Introduction

- RISC-based.
  - Simplified control logic
  - Easy to pipeline and parallelism

- Code density.
  - Size of embedded memory (onchip memory)
  - Overhead of off-chip memory access

- We design a RISC+DSP core that focuses on performance and code density.
Overview of Proposed Architecture

- Modified Harvard architecture (PM, XDM, YDM)
- Two parallel instructions per cycle
- Five-stage pipeline
- Zero-overhead Loop
- Dual-ALU and Dual-MAC
- Sixteen address generation units (AGU)
- Compound Instructions
Proposed Architecture

DECODER PC_CNTL INT_CNTL

Dual MAC

XDMEM

YDMEM
Zero-overhead Loop

- Most Loop bodies have **predictable termination condition**

- Zero-overhead loop design: Make the predictable loop instructions not to **stall the pipeline**.
Zero-overhead Loop

- Beginning address of loop
- PC Stack
  - top_pc_stack
- program status
  - terminate condition detector
  - top_loop_stack
  - comparator
- ending address of loop & terminate condition
- Loop Stack
- PC reg
- Incrementor
- pc select mux
- pc select logic
Computational Units

- Dual ALUs can perform a full 32-bit operation or two 16-bit operations.

- Dual MACs can perform two 16x16 MAC operations per cycle.

- 32-bit Barrel Shifter
Architecture of Fast Dual ALU

- ALUH1: in2[31:16], cin, 1'b1, mux, result[31:16]
- ALUH0: in1[31:16], cin, 1'b0, result[31:16]
- ALUL: in2[15:0], in1[15:0], cin, cout, result[15:0]

32 bits mode

Dual ALU
Dual MACs can perform dual 16-bit multiply-accumulate operations in the same clock cycle.

- The input sources of dual MACs come from special registers MX0, MX1, MY0 and MY1.
- The value of input registers of the dual MACs updates from register file, Xdata memory, Ydata memory.
The Dual MAC architecture
Address Generation Unit

- AGU operates independent of the ALU.

- There are sixteen AGUs in our design.

- Each AGU consists of index register (AI), length register (AL), and modification register (AM).
  - Actual address <= content (AI) + content of (AM).
Address Generation Unit (cont.)

- Linear addressing mode
- Circular addressing mode
Why compound instructions?
- Provide higher code density.
- Provide higher computational power.
- There is close relation between some instructions.

Two kinds of compound instructions
- Memory access and next address generation
- MAC and memory access
Memory Access and Next Address Generation

- Array operation: the memory access sequence is usually **linear** and **regular**.

```plaintext
Non-compound instructions

dreg <= mem(ai) processing
..................
..................
ai <= ai + 5
dreg <= mem(ai) processing
..................
..................
ai <= ai + 5
dreg <= mem(ai) processing
..................
..................

access sequence

a0
a5
a10
a15

Compound instructions

dreg <= mem(ai) || ai += 5 processing
..................
..................

..................

..................

..................

..................

dreg <= mem(ai) || ai += 5
```
MAC and Memory Access

MAC and memory access

\[
C = \sum a[i] \times b[i]
\]

Non-compound instruction

Compound instruction

mx0 <= mem(ai0) ;
my1 <= mem(ai8) ;
macc0 <= mx0.h * my0.h + macc0 ;
macc1 <= mx0.l * my0.l + macc1 ;
ai0 <= ai0 + 1 ;
ai8 <= ai8 + 1 ;
mx0 <= mem(ai0) ;
my1 <= mem(ai8) ;
macc0 <= mx0.h * my0.h + macc0 ;
macc1 <= mx0.l * my0.l + macc1 ;
ai0 <= ai0 + 1 ;
ai8 <= ai8 + 1 ;

mx0 <= mem(ai0) || ai0 += 1 ;
my0 <= mem(ai8) || ai8 += 1 ;
mx1 <= mem(ai0) || ai0 += 1 ;
my1 <= mem(ai8) || ai8 += 1 ;
macc0 <= mx0.h * my0.h + macc0 ||
macc1 <= mx0.l * my0.l + macc1 ||
mx0 <= mem(ai0) || ai0 += 1 ||
my0 <= mem(ai8) || ai8 += 1 ;
macc0 <= mx1.h * my1.h + macc0 ||
macc1 <= mx1.l * my1.l + macc1 ||
mx1 <= mem(ai0) || ai0 += 1 ||
my1 <= mem(ai8) || ai8 += 1 ;

.................
Experimental Results

- We have implemented the proposed RISC+DSP core in synthesizable Verilog HDL and targeted towards TSMC 0.35 μm cell library.

- 100Kgates (ex memory); 120MHz

- We compare ours experimental results to TI TMS320C6X.
Experimental Results To TI TMS320C6X

<table>
<thead>
<tr>
<th></th>
<th>RISC-Based</th>
<th>Instruction word</th>
<th>Data width</th>
<th>Pipeline stages</th>
<th>VLIW</th>
<th>Compound instructions</th>
<th>Max speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>Yes</td>
<td>24</td>
<td>32</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>120</td>
</tr>
<tr>
<td>TMS320C6x</td>
<td>Yes</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>No</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TMS320C6X (#cycle)</th>
<th>Ours(#cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) dot product</td>
<td>N/2 + 8</td>
<td>N/2 + 4</td>
</tr>
<tr>
<td>(2) Weighted vector sum</td>
<td>N + 10</td>
<td>3N + 8</td>
</tr>
<tr>
<td>(3) Vector dot product and square</td>
<td>N + 8</td>
<td>N + 8</td>
</tr>
<tr>
<td>(4) Block move</td>
<td>N/2 + 5</td>
<td>N/2 + 5</td>
</tr>
<tr>
<td>(5) Sum of squares</td>
<td>(N - 1)/2 + 9</td>
<td>N/2 + 4</td>
</tr>
<tr>
<td>(6) Vector Max</td>
<td>N/2 + 13</td>
<td>2N/3 + 4</td>
</tr>
<tr>
<td>(7) Vector Max Index</td>
<td>2N/3 + 12</td>
<td>3N + 8</td>
</tr>
</tbody>
</table>

The number of clock cycles of TMS320C6X and ours.
Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>TMS320C6X</th>
<th>Ours</th>
<th>Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) dot product</td>
<td>36</td>
<td>14</td>
<td>61%</td>
</tr>
<tr>
<td>(2) Weighted vector sum</td>
<td>50</td>
<td>30</td>
<td>40%</td>
</tr>
<tr>
<td>(3) Vector dot product and square</td>
<td>31</td>
<td>20</td>
<td>35%</td>
</tr>
<tr>
<td>(4) Block move</td>
<td>26</td>
<td>14</td>
<td>46%</td>
</tr>
<tr>
<td>(5) Sum of squares</td>
<td>40</td>
<td>12</td>
<td>70%</td>
</tr>
<tr>
<td>(6) Vector Max</td>
<td>57</td>
<td>12</td>
<td>79%</td>
</tr>
<tr>
<td>(7) Vector Max Index</td>
<td>47</td>
<td>32</td>
<td>32%</td>
</tr>
</tbody>
</table>

The code sizes of all the benchmarks targeted to our design are smaller than those to TMS320C6X.
Experimental results show that our design gives reasonable computational power and provide higher code density.

The results demonstrate that the compound instruction is useful to improve the performance and code density.

However, compound instructions make the control logic more complex.

FPGA prototyping is under way.