# Concurrent Synopsis of all Notes

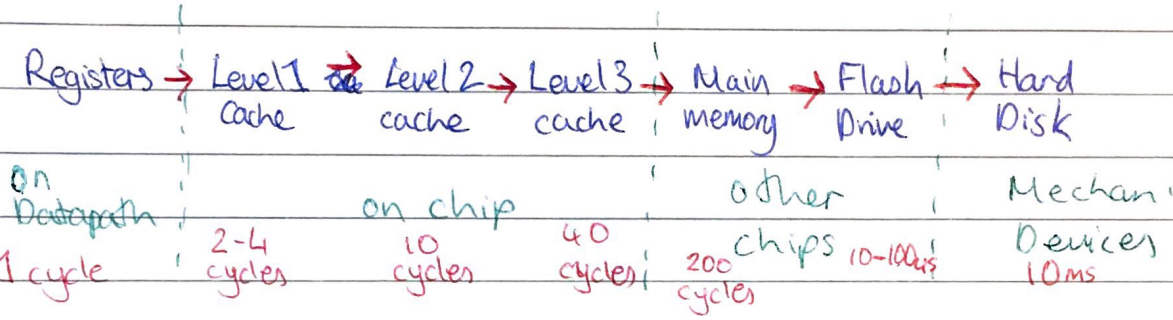
## Why write Parallel Programs?

Parallelism is tied to energy. A parallel computer with several processors is generally faster than one with one super-fast processor.

## Performance issue solutions?

- Use an efficient algorithm and Data structure efficiently
- Locality, but almost invisible to the programmer.
- Parallelism with multiple threads and Vector parallelism.

## Typical Memory Hierarchy




## Locality

Either by time or by addresses close by

## Cache Measures

Hit rate normally so high we talk about Miss rate

Avg. Access time = Hit time + Miss Rate × Miss penalty (ns or clocks)

## Miss Penalty

time to replace a block from lower level including time to replace in CPU.

Access time: time to lower level = f(latency to lower level)

## Types of Cache Misses (3 C's)

- Compulsory - first time used, not in cache
- Capacity - Cache size limited, something not used kicked off.
- Conflict - Cache not fully associative, kicked off if same cache line
- Coherency - Data written by another core/processor.

## Reduce Miss Rate

Larger block size (Compulsory)
Larger Cache size (Capacity)
Higher Associativity (Conflict)

## Reduce Miss penalty

Multilevel Caches

## Reduce Hit Time

Give reads priority over writes

## Data Structures

- Linked lists have terrible locality because each element can be anywhere in memory, replace with arrays because arrays are contiguous
- Replace Binary trees with BTrees etc.
- Standard libraries do this.

# Efficiency

## Types of Efficiency
- Running Time - CPU / Hard Disk Access Time / Network / Other I/O
- Memory - RAM / ROM / Disk / External Storage

## Costs of inefficiency
- Lost user time / Other processes
- Unusability for real-time processing
- More expensive Hardware

## Efficiency Required
- User command response: 300ms
- Music: ~~200ms~~ 20ms
- Animated Software (refresh rate): 12ms
- Improve efficiency until other components dominate
- Often commercial trade offs have to be made.

## Goals
Correctness, Clarity, Simplicity, Development effort/Cost, Maintenance effort/costs, Time-to-market.

## Why can't my Compiler do this?
Does optimise but...
- Stay within semantics of program
- Avoid potential pessimisations
- Attempts transformations that can be identified quickly and with limited memory
- Only includes optimisations used reasonably often
- May be dependencies between optimisations.

## Compiler Stumbling Blocks

Aliasing
Side effects / exceptions
Loops that execute zero times.

## Algorithms & Data Structures

Goals are simplicity, efficiency and flexibility
Problems are abstraction and abstract Data types.

## Language issues

- Pointer Aliasing (C vs. Fortran)
- Nested Objects (Java vs. C)
- Scaling in pointer arithmetic (C vs. Forth)
- Null terminated Strings in C
- "C++ is slow" (Or java or Python).

## Code Motion

Computation must have no side effects or depend on anything computed inside the loop.

```
for(...){
  ...
  a[i] = sin(x) * PI;
  ...
}
```

⟹

```
temp = sin(x) * PI;
for(..){
  ...
  a[i] = temp;
  ...
}
```

---

```
for(i=0; i<strlen(s); i++){
```

⟹

```
length = strlen(s);
for(i=0; i<length; i++){
```

## Parallelism

- Between several CPUs : Multithreading
- Between CPU and Disk: prefetching, write buffering
- Between CPU and Graphics Card : Triple buffering
- Between CPU and Memory : Prefetching
- Between Machine instructions: Instruction scheduling.
- SIMD

## Eliminate Duplicates.

```
a = Exp;          ⟹      a = Exp;
b = Exp;                  b = a;
```

as long as Exp has no side effects.

## Parallelism in a nutshell

Compile time initialisation (instead of run time)
Eliminate subexpressions by quoting other variables (above).
Pairing Computation (Division & Remainder)
Data Structure Augmentation (Danger of inconsistent Data Structures)
Lazy Evaluation (Finite state automation for regular expressions).
Packing (Data Compression/Code size/No unnecessary bytes/bits (bitfields in C, packed in Pascal)/Cache behaviour).
Interpreters/factoring (similiar code sections to functions).
Compiler Flags (gcc provides alot)

## Faster C code

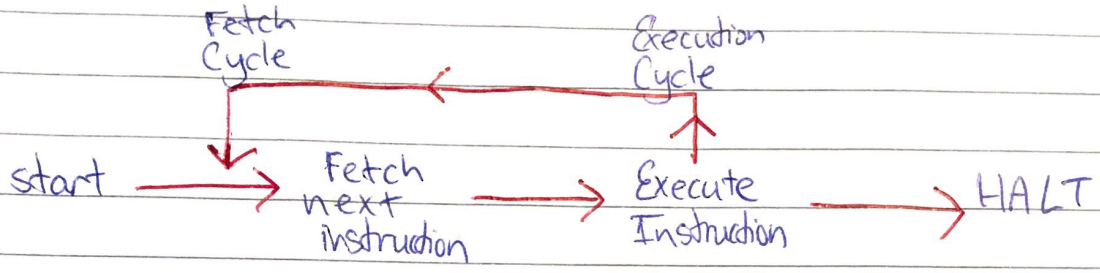
- If function only used in one file declare it "static"
- Function inlined use "inline" declaration
- Ponter aliasing, use "restrict" pointers.
- Use "register" declaration on a variable to ensure not made unavailable for register allocation.
- Copy value into a local variable if used repeatedly within a loop, becomes eligible for register allocation.

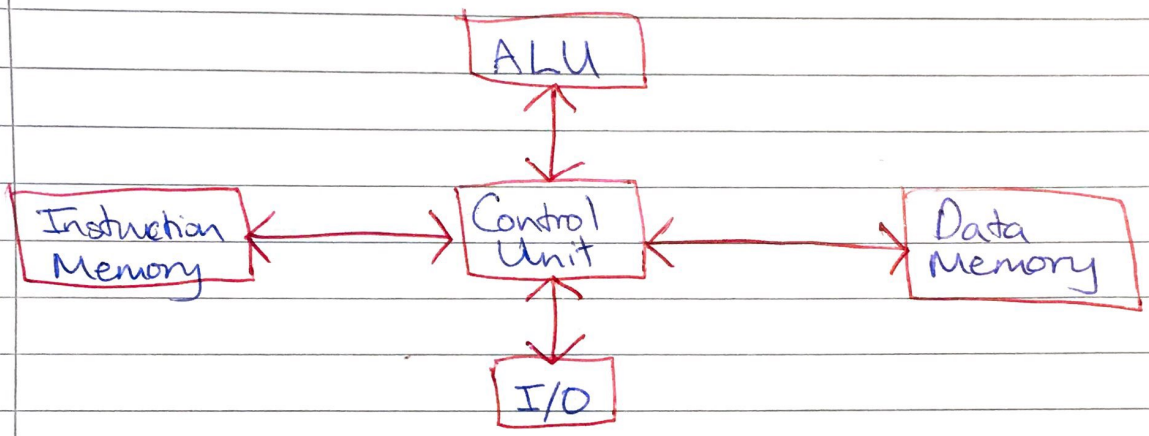
## Programming for Locality

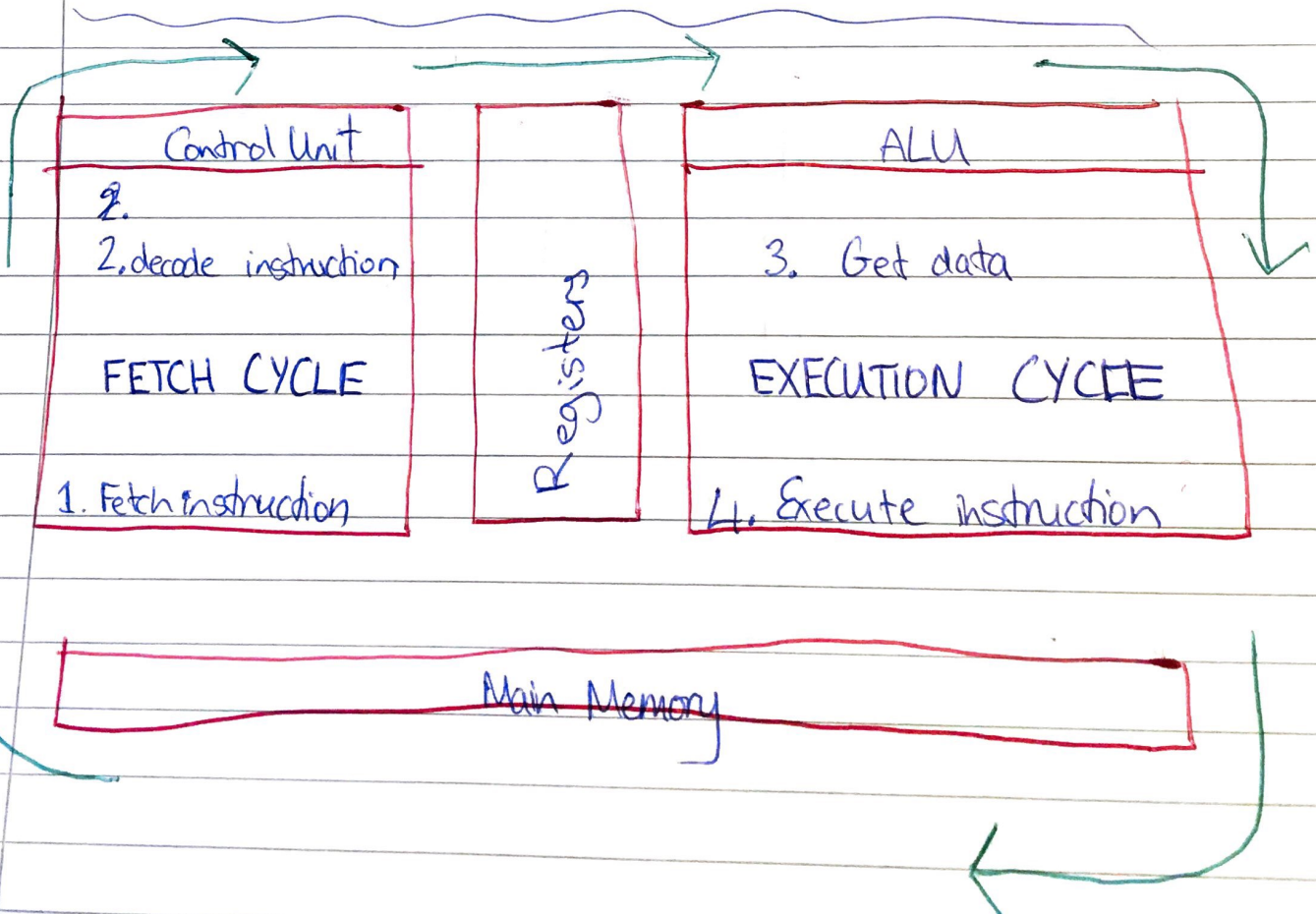
- Design data structures that maintain locality
- Arrays are good, linked lists are bad.
- Hash tables faster than Binary trees usually
- In structs, declare largest component items first
- Consider arrays of structures rather than structures of arrays.

# Flynns Taxonomy

## Computer Instruction Cycle



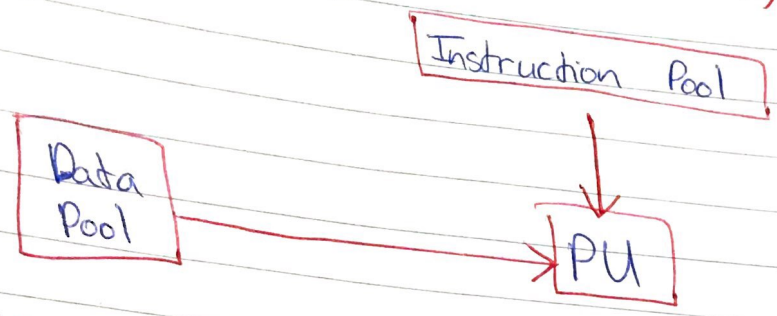



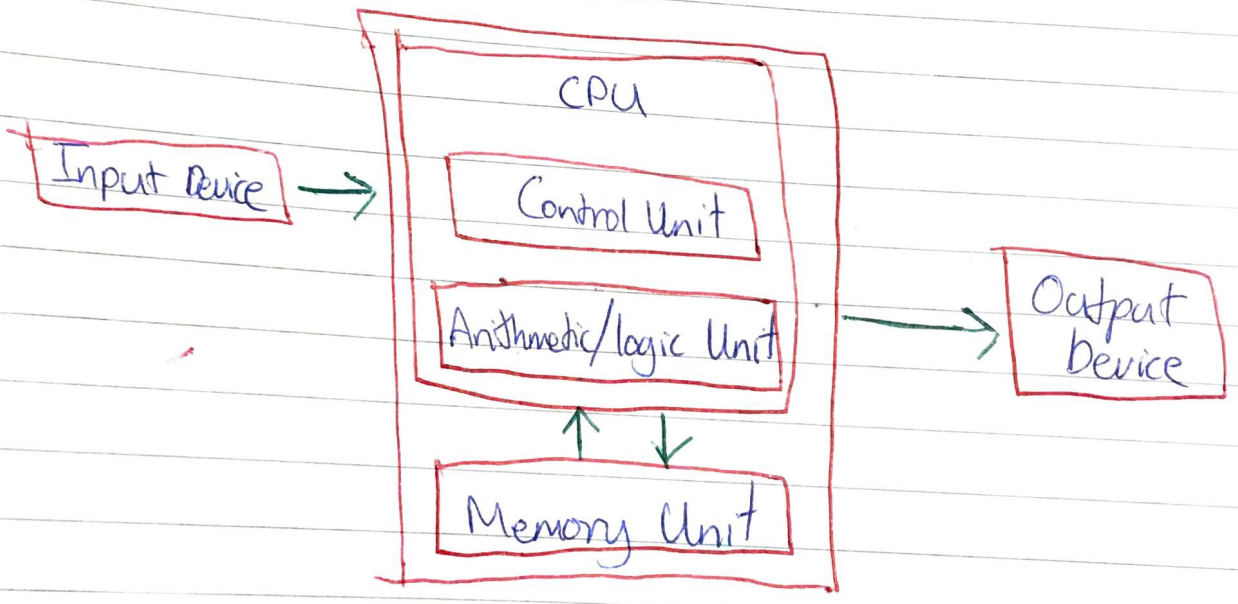
Flynns Taxonomy

How many instructions VS. How much Data can be processed Simultaneously

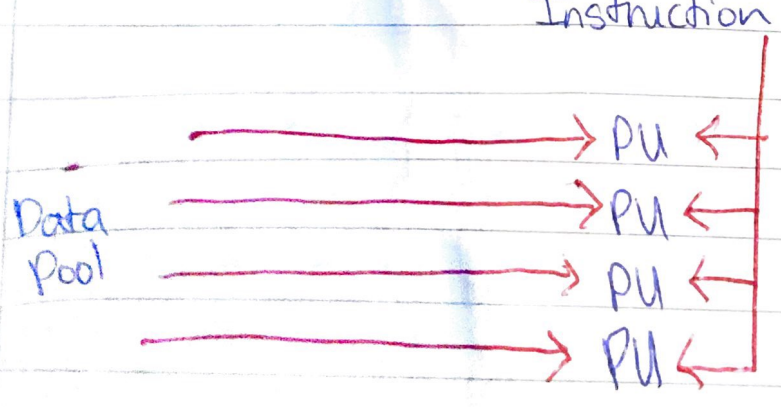
Single Instruction Single Data (SISD)




Von- Neumann Architecture



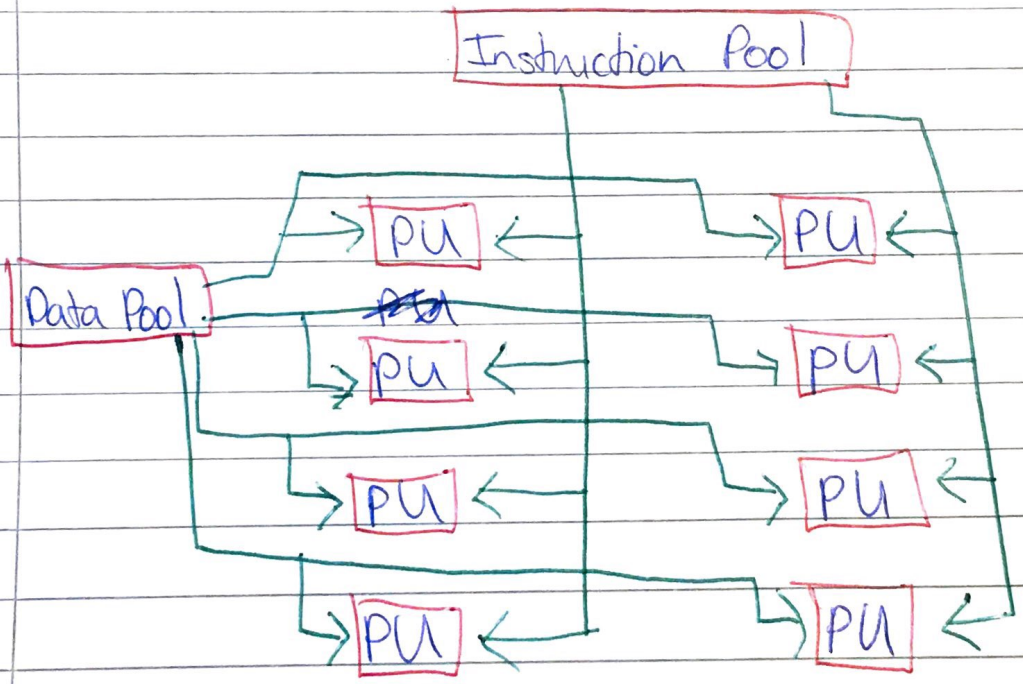

Single Instruction Multiple Data (SIMD)




- Same processing instruction runs on all processors
- Data level parallelism NOT Concurrency
- For image editing
- Multimedia Processing

- Pipelining with no real life examples
- Convolution/Matrix operations/Data sorting

## Multiple instruction Multiple Data (MIMD)




- Different instructions can run on different data
- Asynchronous, independent processing
- Shared or distributed memory
- Applications in Simulation, Emulation, CAD/CAM, Mo

# More MIMD (Important Distinction)

## SPMD

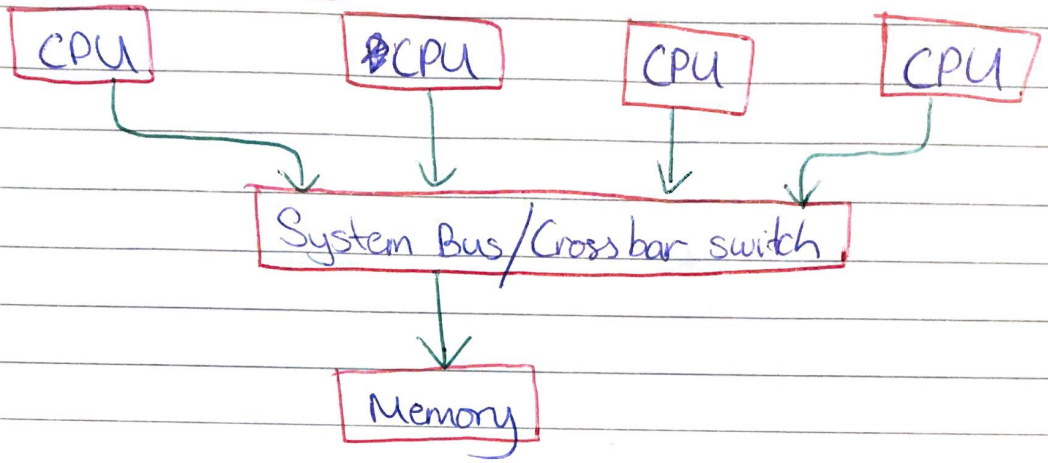
- Same program, multiple Data
- Programs executed at independent execution points
- Most Common style of parallel programming

## MPMD

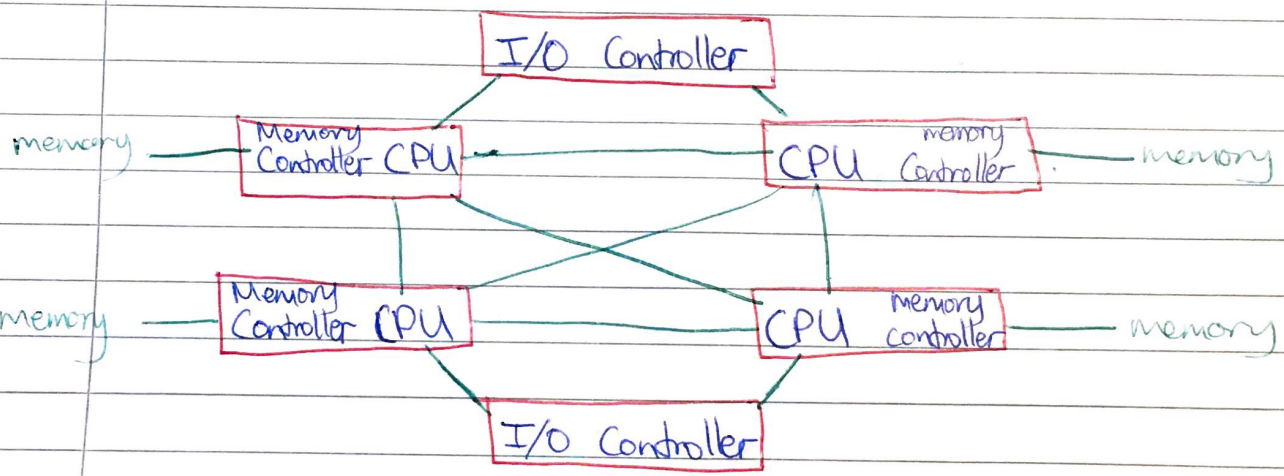
- Multiple program, multiple data
- At least 2 programs
- One program is master/Controller
- Other nodes receive program from master.

# Parllel Architectures.

## Uniform Memory Access (UMA)




## Non-Uniform Memory Access (NUMA)



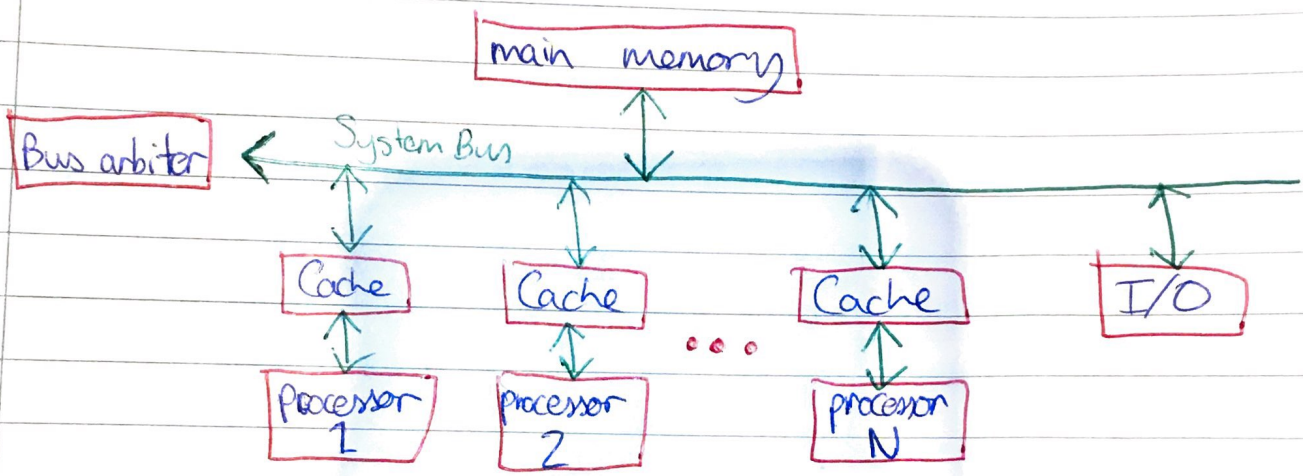

## Memory Access

| | UMA | NUMA |
|---|---|---|
| Latency | Same | Different |
| Bandwidth | Same | Different |
| Memory | Shared | Distributed |

## Symmetric VS. Asymmetric

- 2+ identical processors connected to single shared memory (SMP)
- Most multiprocessors use SMP
- For OS, all processors treated the same
- Tightly coupled, connected at bus level.
- If processors aren't treated the same, then asymmetric
- ASMP is expensive, hence rarer.

## SMP- Single multiprocessor System




## Multicore Processors

- May or may not share cache
- May implement message passing or IPC
- Cores can be connected in bus, ring, 2D Mesh, crossbar
- Homogenous or Heterogenous

## big.LITTLE ARM Architecture.

- Finer-grained control of workloads.
- Implementation in the schedule
  - clustered switching
  - In-Kernel switcher (CPU migration)
  - Heterogenous multi-processing (global task scheduling)
- Easily support non-symmetrical SoCs
- Uses all cores simultaneously to improve/provide peak performance.

## DynamIQ

- Combines big and little cores into a single, fully integrated cluster.
- Better power and memory efficiency
- 1-8 Cortex A-* CPUs in one cluster
- Great for AI and machine learning processing
- Various Configurations.

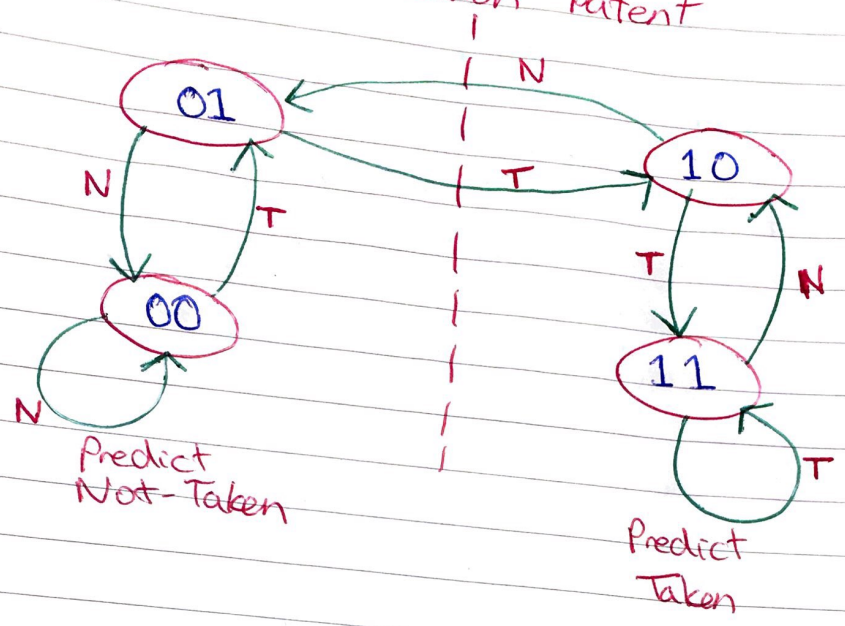
## Instruction Pipelining

- Fetch
- Decode
- Execute
- Write-Back

## Pipeline Branching

- Wasted resources and bubble in pipeline if branch not taken.
- Causes Delay in execution
- Branch Prediction used to predict what branch will be taken using Algorithm
- Very complex to execute accurately.

## Intel Branch Prediction Patent




## Superscalar

- Scalar: each instruction manipulates {1,2} data at a time
- Superscalar: Execute more than one instruction at a time
- How?: ~~Execute~~ Multiple simultaneous instructions to different execution units
- More throughput per clock cycle
- Flynn's Taxonomy
  - SISD for single core (SIMD for vector ops)
  - MIMD for multiple cores.

# Multithreading

## Flynn's Classification Scheme

- SISD - Single instruction, Single data stream (Uniprocessors)
- SIMD - Single instruction, Multiple data streams
  (single control unit broadcasting operations to multiple datapaths).
- MISD - Multiple instruction, single data (no such machine although sometimes vector machines are put in category).
- MIMD - multiple instructions, multiple data streams (aka multiprocessors (SMPs, MPPs, clusters, NOWs).

## Performance beyond single thread ILP

- Can be much higher natural parallelism in certain applications eg. databases or scientific codes.

## Thread or Data Level Parallelism

- Thread is a process with its own instructions and data where it may be a subpart of a program (thread) or an independent program (process).
- Each thread has all the state (instructions, data, PC, register state, so on) necessary to allow it to execute.
- Many kitchens each with its own boss and chef (threads)
- Data level parallelism perform identical operations on data and lots of data
- Data level is 1 kitchen, 1 boss, many chefs.

## Continuum of Granularity

- Coarse
  - each processor is more powerful
  - Usually fewer processors
  - Communication between processors more expensive
  - Tend toward MIMD
- Fine
  - each processor is less powerful
  - Usually more processors
  - Communication between processors is cheaper
  - Processors more tightly coupled
  - Tend toward SIMD

## Thread-Level Parallelism

More cost effective to implement as have to rewrite code compared to Instruction level parallelism

## Conventional Multithreading

- How does a microprocessor run multiple processes/threads at the same time
- How does a program interact with another
- What is pre-emptive multitasking vs. copperative multitasking.

## New Multithreading approach

- Multithreading: ~~need~~ multiple threads to share the functional units of 1 processor via overlapping.
- Processor must duplicate independent state of each thread eg. a separate copy of a register file, a separate PC, and a separate page table for running independent programs.
- Memory is shared through the virtual memory mechanisms, which already support multiple processes.
- HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks.

## Executing Multithreading

- When to switch between threads
  - Alternate instructions per thread (fine grain)
  - When a thread is stalled, perhaps for a MISS, another can be executed (coarse grain).

## Fine Grain MultiThreading

- Switches threads on each instruction, causing multiple instruction executions to be interleaved.
- Usually in round robin skipping stalled threads.
- CPU must be able to switch threads every clock.
- As an advantage, it can hide short and long stalls since other threads instructions will be executed instead.
- As a disadvantage, it slows down the execution of an individual thread as the round robin approach
- Used on SUNs Niagara.

## Coarse Grain MultiThreading

- Switches threads only on costly stalls like L2 cache misses
- Advantages
  - Releives need to have very fast thread switching
  - Doesn't slow down thread as only switches thread when it encounters a costly stall.
- Disadvantage
  - Hard to overcome throughput losses from shorter stalls, due to pipeline start up costs.
  - Since CPU issues instructions from 1 thread, when a stall occurs the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions complete.
- Because of this Coarse-Grain multithreading is better for reducing penalty of high-cost stalls as the start up overhead is big.
  pipeline refill $\ll$ stall time.
- Used in IBM AS/400

- Consider a 2-way CMP replacing a uniprocessor
- Run the CMP at half the uniprocessors clock speed.
  - Each request takes twice as long to process
  - But slowdown is less because request processing is likely limited by memory or disk.
  - If not much contention, overall throughput is the same.
- Half Clock rate → half voltage → quarter power per processor
  so 2x savings overall

~~Atomicity~~ Computing a sum.

Thread 1.

```
local_s1 = 0
for i=0, n/2-1
  local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1;
unlock(lk);
```

```
static int s=0
static lock lk;
```

Thread 2.

```
local_s2 = 0
for i=n/2, n-1
  local_s2 = local_s2 + f(A[i])
lock(lk);
s = s + local_s2;
unlock(lk);
```

# Intel SSE

## Flynns Taxonomy

Classifies computer architectures into four types: SISD, SIMD, MISD, MIMD.

## SIMD

Single instructions operate on multiple data streams. Examples include GPUs, modern vector processors with SIMD extensions such as SSE, Altivec.

## MMX

- Intel introduced SIMD to desktop CPUs in 1997 with MMX. It had 8 new 64-bit registers, MM0 - MM7 for existing floating point registers renamed. Couldnt interleave FP and MMX easily because they used the same registers, had to save and restore registers.
- These had integer only instructions
- Supported packed instructions (2x32bit, 4x16bit, 8x8bit).

## SSE

- 8 new 128 bit registers XMM0 - XMM7
- No overlapping of existing registers, programming was simpler and more efficient
- with EM64T/x86-64/AMD64 now 16 registers (XMM0 - XMM15)
- Supported Floating point.

## SSE 2.

- Major upgrade allowing use of integer instructions in XMM registers and more packing options (doubles)

## SSE3, 4

Minor upgrades with some notable instructions such as horizontal ops, a dot product and lots of integer instructions.

## AVX

Major upgrade adding 256-bit registers, encryption and 3-address instructions, also fused mutiply-add.

## Programming in SSE

- We use intrinsics as the compiler generally has special functions that have a specific optimisation path.
- They generally encode to a specific small list of sequential instructions.
- Speed goes up
- Ease of use goes down.
- Intrinsics a good middle ground, with modern optimising compilers we get results as good as the best hand tuned ASM.
- Code readability not compromised compared to Assembly language.
- Register management handled by compiler.

- `#include xmmintrin.h`

## Intrinsic Types

`__m128` = 4 32-bit single precision floats
`__m128i` = 4 32-bit integers
`__m128d` = 2 64-bit double precision floats.

Load Instructions.

__m128 _mm_load_ps(float *src)
load 4 floats from 16 byte aligned address

__m128 _mm_loadu_ps(float *src)
load from unaligned address (slower) (4x slower)

__m128 _mm_load1_ps(float *src)
load 1 float into 4 fields of an __m128

__m128 _mm_setr_ps(float a, float b, float c, float d)
load 4 floats from parameters into an __m128

__m128 _mm_set1_ps(float w)
load 1 float into all 4 fields of an __m128

Store Instructions

void _mm_store_ps(float *dest, __m128 src)
store 4 floats to an aligned address

void _mm_storeu_ps(float *dest, __m128 src)
store 4 floats to an unaligned address

Aligned stores/loads must operate on a 16 byte aligned address. Some malloc() implementations will provide memory allocated on the correct boundary. Your compiler may also need special command line options or #pragma directives.
Attempting to use an aligned operation on a non 16-byte boundary will result in segfault. If you must use unaligned addresses, use the unaligned intrinsics. Unaligned stores/loads are however, much slower.

## Arithmetic Instructions

__m128 _mm_add_ps (__m128 a, __m128 b)
Add corresponding floats (also "sub")

__m128 _mm_mul_ps (__m128 a, __m128 b)
Multiply corresponding floats (also "div")

__m128 _mm_min_ps (__m128 a, __m128 b)
Take corresponding min (also "max")

## Other Instructions

__m128 _mm_sqrt_ps (__m128 a)
Take square roots of 4 floats (slow like divide)

__m128 _mm_rcp_ps (__m128 a)
Compute rough (12 bit accuracy) reciprocal of 4 floats (fast as an add)

__m128 _mm_rsqrt_ps (__m128 a)
Rough (12 bit accuracy) reciprocal-square-root of all 4 floats.

## Instruction Execution Times

| Instruction | Speed |
|---|---|
| ADD | FAST |
| RCP | FAST |
| MUL | SLOW |
| DIV | SLOWER |
| SQRT | VERY SLOW |

How to speed up?

Instead of dividing many times by the same variable throughout the code, instead find the reciprocal and change the DIVS to MULS. Care must be taken when getting the reciprocal of large numbers as remember not very accurate.

Usage

```
/*  c = a + b  */
__m128  a = _mm_setr_ps(1.0f, 2.0f, 3.0f, 4.0f);
__m128  b = _mm_set1_ps(5.0f);
__m128  c = _mm_add_ps(a, b);
```

Answer /* c = {6.0f, 7.0f, 8.0f, 9.0f} */

Original C SISD code

```
#define SIZE 4096
Float vals [SIZE];
Float a, b;
int main (){
    load_vals ();
    for(int i=0; i<SIZE; i++){
        vals[i] = vals[i] * a+b;
    }
    return 0;
}
```

New SIMD Code

```
#include <xmmintrin.h>
#define SIZE 4096
 float vals [SIZE];
float a, b;
  int main (){
     load_vals();
     __m128 va = _mm_set1_ps(a); contains 4 copies of a
     __m128 vb = _mm_set1_ps (b); contains 4 copies of b

     for(int i=0; i<SIZE; i+=4){ Careful, Size must be multiple of 4
       __m128 v = _mm_load_ps (&vals[i]); Careful of alignment.
       v = _mm_mul_ps(v, va);
       v = _mm_add_ps(v, vb);
       _mm_store_ps(&vals[i], v);
     }
   return 0;
}
```

Newton-Rhapson reciprocal (NR)
One iteration is enough to increase accuracy.
Still faster than divide

$$rcp\_nr(x) = 2 * \frac{1}{x} - \left(\frac{1}{x} * (x * \frac{1}{x})\right)$$

SSE NR Reciprocal

```
__m128 rcp_nr(const __m128 &a){
    const __m128 r = _mm_rcp_ps(a);
    return _mm_sub_ps(_mm_add_ps(r,r),
                      _mm_mul_ps(_mm_mul_ps(r,a),r));
}
```

NR Reciprocal SQRT

$$\frac{1}{2} * rsqrtps(x) * (3 - x * rsqrtps(x) * rsqrtps(x))$$

```
__m128 rsqrt_nr(const __m128 &a){
    const __m128 half = _mm_set1_ps(0.5f);
    const __m128 three = _mm_set1_ps(3.0f);

    const __m128 r = _mm_rsqrt_ps(a);
    return _mm_mul_ps(_mm_mul_ps(half,r),
                      _mm_sub_ps(three,
                                 _mm_mul_ps(_mm_mul_ps(a,r), Ra0)));
}
```

## SSE bitwise AND 128 bit

```
__m128 a = _mm_set_ps(0.0f, 1.0f, 2.0f, 3.0f);
__m128 b = _mm_set_ps(3.0f, 2.0f, 1.0f, 0.0f);
__m128 c;

c = _mm_and_ps(a, mask);
```

## Bitwise Operations

| | | |
|---|---|---|
| _mm_and_ps | (__m128 a, __m128 b) | r = a AND b |
| _mm_or_ps | (__m128 a, __m128 b) | r = a OR b |
| _mm_andnot_ps | (__m128 a, __m128 b) | r = not a AND b |
| _mm_xor_ps | (__m128 a, __m128 b) | r = a XOR b |

## Comparison Operations

| | |
|---|---|
| cmpeq | = |
| cmplt | < |
| cmple | ≤ |
| cmpgt | > |
| cmpge | ≥ |
| cmpneq | != |
| cmpnlt | !< |
| cmpnle | !≤ |
| cmpngt | !> |
| cmpnge | !≥ |

## Convert Comparison Result to 4-bit Mask

__m128 _mm_movemask_ps

| | |
|---|---|
| 1111 | Comparison True for all 4 |
| 0000 | Comparison False for all 4 |
| 1100 | Comparison True for the first two |
| 1010 | Comparison True for first and third |

## Usage of Mask

```
__m128 a = _mm_set1_ps(0.0f);
__m128 b = _mm_set1_ps(1.0f);

__m128 r = _mm_cmpgt_ps(a, b);

if (_mm_movemask_ps(r) == 0xF)
    printf("a is greater than b\n");
else if (_mm_movemask_ps(r) == 0)
    printf("a is NOT greater than b\n");
else
    printf("mixed result");
```

Possible Solutions to Harness power/efficiency

```
float total1, total2, total3, total4;
float *data1, *data2, *data3, *data4;
for (i=0; i<SIZE; i++){
    total1 += data1[i];
    total2 += data2[i];
    total3 += data3[i];
    total4 += data4[i];
}
```

Becomes...

```
__m128 totals;
float *data1, *data2, *data3, *data4;
for(i=0; i<SIZE; i++){
    __m128 v = _mm_setr_ps(data1[i], data2[i],
                           data3[i], data4[i]);
    totals = _mm_add_ps (totals, v);
```

Reordered...

data1 = [d1_1 | d1_2 | ... | d1_n]

etc.

result data = [d1_1 | d2_1 | d3_1 | d4_1 | d1_2 | ...]

can now get into a SIMD more efficiently

```
__m128 totals;
float *data;
for(i=0; i<SIZE; i+=4){
    __m128 v = _mm_load_ps (&data[i]);
    totals = _mm_add_ps(totals, v);
}
```

Shuffle

```
a = _mm_set_ps (0.0, 1.0, 2.0, 3.0);
b = _mm_set_ps (4.0, 5.0, 6.0, 7.0);
__m128 c;

c = _mm_shuffle_ps(a,b,_MM_SHUFFLE(1,0,3,2);

/* c now has value {2.0, 3.0, 4.0, 5.0}
```

Horizontal ADD

```
c = _mm_hadd_ps (a,b)
```

a = | a0 | a1 | a2 | a3 |

b = | b0 | b1 | b2 | b3 |

c now has following

c = | b0+b1 | b2+b3 | a0+a1 | a2+a3 |

* Also a _mm_hsub_ps Function *

2015 Q1.(c)

This code can be vectorised as it operates on a 1D array or vector meaning its vectorisable. Load each value into an SSE vector variable and use the specialised command for add and multiplication to perform the operation

```
void add_vect(float *a, float *b, float *c, float factor){

    __m128 vfactor = _mm_set_ps (factor) ;
    for(i=0; i<1024; i+=4){
        __m128 vb = _mm_load_ps (&b[i]);
        __m128 vc = _mm_load_ps (&c[i]);

        __m128 vx = _mm_mul_ps (vc, vfactor);
        __m128 vy = _mm_add_ps (vx, vb) ;
        _mm_store_ps (&a[i], vy);
    }
}
```