ISAs and Instruction Sets, File Types, & Memory

CS 240 - The University of Illinois

Wade Fagen-Ulmschneider

January 27, 2022

Instruction Set Architecture (ISA)

CPU Registers

Each CPU has a very limited number of ______________ used for general purpose CPU operations.

Move:	MOV, XCHG, PUSH, POP, ...
Arithmetic (int):	ADD, SUB, MUL, DIV, NEG, CMP, ...
Logic:	AND, OR, XOR, SHR, SHL, ...
Control Flow:	JMP, LOOP, CALL, RET, ...
Synchronization:	LOCK, ...
Floating Point:	FADD, FSUB, FMUL, FDIV, FABS, ...

Instruction Set Size

ARM processors have significantly fewer instructions and known as:

Instruction Set Size

x86/x64 processors a greater set (more specialized) instructions and know as:

RISC vs. CISC?

Seeing CPU Instructions in a Real Program

04.c

```c
int main() {
  int a = 0;
  a = a + 3;
  a = a - 2;
  a = a * 4;
  a = a / 2;
  a = a * 5;
  printf("Hi");
  a = a * 479;
  ...
```

Compiling + Printing Instructions

```
gcc 04.c

objdump -d ./a.out
```

```c
int main() {
```

```
f3 0f 1e fa             endbr64
55                      push   %rbp
48 89 e5                mov    %rsp,%rbp
48 83 ec 10             sub    $0x10,%rsp
```

```
4     int a = 0;
```

```
c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
```

```c
a = a + 3;
```

```
83 45 fc 03          addl   $0x3,-0x4(%rbp)
```

```
6     a = a - 2;
```

```
83 6d fc 02                 subl   $0x2,-0x4(%rbp)
```

```
7    a = a * 4;
```

```
c1 65 fc 02                 shll   $0x2,-0x4(%rbp)
```

```
    a = a / 2;
```

```c
8   a = a / 2;
```

```
8b 45 fc        mov    -0x4(%rbp),%eax
89 c2           mov    %eax,%edx
c1 ea 1f        shr    $0x1f,%edx
01 d0           add    %edx,%eax
d1 f8           sar    %eax
89 45 fc        mov    %eax,-0x4(%rbp)
```

```c
a = a * 5;
```

```
8b 55 fc        mov    -0x4(%rbp),%edx
89 d0           mov    %edx,%eax
c1 e0 02        shl    $0x2,%eax
01 d0           add    %edx,%eax
89 45 fc        mov    %eax,-0x4(%rbp)
```

```
10   printf("Hi");
```

```
48 8d 3d f0 0d 00 00   lea     0xdf0(%rip),%rdi
                       # 2004 <_IO_stdin_used+0x4>

b8 00 00 00 00         mov     $0x0,%eax
e8 42 fe ff ff         callq   1060<printf@plt>
```

```
  a = a * 479;
```

```
8b 45 fc                  mov    -0x4(%rbp),%eax
69 c0 df 01 00 00         imul   $0x1df,%eax,%eax
89 45 fc                  mov    %eax,-0x4(%rbp)
```

Program Counter

```
f3 0f 1e fa                 endbr64
55                          push   %rbp
48 89 e5                    mov    %rsp,%rbp
48 83 ec 10                 sub    $0x10,%rsp
c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
83 45 fc 03                 addl   $0x3,-0x4(%rbp)
83 6d fc 02                 subl   $0x2,-0x4(%rbp)
c1 65 fc 02                 shll   $0x2,-0x4(%rbp)
8b 45 fc                    mov    -0x4(%rbp),%eax
89 c2                       mov    %eax,%edx
c1 ea 1f                    shr    $0x1f,%edx
01 d0                       add    %edx,%eax
d1 f8                       sar    %eax
```

Operation Timings

Latency: This is the delay that the instruction generates in a dependency chain if the next dependent instruction starts in the same execution unit. The numbers are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably.

Arithmetic instructions											
ADD, SUB	r,r	1	0	1	0	0.25	0/1	alu0/1		86	c
ADC, SBB	r,r/i	3	0	10	0	10	1	int,alu		86	
CMP	r,r	1	0	1	0	0.25	0/1	alu0/1		86	c
MUL, IMUL	r8	1	0	10	0		1	int	mul	86	
DIV	r8/m8	1	20	74	0	34	1	int	fpdiv	86	a

An ADD with no input carry can be executed in as little as 1 cycle!

Addition with input carry (ADC) can take at least 10x longer.

Comparing two values can be executed in as little as 1 cycle.

Multiplication runs in similar time as ADC.

*Best case for division is **74x** slower than best case for addition!*

Endianness:

Big Idea: How do we represent multi-byte characters in memory?

```c
int main() {
    int i = 3 + (2 << 8) + (1 << 16);
    char *s = (char *)&i;
    printf("%02x %02x %02x %02x\n",
               s[0], s[1], s[2], s[3]);
    return 0;
}
```

04b.c

s[0]	s[1]	s[2]	s[3]

Big Endian

s[0]	s[1]	s[2]	s[3]

Little Endian

s[0]	s[1]	s[2]	s[3]

Network Byte Order

Host Byte Order

File Types and Extensions

File Extensions

The most common way to identify the contents of a file is by the **file extension**. The file extension is defined as:

cs240.png

mp1.c

mp1.h

taylor.swift.mp4

“Plain Text” File

PNG File Format

W3C Recommendation

W3C®

Portable Network Graphics (PNG) Specification (Second Edition)

Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E)

W3C Recommendation 10 November 2003

This version:
http://www.w3.org/TR/2003/REC-PNG-20031110
Latest version:
http://www.w3.org/TR/PNG
Previous version:
http://www.w3.org/TR/2003/PR-PNG-20030520
Editor:
David Duce, Oxford Brookes University (Second Edition)
Authors:
See author list

Please refer to the errata for this document, which may include some normative corrections.

See also the translations of this document.

Abstract

This document describes PNG (Portable Network Graphics), an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. Indexed-color, grayscale, and truecolor images are supported, plus an optional alpha channel. Sample depths range from 1 to 16 bits.

https://www.w3.org/TR/PNG/

Overview

Your second MP in CS 240 is using C to manipulate low-level data. Specifically, we have hidden a few of our favorite GIFs inside of the "classic" CS 240 PNG image:

File: `240.png`

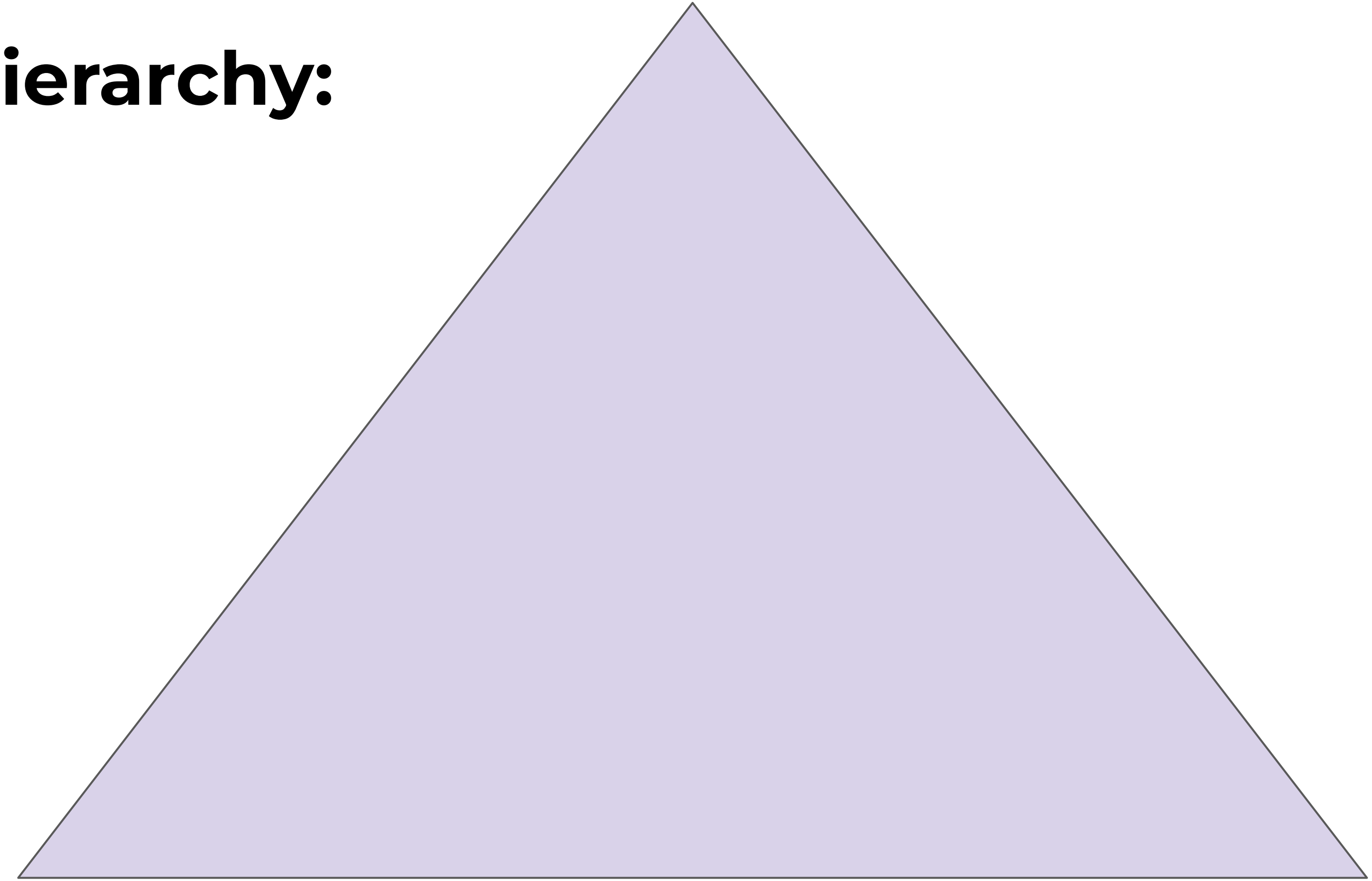
File: `sample/natalia.png`

File: `sample/waf.png`

All of these images look the same, but two have a hidden GIF inside of them!

Memory Hierarchy

Memory Hierarchy:



Processor Registers:

Processor Cache:

Random Access Memory (RAM):

Solid State ("Flash") Memory/Storage:

Hard Disk Storage:

High-Density / “Offline” / “Tape” Storage:

Memory Hierarchy

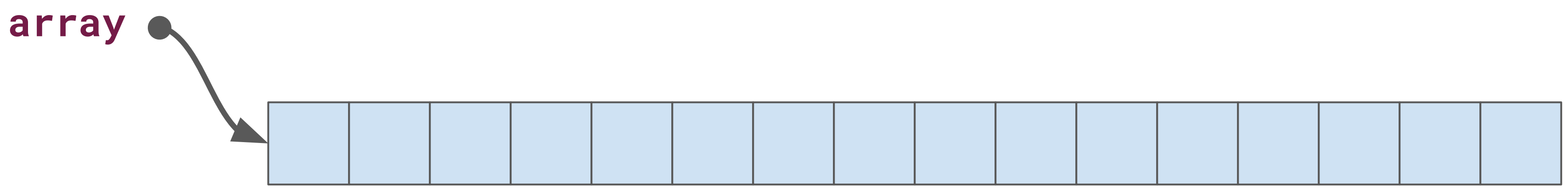
Program #1: `04rc.c`

```c
// Allocate an array of SIZE x SIZE of `unsigned ints`:
unsigned int *array = malloc(SIZE * SIZE *
                                    sizeof(unsigned int));

// Add data to each element of the array:
for (unsigned int c = 0; c < SIZE; c++) {
  for (unsigned int r = 0; r < SIZE; r++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```

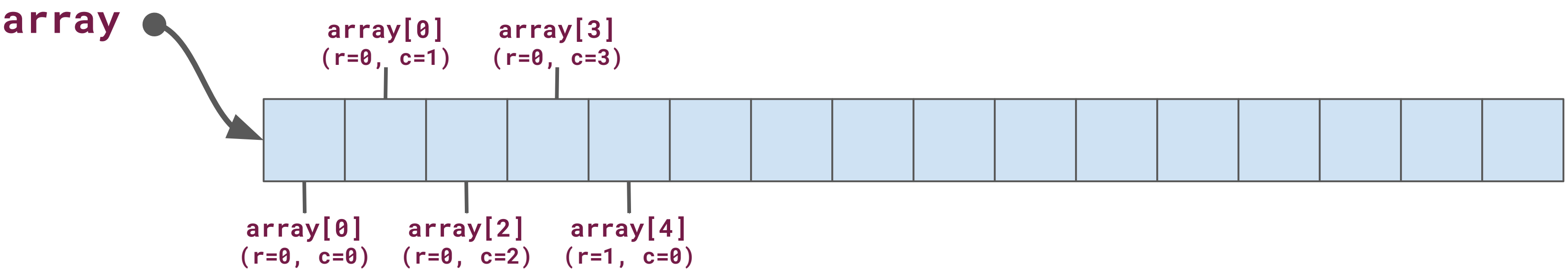
Example: SIZE=4

```c
// Allocate an array of SIZE x SIZE of `unsigned ints`:
unsigned int *array = malloc(SIZE * SIZE *
                             sizeof(unsigned int));
```

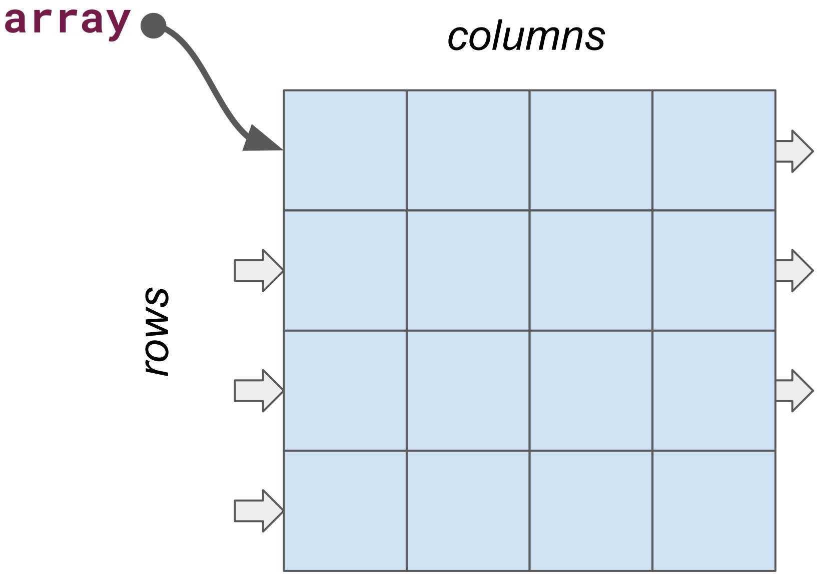


Example: SIZE=4

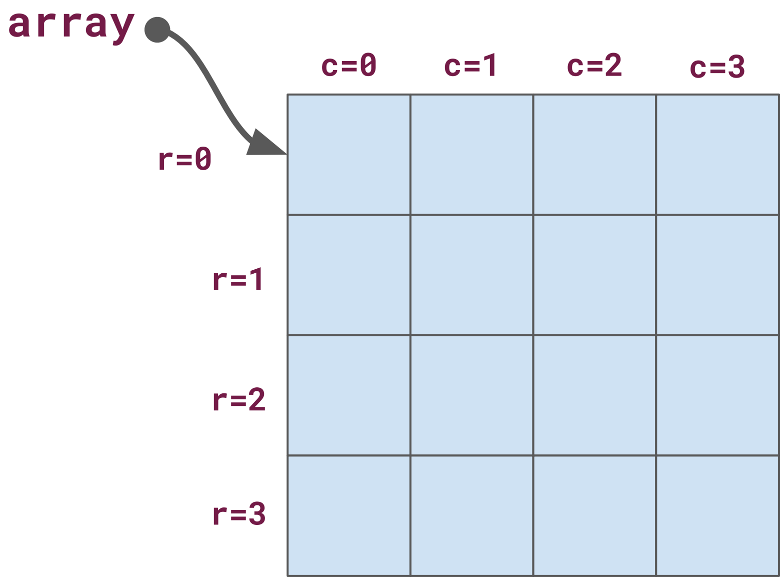
```
// Add data to each element of the array:
for (unsigned int c = 0; c < SIZE; c++) {
  for (unsigned int r = 0; r < SIZE; r++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```



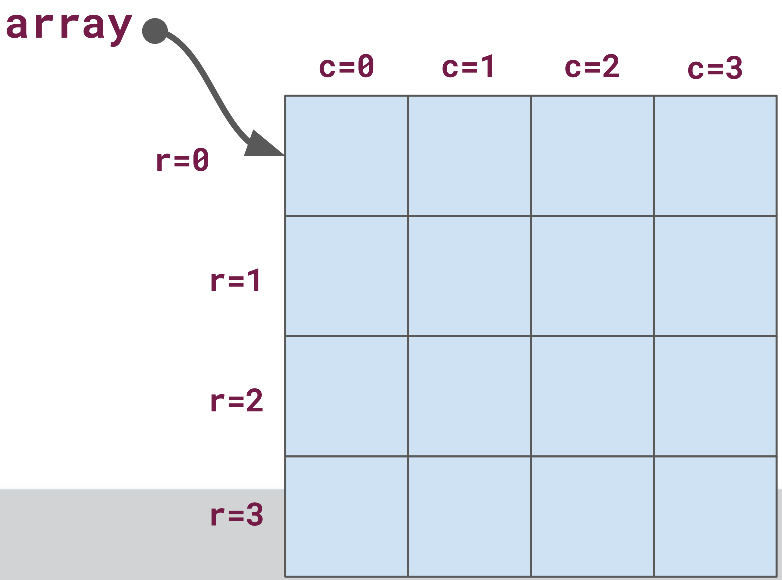
Example: SIZE=4



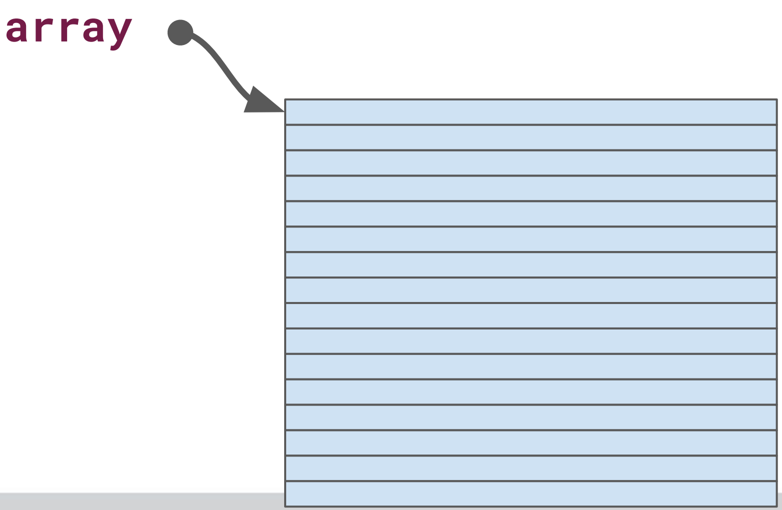
Example: SIZE=4



```
  // Add data to each element of the array:
  for (unsigned int c = 0; c < SIZE; c++) {
    for (unsigned int r = 0; r < SIZE; r++) {
      array[(r * SIZE) + c] = (r * SIZE) + c;
    }
  }
```



```
// Add data to each element of the array:
for (unsigned int c = 0; c < SIZE; c++) {
  for (unsigned int r = 0; r < SIZE; r++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```

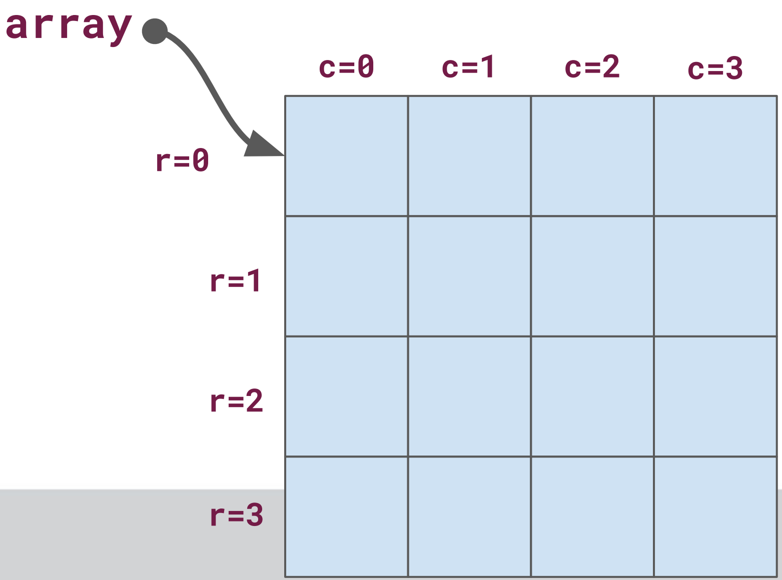


Program #2: `04rc.c`

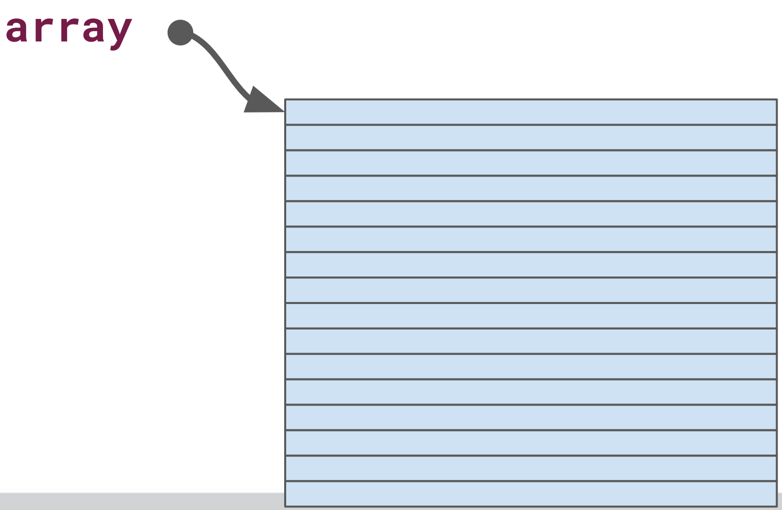
```c
// Allocate an array of SIZE x SIZE of `unsigned ints`:
unsigned int *array = malloc(SIZE * SIZE *
                             sizeof(unsigned int));

// Add data to each element of the array:
for (unsigned int r = 0; r < SIZE; r++) {
  for (unsigned int c = 0; c < SIZE; c++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```

```c
// Add data to each element of the array:
for (unsigned int r = 0; r < SIZE; r++) {
  for (unsigned int c = 0; c < SIZE; c++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```



```
// Add data to each element of the array:
for (unsigned int r = 0; r < SIZE; r++) {
  for (unsigned int c = 0; c < SIZE; c++) {
    array[(r * SIZE) + c] = (r * SIZE) + c;
  }
}
```



Let’s Test It!

```
$ make
$ time ./04cr
```

```
$ time ./04rc
```

How SIZE impacts performance?