Hindawi Publishing Corporation
VLSI Design
Volume 2016, Article ID 8787919, 12 pages
http://dx.doi.org/10.1155/2016/8787919

Research Article

FuMicro: A Fused Microarchitecture Design Integrating In-Order Superscalar and VLIW

Yumin Hou,[1] Hu He,[1] Xu Yang,[2] Deyuan Guo,[1] Xu Wang,[1] Jiawei Fu,[1] and Keni Qiu[3]

[1]*Institute of Microelectronics, Tsinghua University, Beijing 100084, China*
[2]*School of Software, Beijing Institute of Technology, Beijing 100081, China*
[3]*College of Information Engineering, Capital Normal University, Beijing 100048, China*

Correspondence should be addressed to Hu He; hehu@tsinghua.edu.cn

Received 12 June 2016; Revised 4 October 2016; Accepted 19 October 2016

Academic Editor: Jose Carlos Monteiro

This paper proposes FuMicro, a fused microarchitecture integrating both in-order superscalar and Very Long Instruction Word (VLIW) in a single core. A processor with FuMicro microarchitecture can work under alternative in-order superscalar and VLIW mode, using the same pipeline and the same Instruction Set Architecture (ISA). Small modification to the compiler is made to expand the register file in VLIW mode. The decision of mode switch is made by software, and this does not need extra hardware. VLIW code can be exploited in the form of library function and the users will be exposed under only superscalar mode; by this means, we can provide the users with a convenient development environment. FuMicro could serve as a universal microarchitecture for it can be applied to different ISAs. In this paper, we focus on the implementation of FuMicro with ARM ISA. This architecture is evaluated on gem5, which is a cycle accurate microarchitecture simulation platform. By adopting FuMicro microarchitecture, the performance can be improved on an average of 10%, with the best performance improvement being 47.3%, compared with that under pure in-order superscalar mode. The result shows that FuMicro microarchitecture can improve Instruction Level Parallelism (ILP) significantly, making it promising to expand digital signal processing capability on a General Purpose Processor.

1. Introduction

With the evolution of wireless communication protocols, digital signal processing becomes more and more demanding in applications of embedded systems. As digital signal processors (DSPs) become increasingly indispensable, many embedded systems embrace both General Purpose Processor (GPP) cores and DSP cores.

Many SoCs use ARM+DSP architecture [1–4] in recent years. For example, the Integra ARM+DSP architecture integrates the ARM Cortex-A8 processor and high performance DSP. Figure 1 shows a typical ARM+DSP architecture [5].

Architectures incorporating GPP and DSP have their common headaches. The GPP and DSP require different instruction sets and they need independent development environment, which brings overwhelming workload to software design, making such architectures time and effort consumptive and inconvenient to the users at the same time. Communication between GPP and DSP brings more overhead [6].

We aim to enhance digital signal processing ability on GPP cores to replace the GPP+DSP architecture by a single core.

To enhance DSP capability is to enable the processor to do more operations in one cycle. We either arrange more operations in a single instruction or issue more instructions in one cycle. Since we hope to keep the ISA of the GPP core unchanged, we consider the second approach. To issue more instructions in one cycle means to increase Instruction Level Parallelism (ILP).

Several methods increase ILP of a processor, including in-order superscalar, Out-of-Order (OoO) superscalar, and VLIW [7].

As in-order superscalar approach dispatches instructions according to the register dependence table, the count of the instruction parallelism checking is proportional to the square

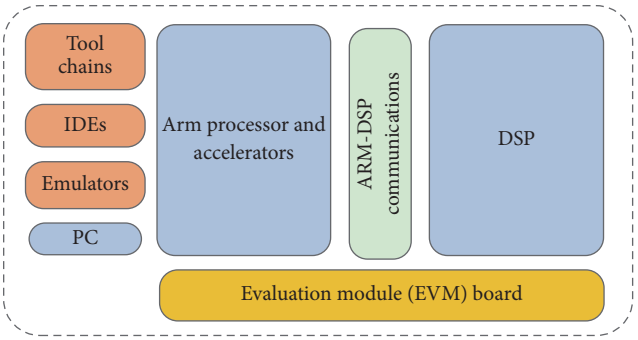

Figure 1: Typical ARM+DSP architecture.

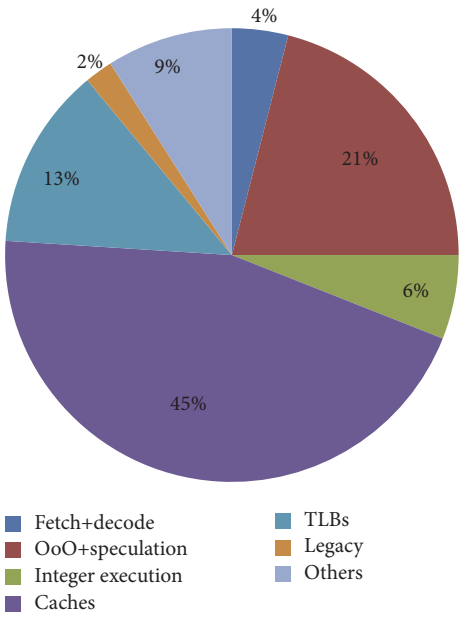

Figure 2: Power consumption for non-compute-heavy applications [10].

of the dispatching width. Slight increase to the issue width can cause great delay to the critical path, which is not allowed by the high frequency demand; thus, though the program provides enough parallelism, the issue width of in-order superscalar microarchitecture is always limited [8].

One of the early cases to introduce OoO dispatch and execution into superscalar pipeline design is the work of Smith and Pleszkun [9]. This largely increased the performance of processors, but at the same time, exacerbated the design complexity of control logic. Along came the increase of chip area, because OoO execution needs a large amount of memory to buffer instructions and intermediate results of execution. Power consumption increases correspondingly. In Micro44 conference, Sodani [10], from Intel, gave the analysis of power consumption of a processor. Some applications adopt SIMD float point instructions to improve digital signal processing performance, and we call them compute-heavy applications. To reduce power consumption, DSPs seldom adopt float point computation. For non-compute-heavy applications, power consumption of OoO dispatch counts for 21% of the total power consumption, as shown in Figure 2. Thus, for digital signal processors, to reduce this portion of power consumption is important.

Many DSPs adopt VLIW microarchitecture [11–13]. VLIW approach has its own drawbacks, and the most obvious one is compiler dependency. As the instructions are statically allocated by the compiler, any modification to the hardware resource requires recompilation of the program.

In conclusion, to improve the performance of a processor just depending on a single method is quite difficult, especially when we expect the processor to show excellent performance both on general purpose and digital signal processing tasks.

We choose in-order superscalar and VLIW approaches for several reasons. We avoid OoO because of the problem of power consumption. Most GPPs adopt superscalar approach considering the need of compatibility. On the basis of in-order superscalar, we arrange some parts of the application to run under VLIW mode, which can further increase parallelism. By this means, we avoid complex hardware and software design and achieve high ILP of the processor.

This paper offers the following contributions.

(i) We propose *FuMicro*, a fused microarchitecture integrating both in-order superscalar and VLIW approaches. The two modes share the same pipeline and use the same ISA.

(ii) Provided with the VLIW library functions, the users can enjoy a simple and convenient development environment.

(iii) This fused microarchitecture could serve as a universal one for it can be applied to different ISAs.

2. Related Works

Since digital signal processing becomes progressively demanding in embedded applications, many processor suppliers expanded DSP capabilities on their GPPs. Eyre and Bier [14] have pointed out that, with the development of DSP, the DSP enhanced GPP cores will be the trend due to its cost, power, and area efficiency.

Hameed et al. [15] evaluated general purposed chips, and they showed that the hardware resources in general purpose chips are limited in a quantitative way. It also gave solution to solve the problem, such as exploiting ILP and DLP by means of VLIW and SIMD, providing customized instructions, and creating specific function units. These approaches are well accepted by other designers. Wong et al. [16] enhanced multimedia capabilities to the GPP by introducing multimedia instructions and expanding hardware to support them.

The researches stated above have something in common. They all need to add instructions to the original ISA. It brings the task of exploitation of binary-tool-chain. While, in our design, in-order superscalar and VLIW mode support the same ISA. For mature industrial processors such as ARM processors, our method provides convenience to the migration of microarchitecture and compatibility to the binary-tool-chain.

Few researches were conducted on pipeline architecture design to enable DSP capabilities on GPP, which might be the most fundamental approach. Though, some researches might be similar to ours in the idea of fused mode in a single core.

Lin et al. [17] propose a unified DSP working under RISC or VLIW mode, and the mode can be switched instruction by instruction. The instruction set was modified on the basis of MIPS32 ISA. They proposed hierarchical instruction encoding, which enables mode switch. When the program jumps to a VLIW packet, which has special tags to be recognized, it switches to VLIW mode. This design mainly focuses on the instruction encoding method. The hierarchical instruction encoding technology realizes mode switch and reduced VLIW code size significantly. Though the concept of fused mode might be similar, our approach is quite different. We do not focus on ISA design; on the contrary, our approach can be applied to any existing ISA, only if the ISA has possibility to be utilized to realize mode switch. Our main technique is the design of pipeline shared by the two modes.

Khubaib et al. [18] propose MorphCore which provides two modes of execution: out-of-order and in-order. This design aims to realize high performance ILP and high throughput TLP. Firstly, MorphCore is a traditional OoO core. When TLP is available, it works as a Simultaneous Multi-Threading (SMT) in-order core. Mode switch depends on the number of active threads. When the number of active threads is less than a threshold, it switches to OoO mode. The overhead of mode switch is pipeline drain and the spill or fill of architectural register state of the threads. While they realize high TLP by means of SMT, we choose the method of VLIW to increase ILP. We switch mode by instructions, which brings little overhead.

Villavieja et al. [19] propose Yoga, a hybrid microarchitecture of OoO and VLIW. Yoga remembers instruction schedules generated under OoO and transforms them into VLIW words. When it comes to the same part of the program, Yoga will arrange these VLIW words to run under VLIW mode to save power. The two approaches work under independent pipelines. The hardware design is complicated and the hardware is underutilized.

Fallin et al. [20] propose heterogeneous block architecture (HBA). The idea of fine-grained heterogeneity is quite similar with our design. It is observed that code sections in a program fit different architectures better. HBA combines heterogeneous architectures, including in-order superscalar, out-of-order superscalar, and VLIW, in one core. It uses simple heuristics to choose backends for different code sections. The difference is that we use software to decide whether the processor works under in-order superscalar or VLIW mode, while they use hardware to make this decision.

3. Design Ideas and Methods

According to pipeline design methodology, the procedure of processing one instruction is divided into several finer grained jobs. The pipelines of in-order superscalar and VLIW microarchitectures share eight common actions as listed as follows.

(1) Fetch instructions from I-Cache.

(2) Predecode instructions to get useful information.

(3) Dispatch instructions dynamically or statically.

(4) Decode the instructions to get the operands.

(5) Read operands from the register files.

(6) Execute.

(7) Access the memory.

(8) Write back the result to the register files.

Steps (1)-(2) are done before instruction dispatch and steps (4)–(8) are done after instruction dispatch. These steps are almost the same in in-order superscalar and VLIW microarchitecture. The issue width of instructions is the main difference between the two microarchitectures, which is decided at dispatch stage. VLIW may have a wider issue width because of its high instruction parallelism, while the design philosophy is of no difference with in-order superscalar.

For in-order superscalar microarchitecture, instruction parallelism detection is implemented depending on the so-called *register dependence table*. Register dependence table does the following things.

(i) Check whether the source registers of the instruction to be dispatched are ready by comparing their indexes with the destination registers' indexes recorded in the register dependence table.

(ii) If an instruction can be dispatched successfully, then record its destination registers' indexes.

(iii) When an instruction finishes execution, remove its destination registers' indexes.

For VLIW microarchitecture, instruction parallelism detection is much simpler than superscalar. In traditional VLIW instruction encoding, there are several bits indicating the parallelism of the instruction which are called the *explicit parallelism indicating bits*. By checking those bits, dispatch stage can determine how many instructions should be dispatched in one cycle. In this paper, we propose a VLIW dispatch method according to the order of function units.

When the dispatch width grows, the complexity of dispatching grows differently between two microarchitectures. For superscalar, the count of the instruction parallelism checking is proportional to the square of the dispatching width because each instruction to be dispatched should be checked with the register dependence table [21]. For VLIW, by adopting the proposed dispatch method, the count of the instruction parallelism checking is proportional to the dispatching width.

While VLIW has simpler dispatch logic, superscalar has two advantages over VLIW. One is backward compatibility, which means old programs can directly run on a new processor. Another one is dynamic cycle instruction efficiency. When a cache miss occurs, superscalar can still dispatch irrelevant instructions into the execution stages while VLIW can only stall dispatching instructions.

We are inspired to incorporate the advantages of the two dispatching patterns given that superscalar and VLIW

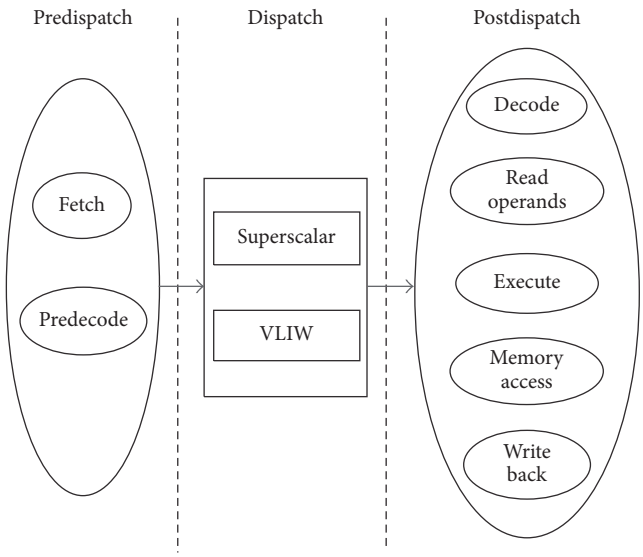

Figure 3: Concept of fused microarchitecture.

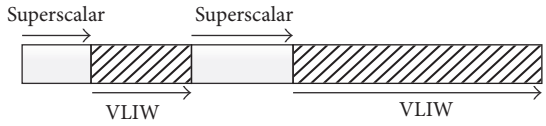

Figure 4: Program flow for fused microarchitecture.

microarchitectures have almost the same pipeline modules. Figure 3 shows the concept of fused microarchitecture.

Superscalar and VLIW present outstanding performance in different applications. Superscalar is good at handling general purpose assignments, while VLIW stands out in compute-intensive tasks. As Figure 4 shows, a program is divided into several sections and each section runs under a certain mode. For those high-parallelism demand parts of the program, we arrange them to run under the VLIW mode and the remaining part runs under superscalar mode. The dispatch stage would recognize the running mode of that section of instructions and dispatch them accordingly. By wisely allocating tasks, the performance can be greatly improved.

To realize the FuMicro based on ARM ISA, we focus on solving the following problems.

(i) *Pipeline Design*. This is the most significant part because in-order superscalar and VLIW mode should share the same pipeline and the processor should work under alternative mode.

(ii) *Mode Switch*. Since mode switch takes place at software level, mode switch method differs according to the instruction set. When we transform a superscalar machine to FuMicro architecture, we should work out method to realize mode switch with the instruction set unchanged.

(iii) *Computing Resource*. VLIW approach realizes higher parallelism and we should provide enough computational resources. As the size of register file is constrained by the instruction set, we may need to expand register file by other means in order to retain the ISA.

4. Microarchitecture

In this design, we aim to fulfil a kind of fused microarchitecture, which combines the advantages of in-order superscalar and VLIW microarchitecture and hides their drawbacks.

We aim to transform ARM in-order superscalar processor into a fused architectural one. By enabling in-order superscalar and VLIW cooperating in the same core, we expect to improve the DSP performance of the ARM GPP core noticeably.

The features of the fused microarchitecture are.

(i) The processor with fused microarchitecture works under alternative in-order superscalar and VLIW mode.

(ii) The two modes share the same pipeline and support the same ISA. The codes for the two modes exist in the same code flow.

(iii) Mode switch takes place at software level, which needs no extra hardware.

(iv) The VLIW code is provided as library function, and the user is exposed to only superscalar mode, making the development environment simple and convenient.

(v) The fused microarchitecture could serve as a universal one for it can be applied to different ISAs.

4.1. Composition. ARM Cortex-A7 [22] is a 2-issue in-order superscalar processor. There are 5 function units available, including 1 integer unit, 1 multiply unit, 1 float point unit, 1 load/store unit, and 1 branch unit. By applying FuMicro to ARM ISA, we aim to expand digital signal processing functionality on ARM general purpose cores. We name the processor as ARM-FuMicro.

In ARM-FuMicro, we design the two modes to be 2-issue in-order superscalar mode and 7-issue VLIW mode. If ARM-FuMicro works only under superscalar mode, it performs just like ARM Cortex-A7. VLIW mode provides higher parallelism and requires sufficient function units and registers. Those in ARM Cortex-A7 are not enough.

We assume that 7 function units are available in ARM-FuMicro, including 2 integer units, 2 multiply units, which can also execute integer instructions, 2 load/store units, and 1 branch unit. Integer instructions are completed in 1 cycle, multiply instruction are completed in 2 cycles, and load/store instructions are finished in 3 cycles. Branch instructions are finished in 1 cycle in superscalar mode and there are 5 branch delay slots in VLIW mode. These are shown in Table 1. For load instructions, the pipeline will be stalled if a cache miss happens, and it will take far more cycles to finish

Table 1: Composition of ARM-FuMicro.

Instruction category	Function unit number	Execution cycle
Integer	2	1
Multiply	2	2
Load/store	2	3
Branch	1	1 or 6

execution. In ARM ISA, preload mechanism is realized by the PLD instruction. When a large number of data is needed for computation, the PLD instruction can preload data from DRAM to cache to avoid cache misses.

4.2. Register File. While function units can be easily expanded, register organization is so constrained by the ISA. In ARM ISA, there are only 4 bits in an instruction encode assigned for register addressing, which implies that only 16 registers are available. Of the limited 16 registers, 4 (R0–R3) are used for parameter transfer in subroutine call and 3 (R13–R15) are used as SP, LR, and PC registers, which cannot be arbitrarily used by programmers. As a result, only 9 registers (R4–R12) can be used freely in program design, which cannot meet the requirement for high parallelism in VLIW mode. Even though ARM provides several working modes, and it is claimed to have 37 registers in total, in most occasions, especially for the programmer visible part, it works just under the user mode.

To address this problem, we expand the register file by fully utilizing the ARM ISA. Most ARM instructions support conditional execution. For those instructions, the highest 4 bits indicate the condition of execution. 0x0000-0x1110 indicate 15 conditions in all. The instructions with the highest 4 bits being 0x1111 are corresponding to the expanded ISA space, such as the NEON and Vector ISA. ARM added these instructions to explore SIMD technique on some series of ARM processors. In this design, these instructions are not supported. And we utilize this ISA space to expand the register file.

Among all the execution conditions, condition AL(0x1110) means always, and if condition is elided, the effect equals that under condition AL. In other words, an instruction is added with the AL suffix or not does not affect the encoding of this instruction. In VLIW mode of ARM-FuMicro, we modified the assembler to distinguish instructions with and without AL suffix. The instructions without AL suffix are compiled to be 0X1110, and instructions with AL suffix are compiled to be 0X1111. Since condition AL does not affect the execution of an instruction, we use condition AL to choose register file. Thus, we can expand the number of registers available.

Table 2 shows the expanded register file. R0–R15 registers are the original ARM registers. The method of expanding register file stated above is just supported under VLIW mode. R0–R3 registers are still used for parameter transfer complying with the ARM-Thumb Procedure Call Standard (ATPCS). K4–K12 compose register file S0, and T4–T12 compose register file S1. Register files S0 and S1 are the expanded registers, and they are selected by condition AL. If condition AL is set, we choose register file S0; else we choose register file S1. A code example is shown in Table 3.

Table 2: Expanded ARM register file organization.

Register	Mode		
	Superscalar	VLIW	
General purpose register		R0	
		R1	
		R2	
		R3	
	R4	K4	T4
	R5	K5	T5
	R6	K6	T6
	R7	K7	T7
	R8	K8	T8
	R9	K9	T9
	R10	K10	T10
	R11	Kll	T11
	R12	K12	T12
		R13(SP)	
		R14(LR)	
Program counter register		R15(PC)	
Status register		CPSR	

Table 3: Code example for register file selection.

Mode	Code	Registers used
Superscalar	add r6, r4, r5	R4, R5 and R6
	add.al r6, r4, r5	R4, R5 and R6
VLIW	add r6, r4, r5	K4, K5 and K6
	add.al r6, r4, r5	T4, T5 and T6

4.3. Pipeline Design. The universal pipeline structure of FuMicro microarchitecture is shown in Figure 5, and the premises of the pipeline design are listed as follows.

(i) The issue width under superscalar mode is 2.

(ii) The issue width under VLIW mode is 7.

(iii) The size of the instruction is 16 bits or 32 bits.

(iv) The size of the Instruction Fetching Packet (IFP) is 256 bits, to meet the demand of VLIW mode.

The universal pipeline structure of FuMicro microarchitecture consists of the following stages.

(i) *Stage 1: PCG (Program Counter Generate Stage)*. The PCG stage generates the next PC of the program. The next PC is selected from the current PC + 4, the branch target, or the entry point of the interrupt service routine.

(ii) *Stage 2: PCS (Program Counter Send Stage)*. This stage simply sends the PC address generated by the PCG stage.

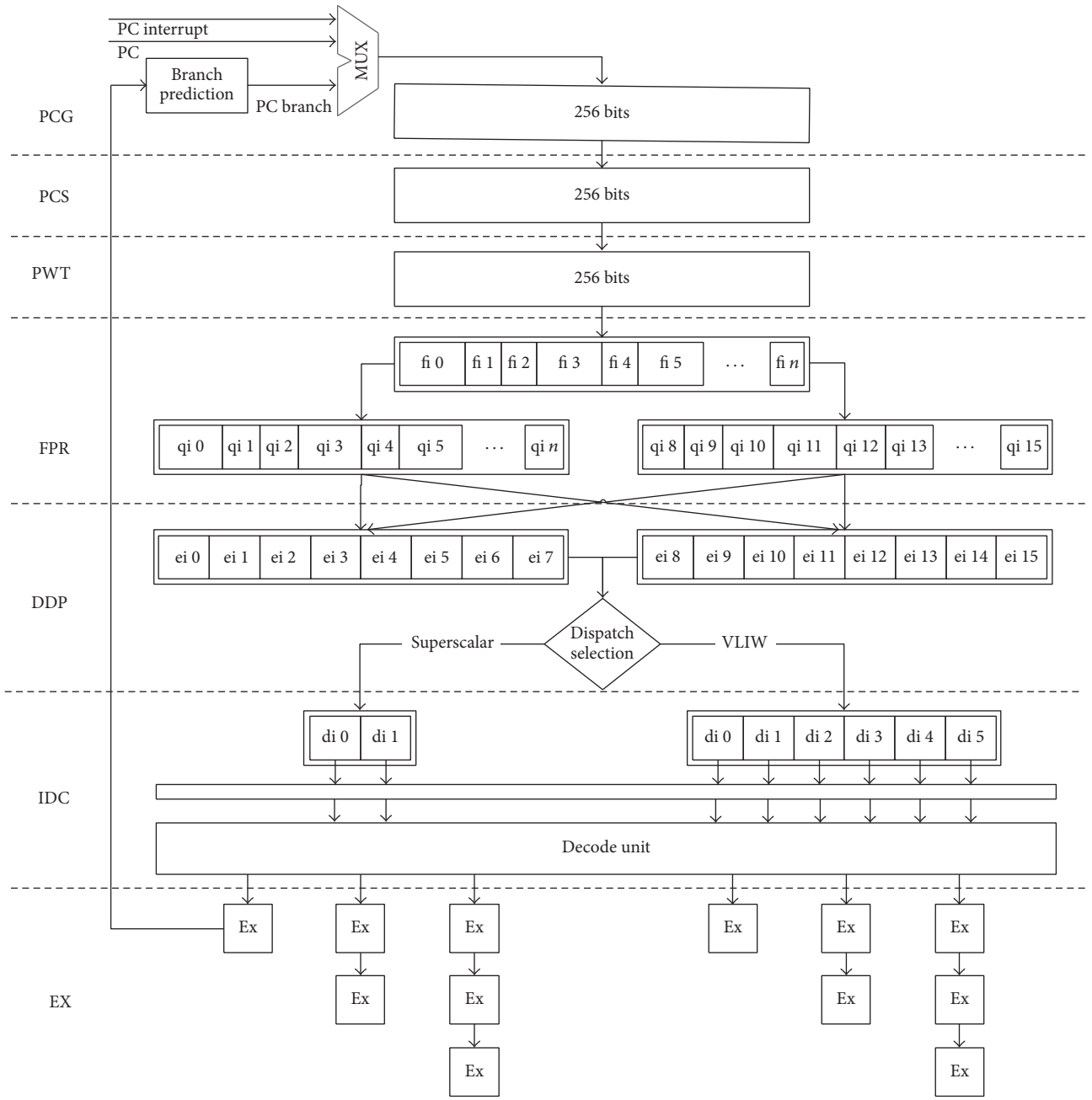

Figure 5: Universal pipeline structure for fused microarchitecture.

(iii) *Stage 3: PWT (Processor Wait Stage)*. Judge whether the instruction is valid in the instruction cache. If not, stall the pipeline and fetch the instruction from the instruction RAM.

(iv) *Stage 4: FPR (Fetch-Packet Receive Stage)*. In the FPR stage, we use two 256-bit fetch registers to mimic an instruction fetch queue. Once a fetch register is empty, instruction fetch will occur. FPR stage should also implement instruction expansion. Because we want all the instructions sent to the following stages have the same length.

(v) *Stage 5: DDP (Dynamic Dispatch Stage)*. In the DDP stage, we use two 192-bit expanding registers to mimic an instruction expanding queue. Once an expanding register is empty, instruction expand will occur.

Instruction parallelism detection is based on the expanding queue. When the CPU is under superscalar mode, register dependence table is used for dynamic instruction dispatching. When the CPU is under VLIW mode, the order of function units is checked for static instruction dispatching.

Figure 6: Structure of ARM CPSR.

Table 4: ASM code for mode switch.

(a)

Switch from superscalar to VLIW	
mrs	r2, cpsr
ldr	r1, =#0x800000
orr	r2, r2, r1
msr	cpsr, r2

(b)

Switch from VLIW to superscalar	
mrs	r3, cpsr
bic	r3, 0x800000
msr	cpsr, r3

All the instructions to be dispatched are put in the dispatching registers. There are only one dispatching register in the decode stage.

(vi) *Stage 6: IDC (Instruction Decode Stage)*. The decode stage just decodes all the instructions.

(vii) *Stages 7–9: EX (Instruction Execution Stages)*. In the execution stages, superscalar mode and VLIW mode have the same behavior. Different instructions need different cycles to finish execution.

4.4. Mode Switch. In order to ensure complete compatibility with ARM ISA, we realize mode switch by utilizing ARM instructions. There is a register called Current Program Status Register (CPSR) in ARM. It is a 32-bit register recording the information of execution condition bits, instruction set, ARM processor mode, and so on. There are several reserved bits we can utilize. As it is described in ARMv7-A architecture, CPSR has 4 reserved bits, and we take one as the mode switch flag, as shown in Figure 6. By setting and clearing this bit, we switch from one mode to the other. CPSR can be modified by instructions. Modification of CPSR should comply with the principle of copy-modify-write back, and direct modification to the CPSR is not allowed. The ASM code for mode switch is shown in Table 4. When mode switches from superscalar to VLIW, we set the MS bit as 1. When MS bit is set as 0, mode switches from VLIW to superscalar.

In conclusion, mode switch is totally decided by software. The programmers can switch the mode from one to the other when they want to. Generally, we switch to VLIW mode when the code section can provide high ILP. What the hardware needs to do is to check the MS bit each cycle. When MS bit is 1, the current mode is VLIW, and it should dispatch instructions according to the principle we are going to describe in the next part. When MS bit is 0, the current mode will be in-order superscalar, and the processor should check register dependence table to dispatch instructions. Thus, we need quite simple logic to realize mode switch, and the overhead of mode switch mainly comes from executing the mode switch instructions, as shown in Table 4. Since these instructions are irrelevant to the program function, they bring about 3-cycle performance loss. But the performance improvement brought by VLIW can easily make up for this loss.

4.5. VLIW Dispatch Method. Under VLIW mode, we dispatch instructions according to the order of function units. ARM-FuMicro has 7 function units, which can be classified into four categories: integer unit (A), multiply unit (M), load/store unit (L), and branch unit (B). The order of the functions is listed as follows. Instructions with ascending order can be dispatched in one cycle.

$$A < M < L < B. \quad (1)$$

The order is defined based on the fact that, in a program, most instructions are arithmetic instructions and the number of branch instructions is the smallest. In other words, this is the order easiest to meet, and more instructions can be dispatched according to this order. As the VLIW code is hand written, the programmer should ensure that there is no dependency between the instructions to be dispatched in one cycle.

The code examples will be given in the next chapter, to help understand this principle.

4.6. Branch Prediction. Branch prediction is applied in superscalar mode. In VLIW mode, we use delayed branch, and each branch instruction has 5 delay slots. So when we write VLIW code, we have to make use of the delay slots to keep the pipeline running. The delayed branch is very effective. Since we always arrange compute-intensive code sections to run under VLIW mode, and such code sections might be performed for thousands of times in one program, so the branch in such code sections is more likely to be taken. Thus, the performance loss in VLIW mode caused by branch is negligible.

Next, we introduce the branch prediction method applied in superscalar mode. In the fused microarchitecture, we should not still use the same method as in pure superscalar mode [23] to deal with branch instructions, mainly because of the concept of IFP. At fetch stage we use two 256-bit fetch registers to mimic an instruction fetch queue, and each is called an IFP.

Figure 7 shows a normal case. B indicates the branch instruction and T indicates the branch target. We call the IFP containing branch instruction the first IFP and we call the next IFP containing the branch target instruction the second IFP. When it comes to the branch instruction, the post-B instructions in the first IFP and the pre-T instructions in the second IFP are all flushed. If the branch prediction is correct, we just do nothing. While the branch prediction is incorrect, we flush all the pipelines before the execution stage.

A special case is also demonstrated in Figure 7. Because of the variable length of the instructions, a 32-bit branch instruction can be partitioned into two IFP, with the 16-bit in the first IFP and 16-bit in the second IFP. This will cause a

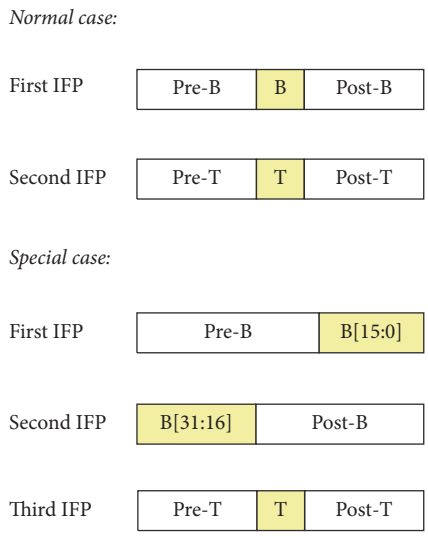

Figure 7: Branch prediction processing flow.

bubble in the pipeline because the post-B instructions in the second IFP will all be flushed. In this case, even if the branch prediction is correct, we will still loss a cycle.

5. Software Design

In this part, we will introduce how we choose the code section to be executed under VLIW mode. Then, we introduce VLIW code design method, to help understand VLIW dispatch method and delayed branch in VLIW mode. We also explain how we provide VLIW library functions to users.

5.1. Code Division. VLIW execution can greatly improve the execution efficiency. To decide whether a code section is suitable to be executed under VLIW mode, we consider whether there are enough computational operations in one basic block and we hope the dependency between them is as little as possible. The fundamental goal is to fully utilize the computational resources.

Figure 8 shows two examples of how we divide the code into two modes. Figure 8(a) is a FFT program. In this program, the subfunction fft_bit_reduct fulfils iteration of the butterfly operation, which in the main operation in fft function. Since this code section is computed intensively and the ILP is relatively high, we arrange this code section to run under VLIW mode. Figure 8(b) shows the mpeg2decode programs. Mpeg2decode programs convert MPEG-2 video bitstreams into uncompressed video. Mpeg2decode includes several subfunctions, of which idct function takes large part of computation task, since it is subsubfuction of many other subfunctions. When mpeg2decode programs run under superscalar mode of FuMicro, idct subfunction takes 28.3% of the total runtime. While the code size of idct takes just 3.7% of the total. We arrange the idct subfunction to run under VLIW mode, and the rest parts of the programs run under superscalar mode. After rewrite the idct code into VLIW pattern; the total code size is just expanded by 0.37%.

Table 5: ASM code example. (a) is a ASM code section run under superscalar mode. (b) is the corresponding VLIW code.

(a)

Cycle	Number	Instruction		Function unit
1	1	mov	r7, r7, asr #8	A
	2	mov	r2, r2, asl #11	A
2	3	add	r2, r2, #128	A
3	4	add	lr, fp, r2	A
	5	rsb	fp, fp, r2	A
4	6	ldr	r2, [sl, r5]	L
	7	add	r5, lr, r4	A
5	8	add	r0, fp, r6	A
	9	rsb	r4, r4, lr	A
6	10	mul	r6, r6, fp	M
	11	mul	ip, r9, r4	M
7	12	add	sl, r8, r5	A
	13	rsb	r9, r9, r4	A
8	14	add	r2, r0, rl	A
	15	add	lr, r6, r7	A
9	16	rsb	r8, r8, r5	A
	17	rsb	rl, rl, r0	A
10	18	rsb	r6, r7, r6	A
	19	ldr	r4, [sp, #12]	L

(b)

Cycle	Number	Instruction		Function unit
1	1	mov	r7, r7, asr #8	A
	2	mov	r2, r2, asl #11	A
	6	ldr	r2, [sl, r5]	L
2	3	add	r2, r2, #128	A
3	4	add	lr, fp, r2	A
	5	rsb	fp, fp, r2	A
4	7	add	r5, lr, r4	A
	8	add	r0, fp, r6	A
	9	rsb	r4, r4, lr	A
	10	mul	r6, r6, fp	M
	19	ldr	r4, [sp, #12]	L
5	12	add	sl, r8, r5	A
	13	rsb	r9, r9, r4	A
	14	add	r2, r0, rl	A
	11	mul	ip, r9, r4	M
6	15	add	lr, r6, r7	A
	16	rsb	r8, r8, r5	A
	17	rsb	rl, rl, r0	A
	18	rsb	r6, r7, r6	A

5.2. VLIW Code Design. As for the VLIW code design method, We firstly consider code sections without branch instructions.

Table 5(a) shows a code section run under superscalar mode. The total runtime is 10 cycles. Table 5(b) shows how

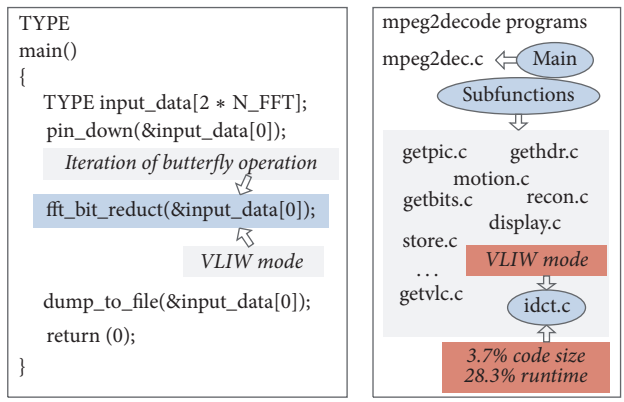

Figure 8: Code division. (a)It is a FFT program. (b) It shows the mpeg2decode programs.

we arrange the same code section to run under VLIW mode, and the runtime is reduced to 6 cycles.

Next, we explain the details of how we design VLIW code. We name the instructions dispatched in cycle n as dispatch package n. As we can see, the instructions dispatched in one cycle all comply with the order of A, M, and L. Compared with the original code as shown in Table 5(a), we simply reorder the instructions to satisfy the dispatch principle of VLIW. But in this process, we should make sure that every instruction will be executed correctly.

For example, instruction number 6 is brought forward to cycle 1. Because the execution of number 6 does not depend on the execution result of numbers 1–5 and when it is brought forward, it does not affect the execution of numbers 3–5, it is notable that instruction number 6 is before numbers 3–5 in VLIW code. Numbers 3–5 use the value of r2, and number 6 uses load number to r2. As loas/store instructions are finished in 3 cycles, instruction number 6 is finished at the end of cycle 3. So the value of r2 is changed by number 6 until the end of cycle 3, which means instruction numbers 3–5 use the correct value of r2. Similarly, instruction No. 19 is brought forward to cycle 4. In this way, instruction number 19 will finish execution at the end of cycle 6.

Next, we consider code sections with branch instructions. As described before, there are 5 delay slots for branch instructions in VLIW mode. We assume the ASM code given in Table 5(a) is in a loop and at the end of the loop is a branch instruction, as shown in Box 1. The corresponding VLIW code is given in Box 2. We bring the branch instruction 5 cycles ahead; thus the delay slots can be fed with useful instructions. The premise is that the condition of branch is produced before the branch instructions; otherwise, we need to relocate the branch label and the branch instruction at the same time.

5.3. VLIW Library Function. We can provide the users with VLIW library functions, so they can enjoy a convenient development environment.

Take the Mpeg2decode program, for example. This program includes many subfunctions, among which idct takes the largest amount of computation tasks. We rewrite the idct ASM code into VLIW pattern and compile it as static library. The users can use this library but do not need to know how to design VLIW code or how to switch mode.

```
loop:
        instruction No. 1
        ...
        instruction No. 19
        b    loop
```

Box 1: ASM code example with branch instruction under VLIW mode. A code example with branch instruction. It is transformed from the code given as follows.

```
loop:
        dispatch package 1
        b    loop
        dispatch package 2
        ...
        dispatch package 6
```

Box 2: ASM code example for branch under VLIW mode. The corresponding VLIW code.

The process is shown in Figure 9. We first compile idct.c into ASM and get idct.s. Then we rewrite idct.s into VLIW pattern. The VLIW code is compiled as static library. The function of idct is called by main function. For the users, what they need to do is just to compile the main function together with the given VLIW library.

6. Evaluation

This architecture is simulated on gem5 simulator [24]. gem5 is a cycle accurate simulation platform. It support many ISAs,

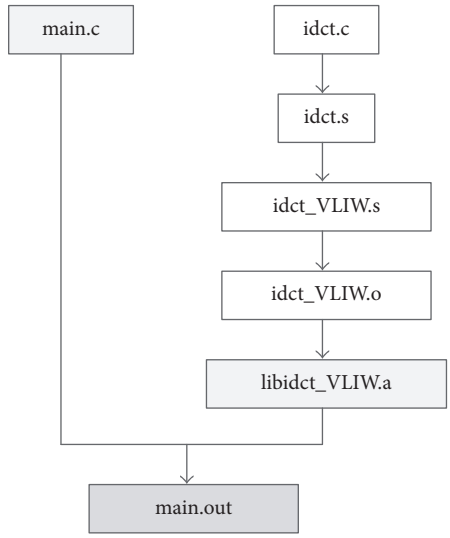

Figure 9: How to provide static VLIW library to the user.

such as Alpha, ARM, MIPS, and Power. It also has several CPU models, including AtomicSimpleCPU, TimingSimpleCPU, inorderCPU, and O3CPU. AtomicSimpleCPU is the most recommended model for self-designed architecture simulation [25]. We realize the fused microarchitecture based on TimingSimpleCPU model, and the target ISA is ARM ISA. TimingSimpleCPU model is similar to AtomicSimpleCPU, but it uses timing memory accesses, which is more accurate.

To provide a comprehensive evaluation of FuMicro, we choose various benchmarks, including Dhrystone2.1 [26], CoreMark1.0, DSPStone [27], EEMBC [28] telecom, and mpeg2decode [29].

Dhrystone and CoreMark benchmarks are used to evaluate the general purpose capability of a processor. DSPStone and EEMBC telecom are widely used benchmarks, which emphasize more the evaluation of digital signal processing ability. Mpeg2decode programs convert MPEG-2 video bitstreams into uncompressed video, and it also requires digital signal processing ability.

All the benchmarks are written in C code. The programs running under VLIW mode are mainly developed by hand coding. We firstly compile the C code into assembling language, from which we pick the code sections that are most suitable to be executed in VLIW mode and rewrite the code into VLIW pattern. At the beginning and ending of the VLIW section we insert instructions to realize mode switch.

ARM compiler is gcc linaro-4.7-2013.06-1.

6.1. Synthesis Result. FuMicro is designed initially based on LILY2 ISA. The predecessor of LILY2 is called LILY [30], which is an independently designed VLIW DSP. We introduced the in-order superscalar on the basis of the VLIW microarchitecture. When the fused microarchitecture works, we consider to adopt FuMicro to ARM ISA.

Since ARM-FuMicro hardware is still under working, we can only give the benchmark evaluation result based on the simulator. LILY and LILY2 are both synthesized based on Xilinx FPGA and the results are shown below. FPGA is ZYNQ device XC7Z045 FFG900 and synthesis tool is Vivado. The synthesis result is given in Table 6.

Table 6: Slice logic synthesis result (utilization rate).

Site type	LILY (VLIW)	LILY2 (FuMicro)	Gain
Slice LUTs	28.22	30.26	7.23%
Slice registers	3.82	4.38	14.66%

Table 7: Dhrystone and CoreMark result (DMIPS/MHz).

Benchmark	ARM-FuMicro superscalar
Dhrystone	1.93
CoreMark	3.14

Table 8: EEMBC result (cycle).

Function	Superscalar mode	Fused mode	Gain
fft	35675	29758	16.6%
autcor	621267	327363	47.3%
conven	182378	109674	39.9%
fbital	187641	243707	−29.9%
viterb	212388	210962	0.67%

The result shows that when transforming the VLIW microarchitecture into the fused microarchitecture, hardware consumption is rather small. LILY2 requires more memory space for branch prediction, and the size of memory depends on the Branch Target Buffer (BTB) table item.

6.2. Superscalar Performance Evaluation. We evaluate the general purpose performance of ARM-FuMicro processor by Dhrystone and CoreMark. The Dhrystone result is 1.93 DMIPS/MHz and the CoreMark result is 3.14 DMIPS/MHz, as shown in Table 7. For general purpose evaluation, ARM-FuMicro processor works just under superscalar mode. The general purpose performance of ARM-FuMicro is similar to ARM Cortex-A7 processor, of which the Dhrystone result is 1.9 DMIPS/MHz.

6.3. Fused Mode Performance Evaluation. We evaluate the digital signal processing ability of ARM-FuMicro processor by EEMBC telecom and DSPStone benchmarks. The result is shown in Tables 8 and 9. For EEMBC benchmarks, when the processor works under alternative fused mode, the performance can be improved on an average of 14.9%, compared with that under pure superscalar mode. It also shows similar or even worse performance when handling viterb and fbital functions. For DSPStone benchmarks, most programs show better performance with an average improvement of 8.2%. There are also some programs that show worse performances.

We also evaluate the image decompression processing ability of ARM-FuMicro processor and the result is shown in Table 10. We arrange the idct function to run under VLIW mode, and the remaining parts of the programs run under superscalar mode. The performance of mpeg2decode

Table 9: DSPStone result (cycle).

Function	Superscalar mode	Fused mode	Gain
dot_product	39	30	23.1%
convolution	162	131	19.1%
lms	439	284	35.3%
iir_biquad_one_section	22	30	−36.4%
iir_biquad_*N*_sections	74	61	17.6%
real_update	22	24	−9.1%
*n*_real_updates	75	62	17.3%
matrix	2178	2030	6.8%
matrix 1 × 3	31	33	−6.4%
fft	1993	1698	14.8%
complex_update	48	40	16.7%
*n*_complex_updates	658	450	31.6%
fir	220	250	−13.6%
fir2dim	530	494	6.8%
complex_multiply	450020	450020	0%

is improved by 10.0%, and the total code size is expanded just by 0.37%, as shown in Table 11.

6.4. Results Analysis. For ARM-FuMicro processor, the general purpose performance is close to ARM Cortex-A7 processor, which is what we expected.

The objective of FuMicro microarchitecture is exploiting digital signal processing ability on general purpose cores. For ARM-FuMicro processor, the performance of most EEMBC and DSPStone functions can be improved. While for some functions, the results remain similar or even become worse.

Two main factors leading to possibly performance degradation are inserting mode switch instructions and delayed branch in VLIW mode. When the performance improvement brought by VLIW can not cover the performance loss caused by executing mode switch instructions, or the VLIW branch delay slots can not be efficiently utilized, the overall performance will decrease. So this result can be partly attributed to the nature of the programs. Through analysis of the programs, we find that the programs, of which the results under fused mode are worse than that under superscalar mode, are among the following categories.

(i) The program is short with few operations.

(ii) There are multilayer nested loops in the program and in each loop, there are few operations.

(iii) The programs provide limited parallelism. High dependency exists between conjoint instructions.

EEMBC and DSPStone both include a few small benchmarks. Some of them are too simple, and the computation amount is quite small. Just considering the basic block, Table 5 shows that we reduce the execution time of a 19-instruction basic block form 10 cycles to 6 cycles. When the basic block is even smaller, the performance improvement brought by VLIW is quite limited. What is more, we have to insert mode switch instructions, and it will take a few cycles. If the disadvantage of mode switch overweights the advantage of VLIW, the performance decreases. Boxes 1 and 2 show an ideal case of branch instruction. The branch target is the basic block itself, and the condition of branch is ready 5 cycles before the branch instruction. When the condition, normally the result of a compare instruction, is ready just before the branch instruction, we can not bring the branch instruction forward. When the branch target is redirected to other basic blocks, it makes the situation even more complex. Then it is difficult to insert useful instructions into the branch delay slots, and this may case several cycles of performance loss.

Table 10: Mpeg2decode result (cycle).

Function	Superscalar mode	Fused mode	Gain
Mpeg2decode	8627001	7762105	10.0%

Table 11: Mpeg2decode code size comparison (byte).

Mpeg2decode	idct superscalar code	idct VLIW code
80332	2664	2960

These programs are not suitable to be executed in VLIW mode in nature. So the results of EEMBC and DSPStone, at the same time, reflect the flexibility of FuMicro microarchitecture. For the programs that are suitable to be executed in VLIW, the performance can be improved up to 47.3%. Mpeg2decode is a relatively large benchmark. Among the mpeg2decode programs, idct is the most suitable one to run under VLIW mode. The result shows that the performance is improved by 10.0% with just 0.37% of code size expansion. The results show the potential capability of FuMicro in digital signal processing filed.

7. Conclusion

In this paper, we propose a universal microarchitecture named as FuMicro, aiming to realize GPP+DSP capability in a single core. Our approach is to allow the processor working under alternative in-order superscalar and VLIW mode using the same pipeline and the same ISA. Superscalar approach is suitable for flow control tasks, while VLIW approach is advantageous in processing DSP applications. Wise assignment of the tasks can bring improvement to the performance of the processor.

FuMicro is applied to ARM ISA. The evaluation of ARM-FuMicro shows that, for the programs including sections that are suitable to be executed in VLIW mode, the result can be improved to a large extend. Such programs are characterized by high parallelism and large amount of computational operations.

It is a trend for many systems to use ARM+DSP cores, indicating the need of both general flow control capability and data processing ability simultaneously, and FuMicro microarchitecture is promising to take place of the ARM+DSP architecture to satisfy all these needs in a single core.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported by the Core Electronic Devices, High-End General Purpose Processor, and Fundamental System Software of China under Grant no. 2012ZX01034-001-002, the National Natural Science Foundation of China under Grants no. 61201182 and no. 61502032, Tsinghua National Laboratory for Information Science and Technology (TNList), and Samsung Tsinghua Joint Laboratory.

References

[1] H.-P. Brueckner, M. Wielage, and H. Blume, "Intuitive and interactive movement sonification on a heterogeneous RISC/DSP platform," in *Proceedings of the 18th Annual International Conference on Auditory Display (ICAD '12)*, pp. 75–82, 2012.

[2] L. Codrescu, W. Anderson, S. Venkumanhanti et al., "Hexagon DSP: an architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, 2014.

[3] TI. Multicore dsp+arm keystone ii system-on-chip (soc), 2013.

[4] Freescale, Frescale Official Website, 2014.

[5] TI, "Linux EZ Software Development Kit (EZSDK) for DaVinci(TM) DM814x and DM816x Video Processors," 2014.

[6] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05)*, pp. 408–419, Madison, Wis, USA, June 2005.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Elsevier, 2012.

[8] G. Steven, B. Christianson, R. Collins, R. Potter, and F. Steven, "A superscalar architecture to exploit instruction level parallelism," *Microprocessors and Microsystems*, vol. 20, no. 7, pp. 391–400, 1997.

[9] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA '85)*, pp. 36–44, ACM, Boston, Mass, USA, 1985.

[10] A. Sodani, "Race to exascale: opportunities and challenges," in *Proceedings of the Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, Porto Alegre, Brazil, December 2011.

[11] N. Seshan and W. Sites, "High velociTI processing," *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 86–101, 1998.

[12] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers et al., "TriMedia CPU64 architecture," in *Proceedings of the International Conference on Computer Design (ICCD '99)*, pp. 586–592, IEEE, October 1999.

[13] T. Kumura, M. Ikekawa, M. Yoshida, and I. Kuroda, "VLIW DSP for mobile applications," *IEEE Signal Processing Magazine*, vol. 19, no. 4, pp. 10–21, 2002.

[14] J. Eyre and J. Bier, "Evolution of DSP processors," *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 43–51, 2000.

[15] R. Hameed, W. Qadeer, M. Wachs et al., "Understanding sources of inefficiency in general-purpose chips," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 37–47, 2010.

[16] S. Wong, S. Cotofana, and S. Vassiliadis, "Multimedia enhanced general-purpose processors," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME '00)*, vol. 3, pp. 1493–1496, IEEE, New York, NY, USA, August 2000.

[17] T.-J. Lin, C.-M. Chao, C.-H. Liu et al., "A unified processor architecture for RISC & VLIW DSP," in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, pp. 50–55, ACM, 2005.

[18] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "MorphCore: an energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *Proceedings of the IEEE/ACM 45th International Symposium on Microarchitecture (MICRO '12)*, pp. 305–316, December 2012.

[19] C. Villavieja, J. A. Joao, R. Miftakhutdinov, and Y. N. Patt, Yoga: A hybrid dynamic VLIW/OoO processor, 2014.

[20] C. Fallin, C. Wilkerson, and O. Mutlu, "The heterogeneous block architecture," in *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*, pp. 386–393, Seoul, South Korea, October 2014.

[21] S. Cotofana and S. Vassiliadis, "On the design complexity of the issue logic of superscalar machines," in *Proceedings of the 24th Euromicro Conference*, pp. 277–284, 1998.

[22] P. Greenhalgh, Big. LITTLE processing with ARM Cortex-A15 & Cortex-A7, September 2011.

[23] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*, vol. 29, pp. 135–148, May 1981.

[24] N. Binkert, B. Beckmann, G. Black et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[25] gem5 official website, http://www.gem5.org.

[26] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[27] V. Živojinović, J. M. Velarde, C. Schläger, and H. Meyr, "Dspstone: a dsp-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 715–720, 1994.

[28] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the EEMBC benchmark suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, 2009.

[29] T. Y. Chung and Y. N. Oh, MPEG2 moving picture encoding/decoding system: US, US6310962, 2001.

[30] Z. Shen, H. He, X. Yang, D. Jia, and Y. Sun, "Architecture design of a variable length instruction Set VLIW DSP," *Tsinghua Science and Technology*, vol. 14, no. 5, pp. 561–569, 2009.