Digital Design and Verification Training

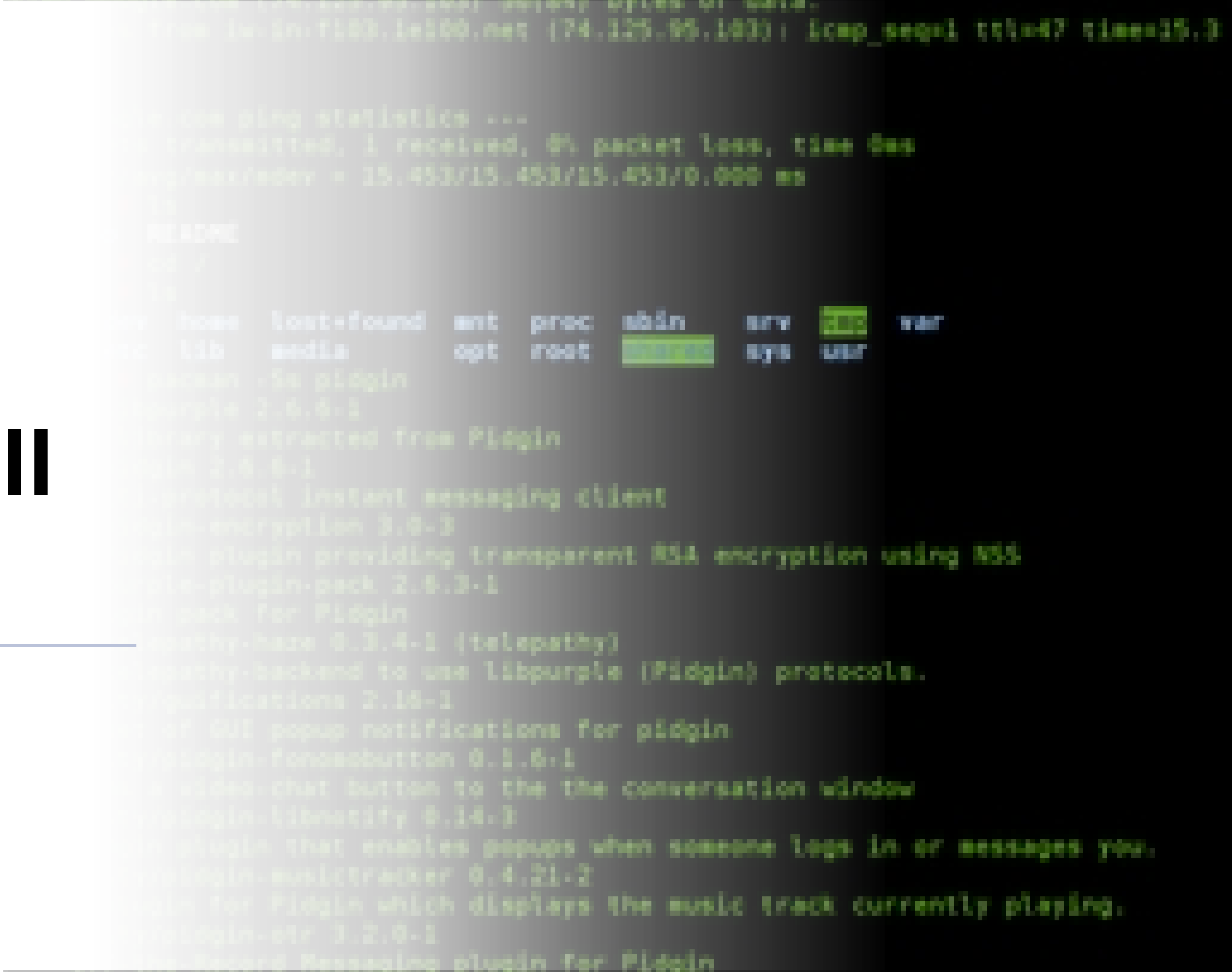
Linux Shell Scripting and Make

Agenda

- Linux Shell Scripting
  - Shell Script Structure
  - Variable in Shell scripts
  - Conditional Statements
  - Inputs and Outputs
- MakeFile

Linux Shell Scripting

Digital Design and Verification Training



Shell Scripting - basics

Creating a shell script file

- Use a text editor (e.g., nano, vim, gedit)

```
nano myscript.sh
```

- Content structure:
    - Start with shebang
    - Add comments
    - Write commands

Shell Scripting - basics

Shebang (#!) and its importance

- Definition: The shebang is the combination of '#' and '!' at the beginning of the script
- Purpose: Tells the system which interpreter to use for executing the script
- Syntax: #!/path/to/interpreter
- Common shebangs:
    - Bash: #!/bin/bash
    - Sh: #!/bin/sh
    - Python: #!/usr/bin/env python3

Shell Scripting - basics

Creating a shell script file

- Use a text editor (e.g., nano, vim, gedit)

```
#!/bin/bash
echo "Hello, World!"
```

- Making scripts executable (chmod)
  - Use the chmod command to change file permissions
  - Syntax: chmod [options] mode file

Shell Scripting - basics

Running scripts

- Methods to run a shell script: a. Using the interpreter explicitly: `$ bash myscript.sh`
- Running as an executable (if chmod +x was used): `$ ./myscript.sh`
- Sourcing the script (runs in current shell environment): `$ source myscript.sh` OR `$ . myscript.sh`
- Providing arguments: `$ ./myscript.sh arg1 arg2`
- Running in background: `$ ./myscript.sh &`

Variables in Shell Scripts

Declaring and using variables

- Syntax: VARIABLE_NAME=value
    - No spaces around the equals sign
- Naming conventions:
    - Use uppercase for constants
    - Use lowercase for other variables
    - Start with a letter or underscore
    - Can contain letters, numbers, underscores
- Accessing variables:
    - Use $ before the variable name

```bash
#!/bin/bash
NAME="John Doe"
age=30

echo "Name: ${NAME}, Age: $age"
```

Variables in Shell Scripts

Variable types

- Strings (default type)
- Integers
- Arrays

Environment variables

Pre-defined variables set by the system

- Common environment variables:
    - HOME: User's home directory
    - PATH: Directories searched for commands
    - USER: Current user's username

```
#!/bin/bash
GREETING="Hello, World!"
COUNT=5
FRUITS=("apple" "banana" "orange")
echo "Home directory: $HOME"
echo "Current user: $USER"


export MY_VAR="custom value"
```

Variables in Shell Scripts

- By default, variables are global within the script. Use local keyword inside functions for local variables
- Readonly variables
  - Declare constants using the readonly keyword
- Unsetting variables
  - Remove a variable using the unset command

```
function example_function() {
   local local_var="I'm local"
   echo $local_var
}

readonly PI=3.14159

unset VARIABLE_NAME
```

Commands and Arithmetic Operators

```
CURRENT_DATE=$(date +%Y-%m-%d)
echo "Today's date is $CURRENT_DATE"

FILES_COUNT=$(ls | wc -l)
echo "Number of files in current directory: $FILES_COUNT"
```

```
X=5
Y=3

SUM=$((X + Y))
echo "Sum: $SUM"
```

Inputs and Outputs

- Command-line arguments ($1, $2, ...)
- Accessing arguments:
  - $1: First argument
  - $2: Second argument
  - $9: Ninth argument
  - ${10}: Tenth argument (and beyond, use braces)
- Special variables:
  - $0: Name of the script itself
  - $#: Number of arguments

```
#!/bin/bash

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Number of arguments: $#"

echo "All arguments: $@"
```

Inputs and Outputs

- Reading user input (read command)
- Basic syntax: read VARIABLE_NAME
- Options:
  - -p: Specify a prompt
  - -s: Silent mode (for passwords)
  - -t: Specify a timeout

```
#!/bin/bash

read -p "Enter your name: " NAME
echo "Hello, $NAME!"

read -sp "Enter password: " PASSWORD
echo

read -t 5 -p "Quick! Enter a number: " NUMBER
echo "You entered: $NUMBER"
```

Inputs and Outputs

- Redirecting input/output (>, >>, <, <<)
- Output redirection:
  - > Redirect and overwrite
  - >> Redirect and append
- Input redirection:
  - < Redirect input from a file
- Pipes (|)
  - Connect the output of one command to the input of another
  - Allows creation of command pipelines

```
#!/bin/bash

# Output redirection
echo "Hello, World!" > output.txt
echo "Appended line" >> output.txt

# Input redirection
sort < unsorted_list.txt > sorted_list.txt

ls -l | grep ".txt"
cat file.txt | sort | uniq > sorted_unique.txt
```

Conditional Statements

- if, elif, else structures
- Comparison operators
- Numeric comparisons:
  - -eq: Equal to, -ne: Not equal to
  - -lt: Less than, -le: Less than or equal to
  - -gt: Greater than, -ge: Greater than or equal to
- String comparisons:
  - =: Equal to, !=: Not equal to
  - <: Less than (ASCII alphabetical order), : Greater than
  - -z: String is empty, -n: String is not empty

```
#!/bin/bash
if [ condition ]; then
   # commands
elif [ condition ]; then
   # commands
else
   # commands
fi


age=25
if [ $age -lt 18 ]; then
   echo "Minor"
elif [ $age -ge 18 ] && [ $age -lt 65 ]; then
   echo "Adult"
else
   echo "Senior"
fi
```

Loops

```
while [ condition ]
do # commands done

count=1
while [ $count -le 5 ]
do
echo "Count: $count"
((count++))
done

count=5
until [ $count -eq 0 ]
do
echo "Countdown: $count"
((count--))
done
```

```
for variable in list
do
# commands
done

# example
for fruit in apple banana orange
do
echo "I like $fruit"
done

for ((i=0; i<5; i++))
do echo "Index: $i" done

for line in $(cat file.txt)
do
echo "Line: $line"
done
```

Defining and calling functions

```
function_name() { # function body }

function function_name { # function body }

# Calling a function:
# function_name



greet() {
echo "Hello, world!" }

greet # Calling the function
```

```
greet_person() { echo "Hello, $1!" }

greet_person "Alice" # Outputs: Hello, Alice!

# Example 2
is_even()
{ if (( $1 % 2 == 0 ));
then return 0 # Success (even number)
else return 1 # Failure (odd number)
fi }

is_even 4
if [ $? -eq 0 ];
then echo "4 is even"
fi
```

Defining and calling functions - Examples

```
get_square()
{
echo $(($1 * $1))
}

result=$(get_square 5)
echo "The square of 5 is $result"

global_var="I'm global"
test_scope() {
   local local_var="I'm local"
   echo "Inside function: global_var = $global_var,
local_var = $local_var"
}

test_scope
echo "Outside function: global_var = $global_var"
echo "Outside function: local_var = $local_var" # This
```

```
# Recursive Function

factorial() {
 if [ $1 -le 1 ];
then echo 1
else
  local prev=$(factorial $(($1 - 1)))
  echo $(($1 * $prev))
fi
}
```

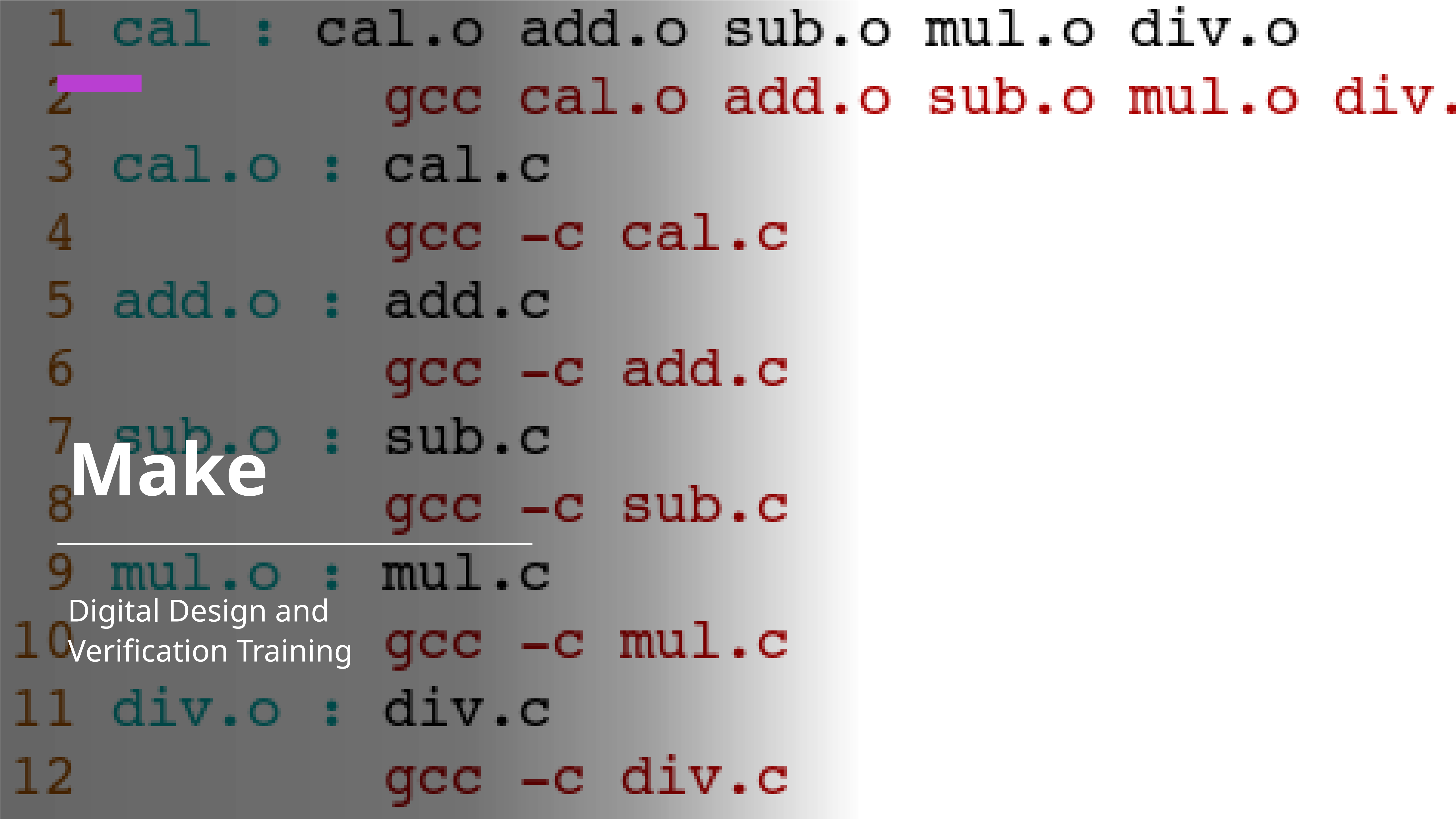
Arithmetic Operators - Examples

```
result=$(expr 5 + 3)
echo $result # Output: 8

result=$(expr 10 \* 2) # Note the escaped *
echo $result # Output: 20

result=$(expr 20 / 3)
echo $result # Output: 6 (integer division)

result=$(expr 20 % 3)
echo $result # Output: 2 (remainder)

result=$((5 & 3)) # Bitwise AND
echo $result # Output: 1

result=$((5 | 3)) # Bitwise OR
echo $result # Output: 7
```

```
let "a = 5 + 3"
echo $a # Output: 8

let "b = 10 * 2"
echo $b # Output: 20

result=$((5 + 3))
echo $result # Output: 8

result=$((10 * 2))
echo $result # Output: 20

a=10
((a += 5))
echo $a # Output: 15
```

Make

Digital Design and Verification Training



```
cal : cal.o add.o sub.o mul.o div.o
        gcc cal.o add.o sub.o mul.o div.
cal.o : cal.c
        gcc -c cal.c
add.o : add.c
        gcc -c add.c
sub.o : sub.c
        gcc -c sub.c
mul.o : mul.c
        gcc -c mul.c
div.o : div.c
        gcc -c div.c
```

Introduction

- Makefile to tell `make` what to do.
- Most often, the makefile tells `make` how to compile and link a program.

```
target : prerequisites
    recipe
    ...
```

- Target is the name of file that will be generated by `make`.
- Prerequisites are the files that are necessary for the target to be generated.
  - Separated by space
  - Also called dependencies
- Recipe is one or many commands needed to generate the target.
  - Each command on a separate line
  - Indented by "Tab"

Example

- File name: makefile or Makefile

```
main.o : main.c
	echo "Compiling the main file"
	gcc -o main.o main.c
```

- Command: `make` or `make main.o`
- Output displayed on Terminal:

```
echo "Compiling the main file"
Compiling the main file
gcc -o main.o main.c
```

Command Silencing

```
main.o : main.c
    @echo "Compiling the main file"
    gcc -o main.o main.c
```

Output displayed on Terminal:

```
Compiling the main file
gcc -o main.o main.c
```

- Add an @ before a command to stop it from being printed
- You can also run make with -s to add an @ before each line

Simple Makefile Example

```makefile
CC = gcc
CFLAGS = -Wall

all: hello

hello: hello.o
	$(CC) -o hello hello.o

hello.o: hello.c
	$(CC) $(CFLAGS) -c hello.c

clean:
	rm -f hello hello.o
```

Phony Targets

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.

- all
- clean

Wildcards

- * and % are called wildcards in Make
- Example: A rule to delete all the object files

```
clean:
    rm -f *.o
```

- % is really useful, but is somewhat confusing because of the variety of situations it can be used in.
  - When used in "matching" mode, it matches one or more characters in a string. This match is called the stem.
  - When used in "replacing" mode, it takes the stem that was matched and replaces that in a string.

Wildcard Function

- Wildcard expansion happens automatically in rules.
- But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function.
- If you want to do wildcard expansion in such places, you need to use the `wildcard` function

```
objects = *.o
```

  - The value of the variable `objects` is the actual string ‘`*.o`’.
  - However, if you use the value of `objects` in a target or prerequisite, wildcard expansion will take place there.
- To set `objects` to the expansion, instead use:

```
objects := $(wildcard *.o)
```

Variables

- Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files, directories to write output in, or anything else you can imagine.
- A variable name may be any sequence of characters not containing ‘:’, ‘#’, ‘=’, or whitespace. However, variable names containing characters other than letters, numbers, and underscores should be avoided as in shell scripting.
- A variable and the value(s) it holds are separated by an equals (=) sign.
- Multiple values are separated by spaces between each other.
- To use our variable, we can enclose it in parentheses beginning with a dollar sign.

Variables Examples

- All the flags for the C compiler.
  - `CFLAGS = -Wall`
- The program for compiling C files.
  - `CC = gcc`
- VPATH is the directories search list for both prerequisites and targets. Paths are separated by the colon sign :
  - `VPATH = src : build`
- `objects = program.o foo.o utils.o`

Automatic Variables

- Automatic variables are special variables that Make sets automatically for each rule.
- The most commonly used automatic variables are:
- $@: The target filename
  - Represents the file name of the target of the rule
  - Useful when the same command is used for multiple targets

```
foo.o bar.o: %.o: %.c
    gcc -c $< -o $@
```

Automatic Variables

- $<: The first dependency filename
  - Represents the name of the first prerequisite (dependency)
  - Particularly useful in pattern rules

```
%.o: %.c
    gcc -c $< -o $@
# Here, $< represents the .c file that corresponds to the .o file
being built
```

- $^: The filenames of all dependencies
  - Represents all prerequisites of the rule, separated by spaces
  - Useful when you need to use all dependencies in the command

```
myprogram: foo.o bar.o baz.o
    gcc $^ -o $@
# In this case, $^ expands to 'foo.o bar.o baz.o'
```

Automatic Variables

- $?: The names of all the prerequisites that are newer than the target
  - when you wish to operate on only the prerequisites that have changed
  - Particularly useful in explicit rules

```
final: main.o hello.o check.o
    gcc -Wall $? -o final

# Here, $? represents the .o file that has been built after previous compilation of final executable
```

Conditional Directives

- ifeq, ifneq, ifdef, ifndef
- Example:

```
ifeq ($(DEBUG),yes)
CFLAGS += -g
endif
```

- Parallel Execution
    - Using -j option for parallel builds
    - Example: make -j4

Debugging Makefiles

- Using -n option to print commands without executing
- Using -d option for detailed debug information
- Example: make -n target