Let’s go

The bridge to possible

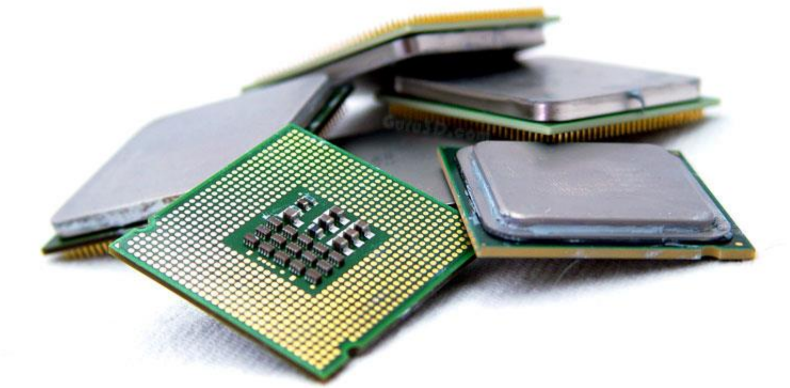
Working with ARM, x86 and more

The mixed compute-architecture world of today

Jens Depuydt, Technical Leader CX EMEAR
@jensdepuydt

Agenda

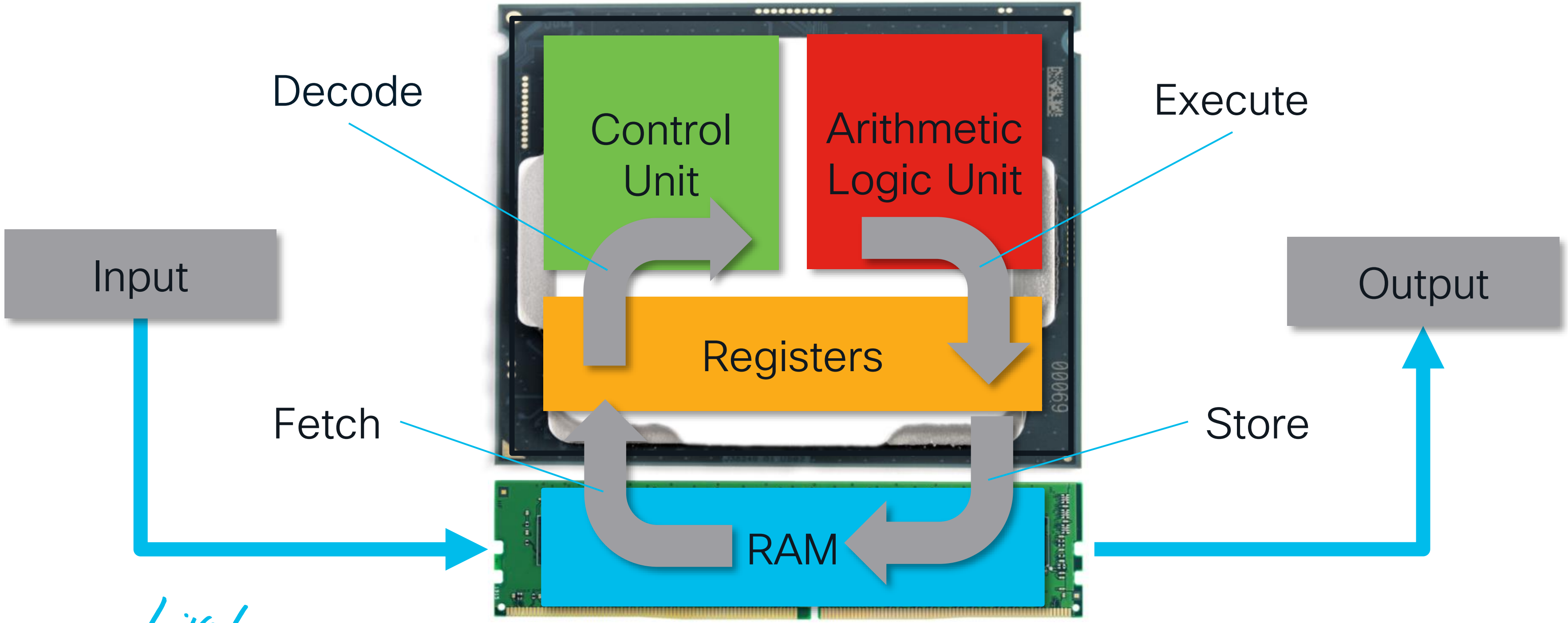
- Background: Compute Architectures
- Challenges: Mix of Architectures
- Solutions & demo
- Summary



Compute Architectures

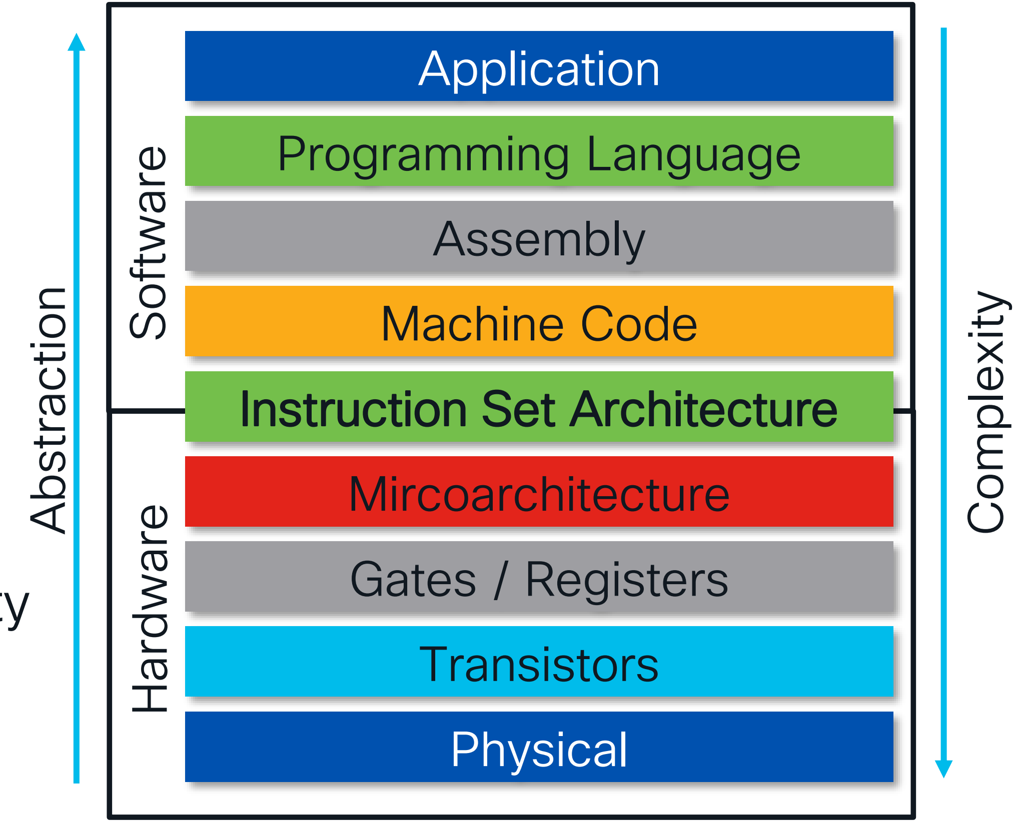
CPU – Central Processing Unit

In essence: **Fetch** – **Decode** – **Execute** – (**Store**)



ISA - Instruction Set Architecture

- Interface between hardware and software
- Instruction Set defines how:
  - Instructions are processed
  - Memory is accessed
  - IO is managed
- High-level programming language abstracts complexity
- Compiler translates code to lower-level instructions



Different ISAs/Compute Architectures - Historical

The table below compares basic information about instruction sets to be implemented in the CPU architectures:

Archi-tecture	Bits	V…
6502	8	
6809	8	
680x0	32	
8080	8	
8051	32 (8→32)	
x86	16, 32, 64 (16→32→64)	
Alpha	64	
ARC	16/32/64 (32→64)	AR…
ARM/A32	32	AR…
Thumb/T32	32	AR… AR…
Arm64/A64	64	AR…
AVR	8	
AVR32	32	Re…
Blackfin	32	

CDC Upper 3000 series	48	
CDC 6000 Central Processor (CP)	60	
CDC 6000 Peripheral Processor (PP)	12	
Crusoe (native VLIW)	32[13]	
Elbrus (native VLIW)	64	Elbr…
DLX	32	
eSi-RISC	16/32	
Itanium (IA-64)	64	
M32R	32	
Mico32	32	
MIPS	64 (32→64)	6[17…
MMIX	64	
Nios II	32	
NS320xx	32	
OpenRISC	32, 64	1.3[…

Archi-tecture	Bits	Version	Intro-duced	Max # operands	Type	Design	Registers (excluding FP/vector)	Instruction encoding	Branch evaluation	Endian-ness	Extensions	Open	Royalty free
Nios II	32		200x	3	Register–Register	RISC	32	Fixed (32-bit)	Condition register	Little	Soft processor that can be instantiated on an Altera FPGA device	No	On Altera/Intel FPGA only
NS320xx	32		1982	5	Memory–Memory	CISC	8	Variable Huffman coded, up to 23 bytes long	Condition code	Little	BitBlt instructions		
OpenRISC	32, 64	1.3[21]	2010	3	Register–Register	RISC	16 or 32	Fixed	?	?	?	Yes	Yes
PA-RISC (HP/PA)	64 (32→64)	2.0	1986	3	Register–Register	RISC	32	Fixed (32-bit)	Compare and branch	Big → Bi	MAX	No	
PDP-8[22]	12		1966		Register–Memory	CISC	1 accumulator 1 multiplier quotient register	Fixed (12-bit)	Condition register Test and branch		EAE (Extended Arithmetic Element)		
PDP-11	16		1970	2	Memory–Memory	CISC	8 (includes stack pointer, though any register can act as stack pointer)	Variable (16-, 32-, or 48-bit)	Condition code	Little	Floating Point, Commercial Instruction Set	No	No
POWER, PowerPC, Power ISA	32/64 (32→64)	3.1[23]	1990	3	Register–Register	RISC	32	Fixed (32-bit), Variable (32- or 64-bit with the 32-bit prefix[23])	Condition code	Big/Bi	AltiVec, APU, VSX, Cell	Yes	Yes
RISC-V	32, 64, 128	20191213[24]	2010	3	Register–Register	RISC	32 (including "zero")	Variable	Compare and branch	Little	?	Yes	Yes
RX	64/32/16		2000	3	Memory–Memory	CISC	4 integer + 4 address	Variable	Compare and branch	Little			No
S+core	16/32		2005			RISC				Little			
SPARC	64 (32→64)	OSA2017[25]	1985	3	Register–Register	RISC	32 (including "zero")	Fixed (32-bit)	Condition code	Big → Bi	VIS	Yes	Yes[26]
SuperH (SH)	32		1994	2	Register–Register Register–Memory	RISC	16	Fixed (16- or 32-bit), Variable	Condition code (single bit)	Bi		Yes	Yes
System/360 System/370 z/Architecture	64 (32→64)		1964	2 (most) 3 (FMA, distinct operand facility) 4 (some vector inst.)	Register–Memory Memory–Memory Register–Register	CISC	16 general 16 control (S/370 and later) 16 access (ESA/370 and later)	Variable (16-, 32-, or 48-bit)	Condition code, compare and branch	Big		No	No
Transputer	32 (4→64)		1987	1	Stack machine	MISC	3 (as stack)	Variable (8 ~ 120 bytes)	Compare and branch	Little			
VAX	32		1977	6	Memory–Memory	CISC	16	Variable	Compare and branch	Little		No	
Z80	8		1976	2	Register–Memory	CISC	17	Variable (8 to 32 bits)	Condition register	Little			

Different ISAs/Compute Architectures - Today

	x86	ARM	RISC-V	Power
Since	1978	1985	2010	1992
# Bit	16/32/64	16*/32/64	16*/32/64*/128*	32/64
Instruction set	CISC	RISC	RISC	RISC
Endianness	Little	Bi (little by default)	Little	Bi (big by default)
Power usage	High	Low	Low	Medium
Linux architecture	amd64/**x86_64**	arm64/**aarch64**	**riscv64**	ppc/ppc64
Licensing	Strict	Flexible	Open-Source	Semi-strict
Examples	Intel 8086 Intel Pentium AMD (Ryzen) VIA	Cortex (A/X) Ampere Altra Snapdragon Apple Silicon	Google Titan SiFive P-series Esperanto ET-SoC Alibaba Xuantie	IBM Power Freescale T-series Xenon (Xbox 360) Espresso (Wii U)

Current situation: ARM vs x86

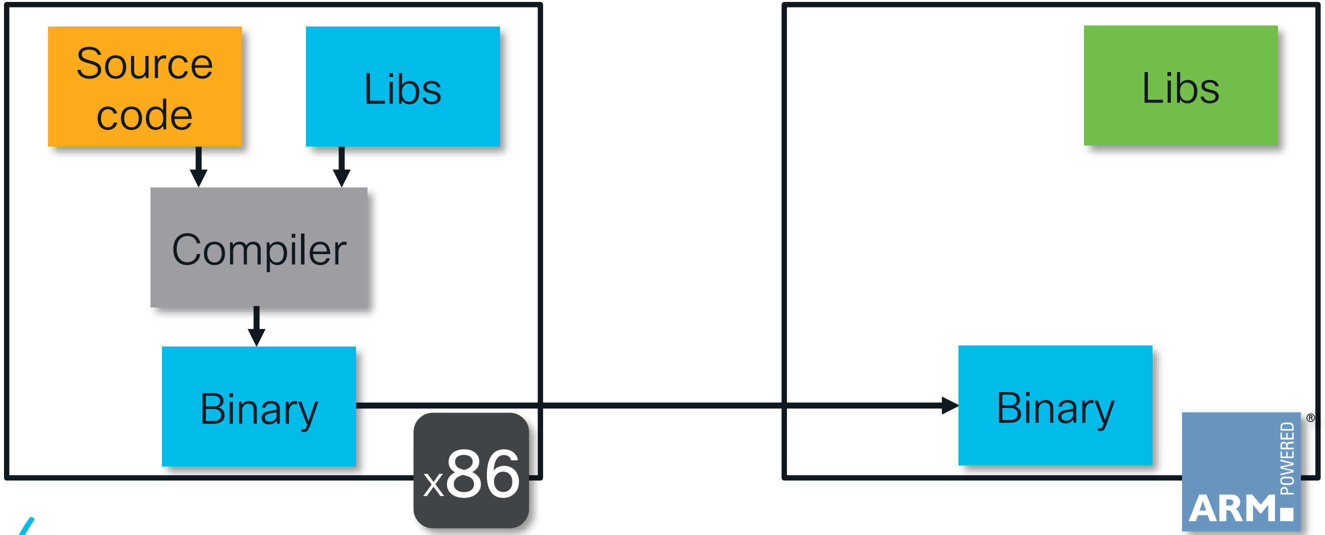
- In essence: RISC vs. CISC
- x86 traditionally targets peak performance, ARM energy efficiency
- For workstations & servers: choice of ISA is not technical
  - With the right hardware, everything can run performant
  - Code compatibility is most important
  - Today, x86 is dominant here
- For embedded, RISC makes sense
  - Smaller, cheaper, less power
  - Today, ARM is dominant here

CISC	RISC
Complex instructions	Simple instructions
More registers	Less Registers
Microprogramming	Complex compilers
Hardware-focused	Software-focused

The problem

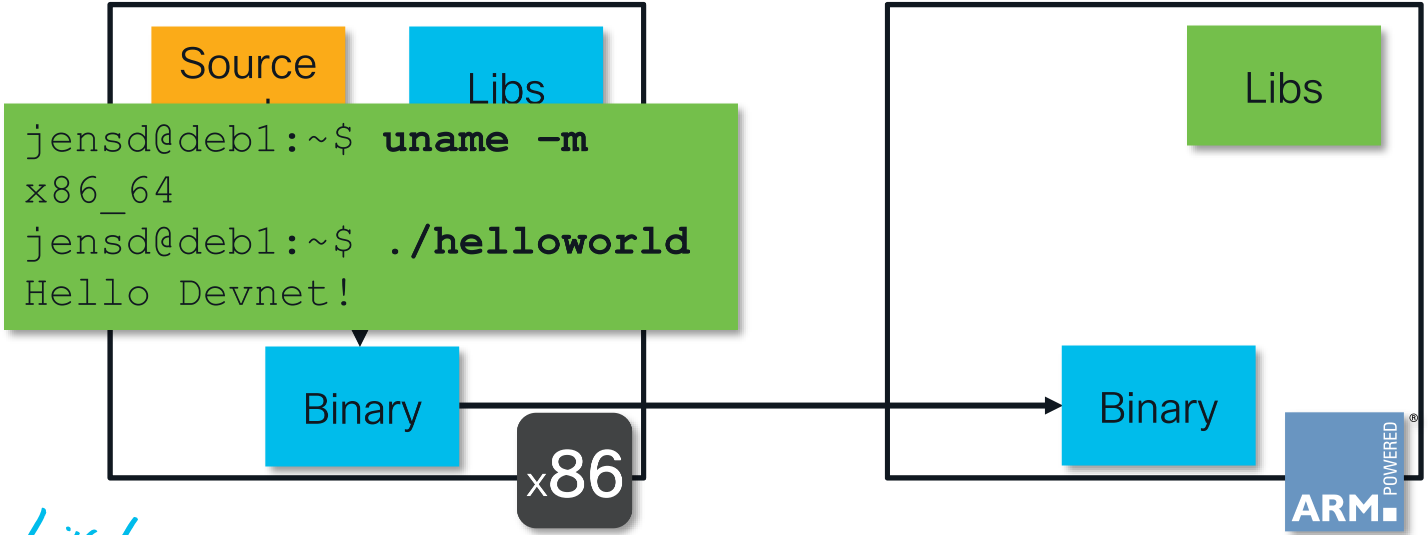
The problem: Current situation

- Mix of architectures is real today
- Developper workstation and destination on different architecure
- Compiled binary or container image: doesn't run



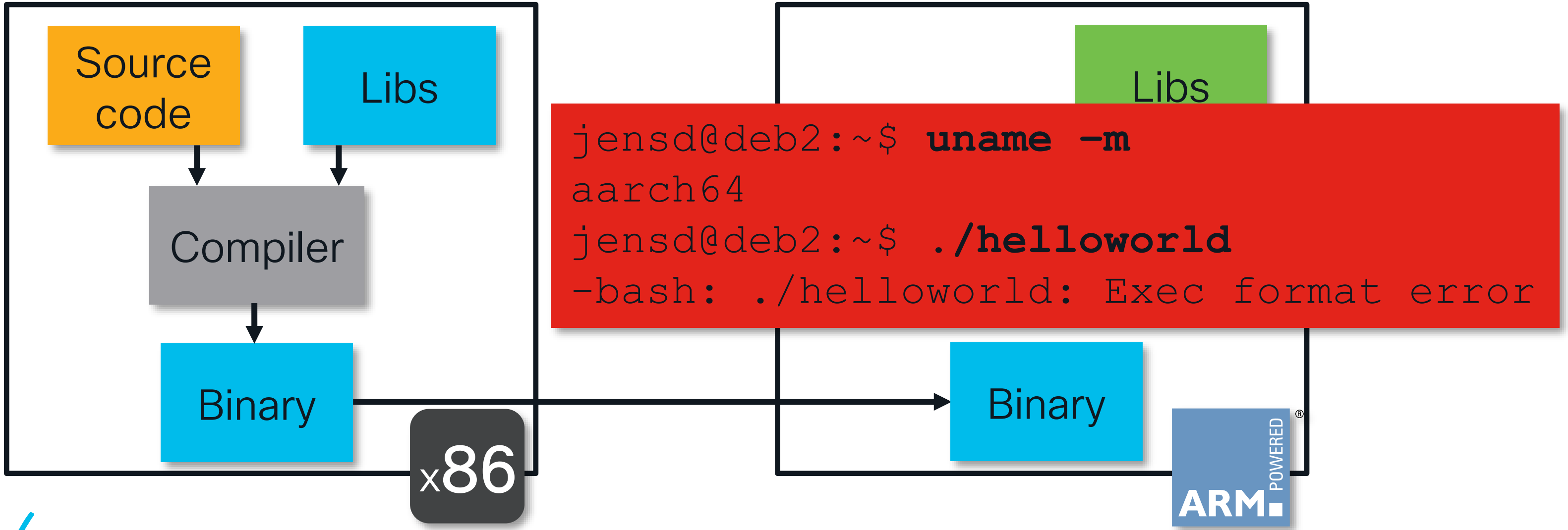
The problem: Current situation

- Mix of architectures is real today
- Developper workstation and destination on different architecure
- Compiled binary or container image: doesn't run



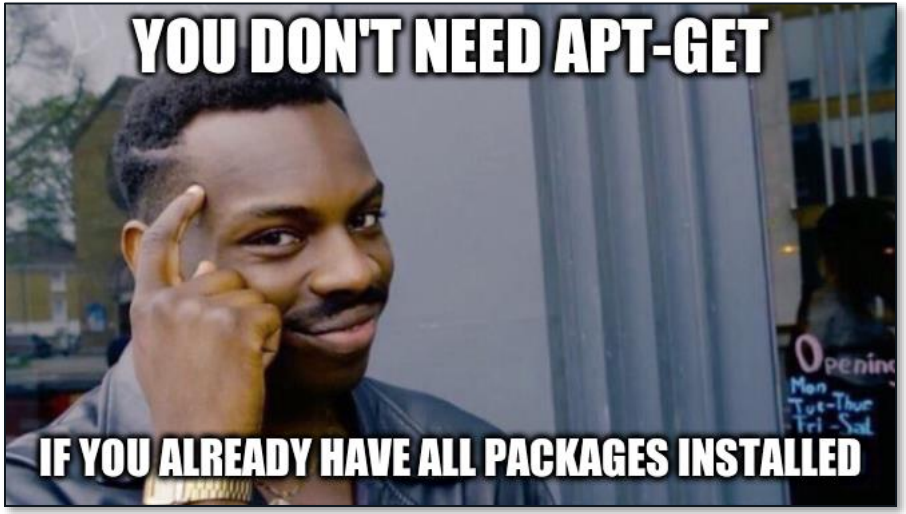
The problem: Current situation

- Mix of architectures is real today
- Developper workstation and destination on different architecure
- Compiled binary or container image: doesn't run



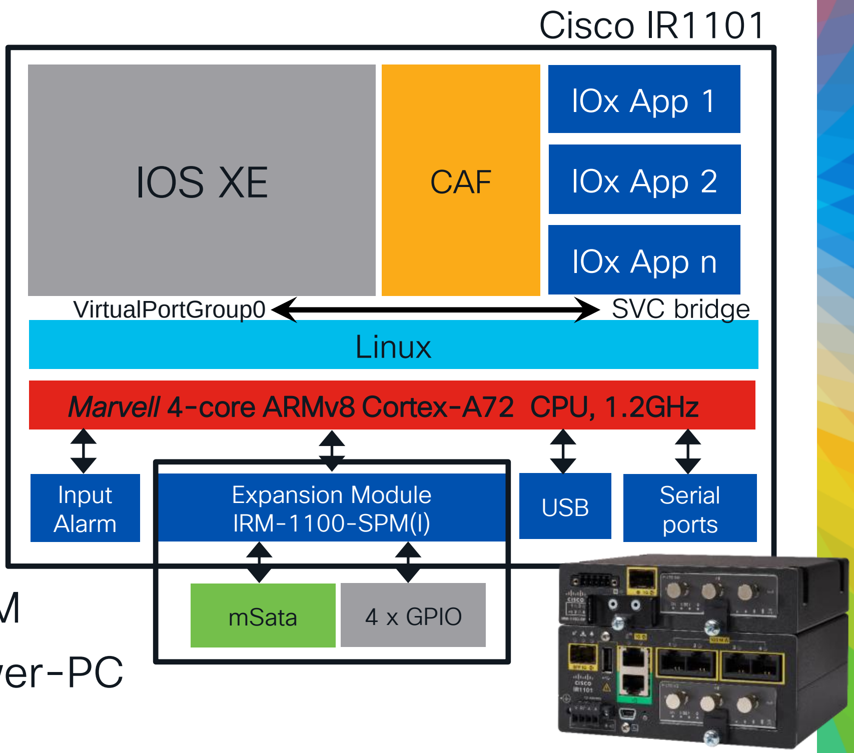
The problem – In practice

- Self-written code/tools/automation/containers:
  - Does not run on other architecture
  - Hard to test
- Common tools:
  - In theory: use package manager
  - In practice:
    - Missing packages
    - Missing dependencies
    - No package manager
    - Old/EOL version
    - Uncommon/custom distro
    - Dark site
    - …



The problem – Relevant for Cisco?

- Where is this relevant?
  - IOx and App-hosting
  - Guest-shell
  - Guest-OS (GOS)
  - Open Agent Container (OAC)
  - Low level troubleshooting tools
- Platforms:
  - **Data Center**: NX-OS: x86
  - **Service Provider**: IOS-XR: x86, 32/64 bit ARM
  - **Enterprise**: IOS-XE: x86, 32/64 bit ARM, Power-PC



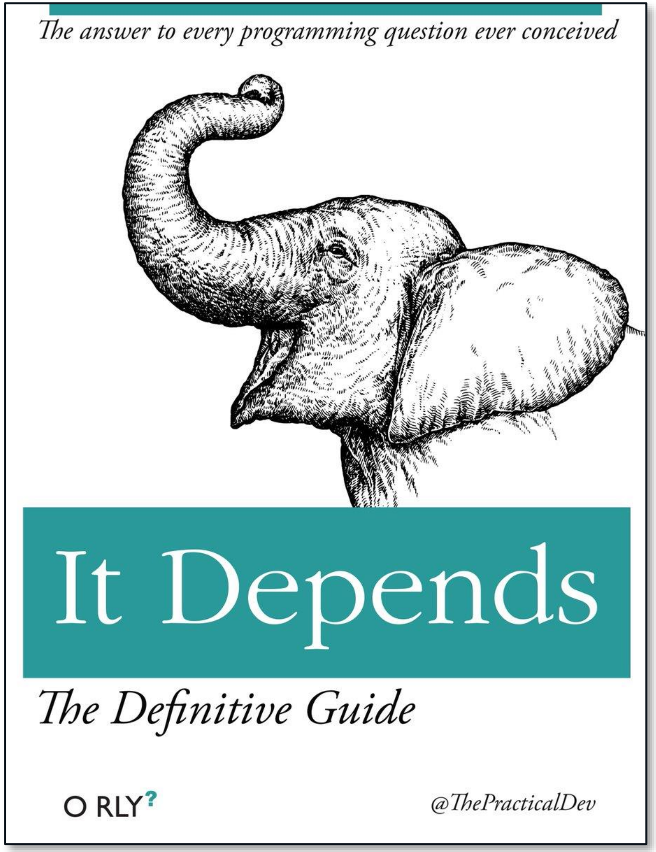
Solution

Solution

It depends...

- Use destination architecture
- Platform independent languages
- Cross compilation
- Emulation

- Automation is your friend
    - CI/CD
    - Docker BuildX

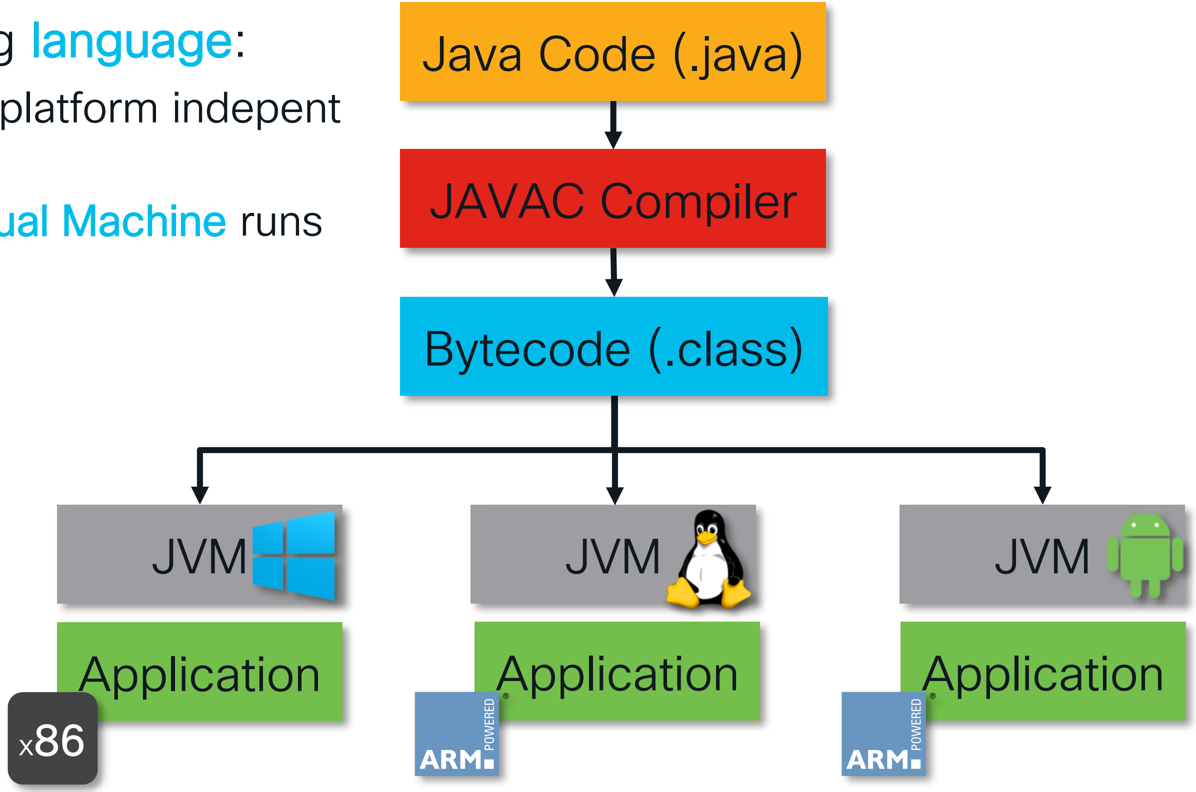


Solution 1: Use destination architecture

- Develop, build and test on native CPU architecture:
  - Physical hardware on destination architecture
  - Cloud-based solution:
    - AWS Gravitron
      - Custom Silicon with Neoverse
    - MS Azure
      - Ampere Altra
    - Google GCP Tau T2A
      - Ampere Altra
    - OCI Ampere A1
      - Ampere Altra

Solution 2: Platform Independent Languages

- Interpreted programming language:
  - Compile (at runtime) into platform indepent bytecode
  - Architecture-specific Virtual Machine runs bytecode
- Popular examples:
  - Java
  - Python
  - PHP
  - Bash
  - TCL ☺



DEMO: Platform Independent Languages

Source code:

```
jensd@Macbook ~ % cat test.py
#!/usr/bin/python3

import os
arch = os.uname().machine
print("Hello Devnet!")
print("This code is running on:", arch)
```

Run on x86:

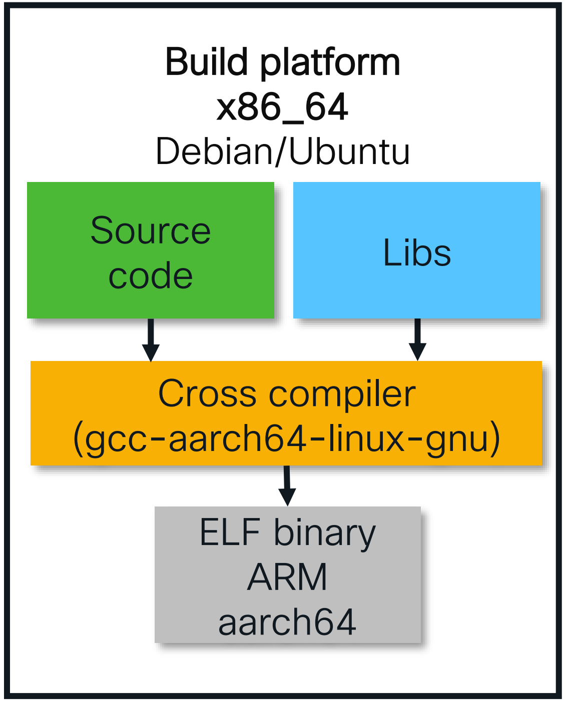
```
jensd@deb1:~$ uname -m
x86_64
jensd@deb1:~$ ./test.py
Hello Devnet!
This code is running on: x86_64
```

Run on ARM:

```
jensd@deb2:~$ uname -m
aarch64
jensd@deb2:~$ ./test.py
Hello Devnet!
This code is running on: aarch64
```

Solution 3: Cross Compiling

- Compiled language: C/C++/Go/Rust/...
- Build for platform X on platform Y
- Terminology:
  - Build platform: Architecture of build machine
  - Host platform: Architecture you are building for
  - Target platform: When building compiler tools
- Build platform can't run resulting binary!



DEMO: Cross Compiling – Prepare machine

For ARM (aarch64):

```
jensd@deb1:~$ sudo apt install gcc make gcc-aarch64-linux-gnu
binutils-aarch64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
...
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for libc-bin (2.28-10) ...
```

cross-compiler

For RISC-V (riscv64):

```
jensd@deb1:~$ sudo apt install gcc make gcc-riscv64-linux-gnu
binutils-riscv64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
...
db (2.8.5-2) ...
-bin (2.28-10) ...
```

assembler (as) , linker (ld) and binary tools

DEMO: Cross Compiling - Build

Source code:

```
jensd@deb1:~$ cat helloworld.c
#include<stdio.h>
int main()
{
        printf("Hello Devnet!\n");
        return 0;
}
```

Build on x86:

```
jensd@deb1:~$ uname -m
x86_64
jensd@deb1:~$ aarch64-linux-gnu-gcc helloworld.c -o helloworld-aarch64 --static
jensd@deb1:~$ file helloworld-aarch64
helloworld-aarch64: ELF 64-bit LSB executable, ARM aarch64, version 1
(GNU/Linux), statically linked,
BuildID[sha1]=eeb6cee92dd8cce1832cee6a3fb236cf659996b8, for GNU/Linux 3.7.0,
not stripped
```

cross-compiler

output (binary)

DEMO: Cross Compiling - Run

- Run on x86:

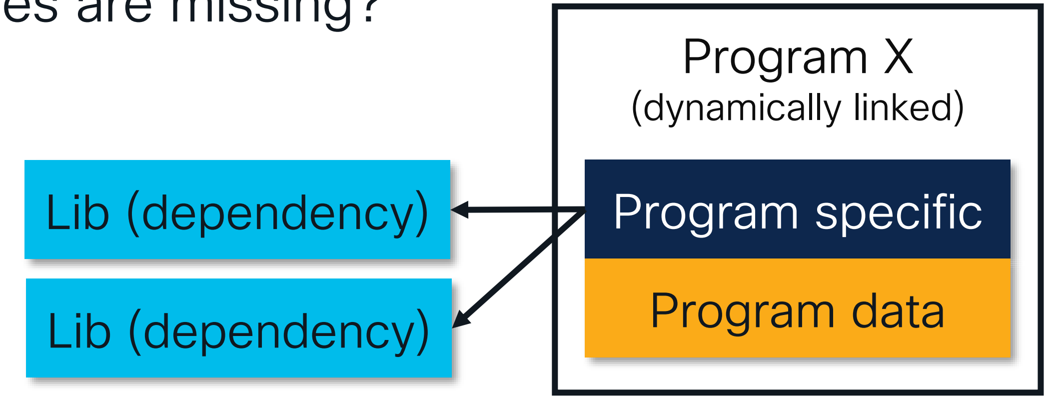
```
jensd@deb1:~$ uname -m
x86_64
jensd@deb1:~$ ./helloworld-aarch64
-bash: ./helloworld-aarch64: cannot execute binary file: Exec format error
```

- Run on ARM:

```
jensd@deb2:~$ uname -m
aarch64
jensd@deb2:~$ ./helloworld-aarch64
Hello Devnet!
```

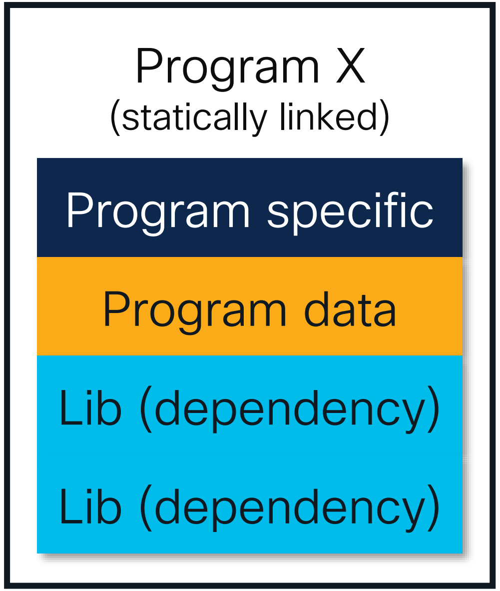
Cross Compiling – Dependencies

- What about dependencies?
- By default and recommended: Dynamic Linking
- Dependencies are external libs and architecture specific
- What if dependencies are missing?



Cross Compiling – Static Linking

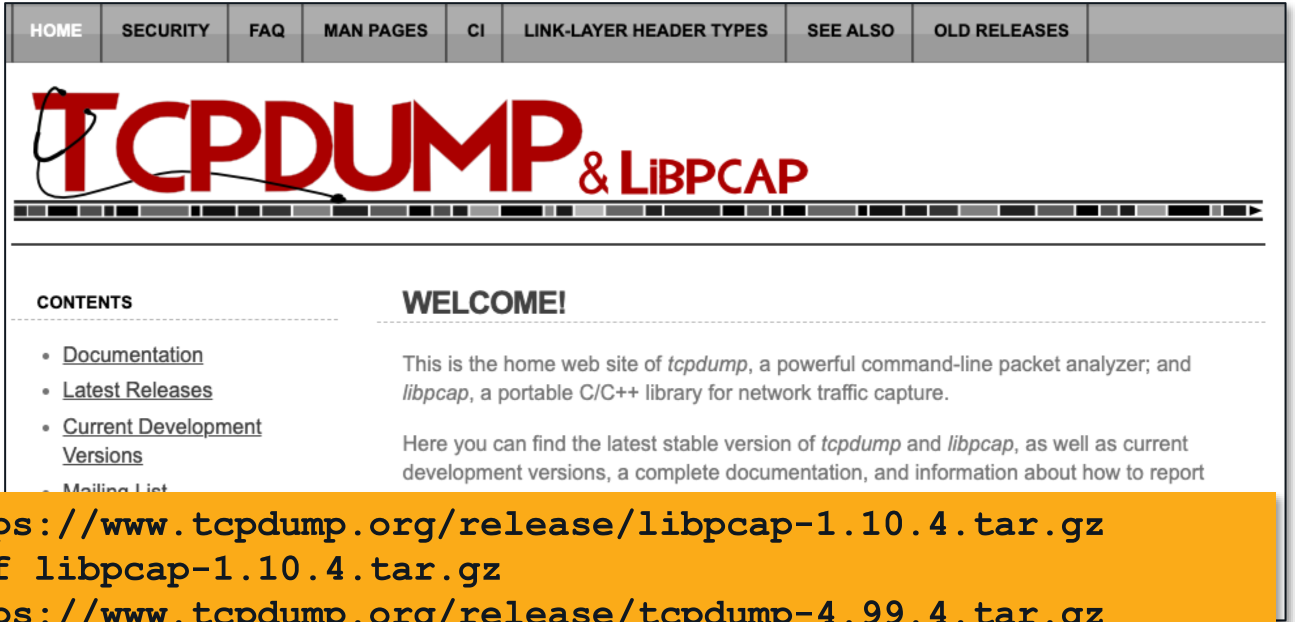
- Static linking: include dependencies in binary
- Not recommended*
  - **Unsecure**: no patches/updates in included libs
  - **Incompatibility**: conflicting libraries that do lower level system calls
  - **Larger** resulting binary
  - Can be **difficult**, especially with libc/glibc



*It works for me ☺

DEMO: Cross Compiling – Static Linking – Source

Build open source tool: tcpdump on x86_64 to use on aarch64



Get source code:

```
jensd@deb1:~$ wget https://www.tcpdump.org/release/libpcap-1.10.4.tar.gz
jensd@deb1:~$ tar -xvzf libpcap-1.10.4.tar.gz
jensd@deb1:~$ wget https://www.tcpdump.org/release/tcpdump-4.99.4.tar.gz
jensd@deb1:~$ tar -xvzf tcpdump-4.99.4.tar.gz
jensd@deb1:~$ cd libpcap-1.10.4/
jensd@deb1:~/libpcap-1.10.4$
```

DEMO: Cross Compiling – Static Linking – Build

- Build libpcap and tcpdump for aarch64 on x86_64 using musl

```
jensd@deb1:~/libpcap-1.10.4$ CC=aarch64-linux-musl-gcc
jensd@deb1:~/libpcap-1.10.4$ ./configure --build x86_64-pc-linux-gnu --host
aarch64-linux-gnu LDFLAGS="-static”
checking build system type... x86_64-pc-linux-gnu
checking host system type... aarch64-unknown-linux-gnu

jensd@deb1:~/libpcap-1.10.4$ make
```

host-architecture

static linking

build-architecture

```
jensd@deb1:~/libpcap-1.10.4$ cd ../tcpdump-4.99.4
jensd@deb1:~/tcpdump-4.99.4$./configure --build x86_64-pc-linux-gnu --host
aarch64-linux-gnu LDFLAGS="-static"
jensd@deb1:~/tcpdump-4.99.4$ make
jensd@deb1:~/tcpdump-4.99.4$ file tcpdump
tcpdump: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV), statically
linked, with debug_info, not stripped
```

DEMO: Cross Compiling – Static Linking – Run

- Run on x86:

```
jensd@deb1:~/tcpdump-4.99.1$ ldd tcpdump
	not a dynamic executable
jensd@deb1:~/tcpdump-4.99.1$ ./tcpdump
-bash: ./tcpdump: cannot execute binary file: Exec format error
```

- Run on ARM:

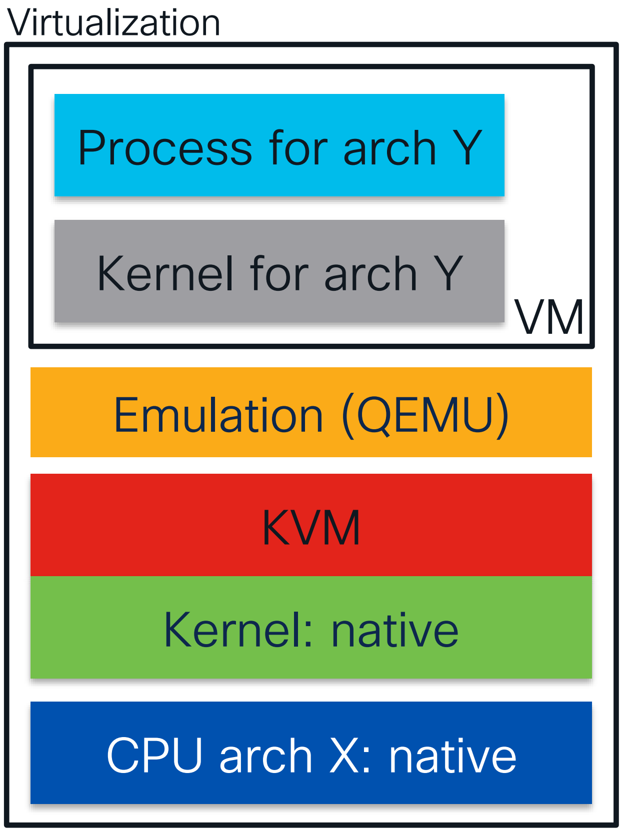
```
jensd@deb2:~$ uname -m
aarch64
jensd@deb2:~$ sudo ./tcpdump -i enp0s9
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on enp0s9, link-type EN10MB (Ethernet), snapshot length 262144
bytes
...
```

Solution 4: Emulation – QEMU Virtualization

- Emulate destination architecture on VM
  - Run virtual machine
  - Build/Test on VM as on native platform

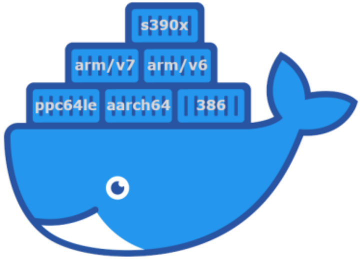
- QEMU: Generic and open source machine emulator and virtualizer

- Supports many CPU architectures
  - For example: ARM, alpha, MIPS, PowerPC, SPARC, RISC-V, s390x, ...



Solution 5: Emulation – User mode emulation

- Emulate with User mode emulation:
  - Run processes for another architecture
  - Build/Run Docker image/container for different arch
  - Test binaries as on native



- Binfmt: Kernel Support for miscellaneous Binary Formats
  - Instructs kernel to run binaries with QEMU

User mode emulation:

Process for CPU arch Y
User space: emulated
Kernel: native
CPU arch X: native

DEMO: Emulation – Preparation

- Install OS/Docker
- Install QEMU emulation binaries and binfmt

```
jensd@deb1:~$ sudo apt-get install qemu-user qemu-user-static binfmt-support
```

- User mode emulation binaries:

```
jensd@deb1:~$ ls /usr/bin/qemu-*static
/usr/bin/qemu-aarch64-static    /usr/bin/qemu-mips-static       /usr/bin/qemu-riscv32-static
/usr/bin/qemu-aarch64_be-static /usr/bin/qemu-mips64-static     /usr/bin/qemu-riscv64-static
/usr/bin/qemu-alpha-static      /usr/bin/qemu-mips64el-static   /usr/bin/qemu-s390x-static
/usr/bin/qemu-arm-static        /usr/bin/qemu-mipsel-static     /usr/bin/qemu-sh4-static
/usr/bin/qemu-armeb-static      /usr/bin/qemu-mipsn32-static    /usr/bin/qemu-sh4eb-static
/usr/bin/qemu-cris-static       /usr/bin/qemu-mipsn32el-static  /usr/bin/qemu-sparc-static
/usr/bin/qemu-hppa-static       /usr/bin/qemu-nios2-static      /usr/bin/qemu-sparc32plus-static
/usr/bin/qemu-i386-static       /usr/bin/qemu-or1k-static       /usr/bin/qemu-sparc64-static
/usr/bin/qemu-m68k-static       /usr/bin/qemu-ppc-static        /usr/bin/qemu-x86_64-static
/usr/bin/qemu-microblaze-static /usr/bin/qemu-ppc64-static      /usr/bin/qemu-xtensa-static
/usr/bin/qemu-micrazeel-static  /usr/bin/qemu-ppc64le-static    /usr/bin/qemu-xtensaeb-static
```

DEMO: Emulation – Test User mode emulation

- On x86:

```
jensd@deb1:~$ uname -m
x86_64
jensd@deb1:~$ docker run -v /usr/bin/qemu-aarch64-static:/usr/bin/qemu-
aarch64-static -i -t arm64v8/alpine
/ # uname -m
aarch64
```

Container needs emulation binary

- In the background:

```
jensd@deb1-x86-64:~$ ps aux | grep qemu
jensd     508680  0.6  0.8 1201160 47844 pts/1   Sl+  16:45   0:00
docker run -v /usr/bin/qemu-aarch64-static:/usr/bin/qemu-aarch64-static
-it --rm arm64v8/alpine
root      508741  0.6  0.1 226092  7700 pts/0    Ssl+ 16:45   0:00
/usr/libexec/qemu-binfmt/aarch64-binfmt-P /bin/sh /bin/sh
```

DEMO: Emulation – non-x86 Docker image 1/2

1) Dockerfile & Node.js source:

```
FROM arm64v8/alpine
COPY qemu-aarch64-static /usr/bin
RUN apk add --no-cache nodejs npm
COPY server.js .
EXPOSE 1337
CMD ["node","server.js"]
```

```
jensd@deb1-x86-64:~$ cat server.js
var http = require('http');
var os = require('os');

kernel=os.release();
arch=process.arch;

server = http.createServer(function (request, response) {
esponse.writeHead(200, {"Content-Type": "text/html"});
esponse.end("<h1>Node & Docker Running <br /> Kernel:
ernel+"<br />Arch:"+arch+"<h1>");

ver.listen(1337);
console.log("Node HTTP Server started at port 1337");
```

2) Build on x86

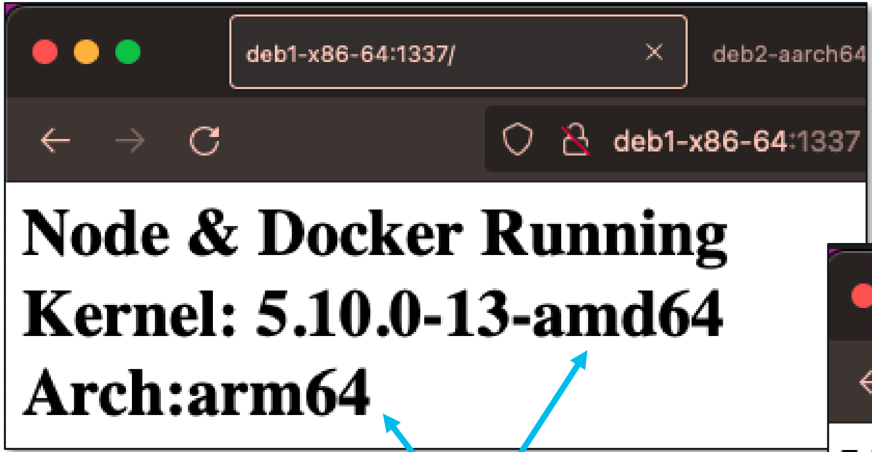
```
jensd@deb1:~$ docker build -t devnetjs .
```

2) Run

```
jensd@deb1:~$ docker run -ti --rm -p 1337:1337 devnetjs
Node HTTP Server started at port 1337
```

DEMO: Emulation – non-x86 Docker image 2/2

Run on x86:



Emulated

Run on ARM:

Native

Emulation – Testing

- Remember the cross-compiled tools we could not run?

- After installing QEMU and binfmt:

```
jensd@deb1:~$ uname -m
x86_64
jensd@deb1:~$ file helloworld-aarch64
helloworld-aarch64: ELF 64-bit LSB executable, ARM aarch64, version 1
(GNU/Linux), statically linked,
BuildID[sha1]=eeb6cee92dd8cce1832cee6a3fb236cf659996b8, for GNU/Linux
3.7.0, not stripped
jensd@deb1:~$ ./helloworld-aarch64
Hello Devnet!
```

Automation

- Integrate in CI/CD pipeline
  - Cross Compile, Emulation, Testing, ...
  - Gitlab runners for each arch
- Docker BuildX:
  - Using QEMU emulation support in the kernel
  - Building on multiple native nodes using same builder instance
  - Using stage in Dockerfile to cross-compile to different architectures

Summary

Situation today: mix of x86 and ARM

Code platform independent

Use Cross Compilation for compiled languages

Use Emulation to build containers and testing

Combine everything with automation

Thank you

cisco Live!

Let’s go