Embedded Systems (3)

- Will start at 15:10
- PDF of this slide is available via ScombZ

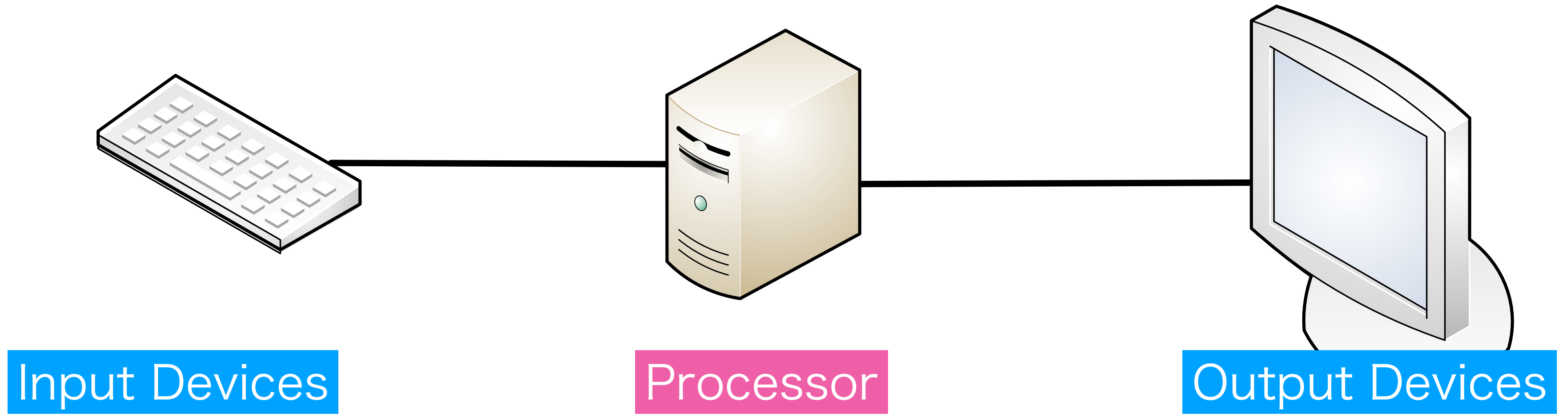
Hiroki Sato <i048219@shibaura-it.ac.jp>

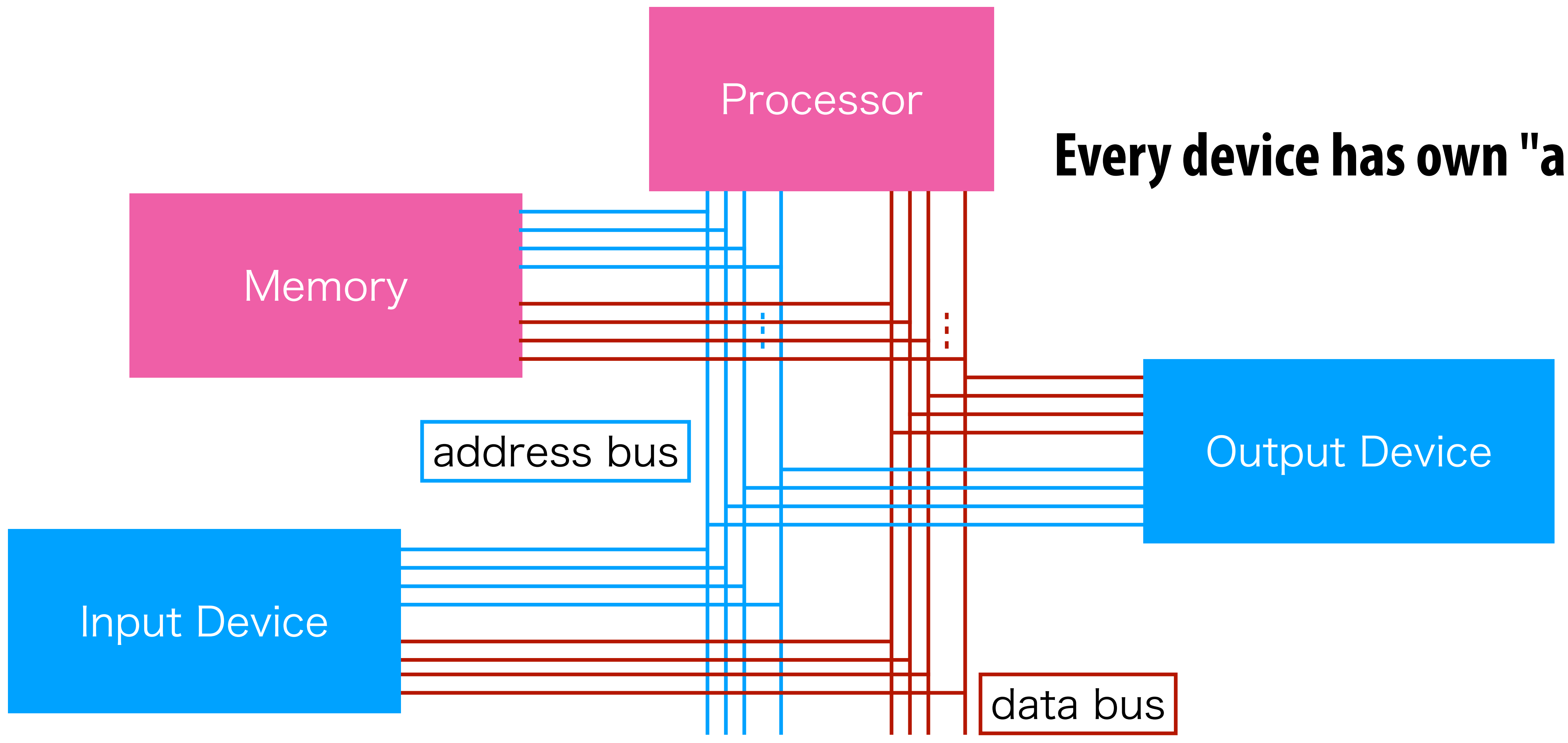
15:10-16:50 on Wednesday

Targets At a Glance

- **What you will learn today: Hardware and Software Components**
  - How a processor works in details
    - Instructions and machine code
    - Writing a C/C++ program
    - Debugging
  - Your 2nd project: 6 LEDs blinker
    - Wiring with breadboard
    - Debugging in action

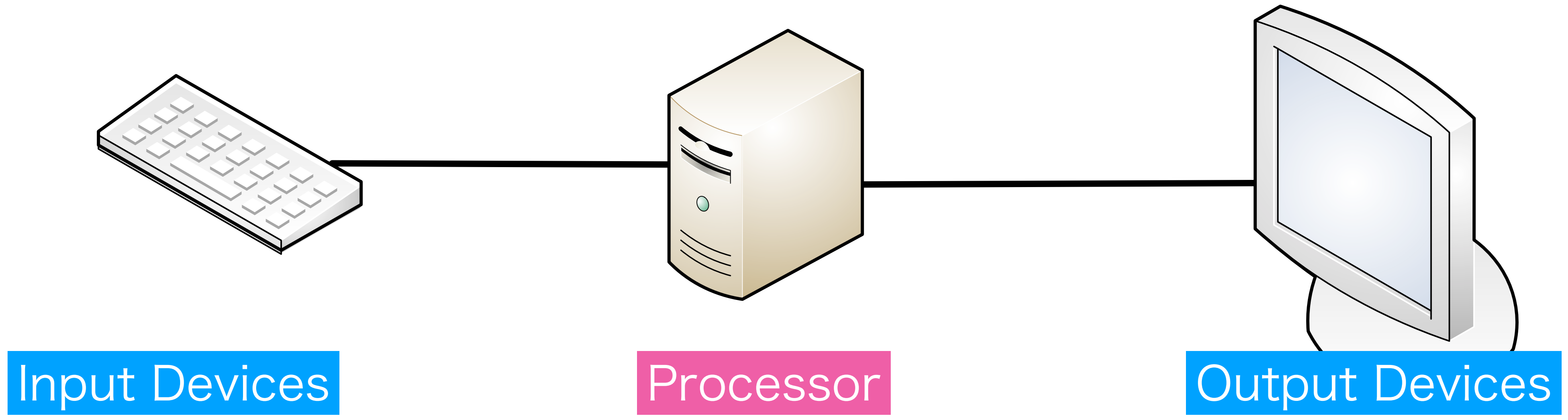
Model of Computer System

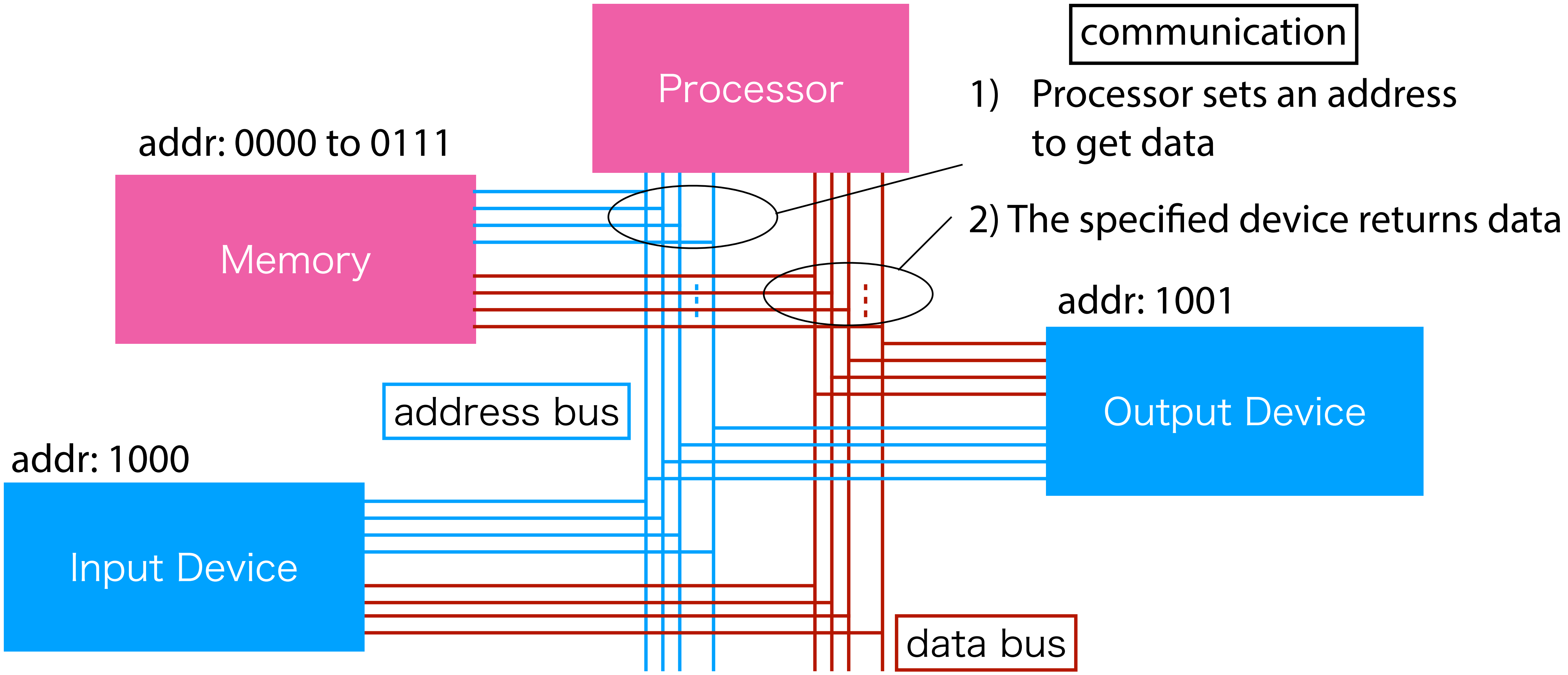




Every device has own "address"

Model of Computer System



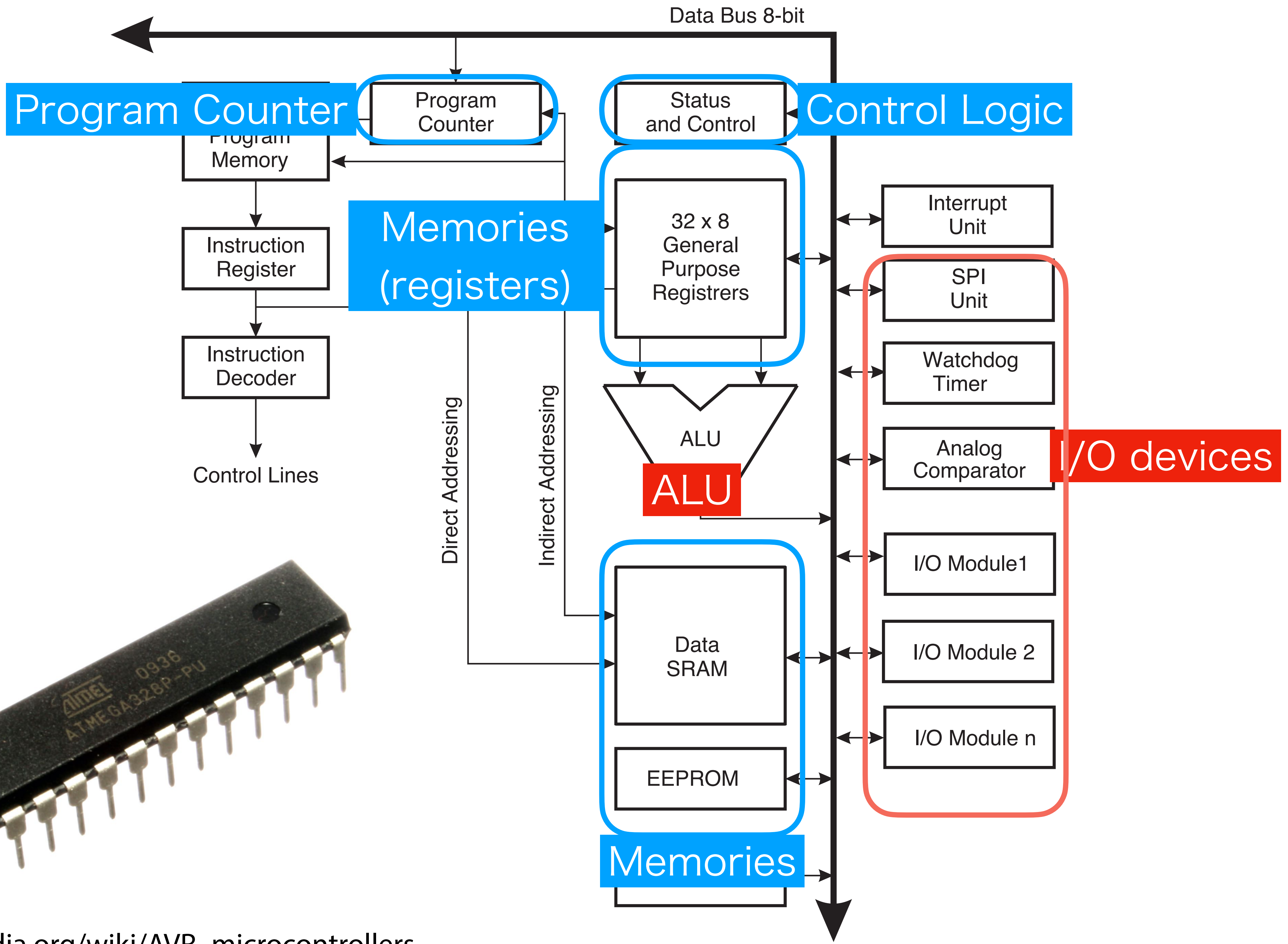


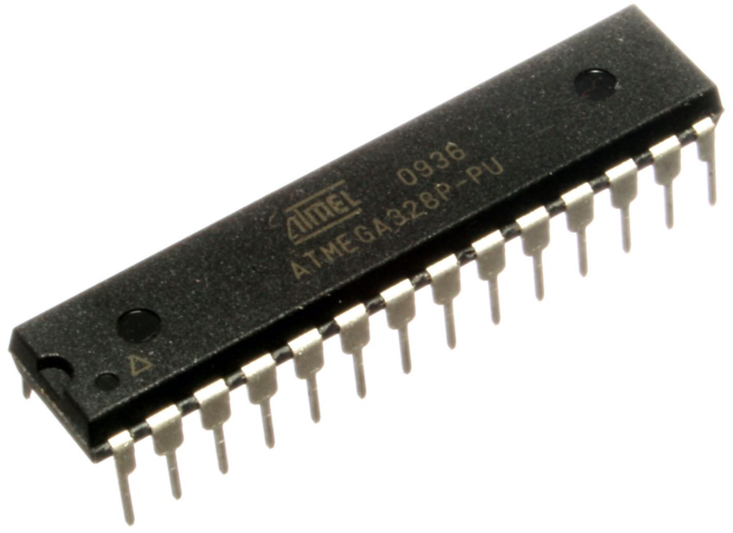
communication

1) Processor sets an address to get data

2) The specified device returns data

ATmega328P





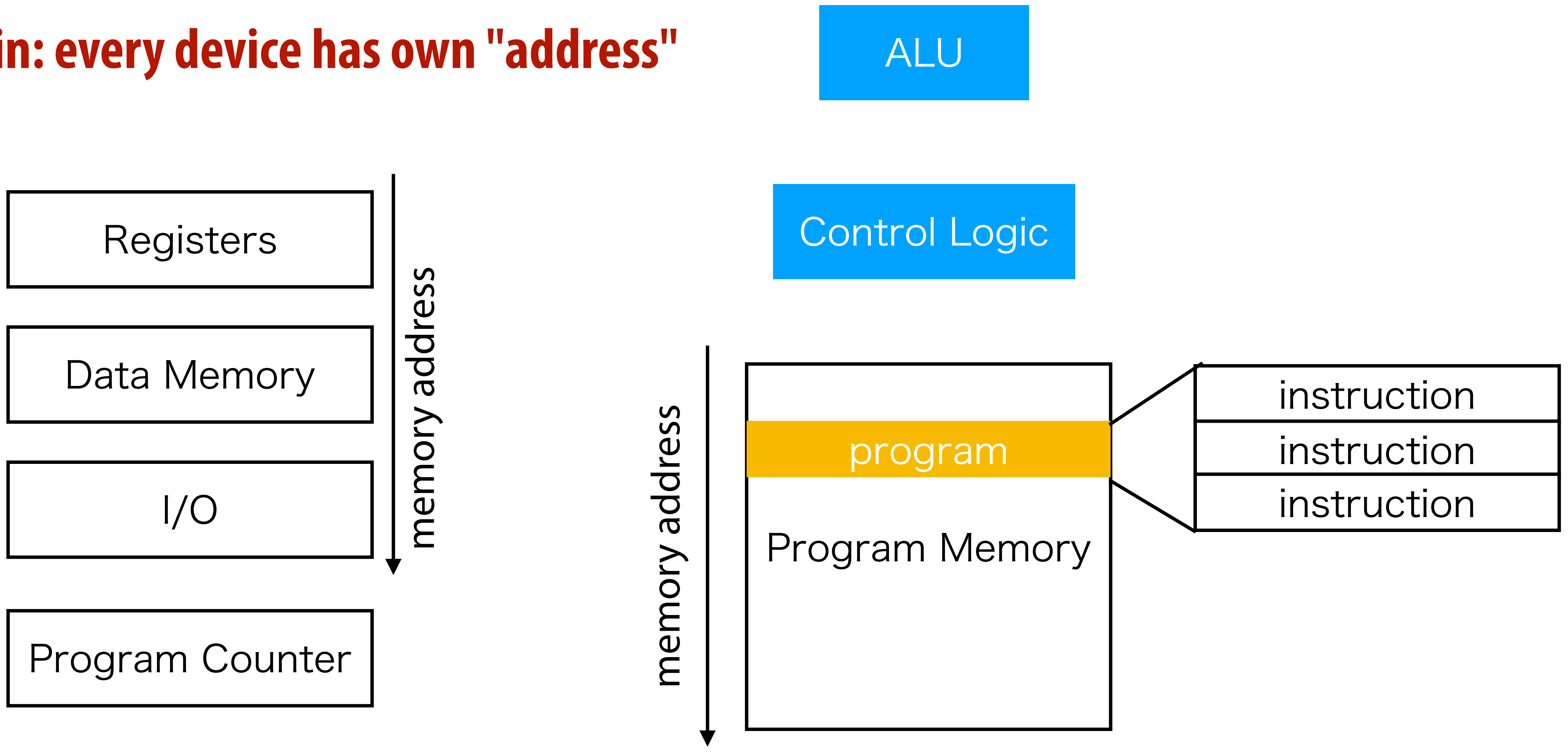
https://en.wikipedia.org/wiki/AVR_microcontrollers

http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf

How Processor Works

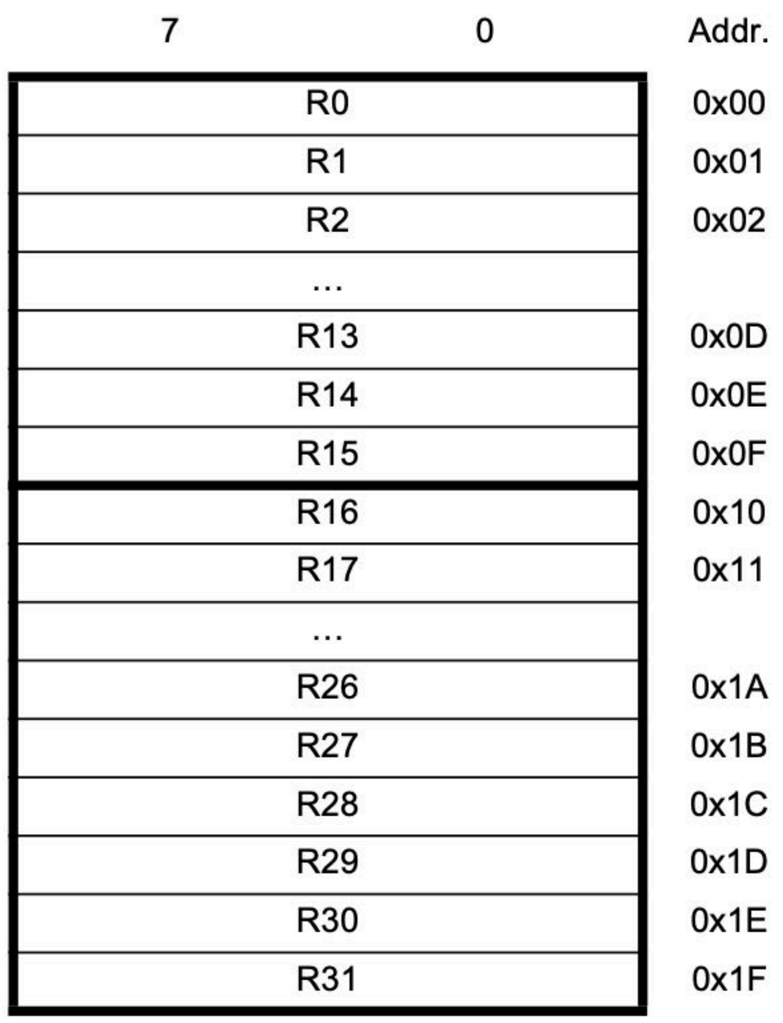
- **Components of a processor from software perspective**

Again: every device has own "address"



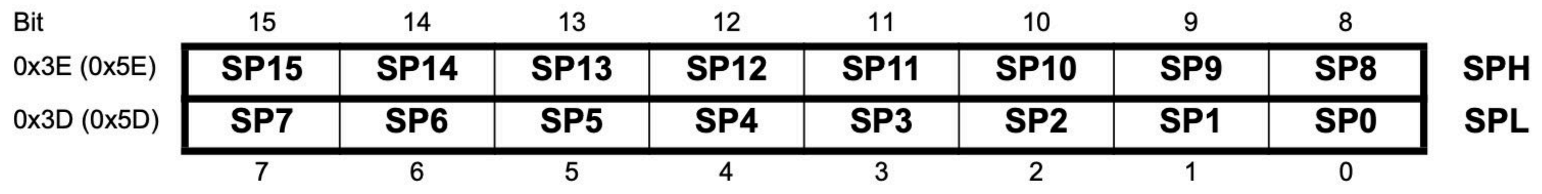
Registers

- **Registers: working memory inside the processor**



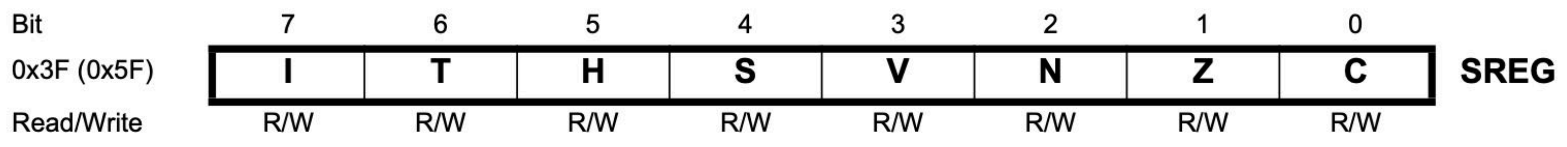
7 … 0	Addr.
R0	0x00
R1	0x01
R2	0x02
…	
R13	0x0D
R14	0x0E
R15	0x0F
R16	0x10
R17	0x11
…	
R26	0x1A
R27	0x1B
R28	0x1C
R29	0x1D
R30	0x1E
R31	0x1F

General-Purpose Registers (8-bit x 32)



Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	

Stack Pointer Registers (8-bit x 2)



Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

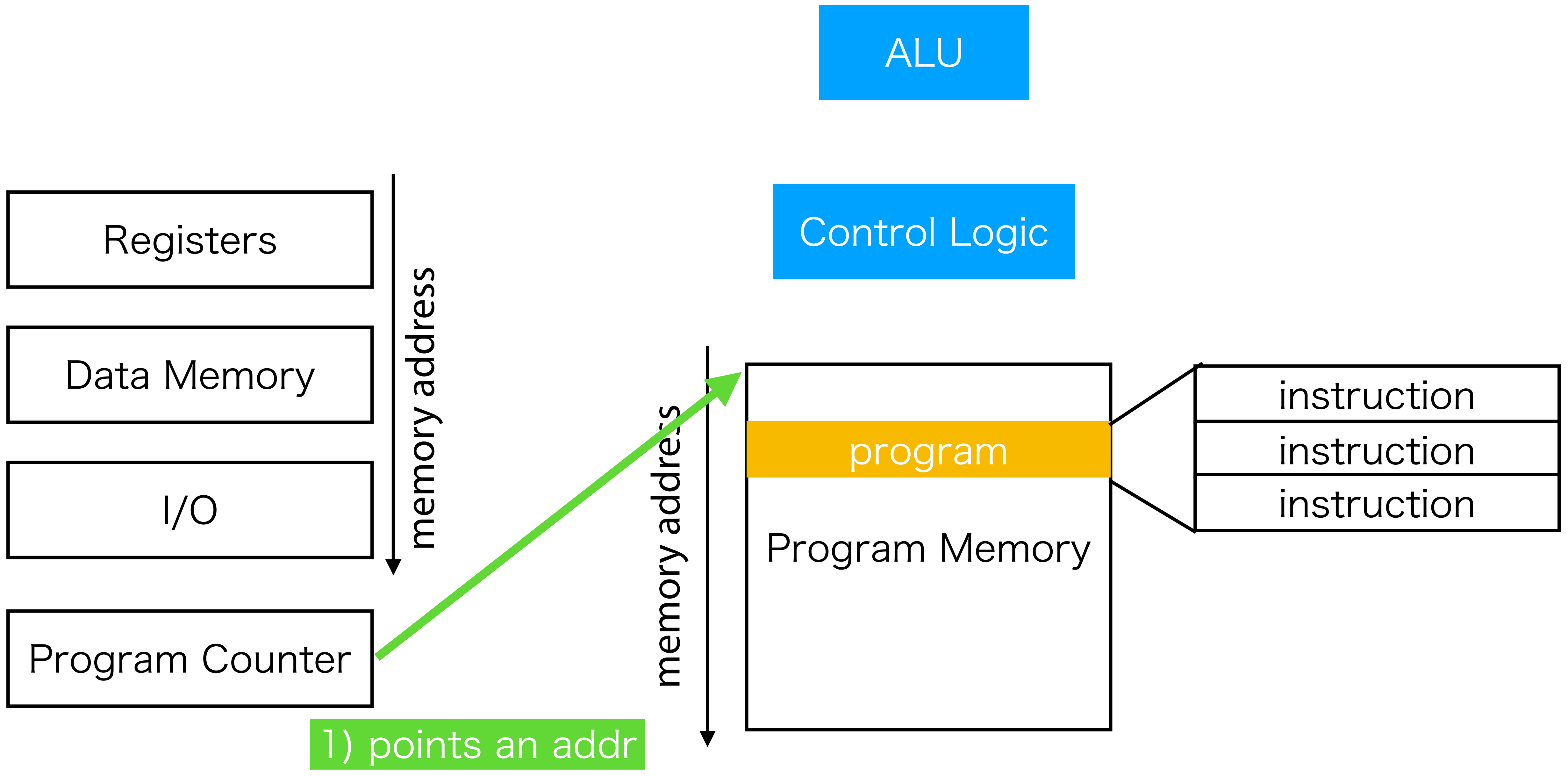
Status Register (8-bit)

Program Counter

PC Register (22-bit)

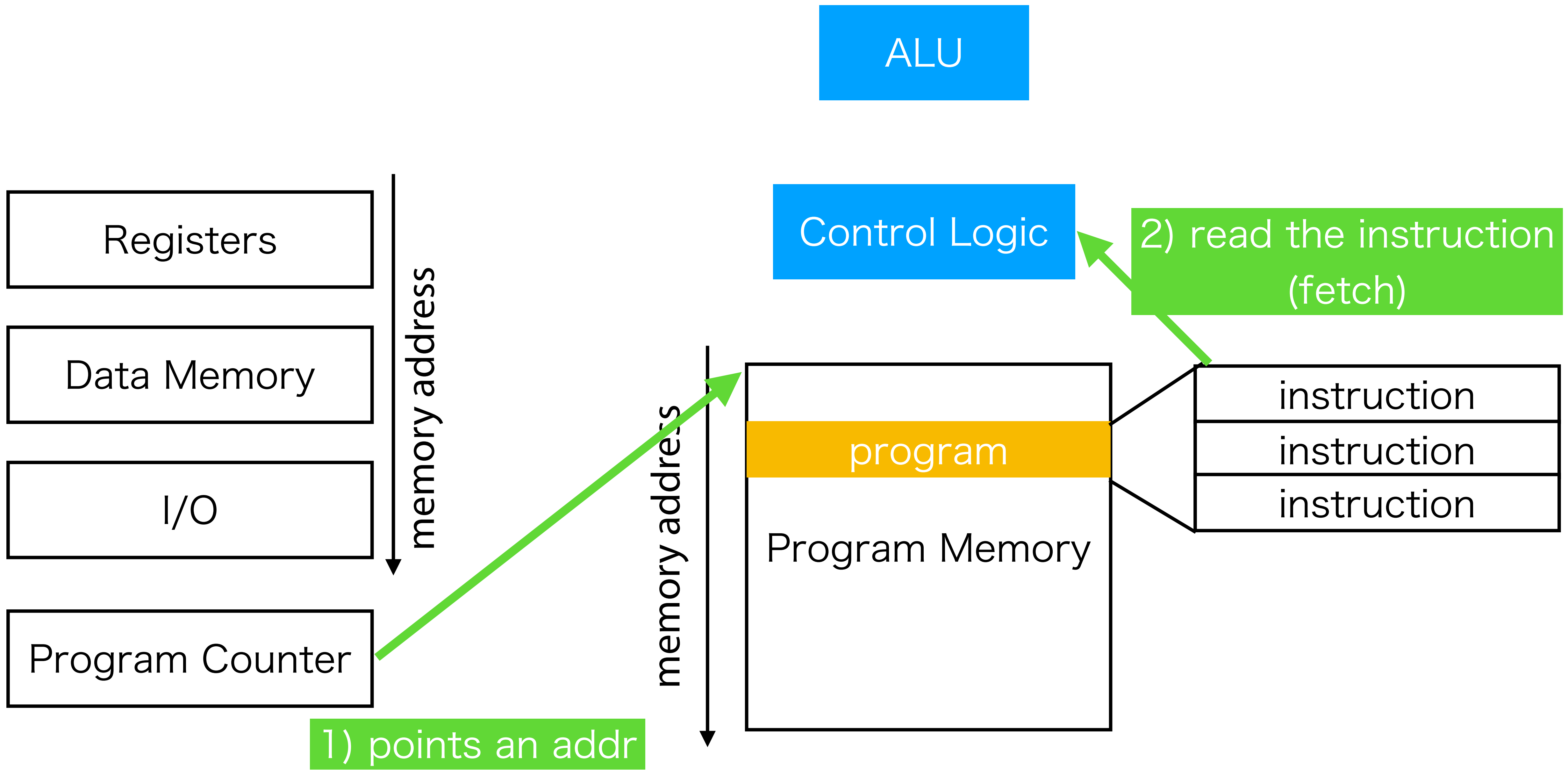
How Processor Works

- **Components of a processor from software perspective**



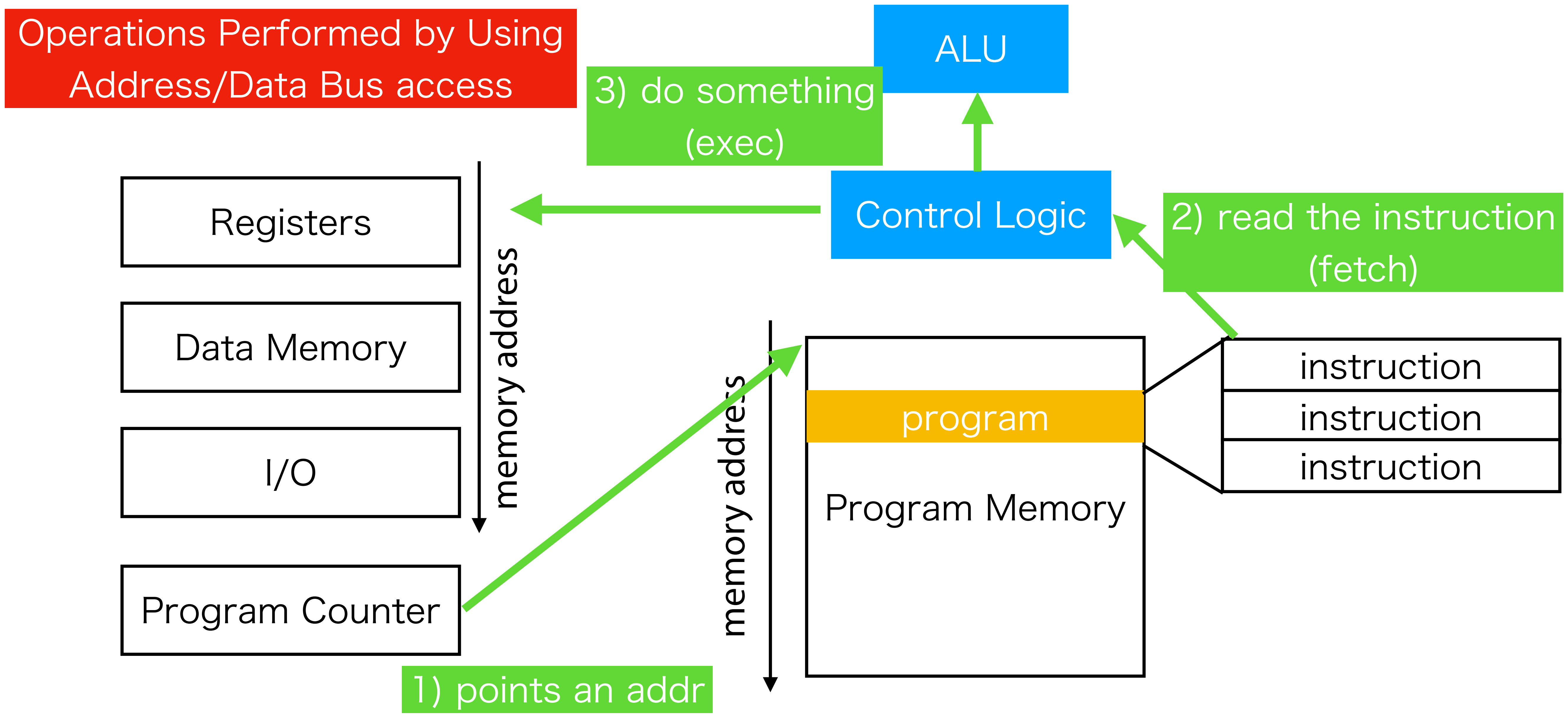
How Processor Works

- **Components of a processor from software perspective**



How Processor Works

- **Components of a processor from software perspective**



How Processor Works

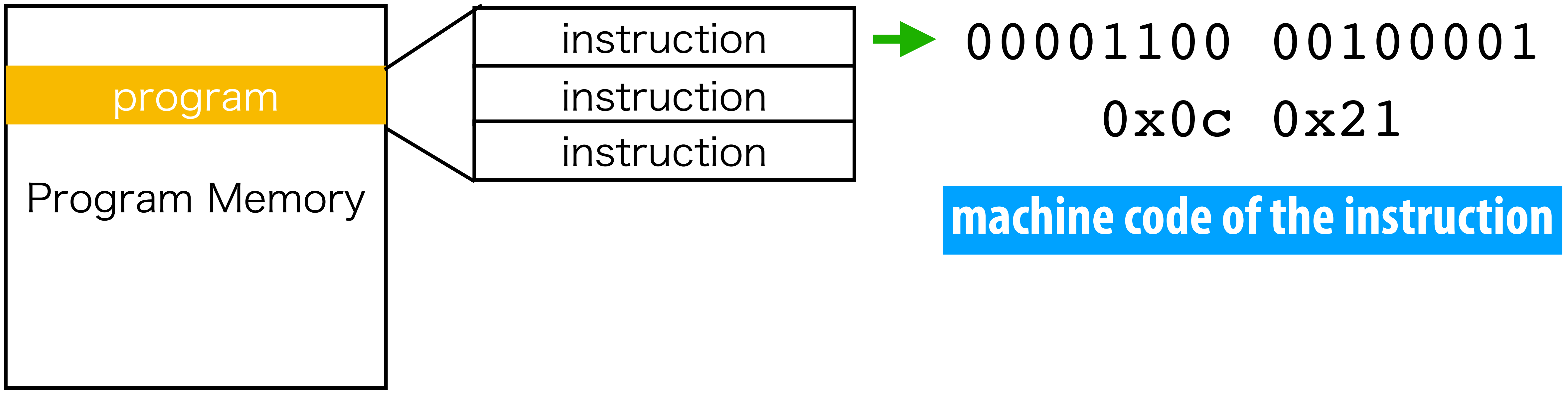
- **Instruction set of the processor (p.625 of the datasheet)**
  - **131 instructions**
    - Arithmetic/logical/bit operations
    - Data transfer
    - Branch
    - Processor control

AVR instruction set manual

`http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf`

How A Program Looks Like

- **A program = series of instructions**
- **An instruction = data for the control logic**



How A Program Looks Like

- **Example:** Arithmetic/logical/bit operations
  - Unary or binary operations on registers
  - addition = ADD, ADC, ADIW

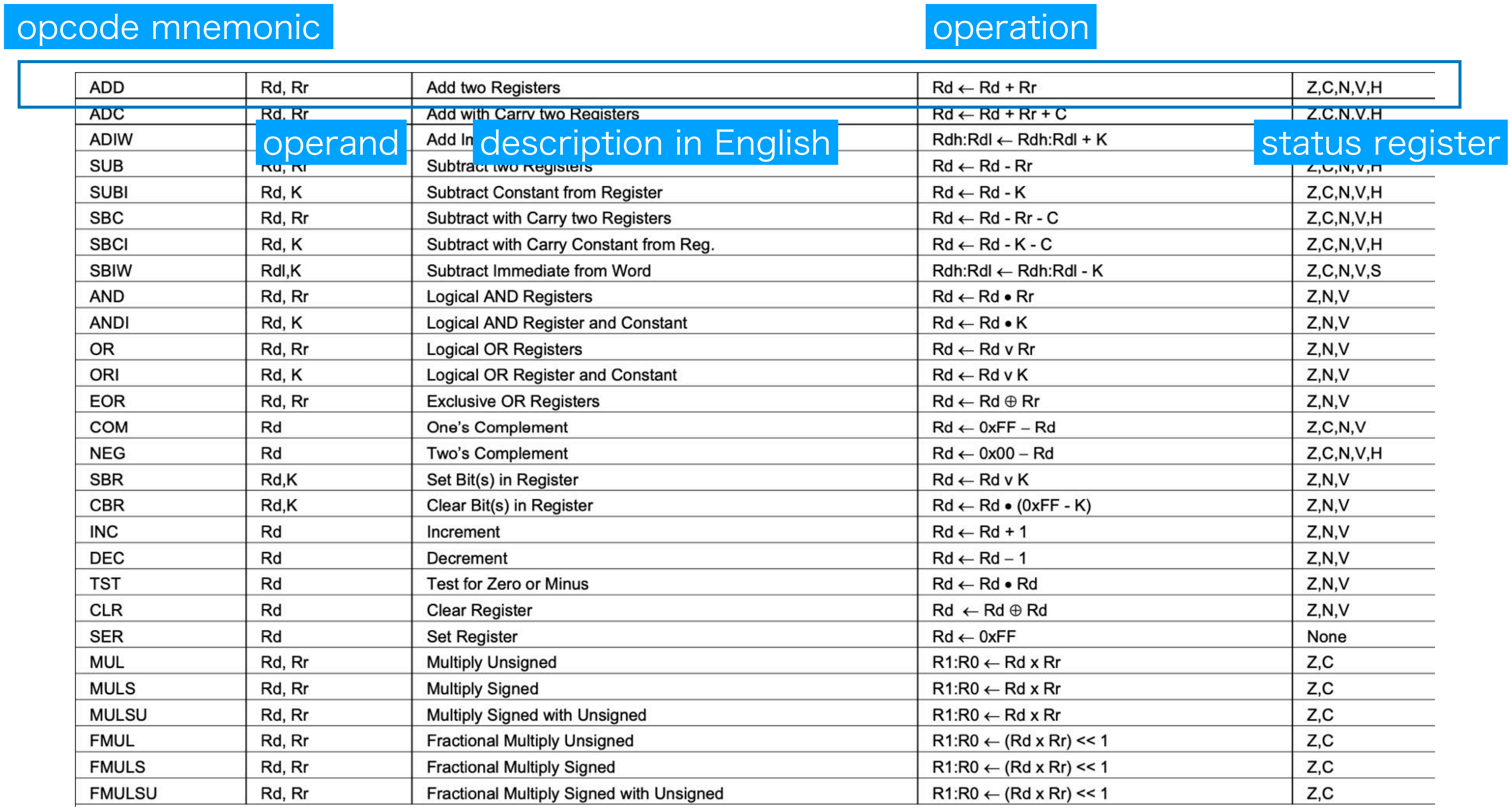
- Basic elements of an instruction:

`ADD r1,r2` means r1 ← r1 + r2

opcode mnemonic

operand

How an instruction looks like


opcode mnemonic
operation
operand
description in English
status register

ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H
ADIW	[illegible]	Add In [illegible]	Rdh:Rdl ← Rdh:Rdl + K	[illegible]
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V
COM	Rd	One's Complement	Rd ← 0xFF – Rd	Z,C,N,V
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd	Z,C,N,V,H
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V
SER	Rd	Set Register	Rd ← 0xFF	None
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C

Sec 37 of the datasheet

"ADD" instruction

- `ADD Rd,Rr` opcode mnemonic and operand
  - Rd ← Rd + Rr operation (this involves addr/data bus manipulations)
  - `000011rd rrrrdddd` opcode of ADD r,d
- `ADD r1,r2` means "r1 ← r1 + r2"
  - `00001100 00100001` = 0x0c 0x21 (2 bytes long)

machine code of the instruction

A Program in "Assembly Language"

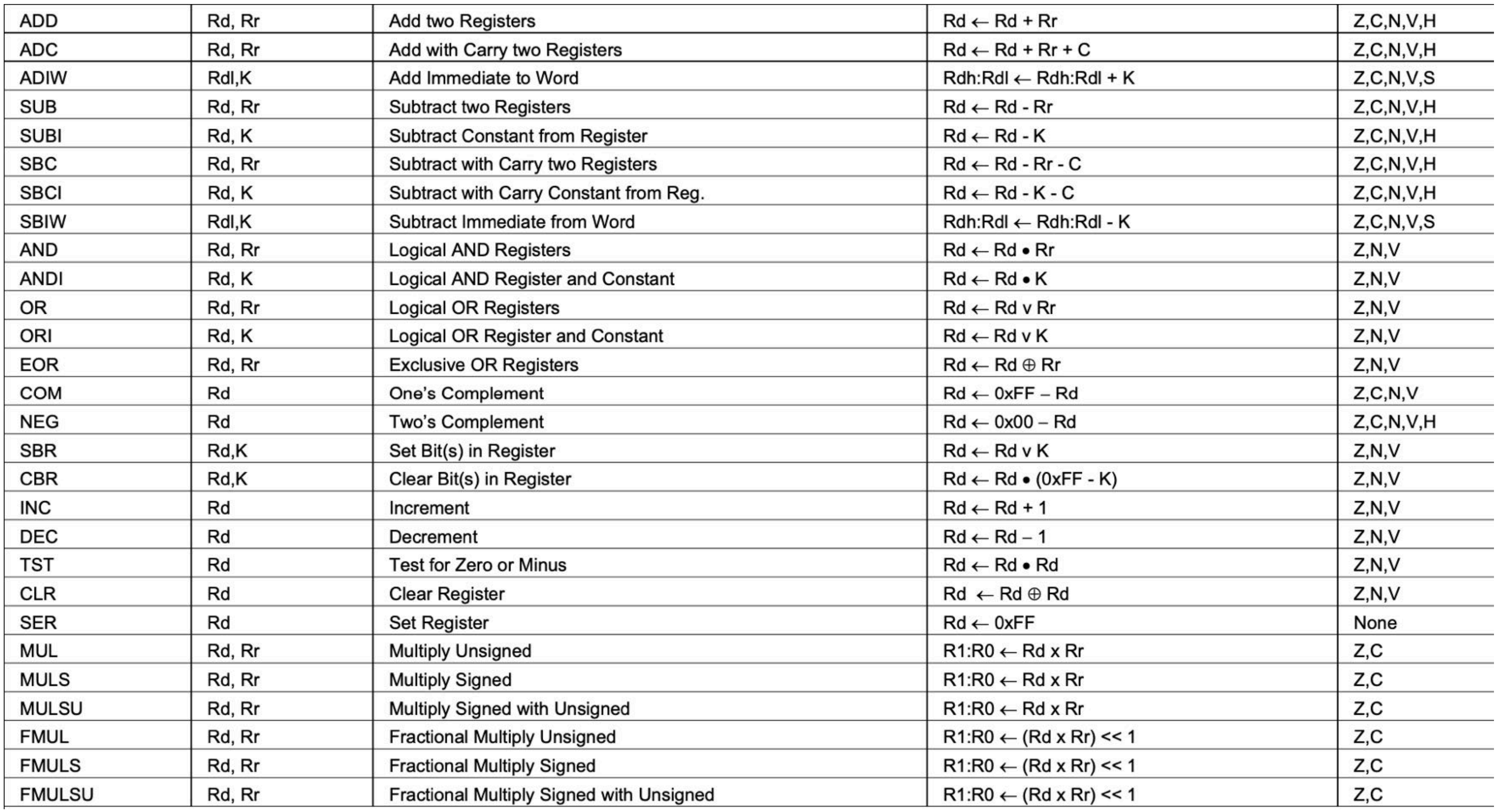
- **A series of machine code in opcode mnemonic and operand.**

variable macro

program address label

```
start:
        out %[port],%[hi]
        ldi r1,100
wait0:
        dec r1
        bnre wait0
        out %[port],%[lo]
        ldi r1,100
wait1:
        dec r1
        bnre wait1
        rjmp start
```

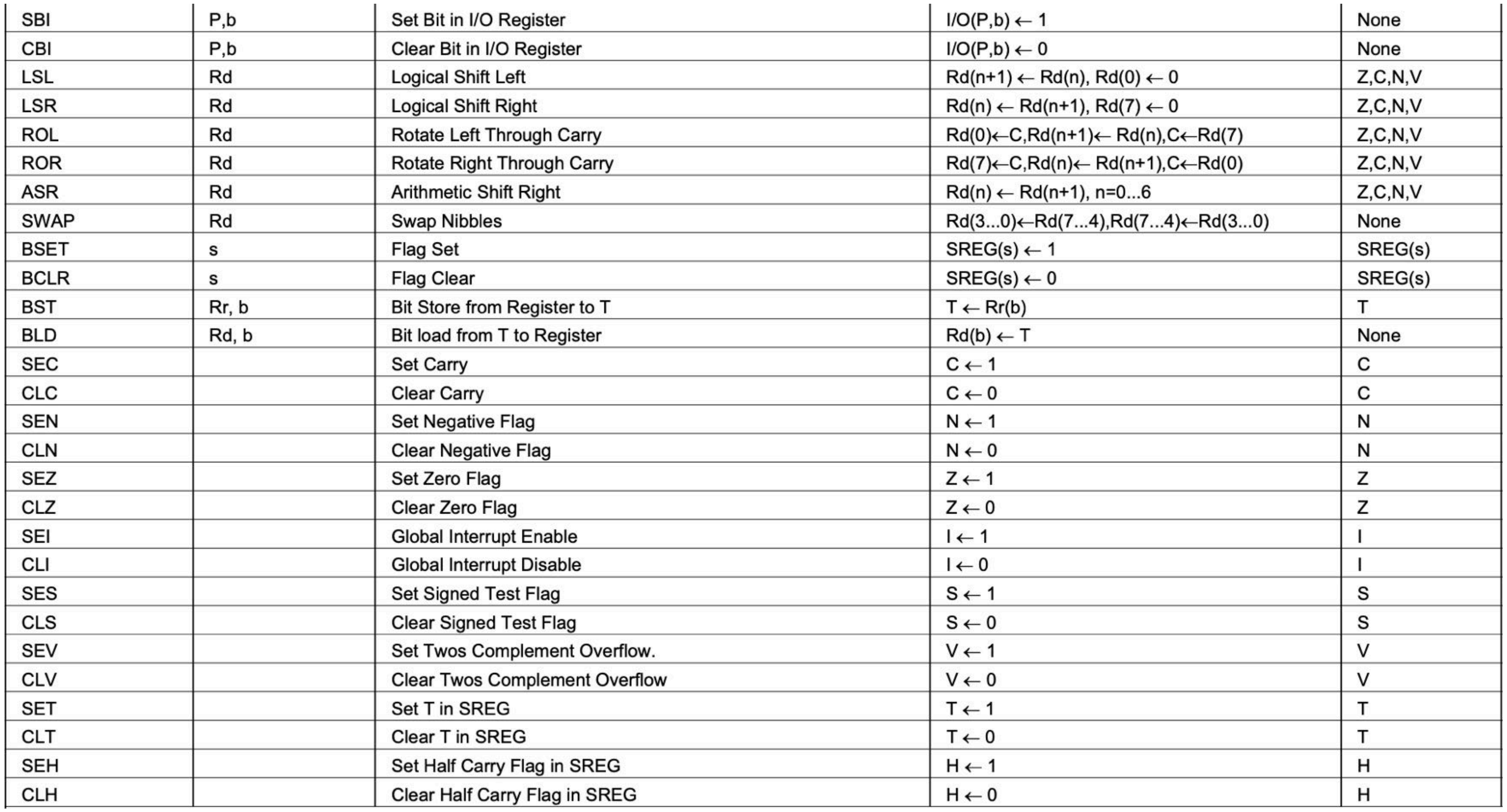
Arithmetic Operations



ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V
COM	Rd	One’s Complement	Rd ← 0xFF – Rd	Z,C,N,V
NEG	Rd	Two’s Complement	Rd ← 0x00 – Rd	Z,C,N,V,H
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V
SER	Rd	Set Register	Rd ← 0xFF	None
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C

Sec 37 of the datasheet

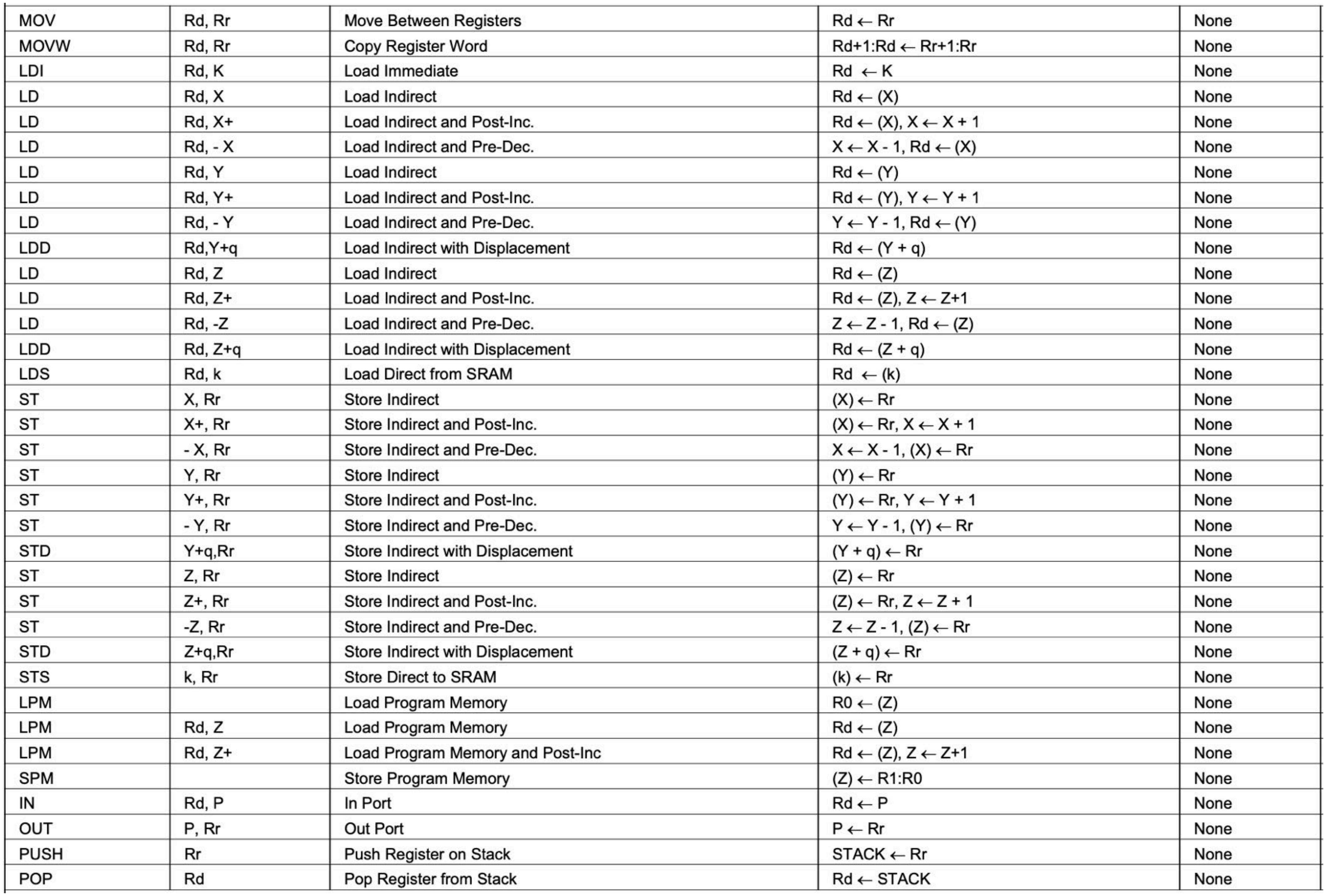
Bit Operations



SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0...6	Z,C,N,V
SWAP	Rd	Swap Nibbles	Rd(3...0)←Rd(7...4),Rd(7...4)←Rd(3...0)	None
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None
SEC		Set Carry	C ← 1	C
CLC		Clear Carry	C ← 0	C
SEN		Set Negative Flag	N ← 1	N
CLN		Clear Negative Flag	N ← 0	N
SEZ		Set Zero Flag	Z ← 1	Z
CLZ		Clear Zero Flag	Z ← 0	Z
SEI		Global Interrupt Enable	I ← 1	I
CLI		Global Interrupt Disable	I ← 0	I
SES		Set Signed Test Flag	S ← 1	S
CLS		Clear Signed Test Flag	S ← 0	S
SEV		Set Twos Complement Overflow.	V ← 1	V
CLV		Clear Twos Complement Overflow	V ← 0	V
SET		Set T in SREG	T ← 1	T
CLT		Clear T in SREG	T ← 0	T
SEH		Set Half Carry Flag in SREG	H ← 1	H
CLH		Clear Half Carry Flag in SREG	H ← 0	H

Sec 37 of the datasheet

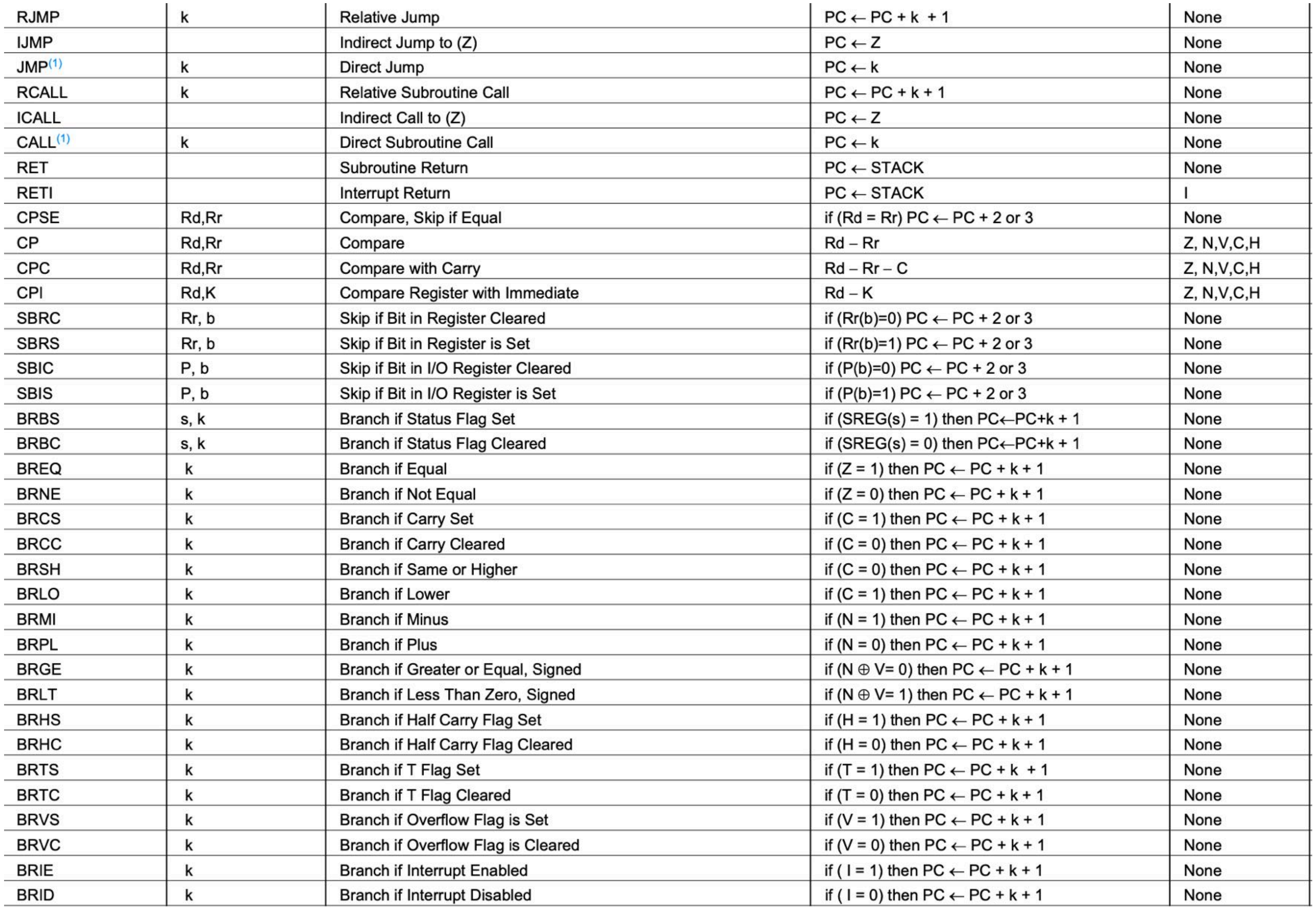
Data Transfer



MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None
LDI	Rd, K	Load Immediate	Rd ← K	None
LD	Rd, X	Load Indirect	Rd ← (X)	None
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None
LD	Rd, Y	Load Indirect	Rd ← (Y)	None
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None
LD	Rd, Z	Load Indirect	Rd ← (Z)	None
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)	None
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None
ST	X, Rr	Store Indirect	(X) ← Rr	None
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None
ST	Y, Rr	Store Indirect	(Y) ← Rr	None
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None
ST	Z, Rr	Store Indirect	(Z) ← Rr	None
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr	None
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None
LPM		Load Program Memory	R0 ← (Z)	None
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None
SPM		Store Program Memory	(Z) ← R1:R0	None
IN	Rd, P	In Port	Rd ← P	None
OUT	P, Rr	Out Port	P ← Rr	None
PUSH	Rr	Push Register on Stack	STACK ← Rr	None
POP	Rd	Pop Register from Stack	Rd ← STACK	None

Sec 37 of the datasheet

Branch



RJMP	k	Relative Jump	PC ← PC + k + 1	None
IJMP		Indirect Jump to (Z)	PC ← Z	None
JMP[(1)]	k	Direct Jump	PC ← k	None
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None
ICALL		Indirect Call to (Z)	PC ← Z	None
CALL[(1)]	k	Direct Subroutine Call	PC ← k	None
RET		Subroutine Return	PC ← STACK	None
RETI		Interrupt Return	PC ← STACK	I
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None
CP	Rd,Rr	Compare	Rd – Rr	Z, N,V,C,H
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None

Sec 37 of the datasheet

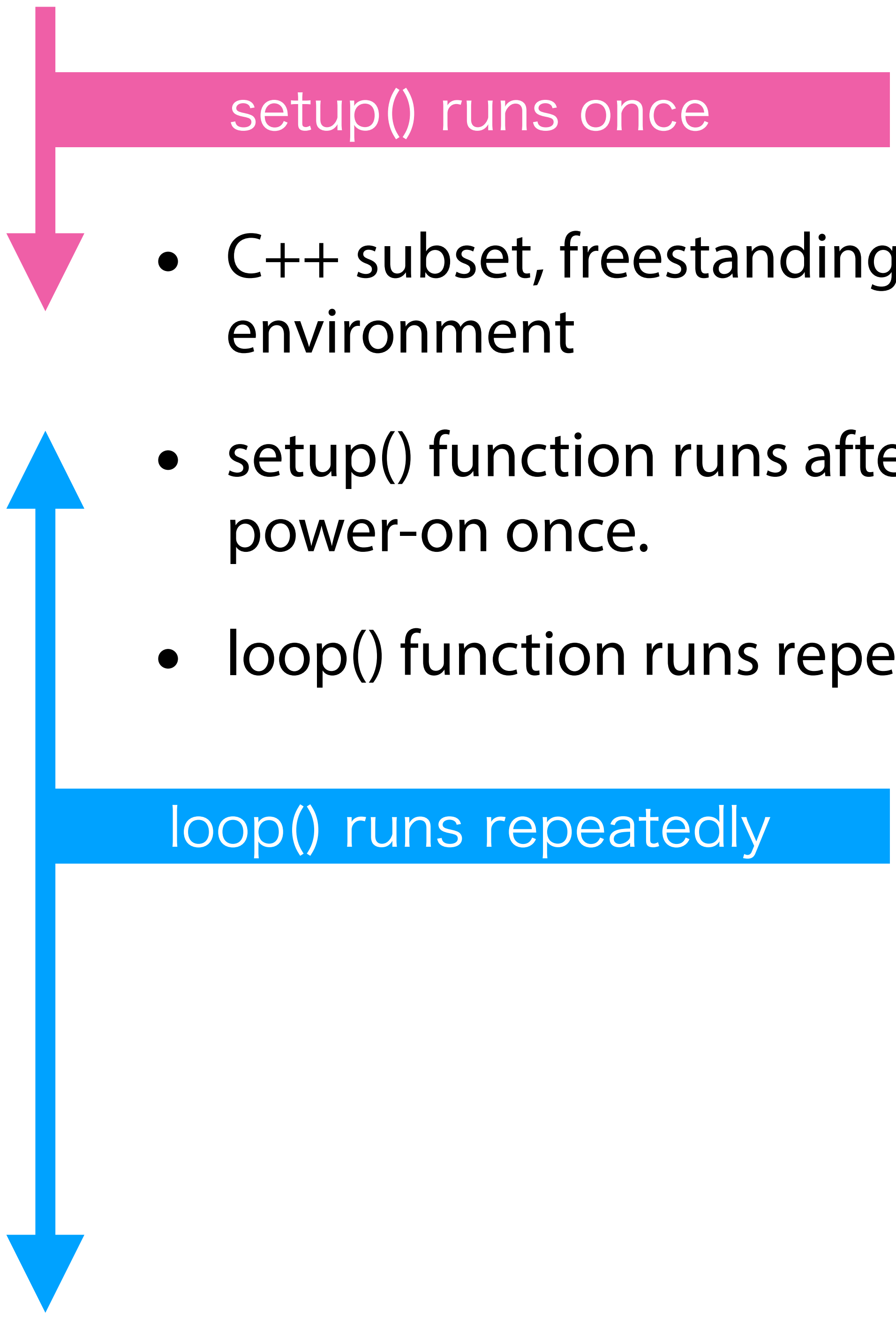
Example of LED Blinker

```
void setup()
{
  pinMode(13, OUTPUT);
}
```

Set 13th I/O as output

```
void loop()
{
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Set HIGH to 13th I/O, wait 1000ms, set LOW, wait 1000ms, ...



- C++ subset, freestanding environment
- setup() function runs after power-on once.
- loop() function runs repeatedly.

Example of LED Blinker

```
loop:
     out %[port],%[hi]
     ldi r1, 100
wait0:
     dec r1
     bnre wait0
     out %[port],%[lo]
     ldi r1, 100
wait1:
     dec r1
     bnre wait1
     rjmp loop
```

`out addr, immediate`

Set `immediate` to I/O address `addr`

`ldi reg, immediate`

Load `immediate` to `reg`

`dec r1`

r1 = r1 - 1 and set status register

`bnre label`

Jump to `label` if zero

`rjmp label`

Jump to `label` unconditionally

Writing a Program

Program in C

```
void loop()
{
     digitalWrite(13, HIGH);
     delay(1000);
     digitalWrite(13, LOW);
     delay(1000);
}
```

Program in assembly language

```
     ldi r16, 250
 wait0:
     nop
     dec r16
     brne wait0
```

- Computer programming language: text representation of the program
- You can write a program in various language (even in Japanese)
- Processors can understand only machine code on memory, however.

Translation to Machine Code

Program in C

```
void loop()
{
    digitalWrite(13, HIGH);
    delay(1000);
    digitalWrite(13, LOW);
    delay(1000);
}
```

Compile

Program in machine code

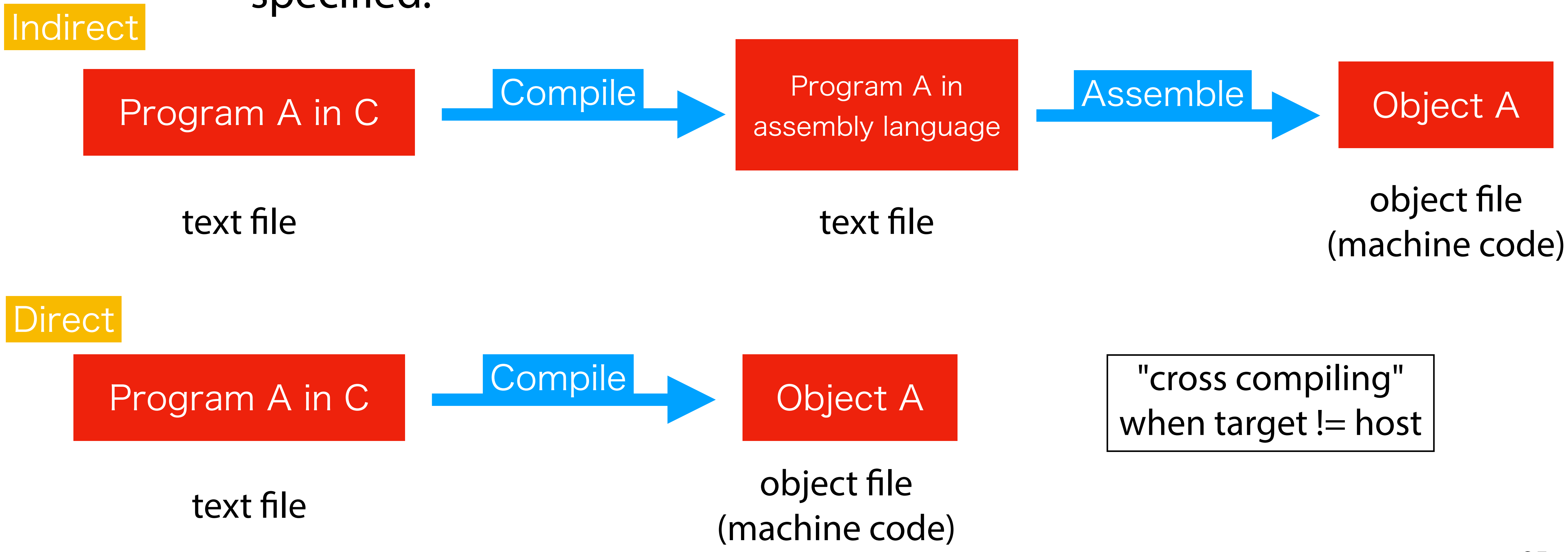
```
07a3d10   1824   8548   74f6   480b   078b   01ba   0000   ff00
07a3d20   6850   3c80   4525   a010   0081   3a75   c931   894c
07a3d30   f0f0   0f49   8fb1   a230   0000   940f   84c1   74c9
07a3d40   4822   d889   415b   415c   415e   5d5f   31c3   49c0
07a3d50   bf8d   a230   0000   8948   e8c6   e362   0021   66e9
07a3d60   ffff   49ff   c689   8149   30c7   00a2   4c00   ff89
```

address　　value

- A program in text must be translated into machine code to run it on a processor.
- Assembly language: "assemble"
- High-level language: "compile"

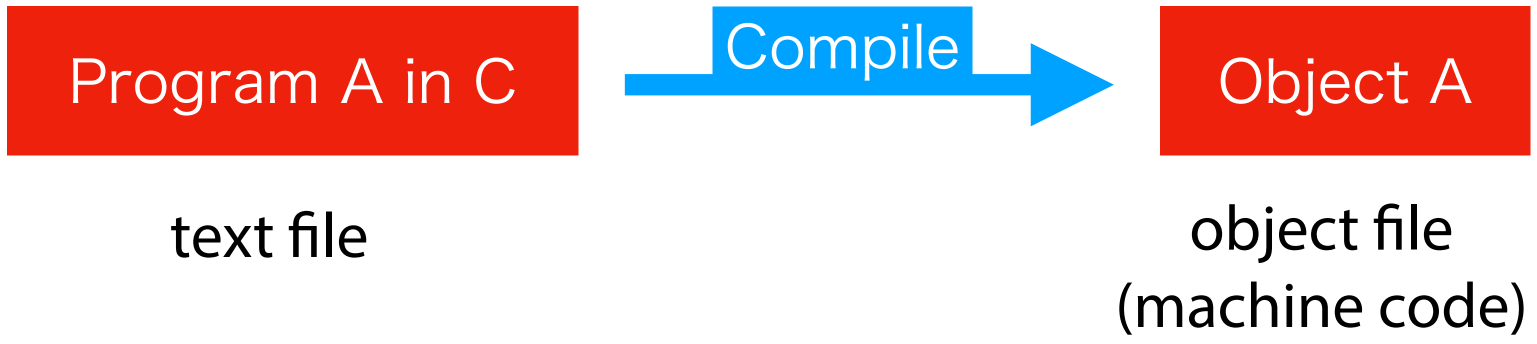
Translation to Machine Code

- **Compile**
  - A machine code is (substantially) generated. It depends on a specific processor while your program does not need to depend on it.
  - The host and target processor architecture must be specified.


Indirect
Program A in C
Compile
Program A in assembly language
Assemble
Object A
text file
text file
object file (machine code)
Direct
Program A in C
Compile
Object A
text file
object file (machine code)
"cross compiling" when target != host

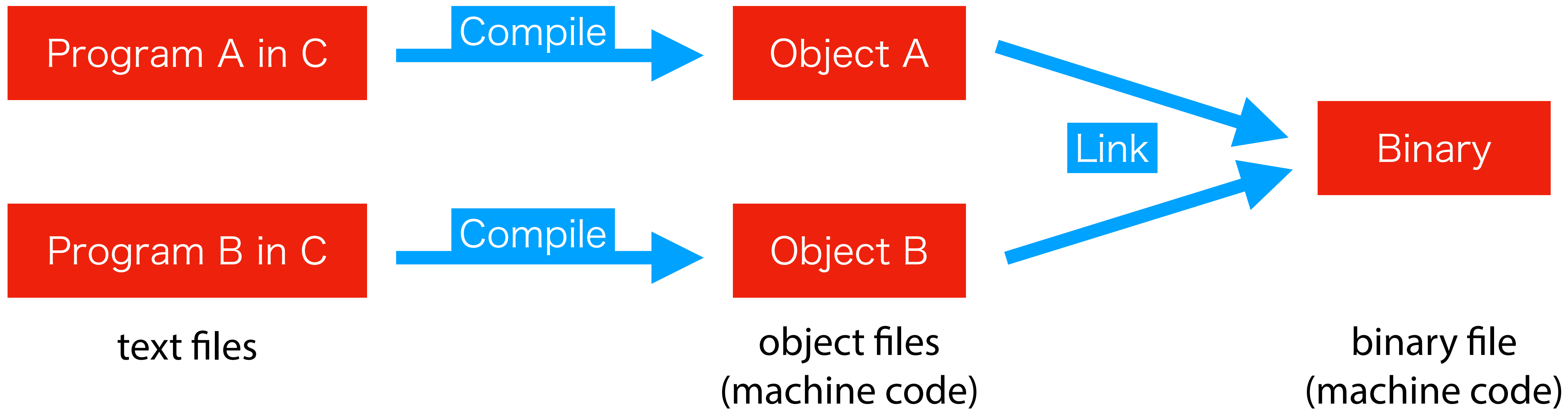
Translation to Machine Code

- **Compile**
  - A machine code is (substantially) generated. It depends on a specific processor while your program does not need to depend on it.
  - The compilation result is not always ready to run on a processor. Why?
    - No information about the entry address
    - You may want to write two or more programs in text to develop a big program.



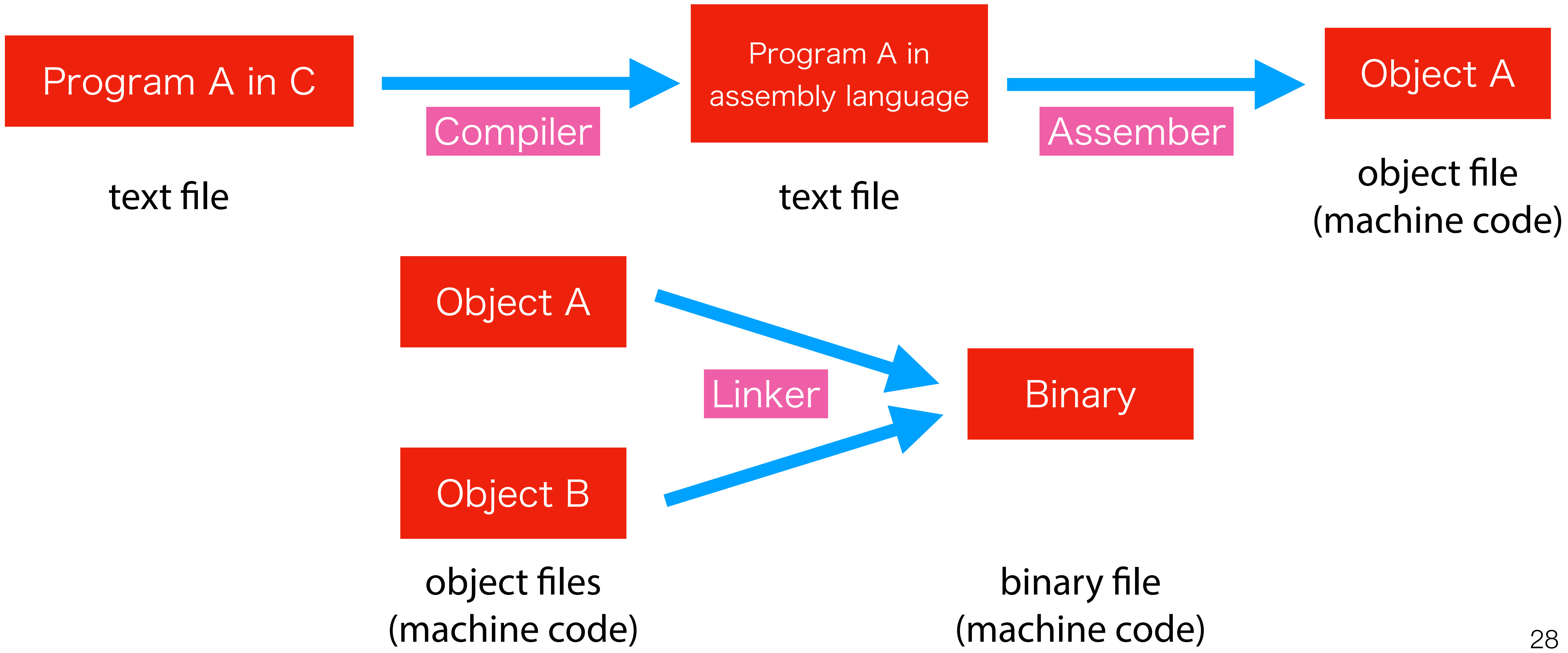
Linking: Generate a "binary"

- "binary" means ready-to-run machine code by a processor
- "Linker" is used to generate a binary
  - Join the multiple compiled results into a single binary
  - Relocate the addresses in the program



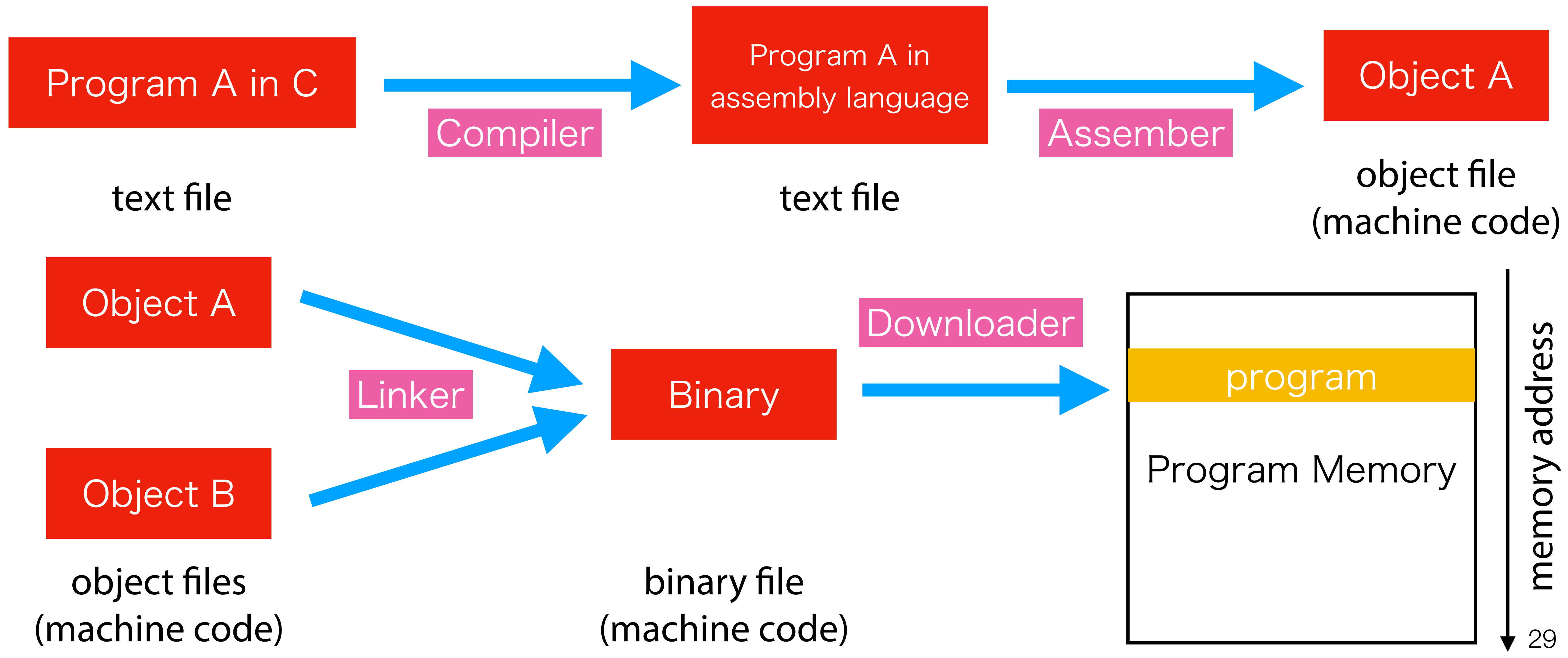
Toolchain

- Toolchain: A set of software tools to generate a binary.
- Compiler, assembler, and linker are typical components.


Program A in C
Compiler
Program A in assembly language
Assember
Object A
text file
text file
object file (machine code)
Object A
Linker
Binary
Object B
object files (machine code)
binary file (machine code)

Downloading to Memory

- Embedded systems do not always have storage (secondary memory) such as HDD.
- Downloading the program into non-volatile memory (flash memory) is popular these days.



Write A Program

- Usually we use a high-level language like C/C++
- If it is not sufficient, we use assembly language (i.e. machine code)
- **Reference for C/C++**: Arduino Sketch Language Reference

  `https://www.arduino.cc/reference/en/`
- **Reference for Instruction Set**:

  `http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf`

Debugging

- It is difficult to monitor the internal states of the processor because embedded systems do not have I/O devices seen on general-purpose computers.
- I/O pins are typically used for debugging. For more systematic monitoring or testing, JTAG interface is popular. Some processors have their own interface dedicated to debugging.
- For Arduino development, we will use built-in serial communication feature on tinkercad simulator.
- We will revisit this after experiencing series of projects.

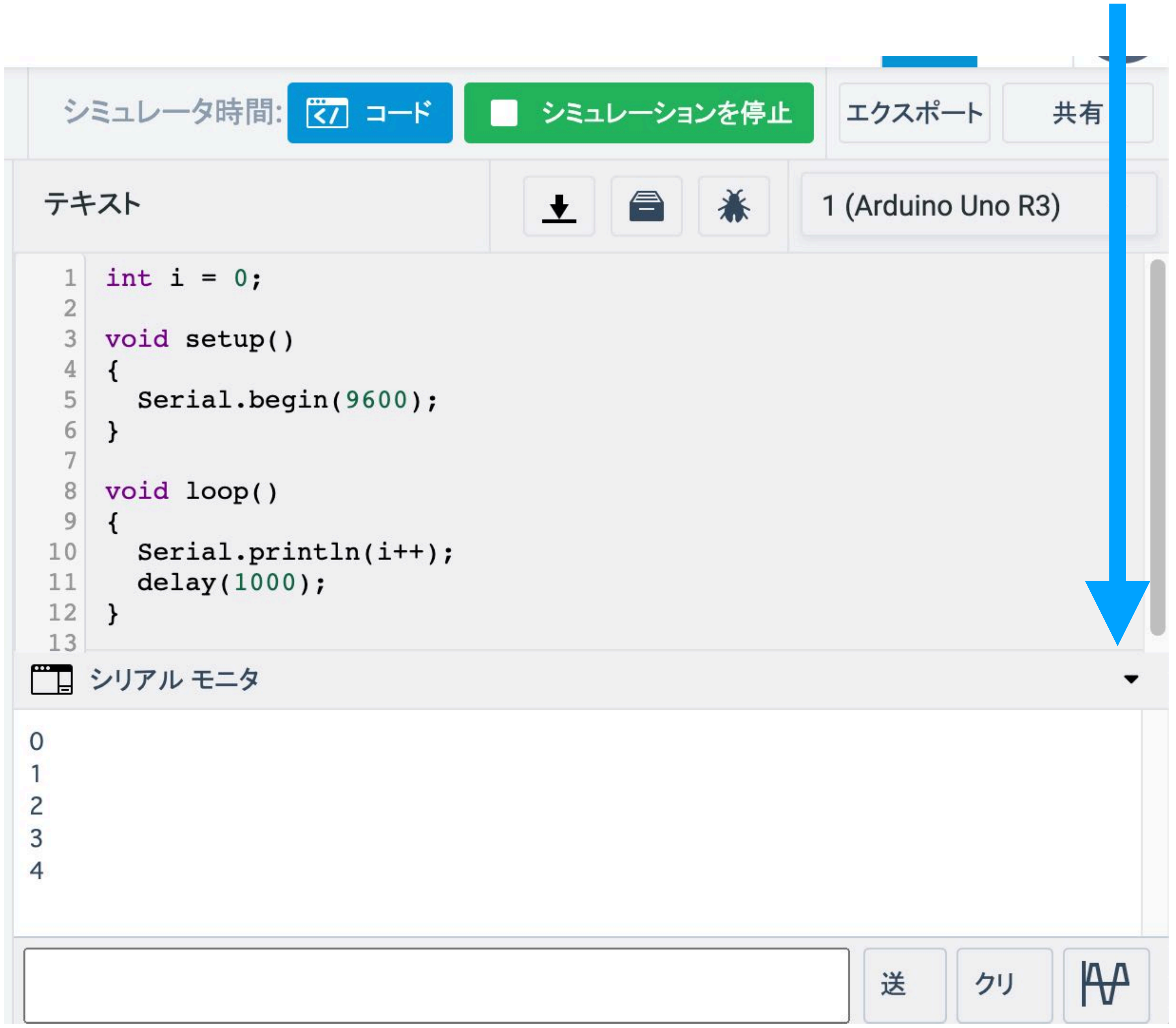
Serial Communication on Arduino Simulator

- Functions for serial communication and simulator are available.
- `Serial.begin(rate)`
  - Initialize the serial port. The "rate" should be 9600.
- `Serial.println(value, format)`
  - Put the value to the serial port. The "format" can be omitted.
- See also: `https://www.arduino.cc/reference/en/language/functions/communication/serial/println/`

Serial Communication on Arduino Simulator

```
int i = 0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.println(i++);
  delay(1000);
}
```

Click this triangle mark



```
int i = 0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.println(i++);
  delay(1000);
}
```

```
0
1
2
3
4
```

Exercise

The 2nd Project

- **6-LED Blinker**
  - Breadboard
  - Serial output for debugging
  - Try various blinking patterns
  - An advanced version: blinking patterns by an array
- **Visit TinkerCad**

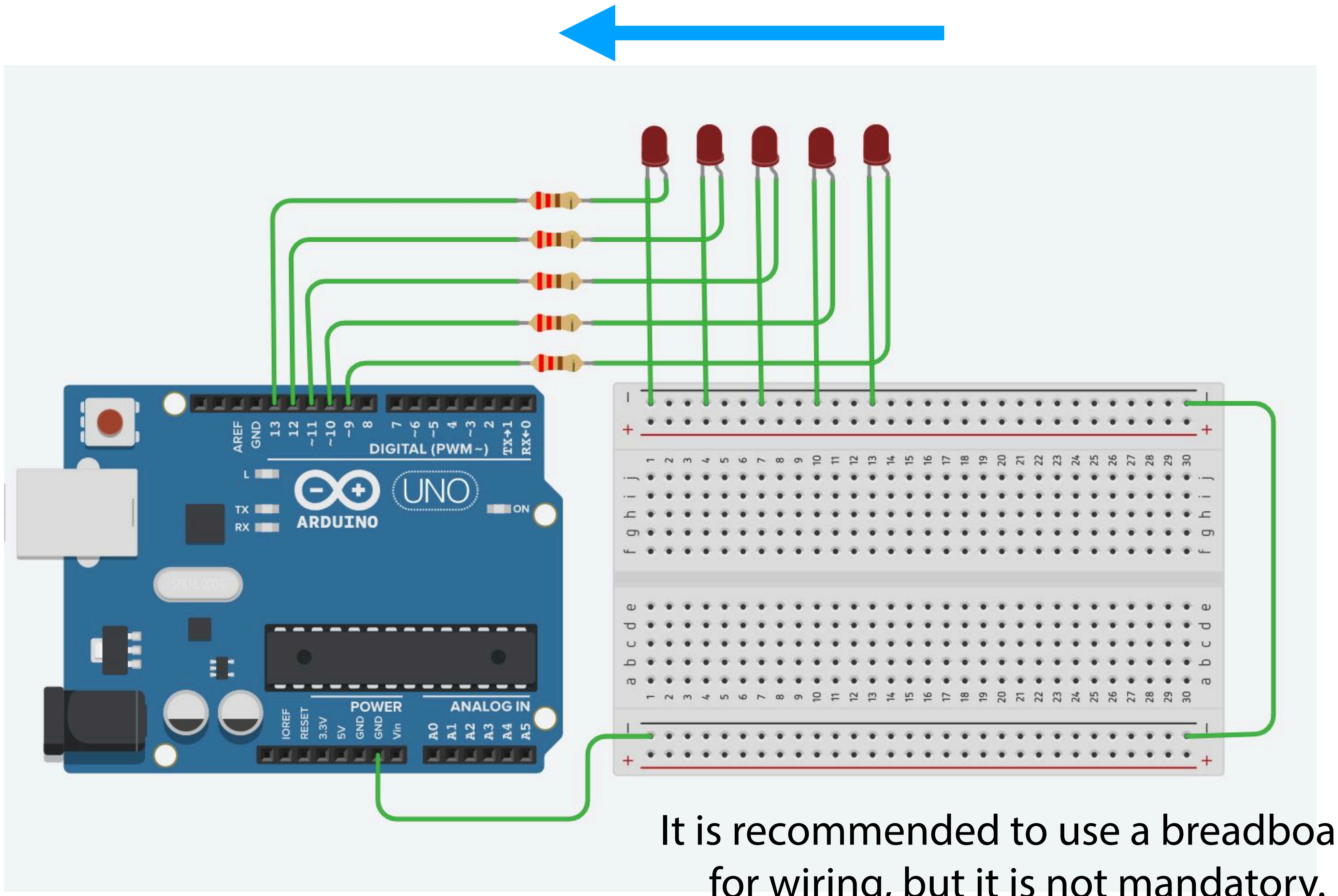
  `https://www.tinkercad.com/joinclass/SKRUGE7BB`
- Enter your nickname. If you do not know it or could not access to the website, contact the teaching assistant

Arduino Sketch Language Reference:

https://www.arduino.cc/reference/en/

5 LEDs Blinker (as example)

blinking in this direction with 1s interval



breadboard

horizontally connected

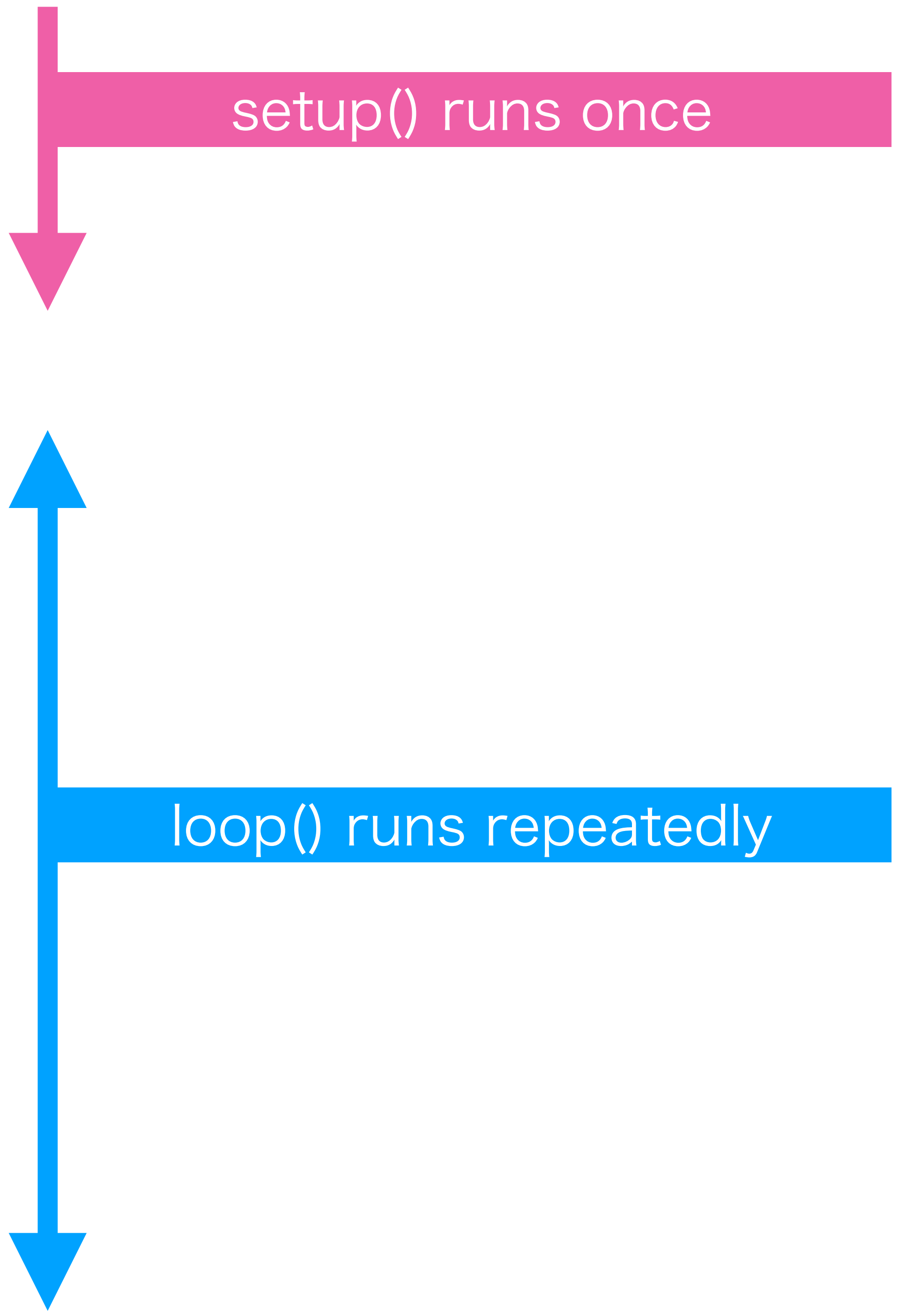
vertically connected

vertically connected

It is recommended to use a breadboard for wiring, but it is not mandatory.

1-LED Blinker w/ Serial Output

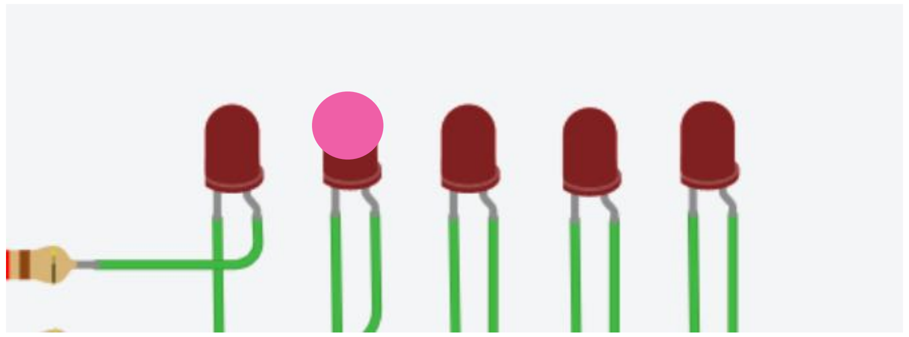
```
void setup()
{
  pinMode(13, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(13, HIGH);
  Serial.println(1);
  delay(1000);
  digitalWrite(13, LOW);
  Serial.println(0);
  delay(1000);
}
```



Advanced 6-LED Blinker

```
const char *pattern[] = {
  "010101",
  "101010",
  "111100",
  "000011"
};
void setled(const char *pat)
{
  ...
}
void setup()
{
  ...
}
void loop()
{
  int i;
  for (i = 0; i < sizeof(pattern)/sizeof(pattern[0]); i++) {
    setled(pattern[i]);
    delay(1000);
  }
}
```

"010000"



(note: try 6-LED, not 5)

- The patterns are defined as an array "pattern".
- Define setled() to read a pattern and set the LED status.
- The defined patterns must repeatedly appear on the LEDs in 1-second interval.

Conclusions

- How a processor works
- How a program looks like
- What software tools are used to generate a "binary" representation of your program

- **Next week**:
  Development Environment and Toolchains
- **Homework**:
  Complete your 6-LED blinker with serial debug output