A Unified Processor Architecture for RISC & VLIW DSP

Tay-Jyi Lin, Chie-Min Chao, Chia-Hsien Liu, Pi-Chen Hsiao,
Shin-Kai Chen, Li-Chun Lin, Chih-Wei Liu, and Chein-Wei Jen
Department of Electronics Engineering
National Chiao Tung University, Taiwan

ABSTRACT
This paper presents a unified processor core with two operation modes. The processor core works as a compiler-friendly MIPS-like core in the RISC mode, and it is a 4-way VLIW in its DSP mode, which has ***distributed and ping-pong register organization*** optimized for stream processing. To minimize hardware, the DSP mode has no control construct for program flow, while the data manipulation RISC instructions are executed in the DSP datapath. Moreover, the two operation modes can be changed instruction by instruction within a single program stream via the ***hierarchical instruction encoding***, which also helps to reduce the VLIW code sizes significantly. The processor has been implemented in the UMC 0.18um CMOS technology, and its core size is 3.23mm ×3.23mm including the 32KB on-chip memory. It can operate at 208MHz while consuming 380.6mW average power.

Categories and Subject Descriptors
C.1 Processor Architectures

General Terms
Design

Keywords
Digital signal processor, dual-core processor, variable-length instruction encoding, register organization

1. INTRODUCTION
The computing tasks of an embedded multimedia system can be roughly categorized into control-oriented and data-dominated, and most computing platforms for media processing contain at least two processors – a RISC and a DSP to handle the specific tasks accordingly [1][2]. Fig. 1 shows a dual-core processor example. The RISC coordinates the system and performs some reactive tasks such as the user interface. In the meanwhile, the DSP core performs transformational tasks with more deterministic and regular behaviors, such as the small and well-defined workloads in signal processing applications. Recent RISC architectures have been enhanced for data-intensive tasks by incorporating single-cycle multiply-accumulators, SIMD (MMX-like) datapaths, or specific functional units [3] to reduce the needs for an additional core, but the performance is still far behind that of a DSP with similar computing resources [4]. This is because data-intensive tasks are very distinct from general-purpose computations.

This work was supported by the National Science Council, Taiwan under Grant NSC93-2220-E-009-034

GLSVLSI'05, April 17-19, 2005, Chicago, Illinois, USA.

This paper presents a unified processor architecture for RISC and DSP. The processor core functions as a MIPS-like RISC in its scalar/program control mode, and it becomes a powerful DSP while switched into its VLIW/data streaming mode. To maximize the utilization, most hardware resources are shared between these two modes. Moreover, the VLIW/data streaming mode has no control construct for program flow, while the data manipulation RISC instructions are executed in the VLIW datapath. The two modes can be changed instruction by instruction within a single program stream via our novel hierarchical instruction encoding, which also helps to reduce the VLIW code sizes significantly.

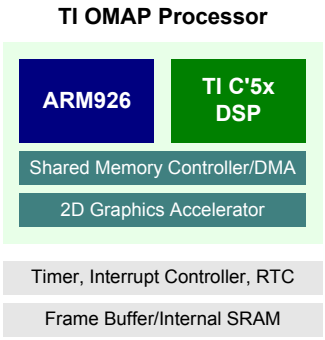

Fig. 1 Dual-core processor [2]

The rest of this paper is organized as follows. Section 2 first reviews the architecture of our proprietary DSP with the novel distributed and ping-pong register organization. What follows is the unified datapath for the DSP and a MIPS-like RISC. Section 3 then describes the hierarchical instruction encoding that blends the RISC and the DSP instructions into a single stream and enables the instruction-by-instruction mode switching. Moreover, the variable-length encoding scheme significantly reduces the code sizes of VLIW programs. The simulation results and our silicon implementation are available in Section 4. Section 5 concludes this paper and outlines our future works.

2. DATAPATH DESIGN

2.1 VLIW DSP Datapath
Today's media processing demands extremely high computations with real-time constraints in audio, image or video applications. Instruction parallelism is exploited to speedup high-performance microprocessors. Compared to dynamically hardware-scheduled superscalar processors, VLIW machines [5] have the low-cost compiler scheduling with deterministic execution times, and thus they become the trends of high-performance DSP processors. But the complexity of the register file grows rapidly as more and more functional units (FU) are integrated on a chip, which concurrently operate to achieve the performance requirements. A centralized register file provides storage for and interconnects to each FU in a general manner and each FU can access any register location. For

N concurrent FU, the area of the centralized register file grows as N^3, the delay as $N^{3/2}$, and the power dissipation as N^3 [6]. Thus, the register file will soon dominate the area, the delay, and the power dissipation in the multi-issue processors as the number of FU increases. To solve this problem, the communications between FU can be restricted by partitioning the register file to significantly reduce the complexity with some performance penalty [7-13]. In other words, each FU can only read and write a limited subset of registers.

2.1.1 Distributed and ping-pong register file

Fig. 2 shows the proposed distributed and ping-pong register file for two FU – a load/store (LS) and an arithmetic unit (AU). The 32 registers are divided into four independent groups, and each group has access ports only for a single FU (two read and two write ports in our case). The eight address registers (a_0~a_7) and the eight accumulators (ac_0~ac_7) are dedicated to the LS and the AU respectively, and they are not visible to any other FU. The remnant 16 registers are shared between the two FU, and they are divided into 'ping' and 'pong' register groups. When the LS access the ping, the AU can only access the pong, and vice versa. In other words, the data registers are partitioned into two banks with exclusive accesses. In summary, each FU has 16 registers – 8 registers are private and 8 registers are dynamic mapped (to be either ping or pong). The mapping is explicitly specified by the programmers in each VLIW packet as described in Section 2.1.2.

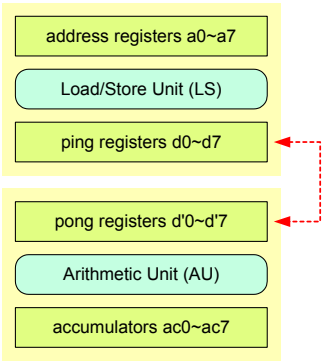

Fig. 2 Distributed & ping-pong register organization

Our DSP supports very powerful SIMD instructions based on the distributed and ping-pong register file. For example, the double load (store) instruction of the form –

`dlw` r_m`,(`r_i`)+k,(`r_j`)+l`.

It performs two memory accesses concurrently (r_m←`Mem[`r_i`]`, r_{m+1}←`Mem[`r_j`]`) and simultaneous address updates (r_i←r_i+k, and r_j←r_j+l). The index `m` must be an even number with `m+1` implicitly specified. The double load/store instructions require six concurrent register file accesses (including two reads and four writes for `dlw`, or four reads and two writes for `dsw`). They do not cause access conflicts, because r_i and r_j are private address registers while r_m and r_{m+1} are ping-pong registers that deliver data to the AU. These registers are implemented in independent banks. The AU supports 16-bit SIMD full-precision multiply-accumulate operations with two 40-bit accumulators:

`fmac.d` r_i, r_m, r_n.

It performs r_i←r_i+r_m`.H`×r_n`.H`, and r_{i+1}←r_{i+1}+r_m`.L`×r_n`.L` in parallel. Similarly, the index `i` must be even with `i+1` implicitly specified. This SIMD instruction needs six register file accesses concurrently (four reads and two writes respectively).

2.1.2 Assembly programming

The syntax of the DSP assembly codes starts with the ping-pong index, followed by the instructions for each issue slot in sequence:

`ping-pong index;` i_0`;` i_1`;`.

The following is an illustrating example of a 64-tap finite-impulse response (FIR) filter that produces 1,024 outputs. Assume there is no delay slot (such as an ALU operation immediately after a load in the classical 5-stage pipeline [5]) for simplicity. The memory is byte addressable and the input and output data are 16-bit and 32-bit numbers respectively.

```
1  0; li a0,coef;              li ac0,0;
2  0; li a1,X;                 nop;
3  0; li a2,Y;                 nop;
4     rpt 1024,8;
5  0; dlw d0,(a0)+4,(a1)+4;    li ac1,0;
6     rpt 15,2;
7  1; dlw d0,(a0)+4,(a1)+4;    fmac.d ac0,d0,d1;
8  0; dlw d0,(a0)+4,(a1)+4;    fmac.d ac0,d0,d1;
9  1; dlw d0,(a0)+4,(a1)+4;    fmac.d ac0,d0,d1;
10 0; li a0,coef;              fmac.d ac0,d0,d1;
11 0; addi a1,a1,-126;         add d0,ac0,ac1;
12 1; sw (a2)+4,d0;            li ac0,0;
```

The zero-overhead looping instructions (`RPT` at line 4 and line 6) are carried out in the instruction dispatcher and do not consume any execution cycle of the datapath. The inner loop (line 7-8) loads two 16-bit inputs and two 16-bit coefficients into the 32-bit d_0 and the 32-bit d_1 with the SIMD load operations (i.e. d_0←`Mem`$_{32}$`[`a_0`]` and d_1←`Mem`$_{32}$`[`a_1`]`), and the address registers a_0 and a_1 are updated simultaneously (i.e. a_0←a_0+4 and a_1←a_1+4). In the meanwhile, the AU performs two 16-bit MAC for two taps concurrently (i.e. ac_0←ac_0+d_0`.H`×d_1`.H` and ac_1←ac_1+d_0`.L`×d_1`.L`). After accumulating 32 32-bit products respectively with two 40-bit accumulators, ac_0 are ac_1 are added together and rounded back to the 32-bit d_0 in the ping-pong registers. Finally, the 32-bit output is stored in the memory by the LS via d_0. In this FIR example, an output requires 35 cycles or the DSP can compute 1.83 taps per cycle. Note that, the loops can be unrolled to achieve similar performance easily if the load slots are taken into account. By the way, the ping-pong register organization helps to reduce the code size, for the programmer can specify different register locations (i.e. to be either in ping or pong) with the same name via the ping-pong index. We will make use of this property in the instruction encoding later in Section 3.

2.2 Unified RISC & VLIW DSP

Each register group in Fig. 2 is actually implemented with two 2R/1W (two reads and 1 write) register banks of half sizes (i.e. 4 registers) as shown in the VLIW block in Fig.3, instead of 2R/2W access ports. Since a RISC core does not contain any data manipulation instruction other than those performed on the LS (i.e. load/store instructions) or the AU (i.e. ALU instructions), the extra resource for the DSP datapath to execute RISC instructions is an additional register file. To simplify data exchange between the two modes, the ping-pong data registers are aliased as the 'saved' (s_0~s_7) and the 'temporary' (t_0~t_7) data registers in the MIPS-compatible scalar/program control mode. The remnant 16

registers are private to the scalar mode (actually, there are only 15 physical registers, for r_0 is hardwired to zero [5]). Fig. 3 shows the unified datapath for the two modes with total nine 2R/1W register banks.

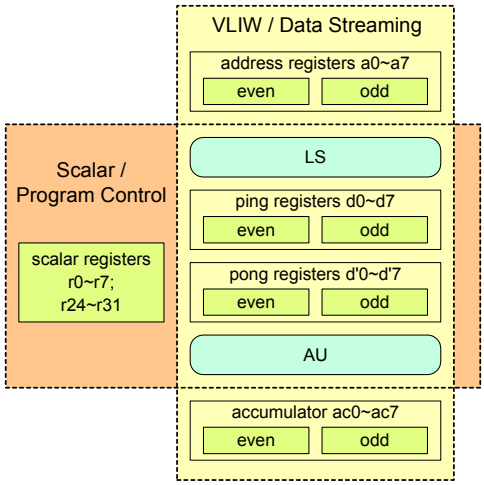

Fig. 3 Unified datapath for RISC & DSP

Fig. 4 shows the add/sub instructions of the unified processor core and the relationship with those of the MIPS32 ISA. The first column shows the MIPS add/sub instructions, where the gray ones denote pseudo instructions. The 2nd column lists the equivalent instructions of the scalar/program control mode, and the 3rd and the 4th columns summarize those of the VLIW/data streaming mode. Note that C compilers do not generate MIPS codes with `addi`, `add`, `sub`, and `neg` (i.e. those cause overflow exceptions), and thus our processor do not support these instructions. Besides, few pseudo MIPS instructions are mapped to physical instructions. Finally, the data manipulation instructions of the scalar mode are actually executed in the DSP datapath, which are highlighted in the yellow background.

MIPS	Scalar	VLIW	
		AU	LS
addiu	**addi**	**addi**	**addi**
addu	**add**	**add**	**add**
subu	**sub**	**sub**	**sub**
addi	NA	NA	NA
add	NA	NA	NA
sub	NA	NA	NA
neg	NA	NA	NA
negu	**neg**	**neg**	**neg**
nop	**nop**	**nop**	**nop**
abs	**abs**	**abs**	**abs**
	rsbi	**rsbi**	
		addi.d	**addi.d**
		add.d	**add.d**
		sub.d	**sub.d**
		abs.d	**abs.d**
		addi.q	**addi.q**
		add.q	**add.q**
		sub.q	**sub.q**
		bf.d	
		saa.q	

Fig. 4 Add/sub instructions

For applications demanding even higher performance, the unified processor core can integrate three more DSP datapaths (up to four clusters). Our first prototype described hereafter contains two clusters – a main cluster as shown in Fig. 3 and a slave cluster as shown in Fig. 2. In other words, it can be configured as a MIPS-like RISC or a 4-way VLIW DSP. For most cases, programmers can exploit the data-level parallelism and arrange the two clusters to operate independently. Otherwise, inter-cluster communication can be performed via the memory subsystem.

3. HIERARCHICAL INSTRUCTION ENCODING

The unified processor core is able to change its operation modes instruction by instruction within a single program stream. This section will describe the enabling technology – the ***hierarchical instruction encoding***, which also helps to reduce the VLIW code sizes significantly.

3.1 VLIW/Data Streaming Mode

VLIW processors are notorious for their poor code density. It comes from the redundancy inside (1) the fixed-length RISC-like instructions, where most operations need not all the control bits actually, (2) the position-coded VLIW packet, where the unused instruction slots must be filled by `NOP`, and (3) the repeated codes due to loop unrolling or software pipelining. ***HAT*** [14] is an efficient variable-length instruction format to solve the first problem. ***Variable-length VLIW*** [10] eliminates the NOP by attaching a dispatch code to each instruction for run-time dispatch and decoding. Moreover, specific marks are required to indicate the boundaries of the variable-length VLIW packets (i.e. with a varying number of effective instructions). ***Indirect VLIW*** [15] uses an addressable internal micro-instruction memory for the VLIW datapath (i.e. the programmable VIM), and the VLIW packets are executed with only very short indices. The RISC-like instructions in the existing packets can be reused to synthesize new packets to reduce the instruction bandwidth. Systemonic proposes an ***incremental encoding scheme*** for the prolog and the epilog of the software pipelined codes [16] to remove the repeated codes. In this paper, we propose a novel hierarchical instruction encoding, which takes into account all the three causes to improve the VLIW code density.

3.1.1 Variable-length instructions

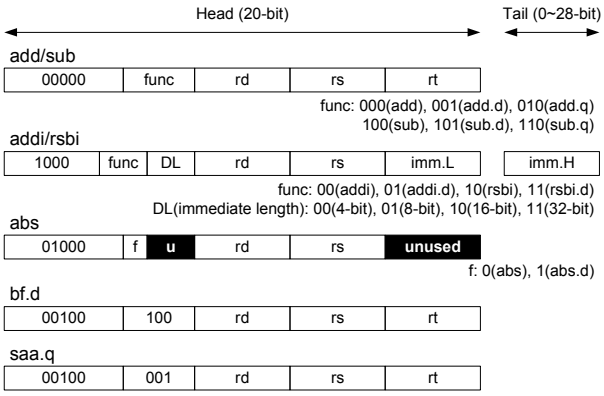

Fig. 5 Machine codes for add/sub instructions

Fig. 5 shows the variable-length encoding of the AU instructions listed in Fig. 4. The code length of an instruction depends on the number of its operands and the frequency of its usage. The variable-length code is divided into a fixed-length 'head' and the remnant variable-length 'tail' as HAT format [14]. This helps to improve the regularity, and reduces the complexity for instruction alignment significantly.

3.1.2 VLIW packets without NOP

The effective instructions for an execution cycle (i.e. without `NOP`) are packed into a VLIW packet with a fixed-length 'CAP'. The CAP has a 'valid' field, where each functional unit (FU) has a corresponding bit to indicate whether it is idle. In other words, the `NOP` is eliminated by turning the corresponding valid bit off. Fig. 6(a) shows the 14-bit CAP format of our prototype with 4-way VLIW. As the example given in Fig. 6(b), the two `addi` instructions are first translated into the machine codes by looking up Fig.5. The 14-bit CAP is set as `00` for VLIW instruction, `0101` to remove `NOP` in the 1st and the 3rd slots, `0010` for total 8-bit tails, `00` for the ping-pong indices of the two clusters, and the ending `00` to disable the SIMD-cluster mode and the conditional executions.

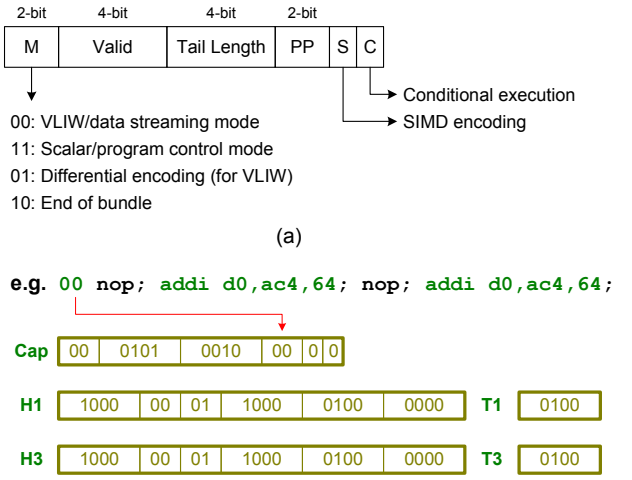

Fig. 6 Instruction packet for a 4-way VLIW (with 2 clusters)

For an *N*-way VLIW processor, our approach uses only *N* 'valid' bits to remove `NOP` in a packet. Variable-length VLIW either uses $\log_2(N+1)$ bits for a VLIW packet to indicate the number of active issue slots, or one bit for each effective instruction to indicate the packet boundary. Moreover, additional $\log_2 N$ bits are required for each instruction to dispatch it to the correspondent slot. Assume a packet has *P* instructions in average ($0 \le P \le N$), and Variable-length VLIW needs $\log_2(N+1)+P\log_2 N$ or $P(\log_2 N+1)$ bits for a packet accordingly. Therefore, it has better compression only for codes with very low parallelism (i.e. with small *P*).

In the VLIW/data streaming mode, the clusters can be configured into SIMD execution by turning on the S bit in the CAP. The instructions of the main cluster will be replicated to all clusters to reduce to code sizes. For the example in Fig. 6(b), 24 bits can be saved by setting `S=1` in the CAP. Moreover, the hierarchical encoding supports the differential mode using a CAP starting with `01`. The VLIW packet of the previous execution cycle will be reused with small updates, such as the ping-pong indices, the destination register for the load instructions, and the source registers for the multiply-accumulate instructions, etc. Finally, all instructions can be conditionally executed by turning on the `C` bit in the CAP.

3.1.3 Instruction Bundle

The variable-length VLIW packets are packed into fixed length instruction bundles to simplify the instruction memory accesses. In order to pipeline the instruction fetch, alignment, and decoding, the fixed-length CAP and the variable-length VLIW packet are placed from the two ends of an instruction bundle respectively as depicted in Fig. 7. For each VLIW packet, the fixed-length heads are placed in order ahead of the variable-length tails. By the way, because the CAP is fixed-length and placed in order, instruction look-ahead is possible to reduce the control overheads.

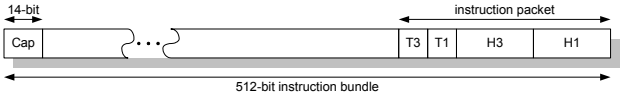

Fig. 7 Instruction bundle

An instruction bundle contains various numbers of VLIW packets, and the code `10` of the leading two bits of the CAP is reserved to denote the bundle end. The total length of the tails is attached in the CAP to locate the next VLIW packet in the pipelined instruction aligner. Finally, depending on the implementations of the instruction aligners described later and our simulations of real programs, the 512-bit instruction bundles are optimal, which have practical decoder complexity and acceptable fragment (i.e. unused bits in a bundle).

3.2 Scalar/Program Control Mode

The scalar instructions are also variable-length encoded, which are similar to those depicted in Fig. 5. But a scalar instruction is decomposed into a fixed-length CAP with leading `11` (instead of head) and a variable-length tail for the remnant bits. The branch instructions change the program flow to a new instruction bundle with the packet index. To easily locate the target VLIW packet, the pointer for its first instruction head is also encoded in the branch instructions. Our first prototype contains a 16KByte on-chip instruction memory, which is equal to a page of 256 bundles.

3.3 Decoder Architecture

To extract from an instruction bundle the appropriate bit fields for decoding is very complicated, especially for the variable-length VLIW packets. Instead of huge multiplexers, we use incremental and logarithmic shifters for instruction alignment, as depicted in Fig. 8. The VLIW packets are continuously shifted out from the two ends of an instruction bundle, and the decoders can operate on the fixed positions. The lengths of the two buffers can be calculated as follows.

$$\text{HT buffer size} = \text{bundle size} - \text{CAP size} \times \left\lceil \frac{\text{bundle size}}{\text{max instr length}} \right\rceil$$

$$= 512 - 14 \times \left\lceil \frac{512}{170} \right\rceil = 456 \text{ (bits)}$$

$$\text{CAP buffer size} = \text{CAP size} \times \left\lceil \frac{\text{bundle size}}{\text{average scalar length}} \right\rceil$$

$$= 14 \times \left\lceil \frac{512}{26} \right\rceil = 280 \text{ (bits)}$$

The buffer size for heads and tails (HT) is the bundle size minus the bits impossible to be HT (i.e. the minimum number of VLIW packets in a bundle multiplied by the fixed length of CAP). The CAP buffer size can be estimated using the average number of instructions in a bundle when the processor stays in the scalar/ program control mode. Note that the CAP and the HT buffers contain overlapped bits, because the boundary between CAP and HT is not deterministic.

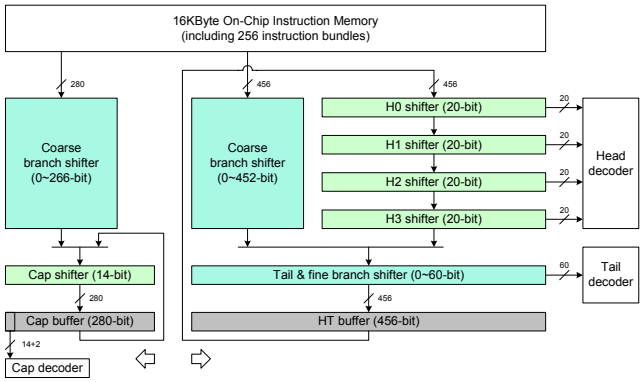

Fig. 8 Instruction aligner

The CAP decoder only examines the leading 16 bits of the 280-bit buffer and shifts out a 14-bit CAP each cycle. Then, the four incremental shifters at the right-hand-side Fig. 8 shift out the 20-bit fixed-length heads depending on the 'valid' bits in the CAP. The logarithmic tail shifter follows to shift out all tails of a VLIW packet. Therefore, the HT buffer will be aligned to the next VLIW packet as the CAP buffer. Finally, two coarse logarithmic shifters are added for branches to align a new bundle with the index and the packet pointer respectively.

4. RESULTS

We have completely verified the proposed **P**acked **I**nstruction & **C**lustered **A**rchitecture (Pica) from the instruction set simulation in C++, the micro-architecture design in cycle-accurate SystemC, to FPGA prototyping and the cell-based silicon implementation. This section will summarize the remarkable results.

4.1 Instruction Set Simulation

We have hand-coded several DSP kernels in assembly to evaluate the performance of the processor core with our instruction set simulator. Table I summarizes the performance comparisons between the state-of-the-art high-performance DSP processors and Pica DSP. The second row shows the number of cycles required for *N*-sample *T*-tap FIR filtering on 16-bit samples, which reveals the on-chip MAC resources. The third row compares the number of execution cycles to perform 2-D discrete cosine transform (DCT). The fourth row lists the performance of the 256-point radix-2 fast Fourier transform (FFT), which is also measured in the execution cycles. Finally, the last row compares the performance of the motion estimation under the MAE (mean absolute error) criteria. The block size is 16×16 and the search range is within ±15 pixels. The simulation results show that the performance of our proposed DSP processor is comparable with the state-of-the-art DSP for various benchmarks once the dataflow is appropriately arranged through the ping-pong register file.

Table I. Performance comparison on various DSP kernels

	TI C64 [10]	TI C55 [17]	NEC SPXK5 [18]	Intel/ADI MSA [19]	**Pica**
FIR	*NT*/4	*NT*/2	*NT*/2	*NT*/2	***NT*/4**
DCT	126	238	240	296	**127**
FFT	2,403	4,768	2,944	3,176	**2,510**
ME	36,538	82,260	-	90,550	**41,372**

(Unit: cycle)

Table II summarizes the performance of Pica for JPEG still image compression [20]. Two 512×512-pixel color images – Lena and Baboon are used in this simulation. The JPEG program written in C is first compiled onto Pica in its MIPS-like scalar/program control mode with a proprietary compiler modified from the GNU tool. The execution cycles are listed in the 2nd and the 3rd columns. Then, the four kernels are hand-coded and optimized for the VLIW/data streaming mode, and the results are given in the 4th and the 5th columns. Note that the performance can be significantly improved by a factor of 10~15.

Table II. Performance evaluation of JPEG

	Pica (scalar only)		Pica (dual-mode)	
	Lena	Baboon	Lena	Baboon
RGB to YCbCr	33,734,912		**487,066**	
DCT	11,181,312		**850,946**	
Q & RLC	12,491,008		**1,187,849**	
Huffman	6,441,701	12,997,472	**1,814,286**	**3,800,285**
Total	63,848,933	70,404,704	**4,340,149**	**6,326,148**

(Unit: cycle)

Finally, three instruction encoding schemes are compared using the above application programs, and the results are given in Table III. The fixed-length encoding uses 152 bits for a VLIW packet, where an AU instruction occupies 36 bits and an LS instruction needs 40 bits respectively. The scalar version of JPEG (JPEG_S) is exceptional, of which the encoding follows the standard 32-bit MIPS instructions. The variable-length VLIW encoding follows the approach by TI [10]. The instructions are first encoded as 40-bit words, and each of them are attached with 1 bit for packet boundary and 2 bits for dispatch. Therefore, every effective instruction requires 43 bits. Finally, all programs are encoded with our hierarchical instruction encoding. The effective instruction bits and the memory requirements while packed into 512-bit bundles are both shown in the table. Note that the hand-optimized JPEG with almost 15× performance requires less instruction memory after the proposed instruction encoding.

Table III. Code size comparison

	Fixed-length	Variable-length	**Hierarchical**	
			Effective	**Bundled**
FIR	5,016	4,559	1,742	**1,834**
DCT	10,944	9,588	3,840	**4,226**
FFT	60,648	50,760	20,946	**22,258**
ME	12,160	10,669	3,948	**4,146**
JPEG_S	36,096*	-	26,690	**27,521**
JPEG	62,472	42,253	19,654	**20,666**

* 32-bit fixed-length RISC instructions (Unit: bit)

4.2 Silicon Implementation

We have implemented the unified processor core in Verilog RTL, which is cross-verified with the cycle-accurate SystemC model to achieve 100% code coverage. The design is synthesized using Physical Compiler from Synopsys with the 0.18um cell library from Artisan. The net-lists are then placed and routed using SoC Encounter from Cadence for the UMC 1P6M CMOS technology. Fig. 9 shows the layout of the proposed unified processor core with on-chip 16-Kbyte data and 16-Kbyte instruction memories. Its gate count is 643,952 (343,284 for core only) and the core area is 3.23mm×3.23mm. The processor has a nine-stage pipeline (4 stages for instruction dispatch and 5 stages for execution), and it can operate at 208 MHz and consume 380.6mW average power (running 2-D DCT).

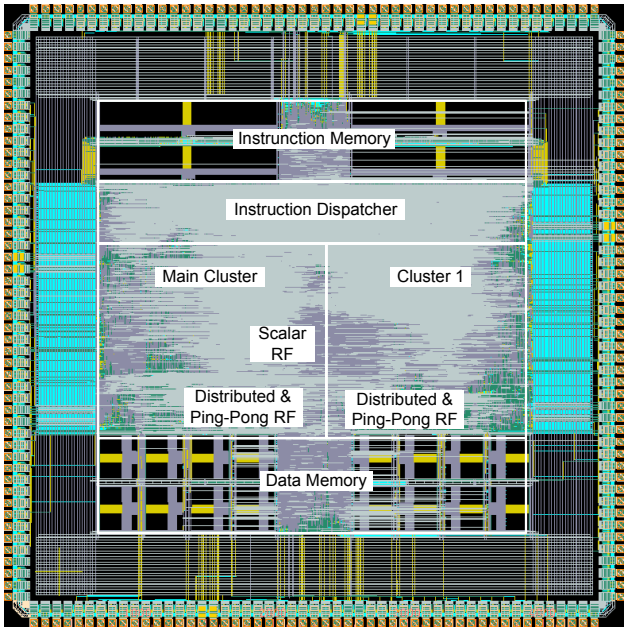

Fig. 9 Layout of the unified processor core

5. CONCLUSIONS

This paper presents the design and the silicon implementation of a unified processor core for RISC and scalable VLIW DSP. The two modes can be changed instruction by instruction within a single program stream via the hierarchical instruction encoding, which also helps to reduce the code sizes. In order to minimize the hardware resources, the DSP has no control construct for program flow, and the data manipulation RISC instructions are performed by DSP. Besides the general applications as the dual-core multimedia systems, new application programs can be easily targeted on its compiler-friendly RISC mode and the performance is then improved by selectively optimizing the kernels on the DSP mode. The tightly-coupled operation modes make such design strategy much more straightforward and efficient. We are now studying the code optimization techniques for the distributed and ping-pong register file and developing a single-pass automatic code generator for the two modes of the unified processor core.

6. REFERENCES

[1] *Intel PXA800F Cellular Processor – Development Manual*, Intel Corp., Feb. 2003

[2] *OMAP5910 Dual Core Processor – Technical Reference Manua*l, Texas Instruments, Jan. 2003

[3] M. Levy, "ARM picks up performance," *Microprocessor Report*, 4/7/03-01

[4] R. A. Quinnell, "Logical combination? Convergence products need both RISC and DSP processors, but merging them may not be the answer," *EDN*, 1/23/2003

[5] J. L Hennessy, and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 3rd Edition, Morgan Kaufmann, 2002

[6] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proc. HPCA-6*, 2000, pp.375-386

[7] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero, "Hierarchical clustered register file organization for VLIW processors," in *Proc. IPDPS*, 2003, pp.77-86

[8] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. M. O. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *Proc. ISCA*, 2000, pp.203-213

[9] E. F. Barry, G. G. Pechanek, and P. R. Marchand, "Register file indexing methods and apparatus for providing indirect control of register file addressing in a VLIW processor," International Application Published under the Patent Cooperation Treaty (PCT), WO 00/54144, Mar. 9 2000

[10] *TMS320C64x DSP Library Programmer's Reference*, Texas Instruments Inc., Apr 2002

[11] K. Arora, H. Sharangpani, and R. Gupta, "Copied register files for data processors having many execution units" U.S. Patent 6,629,232, Sep. 30, 2003

[12] A. Kowalczyk et al., "The first MAJC microprocessor: a dual CPU system-on-a-chip," *IEEE J. Solid-State Circuits*, vol. 36, pp.1609-1616, Nov. 2001

[13] A. Terechko, E. L. Thenaff, M. Garg, J. Eijndhoven, and H. Corporaal, "Inter-cluster communication models for clustered VLIW processors," in *Proc. HPCA-9*, 2003, pp.354-364

[14] H. Pan and K. Asanovic, "Heads and tails: a variable-length instruction format supporting parallel fetch and decode," in *Proc. CASES*, 2001

[15] G. G. Pechanek and S. Vassiliadis, "The ManArray embedded processor architecture," *Euromicro Conf.*, vol.1, pp.348-355, Sep., 2000

[16] G. Fettweis, M. Bolle, J. Kneip, and M. Weiss, "OnDSP: a new architecture for wireless LAN applications," *Embedded Processor Forum*, May 2002

[17] *TMS320C55x DSP Programmer's Guide*, Texas Instruments Inc., July 2000

[18] T. Kumura, M. Ikekawa, M. Yoshida, and I. Kuroda, "VLIW DSP for mobile applications," *IEEE Signal Processing Mag.*, pp.10-21, July 2002

[19] R. K. Kolagotla, et al, "A 333-MHz dual-MAC DSP architecture for next-generation wireless applications," in *Proc. ICASSP*, 2001, pp.1013-1016

[20] W. B. Pennebaker and J. L. Mitchell, *JPEG – Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993

[21] T. J. Lin, C. C. Chang, C. C. Lee, and C. W. Jen, "An efficient VLIW DSP architecture for baseband processing," in *Proc. ICCD*, 2003

[22] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals – Architectures and Features*, IEEE Press, 1996

[23] *TriCore 2-32-bit Unified Processor Core v.2.0 Architecture – Architecture Manual*, Infineon Technology, June 2003

[24] Y. H. Hu, *Programmable Digital Signal Processors – Architecture, Programming, and Applications*, Marcel Dekker Inc., 2002