitions.



```
IR := IMEM[PC];
PC.inc();

$pc:=PC;

DECODE(IR);
$rt:=GPR.read1(rt);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);

$offset := $imm(29 downto 0) & "00";
$target := ADD0.add($pc, $offset);
$flag:=ALU0.cmp($rs,$rt);
if($flag(2)='1') then PC:=$target; end if;
```

Figure 3.13: Micro-operation Description Window.

```
MOT{
  mnemonic{
    "BEQ"{
      clk(1){"IR := IMEM[PC];
             PC.inc();
             $pc:=PC;"},
      clk(2){"DECODE(IR);
             $rt:=GPR.read1(rt);
             $rs:=GPR.read0(rs);
             $imm := EXT0.sign(offset);"},
      clk(3){"$offset := $imm(29 downto 0) & \"00\";
             $target := ADD0.add($pc, $offset);
             $flag:=ALU0.cmp($rs,$rt);
             if($flag(2)='1') then PC:=$target; end if;"},
      clk(4){""},
      clk(5){""}}}}
```

Figure 3.14: Micro-operation Description of instruction BEQ.

Figure 3.14 shows an extracted description of Figure 3.13. In the example, a micro-operation description of an instruction "Branch on Equal (BEQ)" is described. The instruction "BEQ" jumps to "PC + offset * 4" when register values of "rs" and "rt" are the same. Capitalized identifiers, such as "IR" and "ALU0" denote resources declared in the resource declaration phase. Symbol ":=" denotes assignment. Identifiers which begin with '$' are temporal variables. An identifier surrounded by symbols "[" and "]" specifies address to memory or register file. The expression "DECODE(IR)" in the second stage denotes that an instruction code is decoded in the second stage, where "IR" is an instruction register. The expression "$flag := ALU0.cmp($rs, $rt)" in the third stage denotes that values stored in "$rs" and "$rt" are compared using resource "ALU0" and the result will stored in "$flag." The "if" statement in the third stage is an example of conditional execution.

Definition and modification of micro-operation description are easy because designer does not need to take care of selectors, pipeline registers and pipeline control logic. PEAS-III generates HDL description of ASIPs from user-defined micro-operations of instructions and interrupts by inserting selectors and pipeline registers automatically, and generating control logic for pipeline interlock and pipeline flush.

```
Exception{
  "reset"{
    Condition{"rst='1'"},
    Type{"External"},
    Cycles{"1"},
    MOD{
      clk(1){"PC.reset(); GPR.reset();
              EPC.reset(); HI.reset();
              LO.reset(); IR.reset();"}}},
  "ini0"{
    Condition{"int = '1' and intn = \"000\""},
    Type{"External"},
    Cycles{"1"},
    MOD{
      clk(1){"EPC := PC;
              PC :=\"1000000000000000000000000010000000\";"}}}}
```

Figure 3.15: Example of Interrupt Definitions.

Figure 3.15 shows an example of interrupt definition description. Defined interrupt conditions and micro-operation description of interrupts are combined in the description. In the example, the processor detects the interrupt "int0" when input port "int0" receives '1' and value of program counter (PC) is stored in exception program counter (EPC) and PC is updated to "0x800080."

Chapter 4

Processor Model

In this chapter, processor model for processor synthesis is described. In Section 4.1, limitation of target processor is discussed. In Section 4.2, requirements of the processor model is described and proposed processor organization are described. In Section 4.3, organization of datapath and controller are described. Processor control mechanism which includes pipeline interlock and pipeline flush is demonstrated.

4.1 Processor Class

Feature of the target architecture of processor synthesis includes:

- single phase straightforward pipelined processor. PEAS-III assumes pipeline architecture, but the number of pipeline stages and operations assigned to each pipeline stage are flexible. Each pipeline stage is proceeded synchronously with positive edge of a clock.
- delayed branch with predict-not-taken policy. The designer can specify the number of delayed branch slot. The processor executes succeeding specified number of instructions whether branch is taken or not, and nullifies other fetched instructions when branch is taken.
- multi-cycle operation. PEAS-III is able to deal with multi-cycle units such as sequential multiplier, memory access units and so on. The processor synthesized by PEAS-III stalls succeeding instructions until multi-cycle operation is completed.

- in-order instruction issue and in-order completion. Out-of-order completion and out-of-order instruction issue are not supported.
- external interrupt. User-defined external interrupts are supported. On the other hand, internal exceptions are not supported.
- flexible addressing modes, storage organization. The designer is able to design addressing modes freely in micro-operation description of instructions. Multi port memory and multiple memories can be used.
- single word instruction. The width of instruction word is user-defined constant. Multi-word instruction is not directly supported.

The designer can specify data forwarding in micro-operation description of instructions. Data hazard detection and data forwarding logic are not automatically generated from micro-operation description of instructions.

4.2 Processor organization

Since the number of pipeline stages is parameterized and micro-operations of each stage is defined by the designer, flexible processor model is required.

Figure 4.1 shows an example of a pipelined processor organization [39]. This processor consists of five stages, instruction fetch (IF), instruction decode and operand fetch (ID), execution (EXE), memory access (MEM) and register write back (WB) stage. In general, operations in a pipeline stage complete in one clock cycle and store the result to pipeline registers. The operation results are referred from the nest stage at the next clock cycle.

To deal with flexibility in pipeline depth of target processor, datapath and controller is divided into pipeline stages like Fig. 4.2. Specified number of datapath and controller sets for each pipeline stage are arranged and connected together. A set of datapath and controller is added or deleted when the number of pipeline stages is changed.

Figure 4.3 shows a processor model for five stage pipelined processor. The model consists of five sets of datapath and pipeline stage controller, instruction decoder and

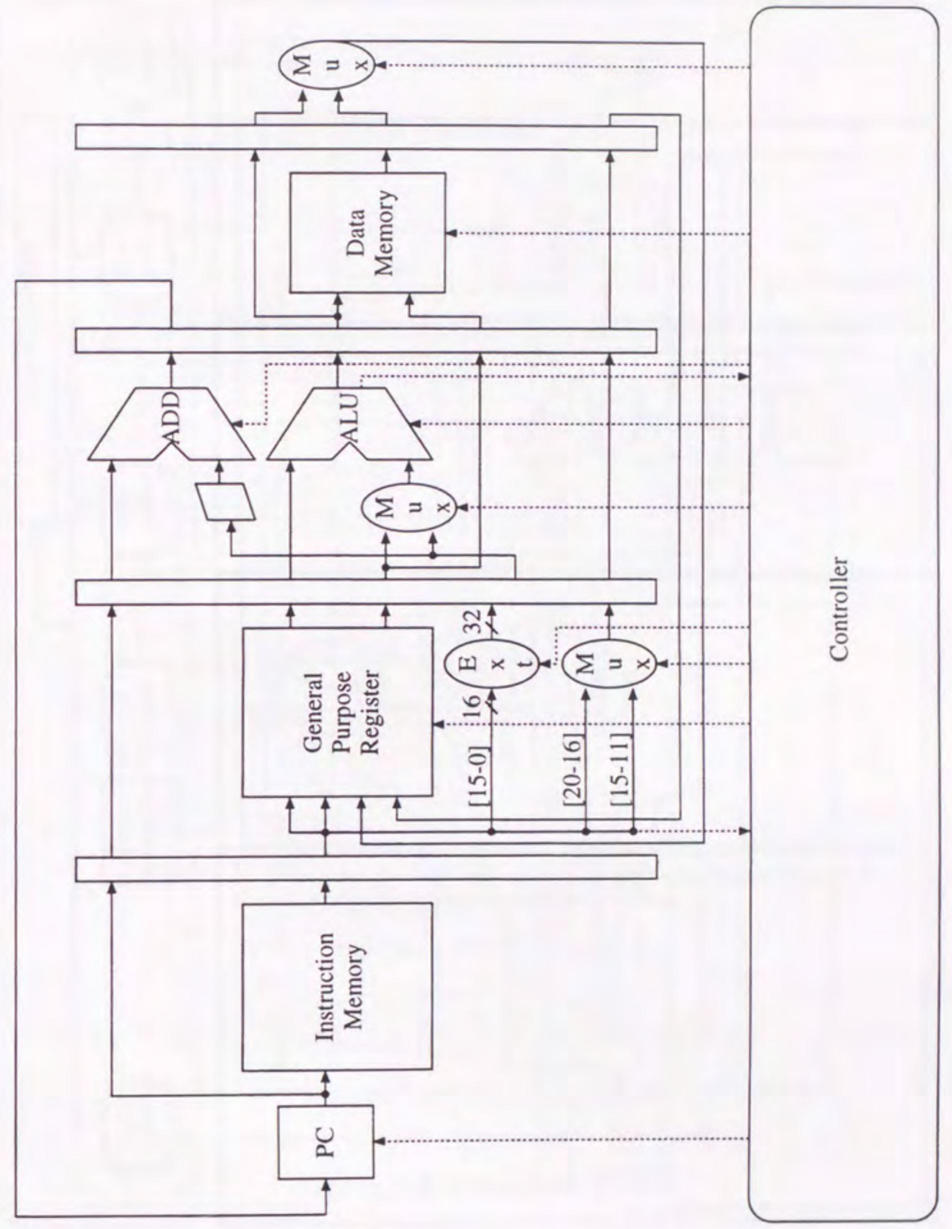


Figure 4.1: Example of Datapath and Controller of Pipelined Processor.

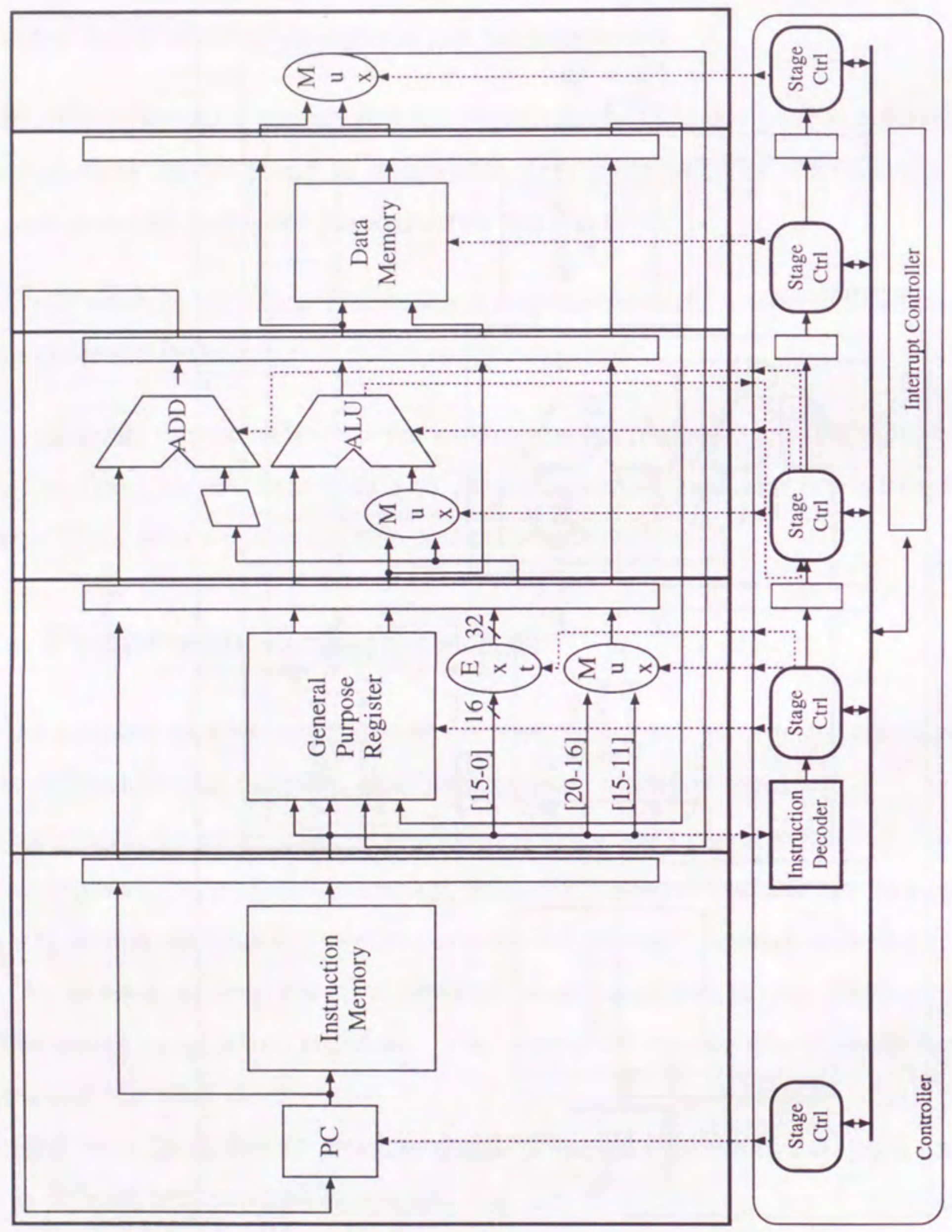


Figure 4.2: Example of Pipelined Processor Divided into Pipeline Stages.

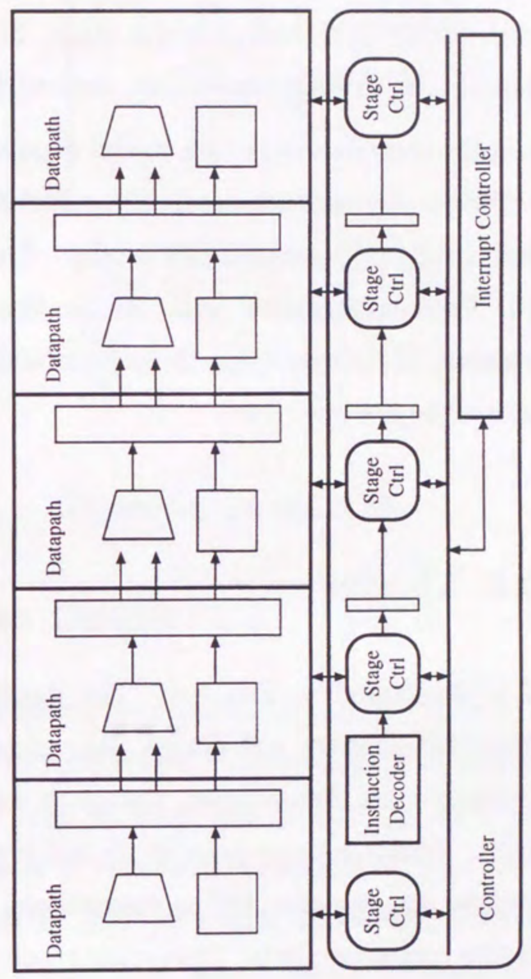


Figure 4.3: Processor Model.

interrupt controller. Instruction decoder is arranged to the instruction decode stage indicated by keyword "DECODE" in micro-operation description. The term "stage controller" is used to indicate a controller arranged to each pipeline stage. The stage controller sends control signals to resources in the datapath and manages pipeline flush and interlock. The stage controllers and the interrupt controller communicate each other. The stage controller determines the pipeline stall and the next state from the output of controller of next and previous stage. Since the load of pipeline control logic is distributed to each stage controllers, controller synthesis is simplified.

The rest of this chapter describes datapath model of pipeline stages, instruction decoder, stage controllers and interrupt controller. Section 4.3 describes datapath model and Section 4.4 describes controller model. The organization of stage controller is described. Pipeline interlock and pipeline flush using proposed stage controller are demonstrated. In Section 4.4.3, the organization of interrupt controller and how to handle interrupts are described.

4.3 Datapath Model

The datapath model is illustrated in Fig. 4.4. The datapath model consists of resources, selectors, pipeline registers and connections among them. From micro-operations that are described by the designer, datapath and controller are implemented using this model. Resource operations in micro-operations are executed by resources, and assignments are implemented as connections between resources. Selectors are used to resolve signal conflicts. Operation results are transferred to the next stage via pipeline registers.

4.4 Controller Model

Controller consists of three major ports, such as instruction decoder, stage controllers, and interrupt controller.

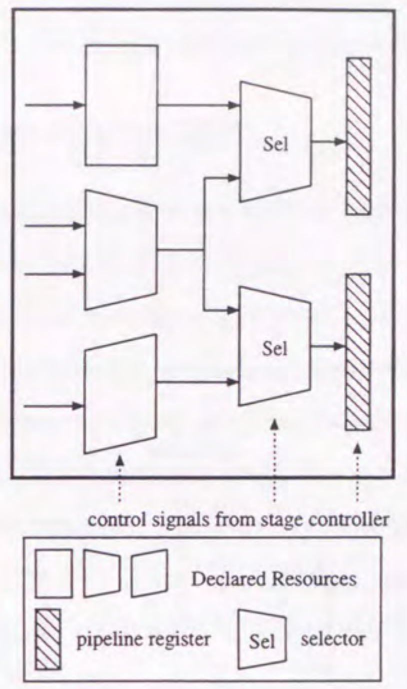


Figure 4.4: Datapath Model.

4.4.1 Instruction Decoder

There are two ideas of instruction decode shown in Fig. 4.5. The one is to execute instruction decode in the instruction decode stage. The other is to send instruction code to pipeline stage step by step and decode the code in each pipeline stage. The former method leads to shorter critical path of pipeline stage than the latter method because the latter method makes additional delay of instruction decode for each pipeline stage. The latter one, however, makes decoding logic simple. In this thesis, the former type instruction decoder is adopted to generate high-speed processor.

Instruction decoder in this thesis identifies which instruction is fetched and generates two types of control signals in the instruction decode stage: control signals for resources and instruction identification signals for stage controllers. The latter is used to judge whether executing instruction in the pipeline stage belongs to a certain set of instructions or not. Generated signals are transferred to the stage controllers

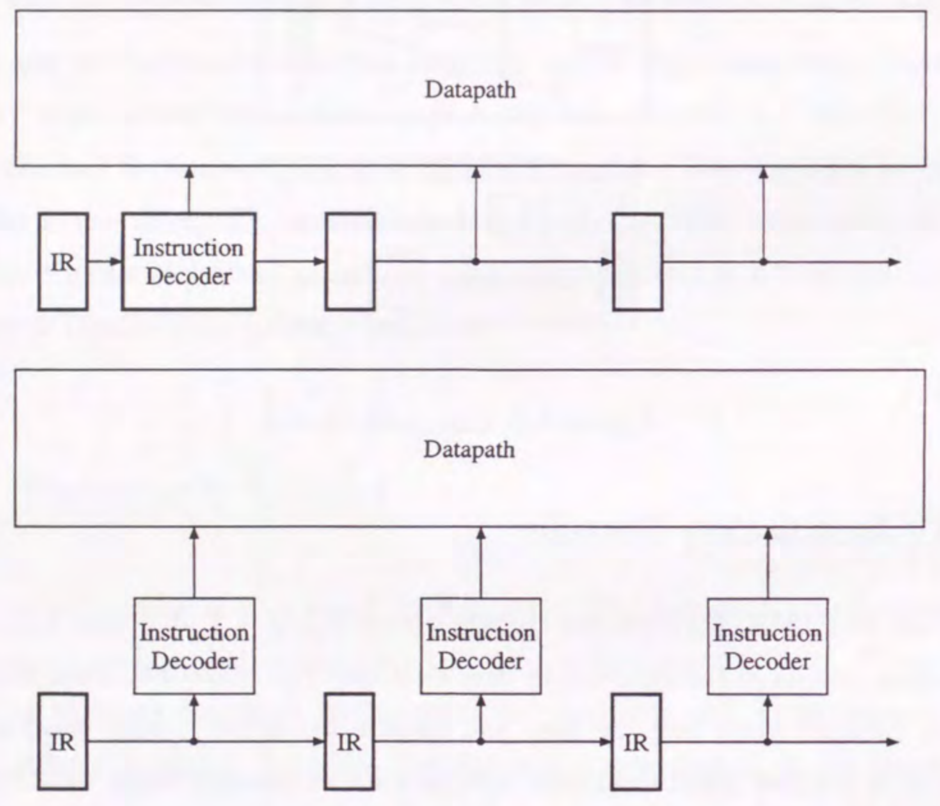


Figure 4.5: Example of Two Types of Instruction Decoder.

step by step synchronously with pipeline execution. The behavior of stage controller is described in Section 4.4.2 and usage of instruction decode result are explained.

4.4.2 Pipeline Stage Controller

The stage controller generates control signals for resources, pipeline registers, and selectors. The controller assigns control signals to resources to execute described micro-operations. The controller also manages pipeline registers to transfer the data to next stage as usual, and to keep the operation results in the case of pipeline interlock. The stage controller also regulates pipeline execution in the sense of pipeline interlock and pipeline flush. The controller stalls the pipeline to wait for completion of multi-cycle operation and resolution of resource conflicts. The controller flushes the pipeline by nullifying executing instructions when branch is taken.

Control model of the stage controller is based on the pipeline control model published in [40]. In [40], pipeline controller synthesis for pipeline interlock from usage information of resources is discussed. In this thesis, instead of usage information of resources, structural hazard detection method is proposed. Furthermore, the pipeline controller is extended to pipeline flush and suspension of instruction fetch.

The controller model is common to all pipeline stages. Decision of next state and generation of control signal are distributed to each pipeline stage. Distributed control logic makes controller organization and synthesis method simple.

Suppose n is the number of pipeline stages and $k (1 \leq k \leq n)$ is the stage number, the controller of each stage k is represented by finite state machine

$$M_k = (q_k, I_k, O_k, \delta_k, \rho_k, nop)$$

and datapath control signal generator. Each item of M_k is defined as follows:

states variable: $q_k \in \{nop, exec\}$

input signals: $I_k \triangleq \{branch, lock_k, go_{k-1}, go_{k+1}, valid_{k-1}, valid_{k+1}\}$

output signals: $O_k \triangleq \{valid_k, go_k\}$

next-state function:

$$\delta_k(q_k, branch, lock_k, go_{k-1}, go_{k+1}, valid_{k-1}, valid_{k+1})$$

$$\triangleq \begin{cases} exec & \text{when} \quad (\overline{branch} + \overline{cancel(k)}) \cdot (valid_{k-1} \cdot go_{k-1} + \\ & \qquad\qquad (q_k = exec) \cdot (lock_k + valid_{k+1} + \overline{go_{k+1}})) \\ nop & \text{otherwise} \end{cases}$$

output functions: $\rho_k \triangleq \{\rho_{valid_k}, \rho_{go_k}\}$

$$\begin{aligned} \rho_{valid_k}(q_k) &\triangleq (q_k = exec) \\ \rho_{go_k}(q_k, lock_k, valid_{k+1}, go_{k+1}) &\triangleq (q_k = exec) \cdot \overline{lock_k} \cdot (\overline{valid_{k+1}} + go_{k+1}) \end{aligned}$$

initial status: *nop*

The status variable q_k indicates whether executable instruction exists in the k-th stage or not. When $q_k = exec$, an instruction exists in the k-th stage. The value of q_k becomes *nop* when pipeline is stalled and valid instruction is not moved to the k-th stage or pipeline flush is executed, etc. $q_k = nop$ means there is “no operation” in the k-th stage. The initial of value q_k is *nop*.

Values of input signals are specified as follows:

$$branch = \begin{cases} true & \text{when branch is taken} \\ false & \text{when branch is not taken} \end{cases}$$

$$lock_k = \begin{cases} true & \text{when an instruction in the } k\text{-th stage causes pipeline interlock} \\ false & \text{otherwise} \end{cases}$$

$$go_k = \begin{cases} true & \text{when an instruction in the } k\text{-th stage is transfered to the next stage} \\ false & \text{when an instruction in the } k\text{-th stage stays} \end{cases}$$

$$valid_k = \begin{cases} true & \text{when valid instruction exists in the } k\text{-th stage} \\ false & \text{when no instruction exists in the } k\text{-th stage} \end{cases}$$

The values of go_{n+1}, $valid_{n+1}$ are defined as $go_{n+1} = true$, $valid_{n+1} = false$. An input signal go_0 is an output signal of interrupt controller.

Next-state function δ_k outputs $exec$ if and only if the following conditions are satisfied.

- branch is not taken or the k-th stage does not need to nullify instruction when branch is taken.
- An instruction in the $(k-1)$-th stage will reach or current instruction in the k-th stage stays.

The function $cancel(k)$ holds if and only if the k-th stage has to nullify current instruction when branch is taken. Detail of $cancel(k)$ is described in the following section.

The interrupt controller outputs $true$ for go_0, as usual. However it output $false$ when interrupt is occurred and suspension of instruction fetch is required. When $go_0 = false$ and $go_1 = true$, next state q_1^+ becomes *nop*, and operations of the first stage will be stopped. If the instruction in the first stage does not stay, execution of the first stage will be stopped at next clock.

Output signal go_k becomes $false$ if and only if at least one of the following conditions is satisfied.

- The k-th stage causes pipeline interlock
- An instruction in the $(k+1)$-th stage does not move to the $(k+2)$-th stage.

When the k-th stage causes pipeline interlock by multi-cycle operations or resource conflicts, go_k becomes $false$ and the instructions in the succeeding $1 \leq i \leq (k-1)$-th stages are also stalled.

Control signals to datapath resources are generated from output signals O_k of stage controller M_k, results of instruction decoder and output signal of interrupt controller. Stage controller outputs control signal for described micro-operation of executing instruction in the k-th stage as usual. The controller outputs the control signal to hold the status of resources when the pipeline is stalled ($go_k = false$).

Pipeline hazards are classified as follows:

- structural hazard, which is caused by multi-cycle operations and resource conflicts,
- and control, hazard which is caused by branch.

For the structural hazard, pipeline is interlocked until the multi-cycle operations are completed and resource conflicts are resolved. For the control hazard, some instructions in the pipeline stages are flushed when branched. In the following section, pipeline control mechanism and the control logic of $lock_k$, $branch$ and function

$cancel(k)$ are described. Pipeline interlock signal $lock_k$ is described as follows:

$$lock_k = lock\_m_k + lock\_r_k$$

$lock\_m_k$ is a pipeline interlock signal for multi-cycle operations and $lock\_r_k$ is a pipeline interlock signal for resource conflicts.

Pipeline Interlock caused by Multi-cycle Operations

When multi-cycle operation is executed in the k-th stage, instruction transfer from stage j $(1 \leq j < k)$ to stage $j+1$ is suspended to stall succeeding instructions.

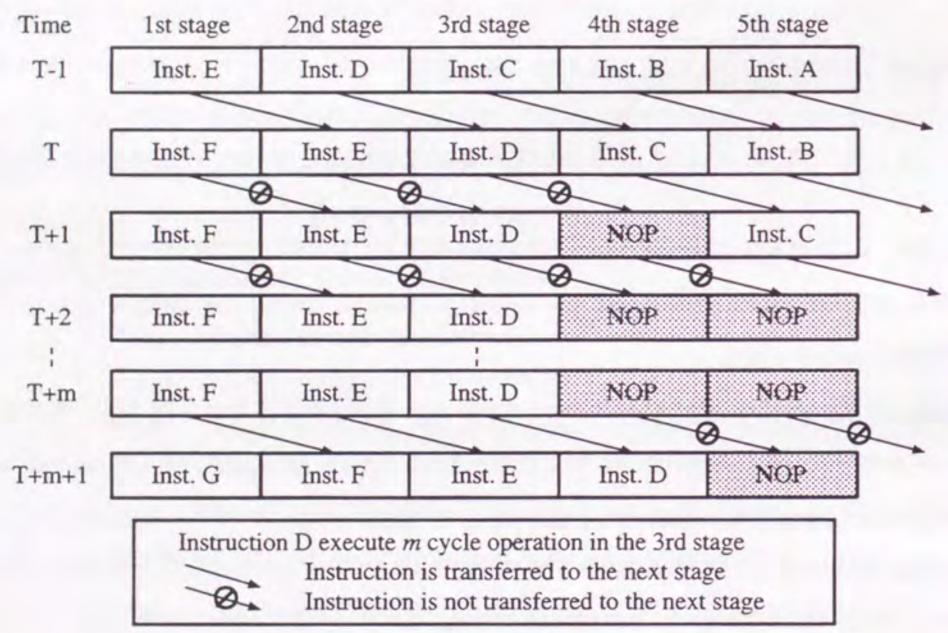

Time
1st stage
2nd stage
3rd stage
4th stage
5th stage
T-1
T
T+1
T+2
T+m
T+m+1
Inst. A
Inst. B
Inst. C
Inst. D
Inst. E
Inst. F
Inst. G
NOP
Instruction D execute m cycle operation in the 3rd stage
Instruction is transferred to the next stage
Instruction is not transferred to the next stage

Figure 4.6: Example of Multi-cycle Operation.

Figure 4.6 shows an example of pipeline interlock caused by multi-cycle operation. Suppose instruction D executes m cycle operation at the third stage from time T. The instructions in the first, second and third stages are not transferred to the next stage while multi-cycle operation is executed. The state of fourth stage becomes "no-operation" because instruction in the third stage is not transferred. At time $T+m$, multi-cycle operation is completed and then instructions in the first, second and third stage are transferred to the next stage at time $T+m+1$.

In the case of pipeline stall, the j-th stage controller assigns control signals to storage resources to disable write back while instruction transfer is suspended.

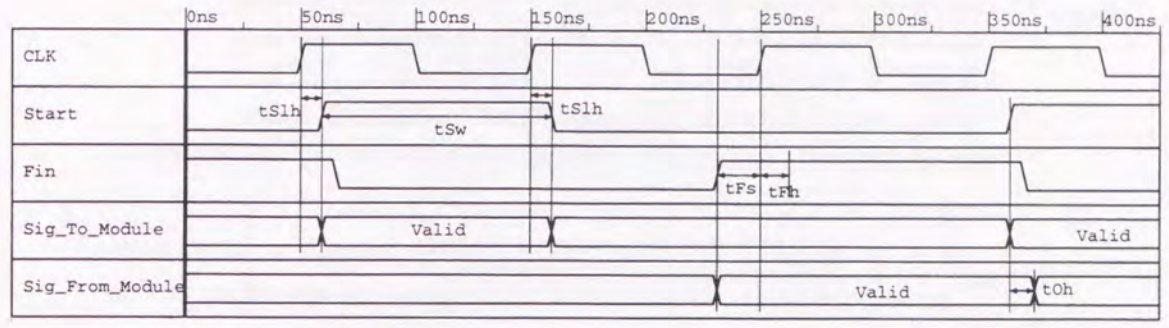


Figure 4.7: Timing Interface Between Controller and Multi-cycle Resource.

Figure 4.7 shows a timing interface between the controller and multi-cycle resources. The controller makes start signal "Start" active for one cycle and then the resource starts operation. After the multi-cycle operation is finished, the resource outputs the result and changes the value of the flag "Fin" active to inform the completion of the operation. When multiple multi-cycle operations are executed in the same stage and the same instruction, the stage controller stalls the pipeline until all multi-cycle operations are finished. The operation results and completion flag must be kept until other multi-cycle operations are finished. Because the resources keep operation results and flag values until next operation starts, additional structure for saving the results and flags are not required. The interface information that includes start signal input port, flag output port, and active value of them can be obtained from FHM-DB.

Suppose $U_{k,r,op} = \{(exp, inst) \mid exp \in Exp, inst \in I\}$ is a set of conditional expression exp and instruction $inst$ pairs, which represent execution conditions of operation op of resource r in the k-th stage. In another words, an operation op of resource r in the k-th stage is executed if and only if one of the executing instructions is $inst$ and condition exp holds. The control logic of $lock\_m_k$ for multi-cycle operations is represented as follows:

$$lock\_m_k = \bigvee_{\substack{r \in R \\ op_m \in OP_m}} \bigvee_{(exp, inst) \in U_{k,r,op_m}} (inst_k = inst) \cdot exp \cdot \overline{fin_{op_m}} \tag{4.1}$$

where

R: a set of resources
OP_m: a set of multi-cycle operations in the k-th stage
$inst_k$: indicates executing instruction name in the k-th stage

$$fin_{op_m} = \begin{cases} true & \text{after multi-cycle operation } op_m \text{ is completed} \\ false & \text{during multi-cycle operation } op_m \text{ is executing} \end{cases}$$

Equation (4.1) means that $lock\_m_k$ holds if and only if at least one multi-cycle operation is not completed. $lock\_m_k$ becomes $false$ after all the multi-cycle operations are completed.

The start signal of the multi-cycle operation is activated at the first cycle, and then negated from the second cycle to the start of the next multi-cycle operation. In the example, control signal for multiplier is activated at time T and then negated at time $T + 1$. Suppose v_{active} for the active value of control signal $start$, control logic of $start$ is as follows:

$$start_{r,k} = \begin{cases} v_{active} & \text{when } flag_k \cdot \bigvee_{op_m \in OP_m} \bigvee_{(exp, inst) \in U_{k,r,op_m}} (inst_k = inst) \cdot exp \\ \overline{v_{active}} & \text{otherwise} \end{cases} \tag{4.2}$$

$$flag_k^+ = go_{k-1} \tag{4.3}$$

$flag_k$ is a register, which indicates whether it is the first cycle of multi-cycle operation or not. The value $flag_k$ becomes $true$ when new instruction is transferred to the k-th stage and becomes nop when execution instruction stays in the k-th stage.

Pipeline Interlock caused by Resource Conflict

When resource conflict is occurred between stage k and stage j ($k < j$), the k-th stage is stalled until completion of the j-th stage's operation.

Figure 4.8 shows an example of resource conflict. An example processor shares a single-memory for data and instructions. The first stage is the instruction fetch

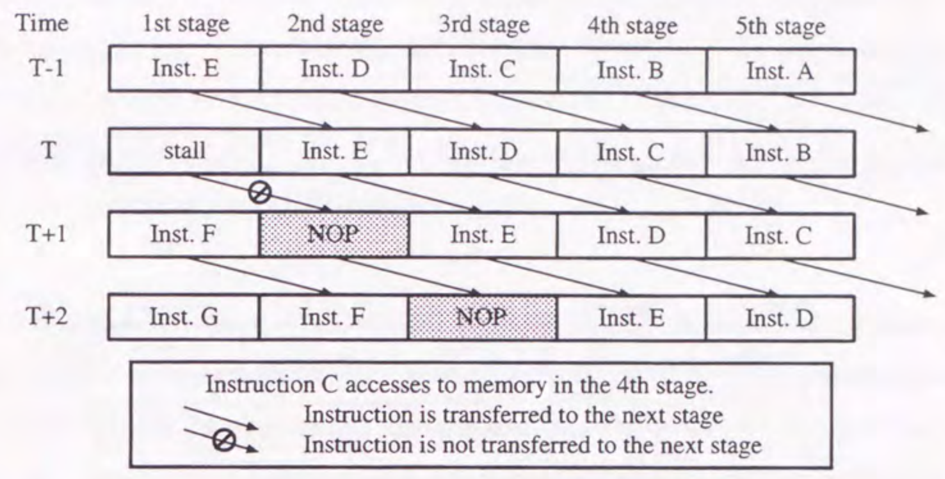


Figure 4.8: Example of Resource Conflict.

stage and the fourth stage is the memory access stage. Suppose an instruction C is a memory access instruction. The first stage is stalled at time T. After the instruction C completes memory access operation and moves to the fifth stage, memory access in the first stage is executed at time $T + 1$.

Suppose $V_{r,k} = \{inst \mid inst \in I\}$ is a set of instructions, which represents the instructions that use the resource r in the k-th stage. To put it in another way, a resource r is accessed from instruction $inst$ in the k-th stage. Suppose n is the number of pipeline stages. The control logic of $lock\_r_k$ for resource conflict is represented as follows:

$$\begin{aligned} lock\_r_k = & \bigvee_{r \in R} \Big(\bigvee_{k < j \leq n} \Big(\bigvee_{i_j \in V_{r,j}} (inst_j = i_j) \cdot valid_j \Big) \cdot \\ & \Big(\bigvee_{i_k \in V_{r,k}} (inst_k = i_k) \cdot valid_k \Big)\Big) \end{aligned} \tag{4.4}$$

Equation (4.4) means that $lock_{k,r}$ holds if and only if at least one resource r is accessed from the k-stage and from at least one stage j where $k < j \leq n$.

Control signals for conflicted resources are generated from multiple stage controllers. Suppose $ctrl_r$ is a control signal for resource r and $ctrl_{r,k}$ is a control signal generated by stage controller of the k-th stage. The control signal is selected as follows:

$$
\begin{aligned}
ctrl_r &= \bigvee_{1\leq k\leq n} ctrl_{r,k} \cdot sel_{r,k} && (4.5)\\
sel_{r,k} &= ((\bigvee_{i_k \in V_{r,k}} (inst_k = i_k) \cdot valid_k) \cdot \overline{\bigvee_{k<j\leq n} \bigvee_{i_j \in V_{r,j}} ((inst_j = i_j) \cdot valid_j)}\ ) && (4.6)
\end{aligned}
$$

Equation (4.6) means that control signal $ctrl_{r,k}$ from the k-th stage controller is selected when $sel_{r,k} = true$. $sel_{r,k}$ becomes $true$ when resource r is not accessed from any stage j ($k < j \leq n$) and is accessed from stage k. Figure 4.9 shows an block diagram of interlock signal generation logic represented in Equation (4.4) and control signal selection represented in Equation (4.6).

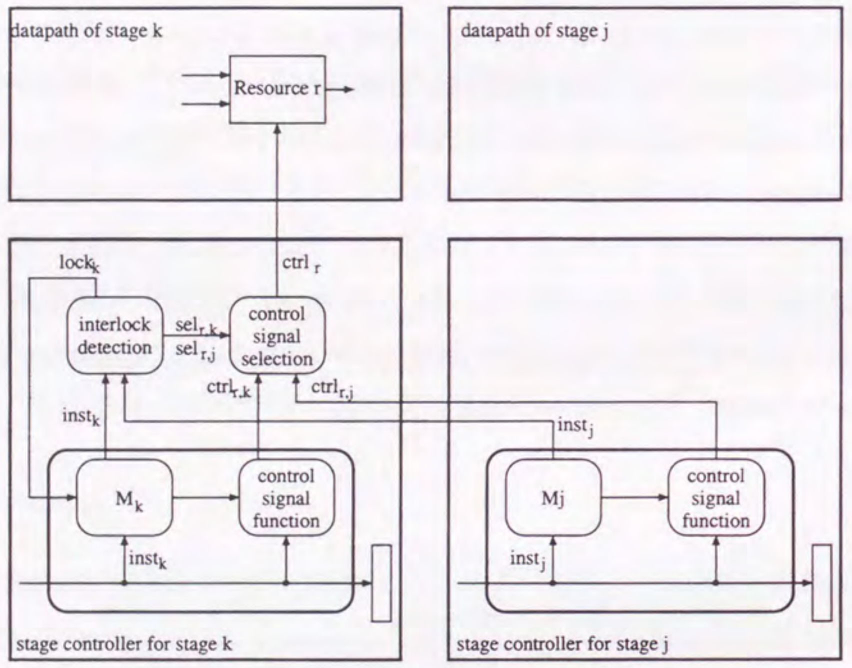


Figure 4.9: Example of Control Signal Selection for Conflicted Resource.

Pipeline Flush

Branch control is based on a predict-not-taken policy and delayed branch. In PEAS-III system, the number of delayed branch slots d is parameterized. The processor

executes succeeding d instructions whether branch is taken or not, and flushes the pipeline by nullifying other fetched instructions. When $d = 0$, the architecture of the processor is pure predict-not-taken architecture. When branch is taken at stage b, the controller of stage k ($1 < k \leq b - d$) nullifies transferred instruction and makes its state "no-operation" at the next clock cycle.

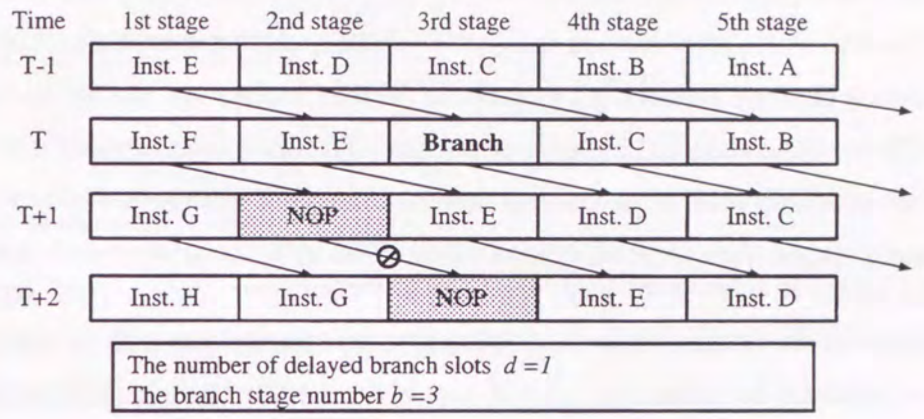


Figure 4.10: Example of Branch.

In the example shown in Fig. 4.10, the branch stage b is the third stage and the number of delayed branch slots d is one. In this example, branch is taken at time T and instruction E that is succeeding to the branch instruction D is executed continuously and the instruction F that is succeeding to instruction E is canceled by stage controller in the second stage at time $T + 1$.

The function $cancel(k)$ is as follows:

$$
cancel(k) \;=\; \begin{cases} true & \text{when } (1 < k \leq b - d) \\ false & \text{otherwise} \end{cases} \qquad (4.7)
$$

Suppose $Br = \{(exp, inst) \mid exp \in Exp, inst \in I\}$ is a set of conditional expression exp and instruction $inst$ pairs, which represent branch condition. The pair $(exp, inst) \in Br$ represents that branch is taken when executing instruction in the b-th stage is $inst$ and conditional expression exp holds. The logic of control signal $branch$ is represented as follows:

$$
branch \;=\; valid_b \cdot (\bigvee_{(exp,inst)\in Br} (inst_b = inst) \cdot exp) \qquad (4.8)
$$

Limitations of the proposed branch control method are as follows:

- Branch stage b must be unique.
- Instructions that change the statuses of resources such as register write and so on, in the k-th $(1 \leq k < b - d)$ stage should not be scheduled within $d+1$ to b slot after branch instruction. If these instructions are scheduled within $d+1$ to b slot after branch instruction, those instructions change the statuses before branch. Restoring mechanisms such as buffers are needed to cancel the effects of the canceled instruction completely. Since the proposed method does not synthesis such a mechanism, instructions that change machine statuses in early stages have to be scheduled within $d+1$ to b slot after branch instruction.
- Instruction that executes a multi-cycle operation in the j-th $(b - d \leq j < b)$ stage must be scheduled after d instructions from branch. When the multi-cycle instruction is scheduled within d instructions from branch instruction, some stages becomes empty between branch stage and stage which includes multi-cycle operations. The empty stages push out instructions in the delayed branch slots. Pushed out instructions are flushed by the controller.

4.4.3 Interrupt Controller

The interrupt controller suspends instruction fetch and executes described interrupt operations. The interrupt controller consists of the following finite state machine M_{intr} and control signals generator.

$$M_{intr} = (q_{intr}, I_{intr}, O_{intr}, \delta_{intr}, \rho_{intr}).$$

Each item of M_{intr} is defined as follows:

status variable: $q_{intr} \in \{intr, exe, wait\}$

input signals: $I_{intr} \triangleq \{interrupt, restart, complete\}$

output signals: $O_{intr} \triangleq \{go_0, int\}$

next-state function:

$$\delta_{intr}(q_{intr}, interrupt, restart, complete) \triangleq \begin{cases} intr & \text{when } (q_{intr} = wait) \cdot complete \\ exe & \text{when } (q_{intr} = intr) \cdot restart \\ wait & \text{when } (q_{intr} = exe) \cdot interrupt \\ q_{intr} & \text{otherwise} \end{cases}$$

output functions: $\rho_{intr} \triangleq \{\rho_{valid_0}, \rho_{int}\}$

$$\begin{aligned} \rho_{valid_0}(q_{intr}) &\triangleq (q_{intr} = exe) \\ \rho_{int}(q_{intr}) &\triangleq (q_{intr} = intr) \end{aligned}$$

initial status: $intr$

States “intr,” “exe” and “wait,” of q_{intr} are execution state of interrupts, execution state of instructions and waiting state for completion of all already fetched instructions, respectively. The initial state of q_{intr} is “intr,” because the processor has to begin with reset interrupt.

Input signal *interrupt* indicates the processor receives an interrupt. Input signal *complete* signal indicates execution of all fetched instructions is competed. *restart* signal indicates interrupt handling is completed and instruction fetch can be started. When an external interrupt occurs, the state of the controller changes the state from “exe” to “wait.” Then, the controller suspends instruction fetch by forcing the go_0 to $false$. It makes the state of the first stage “no-operation.” After all fetched instructions are completed, the states of all stages become “no-operation.” Then, the state of the controller becomes “intr.” An equation below is an control logic of complete signal.

$$complete = \overline{\bigvee_{1 \leq k \leq n} valid_n} \tag{4.9}$$

The controller begins to execute interrupt operations described in micro-operation description of interrupts. When the interrupt is completed, the state of the controller becomes “exe” and the output signal go_0 becomes $true$ to execute the first stage of the pipeline and to restart instruction fetch.

The following items of interrupt controller that depend on processor specification description and have to be synthesized.

1. logic of *restart*,

   Suppose *Intr* is a set of defined interrupts, s_i is a defined execution cycle count of interrupt i and Cnt is a counter which counts execution steps from the status variable q_{intr} becomes *intr*. The control logic for signal *restart* is represented as follows.

$$restart = \bigvee_{i \in Intr} (s_i > Cnt) \qquad (4.10)$$

2. logic of *interrupt*,

$$interrupt = \bigvee_{i \in Intr} (\text{specified condition of interrupt } i) \qquad (4.11)$$

3. and datapath control signal generator.

Chapter 5

Processor Synthesis

In this chapter the processor synthesis method is explained. The processor synthesis method consists of two major parts: datapath synthesis and controller synthesis. In this chapter datapath synthesis method is described first, and then controller synthesis method is described.

5.1 Datapath Synthesis

In datapath synthesis, data-flow graph is generated from micro-operation descriptions of instructions and interrupts at first. Then, techniques in high-level synthesis area [41] are utilized for datapath synthesis. Since the designer performs micro-operation scheduling to the pipeline stages and resource allocations in micro-operation descriptions, interconnection generation and pipeline register insertion are performed in datapath synthesis.

Figure 5.1 shows the datapath synthesis flow. Data-flow graphs (DFGs) of instructions and interrupts are generated from micro-operation descriptions (MODs). Then, DFGs of instructions are merged together to get required data-flow and condition of it. DFGs of interrupts are also merged together. For the resolution of signal conflicts, selectors are inserted to the both merged DFGs of instructions and interrupts. For the pipeline execution, pipeline registers are inserted to the DFGs of instructions. DFGs of instructions and interrupts are merged and signal conflicts are resolved. Then, the DFG that represents the datapath of designed processor is synthesized. Each generation step is described in the following sections in detail.

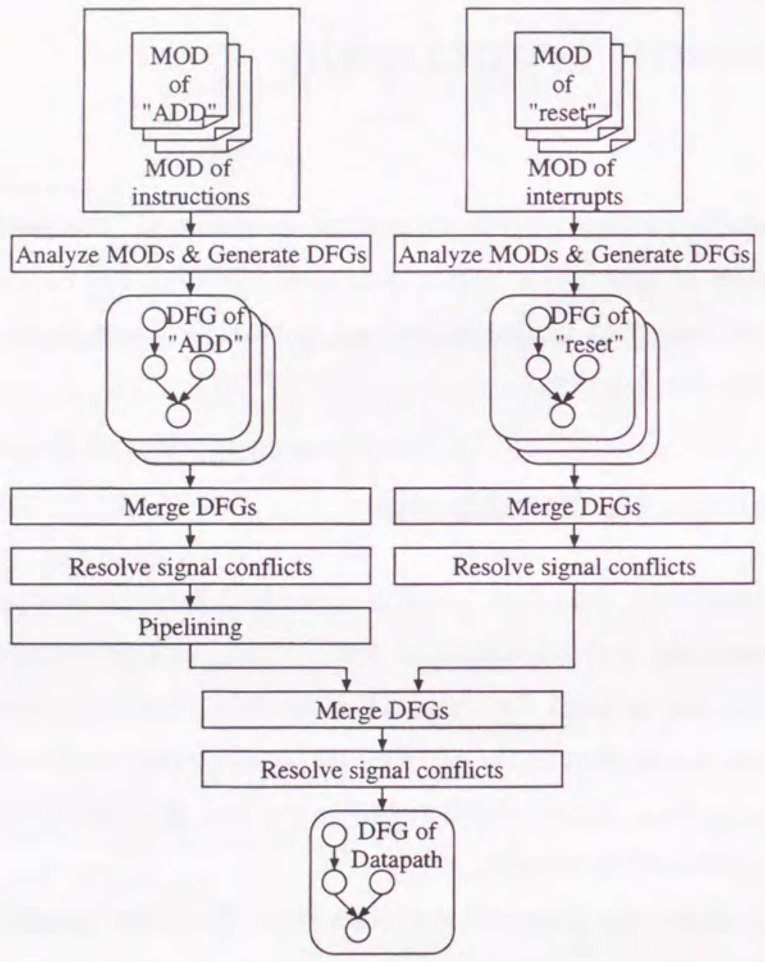


Figure 5.1: Datapath Synthesis Flow.

5.1.1 DFG generation

By analyzing a micro-operation description of each instruction *inst*, a data-flow graph is generated. The data-flow graph is represented by $G_{inst} = (R_{inst}, V_{inst}, E_{inst})$ where R_{inst} is a set of resources, V_{inst} is a set of all resource ports, and E_{inst} is a set of connections between ports of the resources. $(s, d) \in E_{inst}$ represents data transfer from the port $s \in V_{inst}$ to the port $d \in V_{inst}$ which is specified by a micro-operation description. $cond_{e,inst}$ represents a conditional expression for the data transfer represented by $e \in E_{inst}$ for instruction *inst*. If the data transfer is written in an if-statement of MOD, the conditional expression *exp* of the if-statement is extracted to $cond_{e,inst}$. If the data transfer is not written in if-statement, $cond_{e,inst}$ of the transfer e becomes '1'.

adder.add		
input0	:	a
input1	:	b
output0	:	$result$
control	:	$cin \leftarrow 0$

register.read		
output0	:	q

register.write		
input0	:	d
control	:	$enb \leftarrow 1$

Figure 5.2: Interface Information for Resources.

To get input and output ports for resource operations described in the MOD, interface information of resources is used. This information consists of correspondence of input/output arguments of the resource operation to port names. The information also includes required control signals to execute the operation. This information is registered for each model in FHM-DB. Example of registered interfaces for an adder and a register are shown in Fig. 5.2. In Fig. 5.2, the first argument of operation "add" is connected to adder's input port "a" and the second is to "b." The operation result is output from port "result." The controller have to provide control signal '0' to port "cin" to execute "a+b."

An example of the extraction of connections is shown in Fig. 5.3. "RZ," "RX" and "RY" in Fig. 5.3 denote registers and "ADD0" denotes an adder. From the interface information shown in Fig. 5.2, connections e_0, e_1, e_2 are extracted. Where e_0, e_1 and e_2 denote the data transfer from port q of resource "RX" to port a of

micro-operation :
"`RZ := ADD0.add(RX,RY);`"
connections :
$e_0 = (RX.q, ADD0.a)$
$e_1 = (RY.q, ADD0.b)$
$e_2 = (ADD0.result, RZ.d)$

Figure 5.3: Connection Extraction.

resource "ADD0," from port q of resource "RY" to port b of resource "ADD0," and from port *result* of resource "ADD0" to port d of resource "RZ," respectively.

5.1.2 Basic Datapath Synthesis

After the analysis of micro-operation, the data-flow graphs of instructions are merged into a data-flow graph $G = (R, V, E)$. It represents a basic datapath of the processor.

$$R = \bigcup_{inst \in I} R_{inst} \tag{5.1}$$

$$V = \bigcup_{inst \in I} V_{inst} \tag{5.2}$$

$$E = \bigcup_{inst \in I} E_{inst} \tag{5.3}$$

where I is a set of all instructions. $Cond_e$ for each data transfer $e \in E$ is determined as follows:

$$Cond_e = \{(cond_{e,inst}, inst) \mid inst \in I\}. \tag{5.4}$$

$(exp, inst) \in Cond_e$ denotes that the data transfer $e \in E$ is executed when executing instruction is *inst* and condition *exp* holds.

5.1.3 Signal Conflicts Resolution

When the same destination port d is shared by multiple connections in E, input signals for port d must be conflict. This section presents a selector insertion procedure, which resolves input signal conflicts. In this section, basic selector insertion algorithm is introduced first, and then improvement of the algorithm is described.

Suppose that $stage_{src}(e)$ is a stage number where the port s to which data transfer $e = (s, d)$ outputs data, $stage_{dst}(e)$ is a stage number where port d inputs data, and $width(p)$ is bit width of port p. Instructions can be executed correctly if selectors are inserted at any stage from $stage_{src}(e)$ to $stage_{dst}(e)$. For a reduction of pipeline registers, selectors are inserted at each stage from $stage_{src}(e)$ to $stage_{dst}(e)$. Furthermore, a destination port d inputs data from different ports in multiple stage, some selectors are inserted for each stage $stage_{dst}(e)$ to resolve signal conflicts in a stage, first. Then, a selector is inserted to resolve inter-stage signal conflicts.

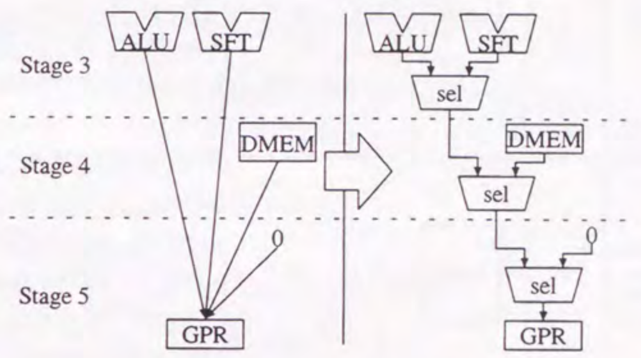


Figure 5.4: Example of Selector Insertion.

In a selector insertion example shown in Fig. 5.4, operation results of ALU, SFT and DMEM are selected by selectors "sel" in the third, the fourth and the fifth stage, respectively. Because selectors are inserted in each stage, data transfers over pipeline stage boundary are reduced. Another example shown in Fig. 5.5 is a case of signal conflict over stages. The example is non-harvard architecture and memory access unit "MEM" is accessed from both the first stage and the fourth stage. Firstly, signal conflict in the fourth stage is resolved and then signal conflict between the first stage and the fourth stage, that is data transfer from PC and from inserted selector, is resolved.

Outlines of selector insertion procedure are shown in Fig. 5.6 and Fig. 5.7. Fig. 5.6 shows an intra-stage signal conflict resolution and Fig. 5.7 shows an inter-stage signal conflict resolution. For each destination port d, a set X_d of stage numbers in which stage the port d receives data. For each member $j \in X_d$, selectors

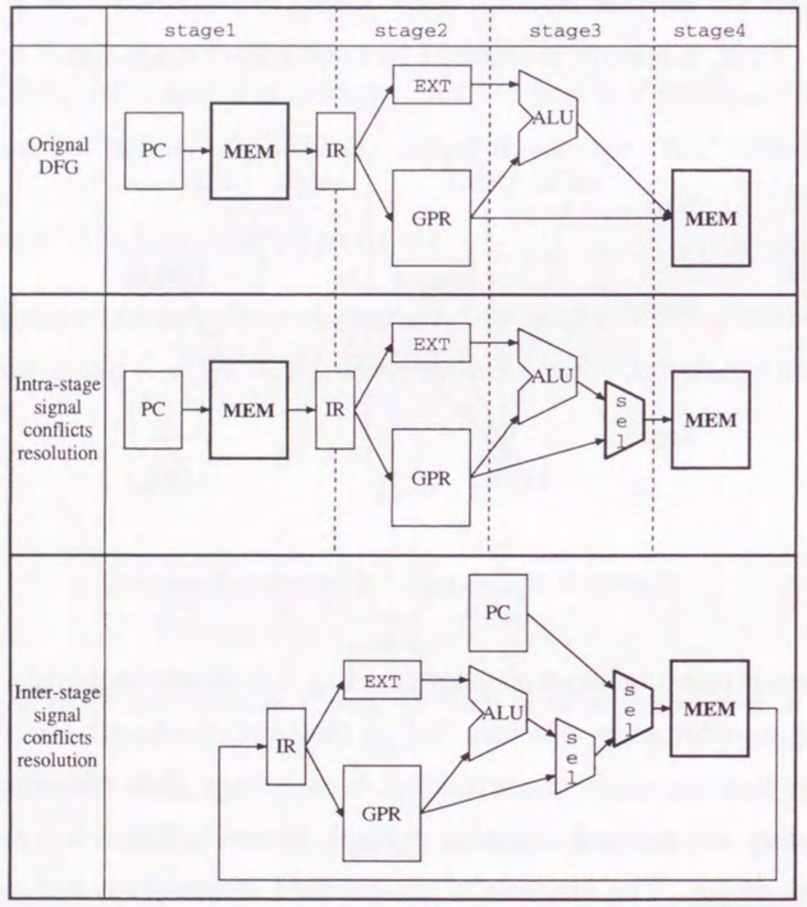


Figure 5.5: Example of Selector Insertion for Inter-stage Signal Conflicts.

```
input:   G = (R, V, E)
output:  G = (R, V, E)

foreach($d \in V$) loop
  $X_d := \{stage_{dst}(e) \mid e = (s,d) \in E\}$
  foreach($j \in X_d$) loop
    $E_{d,j} := \{e \mid e = (s,d) \in E, stage_{src}(e) = j\}$
    $min := minimum(\{stage_{src}(e) \mid e \in E_{d,j}\})$
    for $k := min$ to $j$ loop
      $E_{d,j,k} := \{e \mid e \in E_{d,j}, stage_{src}(e) \le k\}$
      if($|E_{d,j,k}| > 1$) then
        $insert\_selector(|E_{d,j,k}|, width(d))$
        $i := 0$
        $e_{out} := (s_{sel}, d)$
        $stage_{src}(e_{out}) := k$
        $stage_{dst}(e_{out}) := j$
        foreach($e' = (s', d) \in E_{d,j,k}$) loop
          $e_i := (s', d_{sel_i})$
          $Cond_{e_i} := Cond_{e'}$
          $stage_{src}(e_i) := stage_{src}(e')$
          $stage_{dst}(e_i) := k$
          $Cond_{e_{out}} := Cond_{e_{out}} \cup Cond_{e'}$
          $c_i := (p_{sel}, v_{sel_i})$
          $Cond_c := Cond_{e'}$
          $stage(c_i) := k$
          $C := C \cup \{c_i\}$
          $E := E \cup \{e_i\} - \{e'\}$
          $i := i + 1$
        end loop
        $E := E \cup \{e_{out}\}$
      end if
    end loop
  end loop
end loop
```

Figure 5.6: Selector Insertion Procedure. $insert\_selector(x, y)$ is a function to insert x inputs and y bit selector.

to resolve signal conflict in the j-th stage are inserted. $E_{d,j}$ is a set of data transfers that send data to the port d in the j-th stage is calculated. Then, the minimum stage number min of data output stage of all data transfer $e \in E_{d,j}$ is searched.

Selectors are inserted at each stage k from the minimum stage min to the j. A set $E_{d,j,k}$ is calculated. For all $e \in E_{d,j,k}$, the output stage of e is less than k is calculated. $E_{d,j,k}$ is a set of data transfer from the k-th stage or before the k-th stage. When the number of data transfers in $E_{d,j,k}$ is more than one, a selector is instantiated and inserted in the k-th stage. A feature of the selector used here is as follows: Input ports count is equal to the number of data transfers $E_{d,j,k}$ and the bit width is equal to the bit width of input data for port d.

With the selector insertion, the data transfer $e \in E_{d,j,k}$ should be modified. Each data transfer from $e' = (s', d) \in E_{d,j,k}$ is deleted from the connection set E. A new data transfer $e_i = (s', d_{sel_i})$ is added to E. e_i is a data transfer from the port s' to the i-th input port d_{sel_i} of the selector. The condition of e_i is equal to the condition of the deleted data transfer (s', d). The data input stage number for e_i is equal to k and output stage for e_i is equal to that of deleted one. The control signal value $c_i = (p_{sel}, v_{sel_i})$ is added to C. p_{sel} is control input port of the selector and v_{sel_i} is a value of selecting the i-th input. The condition $Cond_{c_i}$ is equal to the condition $Cond_{e_i}$. Addition of selector control signal is described in Section 5.2.1.

In addition, e_{out} that is a data transfer from selector output port s_{sel} to the port d is added to E. The data output stage number is equal to k and input stage is equal to j. The condition of data transfer using (s_{sel}, d) is a conjunction of conditions of connections of $E_{d,j,k}$.

After intra-stage signal conflicts are resolved, selector insertion for inter-stage signal conflicts resolution is executed. The procedure of inter-stage signal conflicts resolution is shown in Fig. 5.7. If the number of stages in X_d is more than one, a selector is inserted over stages. Feature of the selector used here is as follows: Input ports count is equal to the number of stages in X_d and the bit width is equal to the bit width of input data for port d. Each data transfer $e_{d,j} = (s_{d,j}, d)$ in the j-th stage is deleted from the connection set E. A new data transfer $e_i = (s_{d,j}, d_{sel_i})$ is added to E. e_i represents the data transfer from the port $s_{d,j}$ to the i-th input

```
input:   G = (R, V, E)
output:  G = (R, V, E)

foreach($d \in V$) loop
   $X_d := \{stage_{dst}(e) \mid e = (s,d) \in E\}$
   if($|X_d| > 1$) then
      $insert\_selector(|X_d|, width(d))$
      $i := 0$
      $e_{out} := (s_{sel}, d)$
      foreach($j \in X_d$) loop
         $e_{d,j} := (s_{d,j}, d) \in E \cap stage_{dst}(e_{d,j}) = j$
         $e_i := (s_{d,j}, d_{sel_i})$
         $Cond_{e_i} := Cond_{e_{d,j}}$
         $stage_{src}(e_i) := stage_{dst}(e_i) := j$
         $Cond_{e_{out}} := Cond_{e_{out}} \cup Cond_{e_{d,j}}$
         $c_i := (p_{sel}, v_{sel_i})$
         $stage(c_i) := j$
         $Cond_c := Cond_{e_{d,j}}$
         $C := C \cup \{c_i\}$
         $E := E \cup \{e_i\} - \{e_{d,j}\}$
         $i := i + 1$
      end loop
      $E := E \cup \{e_{out}\}$
   end if
end loop
```

Figure 5.7: Selector Insertion Procedure for Inter-stage Signal Conflicts.

port d_{sel_i} of the selector. The condition of e_i is equal to the condition of the deleted data transfer $e_{d,j}$. The data input stage number and the data output stage number for e_i are equal to j. Then, the control signal value $c_i = (p_{sel}, v_{sel_i})$ is added to C. p_{sel} is control input port of the selector and v_{sel_i} is a value of selecting j-th input. The condition $Cond_{c_i}$ is equal to the condition $Cond_{e_i}$. Addition of selector control signal is described in Section 5.2.1.

Improved selector insertion algorithm

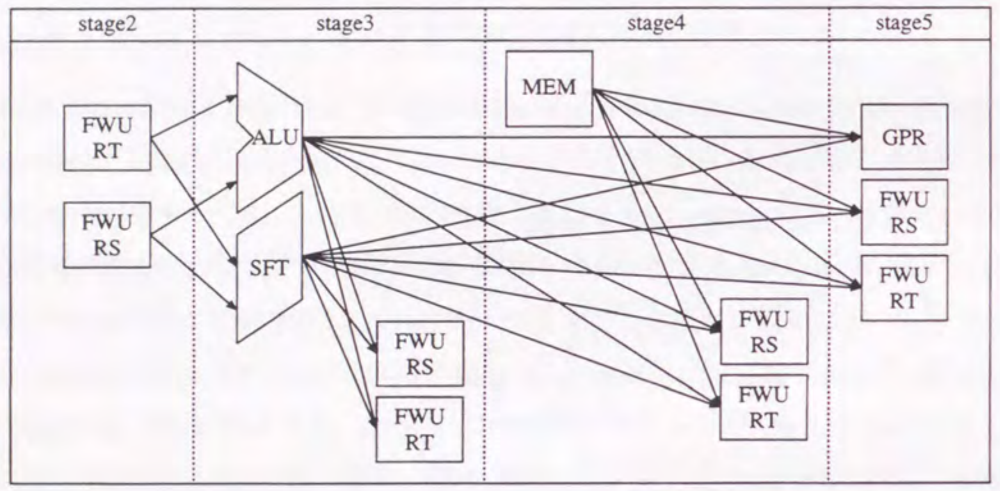


Figure 5.8: Original DFG for Selector Sharing Example.

The algorithm shown in Fig. 5.6 and Fig. 5.7 inserts wasteful selectors. In an example data-flow graph shown in Fig. 5.8, some input ports receive same sets of input signals. Resource "FWURS" and "FWURT" are the data forwarding units. They receives data from "ALU" and "SFT" in the third stage, from "ALU," "SFT" and "DMEM" in the fourth and fifth stage, respectively. Because different input ports of "FWURS," "FWURT" are used in each pipeline stage, there are no inter-stage signal conflicts. The conditions of data transfers from each functional unit to forwarding units and general-purpose registers (GPR) are the same. Figure 5.9 illustrates selector insertion results. The selectors in the third stage always output the same results and the selectors in the fourth stages, too. Therefore, improvement of the selector insertion algorithm is required to reduce selectors. Before line nine

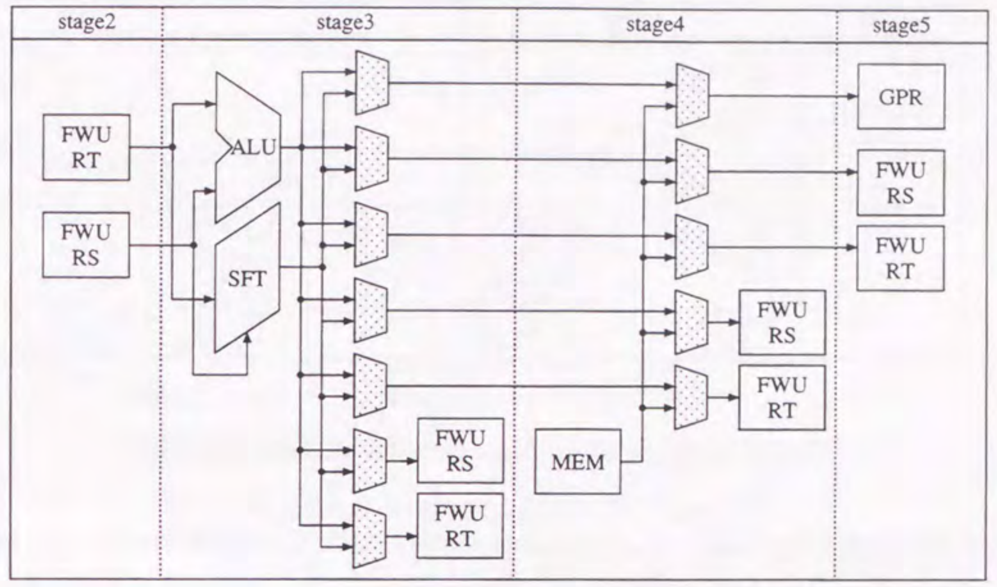


Figure 5.9: Selector Insertion Result without Sharing.

of Fig. 5.6, procedure to search a selector r_{sel} that have common input signals and select conditions. If the selector r_{sel} exists, the data transfer sets $E_{d,j,k}$ are deleted from E and data transfer from the output port of selector r_{sel} to the port d is added.

Suppose $E_{r_{sel}}$ is a set of edges which represent data transfers to the selector input ports, the condition to use the inserted selector r_{sel} is described as follows:

$$\forall e = (s, d) \in E_{d,j,k} \ \exists \ e_{sel} = (s, d_{sel}) \in E_{r_{sel}} \ Cond_{e_{sel}} = Cond_e \qquad (5.5)$$

The condition which is shown in Equation 5.5 becomes true if and only if data transfer e_{sel} exists for all data transfer e of $E_{d,j,k}$. e and e_{sel} have the same input port and condition of data transfer.

Figure 5.10 shows a selector insertion result of improved algorithm. Wasteful selectors are reduced to one for each stage.

5.1.4 Pipelining

When data are transferred over pipeline stage boundary, a pipeline register is required to transfer operation results to the next stage. A data transfer $e \in E$

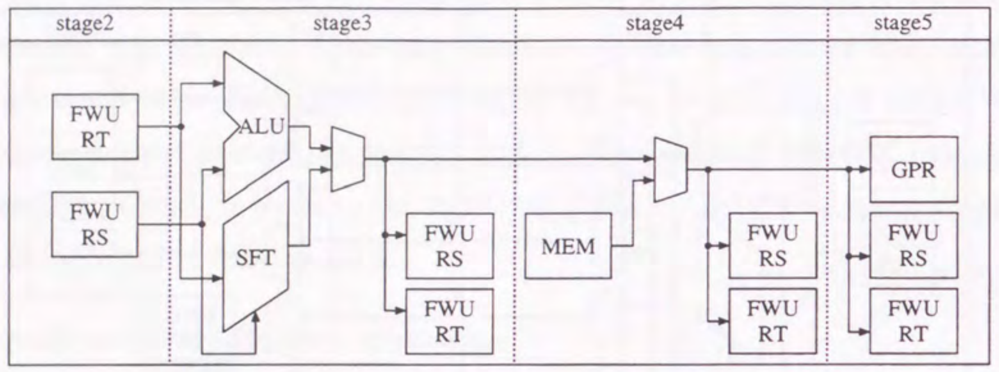


Figure 5.10: Selector Insertion Result with Sharing.

which satisfies $stage_{src}(e) < stage_{dst}(e)$ means data are transferred over pipeline stage boundary, so that pipeline registers are required at each stage boundary from $stage_{src}(e)$ to $stage_{dst}(e)$. In a pipeline register insertion example in Fig. 5.11, pipeline registers are inserted to each pipeline stage boundary.

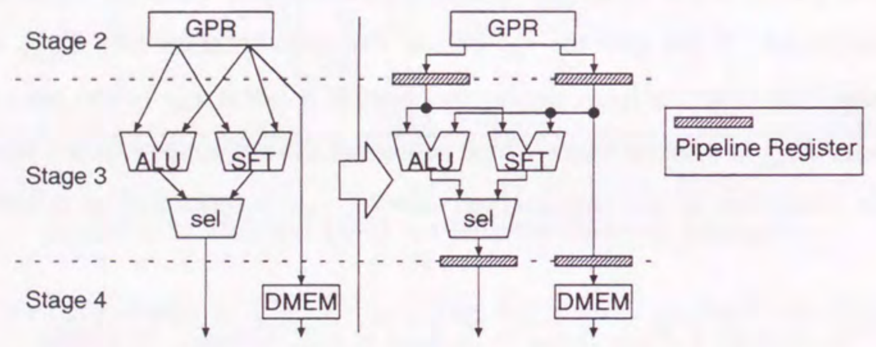


Figure 5.11: Example of Pipeline Register Insertion.

Pipelining procedure is illustrated in Fig. 5.12. First of all, a set of data transfer E_{reg} that is a subset of E is calculated. For all data transfer $e \in E_{reg}$ satisfies $stage_{src}(e) < stage_{dst}(e)$. When the number of data transfers in E_{reg} is not zero, there are some data transfers over pipeline stage boundary. One data transfer $e = (s, d)$ of the E_{reg} is selected arbitrarily, and $width(s)$-bit pipeline register is inserted between the stage $stage_{src}(e)$ and the stage $stage_{src}(e)+1$. According to the register insertion, connection e_{in} from port s to d_{reg} is added to E where d_{reg} and s_{reg} are

```
input:   G = (R, V, E)
output:  G = (R, V, E)

1  $E_{reg} := \{e \mid e \in E, stage_{src}(e) > stage_{dst}(e)\}$;
2  if $E_{reg} \neq \phi$ then
3     $e := (s, d) \in E_{reg}$;
4     instantiate $width(s)$ bit register;
5     $e_{in} := (s, d_{reg})$;
6     $stage_{src}(e_{in}) := stage_{src}(e)$;
7     $stage_{dst}(e_{in}) := stage_{src}(e)$;
9     $E := E \cup \{e_{in}\}$;
8     $E' := \{e' \mid e' = (s, d') \in E_{reg}, stage_{src}(e) = stage_{src}(e')\}$;
10    foreach $e' := (s, d') \in E'$ loop
11       $e_{out} := (s_{reg}, d')$;
12       $stage_{src}(e_{out}) := stage_{src}(e) + 1$;
13       $stage_{dst}(e_{out}) := stage_{dst}(e')$;
14       $E := E \cup \{e_{out}\}$;
15       $E := E - e'$;
16    end loop;
17    goto 1;
18 end if;
```

Figure 5.12: Pipelining Procedure.

an input port and an output port of inserted pipeline register, respectively. Because the register is written in the stage $stage_{src}(e)$, $stage_{dst}(e_{in})$ is equal to $stage_{src}(e)$. For all data transfer $e' = (s, d')$ which transfers data from port s and satisfies $stage_{src}(e') = stage_{src}(e)$, connection $e_{out} = (s_{reg}, d')$ is added to E. Because the register is read at one stage after it is written, stage number $stage_{src}(e_{out})$ becomes n$stage_{dst}(e_{in}) + 1$. An original connection $e' = (s, d')$ is deleted from E. Until no data transfer, which transfers data over pipeline stage boundary, exists in E, pipeline insertion procedure is repeated from line one.

5.2 Controller Synthesis

The controller synthesis is based on the controller model described in Section 4.4. The control logics that depend on processor specification are synthesized from processor specifications, mainly from micro-operation descriptions.

The controller synthesis procedure consists of six parts:

1. Control Signal Extraction from micro-operation descriptions,
2. Interlock Condition Extraction,
3. Branch Condition Extraction,
4. Instruction Decoder Synthesis,
5. Stage Controller Synthesis,
6. and Interrupt Controller Synthesis.

Each synthesis procedures are described in the following sections in detail.

5.2.1 Control Signal Extraction

Control signals for declared resources to execute described micro-operation are extracted in this step. By analyzing a micro-operation description of each instruction $inst$, a set of control signal assignments C_{inst} is generated. A control signal assignment $c \in C_{inst}$ is a tuple (p, v) where p denotes a control input port of a resource and

v denotes the value to the port p. An expression $cond_{c,inst}$ represents a condition for the control signal assignment of c for instruction $inst$. The extraction procedure of control signal c from a micro-operation description is explained using example shown in Fig. 5.13 as follows. Control signal values for cin port of "ADD0" and enb port of register "RZ" are induced by Fig. 5.2. As a consequence, control signal assignments c_0 and c_1 are extracted.

micro-operation :
    "`RZ := ADD0.add(RX,RY);`"
control signals:
    $c_0 = (ADD0.cin, 0)$
    $c_1 = (RZ.enb, 1)$

Figure 5.13: Control signal extraction.

After the analysis of micro-operations, sets of control signals are merged into C. C is obtained as follows:

$$C = \bigcup_{inst \in I} C_{inst} \tag{5.6}$$

where I is a set of all instructions. $Cond_c$ is determined as follows:

$$Cond_c = \{(cond_{c,inst}, inst) \mid inst \in I\}. \tag{5.7}$$

$(exp, inst) \in Cond_{(p,v)}$ denotes that the value v is assigned to port p when executing instruction is $inst$ and condition exp is satisfied. The stage controller assigns the value v to p when all the following conditions are satisfied. (1) The status variable q_k is '1', (2) the executing instruction of the stage is $inst$, and (3) the expression exp holds. The stage controller assigns control signal value for "no-operation" when one of the conditions above does not hold. "no-operation" value means the resource of the port p do not change its status during the port p is receiving the value v_0. For example, the "no-operation" value for register write enable port is its negative value.

When the selectors are inserted in datapath synthesis, control signals for selectors are also added to C described in Fig. 5.6 and Fig. 5.7.

5.2.2 Interlock Condition Extraction

To synthesis pipeline interlock control signal $lock_k$, conditions of multi-cycle operations and resource conflicts are extracted. In this section, condition extraction of multi-cycle operations is described first and then that of resource conflicts is described.

Pipeline interlock logic for multi-cycle operation is synthesized from the Equation (4.1) in Section 4.4.1. Execution conditions of resource operations $U_{k,r,op} = \{(exp, inst) \mid exp \in Exp, inst \in I\}$ are extracted from the micro-operation description of instructions where exp is a conditional expression and $inst$ indicates an execution instruction. If operation op of resource r occurs in the micro-operation description of instruction $inst$ in the k-th stage, execution condition $(exp, inst)$ is added to $U_{k,r,op}$. Completion flag $fin_{op_m} = (p, v)$ is defined for each operation op in FHM-DB. $fin_{op_m} = (p, v)$ denotes that the output signal of the port p becomes v after the operation op_m is finished. From extracted execution conditions $U_{k,r,op}$ and received completion flag expression fin_{op_m} from FHM-DB, interlock control logic for multi-cycle operation $multi\_lock_k$ is synthesized.

Suppose Fin is a set of completion flags of all described multi-cycle operation.

$$Fin = \{fin_{op_m}\} \tag{5.8}$$

OP_{fin} is a set of multi-cycle operations, which have the same completion flag fin.

$$OP_{fin} = \{op \mid fin_{op_m} = fin\} \tag{5.9}$$

Then $Exp_{k,fin}$ and $I_{k,fin,exp}$ are calculated by the following equations.

$$Exp_{k,fin} = \{exp \mid (exp, inst) \in U_{k,r,op}, op \in OP_{fin}\} \tag{5.10}$$

$$I_{k,fin,exp} = \{inst \mid (exp, inst) \in U_{k,r,op}, op \in OP_{fin}\} \tag{5.11}$$

$Exp_{k,fin}$ is a set of execution condition of operation $op \in OP_{fin}$. $I_{k,fin,exp}$ is a set of instructions, which execute multi-cycle operation $op \in OP_{fin}$ when exp holds.

Suppose $output(p)$ indicates output value of port p. Using Equation (5.8), (5.10) and (5.11) $multi\_lock_k$ is represented as follows:

$$\begin{aligned} & multi\_lock_k \\ & = \bigvee_{fin=(p,v)\in Fin} \Big(\bigvee_{exp \in Exp_{k,fin}} ((inst_k \in I_{k,fin,exp}) \cdot exp \cdot valid_k) \cdot (output(p) \neq v)\Big) \end{aligned} \tag{5.12}$$

Signal $multi\_lock_k$ becomes '1' when at least one of the multi-cycle operations is not completed. The signal $multi\_lock_k$ becomes '0' when all the value of completion flag output port p becomes v.

Pipeline interlock detection logic for resource conflicts is synthesized from the Equation (4.4) in Section 4.4.1. A set $V_{r,k}$ is calculated from the set C, function $resource(p)$ and $stage(c)$. A function $resource(p)$ returns resource r of port p and $stage(c)$ returns the stage number in that stage control signal is assigned to port p.

Suppose $C_r = \{c \mid c = (p, v) \in C, resource(p) = r\}$, and $C_{r,k} = \{c \mid c \in C_r, stage(c) = k\}$. Using $C_{r,k}$, a set $V_{r,k}$ is calculated. From $V_{r,k}$ and Equation (4.4), lock signal for resource conflict $res\_conflict_k$ is synthesized. $V_{r,k}$ is also used to synthesis control signal selection for resource r.

$$V_{r,k} = \{inst \mid (inst, exp) \in Cond_{c_{r,k}}, c_{r,k} \in C_{r,k}\} \tag{5.13}$$

$$res\_conflict_k = \bigvee_{r\in R} \Big(\Big(\bigvee_{k<j\leq n} (inst_j \in V_{r,j}) \cdot valid_j\Big) \cdot (inst_k \in V_{r,k}) \cdot valid_k\Big) \tag{5.14}$$

5.2.3 Branch Condition Extraction

By analyzing a micro-operation description of branch instructions, branch stage number b and a condition $Br = \{(exp, inst) \mid exp \in Exp, inst \in I\}$ of branch are extracted. The conditional expression exp is a condition for the case that the program counter PC is written. Suppose Exp_{Br} is a set of conditional expression exp for branches and $I_{Br,exp}$ is a set of instructions, which execute branch when condition exp holds.

The set Exp_{Br} and $I_{Br,exp}$ are calculated as follows:

$$Exp_{Br} = \{exp \mid (exp, inst) \in Br\} \tag{5.15}$$

$$I_{Br,exp} = \{inst \mid (exp, inst) \in Br\} \tag{5.16}$$

From an Equation (4.8) in Section 4.4.2, control logic of *branch* is calculated as follows:

$$branch = valid_b \cdot (\bigvee_{exp \in Exp_{Br}} (inst_b \in I_{Br,exp}) \cdot exp) \tag{5.17}$$

5.2.4 Instruction Decoder Synthesis

Instruction decoder inputs instruction word and generates two types of signals based on the model described in Section 4.4.1. In this section, resource control signal generation is described first, and then instruction identification signal is described.

The control signals that are independent of datapath status signals are selected. Then the instruction decoder logics for selected control signals are generated. A set of instructions $I_{c,exp,k}$ which assigns the value v to control port p in the k-th stage when condition exp holds is calculated as follows:

$$I_{c,exp,k} = \{inst \mid c \in C, (exp, inst) \in Cond_c, stage(c) = k\} \tag{5.18}$$

$I_{c,1,k}$ is a set of instructions which assigns control signal represented by c independently of datapath status. A set of control signal assignment $C_{p,k}$ for the port p in the k-th stage is selected as follows:

$$C_{p,k} = \{c \mid (p, v) \in C, stage(c) = k\} \tag{5.19}$$

$f_{p,k}(inst)$ is an output function of instruction decoder which generates control signal for the port p in the k-th stage. The decoded result $f_{p,k}(inst)$ is sent to each pipeline stage step by step via pipeline register. Suppose $Z_{j,f_{p,k}(inst)}$ is a pipeline register for $f_{p,k}(inst)$ between the $(j-1)$-th stage to j-th stage and d is an instruction decode stage. The value of $Z_{k,f_{p,k}(inst)}$ is assigned to the port p in the k-th stage when k is less than instruction decode stage d. The decoded result $f_{p,k}$ is directly

assigned to the port p when k is equal to d. $f_{p,k}$ and $Z_{j,f_{p,k}(inst)}$ are represented as follows:

$$\begin{aligned}
f_{p,k}(inst) &= (\bigvee_{c=(p,v) \in C_{p,k}} v \cdot (inst \in I_{c,1,k})) + v_0 \cdot \bigwedge_{c \in C_{p,k}} (inst \notin I_{c,1,k}) && (5.20)\\
Z^+_{j+1,f_{p,k}(inst)} &= Z_{j,f_{p,k}(inst)} \cdot go_j + Z_{j+1,f_{p,k}(inst)} \cdot \overline{go_j} \quad (d < j < k) && (5.21)\\
Z^+_{d+1,f_{p,k}(inst)} &= f_{p,k}(inst). && (5.22)
\end{aligned}$$

Function $f_{p,k}(inst)$ returns the value v when fetched instruction $inst$ is a member of $I_{c,1,k}$. If fetched instruction $inst$ is not a member of $I_{c,1,k}$, $f_{p,k}(inst)$ returns "no-operation" value v_0.

The other type of instruction decode, the result indicates whether fetched instruction is a member of a certain set of instructions or not. It is used to generate the following control signals: interlock control signal $lock_k$, branch detection signal *branch* and resource control signal, which depends on datapath status. The decoded result of $m(I, inst)$ is also send to each pipeline stage step by step via pipeline register $Z_{j,m(I,inst)}$. Suppose $Z_{j,m(I,inst)}$ is a pipeline register for $m(I, inst)$ between the $(j-1)$-th stage and the j-th stage. Suppose '$inst$' is a fetched instruction and I is a set of instructions, function of the latter type instruction decode is described as follows:

$$\begin{aligned}
m(I, inst) &= \begin{cases} 1 & \text{when } inst \in I \\ 0 & \text{otherwise} \end{cases} && (5.23)\\
Z^+_{j+1,m(I,inst)} &= Z_{j,m(I,inst)} \cdot go_j + Z_{j+1,m(I,inst)} \cdot \overline{go_j} \quad (d < j < k) && (5.24)\\
Z^+_{d+1,m(I,inst)} &= m(I, inst). && (5.25)
\end{aligned}$$

The stage controllers input decoded result $S_{k,f_{p,k}(inst)}$ and $S_{k,m(I,inst)}$ that are shown in the following equations.

$$\begin{aligned}
S_{k,f_{p,k}(inst)} &= \begin{cases} Z_{k,f_{p,k}(inst)} & (d < k) \\ f_{p,k}(inst) & (d = k) \end{cases} && (5.26)\\
S_{k,m(I,inst)} &= \begin{cases} Z_{k,m(I,inst)} & (d < k) \\ m(I, inst) & (d = k) \\ 1 & (k < d) \end{cases} && (5.27)
\end{aligned}$$

The controller of the k-th stage uses output signal of pipeline register $Z_{k,*}$ when k is greater than d, which represents instruction decode stage number. The stage controller of instruction decode stage uses decoded result directly.

5.2.5 Stage Controller Synthesis

The stage controller, which is based on a model in Section 4.4.2, is described. The following items of stage controller depend on architectural level processor description and are synthesized, and items of finite state machine M_k are generated from the model:

1. interlock detection signal $lock_k$.

$$lock_k = multi\_lock_k + res\_conflict_k \tag{5.28}$$

   $lock_k$ is a logical sum of $multi\_lock_k$ and $res\_conflict_k$. $multi\_lock_k$ and $res\_conflict_k$ are defined in Equation (5.14) and (5.12). Using the result of instruction decode shown in Equation (5.23), $multi\_lock_k$ and $res\_conflict_k$ are represented as follows:

$$multi\_lock_k = \bigvee_{fin \in Fin} (\bigvee_{exp \in Exp_{k,fin}} S_{k,m(I_{k,fin,exp},inst)} \cdot exp \cdot valid_k) \cdot (p \neq v)) \tag{5.29}$$

$$res\_conflict_k = \bigvee_{r \in R} ((\bigvee_{k \leq j \leq n} S_{j,m(V_{r,j},inst)} \cdot valid_j) \cdot S_{k,m(V_{r,k},inst)} \cdot valid_k) \tag{5.30}$$

2. branch detection signal $branch$ that is defined as Equation (5.17). Using the result of instruction decodes shown in Equation (5.23), $branch$ is represented as follows:

$$branch = valid_b \cdot (\bigvee_{exp \in Exp_{Br}} S_{b,m(I_{Br,exp},inst)} \cdot exp) \tag{5.31}$$

3. function $cancel(k)$ that is generated from Equation (4.7) in Section 4.4.2 and extracted branch stage number b.

4. and control signal generation functions.

Control Signal Generation Functions

Control signal generation functions are classified into three types,

1. control signals for resources in the k-th stage,

2. control signals for multi-cycle operations,

3. control signals for the resources, which are accessed from multiple stages.

Control signal of the port p in the k-th stage is generated from $C_{p,k}$ shown in Equation (5.19) and (5.18) and $Exp_{c,k}$. $Exp_{c,k}$ is a set of expression exp for conditional control signal assignment of c.

$$Exp_{c,k} = \{exp \mid c \in C, (exp, i) \in Cond_c, stage(c) = k\} \tag{5.32}$$

Referring to Equations (5.19), (5.18) and (5.32), control signal $S_{p,k}$ for the port p in the k-th stage is represented as follows:

$$\begin{aligned} S_{p,k} = {} & (\bigvee_{c \in C_{p,k}} \bigvee_{\substack{exp \in Exp_{c,k} \\ exp \neq 1}} v \cdot exp \cdot S_{k,m(I_{c,exp,k},inst)} \cdot go_k) \\ & + S_{k,f_{p,k}(inst)} \cdot go_k \cdot (\bigwedge_{c \in C_{p,k}} \bigwedge_{\substack{exp \in Exp_{c,k} \\ exp \neq 1}} (\overline{exp} + \overline{S_{k,m(I_{c,exp,k},inst)}})) \\ & + \overline{go_k} \cdot v_0 \end{aligned} \tag{5.33}$$

Control signal $S_{p,k}$ becomes the "no-operation" value v_0 when the k-th stage is stalled ($go_k = 0$). Control signal $S_{p,k}$ becomes the value v if condition exp holds and executing instruction is a member of the set of instructions $I_{c,exp,k}$. If any condition exp of conditional signal assignment does not hold, the result of instruction decoder $f_{p,k}(inst)$, which is described in Equation (5.20) in Section 5.2.4, is assigned to the port p.

Control signals for multi-cycle operations and conflicted resources are discussed in Section 4.4.2.

Because control signal $S_{p,k}$ becomes active value of start signal when multi-cycle operation should be executed, control signal for multi-cycle operation $S_{p,k,start}$ is described as follows:

$$S_{p,k,start} = flag \cdot S_{p,k} + \overline{flag} \cdot \overline{v_{active}} \tag{5.34}$$

Using the result of instruction decoder, resource usage condition $V_{r,k}$, and control signal $S_{p,k}$ of port p in the k-th stage, control signal S_p for the port p which is accessed from multiple stages is described as follows:

$$S_p = \bigvee_{1 \leq k \leq n} S_{p,k} \cdot sel_{r,k} \tag{5.35}$$

$$sel_{r,k} = S_{k,m(V_{r,k},inst)} \cdot valid_k \cdot \overline{\bigvee_{k<j \leq n} S_{k,m(V_{j,k},inst)} \cdot valid_j} \tag{5.36}$$

5.2.6 Interrupt Controller Synthesis

From an interrupt definition, a state machine shown in Section 4.4.3 is synthesized. The conditions of state transition from "exe" to "wait" are logical sum of defined interrupt conditions. The synthesis method of data-path and control signal values to execute described interrupt operation is the same as that of instructions.

$$interrupt = \bigvee_{i \in I_{intr}} \text{condition of defined interrupt } i \tag{5.37}$$

$$restart = (Cnt \geq s_i) \tag{5.38}$$

$$\tag{5.39}$$

where

s_i	:	execution cycle count for interrupt$i$
Cnt	:	counter for interrupts

When one of the conditions for specified interrupt holds, the output of the signal *interrupt* becomes '1' and detects interrupts. The counter Cnt counts the number of execution cycles of interrupts.

If one of the interrupt conditions *condition* holds, the signal *restart* turns to '1' and detects interrupts. The counter S is used for interrupt control during the status variable keeps $q_{intr} = 0$.

Suppose C_{intr} is a set of control signals assignment for interrupt *intr*. Suppose *cycle*(c) is a function which returns in what stage control signal assignment c is executed in micro-operation description of interrupt *intr*. The control signal for interrupts are defined as follows.

$$S_{p,int} = \bigvee_{c=(p,v) \in C_{intr}} v \cdot exp \cdot (intr = int\_name) \cdot (Cnt = cycle(c)) + \overline{\bigvee_{c=(p,v) \in C_{intr}} v \cdot exp \cdot (intr = int\_name) \cdot (Cnt = cycle(c))} \cdot v_0 \tag{5.40}$$

If execution interrupt is *intr*, the number of steps since interrupt processing is started is equal to *cycle*(c) and condition *exp* holds, the control value v is assigned to port p.

Chapter 6

Experiments

In this chapter, the effectiveness of the proposed processor design method and synthesis method are evaluated through experiments.

6.1 Objective of the Experiments

In this chapter five kinds of experiments are described.

1. existing RISC processor to evaluate variety of instructions that can be designed and synthesized by the proposed method.
2. PEAS-I processor core to compare design quality of synthesized processor to that of manually designed one.
3. embedded RISC controller for comparison between conventional design method and proposed processor synthesis method in terms of design time and design quality.
4. pipeline depth tuning. It is aimed for evaluate modification time and effectiveness of an adjustment of the number of pipeline stages and operation re-scheduling to the pipeline stages
5. architectural design space exploration for FIR filter. It is aimed for evaluate the design time of new instruction specification and the range of explored design space.

To evaluate the effectiveness of the proposed processor synthesis method, processor synthesis time, design productivity and the quality of synthesized processor, and the largeness of explored design space are examined. Processor synthesis time is evaluated using prototyped processor synthesizer of the proposed synthesis method on Pentium III processor. Design productivity is evaluated in terms of design time and the amount of description. Design productivity is evaluated for both new processor design and architectural design space exploration. The quality of synthesized processor is evaluated in terms of area and clock frequency.

6.2 Basic RISC Processor

In the first experiment, the easiness of new processor design and its derivative processor design is explained. First, a MIPS R3000 [4] [42] compatible processor PEAS R3K was designed. Then, it was modified into DLX [43] for evaluation of the easiness of design in micro-operation level processor specification.

At the first step, a subset of MIPS R3000 instruction set was implemented. The number of implemented instructions is 52 out of 74 instructions of all instructions on MIPS R3000. Coprocessor instruction and interrupt instruction were not implemented in this experiment. Required time for design was about eight hours. Required time for synthesis was about two minutes.

Table 6.1: Results of Synthesis for PEAS R3K

component	#	Total Area (gates)	Frequency (MHz)
user specifiedresources	17	45759.22	157.48
registers	20	7064.67	769.23
selectors	10	2046.08	471.70
controller	1	2347.18	200.00
sum of the above	46	57217.17	157.48
processor	1	59818.34	125.63

using Design Compiler (0.5μm CMOS library)

The results of synthesis are summarized in Table 6.1. The column "#" denotes the number of components in the processor. "Total Area" indicates the component

area including wiring area. "Frequency" means the maximal frequency of the corresponding component. "User specified resources" are the resources that are explicitly declared by the designer. "Registers," "selectors," and "controller" are automatically introduced resources by the generator. "Sum of the above" means just the summary of all values above in the column. "Processor" is the synthesis result as a processor.

From these experimental results, it is confirmed that automatically generated parts does not so much affect area and performance of the processor. Area of the generated part is about 15% of the whole processor. Frequency of the introduced resources including the controller is relatively high, hence they do not include the critical path individually. The critical path of PEAS R3K was zero-flag generation by ALU and PC update. This path is synthesized from the micro-operation description in the third stage of some branch instructions like "Branch on Equal (BEQ)." Micro-operation description of BEQ is shown in Fig. 3.14 in Section 3.3.6.

Table 6.2: Results of Synthesis for PEAS DLX

component	#	Total Area (gates)	Frequency (MHz)
user specifiedresources	14	45758.01	157.48
registers	23	7545.82	769.23
selectors	15	1960.82	628.93
controller	1	1790.78	200.80
sum of the above	53	57055.39	157.48
processor	1	48469.03	116.28

using Design Compiler (0.5μm CMOS library)

At the second step in this experiment, a subset of DLX (called PEAS DLX) was implemented based on PEAS R3K. The number of implemented instructions is 51 out of all instructions 91. Similar to the case of PEAS R3K, coprocessor instruction and interrupt instruction were not implemented in this experiment. The reuse ratio for DLX design from the description of PEAS R3K is 59% since both architectures have many similar instructions. Required time for modification is about 3 hours. Table 6.2 shows a logic synthesis results of DLX.

The amount of descriptions for both PEAS R3K and PEAS DLX is shown in Table 6.3. The amount of description for micro-operation level processor specification is about less than one sixth of the case of the corresponding generated HDL description. It is clear that proposed processor synthesis method reduces the designer's load.

Table 6.3: Comparison of the Amount of Descriptions for PEAS R3K and PEAS DLX

	MOD	generated HDL description		
		datapath	ctrl	all
PEAS R3K	512	833	512	5804
PEAS DLX	498	824	490	5541

(unit: lines)

6.3 PEAS-I Processor Core

PEAS-I core is a processor generated by the PEAS-I system [10]. PEAS-I system can generate an optimal processor for a given application program from predefined instruction set. Predefined instruction set consists of a primitive instruction set and optional instructions. The primitive instruction set contains basic instructions that most processors have. Instructions in the primitive instruction set can be categorized into arithmetic instructions, data transfer instructions, and execution sequence control instructions. In this experiment, the existing design and new one designed with PEAS-III's micro-operation level processor specification are compared. First, a PEAS-I core from a primitive instruction set was designed with PEAS-III. The instruction set contains 85 instructions. Then, this processor was extended with adding multiply instructions.

The result of the first step is shown in Table 6.4. The column "original" corresponds to the case of the original design, and the column "with PEAS-III" corresponds to the case of the design with PEAS-III. Workload for the design with PEAS-III is about one third compared to the original one. Maximum delays of each design area almost the same and area of the design with PEAS-III is 20 % larger

than original design.

Table 6.4: Result of PEAS-I Core Design

	original	with PEAS-III
work load (hour)	96	32 (*1)
lines in the HDL description	6431	7194 (1038 for MOD)
maximum delay (ns)	9.80	9.74
# of gates	22,247	26,970

(*1) includes learning about the system and improvement of MODs.

Next in this experiment, this processor was extended with additional multiply with signed/unsigned operations using PEAS-III. The result of the logic synthesis is shown in Table 6.5. To implement the multiply instructions, several functional units for multiply operation can be selected. Using the proposed method, this selection is done by specifying the parameters for the multiplier in the resource declarations. Pipeline interlock logic is automatically generated and the designer has no need to design pipeline control logic.

Table 6.5: Delay and Size of PEAS-I Core with Multiply Operation

Design	delay (ns)	Area (gate)
under 100MHz		
SR	17.93	49567.8
SL	9.77	49946.5
CR	9.78	67905.8
CL	9.72	75089.6
under 200MHz		
SR	17.69	50784.4
SL	7.68	51828.9
CR	6.93	69351.8
CL	6.07	76577.2

S: sequential circuit implementation, C: combinational circuit implementation; R: using ripple carry adder, L: using carry-lookahead adder.

6.4 Embedded RISC Controller

This experiment is aimed for comparison between designs with conventional method and designs with the proposed processor synthesis method used in PEAS-III. The original controller that is used for image processing was designed by manual RT-level description. A compatible controller was designed with PEAS-III in this experiment.

This RISC controller has Harvard architecture. The instruction width is 24 bits. The number of instructions is 54. The controller consists of three-stage pipeline. It has synchronous interrupt facility.

An undergraduate student designed this controller with PEAS-III. He had no experience of processor design with PEAS-III at the beginning of this experiment.

Design proceeded as the following way.

1. He learned the usage of PEAS-III.

2. He designed the controller with 32 bits for instruction width.

3. He modified the design to fit 24 bits for instruction width.

The time required for learning PEAS-III is about seven hours. The learning includes reading manuals and trying design with a sample processor attached to PEAS-III.

In the first design, he designed the 32 bits instruction width for ease of the code assignment, because the code assignment of the original instruction set was not given. He implemented all 54 instructions. The workload for this work is shown in the column "first design" of Table 6.6. The total required time is 58 hours. Though he was not familiar with PEAS-III, he designed a processor in a few days. The designed controller has various addressing modes and special registers and resister files. Because the complex addressing mode makes the micro-operation descriptions difficult, design time became longer than other processors in the experiments.

In the second design, he modified the first design concerning about the instruction width. The main work was the modification of instruction format. While some trivial modifications were required, the most part of the micro-operation description was reused.

Table 6.6: Work Load for Designing a RISC Controller

works	first design (hour)	modification (hour)
selecting resources	3	1
determining instruction set architecture	12	8
writing micro-operation description	40	2
modifying errors	2	2
total	58	13

Examples of instruction code assignment of both 24-bit and 32-bit are shown in Fig. 6.1. In this example, code assignments of ADDU (add unsigned) instruction are shown. Fields named like "opr1" in Fig. 6.1 is referred in the micro-operation description.

An example of a portion of a micro-operation description is shown in Fig. 6.2. In this example, the micro-operation description of ADDU instruction is shown. It consists of behavior of each stage. At the stage 2, the value of operands are referred using the names "opr2" and "opr3." As shown in this example, modification of instruction codes can be done without modification of the micro-operation description.

The column "modification" of the Table 6.6 shows the required time for this work.

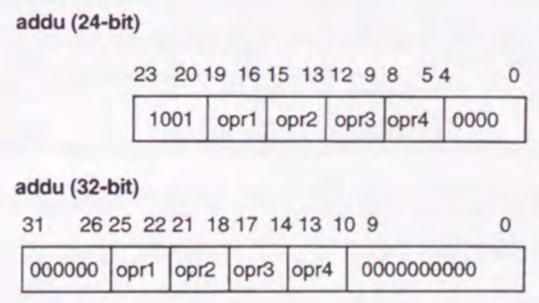


Figure 6.1: Difference of Instruction Code of ADDU between 24-bit and 32-bit in RISC controllers.

The design quality in terms of area and available clock frequency are also exam-

stage 1	IR := IMEM[PC];
	PC.inc();

stage 2	DECODE(IR); $sr1 := freg.read0(opr2); $sr2 := freg.read1(opr3);

stage 3	($result, $aluflg) := ALU.addu($sr1, $sr2, '0');
	aluflg := $aluflg(2) & $aluflg(3);
	freg.write0(opr1, $result);

Figure 6.2: Micro-operation Description of ADDU in RISC Controllers.

ined in this experiment. The generated HDL description of 32-bit version of a RISC controller and the HDL description of real RISC processor were synthesized under the same condition. Table 6.7 shows the result. Two target frequencies 50 MHz and 108 MHz was set up for logic synthesis. Given proper constraint for logic synthesis, both controllers have achieved these frequencies. Note that the original controller has several instructions that were added to the original instruction set for extension, and they were not implemented in the controller designed with PEAS-III. Though rough comparison of the values for the areas is not justified enough, there seems no remarkable difference.

Table 6.7: Comparison of the Design with PEAS-III and with Conventional Method for a RISC Controller

		PEAS-III	conventional method
instruction width (bit)		32	24
work load (hour)		58	420
gate size	[50MHz]	12.7k	14.3k
	[108MHz]	12.9k	14.6k

(using CMOS 0.25 μm library)

6.5 Pipeline Stage Tuning

In this experiment the number of pipeline stages of PEAS R3K-5 was varied from three to five. Then, the design improvement for clock frequency was described.

The improvement includes changing the micro-operation scheduling to the pipeline stages.

PEAS R3K-5 is an extended version of PEAS R3K for data forwarding. Design time for forwarding extension was about half an hour.

6.5.1 Changing the Number of Pipeline Stages

Changing the number of pipeline stages may lead to change the critical path and the number of pipeline registers. In other words, both performance and hardware cost can be improved by proper choice of the number of pipeline stages and micro-operation scheduling to the pipeline stages.

Hardware cost is approximately linear to the number of pipeline stages. Because the number of pipeline registers increases in proportion to the increase of the number of pipeline stages.

On the other hand, maximal frequency is more complicated. If operations in the critical path can be divided into different stages by increasing the number of pipeline stages, the length of the critical path can be reduced. However, if operations in the critical path cannot be divided into different stages, the length of the critical path cannot be reduced.

The critical path of PEAS R3K-5 was the path from pipeline register to program counter (PC) through ALU and stage controller in the third stage. ALU compares operands stored in pipeline registers and output zero-flag, then the stage controller decides whether branch is executed or not and sends control signal for PC to update its value.

To change the number of pipeline stages from five to four, micro-operations in the fourth stage and the fifth stages were merged (PEAS R3K-4) because there were not critical path in these stages. The critical path of PEAS R3K-4 was the same as PEAS R3K-5. To change the number of pipeline stages from four to three, arithmetic and logic operations, address calculation operations in the third stages are merged into previous stage and branch operation was merged into next stage (PEAS R3K-3). Because delay time of a sequential operations such as address calculation operation by ALU and memory access operation need longer time than

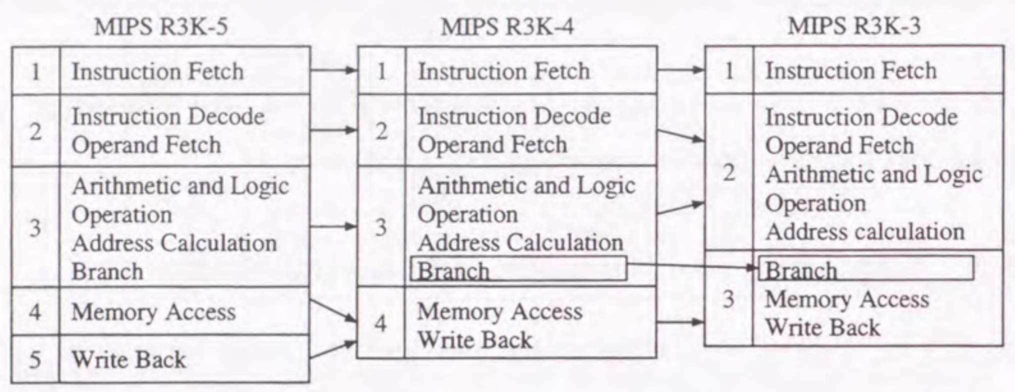

MIPS R3K-5
1 Instruction Fetch
2 Instruction Decode Operand Fetch
3 Arithmetic and Logic Operation Address Calculation Branch
4 Memory Access
5 Write Back
MIPS R3K-4
1 Instruction Fetch
2 Instruction Decode Operand Fetch
3 Arithmetic and Logic Operation Address Calculation Branch
4 Memory Access Write Back
MIPS R3K-3
1 Instruction Fetch
2 Instruction Decode Operand Fetch Arithmetic and Logic Operation Address calculation
3 Branch Memory Access Write Back

Figure 6.3: Scheduling Result of Micro-operations to the Pipeline Stages.

other operations, scheduling these two operations into different stages is preferable. To keep the branch stage same as PEAS R3K-5 and PEAS R3K-4, branch operation was scheduled to the third stage. Figure 6.3 shows a scheduling result of PEAS R3K-4 and PEAS R3K-3.

Table 6.8: Comparison of the Design that has Different number of Pipeline Stages.

# of stages	freq. (MHz)	# of gates (k gates)
3	95.0	57.4
4	121.1	60.4
5	119.9	62.3

using Design Compiler (0.5μm CMOS library)

The number of gates in Table 6.8 is approximately linear to the number of pipeline stages. The difference of clock frequency between four-stage processors and five-stage processors is caused by the difference of the logic of decoder and automatically inserted selectors.

The time of each modification for changing the number of pipeline stages is less than 20 minutes. In micro-operation level processor specification, changing the number of pipeline stages needs rewriting the micro-operation description.

6.5.2 Clock Frequency Improvement

For the improvement of clock frequency, there were two ideas for changing operation scheduling to the pipeline stages.

(a) One was to move the branch stage to the next stage and divide the critical path into two stages as follows: comparison by ALU and zero-flag generation, and conditional program counter update. This modification increased branch penalty. As a result, an execution cycle becomes increased.

(b) The other was addition of dedicated comparator to shorten the delay time of comparison and zero-flag generation.

Table 6.9 shows design modification result for (a) and (b). Because comparison and branch operations were divided into different stages in the design of PEAS R3K-3, the design of R3K-3 (original) and R3K-3 (a) were the same.

Table 6.9: Design Quality of Clock Frequency Improvement.

	original		(a)		(b)	
# of stages	freq. (MHz)	# of gates (k gates)	freq. (MHz)	# of gates (k gates)	freq. (MHz)	# of gates (k gates)
3	95.0	57.4	95.0	57.4	100.7	57.3
4	121.1	60.4	144.3	62.8	140.4	61.1
5	119.9	62.3	141.2	64.5	131.8	62.2

using Design Compiler (0.5μm CMOS library)

From the result shown in Table 6.9, it is confirmed that both modification of (a) and (b) improved clock frequency. Division of branch stage and comparison stage (a) made clock frequency higher than addition of dedicated comparison (b). However, considering the branch penalty increase of processor (a), whether the execution time of processor (a) is less than that of (b) or not it depends on an application program. If an application program includes many branch instructions which are taken frequently, execution cycles of (a) becomes much larger than that of (b), and execution time of (a) becomes larger than that of (b).

On the other hand, the area of (a) was increased because additional pipeline registers were required. The area of (b) was also increased. Additional comparator made the area increase.

The design modification time of (a) and (b) was only two or three minutes for each modification. For the design of (a), micro-operation description of branch and jump instructions were modified with moving branch operation to the next stage. For the design of (b), resource declaration for dedicated comparator was added. Moreover, micro-operation description of branch and jump instructions were modified with replacing comparison resource from ALU to added comparator.

6.6 Design Space Exploration for DSP Application

An FIR filter is one of applications in digital signal processing area. In the second experiment, modules to calculate the following equation are designed as ASIPs:

$$y[n] = \sum_{i=0}^{M} a_i \times x[N-i] \qquad (6.1)$$

where a, x, and y are complex numbers.

Specification of the filter module is as follows. Data size of input/output value is 32 bits. It consists of two 16-bit parts. The higher 16 bits corresponds to the real part of the complex number and the lower 16 bits corresponds to the imaginary part. Both parts are fixed point representation. Input data are provided to the filter module at specified intervals. Output data must be produced before the next data input. The result of the calculation is rounded to round-to-nearest.

An algorithm of this filter is shown in Fig. 6.4. This is a straightforward implementation of Equation (6.1). Variables `ar` and `aj` correspond to real part and imaginary part respectively of coefficients a_i in Equation (6.1). Variables `xr`, `xj`, `yr`, and `yj` follow the same manner.

A program of the filter is coded for PEAS R3K processor in assembly language. The code size is 163 lines.

```
initialize ar and aj;
while (1) {
  retrieve xr[0] and xj[0] from input;
  yr = 0;
  yj = 0;
  for (i = M; i >= 0; i--) {
    yr += ar[M - i] * xr[i] - aj[M - i] * xj[i];
    yj += ar[M - i] * xj[i] + aj[M - i] * xr[i];
    xr[i] = xr[i - 1];
    xj[i] = xj[i - 1];
  }
  output yr and yj;
}
```

Figure 6.4: Pseudo Code of an FIR Filter.

6.6.1 Customization of PEAS R3K

To improve performance, three types of new instructions are added. As another architectural design space exploration, the effect of changing the number of pipeline stages is examined.

Adding New Instructions

Complex MAC Complex MAC type instructions consist of complex MAC operation and related operations such as initialization of complex MAC operation. The instruction 'cmult' performs multiply, accumulation, and rounding. By introducing instructions related to complex MAC operation, drastic improvement of execution cycles of the application is expected.

To implement the Complex MAC type instructions, a complex MAC module was designed. A block diagram of the module is shown in Fig. 6.5. This MAC Unit simultaneously calculates complex multiplication and addition, in other words, real part and imaginary part computation, at once. It also includes a round-to-nearest rounding function.

To add Complex MAC type instructions to PEAS R3K, instruction definitions and micro-operation descriptions were added by the designer. The micro-operation

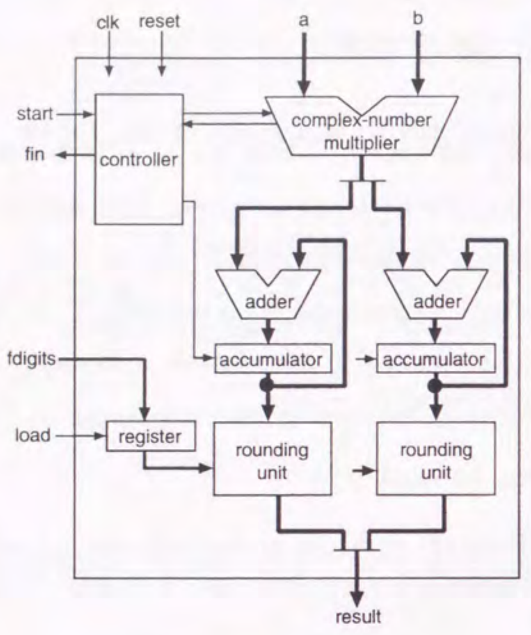


Figure 6.5: Block Diagram of a Complex MAC Unit.

description of 'cmult' instruction is shown in Fig. 6.6. In this description, it is specified that the pipeline is proceeding with instruction fetch at stage1, decoding of fetched instruction at stage2, execution of complex MAC operation with complex MAC module 'CMAC0' at stage3. As shown in this example, multi-cycle operations do not need supplemental description compared to single cycle operations since proposed processor synthesis method can detect multi-cycle operations and generates the controller with multi-cycle handling.

stage1	IR := IMEM[PC]; PC.inc();
stage2	DECODE(IR); $rs := GPR.read0(rs); $rt := GPR.read1(rt);
stage3	($result, $flag) := CMAC0.mac($rs, $rt);
stage4	
stage5	

Figure 6.6: Micro-operation Description of cmult (Complex MAC).

Modulo Addressing Modulo addressing is one of addressing modes to calculate address for queues. In the algorithm in Fig. 6.4, buffer x for preceding inputs has overhead of load/store. Using Modulo addressing, when load/store instruction is executed, the next address is also calculated in the instruction. By introducing instructions related to Modulo addressing, some improvement of execution cycles of the application is expected.

Since these instructions require no special resources, the designer only added instruction definitions and micro-operation descriptions for introducing these instructions.

Loop Loop instruction is one of branch instructions. The loop instruction performs decrement of counter and branch as a single instruction. Though the improvement of the number of the execution cycles is at most one instruction per iteration, relatively large improvement can be expected for the iteration of short basic block length.

To implement Loop instruction, the designer added instruction definitions and micro-operation descriptions for introducing these instructions.

6.6.2 Pipeline Stage Tuning for Derivative Processors

The number of pipeline stages of derivative processors, which are added the instruction described in the previous section, was varied from three to five. Micro-operation re-scheduling that is described Section 6.5 is also done to improve clock frequency.

6.6.3 Results of Design Space Exploration for DSP Applications

Results of logic synthesis for each modification are shown in this section. Design times for each modification are also shown.

Results of Adding Instructions

Five derivative version processors have been designed. Let M mean the processor including complex MAC instructions, L mean the processor including Loop instructions, and A mean the processor including modulo addressing instructions. Results of logic synthesis, i.e., the number of gates and maximal clock frequency, and the number of execution cycles for calculating a single output value for $M = 128$ are summarized in Table 6.10.

Table 6.10: Design Quality for Each Processors

processor	max frequency (MHz)	# of gates (k gates)	# of execution cycles
original	119.9	80.0	23932
M	101.8	71.4	3893
L	104.4	64.7	23805
MA	100.0	92.0	3507
ML	102.7	73.6	3766
MAL	100.7	94.8	3509

using Design Compiler (0.5μm CMOS library)

M	:	including complex MAC instructions
L	:	including Loop instructions
A	:	including Modulo Addressing

In Table 6.10, the number of execution cycles is drastically reduced by introducing CMAC type instructions. In this case, maximal frequency of processor decrease approximately 30%.

Table 6.11: Design Time for Each Instructions

instructions	time (hour)
original instructions	8.5
CMAC	0.8
Mod. Addr.	0.5
Loop	0.8

Design time for each processors is shown in Table 6.11. Original PEAS R3K processor has been designed in eight hours. To add new instructions, less than one hour was required in any type of instruction in this experiment. Furthermore, any processor, which has any combination of already designed instruction, can be easily synthesized by PEAS-III.

Results of Pipeline Tuning

Results of logic synthesis for each design are summarized in Table 6.12. The column "model" denotes the variation of instruction set addition shown in Table 6.10. The column "type" denotes variation of clock frequency improvement shown in Section 6.5.2.

As mentioned in Section 6.5, the number of gates in Table 6.12 is approximately linear to the number of pipeline stages, too. Clock frequency for three stage derivatives is about 40 % less than that of four and five stage derivatives. Clock frequency of both four and five stage derivatives is almost the same.

Table 6.12: Design Quality for Changing the Number of Pipeline Stages

model	type	# of exec. cycles	# of pipeline stages					
			3 stages		4 stages		5 stages	
			freq. (MHz)	area (k gates)	freq. (MHz)	area (k gates)	freq. (MHz)	area (k gates)
original	orig	23932	95.0	57.4	121.1	60.4	119.9	62.4
	a	24063	95.0	57.4	144.3	62.8	141.2	64.5
	b	23932	100.7	57.3	140.4	61.2	131.8	62.2
ML	orig	3766	66.3	70.9	101.9	72.6	102.7	73.6
	a	3895	66.3	70.9	102.8	75.4	100.6	75.3
	b	3766	65.8	71.4	102.6	73.2	104.0	73.8
MA	orig	3507	73.4	85.9	101.9	89.1	100.0	92.0
	a	3636	73.4	85.9	98.3	91.8	98.9	93.9
	b	3507	72.2	86.5	102.0	89.2	101.5	92.1
MAL	orig	3509	65.3	89.3	98.0	92.4	100.7	94.8
	a	3638	65.3	89.3	104.8	94.2	102.7	96.3
	b	3509	65.1	89.3	103.8	92.7	101.2	93.9

using Design Compiler (0.5μm CMOS library)

Relationship between area and execution time for FIR filtering application is plotted in Fig. 6.7. Trade-off between area and execution time is plotted in Fig. 6.8. At various design constraint, various architecture candidates can be selected in terms of the number of pipeline stages, extension instruction set and so on.

Design time of derivative processors in terms of pipeline tuning was within an hour per one model. Total design time of pipeline tuning for four models in Ta-

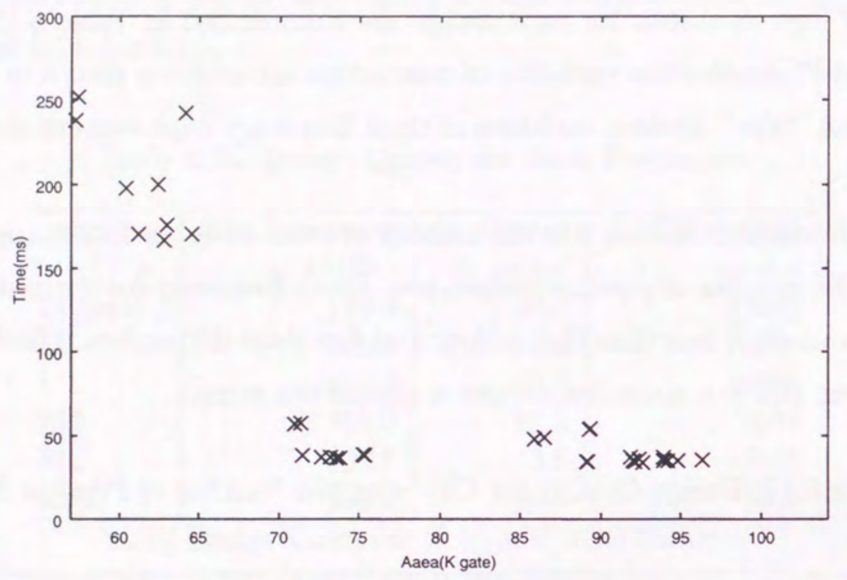


Figure 6.7: Area and Execution Time for all Derivatives.

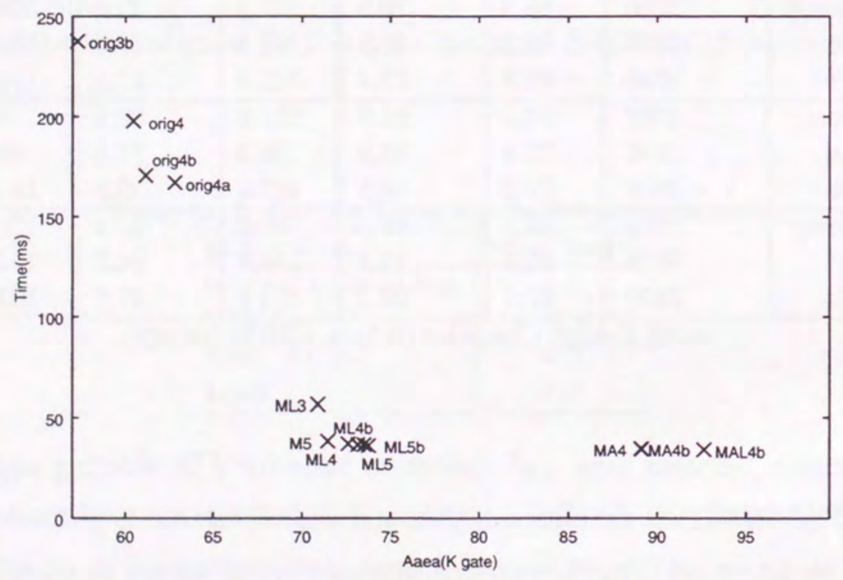


Figure 6.8: Trade-off of Area and Execution Time.

ble 6.12 took four hours. Total design time of all derivatives, which was the addition design time of new instructions and design time of pipeline tuning, was about six hours.

Chapter 7

Discussion

In this chapter, the effectiveness of the proposed processor design and synthesis method is discussed with the results of experiments. The effectiveness is discussed at the following points:

- largeness of explored design space,
- design and design exploration time,
- and design quality.

7.1 Design Space

The proposed synthesis method supports portion of the architectural characteristics shown in Chapter 3. The supported items are as follows:

- hardware module configuration,
- storage units organization,
- pipeline organization that includes the number of pipeline stages and micro-operation assignment to the pipeline stages,
- structural hazard detection and pipeline interlock control synthesis,
- predict-not-take based delayed branch,
- and external interrupt.

From experimental results, effectiveness of design space exploration with these architectural variations was shown.

In the experiments, design space was explored in terms of the following points: Design space was explored in terms of the following points:

- hardware module configuration. Hardware configuration includes changing resource parameters and addition of new resources. Changing resource parameters for PEAS-I core, which is shown in Section. 6.3. Addition of new resource for clock frequency improvement is shown in Section 6.5.2.
- instruction bit width. The instruction bit width of embedded RISC controller was changed from 32-bit to 24-bit.
- the number of pipeline stages. The number of pipeline stages for PEAS R3K processor and derivative processors for DSP applications were changed. It is shown in Section 6.5 and Section 6.6.2.
- operation scheduling to the pipeline stages. Changing the stage of branch operation for PEAS R3K and DSP derivative processors is shown in Section 6.5 and Section 6.6.2.
- organization of storage units. Complex addressing modes for special registers and multiple register files were designed. Complex addressing modes were shown in Section6.4.

Furthermore proposed synthesis method has a potential ability for designing complex memory architecture, such as memory-memory architecture, non-harvard architecture, multiple port memory and so on. Synthesis of structural hazard detection and pipeline interlock logic enables to design such processors.

Proposed synthesis method can deal with much larger design space than that of existing prepared processor based systems. Design space is enlarged in terms of instruction bit width, user-defined pipeline organization in terms of the number of pipeline stages, the number of delayed branch slot, role of each pipeline stage and multi-cycle operations, storage unit organization, and user-defined external interrupt.

For the further expansion of the design space, extension for out-of-order instruction issue and out-of-order completion, VLIW architecture, and internal exception are required. When target processor uses a functional unit that has long latency to calculate the result, the pipeline organization with out-of-order completion is effective. The processor with out-of-order completion can execute other succeeding instructions while executing instructions that have long latency. For the applications which requires high performance, the processor with out-of-order instruction issue and VLIW processors are suitable to execute multiple instructions at the same time.

Extension for branch mechanism is also required. The synthesis method cannot deal with the non-overhead loop instructions which are popular in DSP application because branch architecture is fixed to predict-not-take base delayed branch.

7.2 Design Time and Design Space Exploration Time

7.2.1 Design Time for New Processors

With the higher abstraction level processor specification than RT level, design time of the ASIPs are drastically reduced. Higher abstraction level processor description contributes the easiness of the design.

From the experiments, reduction of the design time was shown. Design time for ASIP with proposed method was about three to seven times shorter than those for conventional RT level design as shown in experiments. Compared with other processor description language AIDL [24], AIDL needs 37 hours to design 23 instructions of PA-RISC processor. AIDL includes complex specification descriptions for complicated processors. From the results, it is obvious that micro-operation level processor description is effective for shortening design time of straightforward pipelined processors.

7.2.2 Design Time for Derivative Processors

Proposed micro-operation level processor specification also reduces design exploration time compared with that of RT level processor specification. Synthesis of datapath structure and controller reduces design and design modification time of them and enables the designer to change the architecture in a short time.

From the experiments, turn around time for derivative processor designs was shown. The derivative processor designs includes the following modifications:

- changing resource parameters. The designer change parameters for the sake of the evaluation of various hardware module which has same functionality and different design quality in the sense of area, clock frequency and execution cycles. This modification needs only few seconds per one parameter.
- addition of application specific user-defined instructions. The designer defines instruction format and describe micro-operation description of new instructions. This modification takes ten to fifteen minutes per one instruction of DSP instructions shown in Section 6.6.
- addition of new resource. The designer declares additional resources to gain performance of the design. It takes only a few minutes.
- changing the number of pipeline stages and changing operation scheduling to pipeline stages. The number of pipeline stages will be decreased for the reduction of the area. On the other hand, the number of pipeline stages will be increased to reduce the delay time of the critical path. The designer also changes the stage of micro-operation to reduce the delay time of critical path. Changing the number of pipeline stages and operation scheduling to the pipeline stages requires re-scheduling of micro-operation to the pipeline stages. The changing time is within a minutes per one instruction. From the experiments, pipeline tuning takes 20 minutes for the PEAS R3K processor that has 52 instructions.

Large design space has been successfully explored. The trade-off of the design is found. The designs of 12 derivatives were tried in a day.

Though design and design modification time is very short, evaluation and validation time for the designed processor makes turn around time long. Effective and rapid estimation and critical path analysis for synthesized processor are required.

For more efficient support of design space exploration, optimization of resource selection, instruction format decision, the number of pipeline stages and micro-operation assignment for pipeline stages are required.

7.3 Design Quality

In the design quality of synthesized processors and manually design processors, clock frequencies of them are almost the same. The area of synthesized processors is about 20% larger than those of manually designed processors. Though the area is inferior to manual design, the advantage of effective design space exploration has an impact on the total design quality. The disadvantage on the area does not affect so much.

To improve the design quality of synthesized processor, optimization of selector and pipeline register insertion is required. For the reduction of pipeline register, pipeline register sharing could be effective. However, sharing the register needs additional selectors. On the other hand, when critical path includes automatically inserted selectors, moving the selector to the previous pipeline stage or to the next pipeline stage, if possible, reduces the critical path. Pipeline register sharing and selector insertion stage optimization based on an RT level rapid and accurate estimation improve the design quality.

Chapter 8

Conclusions and Future Work

In this thesis a micro-operation level processor specification and processor synthesis method is proposed for the architectural design space exploration of ASIPs.

8.1 Conclusion

In this thesis, micro-operation level processor specification for architectural design space exploration has been discussed. The specification includes a parameterized pipeline structure in the sense of the number of pipeline stages and roles of the pipeline stages. Furthermore, a complex mechanism that includes pipeline control logic, designing a datapath structure is not needed. The easiness of the design and design modification and flexibility on the processor enables architectural exploration of a large design space in a short design time.

For processor synthesis, a processor model has been examined. To deal with flexibility in pipeline depth of target processor, datapath and controller is divided into pipeline stages. The sequence of datapath and controller models of each pipeline stage organizes the pipelined processor. The organization of each pipeline stage controller, instruction decoder and external interrupt controller are discussed. The pipeline hazard detection and control mechanism that includes pipeline interlock and pipeline flush are formalized. The pipeline control model and pipeline hazard detection mechanism are used to synthesize pipeline control logic from micro-operation level processor specifications.

Processor synthesis method from micro-operation level processor specification is proposed. Each part of the datapath synthesis, such as data flow graph generation, signal conflicts resolution and pipeline register insertion are described. Instruction decoder synthesis, pipeline control logic synthesis and interrupt controller synthesis are also described. The synthesis method supports user-defined pipeline organization in the sense of the number of pipeline stages and the number of delayed branch slot, multi-cycle operation, structural hazards resolution, and external interrupts. The wide support for the pipelined processor enables exploration of a large design space for ASIPs.

Through the five kinds of experiments, the effectiveness of the micro-operation level processor design is confirmed. The design time and the architectural design space exploration time are reduced while keeping the flexibility in the pipeline organization, hardware configuration and so on. A large design space has been successfully explored. The trade-off of the design is found by trying 12 derivative processors. The designs of 12 derivatives were tried in a day.

In this thesis, micro-operation level processor specification and a processor synthesis method for architectural design space exploration for ASIPs are proposed. It is confirmed that in using the proposed method, large design spaces are easily explored in terms of the number of pipeline stages and delayed branch slots and hardware module configuration, user-defined instructions interface ports and external interrupts, and operation scheduling to pipeline stages.

8.2 Future Work

Future work for further architectural design space exploration includes the expansion of design space and reduction of design exploration time. Improvement of the design quality of synthesized processors is also a future goal. In the following section, the future directions of these articles are described.

8.2.1 Design Space Expansion

Supports for the following characteristics described in Section 3.1 to enlarge design space are required.

- branch prediction mechanism and non-overhead loop,
- out-of-order completion,
- out-of-order instruction issue,
- and other interrupt and exception.

Instruction fetch module synthesis is required to extend branch mechanism because branch control is closely related to instruction fetch. Parameterization of instruction fetch module and branch architecture, and consideration of their synthesis method enable the system to deal with various branch architectures. The parameter seems to include instruction bit width, increment step, predict-taken or predict-not-taken or dynamic branch prediction with branch-prediction buffers or that with branch-target buffers, and parameters of buffers.

Processor model extension for multiple pipeline sequence enables super-scalar and VLIW type processor synthesis. However, the extension for multiple pipeline sequence makes hazard detection and the resolution algorithm more complex. For out-of-order instruction issue, reservation station synthesis is also required.

For the support of the precise exception, restriction of instruction specification and exception handling should be discussed. Extension instruction flash and restoring the processor status mechanism for exception is also required.

8.2.2 Design Exploration Time Reduction

For the further reduction of the design and design modification time, optimization of micro-operation level processor specification is required. The target of the optimization includes instruction format assignment, resource parameter selection, the number of pipeline stages and micro-operation assignment for pipeline stages.

The critical path of each pipeline stage can be reduced by changing the pipeline stage assignment of micro-operations and hardware module parameters.

Data flow graph generation from micro-operation description and ASAP (as soon as possible) base scheduling with design constraint enables optimization of micro-operation assignment to the pipeline stages. At the same time, the parameter selection of resources should be done because the design qualities of the resources affect the delay time of the critical path. For the scheduling and resource parameter selection, fast and accurate estimation of micro-operation level processor specification is required, too.

8.2.3 Improvement of the Design Quality

The design quality of the synthesized processor is slightly inferior to that of manual design using RT level HDL description. For the improvement of the design quality, optimization of selector and pipeline register insertion taking into account the trade-off between clock frequency and hardware cost is required, as discussed in Section 7.3.

Bibliography

[1] Electronic Industries Association of Japan (EIAJ), *EDA Technology Roadmap Toward 2002–Cyber-Giga-Chip Design Technology*, apr 1998.

[2] International SEMATECH, *International Technology Roadmap for Semiconductors 1999 Edition*, 1999.

[3] J. Staunstrup and W. Wolf, eds., *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[4] G. Kane, *mips RISC Architecture*. New Jersey: Prentice-Hall, Inc., 1988.

[5] Hitachi Ltd., *SuperH^{TM} RISC Engine SH7020 and SH7021 Hardware Manual*, rev. 3.0 ed., Nov. 1999.

[6] Hitachi Ltd., *SuperH^{TM} RISC Engine SH7604 Hardware Manual*, rev. 3.0 ed., Nov. 1999.

[7] S. Furber, *ARM System ARchitecture*. Addison Wesley Longman, 1996.

[8] Advanced RISC Machines Ltd., *ARM7TDMI Data Sheet*, Aug. 1995.

[9] S. B. Furber, *VLSI RISC Architecure and Organization*. Marcel Dekker Inc., 1989.

[10] J. Sato, A. Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai, "PEAS-I: A Hardware/Software Codesign System for ASIP Development," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E77-A, pp. 483–491, Mar. 1994.

[11] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura, "The Satsuki Intergrated Processor Synthesis and Compiler Generation System," in *Proc. of the Synthesis and System Integration Mixed Techinologies (SASIMI '96)*, (Fukuoka, Japan), pp. 135–142, Nov. 1996.

[12] M. Small, *An Intro to The First User Definable Processor.* ARC Cores Ltd., Nov. 1998.

[13] M. R. Borbacci and D. P. Siewiorek, *The Design and Analysis of Instruction Set Processors.* McGraw-Hill, 1982.

[14] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, *Code Generation for Embedded Processors*, ch. 4. Kluwer Academic Publishers, 1995.

[15] Tensilica, Inc., *Application Specific Microprocessor Solutions: Data Sheet for Xtensa V1*, 1998.

[16] Hewlett Packard Laboratories Compiler and Architecture Group, New York University ReaCT-ILP Group, University of Illinois IMPACT Group, *Trimaran User Manual -An Infrastructure for Compiler Research in Instruction Level Parallelism-*, 1998.

[17] R. Campasano and J. Wilberg, "Embedded System Design," *Design Automation for Embedded Systems*, vol. 1, Jan. 1996.

[18] J.-H. Yang, B.-W. Kim, S.-J. Nam, Y.-S. Kwon, D.-K. Lee, J.-Y. Lee, C.-S. Hwang, Y.-H. Lee, and C.-M. Kyung, "MetaCore: An Application Specific Programmable DSP Development System," *IEEE Transactions on VLSI SYSTEMS*, vol. 8, pp. 174–183, Apr. 2000.

[19] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargeability," in *Proc. of 34th Design Automation Conference (DAC '97)*, June 1997.

[20] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Com-

piler/Simulator Retargetability," in *Proc. of Design, Automation & Test in Europe (DATE '99)*, (Munich, Germany), pp. 485–490, Mar. 1999.

[21] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," in *Proc. of 36th Design Automation Conference (DAC '99)*, (New Orleans), June 1999.

[22] P. Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors," in *Proc. of the 21th Design Automation Conference (DAC '84)*, pp. 587–593, 1984.

[23] A. Fauth, J. V. Praet, and M. Freericks, "Describing Instruction Sets Using nML (Extended Version)," tech. rep., Technische Universität Berlin and IMEC, Berlin(Germany)/Leuven(Belgium), 1995.

[24] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa, "Advanced Processor Design Using Hardware Description Language AIDL," in *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC '97)*, (Chiba, Japan), pp. 387–390, Jan. 1997.

[25] M. Hamabe, A. Nose, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A Generation System for Hardware Description of Pipelined Processors," in *Tech. Report of IEICE, VLD97-117*, pp. 33–40, 1997. (in japanese).

[26] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "HMDES Version 2 Specification," Tech. Rep. IMPACT-96-03, University of Illinois, Urbana IL., 1996. IMPACT Technical report.

[27] V. Kathail, M. Schlansker, and B. Rau, "HPL Playdoh Architecture Specification: Version 1.0," Tech. Rep. HPL-93-80, HP Laboratories, 1994.

[28] G. Hadjiyiannis, *ISDL: Instruction Set Description Language, User's Manual*, 1998.

[29] N. Savoi, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "VSAT: A Visual Specification and Analysis Tool for System-On-Chip Exploration," in *Proc. of 25th EUROMICRO Conference*, Sept. 1999.

[30] Institute for Integrated Signal Processing Systems, ISS-RWTH Aachen, *LISA User's Guide (Version 2.0)*, Oct. 2000.

[31] V. Zivojnovic, S. Pees, C. Schlager, M. Willems, R. Schoenen, and H. Meyr, "LISA - Machine Description Language and Generic Machine Model," in *International Conference on Signal Processing Applications and Technology (ICSPAT)*, (Boston), Oct. 1996.

[32] M. Freericks, "The nML machine description formalism," tech. rep., Universität Berlin, Fachbereich Informatik, Berlin, 1991.

[33] D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: Retargetable Code Generation for Embedded DSP Processors," in *Code Generation for Embedded Processors*, pp. 85–102, Kluwer Academic Publishers, 1995.

[34] K. Kataoka, A. Shiomi, M. Imai, Y. Aoyama, J. Sato, and N. Hikichi, "Observations on the Implementation of a Codesign Workbench PEAS-III for ASIP Design – Classification and Parameterization of CPU Architectures –," in *IPSJ Technical Report, DA 78-20*, pp. 121–126, Dec. 1995. (in japanese).

[35] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: An ASIP Design Environment," in *IEEE International Coference on Computer Design: VLSI in Computers & Processors (ICCD 2000)*, (Austin), pp. 430–436, Sept. 2000.

[36] D. L. Perry, *VHDL*. McGraw-Hill, Inc., second ed., 1994.

[37] T. Morifuji, Y. Takeuchi, J. Sato, and M. Imai, "Flexible Hardware Model: Implementation and Effectiveness," in *Proc. of the Synthesis and System Integration Mixed Techinologies (SASIMI '97)*, pp. 83–89, Dec. 1997.

[38] M. Imai, A. Shiomi, Y. Takeuchi, and J. Sato, "Hardware/Software Codesign in the Deep Submicron Era," in *Proc. of International Workshop on Logic and Architecture Synthesis (IWLAS '96)*, pp. 236–248, Dec. 1996.

[39] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 1994.

[40] S. Kowatari, H. Iwashita, T. Nakata, and F. Hirose, "Synthesizable Design Generation for Pipeline Controllers," in *Tech. Report of IEICE, VLD94-41*, pp. 17–24, July 1994. (in japanese).

[41] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-level synthesis: introducntion to chip and system design*. Kluwer Academic Publishers, 1992.

[42] C. Price, *MIPS IV Instruction Set Revision 3.2*. MIPS Technologies, Inc., 1995.

[43] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitive Approach*. California: Morgan Kaufmann Publishers, Inc., 2nd ed., 1990.

Appendix A

Grammar of Micro-operation Level Processor Specification

A.1 Organization of Micro-operation Level Specification

Micro-operation level specification is a text file divided into eight parts:

1. Version
2. Design Goal and Architecture Parameter
3. Interface Definition
4. Instruction Type Definition
5. Instruction Definition
6. Resource Declaration
7. Interrupt Definition
8. Micro-operation Description

General grammar of micro-operation level specification is as follows:

```
KEY        := [a-zA-Z][a-zA-Z0-9_]*
STRING     := "([^"]|'\"')*"
<design>   := <item>
<item>     := <keyword> ['{' <item>'}'] {',' <item>}
<keyword>  := (KEY | STRING)
```

"KEY" includes "clk(n)" as an exception. n takes integer value.

Each part has own syntax and keywords. Each part begins with the keywords: **Version**, **Port_declaration**, **Instruction_type**, **Instruction**, **Resource**, **Exception**, and **MOT**. Keywords and syntax for each part are described in the following sections.

```
<Specification> := 'Design' '{'
                     <Version> ','
                     <Architecture_Parameter> ','
                     <Interface_Declaration> ','
                     <Inst_Type_Definition> ','
                     <Instruction_Definition> ','
                     <Resource_Declaration> ','
                     <Interrupt_Definition> ','
                     <Micro_Op_Description>
                   '}'
```

A.2 Architecture Parameter

In the architecture parameter part, design goal and architecture parameters are specified.

```
<Architecture_Parameter> := 'Abstract_level_architecture' '{'
                              <Fhm_Work_Name> ','
                              <Design_Goal> ','
                              <Processor_Type> ','
                              <Pipeline_Arch_Parameter>
                            '}'
<Fhm_Work_Name>          := 'Fhm_workname' '{' STRING '}'
<Processor_Type>         := 'CPU_type' '{'
                              ('"Non-pipeline"' | '"Pipeline"' | '"VLIW"')
                            '}'
STRING                   := "([^"]|'\"')*"
```

Architecture parameter includes login name of FHM-DB, design goal, processor type and architecture parameters for pipeline processor. Candidates of processor type are "Non-pipeline", "Pipeline" and "VLIW". Processor types "Non-pipeline" and "VLIW" are prepared for future extension. Currently, "Pipeline" architecture is supported.

A.2.1 Design Goal

```
<Design_Constraint> := 'construction' '{'
                         <Goal_Frequency> ','
                         <Goal_Area> ','
                         <Goal_Delay> ','
                         <Goal_Power_Static> ','
                         <Goal_Power_Dynamic>
                       '}'
<Goal_Frequency>    := 'Goal_frequency' '{' VALUE '}'
<Goal_Area>         := 'Goal_area' '{' VALUE '}'
<Goal_Delay>        := 'Goal_delay' '{' VALUE '}'
<Goal_Power_Static> := 'Goal_power_S' '{' VALUE '}'
<Goal_Power_Dynamic> := 'Goal_power_D' '{' VALUE '}'
VALUE               := "[0-9]*[\.[0-9]*]"
```

Design goal for clock frequency, chip area, maximum delay, static power consumption and dynamic power consumption. Their values are represented in decimal or integer.

A.2.2 Pipeline Processor Parameters

Architecture parameters for pipeline processor include the number of stages, the number of common stages, the number of phases per stages, synthesis of pipeline interlock logic, data bypassing logic, and branch control logic.

```
<Pipeline_Arch_Parameter>      := 'Pipeline_architecture' '{'
                                    <Number_of_Stages>
                                    <Number_of_Common_Stages>
                                    <Number_of_Phases_Per_Stage>
                                    <Multi_Cycle_Interlock>
                                    <Data_Hazard_Interlock>
                                    <Register_Bypass>
                                    <Memory_Bypass>
                                    <Delayed_Branch>
                                    <Delayed_Branch_Slot>
                                  '}'
<Number_of_Stages>             := 'Number_of_stages' '{' INT '}'
<Number_of_Common_Stages>      := 'Number_of_common_stages' '{' INT '}'
<Number_of_Phases_Per_Stage>   := 'Number_of_phases_per_stages' '{' INT '}'
<Multi_Cycle_Interlock>        := 'Multi_cycle_interlock' '{' USAGE '}'
<Data_Hazard_Interlock>        := 'Data_hazard_interlock' '{' USAGE '}'
<Register_Bypass>              := 'Register_bypass' '{' USAGE '}'
<Memory_Bypass>                := 'Memory_bypass' '{' USAGE '}'
<Delayed_Branch>               := 'Delayed_branch' '{' YESNO '}'
<Delayed_Branch_Slot>          := 'Number_of_exec_delayed_slot'{'
                                    'number' '{' INT '}'
                                  '}'
INT                            := "[0-9]+"
USAGE                          := ( "Use" | "Unuse" )
YESNO                          := ( "Yes" | "No" )
```

The number of pipeline stages takes integer value. The parameter value of delayed branch indicates whether the processor adopts delayed branch architecture or not. If the parameter value of delayed branch is "Yes", the number delayed branch slot should be specified. The number of delayed branch slot must be less than the number of branch stage.

The following parameters are prepared for future extension: the number of common stages, the number of phases per stages, synthesis of pipeline interlock logic and bypassing logic. Pipeline interlock logic for the resolution of structural hazards is automatically synthesized regardless of the parameter value.

A.3 Interface Definition

In the interface definition part, entity name, and input and output ports of target processor are defined.

```
<Interface_Definition> := <Entity_Name> ',' <Port_Declarations>
<Entity_Name>          := 'Entity_name' '{' STRING '}'
<Port_Declarations>    := 'Port' '{'
                            <Port_Declaration> { ',' <Port_Declaration> }
                          '}'
<Port_Declaration>     := <Port_Name> '{'
                            <Port_Direction> ','
                            <Port_Type> ','
                            <Port_Attribute> ','
                          '}'
<Port_Direction>       := 'direction' '{' <inout_direction> '}'
<inout_direction>      := ('"in"' | '"out"' | '"inout"')
<Port_Type>            := 'signal_type' '{' STRING '}'
<Port_Attribute>       := 'signal_attribute' '{' STRING '}'
STRING                 := "([^"]|'\"')*"
```

Interface definition part consists of two parts, entity name definition and interface port declarations. For each interface port, port name, direction, type and attribute are defined. Port direction is selected among "in", "out", and "inout". Type of the port is described in VHDL standard logic style. If the bit width of the port is one, the port type becomes "std_logic". If the bit width of it is more than one, the port type becomes standard logic vector type.

A.4 Instruction Format Definition

Instruction format definition consists of two parts, instruction type definition and instruction definition.

A.4.1 Instruction Type Definition

Instruction type definition part consists of a list of instruction type definitions.

```
<Inst_Type_Definition> := 'Instruction_type' '{'
                            'sub_field_name' '{'
                              'NO_VLIW' <Field_Width> ','
                              'type' '{'
                                <Inst_Type> { ',' <Inst_Type> } '}'
                              '}'
                            '}'
                          '}'
<Field_Width>          := 'width' '{' INT ',' INT '}'
INT                    := "[0-9]+"
```

In the instruction type definition part, bit fields, field type, field name, and binary value of it are defined for each instruction type.

```
<Inst_Type>            := <Inst_Type_Name> '{' <Field> { ',' <Field> } '}'
<Inst_Type_Name>       := STRING
<Field>                := <Field_Type> '{'
                            <Field_Value> ','
                            <Field_Width>
                          '}'
<Field_Type>           := ('"OP-code"' | '"Operand"' | '"Reserved"')
<Field_Contents>       := <Field_Contents_Type> '{' <Field_Value> '}'
<Field_Contents_Type> := ('"name"' | '"binary"')
<Field_Value>          := (STRING | BINARY)
BINARY                 := "[01]+"
STRING                 := "([^"]|'\"')*"
```

For each instruction type, instruction type name is defined. Then, instruction fields of the instruction type are defined. For each instruction field, field type, contents of the field are defined. Field type is selected among "OP-code," "Operand" and "Reserved." "OP-code" means operation code and "Reserved" indicates that the field is reserved for extension in the future.

If the type of instruction field is "OP-code" and the operation code for that field is common to all the instructions belongs to the instruction type, the field contents type becomes "binary" and field value is specified in binary representation. If the operation code is variable for the instructions that belongs to the instruction type, the field contents type becomes "name" and field name of it is defined. The operation code of the field is specified in instruction definition part for each instruction. For the "Operand" field, field type becomes "name" and field name is defined. For the "Reserved" field, field type becomes "binary" and field value is specified.

A.4.2 Instruction Definition

For each instruction, instruction type is selected among defined instruction types and operation code value is decided.

```
<Instruction_Definitions> := 'Instruction' '{'
                               'NO_VLIW' '{'
                                 <Instruction> { ',' <Instruction> }
                               '}'
                             '}'
<Instruction>             := <Instruction_Name> '{'
                               <Field> { ',' <Field> }
                             '}'
<Instruction_Name>        := STRING
STRING                    := "([^"]|'\"')*"
```

Operation code for each instruction is assigned. The syntax of instruction definition is common to instruction type definition part.

A.5 Resource Declaration

In the resource declaration part, hardware modules are selected with appropriate parameters from parameterized hardware library FHM-DB.

```
<Resource_Declarations> := ’Resource’ ’{’
                             <Resource> {’,’ <Resource>}
                           ’}’
<Resource>              := <Resource_Name> ’{’
                             <FHM_Model_Name> ’,’
                             <Class_Path> ’,’
                             <Parameters>
                           ’}’
<Resource_Name>         := STRING
<FHM_Model_Name>        := ’class’ ’{’ STRING ’}’
<Class_Path>            := ’classpath’ ’{’ STRING ’}’
STRING                  := "([^"]|’\"’)*"
```

Resource declaration part consists of a list of resources. For each resource, resource name, FHM model name and its parameter values are specified. The class path field is prepared for future extension.

```
<Parameters>        := ’parameter’ ’{’
                         <Abstraction_Level>
                         {’,’ <FHM_Parameter>}
                       ’}’
<Abstraction_Level> := ’abstraction_level’ ’{’
                         ’for_simulation’ ’{’ <Level> ’}’ ’,’
                         ’for_synthesis’ ’{’ <Level> ’}’
                       ’}’
<Level>             := ( ’"Behavior"’ | ’"RT"’ | ’"Gate"’ )
<FHM_Parameter>     := <Parameter_Name> ’{’ <Parameter_Value> ’}’
<Parameter_Name>    := NAME
<Parameter_Value>   := STRING
NAME                := [a-zA-Z][a-zA-Z0-9_]*
STRING              := "([^"]|’\"’)*"
```

Resource parameter includes abstraction level for synthesized simulation model and logic synthesizable model. Abstraction level must be selected among “Behavior”, “RT” and “Gate”. After the abstraction level, parameters, which are specific to the model, are specified.

A.6 Interrupt Definition

In the interrupt definition part, interrupt condition definitions and micro-operation description of interrupts are combined.

```
<Interrupt_Definition> := ’Exception’ ’{’
                            [<Interrupt> { ’,’ <Interrupt> }]
                          ’}’
<Interrupt>            := <interrupt_Name> ’{’
                            <Interrupt_Condition> ’,’
                            <Interrupt_Type> ’,’
                            <Interrupt_Cycles> ’,’
                            <Behavior_of_Interrupt> ’,’
                            <Assertion_of_Interrupt> ’,’
                            <Comment_for_Interrupt> ’,’
                            <MOD_of_Interrupt>
                          ’}’
```

Interrupt definition includes interrupt name, condition, type, execution cycle count, behavior, assertion, comments and micro-operation description. Interrupt type, behavior and assertion are prepared for future extension.

```
<Interrupt_Condition>    := ’Condition’ ’{’ <Condition> ’}’
<Interrupt_Type>         := ’Type’ ’{’ <Interrupt_Types> ’}’
<Interrupt_Types>        := ( ’"Internal"’ | ’"External"’ )
<Interrupt_Cycles>       := ’Cycles’ ’{’ INT ’}’
<Behavior_of_Interrupt>  := ’Behavior’ ’{’ STRING ’}’
<Assertion_of_Interrupt> := ’Assert’ ’{’ STRING ’}’
<Comment_for_Interrupt>  := ’Comment’ ’{’ STRING ’}’
<MOD_of_Interrupt>       := ’MOD’ ’{’ <MOD> ’}’
INT                      := "[0-9]+"
```

Execution cycle count for the interrupt is defined. In the micro-operation description of interrupts, interrupt handling operations of the processor such as setting specific values to special registers and jumping to the interrupt handler routine, are described. The syntax of Interrupt condition `<Condition>` and micro operation description `<MOD` are explained in Appendix A.7.

A.7 Micro-operation Description

Micro-operation description is used to describe clock based behavior of instructions and interrupts. Micro-operation description of n clock instruction (or interrupt) is described as follows:

```
clk(1){"<Micro-Op>"},
clk(2){"<Micro-Op>"},
          :
clk(k){"<Micro-Op>"},
          :
clk(n){"<Micro-Op>"}
```

Behavior of the instruction (or interrupt) in the k-th clock is described with “clk(k)”. Micro-operation of each clock consists of the following elements:

- Variable,

- Constant,
- Storage,
- Operand,
- Function,
- Assignment statement,
- If-statement,
- Decode designation.

```
<Micro-Op> := <Assignment_Statement> ';' |
              <Function ';' |
              <If_Statement> ';' |
              <Decode_Statement> ';'
```

A.7.1 Variable

```
<Variable> := VAR
VAR        := $[a-zA-Z][a-zA-Z0-9_]*
```

Variables are declared implicitly in assignment statements. The scope of variable is an instruction in which the variable is described. Assignment for a variable is allowed only once. Right value of the assignment can be referred in the same stage and in the following stages. The variable is implemented with net when the variable is referred in the same stage. The variable is implemented with pipeline register when the variable is referred in the following stage.

A.7.2 Constant

```
<Constant> := SignalBit | BitVector
SingleBit  := \'0\' | \'1\'
BitVector  := "[01]+"
```

Constant in binary expression is used in the micro-operation description. Single bit constant is quoted by single quote. Plural bits constant is quoted by double quote. Constant value is referred in assignment statement, conditional expression of if-statement, resource operations and index of addressed storage.

A.7.3 Storage

```
<Storage>       := <Resource_Name> [ <Address> ]
<Resource_Name> := NAME
<Address>       := <Right_Part>
NAME            := [a-zA-Z][a-zA-Z0-9_]*
```

Declared registers, memory access units, register files and so on in the resource declaration part, are referred as storage units. For an addressed storage unit, location is referred by an index. An index is quoted by "[ ]". The contents value of storage unit is referred when it is in a parameter of resource operation, right part of the assignment statement and conditional expression of if-statement. The contents value of storage unit is changed to assigned value when it is in a left part of assignment statement. The value of the storage unit is replaced synchronously with the transition timing of the stage.

A.7.4 Operands

```
<Operand>    := <Field_Name>
<Field_Name> := NAME
NAME         := [a-zA-Z][a-zA-Z0-9_]*
```

Certain bit field of the instruction register is referred by an operand field name, which is specified in instruction format definition part. The field name and bit field of an operand are defined in instruction type definition part.

A.7.5 Function

```
<Function>       := <Resource_Name> '.'  <Function_Name> '(' <Parameter_List> ')'
<Resource_Name>  := NAME
<Function_Name>  := NAME
<Parameter_List> := [ <Parameter> { ',' <Parameter> } ]
<Parameter>      := <Right_Part>
NAME             := [a-zA-Z][a-zA-Z0-9_]*
```

Functions indicate operations of resources. Input data of the operation are described as parameters. Output results of the operation are assigned to variables and storage units with assignment statement. If the function has no output, the statement consists of function expression only.

A.7.6 Assignment statement

```
<Assignment_Statement> := <Left_Parts> ':=' <Right_Part>
<Left_Parts>           := <Left_Part> | '(' <Left_Part> { ',' <left_part> } ')'
<Left_Part>            := <Storage> | <Variable>
<Right_Part>           := <Term> {'&' <Term>}
<Term>i                := <Storage> [<Bit_Select>] | <Variable> [<Bit_Select>] |
                          <Constant> | <Function> | <Operand>
```

In an assignment statement, right part values are assigned to the storages and variables appeared in left part. Right hand includes storage units, variables, operands, constant and function and concatenate of them.

A.7.7 If-statement

```
<If_Statement> := 'if' '(' <Condition> ')' <Then_Phase>
                  [ <Else_Phase> ] 'end if'
<Condition>    := <Equation> | <Expression1>
<Expression1>  := <Expression2> { '||' <Expression2> }
<Expression2>  := <Expression3> { '&&' <Expression3> }
<Expression3>  := '(' <Condition> ')' | 'not' <Expression3>
<Equation>     := <Right_Part> | '=' <Right_Part>
<Then_Phase>   := 'then' <Micro-Op>
<Else_Phase>   := 'else' <Micro-Op>
```

If the condition holds, then-phase is executed. If the condition does not hold, else-phase is executed. The condition of the if-statement is represented in boolean expression of equations, which compare value of variables, constants, storages, operands and functions. The order of priority of boolean operators is 'not', '&&' and '||'.

A.7.8 Decode statement

```
<Decode_Statement>      := 'DECODE' '(' <Instruction_Register> ')'
<Instruction_Register> := <Storage>
```

Decode statement indicates instruction decode stage and instruction register. Instruction decode stage is the stage where the decode statement is described. The storage unit in the decode statement indicates instruction register.

Appendix B

Processor Specification of PEAS R3K

```
Design{
Version{"2.0"},

Abstract_level_architecture{
Fhm_workname{"peas"},
construciton{Goal_frequency{"50"},
Goal_area{"40"},
Goal_delay{"20"},
Goal_power_S{"1000"},
Goal_power_D{"1000"},
Goal_exec{"28000"}},
CPU_type{"Pipeline"},
Pipeline_architecture{Number_of_stages{"5"},
Number_of_common_stages{"0"},
Number_of_phases_par_stage{"1"},
stage{stage1{"IF","1"},
stage2{"ID","1"},
stage3{"EXE","1"},
stage4{"MEM","1"},
stage4{"WB","1"}},
Multi_cycle_interlock{"Use"},
Data_hazard_interlock{"Use"},
Register_bypass{"Use"},
Memory_bypass{"Use"},
Delayed_branch{"Yes"},
Number_of_exec_delayed_slot{number{"1"}}}},

Port_declaration{
entity_name{"CPU"},Port{"clk"{direction{"in"},signal_type{"std_logic"},signal_attribute{"clock"}},
"intn"{direction{"in"},signal_type{"std_logic_vector(2 downto 0)"},signal_attribute{"$"}},
"int"{direction{"in"},signal_type{"std_logic"},signal_attribute{"$"}},
"rst"{direction{"in"},signal_type{"std_logic"},signal_attribute{"$"}},
"instAB"{direction{"out"},signal_type{"std_logic_vector(31 downto 0)"}
,signal_attribute{"instruction_memory_address_bus"}},
"instDB"{direction{"in"},signal_type{"std_logic_vector(31 downto 0)"}
,signal_attribute{"instruction_memory_data_bus"}},
"dataAB"{direction{"out"},signal_type{"std_logic_vector(31 downto 0)"}
,signal_attribute{"data_memory_address_bus"}},
"dataDB"{direction{"inout"},signal_type{"std_logic_vector(31 downto 0)"}
,signal_attribute{"data_memory_data_bus"}},
"we"{direction{"out"},signal_type{"std_logic_vector(3 downto 0)"}
,signal_attribute{"data_memory_write_enable"}}
}},

Instruction_type{
sub_field_name{NO_VLIW{width{"31","16"},type{"Btype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"name"{"bfunct"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},
"Jtype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"target"},width{"25","0"}}
},
"Rtype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Operand"{"name"{"shamt"},width{"10","6"}},
"OP-code"{"name"{"rfunct"},width{"5","0"}}
},
"R1type"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"name"{"rfunct"},width{"5","0"}}
},
"Itype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},
"LStype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},
"R2type"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Reserved"{"binary"{"0000000000"},width{"15","6"}},
"OP-code"{"name"{"rfunct"},width{"5","0"}}
},
"MFtype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Reserved"{"binary"{"0000000000"},width{"25","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"name"{"mffunct"},width{"5","0"}}
},
"MTtype"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Reserved"{"binary"{"000000000000000"},width{"20","6"}},
"OP-code"{"name"{"mtfunct"},width{"5","0"}}
},
"B1type"{
"OP-code"{"name"{"opecode"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
}
}}}},

Instruction{NO_VLIW{"ADD"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100000"},width{"5","0"}}
},"ADDI"{type{"Itype"},"OP-code"{"binary"{"001000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
```

```
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"ADDIU"{type{"Itype"},"OP-code"{"binary"{"001001"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"ADDU"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100001"},width{"5","0"}}
},"ANDI"{type{"Itype"},"OP-code"{"binary"{"001100"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"BGEZ"{type{"Btype"},"OP-code"{"binary"{"000001"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"00001"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BGEZAL"{type{"Btype"},"OP-code"{"binary"{"000001"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"10001"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BGTZ"{type{"Btype"},"OP-code"{"binary"{"000111"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"00000"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BLEZ"{type{"Btype"},"OP-code"{"binary"{"000110"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"00000"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BLTZ"{type{"Btype"},"OP-code"{"binary"{"000001"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"00000"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BLTZAL"{type{"Btype"},"OP-code"{"binary"{"000001"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"OP-code"{"binary"{"10000"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"IAND"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100100"},width{"5","0"}}
},"INOR"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100111"},width{"5","0"}}
},"IOR"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100101"},width{"5","0"}}
},"ISUB"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100010"},width{"5","0"}}
},"IXOR"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100110"},width{"5","0"}}
},"J"{type{"Jtype"},"OP-code"{"binary"{"000010"},width{"31","26"}},
"Operand"{"name"{"target"},width{"25","0"}}
},"JAL"{type{"Jtype"},"OP-code"{"binary"{"000011"},width{"31","26"}},
"Operand"{"name"{"target"},width{"25","0"}}
},"JALR"{type{"Rtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Operand"{"name"{"shamt"},width{"10","6"}},
"OP-code"{"binary"{"001001"},width{"5","0"}}
},"JR"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"001000"},width{"5","0"}}
},"LB"{type{"LStype"},"OP-code"{"binary"{"100000"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"LBU"{type{"LStype"},"OP-code"{"binary"{"100100"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"LH"{type{"LStype"},"OP-code"{"binary"{"100001"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"LHU"{type{"LStype"},"OP-code"{"binary"{"100101"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"LUI"{type{"Itype"},"OP-code"{"binary"{"001111"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"LW"{type{"LStype"},"OP-code"{"binary"{"100011"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"ORI"{type{"Itype"},"OP-code"{"binary"{"001101"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"SB"{type{"LStype"},"OP-code"{"binary"{"101000"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"SH"{type{"LStype"},"OP-code"{"binary"{"101001"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"SLL"{type{"Rtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Operand"{"name"{"shamt"},width{"10","6"}},
"OP-code"{"binary"{"000000"},width{"5","0"}}
},"SLLV"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"000100"},width{"5","0"}}
},"SLT"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
```

```
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"101010"},width{"5","0"}}
},"SLTI"{type{"Itype"},"OP-code"{"binary"{"001010"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"SLTIU"{type{"Itype"},"OP-code"{"binary"{"001011"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"SLTU"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"101011"},width{"5","0"}}
},"SRA"{type{"Rtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Operand"{"name"{"shamt"},width{"10","6"}},
"OP-code"{"binary"{"000011"},width{"5","0"}}
},"SRAV"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"000111"},width{"5","0"}}
},"SRL"{type{"Rtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Operand"{"name"{"shamt"},width{"10","6"}},
"OP-code"{"binary"{"000010"},width{"5","0"}}
},"SRLV"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"000110"},width{"5","0"}}
},"SUBU"{type{"R1type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"100011"},width{"5","0"}}
},"SW"{type{"LStype"},"OP-code"{"binary"{"101011"},width{"31","26"}},
"Operand"{"name"{"base"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"XORI"{type{"Itype"},"OP-code"{"binary"{"001110"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"immediate"},width{"15","0"}}
},"MULT"{type{"R2type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Reserved"{"binary"{"0000000000"},width{"15","6"}},
"OP-code"{"binary"{"011000"},width{"5","0"}}
},"MULTU"{type{"R2type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Reserved"{"binary"{"0000000000"},width{"15","6"}},
"OP-code"{"binary"{"011001"},width{"5","0"}}
},"DIV"{type{"R2type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Reserved"{"binary"{"0000000000"},width{"15","6"}},
"OP-code"{"binary"{"011010"},width{"5","0"}}
```

```
},"DIVU"{type{"R2type"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Reserved"{"binary"{"0000000000"},width{"15","6"}},
"OP-code"{"binary"{"011011"},width{"5","0"}}
},"MFHI"{type{"MFtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Reserved"{"binary"{"0000000000"},width{"25","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"010000"},width{"5","0"}}
},"MFLO"{type{"MFtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Reserved"{"binary"{"0000000000"},width{"25","16"}},
"Operand"{"name"{"rd"},width{"15","11"}},
"Reserved"{"binary"{"00000"},width{"10","6"}},
"OP-code"{"binary"{"010010"},width{"5","0"}}
},"MTHI"{type{"MTtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Reserved"{"binary"{"000000000000000"},width{"20","6"}},
"OP-code"{"binary"{"010001"},width{"5","0"}}
},"MTLO"{type{"MTtype"},"OP-code"{"binary"{"000000"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Reserved"{"binary"{"000000000000000"},width{"20","6"}},
"OP-code"{"binary"{"010011"},width{"5","0"}}
},"BEQ"{type{"B1type"},"OP-code"{"binary"{"000100"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
},"BNE"{type{"B1type"},"OP-code"{"binary"{"000101"},width{"31","26"}},
"Operand"{"name"{"rs"},width{"25","21"}},
"Operand"{"name"{"rt"},width{"20","16"}},
"Operand"{"name"{"offset"},width{"15","0"}}
}}},

Operation{NO_VLIW{}},

Resource{"IR"{class{"register"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
edge_trigger{"positive"}}}
,"HI"{class{"register"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
edge_trigger{"positive"}}}
,"LO"{class{"register"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
edge_trigger{"positive"}}}
,"CSW"{class{"register"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
edge_trigger{"positive"}}}
,"GPR"{class{"registerfile"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
num_register{"32"},
num_read_port{"2"},
num_write_port{"1"}}}
,"ADD0"{class{"adder"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
algorithm{"cla"}}}
,"ALU0"{class{"alu"},classpath{""},
parameter{
```

```
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
algorithm{"cla"}}}
,"DIV0"{class{"divider"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
algorithm{"seq"},
adder_algorithm{"cla"},
data_type{"two_complement"}}}
,"SFT0"{class{"barrelshifter"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"RT"}},
bit_width{"32"}}}
,"EXT0"{class{"extender"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"16"}}}
,"MUL0"{class{"multiplier"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
algorithm{"seq"},
adder_algorithm{"cla"},
data_type{"two_complement"}}}
,"PC"{class{"pcu"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"},
increment_step{"4"},
adder_algorithm{"cla"}}}
,"IMEM"{class{"imcu"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"}}}
,"DMEM"{class{"dmcu"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"32"}}}
,"NOT0"{class{"not"},classpath{""},
parameter{
abstraction_level{for_simulation{"Behavior"},for_synthesis{"Gate"}},
bit_width{"1"}}}
},

Exception{"reset"{Condition{"rst='1'"},Type{"External"},Cycles{"1"},
Behavior{"-- reset behavior"},Assert{""},Comment{""},
MOD{clk(1){"PC.reset(); GPR.reset();
CSW.reset(); HI.reset();
LO.reset(); IR.reset();"}
}},
"init0"{Condition{"int = '1' and intn = \"000\""},Type{"External"},Cycles{"1"},
Behavior{"-- Interrupt behavior"},Assert{""},Comment{""},
MOD{clk(1){"CSW := PC;"}
}}
},

MOT{mnemonic{"ADD"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.add($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"ADDI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm:=EXT0.sign(immediate);"},
clk(3){"($result, $flag):=ALU0.add($rs,$imm);"},
clk(4){""},
clk(5){"GPR[rt]:=$result;"}
}
,"ADDIU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm:=EXT0.sign(immediate);"},
clk(3){"($result, $flag):=ALU0.add($rs,$imm);"},
clk(4){""},
clk(5){"GPR[rt]:=$result;"}
}
,"ADDU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.add($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"ANDI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm:=EXT0.zero(immediate);"},
clk(3){"($result, $flag):=ALU0.and($rs,$imm);"},
clk(4){""},
clk(5){"GPR[rt]:=$result;"}
}
,"BGEZ"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
$flag := ALU0.cmpz($rs);
if($rs(31)='0') then PC:=$target; end if;"},
clk(4){""},
clk(5){""}
}
,"BGEZAL"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
if($rs(31)='0')  then PC:=$target; end if;
$pc2 := PC;"},
clk(4){""},
clk(5){"GPR[\"11111\"]:=$pc2;"}
}
,"BGTZ"{clk(1){"IR := IMEM[PC];
```

```
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
$flag:=ALU0.cmpz($rs);
if(($rs(31) = '0') && ($flag(2) = '0')) then PC:= $target; end if;"},
clk(4){""},
clk(5){""}
}
,"BLEZ"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
$flag:=ALU0.cmpz($rs);
if(($rs(31) = '1') || ($flag(2) = '1')) then PC:= $target; end if;"},
clk(4){""},
clk(5){""}
}
,"BLTZ"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
if($rs(31)='1') then PC:=$target; end if;"},
clk(4){""},
clk(5){""}
}
,"BLTZAL"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
if($rs(31)='1')  then PC:=$target; end if;
$pc2 := PC;"},
clk(4){""},
clk(5){"GPR[\"11111\"]:=$pc2;"}
}
,"IAND"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.and($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"INOR"{clk(1){"IR := IMEM[PC];
```

```
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.nor($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"IOR"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);
;"},
clk(3){"($result, $flag) := ALU0.or($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"ISUB"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.sub($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"IXOR"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.xor($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"J"{clk(1){"$pc:=PC;

IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$target := $pc(31 downto 28) & IR(25 downto 0) & \"00\";"},
clk(3){"PC := $target;"},
clk(4){""},
clk(5){""}
}
,"JAL"{clk(1){"$pc := PC;

IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$target := $pc(31 downto 28) & IR(25 downto 0) & \"00\";"},
clk(3){"PC := $target;
$pc2 := PC;"},
clk(4){""},
clk(5){"GPR[\"11111\"] := $pc2;"}
}
,"JALR"{clk(1){"$pc := PC;

IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);"},
clk(3){"PC:=$rs;
$pc2:=PC;"},
```

```
clk(4){""},
clk(5){"GPR[\"11111\"]:=$pc2;"}
}
,"JR"{clk(1){"$pc:=PC;

IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);"},
clk(3){"PC:=$rs;"},
clk(4){""},
clk(5){""}
}
,"LB"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"($data, $addr_err):=DMEM.lb($target);"},
clk(5){"GPR[rt]:=$data;"}
}
,"LBU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"($data, $addr_err):=DMEM.lbu($target);"},
clk(5){"GPR[rt]:=$data;"}
}
,"LH"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"($data, $addr_err):=DMEM.lh($target);"},
clk(5){"GPR[rt]:=$data;"}
}
,"LHU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"($data, $addr_err):=DMEM.lhu($target);"},
clk(5){"GPR[rt]:=$data;"}
}
,"LUI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$imm:=immediate & \"0000000000000000\";"},
clk(3){""},
clk(4){""},
clk(5){"GPR[rt]:=$imm;"}
}
,"LW"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"($data, $addr_err):=DMEM.read($target);"},
clk(5){"GPR[rt]:=$data;"}
}
,"ORI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm:=EXT0.zero(immediate);"},
clk(3){"($result, $flag):=ALU0.or($rs,$imm);"},
clk(4){""},
clk(5){"GPR[rt]:=$result;"}
}
,"SB"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);
$rt:=GPR.read1(rt);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"$addr_err:=DMEM.sb($target,$rt);"},
clk(5){""}
}
,"SH"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);
$rt:=GPR.read1(rt);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"$addr_err:=DMEM.sh($target,$rt);"},
clk(5){""}
}
,"SLL"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.sll($rt,shamt);"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SLLV"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$shamt:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.sll($rt, $shamt(4 downto 0));"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SLT"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"$flag:=ALU0.cmp($rs,$rt);
$result := \"0000000000000000000000000000000\" & $flag(1);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"SLTI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);
$imm:=EXT0.sign(immediate);"},
clk(3){"$flag:=ALU0.cmp($rs, $imm);
```

```
$result := \"0000000000000000000000000000000\" & $flag(1);"},
clk(4){""},
clk(5){"GPR[rt] := $result;"}
}
,"SLTIU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);
$imm:=EXT0.sign(immediate);"},
clk(3){"$flag:=ALU0.cmp($rs, $imm);
$result := \"0000000000000000000000000000000\" & NOT0.nt($flag(3));"},
clk(4){""},
clk(5){"GPR[rt]:= $result;"}
}
,"SLTU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"$flag:=ALU0.cmpu($rs,$rt);
$result := \"0000000000000000000000000000000\" & NOT0.nt($flag(3));"},
clk(4){""},
clk(5){"GPR[rd]:= $result;"}
}
,"SRA"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.sra($rt,shamt);"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SRAV"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$shamt:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.sra($rt, $shamt(4 downto 0));"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SRL"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.srl($rt,shamt);"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SRLV"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$shamt:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"$result:=SFT0.srl($rt, $shamt(4 downto 0));"},
clk(4){""},
clk(5){"GPR[rd]:=$result;"}
}
,"SUBU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs := GPR.read0(rs);
$rt := GPR.read1(rt);"},
clk(3){"($result, $flag) := ALU0.sub($rs, $rt);"},
clk(4){""},
clk(5){"GPR[rd] := $result;"}
}
,"SW"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$offset :=EXT0.sign(offset);
$base:=GPR.read0(base);
$rt:=GPR.read1(rt);"},
clk(3){"($target, $flag):=ALU0.add($base,$offset);"},
clk(4){"$addr_err:=DMEM.write($target,$rt);"},
clk(5){""}
}
,"XORI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$imm:=EXT0.zero(immediate);"},
clk(3){"($result, $flag):=ALU0.xor($rs,$imm);"},
clk(4){""},
clk(5){"GPR[rt]:=$result;"}
}
,"MULT"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"($result, $flag):=MUL0.mul($rs,$rt);"},
clk(4){""},
clk(5){"HI:=$result(63 downto 32);
LO:=$result(31 downto 0);"}
}
,"MULTU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"($result, $flag):=MUL0.mulu($rs,$rt);"},
clk(4){""},
clk(5){"HI:=$result(63 downto 32);
LO:=$result(31 downto 0);"}
}
,"DIV"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"($q,$r,$flag):=DIV0.div($rs,$rt);"},
clk(4){""},
clk(5){"HI:=$r; LO:=$q;"}
}
,"DIVU"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);

$rs:=GPR.read0(rs);
$rt:=GPR.read1(rt);"},
clk(3){"($q,$r,$flag):=DIV0.divu($rs,$rt);"},
clk(4){""},
clk(5){"HI:=$r; LO:=$q;"}
}
,"MFHI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);"},
```

```
clk(3){"$hi:=HI;"},
clk(4){""},
clk(5){"GPR[rd]:=$hi;"}
}
,"MFLO"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);"},
clk(3){"$lo:=LO;"},
clk(4){""},
clk(5){"GPR[rd]:=$lo;"}
}
,"MTHI"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);"},
clk(3){""},
clk(4){""},
clk(5){"HI:=$rs;"}
}
,"MTLO"{clk(1){"IR := IMEM[PC];
PC.inc();"},
clk(2){"DECODE(IR);
$rs:=GPR.read0(rs);"},
clk(3){""},
clk(4){""},
clk(5){"LO:=$rs;"}
}
,"BEQ"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);
$rt:=GPR.read1(rt);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
$flag:=ALU0.cmp($rs,$rt);
if($flag(2)='1') then PC:=$target; end if;"},
clk(4){""},
clk(5){""}
}
,"BNE"{clk(1){"IR := IMEM[PC];
PC.inc();

$pc:=PC;"},
clk(2){"DECODE(IR);
$rt:=GPR.read1(rt);

$rs:=GPR.read0(rs);
$imm := EXT0.sign(offset);"},
clk(3){"$offset := $imm(29 downto 0) & \"00\";
$target := ADD0.add($pc, $offset);
$flag:=ALU0.cmp($rs,$rt);
if($flag(2)='0') then PC:=$target; end if;"},
clk(4){""},
clk(5){""}
}
}}

}
```

Appendix C

Synthesis Result of PEAS R3K Processor

C.1 VHDL Descriptionf of PEAS R3K Datapath

```
library IEEE;
  use IEEE.std_logic_1164.all;

entity CPU is
  port (
    clk : in std_logic;
    intn : in std_logic_vector(2 downto 0);
    int : in std_logic;
    rst : in std_logic;
    instAB : out std_logic_vector(31 downto 0);
    instDB : in std_logic_vector(31 downto 0);
    dataAB : out std_logic_vector(31 downto 0);
    dataDB : inout std_logic_vector(31 downto 0);
    we : out std_logic_vector(3 downto 0));
end CPU;

architecture syn of CPU is
  component cpu_ctrl
    port (
      instDB : in std_logic_vector(31 downto 0);
      rst : in std_logic;
      int : in std_logic;
      intn : in std_logic_vector(2 downto 0);
      clk : in std_logic;
      IR_data_out : in std_logic_vector(31 downto 0);
      MUL0_fin : in std_logic;
      DIV0_flag : in std_logic_vector(1 downto 0);
      CSW_enb : out std_logic;
      CSW_rst : out std_logic;
      reg39_enb : out std_logic;
      reg38_enb : out std_logic;
      reg37_enb : out std_logic;
      reg36_enb : out std_logic;
      reg35_enb : out std_logic;
      reg34_enb : out std_logic;
      reg33_enb : out std_logic;
      reg32_enb : out std_logic;
      reg31_enb : out std_logic;
      reg30_enb : out std_logic;
      reg29_enb : out std_logic;
      reg28_enb : out std_logic;
      reg27_enb : out std_logic;
      reg26_enb : out std_logic;
      reg25_enb : out std_logic;
      reg24_enb : out std_logic;
      reg23_enb : out std_logic;
      reg22_enb : out std_logic;
      reg21_enb : out std_logic;
      reg20_enb : out std_logic;
      sel19_ctrl : out std_logic_vector(1 downto 0);
      sel18_ctrl : out std_logic_vector(1 downto 0);
      sel17_ctrl : out std_logic_vector(0 downto 0);
      sel16_ctrl : out std_logic_vector(0 downto 0);
      sel15_ctrl : out std_logic_vector(0 downto 0);
      sel14_ctrl : out std_logic_vector(2 downto 0);
      sel13_ctrl : out std_logic_vector(0 downto 0);
      sel12_ctrl : out std_logic_vector(0 downto 0);
      sel11_ctrl : out std_logic_vector(0 downto 0);
      sel10_ctrl : out std_logic_vector(0 downto 0);
      DIV0_ctrl : out std_logic;
      LO_enb : out std_logic;
      LO_rst : out std_logic;
      HI_enb : out std_logic;
      HI_rst : out std_logic;
      MUL0_start : out std_logic;
      MUL0_ctrl : out std_logic;
      SFT0_mode : out std_logic_vector(1 downto 0);
      DMEM_ext_ctrl : out std_logic;
      DMEM_ac_ctrl : out std_logic_vector(1 downto 0);
      DMEM_req : out std_logic;
      DMEM_rw : out std_logic;
      EXT0_ctrl : out std_logic;
      ALU0_ctrl : out std_logic_vector(4 downto 0);
      ALU0_cin : out std_logic;
      GPR_w_enb0 : out std_logic;
      GPR_reset : out std_logic;
      IR_enb : out std_logic;
      IR_rst : out std_logic;
      PC_hold : out std_logic;
      PC_reset : out std_logic;
      PC_load : out std_logic;
      reg20_data_out : in std_logic_vector(31 downto 0);
      sys4_p0 : in std_logic;
      sys2_p0 : in std_logic;
      ALU0_flag : in std_logic_vector(3 downto 0));
  end component;

component pcu_17
  generic (W : integer := 32;
   S : integer := 4);
  port(
    clk : in std_logic;
  load : in std_logic;
  reset : in std_logic;
  hold : in std_logic;
  data : in std_logic_vector(W-1 downto 0);
  q : out std_logic_vector(W-1 downto 0));
end component;
component imcu_18
  generic (W : integer := 32);
   PORT(
      addr   : in     std_logic_vector(W-1 downto 0 ) ;
      data   : out     std_logic_vector(W-1 downto 0 ) ;
      m_addr : out    std_logic_vector(W-1 downto 0 ) ;
      m_data : in  std_logic_vector(W-1 downto 0 )
   );
end component;
component register_9
  generic (W : integer := 32);
  port (clk      : in std_logic;
        rst      : in std_logic;
        enb      : in std_logic;
        data_in  : in std_logic_vector(W-1 downto 0);
        data_out : out std_logic_vector(W-1 downto 0) );
end component;
component registerfile_10
```

```
  generic (W : integer := 32);
  port ( clock      : in std_logic;
         reset      : in  std_logic;
         w_enb0     : in  std_logic;
         w_sel0     : in  std_logic_vector( 4 downto 0);
         data_in0   : in  std_logic_vector(W-1 downto 0);
         r_sel0     : in  std_logic_vector( 4 downto 0);
         r_sel1     : in  std_logic_vector( 4 downto 0);
         data_out0  : out std_logic_vector(W-1 downto 0);
         data_out1  : out std_logic_vector(W-1 downto 0) );
end component;
component alu_12
  generic (W : integer := 32);
  port (a, b  : in std_logic_vector(W-1 downto 0);
        cin     : in std_logic;
        ctrl  : in std_logic_vector(4 downto 0);
        result  : out std_logic_vector(W-1 downto 0);
        flag  : out std_logic_vector(3 downto 0) );
end component;
component extender_15
  generic (W : integer := 16);
  port (data_in  : in std_logic_vector(W-1 downto 0);
        ctrl     : in std_logic;
        data_out : out std_logic_vector(2*W-1 downto 0));
end component;
component adder_11
  generic(W : integer := 32);
  port (a, b   : in std_logic_vector(W-1 downto 0);
        cin    : in std_logic;
        result : out std_logic_vector(W-1 downto 0);
        cout   : out std_logic);
end component;
component dmcu_19
  port ( rw : in std_logic;
       req : in std_logic;
       addr : in std_logic_vector(31 downto 0);
       i_data : out std_logic_vector(31 downto 0);
       o_data : in std_logic_vector(31 downto 0);
       ac_ctrl : in std_logic_vector(1 downto 0);
       ext_ctrl : in std_logic;
       addr_err : out std_logic;
       we : out std_logic_vector(3 downto 0);
       m_addr : out std_logic_vector(31 downto 0);
       m_data : inout std_logic_vector(31 downto 0));
end component;
component barrelshifter_14
  generic(W : integer := 32);
  port (data_in  : in std_logic_vector(W-1 downto 0);
        mode     : in std_logic_vector(1 downto 0);
        ctrl     : in std_logic_vector(4 downto 0);
        data_out : out std_logic_vector(W-1 downto 0));
end component;
component not_20
  port ( data_in  : in   std_logic;
         data_out : out  std_logic);
end component;
component multiplier_16
  generic (W : integer := 32);
  port (clk    : in std_logic;
        reset  : in std_logic;
        a, b   : in std_logic_vector(W-1 downto 0);
        ctrl   : in std_logic;
        start  : in std_logic;
        result : out std_logic_vector(2*W-1 downto 0);
        fin    : out std_logic);
end component;
component divider_13
  generic (W : integer := 32);
  port (clk      : in std_logic;
        a, b     : in std_logic_vector(W-1 downto 0);
        ctrl     : in std_logic;
        result0  : out std_logic_vector(W-1 downto 0);
        result1  : out std_logic_vector(W-1 downto 0);
        flag     : out std_logic_vector(1 downto 0));
end component;
component selector_21
  generic (w    : integer := 32;
   n    : integer := 2;
           logn : integer := 1);
  port (data_in0 : in std_logic_vector(w-1 downto 0);
        data_in1 : in std_logic_vector(w-1 downto 0);
ctrl     : in std_logic_vector(logn-1 downto 0);
data_out : out std_logic_vector(w-1 downto 0));
end component;
component selector_22
  generic (w    : integer := 5;
   n    : integer := 2;
           logn : integer := 1);
  port (data_in0 : in std_logic_vector(w-1 downto 0);
        data_in1 : in std_logic_vector(w-1 downto 0);
ctrl     : in std_logic_vector(logn-1 downto 0);
data_out : out std_logic_vector(w-1 downto 0));
end component;
component selector_23
  generic (w    : integer := 32;
   n    : integer := 8;
           logn : integer := 3);
  port (data_in0 : in std_logic_vector(w-1 downto 0);
        data_in1 : in std_logic_vector(w-1 downto 0);
        data_in2 : in std_logic_vector(w-1 downto 0);
        data_in3 : in std_logic_vector(w-1 downto 0);
        data_in4 : in std_logic_vector(w-1 downto 0);
        data_in5 : in std_logic_vector(w-1 downto 0);
        data_in6 : in std_logic_vector(w-1 downto 0);
        data_in7 : in std_logic_vector(w-1 downto 0);
ctrl     : in std_logic_vector(logn-1 downto 0);
data_out : out std_logic_vector(w-1 downto 0));
end component;
component selector_24
  generic (w    : integer := 32;
   n    : integer := 3;
           logn : integer := 2);
  port (data_in0 : in std_logic_vector(w-1 downto 0);
        data_in1 : in std_logic_vector(w-1 downto 0);
        data_in2 : in std_logic_vector(w-1 downto 0);
ctrl     : in std_logic_vector(logn-1 downto 0);
data_out : out std_logic_vector(w-1 downto 0));
end component;
component pipereg_25
  generic (W : integer := 32);
  port (clk      : in std_logic;
        rst      : in std_logic;
        enb      : in std_logic;
        data_in  : in std_logic_vector(W-1 downto 0);
        data_out : out std_logic_vector(W-1 downto 0) );
end component;
component pipereg_26
  generic (W : integer := 30);
  port (clk      : in std_logic;
        rst      : in std_logic;
        enb      : in std_logic;
        data_in  : in std_logic_vector(W-1 downto 0);
        data_out : out std_logic_vector(W-1 downto 0) );
end component;
component pipereg_27
  generic (W : integer := 5);
  port (clk      : in std_logic;
        rst      : in std_logic;
```

```
        enb      : in std_logic;
        data_in  : in std_logic_vector(W-1 downto 0);
        data_out : out std_logic_vector(W-1 downto 0) );
end component;
  signal d_CSW_data_out : std_logic_vector(31 downto 0);
  signal d_reg39_data_out : std_logic_vector(31 downto 0);
  signal d_reg38_data_out : std_logic_vector(31 downto 0);
  signal d_reg37_data_out : std_logic_vector(31 downto 0);
  signal d_reg36_data_out : std_logic_vector(31 downto 0);
  signal d_reg35_data_out : std_logic_vector(4 downto 0);
  signal d_reg34_data_out : std_logic_vector(31 downto 0);
  signal d_reg33_data_out : std_logic_vector(31 downto 0);
  signal d_reg32_data_out : std_logic_vector(31 downto 0);
  signal d_reg31_data_out : std_logic_vector(4 downto 0);
  signal d_reg30_data_out : std_logic_vector(4 downto 0);
  signal d_reg29_data_out : std_logic_vector(4 downto 0);
  signal d_reg28_data_out : std_logic_vector(31 downto 0);
  signal d_reg27_data_out : std_logic_vector(31 downto 0);
  signal d_reg26_data_out : std_logic_vector(31 downto 0);
  signal d_reg25_data_out : std_logic_vector(31 downto 0);
  signal d_reg24_data_out : std_logic_vector(31 downto 0);
  signal d_reg23_data_out : std_logic_vector(31 downto 0);
  signal d_reg22_data_out : std_logic_vector(31 downto 0);
  signal d_reg21_data_out : std_logic_vector(29 downto 0);
  signal d_reg20_data_out : std_logic_vector(31 downto 0);
  signal d_sel19_data_out : std_logic_vector(31 downto 0);
  signal d_sel18_data_out : std_logic_vector(31 downto 0);
  signal d_sel17_data_out : std_logic_vector(4 downto 0);
  signal d_sel16_data_out : std_logic_vector(31 downto 0);
  signal d_sel15_data_out : std_logic_vector(31 downto 0);
  signal d_sel14_data_out : std_logic_vector(31 downto 0);
  signal d_sel13_data_out : std_logic_vector(4 downto 0);
  signal d_sel12_data_out : std_logic_vector(4 downto 0);
  signal d_sel11_data_out : std_logic_vector(31 downto 0);
  signal d_sel10_data_out : std_logic_vector(31 downto 0);
  signal d_DIV0_flag : std_logic_vector(1 downto 0);
  signal d_DIV0_result1 : std_logic_vector(31 downto 0);
  signal d_DIV0_result0 : std_logic_vector(31 downto 0);
  signal d_LO_data_out : std_logic_vector(31 downto 0);
  signal d_HI_data_out : std_logic_vector(31 downto 0);
  signal d_MUL0_fin : std_logic;
  signal d_MUL0_result : std_logic_vector(63 downto 0);
  signal d_sys10_p0 : std_logic_vector(31 downto 0);
  signal d_NOT0_data_out : std_logic;
  signal d_sys9_p0 : std_logic_vector(31 downto 0);
  signal d_sys8_p0 : std_logic_vector(30 downto 0);
  signal d_SFT0_data_out : std_logic_vector(31 downto 0);
  signal d_sys7_p0 : std_logic_vector(31 downto 0);
  signal d_sys6_p0 : std_logic_vector(15 downto 0);
  signal d_DMEM_addr_err : std_logic;
  signal d_DMEM_i_data : std_logic_vector(31 downto 0);
  signal d_sys5_p0 : std_logic_vector(31 downto 0);
  signal d_sys4_p0 : std_logic;
  signal d_sys3_p0 : std_logic_vector(4 downto 0);
  signal d_sys2_p0 : std_logic;
  signal d_ADD0_cout : std_logic;
  signal d_ADD0_result : std_logic_vector(31 downto 0);
  signal d_sys1_p0 : std_logic_vector(31 downto 0);
  signal d_sys0_p0 : std_logic_vector(1 downto 0);
  signal d_EXT0_data_out : std_logic_vector(31 downto 0);
  signal d_ALU0_flag : std_logic_vector(3 downto 0);
  signal d_ALU0_result : std_logic_vector(31 downto 0);
  signal d_GPR_data_out1 : std_logic_vector(31 downto 0);
  signal d_GPR_data_out0 : std_logic_vector(31 downto 0);
  signal d_IR_data_out : std_logic_vector(31 downto 0);
  signal d_IMEM_data : std_logic_vector(31 downto 0);
  signal d_PC_q : std_logic_vector(31 downto 0);
  signal c_PC_load : std_logic;
  signal c_PC_reset : std_logic;
  signal c_PC_hold : std_logic;
  signal c_IR_rst : std_logic;
  signal c_IR_enb : std_logic;
  signal c_GPR_reset : std_logic;
  signal c_GPR_w_enb0 : std_logic;
  signal c_ALU0_cin : std_logic;
  signal c_ALU0_ctrl : std_logic_vector(4 downto 0);
  signal c_EXT0_ctrl : std_logic;
  signal c_DMEM_rw : std_logic;
  signal c_DMEM_req : std_logic;
  signal c_DMEM_ac_ctrl : std_logic_vector(1 downto 0);
  signal c_DMEM_ext_ctrl : std_logic;
  signal c_SFT0_mode : std_logic_vector(1 downto 0);
  signal c_MUL0_ctrl : std_logic;
  signal c_MUL0_start : std_logic;
  signal c_HI_rst : std_logic;
  signal c_HI_enb : std_logic;
  signal c_LO_rst : std_logic;
  signal c_LO_enb : std_logic;
  signal c_DIV0_ctrl : std_logic;
  signal c_sel10_ctrl : std_logic_vector(0 downto 0);
  signal c_sel11_ctrl : std_logic_vector(0 downto 0);
  signal c_sel12_ctrl : std_logic_vector(0 downto 0);
  signal c_sel13_ctrl : std_logic_vector(0 downto 0);
  signal c_sel14_ctrl : std_logic_vector(2 downto 0);
  signal c_sel15_ctrl : std_logic_vector(0 downto 0);
  signal c_sel16_ctrl : std_logic_vector(0 downto 0);
  signal c_sel17_ctrl : std_logic_vector(0 downto 0);
  signal c_sel18_ctrl : std_logic_vector(1 downto 0);
  signal c_sel19_ctrl : std_logic_vector(1 downto 0);
  signal c_reg20_enb : std_logic;
  signal c_reg21_enb : std_logic;
  signal c_reg22_enb : std_logic;
  signal c_reg23_enb : std_logic;
  signal c_reg24_enb : std_logic;
  signal c_reg25_enb : std_logic;
  signal c_reg26_enb : std_logic;
  signal c_reg27_enb : std_logic;
  signal c_reg28_enb : std_logic;
  signal c_reg29_enb : std_logic;
  signal c_reg30_enb : std_logic;
  signal c_reg31_enb : std_logic;
  signal c_reg32_enb : std_logic;
  signal c_reg33_enb : std_logic;
  signal c_reg34_enb : std_logic;
  signal c_reg35_enb : std_logic;
  signal c_reg36_enb : std_logic;
  signal c_reg37_enb : std_logic;
  signal c_reg38_enb : std_logic;
  signal c_reg39_enb : std_logic;
  signal c_CSW_rst : std_logic;
  signal c_CSW_enb : std_logic;

begin
  ctrl : cpu_ctrl
    port map(
      instDB => instDB,
      rst => rst,
      int => int,
      intn => intn,
      clk => clk,
      IR_data_out => d_IR_data_out,
      MUL0_fin => d_MUL0_fin,
      DIV0_flag => d_DIV0_flag,
      CSW_enb => c_CSW_enb,
      CSW_rst => c_CSW_rst,
      reg39_enb => c_reg39_enb,
      reg38_enb => c_reg38_enb,
      reg37_enb => c_reg37_enb,
```

```
      reg36_enb => c_reg36_enb,
      reg35_enb => c_reg35_enb,
      reg34_enb => c_reg34_enb,
      reg33_enb => c_reg33_enb,
      reg32_enb => c_reg32_enb,
      reg31_enb => c_reg31_enb,
      reg30_enb => c_reg30_enb,
      reg29_enb => c_reg29_enb,
      reg28_enb => c_reg28_enb,
      reg27_enb => c_reg27_enb,
      reg26_enb => c_reg26_enb,
      reg25_enb => c_reg25_enb,
      reg24_enb => c_reg24_enb,
      reg23_enb => c_reg23_enb,
      reg22_enb => c_reg22_enb,
      reg21_enb => c_reg21_enb,
      reg20_enb => c_reg20_enb,
      sel19_ctrl => c_sel19_ctrl,
      sel18_ctrl => c_sel18_ctrl,
      sel17_ctrl => c_sel17_ctrl,
      sel16_ctrl => c_sel16_ctrl,
      sel15_ctrl => c_sel15_ctrl,
      sel14_ctrl => c_sel14_ctrl,
      sel13_ctrl => c_sel13_ctrl,
      sel12_ctrl => c_sel12_ctrl,
      sel11_ctrl => c_sel11_ctrl,
      sel10_ctrl => c_sel10_ctrl,
      DIV0_ctrl => c_DIV0_ctrl,
      LO_enb => c_LO_enb,
      LO_rst => c_LO_rst,
      HI_enb => c_HI_enb,
      HI_rst => c_HI_rst,
      MUL0_start => c_MUL0_start,
      MUL0_ctrl => c_MUL0_ctrl,
      SFT0_mode => c_SFT0_mode,
      DMEM_ext_ctrl => c_DMEM_ext_ctrl,
      DMEM_ac_ctrl => c_DMEM_ac_ctrl,
      DMEM_req => c_DMEM_req,
      DMEM_rw => c_DMEM_rw,
      EXT0_ctrl => c_EXT0_ctrl,
      ALU0_ctrl => c_ALU0_ctrl,
      ALU0_cin => c_ALU0_cin,
      GPR_w_enb0 => c_GPR_w_enb0,
      GPR_reset => c_GPR_reset,
      IR_enb => c_IR_enb,
      IR_rst => c_IR_rst,
      PC_hold => c_PC_hold,
      PC_reset => c_PC_reset,
      PC_load => c_PC_load,
      reg20_data_out => d_reg20_data_out,
      sys4_p0 => d_sys4_p0,
      sys2_p0 => d_sys2_p0,
      ALU0_flag => d_ALU0_flag);
  PC : pcu_17
    port map(
      clk => clk,
      load => c_PC_load,
      reset => c_PC_reset,
      hold => c_PC_hold,
      data => d_sel11_data_out,
      q => d_PC_q);
  IMEM : imcu_18
    port map(
      addr => d_PC_q,
      data => d_IMEM_data,
      m_addr => instAB,
      m_data => instDB);
  IR : register_9
    port map(
      clk => clk,
      rst => c_IR_rst,
      enb => c_IR_enb,
      data_in => d_IMEM_data,
      data_out => d_IR_data_out);
  GPR : registerfile_10
    port map(
      clock => clk,
      reset => c_GPR_reset,
      w_enb0 => c_GPR_w_enb0,
      w_sel0 => d_sel13_data_out,
      data_in0 => d_reg33_data_out,
      r_sel0 => d_IR_data_out(25 downto 21),
      r_sel1 => d_IR_data_out(20 downto 16),
      data_out0 => d_GPR_data_out0,
      data_out1 => d_GPR_data_out1);
  ALU0 : alu_12
    port map(
      a => d_reg20_data_out,
      b => d_reg34_data_out,
      cin => c_ALU0_cin,
      ctrl => c_ALU0_ctrl,
      result => d_ALU0_result,
      flag => d_ALU0_flag);
  EXT0 : extender_15
    port map(
      data_in => d_IR_data_out(15 downto 0),
      ctrl => c_EXT0_ctrl,
      data_out => d_EXT0_data_out);
  ADD0 : adder_11
    port map(
      a => d_reg23_data_out,
      b => d_sys1_p0,
      cin => d_sys2_p0,
      result => d_ADD0_result,
      cout => d_ADD0_cout);
  DMEM : dmcu_19
    port map(
      rw => c_DMEM_rw,
      req => c_DMEM_req,
      addr => d_reg24_data_out,
      i_data => d_DMEM_i_data,
      o_data => d_reg27_data_out,
      ac_ctrl => c_DMEM_ac_ctrl,
      ext_ctrl => c_DMEM_ext_ctrl,
      addr_err => d_DMEM_addr_err,
      we => we,
      m_addr => dataAB,
      m_data => dataDB);
  SFT0 : barrelshifter_14
    port map(
      data_in => d_reg26_data_out,
      mode => c_SFT0_mode,
      ctrl => d_reg35_data_out,
      data_out => d_SFT0_data_out);
  NOT0 : not_20
    port map(
      data_in => d_ALU0_flag(3),
      data_out => d_NOT0_data_out);
  MUL0 : multiplier_16
    port map(
      clk => clk,
      reset => d_sys2_p0,
      a => d_reg20_data_out,
      b => d_reg26_data_out,
      ctrl => c_MUL0_ctrl,
      start => c_MUL0_start,
      result => d_MUL0_result,
      fin => d_MUL0_fin);
```

```
  HI : register_9
    port map(
      clk => clk,
      rst => c_HI_rst,
      enb => c_HI_enb,
      data_in => d_reg37_data_out,
      data_out => d_HI_data_out);
  LO : register_9
    port map(
      clk => clk,
      rst => c_LO_rst,
      enb => c_LO_enb,
      data_in => d_reg39_data_out,
      data_out => d_LO_data_out);
  DIV0 : divider_13
    port map(
      clk => clk,
      a => d_reg20_data_out,
      b => d_reg26_data_out,
      ctrl => c_DIV0_ctrl,
      result0 => d_DIV0_result0,
      result1 => d_DIV0_result1,
      flag => d_DIV0_flag);
  sel10 : selector_21
    port map(
      data_in0 => d_GPR_data_out0,
      data_in1 => d_sys5_p0,
      ctrl => c_sel10_ctrl,
      data_out => d_sel10_data_out);
  sel11 : selector_21
    port map(
      data_in0 => d_reg28_data_out,
      data_in1 => d_ADD0_result,
      ctrl => c_sel11_ctrl,
      data_out => d_sel11_data_out);
  sel12 : selector_22
    port map(
      data_in0 => d_IR_data_out(20 downto 16),
      data_in1 => d_IR_data_out(15 downto 11),
      ctrl => c_sel12_ctrl,
      data_out => d_sel12_data_out);
  sel13 : selector_22
    port map(
      data_in0 => d_reg31_data_out,
      data_in1 => d_sys3_p0,
      ctrl => c_sel13_ctrl,
      data_out => d_sel13_data_out);
  sel14 : selector_23
    port map(
      data_in0 => d_reg25_data_out,
      data_in1 => d_LO_data_out,
      data_in2 => d_HI_data_out,
      data_in3 => d_sys10_p0,
      data_in4 => d_sys9_p0,
      data_in5 => d_SFT0_data_out,
      data_in6 => d_PC_q,
      data_in7 => d_ALU0_result,
      ctrl => c_sel14_ctrl,
      data_out => d_sel14_data_out);
  sel15 : selector_21
    port map(
      data_in0 => d_reg32_data_out,
      data_in1 => d_DMEM_i_data,
      ctrl => c_sel15_ctrl,
      data_out => d_sel15_data_out);
  sel16 : selector_21
    port map(
      data_in0 => d_EXT0_data_out,
      data_in1 => d_GPR_data_out1,
      ctrl => c_sel16_ctrl,
      data_out => d_sel16_data_out);
  sel17 : selector_22
    port map(
      data_in0 => d_GPR_data_out0(4 downto 0),
      data_in1 => d_IR_data_out(10 downto 6),
      ctrl => c_sel17_ctrl,
      data_out => d_sel17_data_out);
  sel18 : selector_24
    port map(
      data_in0 => d_reg20_data_out,
      data_in1 => d_DIV0_result1,
      data_in2 => d_MUL0_result(63 downto 32),
      ctrl => c_sel18_ctrl,
      data_out => d_sel18_data_out);
  sel19 : selector_24
    port map(
      data_in0 => d_reg20_data_out,
      data_in1 => d_DIV0_result0,
      data_in2 => d_MUL0_result(31 downto 0),
      ctrl => c_sel19_ctrl,
      data_out => d_sel19_data_out);
  reg20 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg20_enb,
      data_in => d_GPR_data_out0,
      data_out => d_reg20_data_out);
  reg21 : pipereg_26
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg21_enb,
      data_in => d_EXT0_data_out(29 downto 0),
      data_out => d_reg21_data_out);
  reg22 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg22_enb,
      data_in => d_PC_q,
      data_out => d_reg22_data_out);
  reg23 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg23_enb,
      data_in => d_reg22_data_out,
      data_out => d_reg23_data_out);
  reg24 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg24_enb,
      data_in => d_ALU0_result,
      data_out => d_reg24_data_out);
  reg25 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg25_enb,
      data_in => d_sys7_p0,
      data_out => d_reg25_data_out);
  reg26 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg26_enb,
```

```
      data_in => d_GPR_data_out1,
      data_out => d_reg26_data_out);
  reg27 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg27_enb,
      data_in => d_reg26_data_out,
      data_out => d_reg27_data_out);
  reg28 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg28_enb,
      data_in => d_sel10_data_out,
      data_out => d_reg28_data_out);
  reg29 : pipereg_27
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg29_enb,
      data_in => d_sel12_data_out,
      data_out => d_reg29_data_out);
  reg30 : pipereg_27
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg30_enb,
      data_in => d_reg29_data_out,
      data_out => d_reg30_data_out);
  reg31 : pipereg_27
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg31_enb,
      data_in => d_reg30_data_out,
      data_out => d_reg31_data_out);
  reg32 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg32_enb,
      data_in => d_sel14_data_out,
      data_out => d_reg32_data_out);
  reg33 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg33_enb,
      data_in => d_sel15_data_out,
      data_out => d_reg33_data_out);
  reg34 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg34_enb,
      data_in => d_sel16_data_out,
      data_out => d_reg34_data_out);
  reg35 : pipereg_27
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg35_enb,
      data_in => d_sel17_data_out,
      data_out => d_reg35_data_out);
  reg36 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg36_enb,
      data_in => d_sel18_data_out,
      data_out => d_reg36_data_out);
  reg37 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg37_enb,
      data_in => d_reg36_data_out,
      data_out => d_reg37_data_out);
  reg38 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg38_enb,
      data_in => d_sel19_data_out,
      data_out => d_reg38_data_out);
  reg39 : pipereg_25
    port map(
      clk => clk,
      rst => d_sys2_p0,
      enb => c_reg39_enb,
      data_in => d_reg38_data_out,
      data_out => d_reg39_data_out);
  CSW : register_9
    port map(
      clk => clk,
      rst => c_CSW_rst,
      enb => c_CSW_enb,
      data_in => d_PC_q,
      data_out => d_CSW_data_out);
  d_sys0_p0 <= "00";
  d_sys1_p0 <= (d_reg21_data_out & d_sys0_p0);
  d_sys2_p0 <= '0';
  d_sys3_p0 <= "11111";
  d_sys4_p0 <= '1';
  d_sys5_p0 <= ((d_reg22_data_out(31 downto 28) & d_IR_data_out(25 downto 0)) & d_sys0_p0);
  d_sys6_p0 <= "0000000000000000";
  d_sys7_p0 <= (d_IR_data_out(15 downto 0) & d_sys6_p0);
  d_sys8_p0 <= "0000000000000000000000000000000";
  d_sys9_p0 <= (d_sys8_p0 & d_ALU0_flag(1));
  d_sys10_p0 <= (d_sys8_p0 & d_NOT0_data_out);
end syn;
```

C.2 VHDL Descriptionf of PEAS R3K Controller

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- entity_begin

entity cpu_ctrl is
  port (
    instDB : in std_logic_vector(31 downto 0);
    rst : in std_logic;
    int : in std_logic;
    intn : in std_logic_vector(2 downto 0);
    clk : in std_logic;
    IR_data_out : in std_logic_vector(31 downto 0);
    MUL0_fin : in std_logic;
    DIV0_flag : in std_logic_vector(1 downto 0);
    CSW_enb : out std_logic;
    CSW_rst : out std_logic;
    reg39_enb : out std_logic;
    reg38_enb : out std_logic;
    reg37_enb : out std_logic;
    reg36_enb : out std_logic;
```

```
    reg35_enb : out std_logic;
    reg34_enb : out std_logic;
    reg33_enb : out std_logic;
    reg32_enb : out std_logic;
    reg31_enb : out std_logic;
    reg30_enb : out std_logic;
    reg29_enb : out std_logic;
    reg28_enb : out std_logic;
    reg27_enb : out std_logic;
    reg26_enb : out std_logic;
    reg25_enb : out std_logic;
    reg24_enb : out std_logic;
    reg23_enb : out std_logic;
    reg22_enb : out std_logic;
    reg21_enb : out std_logic;
    reg20_enb : out std_logic;
    sel19_ctrl : out std_logic_vector(1 downto 0);
    sel18_ctrl : out std_logic_vector(1 downto 0);
    sel17_ctrl : out std_logic_vector(0 downto 0);
    sel16_ctrl : out std_logic_vector(0 downto 0);
    sel15_ctrl : out std_logic_vector(0 downto 0);
    sel14_ctrl : out std_logic_vector(2 downto 0);
    sel13_ctrl : out std_logic_vector(0 downto 0);
    sel12_ctrl : out std_logic_vector(0 downto 0);
    sel11_ctrl : out std_logic_vector(0 downto 0);
    sel10_ctrl : out std_logic_vector(0 downto 0);
    DIV0_ctrl : out std_logic;
    LO_enb : out std_logic;
    LO_rst : out std_logic;
    HI_enb : out std_logic;
    HI_rst : out std_logic;
    MUL0_start : out std_logic;
    MUL0_ctrl : out std_logic;
    SFT0_mode : out std_logic_vector(1 downto 0);
    DMEM_ext_ctrl : out std_logic;
    DMEM_ac_ctrl : out std_logic_vector(1 downto 0);
    DMEM_req : out std_logic;
    DMEM_rw : out std_logic;
    EXT0_ctrl : out std_logic;
    ALU0_ctrl : out std_logic_vector(4 downto 0);
    ALU0_cin : out std_logic;
    GPR_w_enb0 : out std_logic;
    GPR_reset : out std_logic;
    IR_enb : out std_logic;
    IR_rst : out std_logic;
    PC_hold : out std_logic;
    PC_reset : out std_logic;
    PC_load : out std_logic;
    reg20_data_out : in std_logic_vector(31 downto 0);
    sys4_p0 : in std_logic;
    sys2_p0 : in std_logic;
    ALU0_flag : in std_logic_vector(3 downto 0));
end cpu_ctrl;

-- entity_end

architecture behavior of cpu_ctrl is
  type Type_Itype is(I_ADD,I_ADDI,I_ADDIU,I_ADDU,I_ANDI,I_BGEZ,I_BGEZAL,I_BGTZ,I_BLEZ,I_BLTZ,
I_BLTZAL,I_IAND,I_INOR,I_IOR,I_ISUB,I_IXOR,I_J,I_JAL,I_JALR,I_JR,I_LB,I_LBU,I_LH,I_LHU,I_LUI,
I_LW,I_ORI,I_SB,I_SH,I_SLL,I_SLLV,I_SLT,I_SLTI,I_SLTIU,I_SLTU,I_SRA,I_SRAV,I_SRL,I_SRLV,I_SUBU,
I_SW,I_XORI,I_MULT,I_MULTU,I_DIV,I_DIVU,I_MFHI,I_MFLO,I_MTHI,I_MTLO,I_BEQ,I_BNE,I_S_ERR);
  type Type_Interruption is(INT_reset, INT_init0);
  subtype Type_Intr_Count is integer range 0 to 2;
  subtype Type_interrupt_state is integer range 0 to 2;
  signal inst : Type_Itype;
  signal go : std_logic_vector(0 to 5);
  signal valid : std_logic_vector(1 to 5);
  signal rreset : std_logic;
  signal Interrupt_Step : Type_Intr_Count;
  signal iinterrupt : std_logic;
  signal interrupt_name : Type_Interruption;
  signal interrupt_state : Type_interrupt_state;
  signal next_multi_st_3 : std_logic;
  signal multi_st_3 : std_logic;
  signal lock_multi_3 : std_logic;
  signal lock_3 : std_logic;
  signal cw_2 : std_logic_vector(35 downto 0);
  signal cw_3 : std_logic_vector(30 downto 0);
  signal cw_4 : std_logic_vector(9 downto 0);
  signal cw_5 : std_logic_vector(3 downto 0);
  signal bbranch : std_logic;
  signal lock_3_ctrl_DIV0_flag : std_logic;
  signal lock_3_ctrl_MUL0_fin : std_logic;
  begin
go(0) <= '1' when (interrupt_state = 1) else '0';
  go(1) <= valid(1) and (not valid(2) or go(2));
  go(2) <= valid(2) and (not valid(3) or go(3));
  go(3) <= valid(3) and (not valid(4) or go(4)) and not lock_3;
  go(4) <= valid(4) and (not valid(5) or go(5));
  go(5) <= valid(5);
  CTRL: process(clk, rreset, bbranch)
    begin
      if(clk'event and clk = '1') then
        if(rreset = '1') then
          valid <= "00000";
        elsif(bbranch = '1') then
          valid <=  go(0) & "0" & valid(2 to 4);
        else
          valid(1) <= go(0);
          valid(2) <= (valid(1) and go(1)) or (not go(2) and valid(2));
          valid(3) <= (valid(2) and go(2)) or (not go(3) and valid(3));
          valid(4) <= (valid(3) and go(3)) or (not go(4) and valid(4));
          valid(5) <= (valid(4) and go(4)) or (not go(5) and valid(5));
        end if;
        if(rreset = '0') then
        end if;
      end if;
    end process CTRL;
  INTERRUPT: process(clk)
    begin
      if(clk'event and clk = '1') then
        if(rreset = '1') then
          interrupt_state <= 0;
          Interrupt_Step  <= 1;
          interrupt_name  <= INT_reset;
        elsif(interrupt_state = 0) then
          if(Interrupt_Step = 1) and (interrupt_name = INT_reset) then
            Interrupt_Step  <= 0;
            interrupt_state <= 1;
          elsif(Interrupt_Step = 1) and (interrupt_name = INT_init0) then
            Interrupt_Step  <= 0;
            interrupt_state <= 1;
          else
            Interrupt_Step <= Interrupt_Step + 1;
          end if;
        elsif(interrupt_state = 1) then
          if (rst='1') or(int = '1' and intn = "000") then
            interrupt_state <= 2;
          end if;
          if (rst='1') then
            interrupt_name <= INT_reset;
          elsif (int = '1' and intn = "000") then
            interrupt_name <= INT_init0;
          end if;
        else
          if(valid = "00000") then
            Interrupt_Step  <= 1;
            interrupt_state <= 0;
```

```
          end if;
        end if;
      end if;
    end process INTERRUPT;
    lock_multi_3 <= '1' when((lock_3_ctrl_DIV0_flag = '1') and (DIV0_flag(0) /= '1'))
 or ((lock_3_ctrl_MUL0_fin = '1') and (MUL0_fin /= '1'))
 else
'0';
  next_multi_st_3 <= '0' when go(2) = '1' else
'1';
  MULT_ST: process(clk)
    begin
      if(clk'event and clk = '1') then
        if(rreset = '1') then
          multi_st_3 <= '0';
        else
          multi_st_3 <= next_multi_st_3;
        end if;
      end if;
    end process MULT_ST;
    lock_3 <= lock_multi_3;
inst <=
I_ADD when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100000") else
I_ADDI when (IR_data_out(31 downto 26) = "001000") else
I_ADDIU when (IR_data_out(31 downto 26) = "001001") else
I_ADDU when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100001") else
I_ANDI when (IR_data_out(31 downto 26) = "001100") else
I_BGEZ when (IR_data_out(31 downto 26) = "000001") and (IR_data_out(20 downto 16) = "00001") else
I_BGEZAL when (IR_data_out(31 downto 26) = "000001") and (IR_data_out(20 downto 16) = "10001") else
I_BGTZ when (IR_data_out(31 downto 26) = "000111") and (IR_data_out(20 downto 16) = "00000") else
I_BLEZ when (IR_data_out(31 downto 26) = "000110") and (IR_data_out(20 downto 16) = "00000") else
I_BLTZ when (IR_data_out(31 downto 26) = "000001") and (IR_data_out(20 downto 16) = "00000") else
I_BLTZAL when (IR_data_out(31 downto 26) = "000001") and (IR_data_out(20 downto 16) = "10000") else
I_IAND when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100100") else
I_INOR when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100111") else
I_IOR when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100101") else
I_ISUB when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100010") else
I_IXOR when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100110") else
I_J when (IR_data_out(31 downto 26) = "000010") else
I_JAL when (IR_data_out(31 downto 26) = "000011") else
I_JALR when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "001001") else
I_JR when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "001000") else
I_LB when (IR_data_out(31 downto 26) = "100000") else
I_LBU when (IR_data_out(31 downto 26) = "100100") else
I_LH when (IR_data_out(31 downto 26) = "100001") else
I_LHU when (IR_data_out(31 downto 26) = "100101") else
I_LUI when (IR_data_out(31 downto 26) = "001111") else
I_LW when (IR_data_out(31 downto 26) = "100011") else
I_ORI when (IR_data_out(31 downto 26) = "001101") else
I_SB when (IR_data_out(31 downto 26) = "101000") else
I_SH when (IR_data_out(31 downto 26) = "101001") else
I_SLL when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000000") else
I_SLLV when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000100") else
I_SLT when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "101010") else
I_SLTI when (IR_data_out(31 downto 26) = "001010") else
I_SLTIU when (IR_data_out(31 downto 26) = "001011") else
I_SLTU when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "101011") else
I_SRA when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000011") else
I_SRAV when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000111") else
I_SRL when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000010") else
I_SRLV when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "000110") else
I_SUBU when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "100011") else
I_SW when (IR_data_out(31 downto 26) = "101011") else
I_XORI when (IR_data_out(31 downto 26) = "001110") else
I_MULT when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "011000") else
I_MULTU when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "011001") else
I_DIV when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "011010") else
I_DIVU when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "011011") else
I_MFHI when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "010000") else
```

```
I_MFLO when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "010010") else
I_MTHI when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "010001") else
I_MTLO when (IR_data_out(31 downto 26) = "000000") and (IR_data_out(5 downto 0) = "010011") else
I_BEQ when (IR_data_out(31 downto 26) = "000100") else
I_BNE when (IR_data_out(31 downto 26) = "000101") else
I_S_ERR;
cw_2(35) <= '1' when (inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_BGEZ) or (inst = I_BGEZAL) or
(inst = I_BGTZ) or (inst = I_BLEZ) or (inst = I_BLTZ) or (inst = I_BLTZAL) or
(inst = I_LB) or (inst = I_LBU) or (inst = I_LH) or (inst = I_LHU) or
(inst = I_LW) or (inst = I_SB) or (inst = I_SH) or (inst = I_SLTI) or
(inst = I_SLTIU) or (inst = I_SW) or (inst = I_BEQ) or (inst = I_BNE)
 else
'0';
cw_2(34) <= '1' when (inst = I_J) or (inst = I_JAL) else
'0';
cw_2(33) <= '1' when (inst = I_ADD) or (inst = I_ADDU) or (inst = I_IAND) or (inst = I_INOR) or
(inst = I_IOR) or (inst = I_ISUB) or (inst = I_IXOR) or (inst = I_SLL) or
(inst = I_SLLV) or (inst = I_SLT) or (inst = I_SLTU) or (inst = I_SRA) or
(inst = I_SRAV) or (inst = I_SRL) or (inst = I_SRLV) or (inst = I_SUBU) or
(inst = I_MFHI) or (inst = I_MFLO) else
'0';
cw_2(32) <= '1' when (inst = I_ADD) or (inst = I_ADDU) or (inst = I_IAND) or (inst = I_INOR) or
(inst = I_IOR) or (inst = I_ISUB) or (inst = I_IXOR) or (inst = I_SLT) or
(inst = I_SLTU) or (inst = I_SUBU) or (inst = I_BEQ) or (inst = I_BNE)
 else
'0';
cw_2(31) <= '1' when (inst = I_SLL) or (inst = I_SRA) or (inst = I_SRL) else
'0';
cw_2(30) <= '1' when (inst = I_J) or (inst = I_JAL) or (inst = I_JALR) or (inst = I_JR)
 else
'0';
cw_2(29) <= '1' when (inst = I_ISUB) or (inst = I_SLT) or (inst = I_SLTI) or (inst = I_SLTIU) or
(inst = I_SLTU) or (inst = I_SUBU) or (inst = I_BEQ) or (inst = I_BNE)
 else
'0';
cw_2(28) <= '1' when (inst = I_IXOR) or (inst = I_XORI) or (inst = I_ADD) or (inst = I_ADDI) or
(inst = I_ADDIU) or (inst = I_ADDU) or (inst = I_LB) or (inst = I_LBU) or
(inst = I_LH) or (inst = I_LHU) or (inst = I_LW) or (inst = I_SB) or
(inst = I_SH) or (inst = I_SW) else
'0';
cw_2(27) <= '1' when (inst = I_SLTU) or (inst = I_ISUB) or (inst = I_SLT) or (inst = I_SLTI) or
(inst = I_SLTIU) or (inst = I_SUBU) or (inst = I_BEQ) or (inst = I_BNE) or
(inst = I_ANDI) or (inst = I_IAND) else
'0';
cw_2(26) <= '1' when (inst = I_ISUB) or (inst = I_SLT) or (inst = I_SLTI) or (inst = I_SLTIU) or
(inst = I_SUBU) or (inst = I_BEQ) or (inst = I_BNE) or (inst = I_INOR) or
(inst = I_BGEZ) or (inst = I_BGTZ) or (inst = I_BLEZ) or (inst = I_ADD) or
(inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_ADDU) or (inst = I_LB) or
(inst = I_LBU) or (inst = I_LH) or (inst = I_LHU) or (inst = I_LW) or
(inst = I_SB) or (inst = I_SH) or (inst = I_SW) else
'0';
cw_2(25) <= '1' when (inst = I_IXOR) or (inst = I_XORI) or (inst = I_IOR) or (inst = I_ORI) or
(inst = I_INOR) or (inst = I_ANDI) or (inst = I_IAND) else
'0';
cw_2(24) <= '1' when (inst = I_SRA) or (inst = I_SRAV) else
'0';
cw_2(23) <= '1' when (inst = I_SRL) or (inst = I_SRLV) or (inst = I_SRA) or (inst = I_SRAV)
 else
'0';
cw_2(22) <= '1' when (inst = I_MULT) else
'0';
cw_2(21) <= '1' when (inst = I_DIVU) else
'0';
cw_2(20) <= '1' when (inst = I_BGEZ) or (inst = I_BGEZAL) else
'0';
cw_2(19) <= '1' when (inst = I_BGTZ) else
'0';
cw_2(18) <= '1' when (inst = I_BLEZ) else
'0';
```

```
cw_2(17) <= '1' when (inst = I_BLTZ) or (inst = I_BLTZAL) else
'0';
cw_2(16) <= '1' when (inst = I_BEQ) else
'0';
cw_2(15) <= '1' when (inst = I_BNE) else
'0';
cw_2(14) <= '1' when (inst = I_ADD) or (inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_ADDU) or
(inst = I_ANDI) or (inst = I_IAND) or (inst = I_INOR) or (inst = I_IOR) or
(inst = I_ISUB) or (inst = I_IXOR) or (inst = I_ORI) or (inst = I_SUBU) or
(inst = I_XORI) or (inst = I_SLL) or (inst = I_SLLV) or (inst = I_SRA) or
(inst = I_SRAV) or (inst = I_SRL) or (inst = I_SRLV) or (inst = I_SLTIU) or
(inst = I_SLTU) or (inst = I_MFLO) else
'0';
cw_2(13) <= '1' when (inst = I_ADD) or (inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_ADDU) or
(inst = I_ANDI) or (inst = I_IAND) or (inst = I_INOR) or (inst = I_IOR) or
(inst = I_ISUB) or (inst = I_IXOR) or (inst = I_ORI) or (inst = I_SUBU) or
(inst = I_XORI) or (inst = I_BGEZAL) or (inst = I_BLTZAL) or (inst = I_JAL) or
(inst = I_JALR) or (inst = I_SLTIU) or (inst = I_SLTU) or (inst = I_MFHI)
 else
'0';
cw_2(12) <= '1' when (inst = I_ADD) or (inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_ADDU) or
(inst = I_ANDI) or (inst = I_IAND) or (inst = I_INOR) or (inst = I_IOR) or
(inst = I_ISUB) or (inst = I_IXOR) or (inst = I_ORI) or (inst = I_SUBU) or
(inst = I_XORI) or (inst = I_BGEZAL) or (inst = I_BLTZAL) or (inst = I_JAL) or
(inst = I_JALR) or (inst = I_SLL) or (inst = I_SLLV) or (inst = I_SRA) or
(inst = I_SRAV) or (inst = I_SRL) or (inst = I_SRLV) or (inst = I_SLT) or
(inst = I_SLTI) else
'0';
cw_2(11) <= '1' when (inst = I_DIV) or (inst = I_DIVU) else
'0';
cw_2(10) <= '1' when (inst = I_MULT) or (inst = I_MULTU) else
'0';
cw_2(9) <= '1' when (inst = I_SB) or (inst = I_SH) or (inst = I_SW) else
'0';
cw_2(8) <= '1' when (inst = I_LB) or (inst = I_LBU) or (inst = I_LH) or (inst = I_LHU) or
(inst = I_LW) or (inst = I_SB) or (inst = I_SH) or (inst = I_SW)
 else
'0';
cw_2(7) <= '1' when (inst = I_LW) or (inst = I_SW) or (inst = I_LH) or (inst = I_LHU) or
(inst = I_SH) else
'0';
cw_2(6) <= '1' when (inst = I_LW) or (inst = I_SW) else
'0';
cw_2(5) <= '1' when (inst = I_LB) or (inst = I_LH) or (inst = I_SB) or (inst = I_SH)
 else
'0';
cw_2(4) <= '1' when (inst = I_LB) or (inst = I_LBU) or (inst = I_LH) or (inst = I_LHU) or
(inst = I_LW) else
'0';
cw_2(3) <= '1' when (inst = I_ADD) or (inst = I_ADDI) or (inst = I_ADDIU) or (inst = I_ADDU) or
(inst = I_ANDI) or (inst = I_BGEZAL) or (inst = I_BLTZAL) or (inst = I_IAND) or
(inst = I_INOR) or (inst = I_IOR) or (inst = I_ISUB) or (inst = I_IXOR) or
(inst = I_JAL) or (inst = I_JALR) or (inst = I_LB) or (inst = I_LBU) or
(inst = I_LH) or (inst = I_LHU) or (inst = I_LUI) or (inst = I_LW) or
(inst = I_ORI) or (inst = I_SLL) or (inst = I_SLLV) or (inst = I_SLT) or
(inst = I_SLTI) or (inst = I_SLTIU) or (inst = I_SLTU) or (inst = I_SRA) or
(inst = I_SRAV) or (inst = I_SRL) or (inst = I_SRLV) or (inst = I_SUBU) or
(inst = I_XORI) or (inst = I_MFHI) or (inst = I_MFLO) else
'0';
cw_2(2) <= '1' when (inst = I_MULT) or (inst = I_MULTU) or (inst = I_DIV) or (inst = I_DIVU) or
(inst = I_MTHI) else
'0';
cw_2(1) <= '1' when (inst = I_MULT) or (inst = I_MULTU) or (inst = I_DIV) or (inst = I_DIVU) or
(inst = I_MTLO) else
'0';
cw_2(0) <= '1' when (inst = I_BGEZAL) or (inst = I_BLTZAL) or (inst = I_JAL) or (inst = I_JALR)
 else
'0';
  lock_3_ctrl_MULO_fin <= '0' when valid(3) = '0' else
```

```
cw_3(10);
  lock_3_ctrl_DIV0_flag <= ’0’ when valid(3) = ’0’ else
cw_3(11);
  bbranch <= ’0’ when go(3) = ’0’ else
’1’ when (ALU0_flag(2) = sys2_p0) and cw_3(15) = ’1’ else
’1’ when (ALU0_flag(2) = sys4_p0) and cw_3(16) = ’1’ else
’1’ when (reg20_data_out(31) = sys4_p0) and cw_3(17) = ’1’ else
’1’ when (reg20_data_out(31) = sys4_p0 or ALU0_flag(2) = sys4_p0) and cw_3(18) = ’1’ else
’1’ when (reg20_data_out(31) = sys2_p0 and ALU0_flag(2) = sys2_p0) and cw_3(19) = ’1’ else
’1’ when (reg20_data_out(31) = sys2_p0) and cw_3(20) = ’1’ else
cw_3(30);
  PC_load <= ’0’ when go(3) = ’0’ else
’1’ when (ALU0_flag(2) = sys2_p0) and cw_3(15) = ’1’ else
’1’ when (ALU0_flag(2) = sys4_p0) and cw_3(16) = ’1’ else
’1’ when (reg20_data_out(31) = sys4_p0) and cw_3(17) = ’1’ else
’1’ when (reg20_data_out(31) = sys4_p0 or ALU0_flag(2) = sys4_p0) and cw_3(18) = ’1’ else
’1’ when (reg20_data_out(31) = sys2_p0 and ALU0_flag(2) = sys2_p0) and cw_3(19) = ’1’ else
’1’ when (reg20_data_out(31) = sys2_p0) and cw_3(20) = ’1’ else
cw_3(30);
  PC_reset <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  PC_hold <= ’1’ when go(1) = ’0’ else
’0’;
  IR_rst <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  IR_enb <= ’0’ when go(1) = ’0’ else
’1’;
  GPR_reset <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  GPR_w_enb0 <= ’0’ when go(5) = ’0’ else
cw_5(3);
  ALU0_cin <= cw_3(29);
  ALU0_ctrl <= cw_3(25) & cw_3(26) & ’0’ & cw_3(27) & cw_3(28);
  EXT0_ctrl <= cw_2(35);
  DMEM_rw <= ’0’ when go(4) = ’0’ else
cw_4(9);
  DMEM_req <= ’0’ when go(4) = ’0’ else
cw_4(8);
  DMEM_ac_ctrl <= cw_4(6) & cw_4(7);
  DMEM_ext_ctrl <= cw_4(5);
  SFT0_mode <= cw_3(23) & cw_3(24);
  MUL0_ctrl <= cw_3(22);
  MUL0_start <= ’0’ when multi_st_3 = ’1’ else
cw_3(10);
  HI_rst <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  HI_enb <= ’0’ when go(5) = ’0’ else
cw_5(2);
  LO_rst <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  LO_enb <= ’0’ when go(5) = ’0’ else
cw_5(1);
  DIV0_ctrl <= cw_3(21);
  sel10_ctrl <= cw_2(34 downto 34);
  sel11_ctrl <= "1" when (reg20_data_out(31) = sys2_p0) and cw_3(20) = ’1’ else
"1" when (reg20_data_out(31) = sys2_p0 and ALU0_flag(2) = sys2_p0) and cw_3(19) = ’1’ else
"1" when (reg20_data_out(31) = sys4_p0 or ALU0_flag(2) = sys4_p0) and cw_3(18) = ’1’ else
"1" when (reg20_data_out(31) = sys4_p0) and cw_3(17) = ’1’ else
"1" when (ALU0_flag(2) = sys4_p0) and cw_3(16) = ’1’ else
"1" when (ALU0_flag(2) = sys2_p0) and cw_3(15) = ’1’ else
"0";
  sel12_ctrl <= cw_2(33 downto 33);
  sel13_ctrl <= cw_5(0 downto 0);
  sel14_ctrl <= cw_3(12) & cw_3(13) & cw_3(14);
  sel15_ctrl <= cw_4(4 downto 4);
  sel16_ctrl <= cw_2(32 downto 32);
  sel17_ctrl <= cw_2(31 downto 31);
  sel18_ctrl <= cw_3(10) & cw_3(11);
  sel19_ctrl <= cw_3(10) & cw_3(11);
```

```
  reg20_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg21_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg22_enb <= ’0’ when go(1) = ’0’ else
’1’;
  reg23_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg24_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg25_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg26_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg27_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg28_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg29_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg30_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg31_enb <= ’0’ when go(4) = ’0’ else
’1’;
  reg32_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg33_enb <= ’0’ when go(4) = ’0’ else
’1’;
  reg34_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg35_enb <= ’0’ when go(2) = ’0’ else
’1’;
  reg36_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg37_enb <= ’0’ when go(4) = ’0’ else
’1’;
  reg38_enb <= ’0’ when go(3) = ’0’ else
’1’;
  reg39_enb <= ’0’ when go(4) = ’0’ else
’1’;
  CSW_rst <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_reset)) else
’0’;
  CSW_enb <= ’1’ when (Interrupt_Step = 1) and ((Interrupt_name = INT_init0)) else
’0’;
  rreset <= ’1’ when (rst=’1’) else ’0’;
  PIPE_REG_CTRL: process(clk)
    begin
      if(clk’event and clk = ’1’) then
        if(go(2) = ’1’) then
          cw_3 <= cw_2(30 downto 0);
        end if;
        if(go(3) = ’1’) then
          cw_4 <= cw_3(9 downto 0);
        end if;
        if(go(4) = ’1’) then
          cw_5 <= cw_4(3 downto 0);
        end if;
      end if;
    end process PIPE_REG_CTRL;
  end behavior;
```

List of Major Publications of the Author

Journal Papers

[1] Makiko Itoh, Akichika Shiomi, Jun Sato, Yoshinori Takeuchi and Masaharu Imai: "Processor Generation Method for Pipelined Processors in Consideration with Pipeline Hazards," Transactions on Information Processing Society of Japan, Vol. 41, No. 4, pp. 851-862, Apr. 2000, (in japanese).

[2] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai and Akichika Shiomi: "Synthesizable HDL Generation for Pipelined Processors from a Micro-Operation Description," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E83-A, No. 3, pp. 394-400, Mar. 2000.

International Conference Papers

[3] Makiko Itoh, Shigeaki Higaki, Jun Sato and Akichika Shiomi, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai: "PEAS-III: An ASIP Design Environment," IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 430-436, Sept. 2000.

[4] Akira Kitajima, Makiko Itoh, Jun Sato, Akichika Shiomi, Yoshinori Takeuchi and Masaharu Imai: "Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined Processors," Asia South Pacific Design Automation Conference 2001 (ASP-DAC 2001), Jan. 2001 (to appear)

National Conference Papers

[5] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai and Akichika Shiomi: "A Synthesizable HDL Generation Method for Pipelined Processor using micro-operations," Proc. of 12th Karuizawa Workshop on Circuits and Systems, pp. 121-126, Apr. 1999, (in japanese).

[6] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai, Akichika Shiomi and Yoshihiro Aoyama: "Processor Architecture Description Generation from a Behavioral Semantics Description of Instructions," Proc. of 11th Karuizawa Workshop on Circuits and Systems, pp. 475-480, Apr. 1998, (in japanese).

[7] Makiko Itoh, Shigeaki Higaki, Akichika Shiomi, Jun Sato, Yoshinori Takeuchi and Masaharu Imai: "A Synthesizable HDL Generation Method for Pipelined Processors in Consideration of Pipeline Hazard," Proc. of Design Automation Symposium, pp. 201-206, July 1999, (in japanese).

[8] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai and Akichika Shiomi: "Instruction Set Processor Synthesis Method based on Behavioral Semantics Description", IEICE Technical Report, VLD 97-89, pp. 77-84, Oct. 1997, (in japanese).

[9] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai, Akichika Shiomi and Yoshihiro Aoyama: "Processor Design Methodology based on a Behavioral Semantics Description of Instructions," Proc. IEICE Fall Conf. '97, A-3-7, Sept. 1997, (in japanese).

[10] Shigeaki Higaki, Makiko Itoh, Jun Sato, Yoshinori Takeuchi and Masaharu Imai: "Proposal of an HDL Generation Method for Pipeline Processors with Out-of-order Completion," IPSJ SIG Notes, 99-SLDM-93 Vol.99, No.101, pp. 71-78, Nov. 1999, (in japanese).