Software Tools Bootcamp

RISC-V ISA Tutorial — HPCA-21
08 February 2015

Albert Ou

UC Berkeley

aou@eecs.berkeley.edu

Preliminaries

To follow along, download these slides at

http://riscv.org/tutorial-hpca2015.html

Preliminaries

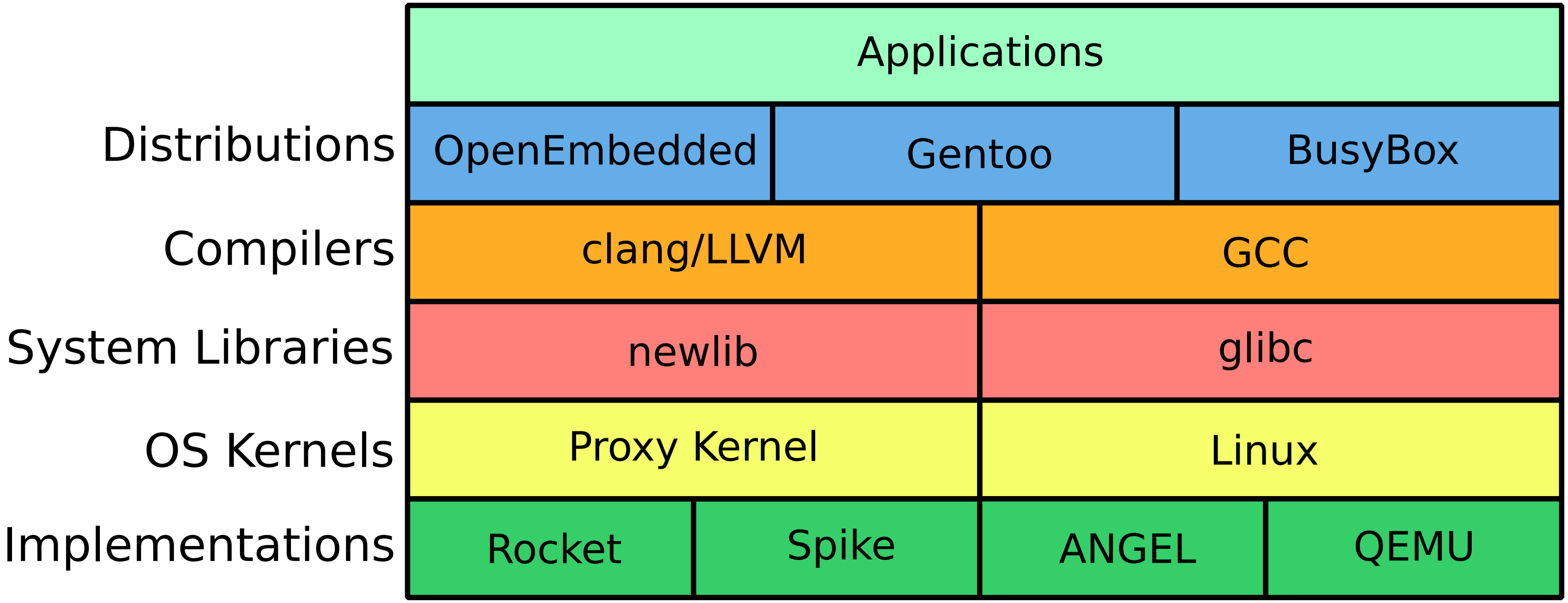
- **Shell commands** are prefixed by a “$” prompt.
- Due to time constraints, we will not be building everything from source in real-time.
  - Binaries have been prepared for you in the VM image.
  - Detailed build steps are documented here for completeness but are not necessary if using the VM.

- **Interactive** portions of this tutorial are denoted with:

```
$ echo 'Hello world'
```

- Also as a reminder, these slides are marked with an icon in the upper-right corner:

Software Stack



Applications

Distributions: OpenEmbedded, Gentoo, BusyBox

Compilers: clang/LLVM, GCC

System Libraries: newlib, glibc

OS Kernels: Proxy Kernel, Linux

Implementations: Rocket, Spike, ANGEL, QEMU

- Many possible combinations (and growing)
- But here we will focus on the most *common workflows* for RISC-V software development

Agenda

1. `riscv-tools` infrastructure
2. First Steps
3. Spike + Proxy Kernel
4. QEMU + Linux
5. Advanced Cross-Compiling
6. Yocto/OpenEmbedded

riscv-tools — Overview

“Meta-repository” with Git submodules for every stable component of the RISC-V software toolchain

Submodule	Contents
riscv-fesvr	RISC-V Frontend Server
riscv-isa-sim	Functional ISA simulator (“Spike”)
riscv-qemu	Higher-performance ISA simulator
riscv-gnu-toolchain	binutils, gcc, newlib, glibc, Linux UAPI headers
riscv-llvm	LLVM, riscv-clang submodule
riscv-pk	RISC-V Proxy Kernel
(riscv-linux)	Linux/RISC-V kernel port
riscv-tests	ISA assembly tests, benchmark suite

All listed submodules are hosted under the **riscv** GitHub organization: https://github.com/riscv

riscv-tools — Installation

- Build riscv-gnu-toolchain (*riscv*-*-elf* / newlib target), riscv-fesvr, riscv-isa-sim, and riscv-pk:
  (*pre-installed in VM*)

```
$ git clone https://github.com/riscv/riscv-tools
$ cd riscv-tools
$ git submodule update --init --recursive
$ export RISCV=<installation path>
$ export PATH=${PATH}:${RISCV}/bin
$ ./build.sh
```

- Build riscv-fesvr, riscv-isa-sim, and riscv-pk only:

```
$ ./build-spike-only.sh
```

riscv-tools — Platform Notes

- Ubuntu: See README.md
- OS X:

```
$ brew tap ucb-bar/riscv
$ brew install riscv-tools
```

- GCC dependencies:

  Bash, Binutils, Coreutils, Diffutils, Findutils, Gawk, Gettext, GMP, Grep, M4, GNU Make, MPC, MPFR, Patch, Perl, Sed, Tar, Texinfo

riscv-tools — Utilities

```
$ ls ${RISCV}/bin
elf2hex                             riscv64-unknown-elf-gcov
fesvr-eth                           riscv64-unknown-elf-gprof
fesvr-rs232                         riscv64-unknown-elf-ld
fesvr-zedboard                      riscv64-unknown-elf-ld.bfd
riscv64-unknown-elf-addr2line       riscv64-unknown-elf-nm
riscv64-unknown-elf-ar              riscv64-unknown-elf-objcopy
riscv64-unknown-elf-as              riscv64-unknown-elf-objdump
riscv64-unknown-elf-c++             riscv64-unknown-elf-ranlib
riscv64-unknown-elf-c++filt         riscv64-unknown-elf-readelf
riscv64-unknown-elf-cpp             riscv64-unknown-elf-size
riscv64-unknown-elf-elfedit         riscv64-unknown-elf-strings
riscv64-unknown-elf-g++             riscv64-unknown-elf-strip
riscv64-unknown-elf-gcc             spike
riscv64-unknown-elf-gcc-4.9.2       spike-dasm
riscv64-unknown-elf-gcc-ar          termios-xspike
riscv64-unknown-elf-gcc-nm          xspike
riscv64-unknown-elf-gcc-ranlib
```

Tutorial VM Structure

- By convention, **$RISCV** refers to the top-level directory where RISC-V tools are installed.

  `~/bar/riscv` in your VM

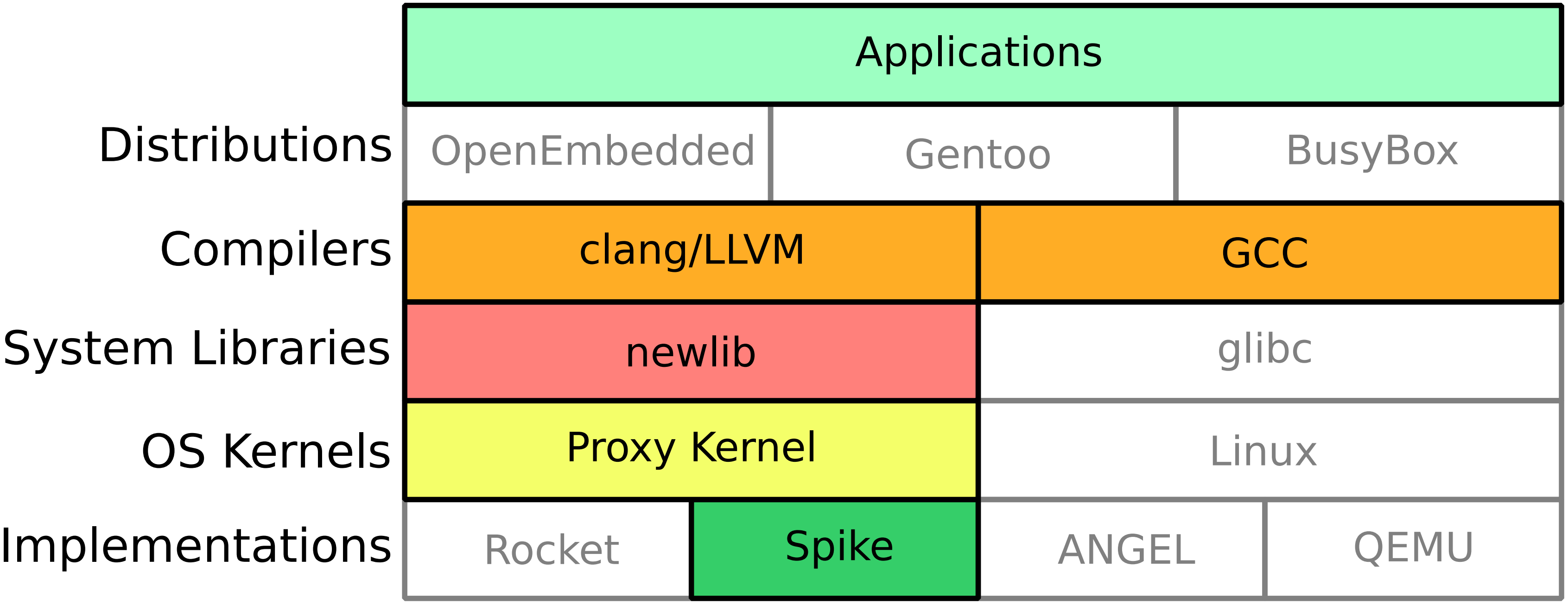
- Double-check that `${RISCV}/bin` is in your `$PATH`.

- In subsequent slides, **$SRCDIR** refers to the directory into which riscv-tools is cloned.

  `~/bar/rocket-chip/riscv-tools` in your VM

Common Workflow — Spike + pk

- Use case: Embedded / single application
- Target triplet: *riscv*-*-elf*



Applications

Distributions: OpenEmbedded | Gentoo | BusyBox

Compilers: clang/LLVM | GCC

System Libraries: newlib | glibc

OS Kernels: Proxy Kernel | Linux

Implementations: Rocket | Spike | ANGEL | QEMU

First Steps — Cross-Compiling

- Write and compile a test program:

```
$ cat > hello.c <<'EOF'
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
EOF

$ riscv64-unknown-elf-gcc -O2 -o hello hello.c
```

- Inspect the output binary:

```
$ riscv64-unknown-elf-readelf -a hello | less
$ riscv64-unknown-elf-objdump -d hello | less
```

- Note that newlib supports only static linking

First Steps — Using Spike

- Run your test program:

```
$ spike pk hello
```

- Proxy kernel is located at
  ${RISCV}/riscv64-unknown-elf/bin/pk

- Invoke interactive debug mode in Spike with -d command line flag or SIGINT (^C)
- Press return key to single-step through instructions
- Enter q to quit or rs to continue running silently
- Consult riscv-isa-sim README.md for how to print register and memory contents, set breakpoint conditions, etc.

riscv-llvm

- Build Clang/LLVM (*pre-installed in VM*):

```
$ mkdir build
$ cd build
$ ${SRCDIR}/riscv-tools/riscv-llvm/configure
  --prefix=$RISCV --enable-optimized --enable-targets=riscv
$ make && make install
```

- Compile a test program:

```
$ clang -target riscv -O2 -S
 -isystem ${RISCV}/riscv64-unknown-elf/include hello.c
```

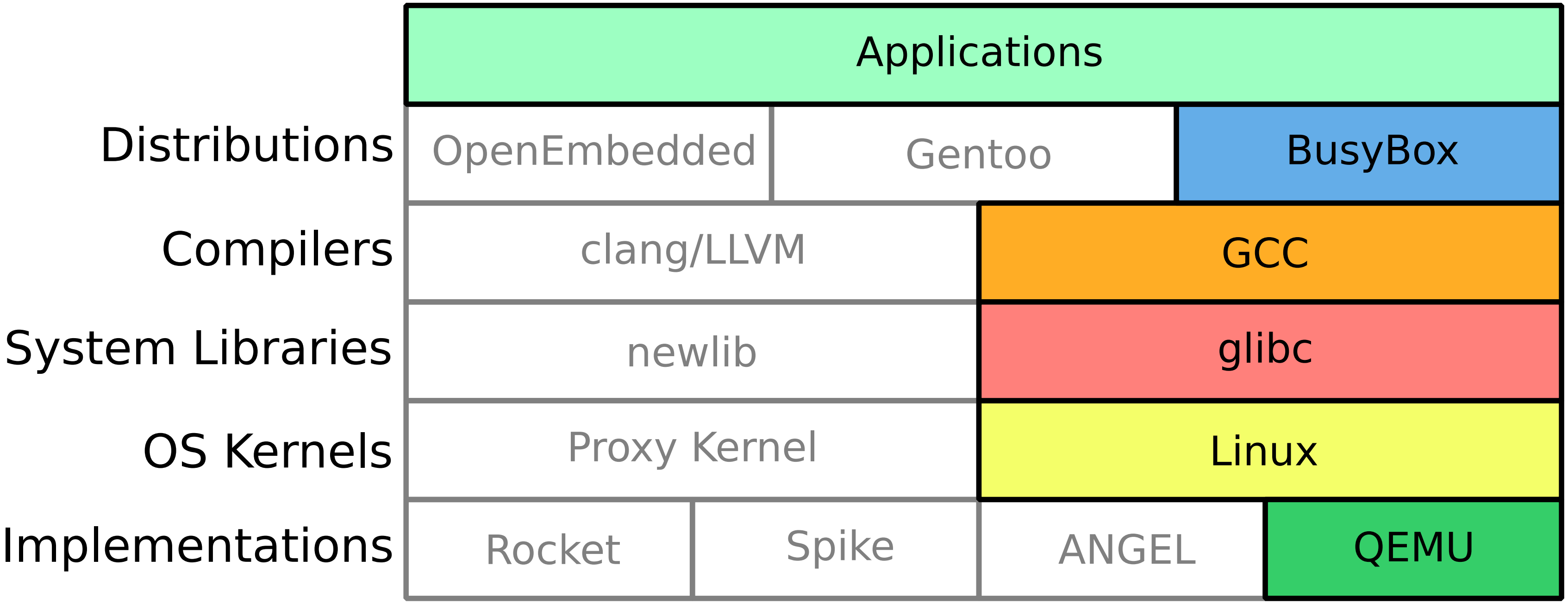
- Assemble and link with gcc/binutils:

```
$ riscv64-unknown-elf-gcc –o hello hello.S
```

  - `llvm-as` and `lld` support is under development

Workflow — QEMU + Linux

- Use case: Simple POSIX environment
- Target triplet: *riscv*-*-linux-gnu*



	Applications			
Distributions	OpenEmbedded	Gentoo		BusyBox
Compilers	clang/LLVM		GCC	
System Libraries	newlib		glibc	
OS Kernels	Proxy Kernel		Linux	
Implementations	Rocket	Spike	ANGEL	QEMU

“Linux From Scratch”

- Build order for a minimal GNU/Linux system:
  1. `riscv64-unknown-linux-gnu-gcc`, glibc
  2. Linux kernel
  3. BusyBox
  4. Root filesystem image

- sysroot:
  - Analogous to a chroot jail
  - Mimics the layout of the target RISC-V installation
  - Used by the cross-compiler as a prefix for header and library search paths

Linux Toolchain

- Not part of build.sh by default

- Build *riscv64-unknown-linux-gnu* toolchain:
  (*pre-installed in VM*)

```
$ cd ${SRCDIR}/riscv-tools/riscv-gnu-toolchain
$ ./configure --prefix=$RISCV
$ make linux
```

Side Note — RV32

- Generate RV32 code: `-m32` or `-march=RV32I[…]`
- Build pure *riscv32-unknown-linux-gnu* toolchain:

```
$ cd ${SRCDIR}/riscv-tools/riscv-gnu-toolchain
$ ./configure --prefix=$RISCV
$ make XLEN=32 linux
```

- 32-bit libraries installed into `${RISCV}/sysroot32`
- *TODO*: multilib support

Linux/RISC-V kernel — Fetching

- Obtain upstream sources:

```
$ curl -L
https://www.kernel.org/pub/linux/kernel/v3.x/linu
x-3.14.32.tar.xz | tar -xJf - -C ${SRCDIR}
```

- Overlay RISC-V architecture-specific subtree:

```
$ cd ${SRCDIR}/linux-3.14.32
$ git init
$ git remote add origin
  https://github.com/riscv/riscv-linux.git
$ git fetch
$ git checkout -f -t origin/master
```

Linux/RISC-V kernel — Building

- Populate default `.config`:

```
$ make ARCH=riscv qemu_defconfig
```

  - Selects virtio guest drivers for QEMU
  - Use `defconfig` instead to select HTIF drivers for Spike

- (*Optional*) Edit Kconfig options:

```
$ make ARCH=riscv menuconfig
```

- Compile kernel image:

```
$ make ARCH=riscv vmlinux
```

BusyBox — Fetching

- “Swiss Army Knife of Embedded Linux”
- Combination of essential Unix utilities in one executable

- Download sources:

```
$ curl -L http://www.busybox.net/downloads/busybox-1.23.1.tar.bz2 | tar -xjf - -C ${SRCDIR}
$ cd ${SRCDIR}/busybox-1.23.1
```

BusyBox — Building

- Populate recommended configuration:

```
$ curl -L –o .config
  http://riscv.org/tutorial-hpca2015/config-
busybox
```

```
$ make menuconfig
```

- Need at minimum `init(8)`, `ash(1)`, `mount(8)`, etc.
- Defaults to dynamic linking

- Compile:

```
$ make
```

Disk Image Creation

- Format root filesystem:

```
$ dd if=/dev/zero of=root.bin bs=1M count=64
$ mkfs.ext2 –F root.bin
```

- Mount as loop device:

```
$ mkdir -p mnt
$ sudo mount -o loop root.bin mnt
```

- Create directory hierarchy:

```
$ cd mnt
$ sudo mkdir -p dev proc sys bin sbin lib
  usr/{bin,sbin,lib} tmp root
```

Disk Image Creation

- Copy shared libraries:

```
$ sudo cp -R ${RISCV}/sysroot64/lib .
```

- Copy BusyBox:

```
$ sudo cp ${SRCDIR}/busybox-1.23.1/busybox bin/
$ sudo ln -s ../bin/busybox sbin/init
```

- Populate `inittab(5)`:

```
$ sudo curl -L –o etc/inittab
  http://riscv.org/tutorial-hpca2015/inittab
```

- Unmount:

```
$ cd .. && sudo umount mnt
```

Hello World Revisited

- Compile a test program:

```
$ riscv64-unknown-linux-gnu-gcc -O2 -o hello hello.c
```

- Inspect the output binary:
  - Notice the `.dynamic` section, PLT, etc.

```
$ riscv64-unknown-linux-gnu-readelf -a hello | less
$ riscv64-unknown-linux-gnu-objdump -d hello | less
```

- Add to root disk image:

```
$ sudo mount root.bin mnt
$ sudo cp hello mnt/usr/bin/
$ sudo umount mnt
```

Clang/LLVM Revisited

- Compile a test program:

```
$ clang -target riscv -isysroot ${RISCV}/sysroot64
  -O2 –S hello.c
```

- Assemble and link:

```
$ riscv64-unknown-linux-gnu-gcc -o hello hello.S
```

riscv-qemu — Installation

- Build QEMU (*pre-installed in VM*):

```
$ mkdir build && cd build
$ ${SRCDIR}/riscv-tools/riscv-qemu/configure
  --target-list=riscv-softmmu --prefix=${RISCV}
  --disable-riscv-htif
$ make && make install
```

- Devices: 8250 UART and virtio backends
- Alternatively, omit `--disable-riscv-htif` to enable support for HTIF block devices instead of virtio (See README.md)

riscv-qemu — Usage

- Boot Linux with SCSI root device:

```
$ qemu-system-riscv -kernel vmlinux –nographic
  -device virtio-scsi-device
  -drive file=root.bin,format=raw,id=hd0
  -device scsi-hd,drive=hd0
```

Adjust paths to `vmlinux` and `root.bin` as necessary

- Or use simple shell alias in VM:

```
$ qemu-linux
```

- Run `halt` in target and Ctrl-A-X to quit QEMU

riscv-qemu — Usage

- Boot Linux with SCSI root device:

```
$ qemu-system-riscv -kernel vmlinux –nographic
  -device virtio-scsi-device
  -drive file=root.bin,format=raw,id=hd0
  -device scsi-hd,drive=hd0
```

Adjust paths to `vmlinux` and `root.bin` as necessary

- Add a network interface (SLIRP backend):

```
-netdev user,id=net0
-device virtio-net-device,netdev=net0
```

- Bridge to physical Ethernet (macvtap, TUN/TAP):

```
-netdev tap,ifname=tap0,script=no,downscript=no,vhost=on,netdev=net0
```

Advanced Cross-Compilation

- Power of abstraction: Most software ports should require few or no source changes
- Ideally, autotools-based packages should cross-compile using `--host=riscv{32,64}-unknown-linux-gnu`
- Caveats:
  - May have to add riscv to config.sub
  - May have to point pkg-config(1) at sysroot

```
$ unset PKG_CONFIG_DIR
$ export PKG_CONFIG_LIBDIR=${RISCV}/sysroot64/usr/lib/pkgconfig
$ export PKG_CONFIG_SYSROOT_DIR=${RISCV}/sysroot64
```

Example — GNU Bash

- Fetch and extract source:

```
$ curl -L https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz |
  tar -xz -C "${SRCDIR}"
$ cd "${SRCDIR} "/bash-4.3
```

- Apply maintenance patches:

```
$ curl -l ftp://ftp.gnu.org/gnu/bash/bash-4.3-patches/ |
  sort -u |
  while read -r p ; do [ "${p%-*[0-9]}" = bash43 ] &&
    echo "https://ftp.gnu.org/gnu/bash/bash-4.3-patches/${p}" ;
  done |
  xargs curl | patch -N -p0
```

Example — GNU Bash

- Preset the results of certain autoconf checks that cannot be performed while cross-compiling:

```
$ cat > config.cache <<'EOF'
ac_cv_rl_version=6.3
bash_cv_func_ctype_nonascii=no
bash_cv_dup2_broken=no
bash_cv_pgrp_pipe=no
bash_cv_sys_siglist=yes
bash_cv_under_sys_siglist=yes
bash_cv_wexitstatus_offset=0
bash_cv_opendir_not_robust=no
bash_cv_ulimit_maxfds=yes
bash_cv_getenv_redef=yes
bash_cv_getcwd_malloc=yes
bash_cv_func_sigsetjmp=present
bash_cv_func_strcoll_broken=no
bash_cv_func_snprintf=yes
bash_cv_func_vsnprintf=yes
bash_cv_printf_a_format=yes
bash_cv_must_reinstall_sighandlers=no
bash_cv_job_control_missing=present
bash_cv_sys_named_pipes=present
bash_cv_wcontinued_broken=no
bash_cv_unusable_rtsigs=no
EOF
```

Example — GNU Bash

- Patch `support/config.sub` to recognize the `riscv` machine type:

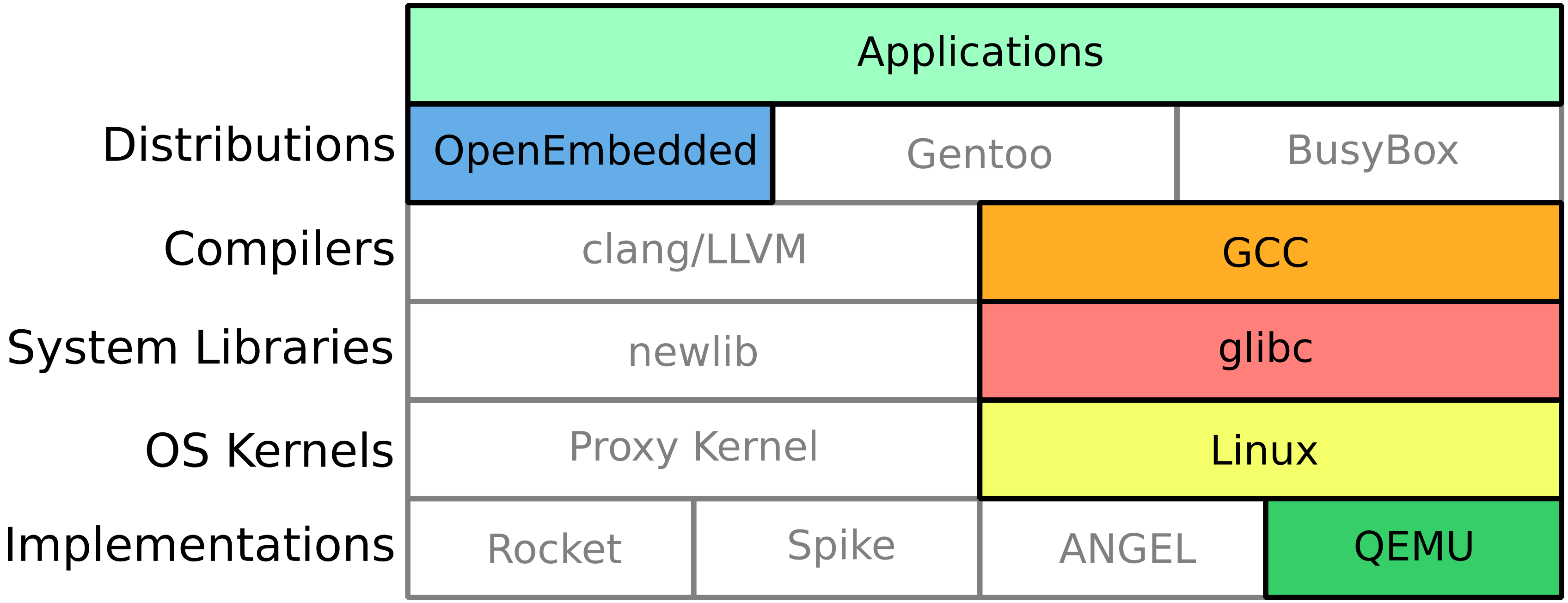
```
--- bash-4.3/support/config.sub  2013-12-17 07:49:47.000000000 -0800
+++ bash-4.3/support/config.sub  2014-08-07 18:50:10.001598071 -0700
@@ -302,4 +302,5 @@
     | powerpc | powerpc64 | powerpc64le | powerpcle \
     | pyramid \
+    | riscv* \
     | rl78 | rx \
     | score \
```

- Compile and install into sysroot:

```
$ ./configure --host=riscv64-unknown-linux-gnu
  --prefix=/usr --bindir=/bin --without-bash-malloc
  --disable-nls --config-cache
$ make
$ make install DESTDIR="${RISCV}"/sysroot64
```

Workflow — QEMU + OpenEmbedded

- Use case: Full-featured userland with package management and automatic dependency resolution



	Applications			
Distributions	OpenEmbedded	Gentoo	BusyBox	
Compilers	clang/LLVM		GCC	
System Libraries	newlib		glibc	
OS Kernels	Proxy Kernel		Linux	
Implementations	Rocket	Spike	ANGEL	QEMU

Why we need a Linux Distribution

- To build an application for RISC-V, you need to:
  - Download and build the RISC-V toolchain + Linux
  - Download, patch and build application + dependencies
  - Create an image and run it in QEMU or on hardware
- Problems with this approach:
  - **Error-prone**: Easy to corrupt FS or get a step wrong
  - **Reproducibility**: Others can’t easily reuse your work
  - **Rigidity**: If a dependency changes, need to do it all over
- We need a Linux distribution!
  - Automatic **build process** with dependency tracking
  - Ability to distribute binary **packages and SDKs**

(Slide courtesy of Martin Maas)

RISCV-POKY: A PORT OF THE YOCTO PROJECT

- We ported the **Yocto Project**
  - Official Linux Foundation Workgroup, supported by a large number of industry partners
  - Part I: **Collection of hundreds of recipes** (scripts that describe how to build packages for different platforms), shared with OpenEmbedded project
  - Part II: **Bitbake, a parallel build system** that takes recipes and fetches, patches, cross-compiles and produces packages (RPM/DEB), images, SDKs, etc.
- Focus on build process and customizability

(Slide courtesy of Martin Maas)

GETTING STARTED WITH RISCV-POKY

- **Let's build a full Linux system** including the GCC toolchain, Linux, QEMU + a large set of packages (including `bash`, `ssh`, `python`, `perl`, `apt`, `wget`,...)
- **Step I**: Clone riscv-poky:

```
git clone https://github.com:riscv/riscv-poky.git
```

- **Step II**: Set up the build system:

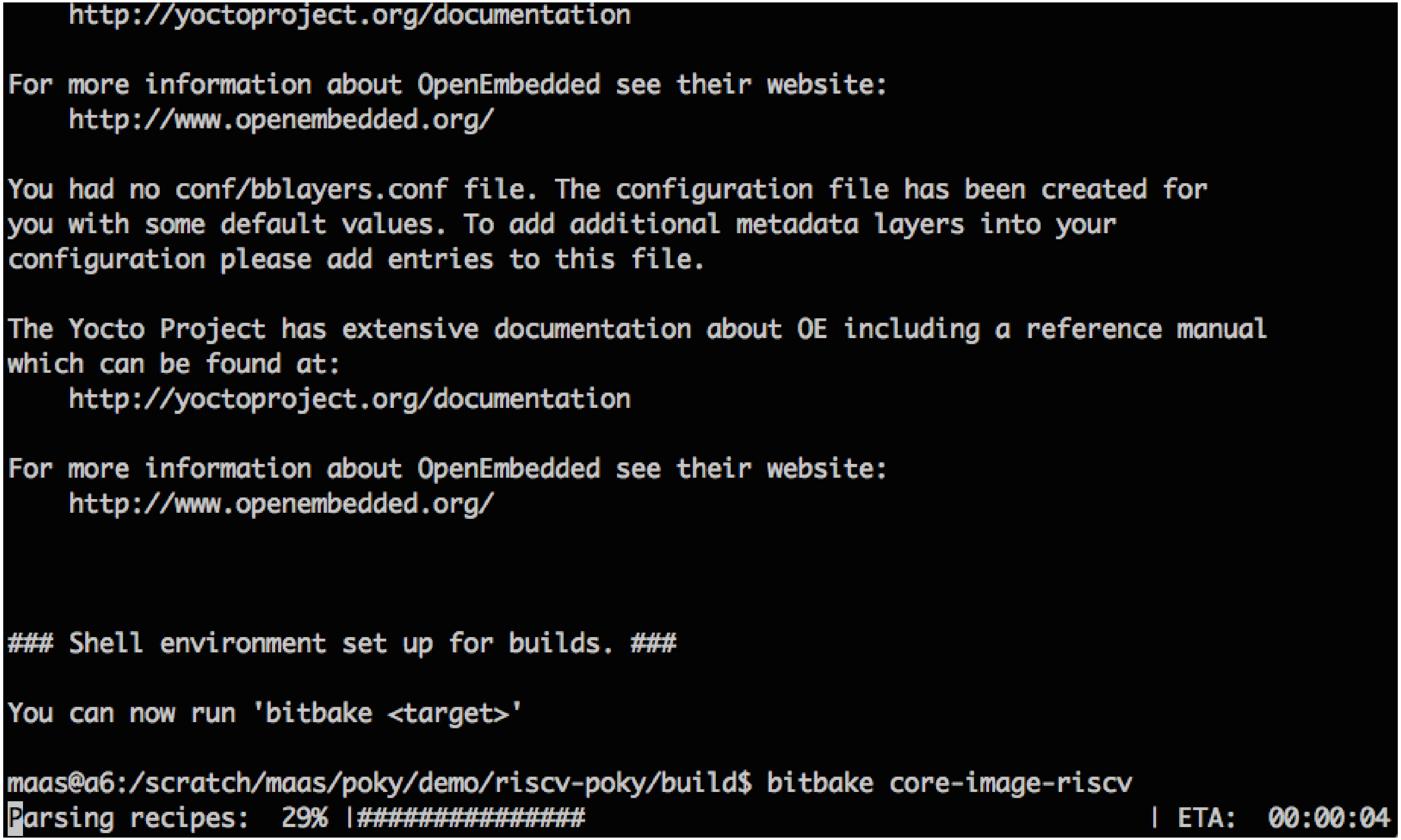
```
source oe-init-build-env
```

- **Step III**: Build an image (may take hours!):

```
bitbake core-image-riscv
```

(Slide courtesy of Martin Maas)

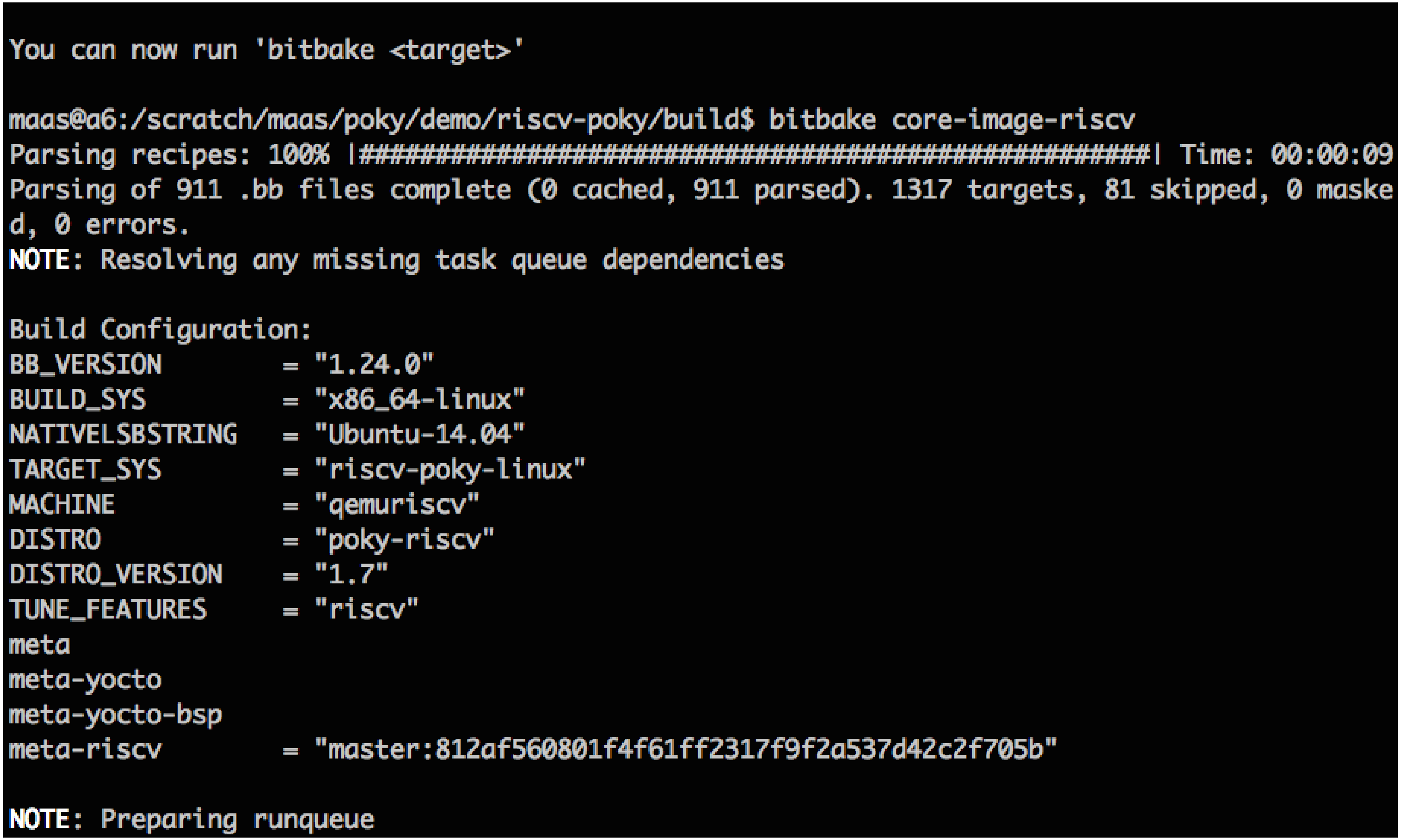
BUILD AN IMAGE (1/3)



```
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

You had no conf/bblayers.conf file. The configuration file has been created for
you with some default values. To add additional metadata layers into your
configuration please add entries to this file.

The Yocto Project has extensive documentation about OE including a reference manual
which can be found at:
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/



### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

maas@a6:/scratch/maas/poky/demo/riscv-poky/build$ bitbake core-image-riscv
Parsing recipes:  29% |###############                              | ETA:  00:00:04
```

(Slide courtesy of Martin Maas)

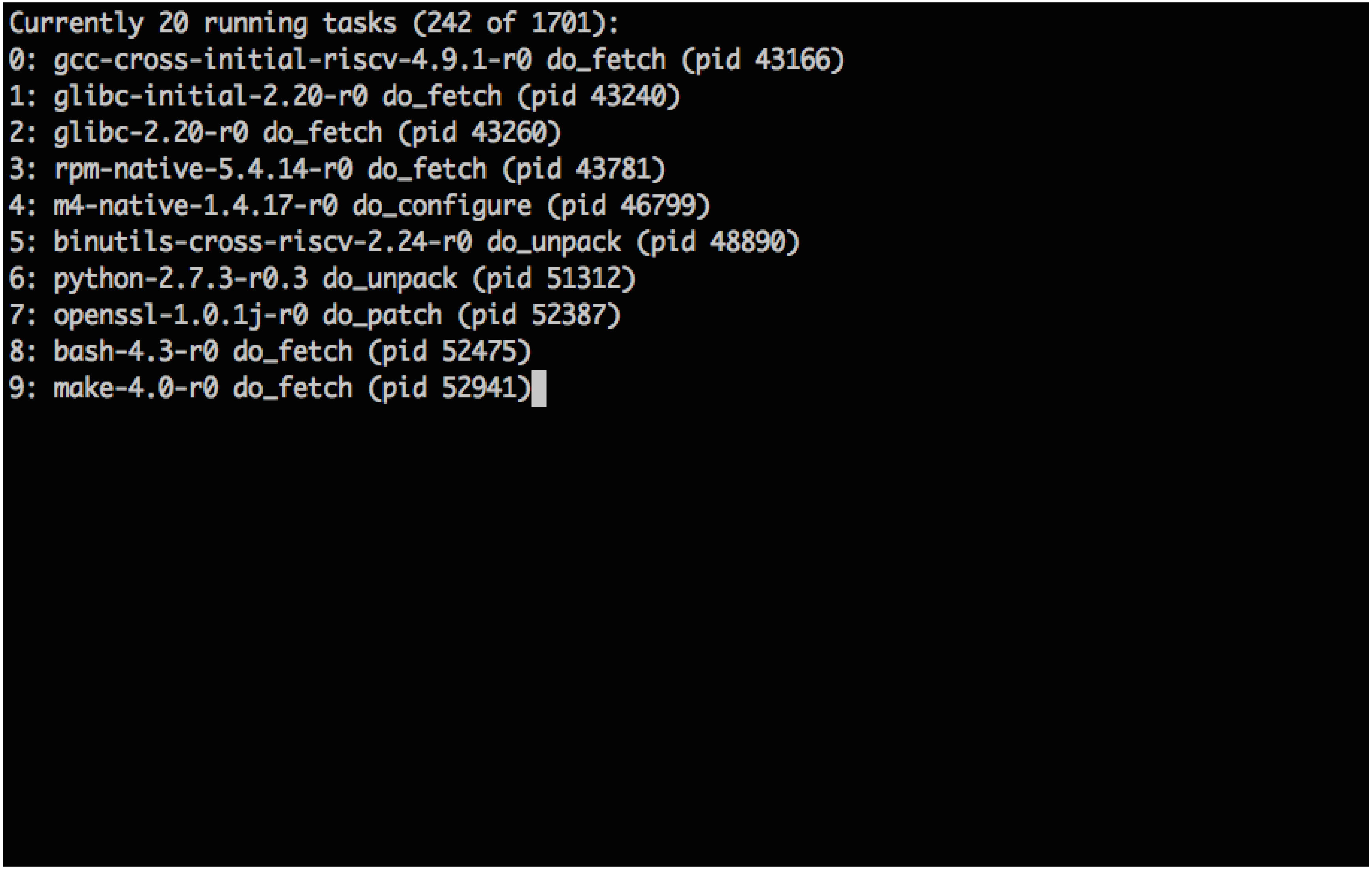
BUILD AN IMAGE (2/3)



```
You can now run 'bitbake <target>'

maas@a6:/scratch/maas/poky/demo/riscv-poky/build$ bitbake core-image-riscv
Parsing recipes: 100% |####################################################| Time: 00:00:09
Parsing of 911 .bb files complete (0 cached, 911 parsed). 1317 targets, 81 skipped, 0 maske
d, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION        = "1.24.0"
BUILD_SYS         = "x86_64-linux"
NATIVELSBSTRING   = "Ubuntu-14.04"
TARGET_SYS        = "riscv-poky-linux"
MACHINE           = "qemuriscv"
DISTRO            = "poky-riscv"
DISTRO_VERSION    = "1.7"
TUNE_FEATURES     = "riscv"
meta
meta-yocto
meta-yocto-bsp
meta-riscv        = "master:812af560801f4f61ff2317f9f2a537d42c2f705b"

NOTE: Preparing runqueue
```

(Slide courtesy of Martin Maas)

BUILD AN IMAGE (3/3)



```
Currently 20 running tasks (242 of 1701):
0: gcc-cross-initial-riscv-4.9.1-r0 do_fetch (pid 43166)
1: glibc-initial-2.20-r0 do_fetch (pid 43240)
2: glibc-2.20-r0 do_fetch (pid 43260)
3: rpm-native-5.4.14-r0 do_fetch (pid 43781)
4: m4-native-1.4.17-r0 do_configure (pid 46799)
5: binutils-cross-riscv-2.24-r0 do_unpack (pid 48890)
6: python-2.7.3-r0.3 do_unpack (pid 51312)
7: openssl-1.0.1j-r0 do_patch (pid 52387)
8: bash-4.3-r0 do_fetch (pid 52475)
9: make-4.0-r0 do_fetch (pid 52941)
```

(Slide courtesy of Martin Maas)

GETTING STARTED WITH RISCV-POKY

- **Let’s build a full Linux system** including the GCC toolchain, Linux, QEMU + a large set of packages (including `bash`, `ssh`, `python`, `perl`, `apt`, `wget`,…)
- **Step I**: Clone riscv-poky:

```
git clone https://github.com:riscv/riscv-poky.git
```

- **Step II**: Set up the build system:

```
source oe-init-build-env
```

- **Step III**: Build an image (may take hours!):

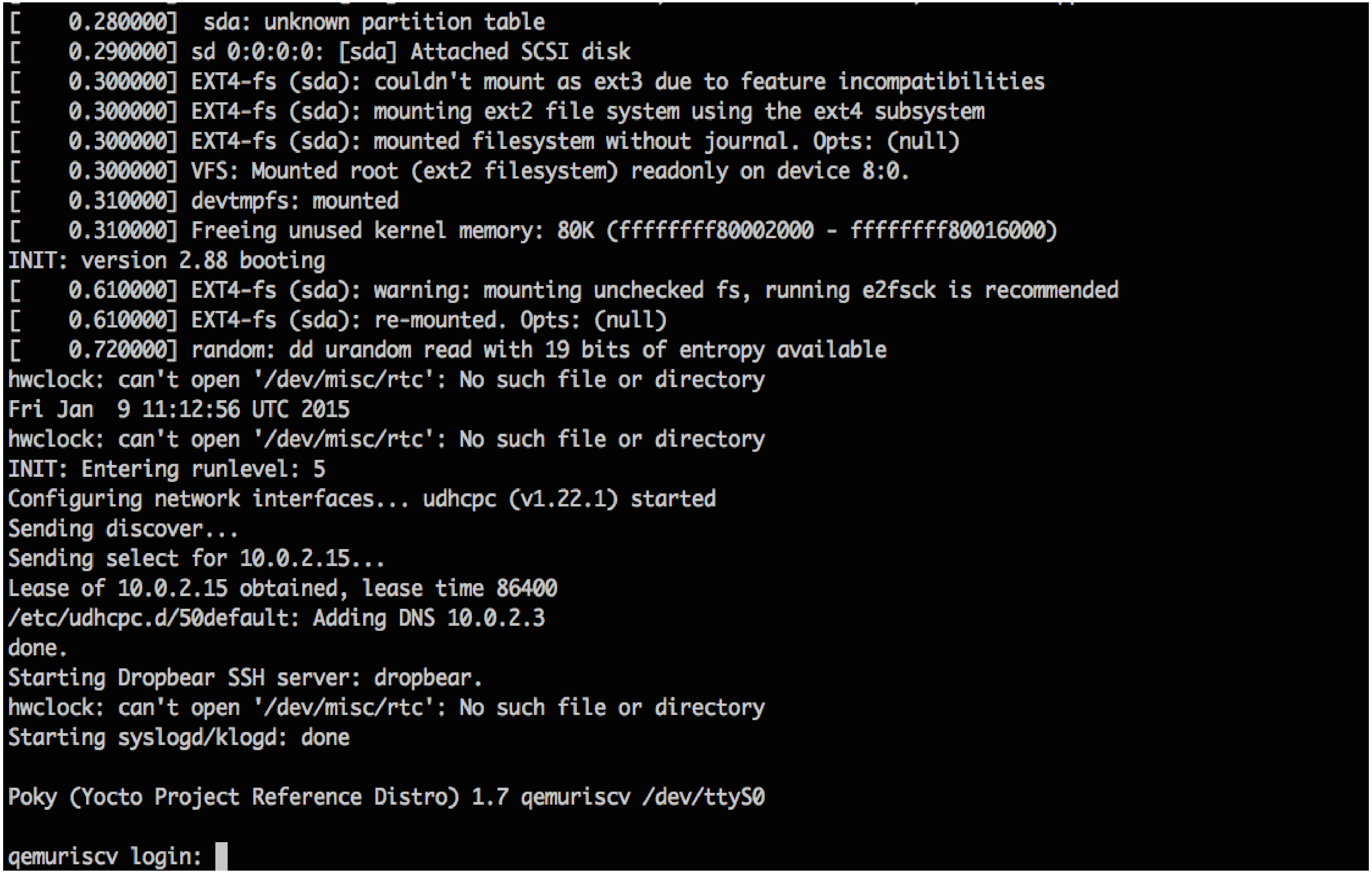
```
bitbake core-image-riscv
```

- **Step IV**: Run in QEMU (and SSH into it):

```
runqemu qemuriscv nographic slirp
hostfwd="tcp::12347-:22”
```

(Slide courtesy of Martin Maas)

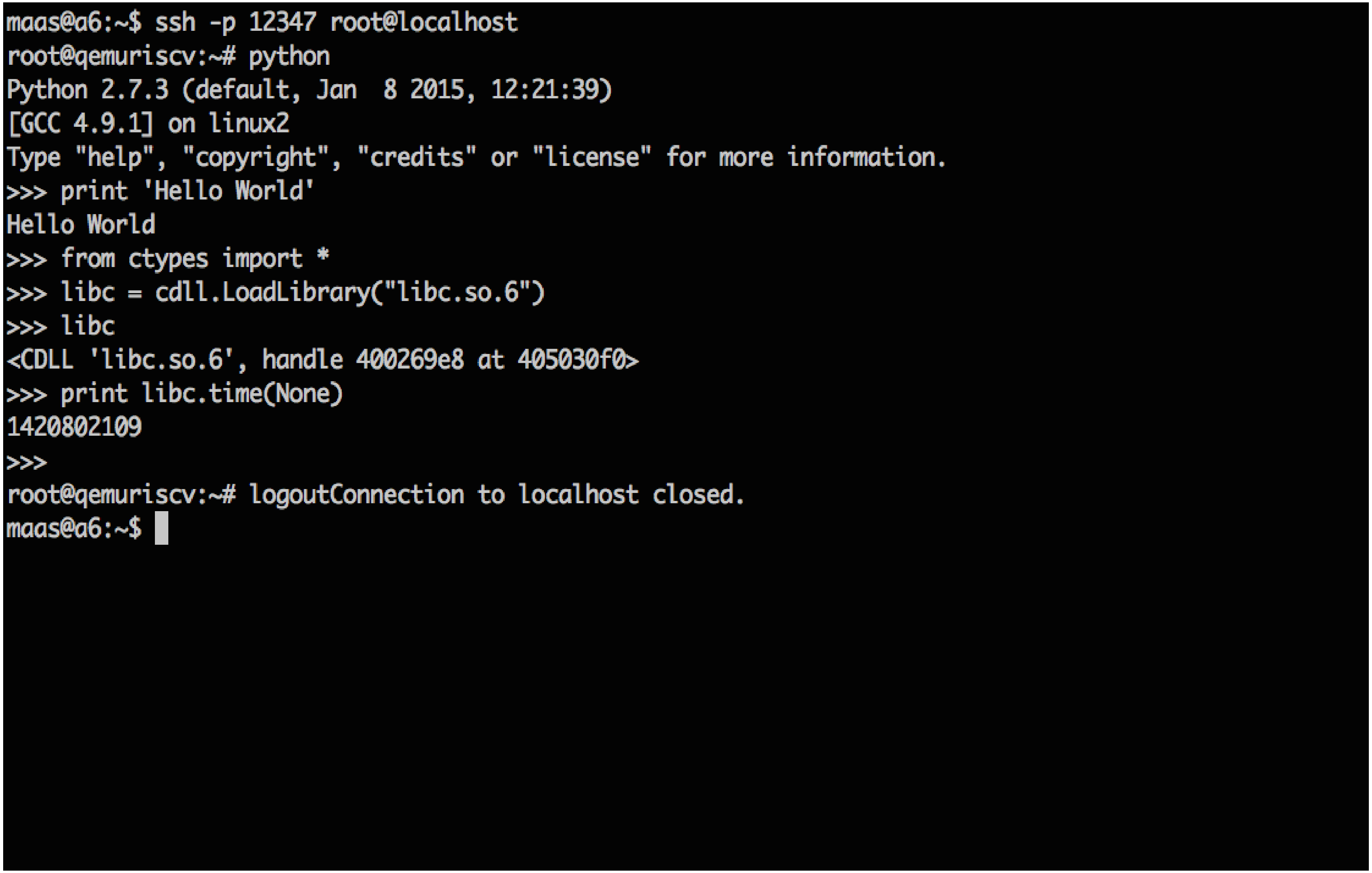
Run in QEMU (1/2)



```
[    0.280000]  sda: unknown partition table
[    0.290000] sd 0:0:0:0: [sda] Attached SCSI disk
[    0.300000] EXT4-fs (sda): couldn't mount as ext3 due to feature incompatibilities
[    0.300000] EXT4-fs (sda): mounting ext2 file system using the ext4 subsystem
[    0.300000] EXT4-fs (sda): mounted filesystem without journal. Opts: (null)
[    0.300000] VFS: Mounted root (ext2 filesystem) readonly on device 8:0.
[    0.310000] devtmpfs: mounted
[    0.310000] Freeing unused kernel memory: 80K (ffffffff80002000 - ffffffff80016000)
INIT: version 2.88 booting
[    0.610000] EXT4-fs (sda): warning: mounting unchecked fs, running e2fsck is recommended
[    0.610000] EXT4-fs (sda): re-mounted. Opts: (null)
[    0.720000] random: dd urandom read with 19 bits of entropy available
hwclock: can't open '/dev/misc/rtc': No such file or directory
Fri Jan  9 11:12:56 UTC 2015
hwclock: can't open '/dev/misc/rtc': No such file or directory
INIT: Entering runlevel: 5
Configuring network interfaces... udhcpc (v1.22.1) started
Sending discover...
Sending select for 10.0.2.15...
Lease of 10.0.2.15 obtained, lease time 86400
/etc/udhcpc.d/50default: Adding DNS 10.0.2.3
done.
Starting Dropbear SSH server: dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 1.7 qemuriscv /dev/ttyS0

qemuriscv login: 
```

(Slide courtesy of Martin Maas)

Run in QEMU (2/2)



```
maas@a6:~$ ssh -p 12347 root@localhost
root@qemuriscv:~# python
Python 2.7.3 (default, Jan  8 2015, 12:21:39)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello World'
Hello World
>>> from ctypes import *
>>> libc = cdll.LoadLibrary("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle 400269e8 at 405030f0>
>>> print libc.time(None)
1420802109
>>>
root@qemuriscv:~# logoutConnection to localhost closed.
maas@a6:~$ 
```

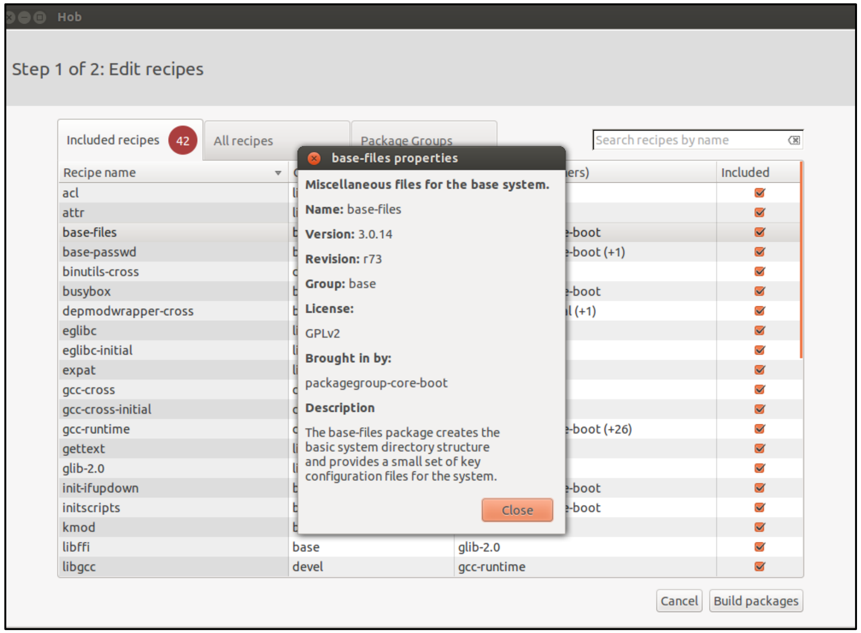
(Slide courtesy of Martin Maas)

DECIDING WHAT TO BUILD

- Decide what should go into the image:
  - Edit `meta-riscv/images/core-image-riscv.bb`
  - **Add packages** to `IMAGE_INSTALL` list, e.g. `IMAGE_INSTALL += “python python-ctypes”`
- Build packages for use with package-manager:
  - They’re already there: `build/tmp/deploy/rpm/riscv`
- Configure build by editing `conf/local.conf`
  - **Select init system**: We use SysV for now, systemd is available in Yocto
  - Switch **target machine** from `qemuriscv` and `riscv` machine to target real hardware instead of QEMU
  - Can use externally built toolchain

(Slide courtesy of Martin Maas)

SOME ADDITIONAL YOCTO FEATURES



Hob

Step 1 of 2: Edit recipes

Recipe name		ers)	Included
acl	li		☑
attr	li		☑
base-files	b	-boot	☑
base-passwd	b	-boot (+1)	☑
binutils-cross	c		☑
busybox	b	-boot	☑
depmodwrapper-cross	b	l (+1)	☑
eglibc	li		☑
eglibc-initial	li		☑
expat	li		☑
gcc-cross	c		☑
gcc-cross-initial	c		☑
gcc-runtime	c	-boot (+26)	☑
gettext	li		☑
glib-2.0	li		☑
init-ifupdown	b	-boot	☑
initscripts	b	-boot	☑
kmod	b		☑
libffi	base	glib-2.0	☑
libgcc	devel	gcc-runtime	☑

base-files properties

Miscellaneous files for the base system.

Name: base-files

Version: 3.0.14

Revision: r73

Group: base

License:

GPLv2

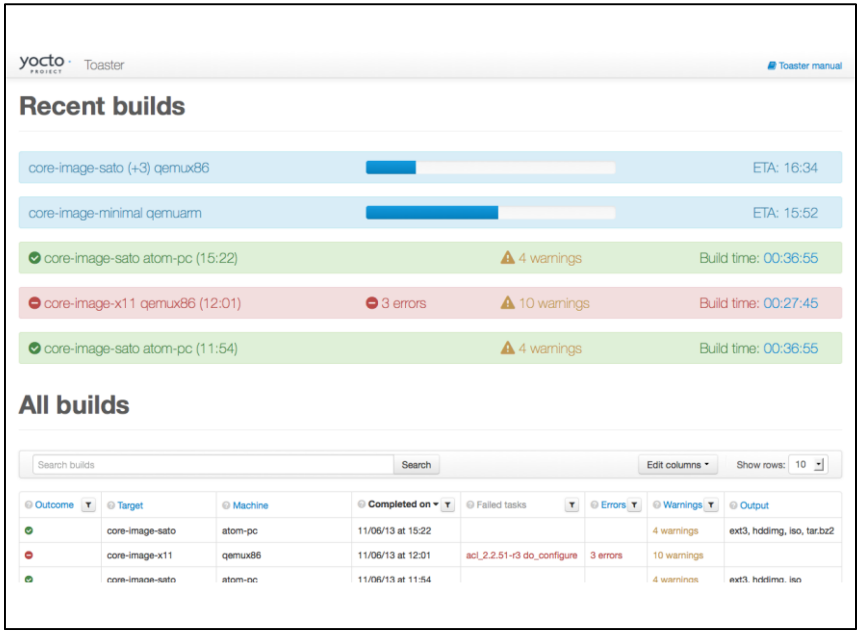
Brought in by:

packagegroup-core-boot

Description

The base-files package creates the basic system directory structure and provides a small set of key configuration files for the system.

Hob: GUI to control Bitbake



Recent builds

core-image-sato (+3) qemux86 ETA: 16:34

core-image-minimal qemuarm ETA: 15:52

✓ core-image-sato atom-pc (15:22) ⚠ 4 warnings Build time: 00:36:55

⊖ core-image-x11 qemux86 (12:01) ⊖ 3 errors ⚠ 10 warnings Build time: 00:27:45

✓ core-image-sato atom-pc (11:54) ⚠ 4 warnings Build time: 00:36:55

All builds

Outcome	Target	Machine	Completed on	Failed tasks	Errors	Warnings	Output
✓	core-image-sato	atom-pc	11/06/13 at 15:22			4 warnings	ext3, hddimg, iso, tar.bz2
⊖	core-image-x11	qemux86	11/06/13 at 12:01	acl_2.2.51-r3 do_configure	3 errors	10 warnings	
✓	core-image-sato	atom-pc	11/06/13 at 11:54			4 warnings	ext3, hddimg, iso

Toaster: Build Server

Yocto provides a lot of industry-strength features:
QA, checking license files, central build repositories, etc.

(Slide courtesy of Martin Maas)

RISC-V

Questions?

Thank you for attending!