Computer Science

PowerPC MMU Simulation

Torbjörn Andersson, Per Magnusson

Bachelor's Project

2001:18

PowerPC MMU Simulation

Torbjörn Andersson, Per Magnusson

This report is submitted in partial fulfilment of the requirements for the Bachelor’s degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Torbjörn Andersson

Per Magnusson

Approved, June 6th 2001

Advisor: Choong-ho Yi, Karlstad University

Examiner: Stefan Lindskog, Karlstad University

Abstract

An instruction-set simulator is a program that simulates a target computer by interpreting the effect of instructions on the computer, one instruction at a time. This study is based on an existing instruction-set simulator, which simulates a general PowerPC (PPC) processor with support for all general instructions in the PPC family. The subject for this thesis work is to enhance the existing general simulator with support for the memory management unit (MMU), with its related instruction-set, and the instruction-set controlling the cache functionality. This implies an extension of the collection of attributes implemented on the simulator with, for example, MMU tables and more specific registers for the target processor.

Introducing MMU functionality into a very fast simulator kernel with the aim not to affect the performance too much has been shown to be a non-trivial task. The MMU introduced complexity in the execution that demanded fast and simple solutions. One of the techniques that was used to increase the performance was to cache results from previous address translations and in that way avoid unnecessary recalculations.

The extension of the simulator, with complex functionality as MMU and interrupts, only decreased the performance approximately two times, when executing with all the MMU functionality turned on. This was possible as a result of the successfully implemented optimisations.

Acknowledgements

First we would like to thank Department of Simulator Technology at Ericsson Infotech, for their confidence in our knowledge and ability to solve this non-trivial task. Especially we would like to thank our advisor at Ericsson, Peter Dyfvelsten, for supporting us during the development of the simulator.

Ericsson Infotech specified the subject for this thesis and gave us the resources needed to solve the assignment. Therefore it’s their needs that this thesis focuses on, which affects the entire thesis.

During the documentation work our advisor Choong-ho Yi, at Karlstad University, helped us create a well-formed document. He made us focus on the important parts and also create a main thread going through the entire document. His insights reminded us that others might see things differently than we do, when we didn’t consider it.

Contents

List of Figures

List of tables

1 Introduction

This first chapter gives the background and aim of this study and it will also give a brief introduction to the simulation area, focusing on the need of simulated systems.

1.1 Background

Simulations are used to learn something about reality without having to confront it directly. They can be applied on various kinds of problems, such as economical, ecological or technical systems. When it’s too dangerous, time-consuming or costly to do experiments on the reality itself, simulations are highly motivated. In simulations it is often important that the model that is used, models the reality well. If the simulator is too rough and simplified, the result from the simulation cannot be used as a basis for some decisions or conclusions. One has to set the demands at a preferred level for each subject [1].

A simulator can be a program that emulates a hardware item or a complex unit. The simulated object has its own special properties, which needs to be represented in the simulating program. Running the simulator program results in a virtual unit, which can be manipulated and studied as a real object. The object of simulation can be a car, refrigerator etc. In this study we simulate a computer.

Ericsson Infotech (EIN) is developing simulators for embedded computer systems (An embedded computer system is a part of a larger system and performs some of the requirements of that system, for example, a computer system used in an aircraft or rapid transit system). These simulators make it possible to test software without access to target hardware. A simulator is an important and vital tool for the development of software for the AXE environment, which is a telecommunication platform developed by Ericsson. Today a number of distributed hardware platforms in the AXE environment use Power PC (PPC) processors (A detailed presentation of PPC is given in Section 2.1). Therefore Ericsson has great interest in simulating these processors [2].

Today a prototype simulation for a PPC processor exists as a result of an earlier thesis work performed by Patrik Seffel [2]. The existing simulation executes a large subset of the PPC instruction set but lacks support for memory management (MM). This means that the memory today only can be represented as a directly addressed array of bytes. With a Memory Management Unit (MMU), the simple array of bytes can be visualised as a complex structure with storage protection. It also gives the user the possibility to implement virtual memory etc.

1.2 Aim of study

The aim of this study is to add support for machine code instructions, which handle MMU and cache functionality, to the existing PPC simulator. The instructions will change the simulator’s state, just as the real processor would. The result will be an extended PPC simulator with support for MM instructions (see the requirements in Chapter 3 for details). The target processor is IBM PPC-405GP but EIN also uses some other PPC processors such as IBM PPC-750 and Motorola PPC-860 as regional processors (RPs) in the AXE environment. Therefore it’s preferable to have a generic simulator for all of them. We will therefore compare the three to see if such simulator can be created.

The goal with the extended simulator is to take a step towards a complete PPC simulator that enables software developers to run and test their applications with all possible instructions. However some instructions may only be implemented as “dummies” with no function. This is due to lack of time available for this thesis work, in combination with that some instructions have unnecessary functions for the simulator and/or that they cannot be implemented in a simulator. Such instructions that not can be implemented are cache related instructions that control the cache functionality, because the cache functionality does not exist in an instruction-set simulator.

The important thing is that the simulator can run any given program, with MMU instructions, with a proper result, i.e. proper changes in all registers and memories as defined by the processor specification.

1.3 Limitations

In this study cache-simulation, which is closely connected to the MMU-simulation, needs only to support the instructions for cache management. No actual caching needs to be performed, because there is no performance gain in implementing a cache in a simulator. Therefore the cache instructions are implemented as “dummies” with no effect on the simulated computer.

The real addressing mode will be implemented but it will lack storage attribute control features.

The simulator only needs to be able to execute preloaded programs and therefore it doesn’t need to be able to load new programs during the execution of the simulator. If one wants to load programs during runtime, the simulator needs to be able to decode instructions during runtime, which is often referred to as JIT (Just In Time) decoding. JIT decoding is avoided due to the limited time assigned for this thesis.

1.4 Disposure

The goal has been to make a compact report that should be easy to follow and understand. The complex and abstract nature of the topic of this thesis has however demanded a great effort to make this possible. To keep the report concise we have tried to explain the basic parts only. Some of the parts however include references to the appendices at the end of this thesis, where more detailed information can be found. As is the case with most technical reports, the thesis contains a number of acronyms for technical terms that are frequently referred to in the thesis. Acronyms are introduced at the first occurrence of each term and they are also listed in Appendix A, where one also could find short explanations of basic concepts that are essential to this thesis.

2 Overview of the basic components

This chapter introduces the PPC processor and explains memory management, including the MMU. The simulator core, SIMGEN, is also explained in detail.

2.1 Summary of PPC structure and model

The PPC processor family, developed jointly by Motorola, IBM and Apple Computer, span over a wide area of microprocessors. The flexibility of the PPC architecture offers many price/performance options. Designers can choose whether to implement architecturally defined features in hardware or in software. To achieve this flexibility, the PPC architecture offers a layered structure. This section is based on a book describing the programming environments for the PPC Microprocessor Family [3]. It describes the register organisation defined by three levels of the PPC architecture. The three register levels are; the most general User Instruction Set Architecture (UISA), the Virtual Environment Architecture (VEA) and the more specific Operating Environment Architecture (OEA).

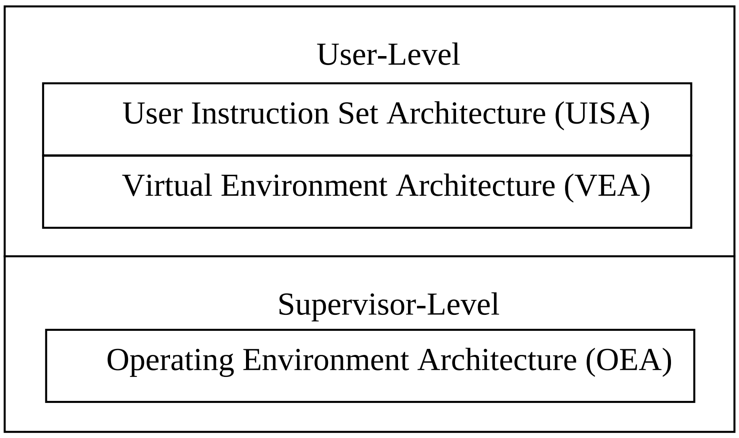

User-Level
User Instruction Set Architecture (UISA)
Virtual Environment Architecture (VEA)
Supervisor-Level
Operating Environment Architecture (OEA)

Figure 2.1 PPC architecture

PPC User Instruction Set Architecture (UISA): UISA defines the level of the architecture to which user-level (referred to as problem state in the architecture specification) software should conform. UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models. The UISA was mainly implemented by Seffel in his earlier thesis work but does also have some extensions in this study.

PPC Virtual Environment Architecture (VEA): VEA defines additional user-level functionality that falls outside typical user-level software requirements. VEA describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, cache control instructions and the time base facility from a user-level perspective. The VEA level are not applicable for this study since it is considered out of this thesis scope.

PPC Operating Environment Architecture (OEA): OEA defines supervisor-level (referred to as privileged mode in the architecture specification) resources typically required by an operating system. OEA defines the PPC memory management model, supervisor-level registers, synchronisation requirements, and the exception model. OEA is the target level for this thesis because the memory management model, which includes the MMU, is defined here.

The PPC executes programs in two modes, which are closely connected to the three levels of the PPC architecture. Programs running in privileged mode can access any register and execute any instruction. In user mode, certain registers and instructions are unavailable to programs. Privileged mode provides operating system software access to all processor resources. Because access to certain processor resources is denied in user mode, application software runs in user mode. Operating system software and other application software is then protected from the effects of errant application programs [5].

To be able to make the simulator work properly with MMU, registers and instructions in the supervisor model of the PPC, OEA, have to be implemented.

2.2 Memory management

2.2.1 Overview

Memory management is a general term that covers all the various techniques by which a logical address is translated into a “real” address. The reason why memory management is such an important feature is discussed in the section below.

The central processing unit (CPU) can be shared by a set of processes (Small parts of programs running in a computer, a program can consist of several processes). To make this possible several processes must share memory. There are many ways to manage memory and each approach has its own advantages and disadvantages. The choice of memory-management scheme depends on several factors, especially the hardware design.

A memory address generated by the CPU, which is the address that the programmer sees, is commonly referred to as a logical or virtual address, whereas an address used to point at a specific point in the memory is commonly referred to as a physical or real address. Another address term that is commonly used is effective address (EA). EA is a general expression for the address of an operand and is used because the address of an operand can be calculated in several ways. Addition of a register and an intermediate value is a common method. A program references memory using the EA computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The EA is translated to a physical address according to the procedures described later in Subsection 2.2.3. The memory subsystem uses the physical address to identify a specific position in a specific memory bank [3].

The concept of a logical address space (a set of logical addresses generated by a program) that is bound to a separate physical address space (a set of physical addresses corresponding to the logical addresses) is central to proper memory management. Computers often tend to have too little memory available, which implies that you have to worry about loading programs that are too large for the available memory. In a system where the machine is asked to perform only one task at a time, the program performing the current task is placed in memory, executed, and then replaced by the program performing the next task. However, in multitasking or multi-user environments, in which the machine may be asked to deal with many tasks at the same time, memory management plays an important role.

2.2.2 An Example

To illustrate the need of memory management the following situation, visualized in Figure 2.2, can be used. Initially, at time *t1*, three processes, A, B and C, are loaded into the memory. When process B has finished its execution, at time *t2*, it is deleted from the memory. The deletion of process B then leaves a hole in the memory.

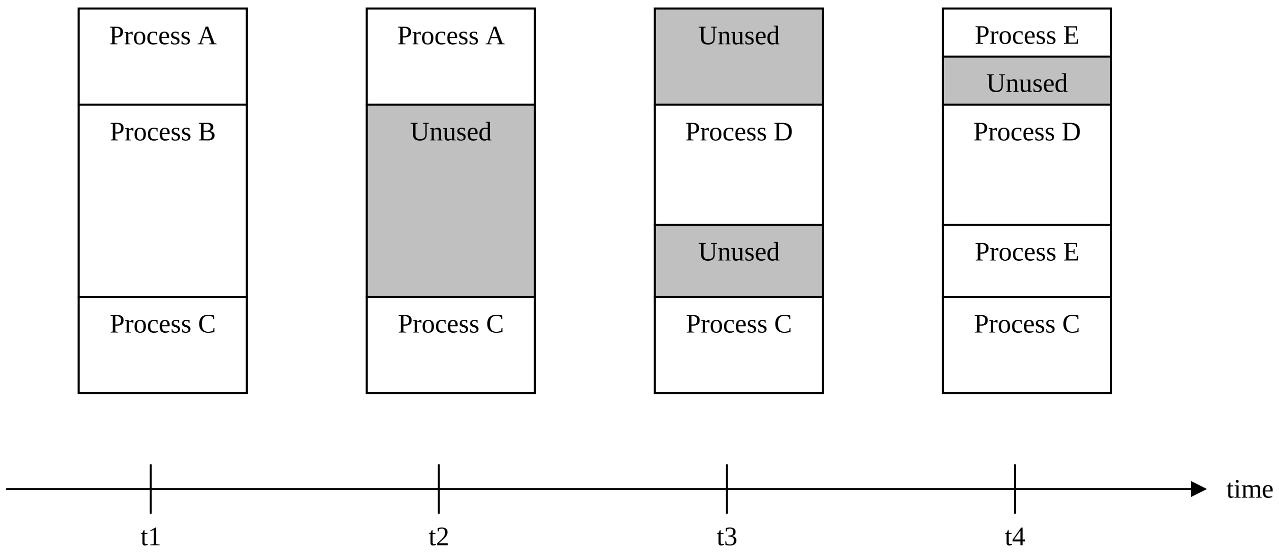


Figure 2.2 Memory allocation in a multiprocessing system.

At time *t3* a new process, D, is loaded into the unused part of the memory and short time later process A has finished its execution and is deleted from the memory.

At time *t4* another new process, E, is loaded into the memory in two parts because it can't fit in any single free block of memory space. In a multitasking system this rapidly runs into memory allocation and memory fragmentation problems.

If all computers had an infinite amount of random access memory (RAM), memory management would not be a problem. Then one could just place a program, loaded from disk, immediately after the last program you loaded into the memory. Of course this is not the reality.

There are different solutions to the fragmentation problem, but there is only one that is covered by the scope of this thesis and that is *paging* (A technique permitting the logical address-space of a process to be non-contiguous. The memory is broken into blocks of the same size called pages). Paging solves the fragmentation problem by permitting the logical address space of a process to be contiguous, and at the same time allow the physical allocated memory to be non-contiguous/fragmented. The fragmented physical reality can be masked with a translation, which maps a logical address to a "real" physical address. This allows the operating system to load a small portion of the program into primary memory and to load the rest on demand.

2.2.3 The memory management unit (MMU)

The run-time mapping from virtual to physical addresses is done by a special purpose hardware called the memory-management unit (MMU). The MMU works between the CPU and the memory as shown in Figure 2.3 [8]. Whenever the CPU generates the address of an operand or an instruction, it places the logical address on its address bus. The MMU then translates the logical address. The result from the translation points out the physical page where the physical address is located. The physical page, together with the part of the logical address that forms an offset, is used to access the location of the operand or instruction in memory. To be able to perform these translations the MMU contains a special, small, fast-lookup cache, called Translation Lookaside Buffer (TLB), which contains data required for translating the addresses and storage attributes used for memory protection control.

The MMU divides storage into pages and the MMU of the PPC-405GP processor supports multiple page sizes, a variety of storage protection attributes and access control options. The supporting of multiple page sizes leads to improved memory efficiency and minimises the number of TLB misses. A page represents the granularity of address translation and protection controls. There are eight page sizes simultaneously supported by the PPC-405GP (1Kb, 4Kb, 16Kb, 64Kb, 256Kb, 1Mb, 4Mb, 16Mb). This gives programmers great flexibility.

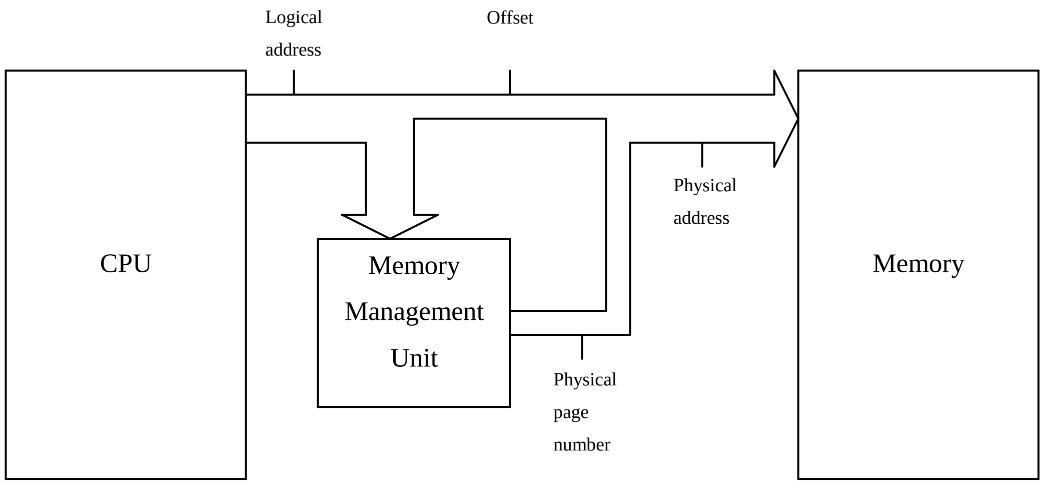


Figure 2.3 Basic address translation

The TLB is used with pages in the following way. The logical address generated by the CPU consists of two parts, a page address and an offset. The logical address is presented to the MMU and a valid entry in the TLB must exist if a translation is to be performed. The first time a logical address refers to a new page, with no valid translation in the TLB, a TLB miss exception occurs. Typically, the exception handler loads a TLB entry for the page through a "tablewalk" procedure. During a tablewalk, the software traverses a tree of data structures that define how the memory is laid out in the system. At the end of a successful tablewalk, the virtual address can be translated, and a TLB entry is loaded. If the logical address doesn't match with any of the TLB entries when a translation is performed there will be a TLB-miss interrupt. Typically the number of TLB entries varies between 8 and 2048. The IBM PPC-405GP has 64 entries in its TLB.

2.3 The instruction set simulator

An instruction set simulator is a program that simulates a target processor by interpreting the effect of instructions on a computer, one instruction at a time [2]. It abstracts away aspects such as pipeline stalls, writeback buffers, etc, but at the same time it models an abstract machine that executes the same instruction-set as a real target machine. This virtual processor can be run on a workstation far from the real processor. This implies that the very same program, intended for the real processor, can be run on the virtual processor without any restrictions.

2.3.1 SIMGEN - The core of the simulator

The simplest instruction-set simulators execute programs by running a central fetch-decode-execute loop [6]. This work scheme is inefficient and consequently not allowing the user to run real full-scale programs on the simulator. A more efficient scheme is to separate the decode phase from the execution. The decode phase will be discussed in detail later, in Subsection 2.3.2, but for now one only need to know that it's time consuming.

The separation, between the decode phase and the execution, has been implemented in several simulators and in this study we use a simulator core named SIMGEN, constructed by Virtutech, a successful Swedish company in the simulator field. SIMGEN takes advantage of the performance increase of the more efficient scheme mentioned above. SIMGEN is not a compiler. It's a tool that generates C-code from a specification language. The generated C-code is then, together with the user's own C-code, built with a standard C-compiler and the result is a simulator. This allows the user to build a simulator that fit his or her special needs.

The possibilities to make a user-specific simulator are very important for EIN. They want complete control of the simulators' functionality, because their simulators must be flawless. The existence of unused and maybe undocumented functionality is error prone and in some cases also inefficient. SIMGEN is therefore a suitable tool that helps them generate a simulator core. EIN can then add desired functionality such as MMU, to the core. SIMGEN is also proven to be fast, which Seffel concluded in his study [2].

2.3.2 The SIMGEN approach

SIMGEN uses something called intermediate code blocks, generated during the decode phase. These blocks have a size of 64 bits and contain one 32-bit pointer to a service routine, the rest is used for storing instruction parameters. The relation between machine instructions and intermediate code is illustrated in Figure 2.4. The figure shows how the decode phase fetches the parameters from the machine code and builds an intermediate code instance. From the name of the machine instruction the simulator can specify which service routine the intermediate code instance should point to.

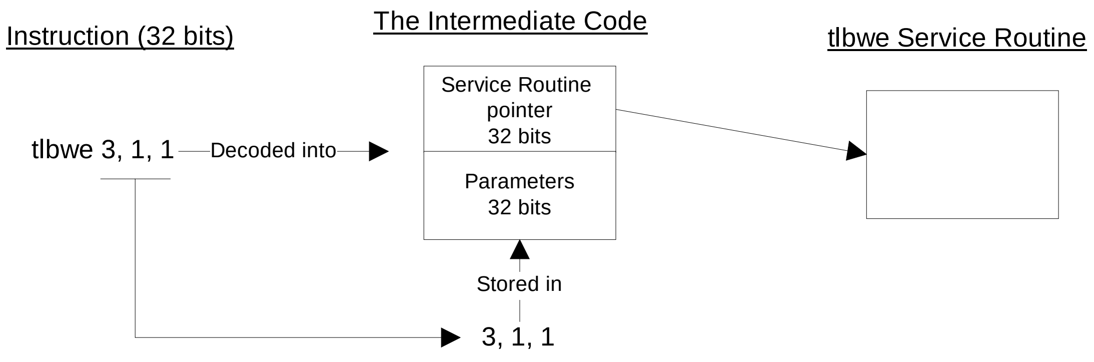


Figure 2.4 The intermediate code block.

All instructions defined in the simulator have their own service routines, which are invoked when an instruction is executed, see Figure 2.5. A service routine can be seen consisting of three separate parts. First there is `Prologue()` that fetches the parameters for a specific instruction. These parameters are stored in an intermediate code block as described above. When `Prologue()` is done, the instruction semantic is executed with the fetched parameters. The last part, `Epilogue(),` figures out which instruction is to be fetched and executed next. This approach allows all instances of one type of instruction to share a single service routine, executed with different parameters defined in each instance of an instruction.

`Prologue()` is generated by the SIMGEN tool but the semantic for each instruction and the epilogues are user-defined. The semantics are defined in the specification file. See Subsection 2.3.3.

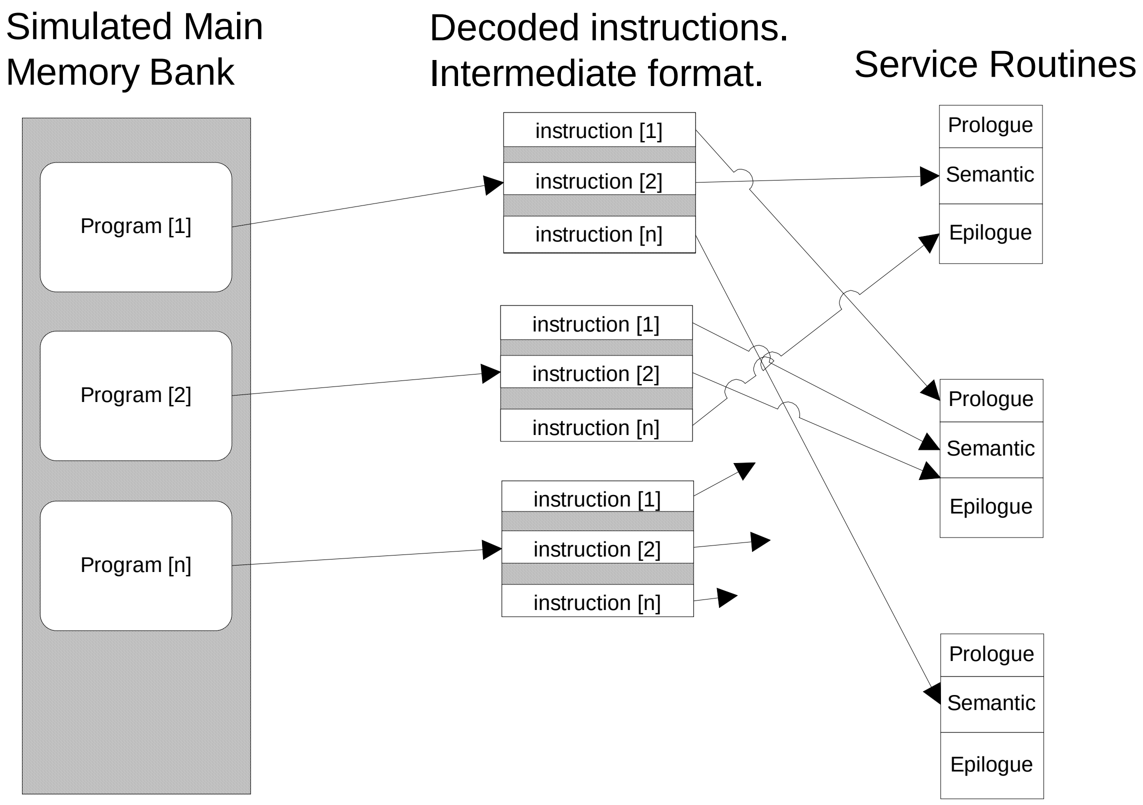


Figure 2.5 The relationship between the intermediate code and the service routines.

2.3.3 The Specification file

The input to SIMGEN is a file containing a specification of all instructions that the simulator is supposed to execute. This study focuses on the PPC processor. The PPC processor is a RISC (Reduced Instruction Set Computer) processor, with 32 bit long instructions. Each instruction has it's own unique bit pattern. These patterns are specified in the specification file one by one. For each instruction specified there is as a block of C-code that specifies the semantic of the instruction. This block is not examined by the SIMGEN tool, which only verifies that there are no instructions with conflicting bit-patterns. The semantic block is only appended to the generated C-code, which is later compiled by a C-compiler.

2.3.4 Output from SIMGEN

The output from SIMGEN contains three functions, which enable the user to control the simulator. These functions offer the user the possibility to define when the instructions shall be decoded and when to pass them on to the interpreter.

- `decode()` Reads raw memory and generates intermediate code for execution.
- `disassemble()` Reads raw memory and identifies an instruction and returns a character string with its assembler syntax with parameters and all.
- `interpreter()` Takes a pointer to an intermediate code block, which is supposed to be executed.

The biggest part of the output is the files with the service routines. The structure of the output files containing the service routines depends on which execution mode the simulator should execute in. For example in threaded mode, discussed in next subsection, the service routines are spread over numerous files. Each service is labelled and the labels are used for the “goto-jumps” which are used in threaded mode. In other modes the labels are replaced to fit other mechanisms such as function-calls etc. In this thesis work we use threaded mode and thus other modes will not be discussed.

2.3.5 Execution possibilities in SIMGEN

In SIMGEN the user must specify by himself when it’s time to decode instructions. This allows the user to construct a simulator for his specific needs and demands. For example, if the user knows that the simulator only runs different programs that don’t alter the memory where the loaded programs are stored, it may be appropriate to decode the complete program at start. But if the programs are too big or if they change during execution (self-modifying code) the user must decode during execution. This is done to ensure that the decoded instructions are up to date with the program loaded in the memory at any given time. See limitations in Chapter 1.

Another attractive feature with SIMGEN is the support for threaded-code. This means that the simulator runs partially as a `goto` program. The aim is to minimise the number of function calls, which create overhead in the execution due to stack manipulation and parameter passing. All service routines with their different parts are implemented as macros, which are expanded to a block of inline code. Inline code solves a task without performing any jumps or function calls, i.e. everything is done inline. Running the simulator in threaded-code mode the simulator starts with a function-call to the `interpreter()`. After the first instruction has been executed, the correspondent `epilogue()` makes a jump to the `prologue()` of the next instruction. This instead of returning from `interpreter()` and then again calling `interpreter()` with the next instruction. Figure 2.6 shows how these `goto` jumps move between the service routines.

Threaded code allows the user to register-map variables that are frequently used. A register-mapped variable does not need to access the main memory banks to retrieve its value. This means that the compiler ensures the user that the specific registers value won't be overwritten, which would result in information loss. This means that the variable is statically stored in a register. By avoiding memory accesses the execution time can be shrunken further.

The danger with register-mapped variables is that they are easily overwritten due to the fact that the number of registers is small, and that they are heavily used during execution. If one is not careful, a register-mapped variable might loose its information if there is a function call in the program. The scope of a register-mapped variable is not bigger than the current function. If a function call cannot be avoided the register-mapped variable must be stored in the main memory and restored after the function is done. It's therefore important to implement a threaded code simulator with as few function-calls as possible. Otherwise the performance gain might be lost in the overhead, generated when storing the register-mapped variables in the main memory. Another disadvantage with register-mapped variables is that the compiler has fewer registers to work with. Registers are vital during execution because all calculations are done in the registers. Having fewer registers at hand might cause additional memory accesses due to that intermediate results have to be stored in memory to make room for subsequent calculations. As a consequence of this, one has to be careful when choosing the number of register-mapped variables.

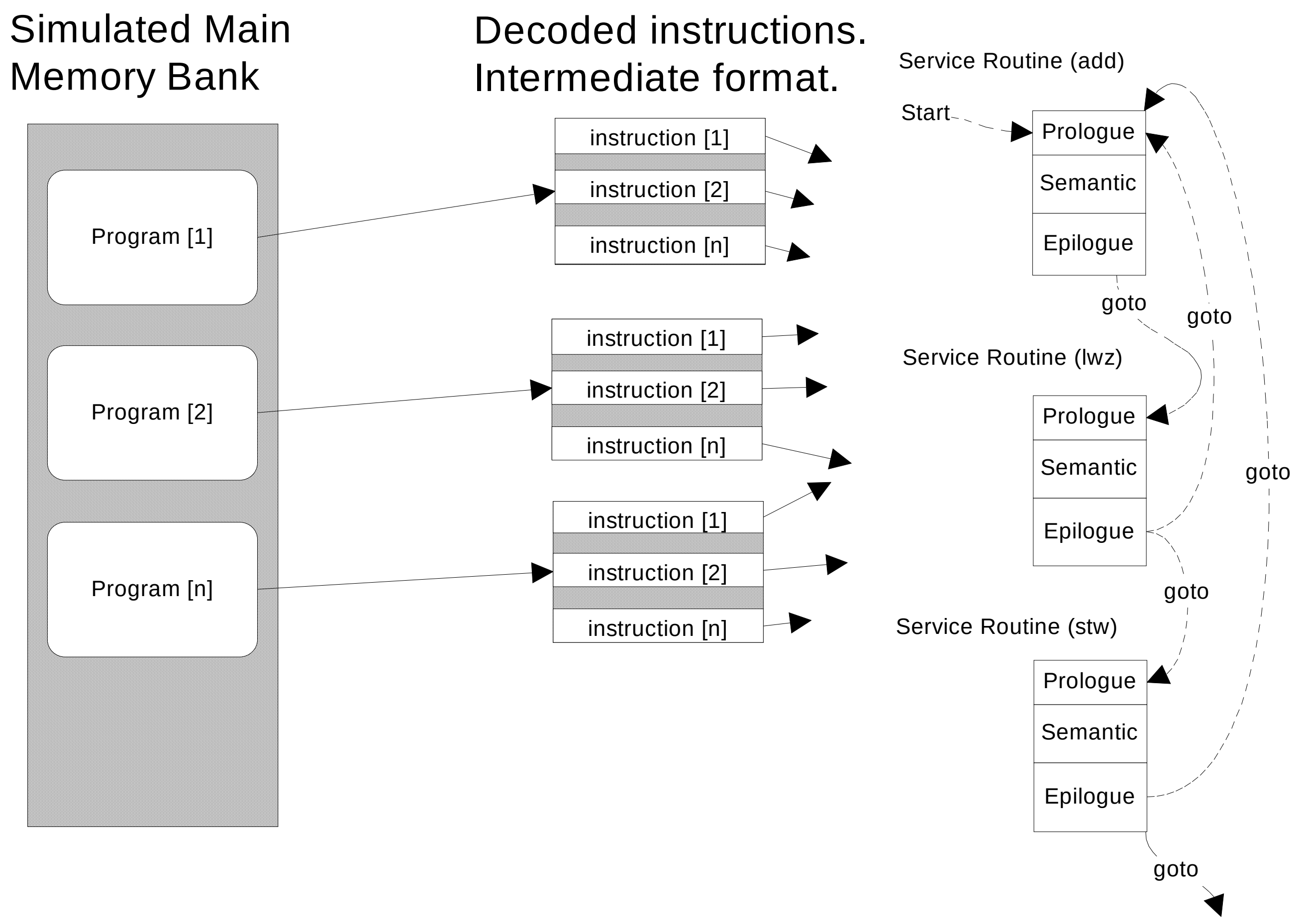


Figure 2.6 Execution flow in SIMGEN, threaded mode.

3 Requirements

This chapter describes the requirements for the simulator developed and implemented in this study. The requirements are divided into two parts. Section 3.1 lists the requirements specific for this thesis work. Section 3.2 contains a summary of the requirements from the thesis work done by Seffel in 1999 [2]. He built the UISA simulator, which this thesis is building upon and therefore its requirements also affect this thesis.

3.1 List of requirements for memory management implementation on the UISA simulator

The identifier before every requirement R.x is the identifier for a specific requirement specified in this section. These identifiers will be used later when evaluating the result.

R.1 Requirements from the UISA simulator should be fulfilled at highest possible extent.

Seffel's requirements will be considered: especially functional correctness and that the performance level should end up at a reasonable level, see Section 3.2. The extended simulator must not be so slow that it becomes impractical to use.

R.2 Generic memory management implementation

Implement a single module, which is compatible with PPC-405, PPC-750 and PPC-860 processors MM design.

R.3 MMU module

- TLB with 64 entries. Fully associated
- 8 entries DTLB shadow and 4 entries ITLB shadow.
- Support for all memory attributes. See Table 3.1 for details.

Upper part	***Lower part***
EPN – Effective Page Number	ZSEL – Zone Select Field
Size – The size of the page	EX – Execute enable
Valid – Bit marking if the page is valid	WR – Write enable
E – Endianess	W- Write-through
U0 – User defined attribute	I – Inhibited
TID – TLB Identifier	M – Memory coherent
	G – Guarded

Table 3.1 Memory attributes stored in the TLB

R.4 Registers

This list specifies the registers aimed to be implemented. All registers have a size of 32 bits.

- Memory management registers:
  1. Zone Protection Register (ZPR).
  2. Process identification (PID).
- Special Purpose Register General (SPRG 0-7).
- Machine State Register (MSR).
- General Purpose Register (GPR 0 -7).
- Exception handling registers:
  1. Exception Vector Prefix Register (EVPR).
  2. Exception Syndrome Register (ESR).
  3. Data Exception Address Register (DEAR).
  4. Save and Restore Registers (SRR0 - SRR3).
- Storage Attribute Control registers (Real Addressing Mode).
- Data Cache Cacheability Register (DCCR)
- Data Cache Write-thru Register (DCWR)
- Instruction Cache Cacheability Register (ICCR)
- Storage Guarded Register (SGR)
- Storage Little Endian Register (SLER)
- Core Configuration Register (CCR0)

Due to the fact that all cache instructions will be implemented as NO-OP (instruction with no semantic meaning), cache related registers is not modified during execution.

R.5 Instructions

Table 3.2 lists all instructions aimed to be implemented in the simulator. Note that all the general instructions inherited from the UISA simulator still are defined and implemented in the extended simulator. Load and store instructions must, though, be modified to support the MMU functionality.

Category	**Instruction**
MMU	
	`tlbia`
	`tlbre`
	`tlbwe`
	`tlbsx`
	`tlbsx`
	`tlbsync`

Cache	
	`Dcba`
	`Dcbf`
	`Dcbi`
	`Dcbst`
	`Dcbt`
	`Dcbtst`
	`Dcbz`
	`Dccci`
	`Dcread`
	`Icbi`
	`Icbt`
	`Iccci`
	`Icread`
Exception and interrupts	
	`Rfi`
	`Rfci`
	`Mtmsr`
	`Mfmsr`
Processor management	
	`Mtspr`
	`Mfspr`
	`Eieio`
	`Isync`

Table 3.2 Summary of instructions aimed for implementation

R.6 Interrupts

The following memory management-associated interrupts will be implemented. The four interrupts below are generated only when instruction or data address translation is enabled. All interrupt types are followed by a description for the events triggering the corresponding interrupt. For more specific details see Appendix E.

- **Data Storage Interrupt**

  When the desired access to the effective address is not permitted for some reason.

- **Instruction Storage Interrupt**

  When execution is attempted for an instruction whose fetch address is not permitted for some reason.

- **Data TLB Miss Interrupt**

  When a valid TLB entry matching the EA and PID is not present.

- **Instruction TLB Miss Interrupt.**

  When execution is attempted for an instruction for which a valid TLB-entry, matching the EA and PID for the instruction fetch, is not present.

3.2 Summary of requirements inherited from UISA simulator

The requirements for the UISA simulator, stated from EIN, focus on how a simulated PPC should work. These requirements are not outspoken to affect our implementation of the simulator. But its requirements also affect this thesis work because it is an extension of the UISA simulator. Requirements concerning efficiency and functional correctness are listed. These two are especially taken under consideration when developing the extended UISA simulator.

Efficiency is important because the simulator must not be so slow that it becomes impractical to use. Both operating system and user programs should be possible to execute in the simulated processor. Some factors that affect the executions speed of the simulator are:

- The performance of the computer that runs the instruction-set simulator
- The implementation of the simulator. The proportion between pure internal code and system calls to different surrounding components affects the runtime performance considerably.

An exact simulation of the processor on the instruction-level is very important for many reasons. For example, users would like to pick low-level information and statistics from the processor. It is very interesting to see how the memory accesses are performed, or how data and instruction cache hits are executed.

4 Solution

This chapter motivates and describes how we implemented the extensions on the UISA simulator. All optimisations are described with figures and code-examples.

4.1 Choice of programming language

C++ was at first the most preferable choice of programming language, because the language allows object-oriented design that is suitable for designing complex systems. But the SIMGEN tool creates a simulator core written in the language C. The functionality of the MMU is heavily nested into the execution flow in the simulator, and to mix C++ code into the core of the simulator, where the execution flow is controlled, would not be easy.

The simplicity and predictability of C is also preferable in contrast with C++, where there is plenty of overhead during execution. For example during the v-table look up, which is performed to handle the polymorphism functionality. Such overhead makes a negative impact on the efficiency, which is important to the simulator. Even if the unwanted overhead in polymorphism can be avoided, by not using the functionality, one should stay with the C language. The simplicity of C helps in creating a correct simulator, which is the most important objective in this study.

4.2 Memory management unit comparison

EIN is interested in having a generic simulator for all PPC processors used as RP's in their AXE phone switches. The three processors currently used are IBM PPC-405GP, IBM PPC-750 and Motorola PPC-860. To see if a generic solution is possible, an investigation has been made, where the differences and similarities between the three processors have been analysed with respect to the MMU. The analysis showed that there are significant differences in the MMU of the three processors, and that it would be hard to find a generic solution. Such a solution, which would include support for all processors, would not be easy to implement or to optimise for speed, which is important for overall simulator performance.

The different processors had different registers dedicated to MMU use, which added complexity. Certain registers existed in one processor, but didn't exist in all other processors. The three MMUs also included more or less hardware supported features/functions which otherwise are done in software as in the target processor PPC-405GP. The semantics of many instructions, aimed for the MMU, differed from each other but had the same syntax. The differences in the semantics are not a problem when implementing in C++ where one could use the inheritance functionality to implement all three MMUs in one single class tree. Implementing the MMU in a single class tree, all the MMUs will have the same basic interface and specific MMUs can extend its interface with specific functionality. But the possibilities in C++ aren't important enough to change programming language to C++, considering the discussion in Section 4.1.

The conclusion of the comparison is that three different MMUs have to be implemented separately, but with an interface to the MMU module as clean and generic as possible. This will make it possible to extract the MMU module designed for one processor and replace it with a MMU module designed for another processor. The interfaces are preferably built with C-macros, a block of C-code that is expanded in the program. To change from one MMU implementation to another one only has to redefine the macros.

With the results from the investigation in mind it seems reasonable to decide that the focus will be placed on the MMU of the IBM PPC-405GP only. For a complete, detailed comparison between the three processors, see Appendix B.

4.3 PPC-simulator architecture

This section gives a detailed view of how MMU functionality and interrupt handling have been implemented into the UISA simulator. It explains how the *Program Counter* (PC), which points at the next instruction to be executed, is translated from a logical address to a physical address. The relocation of the PC is the most central mechanism in the processor, which is also reflected in this implementation description. Subsection 4.3.1 gives a thorough description of how the PC moves, and it also introduces two additional PCs that exist in the simulator to improve the performance. The functionality of the additional two is to acquire the result of the most resent translation and cache it. This enables the simulator to avoid heavy calculations, which are done during translation.

Subsection 4.3.2 explains how address translation is done during data access and how this implementation caches a previous translation to improve the performance. Subsection 4.3.3 introduces the two functions that perform all the translations needed during execution. Then interrupts are introduced in Subsection 4.3.4. It explains how they can alter the execution flow in the simulator. Then the section continues with Subsection 4.3.5, which explains how the epilogues have been implemented. The epilogues are the most central facility in the simulator. Here the PCs are moved and the epilogues also detect the event of a MMU related interrupt.

Finally Subsection 4.3.6 explains why certain variables have been register-mapped. Register mapping variables is a performance tweak, which is utilised in this implementation.

To cope with the complexity introduced in the extended simulator, the implemented architecture resulted in approximately 1650 lines of code. These lines define the different epilogues, the MMU and other functionality as interrupts and more. This is basically the core of the simulator. The specification file for SIMGEN, which defines the instructions and their semantic, consist of roughly 3000 lines of code.

4.3.1 The program counters

In a real processor there exists only one PC. However the simulator architecture implemented in SIMGEN uses intermediate code, as shown in . Therefore a need of two extra PCs arises. The names of the PCs have been numbered: `PC1`, `PC2` and `if_PC`. `PC1` should be seen as the PC that exists in a real processor. `PC1` moves exactly as a real PC in a processor but in this simulator it has two companions, i.e. `PC2` and `if_PC`. How if_PC and PC2 behave and why they exist is explained below.

Intermediate format PC (`If_PC`): When the simulator starts up it loads its simulated main memory with the programs that is to be executed. The entire programs are then decoded into *blocks of intermediate code* (icode blocks), which limits the simulator to only execute preloaded programs, see the limitations in Section 1.3. The icode blocks correspond to the programs loaded in the memory. When the icode blocks are created the simulator defines their start and end addresses in the memory. Those two icode block attributes are used to identify the correct icode block for a specific physical address.

When the decoding is complete, the simulator fetches the instruction at a given start address. This instruction has its corresponding icode in one of the previously created icode blocks. The relationship between a physical address and an icode address, which specifies a specific icode instance inside a block, is linear. A move of 4 bytes in the physical address space is equivalent to a move of 8 bytes in the icode block, because an instruction in the processor uses 32 bits and a decoded instruction uses 64 bits, a 1:2 ratio.

The process of translating the physical address, defined in the `PC1,` to an address for an icode instance in an icode, is very time consuming. To avoid this translation for every instruction fetch, an intermediate PC named `if_PC` was created. `if_PC` keeps the translated value as long as possible and moves in parallel with `PC1`, in a linear fashion. `PC1`’s value is a base address plus an offset into the memory. In contrast `if_PC` has a several base addresses depending on which program being executed, but its offset changes hand in hand with `PC1`, only a constant value separates their moves lengthwise.

To be able to detect when `if_PC` needs to be recalculated, the icode block currently being used, contains upper and lower boundaries. These boundaries are defined for `PC1` if the MMU functionality is turned off, or `PC2` described below, if the MMU functionality is turned on. The relationship between the PCs is shown in . When `PC1`’s or `PC2`’s value is outside these boundaries, the `if_PC` needs to be recalculated and new boundaries to `PC1` or `PC2` are assigned. This occurs when `PC1` or `PC2` moves/jumps from one loaded program to another in the memory.

The block architecture together with `if_PC` offers great flexibility and the utilisation of memory is next to minimal. Jumps between programs are uncommon and therefore recalculation of `if_PC` is rare, which is good for overall performance of the simulator.

Physical PC (`PC2`): The need of the third PC arises when the MMU functionality is turned on. The simulator first needs to translate the logical address stored in `PC1` into a physical address, `PC2`. Then it has to translate the physical address, `PC2`, to the corresponding icode instance, `if_PC`. In Figure 4.1 one can see that `PC1` contains a logical address and `PC2` a physical address. As with the relationship between `PC1` and `if_PC`, `PC1` and `PC2` move in parallel, they simply have different base addresses but the same offset. The translation between the logical and physical addresses is unfortunately to slow to be done for each instruction fetch and data access. This motivates `PC2` as the third PC, which caches the most resent translation.

`PC2` has boundaries just as `if_PC`. The boundaries mark the start and end of the current *page* (the smallest block of memory that is guaranteed to be the non-fragmented in the physical memory, also described in 2.2.2) from where the instructions are being fetched. When `PC1` works in the logical address space, `PC2` contains the address to the corresponding page in the physical address space. As long as `PC1` works in the same page, `PC2` don’t need to be recalculated. But when `PC1` moves from one page to one other, or, for example, the PID register changes value, `PC2` needs to be recalculated. See Figure 4.2 below for details, which describes the number of conditions that must be true to ensure the validity of the cached mapping between the logical address and the physical address stored in `PC2`.

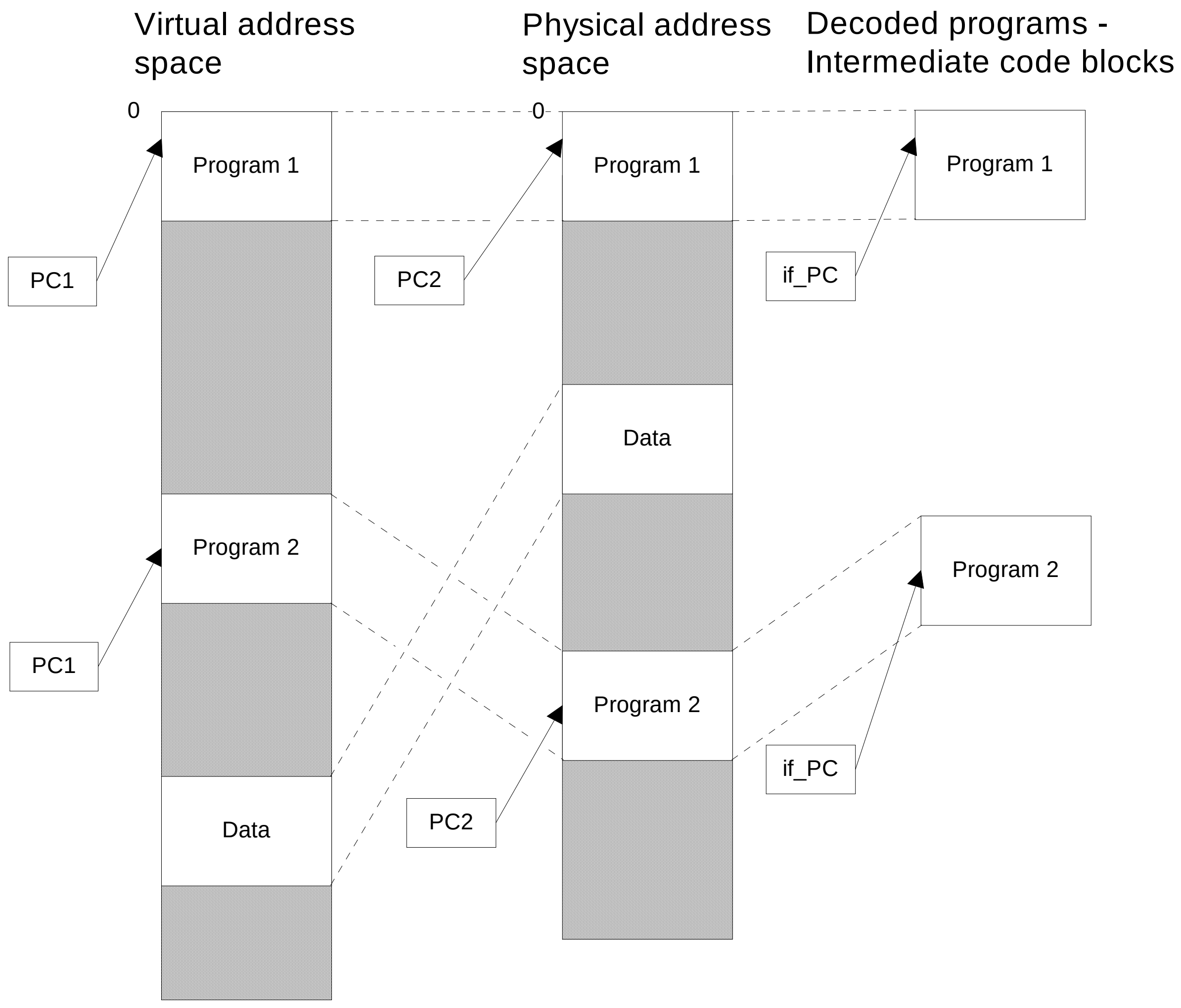


Figure 4.1 PPC simulator PC architecture

This implementation forces the simulator to check if the cached translation is still valid, at each instruction fetch, to ensure that no new translation has to be done. This is time-consuming, but the alternative, forcing the simulator to perform the translation for all instructions fetches, is far worse.

When the MMU functionality is turned off, the value of `PC2` is irrelevant, because there is no translation to be cached between the logical and physical address space. `PC1` can be seen jumping back and forth between the physical and logical address space, when the MMU is turned on and off.

Figure 4.1 shows the three different PCs that exist in our architecture when the MMU functionality is turned on. It shows a configuration where two different programs are loaded into the memory of the simulator. Program 1 is loaded with base address 0 in the physical address space and one can see that the MMU is configured to map the page with the same base address in the logical address space, a 1:1 mapping, i.e. the same address/page in logical and physical address space.

The other loaded program, Program 2, has a different mapping in the MMU, with a different base address in the logical space. When Program 2 executes in the logical address space, `PC1` and `PC2` move in parallel with each other. But it should be pointed out that the relationship between `PC1` and `PC2` could become invalid during runtime, with the result that `PC2` must be recalculated from `PC1` at the next instruction fetch. The chart described in Figure 4.2 summarises when the relationship is invalidated. The Machine State Register (MSR), and Process ID register (PID) that are used in the figure are described in detail in Appendix C. For now we only need to know that the MSR controls the MMU functionality, and that the PID identifies the process currently executing. Figure 4.2 also shows that the cached translation is invalidated at the event of an interrupt. The interrupts are described in Subsection 4.3.4.

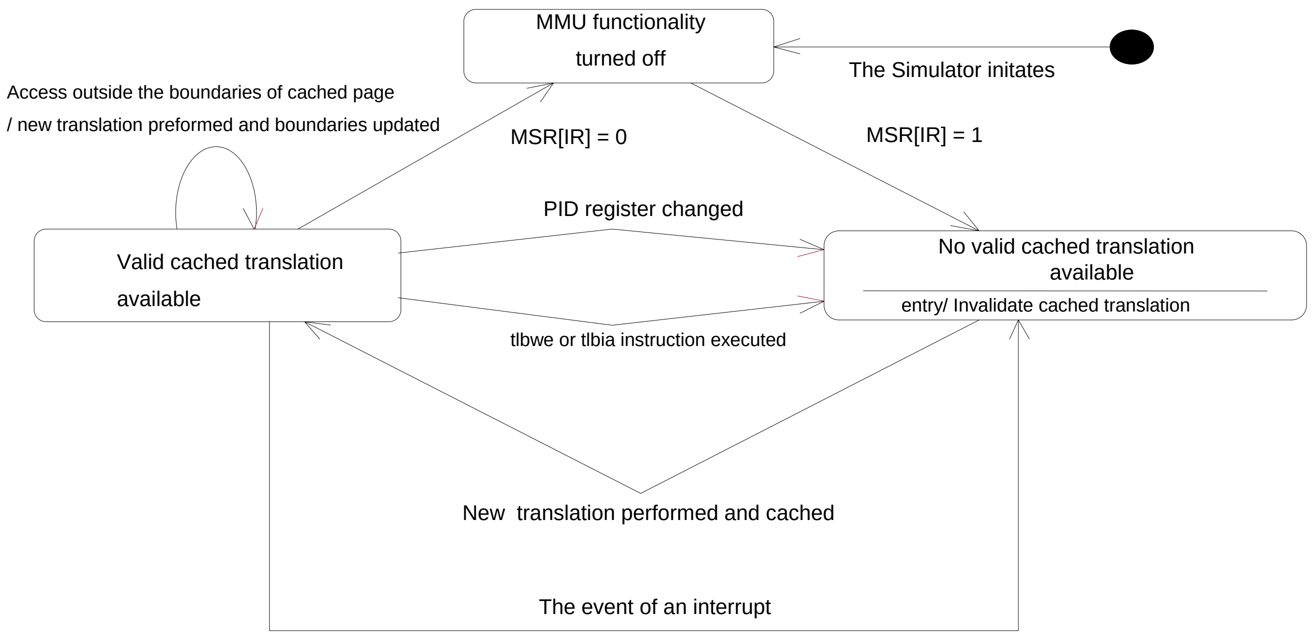

MMU functionality turned off
The Simulator initates
Access outside the boundaries of cached page / new translation preformed and boundaries updated
MSR[IR] = 0
MSR[IR] = 1
PID register changed
Valid cached translation available
No valid cached translation available
entry/ Invalidate cached translation
tlbwe or tlbia instruction executed
New translation performed and cached
The event of an interrupt

Figure 4.2 State-chart for cached translation between `PC1` and `PC2`

4.3.2 Data address translation

The MMU can be seen as two separate parts, one part that handles instruction fetches and another one that handles load and store accesses to the memory. However they work with the same TLB.

To increase the performance of the data address translation, the simulator caches the latest data address translation. This increases the performance when running programs that repeatedly use the same page for data access over and over again. The cached translation must of course have conditions to ensure its validity at all times during execution, just as for `PC2`, described in Subsection 4.3.1. Checking these conditions decrease the performance but the alternative here is not preferable. The conditions that control the validity of the cached translation are summarised in Figure 4.3.

Our implementation has two caches for data translations, one for accessing the memory with load instructions and another for store instructions. This separation is implemented because it may be allowed to load from a page but not to write to the page.

Figure 4.4 contains the code, which checks the validity of the current cached translation, and if all conditions are true, a fast translation can be performed. This translation is done in the figure at the lines 9 to 10, for store access, and at 24 to 25, for load access. If one of the conditions is broken, a new translation must be preformed for the specific access type and a new cached translation is set up. This is done in the figure at the lines 13, 17, 28 and 32.

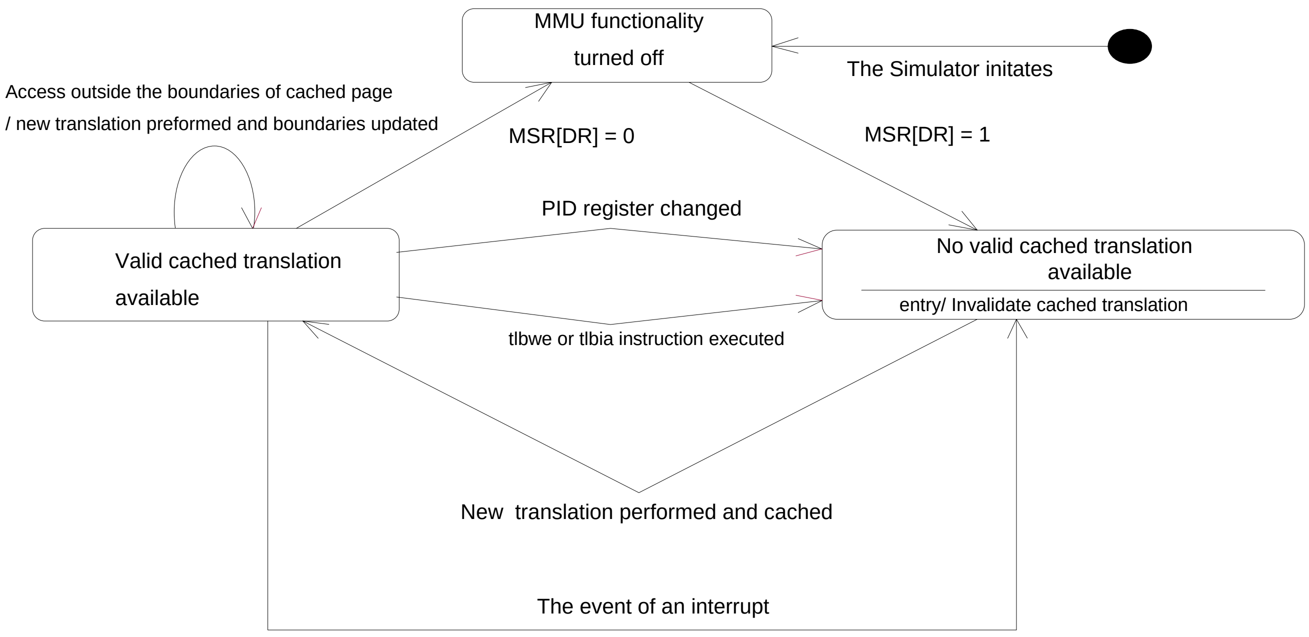

MMU functionality turned off
The Simulator initates
Access outside the boundaries of cached page
/ new translation preformed and boundaries updated
MSR[DR] = 0
MSR[DR] = 1
PID register changed
Valid cached translation available
No valid cached translation available
entry/ Invalidate cached translation
tlbwe or tlbia instruction executed
New translation performed and cached
The event of an interrupt

Figure 4.3 State-chart for cached data translation

```
#define D_LOOK_UP(ret, addr, type_of_access)                            \
{                                                                       \
  UW32 clear_ea_mask=0;                                                 \
  switch((type_of_access)){                                             \
    case D_STORE:                                                       \
      if(cached_store_page.valid){                                      \
        if( ((addr) >= cached_store_page.start) &&                      \
            ((addr) <= cached_store_page.end)){                         \
              GET_EA_MASK(clear_ea_mask,cached_store_page.size);        \
              (ret) = ((addr) & clear_ea_mask) | cached_store_page.page; \
        }                                                               \
        else{                                                           \
          RUN_OUTSIDE( (ret) = TLB_LOOK_UP((addr), D_STORE));           \
        }                                                               \
      }                                                                 \
      else{                                                             \
         RUN_OUTSIDE( (ret) = TLB_LOOK_UP((addr),  D_STORE));           \
      }                                                                 \
      break;                                                            \
    case D_READ:                                                        \
    if(cached_read_page.valid) {                                        \
        if( ((addr) >= cached_read_page.start) &&                       \
            ((addr) <= cached_read_page.end)){                          \
              GET_EA_MASK(clear_ea_mask,cached_read_page.size);         \
              (ret) = ((addr) & clear_ea_mask) | cached_read_page.page; \
        }                                                               \
        else{                                                           \
          RUN_OUTSIDE( (ret) = TLB_LOOK_UP((addr), D_READ));            \
        }                                                               \
      }                                                                 \
      else{                                                             \
        RUN_OUTSIDE( (ret) = TLB_LOOK_UP((addr), D_READ));              \
      }                                                                 \
      break;                                                            \
  }                                                                     \
}                                                                       \
```

Figure 4.4 The code for caching data address translations

4.3.3 Functions performing the translations

The functions mentioned in this subsection are the only two functions that the simulator calls during the execution. However there exist several functions that are used to set up the simulator before execution and functions that print the state of the processor after the termination of the simulator.

- `GET_IF_PC(UW32 addr)`

This function takes one parameter, `addr`, and matches it to an address in one of the icode blocks. If a translation is possible the `if_PC` is updated and new boundaries for the cached translation are calculated. If no valid translation is available the simulator terminates.

The icode blocks are, in our implementation, stored in a list structure. This function uses the boundaries, defined in each block, to match the address, `addr`, with an address to a specific icode instance in one of the blocks in the list.

- `UW32 TLB_LOOK_UP (UW32 addr, int TYPE_OF_ACCESS)`

This function takes two parameters, `addr` and `TYPE_OF_ACCESS`. The first parameter, `addr`, is the logical address that is supposed to be translated to a physical address. The second parameter, `TYPE_OF_ACCESS`, is a descriptor that tells the MMU what kind of access is going to be performed. There are three possible access types: instruction fetch, load data and store data. This information is used when checking if the memory may be accessed. For example, if storing data at a page is prohibited or if instructions fetches are disallowed.

The `TLB_LOOK_UP()` function returns a translation from a logical address. This is done if the address, `addr`, can be translated and the type of access to the page is allowed. It also recalculates the boundaries for the cached translation, `PC2`. But if there isn't a matching entry in the TLB or if the access type is not allowed, the function returns an address to an interrupt handler specific for each exception. The function also sets a global variable, a flag, which indicates that an interrupt is discovered during translation. This flag helps the epilogues to notice the event of an interrupt.

To find a valid entry in the TLB this function iterates through all the 64 entries, containing the information required by the MMU for performing the translation. It stops when it finds a matching entry or when the function concludes that no matching entry in the TLB exists. The occurrence of multiple hits in the TLB is seen as a programming error and therefore the function can stop at the first hit in the TLB. This avoids unnecessary iterations in the TLB. In the real processor, PPC-405GP, multiple entries matching the same page is also seen as a programming error and the outcome of the look up in the processor is undefined.

4.3.4 Interrupts

Interrupts add more complexity to the execution flow in the simulator. An interrupt is a drastic change in the execution flow. At an interrupt the simulator executes a predefined program that is loaded at a predefined address, depending on what kind of interrupt that was triggered. At the event of an interrupt the processor's state is stored into two registers and then the state is changed to a predefined state. The registers defined to store the state are the Save and Restore Registers (SRR0-SRR3). For detailed description of the register see Appendix C.

The program assigned to handle the interrupts, terminates by executing either the instruction `rfi` or `rfci`. These two instructions restore the state of the processor to the state at the event of the interrupt. The state also includes the value of the PC, `PC1`, in the simulator.

The simulator implemented in this thesis work supports four different interrupts. See Appendix E for further details.

4.3.5 The epilogues

The different PCs are controlled in the epilogue part of the service routine. There are four different kinds of epilogues. The first one moves `PC1` to the next instruction in the memory. The second epilogue branches relative from the current PC (`PC1`), i.e. it jumps a specified number of instructions in either direction. The third epilogue branches to an absolute address. The final one is a special epilogue that handles the event of data related interrupts. This one jumps to one of the two possible programs that handle data access interrupts. The handling of interrupts occurring when fetching instructions are nested into all the epilogues except the last one.

The UISA simulator, created by Seffel, has only three short and simple epilogues. Our simulator is more complex and has to cope with a very unpredictable execution flow. Therefore the complexity of the epilogues has increased. This decreased the performance and the ambition to create a simulator with the excellent performance of the UISA simulator had to be abandoned. It should though be pointed out that the UISA simulator is a very primitive simulator with no interrupt handling or MMU functionality. Figure 4.5 shows the epilogue from the UISA simulator that moves the program counter to the next instruction in the memory. Figure 4.6 shows the same epilogue implemented in the extended simulator and when comparing the two epilogues it's obvious that the performance is affected negatively.

The epilogues implemented in our simulator have a complex nature because of the optimisations. All boundaries must be checked to ensure a correct translation, and if some conditions are no longer true, a new translation must be preformed. The worst case is when both `PC2` and `if_PC` must be recalculated before the next instruction can be fetched.

When performing one of these translations, a function-call must be performed to either `TLB_LOOK_UP()` or `GET_IF_PC()`, described in Subsection 4.3.3. Before making this function-call, the simulator needs to store the register-mapped variables in the main memory. This is done by the macro `RUN_OUTSIDE()`, which is expanded in the epilogue during the compilation of the simulator. The use of macros makes the epilogues easier to understand and the macro can be seen as function call, but it isn't. The macro takes one parameter, which is the operand supposed to be executed outside the main function of the simulator. Outside the main function the register-mapped variables are no longer protected and therefore most likely overwritten, see Subsection 2.3.5. The operand is either a function call to `TLB_LOOK_IP()` or to `GET_IF_PC()`.

In Figure 4.6 one can see how unnecessary translations is avoided, when `PC1` and `PC2` are between their respective boundaries. Then all the PCs move the in same way which can be seen in the figure at the lines 3, 7 and 9. In the case of the `epilogue()` in the figure the PCs are increased with a small constant. `if_PC` moves eight bytes forward and `PC1` and `PC2` both moves four bytes forward. The worst-case scenario is as mentioned when both `PC2` and `if_PC` needs to be retranslated. This happens when `PC1` moves from one page to another or when it doesn't exist a valid translation between `PC1` and `PC2`. The recalculations are preformed at lines 16 to 17 and 21 to 22.

When the MMU functionality is turned off, the statement on line 4 is false, the epilogue only needs to check the cached translation between `PC1` and `if_PC`. This is done at lines 26 to 32. Lines 33 to 35 handle the event of an interrupt, which might occur when the MMU is turned on and a TLB look up is performed, at line 16 or 21. The two last rows in the figure specify the new parameters for the next instruction and then there is a *goto* jump to the service routine of the next instruction.

```
#define epilogue()                      \
icount++;                               \
if_pc ++;                               \
rOP=if_pc->ic_Parameters;               \
goto *(if_pc->ic_Handler);
```

Figure 4.5 Epilogue from the UISA simulator

```
#define epilogue()                                                  \
icount++;                                                           \
PC1 +=4;                                                            \
if(MSR & IR){                                                       \
  if(page_border.valid != 0){                                       \
    if(PC1 <= page_border.end - 4){                                 \
      PC2 += 4;                                                     \
      if(PC2 <= (curr_block->end_addr - 4)){                        \
        if_pc++;                                                    \
      }                                                             \
      else{                                                         \
        RUN_OUTSIDE(GET_IF_PC(PC2));                                \
      }                                                             \
    }                                                               \
    else{                                                           \
      RUN_OUTSIDE(PC2 = TLB_LOOK_UP(PC1,I_FETCH));                  \
      RUN_OUTSIDE(GET_IF_PC(PC2));                                  \
    }                                                               \
  }                                                                 \
  else{                                                             \
    RUN_OUTSIDE(PC2 = TLB_LOOK_UP(PC1,I_FETCH));                    \
    RUN_OUTSIDE(GET_IF_PC(PC2));                                    \
  }                                                                 \
}                                                                   \
else{                                                               \
  if(PC1 <= curr_block->end_addr - 4){                              \
    if_pc++;                                                        \
  }                                                                 \
  else{                                                             \
    RUN_OUTSIDE(GET_IF_PC(PC1));                                    \
  }                                                                 \
}                                                                   \
if(!I_SUCCESS){                                                     \
  PC1 = PC2; I_SUCCESS = 1;                                         \
}                                                                   \
rOP=if_pc->ic_Parameters;                                           \
goto *(if pc->ic Handler);
```

Figure 4.6 An example of one of the epilogues from the extended simulator

4.3.6 Distribution of register-mapped variables

The register-mapped variables, in a simulator, should be the variables most frequently used, see Subsection 2.3.5. In this simulator the different PCs are a perfect members to the small group of variables that are allowed to be register-mapped. The PCs are used after every execution of an instruction: more specifically they are accessed in the epilogues. See subsection 4.3.5.

Another suitable member is the MSR register, which is frequently accessed when checking if the MMU functionality is turned on. This is done in the epilogues and in the semantics of all load and store instructions, where main memory is accessed.

`rOP` is a variable that is also used during every execution of an instruction. It is filled with the parameters stored in the icode instance. `rOP` is accessed by the prologues, where the parameters are extracted and stored in separate variables. This makes it a suitable register-mapped variable.

The General Purpose Registers (GPRs) are registers used as storage, for example, when a memory block is loaded from the memory into the processor. The registers are then, for example, added with each other. All the basic mathematical and logical operations are done on these registers. There are 32 GPRs, and in this extended simulator architecture they are stored in an array. The base address of this array is a suitable subject for register mapping.

The Condition Register (CR) is used during execution and it contains 8 entries of 4 bits each, which are used to store the result of logical instructions performed. The result is then used, for example, to decide if a branch is to be taken or not. This is done frequently during the execution of a program, which makes CR a suitable member.

All the suitable members described above in this subsection, `PC1`, `PC2`, `if_PC`, `rOP`, GPR base address, CR and MSR, are implemented as register-mapped variables in our simulator.

4.4 Execution environment

The simulated environment is to be executed in the Unix-environment on Sun SPARC workstations. This is because threaded code is used and it’s only fully supported in the SPARC environment. The possibility to register-map variables, see Subsection 2.3.5, is greater in the SPARC environment than in a standard PC environment with an INTEL or AMD processor. Threaded code is, though, possible to execute in a standard PC, with an INTEL or an AMD processor, but with less or no possibilities to register-map variables. The SPARC environment is currently the main execution environment for EIN, but they are migrating to the more affordable LINUX environment, which is run on cheaper hardware.

4.5 Inline code vs. function calls

This extended UISA simulator has been built with threaded code, described previously in Section 2.3. Implementing the simulator with threaded code has not been done without problems. A program written as a threaded code program is basically a gigantic inline-code program. An inline-coded program is a program with no function calls. The function calls are replaced with the contents of the wanted function, which can be seen as a sequence of instructions (inline code blocks).

As mentioned before there are no function calls in threaded code programs. Instead goto jumps are performed to the next service routine in the inline-code. The drawback with threaded-code is that enormous amount of code is generated during compilation, when the macros are expanded. This is a problem for the compiler, because it cannot compile files that are too big.

To avoid the problem we decided to take a small step away from the threaded code mode. Instead of expanding the MMU functionality as inline blocks in every service routine, all service routines have a function call to `TLB_LOOK_UP()`and to `GET_IF_PC()`. These function calls must be preceded with storing all register-mapped variables into the main memory, and when these functions is done the register-mapped variables have to be restored. The storing and restoring create overhead in the execution, and in this implementation we have tried to avoid the function calls as long as possible. By caching the result of the most recent translation, new translations can be avoided in most cases. See subsections 4.3.1 and 4.3.2 for further details of the cached translations and their constraints.

Another solution for this code problem would have been to tell SIMGEN to create more files. The user may define the number of files and in that manner one can spread the code over an amount of files, so that each file doesn't exceed the size limit. This approach solves the problem with big files, but the downside with this is that the amount of code is still very big. The time it takes to build the simulator, consisting of a large amount of code, is reasonably long even on a very fast computer. We decided to go with the function calls because of the decreased compile time. Faster compiling makes it easier to develop the simulator in a more interactive way. Often a programmer notices that he needs more debugging information, and as a programmer one wants to add this debugging information swiftly and then recompile. Our approach with function calls, resulted in a decrease of the program size by a magnitude of 10 and the compile time also decreased in a similar manner.

4.6 Input data to the Simulator

The simulator implemented in this study can be loaded with multiple programs. These programs are files containing instructions. The structure of the files and the instructions follow the format of “ELF 32-bit MSB relocatable PowerPC version 1”. To generate these files one can use a standard GNU-C compiler for PPC, which is instructed to generate an assembler code file. This assembler file must then be altered to fit the simulator, which is done by removing initialising code that the compiler automatically appends to the program, which the simulator doesn’t support. The modified assembler file is then linked with the GNU-compiler and the last step is to extract the program from the binary file and discard all of the unnecessary data remaining. The result is a file containing a program without the unnecessary/unsupported parts that a standard C program includes, such as IO routines, debugging facilities, etc.

5 Evaluation of our solution

This chapter presents an evaluation of our simulator based on the requirements in Chapter 3. The most important aspects with this extended simulator are functional correctness and efficiency. What can be better than a correct and fast simulator? Therefore continuous testing has been performed throughout the entire phase of development. This testing has assured us that the simulator executes correctly. The efficiency of the simulator is a very interesting attribute, which plays a big part in this chapter.

5.1 Test method

To test the correctness and efficiency of our implementation, a test environment had to be initiated. The TLB, which is usually initiated by the operating system, now had to be initiated by us. To do this we created our own small operating system. First a number of test programs were created. Since Seffel used Tower of Hanoi as his test and benchmark program, we considered it suitable to be used as our benchmark program as well.

TLB initialising and interrupts handling programs were written in C. Adding blocks of assembler in the code and using register-mapped variables, gave us complete control of what happened during execution and which instructions were used. The TLB is initialised with the MMU related instructions, added to the instruction-set in this extended simulator, see Appendix D. The TLB set up program is loaded to the address that is the start address for the execution, normally zero. Next step is to load the program for interrupt handling. We've implemented one interrupt routine, which handles the TLB instruction-miss interrupt. There are three more interrupts currently supported in the simulator: data storage interrupt, instruction storage interrupt and TLB data-miss interrupt. These interrupts have been tested thoroughly as well. Finally the main program, for example, Tower of Hanoi is loaded into the simulators memory, at an address that fits the mapping to be set up in the TLB, by the TLB initialising program during execution.

We decided to initiate the TLB so that all programs should run with start address zero, but the actual program is loaded at a totally different address. The TLB initialising program initialises the TLB with information that tells the MMU to translate the addresses as described above.

When the TLB has been initiated and the MMU functionality is turned on, the parameters that control the execution flow are changed (a change in the processor state). This introduces a problem at the next instruction fetch, which follows after the change of state, is done through the MMU. This creates a problem because an explicit branch might be taken unwillingly. This means that the next instruction to be fetched can be an instruction in an address totally different from the one the user wants to fetch. A trick to solve this is to force the simulator to trigger an interrupt, in this case a TLB instruction miss exception. When the interrupt is triggered, the processor's state is saved into special registers and is also available for modifications. This interrupt handler can be used, for example, to set the PC to zero, which before the event of the interrupt had a totally different value. When the interrupt handler returns the control to the previous program, it starts to execute at address zero. This address is transparently translated by the MMU to a totally different address during execution, in this case the address where the Tower of Hanoi program is stored.

5.2 Result evaluation

To evaluate the result a number of measurements have been performed. The implementation has been carefully tested so that everything works properly. An interesting property to be evaluated is the performance of the simulator. This because this simulator is only the core of a complete simulator and it's most likely to be extended with more complex functions, as JIT decoding etc. The efficiency of the simulator is depending on the core and it is evaluated when the MMU is turned off and when it's turned on in different modes. Finally this section compares the result of our implementation against the requirements listed in Chapter 3.

5.2.1 Performance evaluation and measurements

The UISA simulator is not included in this performance study, since it has a very raw and simple design with no MMU or interrupts. It can only execute programs that don't generate interrupts or other exceptions that alter the execution flow.

The MMU is a complex unit and it's therefore impossible to achieve the same level of performance when the MMU is turned on as when it is turned off. There have however been modifications to make the simulator more efficient, see Section 4.3. Table 5.1 below shows the amount of time taken to perform Tower of Hanoi with 24 disks on a SPARC Ultra 10. This height of the tower leads to 16.777.215 recursive calls, which is high enough to disregard the overhead aspects of the execution. To show how the different modes available in the MMU affect the performance, the table contains four different results.

MMU turned off	Translation of instructions enabled	Translation of data and instructions enabled	Translation of data and instructions enabled with optimisation of data translation
1 min 50 sec	2 min 13 sec	5 min 16 sec	3 min 35 sec

Table 5.1 Performance measurements of CPU-time utilized during execution

Table 5.1 shows the performance decrease when the extended simulator uses more and more MMU functionality. It also shows how the optimisations for instruction fetch and data access, see subsections 4.3.1 and 4.3.2, have increased the performance. The translation performed during the instruction fetch, performed in the `epilogue()`, has been cached in this implementation since the very first version. Therefore there is no value for a completely unoptimised simulator. However the values in the table give us a hint that the optimisation for the instruction fetches is successfully implemented. When the MMU is turned off, the simulator is as fast as it can be. When the loaded programs enable instruction translation, the performance decreases. The performance loss is not big and execution time only increases with 20 %. The optimisation of the data translation is very much the same as the optimisation for instruction fetches. And the result of the implementation was not a surprise, because of the previously evaluated optimisation. The improvement was a 32 % decrease of execution time.

5.2.2 Requirement fulfilment

In this subsection the requirements stated in Chapter 3 are compared with the outcome of this thesis work. Most of the requirements have been fulfilled.

Here follows a list of the requirements, with a discussion for each requirement and its fulfilment.

R.1 Requirements from the UISA simulator should be fulfilled at highest possible extent.

The extended simulator is slower than the UISA simulator. On the other hand this simulator has much more complex features implemented, which creates overhead during execution.

R.2 Generic memory management implementation

With support from our investigation, see Section 4.2, we decided that the different memory management schemes should be separately implemented, and that there should not be a generic memory management implementation.

R.3 MMU module

The MMU module is fully implemented with a fully associated TLB with 64 entries. All memory attributes are also supported.

The 8 entries DTLB shadow and 4 entries ITLB shadow are not implemented. These shadows are only a performance enhancement in the processor. Implementing them in the simulator would only affect the performance of the simulator badly.

R.4 Registers

All registers are implemented but some of the are not used/altered during execution, such as cache control registers. This because of the lack of time and that their functionality is out of this thesis scope.

R.5 Instructions

All instructions in the Table 3.2 are defined in the simulator, but the cache instructions are left as no-op as explained in R.6 in Section 3.1. This also involves `tlbsync`, due to the fact that it has no semantic meaning in the PPC-450GP processor. `eieio` and `isync` are synchronising instructions. Synchronising is a problem in a real processor. In an instruction-set simulator with no instruction pipelines, which increase performance in the real processor, the synchronising instructions can be implemented as no-op.

R.6 Interrupts

All the memory management-associated interrupts were implemented successfully.

5.3 Possible extensions

5.3.1 Statistics

Both MMU and cache have a great influence on the performance. Hence it would be very interesting to study statistics on, e.g. the number of “Page faults” etc. This makes statistic collection a suitable extension.

5.3.2 Interrupts

Interrupts were introduced to the simulator in this study. Only the four MMU related interrupts were implemented. One type of interrupts that is left to be implement is program interrupts. These interrupts occur in the following situations.

1. A privileged instruction is executed in Problem State (MSR[PR]==1).

2. An illegal instruction is executed.
3. Executing a trap instruction with conditions satisfied.
4. A TLB instruction is executed when translation is disabled.

Especially trap interrupt is very interesting and therefore a suitable target for extension. A trap interrupt can be used to extend the instruction-set with custom made programs that act as an instruction. The users of the processor see this trap functionality as a nice and usable functionality.

There also exist external interrupts in the target processor. Supports for them are also left for future work.

5.3.3 Real addressing mode

The simulator lacks support for access protection in real addressing mode, which might be an interesting extension.

5.3.4 Cache statistics

All cache instructions were implemented as NO-OP and cache related registers are not modified during execution. Cache statistics are very interesting but it's left as a potential extension. Statistics are used when evaluating software, debug and enhance the performance of the software. Cache misses and TLB misses are time consuming and should be avoided.

5.3.5 More performance enhancements

Due to lack of time we did not use SIMGEN's own built-in features, which can increase performance. One feature is the virtual instruction approach, where one takes advantage of the existence of combinations of instructions and parameters that are frequently executed together. A new service routine is created for each of these combinations of instruction and parameters. Then parameters can be statically defined in the new service routine and the need for the parameter extraction, the `prologue()`, is circumvented.

The virtual instruction tweak was successfully used in the UISA simulator and the same approach will most likely be successful in this implementation. But it should be pointed out that this optimisation is based on statistics gathered during execution. Therefore the simulator will increase its performance when executing the same program after the optimisation. However when running different programs, the increase in efficiency might be unnoticed or very small, because the differences in the program can be substantial regarding the use of instructions, which the optimisation is based on. One does only specialise the simulator towards the execution of one special program and the increase of efficiency for all programs is small.

5.4 Conclusion

Our implementation satisfies almost all requirements stated in Chapter 3. This simulator doesn’t gather statistics. It only runs the programs loaded into its memory. It’s up to the user to decide if he wants to implement statistics gathering. Adding this functionality, however, implies more load for the simulator. Therefore it should be added only if needed.

The performance of the simulator is not yet investigated in a more thorough way. Comparing the CPU-time gives us a hint. The more complex service routines, together with their epilogues, make it very hard to calculate the amount of SPARC instructions for each PPC instruction. This is because of the functions calls that are performed every now and then in an unpredictable manner. The UISA simulator calculated its performance by counting the amount of instructions needed by each service routine together with its epilogue. This was possible because there was only one execution path in all epilogues and service routines. Everything was programmed inline, with no function calls.

When studying our simulator only an average value, for the amount of SPARC instructions needed to execute a PPC instruction, can be calculated. The performance is not a fixed level. If many programs are loaded in the simulator, and if they switch between each other often, the performance will be lower. The performance also depends on how the TLB is set up. The MMU will stop search at the first hit, because pages in the MMU are not allowed to overlap each other. Therefore pages that are defined in the end of the TLB will take longer time to find than those at the front. The cached results from the MMU are also vital for the overall performance. If the loaded programs are frequently switching between memory pages the performance will be lower.

6 Final thoughts

Great amount of studying has been required for us to understand this problem domain. High performance computing such as simulator programming is a complicated and challenging task to work with indeed. Adding complex features such as memory management has increased the problem domain even more.

The time it took to understand the problem domain, creating the simulator and documenting the implementation ended up a little more than 700 man-hours.

It has been an advantage that we worked together on this assignment especially when discussing potentials solutions. This meant that our advisors have not been tied up with discussions since we have been able to tackle with the most parts of the task ourselves. The implemented solutions are therefore at a high extent based on our own ideas.

The resulting simulator, capable of executing general PPC code extended with memory management related instructions, could be considered as a successful implementation. Unfortunately the evaluation of performance is left for further investigation in a future thesis work, due to the complex execution flow in the simulator and lack of time.

From the viewpoint of EIN the simulator is still not complete. The biggest weakness is the lack of support for JIT (Just In Time) decoding, which allows the simulator to modify and load new programs continuously during simulation. Another feature that EIN would like to have in the simulator is statistics collection from the cache and the TLB. Statistics is vital when developing the software supposed to be executed on a specific processor. Such statistics are cache-misses and TLB-misses, both of which stall the execution in the processor and decrease the overall performance. Before one can say if the simulator is ready to be inserted into an EIN product, one must first examine the special needs of a program that is supposed to be executed in the simulator. Only then one can know what the simulator must be able to execute.

References

[1] Datalogi - en inledande översikt, Hans Lunell, Studentlitteratur, ISBN 91-44-12193-4, 1992.

[2] Efficient PowerPC Simulation: A feasibility study, Patrik Seffel, Karlstad University, 1999.

[3] PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Motorola Inc., Document number MPCFPE32B/AD, 1997.

[4] Programming model differences of the IBM PPC 400 family and 600/700 family processors, IBM corp, 1999.

[5] PowerPC 405GP Embedded Processor User’s Manual, IBM corp., Document number GK10-3118-06, 1999.

[6] Efficient instruction cache simulation and execution profiling with a threaded-code simulator: Peter S. Magnusson – Swedish institute of computer science, 1997.

[7] PPC 405GP User's Manual, IBM corp., Document number GK10-3118-05, 2001.

[8] The Principles of Computer Hardware, Alan Clements, Oxford University Press, ISBN 0-19-856454-6, 2000.

Appendix A Abbreviations and clarifications of basic concepts

1. *AXE* – A telecommunication platform containing a central processor (CP) and several regional processors (RP). The CP distributes the workload to the RPs.
2. *Branch* – A jump from one address to one another, relative or absolute.
3. *CP* – Central processor in the AXE.
4. *EIN* – Ericsson Infotech.
5. *Embedded computer system* - A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system. IEEE Std 610.12-1990.
6. *Endianess* – Is a concept that handles the subject of byte ordering and bit naming. Two modes are present in the PPC-405GP architecture, big- and little-endianess.
7. *LoR* - List of Requirements.
8. *MM* – Memory Management.
9. *MMU* – Memory Management Unit.
10. *OEA* - Operating Environment Architecture.
11. *Page* – The smallest block of memory that is guaranteed to be in-line, i.e. the block is not fragmented in the physical memory.
12. *Paging* – A technique permitting the logical address-space of a process to be non-contiguous. The memory is broken into blocks of the same size called *pages*.
13. *PC* – A Program Counter points at the next instruction to be executed.
14. *PPC* – Power PC.
15. *Process* – A small part of a program running in a computer. A program can consist of several processes.
16. *RAM* - Random Access Memory.
17. *RP* – Regional Processor in the AXE.
18. *TLB* – Translation Lookaside Buffer.
19. *Tower of Hanoi* – An old puzzle-game that consists of a tower of a given height, initially stacked in decreasing size on one of three pegs. The objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never putting a larger disk onto a smaller.
20. *UISA* - User Instruction Set Architecture.
21. *VEA* - Virtual Environment Architecture.

Appendix B MMU differences between PPC-405GP, 750 and 860

Abstract

This is a standalone document, which describes and analyses the differences between PPC-405GP, PPC 750 and PPC 860. The aim with this document is to find the similarities and the differences between the processors. This will help to see if there is a possibility that a generic MMU simulator for all three processors can be created/implemented. The document does not study the whole processors, only their Memory Management Unit (MMU).

Background

This document is based on the fact that there will be an implementation of a MMU simulator for PPC-405GP. Therefore we only investigate the possibility to extend a MMU simulator for PPC-405GP to support all three designs. All processors uses 32 bit addressing

Family overview

PPC-405GP

The 400 family has specific features which optimise its use in embedded environments and maintains compatibility in the base User Instruction Set Architecture (UISA). However it defines separate Virtual and Operating Environment architectures (VEA & OEA) that have been enhanced for embedded applications. These enhancements have prompted changes in features such as memory management. (Programming model differences of the IBM PPC 400 family and 600/700 family processors, IBM corp, 1999).

PPC-750

The 700 family is compatible with the original PPC architecture. The aim is desktop applications and not embedded system as the PPC-405GP. Therefore the differences between 405 family's MMU and 700 family's MMU are significant.

PPC-860

This processor is designed to be a communication CPU, with built-in interfaces for Ethernet and more. It's not designed to simultaneously run several applications. This has an impact on the memory management design.

Detailed MMU feature comparison

PPC-405GP

- One level TLB look up design.
- User has no hardware logic support for MMU handling.
- One TLB for both instructions and data. 64 entries
- 40 bit virtual address.
- Variable page size (1KB, 4KB, 16 KB, 64KB, 256KB, 1MB, 4MB and 16MB) .
- MMU exceptions.
- Software TLB update.
- 2 Shadow TLB. One for data entries (4 entries) and one for instructions (8 entries).
- Support for real addressing mode.

PPC-750

- Implements Segments->Page look up and BAT look up.
- BAT. Block address translation.
- 52 bit virtual address.
- Fixed page size (4 KB).
- Fixed segment size (256MB).
- Variable block size (128KB - 256 MB).
- Hardware assisted TLB lockup.
- Separate TLB for instruction and data. Each has 128 entries.
- MMU exceptions.
- Hardware mechanism for TLB update.
- Hardware mechanism for updating 'referenced' and 'changed' bits.

PPC-860

- Two level TLB look up design. Pages and sub pages.
- Separate TLB for instructions and data. Each with 32 entries
- Multiple page sizes (4KB, 16KB, 512KB and 8 MB).
- Supports 16 virtual address spaces. 2^32 * 16 bits of total virtual memory.
- Supports 16 APG (Access protection group).
- MMU exceptions.
- Hardware mechanism for TLB update/rotation.
- Software updated Changed bit. Used as a write protection attribute in the hardware mechanism.

- Separate valid flags on sub pages when 4 KB page size is used with 1KB sub pages.
- Hardware mechanism for ensuring that multiple pages referencing to the same physical page.
- Different look up schemes for Read and Load/Store access.

Differences visualised in figures

PPC-405GP

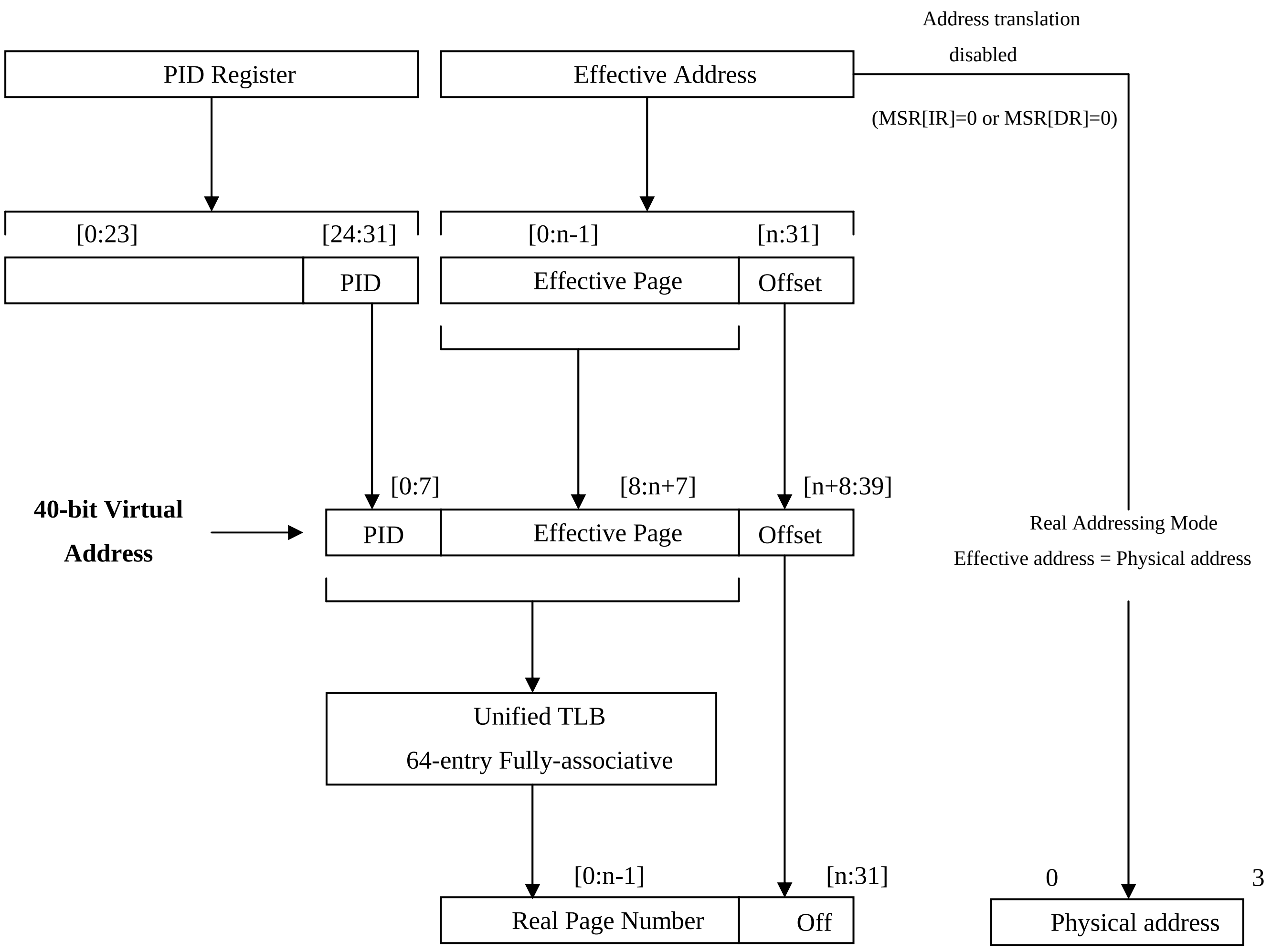


Figure B.1 MMU implementation in PPC-405GP

Figure B.1 shows how the MMU of the PPC-405GP works. It's a very straightforward design where the virtual address is compared with all entries in the TLB. If there is a match and no access violation etc, the MMU appends the offset to the Real Page Number, which is stored in the TLB.

PPC-750

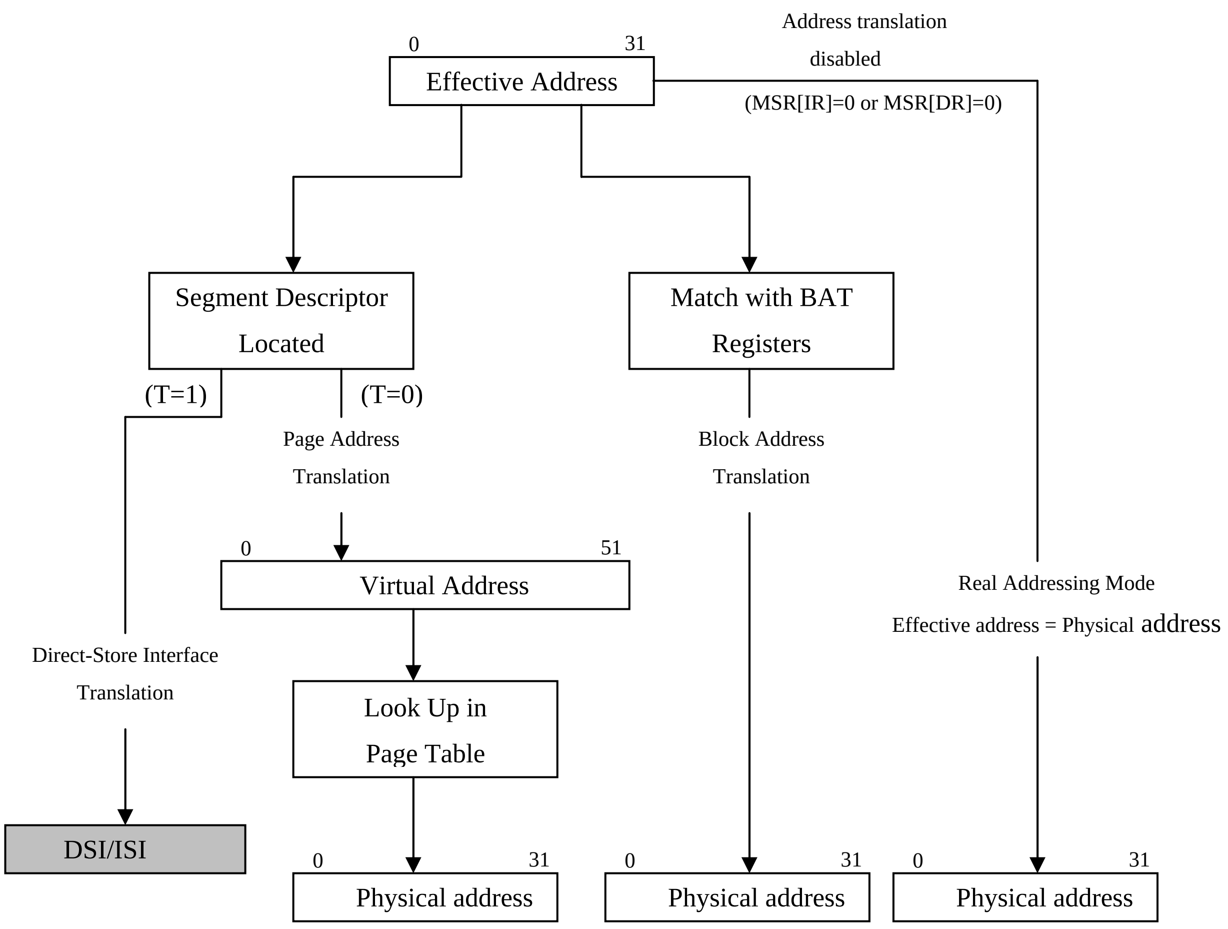
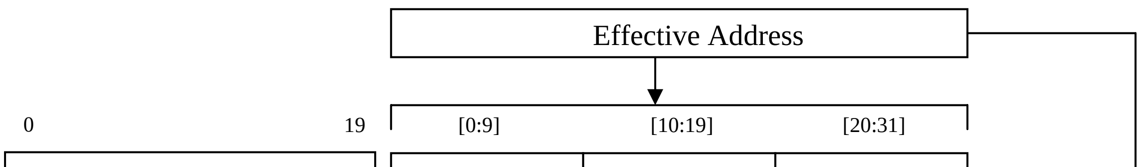


Figure B.2 MMU implementation in PPC-750

Figure B.2 shows the PPC-750 MMU design. Three different types of address translation are used. The one furthest to the right in the figure describes when translation is disabled (Direct Addressing Mode). Then there is the path trough the BAT register. The final translation path is through the segment register and page table. This is done when the BAT register can't translate a given address.

PPC-860

Figure B.3 shows the design for PPC-860. It shows a sequential translation process that works through two the levels present in the PPC-860 design. First it searches for a match in the level-1 table, which contains some protection attributes and the base pointer to level-2 table. The second table contains more protection attributes and of course the Real Page Number.



Real Addressing Mode

Effective address = Physical address

Figure B.3 MMU implementation in PPC-860

Conclusion

The different processors have different registers dedicated to MMU use, which adds complexity. Certain registers exist in one processor, but don’t necessarily exist in all other processors. The three MMUs also include more or less hardware supported features which otherwise is done in software. The semantics of many instructions, aimed for the MMU, differs but has the same name.

The significant differences in the MMU of the three processors make it hard to find a generic solution. Such a solution, which includes support for all processors, will not likely be easy to implement or to optimise for speed, which is important for overall simulator performance.

Implementing a solution in C++ and using the object oriented modelling, makes it possible to implement an interface common for all three processors. A common interface class, which is inherited by all three different solutions, gives an efficient and clear model and the possibility to extend the particular subclass with its specific attributes belonging to each processor.

Appendix C Register summary

This appendix summarises all registers implemented in the simulator developed in this thesis.

PPC-405GP Programming model-registers

The following part lists the registers implemented in this thesis work. The Figure C.1 below shows the complete register-model of PPC-405GP. The registers marked with dark grey colour are not implemented in this study. Most of the registers still not implemented are support debug and timer features, and there are also registers for handling the real addressing mode. Registers implemented in the UISA simulator, developed by Seffel, are marked with light grey colour and the unmarked registers are implemented in this study. As one can see the most of the registers implemented in this extended UISA simulator are supervisor state registers, and this is because the MM is controlled at the supervisor-level.

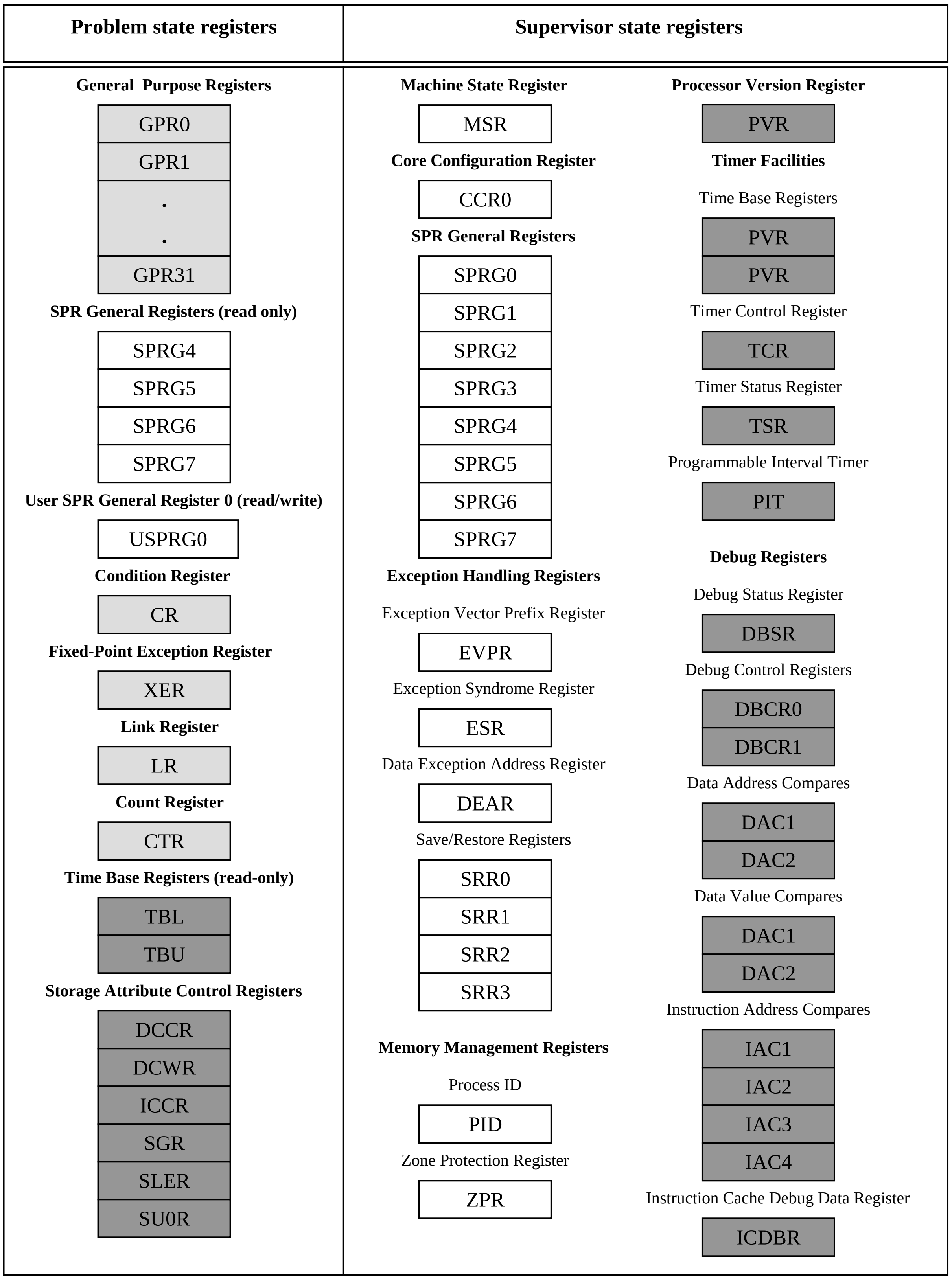


Figure C.1 Register model in PPC-405GP

Zone Protection Register (ZPR)

Each entry in the TLB contains a 4-bit zone-select field (ZSEL). A Zone is defined as being an arbitrary identifier for grouping TLB entries (memory pages) for the purpose of protection. As many as 16 different zones can be defined and each zone can have any number of member pages.

Each zone is associated with a 2-bit filed in the ZPR. The values of the field define how protection is applied to all pages that are member of that zone. Changing the value of the ZPR field changes the protection attributes of all pages in that field. Without the ZPR the change would require finding, reading, changing and rewriting the TLB entry for each page in the zone. The ZPR provides a much faster means of changing the protection groups of memory pages. Each zone determines the TLB page access control for all pages in its zone according to the table below.

ZPR

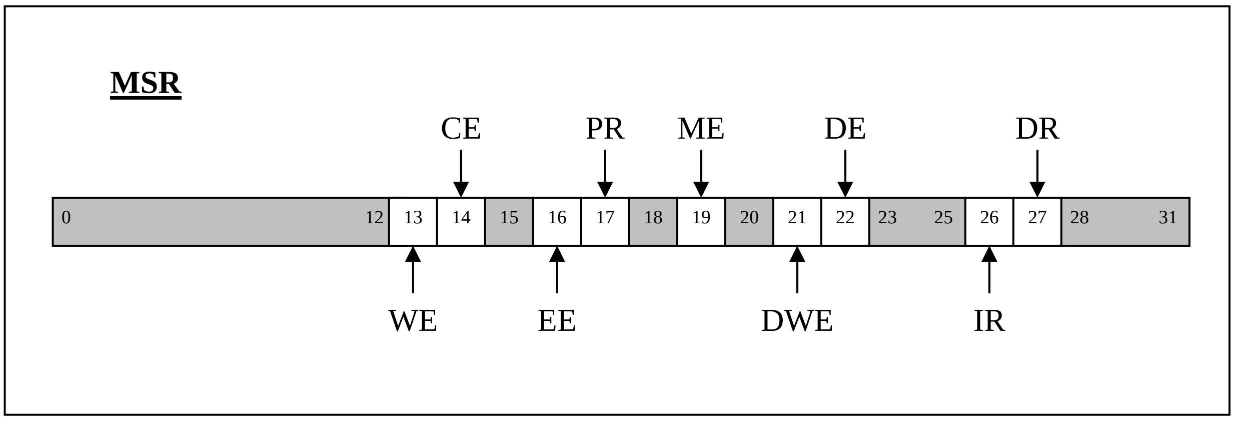
Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15

Bits, n:n+1	**Zone, Zn**	**Problem state (MSR[PR]=1)**	**Supervisor state (MSR[PR]=0)**
		00. No access	00. Access controlled by applicable TLB entry[EX,WR]
		01. Access controlled by applicable TLB entry[EX,WR]	01. Access controlled by applicable TLB entry[EX,WR]
		10. Access controlled by applicable TLB entry[EX,WR]	10. Access controlled by applicable TLB entry[EX,WR]
		11. Accessed as if execute and write permissions (TLB_entry [EX,WR]) were granted	11. Accessed as if execute and write permissions (TLB entry [EX,WR]) were granted

Setting ZPR[Zn] = 00 for a ZPR field is the only way to deny read access to a page defined by an otherwise valid TLB entry because TLB_entry[EX] and TLB_entry[WR] does not support read protection. For a given ZPR field value, a program in supervisor state always has equal or greater access for a valid TLB entry [5].

Machine State Register (MSR)

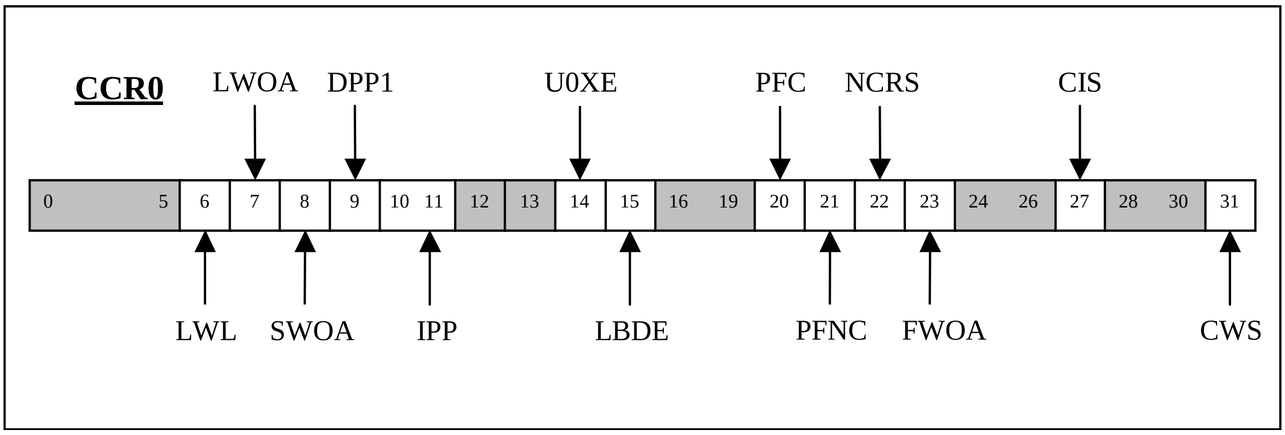
Fields in the MSR control the use of the MMU for address translation. The architecture uses MSR[PR] to control the execution mode. When MSR[PR] = 1, the processor is in user mode (problem state); when MSR[PR] = 0, the processor is in privileged mode (supervisor state).



Bit	**Name**	**Description**
13	WE	Wait State Enable. If MSR[WE] = 1, the processor remains in the wait state until the WE bit is cleared.
14	CE	Critical Interrupt Enable. Controls if critical interrupts are enabled.
16	EE	External Interrupt Enable. Controls if asynchronous interrupts (external to the processor core) are enabled.
17	PR	Problem State. Controls if the processor are in Problem- or Supervisor state.
19	ME	Machine Check Enable. Controls if machine check interrupts are enabled.
21	DWE	Debug Wait Enable. Controls if debug wait mode is enabled.
22	DE	Debug Interrupts Enable. Controls if debug interrupts are enabled.
26	IR	Instruction Relocate. Controls if instruction address translation is enabled.
27	DR	Data Relocate. Controls if data address translation are enabled.

Core Configuration Register 0 (CCR0)

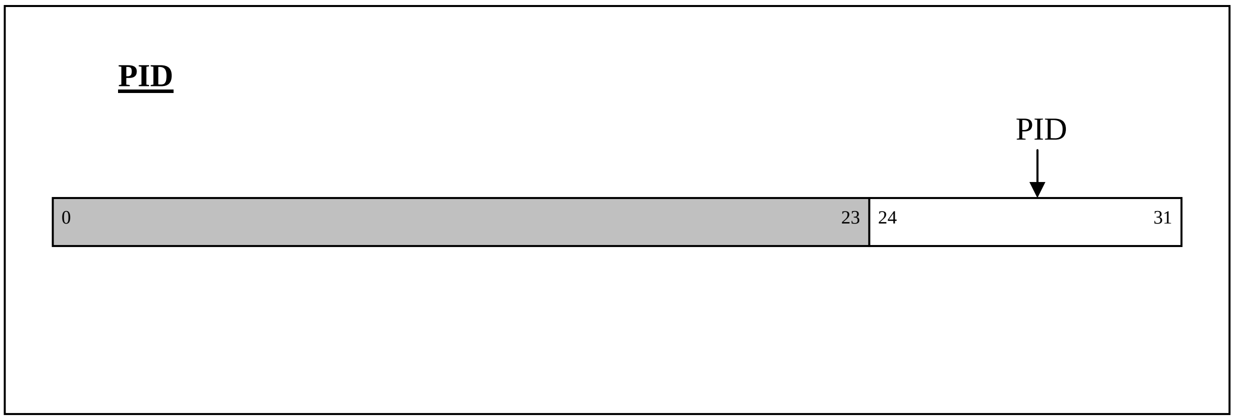
CCR0 controls the behaviour of the `icread` and the `dcread` instructions. The U0XE-bit is used in the MMU together with the attribute U0 (user-defined attribute).



Bit	**Name**	**Description**
6	LWL	Load Word as Line
7	LWOA	Load Without Allocate
8	SWOA	Store Without Allocate
9	DPP1	DCU PLB Priority Bit 1
10-11	IPP	ICU PLB Priority Bits 0:1
14	U0XE	Enable U0 Exception
15	LBDE	Load Debug Enable
20	PFC	ICU Prefetching for Cachable Regions
21	PFNC	ICU Prefetching for Non-Cachable Regions
22	NCRS	Non-cachable ICU request size
23	FWOA	Fetch Without Allocate
27	CIS	Cache Information Select
31	CWS	Cache Way Select

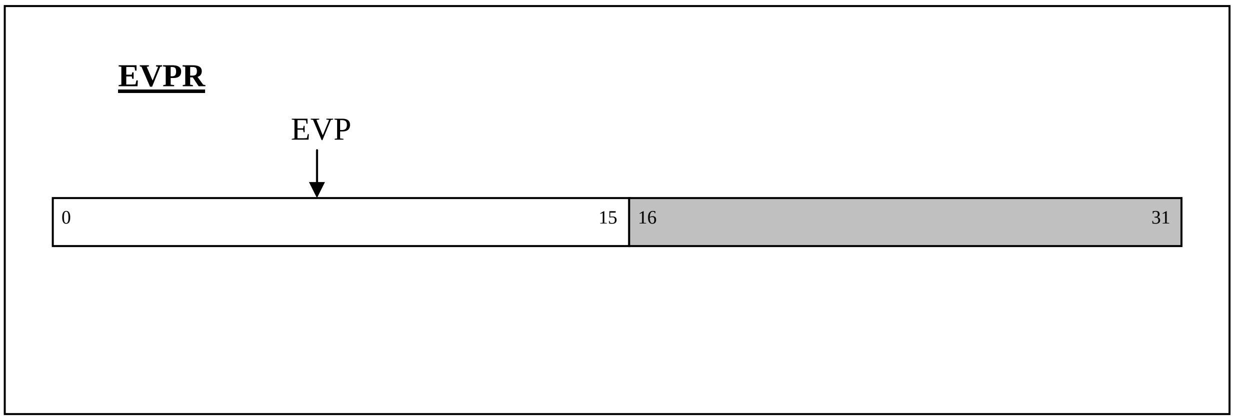
Process ID (PID)

The PID identifies one of 255 unique software entities, usually used as a process or thread ID. The PID is used in address translation where the 8-bit process ID (PID) and the EA is combined to create a 40-bit virtual address.



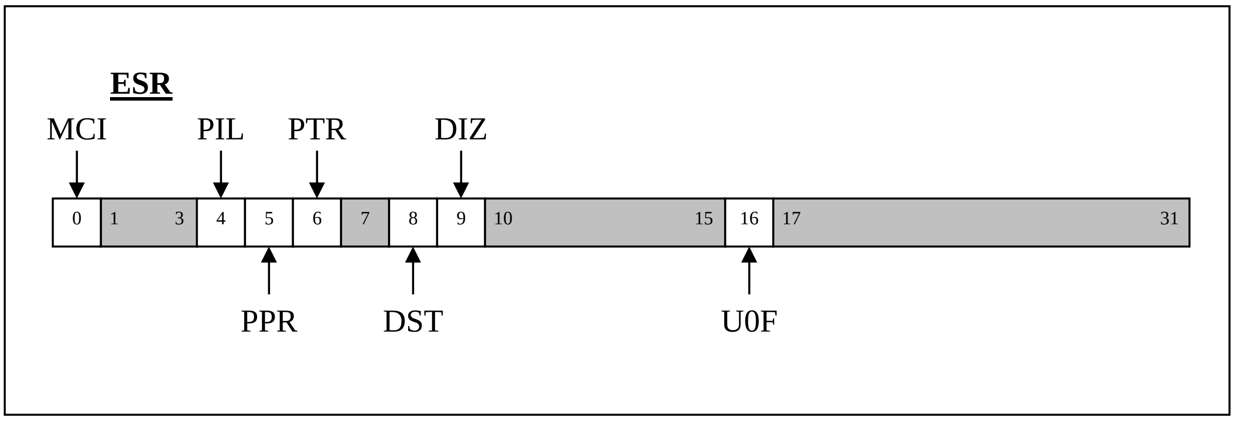
Exception Vector Prefix Register (EVPR)

The EVPR is a 32-bit register whose high-order 16 bits contain the prefix for the address of an interrupt handling routine. The 16-bit interrupt vector offsets are concatenated to the right of the high-order 16 bits of the EVPR to form the 32-bit address of an interrupt handling routine. The contents of the EVPR can be written to a GPR using the `mfspr` instruction. The contents of a GPR can be written to EVPR using the `mtspr` instruction.



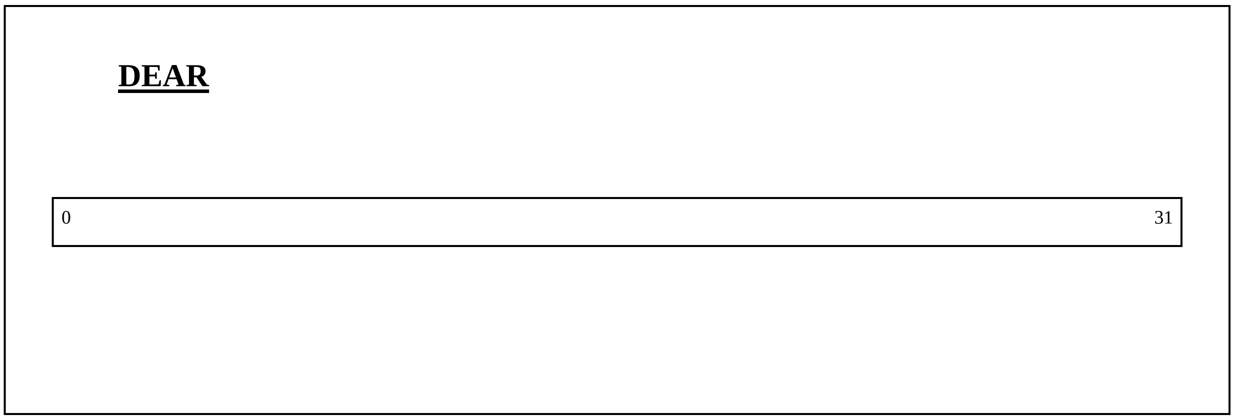
Exception Syndrome Register (ESR)

The ESR is a 32-bit register whose bits help to specify the exact cause of various synchronous interrupts. These interrupts include instruction and data side machine checks, data storage interrupts, and program interrupts, instruction storage interrupts, and data TLB miss interrupts. The contents of the ESR can be written to a GPR using the `mfspr` instruction. The contents of a GPR can be written to the ESR using the `mtspr` instruction.



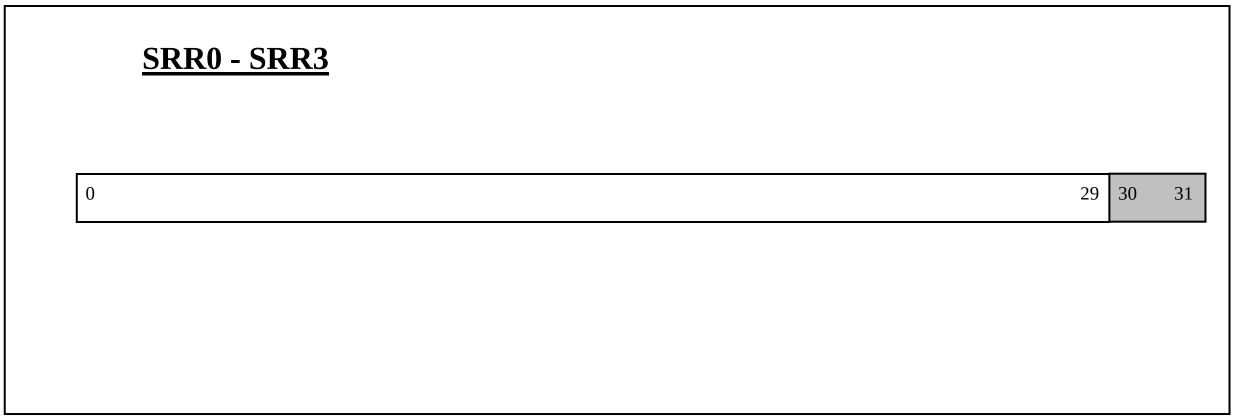
Data Exception Address Register (DEAR)

The DEAR is a 32-bit register that contains the address of the access for which one of the following synchronous precise errors occurred: alignment error, data TLB miss, or data storage interrupts. The contents of the DEAR can be written to a GPR using the `mfspr` instruction. The contents of a GPR can be written to the DEAR using the `mtspr` instruction.



Save/Restore Registers (SRR0-SRR3)

SRR0 and SRR1 are 32-bit registers that hold the interrupted machine context when a noncritical interrupt is processed. On interrupt, SRR0 is set to the current or next instruction address and the contents of the MSR are written to SRR1. When an `rfi` instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR0 and SRR1, respectively. SRR2 and SRR3 serve the same purpose except that they hold the interrupted machine context when a critical interrupt is processed. On interrupt, SRR2 is set to the current or next instruction address and the contents of the MSR are written to SRR3. When an `rfci` instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR2 and SRR3, respectively.



User SPR General Register 0 (USPRG0) and SPR General Registers (SPRG0-SPRG7)

USPRG0 and SPRG0–SPRG7 are provided for general-purpose software use. For example, these registers are used as temporary storage locations. For example, an interrupt handler might save the contents of a GPR to an SPRG, and later restore the GPR from it. This is faster than a save/restore to a memory location. These registers are written using mtspr and read using mfspr. Access to USPRG0 is non-privileged for both read and write. SPRG0–SPRG7 provide temporary storage locations. For example, an interrupt handler might save the contents of a GPR to an SPRG, and later restore the GPR from it. This is faster than performing a save/restore to memory. These registers are written by `mtspr` and read by `mfspr`.

Appendix D Instruction summary

To support embedded real-time applications, the instruction sets of the PPC-405GP and other IBM PPC-400 series embedded controllers, implement the IBM PPC Embedded Environment, which is not part of the PPC Architecture defined in the PPC Architecture: A Specification for a New Family of RISC Processors. Programs using these instructions are not portable to PPC implementations that do not implement the IBM PPC Embedded Environment. The PPC-405GP implements a number of implementation-specific instructions that are not part of the PPC Architecture or the IBM PPC Embedded Environment. This appendix has been heavily based on [5].

TLB Management Instructions

The TLB management instructions read and write entries of the TLB array in the MMU, search the TLB array for an entry that will translate a given address, and be used to invalidate all TLB entries. There is also an instruction for synchronising TLB updates with other processors, but because the PPC-405GP is for use in single-processor environments, this instruction performs no operation. The syntax “[.]” indicates that the instruction has a “record” form that updates CR[CR0].

TLB Invalidate Instruction (tlbia)

With this instruction all of the entries in the TLB are invalidated and become unavailable for translation by clearing the valid (V) bit of each TLB entry. The rest of the fields in the TLB entries are unmodified.

TLB Read Instruction (tlbre)

TLB entries can be accessed for reading by the `tlbre` instruction. The contents of the selected TLB entry are placed into a specific register. The parameters are used as index into the TLB. If this index specifies a TLB entry that does not exist, the results are undefined.

TLB Search Instructions (tlbsx/tlbsx.)

Software can search for specific TLB entries using the `tlbsx` instruction. `tlbsx` locates entries in the TLB, to find the TLB entry associated with an interrupt, or to locate candidate entries to cast out. The logical address is the value to be matched. . If the TLB entry is found, its index is placed in a register can then serve as the source register for a `tlbre` or `tlbwe`. If no match is found, the content of the result register is undefined.

`tlbsx.` sets a bit in the Condition Register (CR). The value of this bit depends on whether an entry is found. It is set to one if an entry is found or zero if no entry is found. The intention is that the bit in the CR can be tested after a `tlbsx`. instruction if there is a possibility that the search may fail.

TLB Sync Instruction (tlbsync)

`tlbsync` guarantees that all TLB operations have completed for all processors in a multi-processor system. PPC405GP provides no multiprocessor support, so this instruction performs no function. The instruction is included to facilitate code portability.

TLB Write Instruction (tlbwe)

TLB entries can be accessed for writing by the `tlbwe` instruction. The contents of the selected TLB entry are replaced with the contents of a specific register. Its parameters are used as an index into the TLB. If this index specifies a TLB entry that does not exist, the results are undefined.

Processor Management Instructions

These instructions move data between the GPRs and SPRs. There are a lot of other processor management instructions but only these two are implemented in this study.

Move To Special Purpose Register (mtspr)

The contents of a GPR can be written to a SPR using the `mtspr` instruction.

Move From Special Purpose Register (mfspr)

The contents of a SPR can be written to a GPR using the `mfspr` instruction.

Exception and interrupt instructions

These instructions handle exceptions and interrupts. `rfi` and `rfci` are return from interrupt handlers, `mfmsr` and `mtmsr` read and write data between the MSR and a GPR to enable and disable interrupts

Return From Interrupt (rfi)

The PC is restored with the contents of SRR0 and the MSR is restored with the contents of SRR1. Instruction execution returns to the address contained in the PC.

Return From Critical Interrupt (rfci)

The PC is restored with the contents of SRR2 and the MSR is restored with the contents of SRR3. Instruction execution returns to the address contained in the PC.

Move To Machine State Register (mtmsr)

The contents of a GPR can be written to the MSR using the `mtmsr` instruction.

Move From Machine State Register (mfmsr)

The contents of the MSR can be written to a GPR using the `mfmsr` instruction.

Appendix E Interrupts

A processor relies on interrupt handling software to implement paged virtual memory, and to enforce protection of specified memory pages. When an interrupt occurs, the processor clears MSR[IR, DR]. Therefore, at the start of all interrupt handlers, the processor operates in real mode for instruction accesses and data accesses. When address translation is disabled for an instruction fetch or load/store, the EA is equal to the real address and is passed directly to the memory subsystem (including cache units). Such untranslated addresses bypass all memory protection checks that would otherwise be performed by the MMU. When translation is enabled, MMU accesses can result in the following interrupts:

• Data storage interrupt

• Instruction storage interrupt

• Data TLB miss interrupt

• Instruction TLB miss interrupt

Data Storage Interrupt

A data storage interrupt is generated when data address translation is active, and the desired access to the EA is not permitted for one of the following reasons:

• In the problem state

- Load/store, `dcbz`, `icbi` or `dcbf` with an EA whose zone field is set to no access
- (ZPR[Zn] = 00).
- Stores, or `dcbz`, to an EA having write access disabled (TLB_entry[WR] = 0) and whose zone field is set to full access (ZPR[Zn] $\neq$ 11 Accessed as if execute and write permissions (TLB_entry[EX, WR]) are granted).

• In supervisor state

- Data store, `dcbi`, `dcbz`, or `dccci` to an EA having TLB_entry[WR] = 0 and ZPR[Zn] other than 11 or 10.

Instruction Storage Interrupt

An instruction storage interrupt is generated when instruction address translation is active and the processor attempts to execute an instruction at an EA for which fetch access is not permitted, for any of the following reasons.

- In the problem state:
  - Instruction fetch from an EA with ZPR[Zn] = 00.
  - Instruction fetch from an EA having TLB_entry[EX] = 0 and ZPR[Zn] ≠ 11.
  - Instruction fetch from an EA having TLB_entry[G] = 1.
- In the supervisor state:
  - Instruction fetch from an EA having TLB_entry[EX] = 0 and ZPR[Zn] other than 11 or 10.
  - Instruction fetch from an EA having TLB_entry[G] = 1.

Data TLB miss interrupt

A data TLB miss interrupt is generated if data address translation is enabled and a valid TLB entry matching the EA and PID is not present. The interrupt applies to data access instructions and cache operations.

Instruction TLB miss interrupt

The instruction TLB miss interrupt is generated if instruction address translation is enabled and execution is attempted for an instruction for which a valid TLB entry matching the EA and PID for the instruction fetch is not present.