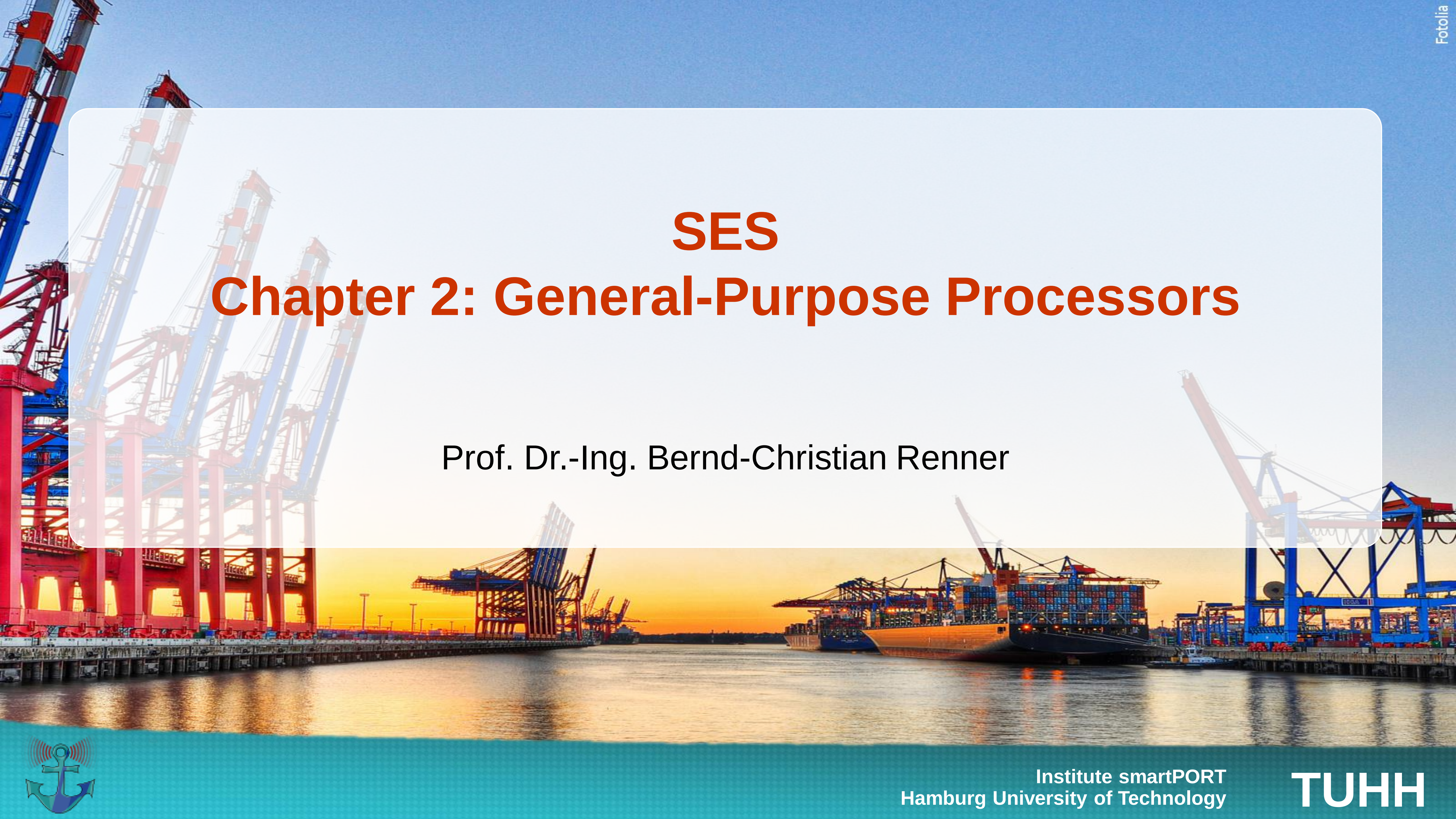


SES
Chapter 2: General-Purpose Processors

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH

Contents

1

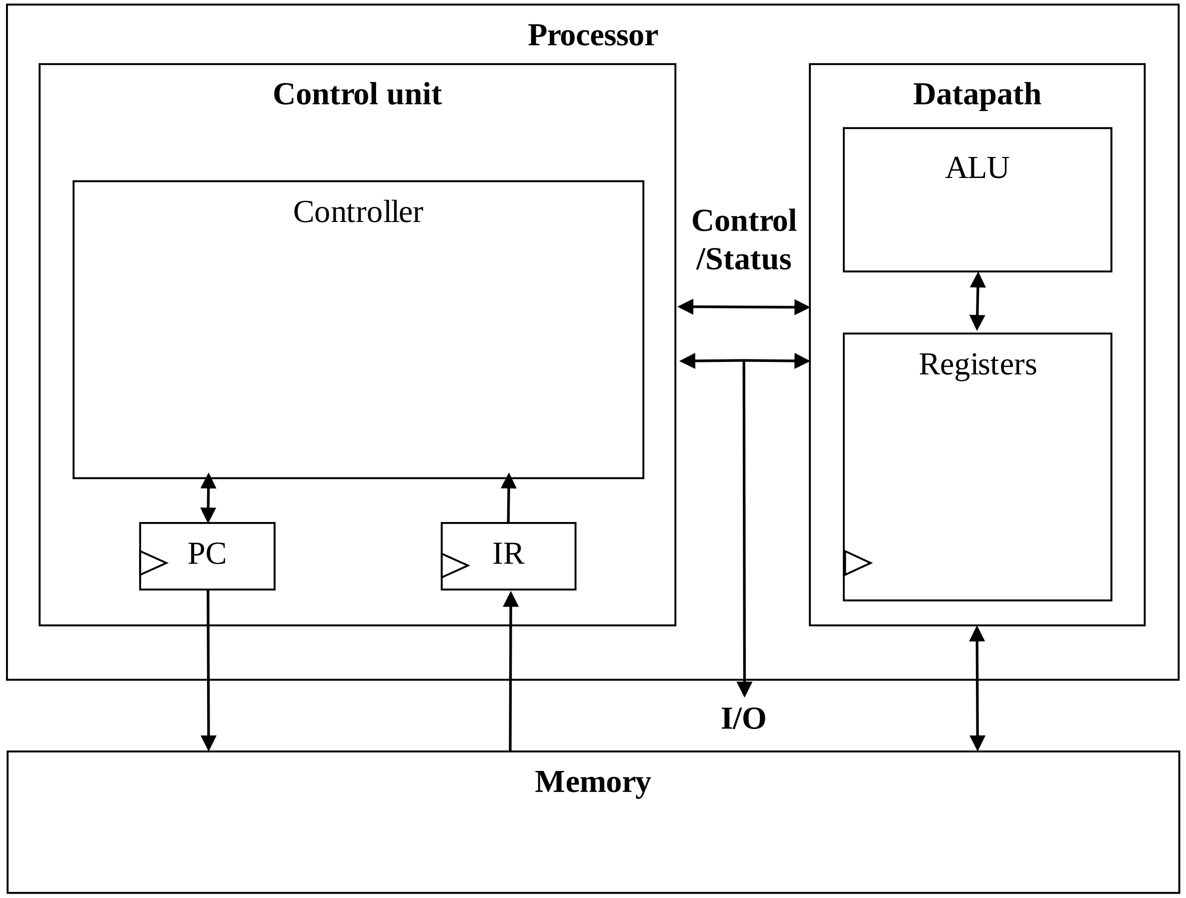
Basic Architecture

Introduction

- General-Purpose Processor
  - Processor designed for a variety of computation tasks (a.k.a. CPU or microprocessor)
  - Low unit cost, in part because manufacturer spreads NRE over large numbers of units
  - Carefully designed since higher NRE is acceptable
    - Can yield good performance, size and power
  - Advantage for embedded systems: Low NRE cost, short time-to-market/prototype, high flexibility
    - User just writes software; no processor design

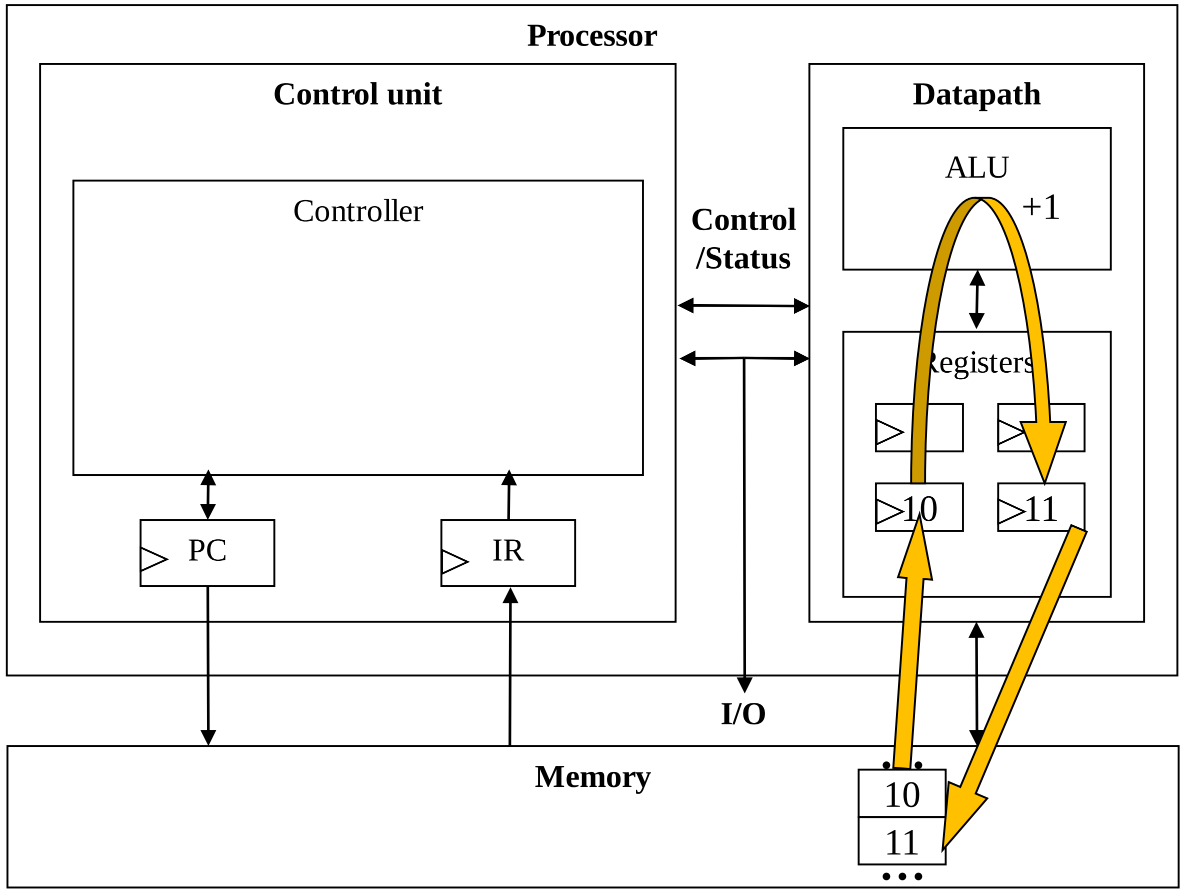
Basic Architecture

- Control unit and datapath
  - Compare with single-purpose processor
- Key differences
  - Datapath is general
  - Control unit doesn't store algorithm
  - algorithm is "programmed" into memory



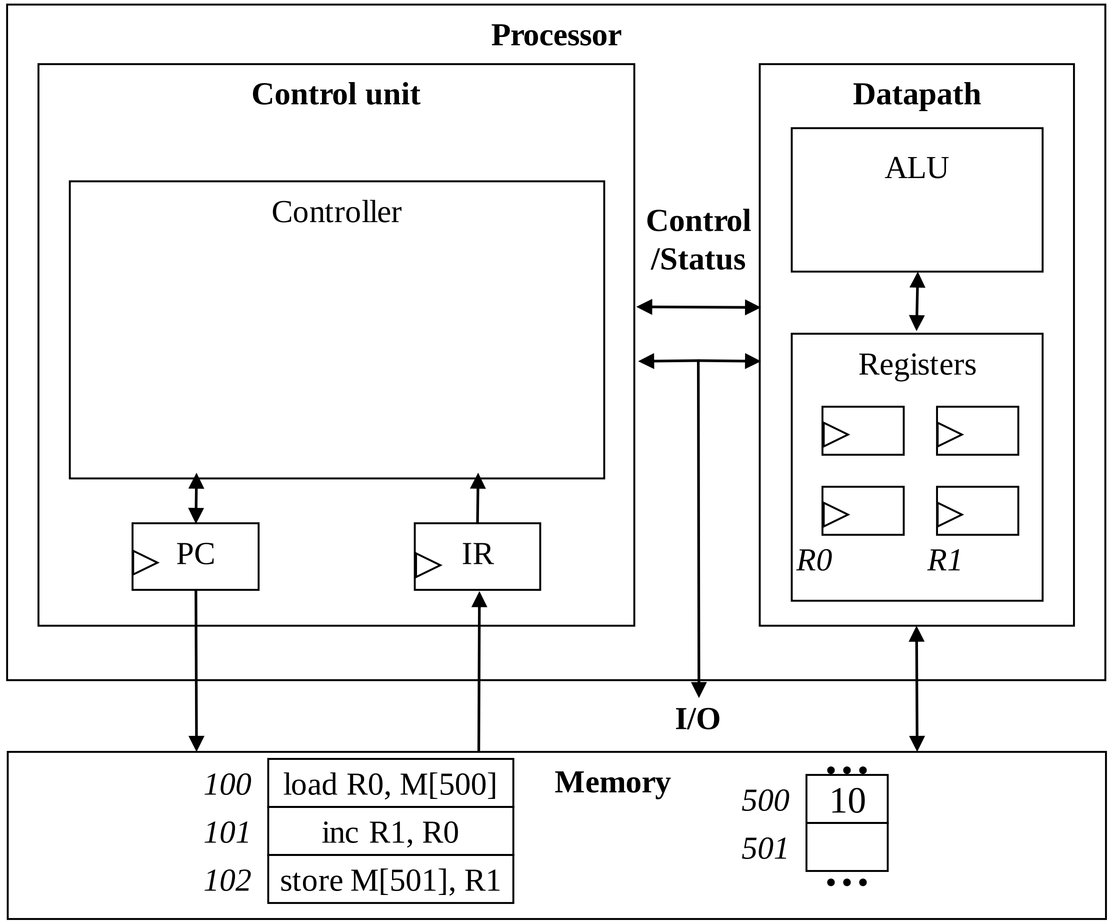
Datapath Operations

- Load
  - Read memory location into register
- ALU operation
  - Input certain registers through ALU, store back in register
- Store
  - Write register to memory location



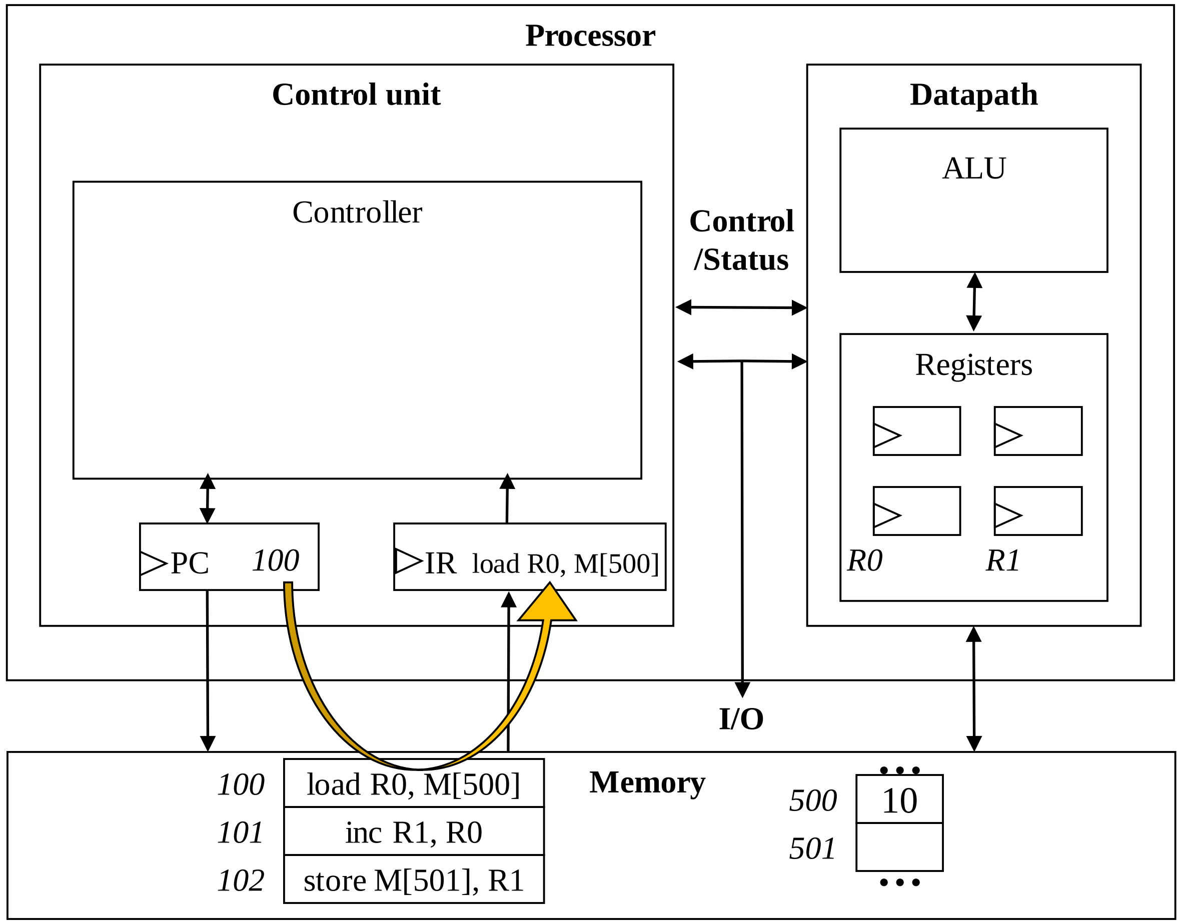
Control Unit

- Control unit: configures datapath operations
  - Sequence of desired operations ("instructions") stored in memory – "program"
- Instruction cycle – broken into several sub-operations, each one clock cycle, e.g.:
  - 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 ALU
  - Store results: Write data from register to memory



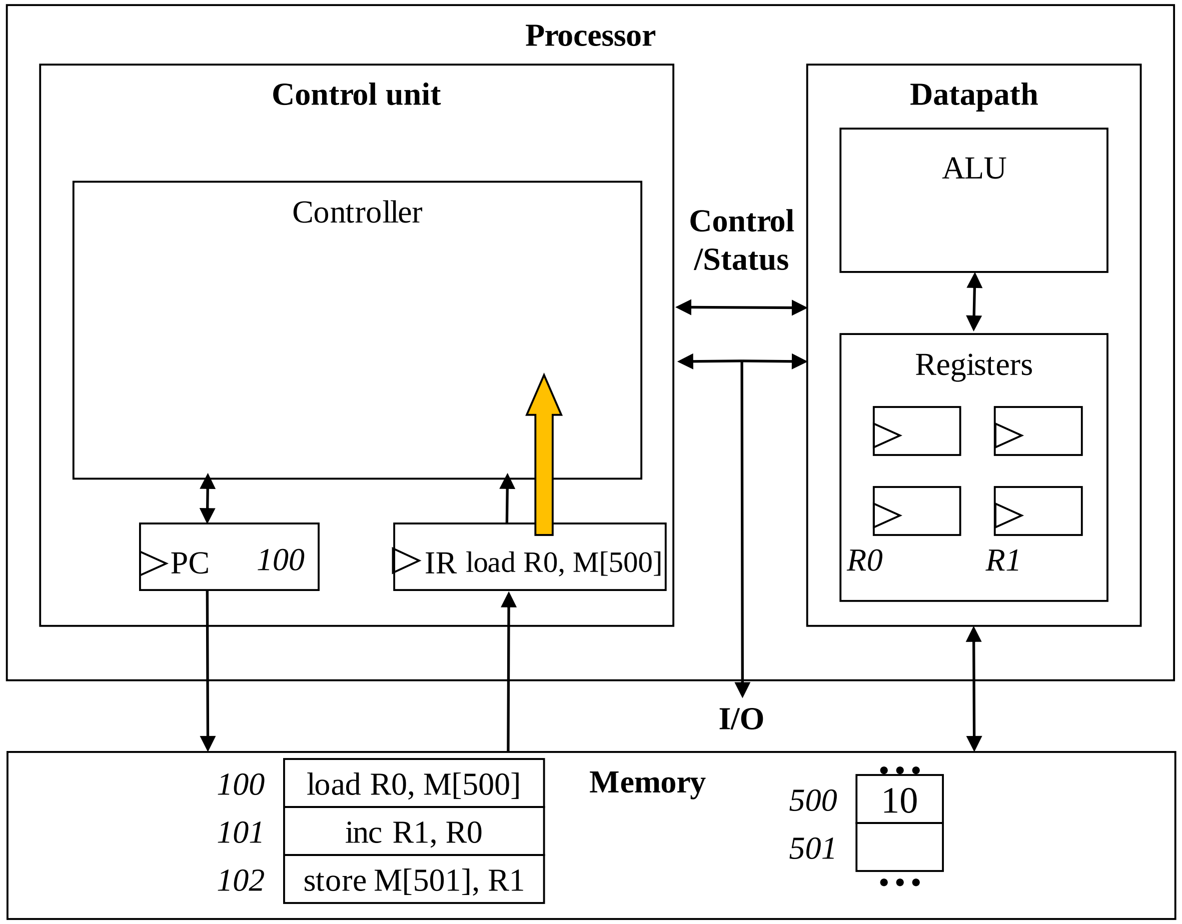
Control Unit Sub-Operations

- Fetch
  - Get next instruction into IR
  - PC: program counter, always points to next instruction
  - IR: holds the fetched instruction



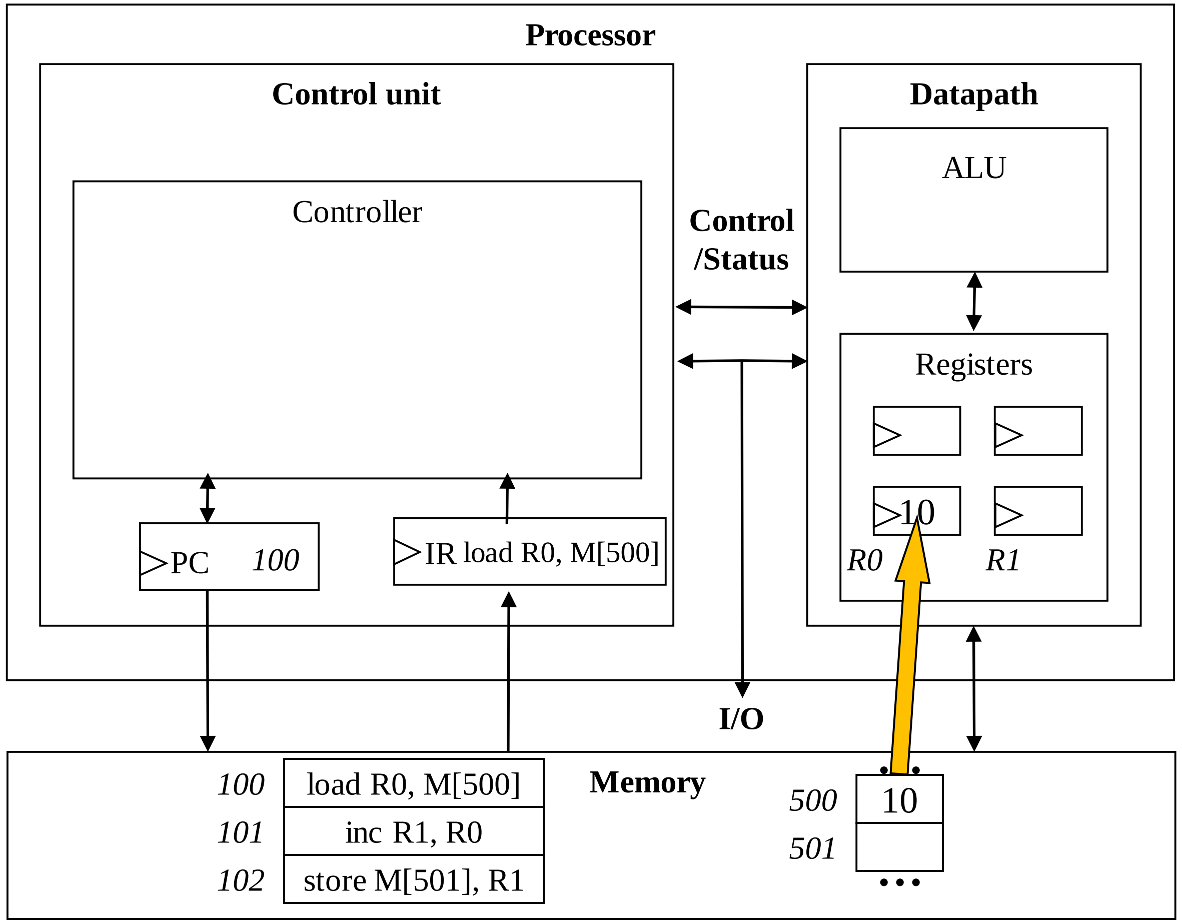
Control Unit Sub-Operations

- Decode
  - Determine what the instruction means



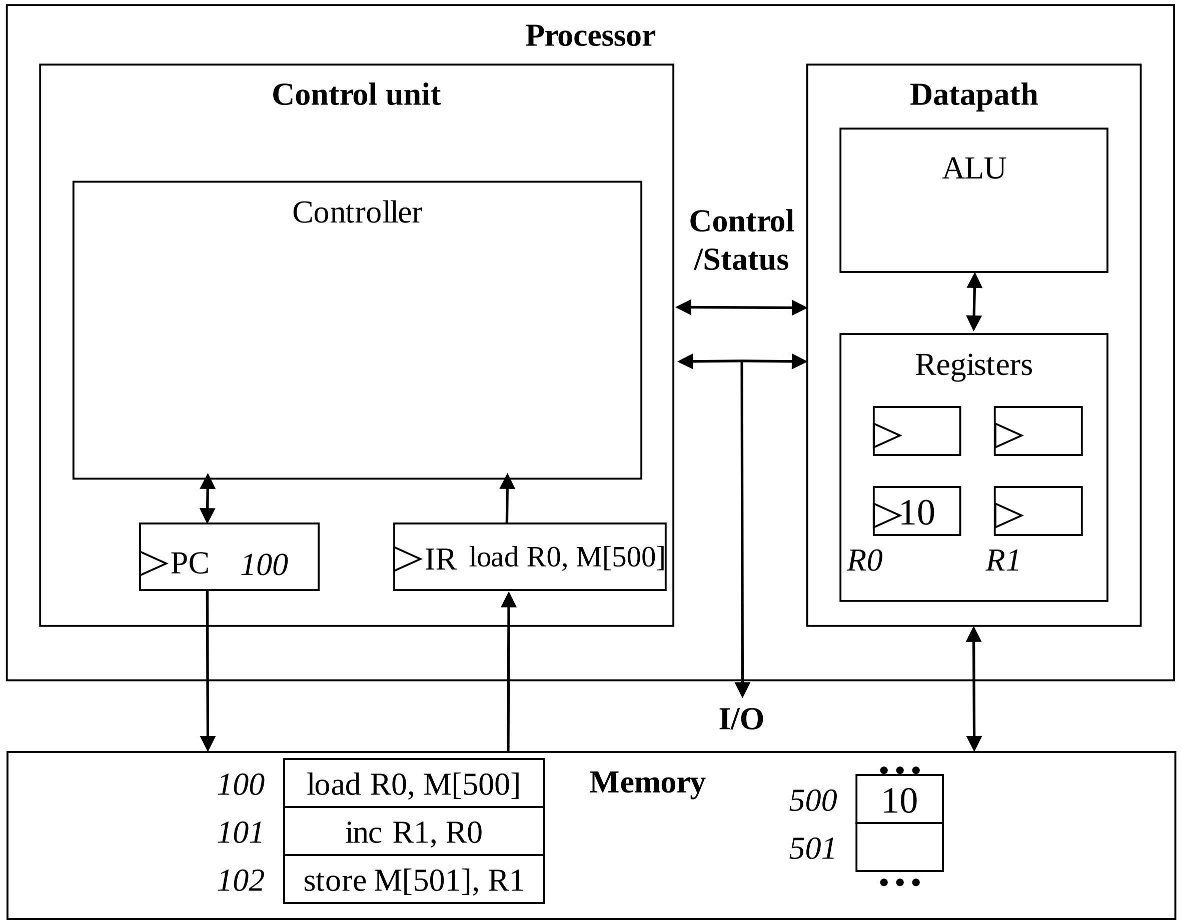
Control Unit Sub-Operations

- Fetch operands
  - Move data from memory to datapath register



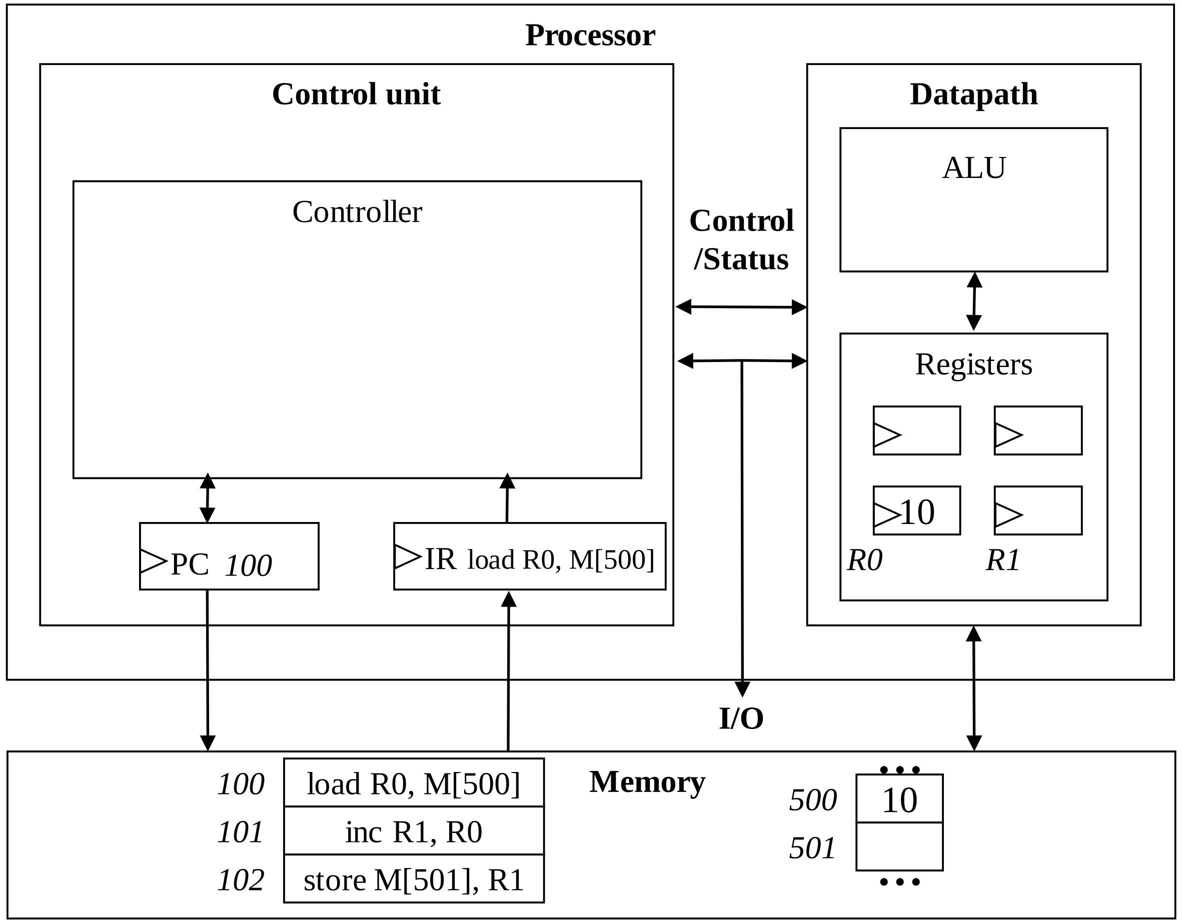
Control Unit Sub-Operations

- Execute
  - Move data through the ALU
  - This particular instruction does nothing during this sub-operation

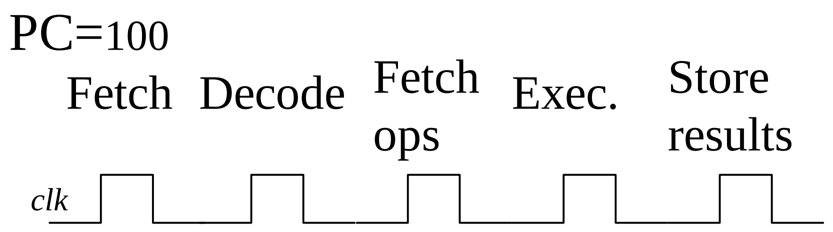


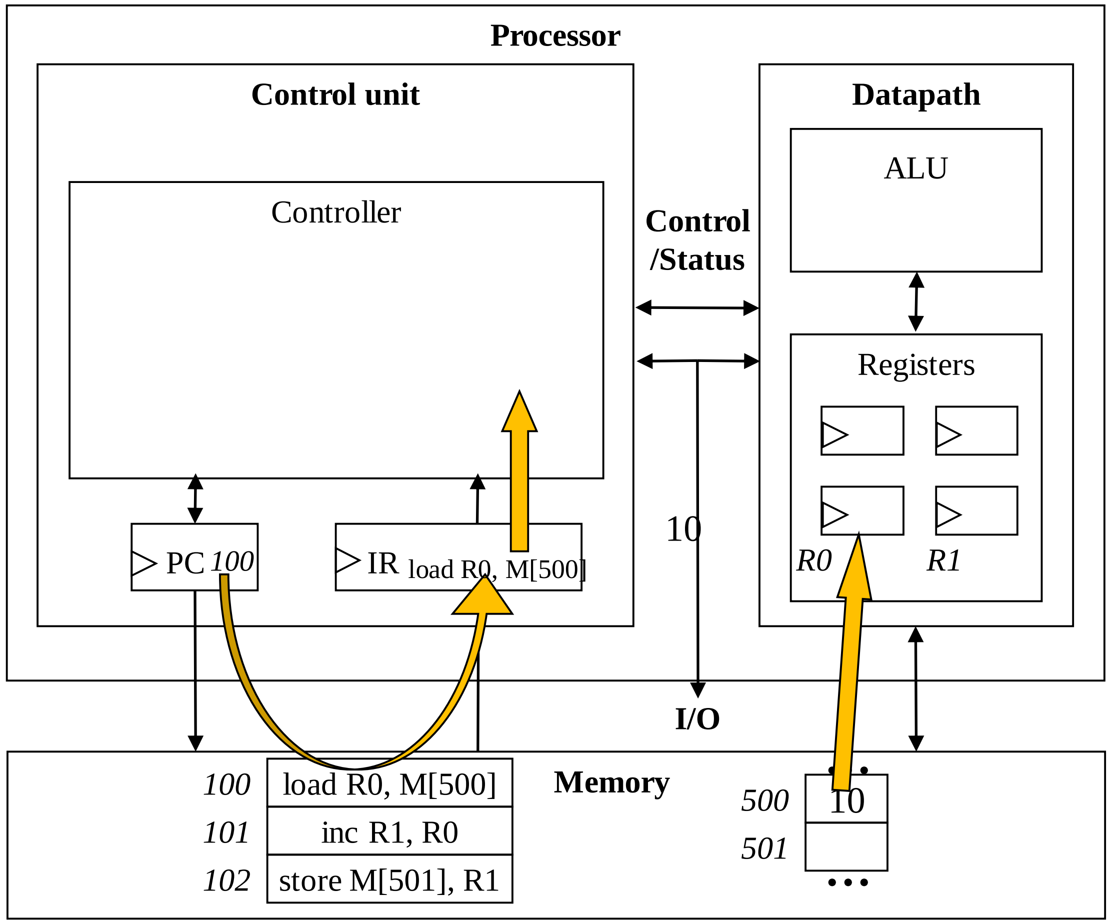
Control Unit Sub-Operations

- Store results
  - Write data from register to memory
  - This particular instruction does nothing during this sub-operation

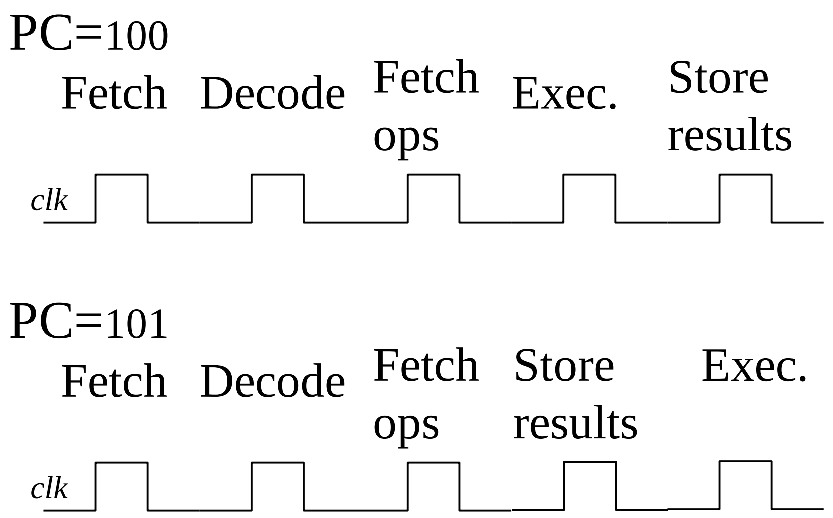


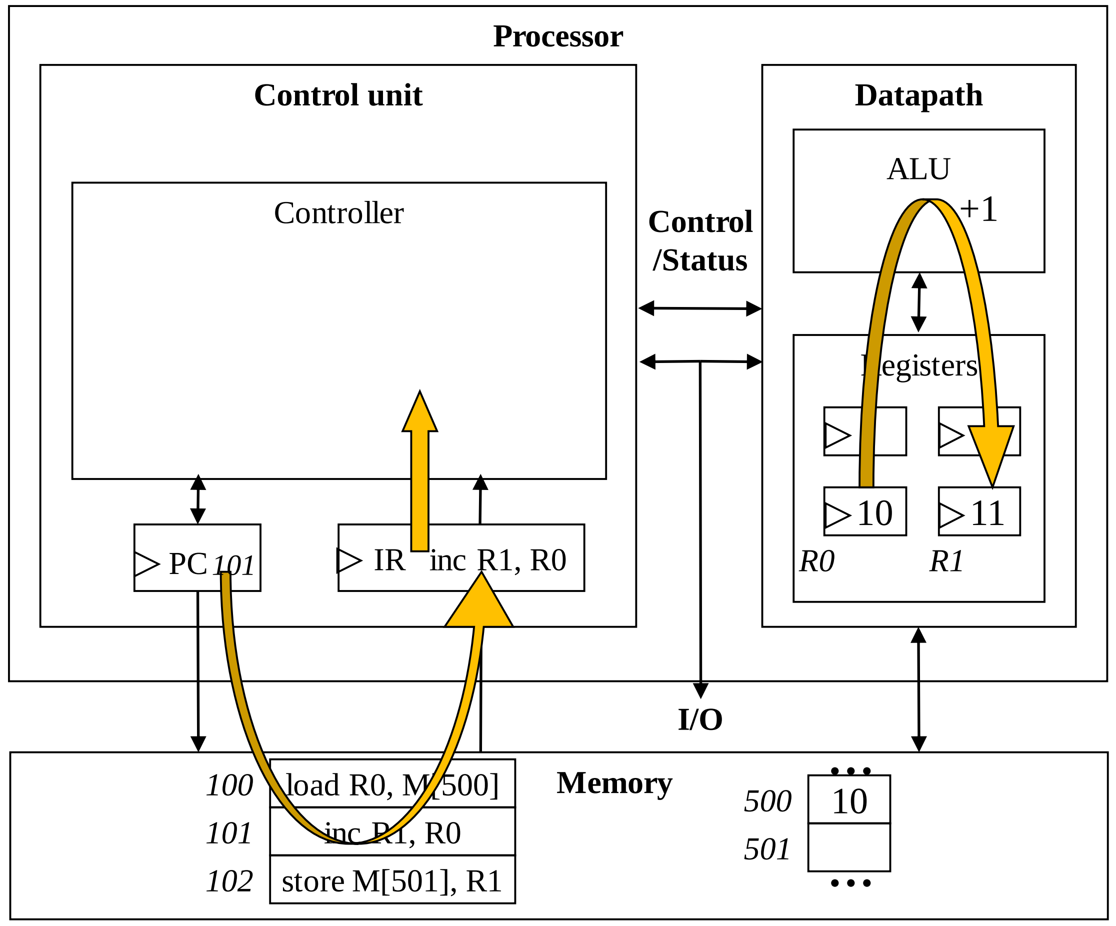
Instruction Cycles



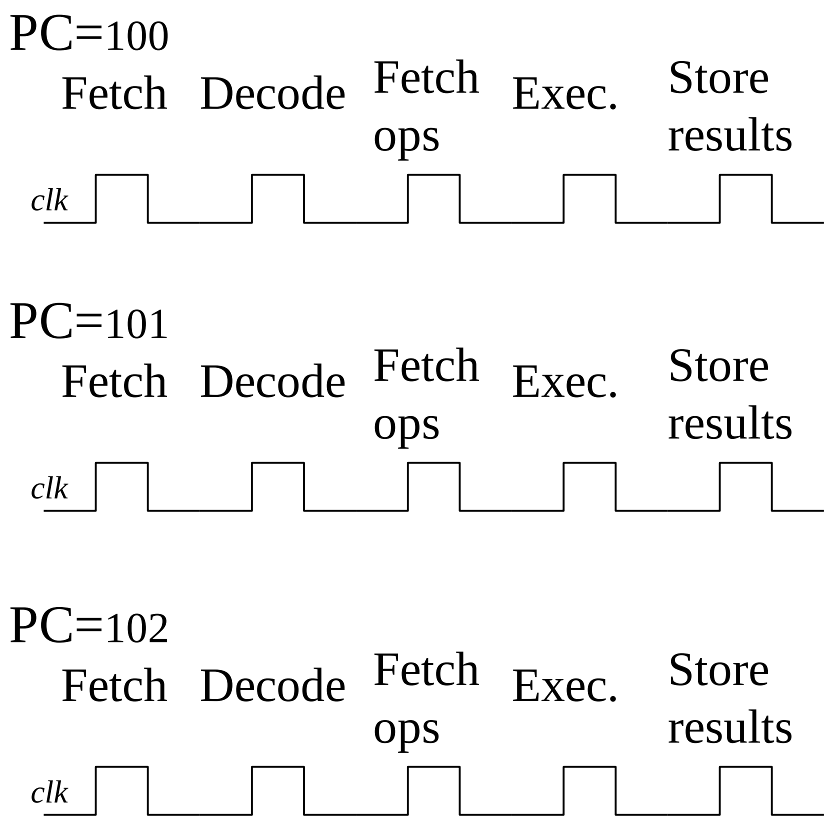


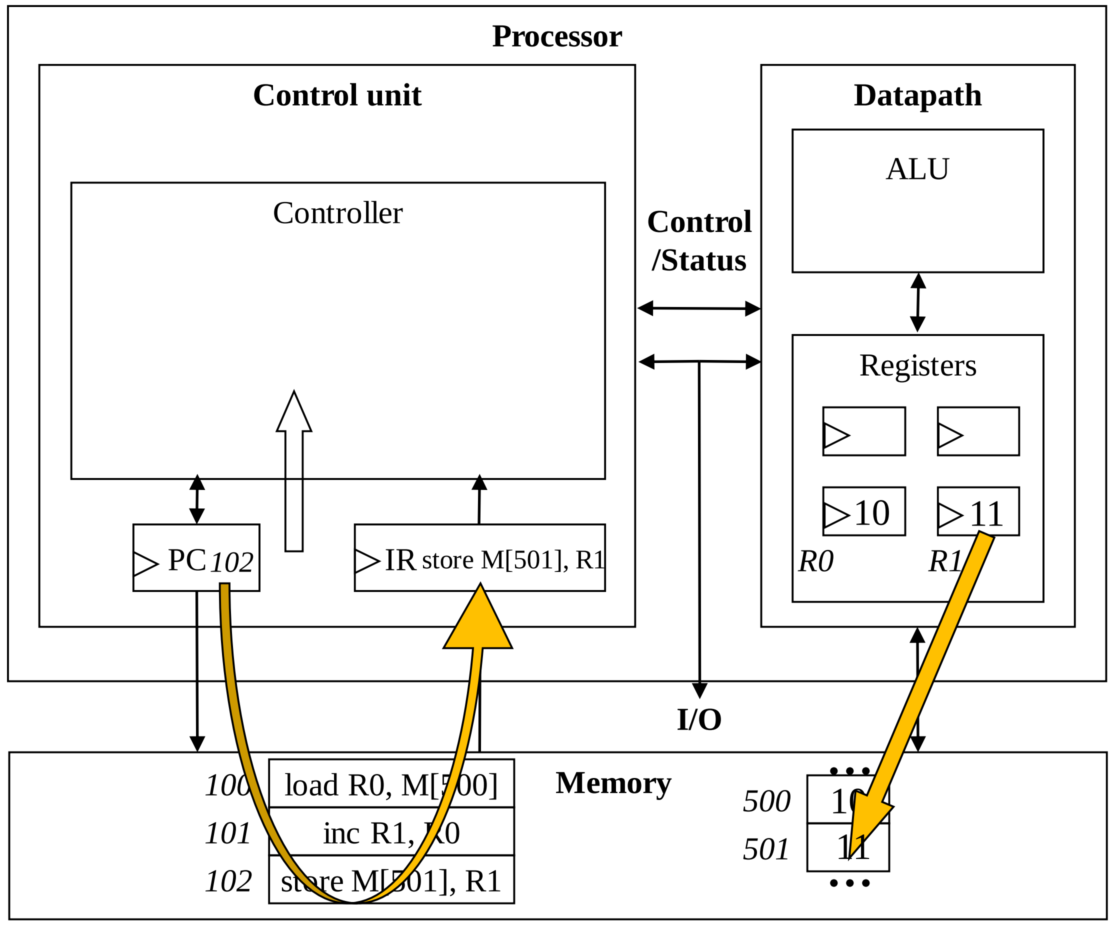
Instruction Cycles





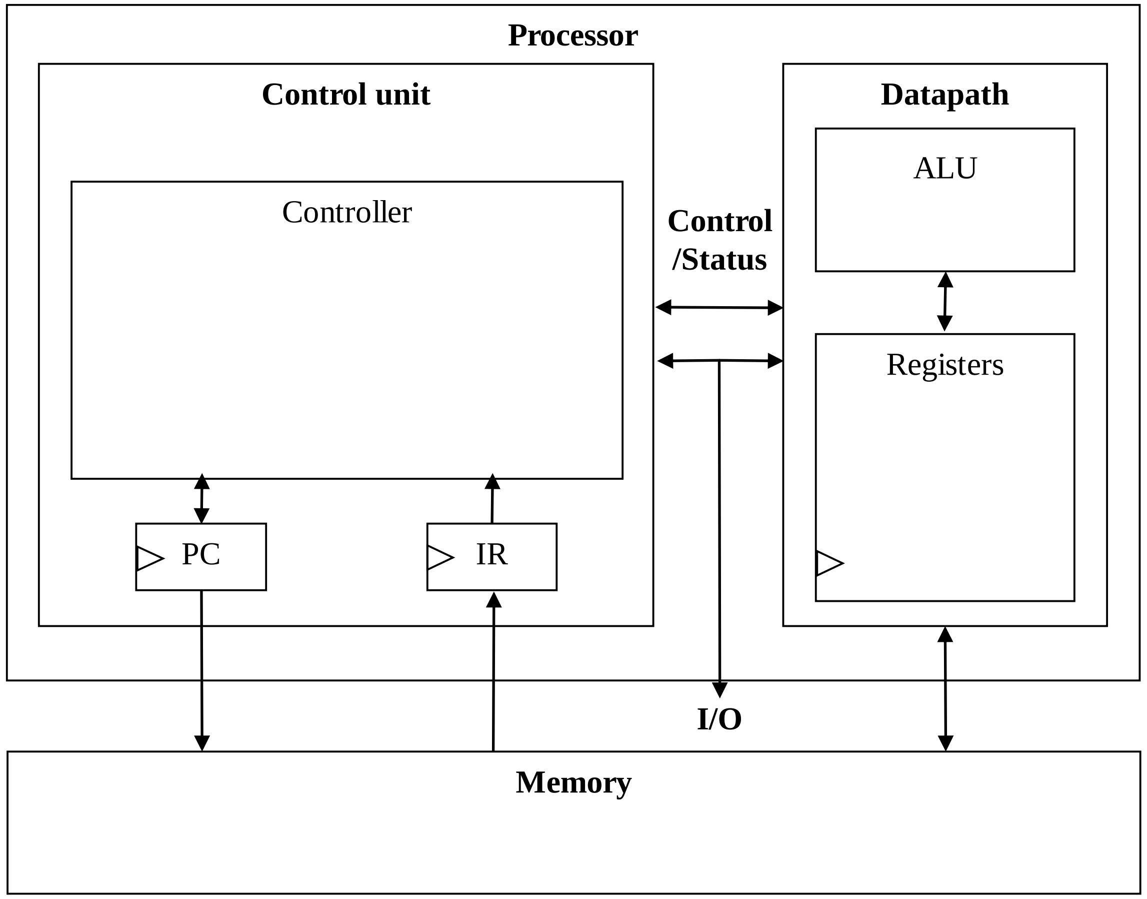
Instruction Cycles





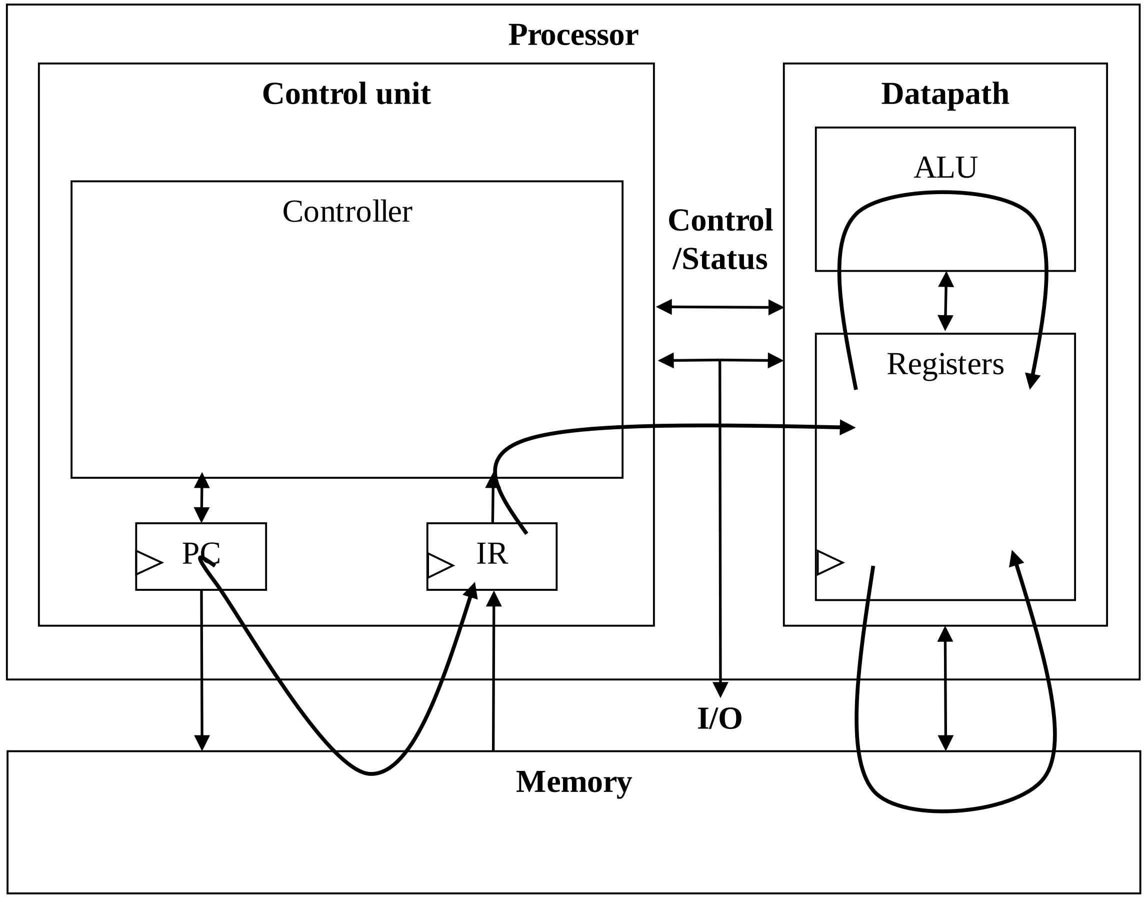
Architectural Considerations

- N-bit processor
  - N-bit ALU, registers, buses, memory data interface
  - Embedded: 8-bit, 16-bit, 32-bit common
  - Desktop/servers: 32 or 64 bit
- PC size determines address space
  - Often larger than data word size
  - PC size M, then address space is 2^M

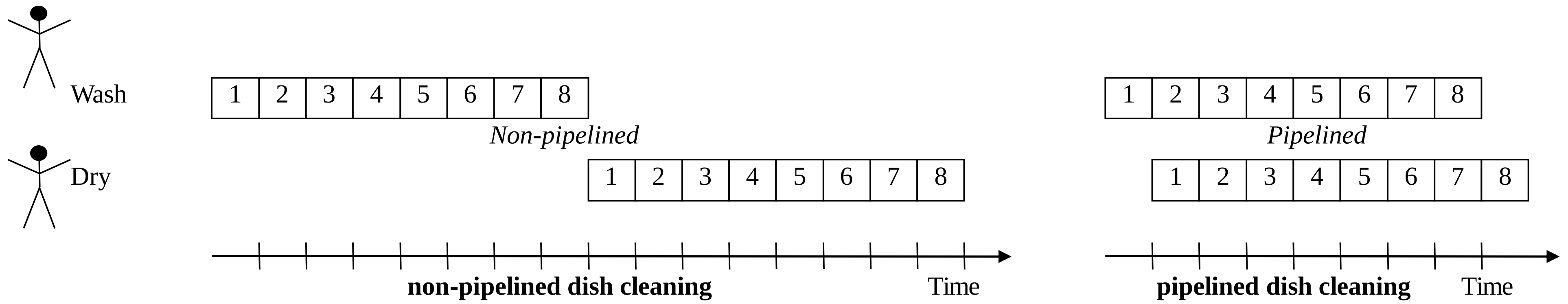


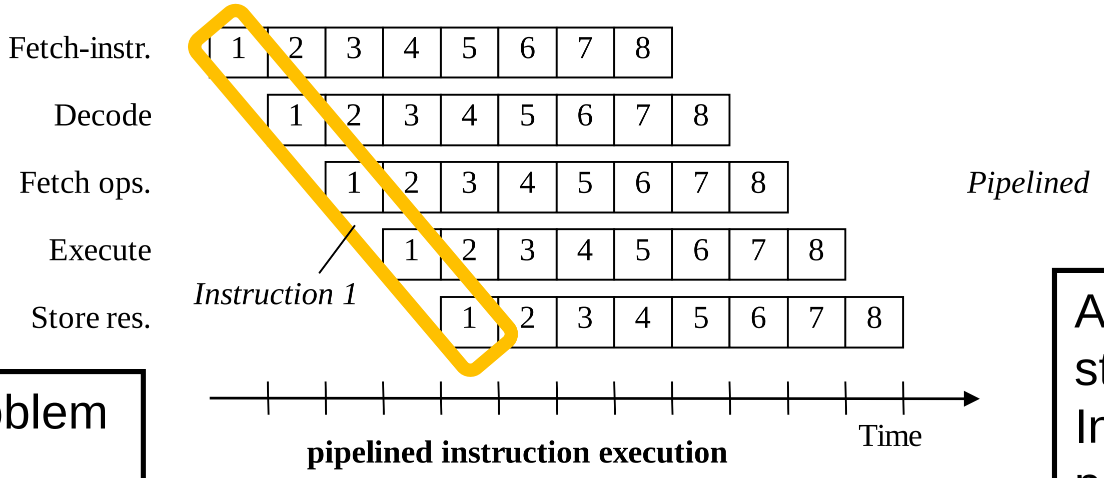
Architectural Considerations

- Clock period
  - Inverse of clock frequency
  - Must be longer than longest register to register delay in entire processor
  - Longest path is called critical path (i.e. from a datapath register through the ALU and back to a datapath register)
  - Memory access is often the longest



Pipelining: Increasing Throughput





Why is branching a problem for pipelining?
What are possible solutions?

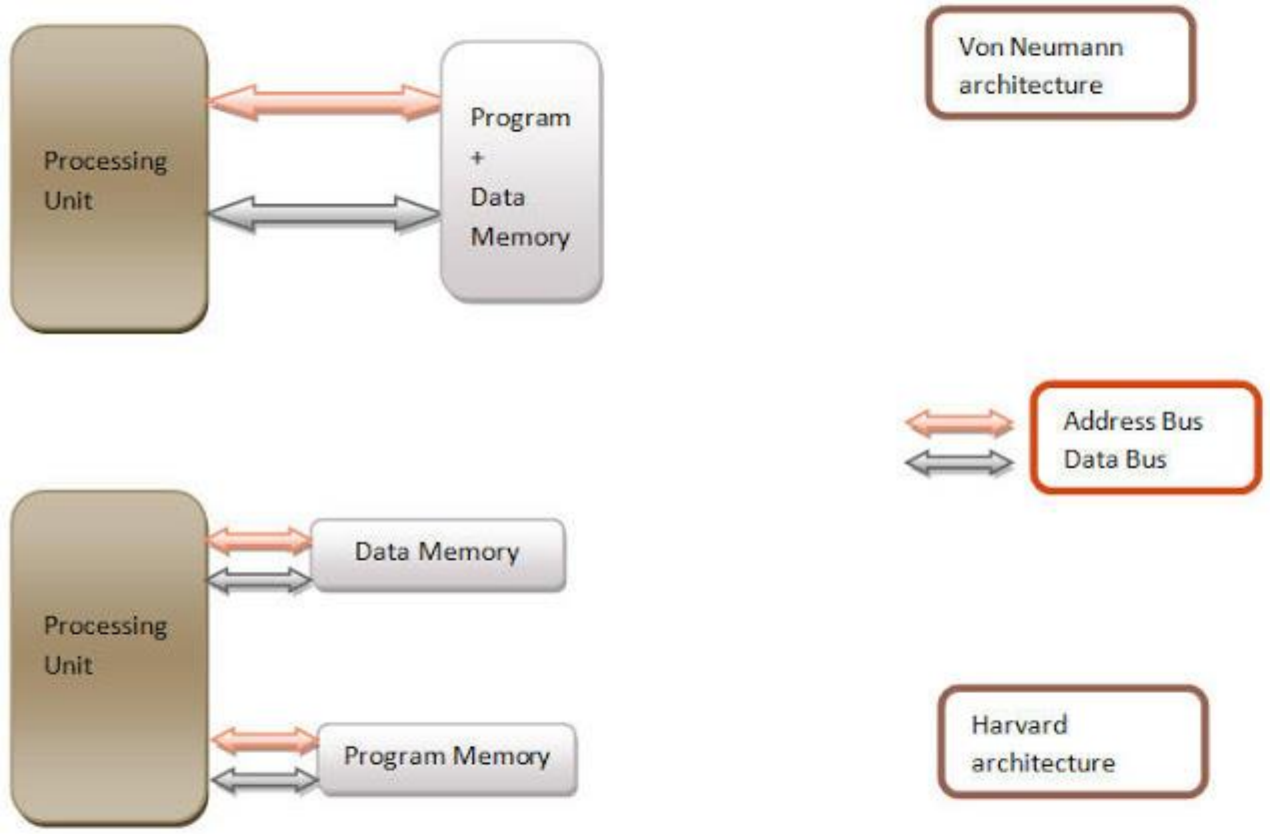
Atmel AVR has 2 stage pipeline
Intel Pentium 4 processors have 20 stage pipelines

2

Memory Architectures

Memory Architectures

- von Neumann
  - Single data bus which fetches both instructions and data
  - Fewer memory wires, i.e. simpler hardware
  - Bus is bottleneck (von Neumann Bottleneck)
- Harvard
  - Two data buses
  - Different address spaces
  - Simultaneous program and data memory access, i.e. improved performance



AVR uses a Harvard architecture

Advantage of Harvard Architecture

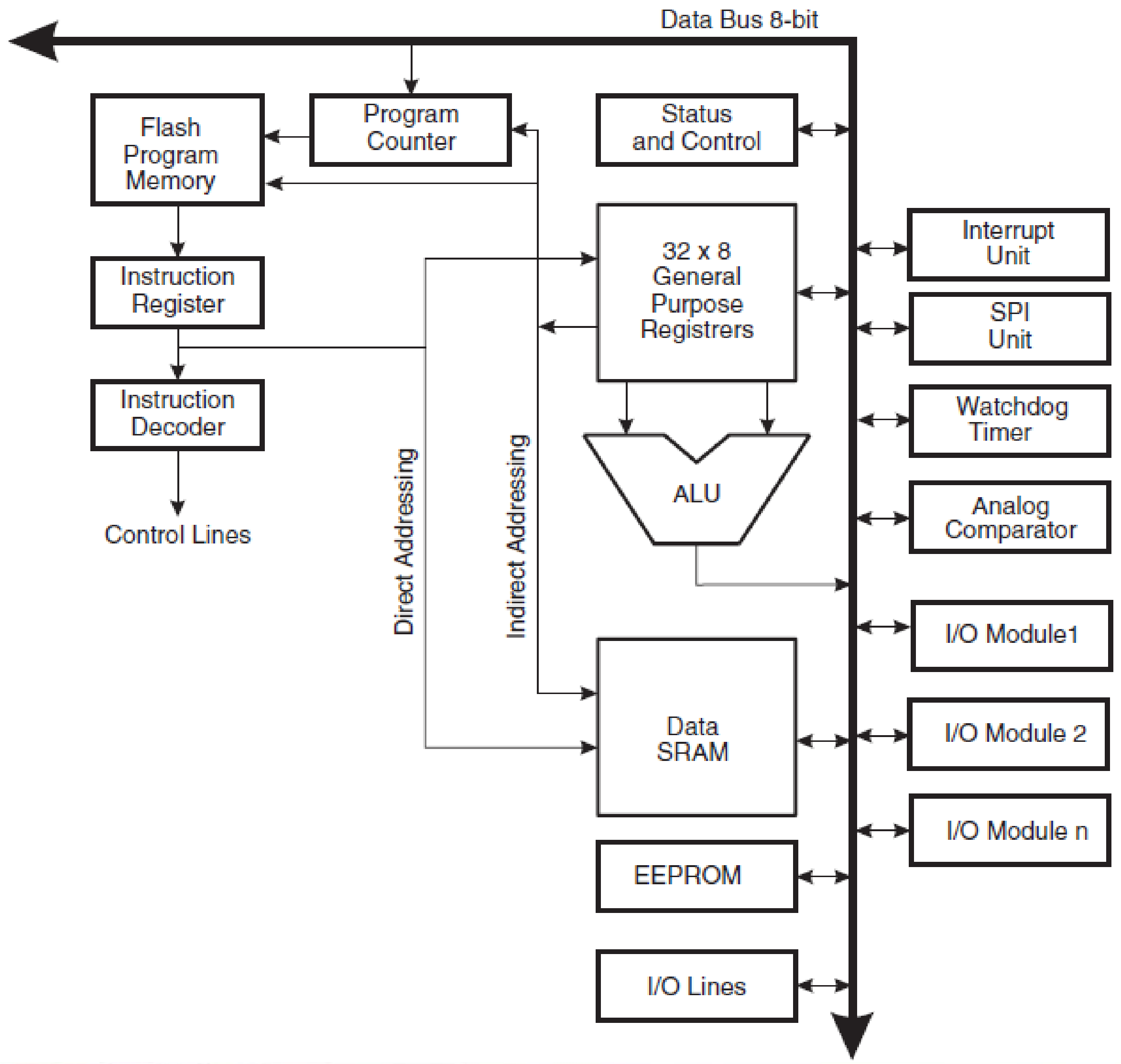
- Parallel fetching of data and instructions
- Optimized storage technology for each purpose
  - Program memory as ROM (e.g. Flash)
  - Data memory as RAM
  - Word length can be optimized
    - Data: byte wise access
    - Program: Size of instruction (e.g. 12, 14, 16 Bits)
- Size of address space is doubled
  - Example: 16 Bit addresses: von Neumann 2^{16} = 64 KBytes, Harvard 128 KBytes

Modified Harvard Architecture

- von Neumann model of stored program computing has advantage over *pure* Harvard machines
  - code can be treated as data, and vice versa
  - reading a program from disk storage and executing it
  - better memory utilization (trade program for data memory)
- Modified Harvard Architecture
  Variation of Harvard computer architecture that allows contents of instruction memory to be accessed as if it were data, allows code to be generated

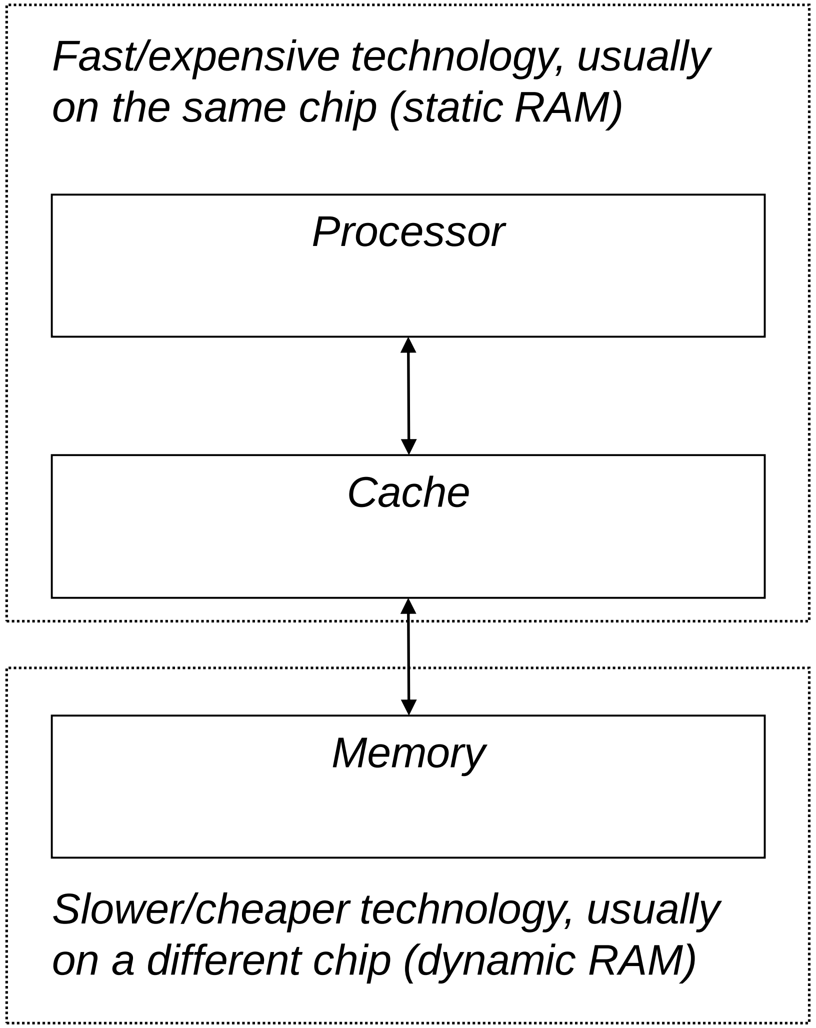
ATmega1281 uses a Harvard Architecture

Source: ATMEL



Cache Memory

- Memory access may be slow
- Cache is small but fast memory close to processor
  - Holds copy of part of memory
  - Hits and misses
  - Caches are used for instructions (I-cache) and data (D-cache)

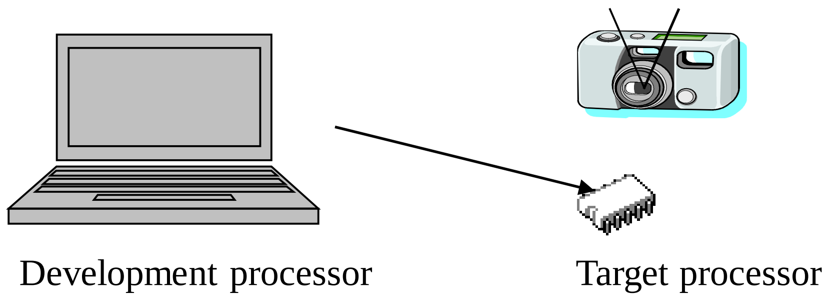

Fast/expensive technology, usually on the same chip (static RAM)
Processor
Cache
Memory
Slower/cheaper technology, usually on a different chip (dynamic RAM)

3

Application Development

Application Development

- Development processor
  - Processor on which we write and debug programs
    - Usually a PC, laptop
- Target processor
  - Processor the program will run on in embedded system
    - Often different from development processor



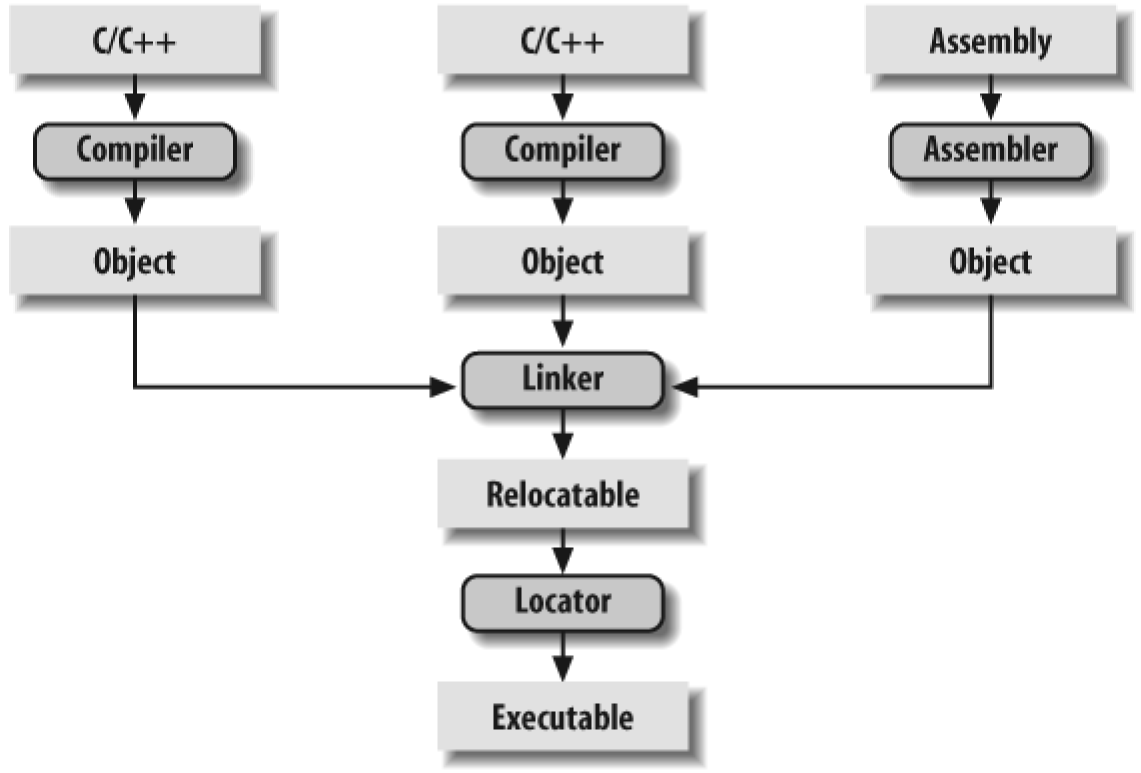
Toolchain

- Toolchain
  - Set of software development tools that are linked (or chained) together to produce an executable application for a processor
  - For embedded development cross toolchains are used
    - Toolchain runs on a development system of a specific architecture (such as x86) but produce binary code (executables) to run on target architecture (e.g. AVR)
- Optionally, a toolchain may contain other tools such as a debugger
- Various open source projects that comprise entire toolchain are available

Tools

- Preprocessor
  - Processes source code (inclusion of header files, macro expansions, conditional compilation, etc.)
- Compiler
  - Translates human readable code into assembly language for a particular processor
- Assembler
  - Translates assembly language into opcodes. Produces an object file
- Linker
  - Organizes object files, necessary libraries, and other data and produces a relocatable file
- Locator
  - Takes relocatable file and information about system’s memory and produces an executable, e.g. transform relocatable addresses into absolute addresses
- gcc (GNU Compiler Collection) takes care of *all steps* at once!

Embedded Software Development Process



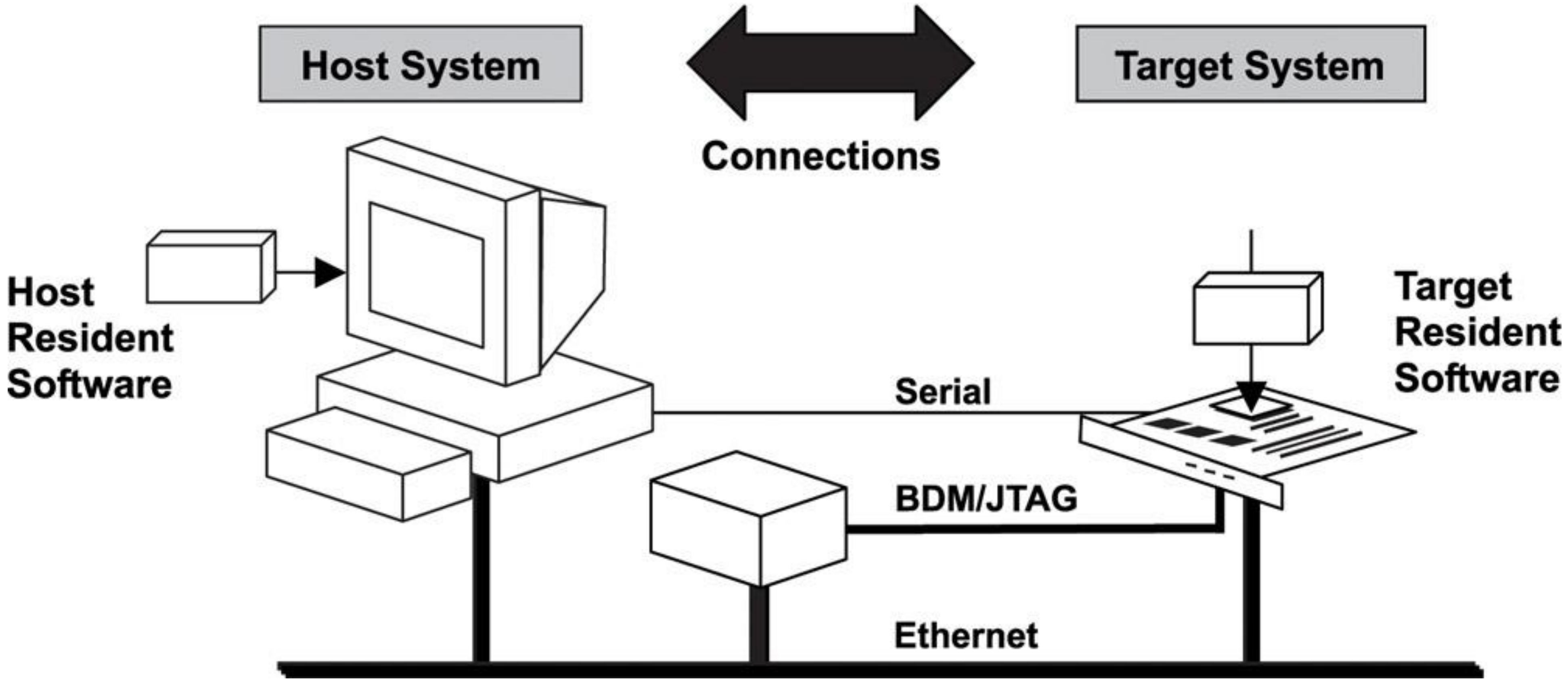
Tools: Embedded System Specifics

- Tools run on development computer, not on embedded computer
- Compiler
  - Has to know specific hardware, optimizes for size & speed
- Assembler
  - Produces “startup code”; not inserted automatically as in general purpose computers (i.e., programmer needs to compile it independently)
- Linker
  - Needs correct libraries
- Locator
  - Needs programmer input for information about memory
- Complete embedded tool chain often integrated into Integrated Development Environment (IDE)
  - e.g. Eclipse, AVR Studio

Moving program onto embedded system

- Compiled program (image) needs to get onto embedded system
- Methods to load image:
  - In-System Programming (ISP)
    - Programmed while installed in a complete system, rather than requiring chip to be programmed prior to installing it into system
    - Entire image installed into EEPROM or flash memory from serial port of PC (e.g. using SPI)
    - Downloading image through a JTAG or BDM interface
  - Bootloader
    - Data transfer utility program on host, as well as a target loader on embedded system
    - Loads programs onto system over a serial line (e.g. RS-232)
    - Requires no special hardware

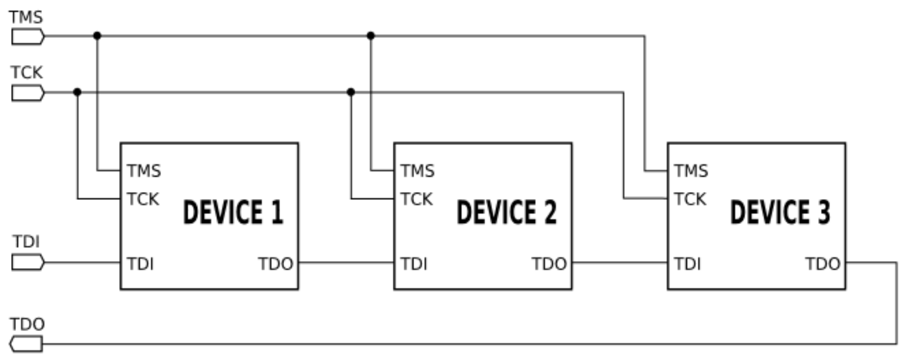
Cross-platform development environment



Quelle: book.opensourceproject.org.cn

Joint Test Action Group (JTAG)

- JTAG: Originally interface used for testing PCBs after assembly
- Chip's IO lines can be controlled and read via JTAG ports allowing a board test sequence to be performed
- JTAG also allows device programmer hardware to transfer data into internal non-volatile memory
- Advantage of JTAG
  - Devices can be daisy chained: One JTAG port can serve to program/ debug/test multiple devices



4

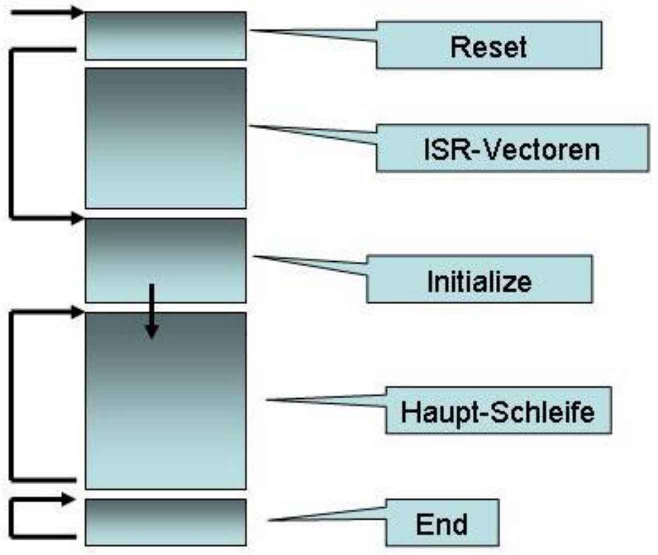
Programming

Programming

- Programmer doesn’t need detailed understanding of architecture
  - Instead, needs to know what instructions can be executed
- Three levels of instructions:
  1. Machine language
  2. Assembly level
  3. High-level structured languages (C, C++, Ada, Java, etc.)
- Today mainly high-level languages are used
  - They do not depend on hardware or abilities of CPU
  - But, some assembly level programming may still be necessary
- High-level languages provide device driver APIs
  - Driver: program communicating with and controls (*drives*) a device

Assembler in Microcontrollers

- Human-readable abbreviations replace binary code words of machine language
- Goal of assembler is to make hardware resources of processor accessible
  - CPU and ALU
  - storage units (internal/external RAM, EEPROM)
  - ports for timers, AD converters, and other devices
- Accessible means directly accessible and not via drivers or other interfaces (e.g. as in operating systems)
- Complete hardware is at programmers command



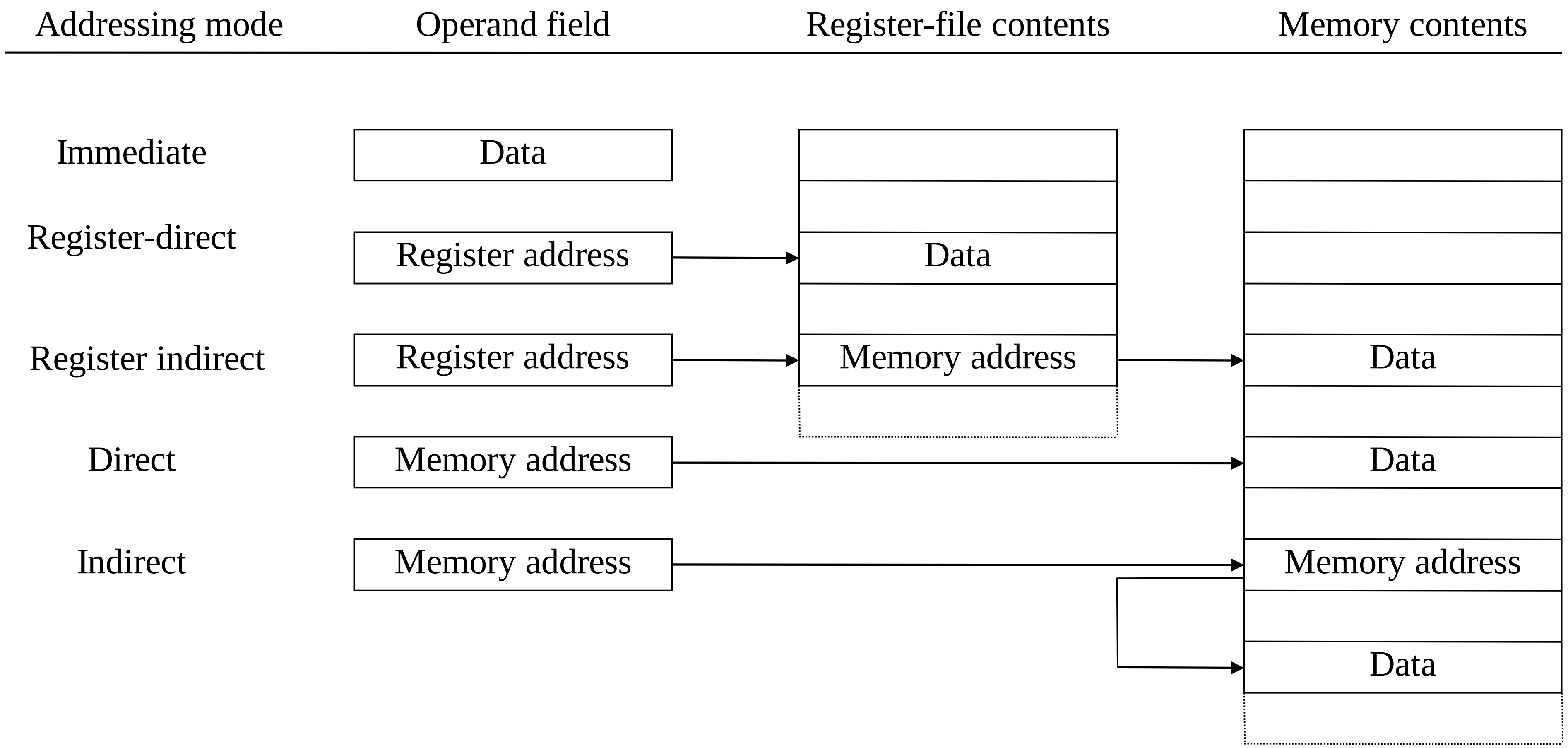
Assembly-Level Instructions

Instruction 1	opcode	operand1	operand2
Instruction 2	opcode	operand1	operand2
Instruction 3	opcode	operand1	operand2
Instruction 4	opcode	operand1	operand2
	...		

- Structure
  - opcode field
  - operand fields
    - number varies (source and destination)
    - addressing modes

- Instruction Set: Legal set of instructions of processor
  - Data transfer: memory/register, register/register, I/O, etc.
  - Arithmetic/logical: move register through ALU
  - Branches: determine next PC value when not just PC+1
- AVR instructions are 16 or 32 bits wide

Addressing Modes



A Simple Instruction Set

Assembly instruct.	First byte		Second byte		Operation
MOV Rn, direct	0000	Rn	direct		Rn = M(direct)
MOV direct, Rn	0001	Rn	direct		M(direct) = Rn
MOV @Rn, Rm	0010	Rn	Rm		M(Rn) = Rm
MOV Rn, #immed.	0011	Rn	immediate		Rn = immediate
ADD Rn, Rm	0100	Rn	Rm		Rn = Rn + Rm
SUB Rn, Rm	0101	Rn	Rm		Rn = Rn - Rm
JZ Rn, relative	0110	Rn	relative		PC = PC+ relative (only if Rn is 0)
	opcode	operands			

Destination is (almost) always given first, then source!

Sample Programs

C program

```
int total = 0;
for (int i=10; i!=0; i--)
   total += i;
// next instructions...
```

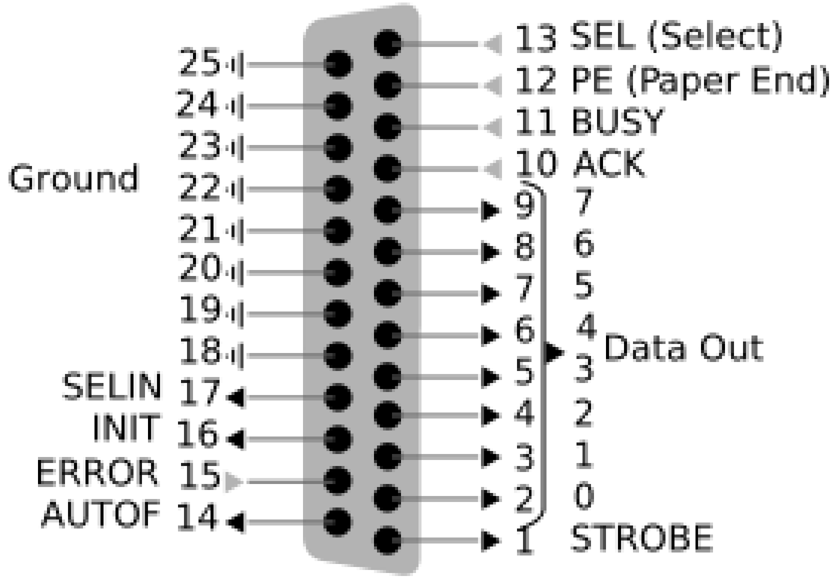
Equivalent assembly program

```
  0       MOV R0, #0;      // total = 0
  1       MOV R1, #10;     // i = 10
  2       MOV R2, #1;      // constant 1
  3       MOV R3, #0;      // constant 0
Loop:     JZ R1, Next;     // Done if i=0
  5       ADD R0, R1;      // total += i
  6       SUB R1, R2;      // i--
  7       JZ R3, Loop;     // Jump always
Next:     // next instructions...
```

- Try some others
  - Handshake: Wait until the value of M[254] is not 0, set M[255] to 1, wait until M[254] is 0, set M[255] to 0 (assume those locations are ports)
  - (Harder) Count the occurrences of zeros in an array stored in memory locations 100 through 199

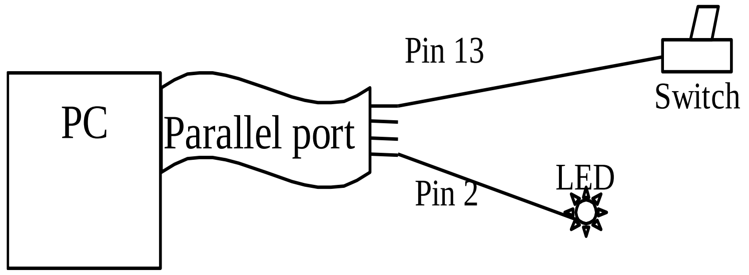
LPT (Line Print Terminal)

- An LPT port (parallel port) has an
  - 8-bit data bus
  - four pins for control output (Strobe, Linefeed, Initialize, and Select In), and
  - five more for control input (ACK, Busy, Select, Error, and Paper End)
- Data transfer speed
  - 12,000 kbit/s



Example: Parallel Port Driver

LPT Connection Pin	I/O Direction	Register Address
1	Output	0th bit of register #2
2-9	Output	0th - 7th bit of register #0
10,11,12,13,15	Input	6,7,5,4,3th bit of register #1
14,16,17	Output	1,2,3th bit of register #2

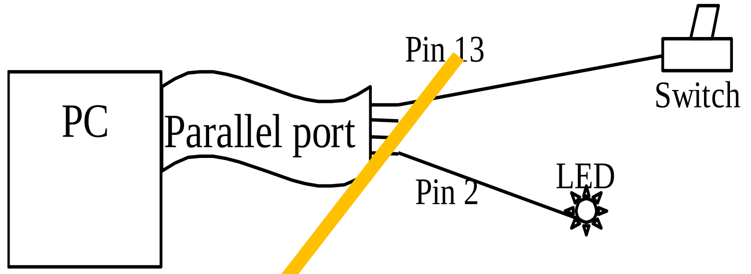


- Using assembly language programming we can configure a PC parallel port to perform digital I/O
  - write and read to three special registers (#0, #2 for output, #1 for input) to accomplish this
  - table provides list of parallel port connector pins and corresponding register location
- Example: parallel port monitors the input switch and turns the LED on/off accordingly

Parallel Port Example

```
; This x86 program consists of a sub-routine reading the state
; of the input pin, determining the on/off state of our switch
; and asserts the output pin, turning LED on/off accordingly

.386

checkPort   proc
push    al                   ; save the content
push    dx                   ; save the content
mov     dx, 0x3BC + 1        ; base + 1 for register #1
in      al, dx               ; read register #1
and     al, 0x10             ; mask out all but bit # 4
cmp     al, 0                ; is it 0?
jne     SwitchOn             ; if not, turn LED on

SwitchOff:
mov     dx, 0x3BC + 0        ; base + 0 for register #0
in      al, dx               ; read current state of port
and     al, 0xfe             ; clear first bit (masking)
out     dx, al               ; write it out to the port
jmp     Done                 ; we are done

SwitchOn:
mov     dx, 0x3BC + 0        ; base + 0 for register #0
in      al, dx               ; read current state of port
or      al, 01h              ; set first bit (masking)
out     dx, al               ; write it out to port

Done:
pop     dx                   ; restore the content
pop     al                   ; restore the content
checkPort   endp
```

```
extern "C" checkPort(void);    // defined in assembly

void main(void) {
   while (1) {
      checkPort();
   }
}
```



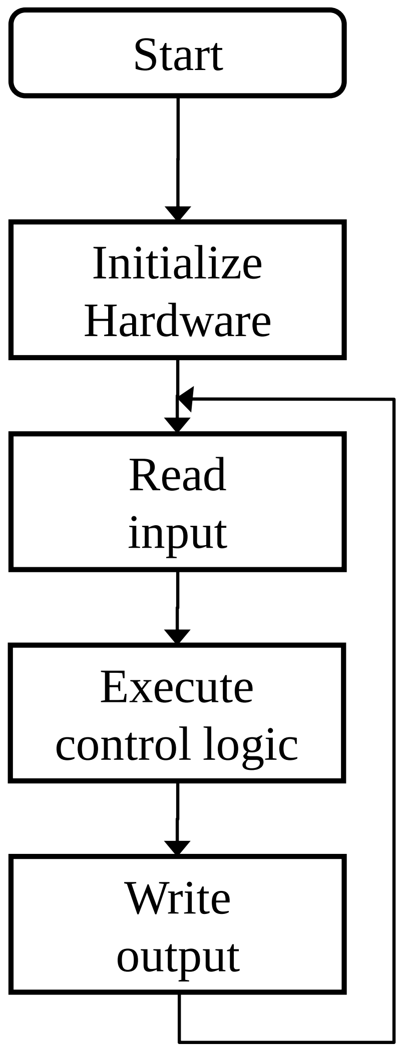
LPT Connection Pin	I/O Direction	Register Address
1	Output	0th bit of register #2
2-9	Output	0th – 7th bit of register #0
10,11,12,13,15	Input	6,7,5,4,3th bit of register #1
14,16,17	Output	1,2,3th bit of register #2

Address of register #0 is 0x3BC

Parallel Port Example

- General register al, dx are saved/restored
- Different notation for hex values: 72h, 72H, 0x72, \$72, 72_{16};
- Switch 0 means switch off
- In/out instructions read/write internal registers
  - Operands: address and data
  - In: content of 8-bit operand will be written to addressed register
  - Out: content of addressed 8-bit register will be read into operand (operation direction right to left)
- Address is calculated by adding address of device (0x3BC) to address of register
- camp al, imm8 : Compares first operand with second operand and sets status flags in EFLAGS register according to result
- cmp instruction typically used in conjunction with conditional jump

Implementation of an Embedded Program



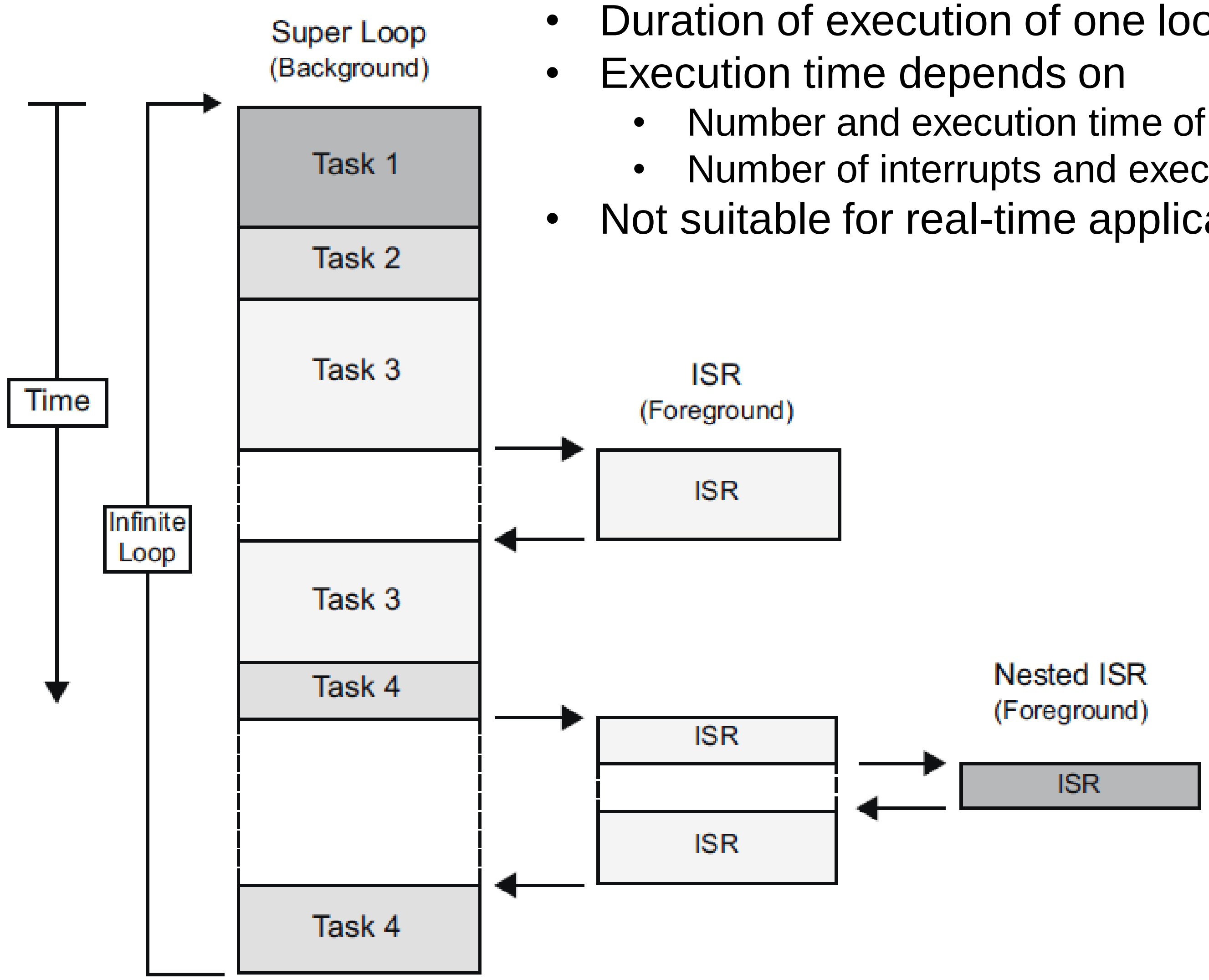
- Super Loop:
  - runs forever
  - is used for tasks of low/medium complexity
  - cannot be used to periodically execute a function at fixed intervals
  - only interrupts intercept the super loop

Super Loop

```
int main(void) {
    initialize();          // executed once

    // super loop starts here
    // tasks are executed repetitively
    while (1)    {
        task1();        // e.g. read temperature
        task2();        // e.g.  process temperature
        task3();        // e.g. display temperature
        delay_loop();    // to slow down the process
    }
}
```

- In delay processor may transit into low power mode
- Alternative: Scheduler, operating system (OS)

Super Loop with Interrupts



- Duration of execution of one loop is unknown
- Execution time depends on
  - Number and execution time of tasks
  - Number of interrupts and execution time of ISRs
- Not suitable for real-time applications

Operating System

- Software layer providing low-level services to a program
  - Loading & executing programs, process
  - Sharing, allocate and manage system resources
  - File management, disk access, keyboard/display interfacing
  - Scheduling multiple programs for execution
  - Program makes system calls to OS
  - System Call:
    Mechanism for application to invoke the OS via software interrupt

```
DB file_name “out.txt”      -- store file name

MOV R0, 1324                -- system call “open” id
MOV R1, file_name           -- address of file-name
INT 34                      -- cause a system call
JZ  R0, L1                  -- if zero -> error

   ... read the file
JMP L2                      -- bypass error cond.
L1:
   ... handle the error

L2:
   ...
```

5

Running a Program

Running a Program

- If development processor is different than target, how can we run our compiled code? Two options:
  - Download to target processor
  - Simulate
- Simulation
  - One method: Hardware description language
    - But slow, not always available
  - Another method: *Instruction set simulator (ISS)*
    - Runs on development processor, but executes instructions of target processor

A Simple Instruction Set

Assembly instruct.	First byte		Second byte		Operation
MOV Rn, direct	0000	Rn	direct		Rn = M(direct)
MOV direct, Rn	0001	Rn	direct		M(direct) = Rn
MOV @Rn, Rm	0010	Rn	Rm		M(Rn) = Rm
MOV Rn, #immed.	0011	Rn	immediate		Rn = immediate
ADD Rn, Rm	0100	Rn	Rm		Rn = Rn + Rm
SUB Rn, Rm	0101	Rn	Rm		Rn = Rn - Rm
JZ Rn, relative	0110	Rn	relative		PC = PC+ relative (only if Rn is 0)
MOV Rn, @Rm	0111	Rn	Rm		Rn = M(Rm)
	opcode	operands			

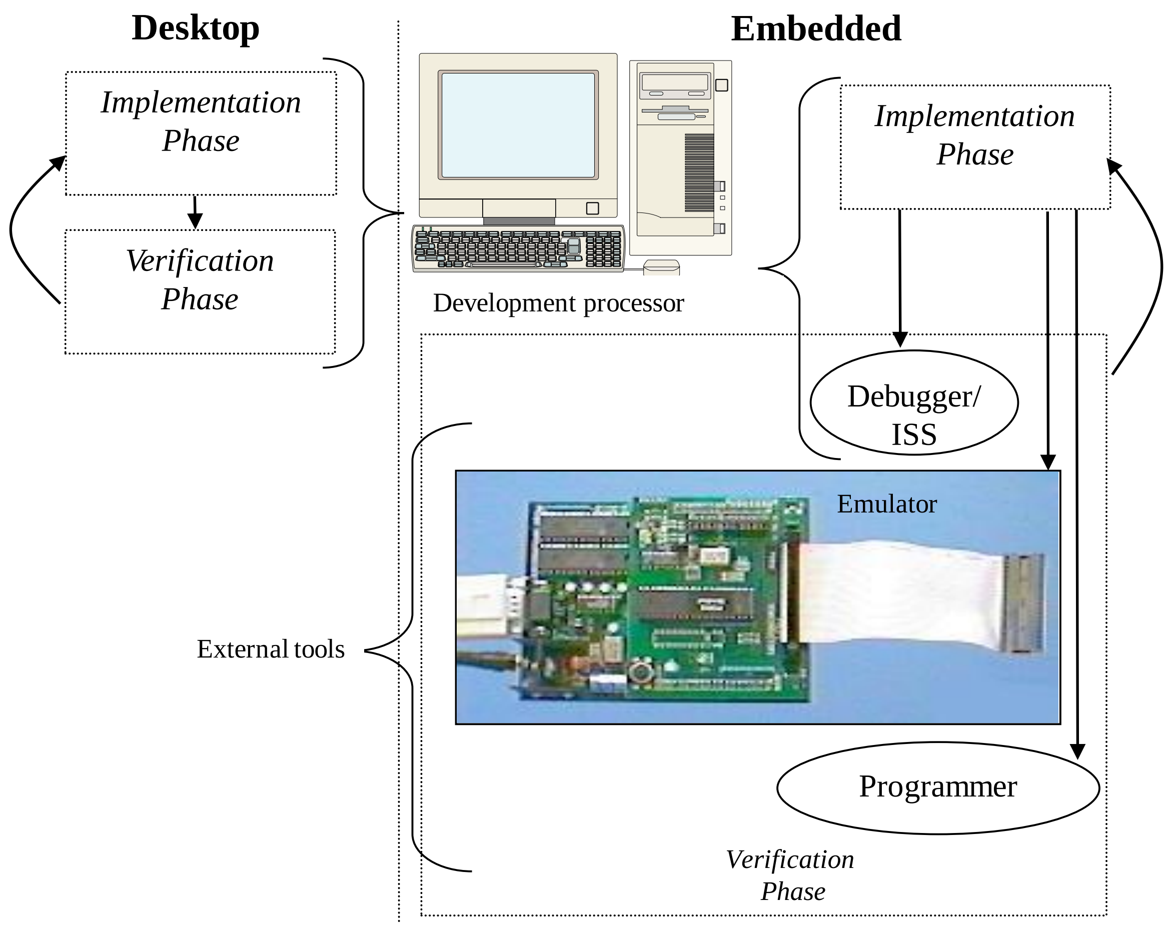
Destination is always given first, then source!

Example for Instruction Set Simulator

```
#include <stdio.h>
typedef struct {
  uint8_t first_byte, second_byte;
} instruction_t;

instruction_t program[PROGRAMSIZE];   // instruction memory
uint8_t memory[MEMORYSIZE];           // data memory
uint8_t reg[NUM_REGISTERS];           // register file

int8_t  run_program(int16_t  num_instr) {
  int16_t  pc = -1;
  uint8_t fb, sb, fb_low, sb_high;

  while( ++pc < num_instr ) {
    fb = program[pc].first_byte;
    sb = program[pc].second_byte;
    fb_low = fb & 0x0f;        // Rn
    sb_high = sb >> 4;         // Rm
    switch( fb >> 4 ) {        // opcode
      case 0: reg[fb_low] = memory[sb]; break;
      case 1: memory[sb] = reg[fb_low]; break;
      case 2: memory[reg[fb_low]] =
              reg[sb_high];  break;
      case 3: reg[fb_low] = sb; break;
      case 4: reg[fb_low] += reg[sb_high]; break;
      case 5: reg[fb_low] -= reg[sb_high]; break;
      case 6: if (req[fb_low] == 0) pc += sb; break;
```

```
      case 7: reg[fb_low] = memory[reg[sb_high]]; break
      default: return -1;
    }
  }
  return 0;
}

void print_memory_contents() {
  // Output of content of memory
}

int16_t main(int argc, char *argv[]) {
  FILE * ifs;

  if (argc != 2 || (ifs = fopen(argv[1], “rb”) == NULL ) {
    return -1;
  }

  if (run_program(fread(program, sizeof(instruction),
    sizeof(program)/sizeof(instruction), ifs)) == 0)
  {
    print_memory_contents();
    return 0;
  }  else  {
    return -1;
  }
}
```

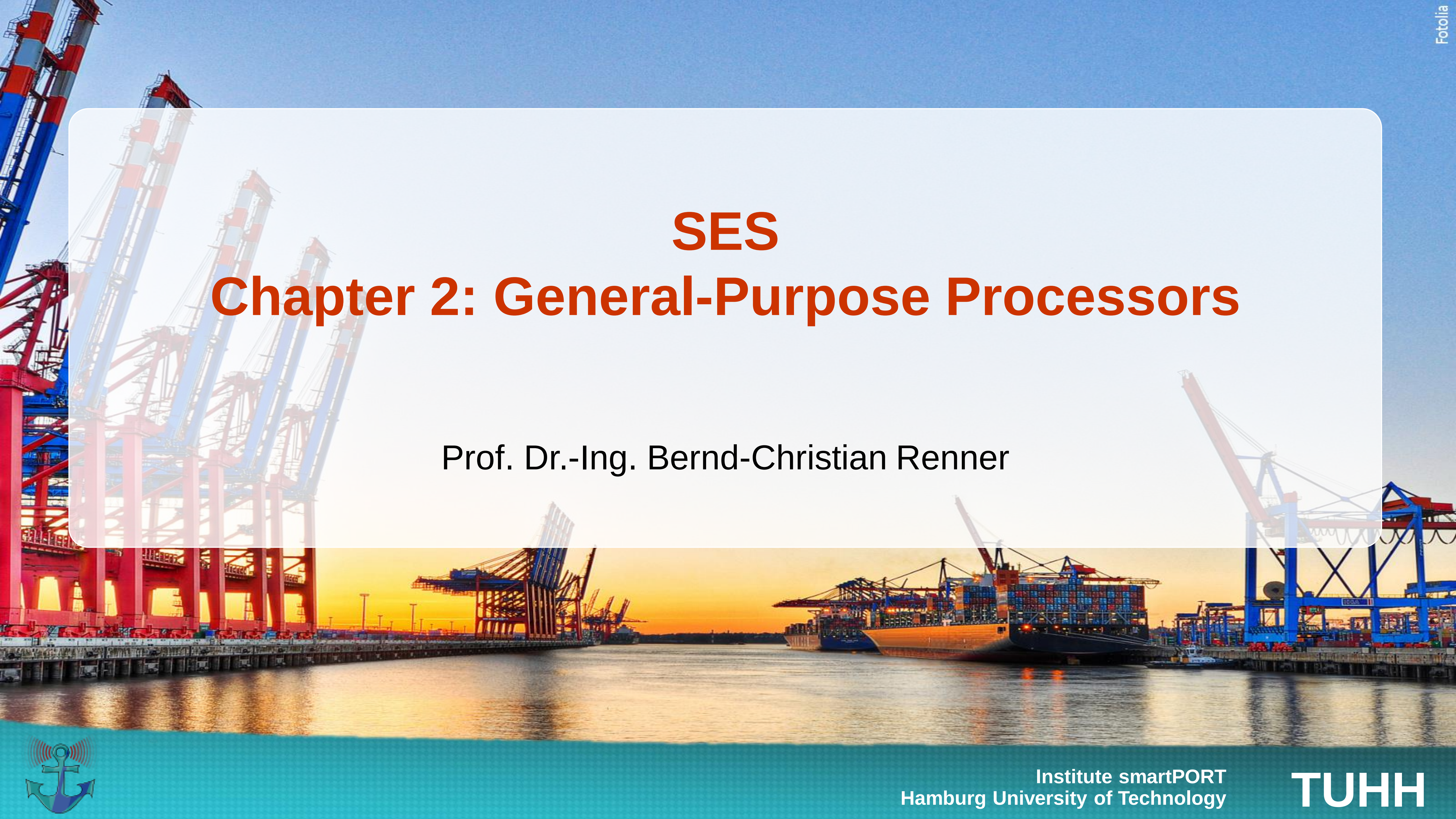
Testing and Debugging



- ISS
  - Gives control over time – set breakpoints, look at register values, set values, step-by-step execution, ...
  - But, doesn't interact with real environment
- Download to board
  - Use device programmer
  - Runs in real environment, but not controllable
- Compromise: emulator
  - Runs in real environment, at speed or near
  - Supports some controllability from the PC

Chapter Summary

- General-purpose processors
  - Good performance, low NRE, flexible
- Controller, datapath, and memory
- Structured languages prevail
  - But some assembly level programming still necessary
- Many tools available
  - Including instruction-set simulators, in-circuit emulators



SES
Chapter 2: General-Purpose Processors

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH