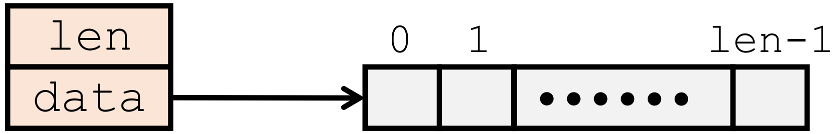
Optimizing Program Performance

CMPU 224 – Computer Organization
Jason Waterman

Benchmark Example: Abstract Data Type for Vectors

```
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;
```



- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

```
/* retrieve vector element and store at val */
/* return 1 if successful, 0 otherwise */
int get_vec_element(*vec v, size_t idx, data_t *val) {
    if (idx >= v->len){
        return 0;
    }
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

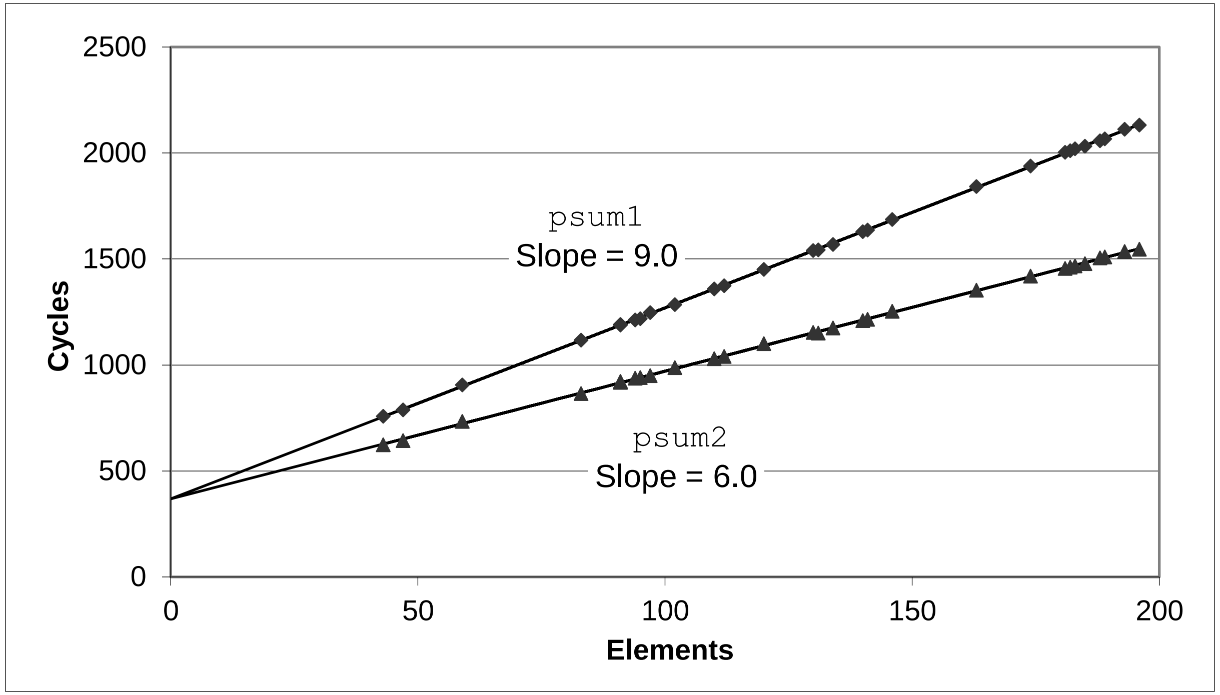
```
void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
      data_t val;
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line



Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest) {
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Eliminating Loop Inefficiencies

- Move call to `vec_length` outside of loop
- This is called code motion

```
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1	**10.12**	**10.12**	**10.17**	**11.14**
Combine2	**7.02**	**9.03**	**9.02**	**11.03**

Reducing Procedure Calls

- `get_vec_start` returns the starting address of the data array
  - This breaks the abstraction barrier!
- Remove call to `get_vec_element`
- We are no longer doing bounds checking
- No effect on performance
  - Other operations are the bottleneck
- Will help us later when we remove these bottlenecks

```
/* Direct access to vector data*/
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine2	7.02	9.03	9.02	11.03
Combine3	7.17	9.02	9.02	11.03

Eliminating Unneeded Memory References

- Disassembly of inner loop for integer addition
- One memory read and one memory write per cycle
- But `*dest` doesn't need to be updated till the end of the loop

```
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```
.L3: # i in %rdx, length in %rbp, data in %rax
cmpq %rbp, %rdx        # i:length
jge .L1                # jump to end of loop
movq (%rax,%rdx,8), %rcx # read data[i]
addq %rcx, (%rbx)      # write *dest
addq $1, %rdx          # i++
jmp .L3
.L1:
```

Accumulate result in temporary

```
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```
.L3: # i in %rdx, data in %rax
cmpq %rbp, %rdx    # i:length
jge .L1            # jump to end of loop
movq (%rax,%rdx,8), %rcx # read data[i]
addq %rcx, (%rbx) # write *dest
addq $1, %rdx      # i++
jmp .L3
.L1:
```

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

```
L3: # data in %rax, i in %rdx, acc in %rcx
cmpq %rbp, %rdx    # i:length
jge .L1            # jump to end of loop
addq (%rax,%rdx,8), %rcx # acc += data[i]
addq $1, %rdx      # i++
jmp .L3
.L1:
addq %rcx, (%rbx) # write *dest
```

- Reduce to only one memory read per element using a temporary variable

Accumulate result in temporary

```
void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

- Why doesn't the compiler perform this transformation automatically?
  - **Memory aliasing**
  - The `dest` could be one of the elements in the vector (e.g., the last element)
    - `combine3` and `combine4` could have different results in this case
  - Using a temporary register tells the compiler not to check for memory aliasing

Effect of Basic Optimizations

- 4x to 18x improvement over original unoptimized code
- To seek better performance, we must consider optimizations that exploit the *microarchitecture* of the processor
  - Code tuned for a specific processor

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v)

    data_t = acc;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 Unoptimized	22.68	20.02	19.98	20.18
Combine3	7.17	9.02	9.02	11.03
Combine4	1.27	3.01	3.01	5.01

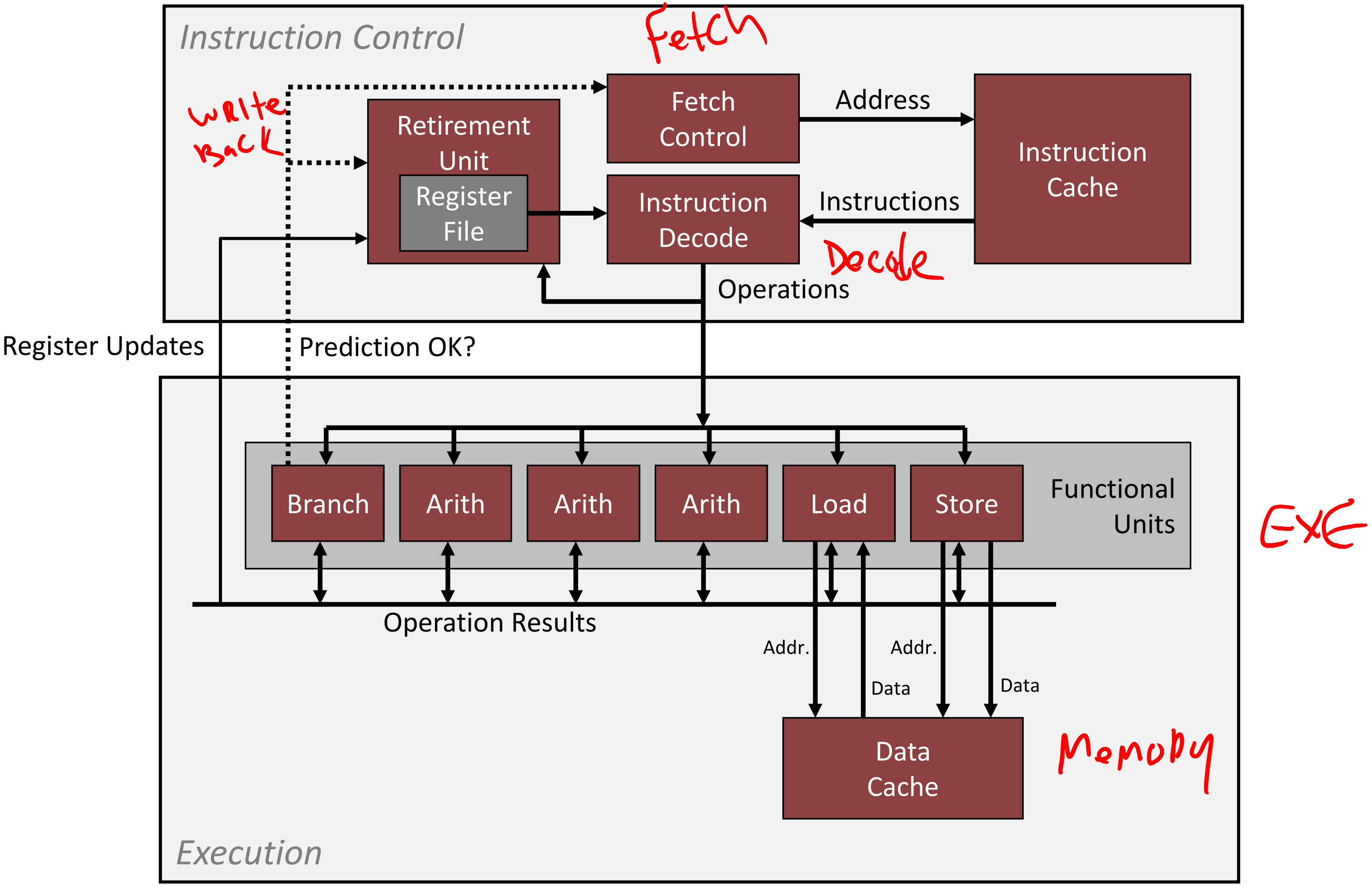
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations

Superscalar Processor

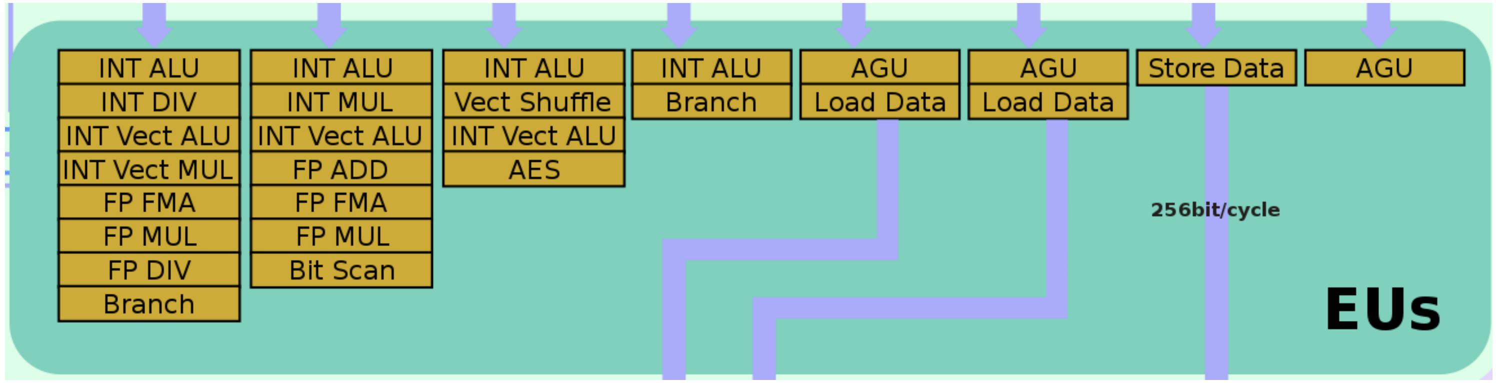
- **Superscalar processors** can issue and execute *multiple instructions in one cycle*
- Most modern CPUs are superscalar
  - Intel: since Pentium (1993)
- Instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
- Benefit: without programming effort, superscalar processor can take advantage of a program's *instruction level parallelism*

Modern CPU Design



Haswell CPU

- 8 Total Functional Units
- Multiple instructions can execute in parallel

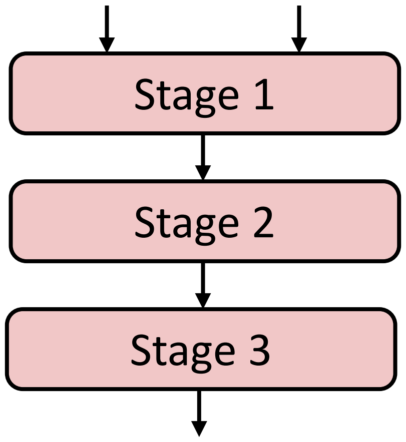


Some instructions take > 1 cycle, but can be pipelined

Instruction	***Latency***	***Cycles/Issue***
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



Time							
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

p1 p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```
.L519:                              # Loop:
   imull  (%rax,%rdx,4), %ecx      # t = t * d[i]
   addq   $1, %rdx                 # i++
   cmpq   %rdx, %rbp               # Compare length:i
   jg     .L519                    # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Loop Unrolling (2x1)

- Perform 2x more useful work per iteration

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

```
x = (x OP d[i]) OP d[i+1];
```

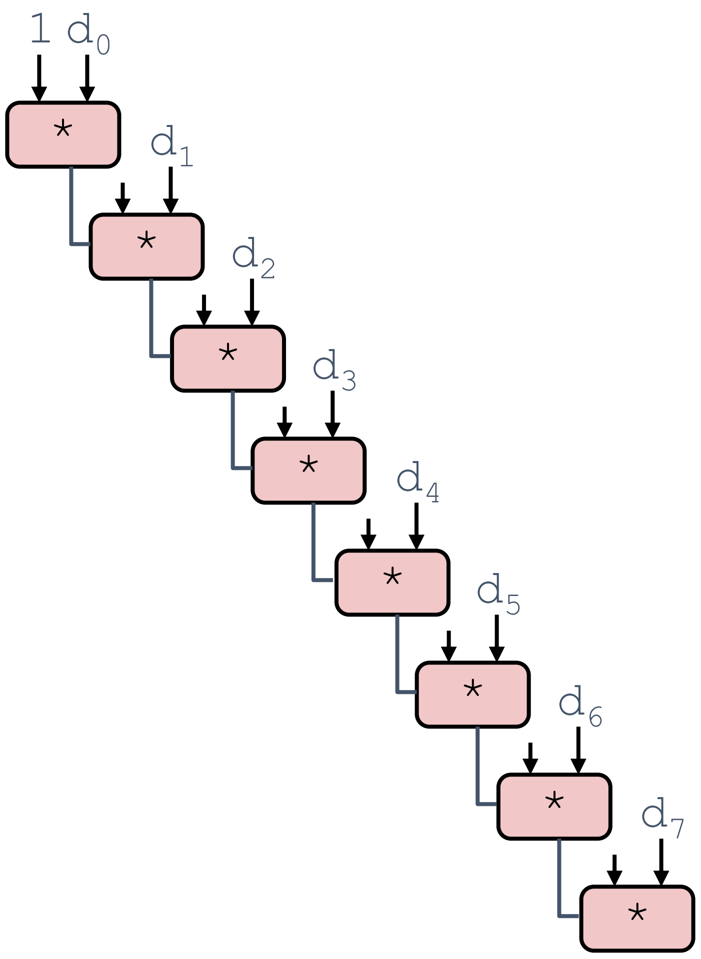
- Reduces overhead to integer add
  - Achieves latency bound
- Others don't improve. *Why?*
  - Still sequential dependency

Combine4 = Serial Computation (OP = *)

- Computation (length=8)

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP



Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- Can this change the result of the computation?
- Yes, for floating point numbers. *Why?*
  - Floating point numbers are not associative in all cases!

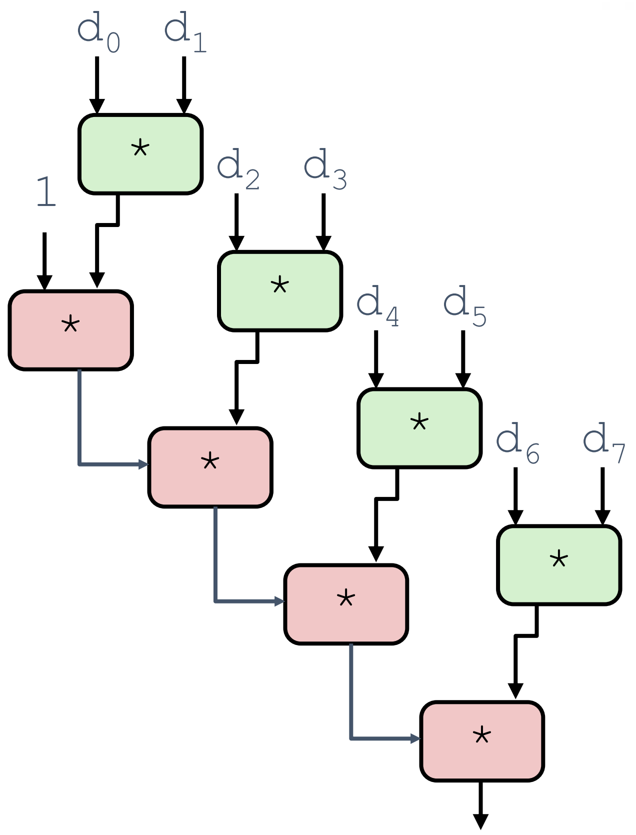
Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

2 func. units for FP *
2 func. units for load



Loop Unrolling with Separate Accumulators (2x2)

- Different form of reassociation

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

Effect of Separate Accumulators

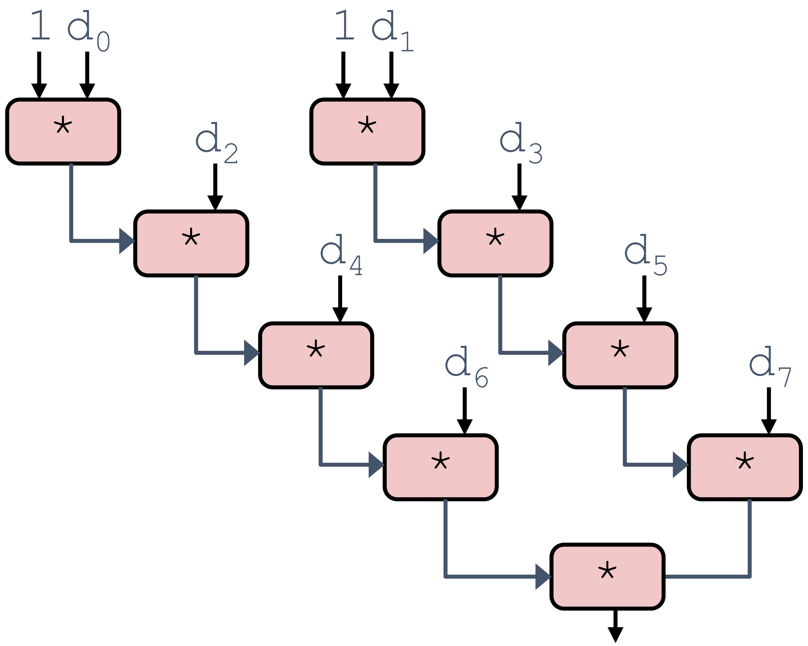
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- 2x speedup (over unroll2x1) for Int *, FP +, FP *
- Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent “streams” of operations

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Double *

- Case
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50 (Issue: 1, Capacity 2)

Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Unrolling & Accumulating: Int +

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 0.50

FP *	**Unrolling Factor L**							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

Factors Limiting Performance

- Why where there diminishing returns for loop unrolling and association?
  - Can't exceed the parallelism of the functional units
  - Register spilling
    - We only have a fixed number of registers that can hold temporary values in memory
    - Extra values will be stored on the stack (in memory)
- Mispredicted branches
  - Pipelined processors must guess which way a branch will go
  - If wrong, must discard the incorrect instructions and start again
  - Converting code to use conditional moves instead of branching can help
    - Good if branching is unpredictable
  - Mostly not a concern as branch prediction is very accurate

Getting High Performance

- Good compiler and flags
- Don’t do anything silly
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- Tune code for machine
  - Exploit instruction-level parallelism
  - Make code cache friendly

End of the Semester Logistics

- Cache Assignment due: Sunday May 8th 11:59pm
- Other Labs and Assignments
  - Check to make sure you have submitted your assignments
  - If not, come talk to me
- Final: Friday May 13th 9:00 AM SP 309 (this room)
  - Review next week
  - Practice problems will be posted to the course website
- CEQs
  - https://go.vassar.edu/course/evals