VLIW Processors – Lecture 18


Ingrid Verbauwhede
Department of Electrical Engineering
University of California Los Angeles
ingrid@ee.ucla.edu

BUT: DSP Software Development

- Complex DSP architecture not amenable to compiler technology
- Algorithms are modeled in high level language (e.g. C++)
- Solutions are implemented and debugged in hand-optimized assembler - large development effort with minimal tool support

HLL algorithmic model → hand coded assembler → prototype code → optimize & debug → production code

- Long, frustrating time to market
- Fragile legacy code

Still used in handhelds, but change in basestations, Part II
Processor style

CISC (Complex Instruction Set)
RISC (Reduced Instruction Set)
VLIW (Very Long Instruction word)
• extension of RISC

Ultimate goal: make the processor “compiler-friendly”

CISC DSP’s:
• hard for the compiler to detect the CISC instructions
• compilers don’t like special registers (Accumulators, T-reg, address registers, etc.)
• but extremely efficient in terms of power & performance

ILP = Instruction level parallelism
• how much is available in the “code”
• how much is achievable on the hardware

ILP Hardware

Data path units in parallel:
• same hardware modules (SIMD)
• different hardware modules (Superscalar or VLIW)

Superscalar:
• hardware detects available parallelism in the Instructions and rearranges the RISC like operations.
• very complex hardware controller
• unpredictable execution time

VLIW: compiler decides the alignment of the RISC like instructions
• might fill it up with NOP’s
• DSP applications well suited for this because lots of inherent parallelism.
• and because flow of operations known “at-compile-time” (recall the “synchronous data flow graph” lecture)
Support by compiler

Key issue:
- Detect available parallelism from the sequential software program

Sequential:
\[
\begin{align*}
t3 &= t2 + 1; \\
\text{store (add0)} &= t3; \\
f6 &= f7 \times f14; \\
t1 &= p2 + p7; \\
t5 &= p2 + p10; \\
t3 &= f6 - t1; \\
\ldots
\end{align*}
\]

Parallel: reorder the instructions, keep the data dependencies intact
\[
\begin{align*}
t3 &= t2+1; \\
f6 &= f7* f14; \\
t1 &= p2 + p7; \\
t5 &= p2 + p10; \\
\text{store (add0)} &= t3; \\
t3 &= f6 - t1;
\end{align*}
\]

Software pipelining:

For tight loops:
Original code: inner loop, 2 cycles per iteration

\[
\begin{align*}
(I: 1 \text{ to } N, \text{ step } 1) \\
\text{begin} \\
a[I] &= c[I] \times x[N-I-1]; \\
s[I] &= a[I] + s[I-1]; \\
\text{end};
\end{align*}
\]

Software pipelining: Inner loop: 1 cycle per iteration, 2 operations in parallel

\[
\begin{align*}
a[0] &= c[0] \times x[N]; \\
(I: 1 \text{ to } N-1, \text{ step } 1) \\
\text{begin} \\
s[I] &= a[I] + s[I-1]; \\
a[I+1] &= c[I+1] \times x[N-I]; \\
\text{end}; \\
s[N] &= a[N] + s[N-1];
\end{align*}
\]

See also presentation by F. Catthoor, Monday June 3, 3pm!
VLIW architectures for DSP

5 architecture features:
- ILP (SIMD versus VLIW)
- memory hierarchy
- register organization
- branch support
- code size

Compiler Driven VLIW

Instruction format: cond/branch ex1 ex2 ex3 ...... exn

- Data memory
- Register Array
- Interconnect

- Large orthogonal register set, regular interconnect
- Atomic RISC-like operations => heavily pipelined, high freq. clock
Explicitly Parallel Instruction Computing

- Execution Clusters

  ![Diagram of execution clusters with data memory, register array, and interconnect](image)

  - Data memory
  - Register Array
  - Interconnect
  - ex1 (alu)
  - ex2 (alu)
  - ex3 (ld/st)
  - ex4 (alu)
  - ex5 (mpy)
  - ex6 (ld/st)

ILP: SIMD

SIMD in general purpose processors:
- cut larger container in smaller pieces
  - e.g. 64 bit word = 8 byte operations
  - adjust ALU for that (no overflow between bytes)
  - e.g. Ueda's presentation for Viterbi acceleration

In GPP: called "subword parallelism" or multimedia extensions
  - e.g. MMX extensions to Intel processors

Compiler problem:
- rewrite code with special instructions
- use special libraries
SIMD code rewriting:

Original code:  
(I: 1 to N, step 1)  
begin  
s[I] = a[I] + b[I];  
end;

Assume 32 bit operator cut in 4 pieces:

(I: 1 to N/4, step 1)  
begin  
t1 = packed4_load a[I];  
t2 = packed4_load b[I];  
t3 = add4 t1, t2;  
packed4_store s[I];  
end;

Requires:  
• rewriting the code  
• alignment of data in memory  
• size is divisible by 4  
• architecture dependent (what if other processor has add6?)

VLIW parallelism

Original code:  
(I: 1 to N, step 1)  
begin  
s[I] = a[I] + b[I];  
end;

Loop unrolling 4 times (compiler can do this):  
(I: 1 to N, step 4)  
in parallel:  
begins[I] = a[I] + b[I];  
s[I+1] = a[I+1] + b[I+1];  
s[I+2] = a[I+2] + b[I+2];  
s[I+3] = a[I+3] + b[I+3];  
end;

Requires:  
• 4 times bigger code size  
• 4 execution units (each might have too large wordlength)  
• 4 times as many registers  
• flexible combination of primitive operations  
• no rewriting of the code
Register file architecture

N unit VLIW:
- 2N input ports, N output ports
- would require 3N port register file
- if N=8, would mean 24 ports!
  - area grows with $N^2$
  - delay grows with N
- Today: around 15 ports register files
- therefore execution clustering

Schedule extra register copy operations:
- done by hardware (invisible to the programmer)
- unpredictable execution time
- done by compiler:
  - needs to cluster operations
  - create Direct acyclic graph + cluster algorithms

Mix subword parallelism in VLIW: tricky

Memory architecture

Data memory access patterns:
- spatial locality but no temporal locality
  (pixel streams, sample streams)
- temporal locality but no spatial locality
  (small look-up tables often reused)

Caches don’t work
- unpredictable worst case execution time
  (worst case becomes too worst case)
Therefore local memory (separate memory space)

How to tell the compiler:
- "pragma’s" or programmer directions:
  
  ```
  # pragma local_memory
  int local_memory_array[1000];
  ```
Prefetch buffer

- Technique to avoid memory accesses to be the bottleneck for throughput driven stream based applications (such as a stream of pixels, samples, etc.)
- Bypass the cache
- Special prefetch instructions:
  - at loop level: separate loop to fill the prefetch buffer
  - at iteration level: (kind of similar to software pipelining)
    fetch the samples for instructions N iterations ahead
    (N is called the prefetch_stride)
  - at instruction level:
    prefetch for the current loop iteration but do it in the beginning of the loop.

Branch architecture
Code size

Applications & coding
Texas Instruments ‘C6201

- 8-way VLIW with two execution clusters
- 256 bit (8x32) instruction fetch with variable length execute set
- Each 32 bit instruction individually predicated
- 11 stage pipeline
- 1600 MIPS, 400 MMACs @ 200 MHz

FIR Filter on TI ‘C6x

Hand-coded assembly: 32-tap FIR filter

```
loop:
  ldw .d1t1 *a4++,a5
  || ldw .d2t2 *b4++,b5
  || [b0] sub .s2 b0,1,b0
  || [b0] b .s1 loop
  || mpy .n1x a5,b5,a6
  || mpyh .n2x a5,b5,b6
  || add .l1 a7,a6,a7
  || add .l2 b7,b6,b7
```

- Outer Loop: 23 cycles, 180 bytes
  - 1 cycle in inner loop
- All 8 exec units used in inner loop - maximum efficiency
  - 2 MACs per cycle

Assembly syntax more difficult to learn.
Hard to get full use of all 8 execution units at once.
Software pipelining difficult to implement, and requires longer prolog/epilog (larger code size).

Courtesy: Gareth Hughes: Bell Labs Australia
Viterbi on TI 'C6x

3-cycle 2-ACS Inner-Loop

```c
LOOP:
  [b1] b .s1 LOOP
  ||[b1] sub .s2 b1,1,b1
  ||[b2] sth .d1 b12,*+a6[8]
  ||[b2] add .d2 b0,b11,b14
  || cmpgt .l1 a11,a10,a1
  || cmpgt .l2 b11,b10,b0
  || mpy .m1x 1,b5,a4
  ||[a2] sub .s1 a2,1,a2
  ||[b2] sth .d1 a12,*a6++
  ||[b0] mpy .m2 1,b11,b12
  || mpy .m1 1,a10,a12
  || sub .l2x a7,b5,b10
  || ldh .d2 *++b9,b5
  || shri .s2 b14,2,b14
  || mpy .m1x 1,b4,2,b14
  || shri .s1 a12,*a6++
  || shri .s2 a2,2,b0,b0
  || mpy .m2 1,b01,b12
  || sub .l2x a7,b4,a12
  || sub .l2x a7,b4,a12
  || mpy .m2 1,b10,b12
  || shri .s1 a12,*a6++
  || mpy .m1 1,a10,a12
  || sub .l2x a7,b5,b10
  || dst .d1 *a5+[2],b13
  ||[end of LOOP]
```

- 16-state Viterbi decoder for GSM
  - 3 cycles per butterfly
  - 32 cycles per GSM timeslot (8 butterflies)
  - MPY instructions used to move data

Utilization of execution units in Viterbi decoder

Lucent / Motorola Star*Core SC140

- 6-way VLIW with 128 bit (8x16) instruction fetch
- Prefix instructions for high performance without sacrificing code density
- Each execution set (parallel instructions + prefix) predicated
- 5 stage pipeline
- 1800 MIPS, 1200 MMACs @ 300 MHz

**Program / Data Memory**

- Program Sequencer
- Address Registers (27)
- Data Registers (16)
- Instruction Dispatcher
- Data Registers
- AAU
- MAC
- ALU
- BFU
Viterbi on Star*Core

**GSM (K=5, 16 states)**

\[
\begin{align*}
\text{move} \cdot 21 (r0)+,d0,d1 & \quad \text{move} \cdot 21 (r1)+,d1,d2 \\
\text{add} d0,d4 & \quad \text{add} d6,d2 \\
\text{sub} d4,d0 & \quad \text{add} d2,d6 \\
\text{max2vit} d4,d2 & \quad \text{max2vit} d0,d6 \\
\text{vsl} d2,d6:d1,d1,(r3)+n0 &
\end{align*}
\]

- Decision bits are manually stored using the Viterbi Shift Left (VSL) instruction:

\[
\begin{align*}
\text{max2vit} d4,d2 & \quad \text{max2vit} d0,d6 \\
\text{vsl} d2:d6:d1:d1,(r2)+n0 &
\end{align*}
\]

- Hardware support for Viterbi algorithm:
  - **max2vit** instruction.
  - **vsl** instruction
- 1 cycle per butterfly through software-pipelining

Results written to memory

**Parallel DSP Architectures**

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Parallelism</th>
<th>Compile?</th>
<th>Power?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S/Scalar</td>
<td>Dynamic instruction level</td>
<td>××</td>
<td></td>
</tr>
<tr>
<td>VLIW</td>
<td>Static instruction level</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>SIMD</td>
<td>Highly regular, data dependent</td>
<td>××</td>
<td></td>
</tr>
<tr>
<td>MIMD</td>
<td>Task level</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

MIMD with VLIW / SIMD provides high order parallel execution

*The future of high performance DSPs is MIMD*
2G Basestation Baseband Processing

- Multiple DSPs used for baseband processing.
- RISC Microcontroller for timing, framing, I/O control
- Software upgradable over the network
- DSPs dominate cost and power consumption

Future trend - integrate baseband processing - low cost Pico BTS

Tx/Rx baseband processing board for 2-carrier GSM basestation