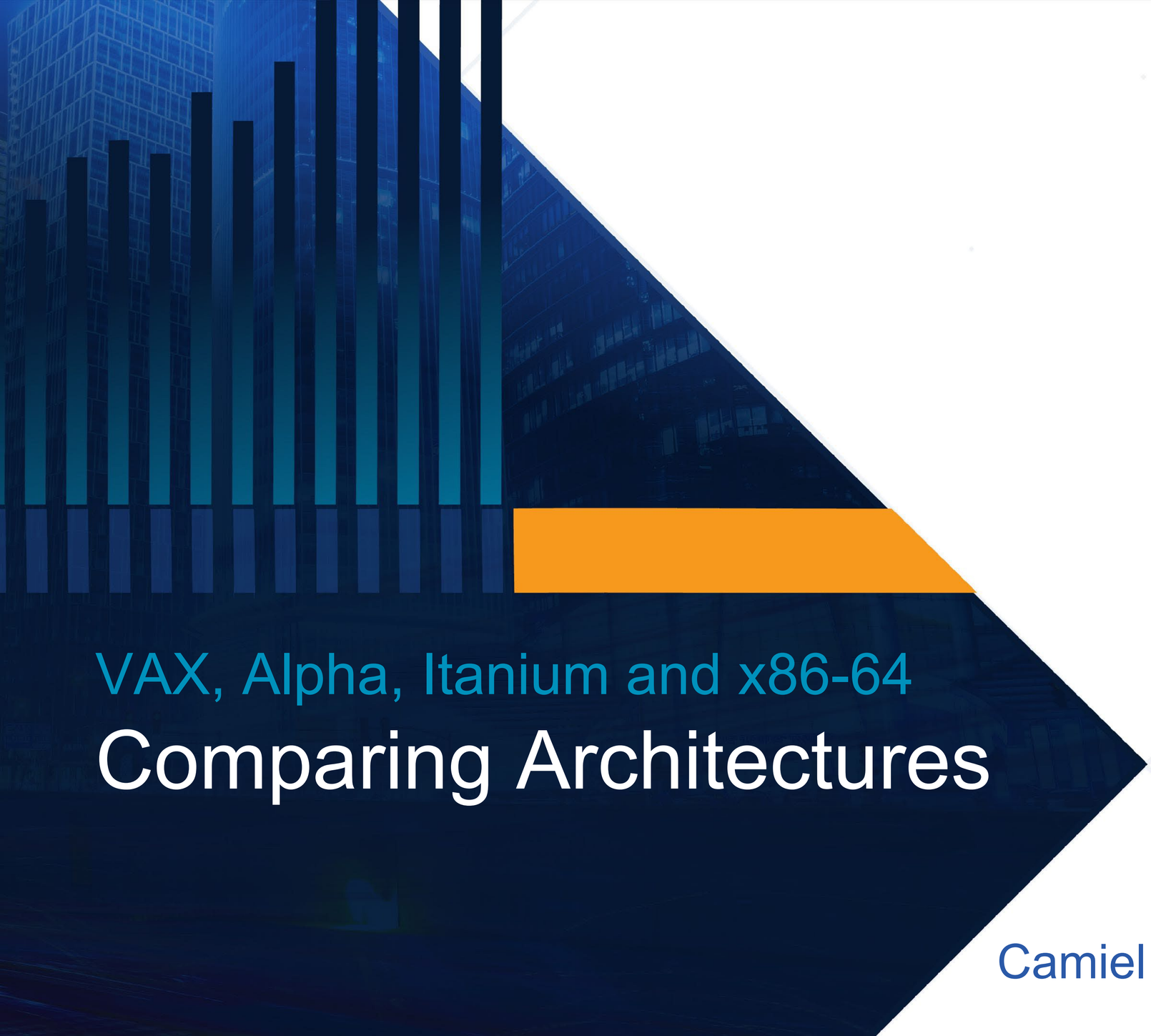
VMS Software

OpenVMS Across Four Architectures

A Trip Down Memory Lane

Berlin, April 2024

Camiel Vanderhoeven | Chief Architect & Strategist



VAX, Alpha, Itanium and x86-64

Comparing Architectures

vms Software

Camiel Vanderhoeven | SEP-2017

vms Software

VAX, Alpha, Itanium and x86-64

Comparing Architectures

Camiel Vanderhoeven | SEP-2017

vms Software

Re-architecting SWIS for X86-64

Teaching a Not-So-Old Dog New Tricks

Camiel Vanderhoeven | SEP-2017

vms Software

Introduction to the x86 Architecture

Camiel Vanderhoeven

September 15, 2015

vms Software

Oddities and surprises

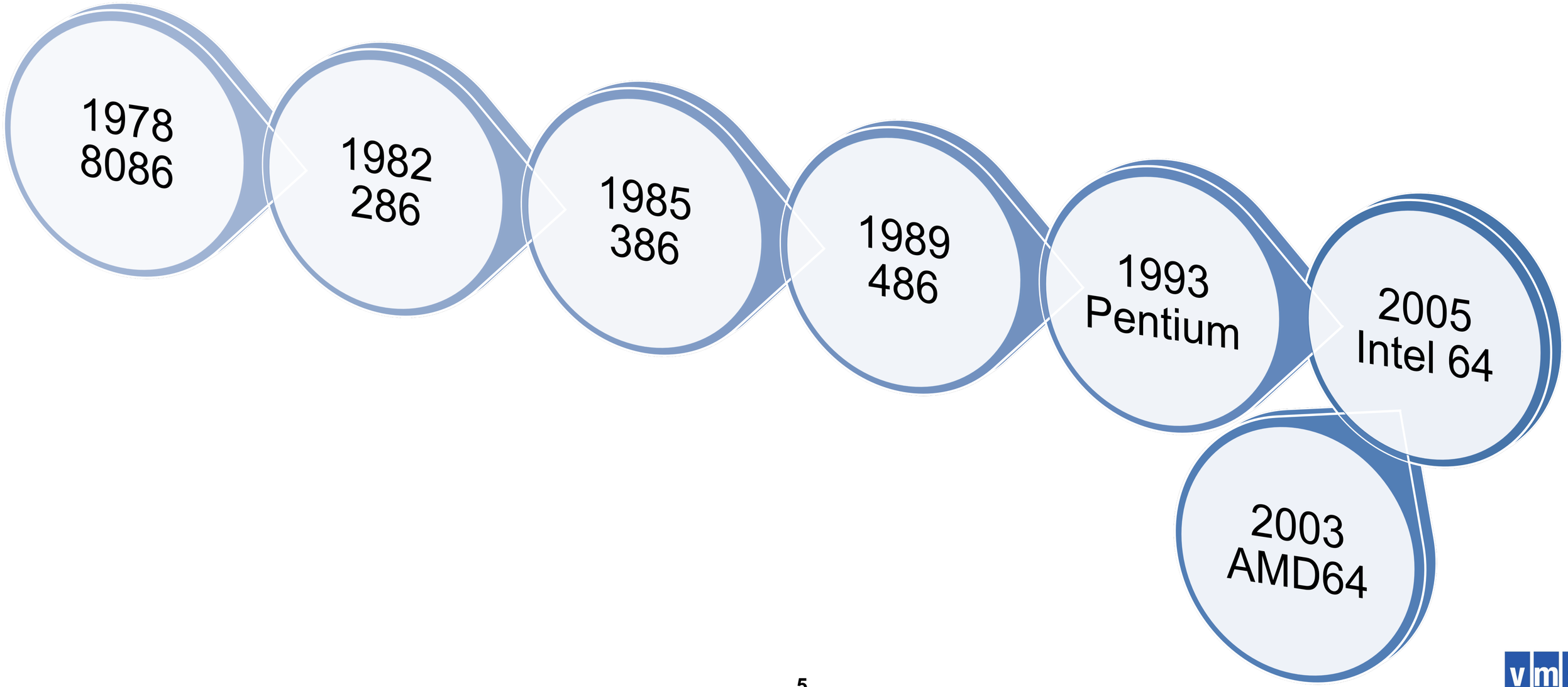
Interesting Features of the X86-64 Architecture

Camiel Vanderhoeven | SEP-2017

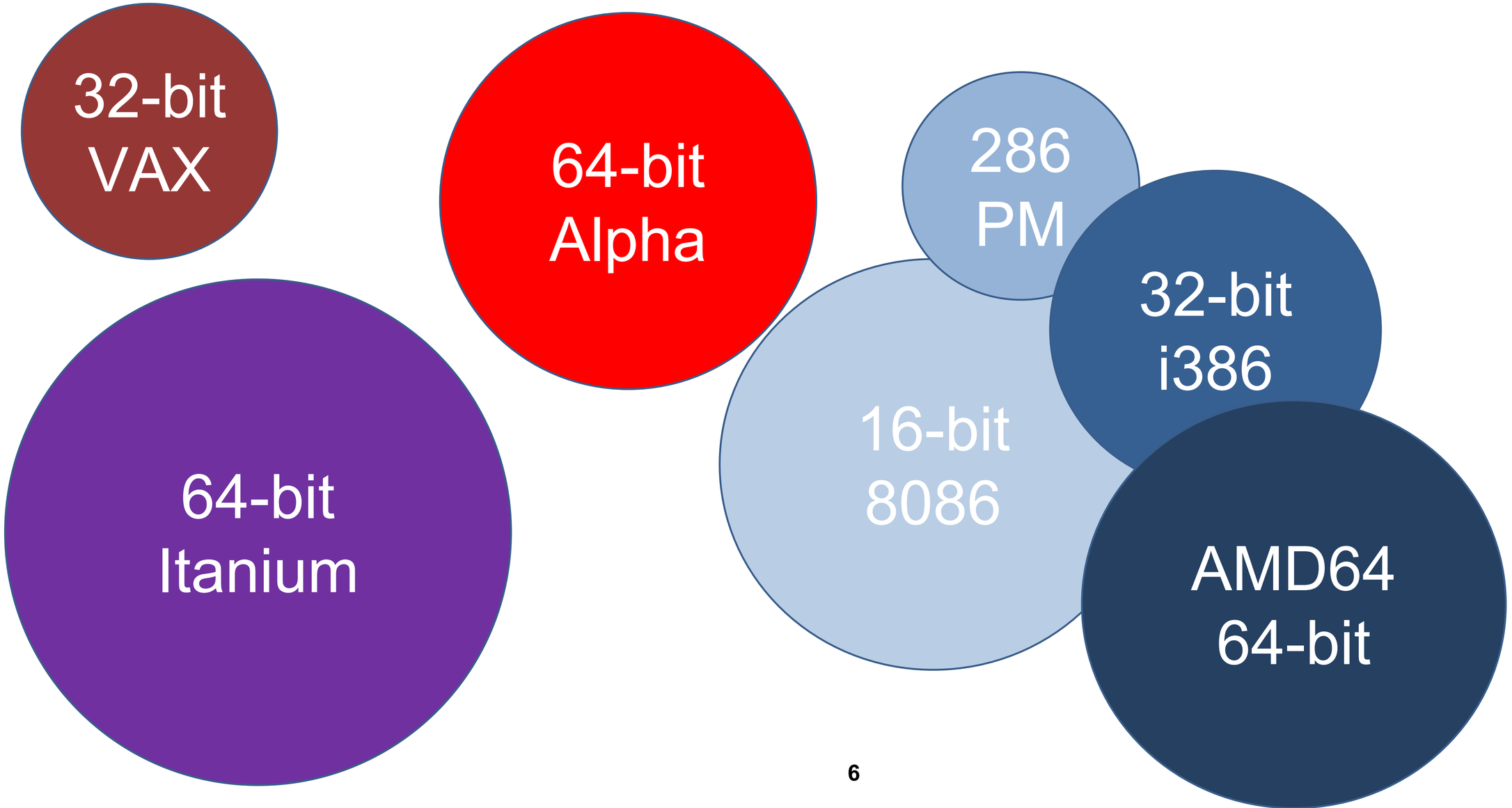
X86 Heritage

Where did this architecture come from?

X86 Development Timeline



New Design vs. Extensions



And boy, did this present us with challenges…

Booting

- Boot Processor put in 64-bit mode by Firmware
- Additional processors halted in 16-bit real mode

Legacy hardware

- Multiple ways to deal with timers, interrupts, etc.

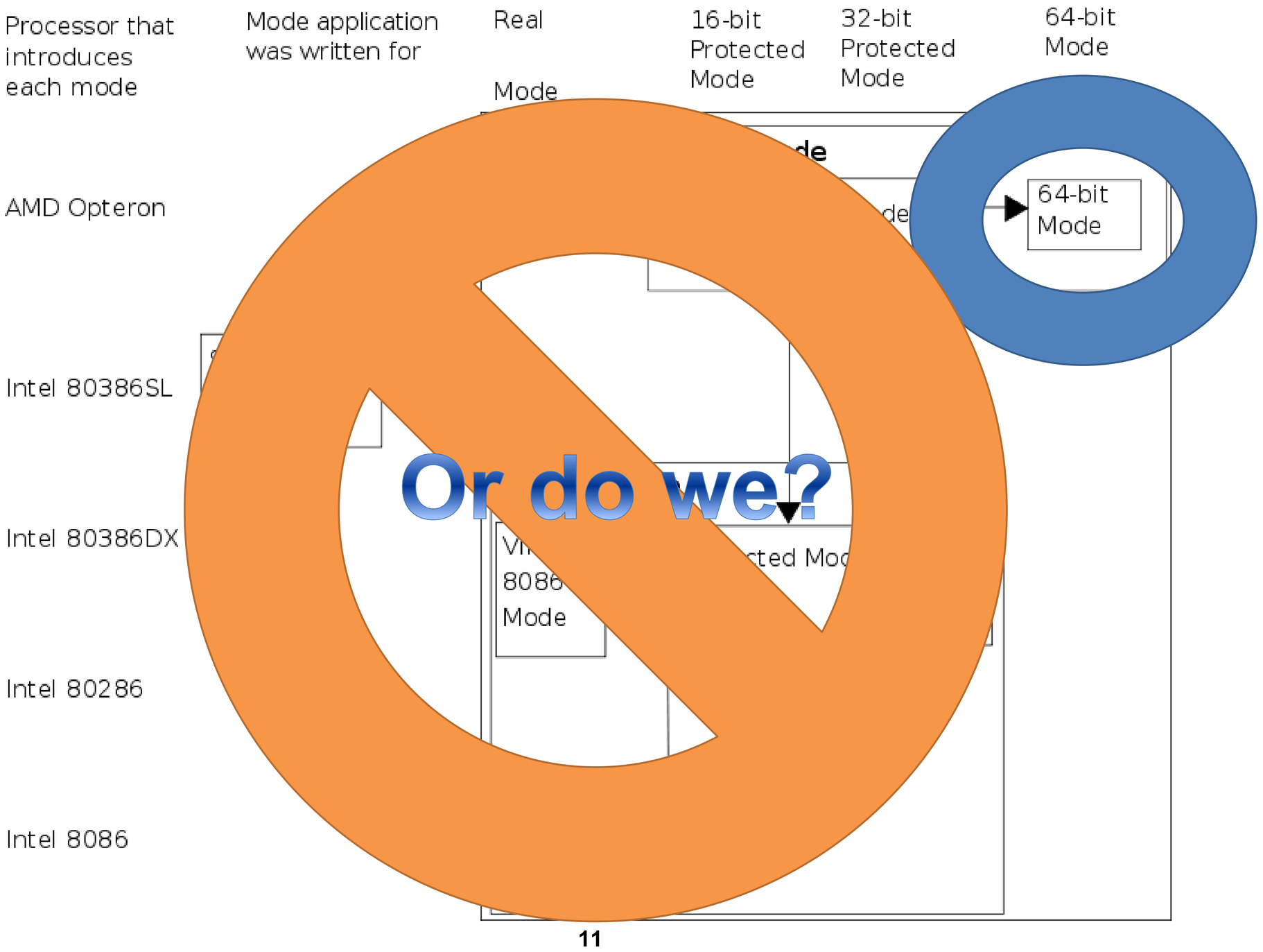
Processor Modes

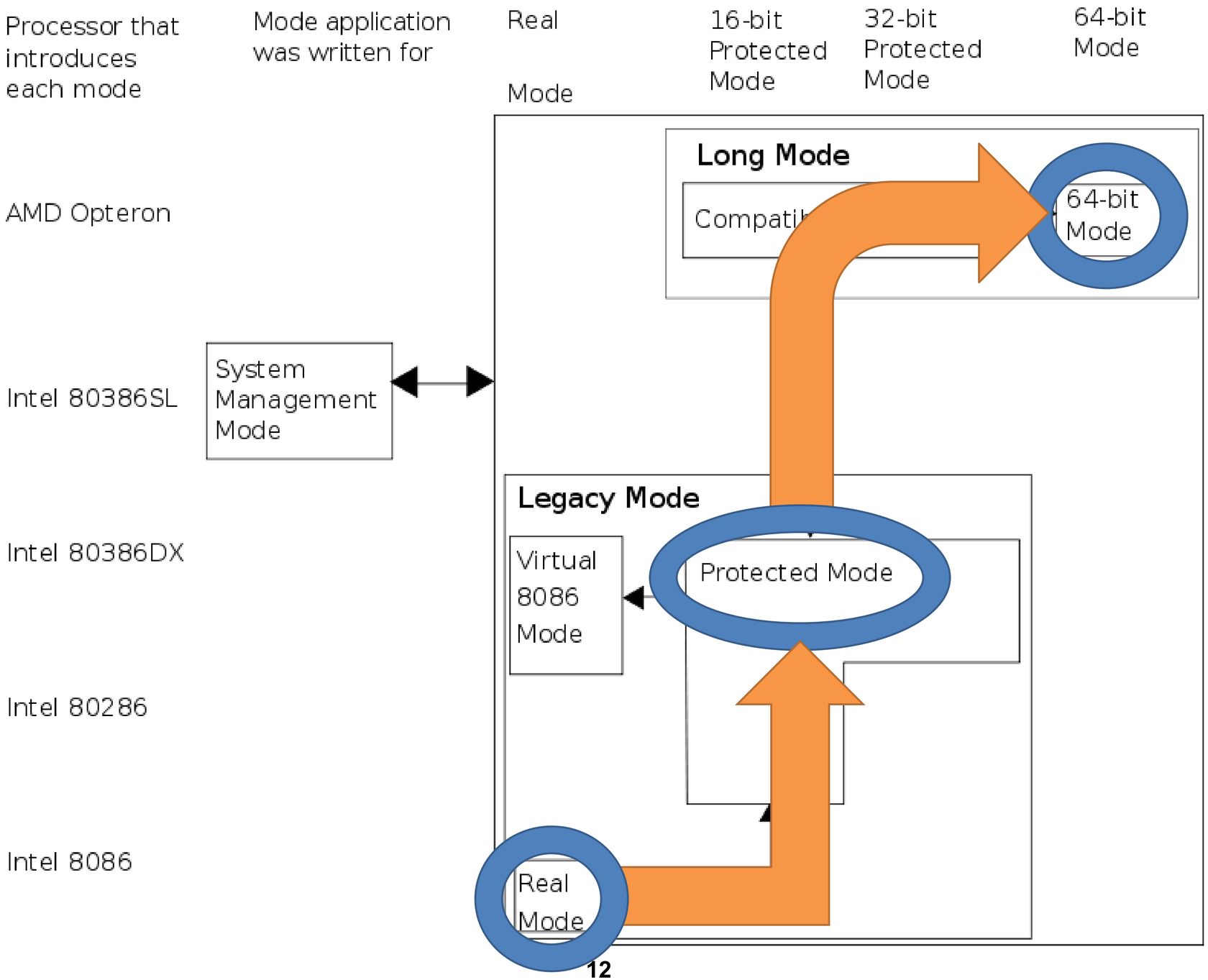
Today, I'd like to be…

All modes are still there

- For compatibility reasons, all processor modes going back to the 8086 are **still present** in modern processors

16 and 32 bit modes on X86


Processor that introduces each mode
Mode application was written for
Real Mode
16-bit Protected Mode
32-bit Protected Mode
64-bit Mode
AMD Opteron
Intel 80386SL
Intel 80386DX
Intel 80286
Intel 8086
64-bit Mode
Or do we?


Processor that introduces each mode
Mode application was written for
Real Mode
16-bit Protected Mode
32-bit Protected Mode
64-bit Mode
AMD Opteron
Intel 80386SL
Intel 80386DX
Intel 80286
Intel 8086
System Management Mode
Long Mode
Compati
64-bit Mode
Legacy Mode
Virtual 8086 Mode
Protected Mode
Real Mode

Architecture Documentation

What is this chip, and what does it do?

We need documentation

- When you're writing – or porting – an operating system, you need to **know a lot** about the underlying platform
- So, we need documentation
- VAX: VAX-11 Architecture Reference Manual
- Alpha: Alpha AXP Architecture Reference Manual
- Itanium: Intel Itanium Architecture Software Developer's Manual
- X86-64: Intel 64 and IA-32 Architecture Software Developer's Manual
- Here's what this looks like…

Filling a bookshelf...


VAX
VAX-11 Architecture Reference Manual
20 May 1982
Revision 6.1
See MIPS Run
Sweetman
RSTS/E V9 Internals and Data Structures
INSIDE THE PERSONAL COMPUTER
A POP-UP GUIDE
ABBEVILLE
DAVID BAILEY BIRTH OF THE COOL
MARTIN HARRISON
VIKING STUDIO
1”


... VAX, Alpha
SECOND EDITION
Alpha AXP Architecture
REFERENCE MANUAL
SITES • WITEK
dp
Digital Press
Alpha AXP
RICHARD L. SITES
RICHARD T. WITEK
SECOND EDITION
Architecture
REFERENCE MANUAL
VAX-11 Architecture Reference Manual
See MIPS Run
Sweetman
RSTS/E V9 Internals and Data Structures
INSIDE THE PERSONAL COMPUTER
A POP-UP GUIDE
ABBEVILLE
MILES DAVIS
BIRTH OF THE COOL
BEST OF BLUE NOTE
MARTIN HARRISON
DAVID BAILEY BIRTH OF THE COOL
VIKING STUDIO
2”


... VAX, Alpha, Itanium
SECOND EDITION
Alpha AXP Architecture
REFERENCE MANUAL
SITES • WITEK
Digital Press
Intel Itanium Architecture Software Developer's Manual Revision 2.2
Volume 1: Application Architecture
Volume 2: System Architecture
Volume 3: Instruction Set Reference
See MIPS Run
Sweetman
VAX-11 Architecture Reference Manual
RSTS/E V9 Internals and Data Structures
INSIDE THE PERSONAL COMPUTER
A POP-UP GUIDE
ABBEVILLE
MILES DAVIS BIRTH OF THE COOL BEST OF BLUE NOTE
MARTIN HARRISON
DAVID BAILEY BIRTH OF THE COOL
VIKING STUDIO
5”


VAX, Alpha, Itanium, X86
12"
VMS for Alpha Platforms
Internals and Data Structures
Preliminary Edition
Volume 1
RSTS/E V9 Internals and Data Structures
Intel® Architecture Instruction Set Extensions Programming Reference
Intel Corporation
Intel® 64 and IA-32 Architectures Optimization Reference Manual
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference A-M
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture
Intel Corporation
Intel® Itanium® Architecture Software Developer's Manual Revision 2.2
Volume 3: Instruction Set Reference
Intel® Itanium® Architecture Software Developer's Manual Revision 2.2
Volume 2: System Architecture
Intel® Itanium® Architecture Software Developer's Manual Revision 2.2
Volume 1: Application Architecture
SECOND EDITION
Alpha AXP Architecture
REFERENCE MANUAL
SITES • WITEK
Digital Press
VAX-11 Architecture Reference Manual
See MIPS Run
Sweetman
INSIDE THE PERSONAL COMPUTER
A POP-UP GUIDE
ABBEVILLE
BIRTH OF THE COOL
BEST OF BLUE NOTE
DAVID BAILEY BIRTH OF THE COOL
VIKING STUDIO
MARTIN HARRISON

ISA Extensions

Name	First in	Function
x87	8086+8087 (1980)	Floating Point Co-processor
PM	80286 (1982)	Protected Mode: Virtual Memory
IA-32	80386 (1985)	32-bit
PAE	Pentium Pro (1995)	Physical Address Extension
MMX	Pentium MMX (1997)	MultiMedia Extension (Integer SIMD)
3Dnow!	AMD K6-2 (1998)	3D Graphics (Floating Point SIMD)
SSE(n)	Pentium III (1999)	Streaming SIMD Extensions (FP SIMD)
x86-64	Opteron (2003)	64-bit
VT-x	Pentium 4 (2005)	Virtualization support
AMD-V	Athlon 64 (2006)	Virtualization support
AES-NI	Westmere (2010)	Advanced Encryption Standard
AVX(n)	Sandy Bridge (2011)	Advanced Vector Extensions (FP SIMD)
TSX	Haswell (2013)	Transactional Synchronization Extension
MPX	Skylake (2015)	Memory Protection Extensions

Some panned out

VT-x and AMD-V

- Help us run on hypervisors with near-bare-metal performance

PCIDs (not on list)

- Help us overcome part of the performance loss from mode changes

Intel TSX

Transactional Synchronization Extensions

The problem with locks

- One of the most **difficult** things to get right in a multi-threaded application (or OS) is **synchronization** between threads and processors
- Simultaneous memory access leads to **conflicts** (reading partially stale data, partial writes)
- Most common mechanism to overcome these issues is **locking**
- Locking is **expensive**
- Even when the lock is free, it is still taken
- Trade-off: **simplicity** (coarse locking) vs **performance** (fine-grained locking)

Intel TSX

- TSX aims to **eliminate locking** by exploiting the CPU's local data cache.
- During a TSX transaction, writes are **only visible** to the local CPU, they are not written to memory, and are **not seen** by the other CPU's.
- After a TSX transaction, the memory writes that occurred during the transaction become visible to the other CPU's **atomically**, and are then written to memory.
- During a TSX transaction, the CPU **monitors** memory writes by other CPU's. If a memory write by another CPU conflicts with a memory write or read from this CPU, the transaction is **aborted** (the memory writes are thrown away and never seen by anyone).

Spinlock example

No TSX:

```
    xorl %ecx, %ecx
    incl %ecx
spin_lock_retry:
    xorl %eax, %eax
    lock; cmpxchgl %ecx, (lock)
    jnz spin_lock_retry

    incl (value)

    movl $0 (lock)

    ret
```

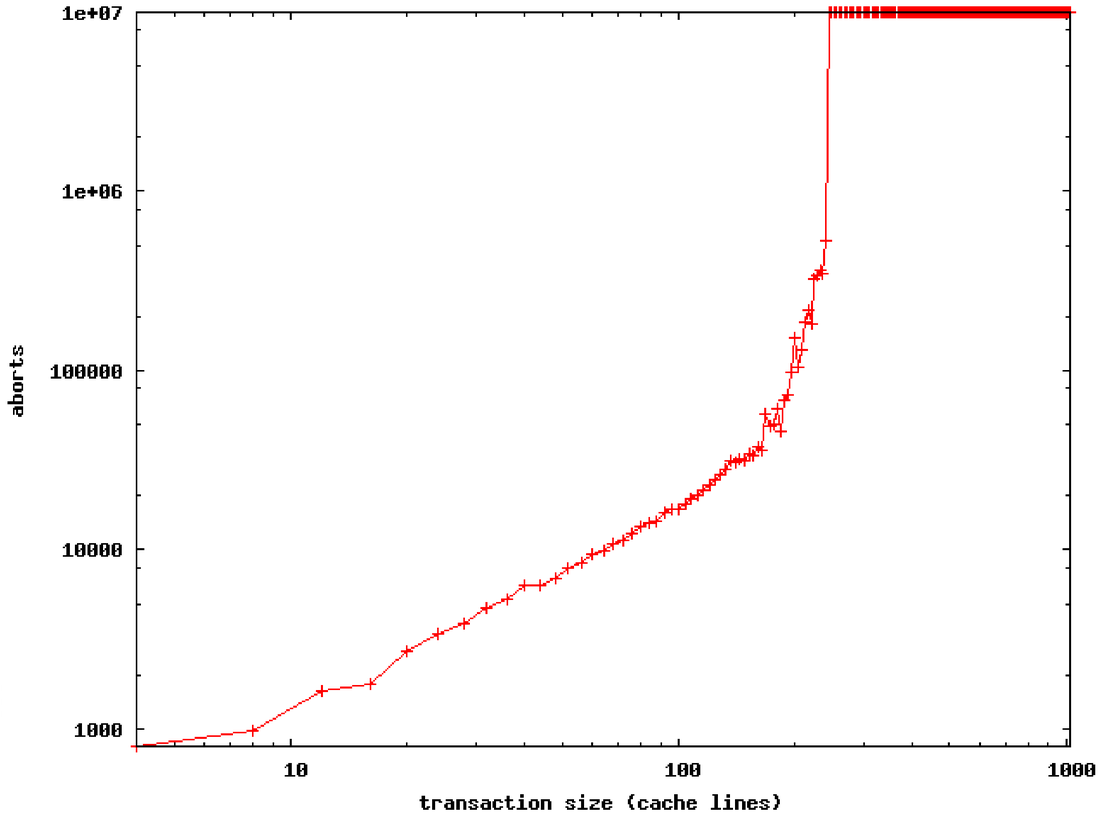
TSX:

```
    xbegin retry_with_spinlock
    incl (value)
    xend

    ret

retry_with_spinlock:
    xorl %ecx, %ecx
    incl %ecx
spin_lock_retry:
    xorl %eax, %eax
    lock; cmpxchgl %ecx, (lock)
    jnz spin_lock_retry

    incl (value)

    movl $0 (lock)

    ret
```

Some Observations (1/2)

- TSX **debuted** in Haswell, subsequently **removed** by a microcode update because of a bug, and **re-introduced** in late Broadwell steppings and Skylake.
- With TSX (either HLE or RTM), locking becomes very fine-grained when there are no conflicts. Fallback traditional locks can therefore be **less fine-grained without too much of an impact on performance**.
- Transactions can be **nested** (but abortions are passed up).
- Granularity of conflict detection is **implementation specific**, and probably equal to the size of a cache line (64 bytes in Skylake).
- There is an implementation specific **limit** to the number of cache lines that can be involved in a transaction (512 cache lines in Skylake). If this limit is exceeded, the transaction will **abort**. Limited associativity of cache lines may cause an abort even **before** this limit is reached.

Some Observations (2/2)



- A **context switch** will also lead to aborted transactions.
- The **risk** of an abortion grows with the **transaction size**, even without any conflicts
- Therefore, **fallback code** must always be provided. This code needs to use a **traditional lock**, in case more than one thread needs to fall back.
- When an abortion happens in RTM, the fallback code is informed of the reason for the abortion, this can be a **conflict**, an **explicit** XABORT instruction, a lack of **resources**, the execution of an **incompatible instruction**, or the occurrence of an **uncommon event** (whatever that may be).

Possible Uses in OpenVMS

- I/O locks
- Scheduling lock
- Interlocked queue operations

TSX

Prototyped queue instructions

- Better performance
- Ran in callers' mode

And then TSX was cancelled and removed from chips (again!)

Register Set

VAX Register Set

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
AP/R12
FP/R13
SP/R14
PC/R15

PSL
IPR’s

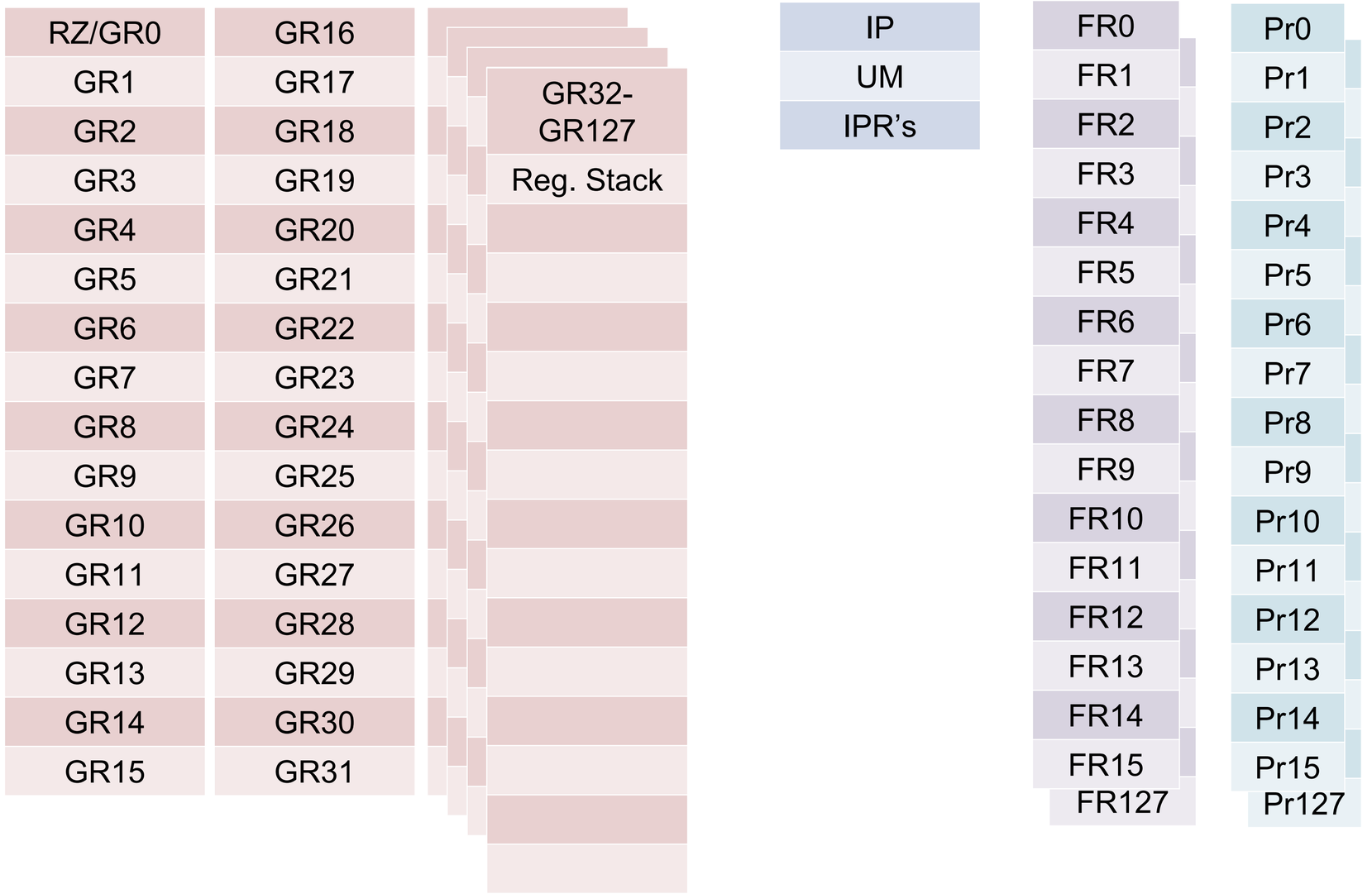
Alpha Register Set

R0	R16
R1	R17
R2	R18
R3	R19
R4	R20
R5	R21
R6	R22
R7	R23
R8	R24
R9	AI/R25
R10	RA/R26
R11	PV/R27
R12	R28
R13	FP/R29
R14	SP/R30
R15	RZ/R31

PC
PS
IPR’s

F0	F16
F1	F17
F2	F18
F3	F19
F4	F20
F5	F21
F6	F22
F7	F23
F8	F24
F9	F25
F10	F26
F11	F27
F12	F28
F13	F29
F14	F30
F15	F31

Itanium Register Set



x86 Register Set

RAX
RCX
RDX
RBX
RSP
RBP
RSI
RDI
R8
R9
R10
R11
R12
R13
R14
R15

MMX0/FPR0
MMX1/FPR1
MMX2/FPR2
MMX3/FPR3
MMX4/FPR4
MMX5/FPR5
MMX6/FPR6
MMX7/FPR7

XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM15
XMM16

RIP
RFLAGS
IPR’s

Dealing with fewer registers

C/Bliss

- No issues, compiler can take care of it

Assembly

- Some juggling required

VAX Macro (w/Alpha extensions)

- Alpha Pseudoregisters (per mode)

Instruction Encoding

VAX

Opcode

1 or 2 bytes, 1 opcode per operation

[Operand 1]

1 byte containing addressing mode and register number, up to 4 bytes of displacement, immediate data, or address

[Operand 2 … n]

same

Alpha

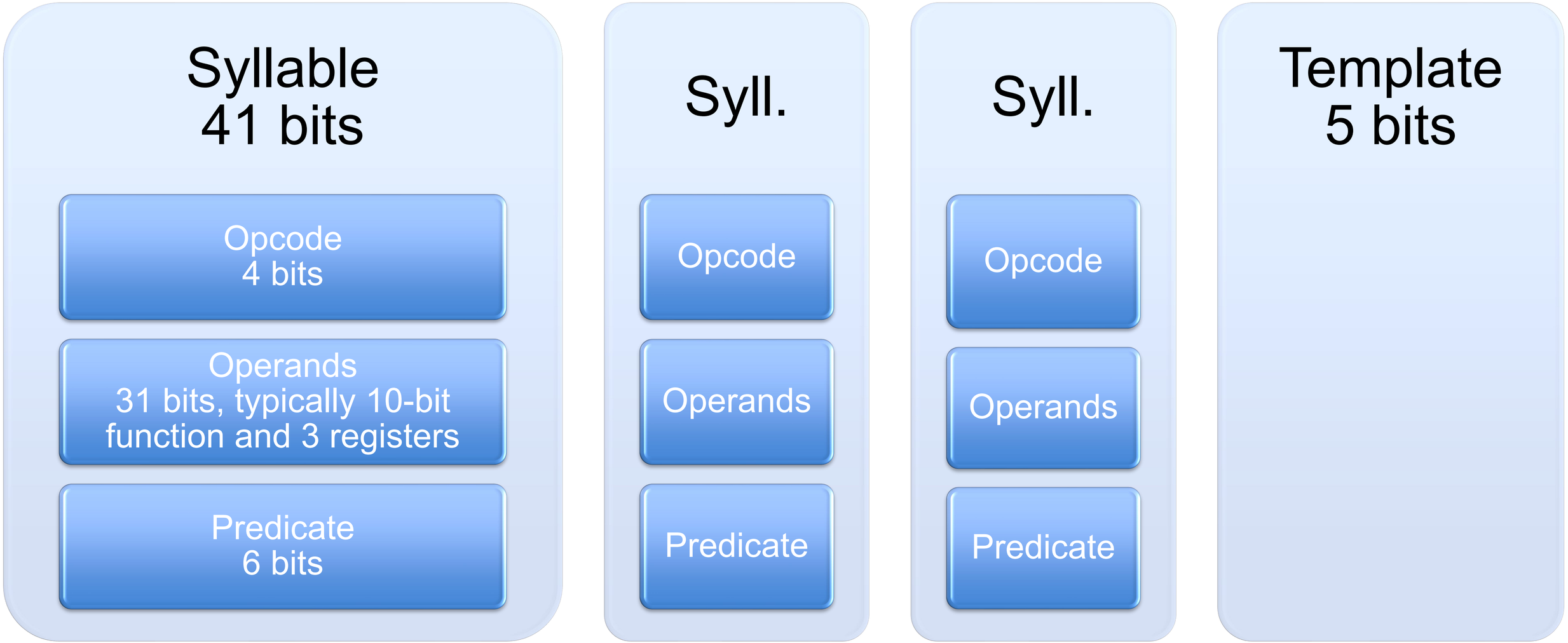
Opcode

6 bits, one opcode per operation

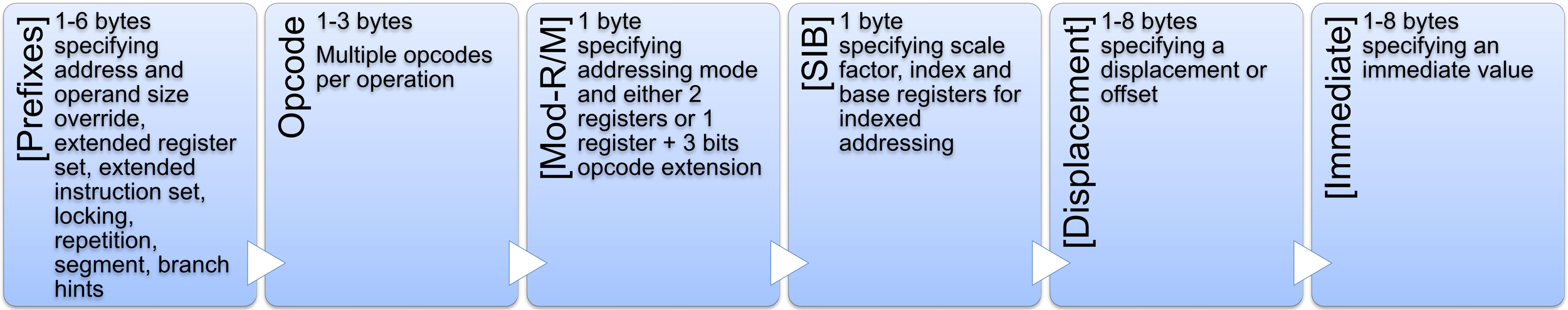
Operands

26 bits, encoding up to 3 registers, up to 21-bit displacement, 8-bit literal value, up to 16-bit function specifier

Itanium


Syllable
41 bits
Opcode
4 bits
Operands
31 bits, typically 10-bit
function and 3 registers
Predicate
6 bits
Syll.
Opcode
Operands
Predicate
Syll.
Opcode
Operands
Predicate
Template
5 bits

X86-64


[Prefixes]
1-6 bytes specifying address and operand size override, extended register set, extended instruction set, locking, repetition, segment, branch hints
Opcode
1-3 bytes
Multiple opcodes per operation
[Mod-R/M]
1 byte specifying addressing mode and either 2 registers or 1 register + 3 bits opcode extension
[SIB]
1 byte specifying scale factor, index and base registers for indexed addressing
[Displacement]
1-8 bytes specifying a displacement or offset
[Immediate]
1-8 bytes specifying an immediate value

Phew…

Compilers

- LLVM

Debugger etc.

- Open-source instruction decoders

Exception handling

- Manual labor

Memory Layout

Memory Layout

	VAX	Alpha	Itanium	x86
Address size	32	64	64	64
H/W page size	512	8K/64K/512K/4M	4K-4G	4K/2M/1G
Split VA Space	-	yes	yes*	yes
Regions	-	-	8	-
PT Levels	2	3	3*	4
PTE Cache	-	-	VHPT	PDE cache
Virt. Addr. Size	32	48	48*	48
Phys. Addr. Size	32	44	50	52 (48)
Physical addressing	yes	yes	yes	-
Segmentation	-	-	-	yes (kind of)
Prot. Bits in TLB	4 enc[KESU][RW]	11 [KESU][RW], FO[RWE]	7 enc[KESU], enc[RWX]	3 R/W, U/S, XD

* OS Implementation Dependent, figures given are for OpenVMS

The Case for SWIS

What is SWIS, and why is it needed?

OpenVMS Assumes Things…

- VAX/VMS was designed in tandem with the VAX hardware architecture.
- Where desirable, hardware features were added to satisfy the OS' needs.
- A lot of OS code was written to make use of these hardware features.

What are these Assumptions?

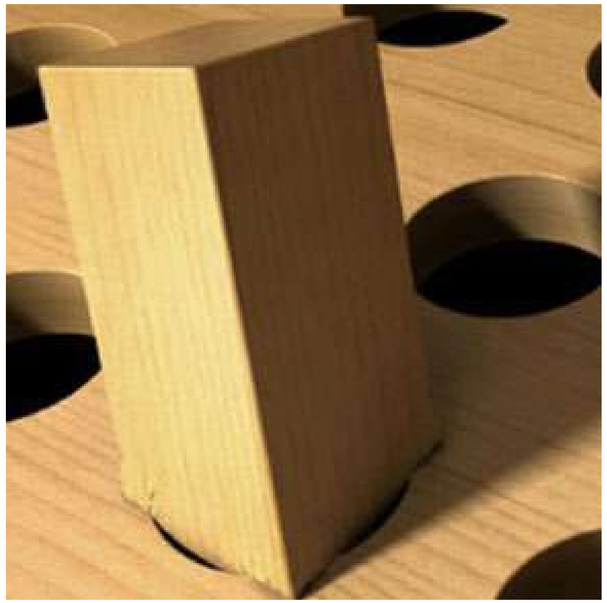
- 4 hardware privilege modes
- Each with different page protections
- And with their own stack
- 32 Interrupt Priority Levels
- 16 for Hardware Interrupts
- 16 for Software Interrupts
- Software Interrupts are triggered immediately when IPL falls below the associated IPL
- Asynchronous Software Trap (AST) associated with each mode, triggered immediately when IPL falls below ASTDEL (equally or less privileged mode)
- The hardware provides atomic instructions for queue operations
- The hardware provides a set of architecturally defined Internal Processor Registers (IPRs)

How does Alpha meet these Assumptions?

- Alpha is a very clean RISC Architecture
- But OpenVMS was definitely in the Alpha Architecture designers' minds
- The 4 modes OpenVMS needs are part of the basic Alpha architecture
- PALcode, code supplied by firmware that has more privileges than even kernel mode, and which is uninterruptible, provides the flexibility to implement OS specific features
- IPLs, Software Interrupts and ASTs are implemented through a combination of hardware support and PALcode
- Atomic queue instructions are provided by PALcode
- PALcode also provides the mapping from IPRs as expected by OpenVMS to the hardware implementation's IPRs

So how about Itanium Hardware?

- Very different story, Itanium's design was finished before OpenVMS as an OS was considered
- Offers the 4 modes OpenVMS needs
- The TPR (Task Priority Register) provides an IPL-like mechanism for hardware interrupts only
- No compatible software interrupt mechanism or ASTs
- No atomic queue instructions
- No OpenVMS-compatible IPRs

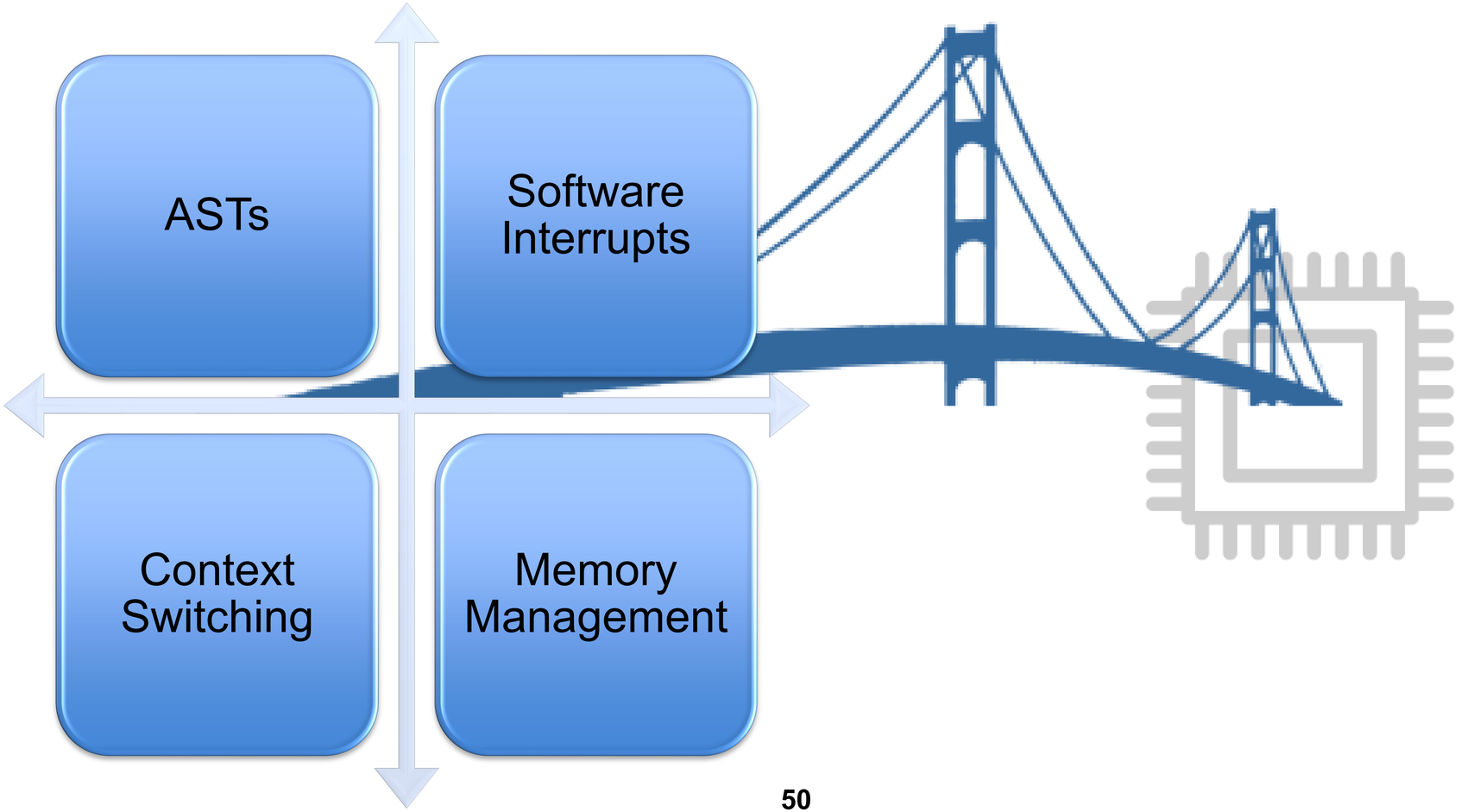


Hence, SWIS

- SWIS (Software Interrupt Services) is a piece of low-level OS code that is involved in mode changes.
- SWIS implements the software interrupt and AST support required by OpenVMS, using hardware support as available.
- Other code in the OS (with some special support from the SWIS code to ensure atomicity) provides atomic queue instructions
- A combination of code in SWIS and other code in the OS provides OpenVMS-compatible IPRs
- SWIS makes the Itanium CPU look more like a VAX to the rest of the OS

Bridge Function

SWIS bridges the gap between the assumptions made by the rest of the OS to the features supported by the hardware



SWIS on X86-64

- Because a similar mismatch exists between OpenVMS' assumptions and the hardware-provided features, SWIS will be ported to X86-64.
- Ported means mostly re-written here, as the provided features are very different between Itanium and X86-64.
- On X86-64, SWIS will have to do more, as the X86-64 architecture does not provide the 4 mode support OpenVMS needs.
- Because of this, SWIS on X86 will not only be active when transitioning from an inner mode to an outer mode, but also when transitioning from an outer mode to an inner mode.
- Also because of this, SWIS now needs to become involved in memory management (in a supporting role).
- There's good news too: the Itanium architecture has some features that are very complex to manage (think RSE), that are absent in X86-64.

Swis on X86-64

OpenVMS Expects:

- 4 Modes, different page protections, separate stacks
- 32 IPLs (16 h/w, 16 s/w)
- Software interrupts tied to IPLs
- Per-process, per-mode ASTs, delivered when below ASTDEL
- Atomic queue instructions
- VAX-like IPRs

X86-64 Offers:

- 2 rings, different page protections, separate stacks
- 14 hardware TPR's, mask off hardware interrupts in groups of 16
- Software interrupts unaffected by TPR's. No IPL's
- No AST-like concept at all
- No atomic queue instructions
- X86-64 IPRs

Itanium vs X86-64

Differences relevant to SWIS between Itanium and X86-64

System Services

Itanium:

- EPC Instruction
- Can only be called within a protected promote page
- First instruction executed in kernel mode is the one following the EPC instruction
- Does not disable interrupts
- Need to consider register stacks

X86-64:

- SYSCALL Instruction
- Can be called anywhere
- First instruction executed in kernel mode is at a fixed, OS-determined (MSR) address
- Can be set up to atomically disable interrupts
- No register stack

Interrupts and Exceptions

Itanium:

- IVT is code
- Does not switch stacks
- Interrupted state is stored in registers
- Need to consider register stack

X86-64:

- IDT contains descriptors that points at code
- Switches to kernel-mode stack if needed
- Interrupted state is stored on kernel-mode stack
- No register stack

Memory Management

Itanium:

- 4 modes
- OS needs to handle an exception on TLB miss
- Memory translations are at the OS' discretion (Translation Registers)
- Support for URKW and similar protections
- 8 2^{61} byte regions in VA

X86-64:

- 2 useable rings (0 and 3)
- Processor walks page tables on TLB miss
- All memory translations are page-table based
- No support for ring 3 read, ring 0 write
- No regions

Finding Per-CPU SWIS Data Structure

Itanium:

- Mapped through a dedicated TR
- At fixed VA (0xE…0)
- Mapping of VA to PA is different on each processor, and TR is not touched on a context switch

X86-64:

- Mapped through the normal page tables
- At a different VA for each CPU
- Mapping of VA to PA is identical for each process, so it doesn't change on a context switch
- Can be found at address 0 in the %GS segment on each CPU
- %GS segment is loaded from segment descriptor 0x28 on each CPU
- Segment 0x28 base VA is different on each CPU

Finding the Per-CPU SWIS Data Structure (2)

Itanium:

- Accessing a field at offset 0x80:

```
movl r28 = 0xe0000…0080
ld8 r28 = [r28]
```

- Getting the address of the structure:

```
movl r28 = 0xe0000…0000
```

X86-64:

- Accessing a field at offset 0x80:

```
movq %gs:0x80, %rax
```

- Getting the address of the structure:

```
rdgsbase %rax
```

– or –

```
movq %gs:0, %rax
```

Porting SWIS

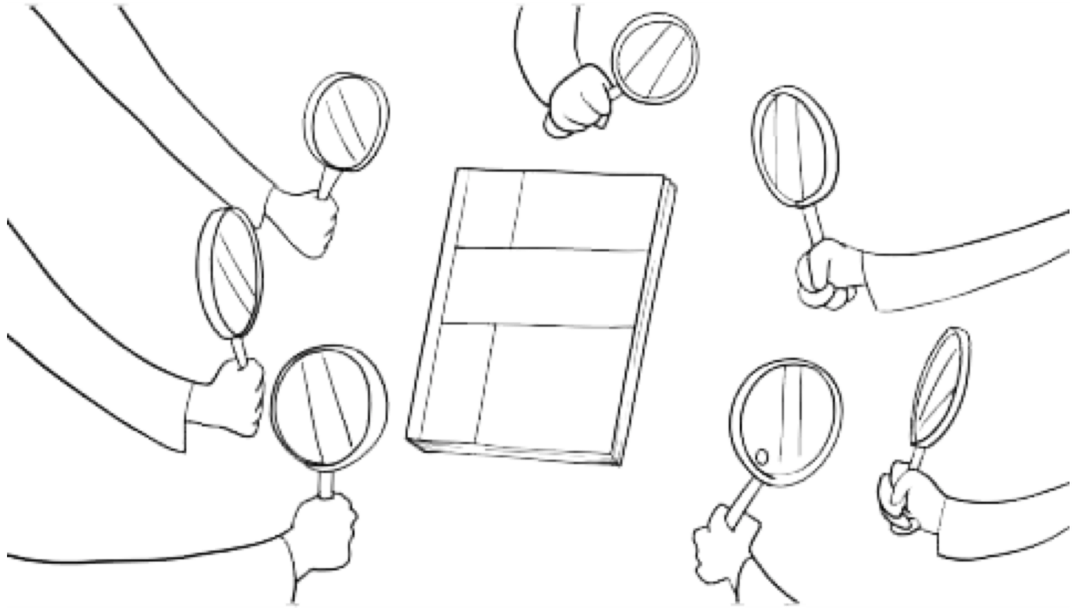
The porting process, a birds-eye overview

Design Phase

SWIS for X86-64 was designed over a period of 1.5 years (1 year part-time, 0.5 years full-time), in several phases:

- Basic design (not detailed enough to base implementation on)
- Detailed design for System Service dispatching
- Detailed design for Hardware Interrupt and Exception handling
- Detailed design for Software Interrupts and ASTs
- Detailed design for Processes and Kernel Threads

Design Review Phase



- Partial reviews as the design progressed
- In-depth 3-day review between myself and Burns Fisher
- This one turned up a design flaw that could have enabled unprivileged code to bring down the system
- Complete walk-through and review in one of our weekly X86-64 engineering meetings
- A lot of the content in this presentation is based on the slides I prepared for that walk-through

Implementation Phase

Implementation started in May 2017, broken down into different parts:

- Quick and Dirty Exception Handling for early code that needs something
- Data Structure Definitions
- VAX/Alpha IPRs
- Hardware Interrupts and Exceptions
- System Services
- Software Interrupts
- ASTs
- Initialization
- Processes and Scheduling

2 SYSTEM_PRIMITIVES execlet builds

- Compatibility build, works on any x86-64 CPU we support
- Performance builds, optimized for CPUs that have support for one or more of the following:

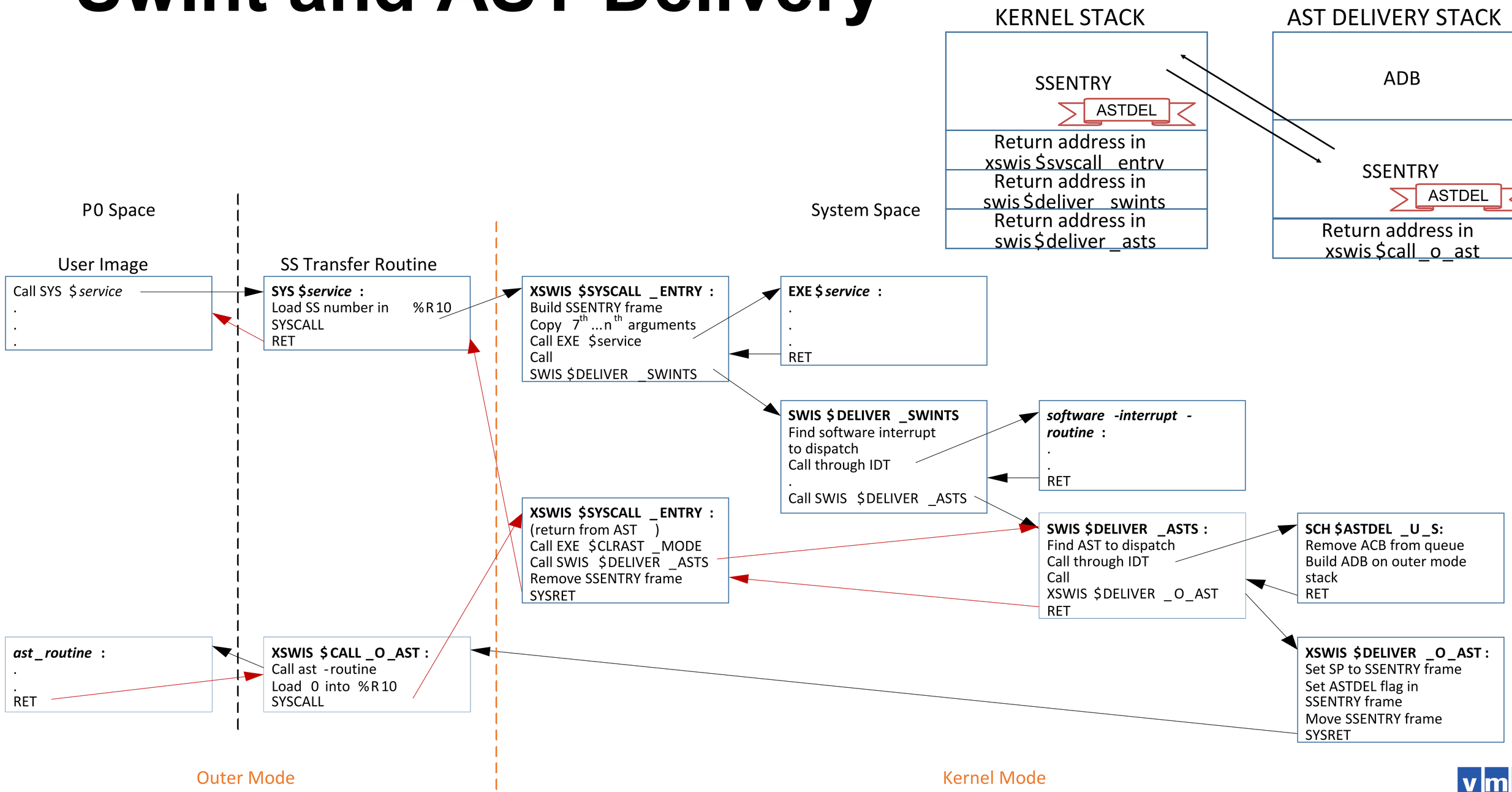
1. Address Space Numbers (PCIDs) in TLB
2. RDGSBASE instruction
3. XSAVES/XRSTORS instructions for saving/restoring extended (“floating point”) registers (MMX, SSE, AVX)

- Highest Performance build targets Intel processors made after 2013 (Ivy Bridge and beyond).

The Guts of SWIS

A technical overview, with lots of details

SwInt and AST Delivery



Mode “Components”

- Processor ring (0 for K, 3 for ESU)
- Stack pointer
- Address Space Number
- Page Table Base
- Current mode as recorded in the SWIS data structure

- A mode is “canonical” when all the above are in agreement
- SWIS should be the only code that ever sees non-canonical modes

- We prototyped this on Itanium

Basics of Mode Switching

- Interrupt or SYSCALL instruction

1. Switches CS and SS to ring 0
2. Switches to the kernel-mode stack (interrupt only, not SYSCALL)
3. Disables interrupts

- Get fully into kernel mode (ASN, PTBR, stack, DS, ES)
- Going in? -> Build return frame on stack
- Going out? -> Deliver SwInts and ASTs as needed
- Get into destination mode (ASN, PTBR, stack, DS, ES)
- IRET or SYSRET instruction

1. Switches CS and SS to ring 3
2. Switches to the outer-mode stack (IRET only, not SYSRET)
3. Enables interrupts

Context Switching

- Need to save/restore 4 PTBRs instead of 1
- MMX/SSE/AVX state will get saved using the XSAVE(S)/XRSTOR(S) instructions
- Need to deal with performance monitoring and debugging registers
- Will have a “never scheduled” flag for initial process/thread creation (Alpha uses canonical stack, Itanium uses the PFS register) ➔ On first scheduling, the stacks get zapped, and we go to the appropriate initialization routine for the process/thread.

So how close were we?

Pretty close

We missed a few things though

- Impact of not having a PROBE instruction
- Need to deal with emulated Alpha registers
- Use of interrupt stacks
- Need to track where the kernel stack is to detect a KSNV

And we made some improvements

- Lightweight system services
- All SWIS variants in a single image

Thank You

To learn more please contact us:

vmssoftware.com
info@vmssoftware.com
+1.978.451.0110