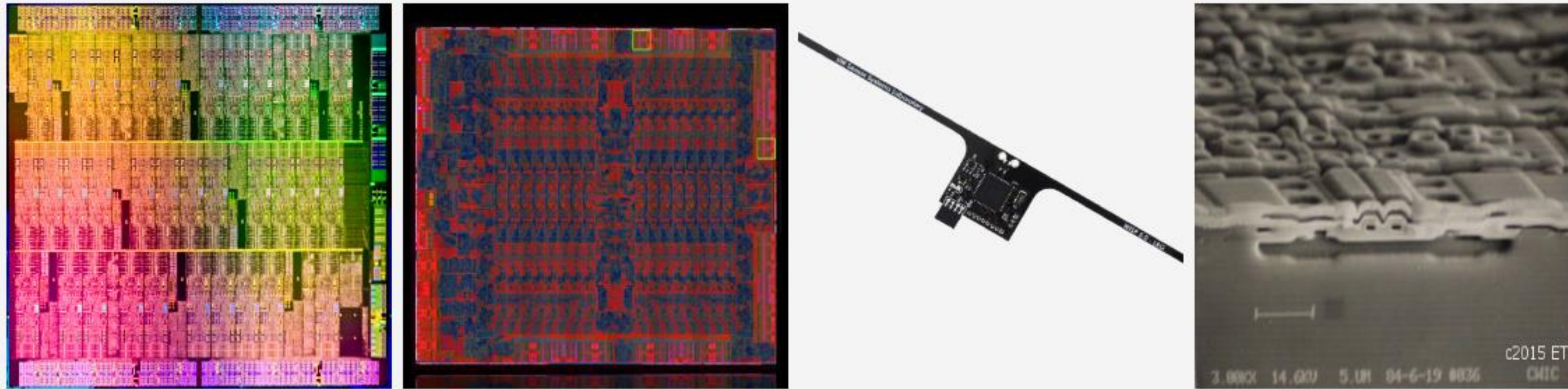
18-344: Computer Systems and the Hardware-Software Interface Fall 2023



Course Description

Lecture 12: Adv. Architecture: Superscalar and Out of Order, cont.

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

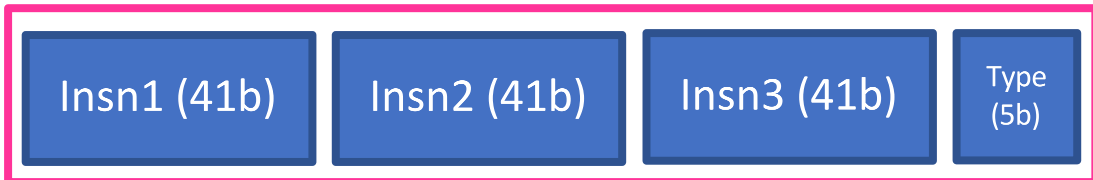
Credit: Brandon Lucia

Today: More Advanced Architecture Concepts

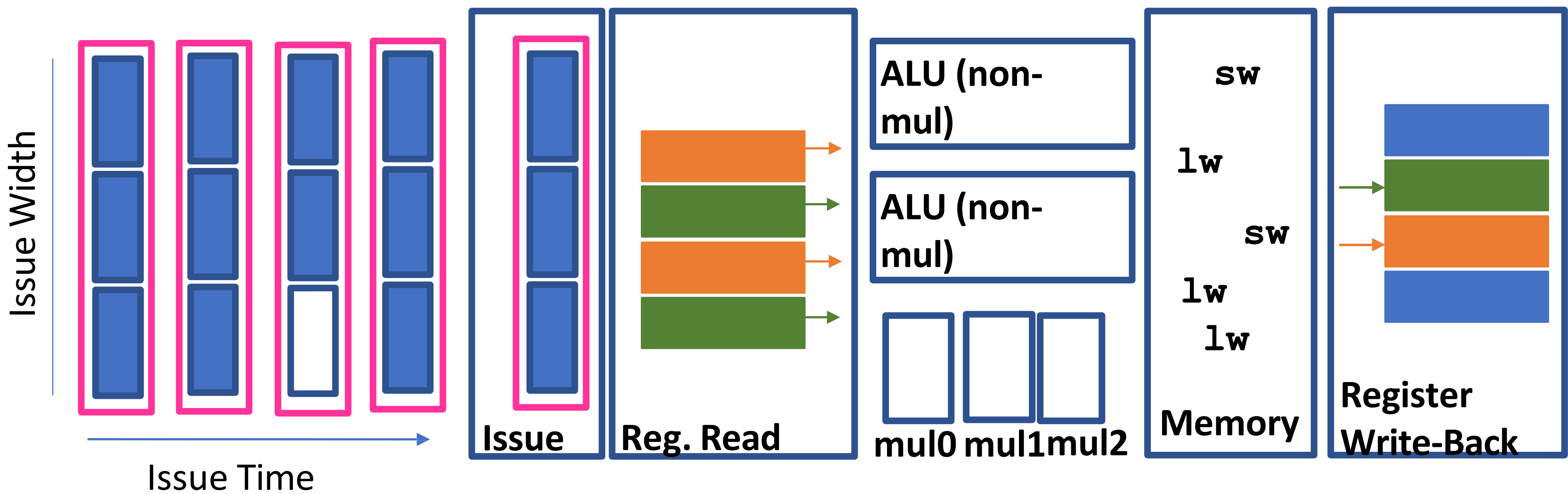
- (more) VLIW
- Vector machines & SIMD
- Dataflow as a hardware/software boundary design problem
- Systolic Array Architectures (if time)

Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

Change the *ISA*! In VLIW, the ISA exposes issue width architecturally. Each fetch / issue is on a *bundle* of instructions that are independent



EPIC/IA-64 bundles up to 3 instructions with a *type* that says whether & how they're dependent or parallelizable

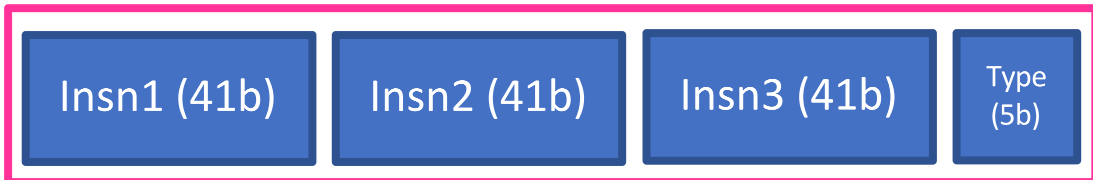


Type:
Mem, Float, Int, Long Imm. Branch
e.g.,
MMI, IIF, MMI
MM/I, M/MI

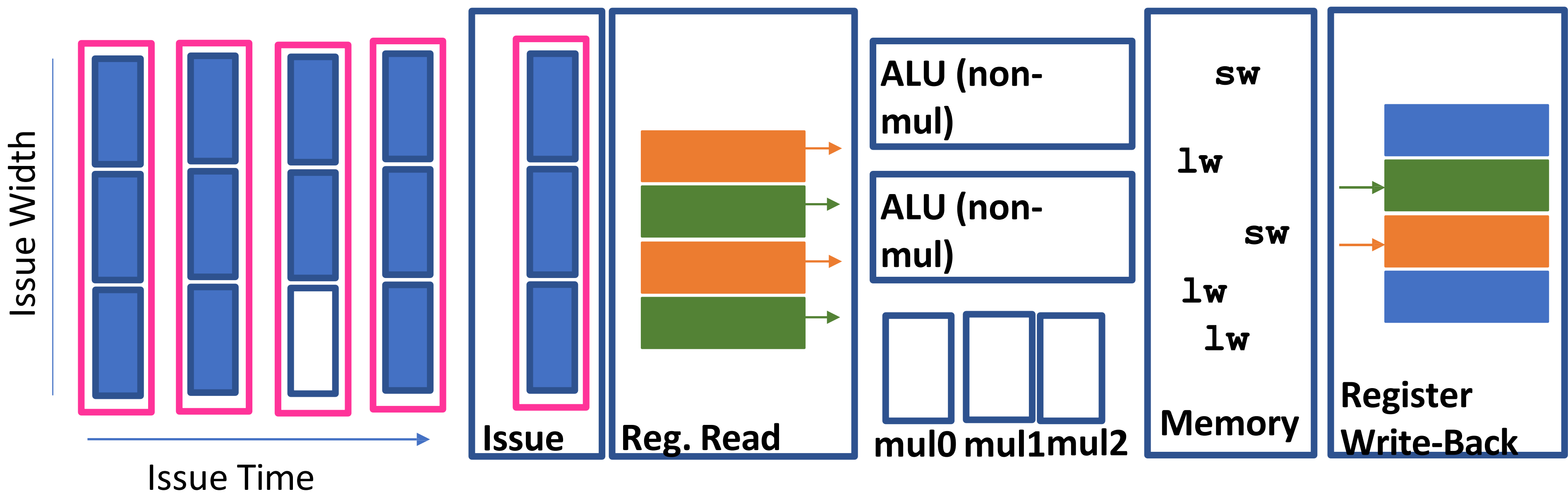
"/" indicates a "stop", break parallelism.

Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

What do we rely on for VLIW to work? What assumptions do we depend on for VLIW to work and be efficient?



EPIC/IA-64 bundles up to 3 instructions with a *type* that says whether & how they're dependent or parallelizable



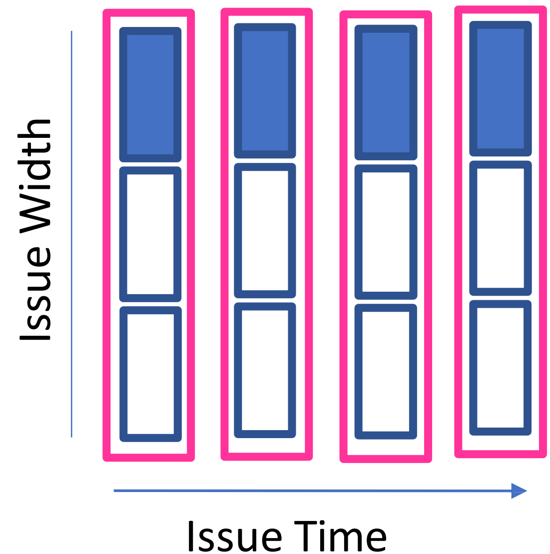
Type:
Mem, Float,
Int, Long Imm.
Branch
e.g.,
MMI, IIF, MMI
MM/I, M/MI

"/" indicates a "stop", break parallelism.

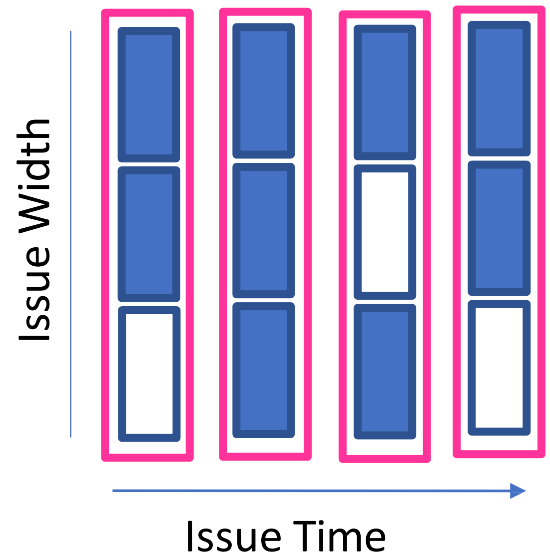
Very Large Instruction Word (VLIW) and the EPIC Architecture (Explicit Parallel Instruction Computer)

Software-constructed (compiler-constructed) bundles of instructions can come from anywhere

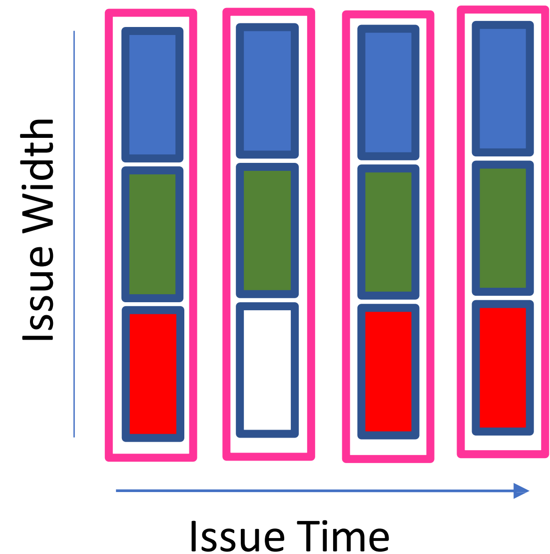
EPIC assumes ***in-order execution*** (static scheduling) and presence of ILP exploiting features, e.g., branch pred., load speculation



Like single-issue scalar execution



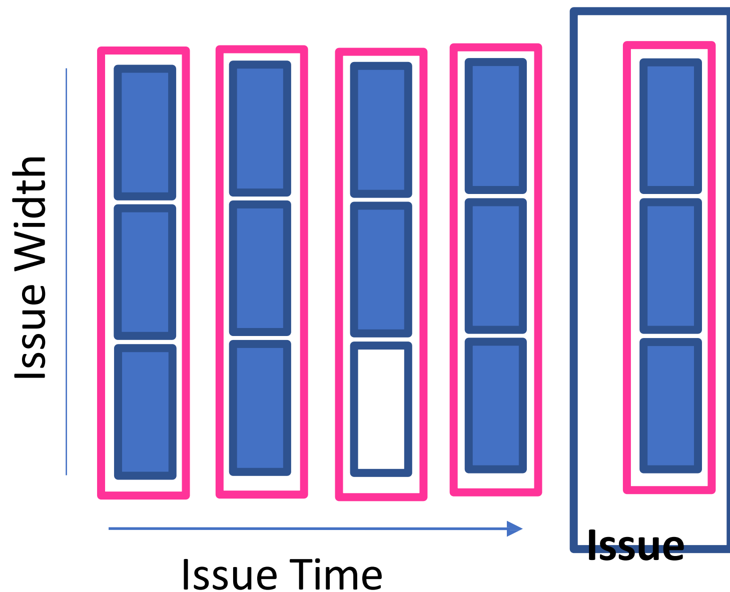
Like multi-issue superscalar execution



Like SMT superscalar, exploiting thread-level parallelism in prog.

Superscalar OoO is great at finding ILP to reduce CPI, but EPIC eliminates dynamic scheduling. Why?

Question: how can static scheduling be good enough to justify eliminating dynamic scheduling & SS/OoO?

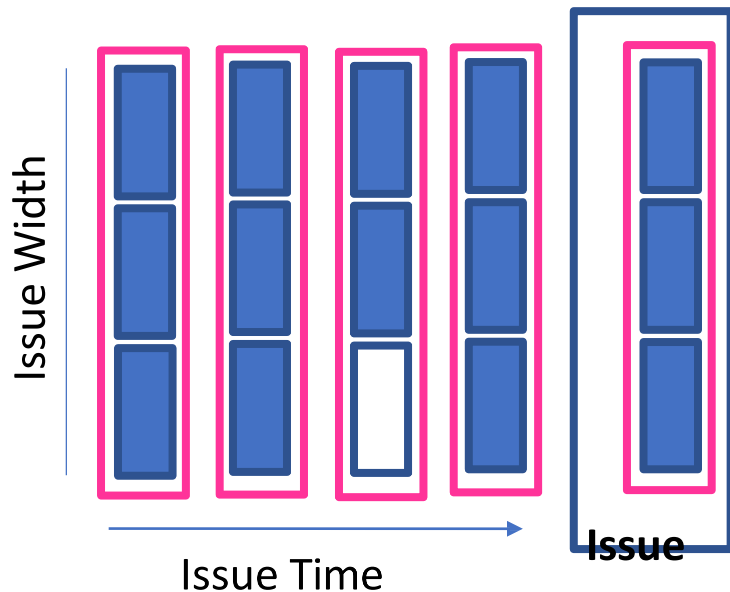


Goal for static & dynamic scheduling: Find instructions to keep the issue window full at all times.

```
ld x11 (x8)
add x2 x3 x4
mul x4 x10 x11
mul x10 x8 x9
st x10 (x8)
add x6 x8 x11
mul x9 x6 x13
add x6 x12 x14
```

Dynamic Scheduling vs. Static VLIW

Question: how can static scheduling be good enough to justify eliminating dynamic scheduling & SS/OoO?



Goal for static & dynamic scheduling: Find instructions to keep the issue window full at all times.

Dynamic scheduling has a limited scope for analysis and optimization

Short window limits reordering distance

```
ld x11 (x8)
add x2 x3 x4
mul x4 x10 x11
mul x10 x8 x9
st x10 (x8)
add x6 x8 x11
mul x9 x6 x13
add x6 x12 x14
```

Static scheduling has *global* scope for reordering / optimization

Long window allows long reorderings

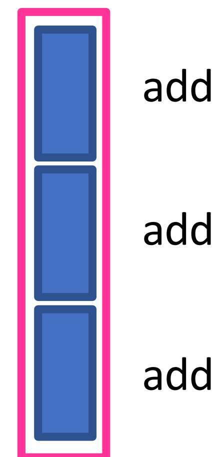
```
ld x11 (x8)
mul t1 x10 x11
mul x10 x8 x9
… //other ops taking t_mul cycles
add x2 x3 x4
st x10 (x8)
```

Latency = t_mul

At this point, muls are done and we keep rolling, overlapped latency.

Effective scheduling relies on approximately equal execution latency for all instructions

- If some instructions in a bundle are long-latency and others short-latency, the longs delay the shorts
  - *Scheduling same-latency ops together keeps the machine moving*
- **What about unpredictable latency instructions like memory operations?**

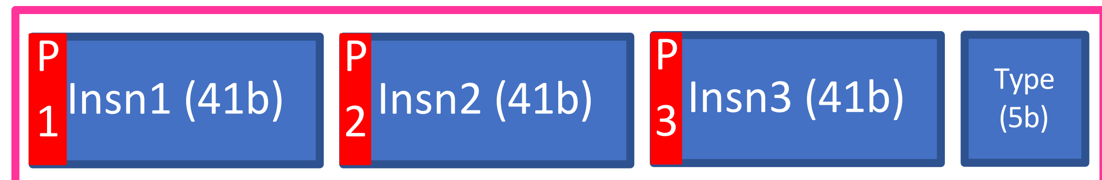


Effective scheduling relies on approximately equal execution latency for all instructions

- If some instructions in a bundle are long-latency and others short-latency, the longs delay the shorts
  - *Scheduling same-latency ops together keeps the machine moving*
- **What about unpredictable latency instructions like memory operations?**
  - **Unpredictable stalls in the pipeline waiting for memory operations**.
  - Can tolerate latencies by scheduling same-latency operations together, if compiler has an expectation about memory latency, cache structure, and producer / consumer relationships.
    - **This is a very difficult compilers problem!!**

Branch instructions in EPIC

- **EPIC / VLIW does branches differently than in SS/OoO**
- **Option 1:** Waste space in a bundle, run branches like in SS/OoO
  - if taken, grab taken bundle, if not, grab sequentially next bundle
- **Option 2:** Predication
  - run both sides of branch and commit only insns with true predicate
- Predication takes pressure off of control logic & branch prediction (why?)
  - Do we need a branch predictor?
- Costs of predication?



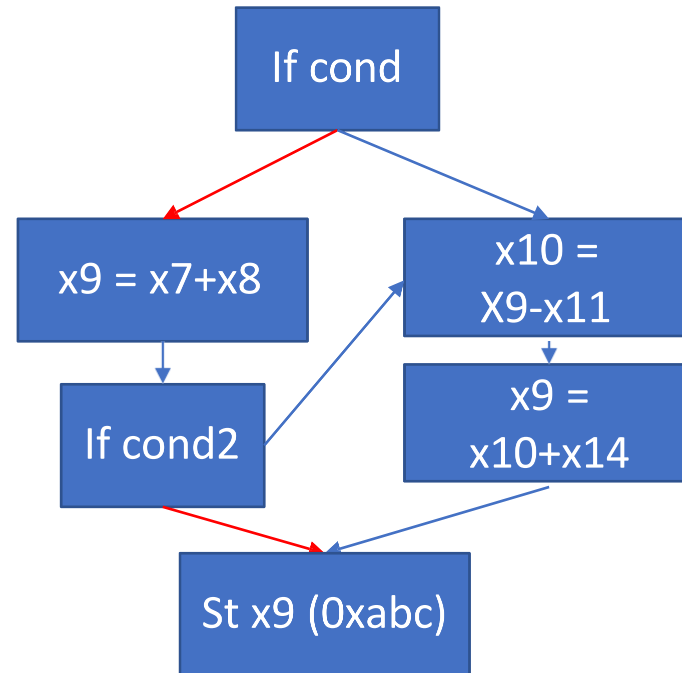
If !P *nullify* insn

If conversion

The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```

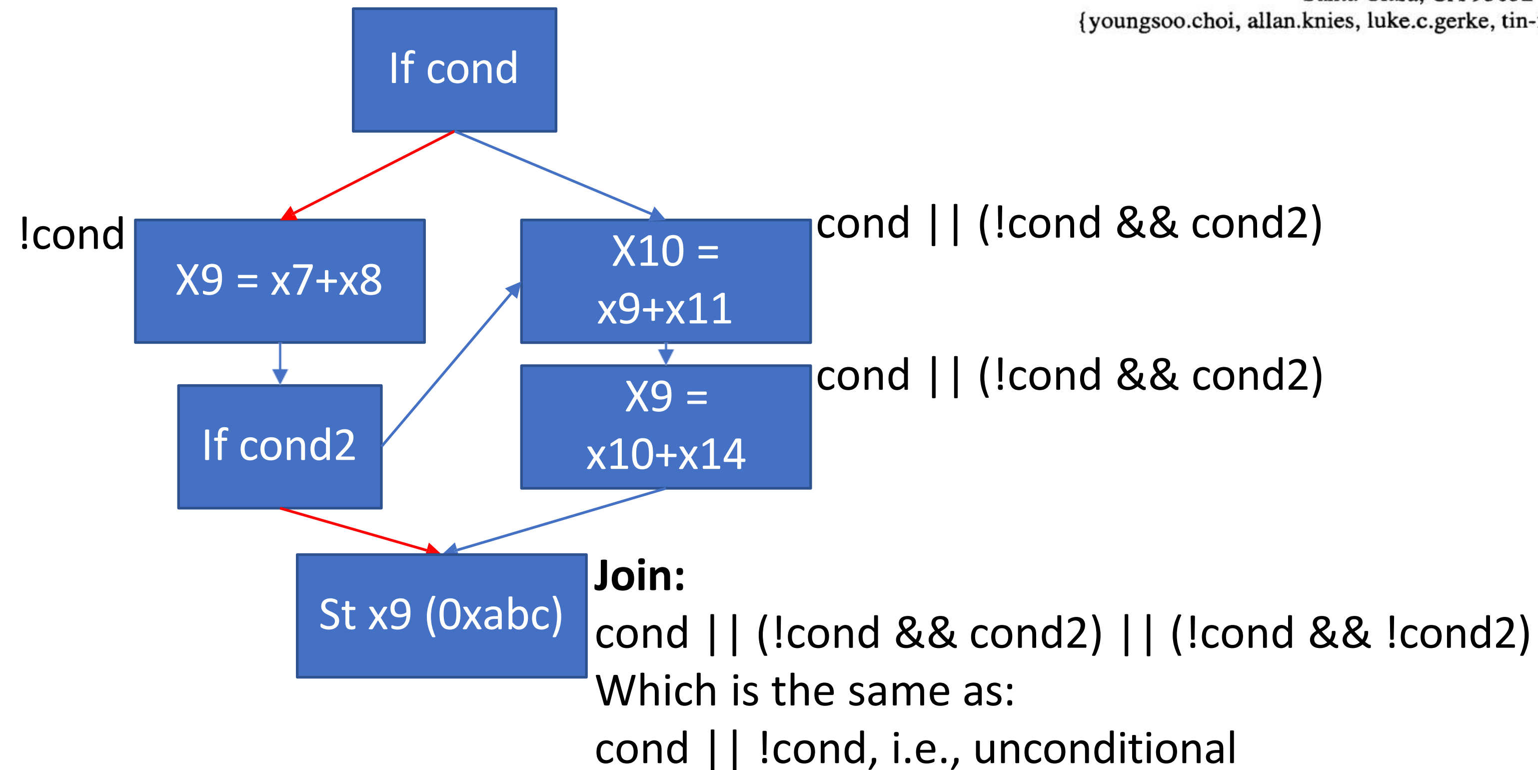


If conversion

The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```

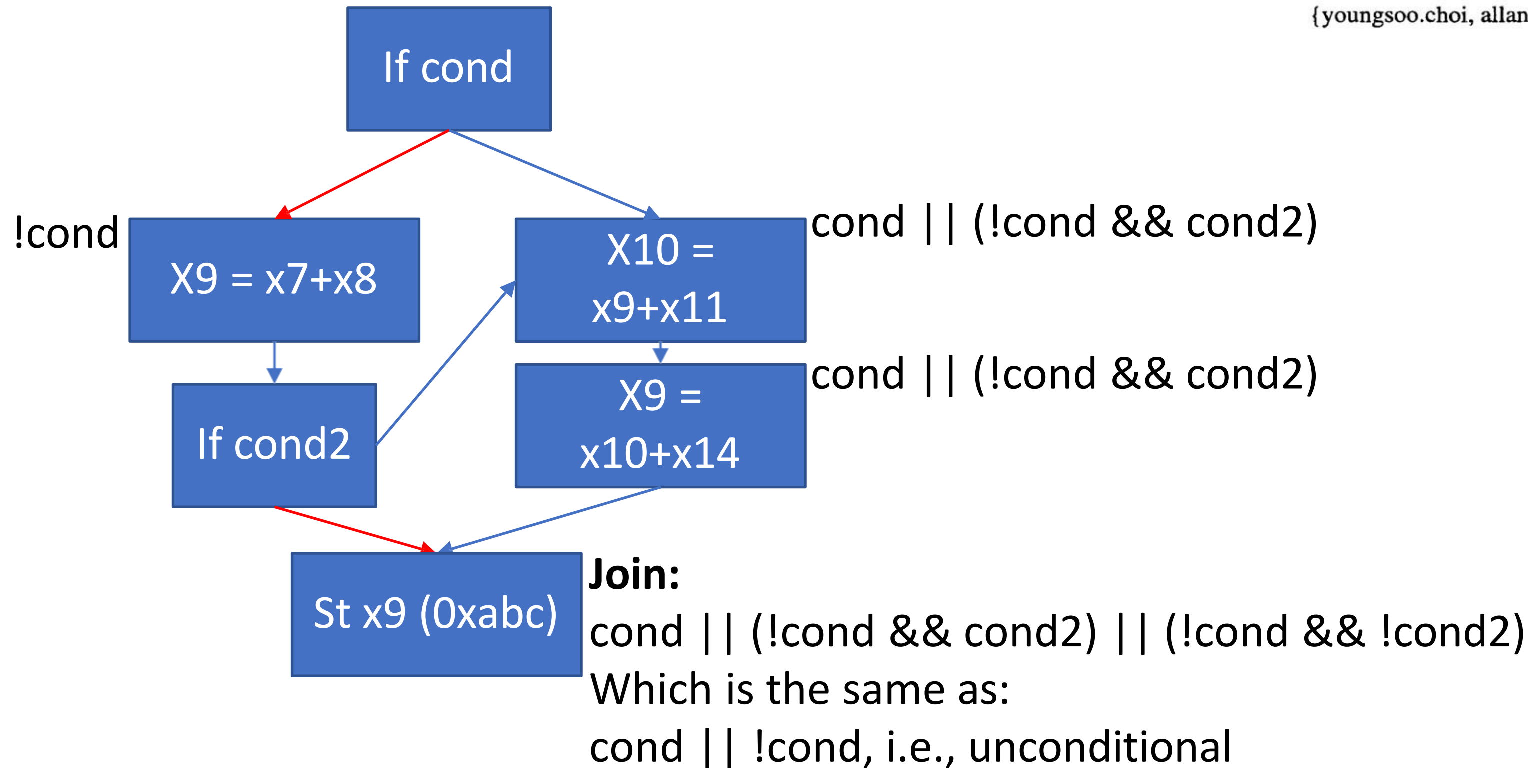


If conversion

The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

```
bne cond, pc+12
Add x9 x7 x8
bne cond2 pc+12
Sub x10 x9 x11
Add x9 x10 x14
St x9 (0xabc)
```



Store predicate results in explicit predicate registers

```
P1=!cond
P2=cond||(!cond&&cond2)
(P1)Add x9 x7 x8
(P2)Sub x10 x9 x11
(P2)Add x9 x10 x14
St x9 (0xabc)
```

Add explicit predicates to the code that executes. Predicates evaluate **dynamically** using predicate registers.

No branch instructions here!
Microarchitectural implication?

Pipeline Characteristics

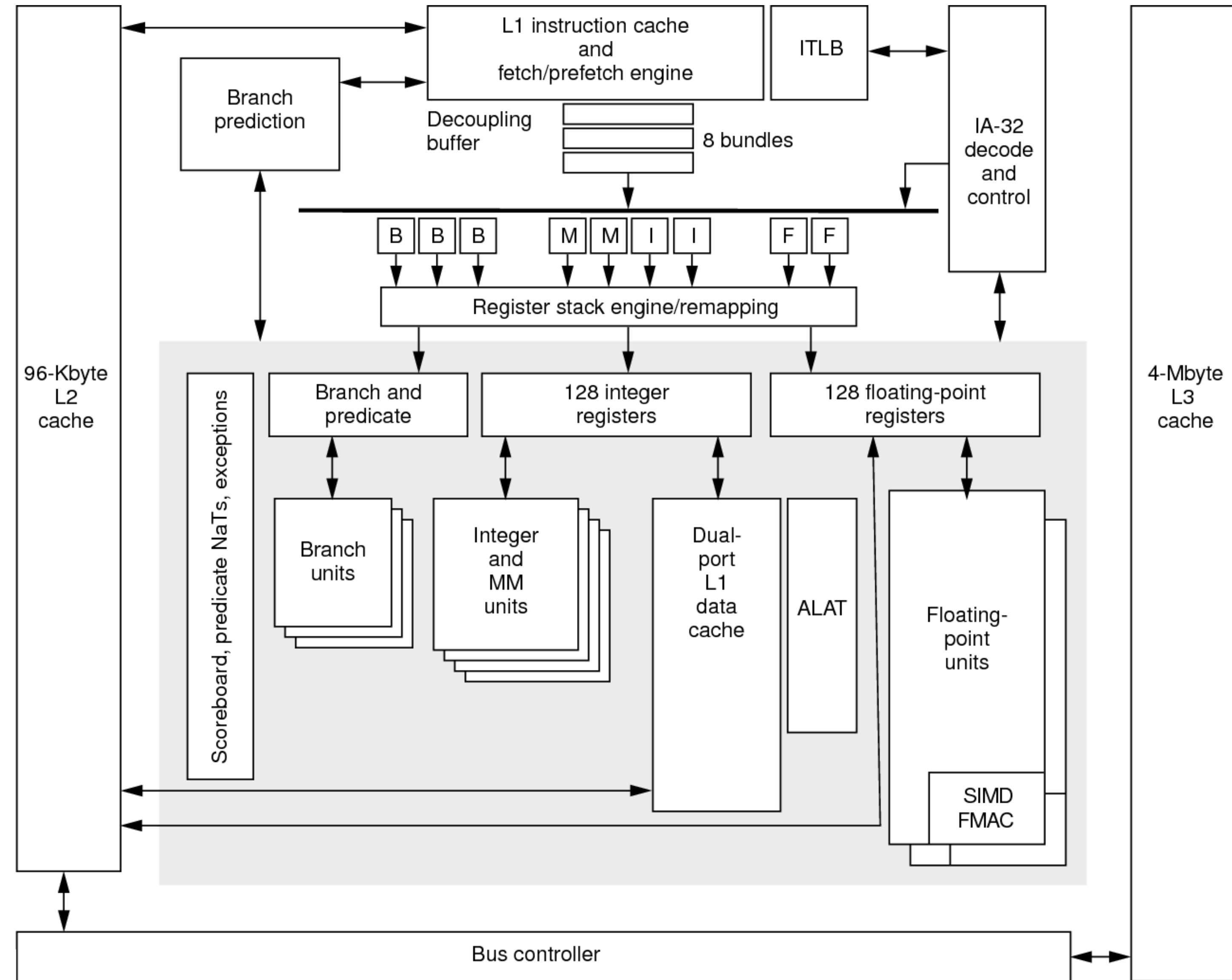
- Execute 2 bundles (6insns) per cycle
- 10 stage pipeline
- 4 Integer Units (2 of which do Ld/St)
- 2 Floating Point Units
- 3 Branch Units
- Issue in order, execute in order
- Simple register dependence tracking using a “scoreboard”

Control Characteristics

- Predication *and* sophisticated two-level branch predictor (why?)
- Instruction queues connect fetch to execute units hiding some fetch bubble latency with execute latency (how?)

Register File

- Fairly complex and highly abundant
- Separate predicate / branch, int, and FP regs
- “Register stack engine” efficiently doles out physical registers, avoiding structural hazard

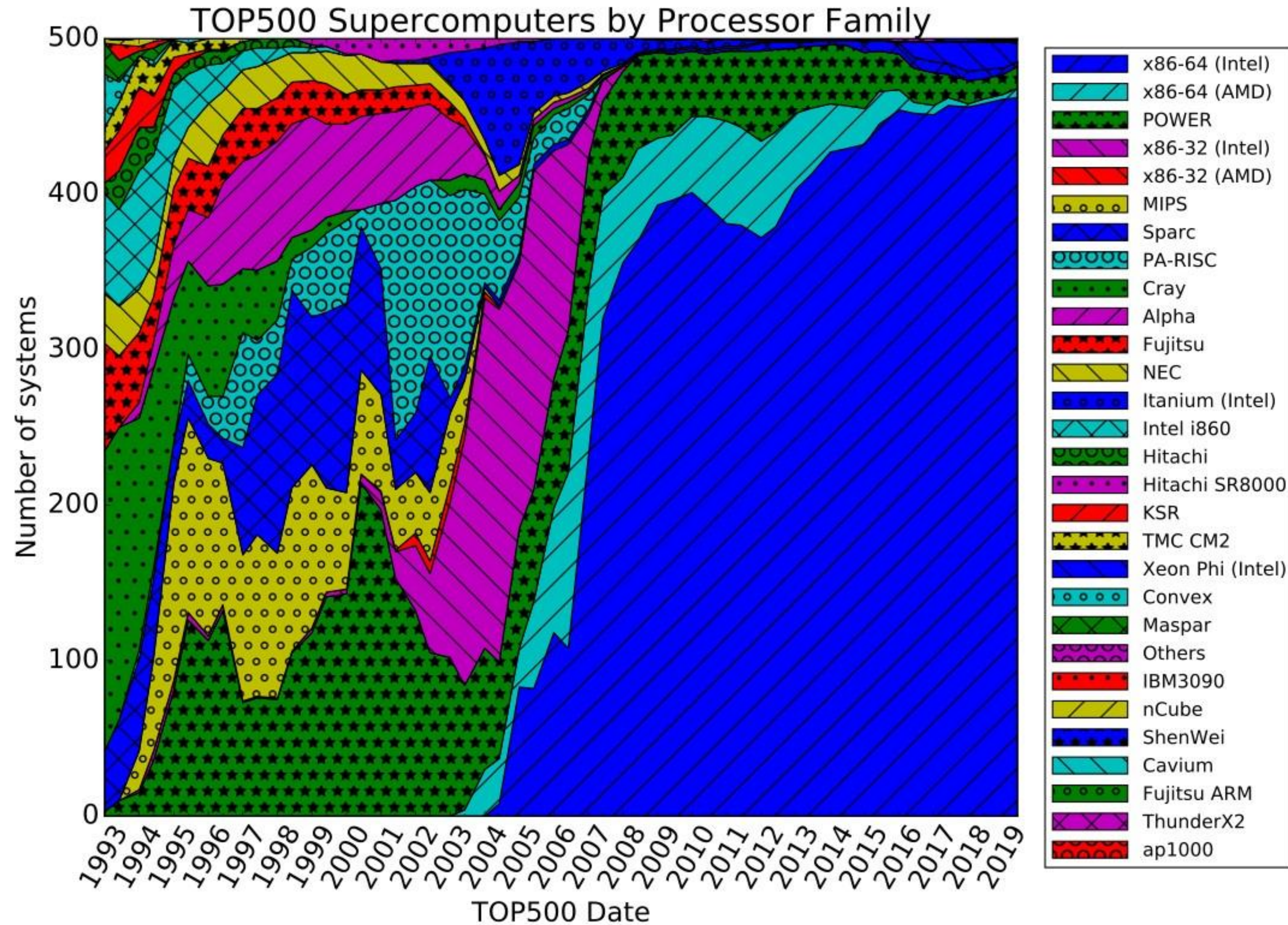


Intel Itanium EPIC Architecture

VLIW / EPIC is a Very Cool HW/SW Interface!

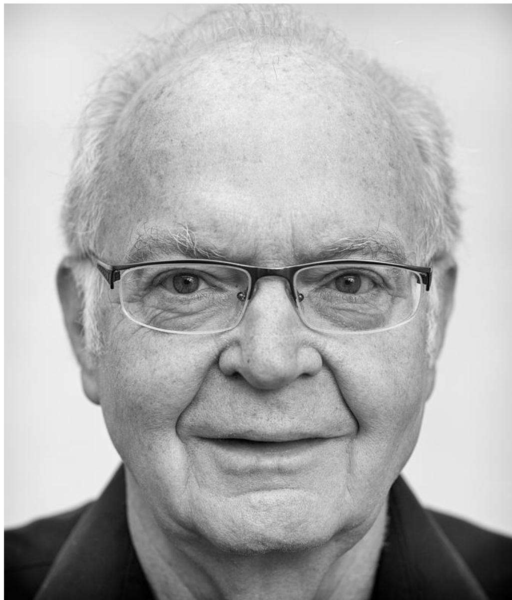
- Why did Itanium not seize the (any?) market as Intel anticipated?

- (In the top500 supercomputers, we mostly have x86-64, not IA64)



VLIW / EPIC is a Very Cool HW/SW Interface!

- Donald Knuth: *“the "Itanium" approach [was] supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write”*



TOP500 Supercomputers by Processor Family

Number of systems

0
100
200
300
400
500

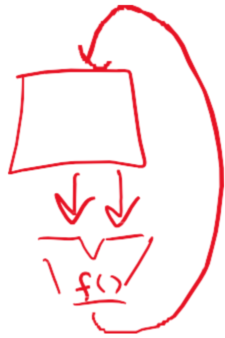
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019

TOP500 Date

x86-64 (Intel)
x86-64 (AMD)
POWER
x86-32 (Intel)
x86-32 (AMD)
MIPS
Sparc
PA-RISC
Cray
Alpha
Fujitsu
NEC
Itanium (Intel)
Intel i860
Hitachi
Hitachi SR8000
KSR
TMC CM2
Xeon Phi (Intel)
Convex
Maspar
Others
IBM3090
nCube
ShenWei
Cavium
Fujitsu ARM
ThunderX2
ap1000

Parallelism Beyond ILP

Flynn's Taxonomy of Parallel Architectures



SISD

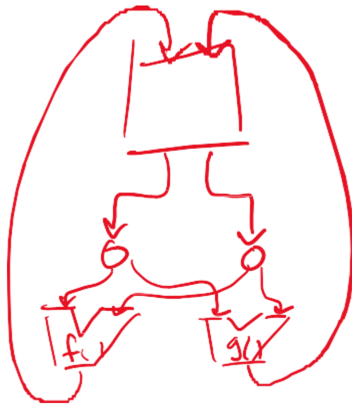
"sequential"

incl. SS/OoO

"vector"

"array"

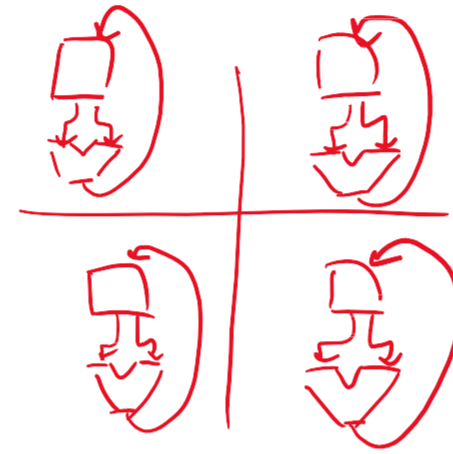
SIMD



MISD

"streaming"

"replicated"



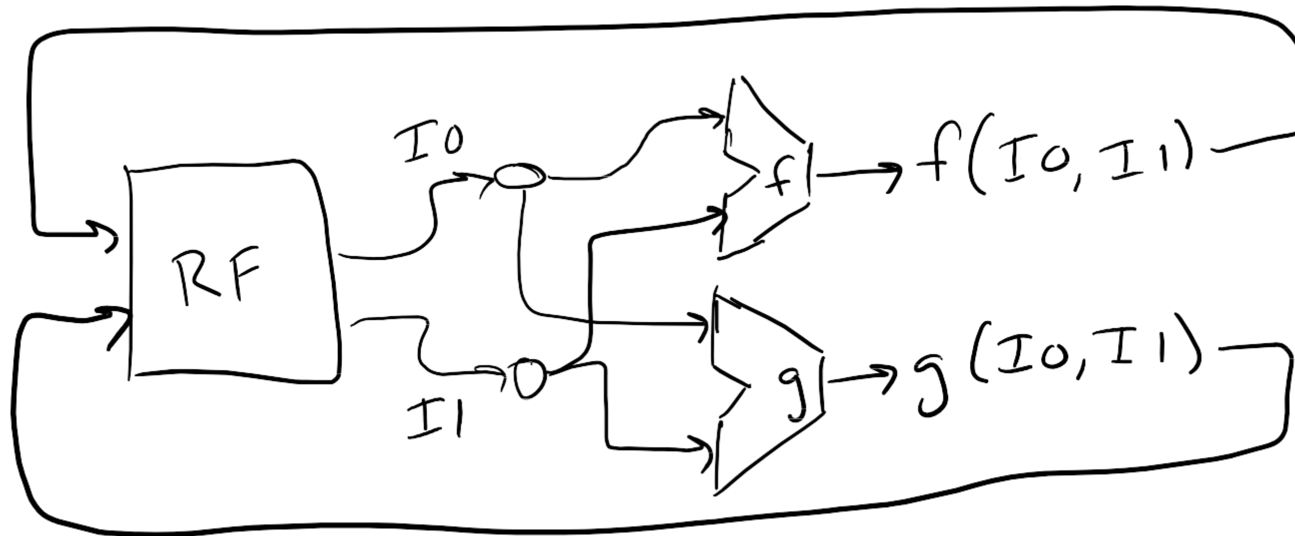
MIMD

"multicore"

Later this sem.

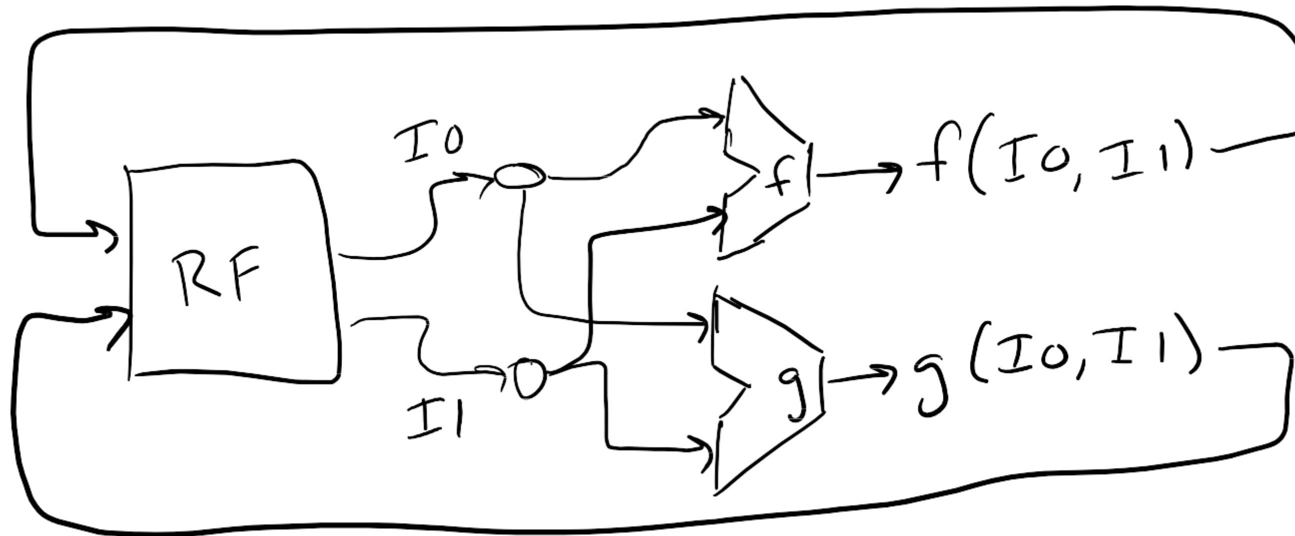
MISD – Multiple Instruction Single Data

- Send same inputs (logically) simultaneously to multiple functions
- Used for …what?



MISD – Multiple Instruction Single Data

- Send same inputs (logically) simultaneously to multiple functions
- Rare, sometimes used in DSP, filter signal using multiple programs
- Modular **redundancy**, replicated hardware for execution (if f == g)



SIMD – Single Instruction Multiple Data

Apply instruction to many data: Single instruction (fetched, decoded, etc) applies its operation to a large number of data elements

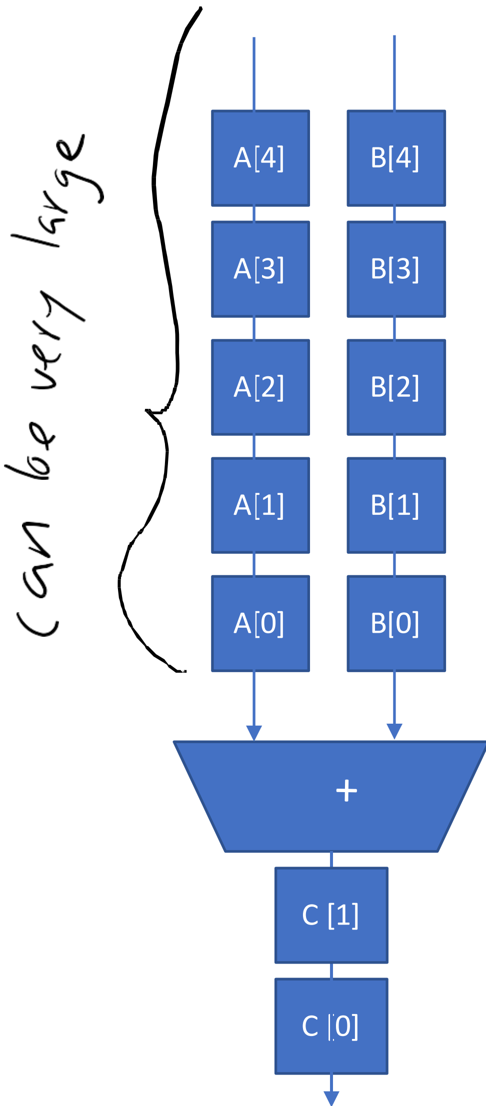
Amortizes Instruction Costs: Each instruction corresponds to a large number of operations

A Few Flavors: most notable/effective are ***Vector*** machines

- Also: historically, ***Array Machines*** but these are not widely used anymore

Data-oriented, Not Necessarily Parallel: Instructions specify what do to to each element of data, but not how to do it (i.e., parallel, partially parallel, sequential)

Vector Machines



`setvl`: tell machine length of input vector. Actual in-memory length can be *thousands* of elements! Machine returns max it can handle in a vector register (varies by implementation, can be tens of elements)

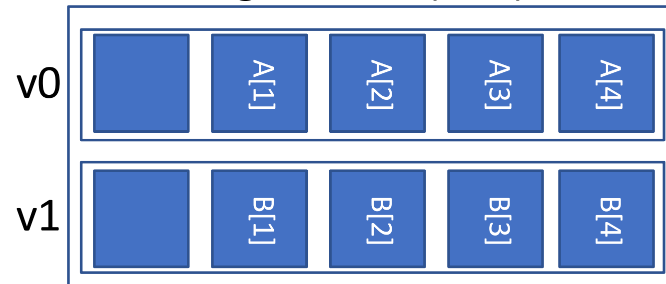
`vld <vector register>, <mem>`: load vector of length `vl` starting at memory location <mem>

`vst <vreg>, <mem>`: store elems of <vreg> to memory starting from memory location <mem>

`vadd <vreg1> <vreg2>`: add element-wise store into vreg1

Assumes explicit vector register file that can temporarily store vector operands

vector register file (VRF)




```
setvl 5
vld v0, a
vld v1, b
vadd v0, v1
vst v0, c
```

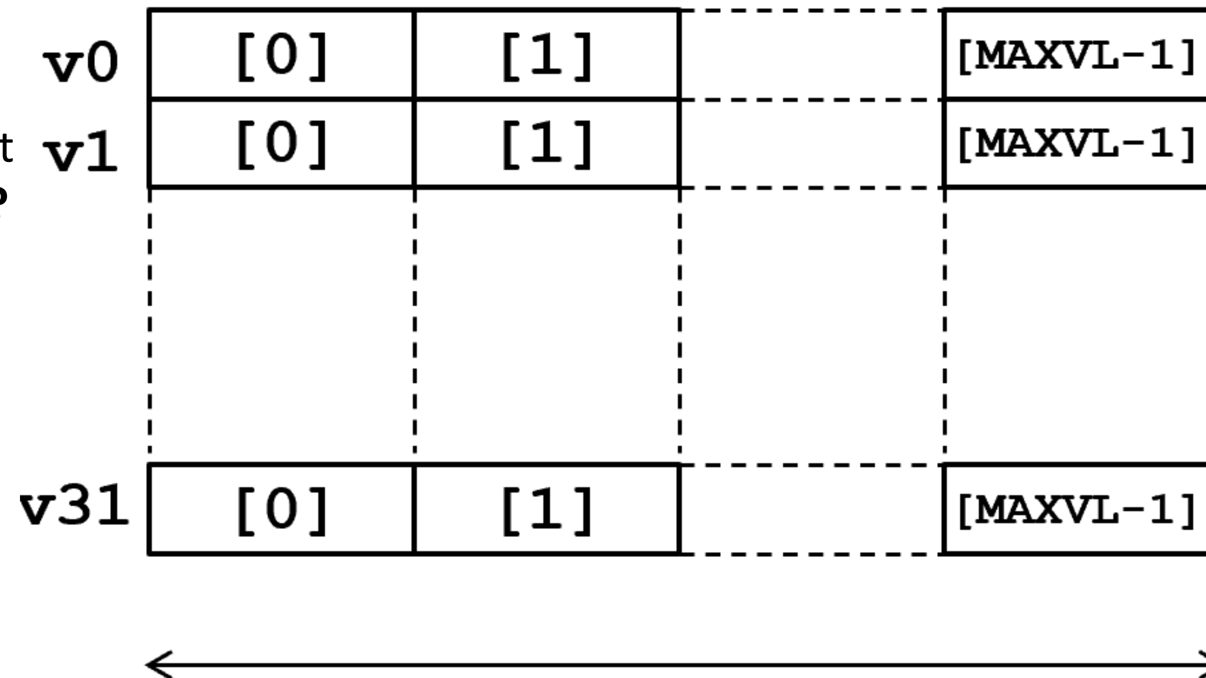
Vector register file

Large, performance-critical structure accessed potentially many times per-cycle during vector operation. **How large? How critical?**

maxvl is an implementation-dependent parameter. **How do we (architects) set maxvl?**

If **setvl** sets **vl** to greater than **maxvl,** then **vl** gets set to **maxvl. HW/SW consequence?**

If **setvl** sets **vl** to less than **maxvl,** then the excess vectors get set to **0** during ops



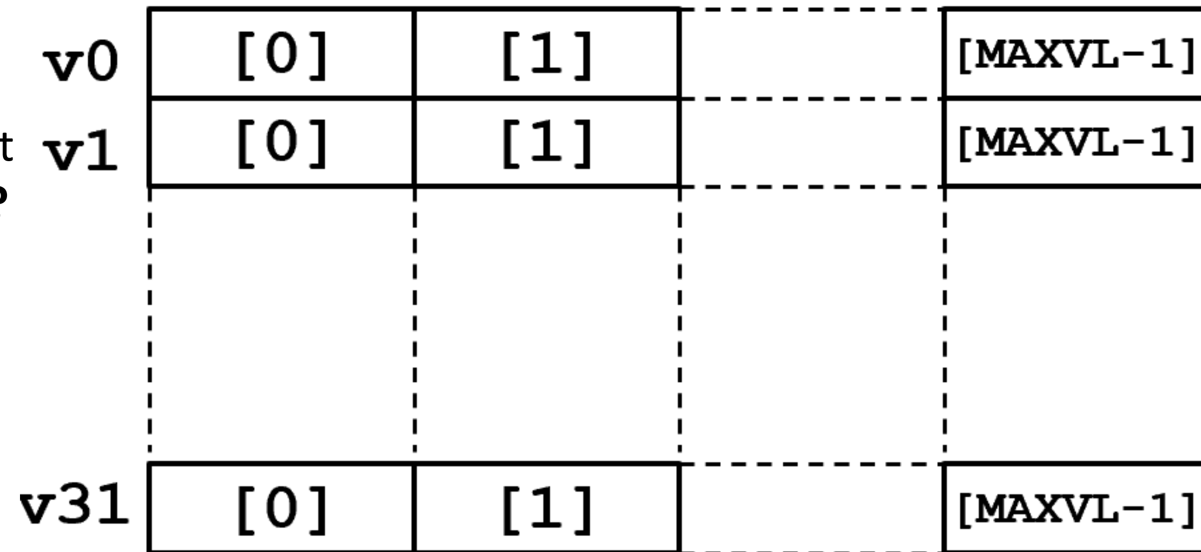
Vector register file

Large, performance-critical structure accessed potentially many times per-cycle during vector operation. **How large? How critical?**

maxvl is an implementation-dependent parameter. **How do we (architects) set maxvl?**

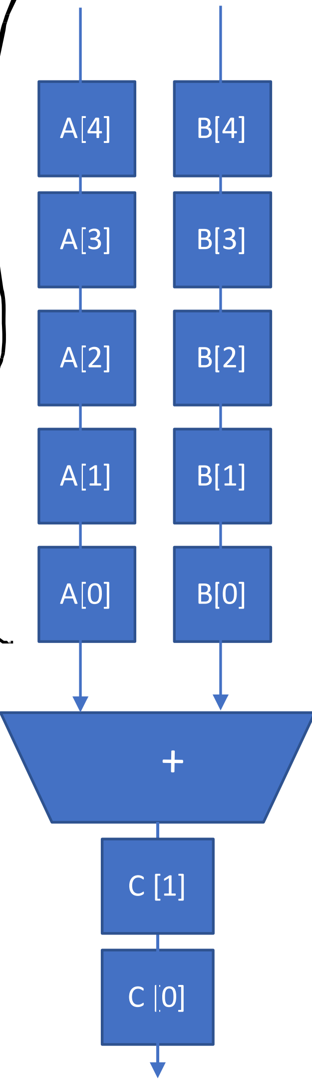
If **setvl** sets **vl** to greater than **maxvl,** then **vl** gets set to **maxvl. HW/SW consequence?**

If **setvl** sets **vl** to less than **maxvl,** then the excess vectors get set to **0** during ops



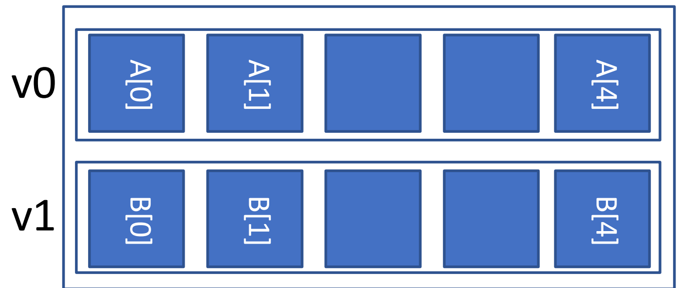
8B / word * 16 words / VRF entry * 32 VRF entries per VRF = 4kB!
Larger than many L1 Caches

Dealing with limited vector size is easy in SW

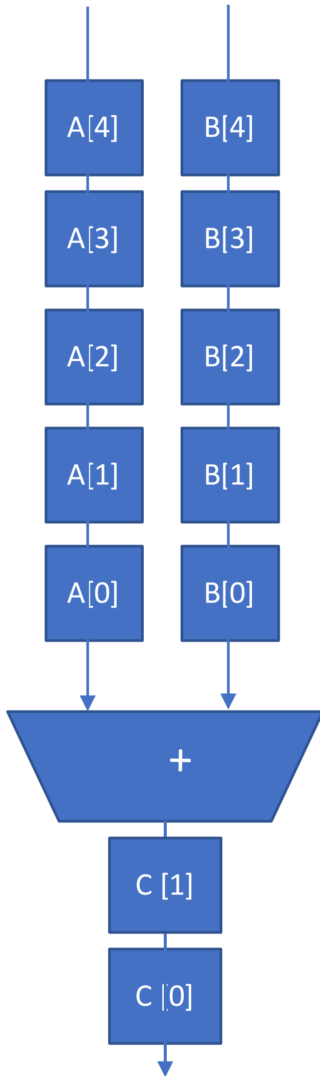



```
setvl 8192 //machine says "vl = 5, actually"
for(i = 0…(8192 / vl) ): //loop vl at a time
  vld v0, a + i*vl
  vld v1, b + i*vl
  vadd v0, v1
  vst v0, c + i*vl
```

maxvl assumed to be 5

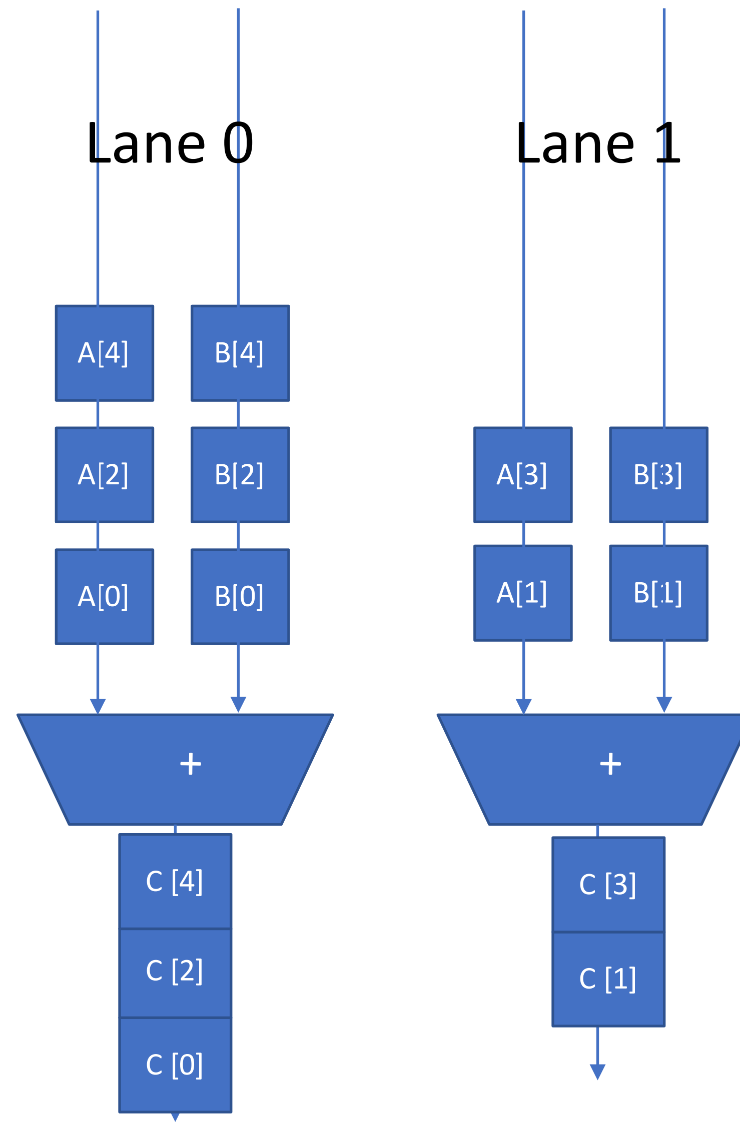


Vector Machines are Easily Parallelizable

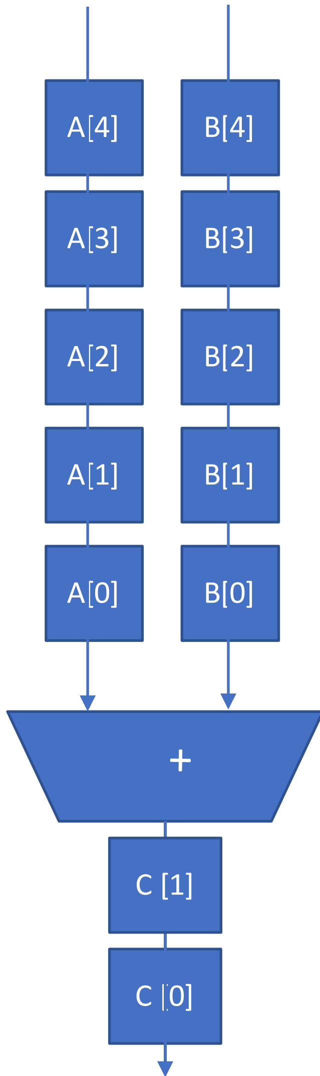


Abstraction: execute an instruction's operation over an entire vector of data

Implementation: Parallel functional units each process parts of a vector, producing a vector output. **Why simple?**



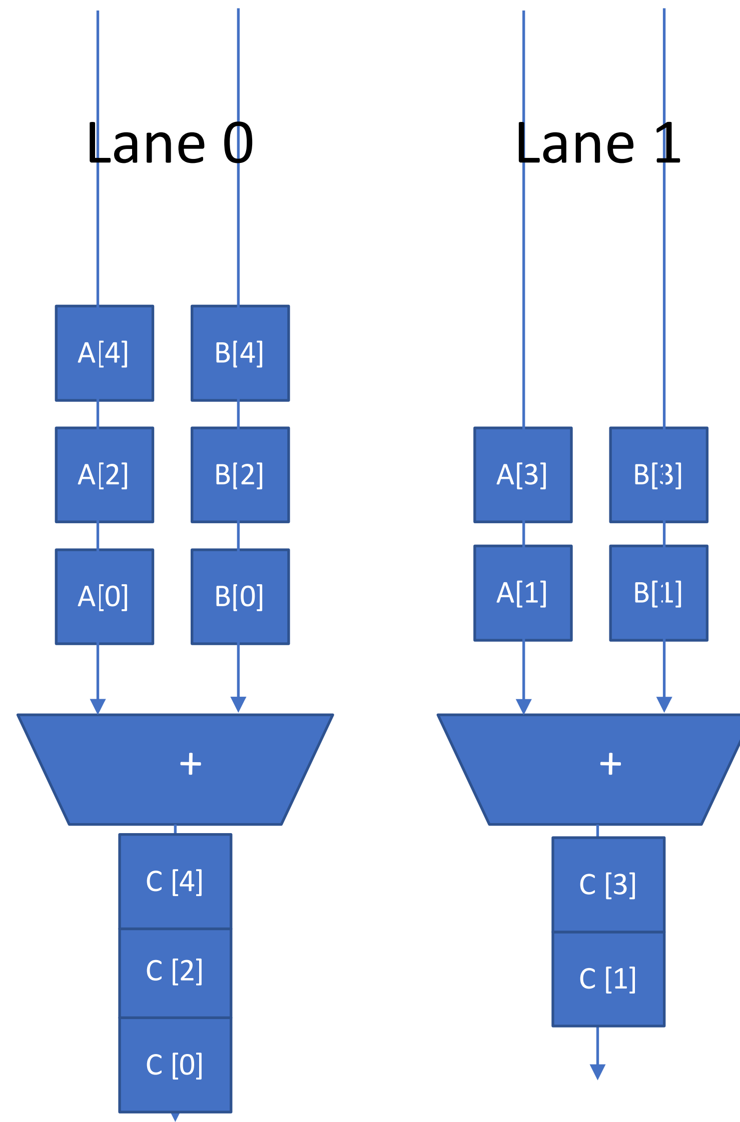
Vector Machines are Easily Parallelizable



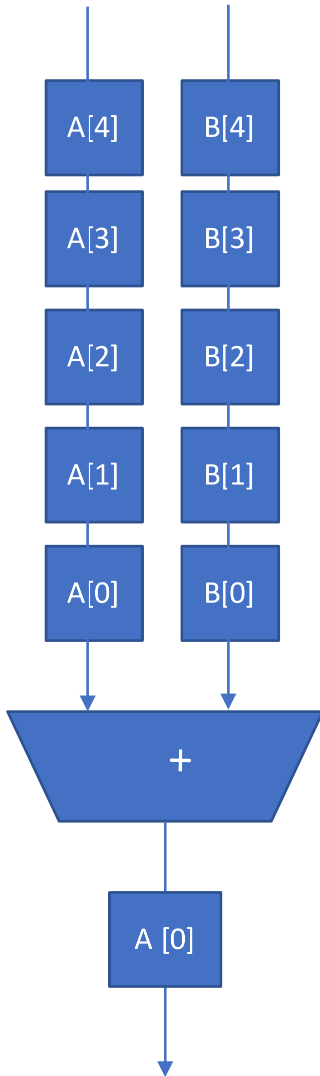
Simple: Vector instruction operates on v0[**i**] and v1[**i**] ***not*** v0[**i**] and elem *v.

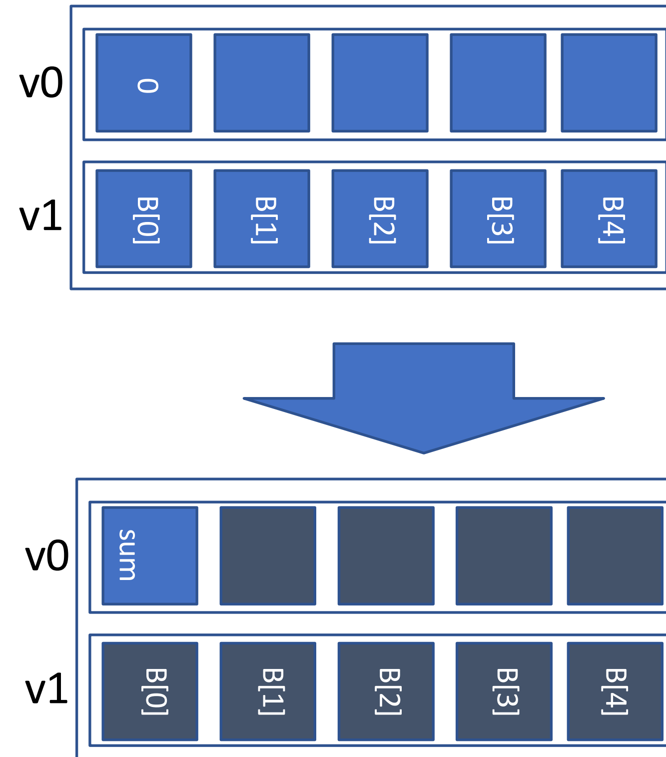
Very simple operand matching logic, no need to track complex producer consumer relationships across inputs of operations.

Primary cost?



Reduction Operations






```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

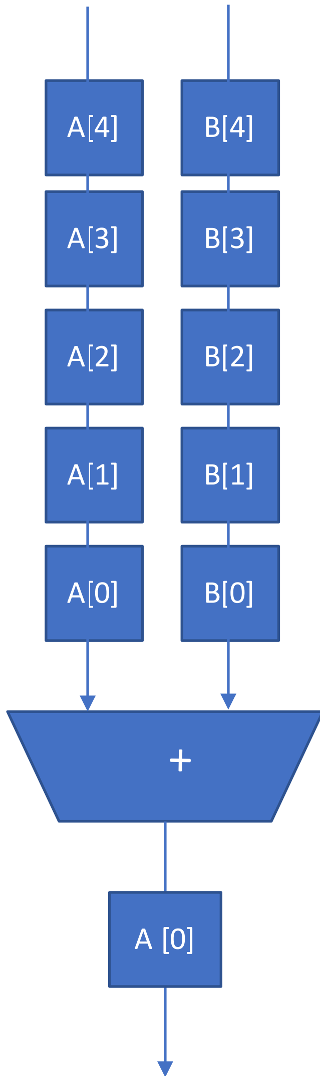
dest vreg [0]

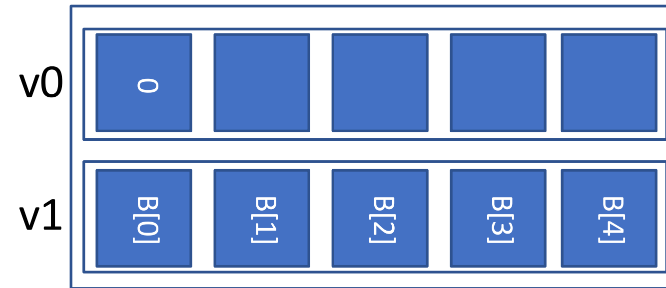
vreg to reduce

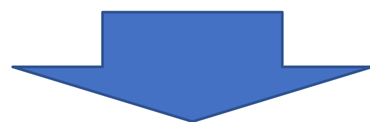
initial value

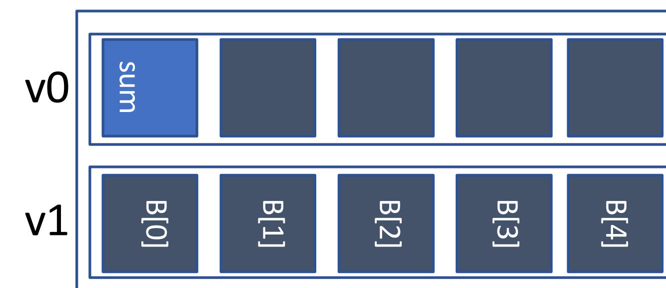
```
v0[0] =
v0[0] + Σ_i v1[i]
```

Reduction Operations










```
setvl 5
vld v0, a
vld v1, b
vredsum v0, v0, v1, vm
```

v0[0] =

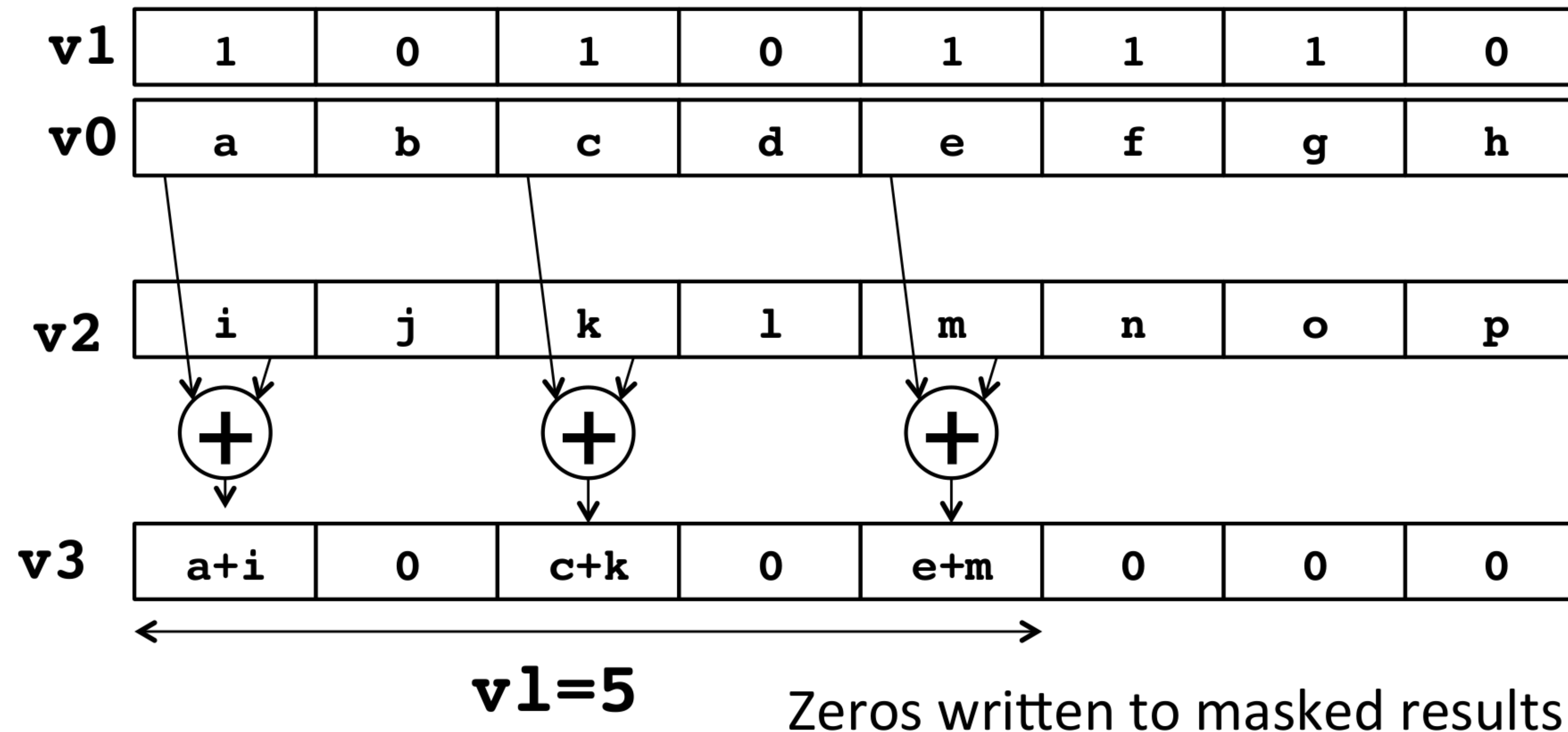
v0[0] + Σ_i v1[i]

Vector Masking

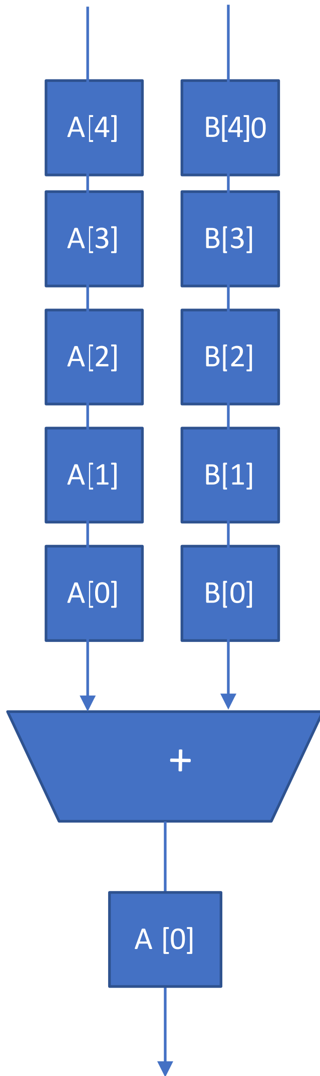
```
vadd v3, v0, v2, v1.t
```

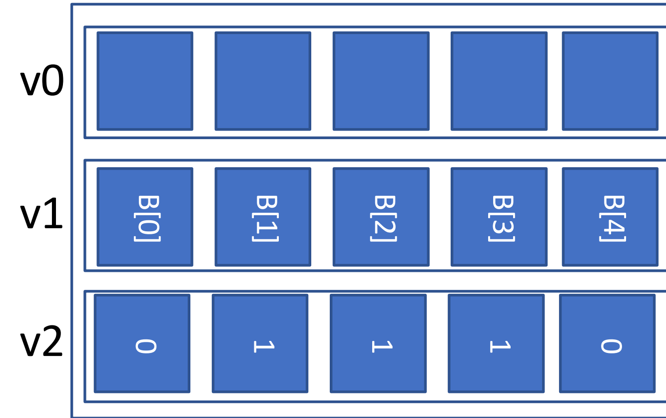
Behavior of a masked vector operation: For elements up to vl in v3, add elements from v0 and v2 if that element in v1's LSB is set to 1, set other v3 elems to 0

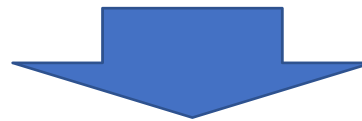
What high-level programming concept does this get used to implement?

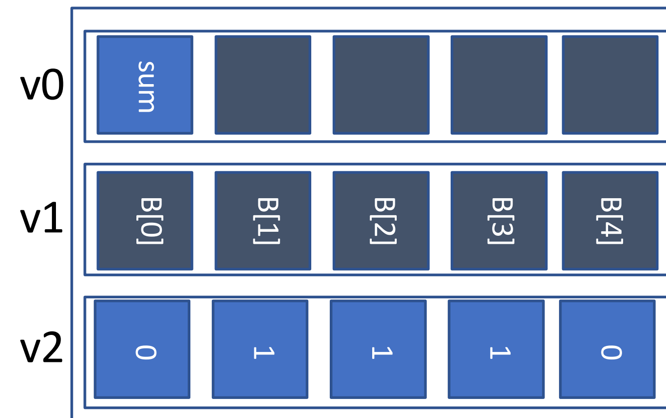


Reduction Operations with a vector mask









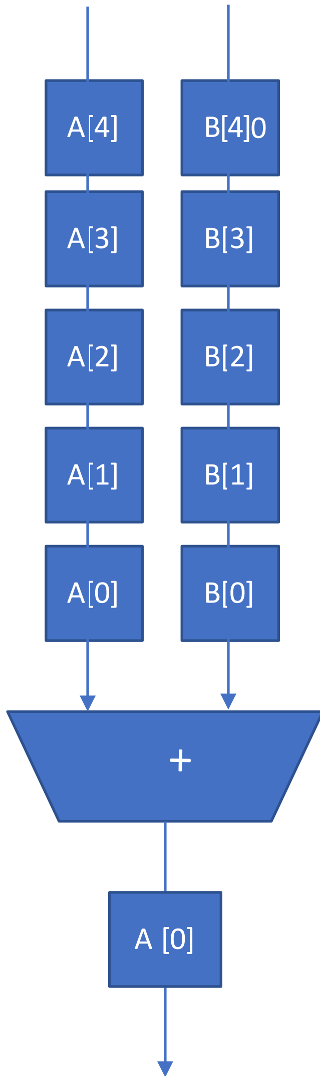

```
setvl 5
vld v0, a            input
vld v1, b            vec
vredsum v0, v0, v1, v2
                       mask
           dst   init
                 val
```

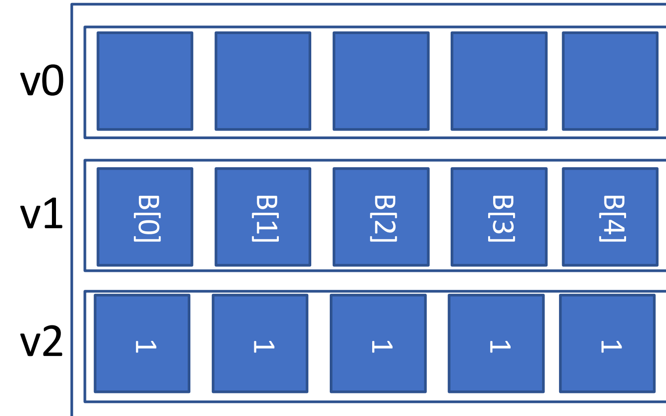
Reduction operations accumulate the result of an operation on a vector into the first element of a destination vector

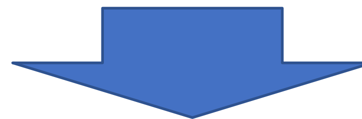
Uses for reduction?

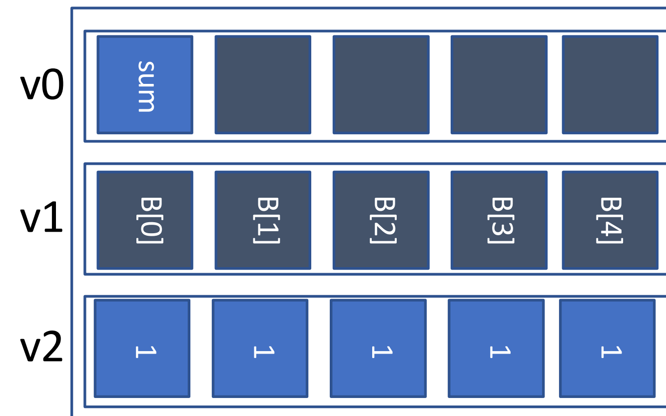
```
v0[0] =
v0[0] +
v1[1] + v1[2] + v1[3]
```

Reduction Operations with a vector mask







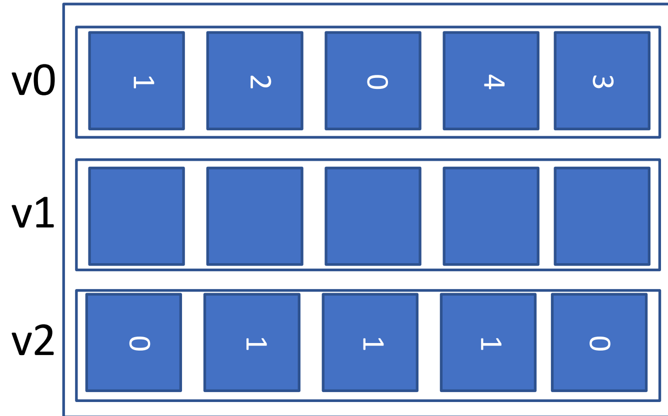


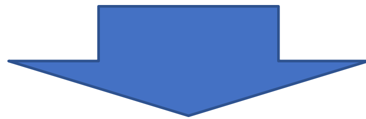
Uses for reduction?
Dot product, e.g.,

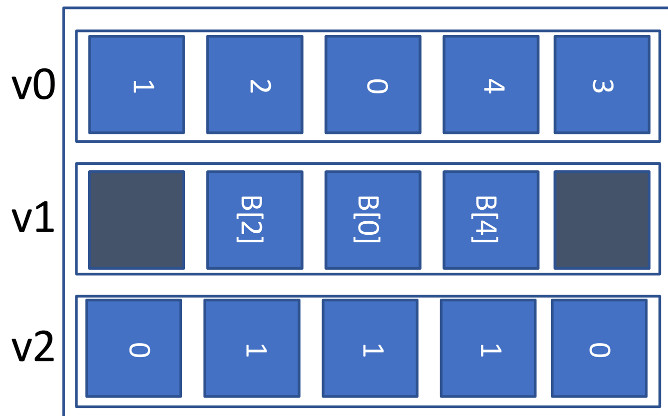
```
setvl 5
vld v0, a
vld v1, b
vmul v0, v1
vredsum v0, v0, v1, v2
```

```
for( i = 0..len){
  v[i] += a[i] * b[i]
}
```

Indexed Memory Accesses (Scatter/Gather)







index vector

dest

```
vluxei64 v1, (&B), v0, v2
```

base addr

mask

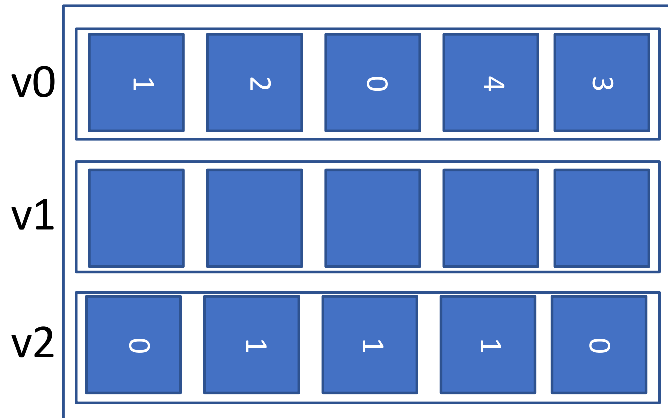
Indexed memory loads **"gather"** elements from all over memory into a contiguous vector register.

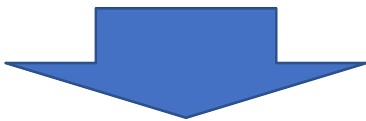
Indexed memory stores **"scatter"** elements from a contiguous vector register into locations all over memory

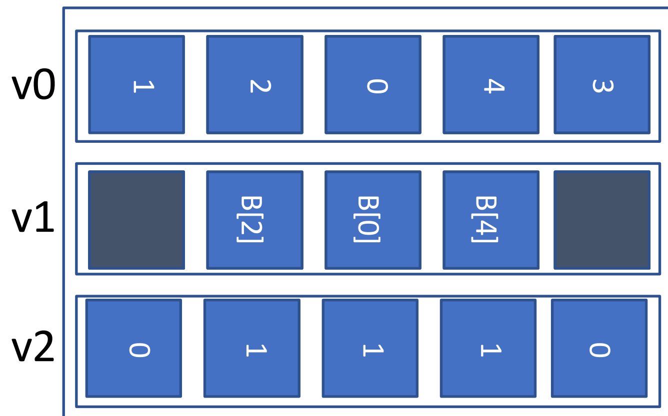
Uses?

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Indexed Memory Accesses (Scatter/Gather)







dest, base addr, index vector, mask

```
vluxei64 v1, (&B), v0, v2
```

Common Use: indirect array accesses. Common in graph analytics

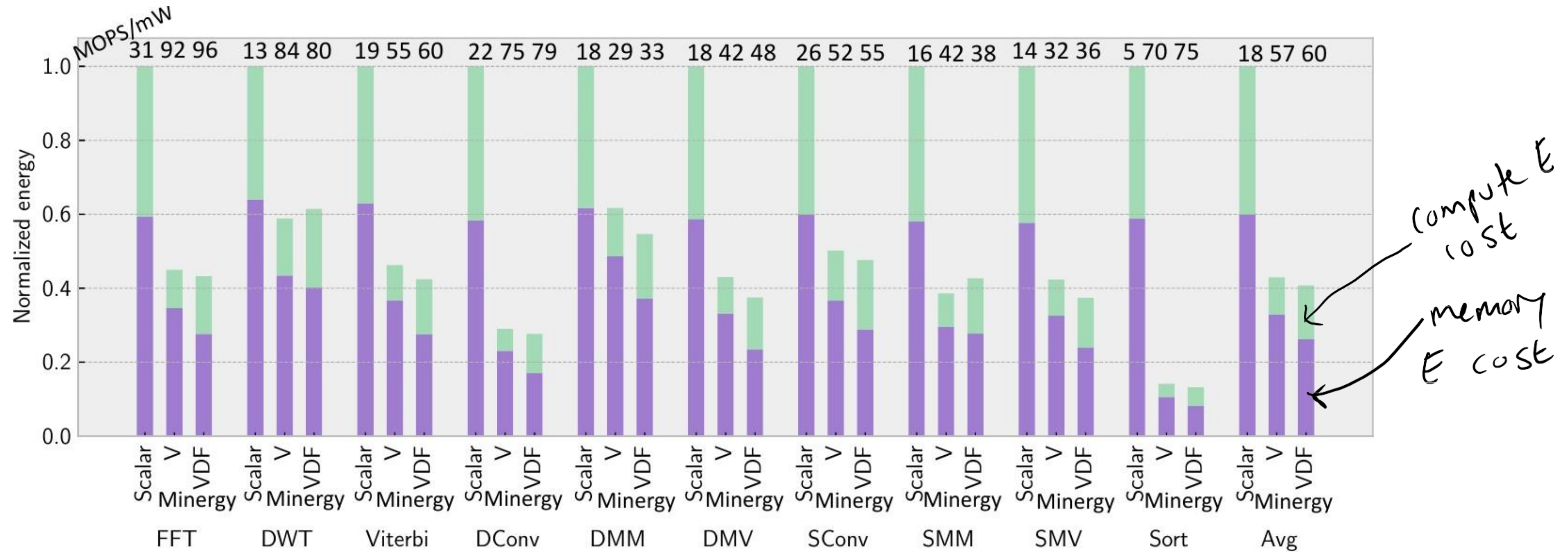
```
for( src in 0 .. n ){
  for( dst in 0..ind[src].len() ){
    data[ ind[src][dst] ]++;
  }
}
```

```
v1[i] = v2[i] ? B[v0[i]] : v1[i]
```

Summary of Benefits: Vector Architectures

- **Compared to scalar architectures:**
  - **Single instruction performs many operations:** one instruction is the equivalent of executing an entire loop of a program!
  - **Control is simpler:** no loops, no branches, no misprediction/misspeculation
  - **Vector interface makes data-independence across vector elements explicit:** simplifies implementations and eliminates complex dependence logic
  - **Dependence checking of vectors, not elements:** what dependence tracking is required pertains to entire vector registers, not individual elements, amortizing its cost significantly
  - **Easy to express data parallelism:** avoids software complexity of multithreading on a multiprocessor (i.e., MIMD)
  - **Maximize value of memory bandwidth:** contiguous/strided vector fetch operations are a good match for highly-banked memories
  - **Energy efficiency**: instruction & data fetch amortize costs across vector saving energy
  - **Require vector programming style, which means changing all of your code. Code doesn't match vector style well? Can't use the vector architecture without lots of extra work!**

Vector execution model saves energy (and time) over scalar processing



Taken from a very recent research project about optimizing for minimum energy by using a new vector processor (**V** bars in the plot) and a customized variant (**VDF** bars in the plot). **V/VDF** use RISCV vector insns., **scalar** plain RISCV insns.

Key take-away: vector processing cuts energy by ***more than half*** compared to scalar processing.

What did we just learn?

- We learned about how VLIW and Vector processing are two different takes on the hardware software boundary that admit more parallelism than SS/OoO’s ILP focus allows
- VLIW did not take over, vector has been a consistent background hum
- Both approaches require the programmer and the compiler to make big changes to code to work well with these new hardware/software interfaces.

What to think about next?

- Next we look at Virtual Memory as an abstraction
- Also look at the underlying mechanisms and options for implementing virtual memory in a modern CPU