# Architecture Models

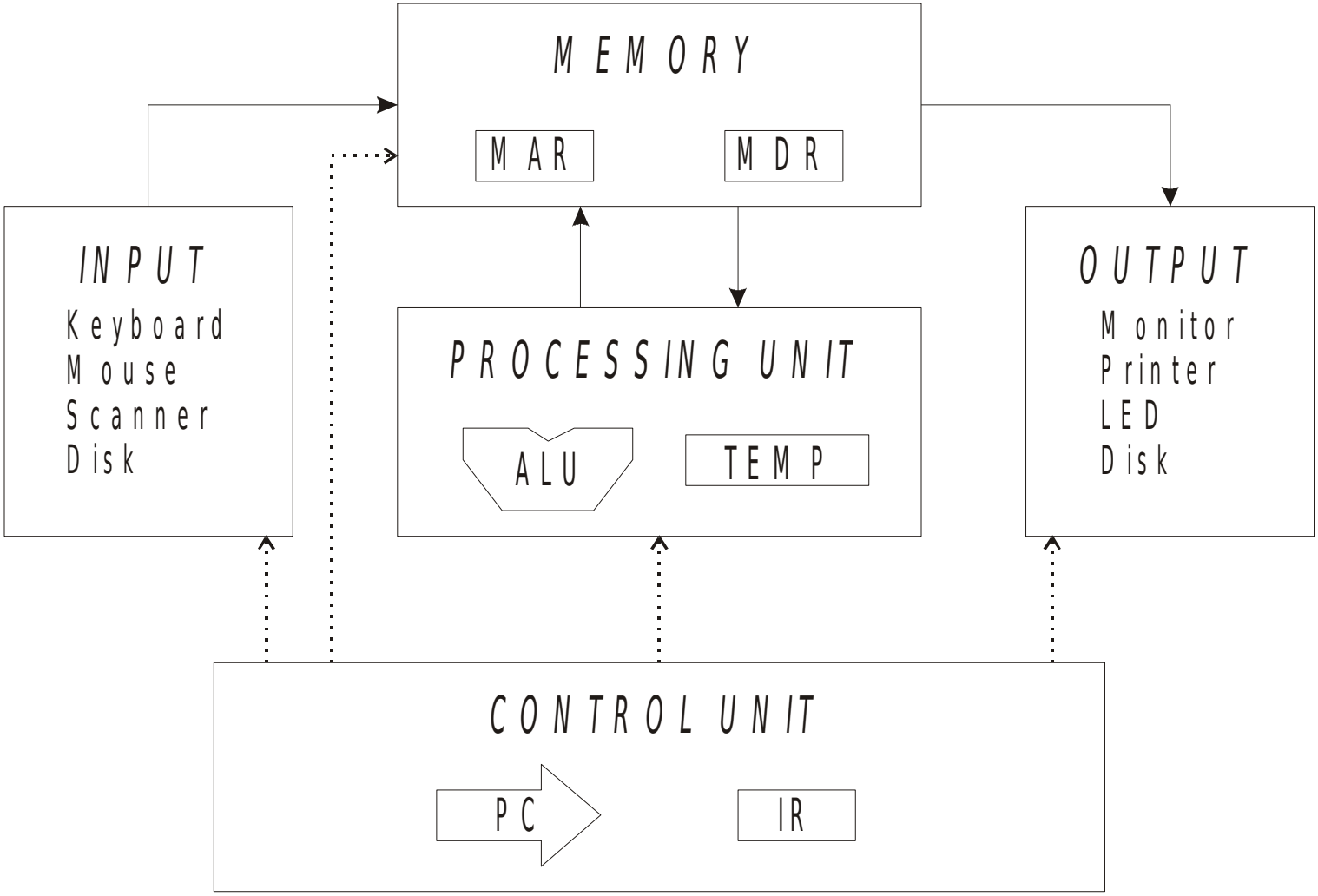
# The Stored Program Computer

- 1943: ENIAC
  - Presper Eckert and John Mauchly -- first general electronic computer. (or was it John V. Atanasoff in 1939?)
  - Hard-wired program -- settings of dials and switches.
- 1944: Beginnings of EDVAC
  - among other improvements, includes program stored in memory
- 1945: John von Neumann
  - wrote a report on the stored program concept, known as the *First Draft of a Report on EDVAC*
- The basic structure proposed in the draft became known as the “von Neumann machine” (or model).
  - a *memory*, containing instructions and data
  - a *processing unit*, for performing arithmetic and logical operations
  - a *control unit*, for interpreting instructions

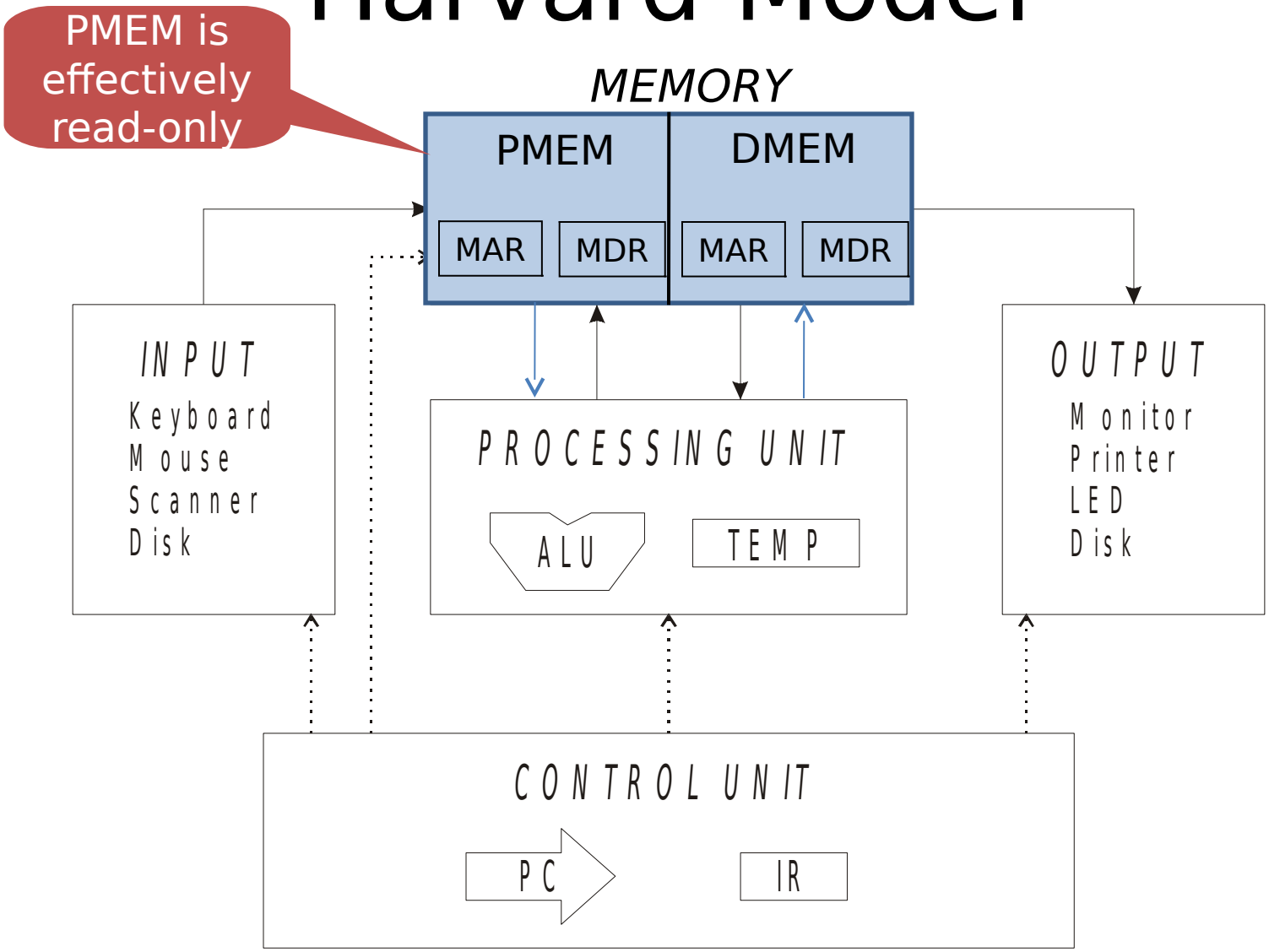
# Harvard Model

- Devised circa 1947
- Refinement of von Neumann Model
  - contains separate memories for data and program storage
  - Program and data stored in same memory in von Neumann Model
  - Advantages? Disadvantages?
- Conceptually the models are very similar
  - synchronous, sequential
  - programs interpreted by the Control Unit

# Von Neumann Model



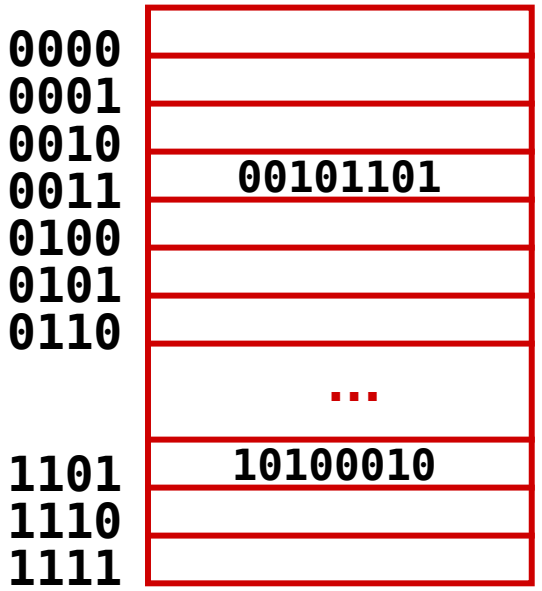
# Harvard Model



# Remembering Memory

- $2^k$ x $m$ array of stored bits
- Address
  - unique ($k$-bit) identifier of location
- Contents
  - $m$-bit value stored in location

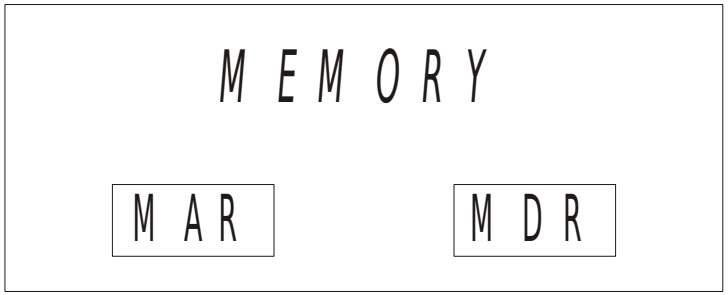
- Basic Operations:
- LOAD
  - read a value from a memory location
- STORE
  - write a value to a memory location



# Interface to Memory

How does processing unit get data to/from memory?

- MAR: Memory Address Register
- MDR: Memory Data Register

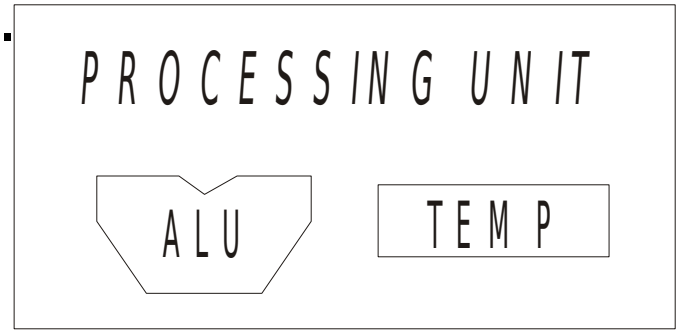



- To LOAD a location (A):
  1. Write the address (A) into the MAR.
  2. Send a “read” signal to the memory.
  3. Read the data from MDR.

- To STORE a value (X) to a location (A):
  1. Write the data (X) to the MDR.
  2. Write the address (A) into the MAR.
  3. Send a “write” signal to the memory.

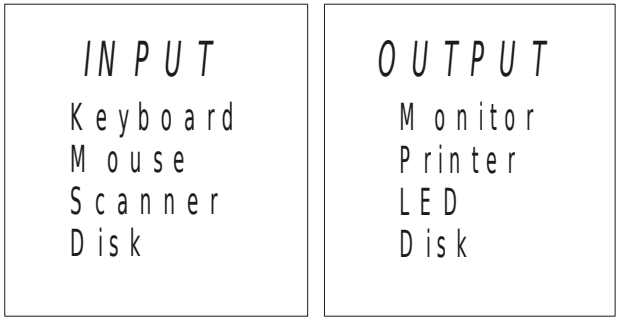
# Processing Unit

- Functional Units
  - ALU = Arithmetic and Logic Unit
  - could have many functional units. some of them special-purpose (multiply, square root, …)
- Registers
  - Small, temporary storage
  - Operands and results of functional units
  - atmega328P has 32 registers (R0, …, R31), each 8 bits wide
- Data Word Size
  - number of bits normally processed by ALU in one instruction
  - also width of registers
  - atmega328P is 8 bits



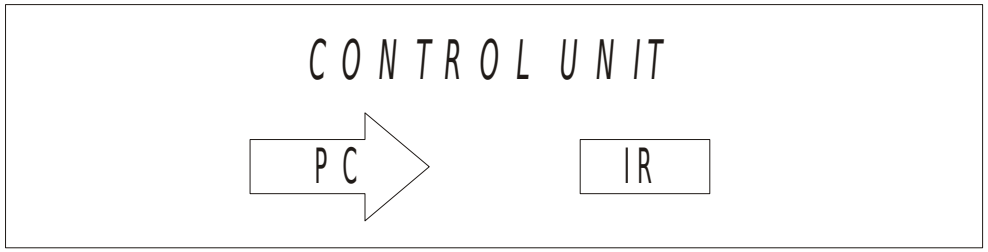
# Input and Output

- Devices for getting data into and out of computer memory
- Each device has its own interface, usually a set of registers like the memory's MAR and MDR
- Some devices provide both input and output
  - disk, network
- Program that controls access to a device is usually called a *driver*.



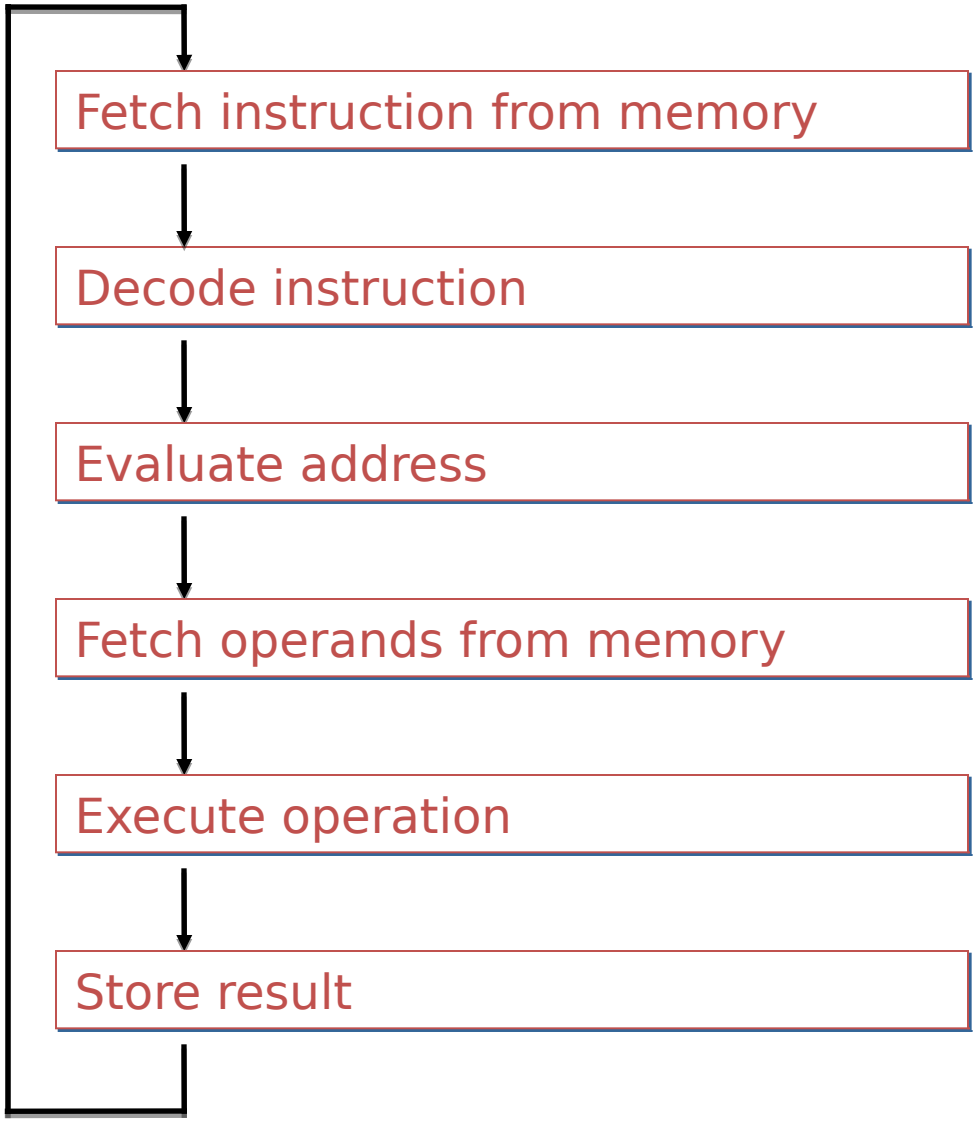
# Control Unit

- Orchestrates execution of the program




- Instruction Register (IR) contains the *current instruction*.
- Program Counter (PC) contains the *address* of the next instruction to be executed.
- Control unit:
  - reads an instruction from memory
    - the instruction's address is in the PC
  - interprets the instruction, generating signals that tell the other components what to do
    - an instruction may take many *machine cycles* to complete

# Instruction Processing



# About Instructions

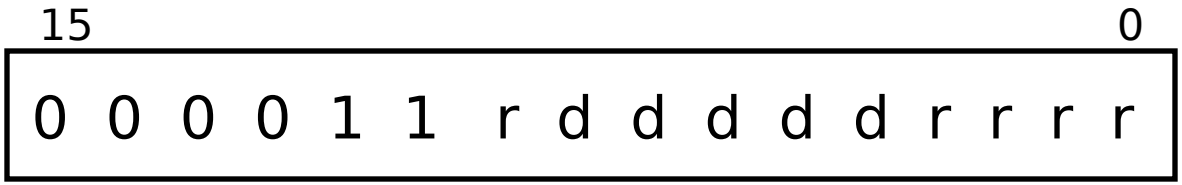
- The instruction is the fundamental unit of work.
- An Instruction specifies two things:
  - *opcode*: operation to be performed
  - *operands*: data/locations to be used for operation

- An instruction is encoded as a sequence of bits.
  *(Just like data!) (Oh Noes!?)*
  - Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
  - Control unit interprets an instruction: generates sequence of control signals to carry out operation.
  - Operation is either executed completely, or not at all.

- A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*.

# AVR add Instruction

- Instruction Format



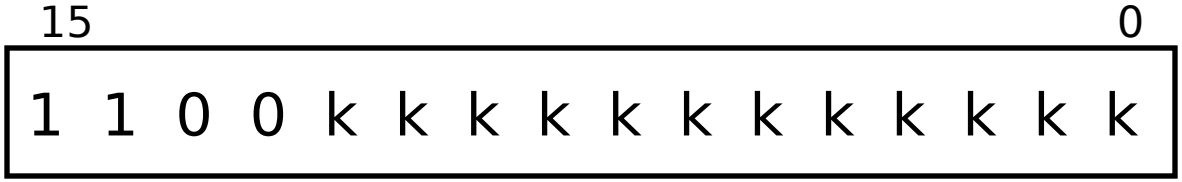

- Syntax: add Rd, Rr  0 <= d <= 31, 0 <= r <= 31

- Operation: Rd := Rd + Rr
- Examples

```
add r1, r3   ; add r3 to r1...     bits are 0000110000010011
add r8, r8   ; add r8 to itself... bits are 0000110010001000
```

# AVR rjmp Instruction

- Instruction Format

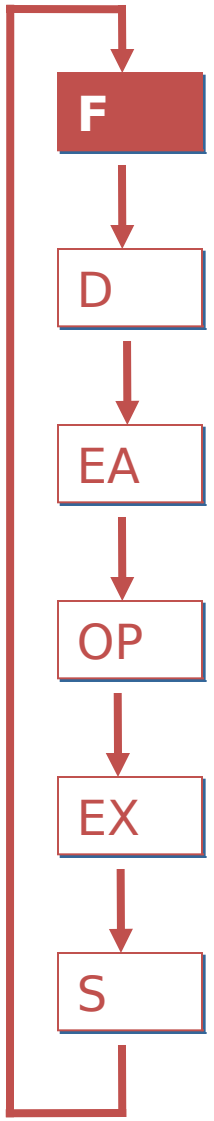



- Syntax: rjmp k  -2K <= k < 2K
- Operation: PC := PC + k (PC is already incremented)
- Example

```
    rjmp DoIt   ; jmp to label DoIt...  bits are 1100000011111111
    ...
.org 0x100
DoIt:
        add r8, r8   ; add r8 to itself... bits are 0000110010001000
```

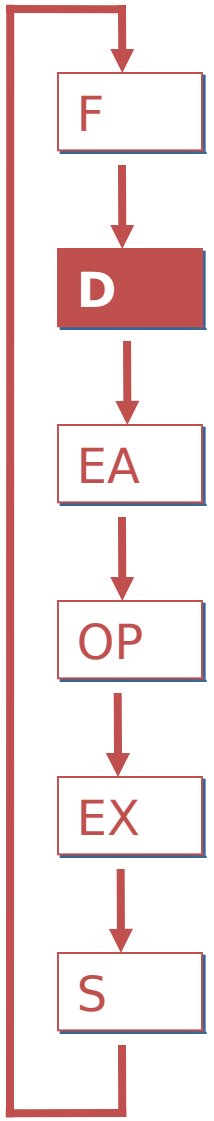
# Instruction Processing: FETCH

- Load next instruction (at address stored in PC) from memory into Instruction Register (IR).
  - Copy contents of PC into PMAR.
  - Send “read” signal to memory.
  - Copy contents of PMDR into IR.
- Then increment PC, so that it points to the next instruction in sequence.
  - PC becomes PC+1.



# Instruction Processing: Decode

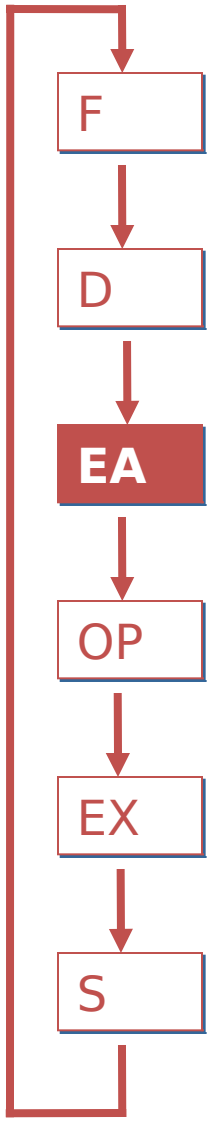
- First identify the opcode.
  - For ADD insn, this is bits [15:10]
- Depending on opcode, identify other operands
  from the remaining bits.
  - Example:
    - for ADD insn, need r and d operands



# Instruction Processing: Evaluate Address

- For instructions that require memory access,
  compute address used for access.

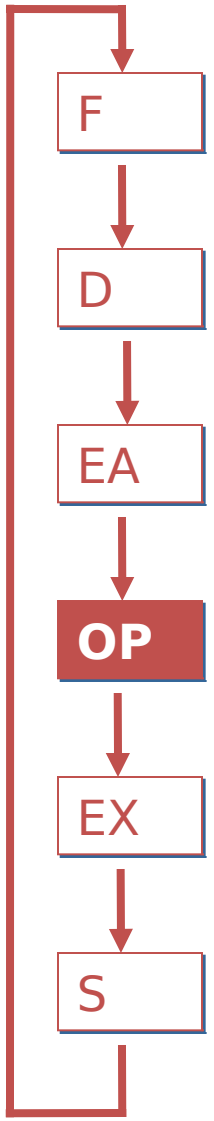
- Example:
  - Get address of jump destination (AVR RJMP insn)



# Instruction Processing: Fetch Operands

- Obtain source operands needed to perform operation.

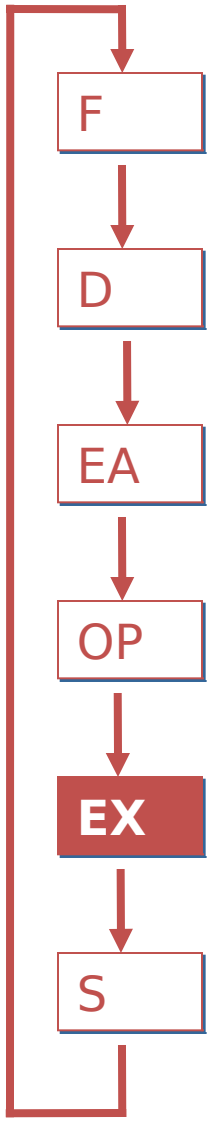
- Examples:
  - load data from memory (LD)
  - read data from register file (ADD)



# Instruction Processing: Execute

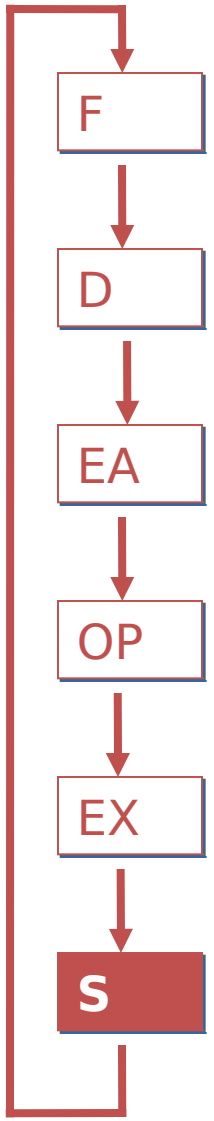
- Perform the operation, using the source operands.

- Examples:
  - send operands to ALU and assert ADD signal
  - do nothing (e.g., for loads and stores)



# Instruction Processing: Store Result

- Write results to destination (register or memory)
  - Also known as “writeback”
- Examples:
  - result of ADD is placed in destination register
  - result of memory load is placed in destination register
  - for store instruction, data is stored to memory
    - write address to MAR, data to MDR
    - assert WRITE signal to memory



# Changing the Sequence of Instructions

In the FETCH phase, the Program Counter is incremented by 1.

But what if we don't want to always execute the instruction that follows this one?

- examples: loop, if-then, function call

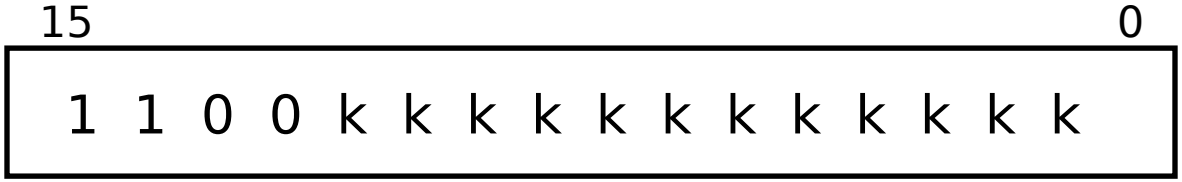
Need special instructions that change the contents of the PC.

These are called *control instructions*.

- jumps are unconditional -- they always change the PC
- branches are conditional -- they change the PC only if some condition is true (e.g., the result of an ADD is zero)
  - Redirecting the flow of control or "control-flow redirection".

# AVR rjmp Instruction

- Instruction Format




- Syntax: rjmp k  -2K <= k < 2K
- Operation: PC := PC + k (PC is already incremented)
- This instruction loads the PC with the value computed above...

  IT DOES NOT MATTER WHETHER OR NOT THE VALUE CAME FROM A VALID INSTRUCTION OR RESOLVES TO A VALID INSTRUCTION

# Instruction Processing Summary

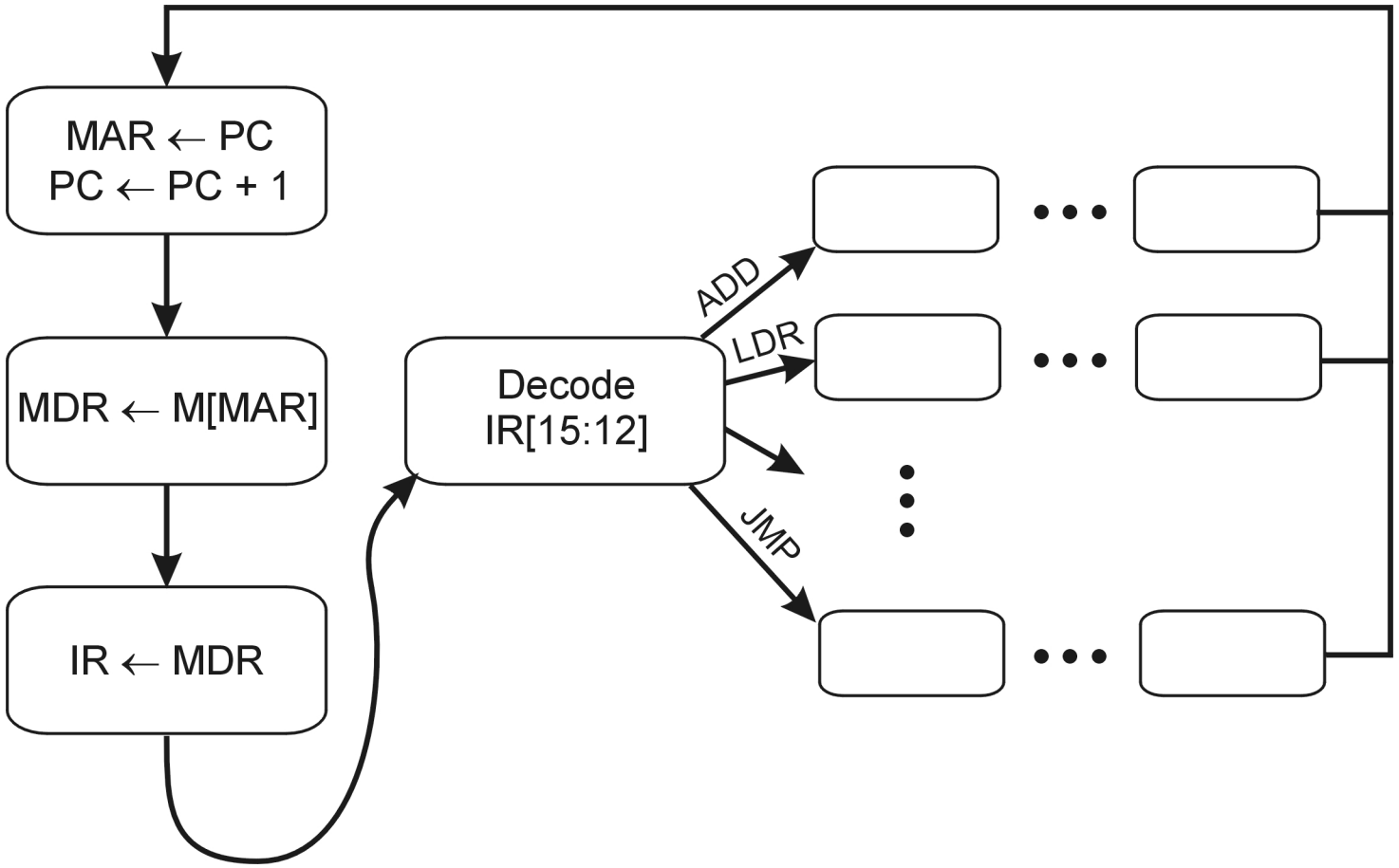
- Instructions look just like data -- it's all interpretation.

- Three basic kinds of instructions:
  - computational instructions (ADD, AND, …)
  - data movement instructions (LD, ST, …)
  - control instructions (JMP, BRxx, …)

- Six basic phases of instruction processing:
- $F \rightarrow D \rightarrow EA \rightarrow OP \rightarrow EX \rightarrow S$
  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles

# Control Unit State Diagram

- The control unit is a state machine. Here is part of a simplified state diagram for the AVR:



# Clarification

Every piece of software that you have ever run or probably ever *will* run on every computing machine that you have ever used or probably ever *will* use

**IS EXECUTED USING THIS METHOD**

Problems?