05: Assembly and the stack

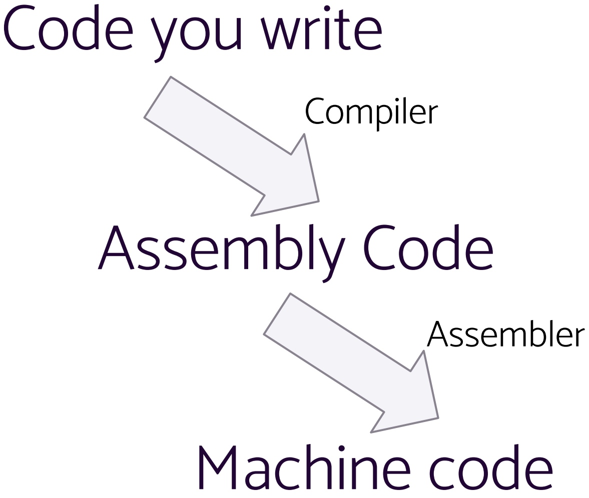
Review

- Sensors and actuators (I/O devices) can be analog or digital
- MCUs can read from/write to I/O devices
    - GPIO pins (for digital signals and PWM)
    - DACs, ADCs (for analog signals)
    - This enables us to use software to interact with the physical world

MCUs are varied

But knowing the theory of how a CPU, peripherals, memory work gives context to reading a data sheet

How software you write becomes code running on an MCU



Code you write

Compiler

Assembly Code

Assembler

Machine code

How a microprocessor executes machine code

Fetch - fetch next instruction from memory

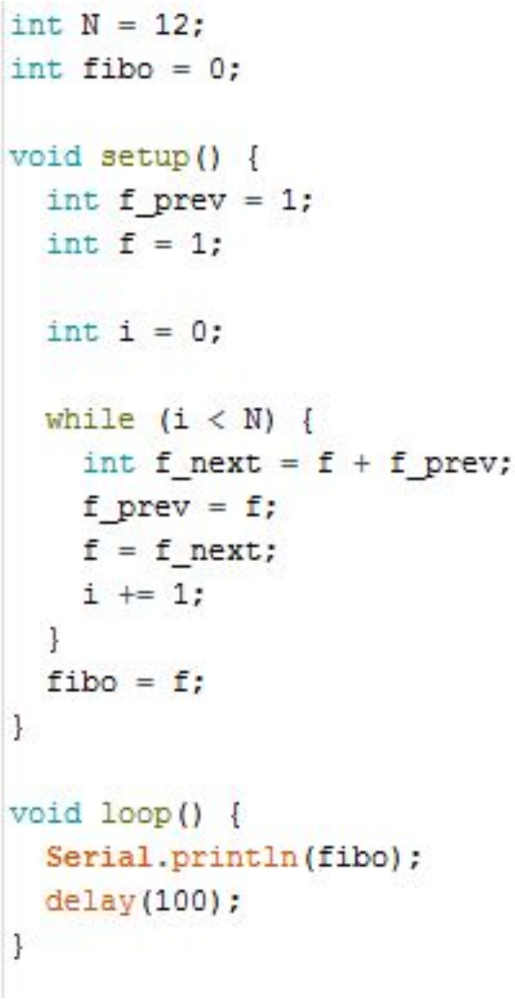
Decode - decode instruction

Execute - perform computation

- ALU (arithmetic logic unit): add, subtract, negate, bit operations
- Shift: used in multiplication/division

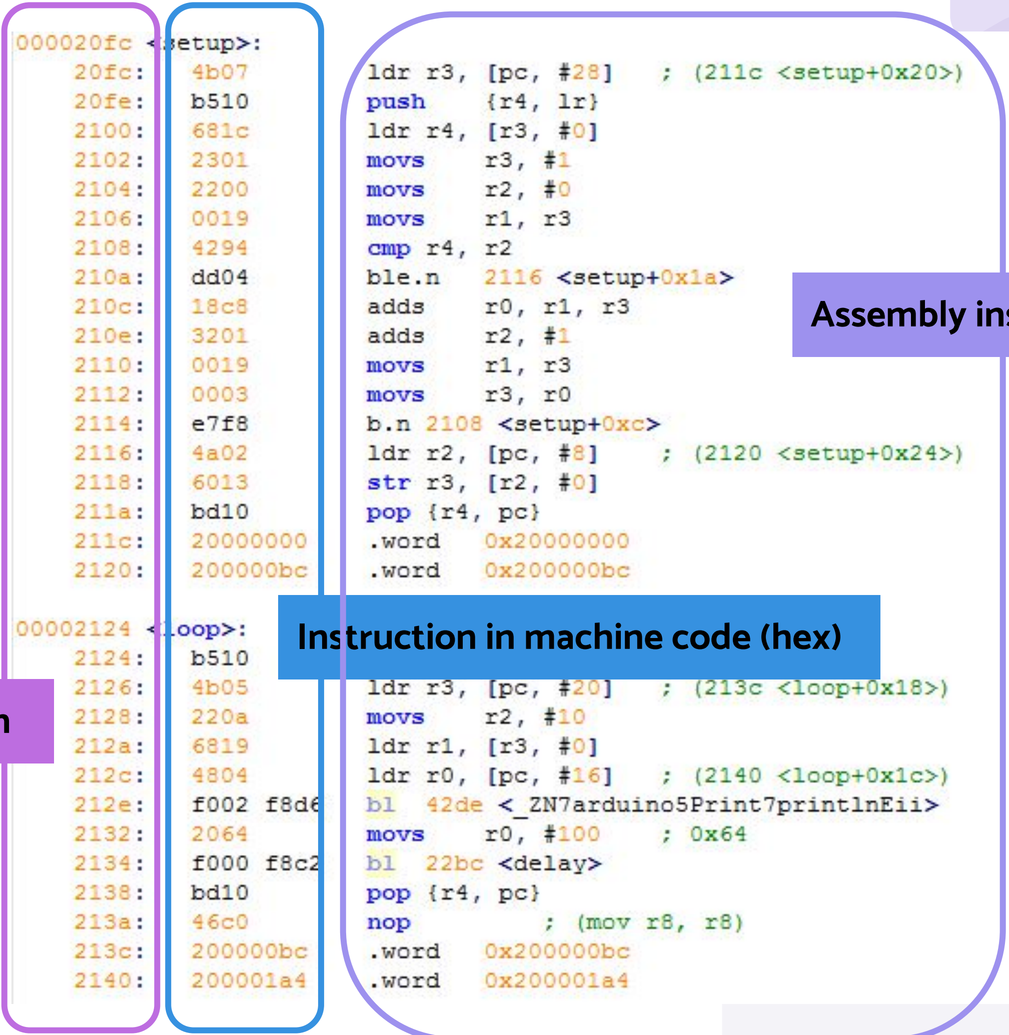
Memory access - read or write registers

Program



```
int N = 12;
int fibo = 0;

void setup() {
  int f_prev = 1;
  int f = 1;

  int i = 0;

  while (i < N) {
    int f_next = f + f_prev;
    f_prev = f;
    f = f_next;
    i += 1;
  }
  fibo = f;
}

void loop() {
  Serial.println(fibo);
  delay(100);
}
```

Assembly



```
000020fc <setup>:
    20fc:   4b07        ldr r3, [pc, #28]     ; (211c <setup+0x20>)
    20fe:   b510        push      {r4, lr}
    2100:   681c        ldr r4, [r3, #0]
    2102:   2301        movs      r3, #1
    2104:   2200        movs      r2, #0
    2106:   0019        movs      r1, r3
    2108:   4294        cmp r4, r2
    210a:   dd04        ble.n     2116 <setup+0x1a>
    210c:   18c8        adds      r0, r1, r3
    210e:   3201        adds      r2, #1
    2110:   0019        movs      r1, r3
    2112:   0003        movs      r3, r0
    2114:   e7f8        b.n 2108 <setup+0xc>
    2116:   4a02        ldr r2, [pc, #8]      ; (2120 <setup+0x24>)
    2118:   6013        str r3, [r2, #0]
    211a:   bd10        pop {r4, pc}
    211c:   20000000    .word     0x20000000
    2120:   200000bc    .word     0x200000bc

00002124 <loop>:
    2124:   b510
    2126:   4b05        ldr r3, [pc, #20]     ; (213c <loop+0x18>)
    2128:   220a        movs      r2, #10
    212a:   6819        ldr r1, [r3, #0]
    212c:   4804        ldr r0, [pc, #16]     ; (2140 <loop+0x1c>)
    212e:   f002 f8d6   bl  42de <_ZN7arduino5Print7printlnEii>
    2132:   2064        movs      r0, #100    ; 0x64
    2134:   f000 f8c2   bl  22bc <delay>
    2138:   bd10        pop {r4, pc}
    213a:   46c0        nop             ; (mov r8, r8)
    213c:   200000bc    .word     0x200000bc
    2140:   200001a4    .word     0x200001a4
```

Memory address of instruction

Instruction in machine code (hex)

Assembly instructions

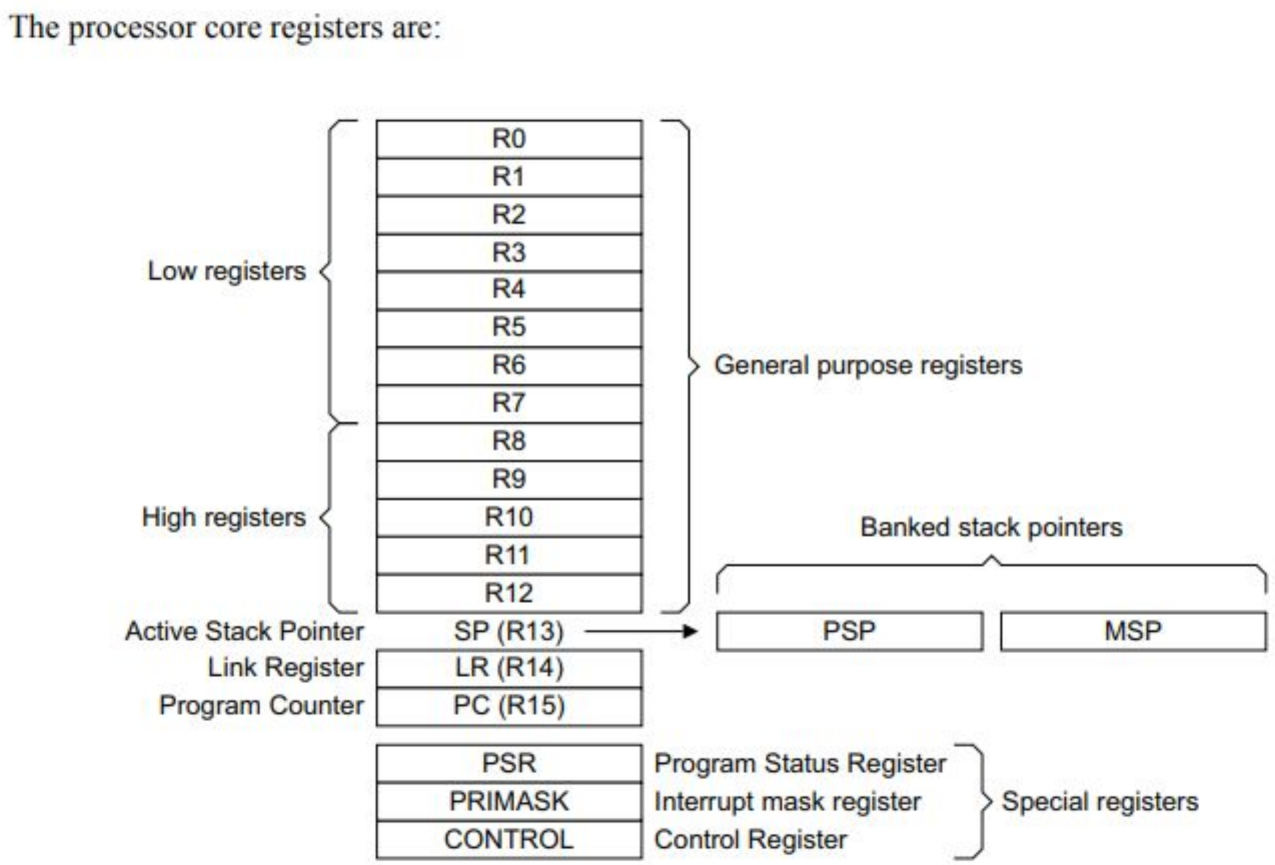
Resources used in this presentation

[ARM Cortex M0+ devices generic user guide]

[ARMv6-M Architecture reference manual]

Registers

- Small pieces of fast memory
- Usually 8-, 16-, 32- or 64-bits
- Many purposes on CPUs and MCUs:
    - Storing temporary data for execution
    - Addressing memory
    - Configuring peripherals (Lab 3)



The processor core registers are:

Low registers

High registers

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12

General purpose registers

Active Stack Pointer SP (R13) → Banked stack pointers: PSP MSP

Link Register LR (R14)

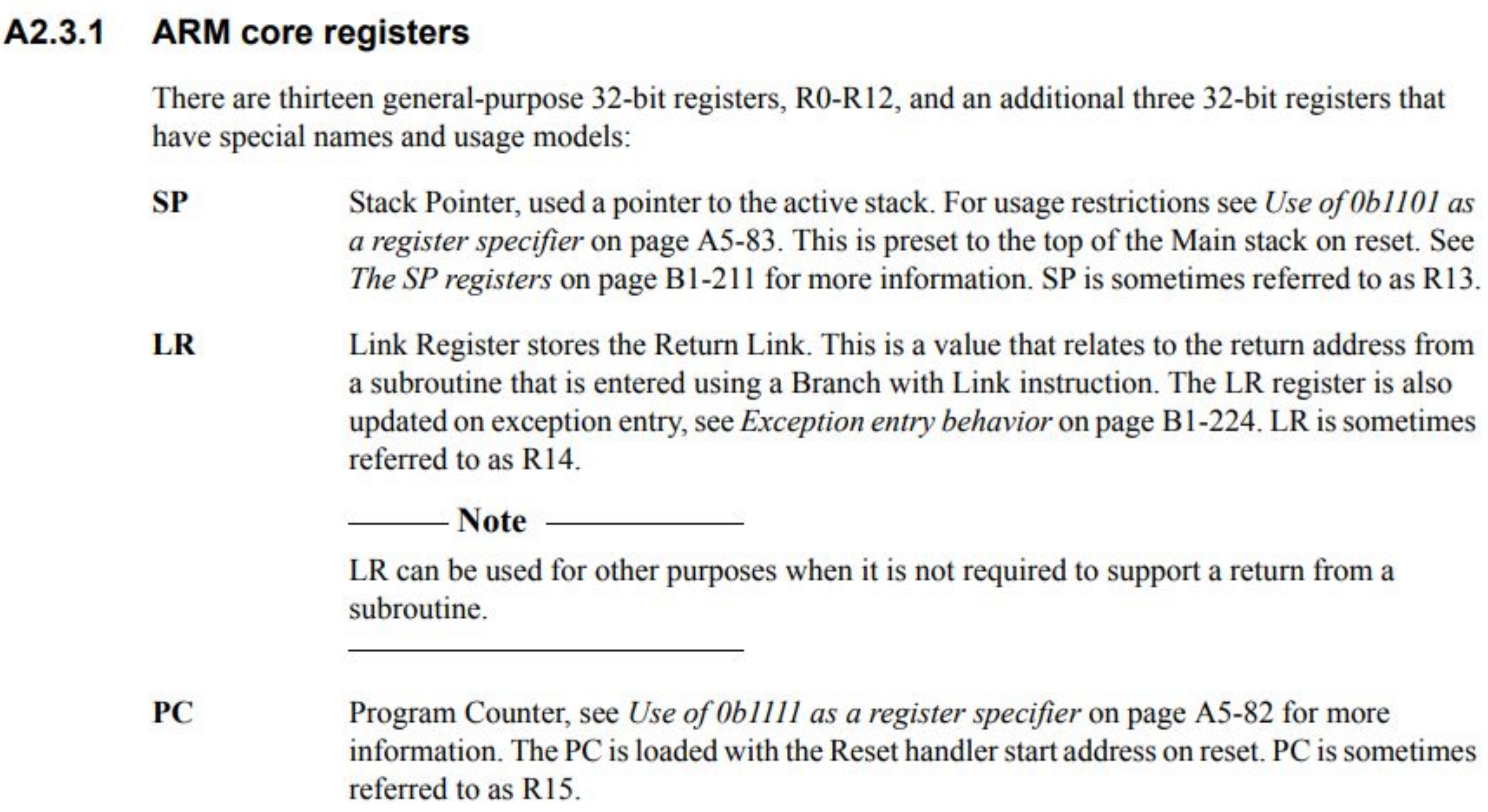
Program Counter PC (R15)

PSR Program Status Register

PRIMASK Interrupt mask register

CONTROL Control Register

Special registers



A2.3.1 ARM core registers

There are thirteen general-purpose 32-bit registers, R0-R12, and an additional three 32-bit registers that have special names and usage models:

SP Stack Pointer, used a pointer to the active stack. For usage restrictions see *Use of 0b1101 as a register specifier* on page A5-83. This is preset to the top of the Main stack on reset. See *The SP registers* on page B1-211 for more information. SP is sometimes referred to as R13.

LR Link Register stores the Return Link. This is a value that relates to the return address from a subroutine that is entered using a Branch with Link instruction. The LR register is also updated on exception entry, see *Exception entry behavior* on page B1-224. LR is sometimes referred to as R14.

Note

LR can be used for other purposes when it is not required to support a return from a subroutine.

PC Program Counter, see *Use of 0b1111 as a register specifier* on page A5-82 for more information. The PC is loaded with the Reset handler start address on reset. PC is sometimes referred to as R15.

Stack

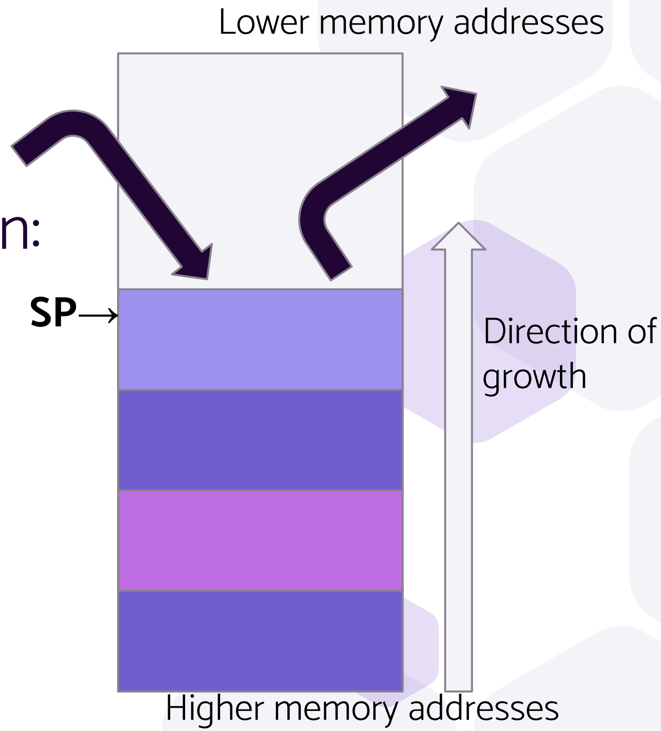
LIFO (last-in, first-out) data structure

Keeps track of information for execution:

- Local variables
- Return pointers

Grows “downward”

Stack Pointer (SP) points to latest value



Cortex M0+ stack operations

push *reglist* - push the registers in *reglist* onto the stack (highest value registers pushed first), decrements stack pointer

pop *reglist* - pop the values on the stack into the registers in *reglist* (lowest value registers popped first)

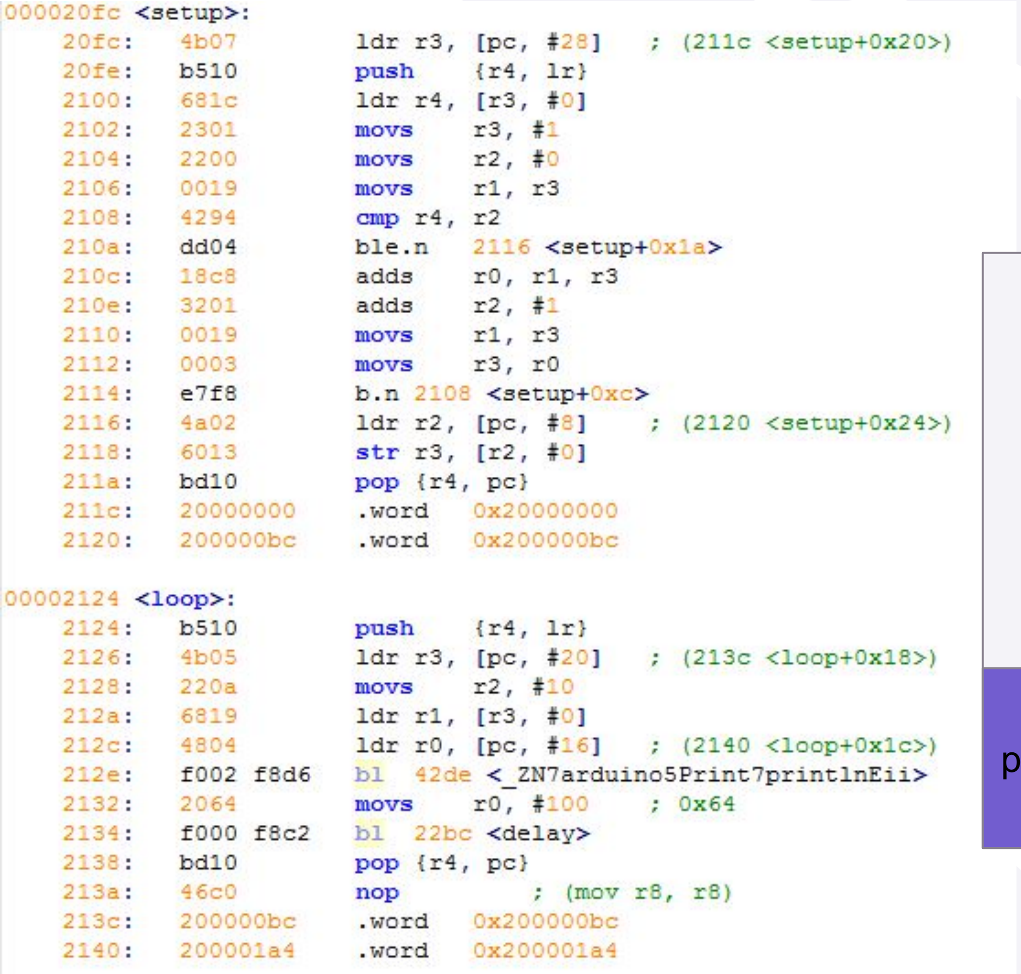
if SP is in *reglist*, branch to where SP is pointing after pop

Loads and stores

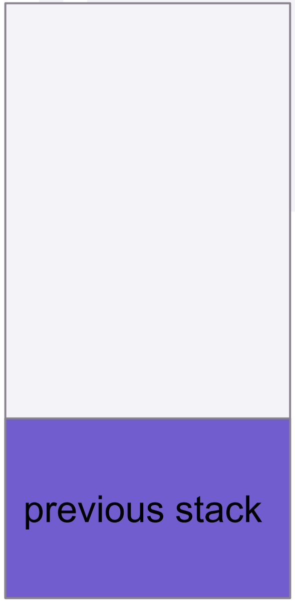
An instruction like `ldr r1 [r2, #8]` means:

- Add 8 to the value in register r2
- Interpret the result as a memory address
- Take the value stored at that memory address and put it in r1

(Similar with `str`, which is for storing values in registers at memory addresses)



```
000020fc <setup>:
    20fc:   4b07         ldr r3, [pc, #28]   ; (211c <setup+0x20>)
    20fe:   b510         push    {r4, lr}
    2100:   681c         ldr r4, [r3, #0]
    2102:   2301         movs    r3, #1
    2104:   2200         movs    r2, #0
    2106:   0019         movs    r1, r3
    2108:   4294         cmp r4, r2
    210a:   dd04         ble.n   2116 <setup+0x1a>
    210c:   18c8         adds    r0, r1, r3
    210e:   3201         adds    r2, #1
    2110:   0019         movs    r1, r3
    2112:   0003         movs    r3, r0
    2114:   e7f8         b.n 2108 <setup+0xc>
    2116:   4a02         ldr r2, [pc, #8]    ; (2120 <setup+0x24>)
    2118:   6013         str r3, [r2, #0]
    211a:   bd10         pop {r4, pc}
    211c:   20000000     .word   0x20000000
    2120:   200000bc     .word   0x200000bc

00002124 <loop>:
    2124:   b510         push    {r4, lr}
    2126:   4b05         ldr r3, [pc, #20]   ; (213c <loop+0x18>)
    2128:   220a         movs    r2, #10
    212a:   6819         ldr r1, [r3, #0]
    212c:   4804         ldr r0, [pc, #16]   ; (2140 <loop+0x1c>)
    212e:   f002 f8d6    bl  42de <_ZN7arduino5Print7printlnEii>
    2132:   2064         movs    r0, #100    ; 0x64
    2134:   f000 f8c2    bl  22bc <delay>
    2138:   bd10         pop {r4, pc}
    213a:   46c0         nop         ; (mov r8, r8)
    213c:   200000bc     .word   0x200000bc
    2140:   200001a4     .word   0x200001a4
```



R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	

LR	

Passing parameters?

Multiple conventions

- Pass on stack
- Pass as registers
- Combination (In gcc: first four arguments passed in registers, then stack)

What does the code that we looked at do?

Why learn about assembly when compilers exist?

Why

Machine code mystery

```
20fe:    b510
```

[redacted]

Decode an instruction like b510

How a microprocessor executes machine code

Fetch - fetch next instruction from memory

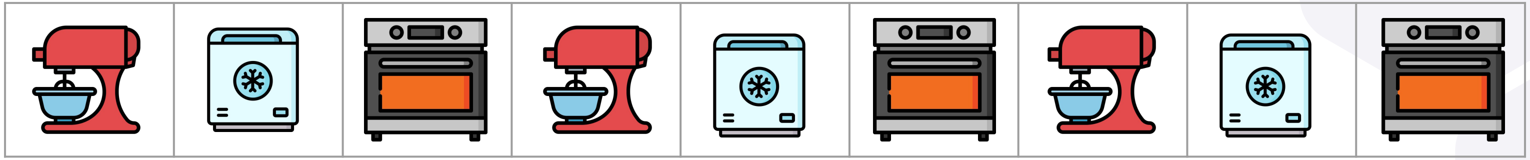
Decode - decode instruction

Execute - perform computation

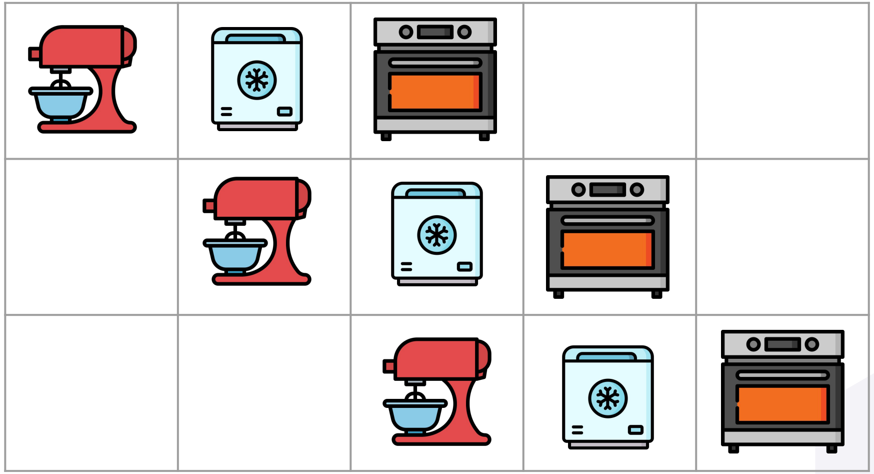
> ALU (arithmetic logic unit): add, subtract, negate, bit operations
>
> Shift: used in multiplication/division

Memory access - read or write registers

Pipelining



vs



images: Flaticon.com

Pipeline hazards (dependencies)

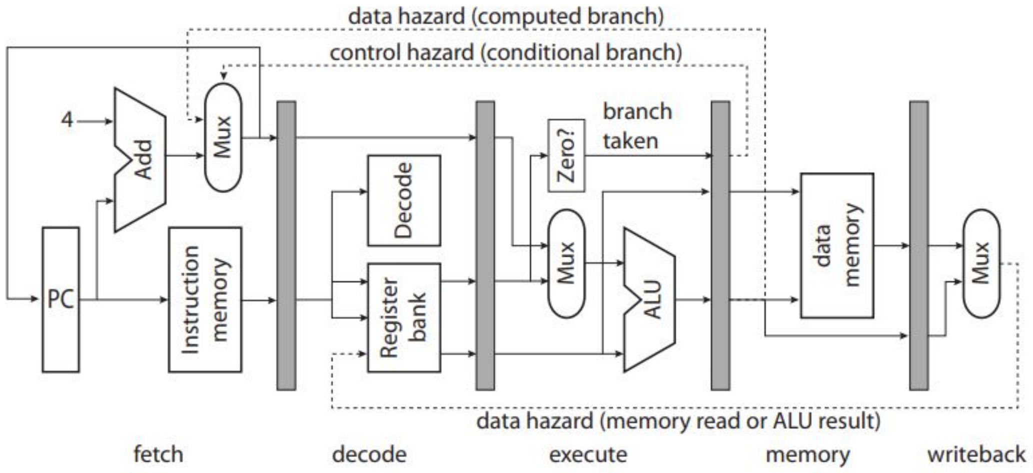


Figure 8.2: Simple pipeline (after Patterson and Hennessy (1996)).

Cortex-M0+

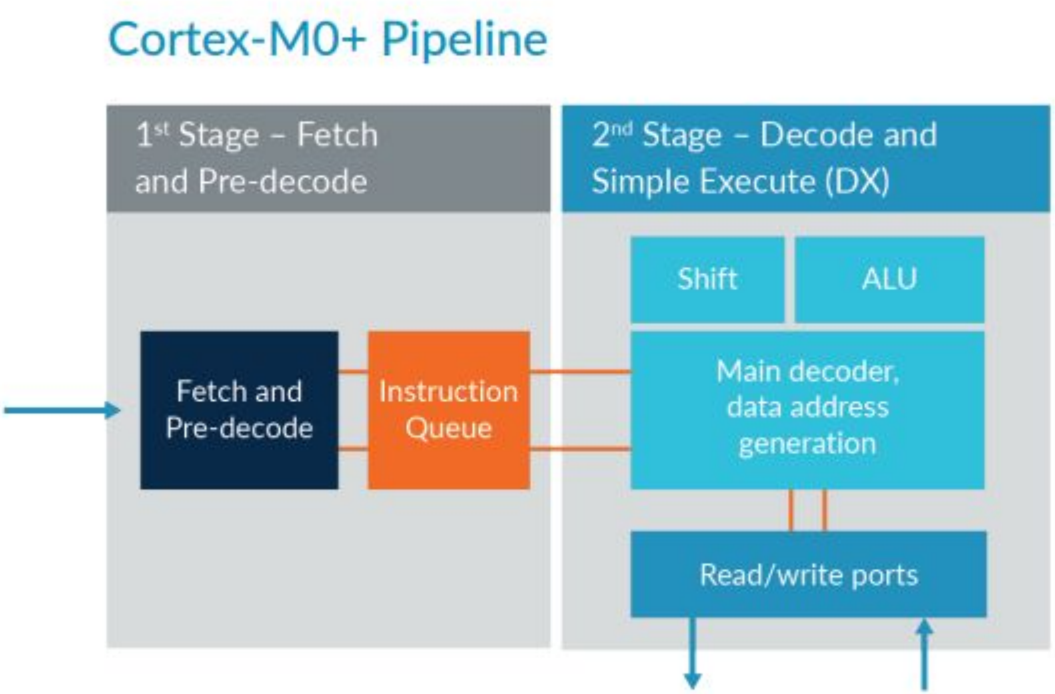

Cortex-M0+ Pipeline
1st Stage – Fetch and Pre-decode
2nd Stage – Decode and Simple Execute (DX)
Shift
ALU
Fetch and Pre-decode
Instruction Queue
Main decoder, data address generation
Read/write ports

Image source