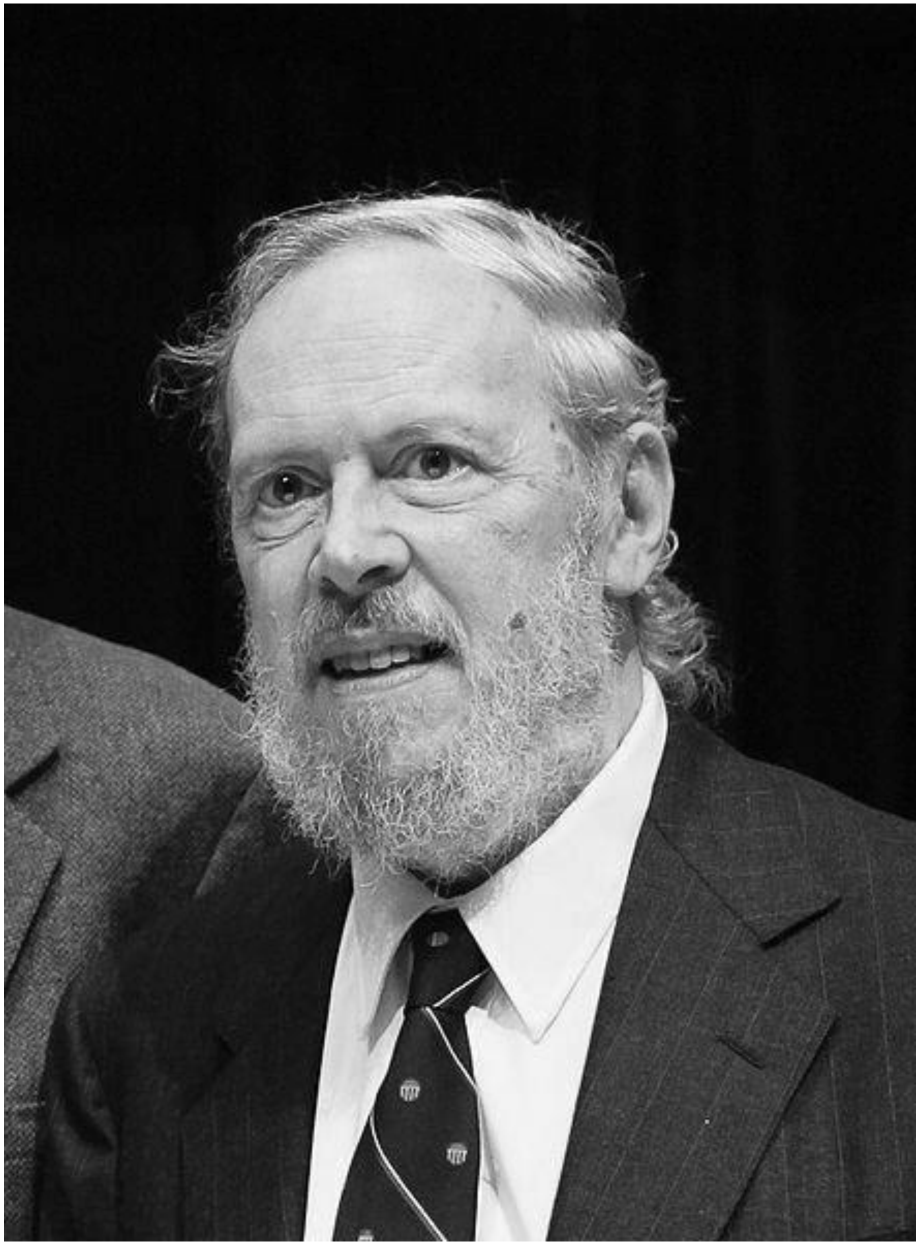
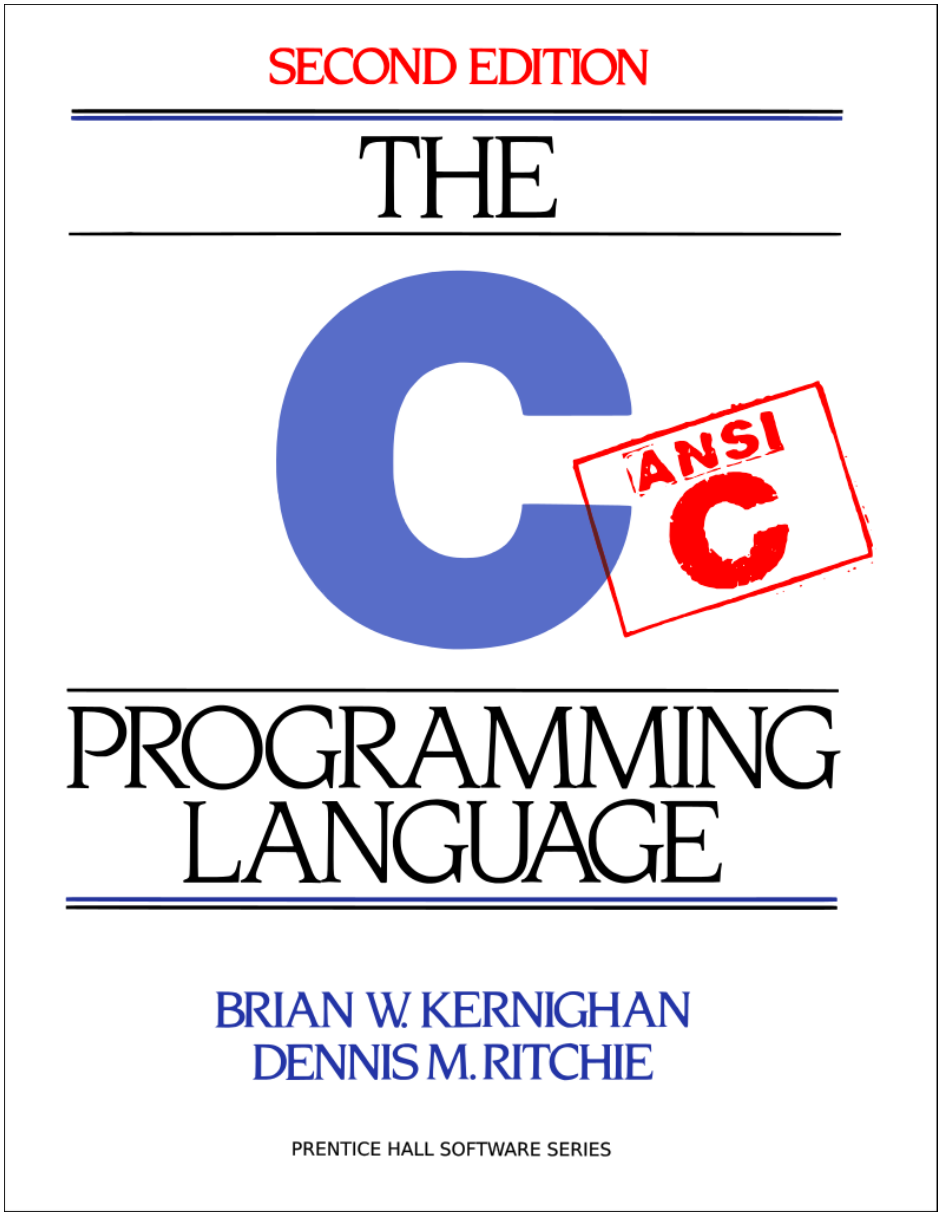


Dennis Ritchie



SECOND EDITION

THE

C

ANSI C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Ken Thompson built UNIX using C

http://cm.bell-labs.com/cm/cs/who/dmr/picture.html

The C Programming Language

“C is quirky, flawed, and an enormous success”
— Dennis Ritchie

“C gives the programmer what the programmer wants; few restrictions, few complaints”
— Herbert Schildt

“C: A language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language”
— Unknown

on.s and on/on.c

```
// Turn on an LED

// configure GPIO 20 for OUTPUT
ldr r0, =FSEL2
mov r1, #1
str r1, [r0]

// set GPIO 20 (to 1, 3.3V)
ldr r0, =SET0
mov r1, #(1<<20)
str r1, [r0]

loop: b loop
```

Assembly Language

the instructions you see
are
the instructions that are executed

```
#define FSEL2 0x20200008
#define SET0  0x2020001C
main()
{
    int *r0;
    int r1;

    // configure GPIO 20 for OUTPUT
    r0 = (int*)FSEL2;      // ldr r0, =FSEL2
    r1 = 1;                // mov r1, #1
    *r0 = r1;              // str r1, [r0]

    // set GPIO 20 (1, 3.3V)
    r0 = (int*)SET0;       // ldr r0, =SET0
    r1 = 1<<20;            // mov r1, #1
    *r0 = r1;              // str r1, [r0]

loop: goto loop;
}
```

Bare Metal

`-ffreestanding`

- **Program does not “stand on” (use) an operating system.**

`-nostdlib`

- **Program does not use standard libraries by default**

`-nostartfiles`

- **Don’t run any start code when the program starts. The program will provide the start code.**

“BCPL, B, and C all fit firmly in the traditional procedural family (of languages) typified by Fortran and Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are “close to the machine” in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. ...
At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.”

- Dennis Ritchie

100 FEET BELOW
SEA LEVEL

Assembly

```
00008000 <main>:
8000: e59f3010  ldr r3, [pc, #0x10]; 8018
8004: e3a02001  mov r2, #1
8008: e5832008  str r2, [r3, #0x08]
800c: e3a02601  mov r2, #0x100000
8010: e583201c  str r2, [r3, #0x1C]
8014: eafffffe  b   8014
8018: 20200000  .word   0x20200000

// This code is faster than our on.s
// (because it uses fewer instructions)
// The compiler has optimized it
```

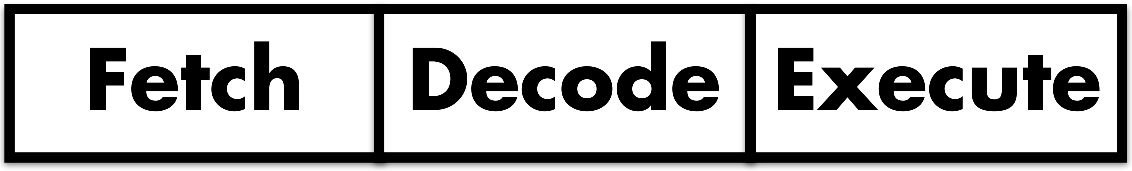
```
# branch instructions
cond      offset
cccc 1010 oooo oooo oooo oooo oooo oooo

1. Condition codes
b = bal = branch always
cond      offset
1110 1010 oooo oooo oooo oooo oooo oooo
   E    A

2. PC relative
if (cond)
    pc += offset;
```

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

3 steps to run an instruction



Fetch

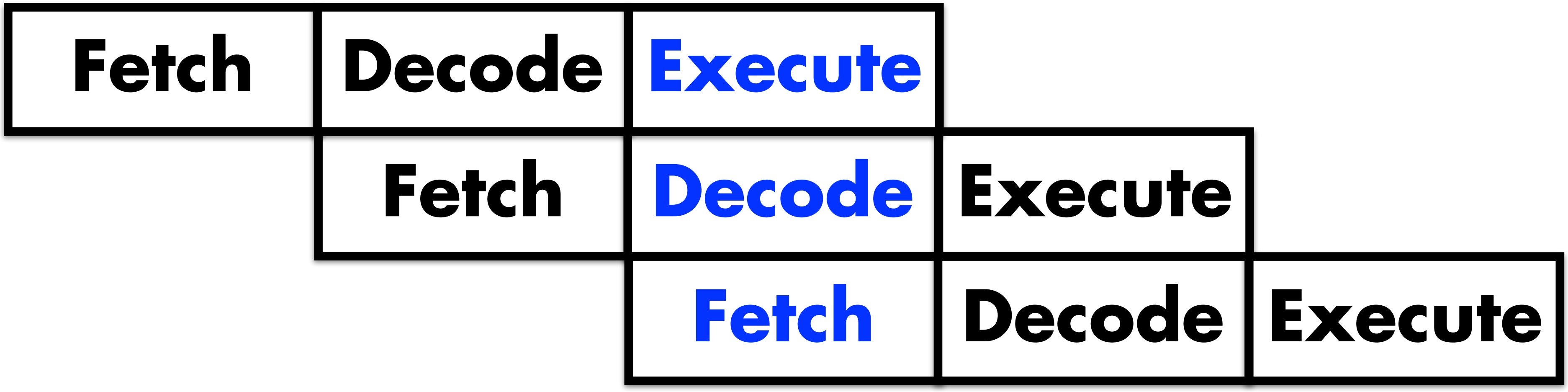
Decode

Execute

3 instructions takes 9 steps

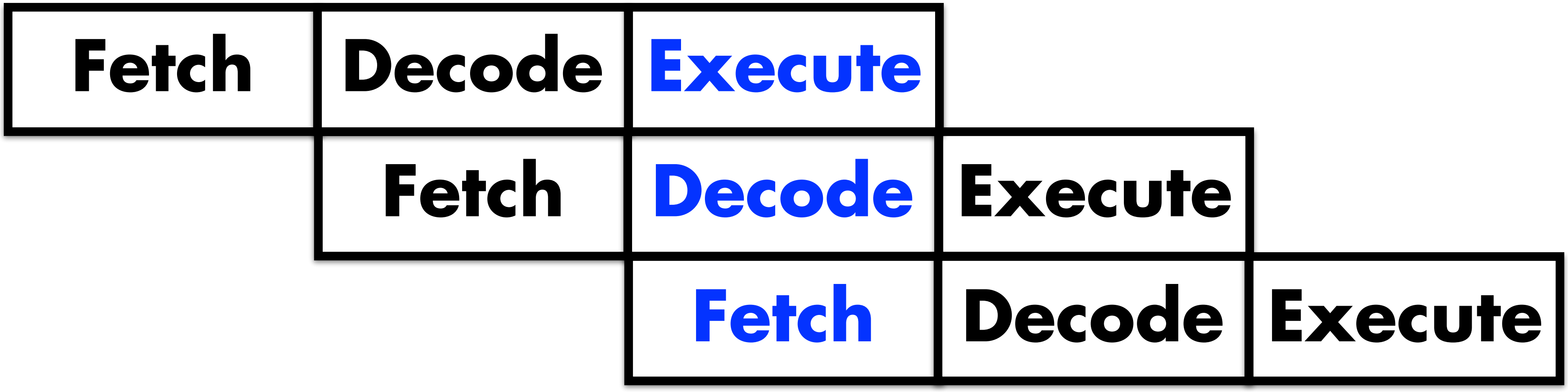
de	Execute	Fetch	Decode	Execute	Fetch	De

To speed things up, steps are overlapped ("pipelined")



Fetch	Decode	Execute		
	Fetch	Decode	Execute	
		Fetch	Decode	Execute

To speed things up, steps are overlapped ("pipelined")



Fetch	Decode	Execute		
	Fetch	Decode	Execute	
		Fetch	Decode	Execute

PC value in the executing instruction is equal to the pc value of the instruction being fetched - which is 2 instructions ahead (PC+8)

Assembly

```
00008000 <main>:
8000: e59f3010  ldr r3, [pc, #0x10]; 8018
8004: e3a02001  mov r2, #1
8008: e5832008  str r2, [r3, #0x08]
800c: e3a02601  mov r2, #0x100000
8010: e583201c  str r2, [r3, #0x1C]
8014: eafffffe  b   8014
8018: 20200000  .word   0x20200000

// ea = branch always
// fffffe = -2 (two's complement)
// pc += -2*4 + 8 (pc is a multiple of 4)
```

Indexed Loads

PC Relative Addressing

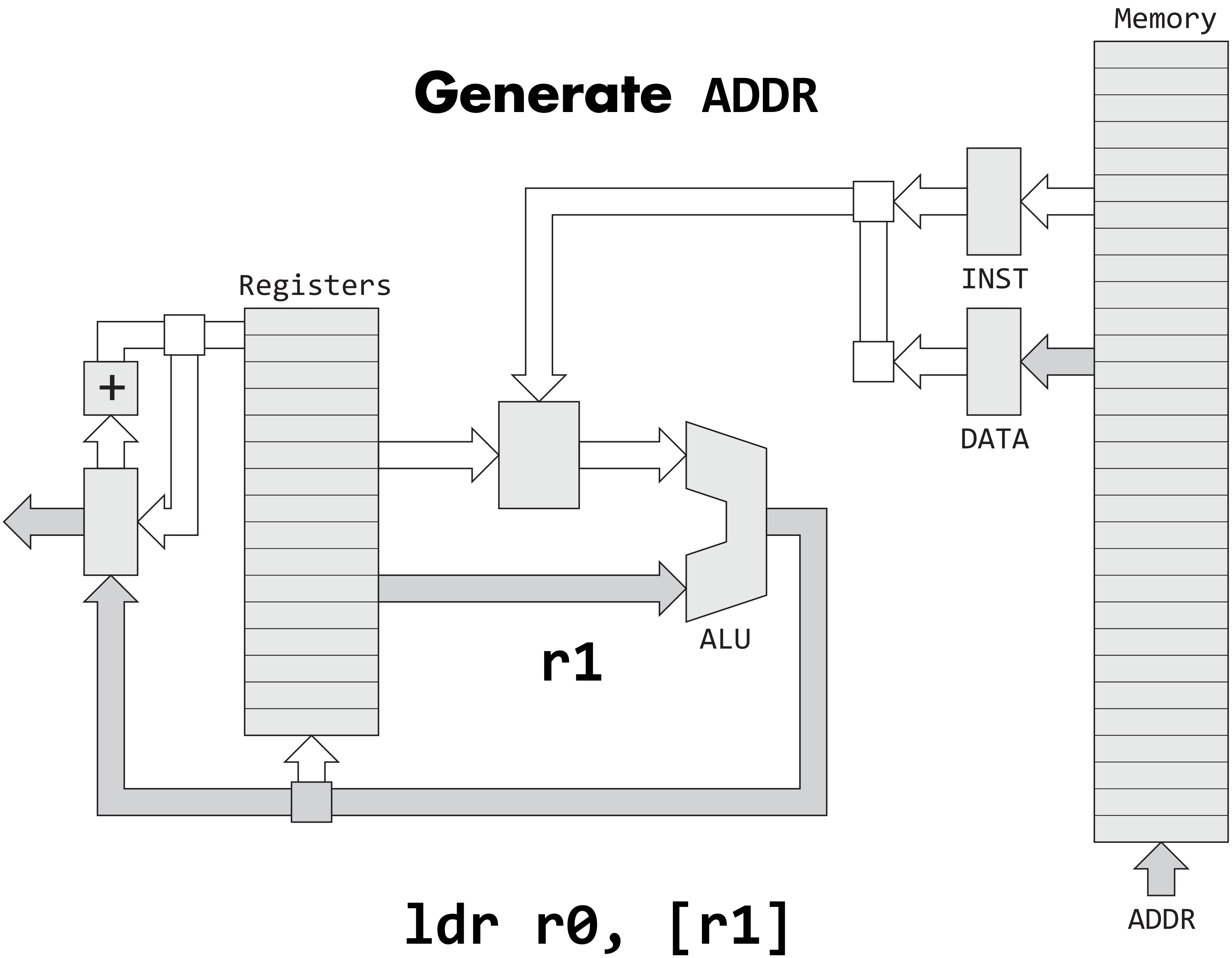
Assembly

```
00008000 <main>:
8000: e59f3010  ldr r3, [pc, #0x10]; 8018
8004: e3a02001  mov r2, #1
8008: e5832008  str r2, [r3, #0x08]
800c: e3a02601  mov r2, #0x100000
8010: e583201c  str r2, [r3, #0x1C]
8014: eafffffe  b   8014
8018: 20200000  .word   0x20200000

// indexed load and store
```

`r0 = mem[r1]`

Generate ADDR

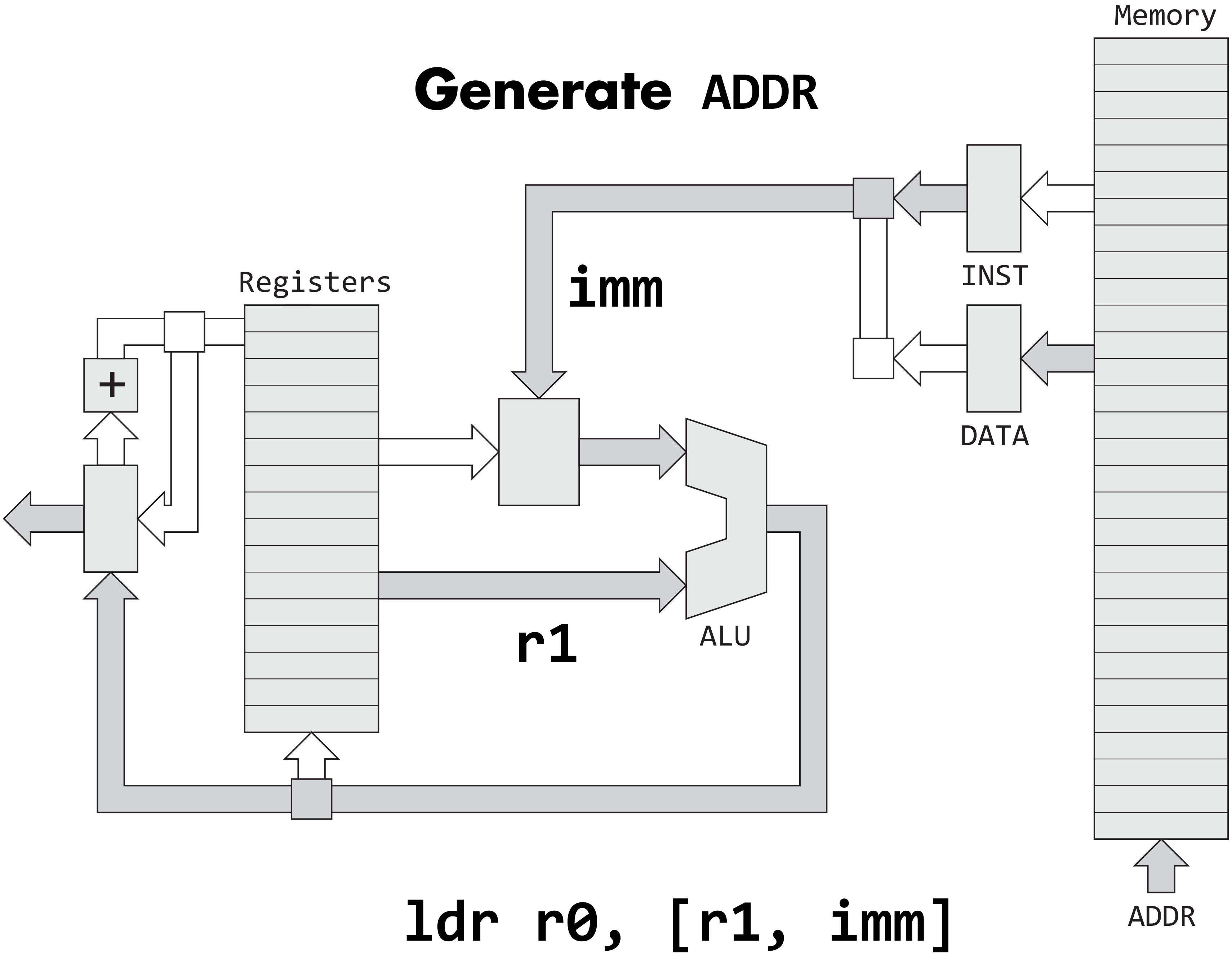


r1

`ldr r0, [r1]`

`r0 = mem[r1+imm]`

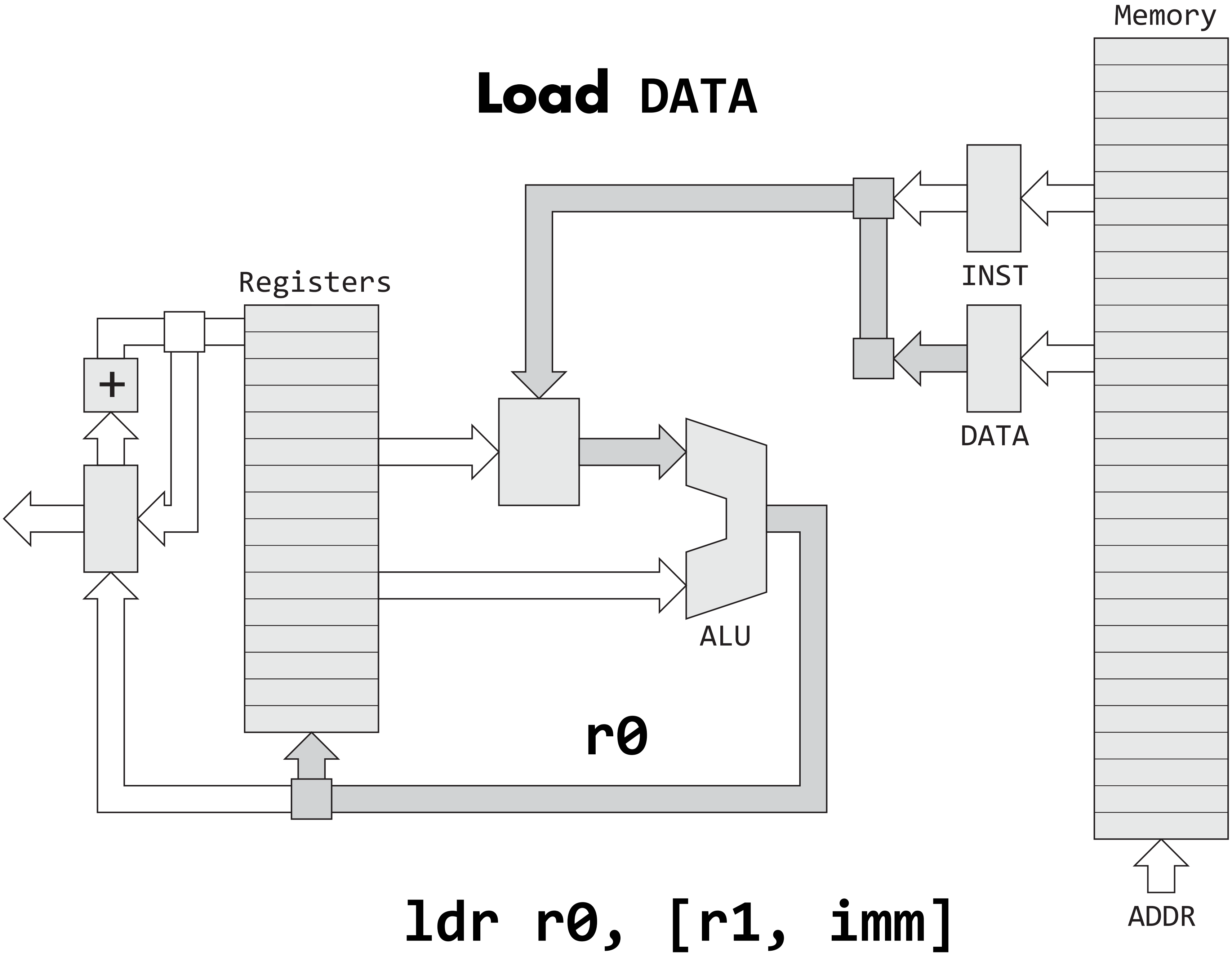
Generate ADDR



`ldr r0, [r1, imm]`

r0 = mem[r1+imm]

Load DATA



`ldr r0, [r1, imm]`

```
// "ldr ="
#define GPIO 0x20200000
ldr r0,=GPIO

// is converted to word in memory and
// a ld with pc relative indexing

8000: e59f3010  ldr r3, [pc, #0x10]; =8018
8004: e3a02001  mov r2, #1
8008: e5832008  str r2, [r3, #8]
800c: e3a02601  mov r2, #0x100000
8010: e583201c  str r2, [r3, #28]
8014: eafffffe  b   8014
8018: 20200000  .word   0x20200000
```

Compiler Optimization

Memory = Storage

Storage

- **Write** `data` **to** `mem[addr]`
- **Read** `mem[addr]`

The read value should equal the value written

```
int i, j;

i = 1;
i = 2;
j = i;

// can be optimized to

i = 2;
j = i;

// this is ok, unless someone else
// watching your writes
```

`volatile`

`volatile` **tells the compiler that another process or peripheral**

- **may read the value**
- **may write the value (this makes it volatile)**

As a result, `gcc` **cannot remove reads and writes to volatile variables**

Also cannot change the order of the reads to writes to volatile variables

Peripheral Registers

These registers are mapped into the address space of the processor (memory-mapped IO)

These registers may NOT behave as memory.

For example: Writing a 1 into a bit in a SET register causes 1 to be output; writing a 0 into a bit in SET register does not effect the output value. Writing a 1 to the CLR register, sets the output to 0; write a 0 to a clear register has no effect. Neither SET or CLR can be read. To read the current value use the LEV (level) register.

delay/delay.c

`-O2` **removes delay loop!**

Hack: Fix with `volatile`

`blink/blink.c`