🤨

Makefile

Comprehensive Guide to Makefiles for Hardware Projects

Introduction to Makefiles

A Makefile is a build automation tool that defines how to compile and link a program. It consists of rules that specify target files, their dependencies, and the commands to create them. Make only rebuilds files when their dependencies have changed, making it efficient for incremental builds.

Basic Makefile Structure

Syntax

```
target: dependencies
    command
```

Key points:

- Commands must be indented with a **tab** character (not spaces)
- Variables are defined with `VARIABLE = value`
- Variables are referenced with `$(VARIABLE)` or `${VARIABLE}`
- Comments start with `#`

Core Components

Variables: Store reusable values

```
CC = gcc
CFLAGS = -Wall -O2
```

Rules: Define how to build targets

```
program: main.o utils.o
   $(CC) -o program main.o utils.o
```

Pattern Rules: Handle multiple similar files

```
%.o: %.c
   $(CC) $(CFLAGS) -c $< -o $@
```

Special Variables:

- `$@` - Target name
- `$<` - First dependency
- `$^` - All dependencies
- `$?` - Dependencies newer than target

Analysis of PicoRV32 Makefile

The PicoRV32 project demonstrates advanced Makefile techniques for hardware development:

Configuration Management

The Makefile uses well-organized variable definitions for tool configuration:

```
RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX = /opt/riscv32
SHELL = bash
PYTHON = python3
VERILATOR = verilator
TOOLCHAIN_PREFIX = $(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)i/bin/riscv32-unknown-elf-
```

This approach allows easy customization of tools and paths without modifying the entire Makefile.

Flexible Testing Framework

The project implements multiple test targets for different scenarios:

- `test`: Basic functionality testing
- `test_vcd`: Testing with VCD trace generation
- `test_wb`: Wishbone interface testing
- `test_synth`: Synthesis testing
- `test_verilator`: Verilator-based testing

Each test variant follows a consistent pattern, making the build system maintainable and extensible.

Advanced Pattern Usage

The Makefile leverages sophisticated Make features:

Dynamic Object File Generation:

```
TEST_OBJS = $(addsuffix .o,$(basename $(wildcard tests/*.S)))
FIRMWARE_OBJS = firmware/start.o firmware/irq.o firmware/print.o ...
```

Conditional Compilation Flags:

```
$(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA))
```

Template-based Rule Generation:
The project uses `define` and `eval` to create multiple similar build targets for different RISC-V configurations.

Dependency Management

The Makefile properly handles complex dependencies between Verilog files, firmware, and test objects, ensuring correct build order and incremental compilation.

Detailed Code Analysis of PicoRV32 Makefile

Variable Definitions and Configuration

The Makefile begins with critical configuration variables:

```
RISCV_GNU_TOOLCHAIN_GIT_REVISION = 411d134
RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX = /opt/riscv32
```

These define the specific toolchain version and installation path, ensuring reproducible builds across different environments.

Tool Configuration Variables:

```
SHELL = bash
PYTHON = python3
VERILATOR = verilator
ICARUS_SUFFIX =
IVERILOG = iverilog$(ICARUS_SUFFIX)
VVP = vvp$(ICARUS_SUFFIX)
```

This approach allows easy tool substitution. For example, if you need a specific version of Icarus Verilog, you can set `ICARUS_SUFFIX = -0.10` to use `iverilog-0.10`.

Dynamic Object File Generation:

```
TEST_OBJS = $(addsuffix .o,$(basename $(wildcard tests/*.S)))
FIRMWARE_OBJS = firmware/start.o firmware/irq.o firmware/print.o firmware/hello.o firmware/sieve.o firmware/multest.o firmware/stats.o
```

The `TEST_OBJS` line automatically discovers all assembly files in the `tests/` directory and converts them to object file names. This means adding a new test file doesn't require Makefile modification.

Toolchain Prefix Construction:

```
TOOLCHAIN_PREFIX = $(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)i/bin/riscv32-unknown-elf-
```

This creates the full path to cross-compilation tools, so `$(TOOLCHAIN_PREFIX)gcc` becomes `/opt/riscv32i/bin/riscv32-unknown-elf-gcc`.

Testing Targets and Workflow

The Makefile implements a comprehensive testing strategy with multiple test variants:

Basic Test Target:

```
test: testbench.vvp firmware/firmware.hex
    $(VVP) -N $<
```

This runs the compiled testbench with the firmware. The dependency ensures both the testbench and firmware are built before testing.

Test with VCD Generation:

```
test_vcd: testbench.vvp firmware/firmware.hex
    $(VVP) -N $< +vcd +trace +noerror
```

The `+vcd +trace` flags enable waveform generation for debugging, while `+noerror` continues simulation despite warnings.

Specialized Test Variants:

- `test_wb`: Tests Wishbone bus interface
- `test_axi`: Tests AXI bus interface
- `test_synth`: Tests synthesized version
- `test_verilator`: Uses Verilator for faster simulation

Testbench Compilation Rules

Standard Verilog Compilation:

```
testbench.vvp: testbench.v picorv32.v
    $(IVERILOG) -o $@ $(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA)) $^
    chmod -x $@
```

The `$(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA))` substitution converts the `COMPRESSED_ISA = C` variable into the `-DCOMPRESSED_ISA` compiler flag. The `chmod -x` prevents accidental execution of the compiled output.

Formal Verification Build:

```
testbench_rvf.vvp: testbench.v picorv32.v rvfimon.v
    $(IVERILOG) -o $@ -D RISCV_FORMAL $(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA)) $^
    chmod -x $@
```

This variant includes the RISC-V formal verification monitor (`rvfimon.v`) and enables formal verification defines.

Firmware Build Pipeline

The firmware build follows a multi-stage process:

Stage 1: Assembly to Object

```
firmware/start.o: firmware/start.S
    $(TOOLCHAIN_PREFIX)gcc -c -mabi=ilp32 -march=rv32im$(subst C,c,$(COMPRESSED_ISA)) -o $@ $<
```

This compiles the startup assembly code with RISC-V specific flags:

- `mabi=ilp32`: 32-bit integer ABI
- `march=rv32im`: Base integer + multiplication extensions

Stage 2: C to Object

```
firmware/%.o: firmware/%.c
    $(TOOLCHAIN_PREFIX)gcc -c -mabi=ilp32 -march=rv32i$(subst C,c,$(COMPRESSED_ISA)) -Os --std=c99 $(GCC_WARNS) -ffreestanding -nostdlib -o $@ $<
```

Note the different architecture flag (`rv32i` vs `rv32im`) - C code doesn't need multiplication extensions, while the startup code might use them.

Stage 3: Linking

```
firmware/firmware.elf: $(FIRMWARE_OBJS) $(TEST_OBJS) firmware/sections.lds
    $(TOOLCHAIN_PREFIX)gcc -Os -mabi=ilp32 -march=rv32im$(subst C,c,$(COMPRESSED_ISA)) -ffreestanding -nostdlib -o $@ \
        -Wl,--build-id=none,-Bstatic,-T,firmware/sections.lds,-Map,firmware/firmware.map,--strip-debug \
        $(FIRMWARE_OBJS) $(TEST_OBJS) -lgcc
```

The linker flags are crucial:

- `T,firmware/sections.lds`: Use custom linker script
- `Map,firmware/firmware.map`: Generate memory map
- `-strip-debug`: Remove debug info for smaller size

Stage 4: Binary Conversion

```
firmware/firmware.bin: firmware/firmware.elf
    $(TOOLCHAIN_PREFIX)objcopy -O binary $< $@

firmware/firmware.hex: firmware/firmware.bin firmware/makehex.py
    $(PYTHON) firmware/makehex.py $< 32768 > $@
```

This converts the ELF file to raw binary, then to hex format suitable for Verilog memory initialization.

Advanced Template System

The Makefile uses sophisticated template generation for multiple RISC-V configurations:

```
define build_tools_template
build-$(1)-tools:
    @read -p "This will remove all existing data from $(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)$(subst riscv32,,$(1)). Type YES to continue: " reply && [[ "$$reply" == [Yy][Ee][Ss] || "$$reply" == [Yy] ]]
    ...
endef

$(eval $(call build_tools_template,riscv32i,rv32i))
$(eval $(call build_tools_template,riscv32ic,rv32ic))
$(eval $(call build_tools_template,riscv32im,rv32im))
$(eval $(call build_tools_template,riscv32imc,rv32imc))
```

This creates four different toolchain build targets (for different RISC-V instruction set combinations) from a single template, reducing code duplication.

Workflow Analysis

Development Workflow

1. **Initial Setup**: Run `make download-tools` to cache git repositories, then `make build-tools` to build toolchains
2. **Code Development**: Modify Verilog RTL or firmware C code
3. **Basic Testing**: `make test` for quick functionality check
4. **Debug Testing**: `make test_vcd` to generate waveforms for analysis
5. **Comprehensive Testing**: Run specialized tests (`test_wb`, `test_axi`, etc.)
6. **Formal Verification**: `make check` for formal property verification
7. **Synthesis Testing**: `make test_synth` to verify synthesized design

Build Dependencies Flow

```
firmware/firmware.hex ← firmware/firmware.bin ← firmware/firmware.elf
← {firmware/*.o, tests/*.o}
                                                                   ↑
testbench.vvp ← {testbench.v, picorv32.v}                    firmware/secti
ons.lds
       ↓
   test targets
```

The firmware build is completely independent of the Verilog compilation, allowing parallel development of hardware and software components.

Test Object Compilation

The test compilation process shows sophisticated pattern matching:

```
tests/%.o: tests/%.S tests/riscv_test.h tests/test_macros.h
   $(TOOLCHAIN_PREFIX)gcc -c -mabi=ilp32 -march=rv32im -o $@ -DTES
T_FUNC_NAME=$(notdir $(basename $<)) \
      -DTEST_FUNC_TXT='"$(notdir $(basename $<))"' -DTEST_FUNC_RET
=$(notdir $(basename $<))_ret $<
```

This rule:

- Compiles any `.S` file in the `tests/` directory
- Defines preprocessor macros based on the filename
- For `tests/add.S`, it defines `TEST_FUNC_NAME=add`, `TEST_FUNC_TXT="add"`, and `TEST_FUNC_RET=add_ret`
- Depends on common header files that all tests need

Formal Verification Integration

The formal verification workflow uses Yosys SMT-BMC:

```
check.smt2: picorv32.v
   yosys -v2 -p 'read_verilog -formal picorv32.v' \
          -p 'prep -top picorv32 -nordff' \
          -p 'assertpmux -noinit; opt -fast; dffunmap' \
          -p 'write_smt2 -wires check.smt2'
```

This converts the Verilog design to SMT2 format for formal verification, then:

```
check-%: check.smt2
    yosys-smtbmc -s $(subst check-,,$@) -t 30 --dump-vcd check.vcd check.smt2
    yosys-smtbmc -s $(subst check-,,$@) -t 25 --dump-vcd check.vcd -i check.smt2
```

Runs bounded model checking with different solvers (yices, z3, etc.) by substituting the solver name from the target.

Verilator Integration

The Verilator compilation demonstrates mixed-language builds:

```
testbench_verilator: testbench.v picorv32.v testbench.cc
    $(VERILATOR) --cc --exe -Wno-lint -trace --top-module picorv32_wrapper testbench.v picorv32.v testbench.cc \
            $(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA)) --Mdir testbench_verilator_dir
    $(MAKE) -C testbench_verilator_dir -f Vpicorv32_wrapper.mk
    cp testbench_verilator_dir/Vpicorv32_wrapper testbench_verilator
```

This:

- Converts Verilog to C++ classes
- Compiles with C++ testbench code
- Uses recursive Make to build the generated C++ code
- Copies the final executable to a convenient location

Toolchain Management Deep Dive

The toolchain building process shows advanced shell scripting within Make:

```
build-$(1)-tools-bh:
    +set -ex; $(GIT_ENV); \
    if [ -d /var/cache/distfiles/riscv-gnu-toolchain.git ]; then reference_riscv_gnu_toolchain="--reference /var/cache/distfiles/riscv-gnu-toolchain.git"; else reference_riscv_gnu_toolchain=""; fi; \
    ...
    rm -rf riscv-gnu-toolchain-$(1); git clone $$reference_riscv_gnu_toolchain https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain-$(1); \
    cd riscv-gnu-toolchain-$(1); git checkout $(RISCV_GNU_TOOLCHAIN_GIT_REVISION); \
    mkdir build; cd build; ../configure --with-arch=$(2) --prefix=$(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)$(subst riscv32,,$(1)); make
```

Key features:

- Uses conditional logic to leverage cached repositories if available
- Checks out a specific git revision for reproducibility
- Configures for specific RISC-V architecture variants
- Uses `set -ex` for immediate error exit and command tracing
- The `+` prefix allows this rule to run even with `n` (dry-run) flag

Key Workflow Features

Incremental Builds: Make only rebuilds changed components and their dependents

Parallel Development: Hardware and firmware teams can work independently since the dependency graphs are separate until the final testing phase

Multiple Test Configurations: Different test scenarios (basic, VCD, wishbone, AXI, synthesis, Verilator) without requiring separate build systems

Tool Flexibility: Easy switching between different versions of simulation tools through variable configuration

Automated Toolchain Management: Ensures consistent development environment across team members by building tools from source with fixed revisions

Error Prevention: The `chmod -x` commands prevent accidental execution of compiled Verilog files, and interactive prompts prevent accidental deletion of toolchains

Memory and Resource Management

The clean target shows comprehensive cleanup:

```
clean:
    rm -rf riscv-gnu-toolchain-riscv32i riscv-gnu-toolchain-riscv32ic \
        riscv-gnu-toolchain-riscv32im riscv-gnu-toolchain-riscv32imc
    rm -vrf $(FIRMWARE_OBJS) $(TEST_OBJS) check.smt2 check.vcd synth.v synth.log \
        firmware/firmware.elf firmware/firmware.bin firmware/firmware.hex firmware/firmware.map \
        testbench.vvp testbench_sp.vvp testbench_synth.vvp testbench_ez.vvp \
        testbench_rvf.vvp testbench_wb.vvp testbench.vcd testbench.trace \
        testbench_verilator testbench_verilator_dir
```

This removes:

- Temporary toolchain build directories
- All compiled objects and executables
- Generated files (hex, maps, VCD traces)
- Intermediate build artifacts
- Tool-specific output directories

The use of `-v` flag provides verbose output showing what's being deleted, which is helpful for debugging cleanup issues.

Advanced Makefile Techniques Demonstrated

1. Template-Based Code Generation

The `define` / `eval` / `call` pattern eliminates code duplication for similar targets while maintaining readability.

2. Conditional Compilation

Using `$(subst)` to convert configuration variables into compiler flags allows the same source to be compiled for different target configurations.

3. Multi-Stage Dependencies

The firmware build pipeline demonstrates how to chain multiple transformation steps while maintaining proper dependency tracking.

4. Tool Integration

Shows how to integrate multiple disparate tools (cross-compilers, HDL simulators, formal verification tools) into a unified build system.

5. Interactive Safety Checks

The confirmation prompts prevent accidental destructive operations while still allowing automation when needed.

6. Recursive Make Usage

Properly handles cases where external build systems (like Verilator's generated Makefiles) need to be invoked as part of the overall build process.

Conclusion

A well-designed Makefile is crucial for hardware development projects. The PicoRV32 example demonstrates many advanced techniques that can be adapted for various hardware development workflows. The key is to start with a solid foundation and gradually add advanced features while maintaining simplicity and reliability.

The sophisticated dependency management, multi-tool integration, and workflow automation shown in this Makefile provide a template for building robust, maintainable build systems for complex hardware projects.make test_vcd `to generate waveforms for analysis 5. **Comprehensive Testing**: Run specialized tests (`test_wb`, `test_axi`, etc.) 6. **Formal Verification**: `make check`for formal property verification 7. **Synthesis Testing**: `make test_synth` to verify synthesized design

Build Dependencies Flow

```
firmware/firmware.hex ← firmware/firmware.bin ← firmware/firmware.elf
← {firmware/*.o, tests/*.o}
                                                   ↑
testbench.vvp ← {testbench.v, picorv32.v}                firmware/secti
ons.lds
     ↓
  test targets
```

The firmware build is completely independent of the Verilog compilation, allowing parallel development of hardware and software components.

Key Workflow Features

Incremental Builds: Make only rebuilds changed components and their dependents

Parallel Development: Hardware and firmware teams can work independently

Multiple Test Configurations: Different test scenarios without separate build systems

Tool Flexibility: Easy switching between different versions of simulation tools

Automated Toolchain Management: Consistent development environment across team members

Detailed Code Analysis of PicoRV32 Makefile

Variable Definitions and Configuration

The Makefile begins with critical configuration variables:

```
RISCV_GNU_TOOLCHAIN_GIT_REVISION = 411d134
RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX = /opt/riscv32
```

These define the specific toolchain version and installation path, ensuring reproducible builds across different environments.

Tool Configuration Variables:

```
SHELL = bash
PYTHON = python3
VERILATOR = verilator
ICARUS_SUFFIX =
IVERILOG = iverilog$(ICARUS_SUFFIX)
VVP = vvp$(ICARUS_SUFFIX)
```

This approach allows easy tool substitution. For example, if you need a specific version of Icarus Verilog, you can set `ICARUS_SUFFIX = -0.10` to use `iverilog-0.10`.

Dynamic Object File Generation:

```
TEST_OBJS = $(addsuffix .o,$(basename $(wildcard tests/*.S)))
FIRMWARE_OBJS = firmware/start.o firmware/irq.o firmware/print.o firmware/hello.o firmware/sieve.o firmware/multest.o firmware/stats.o
```

The `TEST_OBJS` line automatically discovers all assembly files in the `tests/` directory and converts them to object file names. This means adding a new test file doesn't require Makefile modification.

Toolchain Prefix Construction:

```
TOOLCHAIN_PREFIX = $(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)i/bin/riscv32-unknown-elf-
```

This creates the full path to cross-compilation tools, so `$(TOOLCHAIN_PREFIX)gcc` becomes `/opt/riscv32i/bin/riscv32-unknown-elf-gcc`.

Testing Targets and Workflow

The Makefile implements a comprehensive testing strategy with multiple test variants:

Basic Test Target:

```
test: testbench.vvp firmware/firmware.hex
    $(VVP) -N $<
```

This runs the compiled testbench with the firmware. The dependency ensures both the testbench and firmware are built before testing.

Test with VCD Generation:

```
test_vcd: testbench.vvp firmware/firmware.hex
    $(VVP) -N $< +vcd +trace +noerror
```

The `+vcd +trace` flags enable waveform generation for debugging, while `+noerror` continues simulation despite warnings.

Specialized Test Variants:

- `test_wb`: Tests Wishbone bus interface
- `test_axi`: Tests AXI bus interface
- `test_synth`: Tests synthesized version
- `test_verilator`: Uses Verilator for faster simulation

Testbench Compilation Rules

Standard Verilog Compilation:

```
testbench.vvp: testbench.v picorv32.v
    $(IVERILOG) -o $@ $(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA)) $^
    chmod -x $@
```

The `$(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA))` substitution converts the `COMPRESSED_ISA = C` variable into the `-DCOMPRESSED_ISA` compiler flag. The `chmod -x` prevents accidental execution of the compiled output.

Formal Verification Build:

```
testbench_rvf.vvp: testbench.v picorv32.v rvfimon.v
    $(IVERILOG) -o $@ -D RISCV_FORMAL $(subst C,-DCOMPRESSED_ISA,$(COMPRESSED_ISA)) $^
    chmod -x $@
```

This variant includes the RISC-V formal verification monitor (`rvfimon.v`) and enables formal verification defines.

Firmware Build Pipeline

The firmware build follows a multi-stage process:

Stage 1: Assembly to Object

```
firmware/start.o: firmware/start.S
    $(TOOLCHAIN_PREFIX)gcc -c -mabi=ilp32 -march=rv32im$(subst C,c,$(COMPRESSED_ISA)) -o $@ $<
```

This compiles the startup assembly code with RISC-V specific flags:

- `mabi=ilp32`: 32-bit integer ABI
- `march=rv32im`: Base integer + multiplication extensions

Stage 2: C to Object

```
firmware/%.o: firmware/%.c
    $(TOOLCHAIN_PREFIX)gcc -c -mabi=ilp32 -march=rv32i$(subst C,c,$(C
OMPRESSED_ISA)) -Os --std=c99 $(GCC_WARNS) -ffreestanding -nostdlib -o $@ $<
```

Note the different architecture flag (`rv32i` vs `rv32im`) - C code doesn't need multiplication extensions, while the startup code might use them.

Stage 3: Linking

```
firmware/firmware.elf: $(FIRMWARE_OBJS) $(TEST_OBJS) firmware/sections.lds
    $(TOOLCHAIN_PREFIX)gcc -Os -mabi=ilp32 -march=rv32im$(subst C,c,$(COMPRESSED_ISA)) -ffreestanding -nostdlib -o $@ \
        -Wl,--build-id=none,-Bstatic,-T,firmware/sections.lds,-Map,firmware/firmware.map,--strip-debug \
        $(FIRMWARE_OBJS) $(TEST_OBJS) -lgcc
```

The linker flags are crucial:

- `T,firmware/sections.lds`: Use custom linker script
- `Map,firmware/firmware.map`: Generate memory map
- `-strip-debug`: Remove debug info for smaller size

Stage 4: Binary Conversion

```
firmware/firmware.bin: firmware/firmware.elf
    $(TOOLCHAIN_PREFIX)objcopy -O binary $< $@

firmware/firmware.hex: firmware/firmware.bin firmware/makehex.py
    $(PYTHON) firmware/makehex.py $< 32768 > $@
```

This converts the ELF file to raw binary, then to hex format suitable for Verilog memory initialization.

Advanced Template System

The Makefile uses sophisticated template generation for multiple RISC-V configurations:

```
define build_tools_template
build-$(1)-tools:
    @read -p "This will remove all existing data from $(RISCV_GNU_TOOLCHAIN_INSTALL_PREFIX)$(subst riscv32,,$(1)). Type YES to continue: " reply && [[ "$$reply" == [Yy][Ee][Ss] || "$$reply" == [Yy] ]]
    ...
endef

$(eval $(call build_tools_template,riscv32i,rv32i))
$(eval $(call build_tools_template,riscv32ic,rv32ic))
$(eval $(call build_tools_template,riscv32im,rv32im))
$(eval $(call build_tools_template,riscv32imc,rv32imc))
```

This creates four different toolchain build targets (for different RISC-V instruction set combinations) from a single template, reducing code duplication.

Workflow Analysis

Development Workflow

1. **Initial Setup**: Run `make download-tools` to cache git repositories, then `make build-tools` to build toolchains
2. **Code Development**: Modify Verilog RTL or firmware C code
3. **Basic Testing**: `make test` for quick functionality check
4. **Debug Testing**: `