The Reduceron

Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

The Reduceron

Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

The Reduceron

Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

What is Reduction?

`1 + 1`	reduces to	`2`
`double (1 + 1)`	reduces to	`4`
`True && False`	reduces to	`False`
`append [1,2] [3]`	reduces to	`[1,2,3]`

Reduction Strategies

Suppose that ***double*** is defined by

```
double x = x + x
```

and we wish to reduce

```
double (1 + 1)
```

then there are **two** main strategies.

1. Innermost Strategy

Suppose that ***double*** is defined by

```
double x = x + x
```

then, using the **innermost strategy**,

```
    double (1 + 1)
=>  double 2
=>  2 + 2
=>  4
```

2. Outermost Strategy

Suppose that ***double*** is defined by

```
double x = x + x
```

then, using the **outermost strategy**,

```
  double (1 + 1)
=> (1 + 1) + (1 + 1)
=> 2 + (1 + 1)
=> 2 + 2
=> 4
```

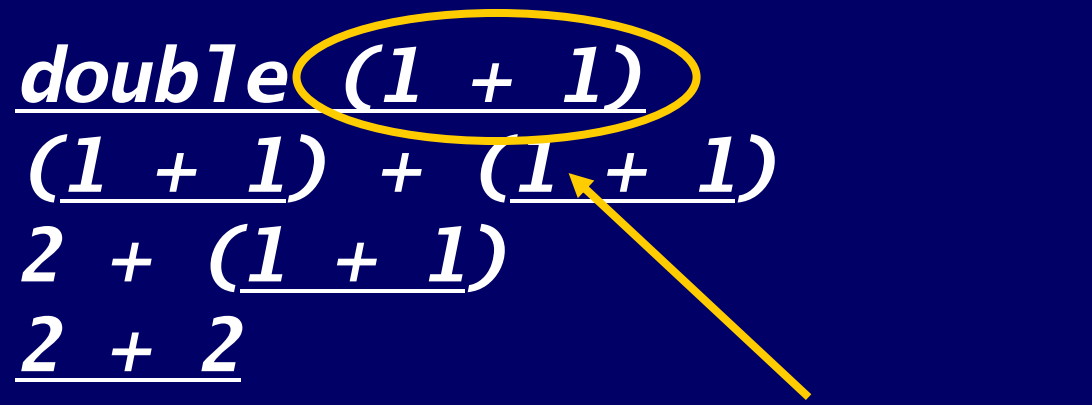
2. Outermost Strategy

Suppose that ***double*** is defined by

```
double x = x + x
```

then, using the **outermost strategy**,

```
   double (1 + 1)
=> (1 + 1) + (1 + 1)
=> 2 + (1 + 1)
=> 2 + 2
=> 4
```



Conclusion

Innermost Reduction performs fewer reductions than Outermost Reduction.

NOT NECESSARILY!

Let’s play the game again, this time with a different function.

Reduction Strategies

Suppose that ***&&*** is defined by

```
x && y  =  if x then y else False
```

and we wish to reduce

```
False && (True && True)
```

then there are two strategies.

1. Innermost Reduction

Suppose that **&&** is defined by

```
x && y  =  if x then y else False
```

then, using the **innermost strategy**,

```
      False && (True && True)
=>    False &&
        (if True then True else False)
=>    False && True
=>    if False then True else False
=>    False
```

2. Outermost Reduction

Suppose that ***&&*** is defined by

```
x && y  =  if x then y else False
```

then, using the **outermost strategy**,

```
      False && (True && True)
=>    if False then True && True
                else False
=>    False
```

2. Outermost Reduction

Suppose that **&&** is defined by

```
x && y  =  if x then y else False
```

then, using the **outermost strategy**,

```
      False && (True && True)
=>    if False then True && True
               else False
=>    False
```

Never reduced!

Conclusion

How many times is each argument reduced?

Innermost: **Exactly once**

Outermost: **Zero or more**

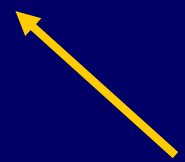
Lazy Reduction

How many times is each argument reduced?

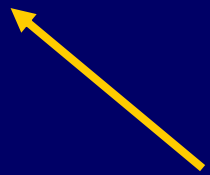
Innermost: **Exactly once**

Outermost: **Zero or more**

Lazy: **Zero or once**



Outermost, with magic!



Optimal

The Reduceron

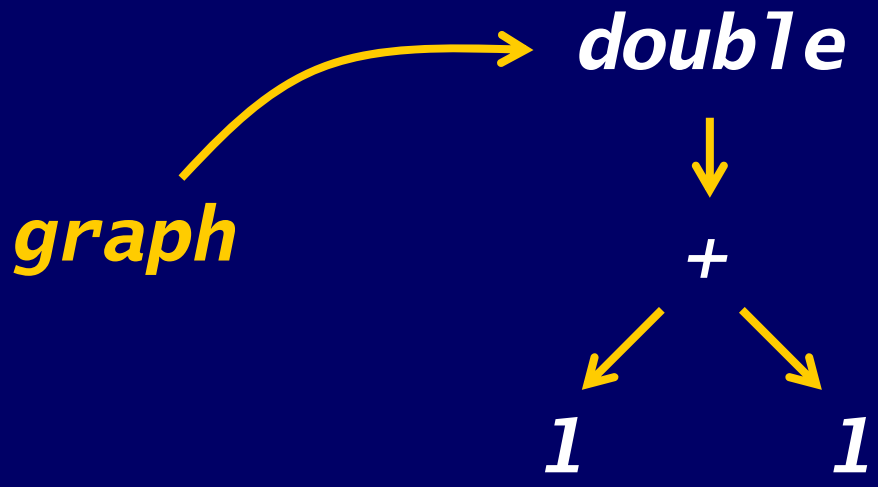
Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

Expressions as Graphs

Suppose that ***double*** is defined by

```
double x = x + x
```

and we wish to reduce ***double (1 + 1)***.

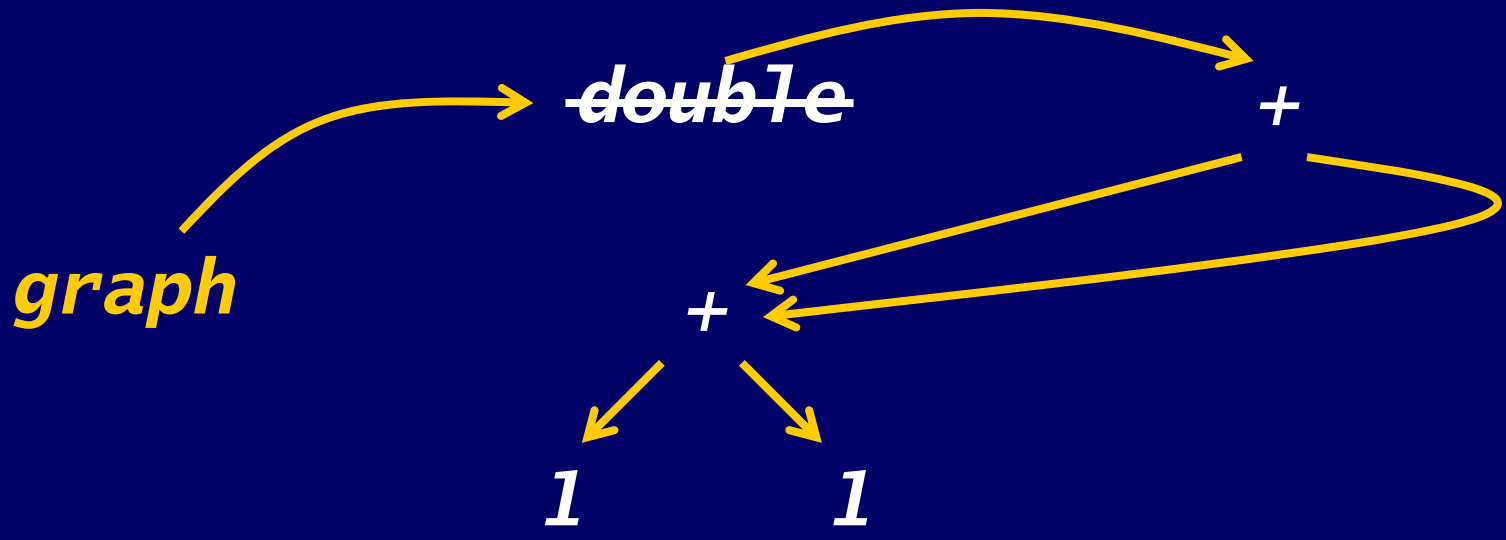


Graph Reduction

Suppose that ***double*** is defined by

double x = x + x

then, using the **lazy reduction** strategy,

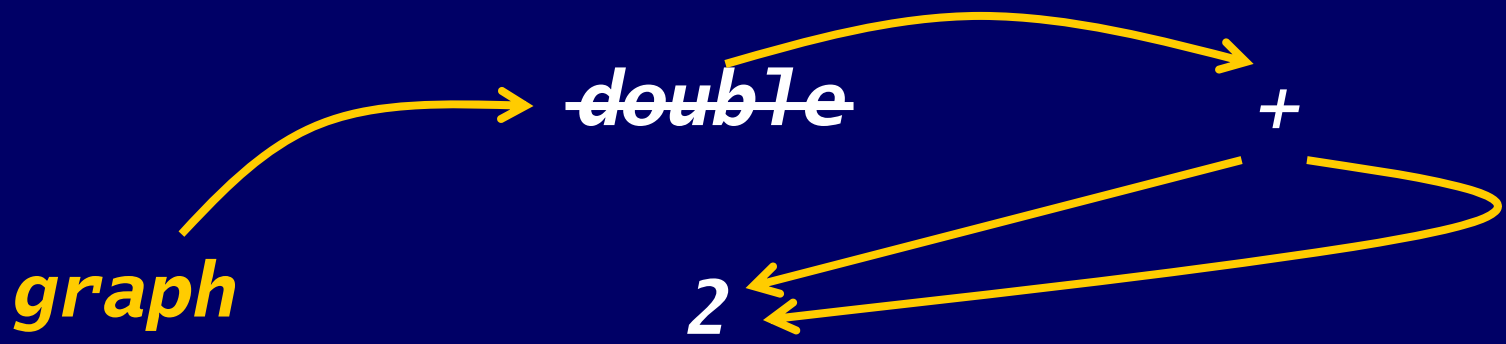


Graph Reduction

Suppose that ***double*** is defined by

```
double x = x + x
```

then, using the **lazy reduction** strategy,



Graph Reduction

Suppose that ***double*** is defined by

```
double x = x + x
```

then, using the **lazy reduction** strategy,

The Reduceron

Widening the von Neumann Bottleneck for Graph Reduction using an FPGA

A Graph Reduction Machine

Suppose that ***f*** is defined by

$$f\ x\ y\ z \quad = \quad g\ y\ (h\ z\ x)$$

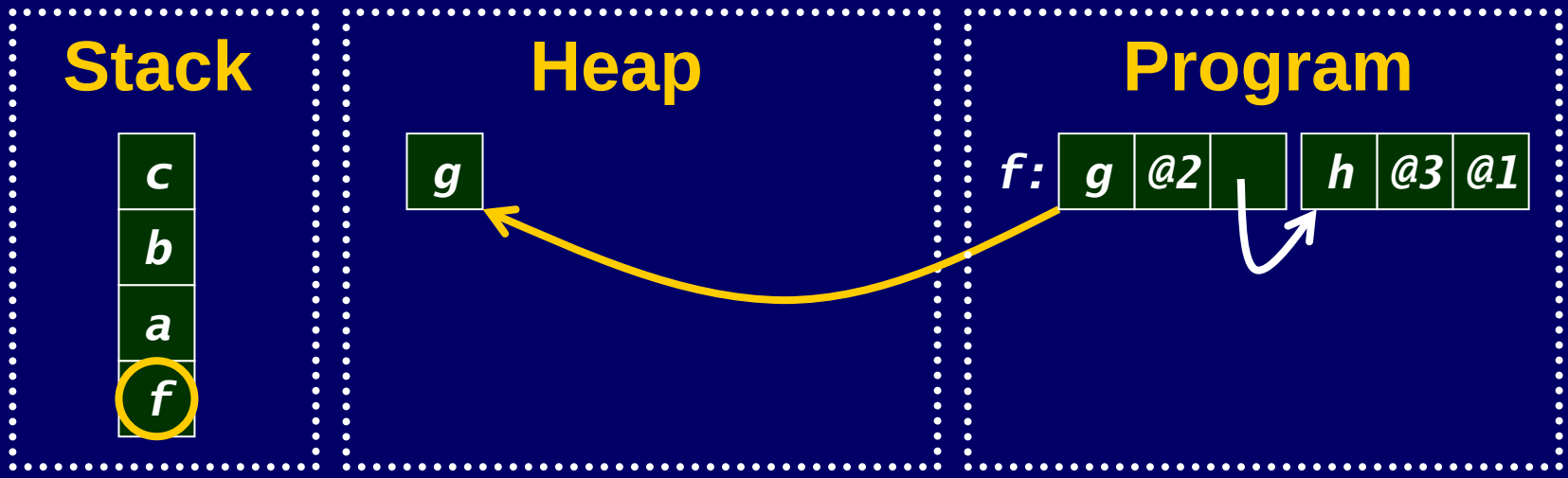
where ***g*** and *h* are functions and the following machine-state arises during reduction.

A Graph Reduction Machine

Operation:

```
f <- Stack[0]
g <- Code[f]
g -> Heap
```

Count: 3



A Graph Reduction Machine

Operation:

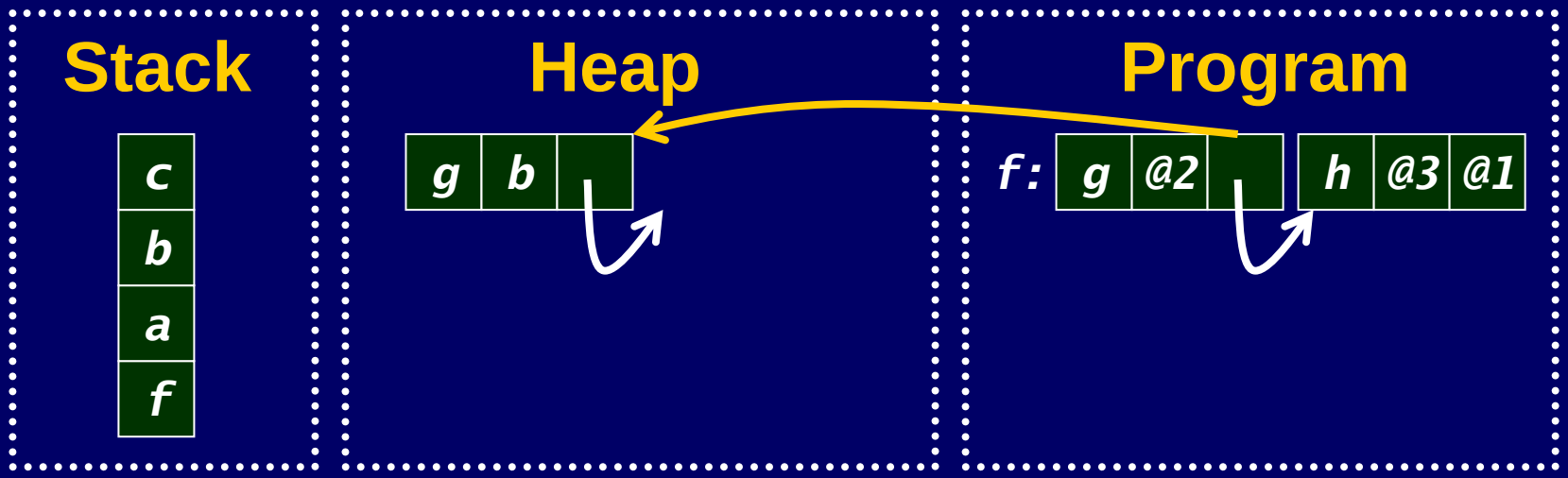
```
arg <- Code[f+1]
b   <- Stack[arg]
b   -> Heap
```

Count: 6

A Graph Reduction Machine

Operation: *ptr <- Code[f+2]*
ptr' -> Heap

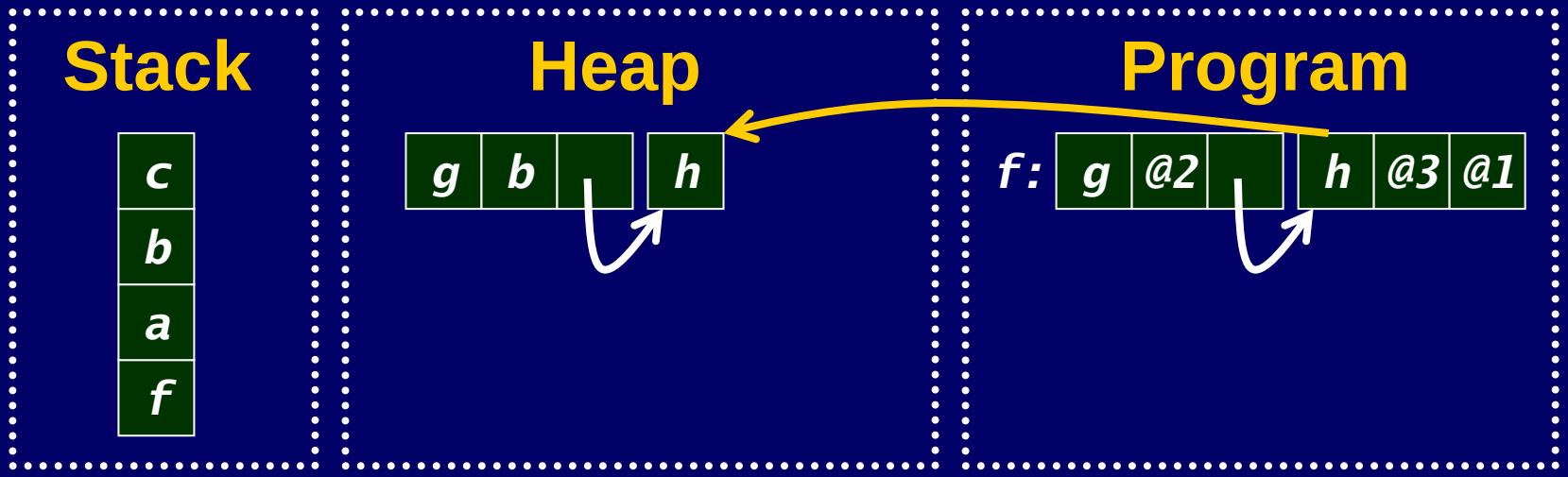
Count: 8



A Graph Reduction Machine

Operation: *h <- Code[f+3]*
h -> Heap

Count: 10

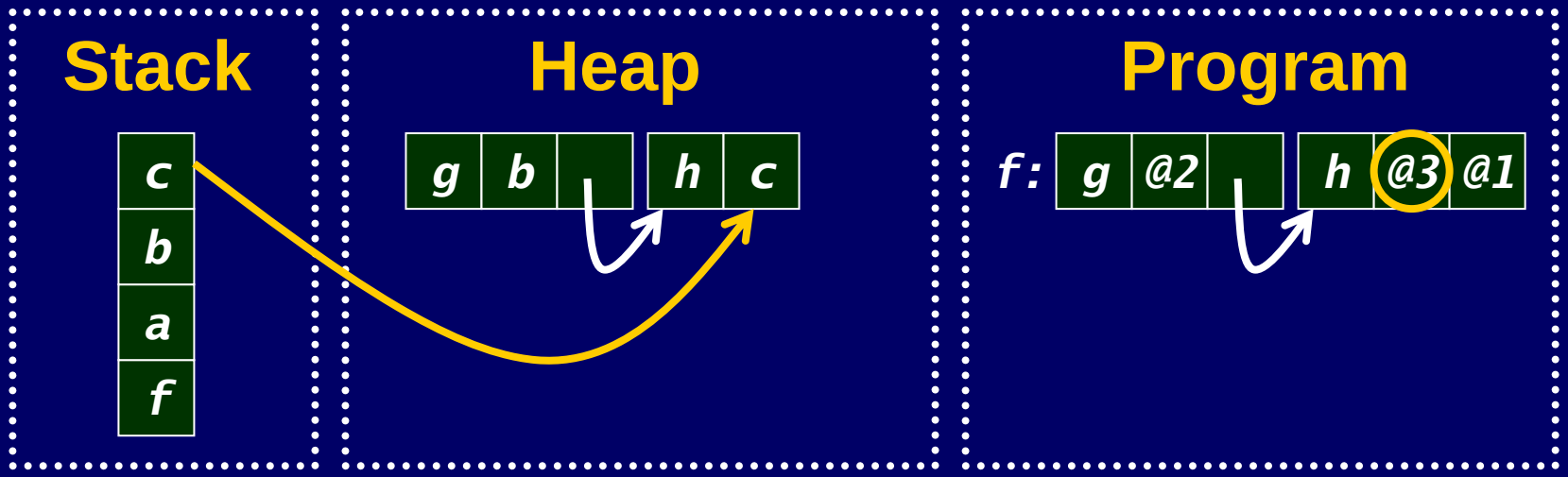


A Graph Reduction Machine

Operation:

```
arg <- Code[f+4]
c   <- Stack[arg]
c   -> Heap
```

Count: 13

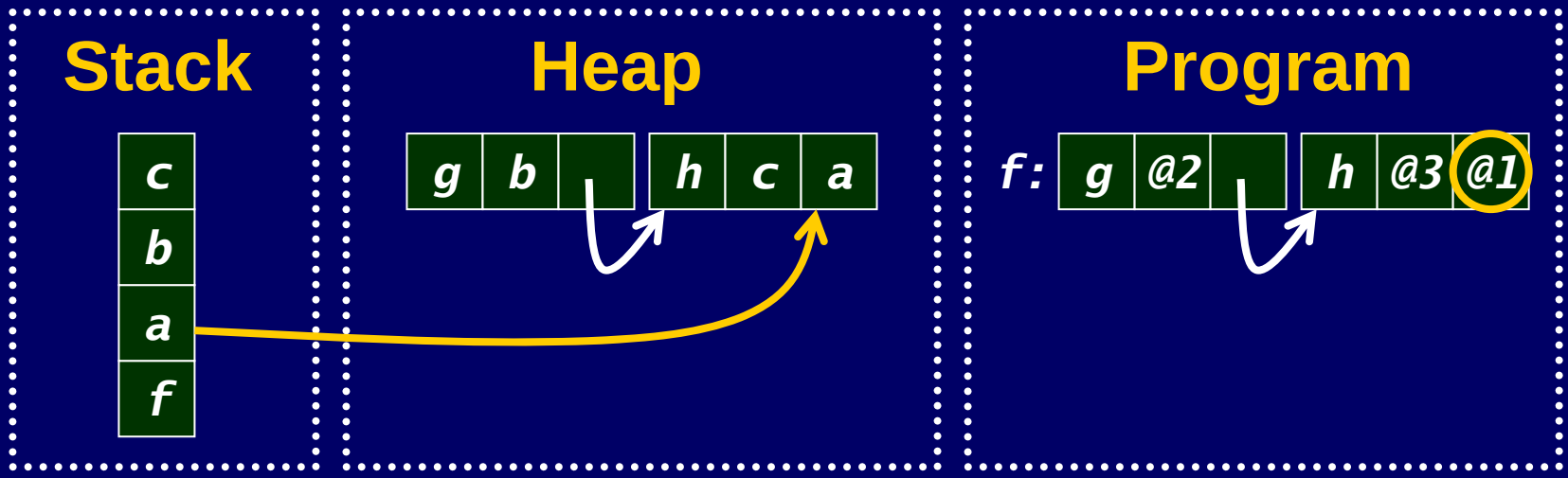


A Graph Reduction Machine

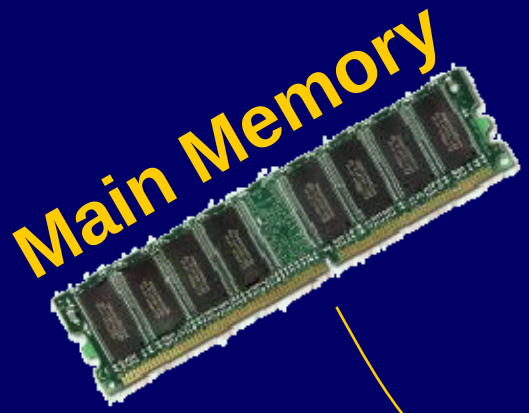
Operation:

```
arg <- Code[f+5]
a   <- Stack[arg]
a   -> Heap
```

Count: 16



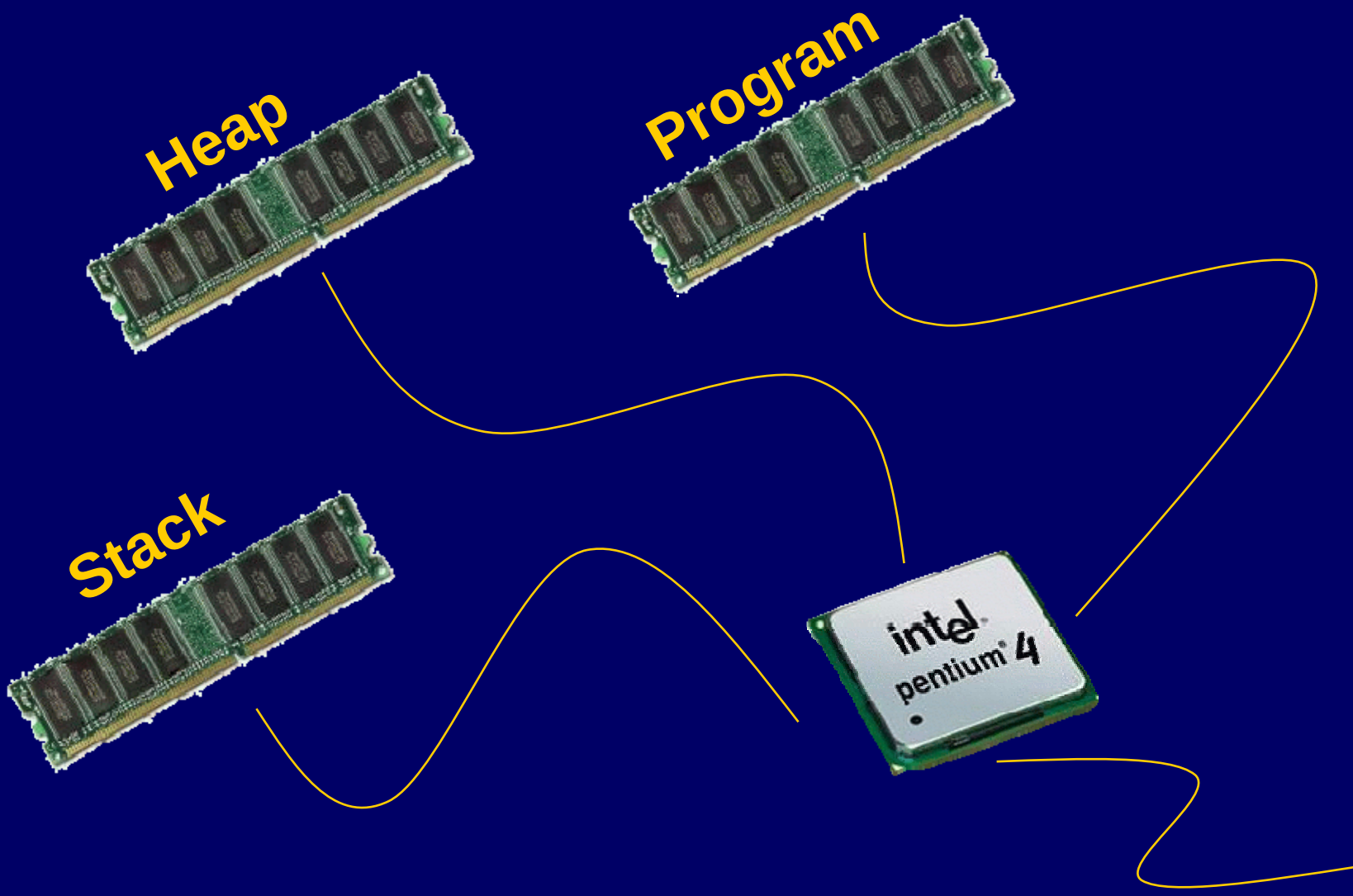
The von Neumann Bottleneck



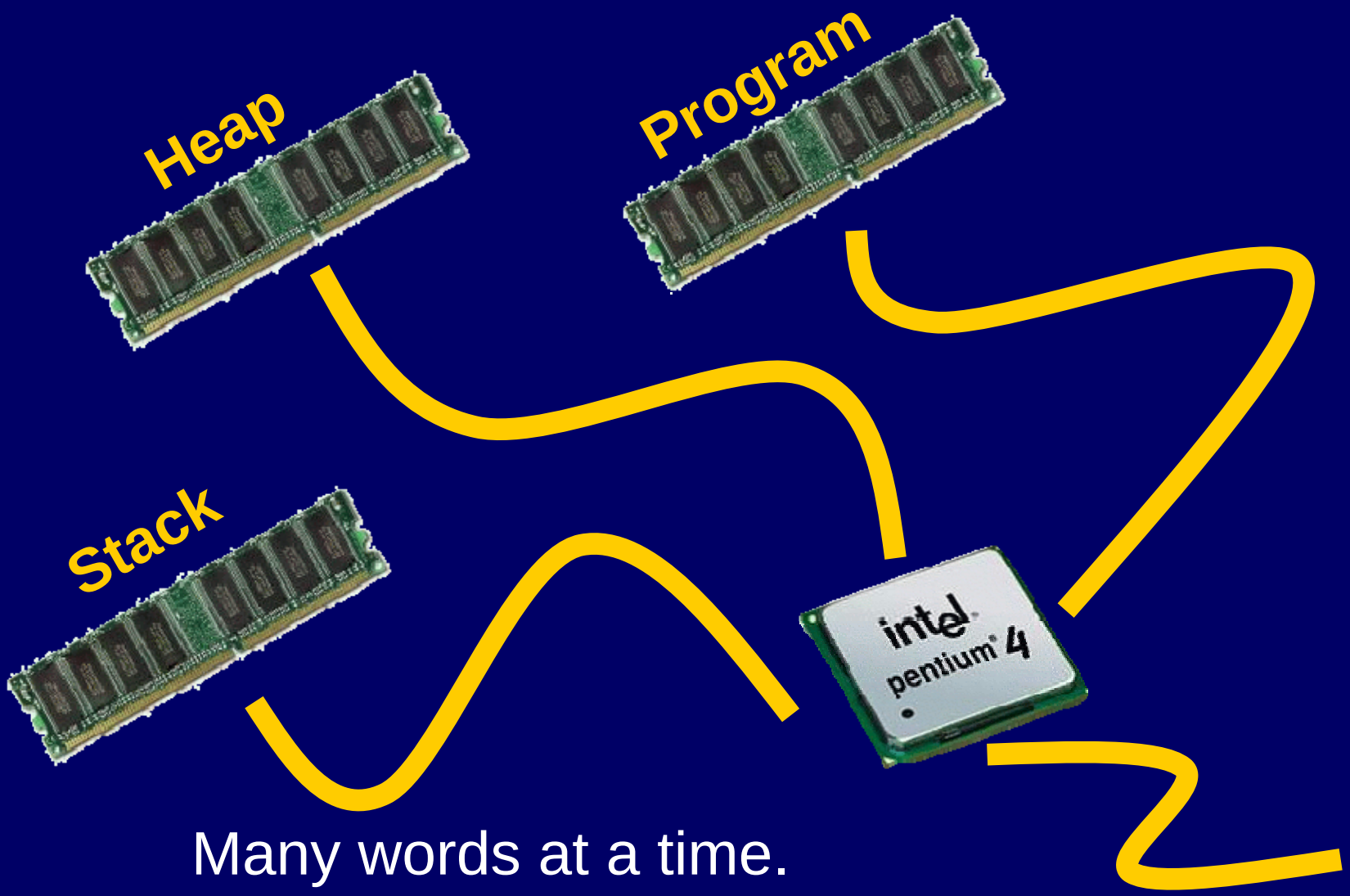
One word at a time.

Each of the **16** memory transactions must be done sequentially.

Widening the Bottleneck

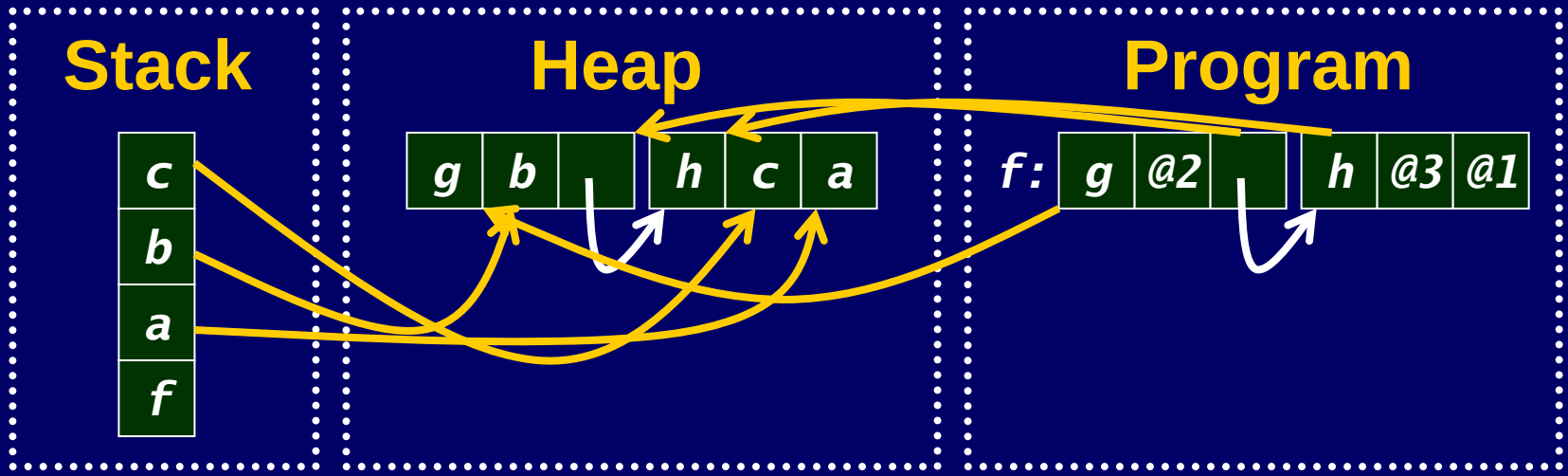


Widening the Bottleneck, again



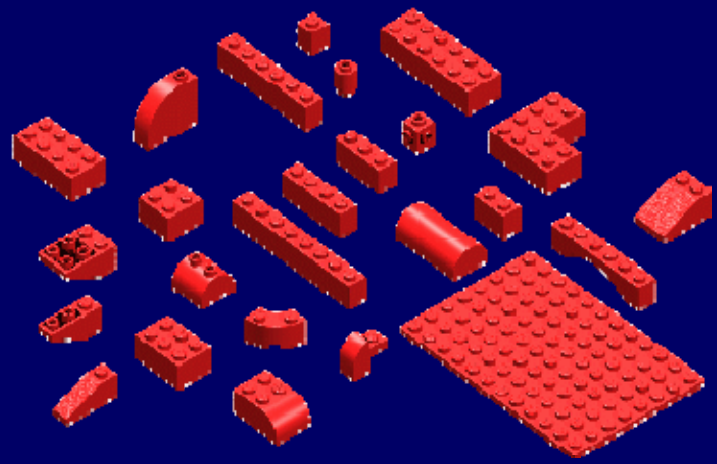
Many words at a time.

Applying a function “in one go”



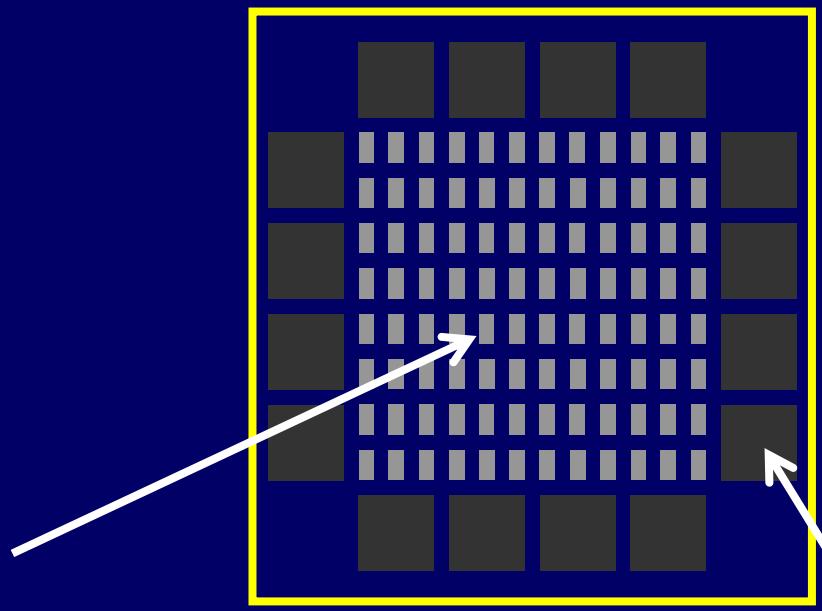
The Reduceron

Widening the von Neumann Bottleneck for Graph Reduction using an FPGA



An FPGA

A large, fixed set of components that can be connected together in any desired way.



Logic cells
(and-gates, flip-flops, …)

Memory blocks
(addressable separately)

Experimental Results

Wide Reduceron
(uses wide, parallel memories)

5x faster than

Narrow Reduceron
(single connection to memory)

Wide Reduceron
at 92MHz on Virtex-II FPGA

5x slower than

GHC -O2
(advanced optimising compiler)
at 2800MHz on Pentium-4 PC

(See thesis for theoretical results.)

Dinner Time!