M2PGi – Internet Of Things

Pr. Olivier Gruber

(olivier.gruber@univ-grenoble-alpes.fr)

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Goal… Bare-metal Discovery...

- Widening your horizon as a developer
  - What you probably know about…

Probably application programming, right?

C, Java, JavaScript, Python...

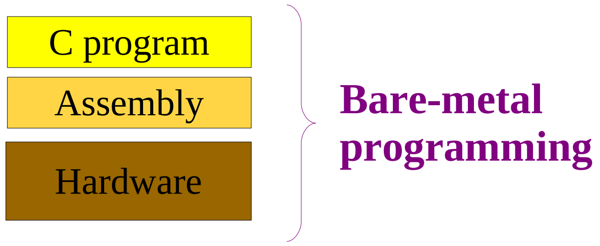
Basic usage of your operating system, right?

File system and Graphical User Interface
Maybe a hint of shell usage/scripting...

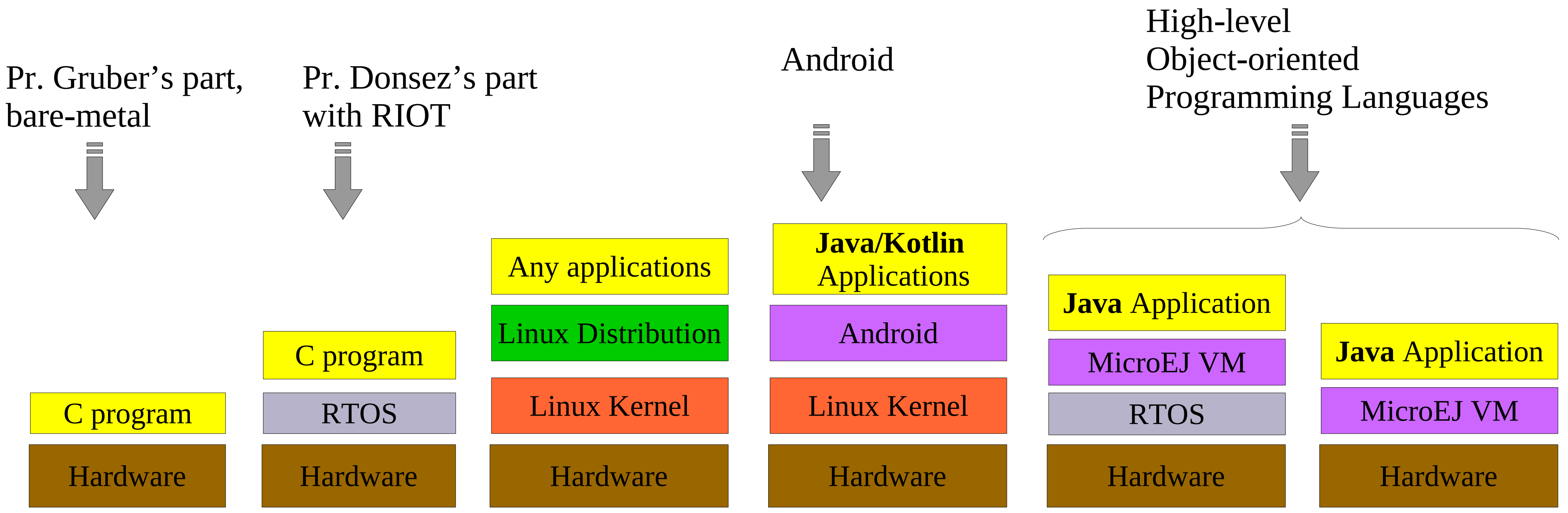
  - What you probably know less about...

How do we program these?

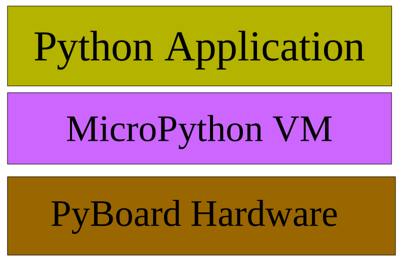
Which tools do we use?

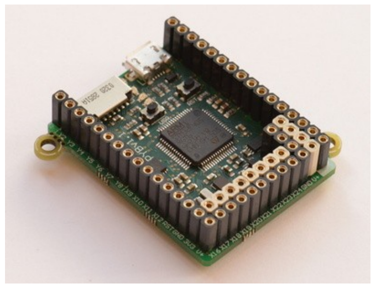


Embedded Development Spectrum



Micro-Python

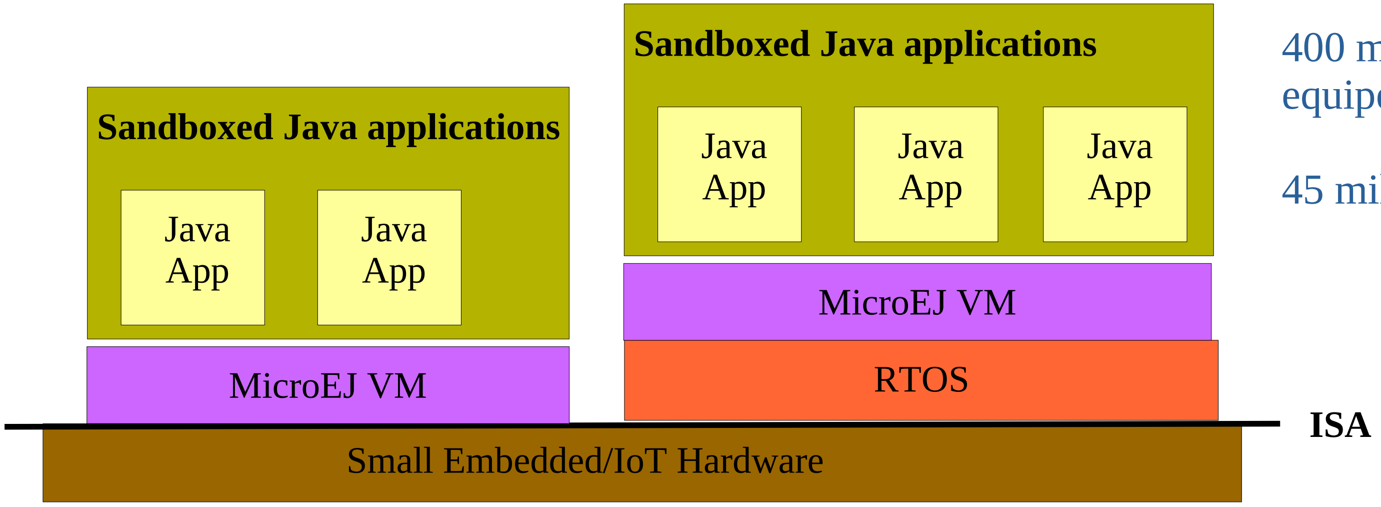




- STM32F405RG microcontroller
  - ***168 MHz Cortex M4 CPU with hardware floating point***
  - **1024 KB flash ROM**
  - **192 KB RAM**
  - Micro USB connector for power and serial communication
  - Micro SD card slot
  - GPIOs

French Tech – MicroEJ[(1)]

- Micro Embedded Java
  - Java for small and smart embedded devices
  - Similar virtualization technology as Android but *much tighter* implementation
    - Android: $15 processor, 32MB of memory
    - MicroEJ: $1 processor, 128KB of memory
  - Google Cloud IoT Partners
    - IoT solutions that leverage Google’s secure, global, and scalable infrastructure


Sandboxed Java applications
Java App
Java App
MicroEJ VM
Sandboxed Java applications
Java App
Java App
Java App
MicroEJ VM
RTOS
ISA
Small Embedded/IoT Hardware

400 million devices equiped with MicroEJ

45 million euros in R&D

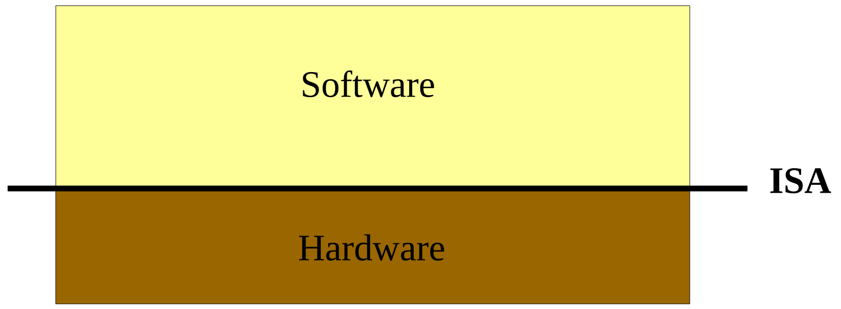
(1) https://www.microej.com/

This Part Pedagogy

- Team learning, individual work
  - Hands-on learning and coding bare-metal software
  - Your own **work-log: *tracking what you do and what you learned***
  - This work-log document is <u>first and foremost for you own records</u>
- The destination is not the goal, the goal is <u>the learning along that path</u>
  - Be curious… expand your skills and know-how
  - Discuss what you learned/understood
  - Explain to others – Ask questions
  - ***WRITE DOWN WHAT YOU HAVE LEARNED / WHAT YOU DID***
- Part-I Evaluation
  - No exam – but <u>weekly progress checkpoints</u>
  - <u>***Every week***:</u> you will surrender your work (***work-log*** and your ***code***)
  - Your <u>weekly involvement</u> is the ***largest part of your final grade***

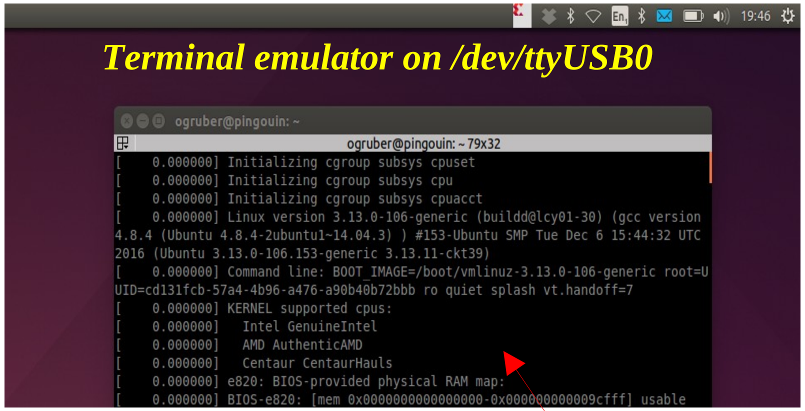
Bare-metal Development

- Bare-metal Software Basics
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set (user and privileged)
  - Defines other privileged concepts (interrupts, traps, page tables, ...)
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools



Bare Metal – Development Environment

- Using a real board
  - Relies on using a USB-Serial cable
  - For both JTAG and a console (/dev/ttyUSB0)
- JTAG through /dev/ttyUSB0
  - JTAG to upload the firmware
  - JTAG for hardware and software debugging
- Using a terminal emulator on your laptop
  - Terminal emulator like minicom or kermit
  - ***Serial line for a console*** (a command-line interface)
    - Send characters for printing on the screen
    - Receive characters from the keyboard



```
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.13.0-106-generic (buildd@lcy01-30) (gcc version
4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3) ) #153-Ubuntu SMP Tue Dec 6 15:44:32 UTC
2016 (Ubuntu 3.13.0-106.153-generic 3.13.11-ckt39)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.13.0-106-generic root=U
UID=cd131fcb-57a4-4b96-a476-a90b40b72bbb ro quiet splash vt.handoff=7
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
[    0.000000]   AMD AuthenticAMD
[    0.000000]   Centaur CentaurHauls
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009cfff] usable
```

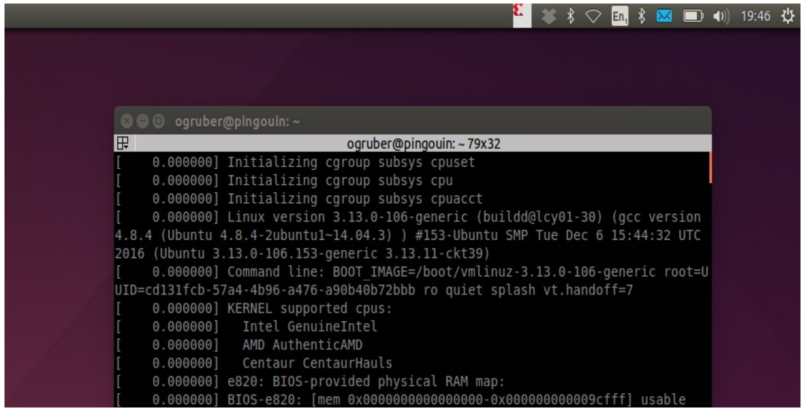
Terminal

Bare-metal Software
Real Machine

USB – Serial Cable – RS 232
JTAG debugging
Console over serial Line

Bare Metal – Development Environment

- Using an emulator – QEMU
  - A regular Linux process
  - Emulates a machine for your bare-metal software
  - Direct support for GDB debugging
  - Serial line for a command-line interface



ogruber@pingouin: ~

ogruber@pingouin: ~ 79x32

```
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.13.0-106-generic (buildd@lcy01-30) (gcc version
4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3) ) #153-Ubuntu SMP Tue Dec 6 15:44:32 UTC
2016 (Ubuntu 3.13.0-106.153-generic 3.13.11-ckt39)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.13.0-106-generic root=U
UID=cd131fcb-57a4-4b96-a476-a90b40b72bbb ro quiet splash vt.handoff=7
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
[    0.000000]   AMD AuthenticAMD
[    0.000000]   Centaur CentaurHauls
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009cfff] usable
```

"The matrix is a prison that you cannot see, taste, or smell."
Morpheus

Board

Serial Line

Terminal

Terminal

For you: an all-software-development experience in the confort of your chair!

~~USB – Serial Cable – RS 232~~

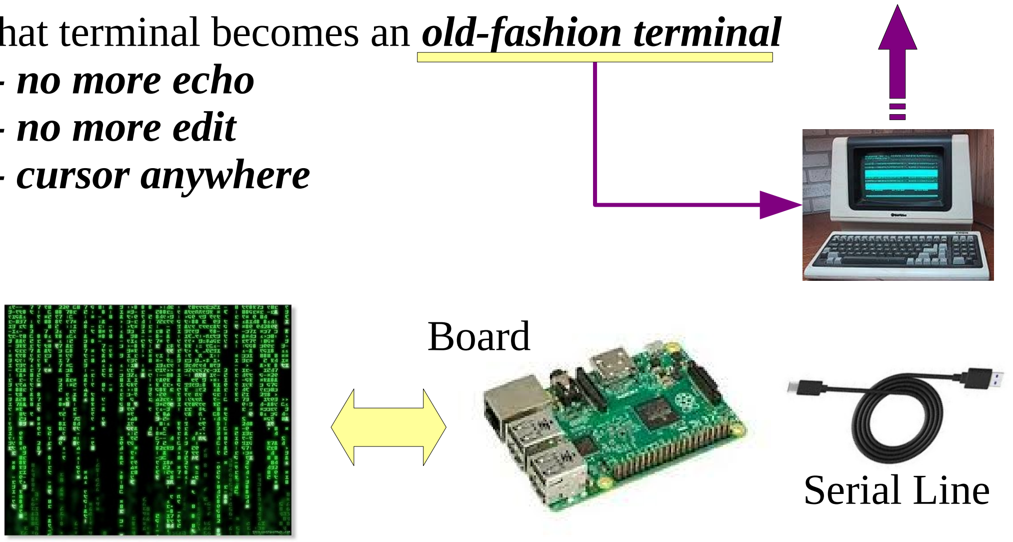
© Pr. Olivier Gruber

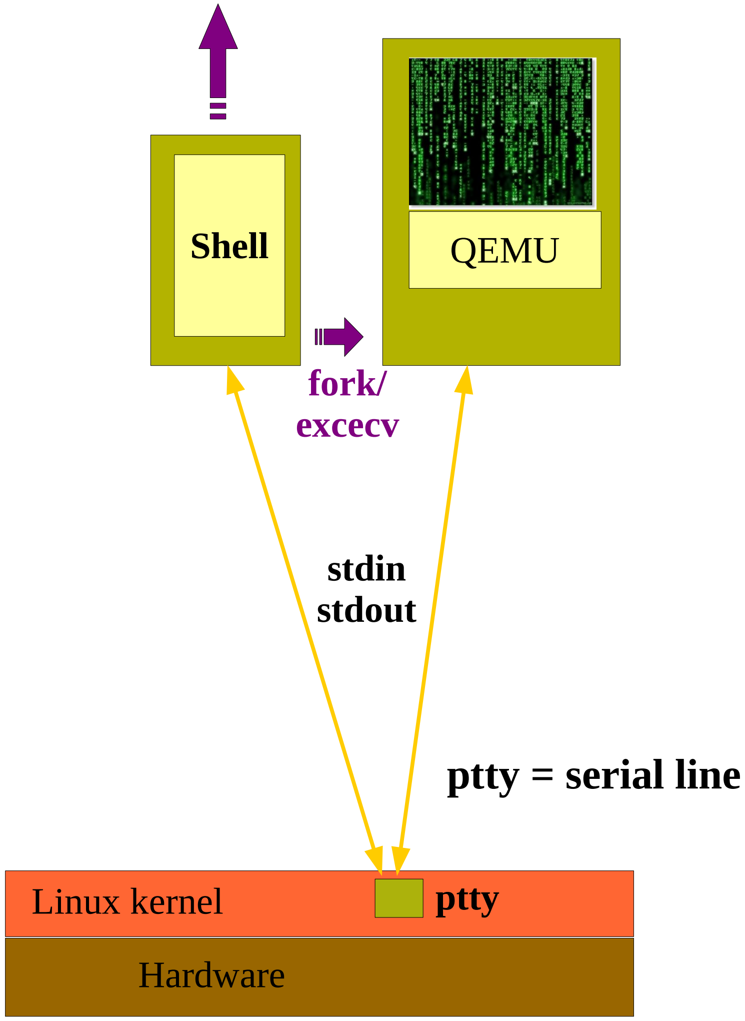
Emulation with QEMU – Board and Terminal

In one terminal:
- *cross-compile* and ***link*** your kernel
- ***launch QEMU***, it will load your kernel

```
$ arm-none-eabi-gcc -g -o kernel.elf *.o
$ qemu-system-arm … -device loader,file=kernel.elf
```

That terminal becomes an ***old-fashion terminal***
- ***no more echo***
- ***no more edit***
- ***cursor anywhere***





Sending characters
through the serial line
will print them on the terminal

Received characters
will be the one typed
on the keyboard

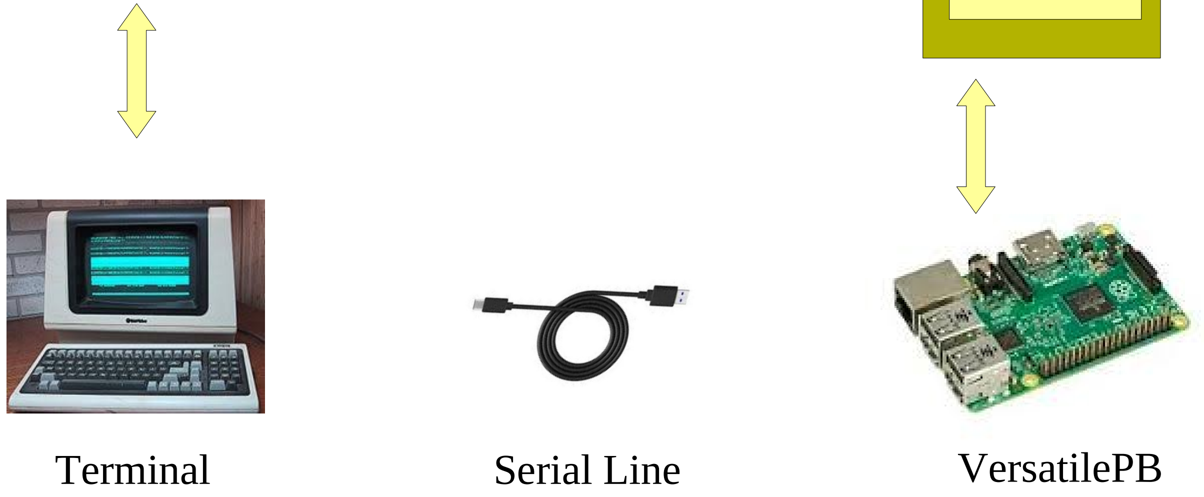
Asking QEMU to Stop the Illusion!

Open the QEMU console and quit the emulation

Crtl-A c

```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "32K"
   -nographic -serial mon:stdio -device loader,file=kernel.elf
QEMU 8.2.2 monitor - type 'help' for more information
...
(qemu) quit
$
```

Or simply use **Ctrl-A x**



Debugging with QEMU

In one terminal:
- cross-compile and link your kernel
- launch QEMU to load your kernel

```
$ arm-none-eabi-gcc -g -o kernel.elf *.o
$ qemu-system-arm … -gdb tcp::1234 -S
```

In another terminal:
- launch your ARM debugger
- with the ELF file of your kernel
- but attach to a remote target

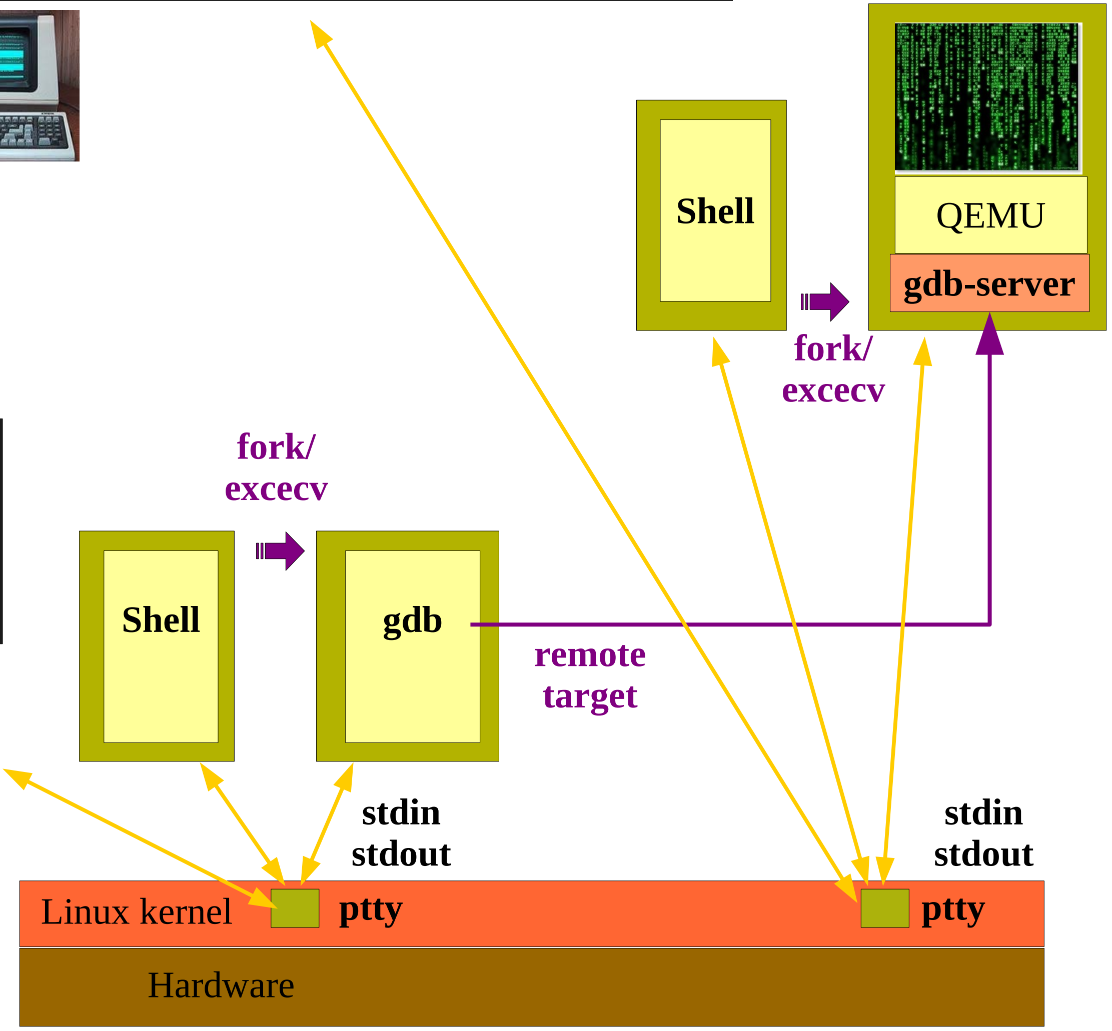
```
$ arm-none-eabi-gdb kernel.elf
…
(gdb) target remote localhost:1234
```

Nota Bene:
you should use the short notation

tar rem :1234

instead of

target remote localhost:1234



Debugging with QEMU

```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "16K"
   -nographic -serial mon:stdio -device loader,file=kernel.elf  -gdb tcp::1234 -S
```

```
$ gdb-multiarch kernel.elf

gdb-multiarch kernel.elf
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.

(gdb) tar rem :1234
Remote debugging using :1234
?? () at exception.s:31
31          ldr pc, reset_handler_addr
(gdb) print /x $r15
$1 = 0x0
(gdb)
```

Where is the execution stopped at?
→ At address 0x0000-0000
→ Register **r15** is the "program counter"

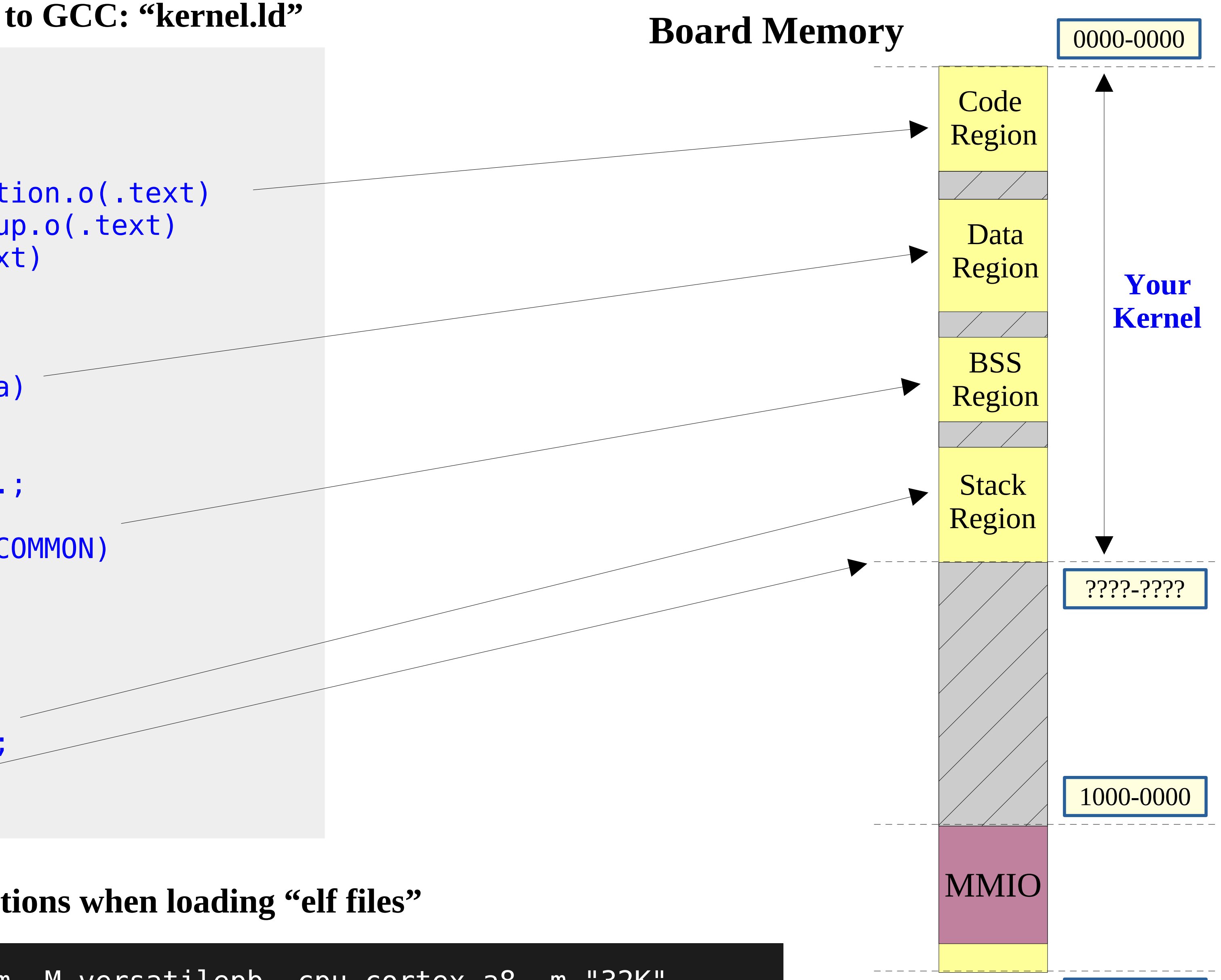
Why is it stopped there?
What code is there to execute?
How did we build that code?

Linker Script – Code / Data Memory Layout

Linker Script given to GCC: "kernel.ld"

```
SECTIONS
{
  . = 0x00;
  .text : {
     build/exception.o(.text)
     build/startup.o(.text)
     build/*(.text)
  }
  . = ALIGN(4);
  .data : {
    build/*(.data)
  }
  . = ALIGN(4);
   _bss_start = .;
  .bss . : {
   build/*(.bss COMMON)
  }
  . = ALIGN(4);
  _bss_end = .;

  . = ALIGN(8);
  . = . + 0x1000;
  stack_top = .;
}
```

Board Memory



QEMU: respect sections when loading "elf files"

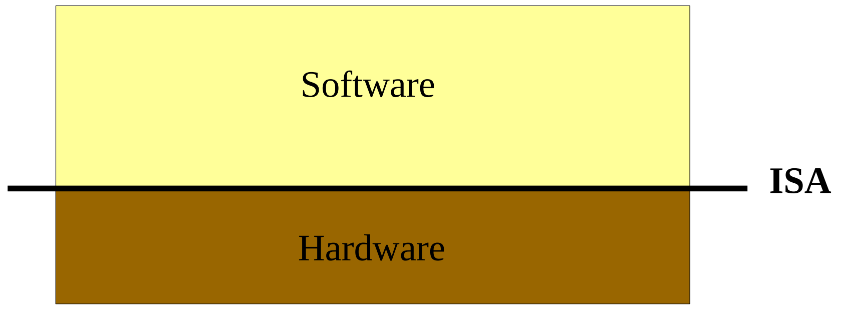
```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "32K"
  -nographic -serial mon:stdio -device loader,file=kernel.elf
```

Cross-compilation & QEMU-based Execution

```
MACHINE=versatilepb
CPU=cortex-a8
MEMSIZE_IN_KB=16

QEMU=qemu-system-arm
QEMU_ARGS= -M $(MACHINE) -cpu $(CPU) -M $(MEMSIZE_IN_KB)K -nographic -serial mon:stdio

CFLAGS= -g -DMEMORY="($(MEMSIZE_IN_KB)*1024)"
ASFLAGS= -g
CFLAGS+= -mcpu=$(CPU) -c -nostdlib -ffreestanding
LDFLAGS= -g -mcpu=$(CPU) -T kernel.ld -nostdlib -static

# Object files to build and link together
#-------------------------------------------------
OBJS= startup.o main.o exception.o

# Compilation Rules
#-------------------------------------------------
%.o: %.c
     arm-none-eabi-gcc $(CFLAGS) -o $@ $<
%.o: %.s
     arm-none-eabi-as $(ASFLAGS) -o $@ $<

# Build and link all, producing an ELF and binary
#-------------------------------------------------
all: $(OBJS)
     arm-none-eabi-ld $(LDFLAGS) $^ -o kernel.elf
     arm-none-eabi-objcopy -O binary kernel.elf kernel.bin

run: all
  $(QEMU) $(QEMU_ARGS) -device loader,file=kernel.elf

debug: all
  $(QEMU) $(QEMU_ARGS) -device loader,file=kernel.elf -gdb tcp::1234 -S
```

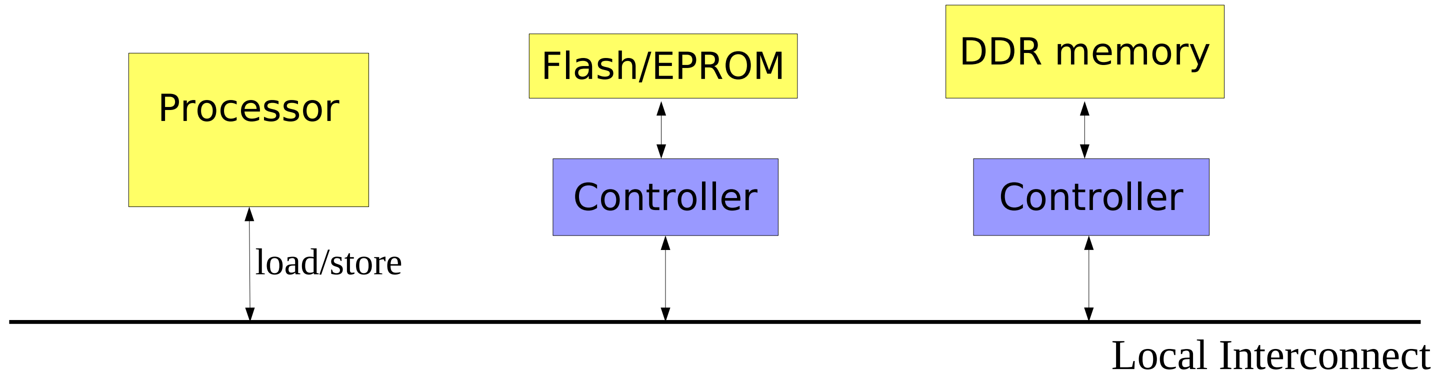
Bare-metal Development Basics

- Bare-metal Software
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set (user and privileged)
  - Defines other privileged concepts (page tables, traps, interrupts, ...)
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools
- Using Integrated Development Environments (IDEs)

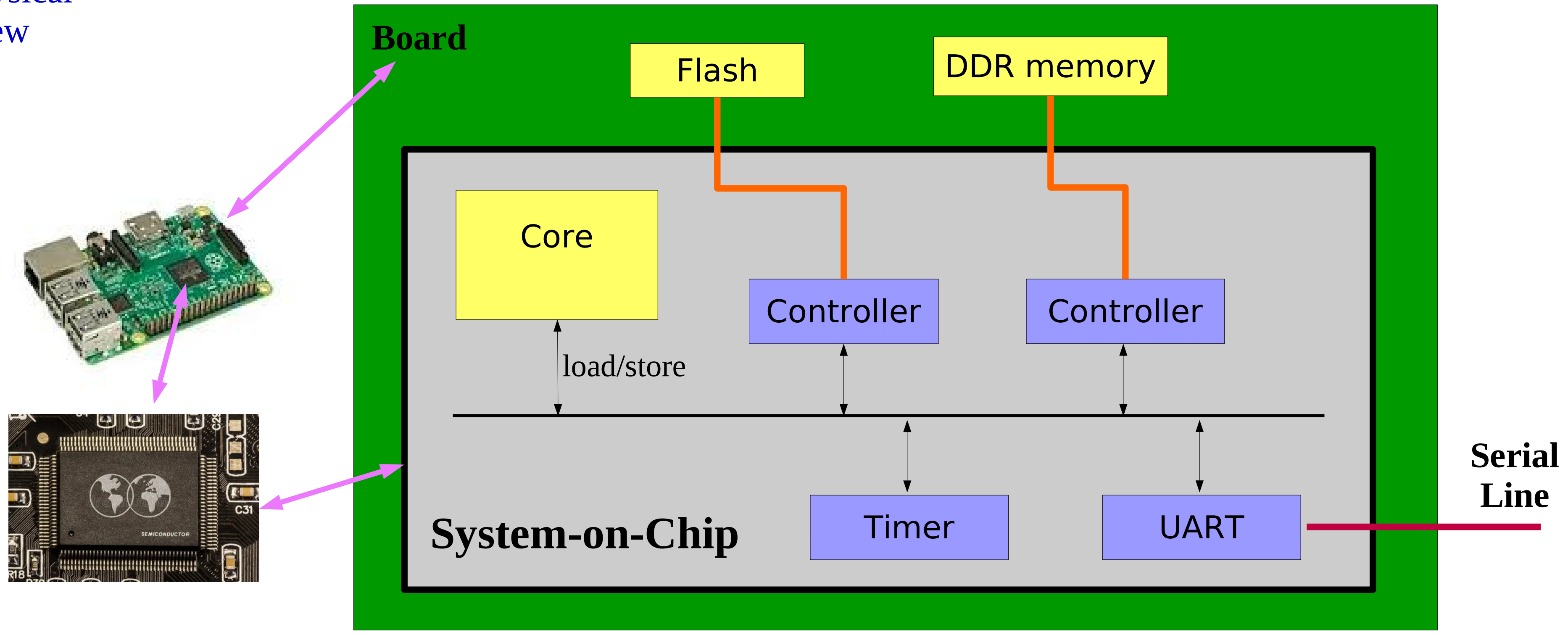


Hardware – Basics

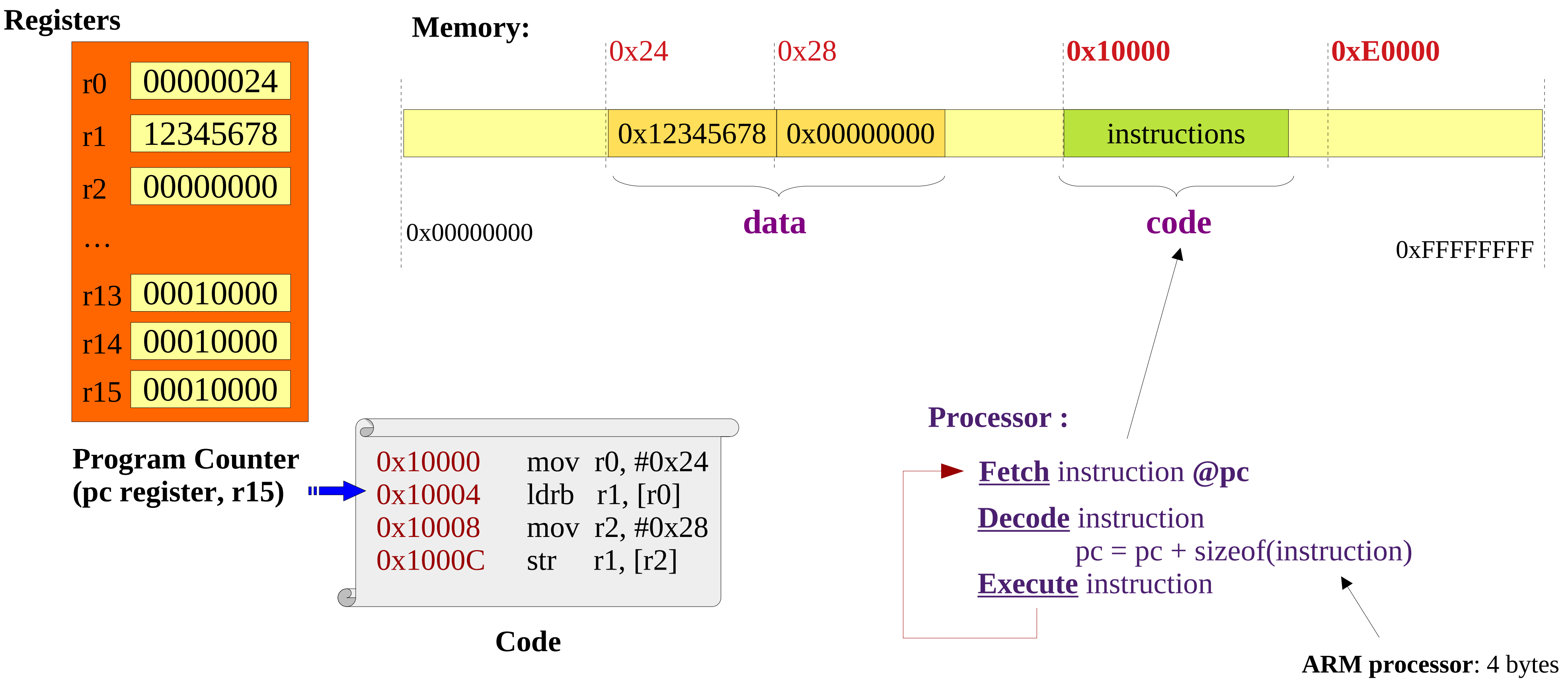
Conceptual View



Physical View



Reminder – Processor & Memory



```
0x10000   mov  r0, #0x24
0x10004   ldrb  r1, [r0]
0x10008   mov  r2, #0x28
0x1000C   str   r1, [r2]
```

Processor :

- **Fetch** instruction **@pc**
- **Decode** instruction
  - pc = pc + sizeof(instruction)
- **Execute** instruction

Processor

Only two operations:
- value ← **load**(address)
- **store**(address,value)

both for data and code...

Memory

Example: Zynq-7000 – Boot Sequence

Processor Reset
→ pc=0x0 000-0000

With:
OCM is mapped low

Boot Steps:

1. initialize DDR
2. copy boot code above 0x0000-FFFF
3. jump there
4. …

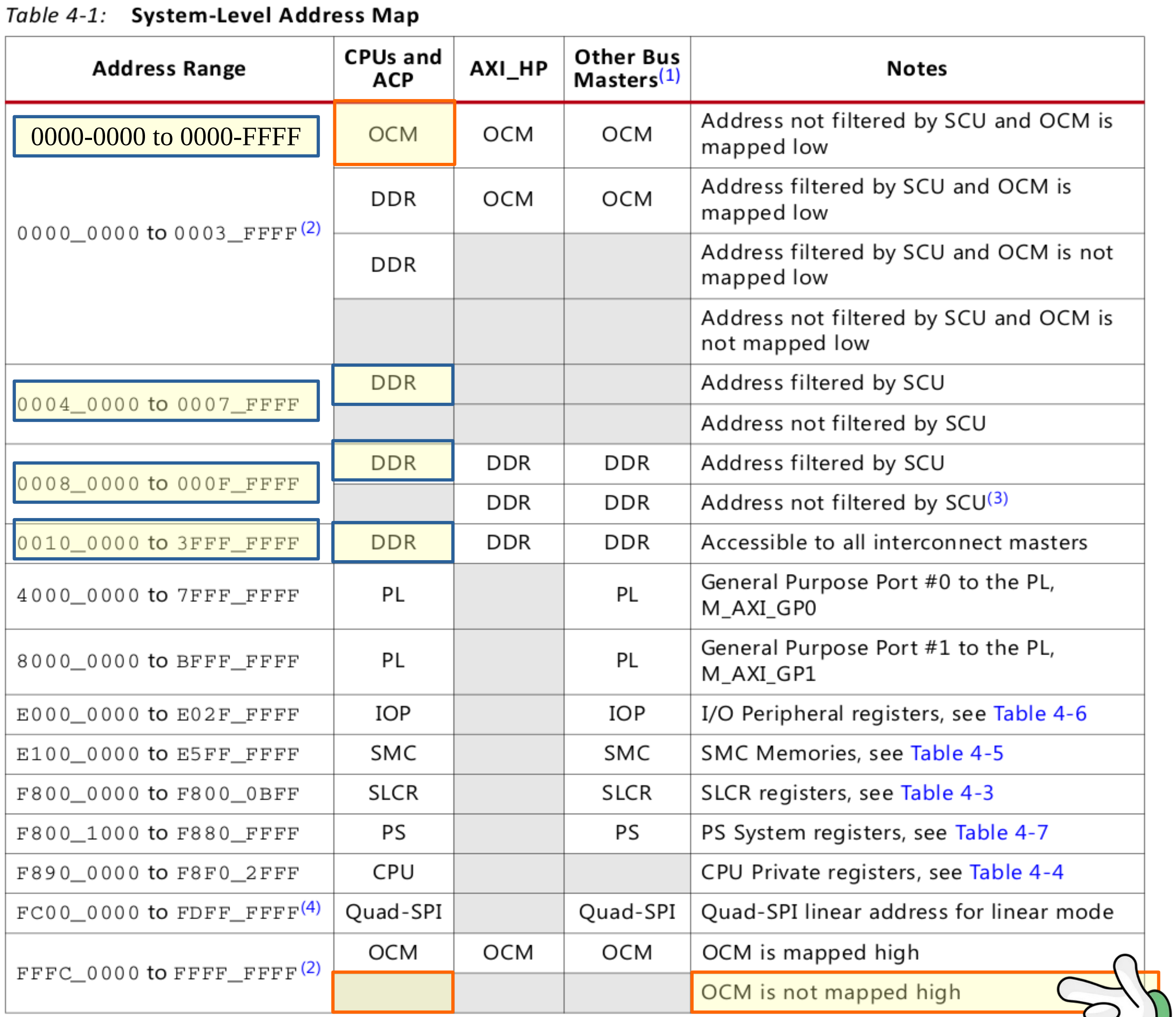


Table 4-1: **System-Level Address Map**

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000-0000 to 0000-FFFF	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
0000_0000 to 0003_FFFF[2]	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

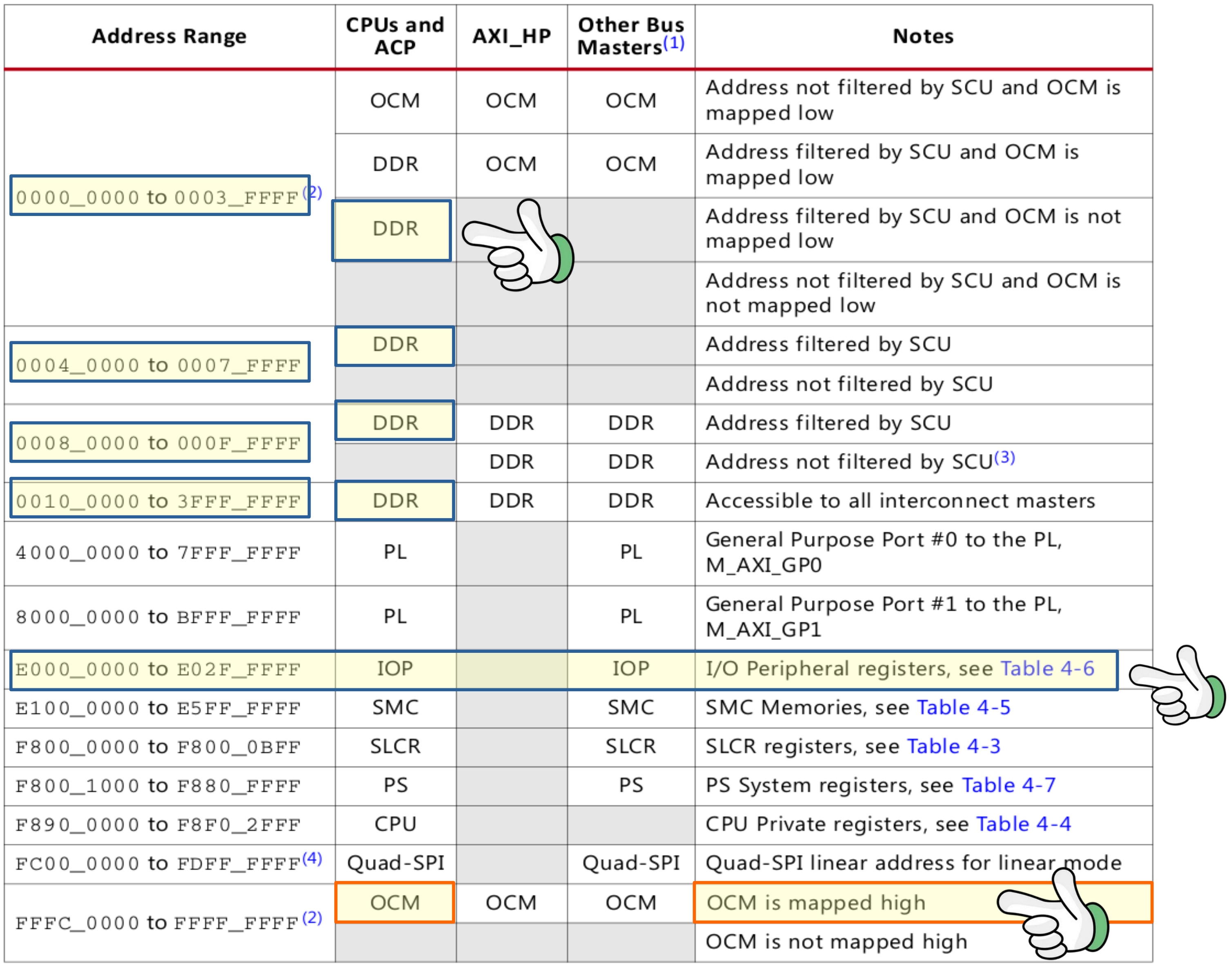
Technical Reference Manual: Zynq-7000 All Programmable SoC

Example: Zynq-7000 – Boot Sequence

Memory Map:
Normal memory (DDR)
from 0x0000-00000
to 0x3FFF-FFFF

Boot Steps:
1. initialize DDR
2. copy boot code
   above 0x0000-FFFF
3. jump there
4. ***map OCM high***
5. initialize boot device,
   like SD-card reader
6. ***Load more code/data…***

Table 4-1: **System-Level Address Map**



Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

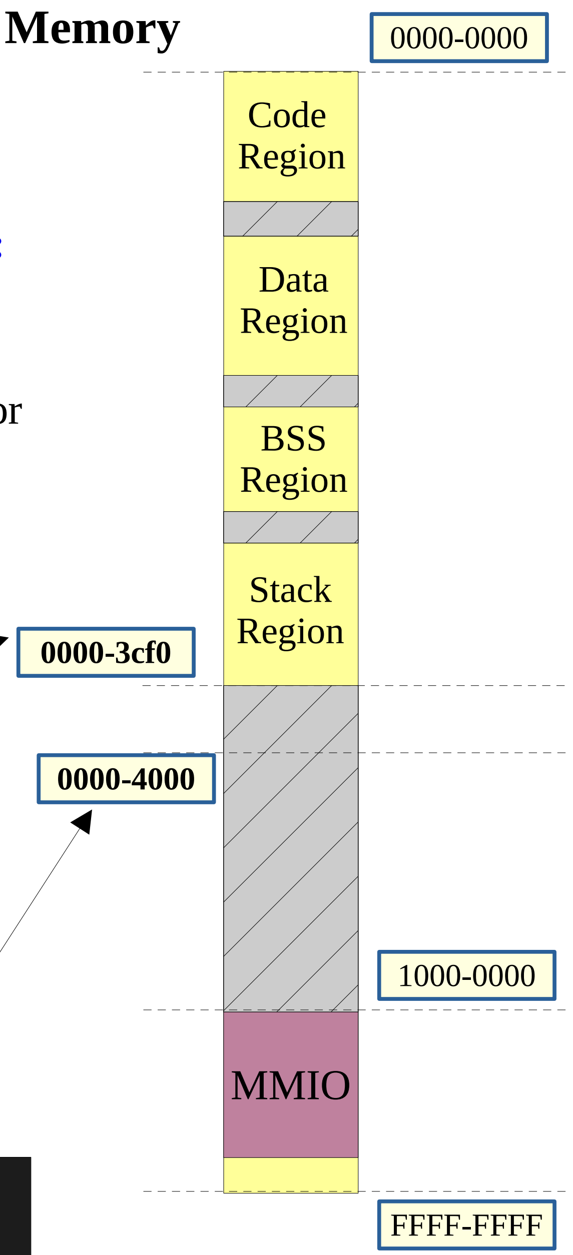
Technical Reference Manual: Zynq-7000 All Programmable SoC

WARNING

```
SECTIONS
{
  . = 0x00;
  .text : {
     build/exception.o(.text)
     build/startup.o(.text)
     build/*(.text)
  }
  . = ALIGN(4);
  .data : {
    build/*(.data)
  }
  . = ALIGN(4);
   _bss_start = .;
  .bss . : {
   build/*(.bss COMMON)
  }
  . = ALIGN(4);
  _bss_end = .;

  . = ALIGN(8);
  . = . + 0x1000;
  stack_top = .;
}
```

Everything must be compatible:

- code and data size
- where sections are loaded
- where sections are compiled for
- total amount of memory
- board memory map



QEMU: respect sections when loading "elf files"

```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "16K"
  -nographic -serial mon:stdio -device loader,file=kernel.elf
```

Cortex-A8 Exception Vector & Code (“exception.s”)

```
// Hardware Exception Vector (loaded 0x0000-0000, see linker script)
0x00     ldr pc, reset_handler_addr
0x04     ldr pc, undef_handler_addr
0x08     ldr pc, swi_handler_addr
0x0C     ldr pc, prefetch_abort_handler_addr
0x10     ldr pc, data_abort_handler_addr
0x14     ldr pc, unused_addr
0x18     ldr pc, irq_handler_addr
0x1C     ldr pc, fiq_handler_addr

0x20  reset_handler_addr: .word _reset_handler
0x24  undef_handler_addr: .word _undef_handler
0x28  swi_handler_addr: .word _swi_handler
0x2C  prefetch_abort_handler_addr: .word _prefetch_abort_handler
0x30  data_abort_handler_addr: .word _data_abort_handler
0x34  not_used_addr: .word _not_used_handler
0x38  irq_handler_addr: .word _isr_handler
0x3C  fiq_handler_addr: .word _fiq_handler

0x40  _isr_handler:                // unexpected interrupt
0x44     b _isr_handler
0x48  _undef_handler:              // undefined instruction
0x4C     b _unused_handler
0x50  _fiq_handler:                // unexpected fast interrupt
0x54     b _fiq_handler
0x58  _undef_handler:              // unexpected undefined instruction
0x5C     b _undef_handler
0x60  _swi_handler:                // unexpected software interrupt
0x64     b _swi_handler
0x68  _prefetch_abort_handler:     // unexpected prefetch-abort
0x6C     b _prefetch_abort_handler
0x70  _data_abort_handler:         // unexpected abort
0x74          b _data_abort_handler

_reset_handler:
     ...  // unexpected abort trap
```

QEMU starts the execution at 0x0000-0000...

It is the “reset exception”…

ldr pc, reset_handler_addr
↔ ldr pc, [pc+0x18]

because pipeline of depth two, so two instructions, so 0x08 bytes.

Cortex-A8 – Assembly Boot Code (“startup.s”)

```
.global _reset_handler
_reset_handler:

   /*
    * Set the core in the SYS_MODE, with all interrupts disabled.
    */
   msr     cpsr_c,#(CPSR_SYS_MODE | CPSR_IRQ_FLAG | CPSR_FIQ_FLAG)

   /* set the C stack pointer for the SYS_MODE */
   ldr     sp,=stack_top

   /*
    * Clear out the bss section, located from _bss_start to _bss_end.
    * This is a C convention, the GNU GCC compiler will group
    * all global variables that need to be zeroed on startup
    * in the bss section of the ELF.
    */
   ldr     r4, =_bss_start
   ldr     r9, =_bss_end
   mov     r5, #0
1:
   stmia   r4!, {r5}
   cmp     r4, r9
   blo     1b

   /*
    * Set the GCC frame-pointer to null (marks the begin of the stack)
    * and then upcall the C entry function  _start(void)
    */
   eor r11, r11, r11
   ldr r3,=_start
   blx r3
halt:
    b halt  // in case the function "_start" returns...
```

Setup the processor mode,
and disable all interrupts
Then set the stack pointer.

Then clear the "bss data"

Done with assembly,
for now, upcalling C code...

Cortex-A8 Exception Vector & Code (“exception.s”)

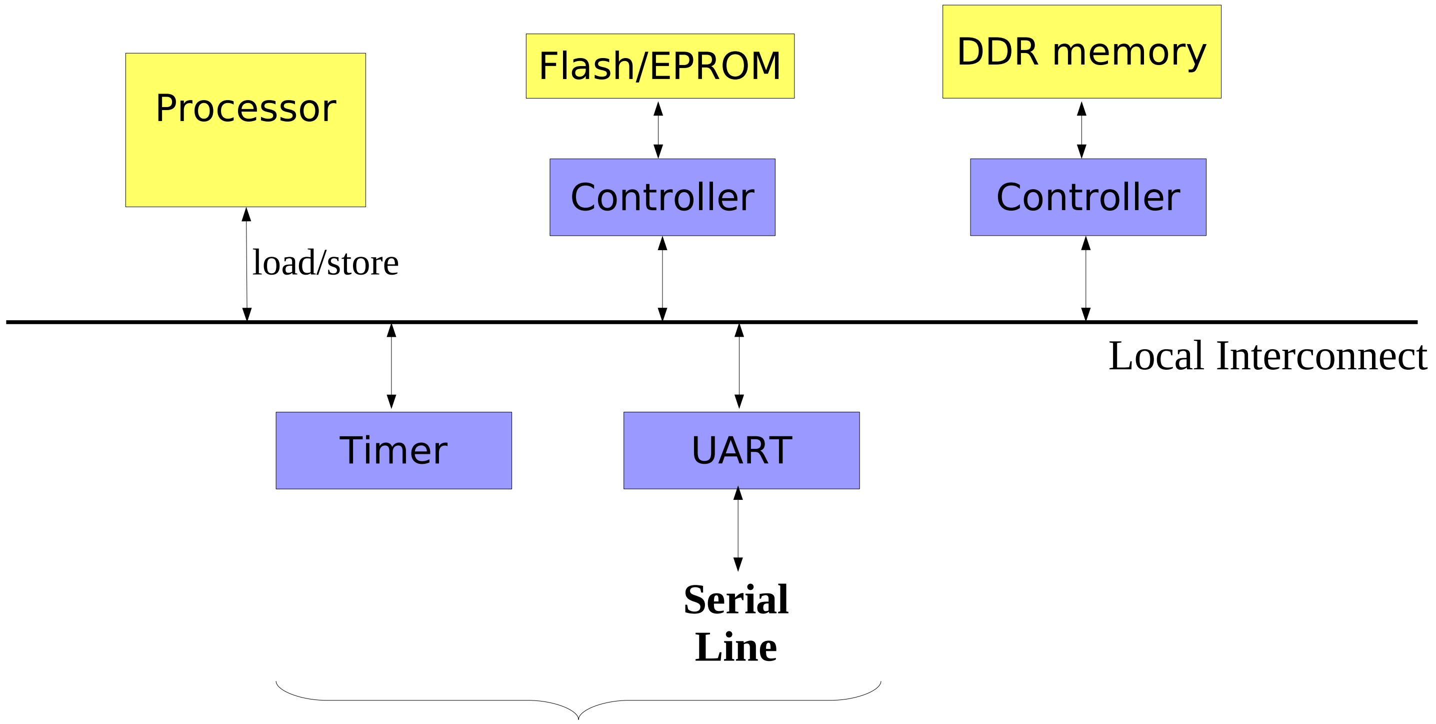
```
// Hardware Exception Vector (loaded 0x0000-0000, see linker script)

    ldr pc, reset_handler_addr
    ldr pc, undef_handler_addr
    ldr pc, swi_handler_addr
    ldr pc, prefetch_abort_handler_addr
    ldr pc, data_abort_handler_addr
    ldr pc, unused_addr
    ldr pc, irq_handler_addr
    ldr pc, fiq_handler_addr

reset_handler_addr: .word _reset_handler
undef_handler_addr: .word _undef_handler
swi_handler_addr: .word _swi_handler
prefetch_abort_handler_addr: .word _prefetch_abort_handler
data_abort_handler_addr: .word _data_abort_handler
not_used_addr: .word _not_used_handler
irq_handler_addr: .word _isr_handler
fiq_handler_addr: .word _fiq_handler


_undef_handler: // undefined instruction
     ...
1:   b 1b // infinite loop for debugging

_prefetch_abort_handler: // prefetch-abort trap
     ...
1:   b 1b // infinite loop for debugging

_data_abort_handler: // load/store abort trap
     ...
1:   b 1b // infinite loop for debugging
```

Prefetch-abort and ***data-abort hardware exceptions*** are triggered by the memory subsystem.

The ***undef hardware exception*** is triggered by the core itself, when it tries to execute an undefined instruction

The execution will trap here when your program has bugs...

Next Time – Hardware Peripherals



Without peripherals,
a system can't do much...

Terminal

Serial Line

VersatilePB