UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS - GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Renan Augusto Starke

DESIGN AND EVALUATION OF A VLIW PROCESSOR FOR REAL-TIME SYSTEMS

Florianópolis
2016

Renan Augusto Starke

DESIGN AND EVALUATION OF A VLIW PROCESSOR FOR REAL-TIME SYSTEMS

A Thesis submitted to the Automation and Systems Engineering Postgraduate Program in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Automation and Systems Engineering.
Supervisor: Rômulo Silva de Oliveira

Florianópolis
2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Starke, Renan Augusto
Design and Evaluation of a VLIW Processor for Real-Time Systems / Renan Augusto Starke ; orientador, Rômulo Silva de Oliveira - Florianópolis, SC, 2016.
204 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Real time systems. 3. Very Long Instruction Word (VLIW) processor. 4. Worst-Case Execution Time (WCET) Analysis. I. Oliveira, Rômulo Silva de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

DESIGN AND EVALUATION OF A VLIW PROCESSOR FOR REAL-TIME SYSTEMS

Renan Augusto Starke

This Thesis is hereby approved and recommended for acceptance in partial fulfillment of the requirements for the degree of "Doctor of philosophy in Automation and Systems Engineering."

June 6th, 2016.

Prof. Rômulo Silva de Oliveira, Dr.
Supervisor

Prof. Daniel Coutinho, Dr.
Coordinator of the Automation and Systems Engineering Postgraduate Program

Examining Committee:

Prof. Rômulo Silva de Oliveira, Dr. – UFSC
Chair

Prof. Anderson Luiz Fernandes Perez, Dr. – UFSC

Prof. Joni da Silva Fraga, Dr. – UFSC

Explanations exist: they have existed for all times, for there is always an easy solution to every problem – neat, plausible and wrong.
– H.L. Mencken, “The Divine Afflatus”, in the New York Evening Mail, November 16, 1917

ACKNOWLEDGEMENTS

I thank to professor Rômulo Silva de Oliveira for his dedication, encouragement, patience and the great suggestions when the way seemed unclear.

I thank my family for their trust and support during the course. In particular I thank my parents allowing access to high quality education. A special thank to my future wife Allessa Mariá Maiochi who waited and supported me for these years.

I also thank to laboratory colleagues for friendship, help in corrections, sharing and improvement of ideas. A special thank to my friend Andreu Carminati who helped to make this work possible with the compiler back-end implementation, ideas and tips.

Finally thanks to CAPES for the financial support that enabled this work.

ABSTRACT

Nowadays, many real-time applications are very complex and as the complexity and the requirements of those applications become more demanding, more hardware processing capacity is necessary. The correct functioning of real-time systems depends not only on the logically correct response, but also on the time when it is produced. General purpose processor design fails to deliver analyzability due to their non-deterministic behavior caused by the use of cache memories, dynamic branch prediction, speculative execution and out-of-order pipelines. In this thesis, we design and evaluate the performance of VLIW (Very Long Instruction Word) architectures for real-time systems with an in-order pipeline considering WCET (Worst-case Execution Time) performance. Techniques on obtaining the WCET of VLIW machines are also considered and we make a quantification on how important are hardware techniques such as static branch prediction, predication, pipeline speed of complex operations such as memory access and multiplication for high-performance real-time systems. The memory hierarchy is out of scope of this thesis and we used a classic deterministic structure formed by a direct mapped instruction cache and a data scratchpad memory. A VLIW prototype was implemented in VHDL from scratch considering the HP VLIW ST231 ISA. We also show some compiler insights and we use a representative subset of the Mälardalen's WCET benchmarks for validation and performance quantification. Supporting our objective to investigate and evaluate hardware features which reconcile determinism and performance, we made the following contributions: design space investigation and evaluation regarding VLIW processors, complete WCET analysis for the proposed design, complete VHDL design and timing characterization, detailed branch architecture, low-overhead full-predication system for VLIW processors.

Keywords: Real time systems, Very Long Instruction Word (VLIW) processor, Worst-Case Execution Time (WCET) Analysis.

RESUMO EXPANDIDO

CONSIDERAÇÕES SOBRE PROJETO DE PROCESSADORES VLIW PARA SISTEMAS DE TEMPO REAL

Palavras-chave: Sistemas de tempo real, processadores VLIW (*Very-Long Instruction Word*), análise de pior tempo de computação (WCET – *Worst-case Execution Time*)

Introdução

Atualmente, aplicações de tempo estão tornando-se cada vez mais complexas e, conforme os requisitos destes sistemas aumentam, maior é a demanda por capacidade de processamento. Contudo, o correto funcionamento destas aplicações não está em função somente da correta resposta lógica, mas também no tempo que ela é produzida.

Nos últimos anos, houve uma quantidade significativa de pesquisas voltadas a arquiteturas de processadores temporalmente previsíveis com o intuito de utilizá-los em sistemas de tempo real. Como o principal objetivo de projeto de processadores de propósito geral tem-se mantido na melhora do desempenho de caso médio, a utilização destes em sistemas de tempo real tornou-se consideravelmente complexa devido à necessidade de análises para obtenção de parâmetros temporais.

Sistemas de tempo real são geralmente modelados por um conjunto de tarefas onde cada uma possui seu pior tempo de execução (WCET – *Worst-case Execution Time*), período e prazo (*deadline*) e estes parâmetros são utilizados em testes de escalonabilidade formais. Em conjunto com um algoritmo de escalonamento de tarefas, é formado um problema de escalonabilidade onde o objetivo é verificar se todas as tarefas cumprem seus *deadlines*: o tempo de resposta de uma tarefa deve sempre ser menor ou igual ao seu respectivo *deadline*. Obter o pior tempo de computação tem se mostrado complexo e dependente de parâmetros relacionados ao *hardware* como arquitetura do processador e da memória. Obter o WCET eficientemente e com precisão é necessário tanto para o mais simples quanto ao mais complexo teste de escalonabilidade.

As principais abordagens para obter o WCET são medições, análises estáticas e análises híbridas. A prática mais comum na indústria é o uso de medições executando o sistema no *hardware* alvo. Nas análises estáticas, não há execução mas a estimação do WCET é realizada utilizando-se de um modelo matemático constituído pelo o binário da tarefa e características da plataforma alvo. No caso das análises híbridas, são combinadas análises estáticas com medições. Independentemente da abordagem, ferramentas de análise WCET fornecem uma estimativa do tempo de execução de um código que deverá ser igual ou maior que WCET real. Aplicações complexas são difíceis de analisar por qualquer abordagem (GUSTAFSSON, 2008). Análises estáticas podem gerar problemas complexos não escaláveis geralmente relacionados ao *hardawre*, enquanto medições necessitam de casos de uso que não necessariamente produzem o WCET de uma aplicação. Para amenizar o problema relacionado a estimação do WCET, tem-se mostrado interesse em arquiteturas projetadas especificamente para sistemas de tempo real, como os trabalhos de (SCHOEBERL et al., 2011), (LIU et al., 2012) e (SCHOEBERL et al., 2015)

Nesta tese, investiga-se uma arquitetura de processador VLIW – *Very-Long Instruction Word* especificamente projetada para sistemas de tempo real considerando sua análise do pior tempo de computação (WCET – *Worst-case Execution Time*). Técnicas para obtenção do WCET para máquinas VLIW são consideradas e quantifica-se a importância de técnicas de *hardware* como previsor de fluxo estático, predicação, bem como velocidade do processador para instruções complexas como acesso a memória e multiplicação. A arquitetura de memória não faz parte do escopo deste trabalho e para tal utilizamos uma estrutura determinista formada por uma memória *cache* com mapeamento direto para instruções e uma memória de rascunho (*scratchpad*) para dados. Nós também consideramos a implementação em VHDL do protótipo para inferir suas características temporais mantendo compatibilidade com o conjunto de instruções (ISA) HP VLIW ST231. Em termos de avaliação, foi utilizado um conjunto representativo de código exemplos da Universidade de Mälardalen que é amplamente utilizado em avaliações de sistemas de tempo-real.

Objetivos

O objetivo desta tese é investigar características de arquitetura de processadores que levam a um projeto determinista mas também que consideram o desempenho de pior caso (redução do WCET). A tese a ser demonstrada é que é possível utilizar elementos de *hardware* que aumentam o desempenho mas que são previsíveis o suficiente para garantir uma análise estática eficiente e precisa. Para demonstrar previsibilidade, é interessante demonstrar as técnicas de análise envolvidas na obtenção de WCET de tarefas. Portanto, a construção de uma ferramenta de WCET assim como os aspectos envolvidos na modelagem do *hardware* também são assuntos cobertos neste trabalho.

Entre os elementos arquiteturais considerados, têm-se:

- *Pipeline*:

  Pipeline é uma técnica que permite que operações complexas sejam organizadas em outras mais simples com o objetivo de aumentar desempenho. No caso de processadores deterministas, *pipelines* são necessários mas as instruções devem ser executadas em ordem. *Pipelines* com execução fora de ordem permitem um alto desempenho de caso médio mas prejudicam a análise de WCET devido a anomalias temporais (LUNDQVIST; STENSTROM, 1999).

- Paralelismo entre instruções:

  Nos processadores modernos, o termo superescalar é utilizado quando mais de uma instrução é executada em cada estágio de *pipeline*. Este projeto supera a limitação de desempenho (*throughput*) de apenas uma instrução executada por ciclo de máquina. No caso de processadores para sistemas de tempo-real, a execução de múltiplas instruções por estágio de *pipeline* também pode ser utilizado utilizando um projeto *Very Long Instruction Word* (VLIW) (FISHER; FARABOSHI; YOUNG, 2005). Máquinas VLIW são mais adequadas para sistemas de tempo real pois o escalonamento de instruções é determinado pelo compilador e não em tempo de execução. Isso simplifica a análise estática, pois escalonamento dinâmico de instruções não precisa ser modelado.

- Primeiro nível do sub-sistema de memória:

  O sub-sistema de memória é um assunto relevante em vários trabalhos (REINEKE et al., 2011). Há diversas abordagens e algumas delas requerem modificações complexas no compilador ou no *hardware* (SCHOEBERL et al., 2011). Neste trabalho, nós tratamos da previsibilidade do sub-sistema de memória utilizando uma memória cache de instruções com mapeamento direto bem como uma memória de rascunho (*scratchpad*) para dados. Memórias de rascunho são similares às caches mas seu conteúdo é gerenciado explicitamente pelo *software*.

- Previsão de fluxo:

  Modificações no controle de fluxo de um programa são realizadas por instruções especiais chamadas *branches*. Elas são utilizadas para estruturas com condicionamentos (*if*), laços (*for* e *while*) e geralmente degradam o desempenho do *pipeline* devido a ciclos de paradas (*stalls*). Uma maneira de reduzir esta limitação é o uso de previsores de fluxo. Há os previsores dinâmicos e os estáticos. Os dinâmicos prejudicam a previsibilidade enquanto os estáticos possuem características interessantes para sistemas de tempo real (BURGUIERE; ROCHANGE; SAINRAT, 2005). Nós consideraremos os previsores estáticos, demonstrando sua importância em termos de desempenho bem como a metodologia para considerá-los na análise WCET.

- Predicação:

  Predicação é uma técnica na qual instruções são condicionalmente executadas baseando-se em um registrador *Booleano*. É diferente dos *branches* pois não há qualquer modificação no fluxo do programa para ignorar instruções. Há dois tipos de predicação: completa e parcial. Predicação completa permite que instruções sejam executadas ou ignoradas diretamente conforme o valor de um registrador *Booleano* (esta predicação é comumente empregada nos processadores ARM). No caso da predicação parcial, instruções não são ignoradas mas dois valores podem ser selecionados utilizando um tipo especial de instrução (*select*). Predicação é uma técnica importante para reduzir caminhos em um programa induzindo o paradigma da programação de caminho único (*Single-path programming paradigm*) (PUSCHNER, 2005). Nós suportamos ambos os tipos de predicação e a versão completa é reestruturada para reduzir o impacto no *hardware*.

- Instruções aritméticas complexas:

  Há várias instruções aritméticas complexas como divisão e multiplicação que diminuem o desempenho do processador. Frequentemente estas instruções são suportadas apenas por *software*, principalmente a divisão. Neste trabalho, tanto divisão quanto multiplicação por *hardware* são implementadas para que tenham previsibilidade independentemente de seus parâmetros de entrada.

Contribuições

Dentre as contribuições deste trabalho, nós podemos citar: investigação sobre o espaço de projeto, avaliação de desempenho através de conjunto representativo de códigos exemplos, detalhamento completo da análise estática, implementação VHDL, caracterização temporal de cada componente de *hardware*, detalhamento da arquitetura de controle de fluxo e um sistema de predicação completo de baixo impacto para processadores VLIW.

Realizou-se uma avaliação extensiva das técnicas apontadas acima considerando os benefícios em termos de desempenho de pior caso. Notou-se que uma avaliação tão ampla nunca havia sido considerada nos trabalhos relacionados pois nestes são focados objetivos específicos como subsistema de memória, *multi-threading* ou *multi-core*. Nossa avaliação pode guiar novas linhas de pesquisas relacionadas com sistemas de tempo real.

Foram também abordadas todas as análises necessárias para estimar o WCET de programas compilados para o processador considerado neste trabalho, incluindo *cache*, modelagem do *pipeline* e busca do pior caminho do programa. Uma descrição completa de uma análise WCET para processadores VLIW também não foi abordada nos trabalhos relacionados.

Todos os detalhes da arquitetura de controle de fluxo são descritos, incluindo a metodologia para modelagem durante a análise WCET. Também ampliamos o ISA para suportar previsão de desvio estático, bem como os benefícios de usar ou não esta tecnologia em sistemas de tempo real.

Quanto ao sistema de predicação completo, foi proposto um sistema que adiciona baixa sobrecarga nos caminhos de dados de *hardware* e na sua lógica de atalhos (*forwarding logic*). O sistema proposto diminui o uso de instruções de controle de fluxo, bem como permite a utilizaçao de técnicas de desenrolamento de laços (*loop unrolling*). No entanto, a predicação sozinha não é suficiente para aumentar o desempenho e previsibilidade e seu uso pode aumentar o tempo de execução (WCET). Devido a isso, propõe-se o uso de uma abordagem híbrida com suporte de *hardware* para predicação e previsão estática de fluxo. Isto leva a uma redução significativa do pior tempo de computação e permite otimizações durante a compilação que pode selecionar a técnica apropriada para cada estrutura.

Conclusão

Neste trabalho considerou-se elementos arquiteturais de processadores que beneficiam a análise estática mas também contribuem para o aumento do desempenho, principalmente para o pior caso de execução.

Como sistemas modernos impõem maiores requisitos funcionais, processadores de maior desempenho são necessários. Portanto, é necessário analisar os pontos fortes das técnicas de hardware usadas em processadores modernos, por exemplo paralelismo temporal e espacial na execução de instruções, previsão de desvios e predição, e para adaptá-los para aplicações em tempo real. Algumas destas técnicas exigem modificações, devido à alta complexidade do hardware, enquanto outros precisam de um comportamento temporal bem definido. Para atingir esses objetivos, foram propostas novas abordagens, a fim de melhorar a eficiência e a escalabilidade da análise temporal, especialmente a análise de tempo de execução do pior caso. Mostrou-se que o projeto de processadores deve aumentar o nível de importância do determinismo.

Foi possível verificar através dos testes de desempenho que as técnicas abordadas aumentaram o desempenho em todos os programas considerados. Além disso, mostrou-se detalhadamente as técnicas necessárias para realizar análise estática obtendo assim o WCET para todos programas testados.

A implementação VHDL do processador mostrou-se desafiadora mas contribuiu significativamente para a caracterização temporal de cada elemento de *hardware*.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic Logic Unit
BCET	Best-Case Execution Time
CISC	Complex Instruction Set Computing
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
ISA	Instruction Set Architecture
LLVM	Low Level Virtual Machine compiler infrastructure
LVT	Live Value Table
NOP	No Operation
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
SRAM	Static Random Access Memory
VEX	VLIW Example
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

LIST OF SYMBOLS

CFG	Control-Flow Graph
d_{j_i}	Edge from node j to i
δ	Transition-flow modeling factor
E	Edge set of a graph
$EMB_i(c)$	Effective Memory Block set of a basic block i and cache line c
G	Graph
n_c	Total number of cache misses inside a basic block
n_m	Total number of high latency memory access inside a basic block
$RMB_i(c)$	Reaching Memory Block set of a basic block i and cache line c
t_{bb}	Basic-block time
t_c	Cache-miss time
t_{i_j}	Time of basic block i when executing from predecessor j
t_m	High-latency memory access time
t_p	Pipeline modeling time
t_i	Time of basic block i
V	Vertex set of a graph
v	Vertex of a graph
x_i	Total number of times a basic block executes

1 INTRODUCTION

Real-time computer systems are defined as those subject to temporal requirements. They are relevant for diverse applications such as aviation, spacecrafts, automotive electronics and diverse industrial plants. Future real-time applications will require processors with higher performance and they must satisfy strict timing constraints for their correct functioning.

General-purpose processors usually are designed to perform the common case fast and the uncommon case correct (SCHOEBERL, 2009b). This design philosophy leads to machines optimized for average-case performance, which is not necessarily suitable for real-time systems. In order to verify strict timing constraints, we need to obtain the computation time or Worst-Case Execution Time (WCET) of every task of the systems. It has a considerable complexity and it depends on hardware features such as processor and memory architecture.

The Worst-case execution time problem, well described in Wilhelm et al. (2008), can be summarized by Figure 1. It is a timing distribution of the possible application execution time. There are two well defined limits: the BCET (*Best-case execution time*) and the WCET. The greater the difference between them, the worst is the predictability of a system. Since there is hardware complexity in modern systems, the BCET and the WCET cannot be calculated precisely, giving us the *lower* and the *upper timing bounds*. There is also a common industry practice of making measurements to obtain the system execution time. Measurements are generally not safe for strict timing constrained systems.

Modern processors use pipeline, various levels of data and instruction caches, dynamic branch prediction, out-of-order and speculative execution and fined-grained multithreading (instructions of various threads are dynamically executed in the processor pipeline). It can be incredibly difficult to model the software timing behavior when executing on a processor which uses such techniques for WCET timing analysis. The purpose of such timing analysis is to provide a safe worst-

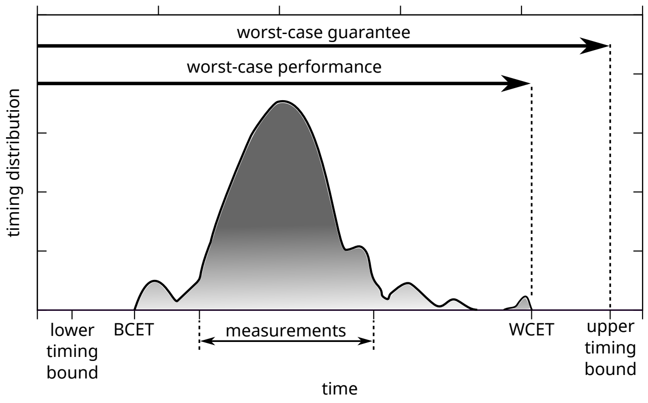

Figure 1 – The worst-case execution time problem (WILHELM et al., 2008).

case execution time bound but it should not be extremely pessimistic for practical use.

Future real-time systems need processors designed for predictability instead of investing a lot of effort in temporal analysis (SCHOEBERL, 2009b), (EDWARDS; LEE, 2007),(SCHOEBERL et al., 2011). But predictability alone is not enough. In such case, we would be employing 1980's processors where instruction timing is entirely known in datasheets. Real-time processors should have predictability and still a reasonable WCET performance. The purpose of a real-time architecture is to increase the worst-case performance (decrease the WCET) and simplify system analysis. This is the object of study of this thesis.

1.1 BASIC CONCEPTS AND MOTIVATION

Real-time systems are classified considering the criticality of their timing requirements. There are *soft real-time systems* and *hard real-time systems* (LIU, 2000). If timing requirements are not completely met in soft real-time systems, the quality of service is jeopardized like video streaming for instance. For hard real-time systems, timing requirements must always be met. Those systems are found in critical applications like aviation or space systems where temporal failures could lead to catastrophic consequences.

Real-time systems are modeled by tasks which are abstractions of works competing for resources. Tasks are comprised by timing parameters like WCET (C), maximum activation frequency or period (T) and deadline (D). The feasibility of a system is verified through schedulability analysis where it is ensured that all deadlines are respected. There is a lot of scheduling approaches for real-time systems (DAVIS; BURNS, 2011) and knowledge of the tasks WCET is always a concern.

The importance of obtaining the WCET of each task for system analysis is not a new problem. The question is how to estimate this fundamental parameter for systems where they are not only restricted to strict timing constraints but reasonable WCET performance is also required. This is the main motivation of this work where we will investigate processor architecture features capable of guaranteeing timing predictability, timing composability and increasing WCET performance.

1.1.1 Timing anomalies

The term anomaly denotes a deviation of the expected behavior from the real behavior. A timing anomaly is the unexpected deviation of real hardware behavior compared with the model used during temporal analysis (WENZEL et al., 2005). Predictions from models become wrong and this could lead to erroneous calculation results by WCET analysis methods. Thus, the concept of timing anomalies rather relates to WCET analysis and does not denote malicious behavior during execution.

WCET analysis is commonly performed in many phases. One phase is responsible for the processor hardware modeling and it computes upper bounds for program code snippets called basic blocks. The timing of each block is usually calculated independently from the others and it must consider an initial state. Due to lack of state information (the processor model may contain simplifications), it is assumed the worst behavior of hardware components (e.g. cache misses). Due to timing anomalies, assuming local worst case not necessarily means that the worst computation time is calculated. Figure 2 illustrates an example of timing anomaly. A cache hit in code snippet “a” triggers a prefetch. This prefetch evicts the contents of snippet “b” and the execution of $a \rightarrow b$ takes 8 cycles. On the other side, if there is a miss in “a” , there is no eviction of “b” and the execution of $a \rightarrow b$ takes 5 cycles. This simple example demonstrates that local worst case of “a” does not necessarily produces the global worst case of $a \rightarrow b$ execution due to a simple 3-cycle prefetch.

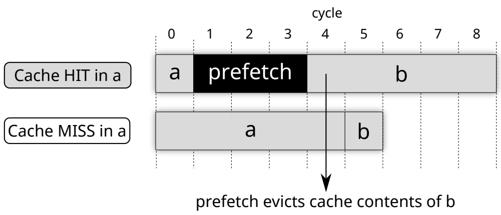

Figure 2 – Example of timing anomaly. Adapted from (WENZEL et al., 2005).

Timing anomalies are common for dynamic scheduling in out-of-order processors (LUNDQVIST; STENSTROM, 1999). There are methods to analyze this type of architecture but usually they lead to overestimations or extremely complex approaches. Sometimes processor features (like dynamic branch prediction) must be disabled to guarantee a feasible WCET calculation.

1.1.2 Timing predictability

A real-time system is not necessarily a high-performance system. A common error is to assume that a task has to run on a fast processor to meet its deadline. Fast processors usually improve average-case performance but can easily jeopardize the WCET analysis due to predictability issues.

To make sure that all tasks of the system meet their respective deadlines, it is necessary to determine the WCET of each one.

This is possible if the hardware is time predictable or time analyzable. This two concepts are formalized in (GRUND; REINEKE; WILHELM, 2011). Predictability is not a Boolean property. It should be expressed by ranges allowing the comparison of systems, i.e., system X is more predictable than Y. Moreover it considers some level of accuracy and the maximum level is obtained when the system can be exactly predicted. Analyzability is how the system timing is modeled and it indicates the capacity of this modeling to predict timing properties.

There are other predictability definitions as the one described in (THIELE; WILHELM, 2004). They use the difference between the real (exact) WCET and the estimated upper bound. Later, in (KIRNER; PUSCHNER, 2010), they used the range between real BCET and real WCET where a lower range implies a better predictability and BCET equals to WCET is the maximum achievable predictability.

Predictability definitions/quantifications that use exact values of WCET/BCET may be impractical because they actually cannot be known, only estimated (SCHOEBERL, 2012). Systems should be compared considering three basic aspects: hardware, compiler and WCET analysis tool. The relevance of two of them can be characterized as follows. A task τ has 1000 WCET cycles running on processor A, but 800 WCET cycles running on processor B (considering hypothetically both exact WCET values). Clearly, if we know those exact WCET values, processor B is better than A. Unfortunately, the WCET estimation is performed by a tool S_a for processor A and another tool S_b for processor B and their estimated upper bounds are 1100 cycles and 1300 cycles for A and B respectively. Now processor A is better. If predictability is estimated by $\frac{WCET\ real}{WCET\ bound}$ for instance, $P_A = 0.91$ and $P_B = 0.61$, A is more predictable than B and the tool S_a is also better. An efficiency problem arises if tool S_a takes one day and tool S_b takes minutes to estimate the WCET of task τ.

As we can see, the notion of predictability should capture whether a specified property of a system can be predicted by an optimal analysis and to what level of precision (GRUND; REINEKE; WILHELM, 2011)

1.1.3 Timing composabilty

The complexity and the level of the requirements of real-time systems are reasonable nowadays. It is necessary to analyse the system to verify if timing requirements are met. But as described in (PUSCHNER; KIRNER; PETTIT, 2009), there is a lack of methods and tools to effectively reason about the timing of software. It is difficult for real-time software systems to be constructed hierarchically from components while still guaranteeing timing properties. To achieve a hierarchical construction, system components should be both composable and compositional from the timing perspective. Composability focuses on preservation of properties of an individual component when it is integrated in an application and compositionality is the ability of deducing global properties of the composed system from properties of its constituent modules (MARTÍNEZ; CUEVAS; DRAKE, 2012). Besides compositionality and composability, other properties should be present (PUSCHNER; KIRNER; PETTIT, 2009) to achieve a composable timing analysis: support for hierarchical development process, predictability, scalability and performance.

Most hardware architectures used today can not provide the properties listed by (PUSCHNER; KIRNER; PETTIT, 2009). Regarding composability, real-time tasks executing on the same hardware compete for resources whose access times are state-dependent such as data cache memories and branch buffers (used in branch prediction). The state of these resources depends on the data addresses and on the access history. We can see that state depends on spatial and temporal aspects and, of course, the update strategy because both cache and branch buffers have space limitations. The use of those hardware mechanisms degrade composability because the property of an individual module or task is not preserved when it is integrated with other tasks. The lack of composability when using branch prediction and data cache memories degrades scalability as well because branch prediction interferes with cache contents. WCET analysis must consider both branch directions when the analysis cannot anticipate the outcome of the prediction.

Composability and predictability are also greatly affected in the presence of timing anomalies. Timing anomalies related to WCET analyses were first described by (LUNDQVIST; STENSTROM, 1999). A timing anomaly is a situation where the local worst case does not contribute to the global worst case, i.e., a cache miss, though increasing the execution time, results in shorter global execution time. The first condition to avoid timing anomalies is the use of in-order resources (LUNDQVIST; STENSTROM, 1999) what is not common in today hardware architectures. Timing anomalies jeopardize the composability because we cannot divide WCET calculation in subproblems. Predictability is also affected because simplifications on the analyses will not produce results with a reasonable accuracy margin.

In order to guarantee a composable timing analysis, state-of-the-art processor technologies such as dynamic branch prediction and cache memories with out-of-order pipelines should be avoided. Yet, as stated by (PUSCHNER; KIRNER; PETTIT, 2009), strategies adopted in real-time architectures should not lead to significant performance losses when compared to state-of-the-art technologies.

1.2 THESIS OBJECTIVE

The objective of this thesis is to investigate various processor architecture features that lead to a predictable design with reasonable WCET performance. The thesis to be demonstrated is that it is possible to assemble together hardware elements that increase performance but are predictable enough to ensure efficient and precise analyses. As described in the previous section, one of the first steps to demonstrate predictability is by obtaining the WCET. The construction of a WCET analysis tool as well as aspects involved with the hardware are also subjects of this work.

Among the architectural elements that are covered, we have:

- Processor pipeline:

Pipelining is a technique where complex operations are organized into sequential simpler ones to increase throughput. In the case of predictable processor design, pipelines are necessary but instructions should be executed in-order. Pipelines with out-of-order execution allow high average-case performance but jeopardize WCET analysis due to timing anomalies (LUNDQVIST; STENSTROM, 1999).

- Instruction parallelism:

  In modern processor design, the concept of superscalar is extensively used where more than one instruction is executed in each pipeline stage. This design overcomes the limitation of standard pipelines where the maximum throughput is one instruction per cycle. In the case of real-time processors, multiple instructions could also be executed in each pipeline stage using the Very Long Instruction Word (VLIW) design philosophy (FISHER; FARABOSHI; YOUNG, 2005). VLIW machines are better for real-time systems because instruction scheduling is fixed and defined offline during compilation time and, that enhances the analyzability. No hardware for instruction scheduling have to be modeled in VLIW design.

- First level of the memory subsystem:

  Memory subsystems designed for real-time systems are the subject of various recent works (REINEKE et al., 2011). There are several approaches and some of them require complex modifications in the compiler and/or overload the hardware (SCHOEBERL et al., 2011). In this work, we will address the memory predictability issues using a direct-mapped instruction cache and a scratchpad memory for data. Scratchpad memories are similar to caches but their contents must be managed explicitly by software.

- Branch prediction:

Branches are instructions that perform conditional control-flow modifications. They are used for *if*, *for* and *while* structures and they usually decrease pipeline performance adding stall cycles to the pipeline. One way to overcome this limitation is the use of branch prediction. There are dynamic branch predictions and static branch predictions. The use of dynamic branch predictions jeopardizes predictability and the static ones provide interesting WCET performance (BURGUIERE; ROCHANGE; SAINRAT, 2005). We support static branch prediction demonstrating its importance in terms of performance and we provide methods for correct WCET analyzability.

- Predication:

  Predication is a technique where instructions are conditionally executed based on a Boolean register. It is different from branches because there is not any control flow modification to execute or ignore instructions. There are two types of predication: partial and full. Full predication allows instructions to be executed or ignored directly based on a Boolean register (this type of predication is common in ARM architectures). In case of partial predication, instructions cannot be ignored through a Boolean operator but two values can be selected using special *select* instructions. Predication is an important technique to reduce program paths through inducing to the single-path programming paradigm (PUSCHNER, 2005). We support both partial and full predication but the latter is simplified to improve WCET analysis without jeopardizing performance.

- Complex arithmetic instructions:

  There are complex arithmetic instructions like division and multiplication that impose considerable overhead to the processor pipeline. Frequently, those instructions are only supported via software, mainly division. In this work we support both hardware division and multiplication and they are implemented to have constant timing independently of their input parameters.

We conduct a study of the impact on the WCET performance of those processor features that are typically disabled or not fully explored in real-time applications. Such features include the use of static scheduling VLIW processor with wide fetch, the importance of static branch prediction, the performance of complex instructions (memory, multiplications, division) and the use of predication.

Increasing the performance of real-time processors while preserving analyzability is a relevant subject. For that purpose, we analyze the WCET performance of the deterministic four-issue Very Long Instruction Word (VLIW) processor prototype describing its features and its timing characteristics. This prototype is implemented in VHDL using an Altera Cyclone IV GX (EP4CGX150DF31C7) in a DE2i-150 development board.

Besides the VHDL prototype, there are other products of this thesis like the implementation of the hardware modeling of the WCET analysis tool and cycle accurate software simulator. Both WCET analysis tool and the simulator are written in C++. In order to have a compiler for the architecture and to provide a customizable environment to research real-time compiler capabilities, a new code generator back-end for LLVM (LATTNER; ADVE, 2004) was implemented but it is out of the scope of this thesis.

In terms of test cases, we used representative examples of Mälardalen WCET benchmarks (GUSTAFSSON et al., 2010) which are commonly used for WCET evaluations.

1.3 CONTRIBUTIONS

Regarding our main contributions to real-time processor architectures, we can highlight:

- Design space investigation and evaluation:

We made a complete evaluation of the proposed techniques and we highlighted their performance benefits.

- Complete WCET analysis for the proposed design:

  We present all necessary analyses to perform WCET estimations of programs compiled for the design proposed in this thesis. Such analyses include cache modeling, pipeline modeling and worst-case path search. An integrated environment between WCET and compiler is utilized to enhance WCET estimation.

- Complete VHDL design and timing characterization:

  We describe the implementation of the researched approaches regarding our deterministic real-time processor. We describe the timing of each module and their VHDL implementation, which are necessary for WCET modeling and estimation.

- Detailed branch architecture:

  All details of the branch architecture are described including our methodology to model it during WCET analyses. We also extended the ISA to support static branch prediction and estimated the performance benefits of using or not this technology in real-time systems.

- Low overhead full predication system for VLIW processors:

  We propose a low-overhead full-predication system without adding overhead to the hardware data paths or its forwarding logic. The proposed predication system enhances the support of the single path execution as well as loop unrolling techniques.

1.4 TEXT ORGANIZATION

This thesis is organized in eight chapters including the Introduction which covers some basic concepts about predictability considerations, our main motivation and the thesis objective.

Chapter 2 describes techniques employed in modern processors and how they affect hardware predictability. Pipeline principles, in- and out-of-order execution, branch prediction and predication are topics in the scope of this chapter.

Chapter 3 presents a list of related researches with their main contribution and objectives.

Chapter 4 is the core of this thesis. We describe the thesis rationale and the problem defining our target system and its requirements. We discuss design decisions for the proposed architecture comparing them with the related researches and we highlight our contributions.

Chapter 5 presents the VHDL hardware implementation of the design. This chapter is important because the implementation is the *architecture realization* of the proposed design. With this realization, we can infer hardware complexity of each component and the timing of the processor. The timing behavior is important for the WCET estimation which is covered in Chapter 6. This particular chapter describes how we propose to analyze the processor using composability. Instruction cache modeling, pipeline analysis and worst-case path search are topics in the scope of this chapter.

Chapter 7 describes our evaluation methodology. We assess the impact of architecture techniques upon WCET performance. The main idea is to quantitatively show WCET performance gains considering features employed in our design such as wide instruction fetching, latency of memory operations, static branch prediction and predication.

Finally, Chapter 8 presents our final remarks and suggestions for future work.

2 PROCESSORS AND PREDICTABILITY CONSIDERATIONS

In this chapter we briefly describe some important processor aspects that will be used throughout this thesis. We also consider the impact of using those techniques in the design of a processor suitable for future real-time systems.

The design and specification of a microprocessor include its Instruction Set Architecture (ISA). It specifies what instructions the processor can execute. The hardware implementation is characterized by a Hardware Description Language (HDL) and it describes simple Boolean logic (logic gates), registers and more complex structures such as decoders and multiplexers. Entire functional modules such as adder and multiplier circuits can be developed with these HDL structures.

Literature usually defines three abstraction levels (SHEN et al., 2005) for computer architectures: architecture, implementation and realization. The architecture specifies the ISA and defines the fundamental processor behavior. All software must be encoded using such ISA. Some examples of ISA are MIPS32, AMD x86_64, ARM e ST200. The implementation specifies the project of the architecture, known as microarchitecture. The implementation is responsible for pipeline, cache memories and branch prediction specifications. The realization of a processor implementation is the physical encapsulation. In the case of microprocessors, this encapsulation is the chip. Realizations can vary depending on operating frequency, cache memory capacity, buses, lithography and packing. Realization also impacts on attributes as die size, power and reliability.

The ISA also defines a contract between the software and the hardware. This contract leads to an independent software and hardware development. Programs encoded using a particular ISA can be executed on different implementations with different levels of performance.

2.1 PIPELINE PRINCIPLES

Pipelining is a powerful technique to increase performance without massive hardware replication. Its main purpose is to increase the processor throughput with a low hardware increment. Throughput is the amount of instructions performed per unit of time – Instructions per cycle (IPC) . If a processor performs one instruction per unit of time, its throughput is $1/D$ where D is the instruction's latency. When using a pipeline, the processor subsystem is decomposed between several multiple simple stages adding a memory (buffer) between each of those stages. The original instruction is now composed of k stages and a new instruction can be initialized immediately after the previous one reaches the second stage. Instead of issuing an instruction after D units of time, the processor issues a new one after D/k units of time and the instructions are overlapped along the pipeline stages.

The performance improvement is ideally proportional to the pipeline length. However, there are physical limitations that reduce the number of feasible pipeline stages. In general, the limit of how a synchronous system can be divided into stages is associated with the minimum time required for the circuit operation: the uncertainty associated on using high frequencies and the distribution of circuit delays. Besides physical limitations, there is a cost and performance ratio that determines the optimal number of pipeline stages for a processor (SHEN et al., 2005) since hardware needs to be added for each added pipeline stage.

There are three basic challenges on using pipelines:

- Pipeline stages should be balanced to provide uniform sub-computations with same latency.

- Instructions should be unified to provide identical sub-computations using all stages.

- Instructions should be independent to reduce pipeline stalls.

Uniform sub-computations with the same latency reduce internal pipeline fragmentation. It is difficult to provide perfect and balanced stages and the pipeline operating frequency will be associated with the stage with higher latency. Processors also execute different types of instructions and not all of them will require all stages to execute. Since usually it is not possible to unify all sub-computations for all types of instructions, some degree of external fragmentation will exist and instructions need to pass every pipeline stage even if a particular stage is not required. If all computations were independent, a pipeline could operate in a continuous stream and it achieves to maximum throughput. Unfortunately, processor instructions are not fully independent and the result of one instruction could be required for the computation of the next one. If there is no hardware mechanism to forward this result to the next instruction, the pipeline must stall to guarantee instruction semantics.

One generic 5-stage pipeline is naturally formed by one basic instruction cycle divided into the next five generic sub-computations:

1. Fetching (F).
2. Decoding (D).
3. Executing (E).
4. Memory accessing (M).
5. Results writing back (WB).

This generic pipeline begins with an instruction being fetched from memory. Next, it is decoded to determine which type of work will be performed and one or more operators are fetched from the processor registers. As soon as the operators are available, the decoded instruction is executed and the cycle ends by writing back the result in the memory or in the registers. During these five generic sub-computations, some exceptions can occur by changing the machine state and they are not explicitly specified in the instruction. The complexity and latency of each stage varies significantly depending on the processor's ISA.

Considering a modern RISC processor, the generic 5-stage pipeline executes three basic instructions classes:

1. Arithmetic and logical instructions (add, sub, shift) – performed by an ALU (Arithmetic Logical Unit);

2. Memory instructions (load/store): performed by a memory unit that moves data between register and memory;

3. Control flow instructions (branch, calls and jumps): performed by a branch unit that manipulates the control flow.

Arithmetic instructions are restricted only to processor register operands. Only memory instructions access the memory. These restrictions form a load/store RISC architecture. Control flow instructions are used to manipulate the control flow to support function calls/returns as well as loop, if-then-else and while structures. They usually use relative addressing and the calculation of target address uses the program counter (PC) as reference.

Tables 1, 2 and 3 present instruction details related to the three basic instruction classes and the 5-stage generic pipeline (SHEN et al., 2005). Although they share sub-computation, they require different resources leading to idle stages.

Table 1 – Arithmetic instructions specification.

Stage	Task
F	Memory instruction access
D	Decode and read registers
E	Perform calculation
M	
WB	Write data into registers

Table 2 – Memory instructions specification.

Stage	*Load*	*Store*
F	Memory instruction access	
D	Decode and read registers	
E	Generate memory address (base + offset)	
M	Access data memory	
WB	Write data into registers	

Table 3 – Control flow instructions specifications.

Stage	*Unconditional*	*Conditional*
F	Memory instruction access	
D	Decode and read registers	
E	Generate target address	
M		Evaluate condition
WB	Write back new PC	If true, write back new PC

2.1.1 Minimizing pipeline stalls

A computer program is an assembly instruction stream. Shen et al. (2005) define an instruction as a function $i : T \leftarrow S_1\ op\ S_2$, where $D(i) = \{S_1, S_2\}$ is its domain and $I(i) = \{T\}$ is its image. S_1 and S_2 are source operands (e.g. registers) and T is the destination (e.g. register). The relation between the domain and the image is defined by operator op. Let these be two instructions i and j where i precedes j. Instruction j could be dependent on i if one of the following three conditions of Equations 2.1, 2.2 and 2.3 holds.

$$I(i) \cap D(j) \neq \emptyset \tag{2.1}$$

$$I(j) \cap D(i) \neq \emptyset \tag{2.2}$$

$$I(i) \cap I(j) \neq \emptyset \tag{2.3}$$

The condition of Equation 2.1 happens when an instruction j needs an operand belonging to instruction's i image. This is a *read-after-write* (RAW) or *true dependency* and it denotes that j must not execute until i completes. Equation 2.4 shows this situation using generic assembly instructions. Instruction i writes to register R_3 which is immediately read by j.

$$\begin{aligned} i &: R_3 \leftarrow R_1 \; op \; R_2 \\ j &: R_5 \leftarrow R_3 \; op \; R_4 \end{aligned} \tag{2.4}$$

The condition of Equation 2.2 happens when an instruction j writes into an operand of i. This is a *write-after-read* (WAR) or anti-data dependency and it denotes that j must not execute before i or i will operate using a wrong operand value. Equation 2.5 exemplifies this dependency in which register R_1 is read by i and it is immediately written by j.

$$\begin{aligned} i &: R_3 \leftarrow R_1 \; op \; R_2 \\ j &: R_1 \leftarrow R_4 \; op \; R_5 \end{aligned} \tag{2.5}$$

The third condition, Equation 2.3, happens when both instructions i and j write into the same register. This is *write-after-write* (WAW) or output dependency and it denotes that an instruction j must not complete before i. Equation 2.6 shows this dependency where both instructions i and j write into register R_3.

$$\begin{aligned} i &: R_3 \leftarrow R_1 \; op \; R_2 \\ j &: R_3 \leftarrow R_6 \; op \; R_7 \end{aligned} \tag{2.6}$$

All above dependencies must be respected to ensure correct program semantics. Pipelines could easily break instruction semantics and there must be mechanisms for identifying and resolving them. A potential dependency violation is known as a *pipeline hazard*.

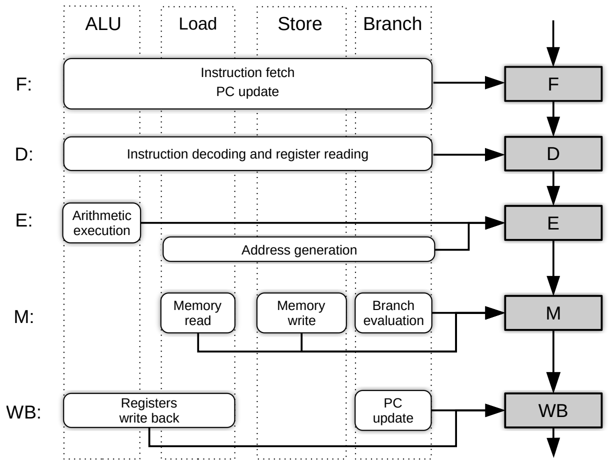

Figure 3 – Generic 5-stage pipeline representation.

Observing our generic 5-stage pipeline represented in Figure 3, not all hazards are automatically resolved. Considering the registers, they are read in stage D and written in stage WB. The *output dependency* is respected since writes are always sequential and only executed by stage WB. The *anti-data dependency* is also respected because every instruction *i* reads its operand before an instruction *j* writes into it. However, *true dependency* is not respected. If two instructions *i* and *j* are directly sequential and true dependent, when *j* reaches the stage D, *i* is still executing in stage E and, therefore, *j* must hold until *i* reaches stage WB.

There are also control dependencies involving program control flow semantics in our generic 5-stage pipeline. A control flow hazard is a *true dependency* on the program counter (PC). Control flow instructions write into PC and stage F reads and updates its value every cycle. If a conditional control flow is executed, it could update the PC with the target address on true condition or the next linear address could be fetched on false. Since the control flow instructions update the PC only in the stage WB, there is a *true dependency* violation because PC is read before WB updates the new address.

A simple way to resolve pipeline hazards is to prevent instructions progression until the dependent instruction leaves the critical pipeline stage. This could be done by an interlock hardware mechanism which identifies dependency violations and stalls the pipeline or by the compiler with instruction reordering (better instruction scheduling) and/or adding "no operation" instructions. Table 4 shows the necessary stall cycles for each type of instruction to guarantee instruction semantics.

Table 4 – Maximum necessary stall cycles for dependency resolution.

	Instruction type		
	ALU	Memory	Flow
Registers	R_i/R_j	R_i/R_j	PC
Write stage (i)	WB (5)	WB (5)	WB (5)
Read stage (j)	D (2)	D (2)	F (1)
Latency	3 cycles	3 cycles	4 cycles

We can see that adding stalls to prevent pipeline hazards substantially decreases the overall processor performance. Another feasible solution is to include hardware to build *forwarding paths* that forwards newer results to the necessary pipeline stage. On using forwarding paths, hardware mechanisms detect possible dependencies and multiplexers are added to units inputs to use newer values. For instance, stalls of two arithmetic sequential true dependent instructions are removed since the result of the first is ready in stage M and it is forwarded to stage W. Table 5 shows the updated stall values if we add this hardware to our 5-stage generic pipeline.

Most of dependency resolution stalls could be removed by using

Table 5 – Maximum necessary stall cycles for dependency resolution with forward paths.

	Instruction type		
	ALU	Memory	Flow
Registers	R_i/R_j	R_i/R_j	PC
Write stage (i)	WB (5)	WB (5)	WB (5)
Read stage (j)	D (2)	D (2)	F (1)
Forward output	E, M, WB	M, WB	
Forward input	ALU	ALU	F
Latency	0 cycles	1 cycle	4 cycles

forward paths, specially for ALU instructions. Memory to ALU stalls could not be totally removed since memory reads are executed in stage M and its value is not ready when a truly dependent instruction is already in stage E. The most critical situation is related to control flow instructions. The fetch stage reads the PC at the beginning of the pipeline stage and control flow instructions update it only at stage WB. It cannot be forwarded since it will never be updated before new instructions are fetched. One way to reduce control flow latency is using branch prediction but it could decrease predictability and it will be discussed in a further section. Figure 4 displays the data-flow overview when using our generic 5-stage pipeline with forward paths.

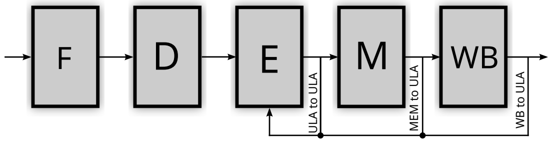

Figure 4 – Generic 5-stage pipeline with forward paths.

2.2 MULTIPLE INSTRUCTION FETCHING

Pipelines described in the previous section reach a theoretical k speed-up factor where k is the number of stages. Their maximum throughput could also reach about 1 instruction per cycle (IPC). Instead of dividing the work in k simpler stages, we can reach this performance replicating the work with k copies allowing a different type of instruction level parallelism (ILP). Standard pipelines provide temporal parallelism while allowing multiple instruction execution give us spatial parallelism. If temporal and spatial parallelism are combined, processors could reach more than one instruction per cycle. Pipelining allied with wider instruction fetch constitute superscalar and VLIW (Very Long Instruction Word) machines. Both types of machines provide higher ILP and the basic difference between them is related to instruction scheduling. In superscalar machines, the hardware is responsible for instruction scheduling while in VLIW machines, instructions are statically scheduled by the compiler.

Figure 5 shows a dual-issue pipeline where the execution stage E is composed of different types of execution units. This machine can, for instance, execute at the same time one ALU and one memory instruction. In this design, each functional unit can be customized to a particular instruction resulting in a more efficient hardware. Each type of instruction has its own latency and uses all stages of its functional unit. If the intra-instruction dependencies are resolved before being forwarded to the functional units, there will be no pipeline stalls. This scheme allows distributed and independent control of each sub-pipeline execution. Replicated units such as ALUs can be added to this design allowing dual arithmetic instruction execution.

2.2.1 Out-of-order execution

If out-of-order execution is allowed in the execution stage of a diversified pipeline, we constitute a superscalar machine with a dynamic pipeline. This is one of the most successful designs and this is the type of machine used in general purpose applications such as smartphones,

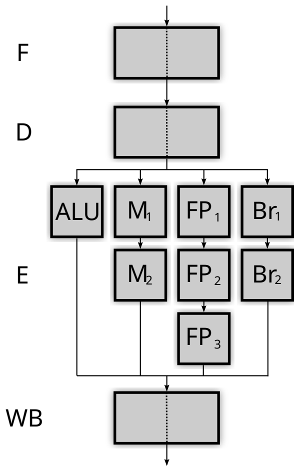

Figure 5 – Diversified pipeline example: M_x are memory sub-stages, FP_x are floating-point stages and BR_x are branch stages.

desktop computers and laptops. Figure 6 shows an example of this design.

Dynamic pipelines have two special buffers (dispatch and reorder) and a hardware instruction scheduler. Instructions are fetched and decoded in a multi-issue fashion and the hardware scheduler is responsible for instruction assignment in the correct execution sub-pipeline. Instruction dependencies are also checked and when there are stall generating instructions, independent ones are issued into the execution stage. After all dependencies are resolved, stalled instructions in the dispatch buffer are executed. In the pipeline end, there is a reorder buffer to guarantee correct semantic order.

Superscalar processors are complex machines intended to extract the maximum performance abstracting latency details during the compilation phase. These abstractions also provide better code compatibility allowing different processor implementations and realizations to be binary compatible. Complex hardware is necessary to provide out-of-order execution and more details of this design is presented in Shen et al. (2005). However, out-of-order execution leads to timing anomalies

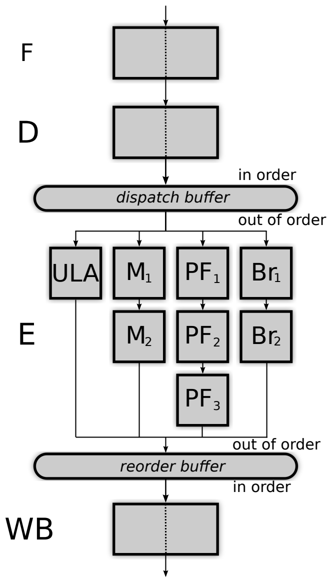

Figure 6 – Dynamic pipeline example: M_x are memory sub-stages, FP_x are floating-point stages and BR_x are branch stages.

and the high hardware complexity leads to unbounded states during WCET analysis.

2.2.2 In-order execution

Multiple instruction fetching with in-order execution and without hardware instruction scheduling leads to a VLIW (Very Long Instruction Word) design. The VLIW philosophy exposes the hardware not only at the ISA level but also at Instruction Level Parallelism (ILP). Superscalar machines extracted ILP using hardware features rearranging operations not directly specified in the code. VLIW processors do not perform any instruction scheduling and ILP must be provided by the compiler. Instruction rearranging (scheduling) is performed offline during compilation and the processor executes instructions exactly as defined by the compiler. The main idea is not to let the hardware do things that cannot be seen in programming, to reduce processor com-

Table 6 – Basic differences between VLIW and Superescalar designs (FISHER; FARABOSHI; YOUNG, 2005).

	Superescalar	VLIW
Instruction flow	Multiple scalar operations are fetched	Instructions are fetched sequentially but with multiple operations
Scheduling	Hardware dynamic scheduler	Compiler static scheduler
Fetch width	Hardware dynamically determines the number of fetched instructions	Compiler determines statically the number of fetched instructions
Instruction ordering	Dynamic fetching allowing out-of-order and in-order execution	Static fetching with only in-order execution
Implications	Superescalar design is related to microarchitecture techniques	VLIW is an architectural technique. Hardware details are exposed to the compiler

plexity, to have only simple instructions and to increase predictability when needed. Table 6 shows basic differences between superscalar and VLIW designs.

VLIW designs are not commonly used in general purpose computing but they are very popular in embedded applications. Several DSPs (Digital Signal Processors) like Texas Instrument C6x, Agere/-Motorola StarCore, Suns's MAJC, Fujitsu's FR-V, STss HP/ST Lx (ST200), Philips' Trimedia, Silicon Hive Avispa, Tensilica Xtensa Lx and Analog Devices TigerShark are VLIW processors. Intel's Itanium architecture for servers also uses this design philosophy.

VLIW compilers are more sophisticated since they must perform operations usually executed by the superscalar hardware. A superscalar control unit avoids sophisticated compilers. The fact that all ILP extraction is done by the hardware every time an instruction is fetched by the processor control unit shows that this work could be easily done by the compiler. The code reordering after out-of-order execution is relatively trivial and compilers can handle it easily and they can evolve faster than a new processor construction.

Considering real-time systems, VLIW is a very interesting solution since hardware modeling is considerably less complex, in-order execution avoids timing anomalies and higher performance can be achieved using multi-issue instruction fetching.

2.3 BRANCH PREDICTION

Conditional control flow instructions are usually responsible for *if* and *loop* program constructions and, unfortunately, they break the ideal pipeline flow reducing performance. Since control flow dependencies cannot be forwarded, as showed in Section 2.1, branch prediction techniques are commonly used to reduce this performance loss.

Instead of stalling the pipeline until the branch condition is evaluated and the target address computed, both are speculated and pipeline continues to execute a speculated path. This additional hardware is the *branch predictor* and it is responsible for condition/target address speculation and the recovery if the wrong path is taken.

General purpose modern processors use advanced history based branch predictors. This type of dynamic predictor registers previously branch directions as taken (T) or not taken (N) and their addresses as well. To predict the path of the next branch, this history is considered. History based predictors are very effective and they can predict 99% of branch directions (SHEN et al., 2005). Figure 7 illustrates the behavior of a two-bit predictor. Two-bit states and target addresses for each branch are registered into hardware buffers known as Branch Target Buffers (BTB) and they are essentially fully-associative caches.

Hardware elements which use execution history to increase performance are not very suitable for real-time systems. They are difficult

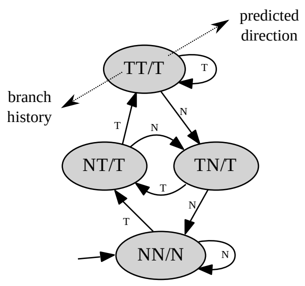

Figure 7 – Example of a two-bit branch predictor (SHEN et al., 2005).

to model in the WCET analysis and can lead to timing anomalies. Instead of using history-based, the use of static branch prediction is more interesting for real-time processors and they affect WCET more positively than dynamic ones (BURGUIERE; ROCHANGE; SAINRAT, 2005).

In processors with static branch predictors, the compiler could choose a static branch direction for each branch. If the common direction of each branch can not be determined, a default one must be used. The most common approach is to use the *not taken* direction as default, because this has less penalty in the case of a missprediction. If a branch is predicted as not taken and the condition is evaluated to false, no penalty occurs, because the fall-through path, which is the correct one, has already been taken by the processor pipeline. However, if this same condition is evaluated to true, the fall-through path must be flushed from the pipeline and the program flow must be resumed along the correct path (branch target). On the other hand, if a branch is predicted as *taken*, and the branch target address is determined, its path is executed until the resolution of the condition. If the condition is evaluated to true, we possibly have only the penalty of the cycles needed for the address calculation. But, if the condition is evaluated to false, the taken path must be flushed and the execution must be resumed along the fall-through path.

2.4 PREDICATION

Predication is an architecture technique that helps the compiler to convert control dependencies into data dependencies. By using predicates, individual instructions can be deactivated by the Boolean value without any program flow operation. This leads to conditional execution or guarded execution (FISHER; FARABOSHI; YOUNG, 2005).

This is a powerful technology specially for real-time systems because it gives the compiler the ability to reduce branch instructions and avoid branch prediction side effects and possible pipeline hazards. Many modern processors use this technique and its use will be illustrated by the simple C language code in Table 7. There are two types of predication: full and partial. In full predication, an additional bit guard (p_x) is read for each instruction and if it is false, this particular instruction is ignored. In partial predication, the processor's ISA supports a select or conditional move and a particular value is moved to its destination based on the bit guard. Table 7 shows the conditional code showed previously in normal mode (with branch), partial and full predication modes.

In the normal execution, the instruction at Line 1 makes the comparison (cmpgt: compare greater than) assuming that variable x is in register $r5$ and the result is stored at the bit register $p1$. brf performs a branch to $L1$ if $p1$ is false (brf: branch if false). We can note that the multiplication at Line 4 is ignored using a control flow instruction. Observing the full-predication execution, there is no control flow operation and the multiplication (mull) is automatically ignored if $p1$ is false. In the partial-predication, the multiplication is always executed at line 1 but its result is only considered if the select instruction (slct) at Line 4 moves the result to register $r3$ if $p1$ is true.

There are several works like (MAHLKE et al., 1995; CHOI et al., 2001) considering the predication for general purpose processors and Intel's Intanium has a very aggressive predication system. Besides helping to reduce control flow instructions, the average-case performance could be easily jeopardized because in both partial and full

Table 7 – Comparison between branch code, full and partial predication (HP ST200 ISA – Apendix A).

	if (x > 0)	x is r5; p1 stores comparison result	
	c = a * b;	c is r3; a is r1; b is r2	
	V[0] = c;	V[0] is (r0 + 0)	
	Branch code	Full predication	Partial predication
1	**cmpgt** p1, r5, 0	**cmpgt** p1, r5, 0	**mull** r4, r1, r2
2			**cmpgt** p1, r5, 0
3	**brf** p1, L1	**(p1) mull** r3, r1, r2	
4	**mull** r3, r1, r2		**slct** p1, r3, r4
5			
6	*L1:*		
7	**stw** r3, r0, 0	**stw** r3, r0, 0	**stw** r3, r0, 0

predication modes, instructions could be executed even if they do not belong to a given path. This situation is very well illustrated by the partial predication in Table 7 where a multiplication is always executed regardless of the path.

Considering the compiler, predication could also lead to better instruction scheduling because larger blocks of code could be produced. Normally it executes a pass known as *if conversion* that converts normal branch to predicated code. The predication system also allows an advanced compiler optimization called *software pipelining* (FISHER; FARABOSHI; YOUNG, 2005; GROSSMAN, 2000).

Full predication seems to be more flexible and easier to implement during *if conversion* but it needs special instruction encoding and increases considerably the forward paths, especially in multi-issue processors. The predicated mul instruction illustrated earlier must read three input operands (one predicate and two multiplication coefficients) and write one operand. This additional predicate operand must be read every time an instruction is decoded and 3 bits of instruction encoding are lost if seven predicates are supported. The partial system, meanwhile, needs only subtle ISA modifications adding few special conditional move instructions.

In the case of real-time systems, predication is important because it increases the code predictably and leads to programs with fewer paths (GEYER et al., 2013). The performance loss of the average case is also not an issue because real-time systems need performance of the worst-case scenario and fewer differences of the best-case, average-case and worst-case times lead to higher predictability.

2.5 CACHE MEMORIES

Cache memories are generally used to minimize the gap between processor and memory performances. Usually, main memory is slower than the processor so a fast cache is placed between them where most recent data is stored promoting faster access.

A cache memory is characterized by its capacity, line size and associativity. Capacity is the caches' total number of bytes; line size is the quantity of bytes transferred from memory to cache when a cache miss occurs. A cache could have $n = \frac{capacity}{line.\,size}$ lines. Caches are much smaller than main memory and associativity consists of the mapping of various main memory addresses to cache lines. A direct-mapped cache has unitary associativity, where a specific main memory address is always mapped to the same cache line. If associativity is two, a main memory address is mapped to 2 different cache lines. When associativity is not unitary some sort of replacement policy must exist to decide which line will have data eviction. There are basically three policies implemented in the cache design: FIFO (first in, first out), LRU (Least Recently Used) and random.

Considering cache misses, there are three main causes:

- Compulsory misses or *cold cache*: they happen when there is the first reference to a particular memory address. These are fundamental misses and they cannot be completely avoided by any technique.

- Capacity misses: they happen when there is no space left in the cache to store a particular memory block. Increasing the cache size can eliminate some or all of the capacity misses.

- Conflict misses: they happen due to imperfect allocation and mapping of memory blocks to cache lines. Changes in associativity or indexing function can improve conflict misses.

Various cache levels could also be used in processor design with different sizes and associativities. However, these complex cache designs are not suitable for real-time processors because they can easily forbid WCET analysis due to complex cache interference and no deterministic data eviction (SCHOEBERL, 2009b). Shared instruction and data caches should also be avoided. During worst-case cache analysis, accessed memory addresses must be predicted. Data from the instruction memory are easier to determine if the call tree is known. However, dynamically allocated data (heap) are difficult to predict statically because such addresses are only available at runtime. Without knowing the address, a single access can influence all cache sets. Predictable eviction policies like LRU are also more suitable for real-time applications.

2.6 SCRATCHPAD MEMORY

Scratchpad memories are designed with the same technology of cache memories, but they do not implement hardware data management. On using scratchpads, the compiler must explicitly manage data memory using static or dynamic allocation. When exposing the memory hierarchy with scratchpads, more work must be done by the compiler. However, timing of memory accesses depends only on which and how many memory blocks are accessed and additional latencies or unknown states provided by the hardware controller are non-existent. The predictability associated with utilizing scratchpads is very interesting for real-time applications. Since hardware controllers are non-existent, this additional area can be used in order to increase scratchpad size.

Scratchpads and main memory address spaces are usually unified and transfers between them are performed by Direct Memory Access (DMA) protocols (BANAKAR et al., 2002). The difference between an access to the main memory and to the scratchpad is only their addresses. Furthermore, there is not any hardware coherence protocol between memories. If necessary, such coherence must be developed in software.

There are various methods of performing data management for scratchpads for real-time systems including analysis for preemptive multitasking (WHITHAM et al., 2012). There are additional approaches that favor static partitioning of the scratchpad among real-time tasks (WHITHAM; AUDSLEY, 2010).

2.7 CHIP MULTITHREADING

Chip multithreading is another technique to increase processor performance. Typically, fine-grained multithreading is implemented where instructions of different threads are overlapped in the processor pipeline. The execution stage with various units is shared among multithreading fetch units and instructions of diverse threads are executed in parallel. Since different execution contexts are necessary, general purpose registers and state registers are replicated by the number of supported hardware threads. If there is some resource contention, like a single memory unit for instance, threads are typically stalled until this resource is available. This is the basic principle of Intel's Hyper-threading technology and a dual multithreading system behaves like a dual core system for the operating system.

Fine-grained multithreading is possible for real-time systems if the hardware thread scheduler and resource contention protocols are well defined and predictable. Round-robin scheduler and Timed Division Access (TDM) protocol are usually employed. Multithreading can also remove pipeline dependencies. If we have a five-stage pipeline with five hardware threads, it is possible to remove all forward logic since for each pipeline stage there will be a different thread instruction executing and pipeline dependencies never occur. This is the fundamental idea of the work of Liu et al. (2012). However, if the number of activated threads is less than the number of hardware threads available, the number of instructions per cycle is severely penalized. Considering a scenario with only one activated thread and five hardware threads, the number of instructions per cycle is only 0.2 since a newer instruction can only be issued after the old one is completed. Memory access for multithreading systems should also be special to remove or decrease thread interference.

Some works like Schoeberl (2009b) criticize chip multithreading for real-time systems since its analysis is more complex and the replicated hardware could be used for chip multiprocessing.

2.8 CHIP MULTIPROCESSING

Most modern computer systems available today use processors with a single chip and multiple cores. Commonly these cores share a single main memory and they are developed with complex and large shared caches. Memory consistency between cores is usually managed by hardware and there are complex internal chip networks to support data transfer and consistency between all cores.

Real-time systems with chip multiprocessing are also viable and this is a tendency as the complexity of these systems grows. Unfortunately, the technology employed in general purpose computing does not apply for real-time systems because it increases average-case performance but predictability proprieties are lost. Shared caches and complex hardware networks make the WCET analysis unfeasible. There are several works like Schoeberl et al. (2015) dealing with multiprocessor designs for real time. In most approaches Timed Division Access (TDM) protocols are accompanied by real-time networks and data transfer between cores is explicitly managed.

2.9 SUMMARY

In this chapter, we briefly surveyed some relevant processor aspects and their relation to real-time systems. These aspects are important for the subsequent chapters and some of them ensure predictability and performance while others can severely jeopardize WCET analysis capabilities. Table 8 shows a comparison of techniques used by general purpose processors and their alternative for real-time systems.

Future real-time applications will need performance and a predictable pipeline is necessary to provide instruction temporal parallelism. When using pipelines, it is equally important to explore and optimize their number of stages, hazard resolution and avoid unnecessary stall cycles to keep the pipeline filled with useful work.

Spatial parallelism is also necessary and it is the natural evolution of the standard pipeline. Fetching multiple instructions and executing them out of order when possible is a common technique employed in modern processors. Unfortunately, this leads to time anomalies and should be avoided for real-time systems. Spatial parallelism can be achieved by using various simple processors, multiple threads execution or with a VLIW design. The first approach must deal mainly with deterministic inter-processor communications and shared resources like main memory. The second must employ deterministic thread schedulers and also deals with shared resources. The last one can achieve high performance with a powerful compiler. In the case of the VLIW approach, it is possible to design systems of multiple threads but chip multiprocessing is more feasible because multithreading support needs critical hardware replication like the register file and this is commonly a very large and critical component in VLIW processors.

Table 8 – Comparison of techniques for general-purpose and real-time processors

Feature	General purpose	Real time
Pipeline	Long with out-of order execution	Simple with in-order execution
Caches	Complex with various levels, shared and complex associativities	Separate caches or scratchpad memory
Branch	Dynamic predictors	Static or non predictors
Superscalability	Multi-issue and out-of-order	Various simple cores or in-order multi issue (VLIW)
Chip multi-threading	Lots of hardware threads competing for shared resources	Avoid, use multiple cores or threads with interleaved pipeline
Chip multiprocessors	Shared caches with core inter-interference	Separated caches or scratchpads with predictable core communication

3 RELATED WORK

The design of deterministic computer architectures for real-time systems is very relevant and there are several works in this research field. Since high-performance general-purpose processors are not suitable, it is desired to have new predictable architectures enhancing the WCET analyzability while not ignoring more stringent performance requirements. The main concern is related to WCET analysis.

In this chapter we survey related work on predictable architectures. Compiler techniques, timing analysis techniques and architectural techniques are all important for the predictability of computer architectures. However, we focus mainly on architectural techniques for hard real-time systems only to limit the scope of this survey. We can also note that all the predictability considerations described in Chapter 2 are considered for all related works.

3.1 THE *KOMODO* APPROACH

Uhrig, Mische and Ungerer (2007) propose a multithreaded Java processor with an integrated real-time scheduler named *Komodo*. It is an IP core for Altera's System-on-Programmable-Chip environment [1] having a five-stage pipeline: instruction fetch, instruction decode, operand fetch, execute and stack cache.

Komodo executes Java applications directly without a Java Virtual Machine or an operating system. Reducing those software layers increases real-time capability and predictability that is very difficult when a virtual machine or operating system has to be statically analyzed. The main purpose of the *Komodo* approach is to verify the suitability of hardware multithreading for real-time event handling in combination with appropriate real-time scheduling techniques (KREUZINGER et al., 2003). They support the following hardware schedulers: Fixed Priority Preemptive (FPP), Earliest Deadline First

[1] https://www.altera.com/products/design-software/fpga-design/quartus-ii/quartus-ii-subscription-edition/qts-qsys.html

(EDF), Least Laxity First (LLF) and Guarantee Percentage (GP).

Java threads are invoked as Interrupt Service Threads (ISTs) instead of the Interrupt Service Routines (ISRs) of conventional processors with zero-cycle context switching overhead. A real-time garbage collector is also included in this approach and it executes within its own hardware thread slot at the same time of other application threads.

In terms of data memory, they used an operand stack of 2k-entry stack cache integrated within the pipeline and it is shared between all hardware thread slots. Also, memories are overclocked to guarantee single cycle pipeline access avoiding threads interference and predictable timing. Static Random Memories (SRAM) should be clocked as a factor of two but dynamic ones (DRAM) require a higher factor. The instruction memory is simple using a scratchpad RAM, cache or combined (scratch and cache). When configured with a scratch RAM, it is used for the most important trap routines and all other instructions are fetched out of the memory. Cache is a small 128 byte direct mapped instruction cache and all instructions including the trap routines are fetched out of the main memory or the cache. The combined method includes a scratchpad memory for traps and cache for applications. Traps are not cacheable.

Measurements were made in Uhrig, Mische and Ungerer (2007) to evaluate the performance of the whole Java system and the resource requirements but WCET obtaining was out of the scope of their work. A frequency of 33 MHz was achieved implementing *Komodo* with four threads in a Altera's Cyclone II EP2C35F484C7 FPGA. Their evaluations and conclusions showed a very low pipeline utilization for single threaded applications (33%) but it could reach 92% when using four threads.

3.1.1 Real-time jamuth

Based on *Komodo*, *jamuth* (*Ja*va *mu*lti*th*readed) was introduced in Uhrig and Wiese (2007). It is also an IP core for Altera's System-on-Programmable-Chip environment but three additional units were introduced: evaluation unit, java timer and Interrupt controller (IRQ). The first is responsible for debugging, timer is used for Java temporal methods and IRQ is responsible for peripheral components and thread wake-ups.

The memory interface of *jamuth* was also changed and now connected to external buses, in this case, the Altera's Qsys interconnect [2] and can no longer support single cycle access. Due to this change, instructions are now fetched from three different sources: external memory, instruction cache and scratchpad memory. Real-time threads and a garbage collector are allocated to the scratchpad. The internal thread scheduler is simplified and supports only Fixed Priority Preemptive (FPP) and Guarantee Percentage (GP) schedulers.

In terms of evaluation, measurements were made in Uhrig and Wiese (2007) to evaluate pipeline utilization and the garbage collector using different Java benchmarks and various memory configurations. WCET obtaining is also not considered in their work.

3.2 THE JOP JAVA PROCESSOR

JOP is a small Java processor for embedded real-time systems introduced by Schoeberl (2008) and it stands for *Java Optimized Processor*. While *Komodo* is intended to investigate the suitability of hardware multithreading, JOP is single threaded and it is optimized for embedded Java systems.

This processor is a stack computer using its own instruction set. Java bytecodes are translated into single or sequences of microcode instructions. It has three microcode pipeline stages: microcode fetch, decode and execute. Another stage is necessary for bytecode fetch. A typical JOP configuration contains the JOP core, a memory interface and IO devices.

What makes JOP very noteworthy is its cache architecture. Instructions are fetched from a special method cache (SCHOEBERL, 2004). Also, no conventional register file is required and Java byte-

[2] http://quartushelp.altera.com/13.1/mergedProjects/system/qsys/qsys_about_sys_interconnect.htm

codes operate directly in a runtime special stack cache (SCHOEBERL, 2005).

The idea of the method cache is to store complete methods where a main memory transaction is only performed on a miss during method invocation or method return. It is organized in blocks and their replacement depends on the methods call tree, instead of instruction addresses. The average-case performance of the method cache is similar to a direct-mapped cache and Schoeberl (2008) argue that it is easier to analyze during WCET computation.

The stack cache operates inside the processor pipeline using a dual-ported memory and it also includes the stack management. The stack is a heavily accessed memory region and it is organized in two levels. The two top elements are implemented as registers and the lower level as an on-chip memory. Hardware controls data between these two levels but data management between the on-chip memory and the main memory are subjected to microcode control. It occurs when there is a method invocation, method return or thread switch.

The simplicity of JOP yields to tight WCET bounds as described in Schoeberl et al. (2010). In this work, they describe a tool for WCET analysis considering JOP and several experiments where benchmarks are conducted.

3.3 THE MCGREP PROCESSOR

MCGREP is a re-configurable processor for real-time systems and it stands for Microprogrammed Coarse Grained Re-configurable Processor (WHITHAM; AUDSLEY, 2006).

It has a two-stage pipeline where most opcodes execute in one machine cycle and longer opcodes stall the pipeline. Memories and I/O devices are connected by a single bus and there are no caches. Essentially MCGREP works as a simple processor but the re-configurable part permits application specific operations to be encoded as single instructions to increase performance.

They use the *gcc* compiler with OpenRisc ISA [3] and a microcode compiler to generate application-specific instructions. Differently from other approaches, MCGREP adds a new option for predictable processor where it achieves performance by accelerating application-specific functions using reconfigurable logic.

Whitham and Audsley (2006) demonstrate the timing of several operations but evaluations are performed using instructions per cycles instead of WCET timing.

3.4 THE PRET ARCHITECTURE

PRET stands for *pre*cision *t*imed machine and its main purpose is repeatable timing instead of time predictability (LIU; REINEKE; LEE, 2010). Repeatable timing gives an exact timing of each processor instruction regardless of any dependencies.

Liu et al. (2012) implements PRET using a subset of the ARMv4 ISA and an in-order, single-issue five-stage pipeline with thread interleaving. It is a multithreaded architecture as *Komodo*, but they use a thread-interleaved pipeline to increase the performance using a round-robin thread scheduling policy to reduce the context-switch overhead to zero and to maintain repeatable timing for all hardware threads.

There is an unique hardware thread executing in each pipeline stage. This minimizes instruction dependencies and inter-pipeline stage forward logic. They also do not use caches, only scratchpad memories explicitly controlled by software.

PRET also introduces timing instructions to the ISA level added to the co-processor instructions slots: *get_ time*, *delay_until*, *exception_on_expire* and *deactivate_exception*. *Get_ time* is used to obtain the current clock value, *delay_until* is used to postpone thread execution until an input timestamp, *exception_on_expire* and *deactivate_exception* controls exception generation using timing values. Those instructions are used for deadline checking or timing checking for a given code snippet.

[3] https://sourceware.org/cgen/gen-doc/openrisc.html

This architecture avoids resource conflicts at the memory interface using bank privatization in Dynamic Random Access Memories (DRAM) as described in Reineke et al. (2011). In this work they introduce repeatable DRAM controller where refresh cycles are not scheduled by the memory controller. Banks are refreshed individually in case the application is not accessing memory. For instance, control flow instructions send refresh commands to the private bank of its current thread slots.

3.5 THE PATMOS APPROACH

Schoeberl et al. (2011) describe a VLIW processor named Patmos. It is a statically scheduled, dual-issue RISC processor that is optimized for real-time systems. Patmos has two five-stage pipelines and a register file shared among them. This architecture is also fully predicated. They could execute up two instructions per cycle with exception of branches and memory access that leads to pipeline stalls.

Patmos follows the same memory architecture as JOP (SCHOEBERL, 2008) with specialized caches: cache for functions (method cache), cache for stack, cache for data and a general purpose scratchpad memory. Each cache is specialized considering data and configuration. For instance, the stack cache has direct-mapped configuration and the data cache has a highly associative configuration. Memory loads are executed in phases: the result of a memory load is stored in a dedicated register that it is read later by a special instruction. Each data cache is also accessed by a different type of memory instruction.

The programming model of Patmos is more complex than that of traditional processors. The compiler must explicitly select memory instructions to access different caches and two instructions are required for memory loads. Greater effort spent in the software development could improve predictability. All instruction delays are explicitly visible at the ISA-level and there are no variable timing instructions except for cache accesses. Schoeberl et al. (2011) argue also that various caches could also improve the analyzability resulting in a less pessimist WCET estimation.

Patmos is evaluated with a Xilinx Virtex 5 FPGA. The primary purpose of Patmos is its use in the T-CREST (SCHOEBERL et al., 2015) project where they propose novel solutions for multi-core architectures designed for real-time applications.

3.6 THE CARCORE PROCESSOR

The CarCore processor (MISCHE et al., 2010) is based on the Infineon's TriCore microcontrollers[4] which target the automotive industry like gasoline and diesel engine control systems. TriCore combines a load-store architecture with DSP (Digital Signal Processor) instructions forming a very large instructions set (700 instructions) (MISCHE et al., 2010). It consists of three different in-order pipelines with four stages and instructions are statically assigned to each pipeline (static scheduled). One pipeline executes integer instructions, another takes care of special loop instructions and the third executes memory operations. If three of these instructions are issued in order into a stream, they are executed in parallel even if they are data-dependent. Two of those pipelines have their own register file with 16 registers.

The main purpose of the CarCore is to have multithreading capability like the *Komodo* approach (UHRIG; MISCHE; UNGERER, 2007) but with a two layer hardware scheduler. The first layer is responsible for thread instructions issue considering instructions with higher priority. Pipeline stages with stall slots generated by control flow or memory instructions are filled with instructions from other active threads. The purpose of this layer is to keep the pipeline utilization as high as possible. The second layer is the thread manager and it analyzes scheduling information like deadline, priorities and thread slots. It determines which thread is the next to be scheduled.

CarCore uses three hardware thread scheduling techniques: Dominant Time Slicing (DTS), Periodic Instruction Quantum (PIQ) and

[4] http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/channel.html?channel=ff80808112ab681d0112ab6b64b50805

Round Robin Slicing (RRS). DTS is used for hard real-time tasks where all DTS threads are executed for a predefined number of pipeline cycles and only a single DTS thread is executed at each time. PIQ threads are soft real-time threads and they are executed for as many instructions as defined by a scheduling parameter. RRS is employed for non real-time tasks and these threads are scheduled via round robin after all DTS and PIQ are executed.

3.7 MULTICORE SOLUTIONS

Multicore architectures for real-time systems usually imply some of the previous cited approaches interconnected via some real-time network-on-chip. Network-on-chips and multicore memory hierarchy are not the subject of this thesis but there are two relevant approaches: Merasa (UNGERER et al., 2010) and T-CREST (SCHOEBERL et al., 2015) projects.

3.7.1 The Merasa project

Ungerer et al. (2010) describe the Merasa architecture. The main focus of this project is to develop a multicore processor design for hard real-time embedded systems and analysis tools.

Merasa connects modified versions of CarCore (MISCHE et al., 2010) processors. The number of cores is configurable and varies from one to eight and they are connected by a system bus to the memory interface and to a central dynamically partitioned cache. This cache has multiple banks and is also connected to the memory interface. Typically each cache bank is allocated to a different core. The interconnection of the system is a bus and it is scheduled by a hybrid priority and time slice (TDM – Time Division Multiplexing) algorithm.

Differently from CarCore, Merasa cores execute only one hard real-time and various soft real-time threads but there is no dynamic thread swapping. Soft real-time thread access to the cache is not fully integrated due to cache interference among hard real-time threads.

Ungerer et al. (2010) also present a quad core FPGA prototype synthesized in an Altera Stratix II EP2S180F1020C3 device running around 25Mhz.

3.7.2 The T-CREST project

The T-CREST project considers design techniques from the multicore processor level regarding cores, memory, network-on-chip (SCHOEBERL et al., 2015) and the analysis tools.

The platform is composed by Patmos (SCHOEBERL et al., 2011) processor cores interconnected by two network-on-chip. One networks-on-chip provides messaging passing between core nodes and the other provides access to the shared external memory. Patmos cores use caches but there is not any kind of hardware support for cache coherence. Processor communication via shared memory is allowed but coherence must be implemented by software. The interconnect network use packet switching and source routing and supports asynchronous message passing across point-to-point virtual circuits. It is implemented using DMA-driven block transfers from the local processor scratchpad into remote scratchpad processor nodes. Network virtual circuits are implemented using TDM (time division multiplexing) of the resources.

Instead of using benchmarks to evaluate the system, they use industry test cases based on avionics and railway applications.

3.8 SUMMARY

In this chapter, we surveyed various related projects. In many of them, the focus is on hardware multithreading while others concentrate on the memory hierarchy. Table 9 summarizes the described projects.

Real-time hardware multithreading allows higher pipeline utilization and promotes deterministic thread execution. There are thread-interleaved approaches as the PRET architecture and sophisticated hardware thread scheduling approaches like *Komodo* and CarCore. Besides some advantages, hardware multithreading is a complex system and additional hardware elements like the register file must be replicated. Schoeberl (2009b) argues that the additional hardware could be employed to chip multiprocessing instead of chip multithreading.

Patmos and JOP use a unique cache system where several specialized caches are proposed. It requires effort on the part of software development, but it can enhance the analyzability of the system. The compiler must explicitly select memory instructions to access different caches and two instructions are required for memory loads.

We can note that all related works have a main objective regarding determinism and differently of them, our work is concentrated in the investigation of a VLIW processor for hard real-time applications. There are many design space investigations and it is not clear how they impact on the WCET performance. Among them, we can highlight: pipeline dependencies resolution, full predication for 4-issue VLIW processor, branch architecture and all of these features are considered in the WCET analysis tool.

Table 9 – Summary of related project objectives

Project	**Main objective**
Komodo	Java processor: feasibility of real-time hardware multithreading
JOP	Java processor: cache architecture (method, stack, ...)
MCGREP	Reconfigurable architecture: application-specific instructions
PRET	Precision timed machine: thread-interleaved pipeline
Patmos	Dual VLIW: memory architecture (method cache, stack cache, data cache, scratchpad)
CarCore	Heterogeneous pipeline: hardware multithreading with different hardware scheduling algorithm
Merasa	Multicore: CarCore processors interconnected by bus
T-CREST	Multicore: PATMOS processor interconnected by dual network-on-chip
Our work	**VLIW processor design space investigation: predication, branch architecture, pipeline dependencies resolution and their WCET analysis**

4 THESIS RATIONALE AND DESIGN DECISIONS

The analysis of real-time systems is a challenging task and requires the union of numerous techniques. Typically the problem is divided into two main areas: scheduling analysis and temporal analysis of individual tasks.

Scheduling analysis is responsible for verifying the system feasibility as a scheduling problem: we want to verify if a task set can execute properly respecting the individual deadline of each task.

The formulation of the scheduling problem will always depend on the temporal system parameters. Obtaining these parameters is the responsibility of the task's individual temporal analysis. Some of these parameters are dependent on what we want to control or sample using a real-time system, like the activation frequency or task period ($\boldsymbol{T}$). However, a key parameter is the Worst-Case Execution Time (WCET) of each task – ($\boldsymbol{C}$). Every scheduling analysis depends on the WCET. The focus of this work is on the temporal analysis of individual tasks, more specifically, on researching deterministic hardware techniques to improve the WCET analysis but also increase processor performance.

As noted briefly in the previous chapters, in recent years considerable research effort has been done on providing time-predictable architectures for real-time systems. Since general-purpose processors usually focus on average-case performance (SCHOEBERL et al., 2015), analyzing real-time systems and obtaining WCET using those processors became very complex. Real-time applications are becoming increasingly more complex requiring more processing power with embedded systems providing more and more "desktop-like" functionality. However, the use of standard processor design choices to improve performance is usually not possible for real-time systems because it greatly jeopardizes determinism proprieties.

In this chapter, we describe the target problem and the thesis rationale. We will present the target systems, their requirements and discuss design decisions for the proposed architecture comparing them with the related researches.

4.1 TARGET SYSTEMS AND REQUIREMENTS

Currently the use of microprocessors is ubiquitous. They are widely embedded in general consumer goods (cameras, TVs, videogame consoles, robots), automotive market (navigation, vision), aviation and communication (cell phones, routers). Another market for microprocessors is the personal computer (PC) as the main processor. Because of the versatility of a PC, its processor is usually called a general-purpose processor and the user can reprogram it and extend its functionality with new software using complex and big applications. In the case of embedded computing systems, they are encapsulated in a device and generally are not intended to reprogramming by the user. Moreover, embedded processors, which are the focus of this work, must comply with various non-functional requirements:

- Instead of running functions as quickly as possible, embedded applications often must respect the real-time requirements (compliance with deadlines).
- Embedded devices, especially powered by batteries, have energy restrictions.
- Instead of binary compatibility, source compatibility is more important.

Safety standards for embedded applications (DO-178B, DO-178C, IEC-61508, ISO-26262, EN-50125, etc.) require identifying potential functional and non-functional hazards and we must demonstrate that the software complies with safety goals. Testing with intensive measurements is tedious and also typically not safe. It is often impossible to prove that the worst-case conditions have actually been taken into account. As DO-178B succinctly puts it, "testing, in general, cannot show the absence of errors".

Despite the differences between general purpose and embedded computing, the addition of new functionality such as audio/video processing, networking, security, automation and control, among others, increases the need for more advanced processors for embedded applications. However, deterministic hardware is necessary when the application has hard real-time requirements.

In terms of target systems and requirements, we will focus:

- *Target systems:* embedded systems.
- *Low-level requirements:* hard real-time. The proposed design must be time predictable allowing safe and precise WCET estimations. We must design techniques to obtain worst-case behavior and timing of every used hardware component.
- *High-level requirements:* the architecture should use an ISA – Instruction Set Architecture – that allows the use of a modern C compiler. It should provide clear and well defined instruction encoding, adequate set of arithmetic instructions, typical memory access instruction with base plus offset addressing mode and allow the use of function calling/returning.

4.2 THESIS OBJECTIVE

The general objective of this thesis is to investigate various processor architecture features that lead to predictable designs with reasonable WCET performance. The thesis to be demonstrated is that it is possible to assemble together hardware elements that increase performance but are predictable enough to ensure precise and efficient temporal analyses. In order to improve them, especially the worst-case execution time analysis, the design of processors should increase the level of importance of the determinism.

Therefore, it is necessary to analyze the strengths of hardware techniques used in state-of-the-art processors, for example temporal and spatial parallelism (pipeline), branch prediction and predication, and to align them to real-time applications. Some of these techniques require modifications due to high hardware complexity while others need a well-defined timing behavior. In order to achieve these goals, we propose new approaches considering the following items which are described in the next sections:

- Target architecture definition: based on the predictability considerations, what is a good target architecture model?
- Predication: how predication could be improved without jeopardizing pipeline performance due to overheads in the encoding and data paths?
- Branch prediction: how much performance is gained by using branch prediction? How we define its timing behavior?
- Pipeline dependencies: which are the main sources of dependencies and how to deal with them?
- High latency operations: how much performance is lost with high latency operations like memory access, multiplication and division? How to deal with it?

Besides the approaches exposed above, one of the most important tasks of this thesis is to propose a WCET analysis for the proposed architecture. This analyzer and Mälardalen WCET benchmarks (GUSTAFSSON et al., 2010) are used to make evaluations regarding determinism and performance.

4.3 ARCHITECTURE

After having laid the ground work for predictability considerations of processors, related research, our target class of system and its requirements, we noticed that there are basically two main approaches for real-time computer architectures: chip multithreading and VLIW – Very Long Instruction Word processors, since superscalabilty and out-of-order execution are not adequate.

Chip multithreading permits several threads to share pipeline resources possibly increasing performance and pipeline utilization. However, it requires some critical components (such as the register file) to be replicated for the number of threads because the hardware needs different execution contexts. Multithreading as a thread interleaved pipeline (used in PRET) could also remove data dependencies between pipeline stages because in each different pipeline stage there is a different thread executing. Using this technique, in (LIU et al., 2012), it is argued that they do not use forwarding logic and use round-robin for fine-grained thread scheduling. If there is no forwarding logic between stages and less than 4 threads, some sort of stalling should occur. On the other hand, the use of round-robin thread scheduling impacts the response time of the task set: fetching the next instruction for a thread must wait the fetch for all other threads, i.e., if we have 4 threads and a pipeline with 4 stages, after 4 cycles another instruction is fetched for a particular thread.

In contrast to chip multithreading, VLIW machines increase single-threading performance where instructions are executed in the pipeline in temporal and spatial parallelism, i.e., more than one operation in each pipeline stage. To do so, execution units like arithmetic units are replicated, being this a common practice in state-of-the-art processors. Differently from those processors, VLIW instructions are statically scheduled during compilation, which is very interesting because it is defined offline and does not depend on hardware fetch states. VLIW typically uses less power comparing to superscalar out-of-order designs since less hardware is required for instruction scheduling. Compared to PRET, VLIW also needs forwarding logic and, with more parallel execution units, this forwarding logic becomes more complex. But according to (FISHER; FARABOSHI; YOUNG, 2005), 4-issue forwarding logic is perfectly feasible, even in soft-core implementations. As noted before, in (SCHOEBERL et al., 2011) they presented a VLIW dual-fetch processor optimized for real-time applications named Patmos. Compared to state-of-the-art and commercial VLIW processors, Patmos is relatively simple: dual issue and five stage pipeline. The major difference of Patmos is its memory architecture, which has different cache memories for each segment: stack, heap, instruction, etc. Access to different data caches in Patmos uses different ISA instructions.

The use of different caches for different kinds of data could simplify the WCET analysis. However, the programming model is also affected, particularly the compiler if there are distinct memory instructions to use. Patmos also supports fully predicated instructions to promote the single-path programming paradigm (PUSCHNER, 2005).

A comparison between single and multithreading is not the purpose of this thesis, since both have suitable applications. Our focus is placed on the VLIW approach. The research of real-time computer architectures is challenging and involves many aspects, not all problems were resolved by recent published papers and, to continue researching, a customizable platform is necessary. Besides being predictable, a real-time processor should have a reasonable worst-case performance. State-of-the-art processors are superscalar since they fetch and execute more than one instruction per cycle. VLIW could be used to do so in a real-time platform.

Our first contribution to the state of the art is the investigation of the design space of VLIW processors for hard real-time applications. It is already known that VLIW approaches are more adequate for real-time applications but static instruction scheduling alone is not enough to guarantee determinism. VLIW requires some sort of instruction encoding since parallel operations are scheduled together by the compiler. Instruction encoding could be as simple as adding *nops* to empty operations slots or as complex as CISC encoding with variable operation delimitation. The use of a simple encoding jeopardizes performance since it adds instruction memory overhead and a complex encoding jeopardizes determinism since pre-fetch buffers could be necessary in the pipeline front-end. There are many other design space investigations and they will be described in the following sections. We can also highlight: pipeline dependencies resolution, full predication for 4-issue VLIW processors, branch architecture and the first level of the memory subsystem.

We can also note that the only recent work that directly applies VLIW for real-time systems is Patmos. Its primary focus is the memory architecture and it was used later in the T-CREST multi-core project.

The WCET estimation of Patmos is not clearly described since it uses a prototype of aiT [1] proprietary software. We make a second contribution since every hardware aspect we used in the processor is clearly described in the WCET analysis tool.

4.4 DESIGN DECISIONS

After we defined VLIW as the target architecture, this section describes our new approaches and answer the questions raised earlier. We also consider important decisions like Instruction Set Architecture (ISA), instruction encoding, compiler and memory architecture, since a processor can not work without them.

4.4.1 Instruction set and encoding

We adopted a subset of the HP VLIW ST231 instruction set with customized special instructions. By using this instruction set, we rely on a vast VLIW documentation, a compiler and cycle accurate simulator. This documentation and tools are presented in Fisher, Faraboshi and Young (2005) covering the VLIW architecture in a way similar to Patterson and Hennessy (2007) covering the MIPS architecture. Both compiler and simulator are called VEX – *VLIW EXample* [2]. They are proprietary and their source code is not available but we can use them as a baseline since the VLIW approach of these tools does not consider real-time aspects and WCET analysis. The VEX compiler is based on the Multiflow C compiler with global trace instruction scheduling and it is highly optimized for VLIW processors.

4.4.1.1 Instruction bundle encoding

In a VLIW machine, an instruction or instruction bundle is comprised of operations or syllables. Considering a 4-issue fetch unit, an instruction can be composed of one, two, three, four operations or many

[1] http://www.absint.com/ait/
[2] http://www.hpl.hp.com/downloads/vex/

operations depending of the architecture width. The operations are always dispatched in fixed slots generating a static instruction scheduling. Operations are typically RISC instructions and instruction bundles are sets of operations which are executed in parallel by the processor.

In terms of instruction encoding, we could have:

- *Simple nop encoding*: *nops* – no operation – are scheduled in empty execution slots when the compiler cannot find suitable parallelizable operations. This is the most simple instruction bundle encoding but greatly increases the memory overhead since wide fetch machines could have lots of *nops* in unparallelizable code.

- *Fixed-overhead encoding*: Bit masks are pretended to all instructions to specify the mapping between parts of the instructions and execution slots. This is a simple approach but shorter instructions will have more overhead due to the prepending mask bits. This also increases the memory overhead for wide machines with unparallelizable code.

- *Distributed encoding*: It is a variable-overhead method of encoding VLIW instructions by explicitly inserting a stop bit to delimit the end of the current instruction. It is not necessary to encode nops when there are not parallel operations to execute. The overhead of this encoding is only one bit per operation.

- *Template-based encoding*: A number of bits per instruction (or group of operations) designates the types of operations in the instruction and the boundaries between parallelizable operations. It is similar to fixed-overhead encoding but templates indicate chains and sequences. The limited set of legal template values needs additional compiler support, and some *nop* instructions may still to be encoded when the available templates do not match the generated code.

We adopted distributed encoding because it is compatible with the HP VLIW ST231 ISA, has low overhead on the instruction memory, is a simple and efficient decoding strategy and does not need additional compiler support. Low overhead on the instruction memory is aligned to the thesis objective since our focus is determinism and a complex and large memory subsystem jeopardizes the WCET analyses. An efficient decoder is important because VLIW have high bandwidth in the pipeline front-end and if pre-fetch buffers are required, they will jeopardize determinism as well. The use of a simple but efficient encoding benefits the compiler back-end support. Compiler support is not covered in this work but a new code generator back-end for LLVM (LATTNER; ADVE, 2004) was implemented in parallel with this thesis development.

The adopted distributed encoding strategy is illustrated in Figure 8. This Figure shows a 256-bit cache line being decoded by a 4-wide fetch unit. Operations are 32-bit or 4 bytes long and each slot (S_0 to S_3) receives an instruction bundle at every cycle according to their stop bits. Figure 8 also shows how we avoid additional fetch complexity not allowing instructions to be encoded in different cache lines. The latest 32-bit operation of a cache line must always have a stop bit. Erroneous instruction encoding generates a hardware error.

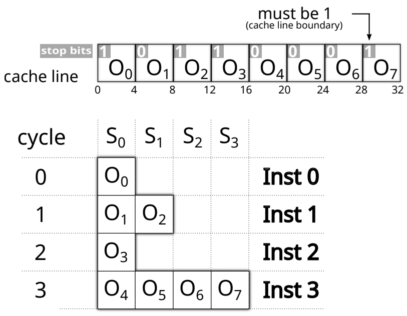

Figure 8 – Instruction bundle decoding example.

4.4.1.2 Operation encoding

Table 10 presents a summary of the VLIW ST231 ISA operation encoding. The encoding is well defined: the format field is used to identify different classes between integer, special, memory or control flow operations. The special class is used for selecting instructions and other special instructions like break point and long immediates. The opcode field specifies the operation type considering the format classes. Bits 20..12 are assigned to short immediate, destination or operand 2 registers depending on the operation format and opcode. Bits 11..6 are assigned to operand 2 or destination registers and it also depends on the format and opcode. Bits 5..0 are only assigned to register operand 1. More details of the operation encoding can be found in the ***ST231 Core and Instruction Set Architecture Manual*** [3].

Table 10 – Operation encoding.

Bits	Function	Description
31	Stop bit	Determines bundle ending
30	Full-predication	Not supported by ST-231, used for full predication extension
29..28	Format	Operation Format: integer, special, memory or control flow classes
27..21	Opcode	Operation opcode within format class
20..12	Immediate/dest/src2	Short immediate, destination register or operand 2 register
11..6	dest/scr2	Register operand 2 or destination
5..0	src1	Register operand 1

The VLIW ST231 ISA supports only partial predication: the hardware supports select instructions in which one value is selected between two input parameters and one binary selector (predicate). We believe that partial predication alone generates too much instruction

[3] http://lipforge.ens-lyon.fr/docman/view.php/53/26/st231arch.pdf

memory and compiler overhead. We extended predication capabilities adding full predication support where blocks of instructions can be turned off. This feature will be described in Section 4.4.3.

Like the ST231 ISA, we support 9-bit immediate data (encoded in a single operation) or long 32-bit immediate data (using 64-bit). Long immediates can be on the left or right of an arithmetic operation using the special long immediate operation.

Branches and jumps use a 23-bit PC (Program Counter) offset. The integer offset is encoded using bits 22..0 when using the control-flow instruction format. Function calls to 32-bit addresses are supported by a register indirect branch and link instruction.

There is a complete set of compare instructions and they are flexible: the destination can be a predicate or a general purpose register and the source can be only register or register/immediate. A total of 64 32-bit registers could be encoded (*$r*) and 8 1-bit predicates (*$br*). The number of predicates can only be increased using a clustered design because there are no available encoding bits. The complete operation list is available in Appendix A.

4.4.2 Processor pipeline

We propose a VLIW design featuring with full forwarded execution stage units to prevent additional stalls from Read after Write (RAW) data dependencies. It means that any calculation performed by any unit is forwarded to the unit which needs the data. Figure 9 shows an overview of the adopted pipeline design. Forwarding logic data paths and the register file are omitted. The black arrows indicate the data path between the pipeline stages and the execution units. A 64 32-bit register file and a 8 1-bit predicate register file are common to all execution units.

Our design has 5 stages:

- "F" – *Cache Buffer*: This stage is connected directly to the cache instruction memory and it represents the pipeline front-end. It controls the cache memory providing its address and receives stall

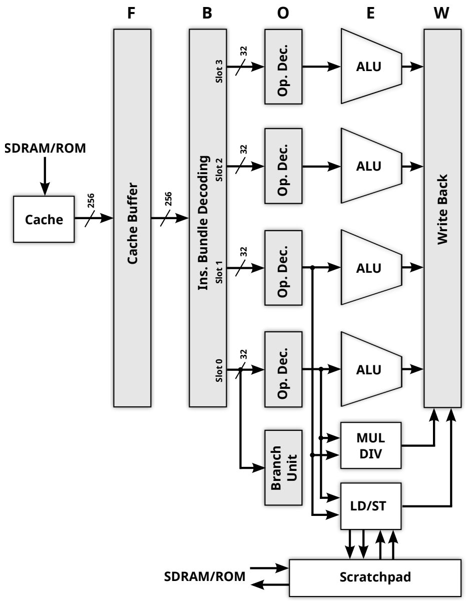

Figure 9 – VLIW processor overview.

signals from the cache or from others pipeline states. It also stores an entire cache line buffering it to the next pipeline stage.

- "B" – *Instruction Bundle Decoding*: This stage receives a cache line and decodes instruction bundles into 32-bit operations. Instruction bundles are encoded using distributed encoding as illustrated in Figure 8.

- "O" – *Operation Decoding*: This stage is composed of four operation decoders and one branch unit. Each operation decoder takes a 32-bit operation and decodes it to control signals. These control signals program the execution units in the next pipeline stage. During this stage, data from the register files for all four execution slots are also read. Branches and other control flow operations are also decoded and pre-executed in this stage in an individual unit connected to the *Slot 0* data path.

- "E" – *Execution stage*: This stage is responsible for executing all operations. It has four identical Arithmetic and Logic Units (ALU), one multiplication and division unit and one unit responsible for memory transactions (LD/ST). The multiplication and division unit is connected to *Slot 0* and *Slot 1* data paths as well as the memory unit. This means that this type of operation must be scheduled only in those specific slots.
- "W" – *Write Back*: This stage is responsible for writing the execution stage results to the register file.

There are two basic types of data dependencies in a VLIW design that must resolved:

1. *data dependencies between parallel operations (spatial parallelism)*: dependencies between operations in an instruction bundle have to be resolved by the compiler: if a write and a read occur from the same register, the current computation uses the old value.
2. *data dependencies between pipeline stages (time parallelism)*: dependencies between pipeline stages usually generates Read-After-Write (RAW) data dependencies. These dependencies are solved by hardware with forward paths or by an interlocking mechanism. No arithmetic instruction executed by the ALUs causes any pipeline stall. However, memory, multiplication and division operations are resolved by a separate hardware mechanism.

Complex operations such as memory, multiplication and division require more pipeline sub-stages (typically called phases (SHEN et al., 2005)) to carry out their calculations. The memory subsystem requires one stage to perform the address calculation, another one to access the memory and one more to perform masks to allow byte, half-word and word access. Multiplication is relatively complex and it needs also more stages to perform a signed or an unsigned 32-bit multiplication. Division is a very slow operation and it requires lots of stages to perform a signed or unsigned division. The number of required sub-stages or additional cycles depends on the hardware/FGPA technology and it will be described in Chapter 5. A 64-bit multiplication result is supported using specific ST-231 ISA operations. However, the ST-231 ISA is not compatible with a full division instruction but only with division steps to enhance software division capabilities. Although division is a slow operation, we added full division support augmenting the ISA with four new instructions for integer signed and unsigned division supporting quotient and remainder results in hardware.

Figure 10 displays a detailed view of the execution units. We can see that complex operations such as memory, multiplication and division require more pipeline sub-stages to carry out their calculations as described in previous paragraph.

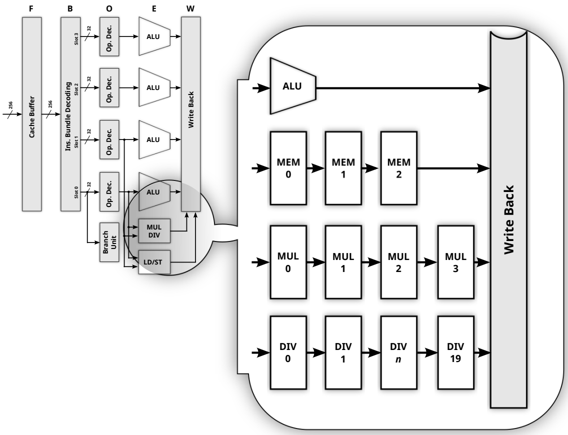

Figure 10 – Detail of execution units stages.

The complex multistage operations increase the pipeline latency during the execution stage. The quantification of those latencies must be linked with the implementing FPGA technology. However, we can notice that those operations will not be as fast as the arithmetic ones. To meet real-time capabilities they should be executed in order (a typical superscalar commercial processor usually allows out-of-order execution inside the execution stage) and so, there will be pipeline stalls during their calculations. These stalls could be implemented by the compiler, like the VLIW ST231, or implemented by hardware. In our design, we decided to resolve all data dependencies in hardware.

Another typical approach used in VLIW machines is to duplicate execution units, allowing multiple operations per instruction, to diminish execution stage latencies. The ST-231, for instance, has 2 independent multiplication units which allow up to two multiplications per bundle and the memory interface, however, permits only one memory operation per bundle. We go further and extend the memory interface to allow two memory operations as well and extended the ISA with additional operations supporting hardware division.

4.4.3 Predication

The HP VLIW ST231 ISA natively supports partial predication and, in order to support this feature, we only need to implement two select instructions: *slct* and *slctf*. The first one selects when the condition is true and the other one when it is false. Both instructions have the following semantics:

- *slct $r_dest = $br_x, $r_x, $r_y* where register $*r_dest* receives the value of $*r_x* if the predicate register $*br_x* is true or the value of $*r_y* if it is false.
- *slctf $r_dest = $br_x, $r_x, $r_y* where register $*r_dest* receives the value of $*r_x* if the predicate register $*br_x* is false or the value of $*r_y* if it is true.

To extend the real-time capabilities of the processor, we propose a low-overhead full predication mechanism. Considering a 4-issue machine, a generic full predication support adds considerable levels of logic to critical paths and degrades the processor clock speed because predicate operands must be forwarded during all operation execution. Predicate operands with 3-bits must also be encoded in every instruction what is not possible considering the HP VLIW ST231 ISA since there is only one unused bit.

The low overhead full predication mechanism consists in:

- Only one predicate register is used ($\$br4$);
- Only one bit is used for predication encoding (bit 30);
- Every operation in an instruction bundle can be individually predicated using this fixed predicate register;
- Two operating modes: simple true mode where operations are executed only if the predicate register value is true; complete true/false mode where operations are executed on true or false. The complete mode is activated/deactivated with two new ISA operations *paron/paroff*;
- Control flow operations cannot be predicated and they reset to complete mode. Branch instructions are already conditionally executed by predicate registers.

The use of only one fixed predicate register reduces the register file overhead since only one additional read port is necessary. If any predicate register could be used, four additional ports would be necessary to read 4 additional predicates per cycle.

The predication system is activated primarily on using bit 30. In the simple true mode, if an operation is encoded with bit 30 set, it will only be executed if the value of predicate register $\$br4$ is true. This feature is illustrated in Table 11. The comparison operation that sets the predicate $\$br4$ is in Line 4. Since there is a RAW dependency between the *cmpgt* and the predicated *mull* and it is not possible to forward the value of $\$br4$ while it is still been calculated, predicated instructions must wait at least one cycle before the predicate is correctly read. This RAW dependency could be either resolved by hardware installing a stall cycle or by software including other instruction before the predicated code snippet.

The complete true/false mode is activated by beginning the predicated region with *paron* operation and ending with *paroff*. This instructs the processor that the following operations are in the complete

Table 11 – Example of simple true mode predication system.

if (x > y)		
z = x * y;		
h = x + y;		
4	**cmpgt** $br4 = $r1, $r2	
5	;	
6	**nop**	nop or useful instruction
7	;	
8	**(p) mull** $r4 = $r1, $r2	Bit 30 is set, executed if $br4 = true
9	**add** $r7 = $r1, $r2	Bit 30 is unset, always executed
10	;	

true/false mode. The operation's bit 30 is used with inverted logic: when set, the operation is executed only if $\$br4$ is true; when unset the operation is executed only if $\$br4$ is false. This mode allows the compiler to parallelize distinct if-then-else paths like the example in Table 12 without a single branch instruction.

Both predication modes are powerful and they allow the compiler to avoid control flow instructions. This reduces the amount of basic blocks and enhances the instruction scheduling capabilities. However, nested if-then-else structures cannot be fully predicated, since there is only a single predicate register capable of disabling instruction execution. This limitation can be overcome by using the full low overhead and the partial predication together as exemplified by Table 13. We can see in this example two nested if-then-else structures without using any branch operation.

4.4.4 Branch prediction

Branch architecture has always generated discussion and research in computer architectures. Generally speaking, branches or other type

Table 12 – Example of complete mode predication system.

if (x > -1)	
z = a + b;	
else	
z = a * b;	
cmpgt $br4 = $r9, -1	
;	
paron	Activate complete mode
add $r50 = $r0, 0	
;	
(p) add $r9 = $r9, $r10	Bit 30 = false, execute if $br4 = false
(p) mull $r9 = $r9, $r10	Bit 30 = true, execute if $br4 = true
;	
par off	Deactivate complete mode
;	
goto $r63	

of instruction that breaks the pipeline streaming will always degrade processor performance. One solution to this problem is branch prediction. High-end processors adapt themselves by remembering the previously taken direction of each conditional branch. This is the solution of general-purpose processors: they use dynamic predictors implemented in their hardware. However, dynamic branch predictors, as well as every history-based component, greatly jeopardizes WCET analysis because they use structures called Branch Target Buffers (BTB). BTBs are essentially full-associative caches which are very difficult to model during WCET analyses and they can generate timing anomalies or jeopardize timing composability. BTBs have another problem in the eviction policy because real-time systems require Least Recent Use (LRU) policy which is nearly implemented by the pseudo-LRU in

Table 13 – Example of nested if-then-else with full and partial predication systems.

```
if ( data[mid].key == x )
    up = low - 1;
else
        if ( data[mid].key > x )
            up = mid - 1;
        else
             low = mid + 1;
```

cmpeq $br4, $r18, $r16	(if data[] == x)
;	
paron	
;	
(p) add $r10 = $r9, -1	up = low - 1 if $br4 = true
(p) cmple $br5, $r17, $r16	(if data[] > x) if $br4 = false
(p) add $r50 = $r11, -1	temp0 = mid - 1 if $br4 = false
(p) add $r51 = $r11, 1	temp1 = mid + 1 if $br4 = false
;	
(p) slct $r10 = $br5, $r50, $r10	write on up if $br4 = false
(p) slctf $r9 = $br5, $r51, $r9	write on low if $br4 = false
paroff	

hardware.

Real-time researchers have recommended the use of predication without branch prediction or the use of static branch prediction. In Section 4.4.3 that described our full predication strategy, we showed an example of an if-then-else structure converted entirely to predication. Besides keeping the pipeline at maximum stream, predication is more suitable for balanced paths. We can see in Table 13 that the "then" path is much smaller than the "else" path but both are always executed regardless of the “if” condition. This situation is worse if the architecture supports only partial predication because high latency instructions cannot be deactivated, only their results could be selected at the end of the conditional code.

Observing these situations, we propose a real-time architecture that supports our low-overhead full-predication mechanism described in Section 4.4.3 and a static branch prediction. With this hybrid approach, WCET-oriented compiler optimizations could be used to select predication, select the appropriate direction for static branch prediction or use both techniques for nested if-then-else structures. Branch prediction benefits the performance of loop structures as well. Our static branch prediction methodology is not a new one. The contribution of this section to the state of the art is to show design methodologies and how this technique is analyzed in the WCET tool. WCET analysis technique for branch prediction is described in Chapter 6 considering the branch behavior exposed below.

The default behavior of our branch unit is the use of a static “not taken” branch direction where the processor front-end keeps always decoding linear addresses. Figure 11 illustrates the “not taken” pipeline behavior. We can see that there is no overhead when a branch is not taken. Pipeline stages follow the same representation as described by Figure 9 where “F” represents “Cache Buffer”, “B” represents the “Bundle decoding”, “O” represents the “Operation decoding”, “E” represents the execution stage and “W” the Write back.

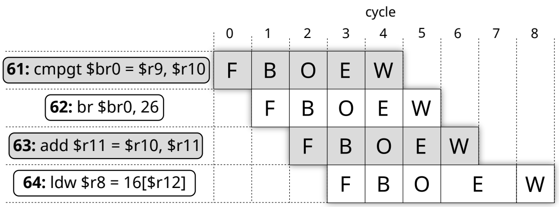

Figure 11 – Pipeline behavior of a “not taken” branch.

Since the HP VLIW ST231 ISA does not have any static branch prediction support, we added a new special instruction that allows the compiler to indicate whether a given path is more likely to be executed. That instruction is called *branch preload* or *preld* at ISA level. When a branch is more likely to be taken, a *preld* instruction can be scheduled in a previous bundle. The scheduling overhead of the *preld* instruction/operation is almost zero when it is scheduled in a free slot in an existing bundle. The common position of a *preld* operation is 2 cycles before a branch operation. The *preld* instruction works by anticipating the calculation of the branch target address, forcing the branch ahead to follow a *taken* behavior. In this way, the *preld* instruction emulates the existence of an entry in a branch target buffer (BTB) relative to the next branch. If such a *preld* instruction does not exist for a determined branch, it has a *not taken* or *fall-through* behavior as described earlier. Figure 12 illustrates the simplified behavior of the pipeline during a "taken" branch with a *preld* instruction. At cycle 3, the *preld* instructs the fetch unit to prepare for a jump and the new cache data is ready at cycle 4. Following the pipeline diagram, if the branch condition is true, the cache transaction is already performed and the processor continues to execute at cycle 5. As we can see, placing a *preld* operation enables the compiler to change the direction of the branch then decreasing the branch penalty. Figure 12 is a simplification of the pipeline behavior and we can see that a "taken branch" has 2 penalty cycles ("lost" arrows). Considering a data dependency between instruction 61 (*cmpgt*) and 62 (*br*) which generates a *stall*, the "taken" branch overhead is only 1 cycle.

Direct control flow operations like *calls*, *gotos* and branches without *preload* always take 4 cycles. This is the time for the architecture realization to flush "Operation decoding – O" and "Execution – E" pipeline stages, reprogram the cache and fetch the target address. This behavior is illustrated by Figure 13. At cycle 5, the branch condition is evaluated and if it is true, the fetch unit prepares for a jump and the new cache data is ready at cycle 6. As the default fetch behavior is keeping fetching linear instructions, the job done at cycles 3 and 4 is lost. The partially executed instruction (*ldw $r23 = 10[$r10]*) is

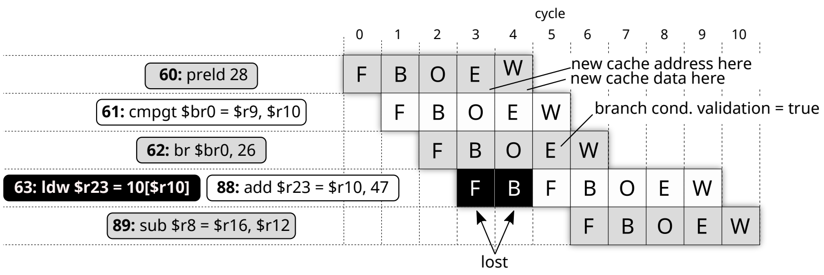

Figure 12 – Pipeline behavior of a "taken" directed branch.

flushed and the new instruction at the new address begins its execution at cycle 7. Figure 13 illustrates a branch instruction, but *calls* and *gotos* follow the same behavior.

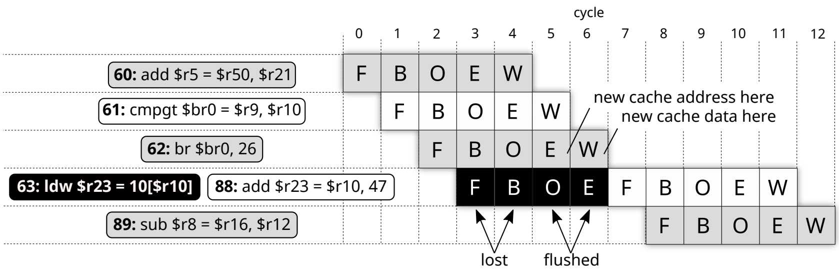

Figure 13 – Pipeline behavior of a "taken" branch without direction change or other direct control flow operations.

As we can see, the timing of branch operations is slightly different because it depends on the branch prediction direction and the condition validation. A "fall-through branch" does not have any overhead if there is a *not taken* direction. A "taken branch" has 1 cycle overhead if there is a *taken* direction. Although, if there is misprediction when a branch is "not taken", the fetch acts like a direct control flow operations and it takes 4 cycles to execute as illustrated by Figure 13. The worst effect happens when there is a "taken" direction (with *preld* operation) but the correct path is not to take the branch. The recovery process is illustrated by Figure 14. At cycle 3, the ***preld*** instructs the fetch unit to fetch the new cache address. The branch evaluation is false, at cycle 5, and the recovery process begins at cycle 6 using the old linear cache address. The cache data is ready at cycle 7 and the execution of the correct instructions resumes at cycle 8.

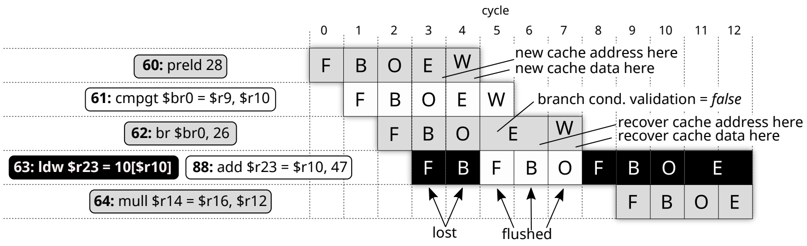

Figure 14 – Pipeline behavior of a mispredicted direction.

The timing of the branch unit depends on the processor front-end implementation and will be described in the next chapter. Commercial processors could easily be faster. Nevertheless, the most import characteristic is the *very well defined timing behavior* of the control flow system, particularly for real-time applications. Besides we can see the “taken” branch direction is fairly worse than the “not taken” one. The support of both directions is important and it brings considerable results. Considering a loop which iterates 100 times: if there is a “not taken” direction, it will have about 400 cycles of overhead because it will mispredict the branch every iteration. With a taken direction with only 1 cycle overhead per iteration, there are only about 100 cycles of overhead.

4.4.5 First level of the memory subsystem

A fast and low-latency memory subsystem is important for any computer architecture. High-end processors typically use various levels of associative caches to provide fast access to the memory subsystem.

However, the use of this approach is not recommended for real-time systems as discussed in Chapter 2 because it jeopardizes WCET analyses.

Recent real-time literature has recommended the use of direct-mapped caches, scratchpad memories or an approach used by Patmos and introduced in Schoeberl (2009a). This approach proposes the use of different caches for different data areas. It is not the purpose of our work to propose a new real-time memory architecture. For the first level of the memory subsystem, we use:

- *instruction memory*: direct-mapped cache.
- *data memory*: scratchpad.

Considering the memory subsystem, we contribute with two investigations. First, we investigate how to connect this cache to VLIW pipeline front-end and mainly how the latency of this memory is considered during the WCET analysis. Second, the use of a typical memory access instruction with base plus offset addressing mode introduces additional latency on the memory instructions, specially for a VLIW processor because they have several parallel data paths. We provide a performance study about using single or multi-port scratchpad access to diminish the memory latency.

4.4.5.1 Instruction cache

The memory interface connected to the pipeline front-end must supply enough data to keep the 4-issue pipeline busy. In VLIW terminology, instructions are composed of operations and our four-issue machine executes up to four operations per cycle forming a 128-bit instruction word.

Since the main focus of this work is not the memory subsystem, any instruction-memory configuration could be used including a scratchpad instruction memory or the method cache presented by PATMOS (SCHOEBERL et al., 2011). We used a simple 32-lines of 256-bit block directed-mapped cache in order to provide sufficient bandwidth to the pipeline, respect real-time capabilities and to simplify the worst-case execution time analysis. The most important contribution is the cache WCET analysis described in Chapter 6.

4.4.5.2 Data memory interface

The data memory interface was designed to access a scratchpad memory since we are not considering a complex memory architecture in order to keep a deterministic design.

We considered a unified 32-bit memory address space. Addresses 0 to 0x3FFFFFFF (1GiB) are mapped to the instruction memory, 0x40000000 to 0x403FFFFF (4MiB) to the scratchpad memory and 0x80000000 to 0xFFFFFFFF (2 GiB) to external SDRAM memory. Address space between scratchpad and external SDRAM could be used for IO. Data movement between memories (ROM/scratchpad and SD-RAM/ scratchpad) is planned to be done via DMA (Direct Memory Access) operations and direct access using memory instructions is only available for the scratchpad memory. A real-time memory controller with a DMA unit is not in the scope of this thesis but it is fully described in the work of Reineke et al. (2011).

We support base plus offset memory addressing with the following semantics:

- *load $rx = immediate_offset [register_base]*: loads data from memory with address [register_base + immediate_offset] to register $*rx*.
- *store immediate_offset [register_base] = $rx*: stores the value of register $*rx* to memory with address [register_base + immediate_offset].

We target an access to the scratchpad of one cycle, but the entire memory instruction transaction is actually a multi-stage operation with more cycles in order to support base plus offset and byte access modes. Memory instructions specification generally determines 2 stages to perform a memory transaction. One stage is necessary to calculate the address and another to read the data memory. If these two stages are imposed to all processor instructions (likewise the "M" stage in MIPS), there is a pipeline fragmentation since one stage is only necessary for memory instructions and it is unused for arithmetic instructions. In our design, we prefer not to add any pipeline fragmentation to arithmetic instructions, so we do not have any unused pipeline stage. Multi-cycle instructions, however, could violate Read-after-Write dependencies. The most common approach for VLIW processors is to resolve this by the compiler adding *nops* and enhancing the instruction scheduling by advancing multicycle operations. We propose stalling the pipeline but allowing two memory operations per bundle using a dual port scratchpad. Figure 15 shows a diagram of the memory interface. ROM and SDRAM accesses are planned to be via DMA operations.

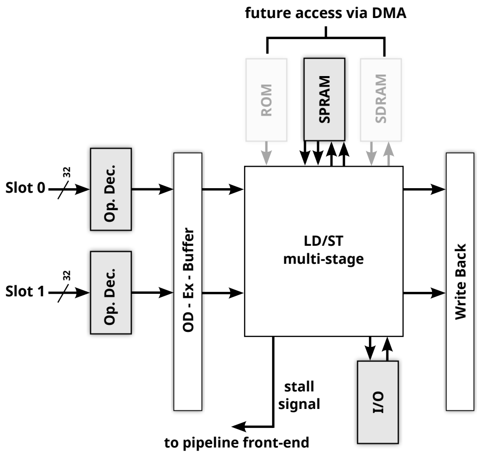

Figure 15 – Memory unit logic diagram.

Using dual-port access to the scratchpad memory does not add any WCET complexity because both memory accesses are executed at the same time. The benefit on performance is described in Chapter 7.

4.4.6 Compiler support

In order to have a compiler for the architecture and to provide a customizable environment to research real-time compiler capabilities, a new code generator back-end for LLVM (LATTNER; ADVE, 2004) was implemented and it makes part of a different thesis. In this Section we only provide an overview of the back-end.

LLVM is a compiler infrastructure targeted to be modular and extensible. As compiler front-end we used Clang[4], which supports C-like languages.

Since the processor has a substantial number of registers, it is possible to reduce the access to memory data in a compiled program. A substantial source of memory operations are the accesses to local variables that are allocated in the procedure stack. Most of the variables that are originally allocated in the stack can be promoted to permanent register variables. This promotion of stack to register variables is done by a pass called *PromoteMemoryToRegisterPass*, which already exists in LLVM, and is enabled by the back-end. The execution of this pass reduces significantly the number of memory access of programs, due to the fact that the Clang front-end always allocates local variables in the stack.

Considering instruction ordering to expose ILP to the processor, we used the default VLIW scheduler. This algorithm performs top-down list scheduling locally in each basic block. As input data, the scheduling process uses a simplified description of the processor resources (represented as a hazard recognizer) and a machine code basic block to be scheduled. Instruction scheduling is performed before the register allocation process, so the output of this phase is not a complete machine code representation. There is no global instruction scheduler for VLIW architectures in LLVM. Global instruction schedulers can explore ILP across basic block boundaries, resulting in better performance when compared with the local ones. Examples of global schedulers are Superblock Scheduling (HWU et al., 1993) and Trace

[4] http://clang.llvm.org/

Scheduling (FISHER; MEMBER, 1981).

The instruction or operation scheduling only affects the instruction ordering before the register allocation. After the register allocation, with all auxiliary code generated (function prologs and epilogs and register spilling code), it is necessary to group operations in instruction bundles. This process is done in a pass implemented in the back-end called *packetizer*. The packetizer pass operates individually on each basic block, analyzing data dependencies between operations to decide whether a operation placement in a determined bundle does not violate the program semantics. The following data dependencies are considered by the packetizer:

- Data: Often called true data dependency, this dependency occurs if an operation sets a register that is read by another operation. If such dependency exists, the two operations must be placed in different bundles. Table 14 shows an example of an assembly code where there is a data dependency between two operations: *shru* writes in *$r18* and *sth* immediately reads it.

Table 14 – Example of data dependency between operations.

add $r11 = $r19, $r10	$r11 = $r19 + $r10
shru $r18 = $r18, 11	$r18 = $r18 << 11
add $r9 = $r9, 1	$r9 = $r9 + 1
;	
sth 2[$r10] = $r18	mem[$r10 + 2] = $r18
;	

- Output: This type of dependency exists between operations that set the same register. If such dependency exists, the two operations must also be placed in different bundles, considering the original order.

- Anti-dependency: This type of dependency occurs when an operation sets a register that is read by a previous operation. Such type of dependency is ignored by the packetizer because it does not affect the semantics of a program when executed in the processor. Ignoring anti-dependencies significantly improves the ILP of programs. Table 15 shows an example of an assembly code where there is an anti-dependency between two operations: *$r10* is read by *sth* and immediately written by *shr*. Since *$r10* is not updated in the same cycle it is read by *shr*, this dependency can be ignored.

Table 15 – Operation packing considering anti-dependencies, and ignoring them.

With anti-dependencies	Without anti-dependencies	Description
sth 0[$r10] = $r19	**sth** 0[$r10] = $r19	mem[$r10] = $r19
add $r16 = $r16, 2	**add** $r16 = $r16, 2	$r16 = $r16 + 2
;	**shr** $r10 = $r18, 18	$r10 = $r10 >> 18
shr $r10 = $r18, 18	**add** $r8 = $r8, 1	$r8 = $r8 + 1
add $r8 = $r8, 1	;	
;		

The packetizer pass also assumes some specific situations, like two memory operands per bundle and only one branch/call instruction per bundle.

In order to respect the cache alignment restrictions of the processor, another pass was implemented in the back-end. This pass aligns bundles aiming to always have a stop bit at the end of a cache line. This alignment is done by inserting *nops* (no operation) into previous bundles, to shift a determined misaligned bundle to a new cache line. Table 16 shows a sequence of misaligned bundles before and after the alignment pass. In this example, we consider that the initial address of the first basic block is aligned with a cache line.

Table 16 – Instruction alignment in basic blocks.

Misaligned bundle sequence	Aligned bundle sequence	Description
shl $r16 = $r8, 2	**shl** $r16 = $r8, 2	$r16 = $r8 << 2
;	;	
add $r17 = $r16, $r9	**add** $r17 = $r16, $r9	$r17 = $r16 + $r9
add $r8 = $r8, 1	**add** $r8 = $r8, 1	$r8 = $r8 + 1
;	**nop**	*alignment nop*
//misaligned bundle		;
ldw $r17 = 0[$r17]	**ldw** $r17 = 0[$r17]	$r17 = mem[$r17]
add $r16 = $r16, $r10	**add** $r16 = $r16, $r10	$r16 = $r16 + $r10
;	;	
stw 0[$r16] = $r17	**stw** 0[$r16] = $r17	mem[$r16] = $r17
;	**nop**	*alignment nop*
	;	

LLVM does not support predication in its intermediate representation (IR), although select instructions for partial predication are supported. Partial predication is supported generically by a set of passes that perform *if conversions* and *control flow simplifications*. These transformations are guided by generic cost-benefit functions, which convert conservatively only the obvious and simple cases, e.g. cases where the benefit is certainly high. If an architecture needs aggressive code predication, it must implement its own predication support at the back-end side of the compiler, as done by Jordan, Kim and Krall (2013).

Considering the WCET support, the code generator provides useful data about the low-level structure of the compiled program, such as the control flow graphs and loop bounds, for instance. These data are also a compiler product in conjunction with the compiled program (the object code) and both are used by the WCET tool (Chapter 6) to calculate the WCET of a program.

4.5 SUMMARY

We started this Chapter discussing the challenging task of analyzing real-time systems, specially due to the motivation of obtaining safe and more precise WCET estimations. There are different hardware requirements between general-purpose and real-time systems although both of them need constant performance enhancements.

Regarding our objective to improve performance and determinism, we exposed several processor architecture features that lead to predictable design but also increases the WCET performance. We have secured our low-level requirements focusing on hard real-time systems but also not forgetting the high-level requirements allowing our design to be applicable to modern systems using a modern ISA and a modern compiler back-end.

We discussed the design decisions through an architecture overview. We focus on a real-time processor architecture using a 4-issue VLIW design with 32-bit RISC operations using HP ST231 ISA. We support both types of predication as well as static branch prediction with well-defined behavior. Table 17 presents a synthesis of real-time processor features considering the standard ST231, PATMOS, PRET and the prototype described in this thesis.

Regarding our contributions for real-time processor architectures, we can highlight:

- Low-overhead full-predication system for VLIW processors.
- Detailed branch architecture.
- VHDL design and timing characteristic.
- Complete WCET analysis for the proposed design.
- Design space investigation and evaluation.

[5] ARMv4 supports full predication, but there is no mention in (LIU et al., 2012) about PRET hardware support.

Feature	**ST-231**	**PATMOS**	**PRET**	**This thesis**
ISA	ST-200	MIPS like	ARMv4	ST231 Subset
Instruc. Mem.	Associative Cache	Method Cache	Scratchpad	Direct-Mapped Cache
Data Mem.	Associative Cache	Semantic Caches	Scratchpad	Scratchpad
Predication	Partial	Full	Full [5]	Partial and Full
Pipeline	8-stage	3-stage	5-stage	5-stage
Multi-threading	No	No	Yes	No
Depen. resol.	Compiler	Compiler	No resolution	Hardware
Issue width	4	2	1	4

Table 17 – Pipeline features comparison

In the next chapters, we will focus on the timing of the proposed design and later its evaluation in terms of WCET performance. The timing is defined in Chapter 5, which describes the VHDL implementation of the processor. The implementation is necessary to infer the timing of each module for WCET modeling and estimation. The WCET analysis is described in Chapter 6, including analyses of the instruction cache, pipeline and the worst-case path search. The evaluation of techniques proposed here will be described in Chapter 7, highlighting their performance benefits.

5 VHDL IMPLEMENTATION AND TIMING CHARACTERISTICS

In this chapter, we describe the implementation of our deterministic real-time processor. We describe the timing of each module and their VHDL implementation which are necessary for WCET modeling and estimation.

The processor designed in this thesis is a 4-wide fetch VLIW microprocessor with 32-bit RISC operations. In order to assess the design, we made a VHDL FPGA implementation on an Altera Cyclone IV GX (EP4CGX150DF31C7) in a DE2i-150 development board. As any soft core, the maximum reachable clock speed relies on the FPGA device family, quality of the synthesis tool and VHDL programming quality. We target and reach a clock speed of 100Mhz. In terms of comparison, Altera's own optimized soft processor (NIOS II – single fetch) runs at 160MHZ on Cyclone IV GX.

We repeat in Figure 16 an overview of the proposed VLIW pipeline previously depicted in Figure 9 . Memories and forwarding logic data paths are omitted and the black arrows show the data path between the pipeline stages and the execution units. The core has four arithmetic units, two multiplication units, one division unit, one memory unit and one branch unit. With this data path, up to four arithmetic operations can be executed in parallel, one branch operation with up to three arithmetic operations, two multiplications with two arithmetic operations, one memory with up to three arithmetic operations or any other combination respecting the data path.

5.1 INSTRUCTION CACHE

The VHDL cache implementation has two main data interfaces as depicted by Figure 17. The *mem_** interface is used to connect the cache to a VHDL ROM (Read Only Memory) component which stores the compiled program. This ROM could also be changed to an external flash memory or a SDRAM with fewer modifications. The *stall_out*, *cache_data_rdy*, *abort* and *data_out* signals compose the second main

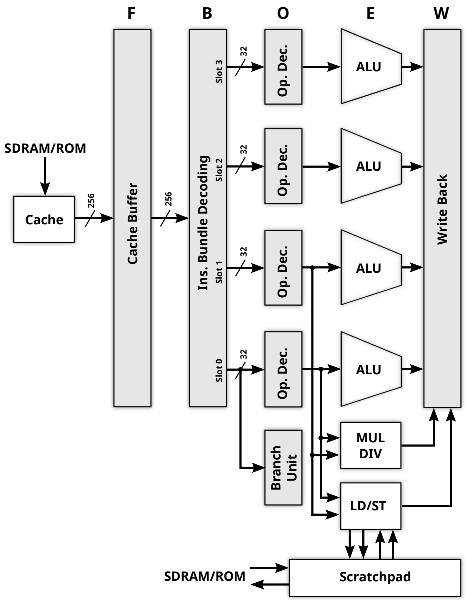

Figure 16 – VLIW processor overview (reproduced from Figure 9).

interface and they make the synchronous connection to the pipeline front-end.

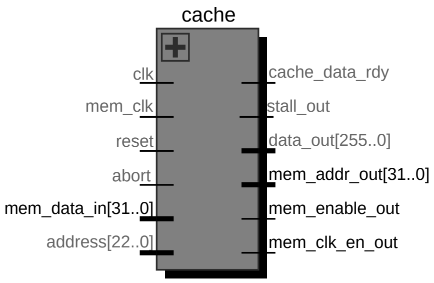

Figure 17 – Cache VHDL component.

The input *address* signal is controlled by the pipeline front-end and instructs the cache to provide the necessary 256-bit data (an entire cache line) at that address. Internally the address is bit-divided according to Table 18.

Table 18 – Cache address fields

Field	Address bits	Description
Block offset	[0..2]	3 bits to index each 32-bit word
Index	[3..7]	5 bits to index each cache line
Tag	[8..22]	15 bits tag memory addresses

Cache tags and memory blocks are stored using two SRAM (Static Random Access Memory) components. The first is a 32x256 bit matrix (32 lines of 256 bits) to store the memory blocks (8 32-bit words). The 256-bit cache line width is the maximum width allowed by the Altera Synthesis to infer RAM blocks. The second is a 32x15 bit matrix to store the tag information for each cached block. A 32x1 bit register matrix is also necessary to store the validity or live bit. These bits are used to mark a cache line as valid or invalid.

The cache controller and behavior are illustrated by the Finite State Machine (FSM) in Figure 18. The cache is constructed using synchronous components (registers and RAMs) and they are synchronized by the rising edge of the processor main clock. Each FSM edge transition is indeed a clock cycle.

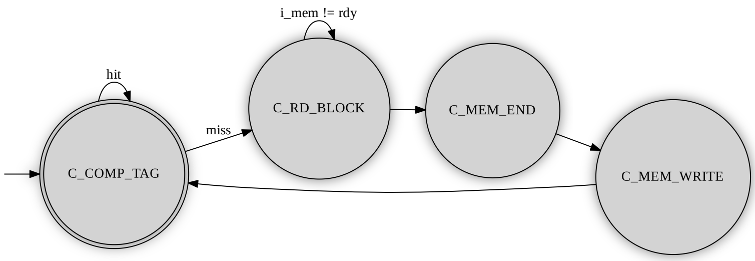

Figure 18 – Cache states and behavior.

The first and the main state of the cache is C_COMP_TAG. In this state, hits or misses are checked dividing the input address into index and tag and performing reads in tag and memory blocks SRAMs. The valid bit register matrix is also read in this state. There is a hit if the valid bit is true and the requested tag is the same as the stored one. Since all information (data blocks, tag and valid bit) is simultaneously accessed, a 256-bit cache line are immediately available. Cached data blocks are accessed like a simple synchronous SRAM: data is ready one cycle after the address is requested.

In case of a miss, a ROM transaction is necessary. The state changes to C_RD_BLOCK and a stall signal is set and it stalls the pipeline front-end. The latency of this state depends on the speed of the instruction memory. When C_RD_BLOCK is reached, this state initializes the ROM controller FSM as depicted in Figure 19. As soon as the cache controller starts the ROM controller, it accesses the ROM memory in state ACC_1 manipulating control bits and generating the correct ROM addresses. Since the ROM is also synchronous, nine cycles are necessary to read eight words. Those words are also concatenated to build a 256-bit word requiring another cycle to register the result. The entire ROM transaction takes ten instruction memory cycles.

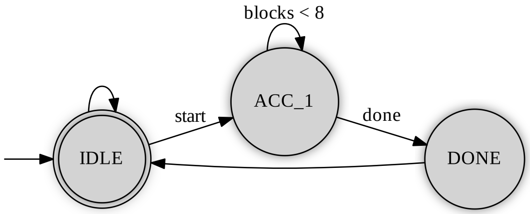

Figure 19 – ROM controller behavior.

When the ROM controller finishes, the cache FSM proceeds to C_ MEM_END. This state prepares the internal memories to receive the new data and tag. After that, a new cache line is stored, data is available and the stall signal is unset. If the ROM is clocked at the same the processor main clock, a cache miss takes 15 cycles: 10 for ROM transaction plus 5 for internal controllers, SRAM transactions and registers.

5.2 PIPELINE FRONT-END

The pipeline front-end fetches an entire 256-bit cache line from the instruction cache or eight 32-bit operations. Considering a 4-issue fetch unit, an instruction can have one, two, three or four operations. The operations are always dispatched in fixed slots (S0 to S3), as shown in Figure 16. Instructions are encoded using distributed encoding. It is a variable-overhead method of encoding VLIW instructions by explicitly inserting a stop bit (Bit 31) to delimit the end of the current instruction. It is not necessary to encode 32-bit nops when there are not four parallel operations to execute.

The front-end is built using two pipeline stages: the “Cache buffer” and “Instruction bundle decoding”. The “Cache Buffer” stage controls the instruction cache address and fetches an entire cache line. After a cache line is fetched, “Ins. Bundle Decoding” detects the stop bits and dispatches up to four operations in four execution slots. Due to distributed encoding, there is no “nop” overhead if the compiler cannot find four parallel operations. There is a basic alignment restriction to make bundle decoding faster and more predictable: one instruction cannot be in different cache lines, so the remaining 32-bit operation of a cache line must always have a stop bit. Figure 8 shows an example of instructions of a cache line being decoded and dispatched.

The two pipeline stages, “Instruction bundle decoding” and “Cache buffer”, are depicted in Figures 20 and 21, respectively. In the VHDL component hierarchy, they are part of a large “Fetch” component.

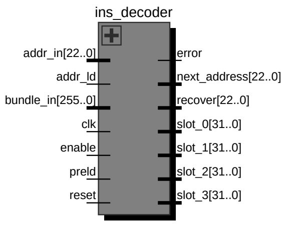

Figure 20 – Instruction bundle decoder VHDL component.

The instruction decoder is in charge of decoding instruction into 32-bit operations inspecting the stop-bits. It receives a full previous registered cache line and its address and divides it into operations as described in Figure 8. Empty slots are reset. The instruction decoder is a synchronous component and is capable of decoding one instruction per cycle. This decoder must also support synchronous load to support control flow instructions since they can move to any address (any cache word offset).

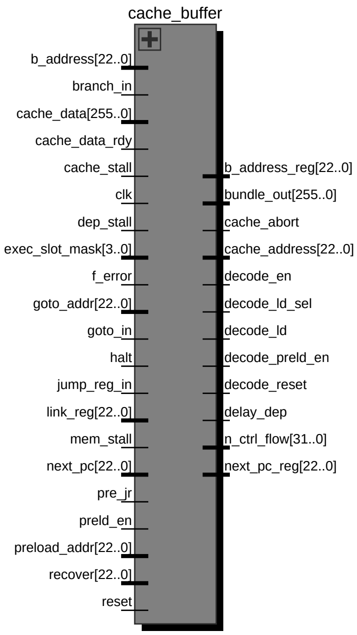

Figure 21 – Cache buffer component.

The “Cache buffer” component, depicted in Figure 21, is a more complex component since it deals with all stall sources like cache misses, pipeline dependency resolutions and stalls coming from the execution stage. Since this unit controls the pipeline front-end, it receives all stall signals and stops the pipeline while their signals are valid. It must also generate the correct signals and update the cache address when control flow instructions are executed.

Figure 22 shows the behavior/controller of the pipeline front-end. The main state is B_L (Bundle Loading), where instructions are decoded and executed in stream mode. In this state, stall signals are monitored. Stalls make the FSM to progress to stall states like STALL and EX_STALL. The first one is a cache stall and the second one is a stall generated by the execution unit.

Different types of control flow instructions are controlled by different states: BRANCHIN for conditional branches, GOTOIN for direct jumps (the address is encoded in the operation) and JUMP_REGIN for indirect jumps (the address comes from the register file). After these transitions, the FSM progresses to ENC_LOAD where the "Instruction bundle decoding" is reprogrammed. States named PRELD, BRANCH_PRED and PRED_ INIT are responsible for branch prediction. The branch behavior and timing will be better described in the next section. There are also auxiliary states like INIT, INIT_2 and EX_STALL_SYNC_2 which are necessary to enable data to be correctly registered during the FSM progression. The states HALTED and ENC_ERROR are reached when the CPU is halted by a halt instruction or when there is an error in the instruction decoding (cache alignment/stop bits missing).

The front-end controller was one of the most challenging parts of the processor implementation, specially to reduce unnecessary latencies and to keep a synchronous implementation. Avoiding latches on using registers enhances the VHDL synthesis but complicates the controller since data must be registered and available earlier. This behavior is illustrated by Figure 23. In this figure, there is a wave form of the pipeline executing a branch. Lines F, B, O, E and WB are the pipeline stages (Fetch, Bundle decoding, Operation decoding, Execution and Write-back). F_state is the FSM state and S0 to S3 are the decoded operation slots. Only Slot 0 is receiving operations in this example: every instruction has a stop bit. This occurs when the front-end is less stressed.

At time 515 ns, a branch is signaled. In the next cycle, the controller sets the new cache address and the cache responds at the

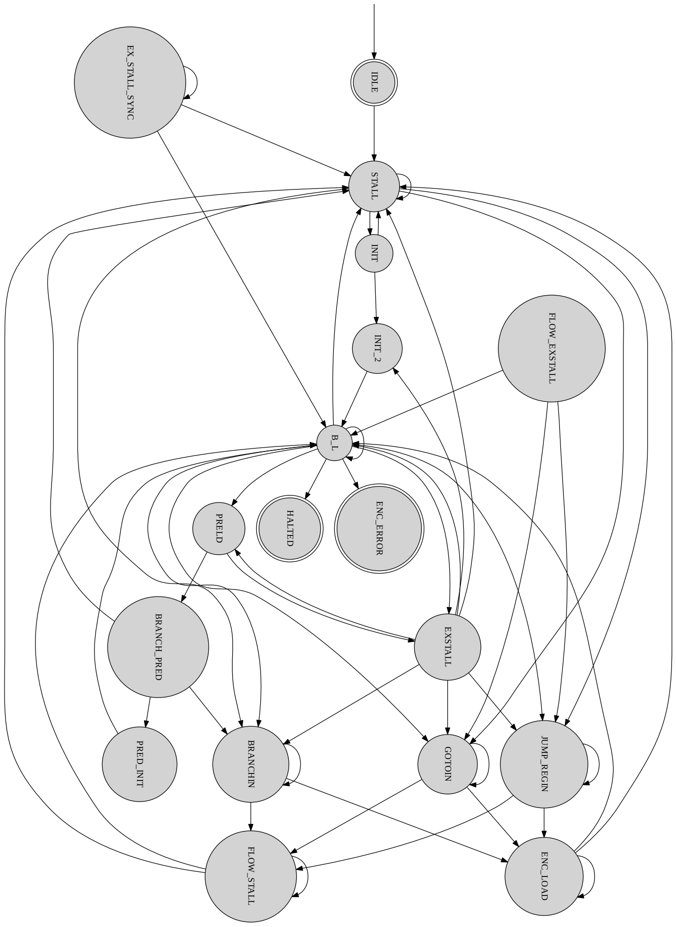

Figure 22 – Front-end control behavior.

next one. Next, stage F receives the data and programs the instruction decoder which decodes the instruction, dispatches operation to slot 0 and resets empty slots. We can observe the pipeline filling in lines

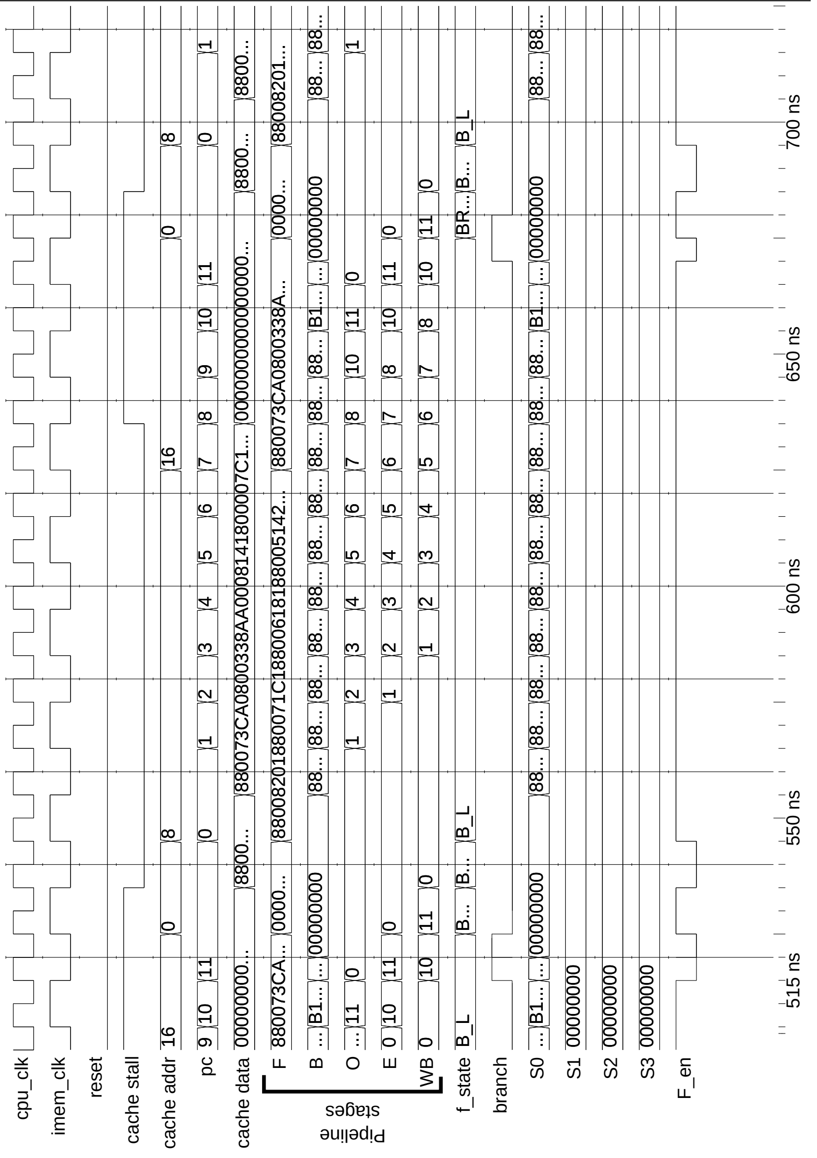

Figure 23 – Wave form of the pipeline front-end.

F to WB. Considering timing, we can note that to produce a decoded operation in slot 0 at 545ns, the cache was programmed at 515ns. When the fetch operates in full stream mode with 4 operations per cycle, a cache line address is generated on every cycle and this behavior is dynamic since instructions may have 1, 2, 3 or 4 operations. The front-end must control the pipeline considering possible high timing stall cycles, possible dependencies with one stall cycle, control flow instructions and dynamic instruction size.

5.3 THE CONTROL-FLOW SYSTEM

The control-flow system is composed by the branch unit and by the pipeline front-end. The front-end, as indicated earlier, controls the cache considering its signals, timing and address. It also controls instruction decoding. The branch unit, on the other side, instructs the front end when there are control-flow operations, evaluates the conditions for conditional branches and calculates the correct target addresses.

The processor branch unit takes care of all control flow operations including direct, indirect and conditional jumps. They must always be encoded in the first operation of an instruction: the branch unit is connected only to Slot S_0 data path as shown in Figure 16. Conditional branch instructions are encoded with a binary predicate (\$*br*) and a 23-bit offset parameter relative to the program counter (23-bit immediate). Branch conditions are calculated separately from the branch operation using arithmetic operations. Direct jumps also use a 23-bit offset parameter relative to the program counter. Indirect jumps use the address specified in the register without offset.

In order to perform a control-flow operation, the processor must decode the correct control-flow type, perform the address calculation and reprogram the pipeline front end. The control-flow system is in charge of detecting the control-flow type and calculating the target address. It signals the front end to reprogram the fetch unit providing the correct target address and the control-flow type. Cache transactions and pipeline flushing are performed by the fetch unit (described in Section 5.2). Figure 24 shows the VHDL component.

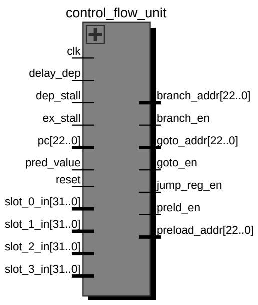

Figure 24 – Control flow unit component.

This unit is synchronous and receives stall signals like instruction dependencies (*delay_dep* and *dep_stall*) or execution stalls (*ex_stall*) to hold the output signals while another unit is stalled. It receives the program counter (*pc*) to perform PC relative jumps and slot operation data (*slot_0_in*) for control-flow decoding and immediate values. *Slot_1_in* to *slot_3_in*) are used for *preld* operations because they can be scheduled in any execution slot.

Each control-flow type has its own output signal: *branch_en* for branches, *goto_en* for direct jumps, *jump_reg_en* for indirect jumps and *preld_en* for branch prediction. There are also different address outputs to avoid 22-bit multiplexers because the use of synchronous adders reduces routing issues and increases the maximum operating frequency. All addresses are simultaneously calculated using independent adders and their values are used according to the control-flow type.

This unit is also responsible for the logic of conditional branches. The ISA supports two type of branches: *br*, branch when the predicate *pred_value* is true; *brf*, branch when the predicate value is false. The branch type is decoded, the predicate value is read and signal *branch_en* is activated accordingly to the branch type and predicate value.

5.4 REGISTER FILES

The register file is the primary and the fastest data storage area. In the case of a RISC processor, all processing data must be in the register file to be manipulated. Essentially, the register file is an array of registers and it must provide fast and dynamic access.

A register file of a single-fetch processor must provide at least two read and one write ports. Two read ports are necessary to read instruction operands while a write port is utilized to write processing results. In the case of pipelined processors, those ports are accessed in every clock cycle since operands are read in earlier pipeline stages and the result of further instruction is stored in the write-back phase.

For VLIW processors, the register file is even more complex. A 4-issue machine needs to read eight operands (two for each slot) and write four results at every cycle. This gives a total of twelve dynamic access ports. Those ports should also provide simultaneous access to every register address and forward a fresh result to a reading port if a write occurs at the same cycle. Multi-ported register files are typically constructed with expensive specialized SRAMs (Static RAMs) since conventional SRAMs provide only two ports.

In the context of this thesis, two register files are necessary with the following characteristics:

- General purpose register file:
  - 8 read ports.
  - 4 write ports.
  - 64 general purpose 32-bit registers (6 address bits).
  - Internal forward unit (writes are immediately forwarded to read ports).
- Predicates register file:
  - 4 read ports.
  - 1 fixed read port for full predication (*$br4*).

- 8 1-bit predicate/branch registers (3 address bits).
- Internal forward unit (writes are immediately forwarded to read ports).

There are many ways to implement a register file in FPGA. Using only combinational logic (register/latches) does not produce an efficient design. As FPGAs have lots of memory blocks, it is better to use them to store register file data. Unfortunately FPGA RAM blocks usually have just 2 ports: 1 for reading and 1 for writing. Therefore, to efficiently implement a register file, we need to replicate the memory blocks and track the most recent copy. For instance, if we have 4 write ports, we have 4 copies of the registers. The problem is how to track the most recent value in a read operation.

There are two recent works, (LAFOREST; STEFFAN, 2010) and (LAFOREST et al., 2012), that show how to efficiently track the most recent copy. The first one uses a structure called Live Value Table (LVT) that stores the write port number in a write operation and this value is used to multiplex the recent value in a read operation (all copies of the register file are read at the same time). LVT is implemented using only logic elements because it needs the same number of write and read ports as the register file but it needs much fewer hardware resources than a whole register file implemented with logic elements.

In the second work (LAFOREST et al., 2012), they use XOR operations to track the most recent copy. Bitwise XOR is commutative, associative and it has 3 interesting properties: $A \oplus 0 = A$, $B \oplus B = 0$ and $A \oplus B \oplus B = A$. In this implementation, a value that has to be written in a bank is first XORED with all other replicated banks. On a read, the register file returns the most recent value using the same XOR operation doing XOR of all replicated banks and returning the most recent one. The drawback of the XOR design is that each write requires reading, since we must store the write value XOR the old value of that location from the other bank. This increases the number of FPGA memories required to implement the design and the register file must be clocked twice as fast as the pipeline. The LVT design uses more logic elements while the XOR design uses more FPGA memories. In our implementation we chose the LVT design because its implementation is more modular compared to the XOR design.

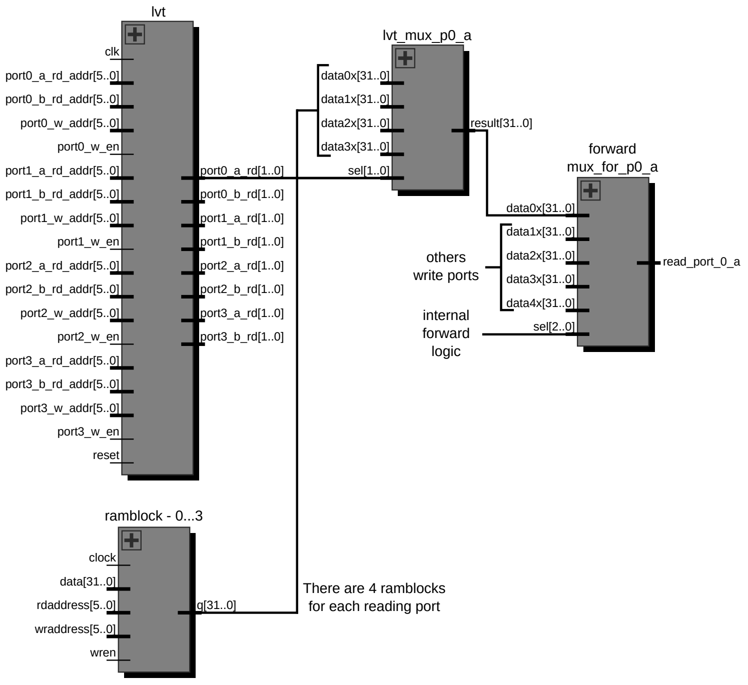

Figure 25 – Logic diagram for one read port.

A simplified register file internal diagram is depicted in Figure 25. There are 4 independent write ports generating 4 different data copies of particular register. These copies are stored using RAM blocks (64 lines of 32-bit – 64 32-bit registers). The LVT structure records the most recent copy and selects it using multiplexer *lvt_mux_p0_a*. After the most recent copy is selected, another multiplex is required. Since the Cyclone IV RAM blocks do not forward fresher data during a simultaneous read/write (they output only the older value or "don't care" value), the multiplexer selects newer data coming from other write ports. This logic is not presented here but it only checks if a read and write occur simultaneously and selects the correct/newer data. The LVT manages a total of 32 RAM blocks: 8 read ports with 4 write ports require 32 copies of the registers.

In order to improve the register file implementation since LVT uses too much memory and bitwise XORs requires a pipelined register file, we can use temporary register attached to each functional unit. This technique was used in (SANTOS; AZEVEDO; ARAUJO, 2006) and present interesting results. Temporary registers in each functional unit minimize the pressure over the global register file and could also reduce the overall number of register file ports. On choosing this technique, we also must optimize the register allocation in the compiler.

5.5 PREDICATION SUPPORT

Typically, supporting full predication adds considerable levels of logic to critical processor data paths (FISHER; FARABOSHI; YOUNG, 2005), specially in instruction encoding and forwarding paths. However, the proposed method uses an available encoding bit and does not add logic to the forwarding paths since only dual paths with only one predicate is considered. Besides being simple, it does not diminish the predication capabilities.

Like other registers, predicate register *$br4* is read immediately after a VLIW instruction is decoded into S0 to S3 operation slots during the "Operation Decoding – O" pipeline stage. Since *$br4* must be read in every cycle, an additional fixed address read port is added to the predicate register file. Adding this new port to this register file has a low impact since there are only 8 1-bit registers and its read address is fixed. Compared to the general purpose register file which has 12 32-bit ports, the predicate register file is considerably simpler having 9 1-bit ports (5 read ports and 4 write ports).

After the operations are decoded and before they are executed by the arithmetic and memory units, they are registered in a inter-stage pipeline buffer as illustrated in Figure 26. The deactivation logic is executed by the "OD - EX - Buffer". This buffer is a synchronous

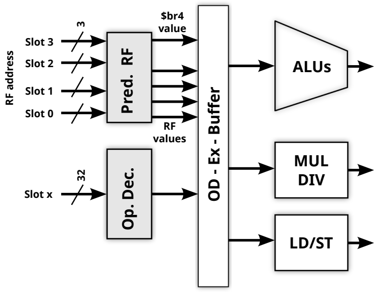

Figure 26 – Pipeline inter-stage responsible for the predication logic.

unit and it already receives all necessary signals to deactivate execution units considering the bit 30 and the value of the *$br4* register.

In the simple predicate mode, the execution of a particular *x* slot is deactivated if:

$$slot\_x.bit[30] \wedge \neg \$br4 = true \tag{5.1}$$

Control signals are reset and the execution unit does not perform any calculation if bit 30 is true and the predicate *$br4* is false.

In the case of complete mode, the instruction *paron* sets a register in the "OD - EX - Buffer": par_on_off. This register, operation bit 30 and predicate *$br4* reset execution unit control signals performing the following logic:

$$par\_on\_off \wedge (slot\_x.bit[30] \oplus \$br4) = true \tag{5.2}$$

This logic deactivates the operation if complete mode is activated when the value of bit 30 and *$br4* are different (XOR logic port). If bit 30 is false, operations are executed if *$br4* is false, but ignored if it is true. If bit 30 is true, operations are executed if *$br4* is true, but ignored if it is false.

As we can see, the deactivation logic of the predication systems is performed using simple “AND” and “XOR” logic ports added to the “OD - EX - Buffer”. The logic added to implement the predication systems does not jeopardize the operating frequency of the prototype since simple registers were reset and the critical data path considering 32-bit words and multiplexers of the forwarding logic were preserved.

5.6 PIPELINE INTERLOCK

An interlock mechanism is necessary when there are instruction dependencies that are not resolved by a forwarding logic due to hardware complexity or when the requested data is not yet available when a read is performed. The proposed VLIW design has full forwarded execution stage units to avoid stalls from Read after Write (RAW) data dependencies. This means that any calculation performed by any unit is forwarded to the unit which needs the data. Unfortunately, there are two dependencies in the control flow unit which cannot be resolved by forwarding logic regarding branch and direct jump instructions. Both cases are illustrated in Table 19 and Figure 27 illustrates the pipeline behavior of the arithmetic one.

Table 19 – Example of interlock cases.

RAW for $br0 between arithmetic (**cmpne**) and branch (**brf**) instructions	
cmpgt $br0, $r18, $r16	write on $br0
;	
brf $br0, $BB4_1	read on $br0
RAW for $lr between memory load (**ldw**) and **goto** instructions	
ldw $lr = 20[$sp]	write on $lr
;	
goto $lr	read on $lr

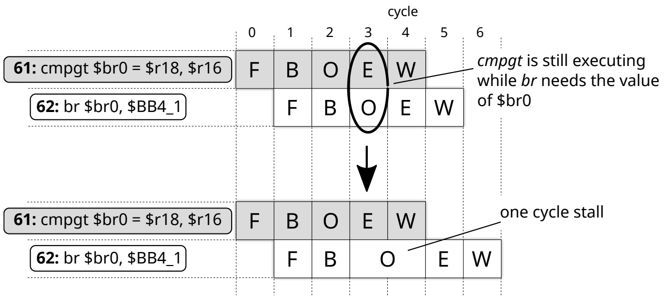
cycle
0
1
2
3
4
5
6
61: cmpgt $br0 = $r18, $r16
62: br $br0, $BB4_1
F
B
O
E
W
cmpgt is still executing while br needs the value of $br0
one cycle stall

Figure 27 – Pipeline behavior of interlock RAW dependency.

The interlock is a hardware protection and it is only activated when those instructions are executed subsequently. In both cases, there is a stall cycle which can be avoided when the compiler schedules a non dependent instruction before the control-flow one.

The interlock is implemented comparing control signals between "Execution" (E) and the "Operation Decoding" (O) stages. The complete logic is described in Algorithm 1.

The RAW branch interlock compares the predicate register (*br* – *branch_reg*), read for the branch operation during the "O" stage, with the predicate register destination written during the "E" stage. When there is subsequent read and write, the signal *branch_wb_en* will be true as well as the destination and read branch register addresses will be equal. In this case, the signal $ctrl\_flow\_stall$ instructs the pipeline front end to stop. Since the "E" stage continues to execute, as soon as the arithmetic instruction follows to the next pipeline stage, interlock logic will be false and the front-end resumes the pipeline execution. We only consider a reading register in the $slot\_0$ because control-flow operations are only scheduled in the $slot\_0$ data path.

The RAW *goto* interlock works like the branch one. First, it checks whether there is a *goto* operation in $slot\_0$ and then it checks if the *goto* operation reads the same register address of the memory load operation (*mem_reg_dest*). The *goto* interlock has only two statements because memory operations are only scheduled in *slot_0* and *slot_1*.

Algorithm 1 Interlock logic

```
1: {Activated when slot_0 has a branch control flow instruction}
2: if $O.slot\_0.branch\_op = true$ then
3:     {Write in br in Slot_0}
4:     if $O.slot\_0.branch\_reg = E.slot\_0.branch\_dest \wedge E.slot\_0.branch\_wb\_en = true$ then
5:         $ctrl\_flow\_stall := true$
6:     end if
7:     {Write in br in Slot_1}
8:     if $O.slot\_0.branch\_reg = E.slot\_1.branch\_dest \wedge E.slot\_1.branch\_wb\_en = true$ then
9:         $ctrl\_flow\_stall := true$
10:    end if
11:    {Write in br in Slot_2}
12:    if $O.slot\_0.branch\_reg = E.slot\_2.branch\_dest \wedge E.slot\_2.branch\_wb\_en = true$ then
13:        $ctrl\_flow\_stall := true$
14:    end if
15:    {Write in br in Slot_3}
16:    if $O.slot\_0.branch\_reg = E.slot\_3.branch\_dest \wedge E.slot\_3.branch\_wb\_en = true$ then
17:        $ctrl\_flow\_stall := true$
18:    end if
19: end if
20: {Activated when slot_0 has a goto to register}
21: if $O.slot\_0.goto\_op = true$ then
22:    if $O.slot\_0.goto\_reg = E.slot\_0.mem\_reg\_dest \wedge E.slot\_0.mem\_reg\_wb\_en = true$ then
23:        $ctrl\_flow\_stall := true$
24:    end if
25:    if $O.slot\_0.goto\_reg = E.slot\_0.mem\_reg\_dest \wedge E.slot\_0.mem\_reg\_wb\_en = true$ then
26:        $ctrl\_flow\_stall := true$
27:    end if
28: end if
```

5.7 ARITHMETIC AND LOGIC UNITS

The arithmetic and logic units (ALUs) perform all arithmetic and logic functions supported by the processor ISA excluding multiplication and division. Those two operations are implemented in a different unit due to their additional complexity. Processor ALUs are typically fast and they should perform calculations in one cycle. In the case of this VLIW processor, there are four symmetric ALUs and they are fully-forwarded. All RAW dependencies are resolved by hardware and they never cause any processor stall.

ALUs are reprogrammed components. They receive two input sources, one input function selector, one output result and one carry-out result as depicted by Figure 28.

ALU implementation in VHDL is straightforward because all arithmetic and logic functions are typically supported by the FPGA tool. The challenging part is the implementation of the full forwarding

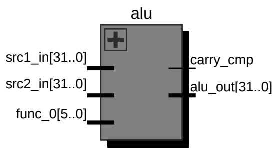

Figure 28 – Arithmetic and logic unit diagram.

logic keeping the ALUs with one cycle latency.

In order to achieve the desired timing requirements, all ALUs are implemented synchronously and together in a single VHDL component. This component contains all four ALUs, all necessary forwarding logic and multiplexers. The synthesis tool should also be instructed to keep all those components together and locked in a specific FPGA region. Figure 29 shows a simplified diagram of a single ALU with forwarding multiplexers with all available input sources.

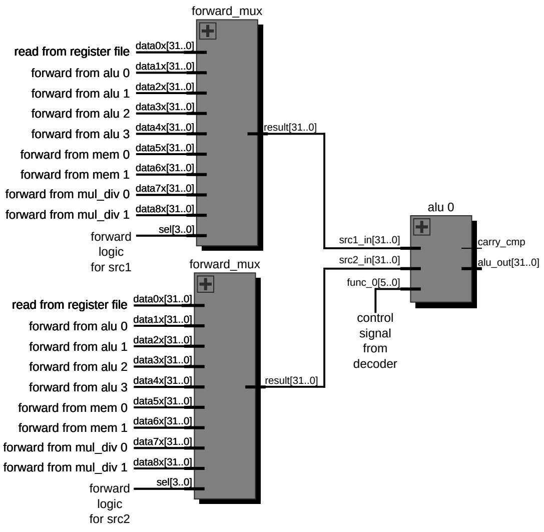

Figure 29 – Forward logic multiplexers for one ALU.

The complete arithmetic execution stage with four fully-forwarded ALUs has:

- Four ALUs (Figure 28).
- Eight multiplexes with 9 ports – one for each ALU input source.
- Eight forward logic (Algorithm 2) – one for each ALU input source.

The forwarding logic is implemented comparing control signals between "Execution" (E) and the "Operation Decoding" (O) stages. The logic for one ALU input source (*src1*) is described in Algorithm 2. All other ALUs and their input sources have similar logic to select the appropriate ALU input value.

Algorithm 2 Forward logic for a single ALU input

```
1: $src1\_mux\_sel := reg\_file$ {default selection}
2:
3: if $O.slot\_0.src1\_reg = E.slot\_0.dest\_dest \wedge E.slot\_0.reg\_wb\_en = true$ then
4:     $src1\_mux\_sel := alu0\_val$ {forward from alu 0}
5: end if
6: if $O.slot\_0.src1\_reg = E.slot\_1.dest\_dest \wedge E.slot\_1.reg\_wb\_en = true$ then
7:     $src1\_mux\_sel := alu1\_val$ {forward from alu 1}
8: end if
9: if $O.slot\_0.src1\_reg = E.slot\_2.dest\_dest \wedge E.slot\_2.reg\_wb\_en = true$ then
10:     $src1\_mux\_sel := alu2\_val$ {forward from alu 2}
11: end if
12: if $O.slot\_0.src1\_reg = E.slot\_3.dest\_dest \wedge E.slot\_3.reg\_wb\_en = true$ then
13:     $src1\_mux\_sel := alu3\_val$ {forward from alu 3}
14: end if
15: if $O.slot\_0.src1\_reg = E.slot\_0.mem\_reg\_dest \wedge E.slot\_0.mem\_reg\_wb\_en = true$ then
16:     $src1\_mux\_sel := mem0\_val$ {forward from mem 0 unit}
17: end if
18: if $O.slot\_0.src1\_reg = E.slot\_1.mem\_reg\_dest \wedge E.slot\_1.mem\_reg\_wb\_en = true$ then
19:     $src1\_mux\_sel := mem1\_val$ {forward from mem 1 unit}
20: end if
21: if $O.slot\_0.src1\_reg = E.slot\_0.mul\_reg\_dest \wedge E.slot\_0.mul\_reg\_wb\_en = true$ then
22:     $src1\_mux\_sel := mul0\_val$ {forward from mul_div 0 unit}
23: end if
24: if $O.slot\_0.src1\_reg = E.slot\_1.mul\_reg\_dest \wedge E.slot\_1.mul\_reg\_wb\_en = true$ then
25:     $src1\_mux\_sel := mul1\_val$ {forward from mul_div 1 unit}
26: end if
```

The forwarding logic compares a source register, read for an arithmetic operation during the "O" stage, with a destination register written during the "E" stage. When there is subsequent read and write, the signal *reg_wb_en* will be true as well as the destination and read register addresses will be equal. In this case, the signal $src1\_mux\_sel$ selects the correct updated value that is registered in the pipeline buffer but not yet written in the register file. This logic enables Read-After-Write dependency resolution without pipeline stalls.

5.8 DATA MEMORY INTERFACE

The memory unit is controlled by the Finite State Machine (FSM) illustrated by Figure 30. This FSM is designed to access the scratchpad, it is also constructed using synchronous components and synchronized by the rising edge of the processor main clock. Each FSM edge transition is a clock cycle.

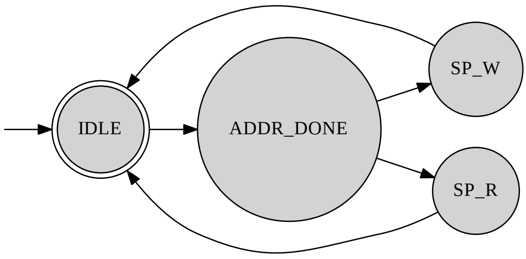

Figure 30 – Memory unit states and behavior

During the *IDLE* state, two adders are always calculating two effective memory addresses. They use the base and offset values provided by the “OD-Ex Buffer”. This state also generates the stall signal to the pipeline front end when a memory operation is issued to the memory unit.

When there is a store operation, the *ADDR_DONE* state calculates the “byte enable” masks to be issued to the scratchpad memory to provide byte, half-word (16-bit) or word (32-bit) writes to the specific address and the specific write signal to be issued to the scratchpad memory. This state also generates error if the memory address is not correctly aligned. An alignment error occurs when the effective address is not a multiple of four for word access ($eff\_addr \mod 4 \neq 0$) or it is not a multiple of two for half-word access ($eff\_addr \mod 2 \neq 0$). The effective scratchpad write occurs during the *SP_W* state using the truncated word address (addresses issued to the scratchpad memory are always multiple of four) and the "byte enable" mask specifies which bytes are written. The *SP_W* states also resets the stall signal and the pipeline resumes execution in the next cycle.

When there is a load operation, the *ADDR_DONE* state immediately makes a 32-bit access to the scratchpad memory using the truncated word address. This state also generates an error if the memory address is not correctly aligned. During the *SP_R* state, byte and half-word masks are applied accordingly and 32-bit signal extension is also performed for signed memory loads. The *SP_R* state also resets the stall signal and the pipeline resumes execution in the next cycle.

Like the arithmetic units, the memory unit needs forwarding logic to resolve RAW dependencies. Figure 31 shows the diagram of the address adder and its input parameters. The multiplexer selector uses similar logic as the arithmetic forward shown in Algorithm 2.

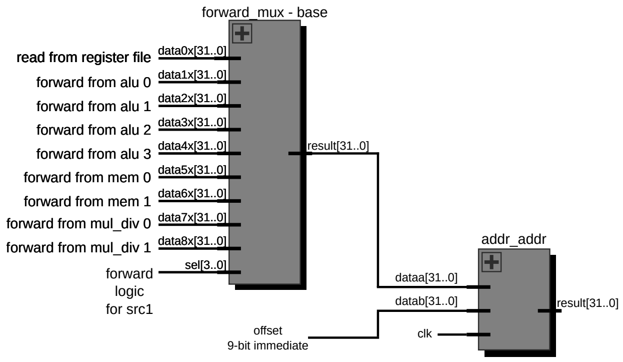

Figure 31 – Memory effective address calculation with forward.

5.9 MULTIPLICATION AND DIVISION UNIT

The multiplication and division unit provides a hardware implementation of these two arithmetic operations. The ST-231 ISA has a large set with 17 types of multiplications but it is not compatible with full division operations. It only supports division steps where a complete division with quotient and remain results of integer divisions must be assisted by software using loops or compiler library calls.

This unit provides signed and unsigned 32-bit multiplication supporting 2 multiplications per instruction bundle. ST-231 ISA's 17 types of multiplications were generalized with 3 different operations: simplified multiplication (*mull*) which returns the 32-bit lower bits of a 64-bit signed result, high order multiplication (*mull64h*) which returns the 32-bit higher bits of a 64-bit signed result and unsigned high order multiplication (*mull64hu*) which returns the 32-bit higher bits of a 64-bit unsigned result. Accordingly with the LLVM compiler documentation (LATTNER; ADVE, 2004) [1], these three operations must be supported and they are enough to provide signed and unsigned hardware multiplications. Further operations could be supported but their use are subject to compiler back end implementation

Considering the division, we added hardware support increasing the ISA with four new operations for integer signed and unsigned division with quotient and remainder individual results. Only one division operation is supported per bundle. The supported operations are: *div_r* which returns the remainder of a signed 32-bit division, *div_q* which returns the quotient of a signed 32-bit division, *div_ru* which returns the remainder of a unsigned 32-bit division and *div_qu* which returns the quotient of a unsigned 32-bit division. Accordingly with the LLVM compiler documentation (LATTNER; ADVE, 2004), these operations must be supported to avoid division implemented by software using library functions.

The multiplication and division arithmetic operations are performed by six components implemented by the Altera function library. They are illustrated by Figure 32.

Two individual 32-bit multipliers are required to support signed and unsigned operations and their result are 64-bit large (*mul_64_signed*

[1] http://llvm.org/

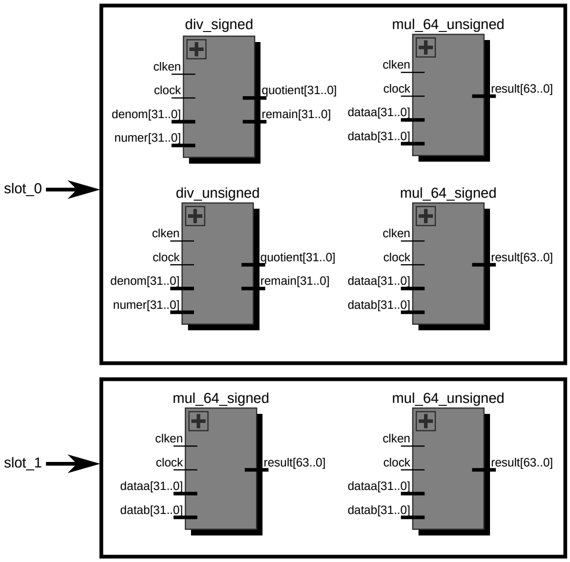

Figure 32 – Multiplication and division unit components per slot.

and *mul_64_unsigned*). The total number of multipliers are four: one signed and one unsigned per slot. Altera's multipliers are composed by the association of 9-bit embedded multipliers. Each 32-bit multiplier uses eight 9-bit embedded ones plus additional control logic. They also require one cycle to perform the multiplication.

Two individual 32-bit dividers are required to support signed and unsigned division: *div_signed* and *div_unsigned*. Each divider provides the quotient and the remainder of the operation. The used FPGA family (Cyclone IV) does not have any embedded component to accelerate division. Besides, Altera's function library provides divider components, that are implemented using only standard logic. Dividers are constructed by the association of adders, making them large and slow, requiring sixteen cycles to perform an operation with 100 Mhz as clock cycle. Adding division in both slots jeopardizes the hardware fitting and the target operating frequency. Division operation could be accelerated using a floating point unit since floating point division is implemented with FPGA embedded specific hardware as the multipliers.

The multiplication and division unit is controlled by a Finite State Machine (FSM) illustrated by Figure 33. This FSM is intended to control and to synchronize multiplication and division components. It is synchronized by the rising edge of the processor main clock and each FSM edge transition is a clock cycle.

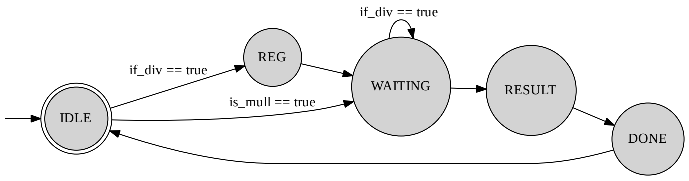

Figure 33 – Multiplication and division unit states and behavior

The multiplication and division unit waits for an operation at the IDLE state. As soon as the pipeline buffer signals a multiplication or a division operation, a stall signal is sent to the pipeline font-end. When a division is requested, internal signals are registered at the REG state. When multiplication is requested, IDLE state goes directly to the WAITING state. At WAITING state, the dividers or multiplicators are activated using the *clken* signals and there is no progression until the operation is concluded. Sixteen cycles are waited for the division and only one for the multiplication. At the RESULT state, 32-bit low order or 32-bit high order of the 64-bit result is registered for multiplication or the remainder or quotient for the division. At the DONE state, all requested operations are done, the pipeline front-end resumes its operation and the write-back stage writes the results to the register file.

Like the arithmetic and the memory units, forwarding logic is necessary as well. 9-input multiplexers are placed before dividers and multiplicators inputs and their selection logic is similar to the previously described Algorithm 2.

5.10 TIMING CHARACTERISTICS AND FPGA RESOURCE UTILIZATION

The FPGA resource utilization is shown in Table 20 with a clock speed of 100MHz. The execution stage length description for each instruction type is shown in Table 21.

We can summaries the timing information required for the WCET analyzer as follows:

- Cycle time: 10ns
- Cache miss: 15 cycles.
- Interlock latency: 1 cycle, only between arithmetic compares and conditional branches.
- Execution stage latencies: Table 21.
- Control-flow latencies:
  - Not taken branch: no latency.
  - Taken branch: 4 cycles.
  - Call, goto: 4 cycles.
  - Predicted taken branch: 1 cycle.
  - Misspredicted branch: 6 cycles.

5.11 SUMMARY

The instruction cache memory was implemented in the FPGA using internal RAM blocks. The instruction cache has 32 lines with 256 bits per line forming a 1kb direct-mapped cache memory.

Table 20 – FPGA resources usage (Cyclone IV GX – EP4CGX150DF31C7). Resource usage reported by Quartus II 64-bit V 15.0.0 Build 145 WEB Edition.

FPGA Global Resource	**Usage / Available**	**%**
Total combinat. functions	21,220 / 149,760	14%
Dedicated logic registers	5,017 / 149,760	3%
Total memory bits	1,188,764 / 6,635,520	15%
Emb. 9-bit multiplayers	30 / 720	4%
Total PLLs	1 / 8	13%
Component	**Usage / Total**	**%**
Cache	378 / 21220	1.78%
Fetch	3194 / 21220	15.05%
Oper. decoding	922 / 21220	4.34%
Memory Units	1434 / 21220	6.76%
Mul. and Div.	4297 / 21220	20.25%
Register Files	3658 / 21220	17.24 %
ALUs w/ forw.	7002 / 21220	33.00%
Pipeline Buffers	331 / 21220	1.56%

A VLIW processor with 4-wide fetch unit needs a register file capable of reading 8 and writing 4 values to the register file each cycle, which produces 12 data ports. Unfortunately FPGA RAM blocks usually have just 2 ports: 1 for reading and 1 for writing. Therefore, to efficiently implement a register file, we need to replicate the memory blocks and track the most recent copy. To do so we use the work of (LAFOREST; STEFFAN, 2010) where they use a structure called Live Value Table (LVT) to track the most recent copy.

The Arithmetic Logic Units (ALU) are capable of executing unsigned and signed arithmetic, compare and logic operations. The 4 ALUs and their forwarding logic have the most critical data path. A full forwarded ALU has, at least, 5 possible internal input values: a value coming from the register file, a more recent value coming from its

Table 21 – Processor instruction timing

Stages	Instruction Type				
	Arithmetic	**Multiply**	**Divide**	**Store**	**Load**
E_0	Compute results	Read operands and start	Read operands and start	Compute address	Compute address
E_1		Compute results	Compute results	Create store masks	Access scratch-pad
E_2		Compute results	Compute results	Access scratch-pad	Apply load masks and register results
E_3		Register results	Compute results		
E_n			Compute results		
E_{19}			Register results		
Timing	1 cycle	4 cycles	19 cycles	3 cycles	3 cycles

own previous calculation and three more values coming from the other ALUs. Additional input values are necessary for each execution unit: 2 for two memory operations and 2 for two multiplications and divisions. Forwarding logic exists for each ALU input selecting the most recent value based on the operation read and write parameters. Timing for all ALU operations is only one cycle and it never stalls.

Multiplications were implemented using dedicated FPGA embedded 9-bit multipliers. Multiplications are more sophisticated and they need 4 stages to complete. Considering the current implementing technology, the number of substages is equal to the number of cycles. A division specialized VHDL block uses only combination logic. A single division requires 19 cycles to target 100Mhz and it is a very large component. To implement one hardware signed and one unsigned division we need as much logic cells as half of the entire arithmetic unit (4 ALU with all forwarding logic).

The Load/Store unit accesses only the scratchpad memory. Data access from other memories should be done via DMA operation but this is not covered in this thesis. A real-time deterministic DRAM controller is described in the work of (REINEKE et al., 2011). The instruction memory was implemented using internal FPGA memory with 2 read ports, one for cache access and one DMA unit. The Scratchpad also used internal FPGA memory. Both scratchpad and SDRAM operate at 100Mhz.

6 WORST-CASE EXECUTION TIME ANALYSIS

The *Worst-Case Execution Time* is the maximum execution time of a program considering a specific processor with input data and initial state well defined. In order to calculate the WCET of programs running on the VLIW processor described earlier, we implemented a static WCET analyzer. Such analyzer usually requires some well defined methods such as value analysis, control-flow reconstruction and data-flow analysis, microarchitecture analysis and finally obtaining the worst-case execution path (CULLMANN et al., 2010).

The inputs of the WCET analyzer are the compiled object files (.o) and the annotated control flow graph (.cfg) generated by the compiler as illustrated by Figure 34. Extracting the control-flow graph (CFG) directly at the compiler give us some advantages because we do not need to reconstruct the CFG from the machine code. This pass is already performed internally by the compiler.

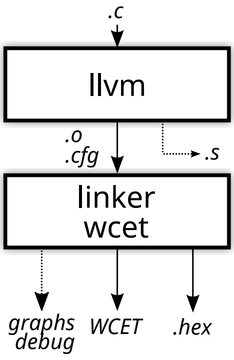

Figure 34 – Data-flow between the compiler and WCET/linker

The steps executed by the WCET analyzer can be summarized as follows:

- **Loop detection:** loops are not directly described in the CFG, only their bounds which are automatically detected by the compiler or annotated in the program source code. The detection of loops uses Tarjan's algorithm (LENGAUER; TARJAN, 1979) for identifying strongly connected components in graphs.

- **Instruction cache analysis:** analysis for classification of cache accesses in *always miss*, *always hit*, *first miss* and *conflict*.
- **Pipeline modeling:** This pass models the VLIW pipeline behavior calculating the basic block times, disregarding any cache effects.
- **Worst-case path search:** This phase searches for the worst-case path and its respective computation time (WCET), considering the previous analyses. We used Implicit Path Enumeration Technique – IPET (LI; MALIK, 1995), which produces an optimal solution, for this purpose.

Since the proposed architecture does not suffer from timing anomalies, we can conduct each analysis in isolation and combine the results at the path search phase.

Like most WCET analyzers, the implemented one is context sensitive. The analyzer considers the paths by which each node of the CFG can be reached, and computes the behavior of *cache* and executions for each node in each of these contexts. For example, loops may have a different execution time for the first iteration, when instructions must be loaded in the cache. This behavior can be extracted by cache analysis, which is the subject of the next section.

6.1 BASIC CONCEPTS

Programs need to be represented mathematically to perform static analysis. They are composed by a control flow graph, basic blocks, loops and paths. Aho et al. (2011) defined each one as follows:

Definition 1. *(Basic Block – bb) A basic block is a maximal sequence of instructions, which can be reached only by its first instruction and the only way out is its last instruction.*

Definition 2. *(Control Flow Graph – CFG) A control flow graph is a directed graph $G = (V, E, i)$, where vertices V represent the basic blocks and the edges $E \subseteq V \times V$ connect two vertices v_i and v_j if, and only if, v_j is immediately executed after v_i. The CFG input vertex is represented by i and it does not have input edges ($\nexists v \in V : (v, i) \in E$).*

Definition 3. *(Path) A path through a CFG $G = (V, E, i)$ is a sequence of basic blocks $(v_1, ..., v_n) \in V$, with $v_1 = i$ e $\forall j \in 1, ..., n-1 : (v_j, v_{j+1}) \in E$.*

Definition 4. *(Loop) A loop is a strongly connected component of a graph G. Loops are composed of only one header and the header is its entry point. There may be different edges returning from internal loop vertices to the header and different loop output edges.*

Figure 35 presents an example of CFG. Vertices are labeled $I[S, E]$, where I is an index number, S is the start address of the basic block and E its end address. The input vertex is 4 and a loop is formed by vertices 1 and 2, where 2 is the loop header.

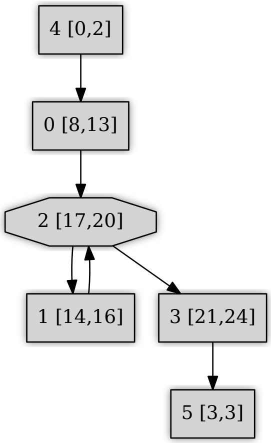

Figure 35 – Control flow graph example.

Two main conditions are important in order to perform WCET analysis: monotonicity and basic composition.

Monotonicity means that an instruction with a higher latency necessarily maintains or increases the time of a sequence of instructions under analysis. A higher latency instruction should never decrease the overall execution time because the local worst case should not diminish the global worst case. In the case of temporal anomalies, this assumption does not hold.

Basic composition means that the WCET values of sub-paths can be safely composed when calculating the overall WCET. Like monotonicity, the basic composition property is also lost in the presence of temporal anomalies.

The processor described in this thesis does not present timing anomalies and so we can assume monotonicity and basic composition. The purpose of the next analyses is to obtain the individual basic block timing which it can be expressed by:

$$t_{bb} = t_p + n_c.t_c + n_m.t_m \tag{6.1}$$

Symbol t_p is the basic block pipeline timing considering all multistage operations and $n_c.t_c$ is the number of cache misses multiplied by the cache penalty. The processor only accesses the scratchpad memory and this is already considered as a multistage operation during the pipeline analysis. An eventual memory penalty (as a DMA transfer) could be modeled by $n_m.t_m$ where n_m is the number of memory accesses and t_m is the memory access timing.

After the timings of all basic blocks are estimated, the program execution time is calculated by:

$$obj = maximize \sum_{\forall bb_i} x_i \times t_i \tag{6.2}$$

Symbol x_i is the number of times basic block bb_i is executed and t_i is the worst execution time of bb_i.

6.2 INSTRUCTION CACHE ANALYSIS

Cache memories are necessary to minimize the gap between processor and memory performances. Usually, main memory has a clock that is slower than the processor, so a fast cache is placed between them where most recent data is stored, promoting faster access. We use a direct-mapped instruction cache memory, so some sort of analysis is necessary to model cache misses during WCET analysis. Usually cache analysis is performed using abstract interpretation as described in (ALT et al., 1996) but here we used traditional data flow analysis to compute cache states. Similar analyses are also used in (MUELLER; WHALLEY, 1995) and (LEE et al., 1998).

A cache memory is characterized by its capacity, line size and associativity.

Definition 5. *(Capacity) Capacity is the cache's total number of bytes.*

Definition 6. *(Line or block size) Line size is the quantity of bytes transferred from memory to cache when a cache miss occurs. A cache should have $n = \frac{capacity}{line.\ size}$ lines.*

Definition 7. *(Associativity) It consists of the mapping of various main memory addresses to cache lines. A direct-mapped cache has unitary associativity, where a main memory specific address is always mapped to the same cache line. If associativity is 2, a main memory address is mapped to 2 different cache lines. When associativity is not unitary some sort of replacement policy must exist to decide which line will have data eviction. The relation $\frac{n}{assoc.}$ defines the number of sets of a cache.*

In the context of this work, we will focus only on unitary associativity (direct mapping) when one memory block can reside only in a specific cache line. This feature does not impose a restriction upon the analysis, but is related to the used direct-mapping configuration in the hardware. A direct-mapping cache memory is formed by a sequence of lines $L = l_1, l_2, l_n...$ which store a set of memory blocks $M = m_1, m_2...m_s$. A memory block m with address $addr$ is stored in line l_i following Equation 6.3. The operator % represents the modulus or remainder of the division.

$$l_i = addr(m) \ \% \ n \tag{6.3}$$

Regarding the presence of the existence of a particular instruction in the cache, we define:

Definition 8. *An instruction could be in the cache if: 1) there was a transition sequence in which the block corresponding to the instruction memory had been referenced in previous basic blocks; 2) this memory block is referenced previously in the same basic block.*

Definition 9. *(Abstract state) An abstract state of the cache of a basic block is the subset of all memory blocks that can be cached before executing the basic block.*

Definition 10. *(Reachable abstract state) A reachable abstract state is the subset of all memory blocks that can be reached by CFG transitions.*

Definition 11. *(Effective abstract state) An effective abstract state of a basic block is a subset of all memory blocks that can be reached by considering all the CFG paths to the basic block in analysis.*

6.2.1 Reachable and effective abstract state

We can construct the reachable abstract state where we can map all memory blocks accessed by every basic block. This analysis uses the same principles of reaching definitions in traditional data flow analysis and it follows Algorithm 3.

Figure 36 shows an example of data flow applied to cache analysis. The abstract reachable state ($RMB_{bb}(cl) = data$) is beside each output edge. bb is the basic block number, cl is the cache line and $data$ represents an identifier of the memory data, the memory address index. Figure 36 also shows which cache line is accessed by a basic block and its memory address index inside the nodes ($l_0 = 0$ for basic block 0 and $l_1 = 1$ for basic block 4). It is very easy to know the

Algorithm 3 Reachable abstract state for every basic block

```
while change do
    change := false
    for all $i \in bb$ do
        for all $c \in$ cache_blocks do
            {$p$: predecessors of $i$}
            RMBin$_i(c) := \bigcup_{\forall p}$ RMBout$_p(c)$

            temp:= RMBout$_i(c)$

            {If this cache block is the last accessed by $i$}
            if $last_i(c) \neq \emptyset$ then
                RMB$_i(c) := last_i(c)$
            else
                RMB$_i(c) :=$ RMBin$_i(c)$
            end if

            if RMB$_i(c) \neq$ temp then
                change := true
            end if
        end for
    end for
end while
```

cache contents after execution of a basic block, we have only to inspect the basic block instruction addresses. The RMBs track the possible state of cache lines after the "execution" of all basic blocks. We know, for instance, when bb_3 executes, cache line 0 should have data which index is 0 ($RMB_3(0) = 0$) because to reach basic block 3, bb_0 must execute. This same logic follows for $RMB_3(1) = 1$ and $RMB_3(2) = 0$, where $RMB_3(1) = 1$ comes from bb_2 and $RMB_3(2) = 0$ comes from execution of bb_2 itself.

Some conditions must hold when using this analysis to address cache hits and misses. First, loops must iterate at least once. If this is not true, we cannot assume a hit in bb_4 in Figure 36 because bb_2 will never execute and l_1 will never receive data index 1 used by bb_4. Secondly, some path checking should be done during the analysis. A hit could only exist in bb_4 if the worst-case path passes at least once through bb_2. In this case we are pessimistic and assume a miss in bb_4. This type of cache analysis could be optimistic if we use only abstract reachable states without path checking.

Path checking is performed constructing the effective abstract state using Algorithm 4.

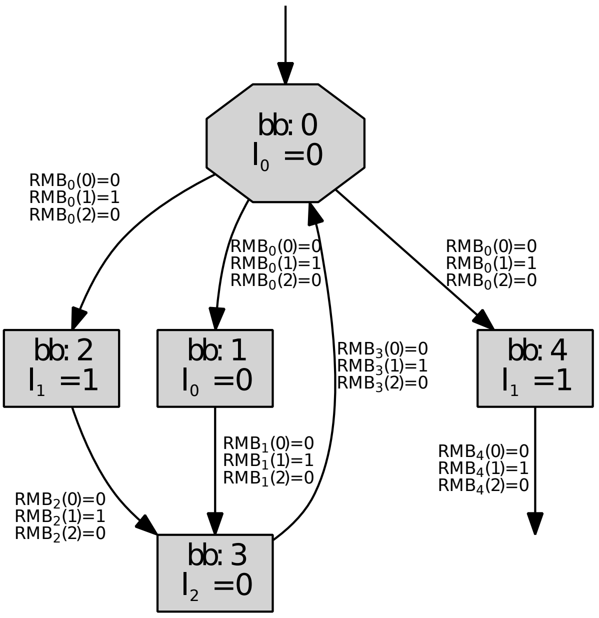

Figure 36 – Cache abstract reachable state example

Algorithm 4 Effective cache state for every basic block

```
 1: for all $i \in bb$ do
 2:     for all $c \in RMB_i(c) \vee |RMB_i(c)| = 1$ do
 3:
 4:         {If this block is accessed by $bb_i$}
 5:         if $last_i(c) \neq \emptyset$ then
 6:             $\mathrm{EMB}_i(c) := c$
 7:         else
 8:             {$P$: paths that leads to $i$}
 9:             if $c \in last_j(c) \mid j \in \forall P$ then
10:                 $\mathrm{EMB}_i(c) := c$
11:             end if
12:         end if
13:     end for
14: end for
```

Effective abstract cache set $EMB_i(c)$ is constructed from reachable abstract set $RMB_i(c)$. First, on Line 2 it is checked if the set cardinality is 1 ($|RMB_i(c)| = 1$). If this cache block is accessed by the basic block, it is added to the abstract set at line 6. Otherwise it is checked when this memory reference is accessed by all paths leading up to the basic block in question.

Considering the example in Figure 36 and the basic block 4, the algorithm has the following execution in the construction of the effective abstract state for cache line 1 of basic block 0 ($EMB_0(1)$):

- if vertex 0 accessed $l_1 = 1$, we could terminate and it will be classified as a cache hit;
- following vertex 0, we check 3, where there is no access;
- following vertex 3, we check 2. In this vertex, $l_1 = 1$ is accessed and this path search is ended;
- following vertex 3, we check 1. There is not a $l_1 = 1$ access and we continue;
- following vertex 1, we check 0. Vertex 0 was already visited. We can conclude that there is a path where $l_1 = 1$ is not referenced;
- $l_1 = 1$ does not belong to the effective abstract state of basic block 0 and therefore there is a cache miss in basic block 4.

6.2.2 Cache accesses classification

We classify all program instructions in “always miss”, “always hit”, “first miss” and “conflict” after the data flow analysis in conjunction with path checking – reachable and effective abstract state.

Definition 12. *(A_MISS) There is a cache fault every time this instruction is executed – always miss.*

A_MISS classification occurs in compulsory or capacity faults. For example, a compulsory miss occurs in bb_0 in Figure 36 since the first instruction of the program is not in the cache memory at the beginning of the program execution. *A_Miss* is also the correct classification for the instruction of bb_4, since $l_1 = 1$ does not exist in the effective abstract state of bb_0 (predecessor of bb_4).

Definition 13. *(A_HIT) There is a cache hit every time this instruction is executed – always hit.*

A_HIT is the class for instructions where: 1) they are not the basic block first instruction or the first instruction of a cache line; 2) they are in the cache effective state. In the case of Figure 36, all instructions of the bb_1 can be classified as hits because their cache line will always be previously accessed by the bb_0.

Definition 14. *(F_MISS) This classification is related to loops. There is a cache miss only at the first iteration of the loop – first miss;*

In the case of F_MISS, if a loop iterates 100 times, there is a cache miss only at the first iteration. For the other 99 iterations, there are cache hits. This classification occurs in the bb_2 of Figure 36 for $l_1 = 1$ since it is not accessed by any predecessor except itself.

Definition 15. *(CONFLICT) This classification occurs when there are multiple reachable paths to a particular basic block and each of these paths has a different effective cache state, which may cause faults or misses depending on the path flow.*

CONFLICT occurs, for example, in bb_3 of Figure 37. If the execution flow is $0 \to 2 \to 3$, there is a cache hit since $l_1 = 1$ is accessed in bb_2; if the execution flow is $0 \to 1 \to 3$, there is a cache miss since there are no references to $l_1 = 1$ in predecessor basic blocks.

After instruction classification, we can count the number of faults (A_MISS) that impact directly on the basic block time. The classes F_MISS and CONFLICT are used during the path analysis to determine the program flow that maximizes the execution time.

6.3 PIPELINE MODELING

The objective of this analysis is to determine the execution time of instructions and basic blocks when executed on the processor pipeline. This analysis does not consider hardware elements like instruction cache.

Pipelined processors, the execution time of a single instruction in cycles will be at least to the number of pipeline stages. However, this time may be higher if any hazard occurs involving data dependencies

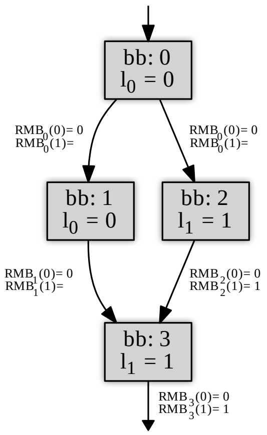

Figure 37 – CONFLICT classification example.

of previous instructions or multistage operations. Considering that the used architecture has 5 pipeline stages, it would take 5 cycles for an instruction to be executed. If there were two simple instructions, the total time would be 6 cycles, and so forth.

As the used VLIW pipeline is simple and deterministic and it stalls during multistage operations, its timing modeling can be visualized in Algorithm 5 and summarized by the following steps:

- Instructions and operations for every basic block are decoded following ST231 ISAs and the VLIW bundle encoding.

- The basic pipeline timing of each basic block is the number of its instructions.

- Multistage operation timing such as multiplications, division and memory access and the latency of branches and calls/gotos are added to the basic pipeline timing.

- Pipeline hazards (data dependency between operations) are added to the basic timing.

- Both paths (activated or not) of full-predicated instructions are considered. The bit 30 of each operation selects the true/false paths. The path that leads to a higher execution time is the basic block time.

The time of an instruction hazard is 1 cycle and all control flow instructions have 4 cycles added to the basic block timing. The timing of each multicycle operation is shown in Table 22.

Algorithm 5 Pipeline timing modeling

```
1: for all $i \in bb$ do
2:     {t[0] is for predications executed on false(0) and t[1] if for predications executed in true(1)}
3:     {This is checked by VLIW operation bit 30}
4:     $t[0] := 0$
5:     $t[1] := 0$
6:
7:     $t_{bb} := |bundles|$
8:
9:     for all $j \in bundles_i$ do
10:        for all $k \in operations_j$ do
11:            if $k \in multicyles\_op$ then
12:                $t[BIT\_30] := t[k.get\_bit(BIT\_30)] + multicycle\_time(k)$
13:            end if
14:
15:            if $k \in ctr\_flow\_op$ then
16:                {Control flow operations are not conditionally executed by bit 30, both paths have over-
                   head}
17:                $t[0] := t[0] + ctr\_flow\_time(k)$
18:                $t[1] := t[1] + ctr\_flow\_time(k)$
19:            end if
20:
21:            if $k \in hazard(bundles_{i-1})$ then
22:                $t[BIT\_30] := t[k.get\_bit(BIT\_30)] + hazard\_time(k)$
23:            end if
24:        end for
25:    end for
26:    {Final basic block time computation}
27:    $t_{bb} := t_{bb} + max(t[0], t[1]) + pipeline\_length - 1$
28: end for
```

After individual basic block times are obtained and cache analysis is performed, it is necessary to consider the execution flow. There are five types of basic block transition considering the branch unit described in Chapter 5:

- **Direct flow:** it happens when there is a direct flow between basic blocks due to linear addresses, for instance, when bb_i ends at address 55 and bb_{i+1} begins at address 56;

Table 22 – Additional time for multicycle instruction timing for $multicycle\_time(op)$ function

Type	Timing
Multiplication	3
Division	18
Mem. scratchpad	2

- **Not taken branch**: it is similar to a direct flow, but there is a branch instruction at the basic block ending;
- **Taken branch**: it happens when there is a branch without direction or a goto/call instruction;
- **Taken branch with direction**: it is a taken branch, but there is a "taken" direction with a "preld" instruction;
- **Not taken branch with direction**: it happens when a branch is directed as "taken" but it does not take the branch.

Each of the five flow transitions is modeled as a different type of edge of the annotated CFG exported by the compiler. The timing of each edge is modeled as δ similar to (ENGBLOM; JONSSON, 2002), as shown in Figure 38 for direct flow transition. In this figure, we want to get the execution time of a direct flow between basic blocks 1 and 2 with times 8 and 6 cycles respectively. The sum of the execution times of both blocks is 14 cycles and this does not represent the real execution time of the flow. The correct execution time is 10 cycles as shown in Figure 38 due to "an amendment" in the pipeline between both basic blocks. Thus, during flow analysis and WCET obtaining, we must subtract $\delta = 4$ cycles for each edge transition between basic blocks. This correction is applied directly in the problem formulation using integer linear programming for obtaining the WCET of the entire program, as we show in the following subsection.

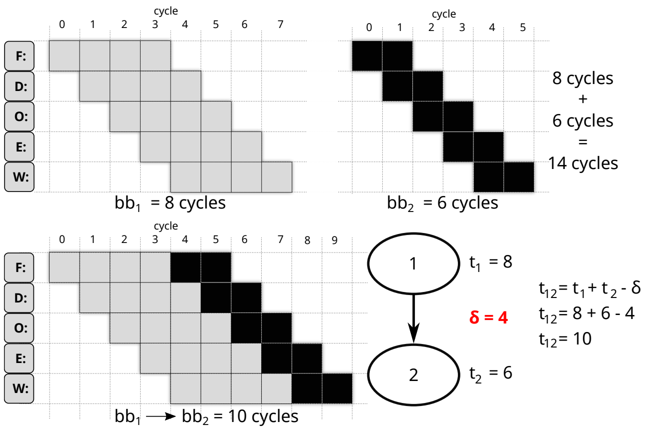

Figure 38 – Example of timing composition of two successive (linear addresses) basic blocks

Table 23 shows δ for each type of edge/transition. Direct flow is $\delta = 4$ cycles as described in Figure 38. When a basic block finishes off with a branch, call or goto instructions, it always includes 4 penalty cycles as a multistage operation. Considering the branch behavior, if the edge is a "not taken branch", we must remove 4 penalty cycles (always included) plus 4 cycles to simulate the linear flow effect because the branch is actually not executed, which is $\delta = 8$ cycles. For taken branches (and calls and gotos), $\delta = 4$ cycles models 4-penalty cycles. Similar rationale happens for "taken" and "not taken" edges. $\delta = 7$ cycles for "taken" because there is still one cycle latency for directed branches and $\delta = 2$ because there are 6 penalty cycles when there is a missprediction on directed branch.

6.4 WORST-CASE PATH SEARCH

We previously referenced IPET (LI; MALIK, 1995) (OTTOSSON; SJöDIN, 1997) as being an efficient technique to search worst-case paths. In this subsection we present the modeling of linear constraints

Table 23 – Modeling factor for each type of transition flow

Flow type	δ	**Behavior**
Direct flow	4	No penalty
Not taken branch	8	No penalty
Taken branch	4	4 cycles penalty
Taken branch dir.	7	1 cycle penalty
Not taken branch dir.	2	6 cycles penalty

made in the context of the implemented analyzer. The modeling follows an approach similar to (LI; MALIK, 1995).

We model the optimization problem considering that each basic block bb_i is executed x_i times and it has a execution time t_i. The objective function is expressed by:

$$obj = maximize \sum_{\forall bb_i} x_i \times t_i \tag{6.4}$$

Note that, with the approach used to obtain the basic block times, this estimation becomes pessimistic. This analysis considers the time from the first instruction of the basic block entering the pipeline until the exit of the last instruction. However, the execution of successive basic blocks is amended within the pipeline and its depends on branch behavior, as shown in the previous subsection. We can express this behavior in ILP as a discount of δ (pipeline modeling factor) each time a basic block is executed as given by:

$$obj = maximize \sum_{\forall bb_i} x_i \times t_i - x_i \times \delta \tag{6.5}$$

Regarding different execution contexts, we can rewrite the previous equation considering the edges of the CFG, instead of nodes (basic blocks). The execution time of a basic block x_i can be rewritten using edges instead of nodes. Let be d_{j_i} the edge from basic block bb_j to basic block bb_i, then x_i could be expressed by:

$$x_i = \sum_{\forall bb_j \rightarrow bb_i} d_{j\_i} \tag{6.6}$$

Since the execution time t_i of a basic block bb_i could also increase due to cache misses for different paths, t_i is also replaced by t_{j_i}. Replacing x_i and t_i, we get the final objective function described by:

$$obj = maximize \sum_{\forall bb_i} \left(\sum_{\forall bb_j \rightarrow bb_i} d_{j\_i} \times t_{j\_i} - d_{j\_i} \times \delta \right) \tag{6.7}$$

For the remainder of this section, we will consider the example in Figure 39 (a simple C program and its respective CFG). There is a loop with an *if-then-else* sentence. The input and output nodes are explicitly drawn as ellipses and the output node is purely symbolic and does not represent any real basic block. An edge $d_{i\_j} \in CFG$ means that the basic block j can be executed after the execution of the basic block i.

Regarding the Example of Figure 39, we get WCET objective function modeling in Table 24 using language MathProg (MAKHORIN, 2008).

Table 24 – Objective function modeling using Language MathProg.

```
maximize wcet: d7_0*14 - d7_0*4 +
d4_1*9 - d4_1*4 + d5_1*9 - d5_1*4 +
d0_2*9 - d0_2*4 + d1_2*9 - d1_2*4 +
d2_3*9 - d2_3*4 + d3_4*15 - d3_4*4 +
d3_5*15 - d3_5*4 + d2_6*8 - d2_6*4 +
dstart7*7 - dstart7*4 + d6_8*5;
```

```
int main(int argc,
    char** argv){

 int i = 1;
 int j = 0;

 for (j = 0; j < 5; j++){
     if(i < 6){
         i++;
         i+=1;
         i+=2;
         i+=3;
     } else {
         i--;
         i-=1;
         i-=2;
         i-=3;
     }
  }
}
```

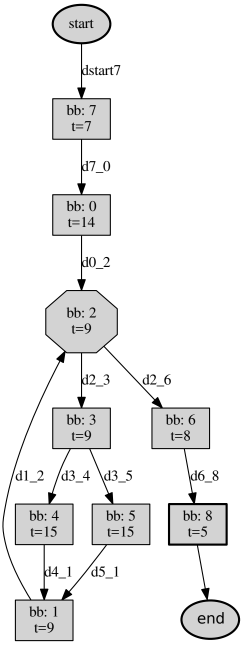

Figure 39 – Example of a C program and its control-flow graph.

6.4.1 ILP Constraints

The WCET objective function defined in Equation 6.7 requires linear constraints to operate, otherwise it can not converge. The IPET technique consists of a set of constraints which mainly consider flow conservation and loop bounding. According to IPET, the following restrictions shall be applied:

Start and end of execution constraint: all program execution must have a begin and an end. So the flow must pass exactly once by the CFG entry and exit nodes, which are represented in the CFG as *dstart* and *dend* respectively. The constraint is modeled by:

$$dstart = 1 \quad \wedge \quad dend = 1 \tag{6.8}$$

Flow conservation constraint: every flow entering a basic block from a predecessor, should come out from that basic block to a successor. The flow must be maintained during the execution of the program. This restriction is modeled by Equation 6.9, which must be valid for each basic block bb_i.

$$\sum_{\forall bb_j \to bb_i} d_{j\_i} - \sum_{\forall bb_i \to bb_k} d_{i\_k} = 0 \tag{6.9}$$

Regarding the Example of Figure 39, we get control-flow conservation restrictions in Table 25 using language MathProg.

Table 25 – Control-flow conservation restrictions using language MathProg.

```
s.t. xc0: d7_0 - d0_2 = 0;
s.t. xc1: d4_1 + d5_1 - d1_2 = 0;
s.t. xc2: d0_2 + d1_2 - d2_3 - d2_6 = 0;
s.t. xc3: d2_3 - d3_4 - d3_5 = 0;
s.t. xc4: d3_4 - d4_1 = 0;
s.t. xc5: d3_5 - d5_1 = 0;
s.t. xc6: d2_6 - d6_8 = 0;
s.t. xc7: dstart7 - d7_0 = 0;
s.t. xc8: d6_8 - dend8 = 0;
```

Loop bound constraint: basic blocks should execute according to the bounds of the loops to which they belong. A basic block can belong to several loops, provided that they are nested. If a basic block is the header of a loop, it can execute once more at the end to test the exit condition. Regarding loop limitation, the constraint of Equation 6.10 must be valid for each basic block bb_i with loop bound lb_i.

$$\sum_{\forall bb_j \to bb_i} d_{j\_i} <= lb_i \tag{6.10}$$

Regarding the Example of Figure 39, we get loop restrictions in Table 26 using language MathProg.

Table 26 – Loop restrictions using language MathProg.

s.t. x0: d7_0 <= 1;
s.t. x1: d4_1 + d5_1 <= 5;
s.t. x2: d0_2 + d1_2 <= 6;
s.t. x3: d2_3 <= 5;
s.t. x4: d3_4 <= 5;
s.t. x5: d3_5 <= 5;
s.t. x6: d2_6 <= 1;
s.t. x7: dstart7 = 1;
s.t. x8: d6_8 <= 1;

Loop execution constraint: as a loop is an auto conservative connected component, it only shall be considered part of the WCET if it is effectively executed. A loop is executed when the flow enters its header. Equation 6.11 ensures this constraint.

$$\sum_{\substack{\forall bb_j \to bb_i \\ \wedge bb_j \notin loop(bb_i)}} d_{j\_i} \times ilb_i - \sum_{\substack{\forall bb_i \to bb_k \\ \wedge bb_k \in loop(bb_i)}} d_{i\_j} = 0 \tag{6.11}$$

The previous equation must be valid for every basic block bb_i which is loop header. ilb_i represents the bound of the loop of which this basic block is header and $loop(bb_i)$ represents the respective loop.

Regarding the Example of Figure 39, we get Conservation restrictions in Table 27 using language MathProg.

Table 27 – Conservation restrictions using language MathProg.

s.t. xal2: d0_2*5 - d2_3 = 0;

Instruction cache constraints: the WCET of a program must take into account that the flow through certain basic blocks can be impacted by different cache behaviors. Another factor is that now the execution of basic blocks is context dependent: a basic block C may have distinct execution times when reached from predecessors A or B. To represent this effect we associate weights to edges with the execution times of basic blocks when reached by these edges.

As an example, consider the CFG of Figure 40 with instruction cache. Basic block 1 will have one execution time when reached from basic block 4 and another time when reached from basic block 5, due to distinct cache states.

From the considered cache states ("always miss", "always hit", "first miss" and "conflict"), only one of them needs special attention in the IPET model, which is the *first miss*. First miss occurs only once for each basic block, each time the loop which contains this basic block is executed. So, all first misses in a loop occur in the first iteration of each complete execution of this loop. To address this, the CFG is expanded into a *multigraph*, and the *first miss* modeled as a separate edge. With a dedicated edge, it is possible to model constraints that control the occurrence of this event. In a Control Flow Multigraph, an edge can have one of three types: if there is a first miss between pairs of basic blocks, then a pair of edges between the two blocks exists, one representing the first miss d_{fm} $(type = fm)$ and another representing flows of subsequent iterations (hits) d_h $(type = h)$. The third edge type represents all other situations (*always hit*, *always miss* and *conflict*). The weights of the edges $W(di\_j)$ represent the cost to execute basic block j, when preceded by the basic block i, considering cache effects. In Figure 40, one can see that a first miss will occur in the basic block 4 represented by the thicker edge.

When modeling first misses, we must follow different approaches for basic blocks that are loop headers and basic blocks that are not. For loop header, a first miss (if any) will occur only at the outer edge of the loop. The constraint is modeled by:

$$\sum_{\forall bb_j \to bb_i \wedge bb_j \notin loop(bb_i)} d_{j\_i} \leq elb_i \tag{6.12}$$

elb_i is the bound of the outer loop bb_i. For normal basic blocks the constraint is modeled by Equation 6.13, for each bb_i.

$$\sum_{\forall bb_j \to bb_i} d_{j\_i} \leq elb_i \tag{6.13}$$

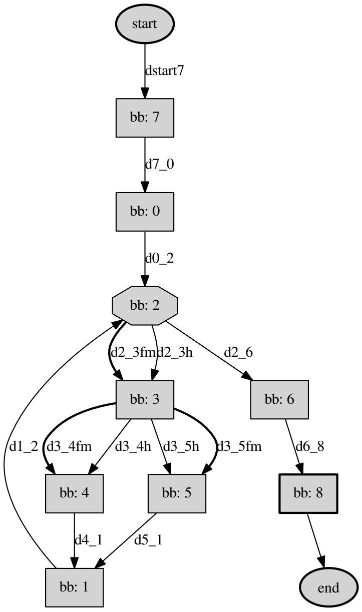

Figure 40 – Multigraph of the example

Regarding the Example of Figure 39, we get cache restrictions in Table 28 using language MathProg.

By joining all the previous constrains, we can calculate the WCET. The solution of the example is shown in Figure 41. For each edge,

Table 28 – Cache restrictions using language MathProg.

```
s.t. xcache3: d2_3fm <= 1;
s.t. xcache4: d3_4fm <= 1;
s.t. xcache5: d3_5fm <= 1;
```

$d_{i\_j} = n \times t_{i\_j}$, where i_j represents the transition from basic block i to j, n the number of times a flow executes and t_{i_j} the time of such transition. One can see that the worst-case execution time for the example is 421 processor cycles, and the result was obtained in 0.00176 seconds. The solver used by our tool is GLKP (MAKHORIN, 2008).

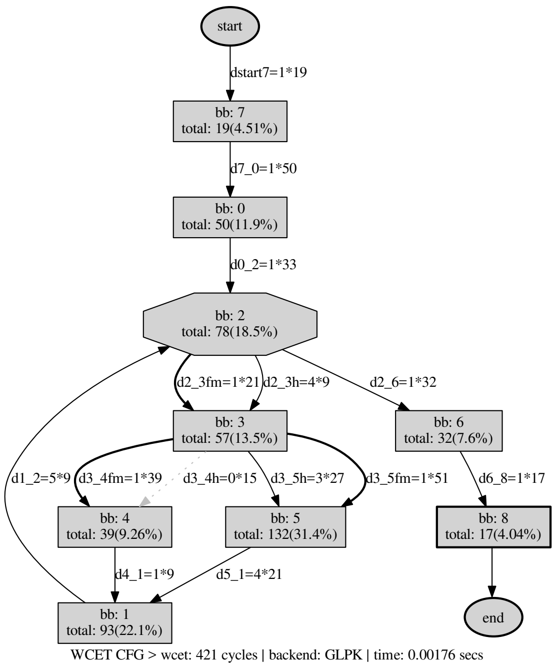

Figure 41 – WCET final result of example of Figure 39

6.5 SUMMARY

This chapter presented techniques to perform the WCET analysis for the VLIW processor considered in this thesis. A tool considering all described aspects was implemented in C++. Since we target performance enhancement on using VLIW machines for real-time systems, all evaluations performed were obtained through this WCET analysis.

In relation to static analysis, we can notice that it is a task that demands the union of numerous techniques such as data flow analysis (cache), path analysis (IPET) and modeling of internal processor components. The more complex is the system in question, more complex are the techniques used. Nevertheless, it is very important that the system allows monotonicity and basic composition properties. If these assumptions hold, we can increase the efficiency and precision of a particular technique, such as cache analysis, and increase the overall WCET analysis capabilities. Otherwise, the analysis will become increasingly complex.

7 EVALUATION

In this chapter, we evaluate the impact of several architecture techniques upon WCET performance. The main idea is to quantitatively show WCET performance gains considering features like wider instructions fetching, multiple memory operations, static branch prediction and predication.

To make WCET evaluations, we used the new code generator back-end for LLVM (LATTNER; ADVE, 2004) and the described WCET analyzer to compute the WCET of representative examples of Mälardalen WCET benchmarks (GUSTAFSSON et al., 2010). Both LLVM and the WCET framework are implemented in C++ and their source code are available upon request as well as the processor VHDL code.

The Mälardalen WCET benchmarks are a set of programs that have been assembled to make comparison between WCET tools. A number of WCET analysis tools have emerged in recent years and a comparison between these tools requires a common set of benchmarks. The typical evaluation metric is the accuracy of the WCET estimate, but of equal importance are other properties such as performance and the ability to handle all code constructs found in real-time systems. The goal of the Mälardalen WCET benchmarks is to provide an easily available, tested, and well documented common set of benchmarks in order to enable comparative evaluations of different algorithms, methods, and tools.

Table 29 shows a description of used benchmarks excluding those which use library functions, floating-point calculations and recursion. Static data are allocated to scratchpad memory during processor boot process and this time is not included. Some benchmarks are so small that this may have a non-null impact on performance.

The performance of a program execution on a processor depends on the compiler capabilities for that specific processor. Besides semantics, a VLIW compiler must also consider other hardware aspects to enhance performance. We also show how our LLVM code generator

Table 29 – Mälardalen WCET benchmarks (GUSTAFSSON et al., 2010)

Bench. name	**Description**
adpcm	Adaptive pulse code modulation algorithm
bs	Binary search for the array of 15 integer elements
bsort100	Bubblesort program
cnt	Counts non-negative numbers in a matrix
cover	Program for testing many paths
crc	Cyclic redundancy check computation on 40 bytes of data
duff	Using “Duff’s device” from the Jargon file to copy 43 byte array
edn	Finite Impulse Response (FIR) filter calculations
fac	Calculates the faculty function
fdct	Fast Discrete Cosine Transform
fibcall	Simple iterative Fibonacci calculation
insertsort	Insertion sort on a reversed array of size 10
janne _complex	Nested loop program
jfdctint	Discrete-cosine transformation on a 8x8 pixel block
lcdnum	Read ten values, output half to LCD
matmult	Matrix multiplication of two 20x20 matrices.
ndes	Complex embedded code
ns	Search in a multi-dimensional array
nsichneu	Simulate an extended Petri Net
prime	Calculates whether two numbers are prime

compares with a state-of-the-art VLIW compiler. In this evaluation, we compare the Mälardalen benchmark performance with measurement/simulations between the VEX system (FISHER; FARABOSHI; YOUNG, 2005) [1] and our LLVM back-end/VLIW prototype. This

[1] http://www.hpl.hp.com/downloads/vex/

is the only evaluation we use measurements instead of WCET performance. Since we do not have access to the framework from other related researches, some of them are proprietary, we use our own framework in order to estimate worst-case performance enhancement.

7.1 IMPACT OF THE COMPILER ON THE PERFORMANCE

We used the VEX system (FISHER; FARABOSHI; YOUNG, 2005) which consists of a VLIW compiler (based on the Multiflow C compiler with global trace instruction scheduling) and a parameterized cycle accurate simulator to demonstrate that different compilers have great influence upon performance. VEX is distributed as a binary package for the Linux operating system and we do not have access to its source code. Unfortunately we could not adapt the WCET analyzer for this platform so this is the only section where the results are evaluated using simulation instead of WCET.

We set VEX in a way as similar as possible to the described architecture. Cache configuration is identical (1KB direct mapped). There are the same number of execution units and branch latency is also identical. The VEX data cache is configured to have null latency during miss or write back in order to simulate a scratchpad memory and its allocation during boot. Some benchmarks are so small that this may have a non-null impact on measured performances. We also utilized standard compiler optimization flags (-O2) for VEX.

The results of this section regarding the LLVM were obtained by executing the compiled and linked benchmarks using the synthesizable VHDL code in ModelSim[2] and executing the code on the DE2i-150 board. There is a VHDL module who monitors the performance information. Since there is no variation in the benchmark input data, identical results are obtained executing each benchmark more than once for both platforms/compilers. In this evaluation, all available hardware features (predication, branch prediction, etc) are enabled in both VEX and LLVM.

[2] Modelsim is a VHDL simulator from Altera's Quartus Design Suit http://www.altera.com

Table 30 – Execution and worst-case performance comparing VEX and our prototype and WCET absolute values. WCET and Exec. time in cycles. IPC is Instructions per cycle

Benchmark	Our prototype			VEX		$R_{sim} = 1 - \frac{t_{VEX}}{t_{LLVM}}$
	WCET	Exec. time	IPC	Exec. time	IPC	
adpcm	18363	16084	0.47	9916	0.39	38.3%
bs	297	256	0.34	389	0.30	-52.0%
bsort	241139	122752	0.71	50740	1.25	58.7%
cnt	2881	2521	0.49	2698	1.02	-7.0%
cover	4672	4664	0.40	4685	0.52	-0.5%
crc	108340	48198	0.83	20989	1.29	56.5%
duff	2224	1541	0.64	905	0.80	41.3%
edn	132298	128105	0.80	24622	1.45	80.8%
fac	1326	834	0.42	693	0.51	16.9%
fdct	1936	1903	0.89	1036	1.33	45.6%
fibcall	463	459	0.89	310	0.71	32.5%
insertsort	2621	1634	0.48	722	0.88	55.8%
janne-cmplx	3120	597	0.32	503	0.70	15.7%
jfdctint	3769	3736	0.90	4773	1.42	-27.8%
lcdnum	893	457	0.37	477	0.39	-4.4%
matmult	292381	259998	0.60	149604	1.11	42.5%
ndes	142408	134901	0.72	47804	0.82	64.6%
ns	22900	18654	0.71	2464	1.48	86.8%
nsichneu	55732	39513	0.25	30722	0.16	22.2%
prime	34069	20701	0.32	91882	0.47	-343.9%

Table 30 shows the execution time of each Mälardalen benchmark with our prototype compiled with LLVM and with VEX compiled with its Multiflow based compiler. The WCET column shows the WCET for all benchmarks regarding our prototype and it represents an absolute value reference for the graphs of the remainder of this chapter. Benchmarks resulting in close values between simulation and WCET, like *fibcall*, have a more predictable code (e.g. static loop bounds) and input data that is more likely to produce the worst-case execution time during simulation/execution.

Regarding the execution time results, we can see that *bs* executes 52% faster when compiled with LLVM but *edn* is 81% faster when compiled with VEX. There is a big discrepancy in *prime* due to the fact that VEX uses an external library to implement division and we added hardware integer division. We can compare the IPC (Instructions per cycle) of both platforms/compilers. We can see that VEX is more effective in terms of operation parallelization where LLVM only outperforms VEX in 4 benchmarks (*adpcm*, *bs*, *fibcall* and *nsichneu*). It is interesting to note that a benchmark with higher IPC does not necessarily executes in less time. We can see this behavior in *jfdctint* and *lcdnum* where VEX has greater IPC but LLVM has smaller execution time. This occurs since Multiflow uses loop unrolling and more aggressive partial predication where calculations of both if-then-else paths are always executed. It shows also that not all optimizations used by VEX, which are appropriate for general-purpose applications, produce good results for real-time applications.

The main goal of Table 30 is to show that there is a huge difference on the performance of each benchmark considering very similar machines but with very different compilers. Different machines with different compilers and optimizations are even harder to compare. Because of such influence, in the next sections we will focus mainly on WCET performance with only the LLVM compiler and the improvement obtained through hardware characteristics.

7.2 USE OF PROCESSORS WITH WIDER FETCH

The purpose of this section is to show the importance of a wider fetch for real-time processors that use in-order pipelines. Since a discussion about the fetch width optimization, VLIW compiler techniques and hardware complexity varying the fetch width would be quite long and out of the scope of this thesis, we compared the same VLIW machine using single and four-issue fetch widths. This evaluation was made configuring the compiler to schedule instructions only in Slot 0 emulating a single-fetch machine but keeping all other parameters like memory configuration, pipeline stages and operational latencies fixed. Figure 42 shows how much the WCET is reduced by using a multi-issue machine. This reduction is defined by Equation 7.1. It establishes a relation between the WCET of each benchmark execution with single-operation and four-operation fetching and it demonstrates how fast the same program is executed considering worst-case conditions.

$$R_{fetch} = 1 - \frac{t_{four}}{t_{single}} \tag{7.1}$$

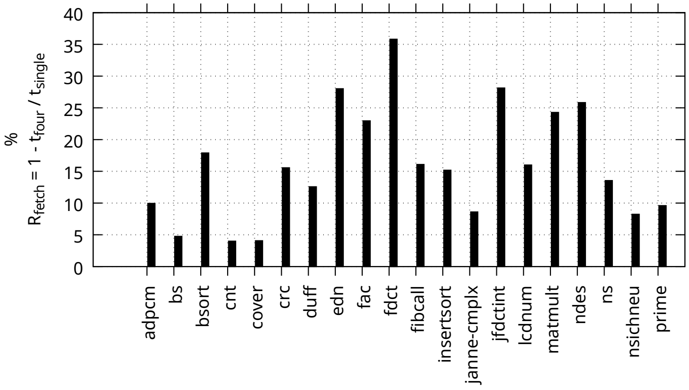

Figure 42 – WCET reduction between single-issue and four-issue.

We can see that WCET performance does not depend only on the hardware fetch width but also on the compiler instruction scheduling capabilities (as discussed earlier in Section 7.1) and on the possibility of parallelization of the code at the instruction level. Because of this, we do not observe a WCET reduction of 75% increasing only the fetch width. However, the performance of all benchmarks was increased demonstrating the advantage of using VLIW machines in real-time systems. Three of the benchmarks (*bs*, *cnt* and *cover*) decreased their WCET less than 5%, demonstrating poor instruction parallelization. Others like *fdct* had 36% of decreased WCET. The *fdct* – Fast Discrete Cosine Transform – has many calculations based on integer array elements and it is highly parallelizable: our LLVM compiler produced an average of 3.20 operations per instruction bundle while *bs* has only 1.23 operations per instruction bundle. With a state-of-the-art compiler, the performance increase on using a VLIW processor will be even higher.

7.3 USE OF "TAKEN"/"NOT TAKEN" STATIC BRANCH PREDICTION

Previously we described the branch unit which supports "taken" and "not-taken" static branch predictions. Figure 43 shows how much the WCET decreases on using a "taken"/"not taken" solution instead of only a "not taken" solution. This reduction is defined by Equation 7.2 and it relates the WCET between both "taken" and "not-taken" (T-NT) enabled and only "not-taken" (NT) enabled. All other hardware features are identically enabled in both WCET computations (two memory execution units, two multiplication units and predication disabled). We have changed only the direction of branches belonging to loops and underlined benchmark labels had null WCET decrease. They require a compiler WCET oriented static branch prediction as show in (BURGUIERE; ROCHANGE; SAINRAT, 2005) and this is hardware independent.

$$R_{branch} = 1 - \frac{t_{T-NT}}{t_{NT}} \tag{7.2}$$

More than half of the benchmarks had WCET reduction when the hardware supports both "taken" and "not-taken" predictions. The reduction depends on the code structure and we can notice that up to 12% of reduction could be achieved even without a WCET oriented static branch prediction. That demonstrates the importance of branch prediction for high performance real-time systems. It is interesting to note that the static branch prediction does impact WCET calculation but could be successfully modeled through different transition flows (Chapter 6).

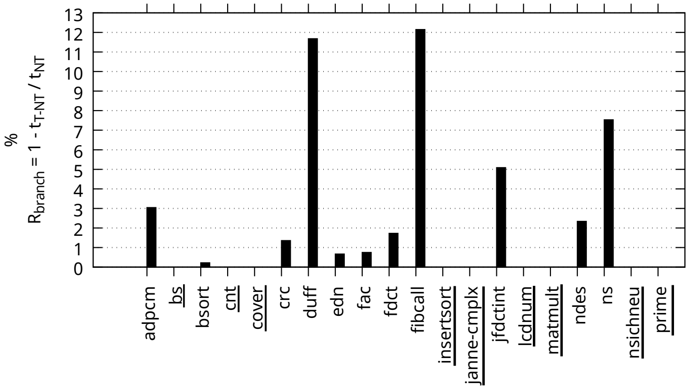

Figure 43 – WCET reduction on using branch prediction. Underlined ones have null reduction.

7.4 REDUCING EXECUTION STAGE OVERHEAD

As described in Chapter 5, there are operations, like memory and multiplication, that require multiple execution stages to complete. We also noted that this overhead is lower when the execution stage supports multiple operations per instruction. Figure 44 shows the WCET reduction when supporting dual-memory operations instead of single-memory operations. The reduction is defined as a WCET relation when executing each benchmarks with dual and single memory operation per instruction. It is illustrated by Equation 7.3. All other hardware features are identically enabled in both WCET computations (predication disabled, two multiplication units and full branch prediction enable).

$$R_{memory} = 1 - \frac{t_{dual}}{t_{single}} \tag{7.3}$$

There is considerable performance increase in most of the benchmarks with a WCET reduction of almost 15% in complex codes like *ndes* (lots of bit manipulation, shifts, array and matrix calculations) and memory intensive algorithms like *bsort* (bubblesort program). The overhead of multiple-stage operations is usually reduced with out-of-order execution which is not interesting for real-time processors. A

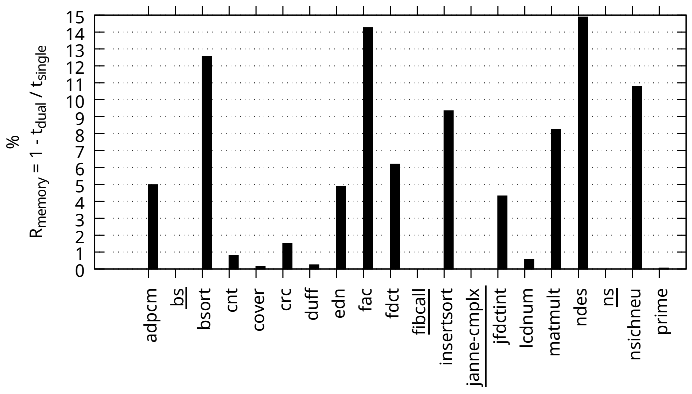

Figure 44 – WCET reduction with dual-ported scratchpad. Underlined ones have null reduction.

considerable improvement is achievable by supporting the parallelization of complex operations. In benchmarks with null improvement, the compiler could not schedule more than one memory operation per bundle.

7.5 PREDICATION

Predication can be used to reduce the control flow overhead of *if-then-else* and *if-then* structures. Figure 45 shows the WCET reduction when using such technique. The reduction is defined as a WCET relation when executing each benchmark with predication enabled and disabled. It is illustrated by Equation 7.4. All other hardware features are identically enabled in both WCET computations (two multiplication units, two memory units and full branch prediction enabled).

$$R_{pred} = 1 - \frac{t_{enab}}{t_{disab}} \tag{7.4}$$

Some of the benchmarks had WCET reduction, specially *fibcall* and *janne-complex* where there is considerable control flow overhead for *if-then-else* and *if-then* structures. On other side, the use of predication increased *nsichneu* WCET. This particular example simulates

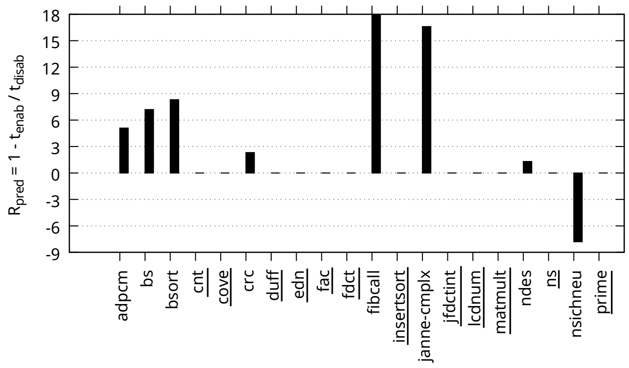

Figure 45 – WCET reduction on using predication.

an extended Petri Net with automatically generated code containing large amounts of if-statements (more than 250). *Nsichneu* also has high cache overhead: it is basically a linear code without any loops. As the predication increases the code, it produces more cache misses and decreases the overall performance and increases the WCET. Predication is a compiler technique and in benchmarks with null WCET reduction, predicated structures could not be detected.

7.6 OVERALL REDUCTION AND IMPACT ON WCET CALCULATION

In this section we establish the overall WCET reduction enabling and disabling all technologies similarly as in Sections 7.2 to 7.5 but we keep the processor with 4-wide fetch unit to make a more fair comparison.

In order to conduct this experiment, first, we compute the WCET for all benchmarks with the following conditions: 4-wide fetch, two multiplication units, one memory unit, predication disabled and only "not-taken" branch prediction. This test is labeled "All-disabled". After that, we compute the WCET using a 4-wide fetch, two multiplication units, two memory units, predication and full branch prediction enabled. This test is labeled "All-enabled".

Figure 46 shows the WCET reduction for all considered benchmarks applying “All-disabled” and ‘All-enabled” conditions. The WCET reduction is defined by Equation 7.5

$$R_{total} = 1 - \frac{t_{all-enab}}{t_{all-disab}} \tag{7.5}$$

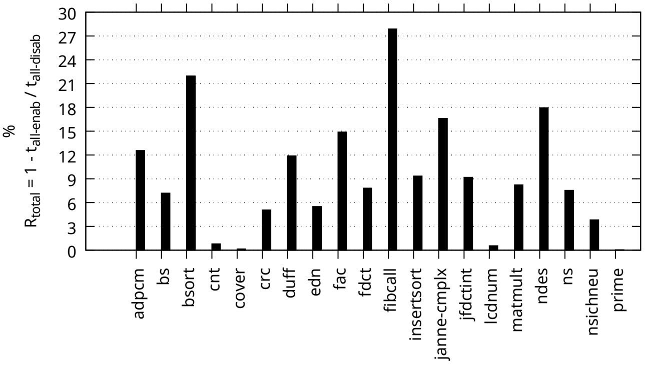

Figure 46 – Total WCET reduction.

We note that all benchmarks had considerable WCET reduction highlighting, the importance of the processor features used for high performance real-time systems. The overall improvement of benchmark ***nsichneu*** could be greater if the predication was disabled. This is the only feature which produces negative effects and the compiler could use the WCET information to disable such feature when it jeopardizes WCET performance. Tuning the compiler for WCET performance is not the subject of this thesis.

Considering the relevant performance improvement achieved, the highlighted processor features have low impact on the effort for the WCET estimation. Regarding the wider fetch, any static WCET analyzer must somehow decode instructions to estimate timing for each basic block. The adopted distributed encoding does not impose additional complexity because the analyzer must only inspect stop bits to decode bundles into fixed slot operations. Predication and various multistage operations are also taken care during the pipeline analysis and operation decoding. The only feature that produces additional overhead during WCET calculation is the static "taken/not-taken" branch prediction which is modeled by different transition flows during the worst-path search.

7.7 CONTRIBUTIONS TO THE STATE OF THE ART

The main objective of this thesis is to investigate various processor architecture features that lead to a predictable design with reasonable WCET performance. We also considered the WCET analysis of such architecture and VHDL implementation in order to infer its timing behavior.

We believe we have achieved our objectives to real-time processor architectures and regarding them, we can highlight the following contributions:

- Design space investigation and evaluation:

  We made an extensive evaluation of the proposed techniques and we highlighted their performance benefits. This evaluation was described in this chapter where we demonstrated the importance of each of the techniques showing how much the WCET is reduced. There were benchmarks where the worst-case execution time was reduced about 20% showing that we can improve the performance and also have a predictable design. Some of them like wide-instruction fetching and static branch prediction have such importance for performance that cannot be ignored for real-time systems.

  We can also noted that such wide analysis have not been considered in related researches since they focus only on specific objectives like memory architecture, multi-threading or multi-core. Such evaluation can guide new real-time processor design lines.

- Complete WCET analysis for the proposed design:

We presented all necessary analysis to perform WCET estimations of programs compiled for the proposed processor design. Such analyses included cache modeling, pipeline modeling and worst-case path search and they are described in Chapter 6. The existence of a WCET analyzer is important in order to achieve a predictable hardware design and, an integrated environment between WCET and compiler enhances WCET estimation.

A complete description of a WCET analyzer for a VLIW architecture have not been considered in related researches. The Patmos approach, which is the only one that uses the VLIW approach, uses the commercial ***aiT***[3] WCET analyzer. With our WCET analyzer and our design space exploration we could identify quantitatively how much hardware techniques contribute to WCET reduction, as well as how to analyze them during WCET static estimation.

- Complete VHDL design and timing characterization:

  We described an implementation of our deterministic real-time processor. We described the timing of each module and their VHDL implementation which are necessary for WCET modeling and estimation.

  Since the related researches focus on specific objectives, a project like ours that involves WCET analyzer, compiler and processor design brings a great research framework for future real-time researches.

- Detailed branch architecture:

  All details of the branch architecture are described including our methodology to model it during WCET analyses. We also extended the ISA to support static branch prediction as well as the performance benefits of using or not this technology in real-time systems.

[3] http://www.absint.com/ait/

Some related researches do not recommend the use of static branch prediction since they add complexity to the WCET analyzer. However, the performance of loop structures are greatly jeopardized without branch prediction technology. We showed that a predictable static branch predictor can be analyzed using different transition flows and it greatly increases performance.

- Low overhead full predication system for VLIW processors:

  We propose a low-overhead full-predication system without adding overhead to the hardware data paths or its forwarding logic. The proposed predication system enhances the support of the single path execution as well as loop unrolling techniques. However, predication alone is not enough in order to increase performance and predictability. The use of predication can actually increase the WCET. Because of this we propose the use of a hybrid approach with hardware support of a low-overhead predication system and static branch prediction. This leads to significant WCET reduction and allows compiler WCET optimization by selecting the appropriate technique.

8 FINAL REMARKS

The objective of this thesis was to investigate various processor architecture features that lead to a predictable design with reasonable WCET performance. The thesis demonstrated that it is possible to assemble together hardware elements that increase performance but are predictable enough to ensure efficient and precise analyses. This is highly motivated by the fact that despite the differences between general-purpose and embedded computing, the addition of new functionality such as audio/video processing, networking, security, automation and control, among others, increases the need for more advanced processors for embedded applications. However, deterministic hardware is necessary when the application has hard real-time requirements. We can also note that safety standards for embedded applications (DO-178B, DO-178C, IEC-61508, ISO-26262, EN-50125, etc.) require identifying potential functional and non-functional hazards and we must demonstrate that the software comply with safety goals. Testing with intensive measurements is tedious and also typically not safe. It is often impossible to prove that the worst-case conditions have actually been taken into account. As DO-178B succinctly puts it, “testing, in general, cannot show the absence of errors.”

Therefore, it is necessary to analyze the strengths of hardware techniques used in state-of-the-art processors, for example temporal and spatial parallelism (pipeline), branch prediction and predication, and to align them to real-time applications. Some of these techniques require modifications due to high hardware complexity while others need a well-defined timing behavior. In order to achieve these goals, we propose new approaches in order to improve the efficiency and scalability of temporal analysis, especially the worst-case execution time analysis. We showed that the design of processors should increase the level of importance of the determinism. In this thesis, we focus on doing so considering several processor architecture features that lead to predictable design but also improving the WCET performance.

Supporting our objective to investigate and evaluate hardware features which reconcile determinism and performance, we made the following contributions:

- Design space investigation and evaluation regarding VLIW processors.
- Complete WCET analysis for the proposed design.
- Complete VHDL design and timing characterization.
- Detailed branch architecture.
- Low-overhead full-predication system for VLIW processors.

Regarding the design-space investigation and performance evaluation, we made an intensive investigation of the literature analyzing the proposed techniques for real-time processors. This investigation lead us to some of our design decisions but there was a lack of performance evaluations. We knew, for instance, that the use of predication is interesting for real-time processors but how much this technique contributes to WCET performance enhancement and how hard the hardware is pushed to support a generic predication system were questions to be answered. Same doubts turned up on branch prediction as well. Some works propose to not use branch prediction at all but how much this hurts pipeline performance? In order to answer these questions, we made an extensive evaluation including the impact of the compiler on the performance which is specially important for processors with wide-fetch, in-order execution and static instruction scheduler. Other experiments were made to evaluate our design decisions considering wider fetch, the use or not of branch prediction and predication and the benefits of dual-port memory access. Most of them lead to significant WCET reduction while others need more compiler support to be fully explored.

It is very important to have a WCET analyzer to infer proprieties and make performance evaluations. In the context of this work, we used state-of-the-art techniques, like the Implicit Path Enumeration Technique (IPET) for worst-case path search, and proposed new strategies in order to estimate WCET of programs running on our VLIW platform. Execution of various paths while using predication was modeled during pipeline analysis as well as the timing of multi-cycle operations. Control-flow instruction timing was also modeled in conjunction with IPET by using different transition flows (δ) for each type of basic block transition. Cache memory behavior was also considered by the WCET analyzer and, even being a single direct-mapped cache, the modeling of its timing was very challenging specially during validation due to complex control-flow structures such as nested loops. We can also note that an integrated data flow between the compiler and the analyzer illustrated by our tool is very important when implementing a WCET analyzer. Much of the data required for WCET estimation is always produced by the compiler, specially the control-flow graph and loop annotations, and an integration of these tools makes WCET computation more efficient and precise. With our WCET analyzer and our design space exploration we could identify quantitatively how much hardware techniques contribute to WCET reduction as well as how to analyze them during WCET static estimation. We concluded that the knowledge of a WCET analyzer is important in order to achieve a predictable hardware design and that an integrated environment between WCET and compiler enhances WCET estimation.

We made also a complete VHDL implementation of the proposed design. This step was important to infer the timing behavior and the hardware complexity of each module. The use of a hardware description language to implement the design was very challenging. In order to reach the desired clock speed and to use the proper hardware design methodology, most of the modules were implemented synchronously. Some of them are naturally implemented this way, but the pipeline front-end required considerable levels of logic with various states in its finite state machine. Large components, like the arithmetic units and its forwarding logic have to be optimized in the VHDL code level and they were pinned to specific FPGA regions in order to reach the desired performance. The processor researched in this thesis was a 4-wide fetch VLIW microprocessor with 32-bit RISC operations and it was synthesized on an Altera Cyclone IV GX (EP4CGX150DF31C7) in a DE2i-150 development board. We targeted and reached a clock speed of 100Mhz and it required a total of 21,220 of 149,760 (14%) FPGA combination function resources. In terms of FPGA memory utilization, it required a total of 1,188,764 of 6,635,520 (15%) memory bits.

Branch architecture has always generated discussion and research in computer architectures. We showed that static branch prediction is very important to increase the WCET performance of real-time systems especially for loops which are very predictable. In special, we proved that the use of static branch predication does not add significant hardware complexity and its WCET analysis is feasible. We must only model different basic block transition flows. Since WCET analysis is feasible and it reduces the WCET for more than half of the benchmarks when the hardware supports both "taken" and "not-taken" predictions, we cannot ignore the use of branch prediction for real-time systems. We also noted that the WCET reduction depends on the code structure and up to 12% of reduction could be achieved even without a WCET-oriented static branch prediction.

Predication has become important for real-time systems because it can improve performance and determinism as well, but it adds more complexity to hardware and software. Generic full predication adds considerable levels of logic to critical processor data paths, specially in instruction encoding and forwarding paths. The compiler must also be designed to use this technology with if-conversions. In terms of hardware, we proposed a low-overhead full-predication mechanism in order to extend the real-time capabilities of the processor. We fixed the predicate register, we used only one operation encoding bit and we added two operation modes assisted by special ISA instruction. With these features, we added full predication support, including complex nested structures using full and partial predication, without adding latencies to critical processor data paths. The support of predication could reduce the WCET of some benchmarks and for some of them this technique reduced the WCET by about 15%. However, predication alone is not enough in order to increase performance and predictability. The use of predication may increase WCET. Because of this we propose the use of a hybrid approach with hardware support of a low-overhead predication system and static branch prediction. This leads to a significant WCET reduction and allows compiler WCET optimization by selecting the appropriate technique.

We believe we have achieved our objectives contributing to the state of the art by providing a clear evaluation of the proposed techniques as well as a complete framework for further real time researches. We have also secured our low-level requirements focusing on hard real-time systems but also not forgetting the high-level requirements allowing our design to be applicable to modern systems using modern ISA and a modern compiler back-end. The VLIW ST231 ISA operation encoding provided a clear and well defined instruction encoding, adequate set of arithmetic instructions, typical memory access instruction with base plus offset addressing mode and allowed the use of function calling/returning. This enhanced the construction of the compiler back-end as well as the decoding during the WCET analysis.

8.1 PUBLICATIONS

This work generated the following publications in conference proceedings:

1. Starke, R. A., Carminati, A., & de Oliveira, R. S. (2015). ***Investigating a four-issue deterministic VLIW architecture for real-time systems.*** In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN) (pp. 215–220). Cambrigde - UK: IEEE. http://doi.org/10.1109/INDIN.2015.7281737

2. Silva, K. P., Starke, R. A., & de Oliveira, R. S. (2015). ***Value analysis for the determination of memory instruction latency in a WCET tool.*** In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN) (pp. 252–257). IEEE. http://doi.org/10.1109/INDIN.2015.7281743

Publication in journals:

1. Starke, R. A., Carminati, A., & de Oliveira, R. S. *Evaluating the Design of a VLIW Processor for Real-Time Systems.* ACM Transactions on Embedded Computing Systems. Accepted on December 7th, 2015. Publication on Vol. 15, No. 3, Article 46. DOI: http://dx.doi.org/10.1145/2889490.

Submitted for publication:

1. Starke, R. A., Carminati, A., & de Oliveira, R. S. *Evaluation of a Low Overhead Predication System for a Deterministic VLIW Architecture Targeting Real-Time Applications.* Microprocessors and Microsystems. Submitted on October 29th, 2015.

2. Carminati, A., Starke, R. A., & de Oliveira, R. S. *On implementing a WCET analyzer with data integration for a compositional architecture.* Springer Design Automation for Embedded Systems. Submitted on August 25th, 2015.

3. Carminati, A., Starke, R. A., & de Oliveira, R. S. *On the Use of Static Branch Prediction to Reduce the Worst-Case Execution Time of Real-Time Applications*. Journal of System and Software. Submitted on January 6th, 2016.

8.2 SUGGESTIONS OF FUTURE WORK

We believe there are several directions in which this research could be continued.

In terms of hardware, the first research is the support of DMA (Direct Memory Access) operations between memories. This feature will allow the use of a memory hierarchy but its design must consider deterministic properties like other processor features. On supporting large memories with a memory hierarchy, the compiler and the WCET analyzer should be modified to include the use and analysis of such features. The use of other types of memories in the first level could be also researched: an instruction scratchpad, a data cache memory or semantic caches like the Patmos approach.

In order to support a complete real-time application with a real-time operating system, peripherals like timers and a vectored interrupt controller should also be included in the design. This allows avionics or automotive applications to be used as case studies inferring proprieties and helping to increase the performance, efficiency and precision of the framework.

In terms of WCET analyzer, a value analysis could be considered in order to detect unfeasible paths and loop bounds. This technique will greatly increase the WCET precision and safety. Cache analysis could also be improved with a better *first miss* detection and the support of associative caches with Least Recent Use (LRU) eviction policy. We could also investigate new methods in order to consider processor timing properties into scheduling equations (e.g. context-switch overhead and shared data).

Working together with the compiler development, the instruction per cycle (IPC) could also be increased with a global instruction scheduler. This also allows a more efficient multi-stage operation dependency resolution not stalling the pipeline for all cycles. Our current compiler back-end implementation does not consider any Read-After-Write dependencies between pipeline stages and a more sophisticated hardware dependency resolution will only increase hardware complexity. If multi-stage operations could be re-scheduled and reordered, a hardware mechanism could detect the necessary time to satisfy dependencies instead of stalling all cycles every time a multi-stage operation is executed. This can also be implemented splitting multiplication and division in two parts like the MIPS architecture: first, calculation is done and its result is read later (a special instruction reads the result after the calculation). If we use this strategy, pipeline stall is not necessary but the compiler or the hardware must be modified in order to keep semantics if a result is requested before it is ready.

We can also increase our performance evaluation considering average-case performance generating random input parameters. Since WCET analysis tools for all related work are not available, we can compare our design with others considering small assembly code snippets.

BIBLIOGRAPHY

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. [S.l.]: Pearson Education, 2011. ISBN 9780133002140. 142

ALT, M. et al. Cache behavior prediction by abstract interpretation. In: *Static Analysis*. [S.l.: s.n.], 1996. p. 52–66. 145

BANAKAR, R. et al. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, Acm, p. 73–78, 2002. 62

BURGUIERE, C.; ROCHANGE, C.; SAINRAT, P. A Case for Static Branch Prediction in Real-Time Systems. In: *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. IEEE, 2005. p. 33–38. ISBN 0-7695-2346-3. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1541053>. 14, 39, 57, 171

CHOI, Y. et al. The Impact of If-conversion and Branch Prediction on Program Execution on the Intel&Reg; Itanium&Trade; Processor. In: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2001. (MICRO 34), p. 182–191. ISBN 0-7695-1369-7. Disponível em: <http://dl.acm.org/citation.cfm?id=563998.564023>. 58

CULLMANN, C. et al. Predictability considerations in the design of multi-core embedded systems. *Ingenieurs de l'Automobile*, v. 807, p. 36–42, 2010. 141

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, v. 43, n. 4, p. 1–44, out. 2011. ISSN 03600300. 33

EDWARDS, S. A.; LEE, E. A. The case for the precision timed (PRET) machine. In: *Proceedings of the 44th annual conference on Design automation - DAC '07*. New York, New York, USA: ACM Press, 2007. p. 264. ISBN 9781595936271. ISSN 0738100X. 32

ENGBLOM, J.; JONSSON, B. Processor pipelines and their properties for static WCET analysis. In: *Embedded Software*. [s.n.], 2002. p. 334–348. ISBN 3-540-44307-X. Disponível em: <http://www.springerlink.com/index/J9C2BLLEWB0LMWC1.pdf>. 153

FISHER, J. A.; FARABOSHI, P.; YOUNG, C. *Emebedded Computing: A VLIW Approach to Architecture Compilers and Tools*. [S.l.]: Morgan Kaufmann Publishers, 2005. 709 p. ISBN 1558607668. 13, 38, 55, 58, 59, 83, 85, 125, 166, 167

FISHER, J. A.; MEMBER, S. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30, n. 7, p. 478–490, jul. 1981. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1675827>. 106

GEYER, C. B. et al. Time-predictable code execution - Instruction-set support for the single-path approach. In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. IEEE, 2013. p. 1–8. ISBN 978-1-4799-2111-9. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6913195>. 60

GROSSMAN, J. P. *Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation*. [S.l.], 2000. Disponível em: <http://www.ai.mit.edu/projects/aries/>. 59

GRUND, D.; REINEKE, J.; WILHELM, R. A Template for Predictability Definitions with Supporting Evidence. In: LUCAS, P. et al. (Ed.). *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011. (OpenAccess Series in Informatics (OASIcs), v. 18), p. 22–31. ISBN 978-3-939897-28-6. ISSN 2190-6807. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2011/3078>. 35

GUSTAFSSON, J. Usability Aspects of WCET Analysis. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, Ieee, p. 346–352, maio 2008. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4519598>. 12

GUSTAFSSON, J. et al. The M{ä}lardalen WCET Benchmarks - Past, Present and Future. In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*. [s.n.], 2010. Disponível em: <http://www.es.mdh.se/publications/1895->. 22, 40, 82, 165, 166

HWU, W. M. W. et al. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, v. 7, n. 1-2, p. 229–248, maio 1993. ISSN 0920-8542. Disponível em: <http://link.springer.com/article/10.1007/BF01205185http://link.springer.com/10.1007/BF01205185>. 105

JORDAN, A.; KIM, N.; KRALL, A. IR-level versus machine-level if-conversion for predicated architectures. In: *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems - ODES '13*. New York, New York, USA: ACM Press, 2013. p. 3. ISBN 9781450319058. Disponível em: <http://dl.acm.org/citation.cfm?doid=2443608.2443611>. 108

KIRNER, R.; PUSCHNER, P. Time-predictable computing. In: *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*. Berlin, Heidelberg: Springer-Verlag, 2010. (SEUS'10), p. 23–34. ISBN 3-642-16255-X, 978-3-642-16255-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1927882.1927890>. 35

KREUZINGER, J. et al. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, v. 27, n. 1, p. 19–31, 2003. ISSN 01419331. 67

LAFOREST, C.; STEFFAN, J. Efficient multi-ported memories for FPGAs. *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays*, ACM Press, New York, New York, USA, p. 41, 2010. 123, 138

LAFOREST, C. E. et al. Multi-ported memories for FPGAs via XOR. *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*, ACM Press, New York, New York, USA, p. 209, 2012. Disponível em: <http://dl.acm.org/citation.cfm?doid=2145694.2145730>. 123

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004. p. 75–86. ISBN 0-7695-2102-9. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281665>. 40, 87, 105, 134, 165

LEE, C.-g. et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, v. 47, n. 6, p. 700–713, jun. 1998. ISSN 00189340. 145

LENGAUER, T.; TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, v. 1, n. 1, p. 121–141, jan. 1979. ISSN 01640925. Disponível em: <http://portal.acm.org/citation.cfm?doid=357062.357071>. 142

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, v. 30, n. 11, p. 88–98, nov. 1995. ISSN 03621340. 142, 154, 155

LIU, I. et al. A PRET microarchitecture implementation with repeatable timing and competitive performance. *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Ieee, p. 87–93, set. 2012. 12, 63, 71, 83, 109

LIU, I.; REINEKE, J.; LEE, E. a. A PRET architecture supporting concurrent programs with composable timing properties. *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, Ieee, p. 2111–2115, nov. 2010. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5757922>. 71

LIU, J. W. S. *Real-Time Systems*. 1. ed. [S.l.]: Prentice Hall, 2000. ISBN 0-13-099651-3. 32

LUNDQVIST, T.; STENSTROM, P. Timing anomalies in dynamically scheduled microprocessors. *Proceedings 20th IEEE Real-Time Systems Symposium*, IEEE Comput. Soc, p. 12–21, 1999. 13, 34, 37, 38

MAHLKE, S. et al. A comparison of full and partial predicated execution support for ILP processors. *Proceedings 22nd Annual International Symposium on Computer Architecture*, p. 138–149, 1995. ISSN 1063-6897. 58

MAKHORIN, A. GLPK (GNU linear programming kit). 2008. 156, 162

MARTÍNEZ, P. L.; CUEVAS, C.; DRAKE, J. M. Compositional real-time models. *Journal of Systems Architecture*, v. 58, n. 6-7, p. 257–276, jun. 2012. ISSN 13837621. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S138376211200029X>. 36

MISCHE, J. et al. How to enhance a superscalar processor to provide hard real-time capable in-order SMT. In: *Lecture Notes in Computer Science*. [S.l.: s.n.], 2010. v. 5974 LNCS, p. 2–14. ISBN 3642119492. ISSN 03029743. 73, 74

MUELLER, F.; WHALLEY, D. Fast instruction cache analysis via static cache simulation. In: *Proceedings of Simulation Symposium*. [S.l.]: IEEE Comput. Soc. Press, 1995. p. 105–114. ISBN 0-8186-7091-6. 145

OTTOSSON, G.; SJöDIN, M. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In: *In Proc. of SIGPLAN 1997 Workshop* *on Languages, Compilers and Tools for Real-Time Systems; LCT-RTS97*. [s.n.], 1997. Disponível em: <http://www.es.mdh.se/publications/18->. 154

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Revised Printing, Third Edition: The Hardware/Software Interface*. [S.l.]: Elsevier Science, 2007. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780080550336. 85

PUSCHNER, P. Experiments with WCET-Oriented Programming and the Single-Path Architecture. *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Ieee, p. 205–210, 2005. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1544795>. 15, 39, 84

PUSCHNER, P.; KIRNER, R.; PETTIT, R. G. Towards Composable Timing for Real-Time Programs. *2009 Software Technologies for Future Dependable Distributed Systems*, Ieee, n. 214373, p. 1–5, mar. 2009. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4804562>. 36, 37

REINEKE, J. et al. PRET DRAM controller. In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '11*. New York, New York, USA: ACM Press, 2011. p. 99. ISBN 9781450307154. Disponível em: <http://dl.acm.org/citation.cfm?doid=2039370.2039388>. 14, 38, 72, 103, 140

SANTOS, R.; AZEVEDO, R.; ARAUJO, G. 2D-VLIW: An Architecture Based on the Geometry of Computation. In: *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*. IEEE, 2006. p. 87–94. ISBN 0-7695-2682-9. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4019496>. 125

SCHOEBERL, M. A Time Predictable Instruction Cache for a Java Processor. In: *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*. Springer, 2004. v. 3292, p. 371–382. Disponível em: <http://www.jopdesign.com/doc/jtres_cache.pdf>. 69

SCHOEBERL, M. Design and Implementation of an Efficient Stack Machine. In: *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*. IEEE, 2005. Disponível em: <http://www.jopdesign.com/doc/stack.pdf>. 70

SCHOEBERL, M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, v. 54, p. 265–286, 2008. ISSN 1383-7621. Disponível em: <http://www.jopdesign.com/doc/rtarch.pdf>. 69, 70, 72

SCHOEBERL, M. Time-predictable cache organization. In: *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*. [S.l.: s.n.], 2009. 102

SCHOEBERL, M. Time-Predictable Computer Architecture. *EURASIP Journal on Embedded Systems*, v. 2009, p. 1–17, 2009. ISSN 1687-3955. 31, 32, 61, 63, 76

SCHOEBERL, M. Is time predictability quantifiable? *2012 International Conference on Embedded Computer Systems (SAMOS)*, Ieee, p. 333–338, jul. 2012. 35

SCHOEBERL, M. et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, Elsevier B.V., 2015. ISSN 13837621. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1383762115000193>. 12, 63, 73, 74, 75, 79

SCHOEBERL, M. et al. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, v. 40/6, p. 507–542, 2010. Disponível em: <http://www.jopdesign.com/doc/wcetana.pdf>. 70

SCHOEBERL, M. et al. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In: LUCAS, P. et al. (Ed.). *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011. (OpenAccess Series in Informatics (OASIcs), v. 18), p. 11–21. ISBN 978-3-939897-28-6. ISSN 2190-6807. 12, 14, 32, 38, 72, 75, 83, 102

SHEN, J. P. et al. *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005. ISBN 0-07-059033-8. Disponível em: <http://opac.inria.fr/record=b1129703>. 19, 43, 44, 46, 47, 53, 56, 57, 91

THIELE, L.; WILHELM, R. Design for Timing Predictability. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 28, n. 2-3, p. 157–177, nov. 2004. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e>. 35

UHRIG, S.; MISCHE, J.; UNGERER, T. An IP Core for Embedded Java Systems. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. [s.n.], 2007. p. 263–272. Disponível em: <http://link.springer.com/10.1007/978-3-540-73625-7_28>. 67, 68, 73

UHRIG, S.; WIESE, J. jamuth: an IP processor core for embedded Java real-time systems. In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems - JTRES '07*. [s.n.], 2007. p. 230. ISBN 9781595938138. Disponível em: <http://dl.acm.org/citation.cfm?doid=1288940.1288974>. 68, 69

UNGERER, T. et al. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, v. 30, n. 5, p. 66–75, set. 2010. ISSN 0272-1732. 74, 75

WENZEL, I. et al. Principles of Timing Anomalies in Superscalar Processors. In: *Fifth International Conference on Quality Software*. [S.l.]: IEEE, 2005. p. 295–306. ISBN 0-7695-2472-9. 33, 34

WHITHAM, J.; AUDSLEY, N. MCGREP–A Predictable Architecture for Embedded Real-Time Systems. *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, Ieee, p. 13–24, 2006. 70, 71

WHITHAM, J.; AUDSLEY, N. Investigating Average versus Worst-Case Timing Behavior of Data Caches and Data Scratchpads. *2010 22nd Euromicro Conference on Real-Time Systems*, Ieee, p. 165–174, jul. 2010. 62

WHITHAM, J. et al. Investigation of Scratchpad Memory for Preemptive Multitasking. *2012 IEEE 33rd Real-Time Systems Symposium*, Ieee, p. 3–13, dez. 2012. 62

WILHELM, R. et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, v. 7, n. 3, p. 1–53, abr. 2008. ISSN 15399087. 31, 32

Appendix

APPENDIX A – LIST OF SUPPORTED OPERATIONS

Operand	Encoding Bits	Type
bcond	25..23	Predicate ($\$br$)
bdest	20..18	Predicate ($\$br$)
btarg	22..0	Immediate
dest	17..12	Register ($\$r$)
idest	11..6	Register ($\$r$)
isrc2	20..12	Immediate
scond	23..21	Predicate ($\$br$)
src1	5..0	Register ($\$r$)
src2	11..6	Register ($\$r$)
pc		Program counter

Table 31 – List of operation operands

Mnemonic	Semantic
ADD	dest = *signext32* (src1) + *signext32* (src2)
ADD_I	idest = *signext32* (src1) + *signext32* (isrc2)
SUB	dest = *signext32* (src1) - *signext32* (src2)
SUB_I	idest = *signext32* (src1) - *signext32* (isrc2)
SHL	dest = *signext32* (src1) $\ll$ src2 (4..0)
SHL_I	idest = *signext32* (src1) $\ll$ *signext32* (isrc2 (4..0))
SHR	dest = *signext32* (src1) $\gg$ src2 (4..0)
SHR_I	idest = *signext32* (src1) $\gg$ *signext32* (isrc2 (4..0))
SHRU	dest = *signext32* (src1) $\gg$ src2 (4..0)
SHRU_I	idest = *signext32* (src1) $\gg$ *signext32* (isrc2 (4..0))
AND	dest = *signext32* (src1) & *signext32* (src2)
AND_I	idest = *signext32* (src1) & *signext32* (isrc2)
ANDC	dest = *signext32* (src1) & *signext32* (src2)

ANDC_I	idest = *signext32* (src1) & *signext32* (isrc2)
OR	dest = *signext32* (src1) | *signext32* (src2)
OR_I	idest = *signext32* (src1) | *signext32* (isrc2)
ORC	dest = *signext32* (src1) | *signext32* (src2)
ORC_I	idest = *signext32* (src1) | *signext32* (isrc2)
XOR	dest = *signext32* (src1) ^*signext32* (src2)
XOR_I	idest = *signext32* (src1) ^*signext32* (isrc2)
MAX	dest = *signext32* (src1) > *signext32* (src2) ? src1 : src2
MAX_I	idest = *signext32* (src1) > *signext32* (isrc2) ? src1 : isrc2
MAXU	dest = *zeroext32* (src1) > *zeroext32* (src2) ? src1 : src2
MAXU_I	idest = *zeroext32* (src1) > *zeroext32* (isrc2) ? src1 : isrc2
MIN	dest = *signext32* (src1) <*signext32* (src2) ? src1 : src2
MIN_I	idest = *signext32* (src1) <*signext32* (isrc2) ? src1 : isrc2
MINU	dest = *zeroext32* (src1) <*zeroext32* (src2) ? src1 : src2
MINU_I	idest = *zeroext32* (src1) <*zeroext32* (isrc2) ? src1 : isrc2
SXTB	idest = *signext32* (src1(7..0))
SXTH	idest = *signext32* (src1(15..0))
ZXTB	idest = *zeroext32* (src1(7..0))
ZXTH	idest = *zeroext32* (src1(15..0))
ADDCG	dest = *signext32* (src1) + *signext32* (src2) + *zeroext1*(scond) ; bdest = carry bit
SUBCG	dest = *signext32* (src1) - *signext32* (src2) + *zeroext1*(scond) – 1 ; bdest = carry bit
CMPEQ_R	dest(1) = *signext32* (src1) == *signext32* (src2)

CMPEQ_B	bdest = *signext32* (src1) == *signext32* (isrc2)
CMPEQ_IR	dest(1) = *signext32* (src1) == *signext32* (isrc2)
CMPEQ_IB	bdest = *signext32* (src1) == *signext32* (isrc2)
CMPNE_R	dest(1) = *signext32* (src1) == *signext32* (src2)
CMPNE_B	bdest = *signext32* (src1) != *signext32* (isrc2)
CMPNE_IR	dest(1) = *signext32* (src1) != *signext32* (isrc2)
CMPNE_IB	bdest = *signext32* (src1) != *signext32* (isrc2)
CMPGE_R	dest(1) = *signext32* (src1) >= *signext32* (src2)
CMPGE_B	bdest = *signext32* (src1) >= *signext32* (isrc2)
CMPGE_IR	dest(1) = *signext32* (src1) >= *signext32* (isrc2)
CMPGE_IB	bdest = *signext32* (src1) >= *signext32* (isrc2)
CMPGEU_R	dest(1) = *zeroext32* (src1) >= *zeroext32* (src2)
CMPGEU_B	bdest = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGEU_IR	dest(1) = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGEU_IB	bdest = *zeroext32* (src1) >= *zeroext32* (isrc2)
CMPGT_R	dest(1) = *signext32* (src1) > *signext32* (src2)
CMPGT_B	bdest = *signext32* (src1) > *signext32* (isrc2)
CMPGT_IR	dest(1) = *signext32* (src1) > *signext32* (isrc2)
CMPGT_IB	bdest = *signext32* (src1) > *signext32* (isrc2)
CMPGTU_R	dest(1) = *zeroext32* (src1) > *zeroext32* (src2)
CMPGTU_B	bdest = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPGTU_IR	dest(1) = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPGTU_IB	bdest = *zeroext32* (src1) > *zeroext32* (isrc2)
CMPLE_R	dest(1) = *signext32* (src1) <= *signext32* (src2)
CMPLE_B	bdest = *signext32* (src1) <= *signext32* (isrc2)

CMPLE_IR	dest(1) = *signext32* (src1) <= *signext32* (isrc2)
CMPLE_IB	bdest = *signext32* (src1) <= *signext32* (isrc2)
CMPLEU_R	dest(1) = *zeroext32* (src1) <= *zeroext32* (src2)
CMPLEU_B	bdest = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLEU_IR	dest(1) = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLEU_IB	bdest = *zeroext32* (src1) <= *zeroext32* (isrc2)
CMPLT_R	dest(1) = *signext32* (src1) <*signext32* (src2)
CMPLT_B	bdest = *signext32* (src1) <*signext32* (isrc2)
CMPLT_IR	dest(1) = *signext32* (src1) <*signext32* (isrc2)
CMPLT_IB	bdest = *signext32* (src1) <*signext32* (isrc2)
CMPLTU_R	dest(1) = *zeroext32* (src1) <*zeroext32* (src2)
CMPLTU_B	bdest = *zeroext32* (src1) <*zeroext32* (isrc2)
CMPLTU_IR	dest(1) = *zeroext32* (src1) <*zeroext32* (isrc2)
CMPLTU_IB	bdest = *zeroext32* (src1) <*zeroext32* (isrc2)
ANDL_R	dest(1) = *zeroext32* (src1) && *zeroext32* (src2)
ANDL_B	bdest = *zeroext32* (src1) && *zeroext32* (isrc2)
ANDL_IR	dest(1) = *zeroext32* (src1) && *zeroext32* (isrc2)
ANDL_IB	bdest = *zeroext32* (src1) && *zeroext32* (isrc2)
NANDL_R	dest(1) = (*zeroext32* (src1) && *zeroext32* (src2))
NANDL_B	bdest = (*zeroext32* (src1) && *zeroext32* (isrc2))
NANDL_IR	dest(1) = (*zeroext32* (src1) && *zeroext32* (isrc2))
NANDL_IB	bdest = (*zeroext32* (src1) && *zeroext32* (isrc2))
NORL_R	dest(1) = (*zeroext32* (src1) || *zeroext32* (src2))

NORL_B	bdest = ~(*zeroext32* (src1) || *zeroext32* (isrc2))
NORL_IR	dest(1) = ~(*zeroext32* (src1) || *zeroext32* (isrc2))
NORL_IB	bdest = ~(*zeroext32* (src1) || *zeroext32* (isrc2))
ORL_R	dest(1) = *zeroext32* (src1) || *zeroext32* (src2)
ORL_B	bdest = *zeroext32* (src1) || *zeroext32* (isrc2)
ORL_IR	dest(1) = *zeroext32* (src1) || *zeroext32* (isrc2)
ORL_IB	bdest = *zeroext32* (src1) || *zeroext32* (isrc2)
SLCT_R	dest = (scond == 1 ? src1 : src2)
SLCT_I	dest = (scond == 1 ? src1 : isrc2)
SLCTF_R	dest = (scond == 0 ? src1 : src2)
SLCTF_I	dest = (scond == 0 ? src1 : isrc2)
BR	pc = (scond == 1 ? *signext32* (pc) + *signext32* (btarg))
BRF	pc = (scond == 0 ? *signext32* (pc) + *signext32* (btarg))
MULL	Dest = (32 lower bits) *signext32* (src1) * *signext32* (src2)
MULL64H	Dest = (32 higher bits) *signext32* (src1) * *signext32* (src2)
MULL64HU	Dest = (32 higher bits) *signext32* (src1) * *signext32* (src2)
DIV_R	dest = *signext32* (src1) % *signext32* (src2)
DIV_Q	dest = *signext32* (src1) / *signext32* (src2)
DIV_RU	dest = *zeroext32* (src1) % *zeroext32* (src2)
DIV_QU	dest = *zeroext32* (src1) / *zeroext32* (src2)
CALL	pc = src1 R63 = pc
ICALL	pc = btarg R63 = pc
GOTO	pc == *signext32* (pc) + *signext32* (btarg)
IGOTO	pc = *signext32* (pc) + *signext32* (btarg)
IMML	imm for previous operation is: *signext32*(btarg $\ll$ 9 + isrc2)

IMMR	imm for next operation is: *signext32*(btarg $\ll$ 9 + isrc2)
LDW	idest = *signext32* (mem [*signext32* (src1) + *signext32* (isrc2))
LDH	idest = *signext32*(mem [*signext32* (src1) + *signext32* (isrc2) (15..0))
LDHU	idest = mem [*signext32* (src1) + *signext32* (isrc2) (15..0)
LDB	idest = *signext32*(mem [*signext32* (src1) + *signext32* (isrc2) (7..0))
LDBU	idest = mem [*signext32* (src1) + *signext32* (isrc2) (7..0)
STW	mem [*signext32* (src1) + *signext32* (isrc2)) = src2
STH	mem [*signext32* (src1) + *signext32* (isrc2)) = src2 (15..0)
STB	mem [*signext32* (src1) + *signext32* (isrc2)) = src2 (7..d0)
HALT	halt the cpu
PAR_ON	enable predication complete mode
PAR_OFF	disable predication complete mode
PRELD	direct next branch to (*signext32* (pc) + *signext32* (btarg))

Table 32 – List of supported operations