

# CIS 371

## Computer Organization and Design

### Unit 14: Exploiting Data-Level Parallelism with Vectors

## Best Way to Compute This Fast?

- Sometimes you want to perform the **same** operations on **many** data items

- Surprise example: SAXPY

```
for (I = 0; I < 1024; I++)  
    Z[I] = A*X[I] + Y[I];  
  
0:  ldf X(r1), f1    // I is in r1  
   mulf f0, f1, f2  // A is in f0  
   ldf Y(r1), f3  
   addf f2, f3, f4  
   stf f4, Z(r1)  
   addi r1, 4, r1  
   blti r1, 4096, 0
```

- One approach: superscalar (instruction-level parallelism)
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Problem: wide-issue superscalar scaling issues
    - $N^2$  bypassing,  $N^2$  dependence check, wide fetch
    - More register file & memory traffic (ports)
- Can we do better?

## Better Alternative: Data-Level Parallelism

- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
    - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point “Vector registers”
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector addition, subtraction, multiply, etc.
    - Vector+Constant addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!

## Example Vector ISA Extensions

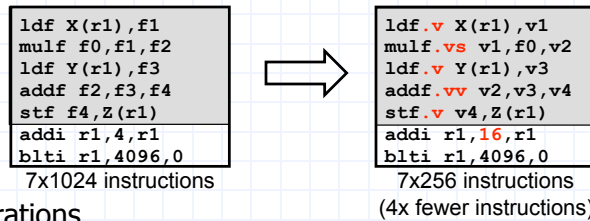
- Extend ISA with floating point (FP) vector storage ...
  - **Vector register**: fixed-size array of 32- or 64- bit FP elements
  - **Vector length**: For example: 4, 8, 16, 64, ...
- ... and example operations for vector length of 4
  - Load vector: `ldf.v X(r1), v1`

```
ldf X+0(r1), v1[0]  
ldf X+1(r1), v1[1]  
ldf X+2(r1), v1[2]  
ldf X+3(r1), v1[3]
```
  - Add two vectors: `addf.vv v1, v2, v3`

```
addf v1[i], v2[i], v3[i] (where i is 0,1,2,3)
```
  - Add vector to scalar: `addf.vs v1, f2, v3`

```
addf v1[i], f2, v3[i] (where i is 0,1,2,3)
```

## Example Use of Vectors – 4-wide



- Operations
  - Load vector: `ldf.v X(r1), v1`
  - Multiply vector to scalar: `mulf.vs v1, f2, v3`
  - Add two vectors: `addf.vv v1, v2, v3`
  - Store vector: `stf.v v1, X(r1)`
- Performance?
  - If CPI is one, 4x speedup
  - But, vector instructions don't always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

## Vector Datapath & Implementatoin

- Vector insn. are just like normal insn... only "wider"
  - Single instruction fetch (no extra  $N^2$  checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid  $N^2$  bypass problem)
  - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width
- Because they are just instructions...
  - ...superscalar execution of vector instructions is common
  - Multiple n-wide vector instructions per cycle

## Intel's SSE2/SSE3/SSE4...

- **Intel SSE2 (Streaming SIMD Extensions 2)** - 2001
  - 16 128bit floating point registers (`xmm0-xmm15`)
  - Each can be treated as 2x64b FP or 4x32b FP ("packed FP")
    - Or 2x64b or 4x32b or 8x16b or 16x8b ints ("packed integer")
    - Or 1x64b or 1x32b FP (just normal scalar floating point)
  - Original SSE: only 8 registers, no packed integer support
- Other vector extensions
  - AMD 3DNow!: 64b (2x32b)
  - PowerPC AltiVEC/VMX: 128b (2x64b or 4x32b)
- Looking forward for x86
  - Intel's "Sandy Bridge" will bring 256-bit vectors to x86
  - Intel's "Larrabee" graphics chip will bring 512-bit vectors to x86

## Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
- Some examples
  - Vector reduction (sum all elements of a vector)
  - Geometry processing: 4x4 translation/rotation matrices
  - Saturating (non-overflowing) subword add/sub: image processing
  - Byte asymmetric operations: blending and composition in graphics
  - Byte shuffle/permute: crypto
  - Population (bit) count: crypto
  - Max/min/argmax/argmin: video codec
  - Absolute differences: video codec
  - Multiply-accumulate: digital-signal processing

## Options for Using Vectors in Your Code

---

- Write in assembly
  - Ugh
- Use "intrinsic" functions and data types
  - For example: `_mm_mul_ps()` and `__m128` datatype
- Use a library someone else wrote
  - Let them do the hard work
  - Matrix and linear algebra packages
- Let the compiler do it (automatic vectorization)
  - GCC's `-ftree-vectorize` option
  - Doesn't yet work well for C/C++ code (old, very hard problem)