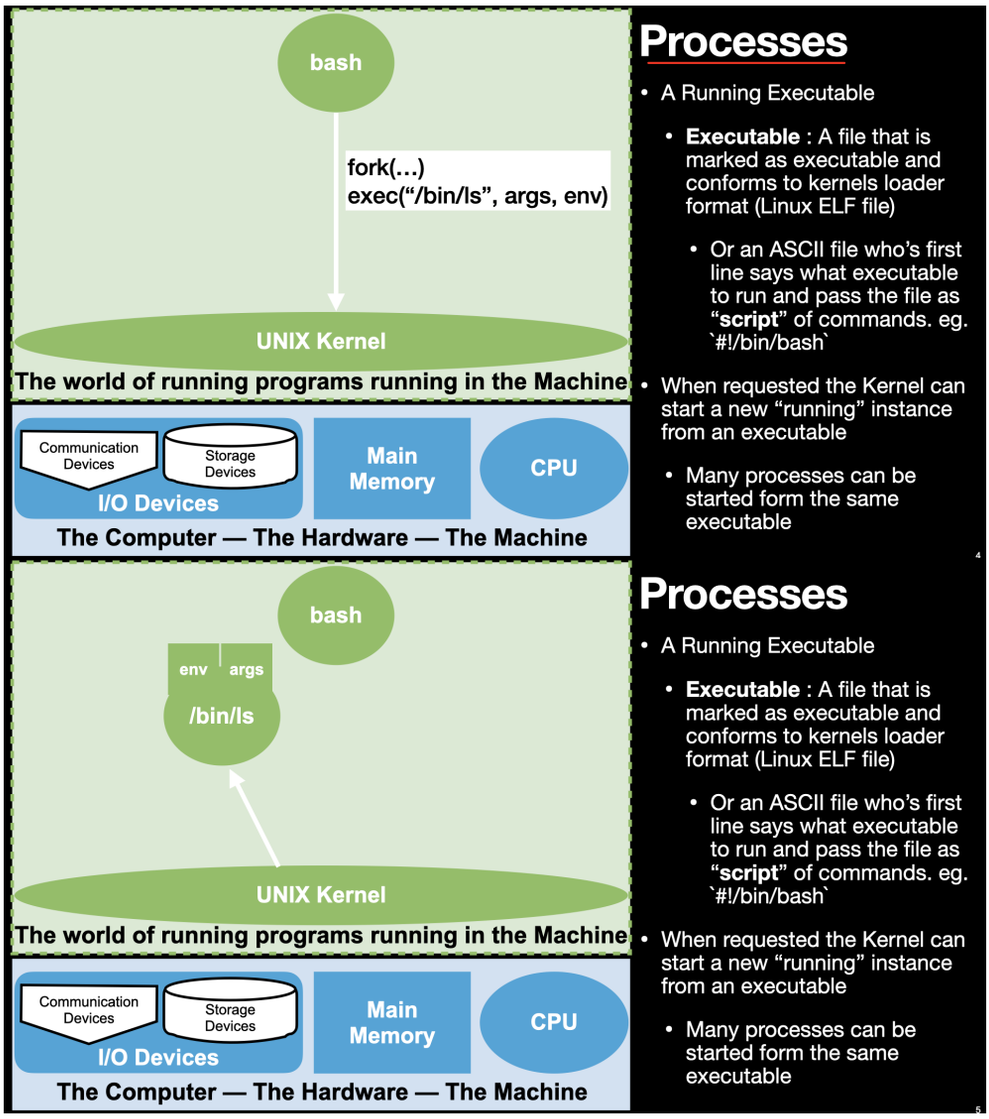
SLS Lecture 7 : Assembly Programming Introduction

Contents

Processes and Binaries

7.1. Revisit Processes



7.1.1. A Process is a Running executable but what really is an executable???

Let’s see what we can figure out poking at the file a little

```
$ ls -l /bin/ls
-rwxr-xr-x. 1 root root 142144 Sep  5  2019 /bin/ls
$
```

- So it is marked as a executable “x” in the permissions bits
- How big is it?
- Lets see if we can look at its contents

Why did `cat /bin/ls` not help?

- Because whatever is in it its is not ASCII encode.

How about looking trying to dump its contents by look at the values of its bytes

- there are several utilities that we can use to “dump” a file’s contents
    - These programs read the file and convert the bytes into ascii representations of the value for each byte
        - you can use `man -k dump` to see commands that have the word dump in their names
            - the one we will focus on are `xxd` but two others that are really useful are `od` (octal dump) and `hd` (hexdump)

```
$ man xxd
XXD(1)                      General Commands Manual
XXD(1)

NAME
       xxd - make a hexdump or do the reverse.

SYNOPSIS
       xxd -h[elp]
       xxd [options] [infile [outfile]]
       xxd -r[evert] [options] [infile [outfile]]

DESCRIPTION
       xxd  creates a hex dump of a given file or standard input.  It can
also
       convert a hex dump back to its original binary form.  Like
uuencode(1)
       and  uudecode(1)  it allows the transmission of binary data in a
`mail-
       safe' ASCII representation, but has the advantage of decoding to
stan-
       dard output.  Moreover, it can be used to perform binary file
patching.

OPTIONS
       If  no infile is given, standard input is read.  If infile is
specified
       as a `-' character, then input is taken from  standard  input.   If
no
       outfile is given (or a `-' character is in its place), results are
sent
       to standard output.

       Note that a "lazy" parser is used which does not check  for  more
than
       the  first option letter, unless the option is followed by a
parameter.
       Spaces between a single option letter and its parameter  are
optional.
       Parameters to options can be specified in decimal, hexadecimal or
octal
       notation.  Thus -c8, -c 8, -c 010 and -cols 8 are all equivalent.

       -a | -autoskip
              Toggle autoskip: A single '*' replaces nul-lines.  Default off.

       -b | -bits
              Switch to bits (binary digits) dump, rather than hexdump.
This
              option  writes octets as eight digits "1"s and "0"s instead of
a
              normal hexadecimal dump. Each line is preceded by a line
number
              in  hexadecimal and followed by an ascii (or ebcdic)
representa-
              tion. The command line switches -r, -p, -i do not work with
this
              mode.

       -c cols | -cols cols
              Format  <cols> octets per line. Default 16 (-i: 12, -ps: 30, -
b:
              6). Max 256.

       -C | -capitalize
              Capitalize variable names in C include file  style,  when
using
              -i.

       -E | -EBCDIC
              Change the character encoding in the righthand column from
ASCII
              to EBCDIC.  This does not change the hexadecimal
representation.
              The option is meaningless in combinations with -r, -p or -i.

       -e     Switch to little-endian hexdump.  This option treats byte
groups
              as words in little-endian byte order.  The default grouping of
4
              bytes may be changed using -g.  This option only applies to
hex-
              dump, leaving the ASCII (or  EBCDIC)  representation
```

```
unchanged.
              The command line switches -r, -p, -i do not work with this
mode.

       -g bytes | -groupsize bytes
              Separate  the  output of every <bytes> bytes (two hex
characters
              or eight bit-digits each) by a whitespace.  Specify -g 0 to
sup-
              press grouping.  <Bytes> defaults to 2 in normal mode, 4 in
lit-
              tle-endian mode and 1 in bits mode.  Grouping does not apply
to
              postscript or include style.

       -h | -help
              Print  a summary of available commands and exit.  No hex
dumping
              is performed.

       -i | -include
              Output in C include file style. A complete static array
defini-
              tion  is  written (named after the input file), unless xxd
reads
              from stdin.

       -l len | -len len
              Stop after writing <len> octets.

       -o offset
              Add <offset> to the displayed file position.

       -p | -ps | -postscript | -plain
              Output in postscript continuous hexdump  style.  Also  known
as
              plain hexdump style.

       -r | -revert
              Reverse  operation:  convert (or patch) hexdump into binary.
If
              not writing to stdout, xxd writes into its output  file
without
              truncating it. Use the combination -r -p to read plain
hexadeci-
              mal dumps without line number information and without a
particu-
              lar column layout. Additional Whitespace and line-breaks are
al-
              lowed anywhere.

       -seek offset
              When used after -r: revert with <offset> added to file
positions
              found in hexdump.

       -s [+][-]seek
              Start at <seek> bytes abs. (or rel.) infile offset.  +
indicates
              that the seek is relative to the  current  stdin  file
position
              (meaningless when not reading from stdin).  - indicates that
the
              seek should be that many characters from the end  of  the
input
              (or if combined with +: before the current stdin file
position).
              Without -s option, xxd starts at the current file position.

       -u     Use upper case hex letters. Default is lower case.

       -v | -version
              Show version string.

CAVEATS
       xxd -r has some builtin magic while evaluating line number
information.
       If  the  output  file is seekable, then the linenumbers at the start
of
       each hexdump line may be out of order, lines may be missing,  or
over-
       lapping.  In these cases xxd will lseek(2) to the next position. If
the
       output file is not seekable, only  gaps  are  allowed,  which  will
be
       filled by null-bytes.
```

```
       xxd -r never generates parse errors. Garbage is silently skipped.

       When  editing hexdumps, please note that xxd -r skips everything on
the
       input line after reading enough columns of hexadecimal data (see
option
       -c).  This  also means, that changes to the printable ascii (or
ebcdic)
       columns are always ignored. Reverting a  plain  (or  postscript)
style
       hexdump  with  xxd  -r -p does not depend on the correct number of
col-
       umns. Here anything that looks like a  pair  of  hex-digits  is
inter-
       preted.

       Note the difference between
       % xxd -i file
       and
       % xxd -i < file

       xxd  -s +seek may be different from xxd -s seek, as lseek(2) is used
to
       "rewind" input.  A '+' makes a difference if the input source is
stdin,
       and  if  stdin's  file  position is not at the start of the file by
the
       time xxd is started and given its input.  The  following  examples
may
       help to clarify (or further confuse!)...

       Rewind  stdin before reading; needed because the `cat' has already
read
       to the end of stdin.
       % sh -c "cat > plain_copy; xxd -s 0 > hex_copy" < file

       Hexdump from file position 0x480 (=1024+128)  onwards.   The  `+'
sign
       means "relative to the current position", thus the `128' adds to the
1k
       where dd left off.
       % sh -c "dd of=plain_snippet bs=1k count=1; xxd -s +128 >
hex_snippet"
       < file

       Hexdump from file position 0x100 ( = 1024-768) on.
       % sh -c "dd of=plain_snippet bs=1k count=1; xxd -s +-768 >
hex_snippet"
       < file

       However, this is a rare situation and the use of `+' is rarely
needed.
       The  author  prefers  to  monitor  the  effect of xxd with strace(1)
or
       truss(1), whenever -s is used.

EXAMPLES
       Print everything but the first three lines (hex 0x30 bytes) of file.
       % xxd -s 0x30 file

       Print 3 lines (hex 0x30 bytes) from the end of file.
       % xxd -s -0x30 file

       Print 120 bytes as continuous hexdump with 20 octets per line.
       % xxd -l 120 -ps -c 20 xxd.1
       2e54482058584420312022417567757374203139
       39362220224d616e75616c20706167652064666f72
       20787864220a2e5c220a2e5c222032317374204d
       617920313939360a2e5c22204d616e2070616765
       20617574686f723a0a2e5c2220202020546f6e79
       204e7567656e74203c746f6e79407363746e7567

       Hexdump the first 120 bytes of this man page with 12 octets per line.
       % xxd -l 120 -c 12 xxd.1
       0000000: 2e54 4820 5858 4420 3120 2241  .TH XXD 1 "A
       000000c: 7567 7573 7420 3139 3936 2220  ugust 1996"
       0000018: 224d 616e 7561 6c20 7061 6765  "Manual page
       0000024: 2066 6f72 2078 7864 220a 2e5c   for xxd"..\
       0000030: 220a 2e5c 2220 3231 7374 204d  "..\" 21st M
       000003c: 6179 2031 3939 360a 2e5c 2220  ay 1996..\"
       0000048: 4d61 6e20 7061 6765 2061 7574  Man page aut
       0000054: 686f 723a 0a2e 5c22 2020 2020  hor:..\"
       0000060: 546f 6e79 204e 7567 656e 7420  Tony Nugent
       000006c: 3c74 6f6e 7940 7363 746e 7567  <tony@sctnug
```

```
       Display just the date from the file xxd.1
       % xxd -s 0x36 -l 13 -c 13 xxd.1
       0000036: 3231 7374 204d 6179 2031 3939 36  21st May 1996

       Copy input_file to output_file and prepend 100 bytes of value 0x00.
       % xxd input_file | xxd -r -s 100 > output_file

       Patch the date in the file xxd.1
       % echo "0000037: 3574 68" | xxd -r - xxd.1
       % xxd -s 0x36 -l 13 -c 13 xxd.1
       0000036: 3235 7468 204d 6179 2031 3939 36  25th May 1996

       Create a 65537 byte file with all bytes 0x00, except for the  last
one
       which is 'A' (hex 0x41).
       % echo "010000: 41" | xxd -r > file

       Hexdump this file with autoskip.
       % xxd -a -c 12 file
       0000000: 0000 0000 0000 0000 0000 0000  ............
       *
       000fffc: 0000 0000 40                   ....A

       Create a 1 byte file containing a single 'A' character.  The number
af-
       ter '-r -s' adds to the linenumbers found in the file; in  effect,
the
       leading bytes are suppressed.
       % echo "010000: 41" | xxd -r -s -0x10000 > file

       Use xxd as a filter within an editor such as vim(1) to hexdump a
region
       marked between `a' and `z'.
       :'a,'z!xxd

       Use xxd as a filter within an editor such as vim(1) to recover a
binary
       hexdump marked between `a' and `z'.
       :'a,'z!xxd -r

       Use xxd as a filter within an editor such as vim(1) to recover one
line
       of a hexdump.  Move the cursor over the line and type:
       !!xxd -r

       Read single characters from a serial line
       % xxd -c1 < /dev/term/b &
       % stty < /dev/term/b -echo -opost -isig -icanon min 1
       % echo -n foo > /dev/term/b

RETURN VALUES
       The following error values are returned:

       0      no errors encountered.

       -1     operation not supported ( xxd -r -i still impossible).

       1      error while parsing options.

       2      problems with input file.

       3      problems with output file.

       4,5    desired seek position is unreachable.

SEE ALSO
       uuencode(1), uudecode(1), patch(1)

WARNINGS
       The tools weirdness matches its creators brain.  Use entirely  at
your
       own risk. Copy files. Trace it. Become a wizard.

VERSION
       This manual page documents xxd version 1.7

AUTHOR
       (c) 1990-1997 by Juergen Weigert
       <jnweiger@informatik.uni-erlangen.de>

       Distribute freely and credit me,
       make money and share with me,
       lose money and don't ask me.

       Manual page started by Tony Nugent
       <tony@sctnugen.ppp.gu.edu.au> <T.Nugent@sct.gu.edu.au>
```

```
       Small changes by Bram Moolenaar.  Edited by Juergen Weigert.

Manual page for xxd              August 1996
XXD(1)
$
```

7.1.1.1. Lets use xxd to look at the first 80 bytes of the /bin/ls

First in hexadecimal notation and then in binary (base 2)

xxd command to display first 80 bytes (-l 80) of the file in units/groups of 1 byte (-g 1) values with 8 units per line (-c 8):

`xxd -l 80 -g 1 -c 8 /bin/ls`

and

same as above but using binary (base 2) notation (-b) for each value:

`xxd -l 80 -g 1 -c 8 -b /bin/ls`

```
$ xxd -l 80 -g 1 -c 8 /bin/ls
00000000: 7f 45 4c 46 02 01 01 00  .ELF....
00000008: 00 00 00 00 00 00 00 00  ........
00000010: 03 00 3e 00 01 00 00 00  ..>.....
00000018: d0 67 00 00 00 00 00 00  .g......
00000020: 40 00 00 00 00 00 00 00  @.......
00000028: c0 23 02 00 00 00 00 00  .#......
00000030: 00 00 00 00 40 00 38 00  ....@.8.
00000038: 0d 00 40 00 1e 00 1d 00  ..@.....
00000040: 06 00 00 00 04 00 00 00  ........
00000048: 40 00 00 00 00 00 00 00  @.......
$ xxd -l 80 -g 1 -c 8 -b /bin/ls
00000000: 01111111 01000101 01001100 01000110 00000010 00000001 00000001
00000000  .ELF....
00000008: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000  ........
00000010: 00000011 00000000 00111110 00000000 00000001 00000000 00000000
00000000  ..>.....
00000018: 11010000 01100111 00000000 00000000 00000000 00000000 00000000
00000000  .g......
00000020: 01000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000  @.......
00000028: 11000000 00100011 00000010 00000000 00000000 00000000 00000000
00000000  .#......
00000030: 00000000 00000000 00000000 00000000 01000000 00000000 00111000
00000000  ....@.8.
00000038: 00001101 00000000 01000000 00000000 00011110 00000000 00011101
00000000  ..@.....
00000040: 00000110 00000000 00000000 00000000 00000100 00000000 00000000
00000000  ........
00000048: 01000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000  @.......
$
```

Ok so while that's a cool party trick ... so what do they mean? What else can we do?

- Lets see what the `file` command can tell us.

```
$ man file
FILE(1)                   BSD General Commands Manual
FILE(1)

NAME
     file — determine file type

SYNOPSIS
     file [-bcdEhiklLNnprsSvzZ0] [--apple] [--extension] [--mime-encoding]
          [--mime-type] [-e testname] [-F separator] [-f namefile]
          [-m magicfiles] [-P name=value] file ...
     file -C [-m magicfiles]
     file [--help]

DESCRIPTION
     This manual page documents version 5.38 of the file command.

     file tests each argument in an attempt to classify it.  There are three
     sets of tests, performed in this order: filesystem tests, magic tests,
     and language tests.  The first test that succeeds causes the file type
to
     be printed.

     The type printed will usually contain one of the words text (the file
     contains only printing characters and a few common control characters
and
     is probably safe to read on an ASCII terminal), executable (the file
con-
     tains the result of compiling a program in a form understandable to some
     UNIX kernel or another), or data meaning anything else (data is usually
     “binary” or non-printable).  Exceptions are well-known file formats
(core
     files, tar archives) that are known to contain binary data.  When adding
     local definitions to /etc/magic, make sure to preserve these keywords.
     Users depend on knowing that all the readable files in a directory have
     the word “text” printed.  Don't do as Berkeley did and change “shell
     commands text” to “shell script”.

     The filesystem tests are based on examining the return from a stat(2)
     system call.  The program checks to see if the file is empty, or if it's
     some sort of special file.  Any known file types appropriate to the sys-
     tem you are running on (sockets, symbolic links, or named pipes (FIFOs)
     on those systems that implement them) are intuited if they are defined
in
     the system header file <sys/stat.h>.

     The magic tests are used to check for files with data in particular
fixed
     formats.  The canonical example of this is a binary executable (compiled
     program) a.out file, whose format is defined in <elf.h>, <a.out.h> and
     possibly <exec.h> in the standard include directory.  These files have a
     “magic number” stored in a particular place near the beginning of the
     file that tells the UNIX operating system that the file is a binary exe-
     cutable, and which of several types thereof.  The concept of a “magic”
     has been applied by extension to data files.  Any file with some invari-
     ant identifier at a small fixed offset into the file can usually be de-
     scribed in this way.  The information identifying these files is read
     from /etc/magic and the compiled magic file /usr/share/misc/magic.mgc,
or
     the files in the directory /usr/share/misc/magic if the compiled file
     does not exist.  In addition, if $HOME/.magic.mgc or $HOME/.magic
exists,
     it will be used in preference to the system magic files.

     If a file does not match any of the entries in the magic file, it is ex-
     amined to see if it seems to be a text file.  ASCII, ISO-8859-x, non-ISO
     8-bit extended-ASCII character sets (such as those used on Macintosh and
     IBM PC systems), UTF-8-encoded Unicode, UTF-16-encoded Unicode, and
     EBCDIC character sets can be distinguished by the different ranges and
     sequences of bytes that constitute printable text in each set.  If a
file
     passes any of these tests, its character set is reported.  ASCII,
     ISO-8859-x, UTF-8, and extended-ASCII files are identified as “text” be-
     cause they will be mostly readable on nearly any terminal; UTF-16 and
     EBCDIC are only “character data” because, while they contain text, it is
     text that will require translation before it can be read.  In addition,
     file will attempt to determine other characteristics of text-type files.
     If the lines of a file are terminated by CR, CRLF, or NEL, instead of
the
     Unix-standard LF, this will be reported.  Files that contain embedded
es-
     cape sequences or overstriking will also be identified.
```

```
     Once file has determined the character set used in a text-type file, it
     will attempt to determine in what language the file is written.  The
lan-
     guage tests look for particular strings (cf.  <names.h>) that can appear
     anywhere in the first few blocks of a file.  For example, the keyword
.br
     indicates that the file is most likely a troff(1) input file, just as
the
     keyword struct indicates a C program.  These tests are less reliable
than
     the previous two groups, so they are performed last.  The language test
     routines also test for some miscellany (such as tar(1) archives, JSON
     files).

     Any file that cannot be identified as having been written in any of the
     character sets listed above is simply said to be “data”.

OPTIONS
     --apple
             Causes the file command to output the file type and creator code
             as used by older MacOS versions.  The code consists of eight
let-
             ters, the first describing the file type, the latter the
creator.
             This option works properly only for file formats that have the
             apple-style output defined.

     -b, --brief
             Do not prepend filenames to output lines (brief mode).

     -C, --compile
             Write a magic.mgc output file that contains a pre-parsed version
             of the magic file or directory.

     -c, --checking-printout
             Cause a checking printout of the parsed form of the magic file.
             This is usually used in conjunction with the -m flag to debug a
             new magic file before installing it.

     -d      Prints internal debugging information to stderr.

     -E      On filesystem errors (file not found etc), instead of handling
             the error as regular output as POSIX mandates and keep going,
is-
             sue an error message and exit.

     -e, --exclude testname
             Exclude the test named in testname from the list of tests made
to
             determine the file type.  Valid test names are:

             apptype   EMX application type (only on EMX).

             ascii     Various types of text files (this test will try to
                       guess the text encoding, irrespective of the setting
of
                       the ‘encoding’ option).

             encoding  Different text encodings for soft magic tests.

             tokens    Ignored for backwards compatibility.

             cdf       Prints details of Compound Document Files.

             compress  Checks for, and looks inside, compressed files.

             csv       Checks Comma Separated Value files.

             elf       Prints ELF file details, provided soft magic tests are
                       enabled and the elf magic is found.

             json      Examines JSON (RFC-7159) files by parsing them for
com-
                       pliance.

             soft      Consults magic files.

             tar       Examines tar files by verifying the checksum of the
512
                       byte tar header.  Excluding this test can provide more
                       detailed content description by using the soft magic
                       method.

             text      A synonym for ‘ascii’.

     --extension
```

```
              Print a slash-separated list of valid extensions for the file
              type found.

     -F, --separator separator
              Use the specified string as the separator between the filename
              and the file result returned.  Defaults to ':'.

     -f, --files-from namefile
              Read the names of the files to be examined from namefile (one
per
              line) before the argument list.  Either namefile or at least one
              filename argument must be present; to test the standard input,
              use '-' as a filename argument.  Please note that namefile is
un-
              wrapped and the enclosed filenames are processed when this
option
              is encountered and before any further options processing is
done.
              This allows one to process multiple lists of files with
different
              command line arguments on the same file invocation.  Thus if you
              want to set the delimiter, you need to do it before you specify
              the list of files, like: "-F @ -f namefile", instead of: "-f
              namefile -F @".

     -h, --no-dereference
              option causes symlinks not to be followed (on systems that sup-
              port symbolic links).  This is the default if the environment
              variable POSIXLY_CORRECT is not defined.

     -i, --mime
              Causes the file command to output mime type strings rather than
              the more traditional human readable ones.  Thus it may say
              'text/plain; charset=us-ascii' rather than "ASCII text".

     --mime-type, --mime-encoding
              Like -i, but print only the specified element(s).

     -k, --keep-going
              Don't stop at the first match, keep going.  Subsequent matches
              will be have the string '\012- ' prepended.  (If you want a new-
              line, see the -r option.)  The magic pattern with the highest
              strength (see the -l option) comes first.

     -l, --list
              Shows a list of patterns and their strength sorted descending by
              magic(5) strength which is used for the matching (see also the -
k
              option).

     -L, --dereference
              option causes symlinks to be followed, as the like-named option
              in ls(1) (on systems that support symbolic links).  This is the
              default if the environment variable POSIXLY_CORRECT is defined.

     -m, --magic-file magicfiles
              Specify an alternate list of files and directories containing
              magic.  This can be a single item, or a colon-separated list.
If
              a compiled magic file is found alongside a file or directory, it
              will be used instead.

     -N, --no-pad
              Don't pad filenames so that they align in the output.

     -n, --no-buffer
              Force stdout to be flushed after checking each file.  This is
              only useful if checking a list of files.  It is intended to be
              used by programs that want filetype output from a pipe.

     -p, --preserve-date
              On systems that support utime(3) or utimes(2), attempt to pre-
              serve the access time of files analyzed, to pretend that file
              never read them.

     -P, --parameter name=value
              Set various parameter limits.

                    Name         Default    Explanation
                    indir        15         recursion limit for indirect magic
                    name         30         use count limit for name/use magic
                    elf_notes    256        max ELF notes processed
                    elf_phnum    128        max ELF program sections processed
                    elf_shnum    32768      max ELF sections processed
                    regex        8192       length limit for regex searches
                    bytes        1048576    max number of bytes to read from
```

```
file

     -r, --raw
             Don't translate unprintable characters to \ooo.  Normally file
             translates unprintable characters to their octal representation.

     -s, --special-files
             Normally, file only attempts to read and determine the type of
             argument files which stat(2) reports are ordinary files.  This
             prevents problems, because reading special files may have pecu-
             liar consequences.  Specifying the -s option causes file to also
             read argument files which are block or character special files.
             This is useful for determining the filesystem types of the data
             in raw disk partitions, which are block special files.  This op-
             tion also causes file to disregard the file size as reported by
             stat(2) since on some systems it reports a zero size for raw
disk
             partitions.

     -S, --no-sandbox
             On systems where libseccomp
             (https://github.com/seccomp/libseccomp) is available, the -S
flag
             disables sandboxing which is enabled by default.  This option is
             needed for file to execute external decompressing programs, i.e.
             when the -z flag is specified and the built-in decompressors are
             not available.  On systems where sandboxing is not available,
             this option has no effect.

             Note: This Debian version of file was built without seccomp sup-
             port, so this option has no effect.

     -v, --version
             Print the version of the program and exit.

     -z, --uncompress
             Try to look inside compressed files.

     -Z, --uncompress-noreport
             Try to look inside compressed files, but report information
about
             the contents only not the compression.

     -0, --print0
             Output a null character '\0' after the end of the filename.
Nice
             to cut(1) the output.  This does not affect the separator, which
             is still printed.

             If this option is repeated more than once, then file prints just
             the filename followed by a NUL followed by the description (or
             ERROR: text) followed by a second NUL for each entry.

     --help  Print a help message and exit.

ENVIRONMENT
     The environment variable MAGIC can be used to set the default magic file
     name.  If that variable is set, then file will not attempt to open
     $HOME/.magic.  file adds ".mgc" to the value of this variable as appro-
     priate.  The environment variable POSIXLY_CORRECT controls (on systems
     that support symbolic links), whether file will attempt to follow sym-
     links or not.  If set, then file follows symlink, otherwise it does not.
     This is also controlled by the -L and -h options.

FILES
     /usr/share/misc/magic.mgc  Default compiled list of magic.
     /usr/share/misc/magic      Directory containing default magic files.

EXIT STATUS
     file will exit with 0 if the operation was successful or >0 if an error
     was encountered.  The following errors cause diagnostic messages, but
     don't affect the program exit code (as POSIX requires), unless -E is
     specified:
           •   A file cannot be found
           •   There is no permission to read a file
           •   The file type cannot be determined

EXAMPLES
           $ file file.c file /dev/{wd0a,hda}
           file.c:   C program text
           file:     ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV),
                     dynamically linked (uses shared libs), stripped
           /dev/wd0a: block special (0/0)
           /dev/hda: block special (3/0)
```

```
          $ file -s /dev/wd0{b,d}
          /dev/wd0b: data
          /dev/wd0d: x86 boot sector

          $ file -s /dev/hda{,1,2,3,4,5,6,7,8,9,10}
          /dev/hda:   x86 boot sector
          /dev/hda1:  Linux/i386 ext2 filesystem
          /dev/hda2:  x86 boot sector
          /dev/hda3:  x86 boot sector, extended partition table
          /dev/hda4:  Linux/i386 ext2 filesystem
          /dev/hda5:  Linux/i386 swap file
          /dev/hda6:  Linux/i386 swap file
          /dev/hda7:  Linux/i386 swap file
          /dev/hda8:  Linux/i386 swap file
          /dev/hda9:  empty
          /dev/hda10: empty

          $ file -i file.c file /dev/{wd0a,hda}
          file.c:      text/x-c
          file:        application/x-executable
          /dev/hda:    application/x-not-regular-file
          /dev/wd0a:   application/x-not-regular-file

SEE ALSO
     hexdump(1), od(1), strings(1), magic(5)

STANDARDS CONFORMANCE
     This program is believed to exceed the System V Interface Definition of
     FILE(CMD), as near as one can determine from the vague language
contained
     therein.  Its behavior is mostly compatible with the System V program of
     the same name.  This version knows more magic, however, so it will pro-
     duce different (albeit more accurate) output in many cases.

     The one significant difference between this version and System V is that
     this version treats any white space as a delimiter, so that spaces in
     pattern strings must be escaped.  For example,

          >10     string  language impress        (imPRESS data)

     in an existing magic file would have to be changed to

          >10     string  language\ impress       (imPRESS data)

     In addition, in this version, if a pattern string contains a backslash,
     it must be escaped.  For example

          0       string          \begindata      Andrew Toolkit document

     in an existing magic file would have to be changed to

          0       string          \\begindata     Andrew Toolkit document

     SunOS releases 3.2 and later from Sun Microsystems include a file
command
     derived from the System V one, but with some extensions.  This version
     differs from Sun's only in minor ways.  It includes the extension of the
     '&' operator, used as, for example,

          >16     long&0x7fffffff >0              not stripped

SECURITY
     On systems where libseccomp (https://github.com/seccomp/libseccomp) is
     available, file is enforces limiting system calls to only the ones
neces-
     sary for the operation of the program.  This enforcement does not
provide
     any security benefit when file is asked to decompress input files
running
     external programs with the -z option.  To enable execution of external
     decompressors, one needs to disable sandboxing using the -S flag.

MAGIC DIRECTORY
     The magic file entries have been collected from various sources, mainly
     USENET, and contributed by various authors.  Christos Zoulas (address
be-
     low) will collect additional or corrected magic file entries.  A
consoli-
     dation of magic file entries will be distributed periodically.

     The order of entries in the magic file is significant.  Depending on
what
     system you are using, the order that they are put together may be incor-
     rect.
```

```
HISTORY
     There has been a file command in every UNIX since at least Research
     Version 4 (man page dated November, 1973).  The System V version intro-
     duced one significant major change: the external list of magic types.
     This slowed the program down slightly but made it a lot more flexible.

     This program, based on the System V version, was written by Ian Darwin
     ⟨ian@darwinsys.com⟩ without looking at anybody else's source code.

     John Gilmore revised the code extensively, making it better than the
     first version.  Geoff Collyer found several inadequacies and provided
     some magic file entries.  Contributions of the '&' operator by Rob McMa-
     hon, ⟨cudcv@warwick.ac.uk⟩, 1989.

     Guy Harris, ⟨guy@netapp.com⟩, made many changes from 1993 to the
present.

     Primary development and maintenance from 1990 to the present by Christos
     Zoulas ⟨christos@astron.com⟩.

     Altered by Chris Lowth ⟨chris@lowth.com⟩, 2000: handle the -i option to
     output mime type strings, using an alternative magic file and internal
     logic.

     Altered by Eric Fischer ⟨enf@pobox.com⟩, July, 2000, to identify charac-
     ter codes and attempt to identify the languages of non-ASCII files.

     Altered by Reuben Thomas ⟨rrt@sc3d.org⟩, 2007-2011, to improve MIME sup-
     port, merge MIME and non-MIME magic, support directories as well as
files
     of magic, apply many bug fixes, update and fix a lot of magic, improve
     the build system, improve the documentation, and rewrite the Python
bind-
     ings in pure Python.

     The list of contributors to the 'magic' directory (magic files) is too
     long to include here.  You know who you are; thank you.  Many contribu-
     tors are listed in the source files.

LEGAL NOTICE
     Copyright (c) Ian F. Darwin, Toronto, Canada, 1986-1999.  Covered by the
     standard Berkeley Software Distribution copyright; see the file COPYING
     in the source distribution.

     The files tar.h and is_tar.c were written by John Gilmore from his pub-
     lic-domain tar(1) program, and are not covered by the above license.

BUGS
     Please report bugs and send patches to the bug tracker at
     https://bugs.astron.com/ or the mailing list at ⟨file@astron.com⟩ (visit
     https://mailman.astron.com/mailman/listinfo/file first to subscribe).

TODO
     Fix output so that tests for MIME and APPLE flags are not needed all
over
     the place, and actual output is only done in one place.  This needs a
de-
     sign.  Suggestion: push possible outputs on to a list, then pick the
     last-pushed (most specific, one hopes) value at the end, or use a
default
     if the list is empty.  This should not slow down evaluation.

     The handling of MAGIC_CONTINUE and printing \012- between entries is
     clumsy and complicated; refactor and centralize.

     Some of the encoding logic is hard-coded in encoding.c and can be moved
     to the magic files if we had a !:charset annotation

     Continue to squash all magic bugs.  See Debian BTS for a good source.

     Store arbitrarily long strings, for example for %s patterns, so that
they
     can be printed out.  Fixes Debian bug #271672.  This can be done by
allo-
     cating strings in a string pool, storing the string pool at the end of
     the magic file and converting all the string pointers to relative
offsets
     from the string pool.

     Add syntax for relative offsets after current level (Debian bug
#466037).

     Make file -ki work, i.e. give multiple MIME types.

     Add a zip library so we can peek inside Office2007 documents to print
     more details about their contents.
```

```
     Add an option to print URLs for the sources of the file descriptions.

     Combine script searches and add a way to map executable names to MIME
     types (e.g. have a magic value for !:mime which causes the resulting
     string to be looked up in a table).  This would avoid adding the same
     magic repeatedly for each new hash-bang interpreter.

     When a file descriptor is available, we can skip and adjust the buffer
     instead of the hacky buffer management we do now.

     Fix “name” and “use” to check for consistency at compile time (duplicate
     “name”, “use” pointing to undefined “name” ).  Make “name” / “use” more
     efficient by keeping a sorted list of names.  Special-case ^ to flip en-
     dianness in the parser so that it does not have to be escaped, and docu-
     ment it.

     If the offsets specified internally in the file exceed the buffer size (
     HOWMANY variable in file.h), then we don't seek to that offset, but we
     give up.  It would be better if buffer managements was done when the
file
     descriptor is available so move around the file.  One must be careful
     though because this has performance (and thus security considerations).

AVAILABILITY
     You can obtain the original author's latest version by anonymous FTP on
     ftp.astron.com in the directory /pub/file/file-X.YZ.tar.gz.

BSD                             July 13, 2019
BSD
$
```

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=2f15ad836be3339dec0e2e6a3c637e08e48aacbd, for GNU/Linux 3.2.0,
stripped
$
```

Ok it is an ELF file let’s see what, if anything, the manual has to say about `elf`.

```
$ man elf
ELF(5)                     Linux Programmer's Manual
ELF(5)

NAME
       elf - format of Executable and Linking Format (ELF) files

SYNOPSIS
       #include <elf.h>

DESCRIPTION
       The  header  file  <elf.h>  defines the format of ELF executable
binary
       files.  Amongst these files are normal  executable  files,
relocatable
       object files, core files, and shared objects.

       An executable file using the ELF file format consists of an ELF
header,
       followed by a program header table or a section header table, or
both.
       The  ELF  header  is  always  at  offset zero of the file.  The
program
       header table and the section header table's offset in the file are
de-
       fined  in the ELF header.  The two tables describe the rest of the
par-
       ticularities of the file.

       This header file describes the above mentioned headers as C
structures
       and  also includes structures for dynamic sections, relocation
sections
       and symbol tables.

   Basic types
       The following types are used for  N-bit  architectures  (N=32,64,
ElfN
       stands for Elf32 or Elf64, uintN_t stands for uint32_t or uint64_t):

           ElfN_Addr       Unsigned program address, uintN_t
           ElfN_Off        Unsigned file offset, uintN_t
           ElfN_Section    Unsigned section index, uint16_t
           ElfN_Versym     Unsigned version symbol information, uint16_t
           Elf_Byte        unsigned char
           ElfN_Half       uint16_t
           ElfN_Sword      int32_t
           ElfN_Word       uint32_t
           ElfN_Sxword     int64_t
           ElfN_Xword      uint64_t

       (Note:  the  *BSD terminology is a bit different.  There, Elf64_Half
is
       twice as large as Elf32_Half, and Elf64Quarter is  used  for
uint16_t.
       In  order  to avoid confusion these types are replaced by explicit
ones
       in the below.)

       All data structures that the file format defines follow  the
"natural"
       size  and  alignment  guidelines for the relevant class.  If
necessary,
       data structures contain explicit padding to ensure 4-byte alignment
for
       4-byte objects, to force structure sizes to a multiple of 4, and so
on.

   ELF header (Ehdr)
       The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:

           #define EI_NIDENT 16

           typedef struct {
               unsigned char e_ident[EI_NIDENT];
               uint16_t      e_type;
               uint16_t      e_machine;
               uint32_t      e_version;
               ElfN_Addr     e_entry;
               ElfN_Off      e_phoff;
               ElfN_Off      e_shoff;
               uint32_t      e_flags;
               uint16_t      e_ehsize;
```

```
               uint16_t      e_phentsize;
               uint16_t      e_phnum;
               uint16_t      e_shentsize;
               uint16_t      e_shnum;
               uint16_t      e_shstrndx;
           } ElfN_Ehdr;

       The fields have the following meanings:

       e_ident   This  array of bytes specifies how to interpret the file,
in-
                 dependent of the processor or the file's remaining
contents.
                 Within  this array everything is named by macros, which
start
                 with the prefix EI_ and may contain values which  start
with
                 the prefix ELF.  The following macros are defined:

                 EI_MAG0  The  first  byte  of  the  magic number.  It must
be
                          filled with ELFMAG0.  (0: 0x7f)

                 EI_MAG1  The second byte of the magic  number.   It  must
be
                          filled with ELFMAG1.  (1: 'E')

                 EI_MAG2  The  third  byte  of  the  magic number.  It must
be
                          filled with ELFMAG2.  (2: 'L')

                 EI_MAG3  The fourth byte of the magic  number.   It  must
be
                          filled with ELFMAG3.  (3: 'F')

                 EI_CLASS The  fifth byte identifies the architecture for
this
                          binary:

                          ELFCLASSNONE  This class is invalid.
                          ELFCLASS32    This defines the 32-bit
architecture.
                                        It  supports  machines  with files
and
                                        virtual address spaces up to  4
Giga-
                                        bytes.
                          ELFCLASS64    This defines the 64-bit architecture.

                 EI_DATA  The  sixth  byte  specifies the data encoding of
the
                          processor-specific data  in  the  file.
Currently,
                          these encodings are supported:

                          ELFDATANONE   Unknown data format.
                          ELFDATA2LSB   Two's complement, little-endian.
                          ELFDATA2MSB   Two's complement, big-endian.

                 EI_VERSION
                          The  seventh  byte  is the version number of the
ELF
                          specification:

                          EV_NONE       Invalid version.
                          EV_CURRENT    Current version.

                 EI_OSABI The eighth byte identifies the operating system
and
                          ABI to which the object is targeted.  Some fields
in
                          other ELF structures have flags and values that
have
                          platform-specific  meanings;  the  interpretation
of
                          those fields is determined  by  the  value  of
this
                          byte.  For example:

                          ELFOSABI_NONE        Same as ELFOSABI_SYSV
                          ELFOSABI_SYSV        UNIX System V ABI
                          ELFOSABI_HPUX        HP-UX ABI
                          ELFOSABI_NETBSD      NetBSD ABI
                          ELFOSABI_LINUX       Linux ABI
                          ELFOSABI_SOLARIS     Solaris ABI
                          ELFOSABI_IRIX        IRIX ABI
```

```
                        ELFOSABI_FREEBSD     FreeBSD ABI
                        ELFOSABI_TRU64       TRU64 UNIX ABI
                        ELFOSABI_ARM         ARM architecture ABI
                        ELFOSABI_STANDALONE  Stand-alone (embedded) ABI

                EI_ABIVERSION
                        The  ninth byte identifies the version of the ABI
to
                        which the object is targeted.  This field is used
to
                        distinguish  among  incompatible versions of an
ABI.
                        The interpretation of this version number is
depen-
                        dent  on  the  ABI identified by the EI_OSABI
field.
                        Applications conforming to  this  specification
use
                        the value 0.

                EI_PAD  Start  of padding.  These bytes are reserved and
set
                        to zero.  Programs which  read  them  should
ignore
                        them.   The  value for EI_PAD will change in the
fu-
                        ture if currently unused bytes are given meanings.

                EI_NIDENT
                        The size of the e_ident array.

       e_type   This member of the structure identifies the object file
type:

                ET_NONE         An unknown type.
                ET_REL          A relocatable file.
                ET_EXEC         An executable file.
                ET_DYN          A shared object.
                ET_CORE         A core file.

       e_machine This member specifies the required architecture for an
indi-
                vidual file.  For example:

                EM_NONE         An unknown machine
                EM_M32          AT&T WE 32100
                EM_SPARC        Sun Microsystems SPARC
                EM_386          Intel 80386
                EM_68K          Motorola 68000
                EM_88K          Motorola 88000
                EM_860          Intel 80860
                EM_MIPS         MIPS RS3000 (big-endian only)
                EM_PARISC       HP/PA
                EM_SPARC32PLUS  SPARC with enhanced instruction set
                EM_PPC          PowerPC
                EM_PPC64        PowerPC 64-bit
                EM_S390         IBM S/390
                EM_ARM          Advanced RISC Machines
                EM_SH           Renesas SuperH
                EM_SPARCV9      SPARC v9 64-bit
                EM_IA_64        Intel Itanium
                EM_X86_64       AMD x86-64
                EM_VAX          DEC Vax

       e_version This member identifies the file version:

                EV_NONE         Invalid version
                EV_CURRENT      Current version

       e_entry  This  member  gives  the  virtual address to which the
system
                first transfers control, thus starting the process.   If
the
                file has no associated entry point, this member holds zero.

       e_phoff  This  member  holds the program header table's file offset
in
                bytes.  If the file has no program header table, this
member
                holds zero.

       e_shoff  This  member  holds the section header table's file offset
in
                bytes.  If the file has no section header table, this
member
                holds zero.
```

```
       e_flags   This  member  holds  processor-specific flags associated
with
                 the file.  Flag names take the form EF_`machine_flag'.
Cur-
                 rently, no flags have been defined.

       e_ehsize  This member holds the ELF header's size in bytes.

       e_phentsize
                 This  member  holds  the  size  in  bytes of one entry in
the
                 file's program header table; all entries are the same size.

       e_phnum   This member holds the number of entries in the program
header
                 table.  Thus the product of e_phentsize and e_phnum gives
the
                 table's size in bytes.  If a  file  has  no  program
header,
                 e_phnum holds the value zero.

                 If  the  number  of  entries  in  the program header table
is
                 larger than or equal to PN_XNUM (0xffff), this  member
holds
                 PN_XNUM  (0xffff)  and the real number of entries in the
pro-
                 gram header table is held in the sh_info member of  the
ini-
                 tial  entry  in section header table.  Otherwise, the
sh_info
                 member of the initial entry contains the value zero.

                 PN_XNUM  This is defined as 0xffff, the largest number
e_ph-
                          num  can have, specifying where the actual number
of
                          program headers is assigned.

       e_shentsize
                 This member holds a sections header's size in bytes.  A
sec-
                 tion header is one entry in the section header table; all
en-
                 tries are the same size.

       e_shnum   This member holds the number of entries in the section
header
                 table.  Thus the product of e_shentsize and e_shnum gives
the
                 section header table's size in bytes.  If a file has no
sec-
                 tion header table, e_shnum holds the value of zero.

                 If  the  number  of  entries  in  the section header table
is
                 larger than or equal to SHN_LORESERVE (0xff00), e_shnum
holds
                 the  value zero and the real number of entries in the
section
                 header table is held in the sh_size member of the initial
en-
                 try  in  section header table.  Otherwise, the sh_size
member
                 of the initial entry in the section header  table  holds
the
                 value zero.

       e_shstrndx
                 This member holds the section header table index of the
entry
                 associated with the section name string table.  If  the
file
                 has no section name string table, this member holds the
value
                 SHN_UNDEF.

                 If the index of section name string table section  is
larger
                 than  or  equal  to SHN_LORESERVE (0xff00), this member
holds
                 SHN_XINDEX (0xffff) and the real index of  the  section
name
                 string  table  section  is  held in the sh_link member of
the
```

```
                initial  entry  in  section  header  table.   Otherwise,
the
                sh_link  member  of the initial entry in section header
table
                contains the value zero.

   Program header (Phdr)
       An executable or shared object file's program header table is an
array
       of  structures, each describing a segment or other information the
sys-
       tem needs to prepare the program for execution.  An object file
segment
       contains one or more sections.  Program headers are meaningful only
for
       executable and shared object files.  A file specifies its  own
program
       header size with the ELF header's e_phentsize and e_phnum members.
The
       ELF program header is described by the type  Elf32_Phdr  or
Elf64_Phdr
       depending on the architecture:

           typedef struct {
               uint32_t   p_type;
               Elf32_Off  p_offset;
               Elf32_Addr p_vaddr;
               Elf32_Addr p_paddr;
               uint32_t   p_filesz;
               uint32_t   p_memsz;
               uint32_t   p_flags;
               uint32_t   p_align;
           } Elf32_Phdr;

           typedef struct {
               uint32_t   p_type;
               uint32_t   p_flags;
               Elf64_Off  p_offset;
               Elf64_Addr p_vaddr;
               Elf64_Addr p_paddr;
               uint64_t   p_filesz;
               uint64_t   p_memsz;
               uint64_t   p_align;
           } Elf64_Phdr;

       The  main  difference  between the 32-bit and the 64-bit program
header
       lies in the location of the p_flags member in the total struct.

       p_type    This member of the structure indicates what kind  of
segment
                 this  array  element  describes or how to interpret the
array
                 element's information.

                 PT_NULL     The array element is unused and  the  other
mem-
                             bers'  values  are undefined.  This lets the
pro-
                             gram header have ignored entries.

                 PT_LOAD     The array element specifies a  loadable
segment,
                             described  by  p_filesz  and  p_memsz.  The
bytes
                             from the file are mapped to the beginning of
the
                             memory  segment.   If  the  segment's memory
size
                             p_memsz is larger than the  file  size
p_filesz,
                             the "extra" bytes are defined to hold the value
0
                             and to follow  the  segment's  initialized
area.
                             The  file  size may not be larger than the
memory
                             size.  Loadable segment entries  in  the
program
                             header table appear in ascending order, sorted
on
                             the p_vaddr member.

                 PT_DYNAMIC  The array element specifies dynamic  linking
in-
                             formation.
```

```
                PT_INTERP   The array element specifies the location and
size
                            of a null-terminated pathname to invoke as an
in-
                            terpreter.   This segment type is meaningful
only
                            for executable files (though  it  may  occur
for
                            shared  objects).   However it may not occur
more
                            than once in a file.  If it is present,  it
must
                            precede any loadable segment entry.

                PT_NOTE     The array element specifies the location of
notes
                            (ElfN_Nhdr).

                PT_SHLIB    This segment type is reserved but has
unspecified
                            semantics.   Programs  that contain an array
ele-
                            ment of this type do not conform to the ABI.

                PT_PHDR     The array element, if present, specifies the
lo-
                            cation  and  size of the program header table
it-
                            self, both in the file and in the memory image
of
                            the  program.   This  segment  type may not
occur
                            more than once in a file.  Moreover, it may
occur
                            only  if  the program header table is part of
the
                            memory image of the program.  If it  is
present,
                            it must precede any loadable segment entry.

                PT_LOPROC, PT_HIPROC
                            Values   in   the   inclusive  range
[PT_LOPROC,
                            PT_HIPROC] are  reserved  for  processor-
specific
                            semantics.

                PT_GNU_STACK
                            GNU  extension  which is used by the Linux
kernel
                            to control the state of the stack via  the
flags
                            set in the p_flags member.

       p_offset  This  member  holds the offset from the beginning of the
file
                 at which the first byte of the segment resides.

       p_vaddr   This member holds the virtual address at which the first
byte
                 of the segment resides in memory.

       p_paddr   On  systems  for  which physical addressing is relevant,
this
                 member is reserved for the segment's physical address.
Under
                 BSD this member is not used and must be zero.

       p_filesz  This  member  holds  the number of bytes in the file image
of
                 the segment.  It may be zero.

       p_memsz   This member holds the number of bytes in the memory image
of
                 the segment.  It may be zero.

       p_flags   This  member  holds  a bit mask of flags relevant to the
seg-
                 ment:

                 PF_X   An executable segment.
                 PF_W   A writable segment.
                 PF_R   A readable segment.

                 A text segment commonly has the flags PF_X and PF_R.  A
```

```
data
                segment commonly has PF_W and PF_R.

       p_align   This member holds the value to which the segments are
aligned
                 in memory and in the file.  Loadable  process  segments
must
                 have  congruent  values  for p_vaddr and p_offset, modulo
the
                 page size.  Values of zero and one mean no alignment  is
re-
                 quired.   Otherwise,  p_align  should be a positive,
integral
                 power of two,  and  p_vaddr  should  equal  p_offset,
modulo
                 p_align.

   Section header (Shdr)
       A  file's section header table lets one locate all the file's
sections.
       The section header table is an array of Elf32_Shdr or Elf64_Shdr
struc-
       tures.   The ELF header's e_shoff member gives the byte offset from
the
       beginning of the file to the section header table.  e_shnum  holds
the
       number of entries the section header table contains.  e_shentsize
holds
       the size in bytes of each entry.

       A section header table index is a subscript into this array.  Some
sec-
       tion  header  table indices are reserved: the initial entry and the
in-
       dices between SHN_LORESERVE and SHN_HIRESERVE.  The  initial  entry
is
       used  in  ELF  extensions for e_phnum, e_shnum and e_shstrndx; in
other
       cases, each field in the initial entry is set to zero.  An object
file
       does not have sections for these special indices:

       SHN_UNDEF
              This value marks an undefined, missing, irrelevant, or
otherwise
              meaningless section reference.

       SHN_LORESERVE
              This value specifies the lower bound of the  range  of
reserved
              indices.

       SHN_LOPROC, SHN_HIPROC
              Values  greater  in the inclusive range [SHN_LOPROC,
SHN_HIPROC]
              are reserved for processor-specific semantics.

       SHN_ABS
              This value specifies the absolute value  for  the
corresponding
              reference.   For  example,  a symbol defined relative to
section
              number SHN_ABS has an absolute value and is not affected by
re-
              location.

       SHN_COMMON
              Symbols  defined  relative  to  this section are common
symbols,
              such as FORTRAN COMMON or unallocated C external variables.

       SHN_HIRESERVE
              This value specifies the upper bound of the  range  of
reserved
              indices.   The system reserves indices between SHN_LORESERVE
and
              SHN_HIRESERVE, inclusive.  The section  header  table  does
not
              contain entries for the reserved indices.

       The section header has the following structure:

           typedef struct {
               uint32_t   sh_name;
               uint32_t   sh_type;
               uint32_t   sh_flags;
```

```
               Elf32_Addr sh_addr;
               Elf32_Off  sh_offset;
               uint32_t   sh_size;
               uint32_t   sh_link;
               uint32_t   sh_info;
               uint32_t   sh_addralign;
               uint32_t   sh_entsize;
           } Elf32_Shdr;

           typedef struct {
               uint32_t   sh_name;
               uint32_t   sh_type;
               uint64_t   sh_flags;
               Elf64_Addr sh_addr;
               Elf64_Off  sh_offset;
               uint64_t   sh_size;
               uint32_t   sh_link;
               uint32_t   sh_info;
               uint64_t   sh_addralign;
               uint64_t   sh_entsize;
           } Elf64_Shdr;

       No  real  differences exist between the 32-bit and 64-bit section
head-
       ers.

       sh_name   This member specifies the name of the section.  Its value
is
                 an index into the section header string table section,
giving
                 the location of a null-terminated string.

       sh_type   This member categorizes the section's contents and
semantics.

                 SHT_NULL       This value marks the section header  as
inac-
                                tive.  It does not have an associated
section.
                                Other members of the section header have
unde-
                                fined values.

                 SHT_PROGBITS   This  section holds information defined by
the
                                program, whose format and meaning  are
deter-
                                mined solely by the program.

                 SHT_SYMTAB     This section holds a symbol table.
Typically,
                                SHT_SYMTAB provides symbols for link
editing,
                                though  it  may also be used for dynamic
link-
                                ing.  As a complete symbol table, it may
con-
                                tain  many  symbols  unnecessary  for
dynamic
                                linking.  An object file can  also  contain
a
                                SHT_DYNSYM section.

                 SHT_STRTAB     This  section holds a string table.  An
object
                                file may have multiple string table sections.

                 SHT_RELA       This section holds relocation entries with
ex-
                                plicit  addends,  such  as type Elf32_Rela
for
                                the 32-bit class of object files.   An
object
                                may have multiple relocation sections.

                 SHT_HASH       This  section  holds  a symbol hash table.
An
                                object participating in dynamic  linking
must
                                contain  a  symbol hash table.  An object
file
                                may have only one hash table.

                 SHT_DYNAMIC    This section  holds  information  for
dynamic
                                linking.  An object file may have only one
```

```
dy-
                                namic section.
                SHT_NOTE        This section holds notes (ElfN_Nhdr).
                SHT_NOBITS      A section of this type occupies  no  space
in
                                the file but otherwise resembles
SHT_PROGBITS.
                                Although this section contains no  bytes,
the
                                sh_offset  member contains the conceptual
file
                                offset.
                SHT_REL         This section holds relocation offsets
without
                                explicit  addends,  such as type Elf32_Rel
for
                                the 32-bit class of object files.   An
object
                                file may have multiple relocation sections.
                SHT_SHLIB       This  section  is reserved but has
unspecified
                                semantics.
                SHT_DYNSYM      This section holds a minimal  set  of
dynamic
                                linking symbols.  An object file can also
con-
                                tain a SHT_SYMTAB section.
                SHT_LOPROC, SHT_HIPROC
                                Values in  the  inclusive  range
[SHT_LOPROC,
                                SHT_HIPROC]  are  reserved  for processor-
spe-
                                cific semantics.
                SHT_LOUSER      This value specifies the lower  bound  of
the
                                range of indices reserved for application
pro-
                                grams.
                SHT_HIUSER      This value specifies the upper  bound  of
the
                                range of indices reserved for application
pro-
                                grams.  Section types between  SHT_LOUSER
and
                                SHT_HIUSER  may  be  used  by the
application,
                                without conflicting  with  current  or
future
                                system-defined section types.
       sh_flags Sections  support  one-bit  flags that describe
miscellaneous
                attributes.  If a flag bit is set in sh_flags, the
attribute
                is  "on"  for the section.  Otherwise, the attribute is
"off"
                or does not apply.  Undefined attributes are set to zero.
                SHF_WRITE       This section  contains  data  that  should
be
                                writable during process execution.
                SHF_ALLOC       This  section  occupies  memory during
process
                                execution.  Some control sections do  not
re-
                                side  in  the  memory image of an object
file.
                                This attribute is off for those sections.
                SHF_EXECINSTR   This section contains executable  machine
in-
                                structions.
                SHF_MASKPROC    All  bits  included  in this mask are
reserved
                                for processor-specific semantics.
```

```
       sh_addr   If this section appears in the memory  image  of  a
process,
                 this  member  holds  the address at which the section's
first
                 byte should reside.  Otherwise, the member contains zero.

       sh_offset This member's value holds the byte offset from the
beginning
                 of  the  file  to the first byte in the section.  One
section
                 type, SHT_NOBITS, occupies no space  in  the  file,  and
its
                 sh_offset  member  locates  the  conceptual  placement in
the
                 file.

       sh_size   This member holds the section's size in  bytes.   Unless
the
                 section  type  is  SHT_NOBITS,  the  section occupies
sh_size
                 bytes in the file.  A section of type SHT_NOBITS may  have
a
                 nonzero size, but it occupies no space in the file.

       sh_link   This  member  holds  a section header table index link,
whose
                 interpretation depends on the section type.

       sh_info   This member holds extra information, whose interpretation
de-
                 pends on the section type.

       sh_addralign
                 Some  sections have address alignment constraints.  If a
sec-
                 tion holds a doubleword, the system  must  ensure
doubleword
                 alignment  for  the  entire  section.   That is, the value
of
                 sh_addr must be congruent to zero, modulo the value of
sh_ad-
                 dralign.   Only  zero and positive integral powers of two
are
                 allowed.  The value 0 or 1 means  that  the  section  has
no
                 alignment constraints.

       sh_entsize
                 Some  sections hold a table of fixed-sized entries, such as
a
                 symbol table.  For such a section, this member gives the
size
                 in  bytes  for  each entry.  This member contains zero if
the
                 section does not hold a table of fixed-size entries.

       Various sections hold program and control information:

       .bss      This section holds uninitialized data that contributes to
the
                 program's  memory  image.  By definition, the system
initial-
                 izes the data with zeros when  the  program  begins  to
run.
                 This  section is of type SHT_NOBITS.  The attribute types
are
                 SHF_ALLOC and SHF_WRITE.

       .comment  This section holds version control information.  This
section
                 is of type SHT_PROGBITS.  No attribute types are used.

       .ctors    This  section holds initialized pointers to the C++
construc-
                 tor functions.  This section is of  type  SHT_PROGBITS.
The
                 attribute types are SHF_ALLOC and SHF_WRITE.

       .data     This  section  holds  initialized data that contribute to
the
                 program's memory image.  This section is  of  type
SHT_PROG-
                 BITS.  The attribute types are SHF_ALLOC and SHF_WRITE.

       .data1    This  section  holds  initialized data that contribute to
the
```

```
                program's memory image.  This section is  of  type
SHT_PROG-
                BITS.  The attribute types are SHF_ALLOC and SHF_WRITE.

       .debug   This  section  holds information for symbolic debugging.
The
                contents are unspecified.  This section is of type
SHT_PROG-
                BITS.  No attribute types are used.

       .dtors   This section holds initialized pointers to the C++
destructor
                functions.  This section is of type SHT_PROGBITS.  The
attri-
                bute types are SHF_ALLOC and SHF_WRITE.

       .dynamic This  section  holds  dynamic  linking information.  The
sec-
                tion's attributes will include the  SHF_ALLOC  bit.
Whether
                the SHF_WRITE bit is set is processor-specific.  This
section
                is of type SHT_DYNAMIC.  See the attributes above.

       .dynstr  This section holds strings needed for dynamic  linking,
most
                commonly the strings that represent the names associated
with
                symbol table entries.  This section is  of  type
SHT_STRTAB.
                The attribute type used is SHF_ALLOC.

       .dynsym  This  section  holds  the dynamic linking symbol table.
This
                section is of type SHT_DYNSYM.  The attribute used is
SHF_AL-
                LOC.

       .fini    This section holds executable instructions that contribute
to
                the process termination code.  When a program exits
normally
                the  system  arranges  to  execute  the code in this
section.
                This section is of type SHT_PROGBITS.   The  attributes
used
                are SHF_ALLOC and SHF_EXECINSTR.

       .gnu.version
                This  section  holds  the  version  symbol table, an array
of
                ElfN_Half elements.  This section is of type
SHT_GNU_versym.
                The attribute type used is SHF_ALLOC.

       .gnu.version_d
                This section holds the version symbol definitions, a table
of
                ElfN_Verdef   structures.    This   section   is   of
type
                SHT_GNU_verdef.  The attribute type used is SHF_ALLOC.

       .gnu.version_r
                This  section holds the version symbol needed elements, a
ta-
                ble of ElfN_Verneed structures.   This  section  is  of
type
                SHT_GNU_versym.  The attribute type used is SHF_ALLOC.

       .got     This  section holds the global offset table.  This section
is
                of type SHT_PROGBITS.  The attributes are processor-
specific.

       .hash    This section holds a symbol hash table.  This section  is
of
                type SHT_HASH.  The attribute used is SHF_ALLOC.

       .init    This section holds executable instructions that contribute
to
                the process initialization code.  When a  program  starts
to
                run  the  system arranges to execute the code in this
section
                before calling the main program entry point.  This section
is
```

```
                of  type SHT_PROGBITS.  The attributes used are SHF_ALLOC
and
                SHF_EXECINSTR.

       .interp  This section holds the pathname of a program interpreter.
If
                the  file  has  a loadable segment that includes the
section,
                the section's attributes  will  include  the  SHF_ALLOC
bit.
                Otherwise,  that  bit  will  be off.  This section is of
type
                SHT_PROGBITS.

       .line    This section holds line number information for  symbolic
de-
                bugging,  which describes the correspondence between the
pro-
                gram source and the machine code.  The contents are
unspeci-
                fied.   This  section  is of type SHT_PROGBITS.  No
attribute
                types are used.

       .note    This section holds various notes.  This section  is  of
type
                SHT_NOTE.  No attribute types are used.

       .note.ABI-tag
                This  section is used to declare the expected run-time ABI
of
                the ELF image.  It may include the operating system name
and
                its  run-time  versions.   This  section is of type
SHT_NOTE.
                The only attribute used is SHF_ALLOC.

       .note.gnu.build-id
                This section is used to hold an ID that  uniquely
identifies
                the contents of the ELF image.  Different files with the
same
                build ID should contain the same executable content.  See
the
                --build-id  option  to  the  GNU linker (ld (1)) for more
de-
                tails.  This section is of type SHT_NOTE.  The only
attribute
                used is SHF_ALLOC.

       .note.GNU-stack
                This  section  is  used  in  Linux object files for
declaring
                stack attributes.  This section is of type SHT_PROGBITS.
The
                only  attribute used is SHF_EXECINSTR.  This indicates to
the
                GNU linker that the object file requires an executable
stack.

       .note.openbsd.ident
                OpenBSD native executables usually contain  this  section
to
                identify  themselves so the kernel can bypass any
compatibil-
                ity ELF binary emulation tests when loading the file.

       .plt     This section holds the procedure linkage table.  This
section
                is  of  type SHT_PROGBITS.  The attributes are processor-
spe-
                cific.

       .relNAME  This section holds relocation information as described
below.
                If  the file has a loadable segment that includes
relocation,
                the section's attributes  will  include  the  SHF_ALLOC
bit.
                Otherwise,  the  bit  will  be off.  By convention, "NAME"
is
                supplied by the section to which the relocations apply.
Thus
                a  relocation  section for .text normally would have the
name
                .rel.text.  This section is of type SHT_REL.
```

```
       .relaNAME This section holds relocation information as described
below.
                 If  the file has a loadable segment that includes
relocation,
                 the section's attributes  will  include  the  SHF_ALLOC
bit.
                 Otherwise,  the  bit  will  be off.  By convention, "NAME"
is
                 supplied by the section to which the relocations apply.
Thus
                 a  relocation  section for .text normally would have the
name
                 .rela.text.  This section is of type SHT_RELA.

       .rodata   This section holds read-only data that typically
contributes
                 to  a nonwritable segment in the process image.  This
section
                 is of type SHT_PROGBITS.  The attribute used is SHF_ALLOC.

       .rodata1  This section holds read-only data that typically
contributes
                 to  a nonwritable segment in the process image.  This
section
                 is of type SHT_PROGBITS.  The attribute used is SHF_ALLOC.

       .shstrtab This section holds section names.  This section  is  of
type
                 SHT_STRTAB.  No attribute types are used.

       .strtab   This  section  holds  strings, most commonly the strings
that
                 represent the names associated with symbol table entries.
If
                 the  file  has  a  loadable  segment that includes the
symbol
                 string table,  the  section's  attributes  will  include
the
                 SHF_ALLOC bit.  Otherwise, the bit will be off.  This
section
                 is of type SHT_STRTAB.

       .symtab   This section holds a symbol table.  If the file has  a
load-
                 able  segment  that  includes the symbol table, the
section's
                 attributes will include the SHF_ALLOC  bit.   Otherwise,
the
                 bit will be off.  This section is of type SHT_SYMTAB.

       .text     This section holds the "text", or executable instructions,
of
                 a program.  This section is of type  SHT_PROGBITS.   The
at-
                 tributes used are SHF_ALLOC and SHF_EXECINSTR.

   String and symbol tables
       String  table  sections  hold null-terminated character sequences,
com-
       monly called strings.  The object file uses these strings to
represent
       symbol and section names.  One references a string as an index into
the
       string table section.  The first byte, which is index zero, is
defined
       to  hold  a null byte ('\0').  Similarly, a string table's last byte
is
       defined to hold a null byte, ensuring null termination for all
strings.

       An object file's symbol table holds information needed  to  locate
and
       relocate a program's symbolic definitions and references.  A symbol
ta-
       ble index is a subscript into this array.

           typedef struct {
               uint32_t      st_name;
               Elf32_Addr    st_value;
               uint32_t      st_size;
               unsigned char st_info;
               unsigned char st_other;
               uint16_t      st_shndx;
           } Elf32_Sym;
```

```
           typedef struct {
               uint32_t      st_name;
               unsigned char st_info;
               unsigned char st_other;
               uint16_t      st_shndx;
               Elf64_Addr    st_value;
               uint64_t      st_size;
           } Elf64_Sym;

       The 32-bit and 64-bit versions have the same members, just in a
differ-
       ent order.

       st_name   This  member  holds  an  index  into the object file's
symbol
                 string table, which holds character  representations  of
the
                 symbol  names.   If  the  value  is  nonzero, it represents
a
                 string table index that gives the  symbol  name.
Otherwise,
                 the symbol has no name.

       st_value  This member gives the value of the associated symbol.

       st_size   Many  symbols  have associated sizes.  This member holds
zero
                 if the symbol has no size or an unknown size.

       st_info   This member specifies  the  symbol's  type  and  binding
at-
                 tributes:

                 STT_NOTYPE  The symbol's type is not defined.

                 STT_OBJECT  The symbol is associated with a data object.

                 STT_FUNC    The symbol is associated with a function or
other
                             executable code.

                 STT_SECTION The symbol is associated with a section.
Symbol
                             table  entries  of  this type exist primarily
for
                             relocation and normally have STB_LOCAL bindings.

                 STT_FILE    By convention, the symbol's name gives  the
name
                             of  the  source  file  associated with the
object
                             file.  A file symbol has STB_LOCAL bindings,
its
                             section  index  is  SHN_ABS,  and it precedes
the
                             other STB_LOCAL symbols of the  file,  if  it
is
                             present.

                 STT_LOPROC, STT_HIPROC
                             Values   in   the  inclusive  range
[STT_LOPROC,
                             STT_HIPROC] are reserved  for  processor-
specific
                             semantics.

                 STB_LOCAL   Local  symbols are not visible outside the
object
                             file containing their definition.  Local
symbols
                             of  the  same  name  may  exist in multiple
files
                             without interfering with each other.

                 STB_GLOBAL  Global symbols are visible to  all  object
files
                             being  combined.   One  file's  definition  of
a
                             global symbol will satisfy another  file's
unde-
                             fined reference to the same symbol.

                 STB_WEAK    Weak  symbols  resemble global symbols, but
their
                             definitions have lower precedence.
```

```
                STB_LOPROC, STB_HIPROC
                           Values  in  the  inclusive   range
[STB_LOPROC,
                           STB_HIPROC]  are  reserved for processor-
specific
                           semantics.

                There are macros for packing and unpacking  the  binding
and
                type fields:

                ELF32_ST_BIND(info), ELF64_ST_BIND(info)
                       Extract a binding from an st_info value.

                ELF32_ST_TYPE(info), ELF64_ST_TYPE(info)
                       Extract a type from an st_info value.

                ELF32_ST_INFO(bind, type), ELF64_ST_INFO(bind, type)
                       Convert a binding and a type into an st_info value.

       st_other  This member defines the symbol visibility.

                STV_DEFAULT     Default  symbol visibility rules.  Global
and
                                weak symbols are available to other
modules;
                                references  in the local module can be
inter-
                                posed by definitions in other modules.
                STV_INTERNAL    Processor-specific hidden class.
                STV_HIDDEN      Symbol is unavailable to other modules;
ref-
                                erences in the local module always resolve
to
                                the local symbol (i.e., the symbol  can't
be
                                interposed by definitions in other modules).
                STV_PROTECTED   Symbol  is  available  to  other modules,
but
                                references in the local module always
resolve
                                to the local symbol.

                There are macros for extracting the visibility type:

                ELF32_ST_VISIBILITY(other) or ELF64_ST_VISIBILITY(other)

       st_shndx  Every  symbol  table  entry  is "defined" in relation to
some
                section.  This member holds the relevant section header
table
                index.

   Relocation entries (Rel & Rela)
       Relocation  is  the process of connecting symbolic references with
sym-
       bolic definitions.  Relocatable files must have  information  that
de-
       scribes  how to modify their section contents, thus allowing
executable
       and shared object files to hold the right information for  a
process's
       program image.  Relocation entries are these data.

       Relocation structures that do not need an addend:

           typedef struct {
               Elf32_Addr r_offset;
               uint32_t   r_info;
           } Elf32_Rel;

           typedef struct {
               Elf64_Addr r_offset;
               uint64_t   r_info;
           } Elf64_Rel;

       Relocation structures that need an addend:

           typedef struct {
               Elf32_Addr r_offset;
               uint32_t   r_info;
               int32_t    r_addend;
           } Elf32_Rela;

           typedef struct {
               Elf64_Addr r_offset;
```

```
               uint64_t   r_info;
               int64_t    r_addend;
           } Elf64_Rela;

       r_offset  This  member gives the location at which to apply the
reloca-
                 tion action.  For a relocatable file, the value is  the
byte
                 offset  from the beginning of the section to the storage
unit
                 affected by the relocation.  For an executable file or
shared
                 object,  the value is the virtual address of the storage
unit
                 affected by the relocation.

       r_info    This member gives both the symbol table index with respect
to
                 which  the relocation must be made and the type of
relocation
                 to apply.  Relocation types are processor-specific.  When
the
                 text refers to a relocation entry's relocation type or
symbol
                 table   index,   it   means   the    result    of
applying
                 ELF[32|64]_R_TYPE  or  ELF[32|64]_R_SYM, respectively, to
the
                 entry's r_info member.

       r_addend  This member specifies a constant addend used to  compute
the
                 value to be stored into the relocatable field.

   Dynamic tags (Dyn)
       The .dynamic section contains a series of structures that hold
relevant
       dynamic linking information.  The d_tag member controls the
interpreta-
       tion of d_un.

           typedef struct {
               Elf32_Sword    d_tag;
               union {
                   Elf32_Word d_val;
                   Elf32_Addr d_ptr;
               } d_un;
           } Elf32_Dyn;
           extern Elf32_Dyn _DYNAMIC[];

           typedef struct {
               Elf64_Sxword    d_tag;
               union {
                   Elf64_Xword d_val;
                   Elf64_Addr  d_ptr;
               } d_un;
           } Elf64_Dyn;
           extern Elf64_Dyn _DYNAMIC[];

       d_tag     This member may have any of the following values:

                 DT_NULL     Marks end of dynamic section

                 DT_NEEDED   String table offset to name of a needed library

                 DT_PLTRELSZ Size in bytes of PLT relocation entries

                 DT_PLTGOT   Address of PLT and/or GOT

                 DT_HASH     Address of symbol hash table

                 DT_STRTAB   Address of string table

                 DT_SYMTAB   Address of symbol table

                 DT_RELA     Address of Rela relocation table

                 DT_RELASZ   Size in bytes of the Rela relocation table

                 DT_RELAENT  Size in bytes of a Rela relocation table entry

                 DT_STRSZ    Size in bytes of string table

                 DT_SYMENT   Size in bytes of a symbol table entry

                 DT_INIT     Address of the initialization function
```

```
                DT_FINI     Address of the termination function
                DT_SONAME   String table offset to name of shared object
                DT_RPATH    String  table offset to library search path
(dep-
                            recated)
                DT_SYMBOLIC Alert linker to search this shared object
before
                            the executable for symbols
                DT_REL      Address of Rel relocation table
                DT_RELSZ    Size in bytes of Rel relocation table
                DT_RELENT   Size in bytes of a Rel table entry
                DT_PLTREL   Type  of relocation entry to which the PLT
refers
                            (Rela or Rel)
                DT_DEBUG    Undefined use for debugging
                DT_TEXTREL  Absence of this entry indicates that  no
reloca-
                            tion  entries  should apply to a nonwritable
seg-
                            ment
                DT_JMPREL   Address of relocation entries  associated
solely
                            with the PLT
                DT_BIND_NOW Instruct  dynamic  linker  to process all
reloca-
                            tions before transferring  control  to  the
exe-
                            cutable
                DT_RUNPATH  String table offset to library search path
                DT_LOPROC, DT_HIPROC
                            Values   in   the   inclusive  range
[DT_LOPROC,
                            DT_HIPROC] are  reserved  for  processor-
specific
                            semantics

       d_val    This  member represents integer values with various
interpre-
                tations.

       d_ptr    This member represents program virtual addresses.   When
in-
                terpreting these addresses, the actual address should be
com-
                puted based on the original file value and  memory  base
ad-
                dress.   Files  do  not  contain  relocation entries to
fixup
                these addresses.

       _DYNAMIC  Array containing all the dynamic structures in  the
.dynamic
                section.  This is automatically populated by the linker.

   Notes (Nhdr)
       ELF  notes  allow for appending arbitrary information for the system
to
       use.  They are largely used by core files (e_type of ET_CORE), but
many
       projects define their own set of extensions.  For example, the GNU
tool
       chain uses ELF notes to pass information from the linker to the  C
li-
       brary.

       Note sections contain a series of notes (see the struct definitions
be-
       low).  Each note is followed by the name field (whose length is
defined
       in  n_namesz) and then by the descriptor field (whose length is
defined
       in n_descsz) and whose starting address has a 4 byte  alignment.
```

```
Nei-
       ther  field  is  defined  in  the  note  struct  due to their
arbitrary
       lengths.

       An example for parsing out two consecutive notes should  clarify
their
       layout in memory:

           void *memory, *name, *desc;
           Elf64_Nhdr *note, *next_note;

           /* The buffer is pointing to the start of the section/segment */
           note = memory;

           /* If the name is defined, it follows the note */
           name = note->n_namesz == 0 ? NULL : memory + sizeof(*note);

           /* If the descriptor is defined, it follows the name
              (with alignment) */

           desc = note->n_descsz == 0 ? NULL :
                  memory + sizeof(*note) + ALIGN_UP(note->n_namesz, 4);

           /* The next note follows both (with alignment) */
           next_note = memory + sizeof(*note) +
                               ALIGN_UP(note->n_namesz, 4) +
                               ALIGN_UP(note->n_descsz, 4);

       Keep in mind that the interpretation of n_type depends on the
namespace
       defined by the n_namesz field.  If the n_namesz field is not set
(e.g.,
       is 0), then there are two sets of notes: one for core files and one
for
       all other ELF types.  If the namespace is unknown, then tools will
usu-
       ally fallback to these sets of notes as well.

           typedef struct {
               Elf32_Word n_namesz;
               Elf32_Word n_descsz;
               Elf32_Word n_type;
           } Elf32_Nhdr;

           typedef struct {
               Elf64_Word n_namesz;
               Elf64_Word n_descsz;
               Elf64_Word n_type;
           } Elf64_Nhdr;

       n_namesz  The length of the name field in bytes.  The contents will
im-
                 mediately follow this note in memory.  The name is null
ter-
                 minated.   For  example,  if the name is "GNU", then
n_namesz
                 will be set to 4.

       n_descsz  The length of the descriptor field in  bytes.   The
contents
                 will immediately follow the name field in memory.

       n_type    Depending  on  the  value  of the name field, this member
may
                 have any of the following values:

                 Core files (e_type = ET_CORE)
                      Notes used by all core files.  These are highly
operat-
                      ing  system  or  architecture specific and often
require
                      close coordination with kernels, C libraries, and
debug-
                      gers.   These are used when the namespace is the
default
                      (i.e., n_namesz will be set to 0), or  a  fallback
when
                      the namespace is unknown.

                      NT_PRSTATUS          prstatus struct
                      NT_FPREGSET          fpregset struct
                      NT_PRPSINFO          prpsinfo struct
                      NT_PRXREG            prxregset struct
                      NT_TASKSTRUCT        task structure
                      NT_PLATFORM          String from sysinfo(SI_PLATFORM)
```

```
                       NT_AUXV             auxv array
                       NT_GWINDOWS         gwindows struct
                       NT_ASRS             asrset struct
                       NT_PSTATUS          pstatus struct
                       NT_PSINFO           psinfo struct
                       NT_PRCRED           prcred struct
                       NT_UTSNAME          utsname struct
                       NT_LWPSTATUS        lwpstatus struct
                       NT_LWPSINFO         lwpinfo struct
                       NT_PRFPXREG         fprxregset struct
                       NT_SIGINFO          siginfo_t (size might increase
over
                                           time)
                       NT_FILE             Contains information  about
mapped
                                           files
                       NT_PRXFPREG         user_fxsr_struct
                       NT_PPC_VMX          PowerPC Altivec/VMX registers
                       NT_PPC_SPE          PowerPC SPE/EVR registers
                       NT_PPC_VSX          PowerPC VSX registers
                       NT_386_TLS          i386 TLS slots (struct user_desc)
                       NT_386_IOPERM       x86 io permission bitmap (1=deny)
                       NT_X86_XSTATE       x86 extended state using xsave
                       NT_S390_HIGH_GPRS   s390 upper register halves
                       NT_S390_TIMER       s390 timer register
                       NT_S390_TODCMP      s390  time-of-day  (TOD) clock
com-
                                           parator register
                       NT_S390_TODPREG     s390 time-of-day (TOD)
programmable
                                           register
                       NT_S390_CTRS        s390 control registers
                       NT_S390_PREFIX      s390 prefix register
                       NT_S390_LAST_BREAK  s390 breaking event address
                       NT_S390_SYSTEM_CALL s390 system call restart data
                       NT_S390_TDB         s390 transaction diagnostic block
                       NT_ARM_VFP          ARM VFP/NEON registers
                       NT_ARM_TLS          ARM TLS register
                       NT_ARM_HW_BREAK     ARM hardware breakpoint registers
                       NT_ARM_HW_WATCH     ARM hardware watchpoint registers
                       NT_ARM_SYSTEM_CALL  ARM system call number

                 n_name = GNU
                       Extensions used by the GNU tool chain.

                       NT_GNU_ABI_TAG
                              Operating  system (OS) ABI information.  The
desc
                              field will be 4 words:

                              • word  0:  OS   descriptor
(ELF_NOTE_OS_LINUX,
                                ELF_NOTE_OS_GNU, and so on)`
                              • word 1: major version of the ABI
                              • word 2: minor version of the ABI
                              • word 3: subminor version of the ABI

                       NT_GNU_HWCAP
                              Synthetic  hwcap information.  The desc field
be-
                              gins with two words:

                              • word 0: number of entries
                              • word 1: bit mask of enabled entries

                              Then follow  variable-length  entries,  one
byte
                              followed  by a null-terminated hwcap name
string.
                              The byte gives the bit number to test if
enabled,
                              (1U << bit) & bit mask.

                       NT_GNU_BUILD_ID
                              Unique  build  ID  as  generated by the GNU
ld(1)
                              --build-id option.  The desc consists of any
non-
                              zero number of bytes.

                       NT_GNU_GOLD_VERSION
                              The  desc  contains  the  GNU Gold linker
version
                              used.

                  Default/unknown namespace (e_type != ET_CORE)
```

```
                       These are used when the namespace is the default
(i.e.,
                       n_namesz will be set to 0), or a fallback when the
name-
                       space is unknown.

                       NT_VERSION          A version string of some sort.
                       NT_ARCH             Architecture information.

NOTES
       ELF first appeared in System V.  The ELF format is an adopted
standard.

       The extensions for e_phnum, e_shnum  and  e_shstrndx  respectively
are
       Linux  extensions.   Sun,  BSD and AMD64 also support them; for
further
       information, look under SEE ALSO.

SEE ALSO
       as(1), elfedit(1), gdb(1), ld(1), nm(1), objdump(1), patchelf(1),
read-
       elf(1),  size(1),  strings(1), strip(1), execve(2),
dl_iterate_phdr(3),
       core(5), ld.so(8)

       Hewlett-Packard, Elf-64 Object File Format.

       Santa Cruz Operation, System V Application Binary Interface.

       UNIX System Laboratories, "Object Files", Executable and Linking
Format
       (ELF).

       Sun Microsystems, Linker and Libraries Guide.

       AMD64  ABI Draft, System V Application Binary Interface AMD64
Architec-
       ture Processor Supplement.

COLOPHON
       This page is part of release 5.05 of the Linux  man-pages  project.
A
       description  of  the project, information about reporting bugs, and
the
       latest    version    of    this    page,    can    be    found
at
       https://www.kernel.org/doc/man-pages/.

Linux                              2019-05-09
ELF(5)
$
```

We could keep going down this road to poke at its contents using command designed to decode the elf file and dump information about it

```
readelf --all /bin/ls
```

```
objdump --all /bin/ls
```

But let's try another approach before we stand back and put the pieces together.

Let's lookup what the OS kernel function for "executing" a binary file has to say

```
$ man 3 exec
EXEC(3)                    Linux Programmer's Manual
EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *pathname, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *pathname, const char *arg, ...
                       /*, (char *) NULL, char *const envp[] */);
       int execv(const char *pathname, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE

DESCRIPTION
       The  exec() family of functions replaces the current process image
with
       a new process image.  The functions described in this manual  page
are
       front-ends  for execve(2).  (See the manual page for execve(2) for
fur-
       ther details about the replacement of the current process image.)

       The initial argument for these functions is the name of a file that
is
       to be executed.

       The  functions can be grouped based on the letters following the
"exec"
       prefix.

   l - execl(), execlp(), execle()
       The const char *arg and subsequent ellipses can be thought of as
arg0,
       arg1, ..., argn.  Together they describe a list of one or more
pointers
       to null-terminated strings that represent the argument  list
available
       to  the  executed  program.   The first argument, by convention,
should
       point to the filename associated with the  file  being  executed.
The
       list  of  arguments  must  be  terminated by a null pointer, and,
since
       these are variadic functions, this pointer must be cast (char *) NULL.

       By contrast with the 'l' functions, the 'v' functions  (below)
specify
       the command-line arguments of the executed program as a vector.

   v - execv(), execvp(), execvpe()
       The  char *const argv[] argument is an array of pointers to null-
termi-
       nated strings that represent the argument list  available  to  the
new
       program.   The first argument, by convention, should point to the
file-
       name associated with the file being executed.  The  array  of
pointers
       must be terminated by a null pointer.

   e - execle(), execvpe()
       The  environment of the caller is specified via the argument envp.
The
       envp argument is an array of pointers to  null-terminated  strings
and
       must be terminated by a null pointer.

       All  other  exec()  functions  (which do not include 'e' in the
suffix)
```

```
       take the environment for the new process image from the external
vari-
       able environ in the calling process.

   p - execlp(), execvp(), execvpe()
       These  functions duplicate the actions of the shell in searching for
an
       executable file if the specified filename does not contain a slash
(/)
       character.  The file is sought in the colon-separated list of
directory
       pathnames specified in the PATH environment variable.  If this
variable
       isn't  defined,  the path list defaults to a list that includes the
di-
       rectories returned by confstr(_CS_PATH) (which  typically  returns
the
       value "/bin:/usr/bin") and possibly also the current working
directory;
       see NOTES for further details.

       If the specified filename includes a slash character, then PATH is
ig-
       nored, and the file at the specified pathname is executed.

       In addition, certain errors are treated specially.

       If permission is denied for a file (the attempted execve(2) failed
with
       the error EACCES), these functions will continue searching the rest
of
       the  search path.  If no other file is found, however, they will
return
       with errno set to EACCES.

       If the header of a  file  isn't  recognized  (the  attempted
execve(2)
       failed  with the error ENOEXEC), these functions will execute the
shell
       (/bin/sh) with the path of the file as its first  argument.   (If
this
       attempt fails, no further searching is done.)

       All  other  exec()  functions  (which do not include 'p' in the
suffix)
       take as their first argument a (relative  or  absolute)  pathname
that
       identifies the program to be executed.

RETURN VALUE
       The  exec() functions return only if an error has occurred.  The
return
       value is -1, and errno is set to indicate the error.

ERRORS
       All of these functions may fail and set errno for  any  of  the
errors
       specified for execve(2).

VERSIONS
       The execvpe() function first appeared in glibc 2.11.

ATTRIBUTES
       For  an  explanation  of  the  terms  used  in  this  section,  see
at-
       tributes(7).
```

Interface	Attribute	Value
execl(), execle(), execv()	Thread safety	MT-Safe
execlp(), execvp(), execvpe()	Thread safety	MT-Safe env

```
CONFORMING TO
       POSIX.1-2001, POSIX.1-2008.

       The execvpe() function is a GNU extension.

NOTES
       The default search path (used when the environment does not contain
the
       variable  PATH)  shows some variation across systems.  It generally
in-
       cludes /bin and /usr/bin (in that order) and may also include the
cur-
```

```
       rent  working directory.  On some other systems, the current working
is
       included after /bin and /usr/bin, as an anti-Trojan-horse measure.
The
       glibc  implementation  long  followed the traditional default where
the
       current working directory is included at the start of the search
path.
       However,  some  code  refactoring  during the development of glibc
2.24
       caused the current working directory to be dropped altogether from
the
       default  search  path.   This  accidental behavior change is
considered
       mildly beneficial, and won't be reverted.

       The behavior of execlp() and execvp() when errors occur while
attempt-
       ing to execute the file is historic practice, but has not
traditionally
       been documented and is not specified by the POSIX standard.   BSD
(and
       possibly  other  systems) do an automatic sleep and retry if ETXTBSY
is
       encountered.  Linux treats it as a hard error and returns immediately.

       Traditionally, the functions execlp() and execvp() ignored  all
errors
       except  for  the  ones described above and ENOMEM and E2BIG, upon
which
       they returned.  They now return if any error other than  the  ones
de-
       scribed above occurs.

BUGS
       Before  glibc 2.24, execl() and execle() employed realloc(3)
internally
       and were consequently not async-signal-safe, in violation  of  the
re-
       quirements of POSIX.1.  This was fixed in glibc 2.24.

   Architecture-specific details
       On  sparc and sparc64, execv() is provided as a system call by the
ker-
       nel (with the prototype shown  above)  for  compatibility  with
SunOS.
       This  function is not employed by the execv() wrapper function on
those
       architectures.

SEE ALSO
       sh(1), execve(2), execveat(2),  fork(2),  ptrace(2),  fexecve(3),
sys-
       tem(3), environ(7)

COLOPHON
       This  page  is  part of release 5.05 of the Linux man-pages project.
A
       description of the project, information about reporting bugs,  and
the
       latest     version     of     this    page,    can    be    found
at
       https://www.kernel.org/doc/man-pages/.

GNU                               2019-08-02
EXEC(3)
$
```

```
$ man 2 execve
EXECVE(2)                  Linux Programmer's Manual
EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() executes the program referred to by pathname.  This causes
the
       program that is currently being run by the calling process  to  be
re-
       placed  with  a  new  program,  with newly initialized stack, heap,
and
       (initialized and uninitialized) data segments.

       pathname must be either a binary executable, or a script starting
with
       a line of the form:

           #!interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an  array  of argument strings passed to the new program.
By
       convention, the first of these strings (i.e., argv[0])  should
contain
       the filename associated with the file being executed.  envp is an
array
       of strings, conventionally of the form key=value, which are  passed
as
       environment to the new program.  The argv and envp arrays must each
in-
       clude a null pointer at the end of the array.

       The argument vector and environment can be accessed by the called
pro-
       gram's main function, when it is defined as:

           int main(int argc, char *argv[], char *envp[])

       Note, however, that the use of a third argument to the main function
is
       not specified in POSIX.1; according to POSIX.1, the environment
should
       be accessed via the external variable environ(7).

       execve()  does  not  return on success, and the text, initialized
data,
       uninitialized data (bss), and stack of the calling  process  are
over-
       written according to the contents of the newly loaded program.

       If the current program is being ptraced, a SIGTRAP signal is sent to
it
       after a successful execve().

       If the set-user-ID bit is set on the program file referred to by
path-
       name,  then  the effective user ID of the calling process is changed
to
       that of the owner of the program file.  Similarly, when the  set-
group-
       ID bit of the program file is set the effective group ID of the
calling
       process is set to the group of the program file.

       The aforementioned transformations of the effective IDs  are  not
per-
       formed (i.e., the set-user-ID and set-group-ID bits are ignored) if
any
       of the following is true:

       *  the no_new_privs attribute  is  set  for  the  calling  thread
(see
          prctl(2));
```

```
       *  the  underlying filesystem is mounted nosuid (the MS_NOSUID flag
for
          mount(2)); or

       *  the calling process is being ptraced.

       The capabilities of the program file (see capabilities(7)) are also
ig-
       nored if any of the above are true.

       The  effective  user ID of the process is copied to the saved set-
user-
       ID; similarly, the effective group ID is copied to the saved set-
group-
       ID.  This copying takes place after any effective ID changes that
occur
       because of the set-user-ID and set-group-ID mode bits.

       The process's real UID and real GID, as well  its  supplementary
group
       IDs, are unchanged by a call to execve().

       If the executable is an a.out dynamically linked binary executable
con-
       taining shared-library stubs, the  Linux  dynamic  linker  ld.so(8)
is
       called  at  the  start of execution to bring needed shared objects
into
       memory and link the executable with them.

       If the executable is a dynamically linked ELF  executable,  the
inter-
       preter named in the PT_INTERP segment is used to load the needed
shared
       objects.  This interpreter is typically /lib/ld-linux.so.2 for
binaries
       linked with glibc (see ld-linux.so(8)).

       All  process  attributes  are  preserved during an execve(), except
the
       following:

       *  The dispositions of any signals that are being caught are  reset
to
          the default (signal(7)).

       *  Any alternate signal stack is not preserved (sigaltstack(2)).

       *  Memory mappings are not preserved (mmap(2)).

       *  Attached System V shared memory segments are detached (shmat(2)).

       *  POSIX shared memory regions are unmapped (shm_open(3)).

       *  Open POSIX message queue descriptors are closed (mq_overview(7)).

       *  Any open POSIX named semaphores are closed (sem_overview(7)).

       *  POSIX timers are not preserved (timer_create(2)).

       *  Any open directory streams are closed (opendir(3)).

       *  Memory locks are not preserved (mlock(2), mlockall(2)).

       *  Exit handlers are not preserved (atexit(3), on_exit(3)).

       *  The   floating-point  environment  is  reset  to  the  default
(see
          fenv(3)).

       The process attributes in the  preceding  list  are  all  specified
in
       POSIX.1.   The following Linux-specific process attributes are also
not
       preserved during an execve():

       *  The prctl(2) PR_SET_DUMPABLE flag is set, unless  a  set-user-ID
or
          set-group ID program is being executed, in which case it is
cleared.

       *  The prctl(2) PR_SET_KEEPCAPS flag is cleared.

       *  (Since  Linux 2.4.36 / 2.6.23) If a set-user-ID or set-group-ID
pro-
```

```
          gram is being executed, then the parent death signal set by
prctl(2)
          PR_SET_PDEATHSIG flag is cleared.

       *  The  process  name, as set by prctl(2) PR_SET_NAME (and displayed
by
          ps -o comm), is reset to the name of the new executable file.

       *  The SECBIT_KEEP_CAPS securebits  flag  is  cleared.   See
capabili-
          ties(7).

       *  The termination signal is reset to SIGCHLD (see clone(2)).

       *  The  file  descriptor  table  is unshared, undoing the effect of
the
          CLONE_FILES flag of clone(2).

       Note the following further points:

       *  All threads other than the calling thread are  destroyed  during
an
          execve().   Mutexes, condition variables, and other pthreads
objects
          are not preserved.

       *  The equivalent of setlocale(LC_ALL,  "C")  is  executed  at
program
          start-up.

       *  POSIX.1  specifies that the dispositions of any signals that are
ig-
          nored or set to the default are left unchanged.   POSIX.1
specifies
          one  exception:  if SIGCHLD is being ignored, then an
implementation
          may leave the disposition unchanged or  reset  it  to  the
default;
          Linux does the former.

       *  Any   outstanding   asynchronous   I/O   operations   are
canceled
          (aio_read(3), aio_write(3)).

       *  For the handling of  capabilities  during  execve(),  see
capabili-
          ties(7).

       *  By  default,  file descriptors remain open across an execve().
File
          descriptors that are marked close-on-exec are closed;  see  the
de-
          scription  of  FD_CLOEXEC  in  fcntl(2).   (If  a file descriptor
is
          closed, this will cause the release of all record locks obtained
on
          the  underlying  file  by  this process.  See fcntl(2) for
details.)
          POSIX.1 says that if file descriptors 0, 1, and 2 would otherwise
be
          closed after a successful execve(), and the process would gain
priv-
          ilege because the set-user-ID or set-group_ID mode bit  was  set
on
          the  executed file, then the system may open an unspecified file
for
          each of these file descriptors.  As a general principle, no
portable
          program, whether privileged or not, can assume that these three
file
          descriptors will remain closed across an execve().

   Interpreter scripts
       An interpreter script is a text file that has  execute  permission
en-
       abled and whose first line is of the form:

           #!interpreter [optional-arg]

       The interpreter must be a valid pathname for an executable file.

       If  the  pathname argument of execve() specifies an interpreter
script,
       then interpreter will be invoked with the following arguments:

           interpreter [optional-arg] pathname arg...
```

```
       where pathname is the absolute pathname of the file  specified  as
the
       first  argument of execve(), and arg...  is the series of words
pointed
       to by the argv argument of execve(), starting at  argv[1].   Note
that
       there  is  no  way  to  get the argv[0] that was passed to the
execve()
       call.

       For portable use, optional-arg should either be absent, or be
specified
       as  a  single word (i.e., it should not contain white space); see
NOTES
       below.

       Since Linux 2.6.28, the kernel permits the interpreter of a  script
to
       itself  be  a  script.   This permission is recursive, up to a limit
of
       four recursions, so that the interpreter may be a script which  is
in-
       terpreted by a script, and so on.

   Limits on size of arguments and environment
       Most  UNIX  implementations  impose some limit on the total size of
the
       command-line argument (argv) and environment (envp) strings that may
be
       passed to a new program.  POSIX.1 allows an implementation to
advertise
       this limit using the ARG_MAX constant (either defined in <limits.h>
or
       available at run time using the call sysconf(_SC_ARG_MAX)).

       On  Linux prior to kernel 2.6.23, the memory used to store the
environ-
       ment and argument strings was limited to 32 pages (defined by the
ker-
       nel  constant  MAX_ARG_PAGES).  On architectures with a 4-kB page
size,
       this yields a maximum size of 128 kB.

       On kernel 2.6.23 and later, most architectures support a size limit
de-
       rived from the soft RLIMIT_STACK resource limit (see getrlimit(2))
that
       is in force at the time of the execve() call.  (Architectures  with
no
       memory  management  unit are excepted: they maintain the limit that
was
       in effect before kernel 2.6.23.)  This change allows programs to have
a
       much larger argument and/or environment list.  For these
architectures,
       the total size is limited to 1/4 of the allowed stack size.
(Imposing
       the  1/4-limit  ensures  that  the  new  program  always has some
stack
       space.)  Additionally, the total size is limited to 3/4 of the value
of
       the  kernel  constant  _STK_LIM (8 Mibibytes).  Since Linux 2.6.25,
the
       kernel also places a floor of 32 pages on this  size  limit,  so
that,
       even  when RLIMIT_STACK is set very low, applications are guaranteed
to
       have at least as much argument and environment space as was provided
by
       Linux  2.6.23  and  earlier.  (This guarantee was not provided in
Linux
       2.6.23 and 2.6.24.)  Additionally, the limit per  string  is  32
pages
       (the kernel constant MAX_ARG_STRLEN), and the maximum number of
strings
       is 0x7FFFFFFF.

RETURN VALUE
       On success, execve() does not return, on error -1 is returned, and
er-
       rno is set appropriately.

ERRORS
       E2BIG  The total number of bytes in the environment (envp) and
argument
```

```
              list (argv) is too large.

       EACCES Search permission is denied on a component of the path prefix
of
              pathname  or  the  name  of  a  script  interpreter.   (See
also
              path_resolution(7).)

       EACCES The file or a script interpreter is not a regular file.

       EACCES Execute permission is denied for the file or a script or ELF
in-
              terpreter.

       EACCES The filesystem is mounted noexec.

       EAGAIN (since Linux 3.1)
              Having  changed  its  real UID using one of the set*uid()
calls,
              the caller was—and is now still—above its RLIMIT_NPROC
resource
              limit  (see  setrlimit(2)).   For a more detailed explanation
of
              this error, see NOTES.

       EFAULT pathname or one of the pointers in  the  vectors  argv  or
envp
              points outside your accessible address space.

       EINVAL An  ELF  executable  had  more than one PT_INTERP segment
(i.e.,
              tried to name more than one interpreter).

       EIO    An I/O error occurred.

       EISDIR An ELF interpreter was a directory.

       ELIBBAD
              An ELF interpreter was not in a recognized format.

       ELOOP  Too many symbolic links were encountered in  resolving
pathname
              or the name of a script or ELF interpreter.

       ELOOP  The  maximum recursion limit was reached during recursive
script
              interpretation (see "Interpreter scripts", above).  Before
Linux
              3.8, the error produced for this case was ENOEXEC.

       EMFILE The per-process limit on the number of open file descriptors
has
              been reached.

       ENAMETOOLONG
              pathname is too long.

       ENFILE The system-wide limit on the total number of open files has
been
              reached.

       ENOENT The file pathname or a script or ELF interpreter does not
exist,
              or a shared library needed for the file or interpreter cannot
be
              found.

       ENOEXEC
              An  executable  is  not in a recognized format, is for the
wrong
              architecture, or has some other format error that means it
can-
              not be executed.

       ENOMEM Insufficient kernel memory was available.

       ENOTDIR
              A  component  of  the path prefix of pathname or a script or
ELF
              interpreter is not a directory.

       EPERM  The filesystem is mounted nosuid, the user is not the
superuser,
              and the file has the set-user-ID or set-group-ID bit set.

       EPERM  The  process  is being traced, the user is not the superuser
```

```
and
              the file has the set-user-ID or set-group-ID bit set.

       EPERM  A "capability-dumb" applications would not obtain the  full
set
              of  permitted  capabilities granted by the executable file.
See
              capabilities(7).

       ETXTBSY
              The specified executable was open for writing  by  one  or
more
              processes.

CONFORMING TO
       POSIX.1-2001,  POSIX.1-2008, SVr4, 4.3BSD.  POSIX does not document
the
       #! behavior, but it exists (with some variations) on  other  UNIX
sys-
       tems.

NOTES
       One  sometimes  sees  execve()  (and the related functions described
in
       exec(3)) described as "executing a new process" (or similar).  This
is
       a  highly  misleading  description:  there  is no new process; many
at-
       tributes of the calling process remain unchanged  (in  particular,
its
       PID).   All  that execve() does is arrange for an existing process
(the
       calling process) to execute a new program.

       Set-user-ID and set-group-ID processes can not be ptrace(2)d.

       The result of mounting a filesystem nosuid varies across  Linux
kernel
       versions:  some  will  refuse execution of set-user-ID and set-group-
ID
       executables when this would give the user powers they did not have
al-
       ready  (and  return  EPERM),  some will just ignore the set-user-ID
and
       set-group-ID bits and exec() successfully.

       On Linux, argv and envp can be specified as NULL.  In both cases,
this
       has  the  same effect as specifying the argument as a pointer to a
list
       containing a single null pointer.  Do not take advantage of  this
non-
       standard and nonportable misfeature!  On many other UNIX systems,
spec-
       ifying argv as NULL will result in an error (EFAULT).  Some other
UNIX
       systems treat the envp==NULL case the same as Linux.

       POSIX.1  says  that  values  returned by sysconf(3) should be
invariant
       over the lifetime of a process.  However, since Linux  2.6.23,  if
the
       RLIMIT_STACK  resource  limit  changes,  then  the  value  reported
by
       _SC_ARG_MAX will also change, to reflect the fact  that  the  limit
on
       space  for holding command-line arguments and environment variables
has
       changed.

       In most cases where execve() fails, control returns to the original
ex-
       ecutable  image,  and the caller of execve() can then handle the
error.
       However, in (rare) cases (typically  caused  by  resource
exhaustion),
       failure  may occur past the point of no return: the original
executable
       image has been torn down, but the new image  could  not  be
completely
       built.  In such cases, the kernel kills the process with a SIGKILL
sig-
       nal.

   Interpreter scripts
       The kernel imposes a maximum length on the text that follows  the
```

```
"#!"
       characters  at  the  start of a script; characters beyond the limit
are
       ignored.  Before Linux 5.1, the limit is 127 characters.   Since
Linux
       5.1, the limit is 255 characters.

       The  semantics  of  the  optional-arg argument of an interpreter
script
       vary across implementations.  On Linux, the entire string following
the
       interpreter name is passed as a single argument to the interpreter,
and
       this string can include white space.  However, behavior differs on
some
       other systems.  Some systems use the first white space to terminate
op-
       tional-arg.  On some systems, an interpreter script can  have
multiple
       arguments, and white spaces in optional-arg are used to delimit the
ar-
       guments.

       Linux (like most other modern UNIX systems) ignores the set-user-ID
and
       set-group-ID bits on scripts.

   execve() and EAGAIN
       A  more  detailed explanation of the EAGAIN error that can occur
(since
       Linux 3.1) when calling execve() is as follows.

       The EAGAIN error can occur when a  preceding  call  to  setuid(2),
se-
       treuid(2),  or  setresuid(2)  caused the real user ID of the process
to
       change, and that change caused the process to exceed  its
RLIMIT_NPROC
       resource limit (i.e., the number of processes belonging to the new
real
       UID exceeds the resource limit).  From Linux 2.6.0 to 3.0, this
caused
       the  set*uid() call to fail.  (Prior to 2.6, the resource limit was
not
       imposed on processes that changed their user IDs.)

       Since Linux 3.1, the scenario  just  described  no  longer  causes
the
       set*uid()  call  to  fail,  because  it too often led to security
holes
       where buggy applications didn't check the  return  status  and
assumed
       that—if  the  caller had root privileges—the call would always
succeed.
       Instead, the set*uid() calls now successfully change the real UID,
but
       the kernel sets an internal flag, named PF_NPROC_EXCEEDED, to note
that
       the RLIMIT_NPROC resource limit has been exceeded.  If the
PF_NPROC_EX-
       CEEDED flag is set and the resource limit is still exceeded at the
time
       of a subsequent execve() call, that call fails with the  error
EAGAIN.
       This kernel logic ensures that the RLIMIT_NPROC resource limit is
still
       enforced for the common privileged daemon  workflow—namely,  fork(2)
+
       set*uid() + execve().

       If  the  resource  limit  was not still exceeded at the time of the
ex-
       ecve() call (because other processes belonging to this real UID
termi-
       nated  between  the set*uid() call and the execve() call), then the
ex-
       ecve() call  succeeds  and  the  kernel  clears  the
PF_NPROC_EXCEEDED
       process flag.  The flag is also cleared if a subsequent call to
fork(2)
       by this process succeeds.

   Historical
       With UNIX V6, the argument list of an exec() call was ended by 0,
while
       the  argument  list  of main was ended by -1.  Thus, this argument
```

```
list
       was not directly usable in a further exec() call.  Since UNIX V7,
both
       are NULL.

EXAMPLE
       The  following  program  is designed to be execed by the second
program
       below.  It just echoes its command-line arguments, one per line.

           /* myecho.c */

           #include <stdio.h>
           #include <stdlib.h>

           int
           main(int argc, char *argv[])
           {
               int j;

               for (j = 0; j < argc; j++)
                   printf("argv[%d]: %s\n", j, argv[j]);

               exit(EXIT_SUCCESS);
           }

       This program can be used to exec the program named in its  command-
line
       argument:

           /* execve.c */

           #include <stdio.h>
           #include <stdlib.h>
           #include <unistd.h>

           int
           main(int argc, char *argv[])
           {
               char *newargv[] = { NULL, "hello", "world", NULL };
               char *newenviron[] = { NULL };

               if (argc != 2) {
                   fprintf(stderr, "Usage: %s <file-to-exec>\n", argv[0]);
                   exit(EXIT_FAILURE);
               }

               newargv[0] = argv[1];

               execve(argv[1], newargv, newenviron);
               perror("execve");   /* execve() returns only on error */
               exit(EXIT_FAILURE);
           }

       We can use the second program to exec the first as follows:

           $ cc myecho.c -o myecho
           $ cc execve.c -o execve
           $ ./execve ./myecho
           argv[0]: ./myecho
           argv[1]: hello
           argv[2]: world

       We  can  also use these programs to demonstrate the use of a script
in-
       terpreter.  To do this we create a script whose  "interpreter"  is
our
       myecho program:

           $ cat > script
           #!./myecho script-arg
           ^D
           $ chmod +x script

       We can then use our program to exec the script:

           $ ./execve ./script
           argv[0]: ./myecho
           argv[1]: script-arg
           argv[2]: ./script
           argv[3]: hello
           argv[4]: world

SEE ALSO
       chmod(2), execveat(2), fork(2), get_robust_list(2), ptrace(2),
exec(3),
```

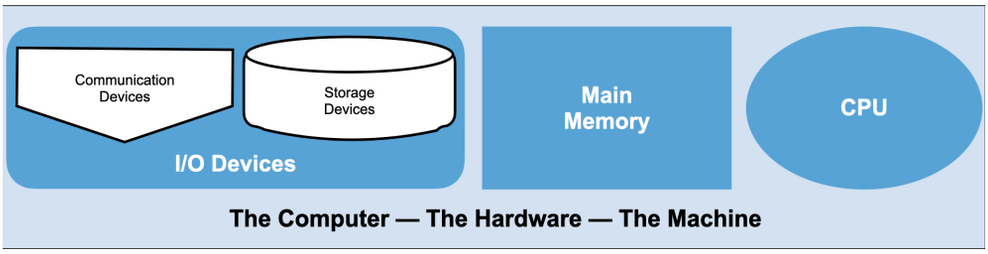
```
       fexecve(3), getopt(3), system(3), credentials(7), environ(7),
path_res-
       olution(7), ld.so(8)

COLOPHON
       This  page  is  part of release 5.05 of the Linux man-pages project.
A
       description of the project, information about reporting bugs,  and
the
       latest     version     of     this     page,     can     be     found
at
       https://www.kernel.org/doc/man-pages/.

Linux                              2019-10-10
EXECVE(2)
$
```

7.2. Executables as “Process Images”

Remember what the Hardware looks like.



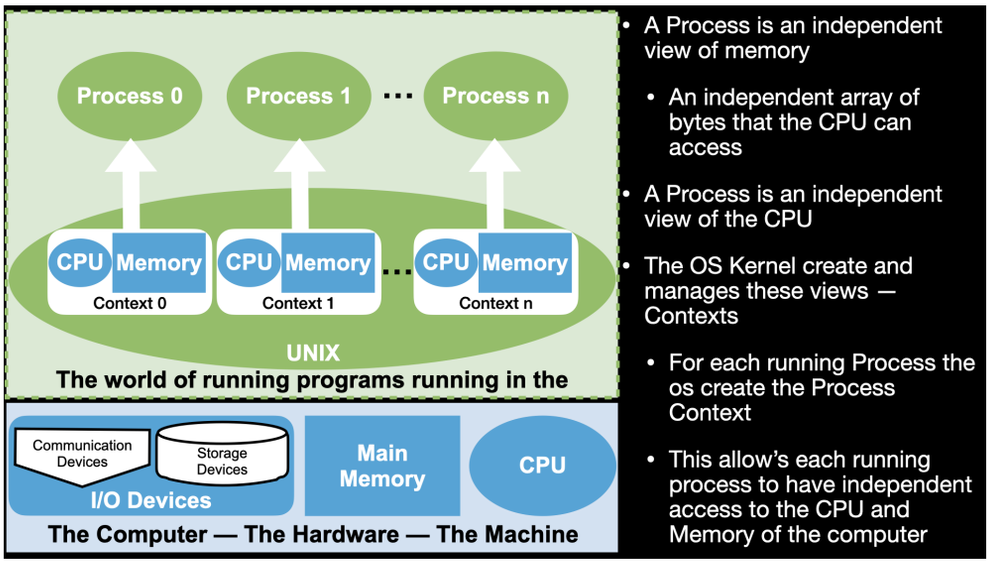
Remember that the OS Kernel is designed to make it easier to use the hardware to run programs.

Now we need to dig into this idea more carefully.

7.2.1. Processes As CPU and Memory context

A process is the way that the Operating System let our programs use the CPU and Memory in a controlled way.

Each Process is independent “Context” to execute a program. Where a context provides a program with its own view of Memory and the CPU



7.2.2. Process as a Context for using the CPU and Memory

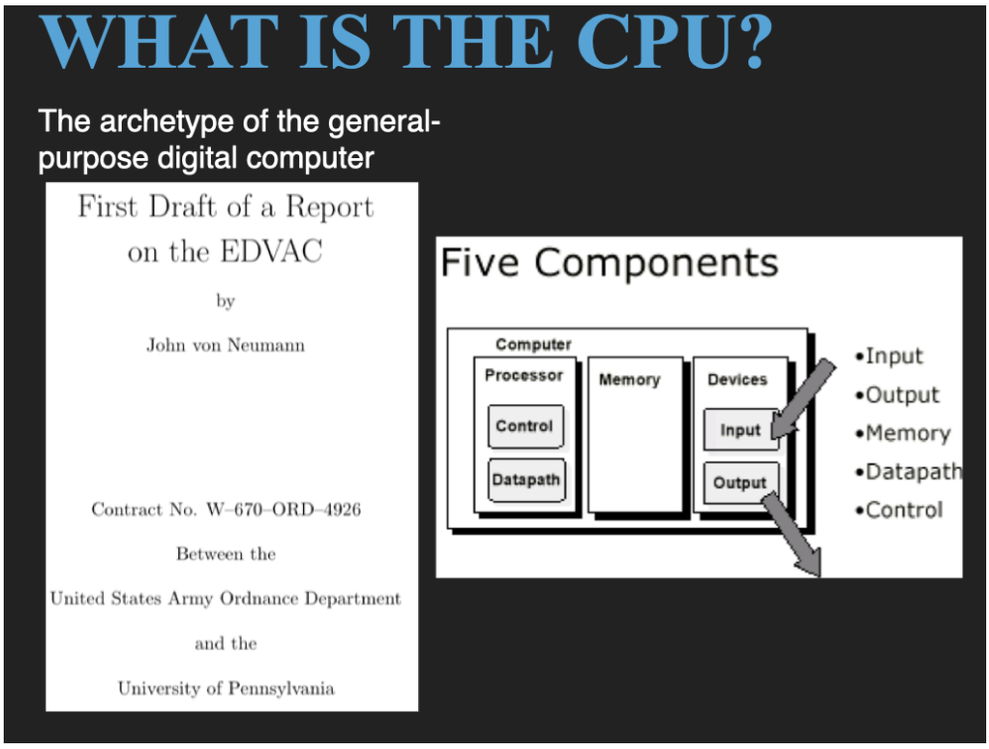
A process is a way for us to use the CPU and memory through the programs we write

- But not the I/O devices – Only the OS can directly access the I/O devices

- as we will see later the only way for our programs to do I/O will be make calls to the OS

To understand what we are doing when we write assembly code to create a program

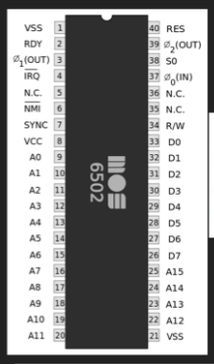
- we need to understand how the CPU works and Memory together as programmable system



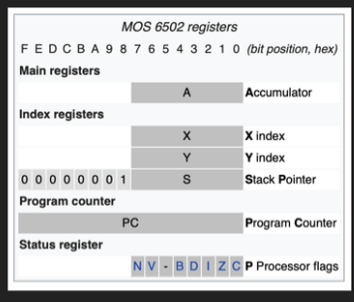
Let's start with a quick overview of the basic Von Neumman computer model and how the CPU and memory work together.

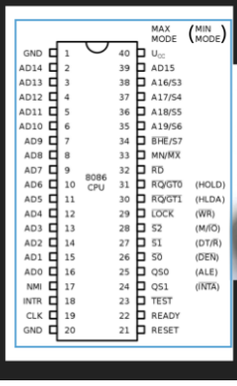
CPU

- Low-level : Answer a complex electrical device composed of transistors and wires
- High-level: A core building block for programmable information processing

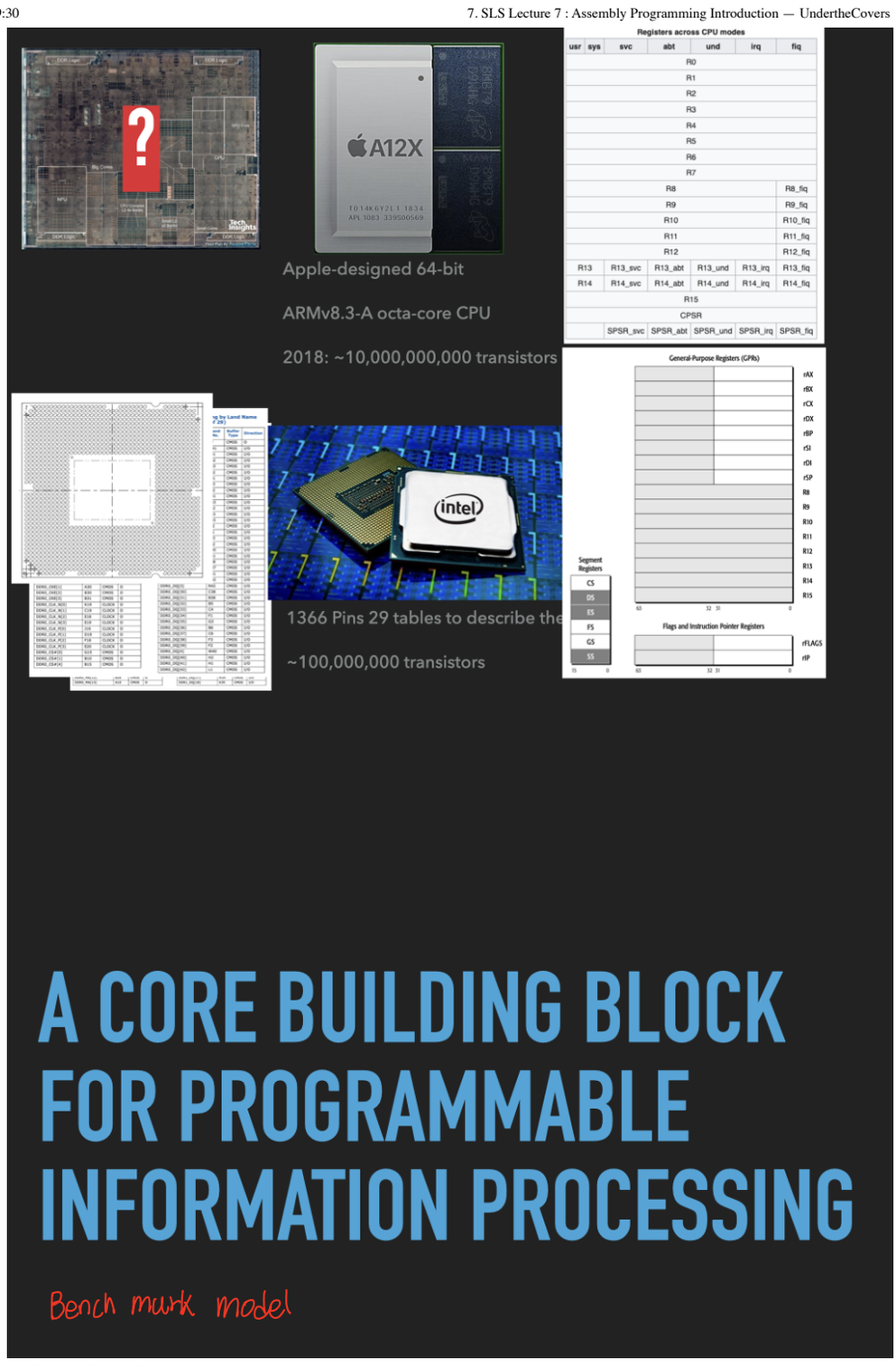


1975: 40 Pins ~4,500 transistors



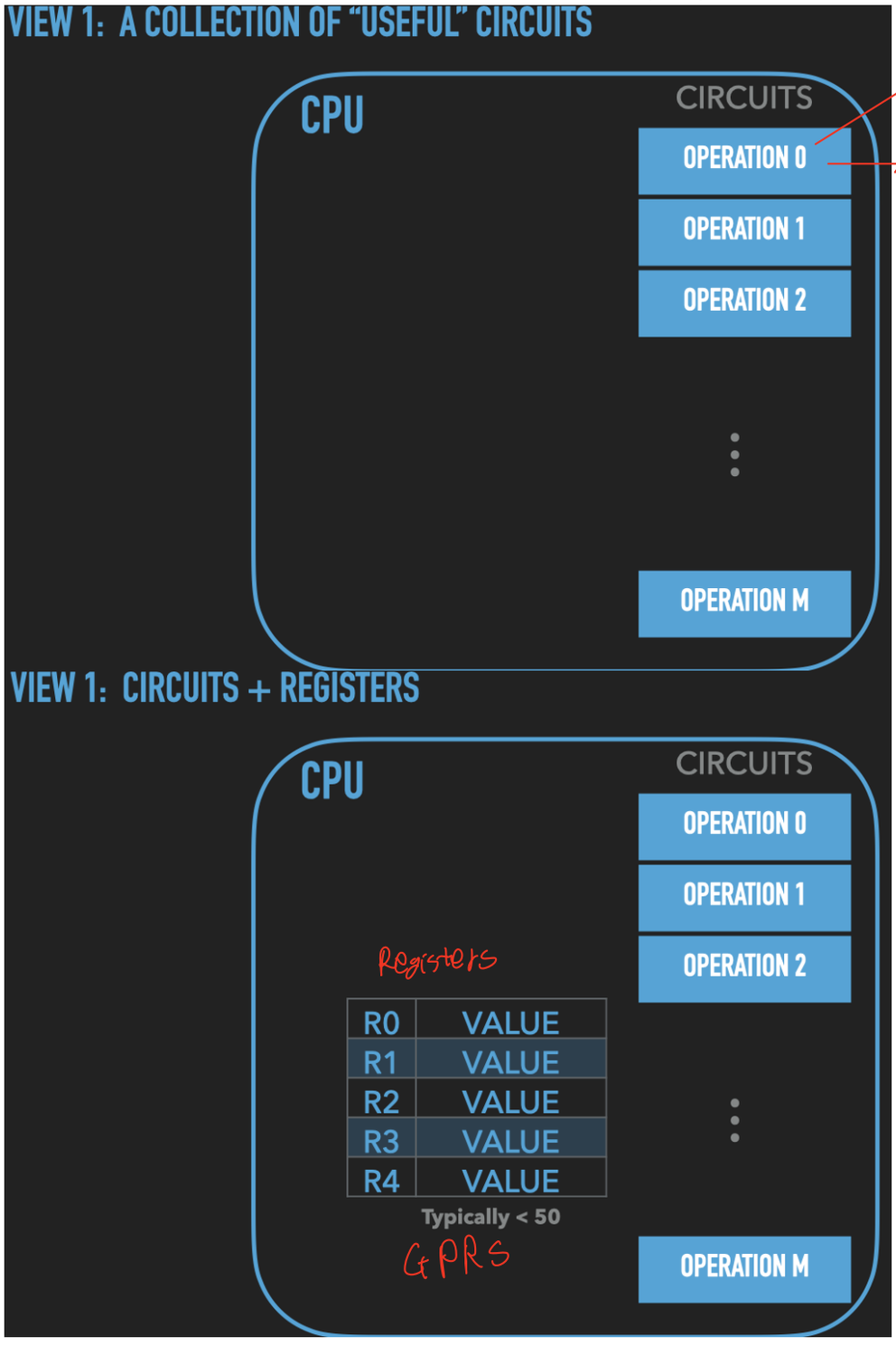


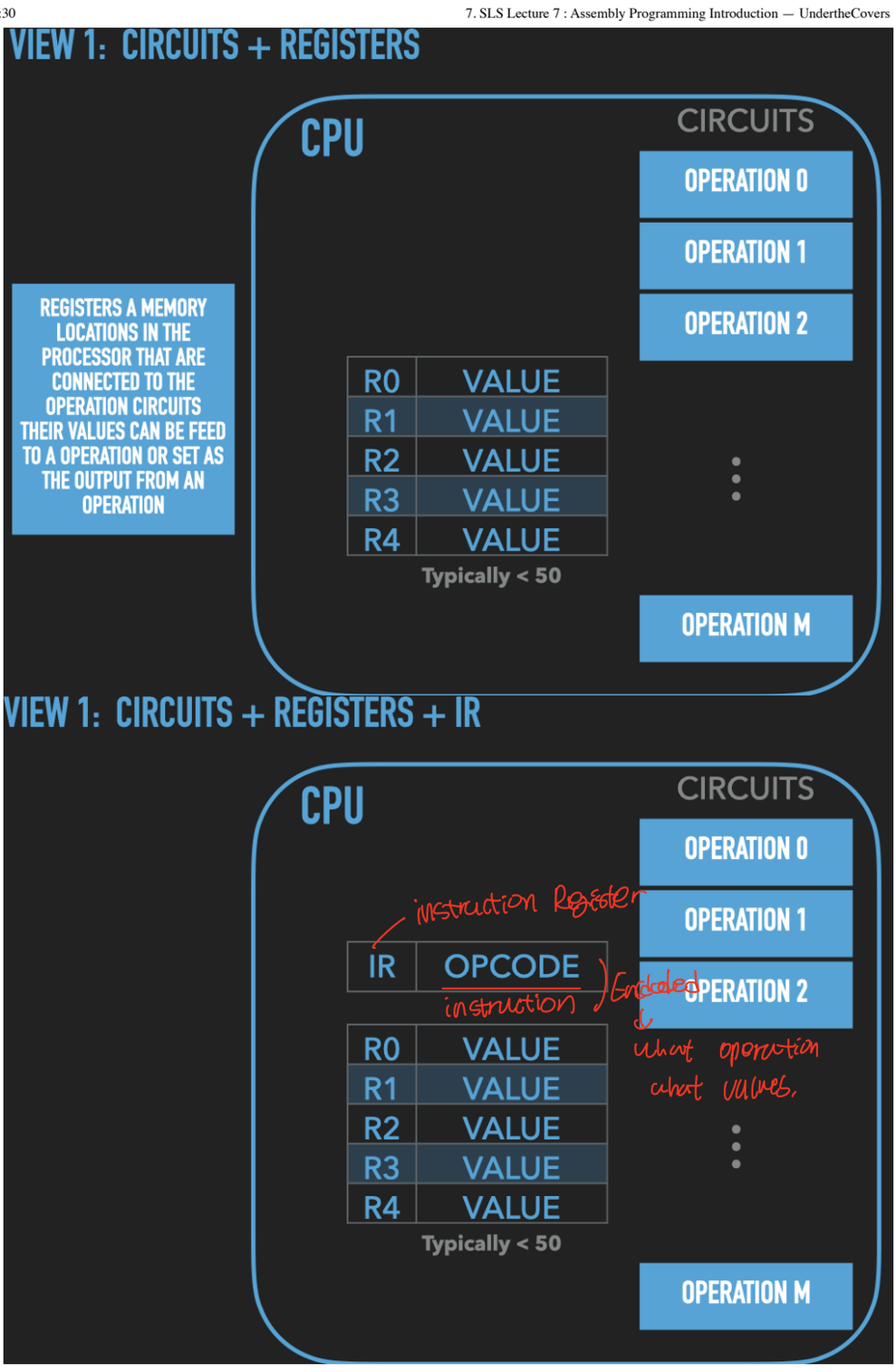
1978: Pins ~29,000 transistors

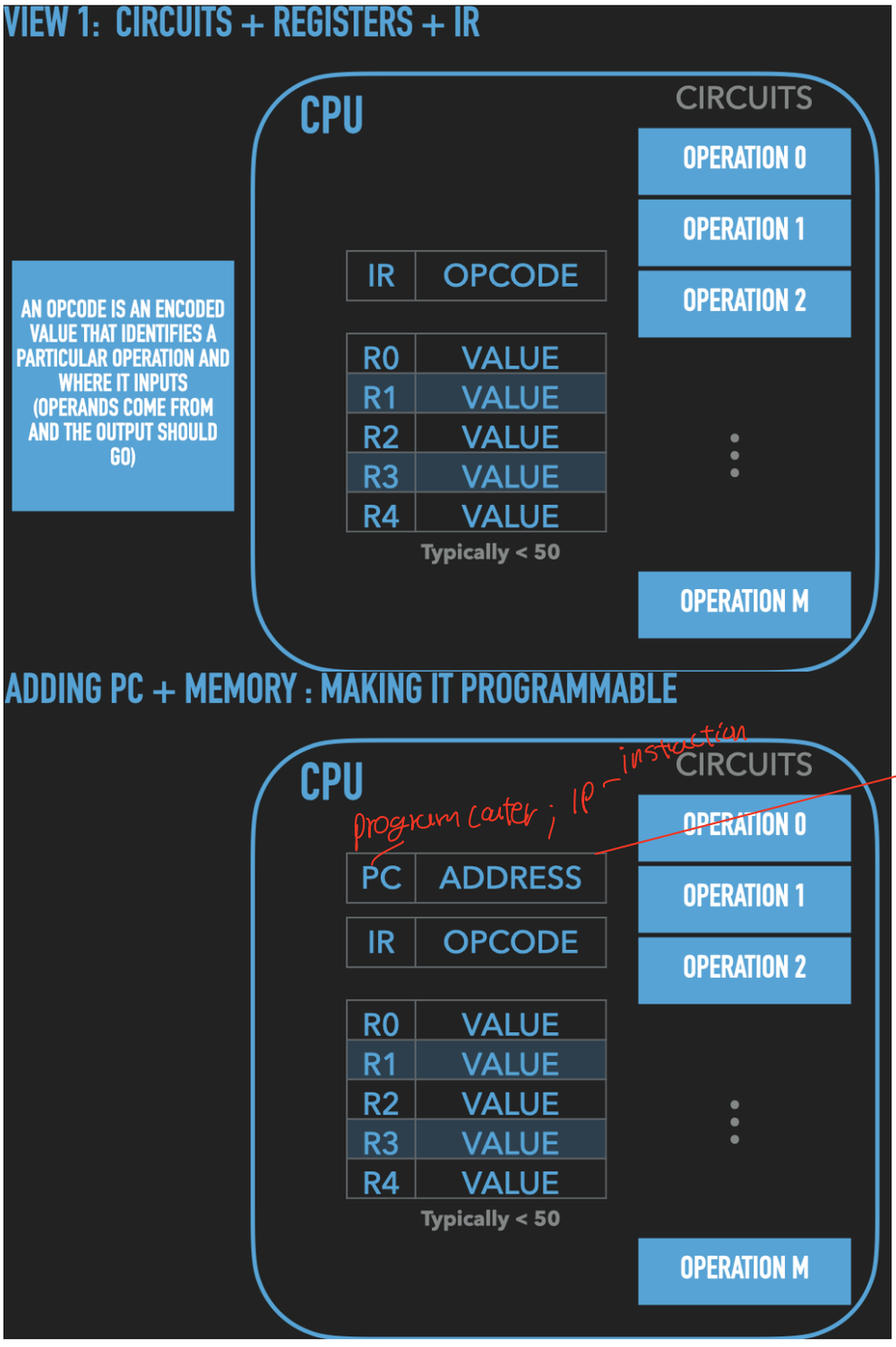


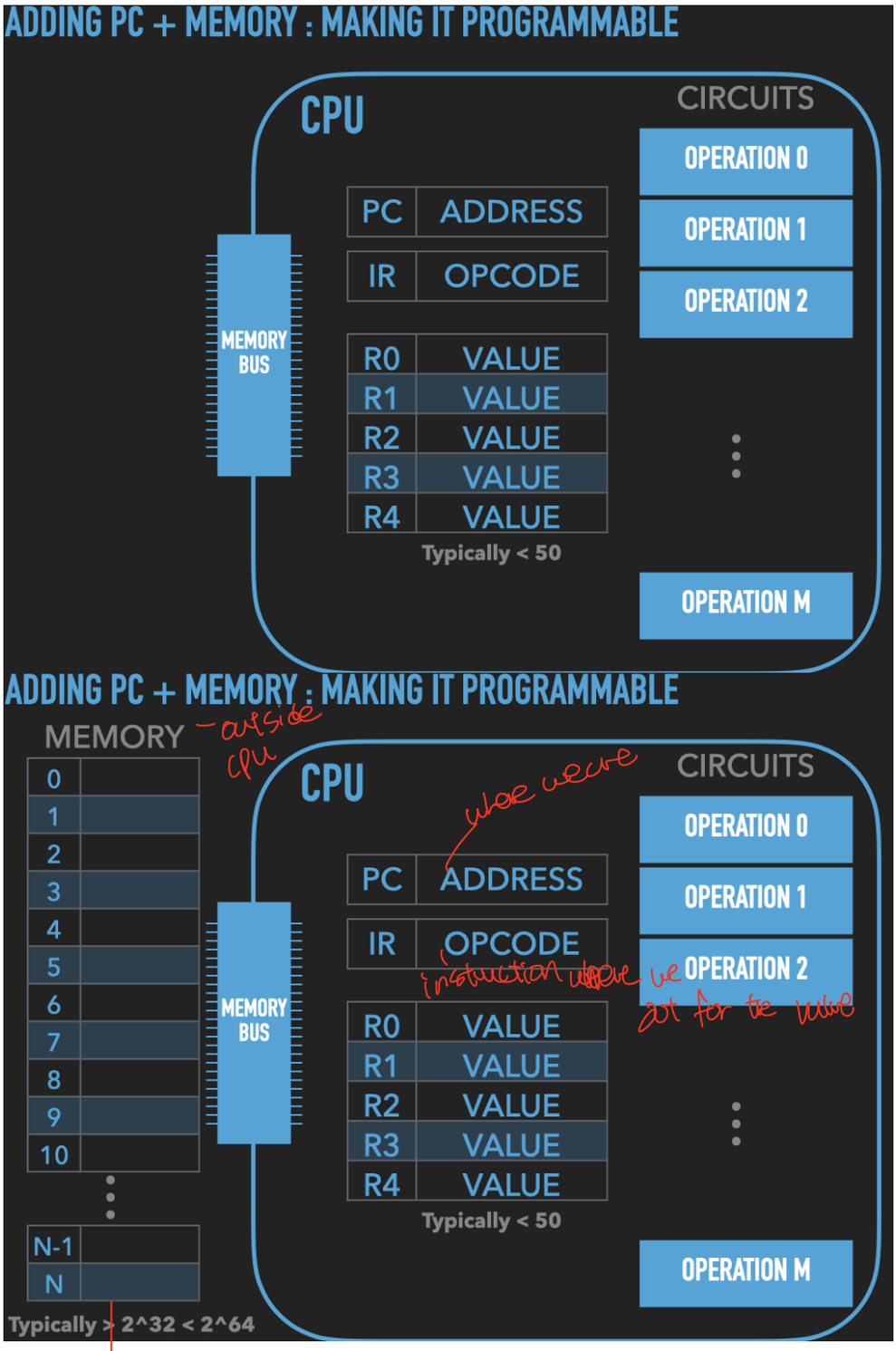
Or lets play with a computer

SOL6502


VIEW 1: A COLLECTION OF "USEFUL" CIRCUITS
CPU
CIRCUITS
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
pre-defined
add operation
VIEW 1: CIRCUITS + REGISTERS
CPU
CIRCUITS
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
Registers
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
GPRS


VIEW 1: CIRCUITS + REGISTERS
CPU
CIRCUITS
OPERATION 0
OPERATION 1
OPERATION 2
REGISTERS A MEMORY LOCATIONS IN THE PROCESSOR THAT ARE CONNECTED TO THE OPERATION CIRCUITS THEIR VALUES CAN BE FEED TO A OPERATION OR SET AS THE OUTPUT FROM AN OPERATION
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION M
VIEW 1: CIRCUITS + REGISTERS + IR
CPU
CIRCUITS
OPERATION 0
OPERATION 1
OPERATION 2
instruction Register
IR OPCODE
instruction
Encoded
what operation
what values.
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION M


VIEW 1: CIRCUITS + REGISTERS + IR
CPU
CIRCUITS
OPERATION 0
OPERATION 1
OPERATION 2
IR
OPCODE
AN OPCODE IS AN ENCODED VALUE THAT IDENTIFIES A PARTICULAR OPERATION AND WHERE IT INPUTS (OPERANDS COME FROM AND THE OUTPUT SHOULD GO)
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION M
ADDING PC + MEMORY : MAKING IT PROGRAMMABLE
CPU
CIRCUITS
program caiter ; IP - instruction
keep track of which instruction we are
PC
ADDRESS
OPERATION 0
OPERATION 1
OPERATION 2
IR
OPCODE
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION M

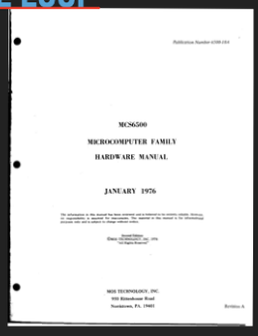


- outside CPU
- where we are
- instruction where we are at for the value
- Array format.

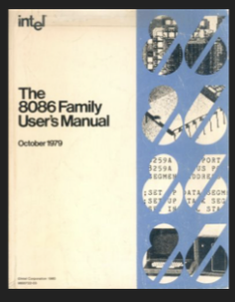
Address bus = PC

memory bus = IR

THE LOOP


MCS6500
MICROCOMPUTER FAMILY
HARDWARE MANUAL
JANUARY 1976

The processor causes the system to perform the desired operations by reading the first instruction in the program, and performing the very simple task dictated by the specific pattern of bits in this instruction (referred to as "executing" that instruction). It then goes on to the next instruction in the program and executes it. This simple operation of fetching an instruction and executing it is performed over and over, each time on the next instruction in sequence. In this way the program instructs the processor to bring about the desired system operation.



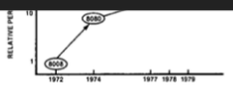


Figure 2-4. Relative Performance of the 8086 and 8088

2.2 Processor Architecture

Microprocessors generally execute a program by repeatedly cycling through the steps shown below (this description is somewhat simplified):

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).

2-3

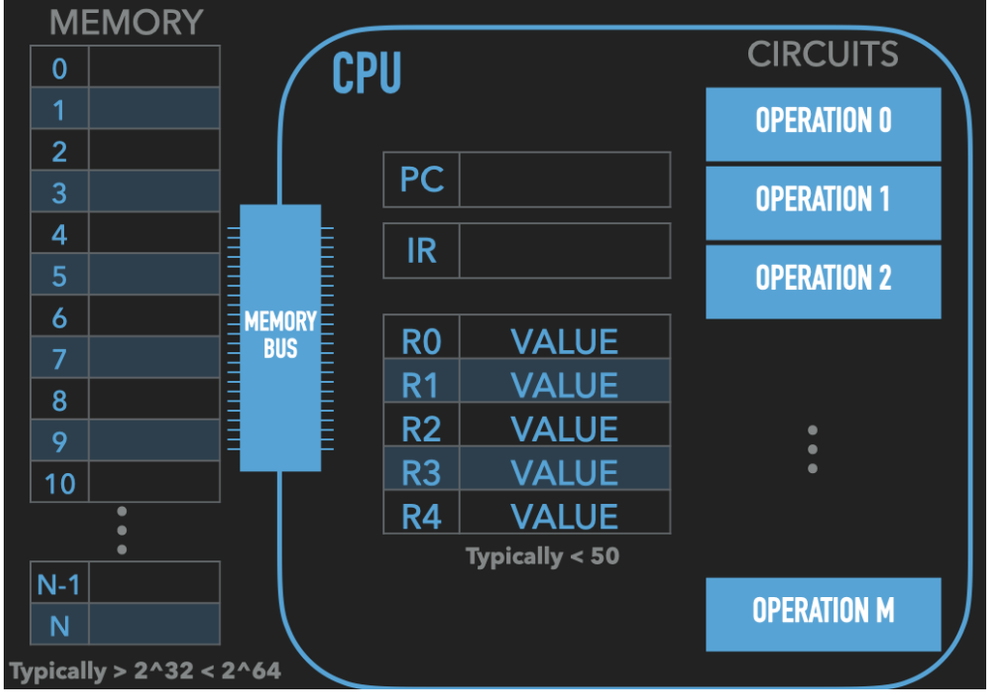
8086 AND 8088 CENTRAL PROCESSING UNITS

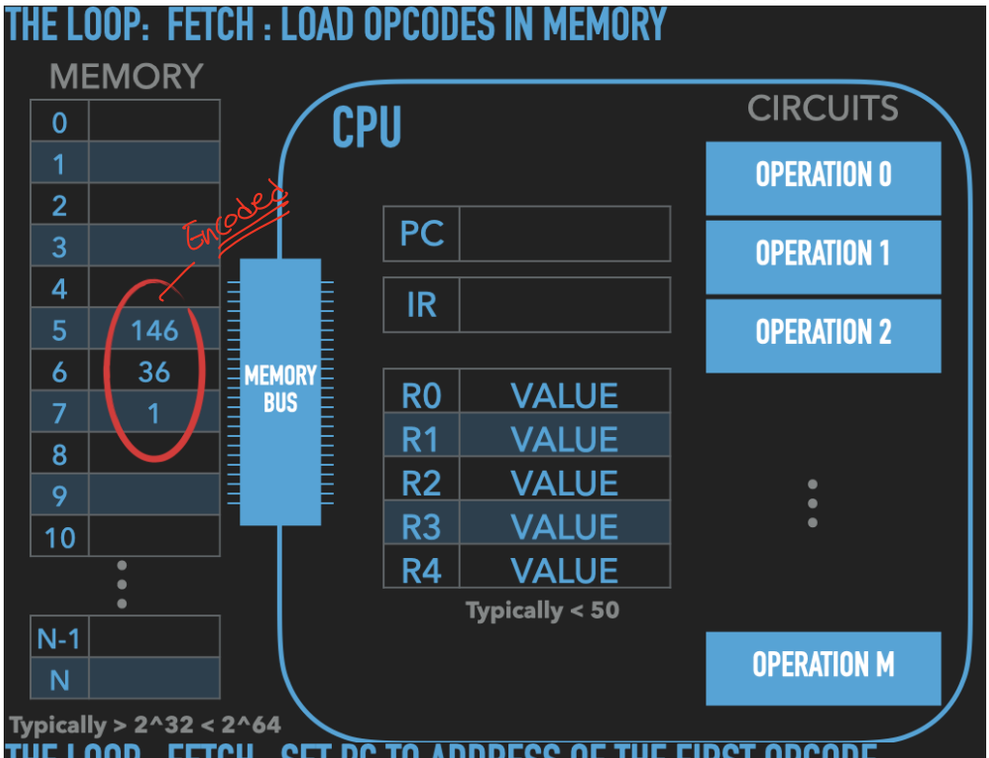
3. Execute the instruction.
4. Write the result (if required by the instruction).

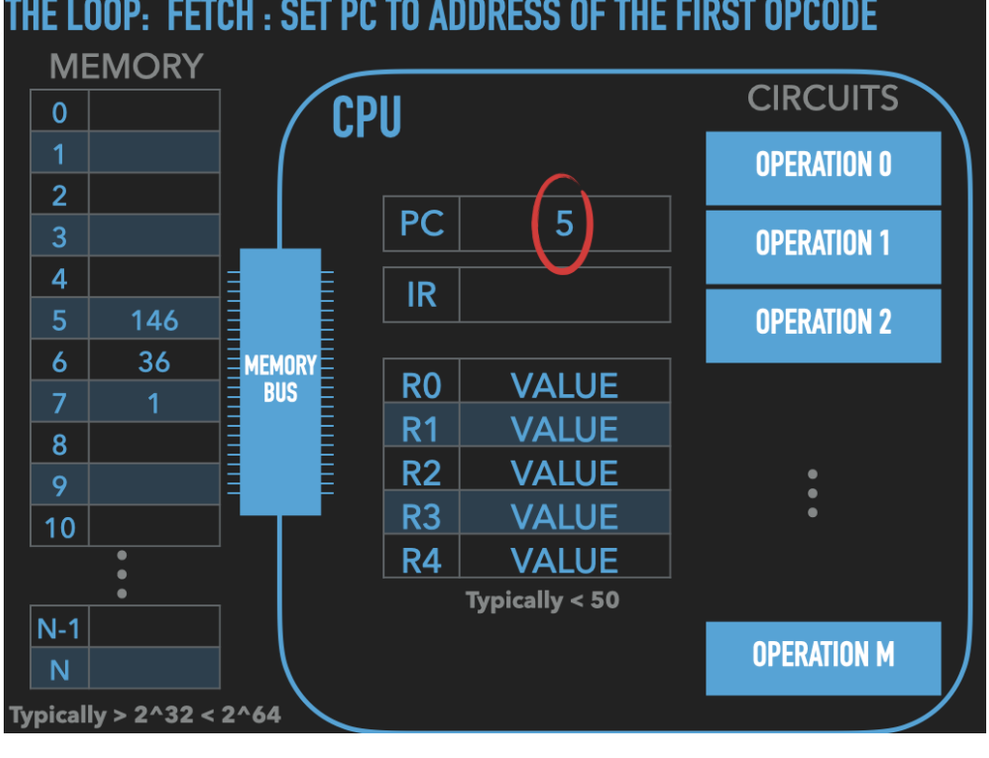
In previous CPUs, most of these steps have been performed serially, or with only a single bus cycle fetch overlap. The architecture of the 8086 and 8088 CPUs, while performing the same steps, allocates them to two separate processing units within the CPU. The execution unit (EU) executes instructions; the bus interface unit (BIU) fetches instructions, reads operands and writes results

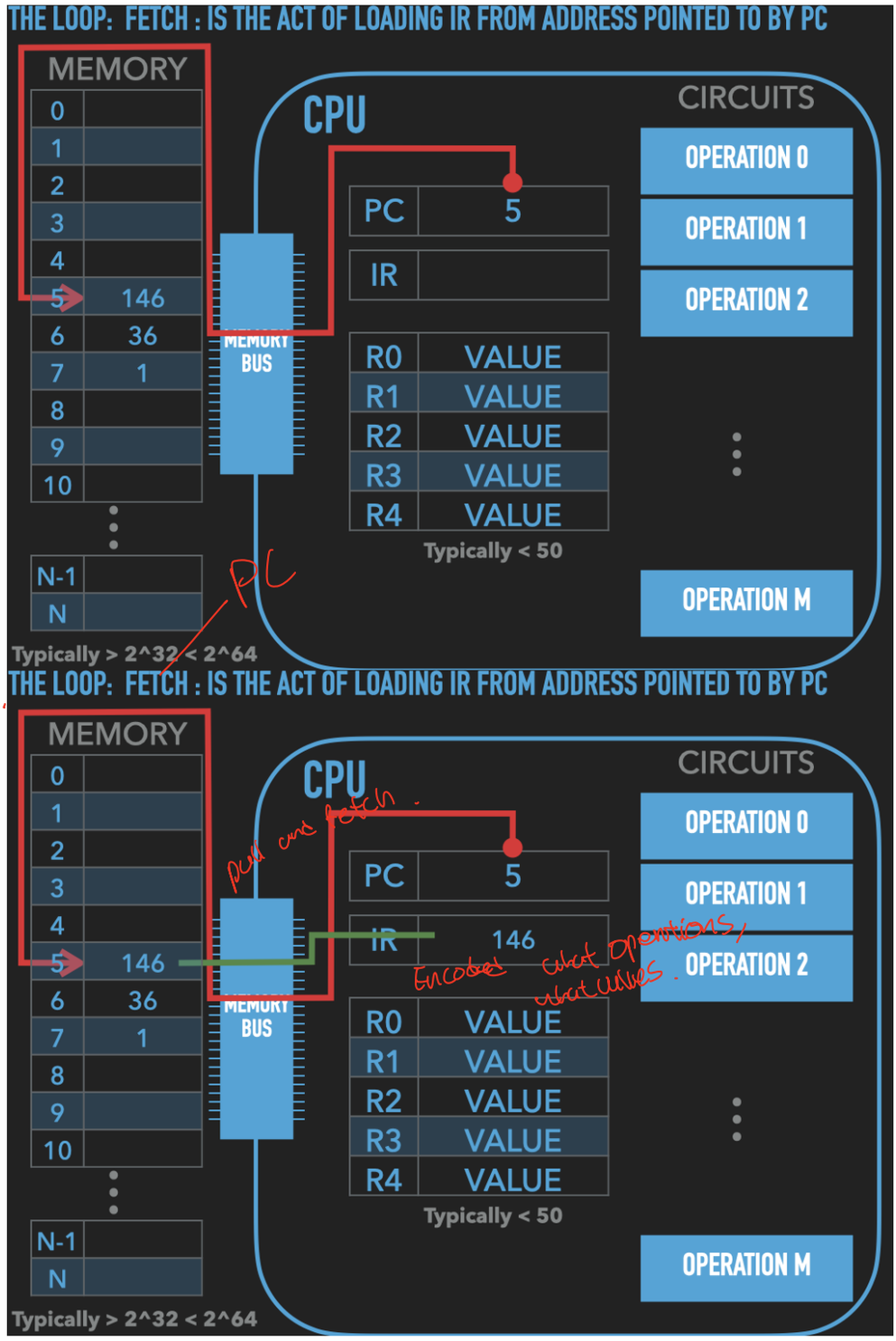
The two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by the BIU. Figure 2-5 illustrates this overlap and compares it with traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and allows two additional instructions to be prefetched as

1. THE LOOP: FETCH

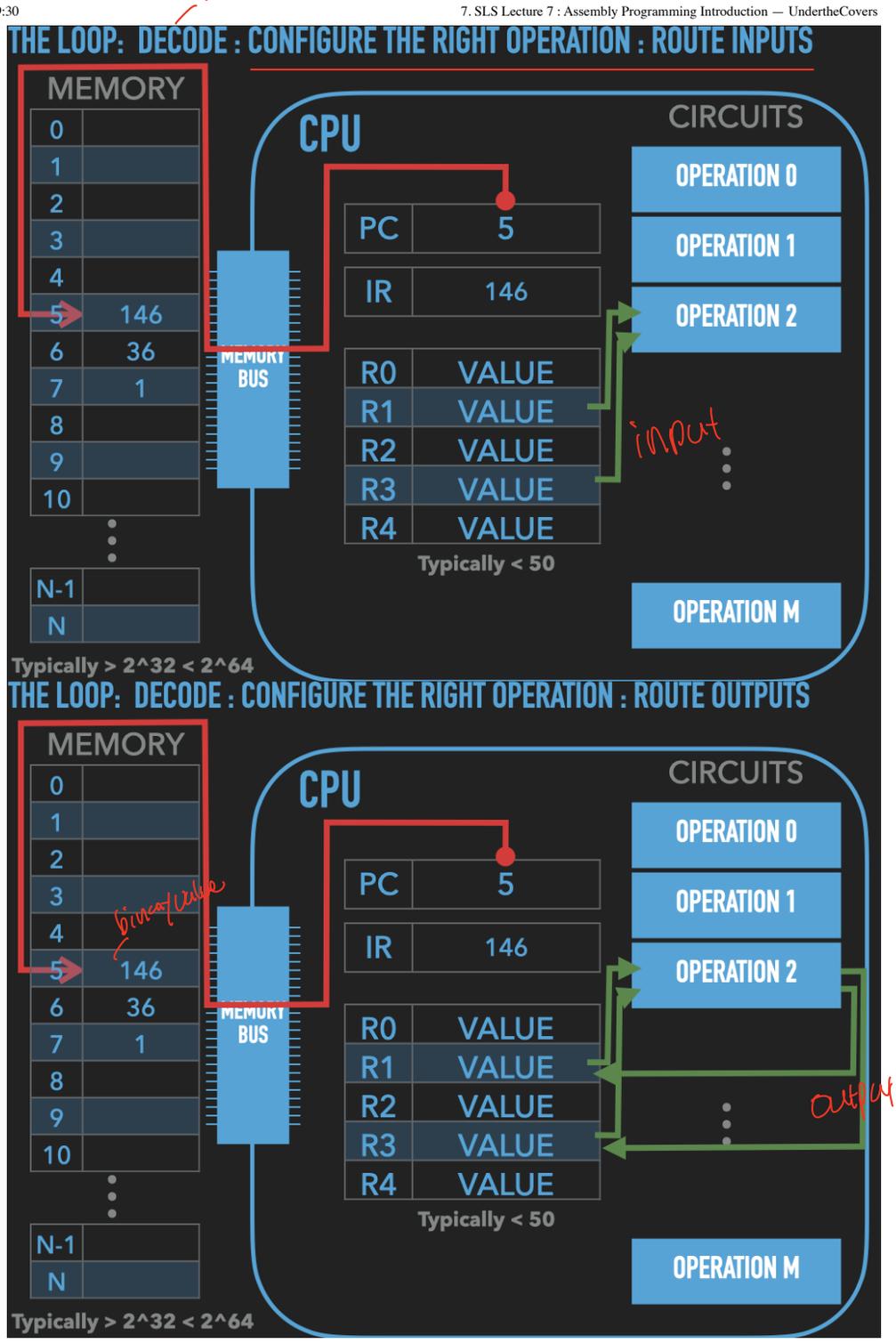



1.
THE LOOP: FETCH : LOAD OPCODES IN MEMORY
MEMORY
0
1
2
3
4
5 146
6 36
7 1
8
9
10
N-1
N
Encoded
MEMORY BUS
CPU
CIRCUITS
PC
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
Typically > 2^32 < 2^64


1.
THE LOOP: FETCH : SET PC TO ADDRESS OF THE FIRST OPCODE
MEMORY
0
1
2
3
4
5 146
6 36
7 1
8
9
10
N-1
N
MEMORY BUS
CPU
CIRCUITS
PC 5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
Typically > 2^32 < 2^64

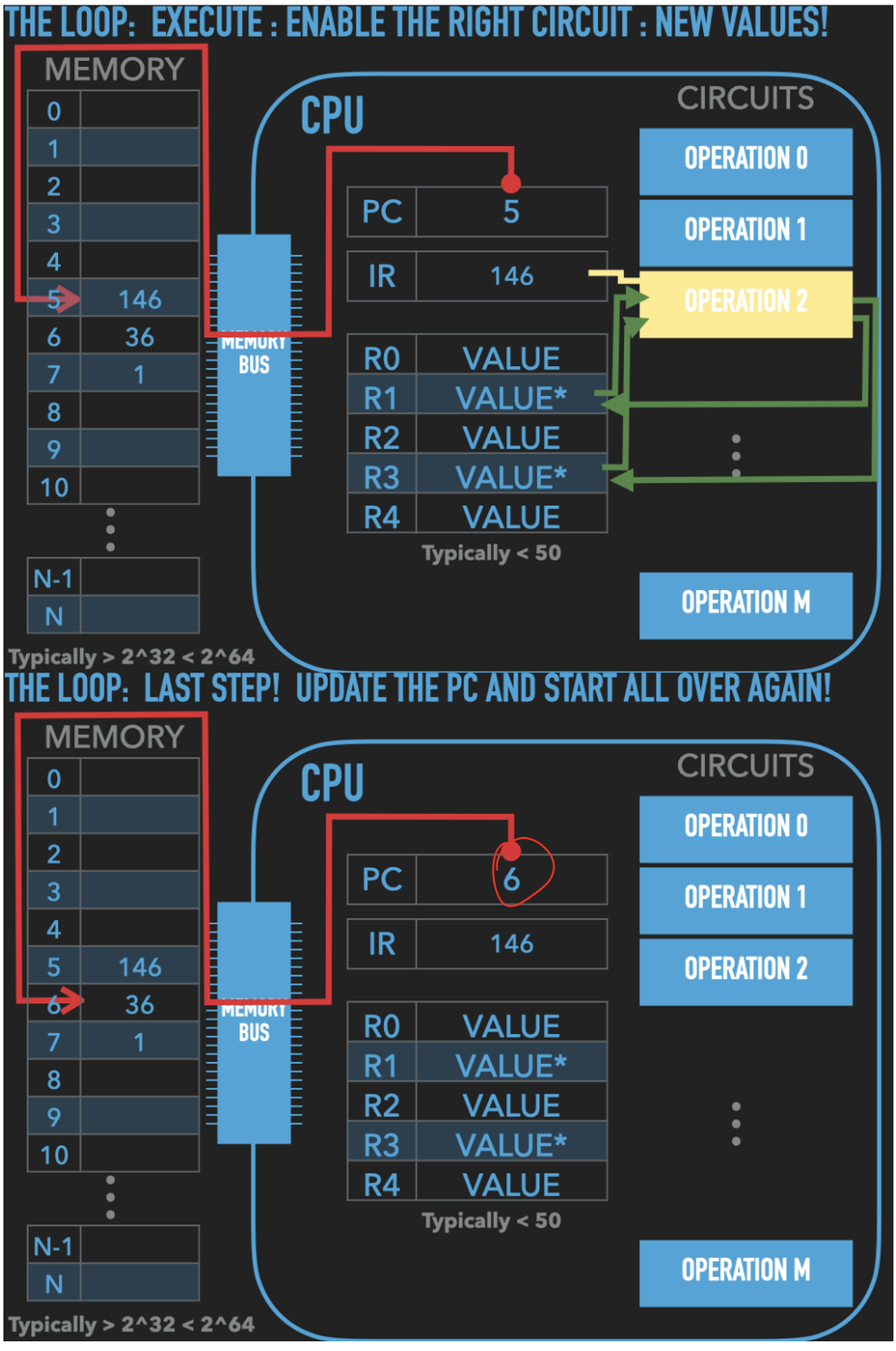

1.
THE LOOP: FETCH : IS THE ACT OF LOADING IR FROM ADDRESS POINTED TO BY PC
MEMORY
0
1
2
3
4
5
146
6
36
7
1
8
9
10
N-1
N
Typically > 2^32 < 2^64
CPU
CIRCUITS
PC
5
IR
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
MEMORY BUS
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
PC
1.
THE LOOP: FETCH : IS THE ACT OF LOADING IR FROM ADDRESS POINTED TO BY PC
MEMORY
0
1
2
3
4
5
146
6
36
7
1
8
9
10
N-1
N
Typically > 2^32 < 2^64
CPU
CIRCUITS
PC
5
IR
146
R0 VALUE
R1 VALUE
R2 VALUE
R3 VALUE
R4 VALUE
Typically < 50
MEMORY BUS
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M

2.



decode IR opcode and specify input and output.

3.


THE LOOP: EXECUTE : ENABLE THE RIGHT CIRCUIT : NEW VALUES!
MEMORY
0
1
2
3
4
5 146
6 36
7 1
8
9
10
N-1
N
Typically > 2^32 < 2^64
CPU
CIRCUITS
PC 5
IR 146
R0 VALUE
R1 VALUE*
R2 VALUE
R3 VALUE*
R4 VALUE
Typically < 50
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
MEMORY BUS

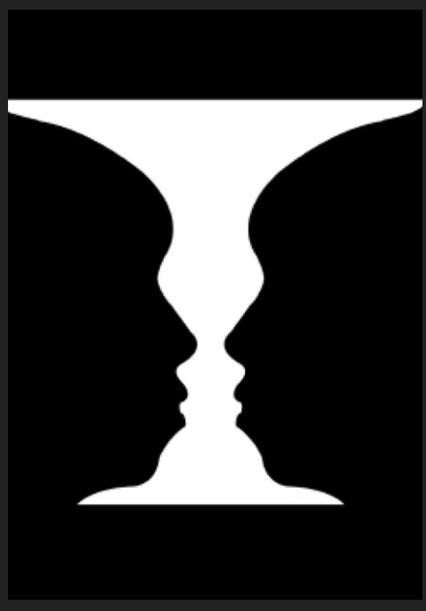
4

THE LOOP: LAST STEP! UPDATE THE PC AND START ALL OVER AGAIN!
MEMORY
0
1
2
3
4
5 146
6 36
7 1
8
9
10
N-1
N
Typically > 2^32 < 2^64
CPU
CIRCUITS
PC 6
IR 146
R0 VALUE
R1 VALUE*
R2 VALUE
R3 VALUE*
R4 VALUE
Typically < 50
OPERATION 0
OPERATION 1
OPERATION 2
OPERATION M
MEMORY BUS

THIS LEADS US TO

TWO VIEWS OF PROGRAMMING

- Low-Level: Loading memory with OPCodes to direct what operations "The Loop" will take
- High-Level: Cleverly use the mechanisms to do what you wan't … and we have been very clever… just look at you smart phone.



TEXT

TO UNDERSTAND WHAT WE CAN DO (AND WHAT HAS BEEN DONE)

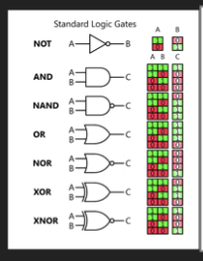
- What exactly are values?
- What kind of operations are there?
- Is there a difference between values and memory addresses?

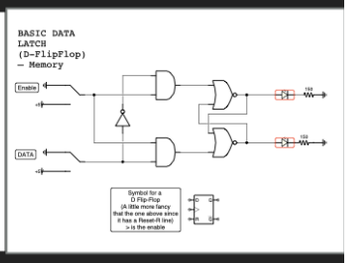
LETS THINK ABOUT VALUES AND MEMORY LOCATIONS FOR A MOMENT

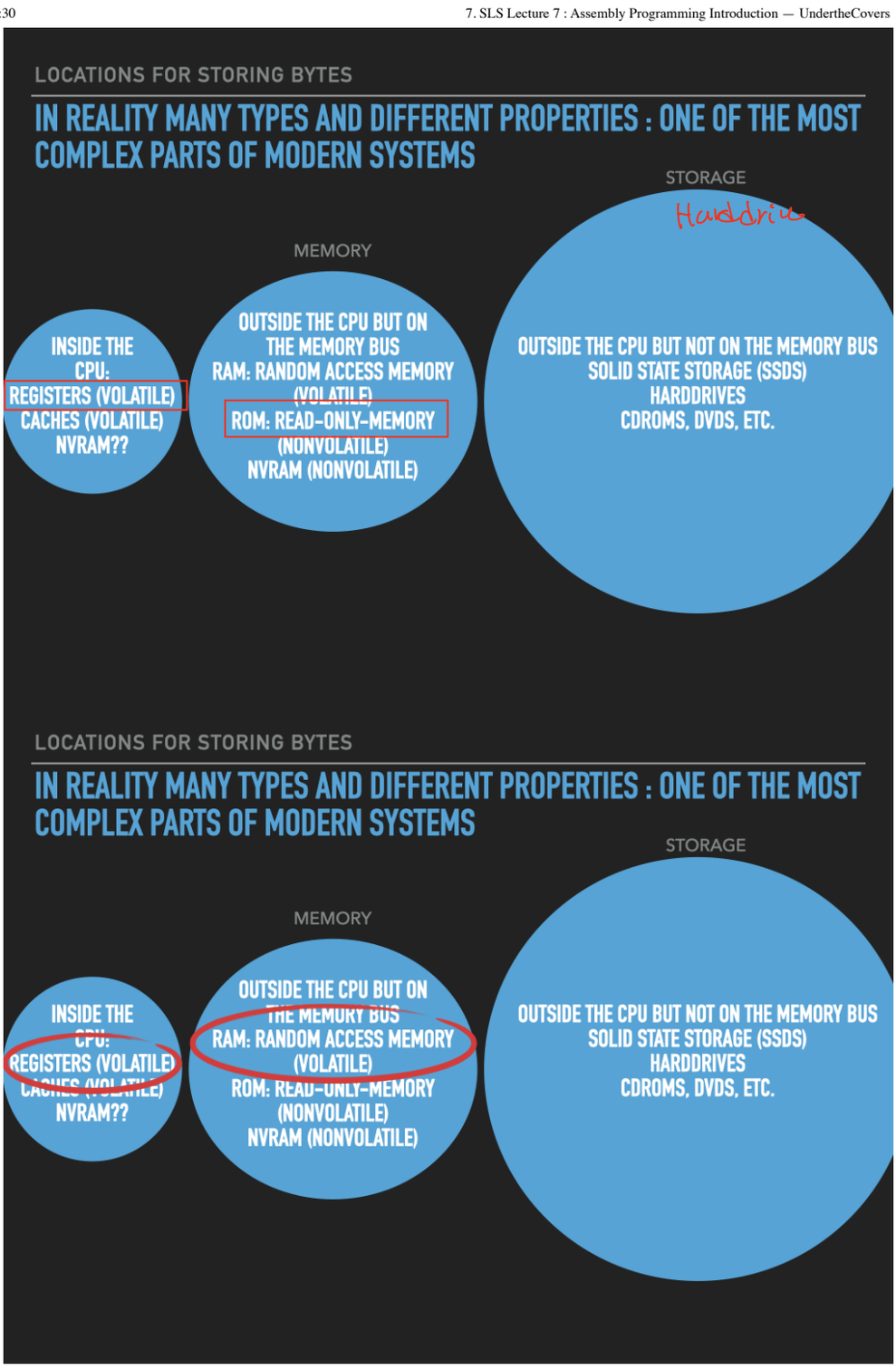
TEXT

VALUES: THE BYTE AS A PATTERN OF "ELECTRICITY"

- What exactly is a register and a memory location
  - A physical location of the computer that can store an electrical pattern of ON's and OFF's (1 bit)
  - Basic unit is 8 bits
- Circuits can "read" and write the value in a **location**







So a Process is :

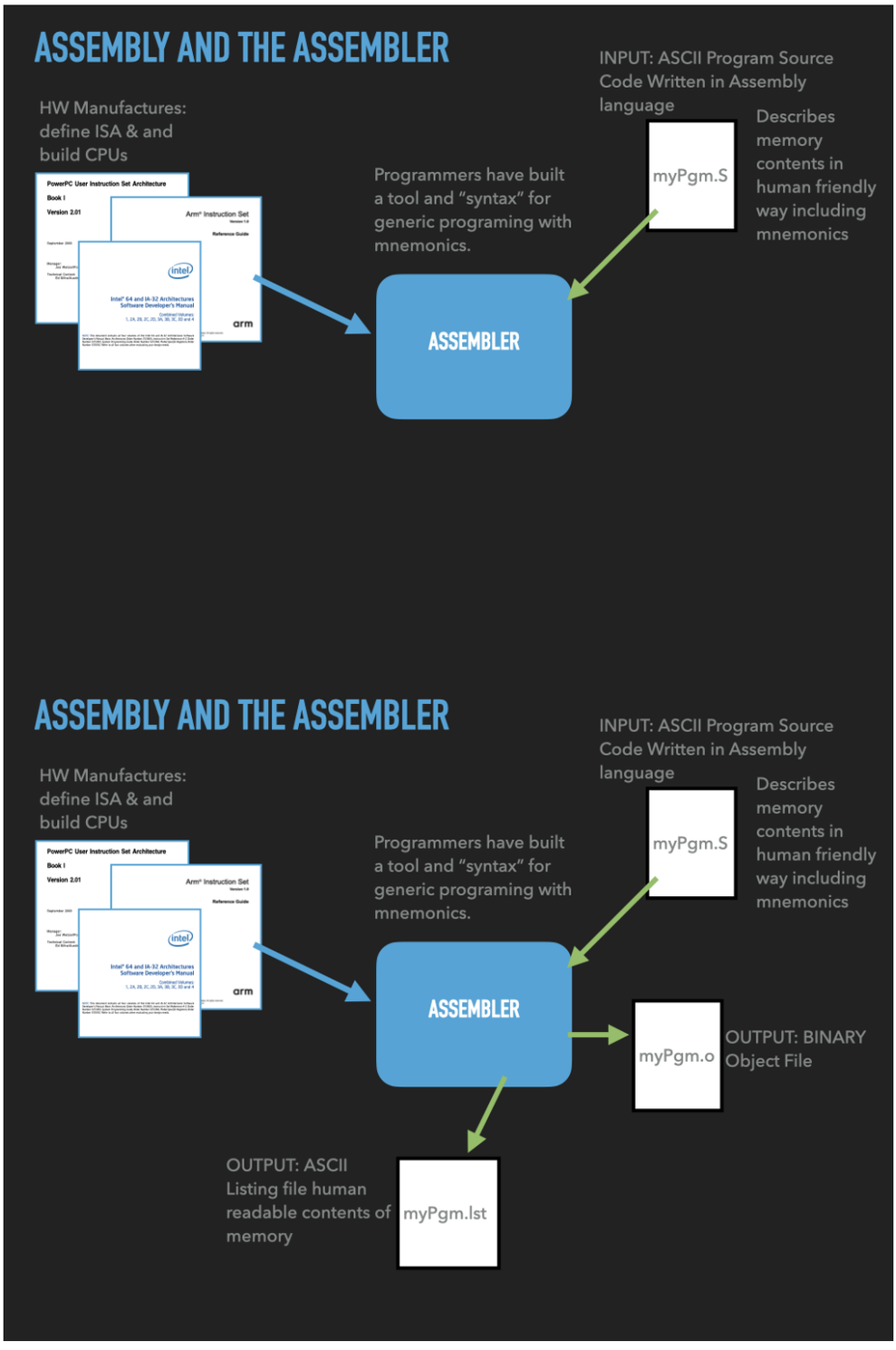
- an OS provided context that lets us
  - direct the CPU via a binary program file
    - that is loaded into the RAM memory array when we “run” it! (exec)
- A binary “contains” the initial contents of memory that the OS “loads” into our process’s memory.
  - “memory image” – the exact byte values and where they go into memory
- A process’s memory is called the process’s address space.

7.3. The Tools and how to use them

1. Preparing / creating binaries
2. Assembler: Tool that translates a programmer’s description of what to put into memory into fragments of an executable file
3. Linker: Tool that combines the fragments into an complete executable that the OS can load
4. Process inspection and manipulation

5. A Debugger that allows us to look at and control a Process

7.3.1. Assembler and Linker



ASSEMBLY AND THE ASSEMBLER

HW Manufactures: define ISA & and build CPUs

Programmers have built a tool and "syntax" for generic programing with mnemonics.

INPUT: ASCII Program Source Code Written in Assembly language

myPgm.S

Describes memory contents in human friendly way including mnemonics

label → human friendly

ASSEMBLER

myPgm.o

OUTPUT: BINARY Object File

OUTPUT: ASCII Listing file human readable contents of memory

myPgm.lst

LINKER/ LOADER

create the exact memory image.

OUTPUT: BINARY Memory "IMAGE"

myPgm.bin

make those human friendly label to CPU friendly

TEXT

GNU BINARY UTILITIES

- All the tools we will use (and are used heavily in the industry)
- Tools for working with "Binaries"
- Most important to us :
  - AS – Gnu Assemble (GAS)
  - LD - Gnu Link

https://www.gnu.org/software/binutils/

GNU Operating System

Sponsored by the Free Software Foundation

ABOUT GNU PHILOSOPHY LICENSES EDU

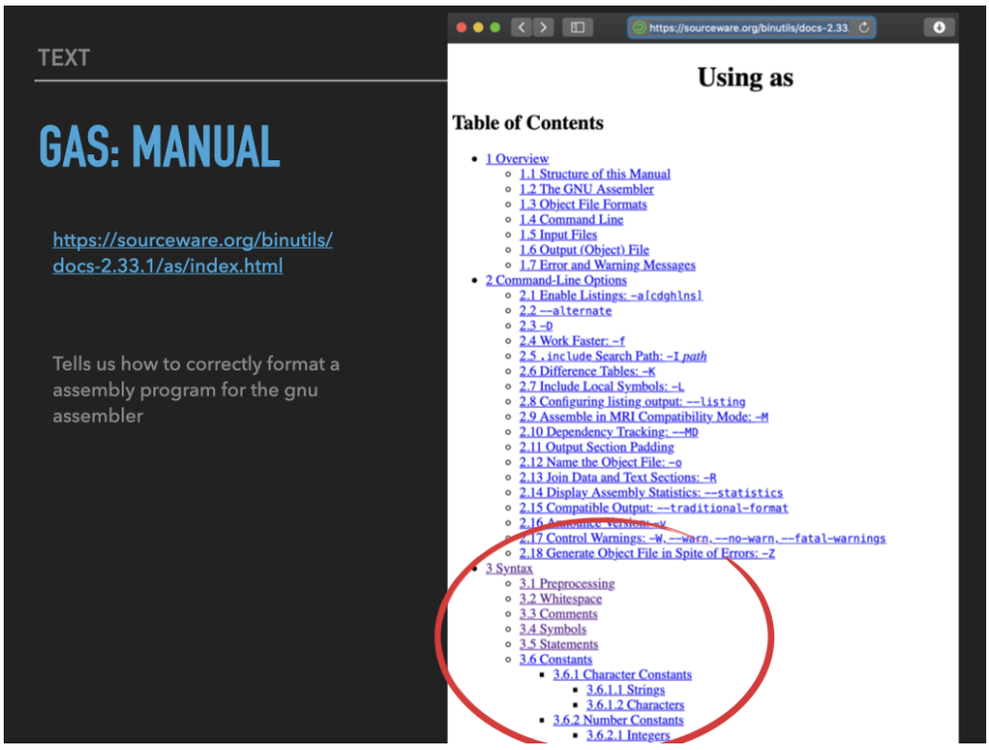
GNU Binutils

The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

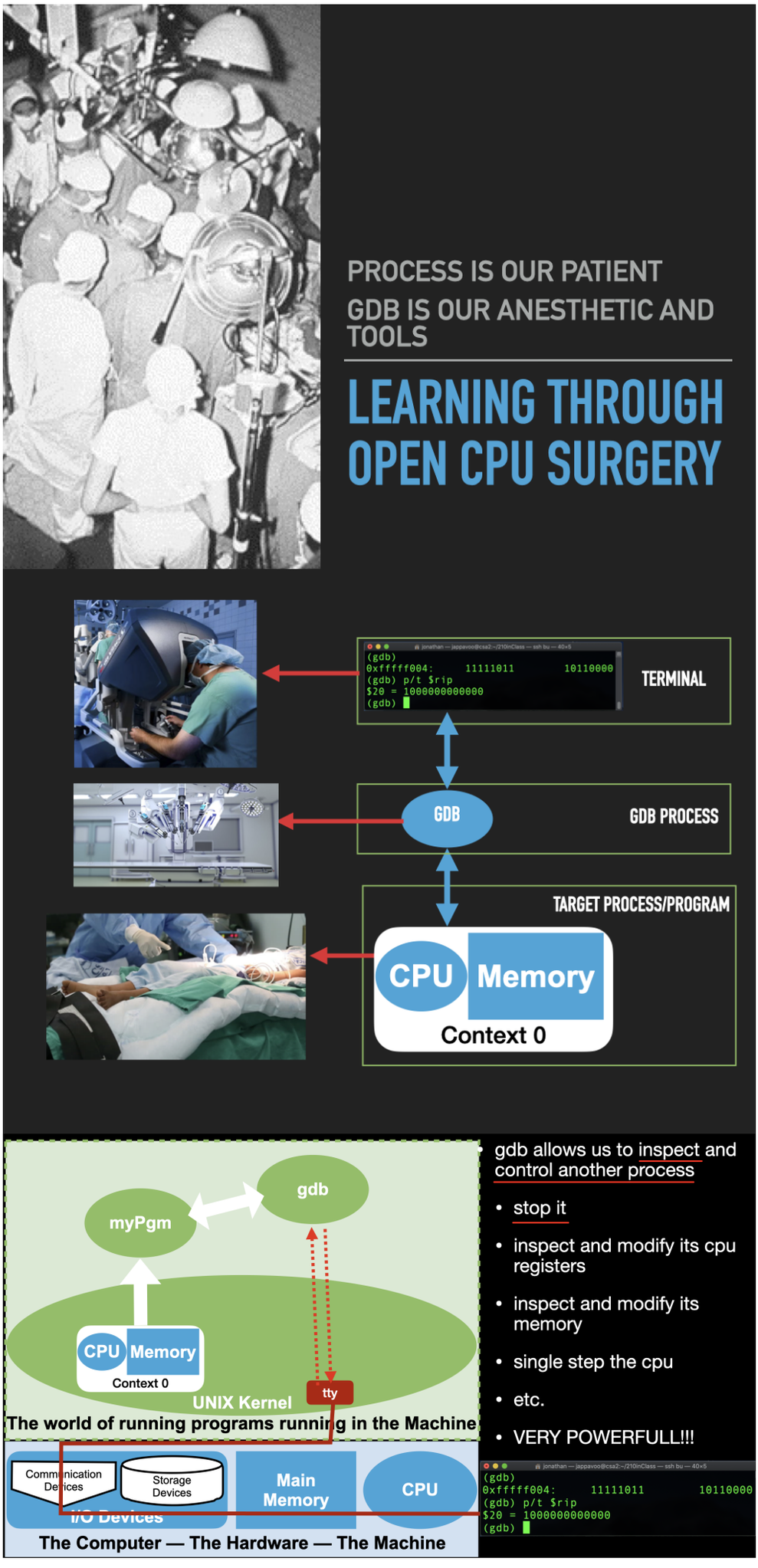
But they also include:

- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.



7.3.2. Debugger

Provides us a way of looking inside a process, freezing its execution, examining and modify the cpu registers and memory.



7.3.3. GDB Manual

https://www.gnu.org/software/gdb/documentation/

CODE: asm - The 'Empty' assembly program

```
/*  General antomy of a assembly program line
[lablel]:   <directive or opcode> [operands]  # comment
*/
        .intel_syntax noprefix          # assembler syntax to use <directive>
                                        # set assembly language format to intel

        .text                           # linker section <directive>
                                        # let the linker know that what follows
are cpu instructions to
                                        # to be executed -- uposed to values
that represent data.
                                        # For historical reasons cpu
instructions are called "text"

        .global _start                  # linker symbol type <directive>
                                        # makes the symbol _start in this case
visible to the linker
                                        # The linker looks for an _start symbol
so that it knows address
                                        # of the first instruction of our
program

_start:                                 # introduce a symbolic (human readable)
label for "this" address
                                        # associates the address of this point
in our program with the
                                        # name following the ':' -- in our case
_start
                                        # In our program or in the debugger we
can use this name to
                                        # to refer to this location -- address.
And thus the values
                                        # that end up here.
        .byte 0x00, 0x00, 0x00, 0x00    # .byte directive place value at
successive locations in memory
        .byte 0x00, 0x00, 0x00, 0x00    #
(https://sourceware.org/binutils/docs/as/Byte.html#Byte)
```

The OS lets us have access to parts of the CPU and Memory via a Process. For everything else we will need to make calls to the OS Kernel functions to do.

Let's use the standard tools to build a "empty" binary, create a process from it and use gdb to explore the parts of the machine that a Process lets us control. Eg use the debugger to read, write memory, explore the internals of the cpu and control it!

setup

```
cd
mkdir empty
cd empty
# of course at this point it would be a good idea to setup a git repository
but we will skip
```

lets write some code!!!!

`.fill` can be used to fill memory ;-) `repeat, size, value`

```
      .fill 16, 1, 0x00             // .fill directive fills memory with n
copies of values
                                    //
(https://sourceware.org/binutils/docs/as/Fill.html#Fill)
```

NOTES: on building empty

```
# To assemble the code we will use the following command:
as empty.s -o empty.o

# ok lets see what happened
ls

# lets link it up
ld empty.o -o empty

ls

# lets examine this elf file
file empty
ls -l empty

# still pretty big given that we only asked to load 4 bytes of zeros into memory
# all the other stuff necessary to describe to the OS and other tools the program image

# use objdump tool to print out ascii data from the executable file
# -s
# --full-contents
# Display the full contents of any sections requested. By default all non-empty
# sections are displayed.
objdump -s empty

# Display the symbol table the locations that each "symbol" got assigned to by the linker
#  This is the kind of extra information that make the elf file larger than just the contents
#  note where _start got assigned to
objdump -t empty

# ok lets got to the debugger
```

```
# Let use gdb command to poke around empty
# To get the debugger going:
gdb -x setup.gdb empty

# To setup the assembly syntax to intel:
set disassembly-flavor intel → intel syntax


# at this point no process has been created yet
# we are just exploring the binary file
p _start
x /4xb _start

# to get things running we use the run command
# but we don't want to execute any instructions so
# we first place a breakpoint at the location of _start
b _start

# now run
run

# lets see if a process was created
!ps auxgww | grep empty
info proc

# ok lets poke around the process
# lets examine the cpu
info registers

# we can print out the value of an individual register with
# variables in gdb $<name> there are convience variables for each
# of the registers
p /x $rax
p /t $rax
p /d $rax
p /x $rip

# lets examine the memory : peek

x/8xb 0x401000
x/2i  instruction
  next 2

# lets write some bytes into memory
# lets write and instruction at _start
#  we will try and use the popcnt instruction
# popcnt rbx, rax -- rbx = population count of rax
# see intel sdm volume 2B popcnt
# tells use how to encode the instruction via
# opcode encoding.
# as we load the memory we will ask gdb to try and interpret
# the opcode for us and see how the values change what
# the cpu will do when we have it fetch the instruction
# rex.w = 0x48 -- 64 bit operand
# D8 -- encodes which registers are the operands

set {unsigned char}(_start) = 0xF3
x/5xb _start              next location
disas _start                 ↓  rax
set {unsigned char}(_start+1) = 0x48
set {unsigned char}(_start+2) = 0x0F
set {unsigned char}(_start+3) = 0xB8
set {unsigned char}(_start+4) = 0xC3 → dB    11011000
x/5xb _start
x/5tb _start
x/5db _start
x/5ub _start
x/1i _start

# lets execute the instruction and play
# around with the register values
# to make life easier we will use the gdb text uit support
# Configure a layout that is more friendly to assembly code:
tui new-layout mylayout regs 3 {src 1 asm 1} 2 cmd 1 status 0
# Switch to the new layout:
layout mylayout
winh cmd 4
focus cmd

# run one interation of the processor loop:
stepi

# look what happened to the registers
set $rax = 0b1011
```

```
p /t $rax

# how do we reexecute the instruction?

set $pc = _start
stepi
```


POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F B8 /r	POPCNT *r16, r/m16*	RM	Valid	Valid	POPCNT on *r/m16*
F3 0F B8 /r	POPCNT *r32, r/m32*	RM	Valid	Valid	POPCNT on *r/m32*
F3 REX.W 0F B8 /r	POPCNT *r64, r/m64*	RM	Valid	N.E.	POPCNT on *r/m64*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{       IF (SRC[ i] = 1) // ith bit
        THEN Count++; Fi;
}
DEST ← Count;
```

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT: int _mm_popcnt_u32(unsigned int a);

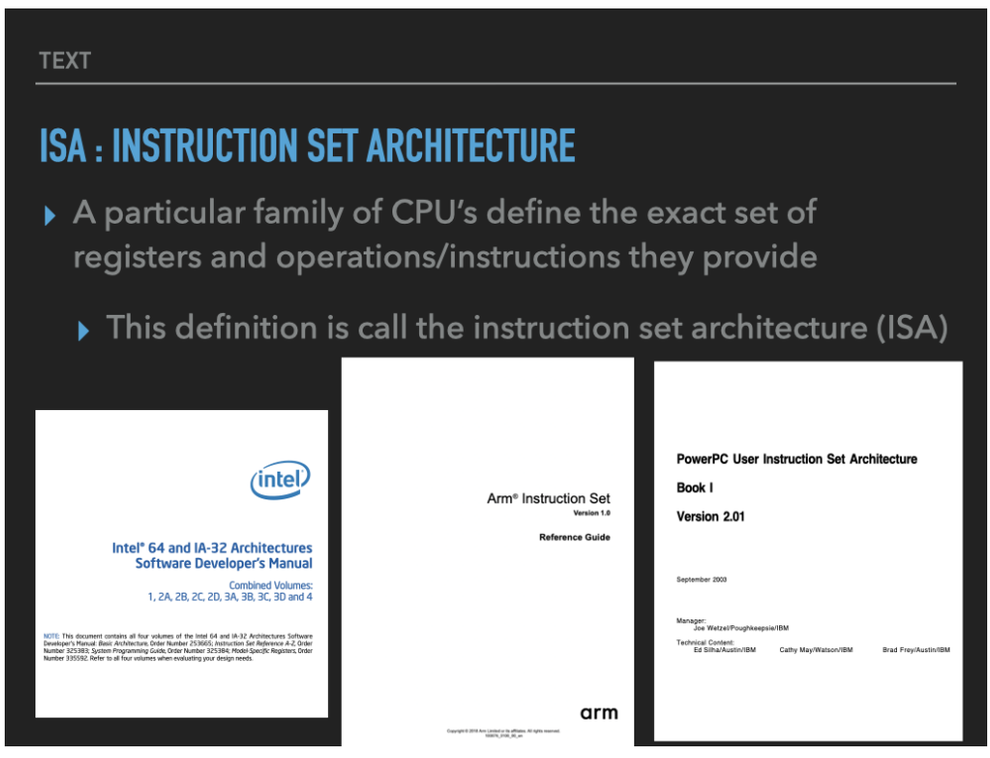
POPCNT: int64_t _mm_popcnt_u64(unsigned __int64 a);

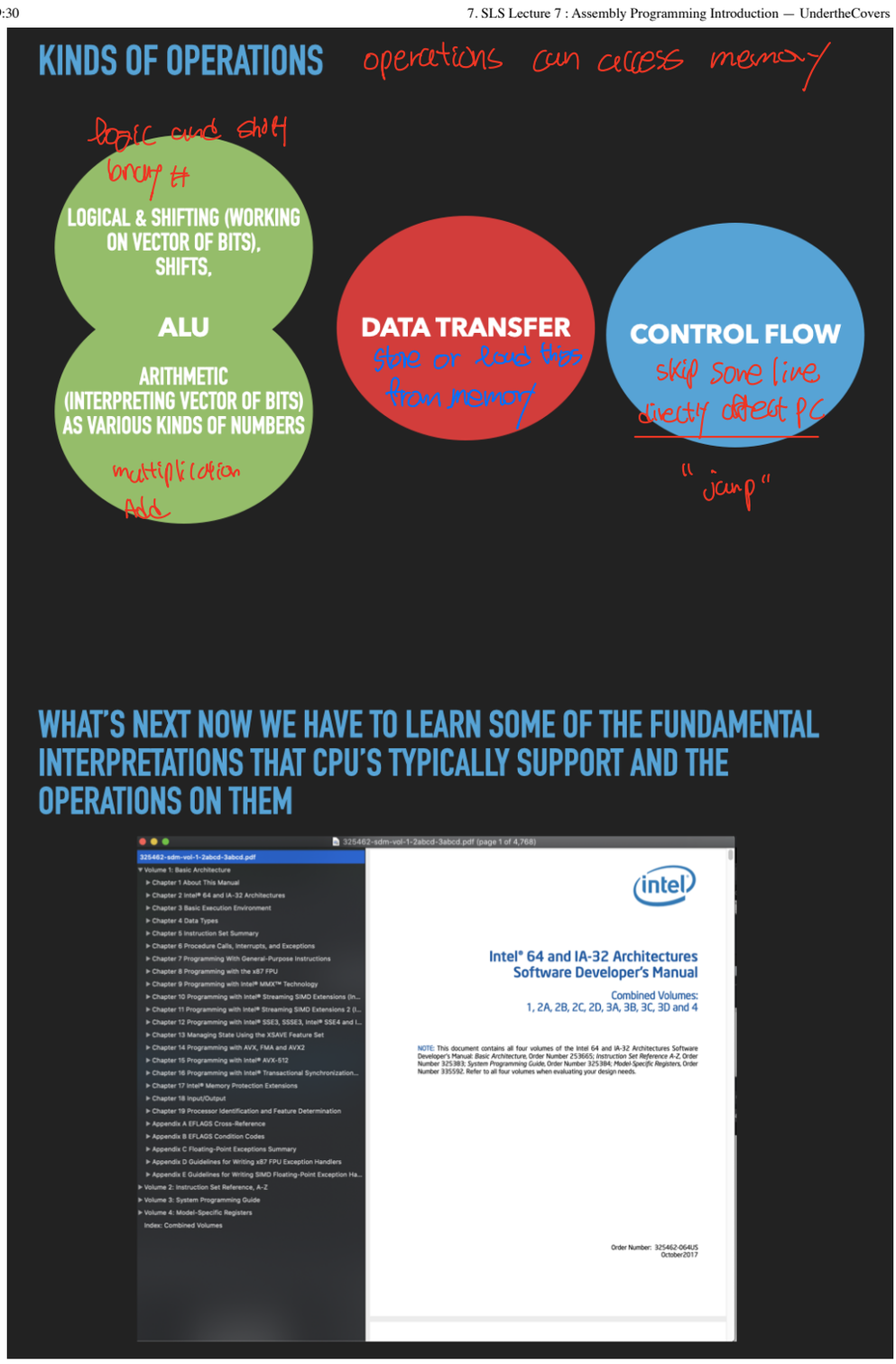
Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.





7.4. Intel Manuals

Freely available online:
https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

1. **Volume 1: Topics regarding general purpose programming**

- **largely what we will focus on**

1. **Volume 2: Is a reference of all the instructions which we can consult as needed**

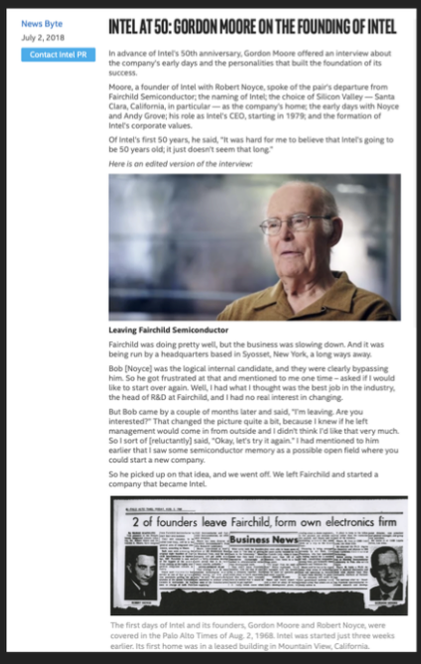
I usually grab the very large manual that combines volumes 1,2,3,4
https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#combined

Which include Volumes 3 and 4. These focus on the systems programming features necessary to write operating systems kernels. But these are not necessary for normal user application programming.

7.5. Extra info about Intel

TEXT

- A fascinating company that has been central to mainstream computing (but don't be fooled there are many others IBM, ARM, SAMSUNG,HP, DEC, MIPS,..)
- Chapter 1 of Volume 1 has a nice history of the processor line if you are interested... Big take way it is this history that can make the ISA so complicated compared to something like RISCV https://riscv.org/



News Byte
July 2, 2018
Contact Intel PR

INTEL AT 50: GORDON MOORE ON THE FOUNDING OF INTEL

In advance of Intel's 50th anniversary, Gordon Moore offered an interview about the company's early days and the personalities that built the foundation of its success.

Moore, a founder of Intel with Robert Noyce, spoke of the pair's departure from Fairchild Semiconductor; the naming of Intel; the choice of Silicon Valley — Santa Clara, California, in particular — as the company's home; the early days with Noyce and Andy Grove; his role as Intel's CEO, starting in 1979; and the formation of Intel's corporate values.

Of Intel's first 50 years, he said, "It was hard for me to believe that Intel's going to be 50 years old; it just doesn't seem that long."

Here is an edited version of the interview:

Leaving Fairchild Semiconductor

Fairchild was doing pretty well, but the business was slowing down. And it was being run by a headquarters based in Syosset, New York, a long ways away.

Bob [Noyce] was the logical internal candidate, and they were clearly bypassing him. So he got frustrated at that and mentioned to me one time – asked if I would like to start over again. Well, I had what I thought was the best job in the industry, the head of R&D at Fairchild, and I had no real interest in changing.

But Bob came by a couple of months later and said, "I'm leaving. Are you interested?" That changed the picture quite a bit, because I knew if he left management would come in from outside and I didn't think I'd like that very much. So I sort of [reluctantly] said, "Okay, let's try it again." I had mentioned to him earlier that I saw some semiconductor memory as a possible open field where you could start a new company.

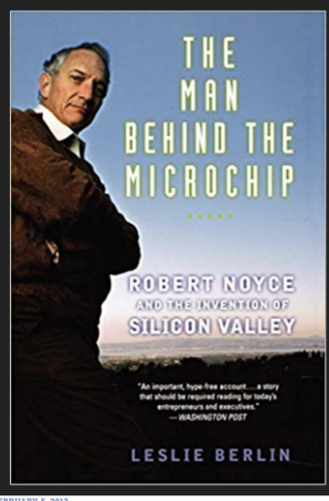
So he picked up on that idea, and we went off. We left Fairchild and started a company that became Intel.

2 of founders leave Fairchild, form own electronics firm

Business News

The first days of Intel and its founders, Gordon Moore and Robert Noyce, were covered in the Palo Alto Times of Aug. 2, 1968. Intel was started just three weeks earlier. Its first home was in a leased building in Mountain View, California.

https://newsroom.intel.com/news/intel-50-gordon-moore-founding-intel/#gs.ub6wgt

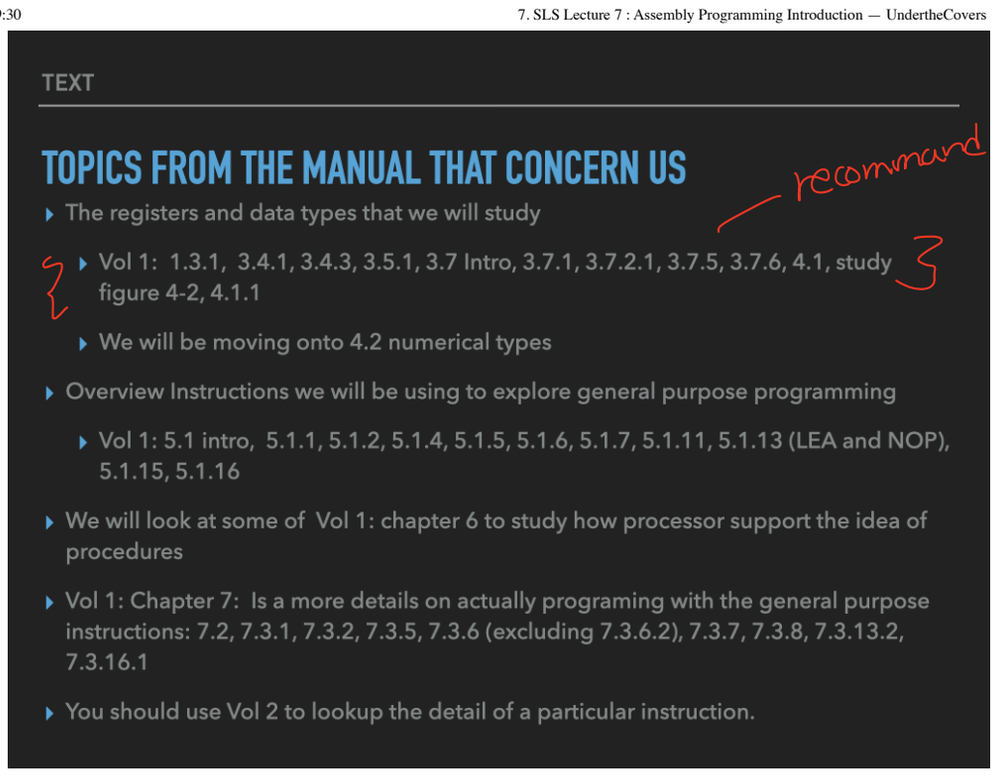


AIRED FEBRUARY 5, 2013

Silicon Valley

In 1957, decades before Steve Jobs dreamed up Apple or Mark Zuckerberg created Facebook, a group of eight brilliant young men defected from the Shockley Semiconductor Company in order to start their own transistor business. Their leader was 29-year-old Robert Noyce, a physicist with a brilliant mind and the affability of a born salesman who would co-invent the microchip — an essential component of nearly all modern electronics today, including computers, motor vehicles, cell phones and household appliances.

http://www.pbs.org/wgbh/americanexperience/films/silicon/



By Jonathan Appavoo