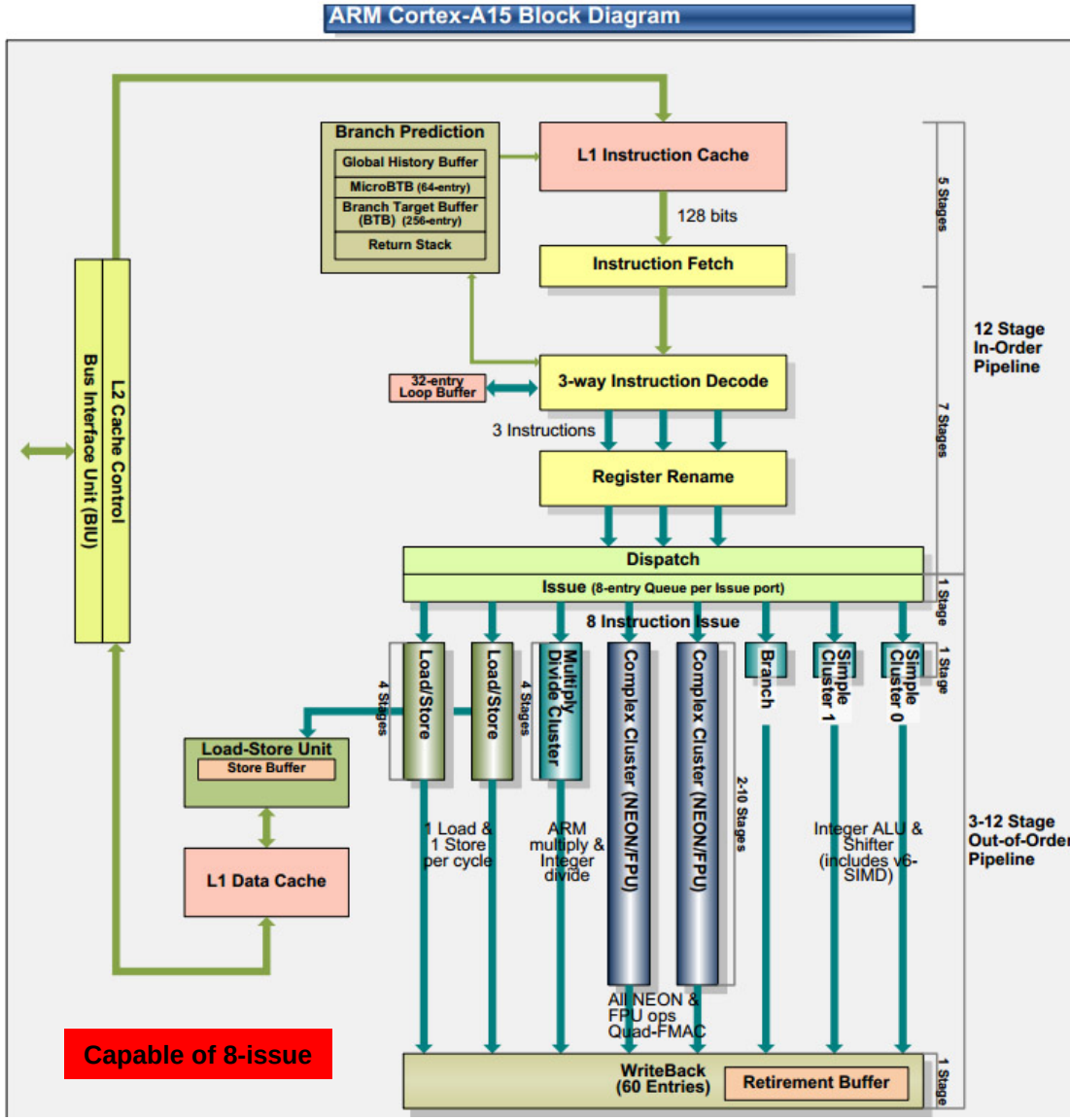
ARM Cortex A15

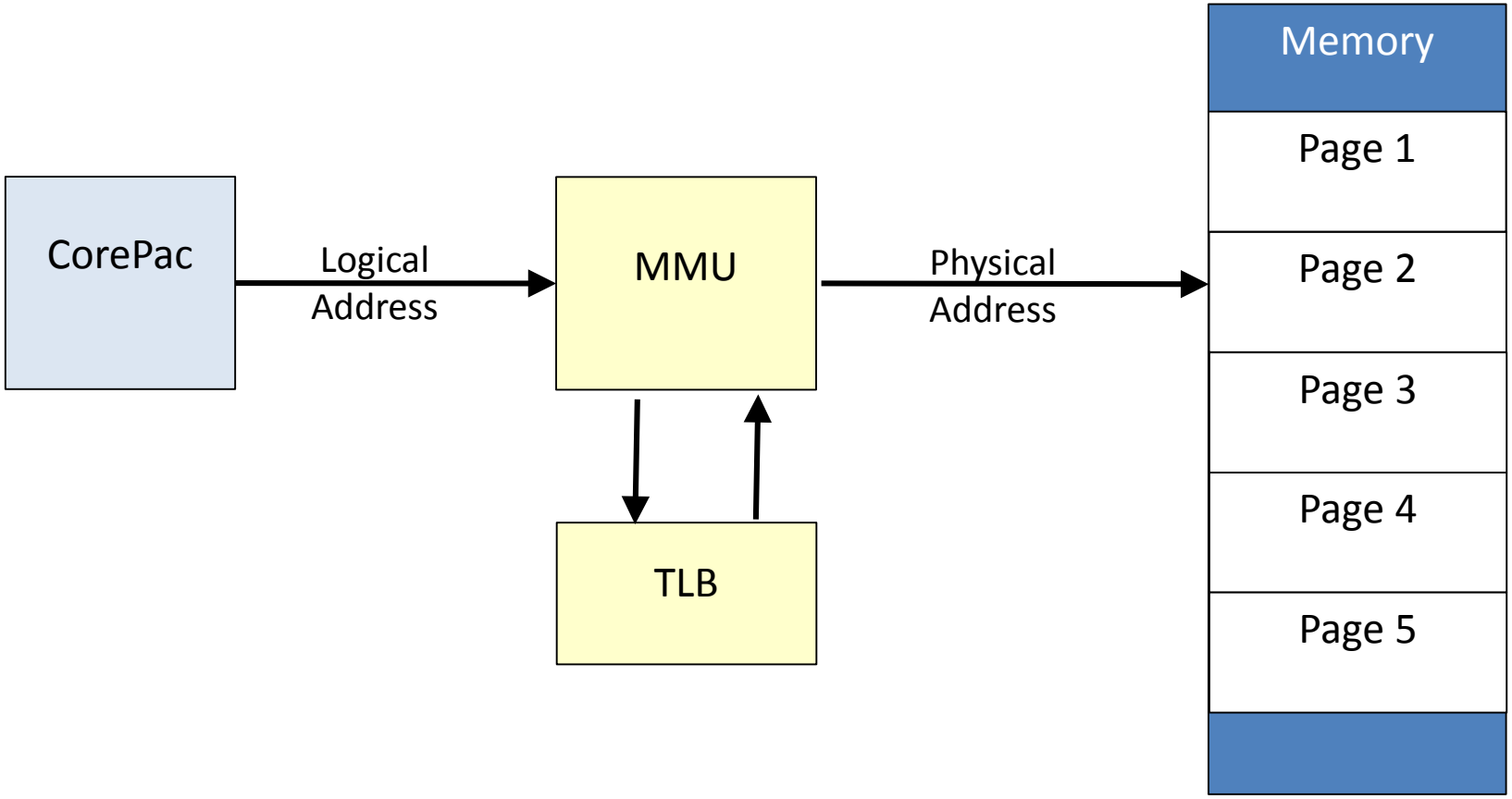
- 2.5Ghz in 28 HP process
  - 12 stage in-order, 3-12 stage OoO pipeline
  - 3.5 DMIPS/Mhz ~ 8750 DMIPS @ 2.5GHz
- ARMv7A with 40-bit PA
  - 1 TB of memory
  - 32-bit limited ARM to 4GB
- Dynamic repartitioning Virtualization
  - Fast state save and restore
  - Move execution between cores/clusters
- 128-bit AMBA 4 ACE bus
- Supports system coherency
- ECC on L1 and L2 caches



Memory Management Unit (MMU)

- Logical-to-physical memory translation:
  - User protected
  - Hardware manages the actual memory
- Large physical addressing; 40-bit (1TB)
- Three-level data structure for virtual 4kB page:
  - Two levels for virtual 2MB pages (Linux huge pages)
  - Translation Lookaside Buffers (TLB) cache one page of address translations per entry to speed up the translation process:
    - L1 instruction access
    - L1 data access
    - L2 TLB

MMU, TLB, and Page

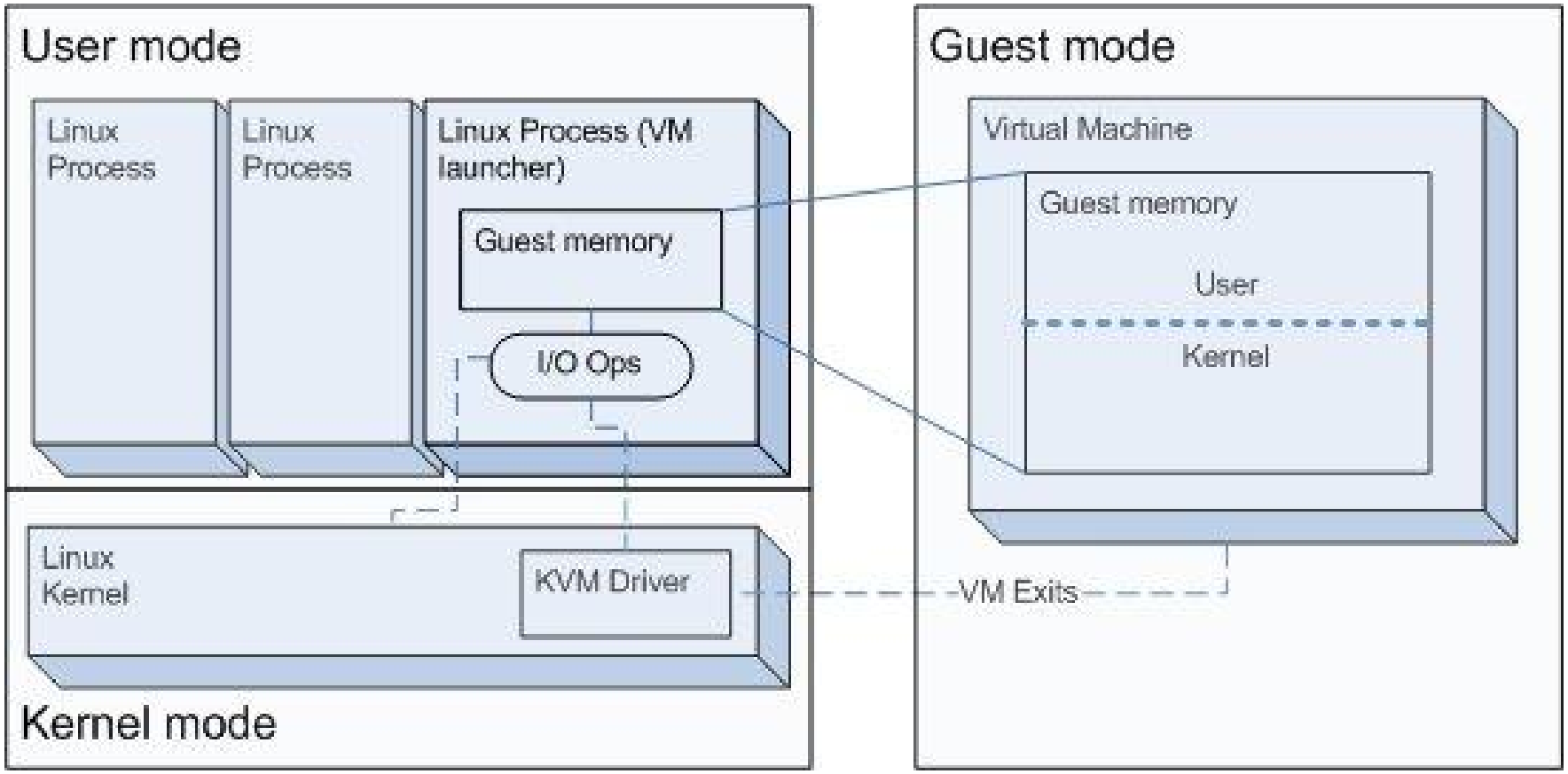


Memory Management Unit (MMU)

To support multiple operating systems (adding a Guest operating system):

- Three privilege layers:
  - User Mode is for “Guest” (application)
  - Supervisor controls multiple guests
  - Hypervisor controls the complete system
- Two-stage translation:
  - From logical to intermediate physical address for supervisor *for each operating system*
  - From intermediate to real address for hypervisor *for the complete system*

Two-Stage MMU: Guest to Supervisor



Two-Stage MMU: Stage One

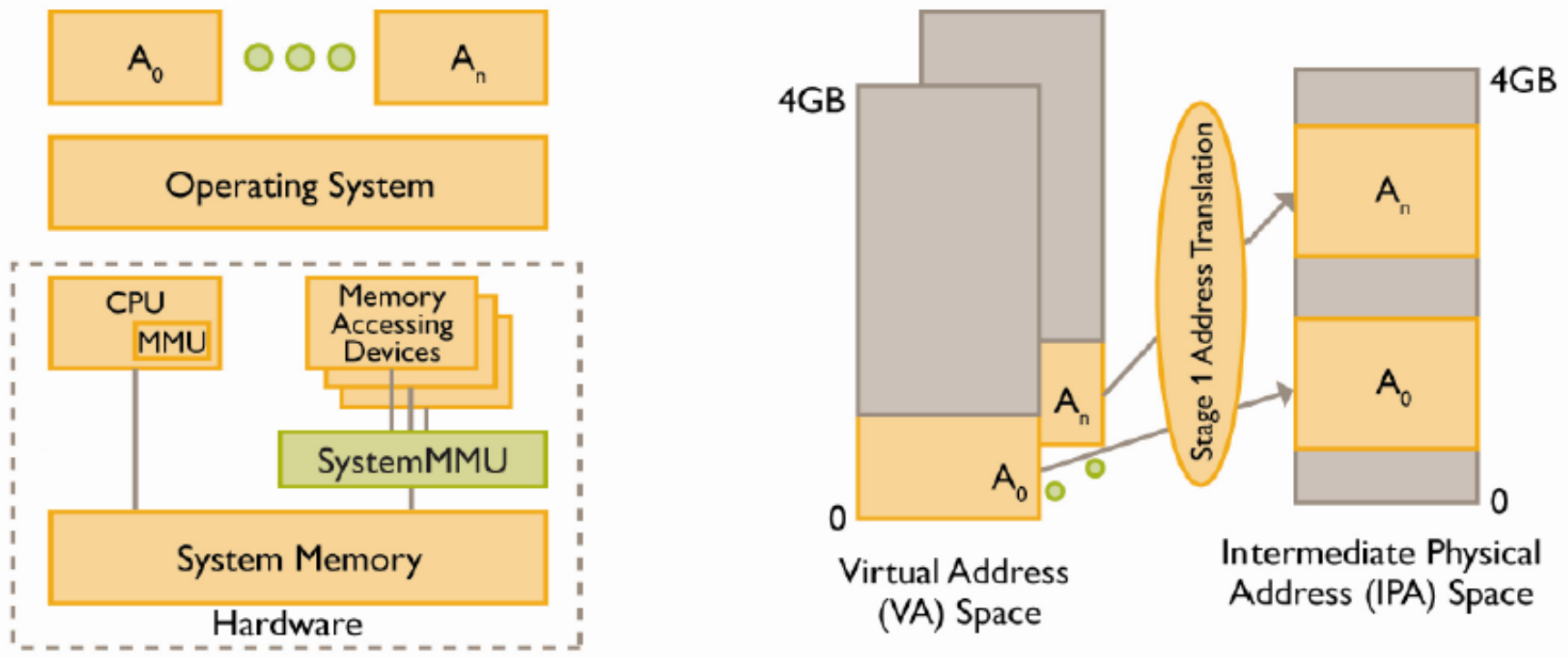


Figure 7 - Application and functionality of the Stage 1 Address Translation

Source: [Virtualization is Coming to a Platform Near You]

Two-Stage MMU: Stage Two

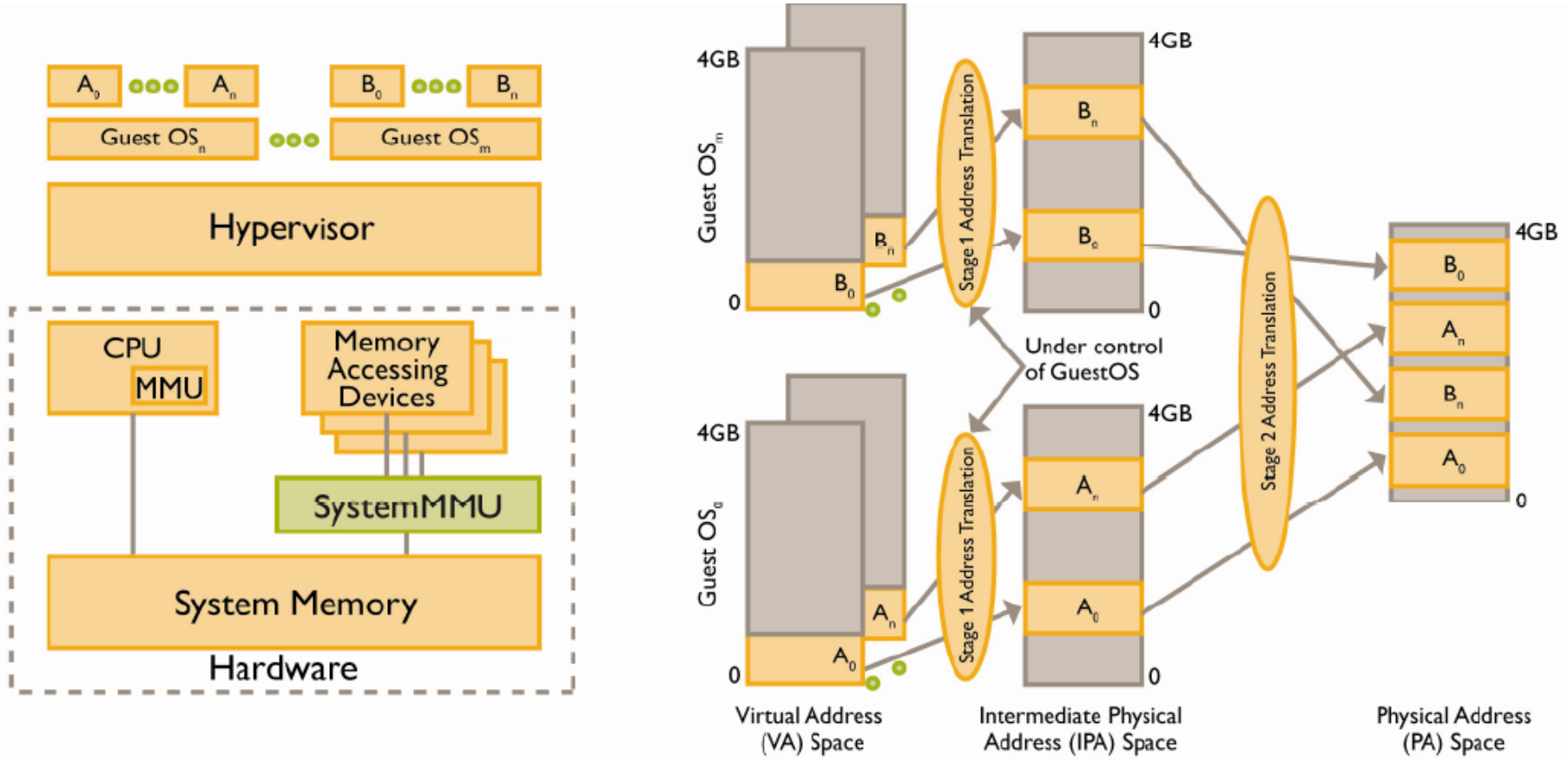


Figure 9 - Application and functionality of the Stage 2 Address Translation

Source: [Virtualization is Coming to a Platform Near You]

Performance from High Frequency

- **Give RAMs as much time as possible**
  - Majority of cycle dedicated to RAM for access
  - Make positive edge based to ease implementation

- **Balance timing of critical "loops" that dictate maximum frequency**
  - Microarchitecture loop:
    - Key function designed to complete in a cycle (or a set of cycles)
      - cannot be further pipelined (with high performance)
- Some example loops:
  - Register Rename allocation and table update
  - Result data and tag forwarding (ALU->ALU, Load->ALU)
  - Instruction Issue decision
  - Branch prediction determination

Feasibility work showed critical loops balancing at about 15-16 gates/clk

ISA Extensions

Instructions added to Cortex-A15 (and all subsequent Cortex-A cores)

- Integer Divide
  - Similar to Cortex-R, M class (driven by automotive)
  - Use getting more common
- Fused MAC
  - Normalizing and rounding once after MUL and ADD
  - Greater accuracy
  - Requirement for IEEE compliance
  - New instructions to complement current chained multiply + add

Hypervisor Debug

- Monitor-mode, watchpoints, breakpoints

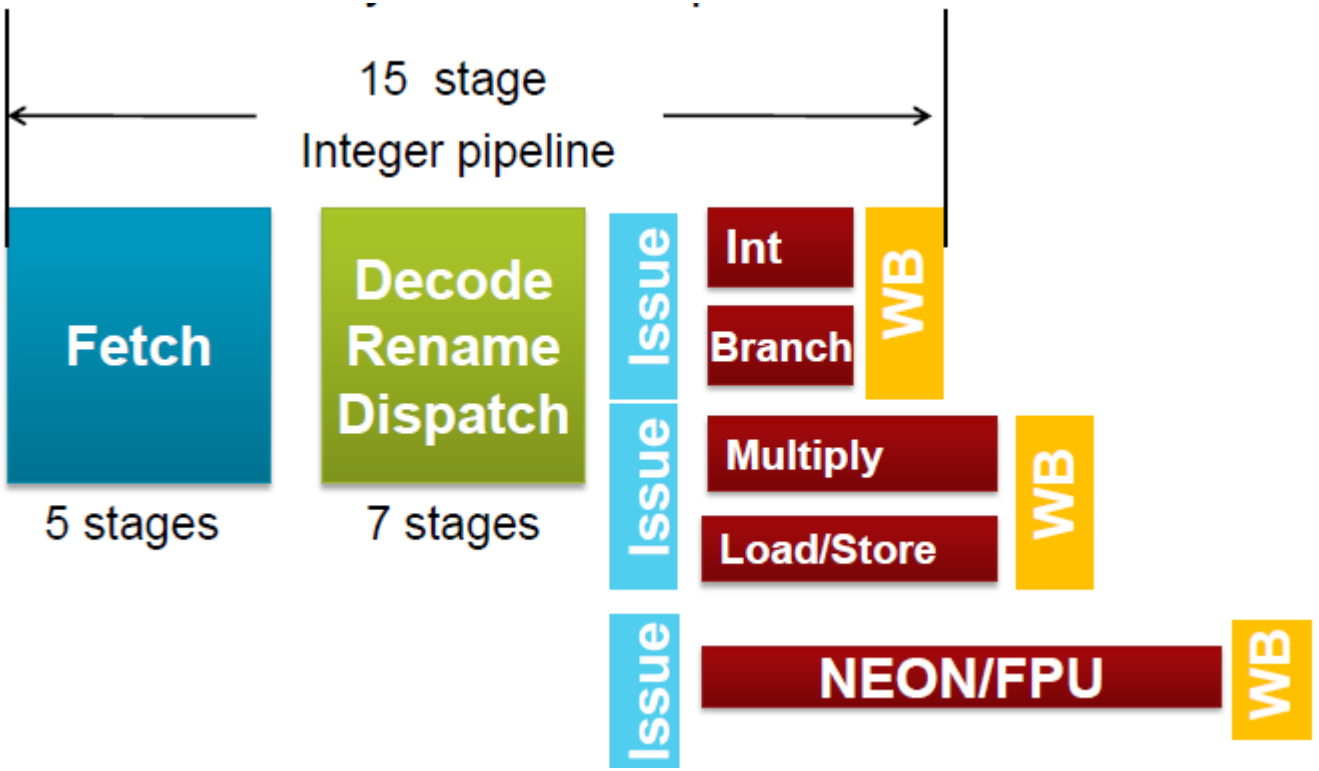
Performance from IPC

- Improved branch prediction
- Wider pipelines for higher instruction throughput
- Larger instruction window for out-of-order execution
- More instruction types can execute out-of-order
- Tightly integrated/low latency NEON and Floating Point Units
- Improved floating point performance
- Improved memory system performance

A15 Pipeline Overview

15-Stage Integer Pipeline

- 4 extra cycles for multiply, load/store
- 2-10 extra cycles for complex media instructions



Improving Branch Handling

- **Similar predictor style to Cortex-A8 and Cortex-A9:**
  - Large target buffer for fast turn around on address
  - Global history buffer for taken/not taken decision
- **Global history buffer enhancements**
  - 3 arrays: Taken array, Not taken array, and Selector
- **Indirect predictor**
  - 256 entry BTB indexed by XOR of history and address
  - Multiple Target addresses allowed per address
- **Out-of-order branch resolution:**
  - Reduces the mispredictpenalty
  - Requires special handling in return stack

Improving Fetch Bandwidth

Increased fetch from 64-bit to 128-bit

- Full support for unaligned fetch address
- Enables more efficient use of memory bandwidth
- Only critical words of cache line allocated

Addition of microBTB

- Reduces bubble on taken branches
- 64 entry target buffer for fast turn around prediction
- Fully associative structure
- Caches taken branches only
- Overruled by main predictor when they disagree

OOO Execution – Register Renaminc

Two main components to register renaming

- Register rename tables
  - Provides current mapping from architected registers to result queue entries
  - Two tables: one each for ARM and Extended (NEON) registers
- Result queue
  - Queue of renamed register results pending update to the register file
  - Shared for both ARM and Extended register results

The rename loop

- Destination registers are always renamed to top entry of result queue
  - Rename table updated for next cycle access
- Source register rename mappings are read from rename table
  - Bypass muxes,present to handle same cycle forwarding
- Result queue entries reused when flushed or retired to architectural state

Boosting OOO Execution

Out-of-order execution improves performance by executing past hazards

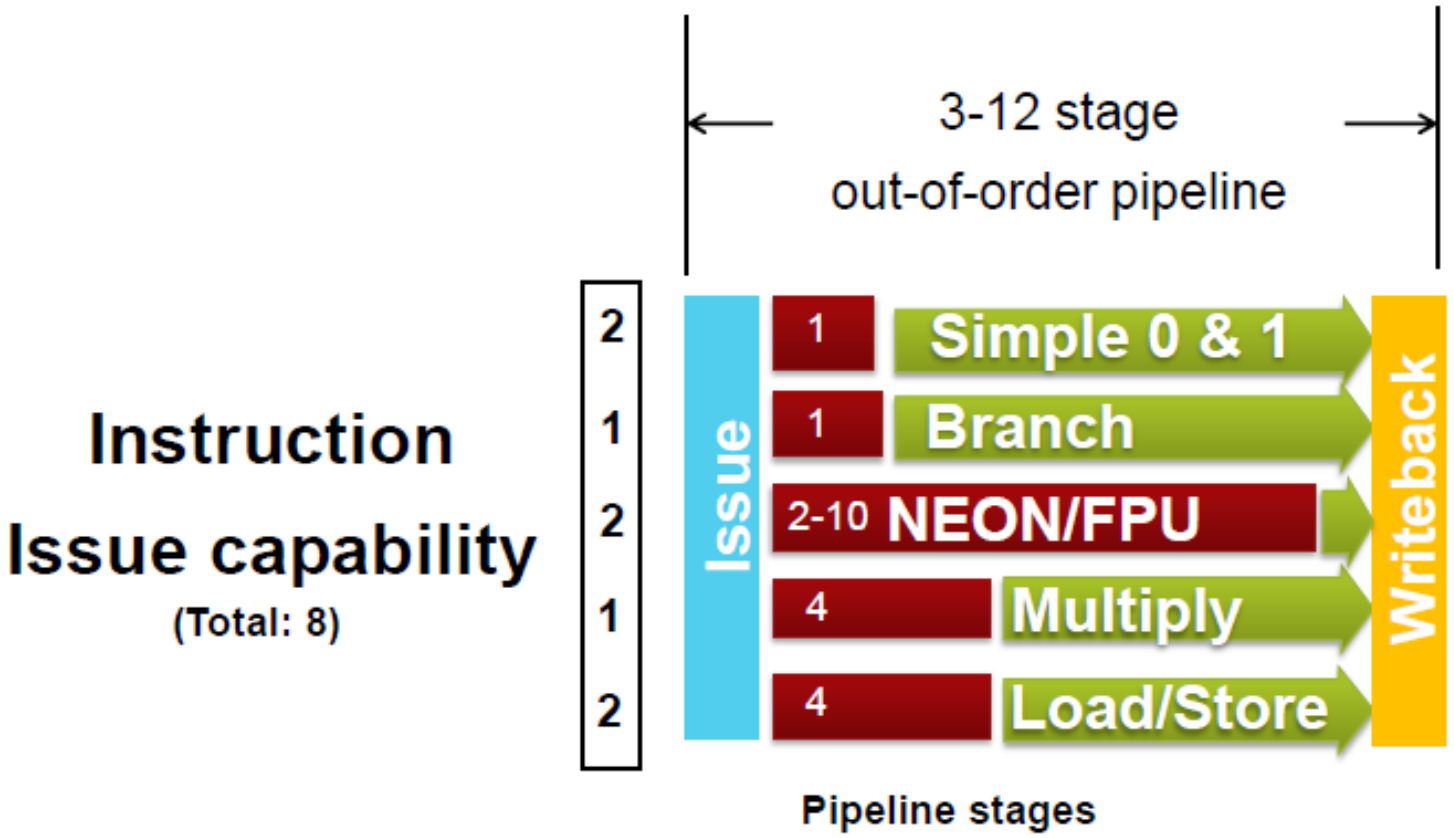
- Effectiveness limited by how far you look ahead
  - Window size of 40+ operations required for Cortex-A15 performance targets
- Issue queue size often frequency limited to 8 entries

Solution: multiple smaller issue queues

- Execution broken down to multiple clusters defined by instruction type
- Instructions dispatched 3 per cycle to the appropriate issue queue
- Issue queues each scanned in parallel

A-15 Execution Clusters

- Each cluster can have multiple pipelines
- Clusters have separate/independent issuing capability



Execution Clusters Overview

- **Simple cluster**
  - Single cycle integer operations
  - 2 ALUs, 2 shifters (in parallel, includes v6-SIMD)
- **Complex cluster**
  - All NEON and Floating Point data processing operations
  - Pipelines are of varying length and asymmetric functions
  - Capable of quad-FMAC operation
- **Branch cluster**
  - All operations that have the PC as a destination
- **Multiply and Divide cluster**
  - All ARM multiply and Integer divide operations
- **Load/Store cluster**
  - All Load/Store, data transfers and cache maintenance operations
  - Partially out-of-order, 1 Load and 1 Store executed per cycle
  - Load cannot bypass a Store, Store cannot bypass a Store

FP and NEON performance

Dual issue queues of 8 entries each

- Can execute two operations per cycle
- Includes support for quad FMAC per cycle

Fully integrated into main Cortex-A15 pipeline

- Decoding done upfront with other instruction types
- Shared pipeline mechanisms
- Reduces area consumed and improves interworking

Specific challenges for Out-of-order VFP/Neon

- Variable length execution pipelines
- Late accumulator source operand for MAC operations

SIMD Engine NEON

- 64/128-bit data instructions
- Fully integrated into the main pipeline
- 32x 64-bit registers that can be arranged as 128-bit registers
- Data can be interpreted as follows:
  - Byte
  - Half-word (16-bit)
  - Word
  - Long

NEON Registers

NEON registers load and store data into 64-bit registers from memory with on-the-fly interleave, as shown in this diagram.

Interleaving

Many instructions in this group provide interleaving when structures are stored to memory, and de-interleaving when structures are loaded from memory. Figure 14 shows an example of de-interleaving. Interleaving is the inverse process.

Figure 14. De-interleaving an array of 3-element structures

Source: ARM Compiler Toolchain Assembler Reference; DUI0489C

Vector Floating Point (VFP)

- Fully integrated into the main pipeline
- 32 DP registers for FP operations
- Native (hardware) support for all IEEE-defined floating-point operations and rounding modes; Single- and double-precision
- Supports fused MAC operation (e.g., rounding after the addition or after the multiplication)
- Supports half-precision (IEEE754-2008); 1-bit sign, 5-bit exponent, 10-bit mantissa

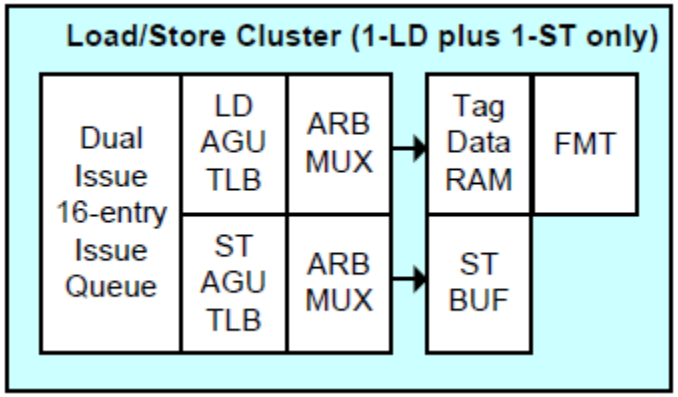
Load/Store Cluster

16 entry issue queue for loads and stores

- Common queue for ARM and NEON/memory operations
- Loads issue out-of-order but cannot bypass stores
- Stores issue in order, **but only require address sources to issue**

4 stage load pipeline

- 1st: Combined AGU/TLB structure lookup
- 2nd: Address setup to Tag and data arrays
- 3rd: Data/Tag access cycle
- 4th: Data selection, formatting, and forwarding



Store operations are AGU/TLB look up only on first pass

- Update store buffer after PA is obtained
- Arbitrate for Tag RAM access
- Update merge buffer when non-speculative
- Arbitrate for Data RAM access from merge buffer

The Level 2 Memory systems

- **Cache characteristics**
  - 16 way cache with sequential TAG and Data RAM access
  - Supports sizes of 512kB to 4MB
  - Programmable RAM latencies
- **MP support**
  - 4 independent Tag banks handle multiple requests in parallel
  - Integrated Snoop Control Unit into L2 pipeline
  - Direct data transfer line migration supported from CPU-to-CPU
- **External bus interfaces**
  - Full AMBA4 system coherency support on 128-bit master interface
  - 64/128 bit AXI3 slave interface for ACP
- **Other key features**
  - Full ECC capability
  - Automatic data prefetching into L2 cache for load streaming

Other Key features

Supporting fast state save for power down

- Fast cache maintenance operations
- Fast SPR writes: all register state local

Dedicated TLB and table walk machine per CPU

- 4-way 512 entry per CPU
- Includes full table walk machine
- Includes walking cache structures

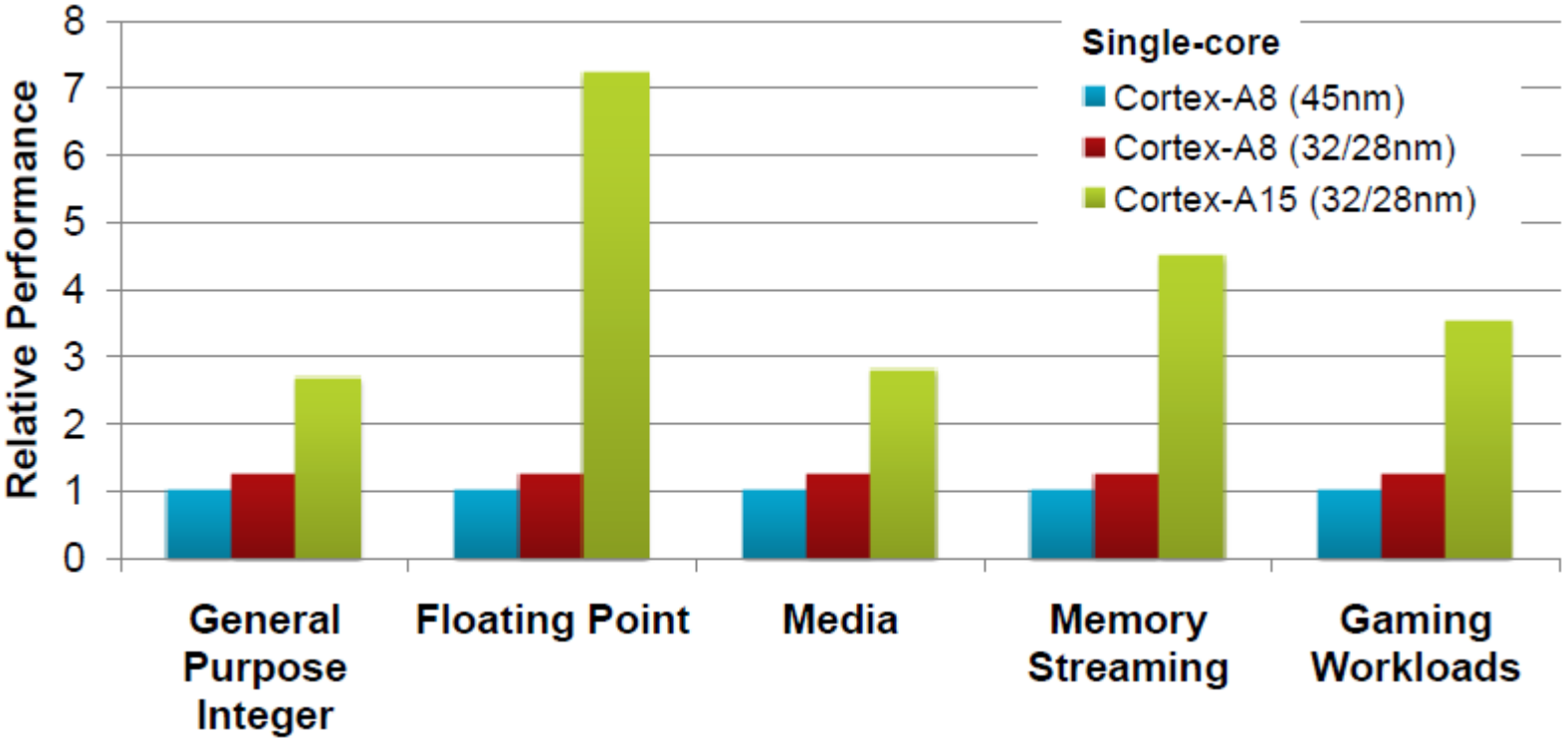
Active power management

- 32 entry loop buffer
- Loop can contain up to 2 fwd branches and 1 backwards branch
- Completely disables Fetch and most of the Decode stages of pipeline

ECC support in software writeable RAMs, Parity in read only RAMs

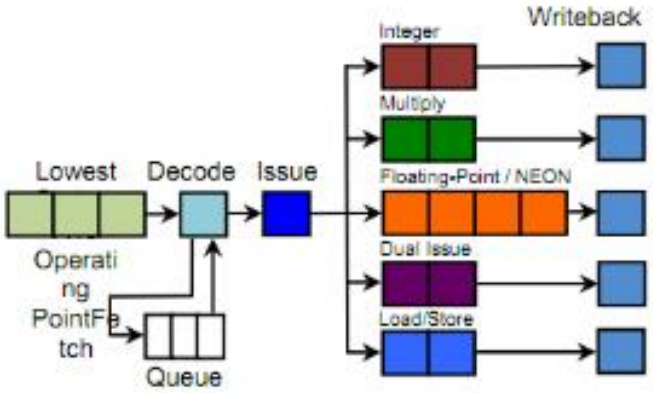
- Supports logging of error location and frequency

Single-thread performance

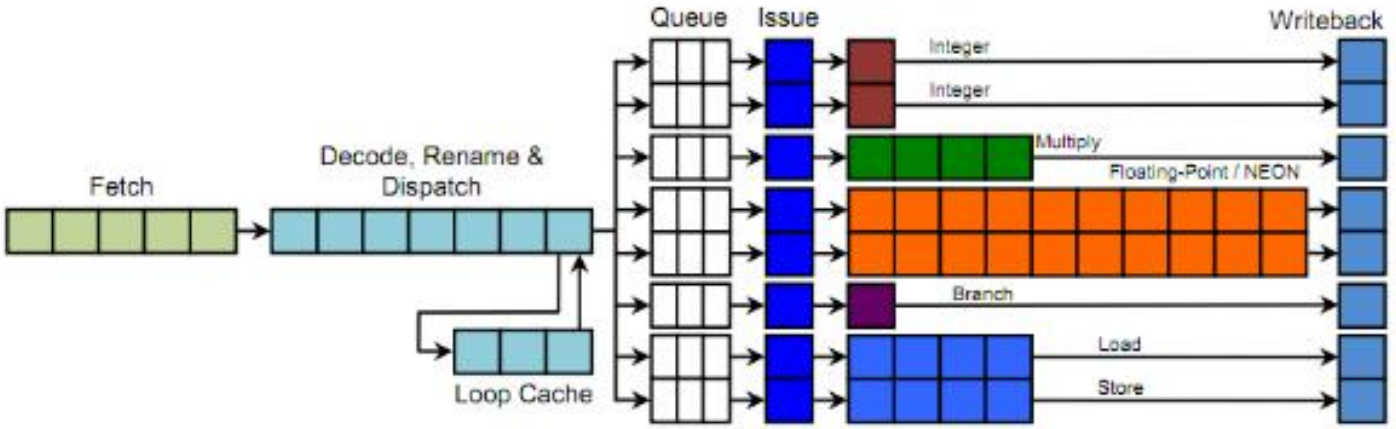


Both processors using 32K L1 and 1MB L2 Caches, common memory system Cortex-A8 andCortex-A15 using 128-bit AXI bus master

Cortex-A7 vs. Cortex-A15



Cortex-A7 Pipeline



Cortex-A15 Pipeline

Processors for Big-Little Architecture (To be discussed later)

Chapter 6

Parallel Processors from Client to Cloud – Part 1

Parallel Processor Architecture: SMT, VECTOR, SIMD

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications
  (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu r4,$s0,#512     ;upper bound of what to load
loop: l.d   $f2,0($s0)      ;load x(i)
      mul.d $f2,$f2,$f0     ;a × x(i)
      l.d   $f4,0($s1)      ;load y(i)
      add.d $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d   $f4,0($s1)      ;store into y(i)
      addiu $s0,$s0,#8      ;increment index to x
      addiu $s1,$s1,#8      ;increment index to y
      subu  $t0,r4,$s0      ;compute bound
      bne   $t0,$zero,loop  ;check if done
```

- Vector MIPS code

```
      l.d     $f0,a($sp)    ;load scalar a
      lv      $v1,0($s0)    ;load vector x
      mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
      lv      $v3,0($s1)    ;load vector y
      addv.d  $v4,$v2,$v3   ;add y to product
      sv      $v4,0($s1)    ;store the result
```

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector vs. Scalar

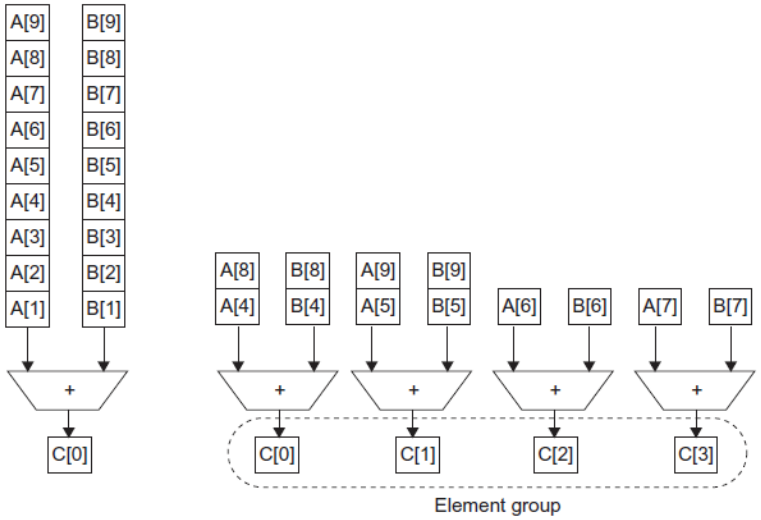
- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

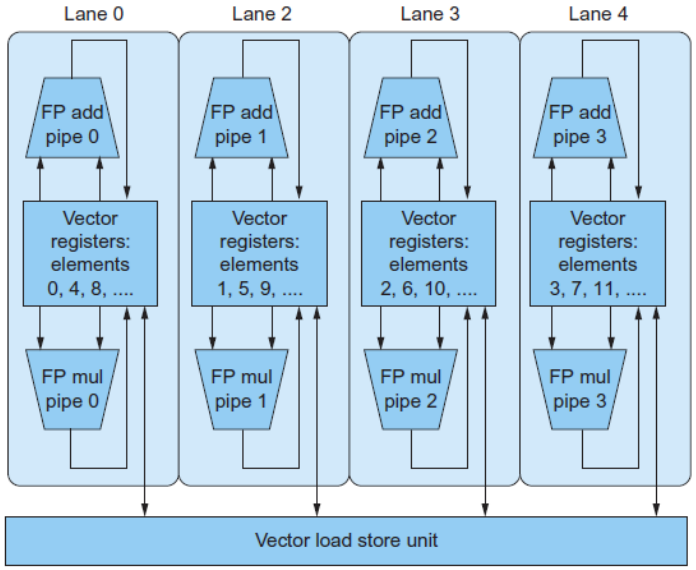
SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:





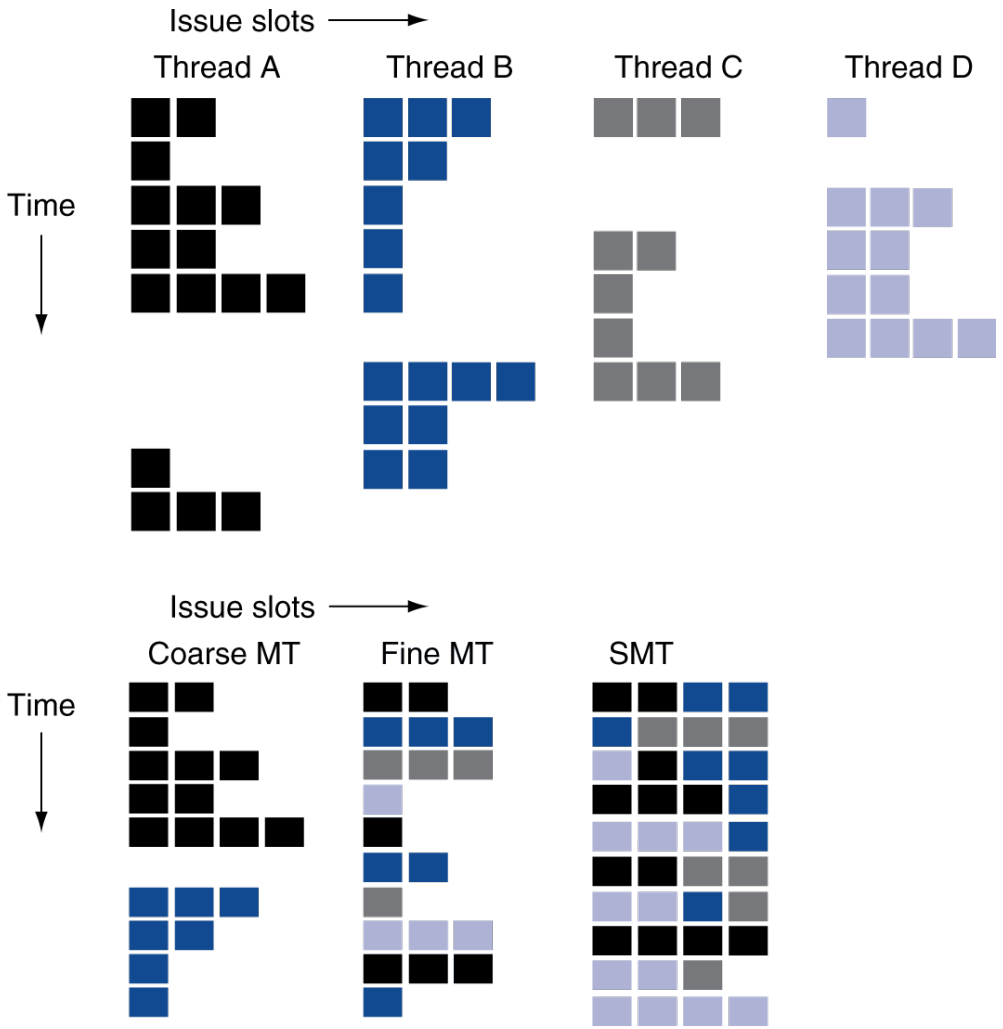
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example



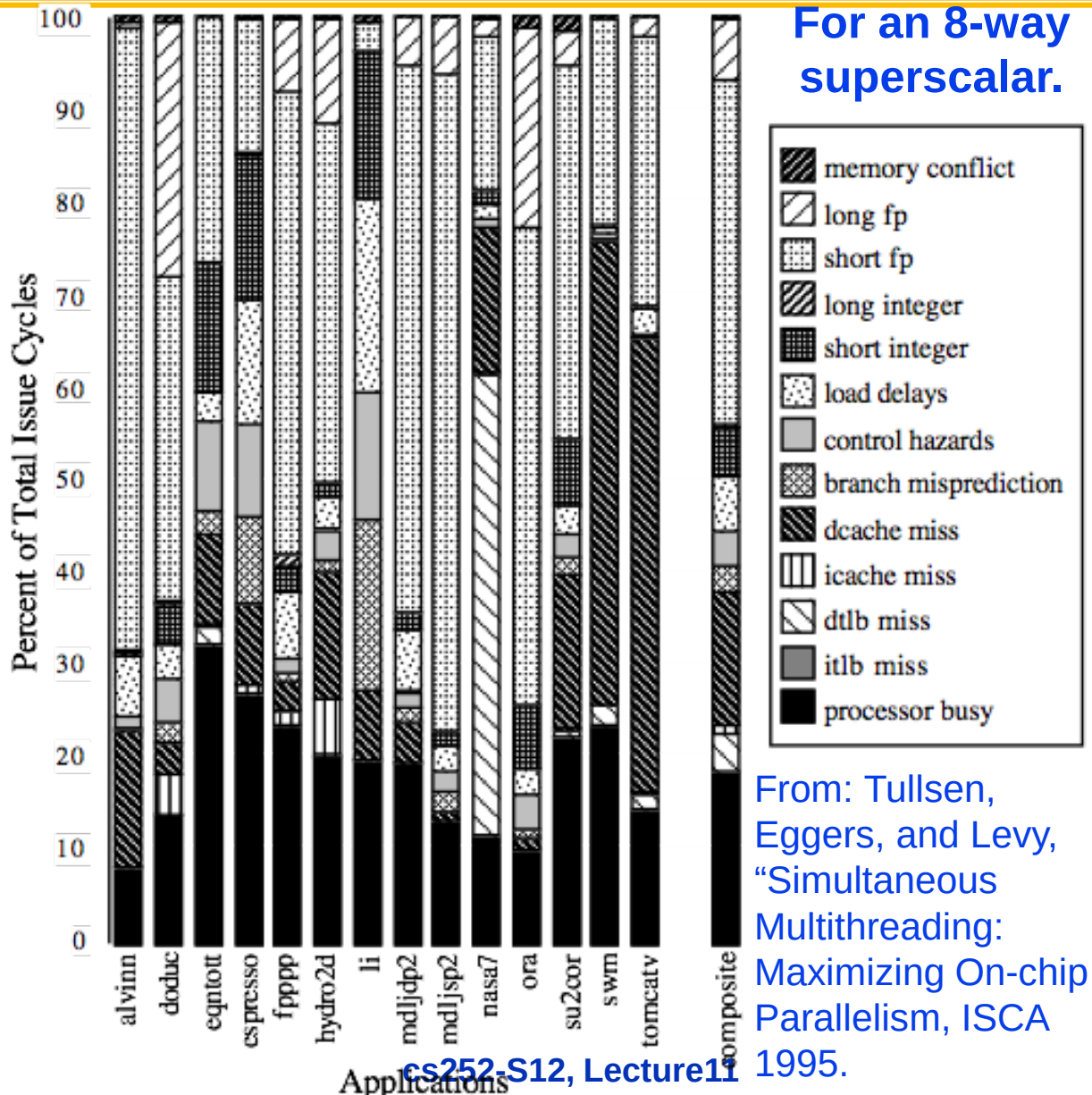
Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara (will see later)

Coarse-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400, Alewife

For most apps: most execution units lie idle



For an 8-way superscalar.

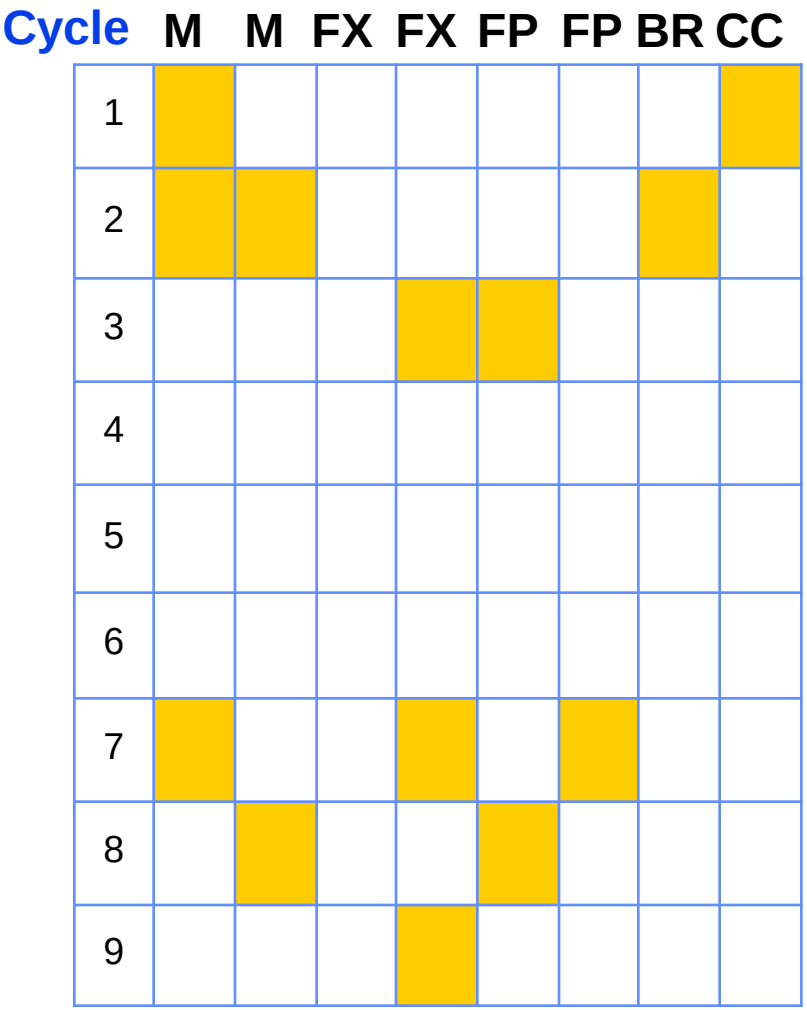
From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Do both ILP and TLP?

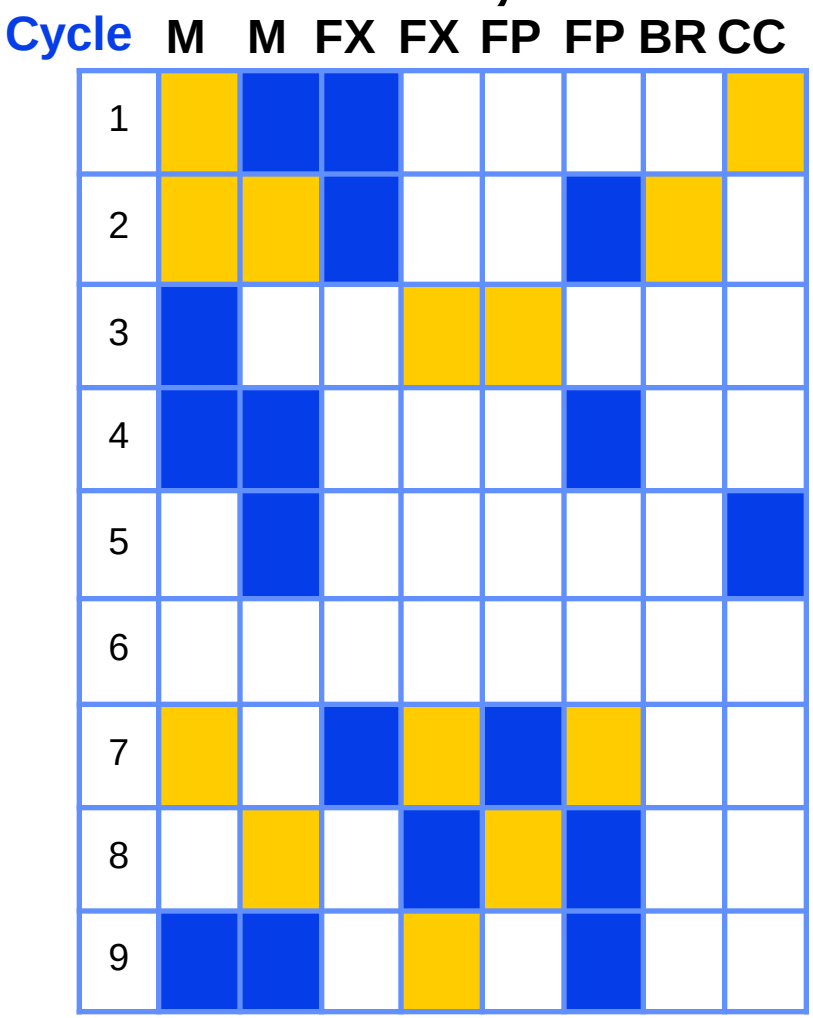
- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Simultaneous Multi-threading ...

One thread, 8 units



Two threads, 8 units



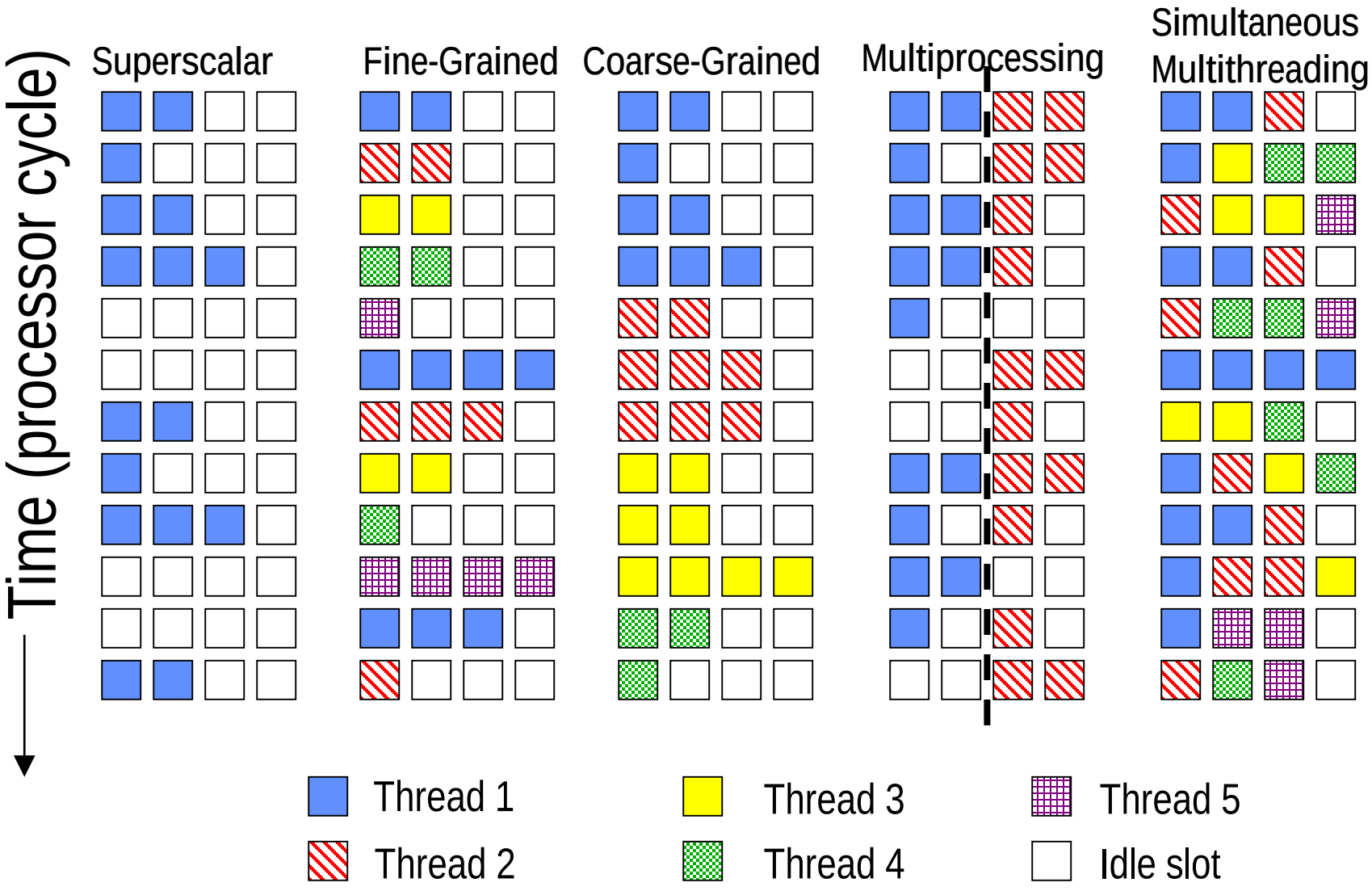
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Simultaneous Multithreading (SMT)

- **Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading**
  - **Large set of virtual registers that can be used to hold the register sets of independent threads**
  - **Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads**
  - **Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW**
- **Just adding a per thread renaming table and keeping separate PCs**
  - **Independent commitment can be supported by logically keeping a separate reorder buffer for each thread**

Source: Micrprocessor Report, December 6, 1999
"Compaq Chooses SMT for Alpha"

Multithreaded Categories

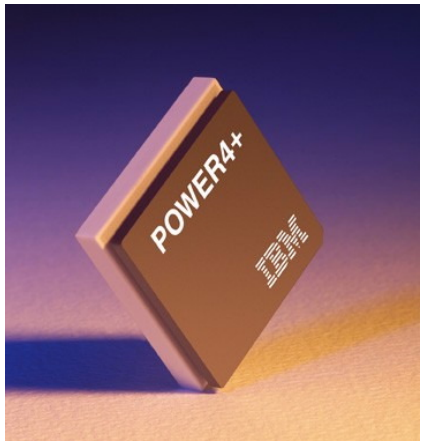


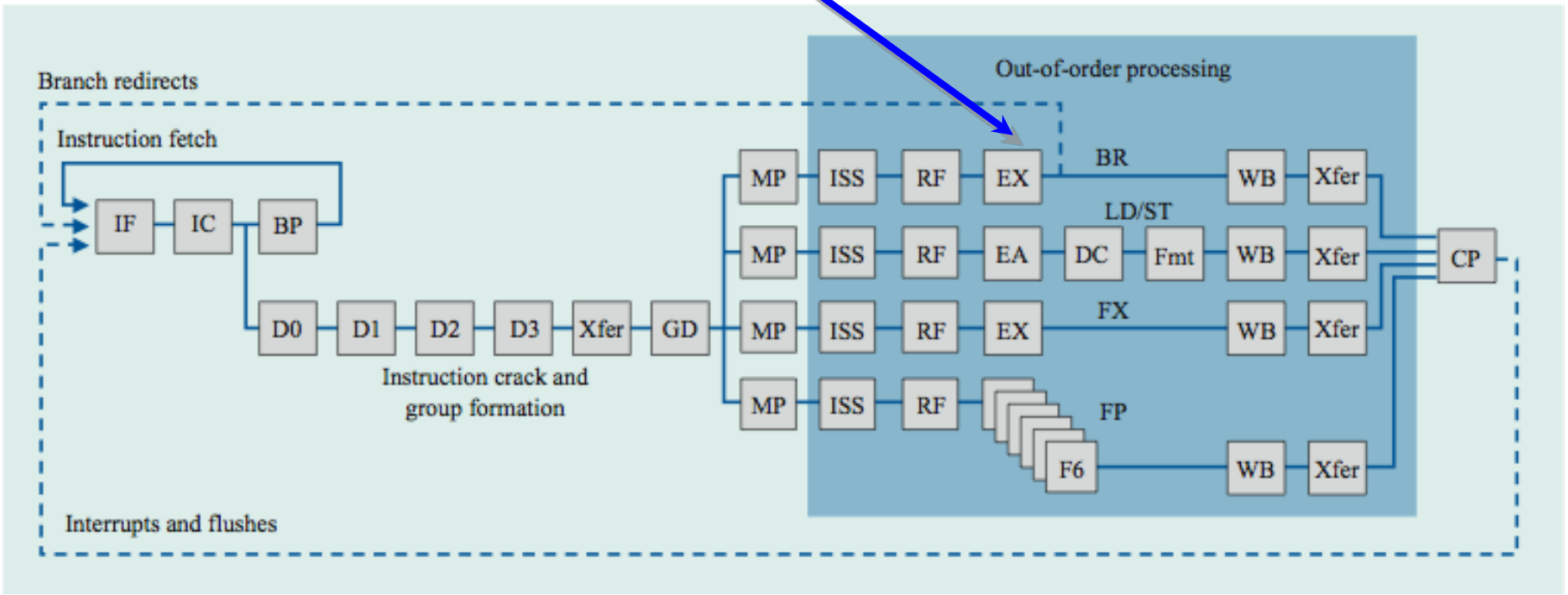
Design Challenges in SMT

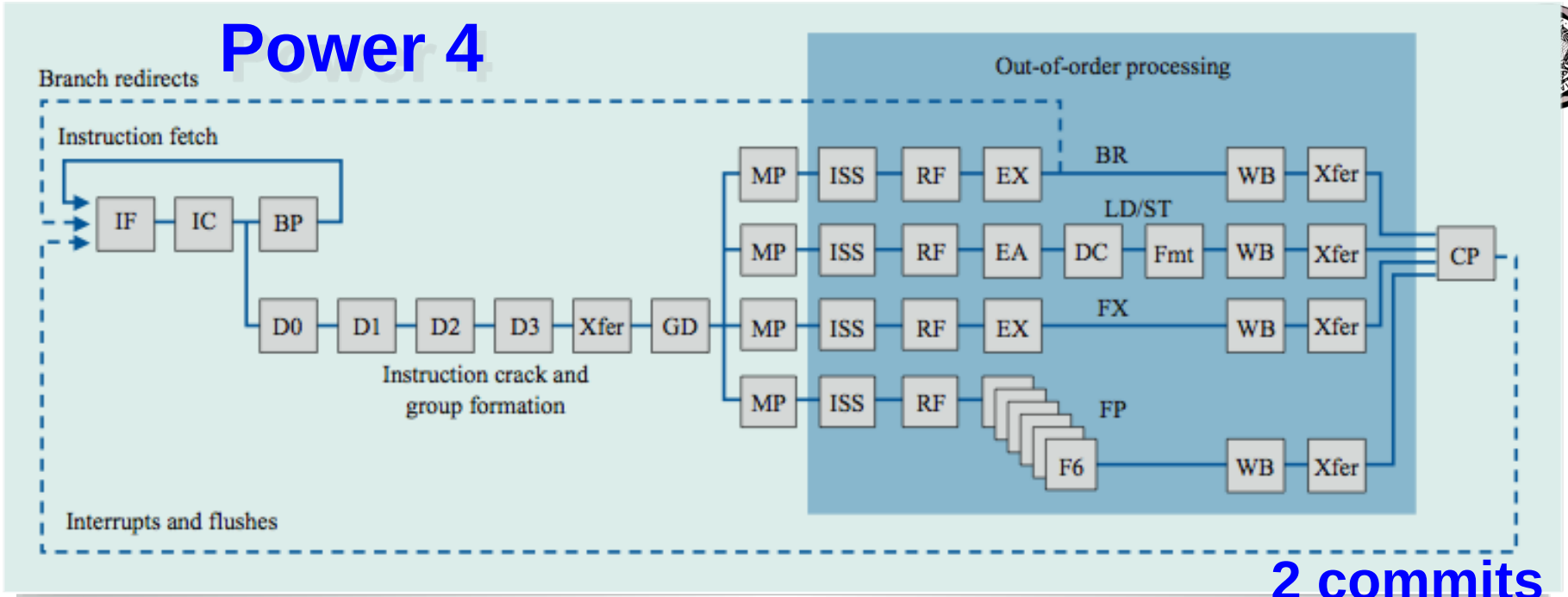
- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Clock cycle time, especially in:
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

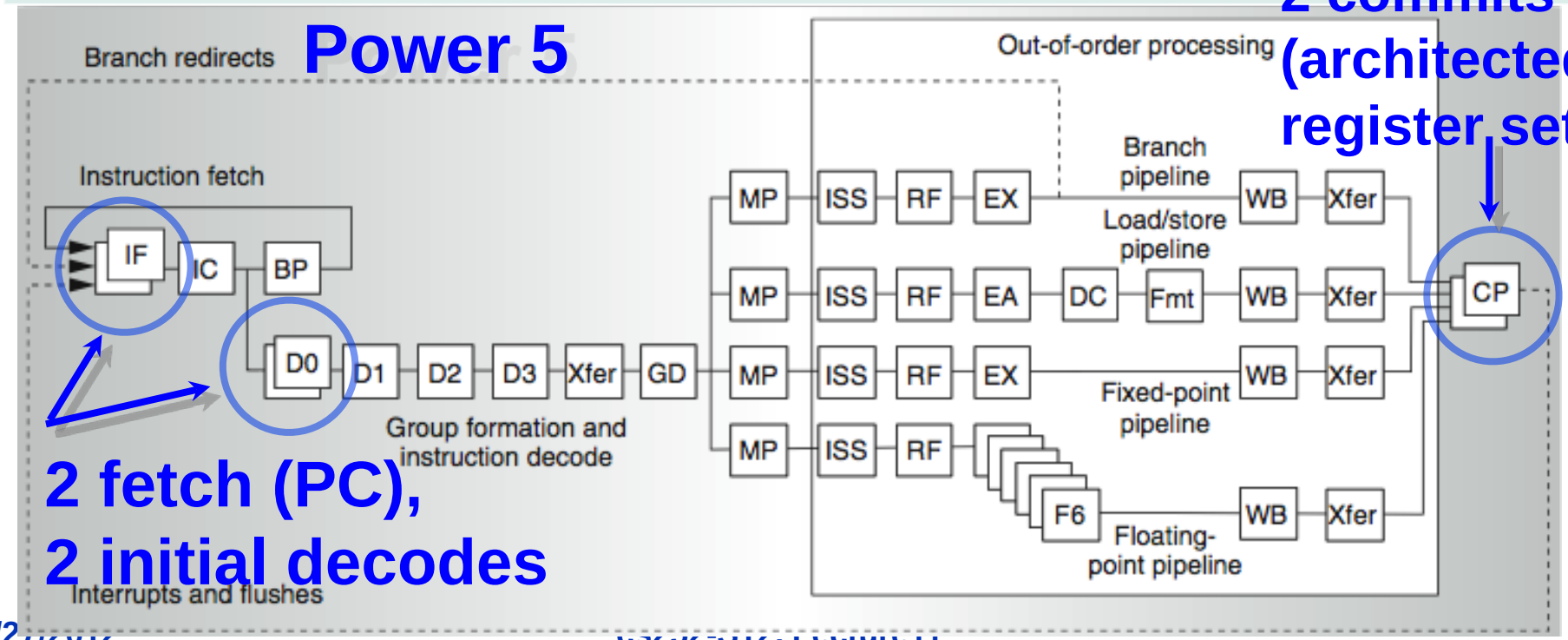
Power 4

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.





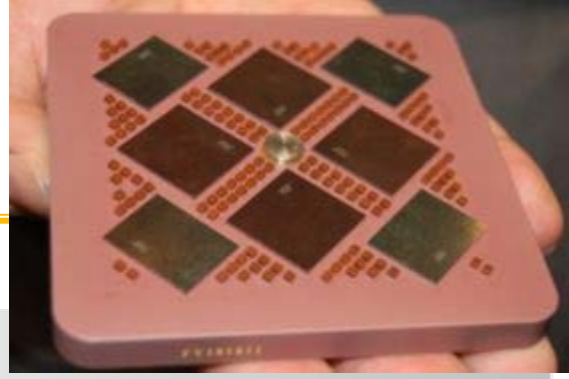


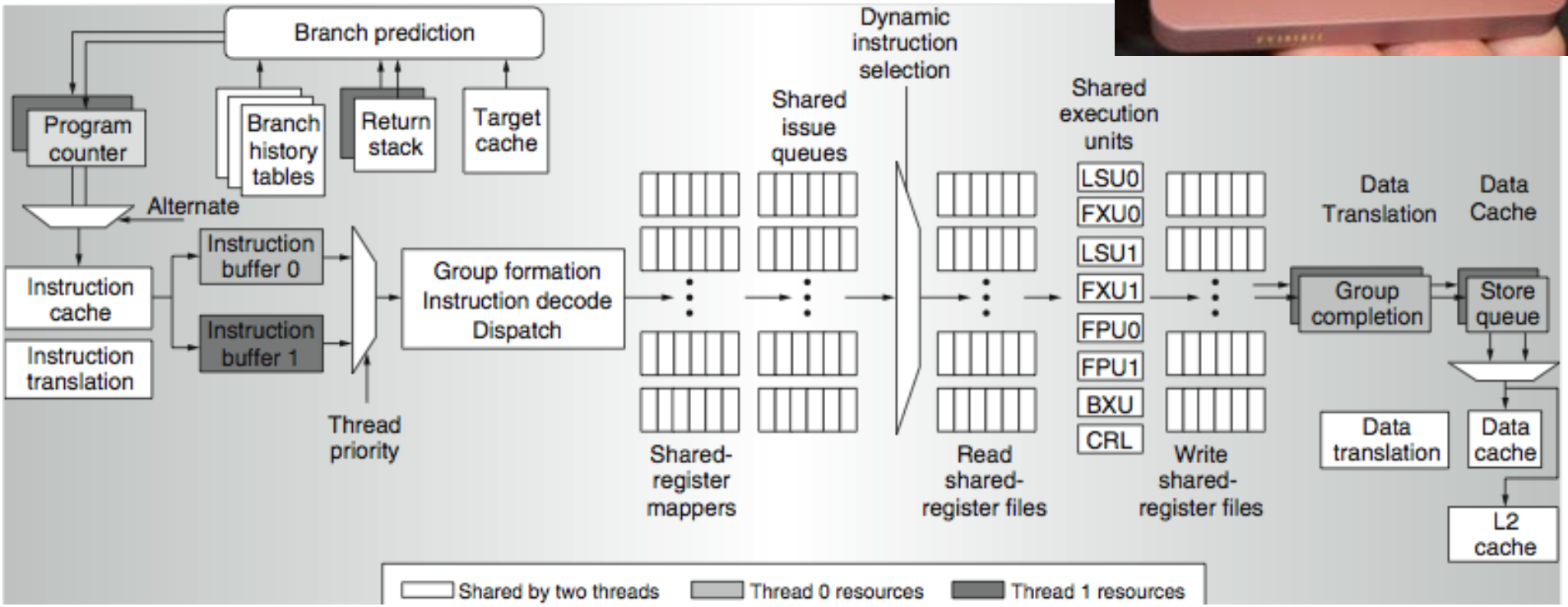


2 commits (architected register sets)

2 fetch (PC), 2 initial decodes

Power 5 data flow ...



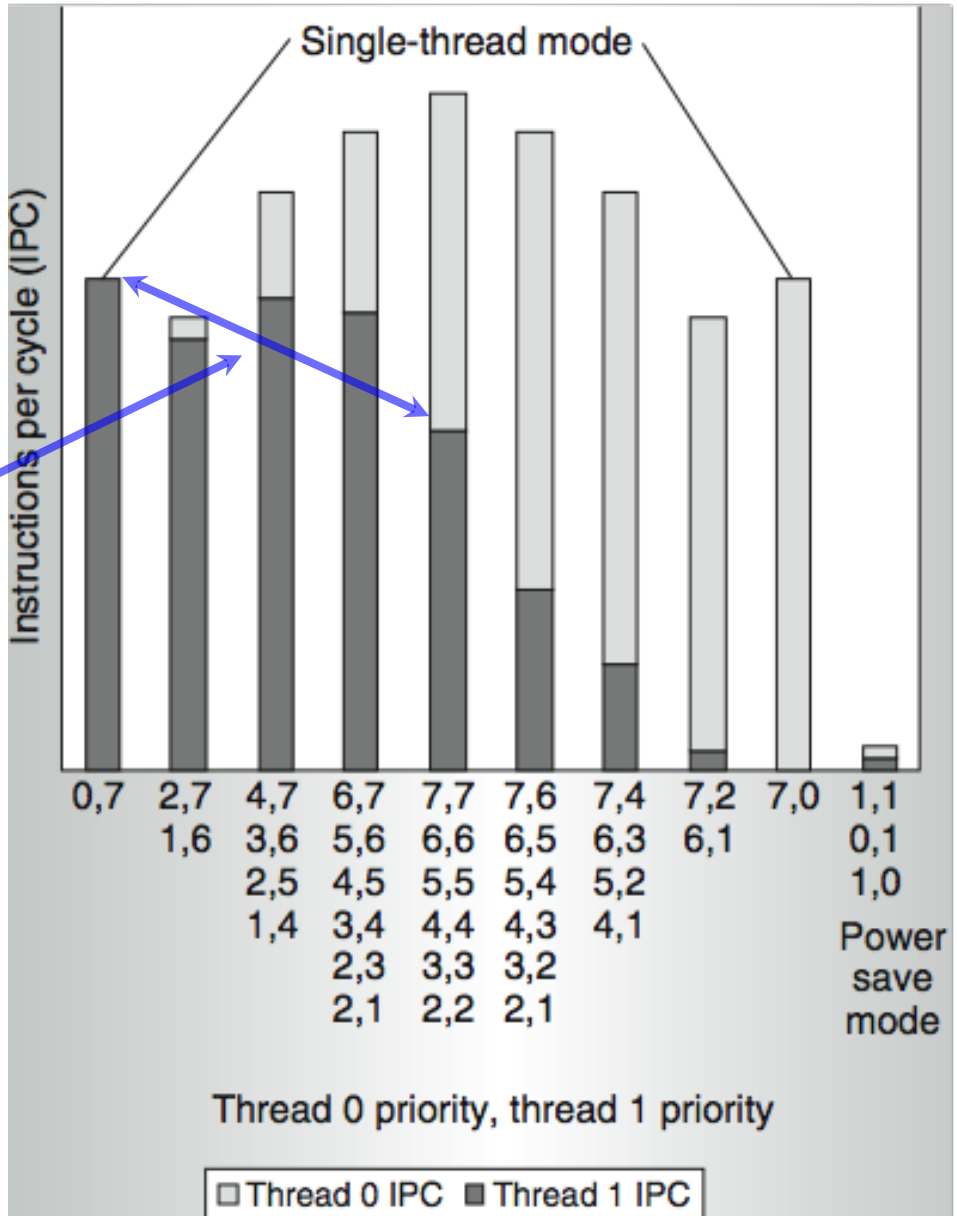


Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they "owned" the machine.



Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support

Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8 processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains

Future of Multithreading

- Will it survive? In what form?
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Vector Supercomputers

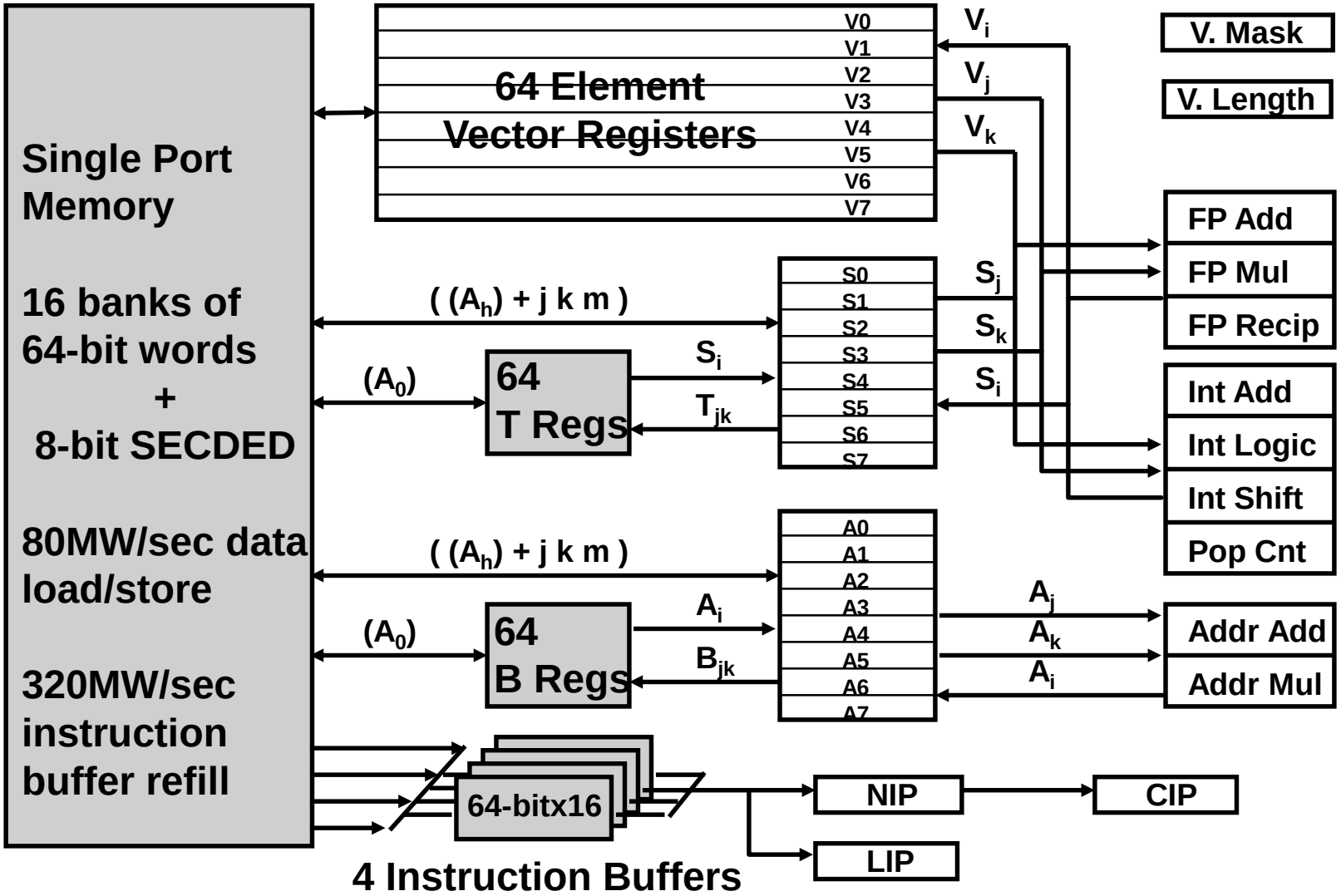
Epitomized by Cray-1, 1976:

Scalar Unit + Vector Extensions

- **Load/Store Architecture**
- **Vector Registers**
- **Vector Instructions**
- **Hardwired Control**
- **Highly Pipelined Functional Units**
- **Interleaved Memory System**
- **No Data Caches**
- **No Virtual Memory**

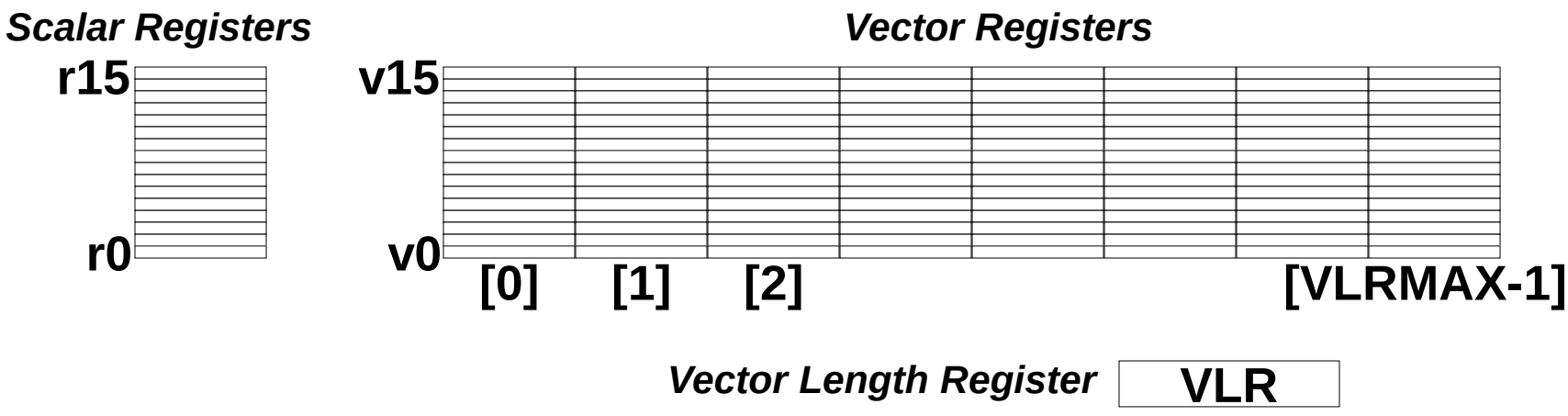
Cray-1 (1976)

Cray-1 (1976)



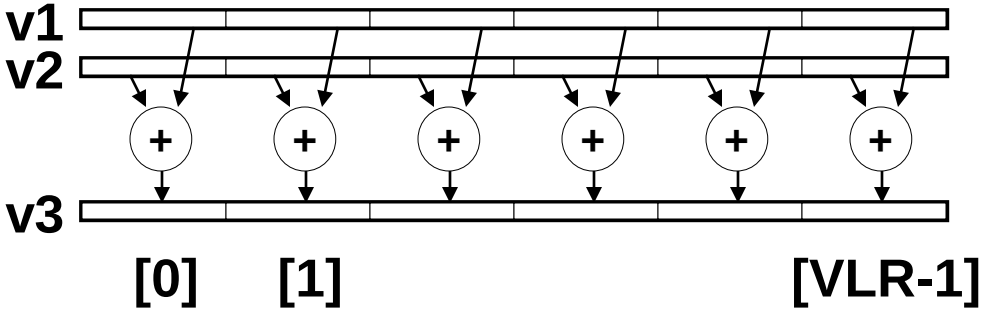
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Programming Model



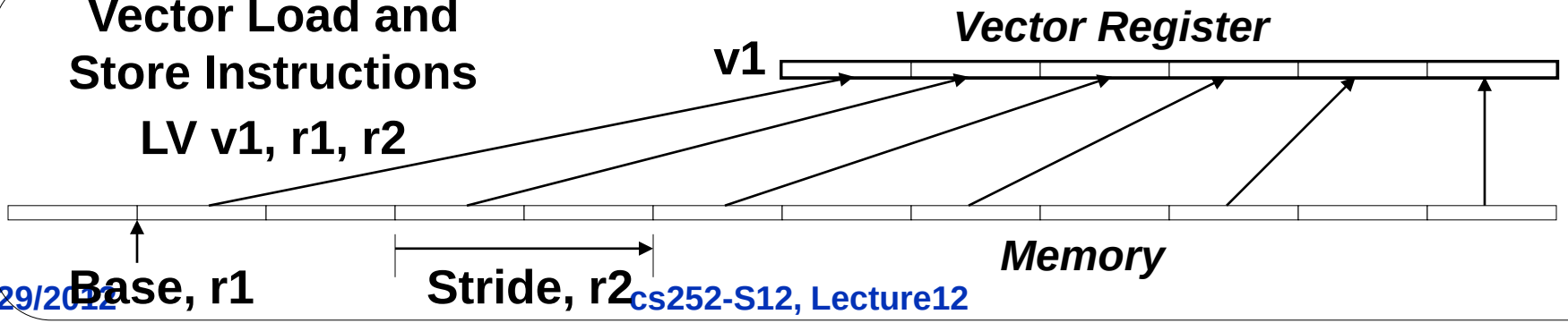
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2



Vector Code Example

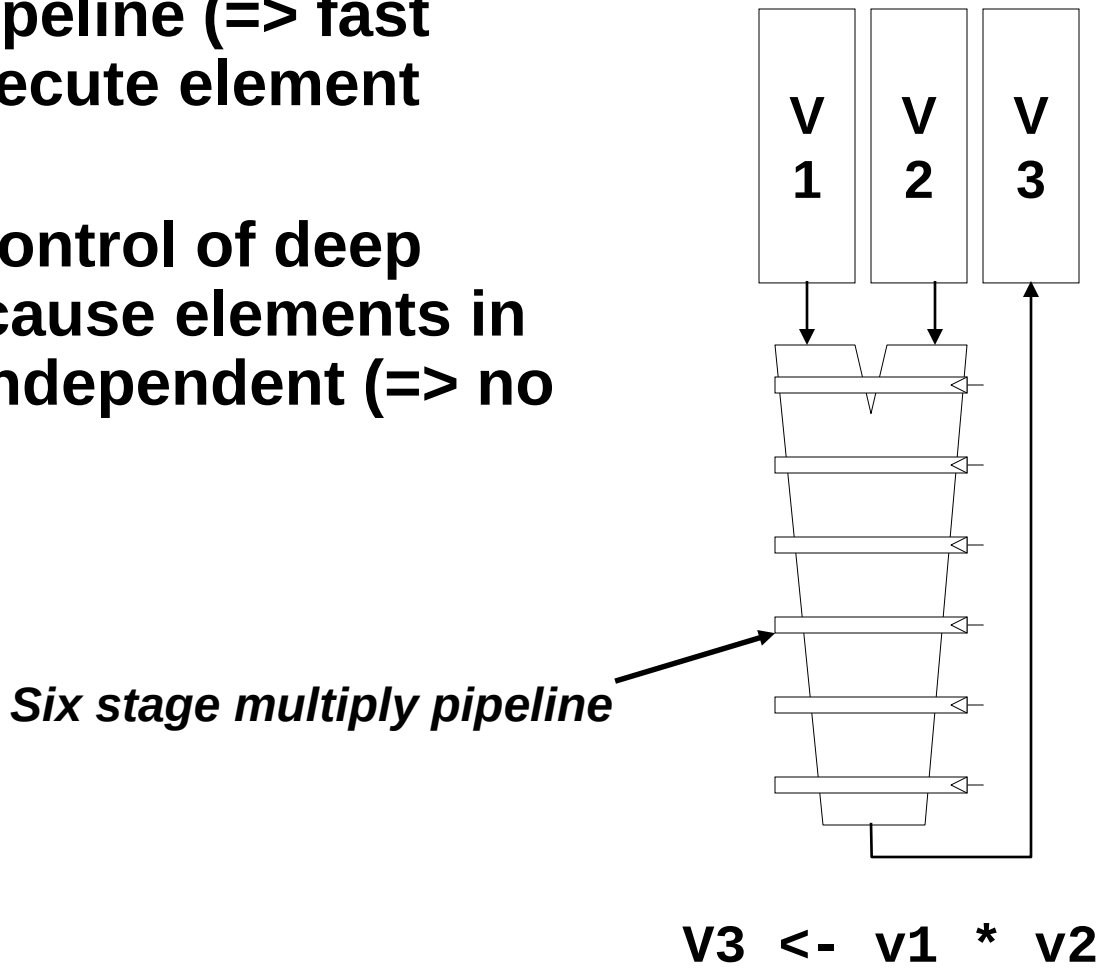
# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++)	LI R4, 64	LI VLR, 64
C[i] = A[i] + B[i];	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

Vector Instruction Set Advantages

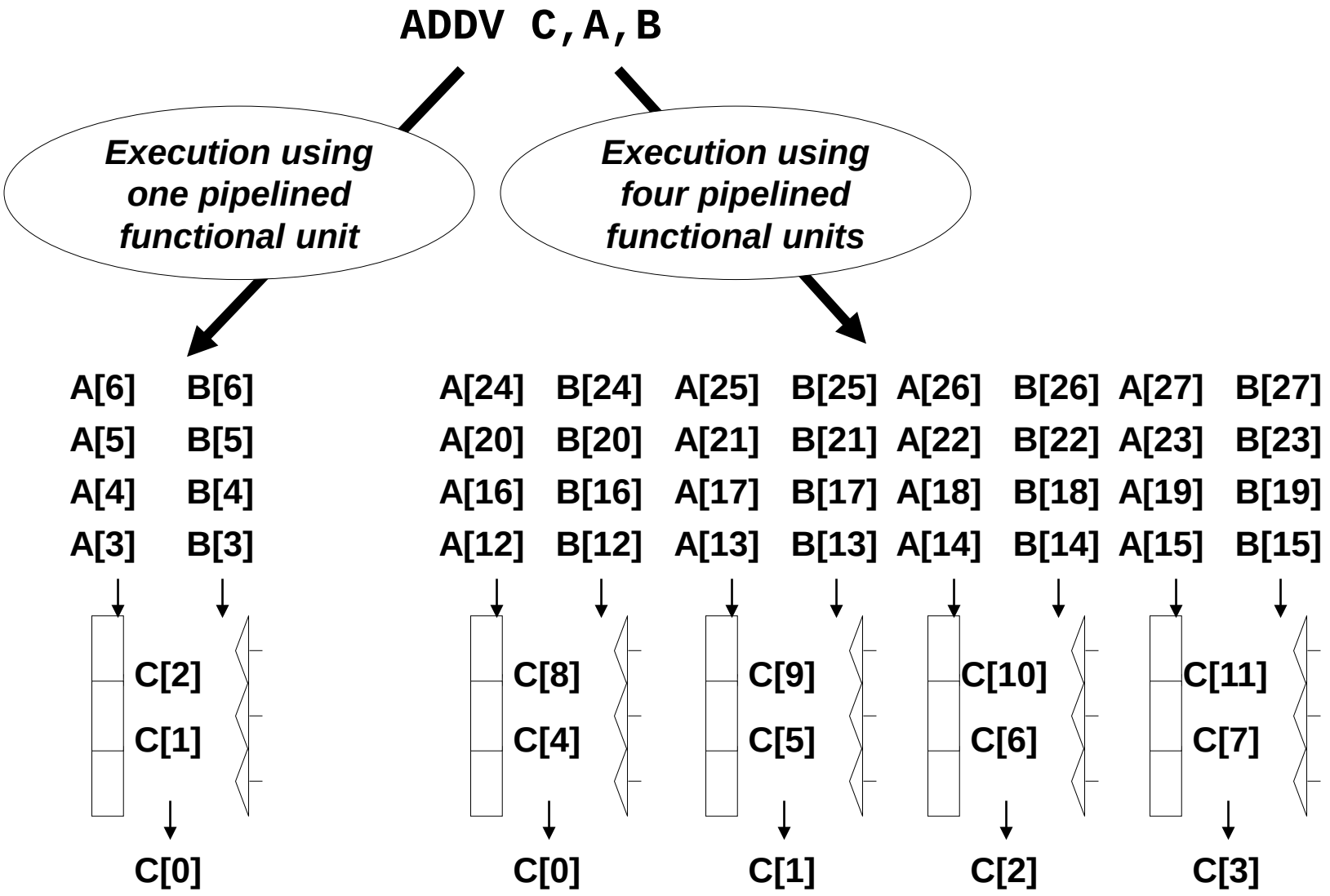
- **Compact**
  - **one short instruction encodes N operations**
- **Expressive, tells hardware that these N operations:**
  - **are independent**
  - **use the same functional unit**
  - **access disjoint registers**
  - **access registers in the same pattern as previous instructions**
  - **access a contiguous block of memory (unit-stride load/store)**
  - **access memory in a known pattern (strided load/store)**
- **Scalable**
  - **can run same object code on more parallel pipelines or *lanes***

Vector Arithmetic Execution

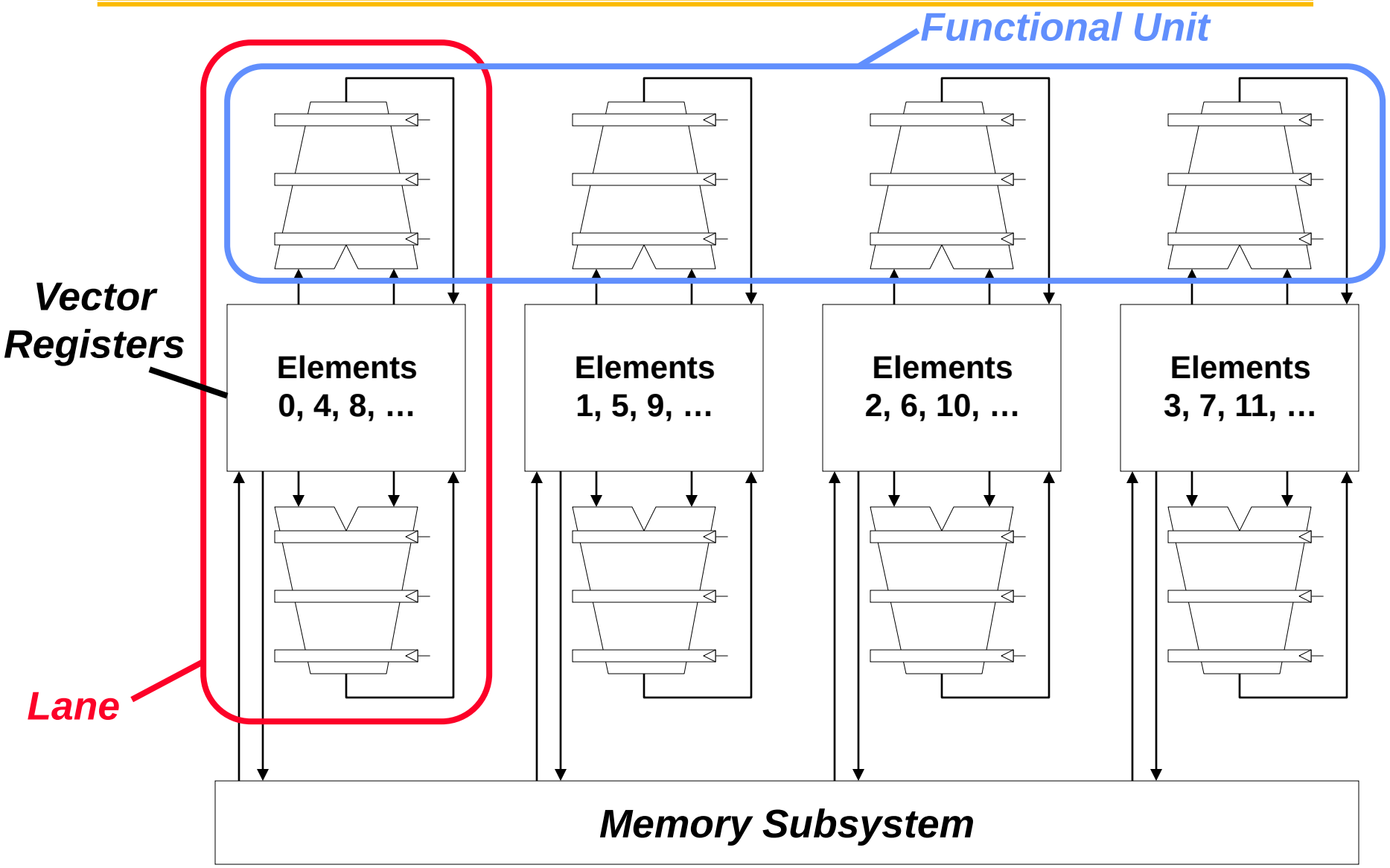
- **Use deep pipeline (=> fast clock) to execute element operations**
- **Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)**



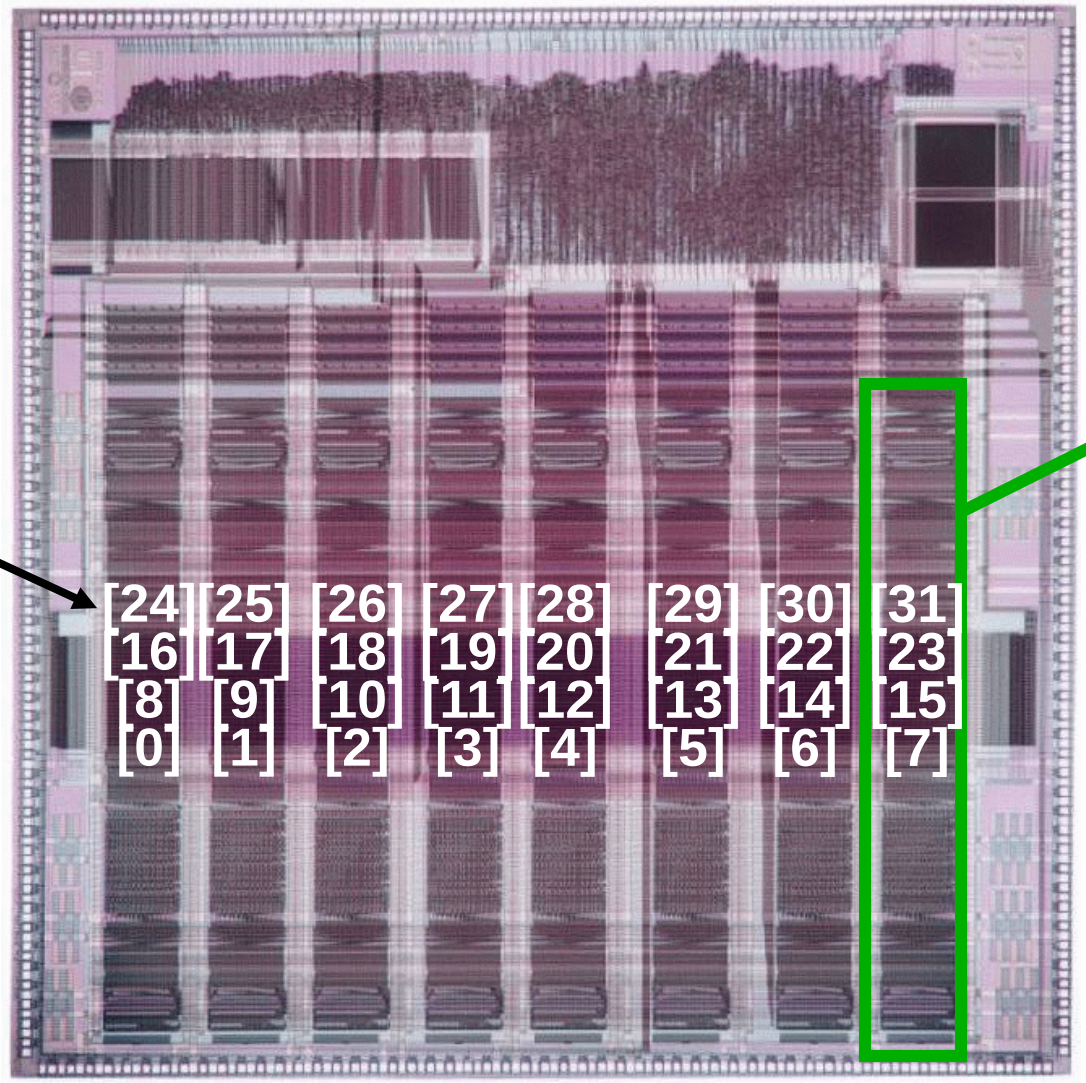
Vector Instruction Execution



Vector Unit Structure



T0 Vector Microprocessor (1995)



Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

Example Source Code

```
for (i=0; i<N; i++)
        {
C[i] = A[i] + B[i];
D[i] = A[i] - B[i];
        }
```

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines

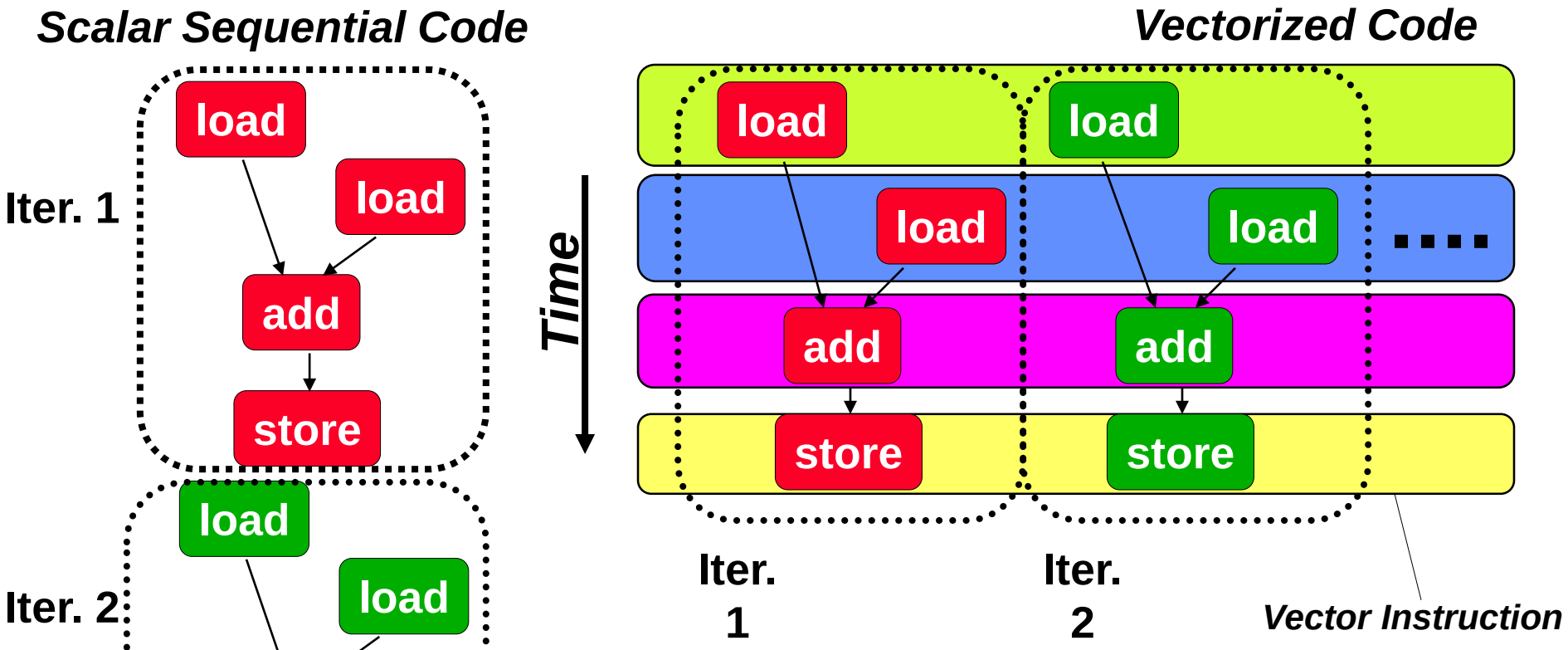
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements

⇒*Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures*

(we ignore vector memory-memory from now on)

Automatic Code Vectorization

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



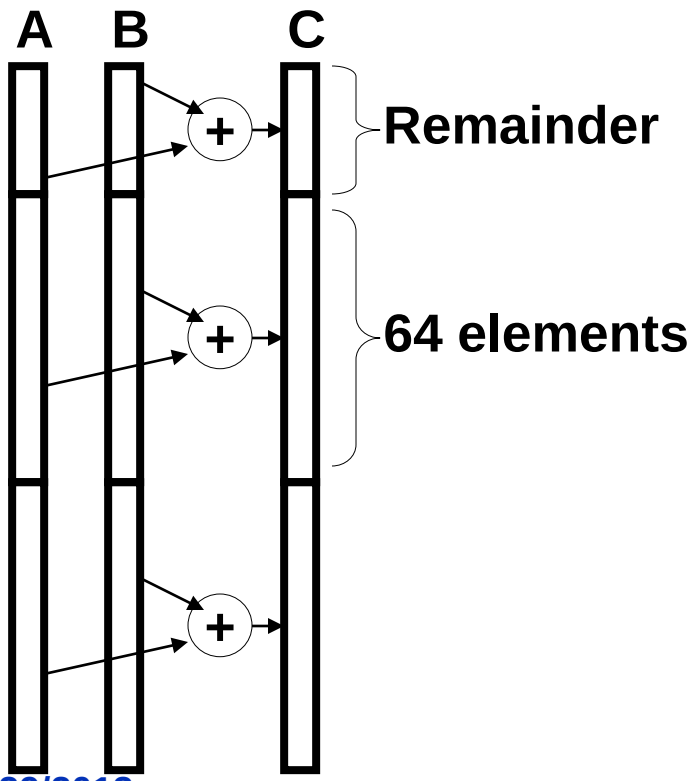
Vectorization is a massive compile-time reordering of operation sequencing ⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit into vector registers, *"Stripmining"*

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```

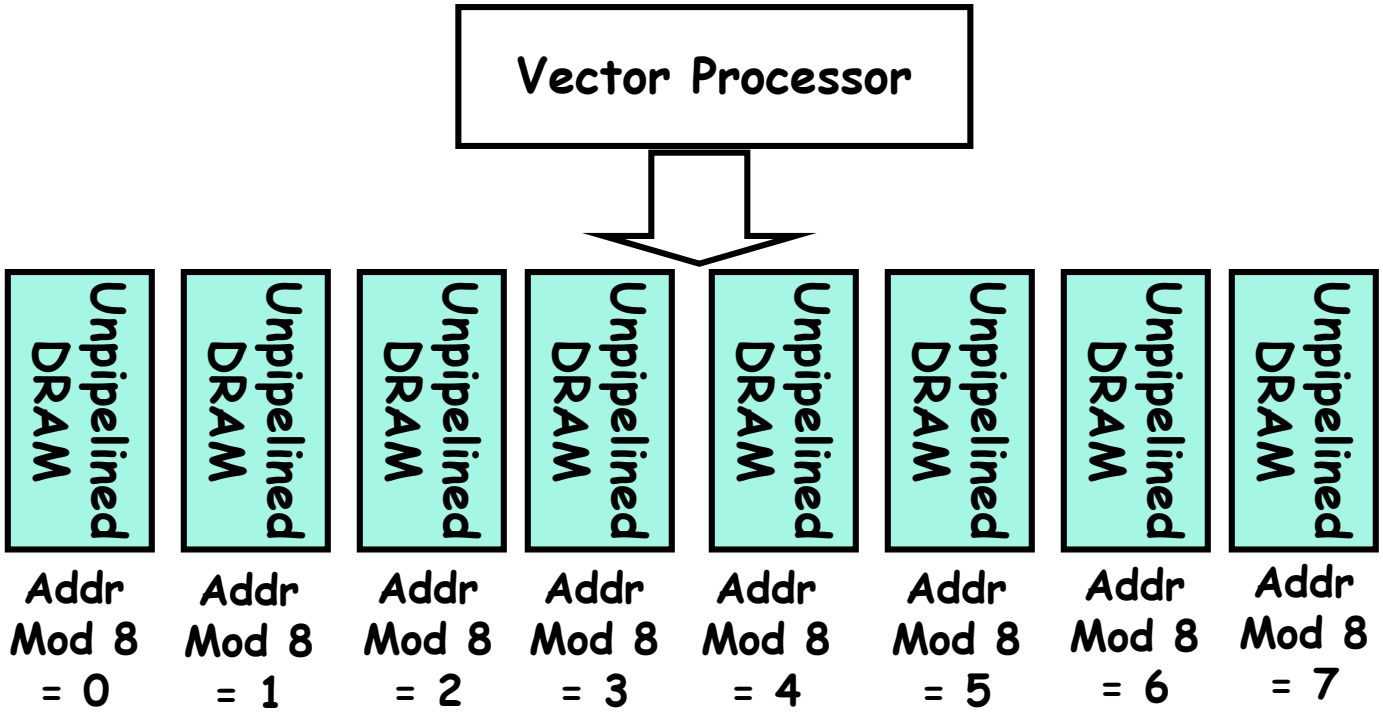



```
        ANDI R1, N, 63     # N mod 64
      MTC1 VLR, R1         # Do remainder
                   loop:
                LV V1, RA
     DSLL R2, R1, 3      # Multiply by 8
     DADDU RA, RA, R2    # Bump pointer
                LV V2, RB
             DADDU RB, RB, R2
            ADDV.D V3, V1, V2
                SV V3, RC
             DADDU RC, RC, R2
    DSUBU N, N, R1 # Subtract elements
                LI R1, 64
    MTC1 VLR, R1    # Reset full length
     BGTZ N, loop    # Any more to do?
```

Memory operations

- **Load/store operations move groups of data between registers and memory**
- **Three types of addressing**
  - **Unit stride**
    - » **Contiguous block of information in memory**
    - » **Fastest: always possible to optimize this**
  - **Non-unit (constant) stride**
    - » **Harder to optimize memory system for all possible strides**
    - » **Prime number of data banks makes it easier to support different strides at full bandwidth**
  - **Indexed (gather-scatter)**
    - » **Vector equivalent of register indirect**
    - » **Good for sparse arrays of data**
    - » **Increases number of programs that vectorize**

Interleaved Memory Layout



- **Great for unit stride:**
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- **What about non-unit stride?**
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4

How to get full bandwidth for Unit Stride?

- Memory system must sustain (# lanes x word) /clock
- No. memory banks > memory latency to avoid stalls
  - *m* banks ⇒ *m* words per memory lantecy *l* clocks
  - if *m* < *l*, then gap in memory pipeline:

clock:	0	…	*l*	*l*+1	*l*+2	…	*l*+*m*- 1	l+m	…	2 *l*
word:	--	…	0	1	2	…	*m*-1	--	…	*m*

  - may have 1024 banks in SRAM
- If desired throughput greater than one word per cycle
  - Either more banks (start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- More banks/weird numbers of banks good to support more strides at full bandwidth
  - can read paper on how to do prime number of banks efficiently

Avoiding Bank Conflicts

- Lots of banks

```
int x[256][512];
    for (j = 0; j < 512; j = j+1)
         for (i = 0; i < 256; i = i+1)
              x[i][j] = 2 * x[i][j];
```

- Even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2^N words per bank

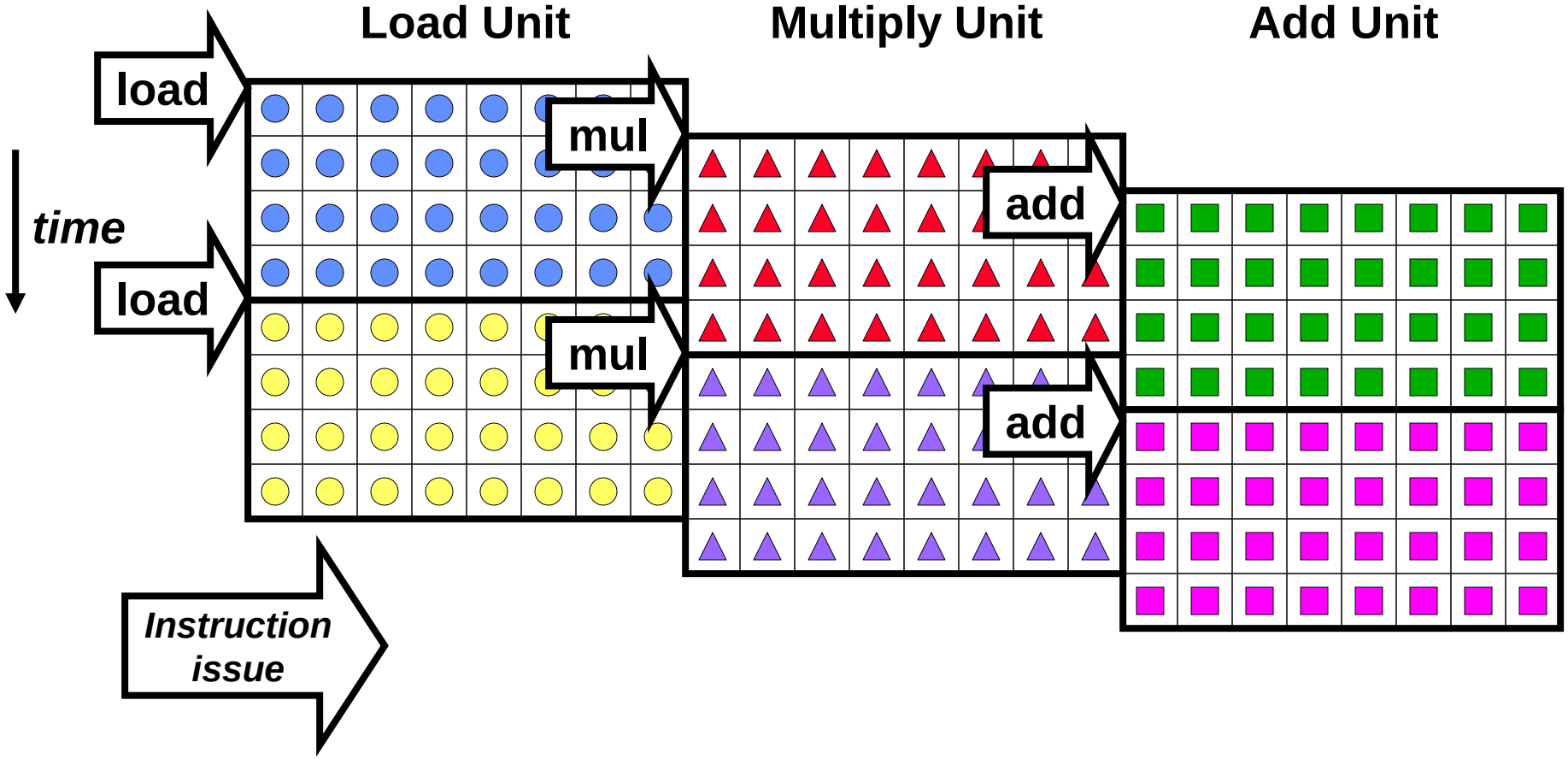
Finding Bank Number and Address within a bank

- **Problem**: Determine the number of banks, N_b and the number of words in each bank, N_w, such that:
  - given address x, it is easy to find the bank where x will be found, B(x), and the address of x within the bank, A(x).
  - for any address x, B(x) and A(x) are unique
  - the number of bank conflicts is minimized
- **Solution**: Use the Chinese remainder theorem to determine B(x) and A(x):
  $B(x) = x \text{ MOD } N_b$
  $A(x) = x \text{ MOD } N_w$ where N_b and N_w are co-prime (no factors)
  - Chinese Remainder Theorem shows that B(x) and A(x) unique.
- Condition allows N_w to be power of two (typical) if N_b is prime of form 2^m-1.
- Simple (fast) circuit to compute (x mod N_b) when $N_b = 2^m-1$:
  - Since $2^k = 2^{k-m}(2^m-1) + 2^{k-m} \Rightarrow 2^k \text{ MOD } N_b = 2^{k-m} \text{ MOD } N_b = \ldots = 2^j$ with $j < m$
  - And, remember that: (A+B) MOD C = [(A MOD C)+(B MOD C)] MOD C
  - for every power of 2, compute single bit MOD (in advance)
  - B(x) = sum of these values MOD N_b
    (low complexity circuit, adder with ~ m bits)

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

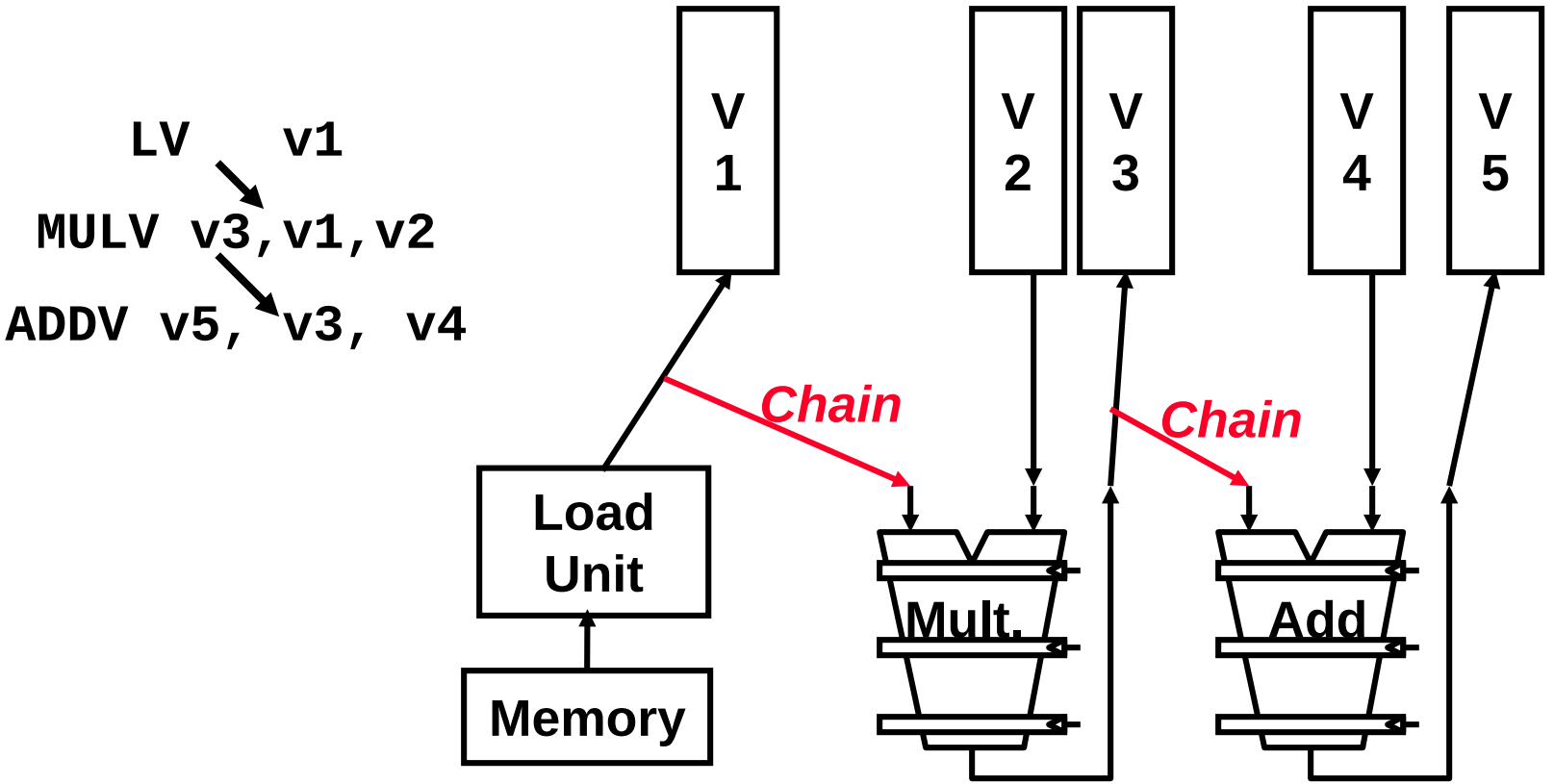
- **example machine has 32 elements per vector register and 8 lanes**



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

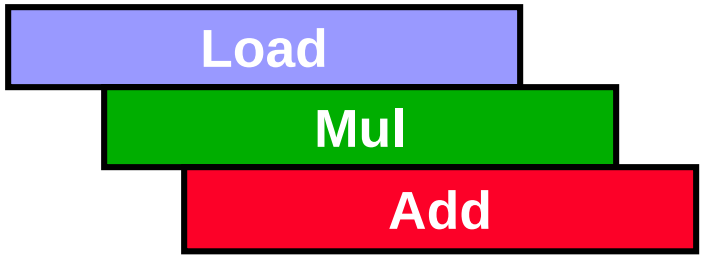
- **Vector version of register bypassing**
  - **introduced with Cray-1**



Vector Chaining Advantage

Without chaining, must wait for last element of result to be written before starting dependent instruction

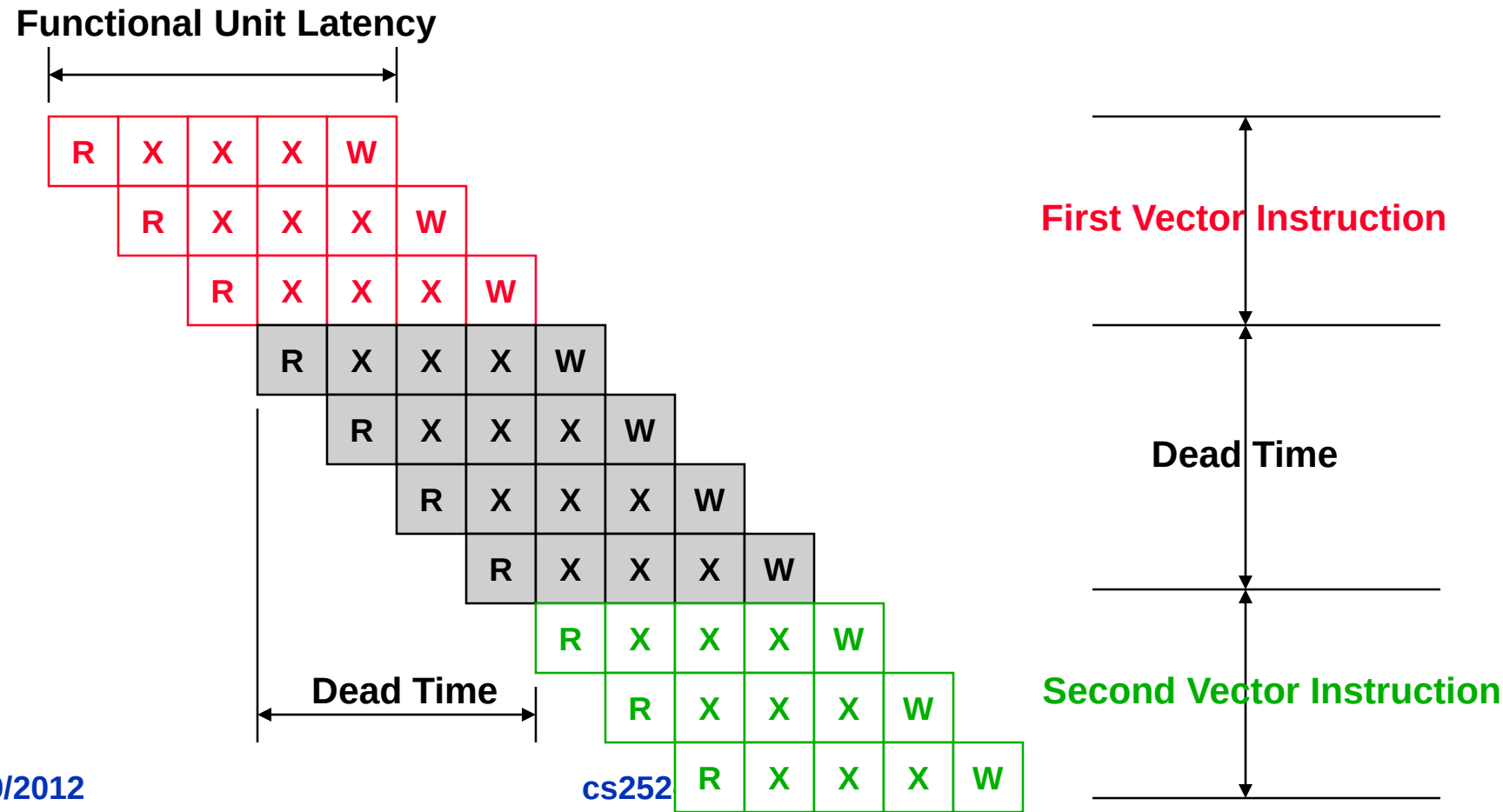
With chaining, can start dependent instruction as soon as first result appears



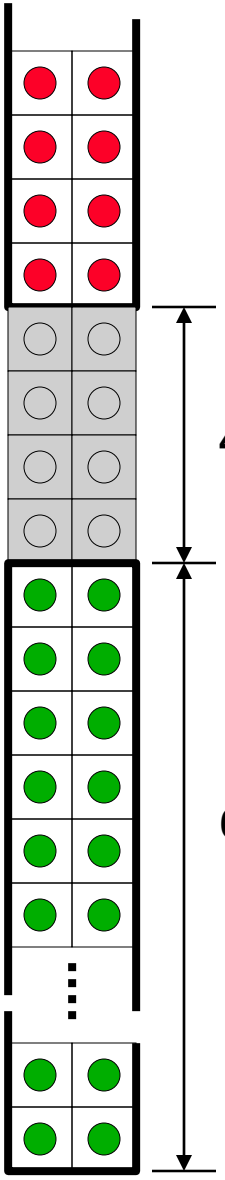
Vector Startup

Two components of vector startup penalty

- functional unit latency (time through pipeline)
- dead time or recovery time (time before another vector instruction can start down pipeline)



Dead Time and Short Vectors



4 cycles dead time

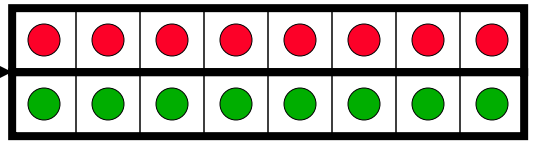
64 cycles active

Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94% with 128 element vectors

No dead time



T0, Eight lanes

No dead time

100% efficiency with 8 element vectors

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (*Gather*)

```
LV vD, rD        # Load indices in D vector
LVI vC, rC, vD   # Load indirect from rC base
LV vB, rB        # Load B vector
ADDV.D vA, vB, vC # Do add
SV vA, rA        # Store result
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

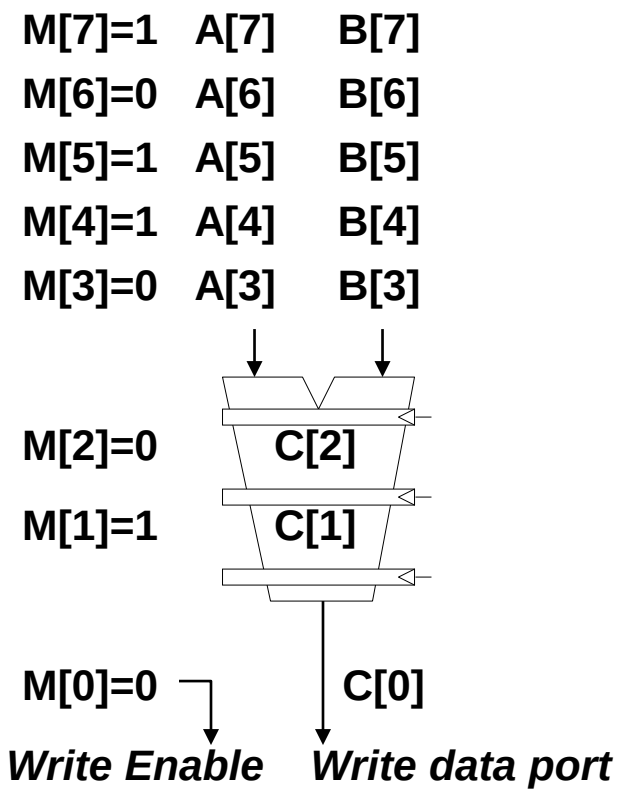
Code example:

```
      CVM              # Turn on all elements
      LV vA, rA        # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
   LV vA, rB       # Load B vector into A under mask
   SV vA, rA       # Store A back to memory under mask
```

Masked Vector Instructions

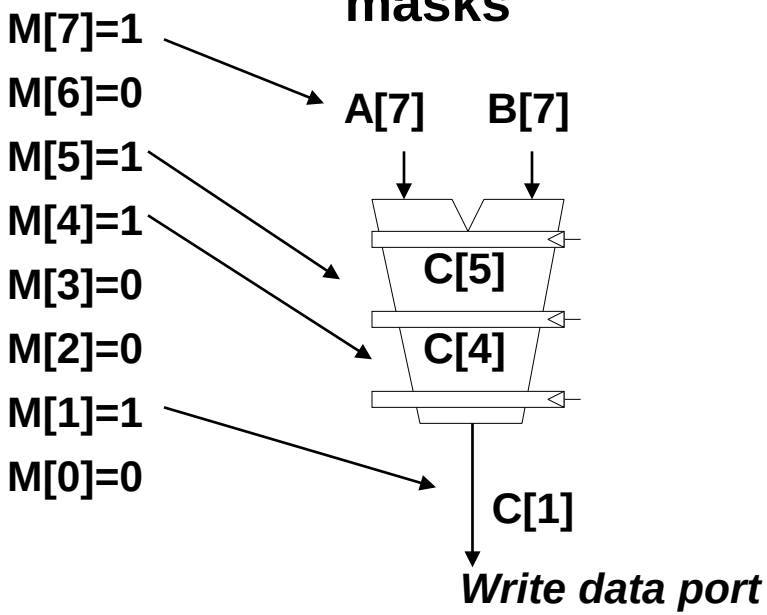
Simple Implementation

– execute all N operations, turn off result writeback according to mask



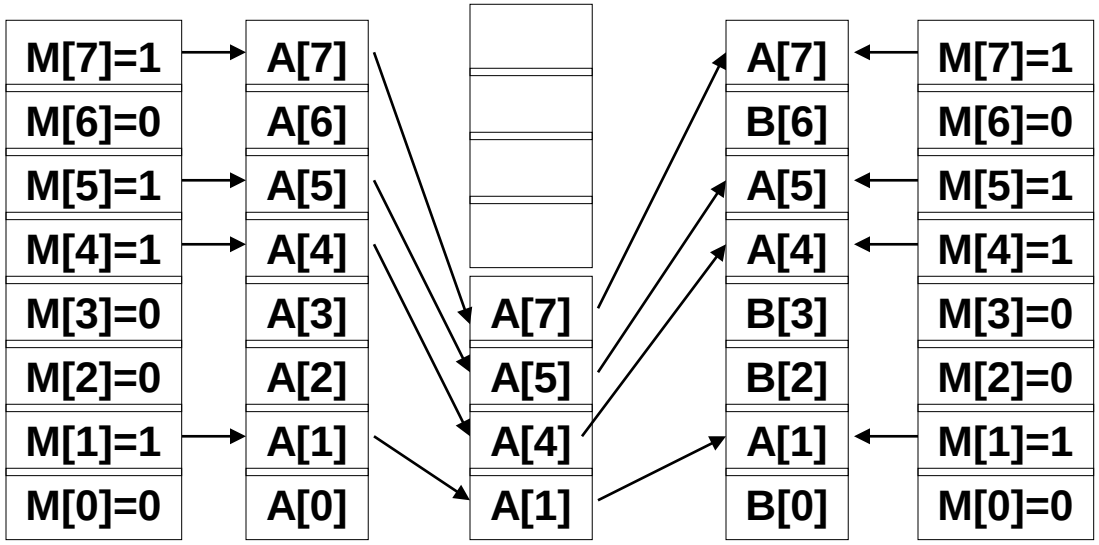
Density-Time Implementation

– scan mask vector and only execute elements with non-zero masks



Compress/Expand Operations

- **Compress packs non-masked elements from one vector register contiguously at start of destination vector register**
  - **population count of mask vector gives packed vector length**
- **Expand performs inverse operation**



Compress ***Expand***

Used for density-time conditionals and also for general selection operations

Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0                     # Vector of VL partial sums
for(i=0; i<N; i+=VL)                # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;                        # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```

Novel Matrix Multiply Solution

- **Consider the following:**

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
    for (j=1; j<n; j++) {
        sum = 0;
        for (t=1; t<k; t++)
          sum += a[i][t] * b[t][j];
        c[i][j] = sum;
    }
}
```

- **Do you need to do a bunch of reductions? NO!**
  - Calculate multiple independent sums within one vector register
  - You can vectorize the j loop to perform 32 dot-products at the same time (Assume Max Vector Length is 32)
- **Show it in C source code, but can imagine the assembly vector instructions from it**

Optimized Vector Example

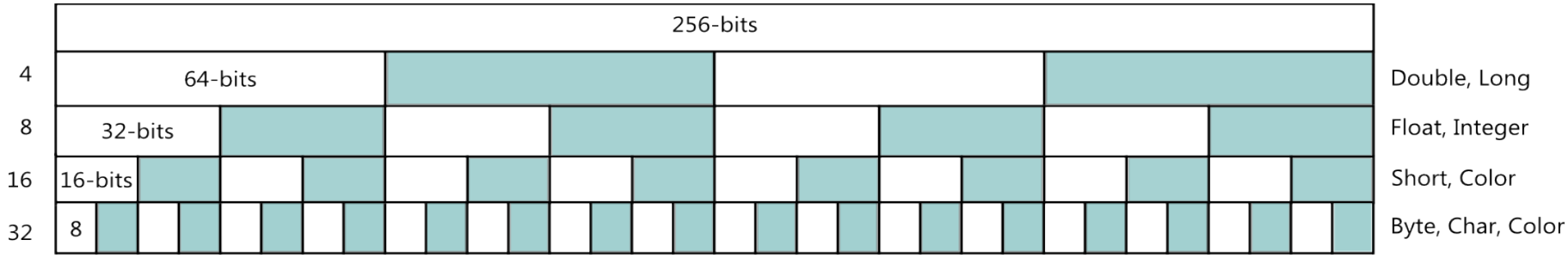
```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
  for (j=1; j<n; j+=32) {/* Step j 32 at a time. */
    sum[0:31] = 0; /* Init vector reg to zeros. */
    for (t=1; t<k; t++) {
    a_scalar = a[i][t]; /* Get scalar */
       b_vector[0:31] = b[t][j:j+31]; /* Get vector */

       /* Do a vector-scalar multiply. */
       prod[0:31] = b_vector[0:31]*a_scalar;

       /* Vector-vector add into results. */
       sum[0:31] += prod[0:31];
    }
    /* Unit-stride store of vector of results. */
    c[i][j:j+31] = sum[0:31];
  }
}
```

Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors

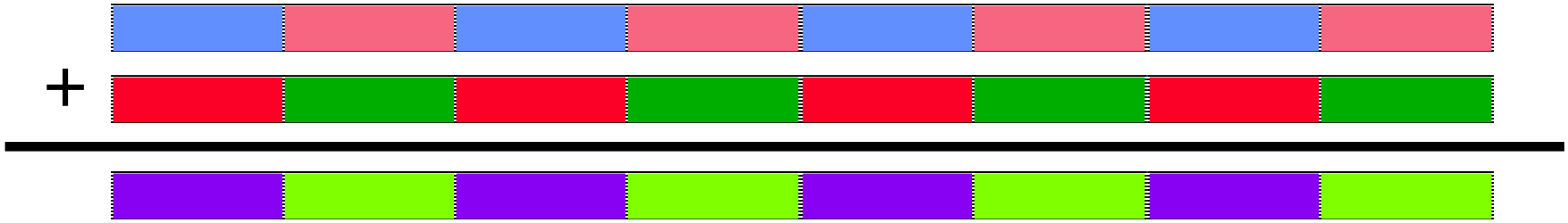
SIMD in Hardware

- Streaming SIMD requires some basic components
  - Wide Registers
    - Rather than 32bits, have 64, 128, or 256 bit wide registers.
  - Additional control lines
  - Additional ALU's to handle the simultaneous operation on up to operand sizes of 16-bytes



“Vector” for Multimedia?

- Intel MMX: 57 additional 80x86 instructions (1st since 386)
  - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
- 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  - reuse 8 FP registers (FP and MMX cannot mix)
- short vector: load, add, store 8 8-bit operands



- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler

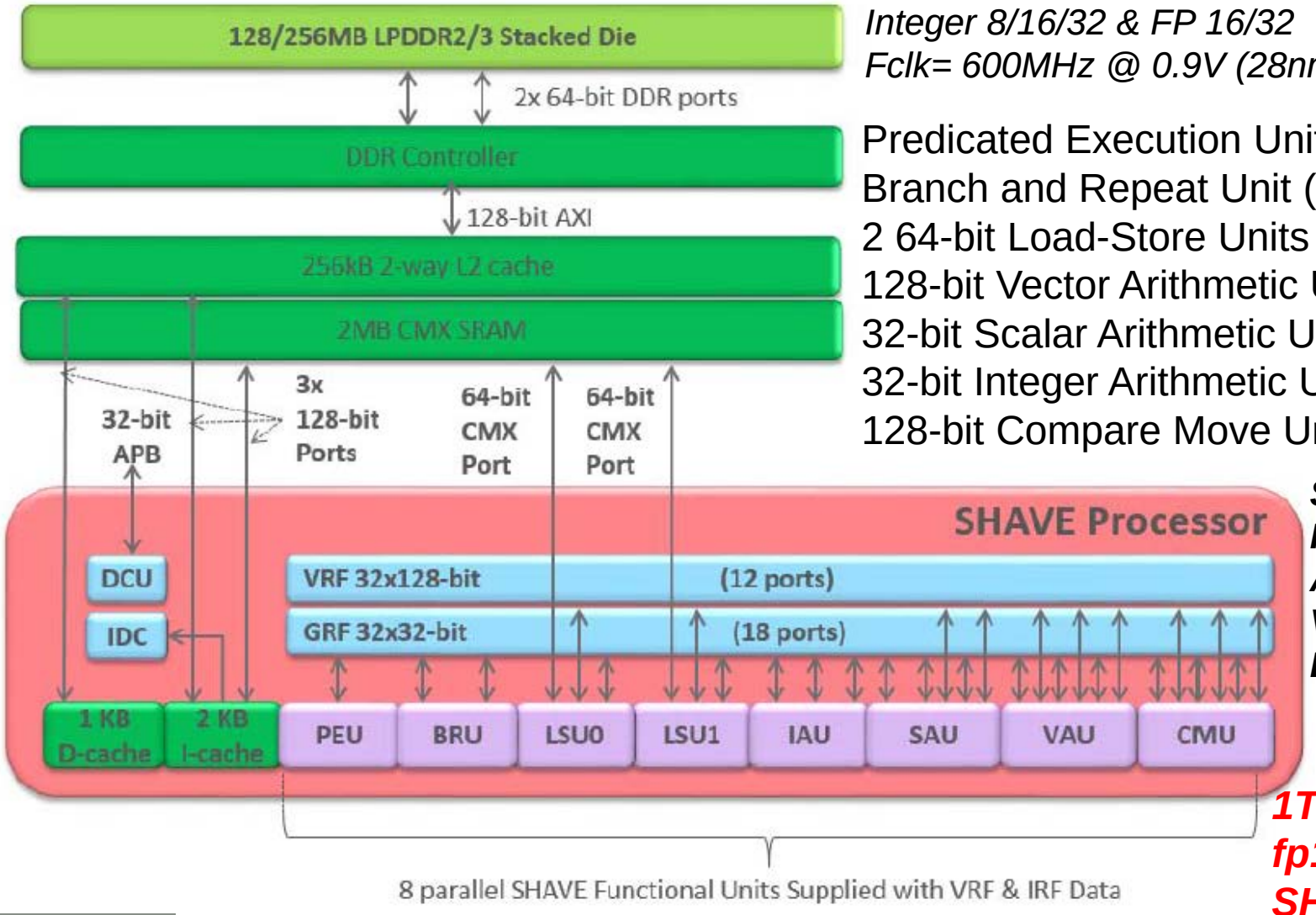
MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll,srl, sra), And, And Not, Or, Xor in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<–> 16b, 16b <–> 8b
  - Pack saturates (set to max) if number is too large

Multithreading and Vector Summary

- Explicitly parallel (Data level parallelism or Thread level parallelism) is next step to performance
- Coarse grain vs. Fine grained multihreading
  - Only on big stall vs. every clock cycle
- Simultaneous Multithreading if fine grained multithreading based on OOO superscalar microarchitecture
  - Instead of replicating registers, reuse rename registers
- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
  - Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
  - With virtual address translation and caching

Movidius' SHAVE VP



Integer 8/16/32 & FP 16/32
Fclk= 600MHz @ 0.9V (28nm)

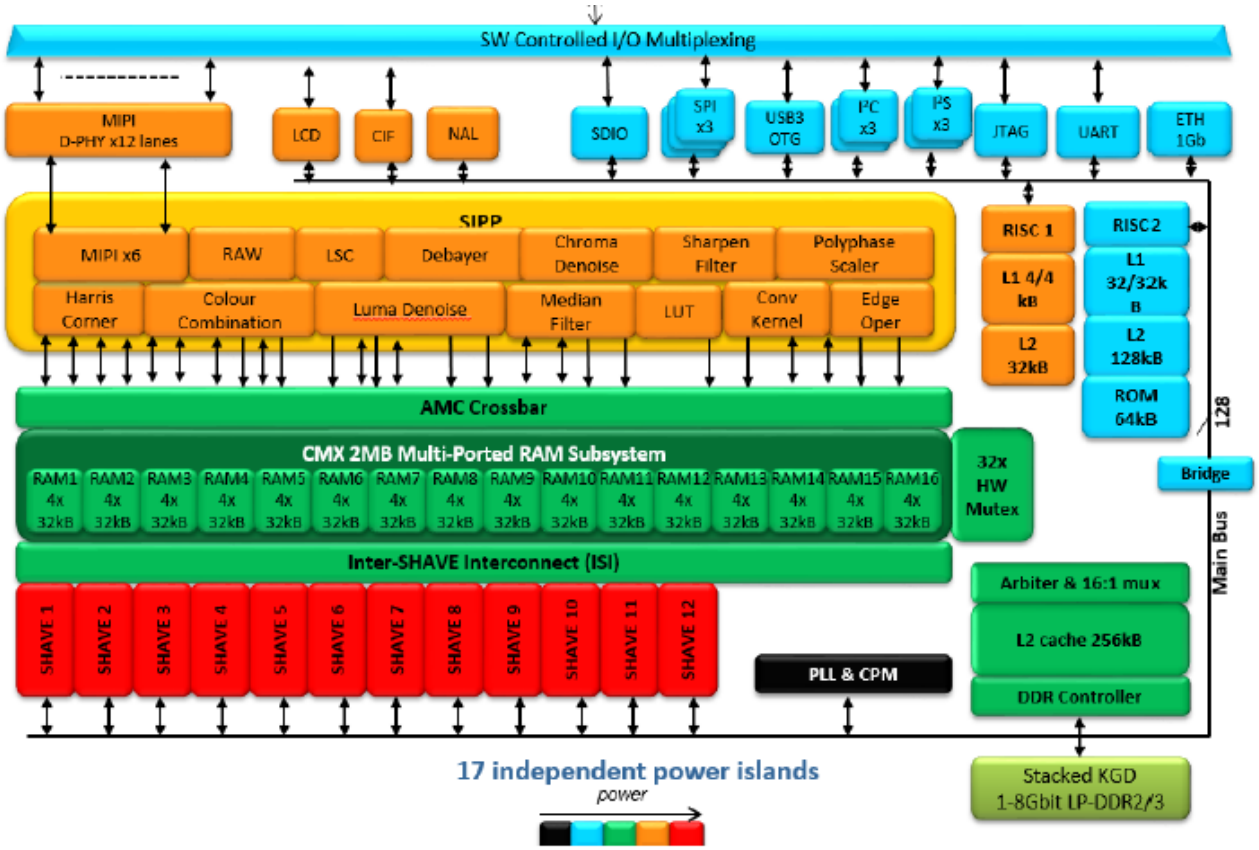
Predicated Execution Unit (PEU)
Branch and Repeat Unit (BRU),
2 64-bit Load-Store Units (LSU0/1),
128-bit Vector Arithmetic Unit (VAU),
32-bit Scalar Arithmetic Unit (SAU),
32-bit Integer Arithmetic Unit (IAU)
128-bit Compare Move Unit (CMU)

Streaming Hybrid Archit. Vector Engine

1TFLOP fp16 with 12 SHAVES

Movidius Myriad

https://www.google.com/atap/projecttango/index.html



20+ programmable HW-accelerators:

–Poly-phase resizer
–Lens shading correction
–Harris Corner detector
–HoG/Edge operator
–Convolution filter
–Sharpening filter
–g correction
–tone-mapping
–Luma/Chroma Denoise

Each accelerator has

–Memory ports
–Decoupling buffers

1op/pixel/cycle

Joint announcement with Google 2014!

Looking forward

- SIMD and vector operations match multimedia applications and are easy to program