NANYANG TECHNOLOGICAL UNIVERSITY

NANYANG TECHNOLOGICAL UNIVERSITY

APPLICATION MAPPING ON A MANY-CORE OVERLAY ARCHITECTURE

by

HUANG QIAN
(G1502141G)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2016

Contents

List of Figures

List of Tables

Abstract

While reconfigurable computing architectures have shown better performance over processor based systems, they are not widely used beyond specialist application domains such as digital signal processing and communications. It is mainly due to the poor design productivity, which limits their effective use to experts in hardware design. Coarse grained overlay architectures (CGRAs) have emerged as an attractive solution for improving design productivity by offering fast compilation and software-like programmability. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction, but at the cost of area and performance overheads. To deal with these overheads, one innovative method is to present a many-core processor array, which adopts the DSP blocks as functional units (FUs) in the CGRA-based architectures. Moreover, the interconnect between different FUs also needs to be carefully designed as it may cost massive area for routing resources.

This report presents a potential direction to optimize our previous work, TMFU overlay, a linear array of time multiplexed FUs. It is an area efficient FPGA overlay which processes data in a streaming manner, consuming minimum interconnect resources. In addition, the requirement of look-up tables (LUTs) and flip-flops (FFs) decreases by implementing DSP48E1 primitive as the processing core. However, it suffers from throughput penalty due to a high value of initial interval (II). We then propose an improved overlay architecture with two time multiplexed FUs working in parallel at one stage. It comprises of two different structures: fully parallelism and first stage parallelism. Simulation results show that the II can be reduced almost half value as the No. of FUs is doubled. We also investigate the internal architecture of floating point operators and selects the most suitable structure according to the TMFU overlay. Finally, we evaluate the performance of TMFU overlay integrated with ARM processor on Zynq platform for a set of benchmarks, and then the result is compared with that of Vectorbox MXP and draw a conclusion.

Chapter 1

Introduction

1.1 Motivation

Coarse grained overlay architectures have emerged as an attractive solution for improving design productivity by offering fast compilation and software-like programmability. Other advantages include application portability across devices, better design reuse, and rapid reconfiguration that is orders of magnitude faster than partial reconfiguration on fine-grained FPGAs. Although research in the area of coarse grained overlay architectures has increased recently, the field is still in its infancy with only a few FPGA overlay architectures demonstrated in prototype form [4, 5, 6]. Area and performance overheads have prevented the realistic use of overlays in practical FPGA-based systems, limiting their use to very small compute kernels [7]. One of the main reasons for this poor performance is that many of the early overlay architectures are designed without serious consideration of the underlying FPGA architecture. To better target the FPGA fabric, an overlay architecture with FUs based on Xilinx DSP hard macros was proposed in [8]. This resulted in a highly pipelined overlay which maps multiple compute kernel operations to a single DSP block with a significant reduction in the number of FUs required, and hence the routing overhead. Even though a data throughput better than the direct FPGA implementation using Xilinx Vivado HLS was reported, the overlay is still relatively small due to the very high area overheads associated with the programmable routing network between the FUs,

and consumes almost all of the fabric resources while supporting only small compute kernels.

In the previous work [9], we developed a novel overlay architecture which attempts to significantly reduce the area overheads associated with the inter-FU routing network while still maintaining fast compilation and software-like programmability characteristics. Instead of an array of tiles, where a tile consists of an FU and programmable interconnect (usually as island style or nearest neighbour (NN) connections), we utilized a linear array of FUs where the FU is shared among compute operations and the interconnect reduces to a time multiplexed single point-to-point link. We tried to maximize the use of embedded hard macros, specifically DSP48E1 and RAM32M primitives, in the design of the tile architecture to minimize the use of fine grained FPGA resources. The results presented show a significant reduction in area requirement for a set of compute kernels, with a reduction of up to 85% in the FPGA resource compared to existing throughput oriented overlay architectures, an operating frequency in excess of 300 MHz on the Xilinx Zynq, but at the cost of reduced throughput. To further improve the computational density of this overlay, we explore new structures for the time multiplexed FU (TMFU) and interconnection between the upper and lower TMFUs. We also aim to develop an efficient interface between the accelerator and an ARM processor on Zynq-7000 platform, which demonstrates the benefit of overlay architecture in a systematic view.

1.2 Contribution

Our project is mainly based on the proposed TMFU overlay. We exploit the microarchitecture of this overlay and improve the throughput with twice resources of FUs. The TMFU overlay is successfully integrated with Zynq via Xillybus to form a more systematic design. The main contributions is summarized as follows:

- II reduction by doubling the ALUs, register files, and instruction memories at each stage.
- Feasibility test for implementing floating-point FUs on the proposed overlay.

- Integration with ARM processor on Zynq platform using Xillybus, and performance evaluation is provided in comparison with commercial soft vector processors, MXP.

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information and a general literature survey on overlay architectures for the coarse-grained many-core overlay architectures. In Chapter 3, we present a time multiplexed functional unit and a linear array of these units as an Overlay architecture. Chapter 4 analyses architecture in three types of floating point operators. Based on the characteristics of operators, we select one structure to optimize TMFU overlay. Chapter 5 investigates the principle of Xillybus, and uses Xillybus to integrate TMFU overlay with an ARM processor on Zynq. Performance results of our system are compared with Vectorbox MXP. We conclude in Chapter 6 and discuss future work.

Chapter 2

Background and Literature Survey

Many coarse grained overlay architectures have been proposed to implement on top of an FPGA. They are generalized to 4 species in terms of interconnects: spatially configured, time multiplexed, packet switched, and circuit switched. Among these overlays, SCFU-SCN [9] and TMFU-TMN [9] are the main overlays on our investigation. While SCFU-SCN performs the same operation over the time by computing logic routing with spatially configuring, TMFU-TMN loops over a short list of instructions for kernel operations with time-multiplexed execution. SCFU-SCN overlays suffer from significant area overhead, as each FU has a fixed functionality at run time. With the benefit of time multiplexed FUs, TMFU-TMN can dramatically reduce the utilization of FUs and interconnection resources. However, TMFU-TMN overlay is generally facing the problem of high memory requirement for storing instructions and higher initial interval (II). Recently, several implementations of TMFU-TMN have been investigated and can be summarized into 2 categories: processor-based overlays, and CGRA-based overlays.

2.1 Processor-based Overlays

In essence, processor-based overlay is fulfilled by building the soft core or an array of soft processors on top of the FPGA fabrics. Soft (scalar) processors and soft vector processor (SVPs) are the two major processor-based overlays. Soft (scalar) processor

is an instruction set architecture and soft vector processor is the design of data-level parallelism.

2.1.1 Soft Scalar Processors

A soft scalar processor is achieved by implementing FPGA logic primitives as an instruction set architecture. Xilinx MicroBlaze and Altera Nios II are the commercial industrial soft processors, which are the conventional MIPS-like architecture for better software portability, and suitable for non-hardware exporter to carry out FPGA implementation with specific software build tools (Xilinx EDK and Altera Eclipse). Nevertheless, compared with hardware processors, the FPGA-based soft processor has inferior performance in the utilization of area and power consumption but better flexibility and portability as it is modifiable for the diverse requirement of applications. To against the FPGA vendor soft processors, a few soft processors have proposed as FPGA-centric soft processor families such that the design space is suitable for trade-offs between area and performance [14, 15, 16, 17, 18]. Table 2.1 indicates the detailed characteristics of these processors in frequency, area aspects, operating frequency increases when area utilization reduces, the future tendency is interconnecting multiple soft cores to gain a better performance.

Table 2.1: Soft Scalar Processors

Year	Name of System	Word Width	Device	Max Freq	Area
2005	CUSTARD	8/16/32-bit	XC2V2000	30MHz	1800-2500 Slices
2005	SPREE	8/16/32-bit	Stratix II	90MHz	900-1500 LEs
2012	Octavo	8-72 bits	Stratix IV	550MHz	300-3000 ALUTs
2012	iDEA	32-bit	Virtex-6	540MHz	404 FFs and 335 LUTs
2016	GRVI	32-bit	Kintex UltraScale	375MHz	320 LUTs

- **CUSTARD**

CUSTARD stands for Customizable Multi-threaded Processor, which is the first customizable multithreaded soft processor [15, 19]. CUSTARD is a fully bypass architecture with a 4-stage pipeline. It supports the parameterized amount of threads, threading type, bit-width and custom instructions. In experiment, the CUSTARD

has 2.41times speedup of average performance for all benchmarks with customized instructions compared with the performance of Xilinx MicroBlaze. However, the available frequency range of CUSTARD and all its extend versions is from 30MHz to 50MHz, this is inferior to operating frequency of MicroBlaze (up to 100MHz). Besides, the customized instruction has more area consumption (2times) and less I/O port available.

- **SPREE**

The Soft Processor Rapid Exploration Environment (SPREE) automatically creates synthesizable HDL implementations of soft processor architectures from text-based ISA and datapath descriptions, which facilitates the microarchitecture of soft processors and optimizes the RTL description [16]. SPREE is 3-stage pipeline architecture, which is made up of SPREE RTL Generator and a SPREE Component Library. RTL generator creates a description of datapath and component library restores the RTL code and interface description for each component. The performance of the processor in area consumption has reduced by 9% and increased speed by 11.4% regarding performance-per-area over the fastest-on-average design [20]. SPREE cannot offer same design space as CUSTARD but provides efficient synthesis and optimization of architecture when instruction setting is fixed. Implementation of functional component abstraction limits the complexity of SPREE, however, when specific situations occurs, the functionality and performance of SPREE still be influenced.

- **Octavo**

The main idea of Octavo is to investigate ‶How do FPGAs want to compute?″ [14]. Octavo operates on the Stratix IV FPGA with maximal frequency of Block RAMs (550MHz) and a multi-threaded 10-stages architecture. Octavo was further tiled (or duplicated) in two dimensions, its multi-locality was preserved via logical partitioning [21]. In [22], the way to limit overheads of addressing and control-flow via processing the branch trigger module (BTM), address offset module (AOM) and developing operation efficiency for looping code. Octavo can process on the high frequency as it is a pipeline and multi-threaded architecture. Furthermore, it offers great design

space since its parameterizable. As mention before, the processor contains BTM and AOM which will consume extra clock cycles for execution, this limits performance result.

- **iDEA**

iDEA is a lightweight soft processor with Xilinx DSP48E1 primitive to be the computing core, DSP48E1 primitive leads better performance of resource consumption and speed [17]. LLVM-MIPS compiler is capable of configuring the loopback potential and modify the assembly instruction, which is integrated with iDEA processor to extend functionality. Additionally, such as approaches of internal loopback and external forwarding, they can be performed for diminishing the execution time and gaining better development up to 25% [23, 24]. Nonetheless, the 32-bit multiplication cannot achieve in iDEA since the DSP48E1 primitive only able to execute 25x18 bits multiplier, and also only one DSP48E1 block is used in the processor, it limits iDEA implementation. Performance can be improved by operating these resource as a multi-processor system.

- **GRVI**

GRVI is an FPGA-based RISC-V RV32I soft processor to improve the computational density for MIPS and LUT. The GRVI has numerous advantages such as: efficient area consumption (only 320 LUTs), high operating frequency (375MHz), compact and fast design. GRVI Phalanx is a multiprocessor accelerator via placing multiple GRVI processors on the above of the FPGA whose resources is joint into some clusters (8 GRVI PEs, 12 BRAMs and a Hoplite router) [18]. The Hoplite is 2-D torus directional interconnect Network-on-chip, it has faster performance and good efficiency of area utilization. Through the performance result from GRVI Phalanx, the BRAMs are used up to 100%, which is a challenge for the future design of size development.

Compared with hardware processors and accelerators, these soft core processors cannot gain the same performance and high-speed operation as hardware but good software programmability. To improve the overall performance of soft core processor, efficient routing approach is vital for future design.

2.1.2 Soft Vector Processors

Soft Vector Processors (SVPs) is processed via parallelism of data-level, it is capable of evaluating the tradeoff between performance and area by a hybrid approach. The architecture of most proposed SVPs is similar and implemented by a scalar soft-processor. Scalar soft-processor controls the vector lanes executing custom instructions on the local vector memory. SVPs have three critical obstacles to restrict the development for example complexity of centralized vector register file, difficulty of processing precise exception and higher cost in on-chip vector memory system. However, the SVPs do achieve acceleration of operation, which are much faster than the soft processor [25]. VESPA [26], VIPERS [27], VEGAS [28], VENICE [29] and MXP [30] are the five types proposed implementations of SVPs. Among these implementations, VESPA and VIPERS are established in parallel and also the first generation of FPGA-centric SVPs, VEGAS is the second generation, which is created based on the on-chip memory of FPGAs, VENICE is processed with high operating frequency and low area as the newest version. Finally, the first commercial design VectorBox MXP is created based on VENICE implementation. All implementations are evaluated on EEMBC application. VESPA, VIPERS and MXP are emphasized in below.

- **VESPA**

VESPA is an MIPS-based scalar core with excellent portability, scalability, and flexibility. It integrates with VIRAM [31] which is compatible vector coprocessor and able to implement on any FPGA devices. Compared with soft scalar processor, the VESPA achieves an average speedup from 1.8x (2-lane) to 6.3x (16-lane). Also the tradeoff between area and flexibility can be adjusted by changing length and width of vector length. VESPA can be extended and implemented on the Stratix III FPGA to gain better performance-per-area (up to 34%) compared with the original design.

- **VIPERS**

VIPERS is composed of a single-threaded (Nios II-compatible) scalar core referred as UTIIe, a memory interface unit and a vector processing unit [25]. The improved

version VIPERS is presented in [32], it provides double vector registers and some new instructions with better flexibility than the original design. Under testing on the same benchmark, 16-lanes VIPERS gets up to 25times better performance and a modest 14x area compared with performance if Nios II processor, it is probable to increase saving of 30% area consumption in further via modifying VIPERS.

VESPA and VIPERS are suitable of processing data form 8-bit to 32-bit, but they are still facing some challenges: 1) Range of width should be large enough in vector engine when processing mixed-width data. 2) The overhead of instruction is unavoidable since the processing of byte-sized data requires zero-extended or sign-extended which are unnecessary in operation. 3) The width of memory and cache should be large to satisfy the memory requirements of building a connection between vector register file and on-chip memory (VIPERS) or cache (VESPA), but it is limited in the range of FPGA devices capacity.

- **MXP**

The VectorBlox MXP can be connected with Altera or Xilinx FPGAs thought Avalon or AXI interfaces to be a commercial IP core. The MXP has additional characteristics: fixed-point arithmetic, 2D-DMA support, and a C++ object based application programming interface (API) for higher level programming [25]. MXP can operate at the high frequency over 200MHz with fewer vector lanes (<16), the 64-lane MXP even can reach 918 times faster than a Nios II/f processor for speedup on matrix multiplication.

In conclusion, the SVPs have better performance result by data parallel application, the throughput of SVPs is increased when the vector lanes rise. Nevertheless, the number of vector lanes should be decided based on the clock frequency as when vector lanes increase the clock frequency would decrease. Moreover, a suitable compiler also is vital for the future development.

2.2 CGRA-based Overlays

CGRA-based FPGA overlays are still in the primary stage, the previous critical tendency of implementation focus on the throughput-oriented which is mapping each operation to a single FU and acquiring II to one. However, the hardware source is unavailable, and enough to fit large compute kernel onto FPGA, then design with efficient area consumption is considered. Recently, time-multiplexed FU (TMFU) is proposed, it is developed as mapping large kernel onto overlay with decreasing of throughput. Some critical CGRA-based FPGA TMFU overlays are listed in Table 2.2.

Table 2.2: CGRA-based Overlays

Year	Name of System	Granularity	Device	Arithmetic	Max Freq	Area
2011	CARBON	32-bit	Stratix III	integer	150MHz	3K ALMs,304 FFs
2012	reMORPH	32-bit	Virtex6	integer	400MHz	200 Slice LUTs
2013	TILT	32-bit	Stratix V	Floating point	200MHz	2.7K-14K eALMs
2013	SCGRA	32-bit	Virtex 7	integer	270MHz	1280 LUTs, 318 FFs

- **CARBON**

CARBON requires larger resources with slow operation speed, which significantly limits the scalability of the architecture compared with the other TMFU-overlay. Moreover, when executing the instruction memory for reading operation, the BRAM cannot fully operate, which leads the requirement of extra memory storage for saving bypass data. In the example of CARBON, it operates as a 2x2 array of tiles on FPGA with maximal 256 instructions and 90MHz operating frequency [33].

- **reMORPH**

The reMORPH overlay has finest performance when is targeted with FPGA fabric, the resource consumption of one FU is made up of 1 DSP block, 3 BRAMs, 196 LUTs and 41 registers. The reMORPH can operate on high operating frequency (400MHz) and implement about 40 tiles, each title is consist of 5-stage pipeline ALU and a BRAM to store instruction, it requires little resource utilization [34]. During

the development of reMORPH, reduction of overhead is achieved by the routing and multiplexers. Hence, the decoder is not required, which leads increasing of instruction bit and the overuse of BRAM blocks, the size of overlay rises due to these reasons. Tiles are implemented in the reMORPH overlay, which involves time consumption of changing between the various application kernels such that the operation will be accordingly slow.

- **TILT**

TILT overlay supports 32-bit floating point operation and is a highly configurable engine for FPGAs. The internal FU can be varied and deeply pipelined such that TILT cores can scale internal FU. Each core contains data memory, crossbar switches, and FUs, all cores are sharing a single instance of instruction memory and can be execute in single-instruction-multiple-data (SIMD) mode [35, 36]. Through evaluation of TILT performance, compared with Altera OpenCL HLS implementations, the TILT overlays is suitable to operate on the high frequency (up to 200MHz). It is similar to the HLS implementation but less area overhead (2times less) when throughputs for both designs are equal. However, the update of kernels requires recompilation of TITL system since TITL does not support customized for kernels, which issues extra time consumption of hardware context (38 seconds on average). Moreover, the TITL system has less density of computation when executing less design, but this limitation can be solved by customizing the FU and specific functionality of applications.

- **SCGRA**

SCGRA overlays as a method to address productivity issue of FPGA design, it presents a reduction from 10 times to 100 times in compilation time compared with AutoESL HLS tool [37]. Some SCGRA overlay are implemented on Xilinx Zynq, which can gain a speedup (up to 9 times) and faster implemented on Xilinx ARM processor. Changing of compute kernel is fully processed by reconfiguring FPGA bitstream, then the context switching of applications is faster. Zynq-based SCGRA overlay operates at 250MHz, and FU includes ALU, multiport data memory (256x32 bits) and variable depth instruction ROM. Conversely, large size design of SCGRA

overlay implementation has several challenges: 1) Requirement of BRAM is high for instruction memory, and it is a tradeoff between utilization of BRAM and I/O buffer, these characteristics influence the data reuse. 2) The cost of routing design between PEs gets higher as increasing of size, which causes the performance of computation. 3) As the size of overlay growing, the operation frequency has to decrease, which causes the deteriorative overall performance result.

Despite CGRA-based overlays are the latest proposed design product of FPGAs, in particular for the TMFU overlays which are proposed within last 5 years. According to the present situations, many CGRA-based overlays have significant performance in speed and area efficiency. The abundant potential is discovered when more coarse-grained modules (DSP, BRAM) are implemented onto FPGAs, which has important meaning for accelerating computation loop.

2.3 Summary

In general, the overlay architectures are divided into SCFU-SCN overlays and TMFN-TMN overlays, the particular and comprehensive characteristics of existing TMFU-TMN overlays are emphasized. These overlays are portable for implementation of processor-based and CGRA-based overlays. Nonetheless, TMFU-TMN overlays are facing the issues of large area overhead and requirements of memory based on its design demand. Therefore, a better and efficient interconnect structure is vital for the future development, linear topology as a good solution to reduce the routing cost with intelligent scheduling method and fulfill better location of memory utilization. TMFU overlay has potential benefit and solves the problem with development and optimization of internal architecture.

Chapter 3

Analysis of TMFU and Linear Overlay

In the previous chapter, we emphasized the various coarse grained FPGA overlay architectures, which have emerged to be a good method for improving design productivity and software like programmability. The time-multiplexed FU (TMFU) overlay can dramatically decrease the amount of functional unit (FU) and the resource demand of interconnection. Although highly pipeline FU overlay can handle the high operating frequency, and requires less kernel context switch time. It still suffers the higher value of initial interval (II), which causes a lessened throughput of the system. Hence, the new architecture Linear overlay is proposed to reduce the II by paralleling FUs. In this chapter, we give an introduction to the TMFU architecture, explanation and overview of new Linear overlay architecture. The main contributions of this chapter can be summarized as follows:

- Introduction of operation for diverse benchmarks on the original TMFU overlay, and the proposed Linear overlays which reduce II by paralleling two FUs at one stage.
- Specification and method of the Linear overlay architecture, it has two manners for paralleling stages: 1) Fully Parallel, each stage has two FUs to process the instructions. 2) First-stage Parallel, only the first stage has two FUs and the rest of stages keep same with original overlay architecture (one FU).

- Simulation result of the new linear overlay compared with the original TMFU overlay. In addition, the limitation of this Linear overlays and the future work are presented.

3.1 The TMFU Architecture

The architecture implements a linear connection of FUs to form a processing pipeline, which follows the execution of feed-forward data flow graphs (DFGs) for diverse benchmarks. The 32-bit pipeline TMFU overlay architecture is shown in Fig 3.1, which is composed of two FIFOs and FUs. The architecture of TMFU overlay is same as that of proposed Linear overlay, which also is made up of two FIFOs and a cascade of FUs.

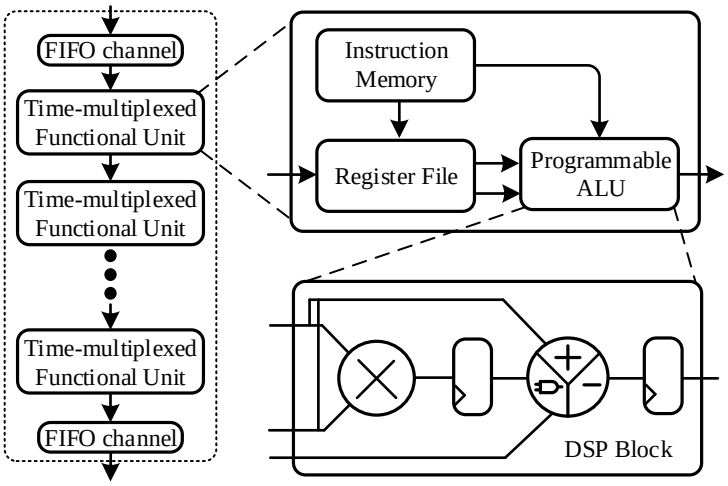


Figure 3.1: TMFU Programmable Processing Pipeline diagram.

3.1.1 Distributed RAM

In Fig 3.1, FIFO channel is Distributed RAM (DRAM) in implementation, the data in DRAM-based FIFO (Top FIFO) feeds into a cascade of FUs, with another DRAM-based FIFO (End FIFO) at the pipeline output. In TMFU overlay implementation, the bit width of data is 32-bit and write depth is 32 in FIFO.

3.1.2 Structure of Function Unit

As the Fig 3.1 showing, the FU is made up of instruction memory, register file and programmable ALU, the detailed information was summarized:

- Instruction Memory (IM): A function of storing input instruction data. IM block is able to offer the corresponding instructions, which are the block output and used by the other functions. In IM block, it also generates the valid output signal via control generator. RAM32M (32 X 6 Simple Dual Port DRAM) is used as the memory of instructions.
- Register File (RF): A function of storing operands for ALU, RAM32M is the storage memory. In RF block, the output operand is obtained from RAM32M through reading the specific address, which is acquired from the instruction of previous IM output.
- Arithmetic and Logic Unit (ALU): DSP48E1 primitive, it is charged with executing arithmetic operations addition, subtraction and multiplication. The execution requires three clock cycles to finish.
- RAM32M primitives: Memory locates in IM and RF, which is implemented in LUTRAMs. It can be configured as a 32-deep 2-bit wide quad port (3 read, 1 read/write), 32 deep 4-bit wide dual port (1 read, 1 read/write) or an 8-bit wide single port (1 read/write) memory.

The FU is mapped to a Xilinx Zynq XC7Z020-1CLG484C using Xilinx ISE 14.7, the synthesized diagram of FU is shown in Fig 3.2. Except the above main blocks, the FU structure also contains some particular blocks to achieve the operation fully.

- Tag Matching: Instruction is 40-bit to be the input of FU, which includes 32-bit FU control instruction and 8-bit tag. ‘Tag ’ is used to compare with the tag of each stage, and distinguish which stage the instruction belongs. Inner FU stores instructions with matched tag number.
- Counter: Three distinct counters are used in IM and RF.
    1. Instruction Counter (IC): 5-bit counter, which is used to keep counting of how many instructions are stored into the FU.

2. Program Counter (PC): Counter counts how many instructions in the FU, and controls the execution of instruction while the 'control' signal is high.
3. Data Counter (DC): It is sequential data counter and counts following the sequence, the number of counts is used when writing data into RF.

- Input Map Logic: The input operands of ALU have diverse bit numbers (a-30bit, b-18bit, c-48bit). Hence, the input map logic is responsible for determining the correct bit and data for ALU input according to the execution of data flow graph (DFG). Moreover, two-bit of instruction is used to check whether the operand is constant or the previous output data from another stage.

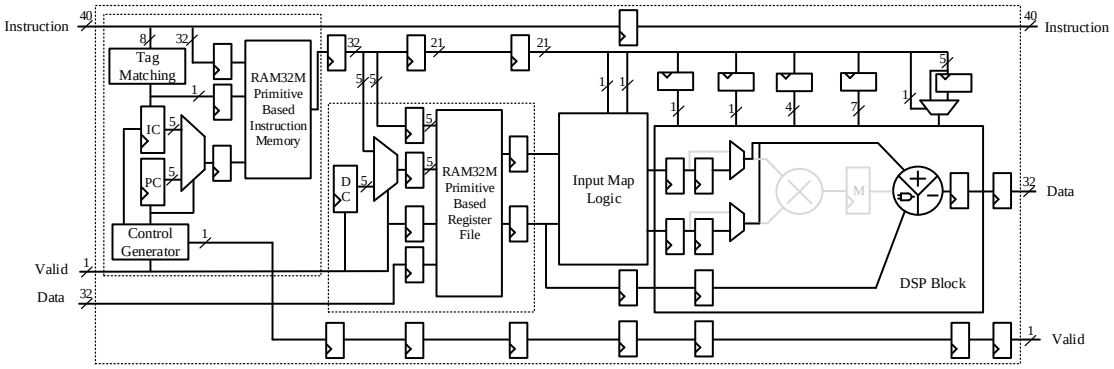


Figure 3.2: Time-multiplexed Functional Unit.

The FU working process roughly summarizes: Data is streamed into the FU and written into RF using a DC when valid signal is high. After all the data is sent into RF, the valid signal is taken low. Then IM starts sending the addresses of operands to the RF, and configuration data is forwarded to the ALU block when the control signal is asserted by control generator. When 'control signal' is high, PC starts counting and controls the sequences of instructions execution by adding one. After implementing all the instructions to scheduling stages, data output sends to the next stage and output FIFO. The DSP block flushes its internal pipeline, and the PC resets allowing the matching sequence of instructions to be reissued. Consequently, only few instructions are executed on each FU, which results in the less logic resource of IM and RF.

3.1.3 Standard of Instruction Format on TMFU

TMFU overlay is 32-bit pipeline architecture, which is capable of executing two 32-bit operands arithmetic calculation in ALU block. ALU operates arithmetic calculation by DSP48E1 primitive. Instruction is 40-bit and includes the address of two operands, constant operand, DSP48E1 setting inputs, tag and determine statements. Instruction of Linear overlay contains similar information as that of TMFU overlay, except the number of instruction bit is distinct. The inputs of DSP48E1 primitive setting: ALUMODE (4-bit), OPMODE (7-bit) and INMODE (5-bit), which are able to decide what operator is executed in DSP48E1 by configuring various number of inputs. Four types of operators are listed in Table 3.1. Accordingly, the ALU block can execute addition, multiplication and subtraction following the DFG (Fig 3.3), ALU data is acquired from instruction. Table 3.2 indicates the meaning of instructions.

Table 3.1: DSP48E1 configuration for each operation

Operation	ALUMODE	OPMODE	INMODE
ADD	0000	011 0011	00000
SUB	0011	011 0011	00000
MUL	0000	000 0101	00000
OR	1100	011 1011	00000

Table 3.2: Meaning of TMFU Instruction

No.of Bit	Function
39-32(8bit)	Tag
31-28(4bit)	ALUMODE
27-23(5bit)	INMODE
22-16(7bit)	OPMODE
15,14(2bit)	Clock Enable of A,B
13(1bit)	multiplex of ALU
12,11(2bit)	input map selection
10-6(5bit)	source1 address
5-0(6bit)	constant (5-1bit: source2 address)

It is noteworthy that the 'input map selection' bit (12-bit to 11-bit) is used to decide the input operands of DSP48E1 primitive (a, b and c). 11-bit is constant selection bit, which is 1 then one of the operand is constant, if it is 0 then the operand is by pass data in Chebyshev benchmark. 12-bit is 'split' bit, when the operation is

multiplication then it is assigned to 1, if the operation is addition or subtraction, then it is 0.

There are two types of instruction: arithmetic and data bypass (used to feed input data to the next scheduling stage). Data bypass is necessary when input data is the operand of after stage. Bypass instruction is implemented by multiplying one with input data. For example, Fig 3.3 is DFG of Chebyshev benchmark, the output data of the first stage needs to multiply with input data on the second stage. Chebyshev benchmark has seven stages, and each FU stores two instructions (one bypass and one arithmetic).

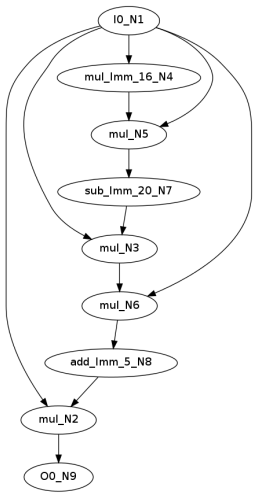


Figure 3.3: Data Flow Graph of Chebyshev.

Table 3.3: Instruction of Chebyshev

TAG	ALU	INMODE	OPMODE	CE	MUX	IMM	SRC1	SRC1/IMM	Function
00000000	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000000	0000	10001	0000101	00	1	01	00000	010000	i/px16=o/p1
00000001	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000001	0000	10001	0000101	00	1	00	00001	000000	o/p1xi/p=o/p2
00000010	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000010	0011	00000	0110011	11	0	11	00001	010100	o/p2-20=o/p3
00000011	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000011	0000	10001	0000101	00	1	00	00001	000000	o/p3xi/p=o/p4
00000100	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000100	0000	10001	0000101	00	1	00	00001	000000	o/p4xi/p=o/p5
00000101	0000	10001	0000101	00	1	01	00000	000001	bypass x1
00000101	0000	00000	0110011	11	0	11	00001	000101	o/p5+5=o/p6
00000110	0000	10001	0000101	00	1	00	00001	000000	o/p6xi/p=o/p

According to the arithmetic operation, the instructions are formatted and listed detailed information in Table 3.3.

3.1.4 Simulation Result of TMFU Overlay

To verify the performance of TMFU overlay whether is correct, ISE Verilog test fixture is used by writing test bench file.

In this scenario, Chebyshev benchmark (shown in Fig 3.3) is used in the simulation, and the inputs of that is set to 1, 2, 3, 4, 5 and 6. After global system resetting, assigning the high signal to write enable bit for loading instructions into the system and delaying 500ns to make sure loading is finished. Then enabling the FIFO inputs to start working, providing input data every 20ns into the input pipe (shown in Fig 3.4). According to the arithmetic expression of Chebyshev benchmark: $Y = x[x^2(16x^2 - 20) - 5]$ the arithmetic calculation result is listed in Table 3.4.

```
1        user_w_write_32_data = 32'h00000001;
2    #20;user_w_write_32_data = 32'h00000002;
3    #20;user_w_write_32_data = 32'h00000003;
4    #20;user_w_write_32_data = 32'h00000004;
5    #20;user_w_write_32_data = 32'h00000005;
6    #20;user_w_write_32_data = 32'h00000006;
```

Figure 3.4: Snapshot of Test Bench for TMFU Simulation

Table 3.4: Arithmetic Result of Chebyshev

Input x	1	2	3	4	5	6
Output Y	1	362	3363	15124	47525	120126

In contrast, the simulation result (shown in Fig 3.5) match the calculation result, which means the TMFU overlay works successfully. Moreover, the II consumes 6 clock cycles for generating one output data in Chebyshev benchmark.

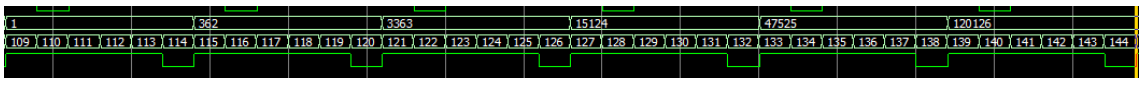


Figure 3.5: Test bench of Chebyshev on TMFU Overlay.

3.2 Fully Parallel Linear Overlay

In the previous section, TMFU overlay is introduced and has a significant reduction in the number of FUs, but at the expense of an increase in the II. Hence, Linear overlay is proposed to solve the problem of higher II. In this section, it is mainly about the architecture and implementation of fully parallel Linear overlay. Compared with the architecture of TMFU overlay (is shown in Fig 3.1), the structure of fully parallel Linear overlay consists of two FIFO channels and a cascade of two FUs, as shown in Fig 3.6. Each stage in fully parallel Linear overlay has two FU, which executes in parallel such that the time consumption of instruction execution decreases and II is lower. The proposed fully parallel Linear overlay still keeps using the same blocks as TMFU overlay. Since the overlay is parallel, then the input bit is changed to 64-bit and some blocks have to use twice for achieving parallelism in FUs.

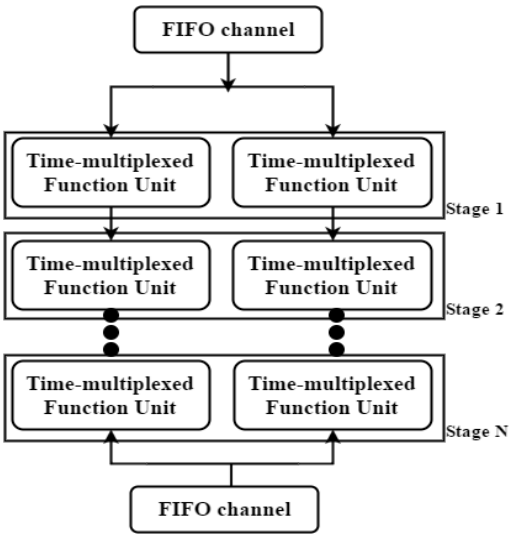


Figure 3.6: Architecture of Fully Parallel Linear Overlay.

3.2.1 Instruction of Fully Parallel Linear Overlay

The Fig 3.3 indicates the detailed function of 40-bit instruction in TMFU overlay, 8-bit starts from the most significant bit (MSB) is tag, the rest of 32-bit is FU control instruction which manages one FU execution. Since fully parallel Linear overlay has

2 FUs in each stage, not only the input bit is doubled, but also the instruction bit is increased to support 2 FUs parallel by combining two FUs control instructions into one instruction. Therefore, the instruction of fully parallel Linear overlay is 73-bit, which consists of tag, selection, and 2 FU control instructions. Compared with the previous instruction of TMFU overlay, there are some differences about:

- Tag: Tag in fully parallel Linear overlay is reduced to 3-bit in compared with TMFU overlay. However, the size of the tag is able to be customized.
- Selection bit: Linear overlay is parallelism of two FUs, which means each stage contains two ALU and RF blocks, then the system must have the ability to choose the corresponded data from one instruction, and feeds into the correct block. Therefore, an extra 4-bit are added into instruction, every 2-bit is implemented to select one FU data.
- FU control instruction: The rest of bits are two FU control instructions, and include information which is same as TMFU overlay. The least 33-bit (src1 address is 6-bit in Linear overlay) belongs to least input.

In summarized, the instruction of Linear overlay is generalized in Fig 3.7:

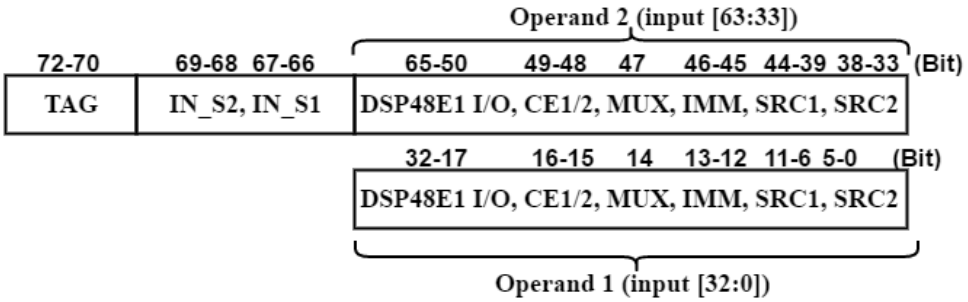


Figure 3.7: Instruction of Linear overlay

Through this regulation, one instruction can control the execution of two FUs at the equivalent time and achieve parallelism. One thing to be noted is the address of operands in instruction, since there are 2 FUs in one stage, which means one extra register can store the data. In TMFU overlay, for the first data series, the addresses of inputs are added one in sequence, but in fully parallel Linear overlay, one stage

contains two registers. Therefore, one instruction includes two same addresses of operands, but these addresses are used for two registers separately. This is the reason why one instruction contains two same addresses of operands.(as shown in Fig 3.8).

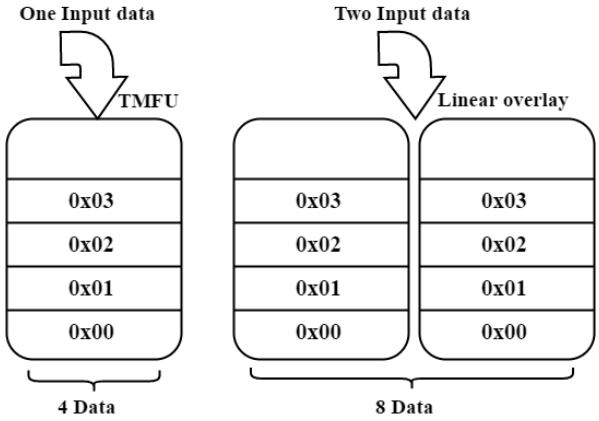


Figure 3.8: Comparison of Address in Instructions

3.2.2 Architecture of Fully Parallel FUs

In the fully parallel Linear overlay architecture, the input and output data is 64-bit (two 32-bit data parallel), the instruction is 73-bit as mention before. In ISE design suite, a full benchmark architecture is made up of 'top_overlay.v 'and 'top_cpu.v' (stage). Chebyshev benchmark has 7 stages (shown in Fig 3.3), then 'top_overlay.v' includes 7 'top_cpu.v' wrapper files to achieve the functionality of 7 stages in the benchmark.

- Top_overlay.v: This file controls overall processing of system, which contains not only FIFOs and the functions of each stage, but also the 'valid' signal which is used to judge when the internal blocks (IM, RF) can start processing. The 'valid' signal is generated according to the clock cycles of II.
- Top_cpu.v: In fully parallel Linear overlay, this file contains one 73-bit IM block, two 32-bit RF blocks, and two ALU blocks. Moreover, the input map logic is divided into two parts for selection of parallel input data as the ALU operands.
    1. IM: The input of IM block is 73-bit instruction, and the function of IM is to store the corresponding instructions, acquire the FU control instruction and send it out. Block RAM memory is used to stored data in IM block.

2. RF: Two 33-bit RF blocks execute data at the same time to fulfill parallelism. Thr structure of RF block in fully parallel Linear overlay is same as TMFU overlay. The RF block captures the corresponding bit from instruction and input data. For instance, when RF executes operand 1, the 32-bit input data is acquired from least significant bit (LSB 0-bit) to 31-bit of 'Din' and the 33-bit instruction is gained from LSB (0-bit) to 32-bit of 'Inst' (output of IM). The outputs of RF are four data sources (operand1: src1/src2 oprand2:src3/src4). Wrapper file is shown in Fig 3.9.
3. ALU: Two DSP48E1 primitives are processed to achieve arithmetic operation. The corresponding input data is earned from 'Inst' according to the standard of Linear overlay instruction (shown in Fig 3.7).

```
/***************** INST MEMORY**************/
inst_mem my_inst_mem(.clk(clk),
.rst(rst),
.valid(valid),
.tag(tag),
.ins(ins),
.inst(inst),
.control_d7(control_d7));
/***************** REGISTER FILE**************/
regfile  my_regfile_1(.clk(clk),    regfile  my_regfile_2(.clk(clk),
.en(en),                                                  .en(en),
.valid(valid),                                            .valid(valid),
.din(din[31:0]),                                           .din(din[63:32]),
.inst(inst[32:0]),                                        .inst(inst[65:33]),
.src1(src1),                                              .src1(src3),
.src2(src2));                                             .src2(src4));
/***************** EXECUTION UNIT***************/
alu_core uut_1 (.clk(clk),          alu_core uut_2 (.clk(clk),
.rst(rst),                                .rst(rst),
.a_i(a_i_1),                              .a_i(a_i_2),
.b_i(b_i_1),                              .b_i(b_i_2), //
.c_i(c_i_1),                              .c_i(c_i_2), //
.alumode_i(inst_d2[32:29]),               .alumode_i(inst_d2[65:62]),
.inmode_i(inst_d2[28:24]),                .inmode_i(inst_d2[61:57]),
.opmode_i(inst_d2[23:17]),                .opmode_i(inst_d2[56:50]),
.cea2_i(inst_d2[16]),                     .cea2_i(inst_d2[49]),
.ceb2_i(inst_d2[15]),                     .ceb2_i(inst_d2[48]),
.usemult_i(inst_d2[14]),                  .usemult_i(inst_d2[47]),
.p_o(p_o_1) );                            .p_o(p_o_2) );
```

Figure 3.9: Wrapper File in Top_cpu.v

Furthermore, in 'top_cpu.v' file, the input map block is modified to adapt the fully parallel Linear overlay, which has two ALU blocks, and system requires two

set of operands to fill into the ALU blocks. Therefore, the 'selection bits' (68-bit to 66-bit) in instruction decides the selection of operand source for ALU block, and 'Imm bit' (13-bit to 12-bit) determines the input of ALU from benchmark arithmetic operation. Since the 'selection bit'is 2-bit, there will be four options to choose for the wanted operands. The detailed information of 'selection bit' in Chebyshev benchmark is listed in Table 3.5:

Table 3.5: Instruction of Chebyshev

In_s1 [69:68]	Data Source	Function
11	Src3/Src4	Operand is constant
01	Src1/Src4	Operand is bypass data
In_s2 [67:66]		
00	Src1/src2	Operand is constant
10	Src3/src2	Operand is bypass data

At the end of one stage, the two outputs ([31:0]) of ALU blocks are combined as a 64-bit data, which is sent to FIFO and stored for next stage input. The FU is mapped to a Xilinx Zynq XC7Z020-1CLG484C using Xilinx ISE 14.7, the processing diagram of fully parallel FU is shown in Fig 3.10.

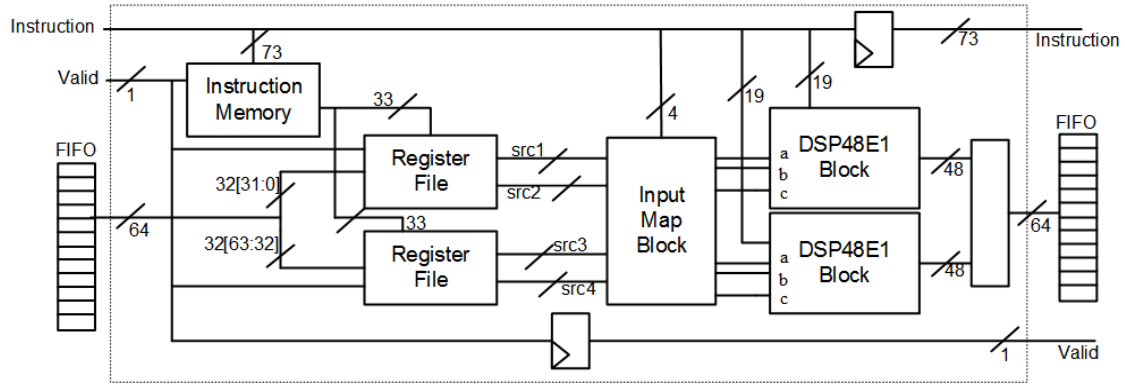


Figure 3.10: Fully Parallel Linear Overlay Process.

3.2.3 Simulation of Fully Parallel Linear Overlay

For the verifying validity of fully parallel Linear overlay, using ISE suit 14.7 Verilog test fixture to write test bench for system verification. The simulation test file of fully

parallel Linear overlay is almost same as that of TMFU overlay, except the input data is changed from 32-bit to 64-bit (a parallel input data) as shown in Fig 3.11.

```
1       user_w_write_32_data = 64'h00000001_00000001;
2     #20;user_w_write_32_data = 64'h00000002_00000002;
3     #20;user_w_write_32_data = 64'h00000003_00000003;
4     #20;user_w_write_32_data = 64'h00000004_00000004;
5     #20;user_w_write_32_data = 64'h00000005_00000005;
6     #20;user_w_write_32_data = 64'h00000006_00000006;
```

Figure 3.11: Test Bench of Fully Parallel Linear Overlay.

The simulation result is shown in Fig 3.12. Compared with the arithmetic calculation (Table 3.4), the simulation is precise, and the II is reduced from six clock cycles to four clock cycles in Chebyshev benchmark.

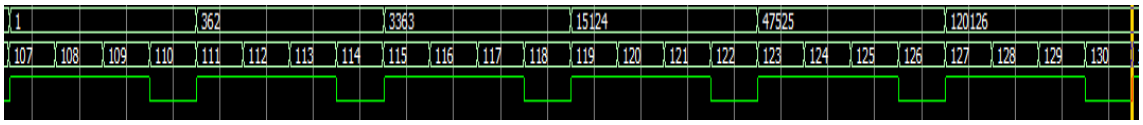


Figure 3.12: Test Bench of Fully Parallel Linear Overlay.

In conclusion, the fully parallel Linear overlay is suitable to reduce the II. However, this system is suffering challenge of higher area utilization as it requires a lot of memory storages to support the functionality of FIFO.

3.3 First Stage Parallel Linear Overlay

In the previous section, fully parallel Linear overlay is introduced for reducing the clock cycles of II, whereas this overlay comes with the limitation of high area requirements. For solving the high utilization of area, a new Linear overlay with first stage parallelism is proposed. The first stage parallel Linear overlay includes parallelism of two FUs in the first stage and the rest of stages which are in a cascade of FUs (as shown in Fig 3.13). This system reduces the area utilization by decreasing the parallelism FUs stage, hence, several memories is unnecessary in this new overlay, such as the FIFO between each stage in the fully parallel linear overlay. In addition, because the rest of stages are not parallelism, the logic utilization of implementing two ALU and RF blocks is no longer needed.

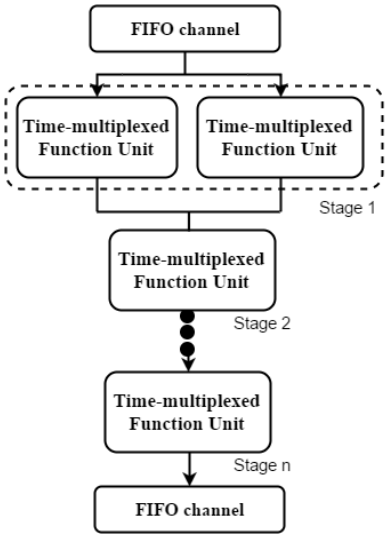


Figure 3.13: Linear Overlay of First Stage Prallel.

In the first stage parallel linear overlay, the input and output are still 64-bit. Moreover, the output data of each stage also is 64-it. Consequently, the processing of data transmission for parallelism stage and non-parallelism stages is critical. The overall main steps can be summarized to:

- Processing of the output data from parallelism stage should be adaptable for the next non-parallelism stage.
- Implementation of the non-parallelism stage with only one FU can execute parallel data, and parallel the result (32-bit) back to 64-bit at the end of the stage.

3.3.1 Challenge of First Stage Parallel Linear Overlay

In the first stage parallel linear overlay, only the first stage has two FUs. The internal structure of the first stage is approximately similar to that of fully parallel Linear overlay. The main difference from the original block is the implementation of output data. In the first stage block, input data keeps being 64-bit, then two FUs process data in parallel and generate output data in 64-bit. However, the rest of stages are not 2 parallel FUs, which means the rest of blocks cannot operate data at the equivalent

time, it leads to data loss. Because at one clock cycle, the input frame contains two data, one FU (non-parallelism) block cannot execute two data simultaneously, then one data loses on every clock cycle if no extra memory is used to store these data. Illustrate problem with below example in Fig 3.14

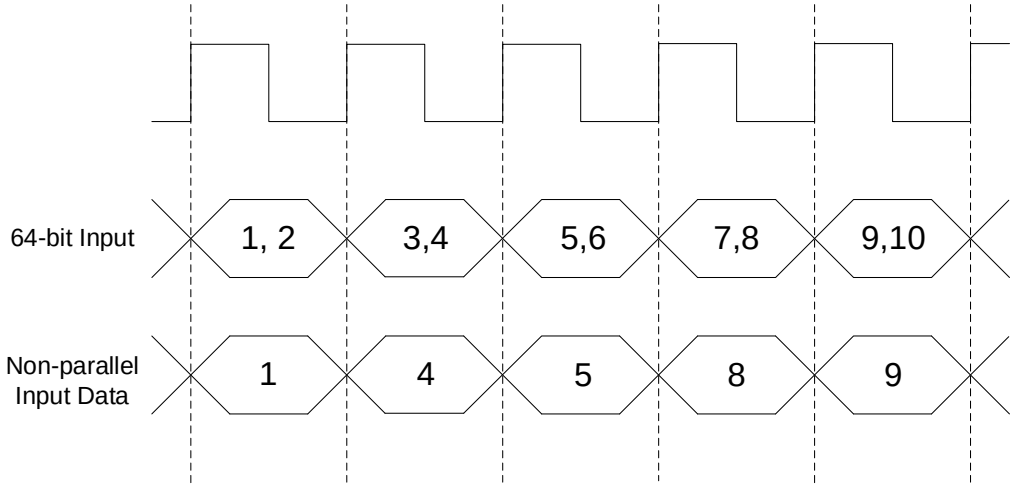


Figure 3.14: Challenge of data loss.

There are two options to solve the problem of input data loss:

1. Modifying non-parallel stage block: Using array stores the incoming parallel data in non-parallel stage blocks. Thus, the FU input data acquires from the array on the corresponding sequence. In this method, data is stored in memory to avoid losing, 32-bit FU can process the 64-bit input data. However, it also causes higher area utilization which is not an advisable option.
2. Modifying parallel stage block: Problem is about the 32-bit non-parallel FU cannot execute two data from the output of parallel stage block on one clock cycle. Besides, new data comes on every clock cycle, then data loss is unavoidable in this scenario. Changing problem for another direction, modifying the parallel stage block by doubling the clock cycle of output and output-valid signal at the end. Less memory is required in this method which is selected.

3.3.2 First Stage Block of First Stage Parallel Linear Overlay

As mentioned in the previous section, the way to avoid data loss is by doubling the clock cycles of output data and output valid signal in parallel 64-bit FU, the expectation of result is shown in Fig 3.15. Because a set of data lasts for two clock cycles, the output valid signal also is doubled to two clock cycles, which controls loading data into RF in the next stage. Then the next stage 32-bit FU can get completed input data from the parallel input.

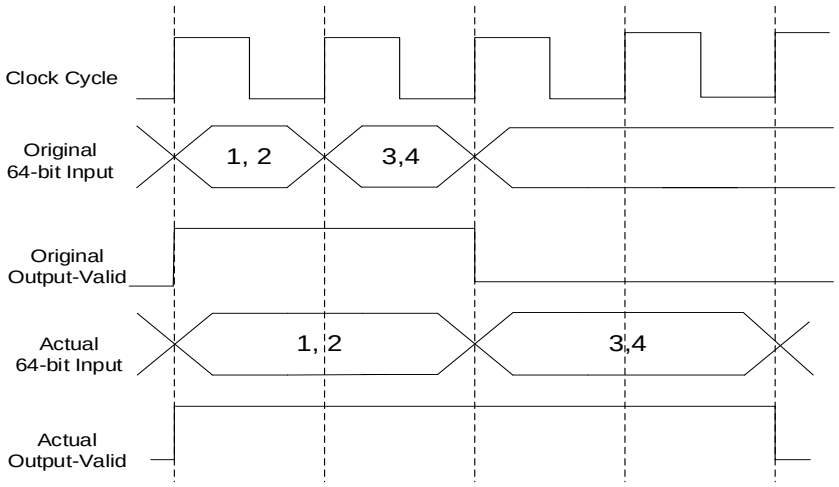


Figure 3.15: Expectation of Processing Data.

To complete this function, some critical points are overcome:

- How to double 'valid' signal — 'control_d7' signal is generated from IM block, which actually is the output valid signal. Hence, using counter to count up when 'control_d7' is high, counter counts down when 'control_d7'is low. If the counter is not zero then 'valid' signal is high, otherwise is low.
- How to double data — Once ALU blocks have results, combining two values into one 64-bit data and storing into an array. Following the above 'valid' signal, when it is high then the system starts sending data out. Using a flag to execute negation, only when flag signal is high, the next data in the array can be sent out.

- How to avoid delay — Inappropriate time delay will cause the following execution has the wrong result. Hence, another flag is used. As the system is triggered in rising edge of every clock cycle, once 'control_d7' is high, the ALU also has 64-bit output at the same time. Before the next rising edge coming to make 'valid' signal high, the flag is zero, data output is directly assigned the ALU result. When 'valid' signal is high on the rising edge of the clock, flag also is high, data output is got from the array.

For better understanding, the process is divided into three parts with flowcharts introduction.

In Fig 3.16 flowchart, it indicates how to double the valid signal, and generate some parameters for using in the other parts: When 'control_d7' is high, triggering counter (count) and number (num) count up, the number is a sequence of sending data into the array. Moreover, valid signal register (valid1) is high and delay-flag (flag1) also is high. when 'control_d7' is low, number (num) clears to stop sending data into the array. Since the system still requires high valid signal, then counter counts down, if the counter is not smaller than one, valid register (valid1) is high and the system keeps subtracting, otherwise, counter and valid register clear to zero.

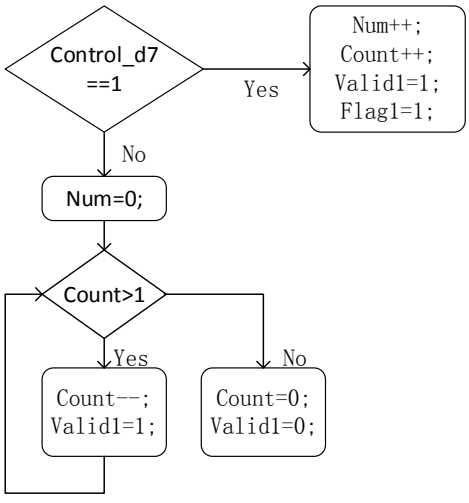


Figure 3.16: Flowchart of Valid signal

In Fig 3.17 flowchart, it shows how to implement clock cycle of the system without delay by using flag from the previous flag1: In this part, flag1 (generated in previous) is used to determine the output data. When flag1 is low, then data output (dout) is directly acquired from ALU output data (dout1), and valid signal (dout_v) is from 'control_d7'. When flag1 is high, the output data and valid signal are acquired from the register (valid1) and data array.

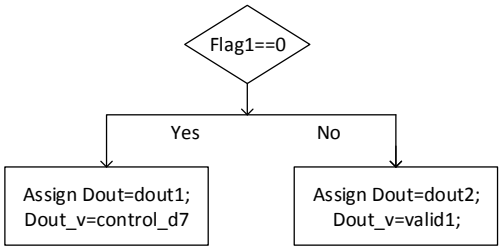


Figure 3.17: Flowchart of Time flag

In Fig 3.18 flowchart, it explains how to send data out on every two clock cycles: when the data valid signal (dout_v) is high, system permits to send data out. Thus, data counter (count_d) is used, when the counter is zero, 'count_d' starts counting up, sending data from the array to data register (Dout2) and setting 'flag' to 1. System reverses flag signal on every clock cycle since counter counts up in every clock cycle, then data is sent from the array to register (dout2) when the 'flag' is high.

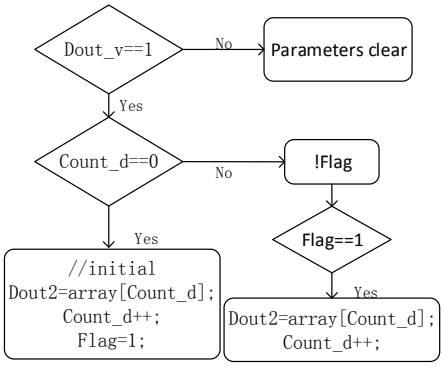


Figure 3.18: Flowchart of Doubling Data.

Code is shown in Fig 3.19

```
reg [63:0] data [0:19];
integer y,z=0,k=0,num=0,count=0,count_d=1;
always@(posedge clk)
begin
   //***valid signal***
   if(control_d7) begin
   num <=num+1;
   flag1 <= 1;
   count<=count+1;
   dout_v1 <=1;
   end
   else begin
   num <=0;
      if(count>1)begin
      count<=count-1;
      dout_v1 <=1;
      end
      else  begin
      count<=0;
      dout_v1 <=0;
      end
   end
   //***send data out***
   if(dout_v) begin
      if(count_d==0)begin
      dout_2<=data[count_d];
      count_d<=count_d+1;
      flag<=1;
      end
      else begin
         if(flag==1) begin
         dout_2<=data[count_d];
         count_d<=count_d+1;
         end
      flag<=~flag;
      end
   end
   else begin
   count_d <=0;
   dout_2<=0;
   flag<=0;
   flag1<=0;
   end
end
//***store data from ALU tp array***
always@(num or dout_1)
begin
   data[num] <= dout_1;
end
//***determine output data***
assign dout_1 = {p_o_2[31:0], p_o_1[31:0]};
assign dout = (flag1==0)?dout_1:dout_2;
assign dout_v=(flag1==0)?control_d7:dout_v1;
```

Figure 3.19: Doubling Clock Cycle of Data and Valid Signal

In conclusion, the rest of structures are same as the structure of fully parallel Linear overlay. The output data and the valid signal of parallel FU are doubled to two cycles.

3.3.3 Critical Steps of Non-Parallel FU Block

In first stage parallel Linear overlay, except the first stage, the rest of stages are 32-bit FUs based on the original FU of TMFU overlay to modify the FU structure. In the previous section, a modified parallel FU block can generate data on every doubled clock cycles. Hence, the 32-bit FU can simply acquire data from parallel FU output without data loss.

The critical points of modifying 32-bit FU are all concluded in:

1. Getting corresponding 32-bit instruction data from parallel 64-bit instruction.
2. Separating parallel data and sending in the correct sequence to RF.
3. Generating the output data in the required format.

3.3.4 Acquiring 32-bit Instruction from Parallel Instruction

In this overlay, the instruction is almost same as the fully parallel instruction, except address of data source is different since the rest of stage is 32-bit non-parallel FU. Following the section 3.2.1 mention, the address of data source should be added in sequence from the 2nd stage, since only one register can store the instruction (shown in Fig 3.8). This implies the 32-bit FU requires to get 32-bit instruction from parallel 64-bit instruction.

For a specific benchmark, instruction is written by the user according to the execution of benchmark. Therefore, how many lines of instruction is known. In this part, the number of instruction lines (INS_NO) is an important parameter for operation, which is used to create the arrays. Two 32-bit instructions combine into one instruction with a particular sequence, first executed instruction [65:33] is stored in 'ins_h' array and the other instruction [32:0] is stored in 'ins_l' array.

The flowchart in Fig 3.20 is the processing steps when instruction input has value, system starts ins-counter (counter) and storing the data into array until number of

counter is greater than that of instructions (INS_NO), then counter clears to zero. Meanwhile, when the counter has value, the system sends instruction into IM using a number (num), which decides the sending sequence. If 'num' is odd then data in 'ins_h' is sent to input of IM (inter_ins), data of 'ins_l' is sent when 'num'is even. Until 'num' is larger than 'INS_NO', 'inter_ins' clears to zero.

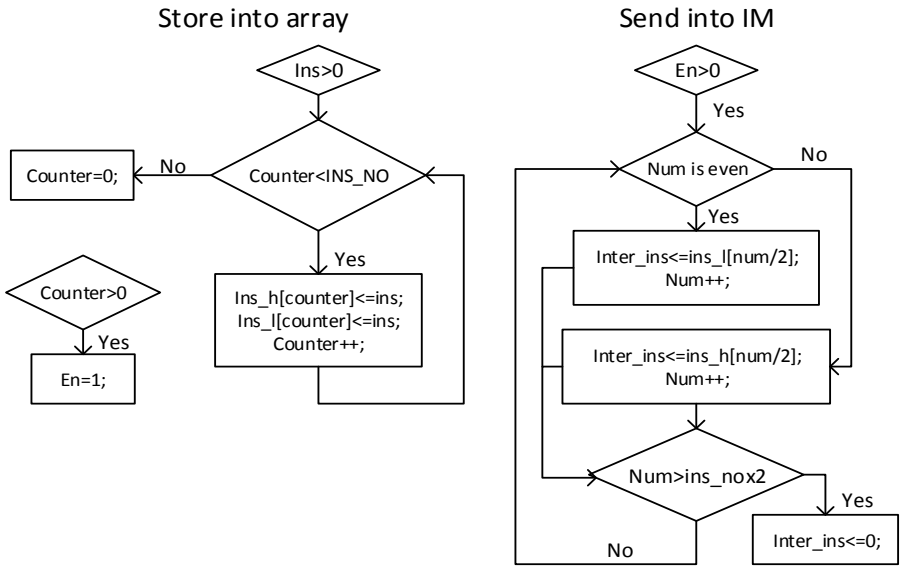


Figure 3.20: Flowchart of Instruction

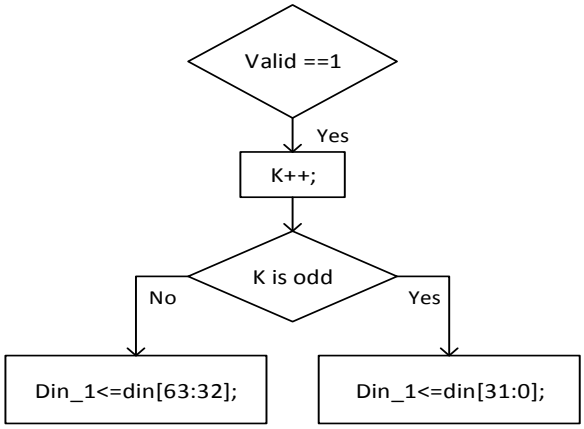
3.3.5 Acquiring Input and Combination of Output

The implementation of receiving data has the similar method to process data, which sends data into RF by checking the parity of the number. When the number is even then data [63:32] is sent into RF (din_1). Otherwise, data [31:0] is sent to 'din_1' when the number is odd. All the above processing requires the 'valid' signal is high. If 'valid' signal is low, the 'din_1' and number clear to zero.

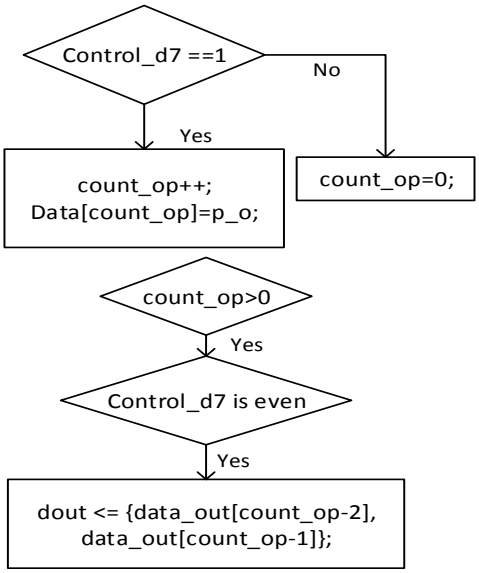
For the combination of output, the critical method is similar to acquire input data. When ALU block has the arithmetic result, calculation results are sent to the array

if valid output signal (control_d7) is high. Counter (count_op) adds one for checking parity. The first generated ALU result is in the higher bit of output data.

The flowcharts Fig 3.21 approximately display the execution process.



(a) Flowchart of Acquiring Data.



(b) Flowchart of Combination Output.

Figure 3.21: Flowchart of Design in 1st Stage Parallelism Overlay .

The Verilog code about the implementation of instruction section 3.3.4, input and output data are shown in Fig 3.22, Fig 3.23 and Fig 3.24 separately.

```
//**************separate and give instruction*****************
integer i,j=0,counter=0,num=0;
reg [34:0] ins_h [0:INS_NO-1];
reg [34:0] ins_l [0:INS_NO-1];
reg [34:0] inter_ins=0;
reg en_ins=0
always@(posedge clk)
begin
   if(ins) begin
      if(counter<INS_NO)    begin
      ins_h[counter] <= {ins[72:70],ins[65:45],ins[43:33]};
      ins_l[counter] <= {ins[72:70],ins[32:12],ins[10:0]};
      counter = counter+1;
      end
      else counter =0;
   end
   if(counter) en_ins=1;
   if(en_ins) begin
      j<=num/2;
      if(num&1) begin //even -- low data
      inter_ins <= ins_h[j];
      num<=num+1;
      end
      else begin  //odd -- high data
      inter_ins <= ins_l[j];
      num<=num+1;
      end
      if(num>(INS_NO<<1)) inter_ins <=0;
   end
end
```

Figure 3.22: Implementation of Instruction

```
//***************separate and give din ************************
integer k=0, z=0, num_d=1;
reg [31:0] din_1=0;
wire en_data;
always@(posedge clk)
begin
   if(valid) begin
   k<=k+1;
      if(k%2==0) din_1 <= din[63:32];
      else din_1<= din[31:0];
   end
   else begin
   din_1<=0;
   end
end
```

Figure 3.23: Implementation of Input Data

```
//--------------------output dout generate----------------------
reg dout_v_1,dout_v_2;
integer count_op=0;
reg [31:0] data_out [0:19];
always@(posedge clk)
begin
   if(control_d7) begin
   count_op <= count_op+1;
   data_out[count_op] <= p_o[31:0];
   end
   else count_op <= 0;
   if(count_op>0 && (count_op%2==0)) dout <= {data_out[count_op-2], data_out[
       count_op-1]};
   dout_v_1  <= control_d7;
   dout_v_2  <= dout_v_1;
   dout_v <= dout_v_2;
end
```

Figure 3.24: Implementation of Combination Output

3.3.6 Simulation of First Stage Parallel Linear Overlay

For testing the functionality of first stage parallel Linear Overlay, Verilog text fixture is used to simulate system execution of Chebyshev benchmark. The test bench is as Fig 3.11, the expected calculation result is shown in table 3.4. Furthermore, the II should be reduced to four clock cycles in Chebyshev benchmark. Running simulation in Modelsim, the simulation output is shown in Fig 3.25. According to the simulation result, the Chebyshev benchmark in this overlay has the correct production and the II also is four clock cycles. This means the first stage parallel Linear overlay is workable in benchmark with a small amount of input data. The scenario of benchmarks with large amounts of input data is compulsory to test.

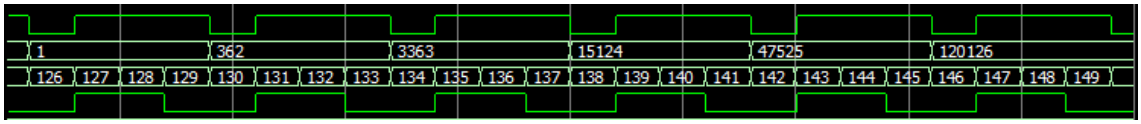


Figure 3.25: Simulation of Chebyshev Benchmark

mm benchmark is selected for testing (Fig 3.26), in the mm benchmark, there are 16 inputs and eight stages in architecture. Compared with Chebyshev benchmark, mm has 14 bypass instructions and 2 arithmetic instructions in the first stage. But Chebyshev only has one bypass and one arithmetic instructions. Hence, the mm

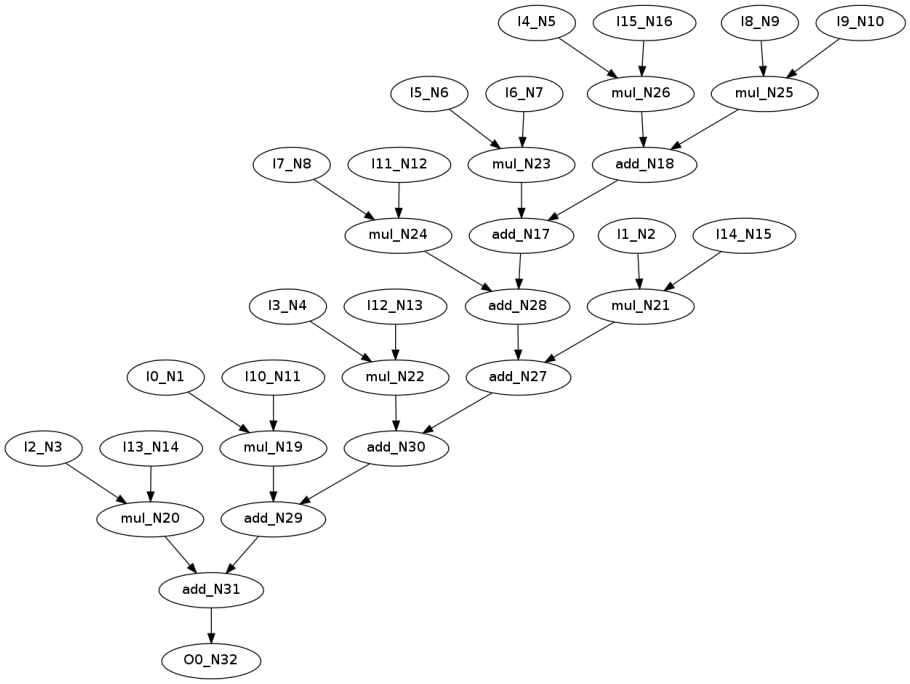


Figure 3.26: MM Benchmark

benchmark is suitable for testing lager input scenario. In simulation, mm has 16 inputs and the simulation input is a set of data 1, 4, 2, 3 and 1, 5, 2, 3 following the sequence. Test bench is shown in Fig 3.27.

```
1  //***first set of data***
2      user_w_write_32_data = 64'h00000001_00000004;
3  #20;user_w_write_32_data = 64'h00000002_00000003;
4  #20;user_w_write_32_data = 64'h00000001_00000004;
5  #20;user_w_write_32_data = 64'h00000002_00000003;
6  #20;user_w_write_32_data = 64'h00000001_00000004;
7  #20;user_w_write_32_data = 64'h00000002_00000003;
8  #20;user_w_write_32_data = 64'h00000001_00000004;
9  #20;user_w_write_32_data = 64'h00000002_00000003;
10 //***second set of data***
11 #20;user_w_write_32_data = 64'h00000001_00000005;
12 #20;user_w_write_32_data = 64'h00000002_00000003;
13 #20;user_w_write_32_data = 64'h00000001_00000005;
14 #20;user_w_write_32_data = 64'h00000002_00000003;
15 #20;user_w_write_32_data = 64'h00000001_00000005;
16 #20;user_w_write_32_data = 64'h00000002_00000003;
17 #20;user_w_write_32_data = 64'h00000001_00000005;
18 #20;user_w_write_32_data = 64'h00000002_00000003;
```

Figure 3.27: Snapshot of Test Bench for mm Benchmark

The expected results from the calculation are shown in table 3.6 and simulation result is shown in Fig 3.28. By comparison, the simulation result is correct, but the

clock cycle of II is quite higher than the expectation. In TMFU overlay, the II is 32 clock cycles for mm benchmark. However, II is 30 clock cycles in the first stage parallel linear overlay, it did not decrease as the expectation to almost half of original II.

Table 3.6: Arithmetic Calculation Result of MM

Input	1, 4, 2, 3	1, 5, 2, 3
	40	44

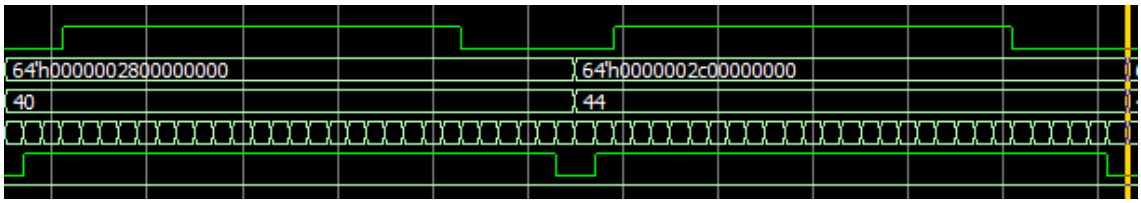


Figure 3.28: Simulation Result of MM Benchmark

After checking the processing steps in the simulation, the reason why II did not reduce as the expectation is that: When parallel data sends into the non-parallel stage (32-bit FU) from the parallel stage (64-bit FU), the parallel 64-bit data must expand to 32-bit format data. Hence, when executing the arithmetic operation in RF and ALU block, the system still requires more clock cycles to finish the execution. Thus, II only reduces few clock cycles when the number of input data is large in first stage parallel Linear overlay. But when the number of input data is less, this overlay is suitable for implementation. In conclusion, the first stage parallel Linear overlay has less reduction of II when the number of input data is large, which is a tradeoff between II and area utilization.

3.4 Summary

In this chapter, a proposed TMFU overlay architecture is introduced, which significantly reduces the FU and interconnect resource, but causes the cost of a higher II and a reduced throughput. Hence, a new Linear overlay is investigated and built

to decrease the clock cycles of II. The Linear overlay includes two types: 1. Fully parallel Linear overlay. 2. First stage parallel Linear overlay.

Through simulation and testing, we generalize the characteristics of this two overlay: 1) Fully parallel Linear overlay has higher area consumptions since it contains more memory and function blocks. In contrast, the first stage parallel Linear overlay has less area space utilization as only one stage has parallel FUs. 2) According to the performance of reduction II, fully parallel Linear overlay has a greater advantage than first stage parallel Linear overlay. When benchmark has a larger number of input data, fully parallel Linear overlay can reduce II as expectation, but first stage parallel Linear overlay cannot achieve same payoff with less reduction of II. This is a tradeoff, which should be considered when selecting overlay for various benchmarks. 3) Because both Linear overlays have parallel stages (2 FUs), when benchmark has odd instructions, Linear overlay might not execute properly. This should be tested and investigated in the future work. Generally, the Linear overlay is a good idea to reduce the II with a reasonable selection of diverse benchmarks and requirements.

Chapter 4

Floating Point Computation Block

In the previous chapters, we enunciated the architecture of TMFU overlay and the proposed Linear overlays with reducing initial interval. The Linear overlay is constructed to optimize the throughput of TMFU overlay and achieves better functionality. In this chapter, we present architectures of floating point computation blocks using FPGA for investigating how to accomplish flowing point execution in TMFU overlay architecture for the future design. Floating point computation blocks using FPGA has been proposed as a method to solve the problem based on an issue [2]. We summarized and recorded the critical information. The key contributions of this chapter are concluded:

- Basic processing of floating point execution in single precision IEEE 754-2008 format.
- Introduction of three types floating point computation blocks in FPGA: 1) Logic-only fixed configuration floating point operators. 2) Fixed configuration floating point operators with DSP blocks. 3) Iterative DSP-based floating point unit.
- Result of payback and tradeoff between each operator, and the future work of TMFU overlay in floating point execution.

4.1 Execution of IEEE 754 Binary 32

In IEEE 754-2008 binary32 standard, the floating point number can be expressed by three parts: sign bit, exponent, and mantissa. The bit field format of 32-bit floating point consists of 1-bit sign bit, 8-bit exponent (E) and 23-bit (F) mantissa in this standard. The expression is written as:

$$F_{E,F} = (-1)^{-s} * 2^{w_E - 127} * (1 + w_F)$$

When a decimal floating point number is given, it is necessary to convert the decimal number to the IEEE 754 format. The procedures of converting are listed.

- Convert the decimal floating point number to binary number. ($20.59375 \rightarrow 10100.10011$)
- Move the radix point to the place between the most significant bit and the second bit. The decimal part is mantissa. ($10100.10011 \rightarrow 1.010010011 \times 2^4, F = 010010011$)
- Get the exponent bit from e=E-127 and convert to binary format. ($E = 127 + 4 = 131 = 10000011$)
- When the number is negative, the sign bit is one. Otherwise, the sign bit is zero. After all steps, the decimal number is converted to IEEE 754 320-bit format number.

In the flowing floating point operation, the input number is required to be IEEE 754 format. In general, the addition and subtraction of floating point number have six phases to complete the operation, architecture of floating point operator is built following these steps:

1. Operand checking: Check whether the operand is zero or not.
2. Exponent Alignment: Compare the exponent of two operands, calculate the difference value ΔE, and the smaller exponent is shifted the ΔE bit so that the smaller exponent and the larger exponent have same exponent-bit.
3. Mantissa operation: After exponent alignment, then do the mantissa operation as general. Herein, the exponent number of two operands should be identical.

4. Result normalization: Check the results by the above step, if the result bit of exponent and mantissa is not regular (overflow or carrier bit), that system executes normalization by shifting.
5. Rounding: In this structure, the default rounding mode is round to nearest and tie to even.

In addition, there are five exception flags defined in the IEEE 754 standard, which are an invalid operation, division by zero, overflow, underflow and inexact. The floating point operator structure includes the above characteristics of addition and subtraction processing steps.

4.2 Logic-Only Fixed Configuration Floating Point Operators

This operator is LUT-only implementation, which consumes a significant amount of logic resources since no DSP block used. The previous section noticed, the arithmetic operation of the floating point requires shifting when executing mantissa alignment and normalization. Addition and subtraction of floating point are expensive to fulfill as shifting is difficult to implement. This logic-only operator has a limitation of costly design requirement.

In the fixed configuration floating point adder architecture for logic-only implementation, it consists of six pipeline stages which match the processing of addition for floating point numbers.

Compared with the steps in section 4.1, there are few slightly different parts. In Fig 4.1, the first pipeline not only has operand checking, but also has sign bit checking and calculation of exponent difference E. Calculation of exponent difference is in step 2 mentioned before, and it is calculated using 8-bit subtraction. Except these function blocks, this pipeline stage contains multiplexers for the selection of next pipeline stage. The second stage is alignment shift and fulfilled by barrel shifter with Guard, Round and Sticky bits. The third pipeline stage is executing arithmetic operation (addition or subtraction) if operation is subtraction, then the lesser operand is negative. The

operands are in two's complement format in execution. The fourth pipeline stage is normalization, in the previous execution processing mention, this step requires the function of shifting (barrier shifter) and leading zero counter (a series of multiplexer). The fifth pipeline stage is rounding and uses default IEEE rounding mode. The last pipeline stage is exception condition checking, which implements to check whether the five conditions occur in the exponent and mantissa part.

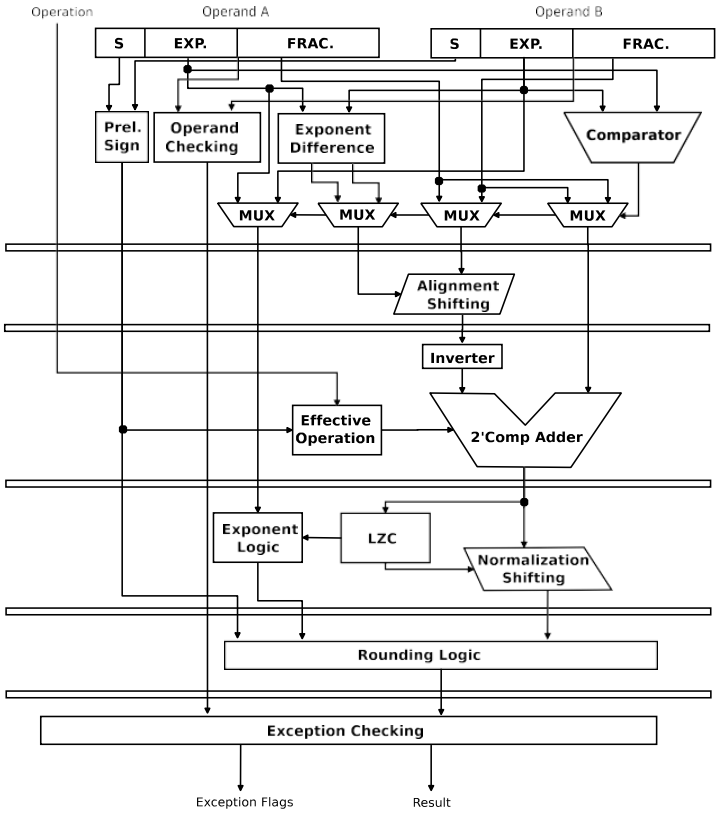


Figure 4.1: Adder in Logic-Only Fixed Configuration Operators [2]

The multiplication in logic-only operator has fewer pipeline stages since the alignment (second pipeline stage of addition) is not required, but multiplication is more complicated to execute. Thus, the logic utilization will increase compared with addition and subtraction.

4.3 Operators with DSP Blocks

In the last section 4.2, an LUT-only implementation operator is emphasized, which has higher resource requirements as no DSP block used. DSP block is flexible when executing floating point calculation and it can be configured at runtime and deign time, which makes the system more flexible and reduces the logic utilization of pipelines.

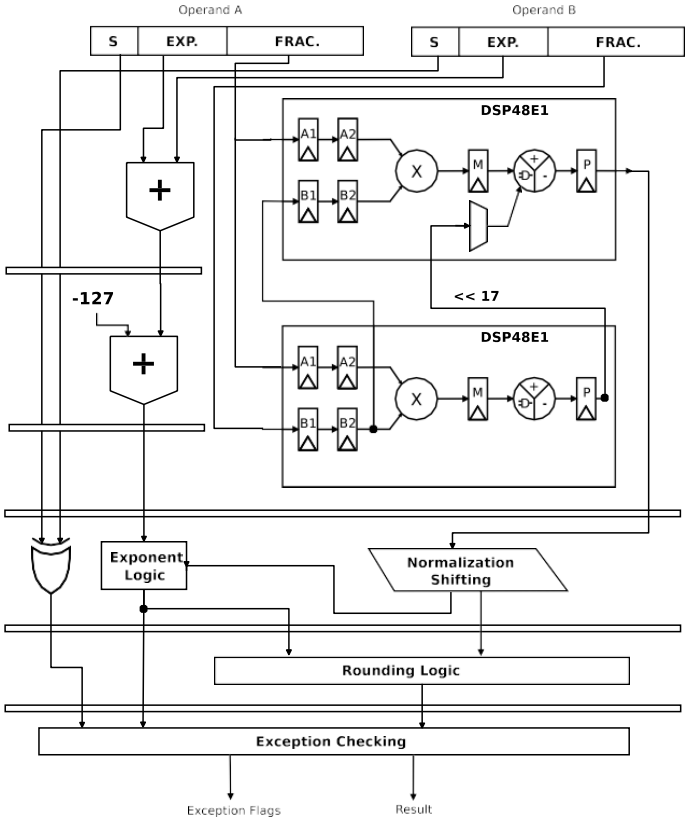


Figure 4.2: Multiplier in DSP Block Configuration Operators [2]

In this section, operators with DSP block is presented. For addition and subtraction in DSP block operators, the architecture is only changed the third stage in logic-only operators (Addition) to a DSP48E1 primitive through using Xilinx DSP48E1 primitive to do the addition and subtraction, the rest of pipelines keep the same with logic-only. However, this operator is not suitable for 25×25 bit multiplication since it can only operate 25×18 bit multiplication (one DSP48E1 inside), which is

not suitable for design requirement. Thus, a cascade of 2 DSP48E1 primitives solves the problem of not enough bit width in the multiplier. Diagram of the multiplier is shown in Fig 4.2.

Multiplier is achieved by cascading two DSP48E1 primitives, 0000101 and 1010101 are set to OPMODE in DSP1 and DSP2 blocks separately to distinguish which block is used, the execution result of DSP blocks operation can be drawn. From the Fig 4.2, the above DSP block is DSP2 and the result expression is $P_2 = (A[23:0] * B[23:18]) + (P_1 << 17)$. The below DSP block is DSP1 and expression is $P_1 = A[23:0] * B[17:0]$. The output of DSP2 is the calculation result of the multiplier.

Not only is the multiplier changed to DSP block, but also the function of exponent and the sign bit are changed. The exponent subtraction now is executed by adding two exponents of operands together and subtracting with 127. The sign bit is used XOR logic gate to accomplish.

4.4 Iterative DSP-based floating point unit

Iterative DSP-based floating point unit is a new proposed operator [2] with one DSP48E1 primitive inside, which is different from the last two mentioned operators. In iterative unit, all the arithmetic operations can be finished in one structure, but the previous two operators have to use different structures for supporting various arithmetic operation. Therefore, iterative floating unit has more flexible and saves more logic resources as only one DSP block in the design.

Fig 4.3 shows the block diagram of iterative DSP-based floating point unit, in contrast, this structure has some new components inside: the control unit and RAM32M memory. DSP block is flexible for floating point operators since it configures in run-time and configured time. The unit exerts these characteristics of DSP, thus unit can be programmed to choose which arithmetic operation should be through the control unit. The control unit is a state machine and saves the instructions, it contains the DSP48E1 inputs: ALUMODE, OPMODE, INMODE and a ready signal. RAM32M is a memory to store the input operands and sends the corresponding input operand to DSP block by getting selection signal from the control unit. The rest of pre-alignment

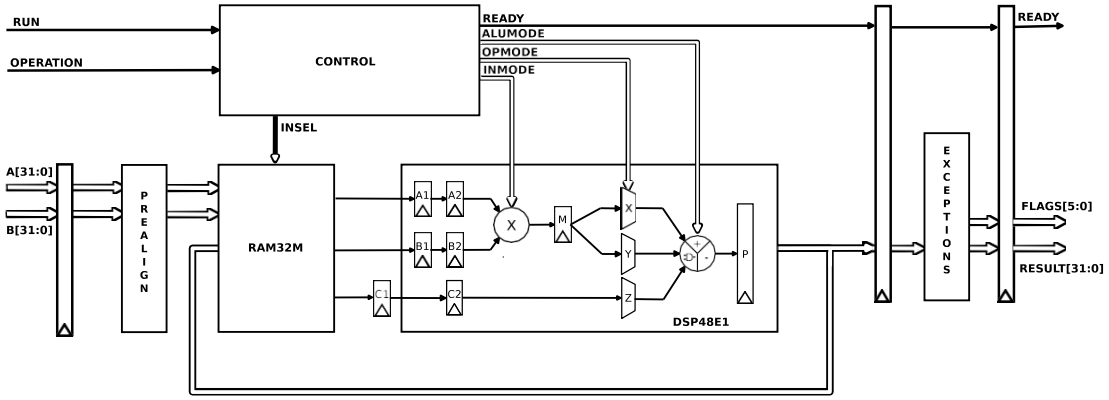


Figure 4.3: Iterative DSP-Based Floating Point Operator [2]

and exception checking are implemented in the logic and can be overlapped.

This floating point unit saves the logic resource using one DSP block, which does many sub-operation during the execution. Conversely, DSP block has a latency of 3 clock cycles, thus every operation will consume extra time for execution, which leads higher II and lower throughput. Generally, the II of a multiplier is 20, II of a adder is 23 cycles as each sub-operation takes the time to execute. It is a tradeoff for the designer to choose the operators.

In iterative floating point unit, the critical phases are still concluded in 6 parts. They are similar to the previous operator implementation in section 4.2 4.3. The first pipeline stage is pre-alignment, including operand checking, the sign bit and exponent different, which are same as logic-only operators. The time consumption of operating exponent difference in DSP block is 3 clock cycles. The second pipeline stage is alignment, shifting is required in this step, it is achieved in DSP48E1 block by processing multiplication. The third pipeline is execution, the adder is capable of processing with one DSP block, but multiplier requires two DSP blocks to cascade by setting the OPMODE, this is same as the section 4.3 mention of cascading result. Adder needs 3 clock cycles and multiplier requires over 6 clock cycles in this implementation. The fourth is normalization, it consumes 6 clock cycles to fulfill operation in DSP slice by using multiplication, the procedures are same as alignment. The

fifth stage rounding and sixth stage exception condition checking are similar to the previous work. They are executed in DSP primitive and logic respectively but the methods to operate are same as the previous standard.

4.5 Comparison of Operators and Future Work

In comparison with three types of operators: logic-only, DSP block and iterative unit. Among this operators, the logic-only operator requires the higher logic resources than the rest of operators. The DSP block operator exerts the DSP48E1 primitive to fulfill the arithmetic operations, hence, less logic resource is required. Comparing the iterative unit with DSP block operators, the adder of an iterative unit is smaller than that of DSP block operators, because shifting of an adder in the iterative unit also is accomplished in DSP48E1 slice but DSP block operator implements shifting in logic, this is costly. In summarized, the logic-only implementation has the highest logic utilization, in adder design, the iterative unit is smaller than DSP block operators but DSP block operator is smaller than iterative unit when executing multiplier.

Except considering utilization of the logic resource, the throughput and latency should also be considered. As iterative unit combines two arithmetic operations together into one structure by using the configurable function of DSP48E1 slice, it causes higher II than the other two operators. Therefore, the latency of iterative unit is higher, it leads to a lower throughput among the operators result. Nevertheless, it is undeniable that the iterative unit has the best flexibility with a tradeoff of latency.

For the TMFU overlay 3.1, it executes arithmetic operation in DSP48E1 primitive and is capable of getting the desired result when input operand is an integer. After testing the execution of floating point number, we found out that the TMFU overlay cannot support the floating point execution. Based on the previous investigation of floating point operators, the second operators 4.3 which is DSP block operators is suitable to be a reference when modifying the structure of TMFU overlay. In TMFU overlay, each FU contains one DSP48E1 primitive inside, the ALU block in FU can be reconfigured to the similar structure of DSP operators.

4.6 Summary

In this chapter, we presented the characteristics of three types operators in throughput and logic utilization aspects. To sum up, the operators with DSP block is suitable for modifying the TMFU overlay to support floating point execution. Each FU has ALU block which is made up of one DSP48E1 slice to do an arithmetic operation, we could modify this part of code according to the six pipeline stages structure of DSP block operators. The more detailed information is explained in [2].

Chapter 5

Experiments

5.1 Introduction

The experiments investigate and measure the performance of interface between the overlay accelerator and host processor via Xillybus. In this chapter, we present the TMFU overlay integrated with ARM processor through Xillybus interface, and analysis of performance result in comparison with commercial soft vector processors, Vectorbox MXP over a set of benchmarks. The main contributions of this chapter are emphasized as follows:

- An introduction of Xillybus interface.
- A systematic integration of TMFU overlay and ARM processor on Zynq platform via Xillybus.
- Benchmark evaluations between our proposed system with Vectorbox MXP.

5.2 Xillybus

Xillybus provides a DMA-based end-to-end connection interface between host processor and accelerator. Thus the FPGA and host can transfer data through Xillybus running in Linux or Windows operating system. The experiments are fully achieved in Linux environment. The host driver generates device files which is similar to pipes

between processors and capable of reading from and writing to. When executing the program on the host, the other side of the stream is a FIFO in the FPGA instead of another process. Moreover, the Xillybus stream is suitable for high data rate transmission, in this experiments, the operating frequency is 100MHz, and theoretical bound of writing and reading bandwidth is 100MS/s (Million Sample per second), as the Xillybus interface is round trip processing then the theoretical bound is 50MS/s. Xillybus is ready-made infrastructure in the ACP interface between programmable logic (PL) and processing system (PS) of Zynq platform and offers a straightforward and convenient option to acquire data.

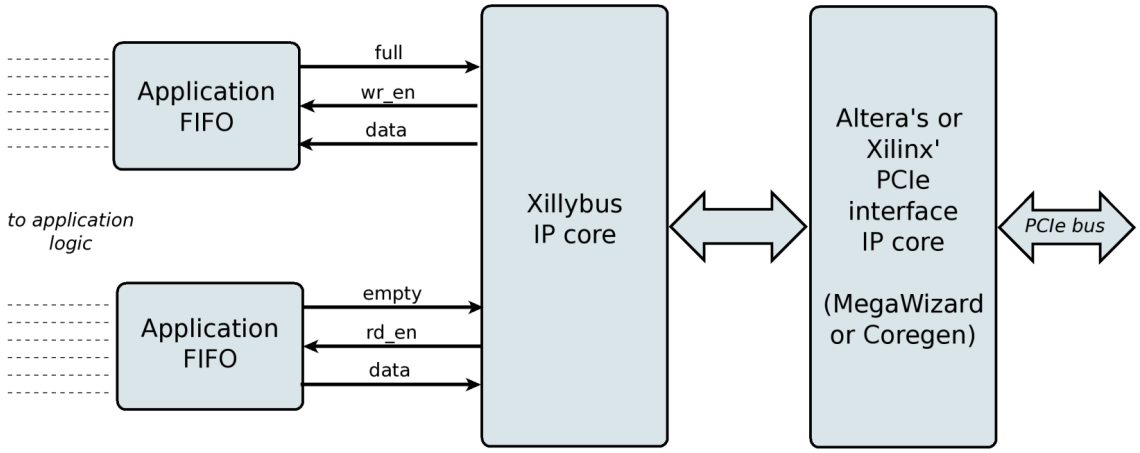


Figure 5.1: Simplified Block Diagram of Xillybus [3]

Fig 5.1 demonstrates the Xillybus simplified block diagram which is the connection of only one data stream in both directions. In the general implementation design, the system usually contains a few data streams which can transmit on each side. In summary, the system is consist of user logic and a host processor, while Xillybus acts as the connection between them.

The leftmost block in the picture is 'application FIFOs' which is a customizable standard FIFO and connected with Xillybus IP core to fulfill data communication between user logic and Xillybus IP core, this FIFO has excellent flexibility as the depth of FIFO, and the user logic can be decided and changed by the designer. The application FIFO is a connector of application logic (user logic) which is TMFU

overlay in the experiments and controlled by empty/full signals and write/read enable signals when transferring data. The user logic accepts data from the input FIFOs unless the input FIFOs us not empty. Otherwise, no data transmit into the user logic. After getting data from input FIFOs and executing the program in user logic, the processing result will be sent to output FIFOs when FIFOs is not full, and the Xillybus IP core keeps detecting whether data is available for transmission to the host. During the processing, Xillybus core is charged to check FIFOs empty and full signals, the ready controls initiation of data transmission.

The rightmost block in the picture is the host application interface, and ARM processor is host processor in the experiment. As mention above, the host driver generates writable and readable device file (pipes) between processes. The configuration of host is approximate that the data in the stream is detected by driver and loaded into the operating system of the host, additionally, the corresponding device file is generated. When Xillybus is implemented into design and modification is needed in the structure of accelerator, as the simplicity of implementing Xillybus, the modification of accelerator can be made by generating and uploading the new specific bit stream file to the location of root file system accordingly and rebooting. Xillybus is convenient and mature technique, the Xillybus team provides an IP Core Factory to benefit the FPGA designer and make Xillybus has diversification. The characteristics and parameters of Xillybus like bandwidth of the system, data width and the number of pipes, which can be customized following the user requirement. The Xillybus dose offer a good stage for FPGA designer.

5.2.1 Zynq FPGA Demo Bundle

The Zynq platform is a commercial computing platform which is comprised of programmable logic (PL) and processing system (PS). The PS includes a dual-core ARM processor with peripherals, buses and memory interfaces, and it is coupled together with PL which is formed by customized hardware. The data communication interface of PL and PS is achieved by AXI channels which enable high throughput data transmission. In this experiment, a Zedboard which contains the Xilinx Zynq-7000

All Programmable SoC is used as the platform. The Zynq FPGA demo bundle is the foundation structure of the experiment, which is made up of Integrated Software Environment (ISE), Xilinx Platform Studio (XPS) project, boot.bin and device tree files, through these file and projects, the netlist and bitstream can be generated. Additionally, the FPGA demo bundle has customized device files such as read ports, write ports, and address/data interfaces when it is created in the IP Core Factory. According to the user guide, the custom application logic can be integrated with Xillybus IP core through the FIFOs in a stream manner.

At the beginning of processing, system firstly generates a netlist in XPS project file, then following the given FIFO IP cores in ISE, the system reconfigures all the netlists which meet the demand of users. In top level HDL file, the bitstream is created and downloaded into the FPGA. In this infrastructure, the FIFOs are connected to Xillybus IP core via loopback connection and user logic is attached with the FIFOs to achieve application logic (shown in Fig 5.2). Xillybus infrastructure implements in the ACP interface of Zedboard and includes host program (on the processor), synthesized function (on reconfigurable fabric) and wrapper function, designers can interact and send data to the system through synthesized function by API interface.

5.3 Integration with Custom Logic

5.3.1 Round Trip Loopback Evaluation

Before integrating the TMFU overlay with ARM processor, we first use the demo bundle with default settings to evaluate the performance of Xillybus. The initial Demo bundle contains FIFOs connected in a loopback fashion thereby causing the Xillybus core to act both as a source and sink. As shown in Figure 5.3, the input (user_w_write) and output (user_r_read) pipes connect to the same FIFO to send and receive data in a loopback manner. Since there is no application logic in between the pipes, the round trip time can be treated as the PS-PL communication overhead.

Figure 5.4 demonstrates a simplified block diagram of Xillybus demo bundle. Xillybus.v build up the interface between Xillybus IP core and ARM processor core

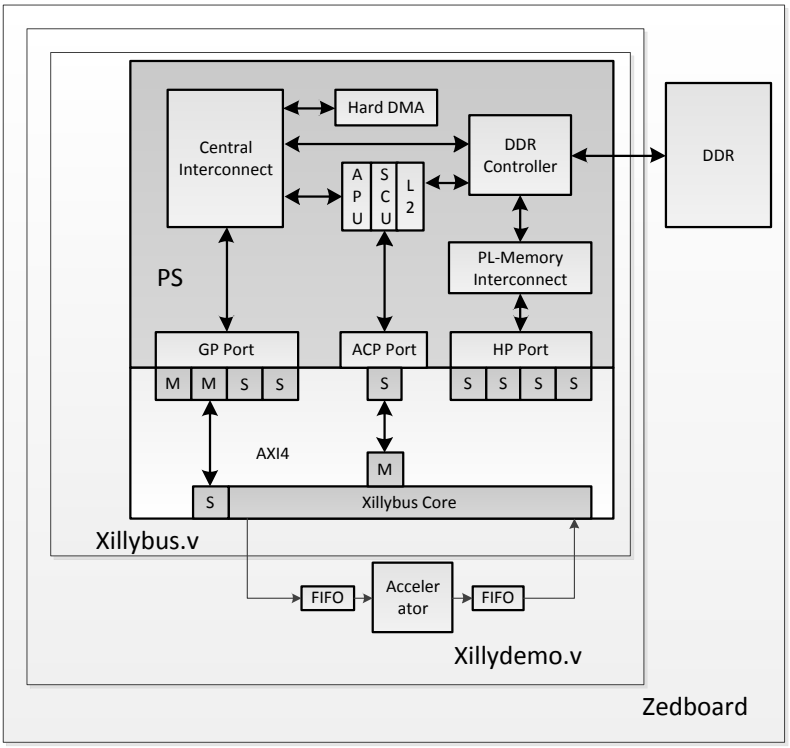


Figure 5.2: Xillybus Demo Bundle [3]

```
// 32-bit loopback
fifo_32x512 fifo_32
(
.clk(bus_clk),
.srst(!user_w_write_32_open && !user_r_read_32_open),
.din(user_w_write_32_data),
.wr_en(user_w_write_32_wren),
.rd_en(user_r_read_32_rden),
.dout(user_r_read_32_data),
.full(user_w_write_32_full),
.empty(user_r_read_32_empty)
)

assign user_r_read_32_eof = 0;
```

Figure 5.3: Xillybus 32-bit Loopback FIFO Connection

via AXI interconnect. And the top level Xillydemo.v wrapper file present in the Xillybus bundle is used to integrate our application logic. In this specific case, there is no application logic involved, data from Xillybus IP core will just loopback through a FIFO. After generating the bitstream for Xillybus interfaces, we download it to the Zedboard, along with boot file and device tree to set up a mini Linux system. An

example of host program is shown in Figure 5.5 to do the round trip time measurement, and Table 5.1 demonstrates the round trip time and throughput for different No. of samples.

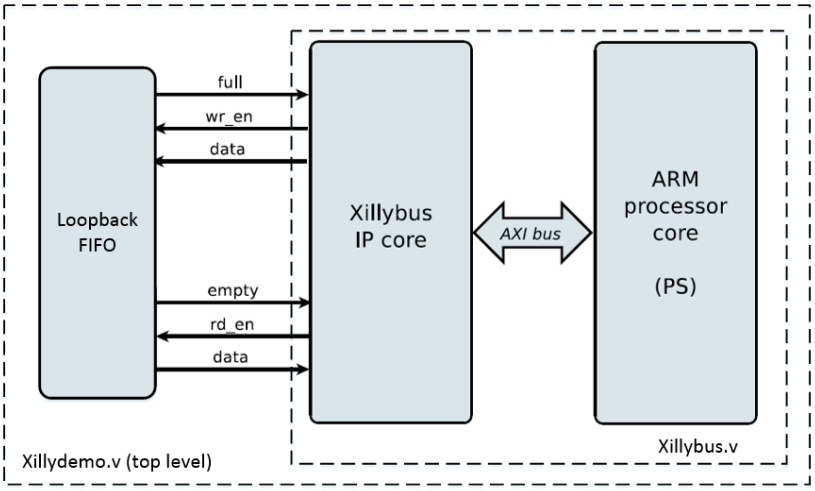


Figure 5.4: Xillybus Demo Bundle Block Diagram

Table 5.1: Single Pipe Loopback Results

No. of Samples	Round Trip Time (us)	Throughput (KS/s)
256	129	1984.5
512	129	3969.0
1K	129	7938.0
2K	147	13932.0
4K	166	24674.7
8K	258	31751.9
16K	406	40354.7
32K	756	43343.9
64K	1567	41822.6
128K	2820	46479.4
256K	5714	45877.5
512K	11336	46249.8
1M	22727	46137.9

From the results listed in Table 5.1, it is clear that for a small amount of data streaming, the Xillybus performance is reduced which means that communication overhead is high. But for large amount of data, its performance is close to theoretical

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <pthread.h>
#include <stdint.h>
#include <sys/time.h>
int fdr32 = 0;
int fdw32 = 0;
int N = 0;
int i;
int *array_input;
int *array_hardware;
struct timeval tstart, tend;
ssize_t t1,t2, temp;
int main(int argc, char *argv[]) {
fdr32 = open("/dev/xillybus_read_32", O_RDONLY);
fdw32 = open("/dev/xillybus_write_32", O_WRONLY);
N = atoi(argv[1]);
if (fdr32 < 0 || fdw32 < 0) {
perror("Failed to open devfiles");
exit(1);
}
//allocate memory
array_input = (int*) malloc(N*sizeof(int));
array_hardware = (int*) malloc(N*sizeof(int));
// generate inputs and prepare outputs
for(i=0; i<N; i++){
array_input[i] = i;
array_hardware[i] = 0;
}
//Measure the excution time
gettimeofday(&tstart, NULL);
t1=write(fdw32, array_input, sizeof(int)*N);
temp = write(fdw32, NULL, 0);
t2= read(fdr32,array_hardware, sizeof(int)*N);
gettimeofday(&tend, NULL);
printf("Execution time is %f us\n\r", (double)1000000*(tend.tv_sec-tstart.tv_sec)+(tend.
    tv_usec-tstart.tv_usec));
return 0;
}
```

Figure 5.5: Code Example: Execution Time Measurement

limit (50 MS/s).

5.3.2 Interfacing the TMFU Overlay

Because of the linear topology with FIFO channels, the TMFU overlay can be easily connected with Xillybus IP core, as shown in Figure 5.6. Since the system contains

only one input pipe and one output pipe, we can use the same program to measure the execution time as in Figure 5.5.

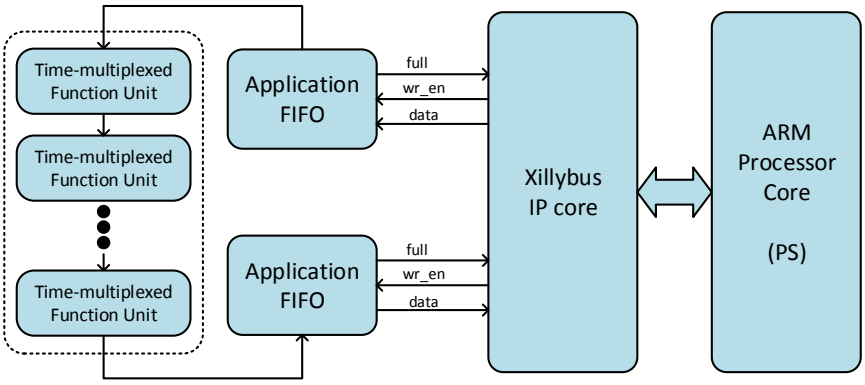


Figure 5.6: TMFU Overlay Integration via Xillybus

Table 5.2 lists the round trip time and throughput of the integrated TMFU overlay over benchmark *chebyshev*. According to the results, this system shows similar performance compared with the single pipe loopback when the No. of samples is less than 512. However, when the No. of samples grows gradually, the round trip time of integrated TMFU overlay is about $3-4\times$ more than that of single pipe loopback, which is due to the reason that, for benchmark *chebyshev* the II equals 6.

Table 5.2: Integrated TMFU Overlay Results

No. of Samples	Round Trip Time (us)	Throughput (KS/s)
256	129	1984.5
512	184	2782.6
1K	276	3710.1
2K	479	4275.6
4K	886	4623.0
8K	1696	4830.2
16K	3060	5354.2
32K	5751	5697.8
64K	11004	5955.7
128K	19372	6766.1
256K	28274	9271.6
512K	40606	12911.6
1M	62485	16781.2

5.4 Benchmark Evaluation with MXP

The VectorBlox MXP was developed as a commercial IP core which can be connected with Xilinx FPGAs via AXI interfaces [30]. It is able to explore the tradeoff between performance and area, with a hybrid approach which shares the benefits of traditional vector processing and modern SIMD mode. The MXP consists of a Nios II/f scalar soft processor, along with some vector lanes executing custom instructions on a local vector memory. Significant performance is achieved by effectively unrolling loops into vector operations. Instead of traditional vector load/store instructions, the MXP adopted direct memory access (DMA) read/write commands to achieve better storage efficiency and less memory latency. To have a fair comparison with the Xillybus-interfaced system, we perform the same benchmark *chebyshev* using MXP. Figure 5.7 shows an example of vector processing functions and its inherent timer to measure the whole operation time.

```
1   vbx_timestamp_start();
2   start = vbx_timestamp();
3   for(i = 0; i < div_factor; i++)
4   {
5   vbx_dma_to_vector(vb_input, (input + i*no_of_samples), no_of_bytes);
6   vbx(SVWS, VMUL,vb_result,16,vb_input);
7   vbx(VVWS, VMUL,vb_result,vb_result,vb_input);
8   vbx(SVWS, VADD,vb_result,-20,vb_result);
9   vbx(VVWS, VMUL,vb_result,vb_result+i,vb_input);
10  vbx(VVWS, VMUL,vb_result,vb_result+i,vb_input);
11  vbx(SVWS, VADD,vb_result,5,vb_result);
12  vbx(VVWS, VMUL,vb_result,vb_result,vb_input);
13  vbx_dma_to_host((result + i*no_of_samples) , vb_result, no_of_bytes);
14  vbx_sync();
15  }
16  end = vbx_timestamp();
17  vbx_print_scalar_time(start, end);
```

Figure 5.7: A Snapshot of MXP Program

We list the execution time and throughput of MXP over benchmark *chebyshev* in Table 5.3 and draw the curves to have a better observation for each system in Figure 5.8 and Figure 5.9.

From the above results, the MXP vector processor shows the best performance because of its SIMD mode data processing. The theoretical throughput can be 100 MS/s. However, due to the communication overhead, it reaches around 70 MS/s.

Table 5.3: MXP Results

No. of Samples	Execution Time (us)	Throughput (KS/s)
256	92	2782.6
512	92	5565.2
1K	92	11130.4
2K	101	20277.2
4K	110	37236.4
8K	138	59362.3
16K	258	63503.9
32K	498	65799.2
64K	977	67078.8
128K	1935	67737.5
256K	3834	68373.5
512K	7686	68213.4
1M	15340	68355.7

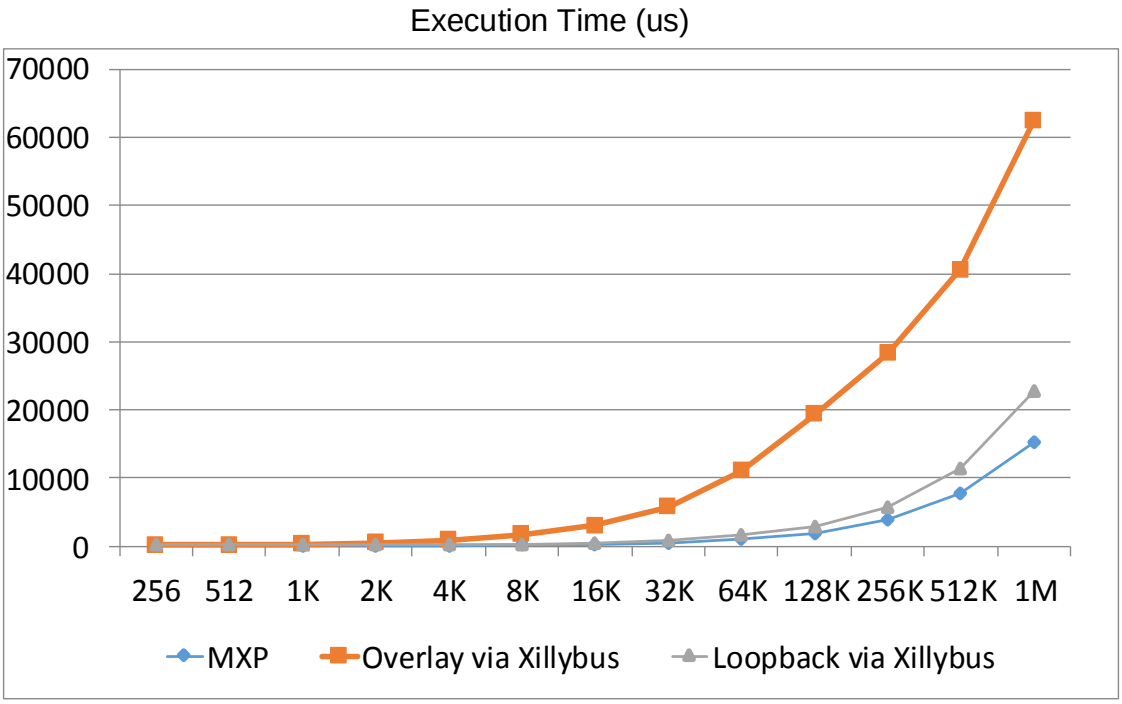


Figure 5.8: Execution Time of Integrated TMFU Overlay

Though the performance of our integrated system is less than 20 MS/s at the current time, it is capable of reaching the theoretical throughput of Xillybus round trip (50 MS/s) by replicating multiple pipelines to make II equals one. Another potential

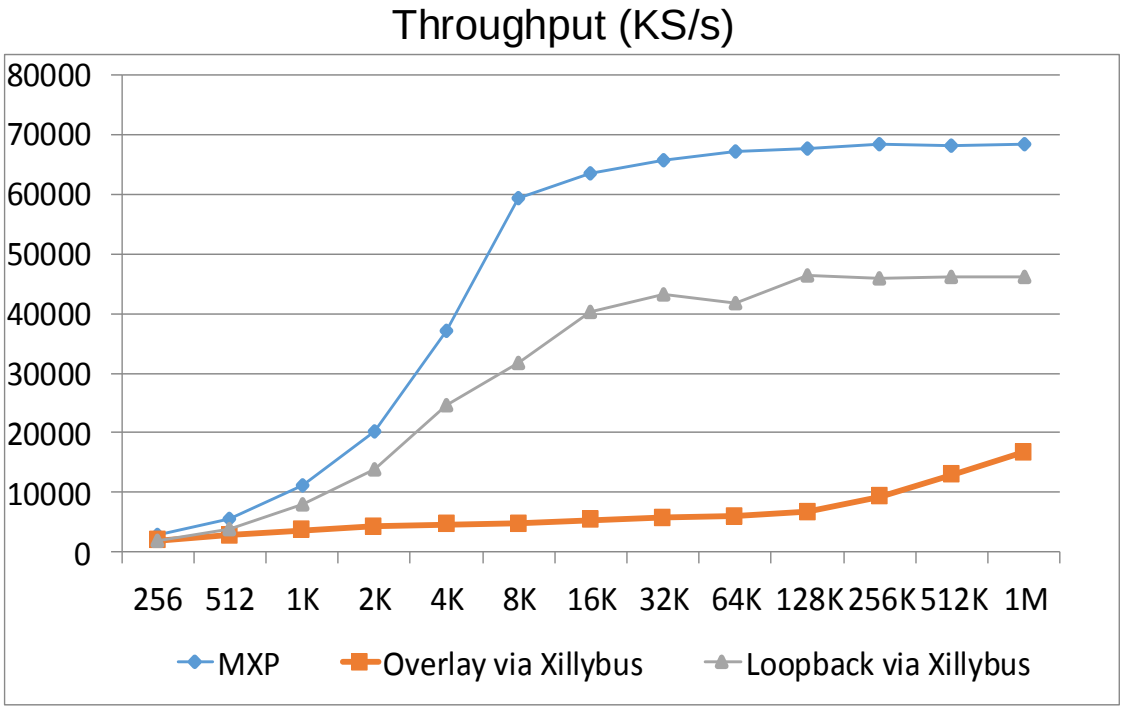


Figure 5.9: Throughput of Integrated TMFU Overlay

for us to improve the throughput significantly is that the theoretical throughput of Xillybus itself can be doubled if it makes use of the whole 64-bit AXI interconnect, instead of merely 32-bit interconnect.

Chapter 6

Summary and Future Work

6.1 Summary

In this report, we investigated the architecture of TMFU overlay and presented an analysis of Linear overlay architecture which is a new proposed overlay to decreasing the II. The architecture of Linear overlay is based on the proposed TMFU overlay. It can be classified into two options: fully parallel and first stage parallel. This two types of Linear overlay consist of parallel FUs on internal stages to reduce II. Additionally, the experiments of Xillybus infrastructure are presented which is using Xillybus to transfer data between the accelerator and ARM processor on Zynq platform. Finally, we evaluated the performance of our proposed system in comparison with Vectorbox MXP over several benchmarks.

In chapter 3, we demonstrated the execution principle of TMFU and new proposed Linear overly in two types for decreasing II. The fully parallel Linear overlay nearly reduces half of II but comes with the higher area consumption. The first stage parallel Linear overlay achieves the similar result when the number of data input is small. When the number of data input is high, it cannot reduce II as expectation. However, the area resource requirement of first stage parallel overlay is lower. Tradeoff between area and throughput is inevitable when processing various benchmarks. In chapter 4, we indicated three types structures of floating point operators on FPGA. Comparing the internal structure with elements of logic resource and latency, the operator with

DSP blocks is the most suitable for the TMFU overlay to support floating point execution among these designs. In chapter 5, we presented the performance outcomes of TMFU overlay integrated with ARM processor via Xillybus on various benchmarks and demonstrated the analysis of performance result between ARM processor and Vectorbox MXP.

6.2 Future work

As the above summary of the overall contributions for this dissertation, the proposed Linear overlay still has several limitations, which should be noticed in the future work. Considering portability and compatibility of Linear overlay, we present two potential directions for the future work:

- **Compatibility of Linear Overlay**: As the structure of Linear overlay is achieved by a parallelism of 2 FU on the internal stages, it causes a restriction on the number of instructions. When instruction number is even, the system operates in normal. Nonetheless, when instruction number is odd, the parallel FU stage might have faulty operation since one FU does not have instruction to execute. Thus, the corresponding measure in this scenario should be investigated in the future work.
- **Floating Point Execution**: The current TMFU overlay is capable of executing the integer number as the input operand. However, it cannot process floating point operands. In chapter 4, we mentioned the three types of floating point execution on FPGA and investigated internal architecture by comparing the latency and area resource requirements. The structure of operators with DSP block should be referenced in the future work to make TMFU support floating point operations.

Bibliography

[1] Kiyoung Choi. Coarse-grained reconfigurable array: Architecture and application mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46, 2011.

[2] Fredrik Brosser, Hui Yan Cheah, and Suhaib A Fahmy. Iterative floating point computation using fpga dsp blocks. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6. IEEE, 2013.

[3] Xillybus Ltd. Xillybus: IP Core Product Brief. `http://xillybus.com/downloads/xillybus_product_brief.pdf`.

[4] Neil W Bergmann, Sunil K Shukla, and Jürgen Becker. Quku: A dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):63, 2013.

[5] Greg Stitt and James Coole. Intermediate fabrics: Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, 2011.

[6] Davor Capalija and Tarek S Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[7] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[8] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. Efficient overlay architecture based on dsp blocks. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 25–28. IEEE, 2015.

[9] Xiangwei Li, Abhishek Jain, Douglas Maskell, and Suhaib A Fahmy. An area-efficient fpga overlay using dsp block based time-multiplexed functional units. *arXiv preprint arXiv:1606.06460*, 2016.

[10] Farhad Mehdipour, Hamid Noori, Morteza Saheb Zamani, Kazuaki Murakami, Mehdi Sedighi, and Koji Inoue. An integrated temporal partitioning and mapping framework for handling custom instructions on a reconfigurable functional unit. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 219–230. Springer, 2006.

[11] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, 49(5):465–481, 2000.

[12] Abhishek Kumar Jain, Xiangwei Li, Pranjul Singhai, Douglas L Maskell, and Suhaib A Fahmy. Deco: A dsp block based fpga accelerator overlay with low overhead interconnect. 2016.

[13] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Are coarse-grained overlays ready for general purpose application acceleration on fpgas? In *Proceedings of IEEE International Conference on Pervasive Intelligence and Computing*. IEEE, 2016.

[14] Charles Eric LaForest and John Gregory Steffan. Octavo: an fpga-centric processor family. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 219–228. ACM, 2012.

[15] Robert Dimond, Oskar Mencer, and Wayne Luk. Custard-a customisable threaded fpga soft processor and tools. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 1–6. IEEE, 2005.

[16] Peter Yiannacouras, Jonathan Rose, and J Gregory Steffan. The microarchitecture of fpga-based soft processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 202–212. ACM, 2005.

[17] Hui Yan Cheah, Suhaib A Fahmy, and Douglas L Maskell. idea: A dsp block based fpga soft processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 151–158. IEEE, 2012.

[18] Jan Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. *arXiv preprint arXiv:1606.01037*, 2016.

[19] R Dimond, O Mencer, and W Luk. Application-specific customisation of multithreaded soft processors. *IEE Proceedings-Computers and Digital Techniques*, 153(3):173–180, 2006.

[20] Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. Application-specific customization of soft processor microarchitecture. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 201–210. ACM, 2006.

[21] Charles Eric LaForest and J Gregory Steffan. Maximizing speed and density of tiled fpga overlays via partitioning. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 238–245. IEEE, 2013.

[22] Charles Eric LaForest, Jason Anderson, and J Gregory Steffan. Approaching overhead-free execution on fpga soft-processors. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 99–106. IEEE, 2014.

[23] Hui Yan Cheah, Fredrik Brosser, Suhaib A Fahmy, and Douglas L Maskell. The idea dsp block-based soft processor for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):19, 2014.

[24] Cheah Hui Yan, Suhaib Fahmy, and Nachiket Kapre. On data forwarding in deeply pipelined soft processors. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 181–189. ACM, 2015.

[25] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. *ACM SIGARCH Computer Architecture News*, 31(2):399–409, 2003.

[26] Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70. ACM, 2008.

[27] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core cpu accelerator. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232. ACM, 2008.

[28] Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. Vegas: soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 15–24. ACM, 2011.

[29] Aaron Severance and Guy Lemieux. Venice: A compact vector processor for fpga applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 261–268. IEEE, 2012.

[30] Aaron Severance and Guy GF Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[31] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the 35th*

annual ACM/IEEE international symposium on Microarchitecture, pages 283–293. IEEE Computer Society Press, 2002.

[32] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):12, 2009.

[33] Alexander Dunlop Brant. *Coarse and fine grain programmable overlay architectures for FPGAs*. PhD thesis, University of British Columbia, 2013.

[34] Alexander Sonek. Header parsing logic in network switches using fine and coarse-grained dynamic reconfiguration strategies. 2014.

[35] Rafat Rashid, J Gregory Steffan, and Vaughn Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 20–27. IEEE, 2014.

[36] Rafat Rashid. *A Dual-Engine Fetch/Compute Overlay Processor for FPGAs*. PhD thesis, University of Toronto, 2015.

[37] Cheng Liu, Colin Lin Yu, and Hayden Kwok-Hay So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme fpga frequency. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 228–228. IEEE, 2013.