

B3CC: Concurrency

08: Parallelism

Tom Smeding

Announcement

- Midterm exam
 - Tuesday December 17th @ 13:30 – 15:30 (2h): Olympos hal 1
 - "Minder massaal": BBG 0.20 @ 13:30 (only go here if you know you should be here)

Recap

- **Concurrency:** dealing with lots of things at once
 - Collection of independently executing processes
 - Two or more threads are making progress
- **Parallelism:** doing lots of things at once
 - Simultaneous execution of (possibly related) computations
 - At least two threads are executing simultaneously

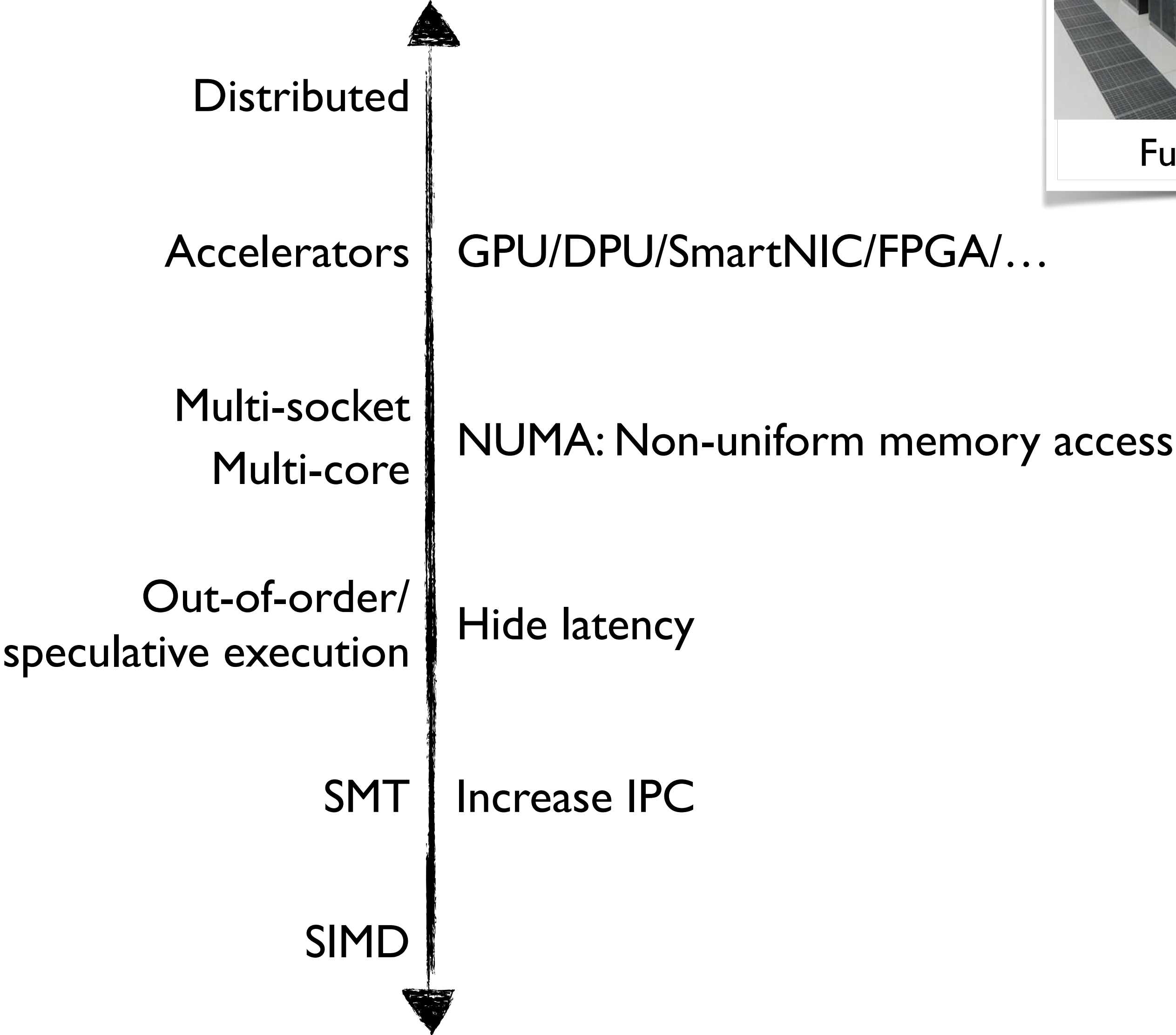
Recap

- So far we have discussed concurrency as a means to write modular code with multiple interactions
 - Example: network server that interacts with multiple clients simultaneously
 - Sometimes this can speed up the program by overlapping the I/O or time spent waiting for clients to respond, but this speedup doesn't require multiple processors to achieve
- In many cases we can use the same method to achieve *real* parallelism
 - We have used this as a way to test whether your programs can actually execute concurrently
 - From now, we will talk about some of the considerations for doing this with intent

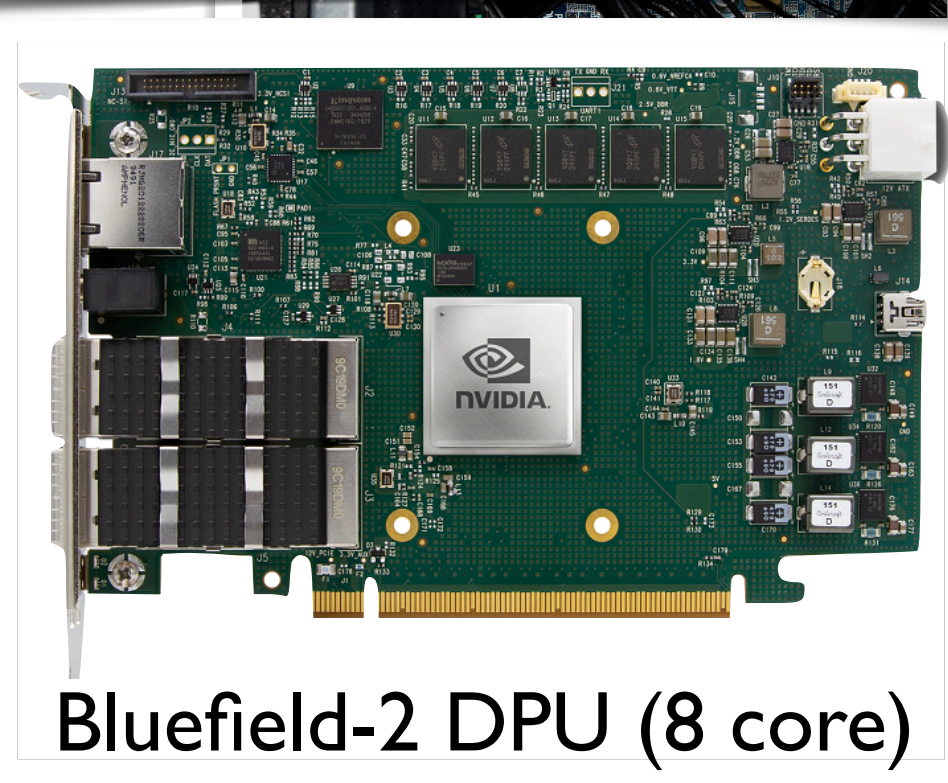
Parallelism

A (mostly) hardware perspective

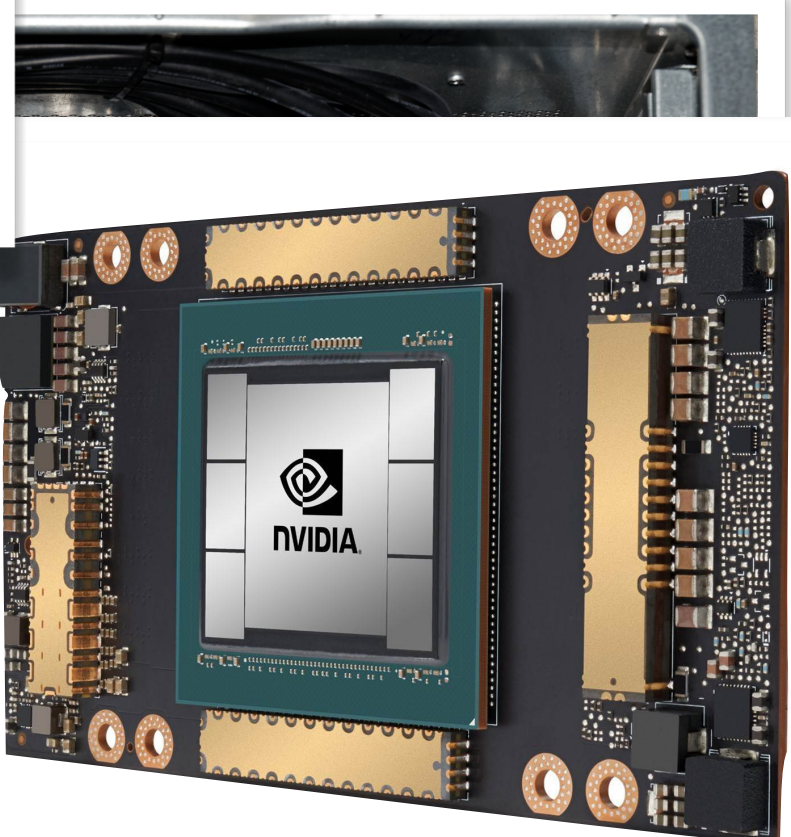
Where is the parallelism?



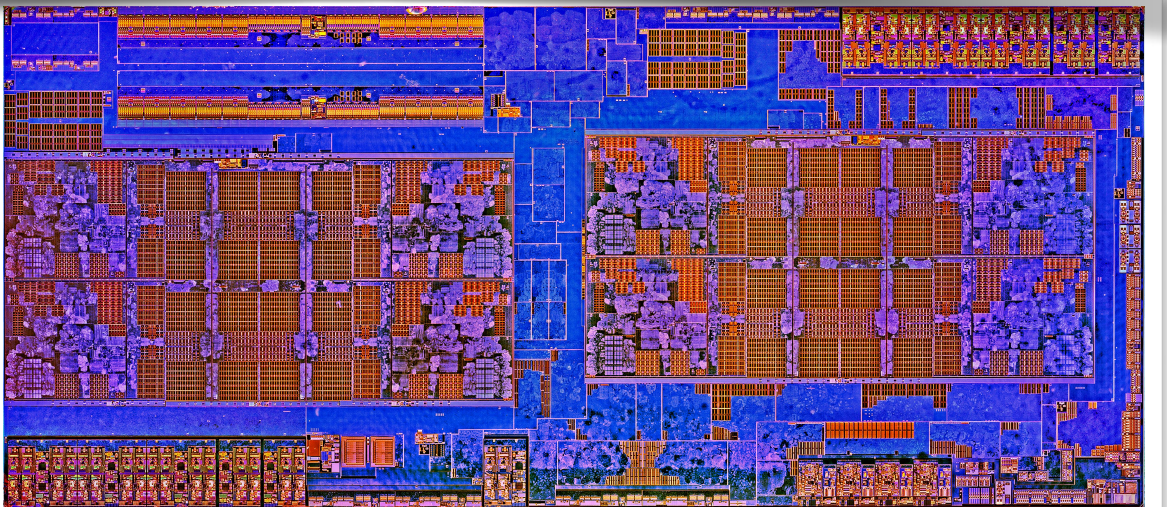
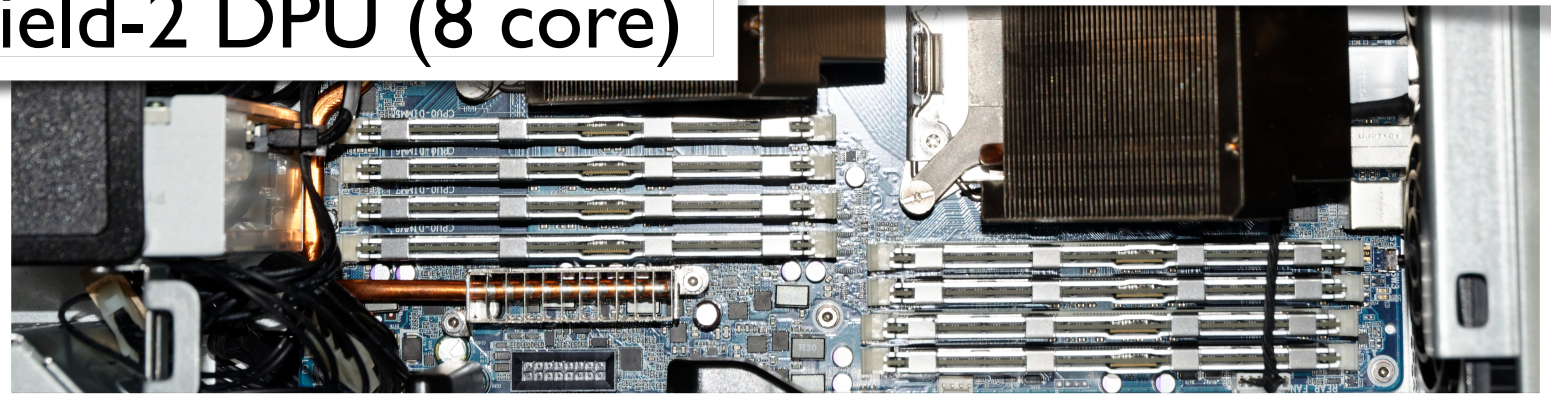
Fugaku (158,976 x 48 core)



Bluefield-2 DPU (8 core)

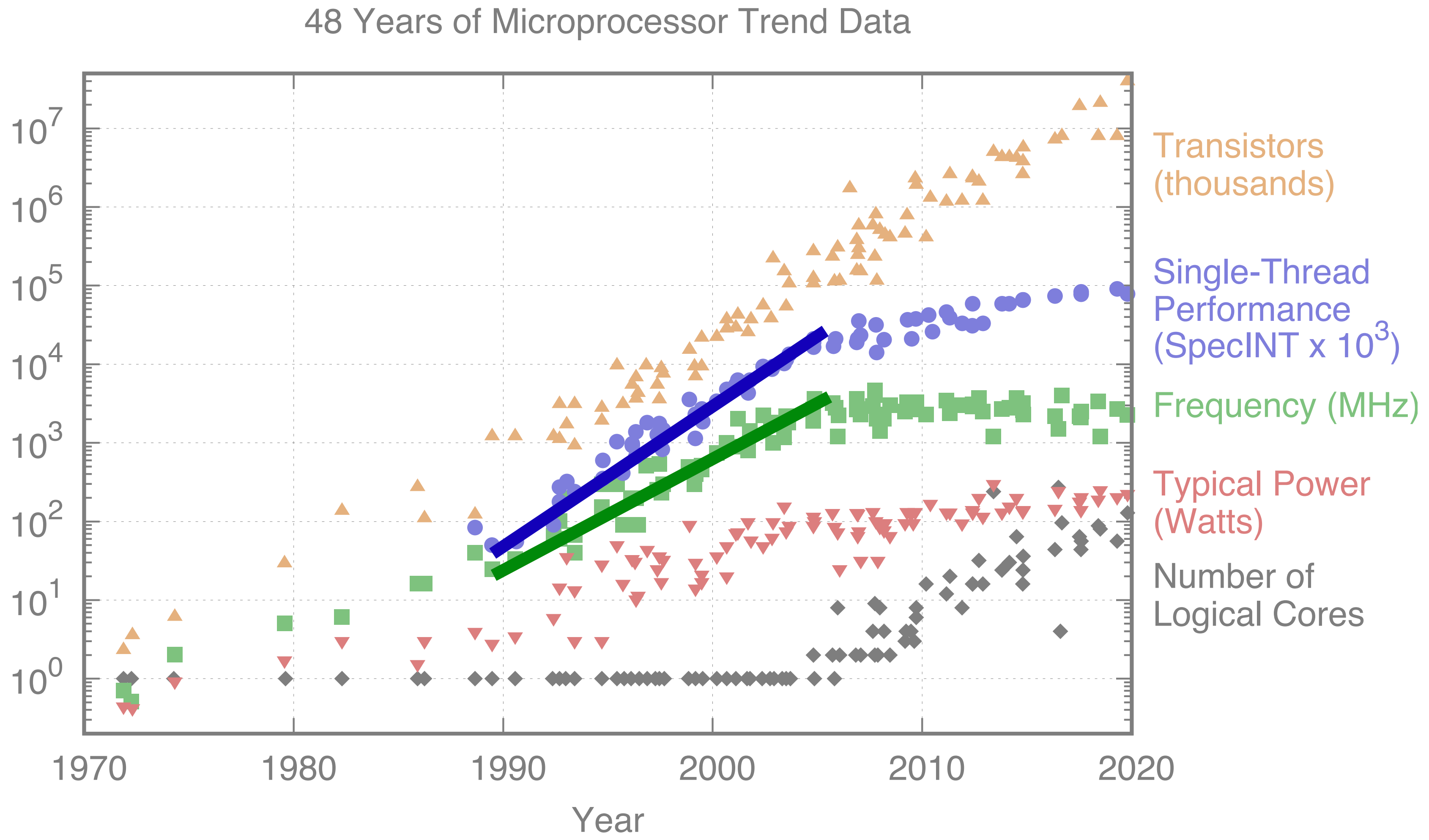


A100 GPU (6912 core)



Ryzen 7 1800X CPU (8 core)

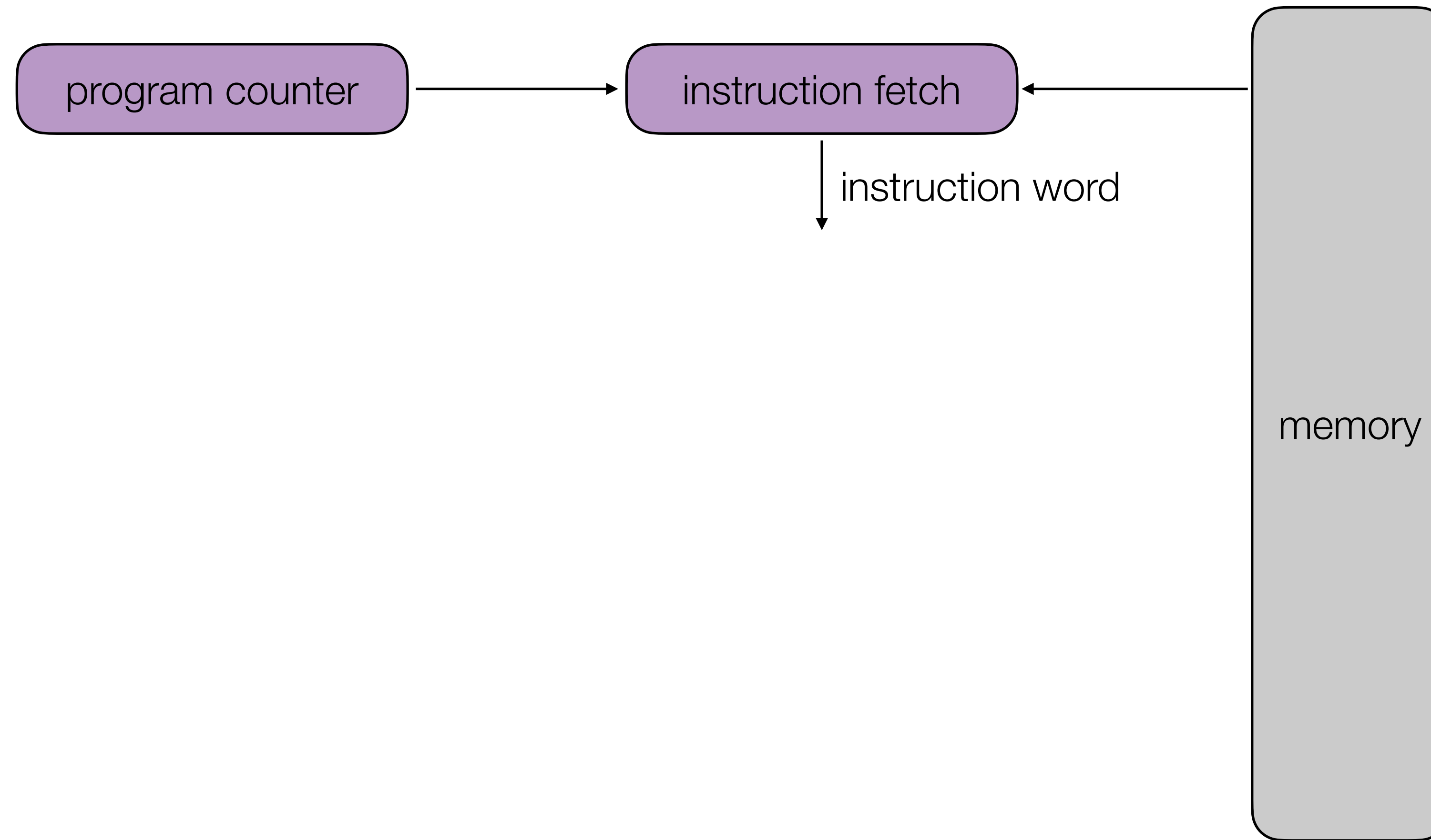
Recall



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

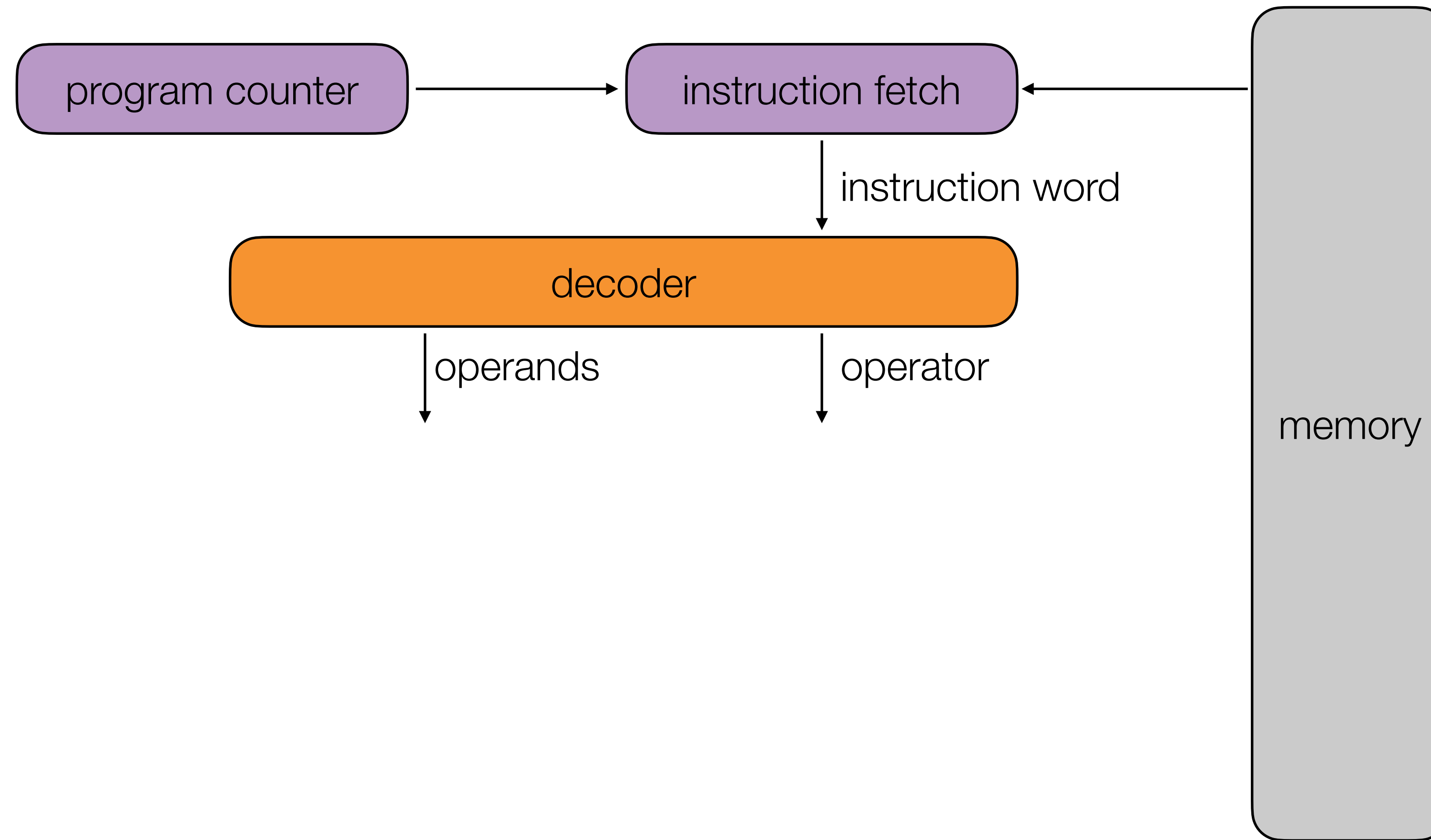
How does a processor work?

- Fetch instruction pointed to by PC in memory



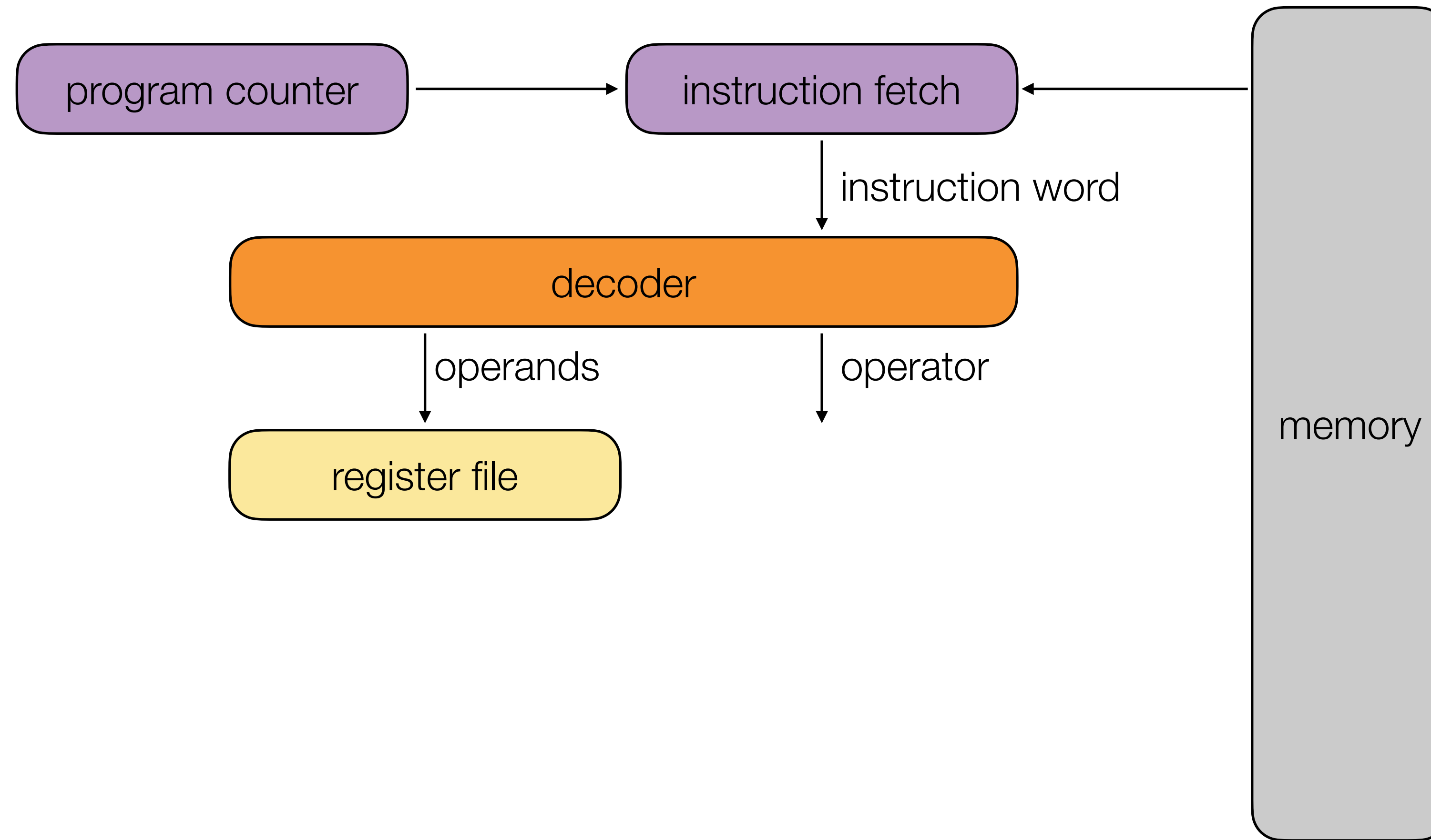
How does a processor work?

- Decode instruction into operator/operand



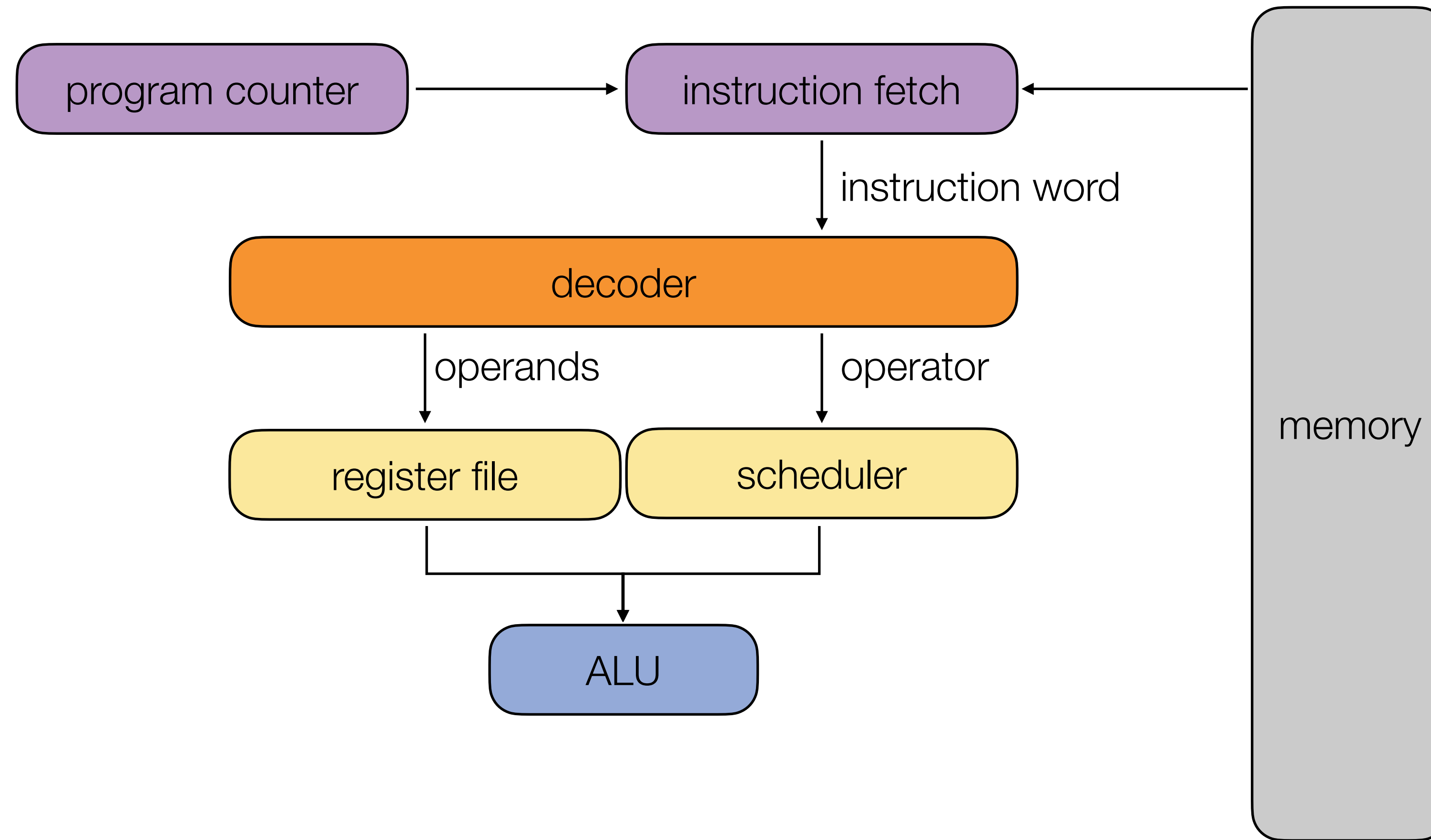
How does a processor work?

- Get operands from the register file



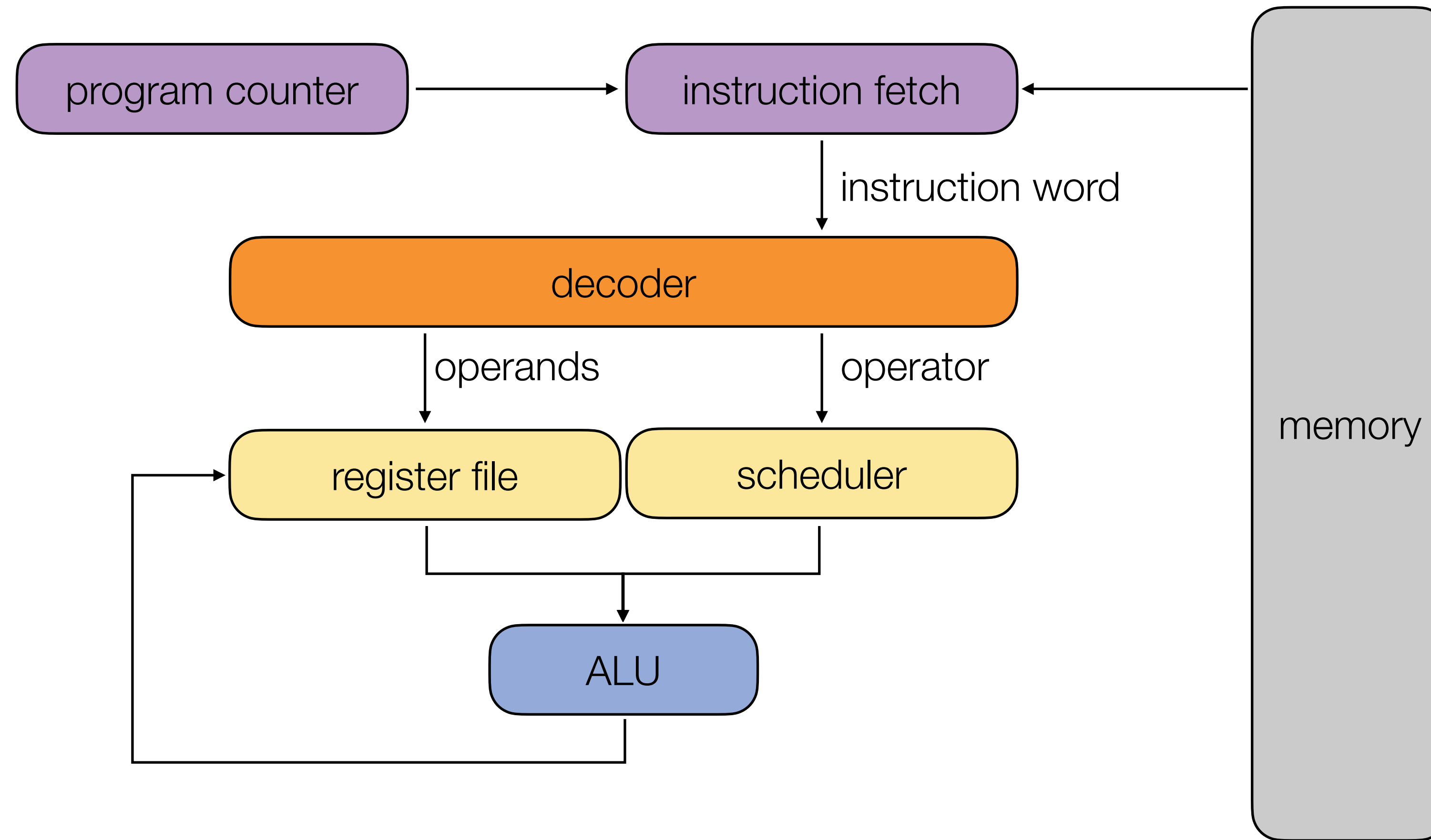
How does a processor work?

- Execute the instruction



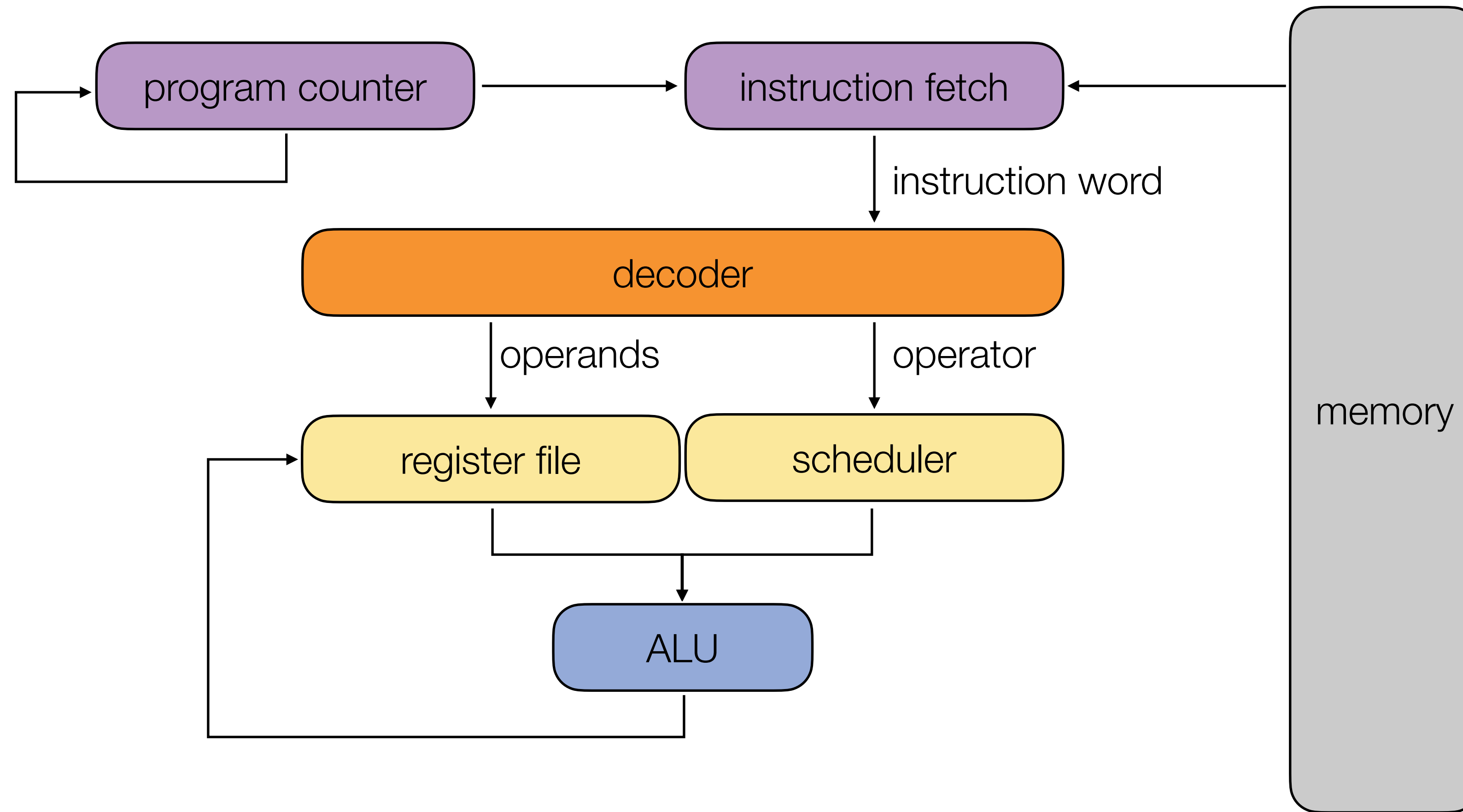
How does a processor work?

- Write back result to the register file



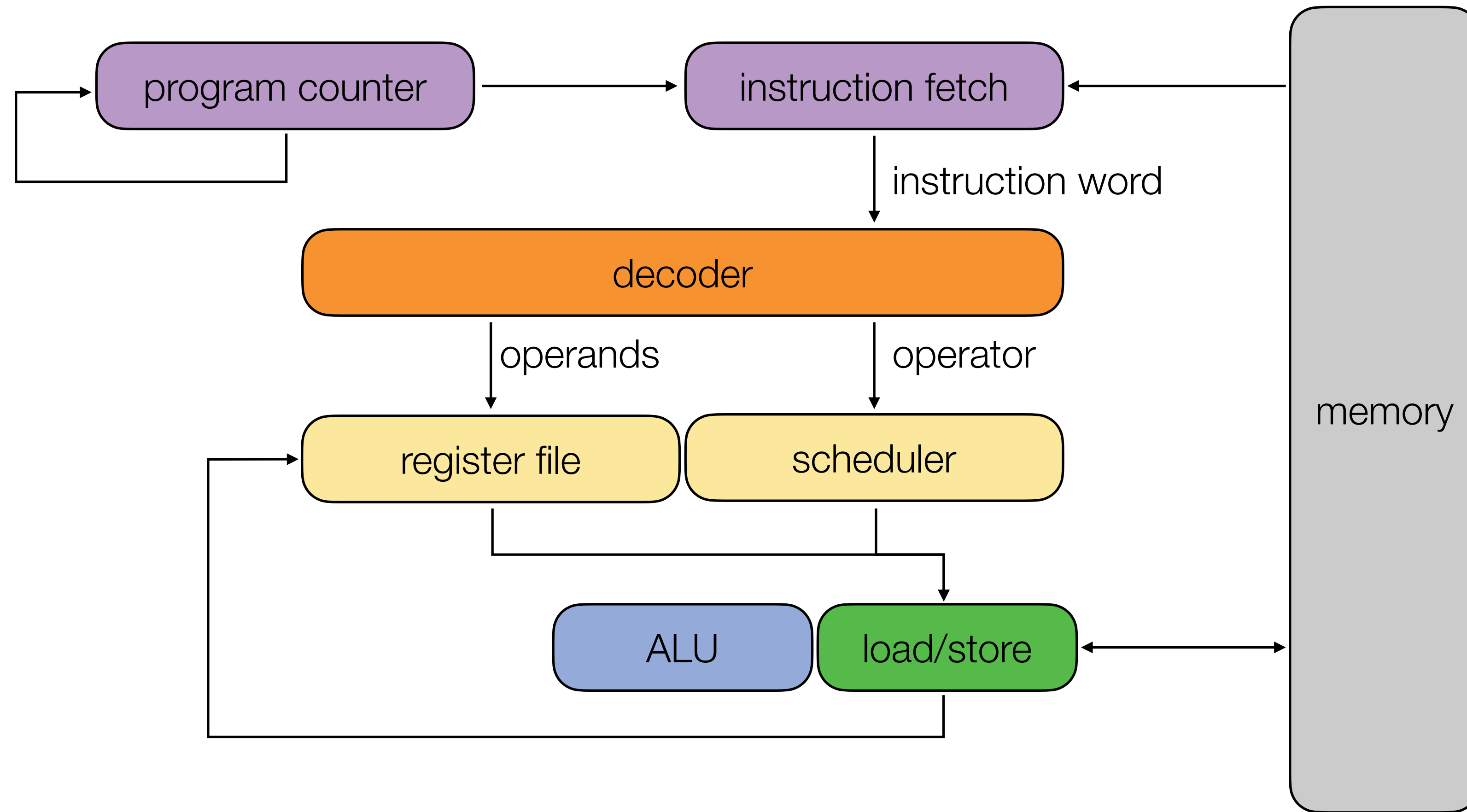
How does a processor work?

- Increment program counter



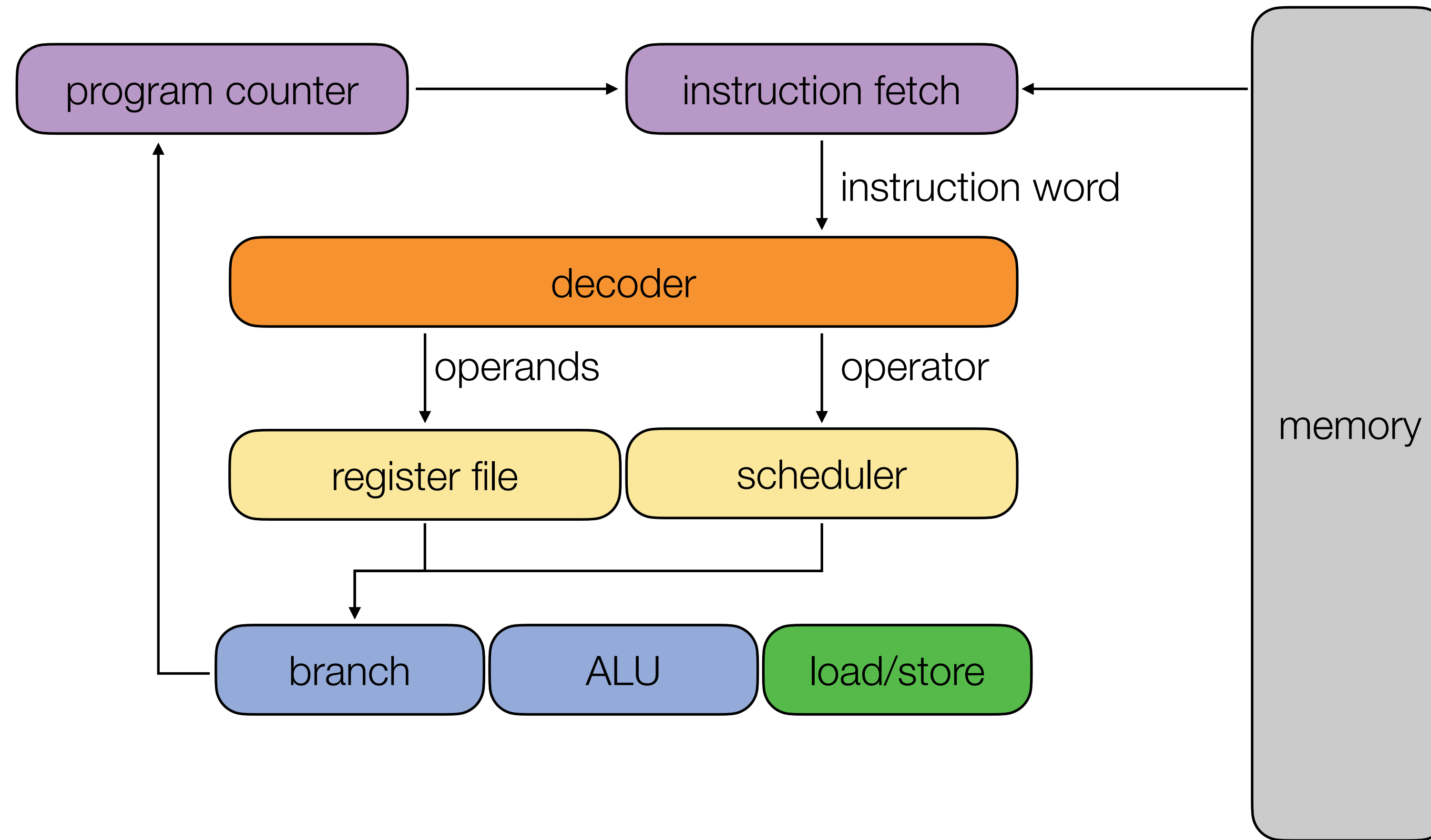
How does a processor work?

- Read and write memory from a computed address

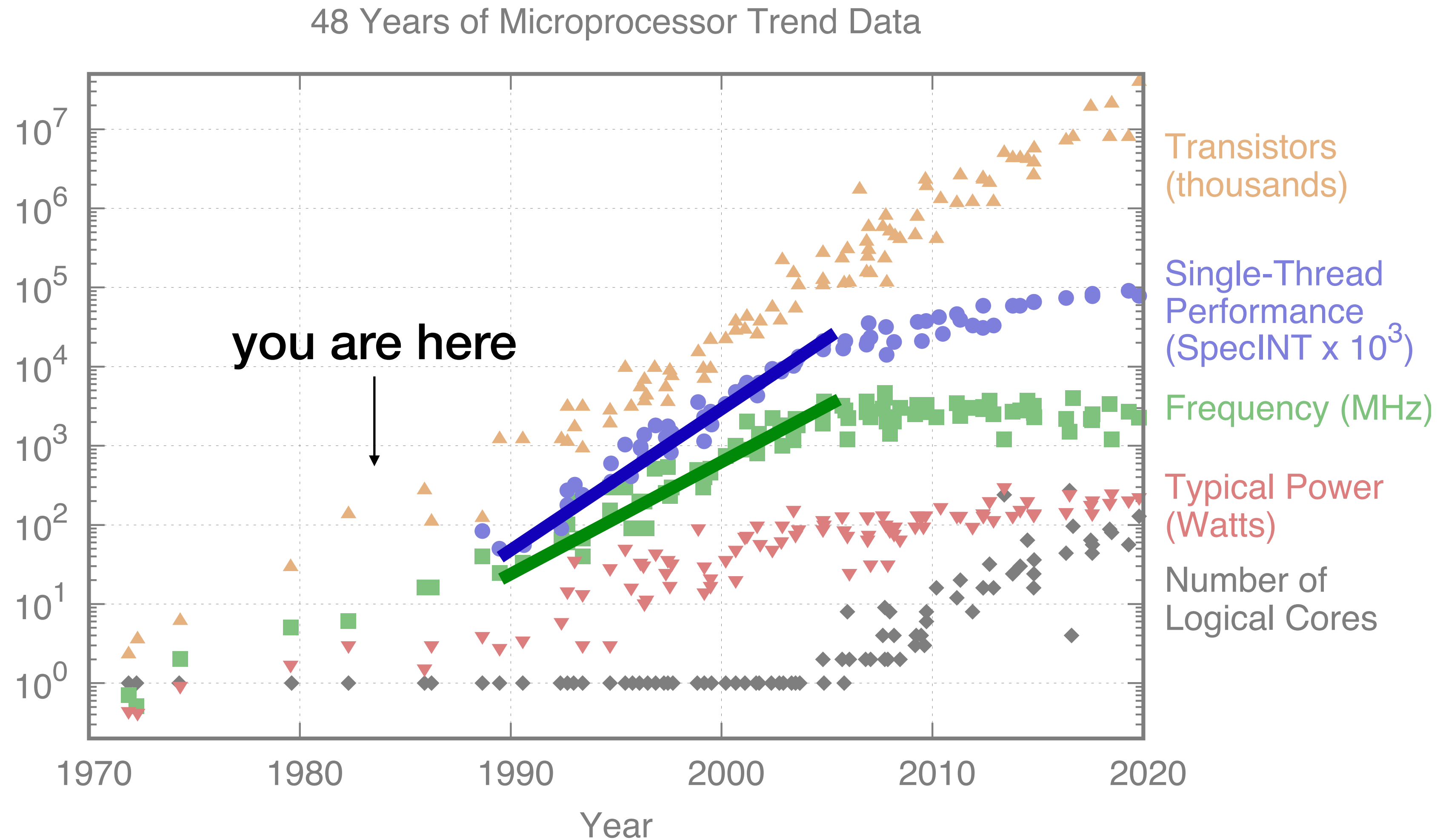


How does a processor work?

- Instead of incrementing the PC, set it to a computed value



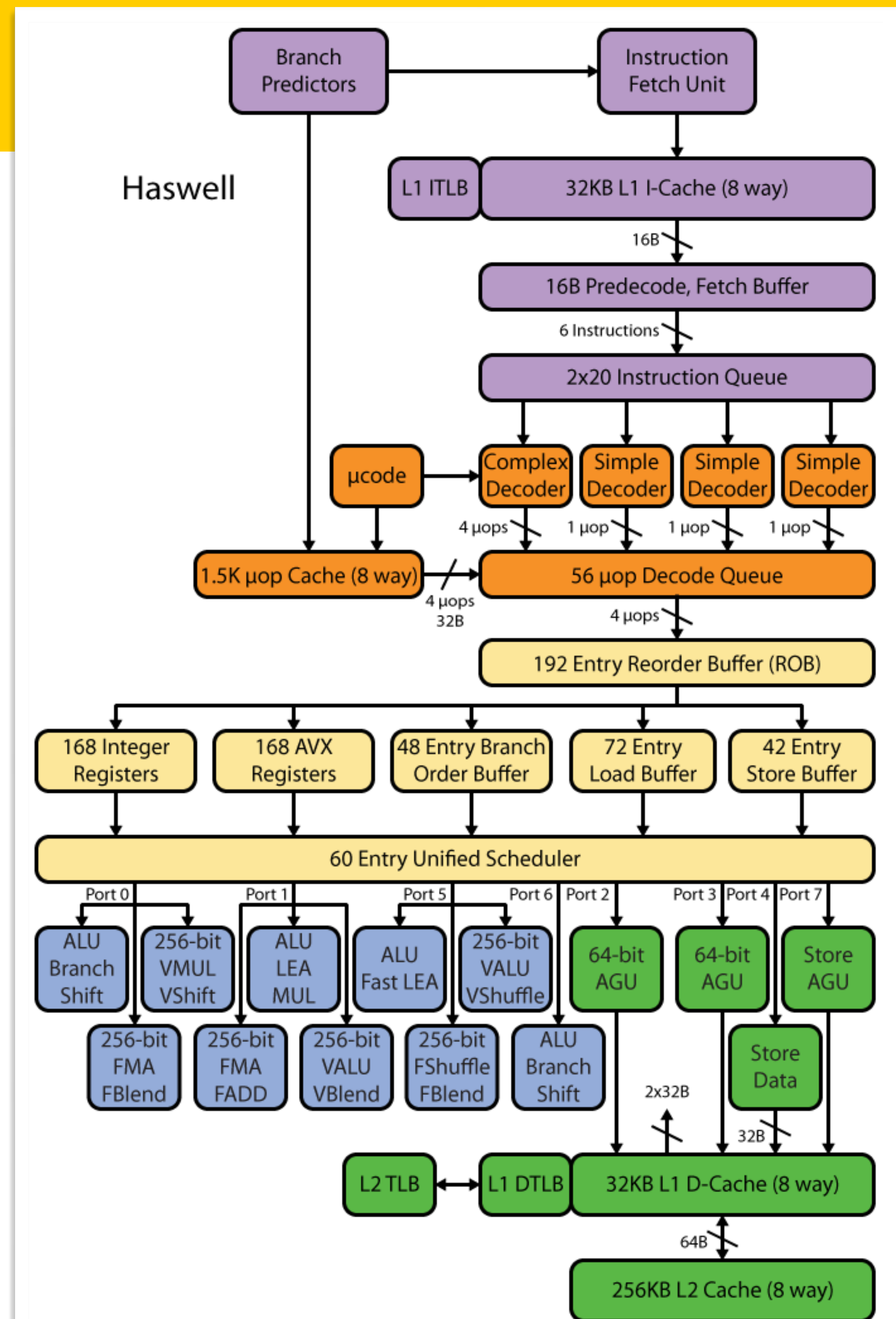
How does a processor work?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Intel Core i7 Haswell

- Up to 192 instructions in-flight
- Up to 48 predicted branches ahead
- Up to 8 instructions/cycle issued out-of-order
- About 25 pipeline stages at ~4GHz
 - Question: how far does light travel during a clock cycle?
 - How large is a motherboard?



How to make a processor faster

- Instruction pipelining
 - Independent instructions can follow each other
 - Start instruction $n+1$ after instruction n finishes the first stage
- Superscalar execution
 - Multiple execution units in parallel
 - Scheduler issues multiple instructions in the same cycle
- Out of order execution
- Simultaneous multi-threading (SMT) (a.k.a. hyper threading)
 - Scheduler issues multiple instructions in the same cycle, from different threads
- Do more per instruction

- Single-Instruction Multiple-Data (SIMD) is a kind of data parallelism
 - Amortise the control overhead over the instruction width
 - In contrast to the SIMT model of CUDA/OpenCL, the vector width is exposed directly to the programmer

SIMD

- Brief history of x86 SIMD extensions
 - MMX (Pentium): 8 x 8-bit integer operations
 - SSE (Pentium 3): 4 x 32-bit floating-point operations
 - SSE2 (Pentium 4): 2 x 64-bit operations
 - SSE3, SSSE3, SSE4.1, SSE4.2 ...
 - AVX (Sandy Bridge), AVX-2 (Haswell): 256-bit operations
 - AVX-512 (KNL, Skylake): 512-bit operations
 - AVX-VNNI (Cascade lake): Vector neural network instructions
- Available instructions depends on the target architecture!

- Neglecting SIMD is becoming more costly
 - Sandy Bridge (2009): 8-way SIMD
 - Skylake (2015): 16-way SIMD
 - Centaur CNS (2019): 4096-way SIMD

SIMD

- Example: SAXPY
 - How to vectorise:

```
float saxpy(float alpha, float x, float y) {  
    return alpha*x + y;  
}
```

```
#include <immintrin.h>
```

```
__m128 saxpy4(float alpha, __m128 xs, __m128 ys) {  
    __m128 a = _mm_set_ps1(alpha);  
    __m128 b = _mm_mul_ps(a, xs);  
    __m128 c = _mm_add_ps(b, ys);  
    return c;  
}
```

SIMD

- Example: dot product
 - How to vectorise?

```
float dotp(float xs[4], float ys[4]) {  
    float r = 0;  
    for (int i = 0; i < 4; ++i) {  
        r += xs[i] * ys[i];  
    }  
    return r;  
}
```

SIMD

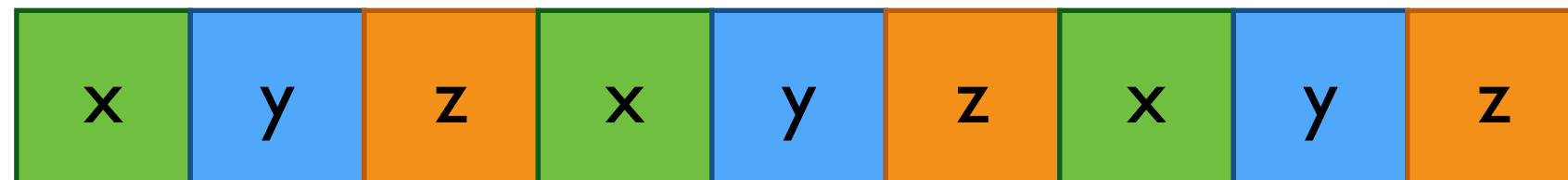
- Example: dot product
 - How to tell which one is better?
 - Benchmark! This won't tell us why; tools such as LLVM MCA can help

```
float dotp_v1(__m128 xs, __m128 ys) {  
    __m128 a = _mm_mul_ps(xs, ys);  
    __m128 b = _mm_hadd_ps(a, a);  
    __m128 c = _mm_hadd_ps(b, b);  
    float d = _mm_cvtss_f32(c);  
    return d;  
}
```

```
float dotp_v2(__m128 xs, __m128 ys) {  
    __m128 a = _mm_mul_ps(xs, ys);  
    __m128 d = _mm_shuffle_ps(a, a, 0x55);  
    __m128 e = _mm_add_ss(a, d);  
    __m128 f = _mm_unpackhi_ps(a, a);  
    __m128 g = _mm_shuffle_ps(a, a, 0xff);  
    __m128 h = _mm_add_ss(f, e);  
    __m128 i = _mm_add_ss(h, g);  
    float r = _mm_cvtss_f32(i);  
    return r;  
}
```

Array of Structures (AoS)

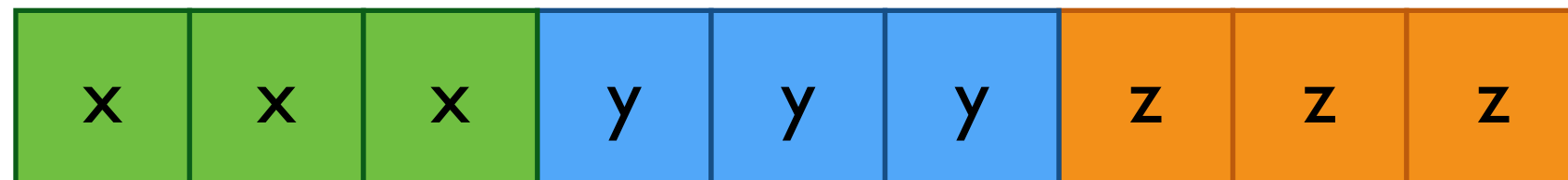
- Array of structures
 - Most logical data organisation layout
 - Extremely difficult to access memory for reads (gather) and writes (scatter)
 - Prevents efficient vectorisation
 - May lead to better cache utilisation if data is accessed randomly



```
struct Point {  
    float x,y,z;  
};  
  
void dotp_aos(  
    Point p1[128],  
    Point p2[128],  
    float rs[128]  
) {  
    for (int i = 0; i < 128; ++i) {  
        rs[i] = p1[i].x * p2[i].x  
                + p1[i].y * p2[i].y  
                + p1[i].z * p2[i].z;  
    }  
}
```

Structure of Arrays (SoA)

- Structure of arrays
 - Separate array for each field of the structure
 - Keeps memory access contiguous when vectorisation is performed over structure instances
 - Typically better for vectorisation and to avoid *false sharing*



```
struct Points {  
    float x[128], y[128], z[128];  
};  
  
void dotp_soa(  
    Points p1,  
    Points p2,  
    float rs[128]  
) {  
    for (int i = 0; i < 128; ++i) {  
        rs[i] = p1.x[i] * p2.x[i]  
            + p1.y[i] * p2.y[i]  
            + p1.z[i] * p2.z[i];  
    }  
}
```

Parallelism

A software developer's perspective

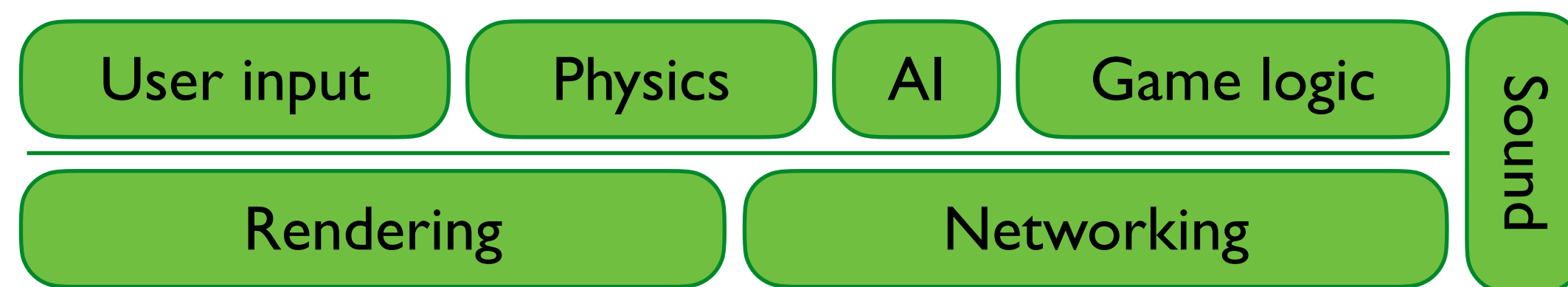
What's in a game?

- Multiple kinds of code:
 - Gameplay simulation
 - Models the state of the game world as interacting entities
 - Sound, networking, user input, etc.
 - Numeric computation
 - Physics, collision detection, path finding, scene graph traversal, etc.
 - Rendering
 - Pixel & vertex attributes; runs on the GPU



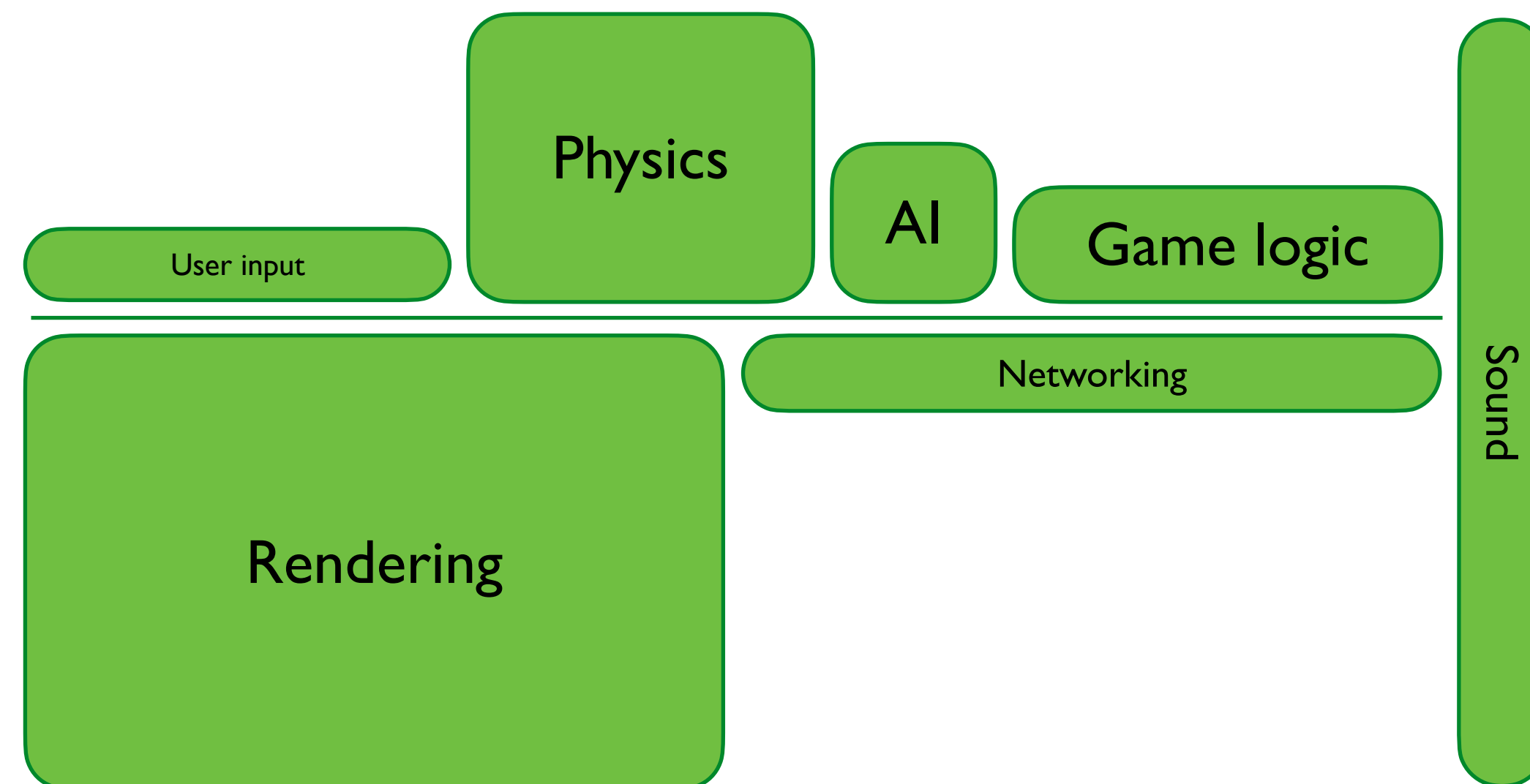
What's in a game?

- Parallel application design
 - In practice large applications consist of a mix of concurrency and parallelism
 - Parts may be run concurrently, but there are also (data) dependencies
 - Usually, individual tasks are not the same size



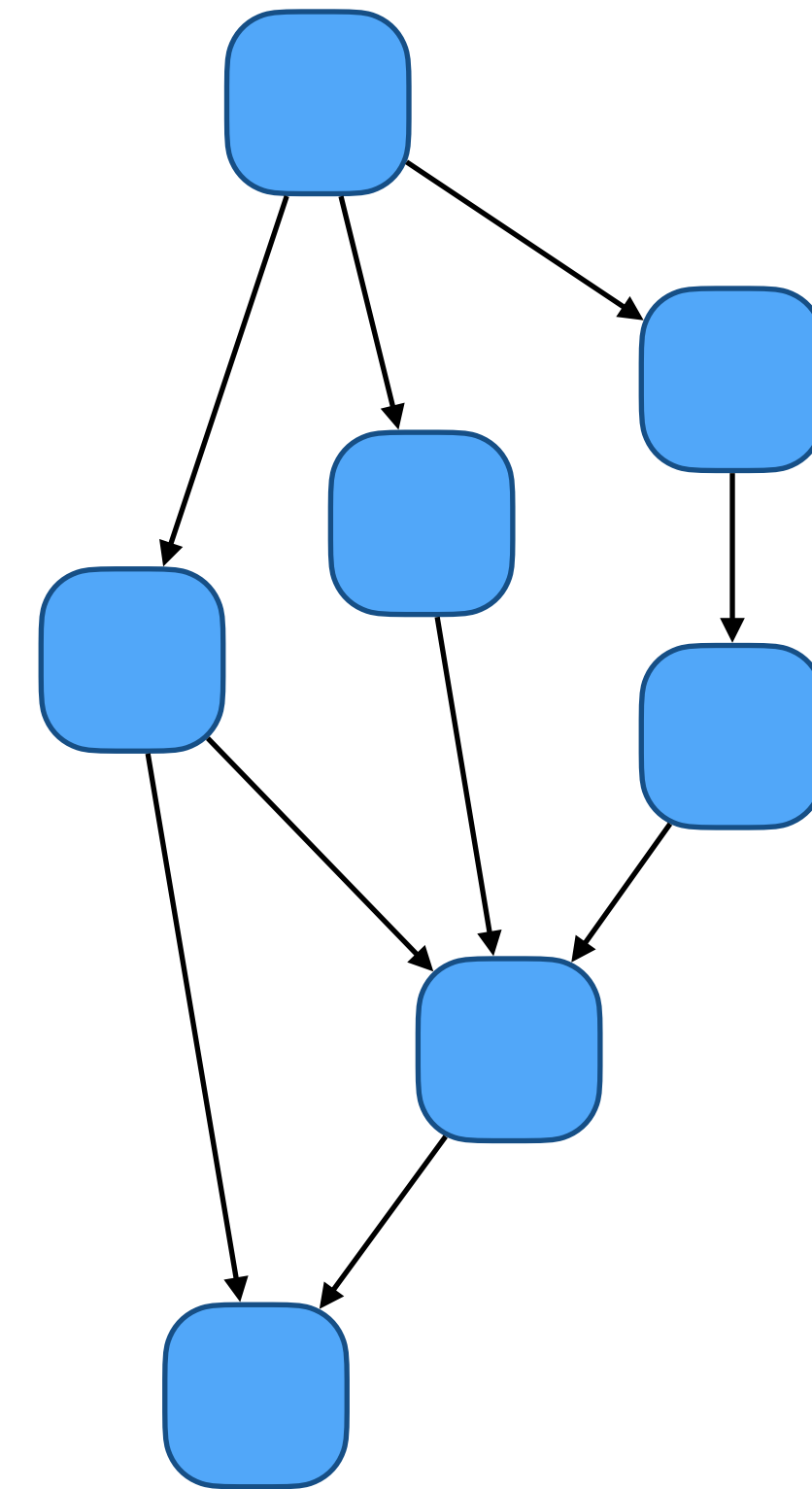
What's in a game?

- Parallel application design
 - In practice large applications consist of a mix of concurrency and parallelism
 - Parts may be run concurrently, but there are also (data) dependencies
 - Usually, individual tasks are not the same size



Task parallelism

- Task parallelism
 - Problem is broken down into separate tasks
 - Individual tasks are created and communicate/synchronise with each other
 - Task decomposition dictates scalability



Fork-Join

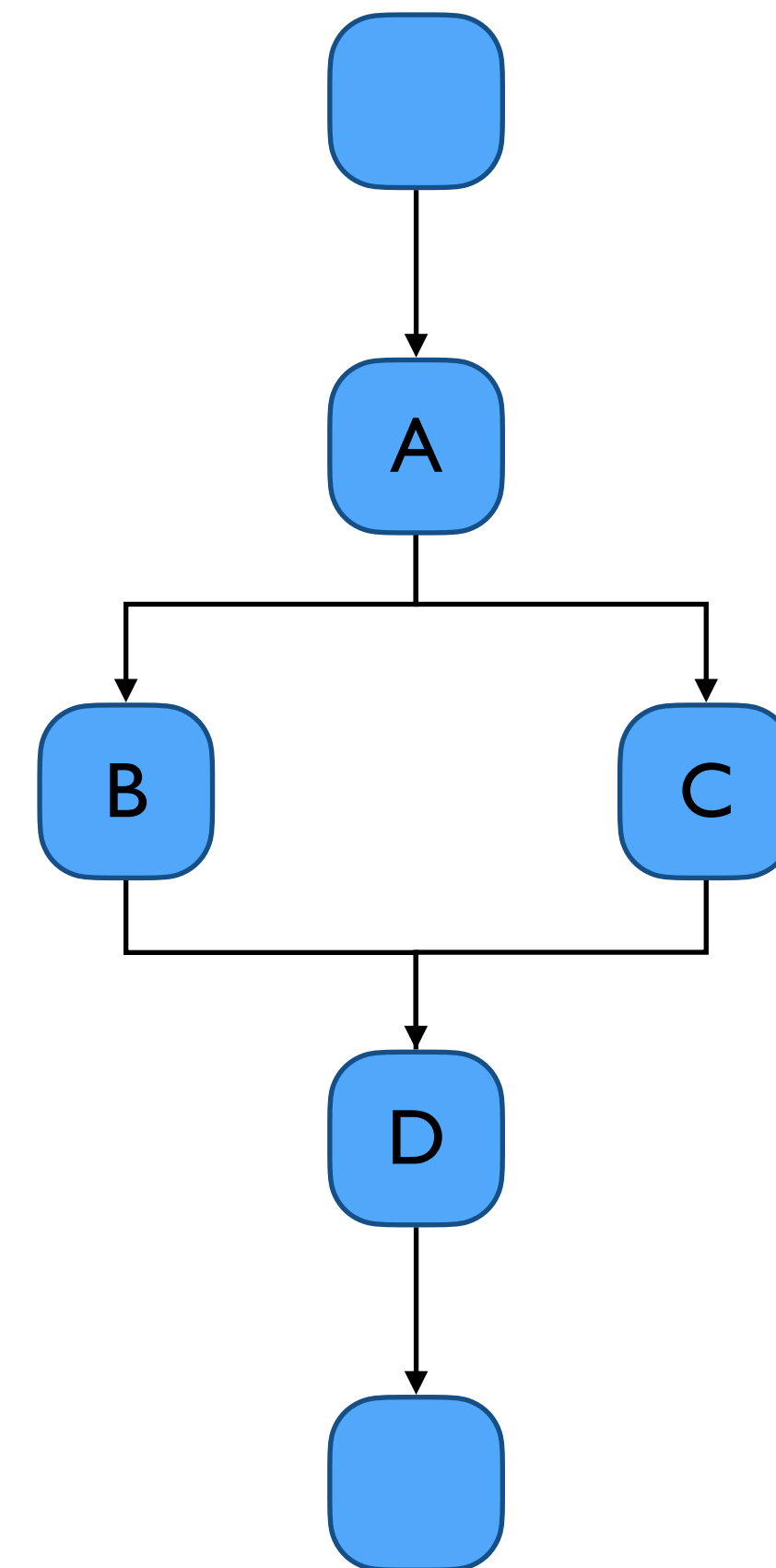
- Splits control flow into multiple forks which later rejoin
 - Can be used to implement many other patterns

```
data Async a
```

```
do
```

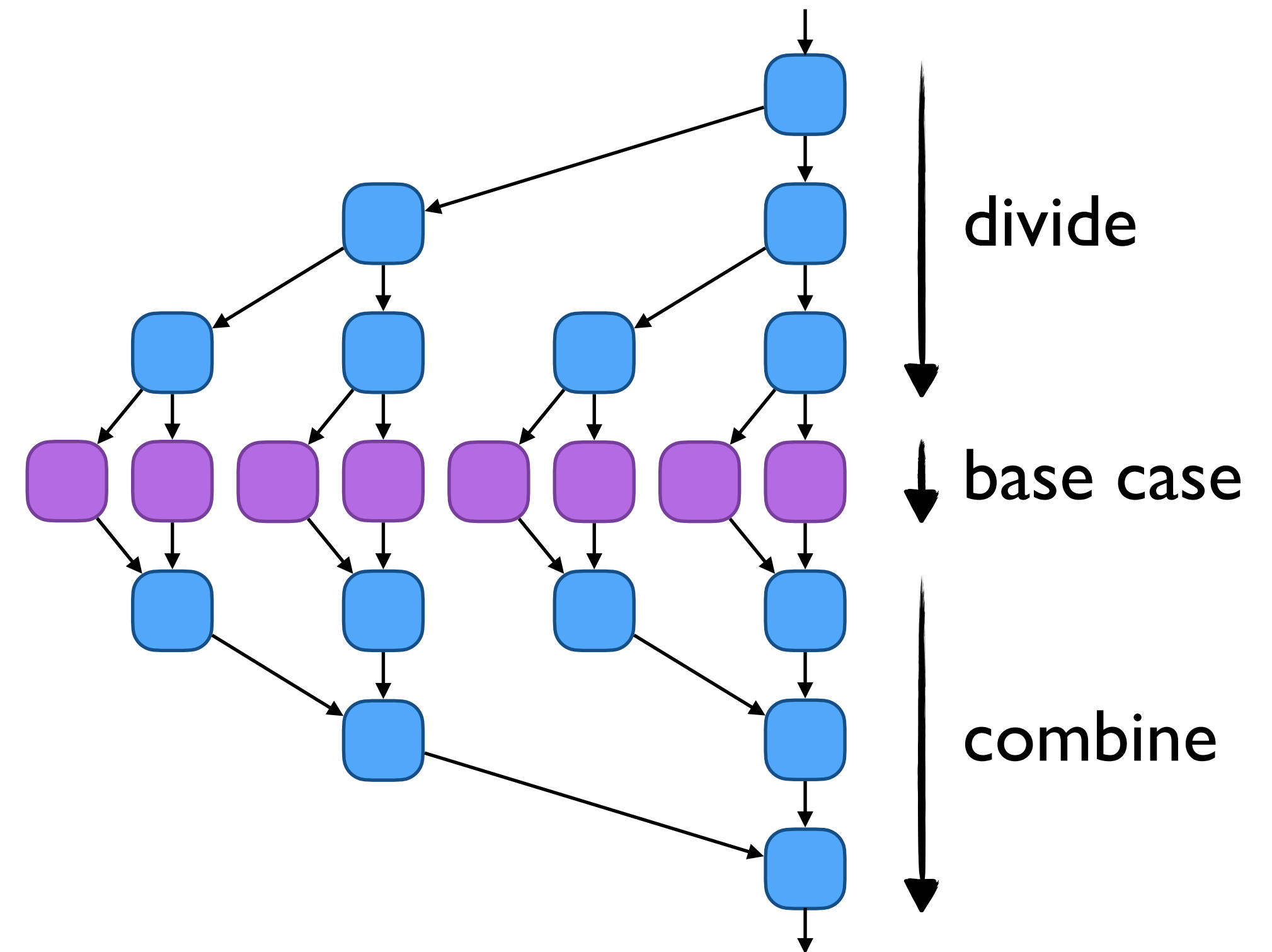
```
  a  <- compute_A  
  b' <- async (compute_B a)  
  c' <- async (compute_C a)
```

```
  b  <- wait b'  
  c  <- wait c'  
  d  <- compute_D b c
```

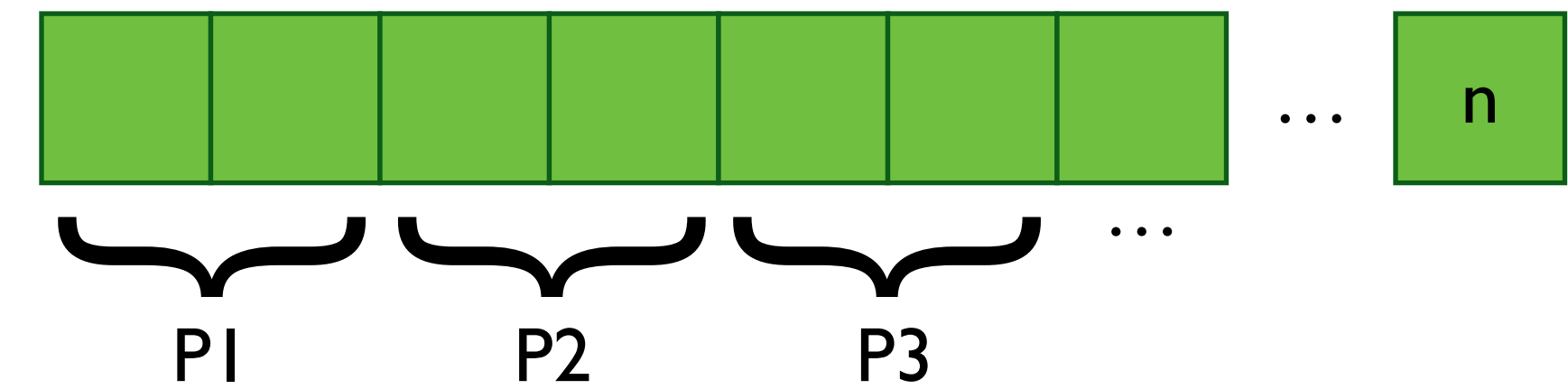


Divide-and-conquer

- Many divide-and-conquer algorithms lend themselves to fork-join parallelism
 - Sub-problems must be independent so that they can execute in parallel
 - Correct task *granularity* is vital
 - Deep enough to expose enough parallelism
 - Not so fine-grained that scheduling overheads dominate



Data parallelism



- Data parallelism
 - Problem is viewed as operations over parallel data
 - The *same* operation is applied to subsets of the data
 - Scales to the amount of data & number of processors

Considerations

Parallelism

- Improving application performance through parallelisation means:
 - Reducing the *total time* to compute a single result (latency)
 - Increasing the *rate* at which a series of results are computed (throughput)
 - Reducing the *power consumption* of a computation

Problem

- To go faster: gain from parallelisation $>$ overhead of adding it
 - Granularity: If the tasks are too small: benefit from running them in parallel $<$ task management overhead
 - Data dependencies: When one task depends on another, they must be performed sequentially

Load balancing

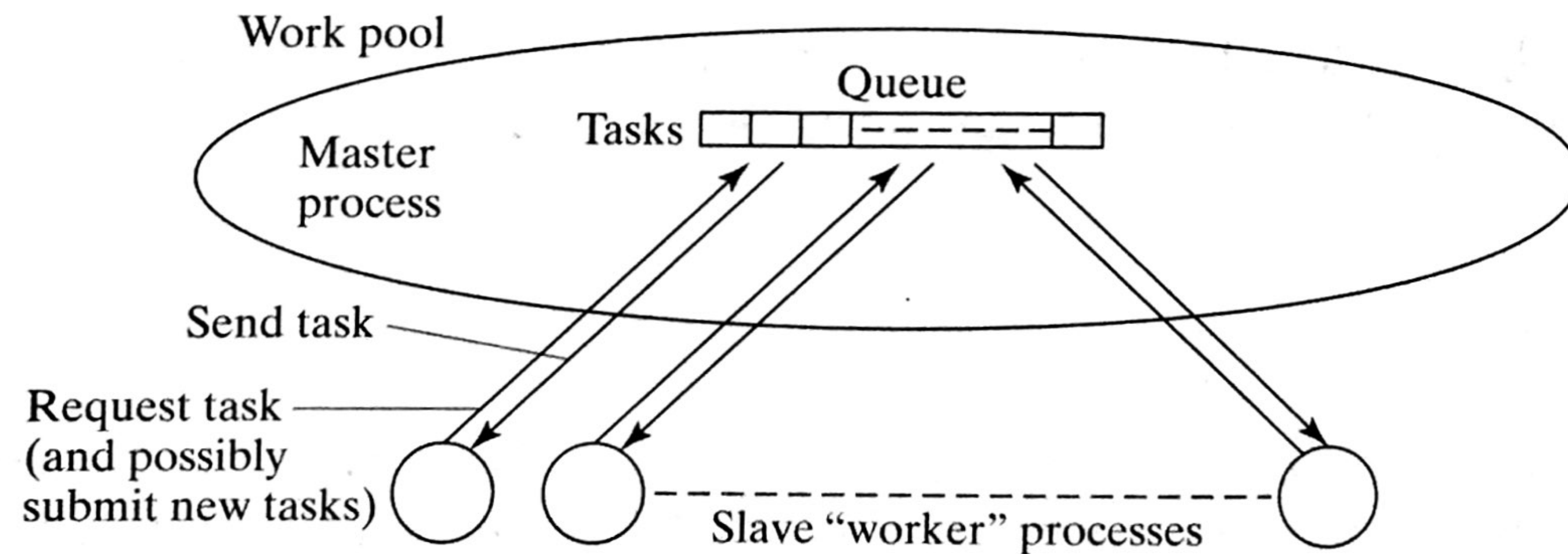
- The computation should be distributed evenly across the processors
 - Uneven distribution => maybe some processors complete their tasks before others and become idle
 - The amount of work may not be known prior to execution
 - Differences in processor speeds (e.g. noisy system, frequency boost...)

Load balancing

- *Static* load balancing can be viewed as a *scheduling* or *bin packing* problem
 - Estimate the execution time for parts of the program and their interdependencies
 - Generate a *fixed* number of equally sized tasks and distribute amongst the processors in some way (e.g. round robin, recursive bisection, random...)
 - Limitations:
 - Accurate estimates of execution time are difficult
 - Does not account for variable delays (e.g. memory access time) or number of tasks (e.g. search problems)

Load balancing

- In *dynamic* load balancing tasks are allocated to processors during execution
 - In a centralised dynamic scheme one process holds all tasks to be computed
 - Worker processes request new tasks from the *work-pool*
 - Readily applicable to divide-and-conquer problems



Speedup

- The performance improvement, or *speedup* of a parallel application, is:
 - Where T_P is the time to execute using P threads/processors

$$\text{speedup} = S_P = \frac{T_1}{T_P}$$

- The *efficiency* of the program is:

$$\text{efficiency} = \frac{S_P}{P} = \frac{T_1}{P T_P}$$

- Here, T_1 can be:
 - The parallel algorithm executed on one thread: *relative speedup*
 - An equivalent serial algorithm: *absolute speedup*

Maximum speedup

- Several factors appear as overhead in parallel computations and limit the speedup of the program
 - Periods when not all processors are performing useful work
 - Extra computations in the parallel version not appearing in the sequential version (example: recompute constants locally)
 - Communication time between processes

Amdahl

- The execution time (T_1) of a program splits into:
 - W_{ser} : time spent doing (non-parallelisable) serial work
 - W_{par} : time spent doing parallel work

$$T_P \geq W_{\text{ser}} + \frac{W_{\text{par}}}{P}$$

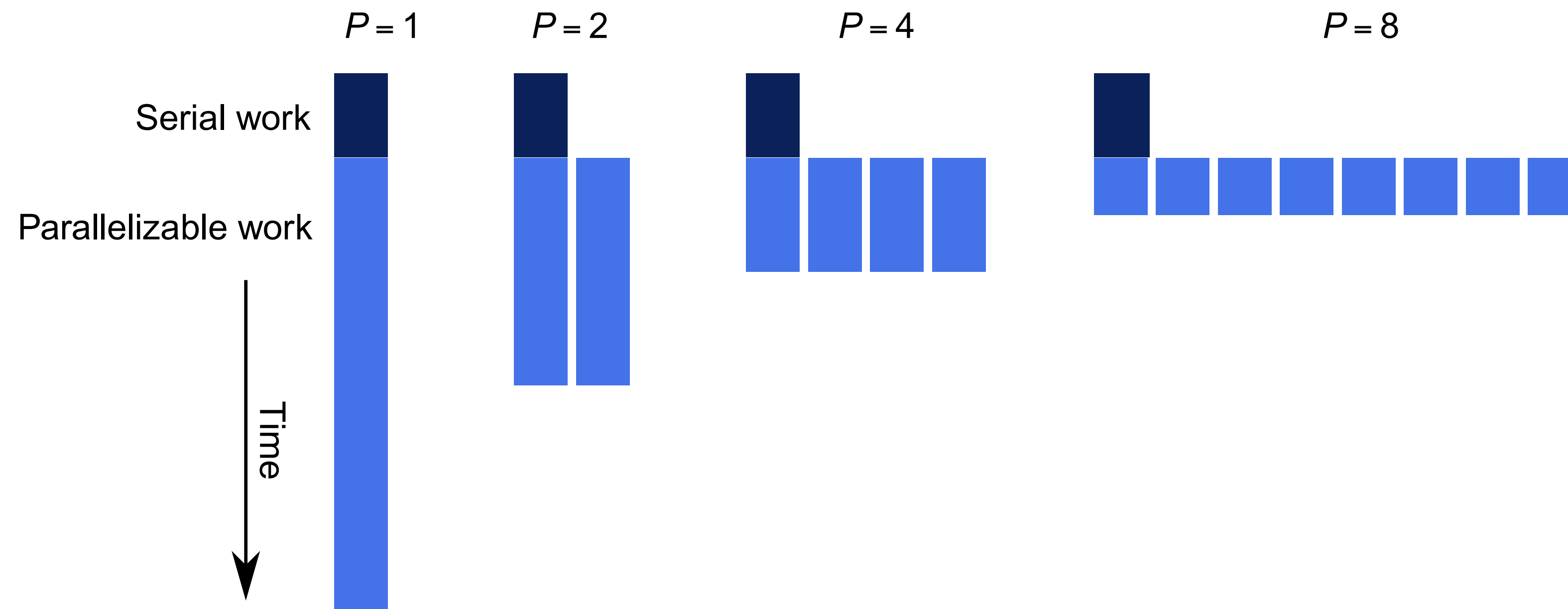
- If $f = \frac{W_{\text{ser}}}{W_{\text{ser}} + W_{\text{par}}}$ is the fraction of serial work to be performed, we get the parallel speedup:

$$S_P \leq \frac{1}{f + (1 - f)/P}$$

- This is called *Amdahl's Law*

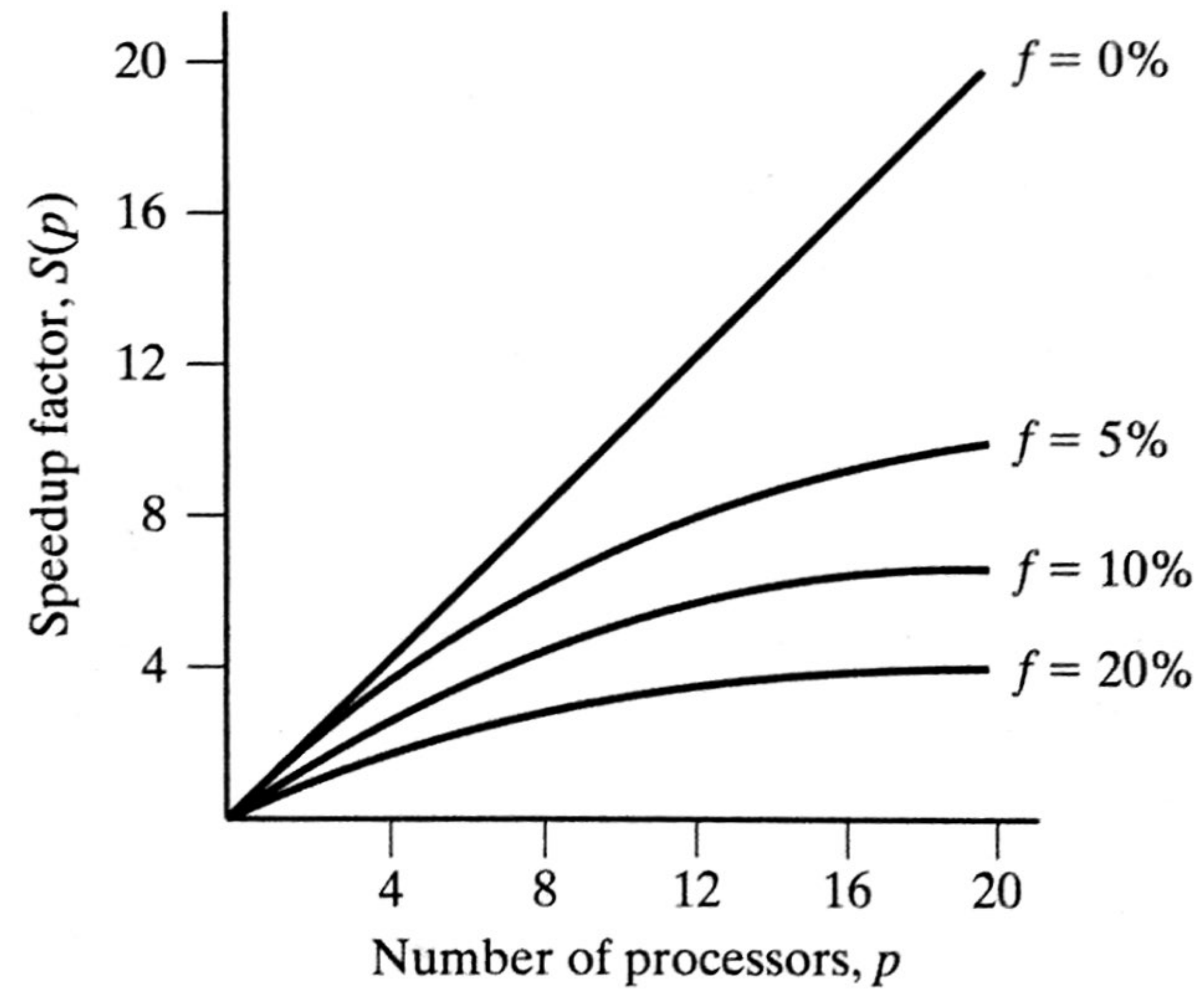
Amdahl

- The speedup bound is determined by the degree of sequential execution in the program, not the number of processors
 - *Strong scaling* (fixed-sized speedup): $\lim_{P \rightarrow \infty} S_P \leq 1/f$



Amdahl

- The serial fraction of the program limits the achievable speedup



Gustafson-Barsis

- Often the problem size can increase as the number of processes increases
 - The proportion of the serial part decreases
 - *Weak scaling* (scaled speedup): $S'_P = f + (1 - f)P$

