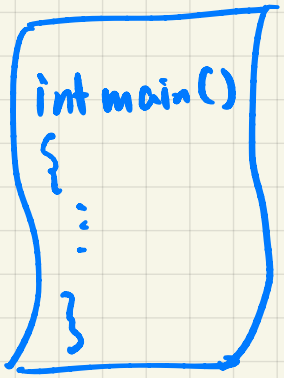
Embedded Systems Programming (mostly in C)

Unit 6 : Internal Working

Compilation

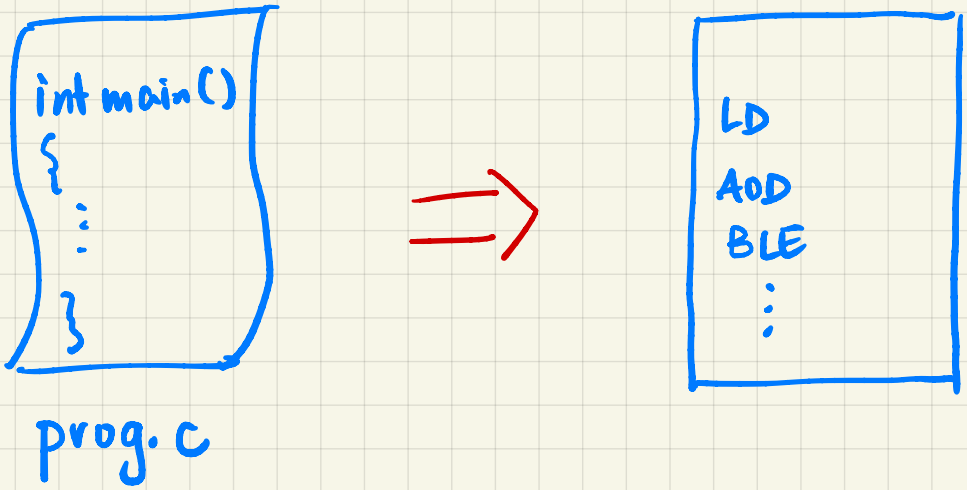
Compiling



prog.c

High level
human readable
and
writable
language

Compiling



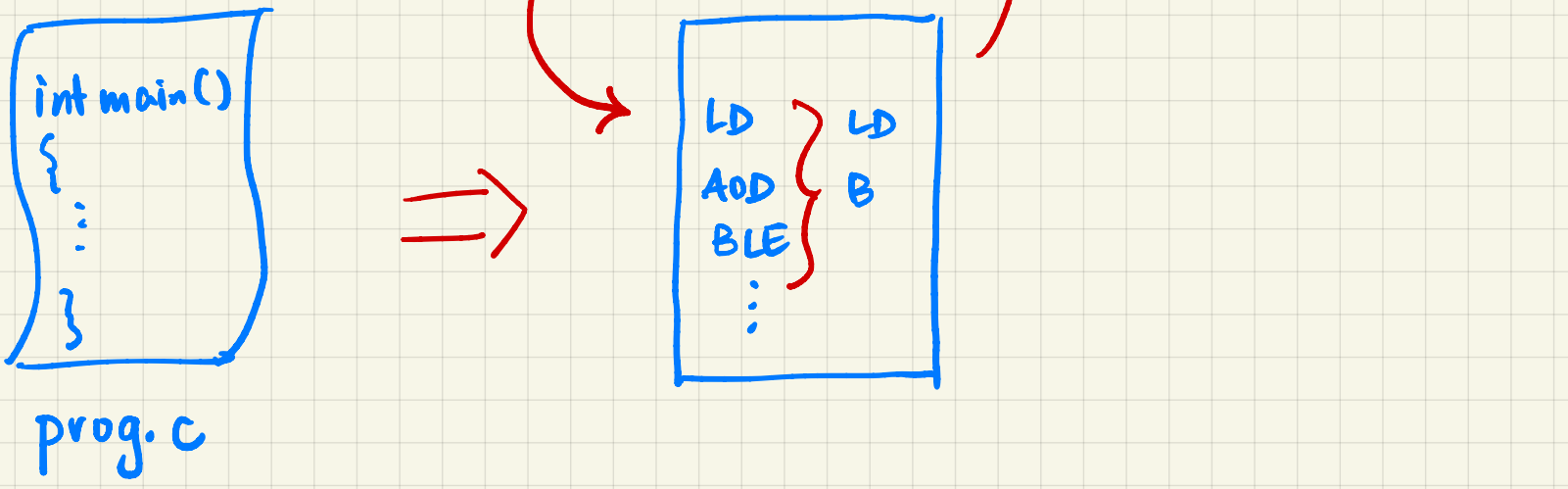
prog.c

High level
human readable
and
writable
language

Intermediate
representation:
readable, but
not meant to
be written/modified

Assembly / IR

Compiling



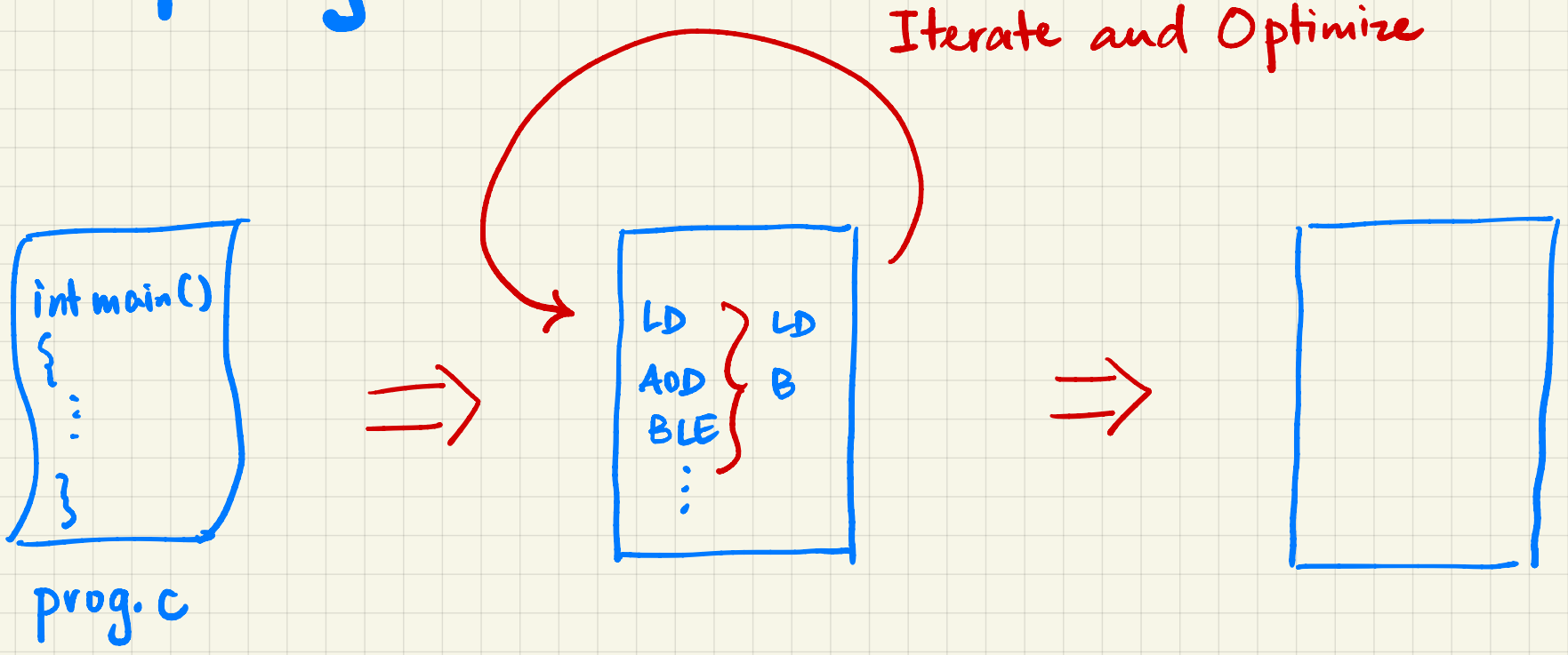
prog.c

High level
human readable
and
writable
language

Intermediate
representation:
readable, but
not meant to
be written/modified

Assembly / IR

Compiling



prog.c

High level
human readable
and
writable
language

Intermediate representation:
readable, but
not meant to
be written/modified

Assembly / IR

Machine Code
Not for humans

Binary

Steps

- Parsing (Syntax) - Break input program to tokens
  - keywords (for, if), function calls, operations ...
- Semantics - Meaning of the program
  - How to implement for, if etc; function calls
  - Intermediate Representation
- Optimization - Iterate on IR: speed, memory
- Generation - Create output binary code.

Assembly Language

Programmers Model - Abstract view of CPU

- Arithmetic and Logic - Add, sub, mul, div, shift, OR, AND ...
- Load / Store - Memory interface, Push/Pop
- Branch - Control flow
- Special - Semaphore (mem), state change, synchronization

ARM Cortex M4 reference manual

Registers

- Normal CPU regs - temporary storage
- Control and Status Registers
  - Modify behaviour
  - Check status / attributes

Translation

```
void testfor()
{
        for (int i=0; i<100; i++);
}
```

Input : C code

```
00000000 <testfor>:
   0:   e52db004        push    {fp}            @ (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e3a03000        mov     r3, #0
  10:   e50b3008        str     r3, [fp, #-8]
  14:   ea000002        b       24 <testfor+0x24>
  18:   e51b3008        ldr     r3, [fp, #-8]
  1c:   e2833001        add     r3, r3, #1
  20:   e50b3008        str     r3, [fp, #-8]
  24:   e51b3008        ldr     r3, [fp, #-8]
  28:   e3530063        cmp     r3, #99 @ 0x63
  2c:   dafffff9        ble     18 <testfor+0x18>
  30:   e1a00000        nop                     @ (mov r0, r0)
  34:   e1a00000        nop                     @ (mov r0, r0)
  38:   e28bd000        add     sp, fp, #0
  3c:   e49db004        pop     {fp}            @ (ldr fp, [sp], #4)
  40:   e12fff1e        bx      lr
```

Output : Assembly

Translation

```
void testfor()
{
        for (int i=0; i<100; i++);
}
```

```
00000000 <testfor>:
   0:   e12fff1e        bx      lr
```

Input: C code

Output: Assembly -O3

Optimized!

Optimization

- Eliminate unnecessary code
- Compute at compile time better than at run time
- Different instruction combinations
  - ISA dependent
- Memory access, caches
  - Architecture / hardware dependent

Optimal, Multi-pass, Multi-objective

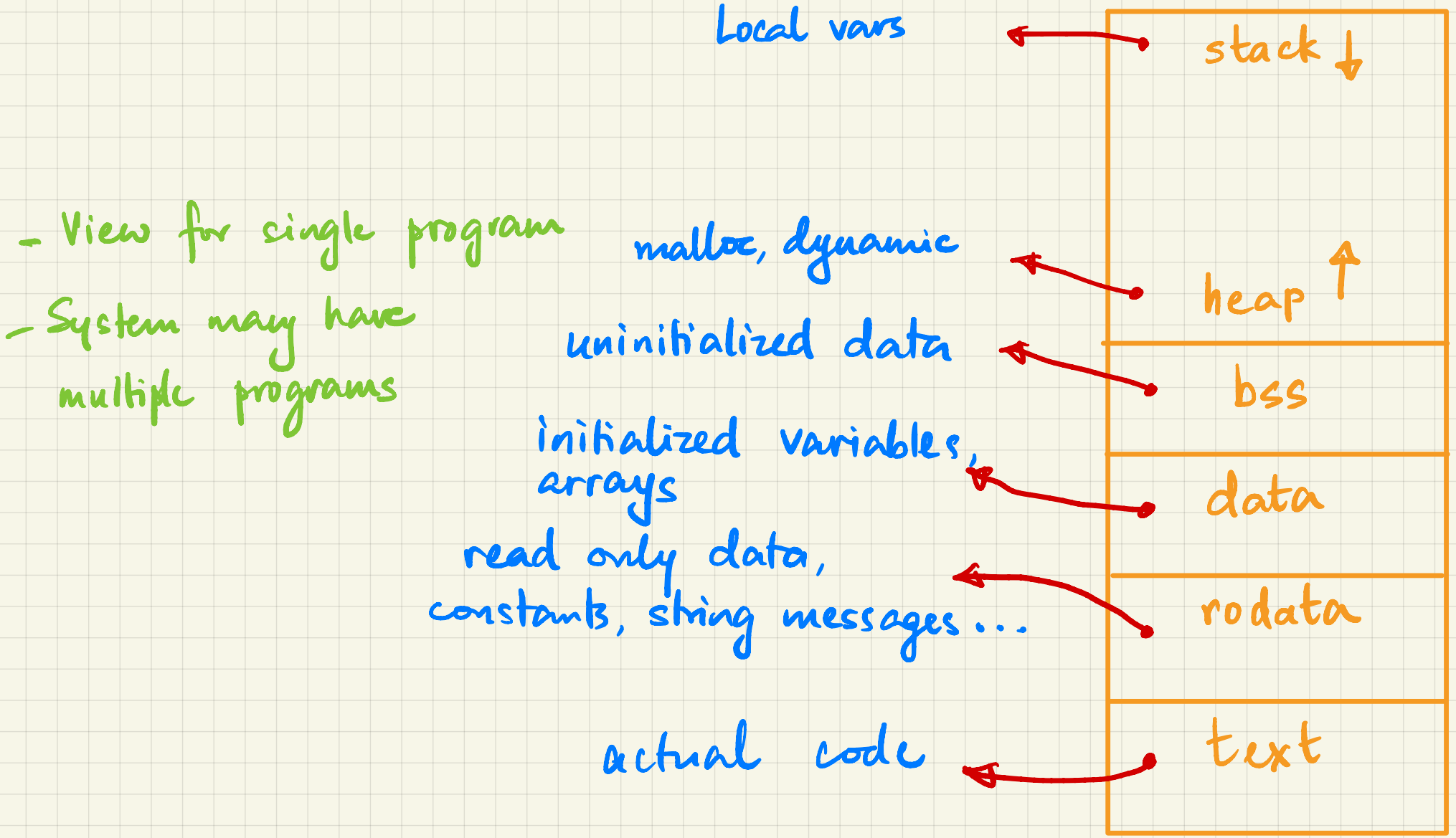
Cross - Compilation

- Host : where you run a program (compiler)
- Target : where you want to run the program
- Different ISAs not a problem
  - code generation

Linking

Memory Layout

- View for single program
- System may have multiple programs



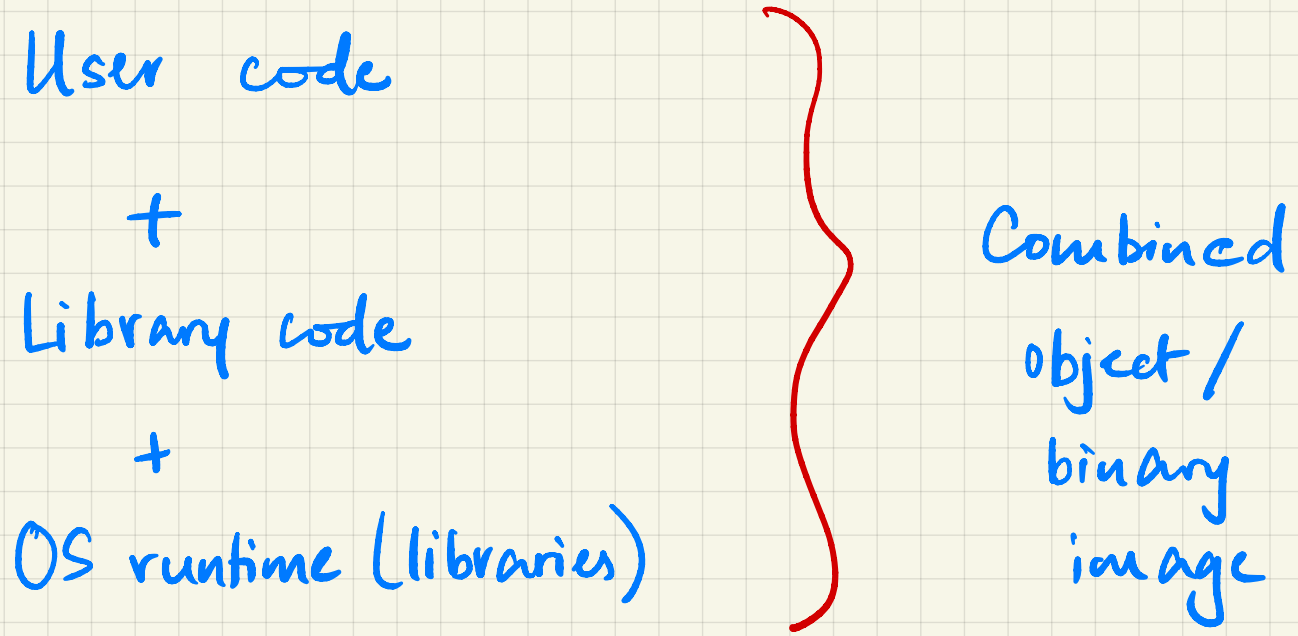
Libraries

- Collections of pre-compiled code

- Standard functions
  - printf, puts, ...
- Specialized functions
  - sin/cos (math), network

Reuse existing code

Linking


User code
+
Library code
+
OS runtime (libraries)
Combined
object /
binary
image

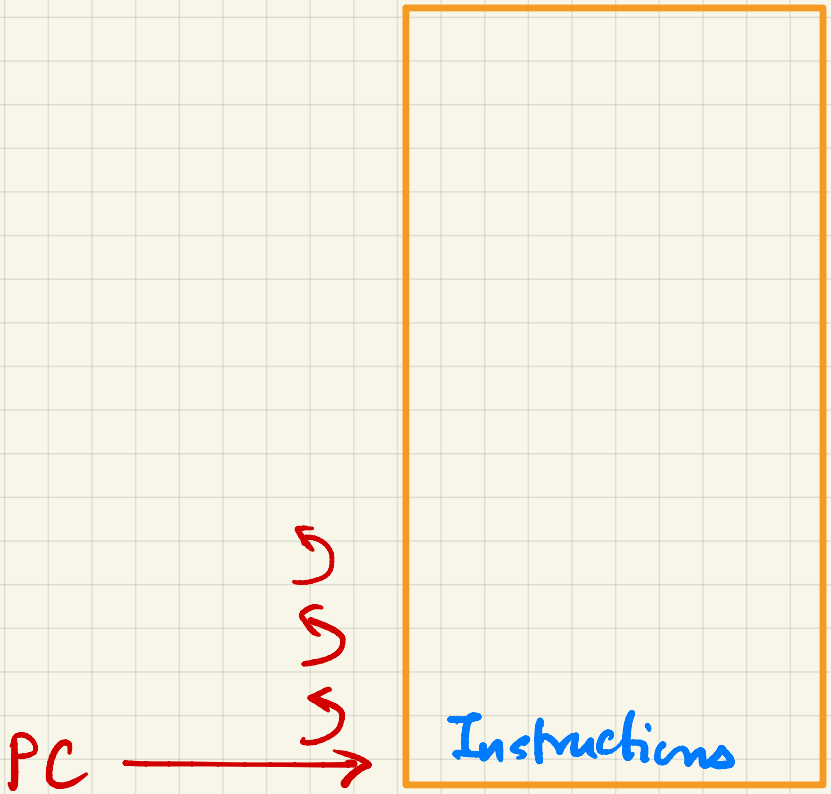
Loading

- ld command usually works as linker + loader
- Run-time requires loading program into memory
  + additional setup

Run time

Boot up

- Reset - electrical initialization of registers



- Program Counter starts at known value
- Automatic increment
- Branch to custom code

System Start

- Reset guarantees known starting address on PC
- Possible modifications through Pin/Jumper settings
- Reset Handler : Code to run after reset
  - can load from external storage

Bare Metal

- Directly load executable code to memory

  ⇒ no run-time linker/loader available

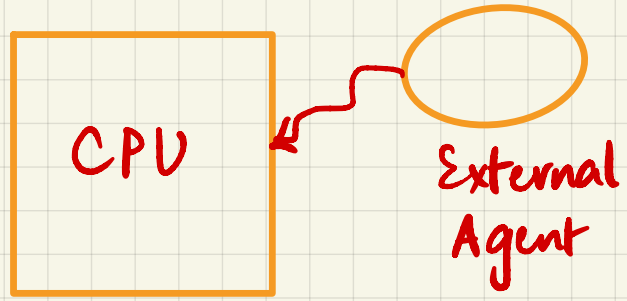
- Loading usually done through
  - debug interface
  - Flash or other known start address

Operating System

- Kernel - First program that runs after start
- Initialize subsystems, peripherals etc
- Loader program can load other code
  - Resolve run-time addresses, pointers
  - Jump to start of code.
  - Wait for exit, return values

Interrupts & Exceptions

Interrupts



- "Interrupt" normal flow of CPU
- Reset, hardware error, timer, communication
- Asynchronous

ISR

Interrupt Service Routine

- On interrupt, halt normal flow (PC ← PC + 1 etc)
- Save state
- Jump to new address
  - Interrupt vector
  - Vector Table for different interrupts

Masking

- Do-Not-Disturb
- Selective or Collective Disabling of Interrupts
  - Use with care
- Non-maskable Interrupts

Exceptions

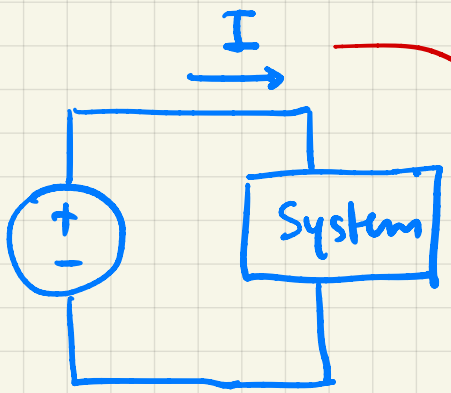
- Internal to CPU
  - Divide by Zero, undefined instruction
- Synchronous
- Exception Handler
  - Software routines
  - Context saving etc software controlled

Priority

- Multiple simultaneous interrupts / exceptions
- Service all? Prioritize? Queue?
- Nesting
  - Reentrant ISRs
    - Resource / state management
- Keep ISRs as short as possible
  - Queue up tasks if needed for further processing

Power Management

Power and Energy



Current drawn by system

- CPU operations
- Peripherals, Timers, Communication

Typically fixed voltage needed

$$P = V \times I$$

$$E = V \times I \times T$$

Power

- Usually directly determined by current
  (Voltage scaling possible in some systems)
- Directly affects heating
  - heat sinks, fans, thermal design

Energy

- Long term impact of operation
- High power x Short Time

  or

  Low power x Long Time

  ?
- Battery Life

Power Modes

- Microcontroller dependent
- Sleep, Standby, Shutdown etc
  - vs Normal operation
  - different levels of power
- Software switchable modes
  +
  Hardware wakeup / interrupts

} Low power / Low energy design.