Fall 2019
EE 6633: Architecture of Parallel Computers
Lecture 4: Multicore Architectures - 1

Avinash Karanth
Department of Electrical Engineering & Computer Science
Ohio University, Athens, Ohio 45701
E-mail: karanth@ohio.edu
Website: http://ace.cs.ohio.edu/~avinashk/classes/ee663/ee663.htm
Adapted from Prof. Kayvon Fatahalian (CMU)

1

Roadmap

- Parallel Execution
  - Superscalar, SIMD, multicores

2

Example Program

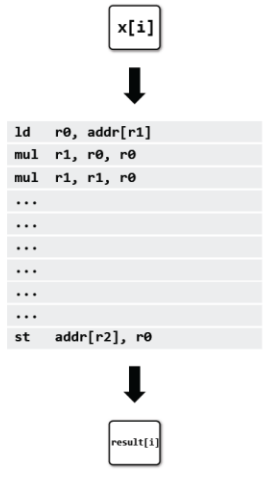
- Compute `sin(x)` using Taylor expansion: `sin(x) = x - x3/3! + x5/5! - x7/7! + …` for each element of an array of N floating point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

3

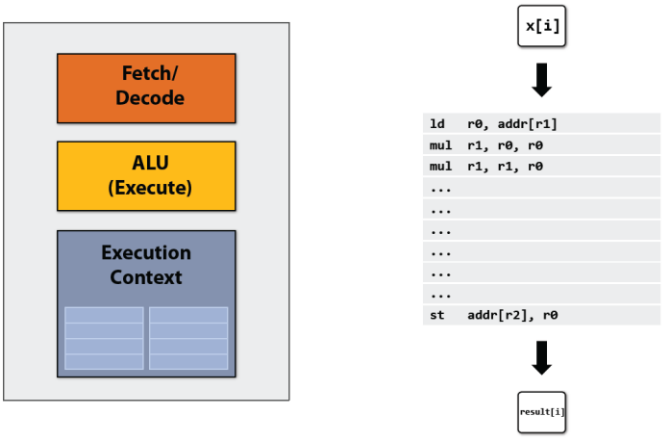
Compile Program

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```



4

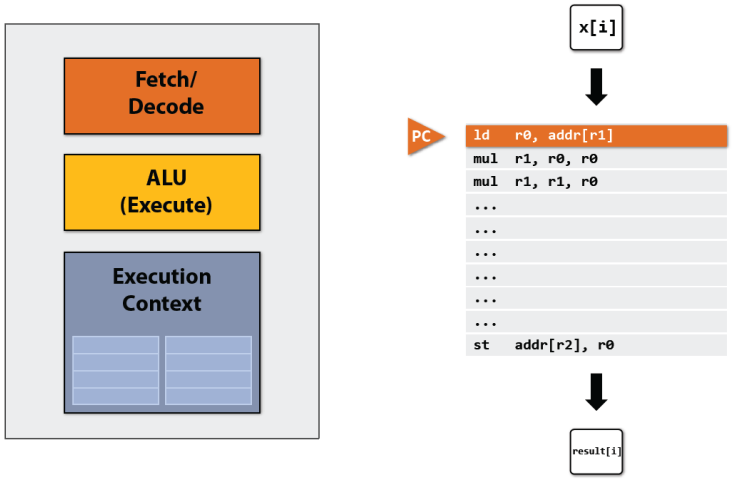
Execute the Program (1/4)



5

Execute Program (2/4)

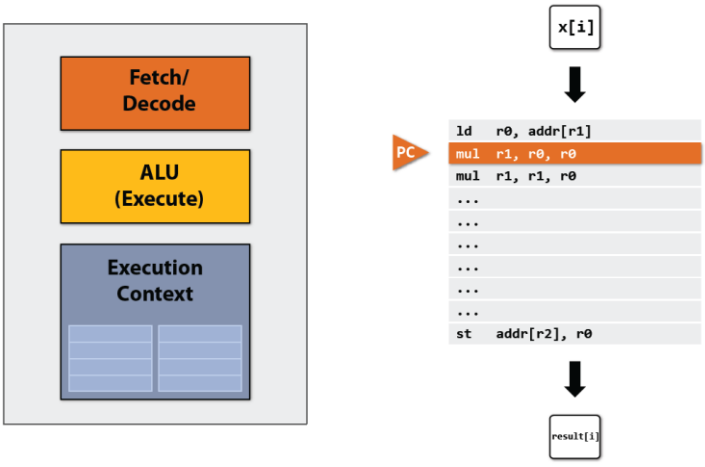
- My very simple processor: executes one instruction per clock



6

Execute Program (3/4)

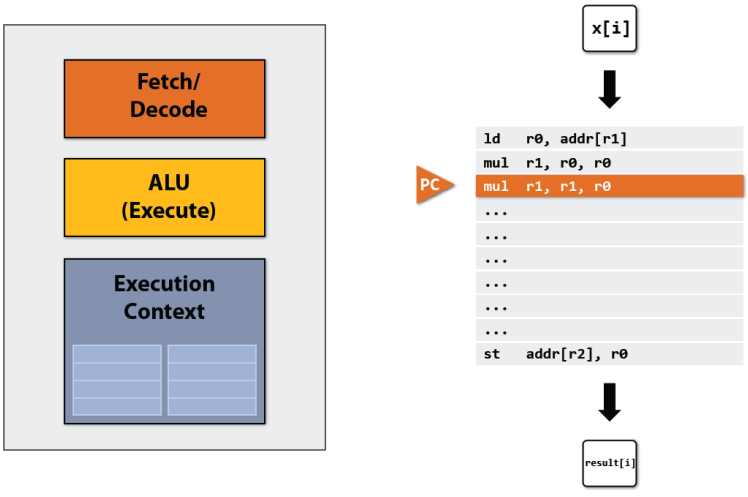
- My very simple processor: executes one instruction per clock



7

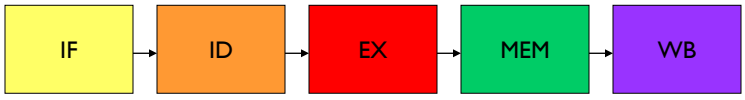
Execute Program (4/4)

- My very simple processor: executes one instruction per clock

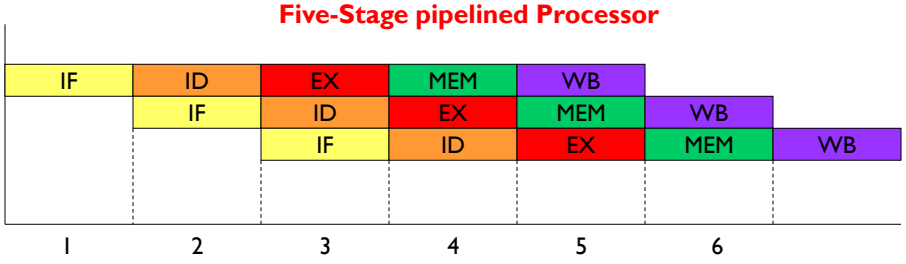


8

Base Machine



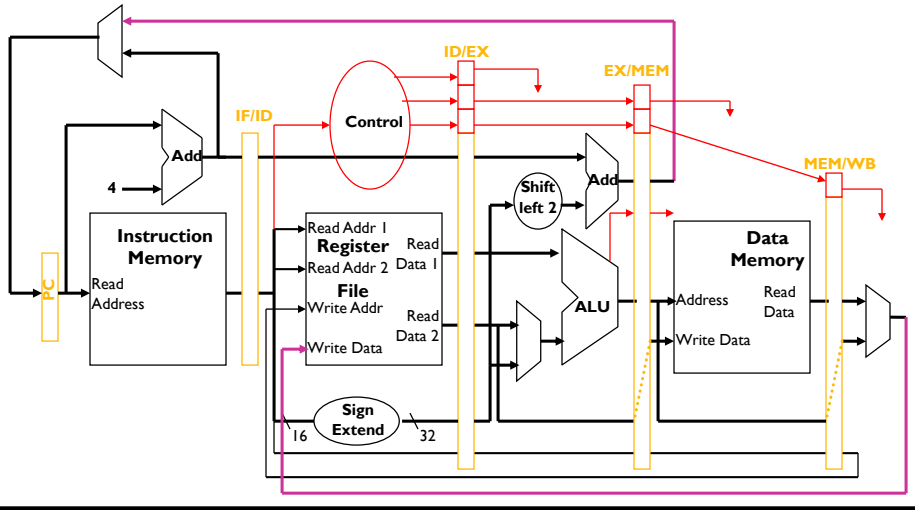
Five-Stage pipelined Processor



9

Pipeline

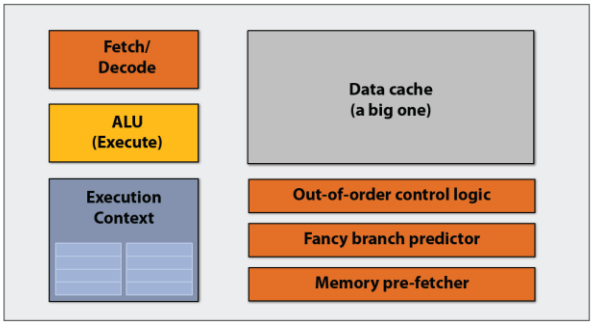
- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



10

Pre-Multicore Era

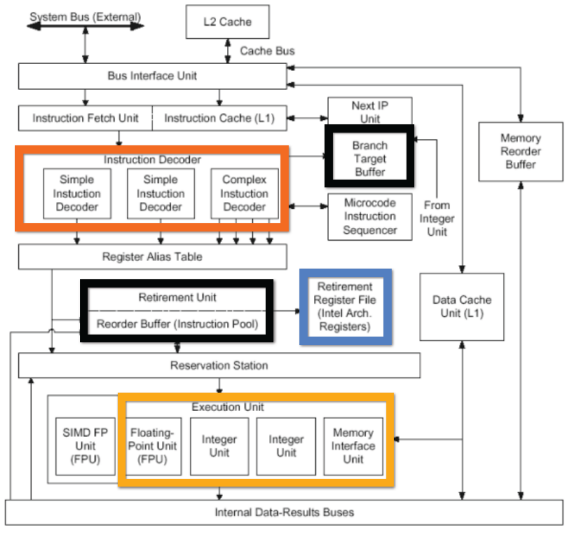
- Majority of chip transistors used to perform operations that help a single instruction stream run fast



- More transistors = larger cache, smarter order-of-order logic, smarter branch predictor
- Also, more transistors = smaller transistors = higher clock frequencies

11

Pentium Machine



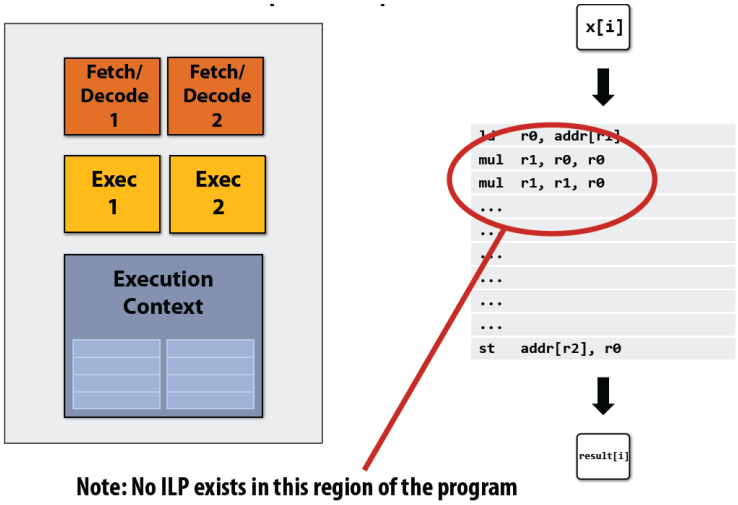
12

Superscalar & Superpipelining

- Are used to increase the performance of the base machines (like pure pipelined machines), other techniques can be used
- Some definitions for a pure base machines
  - Operation Latency (OL): Time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction
  - Issue Latency (IL): Time (in cycles) required between issuing two adjacent instructions
  - Issue rate (IR): The number of instructions issued per cycle
  - Base Machine: OL = IL = IR = 1

13

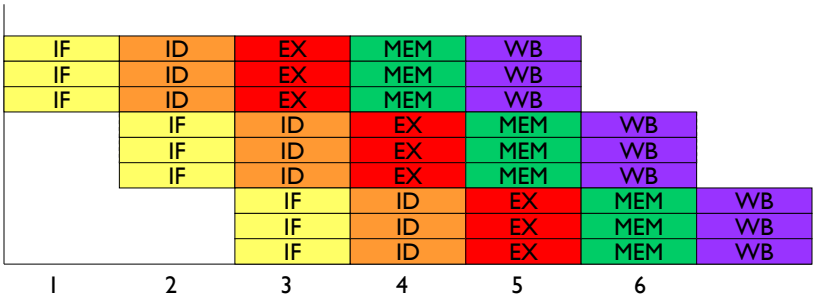
Superscalar Processor



14

Superscalar

- The major innovation of the superscalar architecture over the traditional pipelined processor is the multiple instruction issue advantage through dynamic instruction scheduling



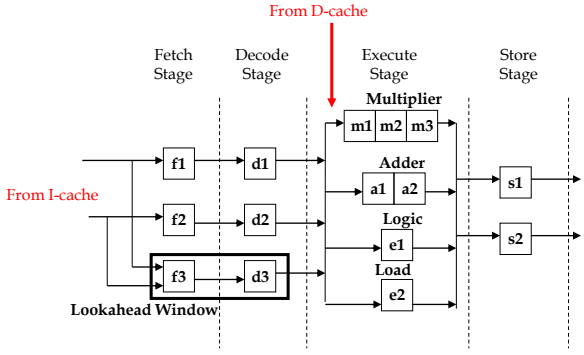
15

Superscalar

- *m* instructions per cycle are issued
- The IF, ID and EX units are essentially duplicated *m* times plus some extra hardware to take care of data dependencies among simultaneous instructions, some units could be shared
- IL = 1, OL = 1, IR = *m* (degree of superscalar)

16

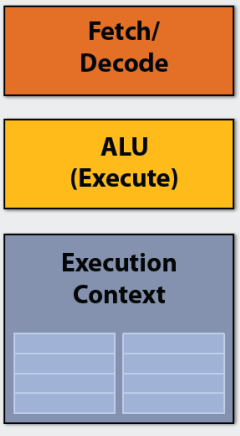
Dual-Pipeline Superscalar Processor: An Example




```
I1. Load R1, A          / R1 ← Memory(A)
I2. Add R2, R1          / R2 ← (R2) + (R1)
I3. Add R3, R4          / R3 ← (R3) + (R4)
I4. Mul R4, R5          / R4 ← (R4)x(R5)
I5. Comp R6             / R6 ← (R6)'
I6. Mul R6, R7          / R6 ← (R6)x(R7)
```

17

Multicore Era



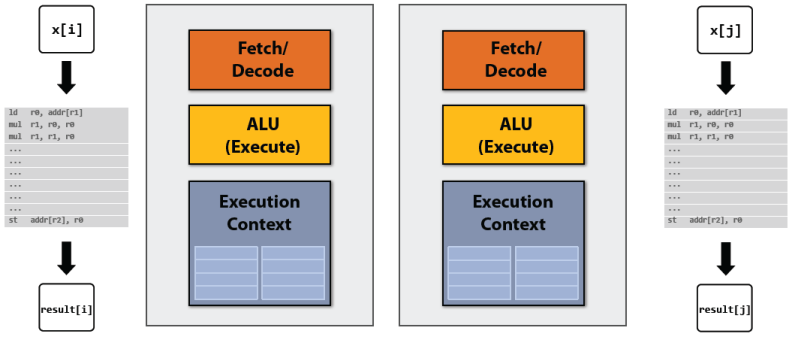
Idea #1:

Use increasing transistor count to add more cores to the processor

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

18

Two cores in parallel



- Simpler cores: Each core is slower at running a single instruction stream than our original "fancy" core (e.g. 25% slower)
- But there are two cores: 2 x 0.75 = 1.5 (potential for speedup)

19

How to exploit parallelism?

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

- This program exploits no parallelism and when compiled with gcc will run as one thread on one of the cores.
- Now if each core runs slower by 25% from the original core, then our program will run 25% slower on a multicore!

20

Exploiting Parallelism via pthreads

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

21

Data-Parallel Expression (1/2)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

Loop iterations declared by the programmer to be independent

With this information, you could imagine how a compiler might automatically generate parallel threaded code

22

Data Parallel Expression (2/2)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

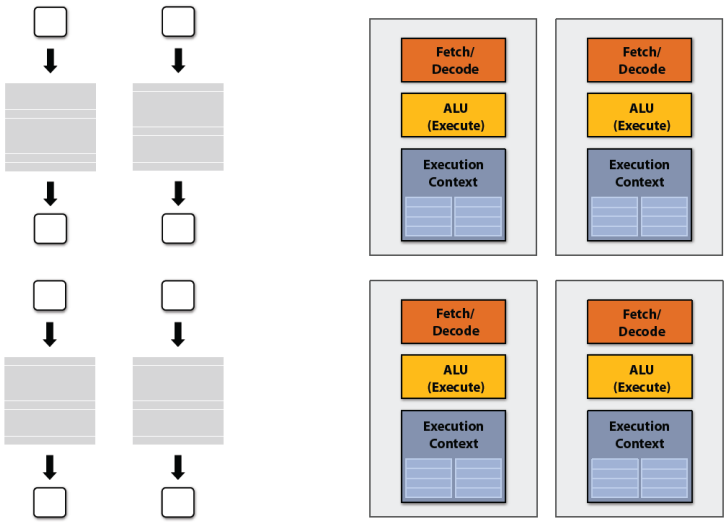
Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop do the same thing: evaluate the sine of a single input number

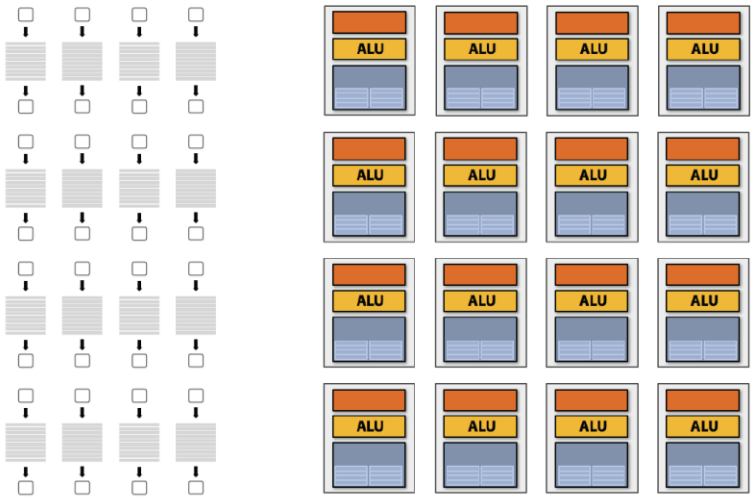
23

Four Cores: Compute 4 elements in parallel



24

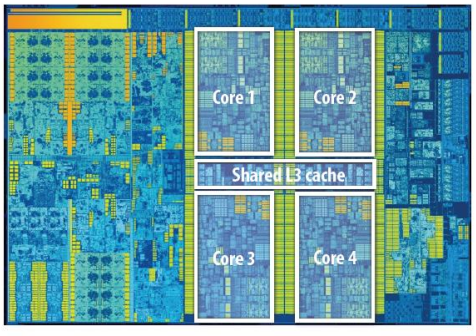
16 core



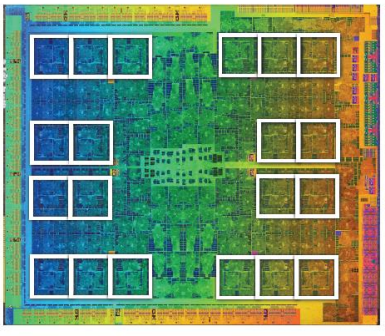
Sixteen cores, sixteen simultaneous instruction streams

25

Multicore Examples (1/2)



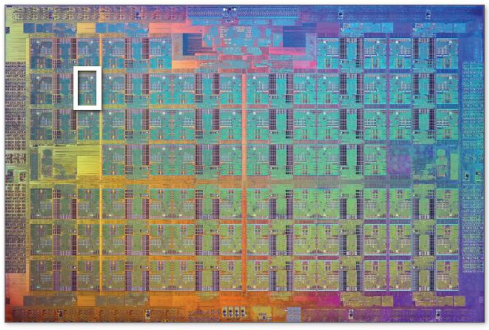
Intel "Skylake" Core i7 quad-core CPU
(2015)



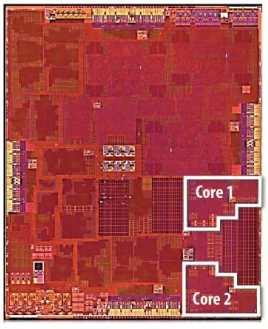
NVIDIA GP104 (GTX 1080) GPU
20 replicated ("SM") cores
(2016)

26

Multicore Examples (2/2)



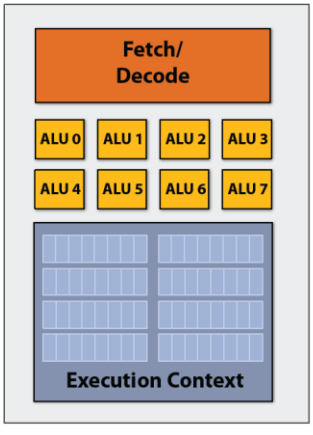
Intel Xeon Phi "Knights Corner" 72-core CPU (2016)



Apple A9 dual-core CPU (2015)

27

Add ALUs to Increase Computation!




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Recall original compiled program:

Instruction stream processes one array element at a time using scalar instructions on scalar registers (e.g., 32-bit floats)

28

Scalar Program

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

Original compiled program:

Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

29

Vector Program (using AVX Instrinsics) (1/2)

Intrinsics available to C programmers

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
      __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         // value += sign * numer / denom
         __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
         value = _mm256_add_ps(value, tmp);

         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
         sign *= -1;
      }
      _mm256_store_ps(&result[i], value);
   }
}
```

30

Vector Program (using AVX Instrinsics) (2/2)

```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         // value += sign * numer / denom
         __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom);
         value = _mm256_add_ps(value, tmp);

         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
         sign *= -1;
      }
      _mm256_store_ps(&sinx[i], value);
   }
}
```

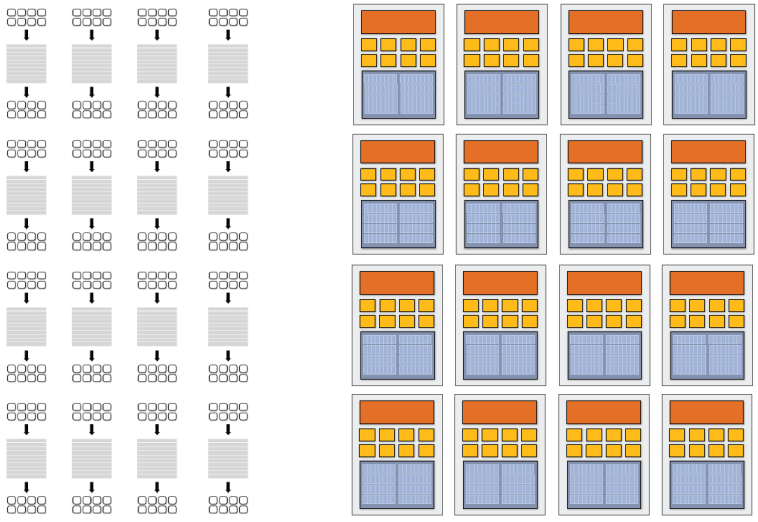
```
vloadps  xmm0, addr[r1]
vmulps   xmm1, xmm0, xmm0
vmulps   xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

31

16 SIMD Cores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

32

Data-parallel expression

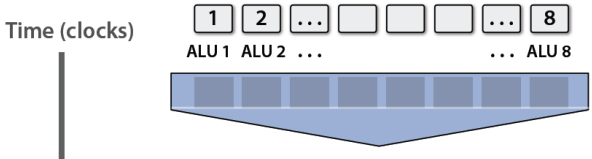
```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

Compiler understands loop iterations are independent, and that same loop body will be executed on a large number of data elements.

Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

33

What about conditional execution?

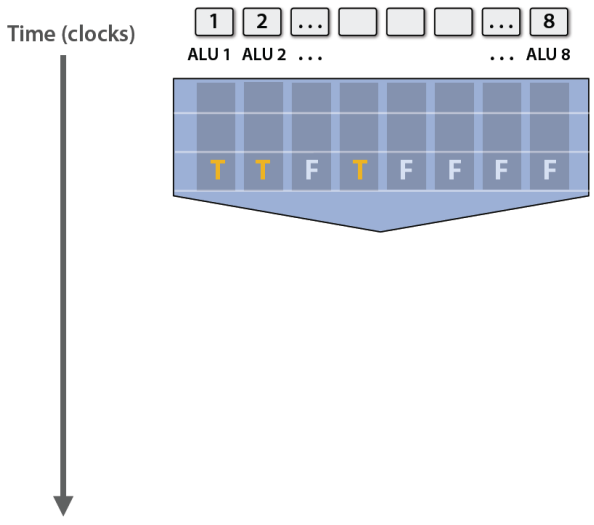


(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

34

What about conditional execution?

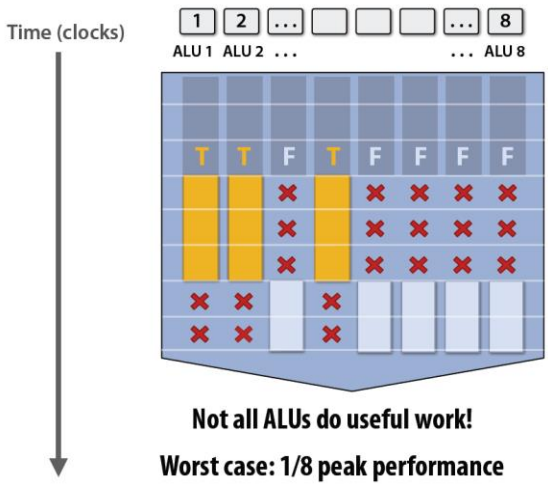


(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>
float x = A[i];
if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}
<resume unconditional code>

result[i] = x;
```

35

Mask (discard) output of ALU

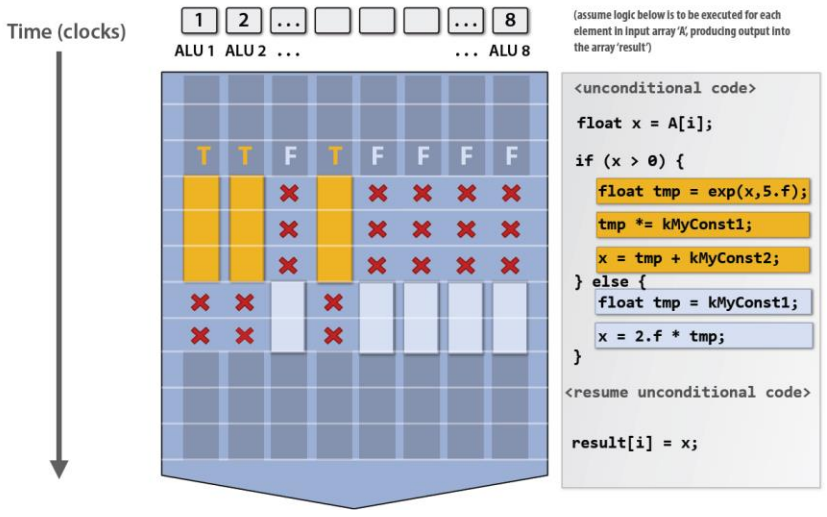


(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>
float x = A[i];
if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}
<resume unconditional code>

result[i] = x;
```

36

After Branch Continue at Full Pace



37

Terminology

- Instruction stream coherence ("coherent execution")
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- Divergent execution
  - A lack of instruction stream coherence

38

SIMD Execution on modern CPUs

- SSE (streaming SIMD Extensions):
  - 128-bit operations: 4 x 32 bits or 2 x 64 bits (4-wide float vectors)
- AVX2 (advanced vector extensions):
  - 256-bit operations: 8 x 32 bits or 4 x 64 bits (8-wide float vectors)
- AVX512 instructions:
  - 512-bit operations: 16 x 32 bits …
- Instructions are generated by the complier
  - Parallelism is explicitly requested by the programmer using instrinsics
  - Parallelism is conveyed using parallel language semantics (e.g. `forall` example)
  - Parallelism inferred by dependency analysis of loops (hard problem, even the best compiler are not great on arbitrary c/c++ code
- Terminology: "explicit SIMD": SIMD parallelization is performed at compile time
  - Can inspect program binary and see instructions (`vstoreps, vmulps`, etc)

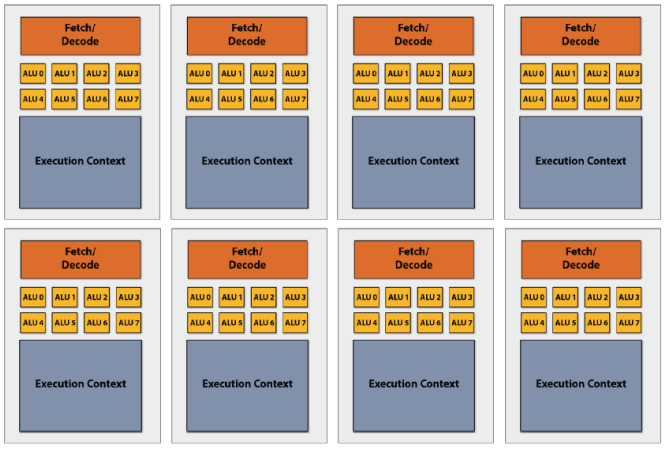
39

SIMD Execution on modern GPUs

- "Implicit SIMD"
  - Compiler generates a scalar binary (scalar instructions)
  - But N instances of the program are *always run* together on the processor
    `execute(my_function, N) \\ execute my_function N times`
  - In other words, the interface to the hardware itself is data-parallel
  - Hardware, and not compiler is responsible for simultaneously executing the same instructions from multiple instances of different data on SIMD ALUs
- SIMD width of most modern GPUs range from 8 to 32
  - Divergence can be a big issue as poorly written code might execute at 1/32 the peak capability of the machine!

40

Example: Eight-core Intel Xeon E5-1660 v4



8 cores
8 SIMD ALUs per core (AVX2 instructions)

490 GFLOPs (@3.2 GHz) (140 Watts)

41

Example: NVIDIA GTX 1080 GPU

20 cores ("SMs")
128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)

42

Summary: Parallel Execution

- Several forms of parallel execution in modern processors
- Multicore: use multiple processing cores
  - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
  - Software decides when to create threads (e.g. via pthreads API)
- SIMD: use multiple ALUs controlled by the same instruction stream (within a core)
  - Efficient design for data-parallel workloads – control amortized over many ALUs
  - Vectorization can be done by the compiler (explicit SIMD) or at runtime by hardware
  - [Lack of] dependencies is known prior to execution (usually declared by the programmer, but can be inferred by loop analysis by advanced compiler)
- Superscalar: exploit the ILP within an instruction stream
  - Process different instructions from the same stream in parallel (within a core)
  - Parallelism is automatically and dynamically discovered by the hardware during execution (not programmer visible)