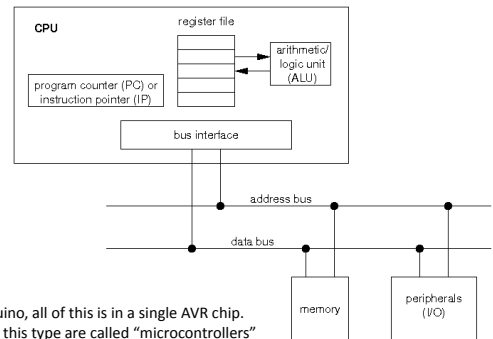


# Hardware and Instruction Set Architecture

CSE 102

## Simple Computer System

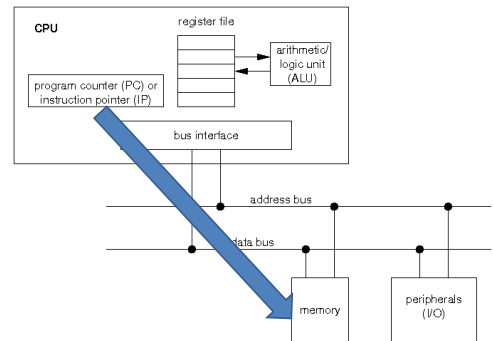


For Arduino, all of this is in a single AVR chip. Chips of this type are called "microcontrollers"

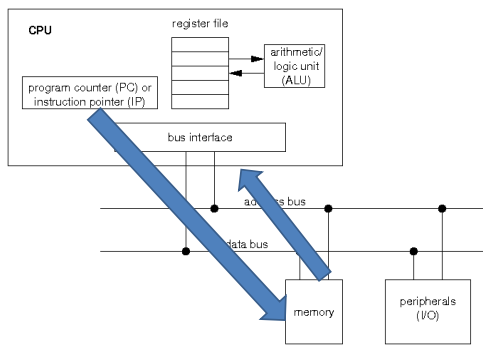
## Fetch-Decode-Execute Cycle

- Fetch: grab (fetch) the instruction to be executed. It's address is in the instruction pointer (IP) or program counter (PC)
- Decode: figure out what instruction it is and what is to be done (e.g., this is an ADD inst. that needs two values from the register file)
- Execute: do the real work and store the result somewhere (as told by the instruction)

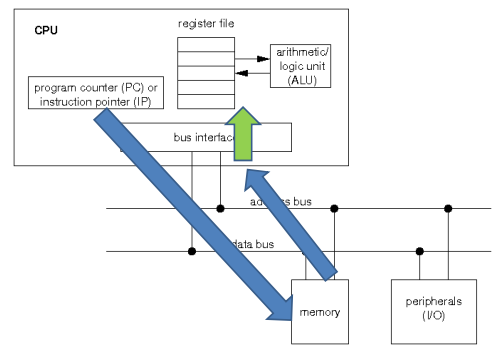
## Fetch

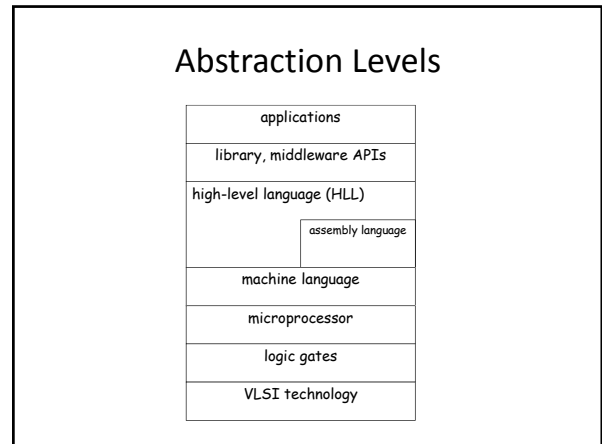
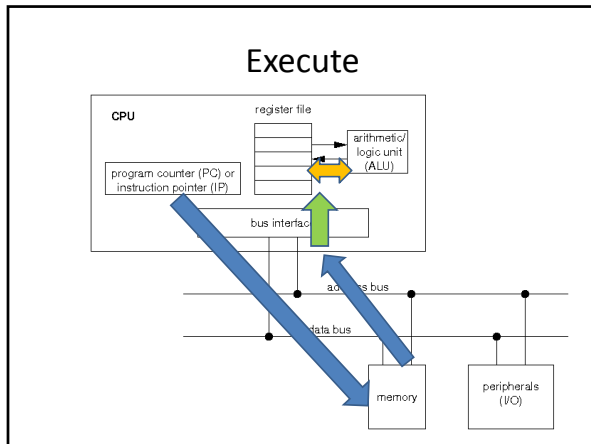


## Fetch



## Decode





### HLLs, assembly, vs. machine language

- machine language = binary (i.e., computer readable) image of program code
- assembly language = human readable (and writeable) syntax directly representing machine language

```

    graph LR
      asm((asm)) <--> ml((ml))
    
```

- one-to-one mapping between asm and machine language

### HLLs, assembly, vs. machine language

- high-level language = designed for human specification of algorithms and applications

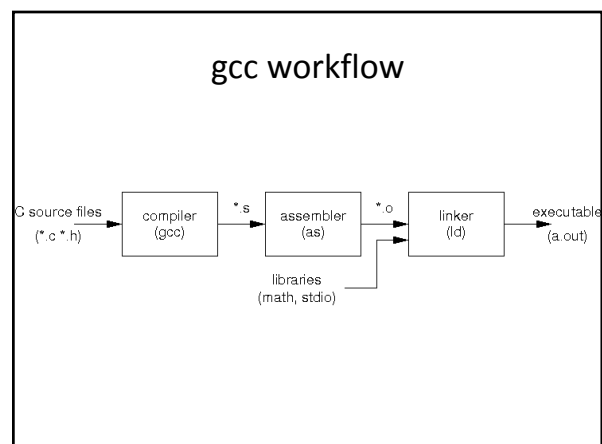
```

    graph LR
      HLL((HLL)) --> ML((ml))
      HLL --> ML2((ml))
      HLL --> ML3((ml))
      HLL --> ML4((ml))
    
```

- one-to-many mapping between HLL and machine language
- Assembly/machine language is very much architecture dependent
- HLLs are (largely) architecture independent

### Why use assembly language?

- Direct control over the hardware
  - “I want the machine to execute these exact instructions.”
- Historical reasons
  - execution efficiency (speed, code size, etc.)
  - lack of suitable HLL compiler (embedded)
- Today
  - Limited need as an authoring language
  - Very useful for investigation – understanding!



## Instruction Set Architecture (ISA)

- Programmer's view of the processor. It includes the following components:
  - Instruction set: the collection of instructions that are supported by the processor.
  - Register file: the programmer-visible storage within the processor.
  - Memory: the logical organization of the memory (again, programmer's view)
  - Operating modes: some processors have subsets of the instructions that are privileged based on being in a given "mode." (The Arduino AVR processor doesn't have this, but the x86 processor inside a PC does.)

## AVR Instruction Set

- Arithmetic operations: (add, sub, mul, etc.)
- Boolean operations: (and, or, etc.)
- Shift operations: (left shift, right shift)
- Comparison operations: (<, ≤, >, ≥, =, ≠)
- Memory operations: (load, store)
  - Data movement operations are the only ops that reference memory, all others are to/from registers

## AVR Instruction Set

- Control flow operations:
  - Unconditional branch: (jmp)
  - Conditional branch: (breq, brne, etc.)
  - Procedure call/return: (call, ret)
- Peripheral access: (in, out)
- System operations: (nop, sleep, etc.)

## AVR Register File

- 32 general-purpose registers in the AVR ISA:
  - Each 8 bits wide, named R0 to R31
  - Sometimes paired for 16-bit data – e.g., (R5:R4) has least significant bits in R4 and msbits in R5
  - Last 3 register pairs used for addressing – they are named X (R27:R26), Y (R29:R28), and Z (R31:R30)
- 3 special-purpose registers
  - PC – program counter (16 bits wide)
  - SREG – Status register (8 bits wide)
  - SP – stack pointer (16 bits wide), for system stack

## Status Register

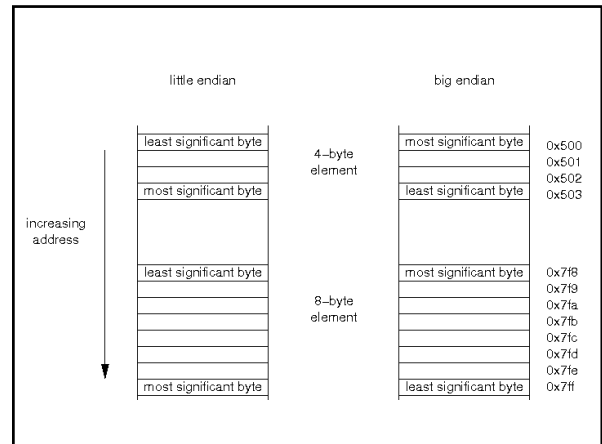
- SREG is status register ⇒ status bits retaining results of previous operations:
  - C – carry – result of unsigned add is too large
  - Z – zero – result of previous operation is 0
  - N – negative – result of operation is negative
  - V – overflow – result of signed op is out of range
  - S – sign – true sign = N xor V
  - H – half carry – used for BCD arithmetic
  - T – bit copy – used by bit load and store inst.
  - I – interrupt – interrupts are enabled

## Memory Organization

- On an AVR processor, there are multiple memories
  - Program memory
  - Data memory
- Data memory is byte addressable (i.e., each byte in the memory has a unique address)
- For multi-byte data elements, the address of the element is the lowest address the element occupies (e.g., for a 16-bit integer occupying 0x500 and 0x501, the address of the integer is 0x500)
- Program memory is 16-bit word addressable (the size of an AVR instruction)

### Little Endian vs. Big Endian

- The “endian”-ness of a processor defines the ordering it uses for multi-byte primitive data elements (e.g., 16-bit int on AVR).
- Little endian  $\Rightarrow$  the least significant byte (LSB) goes in the lowest address, “littlest end first”
- Big endian  $\Rightarrow$  the most significant byte (MSB) goes in the lowest address, “biggest end first”
- AVR is a little endian machine



### Primitive Elements vs. Collections

- Note: endianness only applies to primitive elements (e.g., integers, floats, etc.), not collections of elements
- A “string” is an array of characters, and therefore is not impacted by the endianness of the machine. The first character is in the lowest address.

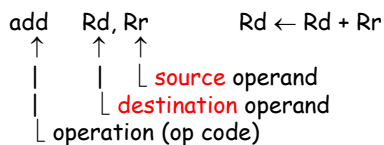
### Little Endian 32-bit Long Integer

long val = 0x12345678; /\* &val is 0x5000 \*/

addr	byte
0x5000	0x78
0x5001	0x56
0x5002	0x34
0x5003	0x12

### Addressing Modes

- Register addressing – operand is in register



$$0 \leq d \leq 31, 0 \leq r \leq 31$$

- Immediate addressing – operand is explicitly present in code



- Constant values use C notation:
  - Default base is 10
  - Hex uses 0x notation (10<sub>16</sub> is written 0x10)
  - Negative constants are allowed, e.g., -12

- Direct addressing – memory address of operand is explicit in code

`lds Rd, (k)`                       $Rd \leftarrow M[k]$

$0 \leq d \leq 31, 0 \leq k \leq 65,535$

- Note: typically use symbols (variable names) instead of explicit  $k$

– Assembler translates into actual address

– Or linker if specified in another file

`sts (var), Rr`                       $M[var] \leftarrow Rr$

- Indirect addressing – memory address is stored in register

`st X, Rr`                               $M[X] \leftarrow Rr$

- Address register can be X, Y, or Z
  - X is (R27:R26), Y is (R29:R28), and Z is (R31:R30)

- There are also post-increment and pre-decrement versions

`st X+, Rr`                               $M[X] \leftarrow Rr, X \leftarrow X+1$   
`st -X, Rr`                               $X \leftarrow X-1, M[X] \leftarrow Rr$

## General Form

label: opcode operands      comment

- Label is optional
- Comments use different notations
  - Many assemblers (incl. AVR)      `; comment`
  - Or some other notation, e.g.,      `# comment`
- Pseudo-operations are commands to assembler
  - `.text`      means “text section”, or instructions are next

## Inline Assembly

```
asm volatile(
"opcode operands \n\t"
"opcode operands \n\t"
::: opt. list of altered regs
);

```

C code

```
a = 10;
x = a;

```

Declarations

```
byte a;
int x;

```

Assembly code

```
asm volatile(
"ldi r24,10 \n\t"
"sts (a),r24 \n\t"
::: "r24"
);
asm volatile(
"lds r24,(a) \n\t"
"ldi r25,0 \n\t"
"sts (x),r24 \n\t"
"sts (x+1),r25 \n\t"
::: "r24","r25"
);

```