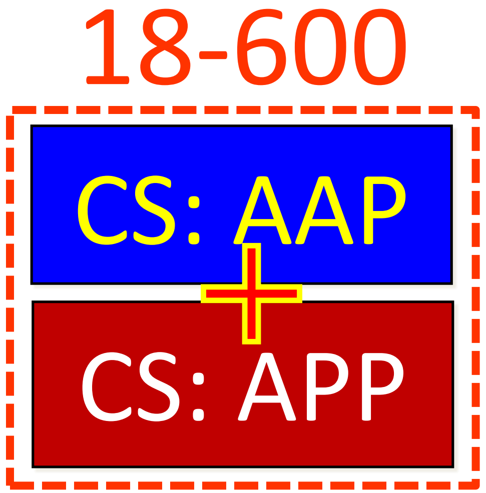
18-600 Foundations of Computer Systems

Lecture 2:
“Computer Systems: The Big Picture”

John P. Shen & Gregory Kesden
August 30, 2017


18-600

CS: AAP

+

CS: APP

- Recommended Reference:
  - **Chapters 1 and 2 of Shen and Lipasti (SnL).**
- Other Relevant References:
  - “A Detailed Analysis of Contemporary ARM and x86 Architectures” by Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam . (2013)
  - “Amdahl’s and Gustafson’s Laws Revisited” by Andrzej Karbowski. (2008)

Electrical & Computer ENGINEERING

Carnegie Mellon University

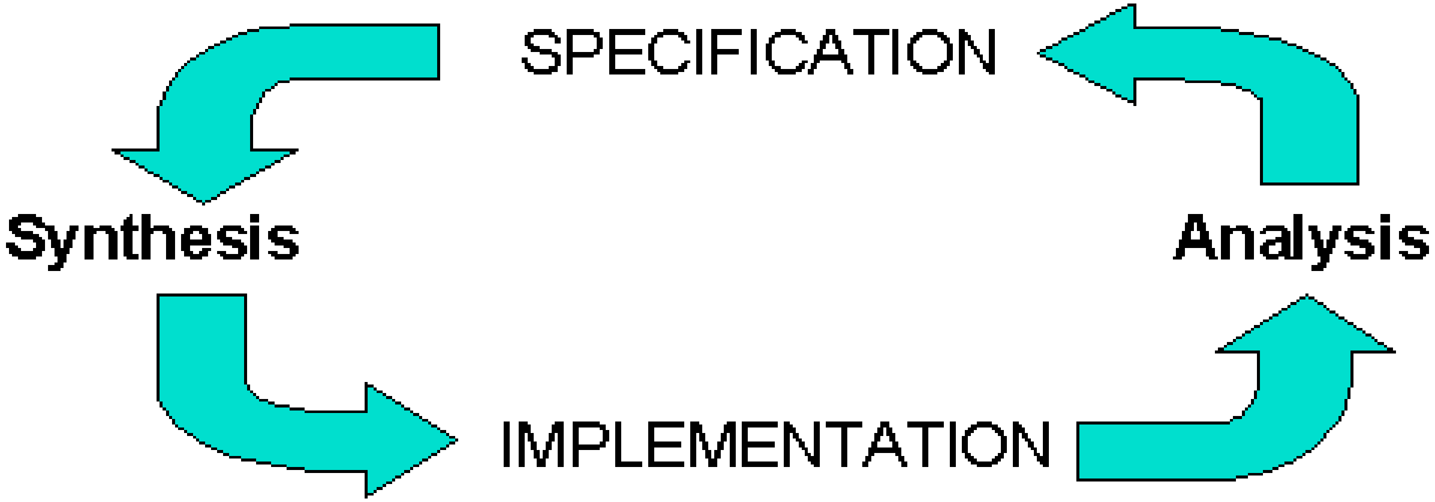
18-600 Foundations of Computer Systems

Lecture 2:
"Computer Systems: The Big Picture"

1. **Instruction Set Architecture (ISA)**
   a. Hardware / Software Interface (HSI)
   b. Dynamic / Static Interface (DSI)
   c. Instruction Set Architecture Design & Examples
2. **Historical Perspective on Computing**
   a. Major Epochs of Modern Computers
   b. Computer Performance Iron Law (#1)
3. **"Economics" of Computer Systems**
   a. Amdahl's Law and Gustafson's Law
   b. Moore's Law and Bell's Law

Electrical & Computer ENGINEERING

Anatomy of Engineering Design



SPECIFICATION: Behavioral description of *"What does it do?"*

Synthesis: Search for possible solutions; pick best one. *Creative process*

IMPLEMENTATION: Structural description of *"How is it constructed?"*

Analysis: Validate if the design meets the specification.

"Does it do the right thing?" + "How well does it perform?"

[Gerrit Blaauw & Fred Brooks, 1981]

Instruction Set Processor Design

ARCHITECTURE: (ISA) programmer/compiler view = SPECIFICATION

- Functional programming model to application/system programmers
- Opcodes, addressing modes, architected registers, IEEE floating point

IMPLEMENTATION: (μarchitecture) processor designer view

- Logical structure or organization that performs the ISA specification
- Pipelining, functional units, caches, physical registers, buses, branch predictors

REALIZATION: (Chip) chip/system designer view

- Physical structure that embodies the implementation
- Gates, cells, transistors, wires, dies, packaging

Lecture 2: “Computer Systems: The Big Picture”

1. Instruction Set Architecture (ISA)

a. Hardware / Software Interface (HSI)

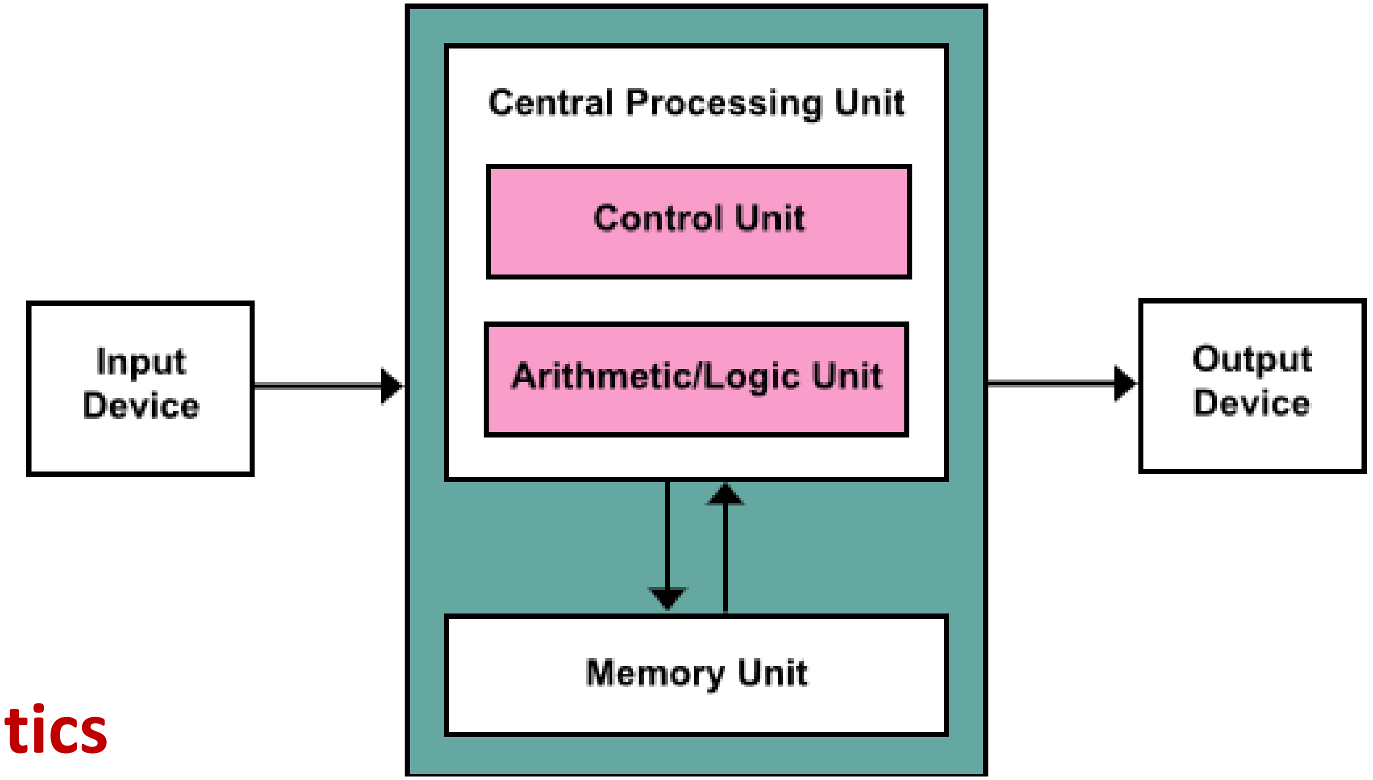
b. Dynamic / Static Interface (DSI)

c. Instructure Set Architecture Design & Examples

Electrical & Computer ENGINEERING

The Von Neumann Stored Program Computer

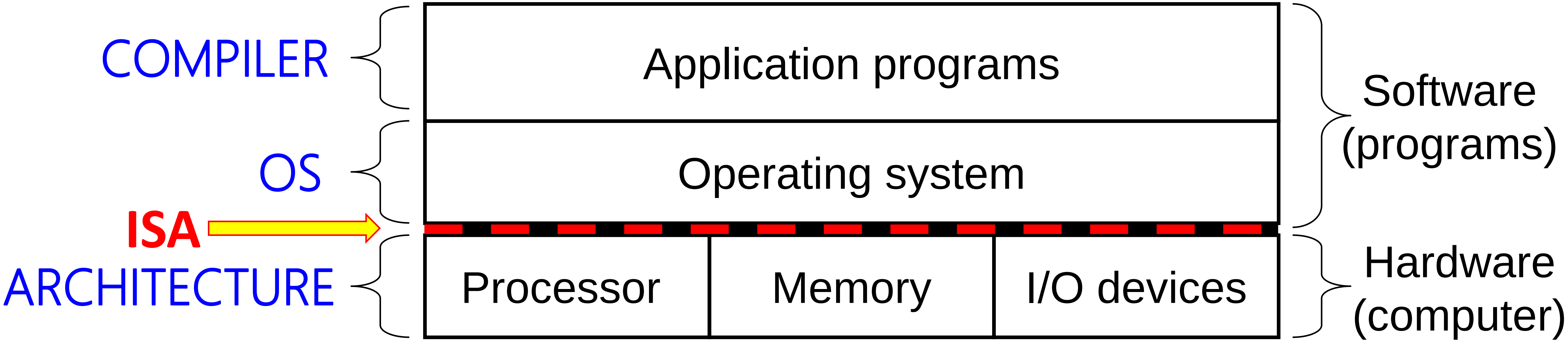
- **The Classic Von Neumann Computation Model**: Proposed in 1945 by John Von Neumann and others (Alan Turing, J. Presper Eckert and John Mauchly).
- **A "Stored Program Computer"**
  1. **One CPU**
     - One Control Unit
       - Program Counter
       - Instruction Register
     - One ALU
  2. **Monolithic Memory**
     - Data Store
     - Instruction Store
  3. **Sequential Execution Semantics**
     - **Instructions from an Instruction Set (ISA)**



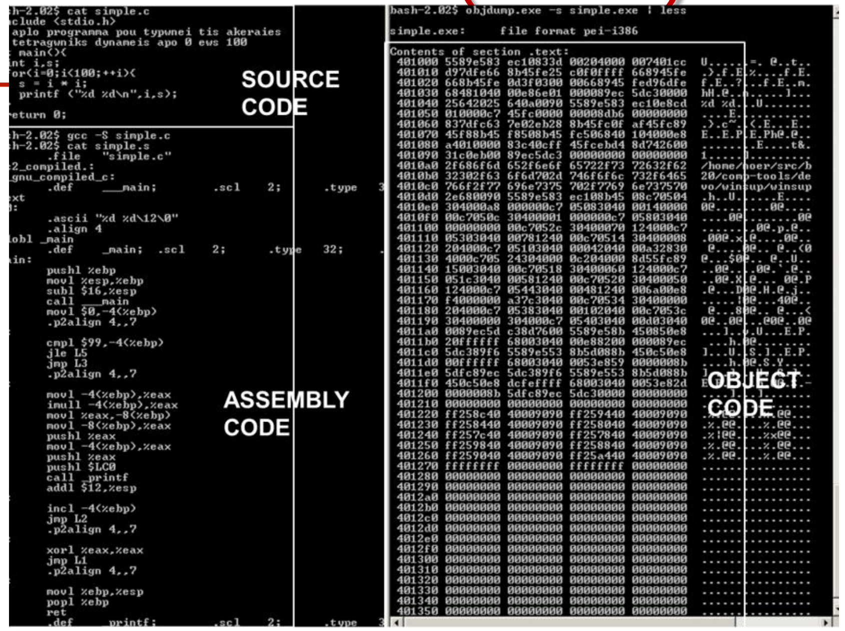
Anatomy of a Computer System: (ISA)

➢ What is a Computer System?

- ❖ Software + Hardware
- ❖ Programs + Computer ➔ [Application program + OS] + Computer
- ❖ Programming Languages + Operating Systems + Computer Architecture



Computer Hardware/Software Interface (HSI)



```
h-2.02$ cat simple.c
clude <stdio.h>
 aplo programma pou typwnei tis akeraies
 tetragwniks dynameis apo 0 ews 100
 main(){
int i,s;
for(i=0;i<100;++i){
  s = i * i;
  printf ("%d %d\n",i,s);

eturn 0;
```

SOURCE CODE

```
h-2.02$ gcc -S simple.c
h-2.02$ cat simple.s
        .file   "simple.c"
2_compiled.:
gnu_compiled_c:
        .def    ___main;        .scl    2;      .type   3
xt
:
        .ascii "%d %d\12\0"
        .align 4
obl _main
        .def    _main;  .scl    2;      .type   32;
in:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        call ___main
        movl $0,-4(%ebp)
        .p2align 4,,7

        cmpl $99,-4(%ebp)
        jle L5
        jmp L3
        .p2align 4,,7

        movl -4(%ebp),%eax
        imull -4(%ebp),%eax
        movl %eax,-8(%ebp)
        movl -8(%ebp),%eax
        pushl %eax
        movl -4(%ebp),%eax
        pushl %eax
        pushl $LC0
        call _printf
        addl $12,%esp

        incl -4(%ebp)
        jmp L2
        .p2align 4,,7

        xorl %eax,%eax
        jmp L1
        .p2align 4,,7

        movl %ebp,%esp
        popl %ebp
        ret
        .def    _printf;        .scl    2;      .type   3
```

ASSEMBLY CODE

```
bash-2.02$ objdump.exe -s simple.exe | less

simple.exe:     file format pei-i386

Contents of section .text:
 401000 5589e583 ec10833d 00204000 007401cc  U......=. @..t..
 401010 d97dfe66 8b45fe25 c0f0ffff 668945fe  .}.f.E.%....f.E.
 401020 668b45fe 0d3f0300 00668945 fed96dfe  f.E..?...f.E..m.
 401030 68481040 00e86e01 000089ec 5dc30000  hH.@..n.....]...
 401040 25642025 640a0090 5589e583 ec10e8cd  %d %d...U.......
 401050 010000c7 45fc0000 00008db6 00000000  ....E...........
 401060 837dfc63 7e02eb28 8b45fc0f af45fc89  .}.c~..(.E...E..
 401070 45f88b45 f8508b45 fc506840 104000e8  E..E.P.E.Ph@.@..
 401080 a4010000 83c40cff 45fcebd4 8d742600  ........E.....t&.
 401090 31c0eb00 89ec5dc3 00000000 00000000  1.....].........
 4010a0 2f686f6d 652f6e6f 65722f73 72632f62  /home/noer/src/b
 4010b0 32302f63 6f6d702d 746f6f6c 732f6465  20/comp-tools/de
 4010c0 766f2f77 696e7375 702f7769 6e737570  vo/winsup/winsup
 4010d0 2e680090 5589e583 ec108b45 08c70504  .h..U......E....
 4010e0 304000a8 000000c7 05083040 00140000  0@........0@....
 4010f0 00c7050c 30400001 000000c7 05803040  ....0@........0@
 401100 00000000 00c7052c 30400070 124000c7  .......,0@.p.@..
 401110 05303040 00781240 00c70514 30400008  .00@.x.@....0@..
 401120 204000c7 05103040 00042040 00a32830   @....0@.. @..(0
 401130 4000c705 24304000 0c204000 8d55fc89  @...$0@.. @..U..
 401140 15003040 00c70518 30400060 124000c7  ..0@....0@.`.@..
 401150 051c3040 00581240 00c70520 30400050  ..0@.X.@... 0@.P
 401160 124000c7 05443040 00481240 006a00e8  .@...D0@.H.@.j..
 401170 f4000000 a37c3040 00c70534 30400000  .....|0@...40@..
 401180 204000c7 05383040 00102040 00c7053c   @...80@.. @...<
 401190 30400000 304000c7 05403040 00d03040  0@..0@...@0@..0@
 4011a0 0089ec5d c38d7600 5589e58b 450850e8  ...]..v.U...E.P.
 4011b0 20ffffff 68003040 00e88200 000089ec   ...h.0@........
 4011c0 5dc389f6 5589e553 8b5d088b 450c50e8  ]...U..S.]..E.P.
 4011d0 00ffffff 68003040 0053e859 0000008b  ....h.0@.S.Y....
 4011e0 5dfc89ec 5dc389f6 5589e553 8b5d088b  ]...]...U..S.]..
 4011f0 450c50e8 dcfeffff 68003040 0053e82d  E.P.....h.0@.S.-
 401200 0000008b 5dfc89ec 5dc30000 00000000  ....]...].......
 401210 00000000 00000000 00000000 00000000  ................
 401220 ff258c40 40009090 ff259440 40009090  .%.@@....%.@@...
 401230 ff258440 40009090 ff258040 40009090  .%.@@....%.@@...
 401240 ff257c40 40009090 ff257840 40009090  .%|@@....%x@@...
 401250 ff259840 40009090 ff258840 40009090  .%.@@....%.@@...
 401260 ff259040 40009090 ff25a440 40009090  .%.@@....%.@@...
 401270 ffffffff 00000000 ffffffff 00000000  ................
 401280 00000000 00000000 00000000 00000000  ................
 401290 00000000 00000000 00000000 00000000  ................
 4012a0 00000000 00000000 00000000 00000000  ................
 4012b0 00000000 00000000 00000000 00000000  ................
 4012c0 00000000 00000000 00000000 00000000  ................
 4012d0 00000000 00000000 00000000 00000000  ................
 4012e0 00000000 00000000 00000000 00000000  ................
 4012f0 00000000 00000000 00000000 00000000  ................
 401300 00000000 00000000 00000000 00000000  ................
 401310 00000000 00000000 00000000 00000000  ................
 401320 00000000 00000000 00000000 00000000  ................
 401330 00000000 00000000 00000000 00000000  ................
 401340 00000000 00000000 00000000 00000000  ................
 401350 00000000 00000000 00000000 00000000  ................
```

OBJECT CODE

Software Engineering

CS: APP

Program Development

Application Software
- Program development
- Program compilation

SPECIFICATION

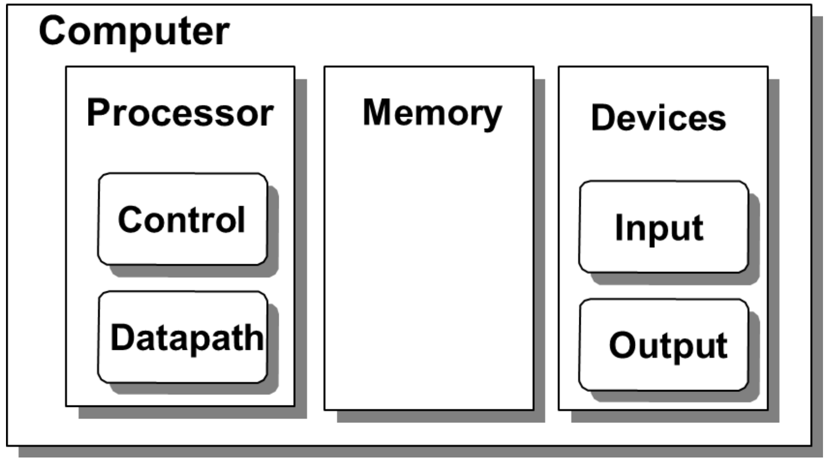
Instruction Set Architecture (ISA)

CS: AAP

Computer Engineering

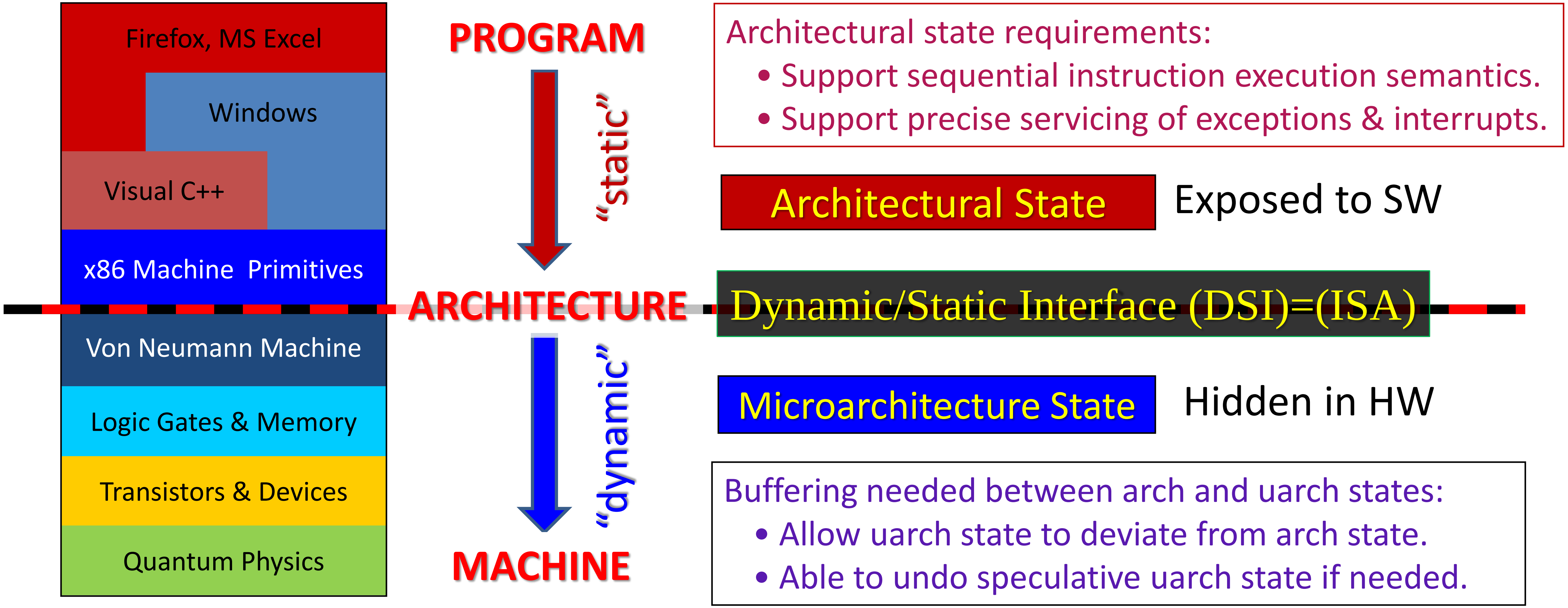
Processor Design

Hardware Technology
- Program execution
- Computer performance



IMPLEMENTATION

Computer Dynamic/Static Interface (DSI)



Architectural state requirements:

- Support sequential instruction execution semantics.
- Support precise servicing of exceptions & interrupts.

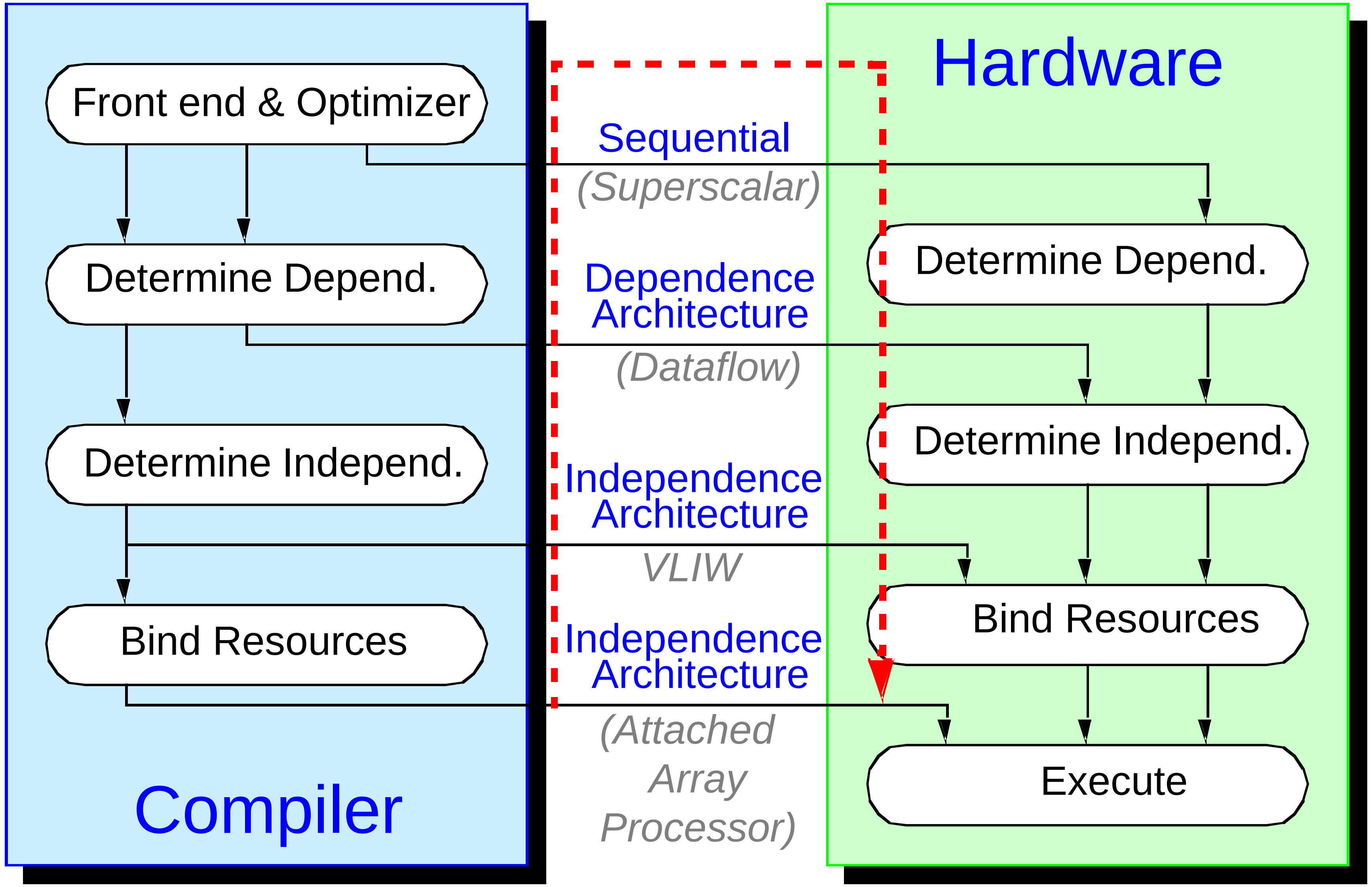
Buffering needed between arch and uarch states:

- Allow uarch state to deviate from arch state.
- Able to undo speculative uarch state if needed.

DSI = ISA = a contract between the program and the machine.

[B. Rau & J. Fisher, 1993]

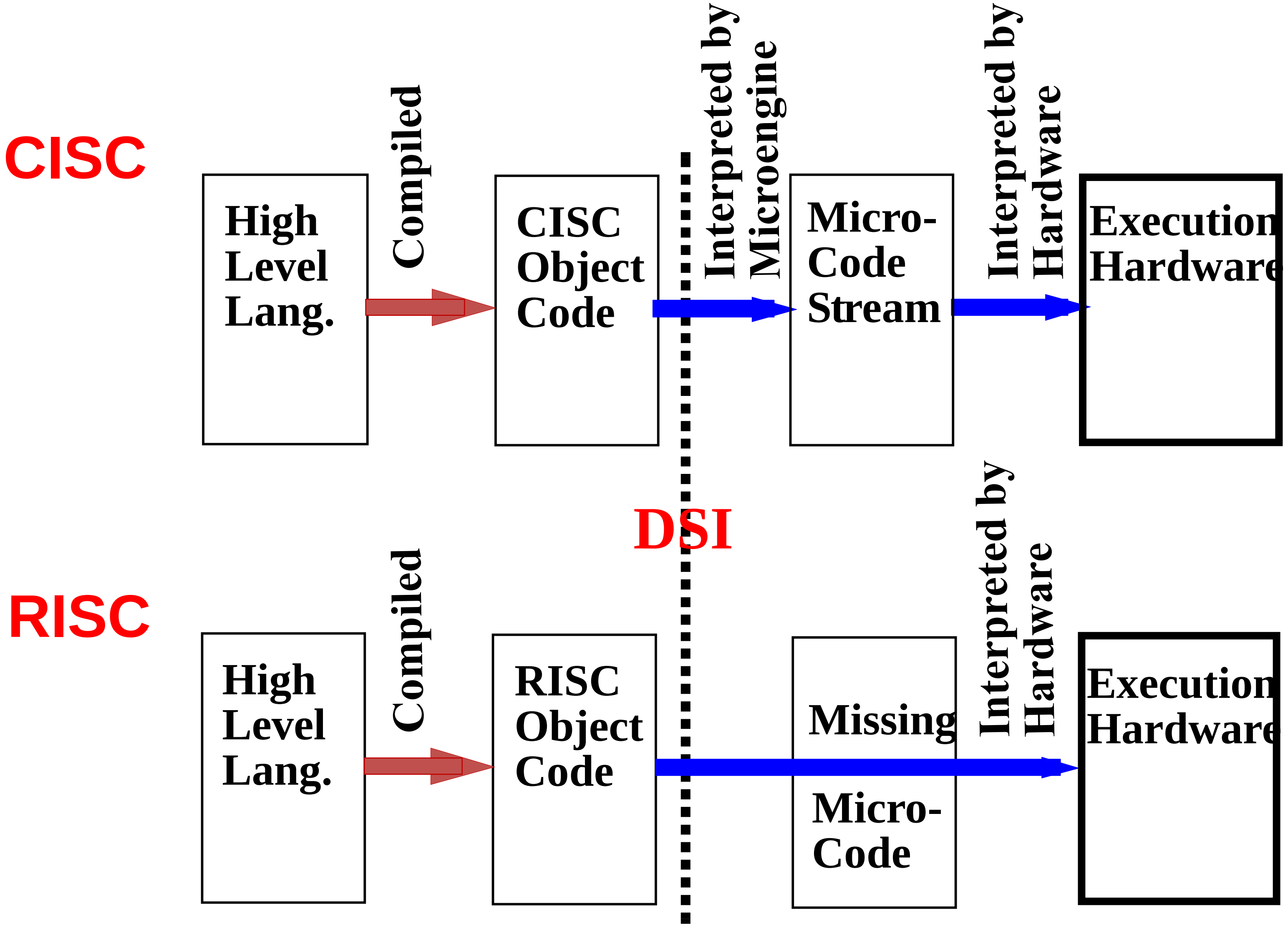
Dynamic/Static Design Space: DSI Placement



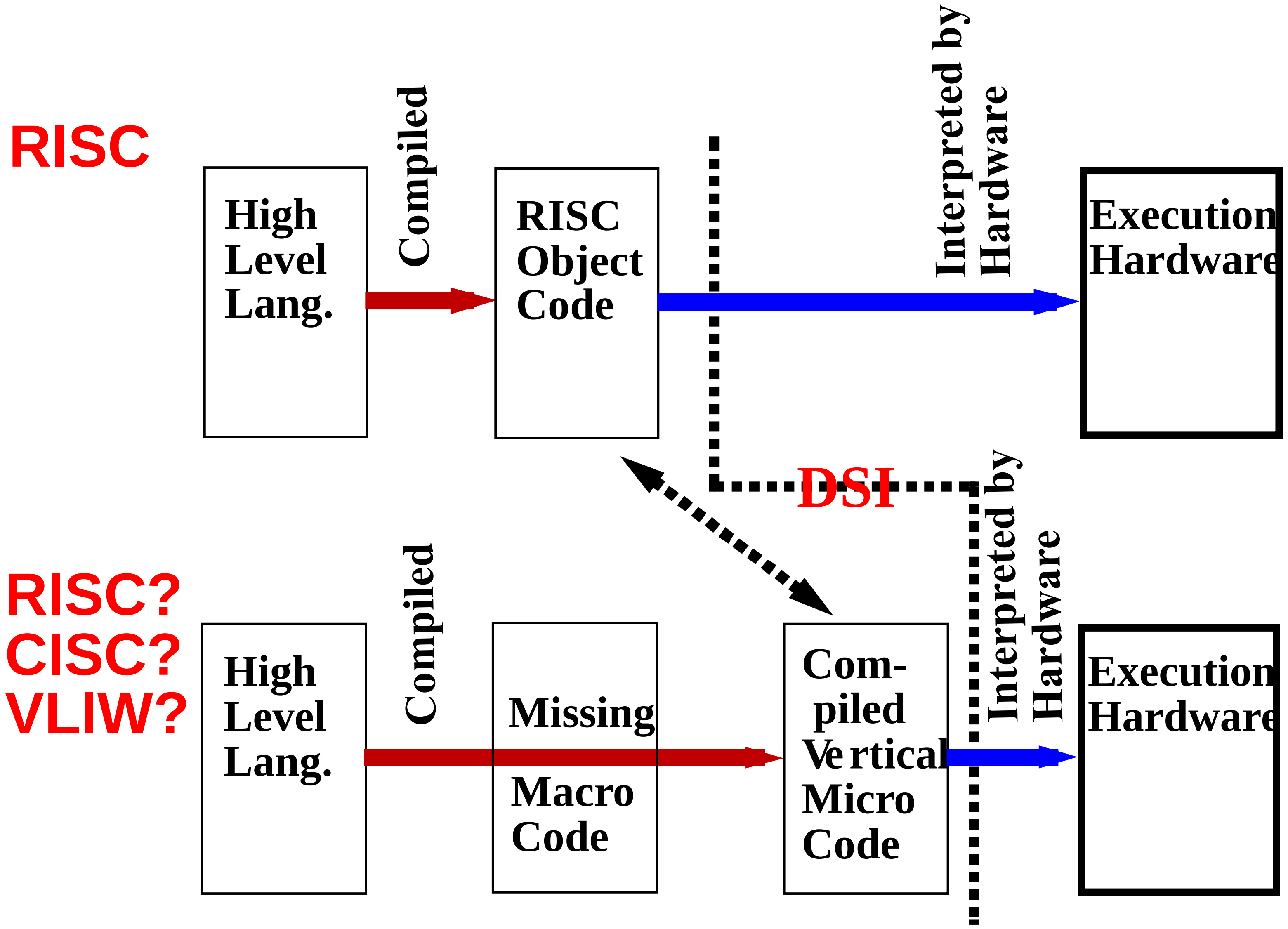
RISC vs. CISC

[Josh Fisher, HP]

Transition from CISC to RISC:

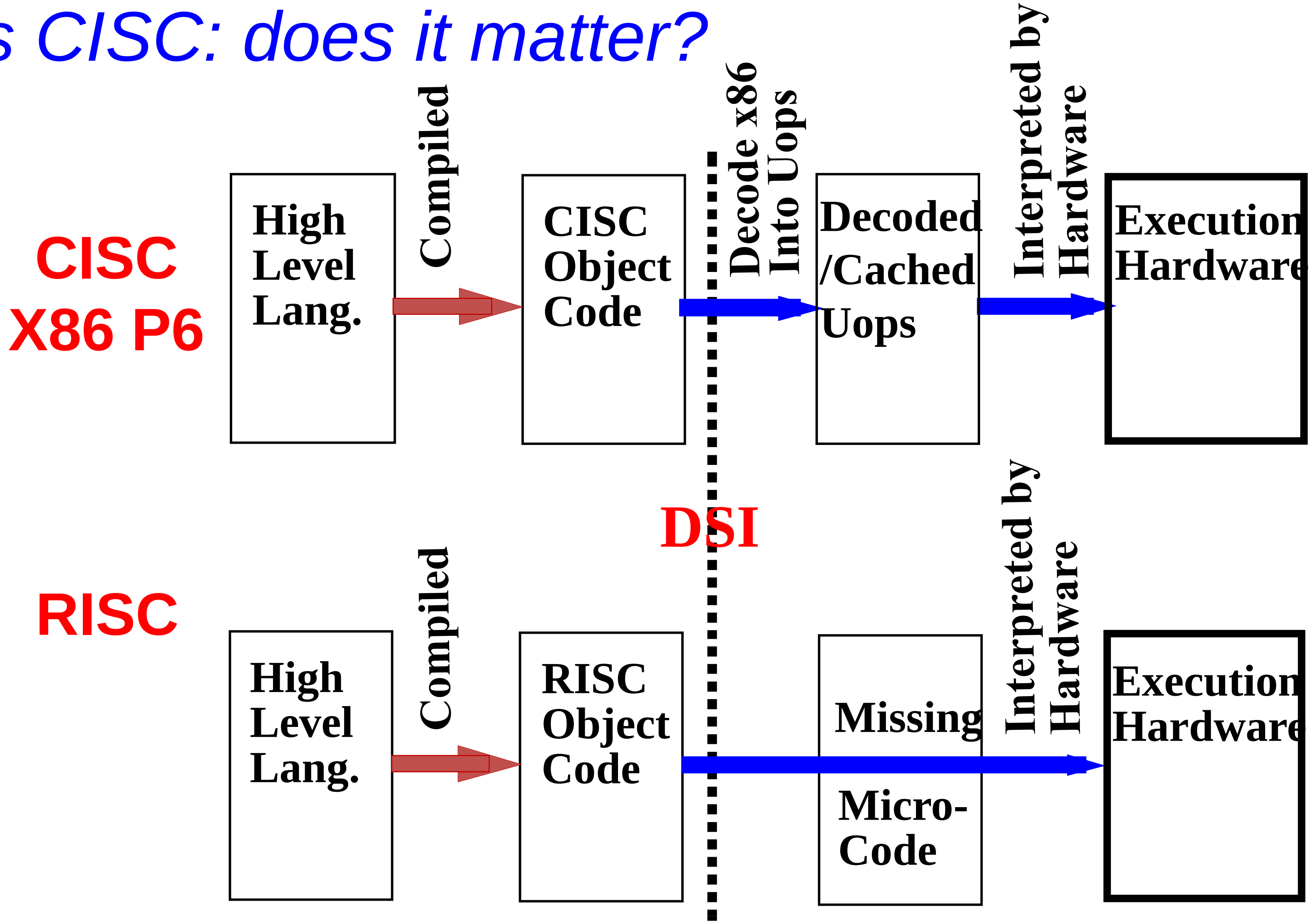


Another way to view RISC



All x86 processors since Pentium Pro (P6)

RISC vs CISC: does it matter?



Instruction Set Architecture (ISA) Design

- **Instruction Types**
  - Operation Specifiers (OpCodes)
  - Operand Specifiers
  - Addressing Modes
- **Exceptions Handling**
- **Design Styles**: Placement of DSI
  - RISC vs. CISC
  - Historically Important ISAs

Instruction Types and OpCodes

FOUR CLASSES OF INSTRUCTIONS ARE CONSIDERED:

- **INTEGER ARITHMETIC/LOGIC INSTRUCTIONS**
  - ADD, SUB, MULT
  - ADDU, SUBU,MULTU
  - OR, AND, NOR, NAND
- **FLOATING POINT INSTRUCTIONS**
  - FADD, FMUL, FDIV
  - COMPLEX ARITHMETIC
- **MEMORY TRANSFER INSTRUCTIONS**
  - LOADS AND STORES
  - TEST AND SET, AND SWAP
  - MAY APPLY TO VARIOUS OPERAND SIZES
- **CONTROL FLOW INSTRUCTIONS**
  - BRANCHES ARE CONDITIONAL
  - CONDITION MAY BE CONDITION BITS (ZCVXN)
  - CONDITION MAY TEST THE VALUE OF A REGISTER (SET BY SLT INSTRUCTION)
  - CONDITION MAY BE COMPUTED IN THE BRANCH INSTRUCTION ITSELF
  - JUMPS ARE UNCONDITIONAL WITH ABSOLUTE ADDRESS OR ADDRESS IN REGISTER
  - JAL (JUMP AND LINK) NEEDED FOR PROCEDURES

CPU Operands

- **INCLUDE: ACCUMULATORS, EVALUATION STACKS, REGISTERS, AND IMMEDIATE VALUES**
- **ACCUMULATORS:**
  - ADDA <mem_address>
  - MOVA <mem_address>
- **STACK**
  - PUSH <mem_address>
  - ADD
  - POP <mem_address>
- **REGISTERS**
  - LW R1, <memory-address>
  - SW R1, <memory_address>
  - ADD R2, <memory_address>
  - ADD R1,R2,R4
    - LOAD/STORE ISAs
  - MANAGEMENT BY THE COMPILER: REGISTER SPILL/FILL
- **IMMEDIATE**
  - ADDI R1,R2,#5

Memory Operands

- **OPERAND ALIGNEMENT**
  - BYTE-ADDRESSABLE MACHINES
  - OPERANDS OF SIZE S MUST BE STORED AT AN ADDRESS THAT IS MULTPIPLE OF S
  - BYTES ARE ALWAYS ALIGNED
  - HALF WORDS (16BITS) ALIGNED AT 0, 2, 4, 6
  - WORDS (32 BITS) ARE ALIGNED AT 0, 4, 8, 12, 16,..
  - DOUBLE WORDS (64 BITS) ARE ALIGNED AT 0, 8, 16,...
  - COMPILER IS RESPONSIBLE FOR ALIGNING OPERANDS. HARDWARE CHECKS AND TRAPS IF MISALIGNED
  - OPCODE INDICATES SIZE (ALSO: TAGS IN MEMORY)

- **LITTLE vs. BIG ENDIAN**
  - BIG ENDIAN: MSB IS STORED AT ADDRESS XXXXXX00
  - LITTLE ENDIAN: LSB IS STORED AT ADDRESS XXXXXX00
  - PORTABILITY PROBLEMS, CONFIGURABLE ENDIANNESS

Addressing Modes

MODE	EXAMPLE	MEANING
REGISTER	ADD R4,R3	reg[R4] <- reg[R4] +reg[R3]
IMMEDIATE	ADD R4, #3	reg[R4] <- reg[R4] + 3
DISPLACEMENT	ADD R4, 100(R1)	reg[R4] <- reg[R4] + Mem[100 + reg[R1]]
REGISTER INDIRECT	ADD R4, (R1)	reg[R4] <- reg[R4] + Mem[reg[R1]]
INDEXED	ADD R3, (R1+R2)	reg[R3] <- reg[R3] + Mem[reg[R1] + reg[R2]]
DIRECT OR ABSOLUTE	ADD R1, (1001)	reg[R1] <- reg[R1] + Mem[1001]
MEMORY INDIRECT	ADD R1, @R3	reg[R1] <- reg[R1] + Mem[Mem[Reg[3]]]
POST INCREMENT	ADD R1, (R2)+	ADD R1, (R2) then R2 <- R2+d
PREDECREMENT	ADD R1, -(R2)	R2 <- R2-d then ADD R1, (R2)
PC-RELATIVE	BEZ R1, 100	if R1==0, PC <- PC+100
PC-RELATIVE	JUMP 200	Concatenate bits of PC and offset

Exceptions and Interrupts

- **EVENTS TRIGGERED BY PROGRAM and HARDWARE, FORCING THE PROCESSOR TO EXECUTE A HANDLER**
  - INCLUDES EXCEPTIONS AND INTERRUPTS
- **EXCEPTION & INTERRUP EXAMPLES:**
  - I/O DEVICE INTERRUPTS
  - OPERATING SYSTEM CALLS
  - INSTRUCTION TRACING AND BREAKPOINTS
  - INTEGER OR FLOATING-POINT ARITHMETIC EXCEPTIONS
  - PAGE FAULTS
  - MISALIGNED MEMORY ACCESSES
  - MEMORY PROTECTION VIOLATIONS
  - UNDEFINED INSTRUCTIONS
  - HARDWARE FAILURE/ALARMS
  - POWER FAILURES
- **PRECISE EXCEPTIONS:**
  - SYNCHRONIZED WITH AN INSTRUCTION
  - MUST RESUME EXECUTION AFTER HANDLER
  - SAVE THE PROCESS STATE AT THE FAULTING INSTRUCTION
  - OFTEN DIFFICULT IN ARCHITECTURES WHERE MULTIPLE INSTRUCTIONS EXECUTE

Historically Important ISAs & Implementations

ISA	Company	Implementations	Type
System 370	IBM	IBM 370/3081	CISC--Legacy
x86	Intel/AMD	Many, many, …	CISC-Legacy
Motorola68000	Motorola	Motorola 68020	CISC-Legacy
Sun SPARC	Sun Microsystems	SPARC T2	RISC
PowerPC	IBM/Motorola	PowerPC-6	RISC
Alpha	DEC/Compaq/HP	Alpha 21264	RISC-Retired
MIPS	MIPS/SGI	MIPS10000	RISC
IA-64	Intel	Itanium-2	RISC-Retired
ARM	ARM/QC/Samsung	Many, many, …	RISC

X86-64 Instruction Set Architecture

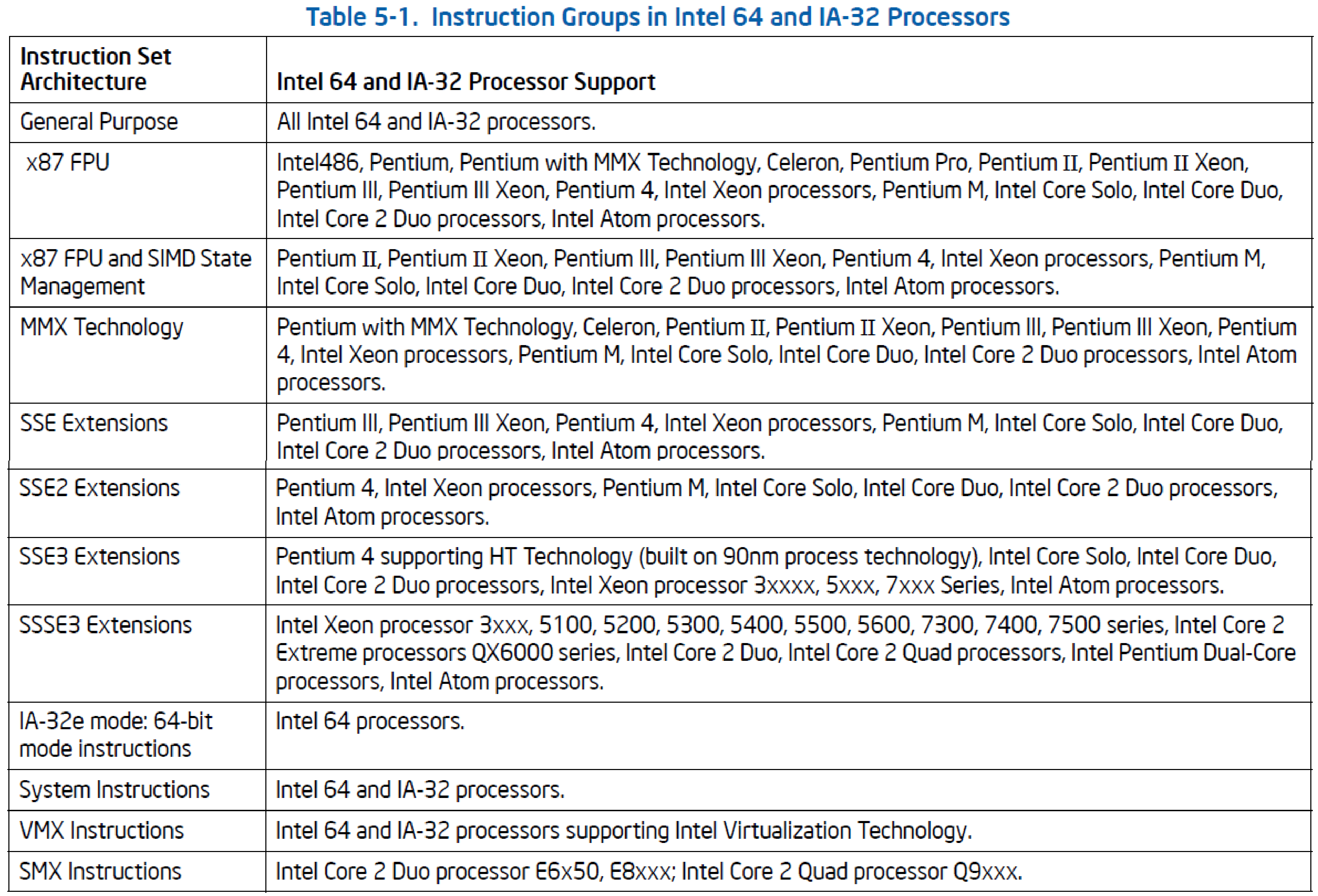
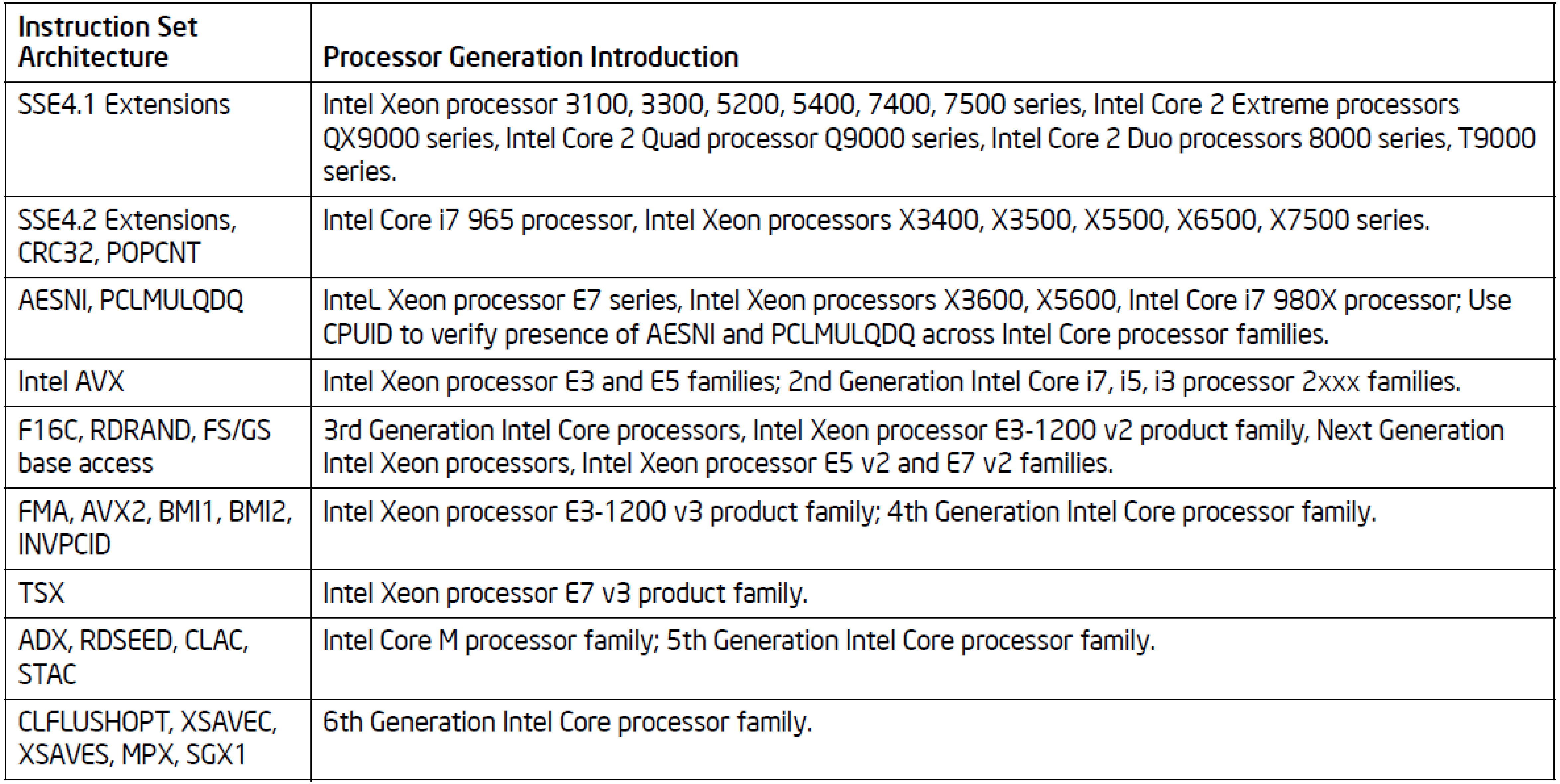


Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors.
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors.
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.
IA-32e mode: 64-bit mode instructions	Intel 64 processors.
System Instructions	Intel 64 and IA-32 processors.
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology.
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.

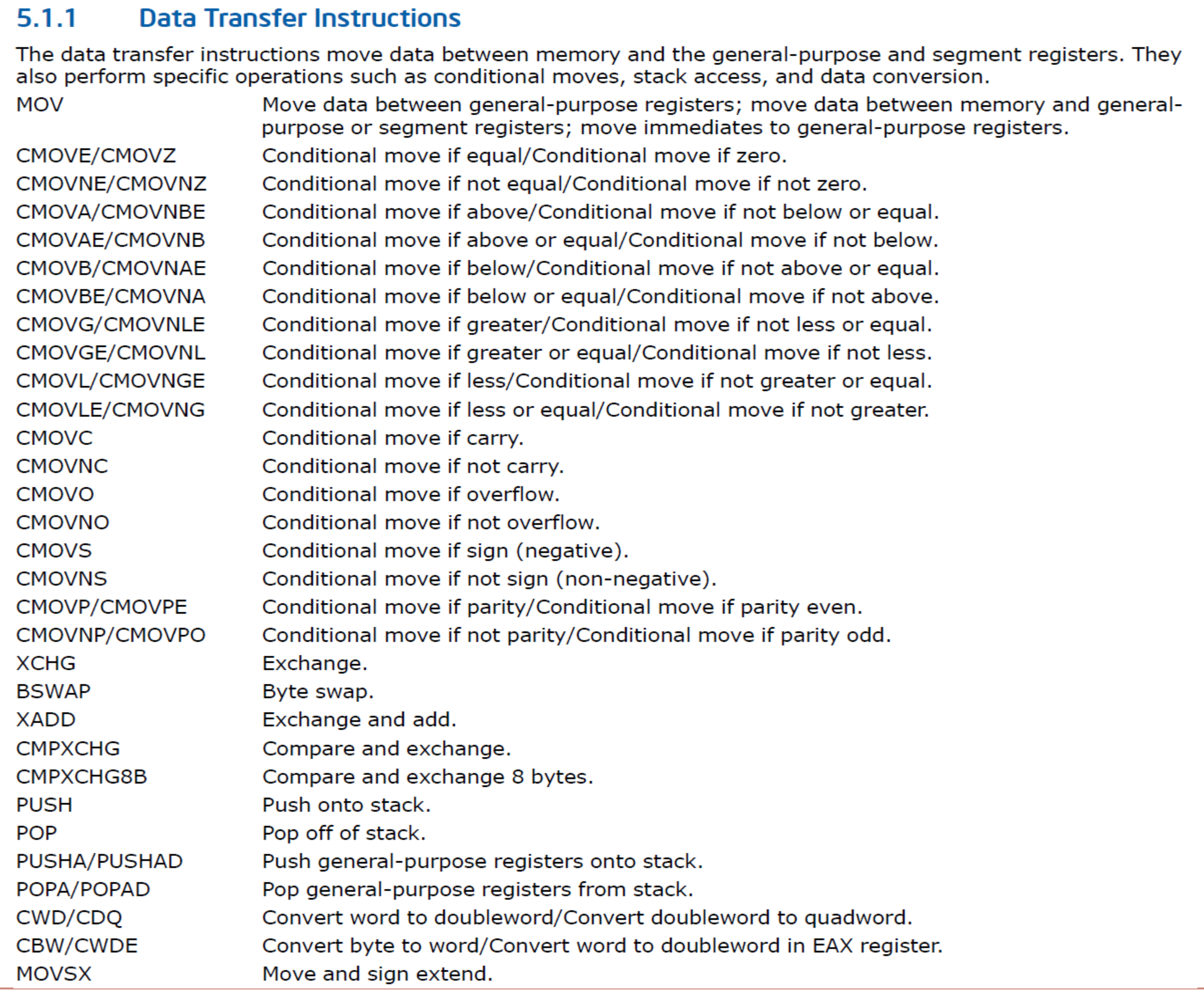
X86-64 Instruction Set Architecture

Table 5-2. Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors



Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel Xeon processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel Core 2 Extreme processors QX9000 series, Intel Core 2 Quad processor Q9000 series, Intel Core 2 Duo processors 8000 series, T9000 series.
SSE4.2 Extensions, CRC32, POPCNT	Intel Core i7 965 processor, Intel Xeon processors X3400, X3500, X5500, X6500, X7500 series.
AESNI, PCLMULQDQ	InteL Xeon processor E7 series, Intel Xeon processors X3600, X5600, Intel Core i7 980X processor; Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel Core processor families.
Intel AVX	Intel Xeon processor E3 and E5 families; 2nd Generation Intel Core i7, i5, i3 processor 2xxx families.
F16C, RDRAND, FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Next Generation Intel Xeon processors, Intel Xeon processor E5 v2 and E7 v2 families.
FMA, AVX2, BMI1, BMI2, INVPCID	Intel Xeon processor E3-1200 v3 product family; 4th Generation Intel Core processor family.
TSX	Intel Xeon processor E7 v3 product family.
ADX, RDSEED, CLAC, STAC	Intel Core M processor family; 5th Generation Intel Core processor family.
CLFLUSHOPT, XSAVEC, XSAVES, MPX, SGX1	6th Generation Intel Core processor family.

X86-64 Instruction Set Architecture

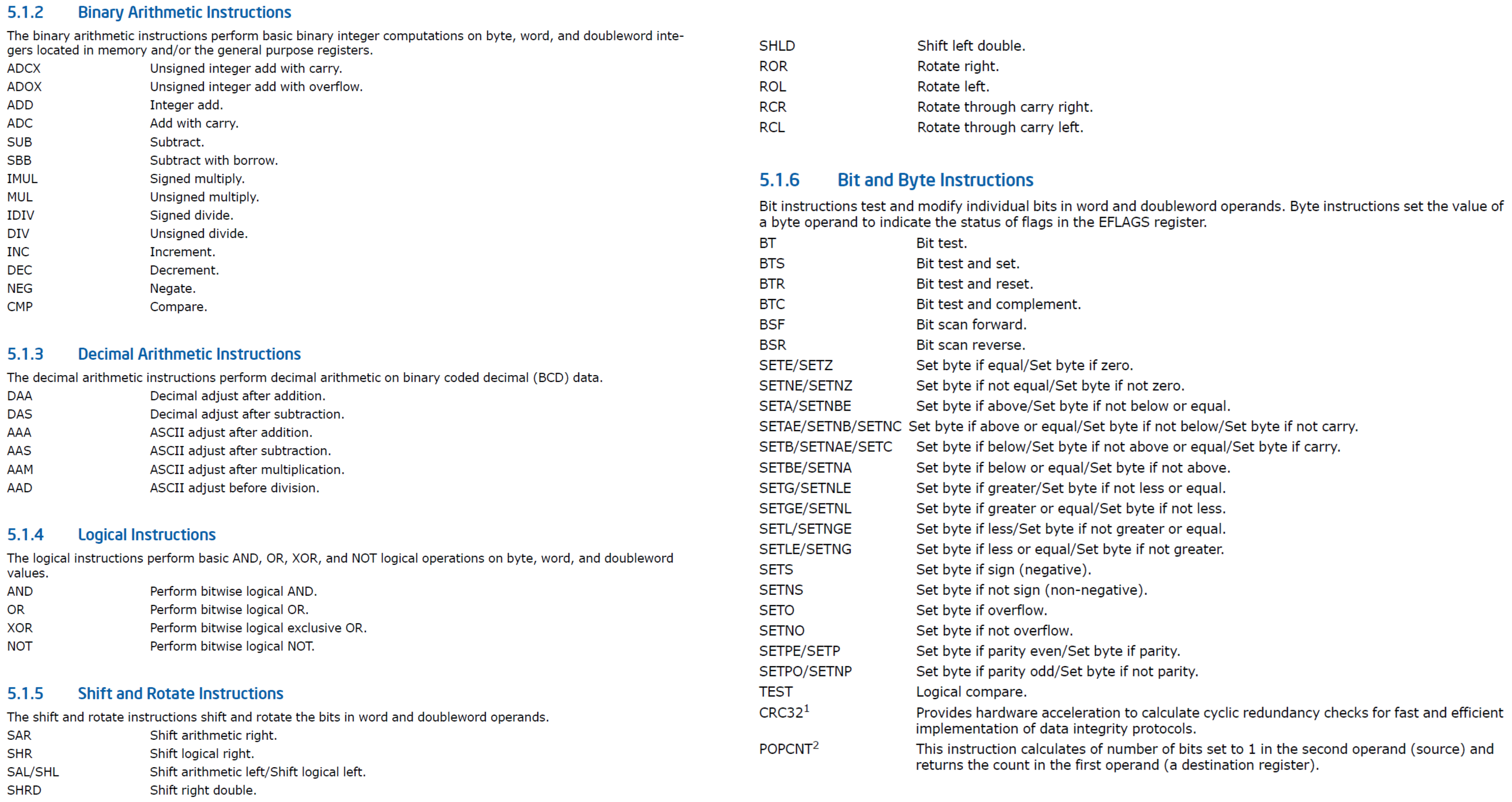


5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).
CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.

X86-64 Instruction Set Architecture



5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADCX	Unsigned integer add with carry.
ADOX	Unsigned integer add with overflow.
ADD	Integer add.
ADC	Add with carry.
SUB	Subtract.
SBB	Subtract with borrow.
IMUL	Signed multiply.
MUL	Unsigned multiply.
IDIV	Signed divide.
DIV	Unsigned divide.
INC	Increment.
DEC	Decrement.
NEG	Negate.
CMP	Compare.

5.1.3 Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition.
DAS	Decimal adjust after subtraction.
AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.
AAM	ASCII adjust after multiplication.
AAD	ASCII adjust before division.

5.1.4 Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

AND	Perform bitwise logical AND.
OR	Perform bitwise logical OR.
XOR	Perform bitwise logical exclusive OR.
NOT	Perform bitwise logical NOT.

5.1.5 Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands.

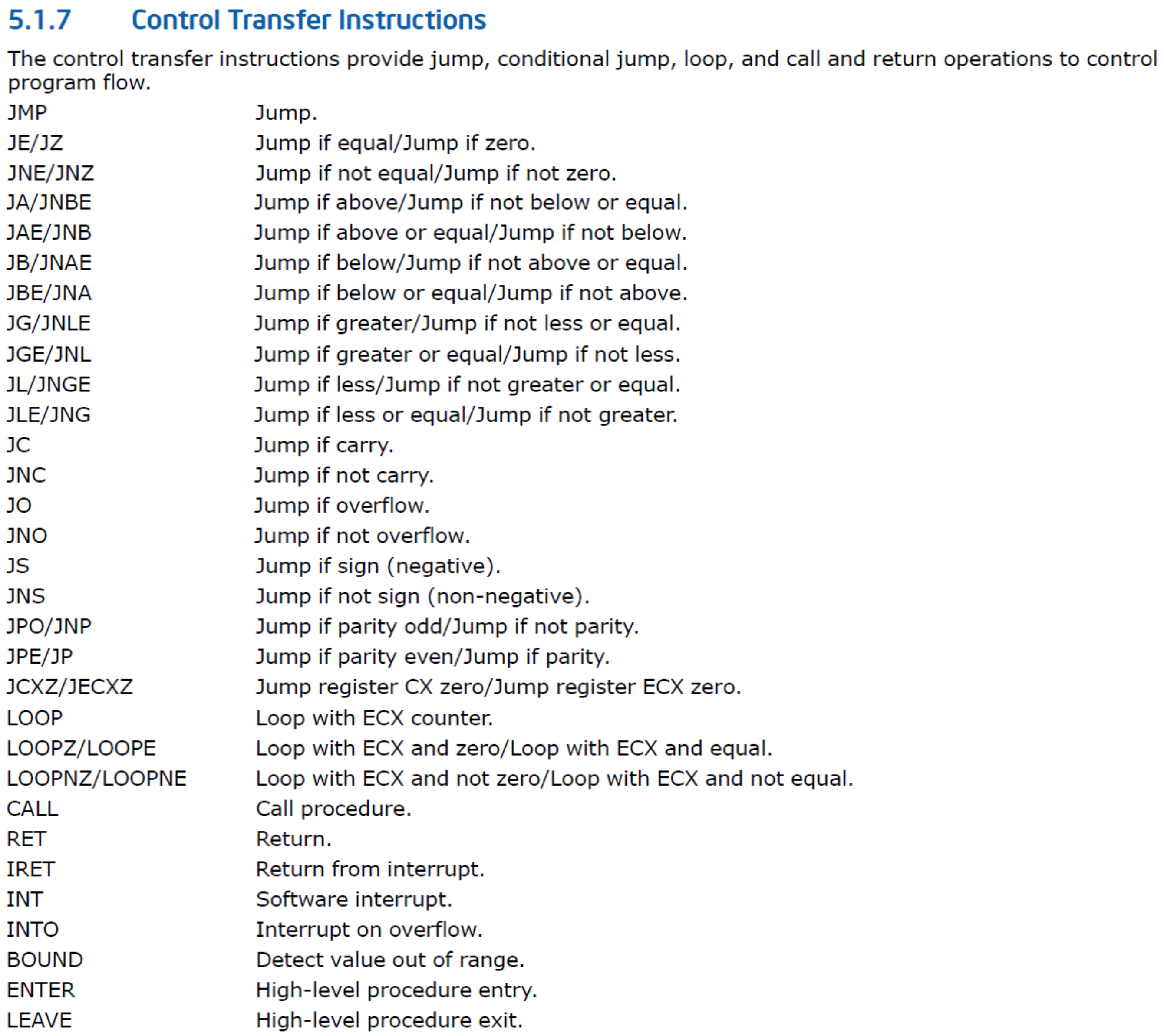
SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.
SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

5.1.6 Bit and Byte Instructions

Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.
BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.
TEST	Logical compare.
CRC32[1]	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT[2]	This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

X86-64 Instruction Set Architecture

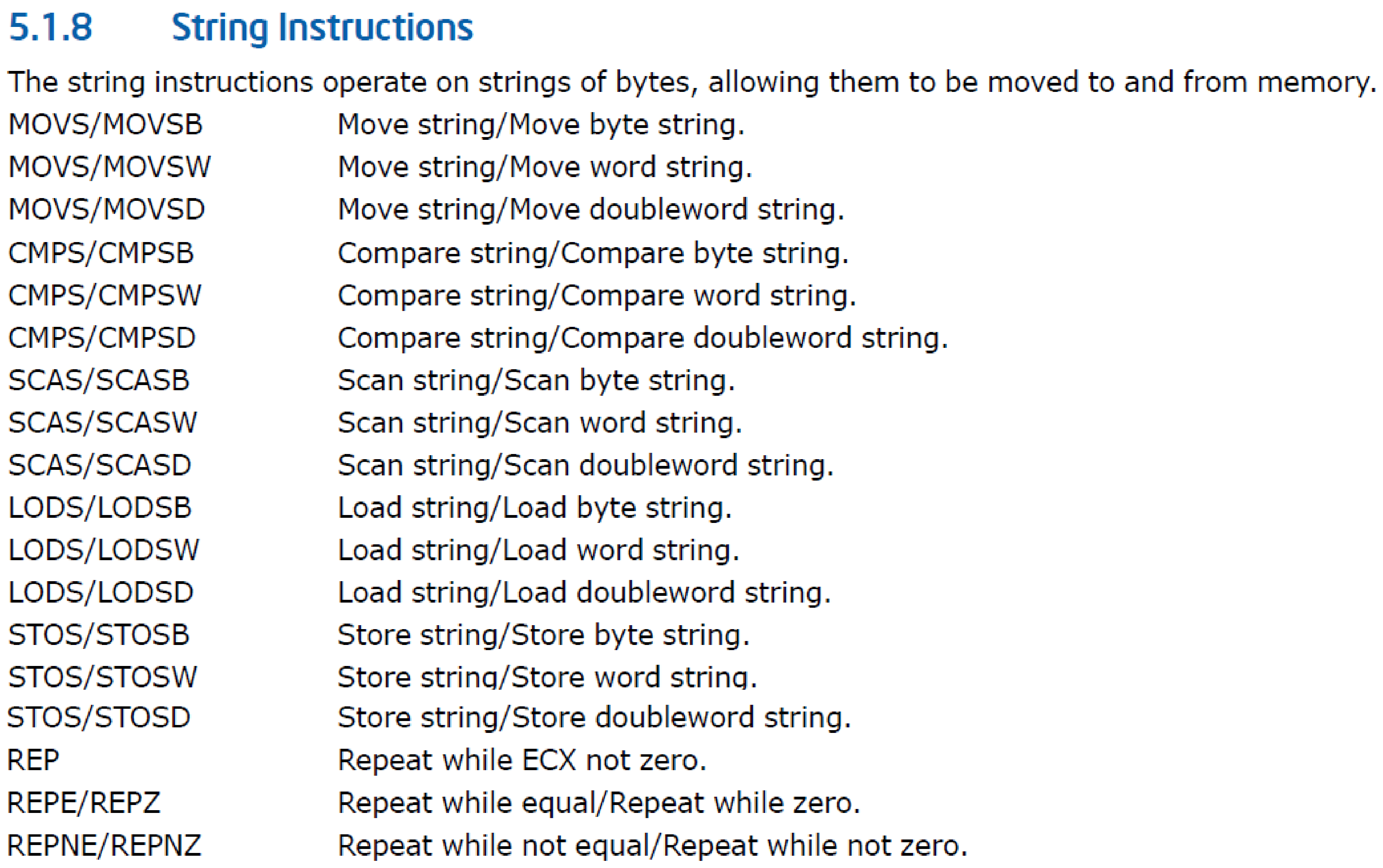


5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump.
JE/JZ	Jump if equal/Jump if zero.
JNE/JNZ	Jump if not equal/Jump if not zero.
JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JG/JNLE	Jump if greater/Jump if not less or equal.
JGE/JNL	Jump if greater or equal/Jump if not less.
JL/JNGE	Jump if less/Jump if not greater or equal.
JLE/JNG	Jump if less or equal/Jump if not greater.
JC	Jump if carry.
JNC	Jump if not carry.
JO	Jump if overflow.
JNO	Jump if not overflow.
JS	Jump if sign (negative).
JNS	Jump if not sign (non-negative).
JPO/JNP	Jump if parity odd/Jump if not parity.
JPE/JP	Jump if parity even/Jump if parity.
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero.
LOOP	Loop with ECX counter.
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal.
CALL	Call procedure.
RET	Return.
IRET	Return from interrupt.
INT	Software interrupt.
INTO	Interrupt on overflow.
BOUND	Detect value out of range.
ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

X86-64 Instruction Set Architecture

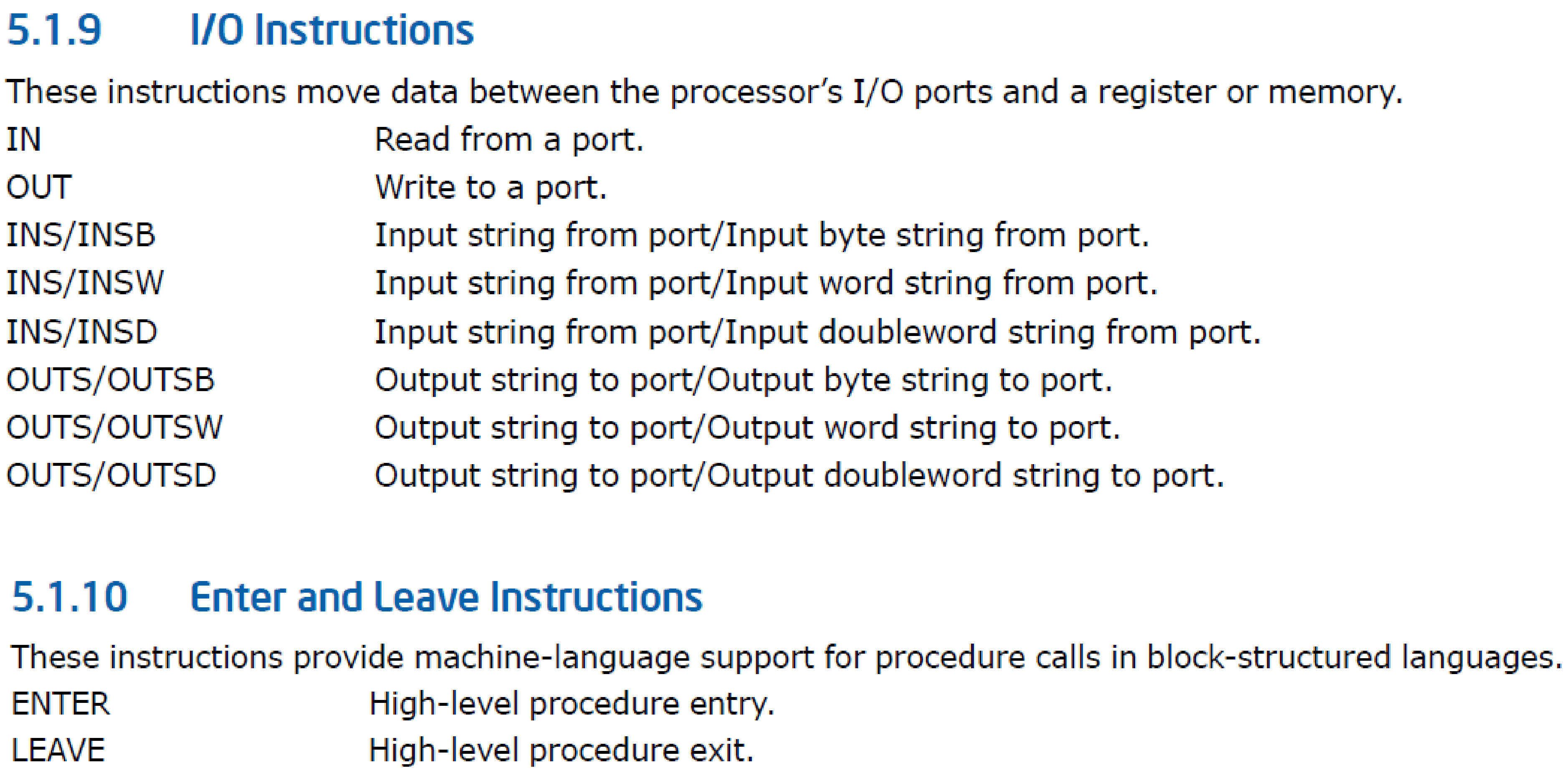


5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVSB	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.
STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPNZ	Repeat while not equal/Repeat while not zero.

X86-64 Instruction Set Architecture



5.1.9 I/O Instructions

These instructions move data between the processor’s I/O ports and a register or memory.

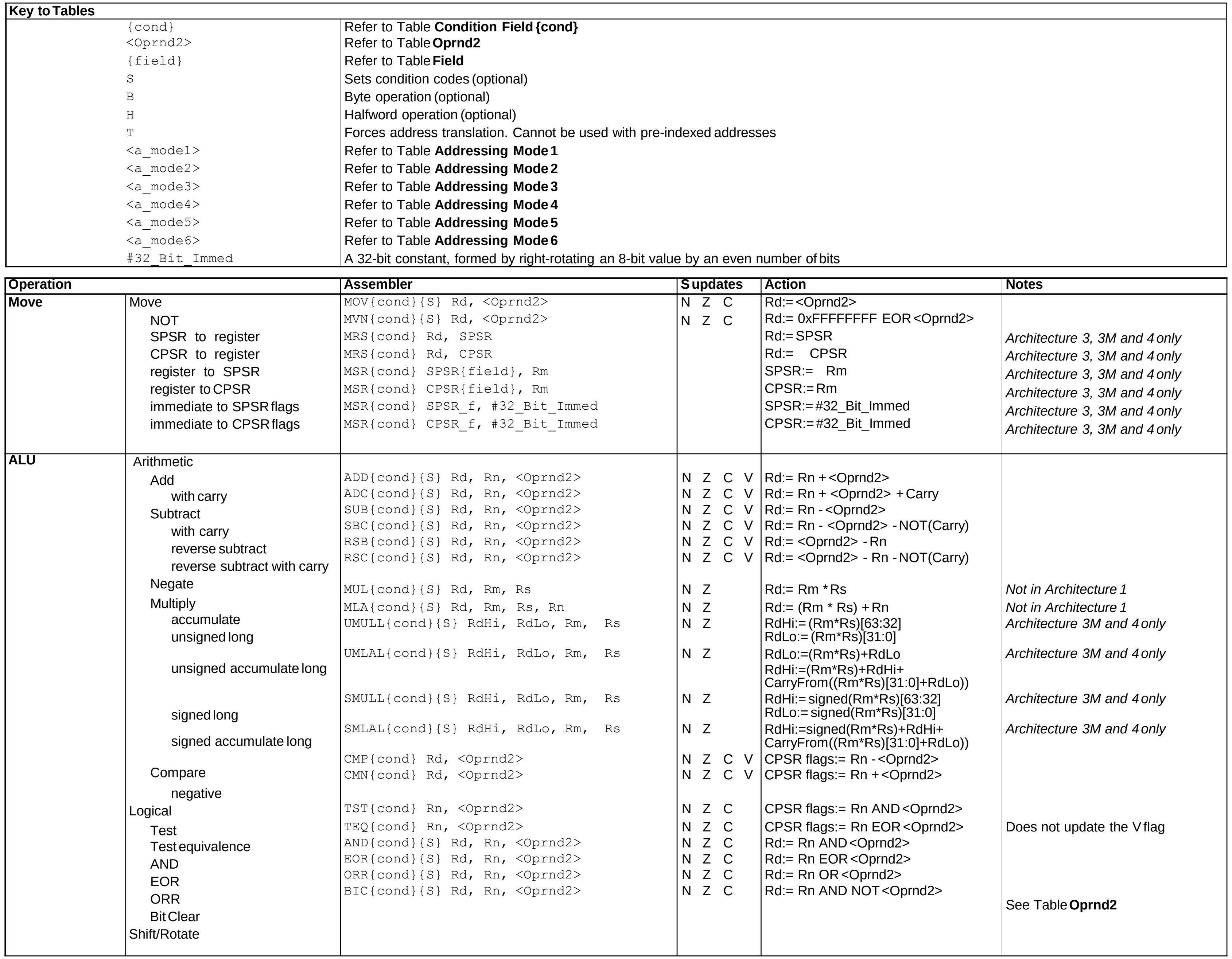
IN	Read from a port.
OUT	Write to a port.
INS/INSB	Input string from port/Input byte string from port.
INS/INSW	Input string from port/Input word string from port.
INS/INSD	Input string from port/Input doubleword string from port.
OUTS/OUTSB	Output string to port/Output byte string to port.
OUTS/OUTSW	Output string to port/Output word string to port.
OUTS/OUTSD	Output string to port/Output doubleword string to port.

5.1.10 Enter and Leave Instructions

These instructions provide machine-language support for procedure calls in block-structured languages.

ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

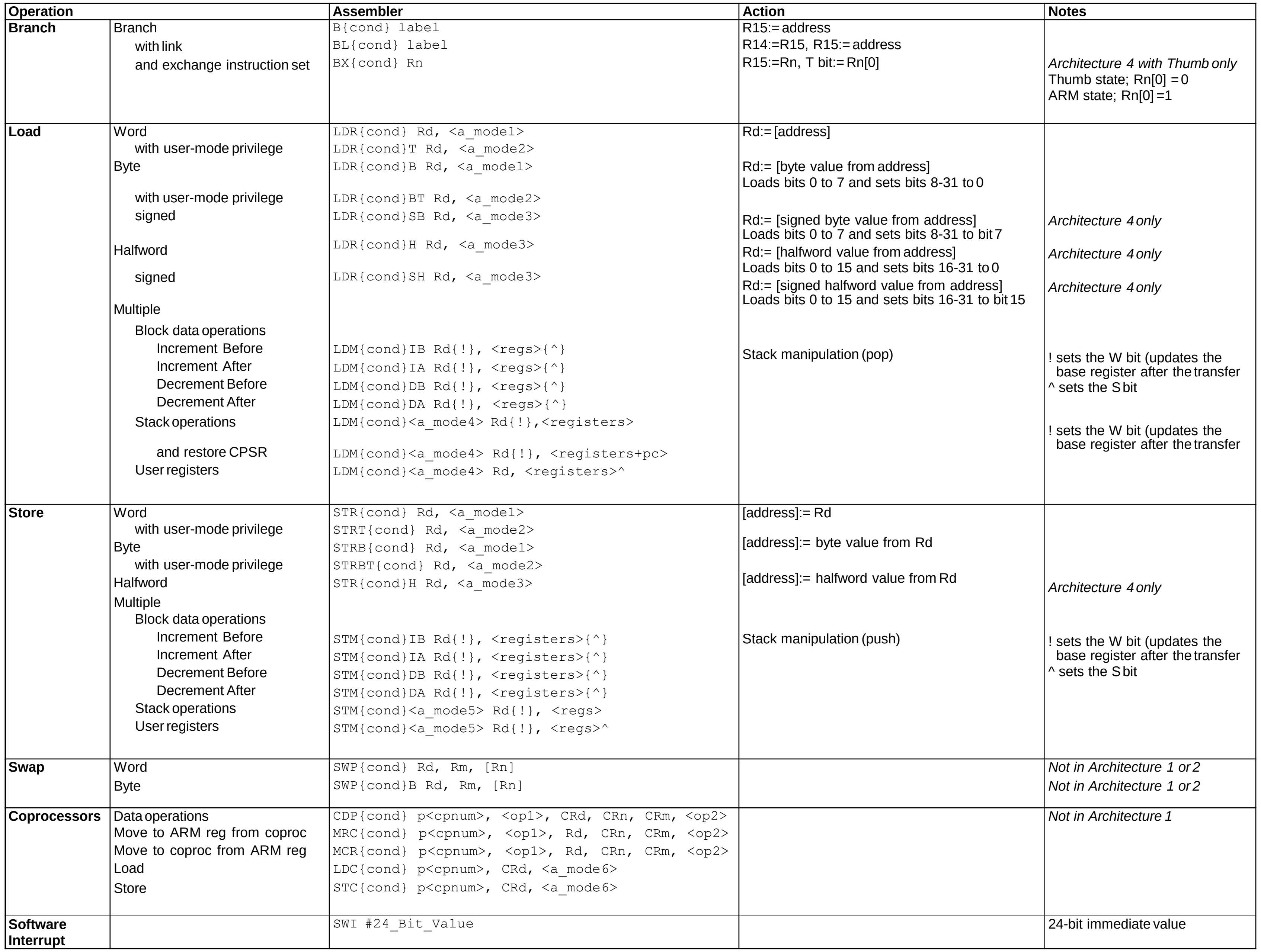
ARM Instruction Set Architecture



Key to Tables	
{cond}	Refer to Table **Condition Field {cond}**
<Oprnd2>	Refer to Table **Oprnd2**
{field}	Refer to Table **Field**
S	Sets condition codes (optional)
B	Byte operation (optional)
H	Halfword operation (optional)
T	Forces address translation. Cannot be used with pre-indexed addresses
<a_mode1>	Refer to Table **Addressing Mode 1**
<a_mode2>	Refer to Table **Addressing Mode 2**
<a_mode3>	Refer to Table **Addressing Mode 3**
<a_mode4>	Refer to Table **Addressing Mode 4**
<a_mode5>	Refer to Table **Addressing Mode 5**
<a_mode6>	Refer to Table **Addressing Mode 6**
#32_Bit_Immed	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits

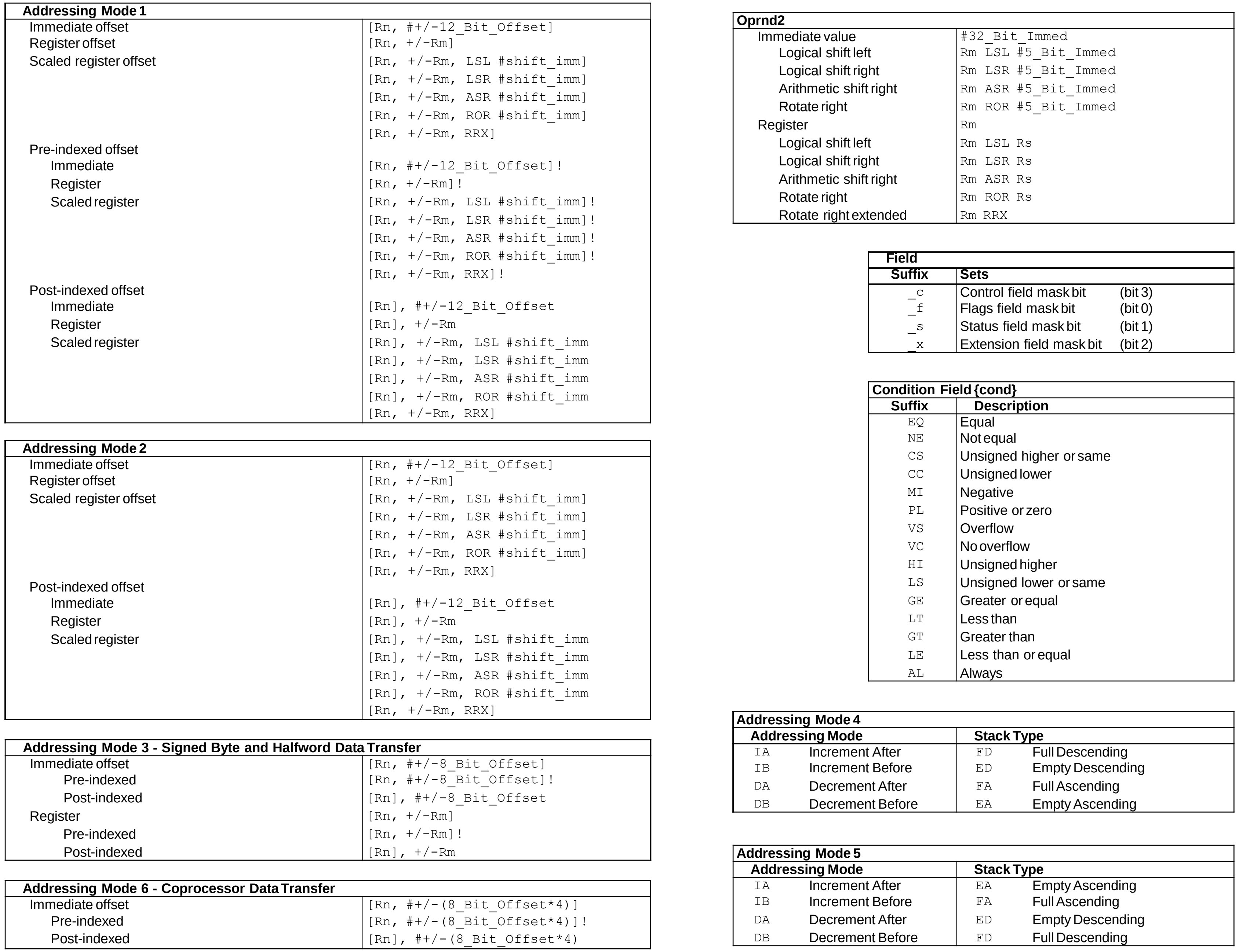
Operation		Assembler	S updates	Action	Notes
Move	Move	MOV{cond}{S} Rd, <Oprnd2>	N Z C	Rd:= <Oprnd2>	
	NOT	MVN{cond}{S} Rd, <Oprnd2>	N Z C	Rd:= 0xFFFFFFFF EOR <Oprnd2>	
	SPSR to register	MRS{cond} Rd, SPSR		Rd:= SPSR	*Architecture 3, 3M and 4 only*
	CPSR to register	MRS{cond} Rd, CPSR		Rd:= CPSR	*Architecture 3, 3M and 4 only*
	register to SPSR	MSR{cond} SPSR{field}, Rm		SPSR:= Rm	*Architecture 3, 3M and 4 only*
	register to CPSR	MSR{cond} CPSR{field}, Rm		CPSR:= Rm	*Architecture 3, 3M and 4 only*
	immediate to SPSR flags	MSR{cond} SPSR_f, #32_Bit_Immed		SPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
	immediate to CPSR flags	MSR{cond} CPSR_f, #32_Bit_Immed		CPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
ALU	Arithmetic				
	Add	ADD{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn + <Oprnd2>	
	with carry	ADC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn + <Oprnd2> + Carry	
	Subtract	SUB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn - <Oprnd2>	
	with carry	SBC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= Rn - <Oprnd2> - NOT(Carry)	
	reverse subtract	RSB{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= <Oprnd2> - Rn	
	reverse subtract with carry	RSC{cond}{S} Rd, Rn, <Oprnd2>	N Z C V	Rd:= <Oprnd2> - Rn - NOT(Carry)	
	Negate				
	Multiply	MUL{cond}{S} Rd, Rm, Rs	N Z	Rd:= Rm * Rs	*Not in Architecture 1*
	accumulate	MLA{cond}{S} Rd, Rm, Rs, Rn	N Z	Rd:= (Rm * Rs) + Rn	*Not in Architecture 1*
	unsigned long	UMULL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:= (Rm*Rs)[63:32] RdLo:= (Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	unsigned accumulate long	UMLAL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdLo:=(Rm*Rs)+RdLo RdHi:=(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	signed long	SMULL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:= signed(Rm*Rs)[63:32] RdLo:= signed(Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	signed accumulate long	SMLAL{cond}{S} RdHi, RdLo, Rm, Rs	N Z	RdHi:=signed(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	Compare	CMP{cond} Rd, <Oprnd2>	N Z C V	CPSR flags:= Rn - <Oprnd2>	
	negative	CMN{cond} Rd, <Oprnd2>	N Z C V	CPSR flags:= Rn + <Oprnd2>	
	Logical				
	Test	TST{cond} Rn, <Oprnd2>	N Z C	CPSR flags:= Rn AND <Oprnd2>	
	Test equivalence	TEQ{cond} Rn, <Oprnd2>	N Z C	CPSR flags:= Rn EOR <Oprnd2>	Does not update the V flag
	AND	AND{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn AND <Oprnd2>	
	EOR	EOR{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn EOR <Oprnd2>	
	ORR	ORR{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn OR <Oprnd2>	
	Bit Clear	BIC{cond}{S} Rd, Rn, <Oprnd2>	N Z C	Rd:= Rn AND NOT <Oprnd2>	
	Shift/Rotate				See Table **Oprnd2**

ARM Instruction Set Architecture



Operation		Assembler	Action	Notes
Branch	Branch	`B{cond} label`	R15:= address	
	with link	`BL{cond} label`	R14:=R15, R15:= address	
	and exchange instruction set	`BX{cond} Rn`	R15:=Rn, T bit:= Rn[0]	*Architecture 4 with Thumb only*
				Thumb state; Rn[0] = 0
				ARM state; Rn[0] = 1
Load	Word	`LDR{cond} Rd, <a_mode1>`	Rd:= [address]	
	with user-mode privilege	`LDR{cond}T Rd, <a_mode2>`		
	Byte	`LDR{cond}B Rd, <a_mode1>`	Rd:= [byte value from address]	
			Loads bits 0 to 7 and sets bits 8-31 to 0	
	with user-mode privilege	`LDR{cond}BT Rd, <a_mode2>`		
	signed	`LDR{cond}SB Rd, <a_mode3>`	Rd:= [signed byte value from address]	*Architecture 4 only*
			Loads bits 0 to 7 and sets bits 8-31 to bit 7	
	Halfword	`LDR{cond}H Rd, <a_mode3>`	Rd:= [halfword value from address]	*Architecture 4 only*
			Loads bits 0 to 15 and sets bits 16-31 to 0	
	signed	`LDR{cond}SH Rd, <a_mode3>`	Rd:= [signed halfword value from address]	*Architecture 4 only*
			Loads bits 0 to 15 and sets bits 16-31 to bit 15	
	Multiple			
	Block data operations			
	Increment Before	`LDM{cond}IB Rd{!}, <regs>{^}`	Stack manipulation (pop)	! sets the W bit (updates the
	Increment After	`LDM{cond}IA Rd{!}, <regs>{^}`		base register after the transfer
	Decrement Before	`LDM{cond}DB Rd{!}, <regs>{^}`		^ sets the S bit
	Decrement After	`LDM{cond}DA Rd{!}, <regs>{^}`		
	Stack operations	`LDM{cond}<a_mode4> Rd{!},<registers>`		! sets the W bit (updates the
				base register after the transfer
	and restore CPSR	`LDM{cond}<a_mode4> Rd{!}, <registers+pc>`		
	User registers	`LDM{cond}<a_mode4> Rd, <registers>^`		
Store	Word	`STR{cond} Rd, <a_mode1>`	[address]:= Rd	
	with user-mode privilege	`STRT{cond} Rd, <a_mode2>`		
	Byte	`STRB{cond} Rd, <a_mode1>`	[address]:= byte value from Rd	
	with user-mode privilege	`STRBT{cond} Rd, <a_mode2>`		
	Halfword	`STR{cond}H Rd, <a_mode3>`	[address]:= halfword value from Rd	*Architecture 4 only*
	Multiple			
	Block data operations			
	Increment Before	`STM{cond}IB Rd{!}, <registers>{^}`	Stack manipulation (push)	! sets the W bit (updates the
	Increment After	`STM{cond}IA Rd{!}, <registers>{^}`		base register after the transfer
	Decrement Before	`STM{cond}DB Rd{!}, <registers>{^}`		^ sets the S bit
	Decrement After	`STM{cond}DA Rd{!}, <registers>{^}`		
	Stack operations	`STM{cond}<a_mode5> Rd{!}, <regs>`		
	User registers	`STM{cond}<a_mode5> Rd{!}, <regs>^`		
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
	Byte	`SWP{cond}B Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`		*Not in Architecture 1*
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode6>`		
	Store	`STC{cond} p<cpnum>, CRd, <a_mode6>`		
Software Interrupt		`SWI #24_Bit_Value`		24-bit immediate value

ARM Instruction Set Architecture



Addressing Mode 1	
Immediate offset	[Rn, #+/-12_Bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #shift_imm]
	[Rn, +/-Rm, LSR #shift_imm]
	[Rn, +/-Rm, ASR #shift_imm]
	[Rn, +/-Rm, ROR #shift_imm]
	[Rn, +/-Rm, RRX]
Pre-indexed offset	
Immediate	[Rn, #+/-12_Bit_Offset]!
Register	[Rn, +/-Rm]!
Scaled register	[Rn, +/-Rm, LSL #shift_imm]!
	[Rn, +/-Rm, LSR #shift_imm]!
	[Rn, +/-Rm, ASR #shift_imm]!
	[Rn, +/-Rm, ROR #shift_imm]!
	[Rn, +/-Rm, RRX]!
Post-indexed offset	
Immediate	[Rn], #+/-12_Bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #shift_imm
	[Rn], +/-Rm, LSR #shift_imm
	[Rn], +/-Rm, ASR #shift_imm
	[Rn], +/-Rm, ROR #shift_imm
	[Rn, +/-Rm, RRX]

Addressing Mode 2	
Immediate offset	[Rn, #+/-12_Bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #shift_imm]
	[Rn, +/-Rm, LSR #shift_imm]
	[Rn, +/-Rm, ASR #shift_imm]
	[Rn, +/-Rm, ROR #shift_imm]
	[Rn, +/-Rm, RRX]
Post-indexed offset	
Immediate	[Rn], #+/-12_Bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #shift_imm
	[Rn], +/-Rm, LSR #shift_imm
	[Rn], +/-Rm, ASR #shift_imm
	[Rn], +/-Rm, ROR #shift_imm
	[Rn, +/-Rm, RRX]

Addressing Mode 3 - Signed Byte and Halfword Data Transfer	
Immediate offset	[Rn, #+/-8_Bit_Offset]
Pre-indexed	[Rn, #+/-8_Bit_Offset]!
Post-indexed	[Rn], #+/-8_Bit_Offset
Register	[Rn, +/-Rm]
Pre-indexed	[Rn, +/-Rm]!
Post-indexed	[Rn], +/-Rm

Addressing Mode 6 - Coprocessor Data Transfer	
Immediate offset	[Rn, #+/-(8_Bit_Offset*4)]
Pre-indexed	[Rn, #+/-(8_Bit_Offset*4)]!
Post-indexed	[Rn], #+/-(8_Bit_Offset*4)

Oprnd2	
Immediate value	#32_Bit_Immed
Logical shift left	Rm LSL #5_Bit_Immed
Logical shift right	Rm LSR #5_Bit_Immed
Arithmetic shift right	Rm ASR #5_Bit_Immed
Rotate right	Rm ROR #5_Bit_Immed
Register	Rm
Logical shift left	Rm LSL Rs
Logical shift right	Rm LSR Rs
Arithmetic shift right	Rm ASR Rs
Rotate right	Rm ROR Rs
Rotate right extended	Rm RRX

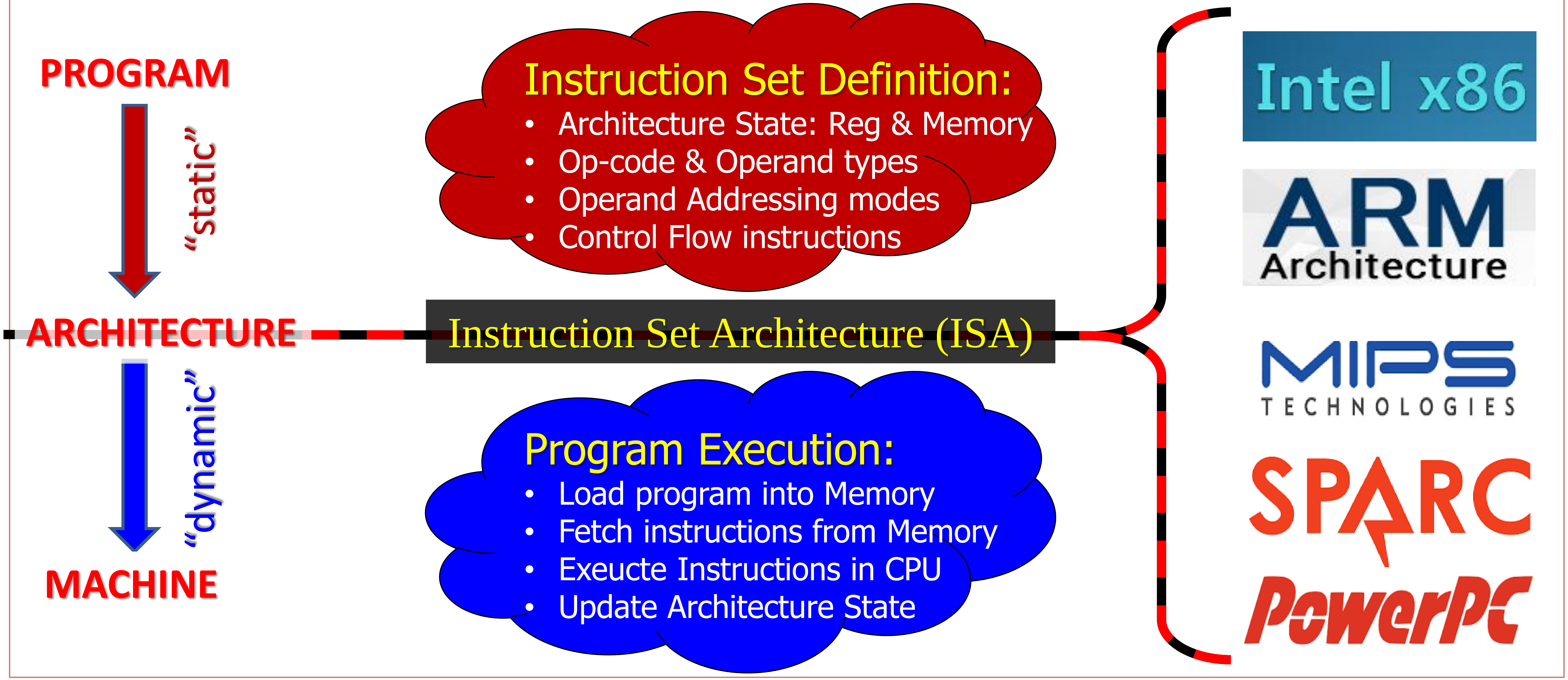
Field		
Suffix	**Sets**	
_c	Control field mask bit	(bit 3)
_f	Flags field mask bit	(bit 0)
_s	Status field mask bit	(bit 1)
_x	Extension field mask bit	(bit 2)

Condition Field {cond}	
Suffix	**Description**
EQ	Equal
NE	Not equal
CS	Unsigned higher or same
CC	Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Greater or equal
LT	Less than
GT	Greater than
LE	Less than or equal
AL	Always

Addressing Mode 4			
Addressing Mode		**Stack Type**	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending

Addressing Mode 5			
Addressing Mode		**Stack Type**	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Commercially Successful ISA's



Instruction Set Definition:

- Architecture State: Reg & Memory
- Op-code & Operand types
- Operand Addressing modes
- Control Flow instructions

Instruction Set Architecture (ISA)

Program Execution:

- Load program into Memory
- Fetch instructions from Memory
- Exeucte Instructions in CPU
- Update Architecture State

Intel x86

ARM Architecture

MIPS TECHNOLOGIES

SPARC

PowerPC

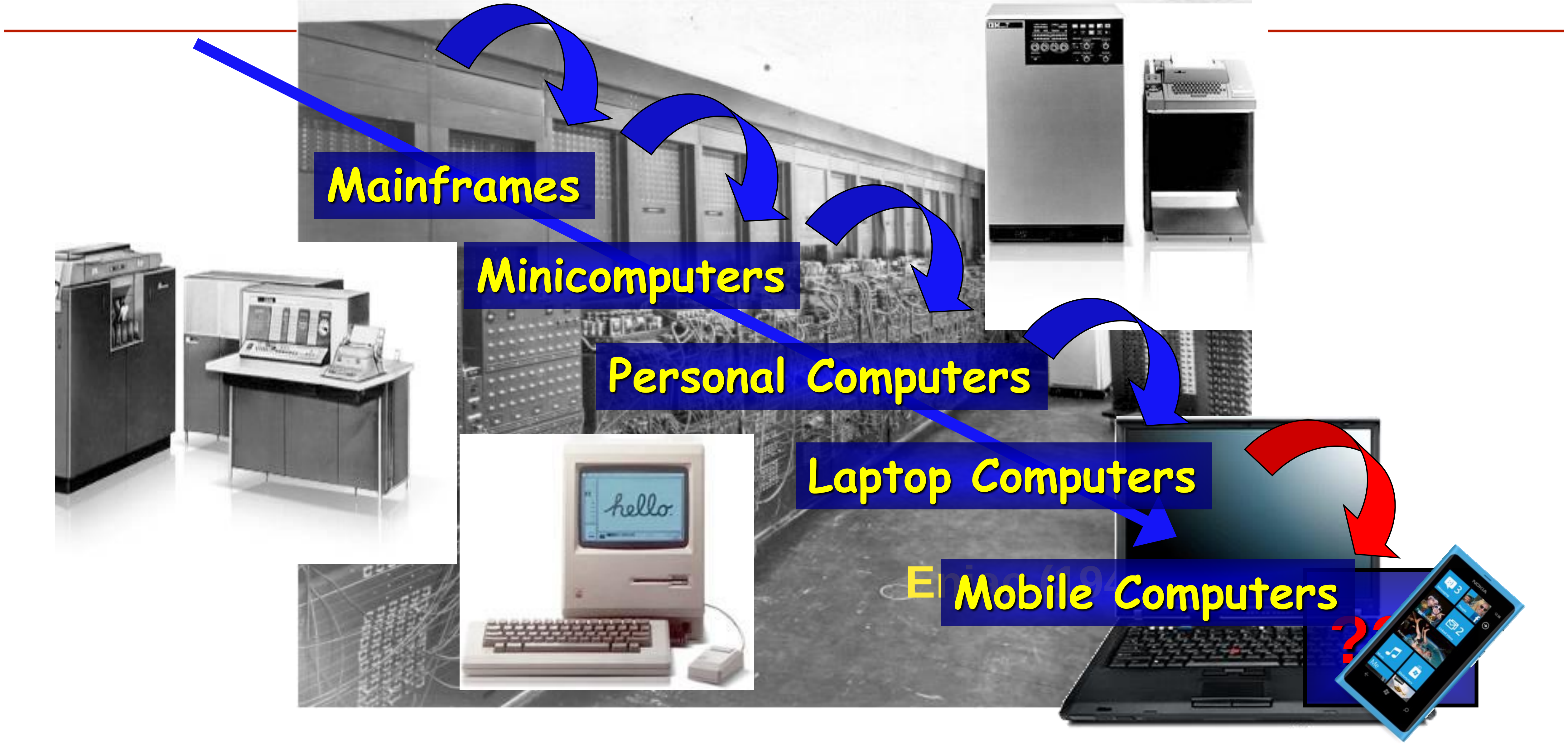
Lecture 2: “Computer Systems: The Big Picture”

2. Historical Perspective on Computing

a. Major Epochs of Modern Computers

b. Computer Performance Iron Law (#1)

Seven Decades of Modern Computing . . .



Mainframes

Minicomputers

Personal Computers

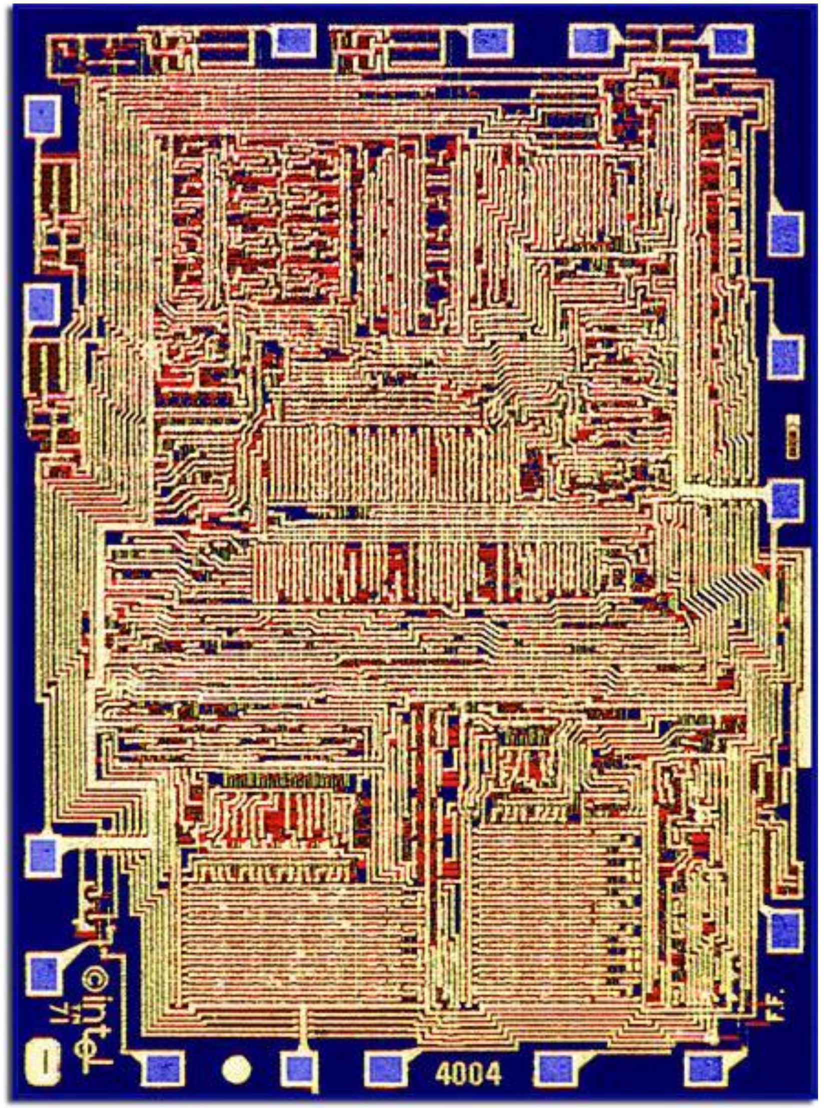
Laptop Computers

Mobile Computers

Historical Perspective on the Last Five Decades

- The Decade of the 1960's: *"Computer Architecture Foundations"*
  - Von Neumann computation model, programming languages, compilers, OS's
  - Commercial Mainframe computers, Scientific numerical computers
- The Decade of the 1970's: *"Birth of Microprocessors"*
  - Programmable controllers, bit-sliced ALU's, single-chip processors
  - Emergence of Personal Computers (PC)
- The Decade of the 1980's: *"Quantitative Architecture"*
  - Instruction pipelining, fast cache memories, compiler considerations
  - Widely available Minicomputers, emergence of Personal Workstations
- The Decade of the 1990's: *"Instruction-Level Parallelism"*
  - Superscalar, speculative microarchitectures, aggressive compiler optimizations
  - Widely available low-cost desktop computers, emergence of Laptop computers
- The Decade of the 2000's: *"Mobile Computing Convergence"*
  - Multi-core architectures, system-on-chip integration, power constrained designs
  - Convergence of smartphones and laptops, emergence of Tablet computers

Intel 4004, circa 1971

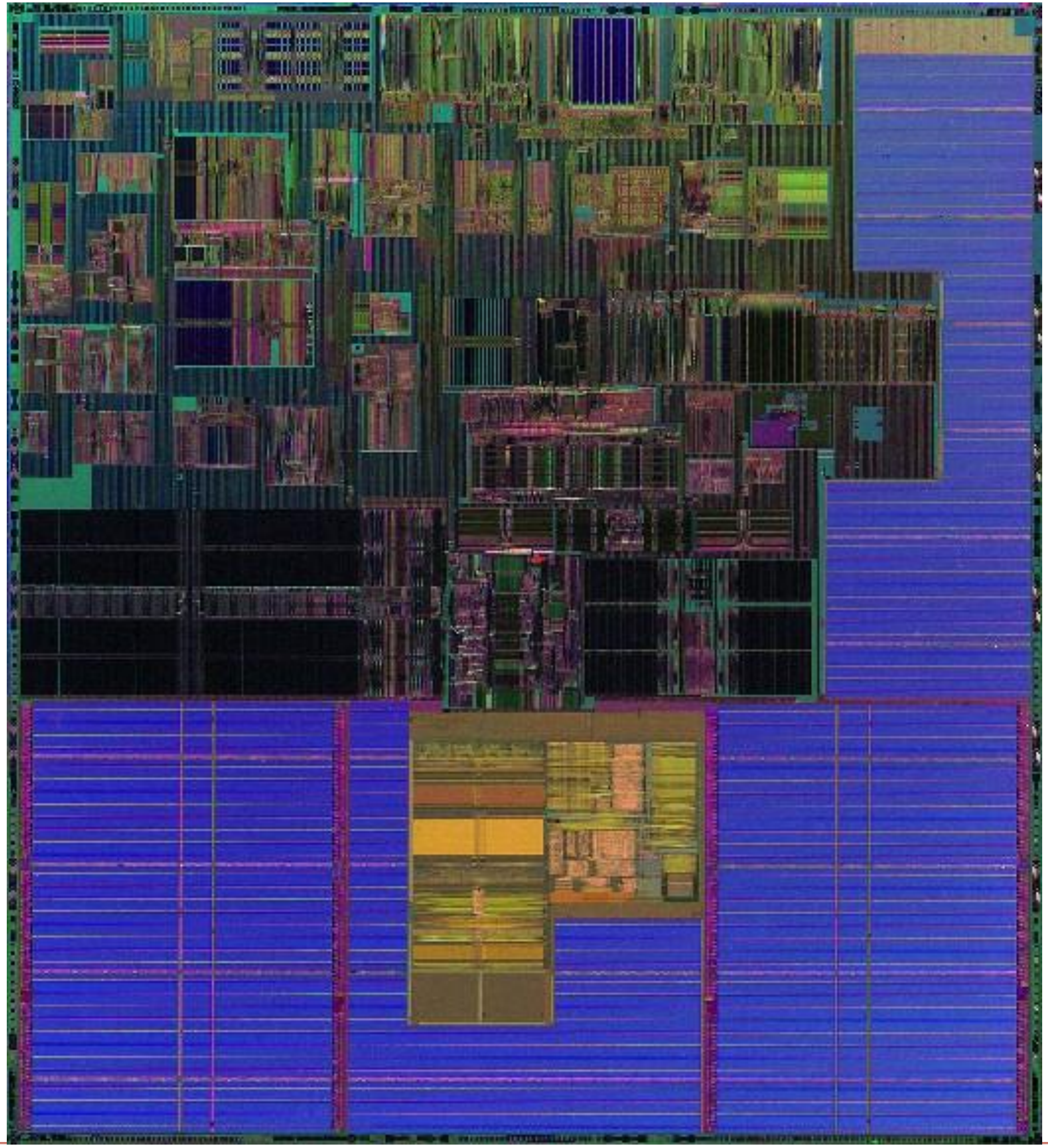


The first single chip CPU

- 4-bit processor for a calculator.
- 1K data memory
- 4K program memory
- 2,300 transistors
- 16-pin DIP package
- 740kHz (eight clock cycles per CPU cycle of 10.8 microseconds)
- ~100K OPs per second

Molecular Expressions: Chipshots

Intel Itanium 2, circa 2002



Performance leader in floating-point apps

- 64-bit processor
- 3 MByte in cache!!
- 221 million transistor
- 1 GHz, issue up to 8 instructions per cycle

In ~30 years, about 100,000 fold growth in transistor count!

http://cpus.hp.com/images/die_photos/McKinley_die.jpg

[John Crawford, Intel, 1993]

Performance Growth in Perspective

- Doubling every 18 months (1982-2000):
  - total of 3,200X
  - Cars travel at 176,000 MPH; get 64,000 miles/gal.
  - Air travel: L.A. to N.Y. in 5.5 seconds (MACH 3200)
  - Wheat yield: 320,000 bushels per acre
- Doubling every 24 months (1971-2001):
  - total of 36,000X
  - Cars travel at 2,400,000 MPH; get 600,000 miles/gal.
  - Air travel: L.A. to N.Y. in 0.5 seconds (MACH 36,000)
  - Wheat yield: 3,600,000 bushels per acre

Unmatched by any other industry!!

Convergence of Key Enabling Technologies

- CMOS VLSI:
  - Submicron feature sizes: 0.3u → 0.25u → 0.18u → 0.13u → 90n → 65n → 45n → 32nm…
  - Metal layers: 3 → 4 → 5 → 6 → 7 (copper) → 12 …
  - Power supply voltage: 5V → 3.3V → 2.4V → 1.8V → 1.3V → 1.1V …
- CAD Tools:
  - Interconnect simulation and critical path analysis
  - Clock signal propagation analysis
  - Process simulation and yield analysis/learning
- Microarchitecture:
  - Superpipelined and superscalar machines
  - Speculative and dynamic microarchitectures
  - Simulation tools and emulation systems
- Compilers:
  - Extraction of instruction-level parallelism
  - Aggressive and speculative code scheduling
  - Object code translation and optimization

“Iron Law” of Processor Performance

$$1/\text{ComputerPerformance} = \frac{\text{Time}}{\text{Program}}$$

$$= \boxed{\frac{\text{Instructions}}{\text{Program}}} \text{ X } \boxed{\frac{\text{Cycles}}{\text{Instruction}}} \text{ X } \boxed{\frac{\text{Time}}{\text{Cycle}}}$$

(inst. count) (CPI) (cycle time)

Architecture → Implementation → Realization

Compiler Designer Processor Designer Chip Designer

- In the 1980’s (decade of pipelining):
  - CPI: 5.0 → 1.15
- In the 1990’s (decade of superscalar):
  - CPI: 1.15 → 0.5 (best case)
- In the 2000’s:
  - we learn the power lesson
  - ILP → TLP

Iron Law #1 – Processor (Latency) Performance

- Time to execute a program: T (latency)

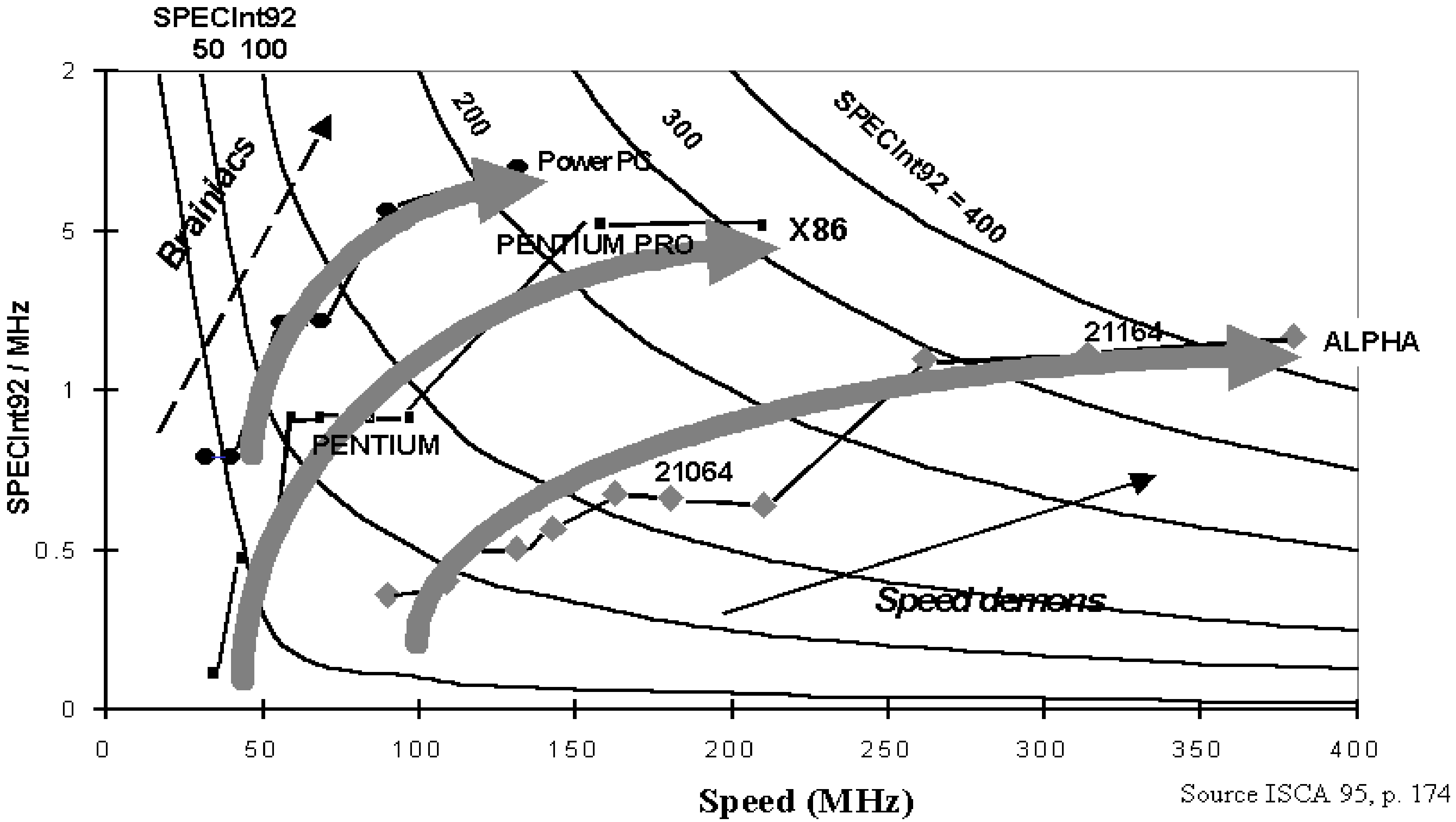
$$T = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle}$$

$$T = PathLength \times CPI \times CycleTime$$

- Processor performance: Perf = 1/T

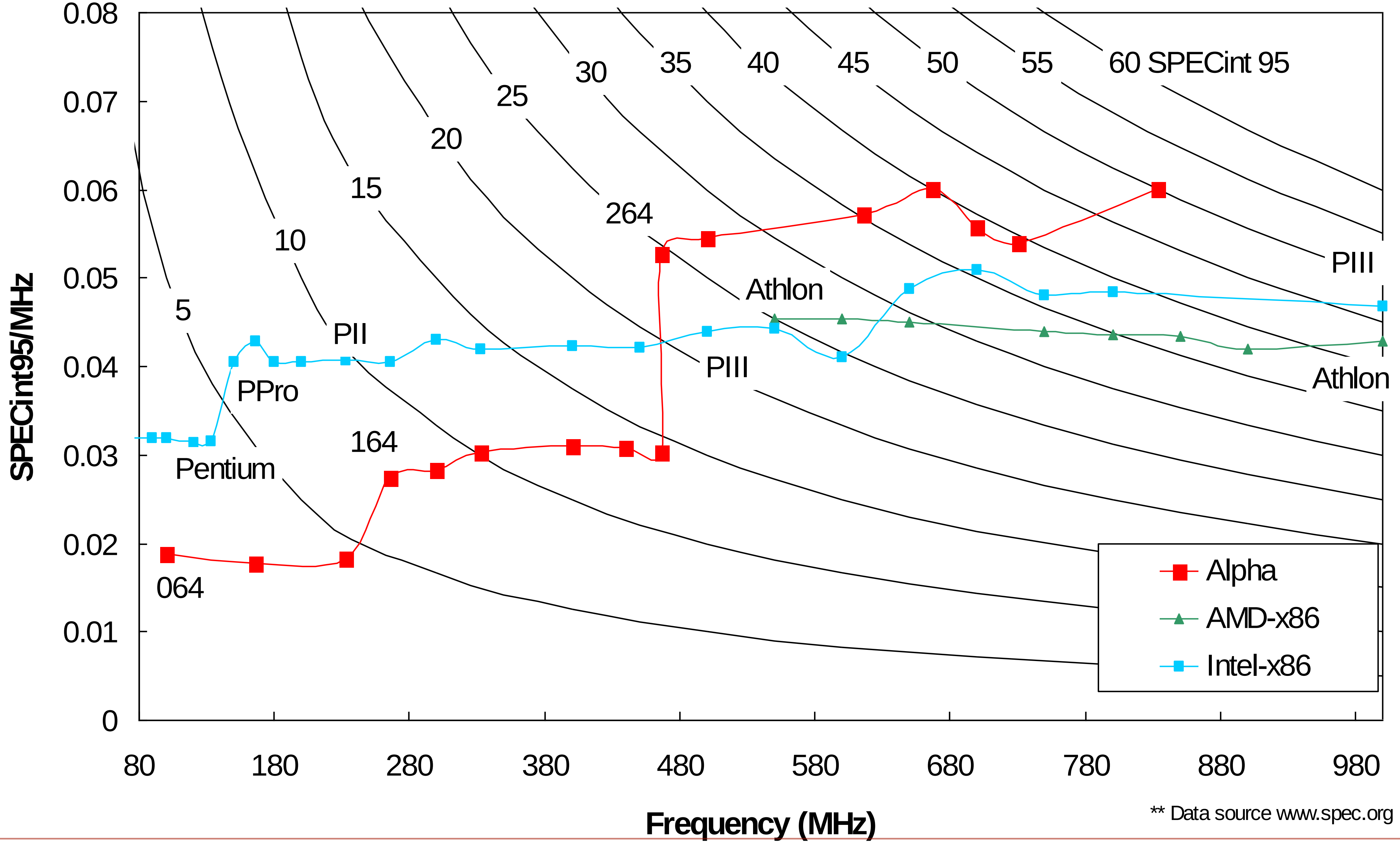
$$Perf_{CPU} = \frac{1}{PathLength \times CPI \times CycleTime} = \boxed{\frac{Frequency}{PathLength \times CPI}}$$

Landscape of Processor Families [SPECint92]



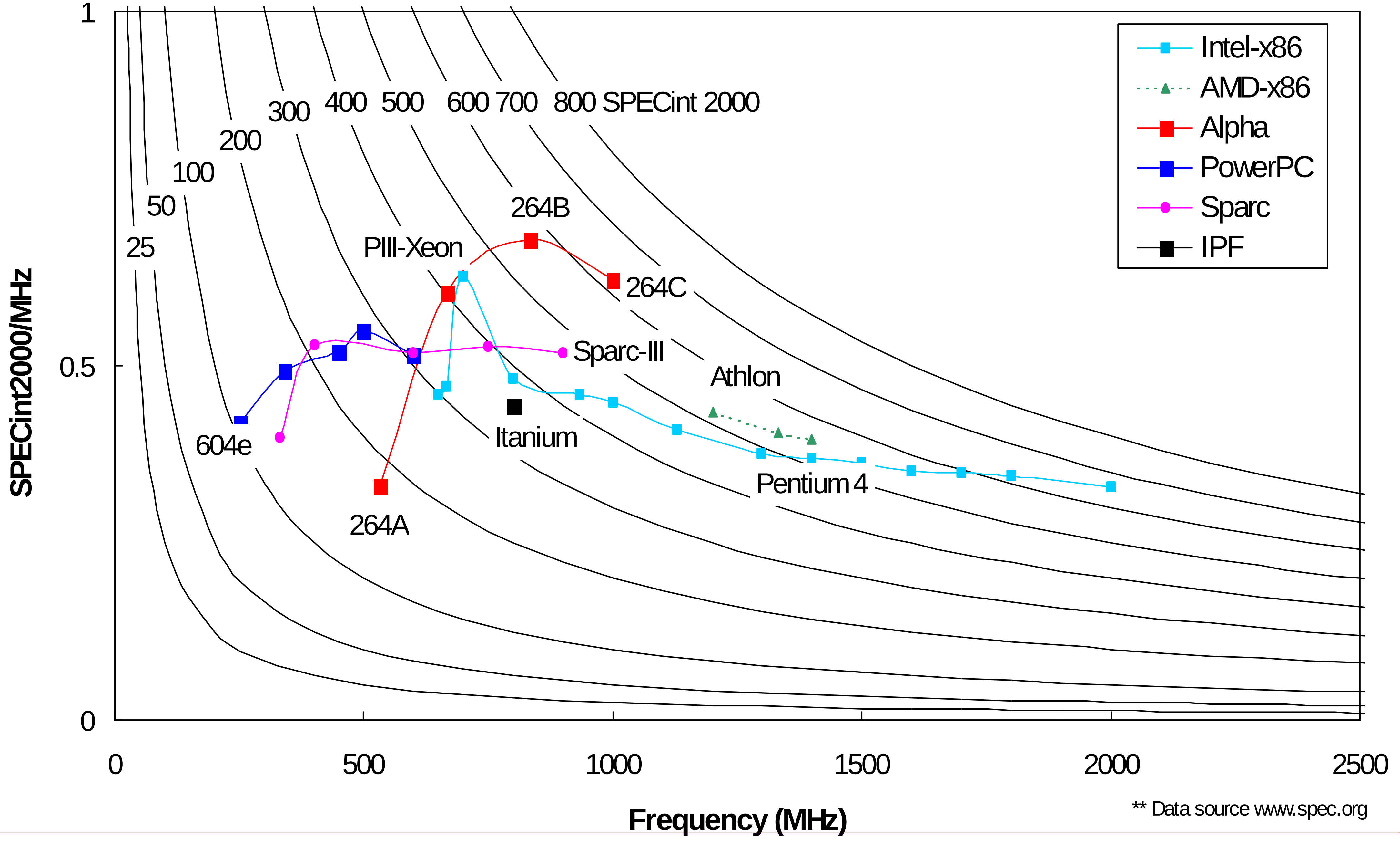
Source ISCA 95, p. 174

Landscape of Processor Families [SPECint95]



** Data source www.spec.org

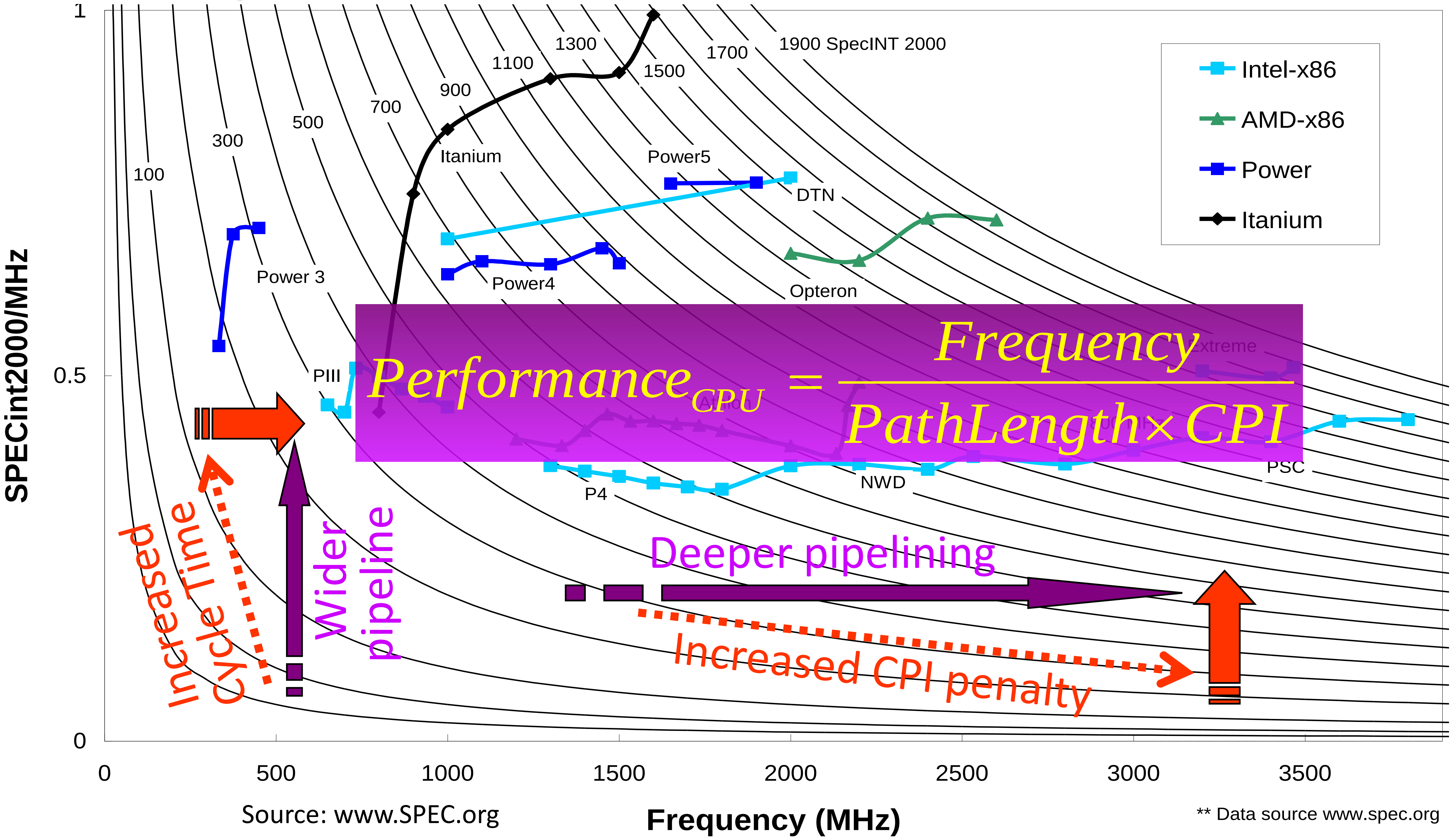
Landscape of Processor Families [SPECint2000]



** Data source www.spec.org

[John DeVale & Bryan Black, 2005]

Landscape of Processor Families [SPECint2000]



$$Performance_{CPU} = \frac{Frequency}{PathLength \times CPI}$$

Increased Cycle Time

Wider pipeline

Deeper pipelining

Increased CPI penalty

Source: www.SPEC.org

** Data source www.spec.org

Lecture 2: "Computer Systems: The Big Picture"

3. "Economics" of Computer Systems

a. Amdahl's Law and Gustafson's Law

b. Moore's Law and Bell's Law

Electrical & Computer ENGINEERING

"Economics" of Computer Architecture

- Exercise in engineering tradeoff analysis
  - Find the fastest/cheapest/power-efficient/etc. solution
  - Optimization problem with 10s to 100s of variables
- All the variables are changing
  - At non-uniform rates
  - With inflection points
  - Only one guarantee: Today's right answer will be wrong tomorrow

➢ Two Persistent high-level "forcing functions":

- ➢ **Application Demand** (PROGRAM)
- ➢ **Technology Supply** (MACHINE)

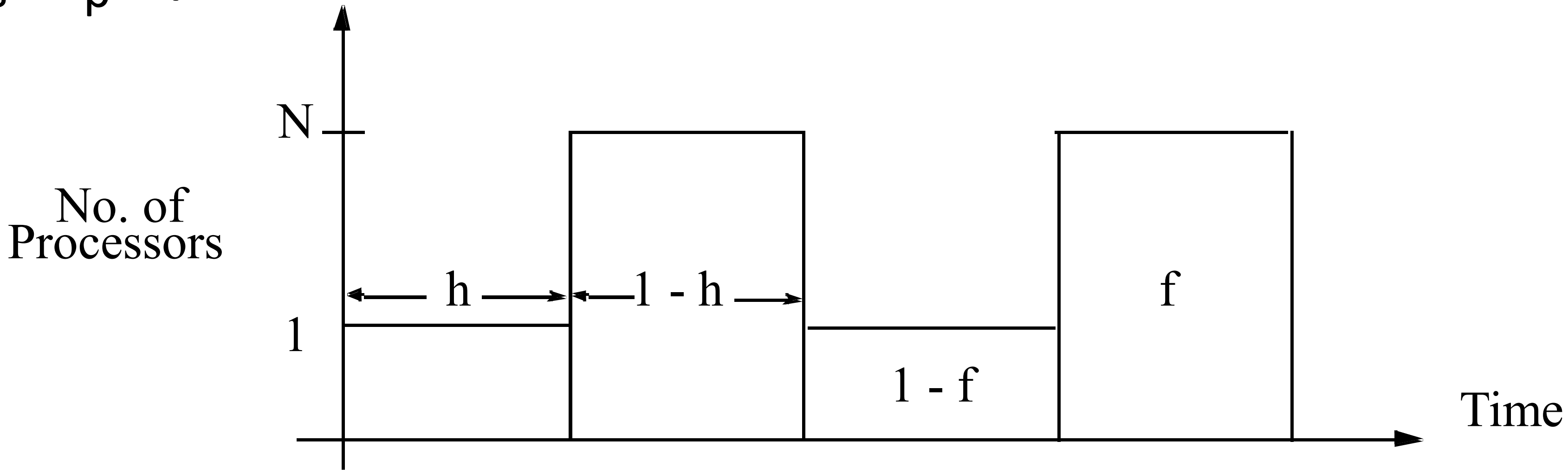
Foundational "Laws" of Computer Architecture

- **Application Demand** (PROGRAM)
  - **Amdahl's Law** (1967)
    - Speedup through parallelism is limited by the sequential bottleneck
  - **Gustafson's Law** (1988)
    - With unlimited data set size, parallelism speedup can be unlimited

- **Technology Supply** (MACHINE)
  - **Moore's Law** (1965)
    - (Transistors/Die) increases by 2x every 18 months
  - **Bell's Law** (1971)
    - (Cost/Computer) decreases by 2x every 36 months

Amdahl’s Law

- **Speedup** = (Execution time on Single CPU)/(Execution on N parallel processors)
  - t_s /t_p (Serial time is for ***best*** *serial algorithm)*

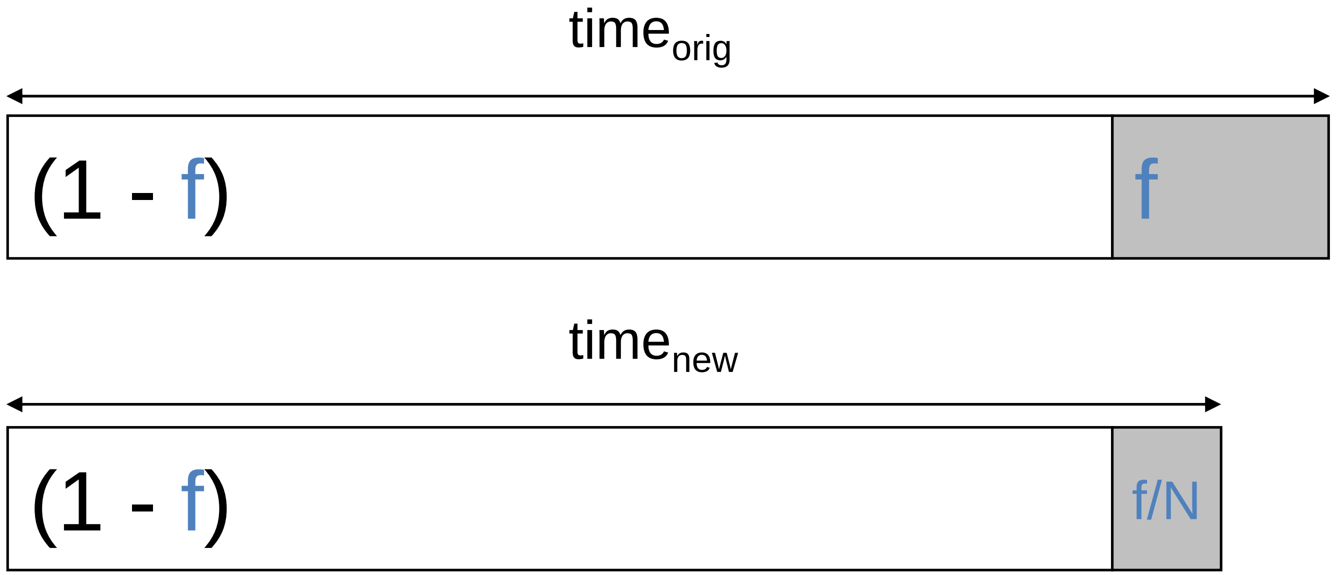


- h = fraction of time in serial code
- f = fraction that is vectorizable or parallelizable
- N = max speedup for f
- Overall speedup → →

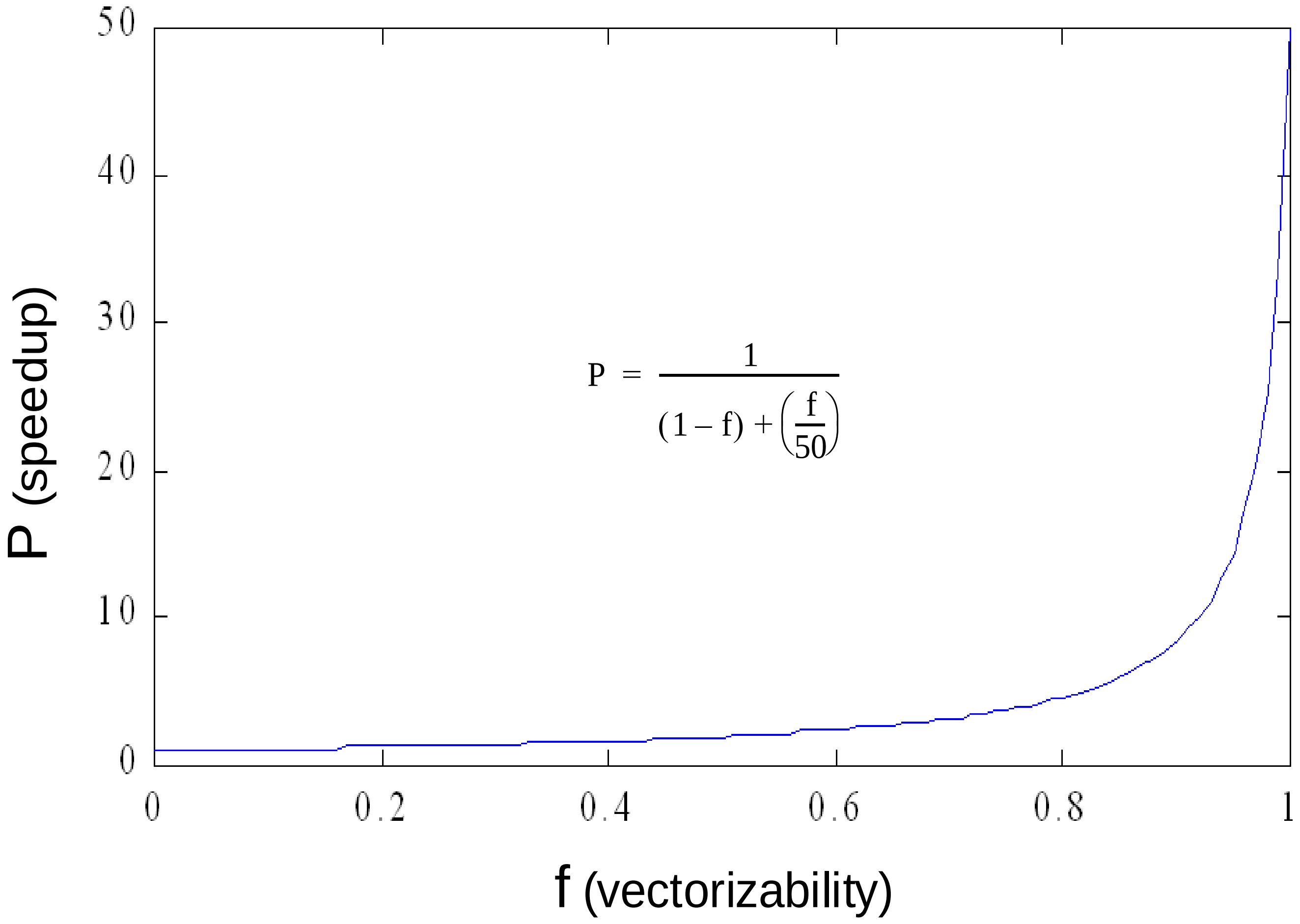
$$\boxed{Speedup = \frac{1}{(1-f)+\dfrac{f}{N}}}$$

Amdahl's Law Illustrated

- Speedup = $\text{time}_{\text{without enhancement}}$ / $\text{time}_{\text{with enhancement}}$
- If an enhancement speeds up a fraction f of a task by a factor of N
- $\text{time}_{\text{new}} = \text{time}_{\text{orig}} \cdot ((1-f) + f/N)$
- $S_{\text{overall}} = 1 / ((1-f) + f/N)$



"Tyranny of Amdahl's Law" [Bob Colwell, CMU-Intel-DARPA]



- Suppose that a computation has a 4% serial portion, what is the limit of speedup on 16 processors?
  - 1/((0.04) + (0.96/16)) = 10
  - What is the maximum speedup?
    - 1/0.04 = 25 (with N → ∞)

From Amdahl's Law to Gustafson's Law

- Amdahl's Law works on a *fixed* problem size
  - This is reasonable if your only goal is to solve a problem faster.
  - What if you also want to solve a larger problem?
    - Gustafson's Law (Scaled Speedup)
- Gustafson's Law is derived by fixing the parallel execution time (Amdahl fixed the problem size -> fixed serial execution time)
  - For many practical situations, Gustafson's law makes more sense
    - Have a bigger computer, solve a bigger problem.
- "Amdahl's Law turns out to be too pessimistic for high-performance computing."

Gustafson's Law

- Fix execution of the computation on a single processor as
  - $s + p$ = serial part + parallelizable part **= 1**
- Speedup(N) = $(s + p)/(s + p/N)$

  $= 1/(s + (1 - s)/N) = 1/((1-p) + p/N)$ ← Amdahl's law

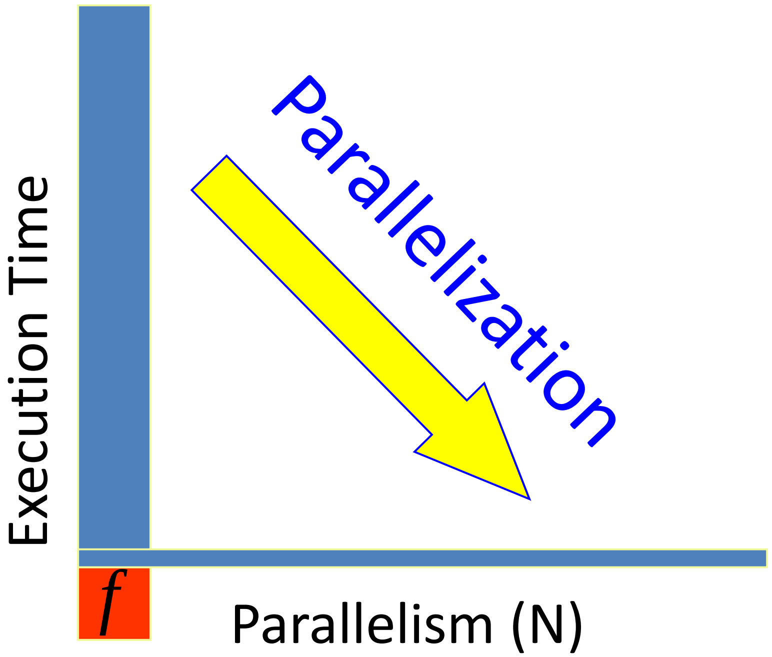
- Now let $1 = (a + b)$ = execution time of computation on N processors (fixed) where a = sequential time and b = parallel time on any of the N processors
  - Time for sequential processing = $a + (b \times N)$ and Speedup = $(a + b \times N)/(a + b)$
  - Let $\alpha = a/(a+b)$ be the sequential fraction of the parallel execution time
  - $\text{Speedup}_{\text{scaled}}(N) = (a + b \times N)/(a + b) = (a/(a+b) + (b \times N)/(a+b)) = \alpha + (1- \alpha)N$
  - If α is very small, the scaled speedup is approximately N, i.e. linear speedup.

Two Laws on Algorithm and Performance

Amdahl's Law

$$Speedup(N)_{MC} = \frac{1}{\left(\frac{f}{1}\right) + \left(\frac{(1-f)}{N}\right)}$$

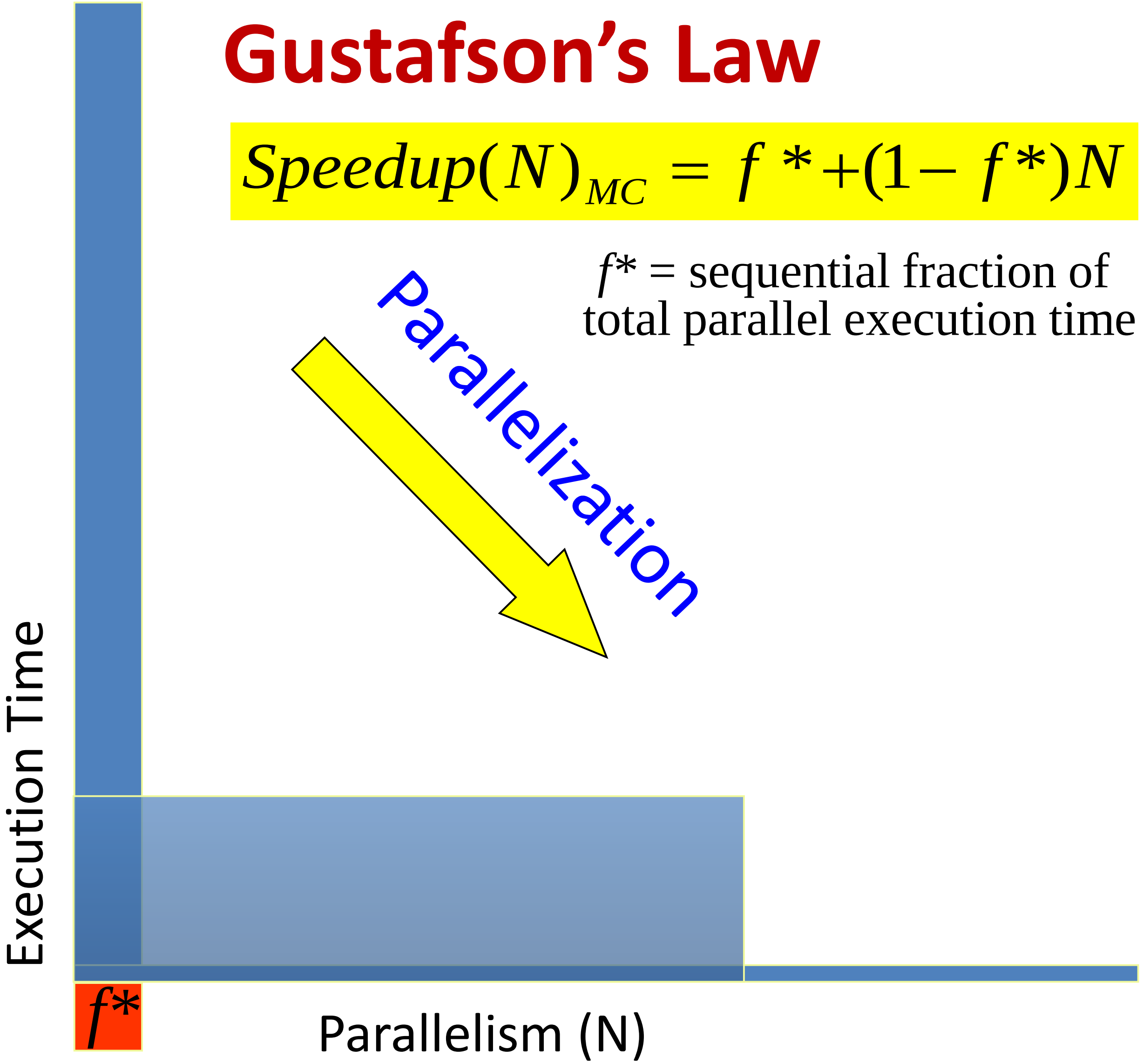
f = sequential %



Gustafson's Law

$$Speedup(N)_{MC} = f^* + (1 - f^*)N$$

f^* = sequential fraction of total parallel execution time



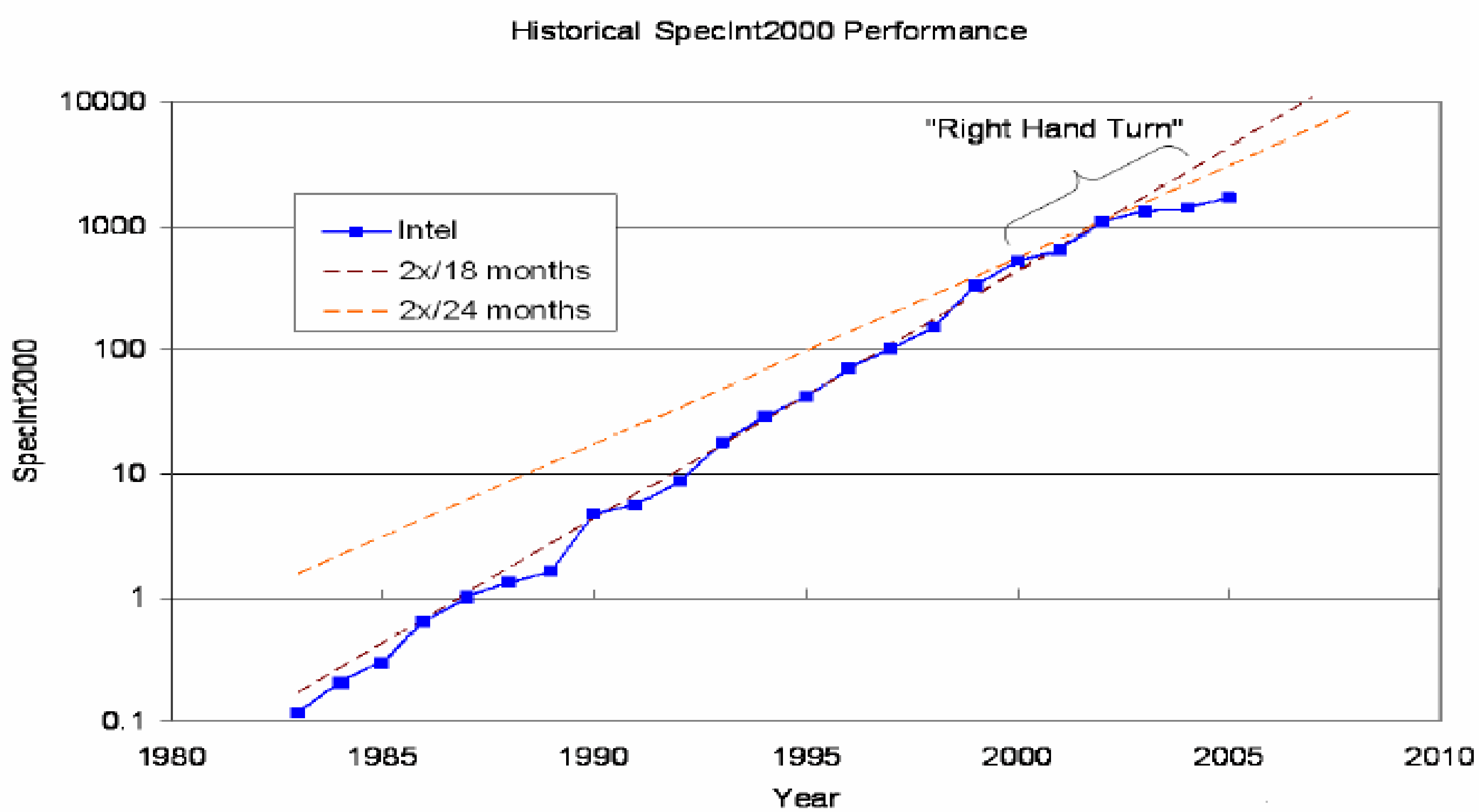
Two "Gordon" Laws of Computer Systems

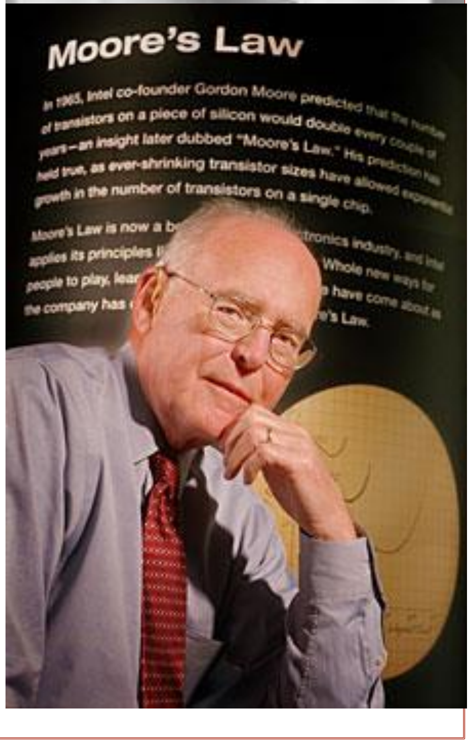
- **Gordon Moore's Law** (1965)
  - (Transistors/Die) increases by 2X every 18 months
  - Constant price, increasing performance
  - Has held for 40+ years, and will continue to hold
- **Gordon Bell's Law** (1971)
  - (Cost/Computer) decreases by 2X every 36 months (~ 10X per decade)
  - Constant performance, decreasing price
  - Corollary of Moore's Law, creation of new computer categories

"In a decade you can buy a computer for less than its sales tax today." – Jim Gray

We have all been living on this exponential curve and assuming it…

Moore’s Law Trends

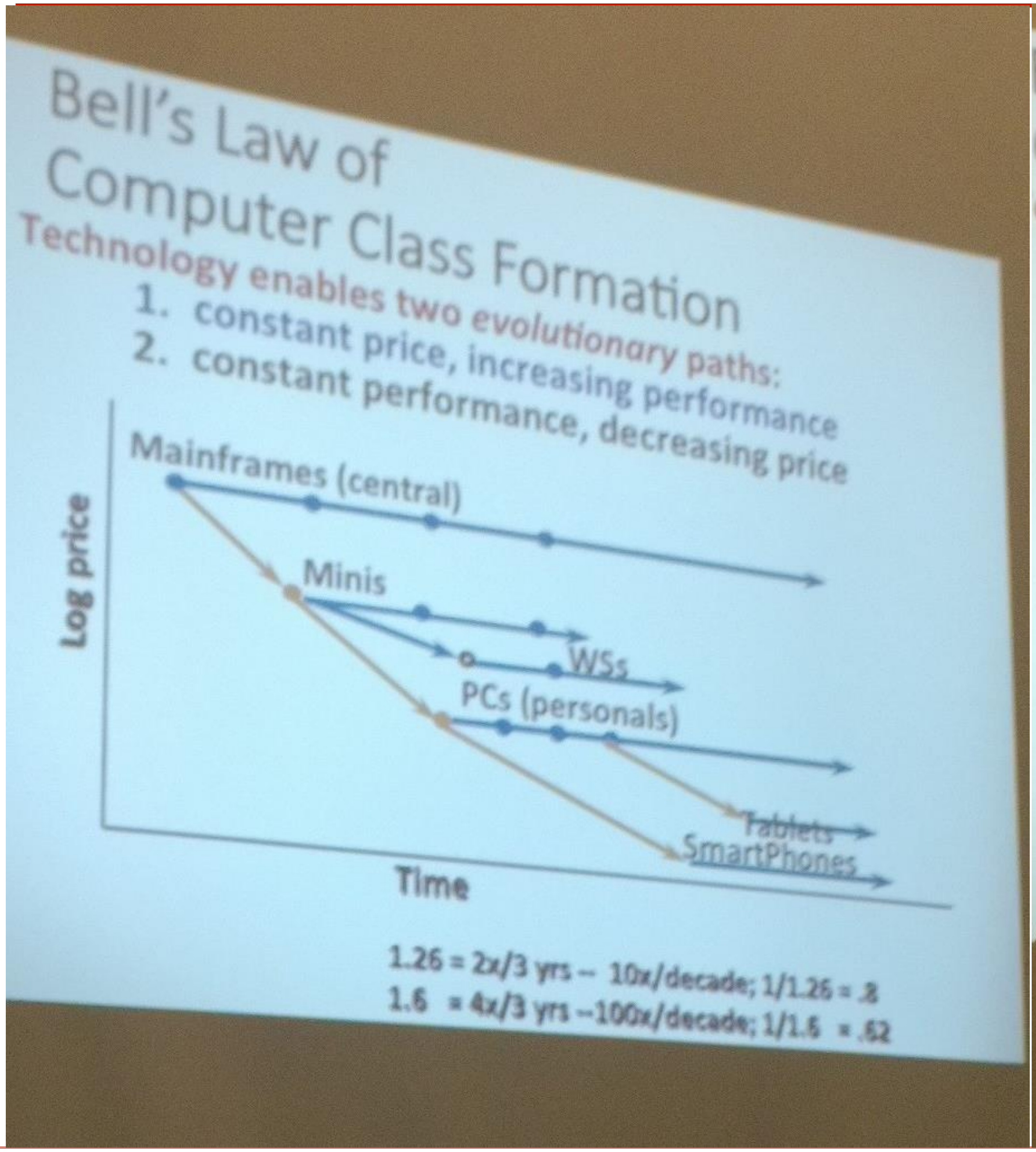




- Moore’s Law for device integration
- Chip power consumption
- Single-thread performance trend

[source: Intel]

Bell's Law Trends



Bell's Law of Computer Class Formation

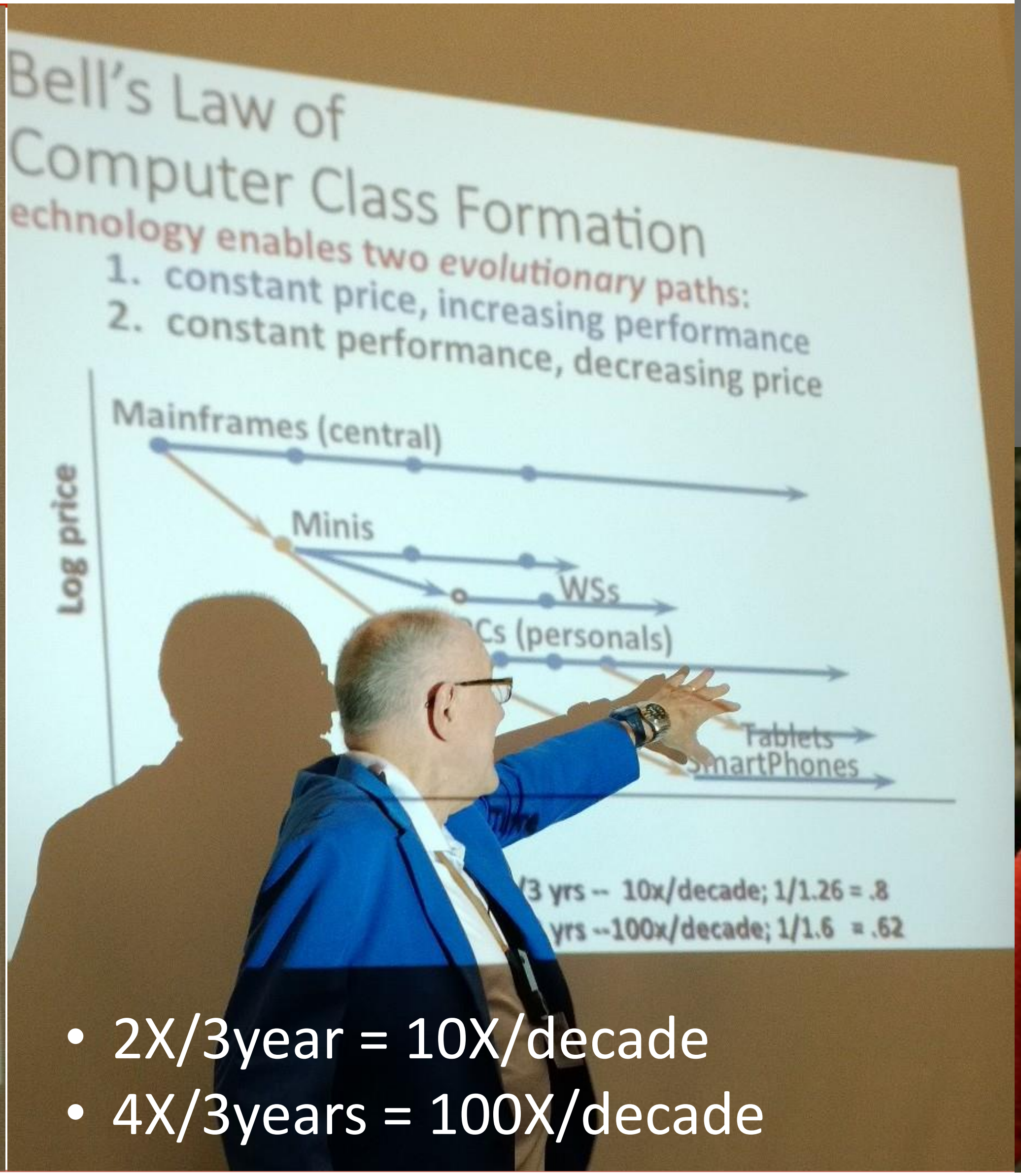
Technology enables two *evolutionary* paths:

1. constant price, increasing performance
2. constant performance, decreasing price

Log price
Mainframes (central)
Minis
WSs
PCs (personals)
Tablets
SmartPhones
Time

1.26 = 2x/3 yrs -- 10x/decade; 1/1.26 = .8

1.6 = 4x/3 yrs --100x/decade; 1/1.6 = .62



Bell's Law of Computer Class Formation

echnology enables two *evolutionary* paths:

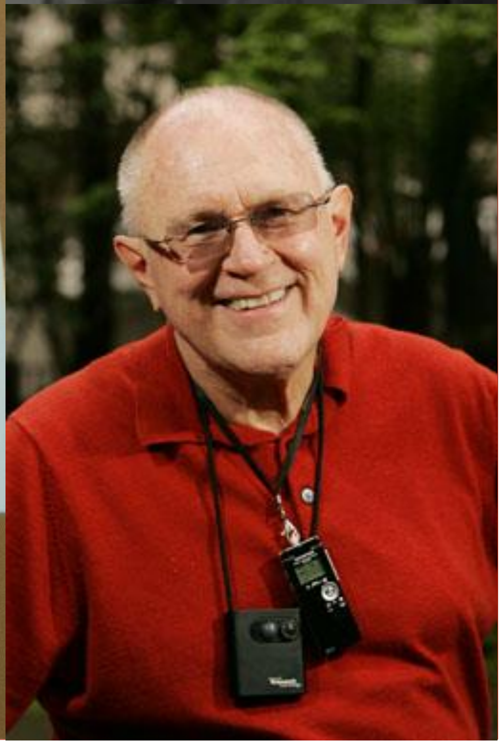
1. constant price, increasing performance
2. constant performance, decreasing price

Log price
Mainframes (central)
Minis
WSs
Cs (personals)
Tablets
SmartPhones

3 yrs -- 10x/decade; 1/1.26 = .8

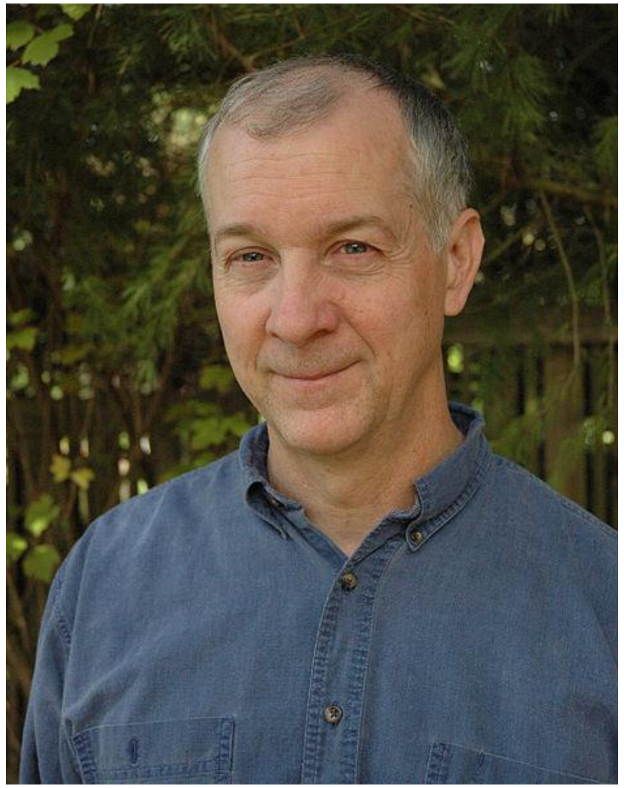
yrs --100x/decade; 1/1.6 = .62

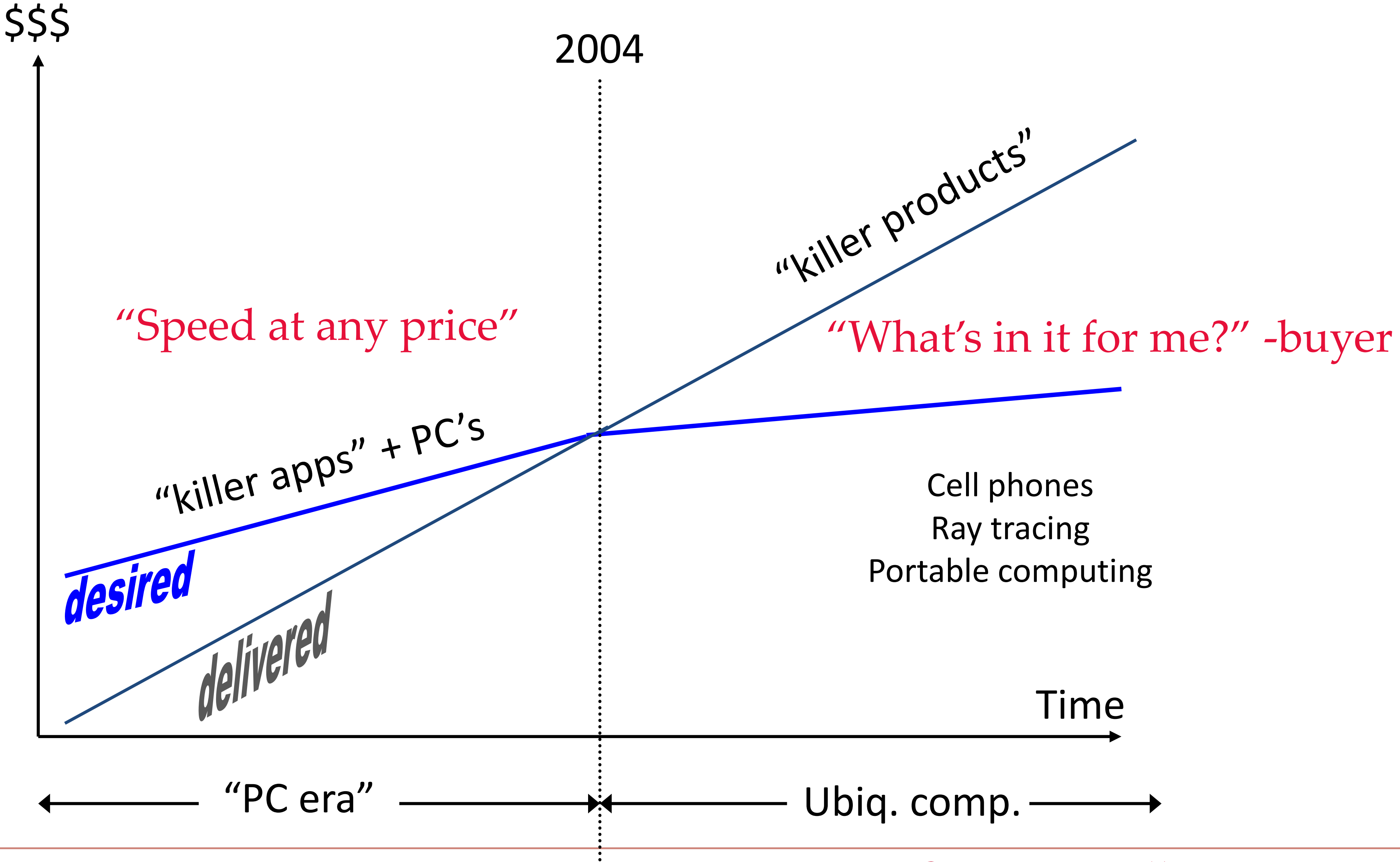
- 2X/3year = 10X/decade
- 4X/3years = 100X/decade



[Bob Colwell CRA Grand Challenges panel 2005]

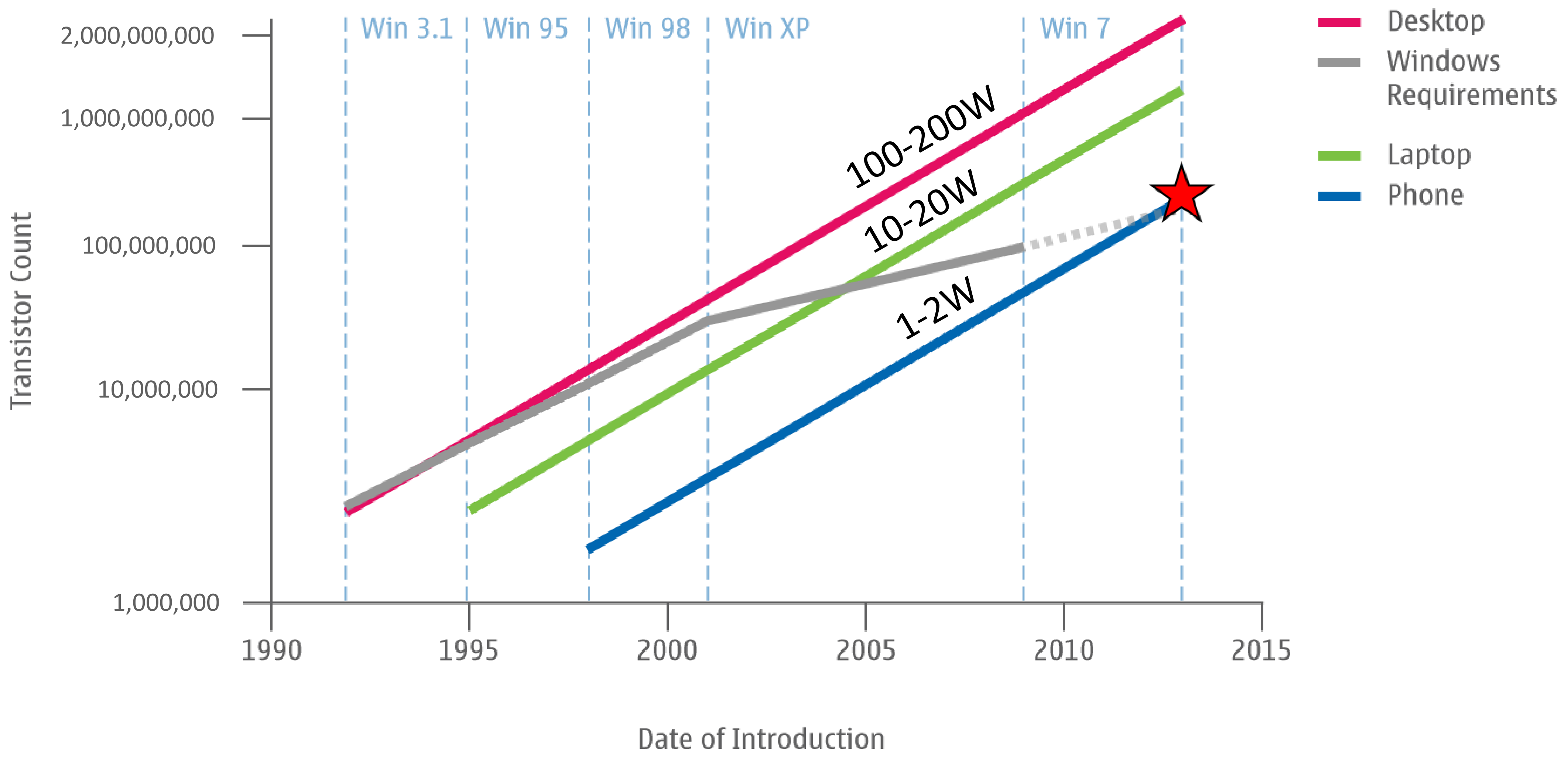
Know Your "Supply & Demand Curves"





Moore’s Law and Bell’s Law are Alive and Well

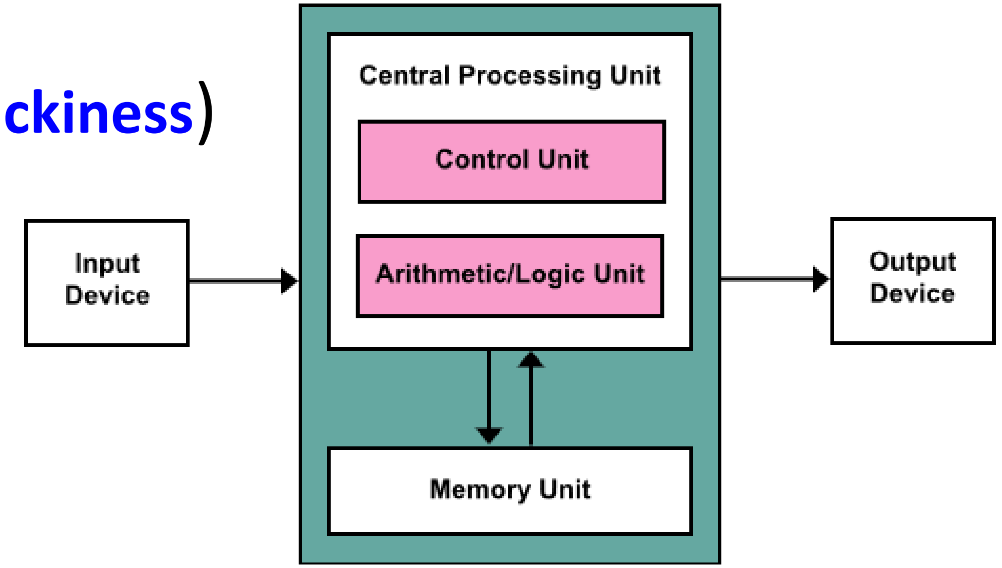
Cross Over Point in 2013 !!!



18-600 Course Coverage: Processor Architecture

Persistence of Von Neumann Model (**Legacy SW Stickiness**)

1. One CPU
2. Monolithic Memory
3. Sequential Execution Semantics



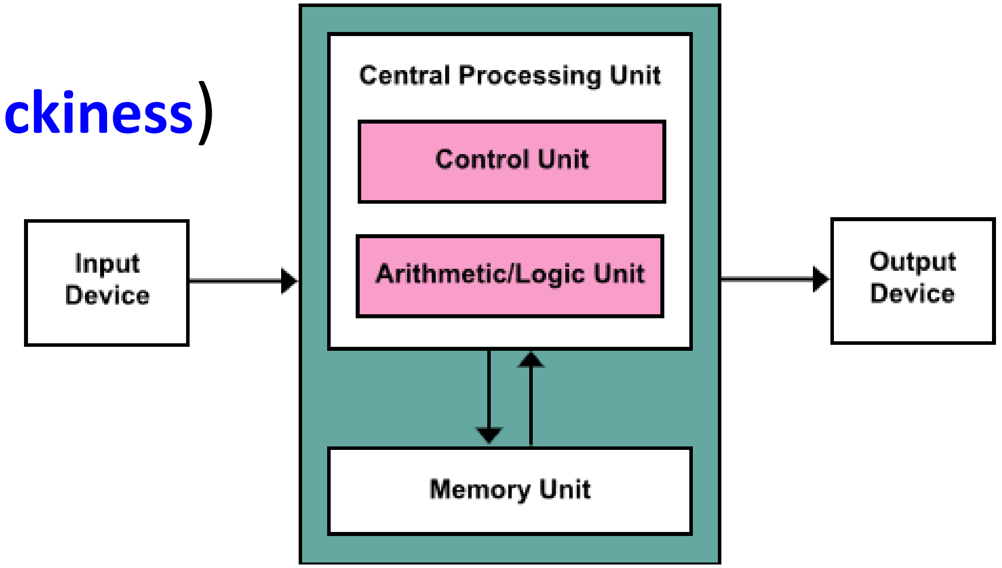
Evolution of Von Neumann Implementations:

➢ SP:	Sequential Processors	(direct implementation of sequential execution)
➢ PP:	Pipelined Processors	(overlapped execution of in-order instructions)
➢ SSP:	Superscalar Processors	(out-of-order execution of multiple instructions)
➢ MCP:	Multi-core Processors = CMP: Chip Multiprocessors	(concurrent multi-threads)
➢ PDS:	Parallel & Distributed Systems	(concurrent multi-threads and multi-programs)

18-600 Course Coverage: Parallelism Exploited

Persistence of Von Neumann Model (**Legacy SW Stickiness**)

1. One CPU
2. Monolithic Memory
3. Sequential Execution Semantics



Parallelisms for **Performance** → for **Power** Reduction → for **Energy** Efficiency

➢ **ILP**:	Basic Block	(exploit ILP in PP, SSP)	
➢ **ILP**:	Loop Iteration	(exploit ILP in SSP, VLIW)	
➢ **DLP**:	Data Set	(exploit DLP in SIMD, GPU)	Parallel Programming
➢ **TLP**:	Task/Thread	(exploit TLP in MCP)	Parallel Programming
➢ **PLP**:	Process/Program	(exploit PLP in MCP, PDS)	Concurrent Programming

18-600 Foundations of Computer Systems

Lecture 3:
“Bits, Bytes, Integers, & Floating Points”

John P. Shen & Gregory Kesden
September 6, 2017

Next Time …

- ➢ Required Reading Assignment:
  - Chapter 2 of CS:APP (3rd edition) by Randy Bryant & Dave O’Hallaron
- ➢ Lab Assignment for This Week:
  - ❖ Lab #1 (Data Lab)

Electrical & Computer ENGINEERING