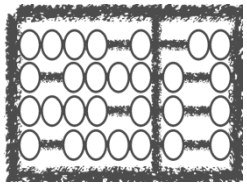
INF-741: Embedded systems programming for IoT

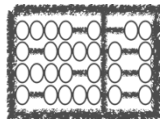
Prof. Edson Borin



Institute of Computing

University of Campinas

Embedded Systems Organization



Agenda

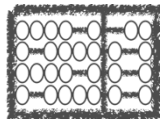
Processing Unit

Memory

Buses

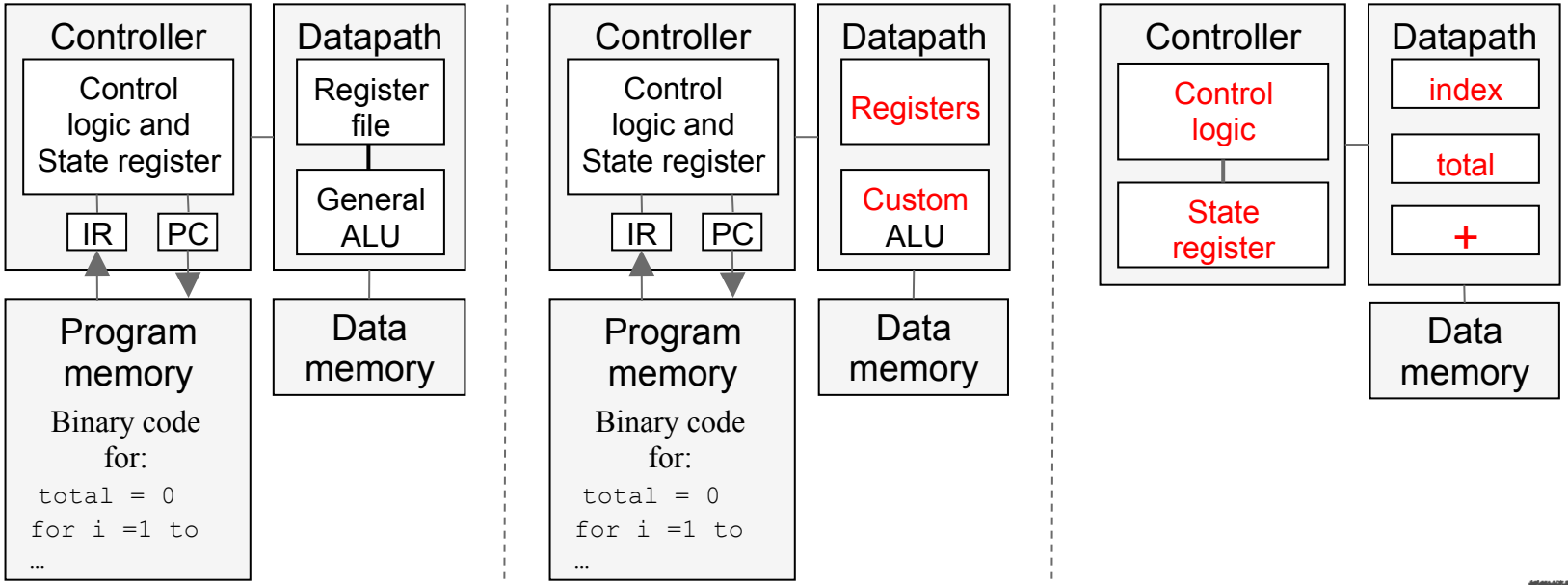
Peripherals

SoCs and MCUs



Processing Unit

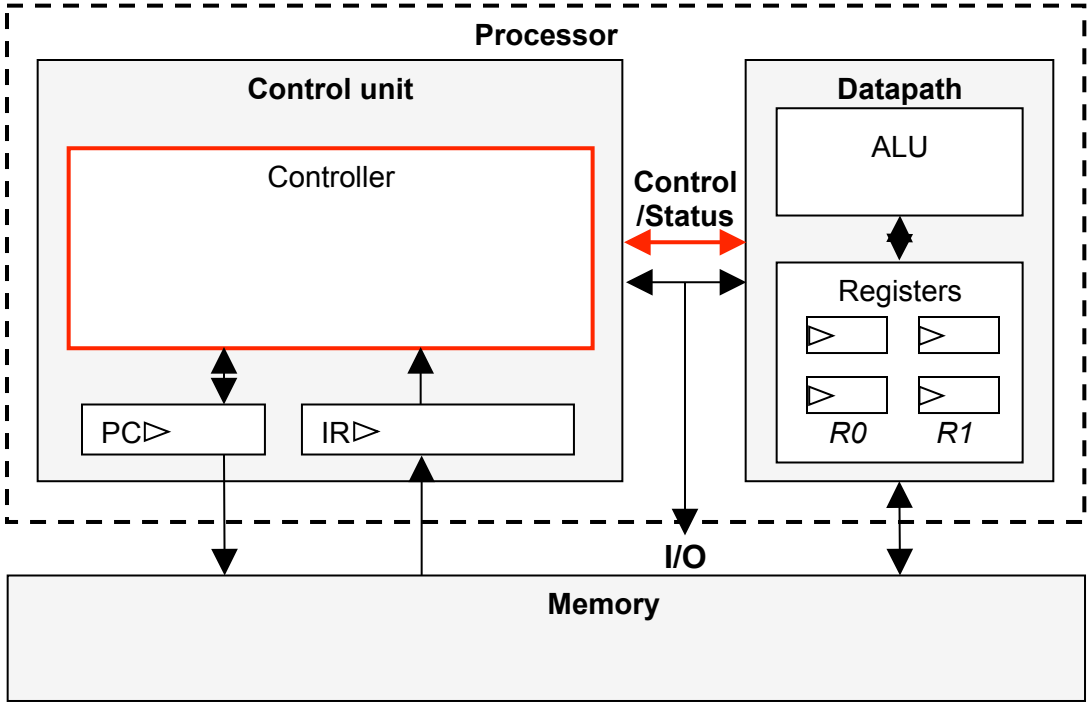
- GPP vs ASIP vs Single-Purpose Processors



General Purpose (*software*) Application Specific Single Purpose (*hardware*)

Processing Unit

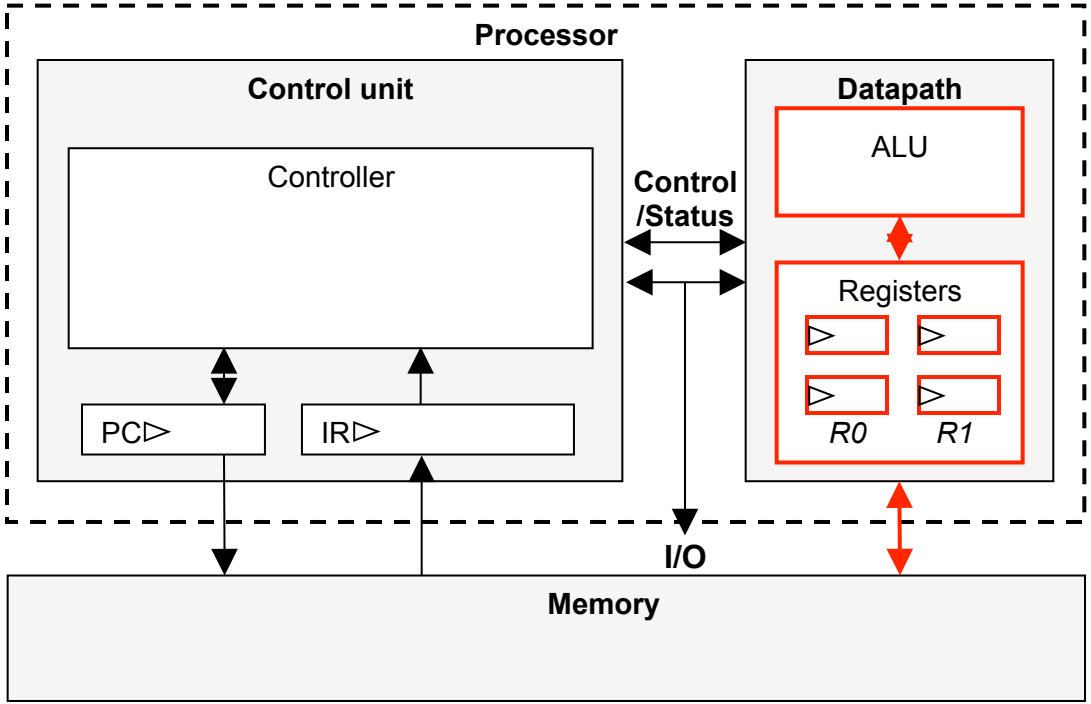
Control Unit: controls the processor operations.



Processing Unit

Control Unit: controls the processor operations.

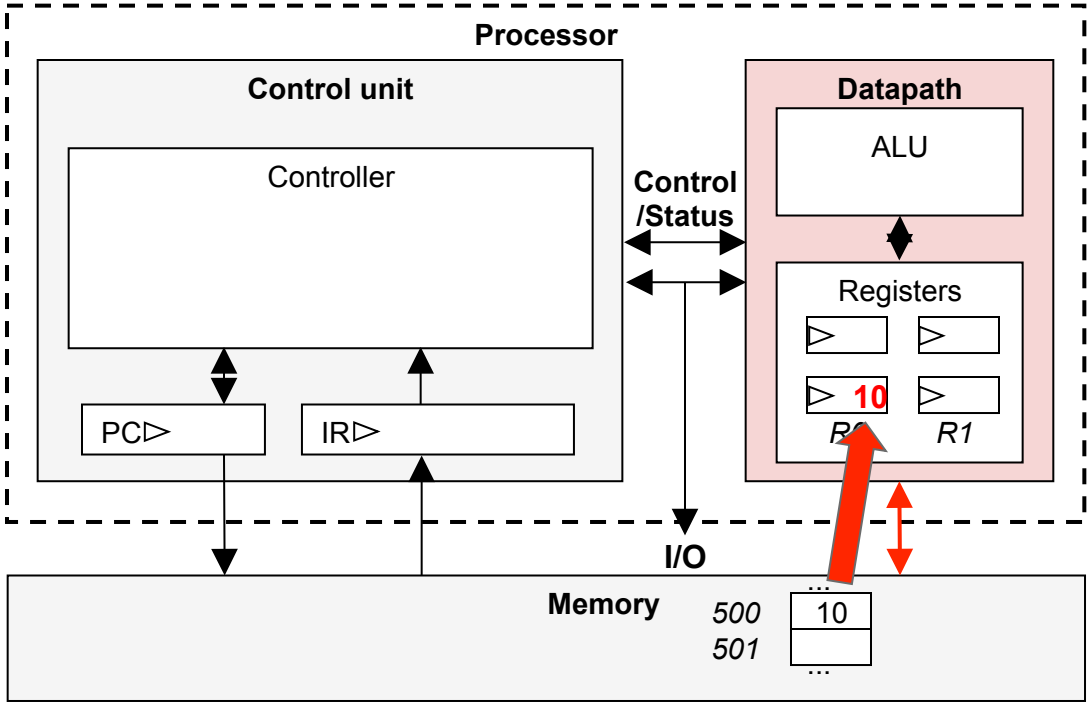
Datapath: collection of functional units, registers and buses to support operations on data.



Processing Unit

Datapath operations:

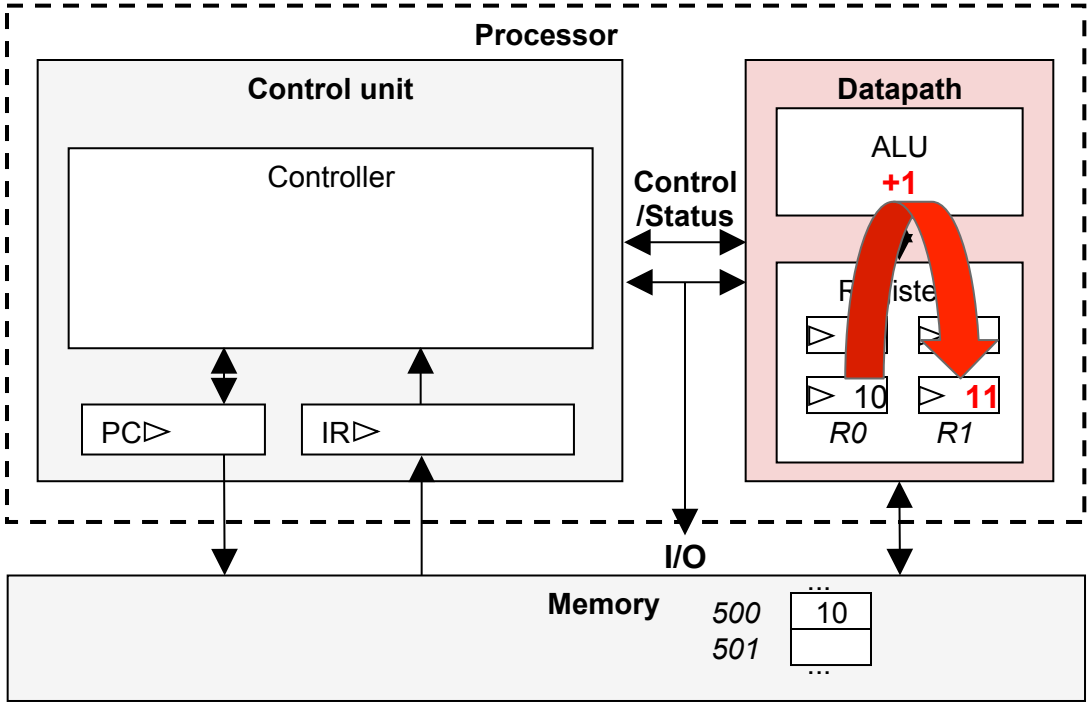
- **Load**: read memory location into register



Processing Unit

Datapath operations:

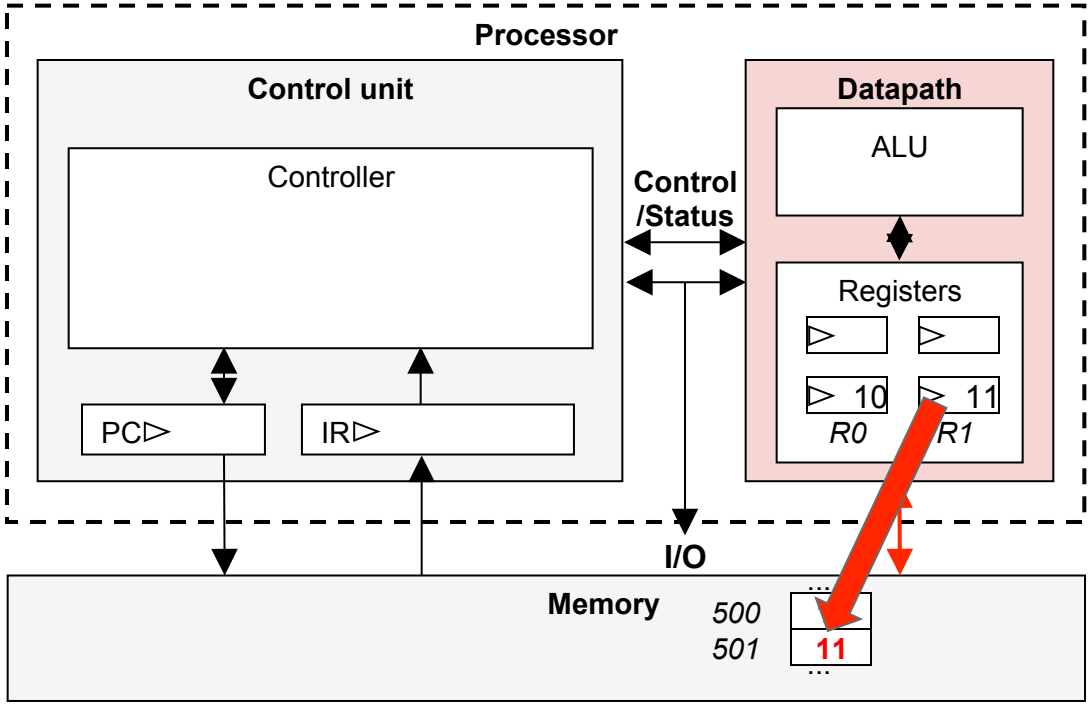
- **Load:** read memory location into register
- **ALU operation**: Input certain registers through ALU, store back in register



Processing Unit

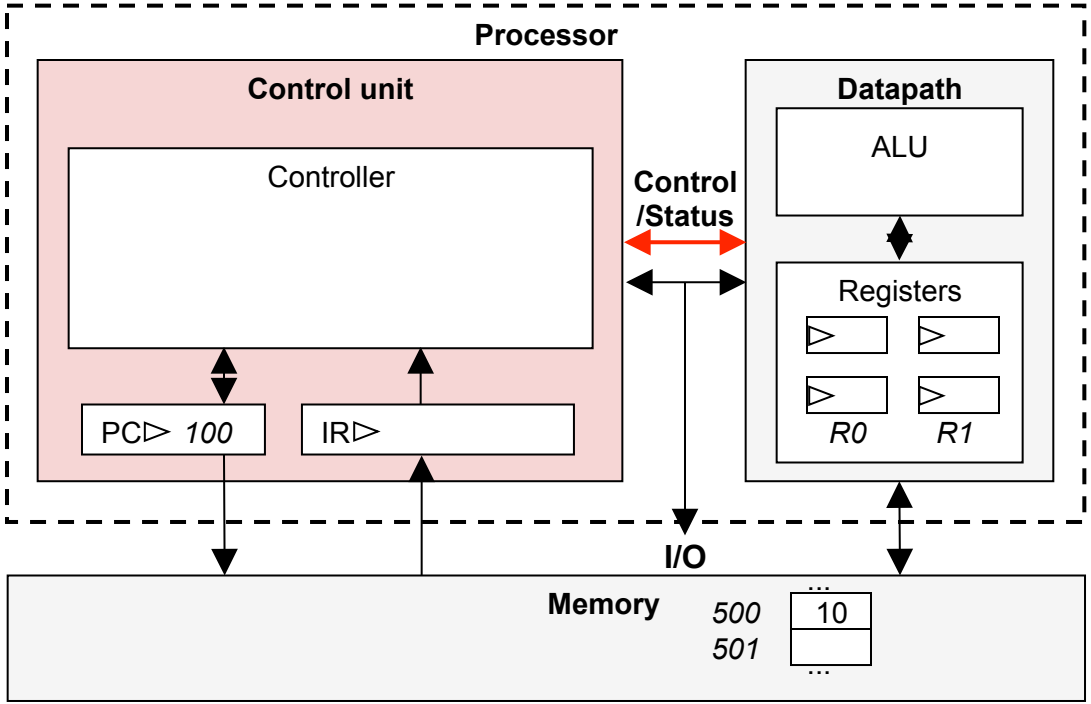
Datapath operations:

- **Load:** read memory location into register
- **ALU operation**: Input certain registers through ALU, store back in register
- **Store**: Write register contents to memory location



Processing Unit

Control Unit: control the processor operations!

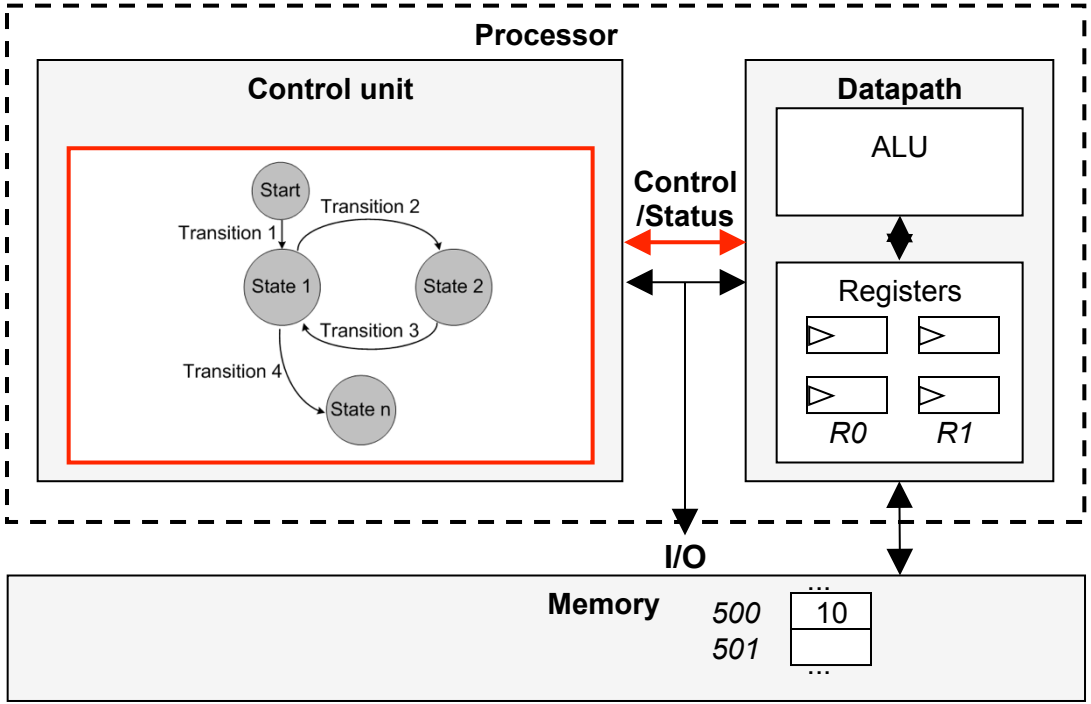


Processing Unit

Control Unit: control the processor operations!

Single Purpose Processors: sequence of operations is coded inside the control unit.

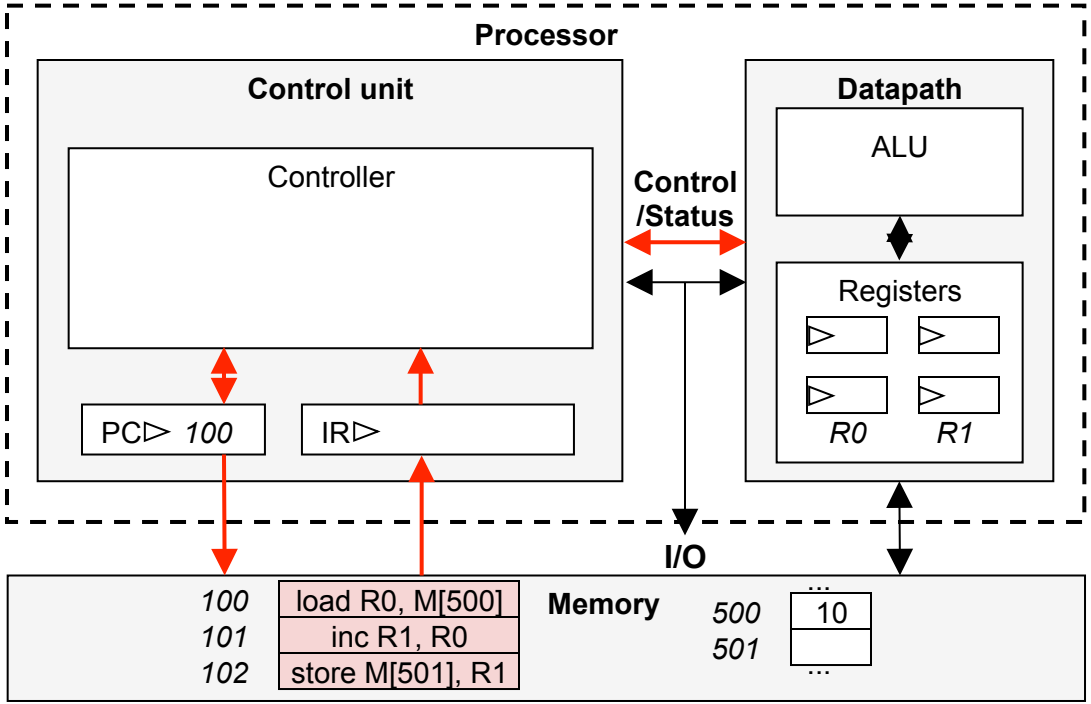
- Ex: using a finite state machine.



Processing Unit

Control Unit: control the processor operations!

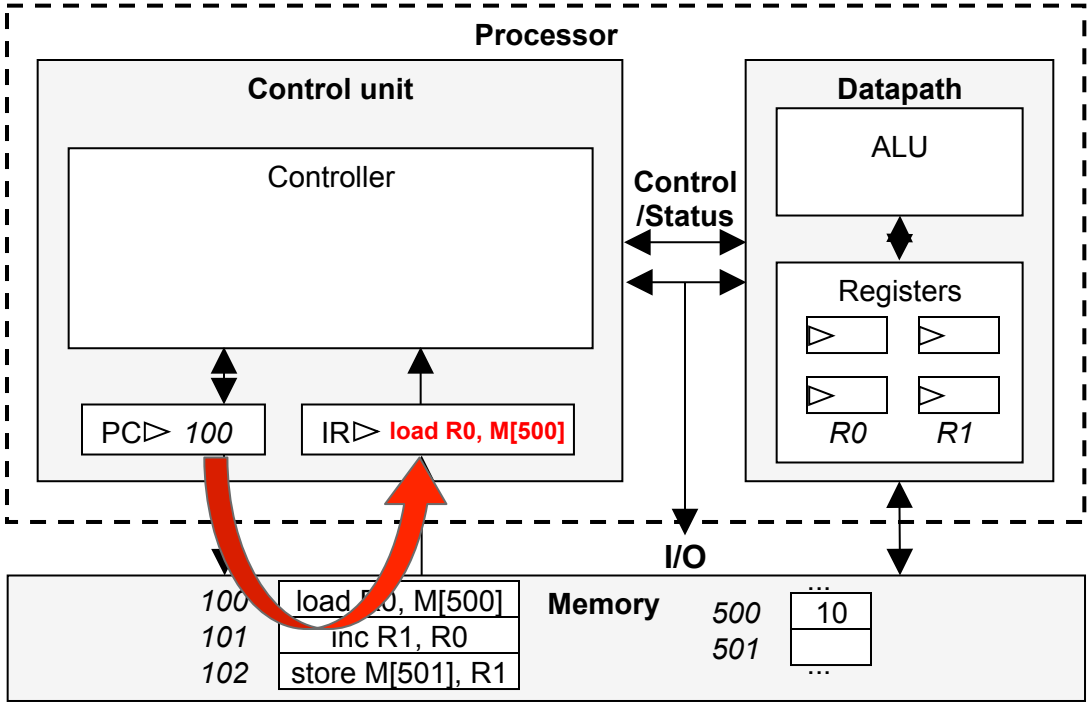
Programmable processors (GPPs and ASIPs): sequence of operations are defined by the program instructions, which are stored on memory



Processing Unit

Instruction Cycle: several sub-operations. Ex:

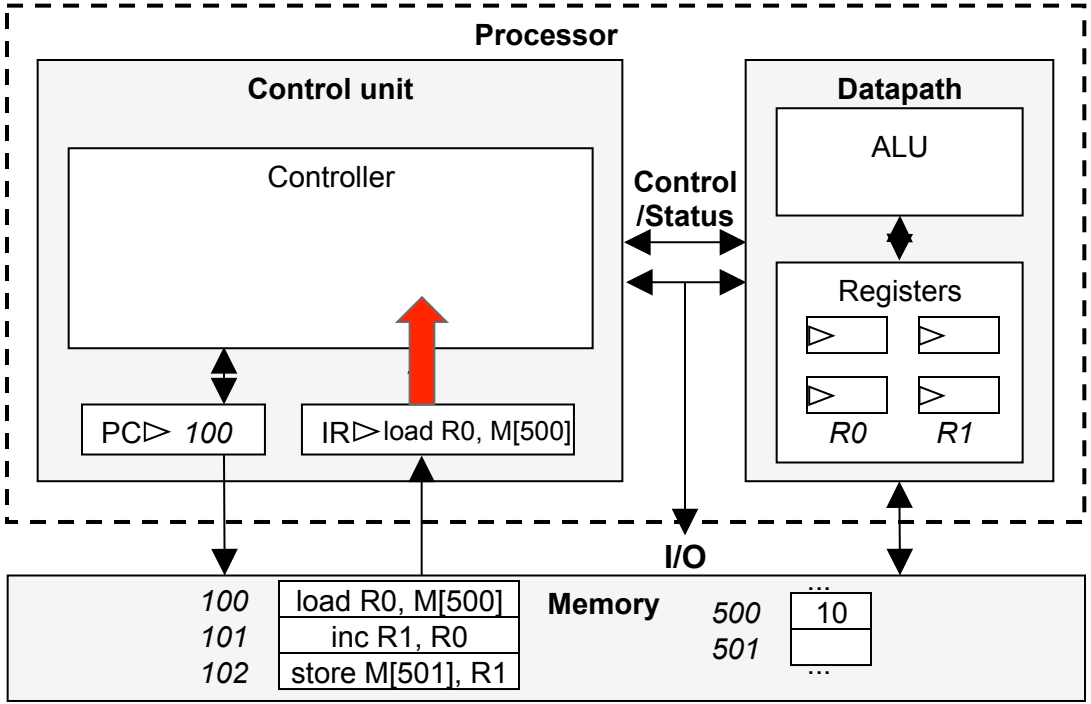
- **Fetch**: get next instruction into IR.



Processing Unit

Instruction Cycle: several sub-operations. Ex:

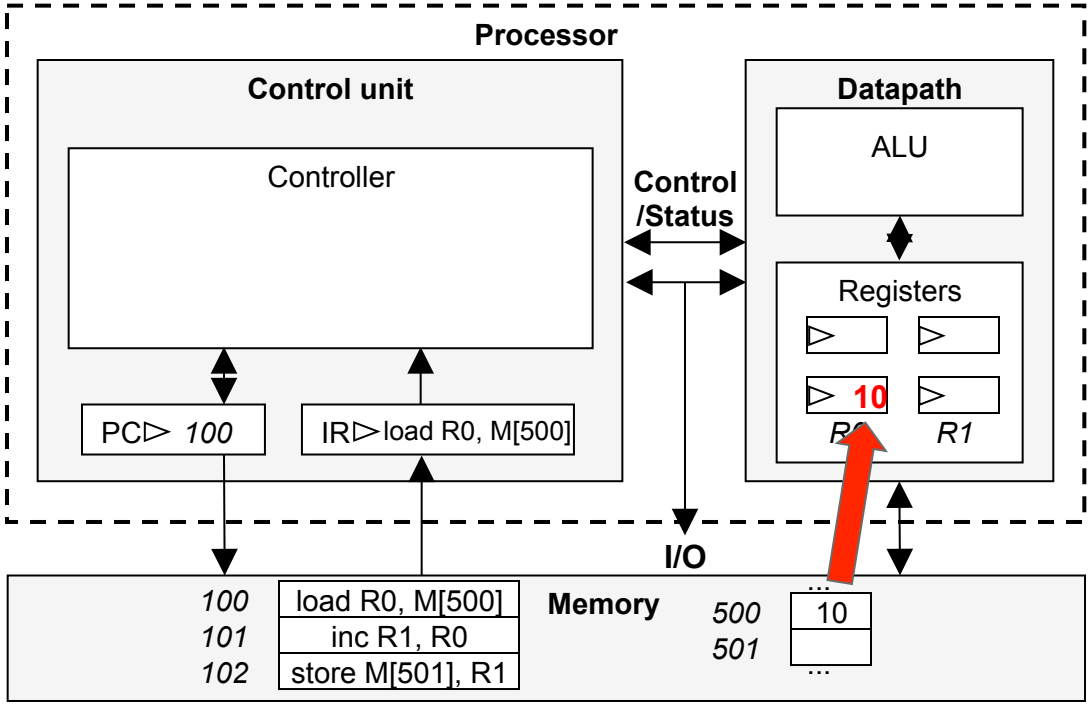
- **Fetch**: get next instruction into IR.
- **Decode**: Determine what the instruction means.



Processing Unit

Instruction Cycle: several sub-operations. Ex:

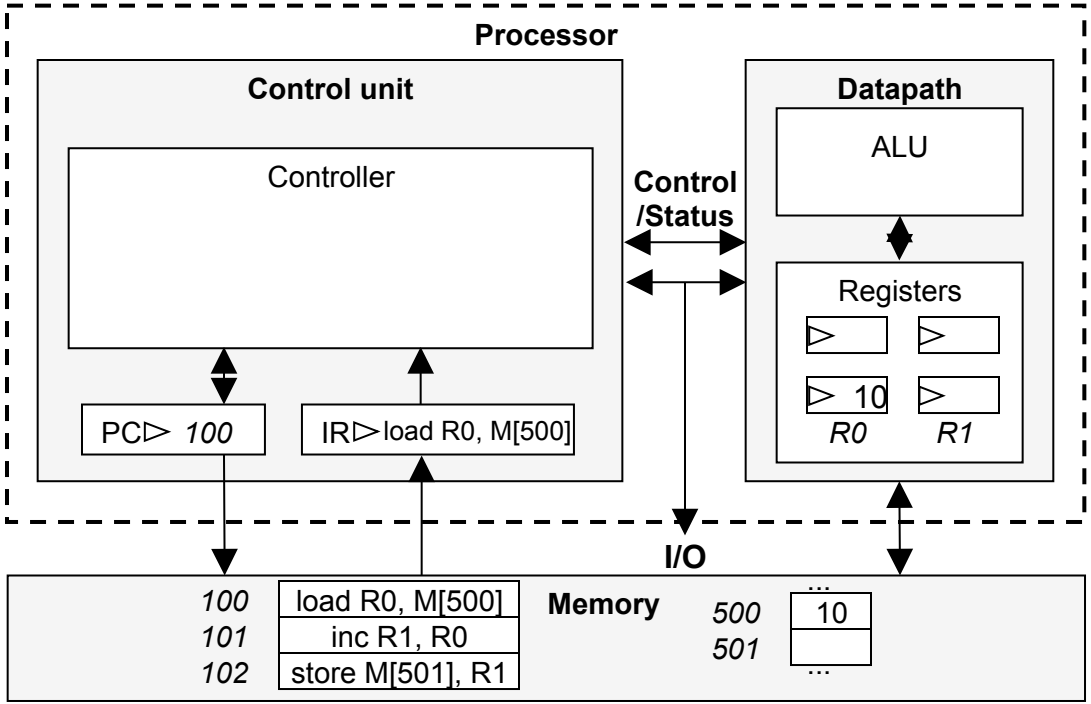
- **Fetch**: get next instruction into IR.
- **Decode**: Determine what the instruction means.
- **Fetch operands**: Move data from memory to datapath register.



Processing Unit

Instruction Cycle: several sub-operations. Ex:

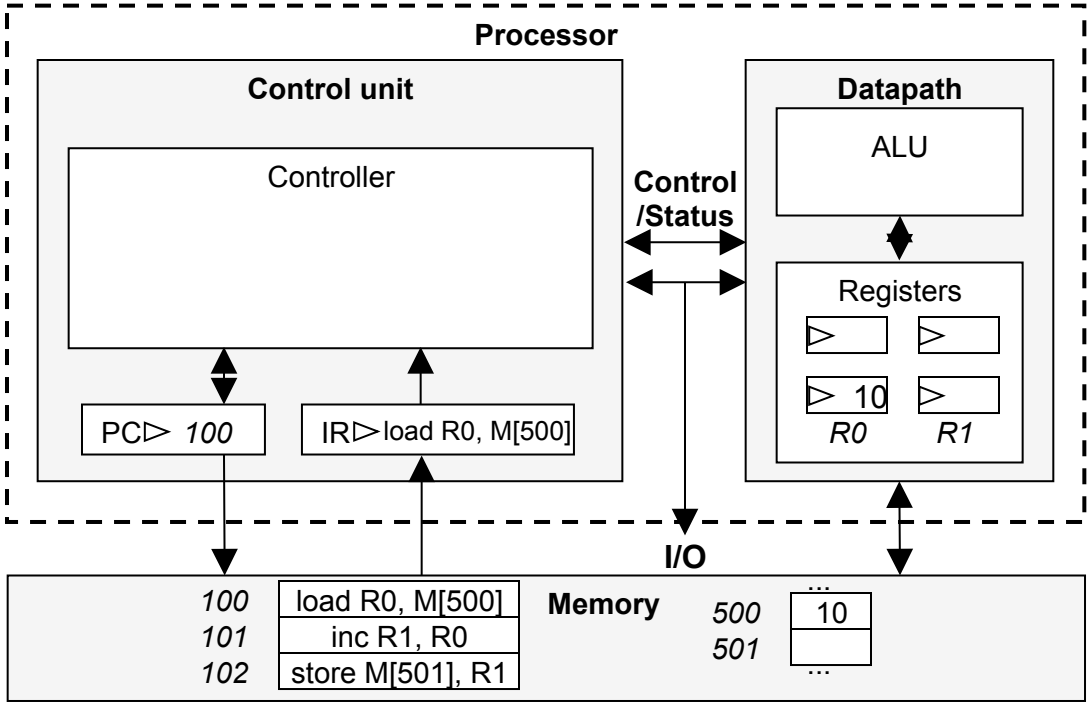
- **Fetch**: get next instruction into IR.
- **Decode**: Determine what the instruction means.
- **Fetch operands**: Move data from memory to datapath register.
- **Execute**: Move data through the ALU.



Processing Unit

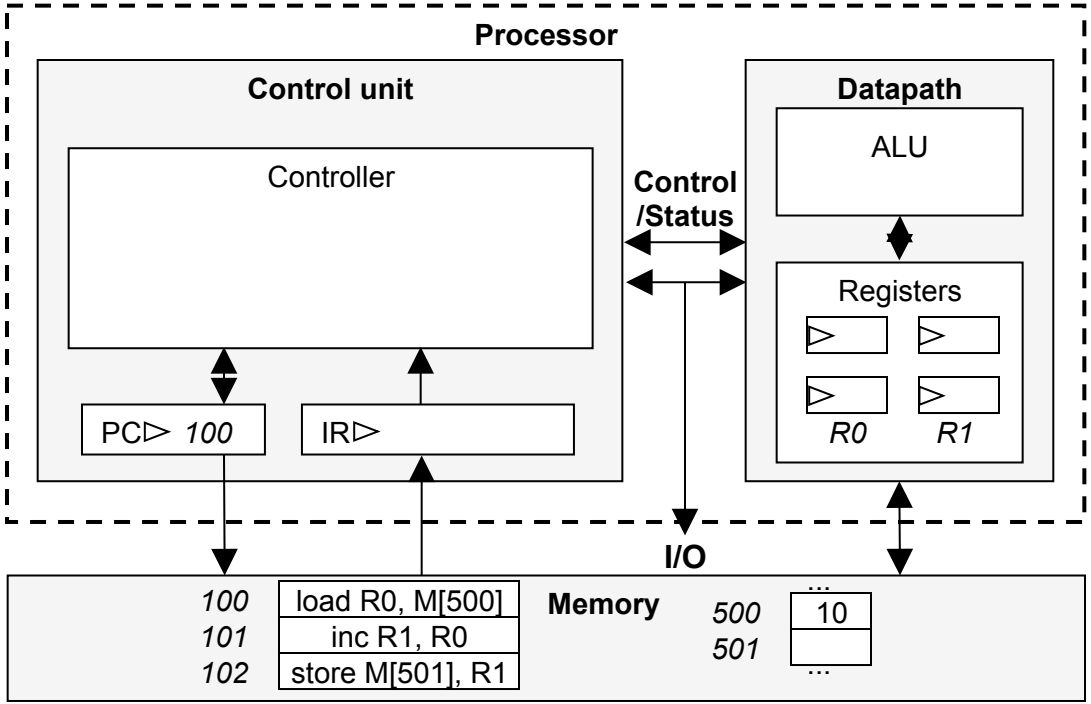
Instruction Cycle: several sub-operations. Ex:

- **Fetch**: get next instruction into IR.
- **Decode**: Determine what the instruction means.
- **Fetch operands**: Move data from memory to datapath register.
- **Execute**: Move data through the ALU.
- **Store results**: Write data from memory to register



Processing Unit

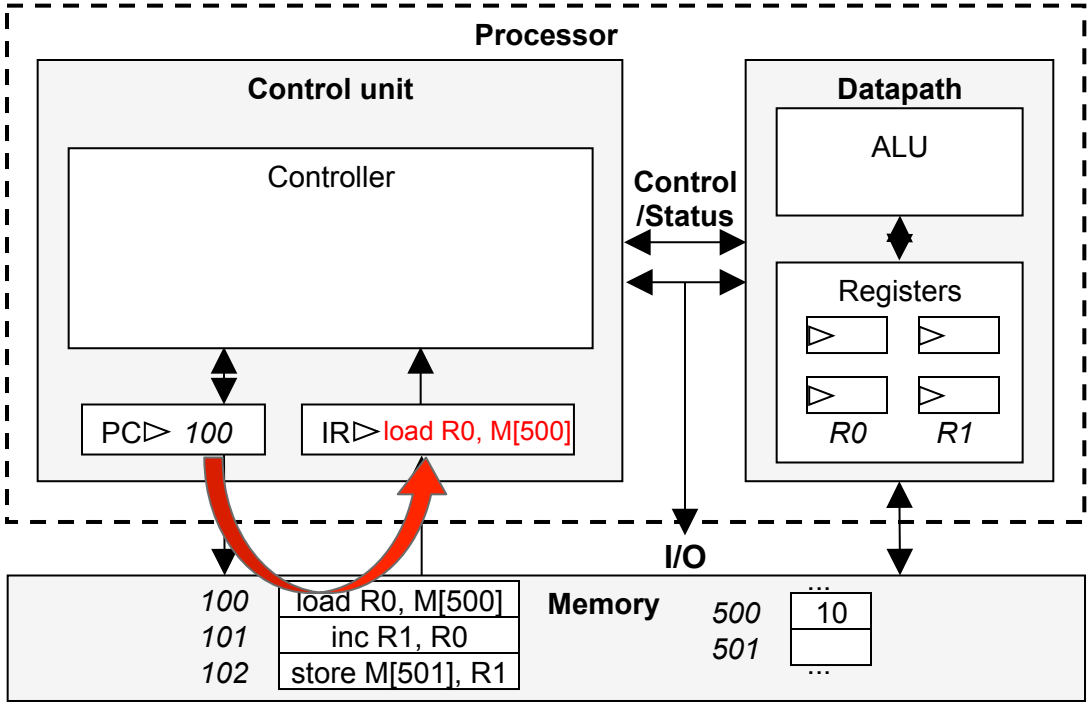
PC=100

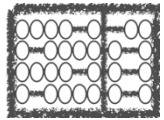


Processing Unit

PC=100

Fetch

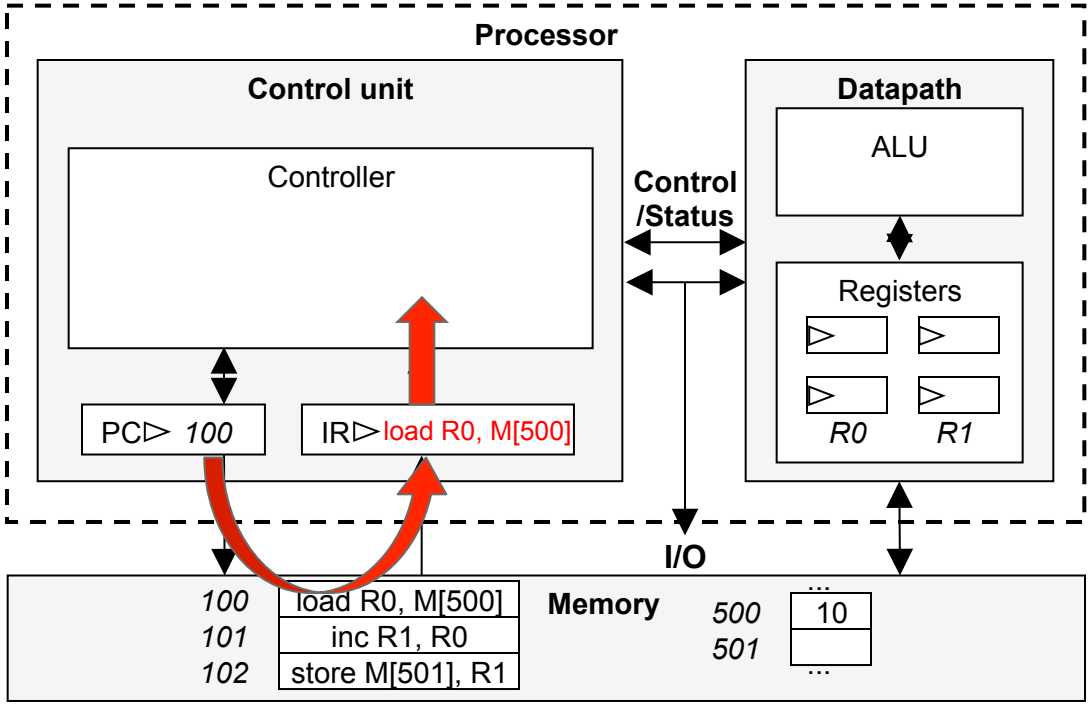




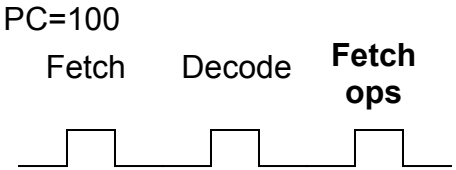
Processing Unit

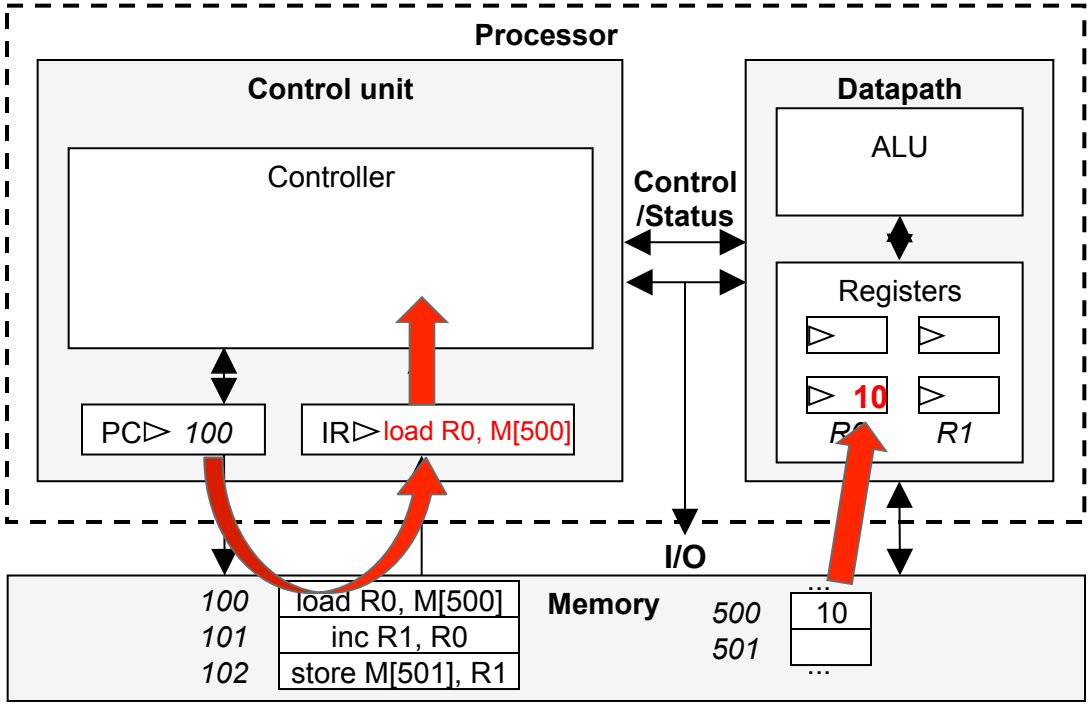
PC=100

Fetch **Decode**

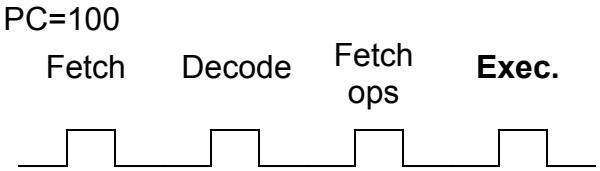


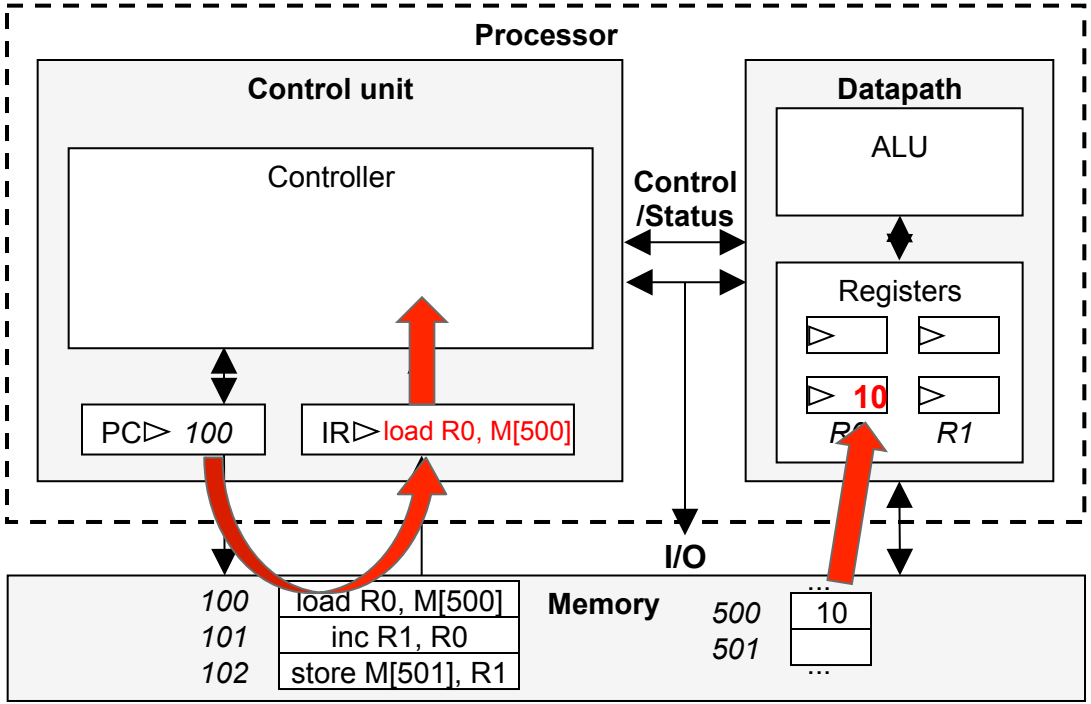
Processing Unit



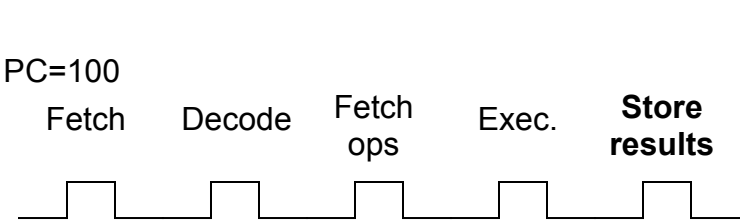


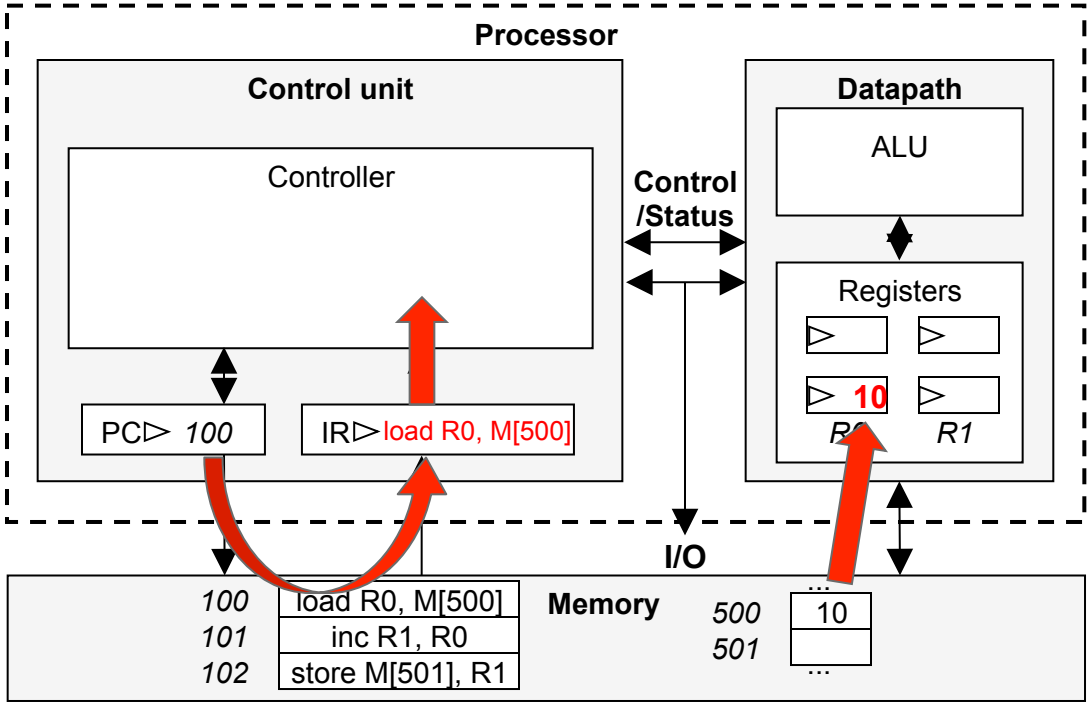
Processing Unit



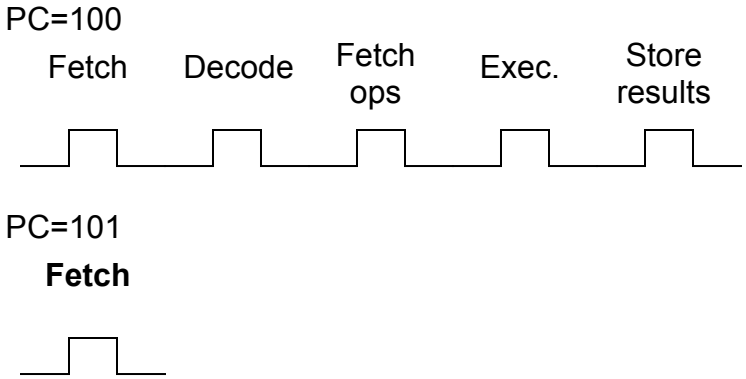


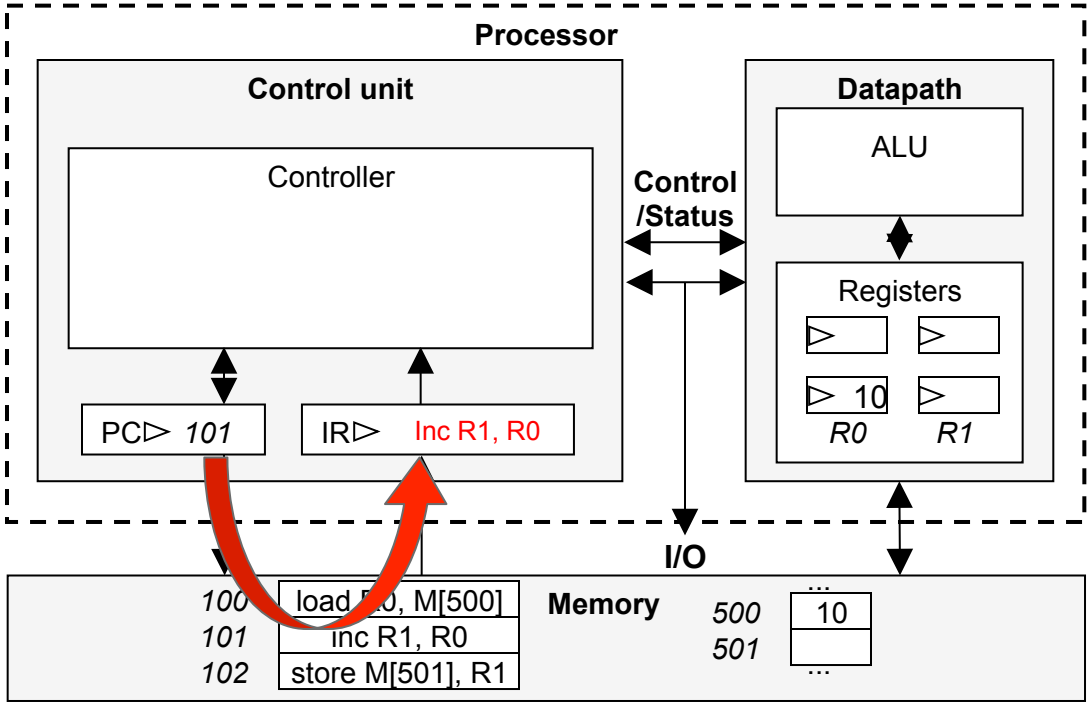
Processing Unit



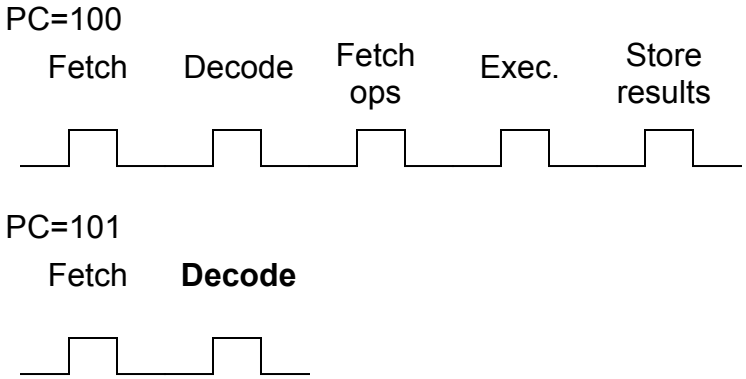


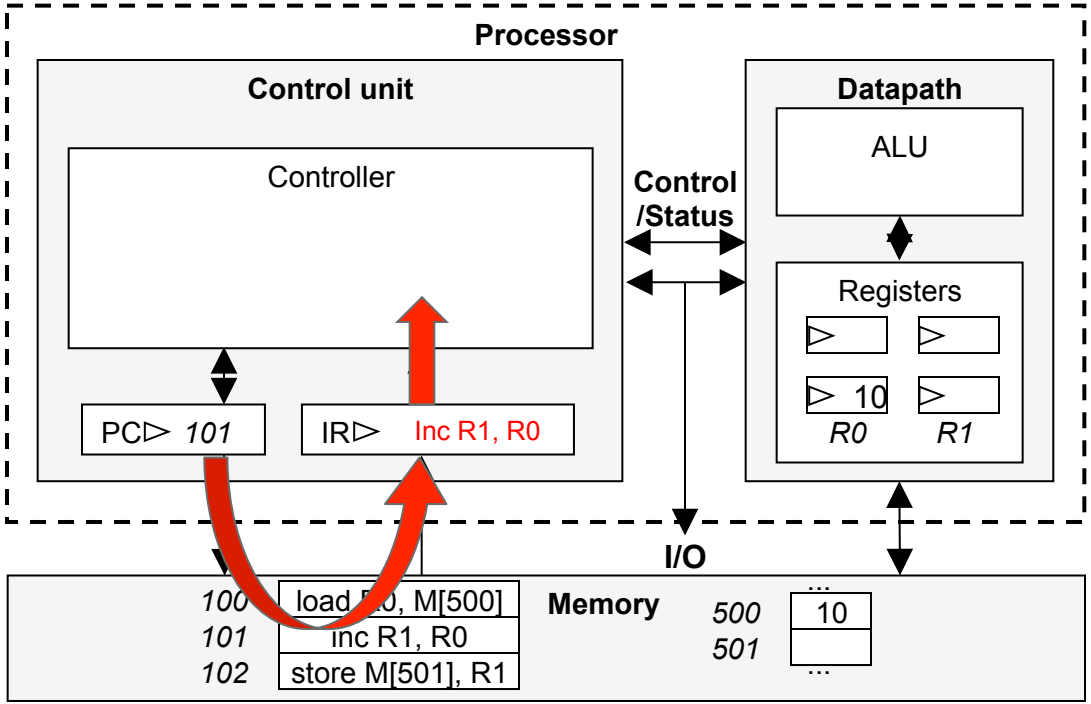
Processing Unit



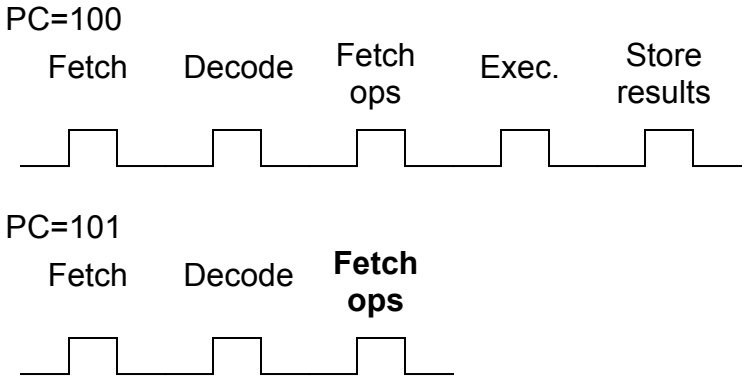


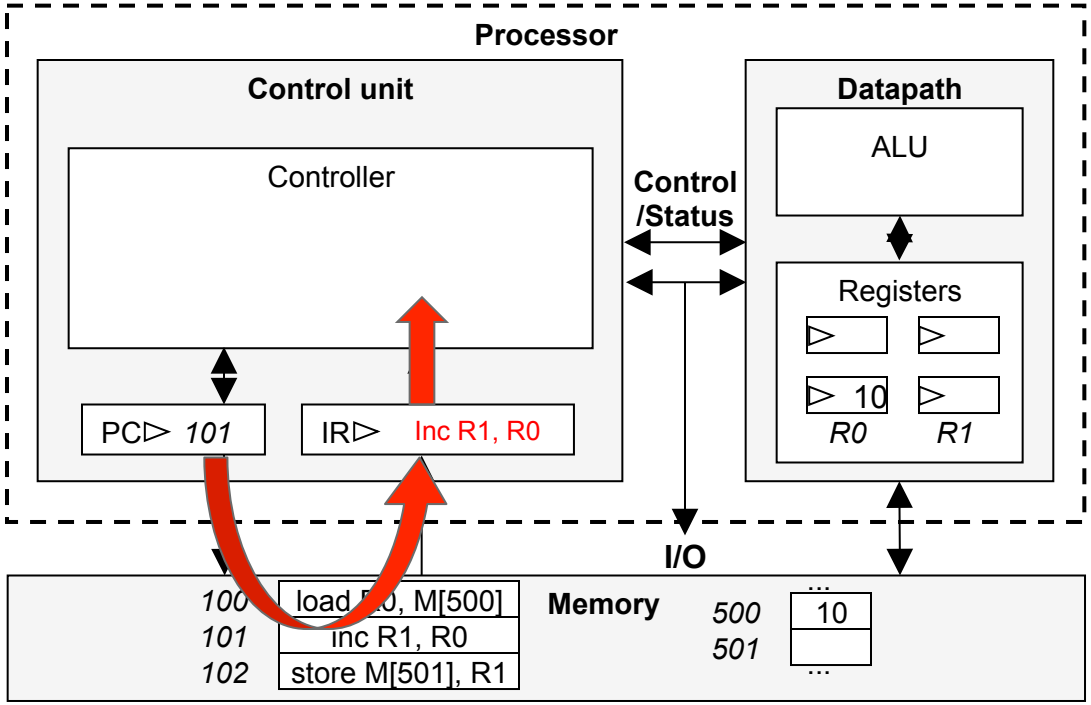
Processing Unit



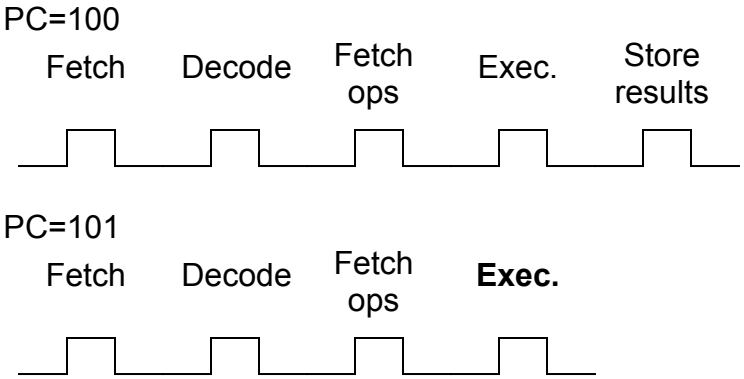


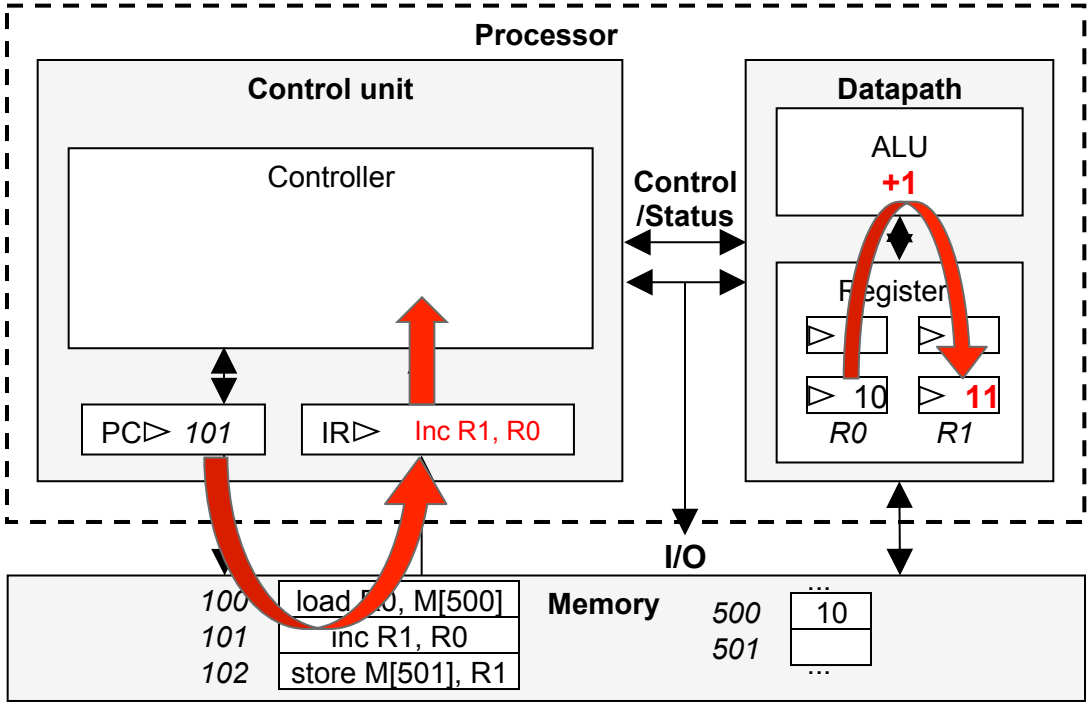
Processing Unit



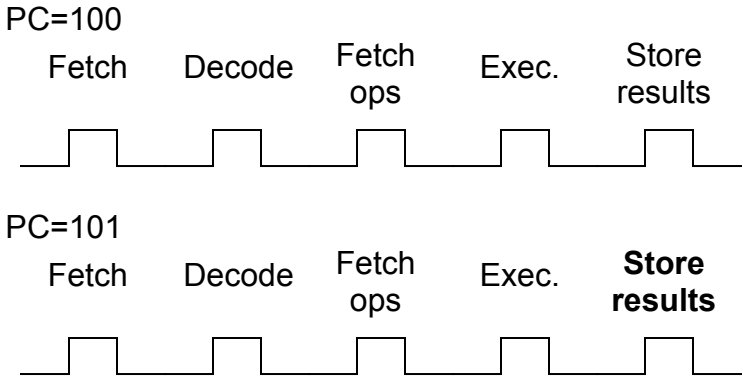


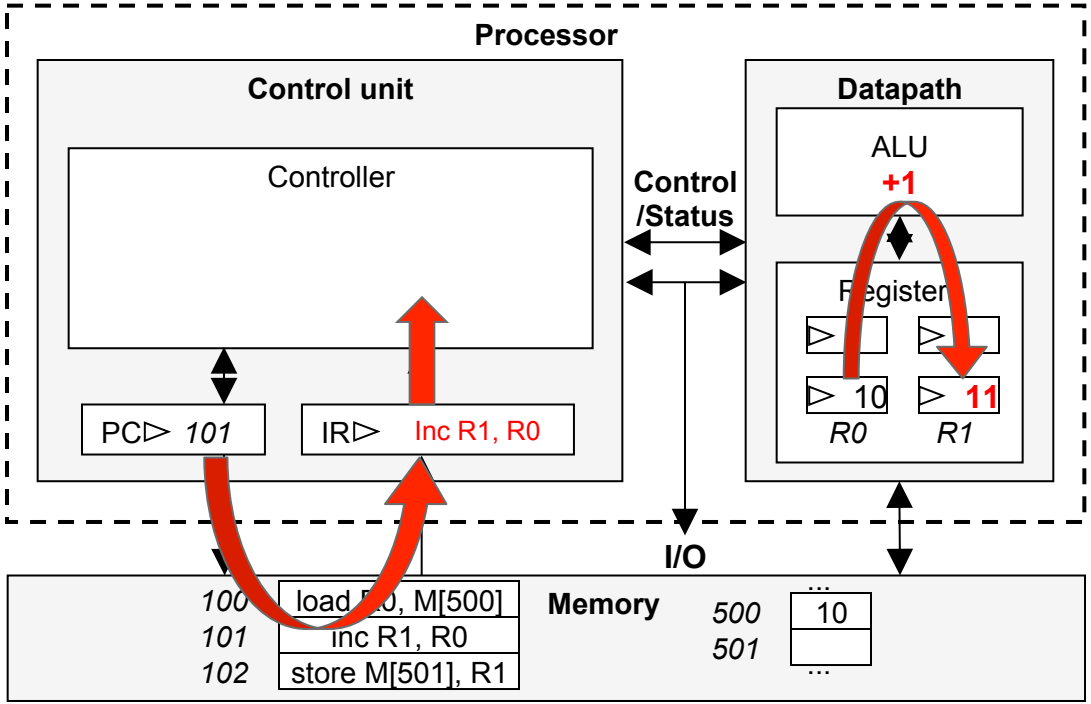
Processing Unit



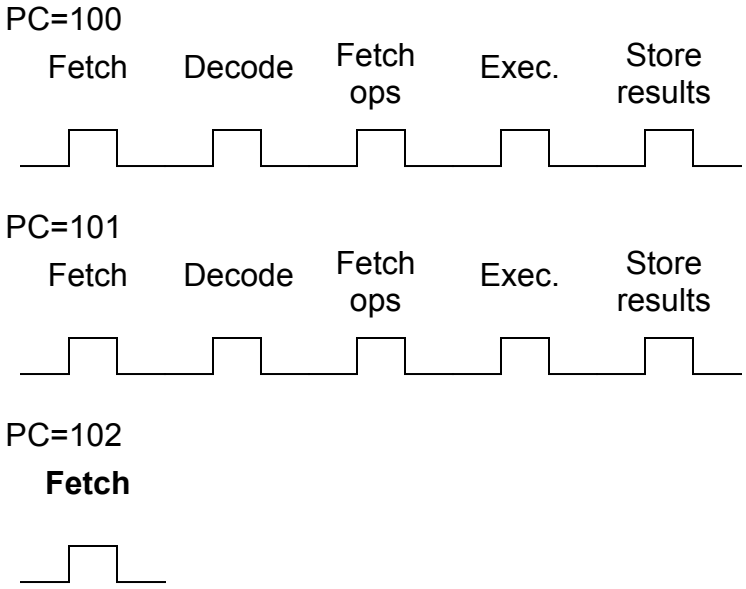


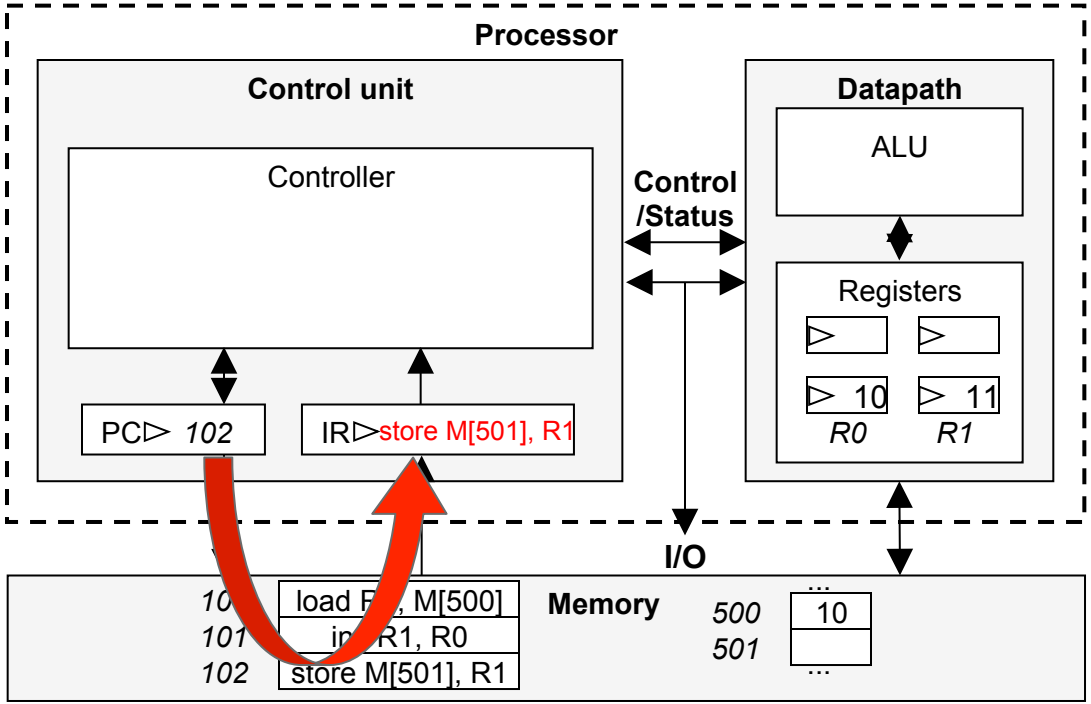
Processing Unit



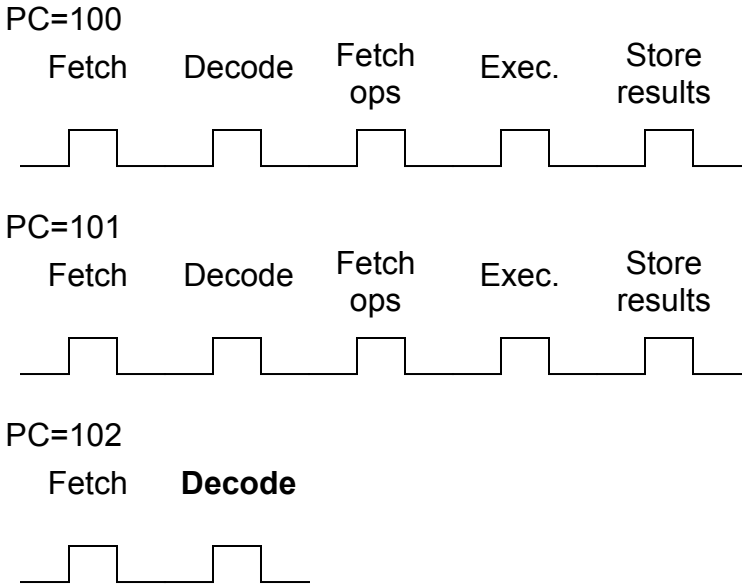


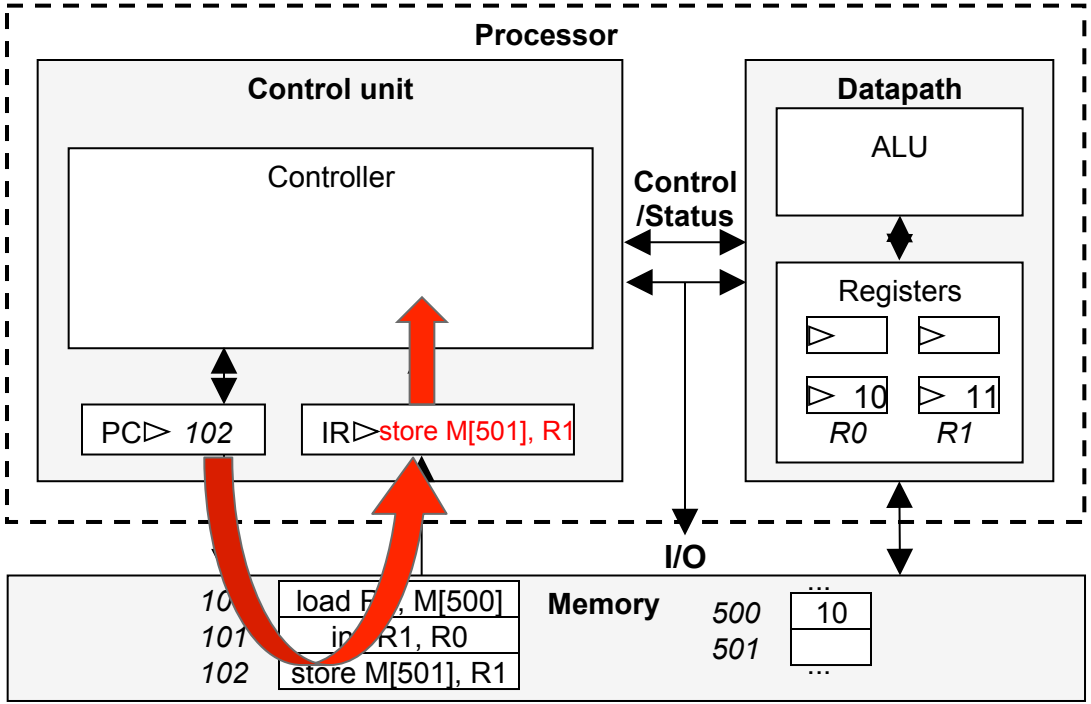
Processing Unit



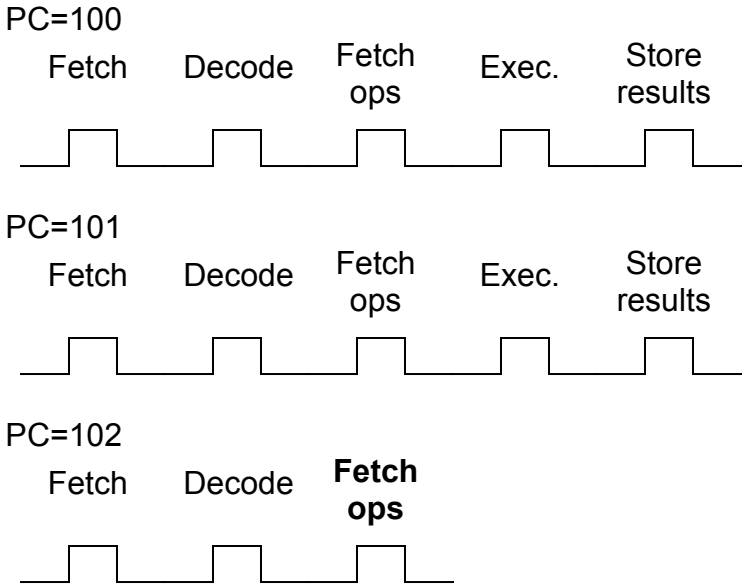


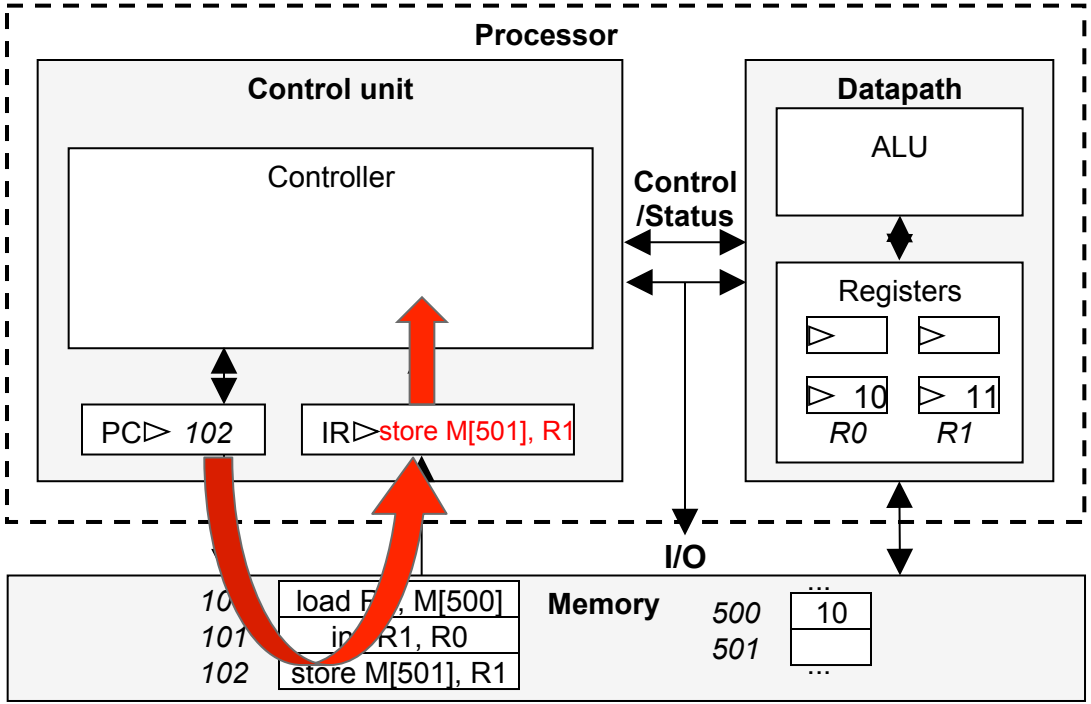
Processing Unit



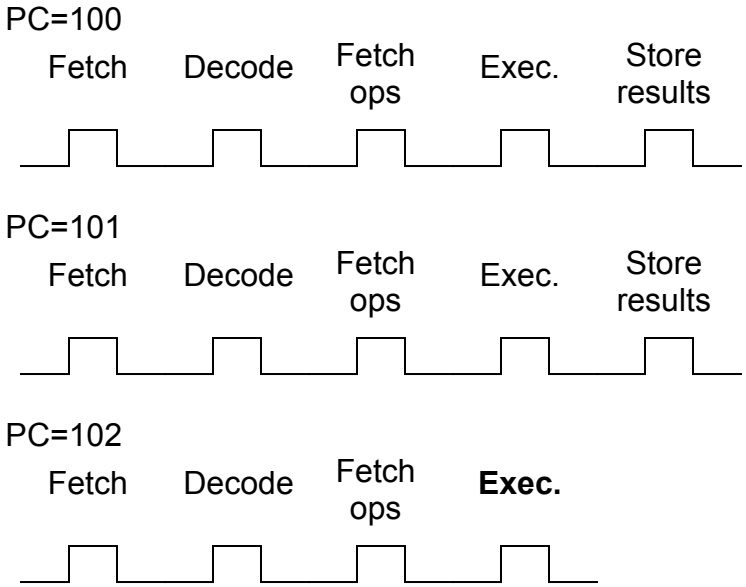


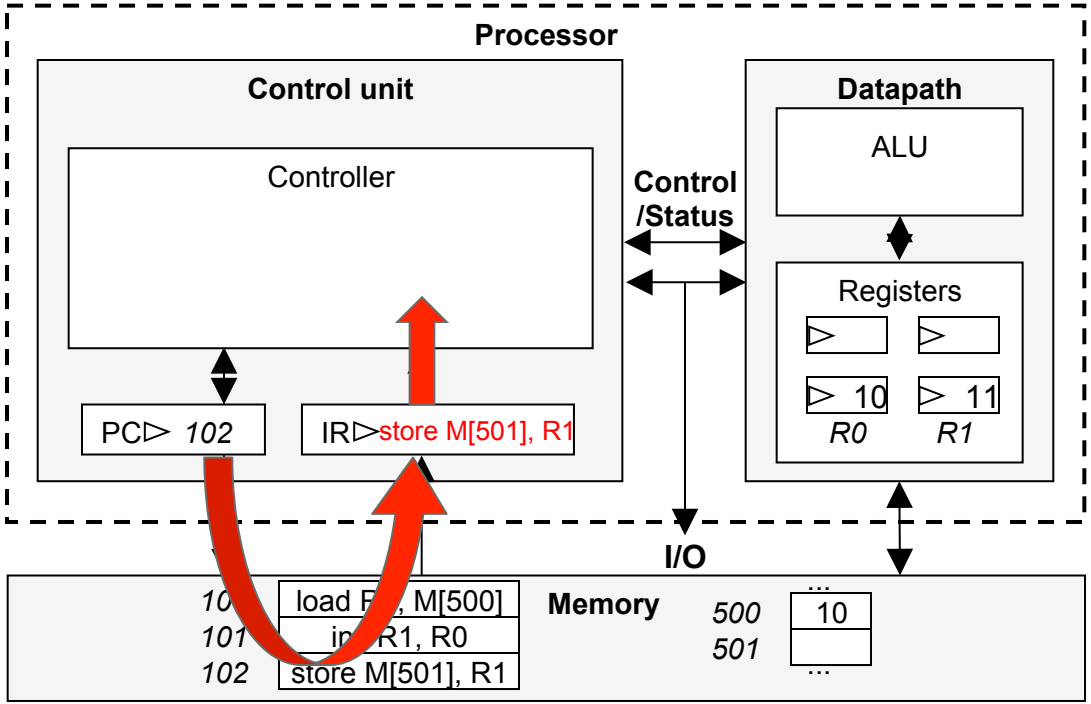
Processing Unit



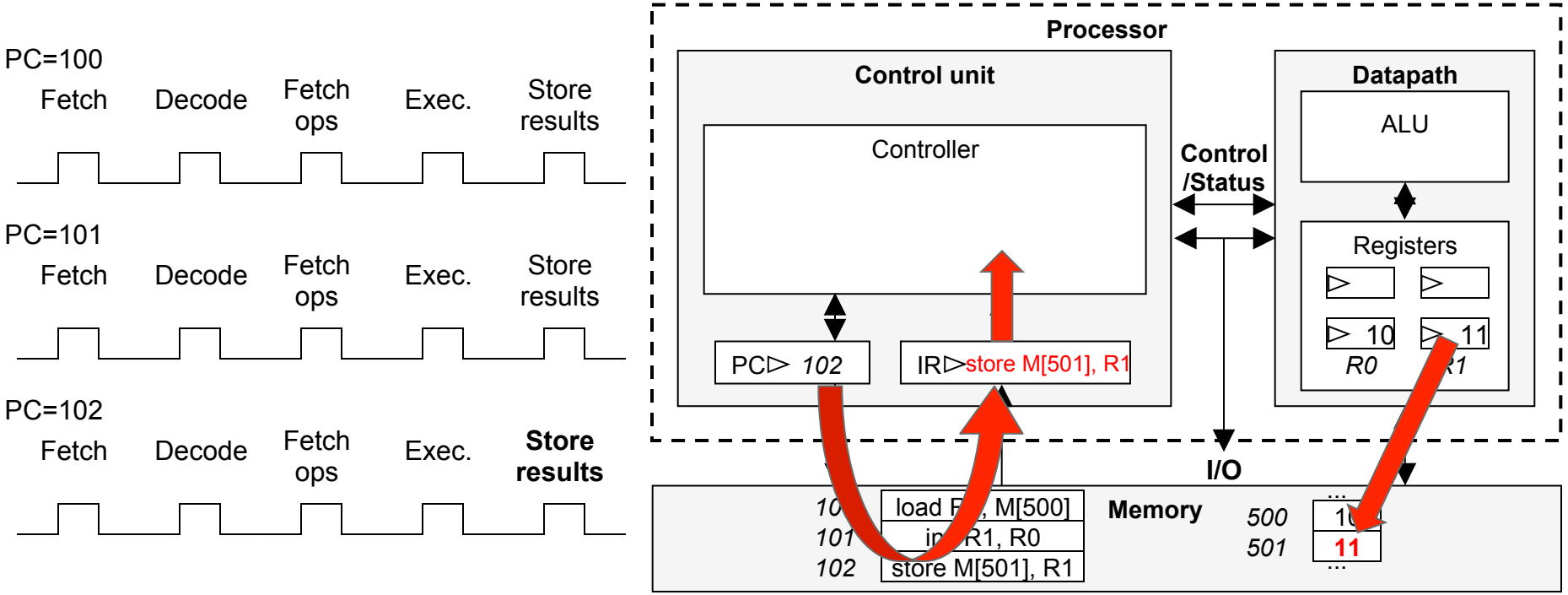


Processing Unit

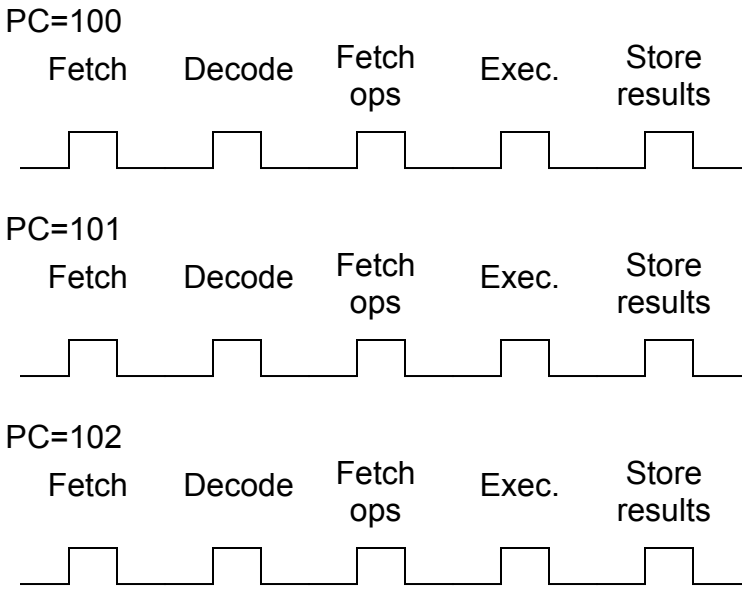


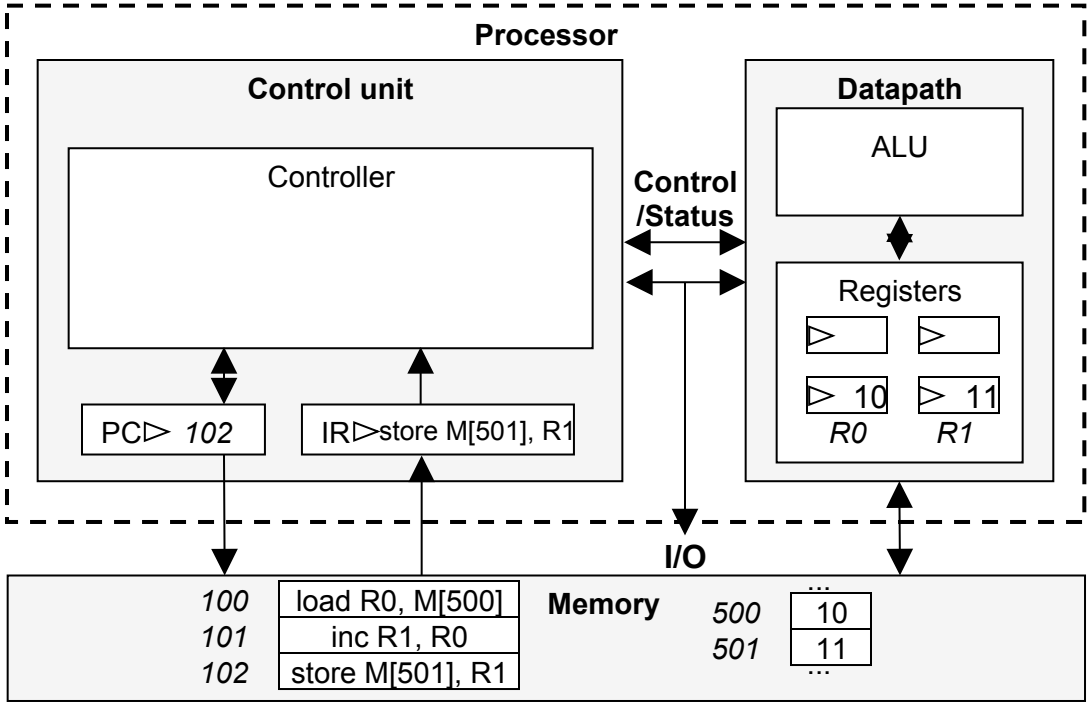


Processing Unit



Processing Unit

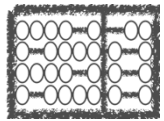




Processing Unit - Programming

Instruction Set Architecture: the part of the computer architecture related to programming, including the **native data types**, **instructions**, **registers**, **addressing modes**, **memory architecture**, **interrupt** and **exception handling**, and **external I/O**.

Microarchitecture: also called computer organization and sometimes abbreviated as μarch or uarch, is "the way a given instruction set architecture (ISA) is implemented in a particular processor". Ex: executing instructions with 5 sub-operations.

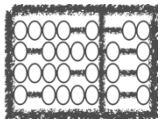


Processing Unit - Programming

Instruction Set Architecture: the part of the computer architecture related to programming, including the **native data types**, **addressing modes**, **[illegible]** and **external I/O**.

Microarchitecture: [illegible] as µarch or uarch, is [illegible] implemented in a parti[illegible] operations.

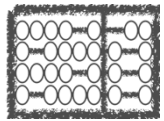
The programmer needs to know how to program the processor (**ISA**), but does not need to know how the processor executes each instruction (**microarchitecture**).



Processing Unit - Programming

Instruction Set Architecture - Ex. ARM

- User visible registers:
  - R0, R1, R2, ... , R15 -- Does not include IR!
- Instructions:
  - Data transfer: MOV R0, R1; LDR R0, [R1], STR R2, [R4], …
  - Arithmetic: ADD R1, R2, R3; MUL R2, R3, R6; …
  - Branches: BR target1; BREQ target2; …



Processing Unit - Programming

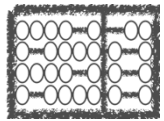
Source program in C

```
int a;
int b;
int c;

void foo()
{
 a = b + c;
}
```

x86 Assembly Language

```
_foo:
 movq    _c@GOTPCREL(%rip), %rax
 movl    (%rax), %eax
 movq    _b@GOTPCREL(%rip), %rdx
 addl    (%rdx), %eax
 movq    _a@GOTPCREL(%rip), %rdx
 movl    %eax, (%rdx)
 ret
```



Processing Unit - Programming

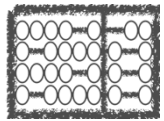
x86 Assembly Language

```
_foo:
 movq    _c@GOTPCREL(%rip), %rax
 movl    (%rax), %eax
 movq    _b@GOTPCREL(%rip), %rdx
 addl    (%rdx), %eax
 movq    _a@GOTPCREL(%rip), %rdx
 movl    %eax, (%rdx)
 ret
```

Assembler

x86 Machine Language

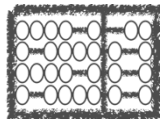
```
 0: 48 8b 05 00 00 00 00
 7: 8b 00
 9: 48 8b 15 00 00 00 00
10: 03 02
12: 48 8b 15 00 00 00 00
19: 89 02
1b: c3
```



Processing Unit - Programming

Instruction Set Architecture

- **Registers:** internal storage, similar to memory. Often identified by names instead of memory addresses. Ex: R0, R1, EAX, EBX, ...
- **Instructions:** a single operation of a processor defined by the processor instruction set.
  - Data handling and memory operations: move data between registers and the memory;
  - Arithmetic and logic operations: transform data stored on registers;
  - Control flow operations: change the flow of execution -- call, ret, jump, ...



Agenda

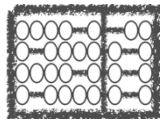
Processing Unit

Memory

Buses

Peripherals

SoCs and MCUs



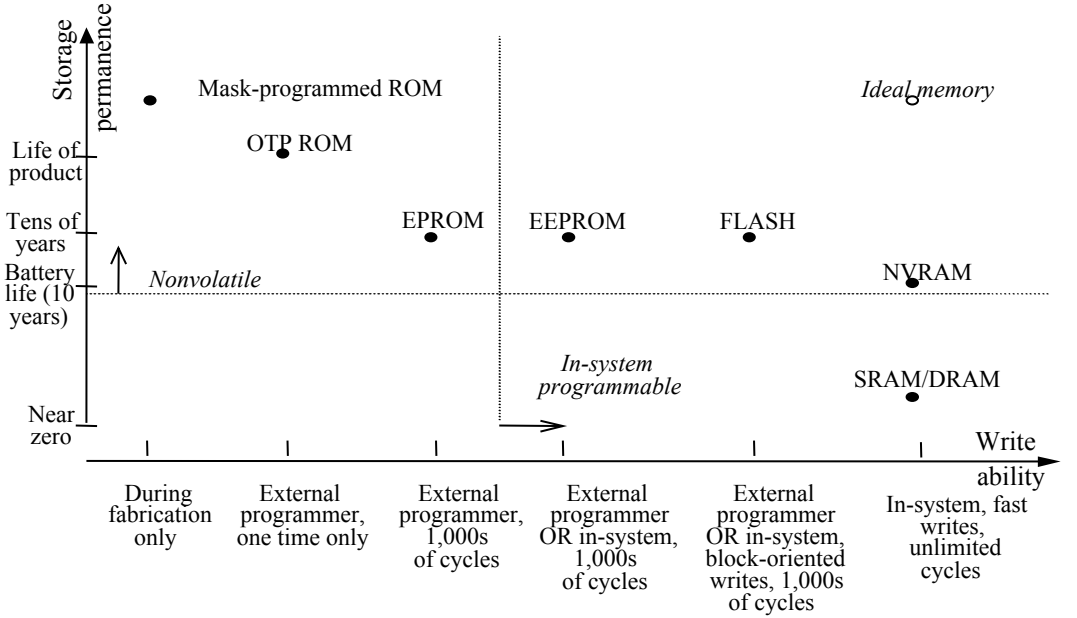
Memory

Traditional ROM/RAM distinctions

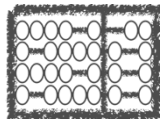
- ROM: read only, bits stored without power
- RAM: read and write, lose stored bits without power

Traditional distinctions blurred

- Advanced ROMs can be written to - e.g., EEPROM
- Advanced RAMs can hold bits without power - e.g., NVRAM

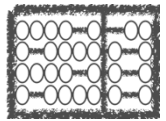


Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).



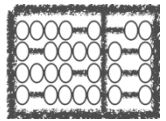
Memory - ROM

- "Read Only Memory"
- Nonvolatile memory
- Can be read from but not written to, by a processor in an embedded system
- Traditionally written to, “programmed”, before inserting to embedded system
- Uses
  - Store software program for general-purpose processor
    - program instructions can be one or more ROM words
  - Store constant data needed by system
  - Implement combinational circuit



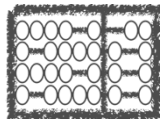
Memory - Mask-Programmed ROM

- Connections “programmed” at fabrication
  - set of masks
- Lowest write ability
  - only once
- Highest storage permanence
  - bits never change unless damaged
- Typically used for final design of high-volume systems
  - spread out NRE cost for a low unit cost



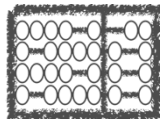
Memory - OTP ROM

- Connections “programmed” after manufactured by user
  - user provides file of desired contents of ROM
  - file input to machine called ROM programmer
  - each programmable connection is a fuse
  - ROM programmer blows fuses where connections should not exist
- Very low write ability
  - typically written only once and requires ROM programmer device
- Very high storage permanence
  - bits don’t change unless reconnected to programmer and more fuses blown
- Commonly used in final products
  - cheaper, harder to inadvertently modify



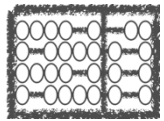
Memory - EPROM

- Erasable programmable ROM
- Programmable component is a MOS transistor
  a. May apply voltage to program gates as logic 0
  b. (Erase) Shining UV rays on surface of floating-gate causes negative charges to return to channel from floating gate restoring the logic 1
- Better write ability
  a. can be erased and reprogrammed thousands of times
- Reduced storage permanence
  a. program lasts about 10 years but is susceptible to radiation and electric noise



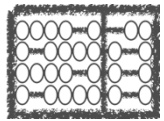
Memory - EEPROM

- Electrically Erasable Programmable ROM
- Programmed and erased electronically
  a. typically by using higher than normal voltage
  b. can program and erase individual words
- Better write ability
  a. can be in-system programmable with built-in circuit to provide higher than normal voltage
     - built-in memory controller commonly used to hide details from memory user
  b. writes very slow due to erasing and programming
     - “busy” pin indicates to processor EEPROM still writing
  c. can be erased and programmed tens of thousands of times
- Similar storage permanence to EPROM (about 10 years)
- Far more convenient than EPROMs, but more expensive



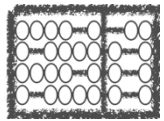
Memory - Flash

- Extension of EEPROM
  - Same floating gate principle
  - Same write ability and storage permanence
- Fast erase
  - Large blocks of memory erased at once, rather than one word at a time
  - Blocks typically several thousand bytes large
- Writes to single words may be slower
  - Entire block must be read, word updated, then entire block written back
- Used with embedded systems storing large data items in nonvolatile memory
  - e.g., digital cameras, TV set-top boxes, cell phones



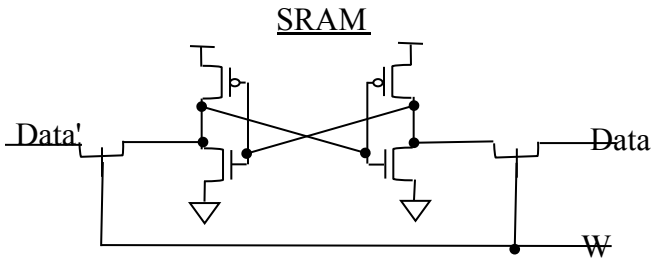
Memory - RAM

- Random-Access Memory
- Typically volatile memory
    - bits are not held without power supply
- Easy/Fast to read and write to by embedded system during execution
- Internal structure more complex than ROM

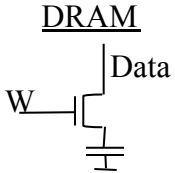


Memory - RAM

- SRAM: Static RAM
  - Memory cell uses flip-flop to store bit
  - Requires 6 transistors
  - Holds data as long as power supplied

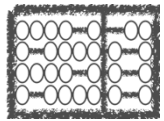


- DRAM: Dynamic RAM
  - Memory cell uses MOS transistor and capacitor to store bit
  - More compact than SRAM
  - “Refresh” required due to capacitor leak
    - word’s cells refreshed when read
  - Typical refresh rate 15.625 microsec.
  - Slower to access than SRAM



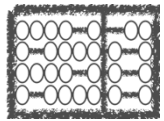
Memory - RAM Variations

- PSRAM: Pseudo-static RAM
  - DRAM with built-in memory refresh controller
  - Popular low-cost high-density alternative to SRAM
- NVRAM: Nonvolatile RAM
  - Holds data after external power removed
  - Battery-backed RAM
    - SRAM with own permanently connected battery
    - writes as fast as reads
    - no limit on number of writes unlike nonvolatile ROM-based memory
  - SRAM with EEPROM or flash
    - stores complete RAM contents on EEPROM or flash before power turned off



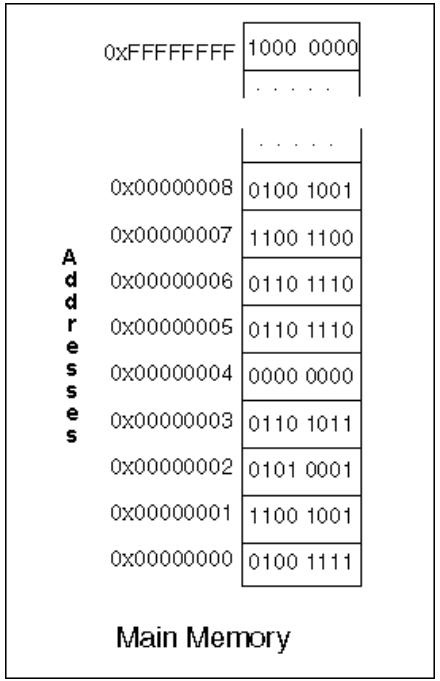
Memory - Common usage

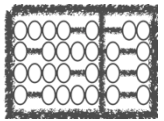
- **DRAM (volatile):** used to build large memory modules (Gigabytes). Off chip.
- **SRAM (volatile):** used to build registers, cache and small on-chip memories.
- **FLASH (non-volatile):** used to build fast hard drives, pen-drives, and on-chip non-volatile memories to store programs.



Memory - Organization

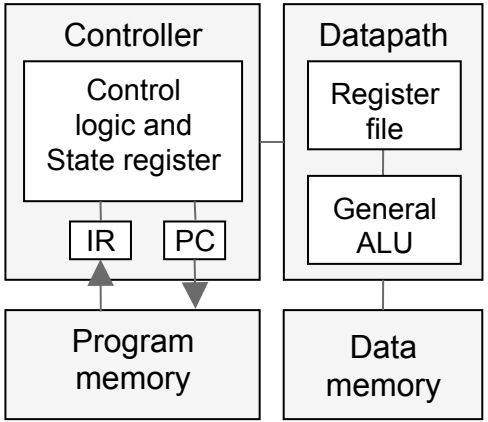
- A memory device contains one or more memory words (basic unit of storage)
  - Typically each word stores 1 byte.
- Each word may be identified by a number (like an array). This number is also known as address.



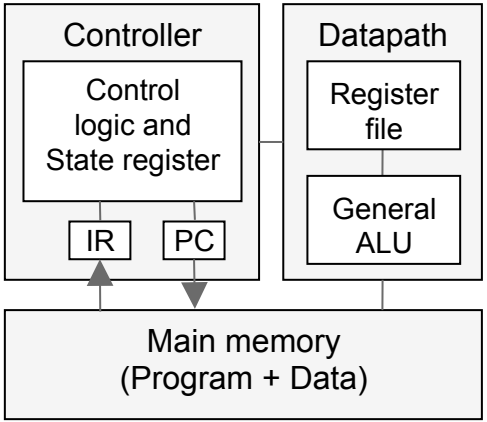


Memory - Organization

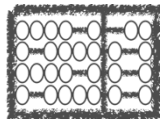
- Memory Architecture



Harvard Architecture

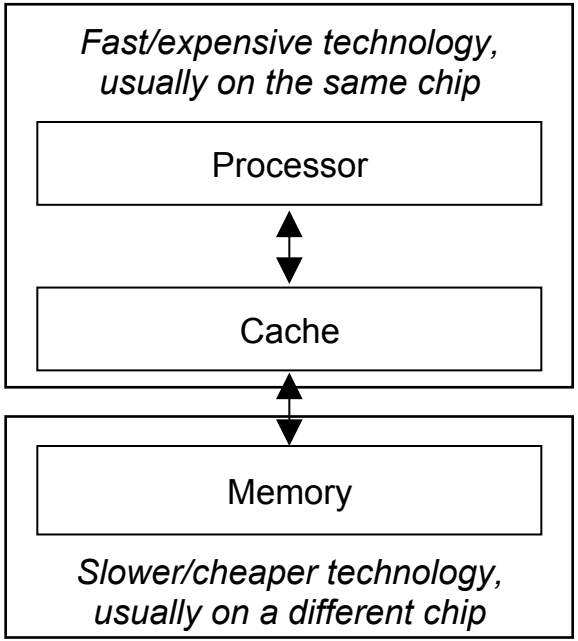


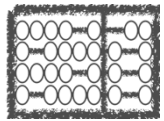
Princeton Architecture



Memory - Hierarchy

- Memory access may be slow
- Cache is small but fast memory close to processor
  - Holds copy of part of memory
  - Hits and misses





Agenda

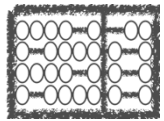
Processing Unit

Memory

Buses

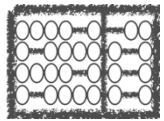
Peripherals

SoCs and MCUs

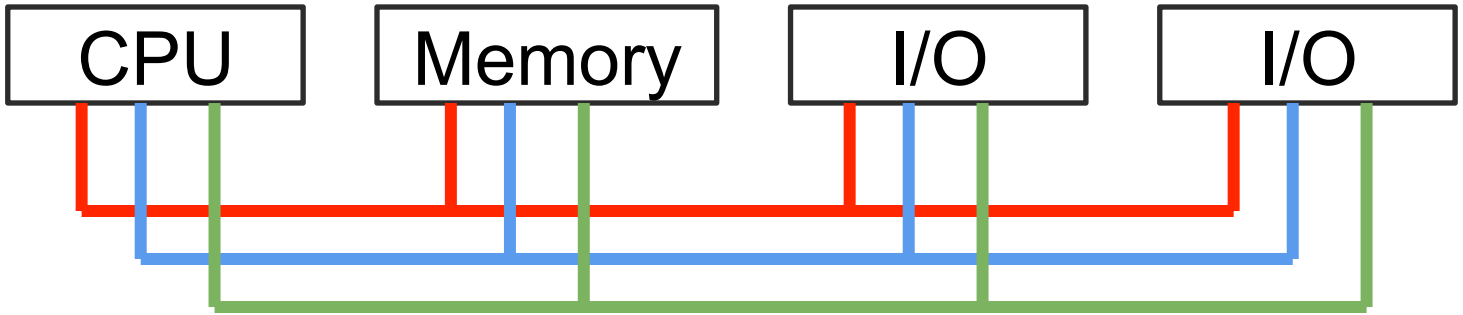


Buses

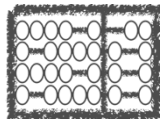
- Communication paths between two or more devices (Processor, memory, …)
- May contain several communications lines which may be classified as:
  - Control lines (or control bus);
  - Address lines (or address bus);
  - Data lines (or address bus);
- Examples:
  - PCI: developed originally by Intel. Currently a public standard.
  - AMBA: developed by ARM



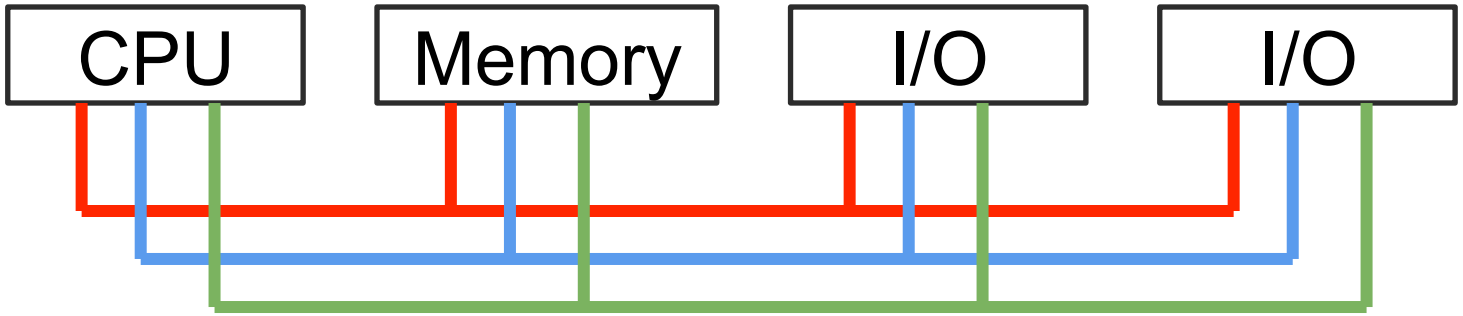
Buses



Data bus
Address bus
Control bus



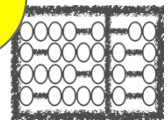
Buses



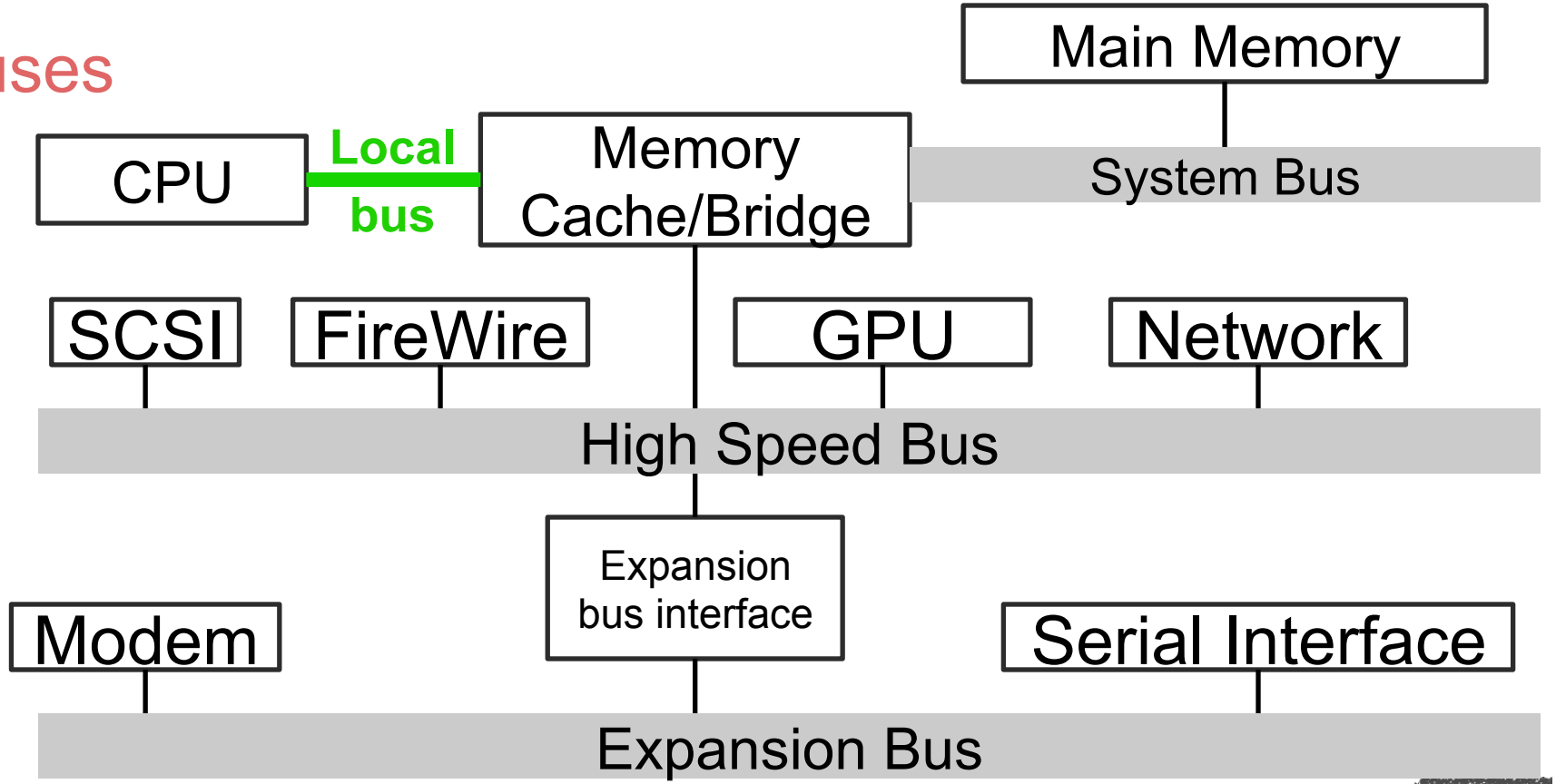
Data bus
Address bus
Control bus

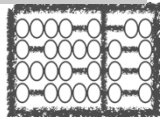
All devices share the same bus. Problems:

- Competition for the bus may cause performance issues!
- they may be required to operate on the same speed!



Buses


Main Memory
CPU
Local bus
Memory Cache/Bridge
System Bus
SCSI
FireWire
GPU
Network
High Speed Bus
Expansion bus interface
Modem
Serial Interface
Expansion Bus



Agenda

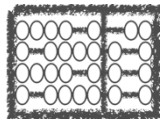
Processing Unit

Memory

Buses

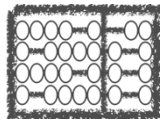
Peripherals

SoCs and MCUs



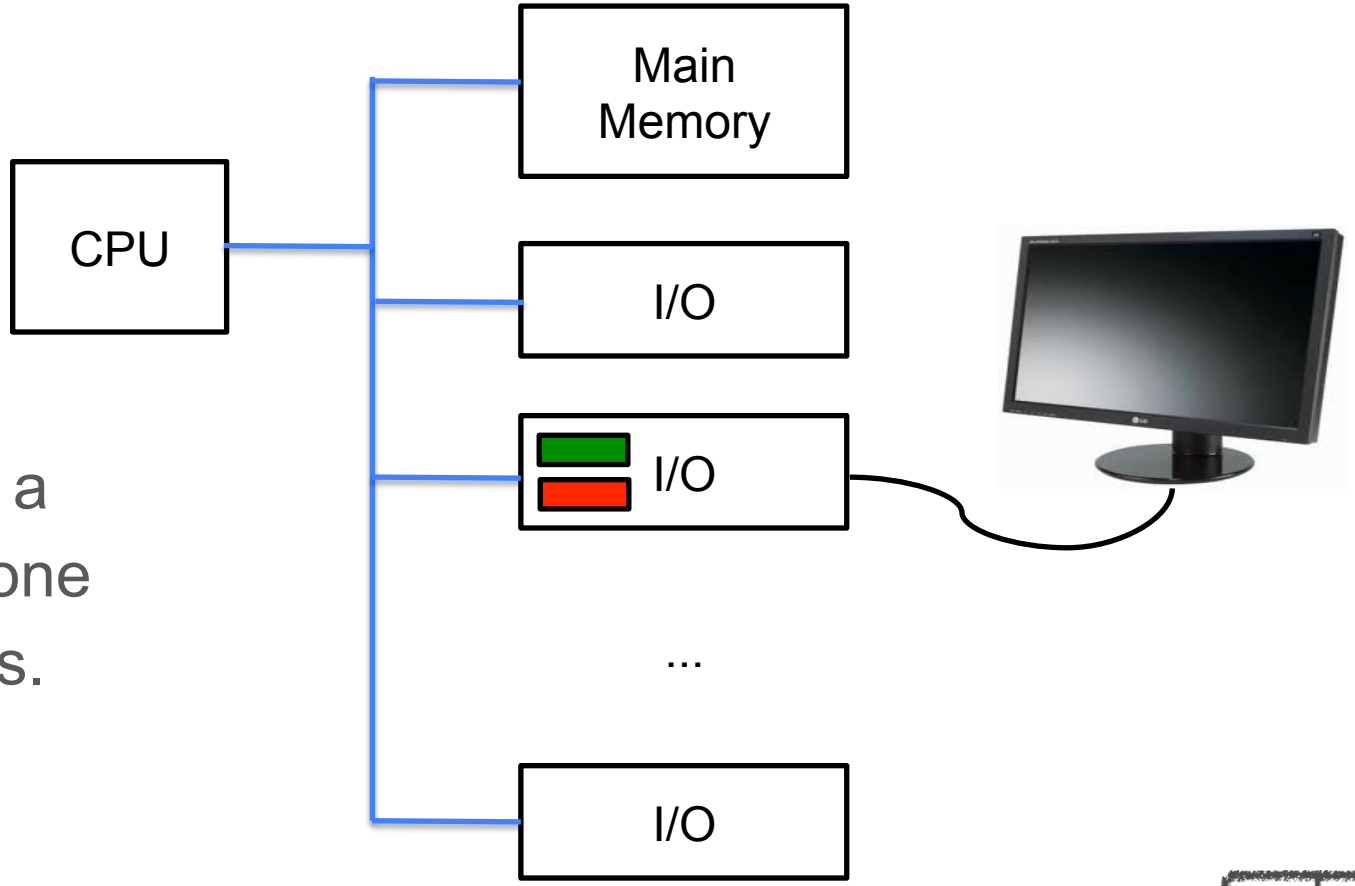
Peripherals

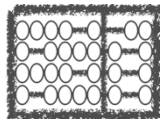
- input devices: receive data from the outer world (mice, keyboards, etc.)
- output devices: send data to the outer world (monitors, printers, etc.)
- input/output device: both functions (network device, USART, etc.)



Peripherals

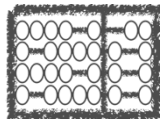
Writing data to a peripheral is done through the bus.





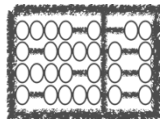
Peripherals

How does the CPU “program” send information to a peripheral?



Peripherals

How does the CPU “program” send information to a peripheral?



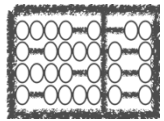
Peripherals

How does the CPU “program” send information to a peripheral?

Two options:

a) Port-mapped I/O: special output instruction.

```
out 0x10, r1
```



Peripherals

How does the CPU “program” send information to a peripheral?

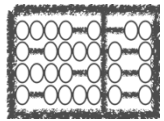
Two options:

a) Port-mapped I/O: special output instruction.

```
out 0x10, r1
```

b) Memory-mapped I/O: regular “store” instruction on a reserved address space

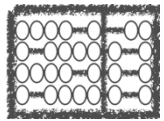
```
str r1, [0x80000]
```



Peripherals

Address Space

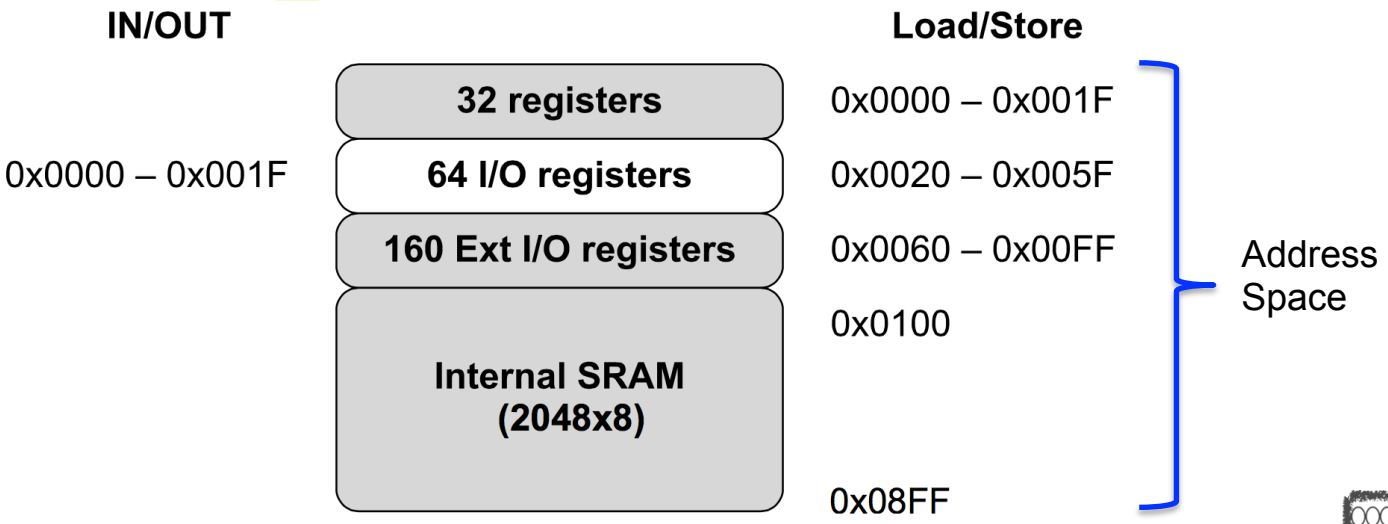
- Set of address used by the CPU to access memory and peripherals.
- A large portion of the address space is assigned to memory cells;
- Nonetheless, part of it may be assigned peripherals (ports).
  - Using LOAD/STORE instructions to access peripherals is known as Memory Mapped I/O.



Peripherals

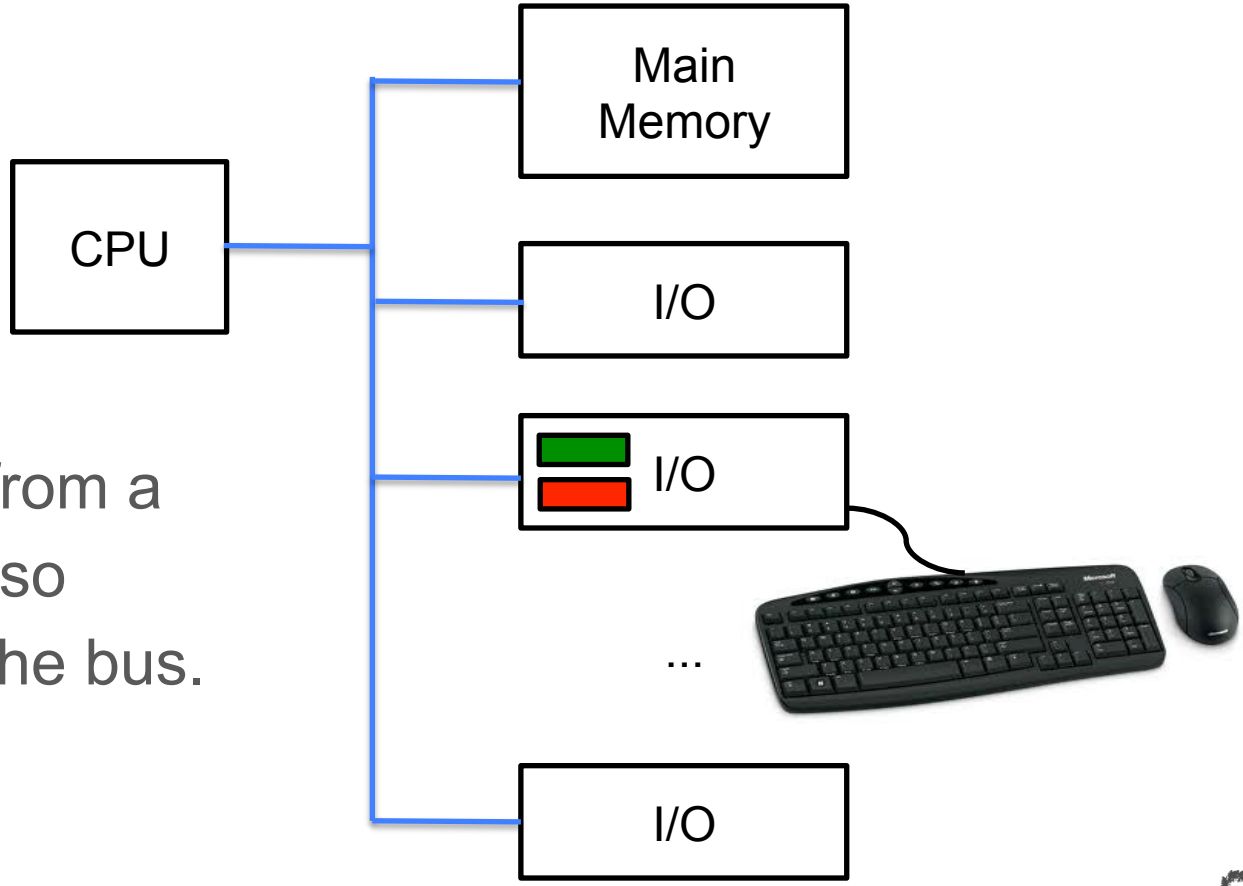
AVR Address Space

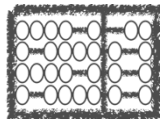
- in/out instructions and load/store instructions



Peripherals

Reading data from a peripheral is also done through the bus.





Peripherals

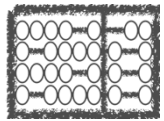
Also two options:

a) Port-mapped I/O: special input instruction.

```
in 0x10, r1
```

b) Memory-mapped I/O: regular “load” instruction from a reserved address space

```
ldr r1, [0x80000]
```



Peripherals

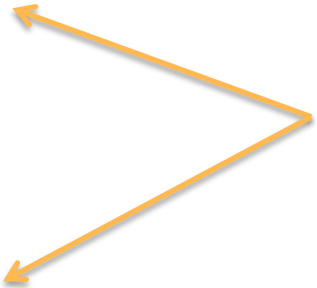
Example: Memory-mapped I/O on ARM

Input:

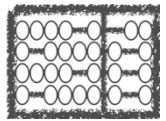
```
ldr r0, =0x53FA0008
ldr r1, [r0]
```

Output:

```
ldr r0, =0x53FA0000
str r1, [r0]
```



These addresses are mapped onto peripherals



Agenda

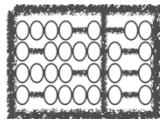
Processing Unit

Memory

Buses

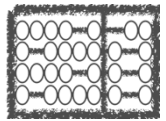
Peripherals

SoCs and MCUs



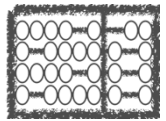
SoC: System-on-a-Chip

- SoC definition suggests that a single chip contains all the components of the system (e.g. Processor, memory, buses, peripheral, ...)
  - This is not always the case.
- Some SoCs contain only part of the system.



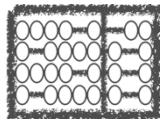
Microcontroller

- a.k.a. MCU (Microcontroller Unit), a microcontroller a system integrated on a single chip that contains the processor core, memory, and programmable input/output peripherals.
- Designed to support control applications.
  - Reading sensors, setting actuators (Several resources for I/O)
  - Usually little memory for code and data

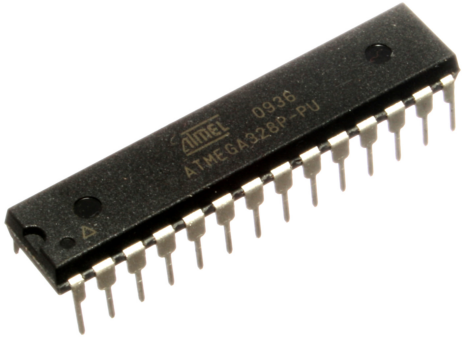


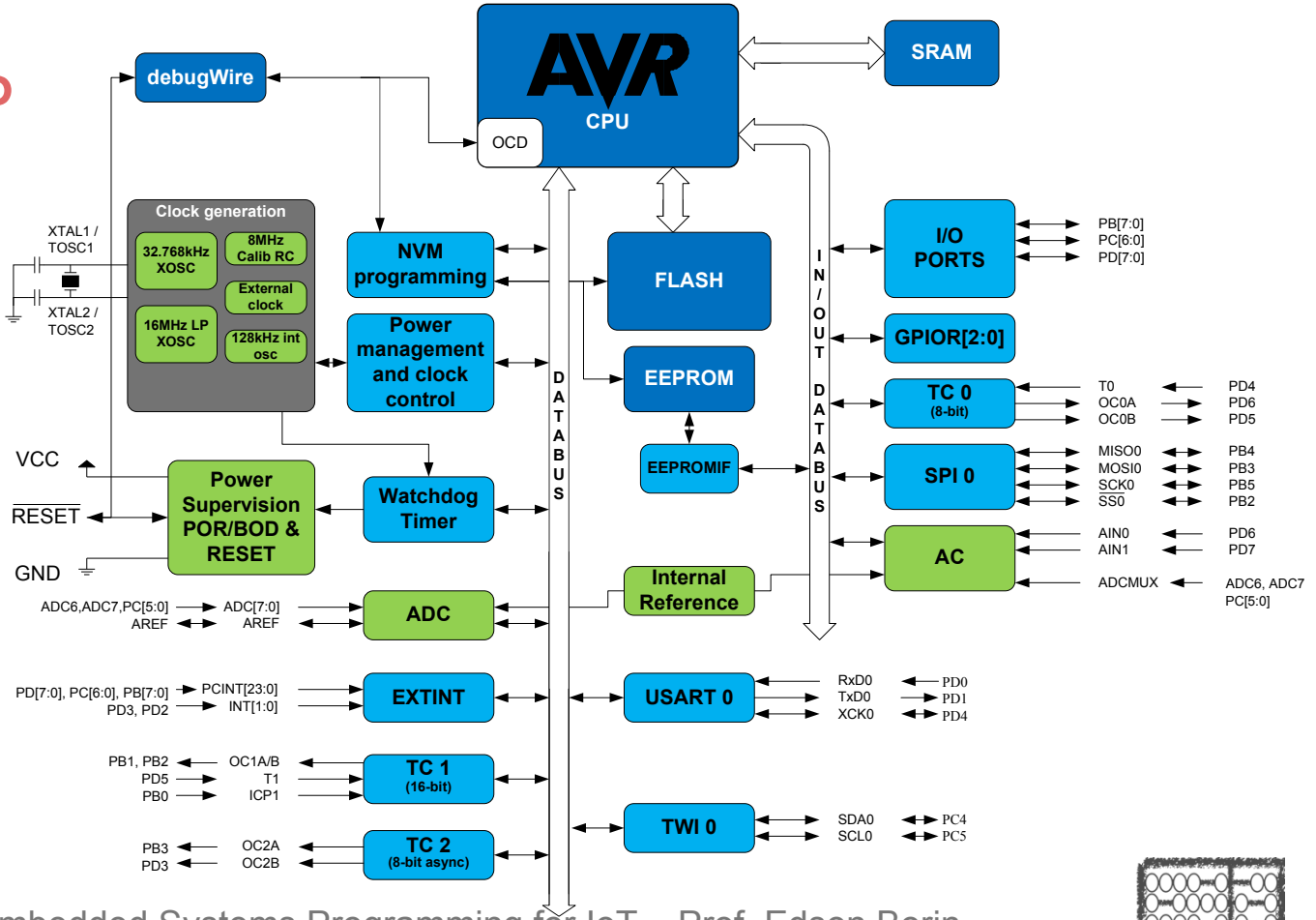
Microcontroller

- Common features:
  - On-chip program and data memory
  - On-chip peripherals
    - Timers, analog-digital converters, serial communication, etc.
  - Direct programmer access to many of the chip's pins
  - Specialized instructions for bit-manipulation and other low-level operations

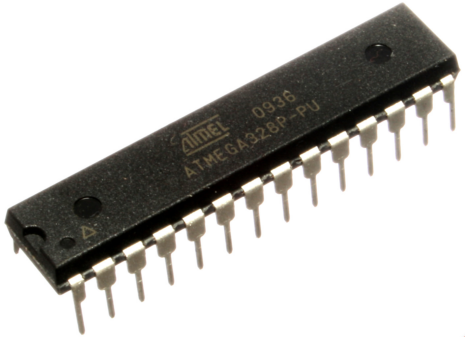


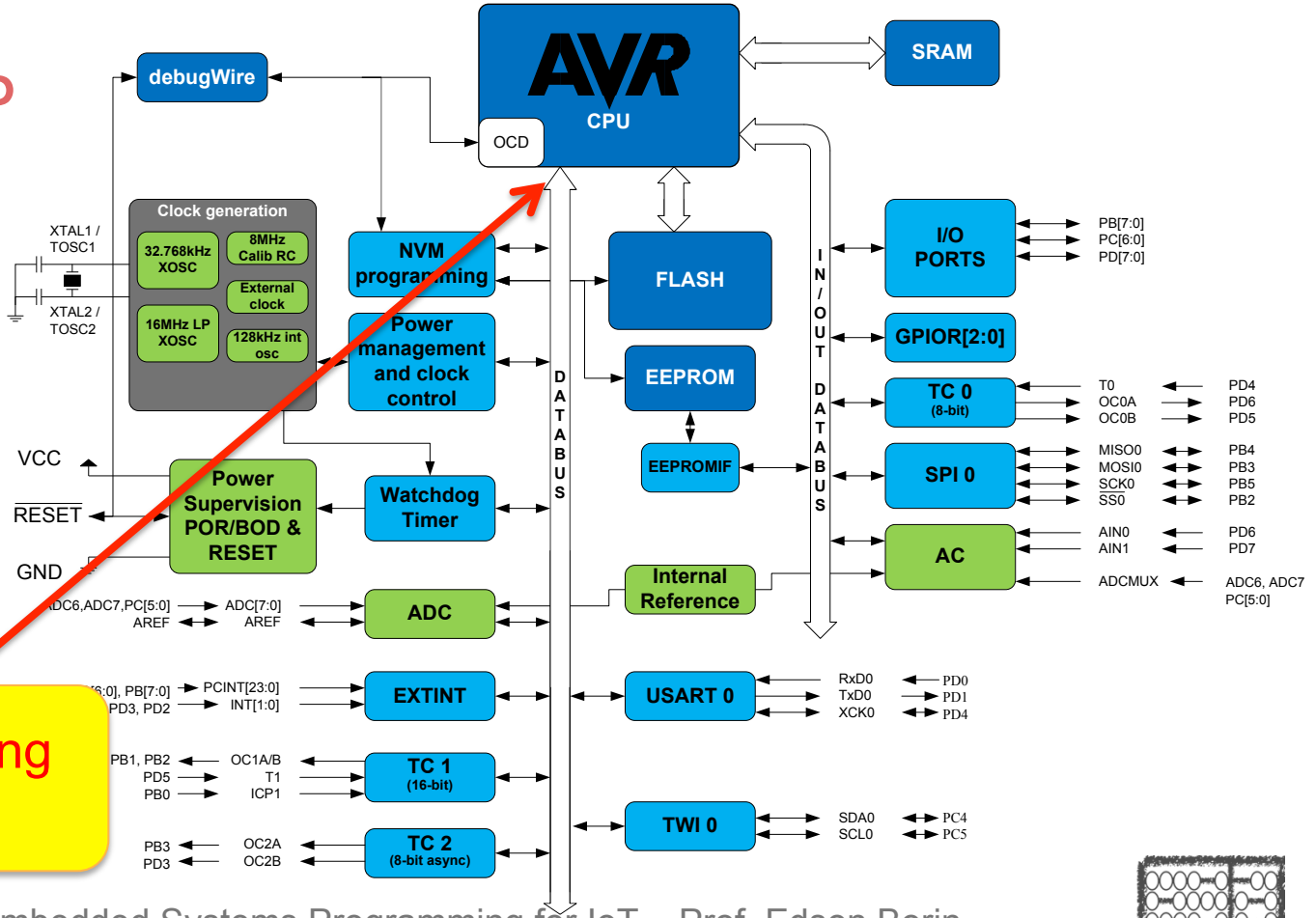
ATMega328/P


ATMEGA328P-PU


AVR
CPU
OCD
debugWire
SRAM
Clock generation
32.768kHz XOSC
8MHz Calib RC
External clock
16MHz LP XOSC
128kHz int osc
XTAL1 / TOSC1
XTAL2 / TOSC2
NVM programming
Power management and clock control
FLASH
EEPROM
EEPROMIF
I/O PORTS
PB[7:0]
PC[6:0]
PD[7:0]
GPIOR[2:0]
TC 0 (8-bit)
T0
OC0A
OC0B
PD4
PD6
PD5
SPI 0
MISO0
MOSI0
SCK0
SS0
PB4
PB3
PB5
PB2
AC
AIN0
AIN1
ADCMUX
PD6
PD7
ADC6, ADC7
PC[5:0]
I/O DATABUS
DATABUS
VCC
RESET
GND
Power Supervision POR/BOD & RESET
Watchdog Timer
Internal Reference
ADC6,ADC7,PC[5:0]
AREF
ADC[7:0]
AREF
ADC
PD[7:0], PC[6:0], PB[7:0]
PD3, PD2
PCINT[23:0]
INT[1:0]
EXTINT
USART 0
RxD0
TxD0
XCK0
PD0
PD1
PD4
PB1, PB2
PD5
PB0
OC1A/B
T1
ICP1
TC 1 (16-bit)
TWI 0
SDA0
SCL0
PC4
PC5
PB3
PD3
OC2A
OC2B
TC 2 (8-bit async)

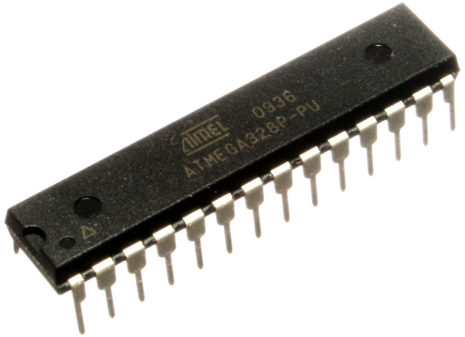
ATMega328/P

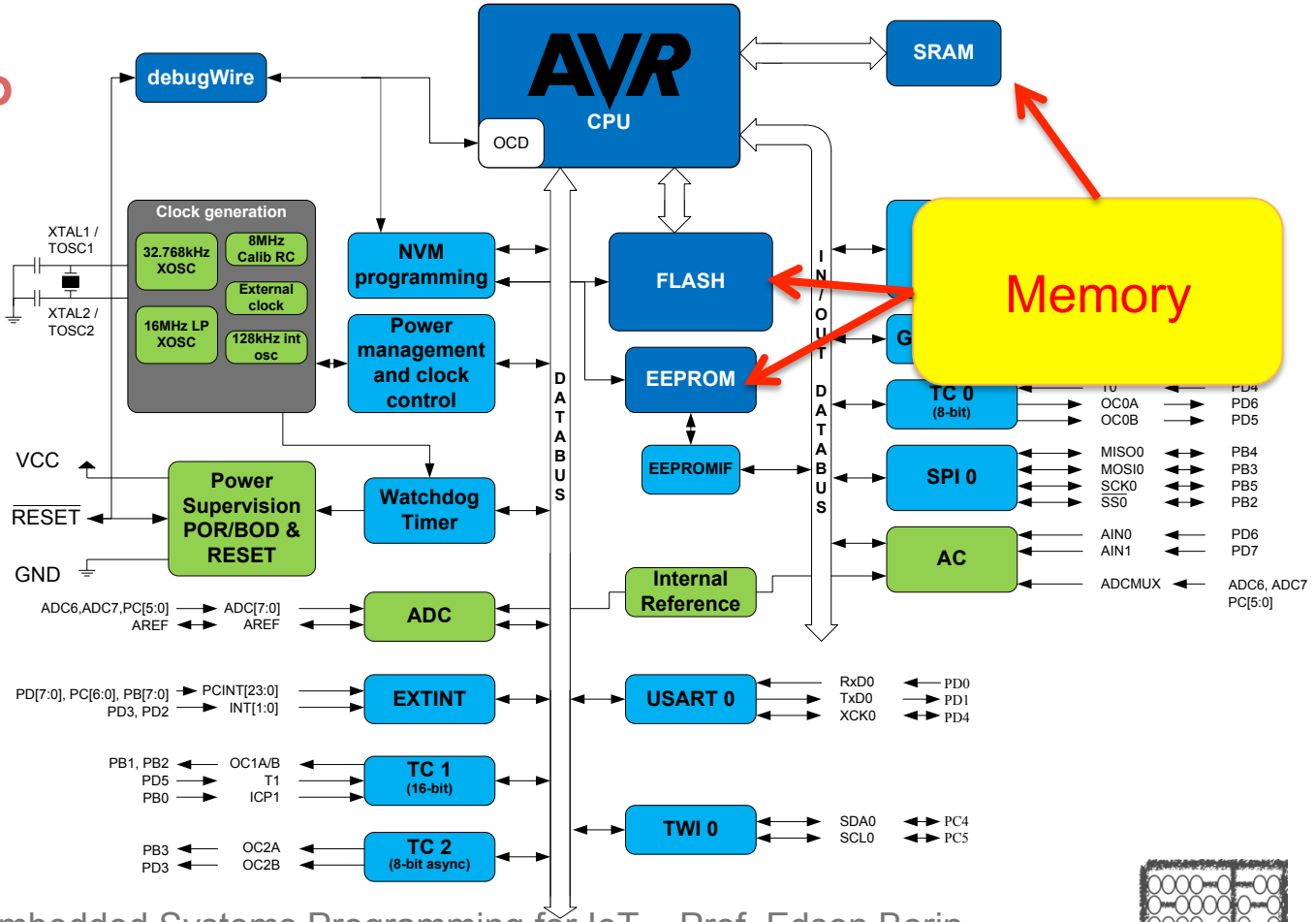




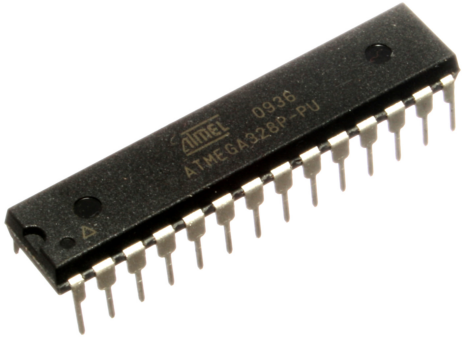
Processing Unit

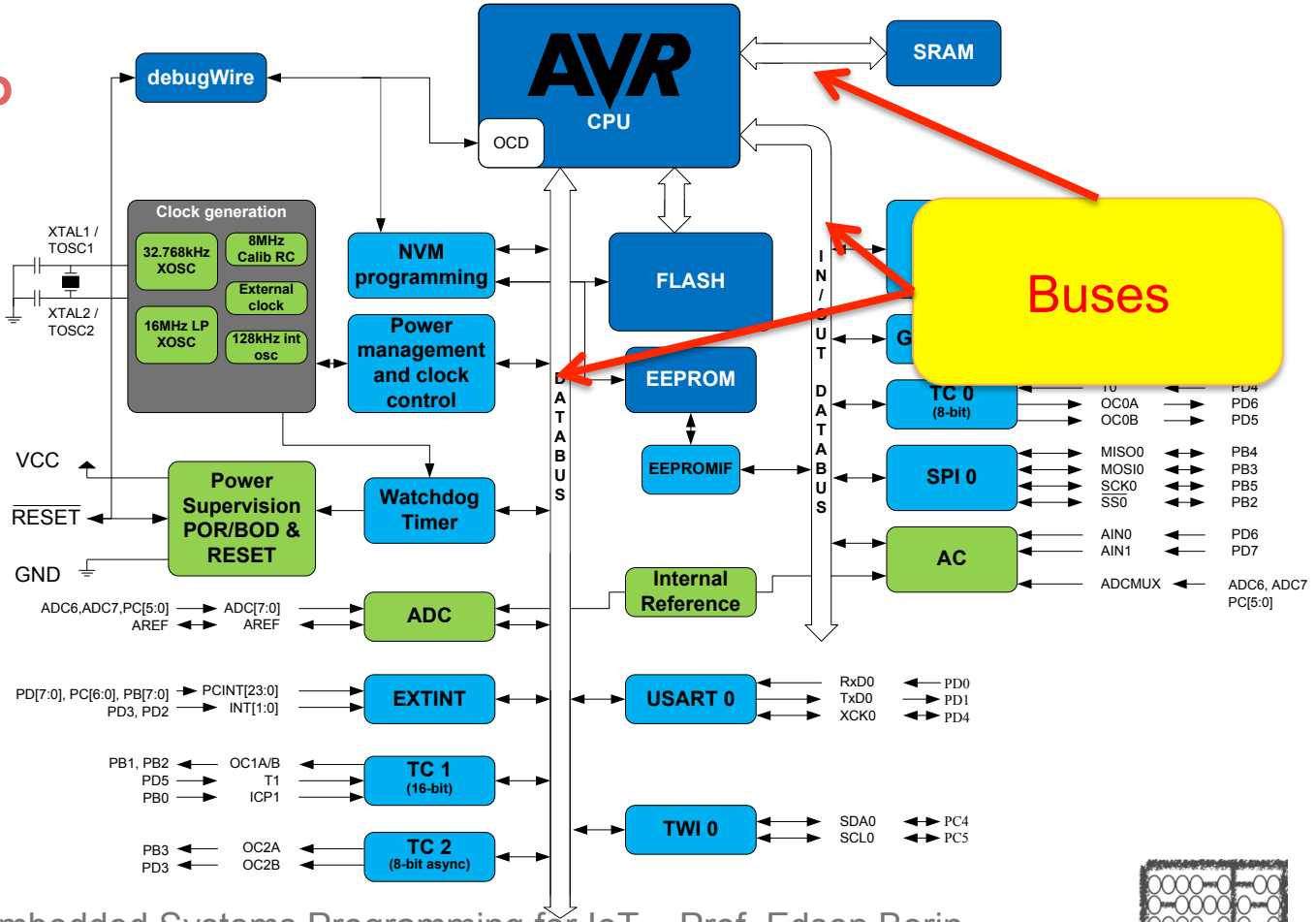
ATMega328/P


ATMEGA328P-PU

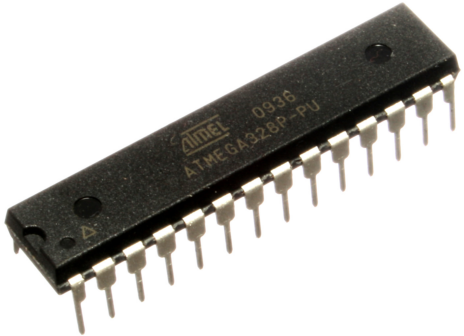

AVR
CPU
OCD
debugWire
SRAM
Memory
Clock generation
XTAL1 / TOSC1
XTAL2 / TOSC2
32.768kHz XOSC
8MHz Calib RC
External clock
16MHz LP XOSC
128kHz int osc
NVM programming
Power management and clock control
FLASH
EEPROM
EEPROMIF
DATABUS
I/O DATABUS
TC 0 (8-bit)
SPI 0
AC
VCC
RESET
GND
Power Supervision POR/BOD & RESET
Watchdog Timer
Internal Reference
ADC6,ADC7,PC[5:0]
AREF
ADC[7:0]
ADC
PD[7:0], PC[6:0], PB[7:0]
PD3, PD2
PCINT[23:0]
INT[1:0]
EXTINT
USART 0
RxD0
TxD0
XCK0
PD0
PD1
PD4
PB1, PB2
PD5
PB0
OC1A/B
T1
ICP1
TC 1 (16-bit)
TWI 0
SDA0
SCL0
PC4
PC5
PB3
PD3
OC2A
OC2B
TC 2 (8-bit async)
OC0A
OC0B
PD6
PD5
MISO0
MOSI0
SCK0
SS0
PB4
PB3
PB5
PB2
AIN0
AIN1
PD6
PD7
ADCMUX
ADC6, ADC7
PC[5:0]

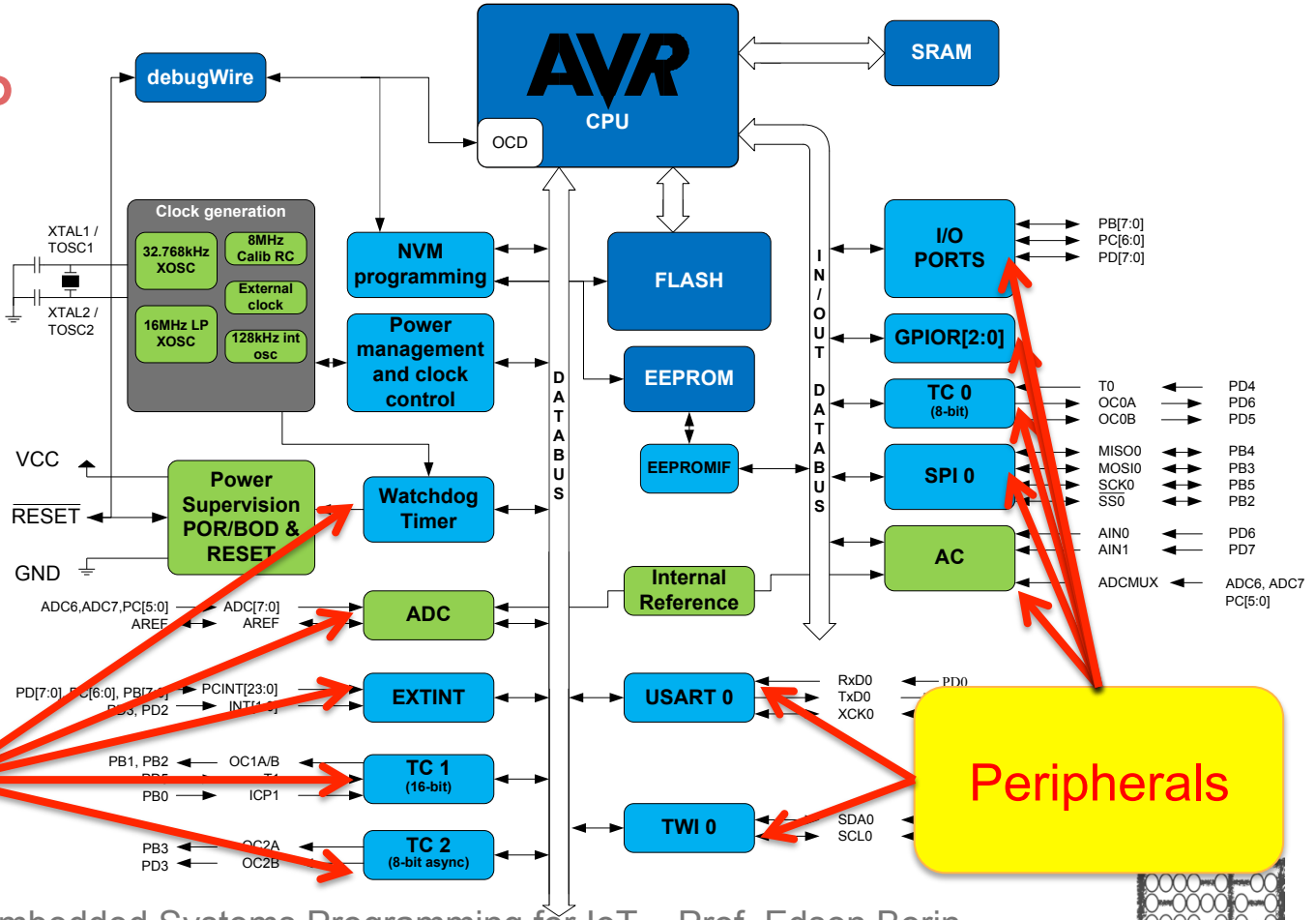
ATMega328/P


ATMEGA328P-PU


AVR
CPU
OCD
debugWire
SRAM
Clock generation
32.768kHz XOSC
8MHz Calib RC
External clock
16MHz LP XOSC
128kHz int osc
XTAL1 / TOSC1
XTAL2 / TOSC2
NVM programming
Power management and clock control
FLASH
EEPROM
EEPROMIF
Buses
DATABUS
I/O DATABUS
VCC
RESET
GND
Power Supervision POR/BOD & RESET
Watchdog Timer
Internal Reference
ADC6,ADC7,PC[5:0]
AREF
ADC[7:0]
AREF
ADC
PD[7:0], PC[6:0], PB[7:0]
PD3, PD2
PCINT[23:0]
INT[1:0]
EXTINT
PB1, PB2
PD5
PB0
OC1A/B
T1
ICP1
TC 1 (16-bit)
PB3
PD3
OC2A
OC2B
TC 2 (8-bit async)
USART 0
RxD0
TxD0
XCK0
PD0
PD1
PD4
TWI 0
SDA0
SCL0
PC4
PC5
TC 0 (8-bit)
OC0A
OC0B
PD6
PD5
SPI 0
MISO0
MOSI0
SCK0
SS0
PB4
PB3
PB5
PB2
AC
AIN0
AIN1
ADCMUX
PD6
PD7
ADC6, ADC7
PC[5:0]

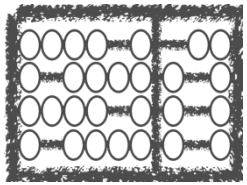
ATMega328/P


ATMEGA328P-PU


AVR
CPU
OCD
SRAM
debugWire
Clock generation
32.768kHz XOSC
8MHz Calib RC
External clock
16MHz LP XOSC
128kHz int osc
XTAL1 / TOSC1
XTAL2 / TOSC2
NVM programming
Power management and clock control
FLASH
EEPROM
EEPROMIF
DATABUS
I/O PORTS
PB[7:0]
PC[6:0]
PD[7:0]
GPIOR[2:0]
TC 0 (8-bit)
T0
OC0A
OC0B
PD4
PD6
PD5
SPI 0
MISO0
MOSI0
SCK0
SS0
PB4
PB3
PB5
PB2
AC
AIN0
AIN1
PD6
PD7
ADCMUX
ADC6, ADC7
PC[5:0]
VCC
RESET
GND
Power Supervision POR/BOD & RESET
Watchdog Timer
Internal Reference
ADC6,ADC7,PC[5:0]
AREF
ADC[7:0]
ADC
PCINT[23:0]
EXTINT
USART 0
RxD0
TxD0
XCK0
PD0
PB1, PB2
OC1A/B
PB0
ICP1
TC 1 (16-bit)
TWI 0
SDA0
SCL0
PB3
PD3
OC2A
OC2B
TC 2 (8-bit async)
Peripherals
Peripherals

INF-741: Embedded systems programming for IoT

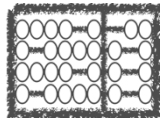
Prof. Edson Borin



Institute of Computing

University of Campinas

Morse Code Sketch



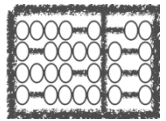
Morse code sketch

```
const char* msg[] = “HELLO IOT WORLD”;
void setup()
{
  pinMode(LED_PIN, OUTPUT);
  char* p = msg;

  while (*p != 0) {
    delay_between_letters();
    char* code = get_code(*p);
    display(code);
    p++;
  }
}
```

```
void loop(){}

void delay_between_letters()
{
  delay(DELAY_BETWEEN_LETTERS);
}
```

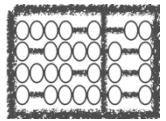


Morse code sketch

```
const char* msg[] = “HELLO IOT WORLD”;
void setup()
{
  pinMode(LED_PIN, OUTPUT);
  char* p = msg;

  while (*p != 0) {
    delay_between_letters();
    char* code = get_code(*p);
    display(code);
    p++;
  }
}
```

```
const char* get_code(const char c)
{
  switch (c){
    case 'A':
      return ".-";
      break;
    case 'B':
      return "-...";
      break;
    ....
}
```

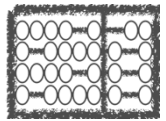


Morse code sketch

```
void display (const char* code) {
  for (char* p = code; *p; p++) {
    switch(*p){
    case '.':
      dot();
      break;
    case '-':
      dash();
      break;
    case ' ':
      delay_between_words();
      break;
    default:
      break;
    }
    delay_between_dots_and_dashes();
  }
```

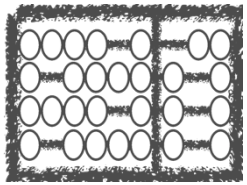
```
void dot ()
{
  digitalWrite(LED_PIN, HIGH);
  delay(DOT_TIME);
  digitalWrite(LED_PIN, LOW);
}

void dash (){
  digitalWrite(LED_PIN, HIGH);
  delay(DASH_TIME);
  digitalWrite(LED_PIN, LOW);
}
```



INF-741: Embedded systems programming for IoT

Prof. Edson Borin



Institute of Computing

University of Campinas