Multimedia Extensions For Microprocessors: SIMD Within A Register

Hank Dietz
Assoc. Prof. of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu
http://dynamo.ecn.purdue.edu/~hankd

Abstract

SIMD (Single Instruction stream, Multiple Data stream) parallel processing has long been used to speed up image processing and other multimedia operations. However, SIMD was usually implemented using large numbers of custom processing elements. With the importance of multimedia growing rapidly, it is a natural step to extend processor instruction sets with some form of SIMD multimedia support.

Generically, SWAR (SIMD Within A Register) is implemented by partitioning the k-bit registers, data paths, and function units of a conventional processor into n k/n-bit fields that can be processed using SIMD-parallel instructions. Ordinary instructions can be used, but special "SIMD partitioned" instructions will often yield better performance. AMD, Cyrix, and Intel have MMX (MultiMedia eXtensions); Digital Alpha has MAX (MultimediA eXtensions); Hewlett-Packard PA-RISC has MAX (Multimedia Acceleration eXtensions); Sun SPARC V9 has VIS (Visual Instruction Set).

This talk will briefly overview all the above SWAR models... and how SWAR can be made into a viable target for data-parallel high-level language compilers.

An introduction to SWAR is available online at `http://dynamo.ecn.purdue.edu/~hankd/SWAR/`

Multimedia!

- Multimedia support sells systems
- Typical applications:
  - Video (e.g., MPEG, editing, phones)
  - 3D graphics (e.g., Doom, VRML)
  - Digital photography (e.g., Photoshop)
  - Audio (e.g., digital effects, mixing)
- Want to do these without custom chips

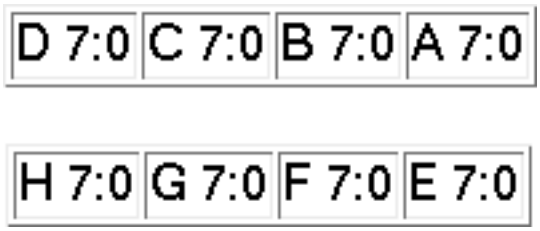
Characteristics of Multimedia

- Generally bandwidth intensive
- Mostly operations on small integers
  - 8-bit pixel color values
  - 16-bit audio samples
- Lots of SIMD algorithms

SIMD?

- Single Instruction, Multiple Data parallelism
- Very "VLSI friendly"
- Vector and data parallel programming models
- Deterministic performance and debugging
- Synchronous inter-PE communication
- Enable masking to turn-off PEs
- `ANY` and `ALL` testing for jumps

SIMD Within A Register (SWAR)



- Divide 32-bit/64-bit/128-bit registers, datapaths, and function units into multiple *k*-bit fields
- Perform SIMD operations across fields
- Improved bandwidth, Loads/Stores treat fields as a block
- RISC-like SIMD control minimizes VLSI complexity, pipeline constraints

Target Hardware for SWAR

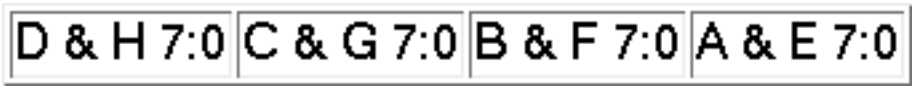
- *Ordinary 32-bit/64-bit processors*
- AMD K6 MMX (MultiMedia eXtensions)
- Cyrix M2 MMX (MultiMedia eXtensions)
- Digital Alpha MAX (MultimediA eXtensions?)
- Hewlett-Packard PA-RISC MAX (Multimedia Acceleration eXtensions)
- Intel Pentium & Pentium Pro MMX (MultiMedia eXtensions)
- Sun SPARC V9 VIS (Visual Instruction Set)
- PowerPC will also have SWAR support....

A Portable SWAR Model

- Manufacturer SWAR support is **machine dependent**
  - Different (often irregular) instructions
  - Different width registers, fields
  - Different register use constraints
    (e.g., can’t mix MMX with floating point)
  - HLL models specify each instruction
- Need complete SIMD/vector features
- Need variable size/parallelism-width data
- Cannot have HLL-visible "holes"
  (i.e., omit quirky SWAR instructions)

Polymorphic Operations

- Parallel ops independent of field type (size)
  Don’t need special instructions
- All bitwise operations are polymorphic
- `ANY` operation is polymorphic



Partitioned Operations

- Parallel ops that enforce field partitioning (e.g., cut carry/borrow chains)
- Most arithmetic requires partitioned ops
  - Wrap-around (conventional)
  - Saturation
- Three implementation methods (mix & match):
  - Partitioned instructions
  - Unpartitioned ops with correction code
  - Controlling field values

Partitioned Instructions

Instruction	AMD/ Cyrix/ Intel MMX	Digital MAX	HP MAX	Sun VIS
Add	8, 16, & 32		16	16 & 32
Multiply	16			8 x 16
Compare	8, 16, & 32			16 & 32
Merge Max		8 & 16		
Absolute Difference		8		8

Unpartitioned Ops with Correction

- Use ordinary 32-bit/64-bit op, but correct for field interactions
- For 4 8-bit fields in 32-bit registers,

```
return(x + y);

t = ((x & 0x7f7f7f7f) + (y & 0x7f7f7f7f));
return(t ^ ((x ^ y) & 0x80808080));
```

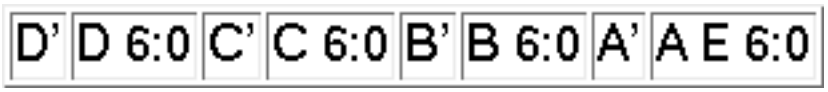
Unpartitioned Ops with Correction

- Can be expensive, especially for saturation
- Speedup comes from:
  - More parallelism (e.g., 2-bit fields)
  - Vectorized field Load/Store
  - Reduced register pressure
  - No partial-value forwarding stalls

Controlling Field Values

- Use ordinary 32-bit/64-bit op, but ensure field values remain within fields
  - By range analysis
    (e.g., 0..100 + 0..100 fits in 8 bits)
  - By clipping field values
    (e.g., using partitioned vector minimum/maximum)
  - By using "carry/borrow spacers"
- Effectively trades register space utilization for simpler, faster, instruction sequences

Controlling Field Values



- 4 **7-bit** fields in 32-bit registers
- Partitioned addition:

```
return(x + y);

return((x + y) & 0x7f7f7f7f);
```

- Partitioned subtraction:

```
return(a - b);

return(((a | 0x80808080) - b) & 0x7f7f7f7f);
```

- Can optimize by tracking spacer values

Communication

- Cheap, synchronous, inter-PE communication is a key benefit of SIMD
- Two basic flavors:
  - Neighbor-based
    (e.g., like MasPar `xnet`)
  - Router-based, "random"
    (e.g., like MasPar `router`)
- `x[y]` is inefficient without hardware support... HP MAX has a `Permute` instruction

Neighbor Communication

- 4 8-bit fields in 32-bit registers
- Send to "right" neighbor:

```
return(x << 8);
```

- With wraparound:

```
return((x << 8) | ((x >> 24) & 0x000000ff));
```

- Multi-hop neighbor communication is also easy

Type Conversion

- Type conversion is field pack/unpack
- Partitioned instructions support some pack/unpack
- Can be very inefficient serial code
- Conceptual problems:
  - Field memory layout/order?
  - How to deal with size/width relationships?
- Suggest isolating native/SWAR layouts at the "SWAR function" interface...

Recurrence Operations

- Basic flavors:
  - Associative reductions
  - Scans (parallel prefix)
- Not much hardware support for these, and scans are very expensive without it

Reductions: Summation

- Conventional C summation:

```
t = 0;
for (i=0; i<MAX; ++i) t += x[i];
return(t);
```

- 4 8-bit fields in 32-bit register, SWAR summation:

```
t = ((x & 0x00ff00ff) +
     ((x >> 8) & 0x00ff00ff));
return((t + (t >> 16)) & 0x000003ff);
```

- Not really very expensive...
  C-coded SWAR summation of 1-bit fields easily outruns the Intel Pentium population count hardware instruction!

Enable Masking

- Need to be able to "disable" PEs...
  but can't really turn them off
- Some partitioned compare instructions can be used to generate a bit mask...
- The basic "trick":

```
where (c) a = b;

t = (-!c);
a = ((c & b) | ((~c) & a));
```

- Optimize by using masking only where:
  - Some PEs might be disabled
  - Not disabling could be observable

Compiler Optimizations

- Spacer value tracking
- Enable masking optimizations
- Aggressive bitwise value tracking, e.g.:

```
return((x & 0x00ff) << 4) & 0xff00);

return((x << 4) & 0x0f00);
```

- Plus the usual stuff...

So, What Are We Doing?

- Fall 1996, EE468 built a really crude SIMD-to-MMX compiler
- Now, SIMD-to-any-SWAR compilers:
  - Will be full public domain releases
  - Small HLL designed to build functions callable from C
  - HLL independent of target, except for parallelism width and efficiency variations
  - Arbitrary precision integer fields,
    1-bit, 2-bit, 3-bit, ..., 32-bit
  - SWAR code-generation target libraries
  - Manufacturers & applications researchers becoming involved
- Later, SWAR across SMP and/or PAPERS clusters....