IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 5, 2020, accepted September 13, 2020, date of publication September 18, 2020, date of current version October 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3024851

Design and Implementation of a 256-Bit RISC-V-Based Dynamically Scheduled Very Long Instruction Word on FPGA

NGUYEN MY QUI, CHANG HONG LIN, (Member, IEEE), AND POKI CHEN, (Member, IEEE)
Department of Electronics and Computer Engineering, National Taiwan University of Science and Technology, Taipei City 106, Taiwan

Corresponding author: Nguyen My Qui (m10702809@mail.ntust.edu.tw)

This work was supported by the Ministry of Science and Technology (MOST), Taiwan, under Grant 108-2221-E-011-137.

ABSTRACT This study describes the design and implementation of a 256-bit very long instruction word (VLIW) microprocessor based on the new RISC-V instruction set architecture (ISA). Base integer RV32I and extension instruction sets, including RV32M, RV32F, and RV32D, are selected to implement our VLIW hardware. The proposed architecture packs up eight 32-bit instruction flows, each of which performs fixed operational functions to create a 256-bit long instruction format. However, one obstacle of studying new ISAs, similar to RISC-V, to design VLIW microprocessors is the lack of dedicated compilers. Developing an architecture-specific compiler is really challenging. An instruction scheduler is integrated to dynamically schedule independent instructions into the VLIW instruction format. This scheduler is used to overcome the lack of a dedicated RISC-V VLIW compiler and leverage the available RISC-V GNU toolchain. Unlike conventional VLIWs, our proposed architecture is organized into six main stages, namely, fetch, instruction scheduler, decode, execute, data memory, and writeback. The complete design is verified, synthesized, and implemented on a Xilinx Virtex-6 (xc6vlx240t-1-ff1156). Maximum synthesis frequency reaches 83.739 MHz. The proposed RISC-V-based VLIW architecture obtains an average instructions per cycle value that outperforms that of existing open-source RISC-V cores.

INDEX TERMS Very long instruction word (VLIW), RISC-V, microprocessor, dynamic scheduling, field-programmable gate arrays (FPGA).

I. INTRODUCTION

Exploiting instruction-level parallelism (ILP) is key to achieving high performance for microprocessors. The implementation of processor architectures to exploit high ILP ranges from pipelining, multiple processors, multithreading, superscalar, and very long instruction word (VLIW) [1]. Superscalar and VLIW processors exploit spatial parallelism by utilizing multiple functional units to issue several operations simultaneously. However, the superscalar architecture demands specific hardware control to schedule instructions, thereby making superscalar hardware highly complicated. In the VLIW, parallelism potential among instructions is determined through the support of a powerful VLIW compiler. Concurrent operations are packed into very long instructions without any dependency. This compiler-based scheduling reduces VLIW hardware complexity compared with superscalar hardware.

This study proposes a general-purpose 256-bit VLIW architecture based on the new RISC-V instruction set architecture (ISA) [2]. Formal research on the design and implementation of VLIW microprocessors based on the RISC-V ISA is rarely reported. The selected instruction sets include base integer RV32I and optional extensions, including multiply-and-divide integer RV32M, single- and double-precision floating-point RV32F, and RV32D. Our proposed VLIW implementation includes eight 32-bit operational flows, creating a 256-bit VLIW instruction format. Nevertheless, the difficulty of studying and constructing VLIW architectures based on the latest ISAs, such as RISC-V, lies in the lack of a specific compiler to schedule the sequential instructions of an original program into the VLIW instruction format. Moreover, programs scheduled for a given VLIW implementation are not binary compatible with other implementations with a different number of functional units

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

or functional units with different latencies [3]. To solve this lack of a dedicated RISC-V VLIW compiler, we integrate an instruction scheduler to dynamically fit independent instructions in the instruction memory into their corresponding flow in our VLIW instruction format. Thus, leveraging the available RISC-V GNU toolchain [4] is possible. Consequently, the VLIW architecture is organized into six stages, namely, fetch, instruction scheduler, decode, execute, data memory, and writeback. The main contributions of our paper can be summarized as two folds:

+ We study the new RISC-V ISA and propose a general-purpose VLIW architecture based on the selected RISC-V subsets, each of which can be implemented as one or more flows in VLIW architecture.

+ We integrate a dynamic instruction scheduler to overcome the shortage of a specific RISC-V VLIW compiler for the proposed VLIW architecture. Thus, we can utilize the existing RISC-V GNU toolchain rather than developing a new compiler.

The entire design is constructed, simulated, synthesized, and implemented on a targeted Xilinx field-programmable gate array (FPGA) Virtex-6 (xc6vlx240t-1-ff1156) using Verilog Hardware Description Language (HDL) in ISE 14.7 software suite. The synthesis results demonstrate that the dynamic instruction scheduler is simple and consumes minimal hardware resources. The burden of instruction scheduling is shouldered by the hardware. The existing RISC-V GNU toolchain can be used to compile and execute C programs for functional verification and performance measurement. The performance of our VLIW core is measured by using benchmark programs. The average instructions per cycle (IPC) of the proposed design outperforms that of open-source RISC-V cores. The rest of this paper is organized as follows. Section II reviews the previous VLIW designs. Section III introduces the selection of RISC-V ISAs, and Section IV reveals the VLIW instruction format structured with selected ISAs. Section V presents the entire VLIW architecture. Section VI shows the simulation, synthesis, and implementation results. Section VII provides the conclusions.

II. RELATED WORKS

In academia, several studies have been conducted on VLIW architecture implementation on application-specific integrated circuit (ASIC) and FPGA technology. An ASIC-based four-slot VLIW for multiple stream cipher operations is fabricated on a 180 nm technology to achieve an operating frequency of 200 MHz [5]. It achieves a good tradeoff between high performance and flexibility for multiple basic stream cipher operations. Another 28 nm four-slot VLIW based on a vector ISA operates at 400 MHz [6] at the expense of high power consumption. Despite achieving competitive area and throughput efficiency, high power consumption is the demerit of this design compared with its counterparts. In general, the ASIC implementation provides the best solution for high performance of VLIW processors. Besides, studies have been conducted on FPGA-based VLIW design. A 90- and 45-nm sub-word parallelism RISC architecture with various sub-word-sizes [7] is proposed to obtain the performance comparable with the DSP core TMS320C64X [8]. An adaptable VLIW, whose main parameters are reconfigured at design time [9], is built into the 32-bit VEX ISA [10] to operate up to 174.89 MHz. However, any experimental results and the IPC used to estimate execution time results are not reported. Another VLIW architecture, which is also based on the VEX ISA, is the 32-bit four-issue ρ-VEX [11]. Its maximum frequency reaches up to 74.369 MHz. The substantial advantage of the VEX compiler involves the scheduling of multicycle memory operations to maintain high ILP [12]. VEX-based architectures can be kept relatively simple by using this powerful compiler to achieve a high frequency. Next, a two-stage VLIW based on the HPL-PD ISA [13] can reach 41.8 MHz [14]; its compiler and assembler are based on the Trimaran framework [15]. However, the frequency of this design is relatively low. Another 128-bit four-slot VLIW for issuing multiscalar and vector instructions can achieve 75.825 MHz [16]. Nevertheless, only the synthesis results of stages in the architecture are shown, without any experimental and performance results.

Several VLIW architectures share the burden of ILP exploitation by applying dynamic instruction scheduling methods in the hardware. Thus, they can avoid the problem of binary incompatibility caused by different instruction formats and provide higher performance than traditional VLIW models. Dynamic instruction scheduling VLIW (DISVLIW) [17], which is a hybrid architecture with inherited features, such as ILP exploitation at the compile time of the VLIW processor and dynamic scheduling at the run time of the superscalar, is proposed. The dynamically trace scheduling VLIW [3] integrates a VLIW engine with a conventional superscalar core into a processor with a complicated hardware scheduler. This architecture is used to maintain instruction trace and dispatch suitable instructions to the VLIW engine superscalar core. Besides, the Avatar VLIW processor [18] integrates DynaPack scheduling and packing mechanisms that can analyze the dependence relations of instructions, maintain their correctness, and pack concurrent instructions into a VLIW bundle during run time. Generally, DISVLIW architectures are complicated in terms of hardware implementation owing to their complex instruction tracing and scheduling mechanisms. Furthermore, such architectures require considerable amounts of hardware resources and execution time.

III. RISC-V ISA

In this section, we introduce the RISC-V ISA sets selected for our VLIW implementation, comprising base integer RV32I, multiply-and-divide RV32M, single- and double-precision floating-point RV32F, and RV32D. The details of these ISAs, including instruction layout, opcodes, format types, names, and usage, can be referred in the specifications [2]. Although we choose 32-bit RISC-V subsets to implement our VLIW, this selection does not indicate that RISC-V has only 32-bit instructions. RISC-V proposes 16-bit, 32-bit, 48-bit,

TABLE 1. Summary of selected RISC-V ISAs.

Instr. Set	Type	Instructions	Description
RV32I (base integer)	R	add, sub, sll, xor, srl, sra, or, and, slt	Register–register operations
	I	addi, xori, ori, andi, slti, slli, srli, srai	Register–immediate operations
		lb, lh, lw	Loads
		jalr	Jump and link register
	S	sb, sh, sw	Stores
	B	beq, bne, blt, bge	Conditional branches
	U	lui, auipc	Computational operations
	J	jal	Unconditional jump
RV32M (multiply/ divide)	R	mul, mulh	Multiply operations
		div, rem	Divide operations
RV32F (single-precision floating-point)	R	fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s, fmin.s, fmax.s, fcvt.w.s, fcvt.s.w, fcvt.s.d fmv.x.w, fmv.w.x,	Register–register operations
	S	fsw	Store
	I	flw	Load
	R4	fmadd.s, fmsub.s, fnmadd.s, fnmsub.s	Fused computational operations
RV32D (double-precision floating-point)	R	fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d, fmin.d, fmax.d, fcvt.w.d, fcvt.d.w, fcvt.d.s,	Register–register operations
	S	fsd	Store
	I	fld	Load
	R4	fmadd.d, fmsub.d, fnmadd.d, fnmsub.d	Fused computational operations

and 64-bit instructions. Migration to the VLIW architecture has several attractive advantages. For instance, a 64-bit instruction interface can fetch and process three instructions (1×32-bit $+ 2 \times 16$-bit) in parallel. In various applications, designers can select suitable RISC-V instruction subsets and lengths for the VLIW instruction format. Table 1 summarizes the RISC-V instruction sets selected to implement our VLIW design.

A. RV32I: BASE INTEGER

The first essential subset requirement in any implementation is the base integer RV32I, because it can run a full software stack at the core. RISC-V simplifies instruction decoding. Register operands are consistently in the same locations for all instructions, which means that registers to be read and written are consistently accessed before an instruction is decoded [19]. R-type offers arithmetic instructions (add, sub), logical instructions (and, or, xor), shift instructions (sll, srl, sra), and set less than instructions (slt). I-type provides immediate versions of R-type instructions and loads for words (lw), halfwords (lh), and bytes (lb). S-type has store instructions for words (sw), halfwords (sh), and bytes (sb). B-type instructions compare two registers, and a conditional branch is taken if they are equal (beq), not equal (bne), greater than or equal (bge) or less than (blt). I-type jump and link register (jalr) and J-type jump and link (jal) are unconditional jumps that support procedure calls. In U-type, the load upper immediate (lui) instruction followed by an immediate instruction creates a 32-bit constant. The add upper immediate to PC (auipc) can combine with a jalr for control flow transfers or with a load or store for data accesses.

B. RV32M: MULTIPLY AND DIVIDE

An evident characteristic of RISC-V is its modularity. In addition to the base ISA, RISC-V supports optional standard extensions. One extension can be implemented as one or more separate flows in the VLIW architecture. Next, we expand our implementation to RV32M operations. RV32M adds integer multiply, divide, and remainder instructions to RV32I. Multiplying two 32-bit integers produces a 64-bit product, and the length of registers in the integer register file is 32 bits. RV32M requires two multiply instructions to obtain the 64-bit product. The instruction mul is to obtain the lower 32-bit of the full product, and mulh is to obtain the upper 32 bits. RV32M also offers divide instructions: divide (div) and remainder (rem).

C. RV32F AND RV32D: SINGLE- AND DOUBLE-PRECISION FLOATING POINT

Our design also supports floating-point RV32F and RV32D operations. The implementation complies with the IEEE 754-2008 floating-point standard [20]. RISC-V provides two L-type load instructions (flw, fld) and two S-type store instructions (fsw, fsd) for RV32F and RV32D. In R-type, RV32F and RV32D also support instructions finding maximum (fmax.s, fmax.d) and minimum values (fmin.s, fmin.d) from the pair of source operands in addition to apparent arithmetic operations (fadd.s, fadd.d, fsub.s, fsub.d, fmul.s, fmul.d, fdiv.s, fdiv.d, fsqrt.s, fsqrt.d). No floating-point branch instructions are found. RV32F and RV32D establish conditions for integer branch by comparing two floating-point values and setting the destination register to be one if they are equal (feq.s, feq.d), less than (flt.s, flt.d) or less than or equal (fle.s, fle.d). Conditional integer branch is taken or not taken on the basis of the value at the destination register. For many floating-point algorithms, RISC-V defines fused R4-type instructions that multiply two floating-point values. It then adds (fmadd.s, fmadd.d) or subtracts (fmsub.s, fmsub.d) that product to the third value. Versions that negate the product before summation or subtraction (fnmadd.s, fnmadd.d, fnmsub.s, fnmsub.d) are also found. The purpose of these instructions is to increase the accuracy: they only round once (after multiply) rather than twice (after multiply, then after add or subtract). RV32F and RV32D also support data conversion instructions between data types: integers, 32- and 64-bit floating points (fcvt.w.s, fcvt.s.w, fcvt.w.d, fcvt.d.w, fcvt.d.s, fcvt.s.d). Only RV32F can transpose data between integer and floating-point register files (fmv.x.w, fmv.w.x).

IV. VLIW MICROPROCESSOR ARCHITECTURE

A. VLIW INSTRUCTION FORMAT

After the ISAs to be used are selected, a VLIW instruction format is structured. Eight 32-bit instruction flows are packed

together, forming a 256-bit VLIW instruction, as shown in Figure 1. Each flow in the long instruction performs fixed functions. In various applications, the format of a VLIW instruction, and the number, type, and order of operations in the VLIW instruction can be customized by designers. In our architecture, five flows (1–5) perform the RV32I instructions. Flow 1 performs R-type operations (R), and Flow 2 computes between one register value and one immediate (I/U). Flows 3 and 4 are memory loads and stores (L/S), and Flow 5 is for branch and jump operations (B/J). Flow 6 is used for the multiply and divide instructions of RV32M (M/D). The last two flows (i.e., 7 and 8) contain floating-point RV32F and RV32D instructions (F/D).

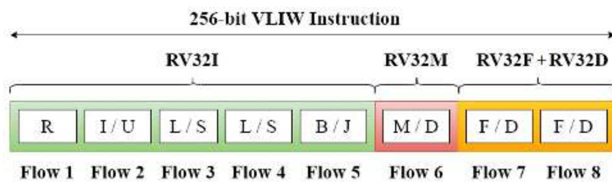

FIGURE 1. VLIW instruction format.

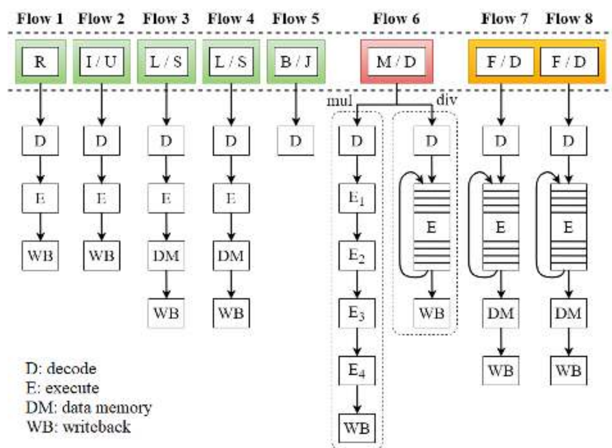

FIGURE 2. Diagram of latency of eight flows.

The datapath for each flow in the architecture is designed with different latencies, as described in Figure 2. Instructions in Flows 1 and 2 are issued through three stages of decode, execute, and writeback without accessing the data memory stage. Load and store instructions in Flows 3 and 4 need to access data memory to read out or store values. Jump or conditional branch decision is taken on the basis of register file values. B/J instructions in Flow 5 are completely processed at the decode stage. However, in the execute stage, multiply operations for Flow 6, consume four cycles (annotated as E_1, E_2, E_3, and E_4) to complete the calculation. The latency of divide in Flow 6 and the floating-point calculations in Flows 7 and 8 are usually unpredictable. The time interval to achieve the final result of these operations is usually unknown and is dependent on the magnitude of input values. Therefore, our design must include stalls during these operations owing to the hazard unit. In conventional VLIW architectures, the instruction scheduling method relies on dedicated compilers. Thus, their compilers must know the latency of all supported operations. They insert sufficient no operations (NOPs) equal to the most prolonged latency of independent operations within a VLIW instruction to solve data dependencies between VLIW instructions [21]. In other words, the subsequent instruction cannot be processed until the previous one passes the writeback stage. Despite solving dependencies, inserting long NOPs among VLIW instructions exaggerates code size. In our architecture, the hazard unit can perform data forwarding among long instructions to puzzle out data dependencies without long intermediary NOPs.

B. PROPOSED VLIW ARCHITECTURE

After the datapath latencies are determined, each flow is cumulatively designed. The design steps are conceptualized similar to those of building a classical pipepined five-stage RISC-V architecture in [22] with minor changes. Figure 3 describes the entire VLIW hardware architecture, including the modules, namely, fetch, instruction scheduler, decode, execute, data memory, and writeback. Fetch is responsible for reading out original sequential instructions from the instruction memory. Subsequently, the instruction scheduler exploits potential parallelism among the instructions from fetch. Next, independent instructions are packed into one VLIW instruction and delivered to the decode stage. In this stage, the control units receive opcode and function fields in sub-instructions to generate appropriate corresponding control signals for the execute, data memory, and writeback stages. In addition, source operands required for calculations in the execute stage are read out from floating-point 64-bit F and integer 32-bit X register files based on source addresses. The execute module contains arithmetic logic units (ALUs) that perform on source operands. Moreover, multiplexers exist ahead of ALUs for the data forwarding mechanism. Only load and store instructions of two L/S and two F/D flows (i.e., 3–4 and 7–8) need to access the data memory. The last writeback stage merely writes the calculation results or data from the data memory back to the register files with a corresponding destination address.

1) FETCH STAGE

The design of fetch stage is illustrated in Figure 4. The significant functionality of the fetch stage is to control the reading out of eight 32-bit instructions from the instruction memory simultaneously based on the program counter (PC). Given the original sequential program stored in the instruction memory, the fetch stage must integrate a BJChecker module to examine whether branch or jump instructions are present in the eight received 32-bit instructions. The BJChecker module then calculates the address of branch instructions (*BrInstAdd*) or the

FIGURE 3. The diagram of entire implemented VLIW microprocessor.

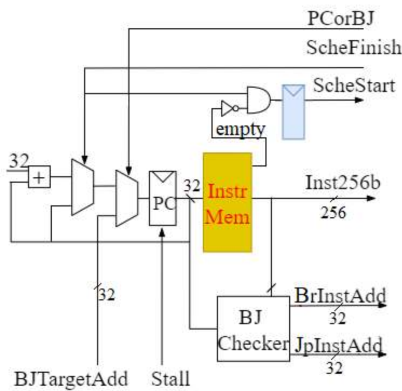

FIGURE 4. Structure of fetch module.

address of jump instructions (*JpInstAdd*) based on the current PC and the positions of these branch or jump instructions. Subsequently, the eight instructions are transferred to the instruction scheduler along with a *ScheStart* signal. When scheduling is completed, the scheduler notifies fetch using a *ScheFinish* signal to retrieve the next eight instructions. The decode stage calculates the branch targeting address *BJTargetAdd* based on *BrInstAdd* and *JpInstAdd* from fetch. If the branch or jump is taken, then fetch moves to the *BJTargetAdd* to retrieve the eight new instructions.

2) DYNAMIC INSTRUCTION SCHEDULER

As aforementioned, the shortcoming of studying a new ISA and implementing a VLIW architecture based on that ISA is the lack of a powerful compiler to schedule original instructions in accordance with the VLIW instruction format. Designing a compiler for a new VLIW architecture is rigorous, despite the compiler being designed to be compatible with the instruction format of only a specific VLIW architecture. This binary incompatibility restricts VLIW usage popularity. Meanwhile, the RISC-V GNU toolchain, which can be downloaded from [4], is a capable C compiler that

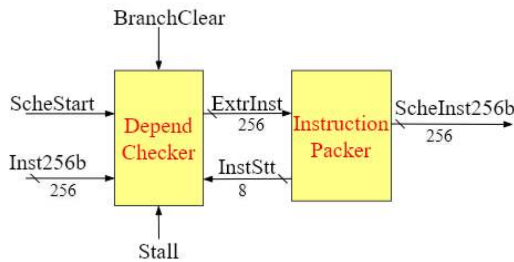

FIGURE 5. Block diagram of instruction scheduler.

supports 32-bit and 64-bit RISC-V ISAs. We design a simple instruction scheduler to exploit the parallelism potential of sequentially original instructions dynamically. This scheduler is used to overcome the lack of an RISC-V VLIW compiler and leverage the open-source RISC-V compiler. The instruction scheduling algorithm traces dependencies throughout the original eight instructions from the fetch stage based on instruction types, source, and destination addresses. Our goal is to guarantee that (i) no instruction is lost to maintain program correctness, (ii) no data dependencies occur within one long instruction, and (iii) the cost of hardware resource is reasonable. Centered on these principles, the instruction scheduler is constructed with two submodules, namely, the dependence checker (DC) and instruction packer (IP), as shown in Figure 5. The former is responsible for receiving and checking dependencies between eight instructions simultaneously. The latter indicates the ready status of eight instruction slots (*InstStt*) and fits the extracted instructions from the checker (*ExtrInst*) into their corresponding slot. The instruction scheduling operation is described in the algorithm flowchart of Figure 6. When receiving the asserted *ScheStart* signal from fetch, the scheduler examines eight instructions line-by-line. If the current instruction I[i] is branch/jump and independent on previous examined independent instructions I[1]–I[i−1], all the instructions will be extracted to IP and removed. However, if I[i] is dependent, I[1]–I[i−1] will be extracted. In the next scheduling turn, only I[i] will be scheduled in another separate VLIW instruction. In case that the branch decision or jump is taken, instructions behind branch or jump instruction I[i] will not be scheduled. In another case, I[i] is not branch or jump and it does not depend on I[1]–I[i−1], then the instructions will be packed together if its slot checked by IP is still available. Next, the scheduler will check the next instruction. In general, when dependencies occur between I[i] and I[1]–I[i-1], the dependent instruction I[i] is examined and arranged with subsequent instructions in the next turn. The whole process is repeated until all eight instructions are sorted. Given this simplicity, only several temporary registers are needed to store the remaining instructions for the next scheduling turn. Figure 7 illustrates the scheduling method for eight sequential instructions. In the instruction dependence analysis graph, a node represents an instruction, and an arrow indicates that the source of subsequent instructions utilizes the result of the previous instruction. The I_1 and I_2 instructions are independent; thus, the DC prioritizes them and retains the six remaining instructions for the next scheduling turn. Next, the IP fits I_1 and I_2 into their matching slots, inserts NOPs into empty slots to create a V_1 instruction, and transfers it to the decode stage. Dependencies occur from I_3 to I_8, as described in the graph. Hence, I_3–I_8 are arranged into different V_2–V_4 instructions. After the arrangement has is completed, the instruction scheduler receives the next eight instructions from the fetch stage. Our dynamic instruction scheduling feature ensures the proper matching of sequential instructions in their slot while preserving program accuracy.

3) DECODE STAGE

The decode module decodes the instructions passed down from the fetch module. The structure of the decode stage is illustrated in Figure 8. The control units (from *CuOp1* to *CuOp8*) create control signals based on the opcode and function fields. These control signals must be pipelined along with the data such that they remain synchronized with the instruction. Moreover, the decode stage reads the source operands from two integer 32-bit X and floating-point 64-bit F register files and passes them to the execute module for execution. RV32F and RV32D use a separate set of 32 64-bit floating-point registers. Increasing the space of register address fields for decoding RISC-V instruction formats is unnecessary by doubling the number of registers. The length of floating-point registers is 64-bit, and RV32F uses only the lower 32 bits. The first register of the floating-point register file is not hardwired to zero, which differs from that of the integer register file. The values of the registers (*Rs1Val*, *Rs2Val*, or *Rs3Val* of flows 1 – 8) are read and passed to the execute module, whereas the results of operations are written back to the register files (*AluReOp1/2/6/7/8* or *DmReOp3/4*). Besides, SignEx modules are used to extend the bit sign of immediates.

Branch instructions may pose control hazards when the VLIW processor cannot decide which instructions to pick next, as the branch decision has yet to be made. Waiting until the end of the execution stage to determine whether the branch is taken can result in a substantial misprediction penalty. The VLIW makes a branch decision early in the decode stage of Flow 5 by integrating *BranchCheck* module to reduce the penalty. The decision is simply a comparison between the values of two registers *Rs1ValOp5* and *Rs2ValOp5*. If the branch is taken, then the instruction scheduler discards the long instruction being processed, and fetch moves to the new target PC *BJTargetAdd*. If a previous instruction is issued to determine one of the sources operating for the branch and has not been written into the register files, then the data forwarding forwards the necessary operands through multiplexers when available, or the pipeline is stalled by the hazard unit until the data are ready.

4) EXECUTE STAGE

From the decode stage, the execute stage receives appropriate control signals and source operands to perform calculations. As shown in Figure 9, this stage consists of several functional

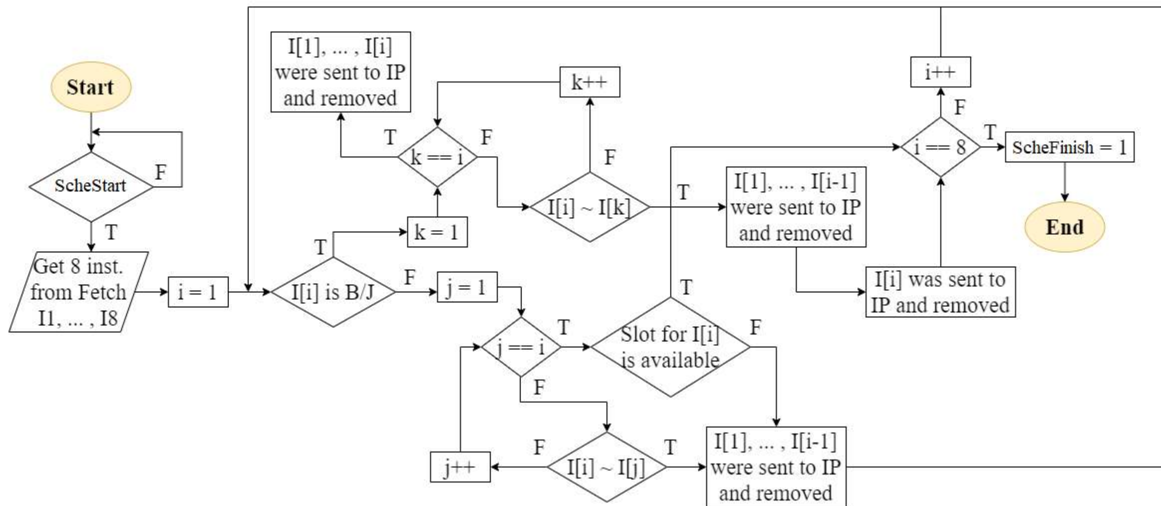

FIGURE 6. The algorithm flowchart of sequential dynamic instruction scheduling.

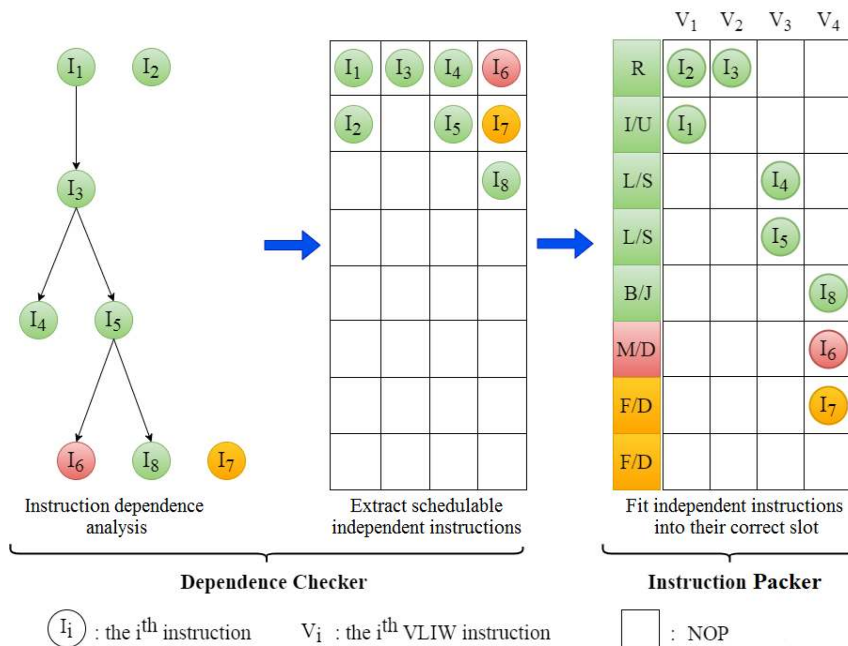

FIGURE 7. Illustration of sequential dynamic instruction scheduling algorithm.

units, such as R-type register–register (AluR), I- and U-type register–immediate (AluIU), two 32-bit adders for L/S flows, AluM for M/D operations, and two floating-point ALUs for F/D flows (AluFDs). Moreover, multiplexers exist ahead of the functional units to select source operands directly from register files or either the data memory or writeback in case of data dependencies. Operations in AluR and AluIU take one clock cycle to complete their calculation. Whereas, in AluM, integer multiply consumes four clocks and divide loops in a variable iteration number to obtain the final result. The execute unit of Flows 3 and 4 are simply 32-bit adder to calculate the address for data memory access.

The floating-point coprocessor operates in parallel with integer cores in most general-purpose processors to offload massive computational and high-latency floating-point instructions from the central processor [23]. In the VLIW, floating-point operations can be combined within a long VLIW instruction and run in parallel with integer operations. In particular, the AluFD of Flows 7 and 8 are integrated to perform single- and double-precision floating-point operations, such as add (ADD), subtract (SUB), divide (DIV), square root (SQRT), multiply (MUL), fused multiply and accumulate (FUSED), convert (CVT), compare (CMP), and classify (CLS). The particular architecture for the floating-point

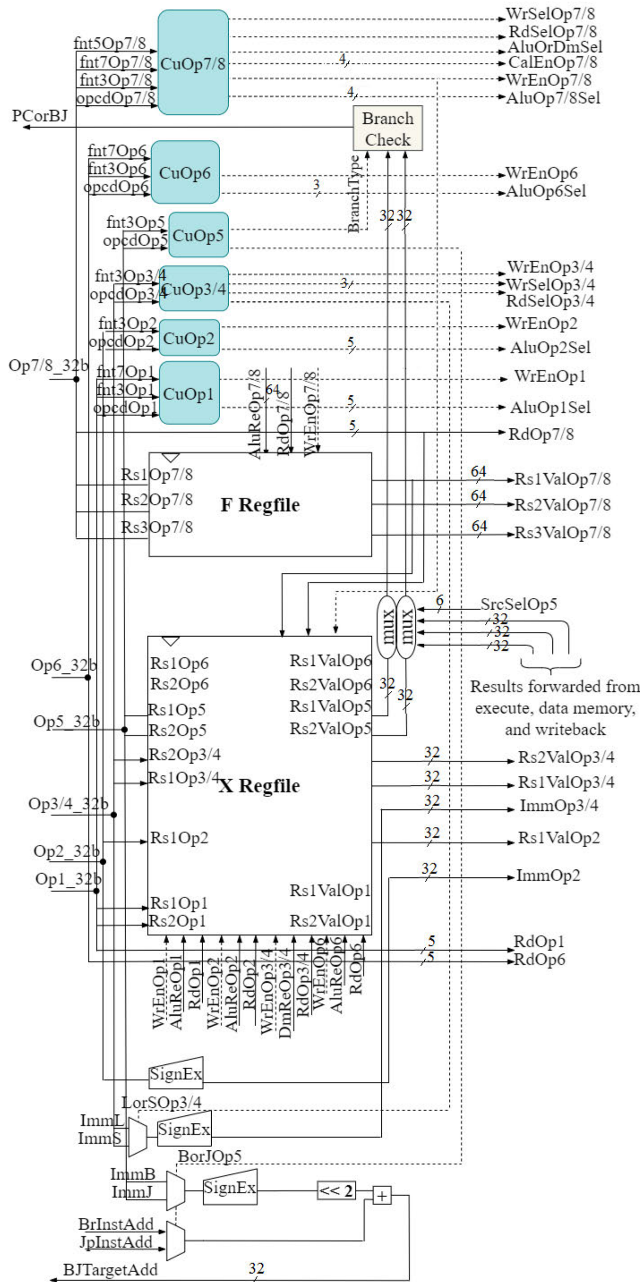

FIGURE 8. Structure of decode module.

adder, subtractor, divider, multiplier, and comparator modules are detailed in [24]. The two floating-point flows designed with floating-point RISC-V instructions adhere to the IEEE 754-2008 specification. They obtain source operands, a rounding mode, enable, and control signals from the decode stage. Only the module with an asserted enable signal calculates the input data operands. Next, it outputs the computational results and exceptions as invalid (*invalidOp7/8*), divide-by-zero (*divByZeroOp7/8*), overflow (*overflowOp7/8*), underflow (*underflowOp7/8*), and inexact operations (*inExactOp7/8*). During the floating-point calculations, the pipeline stages are stalled. Upon completion

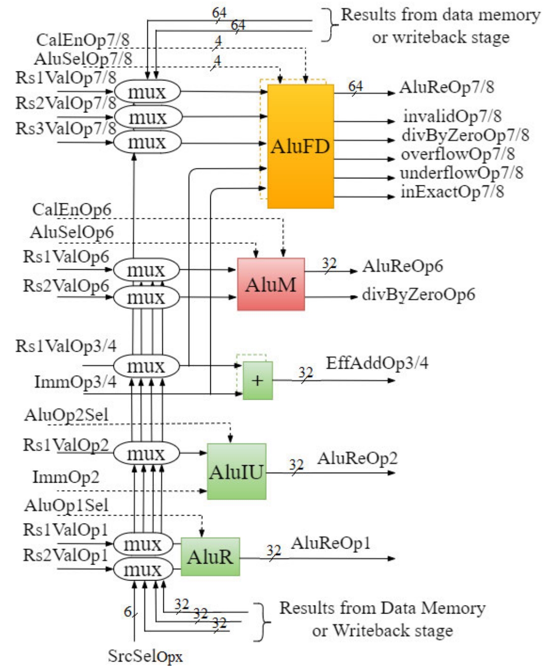

FIGURE 9. Structure of execute module.

of the current calculation, a finish signal is asserted, and the final results (*AluReOp7/8*) are transferred back to the floating-point register file. Furthermore, an extra special case detector module examines particular inputs, such as not-a-number, infinity, denormalized, and zero. In exceptional matched cases, the module provides output for these specific inputs, as defined in the IEEE 754-2008 standard.

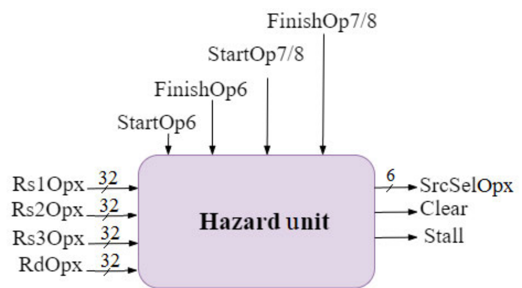

FIGURE 10. Block diagram of hazard unit.

5) HAZARD UNIT

A hazard unit is designed as a combinational logic block to solve pipeline hazards through data forwarding and instruction stalling. The forwarding technique is used to address data hazards that occur when an instruction requires the results of previous pipeline instructions that have not been written on the register file by the time the current instruction reads its source operands from the register file. As shown in Figure 10, the hazard unit receives the source and destination addresses from the decode, execute, data memory,

and writeback stages of flows (annotated as *Rs1Opx*, *Rs2Opx*, *Rs3Opx*, and *RdOpx*). The hazard unit bypasses the necessary operands from the memory or writeback stage to the ALUs in the execute stage or to the BranchCheck in the decode stage. This technique requires additional multiplexers ahead of the ALUs to select the operands from either the register file, execute (only for BranchCheck), memory, or writeback. For ALUs, if the memory and writeback stages contain matching destination addresses, then the memory stage is prioritized, as it contains the latest executed instructions. In addition, the hazard unit stalls the pipelined stages (when *StartOp6/7/8* are asserted) until the results from the load, divide, or floating-point calculations are usable (*FinishOp6/7/8* are asserted). When the calculations of execute stage in Flow 7 and 8 are finished, a *Clear* signal is used to clear internal registers for avoiding bogus information.

V. EXPERIMENTAL RESULTS

A. FUNCTIONAL VERIFICATION METHOD

To validate the function of our proposed VLIW architecture, we verify each flow and the entire design. First, we write C programs corresponding to the operations of the tested flow to test each flow separately. Second, we install the RISC-V GNU toolchain available at [4] to compile the C programs in RISC-V-executable and machine code files. In addition, we build the Whisper simulator [25], which is an RISC-V instruction set simulator developed to verify the Swerv microcontroller. This simulator allows the user to run RISC-V codes without an RISC-V hardware, as a "golden model," to compare results. The C programs are compiled and run on the Whisper simulator to obtain the execution results. Next, we input the same testing values into the testbench written in Verilog HDL to conduct the simulation. Ultimately, we compare the execution results from the Whisper simulator with the simulation results from the VLIW.

Subsequently, we utilize five integer benchmark programs, including quick sort (qsort), matrix multiplication (matmul), vector–vector addition (vvadd), median filter (median), and binary multiply (multiply) to verify the overall design. These benchmarks are available in RISC-V GNU toolchain [26]. Given that they mainly operate on six integer flows (1–6), to verify the entire design, including both Flows 7 and 8, we define a floating-point benchmark (fbench). This program performs floating-point operations on 100 pairs of different float and double values. These programs are then compiled and run on the Whisper simulator to obtain the execution results. Next, we dump corresponding RISC-V assembly codes and store them into the instruction memory as text files to conduct simulations. The results processed by the VLIW engine are retained in the data memory. Finally, we compare the execution results from the Whisper simulator with the simulation results from the VLIW design.

To prove that our proposed VLIW design functions properly after implementation, we conduct experiments on a Xilinx FPGA Virtex-6 platform (xc6vlx240t-1-ff1156). Each benchmark program is still compiled and stored in the instruction memory. The whole system is then downloaded to the FPGA board. The output results processed by VLIW engine are stored in the data memory. Given that the data in the data memory are stored in bytes, we integrate a memory controller to control reading bytes. We use ChipScope [27], which is a software-based logic analyzer, to observe the onboard results. We can set triggering options and display the waveform of the desired FPGA chip signals by inserting an integrated controller core and logic analyzer into the design. The simulation and onboard experimental results show that our VLIW core functions correctly as expected. The design performance is evaluated through these benchmarks, as presented in the succeeding IPC section.

TABLE 2. Performance of the VLIW core with the single-ended oscillator frequency $f_{osc} = 66$ MHz.

Benchmark	Number of instructions	Clock cycles	IPC	Throughput
matmul	32,249	38,725	0.833	54.978 (MOPS)
multiply	20,899	21,769	0.960	63.362 (MOPS)
vvadd	8,026	8,207	0.978	64.548 (MOPS)
qsort	123,505	125,871	0.981	64.746 (MOPS)
median	4,152	4,054	1.024	67.584 (MOPS)
fbench	60,537	153,291	0.395	26.07 (MFOPS)

MOPS: Million operations per second
MFOPS: Million floating-point operations per second

B. IPC

IPC is used to evaluate the performance of a processor architecture. IPC is defined as the average instruction number executed in every clock cycle. In addition to the statistical counters for exceptional cases mentioned in the Execute Stage section, we integrate two other performance counters into our design: one to count the number of operations and one to count the total clock cycles to complete a benchmark program. The final IPC result is derived by dividing the number of instructions with the number of clock cycles. Table 2 presents the performance of our VLIW architecture on six selected benchmarks. The proposed VLIW obtains the highest IPC for the median benchmark (1.024). By contrast, the lowest IPC value belongs to the fbench program (0.395), as the floating-point program execution time is generally unpredictable. The larger the number is, the more time the calculation needs. This finding leads to an unexpectedly low IPC for floating-point-related programs. The table also shows the throughput for each benchmark that is defined by the number of operations or floating-point operations per second. The throughput can be derived by multiplying the number of IPC with the operating frequency of the design. In particular, our design utilizes the onboard single-ended oscillator frequency $f_{osc} = 66$ MHz. The average IPC from five integer benchmarks obtains 0.955 and the average throughput value is

TABLE 3. Device utilization for the entire VLIW architecture and its main modules.

Device utilization summary		The entire design	Fetch	Instruction Scheduler	Decode	Execute	Data Memory
Slice Logic Utilization	No. of slice registers	21,476 (7%)	185 (0%)	799 (0%)	3,776 (1%)	9,188 (3%)	32 (0%)
	No. of slice LUTs	69,572 (46%)	662 (0%)	2,462 (1%)	18,469 (12%)	14,398 (9%)	230 (0%)
	+ Number used as logic	69,522 (46%)	662 (0%)	2,462 (1%)	18,469 (12%)	14,353 (9%)	230 (0%)
	+ Number used as memory	50 (0%)	N/A	N/A	N/A	45 (0%)	N/A
Slice Logic Distribution	No. of LUT–FF pairs used	76,327	717	2,555	18,491	16,874	230
	+ Number with an unused FF	54,851 (71%)	532 (74%)	1,756 (68%)	14,715 (79%)	7,686 (45%)	198 (86%)
	+ Number with an unused LUT	6,755 (8%)	55 (7%)	93 (3%)	22 (0%)	2,476 (14%)	0 (0%)
	+ No. of fully used LUT–FF pairs	14,721 (19%)	130 (18%)	706 (27%)	3,754 (20%)	6,712 (39%)	32 (13%)
Maximum frequency (MHz)		83.739	229.991	173.805	150.082	169.722	238.207

63.044 million operations per second (MOPS). Corresponding to the IPC of 0.395 from fbench, the throughput gains 26.07 million floating-point operations per second (MFOPS). For all six benchmarks, our eight-issue VLIW core achieves the average IPC of 0.862.

To measure the acceleration of our VLIW core compared with a single-issue core, we design a single-issue pipelined five-stage 32-bit RISC-V architecture. The architecture's ALU supports the same integer and floating-point instructions with the same latencies, as described in Figure 2. Subsequently, the same benchmarks are utilized to verify the function and measure of the IPC values for the single-issue core. Figure 11 shows a comparison of IPC values from six benchmarks between the proposed VLIW and the single-issue core. Figure 12 presents the IPC speedup of our VLIW engine for each benchmark. The speedup results show that our VLIW architecture accelerates 1.344 times faster than the single-issue core on average.

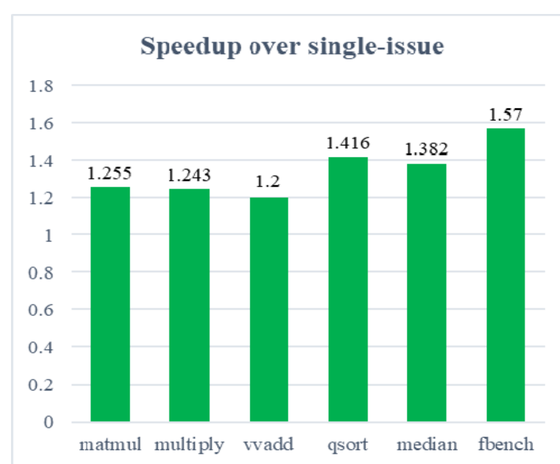

FIGURE 12. Speedup of the proposed VLIW over the single-issue core.

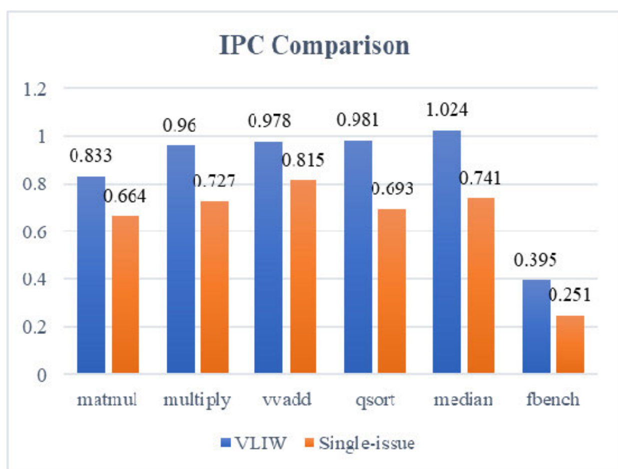

FIGURE 11. IPC comparison between the VLIW and single-issue core.

C. SYNTHESIS RESULTS

The complete architecture of our proposed VLIW is synthesized on the targeted Virtex-6 using ISE 14.7 software suite. The platform has 37,680 slices, with each slice structured with four six-input LUTs and eight D-type FFs, totaling to 150,720 LUTs and 301,440 FFs [28]. Table 3 illustrates the slice logic utilization and distribution of the entire proposed VLIW design and its main modules, including fetch, instruction scheduler, decode, execute, and data memory. For slice logic utilization, the ISE XST synthesizer represents the number of slice registers (or FFs) and LUTs used. The decode module consumes 18,469 (12%) slice LUTs, with an available frequency of 150.082 MHz. The execute module uses the greatest number of slice registers, that is, 9,188 (3% compared with a total number of 301,440 FFs), and 14,398 LUTs (14,353 LUTs are used to generate combinational functions and 45 LUTs are configured as distributed RAM because the HDL may contain a small array of read/write registers). In terms of slice logic distribution, the execute module has 7,686 unused FFs (using only slice LUTs), 2,476 unused LUTs (using only FFs in slices), and 6,712 fully used LUT–FF pairs. Compared with other modules, the instruction scheduler consumes reasonable resources, and its operating frequency is at 173.805 MHz. It satisfies one of our initial criteria about hardware resource for dynamic instruction

TABLE 4. Performance comparison among RISC-V cores.

	This work		DarkRISCV [29]	Kronos RISC-V [30]	PicoRV32 [31]	NEORV32 [32]
	Eight-issue VLIW core	Single-issue core				
ISA	RV32IMFD	RV32IMFD	RV32I	RV32I + Zicsr, Zifenci	RV32IMC	RV32IMCU + Zicsr, Zifenci
FPGA Chip	Virtex-6 xc6vlx240t	Virtex-6 xc6vlx240t	Spartan-6 XC6SLX9	iCE40UP5K	Kintex-7T xc7k70t	Cyclone IV EP4CE22F17C6N
Maximum frequency (MHz)	83.739	89.440	100	24	400	100
Average IPC	0.862	0.65	0.7	0.578	0.25	0.184
Hardware utilization (LUTs)	69,572	31,676	1,500	5,280	2,019	3,800

scheduling. Our VLIW architecture can reach the maximum frequency of 83.739 MHz. In addition, we strive to synthesize our design without two floating-point flows. We find that the maximum frequency of our proposed design can reach 97.943 MHz, with slice registers reduced to 6,177 and slice LUTs reduced to 20,109, respectively.

D. DISCUSSION

Dynamic instruction scheduler integration can overcome the lack of a specialist compiler for our architecture. The available RISC-V GNU toolchain can be leveraged for functional verification and performance evaluation. However, our design recognizes a few tradeoffs. First, as stated in the Synthesis Results section, the instruction scheduler consumes substantial hardware utilization. The instruction scheduler is simple to utilize less hardware than other modules. Second, scheduling eight sequential instructions from the fetch stage requires one extra clock cycle for each scheduling turn. This requirement increases program execution time, indicating that IPC is slightly degraded. Our VLIW core still achieves higher IPC compared with existing open-source RISC-V cores at the expense of hardware resource, as presented in Table 4. The integration of floating-point flows is the main reason for this high resource consumption. Hardware optimization and effective scheduling algorithm implementation should be considered in future studies to achieve outstanding performance.

VI. CONCLUSION

RISC-V is a potential ISA that is promising for academic studies and as a future industry standard. VLIW microprocessors can benefit from the flexibility and modularity of the RISC-V ISA. This study describes how the VLIW architecture can be implemented with an instruction format for eight operations from RISC-V subsets, including RV32I, RV32M, RV32F, and RV32D. This study suggests that an instruction scheduler should be integrated with sequential algorithm scheduling instructions to solve the compiler shortage for our VLIW architecture. Thus, we can utilize the available RISC-V GNU toolchain without needing to develop a new RISC-V VLIW compiler. The instruction scheduling algorithm is relatively simple and requires less hardware utilization. Functional correctness is verified through benchmark programs by comparing the results of our VLIW with those of the Whisper simulator and observing onboard waveforms using ChipScope. The benchmark programs are also used to evaluate the IPC performance of our VLIW and demonstrate that our design still speeds up compared with existing open-source RISC-V cores. The maximum frequency of our proposed VLIW reaches 83.739 MHz, and the number of slice registers and LUTs are 21,476 (7%) and 69,572 (46%), respectively.

ACKNOWLEDGMENT

The authors would like to thank the Taiwan Semiconductor Research Institute (TSRI) for the support of design and simulation tools.

REFERENCES

[1] X. Li and D. L. Maskell, "Time-multiplexed FPGA overlay architectures: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 5, pp. 1–19, Oct. 2019.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovíc. (Dec. 2019). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*. [Online]. Available: https://riscv.org/specifications/

[3] A. F. de Souza and P. Rounce, "Dynamically scheduling VLIW instructions," *J. Parallel Distrib. Comput.*, vol. 60, no. 12, pp. 1480–1511, Dec. 2000.

[4] *RISC-V GNU Compiler Toolchain*. Accessed: Feb. 5, 2020. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

[5] L. Nan, X. Yang, X. Zeng, W. Li, Y. Du, Z. Dai, and L. Chen, "A VLIW architecture stream cryptographic processor for information security," *China Commun.*, vol. 16, no. 6, pp. 185–199, Jun. 2019.

[6] A. Bytyn, R. Leupers, and G. Ascheid, "An application-specific VLIW processor with vector instruction set for CNN acceleration," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[7] S. Khan, M. Rashid, and F. Javaid, "A high performance processor architecture for multimedia applications," *Comput. Electr. Eng.*, vol. 66, pp. 14–29, Feb. 2018.

[8] *Digital Signal Processor TMS320C64x From Texas Instruments*. Accessed: Jul. 15, 2020. [Online]. Available: https://www.ti.com/lit/ug/spru395b/spru395b.pdf

[9] C. Pham-Quoc, B. Kieu-Do-Nguyen, and A.-V. Dinh-Duc, "Adaptable VLIW processor: The reconfigurable technology approach," in *Proc. Int. Conf. Adv. Technol. Commun. (ATC)*, Oct. 2017, pp. 120–125.

[10] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. San Mateo, CA, USA: Morgan Kaufmann, 2004.

[11] S. Wong, F. Anjam, and F. Nadeem, "Dynamically reconfigurable register file for a softcore VLIW processor," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2010, pp. 969–972.

[12] M. Koester, W. Luk, and G. Brown, "A hardware compilation flow for instance-specific VLIW cores," in *Proc. Int. Conf. Field Program. Log. Appl.*, 2008, pp. 619–622.

[13] V. Kathail, M. Schlansker, and B. R. Rau, "HPL-PD architecture specsification: Version 1.1," HP Lab., Palo Alto, CA, USA, Tech. Rep. HPL-93-80 (R.1), 2000.

[14] W. W. S. Chu, R. G. Dimond, S. Perrott, S. P. Seng, and W. Luk, "Customisable EPIC processor: Architecture and tools," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, vol. 3, 2004, pp. 236–241.

[15] *Trimaran: An Infrastructure for Research in Instruction Level Parallelism*. Accessed: May 23, 2020. [Online]. Available: http://www.trimaran.org

[16] M. I. Soliman, "A VLIW architecture for executing multi-scalar/vector instructions on unified datapath," in *Proc. Saudi Int. Electron., Commun. Photon. Conf.*, Apr. 2013, pp. 1–7.

[17] S. Jee and K. Palaniappan, "Dynamically scheduling VLIW instructions with dependency information," in *Proc. 6th Annu. Workshop Interact. Between Compil. Comput. Archit. (INTERACT)*, 2002, pp. 15–23.

[18] S. L. Chu, G. S. Li, and R. Q. Liu, "DynaPack: A dynamic scheduling hardware mechanism for a VLIW processor," *Appl. Math. Inf. Sci., Int. J.*, vol. 6, no. 3, pp. 983–991, 2011.

[19] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. San Francisco, CA, USA: Strawberry Canyon LLC, 2017.

[20] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4610935

[21] M. Dubois, M. Annavaram and P. Stenstrom, *Parallel Computer Organization and Design*. Cambridge, U.K.: Cambridge Univ. Press, 2012.

[22] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface—RISC-V Edition*. Amsterdam, The Netherlands: Elsevier, 2018.

[23] V. Patil, A. Raveendran, P. M. Sobha, A. D. Selvakumar, and D. Vivian, "Out of order floating point coprocessor for RISC V ISA," in *Proc. 19th Int. Symp. VLSI Design Test*, Jun. 2015, pp. 1–7.

[24] Y. Li, *Computer Principles and Design in Verilog HDL*. Beijing, China: Tsinghua Univ. Press, 2015.

[25] *Western Digital's Open Source RISC-V SweRV Instruction Set Simulator*. Accessed: Feb. 12, 2020. [Online]. Available: https://github.com/westerndigitalcorporation/swerv-ISS

[26] *RISC-V GNU Toolchain, Riscv-Tests*. Accessed: Feb. 5, 2020. [Online]. Available: https://github.com/riscv/riscv-tests

[27] Xilinx. (Mar. 20, 2013). *PlanAhead Tutorial: Debugging With ChipScope (UG677)*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/PlanAhead_Tutorial_Debugging_w_ChipScope.pdf

[28] Xilinx. (Feb. 3, 2012). *Virtex-6 FPGA Configurable Logic Block User Guide (UG364)*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug364.pdf

[29] *DarkRISCV*. Accessed: Aug. 3, 2020. [Online]. Available: https://github.com/darklife/darkriscv

[30] *Kronos RISC-V*. Accessed: Aug. 3, 2020. [Online]. Available: https://github.com/SonalPinto/kronos

[31] *PicoRV32—A Size-Optimized RISC-V CPU*. Accessed: Aug. 3, 2020. [Online]. Available: https://github.com/cliffordwolf/picorv32

[32] *The NEORV32 Processor*. Accessed: Aug. 3, 2020. [Online]. Available: https://github.com/stnolting/neorv32

NGUYEN MY QUI was born in Vietnam, in 1994. He received the B.S. degree in electronics and telecommunications from the Ho Chi Minh City University of Science, Ho Chi Minh City, Vietnam, in 2016, and the M.S. degree in electronics and computer engineering from the National Taiwan University of Science and Technology, Taipei City, Taiwan. His research interests include digital systems design with field-programmable gate array and computer architecture.

CHANG HONG LIN (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from National Taiwan University, Taipei City, Taiwan, in 1997 and 1999, respectively, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, USA, in 2003 and 2007, respectively. He is currently a Professor with the Department of Electronics and Computer Engineering, National Taiwan University of Science and Technology. His research interests include ubiquitous camera frameworks, code compression for embedded systems, and hardware/software co synthesis.

POKI CHEN (Member, IEEE) was born in Chiayi, Taiwan, R.O.C., in 1963. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from National Taiwan University, Taipei City, Taiwan, in 1985, 1987, and 2001, respectively.

He was a Lecturer, an Assistant Professor, and an Associate Professor with the Department of Electronics Engineering, National Taiwan University of Science and Technology (NTUST), from 1998 to 2001, from 2001 to 2006, and from 2006 to 2011, respectively. He is currently a Professor with the Department of Electronics and Computer Engineering, NTUST. His research interests include analog/mixed-signal IC design and layout, with a special focus on time–domain signal processing circuits such as time–domain smart temperature sensors, time-to-digital converters, digital pulse generators (DTCs), time-domain ADCs, and high-accuracy DACs. He is also interested in creating innovative analog applications for field-programmable gate array (FPGA) platforms such as FPGA smart temperature sensors and FPGA digital-to-time and time-to-digital converters. He has been an Organizer of IEEE International Conferences on Intelligent Green Building and Smart Grid since 2014 and has served as a Keynote/Invited Speaker, a TPC Member, and a Session Chair of various IEEE conferences such as the SOCC, VLSI-DAT, IFEEC, ISESD, NoMe TDC, ISNE, ASID, and so on. Moreover, he has been served as an Associate Editor for IEEE Transactions on Very Large-Scale Integration Systems and IEEE Access since 2011.