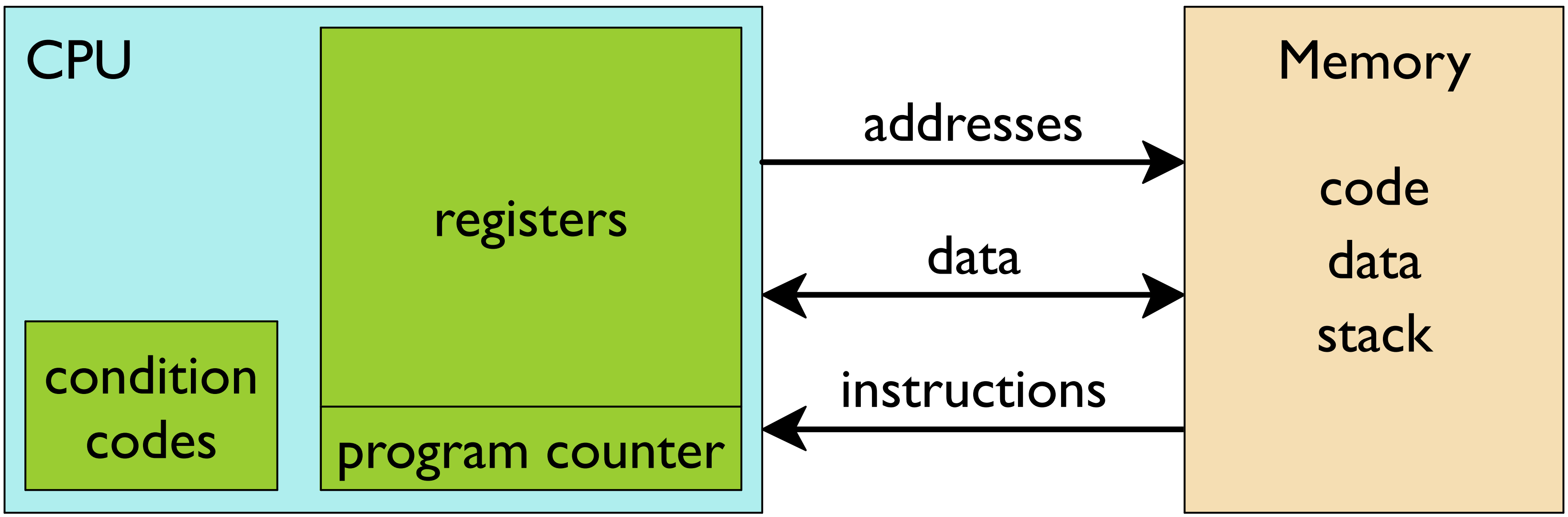
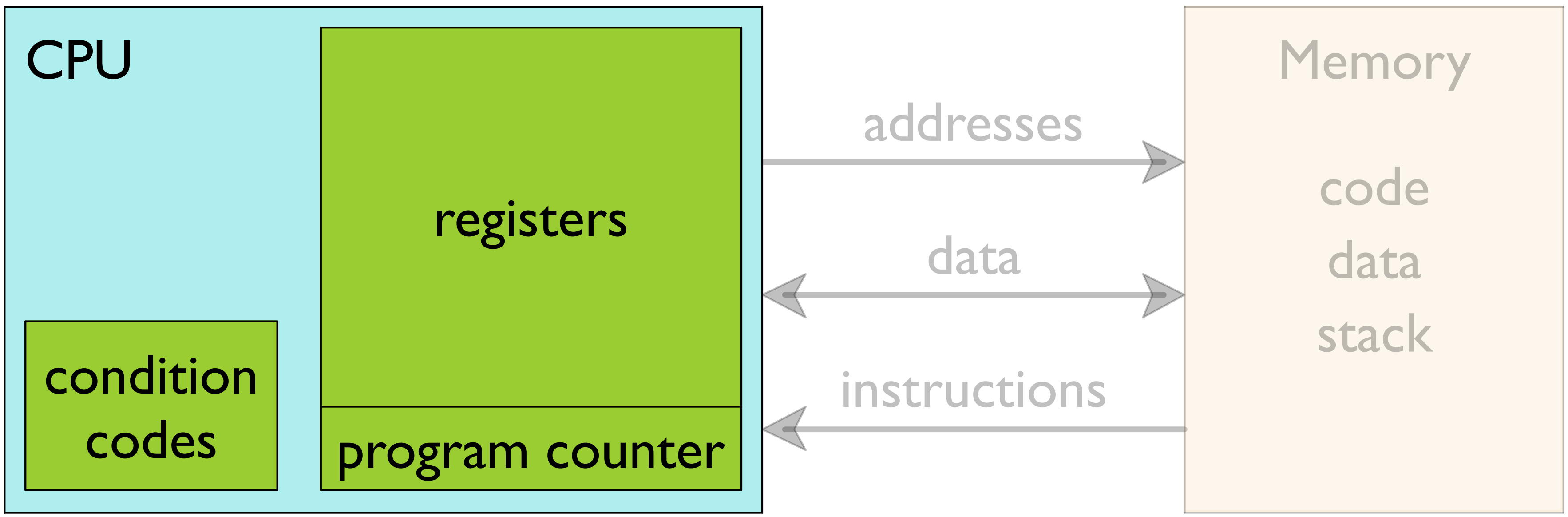
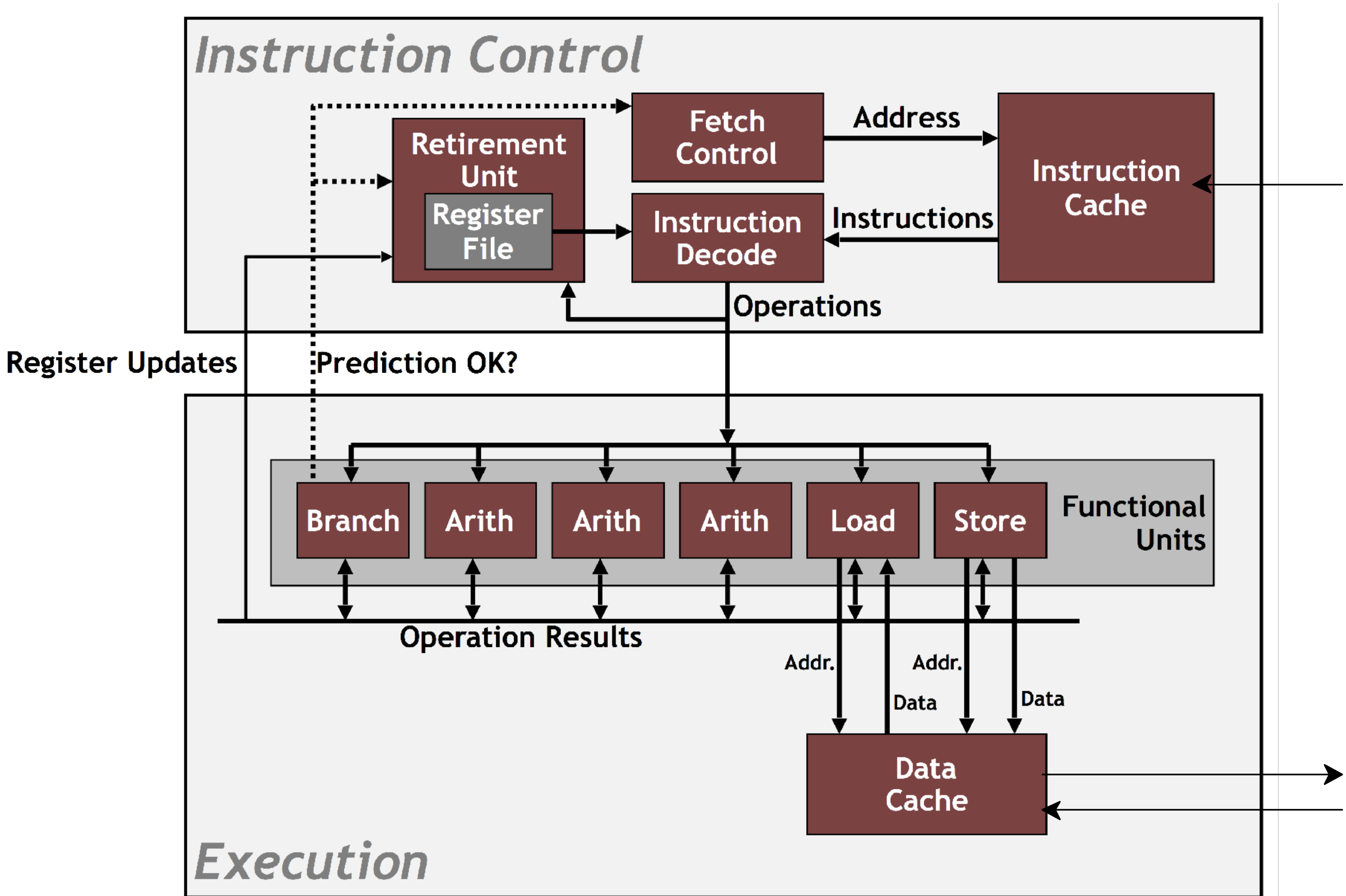
Simple CPU


CPU
registers
condition codes
program counter
addresses
data
instructions
Memory
code
data
stack

Simple CPU


CPU
registers
condition codes
program counter
addresses
data
instructions
Memory
code
data
stack

Modern CPU



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Definitions

Superscalar processor

- Issue and execute multiple instructions within a cycle
- Instructions are determined dynamically

Instruction-level parallelism

- Some instructions in a program can execute at once
- No explicit declaration needed

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;
}
```

Multiplication takes 3 cycles, so... 9 cycles minimum?

No, because

- one new multiplication can start every cycle
- **`a*b`** and **`a*c`** are independent calculations

Pipelined Functional Units

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

	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 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`

Instruction Performance

Haswell:

	latency	*cycles/issue*
load	4	1
store	4	1
integer add	1	1
integer multiply	3	1
integer divide	3-30	3-30
FP add	3	1
FP multiply	5	1
FP divide	3-15	3-15

Instruction Performance

Haswell:

How long one takes

	latency	*cycles/issue*
load	4	1
store	4	1
integer add	1	1
integer multiply	3	1
integer divide	3-30	3-30
FP add	3	1
FP multiply	5	1
FP divide	3-15	3-15

Instruction Performance

How long to wait before starting a new one

	latency	*cycles/issue*
load	4	1
store	4	1
integer add	1	1
integer multiply	3	1
integer divide	3-30	3-30
FP add	3	1
FP multiply	5	1
FP divide	3-15	3-15

CPE for Benchmark

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

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

CPE for Benchmark

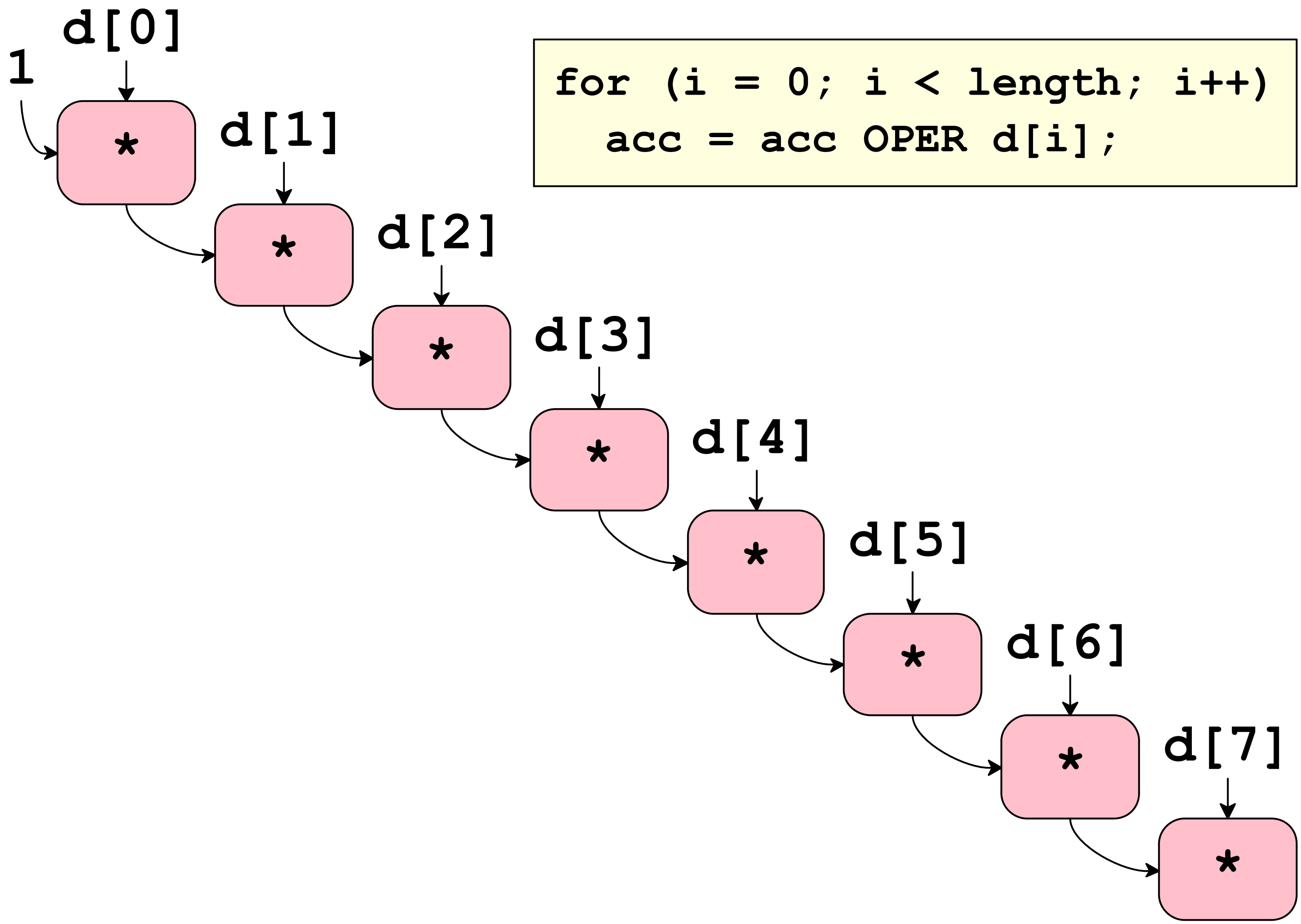
FP *

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

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

	int		double	
	+	*	+	*
accumulate to local	1.27	3.01	3.01	5.01
latency bound	1.00	3.00	3.00	5.00

Sequential Computation




```
for (i = 0; i < length; i++)
  acc = acc OPER d[i];
```

Loop Unrolling

```
void combine5(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  int limit = length - 1;
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;

  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i += 2)
    acc = (acc OPER data[i]) OPER data[i+1];

  /* Finish any remaining elements */
  for (; i < length; i++)
    acc = acc OPER data[i];
  *dest = acc;
}
```

Loop Unrolling

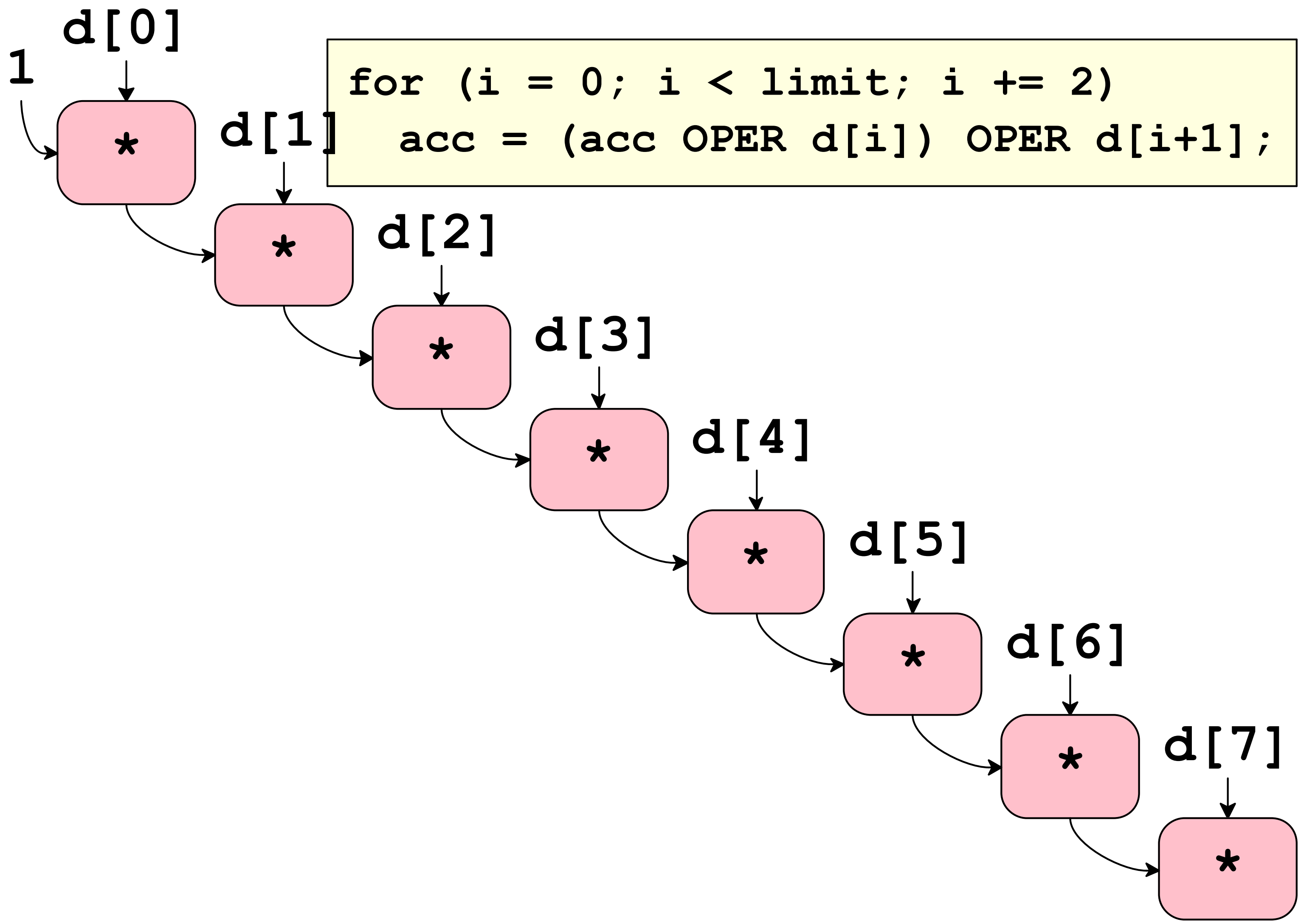
```
void combine5(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  int limit = length - 1;
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;

  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i += 2)
    acc = (acc OPER data[i]) OPER data[i+1];
```

	int		double	
	+	*	+	*
accumulate to local	1.27	3.01	3.01	5.01
2x unroll	1.01	3.01	3.01	5.01
latency bound	1.00	3.00	3.00	5.00

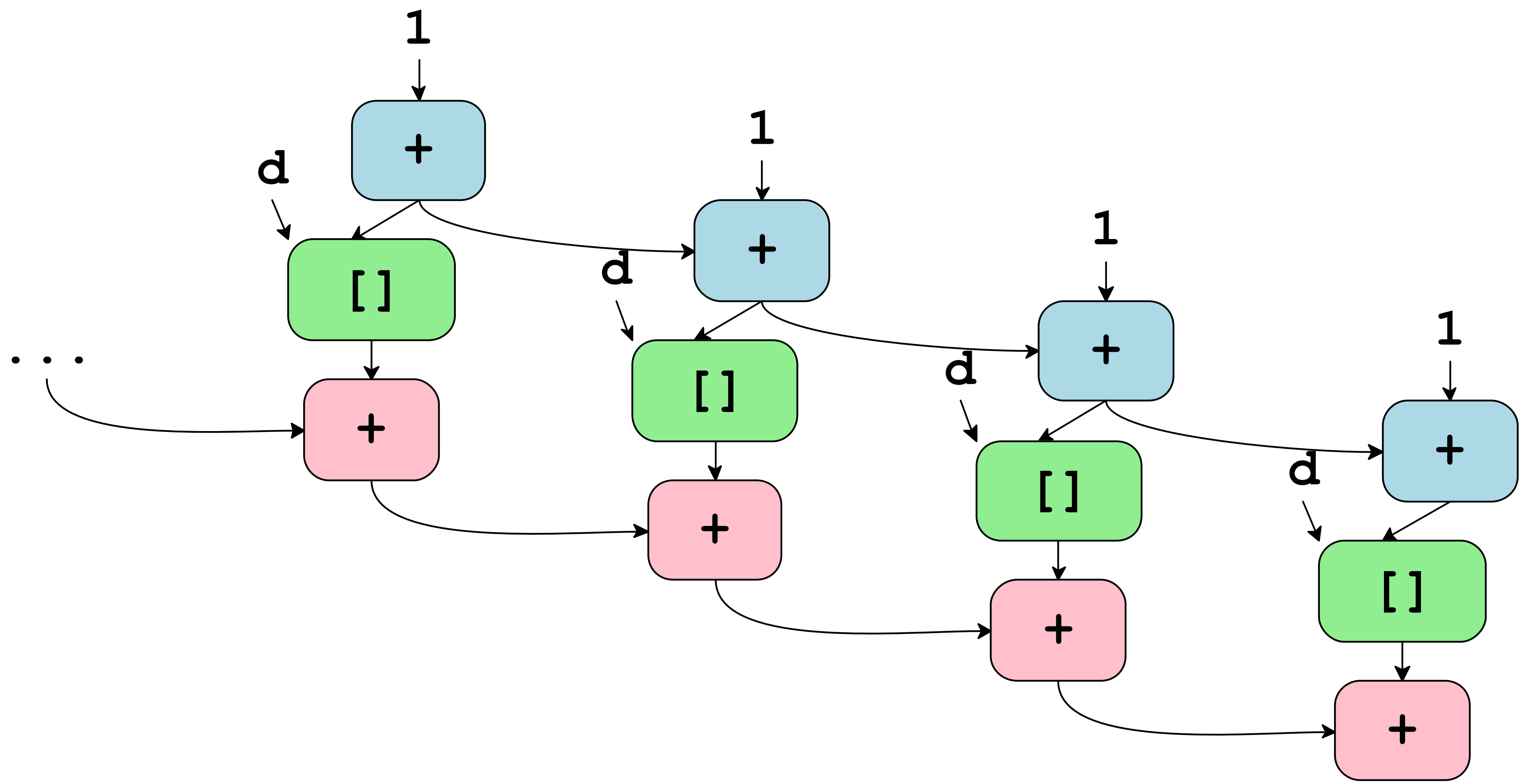
Sequential Computation

```
for (i = 0; i < limit; i += 2)
  acc = (acc OPER d[i]) OPER d[i+1];
```



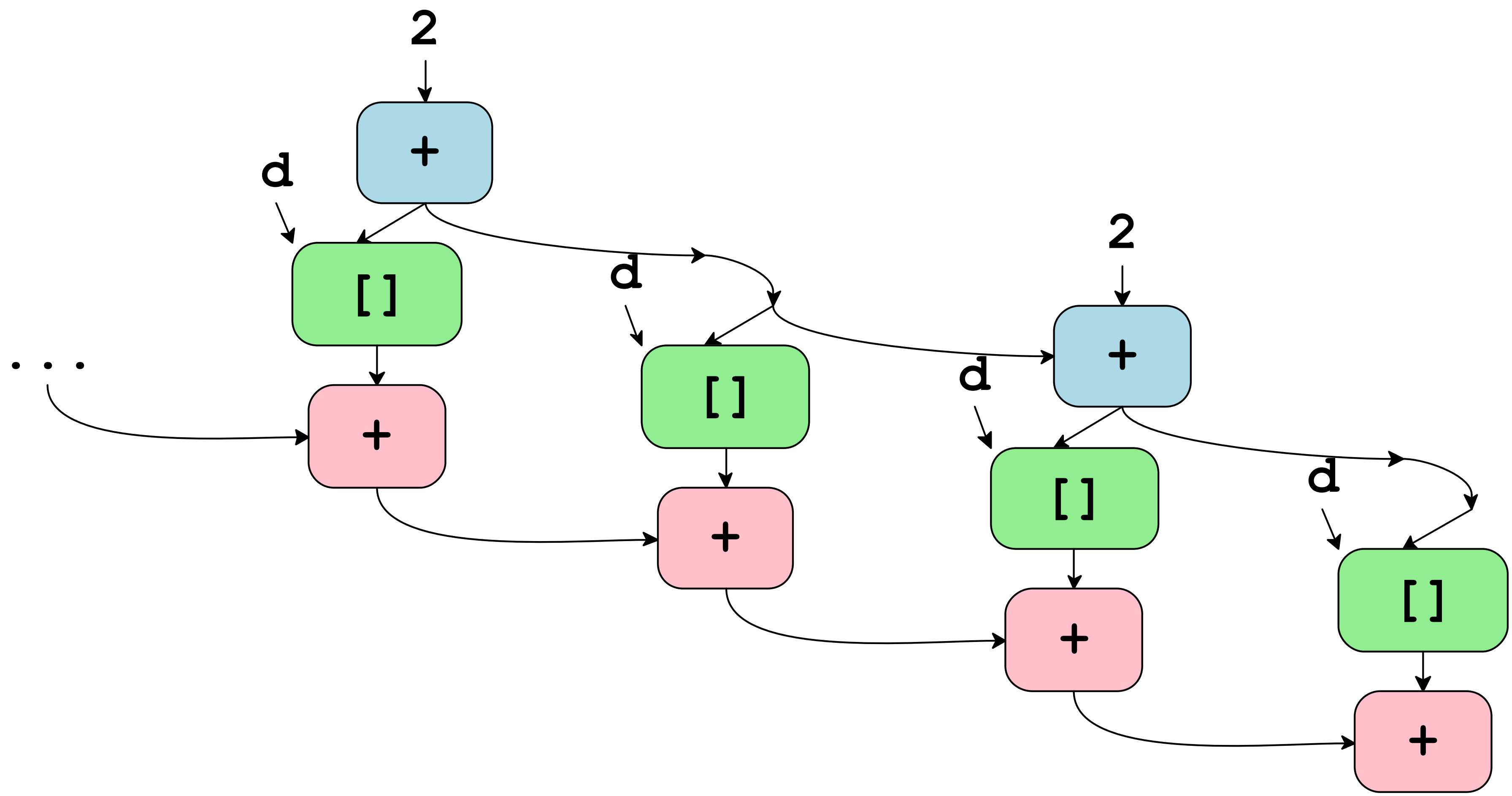
Why Addition Improves, Anyway

Orignal:



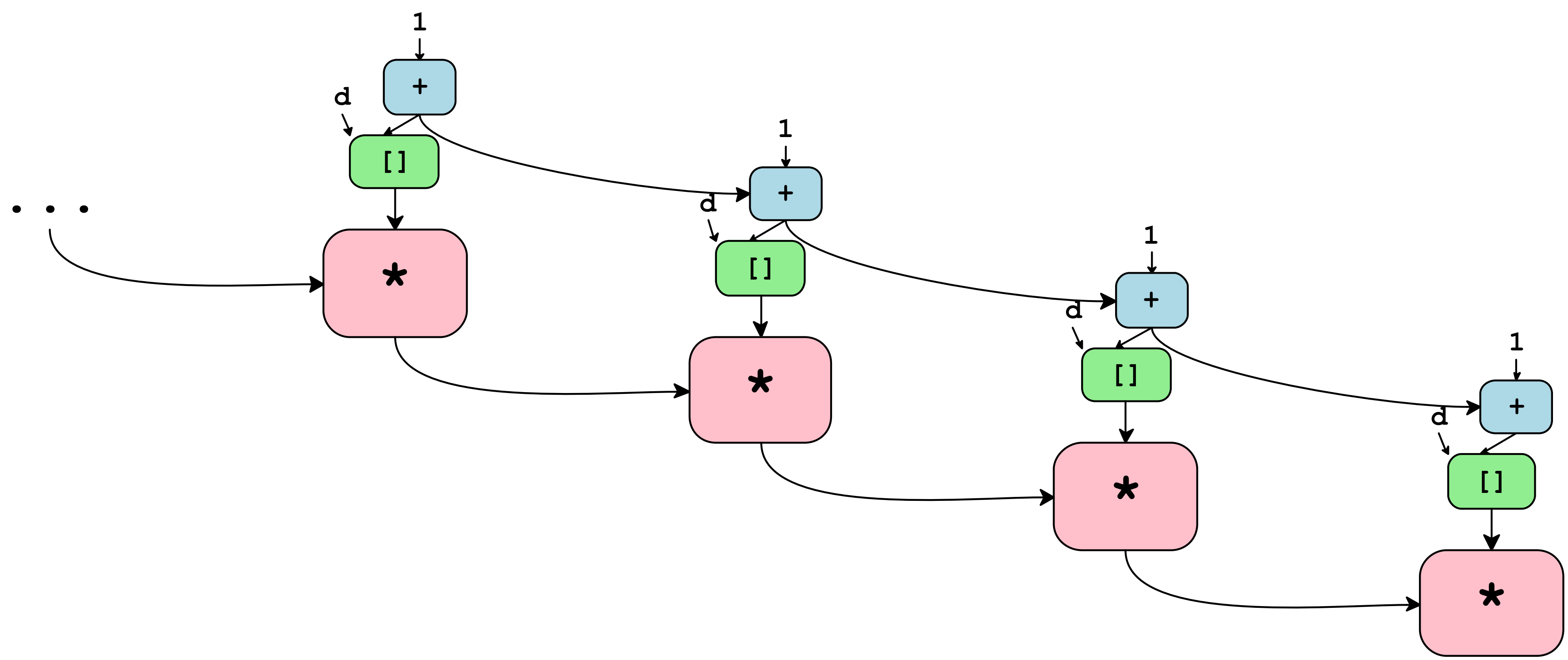
Why Addition Improves, Anyway

Unrolled:



Why Addition Improves, Anyway

Original multiplication:



Reassociation

```
for (i = 0; i < limit; i += 2)
  acc = (acc OPER d[i]) OPER d[i+1];
```

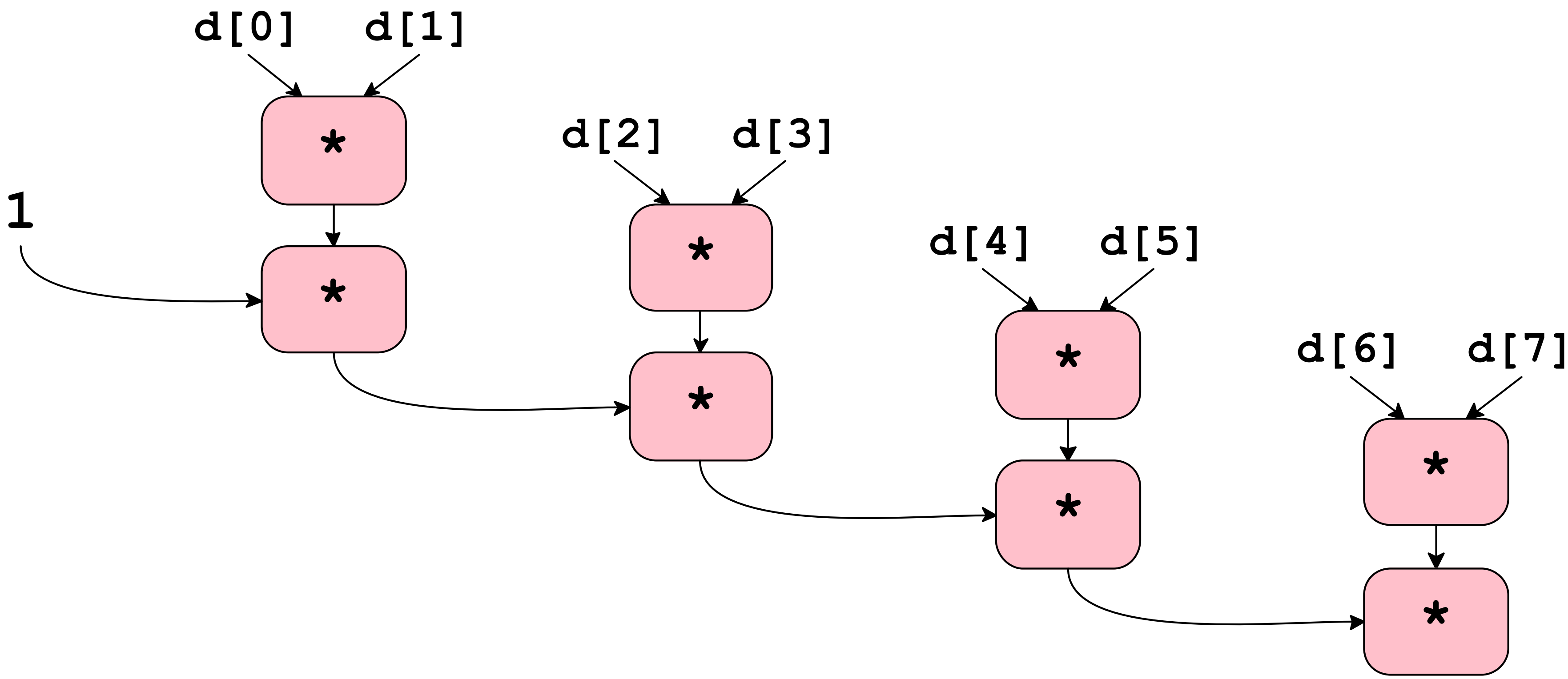
```
for (i = 0; i < limit; i += 2)
  acc = acc OPER (d[i] OPER d[i+1]);
```

Always the same result?

Not for floating-point, but probably good enough

	int		double	
	+	*	+	*
2x unroll	1.01	3.01	3.01	5.01
2x + reassoc	1.01	1.51	1.51	2.51
latency bound	1.00	3.00	3.00	5.00

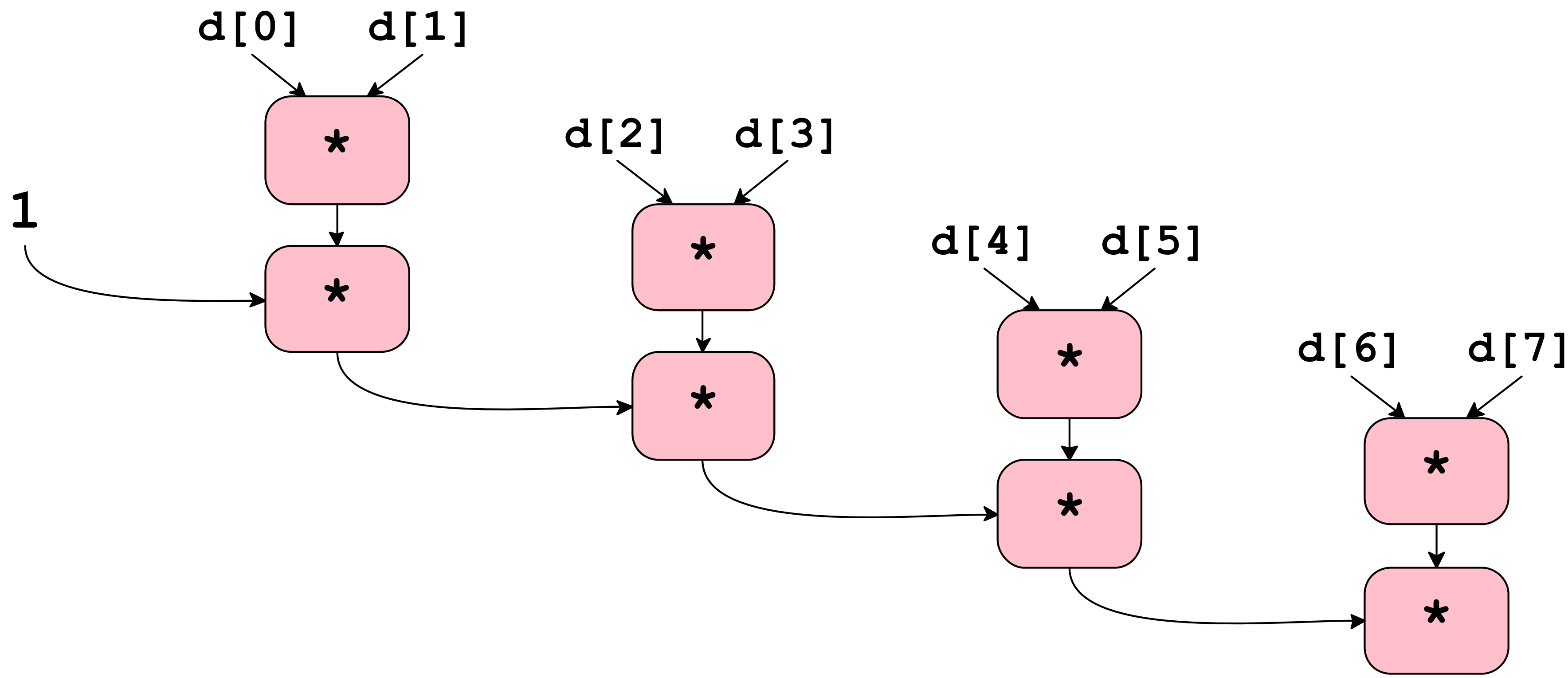
Reassociated Computation



***** for **double**: 5 cycles

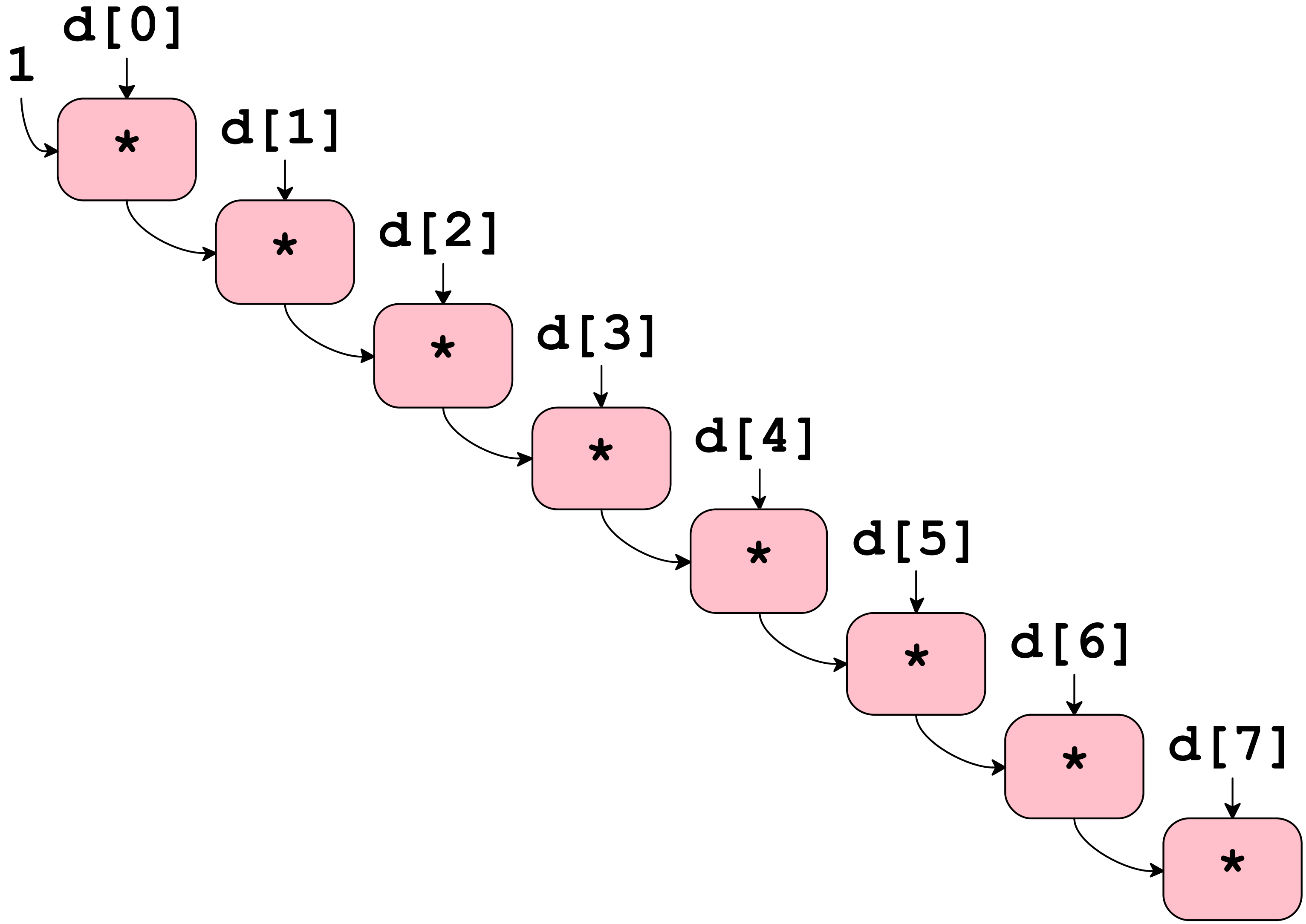
per 2 elements ⇒ 2.5 CPE

Reassociated Computation

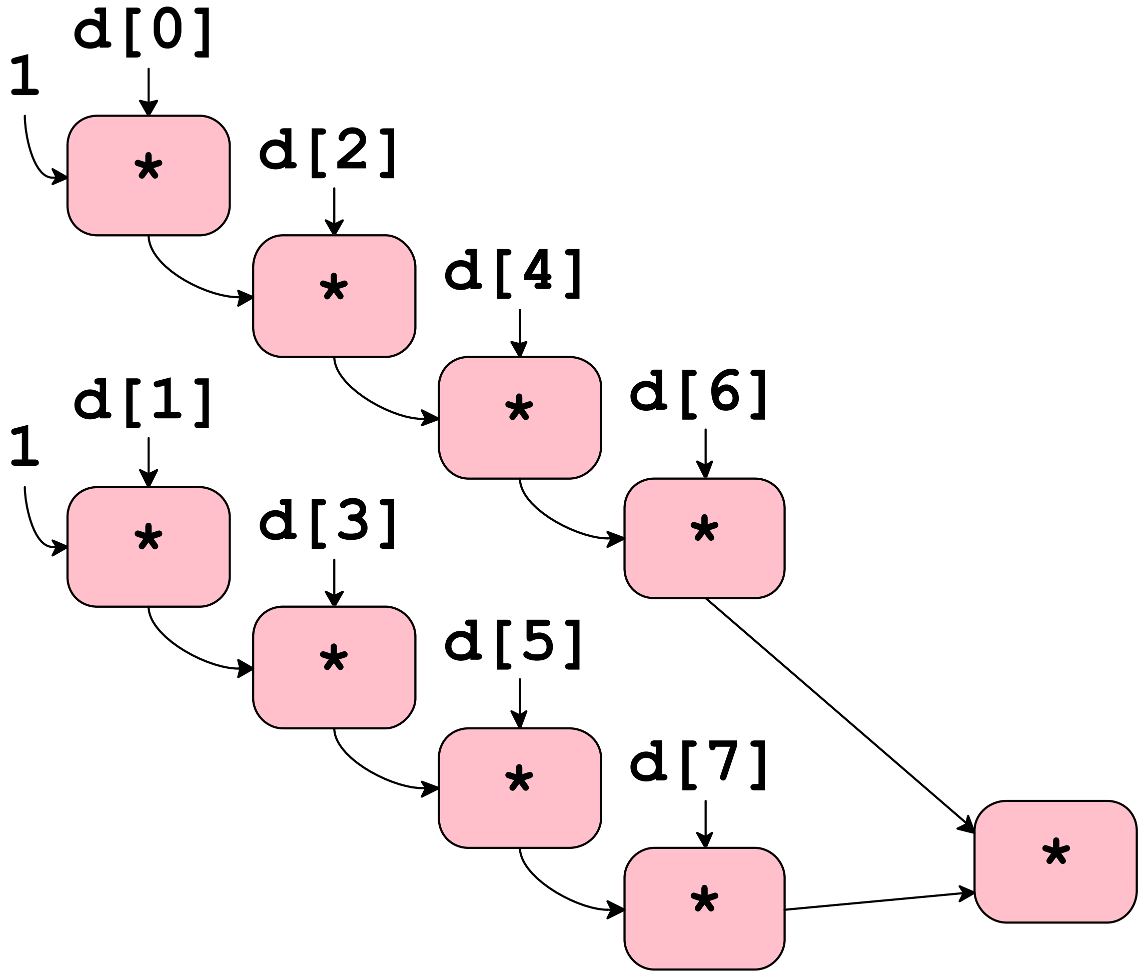


	int		double	
	+	*	+	*
2x unroll	1.01	3.01	3.01	5.01
2x + reassoc	1.01	1.51	1.51	2.51
latency bound	1.00	3.00	3.00	5.00

Another Reassociation



Another Reassociation



“2x2” = 2x unrolling with 2 accumulators

Unrolling with Multiple Accumulators

```
void combine6(vec_ptr v, data_t *dest) {
  long int i;
  int length = vec_length(v);
  int limit = length - 1;
  data_t* data = get_vec_start(v);
  data_t acc0 = IDENT, acc1 = IDENT;

  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i += 2) {
    acc0 = acc0 OPER data[i];
    acc1 = acc1 OPER data[i+1];
  }

  /* Finish any remaining elements */
  for (; i < length; i++)
    acc0 = acc0 OPER data[i];
  *dest = acc0 OPER acc1;
}
```

Keep Unrolling

	int		double	
	+	*	+	*
2x + reassoc	1.01	1.51	1.51	2.51
2x2 unroll	0.81	1.51	1.51	2.51
4x4 unroll	0.72	1.07	1.01	1.25
latency bound	1.00	3.00	3.00	5.00

Why aren't all 4x4 times half of 2x2 times?

Parallelism by Pipeline

	latency	*cycles/issue*
integer multiply	3	1

	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
Stage 1	**`a*b`**	**`c*d`**	**`e*f`**	**`g*h`**		
Stage 2		**`a*b`**	**`c*d`**	**`c*d`**	**`g*h`**	
Stage 3			**`a*b`**	**`c*d`**	**`c*d`**	**`g*h`**

Parallelism by Pipline × Capacity

	latency	*cycles/issue*	*capacity*
integer multiply	3	1	2

	t = 1	t = 2	t = 3	t = 4	t = 5
Stage 1	**`a*b`**	**`c*d`**	**`e*f`**		
Stage 2		**`a*b`**	**`c*d`**	**`c*d`**	
Stage 3			**`a*b`**	**`c*d`**	**`c*d`**

Stage 1	**`g*h`**		
Stage 2		**`g*h`**	
Stage 3			**`g*h`**

Throughput Bound

	latency	*cycles/issue*	*capacity*	*throughput bound*
load	4	1	2	0.5
store	4	1	1	1
integer add	1	1	4	0.25
integer multiply	3	1	1	1
integer divide	3-30	3-30	1	3-30
FP add	3	1	1	1
FP multiply	5	1	2	0.5
FP divide	3-15	3-15	1	3-15

Throughput Bound

cycles/issue / capacity

	latency	*cycles/issue*	*capacity*	*throughput bound*
load	4	1	2	0.5
store	4	1	1	1
integer add	1	1	4	0.25
integer multiply	3	1	1	1
integer divide	3-30	3-30	1	3-30
FP add	3	1	1	1
FP multiply	5	1	2	0.5
FP divide	3-15	3-15	1	3-15

`combine`	**`int`**		**`double`**	
	`+`	**`*`**	**`+`**	**`*`**
throughput bound	`0.50`	`1.00`	`1.00`	`0.50`

Throughput Bound

cycles/issue / capacity

	latency	*cycles/issue*	*capacity*	*throughput bound*
load	4	1	2	0.5
store	4	1	1	1
integer add	1	1	4	0.25
integer multiply	3	1	1	1
integer divide	3-30	3-30	1	3-30
FP add	3	1	1	1
FP multiply	5	1	2	0.5
FP divide	3-15	3-15	1	3-15

Limited by load instead of +

	`int`		`double`	
	`+`	`*`	`+`	`*`
throughput bound	`0.50`	`1.00`	`1.00`	`0.50`

Keep Unrolling?

	int		double	
	+	*	+	*
2x + reassoc	1.01	1.51	1.51	2.51
2x2 unroll	0.81	1.51	1.51	2.51
4x4 unroll	0.72	1.07	1.01	1.25
latency bound	1.00	3.00	3.00	5.00
throughput bound	0.50	1.00	1.00	0.50

More unrolling:

+ more parallelism

− more register pressure

Keep Unrolling?

Floating-point *

accumulators \ unrolling factor	1x	2x	3x	4x	6x	8x	10x	12x
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

Keep Unrolling?

Integer +

unrolling factor

accumulators	1x	2x	3x	4x	6x	8x	10x	12x
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

Summary for Unrolling & Reassociation

	int		double	
	+	*	+	*
2x unroll	1.01	3.01	3.01	5.01
2x + reassoc	1.01	1.51	1.51	2.51
2x2 unroll	0.81	1.51	1.51	2.51
4x4 unroll	0.72	1.07	1.01	1.25
10x10 unroll	0.55	1.00	1.01	0.52
latency bound	1.00	3.00	3.00	5.00
throughput bound	0.50	1.00	1.00	0.50

What About Branches?

```
for (i = 0; i < limit; i += 2) {
  acc0 = acc0 + data[i];
  acc1 = acc1 + data[i+1];
}
```

CPE = `0.81`

What About Branches?

```
for (i = 0; i < limit; i += 2) {
  acc0 = acc0 + data[i];
  acc1 = acc1 + data[i+1];
}
```

CPE = 0.81

```
.L3:
  addl   (%rdi,%rax,4), %ecx
  addl   4(%rdi,%rax,4), %r8d
  addq   $2, %rax
  cmpq   %r9, %rax
  jl     .L3
  jmp    .L2
  ...
.L2:
  finish...
```

What About Branches?

```
for (i = 0; i < limit; i += 2) {
  acc0 = acc0 + data[i];
  acc1 = acc1 + data[i+1];
}
```

CPE = `0.81`

```
.L3:
  addl   (%rdi,%rax,4), %ecx
  addl   4(%rdi,%rax,4), %r8d
  addq   $2, %rax
  cmpq   %r9, %rax
  jl     .L3
  jmp    .L2
  ...
.L2:
  finish...
```

at the same time

What About Branches?

```
for (i = 0; i < limit; i += 2) {
  acc0 = acc0 + data[i];
  acc1 = acc1 + data[i+1];
}
```

CPE = 0.81

```
.L3:
  addl   (%rdi,%rax,4), %ecx
  addl   4(%rdi,%rax,4), %r8d
  addq   $2, %rax
  cmpq   %r9, %rax
  jl     .L3
  jmp    .L2
  …
.L2:
  finish...
```

at the same time

Might need to get an early start on the next iteration; *can't just wait on a comparison!*

Branch Prediction

Branch prediction is a *guess* about whether a jump will be taken or not

- Make a good guess by recording previous experience
- Perform work in parallel based on prediction
- Don't expose that work (by writing to memory or to registers) until the branch is known

Branch Prediction

```
addl  (%rdi,%rax,4), %ecx
addl  4(%rdi,%rax,4), %r8d
addq  $2, %rax
cmpq  %r9, %rax
jl    .L3                i = 0
```

Branch Prediction

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3                  i = 0
```

branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3                  i = 2
```

Branch Prediction

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3                 i = 0
```

branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3                 i = 2
```

branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3                 i = 4
```

...

Prediction: branch is always taken

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3           i = 96
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3              i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3              i = 98
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3             i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3             i = 98
```

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 100
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 98
```

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 100
```

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 102
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3           i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3           i = 98
```

branch not taken

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3          i = 100
```

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3          i = 102
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 98
```

assume branch taken
(oops!)

branch not taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 100
```

assume branch taken
(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 102
```

Branch Prediction

... for length `100`...

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 96
```

assume branch taken

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3            i = 98
```

assume branch taken

(oops!)

branch not taken

spend cycles to reload pipeline

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3           i = 100
```

assume branch taken

(oops!)

```
addl   (%rdi,%rax,4), %ecx
addl   4(%rdi,%rax,4), %r8d
addq   $2, %rax
cmpq   %r9, %rax
jl     .L3           i = 102
```

Helping the Branch Predictor

Mostly, branch prediction “just works”

- At the assembly level: keep **`call`** and **`ret`** balanced
- At the C level: don’t create jumps that are random