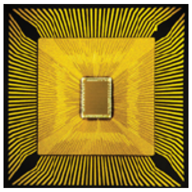
CS 61C: Great Ideas in Computer Architecture

The Flynn Taxonomy, Intel SIMD Instructions

Guest Lecturer: Alan Christopher

Neuromorphic Chips

Researchers at IBM and HRL laboratories are looking into building computer chips that attempt to mimic natural thought patterns in order to more effectively solve problems in AI.



http://www.technologyreview.com/featuredstory/522476/thinking-in-silicon/

Review of Last Lecture

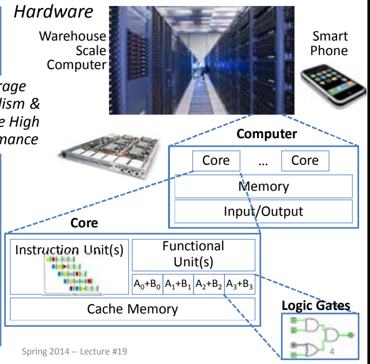
- Amdahl’s Law limits benefits of parallelization
- Request Level Parallelism
  - Handle multiple requests in parallel (e.g. web search)
- MapReduce Data Level Parallelism
  - Framework to divide up data to be processed in parallel
  - Mapper outputs intermediate key-value pairs
  - Reducer “combines” intermediate values with same key

Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search “Garcia”
- Parallel Threads
  Assigned to core
  e.g. lookup, ads
- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions
- Parallel Data We are here
  > 1 data item @ one time
  e.g. add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

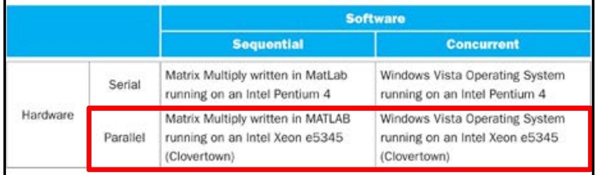
Leverage Parallelism & Achieve High Performance



Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: Loop Unrolling

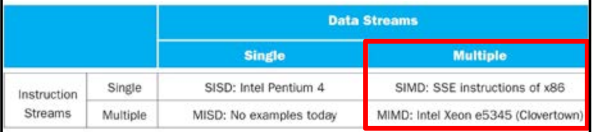
Hardware vs. Software Parallelism



		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn’s Taxonomy* is for parallel hardware

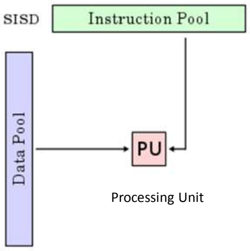
Flynn's Taxonomy



		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

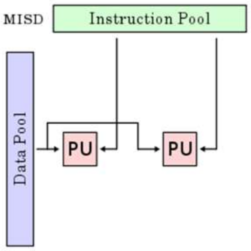
- SIMD and MIMD most commonly encountered today
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

Single Instruction/Single Data Stream



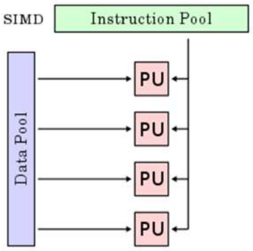
- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines

Multiple Instruction/Single Data Stream



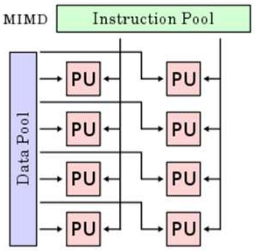
- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)
- MISD no longer commonly encountered, mainly of historical interest only

Single Instruction/Multiple Data Stream



- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)

Multiple Instruction/Multiple Data Stream



- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers

Agenda

- Flynn's Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: Loop Unrolling

Administrivia

- Midterm next Wednesday
  - May have 1 double sided 8.5’’ x 11’’ sheet of hand written notes
  - May also bring an unedited copy of the MIPS green sheet.
    - We will provide a copy if you forget yours
- Proj2 (MapReduce) to be released soon
  - Part 1 due date pushed later
  - Work in **partners**, 1 of you must know Java

Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: Loop Unrolling

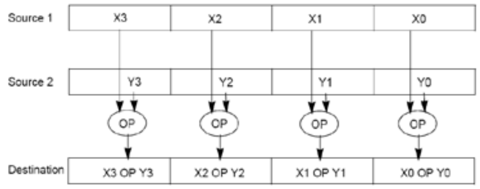
SIMD Architectures

- *Data-Level Parallelism (DLP):* Executing one operation on multiple data streams
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

  $y[i] := c[i] \times x[i],\ 0 \le i < n$
- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well

“Advanced Digital Media Boost”

- To improve performance, Intel’s SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - *SSE (Streaming SIMD Extension, Pentium III and beyond)*



Example: SIMD Array Processing

```
for each f in array:
    f = sqrt(f)
```

pseudocode

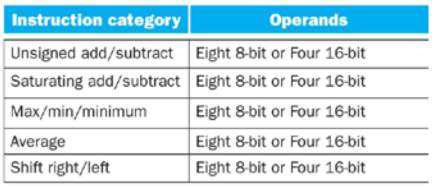
```
for each f in array {
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SISD

```
for every 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

SIMD

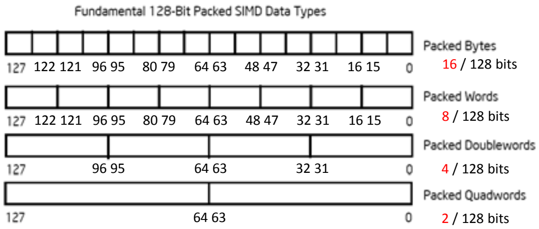
SSE Instruction Categories for Multimedia Support



Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

- Intel processors are CISC (complicated instrs)
- SSE-2+ supports wider data types to allow 16 × 8-bit and 8 × 16-bit operands

Intel Architecture SSE2+ 128-Bit SIMD Data Types



- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)

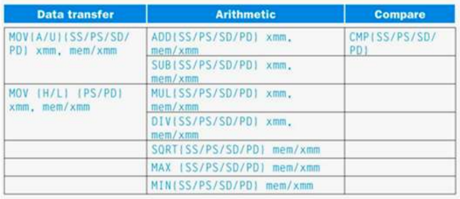
XMM Registers

- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously

127 0

XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

SSE/SSE2 Floating Point Instructions



Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

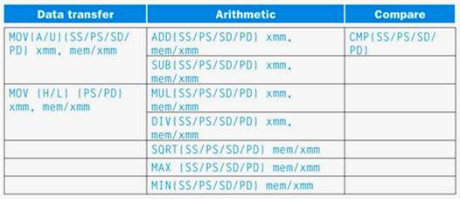
{SS} Scalar Single precision FP: **1** 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: **4** 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: **1** 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or **2** 64-bit operands in a 128-bit register

SSE/SSE2 Floating Point Instructions



Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

xmm: one operand is a 128-bit SSE2 register
mem/xmm: other operand is in memory or an SSE2 register
{A} 128-bit operand is aligned in memory
{U} means the 128-bit operand is unaligned in memory
{H} means move the high half of the 128-bit operand
{L} means move the low half of the 128-bit operand

Example: Add Single Precision FP Vectors

Computation to be performed:

```
  vec_res.x = v1.x + v2.x;
  vec_res.y = v1.y + v2.y;
  vec_res.z = v1.z + v2.z;
  vec_res.w = v1.w + v2.w;
```

move from mem to XMM register, memory **a**ligned, **p**acked **s**ingle precision

add from mem to XMM register, **p**acked **s**ingle precision

move from XMM register to mem, memory **a**ligned, **p**acked **s**ingle precision

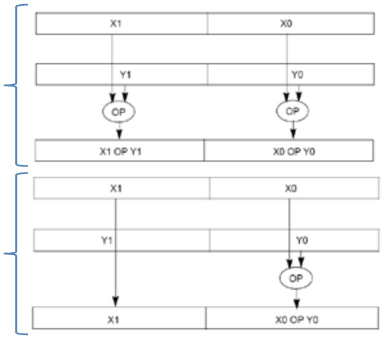
SSE Instruction Sequence:

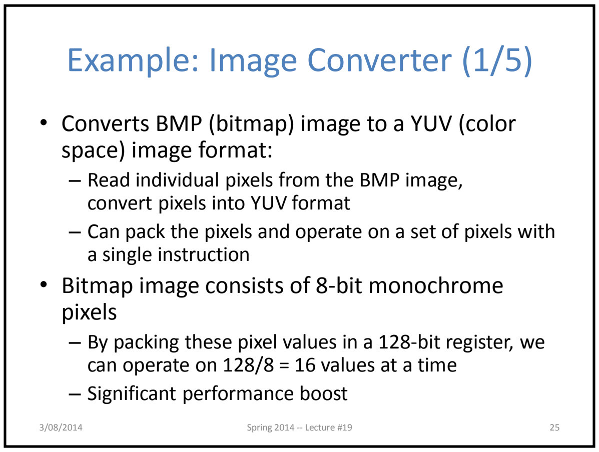
```
movaps  address-of-v1, %xmm0
  // v1.w | v1.z | v1.y | v1.x -> xmm0
addps   address-of-v2, %xmm0
  // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
  -> xmm0
movaps  %xmm0, address-of-vec_res
```

Packed and Scalar Double-Precision Floating-Point Operations

Packed Double (PD)

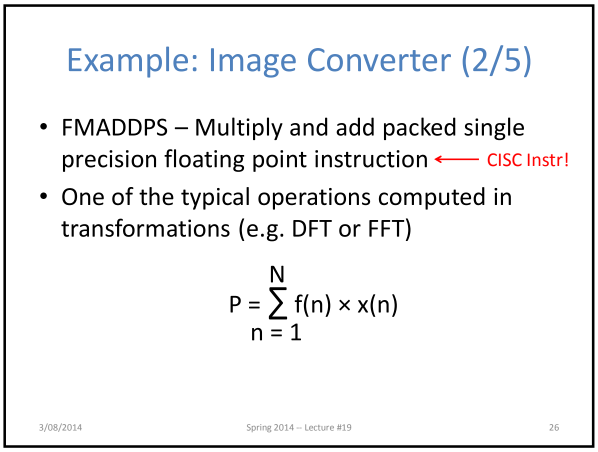
Scalar Double (SD)





Example: Image Converter (1/5)

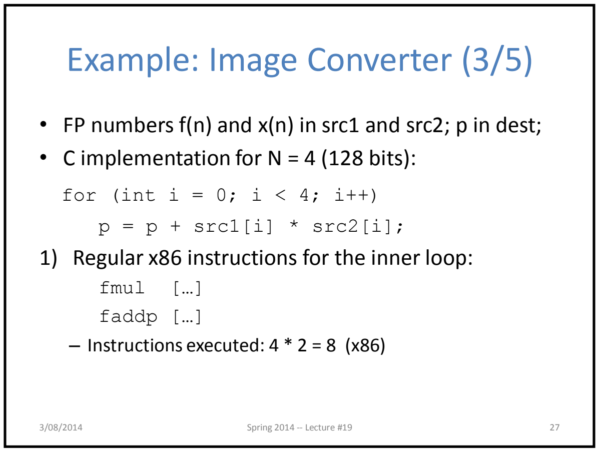
- Converts BMP (bitmap) image to a YUV (color space) image format:
  - Read individual pixels from the BMP image, convert pixels into YUV format
  - Can pack the pixels and operate on a set of pixels with a single instruction
- Bitmap image consists of 8-bit monochrome pixels
  - By packing these pixel values in a 128-bit register, we can operate on 128/8 = 16 values at a time
  - Significant performance boost



Example: Image Converter (2/5)

- FMADDPS – Multiply and add packed single precision floating point instruction ⟵ CISC Instr!
- One of the typical operations computed in transformations (e.g. DFT or FFT)

$$P = \sum_{n=1}^{N} f(n) \times x(n)$$



Example: Image Converter (3/5)

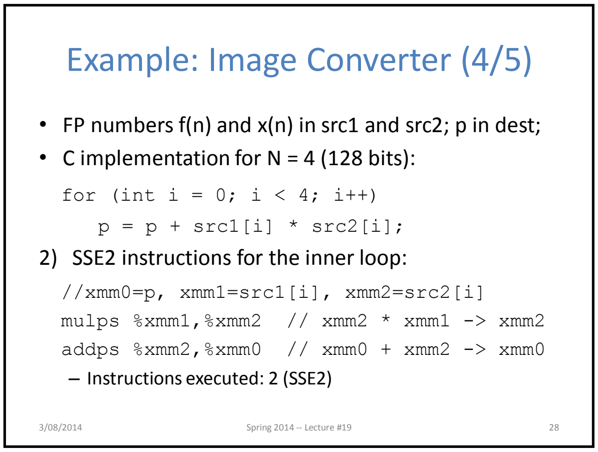
- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):

```
for (int i = 0; i < 4; i++)
   p = p + src1[i] * src2[i];
```

1) Regular x86 instructions for the inner loop:

```
   fmul   […]
   faddp  […]
```

  - Instructions executed: 4 * 2 = 8 (x86)



Example: Image Converter (4/5)

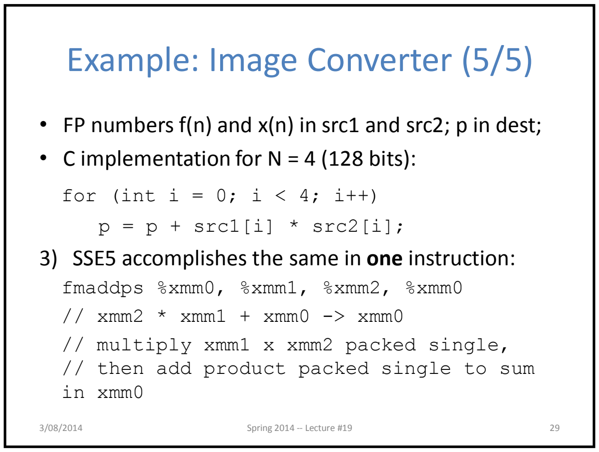
- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):

```
for (int i = 0; i < 4; i++)
   p = p + src1[i] * src2[i];
```

2) SSE2 instructions for the inner loop:

```
//xmm0=p, xmm1=src1[i], xmm2=src2[i]
mulps %xmm1,%xmm2   // xmm2 * xmm1 -> xmm2
addps %xmm2,%xmm0   // xmm0 + xmm2 -> xmm0
```

  - Instructions executed: 2 (SSE2)



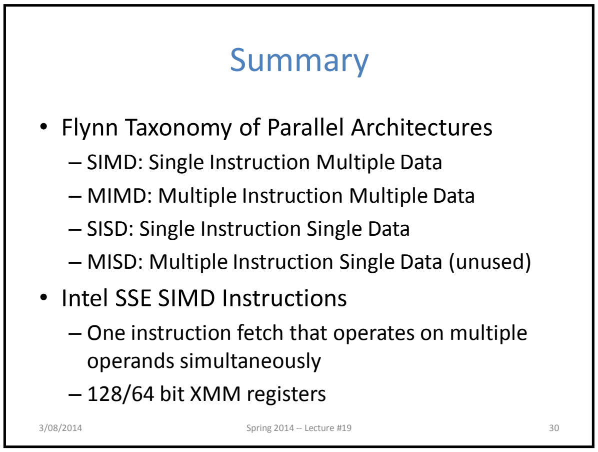
Example: Image Converter (5/5)

- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):

```
for (int i = 0; i < 4; i++)
   p = p + src1[i] * src2[i];
```

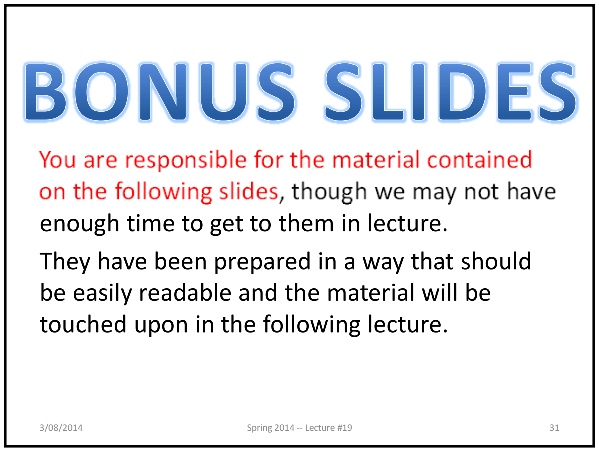
3) SSE5 accomplishes the same in **one** instruction:

```
fmaddps %xmm0, %xmm1, %xmm2, %xmm0
// xmm2 * xmm1 + xmm0 -> xmm0
// multiply xmm1 x xmm2 packed single,
// then add product packed single to sum
in xmm0
```



Summary

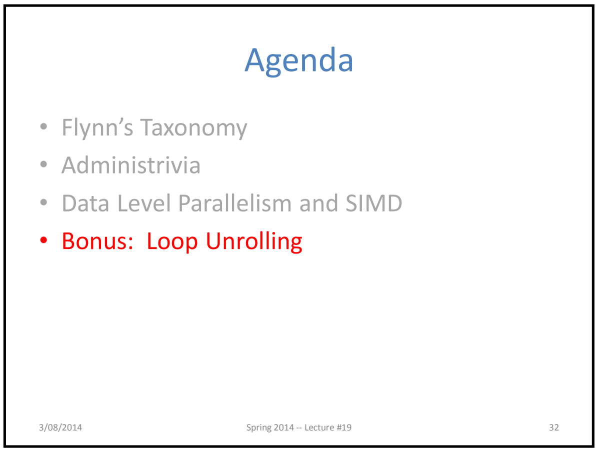
- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - SISD: Single Instruction Single Data
  - MISD: Multiple Instruction Single Data (unused)
- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 128/64 bit XMM registers



BONUS SLIDES

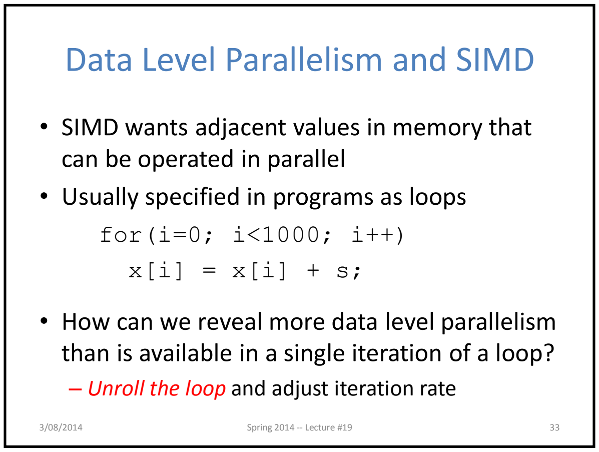
You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.



Agenda

- Flynn's Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: Loop Unrolling

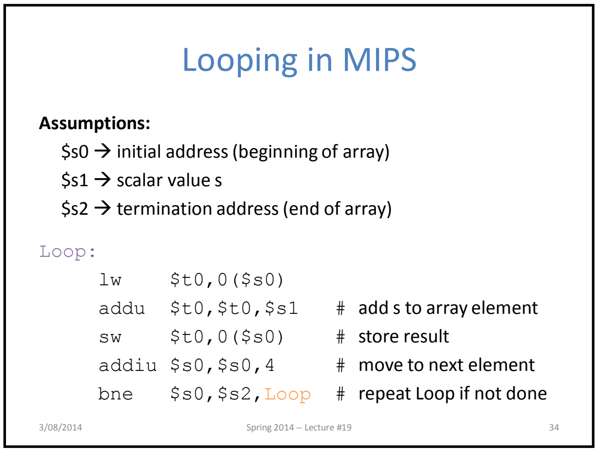


Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate

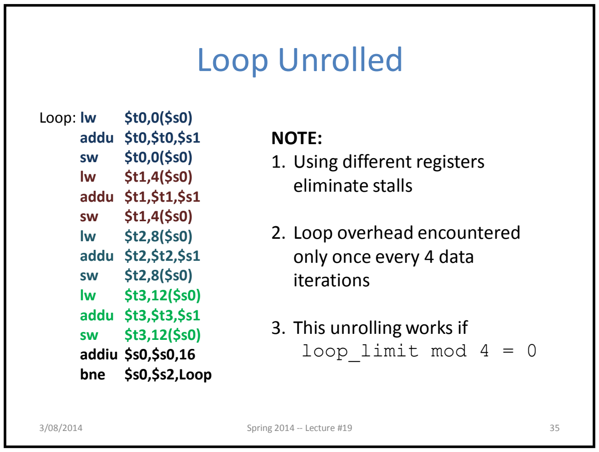


Looping in MIPS

Assumptions:

$s0 → initial address (beginning of array)
$s1 → scalar value s
$s2 → termination address (end of array)

```
Loop:
        lw     $t0,0($s0)
        addu   $t0,$t0,$s1     # add s to array element
        sw     $t0,0($s0)      # store result
        addiu  $s0,$s0,4       # move to next element
        bne    $s0,$s2,Loop    # repeat Loop if not done
```

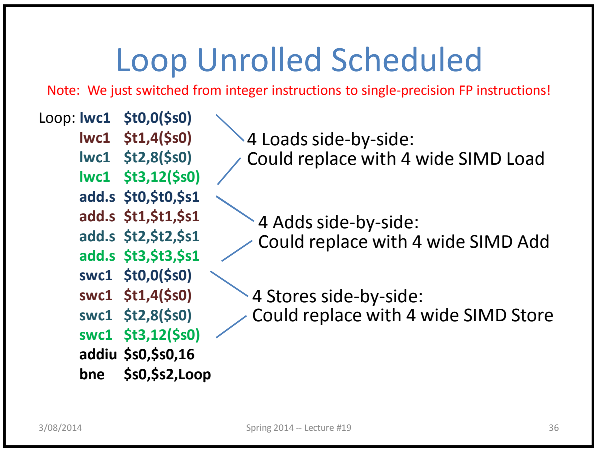


Loop Unrolled

```
Loop: lw    $t0,0($s0)
      addu  $t0,$t0,$s1
      sw    $t0,0($s0)
      lw    $t1,4($s0)
      addu  $t1,$t1,$s1
      sw    $t1,4($s0)
      lw    $t2,8($s0)
      addu  $t2,$t2,$s1
      sw    $t2,8($s0)
      lw    $t3,12($s0)
      addu  $t3,$t3,$s1
      sw    $t3,12($s0)
      addiu $s0,$s0,16
      bne   $s0,$s2,Loop
```

NOTE:

1. Using different registers eliminate stalls
2. Loop overhead encountered only once every 4 data iterations
3. This unrolling works if `loop_limit mod 4 = 0`



Loop Unrolled Scheduled

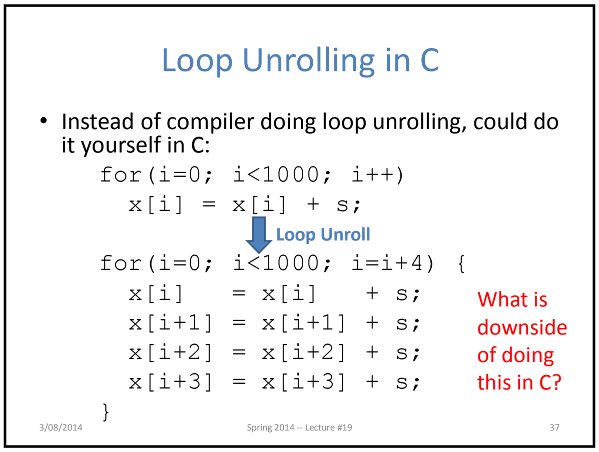
Note: We just switched from integer instructions to single-precision FP instructions!

```
Loop: lwc1  $t0,0($s0)
      lwc1  $t1,4($s0)
      lwc1  $t2,8($s0)
      lwc1  $t3,12($s0)
      add.s $t0,$t0,$s1
      add.s $t1,$t1,$s1
      add.s $t2,$t2,$s1
      add.s $t3,$t3,$s1
      swc1  $t0,0($s0)
      swc1  $t1,4($s0)
      swc1  $t2,8($s0)
      swc1  $t3,12($s0)
      addiu $s0,$s0,16
      bne   $s0,$s2,Loop
```

4 Loads side-by-side:
Could replace with 4 wide SIMD Load

4 Adds side-by-side:
Could replace with 4 wide SIMD Add

4 Stores side-by-side:
Could replace with 4 wide SIMD Store



Loop Unrolling in C

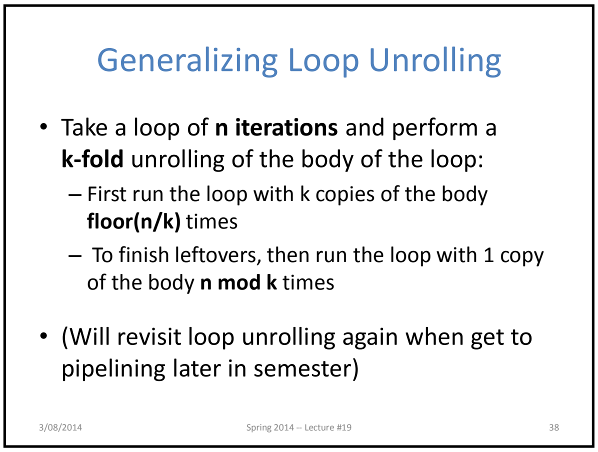
- Instead of compiler doing loop unrolling, could do it yourself in C:

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

Loop Unroll

```
for(i=0; i<1000; i=i+4) {
  x[i]   = x[i]   + s;
  x[i+1] = x[i+1] + s;
  x[i+2] = x[i+2] + s;
  x[i+3] = x[i+3] + s;
}
```

What is downside of doing this in C?



Generalizing Loop Unrolling

- Take a loop of **n iterations** and perform a **k-fold** unrolling of the body of the loop:
  - First run the loop with k copies of the body **floor(n/k)** times
  - To finish leftovers, then run the loop with 1 copy of the body **n mod k** times
- (Will revisit loop unrolling again when get to pipelining later in semester)