

PERFORMANCE & OPTIMIZATION

Andrey Kamaev
Itseez UNN 2015

WHAT IS OPTIMIZATION?

*Finding an **alternative** with the most cost effective or highest achievable **performance** under the given **constraints**, by maximizing desired factors and minimizing undesired ones.*

WHAT IS PERFORMANCE?

- **WALL TIME**

- **POWER**

- **ANYTHING ELSE?**

- ~~Throughput~~

- ~~Scalability~~

- ~~Development cost~~

- ~~Maintenance cost~~

- ~~etc~~

WHAT YOU NEED TO KNOW TO OPTIMIZE THE CODE?

- **ALGORITHMS**

$O(n \cdot \log(n))$ or $O(n^2)$

- **HARDWARE**

architecture and micro-architecture

- **COMPILER**

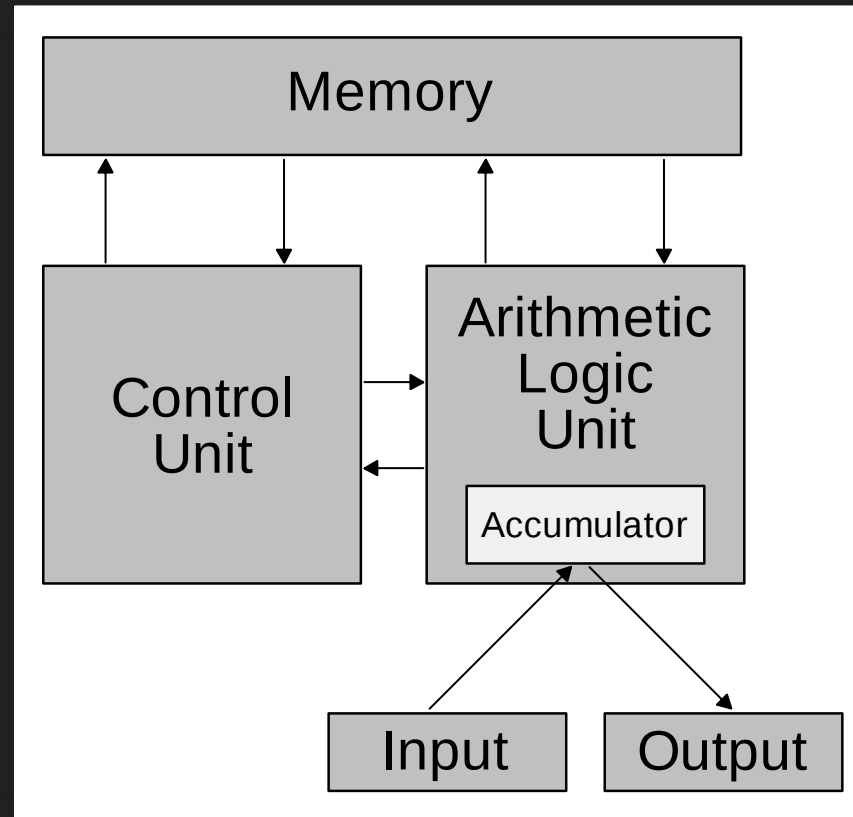
DEFINITION OF “ARCHITECTURE”

- The **Architecture** is the contract between the **Hardware** and the **Software**
 - Confers rights and responsibilities to both the Hardware and the Software
 - **MUCH** more than just the instruction set

- The **Architecture** distinguishes between:
 - Architected behaviors:
 - Must be obeyed
 - May be just the limits of behavior rather than specific behaviors
 - Implementation specific behaviors – that expose the **micro-architecture**
 - Certain areas are declared implementation specific.
E.g.:
 - Power-down
 - Cache and TLB Lockdown
 - Details of the Performance Monitors

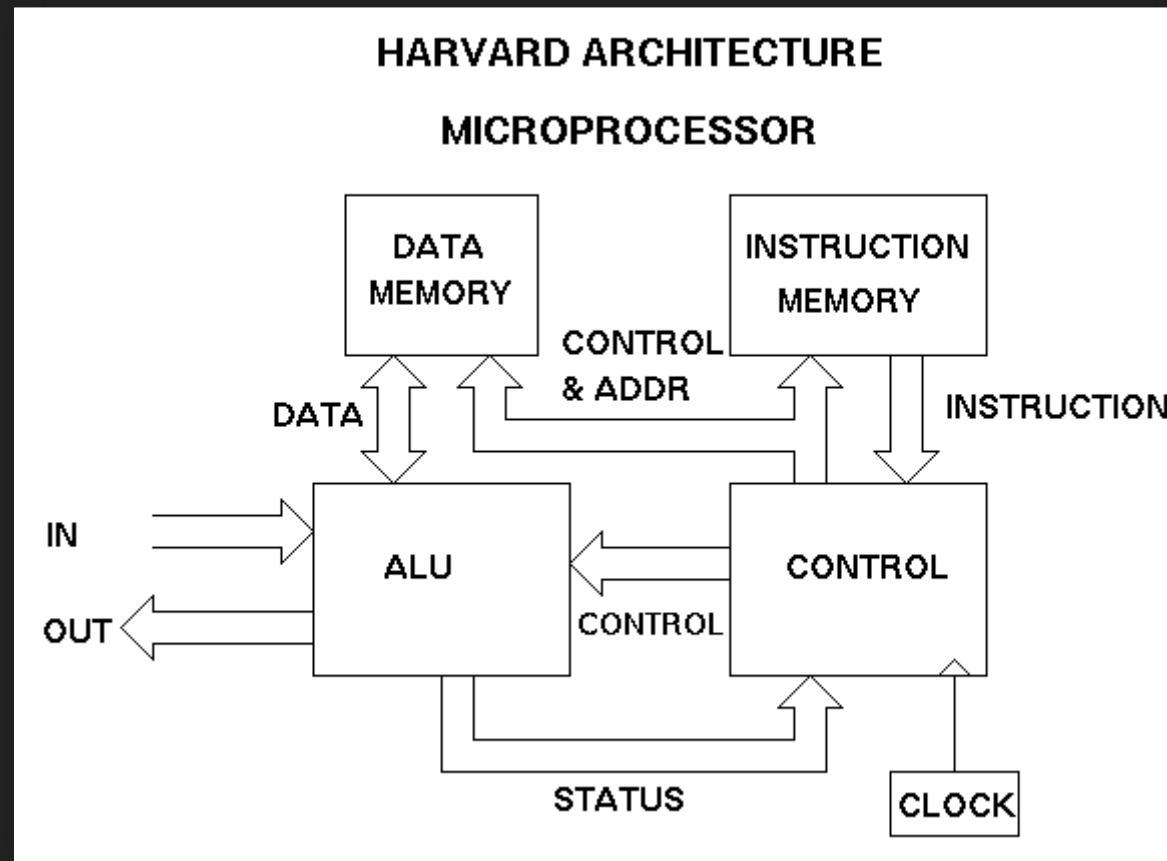
- Code obeying the **architected behaviors** is **portable** across implementations
 - Reliance on implementation specific behaviors gives no such guarantee
- **Architecture** is different from **micro-architecture**
 - **What** vs **How**

VON NEUMANN ARCHITECTURE



single memory for both instructions and data

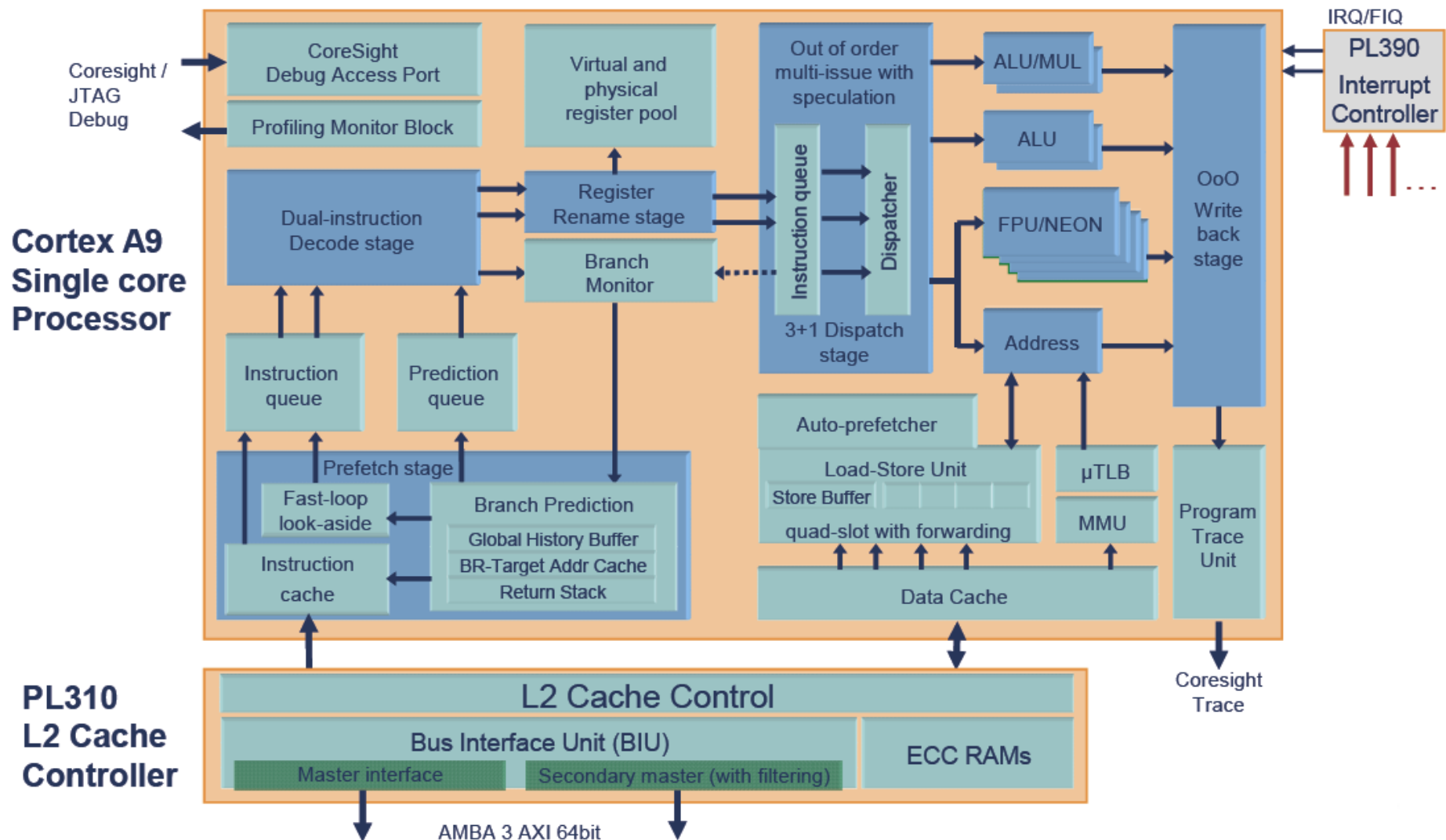
HARVARD ARCHITECTURE



separate instruction and data memory

MODIFIED HARVARD ARCHITECTURE

Cortex A9 Microarchitecture (single core variant)



CISC/RISC

CISC

Complex Instruction Set Computer

Emphasis on hardware

Includes multi-clock
complex instructions

Memory-to-memory: "LOAD"
and "STORE" incorporated in
instructions

Small code sizes, high cycles
per second

Transistors used for storing
complex instructions

RISC

Reduced Instruction Set Computer

Emphasis on software

Single-clock, reduced
instruction only

Register to register: "LOAD"
and "STORE" are
independent instructions

Low cycles per second, large
code sizes

Spends more transistors on
memory registers

ISAs (still) USED TODAY

CISC

- **x86** = IA-32 (Intel)
- **x86_64** = Intel64 = amd64 (AMD/Intel)
- s390 (IBM Mainframes)

RISC

- MIPS, SPARC, PowerPC, **ARM**

VLIW (EPIC)

- Itanium (Intel)

Old ISAs

- 680x0 (old CISC, Motorola)
- PA-RISC, Alpha (old RISC)

ARM CORTEX-M INSTRUCTION SET

| | | | | | | | |
|-------|-------|-------|-------|-------|-------|------|------|
| VABS | VADD | VCMP | VCMP | VCVT | VCVTR | VDIV | VLDH |
| VLDR | VMLA | VMLS | VMOV | VMRS | VMSR | VMUL | VNEG |
| VNMLA | VMMLS | VNMUL | VPOP | VPUSH | VSQRT | VSTM | VSTR |
| VSUB | VFMA | VFMS | VFNMA | VFNMS | | | |

Cortex-M4 FPU

| | | | | | | | |
|--------|--------|---------|---------|---------|---------|---------|---------|
| PKH | QADD | QADD16 | QADD8 | QASX | QDADD | QDSUB | QSAX |
| QSUB | QSUB16 | QSUB8 | SADD16 | SADD8 | SASX | SEL | SHADD16 |
| SHADD8 | SHASX | SHSAX | SHSUB16 | SHSUB8 | SMLABB | SMLABT | SMLATB |
| SMLATT | SMLAD | SMLALBB | SMLALBT | SMLALTB | SMLALTT | SMLALD | SMLAWB |
| SMLAWT | SMLS | SMLS | SMMLA | SMMLS | SMMUL | SMUAD | SMULBB |
| | | | | | | SMULBT | SMULTT |
| | | | | | | SMULTB | SMULWT |
| | | | | | | SMULWB | SMUSD |
| | | | | | | SSAT16 | SSAX |
| | | | | | | SSUB16 | SSUB8 |
| | | | | | | SXTAB | SXTAB16 |
| | | | | | | SXTAH | SXTB16 |
| | | | | | | UADD16 | UADD8 |
| | | | | | | UASX | UHADD16 |
| | | | | | | UHADD8 | UHASX |
| | | | | | | UHSAX | UHSUB16 |
| | | | | | | UHSUB8 | UMAAL |
| | | | | | | UQADD16 | UQADD8 |
| | | | | | | UQASX | UQSAX |
| | | | | | | UQSUB16 | UQSUB8 |
| | | | | | | USAD8 | USADA8 |
| | | | | | | USAT16 | USAX |
| | | | | | | USUB16 | USUB8 |
| | | | | | | UXTAB | UXTAB16 |
| | | | | | | UXTAH | UXTB16 |

Cortex-M0/M1

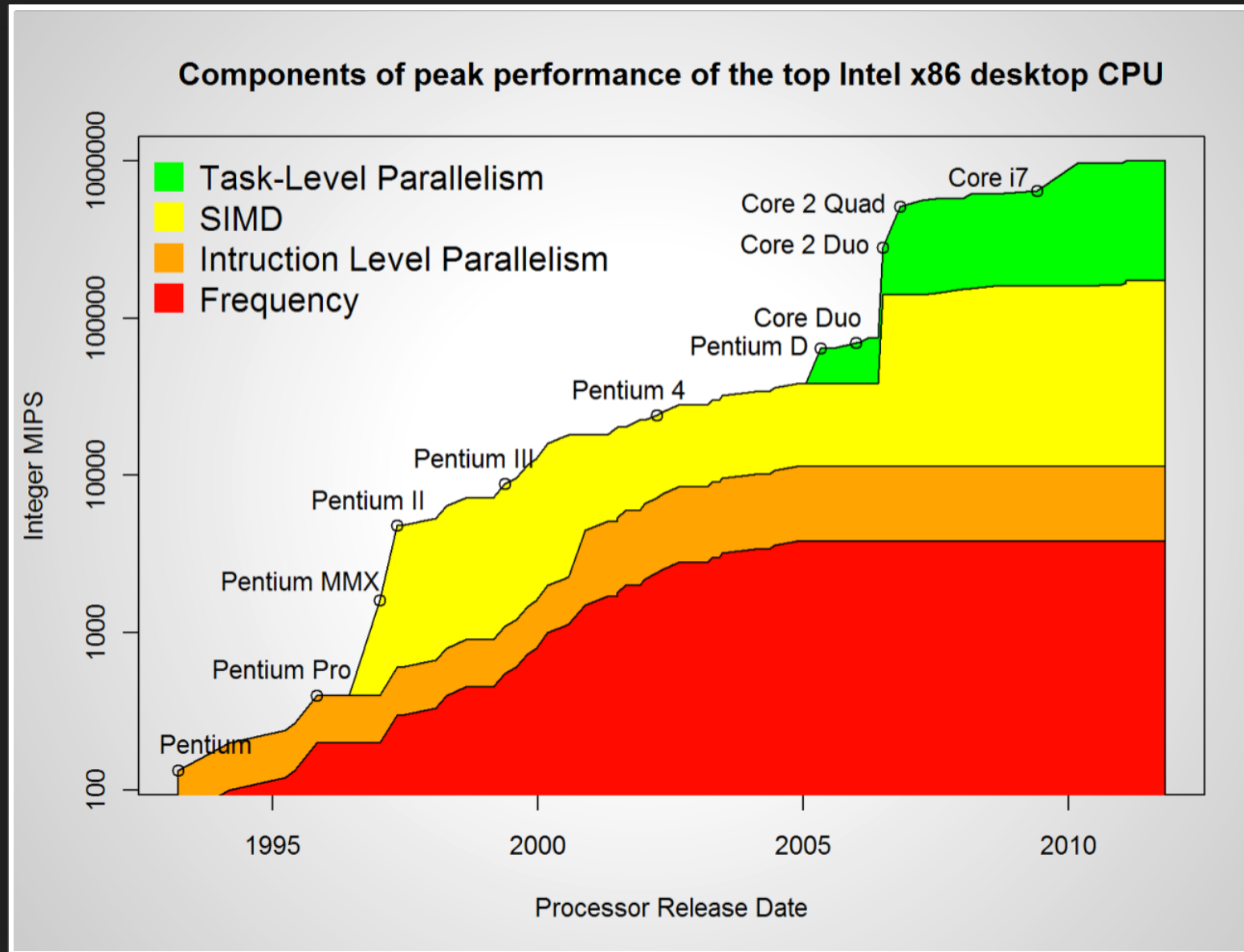
| | | | | |
|-------|-------|------|-------|-------|
| BKPT | BLX | ADC | ADD | ADR |
| BX | CP5 | AND | ASR | B |
| | | | BL | BIC |
| DSB | | CMN | CMP | EOR |
| ISB | | LDR | LDRB | LDM |
| MRS | | LDRH | LDRSB | LDRSH |
| MSR | | LSL | LSR | MOV |
| NOP | REV | MUL | MYN | ORR |
| REV16 | REVSH | POP | PUSH | ROR |
| SEV | SXTB | RSB | SBC | STM |
| SXTH | UXTB | STR | STRB | STRH |
| UXTH | WFE | SUB | SVC | TST |
| WFI | YIELD | | | |

Cortex-M3

| | | | | | |
|--------|--------|--------|-------|--------|--------|
| ADC | ADD | ADR | AND | ASR | B |
| CLZ | BFC | BF1 | BIC | CDP | CLREX |
| CBNZ | CBZ | CMN | CMP | DBG | EOR |
| LDMIA | LDMDB | LDR | LDRB | LDRBT | LDRD |
| LDREX | LDREXB | LDREXH | LDRH | LDRHT | LDRSB |
| LDRSBT | LDRSHT | LDRSH | LDRT | MCR | LSL |
| LSR | MCRR | MLS | MLA | MOV | MOVT |
| MRC | MRRC | MUL | MYN | NOP | ORN |
| ORR | PLD | PLDW | PLI | POP | PUSH |
| RBIT | REV | REV16 | REVSH | ROR | RRX |
| | | | RSB | SBC | SBCFX |
| | | | SDIV | SEV | SMLAL |
| | | | SMULL | SSAT | STC |
| | | | STMIA | STMDB | STR |
| | | | STRB | STRBT | STRD |
| | | | STREX | STREXB | STREXH |
| | | | STRH | STRHT | STRT |
| | | | SUB | SXTB | SXTH |
| | | | TBB | TBH | TEQ |
| | | | TST | UBFX | UDIV |
| | | | UMLAL | UMULL | USAT |
| | | | UXTB | UXTH | WFE |
| | | | WFI | YIELD | IT |

Cortex-M4

WHERE TO GET PERFORMANCE?

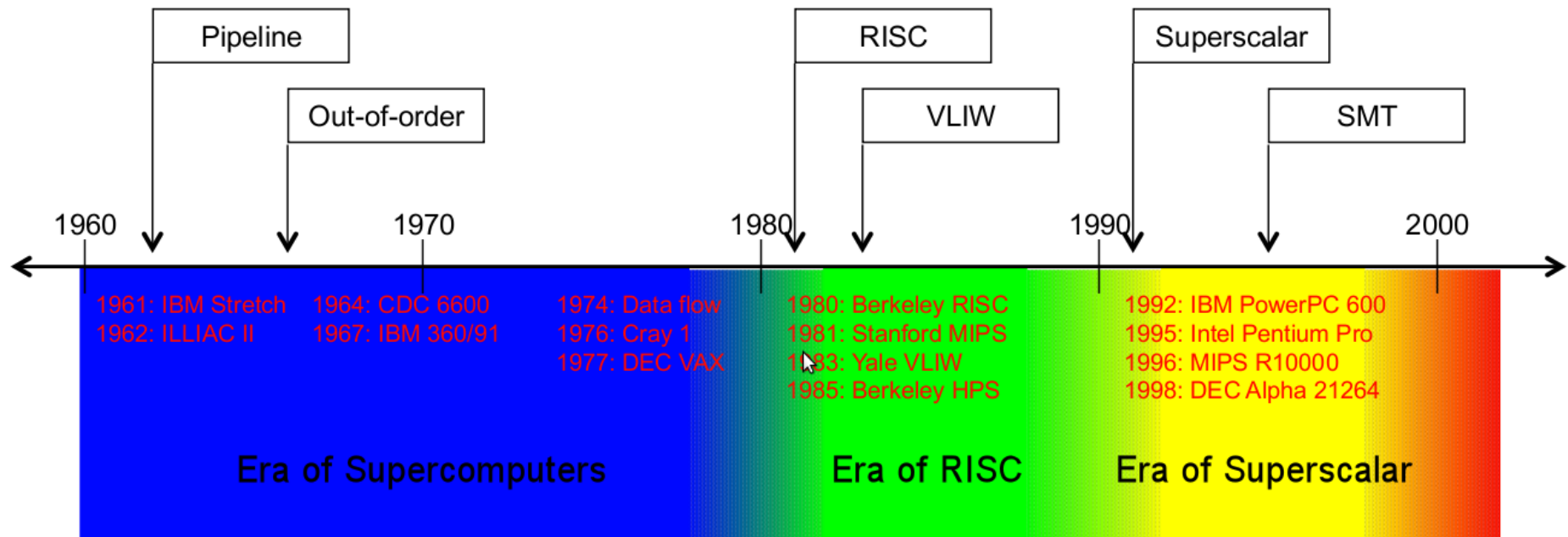


SISD - SINGLE INSTRUCTION SINGLE DATA

- Higher clock rate
- Pipelined execution (1958, 1970's)
- Superscalar execution (1965, 1988)

MIMD - MULTIPLE INSTRUCTION MULTIPLE DATA

- VLIW (1980's)
- SIMD (1970's, 1996)
- SMT - Simultaneous MultiThreading (1968, 2002)
- SMP - Symmetric MultiProcessing (1962, 2006)
- AMP - Asymmetric MultiProcessing (1970, 2013)

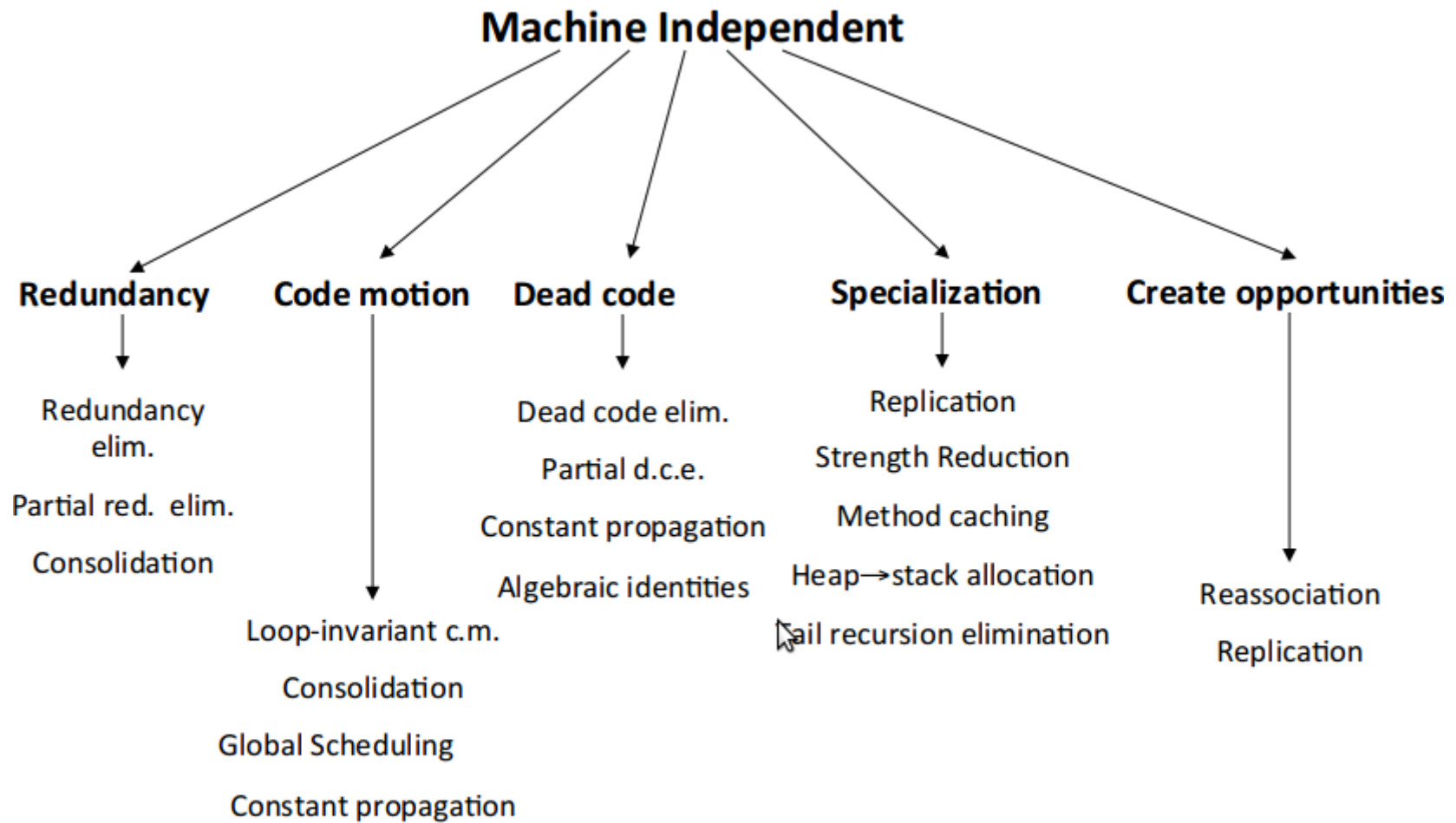


UNDERSTANDING COMPILERS

- Compiler has limited understanding of the program's behavior and the environment in which it will be used
 - Most analysis is performed only within procedures
 - Most analysis is based only on static information
- Compilers emphasize **correctness** rather than **performance**
 - Must not cause any change in program behavior under any possible condition
- On well recognized constructs, compilers will usually do better than the developer
- Often, by simply slightly reorganizing existing code, it is possible to improve both code size and speed

CLASSES OF COMPILER OPTIMIZATIONS

- High-level optimizations
 - Loop optimizations
 - Interprocedural optimization
- Global optimizations
- Local optimizations
- Processor dependent optimizations
 - Register allocation
 - Peephole optimizations



*From §6 of Cooper, McKinley, & Torczon
And Chapter 10 of EaC2e*

Machine Dependent

```
graph TD; MD[Machine Dependent] --> HL[Hide latency]; MD --> MR[Manage resources]; MD --> SF[Special features]; HL --> S1[Scheduling]; HL --> B1[Blocking references]; HL --> P1[Prefetching]; HL --> C1[Code layout]; HL --> D1[Data packing]; MR --> A1[Allocate registers, tlb slots]; MR --> S2[Schedule]; MR --> DP1[Data packing]; MR --> CL1[Coloring memory locations]; SF --> IS[Instruction selection]; SF --> PO[Peephole optimization];
```

Hide latency

Scheduling
Blocking references
Prefetching
Code layout
Data packing

Manage resources

Allocate (registers, tlb slots)
Schedule
Data packing
Coloring memory locations

Special features

Instruction selection
Peephole optimization

GCC -O3 [-mtune=generic -march=x86-64] (ver. 4.7.1)

-falign-labels -fasynchronous-unwind-tables -fauto-inc-dec -fbranch-count-reg -fcaller-saves -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers -fcrossjumping -fcse-follow-jumps -fdebug-types-section -fdefer-pop -fdelete-null-pointer-checks -fdevirtualize -fdwarf2-cfi-asm -fearly-inlining -feliminate-unused-debug-types -fexpensive-optimizations -fforward-propagate -ffunction-cse -fgcse -fgcse-after-reload -fgcse-lm -fgnu-runtime -fguess-branch-probability -fident -fif-conversion -fif-conversion2 -findirect-inlining -finline -finline-atomics -finline-functions -finline-functions-called-once -finline-small-functions -fipa-cp -fipa-cp-clone -fipa-profile -fipa-pure-const -fipa-reference -fipa-sra -fira-share-save-slots -fira-share-spill-slots -fivopts -fkeep-static-consts -fleading-underscore -fmath-errno -fmerge-constants -fmerge-debug-strings -fmove-loop-invariants -fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeeephole -fpeeephole2 -fpredictive-commoning -fprefetch-loop-arrays -free -freg-struct-return -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-critical-path-heuristic -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-interblock -fsched-last-insn-heuristic -fsched-rank-heuristic -fsched-spec -fsched-spec-insn-heuristic -fsched-stalled-insns-dep -fschedule-insns2 -fshow-column -fshrink-wrap -fsigned-zeros -fsplit-ivs-in-unroller -fsplit-wide-types -fstrict-aliasing -fstrict-overflow -fstrict-volatile-bitfields -fthread-jumps -ftoplevel-reorder -ftrapping-math -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-cselim -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-loop-distribute-patterns -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon -ftree-loop-optimize -ftree-parallelize-loops= -ftree-phi-prop -ftree-pre -ftree-pta -ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slp-vectorize -ftree-sra -ftree-switch-conversion -ftree-tail-merge -ftree-ter -ftree-vect-loop-version -ftree-vectorize -ftree-vrp -funit-at-a-time -funswitch-loops -funwind-tables -fvar-tracking -fvar-tracking-assignments -fvect-cost-model -fzero-initialized-in-bss -m128bit-long-double -m64 -m80387 -maccumulate-outgoing-args -malign-stringops -mfancy-math-387 -mfp-ret-in-387 -mglibc -mieee-fp -mmmx -mno-sse4 -mpush-args -mred-zone -msse -msse2 -mtls-direct-seg-refs

COMPILER OPTIMIZATION OBSTACLES

- Pointer aliasing
 - Pointer arithmetic
 - Global variables
 - Dynamic memory allocation
- Control dependencies
 - Indirect addressing
 - Floating point
- Function calls
 - Side effects
 - External functions
 - Virtual functions
- Optimization barriers
 - `volatile` variables
 - `printf`
 - Intrinsic functions
 - inline assembly
- Correctness overhead
 - Sign-extend
 - Stack overusage
 - Exception handling
 - Hardware bugs

POINTER ALIASING

- C/C++ language has pointers, C/C++ compiler has its worst problem - **pointer aliasing**
- Pointers **alias** when they point to the same address
 - Writing via one pointer will change the value read through another
- The compiler often doesn't know which pointers alias
 - The compiler must assume that any write through a pointer may affect the value read from any another pointer!
 - This can significantly reduce code efficiency

```
/* what if: timers(t1, t1, t1) */  
void timers(int *t1, int *t2, int *step)  
{  
    *t1 += *step;  
    *t2 += *step;  
}
```




EVERY TIME YOU OPTIMIZE
POINTER ARITHMETIC

GOD KILLS A KITTEN

OPTIMIZATION RULE #1

The First Rule of Program Optimization:

Don't do it.

The Second Rule (for experts only!):

Don't do it yet.

FALSE OPTIMIZATIONS

DON'T SECOND-GUESS THE COMPILER

There are things that the compiler can easily optimize by itself, for example, writing `a = a >> 1` should never replace `a = a / 2` if your intention is to divide the variable “a” by 2 and not shifting it. By doing so you are reducing the readability of your code without really improving anything, modern compilers are perfectly able to do that optimization by themselves.

PREFIX OR POSTFIX

```
for (int i = 0; i < 1000; i++)  
{  
    // do something  
}
```

```
for (int i = 0; i < 1000; ++i)  
{  
    // do something  
}
```

The advice to use prefix form was good in the 90ies, today it rarely matters, even in C++

SMART MATH

```
int mul320_normal(int x)
{
    return x*320;
}
```

```
int mul320_fast(int x)
{
    return (x<<8) + (x<<6);
}
```

gcc -O3 compiles both versions to

```
imull 320, edi, eax
```

SMART MATH #2

Here's how you divide an unsigned int by 13 in C:

```
unsigned divide_by_13(unsigned x)
{
    return x / 13;
}
```

Here is how compiler do it:

```
unsigned divide_by_13(unsigned x)
{
    return (unsigned)(x*1321528399ULL >> 34);
}
```

#DEFINE FOR NUMERIC CONSTANTS

- `#define CONSTANT 23`
- `const int Constant=23;`
- `static const int Constant=23;`
- `enum { constant=23 };`

No memory references and additions in generated code:

```
void foo(void)
{
    a(constant+3);
    a(CONSTANT+4);
    a(Constant+5);
}
```

```
foo:
```

```
subq $8, %rsp
movl $26, %edi
call a
movl $27, %edi
call a
movl $28, %edi
addq $8, %rsp
jmp a
```

MACRO VS INLINE

```
#define abs(x) ((x)>0?(x):- (x))
static long abs2(int x) { return x>=0?x:-x; }

int foo(int a) { return abs(a); }
int bar(int a) { return abs2(a); }
```

Compiler emits inlined branchless code:

```
foo:
bar:
    mov edx,edi
    sar edx,31      @ int tmp = x >> (sizeof(x) * 8 - 1);
    mov eax,edx
    xor eax,edi     @ return (tmp ^ x) - tmp;
    sub eax,edx
    ret
```


OUTSMARTING THE COMPILER

```
/* original code */
unsigned foo(unsigned char i)
{ // 3*shl, 3*or
    return i | (i<<8) | (i<<16) | (i<<24);
}
```

```
/* attempt to improve foo */
unsigned bar(unsigned char i)
{ // 2*shl, 2*or
    unsigned int j = i | (i << 8);
    return j | (j<<16);
}
```

```
/* "let the compiler do it" */
unsigned baz(unsigned char i)
{ // 1*imul
    return i*0x01010101;
}
```

C/C++ KEYWORDS

- **inline** - language overhead
- **const** - means nothing for optimizer (by default)
- **register** - complete placebo
- **restrict** - ~~C only~~; mostly useless
- **volatile** - optimization barrier

“STATIC” KEYWORD

- Mark constants **static**
 - compiler will not reserve memory slots for static constants
- Mark helper functions **static**
 - **static** gives you internal linkage: compiler will know that it sees all usages of that function
 - compiler will inline static functions used only once
- In C++11 anonymous namespace has exactly the same effect as **static** keyword (it does not in C++98)
- If you have to use global variables then make them **static**

THE END