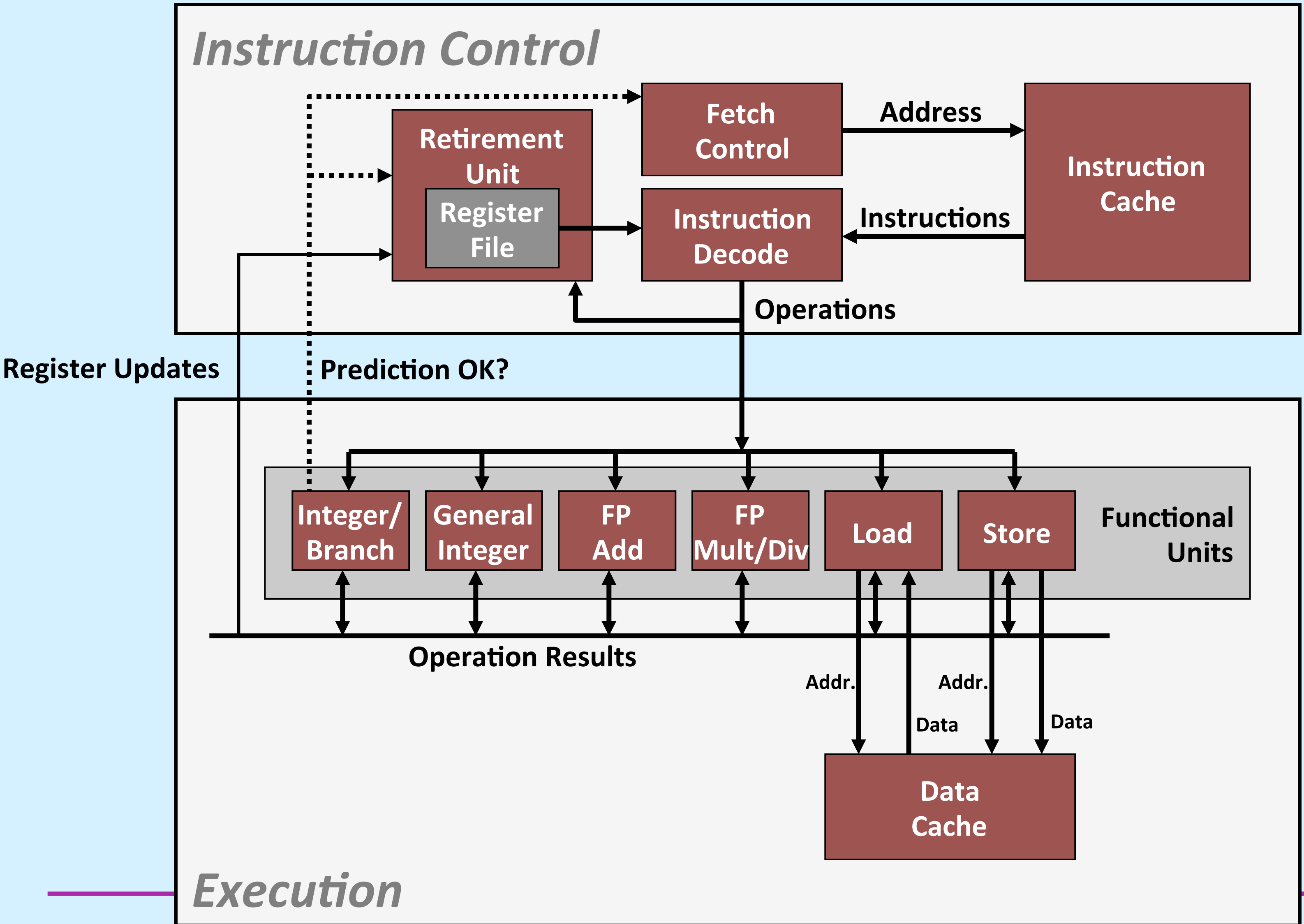
CS 33

Architecture and Optimization (2)

Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Modern CPU Design



Superscalar Processor

- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***
  - instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
    - » instructions may be executed *out of order*
- Benefit: without programming effort, superscalar processors can take advantage of the ***instruction-level parallelism*** that most programs have
- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro (1995)

Multiple Operations per Instruction

- addl %eax, %edx
  - a single operation
- addl %eax, 4(%edx)
  - three operations
    - » load value from memory
    - » add to it the contents of %eax
    - » store result in memory

Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Instruction-Level Parallelism

- `addl 4(%eax), %eax`
  `addl %ebx, %edx`
  - can be executed simultaneously: completely independent
- `addl 4(%eax), %ebx`
  `addl %ebx, %edx`
  - can also be executed simultaneously, but some coordination is required

Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Out-of-Order Execution

- ```
  movss   (%rbp), %xmm0
  mulss   (%rax, %rdx, 4), %xmm0
  movss   %xmm0, (%rbp)
  addq    %r8d, %r9d
  imulq   %rcx, %r12d
  addq    $1, %rdx
  ```

these can be executed without waiting for the first three to finish

Copyright © 2015 Thomas W. Doeppner. All rights reserved.
  ```

Speculative Execution

```
80489f3:    movl    $0x1,%ecx
80489f8:    xorl    %edx,%edx
80489fa:    cmpl    %esi,%edx
80489fc:    jnl     8048a25
80489fe:    movl    %esi,%esi
8048a00:    imull   (%eax,%edx,4),%ecx
```

perhaps execute these instructions

Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Nehalem CPU

- **Multiple instructions can execute in parallel**

  1 load, with address computation

  1 store, with address computation

  2 simple integer (one may be branch)

  1 complex integer (multiply/divide)

  1 FP Multiply

  1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Add	1	.33
Integer Multiply	3	1
Integer/Long Divide	11–21	11–21
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	10–23	10–23

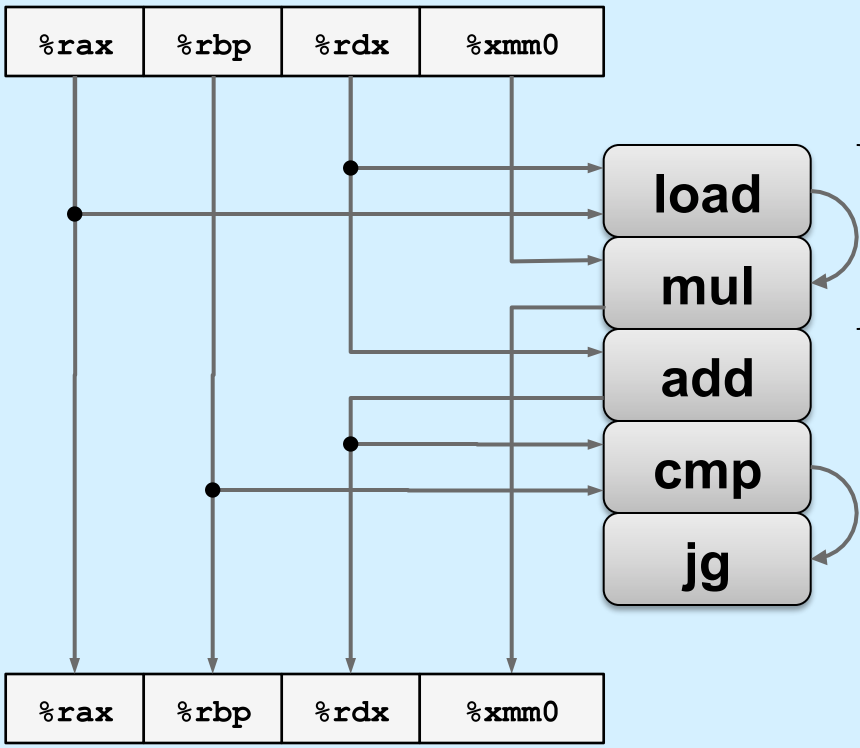
x86-64 Compilation of Combine4

- **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
```

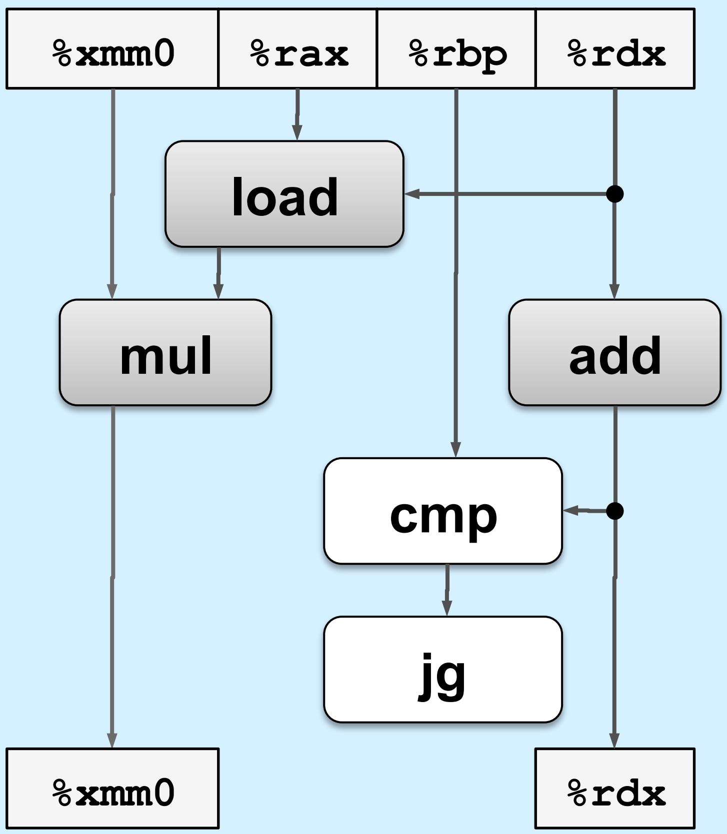
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

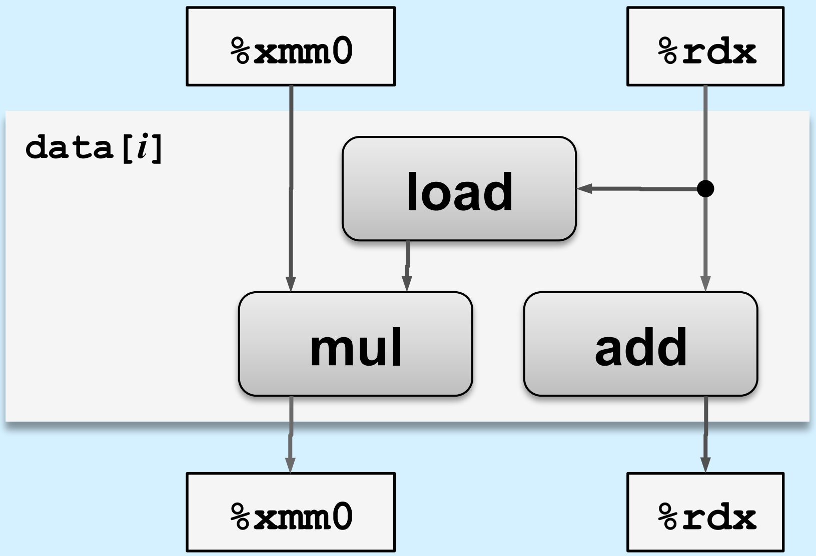
Inner Loop



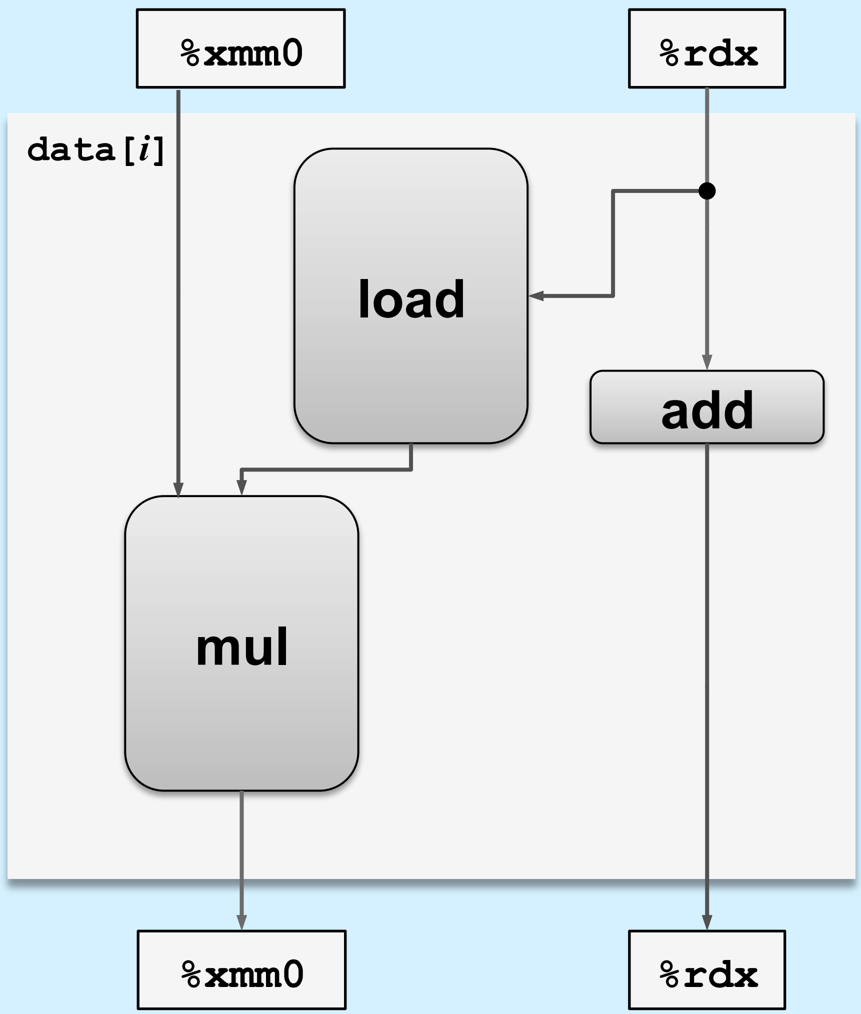

```
mulss (%rax,%rdx,4), %xmm0
addq $1,%rdx
cmpq %rdx,%rbp
jg loop
```

Data-Flow Graphs of Inner Loop



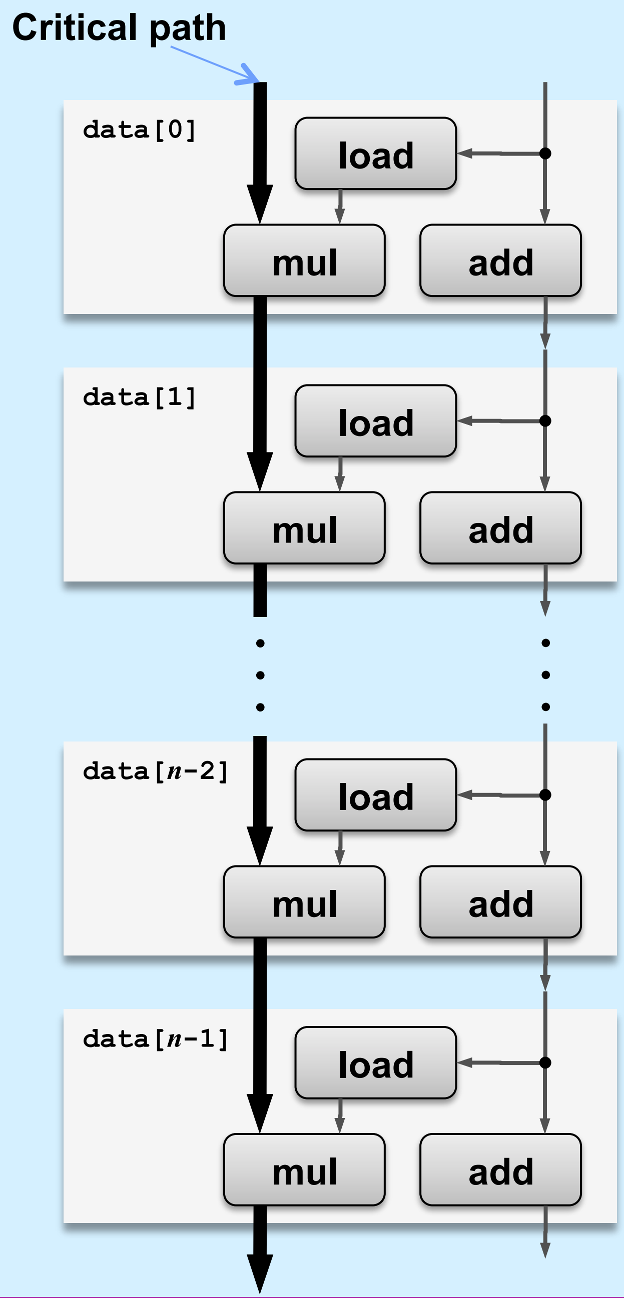


Relative Execution Times

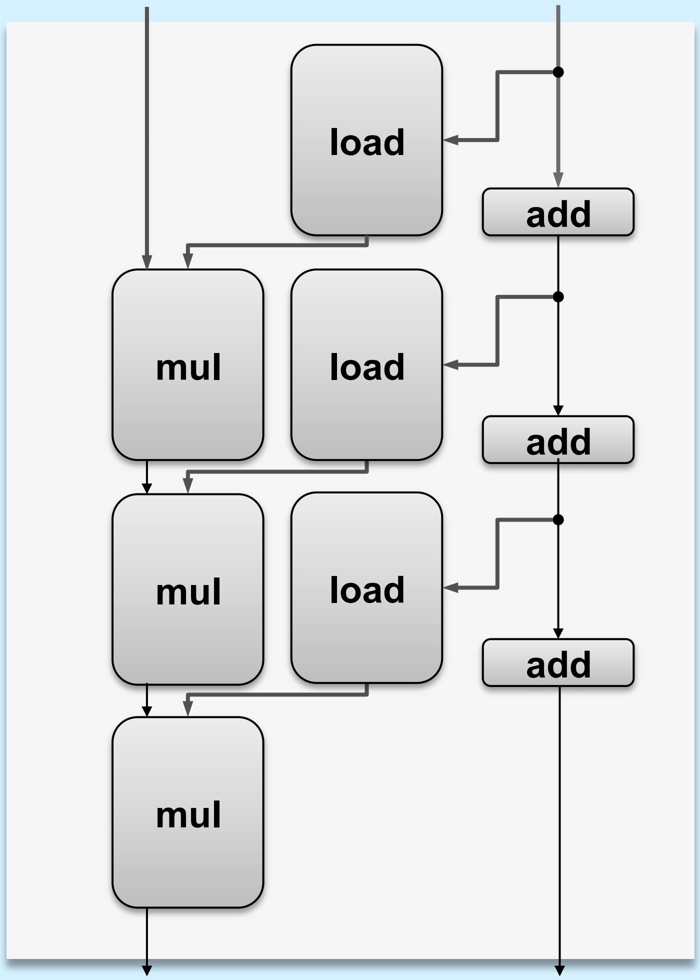


 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Data Flow Over Multiple Iterations

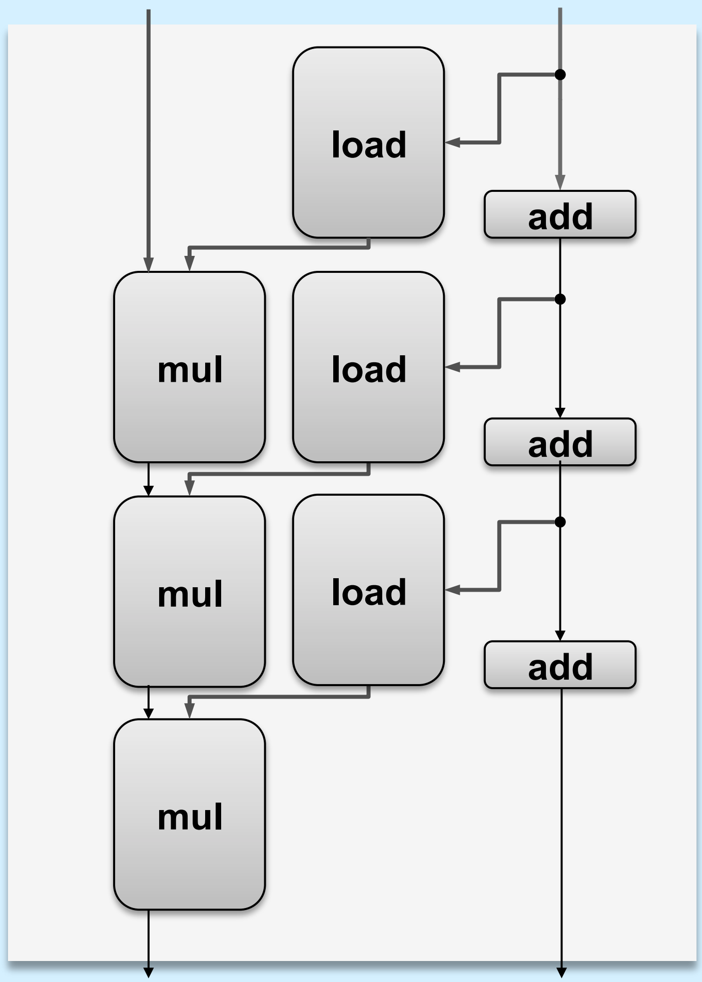


Pipelined Data-Flow Over Multiple Iterations



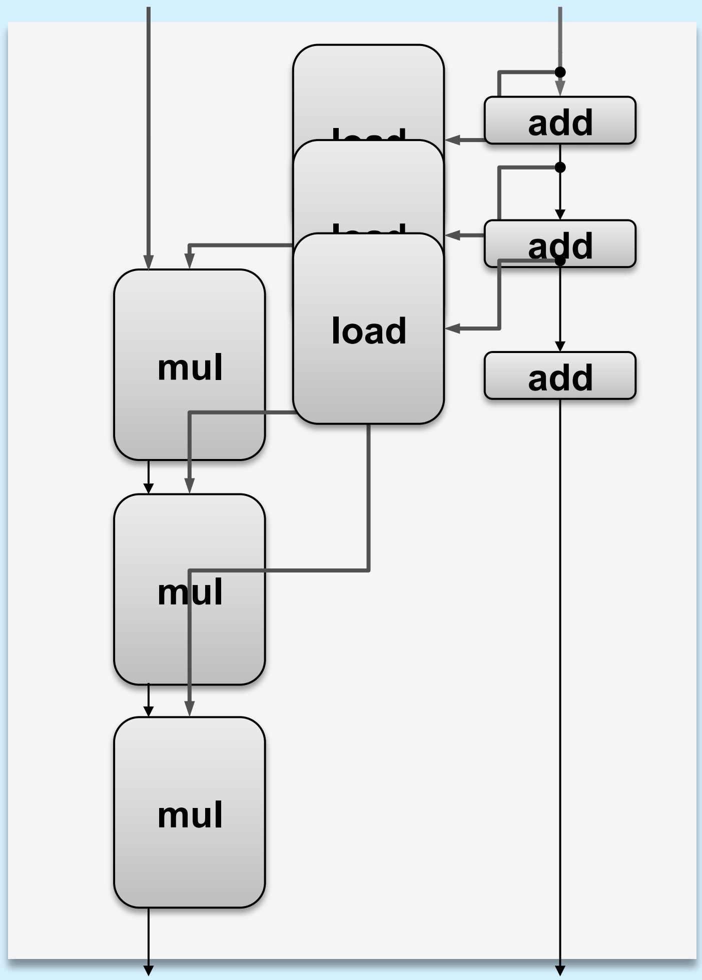
 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Pipelined Data-Flow Over Multiple Iterations



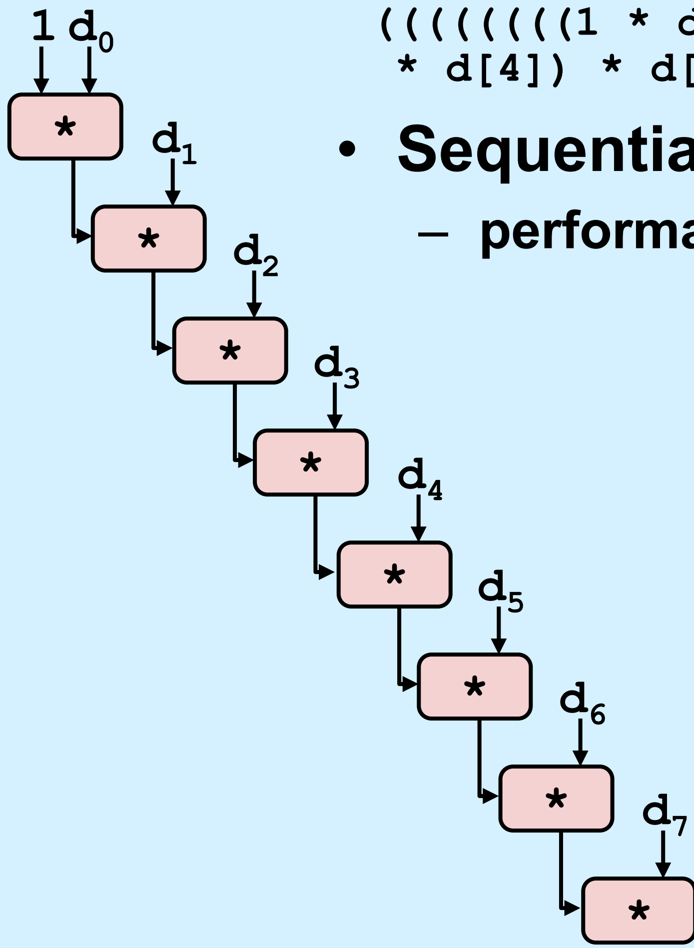
 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Pipelined Data-Flow Over Multiple Iterations



 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

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

```
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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;
}
```

- **Perform 2x more useful work per iteration**

Loop Unrolling

Quiz 1

Does it speed things up?

a) yes
b) no

```
void unroll2x(vec_ptr_t v
{
    int length = vec_leng
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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;
}
```

- **Perform 2x more useful work per iteration**

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

- **Helps integer multiply**
  - below latency bound
  - compiler does clever optimization
- **Others don't improve. *Why?***
  - still sequential dependency

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

Loop Unrolling with Reassociation

```
void unroll2xra(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int 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 FP. *Why?***

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

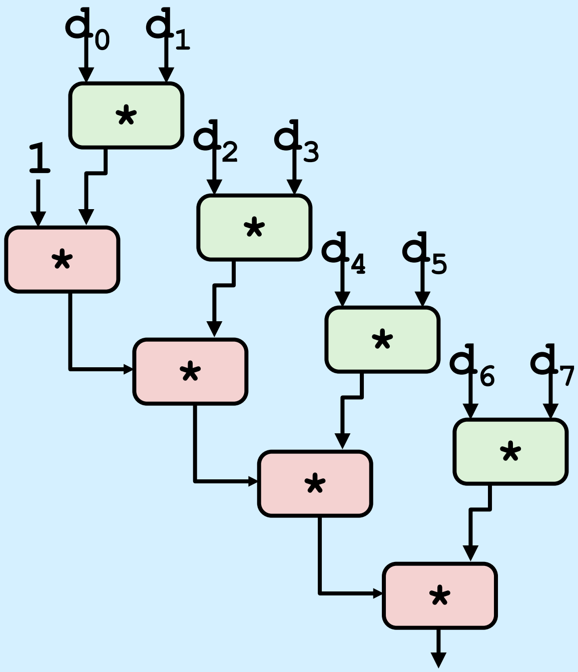
- Nearly 2x speedup for int *, FP +, FP *
  – reason: breaks sequential dependency

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

  – why is that? (next slide)

Reassociated Computation

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



- **What changed:**
  - ops in the next iteration can be started early (no dependency)
- **Overall Performance**
  - N elements, D cycles latency/op
  - should be (N/2+1)*D cycles: **CPE = D/2**
  - measured CPE slightly worse for FP mult

Loop Unrolling with Separate Accumulators

```
void unroll2xp2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int 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;
}
```

- Different form of reassociation

Effect of Separate Accumulators

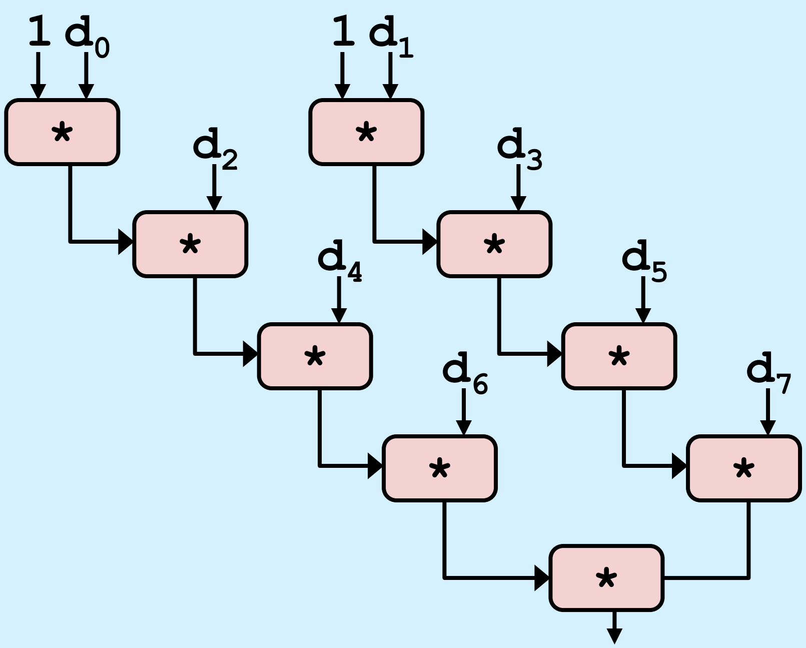
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Unroll 2x parallel 2x	1.5	1.5	1.5	2.5
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	1.0	1.0	1.0	1.0

- 2x speedup (over unroll2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

```
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
- **Overall Performance**
  - N elements, D cycles latency/op
  - should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Quiz 2

With 3 accumulators there will be 3 independent streams of instructions; with 4 accumulators 4 independent streams of instructions, etc.

Thus with n accumulators we can have a speedup of O(n), as long as n is no greater than the number of available registers.

a) true

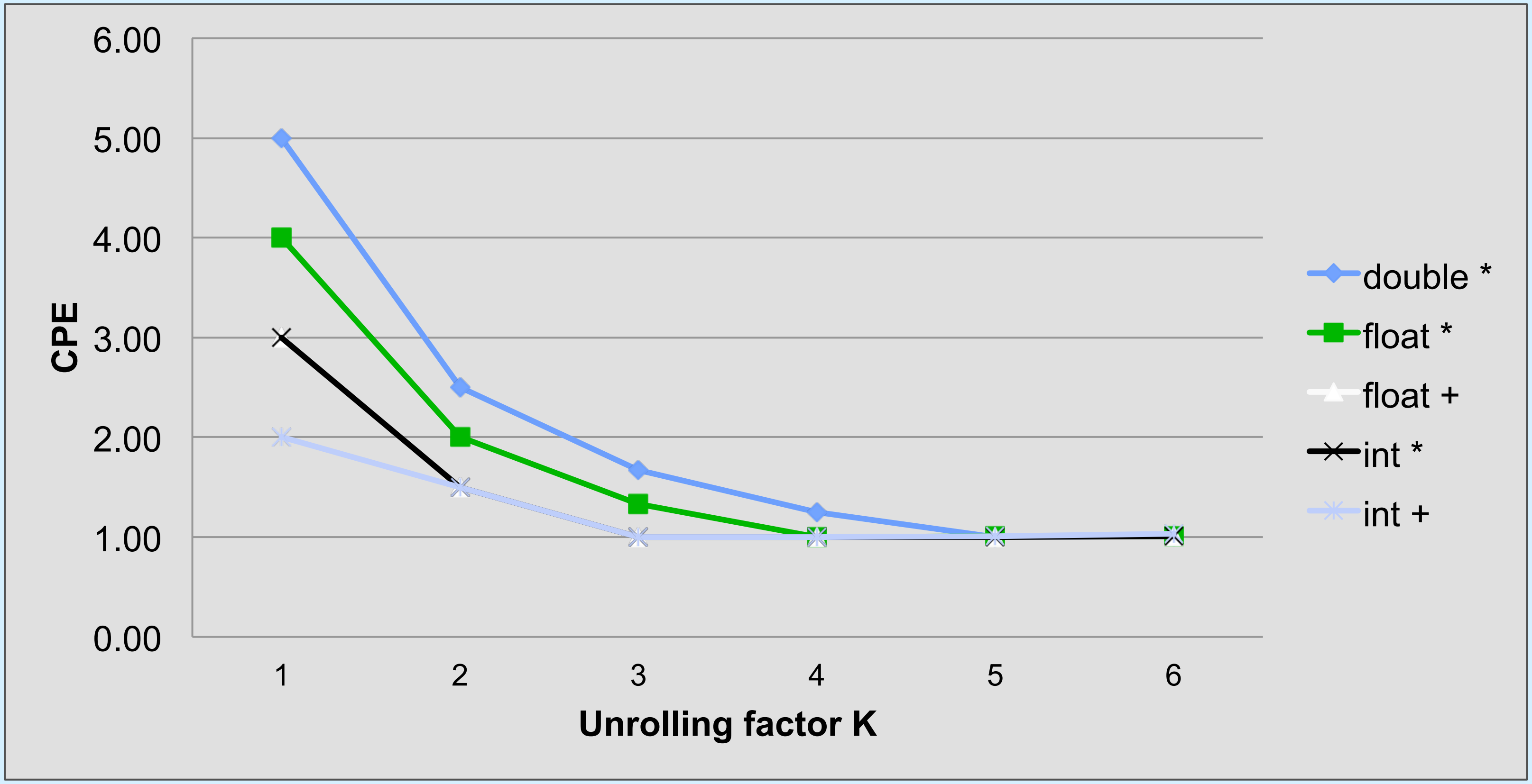
b) false

Copyright © 2015 Thomas W. Doeppner. All rights reserved.

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

Performance



- **K-way loop unrolling with K accumulators**

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar optimum	1.00	1.00	1.00	1.00
Vector optimum	0.25	0.53	0.53	0.57
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	1.00	1.00	1.00	1.00
Vec throughput bound	0.25	0.50	0.50	0.50

- **Make use of SSE Instructions**
  – parallel operations on multiple data elements

What About Branches?

- Challenge
  - **instruction control unit** must work well ahead of **execution unit** to generate enough operations to keep EU busy

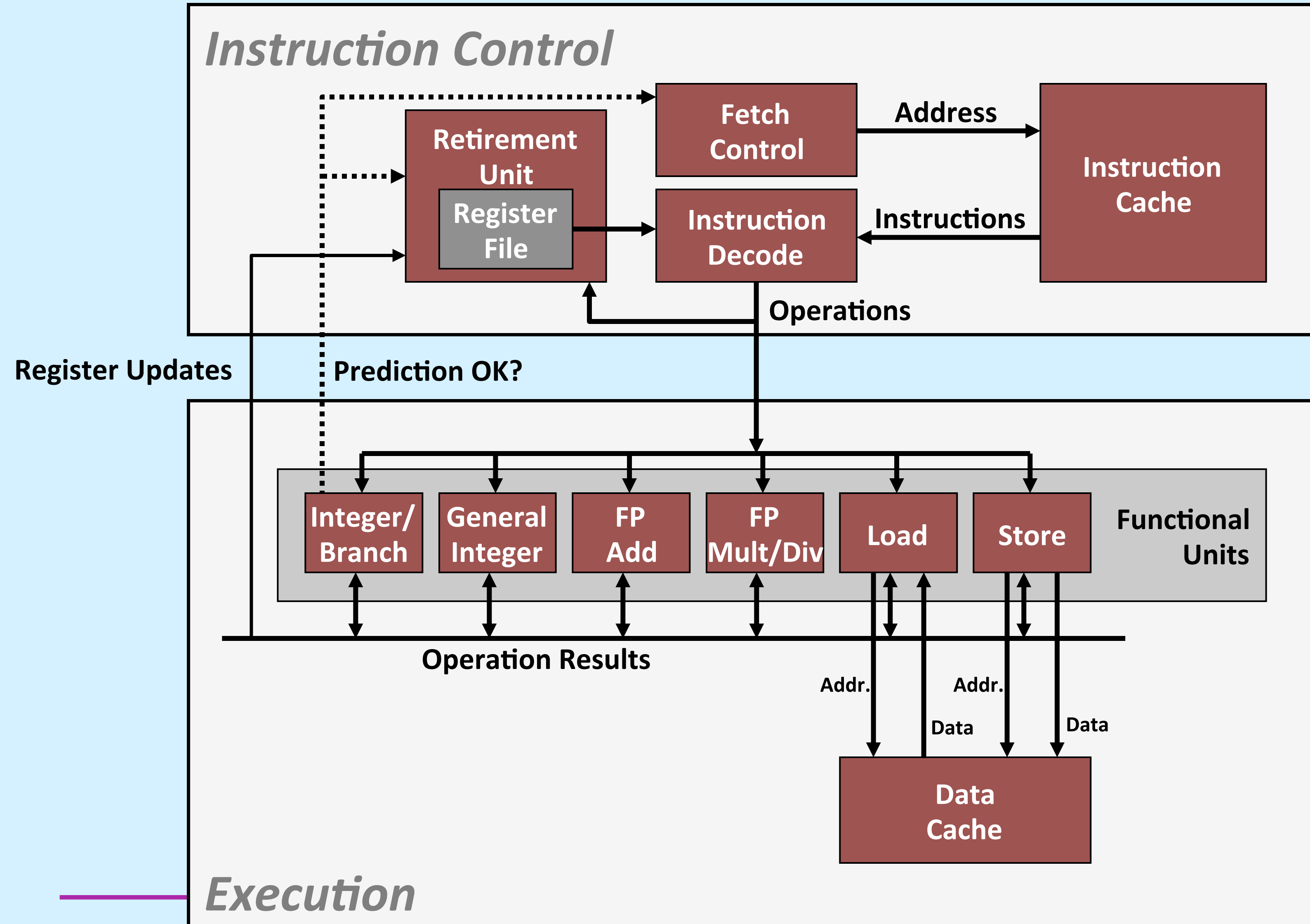
```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

Executing

How to continue?

  - when it encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  – branch taken: transfer control to branch target
  – branch not-taken: continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%eax,%edx,4),%ecx
```

Branch not-taken

Branch taken

```
8048a25:  cmpl    %edi,%edx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%ebp),%eax
8048a2c:  leal    0xffffffe8(%ebp),%esp
8048a2f:  movl    %ecx,(%eax)
```

Branch Prediction

- **Idea**
  - guess which way branch will go
  - begin executing instructions at predicted position
    - » but don't actually modify register or memory data

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
. . .
```

Predict taken

```
8048a25:  cmpl    %edi,%edx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%ebp),%eax
8048a2c:  leal    0xffffffe8(%ebp),%esp
8048a2f:  movl    %ecx,(%eax)
```

Begin execution

Branch Prediction Through Loop

Assume
vector length = 100

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict taken (oops)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 100

Read invalid location

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 101

Executed

Fetched

Branch Misprediction Invalidation

Assume
vector length = 100

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict taken (oops)

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

~~*i = 100*~~

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~

~~*i = 101*~~

Invalidate

Branch Misprediction Recovery

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
80488bb:   leal   0xffffffe8(%ebp),%esp
80488be:   popl   %ebx
80488bf:   popl   %esi
80488c0:   popl   %edi
```

i = 99

Definitely not taken

- **Performance Cost**
  - multiple clock cycles on modern processor
  - can be a major performance limiter

Conditional Moves

```
void minmax1(int *a, int *b, int n {
  int i;
  for (i=0; i<n; i++) {
    if (a[i] > b[i]) {
      int t = a[i];
      a[i] = b[i];
      b[i] = t;
    }
  }
}
```

- Compiled code uses conditional branch
  - 14.5 CPE for random data
  - 2.0 – 4.0 CPE for predictable data

```
void minmax2(int *a, int *b, int n {
  int i;
  for (i=0; i<n; i++) {
    int min = a[i] < b[i]?
          a[i] : b[i];
    int max = a[i] < b[i]?
          b[i] : a[i];
    a[i] = min;
    b[i] = max;
  }
}
```

- Compiled code uses conditional move instruction
  - 5.0 CPE regardless of data's pattern

Latency of Loads

```
typedef struct ELE {
  struct ELE *next;
  int data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
  int len = 0;
  while (ls) {
    len++;
    ls = ls->next;
  }
  return len;
}
```

```
# len in %eax, ls in %rdi

.L11:                       # loop:
  addl   $1, %eax           # incr len
  movq   (%rdi), %rdi       # ls = ls->next
  testq  %rdi, %rdi         # test ls
  jne    .L11               # if != 0
                            #   go to loop
```

- **4 CPE**

Clearing an Array ...

```
#define ITERS 100000000
int main() {
  volatile int dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<100; i++)
      dest[i] = 0;
  }
}
```

Clearing an Array ... Unwound

```
#define ITERS 100000000
int main() {
  volatile int dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<97; i+=4) {
      dest[i] = 0;
      dest[i+1] = 0;
      dest[i+2] = 0;
      dest[i+3] = 0;
    }
  }
}
```

Store/Load Interaction

```
void write_read(int *src, int *dest, int n) {
  int cnt = n;
  int val = 0;

  while(cnt--) {
    *dest = val;
    val = (*src)+1;
  }
}
```

Store/Load Interaction

```
int A[] = {-10, 17};
```

Example A: `write_read(&a[0],&a[1],3)`

	Initial	Iter. 1	Iter. 2	Iter. 3
`cnt`	3	2	1	0
`a`	–10	17	–10	0	–10	–9	–10	–9
`val`	0	–9	–9	–9

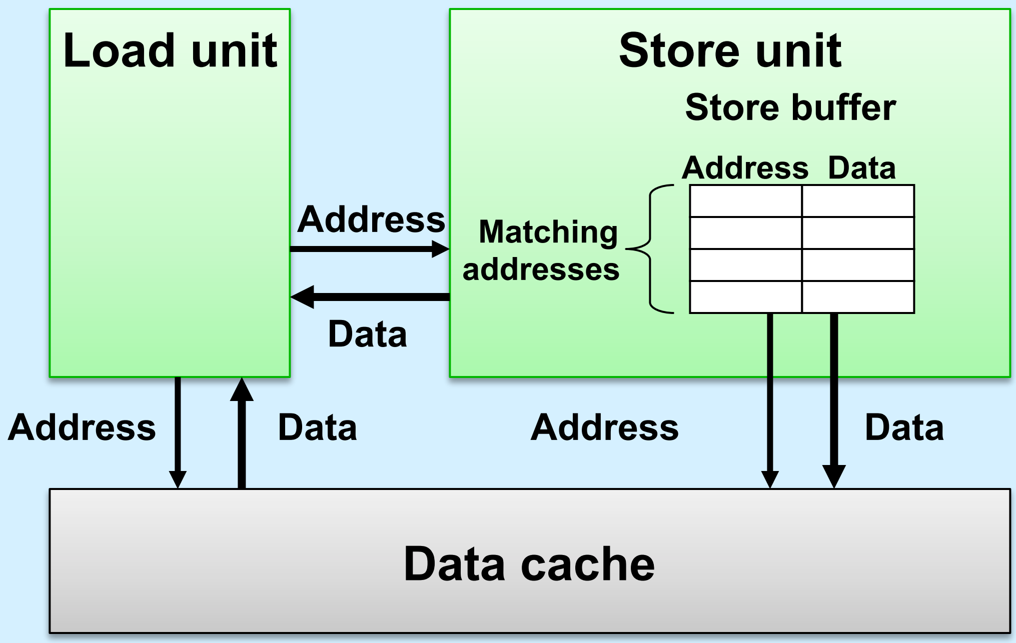
- CPE 2.0

Example B: `write_read(&a[0],&a[0],3)`

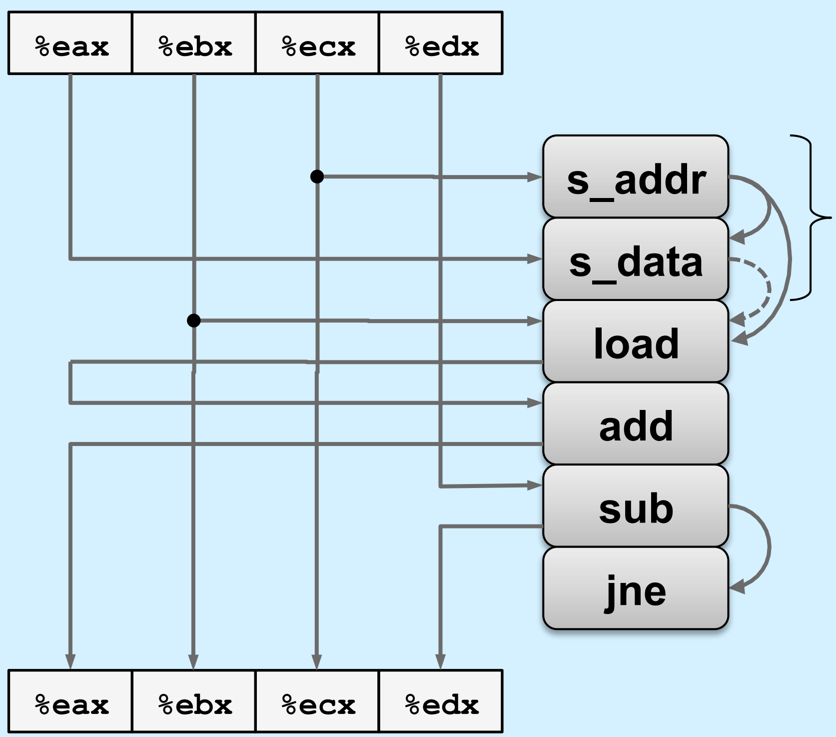
	Initial	Iter. 1	Iter. 2	Iter. 3
`cnt`	3	2	1	0
`a`	–10	17	0	17	1	17	2	17
`val`	0	1	2	3

- CPE 6.0

Some Details of Load and Store

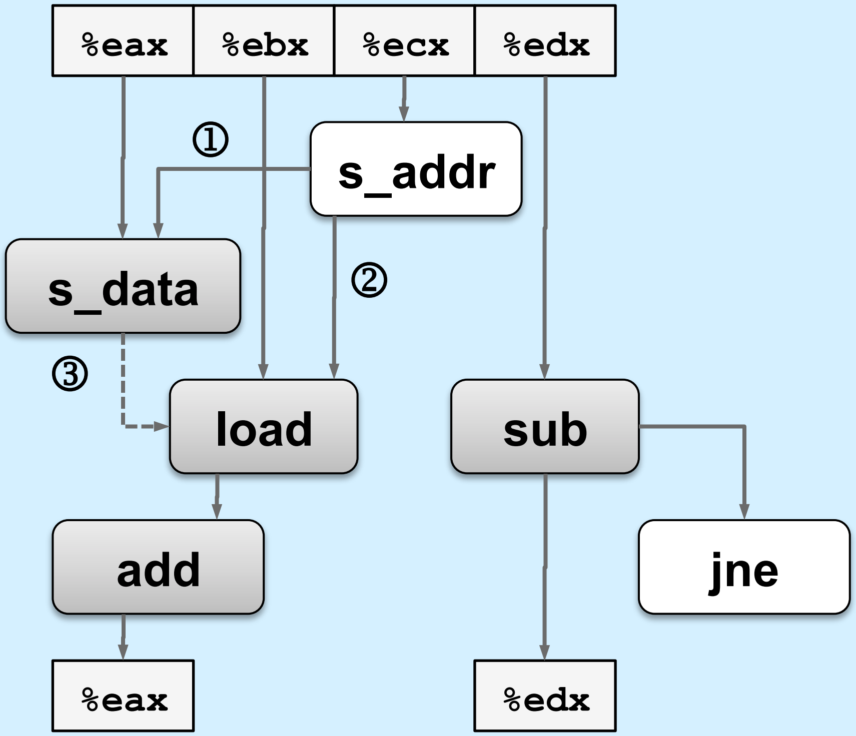


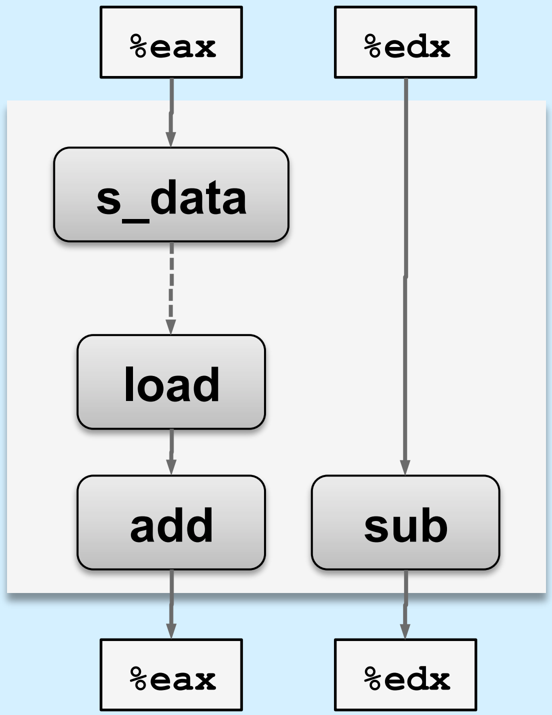
Inner-Loop Data Flow of Write_Read



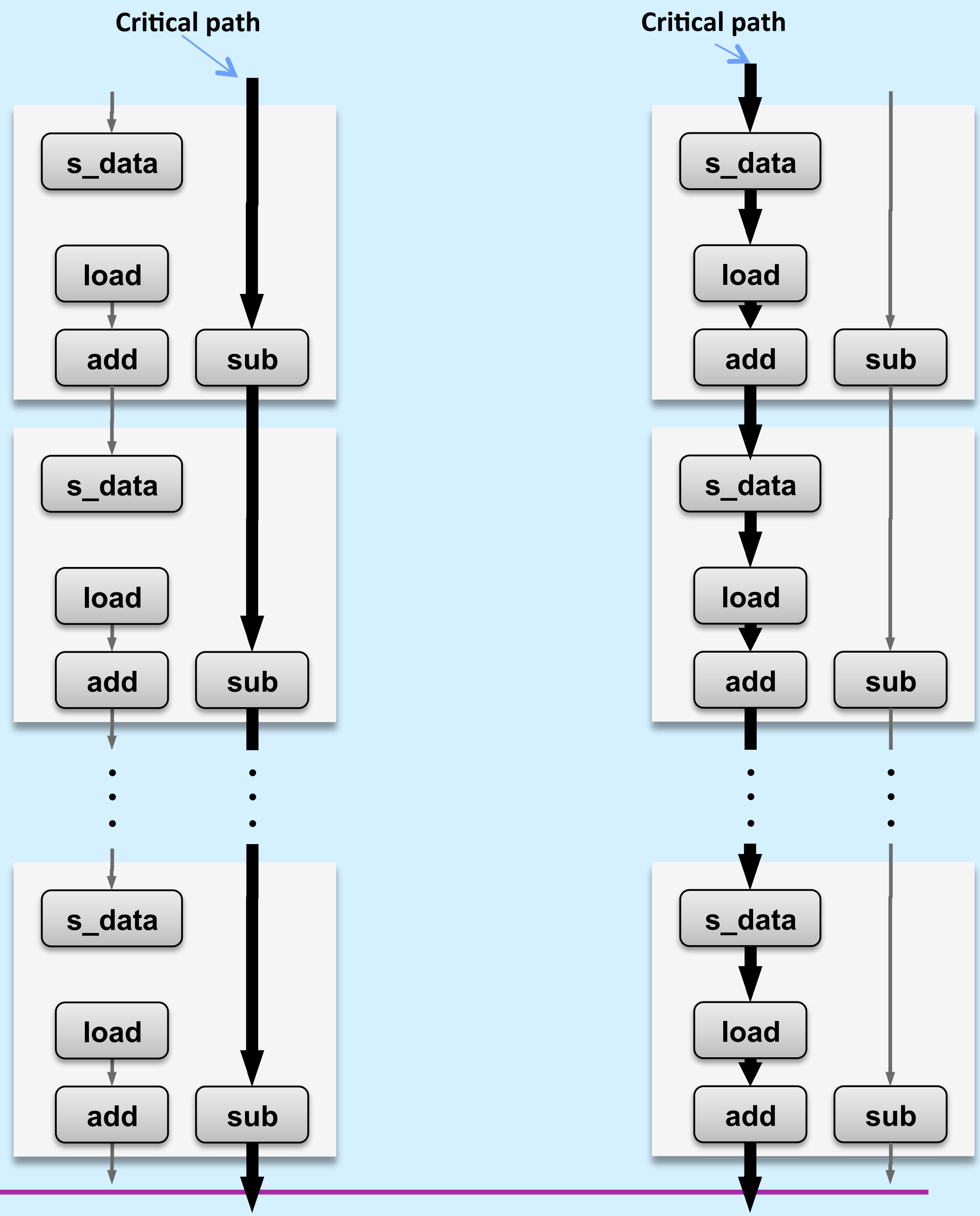

```
movl %eax,(%ecx)     *dest = val;
movl (%ebx),%eax     val = *src
addl $1,%eax         val++;
subl $1,%edx         cnt--;
jne loop
```

Inner-Loop Data Flow of Write_Read





Data Flow



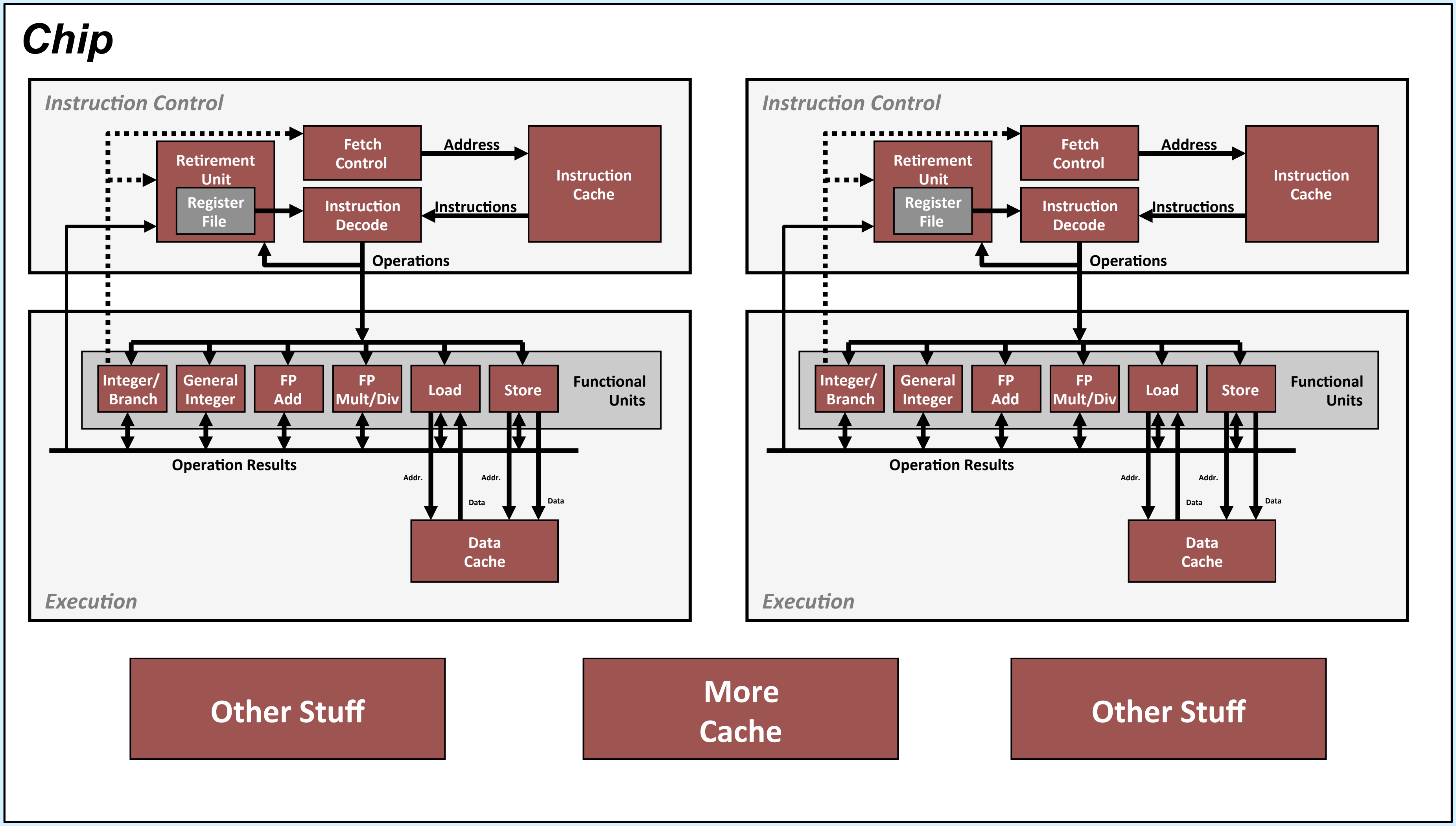
Critical path
s_data
load
add
sub
s_data
load
add
sub
s_data
load
add
sub
Critical path
s_data
load
add
sub
s_data
load
add
sub
s_data
load
add
sub

Getting High Performance

- Good compiler and flags
- Don't do anything stupid
  - 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)

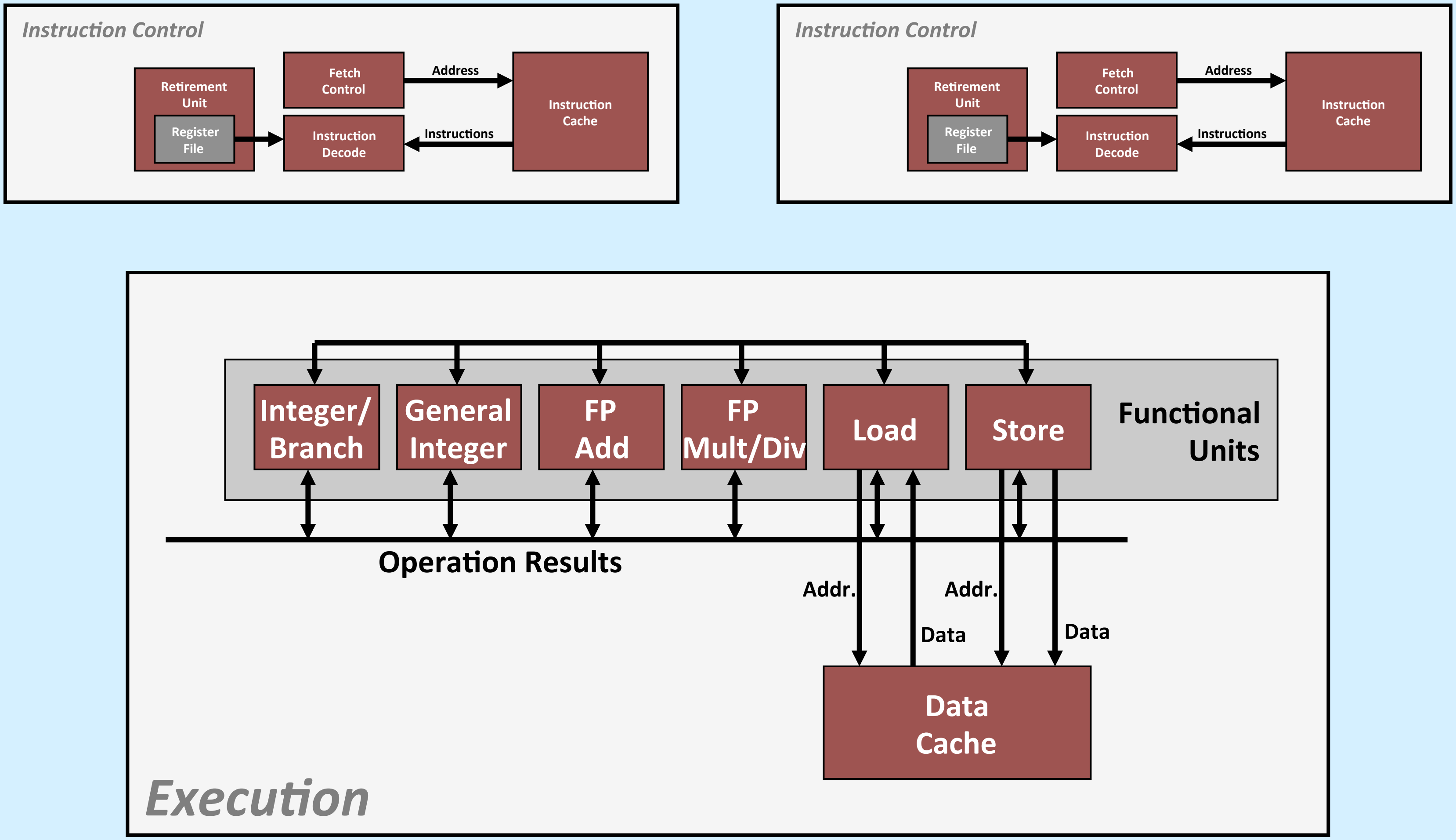
- Tune code for machine
  - exploit instruction-level parallelism
  - avoid unpredictable branches
  - make code cache friendly (covered soon)

Multiple Cores



 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

Hyper Threading



Copyright © 2015 Thomas W. Doeppner. All rights reserved.