CS250P: Computer Systems Architecture
Explicit Parallelism

Sang-Woo Jun

Fall 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Modern Processor Topics - Performance

- Transparent Performance Improvements
  - Pipelining, Caches
  - Superscalar, Out-of-Order, Branch Prediction, Speculation, …
  - Covered in CS250A and others
- Explicit Performance Improvements
  - SIMD extensions, AES extensions, …
  - …

SIMD operations

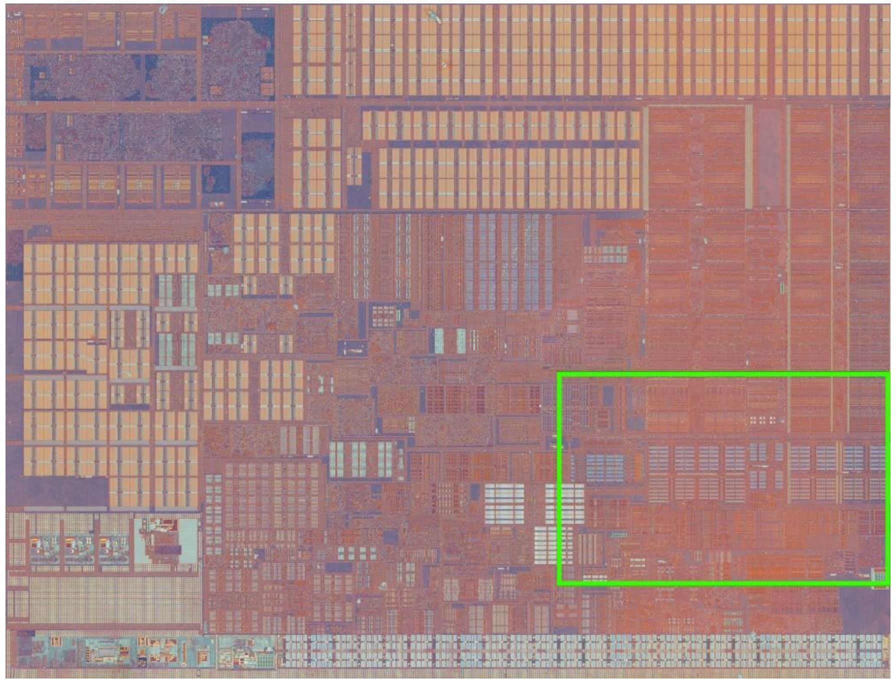
- ❑ Single ISA instruction performs same computation on multiple data
- ❑ Typically implemented with special, wider registers
- ❑ Example operation:
  - Load 32 bytes from memory to special register X
  - Load 32 bytes from memory to special register Y
  - Perform addition between each 4-byte value in X and each 4 byte value in Y
  - Store the four results in special register Z
  - Store Z to memory

For i in (0 to 7): Z[i] = X[i] + Y[i];

- ❑ RISC-V SIMD extensions (P) is still being worked on (as of 2021)

Example: Intel SIMD Extensions

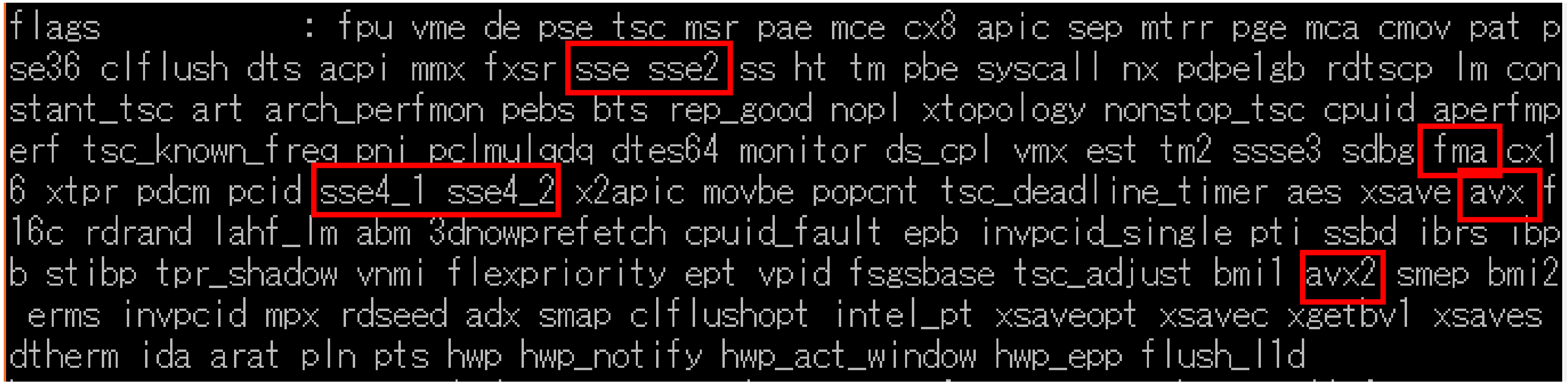
- More transistors (Moore's law) but no faster clock, no more ILP…
  - More capabilities per processor has to be explicit!
- New instructions, new registers
  - Must be used explicitly by programmer or compiler!
- Introduced in phases/groups of functionality
  - SSE – SSE4 (1999 –2006)
    - 128 bit width operations
  - AVX, FMA, AVX2, AVX-512 (2008 – 2019)
    - 256 – 512 bit width operations
  - F16C, and more to come?



Performance matters, "AVX-512 Mask Registers, Again." 2020

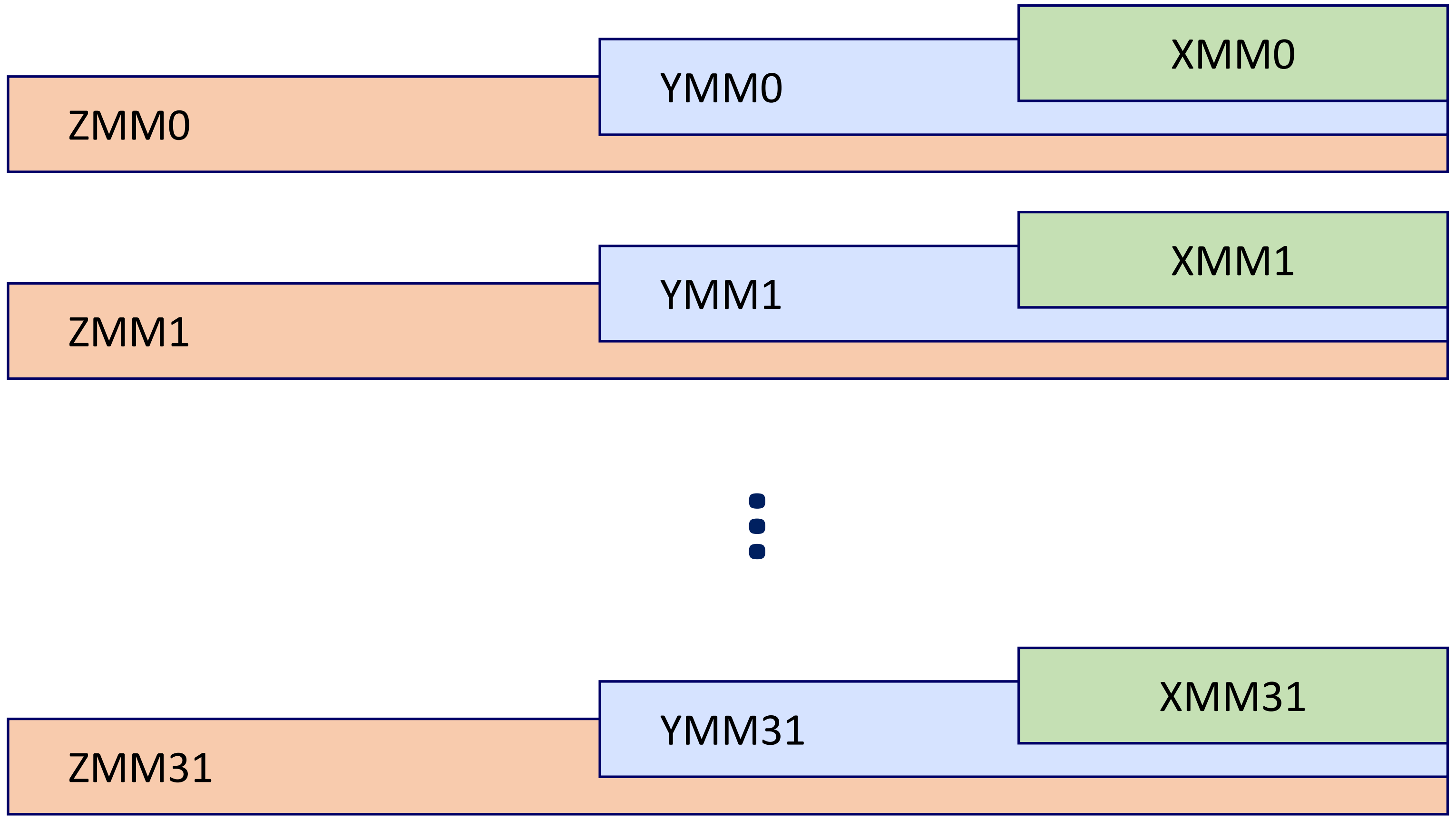
Aside: Do I Have SIMD Capabilities?

- less /proc/cpuinfo



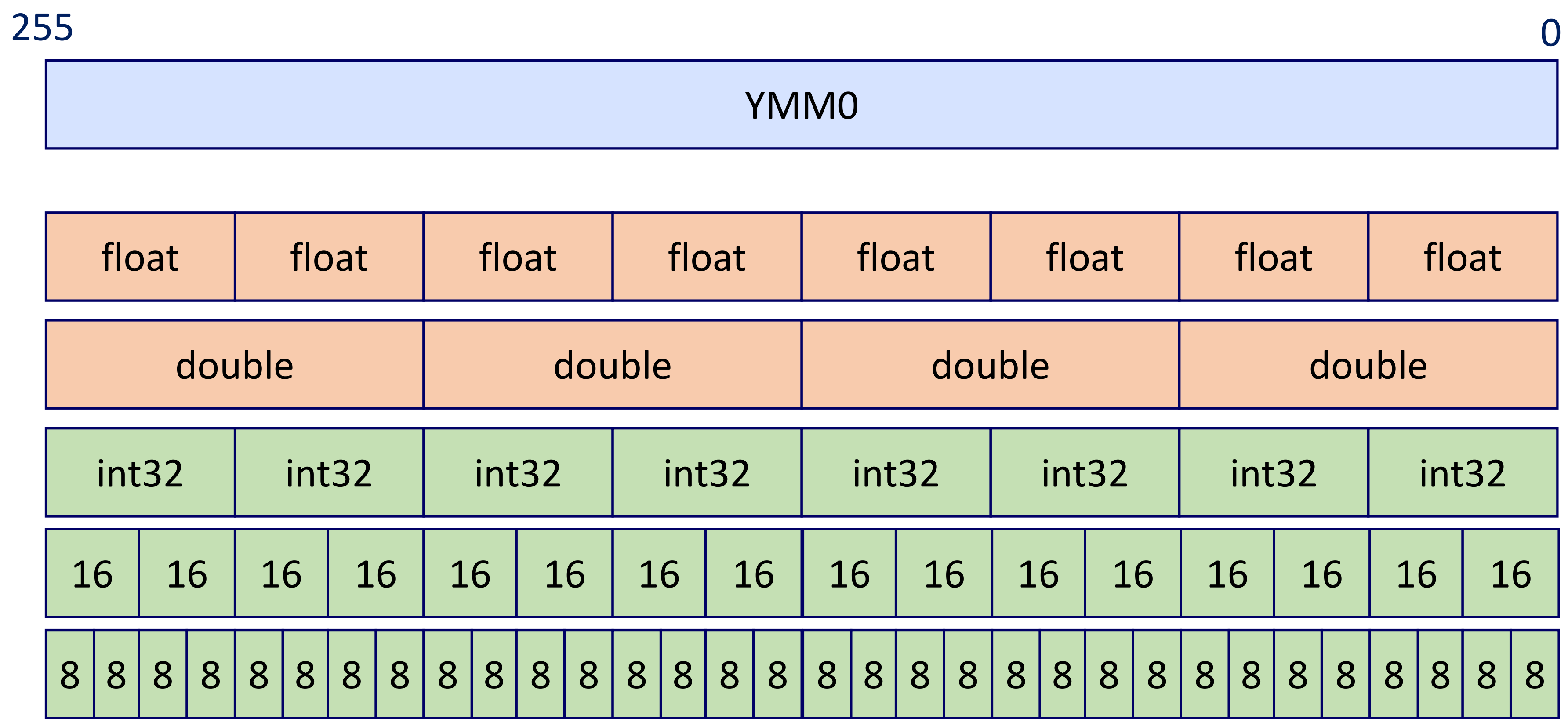
```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm con
stant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
erf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f
16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibp
b stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

Intel SIMD Registers (AVX-512)



- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512

SSE/AVX Data Types



Operation on
32 8-bit values
in one instruction!

Compiler Automatic Vectorization

- ❑ In gcc, flags “-O3 -mavx -mavx2” attempts automatic vectorization
- ❑ Works pretty well for simple loops

```c
int a[256], b[256], c[256];
void foo () {
    for (int i=0; i<256; i++) a[i] = b[i] * c[i];
}
```

```
.L2:
        vmovdqa xmm1, XMMWORD PTR b[rax]
        add     rax, 16
        vpmulld xmm0, xmm1, XMMWORD PTR c[rax-16]
        vmovaps XMMWORD PTR a[rax-16], xmm0
        cmp     rax, 1024
        jne     .L2
```

Generated using GCC explorer: https://gcc.godbolt.org/

- ❑ But not for anything complex
  - E.g., naïve bubblesort code not parallelized at all

Intel SIMD Intrinsics

- Use C functions instead of inline assembly to call AVX instructions
- Compiler manages registers, etc
- Intel Intrinsics Guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide
  - One of my most-visited pages…

e.g.,

```
__m256 a, b, c;
__m256 d = _mm256_fmadd_ps(a, b, c); // d[i] = a[i]*b[i]+c[i] for i = 0 ...7
```

Intrinsic Naming Convention

- **_mm<width>_[function]_[type]**
  - E.g., _mm256_fmadd_ps :
    perform fmadd (floating point multiply-add) on
    256 bits of
    packed single-precision floating point values (8 of them)

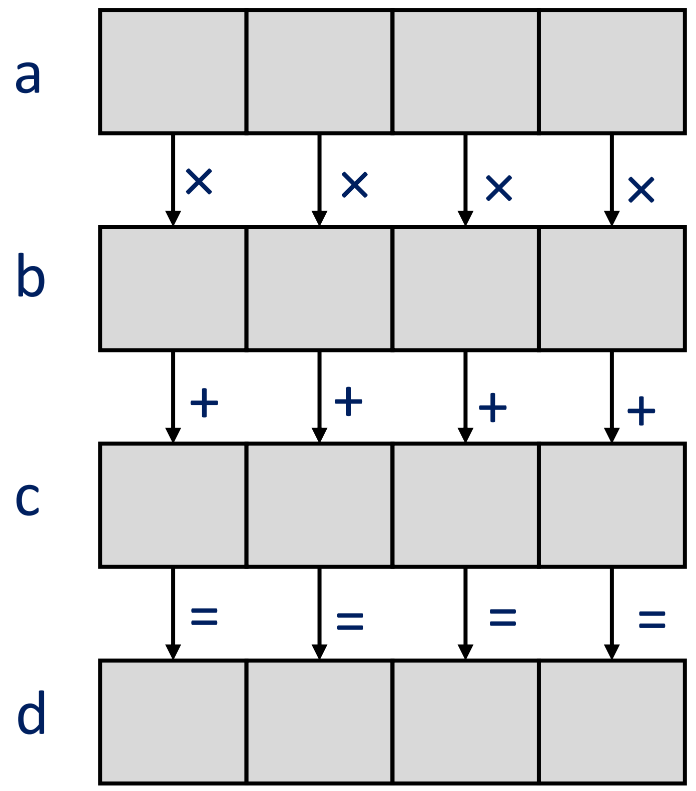
Width	Prefix
128	_mm_
256	_mm256_
512	_mm512_

Type	Postfix
Single precision	_ps
Double precision	_pd
Packed signed integer	_epiNNN (e.g., epi256)
Packed unsigned integer	_epuNNN (e.g., epu256)
Scalar integer	_siNNN (e.g., si256)

Not all permutations exist! Check guide

Example: Vertical Vector Instructions

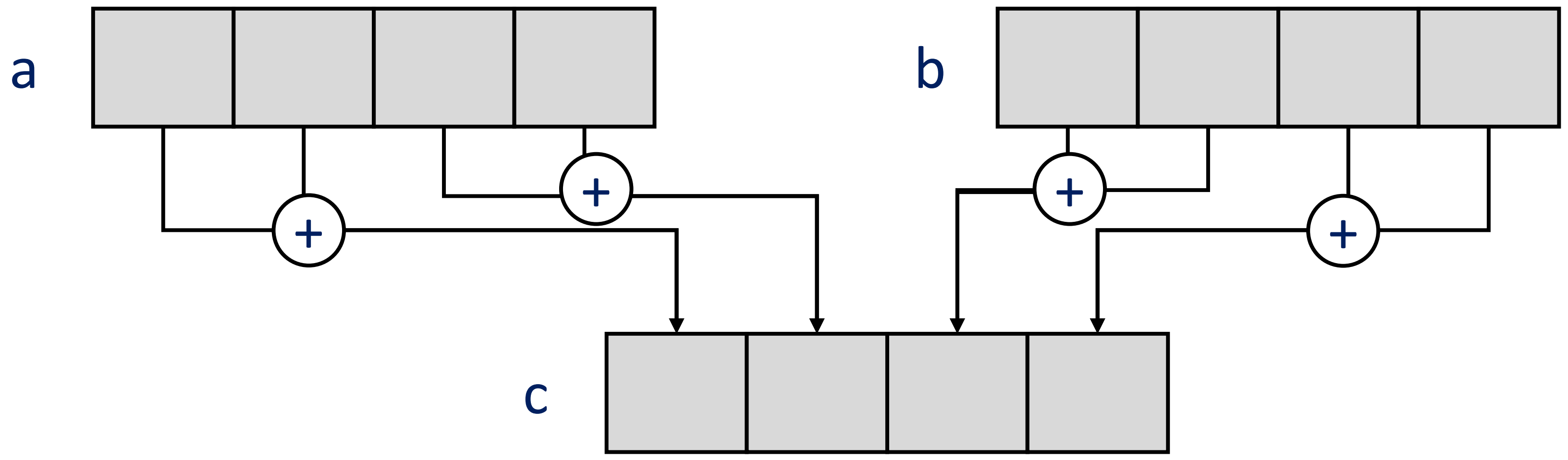
- Add/Subtract/Multiply
  - _mm256_add/sub/mul/div_ps/pd/epi
    - Mul only supported for epi32/epu32/ps/pd
    - Div only supported for ps/pd
    - Consult the guide!
- Max/Min/GreaterThan/Equals
- Sqrt, Reciprocal, Shift, etc…
- FMA (Fused Multiply-Add)
  - (a*b)+c, -(a*b)-c, -(a*b)+c, and other permutations!
  - Consult the guide!
- …



```
__m256 a, b, c;
__m256 d = _mm256_fmadd_pd(a, b, c);
```

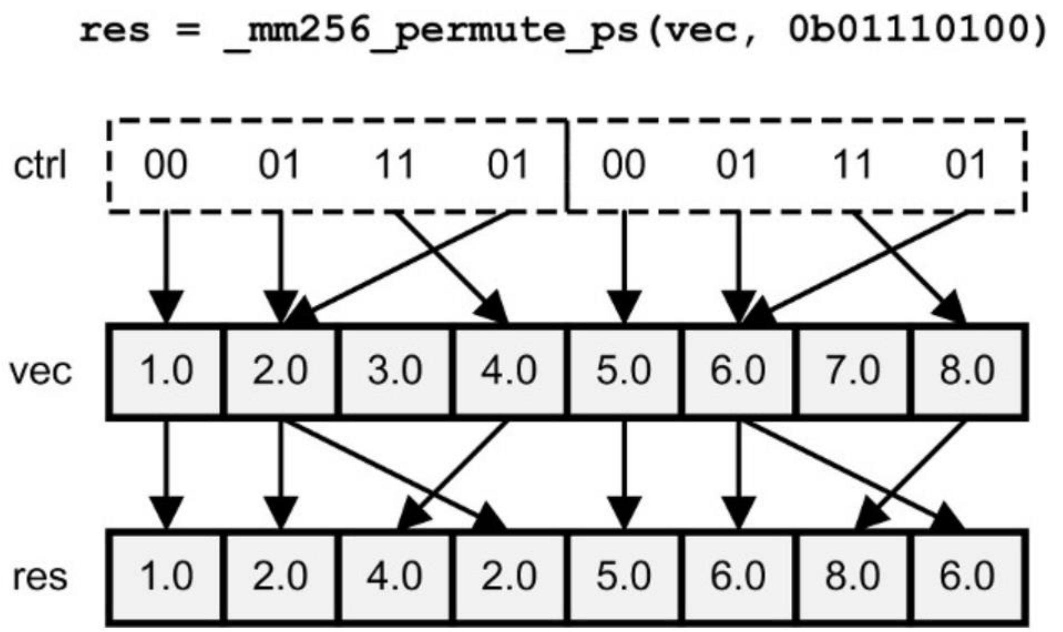
Horizontal Vector Instructions

- Horizontal add/subtraction
  - Adds adjacent pairs of values
  - E.g., __m256d _mm256_hadd_pd (__m256d a, __m256d b)



Shuffling/Permutation

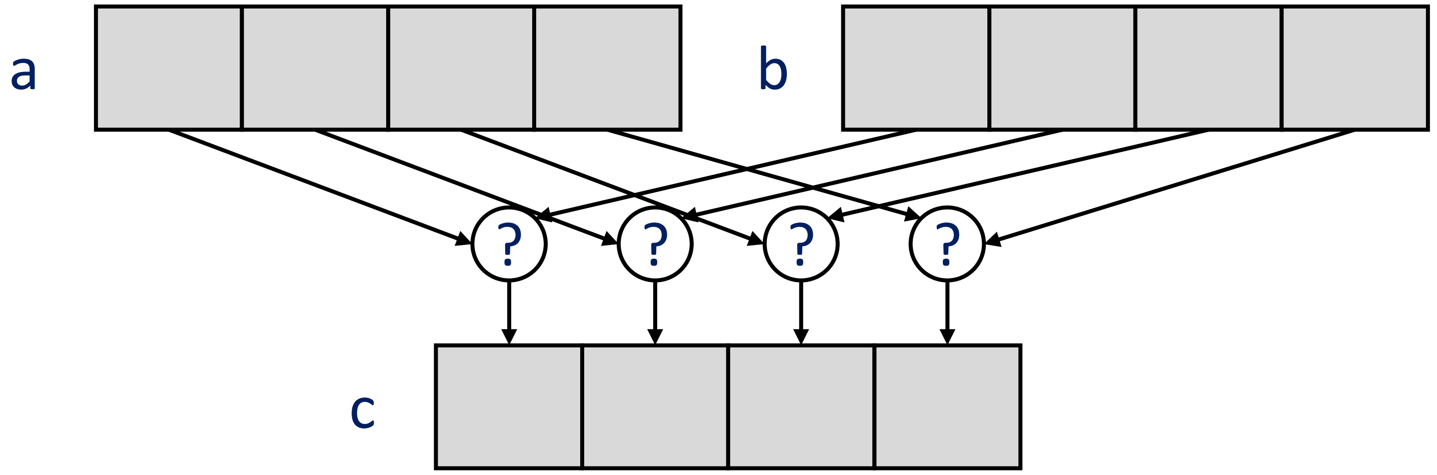
- Within 128-bit lanes
  - _mm256_shuffle_ps/pd/… (a,b, imm8)
  - _mm256_permute_ps/pd
  - _mm256_permutevar_ps/…
- Across 128-bit lanes
  - _mm256_permute2x128/4x64 : Uses 8 bit control
  - _mm256_permutevar8x32/… : Uses 256 bit control
- Not all type permutations exist for each type, but variables can be cast back and forth between types



Matt Scarpino, “Crunching Numbers with AVX and AVX2,” 2016

Blend

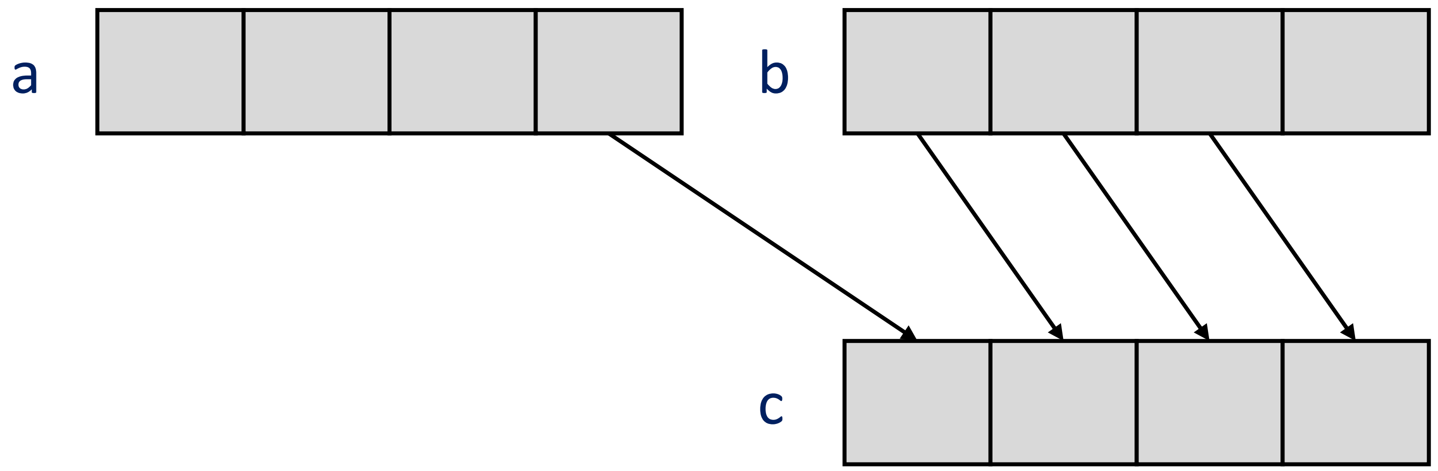
- Merges two vectors using a control
  - _mm256_blend_... : Uses 8 bit control
    - e.g., _mm256_blend_epi32
  - _mm256_blendv_... : Uses 256 bit control
    - e.g., _mm256_blendv_epi8



Alignr

- Right-shifts concatenated value of two registers, by byte
  - Often used to implement circular shift by using two same register inputs
  - _mm256_alignr_epi8 (a, b, count)

Example of 64-bit values being shifted by 8

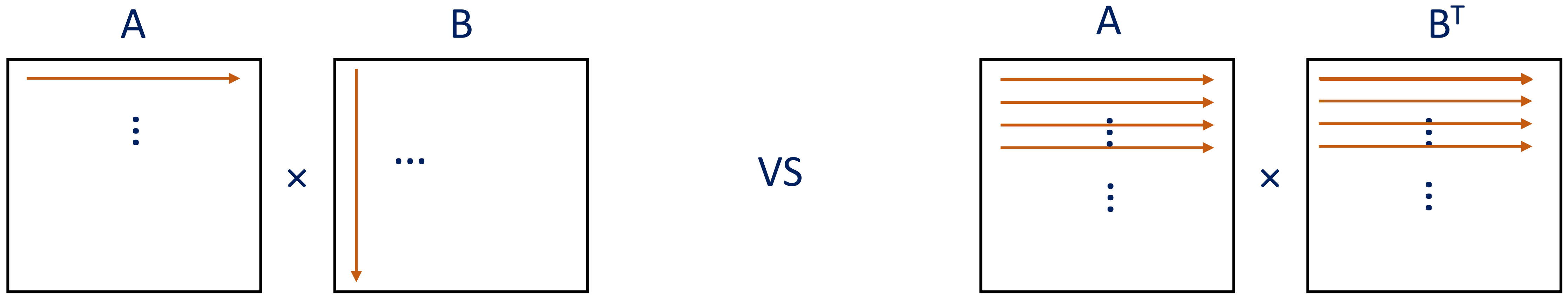


Helper Instructions

- Cast
  - __mm256i <-> __mm256, etc…
  - Syntactic sugar -- does not spend cycles
- Convert
  - 4 floats <-> 4 doubles, etc…
- Movemask
  - __mm256 mask to -> int imm8
- And many more…

Case Study: Matrix Multiplication

- Remember simply transposing matrix B brought 6x performance
  - At that point, we are bottlenecked by single-thread processing performance
  - Adding SIMD gets us more!
  - After this we are again bottlenecked by memory, but that is for another time



63.19 seconds

10.39 seconds
(6x performance)

2.20 seconds
(29x performance!)

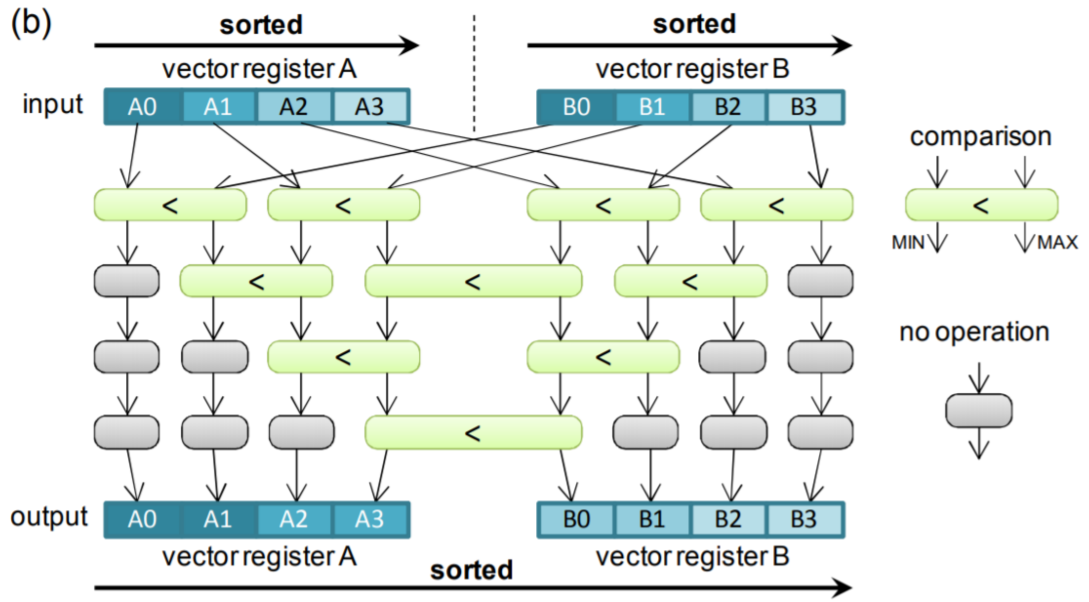
Case Study: Sorting

- Important, fundamental application!
- Can be parallelized via divide-and-conquer
- How can SIMD help?

The Two Register Merge

- Sort units of two pre-sorted registers, K elements
  - minv = A, maxv = B

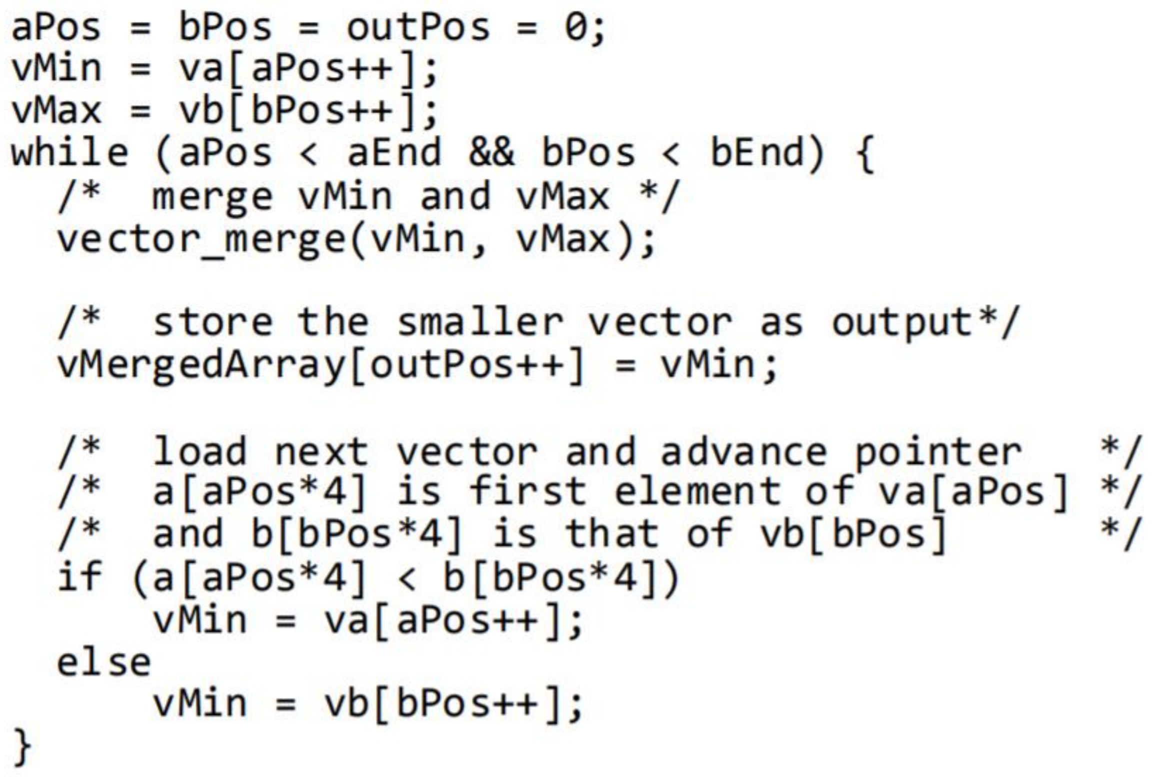
  - // Repeat K times
    - minv = min(minv,maxv)
    - maxv = max(minv,maxv)
    - // circular shift one value down
    - minv = alignr(minv, minv, sizeof(int))



Inoue et.al., “SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures,” VLDB 2015

SIMD And Merge Sort

- ❑ Hierarchically merged sorted subsections
- ❑ Using the SIMD merger for sorting
  - o vector_merge is the two-register sorter from before



```
aPos = bPos = outPos = 0;
vMin = va[aPos++];
vMax = vb[bPos++];
while (aPos < aEnd && bPos < bEnd) {
  /*  merge vMin and vMax */
  vector_merge(vMin, vMax);

  /*  store the smaller vector as output*/
  vMergedArray[outPos++] = vMin;

  /*  load next vector and advance pointer   */
  /*  a[aPos*4] is first element of va[aPos] */
  /*  and b[bPos*4] is that of vb[bPos]      */
  if (a[aPos*4] < b[bPos*4])
      vMin = va[aPos++];
  else
      vMin = vb[bPos++];
}
```

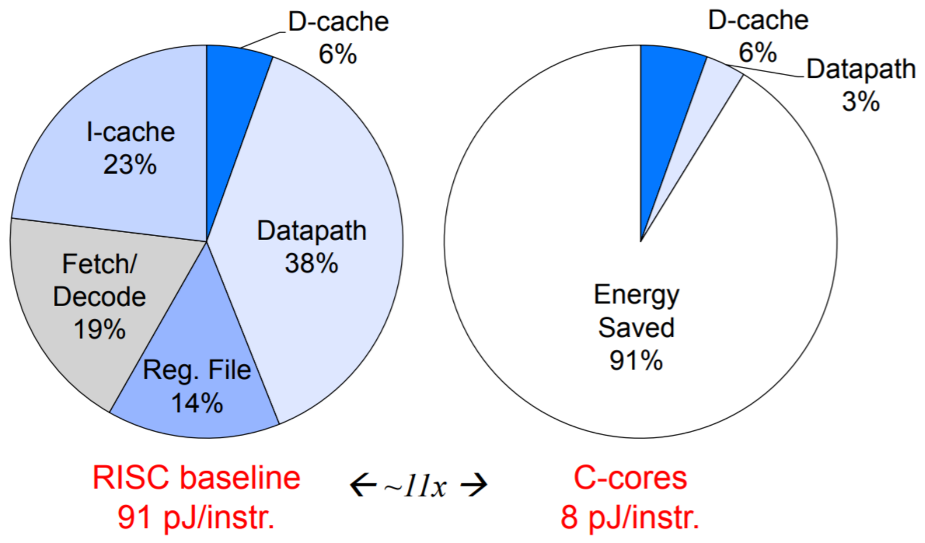
Inoue et.al., “SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures,” VLDB 2015

Topic Under Active Research!

- Papers being written about…
  - Architecture-optimized matrix transposition
  - Register-level sorting algorithm
  - Merge-sort
  - … and more!
- Good find can accelerate your application kernel Nx

Processor Microarchitectural Effects on Power Efficiency

- The majority of power consumption of a CPU is not from the ALU
  - Cache management, data movement, decoding, and other infrastructure
  - Adding a few more ALUs should not impact power consumption
- Indeed, 4X performance via AVX does not add 4X power consumption
  - From i7 4770K measurements:
  - Idle: 40 W
  - Under load : 117 W
  - Under AVX load : 128 W



Michael Taylor, "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse," 2012

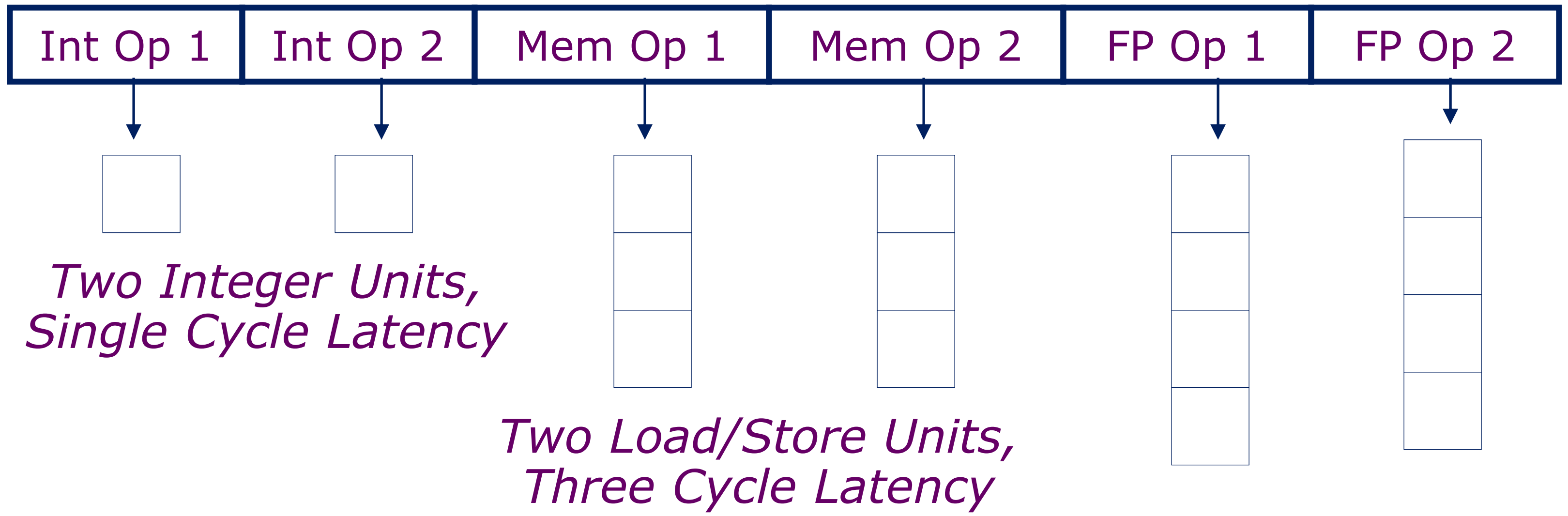
Very Long Instruction Word (VLIW)

- Superscalar does not change the ISA
  - Complicates hardware in charge of detecting dependencies!
- What if we changed the ISA, and made the compiler manage ILP?

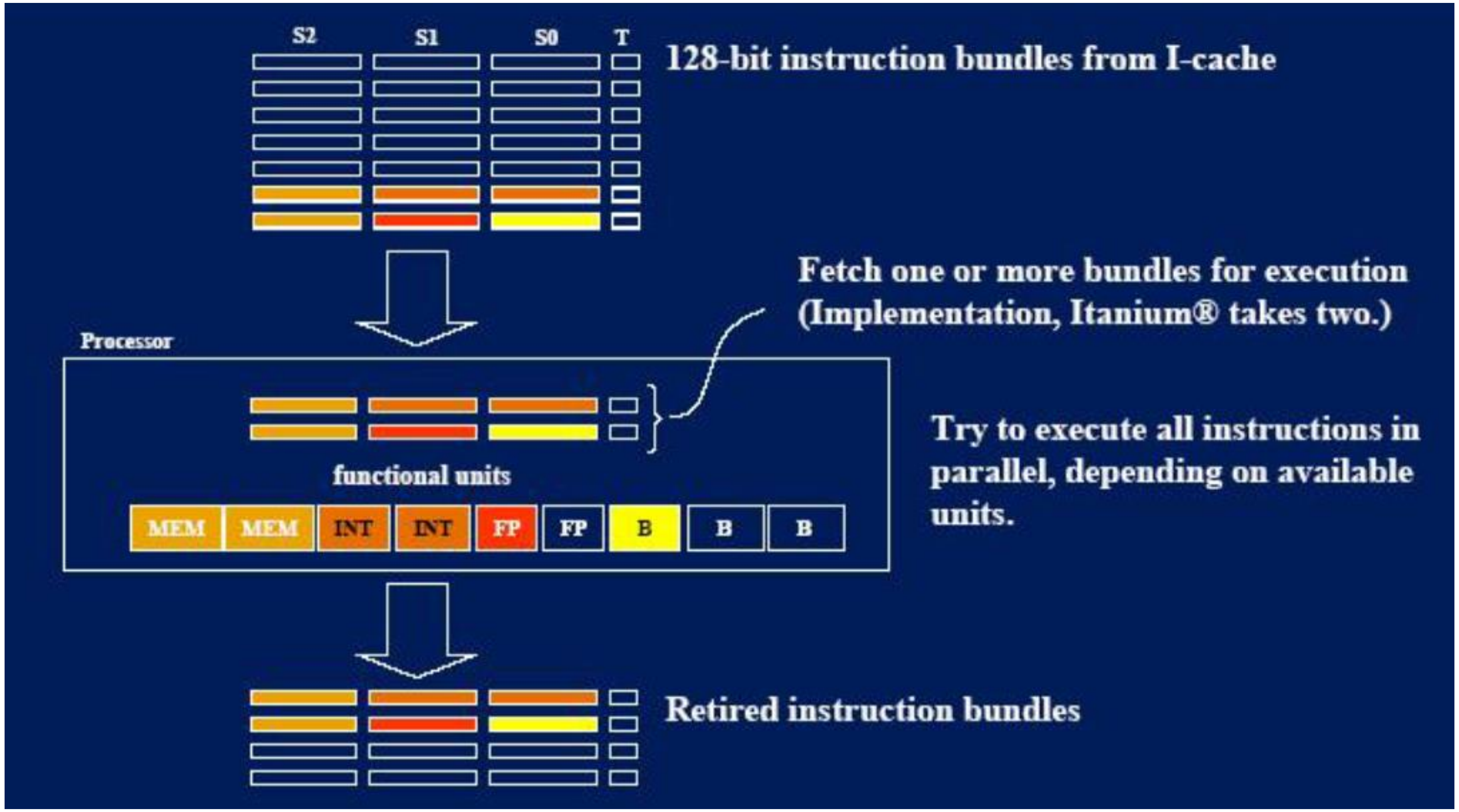
- Not in x86/RISC-V/ARM/…
  - Sometimes as accelerator extensions!
  - (RISC-V “V” extension)

Very Long Instruction Word (VLIW)

- Multiple instructions packaged into a Very Long Instruction
  - Sometimes "bundle"
- Each execution operation slot has a fixed function (ALU, Mem, FP, etc)
- Compiler's responsibility to create efficient instructions
  - Inter-slot dependency is not checked by hardware!

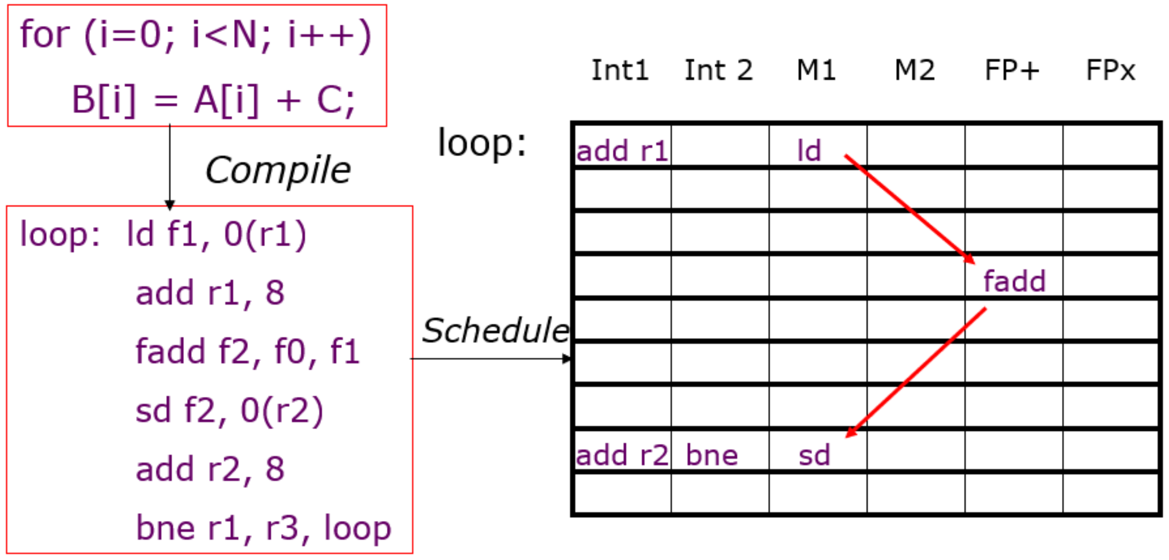

Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency

Intel Explicitly Parallel Instruction Computing (EPIC, Itanium)


S2
S1
S0
T
128-bit instruction bundles from I-cache
Fetch one or more bundles for execution (Implementation, Itanium® takes two.)
Processor
functional units
MEM
MEM
INT
INT
FP
FP
B
B
B
Try to execute all instructions in parallel, depending on available units.
Retired instruction bundles

VLIW Characteristics

- ❑ Very good performance for computation-intensive code
- ❑ Very bad performance for code with many dependencies/hazards!
  - Much more sensitive to hazards than single-issue pipelines
  - Example: short loops



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

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Compiler's job is important!

- e.g., Loop unrolling to keep execution units busy

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

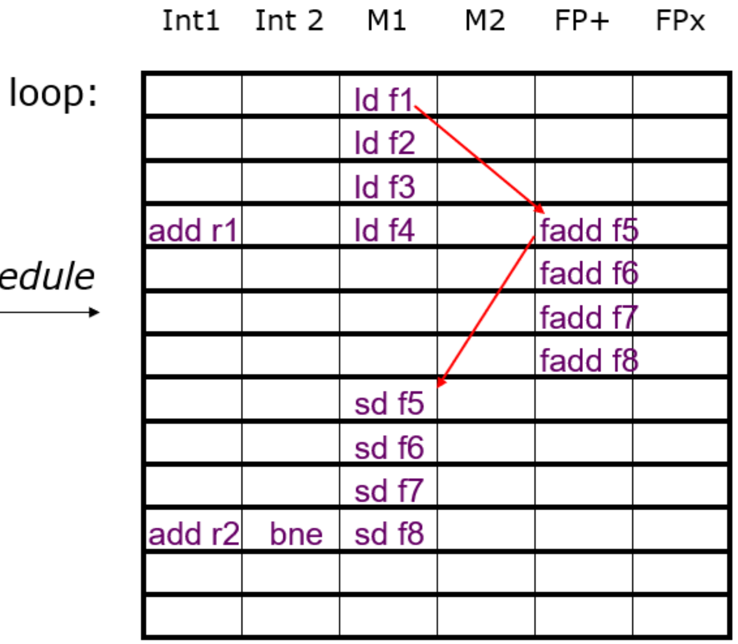
Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

Schedule



loop:

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

CS152-Spring'09

Issues with VLIW

- Execution unit configurations change across models
  - How many Integer units, how many float units, neural units …?
  - Cannot be binary compatible across models!
    - Unless hardware provides an abstraction layer…?
    - But that would add scheduler overhead, undermining VLIW (Itanium tried a good balance)
- Dependency/hazards difficult for compiler to manage
  - Too many slots end up empty (low performance, large binary)

- But when it works well, it works remarkably well
  - e.g., Scientific computing
  - That’s why it is often resurrected as potential solution (Itanium, ATi TeraScale, …)

CS250P: Computer Systems Architecture
Out of Order Processing

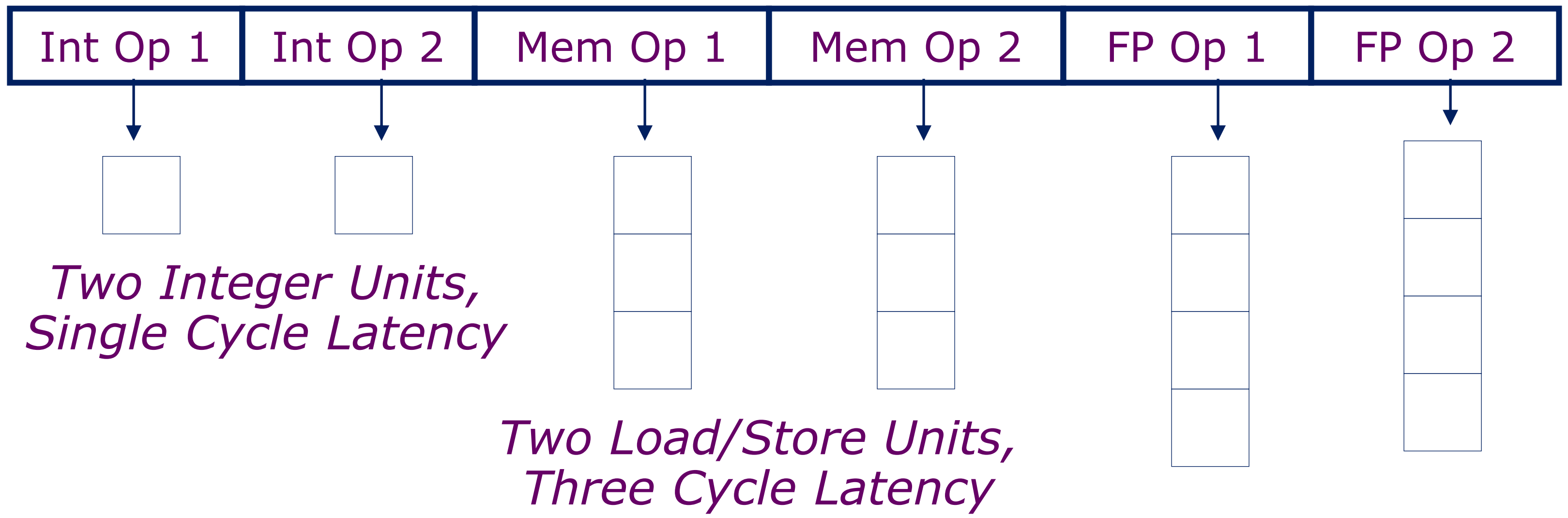
Sang-Woo Jun

Fall 2022

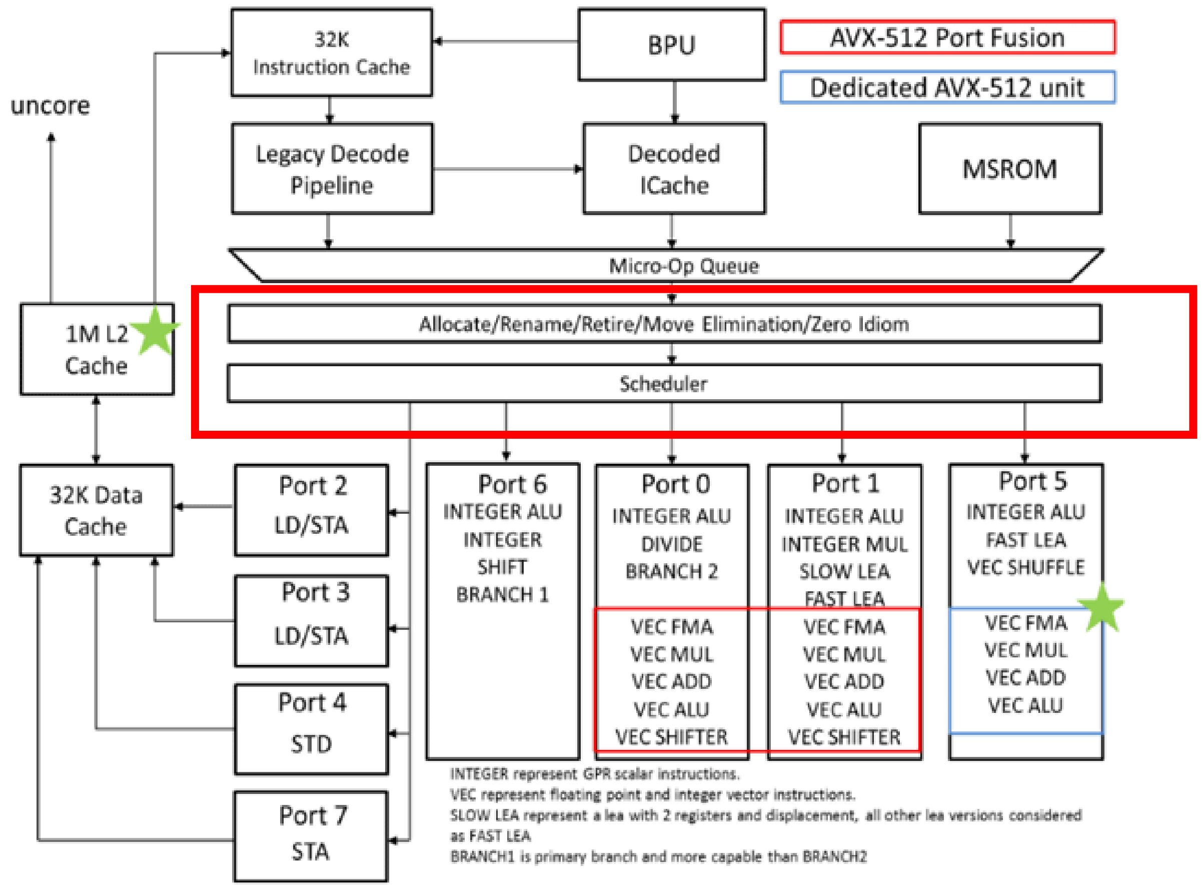
Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Back to Transparent Parallelism

- Explicit parallelism is not as popular as transparent
  - Everyone wants performance for free!
- Can we keep execution slots busy, using backwards-compatible single-thread instruction streams?


Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency

Skylake-X Microarchitecture (2019)



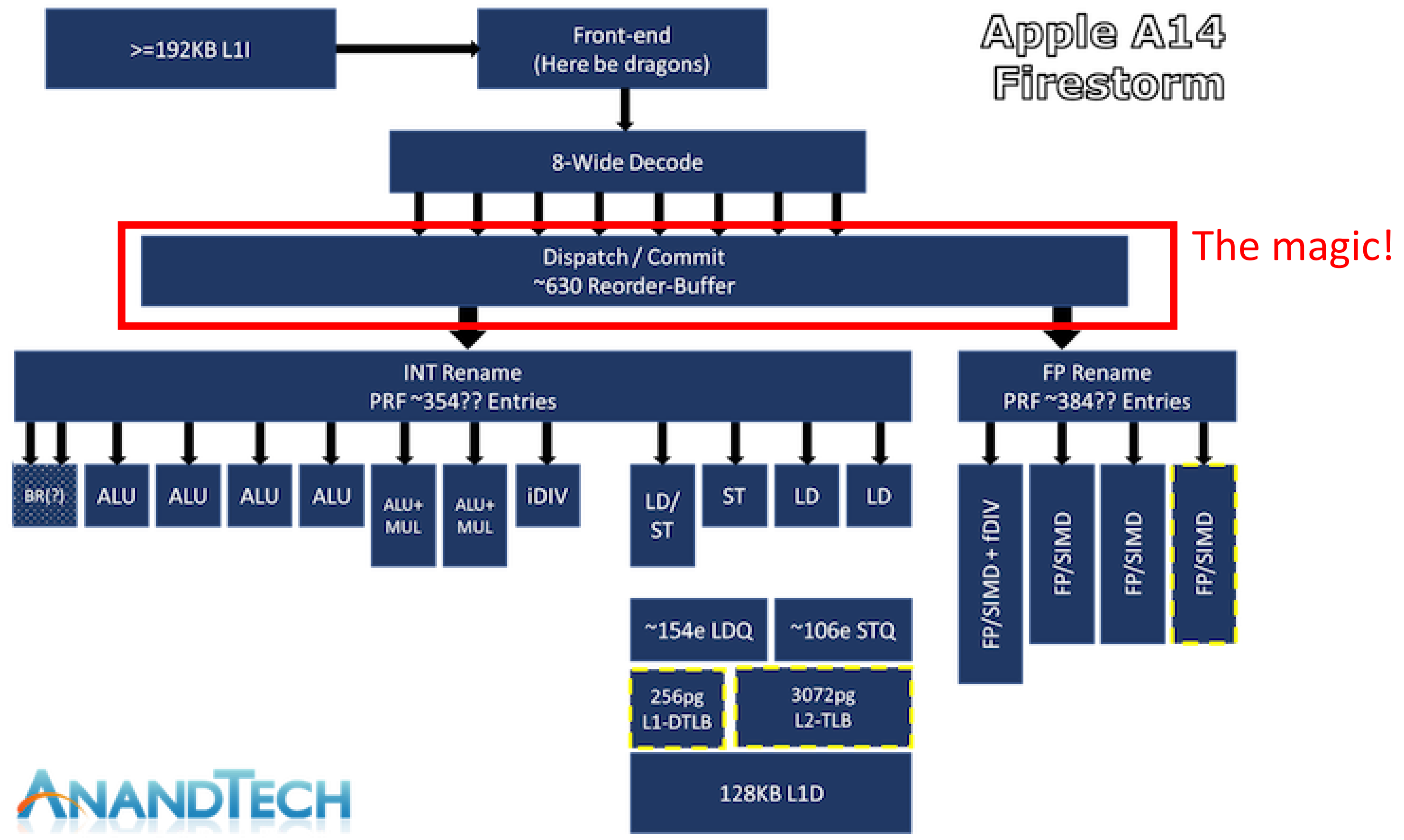
32K Instruction Cache
BPU
AVX-512 Port Fusion
Dedicated AVX-512 unit
uncore
Legacy Decode Pipeline
Decoded ICache
MSROM
Micro-Op Queue
1M L2 Cache
Allocate/Rename/Retire/Move Elimination/Zero Idiom
Scheduler
32K Data Cache
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 6 INTEGER ALU INTEGER SHIFT BRANCH 1
Port 0 INTEGER ALU DIVIDE BRANCH 2 VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 1 INTEGER ALU INTEGER MUL SLOW LEA FAST LEA VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 5 INTEGER ALU FAST LEA VEC SHUFFLE VEC FMA VEC MUL VEC ADD VEC ALU

INTEGER represent GPR scalar instructions.
VEC represent floating point and integer vector instructions.
SLOW LEA represent a lea with 2 registers and displacement, all other lea versions considered as FAST LEA
BRANCH1 is primary branch and more capable than BRANCH2

The magic!

Anandtech

Apple M1 Microarchitecture (2020)

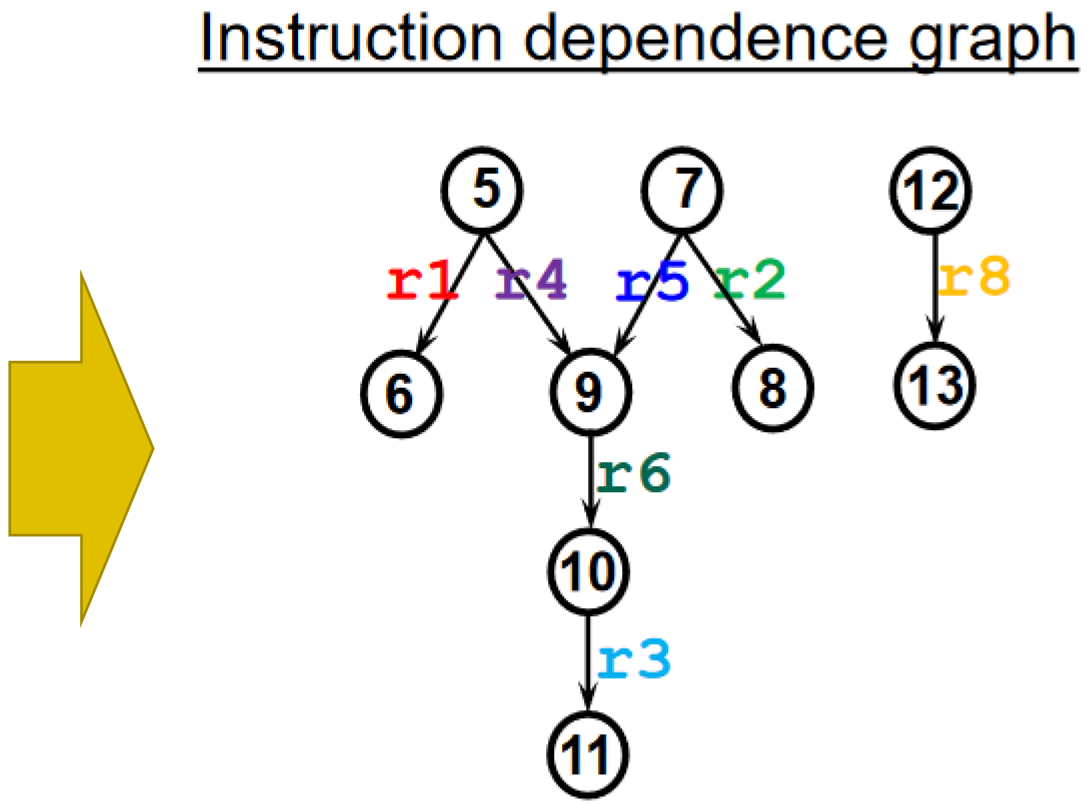


Anandtech

OoO: Determining dependencies

```
   (1)  r8 ← 20
   (2)  r1 ← addr1
   (3)  r2 ← addr2
   (4)  r3 ← addr3

LOOP:
   (5)  r4 ← MEM[r1]
   (6)  r1 ← r1 + 4
   (7)  r5 ← MEM[r2]
   (8)  r2 ← r2 + 4
   (9)  r6 ← r4 + r5
   (10) MEM[r3] ← r6
   (11) r3 ← r3 + 4
   (12) r8 ← r8 - 1
   (13) bnz r8, LOOP
```

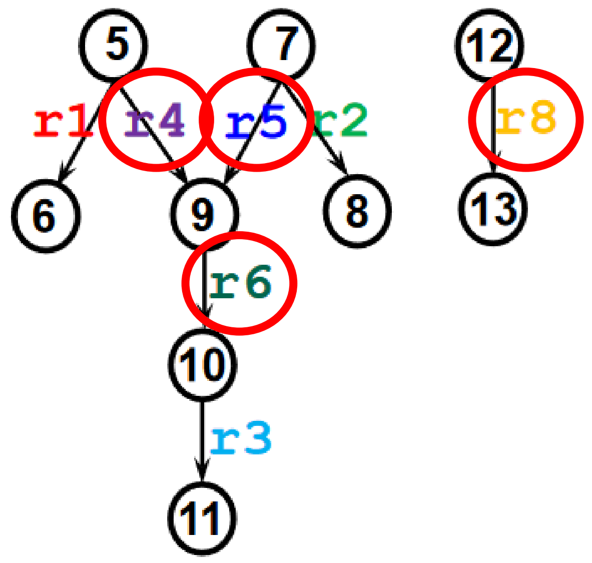


Data dependency types: RaW

- Read-after-Write
  - A “true” dependency
  - We must wait until r5’s value is materialized… No other choice

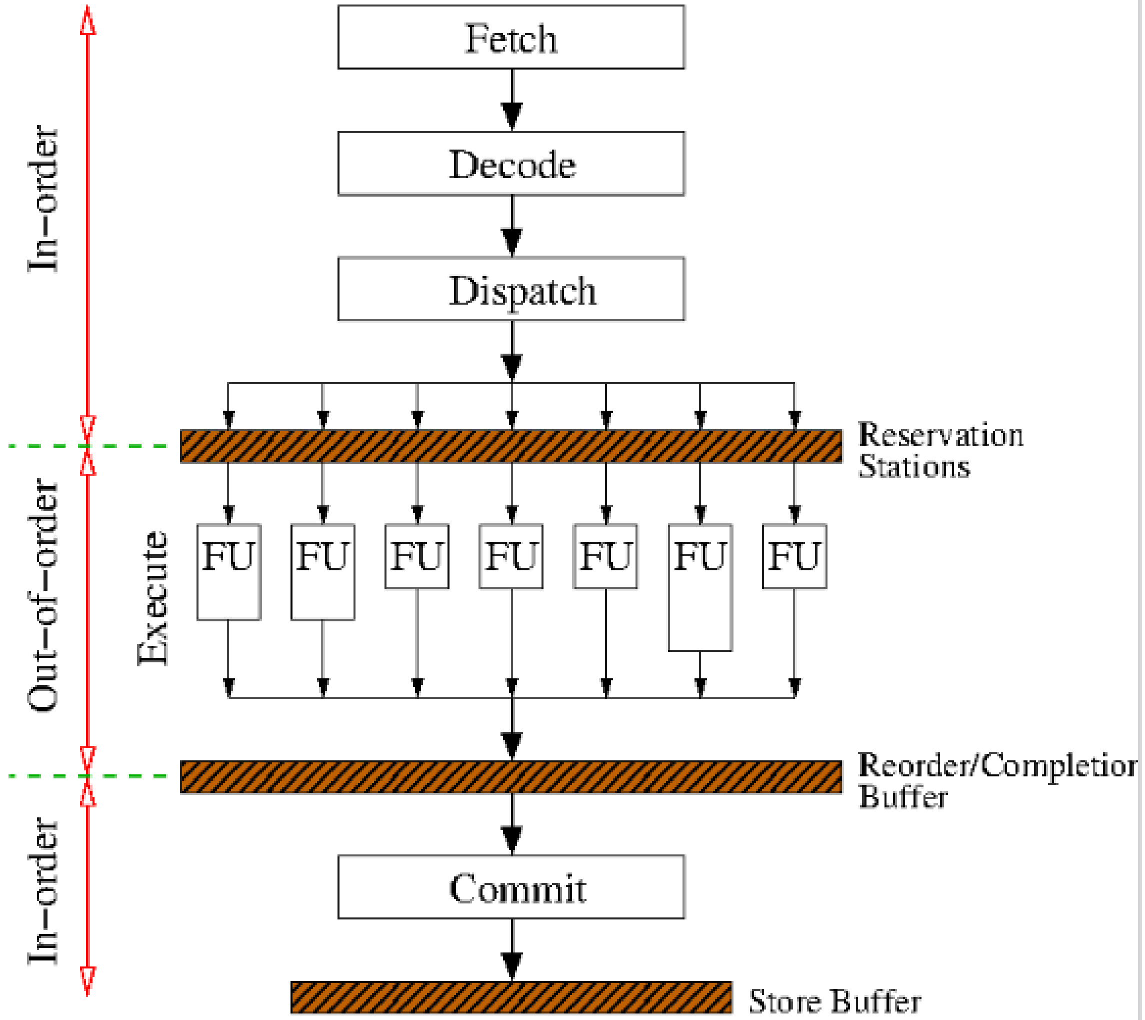
```
(7)   r5 ← MEM[r2]
(9)   r6 ← r4 + r5
```

```
LOOP:
    (5)   r4 ← MEM[r1]
    (6)   r1 ← r1 + 4
    (7)   r5 ← MEM[r2]
    (8)   r2 ← r2 + 4
    (9)   r6 ← r4 + r5
    (10)  MEM[r3] ← r6
    (11)  r3 ← r3 + 4
    (12)  r8 ← r8 - 1
    (13)  bnz r8, LOOP
```



OoO managing dependencies

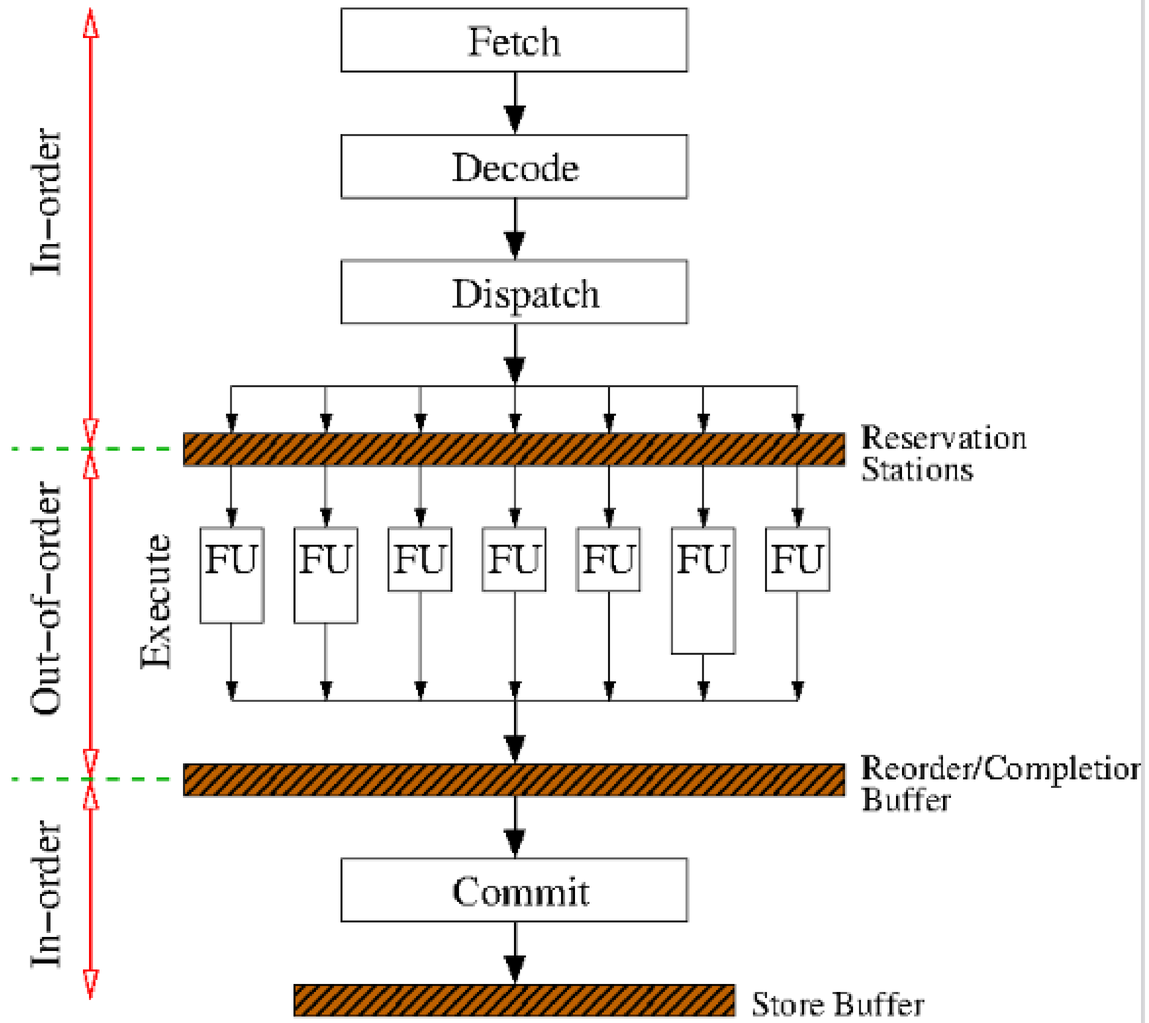
- Looks like dispatch+Commit stages added to VLIW
  - Instructions wait at “reservation stations”
  - Listens to forwarding paths
    - “Is my input operand being written to”
  - Forwarded to FU when ready
    - Out of order



OoO managing dependencies

- ❑ Arithmetic can happen OoO, BUT Commits should happen In-Order!
  - Register writes, memory updates, etc

- ❑ Decoded instructions line up at Reorder Buffer(RoB)
  - Wait until execute results available
  - Wait until branch mispredict ruled out
  - Commits in order of insertion



Many topics we won’t go into today!

- Effectively matching available operands to waiting instructions
  - Looping over instructions is too slow
  - N-to-N broadcast is too expensive (slow clocks!)
  - Tomasulo’s algorithm!

- Precise interrupts become complicated
  - Things are executing OoO, when a breakpoint happens, how do we line things back up for debugging?

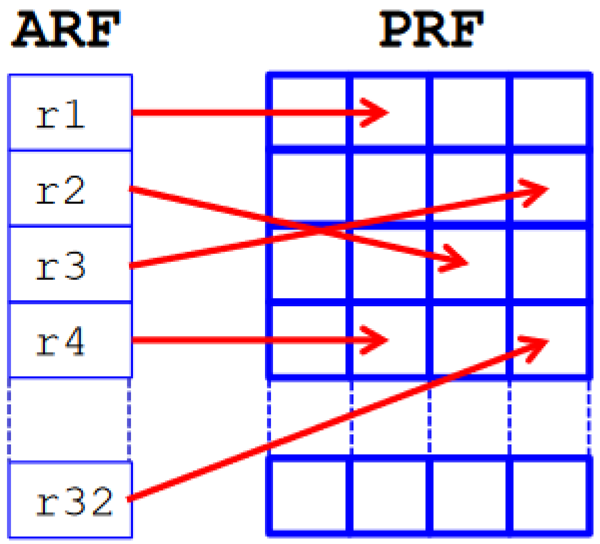
Just one more topic: Register renaming

- Not all dependencies are RaW. Some can be resolved!
  - Write-after-Read (WaR)
    - e.g., 5->6
    - “Anti-dependence”: r1’s value clobbered after (6)
    - If we had used “r9” instead, no dependency!
  - Write-after-Write (WaW)
    - e.g., 7->7 across loop iteration
    - (7) does not read from “r5”…
    - If each loop iteration used a different reg, (r9, r10, r11,…) no dependency!

```
(5)   r4 ← MEM[r1]
(6)   r1 ← r1 + 4
(7)   r5 ← MEM[r2]
(8)   r2 ← r2 + 4
(9)   r6 ← r4 + r5
(10)  MEM[r3] ← r6
(11)  r3 ← r3 + 4
(12)  r8 ← r8 - 1
(13)  bnz r8, LOOP
```

OoO: Register renaming

- Two different concepts of registers
  - "Architectural Registers": Conceptually defined in ISA, software abstraction
    - "RISC-V has 32 registers in the register file"
  - "Physical Registers": Larger number of registers actually in silicon
    - Scheduler dynamically renames registers to an empty slot in the physical register file
    - When WaW or WaR dependencies are discovered



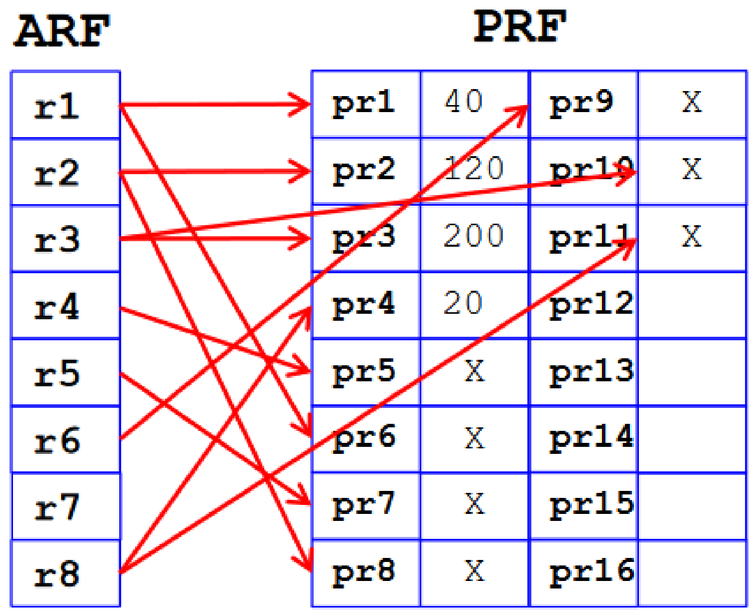
Register renaming: Previous example…

Original code:

```
(5)  r4 ← MEM[r1]
(6)  r1 ← r1 + 4
(7)  r5 ← MEM[r2]
(8)  r2 ← r2 + 4
(9)  r6 ← r4 + r5
(10) MEM[r3] ← r6
(11) r3 ← r3 + 4
(12) r8 ← r8 - 1
(13) bnz r8, LOOP
```

Post-decode(as seen by RS):

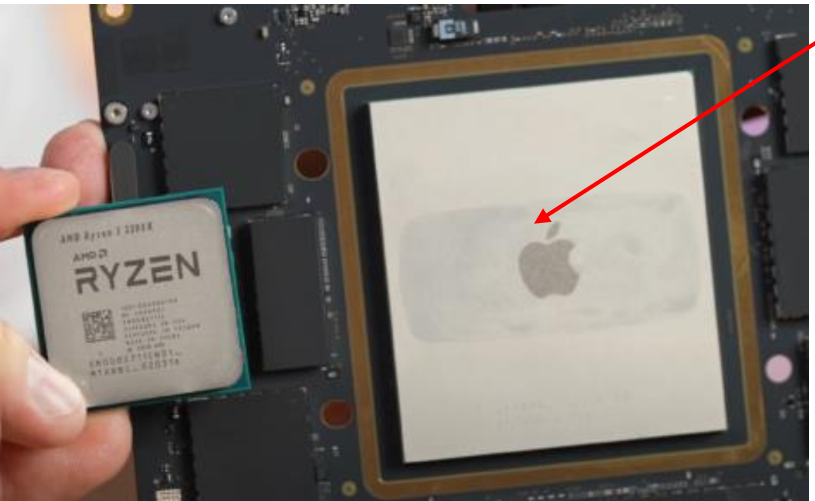
```
(5)  pr5  ← MEM[40]
(6)  pr6  ← 40 + 4
(7)  pr7  ← MEM[120]
(8)  pr8  ← 120 + 4
(9)  pr9  ← pr5 + pr7
(10) MEM[200] ← pr9
(11) pr10 ← 200 + 4
(12) pr11 ← 20 - 1
(13) bnz  pr11, LOOP
```



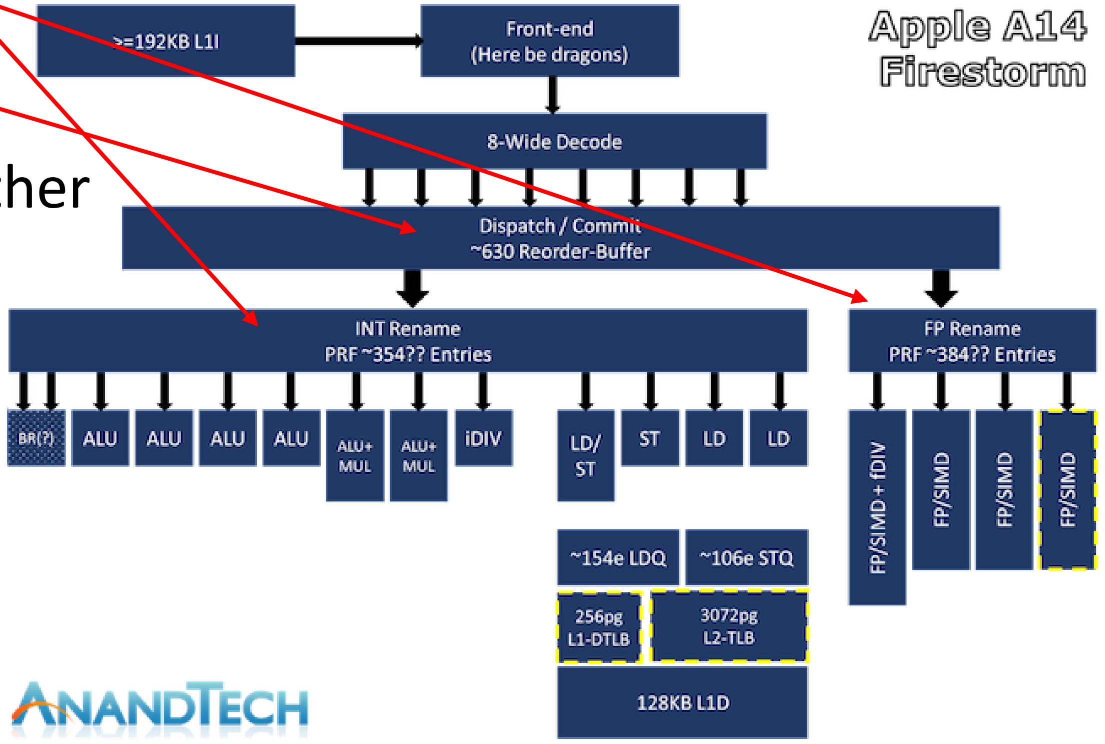
pr1	40	pr9	X
pr2	120	pr10	X
pr3	200	pr11	X
pr4	20	pr12	
pr5	X	pr13	
pr6	X	pr14	
pr7	X	pr15	
pr8	X	pr16	

Back to timely example: Apple M1

- Really good single-thread performance!
- How?
  - RISC!
  - "8-wide decoder" […] "16 execution units (per core)"
  - "(Estimated) 630-deep out-of-order"
  - "Unified memory architecture"
  - Hardware/software optimized for each other



M1 Ultra
Image source: wccftech



ANANDTECH