Aalto-yliopisto
Aalto-universitetet
Aalto University

Master’s Programme in Electronics and Nanotechnology

Multithreading in Application Specific Instruction Set Processor

Doan Nguyen

Master’s Thesis
2022

Aalto-yliopisto
Aalto-universitetet
Aalto University

Author Doan Ha Nguyen		
Title of thesis Multithreading in Application Specific Instruction Set Processor		
Programme Master Program in Electronics and Nanotechnology		
Major Micro and Nano-electronic Design		
Thesis supervisor Prof. Jussi Ryynänen		
Thesis advisor MSc Heikki Kultala		
Collaborative partner Nokia Networks		
Date 11.11.2022	**Number of pages** 75	**Language** English

Abstract

Application Specific Instruction Set Processor (ASIP) is an application domain-specific processor designed to obtain the maximum performance when executing a domain of particular applications. ASIPs have been increasingly studied and deployed into SoC chips needed for fifth-generation (5G) wireless networks and even potential sixth-generation. That fact has led to a need for better performance, energy efficiency, and application flexibility. Furthermore, different architectures could be deployed on top of an ASIP to achieve a better performance ASIP. This work mentions three models: super pipelining, superscalar, and Very Long Instruction Word (VLIW). Each model is studied, resulting in advantages and disadvantages for each model in the ASIP deployment perspective. As a result, the author selects a VLIW model for the implementation. After that, the author performs a further performance study on the VLIW-based ASIP with multithreading. Multithreading is a traditional technique for tolerating latencies, improving functional unit utilization, and increasing instruction throughput. There are three models in multithreading: coarse-grained, simultaneous, and fine-grained. These models are studied, and a fine-grained multithreading model is chosen for the final implementation. In fine-grained multithreading, the processor switches to a new thread every cycle in a round-robin fashion. The implementation includes numerous changes to the existing register file, instructions, fetching, and controlling logic of the VLIW-based ASIP. In addition, the implementation introduces some new instructions and intrinsics to support writing programs for the multithreaded core. Benchmarking activity includes two tests for parsing ORAN messages. The results show that with multithreading when running these tests, there are significant speed-up in processing time (> 20% for the control plane and >10% for the user plane) in the case of a high number of sections per processed message. Regarding synthesis results, fine-grained multithreading implementation leads to an increase of 72% in the total processor area.

Keywords ASIP, processor, 5G, SoC, performance, instruction throughput, energy efficiency, application flexibility, architecture models, VLIW, multithreading, fine-grained multithreading, ORAN, message section.

Contents

Acknowledgements 8
Abbreviations 9
Abbreviations 9
1 Introduction 13
1.1 Problem Statement 13
1.2 Thesis Structure 14
2 ASIP Overview 15
2.1 ASIP and Benefits 15
2.2 RISC-V Instruction Set Architecture 18
2.2.1 Register File 19
2.2.2 RV32I Instruction Set 19
2.3 Basic RISC-V Core Implementation 22
2.4 Basic RISC-V Implementation with Pipeline 24
2.5 Pipeline Hazards 26
2.5.1 Data Hazards 26
2.5.2 Structural Hazards 29
2.5.3 Control Hazards 31
3 Investigation of Feasible Performance Improvement Method 33
3.1 Functional Unit Utilization and Issue Bandwidth Waste 33
3.2 Super Pipelining Processor 34
3.3 Multi-issue Processor 35
3.3.1 Superscalar 35
3.3.2 VLIW 36
3.4 Investigation Summary and Choice of Architecture 38
3.5 pktAsip core 40
3.5.1 Overview 40
3.5.2 pktAsip Pipeline 41
3.5.3 Fetching Logic and Processor Controller 43
4 Hardware Threads and Multithreading 47
4.1 Hardware Threads 47
4.2 Types of Multithreading 48

4.2.1 Coarse-grained Multithreading 48
4.2.2 Simultaneous Multithreading 49
4.2.3 Fine-grained multithreading 50
4.2.4 Multithreading Summary and Model Choice 52
5 Implementation of fine-grain multithreading on VLIW core 54
5.1 Duplication of Registers and Register File 54
5.2 Modification to Fetch Logic and Controller 56
5.3 Instruction Modifications 58
5.4 Custom Instructions 59
5.4.1 thread_id instruction 59
5.4.2 state_change instruction 59
5.4.3 tk_set instruction 59
5.4.4 tk_clear instruction 60
5.5 Custom Intrinsics 60
5.5.1 thread_id() intrinsic 60
5.5.2 set_bit() intrinsic 60
5.5.3 tk_set() intrinsic 60
5.5.4 tk_clear() intrinsic 60
6 Benchmarking Results and Discussions 61
6.1 Pipeline View of Multithreaded Core 61
6.2 Message Queue Generator and Message Processing 61
6.3 Fast Control Plane Message Parsing 62
6.4 User Plane Message Parsing 64
6.5 Hardware Synthesis Result 65
6.6 Limitations 66
7 Summary and Potential Future Improvement 68
7.1 Summary 68
7.2 Future Improvements 69
7.2.1 Separation of Register File 69
7.2.2 Registering Loop Start Instruction 69
7.2.3 Sharing The Same Program Memory Between Multiple Threads 70
8 Reference 71

Appendix A – Register File 74

Acknowledgements

My acknowledgments and warmest thank to my instructor Mr. Heikki Kultala in the ASIP team at Nokia company for his thorough support and guidance during the thesis work.

I want to thank my whole family, especially my parents, for all the sacrifices they made for caring, raising, and providing me with a good education they could.

Espoo, 11 November 2022
Doan Ha Nguyen

Abbreviations

Abbreviations

AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
ASIP	Application Specific Instruction Set Processor
CGMT	Coarse-Grained Multi-Threading
DMA	Direct Memory Access
DSP	Digital Signal Processor
EOL	End-Of-Loop
FGMT	Fine-Grained Multi-Threading
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPC	Fetch Program Counter
FPGA	Field-Programmable Gate Array
GPP	General Purpose Processor
ID	Instruction Decode
IF	Instruction Fetch
IP	Intellectual Property
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
MIMO	Multiple Input Multiple Output
NOP	No-OP
ORAN	Open Radio Access Network
PC	Program Counter
PCU	Processor Controller Unit
PM	Program Memory
RAW	Read After Write
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-Threading
SoC	System-on-Chip
SP	Stack Pointer
TLP	Thread Level Parallelism
TSU	Thread Scheduling Unit
WAW	Write After Write
WAR	Write After Read
WB	Write-Back
VLIW	Very Long Instruction Word
5G	Fifth Generation Wireless Network

List of Figures

Figure 1: Power, Performance Efficiency and Application Flexibility of several hardware implementations. Retrieved and adapted from Synopsys [3] 16
Figure 2: RV32I instruction formats. Retrieved from RISC-V Instruction Set Manual [9] 19
Figure 3: Basic implementation of RISC-V core. Retrieved from [10] 22
Figure 4: Basic 5-stage pipeline implementation of a RISC-V core. Retrieved from [10] 24
Figure 5: Instructions are overlapped in a 5-stage pipeline 26
Figure 6: Example of RAW hazard 26
Figure 7: Data needed by subsequent instruction can be forwarded from prior one. Retrieved from [10] 28
Figure 8: Basic RISC-V core with forwarding. Retrieved from [10] 29
Figure 9: Structure hazard caused by same hardware resources shared between different pipeline stages 30
Figure 10: Structure hazard caused by multi-cycle unit 30
Figure 11: Control hazard when branch is taken 31
Figure 12: Control hazard when branch is not taken 32
Figure 13: Functional unit utilization 33
Figure 14: Vertical and horizontal waste of an ASIP with 4 issue slots 34
Figure 15: Intel Pentium 3 with 10 pipeline stages 35
Figure 16: Intel Pentium 4 with 20 pipeline stages 35
Figure 17: Simple 2-way superscalar pipeline 36
Figure 18: a 3-way VLIW pipeline. Note that unlike superscalar pipeline when each issue slot has its own instruction visible at IF stage. In VLIW case, the fetched instruction appears to be a wide one that contains one or more sub-instruction for the issue slots. 37
Figure 19: A 4-way (4 issue slots) VLIW core with each slot has its own ALU 40
Figure 20: Multiple Lengths Instructions 41
Figure 21: pktAsip Pipeline 42
Figure 22: Fetching Logic and Processor Controller. Described and adapted based on original Synopsys example fetch and controlling logic implementation [3] 44
Figure 23: Hardware Thread Component. In the case of pktAsip, each hardware thread contains a register file, some registers like PC, Stack Pointer (SP), or instruction fetch buffer 47
Figure 24: Single thread Processor vs Multithreaded Processor. Typically, a processor with more hardware threads results in a higher functional unit and issue slot utilization. 48
Figure 25: CGMT pipeline with 2 threads and 4 issue slots 49
Figure 26: SMT model with 4 threads and 4 issue slots 50

Figure 27: FGMT with 4 threads and 4 issue slots..51
Figure 28: Duplication of Register File. Note that in this figure, only Register File is duplicated 4 times while the number of FUs remain.......... 55
Figure 29: Modification to Fetch Logic and Control for FGMT implementation.. 56
Figure 30: The execution of Jal instruction is moved from ID to EX stage for decoder simplicity... 58
Figure 31: Neither data nor control hazard exist in this case as instructions belong to different threads with each thread has their own register file..... 59
Figure 32: Pipeline view of multithreaded core when running a test program 61
Figure 33: Message processing program with 4 threads 62
Figure 34: Fast control plane message parsing comparison when running on pktAsip and multithread pktAsip. 63
Figure 35: User Plane Message Parsing 64
Figure 36: RISC-V base integer register file structure................................. 74

List of Tables

Table 1: Investigation summary of different architecture models on ASIP and their implementations on ASIP perspective. 39
Table 2: Summary of different multithreading models and their implementations on pktAsip perspective. 52
Table 3: Synthesis Results Comparison between single-thread pktAsip and multithread pktAsip 65
Table 4: Summary Result 66

1 Introduction

Recently, the development of fifth-generation (5G) wireless networks has become more complex and time-consuming due to the evolution of 5G specifications. This development has led to complexity in developing System-on-Chip (SoC) used in 5G networks. As processor vendors have been competing to develop and release new processors for 5G networks, there is a need to reduce time-to-market. Unfortunately, the bottlenecks are caused by the lengthy development time required for SoC when defining the specifications, designing, and rigorous verification activities. Moreover, instead of waiting for a more mature standard, many chip makers have decided to opt for a more software-programmable solution in blocks that were previously implemented as fixed-function hardware [1] Fixed-function hardware is a type of processor designated for maximum performance and silicon cost, but this type of processor is non-programmable. Traditional processors used in many SoC for wireless and radio packet processors do not fulfill the required power, energy, and performance efficiency, nor do they fulfill application flexibility. Examples of these processors include general purpose processors (GPPs), Field Programmable Gate Arrays (FPGAs), or Application Specific Integrated Circuits (ASICs).

There is an alternative solution for the deployment of many 5G SoCs called Application Specific Instruction Set Processor (ASIP). ASIPS are superior in providing hardware specialization performance while retaining sufficient programmability and flexibility. In terms of designing an ASIP, processor designers are beneficial from the availability of several commercial tools, for example, Synopsys ASIP Designer, Codasip Studio, or an open-source toolset, such as TTA-based Co-Design Environment (TCE). These tools are typically equipped with specific templates, thus allowing ASIP designers to create their custom instructions and intrinsics and define the data path or different functional units on top of a basic processor template. These templates' availability helps designers avoid designing the processors from scratch, which is labor-intensive and time-consuming. According to [2] the deployment of ASIPs as key architectural blocks will offer an attractive solution and opportunity to reduce 5G chip development time and 5G system acceleration rollouts.

1.1 Problem Statement

As ASIPs are being increasingly deployed into SoC for communication systems, better ASIPs in terms of performance, instruction throughput, and reduced latency are needed. However, modern ASIPs typically experience many latency sources, including data and instruction cache misses, branch misprediction penalties, or pipeline hazards. The most common latency sources are pipeline hazards, including structural, data, and control hazards. These hazards downgrade the instruction throughput and the system's overall performance. One of the designer challenges when designing an ASIP is the ability to identify, tolerate and hide processor latencies. Traditionally, typical solutions for resolving pipeline hazards include additional hardware (for structural hazards), bypassing network (for data hazards), or branch predictor (for control hazards). However, these methods for resolving pipeline hazards are complex and expensive.

Nevertheless, the challenges and costs in resolving the pipeline hazards have led to a promising solution using a traditional multithreading technique. Multithreading is a popular architecture technique for tolerating and hiding different types of latencies and improving functional unit utilization in processors. Additionally, Very Long Instruction Word (VLIW) processors are popular solutions for exploiting instruction-level parallelism (ILP) to achieve high instruction throughput by exposing the internal structure to the compiler. An ASIP with much better performance can be implemented by combining with VLIW, allowing the issue and execution of multiple instructions simultaneously while sustaining a reasonable level of power and area cost.

Therefore, the objective of this thesis is to evaluate the feasibility and deploy a multithreading architecture model on top of a VLIW-based ASIP. To accomplish this objective, the thesis will perform different feasible performance studies on ASIP and then select one of the studies for implementation.

1.2 Thesis Structure

This thesis is structured as follows. **Chapter 2** addresses the definition, basic implementation of an ASIP, and ASIP-related pipeline hazards. **Chapter 3** performs a feasible performance investigation for ASIP on three different architecture models. This chapter also provides the readers with an overview of the VLIW-based ASIP core, namely pktAsip, used as the basis for feasible performance improvement. **Chapter 4** describes a number of multithreaded execution models and reasons for the chosen model for this work. **Chapter 5** explains the implementation of the chosen multithreading model on the top of pktAsip. **Chapter 6** presents the benchmarking results, some discussions about the results, and some limitations during the work implementation. **Finally, chapter 7** summarizes the work and addresses some potential future improvements for the processor.

2 ASIP Overview

Section 2.1 provides the audience with an overview of an ASIP and reasons ASIPs advantages as well as explains the purpose of why ASIPs are being deployed in many SoCs.

After that, before going to the feasibility of some techniques used for improving ASIP performance, it is beneficial for the readers to understand the RISC-V Instruction Set Architecture (ISA) described in **section 2.2** as later on in this thesis, some modifications will be done to the units that executes one of those instructions listed in the ISA as well as the Register File. Further, some basic implementations of RISC-V Core, such as non-pipeline and pipeline version with related hazards, are listed in **sections 2.3, 2.4, and 2.5**. They are described with the primary purpose of getting readers familiarized with pipeline stages and internal components inside a basic RISC-V core as later, the target ASIP core with further modifications used in this work for the study and implementation are built on top of a similar 5-stage RISC-V core.

2.1 ASIP and Benefits

Application Specific Instruction Set Processor (ASIP) is an application domain-specific processor usually used in System-on-Chip (SoC). It is a processor designed to obtain a high performance when executing a particular application while retaining a sufficient level of software programmability flexibility. The assembly instruction set is tailored to benefit the execution of specific applications: it supports the acceleration of complex and frequent instructions execution.

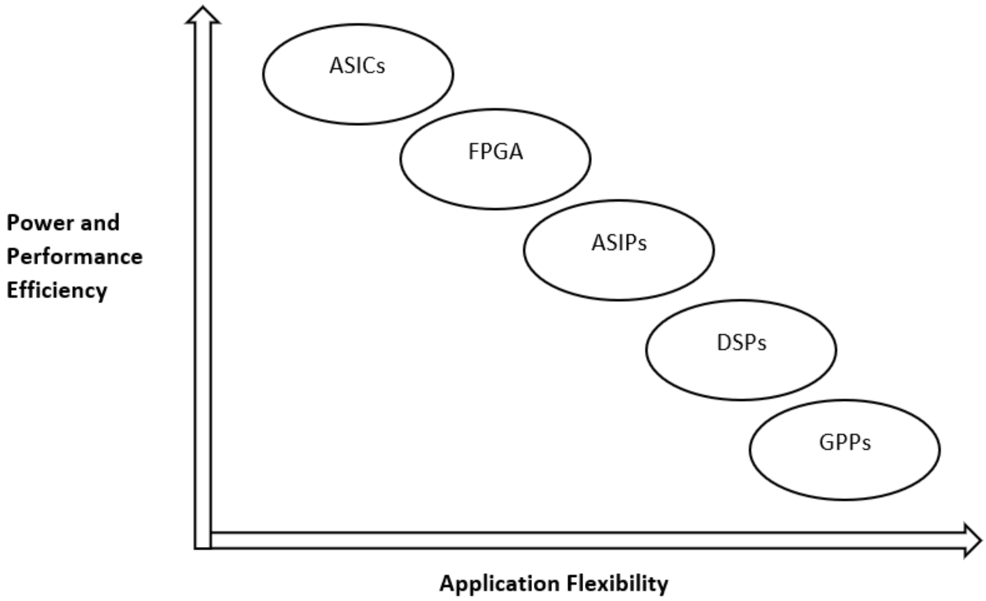


Figure 1: Power, Performance Efficiency and Application Flexibility of several hardware implementations. Retrieved and adapted from Synopsys [3]

General-purpose processors (GPPs), as their names state, are designed to offer very high programming flexibility. However, because general-purpose processors are aimed to run a wide range of applications, when running some specific applications, often result in poor energy and performance efficiency. Some examples of general-purpose processors are x86 and ARM-based processors.

On the other side of Figure 1 are Application-Specific Integrated Circuits (ASICs), which are designed for a dedicated task rather than general-purpose use. As a result, they are superficial in terms of performance and silicon cost but non-programable. Another disadvantage of ASICs is the difficulty in testing and debugging, resulting longer time-to-market path, and thus higher development costs. Some examples of ASICs are cryptocurrency mining machine processors or satellite chips.

According to [4] ASIPs are a balance between general-purpose processors and ASICs. In terms of achieving energy, power, performance efficiency, and application flexibility, ASIPs deploy customized data path and specialized hardware to allow more parallelism [3] Compared to GPPs, ASIPs provide much greater performance and power by utilizing special hardware architecture and a set of instructions that allow the acceleration of some domain-specific operations. Some parallel mechanisms, including instruction, data , or task-level parallelism, can be deployed to the architecture to improve performance while reducing processor power consumption. Unlike ASICs, ASIPs allow sufficient flexibility via software programmability. With 5G network standards getting more mature, there are likely upcoming changes in the existing standards and SoC requirements. In many cases, users can re-program the processors without the expensive cost of any SoC re-spin. The ability to re-program the processor is critical in reducing the costs, development time, and time-to-market path.

The primary purpose of deploying an ASIP is not to surpass ASICs or DSPs in terms of throughput but to add a certain level of flexibility while maintaining satisfactory performance when running specific applications. According to [5] the aim of an ASIP design is to achieve the maximum performance over silicon, the highest performance over power consumption, and the highest performance over design cost, based on the carefully specified function coverage. Instead of maximizing flexibility, a sufficient level of flexibility for the specified class of application is more important. Similar to flexibility, the performance is not aimed to be the best overall but application-specific.

Another benefit of ASIP in terms of development time and implementation lies in the variety of existing open-source or commercial toolsets, including PEAS, Tensilica, Synopsys ASIP Designer, RISC-V, Rocket, or TCE. As the development of an ASIP, its components, and extensions depend on these tools, the need for creating an entire ASIP from scratch no longer exists. That is a significant advantage during the development in terms of timeline.

Additionally, besides designing activities, different additional features from the tools have been incorporated, such as supporting, debugging, and simulating the core on both the simulator and RTL-based environment. The availability of different features helps the testing and verification activities on ASIPs, leading to a shorter development time and reducing failure risks, thus significantly bringing down the development cost.

Over the past two years, chip makers have deployed some ASIPs for their use as hardware accelerators and algorithm processors in 5G networks. [6] deployed an ASIP for accelerator task allocation, load-store operation, and scheduling for 5G micro base stations. [7] also introduced an ASIP based on a simple RISC-V core with custom instruction extensions for maximizing the throughput of a set of radio resource management algorithms. Another programmable 16-lane Single Instruction Multiple Data (SIMD) ASIP for baseband processing in massive Multiple-Input Multiple-Output (MIMO) [8] was presented, showing its high performance when doing vector processing operations and complex accumulator for various massive MIMO algorithm.

2.2 RISC-V Instruction Set Architecture

Instruction Set Architecture is a set of instructions that can be implemented and executed on the processor architecture. It can be seen as an interface between hardware and software that allows programmers or compiler writers to write programs or compilers for the processor. When a processor runs a program, it sees the program as a sequence of instructions supported by its microarchitecture and guidance on what to do. ISAs can be licensed, custom, or open-sourced. Many ASIPs use RISC-V open-source base ISA with the option of adding vendor-specific custom extensions to allow more flexibility.

RISC-V is a family of open-source ISAs developed by the University of California, Berkeley. RISC-V has several base integer variants, including 32, 64, and 128-bit variants. Different ISAs are classified based on integer register widths, address space size, and a number of registers. In addition to the base integer ISA, some standard extensions have been developed to support different operations in general software development:

- I: contains arithmetic operations, load, store, and control instructions for 32-bit integer numbers.
- M: contains standard multiplication and division instructions for integer numbers.
- A: contains atomic operations.
- F: contains single-precision floating-point operations, including single-precision load and store instructions.
- D: contains double-precision floating-point operations, including double-precision load and store instructions.
- C: contains compressed instruction set extension including arithmetic operations, load, store, and control instructions for 16-bit instruction encoding.

From the mentioned variants, developers can optionally add their own extensions containing custom instructions and custom registers per needs. The flexibility in adding custom instructions allows developers to create an ASIP that is customized for a specific application, hence maximizing performance and energy efficiency. As the RISC-V communication is growing, ISA specification keeps evolving and updating. Therefore, during the development of this work, the base integer 32-bit ISA described from "RISC-V Unprivileged

ISA V20191213" released on December 12, 2019, is used as a reference source for the RISC-V ISA literature explanation.

The ISA defines some specifications and requirements about register file structure, memory size, and function units required to perform arithmetic operations such as arithmetic logic unit (ALU) or load store unit (LDST). Details about these are explained in the following subchapters.

2.2.1 Register File

Register File is a component in processor architecture for storing general-purpose registers. Depending on the ISA type and processor architecture decision, register file can have different structures in different processors. The 32-bit base integer ISA, RV32I [RISC-V Instruction Set Manual, 2017] specifies the required register file structure and size for RISC-V Core. RV32I contains 32 fields, with each field being a 32-bit integer number (See appendix A).

For some applications requiring fewer register file accesses and fields, an optional 16-bit compressed extension can be added with only eight registers to save the instruction memory and encoding space. In that situation, eight registers can be mapped or aliased to registers x8-x15 of the RV32I register file. In addition to the mentioned 32 registers, RV32I also specifies an extra program counter (PC) register needed to contain the current instruction address of the executed program. PC is a 32-bit integer number, and its value should be positive only.

2.2.2 RV32I Instruction Set

RV32I ISA has six instruction formats including: R-Type, I-Type, S-type, B-type, U-type, and J-type. These formats shown in the Figure 2 below.

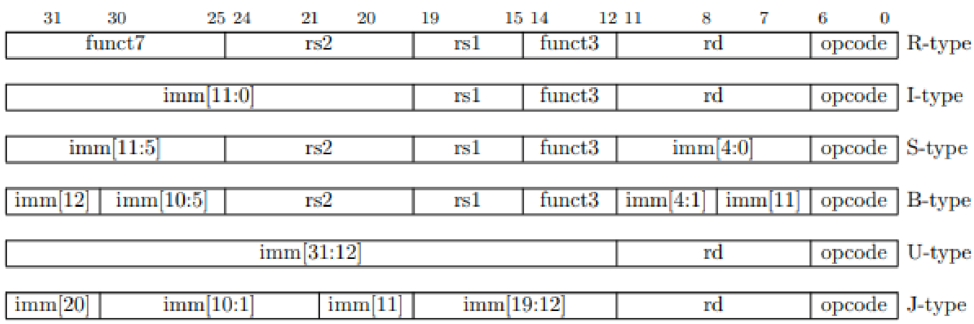


Figure 2: RV32I instruction formats. Retrieved from RISC-V Instruction Set Manual [9]

R-Type is the format for integer register-register arithmetic and logical operations. These instructions take two source registers as operands and store the result into destination register *rd* in the register file. Instruction type and operation are selected via 7-bit field funct7 and 3-bit field funct3. R-type format instructions including:

- ADD
- SUB
- XOR
- OR
- AND
- Shift Left Logical (SLL)
- Shift Right Logical (SRL)
- Shift Right Arithmetic (SRA)
- Set Less Than (SLT)
- Set Less Than Unsigned (SLTU).

I-type is the format for integer register-immediate arithmetic, logical, and memory load operations. These instructions take one source register and one immediate as operands and store the result into destination register *rd* in the register file. The immediate field is a 12-bit signed value and operation type is selected via 3-bit field funct3. I-type format instructions including:

- ADD Immediate (ADDI)
- SUB Immediate (SUBI)
- XOR Immediate (XORI)
- OR Immediate (ORI)
- AND Immediate (ANDI)
- Shift Left Logical Immediate (SLLI)
- Shift Right Logical Immediate (SRLI)
- Shift Right Arithmetic Immediate (SRAI)
- Set Less Than Immediate (SLTI)
- Set Less Than Unsigned Immediate (SLTUI).
- Load Byte (LB)
- Load Half Word (LH)
- Load Word (LW)
- Load Byte Unsigned (LBU)
- Load Half Word Unsigned (LHU)
- Jump and Link Register (JALR)
- Environment Call (ECALL)
- Environment Break (EBREAK)

S-type is the format for integer memory store operations. These instructions take two source registers: *rs1* for the base memory address, *rs2* for the data to be stored ,and one immediate offset *i* value as operands. The result is stored into memory address *rs1 + i* with the content of *rs2*. The operation type is selected via 3-bit field funct3. S-type format instructions including:

- Store Byte (SB)
- Store Half Word (SH)
- Store Word (SW)

B-type is the format for branch operations. These instructions take two source registers for comparison and one immediate as an operand. The immediate is a 13-bit value in two's complement representation indicating a number of 32-bit instructions move to when branch is taken. When branch is taken, the program counter (PC) is advanced to PC + immediate*4. When branch is not taken, program counter is advanced normally (PC+4). The operation type is selected via 3-bit field funct3. B-type format instructions including:

- Branch If Equal (BEQ)
- Branch If Not Equal (BNE)
- Branch If Less Than (BLT)
- Branch If Greater Than Or Equal (BGE)
- Branch If Less Than Unsigned (BLTU)
- Branch If Greater Than Or Equal Unsigned (BGEU)

U-type is the format for large immediate operations. These instructions take a 20-bit unsigned immediate to load upper 20 bits of destination register *rd* with this value. U-type format instructions including:

- Load Upper Immediate (LUI)
- Add Upper Immediate To PC (AUIPC)

J-type is the format for dealing with jump instruction (JAL). This instruction takes a 20-bit immediate value for the jump offset (PC + offset) while saving the next instruction address (PC+4) into destination register *rd*.

The simplicity of the RISC-V core in terms of the same instruction length allows easier fetching of all the instructions. In addition, similar location for source and destination fields in all the instructions significantly reduces the decoder complexity. As a result, all the instructions can be fetched in one cycle and decoded in the following cycle. Finally, memory access is only allowed for load and store instructions. This restriction means one stage can be used to compute the memory address while the later stage can use the computed address to access the memory. Those benefits can lead to a simple, short pipeline design, which will be discussed in a later chapter.

2.3 Basic RISC-V Core Implementation

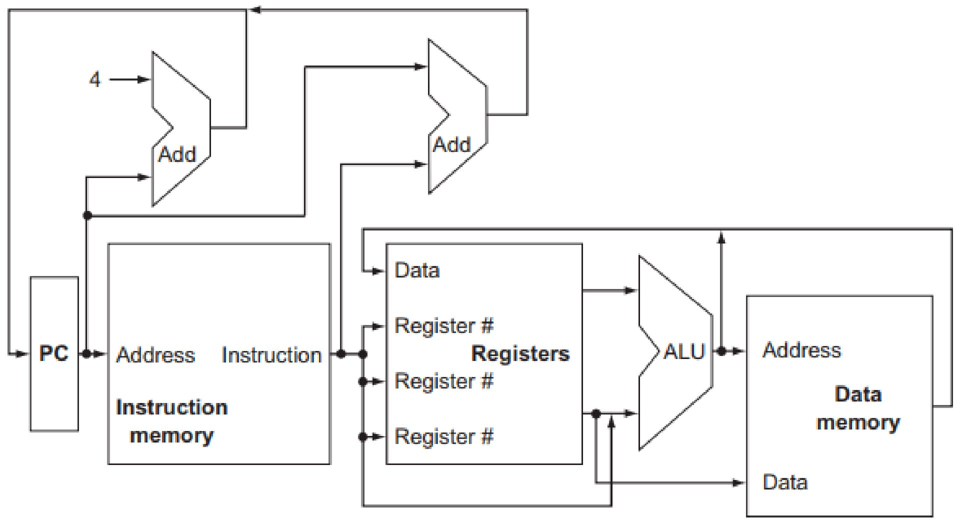


Figure 3: Basic implementation of RISC-V core. Retrieved from [10]

Figure 3 above represents a basic RISC-V core implementation with some basic units:

- Program Counter (PC): holds the current instruction address to be executed.
- Instruction Memory: contains sequence of 32-bit instructions to be fetched by the core with given address.
- Adders: to advance PC (PC+4) to next instruction or compute jump targe address (PC + immediate).
- Register File: contains 32 32-bit registers for different instructions needs, has two read ports and one write port.
- Arithmetic Logical Unit (ALU): performs arithmetic operations.
- Data memory: keeps data for load and store operations.

The datapath can be visualized from the left side of the figure. PC is supplied to Instruction Memory to fetch instruction. The instruction is then decoded by the decoder, containing different fields, including:

- Instruction Opcode: to identify the instruction type (R, S, I, B, U, or J).
- Source registers: registers to be read from Register File.
- Immediate: Immediate value for S, I, B, U, and J instructions
- Destination register: register to be written to Register File

ALU always takes the first input from Register File and produces one output. However, depending on the opcode field, the way how the core accesses and uses different blocks can vary:

- For R-type format, the second ALU input is taken from Register File. ALU performs an arithmetic operation, and its output is written back to a destination register (rd) in Register File.
- For I-type format except for load instructions, the second ALU input is taken from instruction immediate field. ALU performs an arithmetic operation, and its output is written back to a destination register (rd) in Register File.
- For load and store instructions, ALU second input is taken from instruction immediate field. ALU performs arithmetic add operation to compute an address. The computed address is then used to read the data from Data Memory in case of a load or to write data to Data Memory in case of a store. The loaded result is then written back to a destination register (rd) in Register File.
- For branch instructions, an adder is used to add the source register (rs) with an immediate offset to compute a jump target address. Whether the branch condition is met (branch taken) or not (branch not taken), PC will be either advanced to the next instruction or the jump target address.

While the above implementation is easy to follow and understand, there is a fundamental issue: for every instruction, all executions must be completed within one clock cycle. This limitation causes an increase in the core clock cycle period as it has to accommodate time for all the instructions to be executed in a single cycle, which leads to an inefficient and low throughput system and is not suitable for modern processor design. This fact leads to the introduction of a better and more practical implementation using pipelining.

2.4 Basic RISC-V Implementation with Pipeline

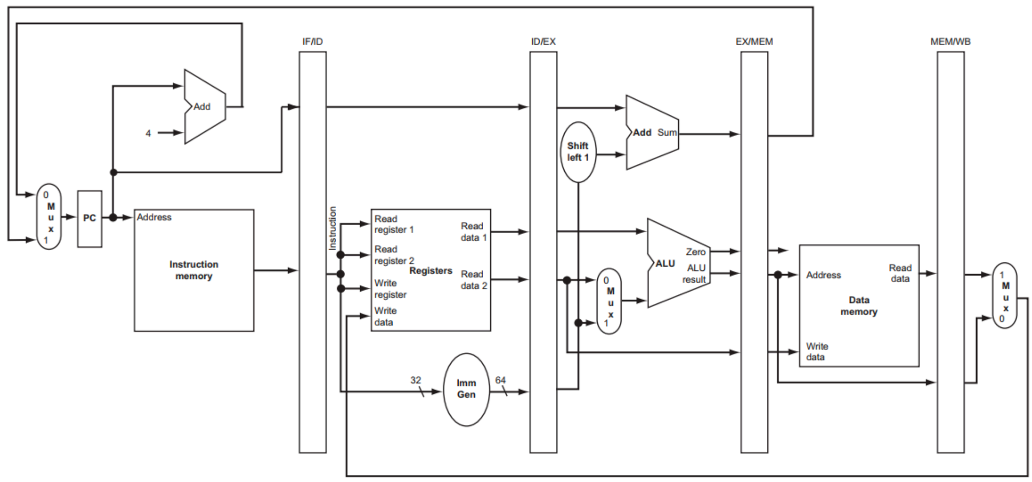


Figure 4: Basic 5-stage pipeline implementation of a RISC-V core. Retrieved from [10]

Figure 4 above shows a basic high-level implementation and datapath of a basic RISC-V core with pipelined implementation. Compared to the implementation explained in Figure 4, this implementation has two more units. They are an Immediate Generate block to sign-extend 12-bit immediate to a 64-bit value and four pipeline registers used for storing data passing through different pipeline stages, including IF/ID, ID/EX, EX/MEM, MEM/WB. Note that the core in the figure above supports the execution of 64-bit instructions, while a typical basic ASIP core supports 32-bit instructions.

The pipeline above has five stages that are connected with each stage is responsible for performing a specific task:

1. Instruction Fetch (IF): PC is given to the Instruction Memory port for which a 32-bit instruction is fetched out and stored into pipeline register IF/ID to be decoded later. In addition to instruction fetch, the logic for updating PC is also implemented at this stage. The controlling logic allows the update of PC with different possible values:
    - During normal execution, PC is incremented by 4 to advance to the next instruction in the sequence.
    - If a jump signal is taken, PC is updated with the jump target address. A jump signal can be asserted as a result of a normal jump instruction or PC is at loop end instruction which needs a jump to the loop start instruction.

- If any of the hazards occur for which the processor inserts any stall cycle, PC value is not updated as the exiting value remains unchanged. More information about different hazard types will be discussed later in chapter 2.5.

2. Instruction Decode (ID): After fetched instruction is loaded into pipeline register IF/ID, it will be decoded at this stage. Depending on the instruction opcode, different possible fields can be extracted, including source register numbers (rs) and immediate value. The source register numbers are then forwarded to read out the corresponding register fields from Register File and store that information into pipeline register ID/EX. The calculated PC at IF stage, which is stored in pipeline register IF/ID as well as sign-extended immediate, are also passed into pipeline register ID/EX. Note that in order to read two register fields at the same stage, Register File needs to have two read ports.

3. Execute (EX): This stage reads the content of pipeline register ID/IF to obtain operands for ALU operation or branch condition check. For arithmetic operations, the first operand is always taken from the register file. The second ALU operand is taken from the bottom mux, which is controlled by the control unit. The multiplexor decides whether the second operand value should come from the register file or from the sign-extended immediate. In the case of a branch instruction, the jump target address is computed by adding the PC with a sign-extended immediate value. In the case of load/store instruction, the address for accessing data memory is computed using ALU to add rs1 with immediate value. All the computed outputs at this stage are once again passed into pipeline register EX/MEM.

4. Memory Access (MEM): This stage is needed for data memory access operation using load or store instructions with the address taken from pipeline register EX/MEM. In the case of arithmetic operation, this stage is bypassed with the ALU result that is passed from EX/MEM to MEM/WB register. In case of a load instruction, load result is read from memory to store into MEM/WB register. In case of store instruction, data memory is written with a value stored in MEM/WB register.

5. Memory Write-back (WB): This stage takes either a bypassed ALU computed result or a memory load result from MEM/WB register and writes that value into the destination register (rd) of the Register File. If rd is x0, no write is made.

	Time								
	t0	t1	t2	t3	t4	t5	t6	t7	t8
Instruction 1	IF	ID	EX	MEM	WB	-	-	-	-
Instruction 2		IF	ID	EX	MEM	WB	-	-	-
Instruction 3			IF	ID	EX	MEM	WB	-	-
Instruction 4				IF	ID	EX	MEM	WB	-
Instruction 5					IF	ID	EX	MEM	WB
…									

Figure 5: Instructions are overlapped in a 5-stage pipeline

Considering instructions in Figure 5 above, each instruction enters the pipeline at different clock cycles. They are all passing through different pipeline stages. Note that no more than one instruction can be at the same pipeline stage simultaneously.

2.5 Pipeline Hazards

While the pipeline implementation reduces the core cycle time and increases instruction throughput as more instructions can be issued and executed concurrently, some hazards always exist. A hazard prevents the processor from executing a new instruction in the next cycle. Pipeline hazards include three types, which all need to be resolved.

2.5.1 Data Hazards

Data hazards occur when different instructions in different pipeline stages access the same register. When that situation happens, the core has to stall to allow the operation that has earlier access to that register to be completed first. There are three situations when data hazards could occur:

- Read After Write (RAW): occurs when the later instruction tries to read a register before it is written by earlier instruction.

	Time					
	t0	t1	t2	t3	t4	t5
add x3, x2, x4	IF	ID	EX	MEM	WB (Write x3)	-
sub x5, x3, x4		IF	ID (Read x3)	EX	MEM	WB

Figure 6: Example of RAW hazard

Figure 6 demonstrates an example of the RAW hazard between 2 instructions: ADD and SUB. The ADD instruction reads two registers x2, x4 at the ID stage then performs x3 = x2 + x4 at the EX stage and writes x3 at the WB stage. The SUB instruction reads two registers, x3 and x2, at the ID stage. RAW hazard occurs as the SUB instruction tries to read register x3 at the ID stage (at t2) when x3 content has not yet been written by the ADD instruction (at t5). As a result, the processor has to stall the SUB instruction for three cycles until the ADD instruction is finished (when writing to x3 is done).

- Write After Read (WAR): occurs when later instruction tries to write to a register before it is read by earlier instruction.

```
add x2, x1, x3
or  x1, x4, x5
```

In this case, if there exists any situation when the OR instruction writes to x1 before it is read by the ADD instruction, WAR hazard occurs.

- Write After Write (WAW): occurs when later instruction tries to write a register before it is written by earlier instruction.

```
sub x1, x2, x3
and x1, x4, x5
```

In this case, if there exists any situation when the AND instruction writes to x1 before it is written by the SUB instructions, WAW hazard occurs.

The data hazard is caused by the unavailability of needed data at subsequent instruction. Therefore, the data hazards can be practically resolved by bypassing or forwarding. Forwarding means having additional hardware to create a bypass network in which data needed by the subsequent instructions can be available at the ID stage without having to wait for the data to be written into memory or register file at the later instruction. This concept is represented in the example in Figure 7 below.

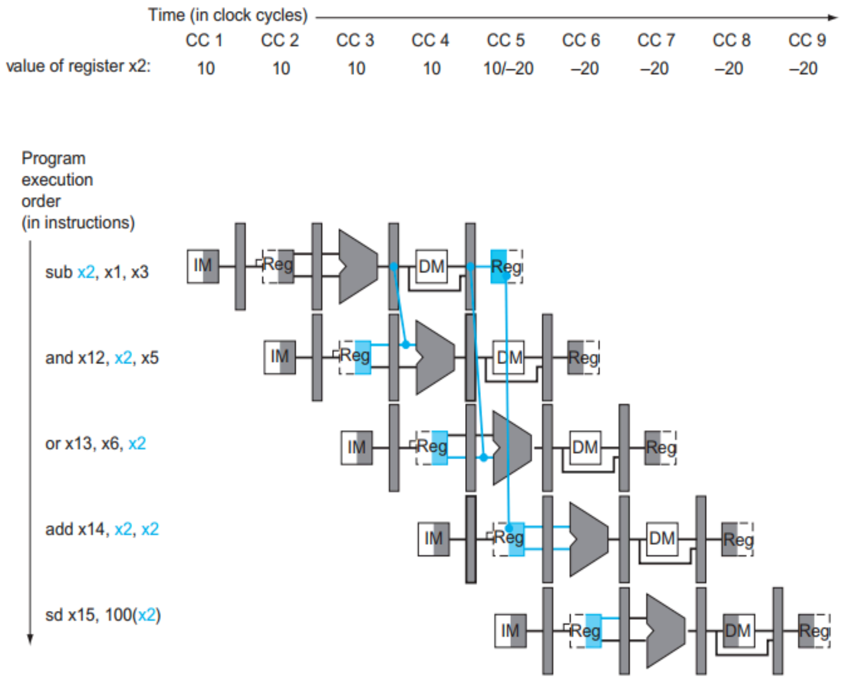


Figure 7: Data needed by subsequent instruction can be forwarded from prior one. Retrieved from [10]

Figure 7 shows the data dependency of accessing register x2 between the first four instructions. The first instruction (SUB) computes a subtraction of two values, x1 and x3, and writes the result into x2 at the WB stage. The following three instructions (AND, OR, ADD) require the read of x2 at the ID stage for decoding. Usually, without forwarding logic, the SUB instruction will write x2 at clock cycle 5. All the instructions that want to read x2 must wait until clock cycle 5 when the x2 data is available in the register file. Otherwise, the wrong value of x2 will be read. With forwarding or bypassing, as soon as the data for x2 is computed, in this case after ALU at clock cycle 3, it can be forwarded to the later instructions without having them to wait till clock cycle 5:

- the AND instruction gets x2 value by having a path from EX/MEM pipeline register to its ALU first input.
- the OR instruction gets x2 value by having a path from MEM/WB pipeline register to its ALU second input
- The later ADD and SD instructions do not need any path because x2 value is already available in the register file.

The needed pipelined implementation context with forwarding showed in Figure 8. Forwarding logic is implemented by adding a path from each of the pipeline registers back to the ALU inputs.

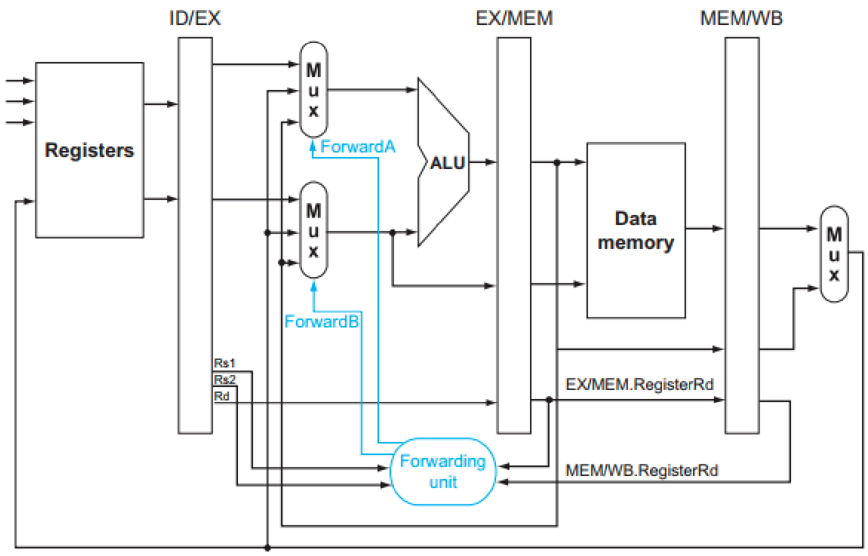


Figure 8: Basic RISC-V core with forwarding. Retrieved from [10]

For each of the ALU inputs, the data source can either come from the register file (ID/EX), prior ALU result (EX/MEM), or from prior ALU result or data memory (MEM/WB). Note that in this implementation, the multiplexors have to be expanded to accommodate new data sources from the new forwarding unit.

2.5.2 Structural Hazards

Structure hazards occur when there is hardware resource contention between different instructions in the pipeline at the same clock cycle. Typically, structure hazards can be seen at multi-cycle functional units when the use of those units is spread into several clock cycles. Some examples of such units are multipliers or dividers. In addition, hardware resources used and shared by different pipeline stages are also the sources of structure hazards. When the functional unit or hardware resource is occupied by one instruction, if any other instructions intend to use the unit, they have to be stalled until the unit is free.

Consider the following example in Figure 9 of a processor with a single memory unit being used to fetch instruction at the IF stage and to be accessed at the MEM stage for Load and Store operations.

	Time								
	t0	t1	t2	t3	t4	t5	t6	t7	t8
Load/Store	IF	ID	EX	MEM	WB	-	-	-	-
Instruction 2		IF	ID	EX	MEM	WB	-	-	-
Instruction 3			IF	ID	EX	MEM	WB	-	-
Instruction 4				stall	IF	ID	EX	MEM	WB

Figure 9: Structure hazard caused by same hardware resources shared between different pipeline stages

At time t3, the memory unit is occupied by the Load/Store instruction at the MEM stage. At the same time, instruction 4 enters the pipeline at the IF stage and needs to fetch instruction from the same memory. In this case, a structural hazard occurs as two instructions are racing for the memory interface occupation. The prior instruction has higher priority than the subsequent one, so the processor needs to stall instruction 4 for one clock cycle. Another example is shown in Figure 10 below.

	Time								
	t0	t1	t2	t3	t4	t5	t6	t7	t8
mul x1, x2, x3	IF	ID	EX	MEM	WB	-	-	-	-
mul x4, x5, x6		IF	ID	stall	EX	MEM	WB	-	-
Instruction 3			IF	stall	ID	EX	MEM	WB	-
Instruction 4				stall	IF	ID	EX	MEM	WB

Figure 10: Structure hazard caused by multi-cycle unit

The processor has a multiplier functional unit that takes two cycles to compute the multiplication operation: EX and MEM stages. The first multiplication instruction occupies the function unit at time t2-t3. When the second multiplication instruction at EX stage at time t3 tries to use the multiplier, the processor inserts a stall cycle because the unit is occupied by the first instruction. The structure hazards can be avoided by:

- Making the functional unit fully pipelined
- Having additional hardware units

The penalty for making the functional unit fully pipelined is increased delay and possible gate counts. On the other hand, having an additional hardware unit could increase the processor area significantly if the hardware unit itself is already area-consuming. The tradeoffs between the cost of adding hardware versus the increase in system throughput (by avoiding stall cycles) should be examined carefully based on application specifics. Sometimes, it could be worthful to have structural hazards in the system if the hazards occur infrequently.

2.5.3 Control Hazards

Control hazards arise when there appear delays in determining the subsequent instructions after a branch instruction. Recapping the branch instructions (B-type instructions, JAL, and JALR) described in chapter 2.2.2, they are instructions in which the branch taken condition is checked to determine whether the subsequent instruction address is the jump target (PC + offset, branch taken) or the one already in the pipeline (PC+4, branch not taken). A new instruction is already in the pipeline because in every clock cycle, in a regular flow, instruction memory always fetches out a new instruction for decoding. In case of a branch is not taken, as the next instruction is already fetched and enters the pipeline after the branch instruction, there is no delay, thus no control hazard. In the case of a taken branch, as the branch taken result is only known until the ID or EX stage, there is one or two stalled cycles delay until the jump target instruction is determined and fetched. The fact that the processor has to insert one or two stall cycles due to the branch instruction is called a control hazard. The branch is always taken in the case of JAL or JALR instruction. The example in Figure 11 shows how the control hazard exists in the pipeline due to a branch instruction and how the processor resolves it.

		Time								
		t0	t1	t2	t3	t4	t5	t6	t7	t8
PC	jal x3, 20	IF	ID	EX	MEM	WB	-	-	-	-
PC + 4	add x2, x4, x5		IF	stall	stall	-	-	-	-	-
PC + 8	mul x7, x7, x4			stall	stall	-	-	-	-	-
PC + 20 (Target)	sub x8, x9, x8				IF	ID	EX	MEM	WB	-

Figure 11: Control hazard when branch is taken

For the first instruction (PC), the jump signal is asserted at the ID stage at time t1, and the jump target address is computed and known at the EX stage at time t2. However, to sustain the pipeline, a new instruction (add) is already fetched at cycle t2. Therefore, at time t2, the processor has to:

- Signal the controller to kill any fetched instruction (PC+4) at previous cycle
- Signal the decoder not to decode any instruction for the next two cycles t2 and t3
- Signal the fetch unit not to fetch any new instruction (PC+8)

Finally, the target instruction (sub) entered the pipeline at time t3 and decoded at time t4.
In case the branch is not taken (Figure 12), the processor needs to stall for only one cycle, then continue to decode the fetched subsequent instruction in the pipeline.

		Time t0	t1	t2	t3	t4	t5	t6	t7	t8
PC	beq x3, x7, 20	IF	ID	EX	MEM	WB	-	-	-	-
PC + 4	add x2, x4, x5		IF	stall	ID	EX	MEM	WB	-	-
PC + 8	lui x4, 2000			stall	IF	ID	EX	MEM	WB	-
PC + 12	sub x8, x8, 1					IF	ID	EX	MEM	WB

Figure 12: Control hazard when branch is not taken

At time t2, the branch taken condition is checked. Therefore, at time t2, the processor has to:

- Signal the decoder not to decode any instruction at this cycle
- Signal the fetch unit not to fetch any new instruction at this cycle

The add instruction (PC + 4) is ready to enter the decode stage at time t3.

3 Investigation of Feasible Performance Improvement Method

Before conducting this research, different options have been considered for improving the core throughput. However, before addressing different architectures, it is advisable to understand the concept of processor waste and functional unit utilization in the processor. Each architecture is designed to tackle either or both mentioned issues.

3.1 Functional Unit Utilization and Issue Bandwidth Waste

An ASIP architecture typically contains several functional units, including an arithmetic logic unit (ALU), an address generation unit (AGU), a load-store unit (LDST), a multiply unit (MPY), and other units. For each unit in a pipelined processor, functional unit utilization is an important metric indicating how frequently the instructions are executed on the unit: the higher utilization, the better.

Figure 13: Functional unit utilization

Each filled box in Figure 13 above indicates that the functional unit is executing an instruction, while an empty box indicates that the unit is free or not being utilized. An unutilized unit is typically caused by data dependency between instructions (pipeline data hazards). For example, an ALU in some cycles might not be used because the unit is waiting for one of its operand inputs, resulting from an earlier instruction. In this case, the ALU is not utilized.

An ASIP could have multiple issue slots to allow the issue of multiple instructions in parallel. This technique increases processor performance by exploiting instruction level parallelism (ILP). However, a single thread ASIP with multiple issue slots can still encounter some waste in processor issue bandwidth due to data dependency and long latency as stalling penalty due to the hazards. This kind of waste is classified into two types: vertical and horizontal waste.

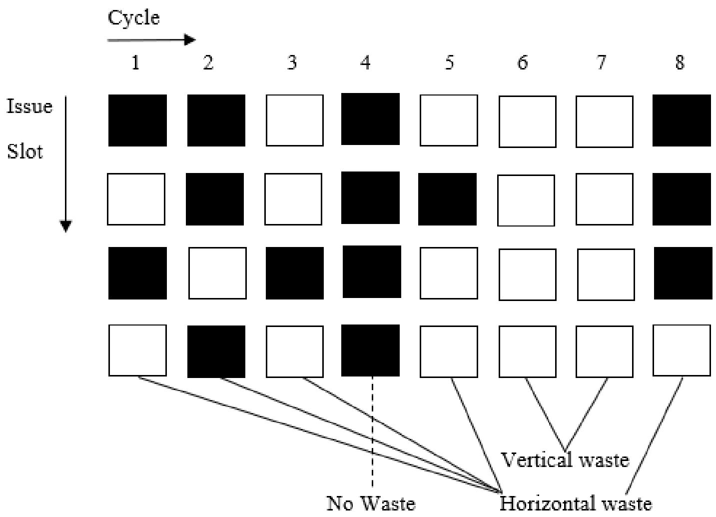


Figure 14: Vertical and horizontal waste of an ASIP with 4 issue slots

Vertical waste occurs during a cycle when no instruction from any slot is issued. This problem results in a big waste in the issue bandwidth and is normally caused by a long latency operation in the system like memory access or division instruction. For example, from Figure 14 above, vertical waste can be seen in cycle six and seven when none of the issue slots is able to issue any instruction.

Horizontal waste, on the other hand, is the waste in the issue bandwidth when only some of the issue slots are able to issue instructions in the same cycle. The causes of horizontal waste are the dependencies between instructions that prevent the processor from consuming all the issue slots in a given time. In Figure 14, horizontal wastes can be seen in most of the cycles, including cycles one, two, three, five, and eight. Cycle four is the ideal case when there is no waste in the issue bandwidth. An ASIP with a single pipeline or single-issue slot should never have horizontal waste because there can be only one issued instruction at a time, while vertical waste could exist because in a single-issue core, vertical waste occurs as a result of any stall.

3.2 Super Pipelining Processor

As discussed in chapter 2.4, despite the fact that pipelining is valuable in improving processor clock cycle time and instruction throughput, having a pipeline in a processor also has some limitations.

The processor speed is determined and limited by the slowest stage of the entire pipeline. As a result, some processors decompose the long latency stage into several smaller ones, and this technique is called super pipelining. A deep pipeline allows multiple instructions to run simultaneously at a very high clock rate. Some commercial examples of super pipelining are the 10-stage pipeline Intel Pentium 3 shown in Figure 15 and the 20-stage pipeline Intel Pentium 4 processors [11] shown in Figure 16. However, the downside of having a very deep pipeline is the difficulty in avoiding stalls, for example, stalls caused by control hazards, cache misses, or stall penalty consequences. When the hazard occurs, the pipeline needs to be flushed, and the deeper the pipeline, the worse this penalty gets. Typically, in such cases, a very accurate branch prediction logic needs to be implemented inside the processor, adding more complexity to the design. Furthermore, high power consumption is typically seen as a result of a high clock rate.

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Figure 15: Intel Pentium 3 with 10 pipeline stages

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Allocate	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	EX	Flags	Br Ck	Drive

Figure 16: Intel Pentium 4 with 20 pipeline stages

3.3 Multi-issue Processor

An improved pipeline implementation for improving instruction throughput by exploiting ILP is to have multiple pipelines or issue slots. This type of architecture allows the issue of multiple instructions in the same clock cycle. Practically, in terms of implementation, multi-issue processors can be classified into two types: superscalar and very long instruction word (VLIW).

3.3.1 Superscalar

Superscalar can schedule multiple instructions statically and dynamically, while VLIW can only statically issue multiple instructions. Superscalar can be understood simply as having multiple pipelines overlapped that allow the issue, decode, execute, memory access, and writeback of multiple instructions simultaneously, as shown in the example in Figure 17 below.

	Clock cycle					
	0	1	2	3	4	5
PC	IF	ID	EX	MEM	WB	-
PC + 4	IF	ID	EX	MEM	WB	-
PC + 8		IF	ID	EX	MEM	WB
PC + 12		IF	ID	EX	MEM	WB

Figure 17: Simple 2-way superscalar pipeline

Similar to the basic pipeline implementation, single thread superscalar processor also encounters some hazards: data, structural, and control. During the instruction scheduling, the scheduler examines these hazards and then decides which instructions should be issued in parallel. There are two execution modes in superscalar architecture: in-order and out-of-order. In in-order issue implementation, the instructions are scheduled statically, and the compiler generates the order of instruction execution in the pipeline. During the instruction execution, if there exists any stall sourced by any instruction, the entire pipeline stalls. For example, suppose there exists a situation when the second instruction in the pipeline is not issued due to the data dependency with the first instruction. In that case, the entire pipeline is stalled, and no new instruction is issued until the data dependency is resolved. Out-of-order execution, on the other hand, allows dynamic scheduling. In the same case with in-order execution, the compiler looks at the third instruction instead of stalling the entire pipeline. In case of no data dependency of the third instruction with the first one, the compiler can schedule both the first and the third instruction to issue them simultaneously. Despite the instruction throughput gain in superscalar processors, this architecture has some limitations when implemented in ASIP:

- The limitation of performance in in-order architecture due to static scheduling.
- The dynamic scheduling mechanism of out-of-order architecture provides better performance than in-order architecture. However, dispatching multiple instructions into multiple execution units relies heavily on the scheduler, which is typically hardware expensive and complex to design.

3.3.2 VLIW

Another multi-issue architecture called very long instruction word (VLIW), while avoiding the complex scheduler and hardware costs from additional pipelines, can still exploit ILP. VLIW is a computer architecture technique that increases system performance by exploiting instruction level parallelism (ILP) via deploying multiple issue slots. As its name states, VLIW supports instructions that can be seen as a large instruction containing encoding for several smaller instructions that can be run simultaneously. With VLIW, there is great potential in increasing instruction throughput by having multiple instructions executed in parallel. Unlike out-of-order superscalar processors that rely heavily on the hardware scheduler, in VLIW, the compiler performs the instruction scheduling. During the runtime, the compiler determines the instructions dependencies and packs the sub-instructions that can be executed concurrently into a wider one. The wide instruction can contain one or more small instructions depending on the number of issued slots in the microarchitecture and how tight the compiler can schedule the instructions. The width of the wider instruction is the sum of the widths of all sub-instructions it contains. For example, a 3-way VLIW ASIP core shown in Figure 18 contains three issued slots, each capable of executing a 32-bit instruction. When scheduling, the compiler can pack all the instructions, resulting in a 96-bit wide instruction.

Clock Cycle

0	1	2	3	4	5	6
IF	ID	EX	MEM	WB	-	-
	ID	EX	MEM	WB	-	-
	ID	EX	MEM	WB	-	-
	IF	ID	EX	MEM	WB	-
		ID	EX	MEM	WB	-
		ID	EX	MEM	WB	-
		IF	ID	EX	MEM	WB
			ID	EX	MEM	WB
			ID	EX	MEM	WB

Figure 18: a 3-way VLIW pipeline. Note that unlike superscalar pipeline when each issue slot has its own instruction visible at IF stage. In VLIW case, the fetched instruction appears to be a wide one that contains one or more sub-instruction for the issue slots.

For the parallel execution of three instructions, each instruction is executed in a separate slot, and they all access the register file in the same cycle. Because all the slots share the same register file, each slot needs a separate register file read and write ports.

VLIW processors encounter several drawbacks. First, in a practical ASIP implementation with VLIW, each slot can have more read and write ports to allow the creation of more complex instructions that need access to register file. However, the downside of having more read/write ports is the increase in the register file size, thus impacting the total core area and size. Therefore, the tradeoff between more parallelism versus core area needs to be evaluated carefully.

Another bottleneck when designing a VLIW core is the compiler complexity and intelligence. As the compiler is responsible for parallel scheduling instructions that can be packed together, the question of how smart the compiler is or how tight the compiler can schedule instructions to fill all the VLIW slots as up as possible is to be discussed and considered. So, the tighter the scheduling, the higher the issue slot utilization. When the slots are not utilized well, the compiler needs to add no-ops (NOPs) instructions to fill up these slots. As a result, code density might suffer because of the mentioned reason. Finally, the functional unit utilization can worsen due to the introduction of new functional units and issue slots. The more issue slots and functional units the core has, the more likely these slots and units are not utilized well even with a good compiler. Additionally, similar to superscalar, single-thread VLIW still encounters some stalls related to pipeline hazards, thus, hardware resource waste.

3.4 Investigation Summary and Choice of Architecture

Different architecture models have been studied, and they are summarized in Table 1 below.

Table 1: Investigation summary of different architecture models on ASIP and their implementations on ASIP perspective.

	Pros	Cons	ASIP perspective
Super Pipelining	• Allow very high clock rate	• Difficult in avoiding stall • Stall penalty is significant • Typically requires branch predictor, which is expensive • High power consumption	• Stall penalty is significant • High power consumption is unwanted • Not targeting for maximum clock rate
Superscalar	• Allow high instruction throughput • Allow high issue slot and functional unit utilization	• Hardware scheduler is complex and expensive • Pipeline hazards still exist	• Hardware scheduler is complex and expensive for ASIP
VLIW	• Allow high instruction throughput • Instructions scheduling is done by compiler, less expensive than hardware scheduler	• Issue slot and functional unit utilization could be low • Code density might suffer as compiler need to add no-operations (NOPs) when one or more slots cannot be utilized. • Good compiler for tight instructions scheduling could be complex. • Pipeline hazards still exist	• Issue slot and functional unit utilization could be improved by deploying another architecture on the top of an VLIW ASIP • Good compiler is available from some ASIP designing tool manufacturers, thus less overhead. • Pipeline hazards could be resolved by deploying another architecture on the top of an VLIW ASIP

Despite the benefit of achieving a high clock rate, super pipelining has several drawbacks when considering deploying into an ASIP. First, the stall penalty of having a deep pipeline is significant for an ASIP. Secondly, for this work, the main intention when designing an ASIP is not to maximize the clock rate but to achieve a sufficient level of energy efficiency and application flexibility. Additionally, a high clock rate typically means high power consumption, which is against ASIP characteristics. Therefore, an implementation of super pipelining on ASIP is neither carried out nor described in this work.

The second studied architecture is superscalar. Despite its superior characteristics in allowing high instruction throughput and functional unit utilization, its hardware schedule is very complex and expensive for an ASIP and this work. Thus, a superscalar model is not implemented in this work.

Finally, VLIW architecture is studied, showing its ability to statically issue multiple instructions via compiler techniques. Compared with the hardware scheduler in superscalar architecture, the tacking compiler is simpler and less expensive in the VLIW model. Furthermore, other techniques can still be deployed on top of the VLIW model to resolve pipeline hazards, hide latencies, increase instruction throughput, and function unit utilization. Therefore, a VLIW model is selected for the implementation.

3.5 pktAsip core

Fortunately, before conducting this work, an ASIP named pktAsip with 4-way VLIW implementation is already developed on top of an ASIP with bit manipulation extension [12] and available for radio packet processing. Therefore, this section describes an overview of the pktAsip core, its internal pipeline, fetching, and controlling logic.

3.5.1 Overview

pktAsip is a 4-way VLIW ASIP developed for radio packet processing, with each lane capable of issuing and executing 32-bit instructions via independent functional units.

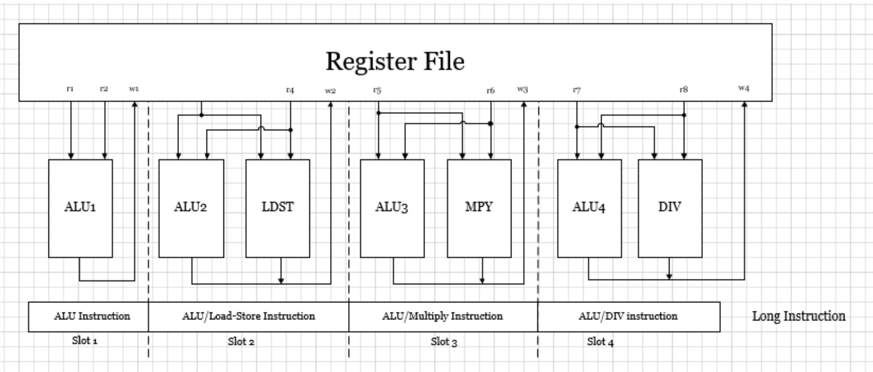


Figure 19: A 4-way (4 issue slots) VLIW core with each slot has its own ALU

In Figure 19, there are four ALUs, with each issuing slot having its own ALU. Each ALU requires two read ports and one write port to register file access. For example, slot two has an extra Load-Store Unit (LDST), which requires three read ports and one write port. Next, slot three has an additional multiply unit (MPY), and slot four has a division unit (DIV), each requiring 2 read ports and 1 write port.

pktAsip features variable length instruction property. Variable length instruction means that when there are not enough instructions to fill all the VLIW slots, short instructions are present instead to improve the code density. In the case of pktAsip, the wide instruction is 128-bit. A shorter instruction is visible to the pipeline when one or more slots are not filled. For example, they could be 32-bit(for one filled slot), 64-bit(for two filled slots), or 96-bit(for three filled slots).

```
 896  add     x13, x0,  x0     | mv     x20, x1      | li     x12, 0       | addi   x2,  x2,  -4
 912  add     x11, x0,  x0     | li     x10, 0       | call   -144
 924  lui     x13, 0           | li     x10, 1       | li     x11, 0       | li     x12, 1
 940  call    -172
 944  lui     x13, 0           | li     x12, 20      | li     x11, 0       | li     x10, 10
 960  call    -192
 964  lui     x13, 0           | li     x10, -25     | li     x12, 4       | li     x11, -1
 980  call    -212
 984  lui     x13, 0           | li     x10, -122    | li     x12, 53      | li     x11, -1
1000  call    -232
1004  lui     x10, 1           | li     x21, 17
1012  addi    x22, x10, -2048
1016  c.swsp  x22, 0(x2)
1018  do      x21, 36
1022  lw      x23, 0(x2)       | do     x21, 30
1030  lw      x10, 4(x22!)
1034  lw      x12, 4(x23!)
1038  lw      x11, -4(x22!)
1042  lw      x13, 4(x23!)     | call   -274
```

Figure 20: Multiple Lengths Instructions

As can be seen from Figure 20, multiple instructions with different widths are scheduled for code density purposes. For example, instructions at row one (PC = 892), row three (PC = 924), row five (PC = 944), row seven (PC = 964), and row nine (PC = 984) are both 128 bits as all four slots are filled with instructions. Others are partially filled. However, from the decoder and compiler perspective, they see all these instructions as 128-bit wide. When scheduling, the no-ops (NOPs) instructions are inserted into the empty slots by the compiler and then are seen by the decoder when decoding for the sake of complexity reduction.

3.5.2 pktAsip Pipeline

Similar to basic ASIP Core implementation with pipeline discussed in chapter 2.4, pktAsip also features a 5-stage pipeline including Fetch, Decode, Execute, Memory Access, and WriteBack.

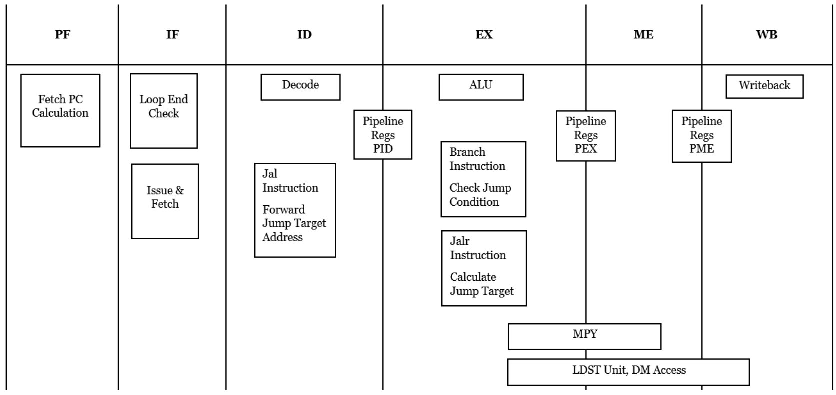


Figure 21: pktAsip Pipeline

However, an extra step is hidden from the compiler, the prefetch stage (PF), shown on the left side of Figure 21 above. This stage occurs before the IF stage. In this stage, the instruction from the program memory is prefetched into an instruction fetch buffer, and the fetch program counter (FPC) is calculated. At the IF stage, besides issuing and fetching a new instruction, the processor controller performs the loop end check to determine whether the current PC matches the loop end address. In a later chapter, more details about the loop checks and related signals that the processor controller triggers will be addressed. The ID stage decodes the fetched instruction from the IF stage and reads corresponding register fields from the Register File. The decoder then outputs decoded operands for different functional units, including ALU, LDST, Multiplier, or Divider, and forwards them into pipeline registers PID.

- For some instructions such as *Jal*, as the jump is direct, the jump target address can be extracted from the instruction itself at the ID stage, and this information is sent to the controller for calculating the next PC and the next Fetch PC (FPC). Furthermore, as the jump is unconditional, the controller can already trigger a jump signal to flush the pipeline (flush all signals come after the Jal), the fetch buffer as well as insert the number of required stall cycles.

At EX stage, there are ALU, LDST, Multiplication, or Division operations with its operands are obtained from the PID registers.

- For branch instructions, as the jump is conditional, the logic for checking whether the branch is taken is implemented at this stage. Depending on whether the branch is taken or not, the controller provides different behaviors. First, a jump signal is initiated with the taken branch to flush the pipeline and the fetch buffer. The controller then inserts a number of stall cycles into the pipeline due to the jump when determining the target address. After that, it updates the PC/FPC with the direct jump target address. With the not taken branch, neither the jump signal is initiated nor PC/FPC is updated with the jump target address. The PC /FPC is, on the other hand, updated with the next sequential instruction address in the pipeline.

- For some instructions, such as *Jalr*, as the jump is indirect, the jump target address needs to be computed using ALU at the EX stage before updating the program counter.
- The multiplication operation starts from this stage and takes two cycles.
- For load/store instructions, the load/store addresses are computed using AGU.

At the MEM stage, for non-memory access operations, data is passed between pipeline registers PEXs and PMEs.

- The multiplication result of 2-stage multiplier is obtained at this stage.
- For load instruction, the data memory read result is read, and for store instruction, the data written to memory is prepared.

Finally, the results are written into registers at the WB stage for instructions that need written access to the register file. For example, for a load operation, the load result is available for writing to register file fields. In contrast, the data memory is accessed and written with the prepared data from the MEM stage for store operation.

3.5.3 Fetching Logic and Processor Controller

As mentioned in chapter 2.7, there is an extra stage for prefetching instructions from program memory into a fetch buffer. The idea of having an instruction buffer is to support the issue of unaligned instructions to avoid sending unaligned addresses directly to the program memory.

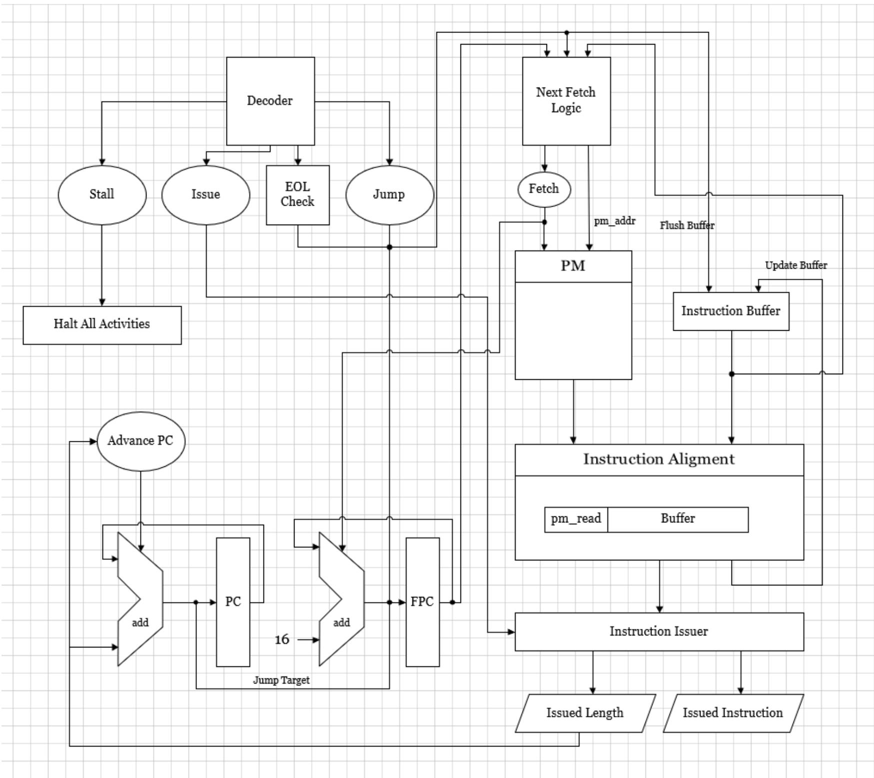


Figure 22: Fetching Logic and Processor Controller. Described and adapted based on original Synopsys example fetch and controlling logic implementation [3]

Figure 22 describes the instruction fetching logic and the calculation of the next instruction program counter. The fetching process starts with the Next Fetch Logic unit on the top right for determining the fetch signal and computing the fetch address (pm_addr) that go to Program Memory (PM). The fetch signal is forwarded to PM when:

- In a normal linear advance of PC and the Instruction Buffer is not full.
- A jump signal indicates a change in control flow.

In these cases, pm_addr is computed from FPC as an input to the Next Fetch Logic unit. At the same time, FPC is updated by adding a value of 16 (bytes) for the next fetch or with the jump destination address. If the buffer is full, indicated by a feedback signal from the Instruction Buffer to the Next Fetch Logic unit, the fetch signal is driven low, and no new instruction is fetched from the PM. After that comes the Instruction Alignment unit. The primary duty of the Instruction Alignment unit is to produce instructions for issuing even with an unaligned address. After producing an instruction for issuing, the Instruction Buffer is updated according to the pm_read from the PM. After an instruction is fetched from the Instruction Alignment unit, the Instruction Issuer handles the issuing. As mentioned earlier, pktAsip features variable length instruction property. Therefore, the instruction width format needs to be determined for issuing. The opcode field inside the instruction contains instruction format information indicating the instruction length (32/64/96/128 bits). Instruction Issuer unit outputs issued instruction and its length for advancing the current PC. PC should remain the current value if the Issued Length has zero value, meaning no instruction is issued. Otherwise, PC is advanced by adding the corresponding Issued Length (PC = PC + Issued Length).

On the top left is the decoder that decodes the fetched instruction from the IF stage. The decoder then generates corresponding control signals: stall, issue, end-of-loop (EOL) related signals, or jump signal.

- The stall signal could arise from pipeline hazards, previous instructions waiting for the memory read access, or when an interrupt request is detected. In this case, the controller halts all the ongoing activities, and no instruction is issued.

- The issue signal is an indicator to the Instruction Issuer for issuing the fetched instruction from the Instruction Alignment unit.

- The jump signal can come from control instructions (jal, jalr), branch instructions, EOL jump, or an interrupt request. When a jump occurs, a target address is computed by the ALU or directly taken from the instruction field. For example, if the jump is an end-of-loop jump, the jump target is the loop start address. Finally, if the jump is caused by an interrupt request, the jump target is the interrupt vector, which is the address of the associated interrupt handler called for the request. This jump target address is then forwarded to both FPC and PC to update these registers with that value. Additionally, the Instruction Buffer needs to be flushed because there is a change in the control flow. That is indicated by the Flush Buffer signals as input to the Instruction Buffer.

Despite the benefit of executing multiple instructions concurrently, when it comes to issue slot utilizations, there is a great challenge in pktAsip. With more issue slots, the issue slot utilization tends to be relatively low in pktAsip. In other words, if the compiler is not able to perform an efficient code generation, poor hardware resource utilization, where all the fields of the long instruction word are not filled [13] can appear. In addition, pktAsip experiences multiple short stalls related to pipeline hazards. When a stall occurs, the whole pipeline is stalled, and no functional unit is utilized. Therefore, some techniques for hiding the stalls and avoiding pipeline hazards for pktAsip are needed, and they are described in the next chapter.

4 Hardware Threads and Multithreading

In computer architecture, multithreading is a traditional solution for reducing the processing time of programs containing tasks that can be run independently by exploiting thread-level parallelism (TLP). A multithread processor can be understood as one in which a single processor can follow a number of streams of execution without the help of software context switches [14] In this work, the study of implementing hardware multithreading on pktAsip is carried out. Hardware multithreading means having multiple thread contexts (threads) in a single-core processor to benefit from task-level parallelism by utilizing the resources more efficiently. This architecture is particularly useful in cases of long instruction latencies and multiple execution units where the compiler cannot effectively utilize all the execution slots of the processors. There are several types of multithreading, each with different scheduling methods for tolerating different sources of latency and utilizing the issue bandwidth in the case of a multi-issue processor. These types will be discussed in this chapter.

4.1 Hardware Threads

In the computer architecture context, a hardware thread is a separate execution context with its own set of registers, stack memory, fetch buffer, and sequential instruction stream for execution. Figure 23 below displays a simple view of the hardware thread.

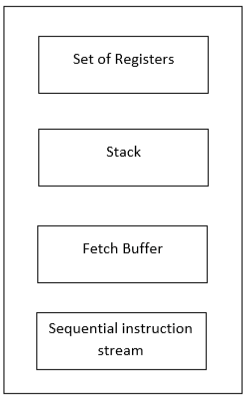


Figure 23: Hardware Thread Component. In the case of pktAsip, each hardware thread contains a register file, some registers like PC, Stack Pointer (SP), or instruction fetch buffer.

Since each thread can independently execute a sequential instruction stream, this enables the creation of a system of multiple threads to execute parallel tasks in a given process. In this system, some common functional units are shared among the threads. Typically, compared to a single processor, a multithreaded one can achieve better functional unit utilization (by counting the number of white squares in Figure 24 below).

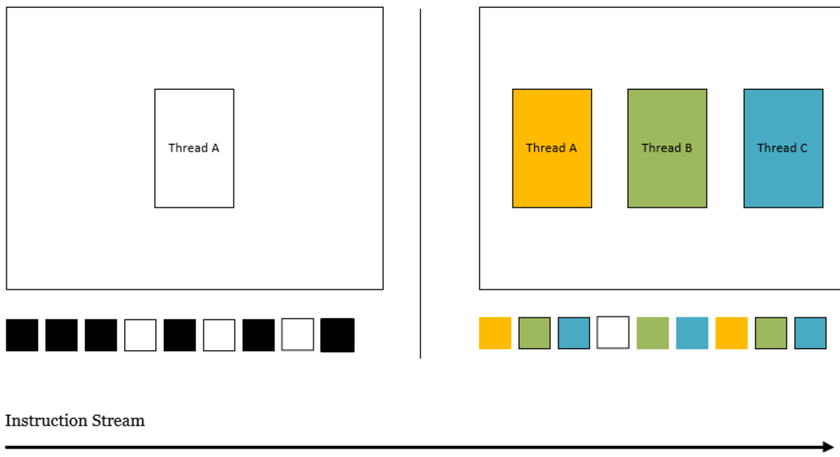


Figure 24: Single thread Processor vs Multithreaded Processor. Typically, a processor with more hardware threads results in a higher functional unit and issue slot utilization.

4.2 Types of Multithreading

There are three types of multithreading, and they are described in this section.

4.2.1 Coarse-grained Multithreading

Coarse-grained multithreading [15] (CGMT) tolerates the latency by switching to another thread when the current thread, during its program execution, encounters a medium or long latency operation. Those latencies typically arise from accessing slow memory, or L2/L3 cache misses. In other words, from the issue bandwidth utilization perspective, CGMT tackles the idea of preventing vertical wastes. Because there can be only one thread at the pipeline at a time, CGMT is sometimes called block multithreading or switch-on-event multithreading. An example of a CGMT pipeline with two threads and four issue slots is introduced in Figure 25 below.

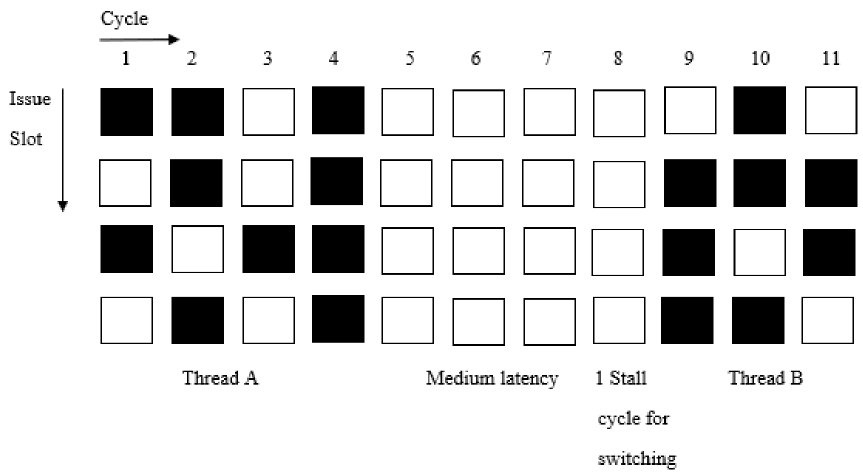


Figure 25: CGMT pipeline with 2 threads and 4 issue slots

CGMT hardware allows the detection of long latency events and the store of the current thread context before switching to a selected thread. As the CGMT pipeline can have only one thread at a time, one advantage of this model lies in the number of required threads to keep the processor busy. Furthermore, single-thread application execution is much less likely to slow down in CGMT. Some examples of coarse-grained multithreaded processors include Sparcle processor [16] Sun MAJC processor [17]

However, CGMT suffers a major drawback: throughput loss exists due to the stall cycle when switching between 2 threads. This stall cycle is needed to flush and fill the pipeline for the following thread. Therefore, the overhead of the switching penalty and start-up cost becomes more significant in the case of processors that has shorter stalls.

4.2.2 Simultaneous Multithreading

Simultaneous multithreading [18] (SMT) is a common multithreading implementation for tolerating long latency events. In SMT architecture, multiple instructions from multiple threads can be dynamically scheduled and issued simultaneously. Figure 26 demonstrates an SMT model with four threads and four issue slots.

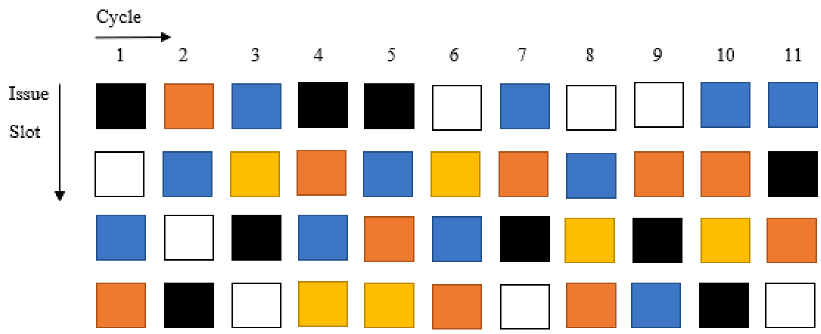


Figure 26: SMT model with 4 threads and 4 issue slots

As seen from Figure 26 above, SMT can issue multiple instructions from different threads into all of its issue slots at the same clock cycle. Looking at multiple instructions from different threads, both horizontal and vertical wastes are tackled and minimized. The latency that arised from the execution of one thread can be hidden by the execution of other threads.

In the SMT model, the functional unit and issue bandwidth are maximally utilized as all the threads compete for the available execution unit. Because of its similarity in dynamic instruction scheduling with superscalar, SMT fits well with superscalar processors. Some examples of simultaneous multithreading include Compaq Alpha 21464 [19] Intel Pentium 4 [11] IBM Power5 [20] or IMB Power6 [21] Though SMT maximizes the processor throughput, the main disadvantage lies in its implementation practically requiring an out-of-order superscalar processor, which is complex and expensive.

4.2.3 Fine-grained multithreading

Finally, the last type of multithreading to mention is fine-grained multithreading [22] (FGMT). Fine-grained (interleaved) multithreading means the processor interleave switches to a new thread in every clock cycle on the same pipeline. Therefore, FGMT sometimes is called interleaved multithreading [23] (IMT). In this mechanism, threads enter and execute in the pipeline one by one in a round-robin fashion, hence the interleaved instruction executions. Whenever one of the threads causes a stall, that thread will be skipped, and another thread instruction will be executed instead. This mechanism allows the elimination of vertical wastes, as there should always be a new instruction from a different thread in every clock cycle. However, horizontal wastes still exist because the elimination depends on how well the pipeline is filling up and how tight the compiler can schedule instructions to fit all the slots in the case of the VLIW processor. Unlike CGMT, the main advantage of FGMT lies in its ability to tolerate and hide both long and short latency: when one thread is stalled, the pipeline can still handle useful work from other threads. As a result, more instructions are executed, hence increasing instruction throughput. Some examples of fine-grained multithreading include the Denelcor HEP [24] Tera [25] or the Sun Niagara [26]

However, the main disadvantage of FMGT is the downgrade in execution time of single thread program: even if the next instruction from one thread is ready to enter the pipeline after the current one, it has to wait for other thread's instructions until its turn. In an N-thread processor, the waiting time is N cycles because each thread only executes once every N cycles. Secondly, because each thread has its own register file, the more threads the processor has, the higher the gain in the area due to the duplication of registers according to the number of threads. Finally, FGMT adds complexity in development and verification, which is resource costly and increases the time-to-market path. However, the work effort and complexity level are not as bad as SMT. An example of FGMT with four threads and four issue slots is displayed in Figure 27 below.

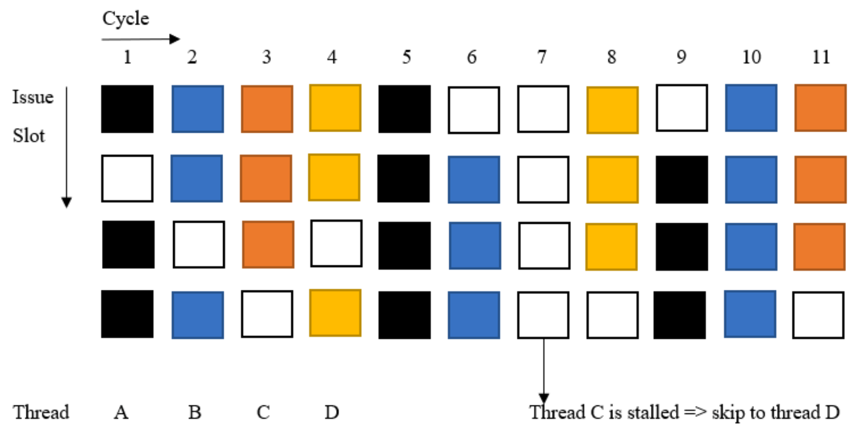


Figure 27: FGMT with 4 threads and 4 issue slots

4.2.4 Multithreading Summary and Model Choice

Different multithreading models have been studied, and they are summarized in Table 2 below.

Table 2: Summary of different multithreading models and their implementations on pktAsip perspective.

	Pros	Cons	pktAsip perspective
Coarse-grained multi-threading	• Requires few threads to keep the core busy • Execution of single thread program is less likely slow down	• Thread switching penalty and thread startup cost could be significant • Area gain in duplication of registers and Register File	• Thread switching penalty and startup cost are significant for pktAsip
Simultaneous multi-threading	• Ability in tolerating long latency • High instruction throughput • Functional units and issue slots are maximize utilized	• Implementation requires an out-of-order superscalar processor • Area gain in duplication of registers and Register File	• Hardware scheduler for out-of-order processor is complex and expensive for pktAsip
Fine-grained multi-threading	• Ability in tolerating both short and long latency • High instruction throughput • Functional units and issue slots are highly utilized	• Execution of single thread program is slow down • Area gain in duplication of registers and Register File	• pktAsip has lots of short stalls related to pipeline hazards so FGMT is a good candidate. • FGMT could potentially remove most of complex bypassing logics in pktAsip

For coarse-grained multithreading, despite its advantage in the number of required threads to keep the core busy, the level of parallelism is insufficient. Furthermore, because pktAsip encounters many short stalls related to pipeline hazards, the thread switching penalty and thread startup cost become significant. Finally, in pktAsip, there do not exist such costly stalls like L2 cache miss or accessing slow memory to benefit from the behavior of CGMT. Therefore, CGMT implementation in pktAsip will not be carried out in this work. However, CGMT could be later introduced if some hardware accelerators are introduced in the data path when there is a need for an easy calling for these accelerators while maintaining a good issue slot utilization.

For simultaneous multithreading implementation, regardless of its ability to keep a high level of issue slot utilization, its implementation on pktAsip requires a hardware scheduler, which is complex and expensive. Therefore, an SMT implementation is not performed in this work.

Finally, fine-grained multithreading has many potentials for implementation on top of pktAsip. pktAsip has several characteristics that need to be considered when adding more parallelism into its architecture. First, it has many short stalls related to control and branch instructions (one-three cycles) as well as medium stalls (<10) due to memory access, and L1 instruction cache misses. This fact opens an opportunity for FMGT implementation in tolerating both short and medium latencies and potentially the removal of the instruction cache. In addition, pktAsip has a 5-stage pipeline with complex bypassing and stalling logic for resolving control and data hazards. Having an FGMT implementation with four threads should remove the need for all the mentioned logic since each thread has its own register file, and instructions from the same thread are four cycles apart, guaranteeing the non-existence of data dependency. Therefore, based on the mentioned feasible benefit, an FGMT with four threads is selected as the implementation target for this work.

5 Implementation of fine-grain multithreading on VLIW core

This chapter describes some work done for the implementation of fine-grained multithreading on pktAsip. During this work, the authors started the work by duplicating some existing registers and Register File in **Section 5.1.**

5.1 Duplication of Registers and Register File

Because each hardware should be seen as a virtual standalone ASIP and all the threads are independent of each other, each thread should have its own set of registers. Therefore, when creating 4-way fine-grain multithreading out of a single thread VLIW core, the first task is to duplicate the register set four times, except the ones that are shared between threads. The ones that need to be duplicated are:

- Register file (X), each contains 32 registers including link register (LR) and stack pointer (SP). Because each thread runs its own program, different stack pointers are expected for multiple threads.
- Program Counter (PC)
- Fetch Program Counter (FPC)
- Loop Start, Loop Count and Loop End registers for loop instruction
- Instruction Fetch Buffer

The registers are shared among all the threads are:

- Thread_ID: Specifying current thread ID in the pipeline.
- Thread_EN: Thread enable status, to keep trach of active threads.
- CSR registers

New registers to be added for the implementation are:

- TK: token registers for supporting mutual exclusion mechanism between threads.

The shared registers between threads need to be notified to the compiler.

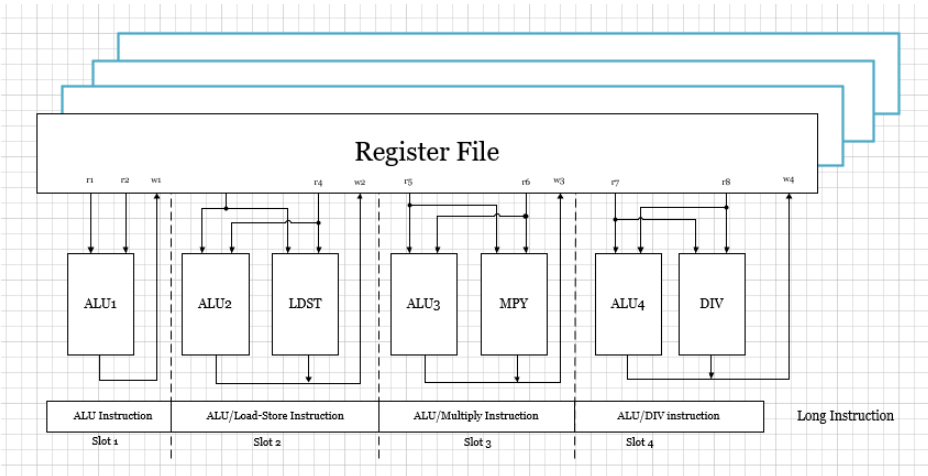


Figure 28: Duplication of Register File. Note that in this figure, only Register File is duplicated 4 times while the number of FUs remain.

Figure 28 above shows the duplication of the Register File in pktAsip as one of the first modifications toward an FMGT model. Note that when Register File is duplicated four times, the associated ports related to it also need to be duplicated. With the current Register File containing 12 ports, the total number of ports needed for FMGT implementation will be 48.

5.2 Modification to Fetch Logic and Controller

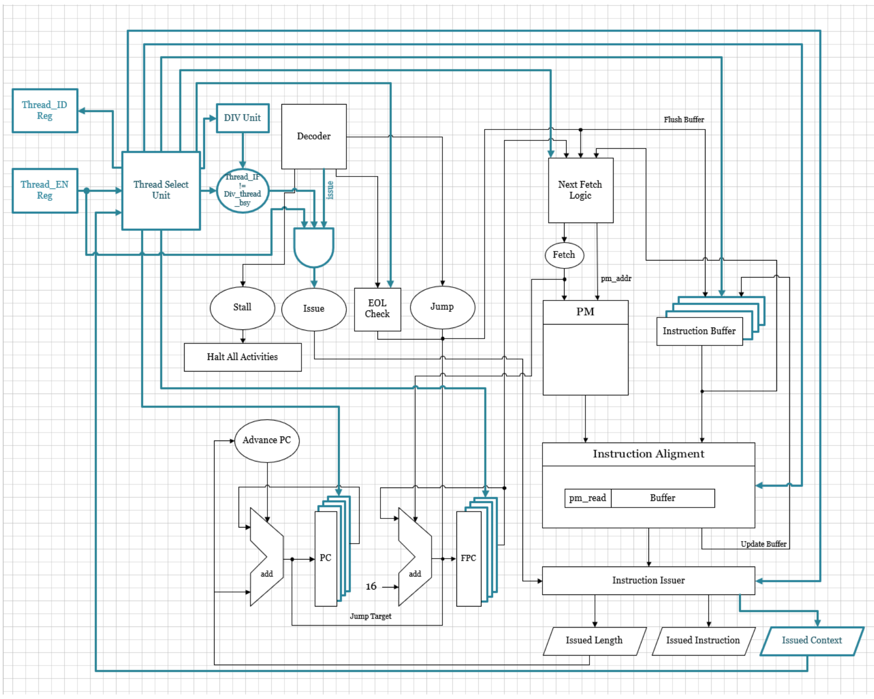


Figure 29: Modification to Fetch Logic and Control for FGMT implementation

The fetch and control logic modifications required for implementing fine-grained multithreading are displayed in Figure 29. The most notable changes are the duplication of registers needed for fetching (FPC and Instruction Buffer) and controlling (PC), which are already mentioned in chapter 5.1. Another significant change is the introduction of a Thread Select Unit (TSU). TSU is responsible for scheduling the threads that enter the pipeline and providing thread (context) numbers for other units and registers. For thread scheduling, as mentioned in chapter 4.2.3, at every clock cycle, a thread switch activity occurs in a round-robin manner: when a thread switches, the following available thread enters the pipeline, and the thread number advances until the last thread. At this point, the first thread is switched back again then the switching process continues until all of the threads have finished their own tasks. With four threads, each is issued every four clock cycles.

TSU also provides the context for writing to the Thread_ID register, to be used in a custom instruction thread_id for indicating the thread number at the pipeline EX stage. More details about this instruction are discussed in chapter 5.4.1. On the other hand, TSU reads the content of the Thread_EN register for issuing an instruction for the switched thread. In case of the switched-to thread is deactivated (Thread_EN register is low), no instruction should be issued, and the core proceeds to the next thread number. In addition to that, as there is only one division unit (DIV) in the core and its operation takes multiple cycles to complete, when one thread is using the division unit, after four clock cycles when the same thread enters the pipeline again, the controller should:

- Not issue any new instruction for the current thread
- Kill any fetched instruction in the pipeline
- Revert the PC to the division instruction address since the current thread is still busy doing the division calculation.

That logic is implemented by having an extra signal called Div_thread_bsy for indicating the thread number that is using the DIV unit. This signal is then compared with the existing thread number at the IF stage (Thread_IF) for determining the issuing of a new instruction.

The fetch logic starts from the Next Fetch Logic. A new signal from TSU is introduced and comes as input to this unit. This signal indicates to which thread the Next Fetch Logic should fetch the instruction. Similar to the change in Next Fetch Logic, there are incoming signals for TSU to all other units, including EOL Check, Instruction Alignment, Instruction Issuer, and to registers, including PC, FPC, and Instruction Buffer. Note that because each thread is supposed to run a different program, they do have different target instructions and contents. Therefore, all the thread register read/write activities and the related units accessing them are tied to their own thread number, guaranteeing the correct fetch and control logic operation. Depending on which pipeline stage each of these units or registers is accessed, TSU provides the corresponding thread number currently at that pipeline stage.

Finally, after issuing an instruction, the Instruction Issuer should provide the issued context (thread) to the Thread Select Unit for its operation to other units. Some new signals are also generated at the end of this phase to connect to other modules, such as the division unit and interrupt unit, to indicate the thread number.

5.3 Instruction Modifications

The core runs on a basic RISC-V instruction set architecture (ISA) with a few other extensions. As with 4-thread interleaved multithreading, pipeline latency is hidden, resulting in instructions from the same threads being four cycles apart. Some control instructions like jump (Jal) can benefit from this as they do not have to be executed in the ID stage but in stage EX instead (shown in Figure 30 below). The uniformity of having all jump instructions executed in the EX stage simplifies the decoder and the logic for advancing PC for a jump.

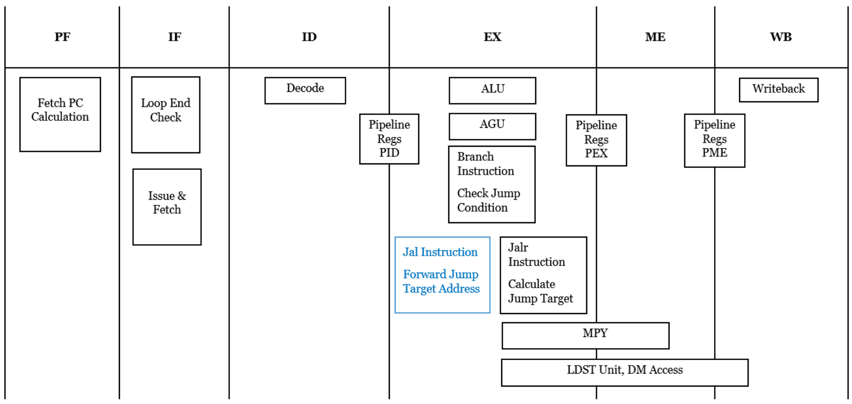


Figure 30: The execution of Jal instruction is moved from ID to EX stage for decoder simplicity

Additionally, certain stalling properties related to control hazards of some instructions and all the bypassing rules are no longer needed. For example, a 3-cycle jump instruction in a single thread core will cause the core to be stalled for two cycles because the target instruction for the next PC is not available until the EX stage. Only when two cycles have passed from the time jump instruction at the EX stage the next instruction after the jump (target instruction) can enter the decode stage. These instructions include:

- Control instructions
- Branch instructions
- Loop instruction

However, in the case of multithreading, there is no data dependency between threads, and control hazards no longer exist. Figure 31 represents an example case. In this example, three consecutive jump instructions belonged to three different threads but resulted in no stall cycle. That is because each thread has its own Register File and Program Counter register, so the latency of one thread when determining the next fetch instruction is hidden by other threads.

		Clock cycle t0	t1	t2	t3	t4	t5	t6	t7
Thread 0	jal x3, 20	IF	ID	EX	MEM	WB	-	-	-
Thread 1	jalr x3, x10, 5		IF	ID	EX	ME	WB	-	-
Thread 2	jal x3, 8			IF	ID	EX	ME	WB	-
Thread 3	beq x8, x3, 28				IF	ID	EX	MEM	WB

Figure 31: Neither data nor control hazard exist in this case as instructions belong to different threads with each thread has their own register file

Division instruction is a particular case as we have only one division unit in the core. This limitation means that when the division unit is busy and occupied by one thread, all other threads with intentions to use the unit must queue for access. In addition, the division instruction is modified to provide the thread number occupied by (Div_thread_bsy signal, which is mentioned in chapter 5.2).

5.4 Custom Instructions

This section lists some custom instructions that are added for FMGT implementation. These instructions are developed with the primary purpose of helping the writing of multithreading programs and providing mutual exclusion mechanisms between threads.

5.4.1 thread_id instruction

This instruction reads the thread_ID register at EX stage and writes the result values at WB stage into destination register X[rd].
This instruction can be used to set different stack pointer (SP) values for different threads. An example of this will be discussed in a test example in benchmarking section.

5.4.2 state_change instruction

This instruction reads the set_bit() function and writes the result into Thread_EN register. This instruction can be used to turn on or off specific thread in the pipeline.

5.4.3 tk_set instruction

This instruction sets the token register (TK). It can be used to implement a mutual exclusion mechanism between threads to allow only one thread to access shared data at a time.

5.4.4 tk_clear instruction

This instruction clears the TK register. Unlike the tk_set instruction, this instruction can be used to release the shared data access by one thread.

5.5 Custom Intrinsics

This section discusses some custom intrinsics for supporting the writing of multithreading programs. They are developed to call the instructions from Section 5.4.

5.5.1 thread_id() intrinsic

This function provides the usage of thread_id instruction at application level. Calling this function in the test program will return the existing thread at pipeline EX stage.

5.5.2 set_bit() intrinsic

This function has three arguments: thread_status, context and value.
It sets a specified context in thread_status with the input value.

- thread_status: a 4-bit value, representing Thread_EN register
- context: a 2-bit value, representing thread number.
- value: a 1-bit value for enable(1), or disable (0) chosen thread.

Example: Calling set_bit(Thread_EN, 2, 1) will turn on thread 2.
Example 2: Calling set_bit(Thread_EN, 3, 0) will turn off thread 3.

5.5.3 tk_set() intrinsic

This function provides the usage of tk_set instruction at application level.

5.5.4 tk_clear() intrinsic

This function provides the usage of tk_clear instruction at application level.

6 Benchmarking Results and Discussions

6.1 Pipeline View of Multithreaded Core

After applying the mentioned modifications in chapter 5, a pipeline example of multithreading pktAsip is presented in Figure 32 below.

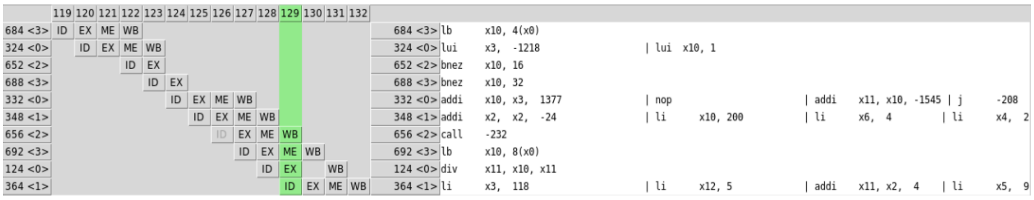


Figure 32: Pipeline view of multithreaded core when running a test program

As illustrated in Figure 32, a new thread is scheduled and entered the pipeline at every clock cycle. The first column on the left indicates the current PC of that thread, followed by the thread number in the pointy bracket. It is presented clearly that each thread runs its own program, indicated by the non-linear PC values between threads. For example, at cycle 119, thread zero enters the pipeline and issues a 64-bit (eight-byte) instruction at address 324. At cycles 120, 121, and 122, threads one, two, and three subsequently enter the pipeline and issue their own instructions. After that comes thread zero again at cycle 123 with the issue instruction has PC (332) is the summation result between the previous PC (at cycle 119, PC = 324) and issue length(cycle 119, length = eight).

6.2 Message Queue Generator and Message Processing

To benchmark the performance of the code, two tests have been taken into use: fast control plane message parsing and user plane message parsing. The motivation behind these tests is to measure the total message processing time of a 4-thread pktAsip core, then compare that with the same test running on a single-thread pktAsip core. In these tests, thread 0 will take control of three tasks, including generating a message queue, starting the message processing, and stopping the program. Figure 33 below presents the process of assigning tasks to different threads and conditional checks to stop the program. At the start of the program, the thread_id() function is called to identify the current thread being executed. Then if the return thread id is zero, a message queue containing a large enough number of messages to be processed is generated by this thread. After the message queue is generated, the next available thread starts processing the message by:

1. Check if the queue is empty and call tk_set() function to lock the queue.
2. Pop the message at the front of the queue by calling MsgQueue.pop()
3. Release the MsgQueue access by calling tk_clear() function
4. Process the message
5. Raise a signal when message processing is done.

The above process is repeated until the MsgQueue is empty, and when all the threads complete their processing, the program will exit and finish.

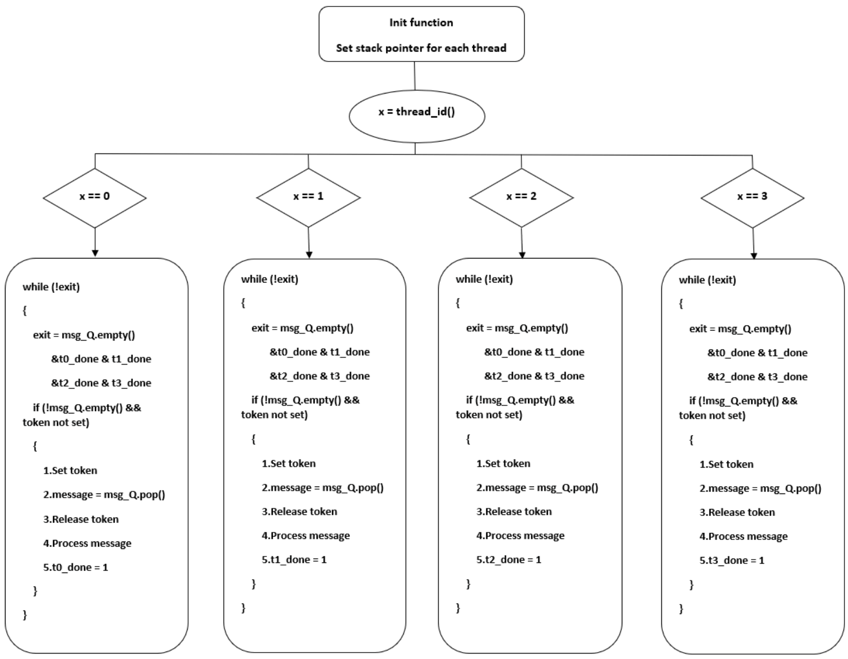
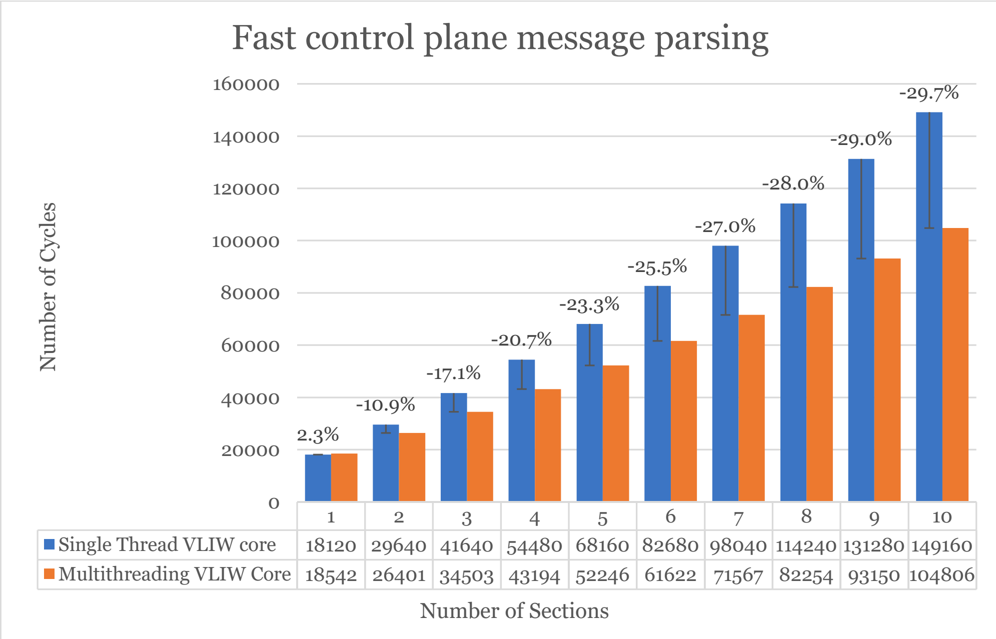


Figure 33: Message processing program with 4 threads

6.3 Fast Control Plane Message Parsing

The Open Radio Access Network (ORAN) fast control plane messages are generated from a python script. A sample of 120 messages was generated via a python script with the number of sections ranging from one to 12. These test samples are then pushed into the message queue to be processed both by single-thread pktAsip and multithread pktAsip.



	1	2	3	4	5	6	7	8	9	10
Single Thread VLIW core	18120	29640	41640	54480	68160	82680	98040	114240	131280	149160
Multithreading VLIW Core	18542	26401	34503	43194	52246	61622	71567	82254	93150	104806

Figure 34: Fast control plane message parsing comparison when running on pktAsip and multithread pktAsip.

Figure 34 represents that, compared to running the test on single-thread pktAsip core, the number of cycles required to parse 120 fast control plane messages decreases proportionally to the number of sections per message, except for one case when there is only one section per message. This result is explained by the fact that with more sections per message, the program contains more control and loop instructions, resulting in fewer spent cycles for stalling and, thus, a shorter total message processing time.

6.4 User Plane Message Parsing

The Open Radio Access Network (ORAN) user plane messages are generated from a python script. A sample of 200 messages was generated via a python script with the number of sections ranging from seven to 30. These test samples are then pushed into the message queue to be processed both by single thread pktAsip and multithread pktAsip.

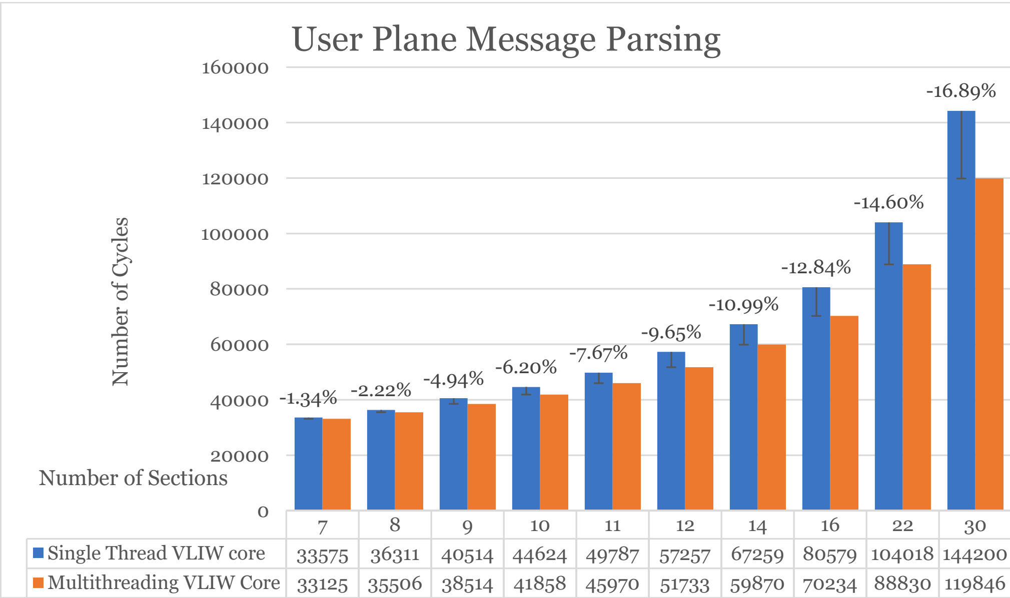


Figure 35: User Plane Message Parsing

Figure 35 above shows different test cases of parsing 200 user plane messages containing different sections per processed message. Comparing the performance of running the same test cases on a single thread pktAsip core, the multithreaded pktAsip results in a reduction in total processing time. The reduction can be seen from parsing a minimum number of seven sections per message. Similar to fast control plane message processing on multithreaded pktAsip, with more sections per message, the test program contains more control and loop operations. As a result, multithreaded version leads to fewer spent cycles for stalling, resulting in shorter total processing time.

6.5 Hardware Synthesis Result

Table 3 below shows the synthesis results between multithreaded pktAsip core and single-thread pktAsip core.

Table 3: Synthesis Results Comparison between single-thread pktAsip and multithread pktAsip

	Relative Area	**Register File percentage on the processor**	**Relative Power**
Single-thread pktAsip	1	0.239	1
Multithread pktAsip	1.72	0.556	1.016

It is clearly visible that with four threads, the gain in the area is primarily dominant from duplication of registers as expected, while the gain in the total power is insignificant (<2%). The register file area impact relative to the total core area is getting much more prominent, from 23.9% to 55.6 %, while the core area increased 72% from its original number. 72% gain in total area is considered significant in a design. However, placing the core into a larger system where it typically is used, at an SoC level, where there are other components that dominate the chip area much more than the core, such as memories, interconnect system, or IP block wrappers, the amount of area increase could be acceptable considering the gain in instruction throughput. Therefore, it is potentially reasonable to implement FGMT on pktAsip with the sacrifice in the mentioned area gain. Table 4 below lists the results for fast control message and user plane message parse tests as well as the synthesis numbers for both processors.

Table 4: Summary Result

	Single-thread pktAsip	**Multithread pktAsip**	**Gain**
Control plane message parsing 1 section	18120	18542	-0.02
Control plane message parsing 2 sections	29640	26401	0.11
Control plane message parsing 3 sections	41640	34503	0.17
Control plane message parsing 4 sections	54480	43194	0.21
Control plane message parsing 5 sections	68160	52246	0.23
Control plane message parsing 6 sections	82680	61622	0.25
Control plane message parsing 7 sections	98040	71567	0.27
Control plane message parsing 8 sections	114240	82254	0.28
Control plane message parsing 9 sections	131280	93150	0.29
Control plane message parsing 10 sections	149160	104806	0.30
User plane message parsing 7 sections	33575	33125	0.013
User plane message parsing 8 sections	36311	35506	0.022
User plane message parsing 9 sections	40514	38514	0.049
User plane message parsing 10 sections	44624	41858	0.062
User plane message parsing 11 sections	49787	45970	0.077
User plane message parsing 12 sections	57257	51733	0.096
User plane message parsing 14 sections	67259	59870	0.110
User plane message parsing 16 sections	80579	70234	0.128
User plane message parsing 22 sections	104018	88830	0.146
User plane message parsing 30 sections	144200	119846	0.169
Relative Area	1	1.72	0.72
Relative Power	1	1.016	0.016
Register File Percentage on the processor	0.239	0.556	4.00

6.6 Limitations

Despite the performance gain with multithreading, there are several notable limitations. The first limitation is the complexity of the design when adding a multithreading feature. Compared to the original fetch and control logic introduced in chapter 3.5.3, a significant amount of work is needed to have a working version, as shown in chapter 5. In addition, extra efforts need to be spent on modifying or creating applications for running on multithreaded core. The amount of work and resources need to be considered when implementing such a big feature not to affect the project and product timeline, especially when the time-to-market is critical in SoC development.

Secondly, the question of how well the multithreaded core can be used in real-world applications or in a larger system needs to be discussed. A typical SoC system can contain multiple similar cores arranged in a cluster manner. The tasks' loading to each core is typically handled by Direct Memory Access (DMA). With the introduction of four hardware threads for all the cores, each thread is seen as a separate virtual core. Therefore, the workload for DMA can easily become more extensive, especially in the case of keeping the work balance between all the cores and ensuring that hardware threads are filled up with enough work not to waste the added resources. As a result, the actual performance might be less expected or even worse with insufficient jobs to run on threads. As mentioned earlier, the execution time is likely worsening when running a single-thread program.

7 Summary and Potential Future Improvement

This chapter summarizes all feasible study findings, implementation, and benchmarking results. Some limitations during this work are also addressed. Finally, some proposed improvements are introduced to serve as a future work reference for the multithreaded pktAsip.

7.1 Summary

In this thesis, the initial goal is to implement a feasible study on different architectures that could be deployed into an ASIP to improve the instruction throughput, functional unit utilization, and hiding the processor stalls. Different models have been studied, including super pipelining, superscalar, and VLIW. VLIW is then chosen as an implementation on ASIP due to its superior ability to issue multiple instructions simultaneously while keeping the work effort at a sufficient level.

pktAsip is a Nokia in-house developed processor for radio packet processing with 4-way VLIW architecture. However, pktAsip features many short stalls related to pipeline hazards that downgrade the core performance. Therefore, a further study about applying multithreading on pktAsip to tolerate pipeline stalls latencies is carried out. Different multithreading models are put into the research, and fine-grained multithreading is chosen due to its ability to hide short and long stalls.

An implementation of four threads on pktAsip is chosen. For the implementation, some modifications are required to the existing core. First is the duplication of existing Register File and other registers that are not shared between threads. With four threads, the duplication is four times. Some new registers are also added, for example, Thread_ID for keeping track of existing thread number in the pipeline or Thread_EN for keeping track of enabling status for all the threads. Next is the modification to the fetch and controller logic. A thread select unit (TSU) is introduced, with the responsibility lying in scheduling the threads entering the pipeline and providing different thread numbers to other functional units. Finally, some new instructions and intrinsics are added to serve as helpers in writing multithreaded programs in pktAsip.

For benchmarking activities, two tests program were used to determine the performance of multithreaded pktAsip. Both programs significantly reduce total processing time when processing a high number of message sections. A basic synthesis is done on multithreaded pktAsip, resulting in an increase of 72% in the total core area. Most of the changes arise from the duplication of the register file and the fetch buffer.

7.2 Future Improvements

Due to the limitation in the timeline in developing the core as well as the limitation from tool for developing, some features could have been added. Nevertheless, they should be mentioned here for potential future implementation.

7.2.1 Separation of Register File

As hardware synthesis results present, with VLIW cores that need multiple read and write ports, the register file and instruction buffer size can easily grow relative to the number of intended hardware threads in the core. However, when the register file and instruction buffer size become considerable and dominate the core area, the performance (or power)/area metrics might get worse to the point where the gain in the instruction throughput is no longer worthy. There is a potential solution for reducing register file size by reducing the number of accessed read/write ports. That is by reusing the read/write ports of one or more execution slots in other slots in different stages. For example, with four threads, there are four cycles apart between two instructions from the same thread. That provides the possibility to move all the single-cycle calculation instructions writing into the register file from the WB stage into the ME stage for two other lanes. For example, two lanes can access the register file using the same write port at different stages without violating the structure hazard rules. However, the current tool for developing pktAsip does not support the full separation of the register file.

A handcraft RTL version of the core was created with the suggested solution, with the number of ports reduced from 12 to seven, showing a reduction in the area of the shared register file by 31%, which is 19% total core area reduction compared to the original version.

7.2.2 Registering Loop Start Instruction

At the existing EOL Check Unit in the processor controller, at loop end address, if the loop is not finished, the EOL Check unit issues a jump signal to:

- Indicate a jump to loop start instruction, then update PC and FPC with the loop start address.
- Flush the instruction buffer

However, if the length of the loop start instruction is too long that the Instruction Issuer Unit cannot issue that in a single cycle, the core has to stall for loading the remaining words before issuing. In typical control plane message processing, there are typically many loop instructions, so the stall from jump to loop start instruction can be problematic. Therefore, having some solutions to register the loop start instruction could theoretically remote this kind of stall.

7.2.3 Sharing The Same Program Memory Between Multiple Threads

With the current multithread implementation, each thread will likely handle an independent task. Therefore, the memory size allocation to each thread should be equal to the size that is allocated to the whole single thread core. The allocation means the total size of the program memory needs to be big enough to accommodate all the programs from all the threads. However, even if each thread will handle an independent task, for pktAsip, the tasks are somewhat similar to each other. The similarity in processing tasks creates an opportunity for some dynamic accessing of the same program memory between threads for reducing memory area impact at the system level outside the core.

8 Reference

[1] Villems, Markus. (2022). Application-Specific Processors for High Throughput, Low Latency, and Flexible 5G Communication SoCs in Synopsys Technical Bulletin. Retrieved 2022-29-10 from https://www.synopsys.com/designware-ip/technical-bulletin/5g-asips-communication-socs.html

[2] S. Shahabuddin, A. Mämmelä, M. Juntti and O. Silvén. (2021). ASIP for 5G and Beyond: Opportunities and Vision in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 68, no. 3, pp. 851-857. doi: 10.1109/TCSII.2021.3050785.

[3] Synopsys. Synopsys ASIP Designer. Retrieved 2022-05-03 from https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[4] Liem, C.; May, T.; Paulin, P. (1994). Instruction-set matching and selection for DSP and ASIP code generation, Proc. EURODAC94, pp. 31-37.

[5] Dake Liu. (2008). Embedded DSP Processor Design: Application Specific Instruction Set Processors. MA: Elsevier Mogan Kaufmann, p.109.

[6] W. Chen, D. Liu and S. Liu. (2022). Register allocation compilation technique for ASIP in 5G micro base stations in China Communications, vol. 19, no. 8, pp. 115-126. doi: 10.23919/JCC.2022.08.009.

[7] G. Paulin, R. Andri, F. Conti and L. Benini. (2022). RNN-Based Radio Resource Management on Multicore RISC-V Accelerator Architectures in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 9, pp. 1624-1637. doi: 10.1109/TVLSI.2021.3093242.

[8] S. Malkowsky, H. Prabhu, L. Liu, O. Edfors and V. Öwall. (2019). A Programmable 16-Lane SIMD ASIP for Massive MIMO in IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-5. doi: 10.1109/ISCAS.2019.8702770.

[9] RISC-V Organization. (2017). The RISC-V Instruction Set Manual. Retrieved 2022-08 from https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[10] Patterson, D. and Hennessy, J. (2018). Computer organization and design. 1st ed. Cambridge: Morgan Kaufmann, p.238, p.285.

[11] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. (2001). The microarchitecture of the Pentium 4 processor. Intel Technology Journal.

[12] Södergren, Oskar. (2021). RISC-V Based Application-Specific Instruction Set Processor for Packet Processing in Mobile Networks. [Master thesis, Linköping University].

[13] A. Dolejsi Santos, A. Wolfe and E. S. T. Fernandes. (1997). Functional units utilization in a multiple-instruction issue architecture, in proceedings 23rd Euromicro Conference New Frontiers of Information Technology - Short Contributions, pp. 228-233.
doi: 10.1109/EMSCNT.1997.658471.

[14] Nemirovsky, Mario and M.Tullse, Dean. (2013). Multithreading architecture. Morgan & Claypool.

[15] W.-D. Weber and A. Gupta. (1989). Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In Proceedings of the International Symposium on Computer Architecture, pp. 273–280.

[16] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle. (1993). An evolutionary processor design for large-scale multiprocessors. IEEE Micro.

[17] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliam and S. S. Tse. (2000). The MAJC architecture: a synthesis of parallelism and scalability, in IEEE Micro, vol. 20, no. 6, pp. 12-25.
doi: 10.1109/40.888700.

[18] M. Gupta, F. Sanchez and J. Llosa. (2010). CSMT: Simultaneous Multithreading for Clustered VLIW Processors, in IEEE Transactions on Computers, vol. 59, no. 3, pp. 385-399
doi: 10.1109/TC.2009.96.

[19] Keith Diefendorff. (1999). Compaq chooses SMT for Alpha. In Microprocessor Report, Vol 13.

[20] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer and J. B. Joyner. (2005). POWER5 system microarchitecture, in IBM Journal of Research and Development, vol. 49, no. 4.5, pp. 505-521.
doi: 10.1147/rd.494.0505.

[21] R. Berridge et al. (2007). IBM POWER6 microprocessor physical design and design methodology, in IBM Journal of Research and Development, vol. 51, no. 6, pp. 685-714.
doi: 10.1147/rd.516.0685.

[22] W. M. Zuberek. (2001). Performance analysis of enhanced fine-grain multithreaded distributed-memory systems, at IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236), pp. 1101-1106 vol.2
doi: 10.1109/ICSMC.2001.973067.

[23] Ungerer, T., Robič, B. and Šilc, J. (2003). A survey of processors with explicit multithreading, ACM Computing Surveys, 35(1), pp. 29–63. Retrieved 2022-06-08 from:
https://doi.org/10.1145/641865.641867.

[24] Burton J. Smith. (1982). Architecture and applications of the HEP mulitprocessor computer system. In Proceedings of the International Society for Optical Engineering.

[25] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. (1990). The Tera computer system. In Proceedings of the International Conference on Supercomputing, pp. 1–6.

[26] L. Geppert. (2005). Sun's big splash [Niagara microprocessor chip], in IEEE Spectrum, vol. 42, no. 1, pp. 56-60
doi: 10.1109/MSPEC.2005.1377878.

Appendix A – Register File

x0 – Zero
x1
x2 - SP
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31
PC

Figure 36: RISC-V base integer register file structure

The length for each field in the figure 37 above has a value of 32, corresponding to 32-bit integer number for each register.
Register field x0 is hardwired to 0 meaning writing to x0 is forbidden. Each register has its own purpose:

- x1: return address
- x2: stack pointer (SP)
- x3: global pointer
- x4: thread pointer
- x5: Temporary/alternate link register
- x6-x7: Temporaries
- x8: Saved register/frame pointer
- x9: Save register
- x10-x11: Function arguments/return values
- x12-x17: Function arguments
- x18-x27: Saved register
- x28-x30: Temporaries