

Go Wrap up

Parallel Architectures

Chris Rossbach

cs378 Fall 2018

10/15/2018

Outline for Today

- Questions?
- Administrivia
- Agenda
 - Go
 - Parallel Architectures (GPU background)
- Rob Pike's 2012 Go presentation is excellent, and I borrowed from it:
<https://talks.golang.org/2012/concurrency.slide>

Faux Quiz questions

- How are promises and futures different or the same as goroutines
- What is the difference between a goroutine and a thread?
- What is the difference between a channel and a lock?
- How is a channel different from a concurrent FIFO?
- What is the CSP model?
- What are the tradeoffs between explicit vs implicit naming in message passing?
- What are the tradeoffs between blocking vs. non-blocking send/receive in a shared memory environment? In a distributed one?
- What is hardware multi-threading; what problem does it solve?
- What is the difference between a vector processor and a scalar?
- Implement a parallel scan or reduction
- How are GPU workloads different from GPGPU workloads?
- How does SIMD differ from SIMT?
- List and describe some pros and cons of vector/SIMD architectures.
- GPUs historically have elided cache coherence. Why? What impact does it have on the the programmer?
- List some ways that GPUs use concurrency but not necessarily parallelism.

Google Search

- Workload:
- Accept query
- Return page of results (with ugh, ads)
- Get search results by sending query to
 - Web Search
 - Image Search
 - YouTube
 - Maps
 - News, etc
- How to implement this?

Search 1.0

- Google function takes query and returns a slice of results (strings)
- Invokes Web, Image, Video search serially

```
func Google(query string) ([]Result) {  
    results = append(results, Web(query))  
    results = append(results, Image(query))  
    results = append(results, Video(query))  
    return  
}
```

Search 2.0

- Run Web, Image, Video searches concurrently, wait for results
- No locks, conditions, callbacks

```
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```

Search 2.1

- Don't wait for slow servers: No locks, conditions, callbacks!

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
    case result := <-c:
        results = append(results, result)
    case <-timeout:
        fmt.Println("timed out")
        return
    }
}
return
```

Search 3.0

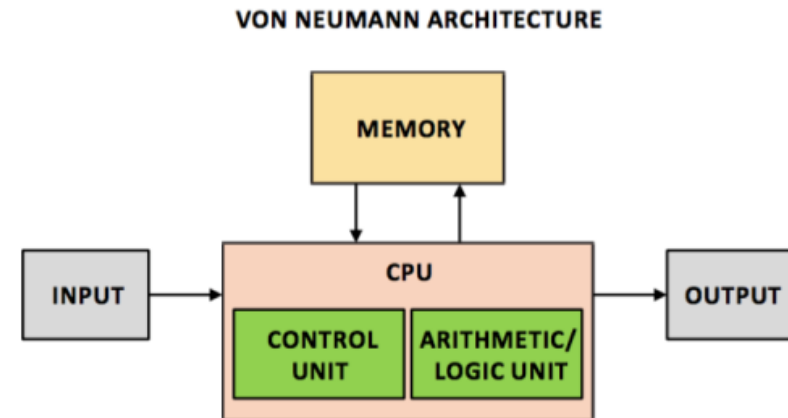
- Reduce tail latency with replication. No locks, conditions, callbacks!

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
    case result := <-c:
        results = append(results, result)
    case <-timeout:
        fmt.Println("timed out")
        return
    }
}
return
```

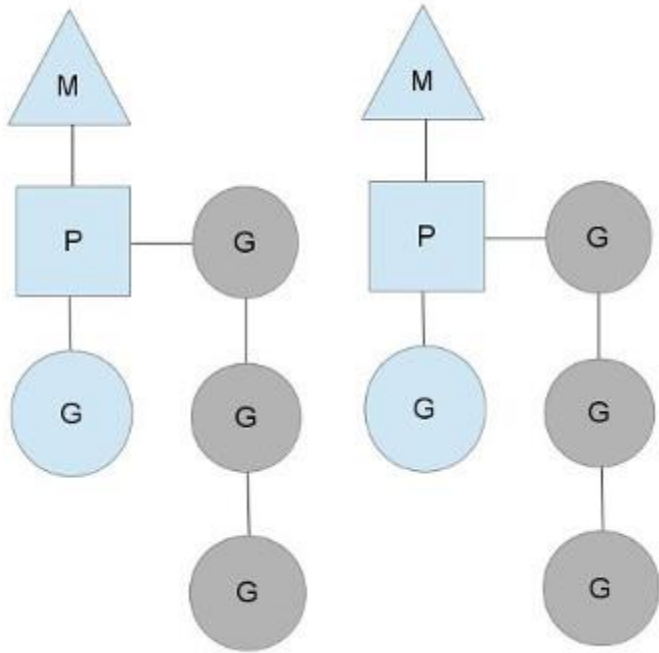
```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```


Go: magic? ...or *threadpools and concurrent Qs*?

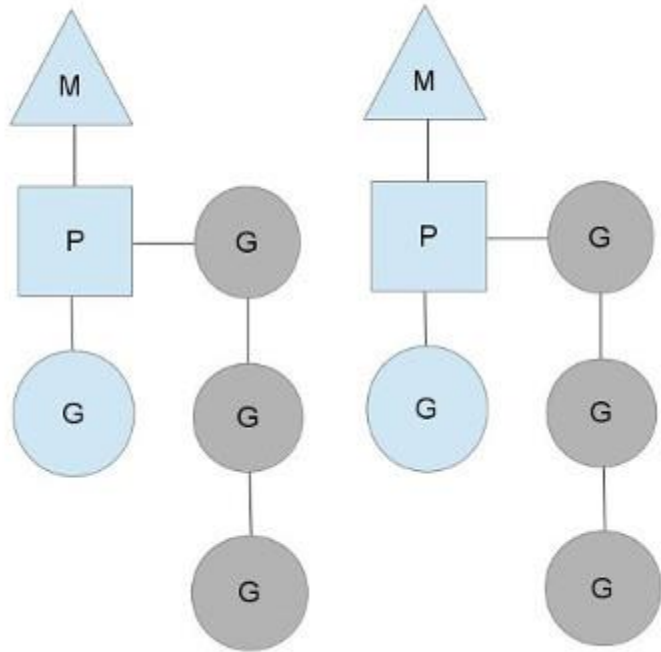
- We've seen several abstractions for
 - Control flow/execution
 - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
 - Why is it OK to have 100,000s of goroutines?
 - Why isn't composition an issue?



Go implementation details

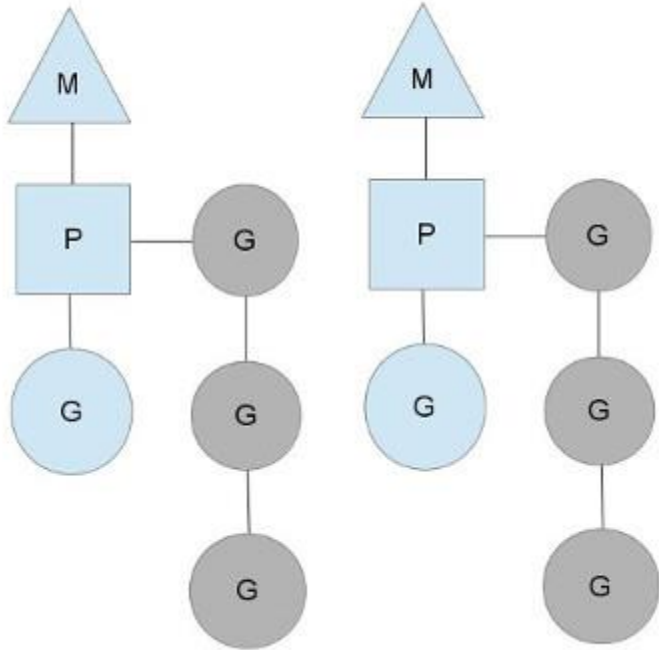


Go implementation details



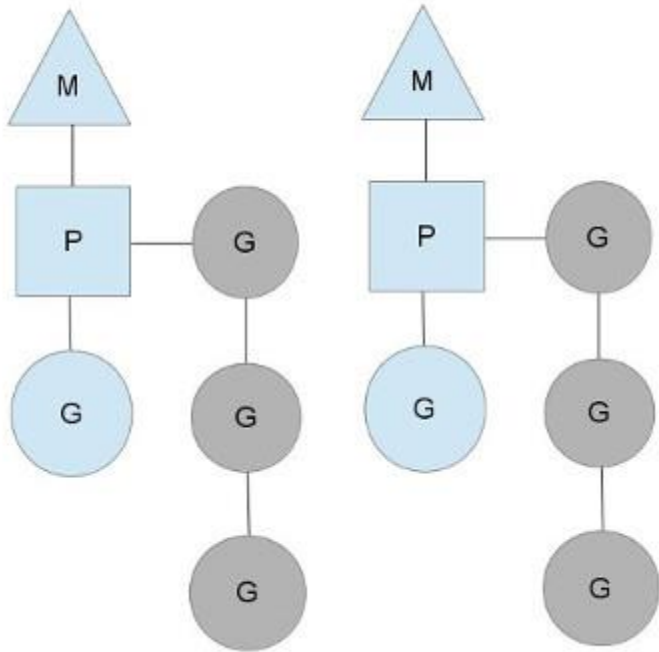
- M = “machine” → OS thread

Go implementation details



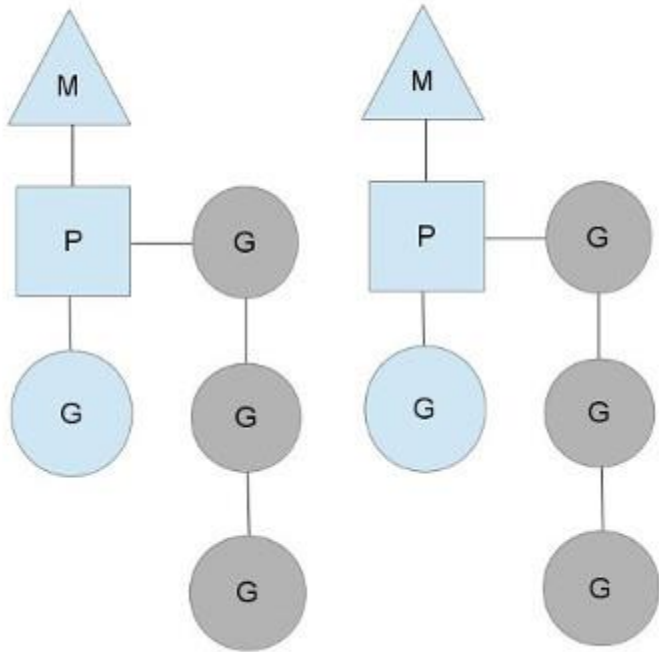
- M = “machine” → OS thread
- P = (processing) context

Go implementation details



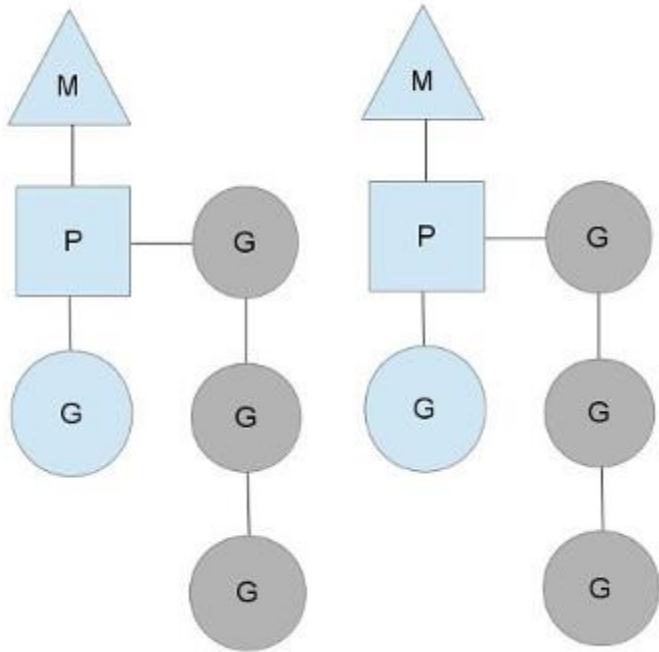
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

Go implementation details



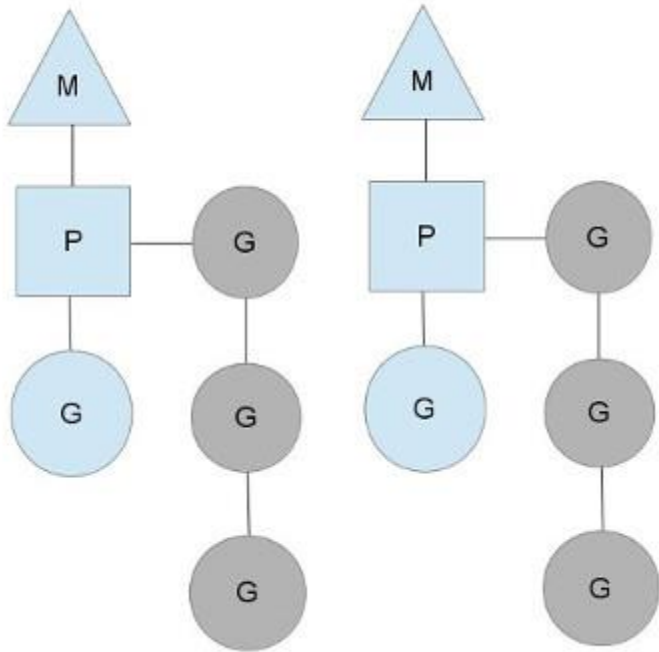
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines

Go implementation details



- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
 - Switch out on complete or block
 - Very light weight (fibers!)
 - Scheduler does work-stealing

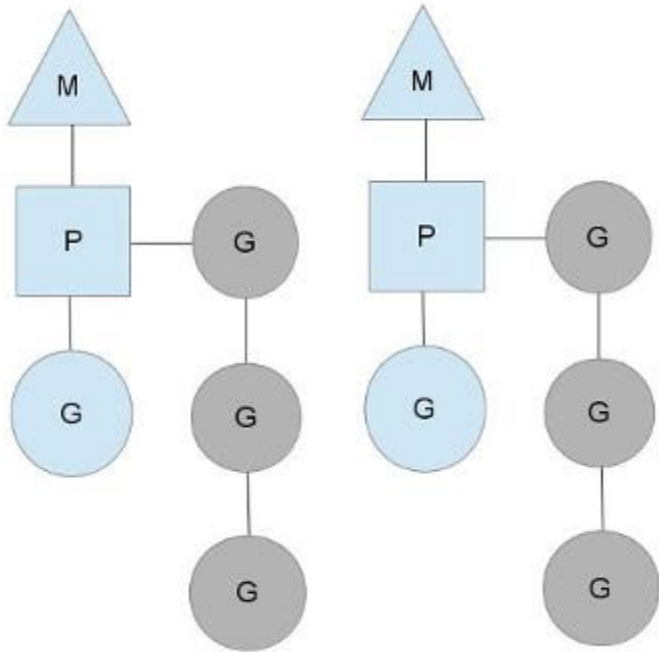
Go implementation details



- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

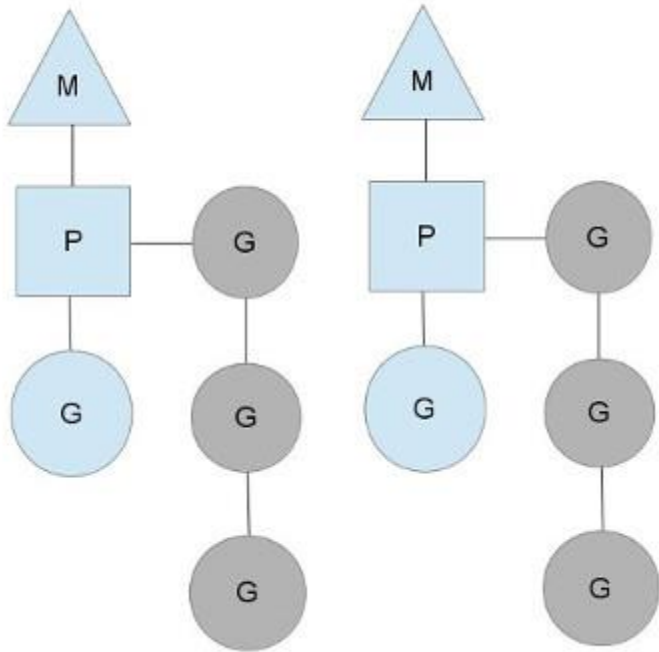
```
struct G
{
    byte*    stackguard; // stack guard information
    byte*    stackbase;  // base of stack
    byte*    stack0;     // current stack pointer
    byte*    entry;      // initial function
    void*    param;      // passed parameter on wakeup
    int16    status;     // status
    int32    goid;       // unique id
    M*       lockedm;    // used for locking M's and G's
    ...
};
```


Go implementation details



- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
 - Switch out on complete or block
 - Very light weight (fibers!)
 - Scheduler does work-stealing

Go implementation details

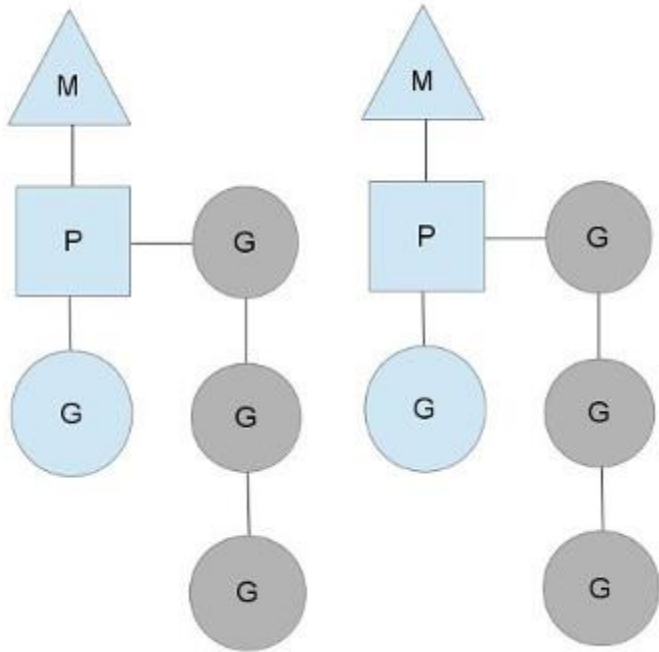


- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

```
struct M
{
    G*    curg;           // current running goroutine
    int32 id;           // unique id
    int32 locks;        // locks held by this M
    MCache *mcache;    // cache for this thread
    G*    lockedg;      // used for locking M's and G's
    uintptr createstack [32]; // Stack that created this thread
    M*    nextwaitm;    // next M waiting for lock
    ...
};
```

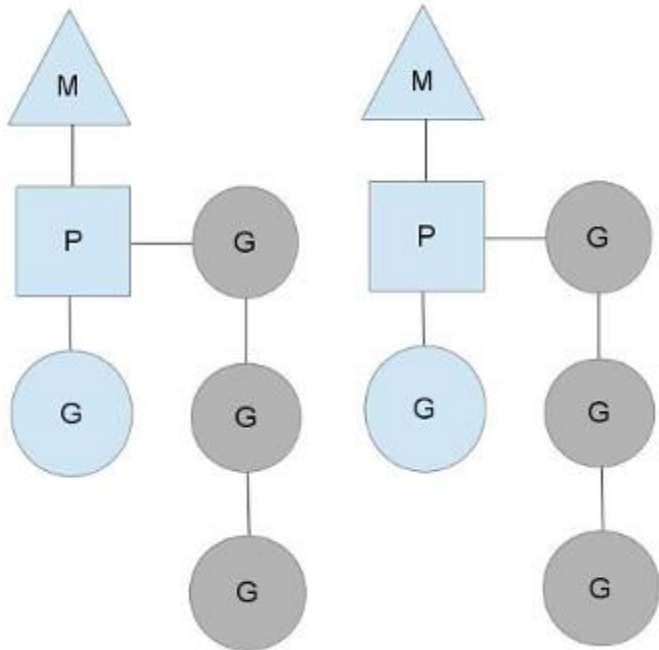
Go implementation details

- M = “machine” → OS thread



```
struct Sched {  
    Lock; // global sched lock.  
          // must be held to edit G or M queues  
  
    G *gfree; // available g's (status == Gdead)  
    G *ghead; // g's waiting to run queue  
    G *gtail; // tail of g's waiting to run queue  
    int32 gwait; // number of g's waiting to run  
    int32 gcount; // number of g's that are alive  
    int32 grunning; // number of g's running on cpu  
                  // or in syscall  
  
    M *mhead; // m's waiting for work  
    int32 mwait; // number of m's waiting for work  
    int32 mcount; // number of m's that have been created  
    ...  
};
```

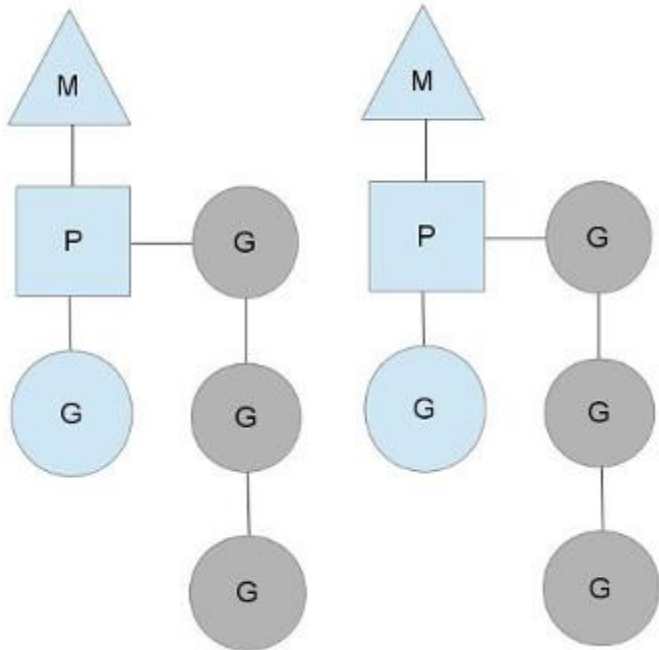
Go implementation details



- M = “machine” → OS thread

```
struct Sched {  
    Lock; // global sched lock.  
          // must be held to edit G or M queues  
  
    G *gfree; // available g's (status == Gdead)  
    G *ghead; // g's waiting to run queue  
    G *gtail; // tail of g's waiting to run queue  
    int32 gwait; // number of g's waiting to run  
    int32 gcount; // number of g's that are alive  
    int32 grunning; // number of g's running on cpu  
                  // or in syscall  
  
    M *mhead; // m's waiting for work  
    int32 mwait; // number of m's waiting for work  
    int32 mcount; // number of m's that have been created  
    ...  
};
```

Go implementation details



- M = “machine” → OS thread

```
struct Sched {  
    Lock; // global sched lock.  
          // must be held to edit G or M queues  
  
    G *gfree; // available g's (status == Gdead)  
    G *ghead; // g's waiting to run queue  
    G *gtail; // tail of g's waiting to run queue  
    int32 gwait; // number of g's waiting to run  
    int32 gcount; // number of g's that are alive  
    int32 grunning; // number of g's running on cpu  
                  // or in syscall  
  
    M *mhead; // m's waiting for work  
    int32 mwait; // number of m's waiting for work  
    int32 mcount; // number of m's that have been created  
    ...  
};
```

1000s of go routines?

```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```

1000s of go routines?

```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```

- Creates a channel
- Creates “consumers” goroutines
- Each of them tries to read from the channel
- Main either:
 - Sleeps for 1 second, closes the channel
 - sends “consumers” values

1000s of go routines?

```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel read")
            }
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```

- Creates a channel
- Creates “consumers” goroutines
- Each of them tries to read from the channel
- Main either:
 - Sleeps for 1 second, closes the channel
 - sends “consumers” values

```
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10
testQ: 1.0016706s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100
testQ: 1.0011655s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000
testQ: 1.0084796s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10000
testQ: 1.0547925s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100000
testQ: 1.3907835s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000000
testQ: 4.2405814s
```


Channel implementation

- You can just read it:
 - <https://golang.org/src/runtime/chan.go>
- Some highlights

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```


Channel implementation

- Y
- S

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    if c == nil {
        if !block {
            return false
        }
        gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    } Race detection! Cool!

    if raceenabled {
        racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
    }
}
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
193 }
```

Channel implementation

- You can just read it:
 - <https://golang.org/src/runtime/chan.go>
- Some highlights

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```

Channel implementation

- You can just read it:
 - <https://golang.org/src/runtime/chan.go>
- Some highlights

```
if sg := c.recvq.dequeue(); sg != nil {  
    // Found a waiting receiver. We pass the value we want to send  
    // directly to the receiver, bypassing the channel buffer (if any).  
    send(c, sg, ep, func() { unlock(&c.lock) }, 3)  
    return true  
}
```

```
122 // entry point for c <- x from compiled code  
123 //go:nosplit  
124 func chansend1(c *hchan, elem unsafe.Pointer) {  
125     chansend(c, elem, true, getcallerpc())  
126 }  
127  
128 /*  
129 * generic single channel send/recv  
130 * If block is not nil,  
131 * then the protocol will not  
132 * sleep but return if it could  
133 * not complete.  
134 *  
135 * sleep can wake up with g.param == nil  
136 * when a channel involved in the sleep has  
137 * been closed. it is easiest to loop and re-run  
138 * the operation; we'll see that it's now closed.  
139 */  
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {  
141     if c == nil {  
142         if !block {  
143             return false  
144         }  
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)  
146         throw("unreachable")  
147     }  
148  
149     if debugChan {  
150         print("chansend: chan=", c, "\n")  
151     }  
152  
153     if raceenabled {  
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))  
155     }  
156  
157     // Fast path: check for failed non-blocking operation without acquiring the lock.  
158     //  
159     // After observing that the channel is not closed, we observe that the channel is  
160     // not ready for sending. Each of these observations is a single word-sized read  
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).  
162     // Because a closed channel cannot transition from 'ready for sending' to  
163     // 'not ready for sending', even if the channel is closed between the two observations,  
164     // they imply a moment between the two when the channel was both not yet closed  
165     // and not ready for sending. We behave as if we observed the channel at that moment,  
166     // and report that the send cannot proceed.  
167     //  
168     // It is okay if the reads are reordered here: if we observe that the channel is not  
169     // ready for sending and then observe that it is not closed, that implies that the  
170     // channel wasn't closed during the first observation.  
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||  
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {  
173         return false  
174     }  
175  
176     var t0 int64  
177     if blockprofrate > 0 {  
178         t0 = cputicks()  
179     }  
180  
181     lock(&c.lock)  
182  
183     if c.closed != 0 {  
184         unlock(&c.lock)  
185         panic(plainError("send on closed channel"))  
186     }  
187  
188     if sg := c.recvq.dequeue(); sg != nil {  
189         // Found a waiting receiver. We pass the value we want to send  
190         // directly to the receiver, bypassing the channel buffer (if any).  
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)  
192     }  
193 }  
194
```

Channel implementation

- You can just read it:
 - <https://golang.org/src/runtime/chan.go>
- Some highlights

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```

2)

)

quiring the lock.

at the channel is
: word-sized read
ig on kind of channel).
nding" to
n the two observations,
not yet closed
annel at that moment,

it the channel is not
it implies that the

: == nil) ||

```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) } 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```

2)

)

quiring the lock.

at the channel is
: word-sized read
ig on kind of channel).
nding" to
n the two observations,
not yet closed
annel at that moment,

it the channel is not
it implies that the

: == nil) ||

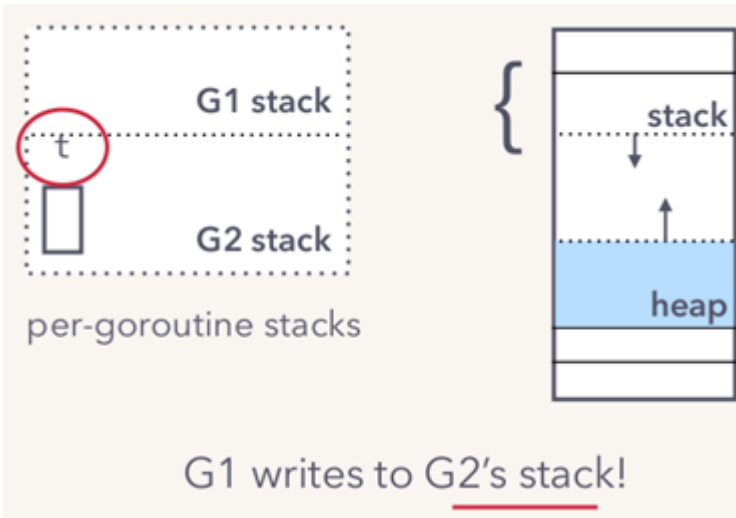
```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) } 3)
```


Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```



```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) }, 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```

2)

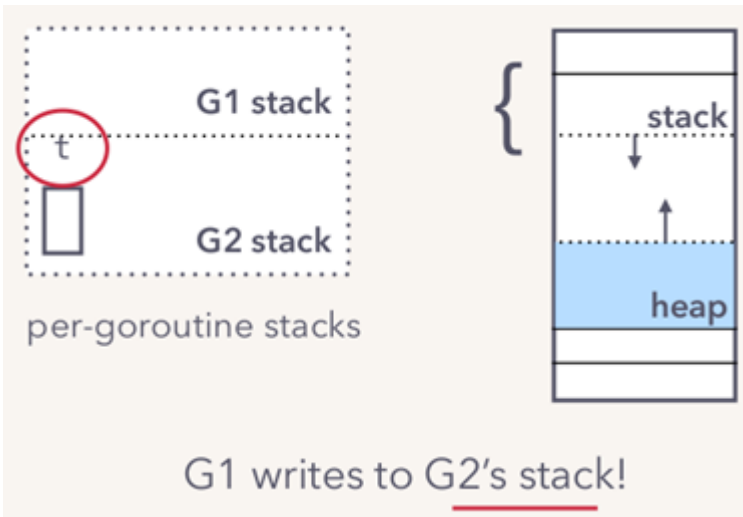
)

quiring the lock.

at the channel is
: word-sized read
ig on kind of channel).
inding" to
in the two observations,
not yet closed
annel at that moment,

it the channel is not
it implies that the

: == nil) ||



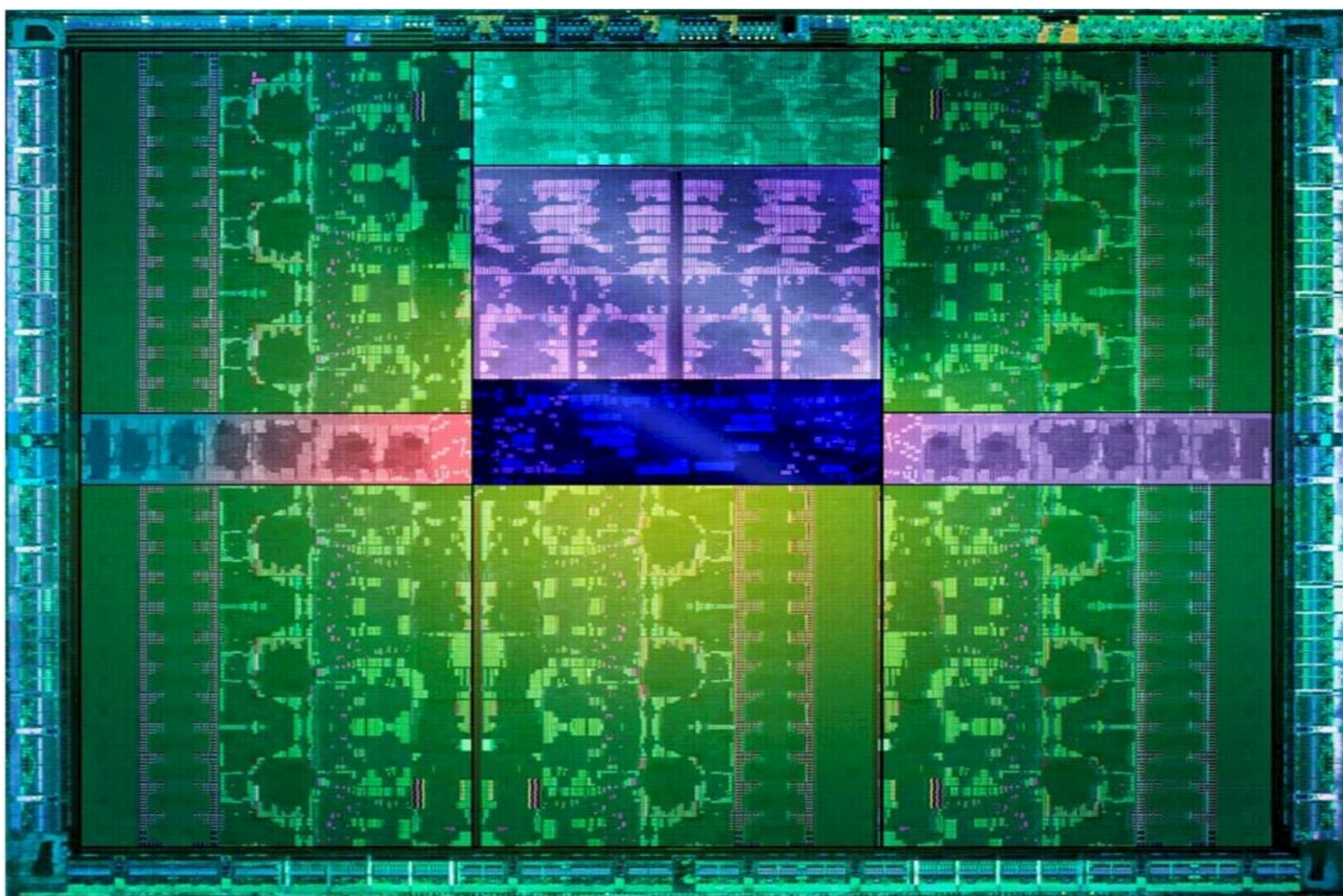
Transputers did this in hardware in the 90s btw.

```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) } 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/src/runtime/chan.go>
- Some highlights:
 - Race detection built in
 - Fast path just write to receiver stack
 - Often has no capacity → scheduler hint!
 - Buffered channel implementation fairly standard

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```



A modern GPU: Volta V100



A modern GPU: Volta V100



- 80 SMs
- Streaming Multiprocessor



A modern GPU: Volta V100



Also:
CU or ACE

- 80 SMs
- Streaming Multiprocessor



A modern GPU: Volta V100



- 80 SMs
- Streaming Multiprocessor



A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS

A modern GPU



- 80 SMs
- Streaming Multiprocessor
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS

Roughly: all of k-means 1,000s X/sec

A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS

A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores

A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
 - 4096-bit bus
 - No cache coherence!

A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
 - 4096-bit bus
 - No cache coherence!
- 16 GB memory
 - PCIe-attached

A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
 - 4096-bit bus
 - No cache coherence!
- 16 GB memory
 - PCIe-attached



A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
 - 4096-bit bus
 - No cache coherence!
- 16 GB memory
 - PCIe-attached



A modern GPU



- 80 SMs
 - Streaming Multiprocessor
 - 64 cores/SM
 - 5210 threads!
 - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
 - 4096-bit bus
 - No cache coherence!
- 16 GB memory
 - PCIe-attached



How do you program a machine like this? pthread_create()?

GPUs: Outline

- Background from many areas
 - Architecture
 - Vector processors
 - Hardware multi-threading
 - Graphics
 - Graphics pipeline
 - Graphics programming models
 - Algorithms
 - parallel architectures → parallel algorithms
- Programming GPUs
 - CUDA
 - Basics: getting something working
 - Advanced: making it perform

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true)  
        do_next_instruction();  
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true)  
        do_next_instruction();  
}
```

```
do_next_instruction() {  
    instruction = fetch();  
    ops, regs = decode(instruction);  
    execute_calc_addr(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    v  
    main() {  
        pthread_create(do_instructions);  
        pthread_create(do_decode);  
        pthread_create(do_execute);  
        ...  
        pthread_join(...);  
        ...  
    }  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    v  
    main() {  
        pthread_create(do_instructions);  
    }  
    pthread_create(do_decode);  
    pthread_create(do_execute);  
    ...  
    pthread_join(...);  
    ...  
}  
access_memory(ops, regs);  
write_back(regs);  
}
```

```
do_instructions() {  
    while(true) {  
        instruction = fetch();  
        enqueue(DECODE, instruction);  
    }  
}
```

```
do_decode() {  
    while(true) {  
        instruction = dequeue();  
        ops, regs = decode(instruction);  
        enqueue(EX, instruction);  
    }  
}
```

```
do_execute() {  
    while(true) {  
        instruction = dequeue();  
        execute_calc_addrs(ops, regs);  
        enqueue(MEM, instruction);  
    }  
}
```

....

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true) {  
        do_next_instruction();  
    }  
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true) {  
        do_next_instruction();  
    }  
}
```

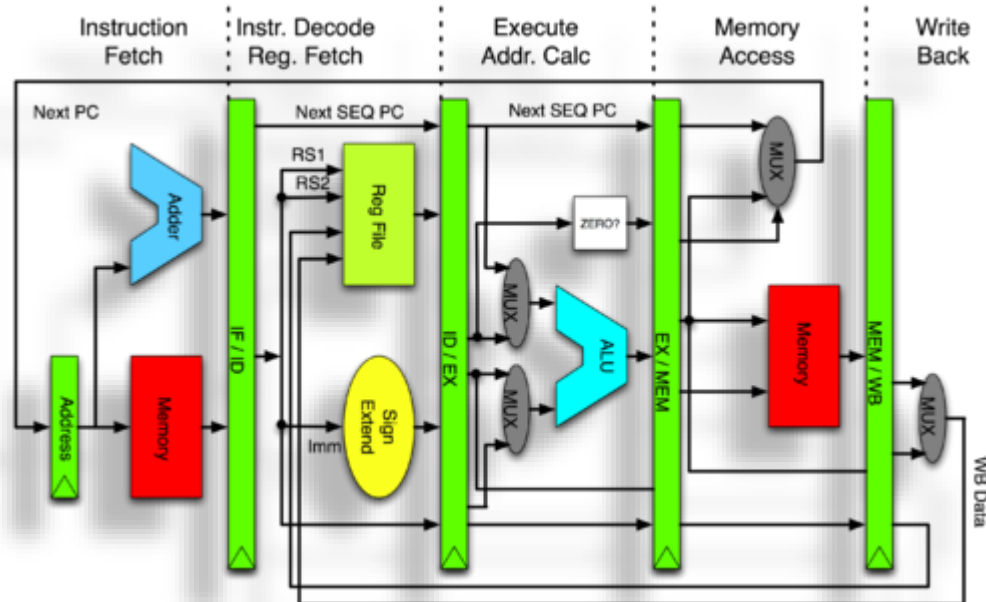
```
do_next_instruction() {  
    instruction = fetch();  
    ops, regs = decode(instruction);  
    execute_calc_addr(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```


Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true) {  
        do_next_instruction();  
    }  
}
```

```
do_next_instruction() {  
    instruction = fetch();  
    ops, regs = decode(instruction);  
    execute_calc_addr(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

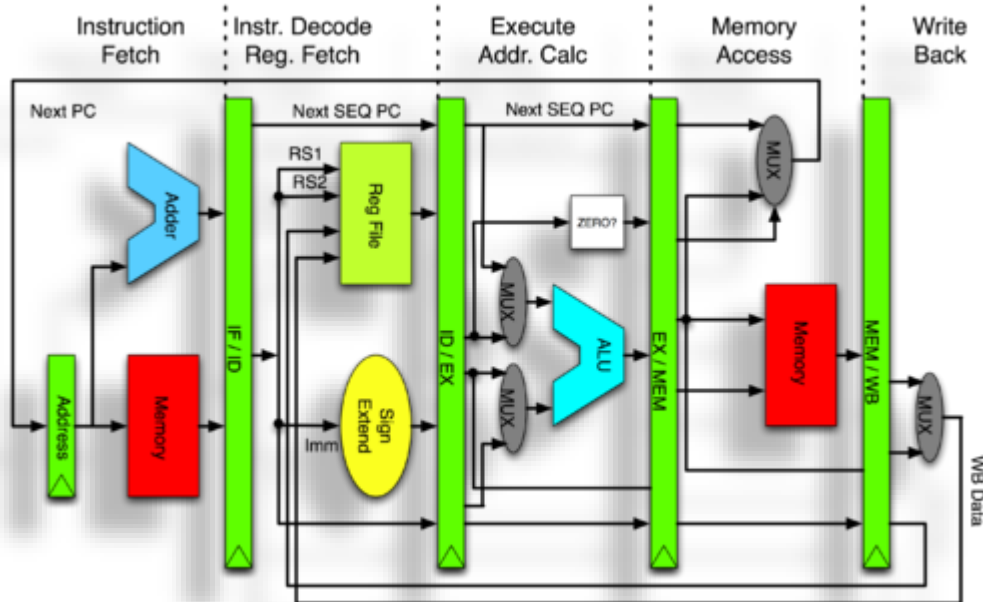


Architecture Review: Pipelines

Processor algorithm:

```
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```



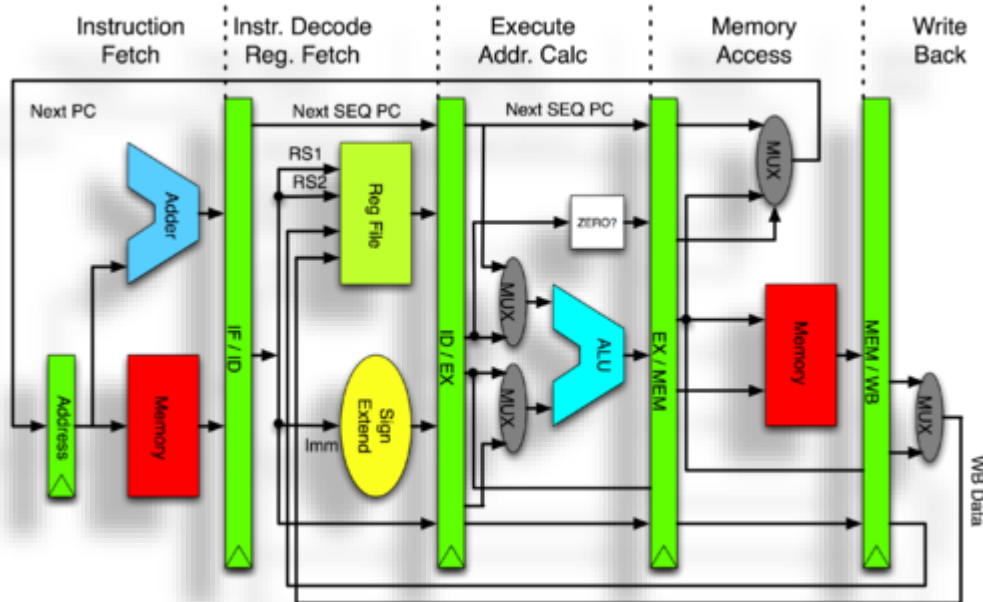
Instr No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Architecture Review: Pipelines

Processor algorithm:

```
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addr(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```



Instr No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

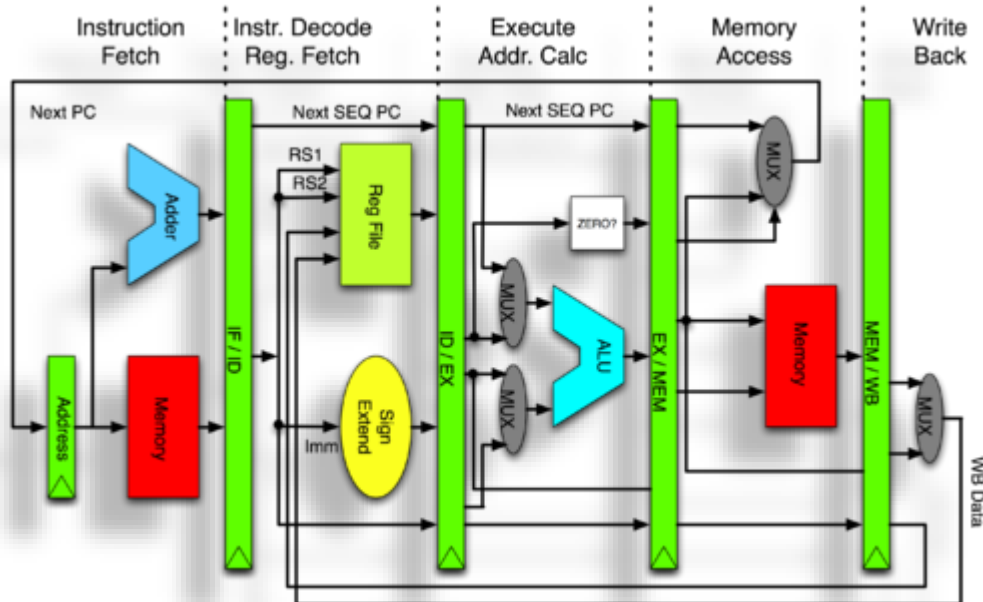
What is the name of this kind of parallelism?

Architecture Review: Pipelines

Processor algorithm:

```
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addr(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```



Instr No.	Pipeline Stage					
	IF	ID	EX	MEM	WB	
1	IF	ID	EX	MEM	WB	
2		IF	ID	EX	MEM	WB
3			IF	ID	EX	MEM
4				IF	ID	EX



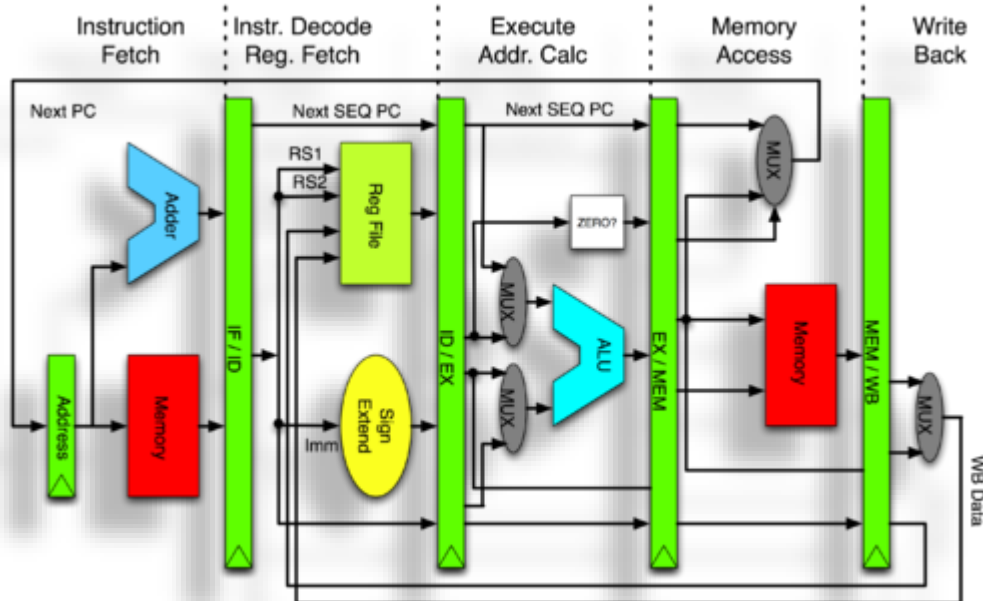
Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?
What is the name of this kind of parallelism?

Architecture Review: Pipelines

Processor algorithm:

```
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addr(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```



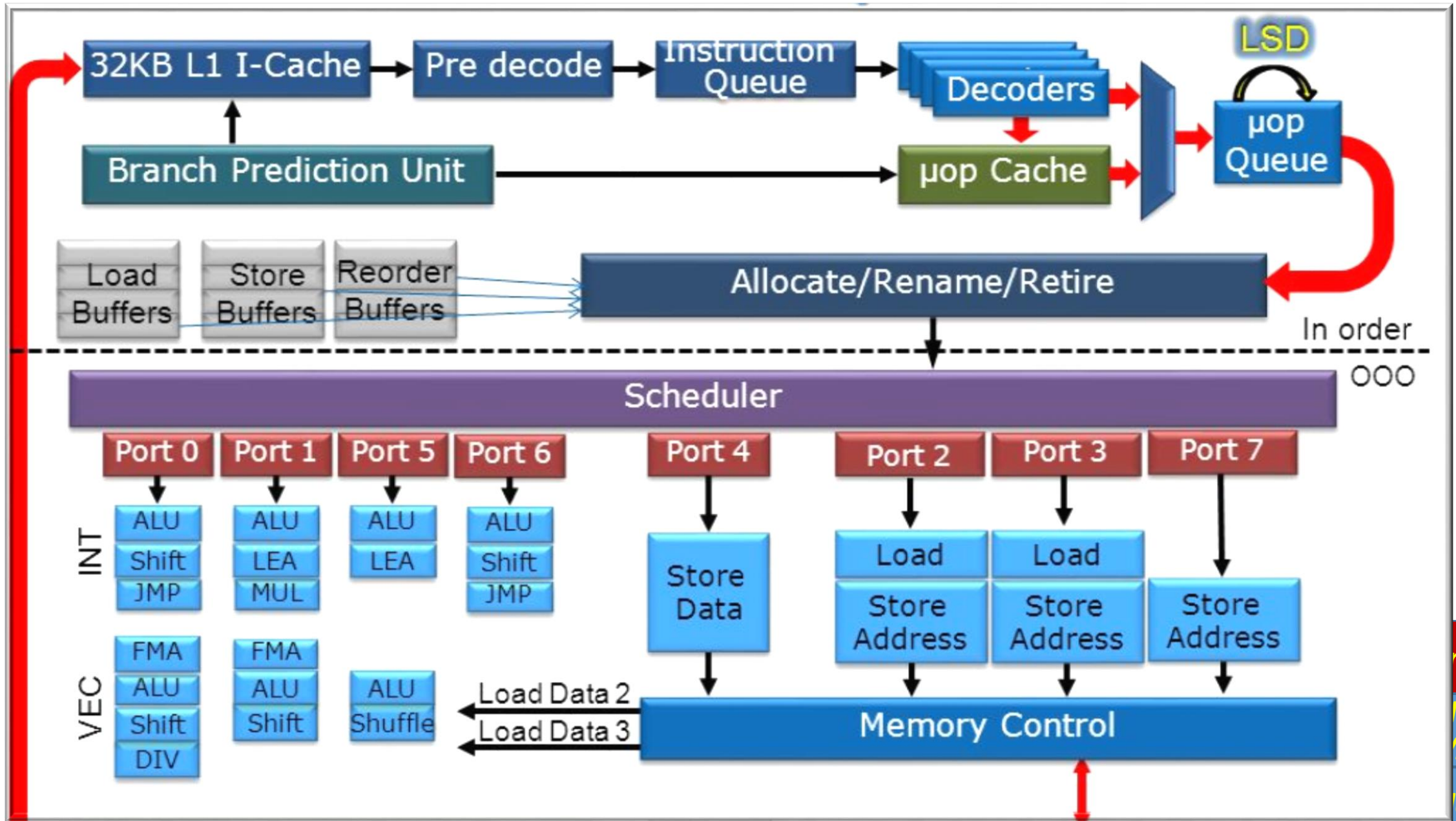
Instr No.	Pipeline Stage					
1	IF	ID	EX	MEM	WB	
2		IF	ID	EX	MEM	WB



How can we get **more parallelism?**

**Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?**

What is the name of this kind of parallelism?



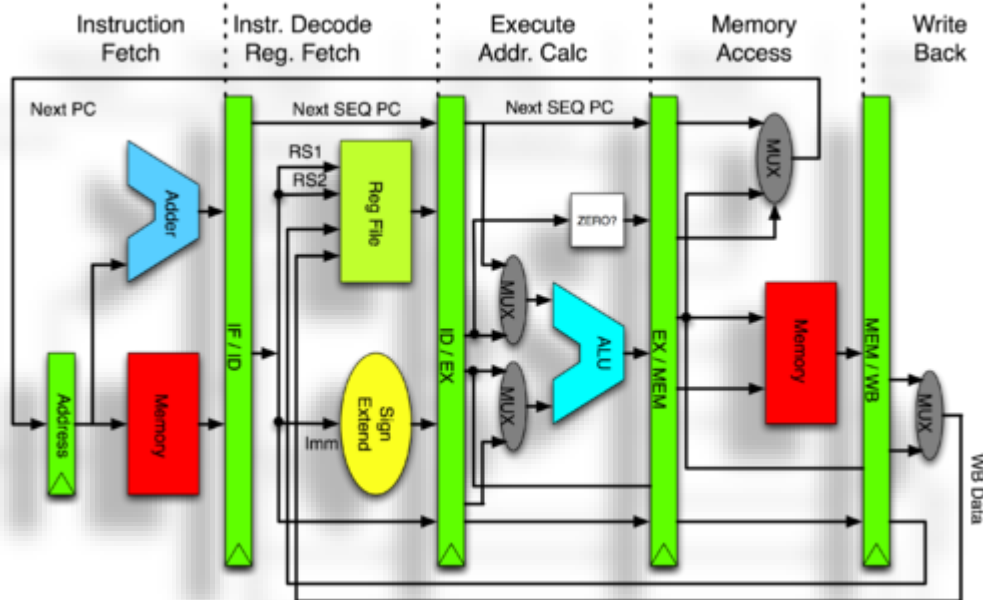
m?
stalls?
elism?

Architecture Review: Pipelines

Processor algorithm:

```
main() {  
    while(true) {  
        do_next_instruction();  
    }  
}
```

```
do_next_instruction() {  
    instruction = fetch();  
    ops, regs = decode(instruction);  
    execute_calc_addr(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```



Instr No.	Pipeline Stage					
1	IF	ID	EX	MEM	WB	
2		IF	ID	EX	MEM	WB



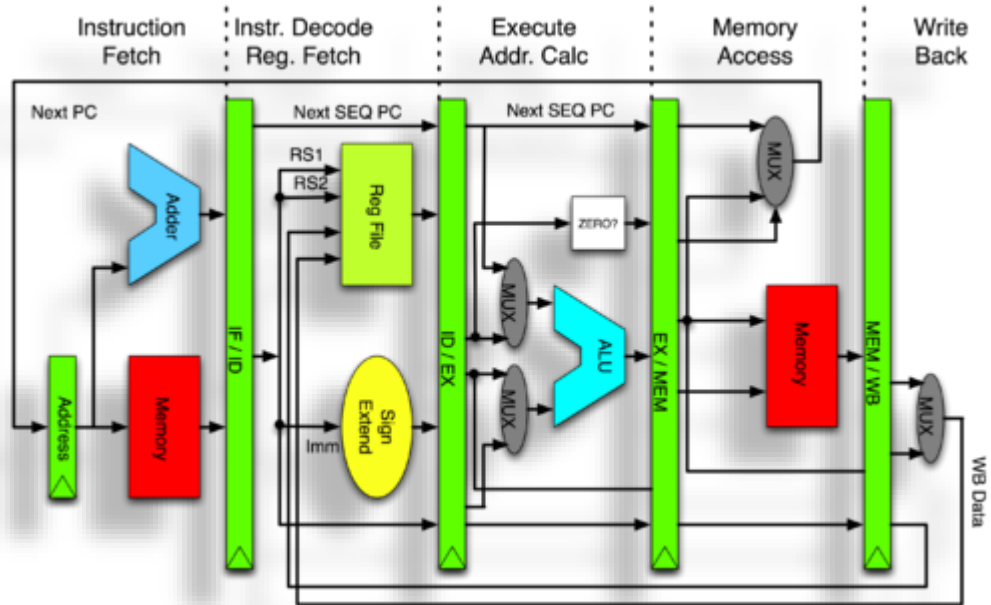
How can we get *more* parallelism?

**Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?**

What is the name of this kind of parallelism?

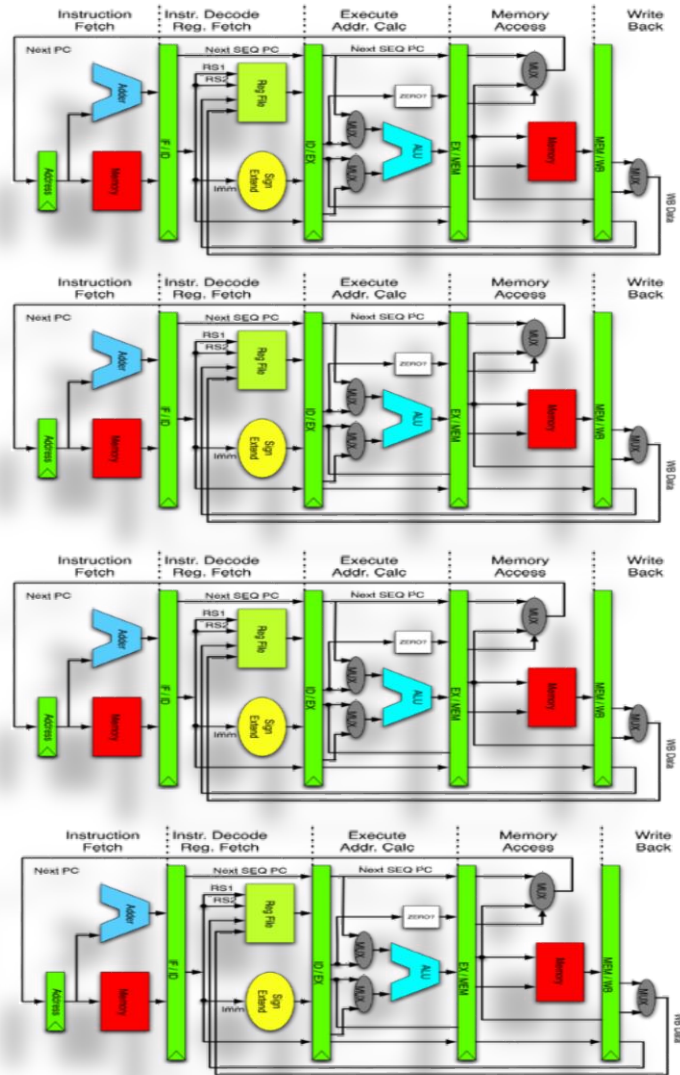
Multi-core/SMPs

Multi-core/SMPs

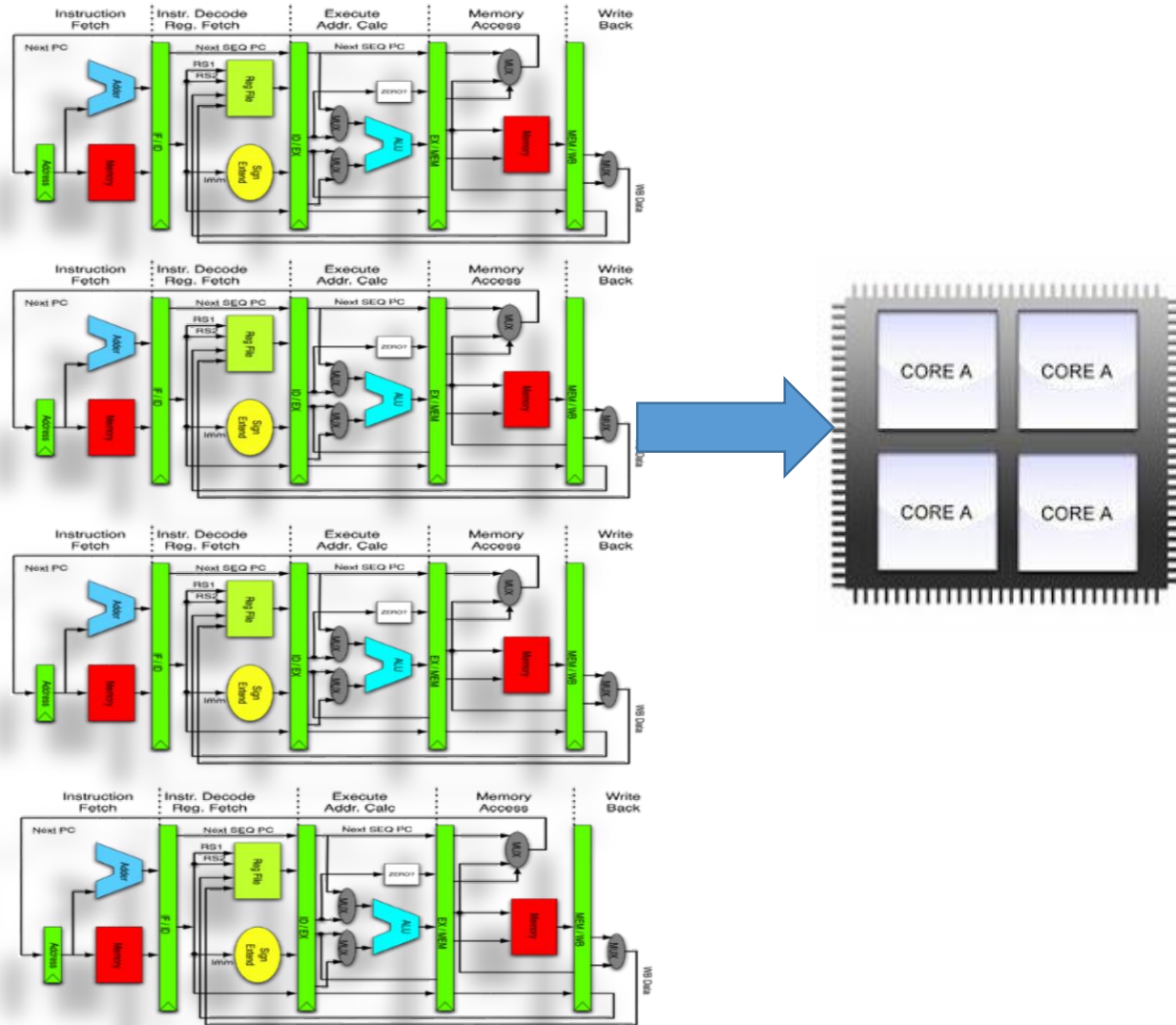


Multi-core/SMPs

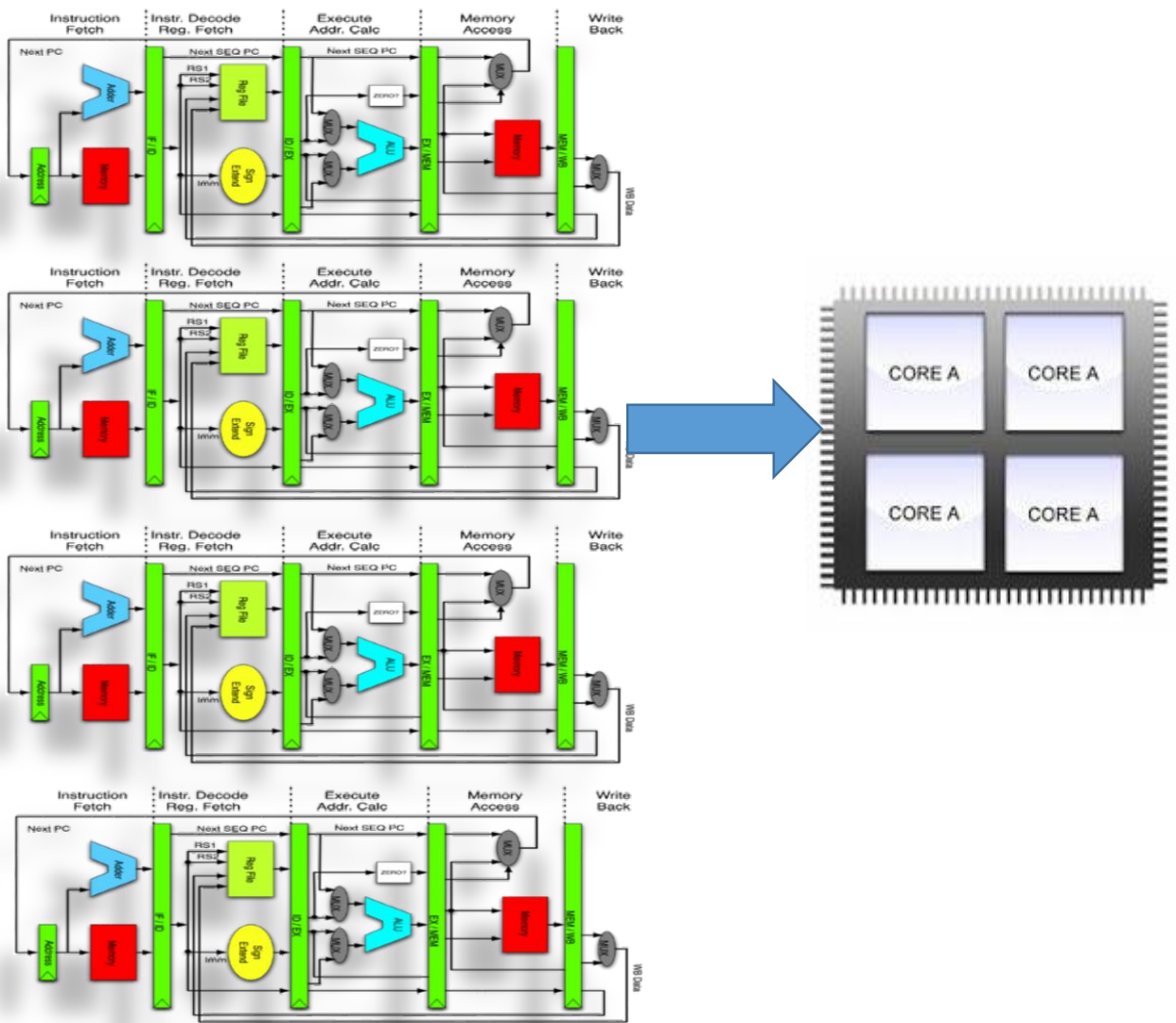
Multi-core/SMPs



Multi-core/SMPs



Multi-core/SMPs

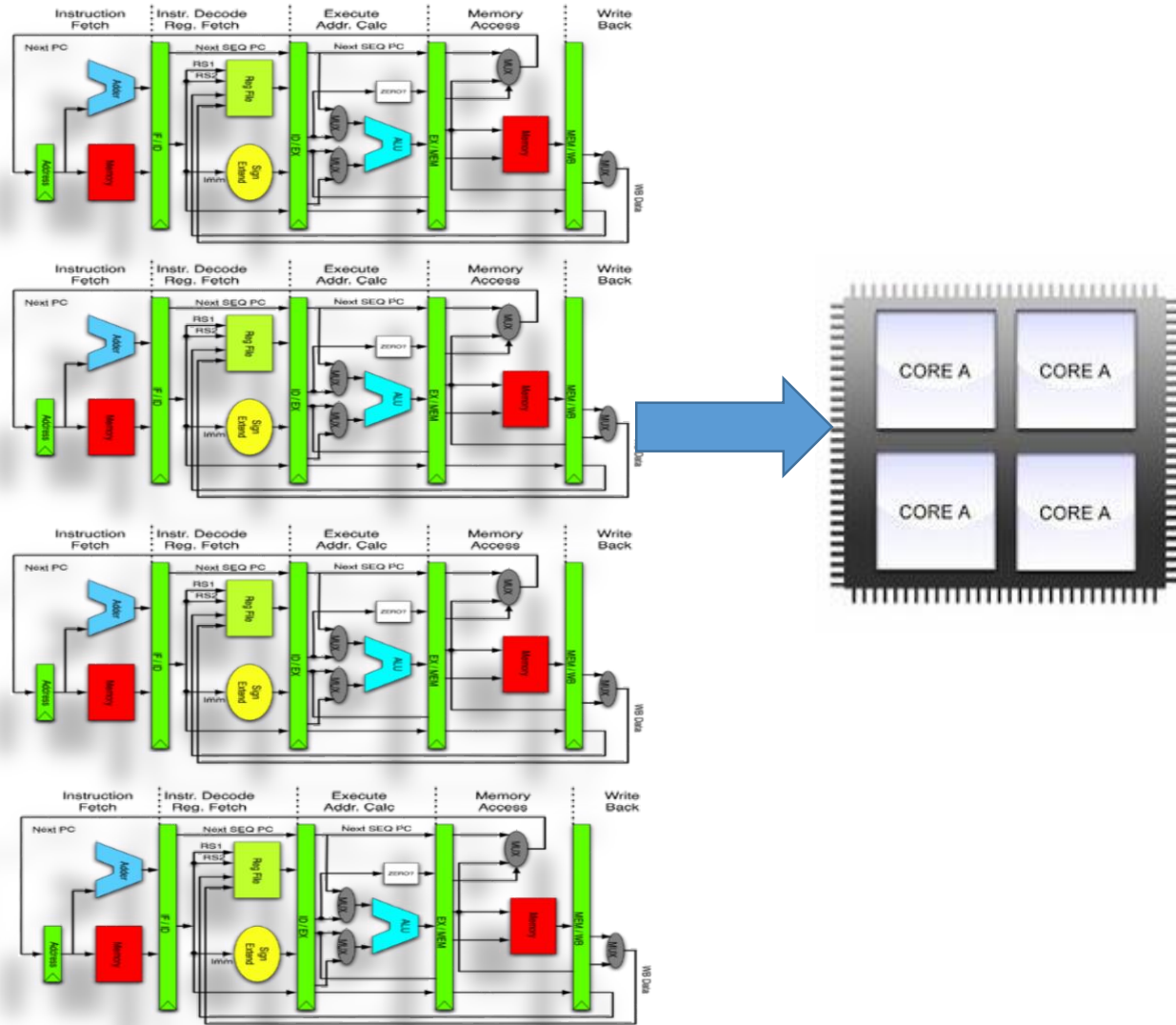


```
main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_instructions());
    }
}

do_instructions() {
    while(true) {
        instruction = fetch();
        ops, regs = decode(instruction);
        execute_calc_addrs(ops, regs);
        access_memory(ops, regs);
        write_back(regs);
    }
}
```

- *Pros: Simple*
- *Cons: programmer has to find the parallelism!*

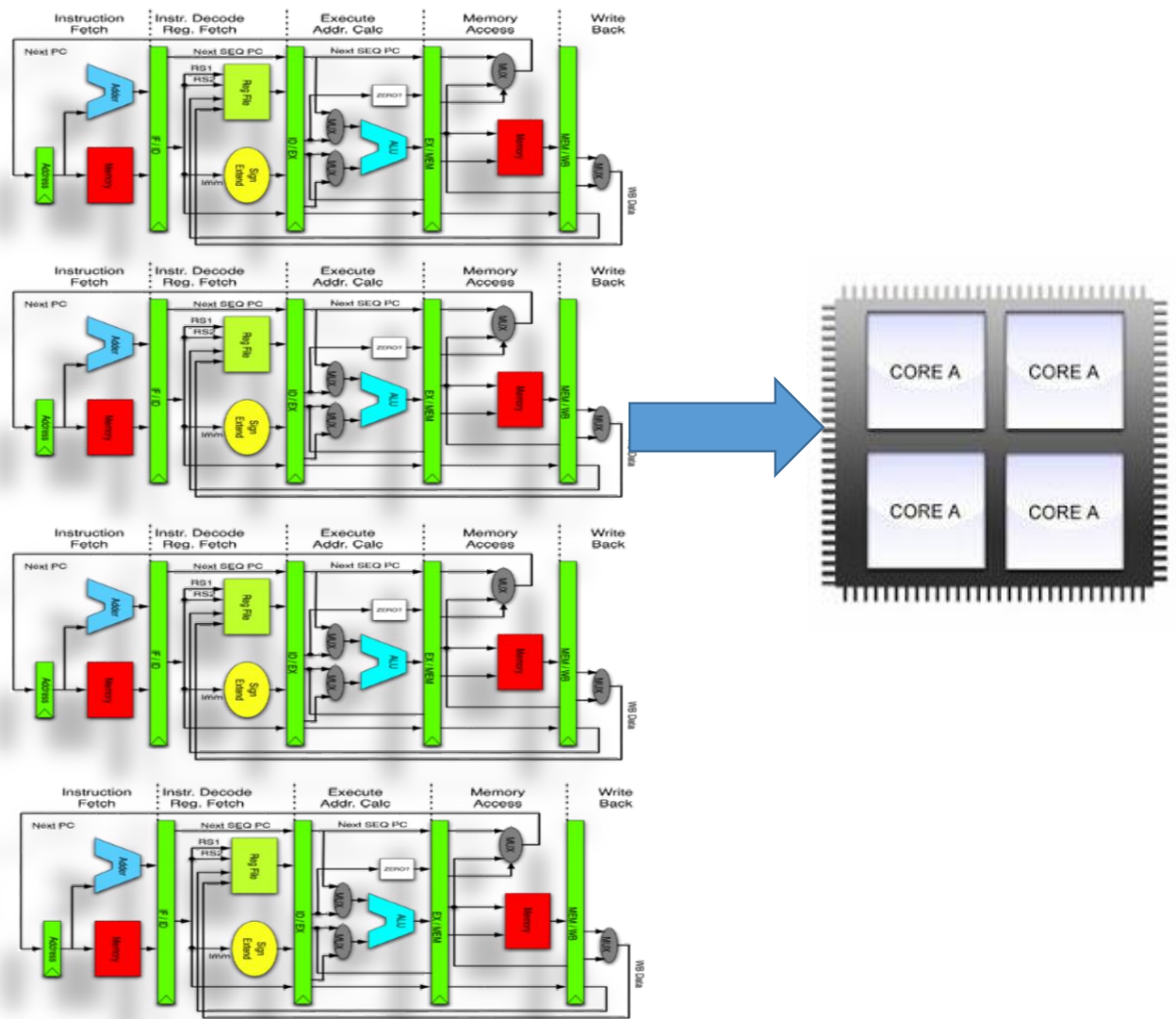
Multi-core/SMPs



```
main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_instructions());
    }
}
do_instructions() {
    while(true) {
        instruction = fetch();
        ops, regs = decode(instruction);
        execute_calc_addrs(ops, regs);
        access_memory(ops, regs);
        write_back(regs);
    }
}
```

- *Pros: Simple*
- *Cons: programmer has to find the parallelism!*

Multi-core/SMPs

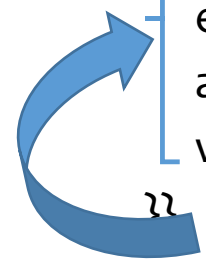


```

main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_instructions());
    }
}

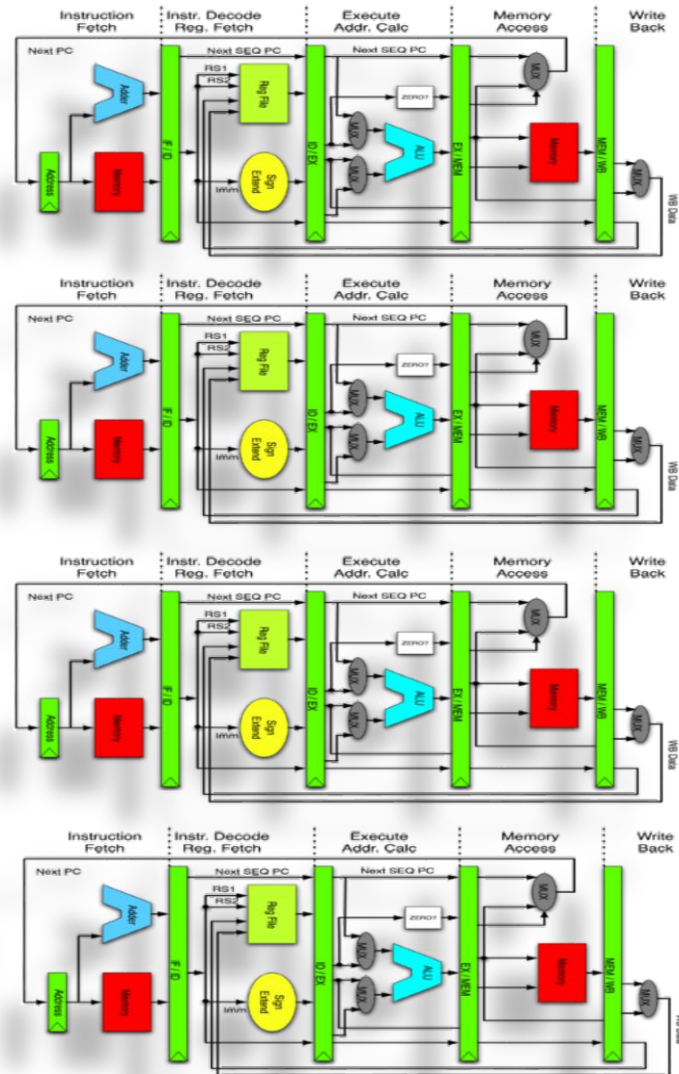
do_instructions() {
    while(true) {
        instruction = fetch();
        ops, regs = decode(instruction);
        execute_calc_addrs(ops, regs);
        access_memory(ops, regs);
        write_back(regs);
    }
}

```



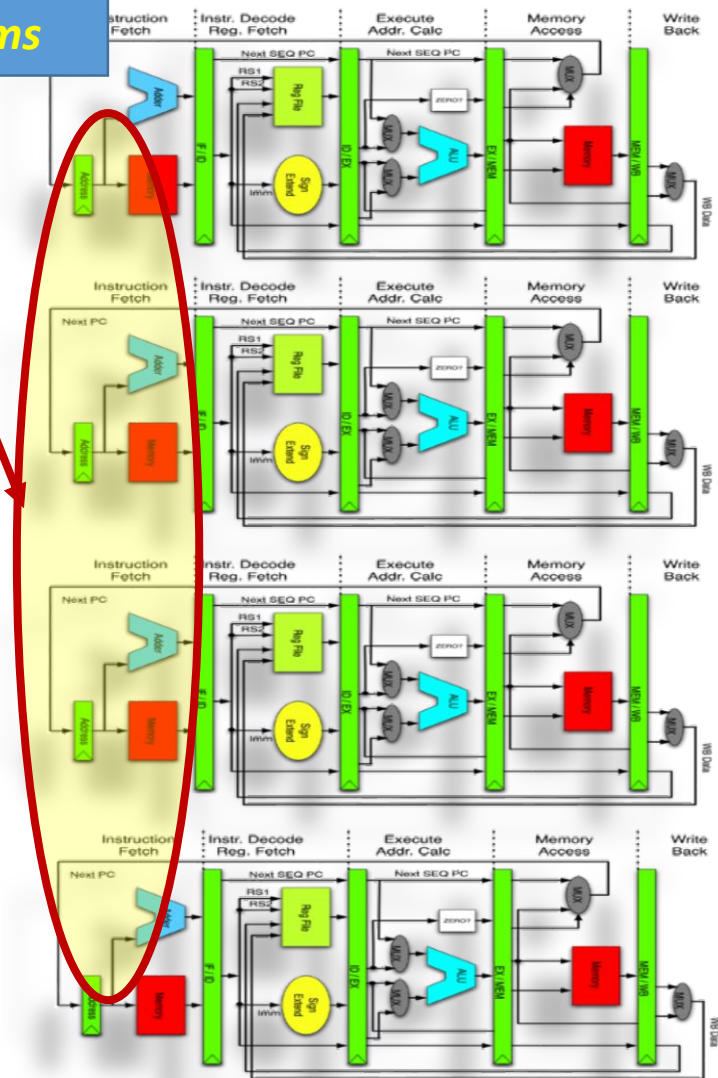
Other techniques extract parallelism here, try to let the machine find parallelism

Superscalar processors



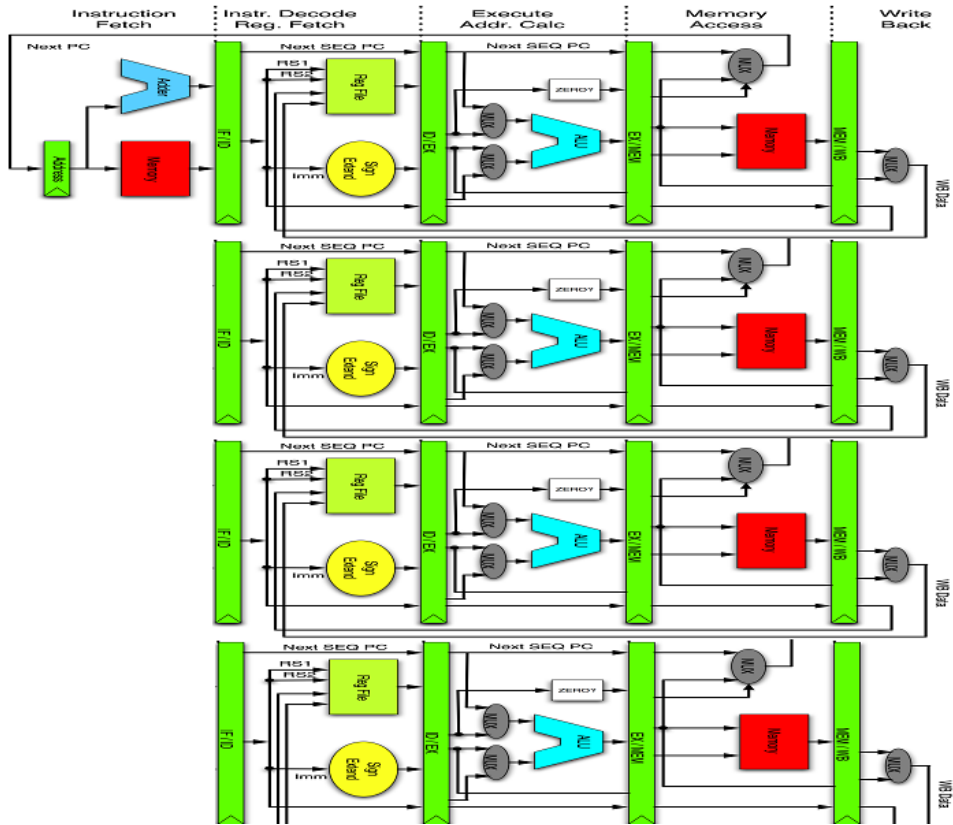
Superscalar processors

Remove extra
instruction streams

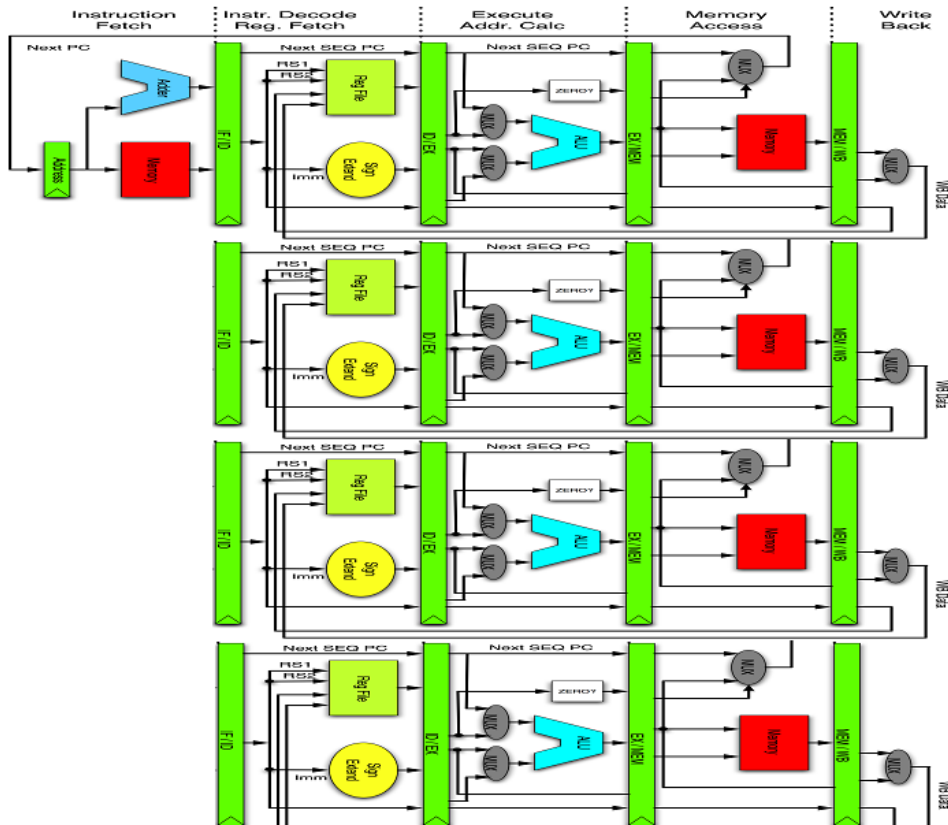


Superscalar processors

Superscalar processors



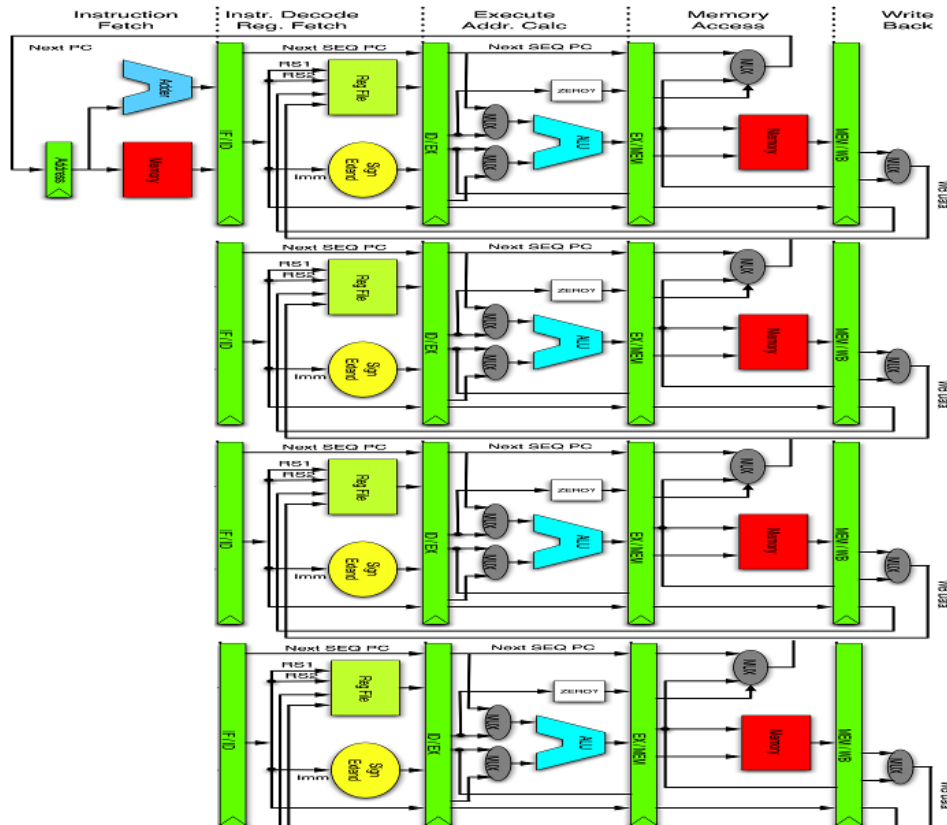
Superscalar processors



```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(decode_exec);  
    while(true) {  
        instruction = fetch();  
        enqueue(instruction);  
    }  
}
```

```
decode_exec() {  
    instruction = dequeue();  
    ops, regs = decode(instruction);  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Superscalar processors

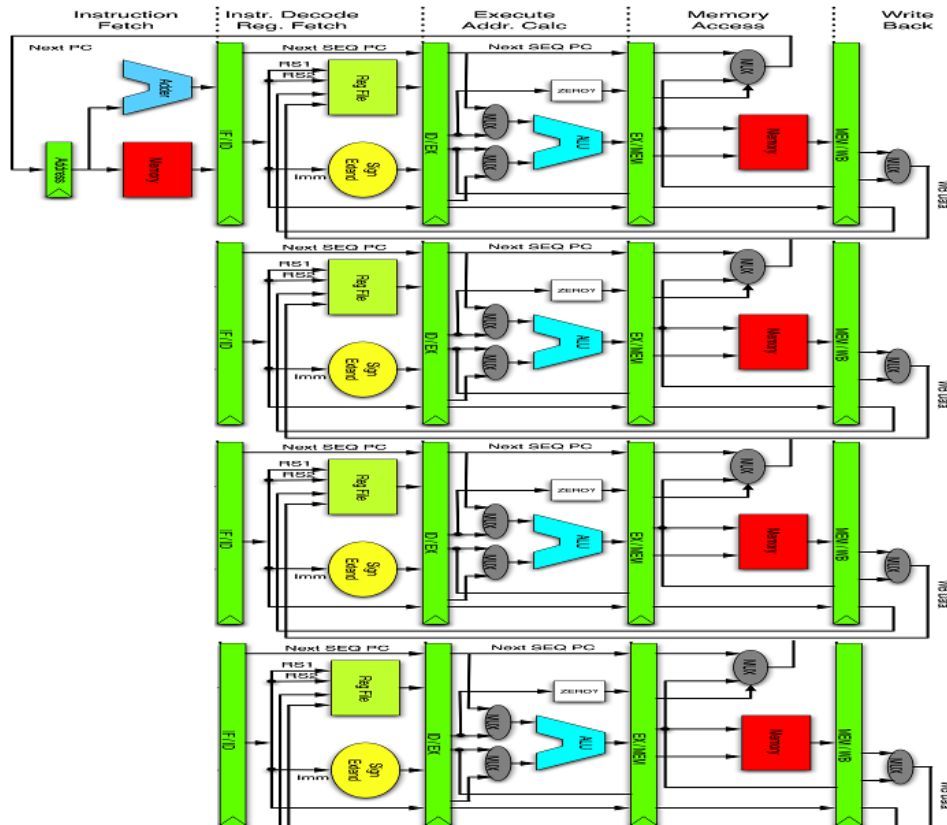


```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(&decode_exec);  
    while(true) {  
        instruction = fetch();  
        enqueue(instruction);  
    }  
}
```

```
decode_exec() {  
    instruction = dequeue();  
    ops, regs = decode(instruction);  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Doesn't look that different does it? Why do it?

Superscalar processors



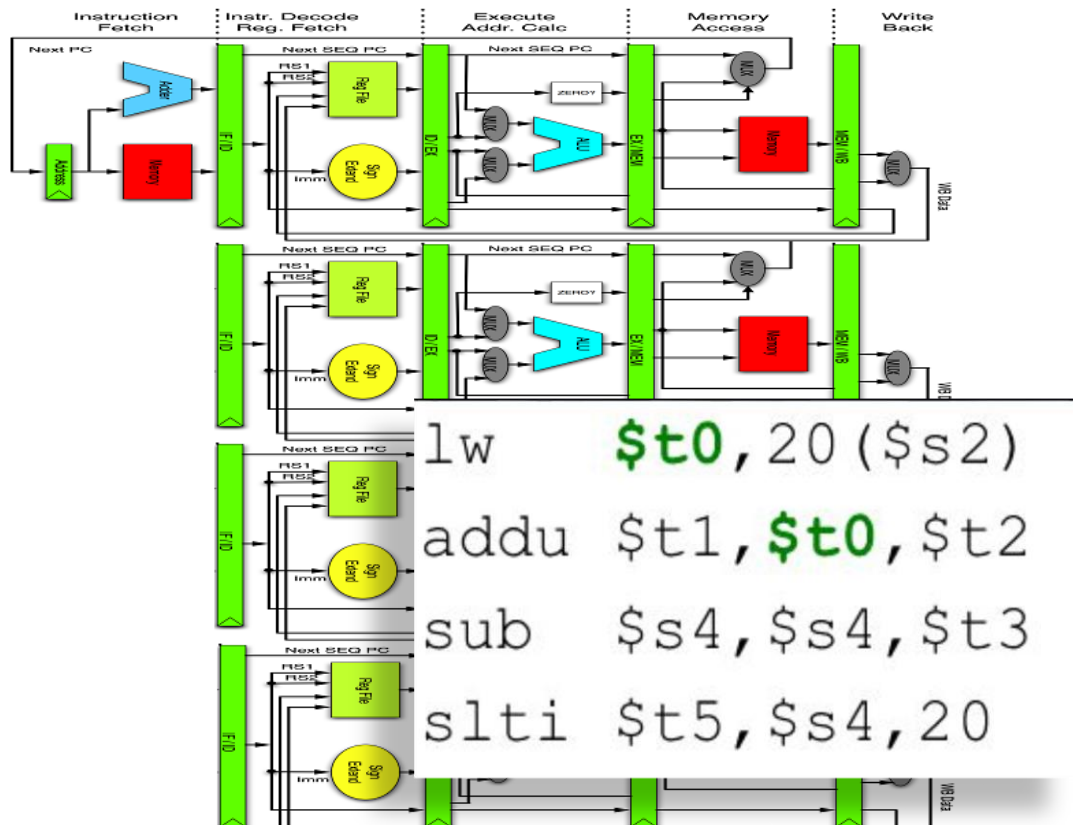
```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(&decode_exec);  
    while(true) {  
        instruction = fetch();  
        enqueue(instruction);  
    }  
}
```

```
decode_exec() {  
    instruction = dequeue();  
    ops, regs = decode(instruction);  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Doesn't look that different does it? Why do it?

Enables independent instruction parallelism.

Superscalar processors



```

main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
  while(true) {
    instruction = fetch();
    enqueue(instruction);
  }
}
    
```

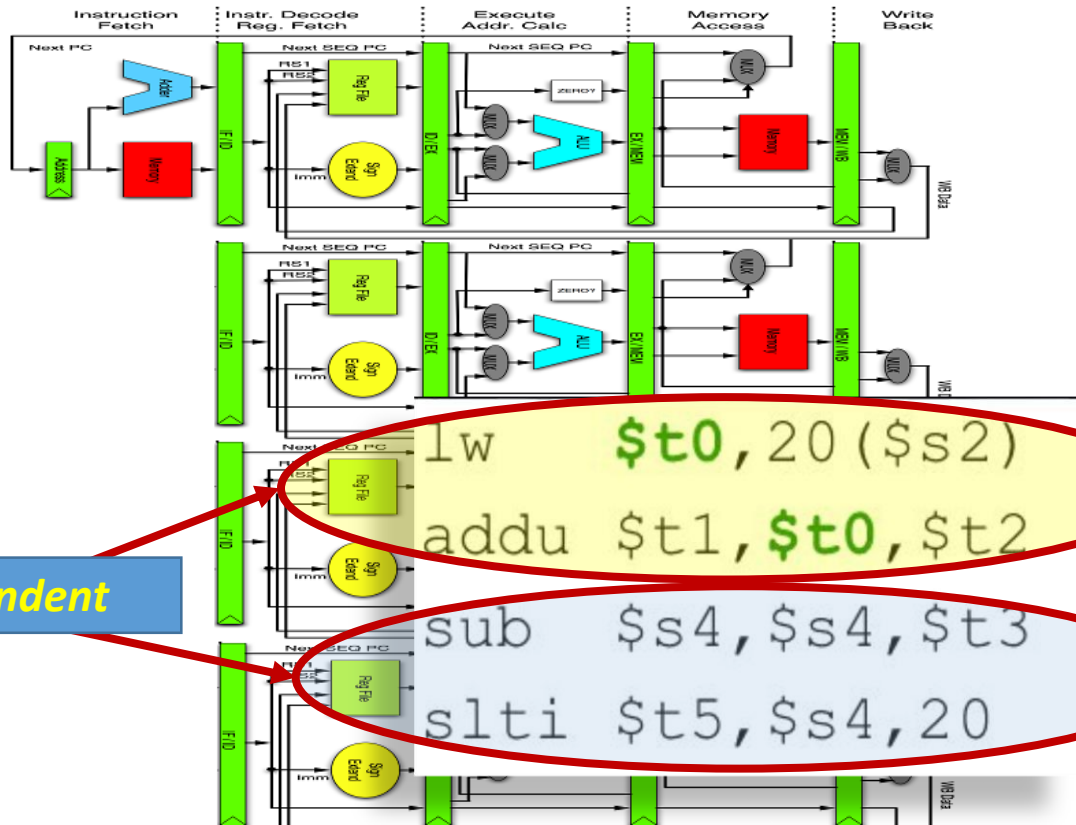
```

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
    
```

Doesn't look that different does it? Why do it?

Enables independent instruction parallelism.

Superscalar processors



```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(&decode_exec);  
    while(true) {  
        instruction = fetch();  
        enqueue(instruction);  
    }  
}
```

```
decode_exec() {  
    instruction = dequeue();  
    ops, regs = decode(instruction);  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Doesn't look that different does it? Why do it?

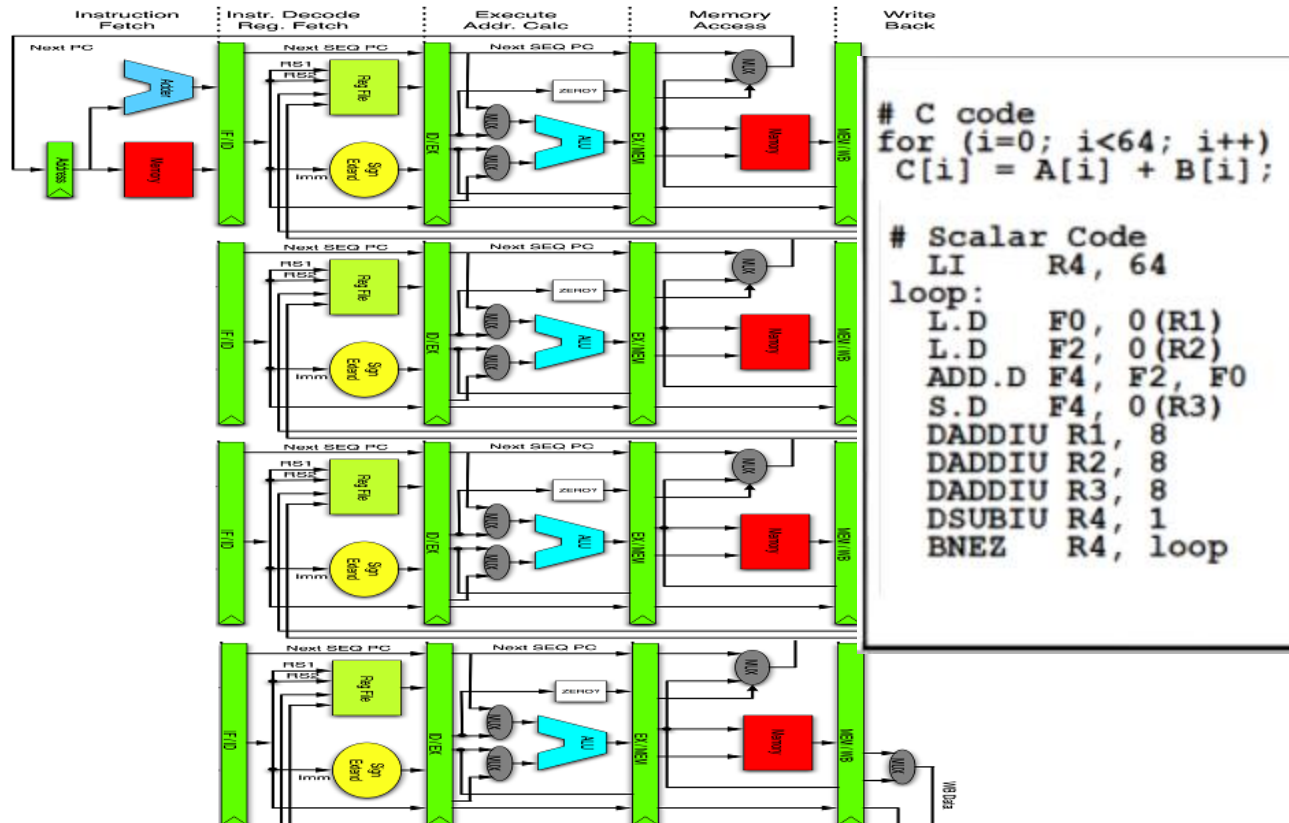
Enables independent instruction parallelism.

Vector/SIMD processors

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];

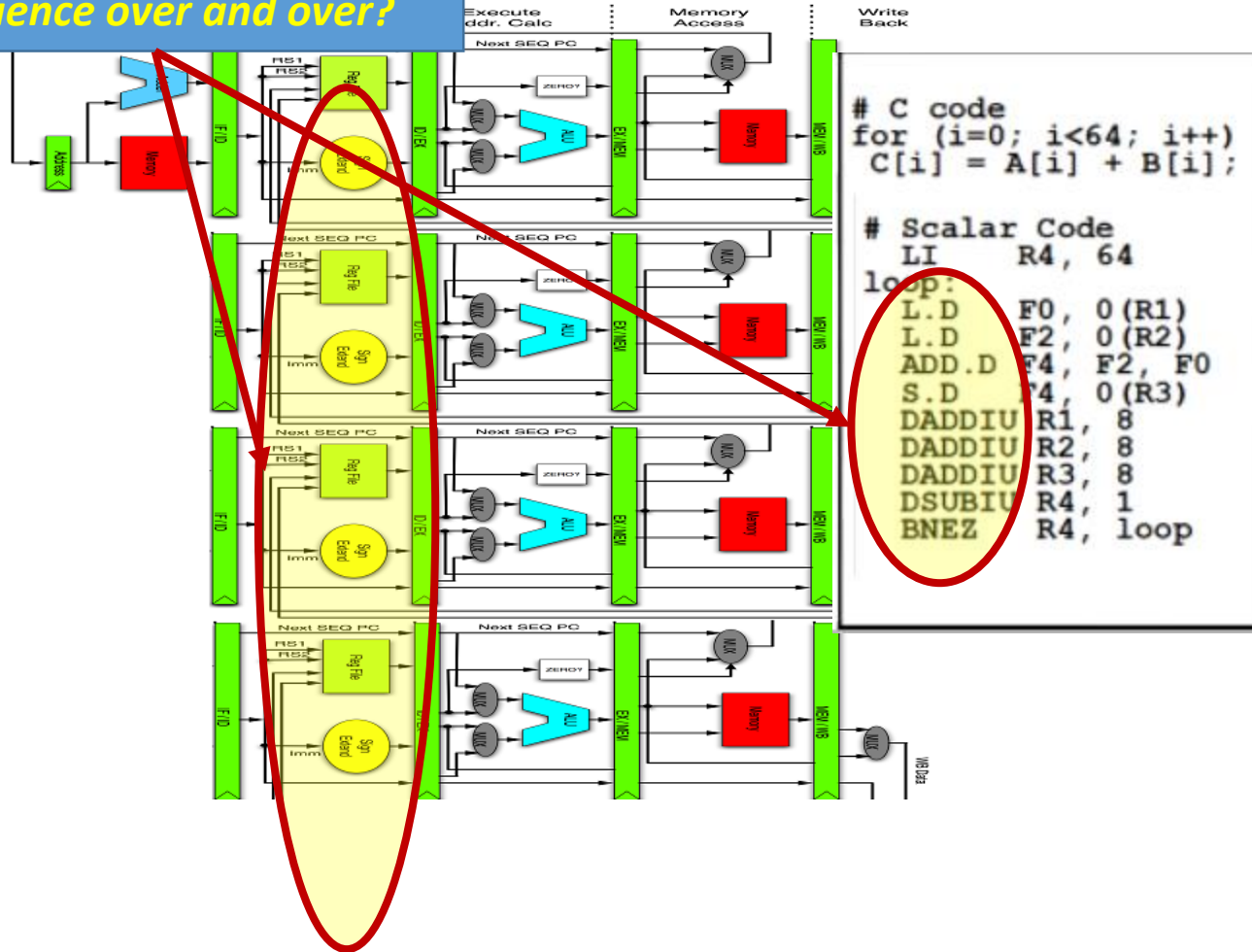
# Scalar Code
LI      R4, 64
loop:
  L.D   F0, 0(R1)
  L.D   F2, 0(R2)
  ADD.D F4, F2, F0
  S.D   F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ  R4, loop
```

Vector/SIMD processors

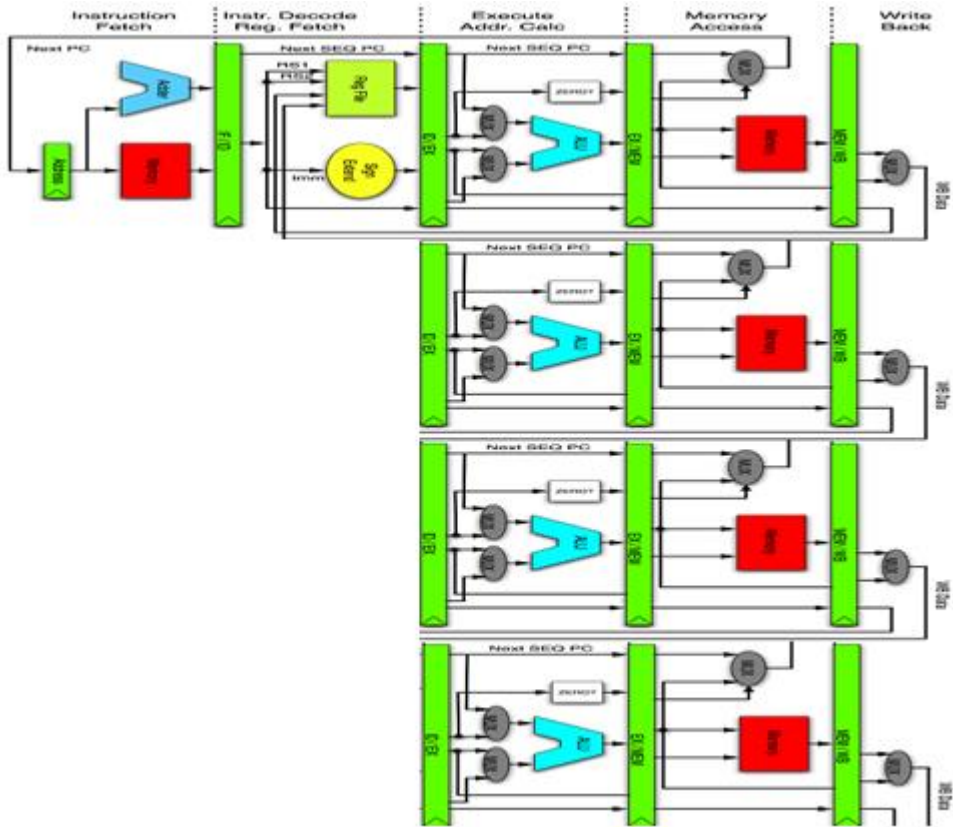


Vector/SIMD processors

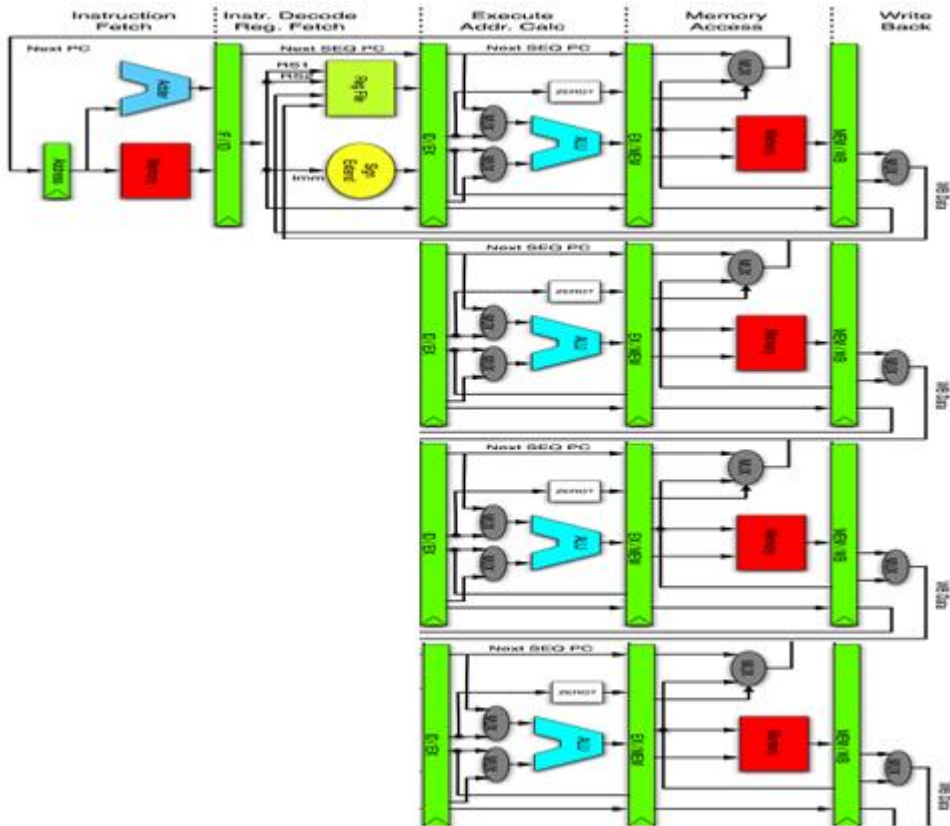
Why decode same instruction sequence over and over?



Vector/SIMD processors



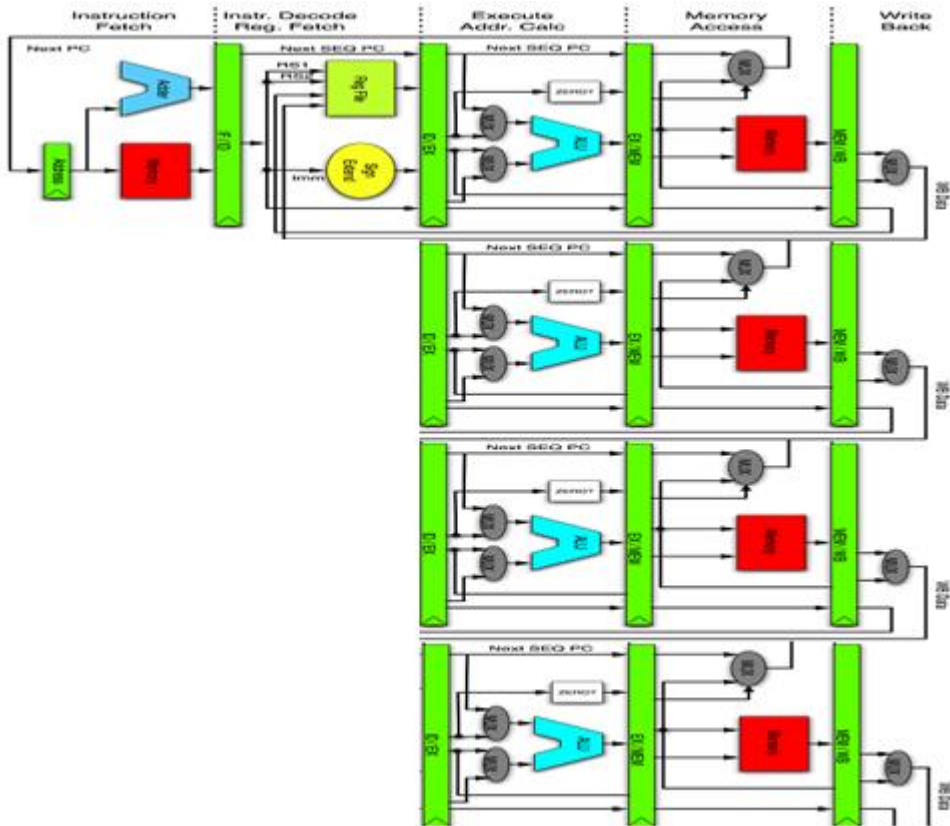
Vector/SIMD processors



```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(exec);  
    while(true) {  
        ops, regs = fetch_decode();  
        enqueue(ops, regs);  
    }  
}
```

```
exec() {  
    ops, regs = dequeue();  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Vector/SIMD processors

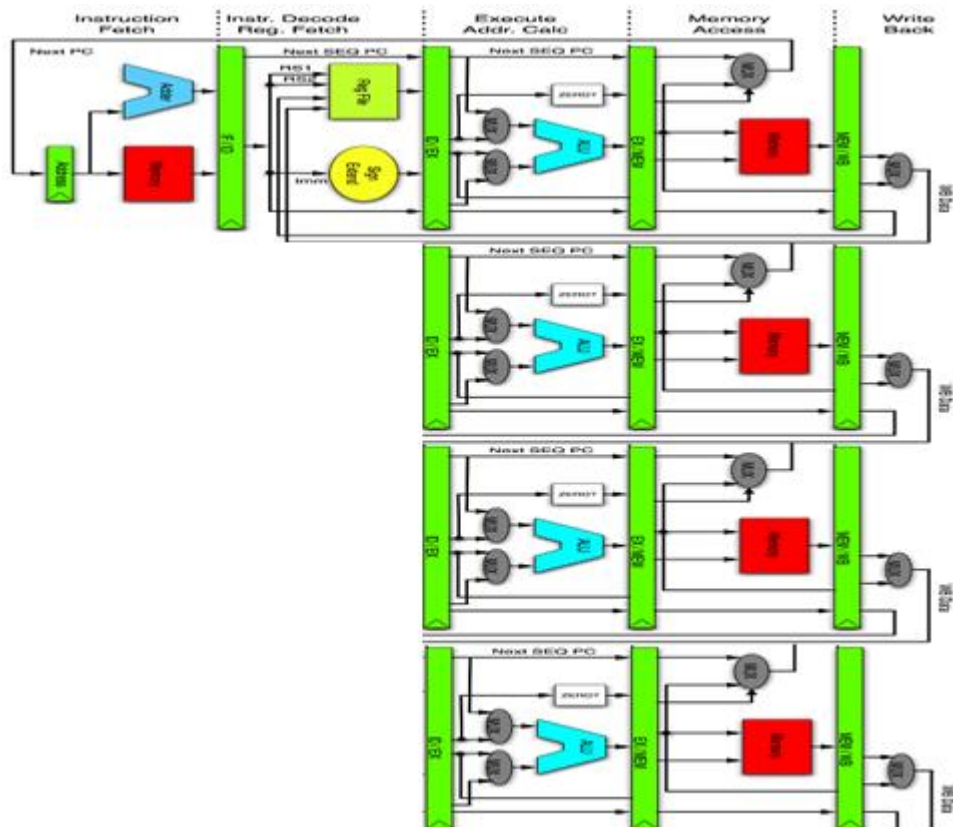


```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(exec);  
    while(true) {  
        ops, regs = fetch_decode();  
        enqueue(ops, regs);  
    }  
}
```

```
exec() {  
    ops, regs = dequeue();  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Single instruction stream, multiple computations

Vector/SIMD processors



```
main() {  
    for(i=0; i<CORES; i++)  
        pthread_create(exec);  
    while(true) {  
        ops, regs = fetch_decode();  
        enqueue(ops, regs);  
    }  
}
```

```
exec() {  
    ops, regs = dequeue();  
    execute_calc_addrs(ops, regs);  
    access_memory(ops, regs);  
    write_back(regs);  
}
```

Single instruction stream, multiple computations
But now all my instructions need multiple operands!

Vector Processors

- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970's 80's and 90's.
- Modern CPUs support some vector processing instructions
 - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
 - SIMD operate on all per clock cycle

Vector Processors

- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970's 80's and 90's.
- Modern CPUs support some vector processing instructions
 - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
 - SIMD operate on all per clock cycle
- 1962 University of Illinois Illiac IV - completed 1972 → 64 ALUs 100-150 MFlops
- (1973) TI's Advance Scientific Computer (ASC) 20-80 MFlops
- (1975) Cray-1 first to have vector registers instead of keeping data in memory



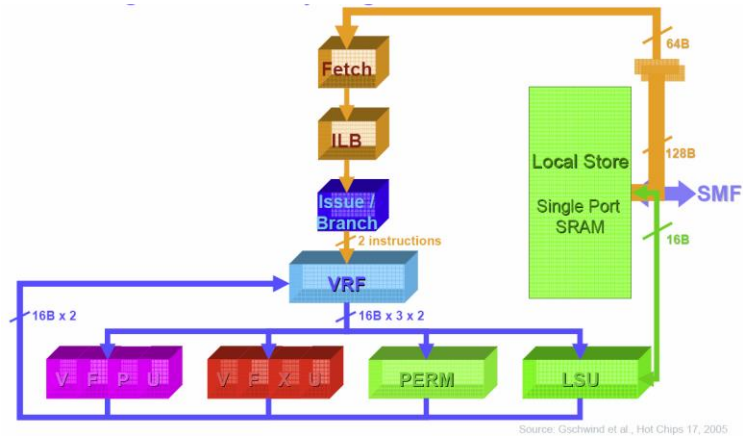
Vector Processors

- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970's 80's and 90's.
- Modern CPUs support some vector processing instructions
 - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
 - SIMD operate on all per clock cycle
- 1962 University of Illinois Illiac IV - completed 1972 → 64 ALUs 100-150 MFlops
- (1973) TI's Advance Scientific Computer (ASC) 20-80 MFlops
- (1975) Cray-1 first to have vector registers instead of keeping data in memory



*Single instruction stream, multiple data →
Programming model has to change*

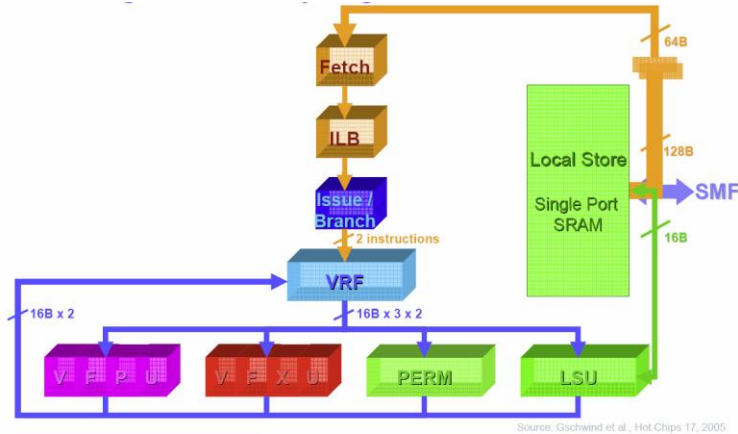
Vector Processors



Implementation:

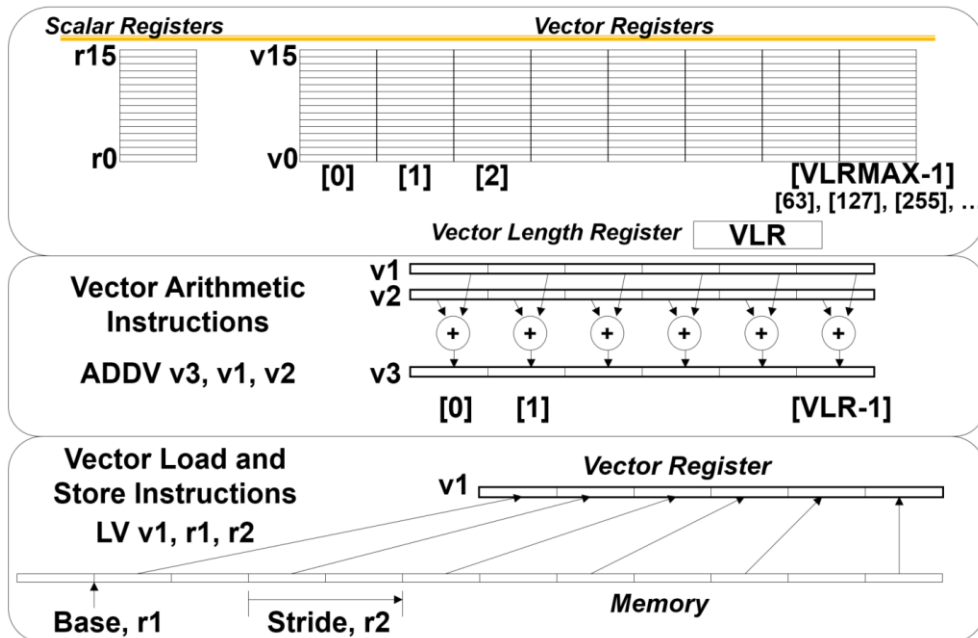
- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel

Vector Processors

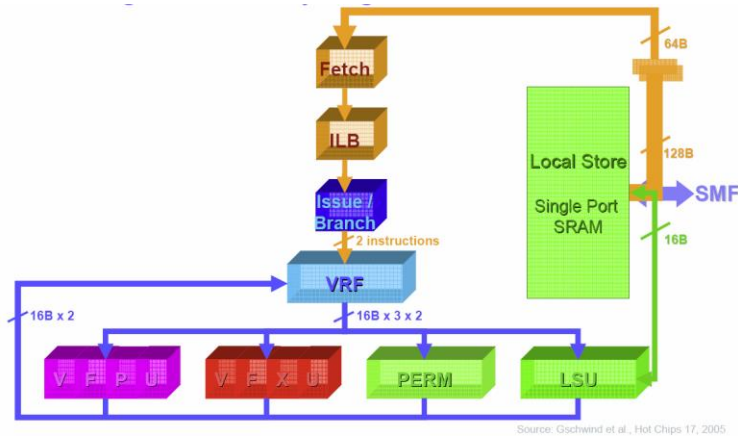


Implementation:

- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel

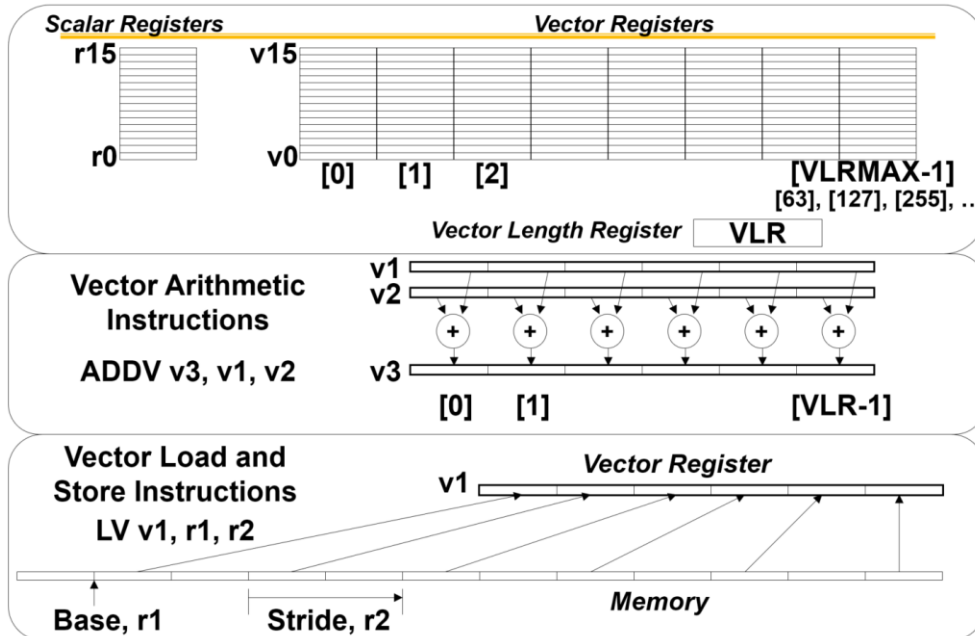


Vector Processors



Implementation:

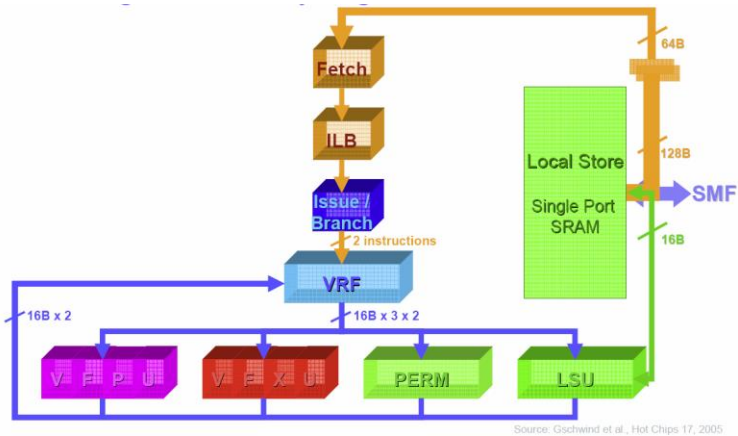
- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel



<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre># Scalar Code LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre># Vector Code LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>
---	---	---

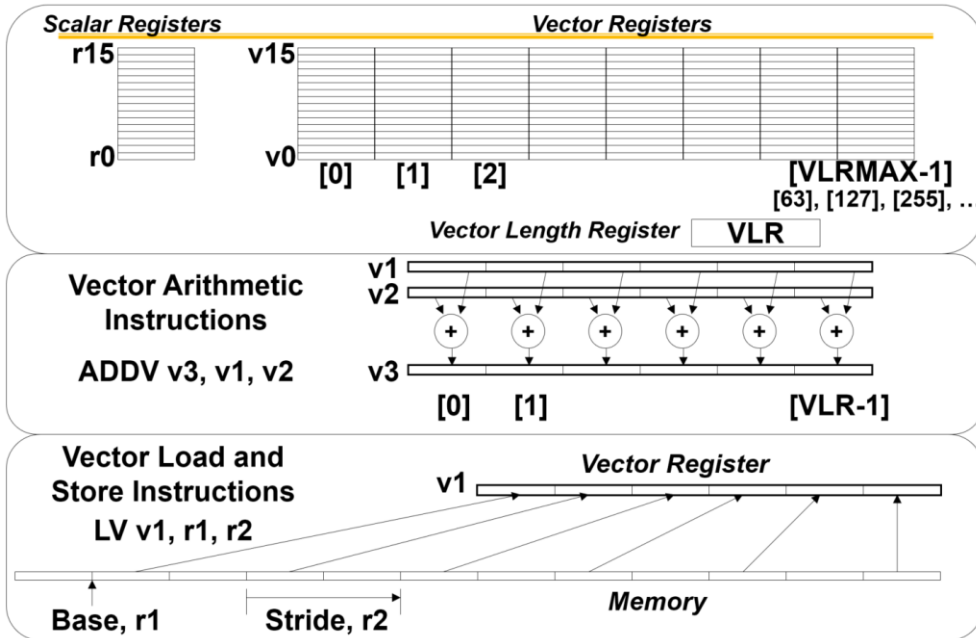
Vector Processors

GPUs: same basic idea



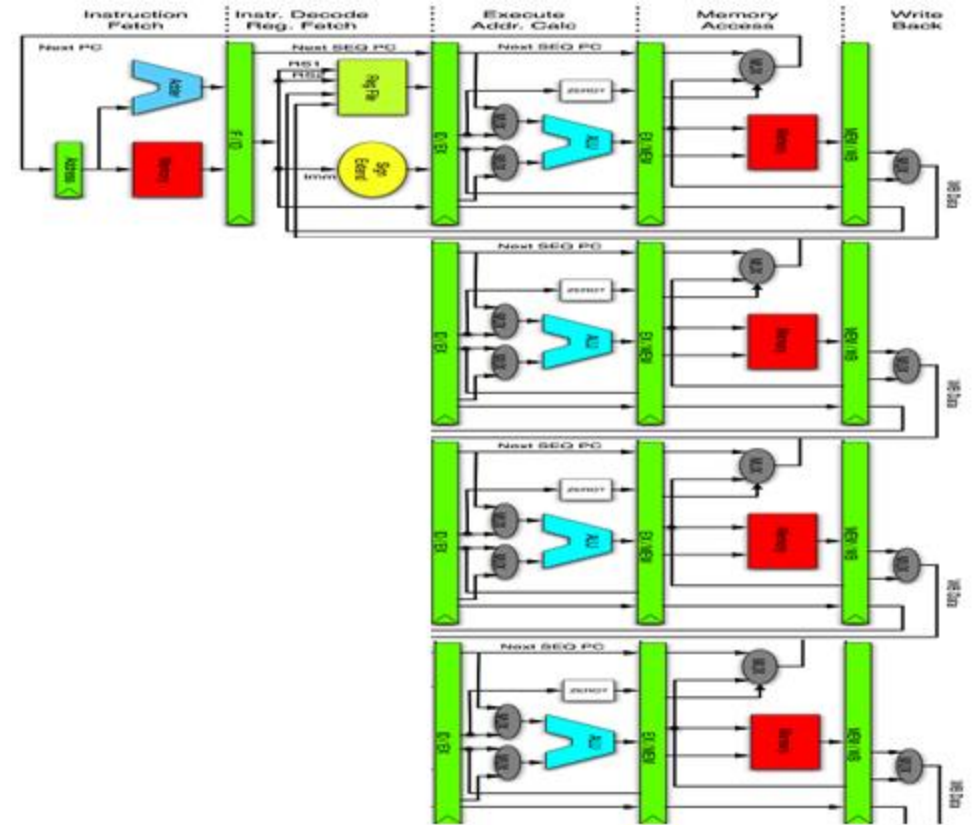
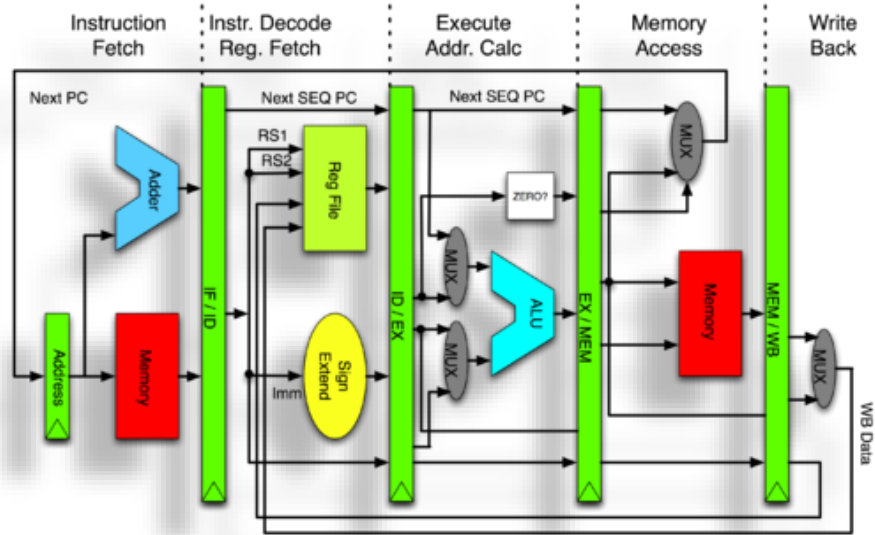
Implementation:

- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel

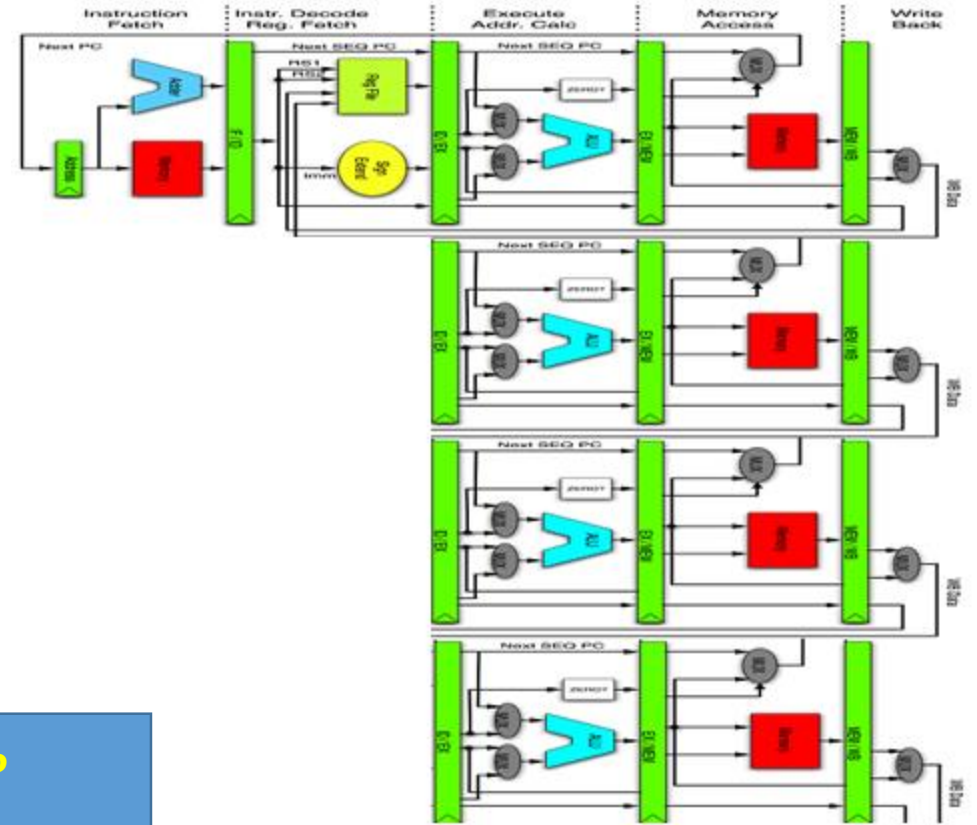
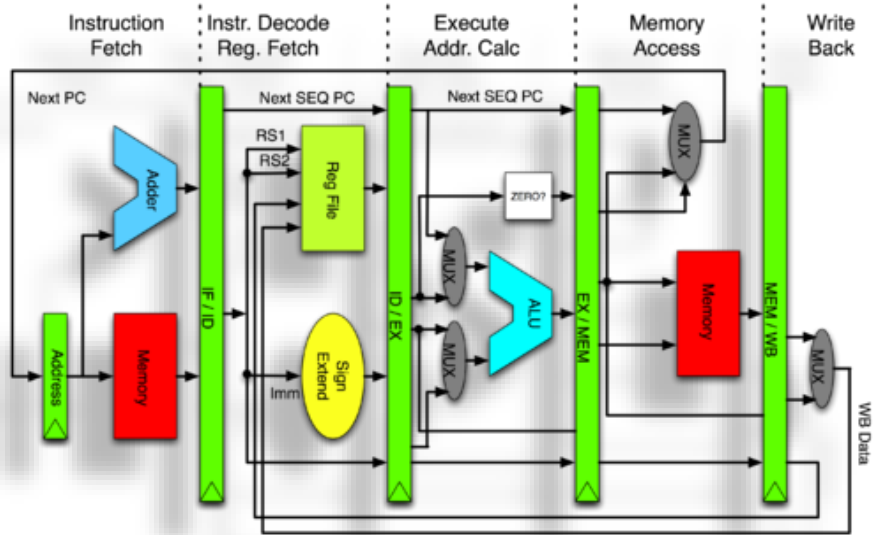


<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre># Scalar Code LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre># Vector Code LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>
---	--	--

When does vector processing help?

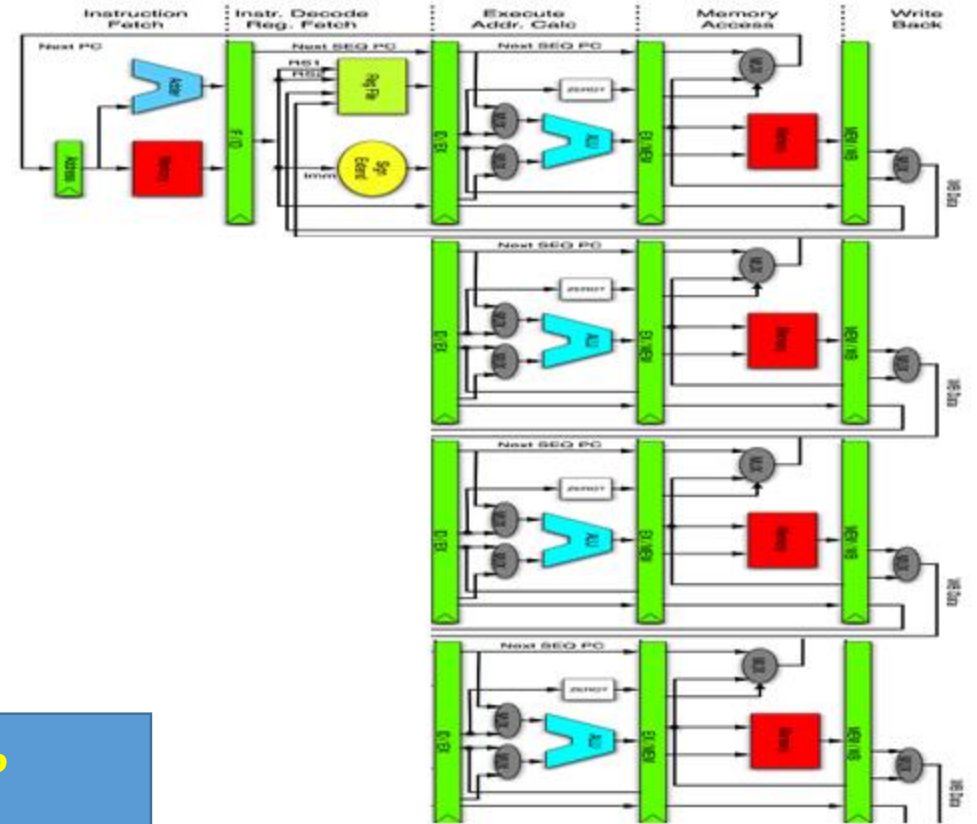
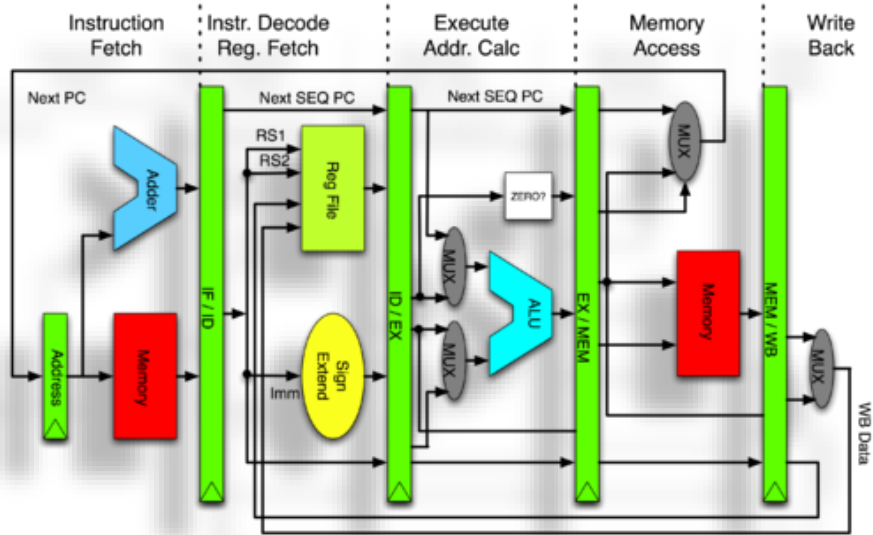


When does vector processing help?



*What are the potential bottlenecks here?
When can it improve throughput?*

When does vector processing help?



*What are the potential bottlenecks here?
When can it improve throughput?*

Only helps if memory can keep the pipeline busy!

Hardware multi-threading

Hardware multi-threading

- Address memory bottleneck

Hardware multi-threading

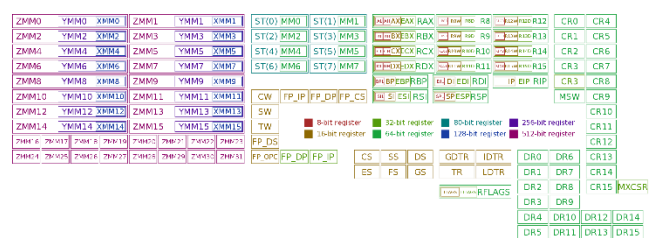
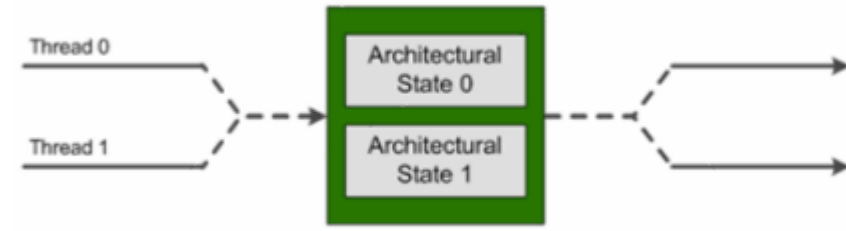
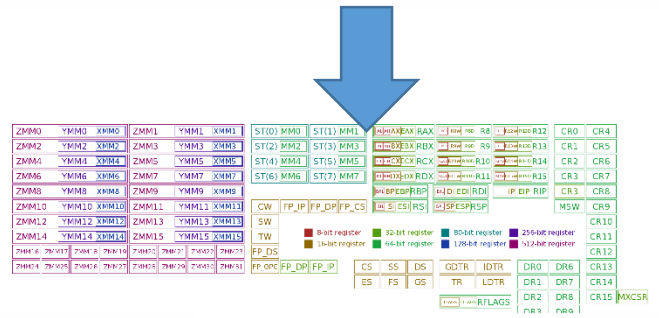
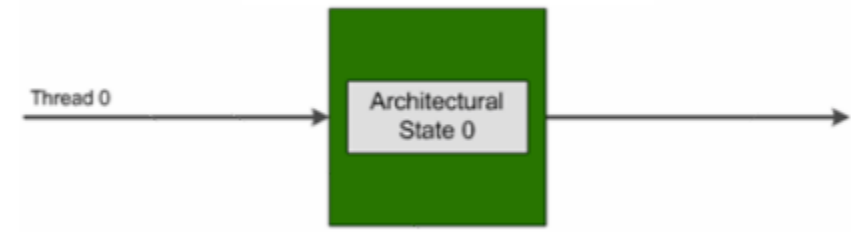
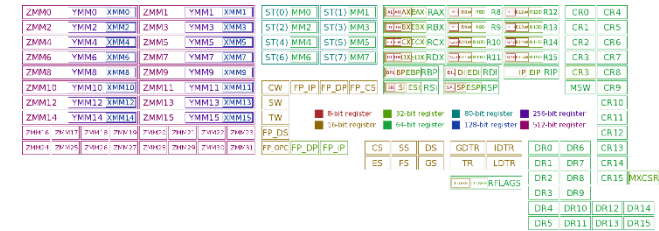
- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls

Hardware multi-threading

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	CR0	CR2	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR1	CR5	
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR4	CR8	
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR5	CR9	
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR6	CR10	
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR7	CR11	
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR8	CR12	
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR9	CR13	
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR10	CR14	
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR11	CR15	
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR12	CR16	
ZMM26	YMM26	XMM26	ZMM27	YMM27	XMM27			CR13	CR17	
ZMM28	YMM28	XMM28	ZMM29	YMM29	XMM29			CR14	CR18	
ZMM30	YMM30	XMM30	ZMM31	YMM31	XMM31			CR15	CR19	
ZMM32	YMM32	XMM32	ZMM33	YMM33	XMM33			CR16	CR20	
ZMM34	YMM34	XMM34	ZMM35	YMM35	XMM35			CR17	CR21	
ZMM36	YMM36	XMM36	ZMM37	YMM37	XMM37			CR18	CR22	
ZMM38	YMM38	XMM38	ZMM39	YMM39	XMM39			CR19	CR23	
ZMM40	YMM40	XMM40	ZMM41	YMM41	XMM41			CR20	CR24	
ZMM42	YMM42	XMM42	ZMM43	YMM43	XMM43			CR21	CR25	
ZMM44	YMM44	XMM44	ZMM45	YMM45	XMM45			CR22	CR26	
ZMM46	YMM46	XMM46	ZMM47	YMM47	XMM47			CR23	CR27	
ZMM48	YMM48	XMM48	ZMM49	YMM49	XMM49			CR24	CR28	
ZMM50	YMM50	XMM50	ZMM51	YMM51	XMM51			CR25	CR29	
ZMM52	YMM52	XMM52	ZMM53	YMM53	XMM53			CR26	CR30	
ZMM54	YMM54	XMM54	ZMM55	YMM55	XMM55			CR27	CR31	
ZMM56	YMM56	XMM56	ZMM57	YMM57	XMM57			CR28	CR32	
ZMM58	YMM58	XMM58	ZMM59	YMM59	XMM59			CR29	CR33	
ZMM60	YMM60	XMM60	ZMM61	YMM61	XMM61			CR30	CR34	
ZMM62	YMM62	XMM62	ZMM63	YMM63	XMM63			CR31	CR35	
ZMM64	YMM64	XMM64	ZMM65	YMM65	XMM65			CR32	CR36	
ZMM66	YMM66	XMM66	ZMM67	YMM67	XMM67			CR33	CR37	
ZMM68	YMM68	XMM68	ZMM69	YMM69	XMM69			CR34	CR38	
ZMM70	YMM70	XMM70	ZMM71	YMM71	XMM71			CR35	CR39	
ZMM72	YMM72	XMM72	ZMM73	YMM73	XMM73			CR36	CR40	
ZMM74	YMM74	XMM74	ZMM75	YMM75	XMM75			CR37	CR41	
ZMM76	YMM76	XMM76	ZMM77	YMM77	XMM77			CR38	CR42	
ZMM78	YMM78	XMM78	ZMM79	YMM79	XMM79			CR39	CR43	
ZMM80	YMM80	XMM80	ZMM81	YMM81	XMM81			CR40	CR44	
ZMM82	YMM82	XMM82	ZMM83	YMM83	XMM83			CR41	CR45	
ZMM84	YMM84	XMM84	ZMM85	YMM85	XMM85			CR42	CR46	
ZMM86	YMM86	XMM86	ZMM87	YMM87	XMM87			CR43	CR47	
ZMM88	YMM88	XMM88	ZMM89	YMM89	XMM89			CR44	CR48	
ZMM90	YMM90	XMM90	ZMM91	YMM91	XMM91			CR45	CR49	
ZMM92	YMM92	XMM92	ZMM93	YMM93	XMM93			CR46	CR50	
ZMM94	YMM94	XMM94	ZMM95	YMM95	XMM95			CR47	CR51	
ZMM96	YMM96	XMM96	ZMM97	YMM97	XMM97			CR48	CR52	
ZMM98	YMM98	XMM98	ZMM99	YMM99	XMM99			CR49	CR53	
ZMM100	YMM100	XMM100	ZMM101	YMM101	XMM101			CR50	CR54	
ZMM102	YMM102	XMM102	ZMM103	YMM103	XMM103			CR51	CR55	
ZMM104	YMM104	XMM104	ZMM105	YMM105	XMM105			CR52	CR56	
ZMM106	YMM106	XMM106	ZMM107	YMM107	XMM107			CR53	CR57	
ZMM108	YMM108	XMM108	ZMM109	YMM109	XMM109			CR54	CR58	
ZMM110	YMM110	XMM110	ZMM111	YMM111	XMM111			CR55	CR59	
ZMM112	YMM112	XMM112	ZMM113	YMM113	XMM113			CR56	CR60	
ZMM114	YMM114	XMM114	ZMM115	YMM115	XMM115			CR57	CR61	
ZMM116	YMM116	XMM116	ZMM117	YMM117	XMM117			CR58	CR62	
ZMM118	YMM118	XMM118	ZMM119	YMM119	XMM119			CR59	CR63	
ZMM120	YMM120	XMM120	ZMM121	YMM121	XMM121			CR60	CR64	
ZMM122	YMM122	XMM122	ZMM123	YMM123	XMM123			CR61	CR65	
ZMM124	YMM124	XMM124	ZMM125	YMM125	XMM125			CR62	CR66	
ZMM126	YMM126	XMM126	ZMM127	YMM127	XMM127			CR63	CR67	
ZMM128	YMM128	XMM128	ZMM129	YMM129	XMM129			CR64	CR68	
ZMM130	YMM130	XMM130	ZMM131	YMM131	XMM131			CR65	CR69	
ZMM132	YMM132	XMM132	ZMM133	YMM133	XMM133			CR66	CR70	
ZMM134	YMM134	XMM134	ZMM135	YMM135	XMM135			CR67	CR71	
ZMM136	YMM136	XMM136	ZMM137	YMM137	XMM137			CR68	CR72	
ZMM138	YMM138	XMM138	ZMM139	YMM139	XMM139			CR69	CR73	
ZMM140	YMM140	XMM140	ZMM141	YMM141	XMM141			CR70	CR74	
ZMM142	YMM142	XMM142	ZMM143	YMM143	XMM143			CR71	CR75	
ZMM144	YMM144	XMM144	ZMM145	YMM145	XMM145			CR72	CR76	
ZMM146	YMM146	XMM146	ZMM147	YMM147	XMM147			CR73	CR77	
ZMM148	YMM148	XMM148	ZMM149	YMM149	XMM149			CR74	CR78	
ZMM150	YMM150	XMM150	ZMM151	YMM151	XMM151			CR75	CR79	
ZMM152	YMM152	XMM152	ZMM153	YMM153	XMM153			CR76	CR80	
ZMM154	YMM154	XMM154	ZMM155	YMM155	XMM155			CR77	CR81	
ZMM156	YMM156	XMM156	ZMM157	YMM157	XMM157			CR78	CR82	
ZMM158	YMM158	XMM158	ZMM159	YMM159	XMM159			CR79	CR83	
ZMM160	YMM160	XMM160	ZMM161	YMM161	XMM161			CR80	CR84	
ZMM162	YMM162	XMM162	ZMM163	YMM163	XMM163			CR81	CR85	
ZMM164	YMM164	XMM164	ZMM165	YMM165	XMM165			CR82	CR86	
ZMM166	YMM166	XMM166	ZMM167	YMM167	XMM167			CR83	CR87	
ZMM168	YMM168	XMM168	ZMM169	YMM169	XMM169			CR84	CR88	
ZMM170	YMM170	XMM170	ZMM171	YMM171	XMM171			CR85	CR89	
ZMM172	YMM172	XMM172	ZMM173	YMM173	XMM173			CR86	CR90	
ZMM174	YMM174	XMM174	ZMM175	YMM175	XMM175			CR87	CR91	
ZMM176	YMM176	XMM176	ZMM177	YMM177	XMM177			CR88	CR92	
ZMM178	YMM178	XMM178	ZMM179	YMM179	XMM179			CR89	CR93	
ZMM180	YMM180	XMM180	ZMM181	YMM181	XMM181			CR90	CR94	
ZMM182	YMM182	XMM182	ZMM183	YMM183	XMM183			CR91	CR95	
ZMM184	YMM184	XMM184	ZMM185	YMM185	XMM185			CR92	CR96	
ZMM186	YMM186	XMM186	ZMM187	YMM187	XMM187			CR93	CR97	
ZMM188	YMM188	XMM188	ZMM189	YMM189	XMM189			CR94	CR98	
ZMM190	YMM190	XMM190	ZMM191	YMM191	XMM191			CR95	CR99	
ZMM192	YMM192	XMM192	ZMM193	YMM193	XMM193			CR96	CR100	
ZMM194	YMM194	XMM194	ZMM195	YMM195	XMM195			CR97	CR101	
ZMM196	YMM196	XMM196	ZMM197	YMM197	XMM197			CR98	CR102	
ZMM198	YMM198	XMM198	ZMM199	YMM199	XMM199			CR99	CR103	
ZMM200	YMM200	XMM200	ZMM201	YMM201	XMM201			CR100	CR104	
ZMM202	YMM202	XMM202	ZMM203	YMM203	XMM203			CR101	CR105	
ZMM204	YMM204	XMM204	ZMM205	YMM205	XMM205			CR102	CR106	
ZMM206	YMM206	XMM206	ZMM207	YMM207	XMM207			CR103	CR107	
ZMM208	YMM208	XMM208	ZMM209	YMM209	XMM209			CR104	CR108	
ZMM210	YMM210	XMM210	ZMM211	YMM211	XMM211			CR105	CR109	
ZMM212	YMM212	XMM212	ZMM213	YMM213	XMM213			CR106	CR110	
ZMM214	YMM214	XMM214	ZMM215	YMM215	XMM215			CR107	CR111	
ZMM216	YMM216	XMM216	ZMM217	YMM217	XMM217			CR108	CR112	
ZMM218	YMM218	XMM218	ZMM219	YMM219	XMM219			CR109	CR113	
ZMM220	YMM220	XMM220	ZMM221	YMM221	XMM221			CR110	CR114	
ZMM222	YMM222	XMM222	ZMM223	YMM223	XMM223			CR111	CR115	
ZMM224	YMM224	XMM224	ZMM225	YMM225	XMM225			CR112	CR116	
ZMM226	YMM226	XMM226	ZMM227	YMM227	XMM227			CR113	CR117	
ZMM228	YMM228	XMM228	ZMM229	YMM229	XMM229			CR114	CR118	
ZMM230	YMM230	XMM230	ZMM231	YMM231	XMM231			CR115	CR119	
ZMM232	YMM232	XMM232	ZMM233	YMM233	XMM233			CR116	CR120	
ZMM234	YMM234	XMM234	ZMM235	YMM235	XMM235			CR117	CR121	
ZMM236	YMM236	XMM236	ZMM237	YMM237	XMM237			CR118	CR122	
ZMM238	YMM238	XMM238	ZMM239	YMM239	XMM239			CR119	CR123	
ZMM240	YMM240	XMM240	ZMM241	YMM241	XMM241			CR120	CR124	
ZMM242	YMM242	XMM242	ZMM243	YMM243	XMM243			CR121	CR125	
ZMM244	YMM244	XMM244	ZMM245	YMM245	XMM245			CR122	CR126	
ZMM246	YMM246	XMM246	ZMM247	YMM247	XMM247			CR123	CR127	
ZMM248	YMM248	XMM248	ZMM249	YMM249	XMM249			CR124	CR128	
ZMM250	YMM250	XMM250	ZMM251	YMM251	XMM251			CR125	CR129	
ZMM252	YMM252	XMM252	ZMM253	YMM253	XMM253			CR126	CR130	
ZMM254	YMM254	XMM254	ZMM255	YMM255	XMM255			CR127	CR131	
ZMM256	YMM256	XMM256	ZMM257	YMM257	XMM257			CR128	CR132	
ZMM258	YMM258	XMM258	ZMM259	YMM259	XMM259			CR129	CR133	
ZMM260	YMM260	XMM260	ZMM261	YMM261	XMM261			CR130	CR134	
ZMM262	YMM262	XMM262	ZMM263	YMM263	XMM263			CR131	CR135	
ZMM264	YMM264	XMM264	ZMM265	YMM265	XMM265			CR132	CR136	
ZMM266	YMM266	XMM266	ZMM267	YMM267	XMM267			CR133	CR137	
ZMM268	YMM268	XMM268	ZMM269	YMM269	XMM269			CR134	CR138	
ZMM270	YMM270	XMM270	ZMM271	YMM271	XMM271			CR135	CR139	
ZMM272	YMM272	XMM272	ZMM273	YMM273	XMM273			CR136	CR140	
ZMM274	YMM274	XMM274	ZMM275	YMM275	XMM275			CR137	CR141	
ZMM276	YMM276	XMM276	ZMM277	YMM277	XMM277			CR138	CR142	
ZMM278	YMM278	XMM278	ZMM279	YMM279	XMM279			CR139	CR143	
ZMM280	YMM280	XMM280	ZMM281	YMM281	XMM281			CR140	CR144	
ZMM282	YMM282	XMM282	ZMM283	YMM283	XMM283			CR141	CR145	
ZMM284										

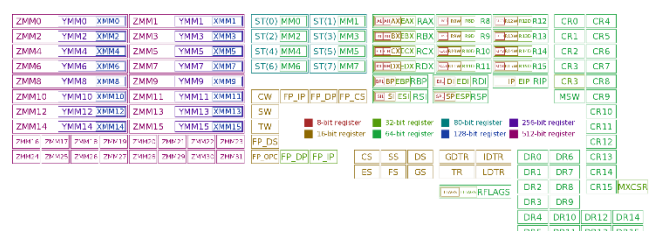
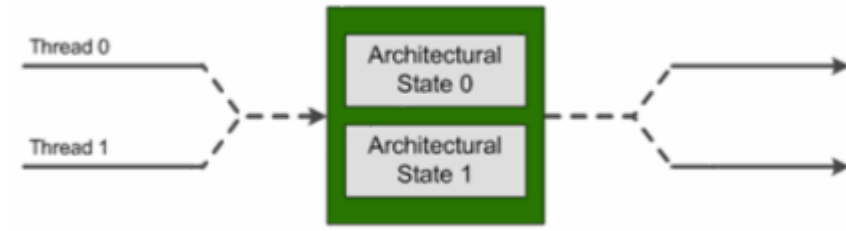
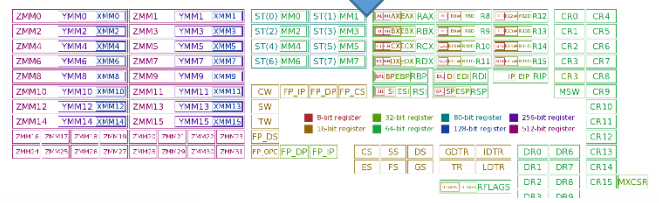
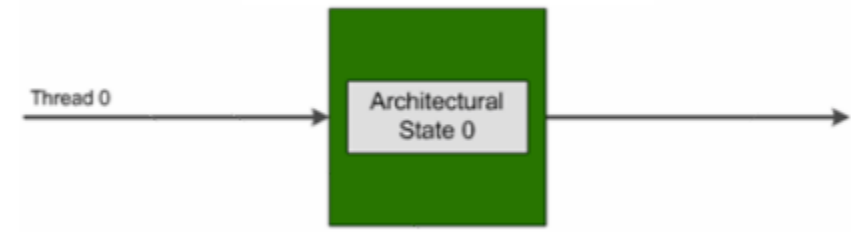
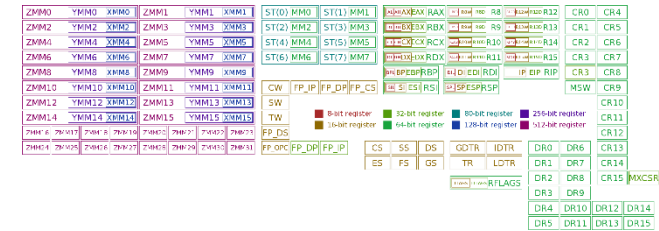
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls



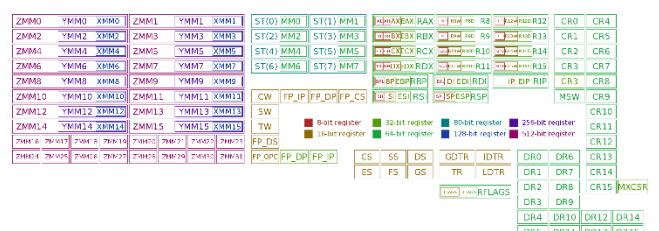
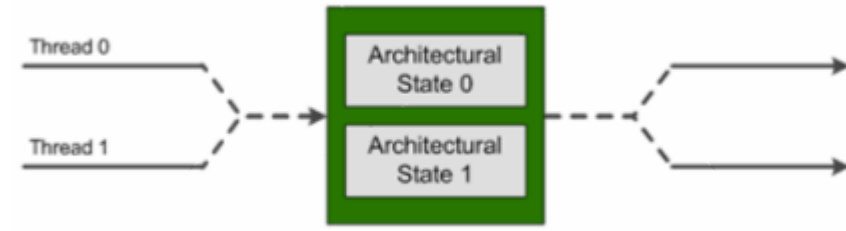
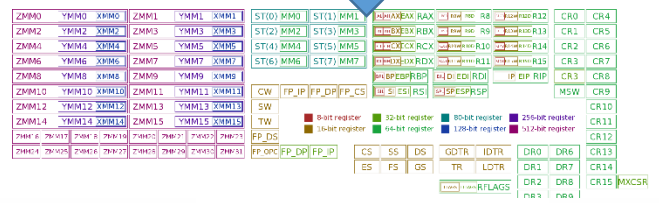
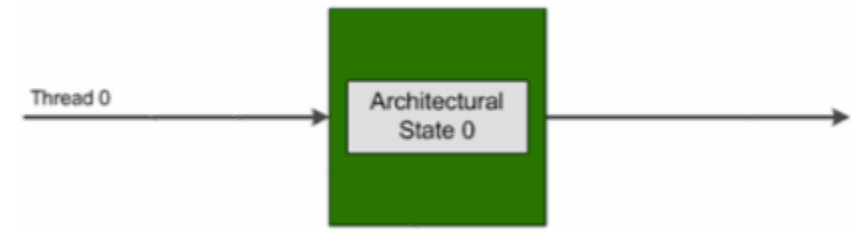
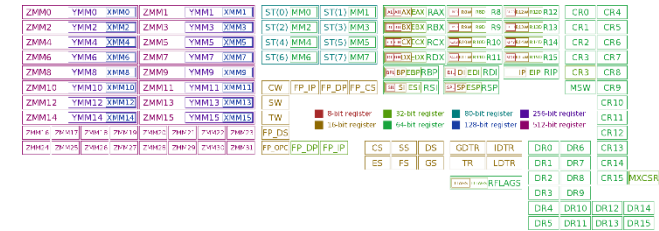
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls
- Looks like multiple cores to the OS



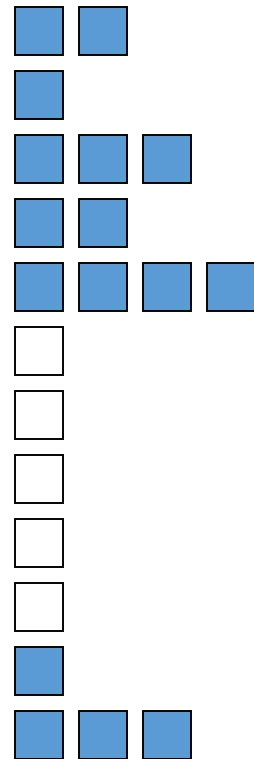
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
 - Coarse
 - Fine-grain
 - Simultaneous

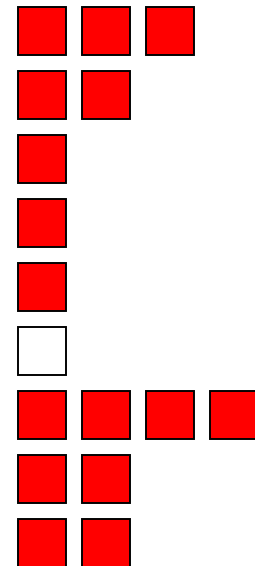


Running example

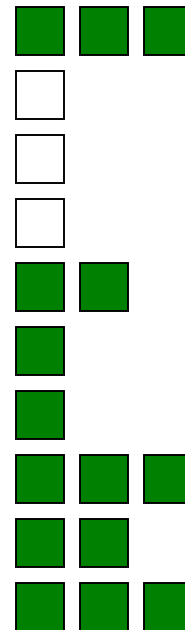
Thread A



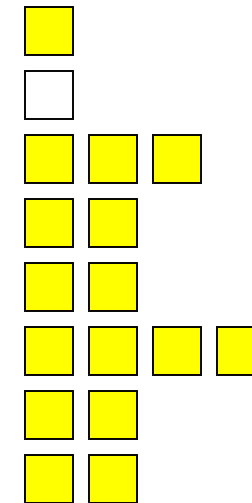
Thread B



Thread C



Thread D



- Colors → pipeline full
- White → stall

Coarse-grained multithreading

Coarse- grained multithreading

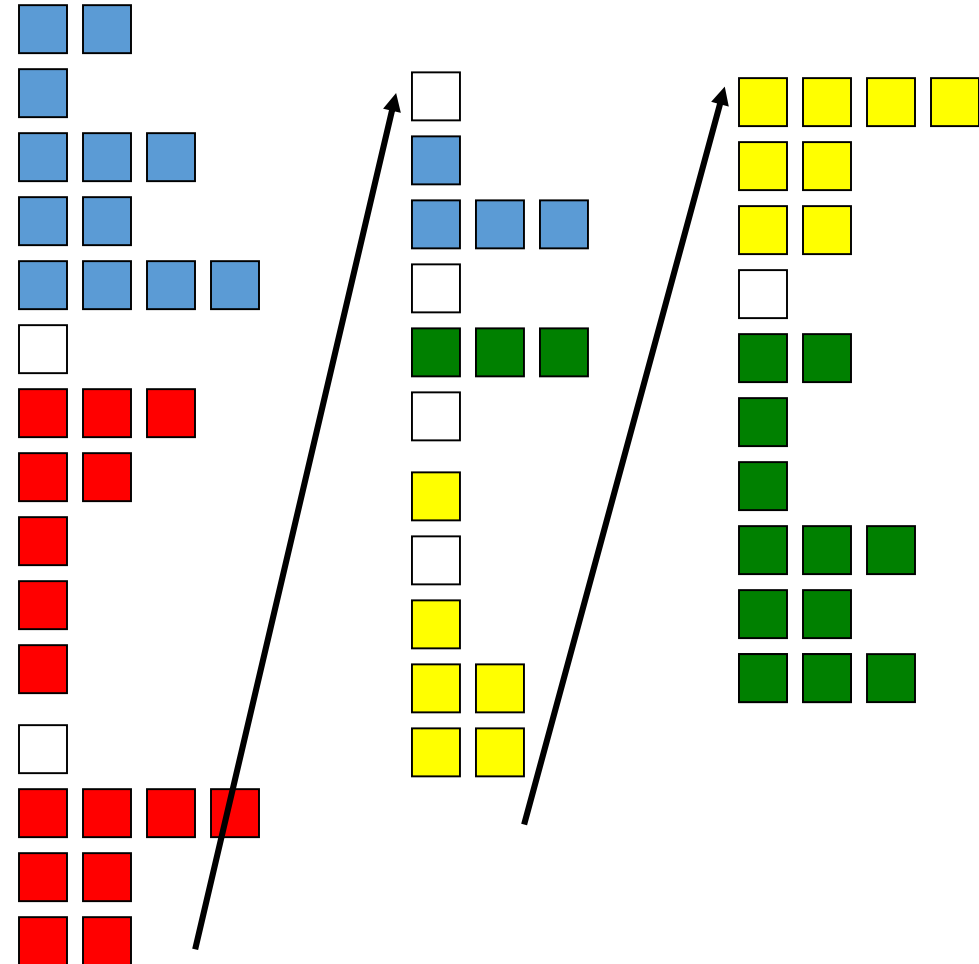
- Single thread runs until a costly stall
 - E.g. 2nd level cache miss

Coarse- grained multithreading

- Single thread runs until a costly stall
 - E.g. 2nd level cache miss
- Another thread starts during stall
 - Pipeline fill time requires several cycles!

Coarse-grained multithreading

- Single thread runs until a costly stall
 - E.g. 2nd level cache miss
- Another thread starts during stall
 - Pipeline fill time requires several cycles!
- Does not cover short stalls



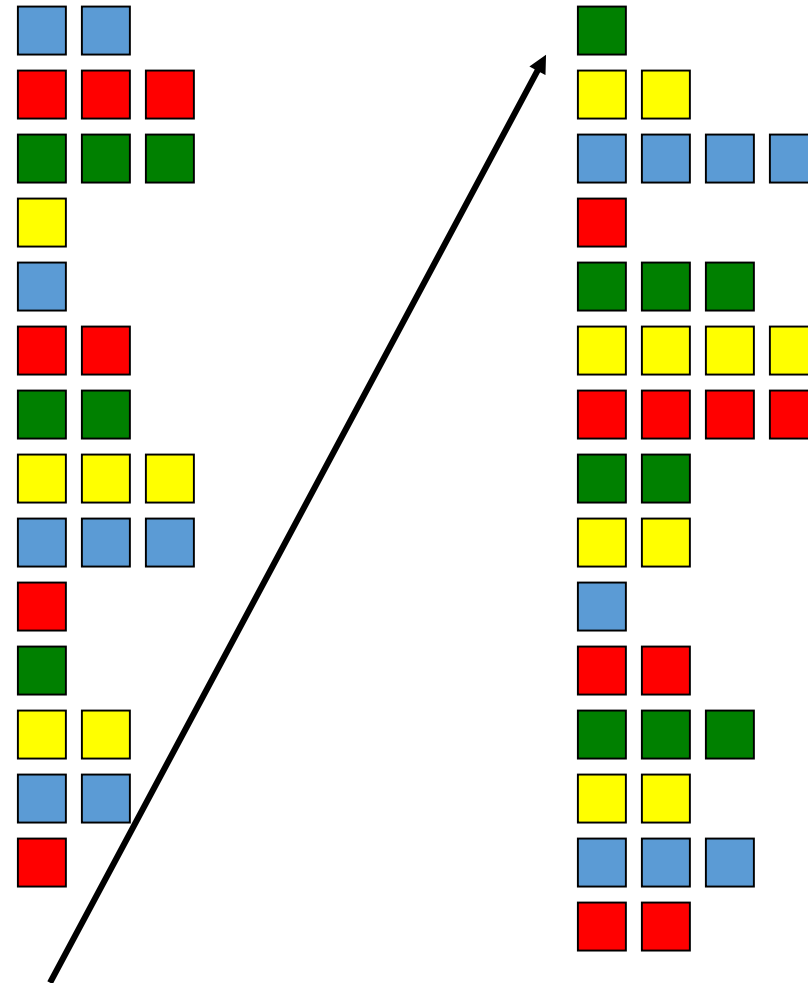
Fine-grained multithreading

Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads

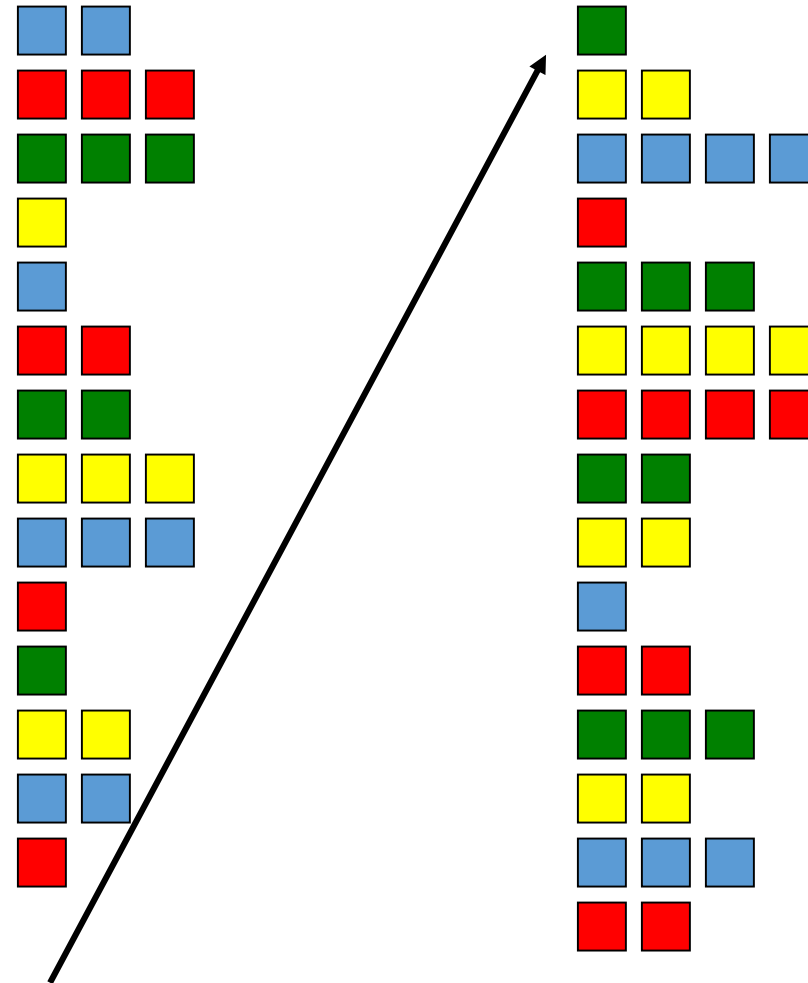
Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads



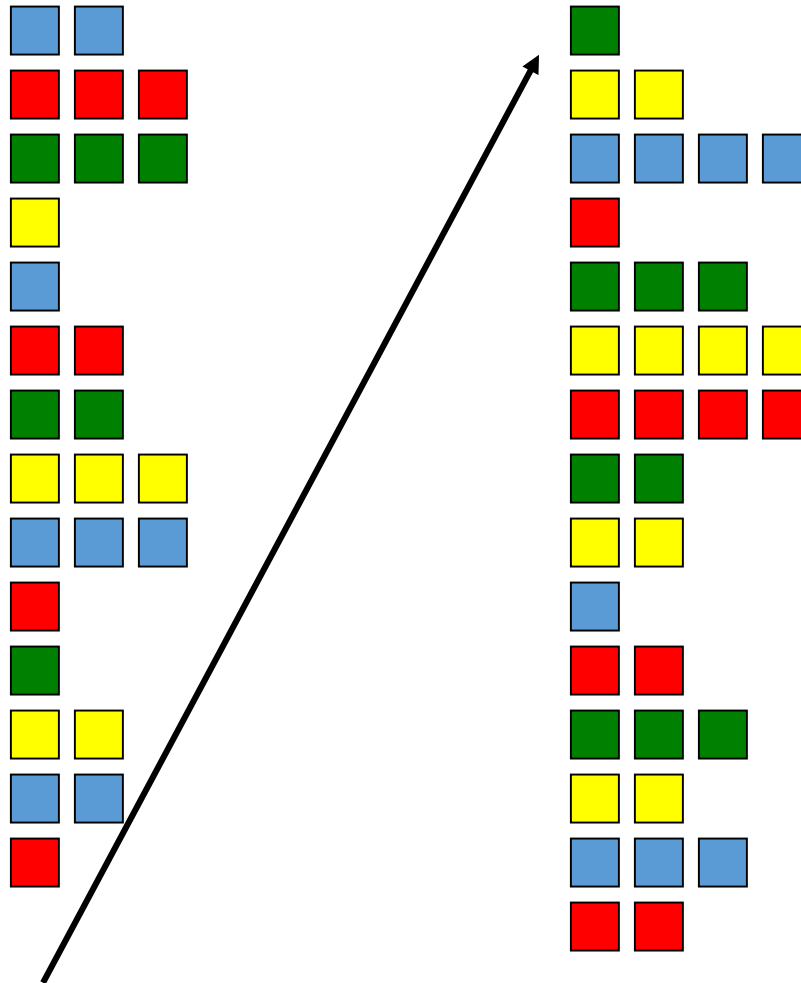
Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads
- Hardware support required
 - Separate PC and register file per thread
 - Hardware to control alternating pattern



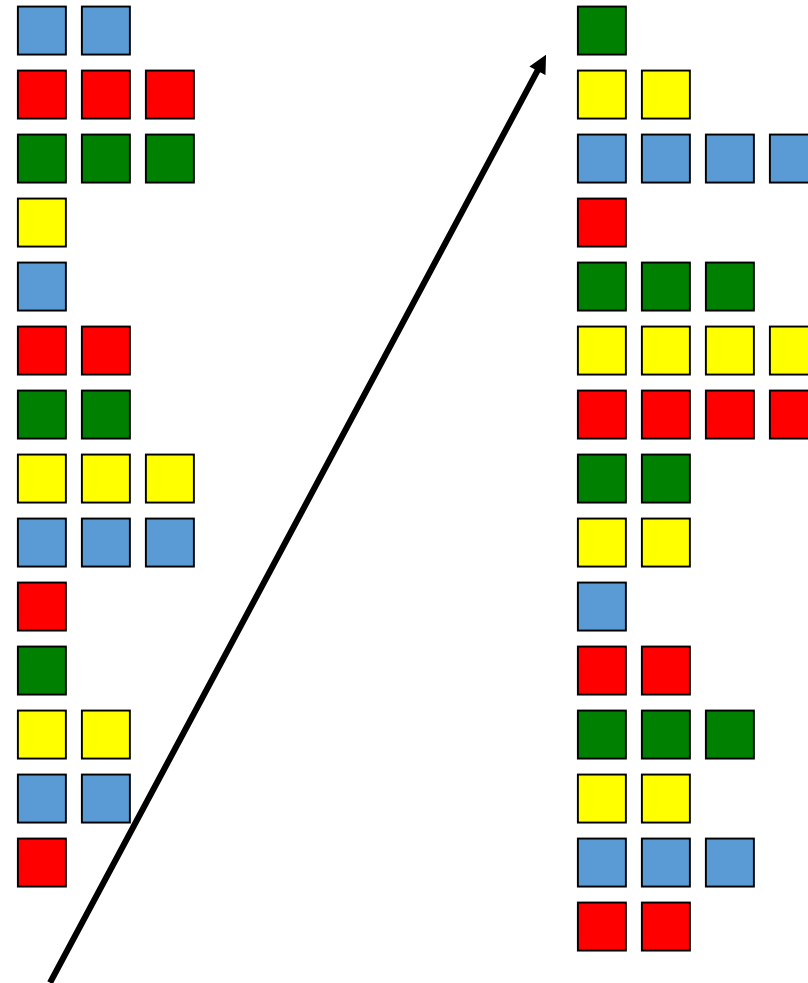
Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads
- Hardware support required
 - Separate PC and register file per thread
 - Hardware to control alternating pattern
- Naturally hides delays
 - Data hazards, Cache misses
 - Pipeline runs with rare stalls



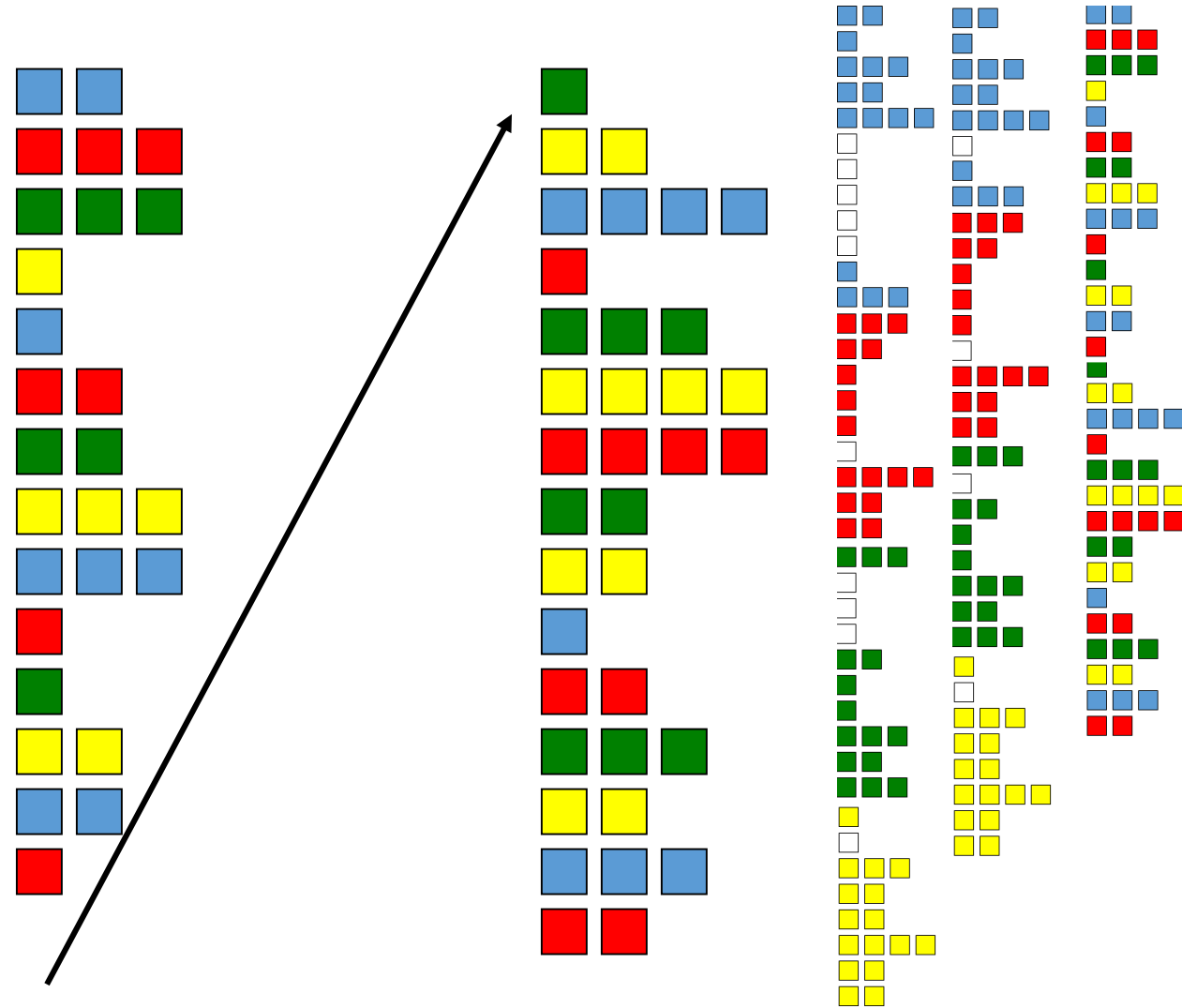
Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads
- Hardware support required
 - Separate PC and register file per thread
 - Hardware to control alternating pattern
- Naturally hides delays
 - Data hazards, Cache misses
 - Pipeline runs with rare stalls
- Doesn't make full use of multi-issue



Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads
- Hardware support required
 - Separate PC and register file per thread
 - Hardware to control alternating pattern
- Naturally hides delays
 - Data hazards, Cache misses
 - Pipeline runs with rare stalls
- Doesn't make full use of multi-issue



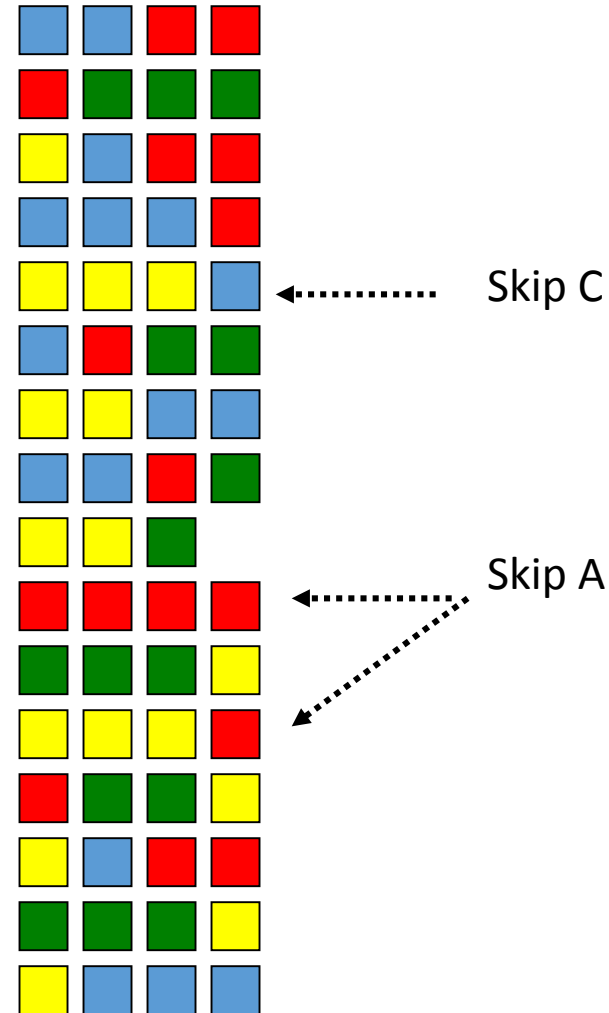
Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture

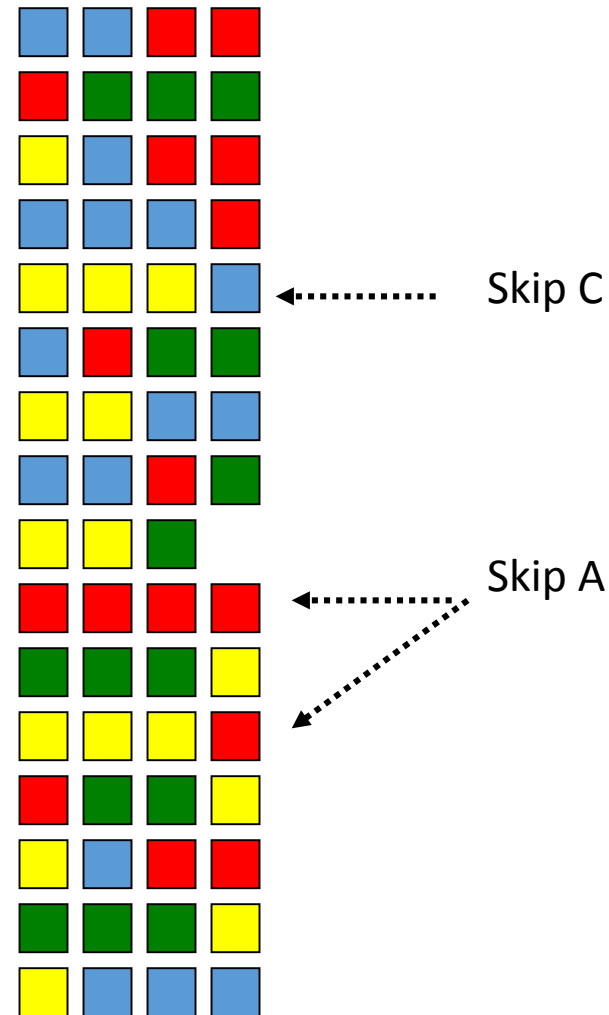
Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture



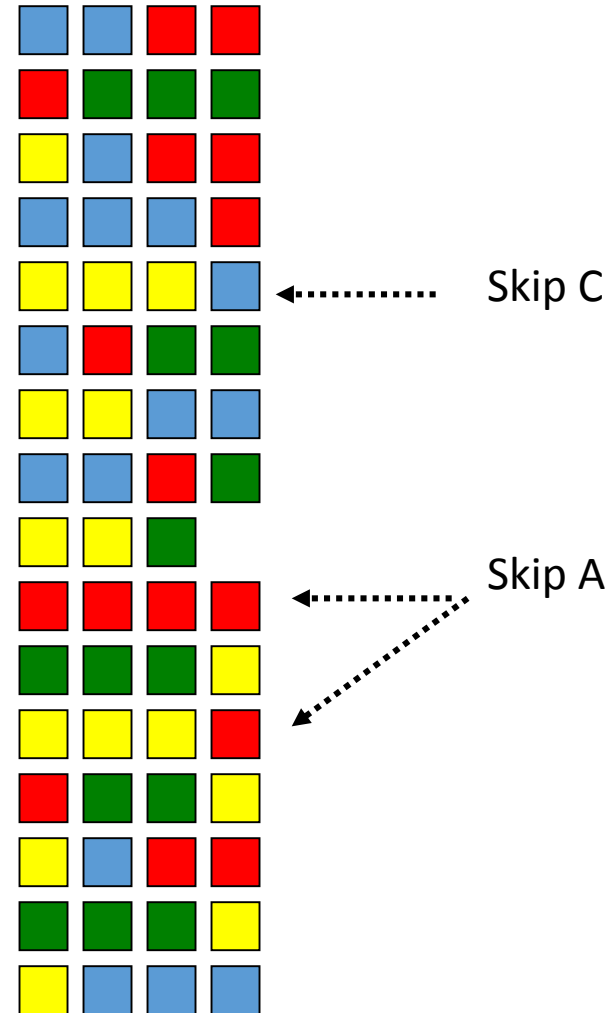
Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture
- Hardware support:
 - Register files, PCs per thread
 - Temporary result registers pre commit
 - Support to sort out which threads get results from which instructions



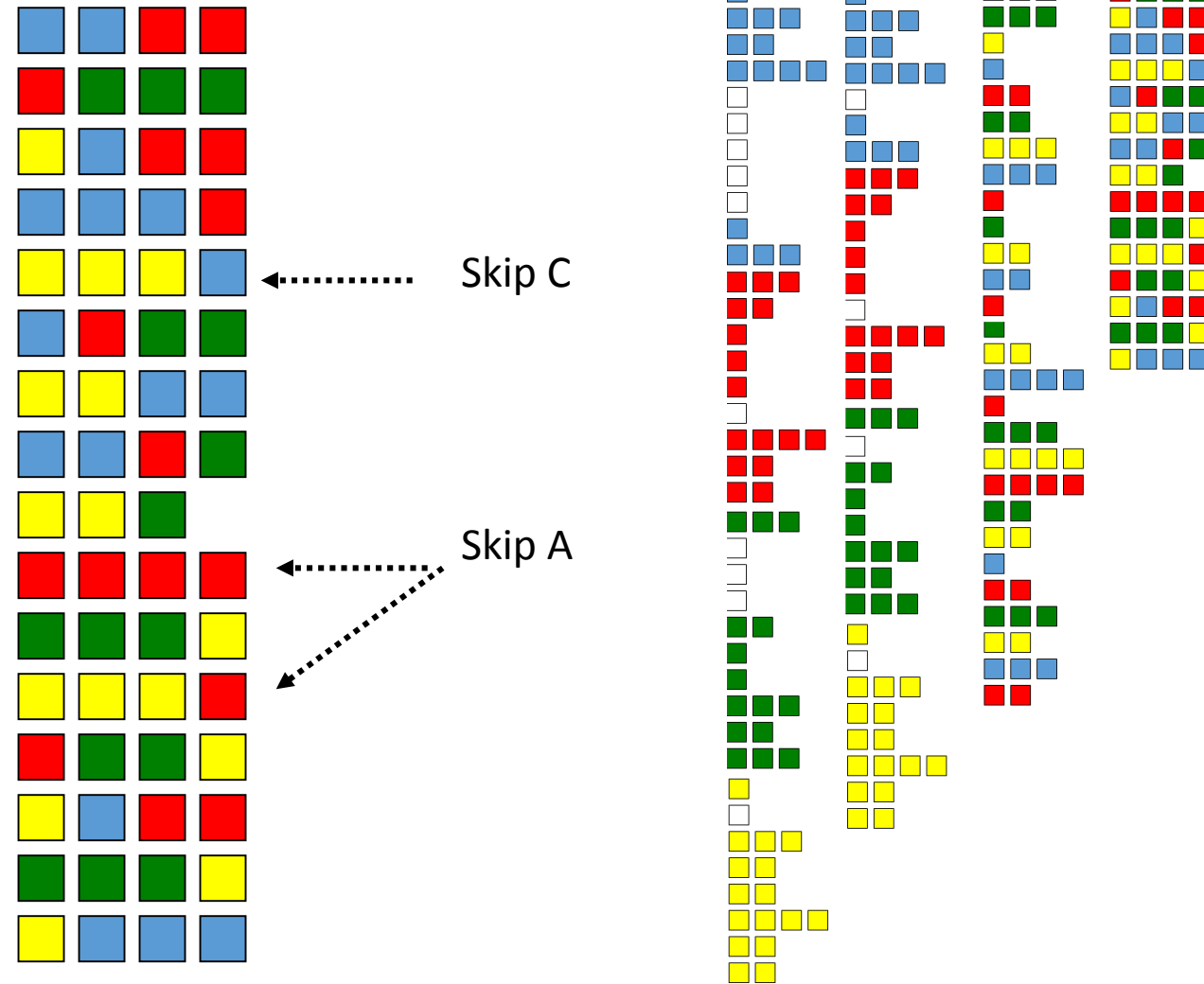
Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture
- Hardware support:
 - Register files, PCs per thread
 - Temporary result registers pre commit
 - Support to sort out which threads get results from which instructions
- Maximal util. of execution units



Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture
- Hardware support:
 - Register files, PCs per thread
 - Temporary result registers pre commit
 - Support to sort out which threads get results from which instructions
- Maximal util. of execution units



Why Vector and Multithreading Background?

GPU:

- A very wide vector machine
- Massively multi-threaded to hide memory latency
- Originally designed for graphics pipelines...

Graphics \approx Rendering

Graphics \approx Rendering

Inputs

Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures → geometry

Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures → geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials

Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures → geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials
- View point

Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures → geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials
- View point

Output

Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures → geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials
- View point

Output

- 2D projection seen from the view-point

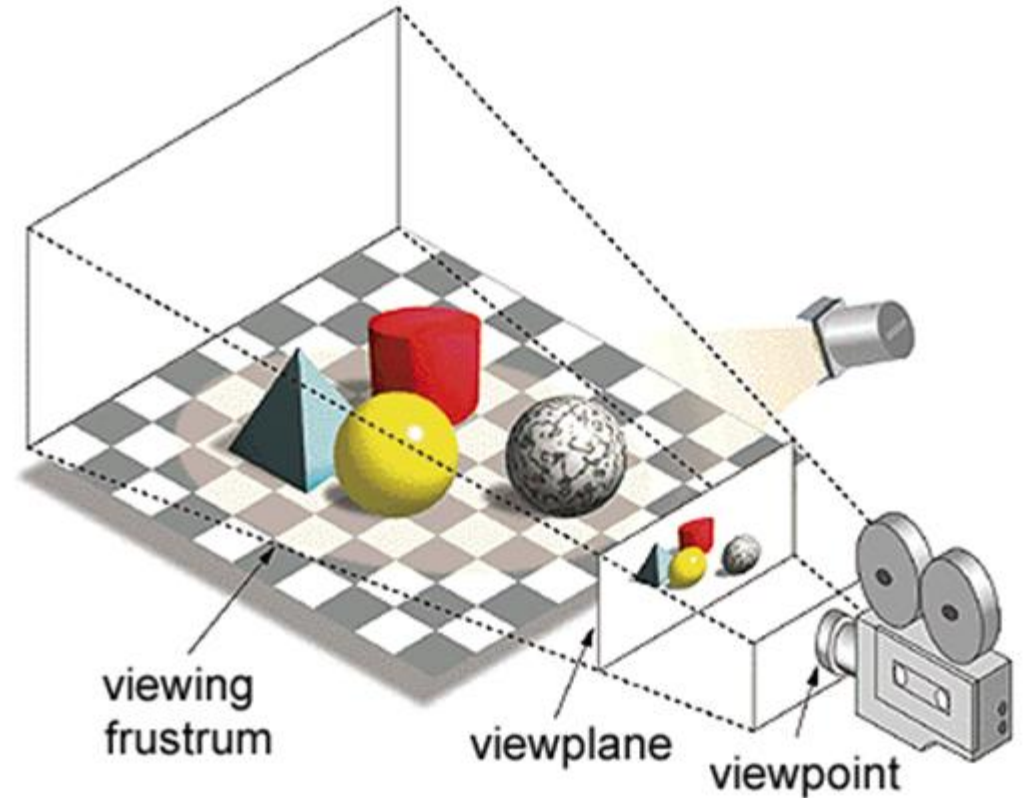
Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterizat
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures \rightarrow geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials
- View point

Output

- 2D projection seen from the view-point



Grossly over-simplified rendering algorithm

Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```


Grossly over-simplified rendering algorithm

foreach(vertex v in model)

map $v_{\text{model}} \rightarrow v_{\text{view}}$

Grossly over-simplified rendering algorithm

foreach(vertex v in model)

map $v_{\text{model}} \rightarrow v_{\text{view}}$



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```

```
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
    fragment[] frags = {};
```



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```

```
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
    fragment[] frags = {};
```

```
    foreach triangle t ( $v_0, v_1, v_2$ )
```



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```

```
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
    fragment[] frags = {};
```

```
    foreach triangle t ( $v_0, v_1, v_2$ )
```

```
        frags.add(rasterize(t));
```



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```

```
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
    fragment[] frags = {};
```

```
    foreach triangle t ( $v_0, v_1, v_2$ )
```

```
        frags.add(rasterize(t));
```

```
    foreach fragment f in frags
```



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
```

```
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

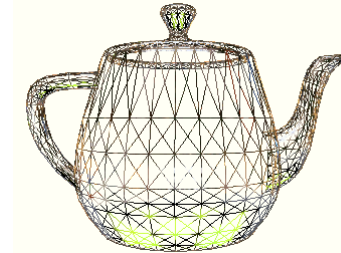
```
fragment[] frags = {};
```

```
foreach triangle t ( $v_0, v_1, v_2$ )
```

```
    frags.add(rasterize(t));
```

```
foreach fragment f in frags
```

```
    choose_color(f);
```



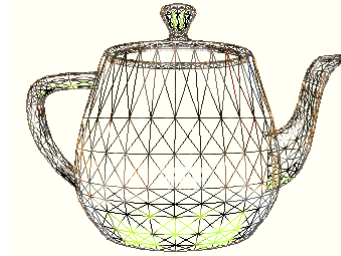
Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



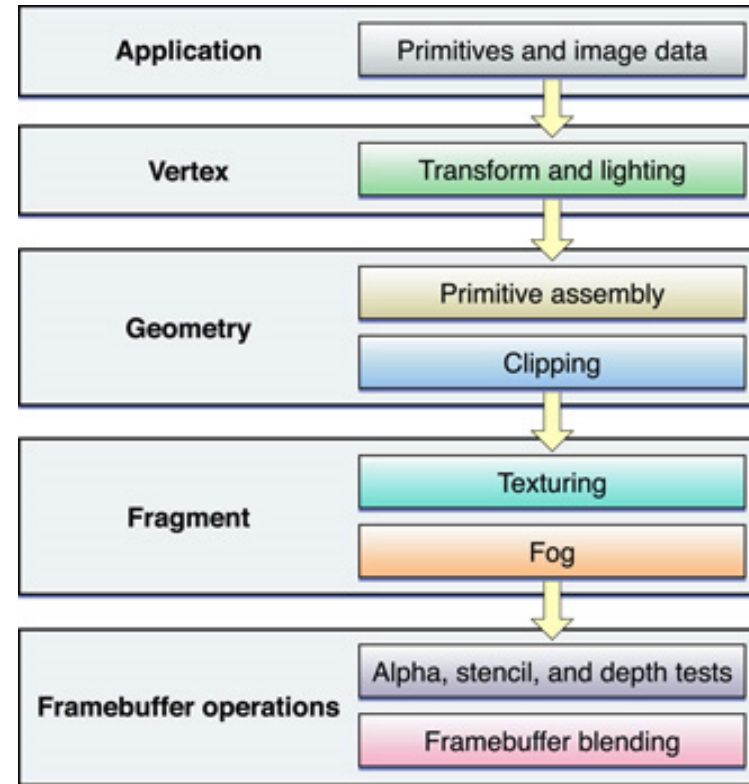
Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



Algorithm \rightarrow Graphics Pipeline

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



OpenGL pipeline

To first order, DirectX looks the same!

Algorithm \rightarrow Graphics Pipeline

```
foreach(vertex v in model)
```

```
  map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
  fragment[] frags = {};
```

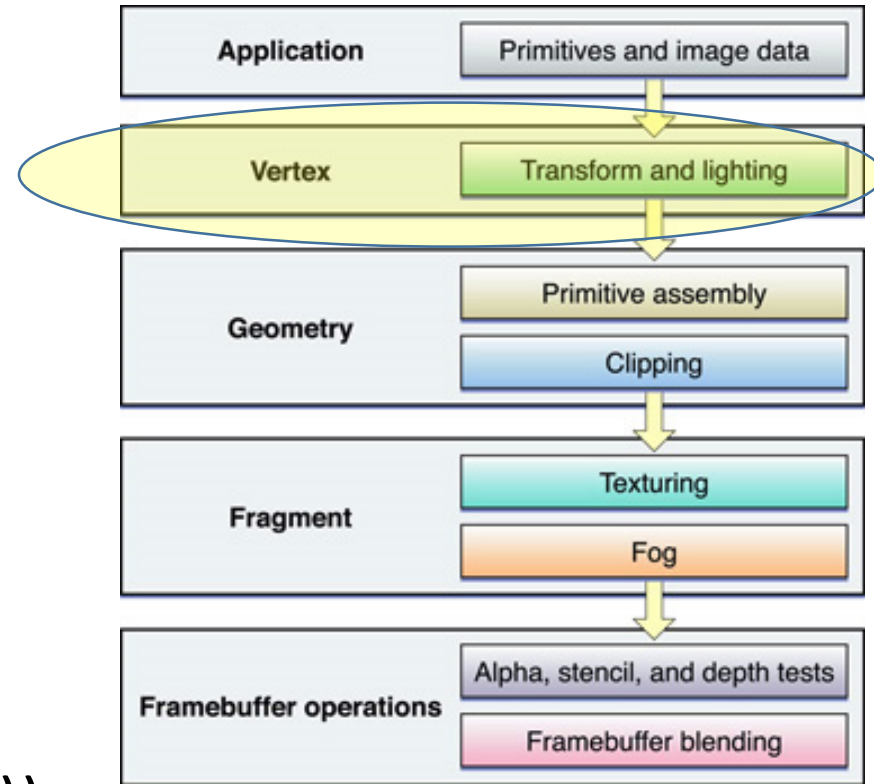
```
  foreach triangle t ( $v_0, v_1, v_2$ )
```

```
    frags.add(rasterize(t));
```

```
  foreach fragment f in frags
```

```
    choose_color(f);
```

```
  display(visible_fragments(frags));
```

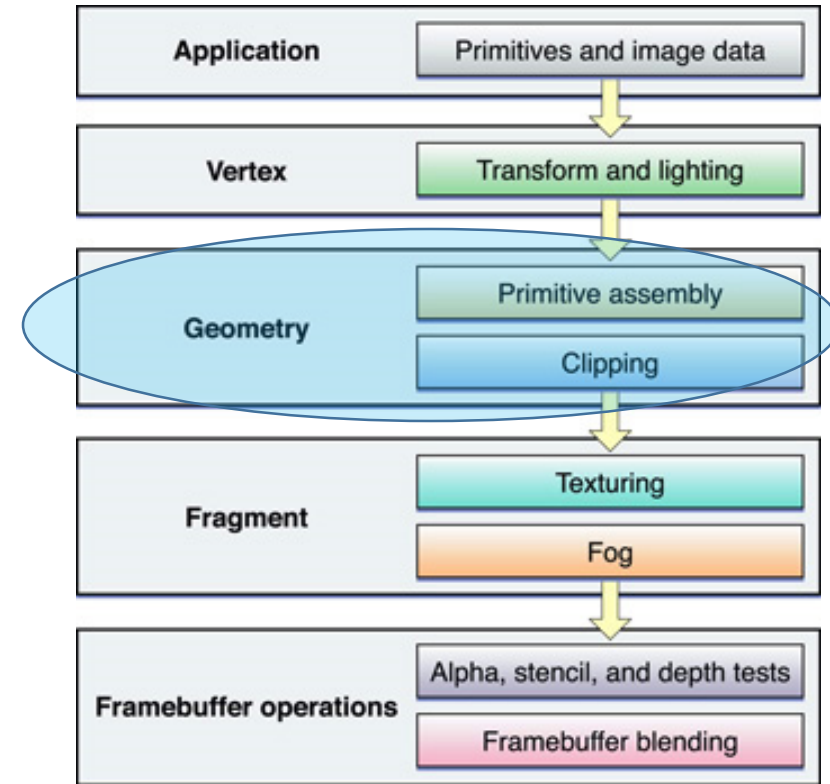


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm \rightarrow Graphics Pipeline

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```

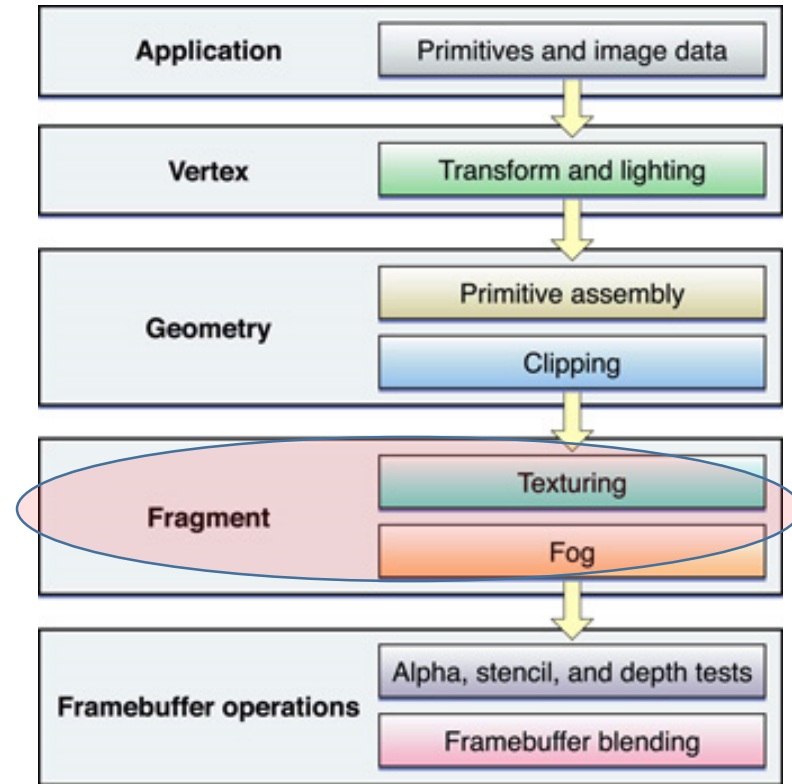


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm \rightarrow Graphics Pipeline

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```

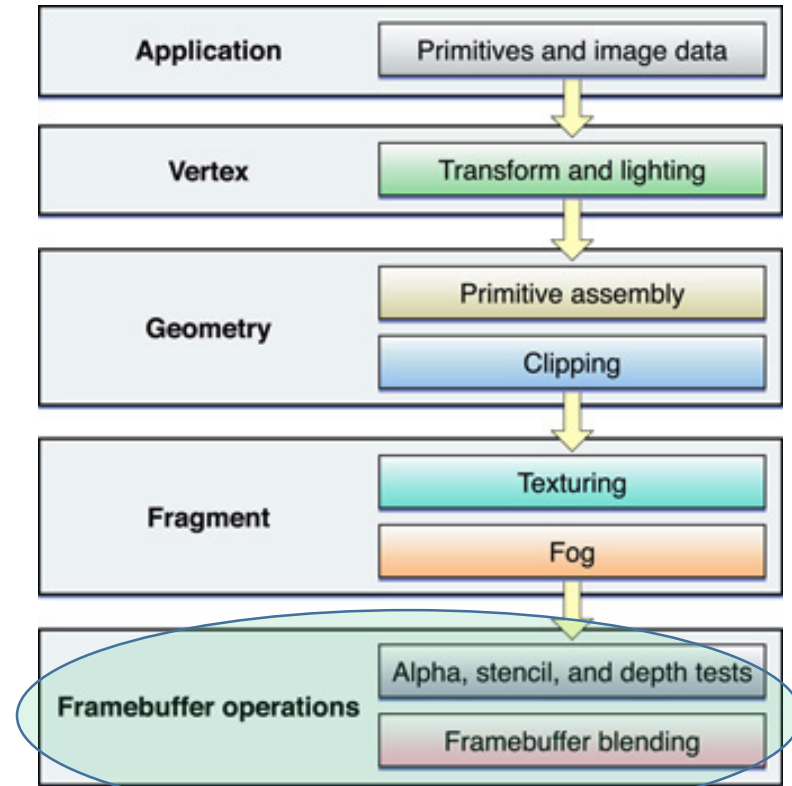


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm \rightarrow Graphics Pipeline

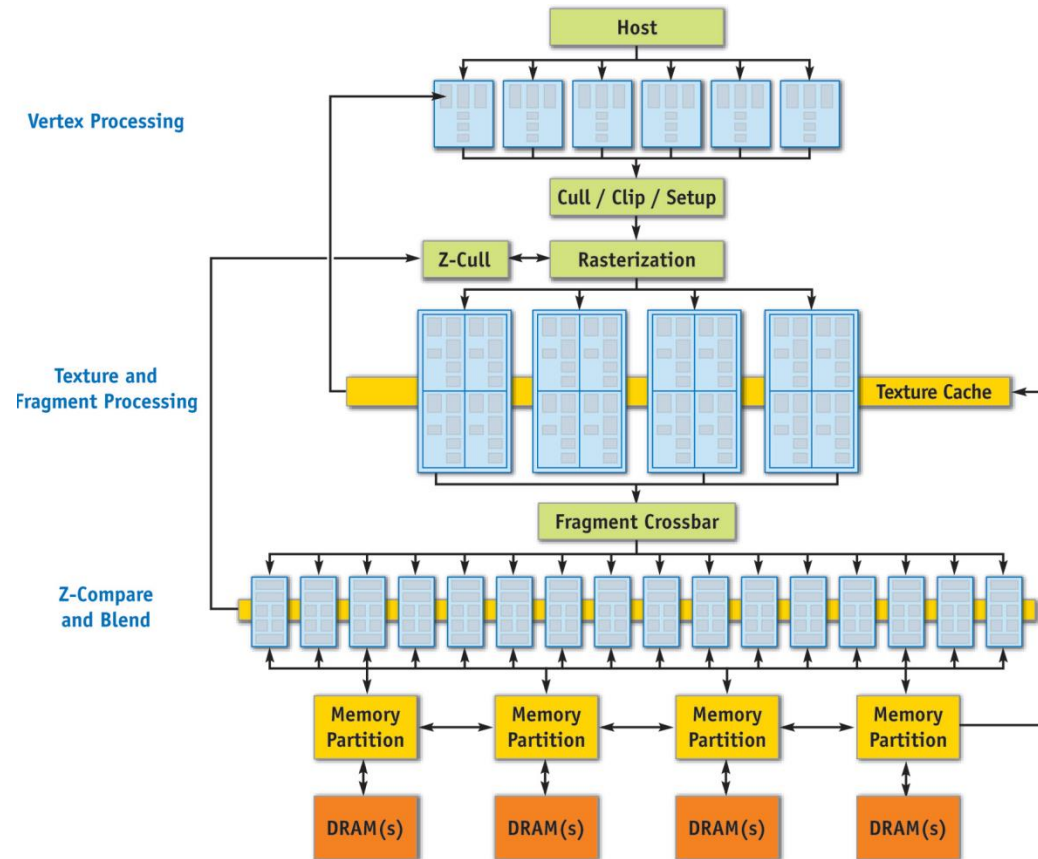
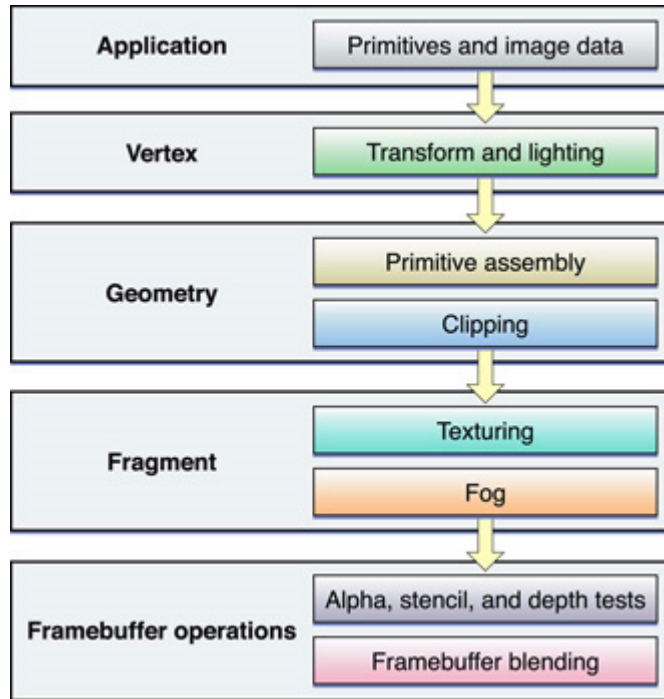
```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



OpenGL pipeline

To first order, DirectX looks the same!

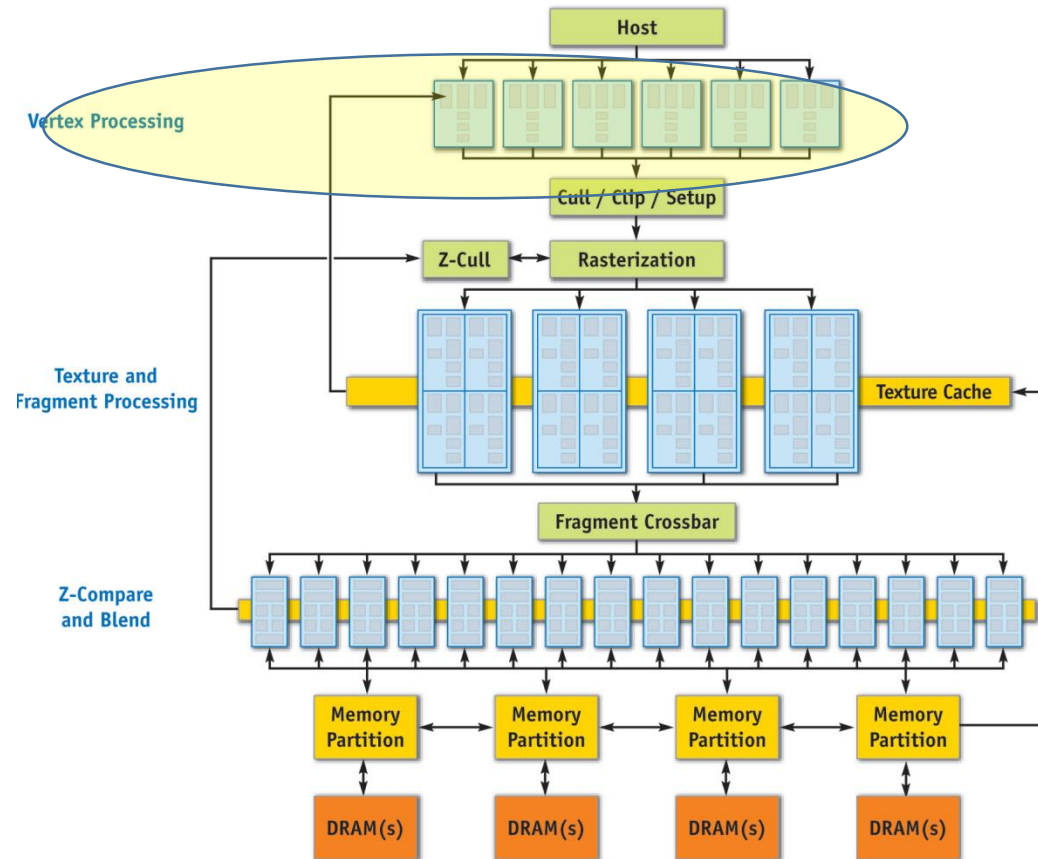
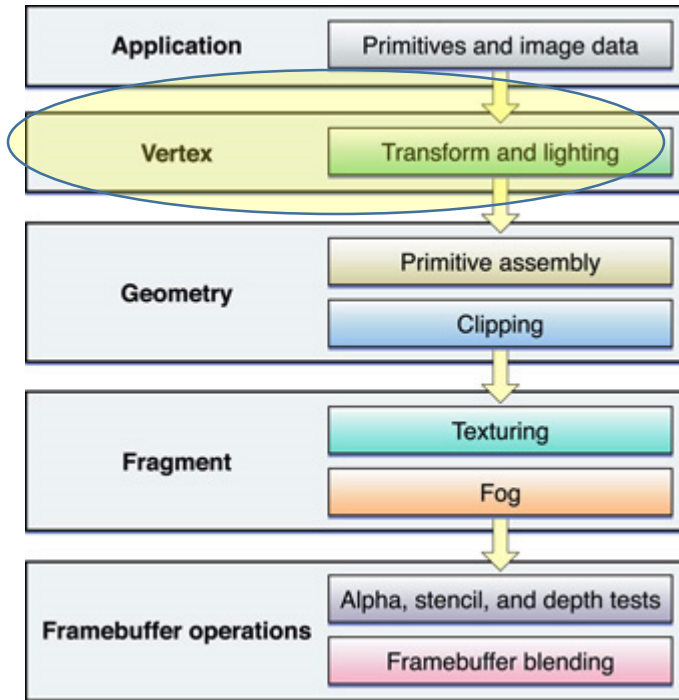
Graphics pipeline → GPU architecture



GeForce 6 series

Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

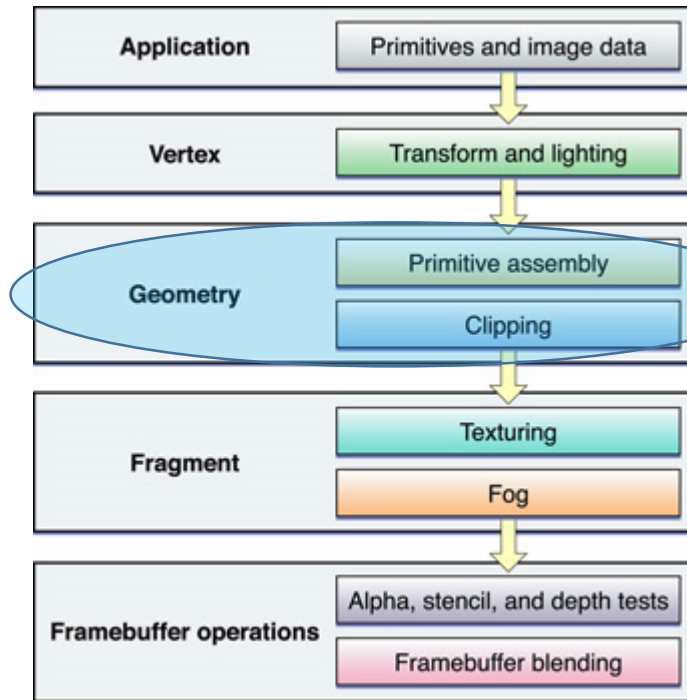
Graphics pipeline → GPU architecture



GeForce 6 series

Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

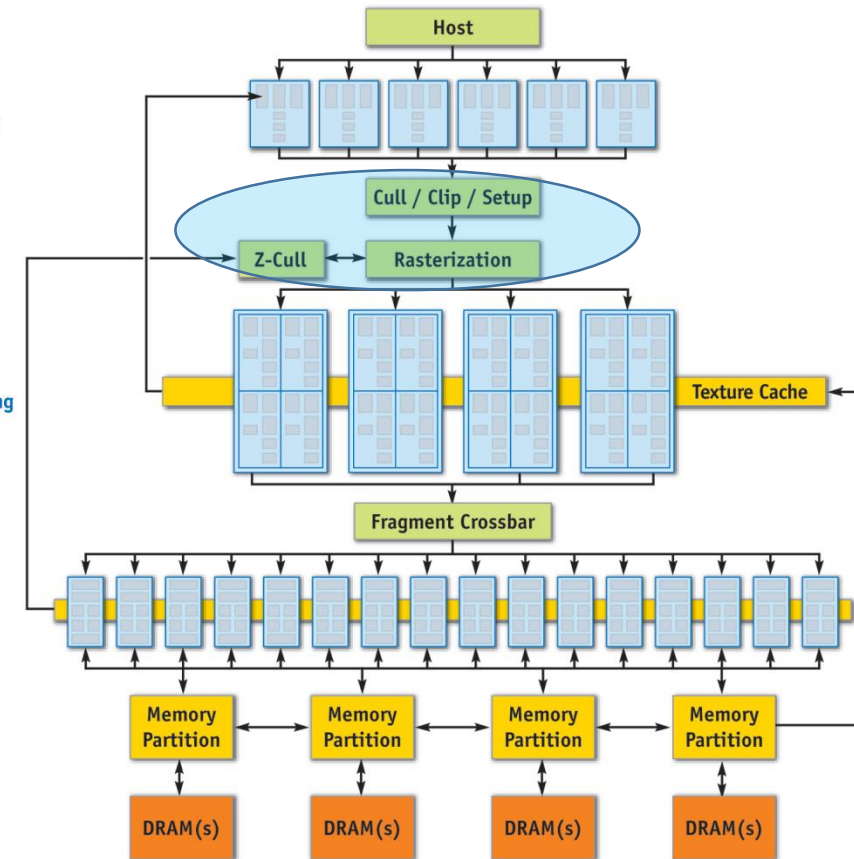
Graphics pipeline → GPU architecture



Vertex Processing

Texture and Fragment Processing

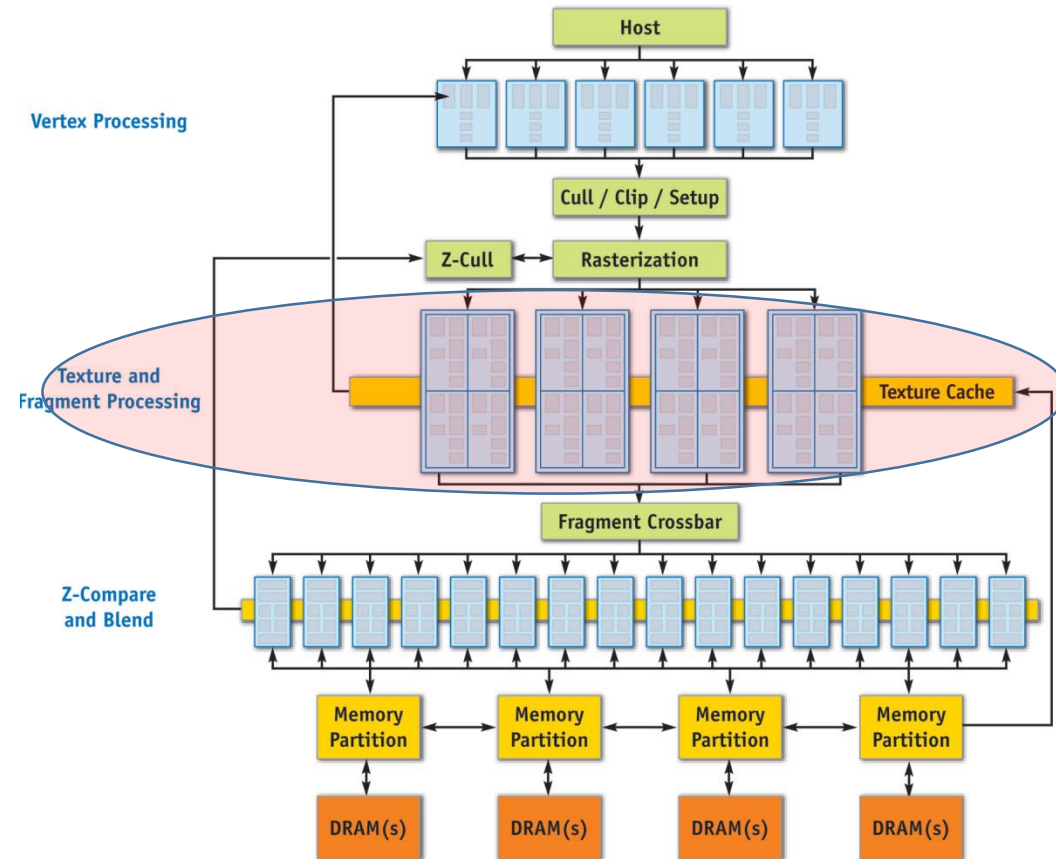
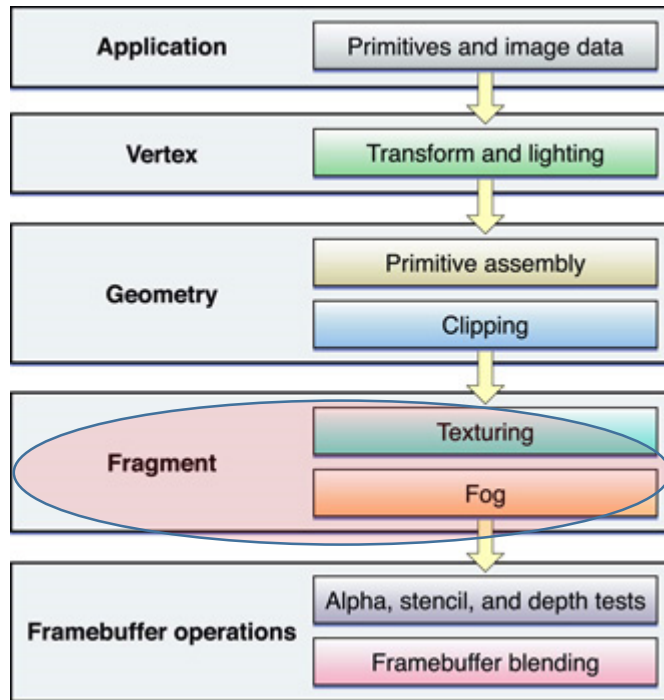
Z-Compare and Blend



GeForce 6 series

Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

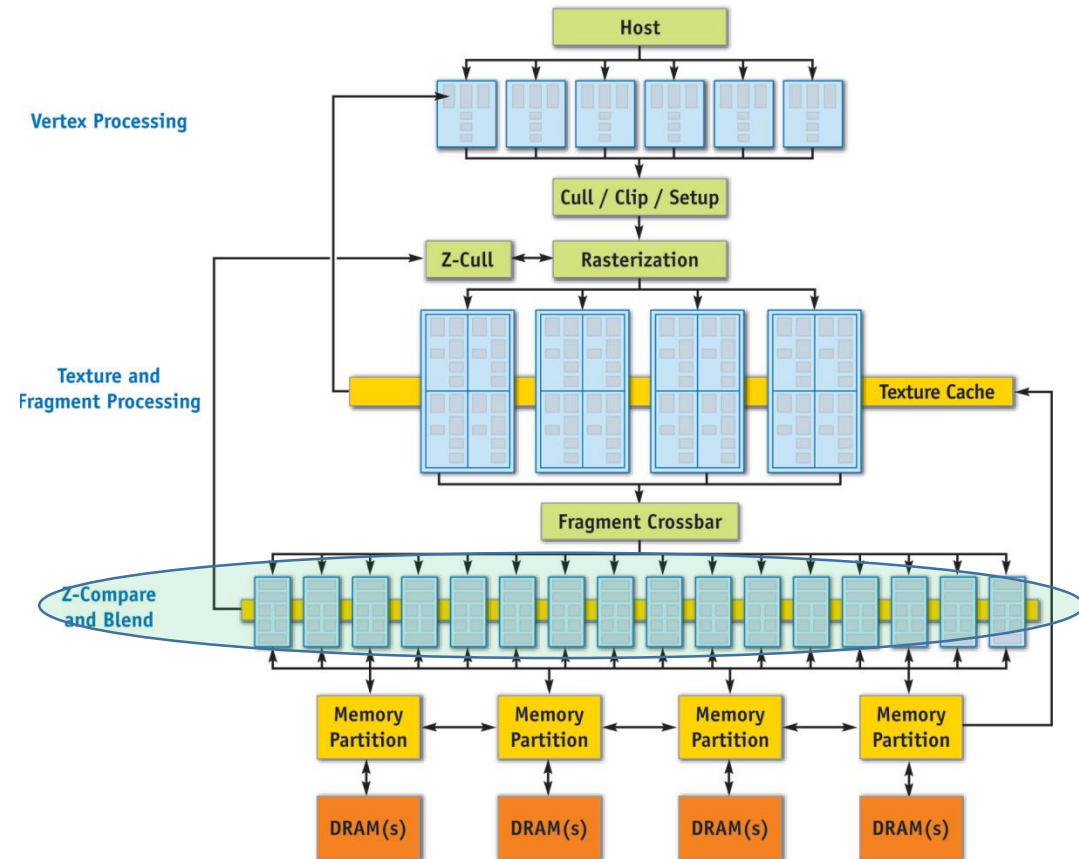
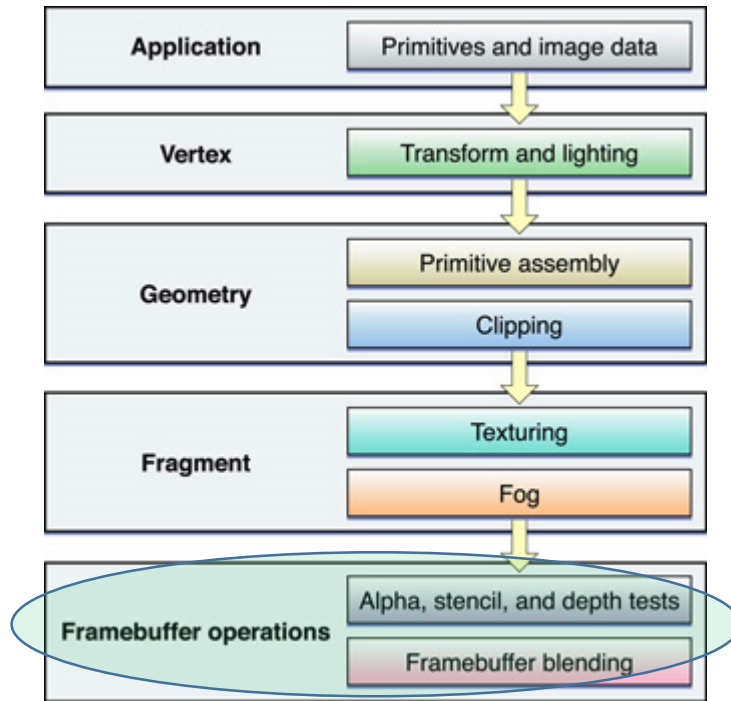
Graphics pipeline → GPU architecture



GeForce 6 series

Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

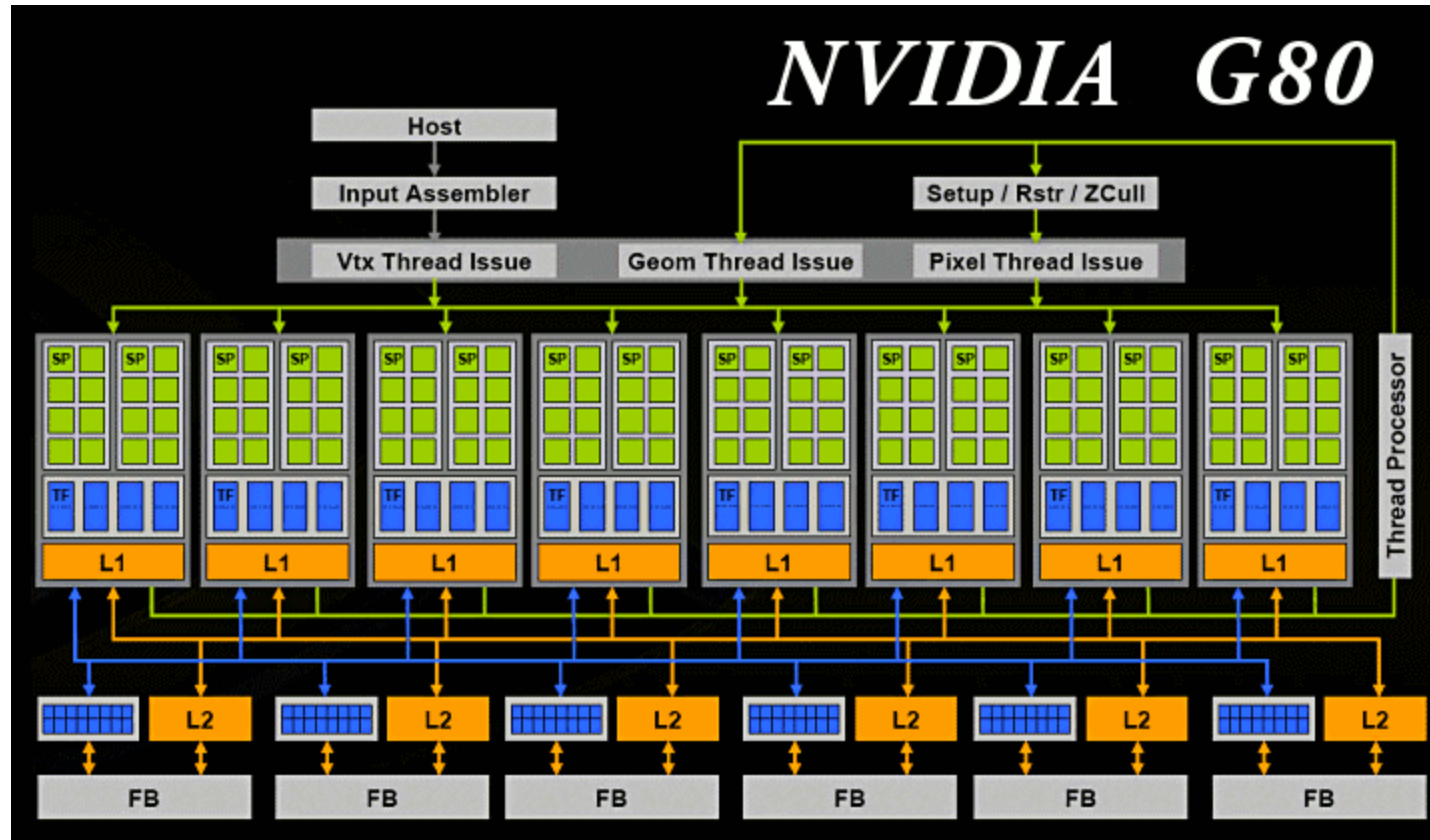
Graphics pipeline → GPU architecture



GeForce 6 series

Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

Late Modernity: unified shaders



Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count

Mostly Modern: Pascal



Modern Enough: Pascal SM



Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

- Data
 - Per-vertex
 - Per-fragment
 - Per-pixel
- Task
 - Vertex processing
 - Fragment processing
 - Rasterization
 - Hidden-surface elimination
- MLP
 - HW multi-threading for hiding memory latency

Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

- Data
 - Per-vertex
 - Per-fragment
 - Per-pixel
- Task
 - Vertex processing
 - Fragment processing
 - Rasterization
 - Hidden-surface elimination
- MLP
 - HW multi-threading for hiding memory latency

Even as GPU architectures become more general, certain assumptions persist:

1. Data parallelism is *trivially* exposed
2. **All** problems look like painting a box with colored dots

Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

- Data
 - Per-vertex
 - Per-fragment
 - Per-pixel
- Task
 - Vertex processing
 - Fragment processing
 - Rasterization
 - Hidden-surface elimination
- MLP
 - HW multi-threading for hiding memory latency

Even as GPU architectures become more general, certain assumptions persist:

1. Data parallelism is *trivially* exposed
2. All problems look like painting a box with colored dots

But what if my problem isn't painting a box?!?!

The big ideas still present in GPUs

- Simple cores
- Single instruction stream
 - Vector instructions (SIMD) OR
 - Implicit HW-managed sharing (SIMT)
- Hide memory latency with HW multi-threading

Programming Model

- ***GPUs are I/O devices, managed by user-code***
- “kernels” == “shader programs”
- 1000s of HW-scheduled threads per kernel
- Threads grouped into independent blocks.
 - Threads in a block can synchronize (barrier)
 - This is the **only** synchronization
- “Grid” == “launch” == “invocation” of a kernel
 - a group of blocks (or warps)

Parallel Algorithms

- Sequential algorithms often do not permit easy parallelization
 - Does not mean there work has no parallelism
 - A different approach can yield parallelism
 - but often changes the algorithm
 - Parallelizing != just adding locks to a sequential algorithm
- Parallel Patterns
 - Map
 - Scatter, Gather
 - Reduction
 - Scan
 - Search, Sort

Parallel Algorithms

- Sequential algorithms often do not permit easy parallelization
 - Does not mean there work has no parallelism
 - A different approach can yield parallelism
 - but often changes the algorithm
 - Parallelizing != just adding locks to a sequential algorithm
- Parallel Patterns
 - Map
 - Scatter, Gather
 - Reduction
 - Scan
 - Search, Sort

If you can express your algorithm using these patterns, an apparently fundamentally sequential algorithm can be made parallel

Map

- Inputs
 - Array A
 - Function $f(x)$
- $\text{map}(A, f) \rightarrow$ apply $f(x)$ on all elements in A
- Parallelism trivially exposed
 - $f(x)$ can be applied in parallel to all elements, in principle

Map

- Inputs
 - Array A
 - Function $f(x)$
- $\text{map}(A, f) \rightarrow$ apply $f(x)$ on all elements in A
- Parallelism trivially exposed
 - $f(x)$ can be applied in parallel to all elements, in principle

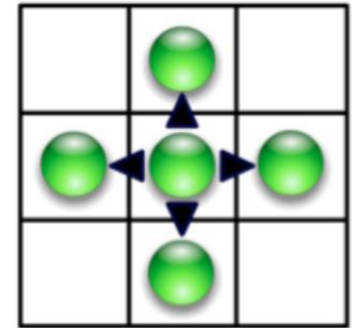
```
for(i=0; i<numPoints; i++) {  
    labels[i] = findNearestCenter(points[i]);  
}
```



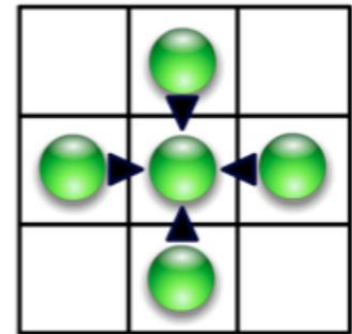
```
map(points, findNearestCenter)
```


Scatter and Gather

- Gather:
 - Read multiple items to single location
- Scatter:
 - Write single data item to multiple locations



Scatter



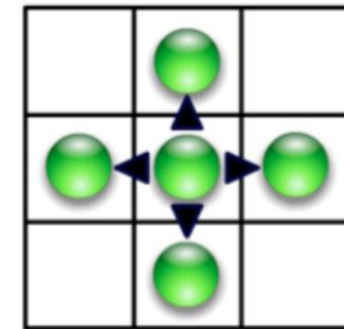
Gather

Scatter and Gather

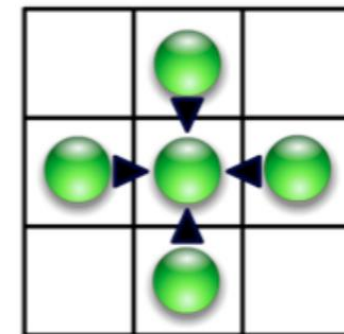
- Gather:
 - Read multiple items to single location
- Scatter:
 - Write single data item to multiple locations

```
for (i=0; i<N; ++i)  
x[i] = y[idx[i]];      →      gather(x, y, idx)
```

```
for (i=0; i<N; ++i)  
y[idx[i]] = x[i];      →      scatter(x, y, idx)
```



Scatter



Gather

Reduce

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
- $\text{Reduce}(op, s)$ returns $a \text{ op } b \text{ op } c \dots \text{ op } z$

Reduce

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
- $\text{Reduce}(op, s)$ returns $a \text{ op } b \text{ op } c \dots \text{ op } z$

```
for(i=0; i<N; ++i) {  
    accum += (point[i]*point[i])  
}
```



```
accum = reduce(*, point)
```

Reduce

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
- $\text{Reduce}(\text{op}, s)$ returns $a \text{ op } b \text{ op } c \dots \text{ op } z$

```
for(i=0; i<N; ++i) {  
    accum += (point[i]*point[i])  
}
```

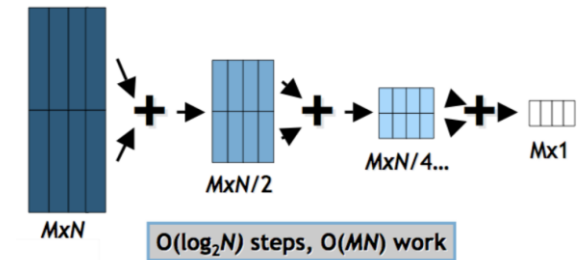
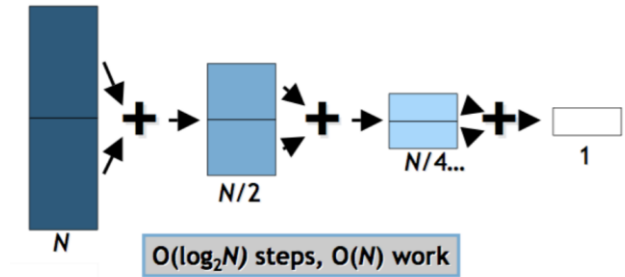


```
accum = reduce(*, point)
```

Why must op be associative?

Reduce

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
- $\text{Reduce}(\text{op}, s)$ returns $a \text{ op } b \text{ op } c \dots \text{ op } z$



```
for(i=0; i<N; ++i) {  
    accum += (point[i]*point[i])  
}
```

accum = reduce(*, point)

Why must op be associative?

Scan (prefix sum)

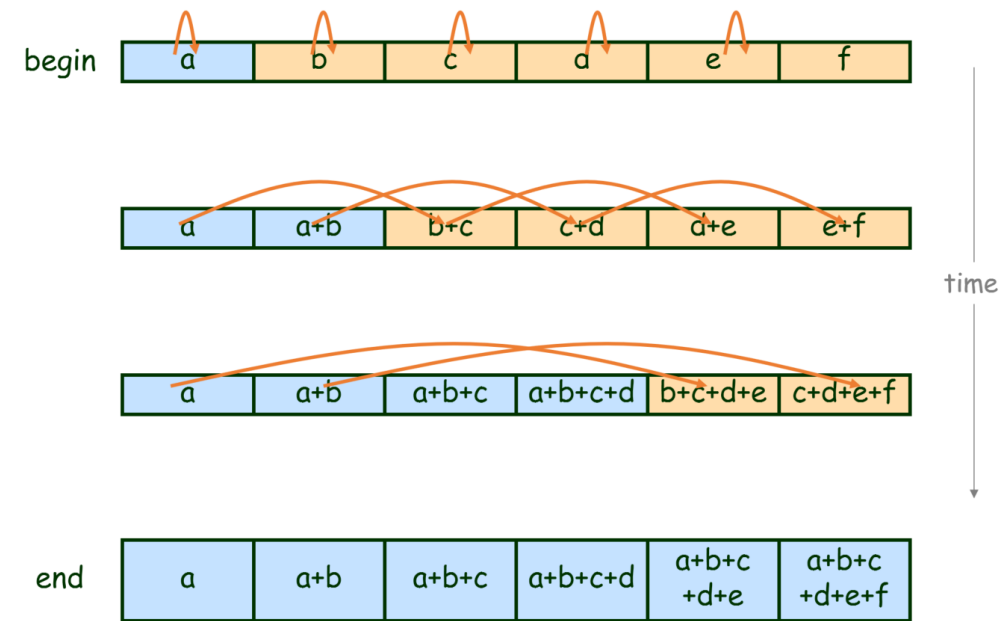
- Input

- Associative operator **op**
- Ordered set $s = [a, b, c, \dots z]$
- Identity I

- $\text{scan}(\text{op}, s) = [I, a, (a \text{ op } b), (a \text{ op } b \text{ op } c) \dots]$

- Scan is the workhorse of parallel algorithms:

- Sort, histograms, sparse matrix, string compare, ...



Summary

- Re-expressing apparently sequential algorithms as combinations of parallel patterns is a common technique when targeting GPUs