Lect. 11: Vector and SIMD Processors

- Many real-world problems, especially in science and engineering, map well to computation on arrays
- RISC approach is inefficient:
  - Based on loops → require dynamic or static unrolling to overlap computations
  - Indexing arrays based on arithmetic updates of induction variables
  - Fetching of array elements from memory based on individual, and unrelated, loads and stores
  - Instruction dependences must be identified for each individual instruction
- Idea:
  - Treat operands as whole vectors, not as individual integer of float-point numbers
  - Single machine instruction now operates on whole vectors (e.g., a vector add)
  - Loads and stores to memory also operate on whole vectors
  - Individual operations on vector elements are independent and only dependences between whole vector operations must be tracked

Execution Model

```
for (i=0; i<64; i++)
      a[i] = b[i] + s;
```

- Straightforward RISC code:
  - F2 contains the value of s
  - R1 contains the address of the first element of a
  - R2 contains the address of the first element of b
  - R3 contains the address of the last element of a + 8

```
loop: L.D    F0,0(R2)   ;F0=array element of b
      ADD.D  F4,F0,F2   ;main computation
      S.D    F4,0(R1)   ;store result
      DADDUI R1,R1,8    ;increment index
      DADDUI R2,R2,8    ;increment index
      BNE    R1,R3,loop ;next iteration
```

Execution Model

```
for (i=0; i<64; i++)
      a[i] = b[i] + s;
```

- Straightforward vector code:
  - `F2` contains the value of `s`
  - `R1` contains the address of the first element of `a`
  - `R2` contains the address of the first element of `b`
  - Assume vector registers have 64 double precision elements

```
LV      V1,R2       ;V1=array b
ADDVS.D V2,V1,F2    ;main computation
SV      V2,R1       ;store result
```

  - Notes:
    - In practice vector registers are not of the exact size of the arrays
    - Only 3 instructions executed compared to 6*64=384 executed in the RISC

Execution Model (Pipelined)

IF	I1	I2						
ID		I1						
EXE			I1.1	I1.2	I1.3	I1.4	I1.5	I1.6
MEM				I1.1	I1.2	I1.3	I1.4	I1.5
WB					I1.1	I1.2	I1.3	I1.4
cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

- With multiple vector units, I2 can execute together with I1
- In practice, the vector units takes several cycles to operate on each element, but is pipelined

Pros of Vector Processors

- Reduced pressure on instruction fetch
  - Fewer instructions are necessary to specify the same amount of work
- Reduced pressure on instruction issue
  - Reduced number of branches alleviates branch prediction
  - Much simpler hardware for checking dependences
- More streamlined memory accesses
  - Vector loads and stores specify a regular access pattern
  - High latency of initiating memory access is amortized

Cons of Vector Processors

- Still requires a traditional scalar unit (integer and FP) for the non-vector operations
- Difficult to maintain precise interrupts (can't rollback all the individual operations already completed)
- Compiler or programmer has to vectorize programs
- Not very efficient for small vector sizes
- Not suitable/efficient for many different classes of applications
- Requires a specialized, high-bandwidth, memory system
  - Usually built around heavily banked memory with data interleaving

Performance Issues

- Performance of a vector instruction depends on the length of the operand vectors
- Initiation rate
  - Rate at which individual operations can start in a functional unit
  - For fully pipelined units this is 1 operation per cycle
- Start-up time
  - Time it takes to produce the first element of the result
  - Depends on how deep the pipeline of the functional units are
  - Especially large for load/store unit

Advanced Features: Masking

- What if the operations involve only some elements of the array, depending on some run-time condition?

```
for (i=0; i<64; i++)
   if (b[i] != 0)
      a[i] = b[i] + s;
```

- Solution: masking
  - Add a new boolean vector register (the vector mask register)
  - The vector instruction then only operates on elements of the vectors whose corresponding bit in the mask register is 1
  - Add new vector instructions to set the mask register
    - E.g., `SNEVS.D V1,F0` sets to 1 the bits in the mask registers whose corresponding elements in V1 are not equal to the value in F0
    - `CVM` instruction sets all bits of the mask register to 1

Advanced Features: Masking

```
for (i=0; i<64; i++)
   if (b[i] != 0)
      a[i] = b[i] + s;
```

- Vector code:
  - F2 contains the value of s and F0 contains zero
  - R1 contains the address of the first element of a
  - R2 contains the address of the first element of b
  - Assume vector registers have 64 double precision elements

```
LV       V1,R2         ;V1=array b
SNEVS.D  V1,F0         ;mask bit is 1 if b !=0
ADDVS.D  V2,V1,F2      ;main computation
CVM
SV       V2,R1         ;store result
```

Advanced Features: Scatter-Gather

- How can we handle sparse matrices?

```
for (i=0; i<64; i++)
   a[K[i]] = b[K[i]] + s;
```

- Solution: scatter-gather
  - Use the contents of an auxiliary vector to select which elements of the main vector are to be used
  - This is done by pointing to the address in memory of the elements to be selected
  - Add new vector instruction to load memory values based on this auxiliary vector
    - E.g. `LVI V1,(R1+V2)` loads the elements of a user array from memory locations `R1+V2(i)`
    - Also `SVI` store counterpart

Advanced Features: Scatter-Gather

```
for (i=0; i<64; i++)
   a[K[i]] = b[K[i]] + s;
```

- Vector code:
  - F2 contains the value of s
  - R1 contains the address of the first element of a
  - R2 contains the address of the first element of b
  - V3 contains the indices of a and b that need to be used
  - Assume vector registers have 64 double precision elements

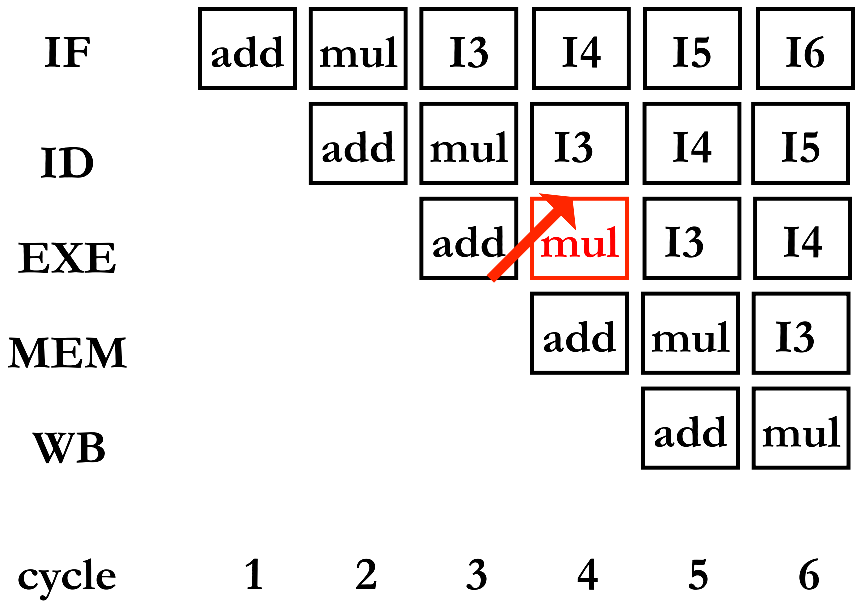
```
LVI      V1,(R2+V3) ;V1=array b indexed by V3
ADDVS.D V2,V1,F2    ;main computation
SVI      V2,(R1+V3) ;store result
```

Advanced Features: Chaining

- Forwarding in pipelined RISC processors allow dependent instructions to execute as soon as the result of the previous instruction is available

value

```
ADD.D R1,R2,R3 # R1=R2+R3
MUL.D R4,R5,R1 # R4=R5+R1
```

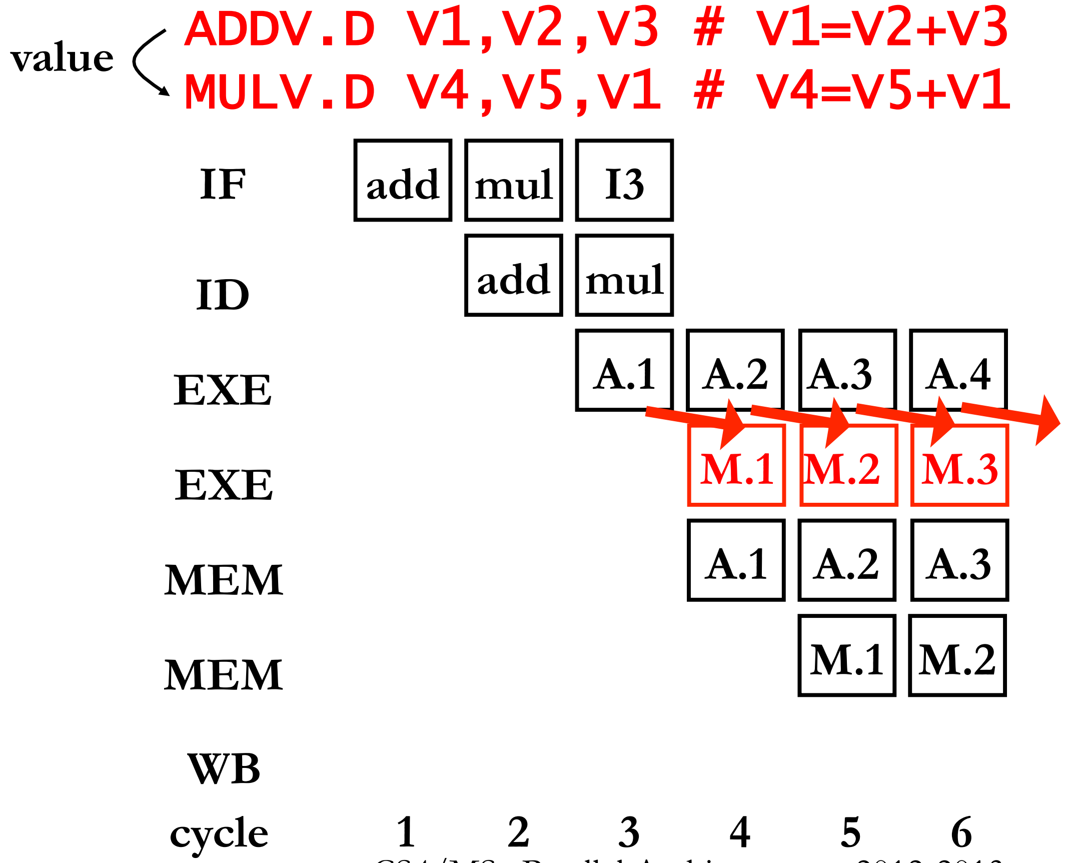


IF	add	mul	I3	I4	I5	I6
ID		add	mul	I3	I4	I5
EXE			add	mul	I3	I4
MEM				add	mul	I3
WB					add	mul
cycle	1	2	3	4	5	6

Advanced Features: Chaining

- Similar idea applies to vector instructions and is called chaining
  - Difference is that chaining of vector instructions requires multiple functional units as the same unit cannot be used back-to-back

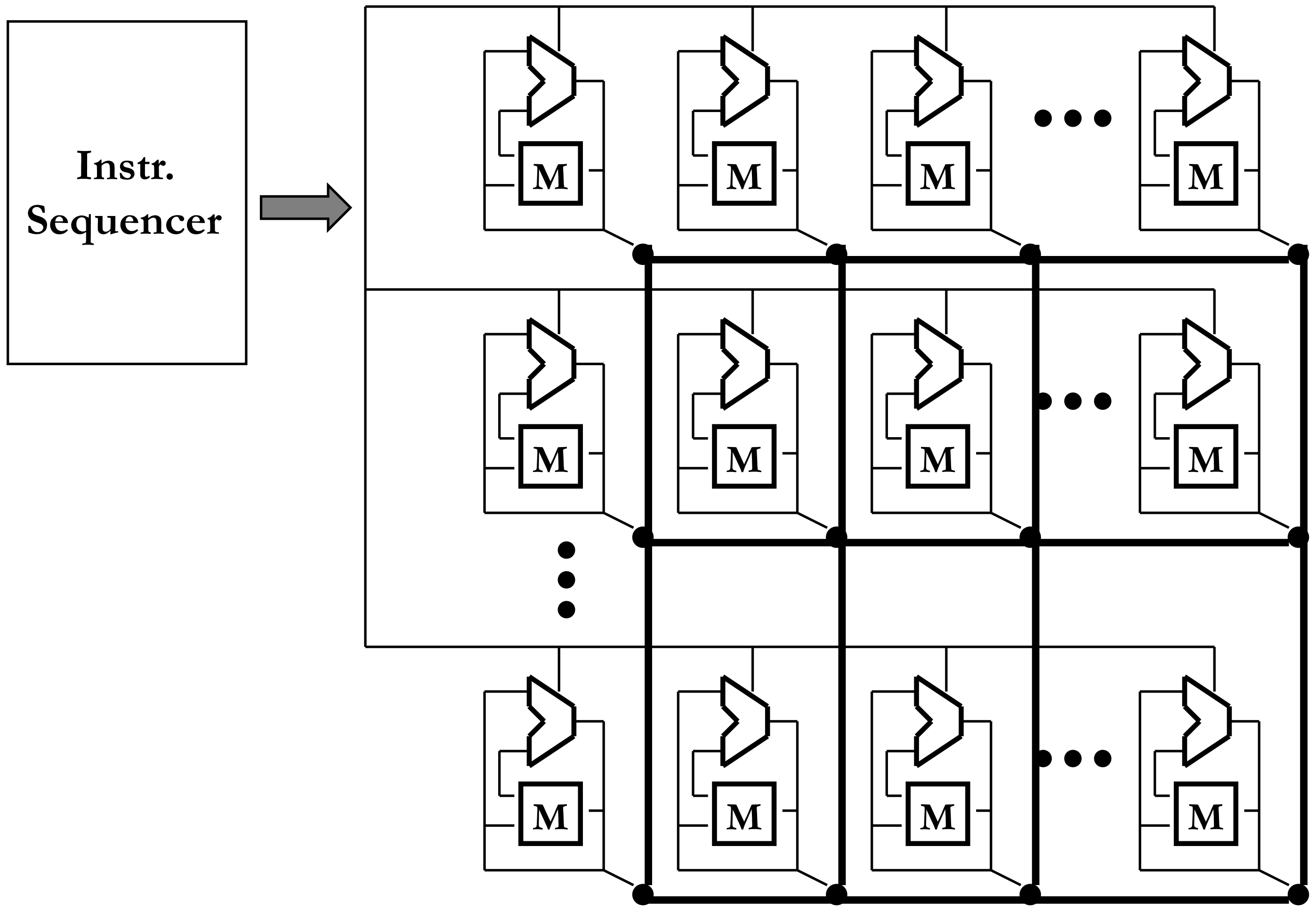
```
ADDV.D V1,V2,V3 # V1=V2+V3
MULV.D V4,V5,V1 # V4=V5+V1
```



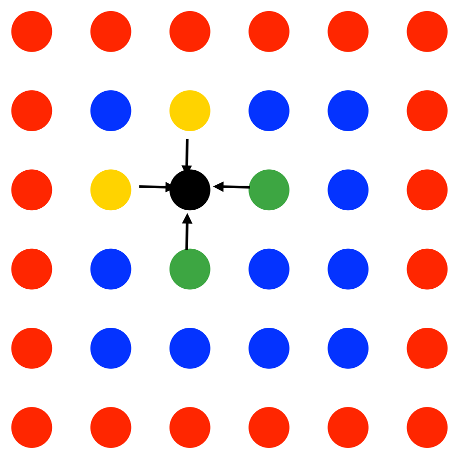
Original SIMD Idea

- Network of simple processing elements (PE)
  - PEs operate in lockstep under the control of a master sequencer
  - PEs can exchange results with a small number of neighbours via special data-routing instructions
  - Each PE has its own local memory
  - PEs operate on narrow operands
  - Very large (up to 64K) number of PEs
  - Usually operated as co-processors with a host computer to perform I/O and to handle external memory
- Intended for use as supercomputers
- Programmed via custom extensions of common HLL

Original SIMD Idea



Example: Equation Solver Kernel



- The problem:
  - Operate on a (n+2)x(n+2) matrix

$$\mathbf{A[i,j] = 0.2 \times (A[i,j] + A[i,j\text{-}1] + A[i\text{-}1,j] + A[i,j+1] + A[i+1,j])}$$

- SIMD implementation:
  - Assign one node to each PE
  - Step 1: all PE's send their data to their east neighbors and simultaneously read the data sent by their west neighbors
  - Steps 2 to 4: same as step 1 for west, south, and north (again, appropriate nodes are masked out)
  - Step 5: all PE's compute the new value using equation above

Multimedia SIMD Extensions

- Key ideas:
  - No network of processing elements, but an array of ALU's
  - No memories associated with ALU's, but a pool of relatively wide (64 to 128 bits) registers that store several narrower operands
  - No direct communication between ALU's, but via registers and with special shuffling/permutation instructions
  - Not co-processors or supercomputers, but tightly integrated into CPU pipeline
  - Still lockstep operation of ALU's

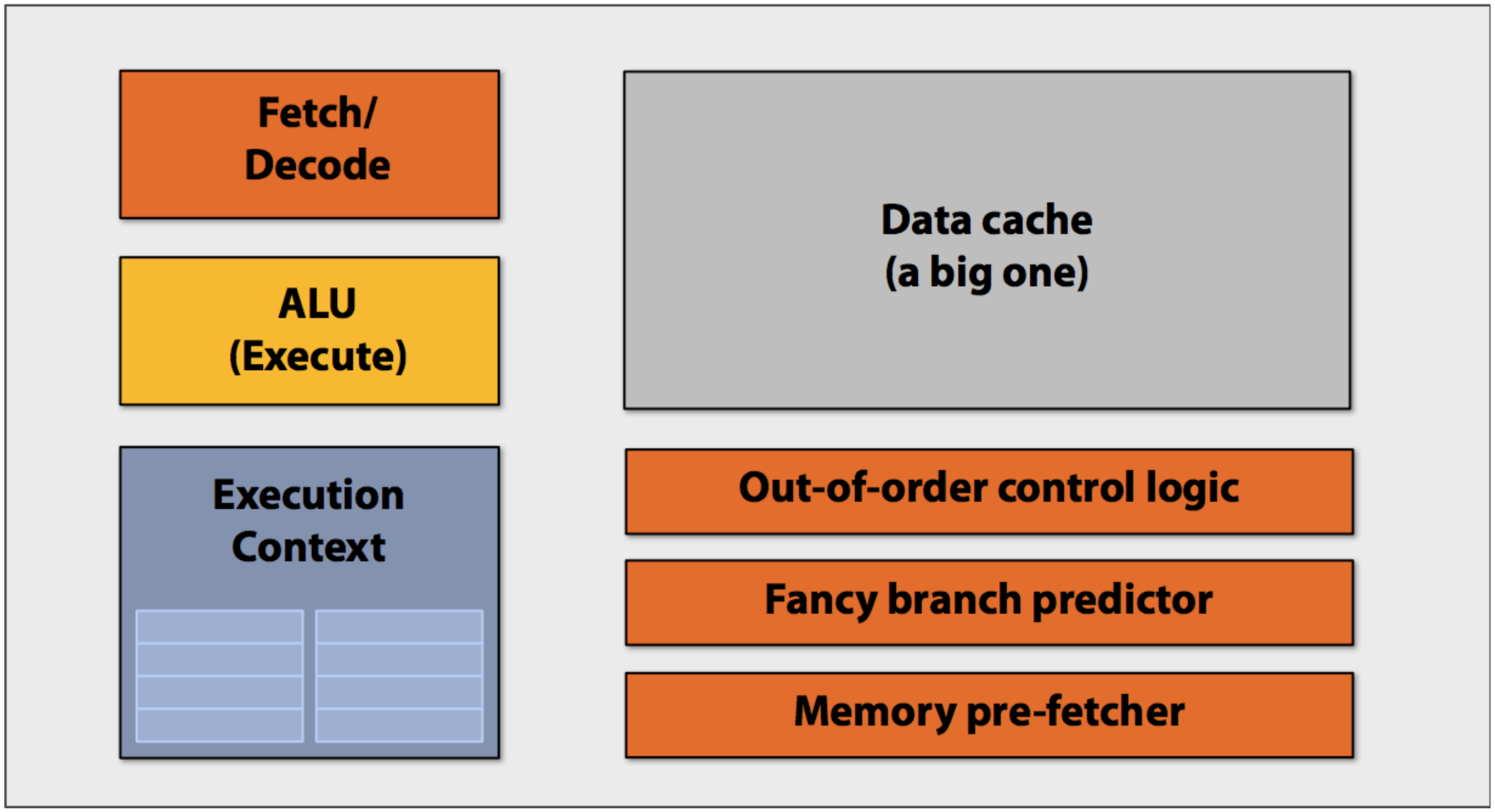
Example: Intel SSE

- **S**treaming **S**IMD **E**xtensions introduced in 1999 with Pentium III
- Improved over earlier MMX (1997)
  - MMX re-used the FP registers
  - MMX only operated on integer operands
- 70 new machine instructions (SSE2 added 144 more in 2001) and 8 128bit registers
  - Registers are part of the architectural state
  - Include instructions to move values between SSE and x86 registers
  - Operands can be: single (32bit) and double (64bit) precision FP; 8, 16, and 32 bit integer
  - SSE2 included instructions for handling the cache (recall that streaming data does not utilize caches efficiently)

Graphics Processing Unit (GPU)

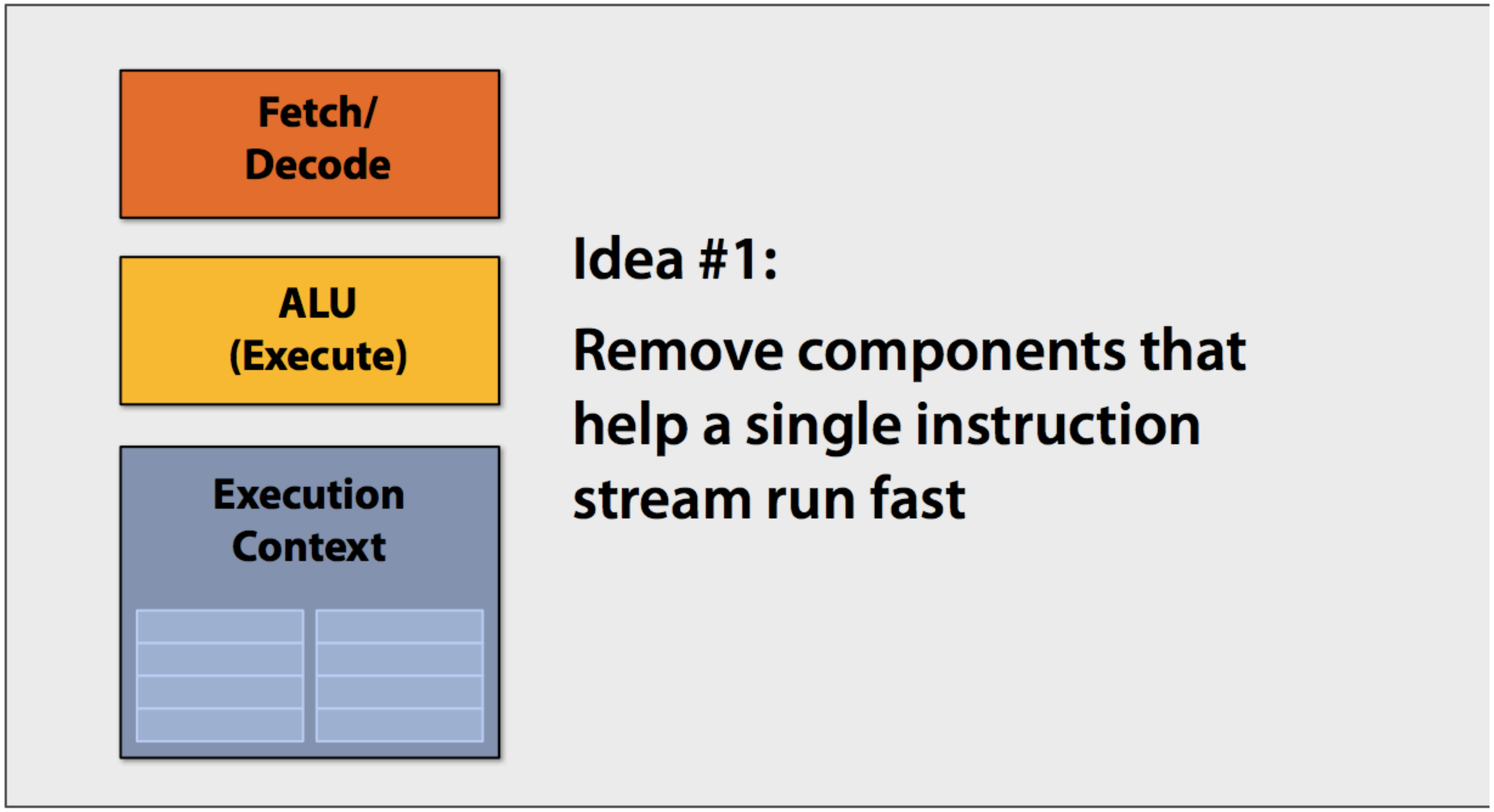
- Graphics apps have lot of parallelism
- Take advantage of hardware invested to do graphics well
- GPU is now ubiquitous!
- Several supercomputers in top500 use GPUs
  - Titan uses Nvidia tesla

CPU-style Core



Slide from Beyond programmable shading course ACM Siggraph '10

Slimming down…

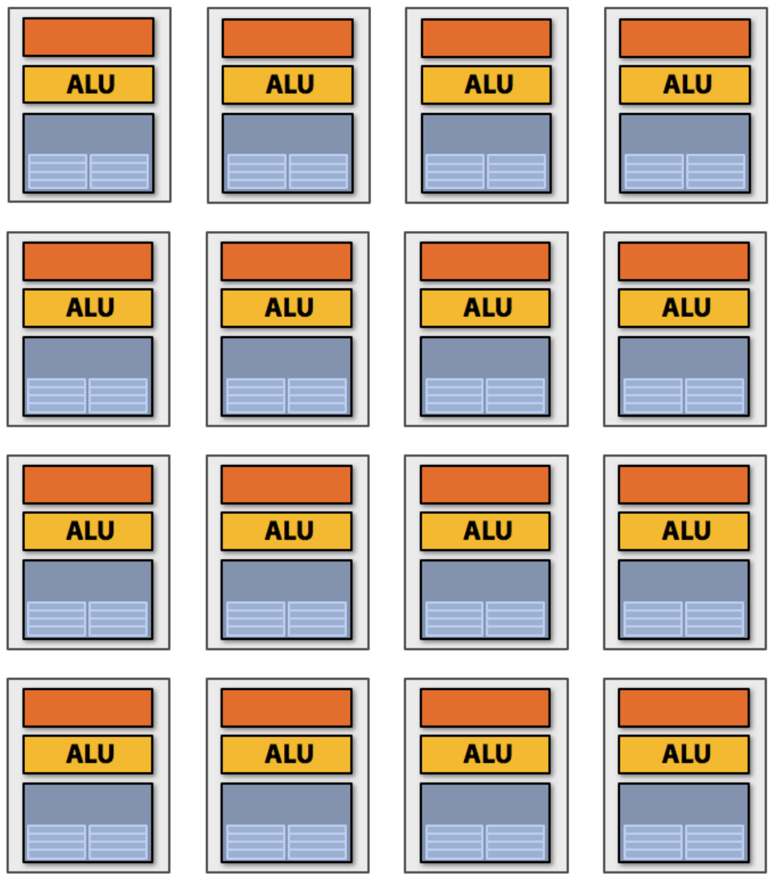


Idea #1:

Remove components that help a single instruction stream run fast

Slide from Beyond programmable shading course ACM Siggraph '10

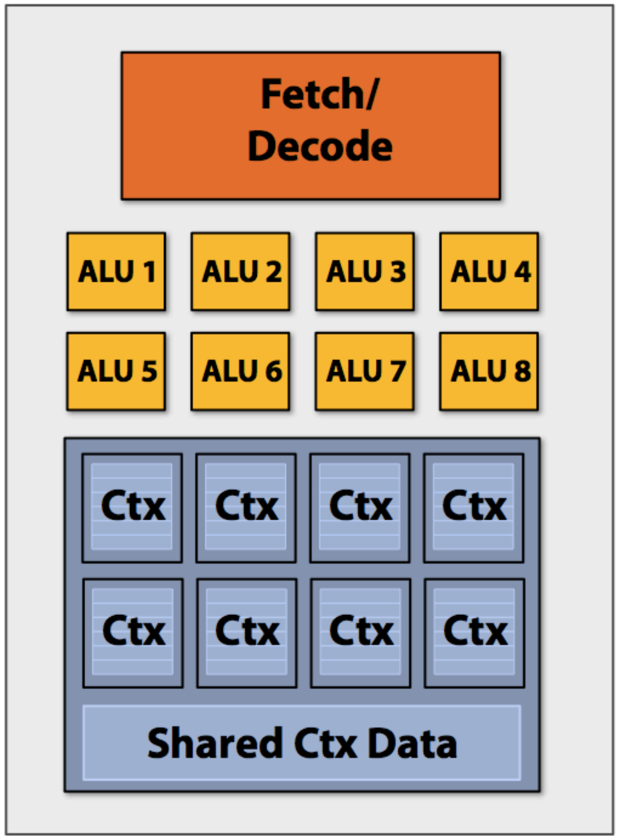
Add Cores…



16 cores = 16 simultaneous instruction streams

Slide from Beyond programmable shading course ACM Siggraph '10

Add ALUs (SIMD)…



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Slide from Beyond programmable shading course ACM Siggraph '10

Stalls!

- No caches
- No dynamic scheduling
- Dependencies and memory accesses can cause stalls!

Solution: Multithreading

But we have **LOTS** of independent fragments.

Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Slide from Beyond programmable shading course ACM Siggraph '10

GPU…

16 cores

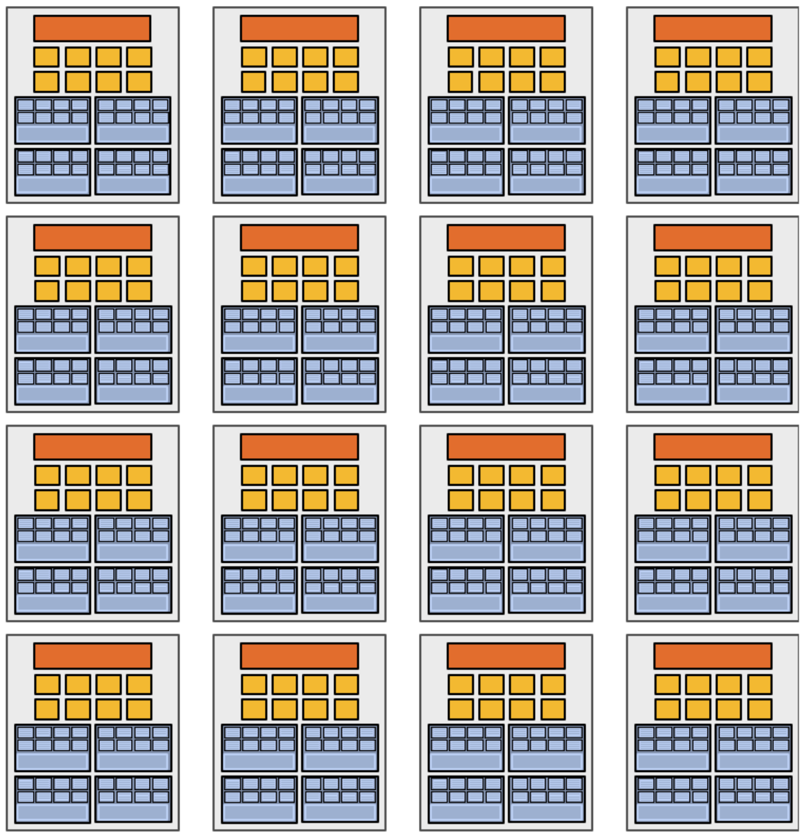
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs   (@ 1GHz)



Slide from Beyond programmable shading course ACM Siggraph '10

Further Reading

- The first truly successful vector supercomputer:

  "The CRAY-1 Computer System", R. M. Russel, Communications of the ACM, January 1978.

- Vector processor on a chip:

  "Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks", C. Kozyrakis and D. Patterson, Intl. Symp. on Microarchitecture, December 2002.

- Integrating a vector unit with a state-of-the-art superscalar:

  "Tarantula: A Vector Extension to the Alpha Architecture", R. Espasa, F. Ardanaz, J. Elmer, S. Felix, J. Galo, R. Gramunt, I. Hernandez, T. Ruan, G. Lowney, M. Mattina, and A. Seznec, Intl. Symp. on Computer Architecture, June 2002.

Further Reading

- Seminal SIMD work:
  - "A Model of SIMD Machines and a Comparison of Various Interconnection Networks", H. Siegel, IEEE Trans. on Computers, December 1979.
  - "The Connection Machine", D. Hillis, Ph.D. dissertation, MIT, 1985.
- Two commercial SIMD supercomputers:
  - "The CM-2 Technical Summary", Thinking Machines Corporation, 1990.
  - "The MasPar MP-1 Architecture", T. Blank, Compcon, 1990.
- SIMD co-processor:
  - "CSX Processor Architecture", ClearSpeed, Whitepaper, 2006.