Journal of Signal Processing Systems 51, 209–223, 2008
© 2007 Springer Science + Business Media, LLC. Manufactured in The United States.
DOI: 10.1007/s11265-007-0061-x

Design and Implementation of a High-Performance and Complexity-Effective VLIW DSP for Multimedia Applications

TAY-JYI LIN, SHIN-KAI CHEN, YU-TING KUO AND CHIH-WEI LIU
Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan

PI-CHEN HSIAO
SoC Technology Center, Industrial Technology Research Institute, Hsinchu, Taiwan

Received: 13 September 2006; Revised: 9 March 2007; Accepted: 17 March 2007

Abstract. This paper presents the design and implementation of a novel VLIW digital signal processor (DSP) for multimedia applications. The DSP core embodies a distributed & ping-pong register file, which saves 76.8% silicon area and improves 46.9% access time of centralized ones found in most VLIW processors by restricting its access patterns. However, it still has comparable performance (estimated in cycles) with state-of-the-art DSP for multimedia applications. A hierarchical instruction encoding scheme is also adopted to reduce the program sizes to 24.1~26.0%. The DSP has been fabricated in the UMC 0.13 μm 1P8M Copper Logic Process, and it can operate at 333 MHz while consuming 189 mW power. The core size is 3.2×3.15 mm^2 including 160 KB on-chip SRAM.

Keywords: VLIW, digital signal processor, register organization, instruction encoding, micro-architecture

1. Introduction

Programmable processors are attractive in embedded multimedia systems because software-based systems need less development effort. Bugs can be fixed with field patches and the products may be upgraded to support new standards with only software. These factors reduce the time-to-market, extend the time-in-market, and thus help to make the greatest profits. However, today's multimedia applications, including speech, audio, image and video processing demand extremely high computing power, and the processors must exploit the inherent parallelism extensively to meet the requirements. For instruction-level parallelism, VLIW architectures [1] have low-cost static instruction scheduling (by compilers or assembly programmers) and thus deterministic execution time compared to the dynamically scheduled superscalar ones. They have already become mainstreams of high-performance DSP designs [2, 3]. However, VLIW processors have two major weaknesses that prevent the integration of more functional units with higher instruction issue rate—the dramatically growing register file complexity and the poor code density. This paper proposes a distributed and ping-pong register organization, which saves 76.8% silicon area and 46.9% access time of non-scalable centralized register files in conventional VLIW

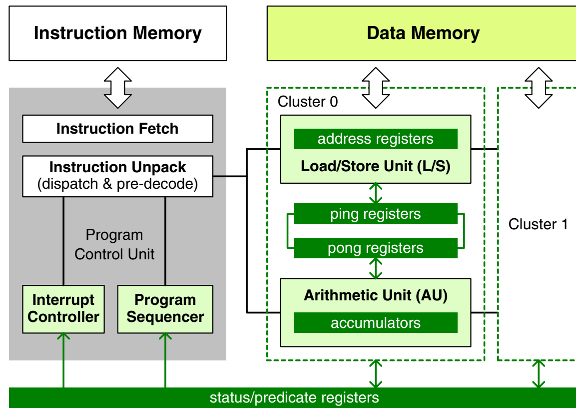

Figure 1. Pica DSP architecture.

processors. A hierarchical instruction-encoding scheme is also proposed to reduce the program sizes to 24.1∼26.0%. Figure 1 shows the VLIW DSP architecture with packed instructions and the clustered architecture (Pica), which integrates up to four clusters depending on application requirements. These clusters can operate independently by exploring data-level parallelism in most DSP applications, or inter-cluster communications can be carried out via the memory subsystem. We have implemented a prototype of Pica DSP with two identical clusters in the UMC 0.13 μm 1P8M Copper Logic Process. The chip can operate at 333 MHz and consumes 189 mW average power. Its core size is 3.2×3.15 mm^2 including the 160 KB on-chip memory.

The rest of this paper is organized as follows. Section 2 describes clustered architectures and our proposed distributed and ping-pong register file respectively that reduce datapath complexity. Section 3 describes hierarchical instruction encoding to improve the code density of VLIW processors. The design and verification flow of programmable processors are summarized in Section 4, while the implementation results from instruction set simulation to silicon proof are available in Section 5. Section 6 concludes this work and outlines our future researches.

2. Register Organization

2.1. Clustered Architectures

A register file provides temporal storage and interconnections for parallel functional units (FU) in a microprocessor. As more and more FU are integrated into a microprocessor, the complexity of the register file grows rapidly. For a centralized register file for N FU, where each FU can read or write any register directly, the area complexity is N^3, the access delay is $N^{3/2}$, and the power dissipation is N^3 [4]. Therefore, register files in modern wide-issue microprocessors are usually partitioned to reduce hardware complexity [5–12]. Figure 2 shows a generic clustered architecture, where each FU has direct accesses only to a subset of the register file. If data need passing across clusters, additional wiring and execution cycles are necessary for inter-cluster communication (ICC).

ICC mechanisms can be classified into three categories [5, 6]. The first one is to use "copy" instructions, which are issued through existing instruction slots [7] or dedicated slots [8], to copy variables from (or to) the register files of other clusters. The former can reduce additional access ports on each of the partitioned register files, but it wastes some effective instruction cycles. The second ICC category is through extended register accesses, where each cluster has limited "read" accesses from

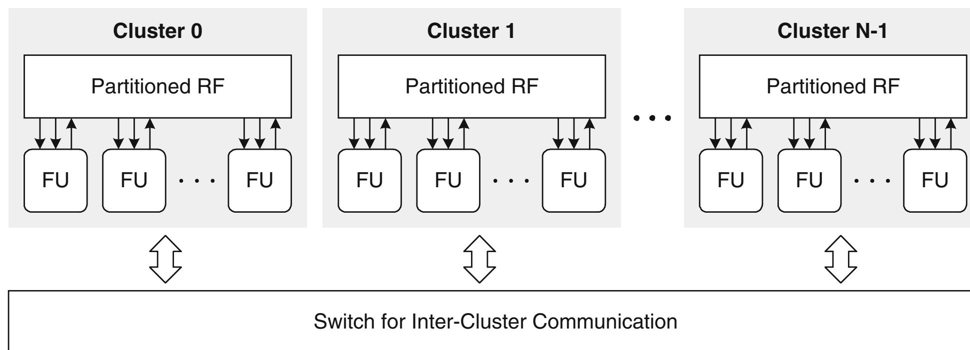

Figure 2. Clustered architecture.

Table 1. Comparison of ICC mechanisms.

	HW complexity
Centralized	$f(x, y, n, w)$
Copy	$c \times f(x/c+p,\ y/c+p,\ n/c,\ w) + g_{CP}(c, p, w)$
Extended read	$c \times f(x/c+p,\ y/c,\ n/c,\ w) + g_{ER}(c, p, w)$
Extended write	$c \times f(x/c,\ y/c+p,\ n/c,\ w) + g_{EW}(c, p, w)$
Load/store pair	$c \times f(x/c,\ y/c,\ n/c,\ w) + g_{LS}(c, p, w)$

[9] (or "write" accesses to [10]) other partitioned register files. The third kind of ICC mechanisms is via some common storage resources, such as shared memory or specific shared registers [11].

Table 1 shows the comparison of ICC mechanisms [6]. The function $f(x, y, n, w)$ denotes the hardware complexity of a centralized register file with x read ports, y write ports, and n w-bit registers, of which the cell-based implementation requires n registers, x n-input multiplexers (n:1 MUX) for the read ports and n y:1 MUX with some glue control for the write ports. The glue control is negligible when w is much larger than y. Thus, the area complexity of f can be approximated with the area of $[x(n-1)+n(y-1)] \times w$ 2:1 MUX and $n \times w$ flipflops, while the time complexity can be approximated as the delay of $\log_2 n$ 2:1 MUX or the delay of $\log_2 y$ 2:1 MUX plus the write control overheads. The register file complexity of the clustered architectures with dedicated ICC copy slots can be described as $c \times f(x/c+p,\ y/c+p,\ n/c,\ w) + g_{CP}(c, p, w)$, where c denotes the number of clusters, each of which has p extra ports. The g function describes the interconnection complexity of the ICC switch network. The architectures without dedicated copy slots have the same register file complexity as those with extended read ICC: $c \times f(x/c+p,\ y/c,\ n/c,\ w) + g_{ER}(c, p, w)$, where no extra write port is needed. Similarly, the register file complexity of those with extended write ICC can be described as $c \times f(x/c,\ y/c+p,\ n/c,\ w) + g_{EW}(c, p, w)$. We have adopted the ICC mechanism through the on-chip data memory [6]. When a load/store pair in a VLIW packet has an identical memory address, the data variable from the "store" will be directly forwarded to the "load". The proposed ICC mechanism with load/store pairs has the lowest complexity of $c \times f(x/c,\ y/c,\ n/c,\ w) + g_{LS}(c, p, w)$. Note that the existing crossbar in the memory subsystem can be used for ICC, and therefore the g complexity can be absorbed.

2.2. Distributed and Ping-Pong Register File

Besides global clustering, the register file complexity of each cluster can be further reduced. Figure 3 shows a cluster of Pica with a load/store unit (LS), an arithmetic unit (AU), and the proposed distributed and ping-pong register file with 32 registers. The 32 registers are divided into four independent banks, and each bank is equipped with the access ports for a single functional unit (FU) only (i.e. two "read" and two "write" ports in our case). The address registers ($a_0 \sim a_7$) and the accumulators ($ac_0 \sim ac_7$) are dedicated to LS and AU respectively, and they are not visible to the other FU. The remnant 16 registers are shared between these two FU and they are divided into two banks with exclusive accesses. In other words, when LS accesses the ping, AU can only access the pong, and vice versa. Each VLIW packet of Pica needs to specify its access mode (to be either ping or pong) with a corresponding bit (i.e. the "ping-pong index" described later) in its encoding.

Pica supports powerful SIMD instructions based on the proposed distributed and ping-pong register file. For example, the double load (store) word instruction:

$$dlw\, d_m, (a_i) + k, (a_j) + l.$$

It performs two memory accesses (i.e. $d_m \leftarrow \mathrm{Mem}_{32}[a_i]$ and $d_{m+1} \leftarrow \mathrm{Mem}_{32}[a_j]$) and two address updates (i.e. $a_i \leftarrow a_i + k$, and $a_j \leftarrow a_j + l$) simultaneously. The index m must be an even number with $m+1$ implicitly specified. These double load/store instructions require six concurrent accesses of the register

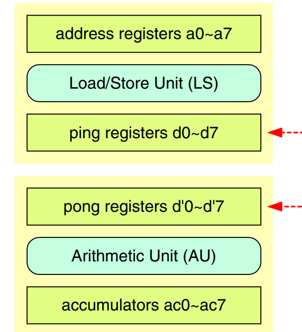

Figure 3. Distributed and ping-pong register organization.

file (including two “reads” and four “writes” for *dlw*, or four “reads” and two “writes” for *dsw*). They do not cause access conflict on the distributed and ping-pong register file, because a_i and a_j are private address registers while d_m and d_{m+1} are ping-pong registers that deliver data between LS and AU. These registers locate at independent banks. The DSP also supports sub-word (16-bit) SIMD multiply-accumulations with full-precision results on two 40-bit accumulators ac_i and ac_{i+1}:

$$fmac.d\, ac_i, d_m, d_n.$$

It performs $ac_i \leftarrow ac_i + d_m.H \times d_n.H$ ($d_m.H$ and $d_n.H$ denote the upper 16 bits of d_m and d_n respectively) and $ac_{i+1} \leftarrow ac_{i+1} + d_m.L \times d_n.L$ ($d_m.L$ and $d_n.L$ denote the lower 16 bits of d_m and d_n) in parallel. Similarly, the index i must be an even number with $i+1$ implicitly specified. This instruction requires six concurrent accesses of the register file (i.e. four “reads” and two “writes”). Without the proposed ping-pong register file, Pica will need a 14-port centralized one for the 14 simultaneous register accesses (i.e. four “reads” and four “writes” for LS and four “reads” and two “writes” for AU).

Finally, each of the four four-port banks (i.e. with two “reads” and two “writes”) in Fig. 3 can be further partitioned into even and odd banks, and the distributed and ping-pong register file can be implemented using eight smaller banks, each of which has only two “read” and one “write” ports. We define $f_{DPP}(n, w) = 8 \times f(2, 1, n/8, w) + g'_{DPP}(w)$ as the hardware complexity of a distributed and ping-pong register file with n w-bit registers. The f function is that for the centralized register file described in Section 2.1, and g'_{DPP} denotes the complexity of the mapping between logical and physical ports. The port mapping contains six 6:1 MUX and two 3:1 MUX (for r_{m+1} and r_{i+1} in LS and AU respectively) for the read ports, and one 2:1 MUX (even private bank), one 4:1 MUX and one 2:1 MUX (odd private banks), two 3:1 MUX (even ping-pong banks) and two 6:1 MUX (odd ping-pong banks) for the write ports respectively. By the way, the ping-pong registers can be extended for a special ICC mechanism via register permutations, such as the proposed “ring-structure register organization” in our previous work [12].

The assembly syntax of our VLIW packet starts from the ping-pong index, followed by the instructions to each issue slot in sequence:

$$ping{-}pong\,index; \quad i0(\text{for LS}); \quad i1(\text{for AU});.$$

In the following, we are going to use FIR and DCT, two popular DSP kernels [13], to illustrate how the proposed distributed and ping-pong register file works, and show some code optimization techniques. Assume there is no delay slot (such as an ALU instruction that follows a load instruction immediately cannot use the load result in classical five-stage pipelined processors) for simplicity. The data memory is byte-addressable.

- FIR filtering

  The following code segment is to perform 64-tap finite-impulse response (FIR) filtering on 1,024 samples. The inputs including both the data samples (pointed by *X*) and the coefficients (pointed by COEF) are 16-bit fractional numbers and the outputs (pointed by *Y*) are 32-bit variables.

	PP	LS	AU
1	0;	*li* a_0, *COEF*;	*li* ac_0, 0;
2	0;	*li* a_1, *X*;	*nop*;
3	0;	*li* a_2, *Y*;	*nop*;
4		*rpt* 1024, 8;	
5	0;	*dlw* d_0, (a_0)+4, (a_1)+4;	*li* ac_1, 0;
6		*rpt* 15, 2;	
7	1;	*dlw* d_0, (a_0)+4, (a_1)+4;	*fmac.d* ac_0, d_0, d_1;
8	0;	*dlw* d_0, (a_0)+4, (a_1)+4;	*fmac.d* ac_0, d_0, d_1;
9	1;	*dlw* d_0, (a_0)+4, (a_1)+4;	*fmac.d* ac_0, d_0, d_1;
10	0;	*li* a_0,*COEF*;	*fmac.d* ac_0, d_0, d_1;
11	0;	*addi* a_1, a_1, -126;	*add* d_0, ac_0, ac_1;
12	1;	*sw* (a_2)+4, d_0;	*li* ac_0, 0;

The zero-overhead loop instructions (*rpt* in the line 4 and line 6) are carried out in the instruction dispatcher and do not consume execution cycles of the datapath. The inner loop (line 7~8 in dark gray) 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. $d_0 \leftarrow \text{Mem}_{32}[a_0]$ and $d_1 \leftarrow \text{Mem}_{32}[a_1]$), and the address registers a_0 and a_1 are updated simultaneously (i.e. $a_0 \leftarrow a_0 + 4$ and $a_1 \leftarrow a_1 + 4$). In the meanwhile, AU performs two 16-bit (SIMD) multiply-accumulations (i.e. $ac_0 \leftarrow ac_0 + d_0.H \times d_1.H$ and $ac_1 \leftarrow ac_1 + d_0.L \times d_1.L$). After accumulating 32

32-bit products respectively with two 40-bit accumulators, ac_0 and 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 LS via d_0. In this example, each output is produced in 35 cycles, and the loops can be unrolled to easily achieve similar performance when the load slots are taken into account.

- Discrete Cosine Transform (DCT)

  Figure 4 shows the eight-point fast DCT algorithm [14], which is extensively used in image and video processing. In this case, the number of operations per sample is larger than the previous FIR example. Thus, LS can assist in performing some simple arithmetic operations to balance the loading.

  The first *dlh* (double load half words; line 3) instruction loads and sign-extends the first and the last 16-bit input data (i.e. x_0 and x_7 in Fig. 4) into 32-bit d_0 and d_1. AU performs a butterfly operation on these two data by switching the ping-pong index in the succeeding VLIW packet. The butterfly operations on the other three input pairs are performed accordingly. The final two butterfly operations for the even outputs (i.e. X_0, X_2, X_4, and X_6 shown in Fig. 4) are performed in LS instead (line 12∼17) after the coefficient multiplication, which must be performed in AU following the two butterfly operations (line 8–11). In this example, an eight-point DCT can be carried out in 24 cycles. Note that two eight-point DCT can be concurrently performed in one cluster with SIMD instructions, and thus a 2D 8 × 8 DCT (i.e. equivalent to sixteen eight-point DCT) requires only 192 cycles.

	PP	LS	AU
1	0;	*li* a_0,*X*;	*nop*;
2	0;	*li* a_1,*COEF*;	*nop*;
3	0;	*dlh* d_0,0(a_0),14(a_0);	*nop*;
4	1;	*dlh* d_0,2(a_0),12(a_0);	*bf* ac_0,d_0,d_1;
5	0;	*dlh* d_0,4(a_0),10(a_0);	*bf* ac_4,d_0,d_1;
6	1;	*dlh* d_0,6(a_0),8(a_0);	*bf* ac_6,d_0,d_1;
7	0;	*lh* d_4,0(a_1);	*bf* ac_2,d_0,d_1;
8	1;	*dlh* d_4,0(a_1),2(a_1);	*bf* d_0,ac_0,ac_2;
9	1;	*dlh* d_6,4(a_1),6(a_1);	*bf* d_2,ac_4,ac_6;
10	1;	*nop*;	*add* d_3,d_1,d_3;
11	1;	*nop*;	*xmpy16* d_3,d_3,d_4;
12	0;	*add* d_4,d_0,d_2;	*add* d_0,ac_3,ac_7;
13	0;	*sub* d_5,d_0,d_2;	*add* d_1,ac_5,ac_7;
14	0;	*dsh* d_4,0(a_0),8(a_0);	*add* d_2,ac_5,ac_1;
15	0;	*add* d_4,d_1,d_3;	*xmpy16* d_4,d_4,d_1;
16	0;	*sub* d_5,d_1,d_3;	*bf* ac_0,ac_1,d_4;
17	0;	*dsh* d_4,4(a_0),12(a_0);	*sub* d_3,d_0,d_2;
18	0;	*nop*;	*xmpy16* d_3,d_3,d_5;
19	0;	*nop*;	*xmpy16* d_6,d_0,d_6;
20	0;	*nop*;	*add* ac_2,d_3,d_6;
21	0;	*nop*;	*xmpy16* d_7,d_2,d_7;
22	0;	*nop*;	*add* d_7,d_7,d_2;
23	0;	*nop*;	*add* ac_3,d_3,d_7;
24	0;	*nop*;	*bf* d_0,ac_1,ac_3;
25	1;	*dsh* d_0,10(a_0),6(a_0);	*bf* d_0,ac_0,ac_2;
26	0;	*dsh* d_0,2(a_0),14(a_0);	*nop*;

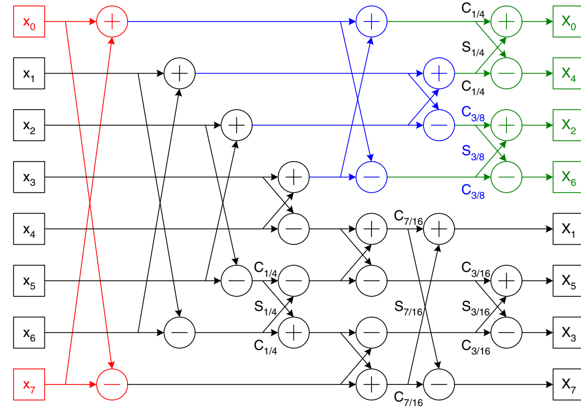

Figure 4. Discrete cosine transform.

3. Instruction Encoding

VLIW architectures are notorious for their poor code density, which comes from the redundancy inside (1) fixed-length RISC-like instructions, for many instructions do not use all bits actually; (2) fixed-length VLIW packets of which the encoding dedicates a bit-field for each issue slot, and NOPs will be filled in the unused slots; and (3) the repeated codes due to loop unrolling or software pipelining. Here, we define an *instruction* to be a RISC-like operation for an issue slot, and a *VLIW packet* (or simply a *packet*) as the instructions issued in the same cycle. *HAT* [15] is an effective variable-length instruction format to improve the first problem. *Variable-length VLIW encoding* applied on TI C64 [9] and NEC SPXK5 [16] can eliminate NOP in a packet by attaching a slot number to each instruction for run-time dispersal. Besides, an additional mark is needed for each instruction to denote the boundary of each variable-length packet (i.e. with a varying number of

effective instructions). *Indirect VLIW* [8] uses a programmable microinstruction memory for the VLIW datapath (i.e. the programmable VIM), and the VLIW packets are stored internally and executed with short indices. The instructions in existing packets may be reused to synthesize new packets to reduce the instruction bandwidth. *Systemonic* proposes an incremental encoding scheme for the prolog and the epilog of software pipelined codes [17], which helps to remove the repeated instructions. Pica uses a novel and integrated encoding scheme [18], which takes into account the three problems to improve the VLIW code density.

3.1. Variable-length Instructions

First, each instruction is variable-length coded, depending on the number of operands, the size of its immediate variable, the frequency of its occurrence, and whether it is conditionally executed. Note that the same operations in different issue slots need not be coded identical. The variable-length instruction is divided into a fixed-length "head" and a variable-length "tail" as the HAT format to reduce the complexity of instruction alignment and dispersal. Figure 5 shows our encoding of the add/sub instructions for the arithmetic unit (AU). Note that the heads need not have the same length across different issue slots.

3.2. VLIW Packets without NOP

NOP instructions are not coded in the VLIW packets for Pica. Here we attach a fixed-length CAP to each

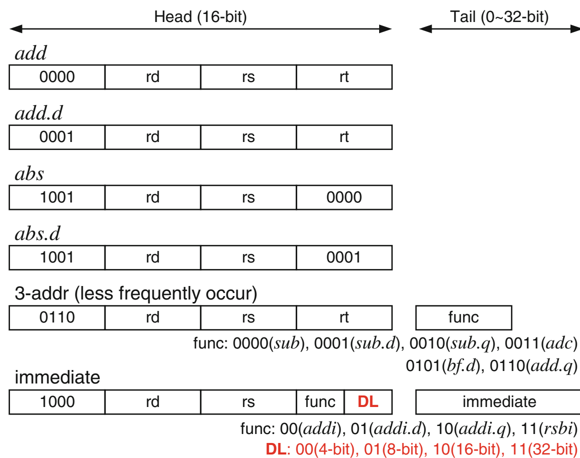

Figure 5. Instruction encoding for add/sub instructions.

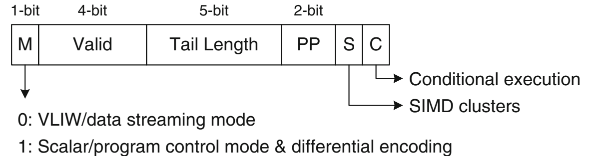

Figure 6. CAP in Pica.

VLIW packet. The CAP has a "valid" field for centralized dispersal, of which each bit indicates whether its corresponding issue slot has an effective instruction. In other words, NOPs are eliminated by turning the corresponding "valid" bits off. Besides, the total length of the "heads" and "tails" (HT) of a VLIW packet can be calculated easily for parallel grouping, with the "valid" for the fixed length "heads" and the "tail length" for the variable-length "tails". To reduce the CAP length and the alignment complexity, while supporting enough flexibility, the tail lengths are restricted to be multiple of 4. Figure 6 shows the 14-bit CAP format of the 4-way Pica DSP. Figure 7a gives an illustrating example with only two effective instructions. The *addi* instructions are converted into machine code first by looking up the encoding table in Fig. 5. The CAP is then encoded as 0 for a VLIW packet; 0101 to remove NOP in the first and the third instruction slots; 00010 for an eight-bit total tail length; 00 for the ping-pong indices; and 00 to disable SIMD clusters and conditional execution.

The identical clusters in the VLIW/data streaming mode of Pica can be configured into SIMD execution by asserting the S bit in the CAP. The instructions of the main cluster will be replicated to reduce the code

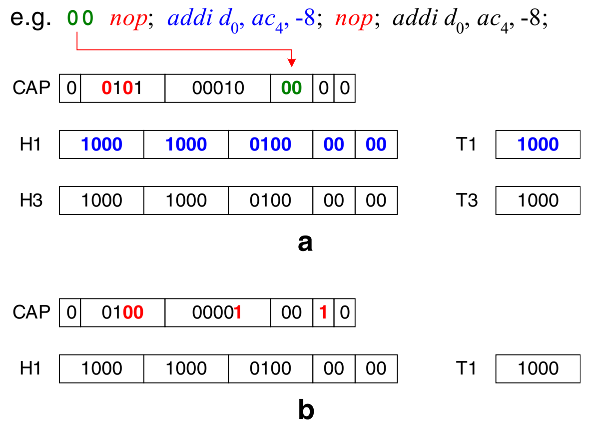

Figure 7. Example of proposed VLIW encoding. **a** An illustrating example with only two effective instructions. **b** SIMD execution by asserting the S bit in the CAP.

size. Figure 7b is an example, which eliminates the encoding of the instruction for the slave cluster by turning on the S bit. Besides SIMD execution, we also supports differential encoding to reduce the repeated instructions in loop-unrolled or software-pipelined programs. A VLIW packet can be reused to synthesize a new one for its succeeding cycle with a small register index offset or new ping-pong indices.

3.3. Instruction Bundles

In order to simplify the accesses of variable-length VLIW packets in the instruction memory, the packets are first packed into fixed-length instruction bundles. The fixed-length CAP and the variable-length HT of a VLIW packet are placed individually from the two ends of an instruction bundle as shown in Fig. 8a. Moreover, all fixed-length "heads" are placed first in order, ahead of their variable-length "tails". The proposed code layout enables parallel packet fetch, alignment, dispersal, and decoding. It is even possible to look ahead succeeding VLIW packets to further reduce control overheads. We attach a "#CAP" field to each bundle to indicate the number of packets in the bundle. In our simulations, the 512-bit bundle size is optimal, which has practical decoding complexity and acceptable fragment (i.e. the unused bits in a bundle that cannot accommodate a new packet). Our first prototype of Pica DSP contains 32 kByte on-chip instruction memory, which can hold 512 512-bit bundles. We will use these parameters hereafter for simplicity.

Instead of huge multiplexers, we use incremental and logarithmic shifters to extract VLIW packets

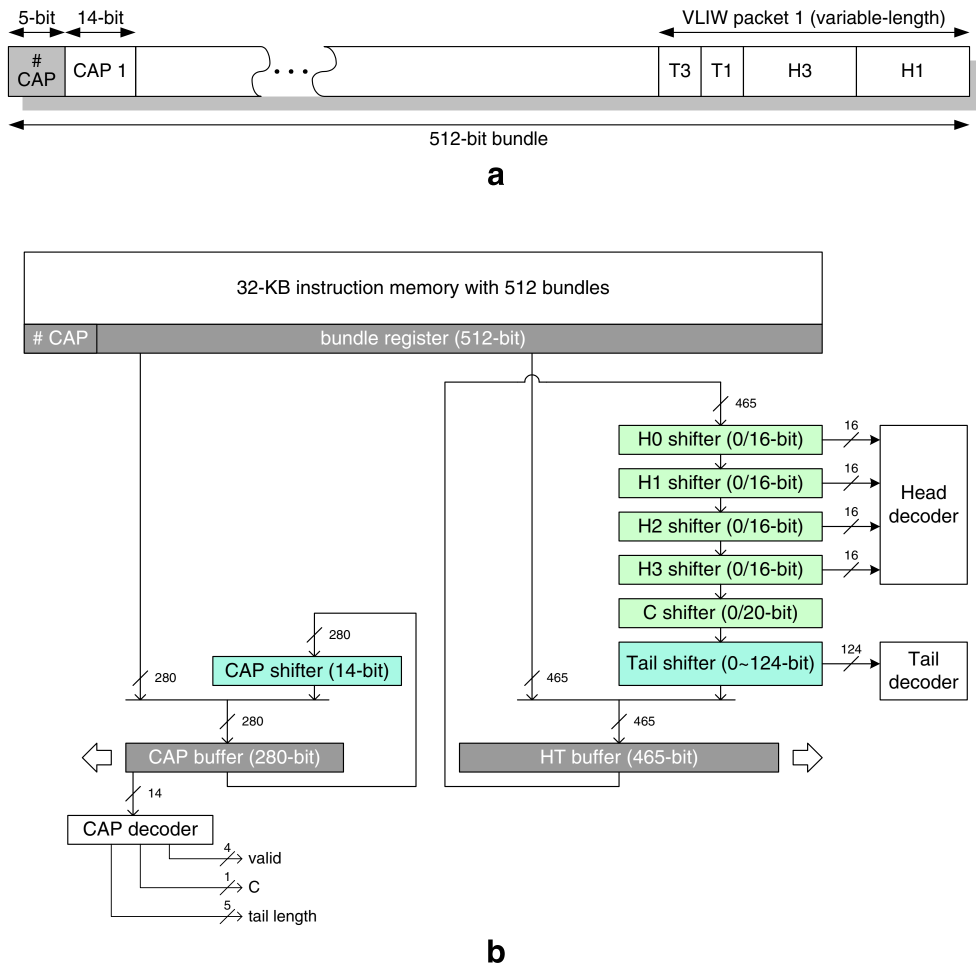

Figure 8. **a** Instruction bundle and **b** packet extractor.

from an instruction bundle as shown in Fig. 8b. The CAP decoder first decodes a leading CAP (14-bit) of the CAP buffer and the CAP will then be shifted out to allocate a new CAP for the next packet. The right-hand-side incremental shifters follow to shift out 0~4 fixed-length "heads" depending on the "valid" of the current CAP. If the C-bit in the CAP is asserted, more 20 bits will be shifted out, where an instruction slot has an independent five-bit condition code. Finally, the logarithmic tail shifter will shift out all "tails" of the VLIW packet. A new VLIW packet can thus be allocated, similarly to the CAP. In brief, a CAP and a HT are continuously shifted out from the two ends of an instruction bundle, and a new VLIW packet will be aligned every cycle to the boundaries of the CAP buffer and the HT buffer respectively. The lengths of the CAP and HT buffers can be estimated as follows:

$$\begin{aligned}
\text{bundle capacity} &= \text{bundle size} - \lceil \log_2(\text{worst}-\text{case \#CAP}) \rceil \\
&= 512 - \lceil \log_2(19.5) \rceil = 507(\text{bits}) \\
\text{worst–case \#CAP} &\cong \frac{\text{bundle capacity}}{\text{average length of scalar instr.}} \\
&= \frac{507}{26} = 19.5(\approx 20) \\
\text{CAP buffer size} &= \text{CAP size} \times \text{worst}-\text{case \#CAP} \\
&= 14 \times 20 = 280(\text{bits}) \\
\text{HT buffer size} &= \text{bundle capacity} - \text{CAP size} \times \left\lceil \frac{\text{bundle capacity}}{\text{maximum packet length}} \right\rceil \\
&= 507 - 14 \times \left\lceil \frac{507}{222} \right\rceil = 465(\text{bits})
\end{aligned}$$

First, the capacity of a bundle to accommodate VLIW packets can be calculated by deducting the "#CAP" bits from the bundle size. We assume that the worst-case number of CAPs turns up when a bundle only holds scalar instructions (encoded as a fixed-length CAP and a variable-length "tail", describe later), which are much shorter than VLIW packets. Thus, the CAP buffer needs to hold this worst-case number of entries, which can be approximated by the maximum number of scalar instructions of average length. The worst-case HT buffer size is the bundle capacity minus the bits impossible to be either "heads" or "tails" (i.e. the minimum number of CAPs in a bundle, which equals to the minimum number VLIW packets). Note that the two buffers have overlapped bits, for the boundary between CAPs and HTs is not deterministic. Finally, it is important to mention that the above estimation is conservative. The buffer sizes can be decided arbitrarily with constraints on the linker and the code generation tools to prevent illegal instruction bundles.

Program flow instructions in Pica have a bundle offset, a CAP index, and a HT pointer, and it can jump to an arbitrary VLIW packet by fetching a new instruction bundle and shifting out unused CAPs and HTs. They are also variable-length encoded as the effective VLIW instructions, where a variable-length program flow instruction is decomposed into a fixed-length CAP with a leading 1 (instead of a fixed-length "head" as the VLIW instruction) and a variable-length "tail". Program flow instructions and VLIW packets are mixed together, and their CAPs, "heads" and "tails" are stuffed in order into instruction bundles. We have developed a linking tool to automatically translate the code labels in Pica assembly (or the instruction offsets of PC-relative jump or branch instructions in a compiled code) into their corresponding bundle offsets, CAP indices and HT pointers in the machine code. Figure 9 shows the complete instruction decoder for Pica DSP, where a CAP selector is used instead of the shifter in Fig. 8b. The "head" dispersal of a packet and the packet alignment with branching are parallel, so the sizes of the incremental shifters are significantly reduced.

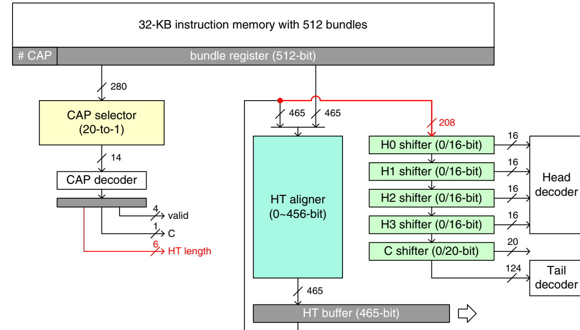

Figure 9. Instruction decoder.

4. Design and Verification Flow

Figure 10 shows a generic design flow for programmable processors, including definition of an instruction set, exploration of an optimal micro-architecture that implements the instruction set architecture (ISA), RTL authoring, and silicon/or FPGA implementations. Designers move forward to the next phase once the function and the performance meet the processor specification. Although iteration is inevitable in the design process, the overall design time can still be minimized by reducing the number of iterations, especially in the major loops. In the following sections, we will describe the design tasks respectively.

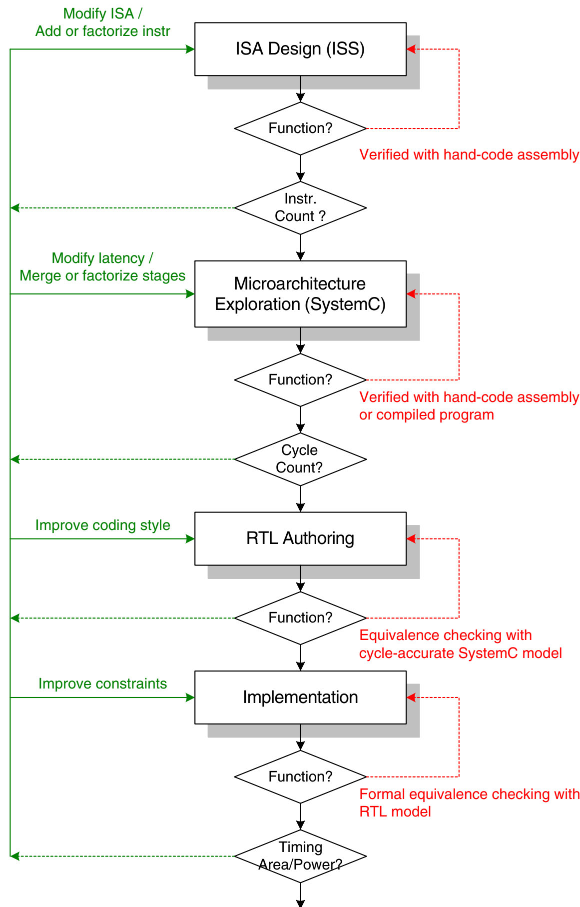

Figure 10. Generic design flow for programmable processors.

4.1. Instruction Set Design

An instruction set characterizes the functional behavior of a processor, and the software must be mapped to or encoded in this instruction set in order to be executed by the processor. In other words, every program is compiled into a sequence of instructions in the instruction set. Attributes associated with an instruction set design include assembly language, instruction format, addressing modes and programming model, which are exposed to the software as perceived by compilers or assembly programmers. The instruction set may influence design effort and implementation complexity. Thus, we focus not only on the micro-architecture techniques but also on the instruction set design. The primitive instructions of Pica DSP are defined by pruning the MIPS32 instruction set. DSP-enhanced instructions are added by surveying the instruction sets of state-of-the-art DSPs, such as TI C64 [9]/C55 [19], NEC SPXK5 [16], and Intel/ADI MSA [20], etc. Whether an instruction is included in our instruction set depends on the tradeoffs between performance (in terms of cycle counts or code sizes of applications) and implementation complexity (both the cycle time and silicon area). An instruction set design can be modeled as an instruction set simulator (ISS) for early performance evaluation, which is usually the most abstract model of a programmable processor. By the way, ISS is also very useful for development of large-scale application software, because it avoids hardware details that are not exposed to the software programming.

4.2. Micro-Architecture Exploration

A micro-architecture is a specific implementation of an ISA design, where all implementations should execute any program encoded in that instruction set. Attributes associated with an implementation include the pipeline design and the memory organization. Figure 11 shows the pipeline design flow. The instructions are first classified to design their corresponding datapaths. In general, a processor must do three generic tasks to perform a computation—(1) arithmetic operations, (2) data movements, and (3) instruction sequencing. The semantics of each of the three instruction types can be specified based on the sub-computations performed by the instruction type. Designers can begin with the five

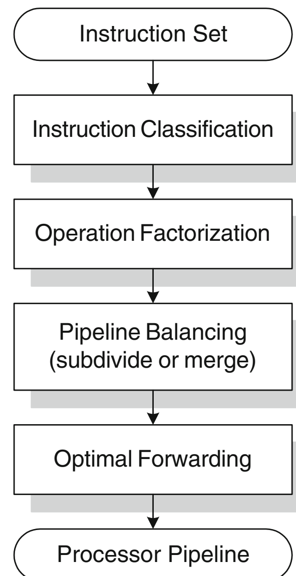

Figure 11. Pipeline design.

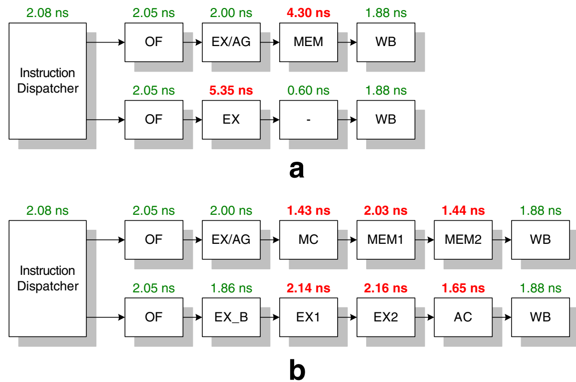

Figure 12. Pipelines for Pica **a** classical and **b** balanced pipeline stages.

generic sub-computations. Instructions with distinct sequences of sub-computations can be separated into different pipelines to improve modularity [21].

One natural operation factorization is based on the five generic sub-computations. That is, each of the five sub-computations is mapped to a pipeline stage, resulting in a five-stage pipeline. The pipeline stages can be further balanced to minimize the internal fragment with two methods—(1) merge multiple sub-computations into one, and (2) subdivide a sub-computation into multiple sub-computations. Figure 12a shows the initial design with two four-stage pipelines, where the critical path lies on the multiply-accumulator (MAC) for arithmetic operations. Figure 12b shows more balanced pipelines by subdividing MAC and memory accesses into three stages.

The forwarding paths can be grouped as a separate module as shown in Fig. 13 to improve timing closure in silicon implementation with registered output ports [22, 23]. We define "tolerable latency (TL)" of a data dependency between consecutive instructions on the fly to be the difference between the time the produced datum is ready and the time the consumer actually needs the datum. A dependency has a negative TL must be avoided with processor stalls or exposed explicitly as delay slots. On the other hand, there is no harm in dependencies with non-negative TLs, where the data item can be passed through either the register file or specific forwarding paths. Thus, dependencies from all data generators to all data consumers in a pipeline can be classified into *non-causal* ($TL < 0$), *timing-critical* ($TL = 0$), and *normal* ($TL > 0$) one. Data for normal dependences are passed through the register file or the forwarding unit without or with small impacts on the timing of the datapath. However, the forwarding paths for timing-critical dependencies increase the number of input ports (MUX) of the functional units, which usually lengthen the critical path. Thus, the inclusion of a timing-critical forwarding path is a difficult tradeoff between the cycle times the cycle counts for application software.

The micro-architecture designs were modeled in SystemC [24] and validated against the ISS to ensure they perform the functional requirements specified by the ISA. Figure 14 is the example dump file from cycle-accurate SystemC simulation, which is faster than RTL simulations by an order of magnitude. After micro-architecture explorations, the final design was described in synthesizable Verilog for the implementation phase. Moreover, the RTL design has been cross-verified with the cycle-accurate SystemC model from bottom up to achieve 100% coverage (e.g. statement, branch, state, arc, etc.) [25].

4.3. *FPGA Prototyping*

Figure 15 depicts a simplified Pica system for multimedia applications based on the AMBA AHB bus. We have prototyped this system on the ARM Versatile platform [26] with an ARM926EJ-S core, 128 MB SDRAM, and a rich set of peripherals such as an LCD controller, a stereo audio codec. Pica was first encapsulated in an AHB wrapper and ported on the Xilinx XC2V6000 FPGA on the Logic Tile daughter card. The system AHB, the expansion AHB and ARM run at 70 MHz, 35 MHz and 210 MHz respectively, while Pica operates at 70 MHz with the embedded DLL-based clock generator on FPGA that doubles the clock rate of the expansion AHB. ARM functions as a smart DMA controller that moves data

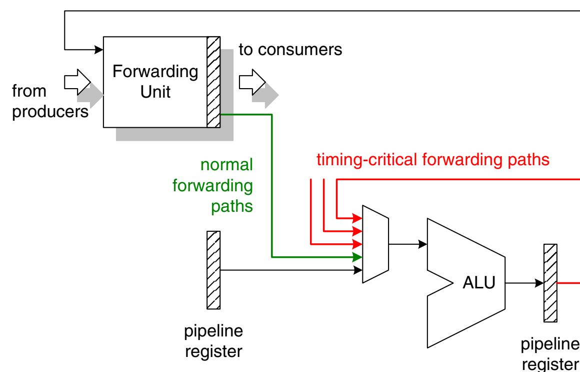

Figure 13. Forwarding unit and simplified "EX" pipeline stage.

```
~~~~~~~~  Cycle 71  ~~~~~~~~

========  BOOLEAN REGISTERS  ========
B0 = 1            B1 = 0            B2 = 0            B3 = 0
B4 = 0            B5 = 0            B6 = 0            B7 = 0

========  CLUSTER 0  ========
========  PING REGISTERS  ========
D'[0] = ffff6400  D'[1] = 6600      D'[2] = fffffe00  D'[3] = db3a
D'[4] = 1413a     D'[5] = ffff8ac6  D'[6] = 0         D'[7] = 5a84
========  ADDRESS REGISTERS  ========
A[0] = 0          A[1] = 8          A[2] = 8          A[3] = 0
A[4] = 0          A[5] = 0          A[6] = 0          A[7] = 0
========  PONG REGISTERS  ========
D[0] = fffe4d00   D[1] = 4d00       D[2] = 1065e      D[3] = 1fc4b
D[4] = 1535e      D[5] = ffff46a2   D[6] = 494b       D[7] = fffc50b5
========  ACCUMULATORS  ========
AC[0] = 113e6     AC[1] = fffff278  AC[2] = e865      AC[3] = fffeb300
AC[4] = ffffcc00  AC[5] = ffff0000  AC[6] = ffff7f00  AC[7] = ffffe700

========  CLUSTER 1  ========
========  PING REGISTERS  ========
D'[0] = 0         D'[1] = 0         D'[2] = 0         D'[3] = 0
D'[4] = 0         D'[5] = 0         D'[6] = 0         D'[7] = 0
========  ADDRESS REGISTERS  ========
A[0] = 0          A[1] = 0          A[2] = 0          A[3] = 0
A[4] = 0          A[5] = 0          A[6] = 0          A[7] = 0
========  PONG REGISTERS  ========
D[0] = 0          D[1] = 0          D[2] = 0          D[3] = 0
D[4] = 0          D[5] = 0          D[6] = 0          D[7] = 0
========  ACCUMULATORS  ========
AC[0] = 0         AC[1] = 0         AC[2] = 0         AC[3] = 0
AC[4] = 0         AC[5] = 0         AC[6] = 0         AC[7] = 0
```

Figure 14. Cycle-accurate SystemC simulation.

between the Pica on-chip memory and the on-board 128 MB SDRAM. Besides, all peripherals are controlled by ARM, too. After the power-on reset, Pica stays in its standby mode until ARM initializes the instruction memory and triggers an interrupt. We have successfully demonstrated a simplified H.264 intra coder on the prototyping system, which can real-time compress 21 CIF-format pictures per second. Besides, we have also implemented a standard JPEG encoder and a real-time MP3 player using this prototyping platform.

5. Results

5.1. Performance Evaluation

We have hand-optimized several DSP kernels in assembly to evaluate the performance of Pica DSP with cycle-accurate instruction set simulation. Table 2 shows the comparison between state-of-the-art DSP processors and Pica DSP. The first column shows the number of cycles required for 40-sample and 16-tap real-value FIR filtering on 16-bit samples. The second column compares the execution cycles to perform the eight-by-eight 2D DCT. The third column is for the 256-point radix-2 fast Fourier transform (FFT) [13]. The last column summarizes the cycle counts for block-based motion estimation under MAE criteria [27], of which the block size is 16×16 and the search range is within ± 15 pixels. TI C64 [9] and NEC SPXK5 [16] are two high-performance VLIW DSP. C64 can issue eight instructions per cycle and its datapath is partitioned into two identical clusters, each of which has a 16-port centralized register file. Besides, the two clusters can communicate with each other via the extended-read mechanism. SPXK5 is a four-issue processor with a centralized register file, which has only eight general-purpose registers to reduce the hardware complexity. TI C55 [19] and Intel/ADI MSA [20] are conventional DSP architectures with dual multiply-accumulators, and the former even allows memory operands (i.e. not a load/store architecture [28]), and these results are included just for reference. All cycle counts are excerpted from the application notes from the vendors to reveal more objective performance indices.

Pica has DSP-oriented instructions such as MAC and rich SIMD instructions (e.g. as those described in Section 2), but they are not new and can be found in some other DSP processors. However, most Pica instructions are simple (i.e. RISC-like). Pica would likely have comparable performance with state-of-the-art DSP if it is equipped with a generic register file, for the supported instructions are very similar. The simulation results in Table 2 show that the clustered architecture with distributed and ping-pong register organization has small impact on highly-parallel DSP kernels, whenever the dataflow is appropriately arranged. By the way, the instruction slots inside the hand-optimized assembly codes can be almost filled to maximize the hardware utilization. Note that the concurrently-issued instructions may be

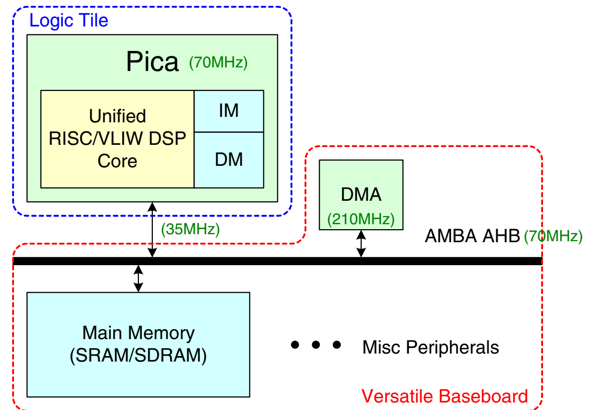

Figure 15. Simplified Pica system for multimedia processing.

Table 2. Performance comparison (in cycles).

	FIR	DCT	FFT	ME	Processor descriptions
TI C64 [9]	194	126	1,246	36,538	8-issue VLIW at 1 GHz (90 nm)
NEC SPXK5 [16]	–	240	2,944	–	4-issue VLIW at 250 MHz (0.13 μm)
TI C55 [19]	394	238	4,922	82,260	Dual MAC at 300 MHz
Intel/ADI MSA [20]	381	284	3,176	90,550	dual MAC at 400 MHz
Pica DSP	232	123	2,510	41,372	4-issue VLIW at 333 MHz (0.13 μm)

severely limited in general applications with irregular dataflow or arbitrary dependency (especially across the clusters), and thus the performance will degrade significantly. Finally, we have also hand-crafted a standard JPEG encoder [14, 29] on Pica DSP, which can compress 512×512 24-bit color Lena and Baboon images in 4,340,149 and 6,326,148 cycles respectively (i.e. about 53.6~76.7 frames/s at 333 MHz).

5.2. *Code Size Analysis*

It is very difficult to evaluate the quality of the instruction encoding of different processor architectures, because it strongly depends on the instruction set architectures (ISA), compilers and code generation strategies. Therefore, we base on a single ISA (the aforementioned Pica DSP) and try our best to make the comparison fair. Table 3 shows the comparison of various instruction encoding schemes. The first four rows summarize the results from the hand-optimized assembly programs described in Table 2. The fifth and the sixth rows are for a standard JPEG encoder [14] and a simplified H.264 encoder [27]. The 1st column lists the code sizes for 192-bit fixed-length VLIW packets, each of which contains four 48-bit instructions. The second column lists the variable-length VLIW encoding used in TI C64 and NEC SPXK5. The third column shows the results with variable-length instruction encoding and the proposed NOP removal scheme with CAP. The fourth column shows the improved code sizes with SIMD execution, while the fifth shows those with additional differential encoding.

Table 3. Code size comparison (in bits).

	Fixed	NOP removal	Hierarchical		
			Original	+SIMD	+Diff
FIR	5,016	4,947	2,686	1,646	1,646
DCT	11,248	10,557	6,160	3,696	3,696
ME	12,616	13,158	6,946	4,486	3,838
FFT	60,648	56,202	32,850	20,058	19,698
JPEG	62,472	47,481	28,936	20,128	19,748
H.264	229,368	167,535	99,424	72,992	72,128

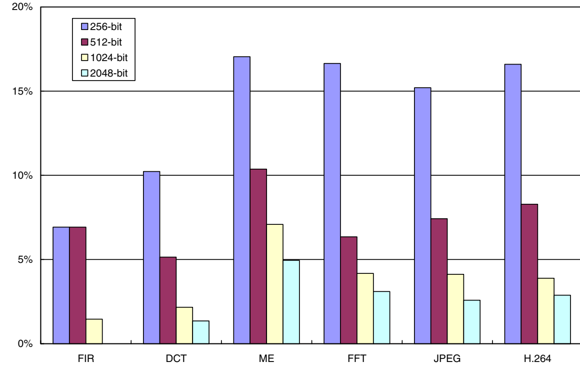

Figure 16. Bundling overheads (due to fragment).

The variable-length VLIW packets are then stuffed into bundles of different sizes to evaluate the fragment (i.e. the unused bits in a bundle that cannot accommodate a new packet). Note that the lengths of program flow instructions depend on the bundle size ("bundle offset" can decrease by 1 bit, but both "CAP index" and "HT pointer" need 1 more bit, as the bundle size is doubled). To minimize the offset of intra-instruction fragment (i.e. the "tails" must be multiple of 4), we use worst-case lengths of program flow instructions for all bundles sizes under comparison. Figure 16 shows the bundling overheads over the effective instructions.

Table 4. Comparison of register organizations.

	Centralized		Proposed
	$f(16, 12, 64, 32)$	$2 \times f(8, 6, 32, 32) + g_{LS}$	$2 \times f'(32, 32) + g_{LS}$
Delay	3.18 ns	1.91 ns	1.69 ns
Gate count	196,920	104,976	45,850
Area	1,440,000 μm^2	766,314 μm^2	334,544 μm^2

Table 5. Complexity comparison.

Bundle size	256-bit	512-bit	1,024-bit	2,048-bit
Delay (ns)	1.79	2.08	2.32	2.71
Gate count	14,780	29,792	66,165	147,195

5.3. *Silicon Implementation*

We have implemented the proposed distributed and ping-pong register file for a cluster with 32 32-bit registers and equivalent centralized register files for both one and two clusters. These three designs are described in Verilog RTL, and then synthesized using Synopsys Design Compiler and UMC L130E High Speed FSG Library (from Faraday Technology). The designs are optimized for timing, and their net-lists are placed and routed for UMC 0.13 μm 1P8M Copper Logic Process using Cadence SoC Encounter. Table 4 summarizes the implementation results from post-layout simulations, where the proposed register organization reduces the access delay by a factor of 1.88 and the silicon area by 4.30 respectively for the dual-cluster configuration.

We have implemented the instruction decoders for various bundle sizes in Verilog RTL, which are synthesized using Synopsys Design Compiler and UMC L130E High Speed FSG Library (from Faraday Technology). Table 5 shows the gate counts and the delays of the synthesis results. Combined with the aforementioned overhead analysis as depicted in Fig. 16, 512-bit bundles are optimal for our implementation.

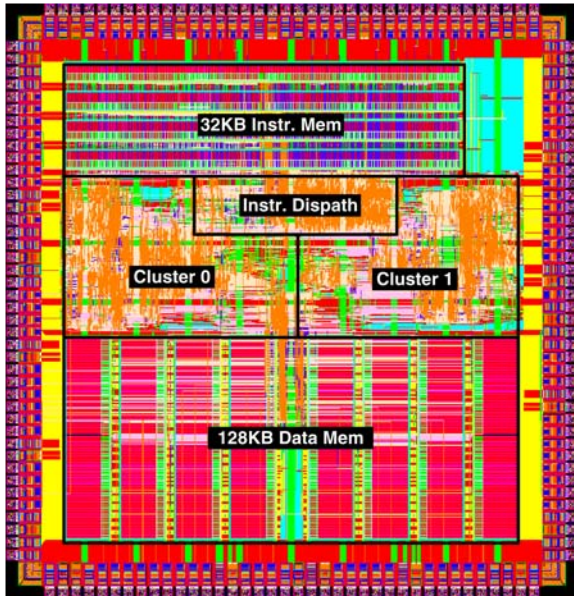

Figure 17. Layout of Pica DSP.

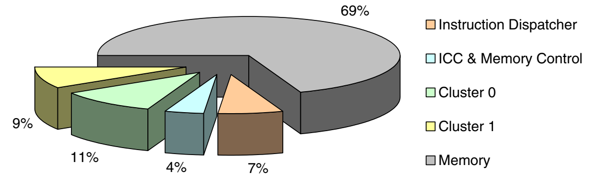

Figure 18. Power breakdown of Pica DSP.

Finally, we have implemented the complete four-way Pica DSP with the proposed ICC mechanism with load/store instruction pairs, the distributed and ping-pong register file, and the hierarchical VLIW decoder. The RTL design is synthesized using Synopsys Physical Compiler and UMC L130E High Speed FSG Library. The gate count is 223,578, where the register file and the instruction dispatcher (i.e. hierarchical VLIW decoder) account for 20.5 and 13.1% respectively. The net-lists are placed and routed using Cadence SoC Encounter for UMC 0.13 μm 1P8M Copper Logic Process. Figure 17 shows the layout of Pica DSP, and the core size is 3.2×3.15 mm^2 including the 128-kB data memory and the 32-kB instruction memory. The DSP core can operate at 333 MHz while consuming 189 mW average power (while running 2D DCT), where the power breakdown is shown in Fig. 18.

6. Conclusions

The paper describes the design and implementation of a high-performance and complexity-efficient VLIW DSP for multimedia applications. A simple inter-cluster communication mechanism with load/store pairs and a novel distributed and ping-pong register organization are proposed to reduce the register file complexity. The simulation results show that a four-issue VLIW DSP with the proposed micro-architecture design has comparable performance with state-of-the-art high-performance DSPs. However, 76.8% silicon area and 46.9% access time are saved from an equivalent centralized register file found in most DSPs. Hierarchical VLIW encoding with flexible variable-length instruction formats, NOP removal and automatic code replication is proposed to solve VLIW code density problem. In our simulations, the code sizes of a four-issue VLIW DSP can be reduced to only 24.1~26.0%. An efficient decoding architecture is proposed, too. Finally, a complete four-issue VLIW

DSP with the area-efficient register organization and the hierarchical instruction decoder is implemented in UMC 0.13 μm Copper Logic Process. The gate count is 223,578 (core only), where the register file and the instruction dispatcher account for 20.5 and 13.3% respectively. The core size is 3.2×3.15 mm^2 including the 128 kB data memory and the 32 kB instruction memory. The DSP core can operate at 333 MHz with 189 mW power dissipation (while performing 2D DCT on a natural image).

The drawback of the proposed register organization is that it significantly complicates the operation scheduling and the code generation. Actually, we can only optimize the assembly codes by hand. We have tried some compilation methods with ping-pong access constraints, but the performance of the compiled codes is still far behind that of hand-optimized ones. Development of customized compilation techniques may be a good research direction [30].

The variable-length VLIW packets are packed into fixed-length instruction bundles to simplify the instruction memory accesses. However, it introduces fragments, especially for small bundles. There is an opportunity for future researches on the optimal code generation and linking methods to reduce such fragments. Besides, automatic synthesis of optimized variable-length instruction encoding for application-specific instruction set processors (ASIP) is also an interesting topic. The overheads of "valid" bits to remove NOP in each VLIW can be reduced by allowing only limited patterns [31] or even applying entropy coding according to the occurrence frequency. Compressing redundant operands such as the reduced ISA in ARM Thumb [32] may also further improve the code density at small costs.

References

1. P. Lapsley, J. Bier, and E. A. Lee, "DSP Processor Fundamentals - Architectures and Features," IEEE Press, 1996.
2. Y. H. Hu, "Programmable Digital Signal Processors—Architecture, Programming, and Applications," Marcel Dekker, 2002.
3. J. A. Fisher, P. Faraboschi, and C. Young, "Embedded Computing—A VLIW Approach to Architecture, Compiler, and Tools," Morgan Kaufmann, 2005.
4. S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register Organization for Media Processing," in *Proc. HPCA*, 2000, pp. 375–386.
5. A. Terechko, E. L. Thenaff, M. Garg, J. Eijndhoven and H. Corporaal, "Inter-Cluster Communication Models for Clustered VLIW Processors," in *Proc. HPCA*, 2003, pp. 354–364.
6. T. J. Lin, P. C. Hsiao, C. W. Liu, and C. W. Jen, "Area-Efficient Register Organization for Fully-Synthesizable VLIW DSP Cores," *Int. J. Electr. Eng.*, vol. 13, pp. 117–127, May 2006.
7. 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.
8. G. G. Pechanek and S. Vassiliadis, "The ManArray Embedded Processor Architecture," in *Proc. Euromicro Conf.*, 2000, pp. 348–355.
9. TMS320C64x DSP Generation. http://www.ti.com.
10. K. Arora, H. Sharangpani, and R. Gupta, "Copied Register Files for Data Processors Having Many Execution Units," US Patent 6,629,232, Sep. 30, 2003.
11. 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.
12. T. J. Lin et al., "Performance Evaluation of Ring-Structure Register File in Multimedia Applications," in *Proc. ICME*, July 2003.
13. A. V. Oppenheim, R. W. Schafer, and J. R. Buck, "Discrete-Time Signal Processing, 2nd ed.," Prentice Hall, 1999.
14. Independent JPEG Group. http://www.ijg.org.
15. H. Pan and K. Asanovic, "Heads and Tails: A Variable-Length Instruction Format Supporting Parallel Fetch and Decode," in *Proc. CASES*, 2001.
16. T. Kumura, M. Ikekawa, M. Yoshida, and I. Kuroda, "VLIW DSP for Mobile Applications," *IEEE Signal Process. Mag.*, pp. 10–21, July 2002.
17. G. Fettweis, M. Bolle, J. Kneip, and M. Weiss, "OnDSP: A New Architecture for Wireless LAN Applications," Presented at Embedded Processor Forum, San Jose, 2002.
18. T. J. Lin et al., "A Unified Processor Architecture for RISC & VLIW DSP," in *Proc. GLSVLSI*, Apr. 2005.
19. TMS320C55x DSP Generation. http://www.ti.com.
20. R. K. Kolagotla et al., "High-Performance Dual-MAC DSP Architecture," *IEEE Signal Process. Mag.*, pp. 42–53, July 2002.
21. J. P. Shen and M. H. Lipasti, "Modern Processor Design—Fundamental of Superscalar Processors," McGraw-Hill, 2005.
22. M. Keating and P. Bricaud, "Reuse Methodology Manual—For System-on-a-Chip Designs, 3rd ed.," Kluwer, 2002.
23. D. Chinnery and K. Keutzer, "Closing the Gap Between ASIC & Custom—Tools and Techniques for High-Performance ASIC Design," Kluwer, 2002.
24. J. Bhasker, "A SystemC Primer," Star Galaxy Publishing, 2002.
25. J. Bergeron, "Writing Testbenches—Functional Verification of HDL Models, 2nd ed.," Kluwer, 2003.
26. Versatile Platform Baseboard for ARM926EJ-S. http://www.arm.com/.
27. I. E. G. Richardson, "H.264 and MPEG-4 Video Compression," Wiley, 2003.
28. J. L Hennessy and D. A. Patterson, Computer Architecture—A Quantitative Approach, 3rd ed.," Morgan Kaufmann, 2002.
29. W. B. Pennebaker and J. L. Mitchell, JPEG—Still Image Data Compression Standard, Van Nostrand Reinhold, 1993.
30. Y. C. Lin, Y. P. You, and J. K. Lee, "Register Allocation for VLIW DSP Processors with Irregular Register Files," in *Proc. CPC*, 2006.
31. Intel 64 and IA-32 Architectures Software Developer's Manual, Intel, Nov. 2006.
32. The Thumb Architecture Extension. http://www.arm.com.

Tay-Jyi Lin received the B.S. degree in Electrical and Control Engineering and the Ph.D. degree in Electronics Engineering, from National Chiao Tung University, Taiwan, in 1998 and 2005, respectively. He is now with the Department of Electronics Engineering and the Institute of Electronics, National Chiao Tung University, as a Researcher Assistant Professor. His research interests include VLSI signal processing, configurable computing, and computer architecture.

Shin-Kai Chen received the B.S. degree in Electronics Engineering from National Chiao Tung University, Taiwan, in 2004, where he is working toward the Ph.D. degree. His researches include system software designs and application mapping of programmable processors.

Yu-Ting Kuo received the B.S. and the M.S. degrees in Electronics Engineering from National Chiao Tung University, Taiwan, in 2004 and 2006, respectively. He is working towards a Ph.D. degree. His researches include low-power signal processing, digital hearing aids, and computer architecture.

Chih-Wei Liu received the B.S. and Ph.D. degrees in Electrical Engineering from National Tsing Hua University, Taiwan, in 1991 and 1999, respectively. From 1999 to 2000, he was an Integrated Circuit Design Engineer at the Electronics Research and Service Organization (ERSO) of Industrial Technology Research Institute (ITRI), Taiwan. Then, near the end of 2000, he started to work for the SoC Technology Center (STC) of ITRI as a project leader and eventually left ITRI at the end of October, 2003. He is currently with the Department of Electronics Engineering, National Chiao Tung University, Taiwan, as an Assistant Professor. His current research interests include SoC and VLSI system design, processor architecture, digital signal processing, digital communications, and coding theory.

Pi-Chen Hsiao received the B.S. degree in Electronics Engineering from National Central University, Taiwan, in 2003, and the M.S. degree in Electronics Engineering from National Chiao Tung University, Taiwan, in 2005. He is now with the SoC Technology Center at Industrial Technology Research Institute, Hsinchu, Taiwan. His research interests include DSP architecture and datapath designs.