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) (results []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

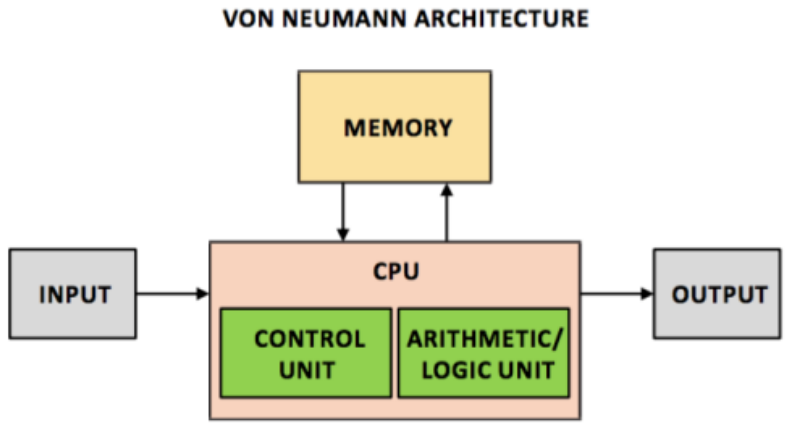
- Reduce tail latency with replication. No locks, conditions, callbacks!

```
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
```

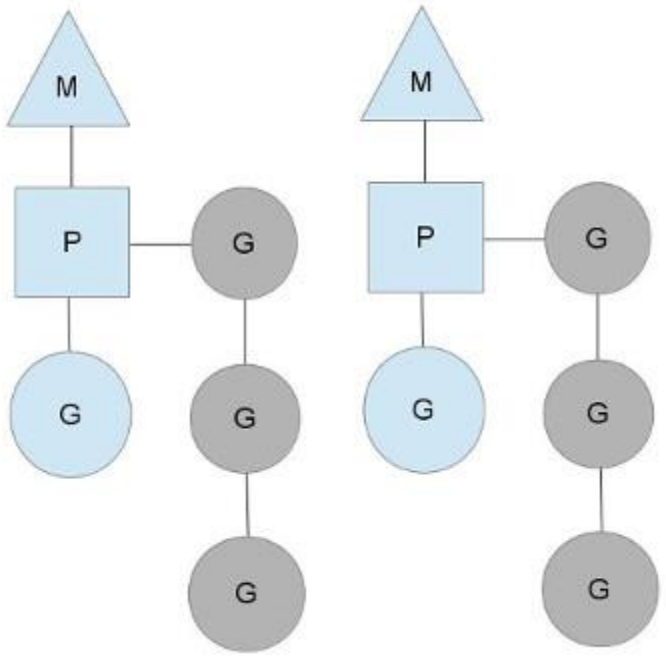
```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

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

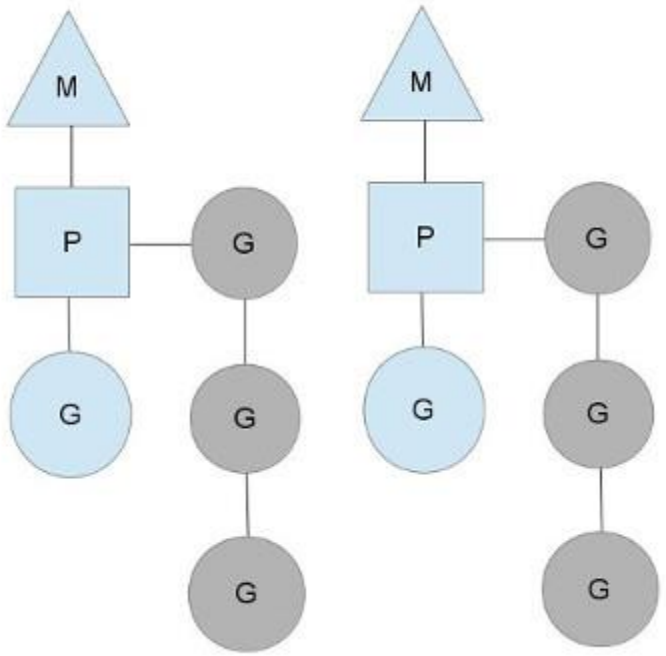
- We've seen several abstractions for
  - Control flow/exection
  - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
  - Why is it OK to have 100,000s of goroutines?
  - Why isn't composition an issue?



Go implementation details

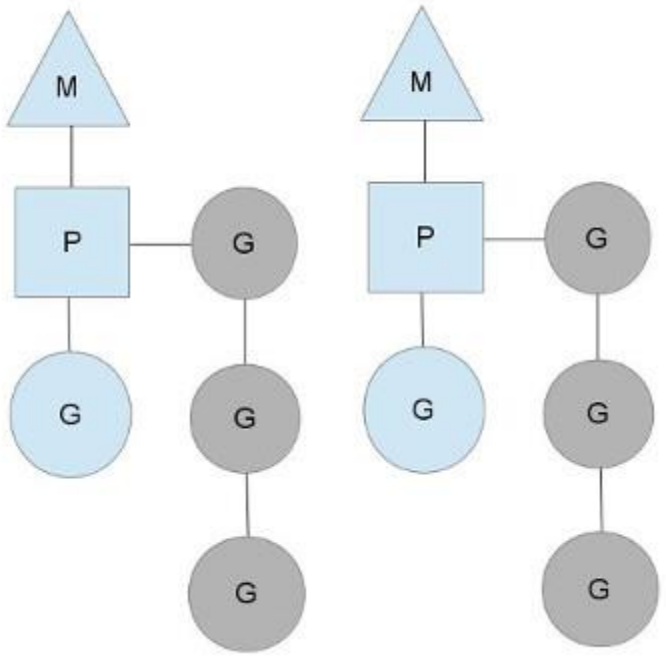


Go implementation details



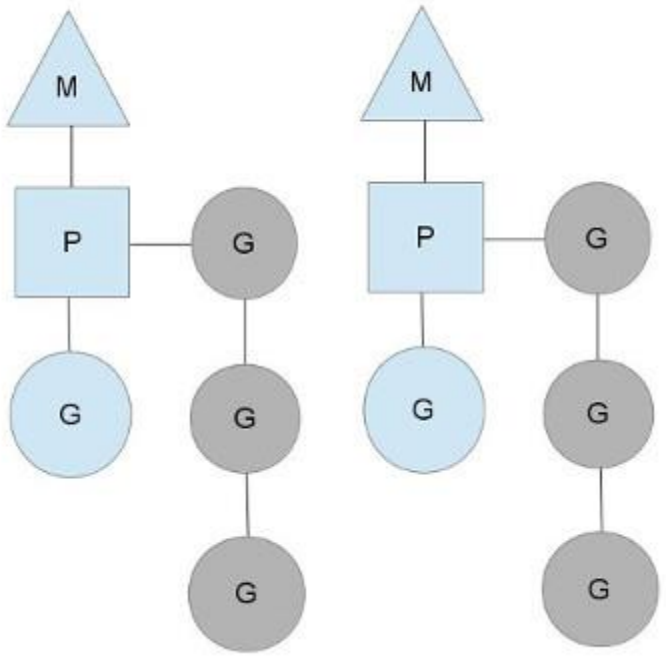
- M = “machine” → OS thread

Go implementation details



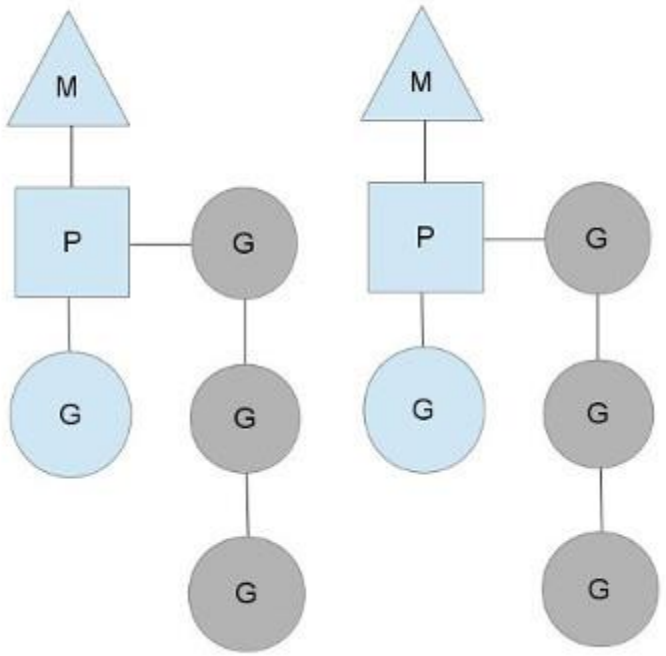
- M = “machine” → OS thread
- P = (processing) context

Go implementation details



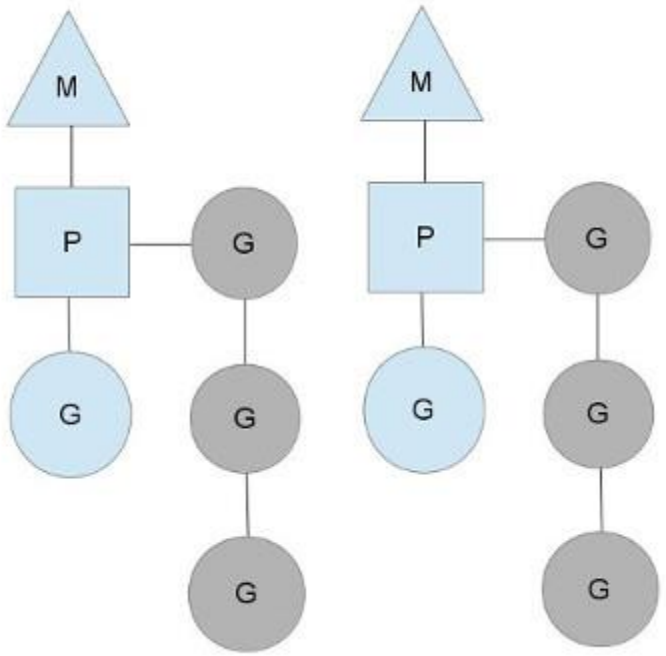
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

Go implementation details



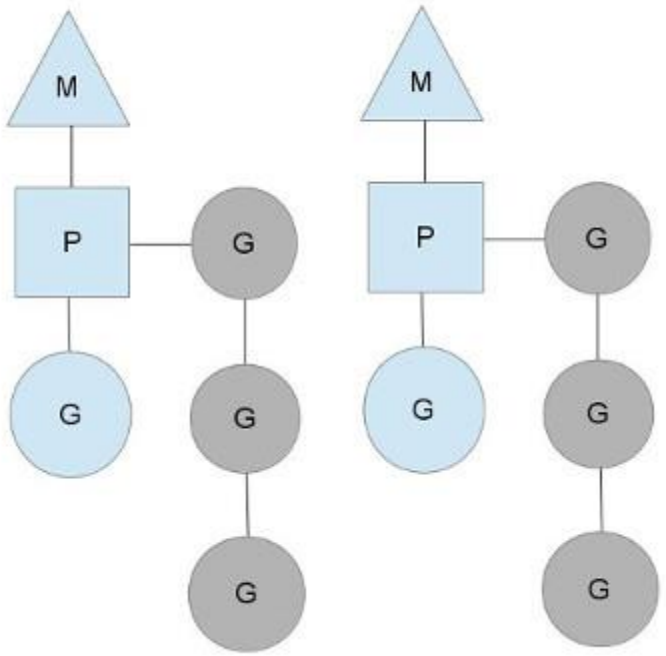
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines

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

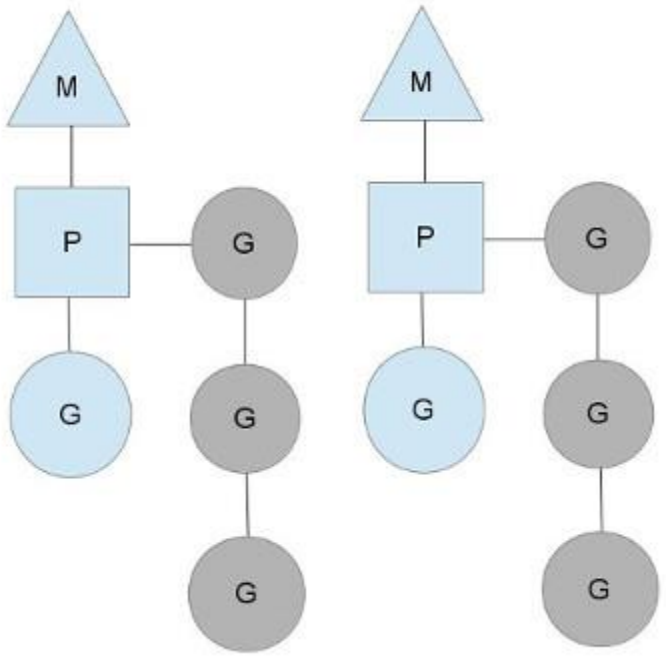
Go implementation details



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

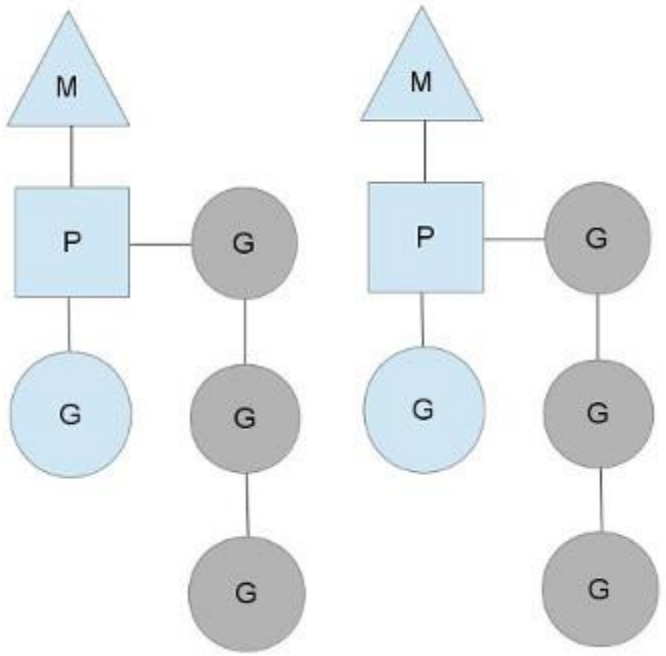
```
struct  G
{
  byte*   stackguard; // stack guard information
  byte*   stackbase;  // base of stack
  byte*   stack0;     // current stack pointer
  byte*   entry;      // initial function
  void*    param;     // passed parameter on wakeup
  int16   status;     // status
  int32   goid;       // 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

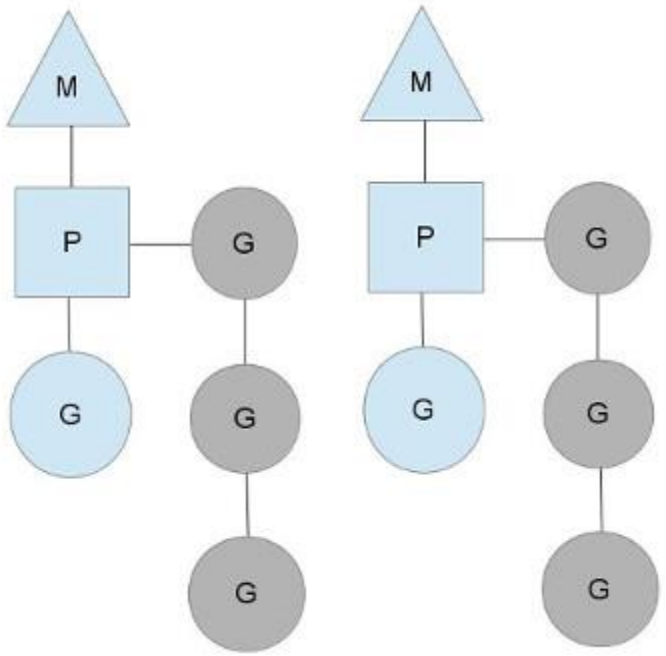
Go implementation details



- 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
   ...
};
```

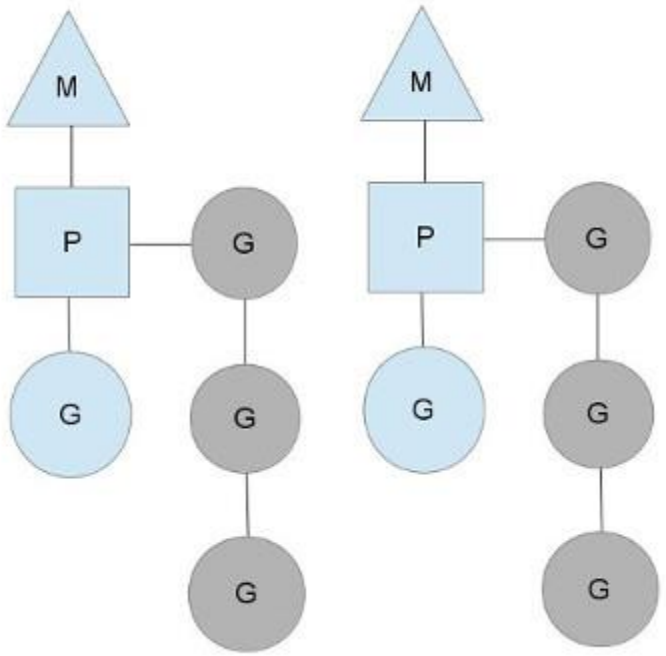
Go implementation details



- 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
  ...
};
```

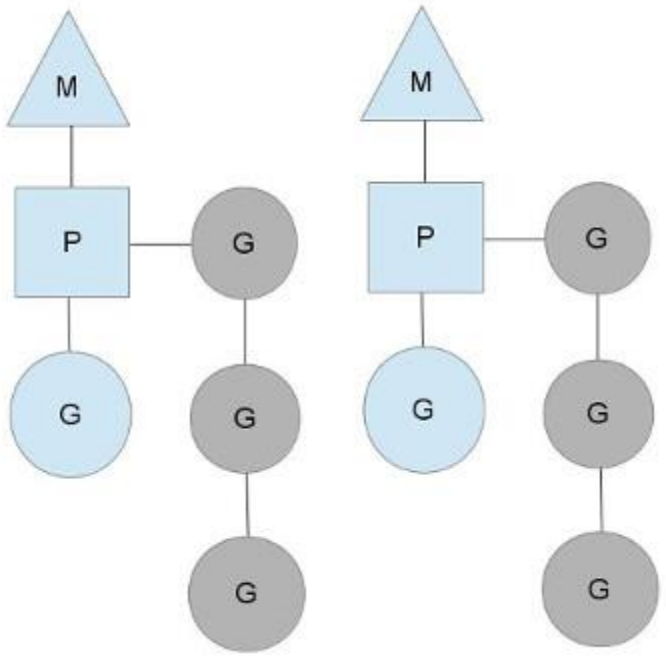
Go implementation details



- 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 reade
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millise
    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

```
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
	chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
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")
	}

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

	// Fast path: check for failed non-blocking operation without acquiring the lock.
	//
	// After observing that the channel is not closed, we observe that the channel is
	// not ready for sending. Each of these observations is a single word-sized read
	// (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
	// Because a closed channel cannot transition from 'ready for sending' to
	// 'not ready for sending', even if the channel is closed between the two observations,
	// they imply a moment between the two when the channel was both not yet closed
	// and not ready for sending. We behave as if we observed the channel at that moment,
	// and report that the send cannot proceed.
	//
	// It is okay if the reads are reordered here: if we observe that the channel is not
	// ready for sending and then observe that it is not closed, that implies that the
	// channel wasn't closed during the first observation.
	if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
		(c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
		return false
	}

	var t0 int64
	if blockprofilerate > 0 {
		t0 = cputicks()
	}

	lock(&c.lock)

	if c.closed != 0 {
		unlock(&c.lock)
		panic(plainError("send on closed channel"))
	}

	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)
```

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")
        }

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

```
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
        chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
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")
        }

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

        // Fast path: check for failed non-blocking operation without acquiring the lock.
        //
        // After observing that the channel is not closed, we observe that the channel is
        // not ready for sending. Each of these observations is a single word-sized read
        // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
        // Because a closed channel cannot transition from 'ready for sending' to
        // 'not ready for sending', even if the channel is closed between the two observations,
        // they imply a moment between the two when the channel was both not yet closed
        // and not ready for sending. We behave as if we observed the channel at that moment,
        // and report that the send cannot proceed.
        //
        // It is okay if the reads are reordered here: if we observe that the channel is not
        // ready for sending and then observe that it is not closed, that implies that the
        // channel wasn't closed during the first observation.
        if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
                (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
                return false
        }

        var t0 int64
        if blockprofilerate > 0 {
                t0 = cputicks()
        }

        lock(&c.lock)

        if c.closed != 0 {
                unlock(&c.lock)
                panic(plainError("send on closed channel"))
        }

        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)
```

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 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 blockprofilerate > 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)
```

Channel implementation

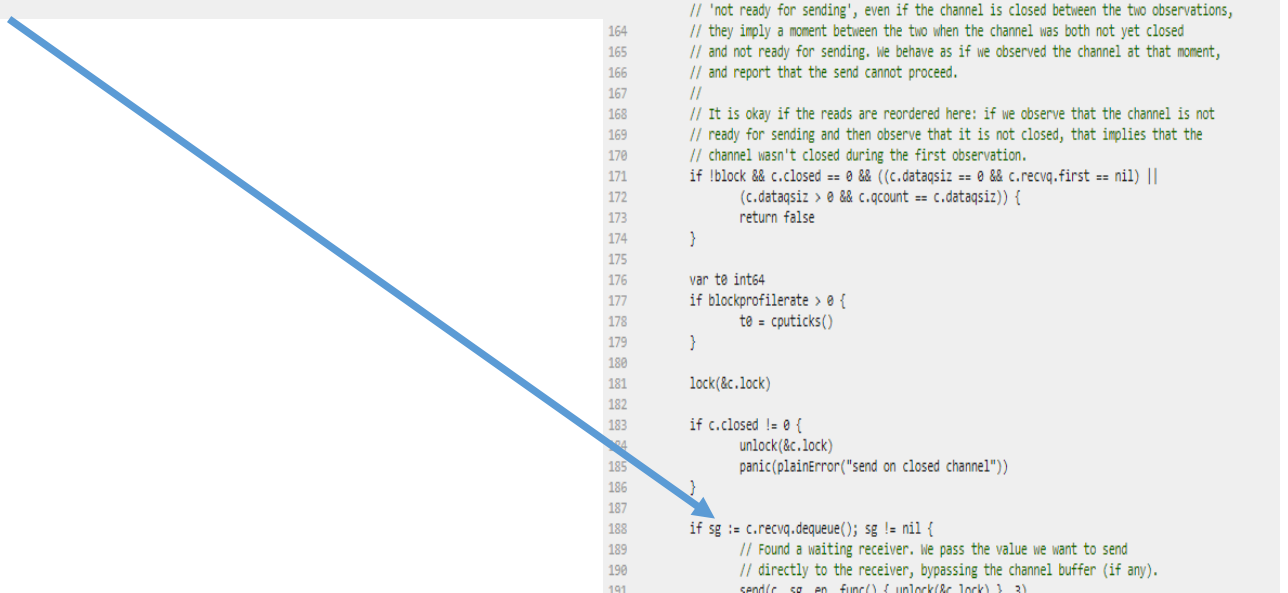
- You can just read it:
  - https://golang.org/src/runtime/chan.go
- Some highlights

```
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
	chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
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")
	}

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

	// Fast path: check for failed non-blocking operation without acquiring the lock.
	//
	// After observing that the channel is not closed, we observe that the channel is
	// not ready for sending. Each of these observations is a single word-sized read
	// (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
	// Because a closed channel cannot transition from 'ready for sending' to
	// 'not ready for sending', even if the channel is closed between the two observations,
	// they imply a moment between the two when the channel was both not yet closed
	// and not ready for sending. We behave as if we observed the channel at that moment,
	// and report that the send cannot proceed.
	//
	// It is okay if the reads are reordered here: if we observe that the channel is not
	// ready for sending and then observe that it is not closed, that implies that the
	// channel wasn't closed during the first observation.
	if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
		(c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
		return false
	}

	var t0 int64
	if blockprofilerate > 0 {
		t0 = cputicks()
	}

	lock(&c.lock)

	if c.closed != 0 {
		unlock(&c.lock)
		panic(plainError("send on closed channel"))
	}

	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)
```

Channel implementation

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

```
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 {
                print("chansend: chan=", c, "\n")
            }

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

            // Fast path: check for failed non-blocking operation without acquiring the lock.
            //
            // After observing that the channel is not closed, we observe that the channel is
            // not ready for sending. Each of these observations is a single word-sized read
            // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
            // Because a closed channel cannot transition from 'ready for sending' to
            // '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 blockprofilerate > 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)
```

Channel implementation

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

```
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
	chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
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")
	}

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

	// Fast path: check for failed non-blocking operation without acquiring the lock.
	//
	// After observing that the channel is not closed, we observe that the channel is
	// not ready for sending. Each of these observations is a single word-sized read
	// (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
	// Because a closed channel cannot transition from 'ready for sending' to
	// 'not ready for sending', even if the channel is closed between the two observations,
	// they imply a moment between the two when the channel was both not yet closed
	// and not ready for sending. We behave as if we observed the channel at that moment,
	// and report that the send cannot proceed.
	//
	// It is okay if the reads are reordered here: if we observe that the channel is not
	// ready for sending and then observe that it is not closed, that implies that the
	// channel wasn't closed during the first observation.
	if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
		(c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
		return false
	}

	var t0 int64
	if blockprofilerate > 0 {
		t0 = cputicks()
	}

	lock(&c.lock)

	if c.closed != 0 {
		unlock(&c.lock)
		panic(plainError("send on closed channel"))
	}

	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)
```

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 }
```

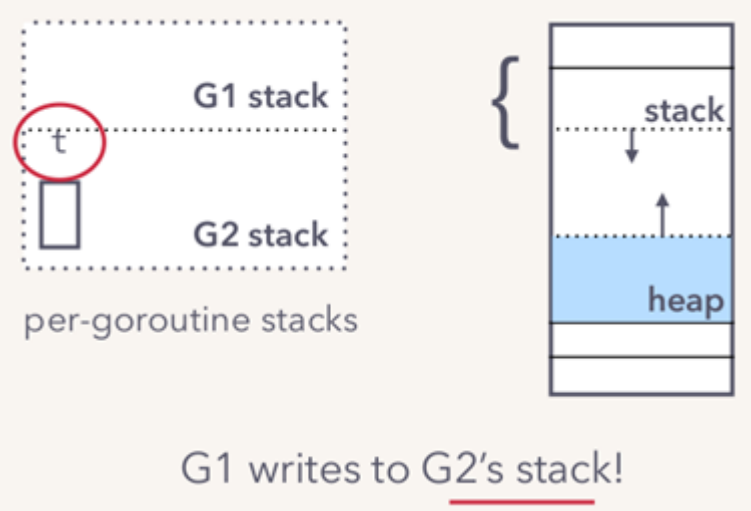
Channel implementation

- 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 }
```

Channel implementation

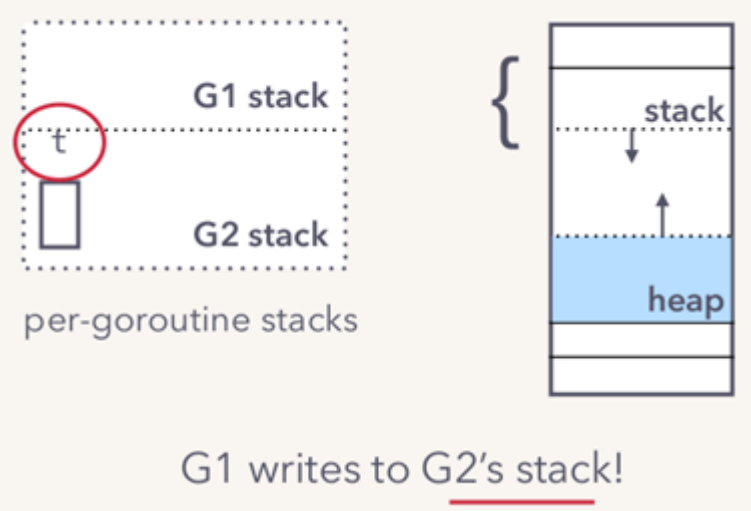
- You can just read it:
  - https://golang.org/s
- Some highlights




```
// Sends and receives on unbuffered or empty-buffered channels are the
// only operations where one running goroutine writes to the stack of
// another running goroutine. The GC assumes that stack writes only
// happen when the goroutine is running and are only done by that
// goroutine. Using a write barrier is sufficient to make up for
// violating that assumption, but the write barrier has to work.
// typedmemmove will call bulkBarrierPreWrite, but the target bytes
// are not in the heap, so that will not help. We arrange to call
// memmove and typeBitsBulkBarrier instead.

func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
        // src is on our stack, dst is a slot on another stack.

        // Once we read sg.elem out of sg, it will no longer
        // be updated if the destination's stack gets copied (shrunk).
        // So make sure that no preemption points can happen between read & use.
        dst := sg.elem
        typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
        memmove(dst, src, t.size)
}
```

Channel implementation

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



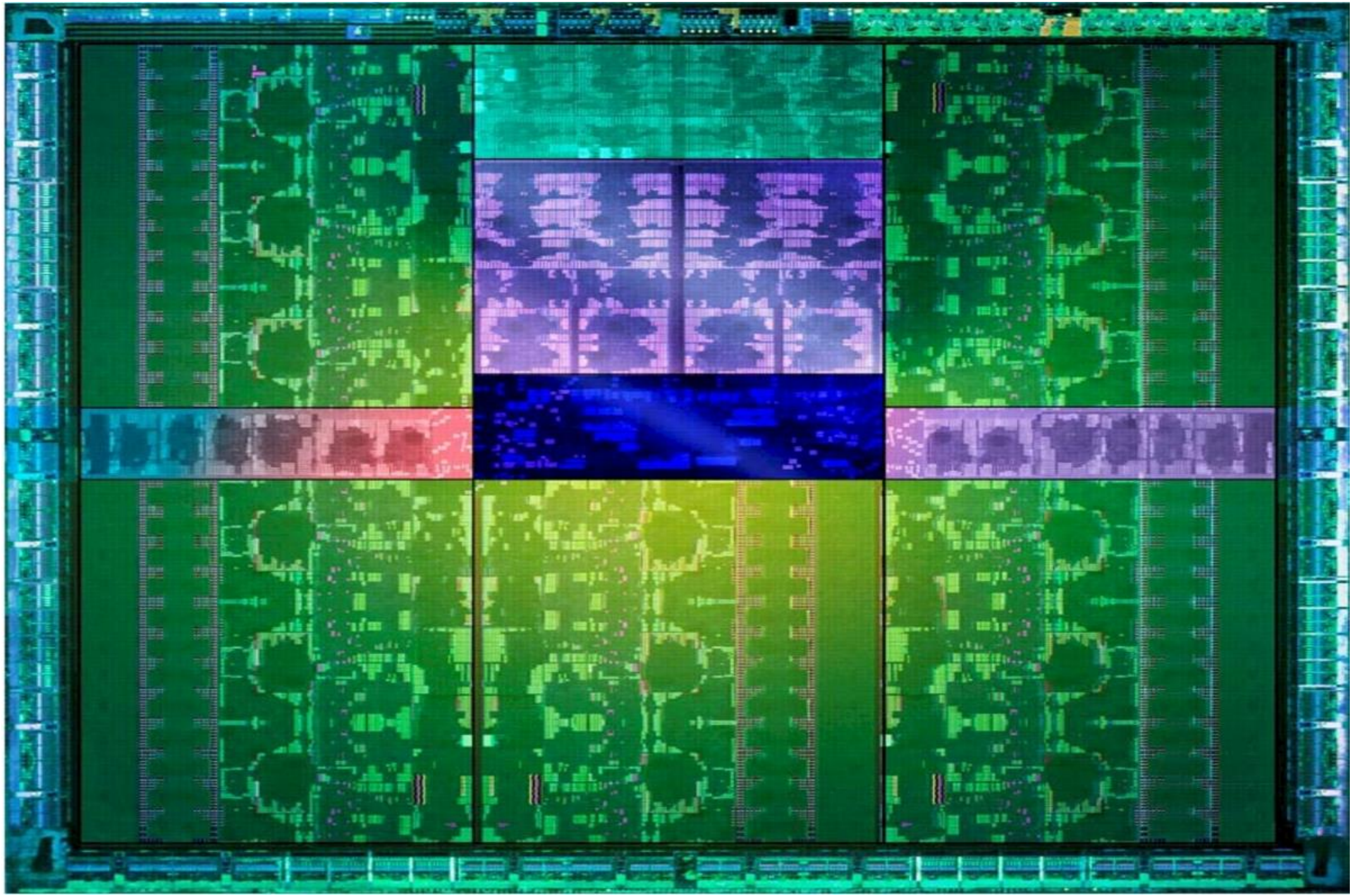

```
// Sends and receives on unbuffered or empty-buffered channels are the
// only operations where one running goroutine writes to the stack of
// another running goroutine. The GC assumes that stack writes only
// happen when the goroutine is running and are only done by that
// goroutine. Using a write barrier is sufficient to make up for
// violating that assumption, but the write barrier has to work.
// typedmemmove will call bulkBarrierPreWrite, but the target bytes
// are not in the heap, so that will not help. We arrange to call
// memmove and typeBitsBulkBarrier instead.

func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
        // src is on our stack, dst is a slot on another stack.

        // Once we read sg.elem out of sg, it will no longer
        // be updated if the destination's stack gets copied (shrunk).
        // So make sure that no preemption points can happen between read & use.
        dst := sg.elem
        typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
        memmove(dst, src, t.size)
}
```

Transputers did this in hardware in the 90s btw.

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

```
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
	chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
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")
	}

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

	// Fast path: check for failed non-blocking operation without acquiring the lock.
	//
	// After observing that the channel is not closed, we observe that the channel is
	// not ready for sending. Each of these observations is a single word-sized read
	// (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
	// Because a closed channel cannot transition from 'ready for sending' to
	// 'not ready for sending', even if the channel is closed between the two observations,
	// they imply a moment between the two when the channel was both not yet closed
	// and not ready for sending. We behave as if we observed the channel at that moment,
	// and report that the send cannot proceed.
	//
	// It is okay if the reads are reordered here: if we observe that the channel is not
	// ready for sending and then observe that it is not closed, that implies that the
	// channel wasn't closed during the first observation.
	if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
		(c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
		return false
	}

	var t0 int64
	if blockprofilerate > 0 {
		t0 = cputicks()
	}

	lock(&c.lock)

	if c.closed != 0 {
		unlock(&c.lock)
		panic(plainError("send on closed channel"))
	}

	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)
```



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

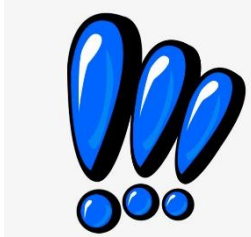
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!

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_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  v
  }

do_n
  inst
  ops
  exe
  access_memory(ops, regs);
  write_back(regs);
}
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(…);
  …
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  v
  }

do_n
  inst
  ops
  exe
  access_memory(ops, regs);
  write_back(regs);
}
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(…);
  …
}
```

```
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:

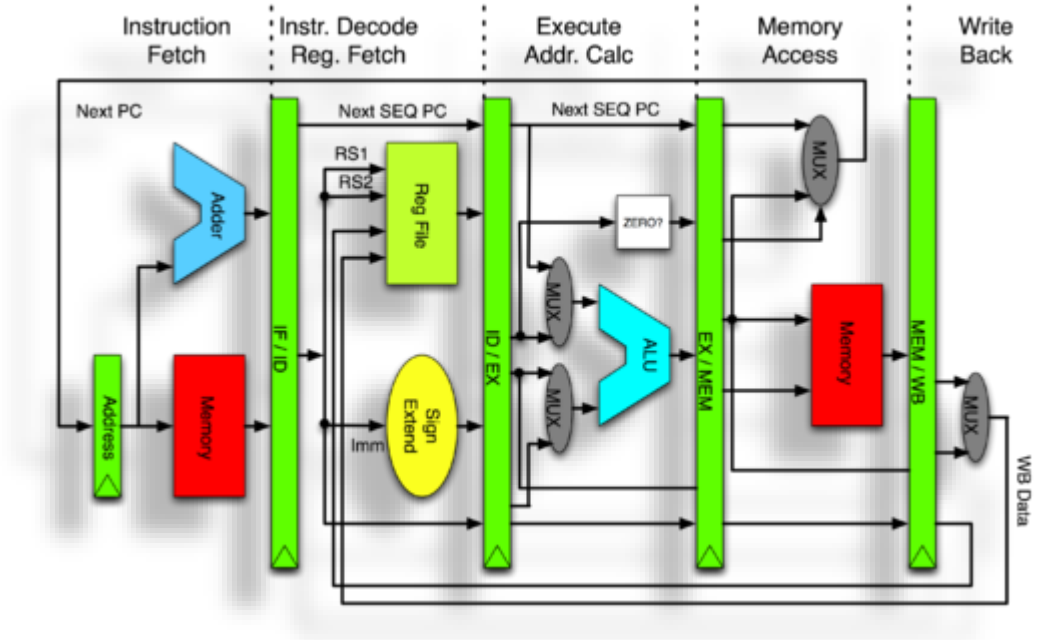
```
main() {
  while(true) {
    do_next_instruction();
}
```

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

Architecture Review: Pipelines

Processor algorithm:

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

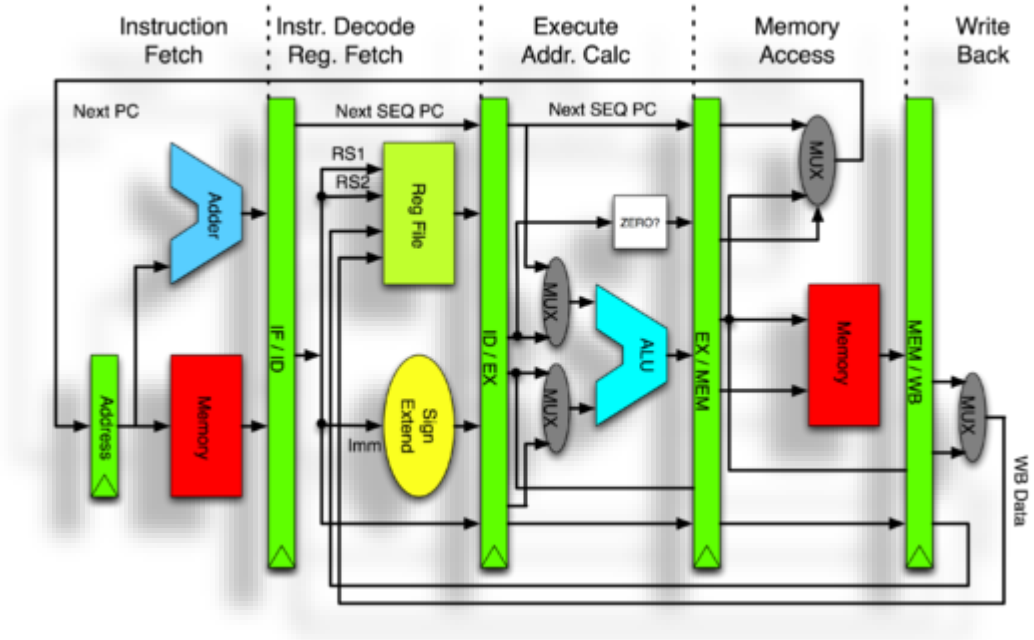



```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

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



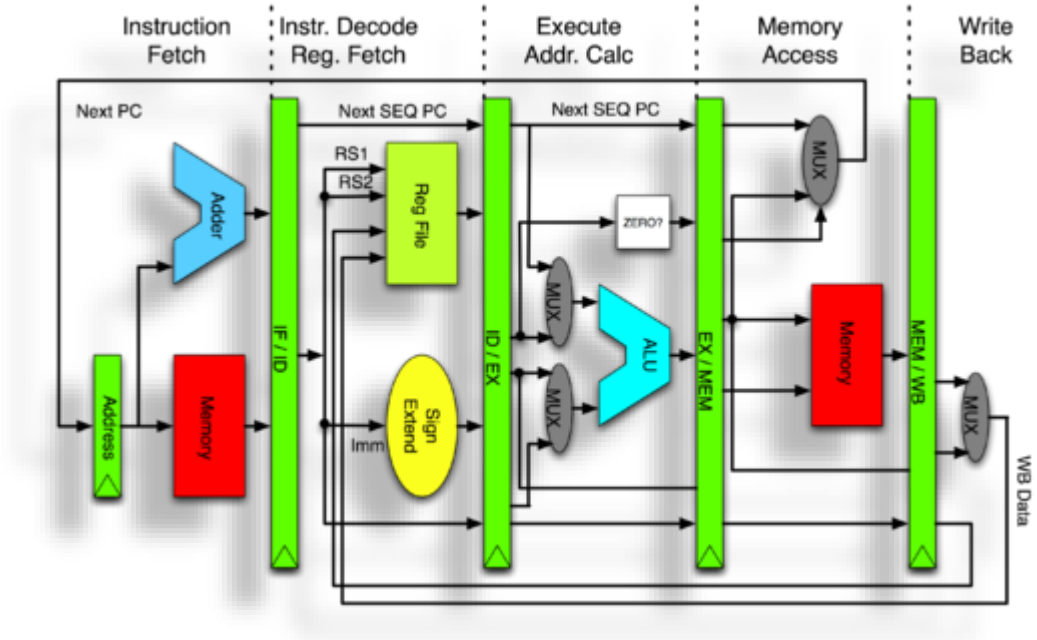

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Instr No.	Pipeline Stage						
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_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

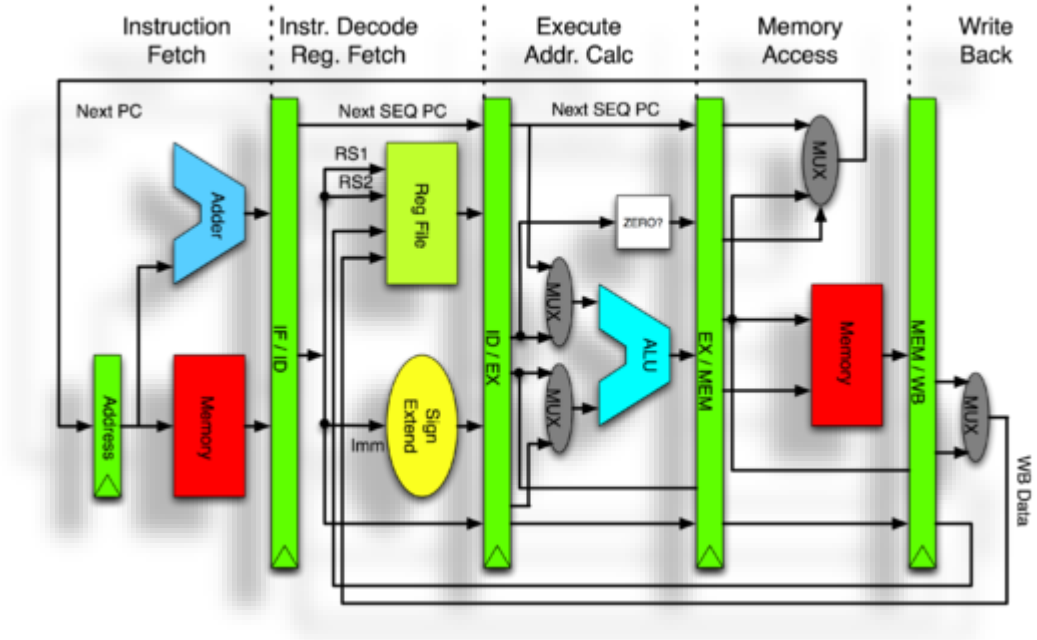
Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

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_addrs(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	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM

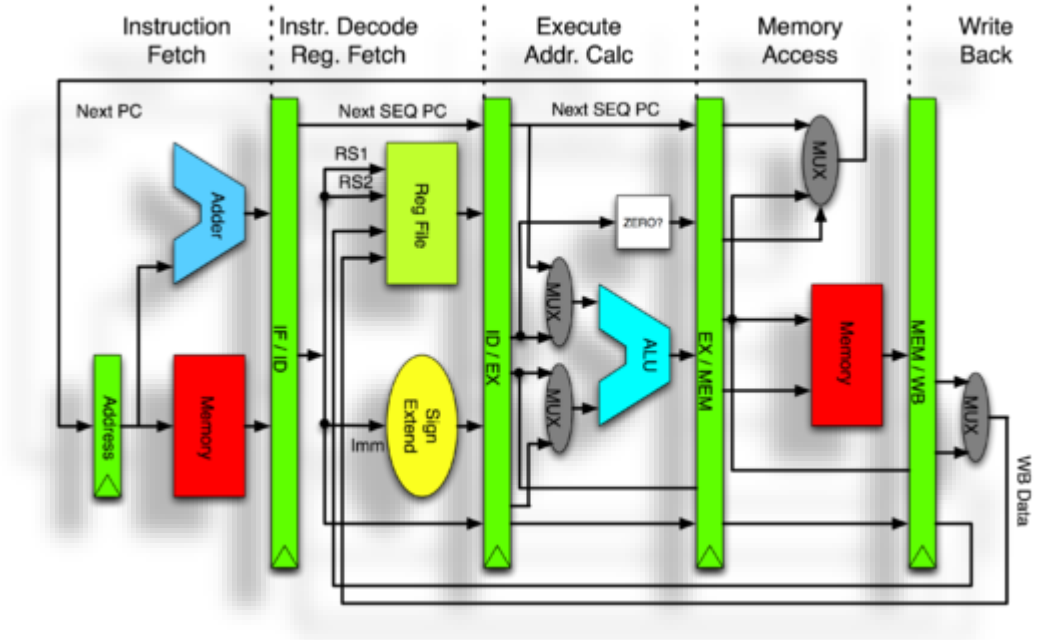
Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

What is the name of this kind of parallelism?

Architecture Review: Pipelines

Processor algorithm:

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



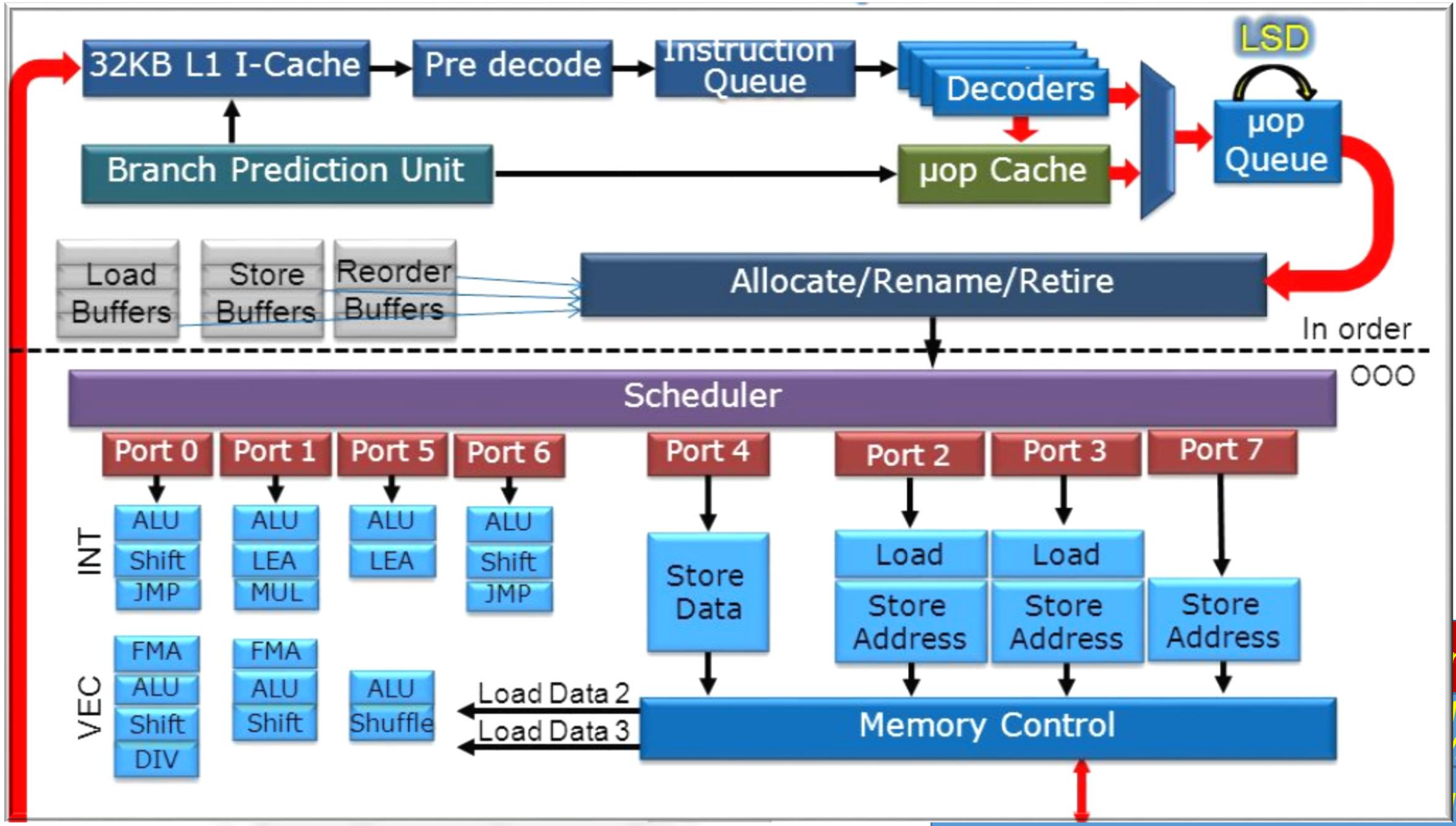

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

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?

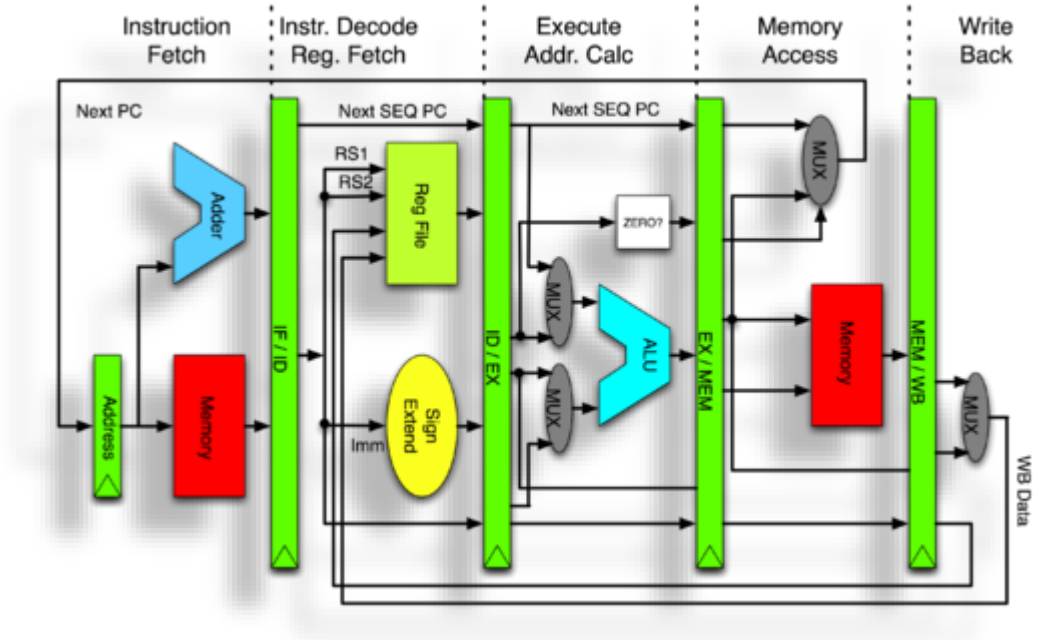
What is the name of this kind of parallelism?


32KB L1 I-Cache
Pre decode
Instruction Queue
Decoders
LSD
µop Queue
Branch Prediction Unit
µop Cache
Load Buffers
Store Buffers
Reorder Buffers
Allocate/Rename/Retire
In order
OOO
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
INT
ALU
Shift
JMP
ALU
LEA
MUL
ALU
LEA
ALU
Shift
JMP
VEC
FMA
ALU
Shift
DIV
FMA
ALU
Shift
ALU
Shuffle
Store Data
Load
Store Address
Load
Store Address
Store Address
Load Data 2
Load Data 3
Memory Control

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						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	

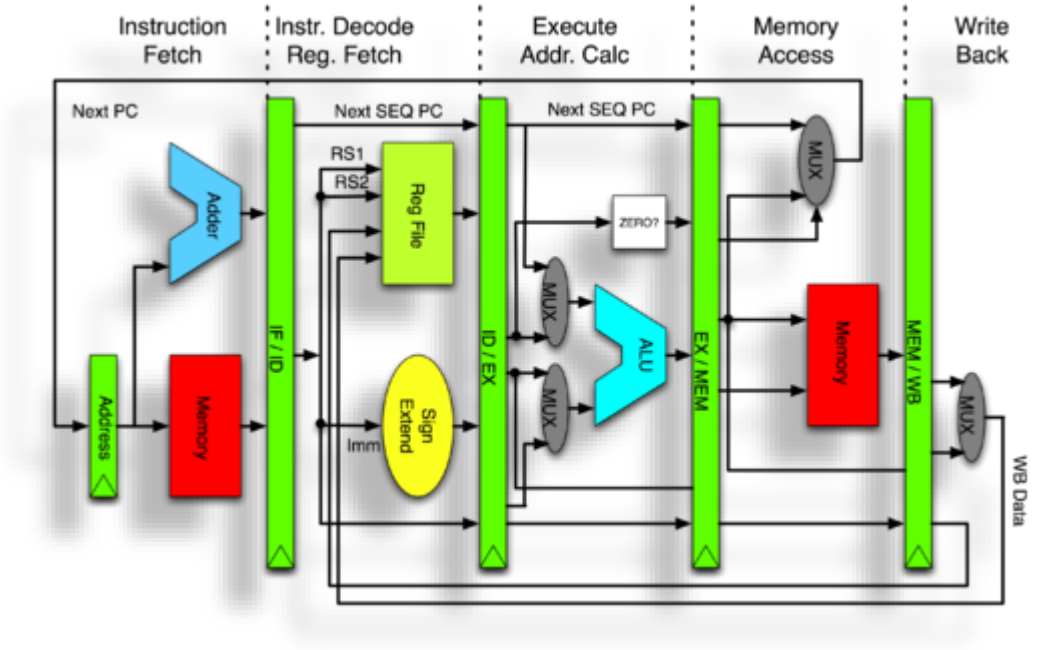
How can we get *more* parallelism?

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

What is the name of this kind of parallelism?

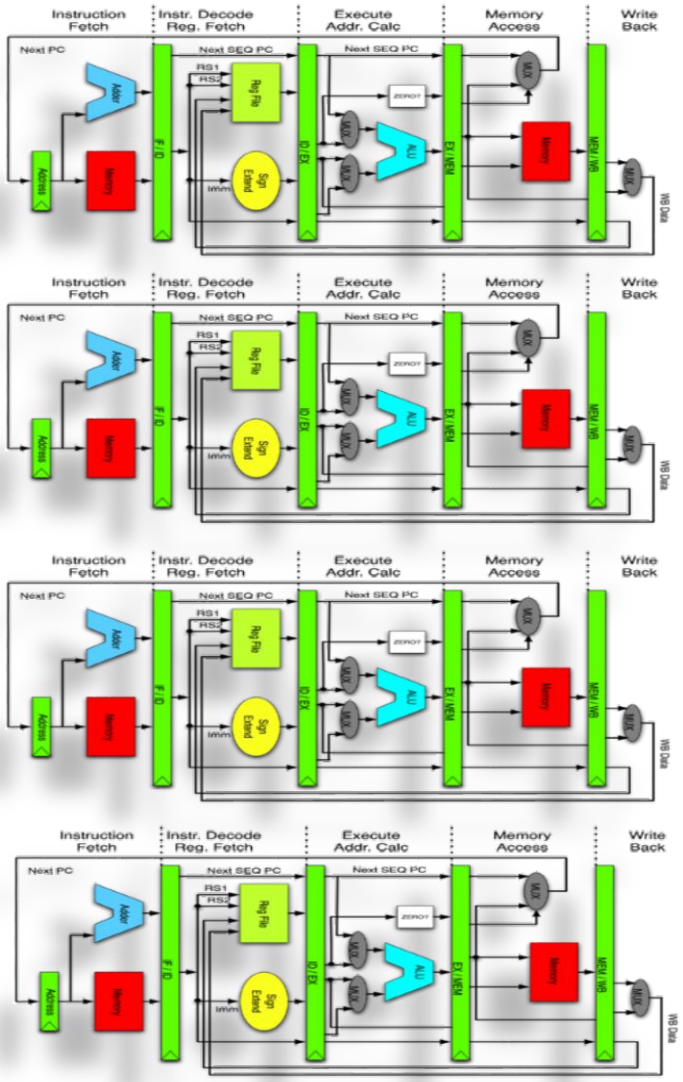
Multi-core/SMPs

Multi-core/SMPs

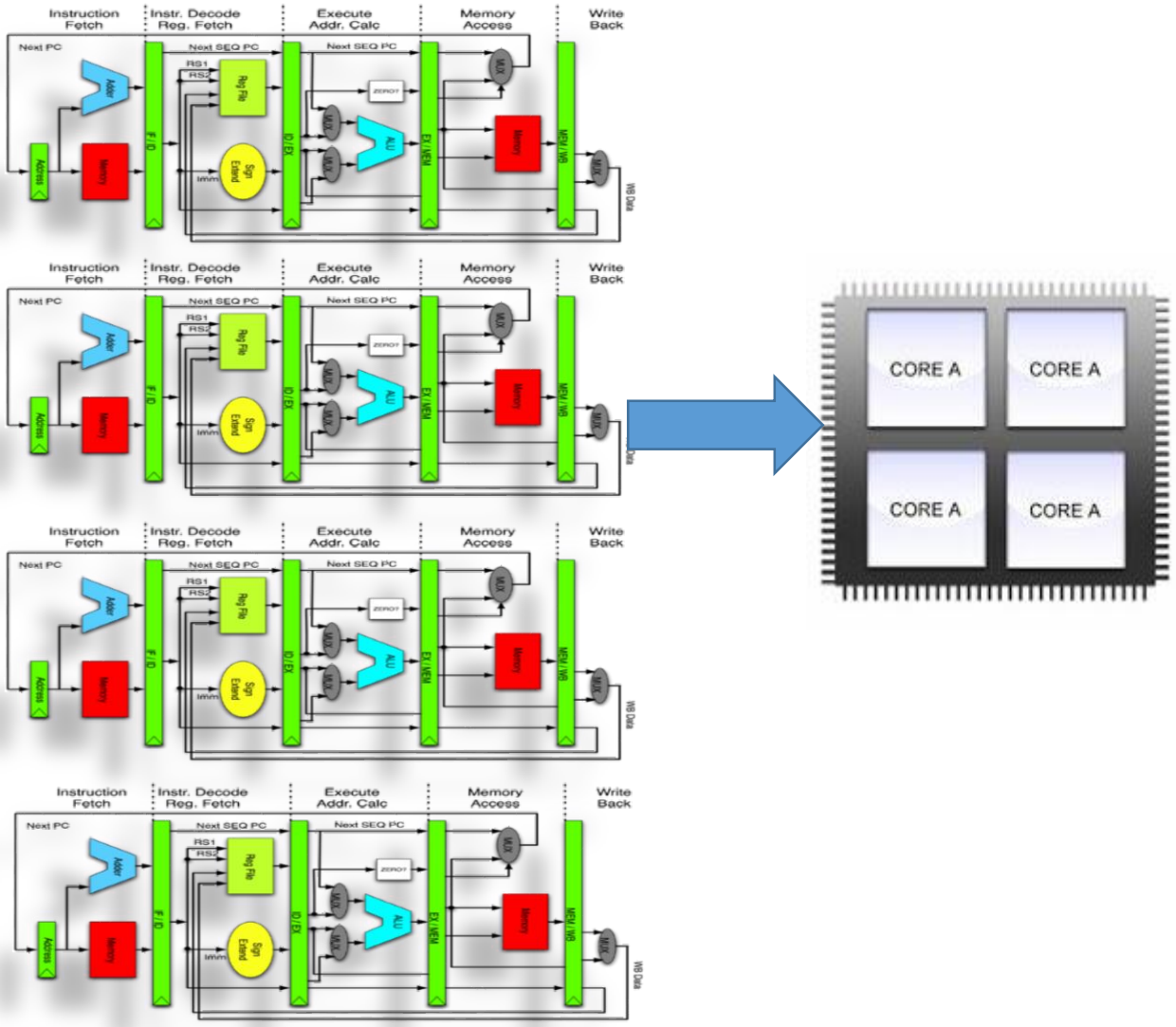


Multi-core/SMPs

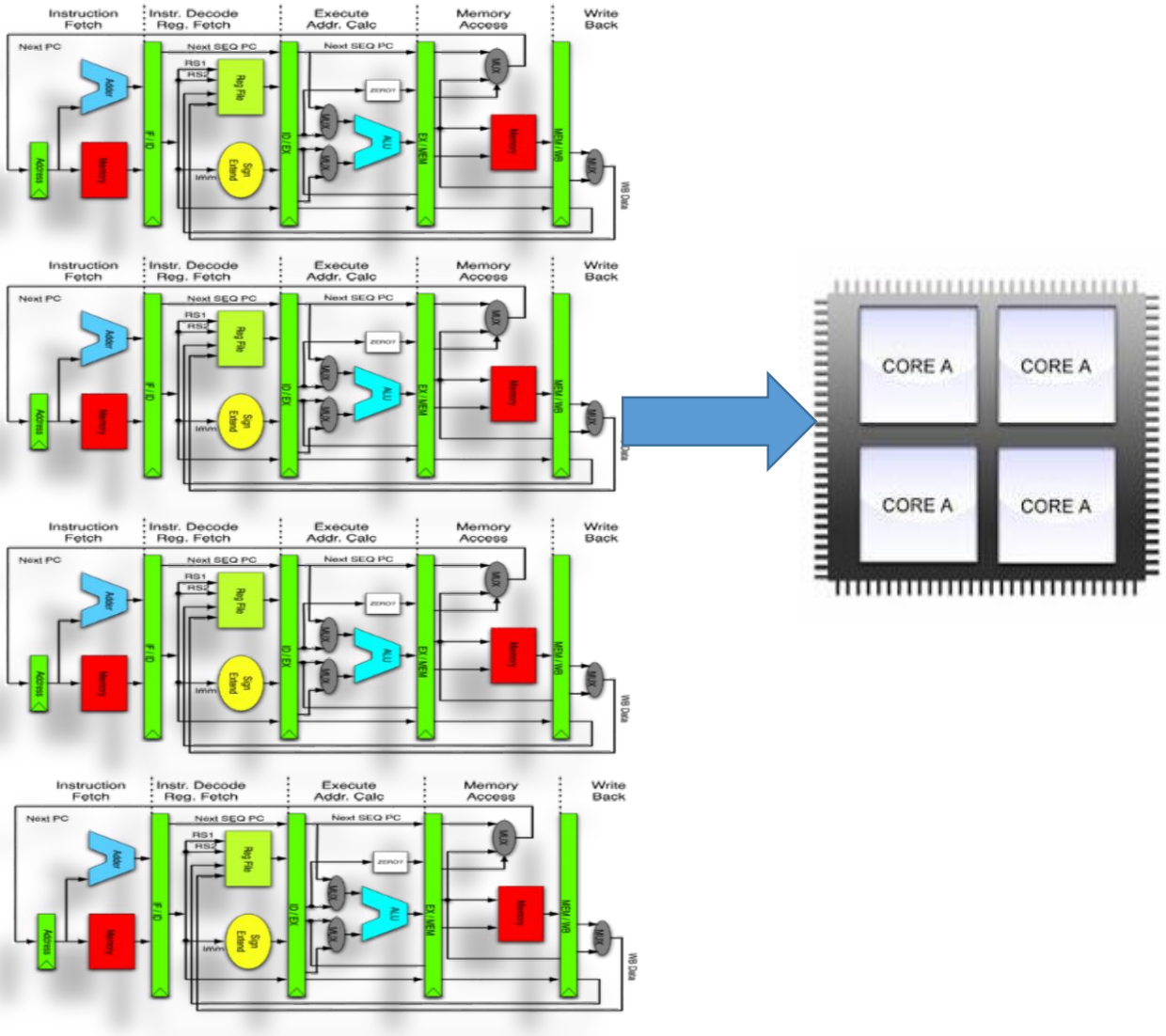
Multi-core/SMPs



Multi-core/SMPs



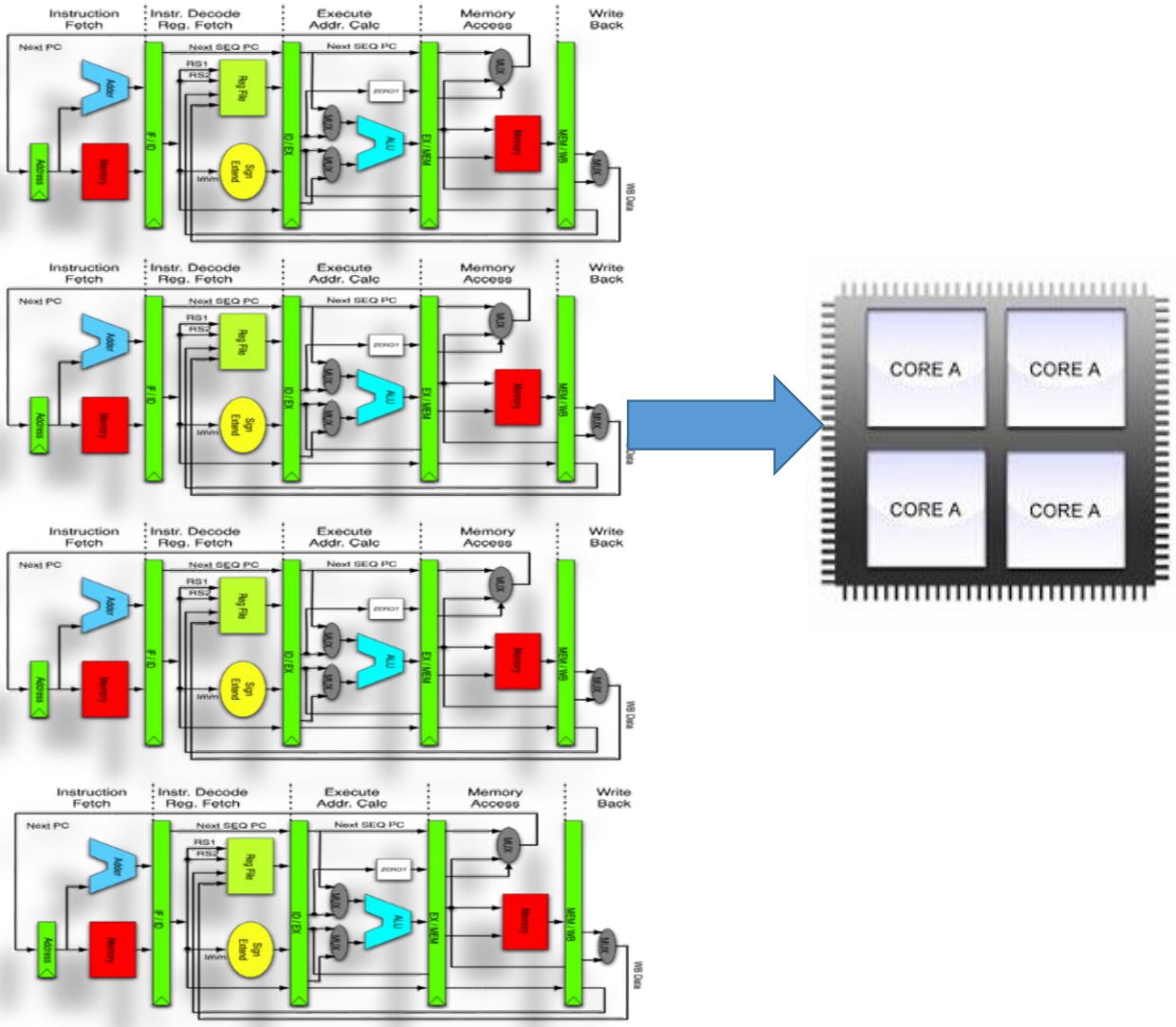
Multi-core/SMPs




```
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!***

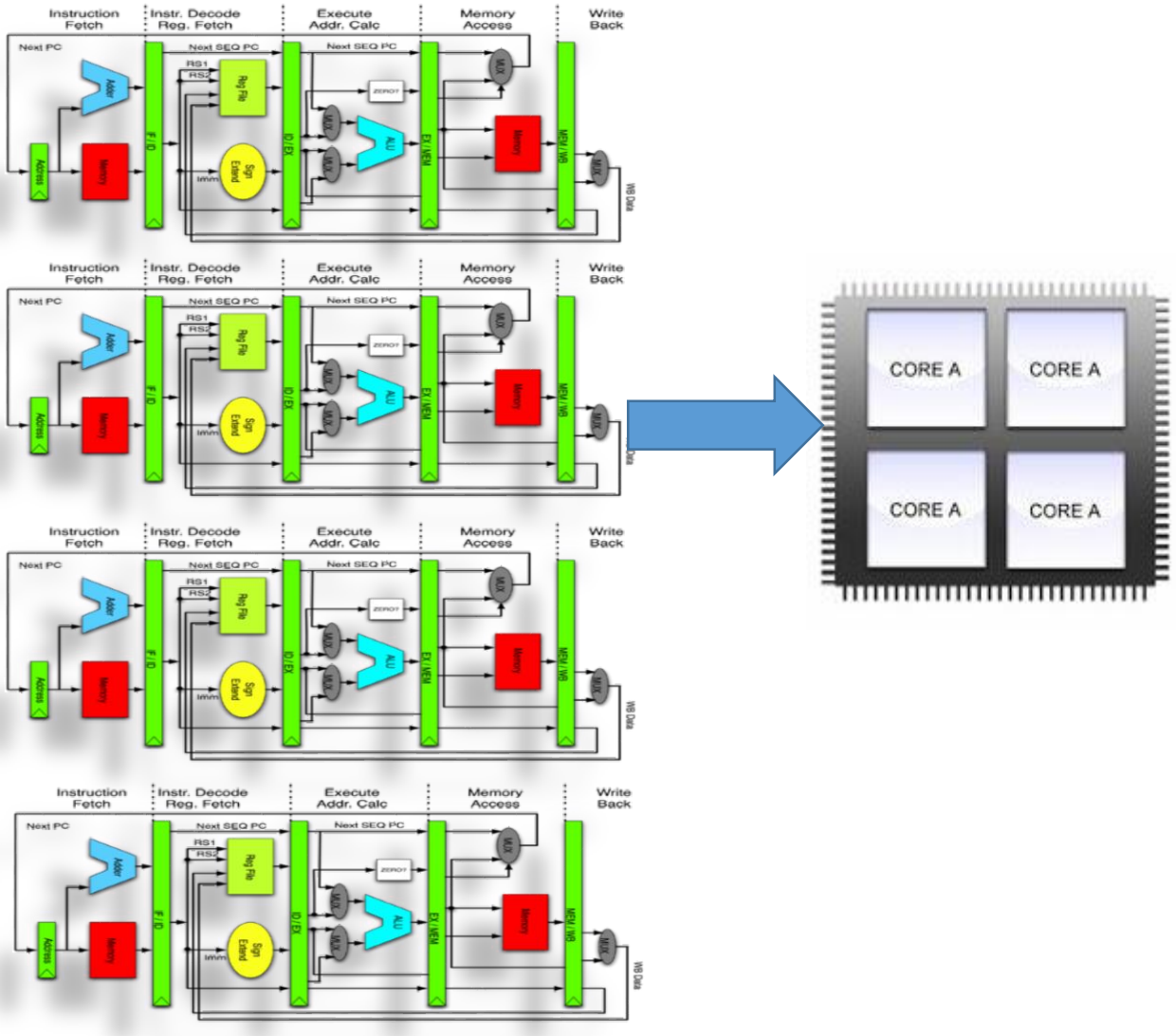
Multi-core/SMPs




```
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!*

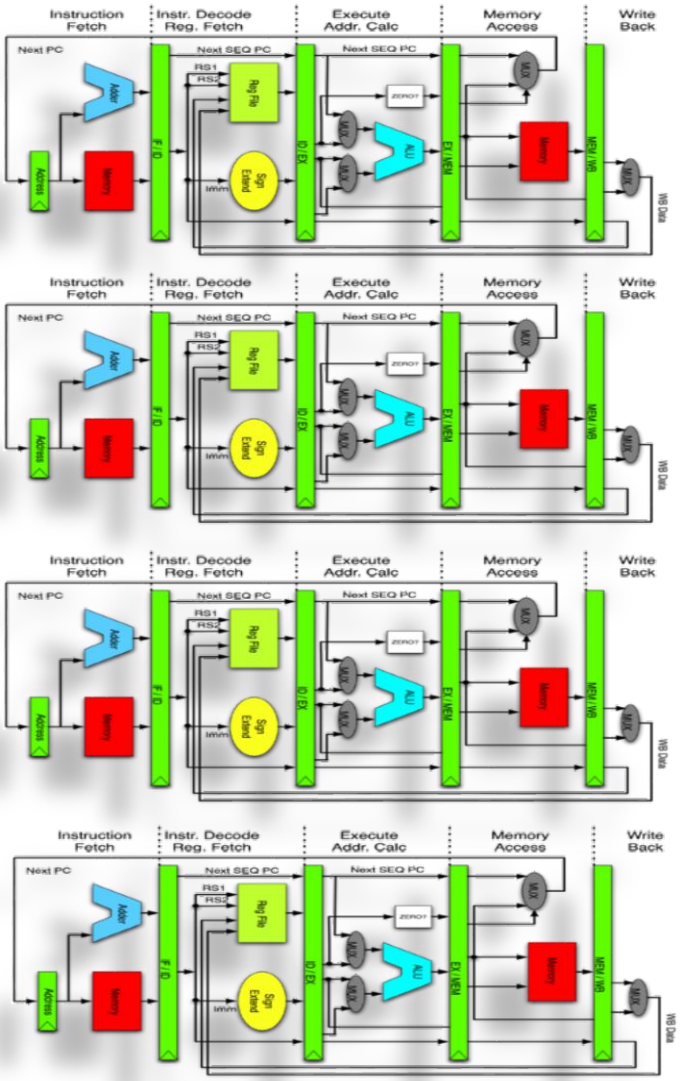
Multi-core/SMPs




```
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);
  }
}
```

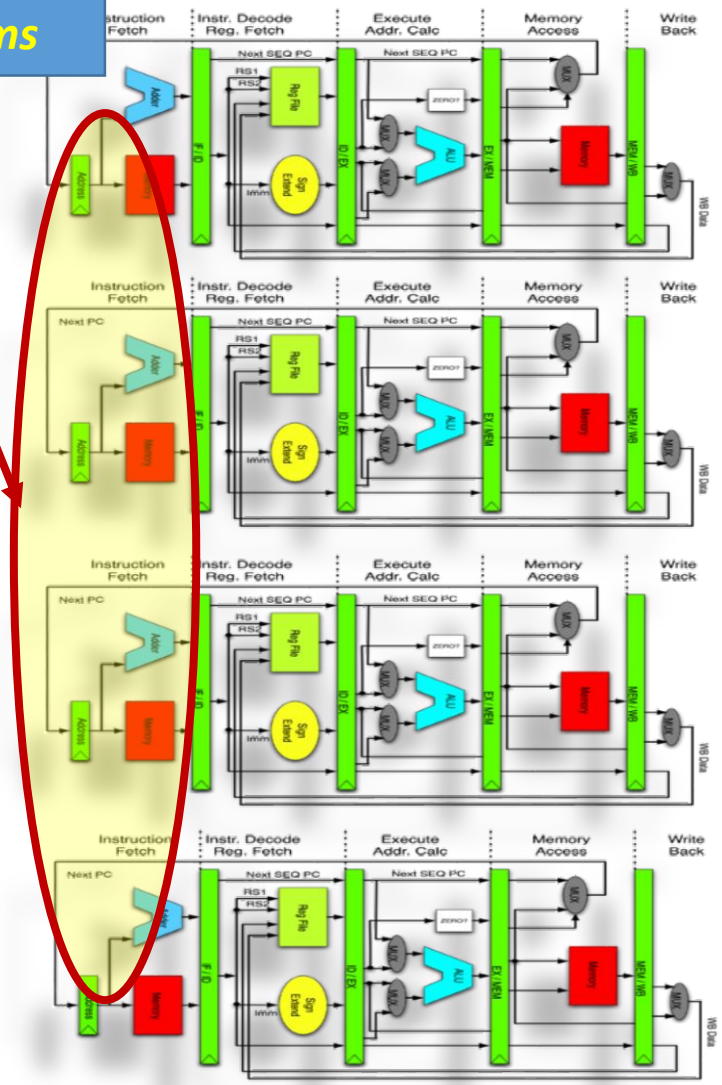
Other techniques extract parallelism here, try to let the machine find parallelism

Superscalar processors



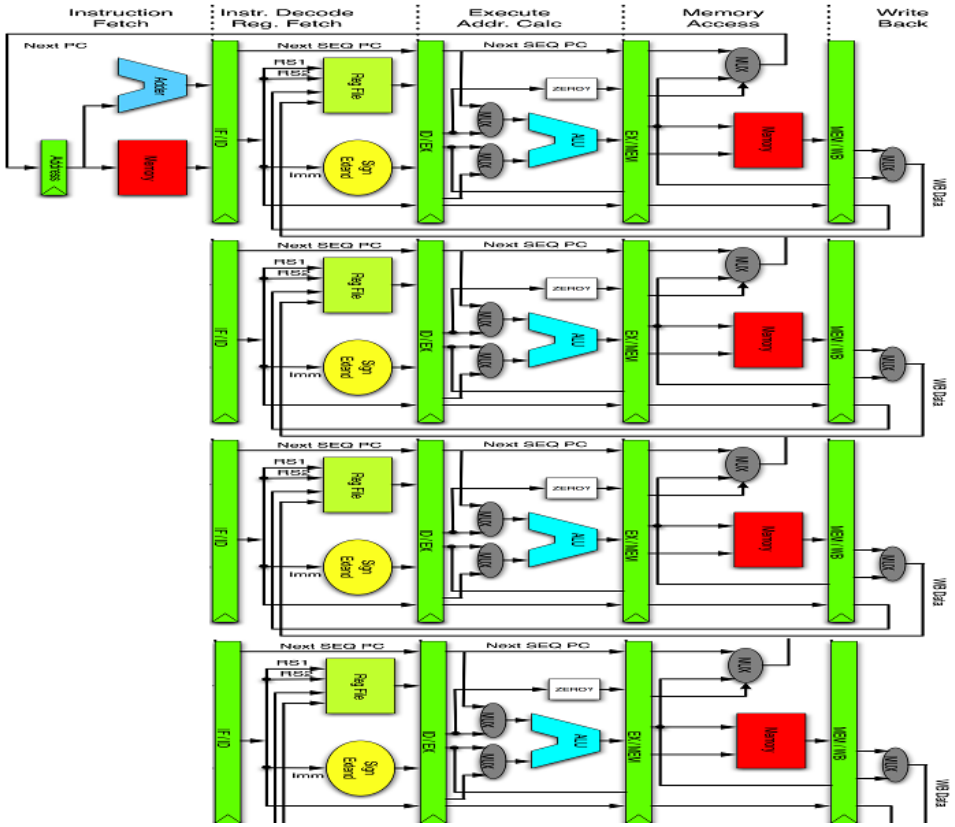
Superscalar processors

Remove extra instruction streams

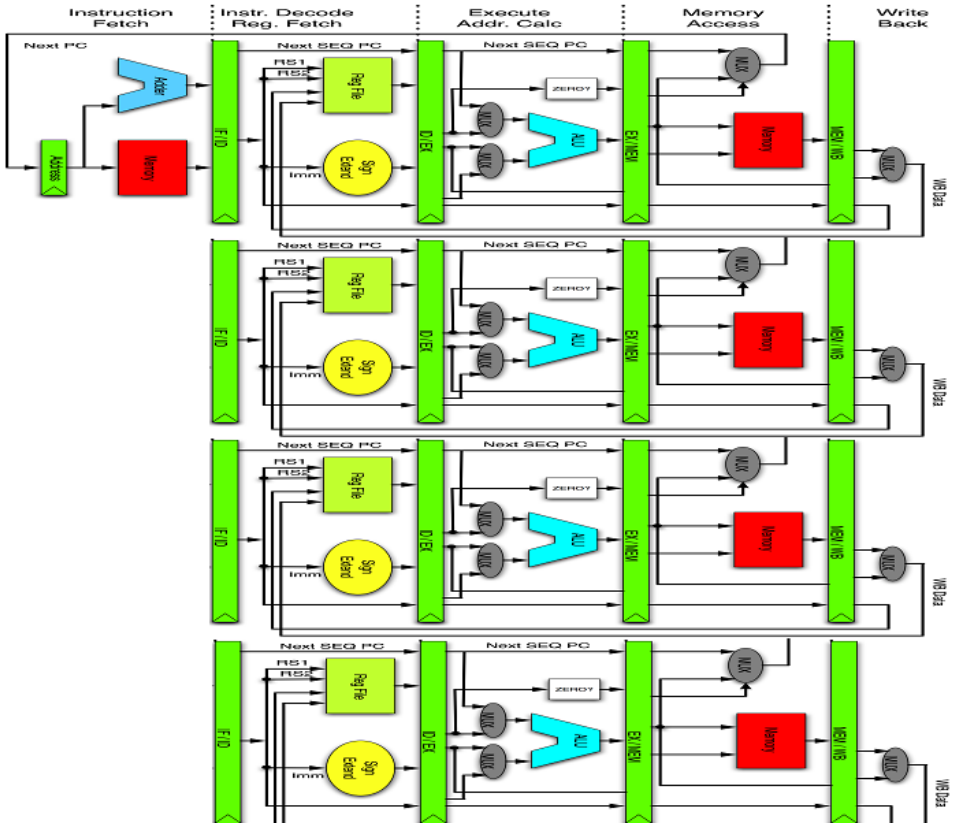


Superscalar processors

Superscalar processors

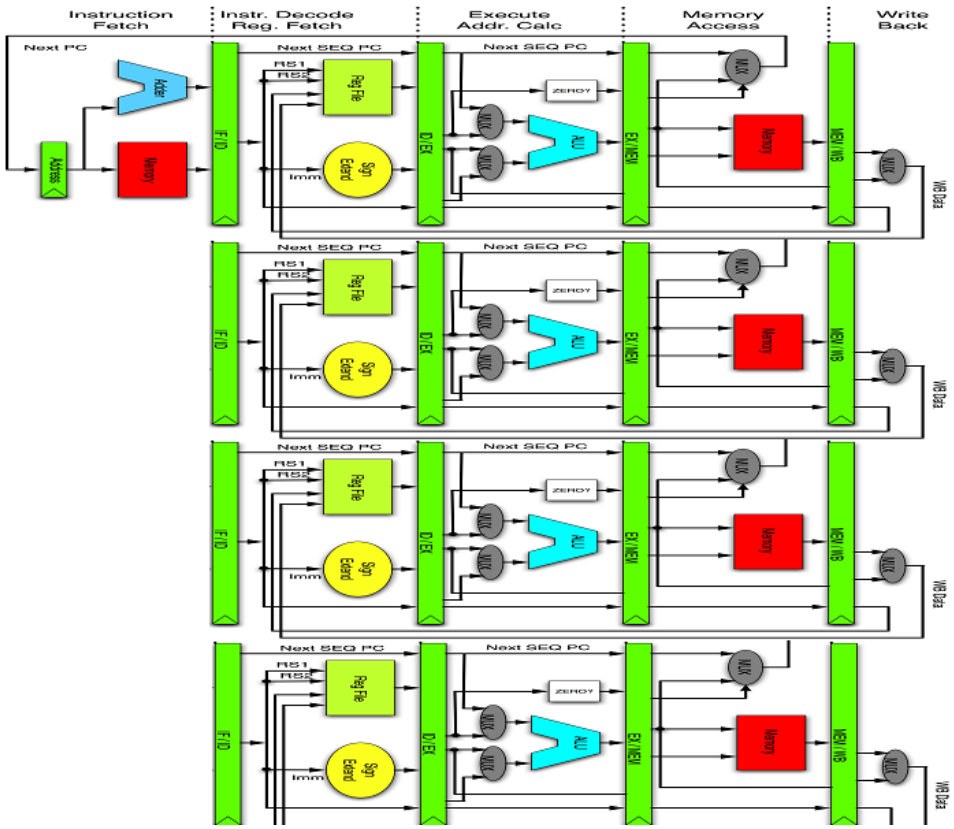


Superscalar processors




```
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);
}
```

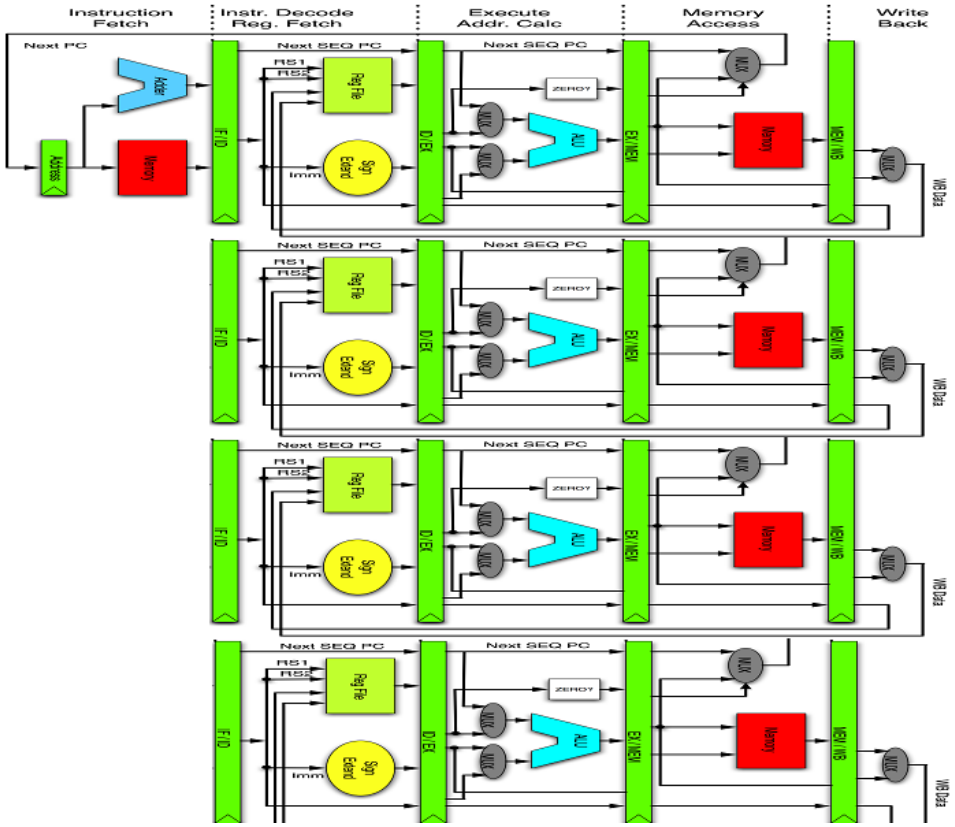
Superscalar processors




```
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

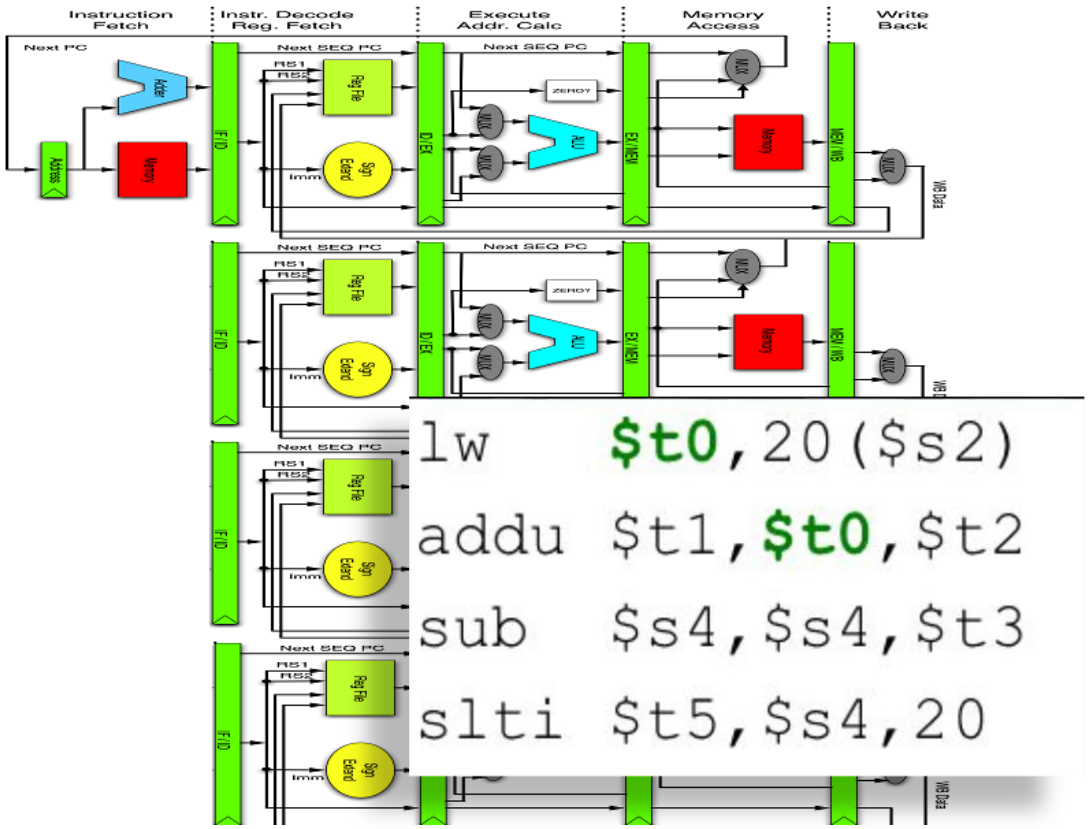



```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

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

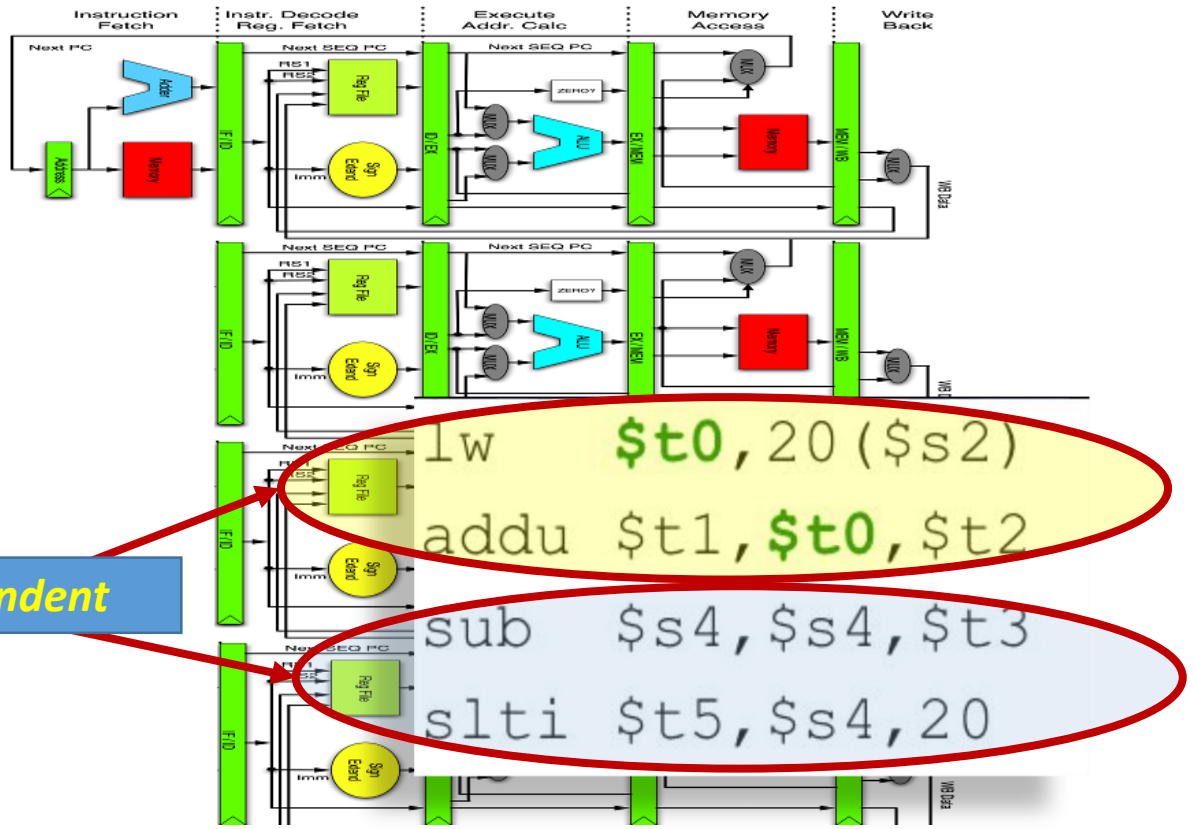



```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

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



independent

```
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);
}
```

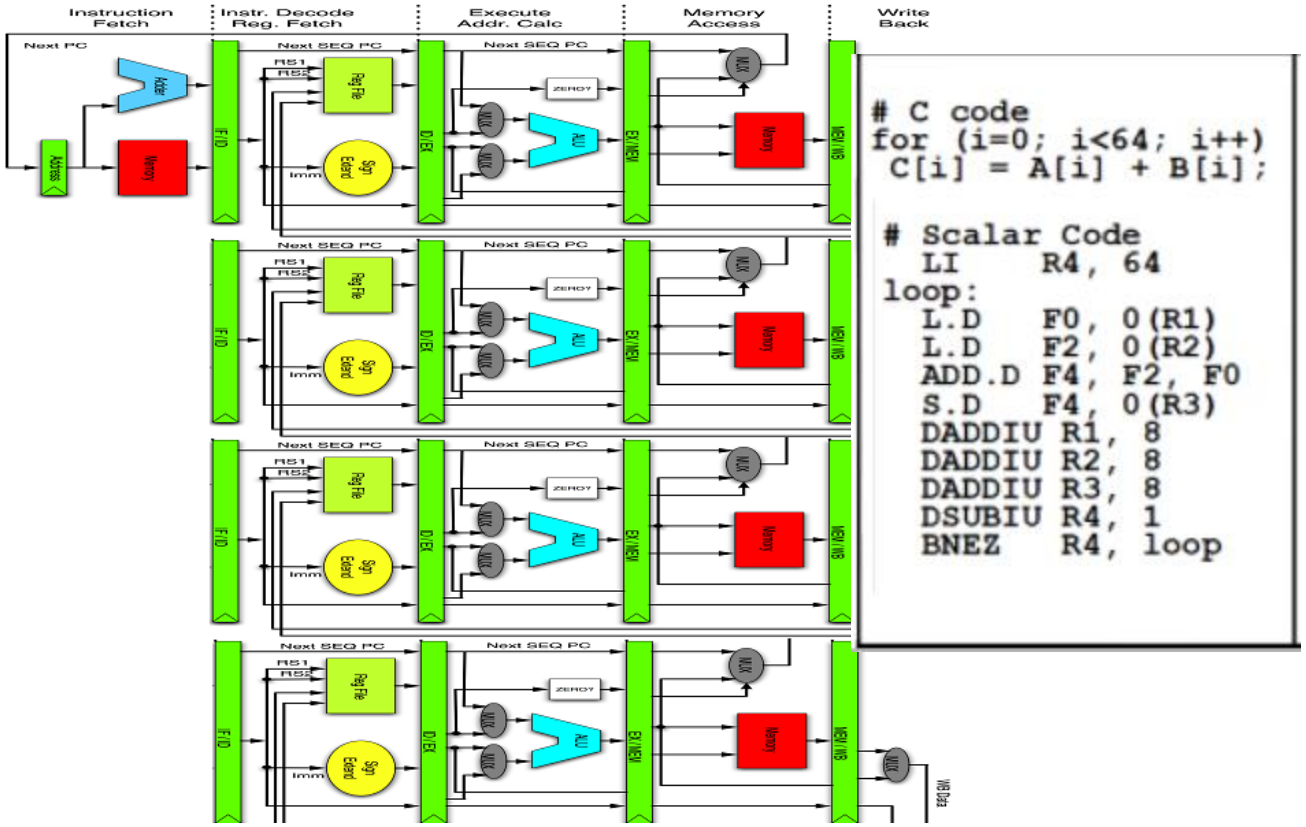
Doesn't look that different does it? Why do it?

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
```

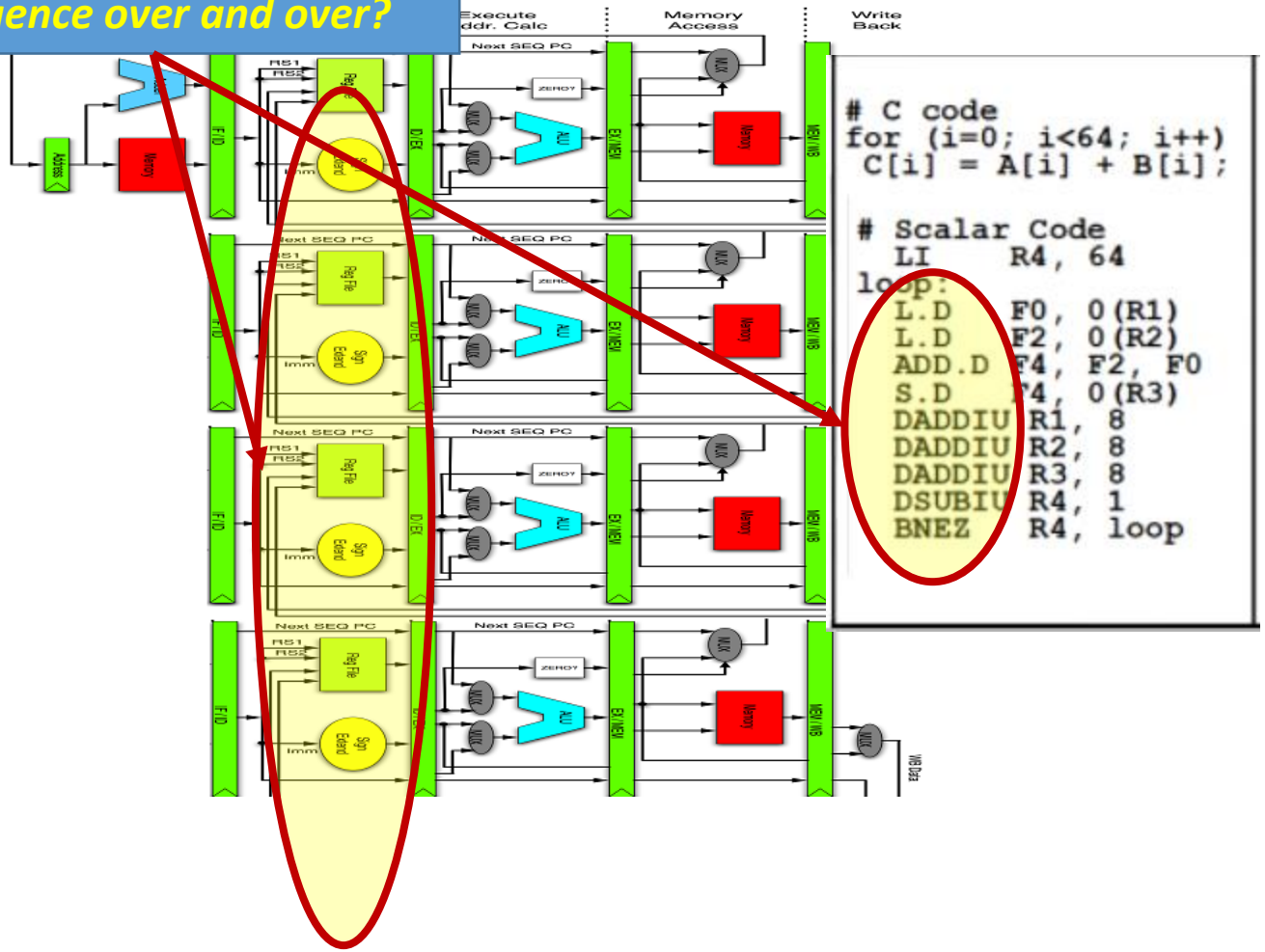
Vector/SIMD processors




```
# 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
```

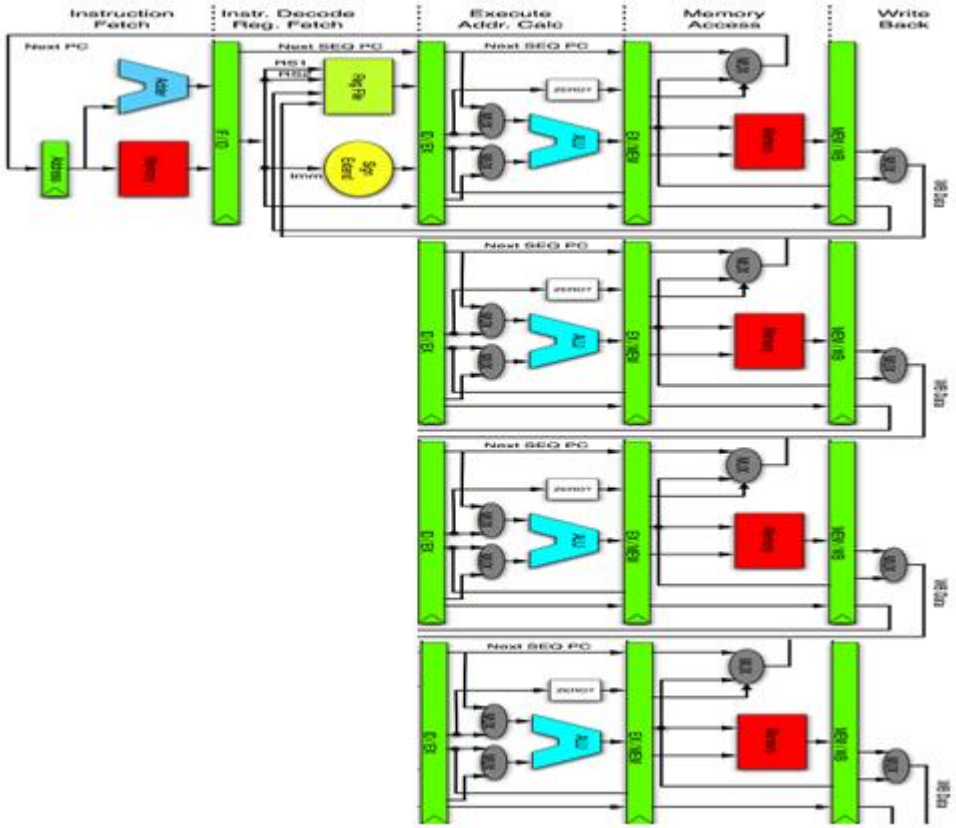
Vector/SIMD processors

Why decode same instruction sequence over and over?

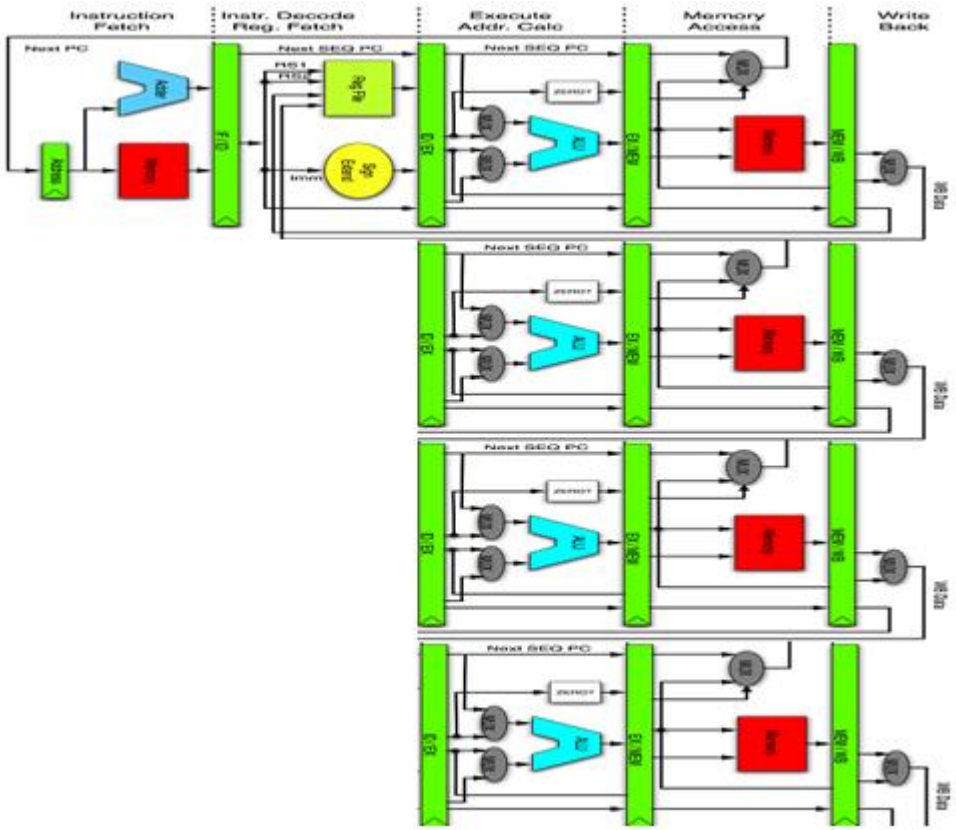



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

# 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

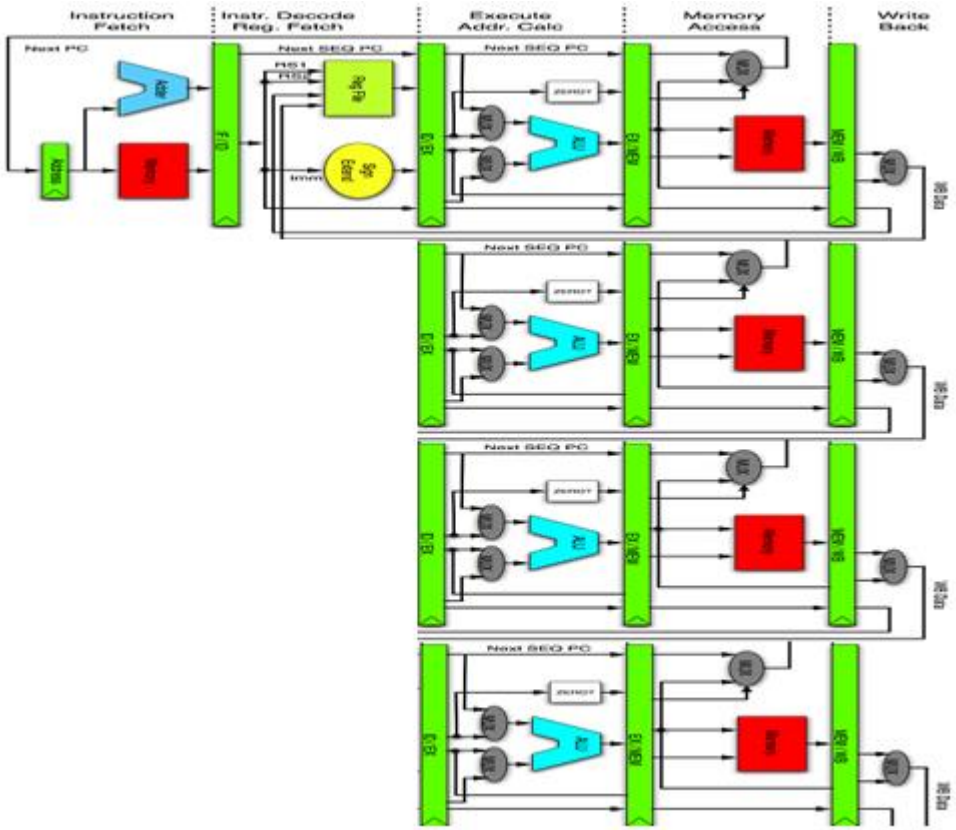


Vector/SIMD processors




```
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);
}
```

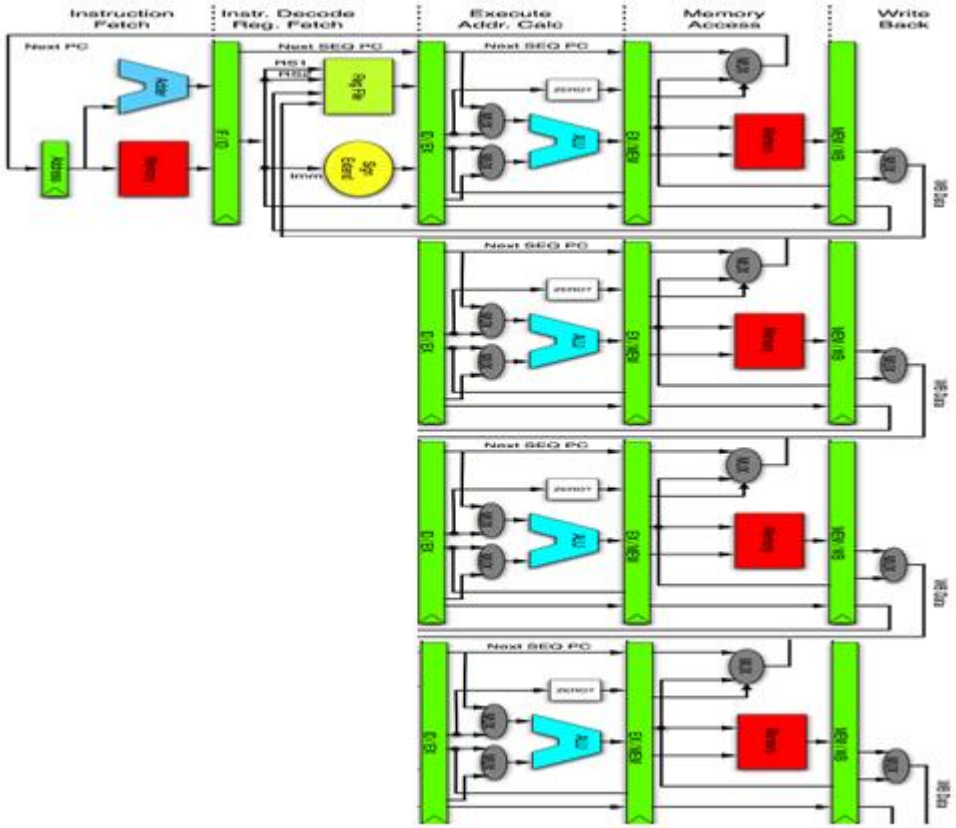
Vector/SIMD processors




```
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);
}
```

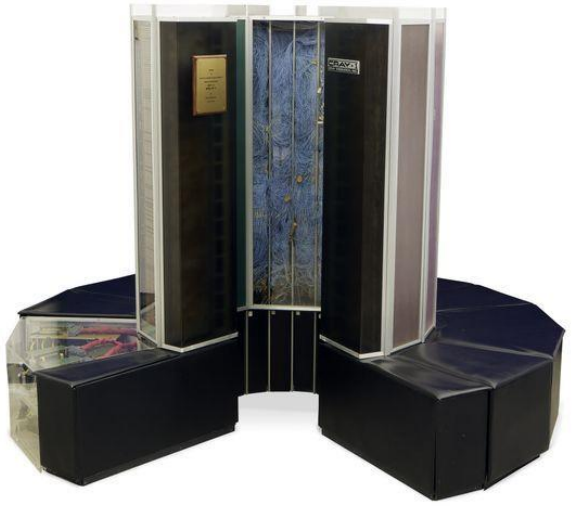
Single instruction stream, multiple computations

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

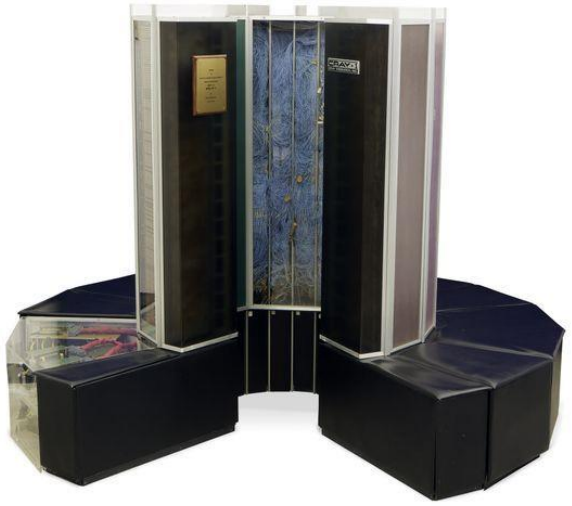
Vector Processors



- 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

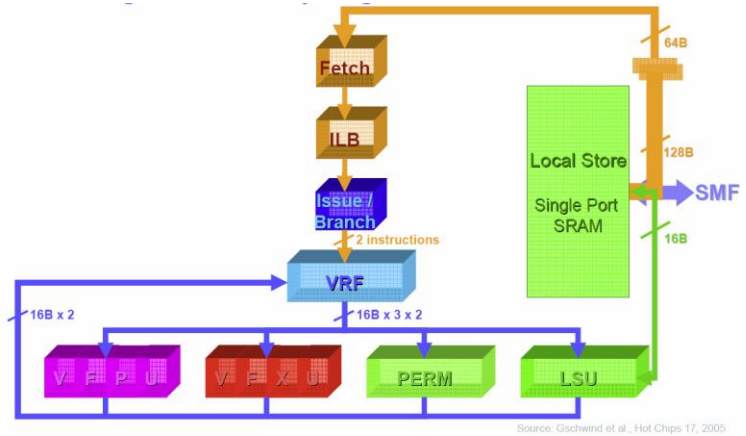


- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970's 80's and 90's.
- Modern CPUs support some vector processing instructions
  - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
  - SIMD operate on all per clock cycle

- 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

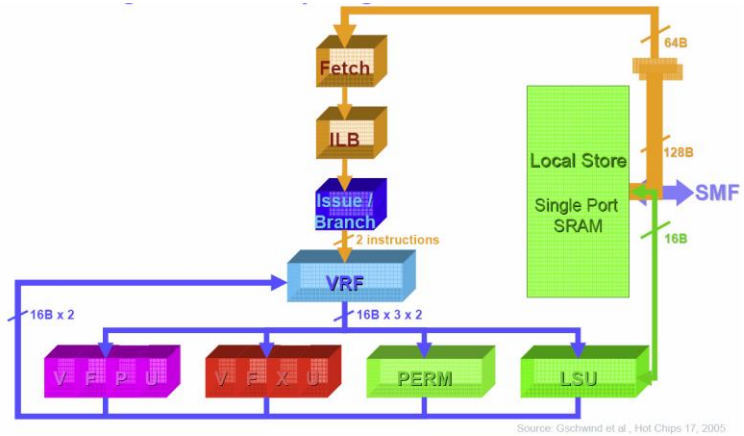
Vector Processors

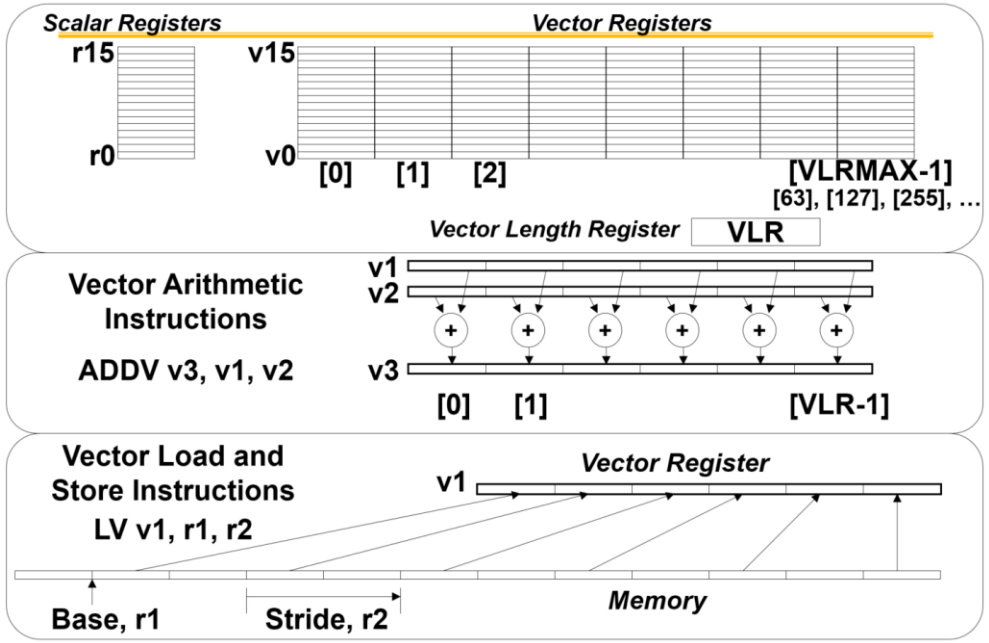


Implementation:

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

Vector Processors

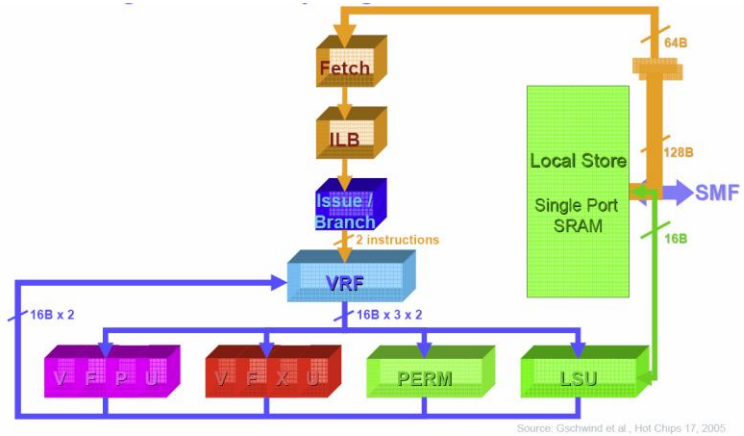


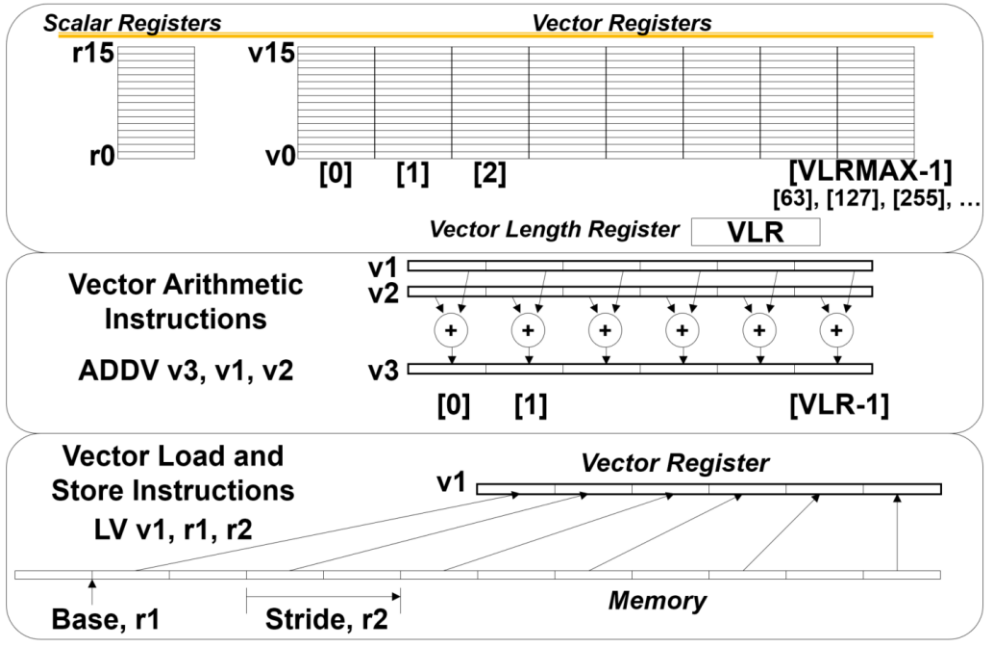


Implementation:

- 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

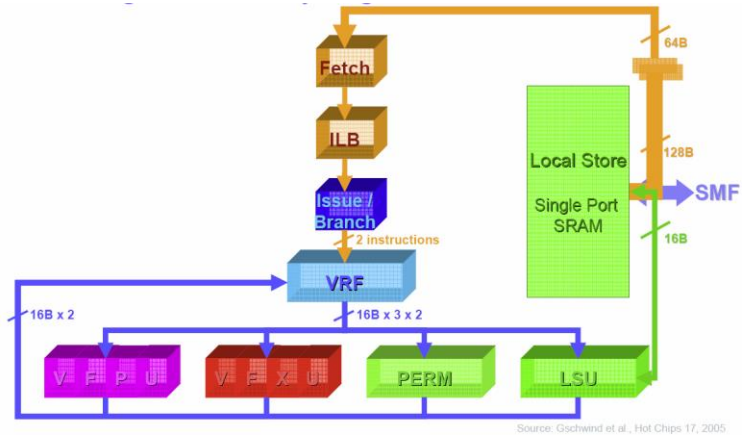
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# 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 Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

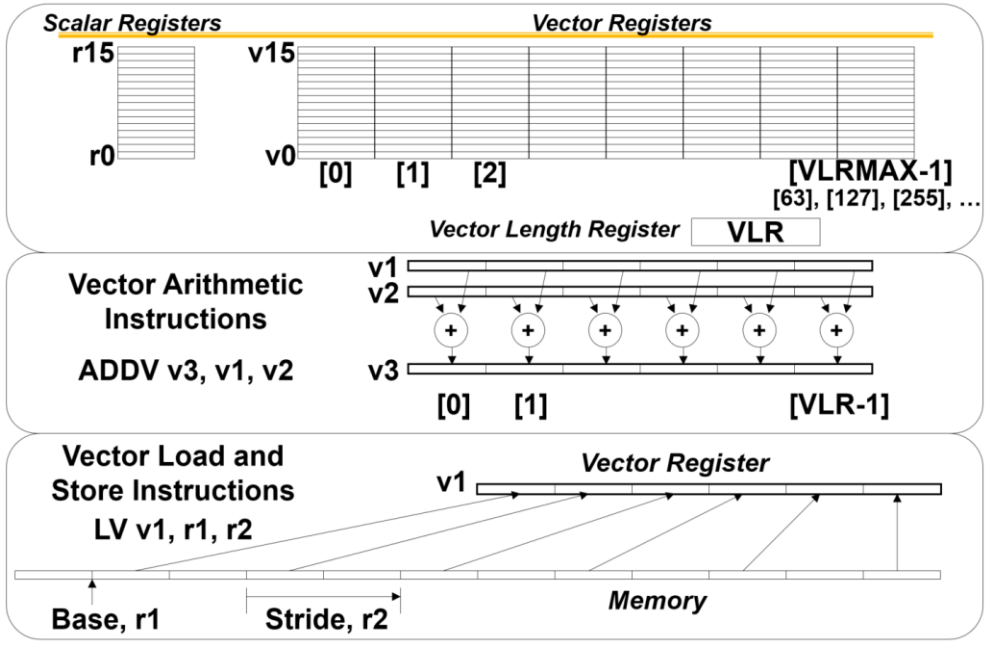
Vector Processors

GPUs: same basic idea



Implementation:

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

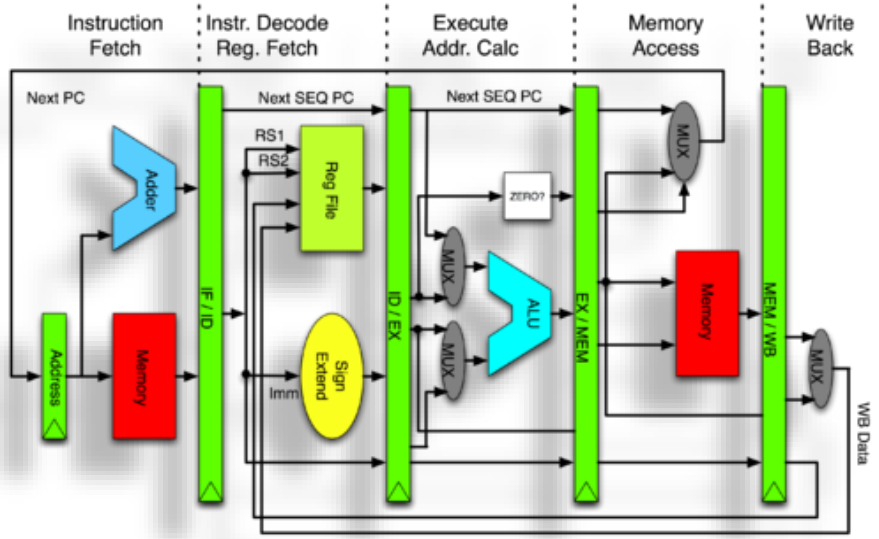


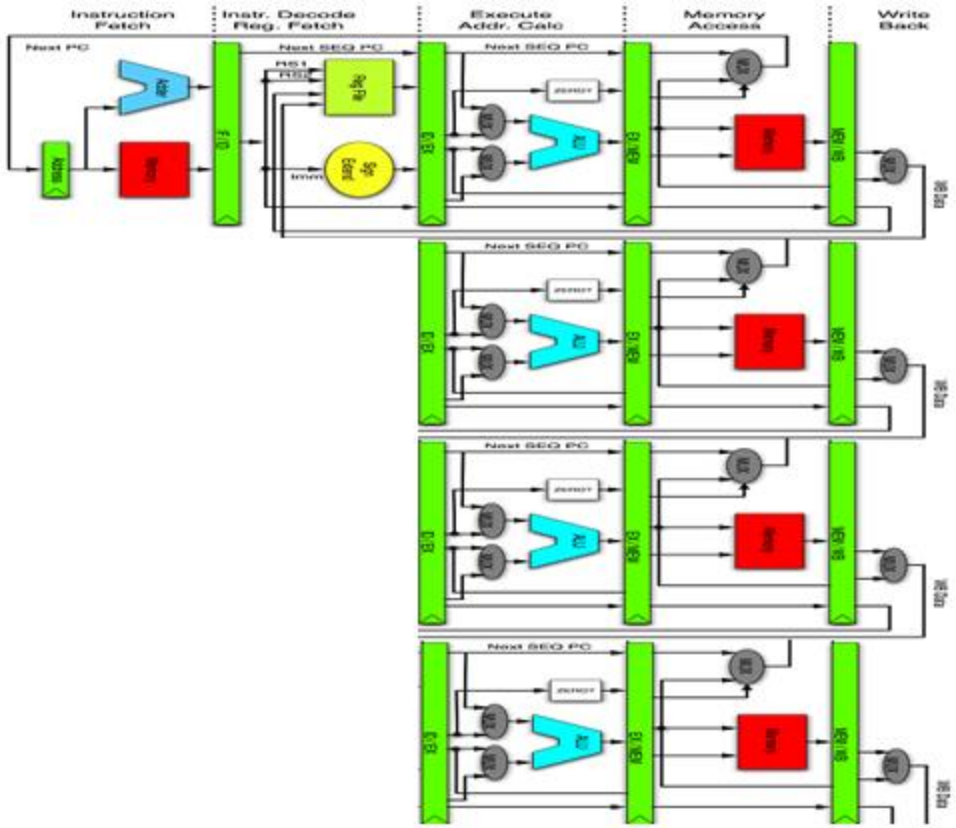

```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# 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
```

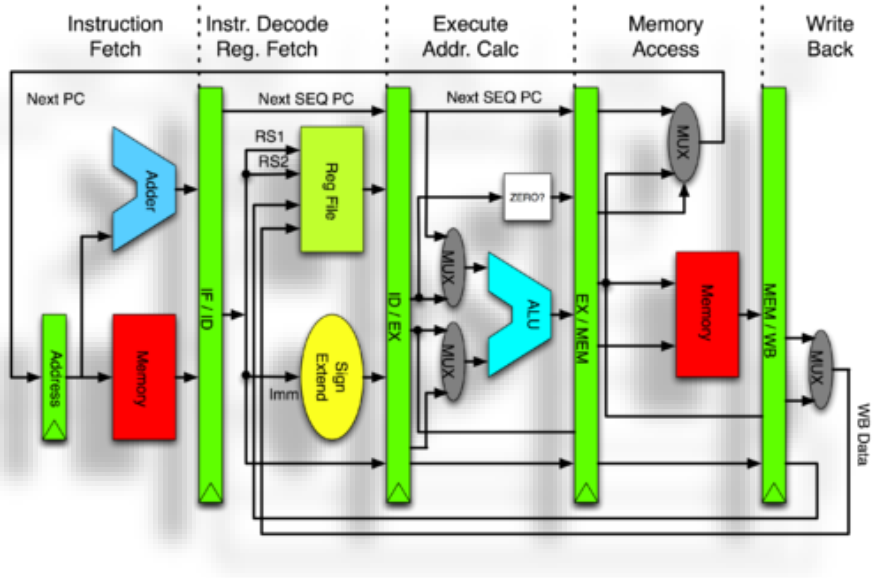
```
# Vector Code
   LI     VLR, 64
   LV     V1, R1
   LV     V2, R2
   ADDV.D V3, V1, V2
   SV     V3, R3
```

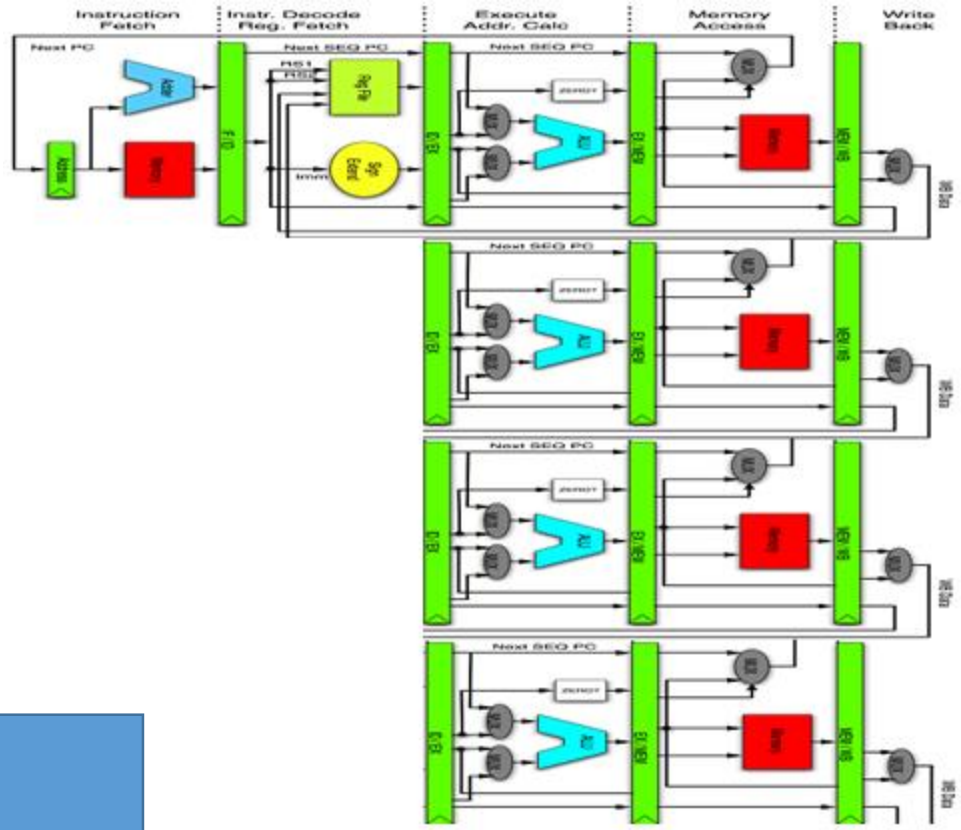
When does vector processing help?





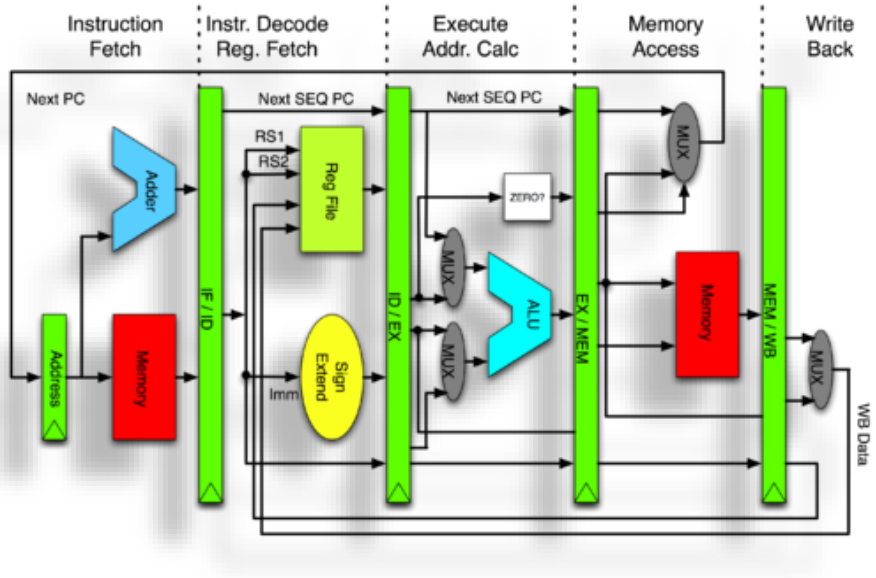
When does vector processing help?

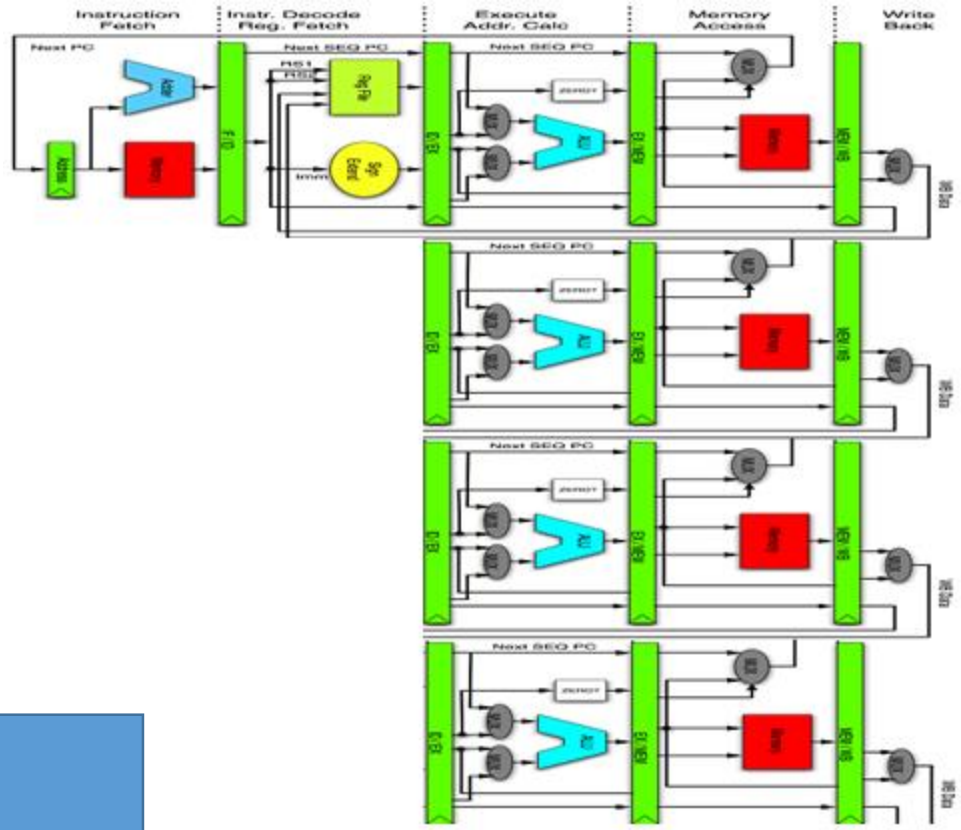




What are the potential bottlenecks here?
When can it improve throughput?

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

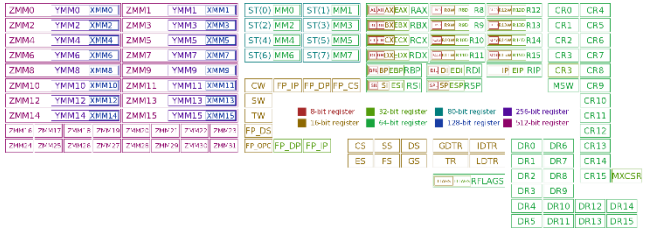
- Address memory bottleneck

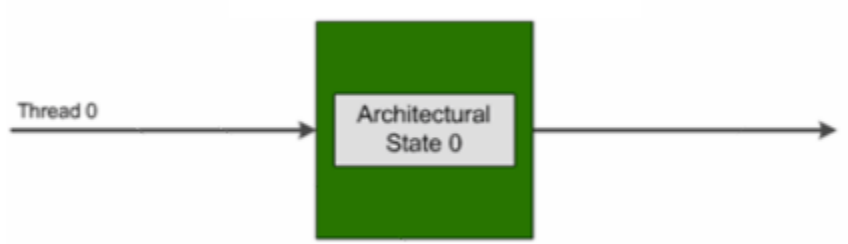
Hardware multi-threading

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

Hardware multi-threading

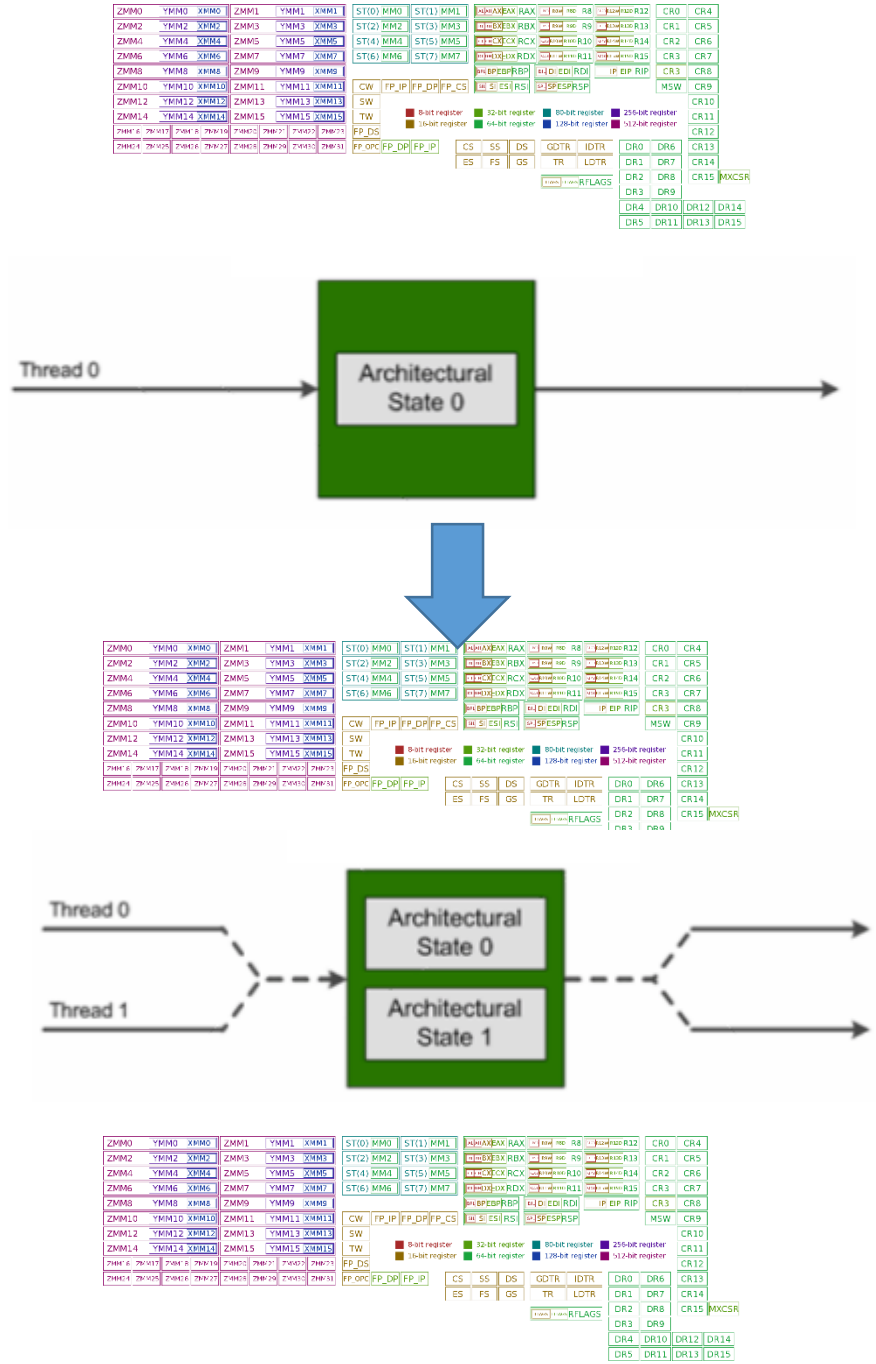
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls





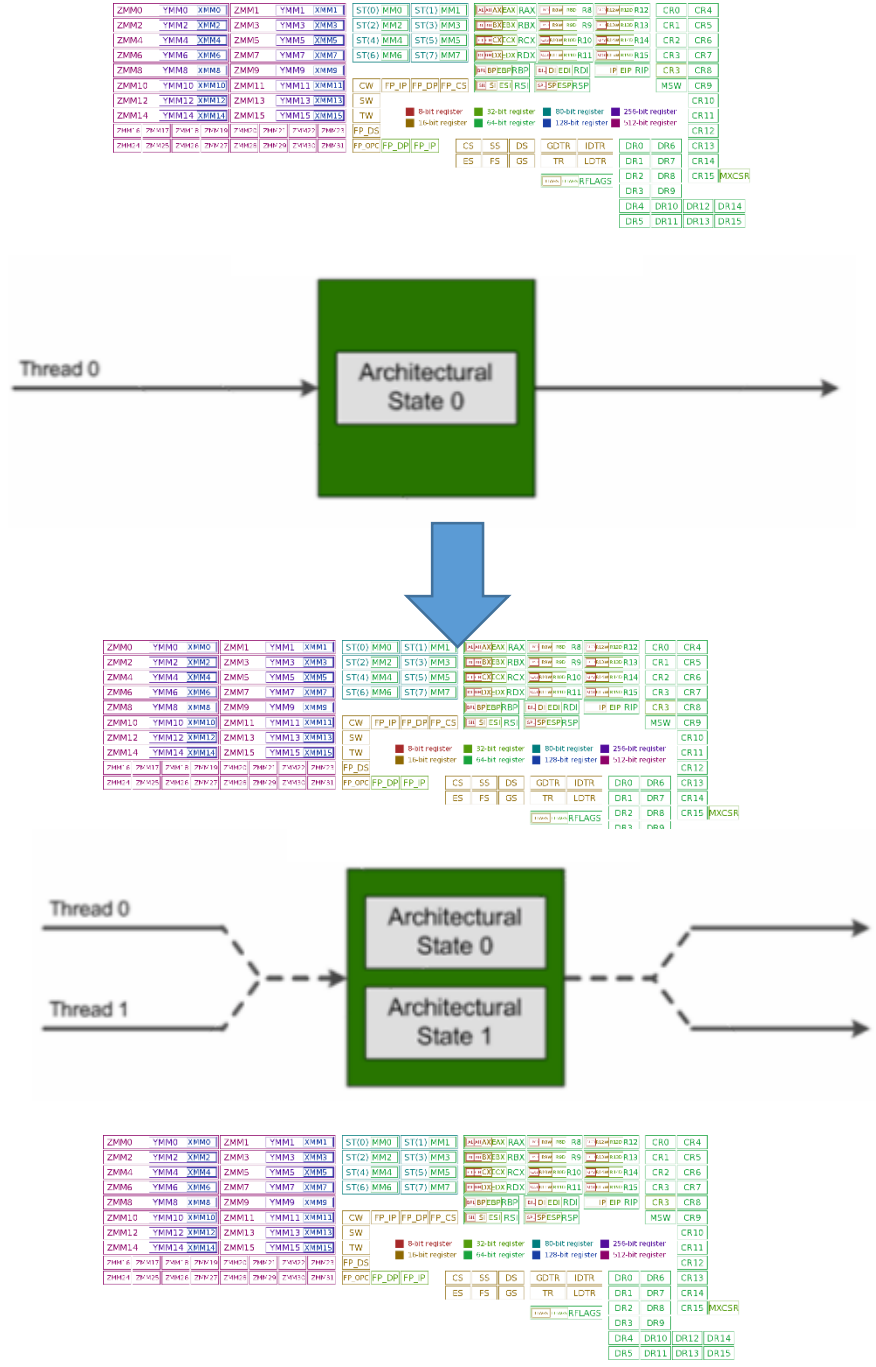
Hardware multi-threading

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



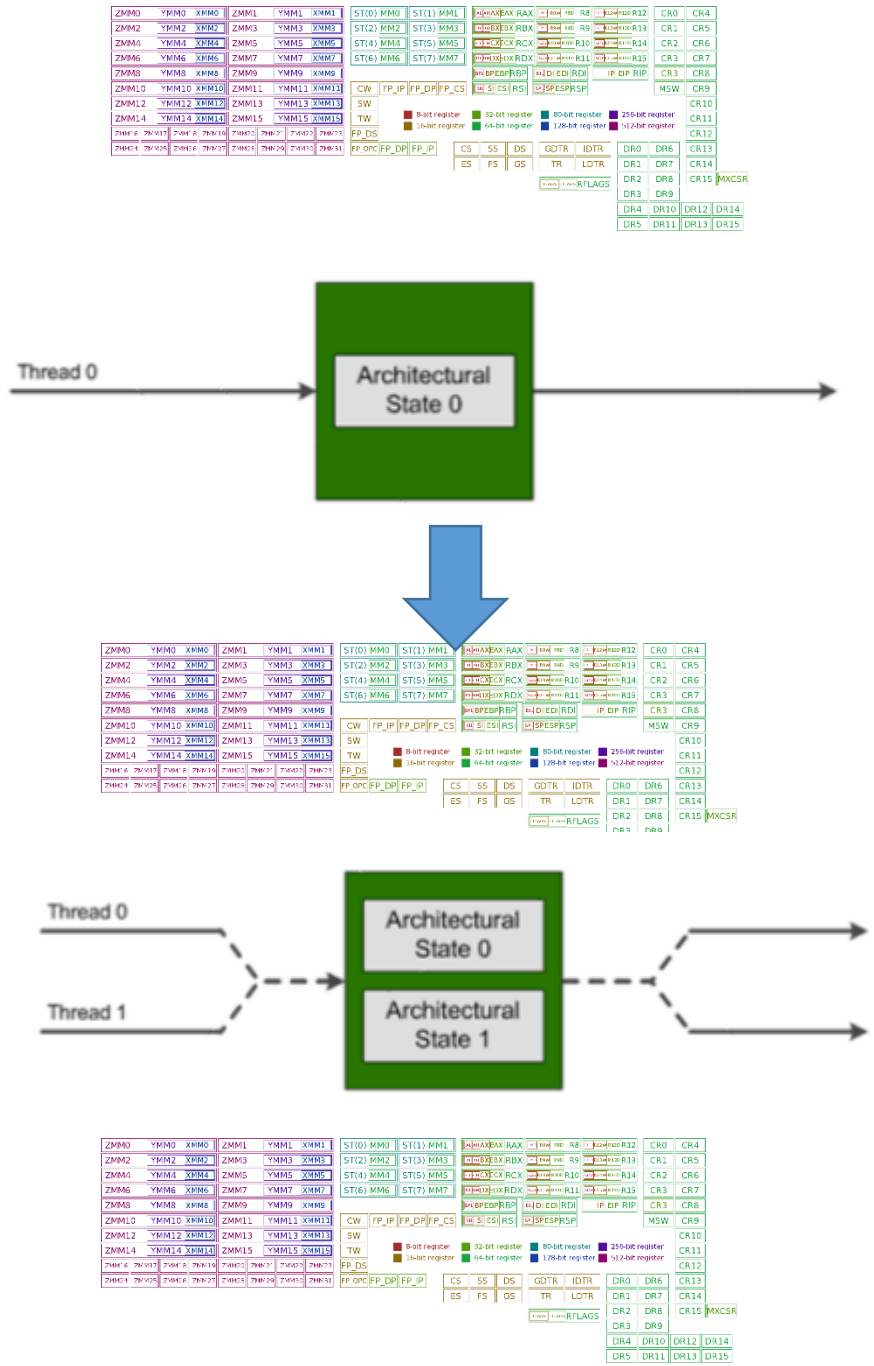
Hardware multi-threading

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



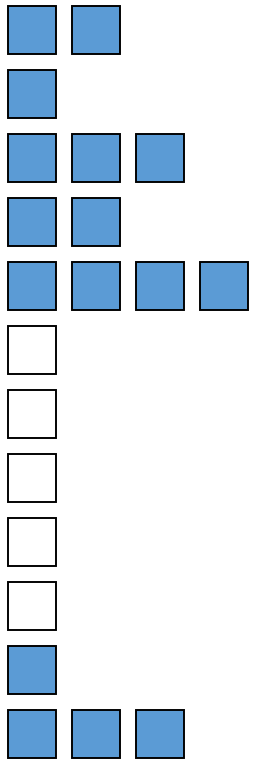
Hardware multi-threading

- 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

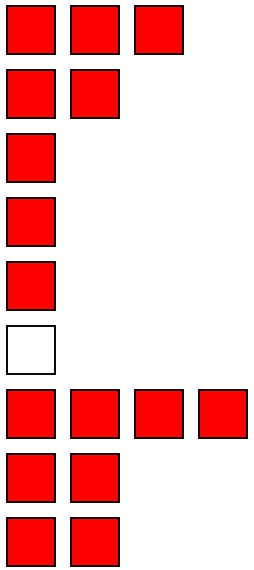


Running example

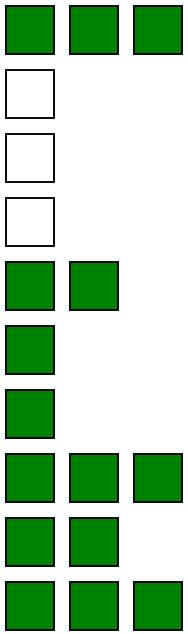
Thread A



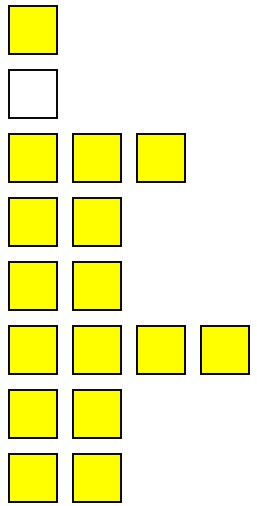
Thread B



Thread C



Thread D



- Colors → pipeline full
- White → stall

Coarse- grained multithreading

Coarse- grained multithreading

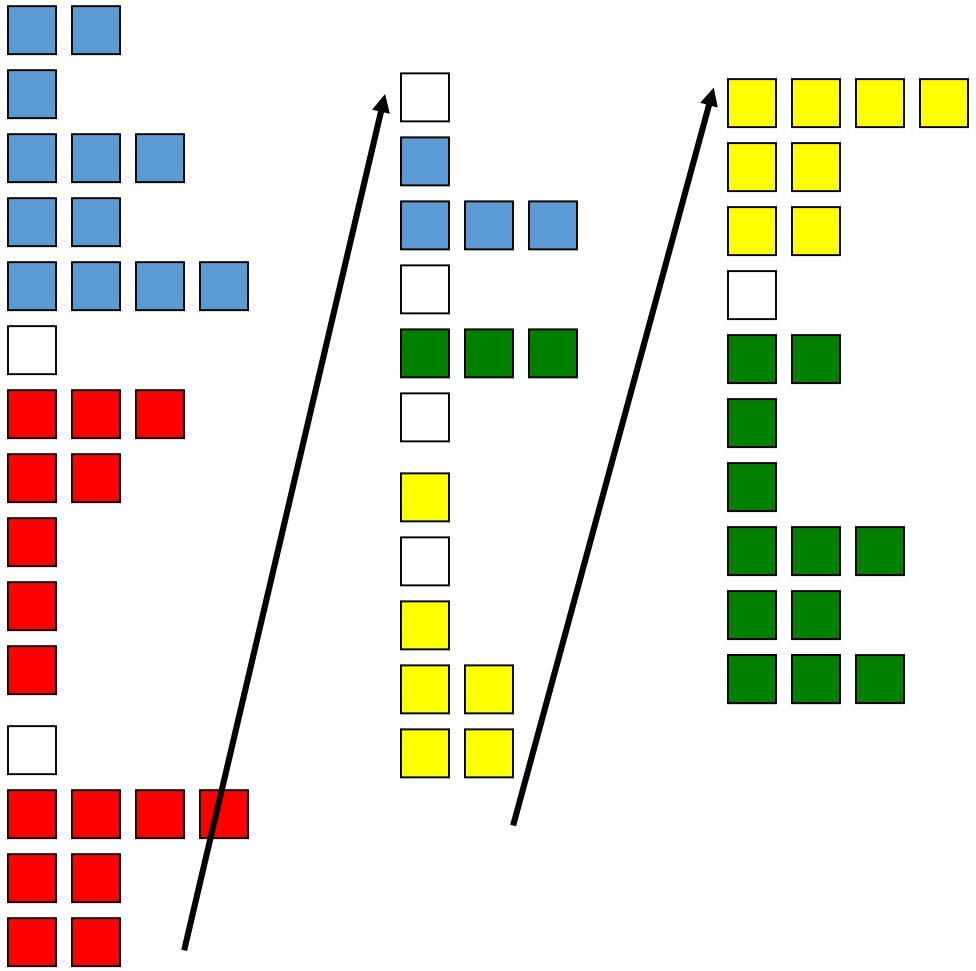
- Single thread runs until a costly stall
  - E.g. 2nd level cache miss

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!

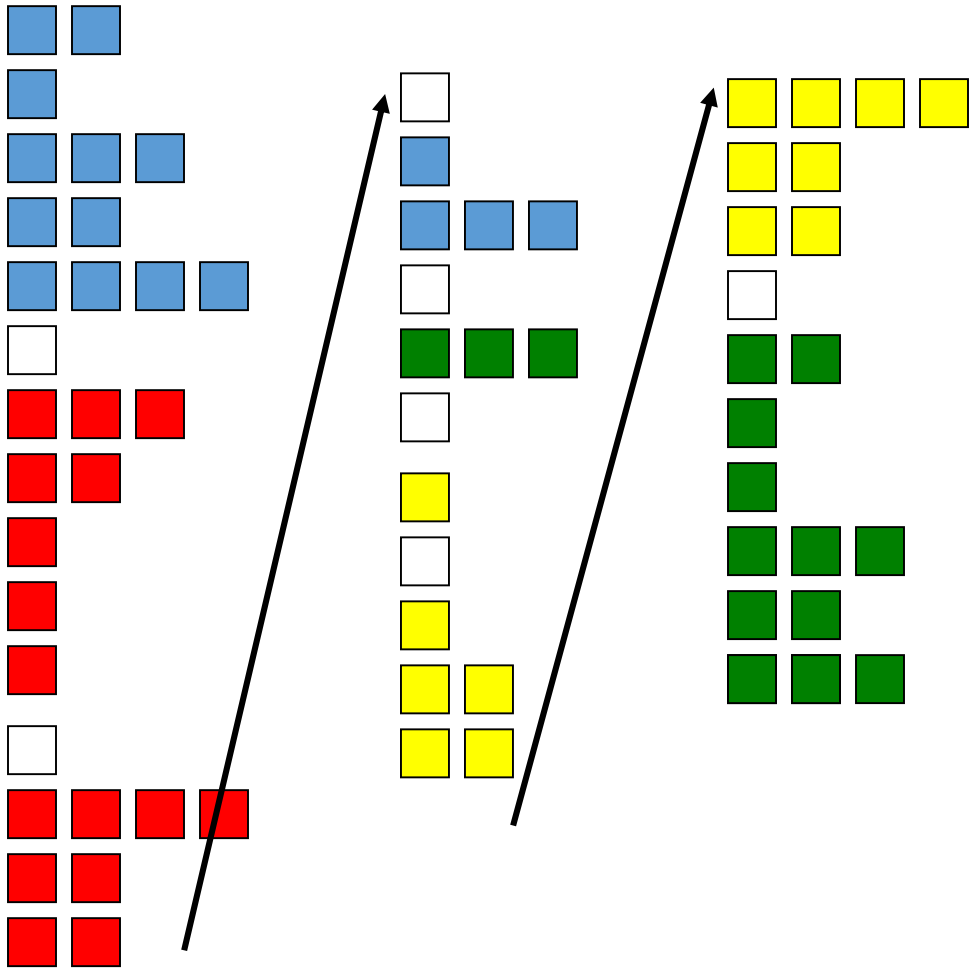
Coarse- grained multithreading

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



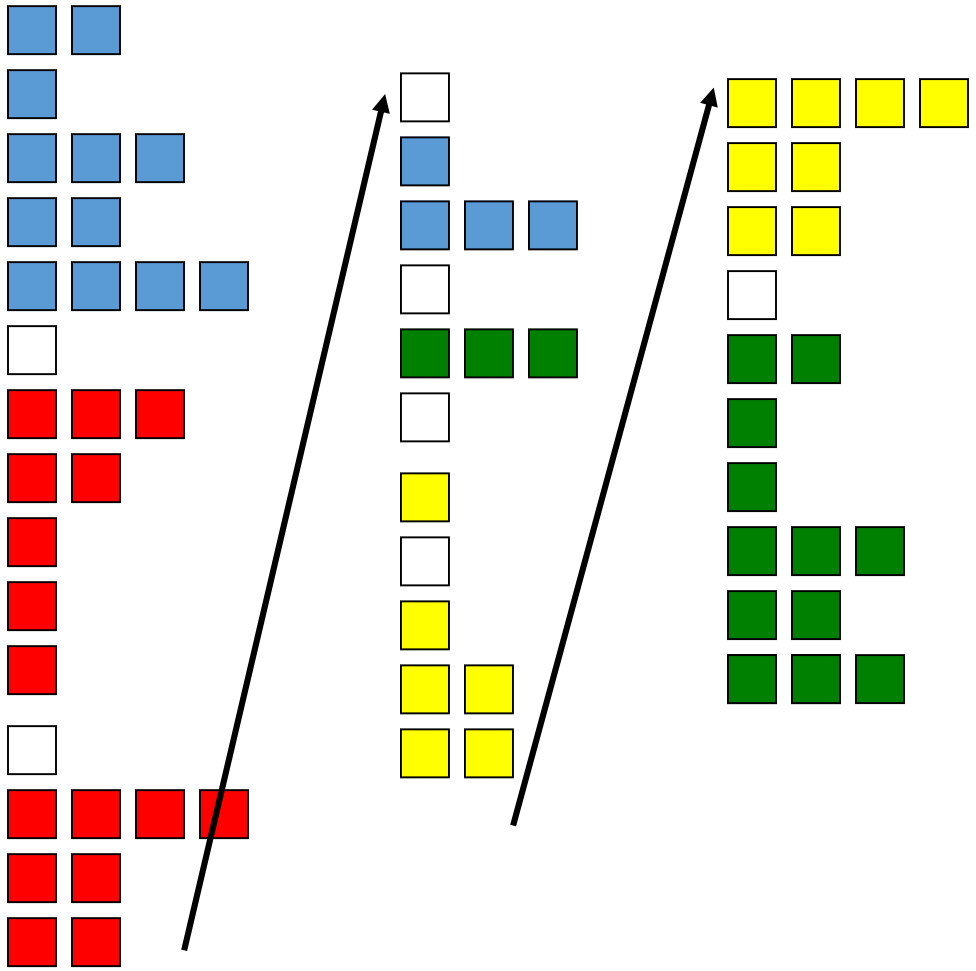
Coarse- grained multithreading

- 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



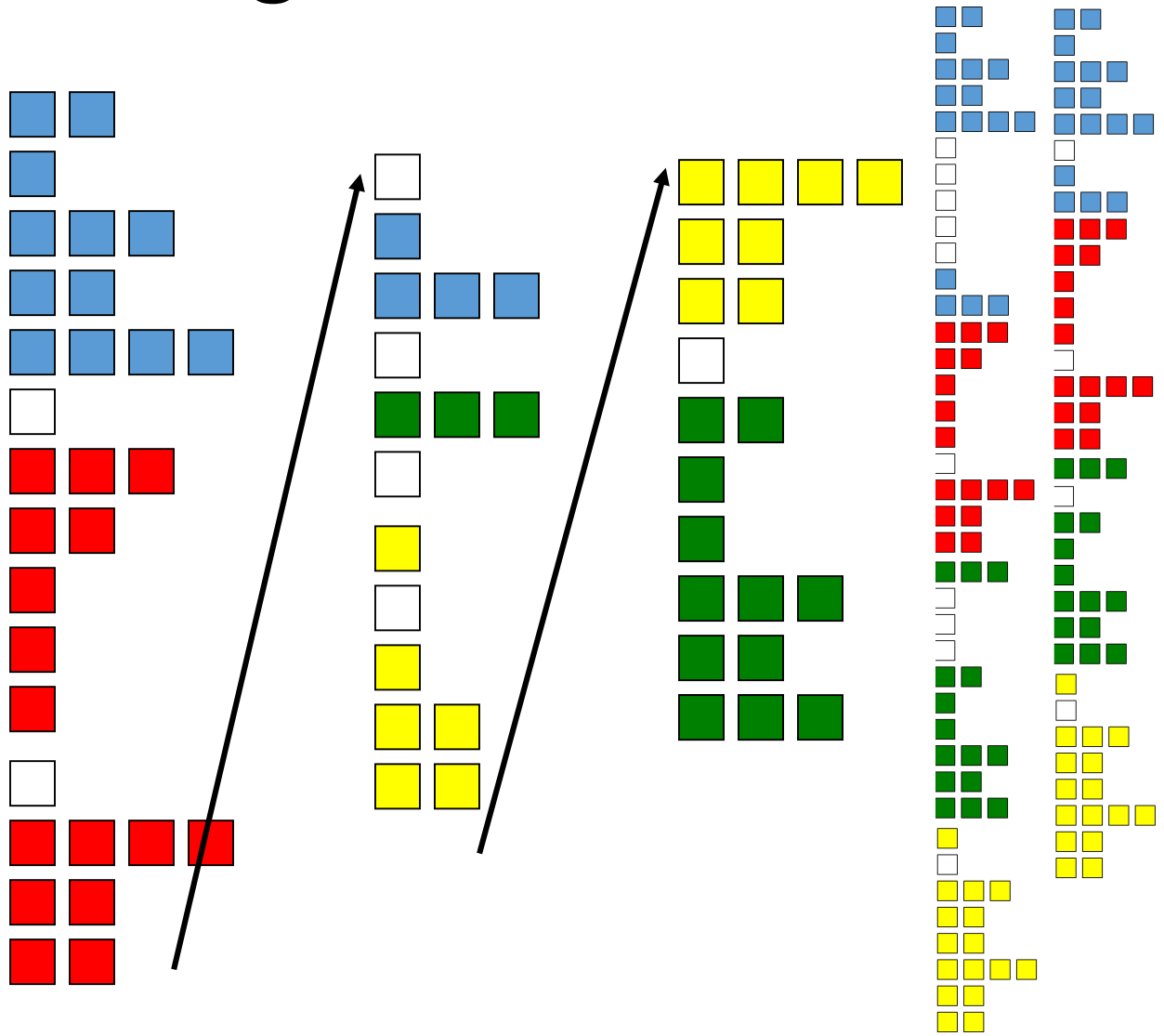
Coarse- grained multithreading

- 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
- Hardware support required
  - PC and register file for each thread
  - little other hardware
  - Looks like another physical CPU to OS/software



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
- Hardware support required
  - PC and register file for each thread
  - little other hardware
  - Looks like another physical CPU to OS/software



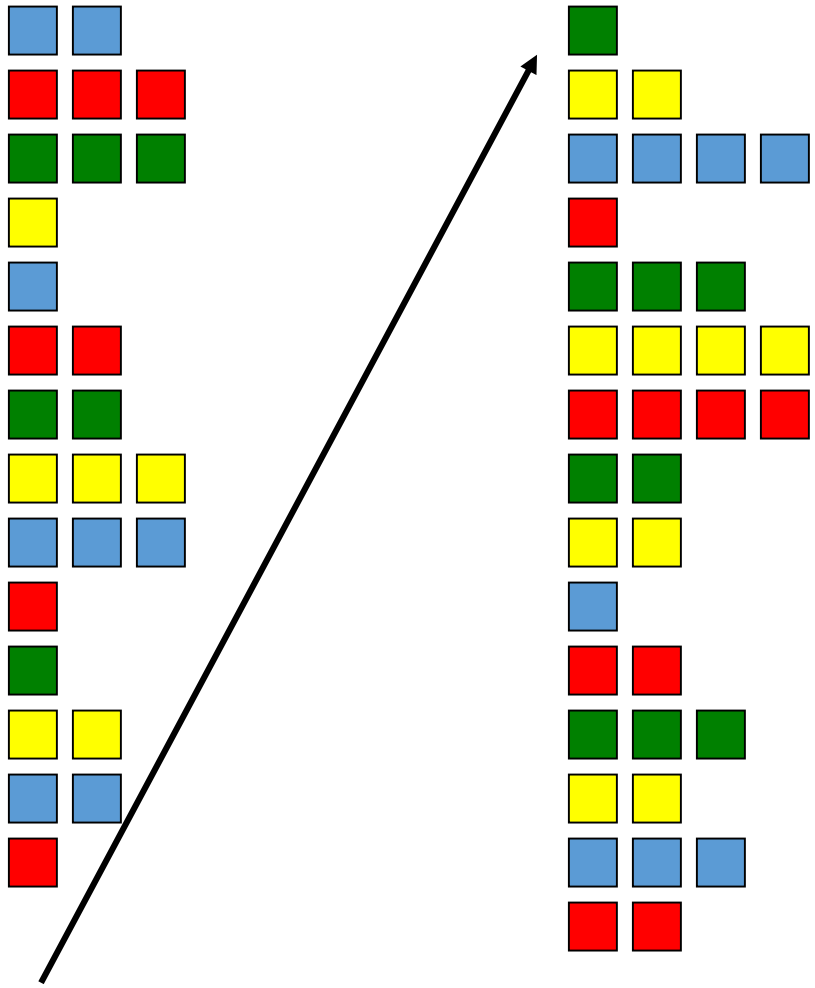
Fine-grained multithreading

Fine-grained multithreading

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

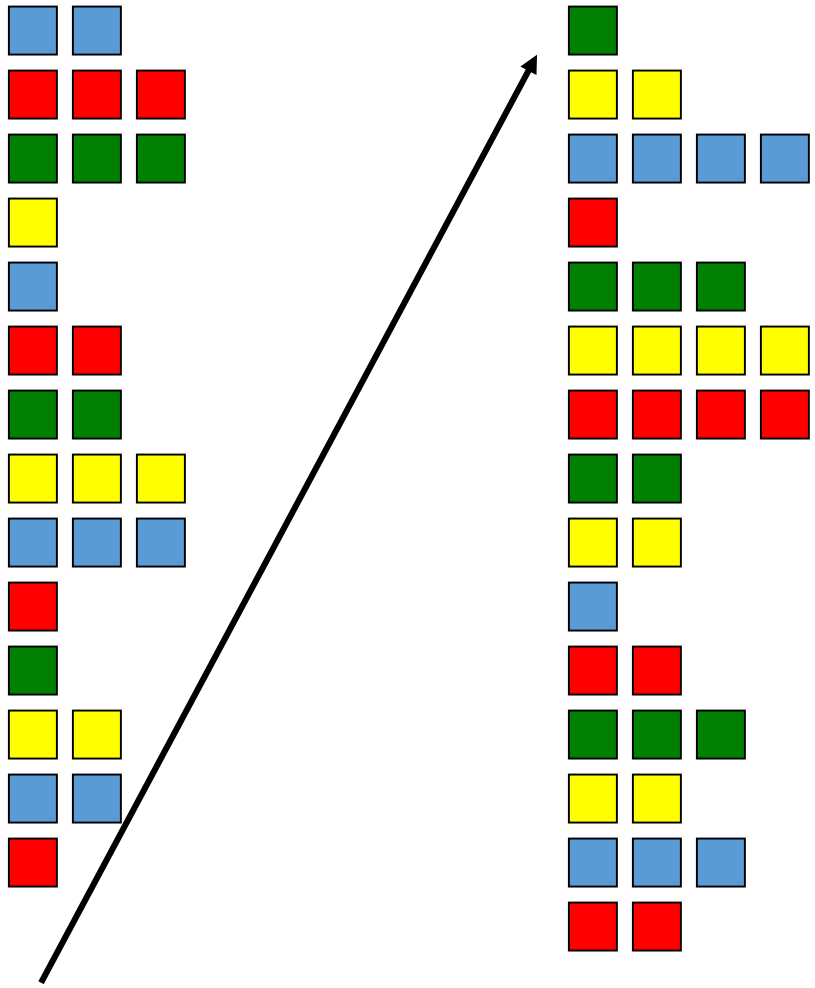
Fine-grained multithreading

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



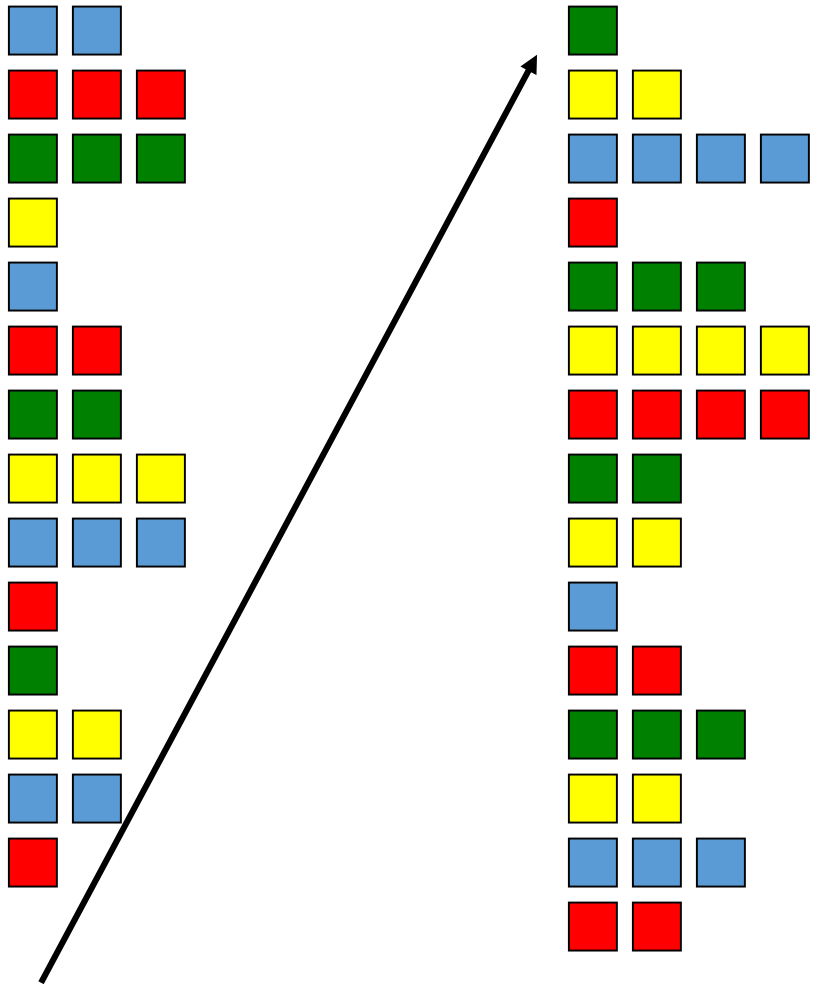
Fine-grained multithreading

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



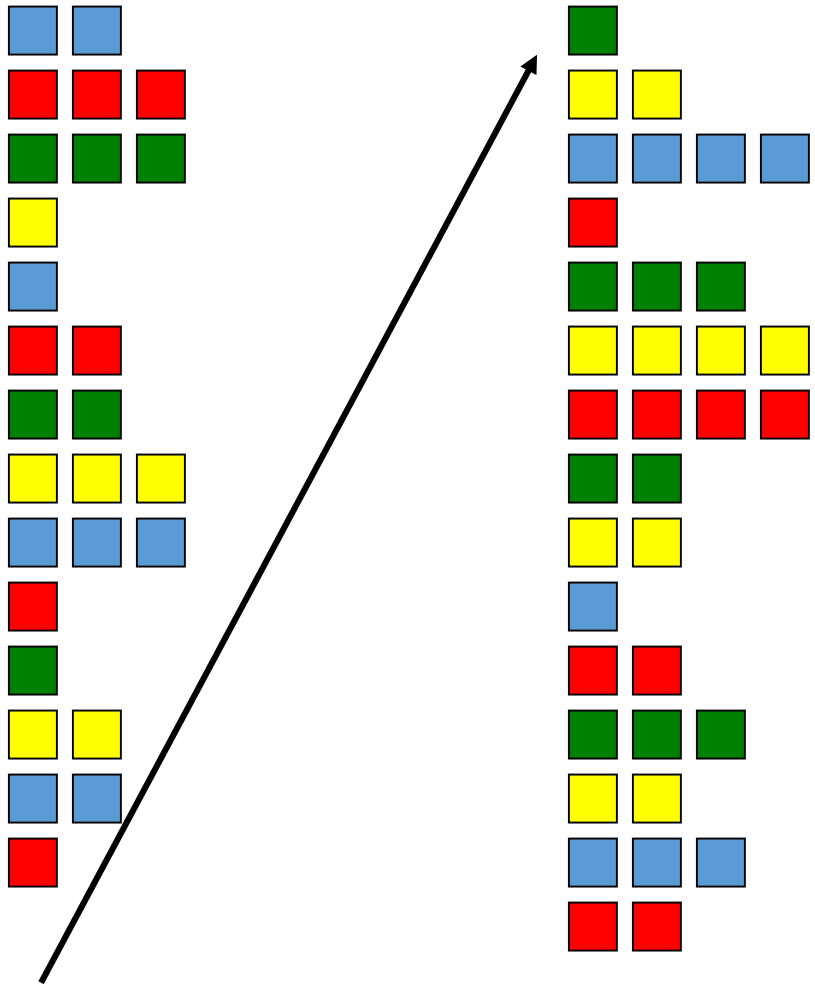
Fine-grained multithreading

- 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



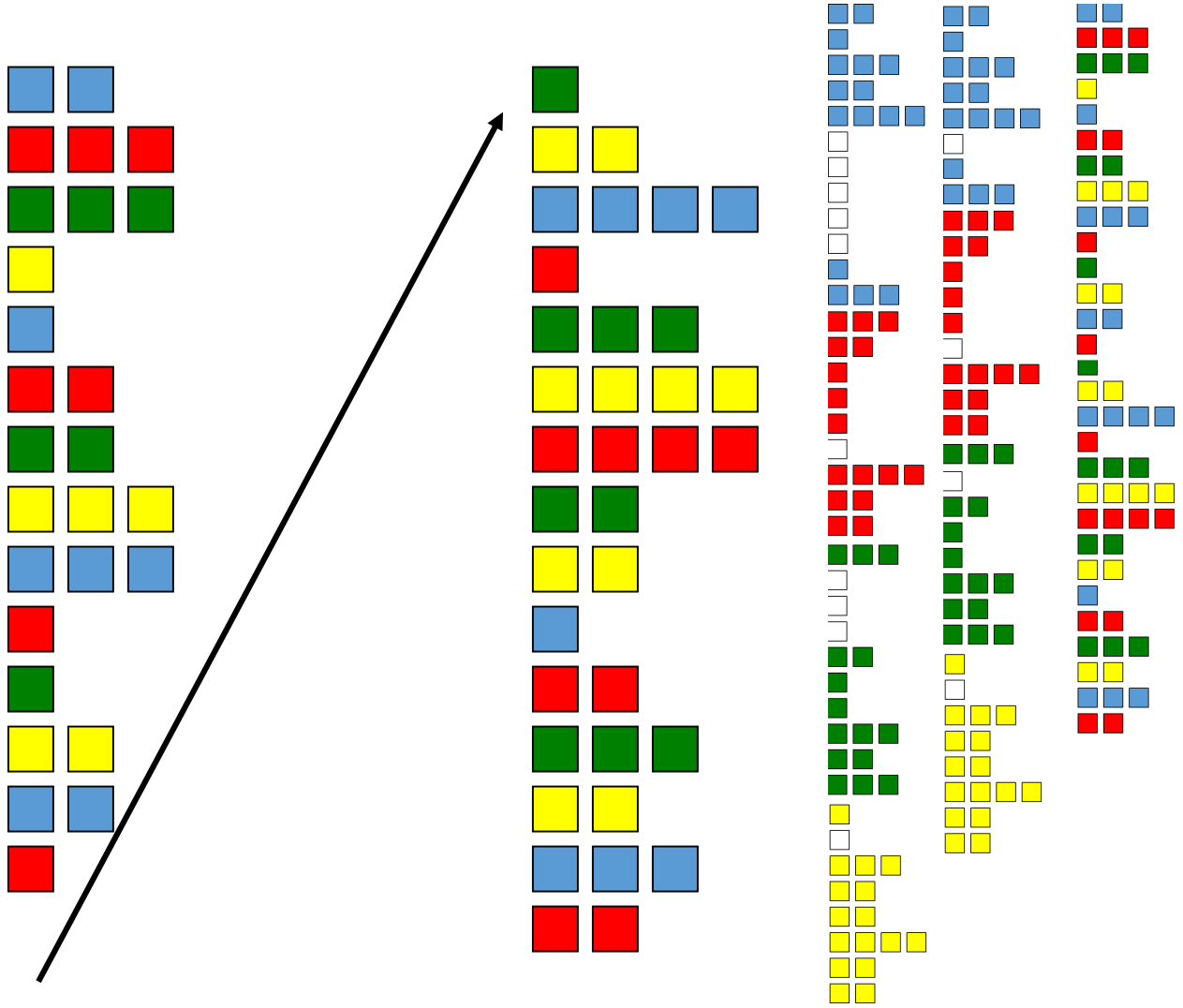
Fine-grained multithreading

- 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



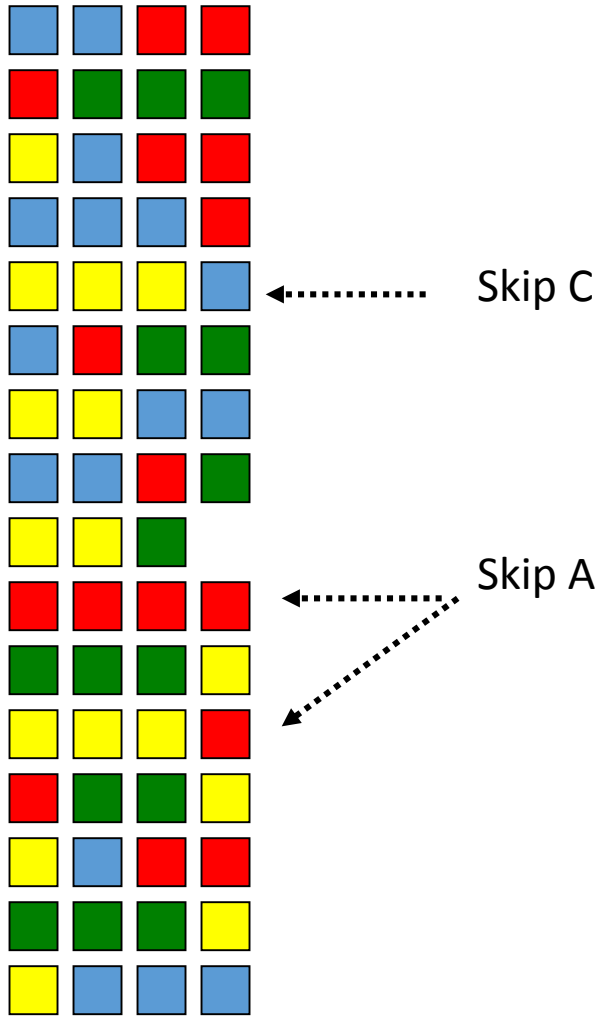
Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT)

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

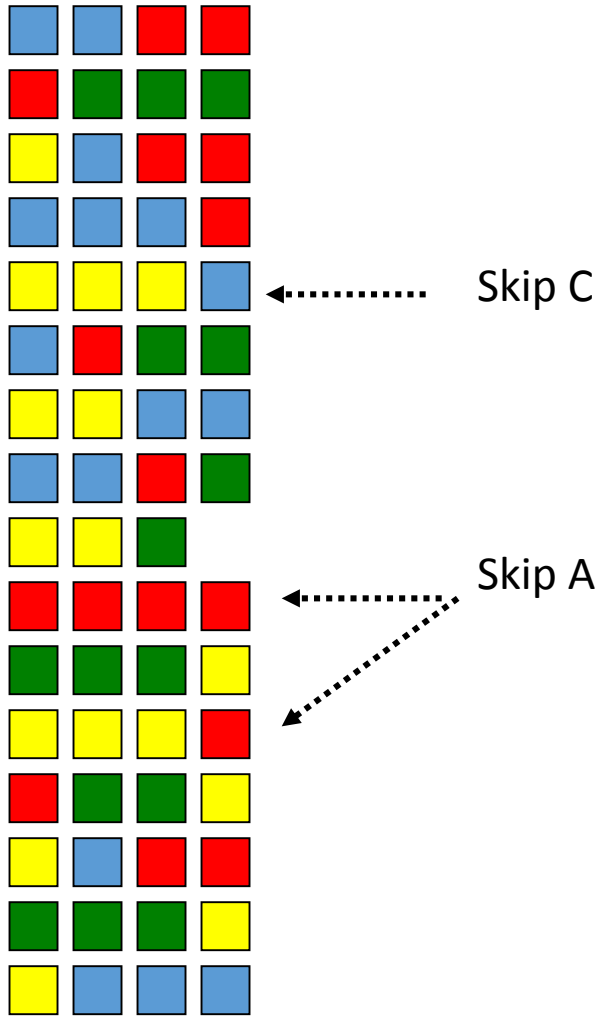
Simultaneous Multithreading (SMT)

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



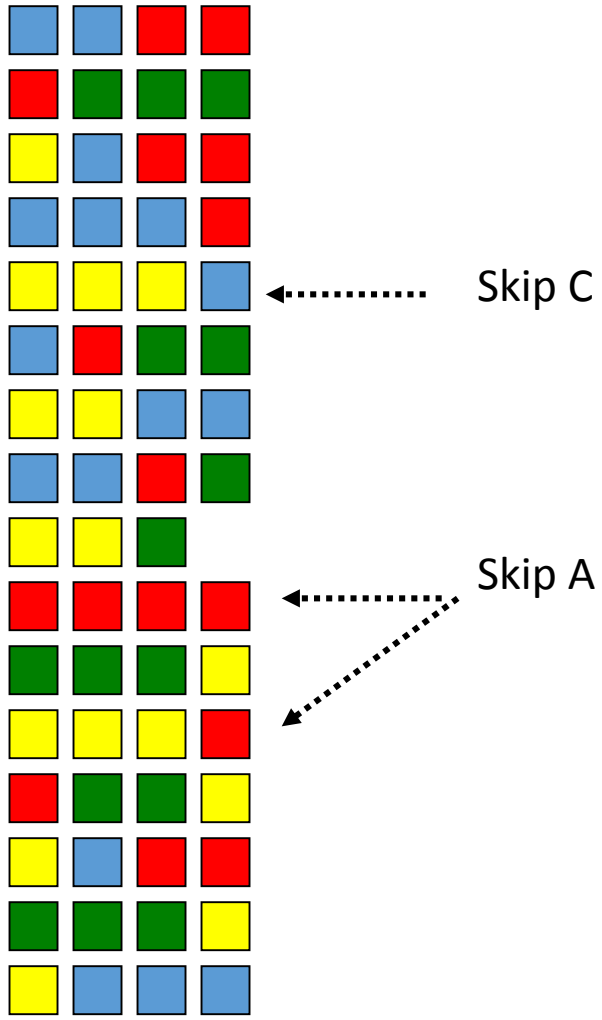
Simultaneous Multithreading (SMT)

- 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



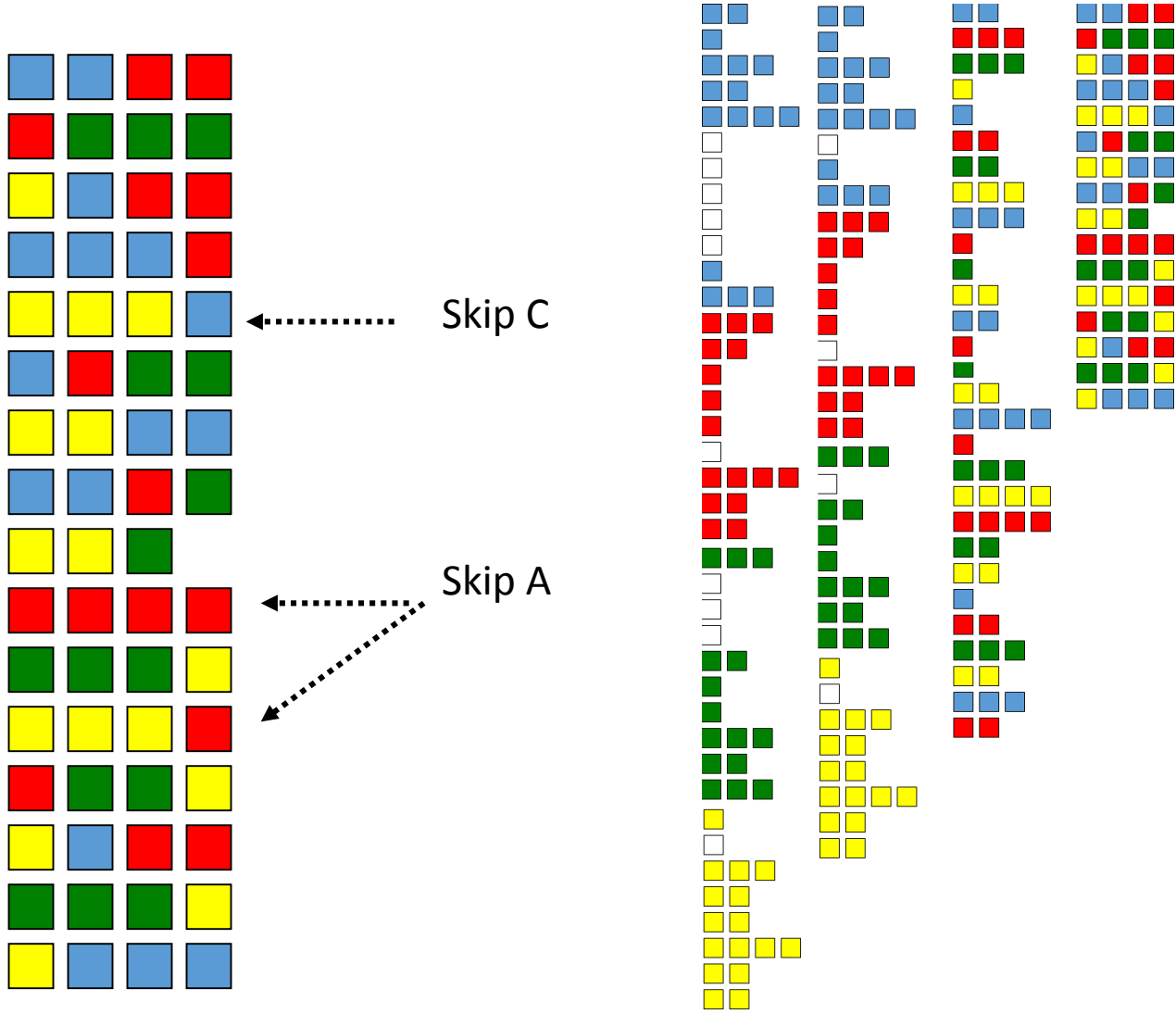
Simultaneous Multithreading (SMT)

- 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 ~= Rendering

Graphics ~= Rendering

Inputs

Graphics ~= 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 ~= 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 surtface/light interactions with modeled objects/materials

Graphics ~= 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 surtface/light interactions with modeled objects/materials
- View point

Graphics ~= 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 surtface/light interactions with modeled objects/materials
- View point

Output

Graphics ~= 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 surtface/light interactions with modeled objects/materials
- View point

Output

- 2D projection seen from the view-point

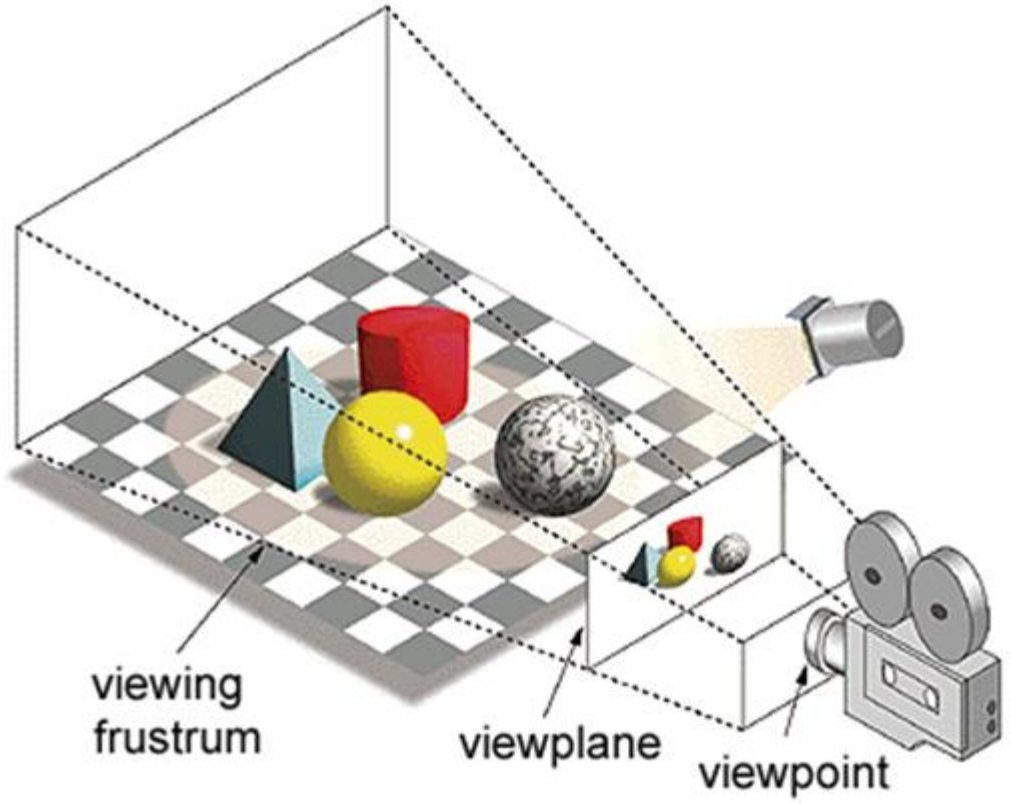
Graphics ~= Rendering

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 → geometry
- Light locations and properties
  - Attempt to model surtface/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 $v_{model} \rightarrow v_{view}$

Grossly over-simplified rendering algorithm

foreach(vertex v in model)

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

Grossly over-simplified rendering algorithm

foreach(vertex v in model)

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

fragment[] frags = {};

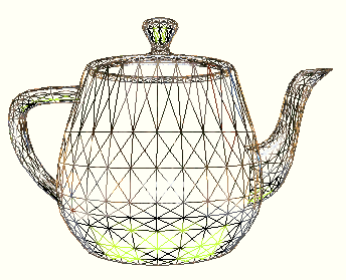
Grossly over-simplified rendering algorithm

foreach(vertex v in model)

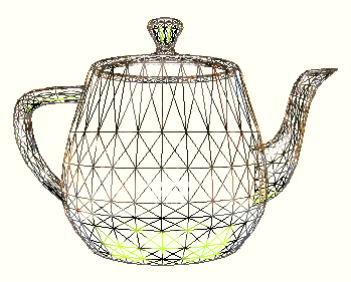
map $v_{model} \rightarrow v_{view}$

fragment[] frags = {};

foreach triangle t (v_0, v_1, v_2)

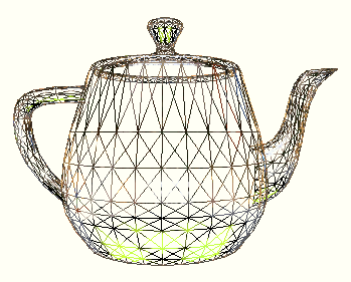


Grossly over-simplified rendering algorithm



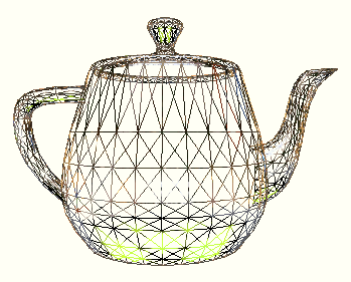
```
foreach(vertex v in model)
        map v_model → v_view
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
        frags.add(rasterize(t));
```

Grossly over-simplified rendering algorithm



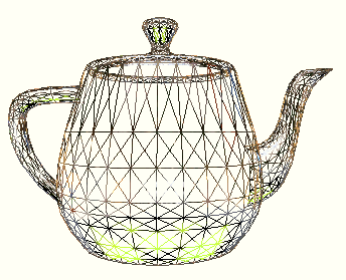
```
foreach(vertex v in model)
        map v_model → v_view
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
        frags.add(rasterize(t));
foreach fragment f in frags
```

Grossly over-simplified rendering algorithm



```
foreach(vertex v in model)
        map v_model → v_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



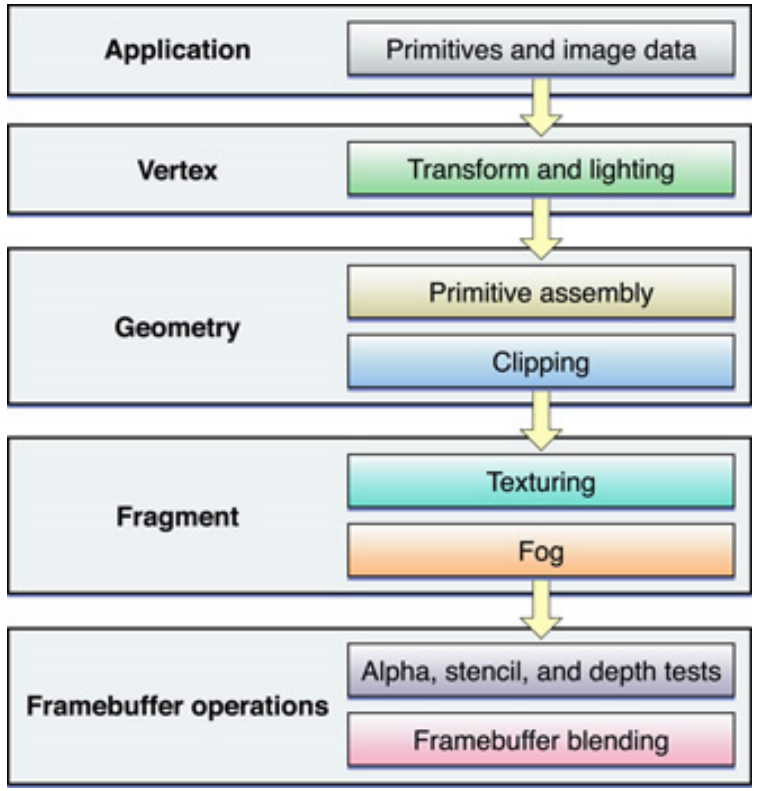
```
foreach(vertex v in model)
        map v_model → v_view
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
        frags.add(rasterize(t));
foreach fragment f in frags
        choose_color(f);
display(visible_fragments(frags));
```

Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
        map v_model → v_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 → Graphics Pipeline

```
foreach(vertex v in model)
        map v_model → v_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));
```

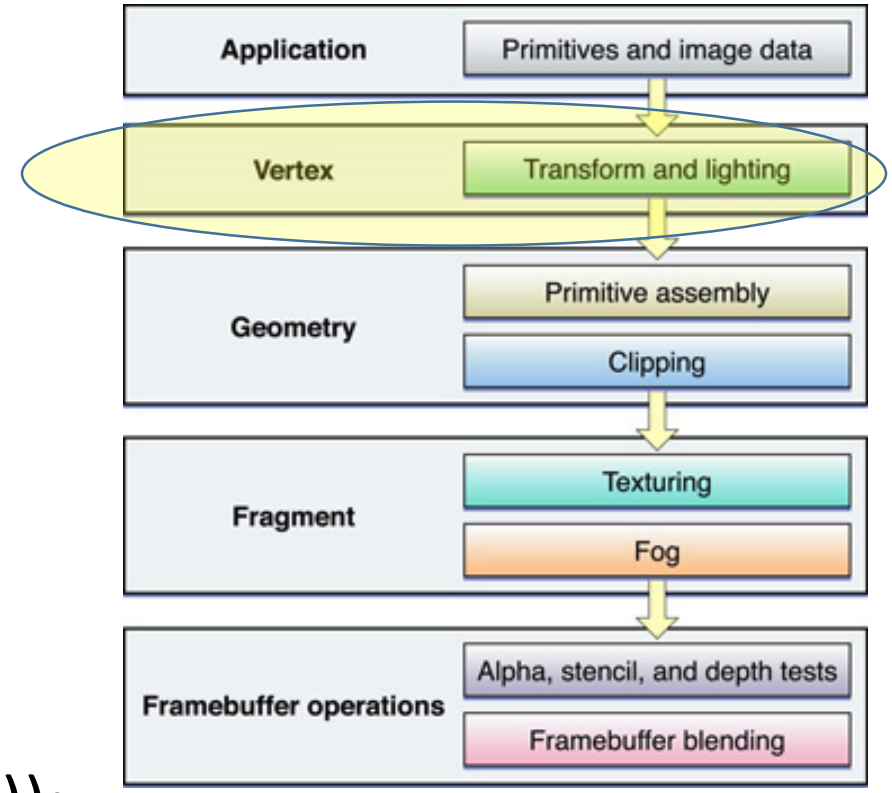


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map v_model → v_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));
```

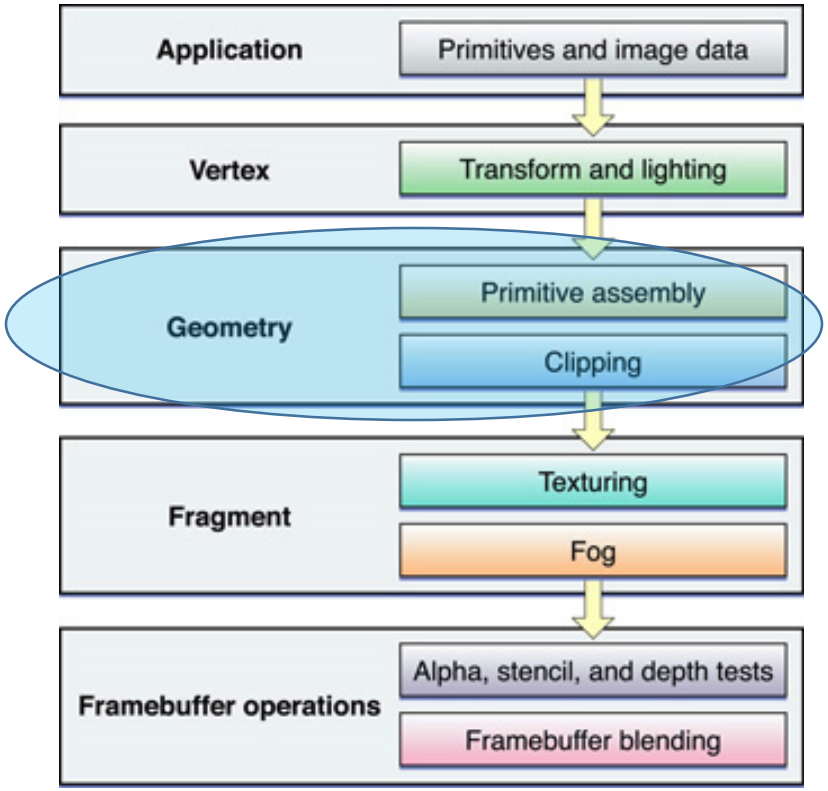


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map v_model → v_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));
```

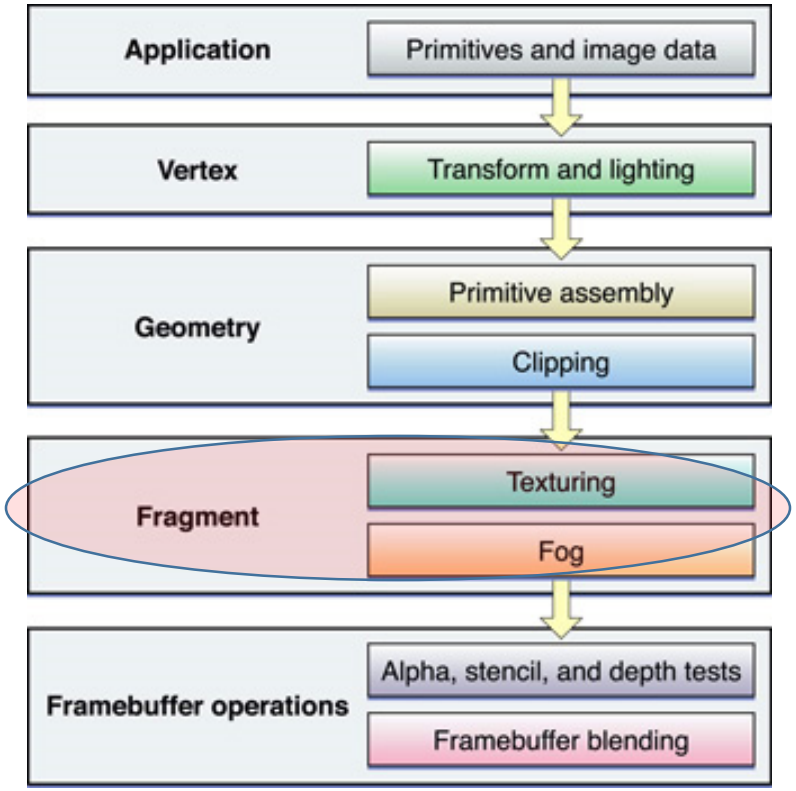


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map v_model → v_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));
```

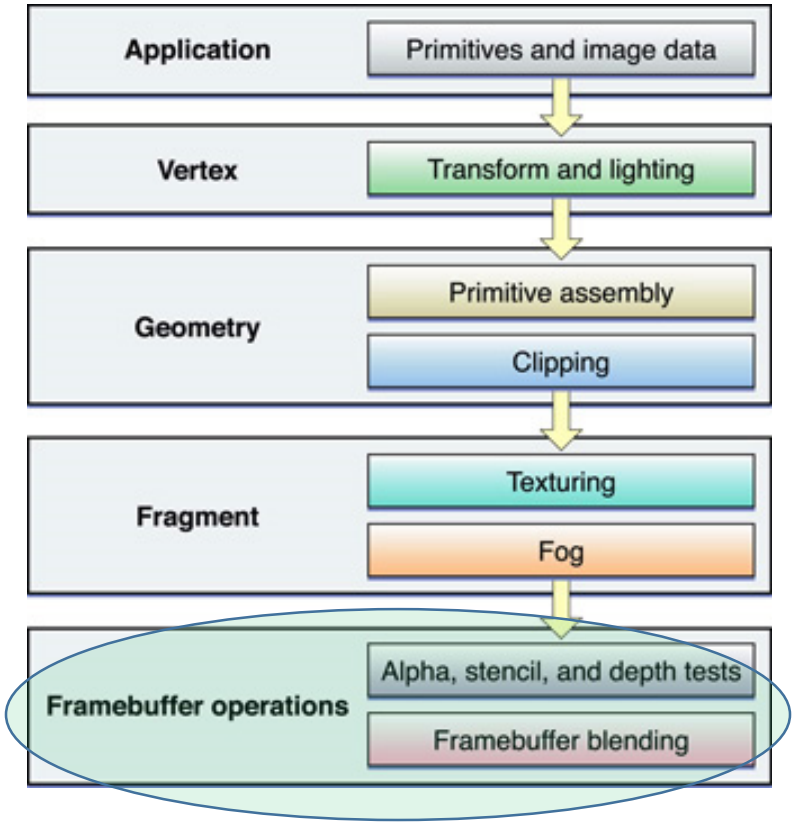


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

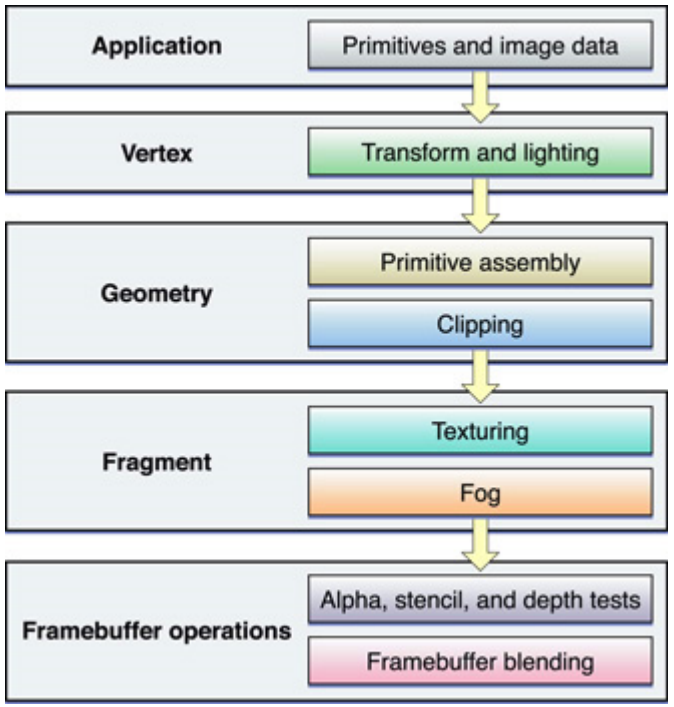
```
foreach(vertex v in model)
        map v_model → v_view
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
        frags.add(rasterize(t));
foreach fragment f in frags
        choose_color(f);
display(visible_fragments(frags));
```

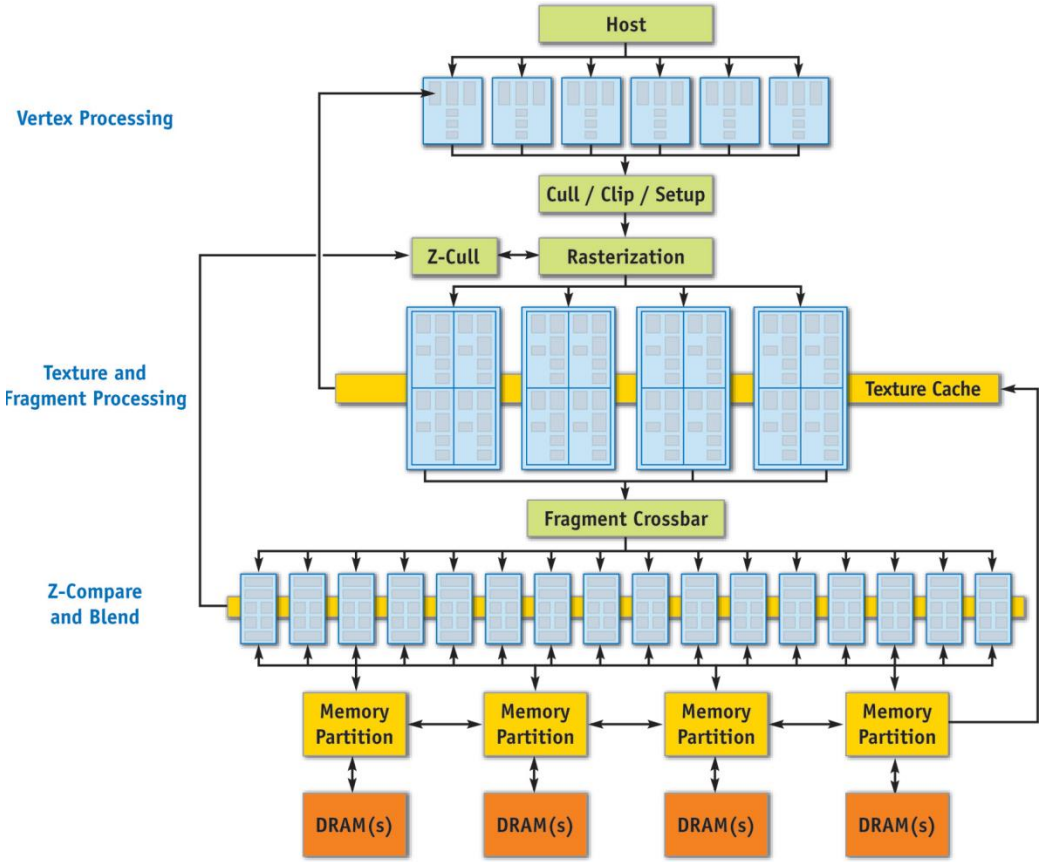


OpenGL pipeline

To first order, DirectX looks the same!

Graphics pipeline → GPU architecture

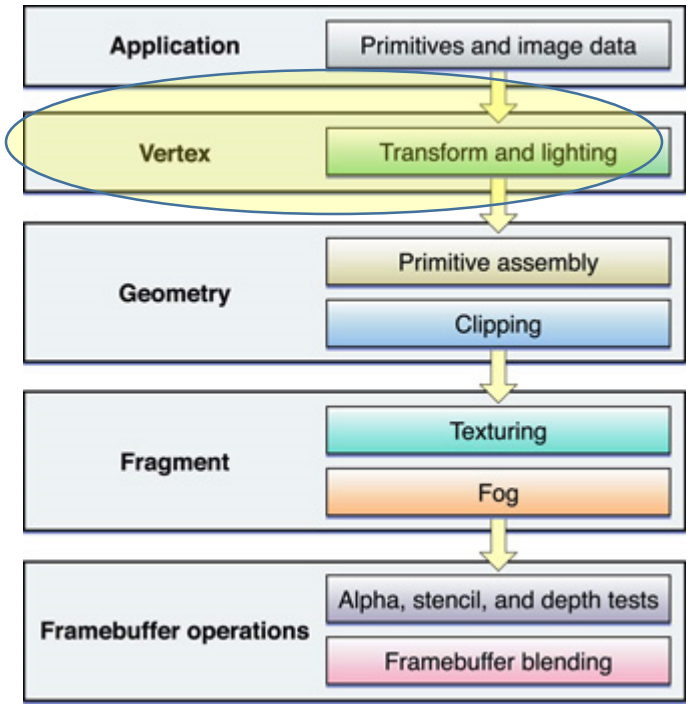


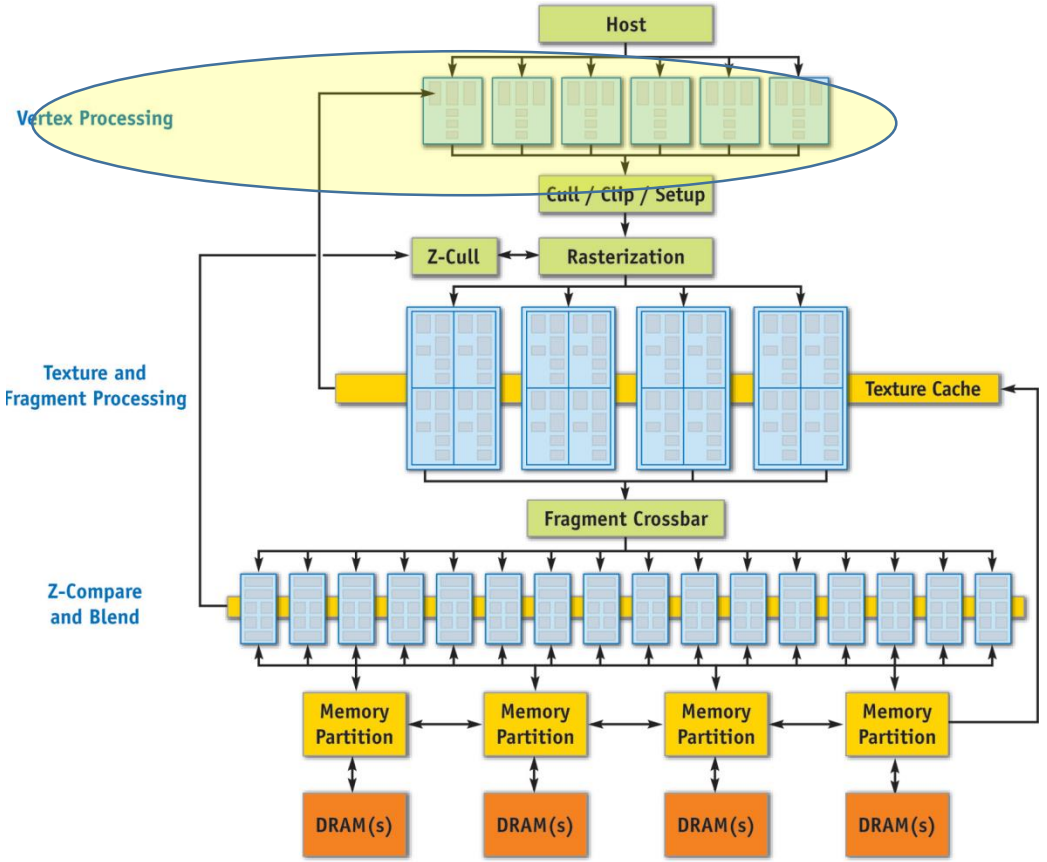


GeForce 6 series

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

Graphics pipeline → GPU architecture

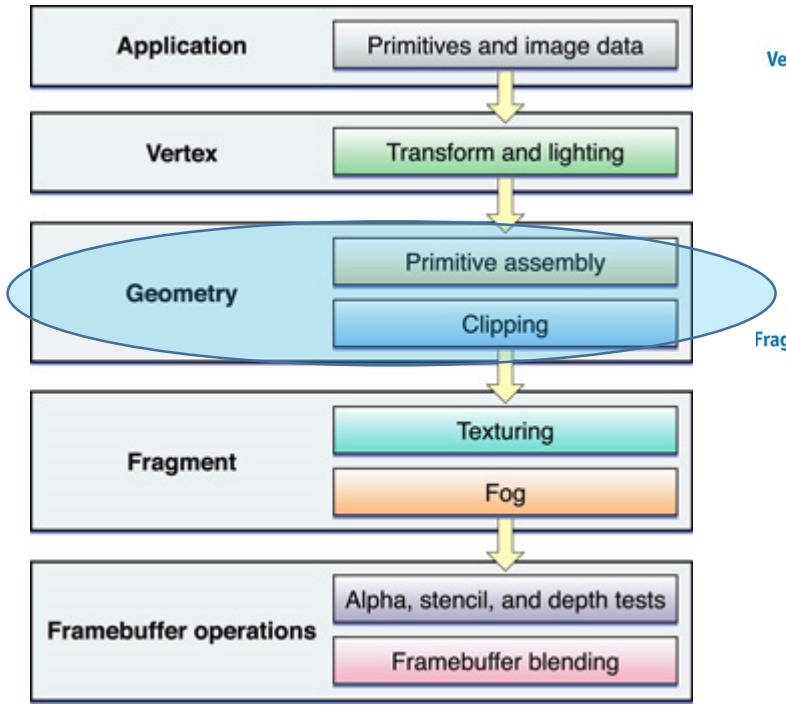


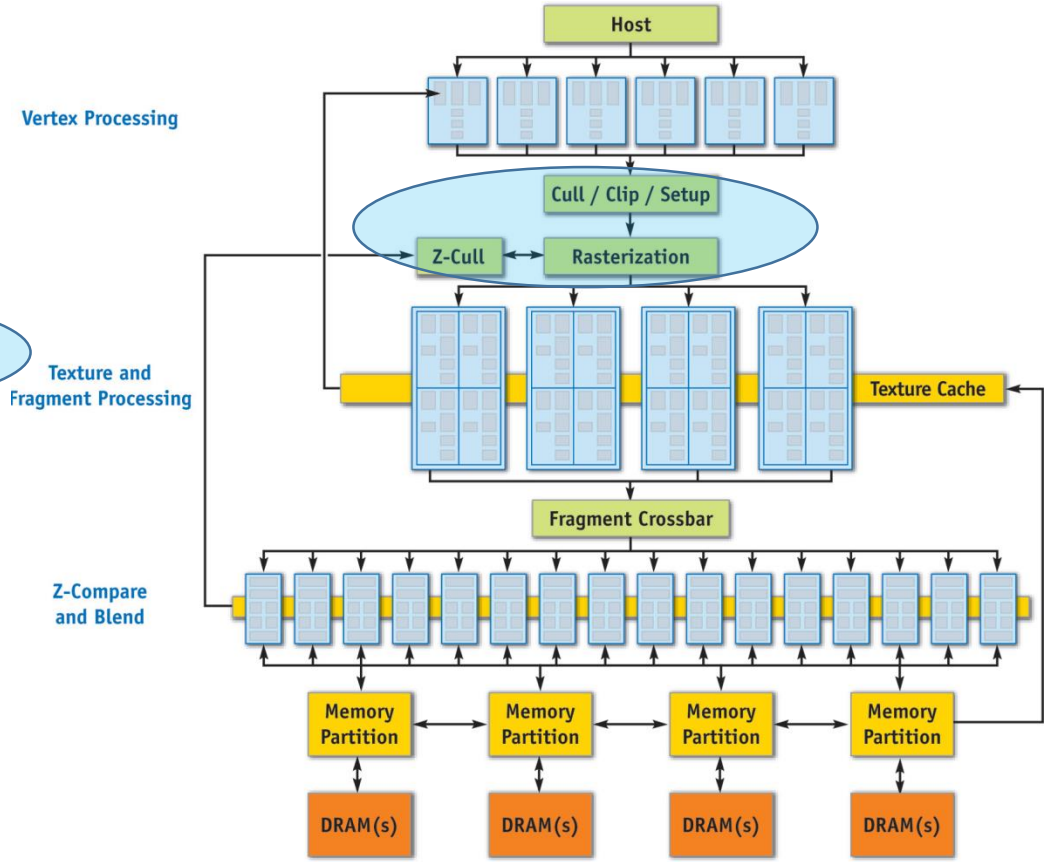


GeForce 6 series

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

Graphics pipeline → GPU architecture

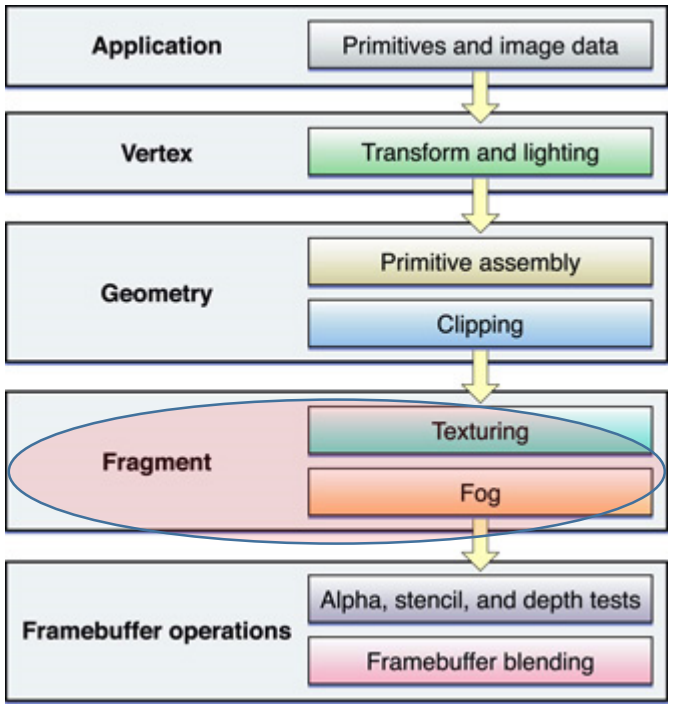


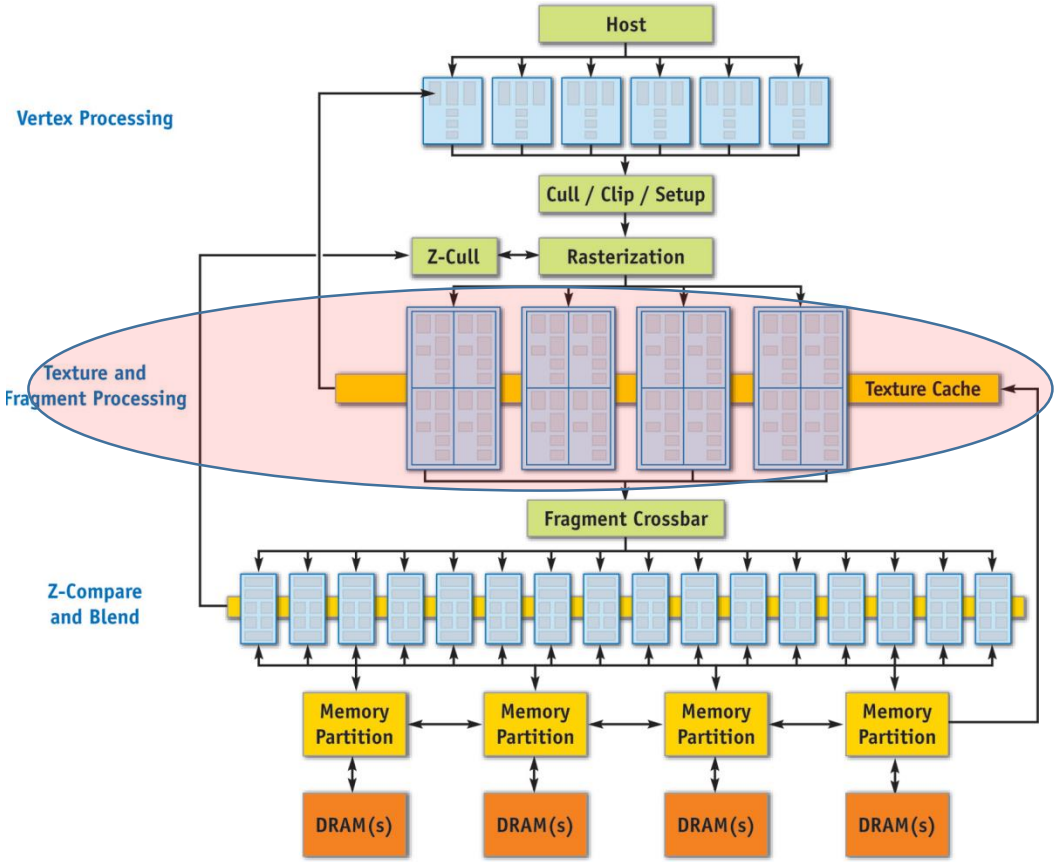


Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series

Graphics pipeline → GPU architecture

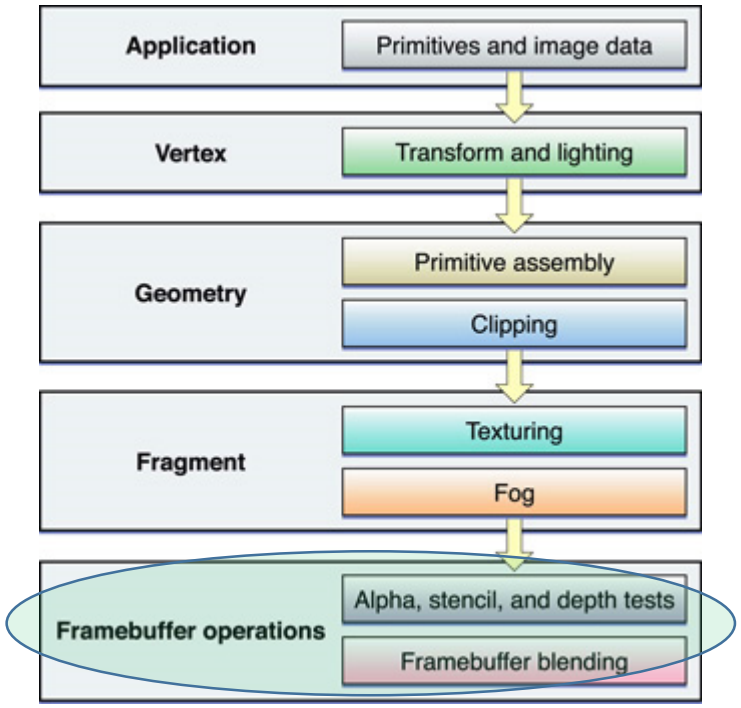


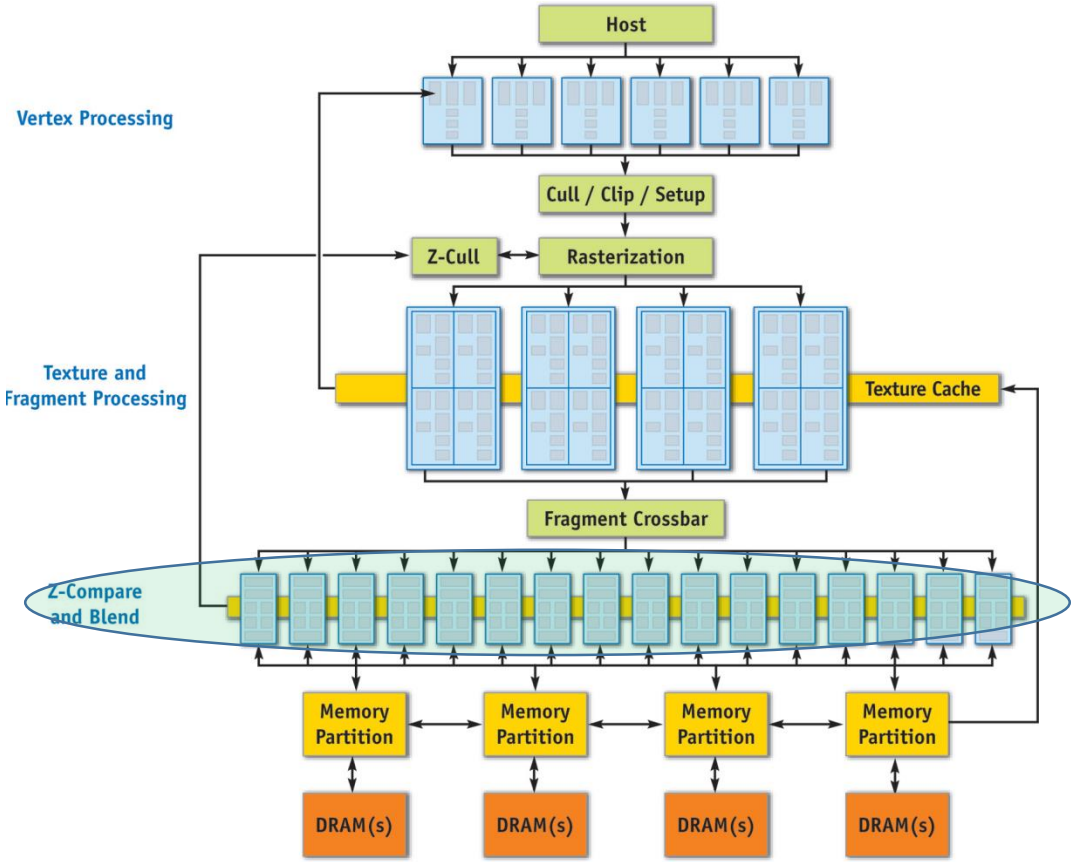


GeForce 6 series

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

Graphics pipeline → GPU architecture

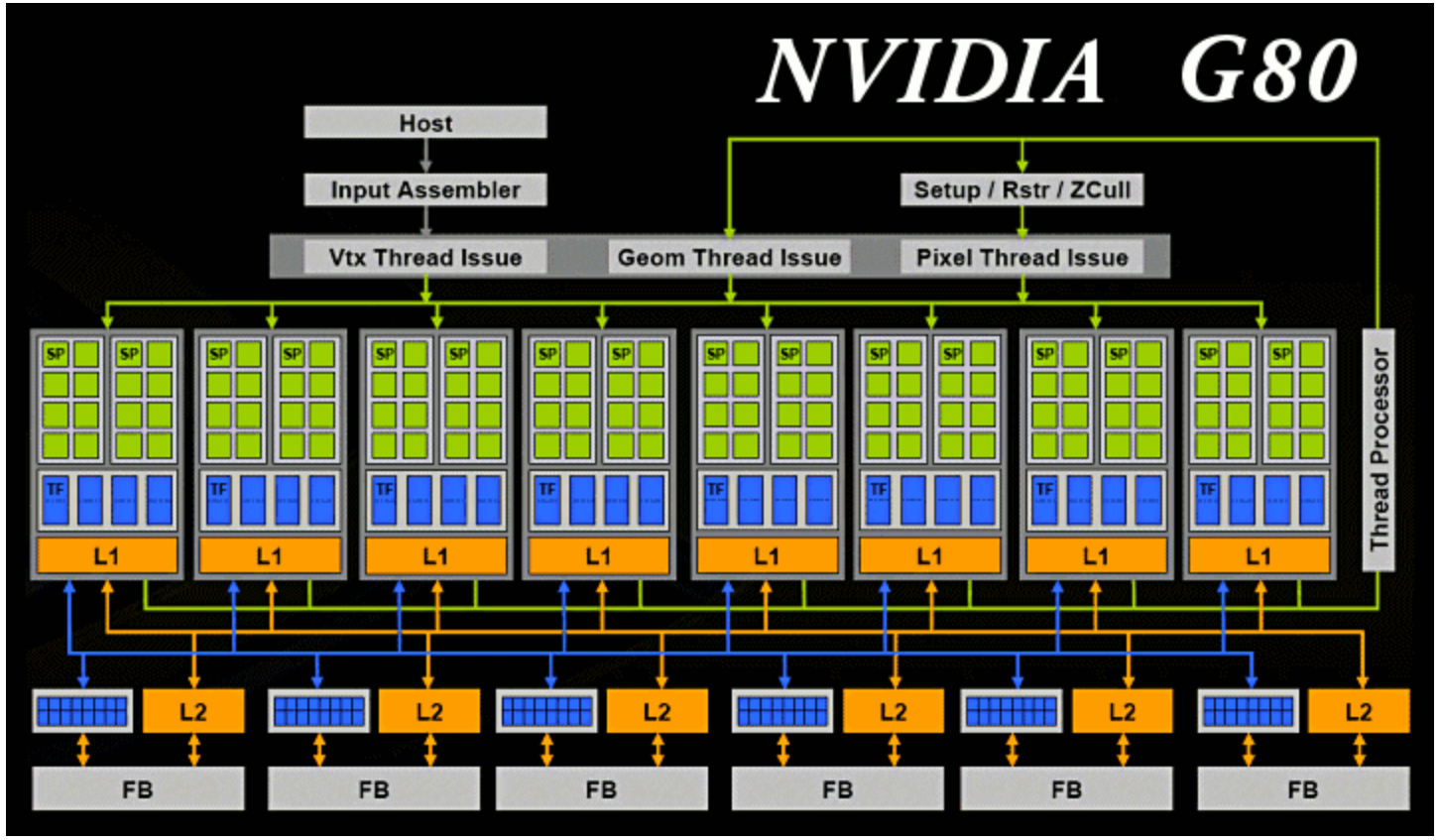




GeForce 6 series

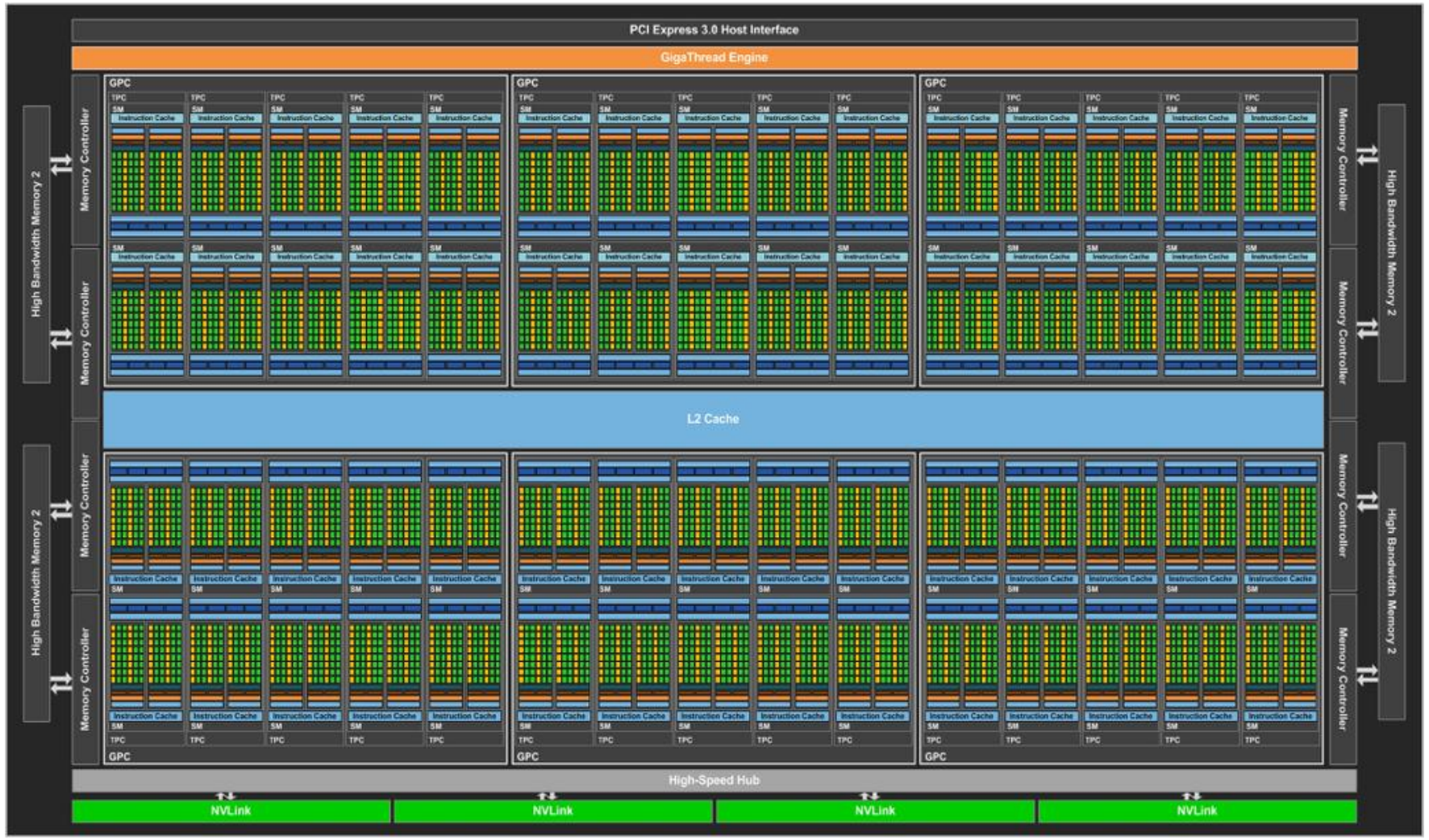
Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

Late Modernity: unified shaders

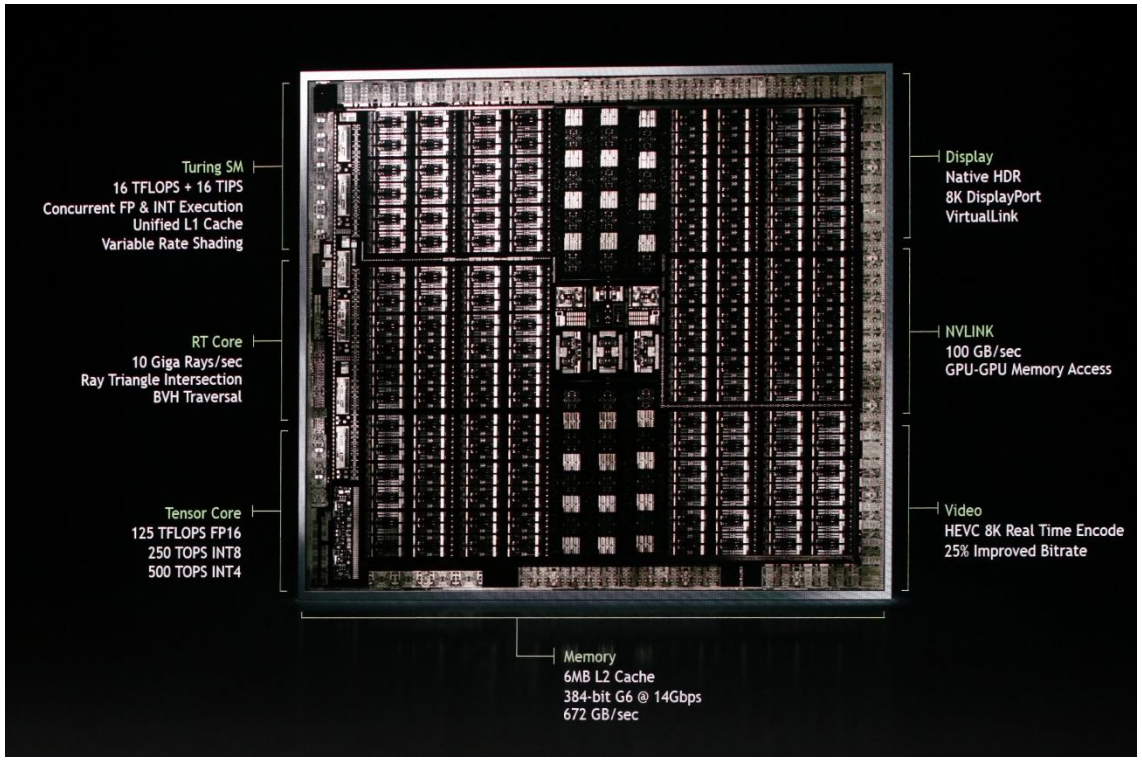


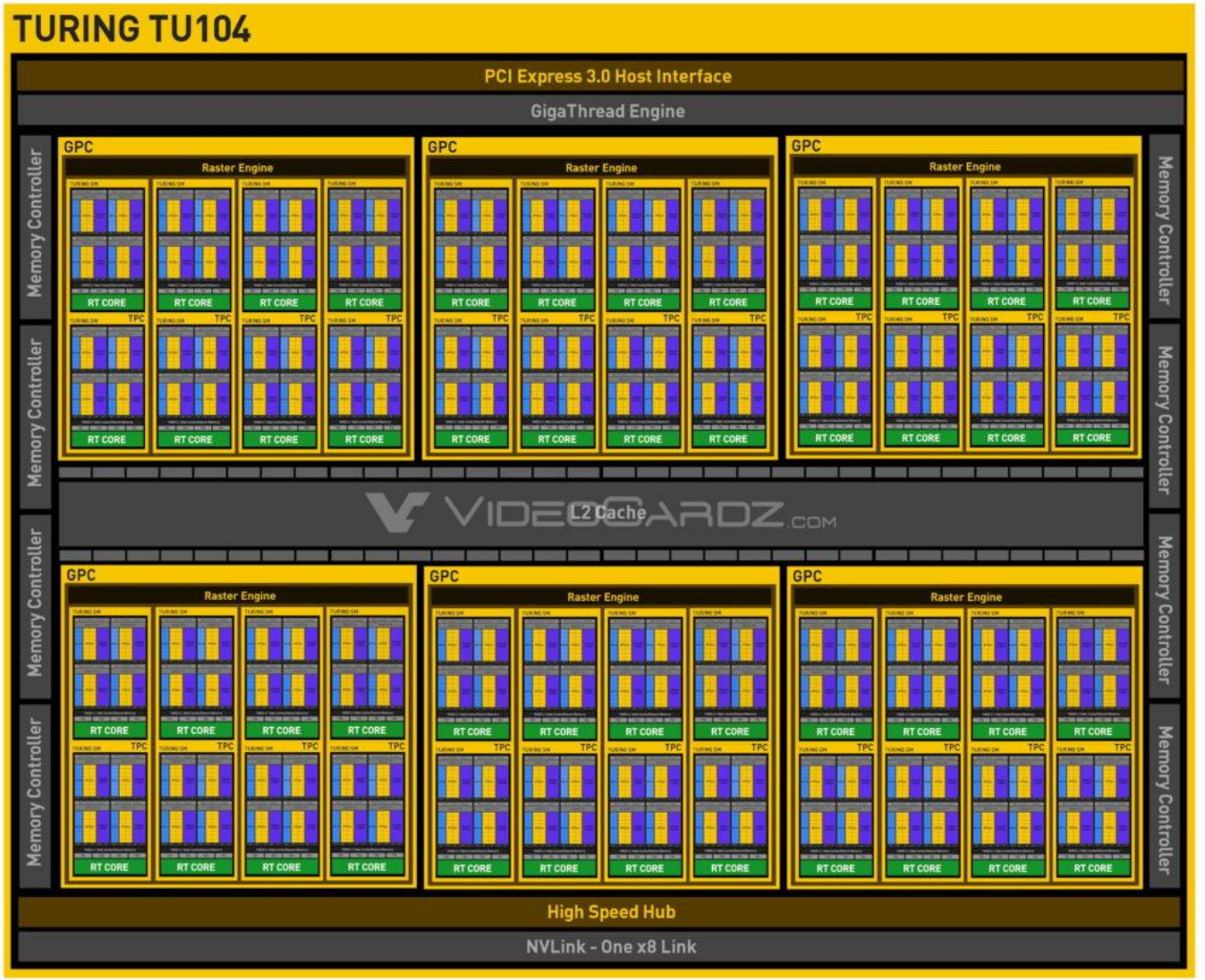
Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count

Mostly Modern: Pascal



Definitely Modern: Turing





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

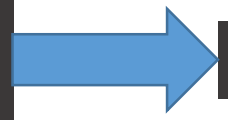
Map

- Inputs
  - Array A
  - Function f(x)
- map(A, f) → 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)
- map(A, f) → apply f(x) on all elements in A
- Parallelism trivially exposed
  - f(x) can be applied in parallel to all elements, in principle

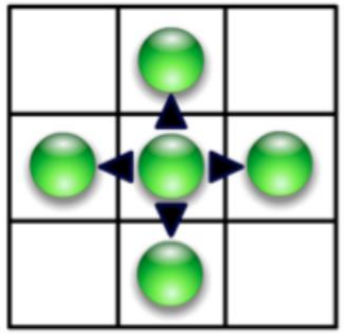
```
for(i=0; i<numPoints; i++) {
   labels[i] = findNearestCenter(points[i]);
}
```



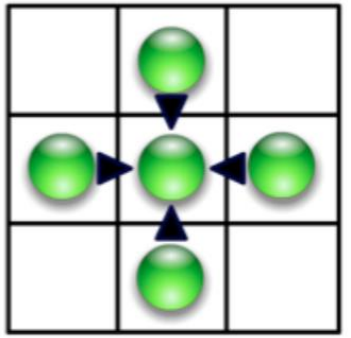
```
map(points, findNearestCenter)
```

Scatter and Gather

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



Scatter



Gather

Scatter and Gather

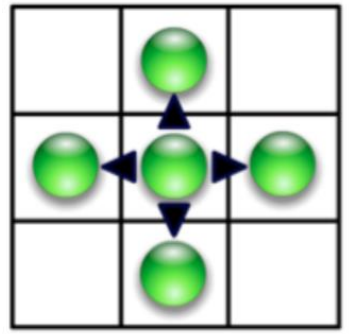
- Gather:
  - Read multiple items to single location
- Scatter:
  - Write single data item to multiple locations

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

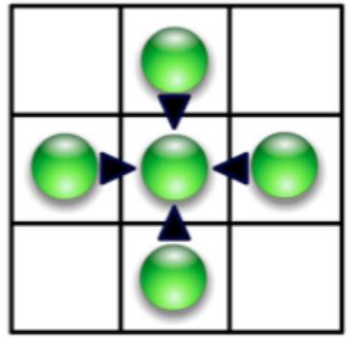
→ gather(x, y, idx)

```
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, … z]
- Reduce(op, s) returns a op b op c … op z

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … 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, … z]
- Reduce(op, s) returns a op b op c … op z

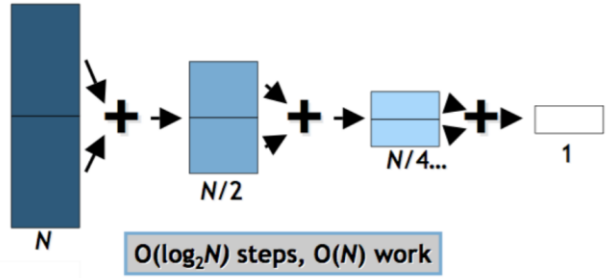
```
for(i=0; i<N; ++i) {
   accum += (point[i]*point[i])
}
```

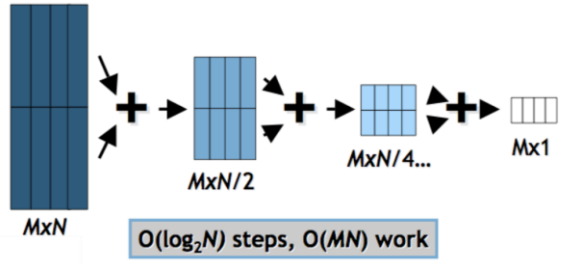
```
accum = reduce(*, point)
```

Why must op be associative?

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z





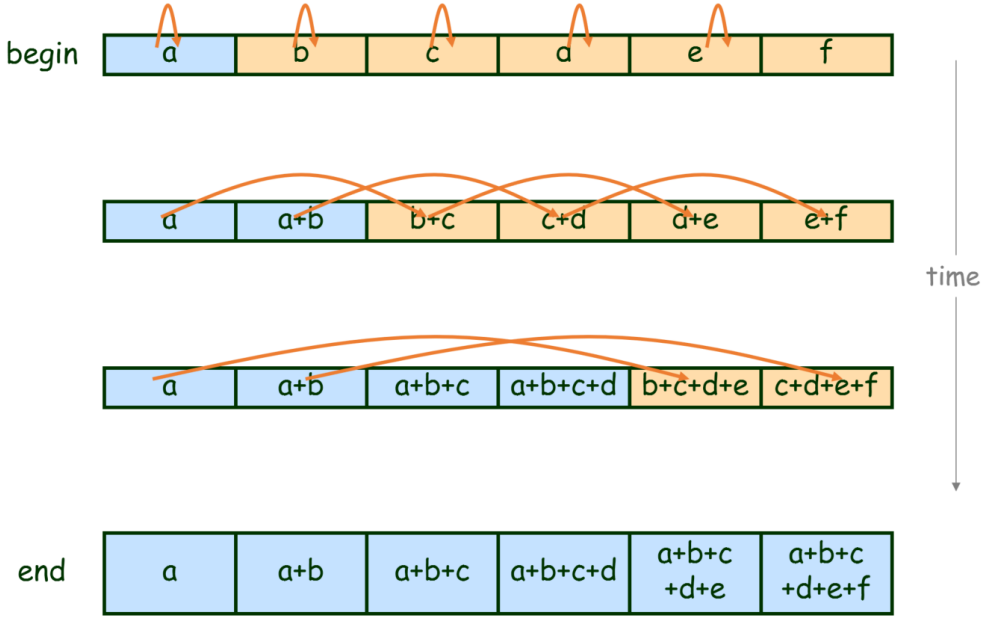

```
for(i=0; i<N; ++i) {
   accum += (point[i]*point[i])
}
```

→ `accum = reduce(*, point)`

Why must op be associative?

Scan (prefix sum)

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
  - Identity I
- scan(op, s) = [I, a, (a op b), (a op b op c) …]
- 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