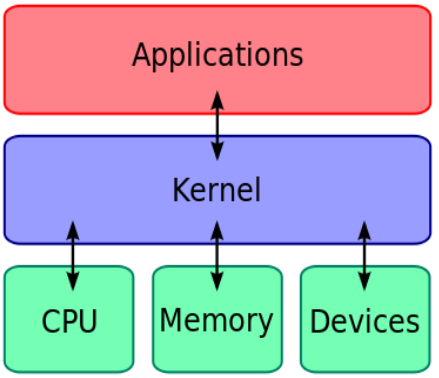
DAT111/DAT112 Sammendrag

Forelesninger

01. Introduction / microprocessor

Operating System Mode



- Most OS are built up from multiple layers
- Each layer has a specific responsibility
- Each layer only "talks" to the layer below it (with exceptions)
- Each layer can be replaced without replacing the layer above or below
- User applications talk to the top most layer
    - Or an API that talks to that layer
- The Kernel is the central component of the OS
    - OS and the kernel are sometimes used interchangeably
    - OS is a term for all the components, not just the kernel

OS kernel and kernel API

- The OS kernel talks to the hardware using drivers
- A driver can either be built into the kernel or loaded on demand
- Most OSes ship with a lot of driver – but not all (Require installation)
- Each driver supports one specific device or a set of similar devices
    - Network card (Realtek)
    - Printer (OKI C301)
    - Sound card (Sound Blaster Z)
- Some device types are now standardized – mostly USB (with exceptions)
    - Mice
    - Keyboards
    - External storage
- The Kernel API can be used by applications to talks to these devices
    - This is called a user-space API to the kernel
    - Do file IO – Independent of what harddrive you have
    - What keys are pressed – independent of what keyboard you have
- The Kernel API is OS specific (Linux / Windows etc.)
- We have not used the kernel API directly
    - Usually avoided as it's often hard to use
    - The C standard library talks to kernel of Linux/Windows for us.

OS process model

The OS is responsible for process startup

- Loads any libraries the process needs to memory.
    - .so (Linux)
    - .dll (Windows)
- Loads the application itself to memory

- Allocates initial heap and stack memory to the process

The OS keeps track of all the processes that are running

- Each process is assigned a process id - PID
- Can be user applications (your hand-in, Word, etc.)
- Or services (processes that run in the background doing work)

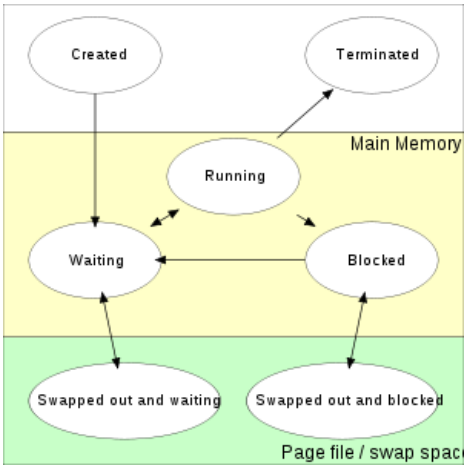
The OS protects processes from each other

- If Word crashes it can't take any other processes with it
- Word can't read memory belonging to another process
- If the process behaves badly (uses all memory or CPU) the OS kills it

The OS allocates CPU time to processes

- Processes are constantly paused and unpaused by the scheduler to perform work
- On systems with multiple CPUs or CPU cores more than one process runs simultaneously

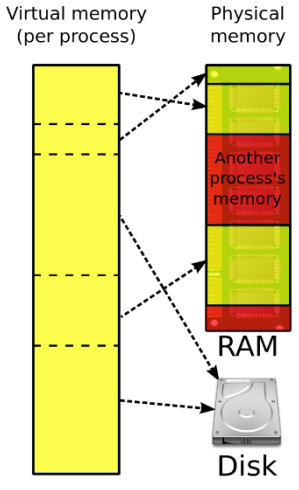
Scheduler Process States



1. Processes start in the created state when they are loaded
2. Once loaded they are moved to the waiting state
3. The scheduler gives the process some CPU time and it is moved to the running state
4. The process tries to fetch some data that is not yet ready and is moved to the blocked state
5. The data is finally available (e.g. read from the harddrive) and the process is continues in the running state
6. This goes on until the process is terminated

While the process is in the waiting state it might be written to the hard drive to save memory. This is called swapping and is generally not good because it makes the computer unresponsive. More memory can help prevent this.

OS Memory Model



- The OS keeps track of all allocated and free memory
- The OS is responsible for giving memory to applications that ask for it - C malloc() and C++ new
- The OS keeps track of virtual memory addresses
    - malloc() returns a pointer, which is just a memory address, e.g. 0x2f8a5178
    - This is not the actual address in physical memory but a virtual address for this process
    - This is called virtual memory
    - Virtual addresses are translated to physical addresses in a piece of hardware called the MMU (Memory Management Unit)
    - Not all hardware have an MMU - no virtual memory support
- The OS + MMU validate that access to any given address is allowed
    - Reading / writing to invalid addresses crash the process
    - Try it yourself: int* crash = (int*)0; *crash = 42;

02. CPU, Memory, Storage, IO

Microcontroller CPU

Introduction

- Often called microprocessor
- Microcontroller CPUs are generally simple and low power
- They are meant to perform simple tasks
- Programmed for a specific purpose
- Central component of the microcontroller SoC

Technical

- Uses a specific instruction set (Architecture) - desktops are x86/ x86-64
    - AVR8
    - AVR32
    - ARM (Multiple variants)
    - MIPS
    - .etc
- Clocked anywhere from 1MHz to 200MHz..
- Most support external memory and flash (usually optional)
- Most do not have an MMU – so no full Oss like Linux

Microcontroller Storage

Introduction

- Microcontroller SoCs usually have some built in storage
- Called on-chip flash
- There's not a lot of it
    - ATmega (8-bit AVR) comes with 32Kbyte
- SoCs are usually available with multiple sizes
- The PCB can be smallet without external storage
- No external storage also lowers cost

Technical

- NAND Flash is the most common type of storage, Read and written in pages ~2Kbyte – slow.
- EEPROM is faster and more expensive, Read and write can be done at byte level.
- Internal is low power compared to external flash

Starting up

- Programming a board means to write the flash with your program.
- When the board starts up it loads your program from flash to memory.
- Some microcontrollers can also run code directly from flash.

Microcontroller Memory

- SoCs often have some built in memory on the chip
- Called on-chip memory
- It's fast compared to external memory.
- It's low power compared to external memory
- SoCs are usually available with multiple sizes
- The pcb can be smaller without external memory

02. Microcontroller IO – Memory mapping

What is it

- Microcontrollers usually don't have a screen
- The "user interface" is simple input and output
- If you don't have an MMU all memory address are physical

How it works

- You "talk" to the microcontroller by reading and writing memory
- Do this from code either directly with pointers, or a library.
- The SoC/ SBC documentation describes the memory map
- Some IO is already connected (onboard buttons / Lights)
- Other IS is free for you to use (pins on the board)

Definitions

- A register is a memory address reserved for functionality
- A port is a register reserved for IO

Microcontroller IO – GPIO

Simple ON / OFF signals

- General purpose input output
- Physically connects to a pin on the SoC
- Can either be input or output (data direction)
    - Reads as 0 or 1
    - Written as 0 or 1
- Prefect for buttons (input) or lights (output)

Components

- Data direction register (DDR).
    - Sets if the port is input or output
- Data register
    - Read to get input, write to set output
- Usually 8 bit registers
- 8bit registers
    - 8 inputs or outputs
- 16bits registers
    - 16 input or outputs

03. IO continued – Digital output

Digital inputs

- Digital inputs read as 0 (low) or 1 (High)
- This is achieved by connecting the pin to either
    - Ground 0v
    - Power 5v
- Inputs have a maximum amount of current they can sink[1]
    - Usually measured in Ma[2]

[1] a flow of positive charges "away from the eye"
a flow of negative charges "towards the eye"

[2] milliampere

    - Arduino Uno is rated for 20 mA sustained source/ sink
    - There are additional limitations on total power sink
- Sinking creates heat which is why there's a limit (chip design)
- The sink limit is the reason we use a resistor when connecting a button
- Resistors limit current so they are used to make sure we don't break the chip

Pullups / pulldowns

- An input that isn't connected to anything is called a floating input
- A floating input will give random values depending on interference
- To lock the value of a floating input you use a resistor
- This resistor can be connected to either ground or 5v
    - Power (Pullup)
        - Input will read as 1 when open, 0 when closed
    - Ground (Pulldown)
        - Input will read as 0 when open, 1 when closed
- The resistor is usually ~10K Ω, see the datasheet for details
- Some inputs can also have an internal pullup/ pulldown resistor
    - If not, they need to be turned on
    - Once turned on you don't need an external (visible) resistor at all
- The Arduino Uno har internal pullup resistors
    - Meaning: You connect your button to the input and ground
        - Internal pullup will ensure the input reads as 1 when not pressed.

04. Interrupts and timers

Polling vs Interrupts

- Polling
    - Means continuously reading an input waiting for a value.
- Interrupt
    - Runs when a condition occurs
    - The code is paused when the interrupt occurs and resumes when the interrupt ends.

Timer

- Timer
    - is a piece of hardware that counts upwards
    - how fast it counts can be configured - 1 per CPU clock or slower

05. Numeral systems and logic gates

Addition:

	1	0	1	1	0	0	1	0	178 dec
+	0	0	1	0	1	0	1	0	42 dec
=	1	1	0	1	1	1	0	0	220 dec

AND

	1	0	1	1	0	0	1	0	178 Dec
&	0	0	1	0	1	0	1	0	42 Dec
=	0	0	1	0	0	0	1	0	34 Dec

Subtraction

	1	0	1	1	0	0	1	0	178 dec
-	1	0	0	0	1	0	0	0	136 dec
=	0	0	1	0	1	0	1	0	42 dec

OR

	1	0	1	1	0	0	1	0	178 Dec
|	0	0	1	0	1	0	1	0	42 Dec
=	1	0	1	1	1	0	1	0	186 Dec

Xor

	1	0	1	1	0	0	1	0	178 dec
^	0	0	1	0	1	0	1	0	42 dec
=	1	0	0	1	1	0	0	0	152 dec

Not

~	1	0	1	1	0	0	1	0	178 dec
=	0	1	0	0	1	1	0	1	77 dec

Binary numbers: left shift and right shift

- left shift
  - 0101 << 1 = 1010
- right shift
  - 1010 >> 1 = 0101

Schematic Symbols

And  $A \cdot B$ or $A \,\&\, B$

Or  $A + B$

Xor  $A \oplus B$

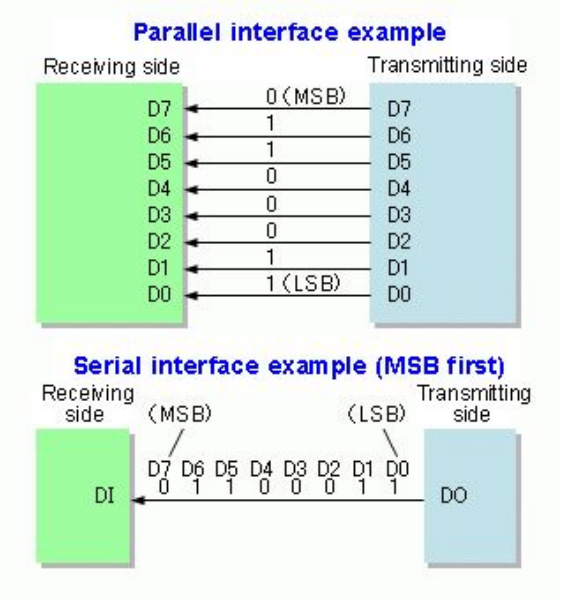
Not  $\overline{A}$ or $\sim A$

06. Serial and Parallel communication

Serial VS Parallel

- Parallel
  - Transmits one bit at a time.
  - A signal called the clock decides when a new batch of bits can be sent.
  - all queues are advanced at the same time
  - Takes up a lots of board space

- Serial
  - Transmits multiple bits at once.
  - A signal called the clock decides when a new bit can be sent.
  - Easier to fit on boards.
  - Uses fewer board traces



Arduino Uno - Communications options

USART - Universal Synchronous Asynchronous Receiver Transceiver

- Many have used this with Serial.println()
- This is the protocol used in old PC COM ports
- Can be used to connect two Arduinos together
- Uses 2 lines minimum for bidirectional
    - Transmit
    - Receive
- Connects 2 devices together, no more

SPI - Serial Peripheral Interface Bus

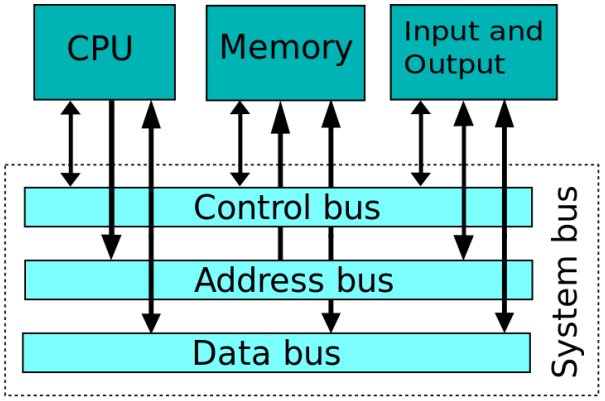
- Used to connect to peripherals as the name implies.
- Often used by sensors (temprature, pressure, etc)
- Uses 3 lines minimum for bidirectional
    - Transmit
    - Receive
    - Clock
- Connects 2 or more devices together

I²C - Inter-Integrated Circuit bus / TWI - Two Wire Interface bus

- Similar in use to SPI
- Only uses 2 lines independen of the number of devices
    - Data
    - Clock

What is a bus?

- A bus is general for term for something that transmits data
- In computing a bus must be able to connect more than two devices (something that only connects 2 is called point-to-point)
- Most buses these days are digital and thus transmit bits
- Buses can be serial or parallel
- An Internal bus connects system components
- An External bus connects peripherals
- Devices on a bus often have an address to tell them appart



Internal bus: System bus

External bus: USB

USART on the Arduino Uno

- The USART on the Uno is connected to both the PC USB port and pins 0 and 1
- The Serial libary is used to send recive data.
- So far you've used it for debugging / logging
- You can also use this for other things, like connecting two unos together
- You will still see the data on the PC side as well, which is good for debugging.
- The two devices need to have common ground
- USART has no clock signal so the speed also has to be known both

10. FreeRTOS

What's an RTOS and why do I need it?

- An RTOS is a small OS for embedded systems
- FreeRTOS is one such "Real Time operating System"
- An RTOS has features similar to a full desktop OS:
    - Process / task switching
    - Mutexes / semaphores
    - IPC: inter process/task communication
    - We'll go through these later in this / the next lecture
- The "real time" part comes with some expectations
    - Lower latency overall (task switching, interrupt handling etc.)
    - Lower OS overhead (no drivers or light weight drivers etc.)
    - Predictability (important for embedded where timing matters)
    - Usually no Memory Management Unit support (bare metal)
- Using an RTOS is useful in larger projects
    - Helps organizing code
    - Separate parts of the program can run in different tasks
    - Tasks can communicate with each other if necessary

Using FreeRTOS on the Arduino

- FreeRTOS is platform dependant so you need a port for your device
- Generally someone has already ported it so you don't have to do it
- Porting involves configuring SoC specific things (memory, registers etc)
- Let's install it:
    - Create a folder called "FreeRTOS" in the lib folder of your PlatformIO project
    - Download the FreeRTOS for Arduino zip from here
    - Extract the zip to the FreeRTOS folder you made
      Make sure the files are directly inside FreeRTOS, no additional top folder
- FreeRTOS will now be built when you build your project
- To use any of the FreeRTOS functions you must include it:
    - #include <Arduino.h>
    - #include <Arduino_FreeRTOS.h>
- Thus: FreeRTOS is more like a library than a traditional OS

Concept: Task

- A task is a piece of code that runs independent of other tasks
- Tasks are similar to threads in desktop OSes
- Tasks have their own stacks (for local variables, like threads)
- Tasks share the same memory (like threads)
- Tasks can use mutexes to protect shared resources
- Tasks can use semaphores to synchronize
- To define a task you create a function:
  void TaskHelloWorld(void *pvParameters) { while(1); }
- The task must have the void return type and take a void* parameter
- The task must never return / exit
- You then create the task in setup() using the xTaskCreate() function:

```
xTaskCreate(
        TaskHelloWorld,             // Function to run as a task
        "Hello World Task",   // Name of the task
        128,                        // Stack size of the task (in words)
        NULL,                       // Parameter to the task, usually NULL
        2,                          // Priority. 0 is the idle task, so we can use 2
for normal
        NULL                        // Task handle, usually NULL
);
```

- You can delay in a task using vTaskDelay(pdMS_TO_TICKS(1000));

Example: Task

```
void TaskBlink(void *pvParameters);  // Forward declare the function so we can use it in
setup()

void setup()
{
  xTaskCreate(TaskBlink, "Blink task", 128, NULL, 2, NULL);
}

void TaskBlink(void *pvParameters)  // This is a task.
{
  // initialize digital LED_BUILTIN on pin 13 as an output.
  pinMode(LED_BUILTIN, OUTPUT);

  for (;;) // A Task shall never return or exit.
  {
    digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
    vTaskDelay(pdMS_TO_TICKS(1000)); // wait for one second
    digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
    vTaskDelay(pdMS_TO_TICKS(1000)); // wait for one second
  }
}
```

Task switching in FreeRTOS

- FreeRTOS for Arduino uses the watchdog (WD) interrupt for task switching
- The default configuration is to switch tasks every 15ms (WDTO_15MS)
- Each task runs for this amount of time (or less if it finishes)
- This is called a time slice
- Tasks are prioritized using the priority you set
- When a task is suspended or the watchdog triggers:
    - FreeRTOS looks for tasks in the ready state
    - Finds the highest priority ready task
    - Runs that task until suspend or the next WD interrupt

Concept: Mutex (mutual exclusion)

- Imagine you have two tasks that access the same variables
- If there's a task switch while data is written the data could become corrupt
    - Half the string is cleared
    - Only the two first variables were written
    - Reading these half-written variables might result in errors
- A mutex can be used to protect access to shared data
- A mutex ensures that only one task access the data
    - Create the mutex with xSemaphoreCreateMutex()
    - Take (lock) the mutex with xSemaphoreTake()
    - Do what you need to do with the data
    - Give/unlock the mutex with xSemaphoreGive()
- If you try to take a mutex that is already taken the task is suspended
- Include the semaphore header to use
    - Mutexes: #include <semphr.h>

Example: Shared struct to hold application data

- We can use structs to share data between tasks
- Just like last year - except the struct now has a mutex
- Tasks lock this mutex when they want to access shared data

```
// Struct holding our data
typedef struct
{
  // Mutex that protects this data struct
  SemaphoreHandle_t mutex;

  // Regular struct members (our data)
  char message[200];
} data_t;
```

Common errors

Stack Overflow
If the stack of the loop() or for any Task overflows, there will be a slow LED blink, with 4 second cycle.

Heap Overflow
If any Task tries to allocate memory and that allocation fails, there will be a fast LED blink, with 100 millisecond cycle.

11. FreeRTOS - semaphores and queues

Semaphores - What are they?

Scenario

- Imagine a task that needs to wait on a button interrupt
- When the interrupt occurs the task is suspended
- When the interrupt finishes the task resumes
- How can the task be told about the interrupt?

Binary semaphore

- Ideal for synchronizing two tasks or a task and an interrupt
- Imagine a queue that is either empty or full (thus binary)
- The task can try to take the semaphore and is blocked if it is empty
- The interrupt can give the semaphore, thus making it full
- The task then resumes and can handle the interrupt

Semaphores cont.

Scenario

- In the previous example the button is clicked quickly
- This will result in the interrupt triggering often
- It might trigger twice without the task resuming
    - We will “lose” an interrupt here because the semaphore is binary
    - There’s no way to signal that more than one interrupt occurred

Counting semaphore

- A counting semaphore can be given / taken more than once
- Thus the imaginary queue can be empty, full or somewhere in between
- If the interrupt occurs twice the semaphore can be taken twice
- This can also be used in tasks to control access to a limited resource
- Imagine having 3 serial ports:
    - Take the semaphore to get a serial port (blocks if none are available)
    - Use the serial port as normal
    - Give the semaphore to free the serial port

New functions - semaphores

xSemaphoreCreate*()

- Creates a semaphore
- SemaphoreHandle_t semaphore = xSemaphoreCreateBinary();
- SemaphoreHandle_t semaphore = xSemaphoreCreateCounting(MAX, CURRENT);

xSemaphoreTake*()

- Block until the semaphore is given elsewhere
- xSemaphoreTake(semaphore, portMAX_DELAY);
- xSemaphoreTakeFromISR(semaphore, portMAX_DELAY);

xSemaphoreGive*()

- Give the semaphore to unblock a task that is trying to take it
- xSemaphoreGive(semaphore, NULL);
- xSemaphoreGiveFromISR(semaphore, NULL);

Queues

- A queue can be used to send a message between tasks and/or interrupts
- A queue message contains actual data, aka variables
- Queues are typically FIFO (first in, first out)
- The message can be any type, e.g. integer or a struct you made
- The queue has a maximum size
- Sending when the queue is full blocks the sending task
- Trying to receive from an empty queue blocks the receiving task

New functions - queues

xQueueCreate(count, size)

Creates a message queue

- Count is the max number of messages, size is the message size
- queue = xQueueCreate(5, sizeof(message_t));

xQueueSend*()

- Sends a message, blocks task if the queue is full
- xQueueSend(queue, &message, portMAX_DELAY);
- xQueueSendFromISR(queue, &message, portMAX_DELAY);

xQueueReceive*()

- Receives a message, blocks task if the queue is empty
- xQueueReceive(queue, &message, portMAX_DELAY);
- xQueueReceiveFromISR(queue, &message, portMAX_DELAY);

12. CPUs in detail

Repetition: Memory available to the CPU

External main memory

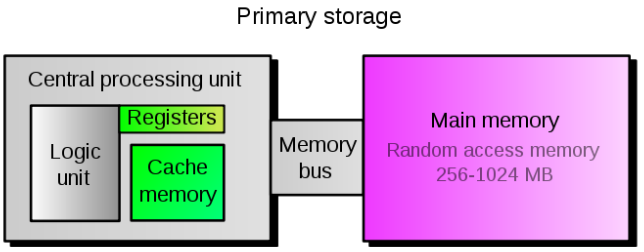
- Connected to the CPU with an external bus (ram chips)
- Typically large, slow and cheap
- Used for less compute intensive data
- Not all SBCs have external memory

Internal main memory

- Can be on-package (right next to the CPU on a separate die)
- Or on-die (manufactured with the CPU)
- Smaller and faster

Cache

- Even smaller and faster
- Internal to the CPU
- Used for important work data
- Can also have multiple levels:
  L1, L2, L3. L1 is the fastest/smallest.



Address and data buses

Address bus

- Tells the ram what address the CPU is interested in reading/writing
- If the address bus i 8bit wide it can address 2^8 addresses, aka 256 bytes
- Most buses use byte addressing, so max ram is usually 2^{width}
- Turns out 2^{32} is 4 GB, max ram of 32bit computers
- The width of the address bus is usually what people mean when they say 32bit etc

Data bus

- Written to by the CPU when performing a RAM write
- Read by the CPU when performing a RAM read
- Can be any width, usually a multiple of 8bit
- Wider is faster
- DDR memory transfers data on both rising and falling clock edge

CPU / general purpose registers

What?

- Temporary storage for data the CPU needs right now
- Used for loading and storing data and doing calculations
- Not to be confused with memory mapped IO registers
- You can think of CPU registers as the fastest memory the CPU has available

Why?

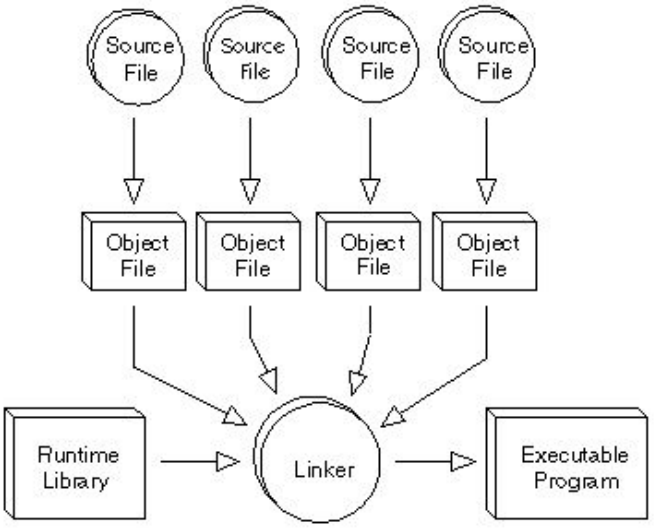
- Data usually needs to be in a CPU register to be used in calculations
- That way instructions can be executed as quickly as possible
- There's no need to wait for main memory which is slow

How?

- The C/C++ compiler handles CPU register usage for you
- If you do, say: int x = 5; int y = 3; int z = x + y;
  the compiler will use CPU registers as needed

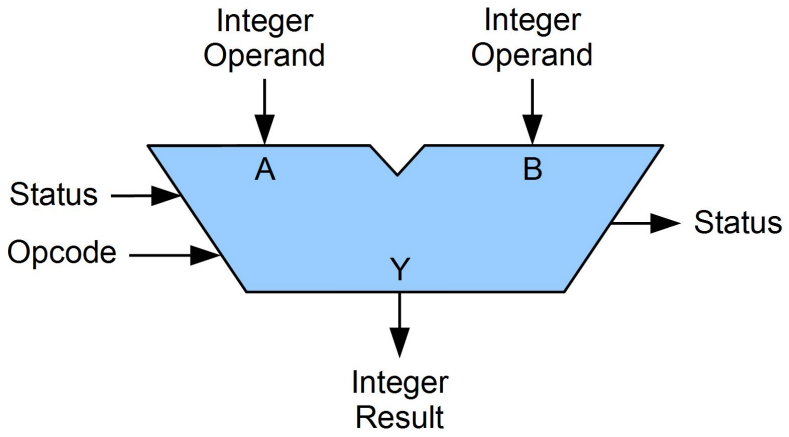
Repetition: How C/C++ compiling works

- The compiler transforms your C/C++ code to binary code, a list of CPU instructions



The instruction set

- The instruction set dictates which commands a CPU knows
- There are many of these instruction sets, like ARM7, x86, x86-64 etc.
- What they all have in common is the concept of an instruction
- An instruction (or opcode) is a single command
    - Load the integer at address 0x100 from memory
    - Add together the two integers you just loaded
- The CPU executes these one by one which in turn runs your code
- Each instruction is decoded so the CPU understands what you want to do



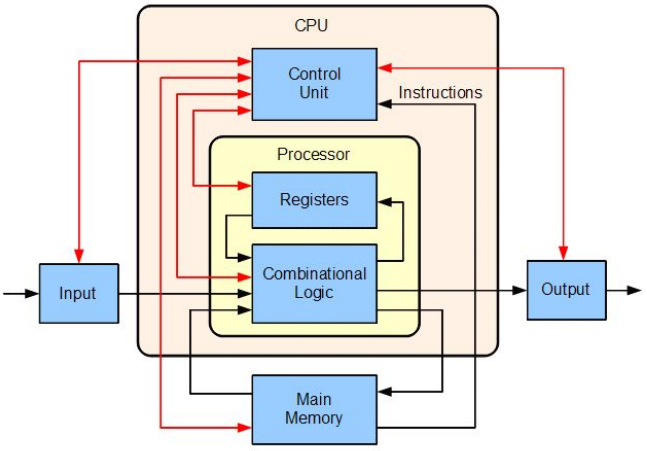
Basic computer example

Control unit

- Fetches instructions
- Decodes instructions
- Fills registers

Logic unit

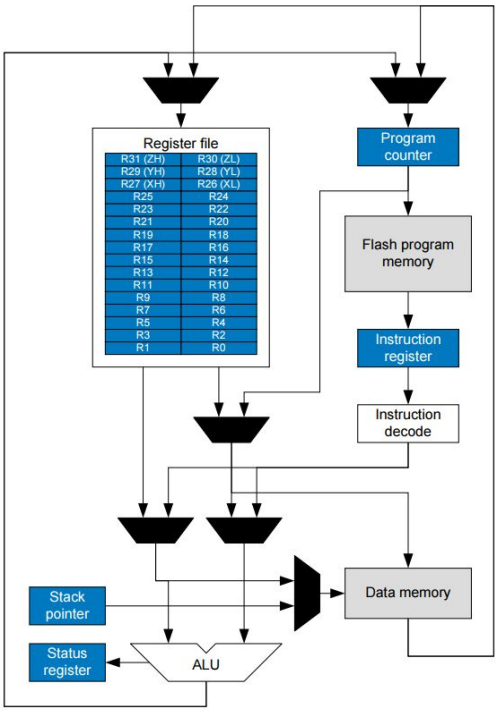
- Reads registers
- Performs math
- Writes to registers



AVR architecture

About

- 32 general purpose registers
- 131 instructions



The program counter

- Keeps track of where we currently are

The stack pointer

- Keeps track of where in the stack we currently are (local memory)

Execution flow

1. The instruction at the current PC is read into the instruction register
2. The instruction is then decoded
3. The ALU (arithmetic and logic unit) calculates the result
4. The result is written back to registers

Assembly programming

- Lower level programming language than C
- Specific to a given instruction set
- Manually write instructions and control CPU register / memory usage
- Rarely used today but sometimes useful for debugging
- Also useful for understanding how stuff actually work
- There are many assembly examples in the ATmega328/P data sheet
- Example:

```
; Define pull-ups and set outputs high
; Define directions for port pins
ldi r16,(1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0)
ldi r17,(1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0)
out PORTB,r16
out DDRB,r17
; Insert nop for synchronization
nop
; Read port pins
in r16,PINB
```