Virtual Machines

Pr. Olivier Gruber

olivier.gruber@imag.fr

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Acknowledgments

- Reference Book

  Virtual Machines
  Versatile Platforms for systems and processes

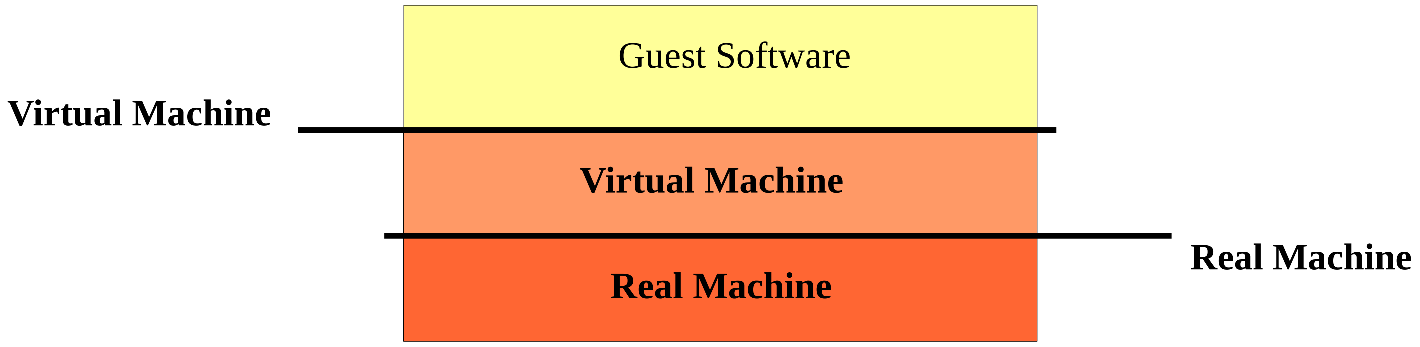
  James E. **Smith**, Ravi **Nair**

  Morgan Kaufmann

- Research Articles
  - Cited on various slides

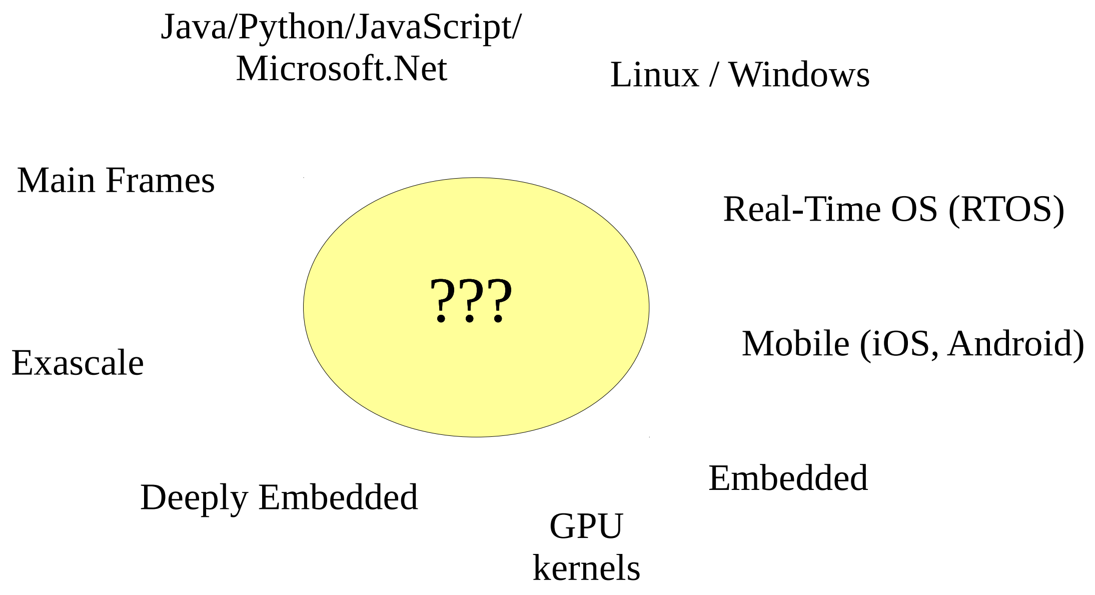
Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *A virtual machine defines a machine (interface)*
  - *A virtual machine is a machine (implementation)*



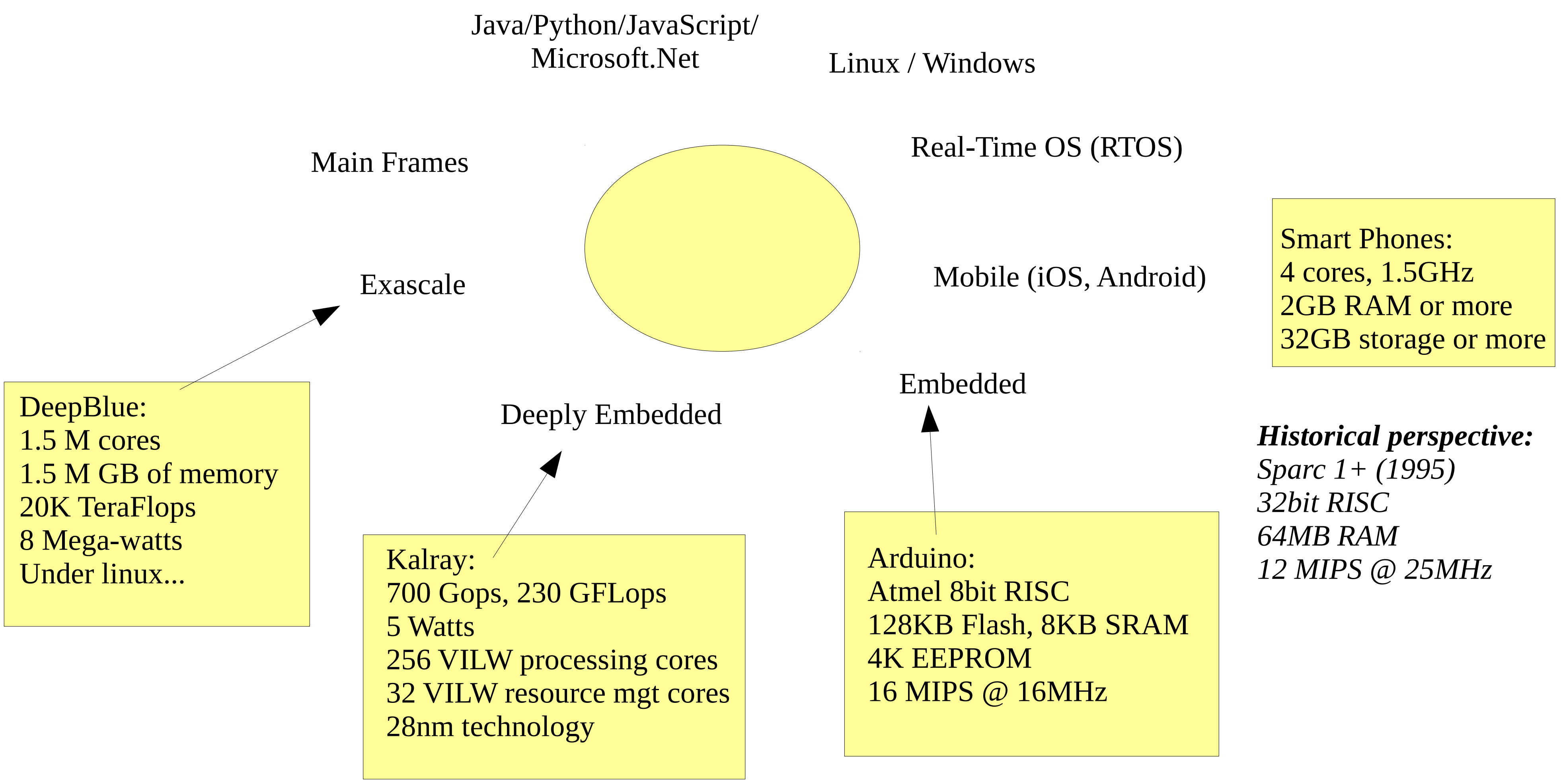
Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *So many virtual machines...*



Virtual Machine Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *So many real machines...*



DeepBlue:
1.5 M cores
1.5 M GB of memory
20K TeraFlops
8 Mega-watts
Under linux...

Kalray:
700 Gops, 230 GFLops
5 Watts
256 VILW processing cores
32 VILW resource mgt cores
28nm technology

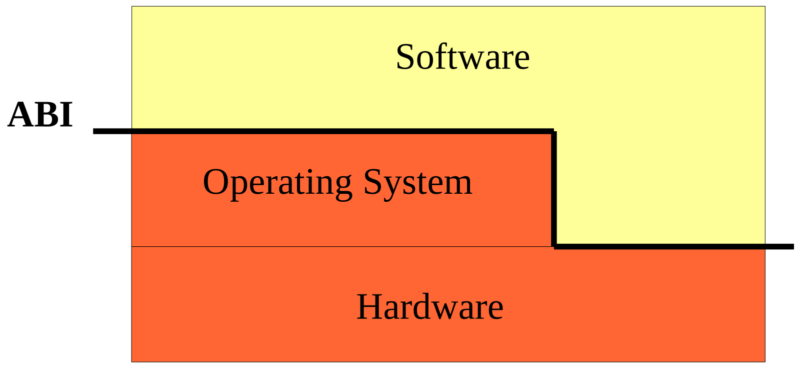
Arduino:
Atmel 8bit RISC
128KB Flash, 8KB SRAM
4K EEPROM
16 MIPS @ 16MHz

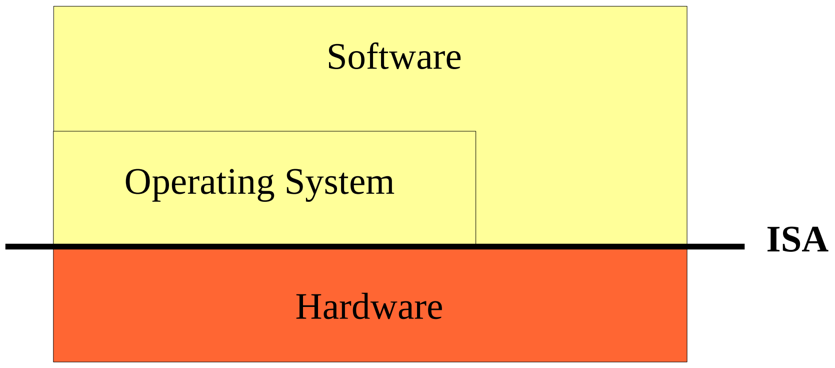
Smart Phones:
4 cores, 1.5GHz
2GB RAM or more
32GB storage or more

Historical perspective:
Sparc 1+ (1995)
32bit RISC
64MB RAM
12 MIPS @ 25MHz

Virtual Machine Basics

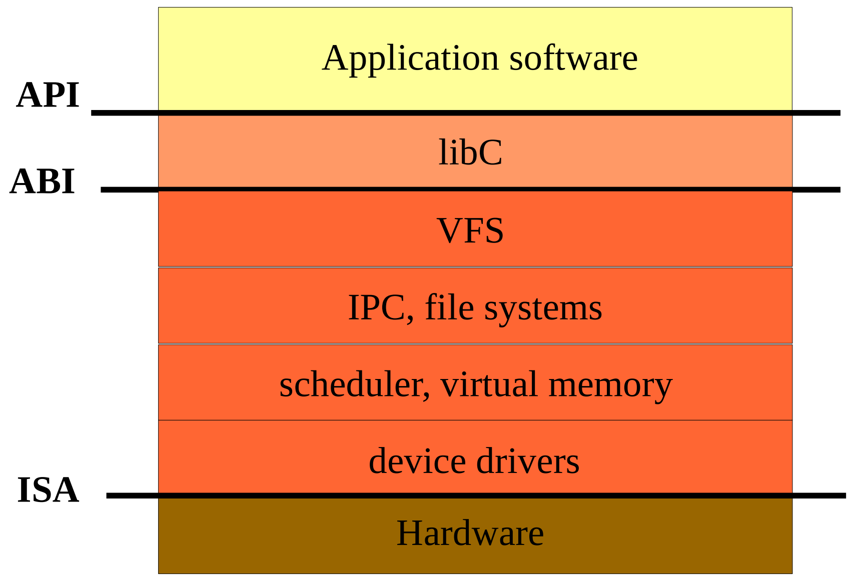
- *Instruction Set Architecture* (ISA)
  - Defines the instruction set
  - Defines other concepts such as page tables, traps, interrupts, etc.
- Application Binary Interface (ABI)
  - Defines core concepts above the ISA
  - Example:
    - Linux kernel system calls
    - Related to processes, threads, files, and devices

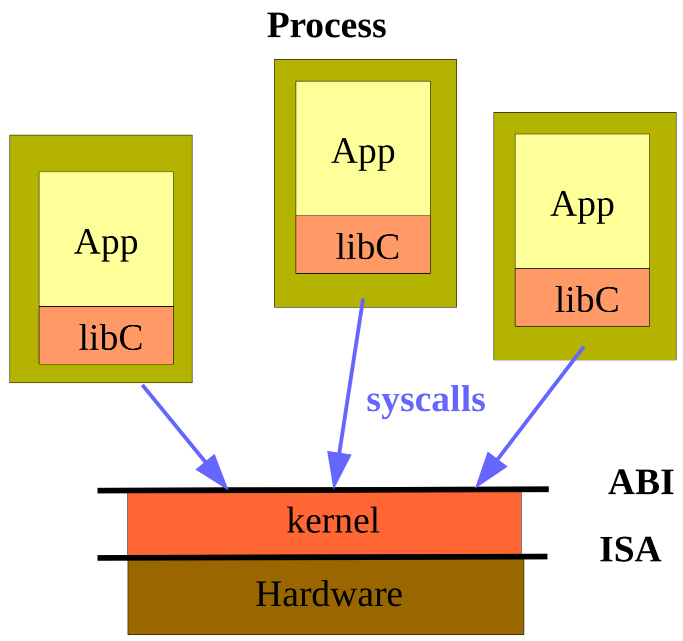




Operating System Architecture

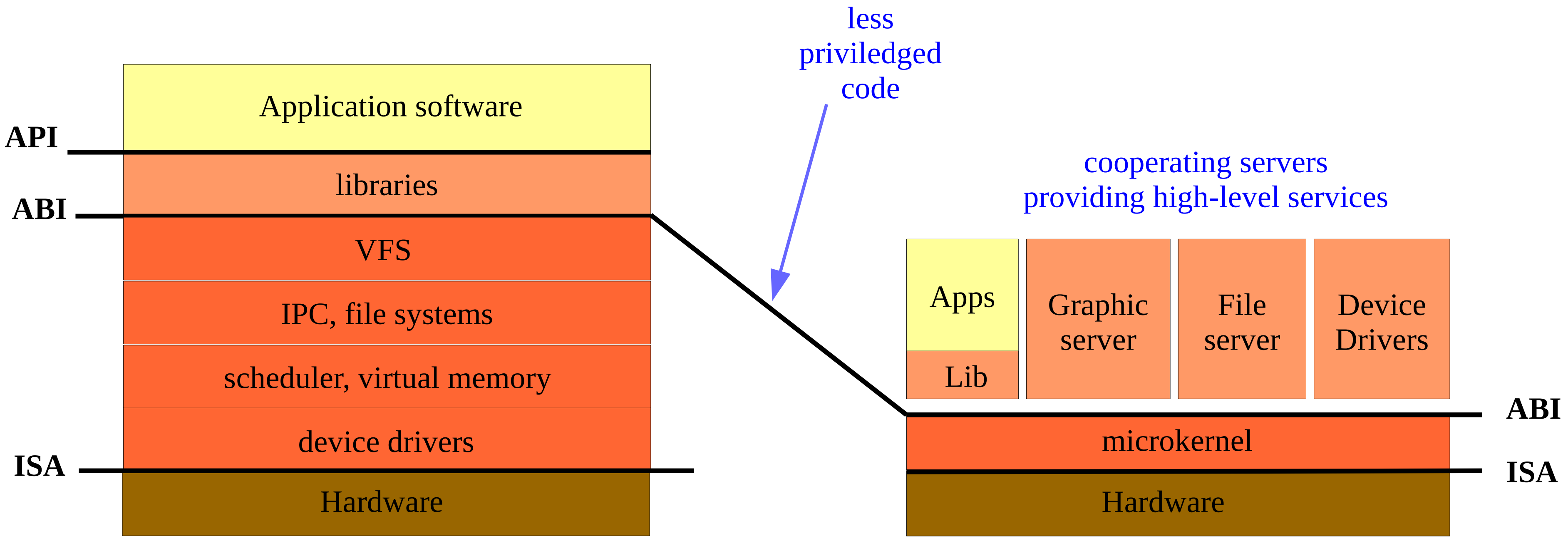
- Typical Unix-like Architecture
  - A monolithic kernel acting as a shared coordinator
  - Multiple client processes hosting the libC and an application





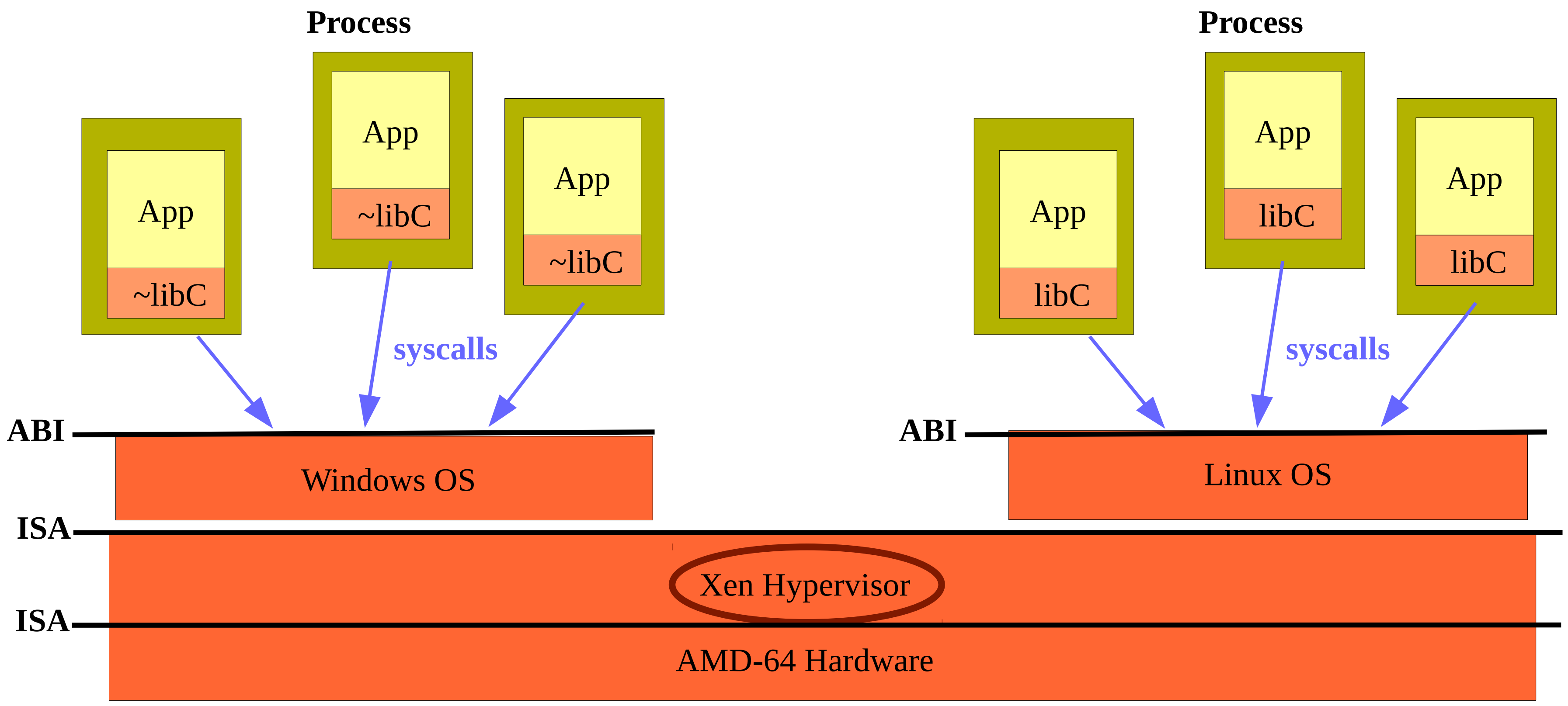
Operating System Architecture

- Microkernels
  - Minimal kernel (TCB) with only few and low-level mechanisms
  - Promote a modular design of cooperative user-level servers
  - Cooperation relies on Inter-Process Communications (IPC)



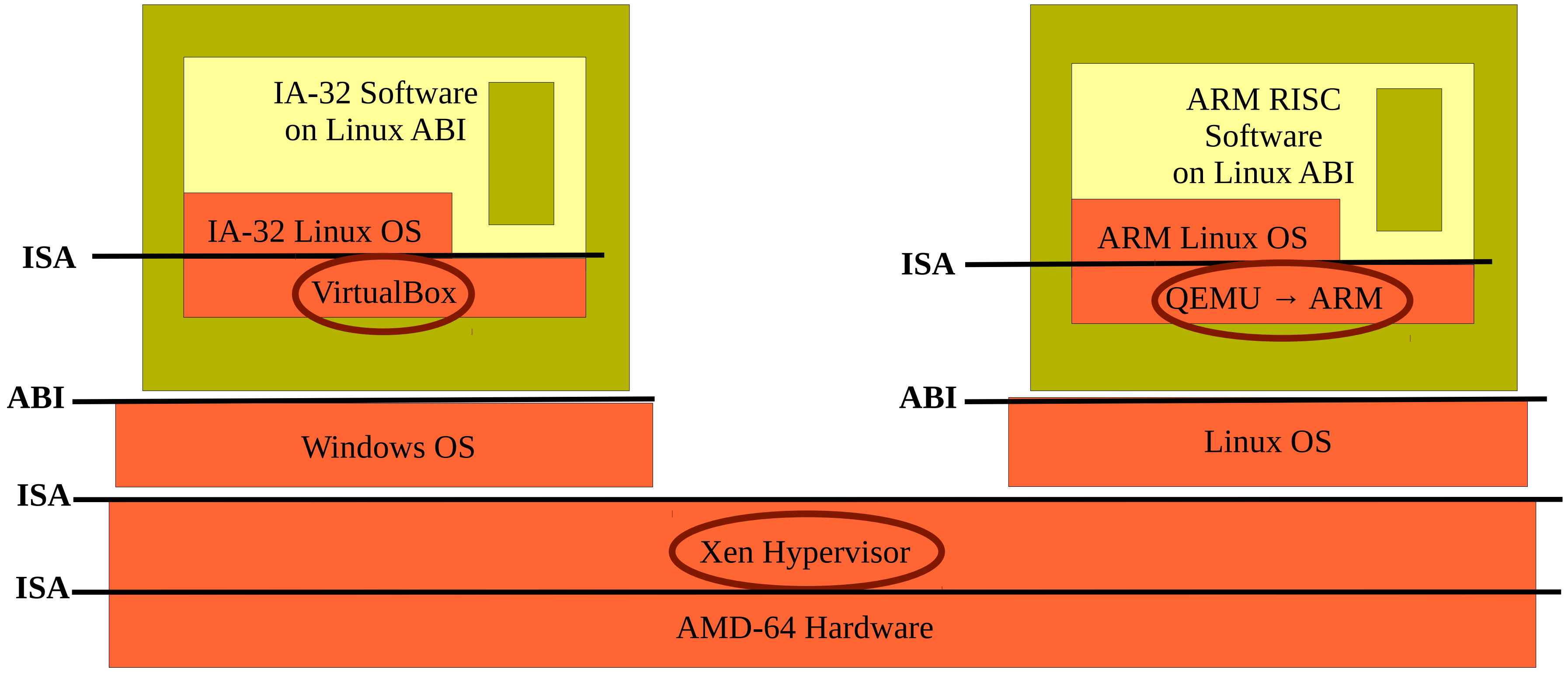
Operating System Architecture

- Type-I Hypervisors
  - A monolithic kernel acting as a shared coordinator
  - Hosting multiple guest operating systems



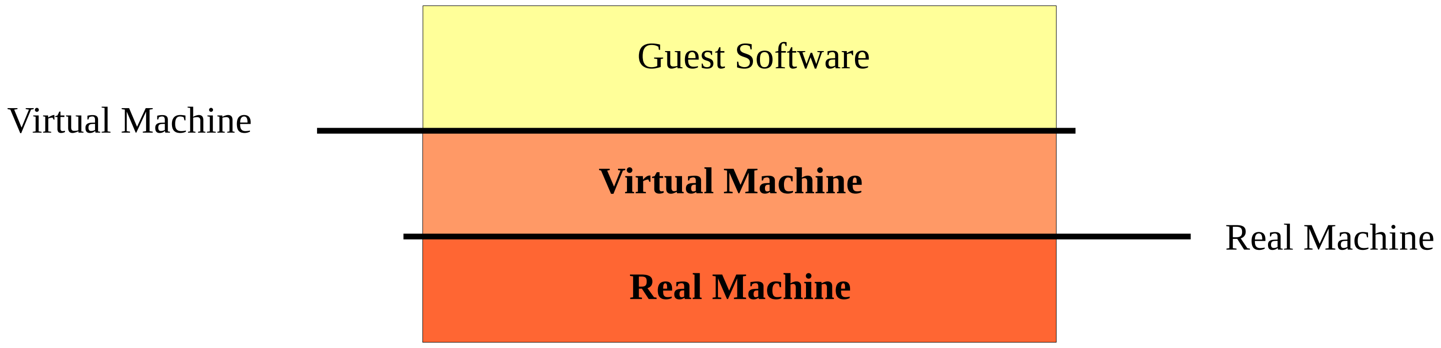
Operating System Architecture

- Type-II Hypervisors
  - Run within a regular process
  - Host a single guest operating system

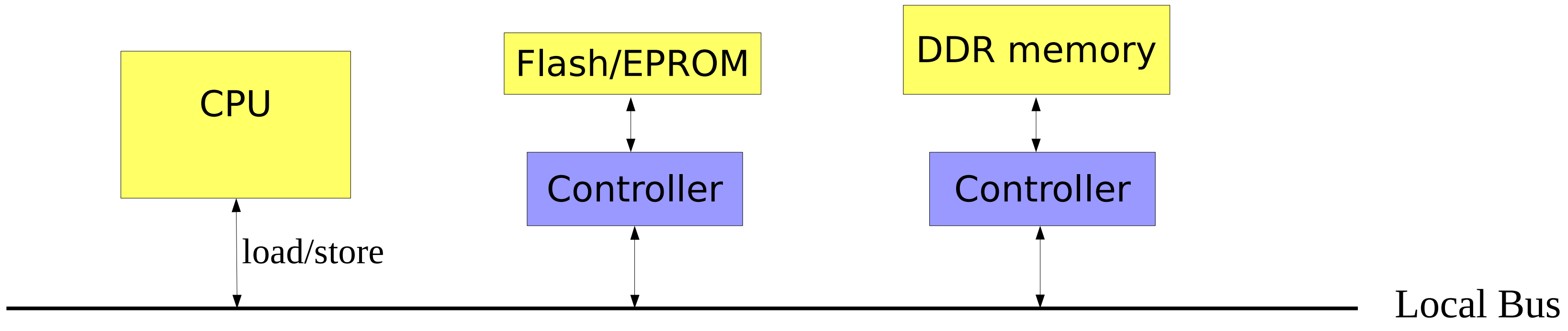


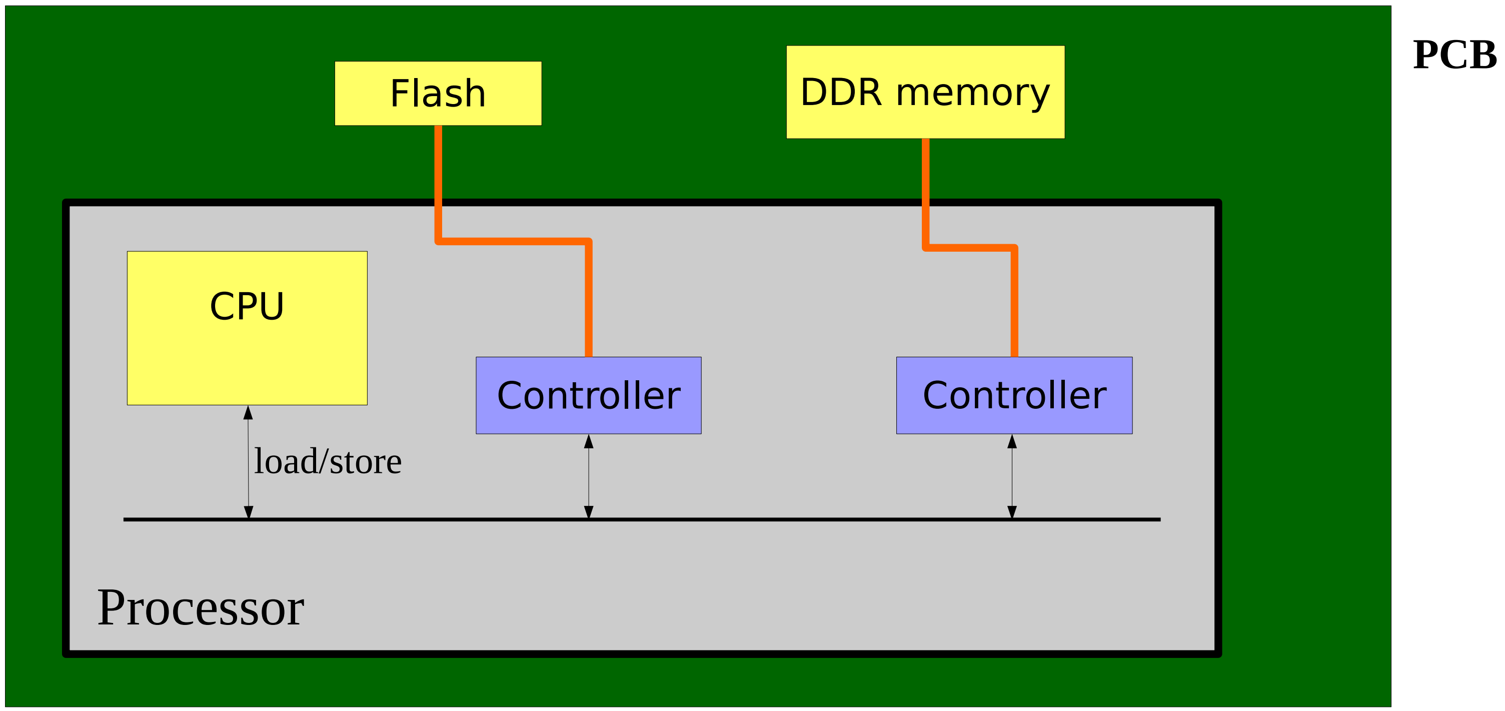
Back to Basics

- ***Virtual Machines*** versus ***Real Machines***
  - *Basics of hardware and operating systems*
  - *From the ground up...*

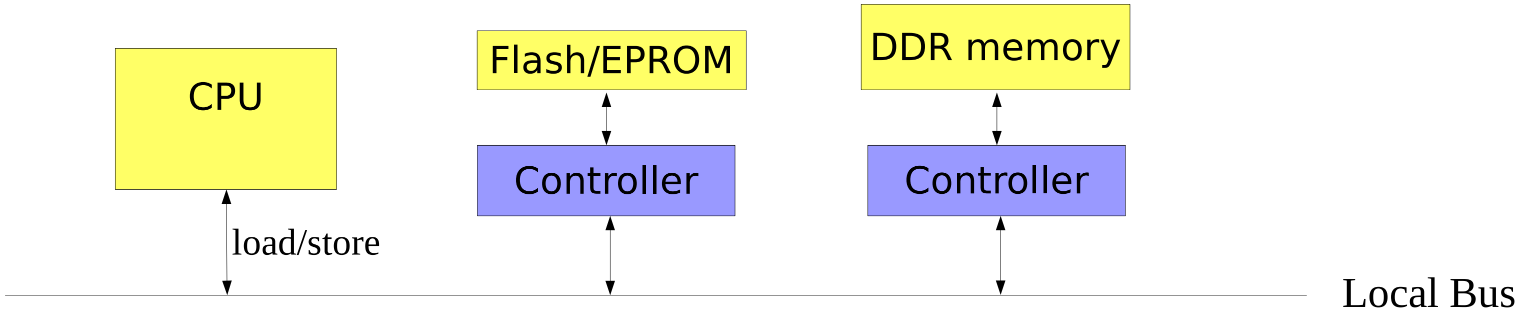


Hardware – Basics



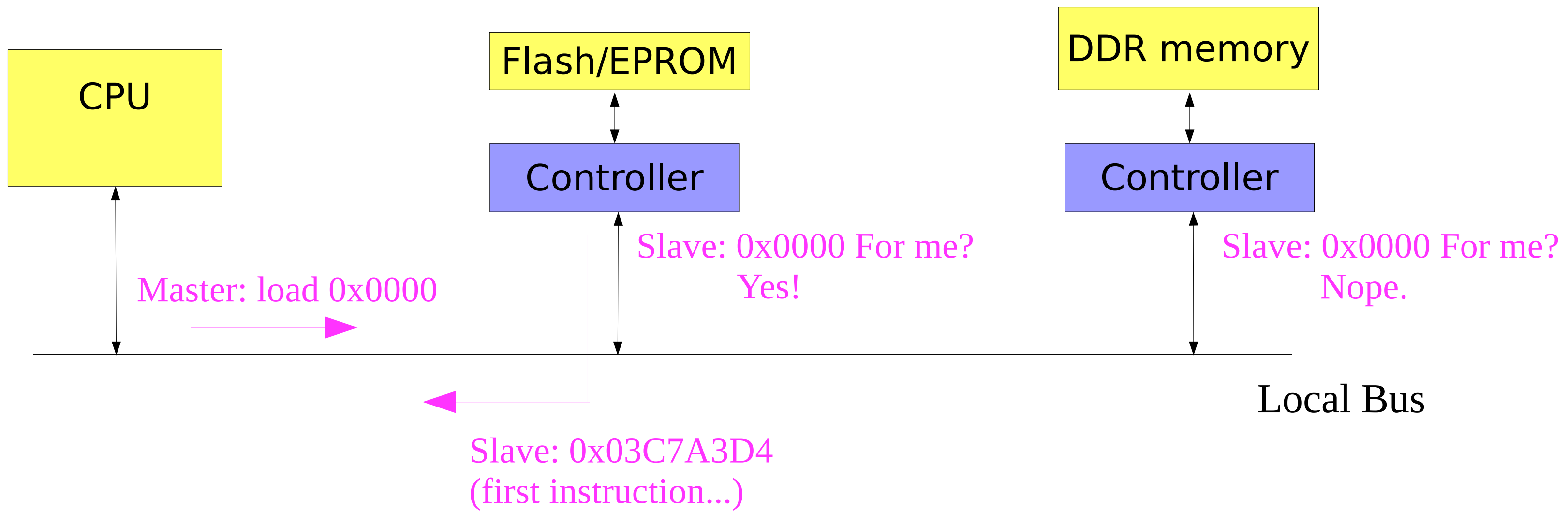


Hardware – Reset / Power-up



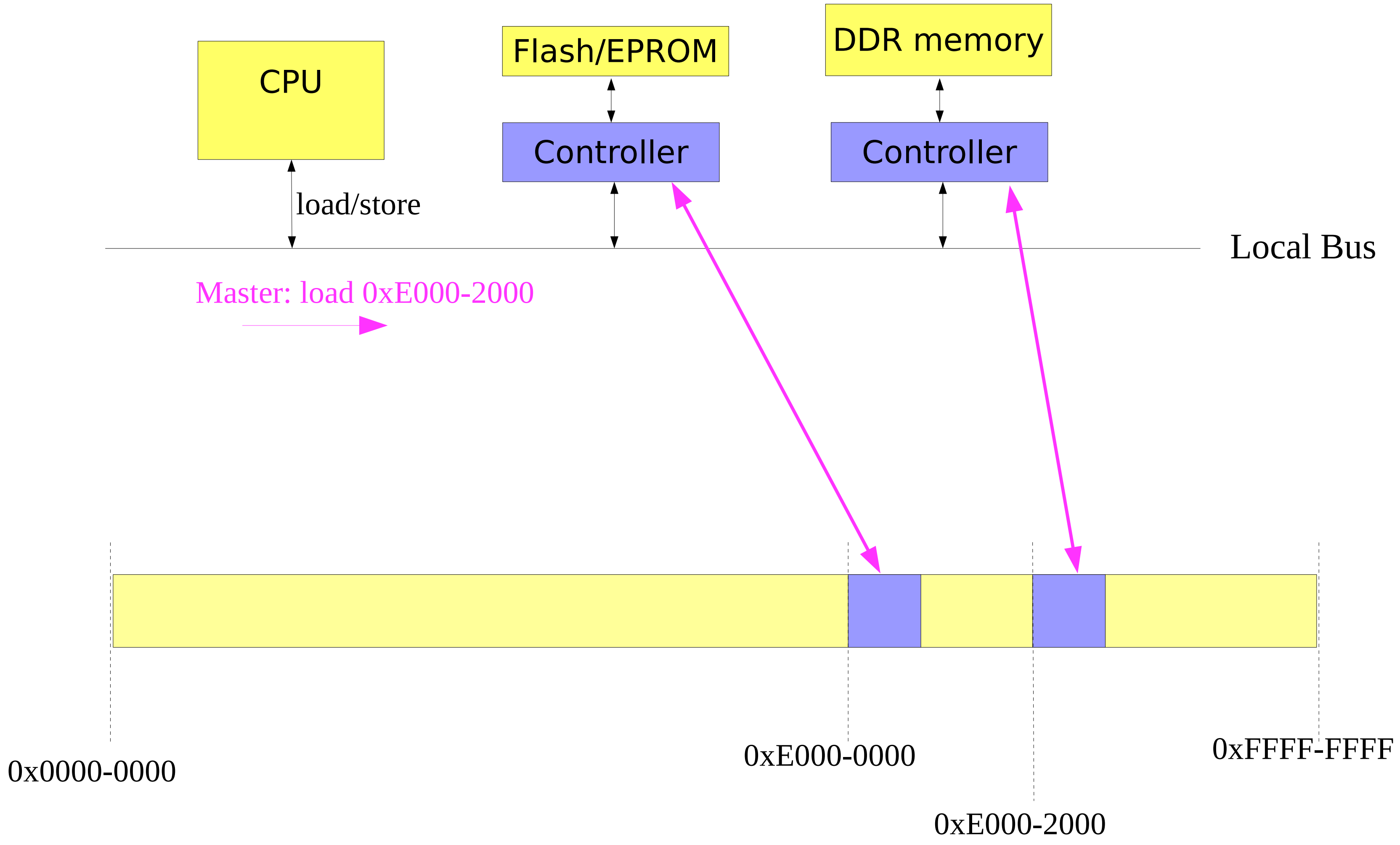
- Boot sequence
  - Wakes up at a given address, let's at 0x0000-0000
  - Starts executing there, but what is there?
  - Pre-loaded Flash/EPROM memory
- Bus
  - 32 data wires + control wires
  - Master/slave → CPU is master, controllers are slaves (for now)
  - EPROM/Flash controller is a slave, factory-configured at @0x0000

Hardware – Reset / Power-up


CPU
Flash/EPROM
DDR memory
Controller
Controller
Master: load 0x0000
Slave: 0x0000 For me? Yes!
Slave: 0x0000 For me? Nope.
Local Bus
Slave: 0x03C7A3D4 (first instruction...)

- Fetching the first instruction...
  - The Flash/EPROM controller is probably the only one that is factory-configured
  - Other controllers are waiting to be software configured
- How is software configuring hardware?
  - Through memory-mapped I/O registers (mmio registers)
  - Also called “hardware registers”

Hardware – MMIO Registers



Zynq-7000 Memory Map

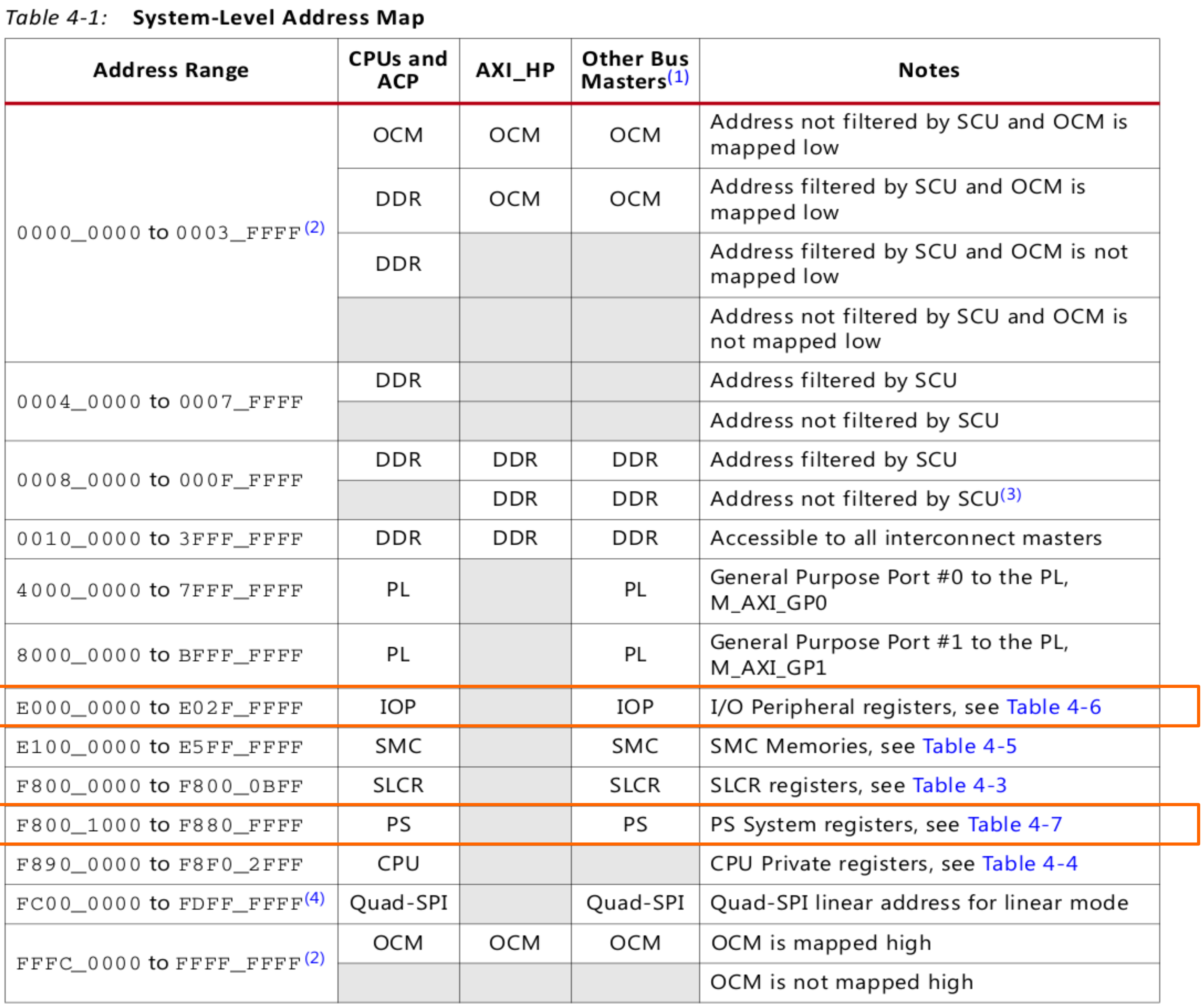


Table 4-1: **System-Level Address Map**

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

Zynq-7000 Memory Map

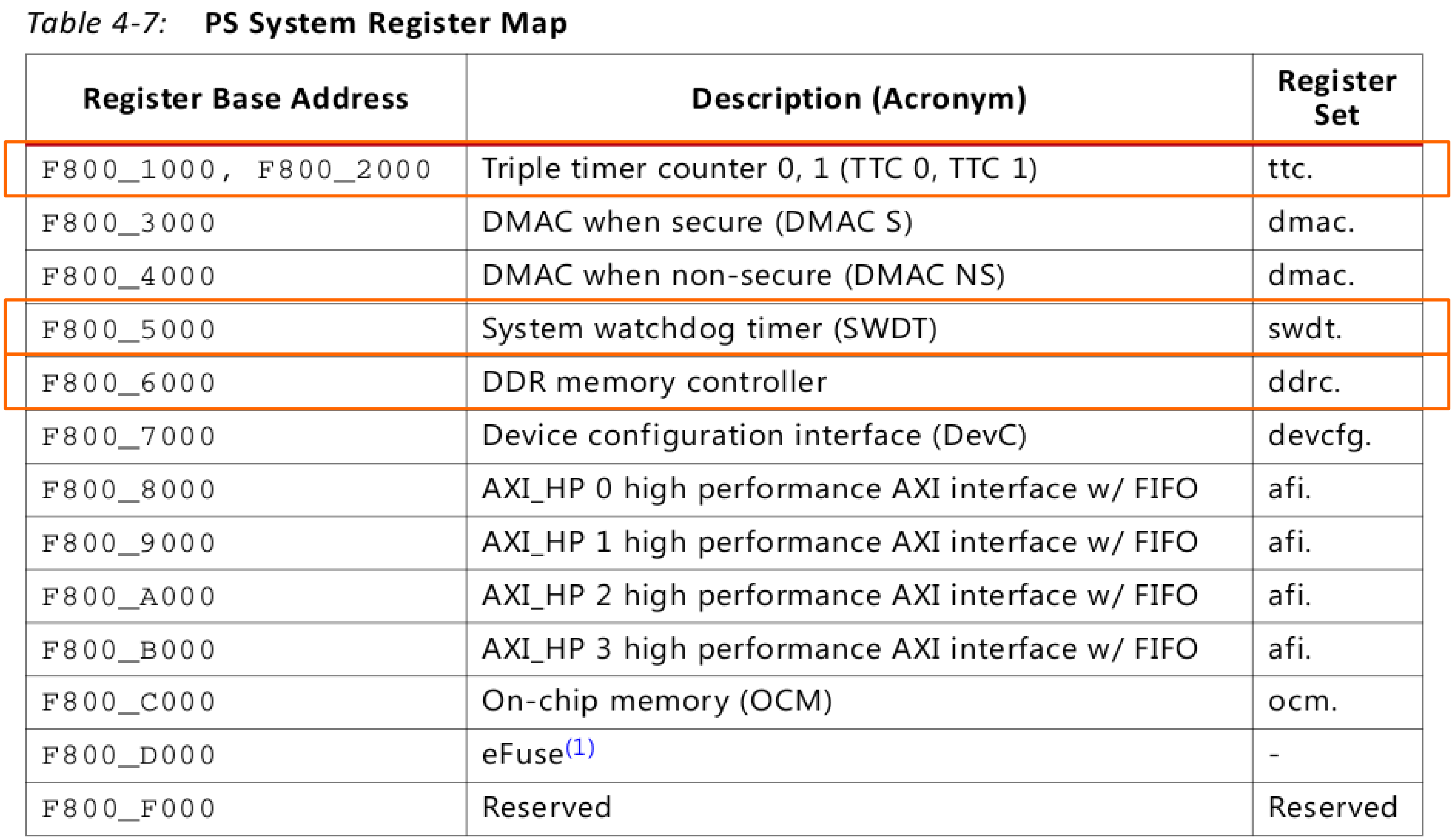


Table 4-7: **PS System Register Map**

Register Base Address	Description (Acronym)	Register Set
F800_1000, F800_2000	Triple timer counter 0, 1 (TTC 0, TTC 1)	ttc.
F800_3000	DMAC when secure (DMAC S)	dmac.
F800_4000	DMAC when non-secure (DMAC NS)	dmac.
F800_5000	System watchdog timer (SWDT)	swdt.
F800_6000	DDR memory controller	ddrc.
F800_7000	Device configuration interface (DevC)	devcfg.
F800_8000	AXI_HP 0 high performance AXI interface w/ FIFO	afi.
F800_9000	AXI_HP 1 high performance AXI interface w/ FIFO	afi.
F800_A000	AXI_HP 2 high performance AXI interface w/ FIFO	afi.
F800_B000	AXI_HP 3 high performance AXI interface w/ FIFO	afi.
F800_C000	On-chip memory (OCM)	ocm.
F800_D000	eFuse[1]	-
F800_F000	Reserved	Reserved

Technical Reference Manual: Zynq-7000 All Programmable SoC

Zynq-7000 Memory Map

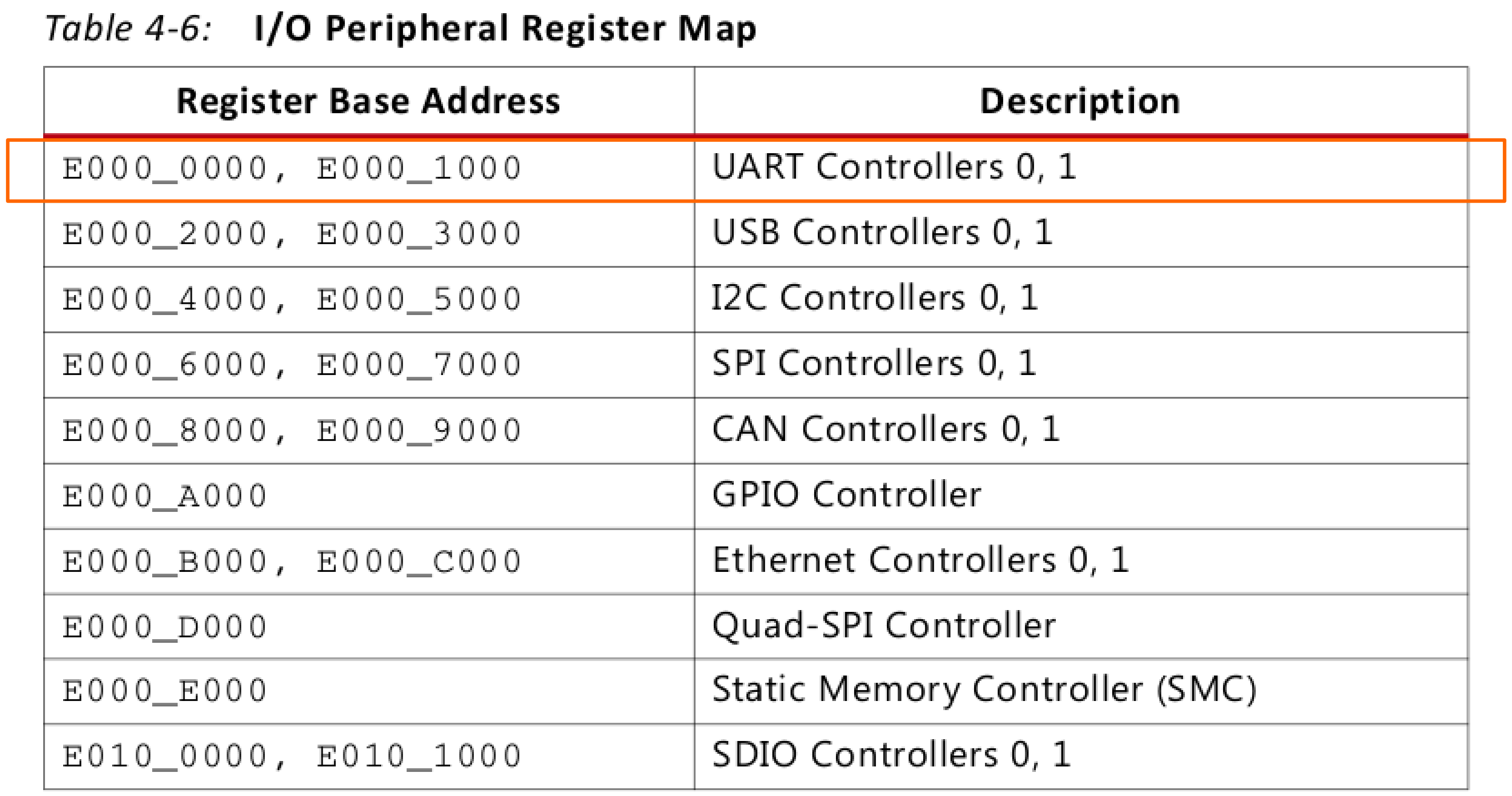


Table 4-6: **I/O Peripheral Register Map**

Register Base Address	Description
E000_0000, E000_1000	UART Controllers 0, 1
E000_2000, E000_3000	USB Controllers 0, 1
E000_4000, E000_5000	I2C Controllers 0, 1
E000_6000, E000_7000	SPI Controllers 0, 1
E000_8000, E000_9000	CAN Controllers 0, 1
E000_A000	GPIO Controller
E000_B000, E000_C000	Ethernet Controllers 0, 1
E000_D000	Quad-SPI Controller
E000_E000	Static Memory Controller (SMC)
E010_0000, E010_1000	SDIO Controllers 0, 1

Technical Reference Manual: Zynq-7000 All Programmable SoC

UART Device Example

UART... serial line controller, following the RS-232 protocol...
→ Essentially a FIFO and a status register...

Constants defined from reading the Zynq-7000/R1P8 Technical Reference Manual...
not the most fun part of it all...

```
#define UART_R1P8_CR          0x0000 /* UART Control Register */
#define UART_R1P8_MR          0x0004 /* UART Mode Register */
#define UART_R1P8_IER         0x0008 /* -- Interrupt Enable Register */
#define UART_R1P8_IDR         0x000C /* -- Interrupt Disable Register */
#define UART_R1P8_IMR         0x0010 /* -- Interrupt Mask Register */
#define UART_R1P8_ISR         0x0014 /* -- Channel Interrupt Status Register */
#define UART_R1P8_BAUDGEN     0x0018 /* Baude Rate Generator Register */
#define UART_R1P8_RXTOUT      0x001C /* -- Receiver Timeout Register */
#define UART_R1P8_RXWM        0x0020 /* -- Receiver FIFO Trigger Level Register */
#define UART_R1P8_MODEMCR     0x0024 /* -- Modem Control Register */
#define UART_R1P8_MODEMSR     0x0028 /* -- Modem Status Register */
#define UART_R1P8_SR          0x002C /* Channel Status Register */
#define UART_R1P8_FIFO        0x0030 /* Transmit & Receive FIFO */
#define UART_R1P8_BAUDDIV     0x0034 /* Baud Rate Divider Register */
#define UART_R1P8_FLOWD       0x0038 /* -- Flow Control Delay Register */
#define UART_R1P8_TXWM        0x0044 /* -- Transmitter FIFO Trigger Level Register */
```

Questions – Your mind at work...

- Imagine your UART
  - From the hardware side – hardware registers, layout, functions
  - From the software side – send and receive a character
- Apply to a console
  - Imagine you have a micro-controller with two UARTs
  - Serial line to keyboard
  - Serial line to a terminal (screen)

UART – Initialization

```
/* Zynq TRM sequence (UG585 p598) */
void
uart_r1p8_init_regs(void* uart){

  /* UART Character frame */
  mmio_reg_write32(uart,UART_R1P8_MR,UART_R1P8_MR_8n1);

  /* Baud Rate configuration */
  mmio_reg_setbits32(uart,UART_R1P8_CR,      UART_R1P8_CR_RXDIS | UART_R1P8_CR_TXDIS);
  mmio_reg_write32(uart,UART_R1P8_BAUDGEN,  UART_R1P8_115200_GEN);
  mmio_reg_write32(uart,UART_R1P8_BAUDDIV,  UART_R1P8_115200_DIV);
  mmio_reg_setbits32(uart,UART_R1P8_CR,      UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES);
  mmio_reg_setbits32(uart,UART_R1P8_CR,      UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN);

  /* Disable Rx Trigger level */
  mmio_reg_write32(uart,UART_R1P8_RXWM,     0x00);

  /* Enable Controller */
  mmio_reg_write32(uart,UART_R1P8_CR,      UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES |
                UART_R1P8_CR_RSTTO | UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN |
                UART_R1P8_CR_STPBRK);

  /* Configure Rx Timeout */
  mmio_reg_write32(uart,UART_R1P8_RXTOUT,   0x00);
  mmio_reg_write32(uart,UART_R1P8_IER,      0x00);
  mmio_reg_write32(uart,UART_R1P8_IDR,      UART_R1P8_IxR_ALL);

  /* No Flow delay */
  mmio_reg_write32(uart,UART_R1P8_FLOWD,    0x00);

  /* Desactivate flowcontrol */
  mmio_reg_clearbits32(uart,UART_R1P8_MODEMCR, UART_R1P8_MODEMCR_FCM);

  /* Mask all interrupts */
  mmio_reg_clearbits32(uart,UART_R1P8_IMR, 0x01FFF);
}
```

UART – Output

```
#define UART0  0xE0000000
#define UART1  0xE0001000

#define UART_R1P8_SR              0x002C /* Channel Status Register */
#define UART_R1P8_FIFO            0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 <<  4)
#define UART_R1P8_SR_TEMPTY   (1 <<  3)
#define UART_R1P8_SR_RFUL     (1 <<  2)
#define UART_R1P8_SR_REMPTY   (1 <<  1)
#define UART_R1P8_SR_RTRIG    (1 <<  0)
```

```
void
uart_r1p8_putc(void* uart, uint8_t c) {
  while((mmio_reg_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_TFUL) != 0);
  if (c == '\n') {
    mmio_reg_write32(uart,UART_R1P8_FIFO, '\r');
    while((mmio_reg_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_TFUL) != 0);
  }
  mmio_reg_write32(uart,UART_R1P8_FIFO, c);
}
```

UART – Input

```
#define UART_R1P8_SR              0x002C /* Channel Status Register */
#define UART_R1P8_FIFO            0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 <<  4)
#define UART_R1P8_SR_TEMPTY   (1 <<  3)
#define UART_R1P8_SR_RFUL     (1 <<  2)
#define UART_R1P8_SR_REMPTY   (1 <<  1)
#define UART_R1P8_SR_RTRIG    (1 <<  0)
```

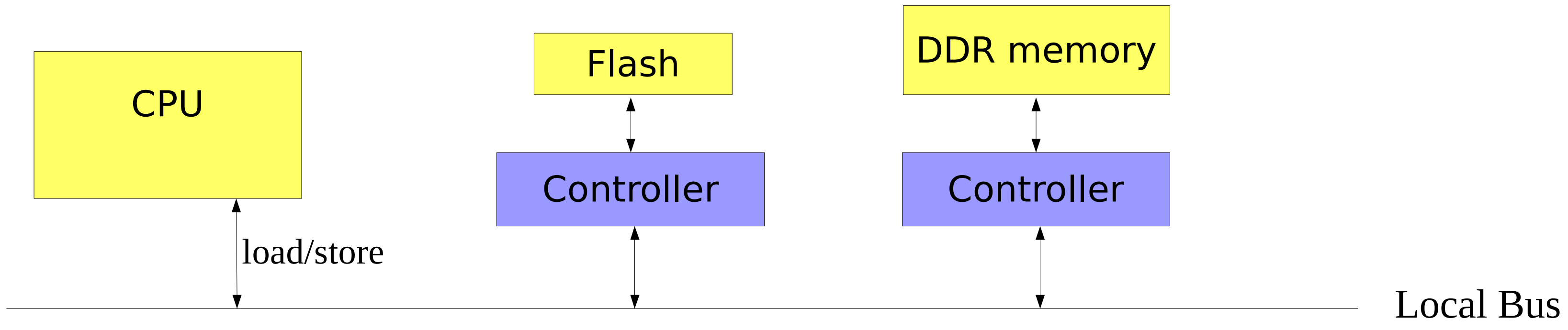
```
uint8_t
uart_r1p8_getc(void* uart){
  while((mmio_reg_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_REMPTY))
    ;
  uint8_t c;
  c = mmio_reg_read32(uart,UART_R1P8_FIFO);
  if (c=='\r')
    c='\n';
  return c;
}
```

Computer Basics – Micro-controllers

- Hardware
  - Single-core CPU, with load/store interface
  - Memory controller and memory
  - Instruction set: load/store, arithmetic, branches, etc.
  - CPU registers, both general purpose and special
- Software
  - Continuous polling of devices
  - Reacting to the status changes of the different devices
  - Example:

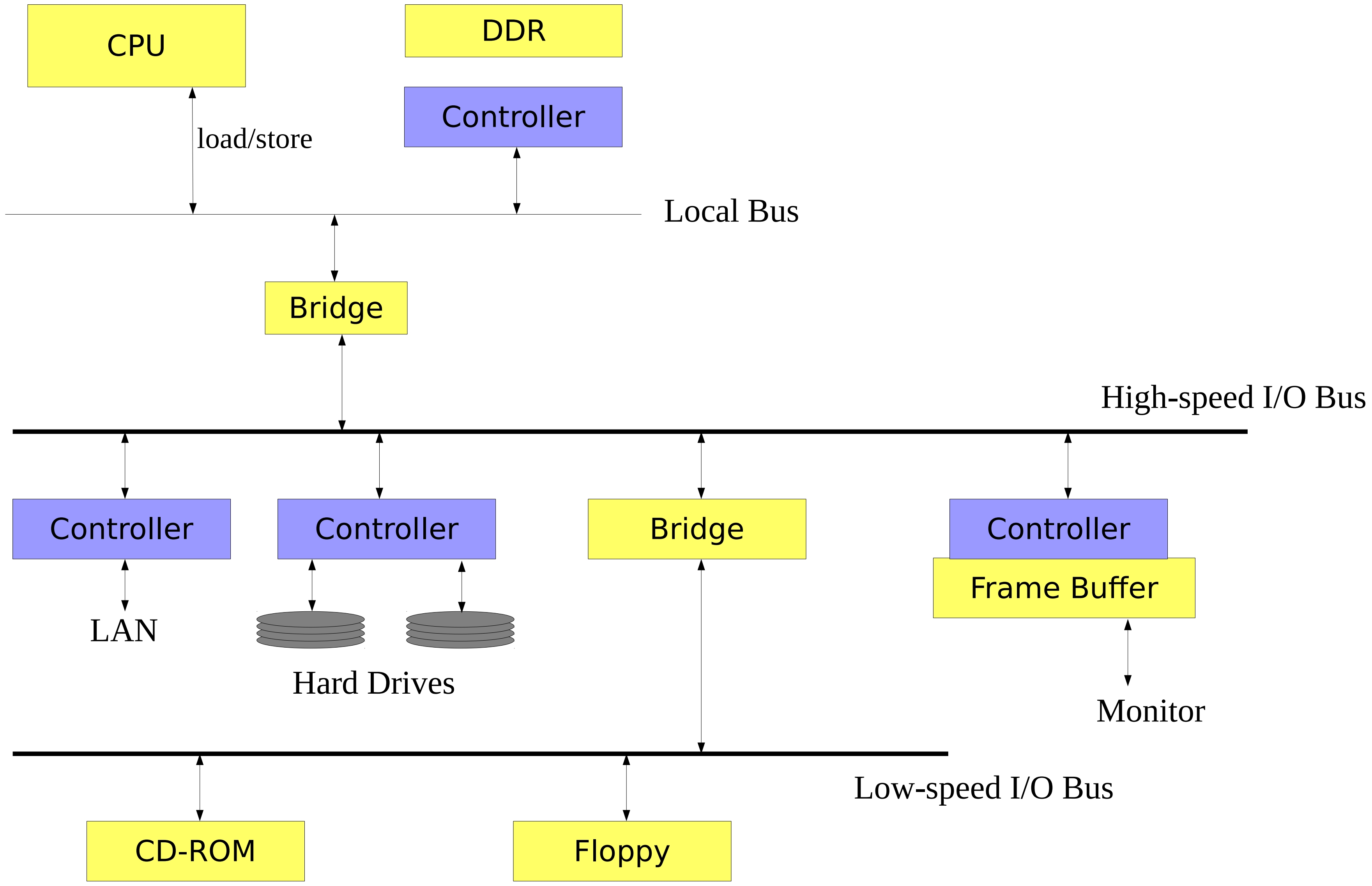
```
void main() {
  uart_r1p8_init_regs(KBD_UART);
  uart_r1p8_init_regs(TERM_UART);
  for (;;) {
     uint8_t code = uart_r1p8_getc(KBD_UART);
     uint16_t unicode = code_to_unicode(code);
     uart_r1p8_putc(TERM_UART, (unicode & 0xFF));
     uart_r1p8_putc(TERM_UART, (unicode & 0xFF)>>8);
  }
}
```

Hardware – Configuration



- Boot sequence – all about configuring the hardware
  - Reset configures the processor itself
  - Boot code
    - First controller configured is the DDR controller
    - But many other controllers... bus bridges, disks, Ethernet, USB, video, etc.

Hardware – Simple Overview



Zynq-7000 Overview

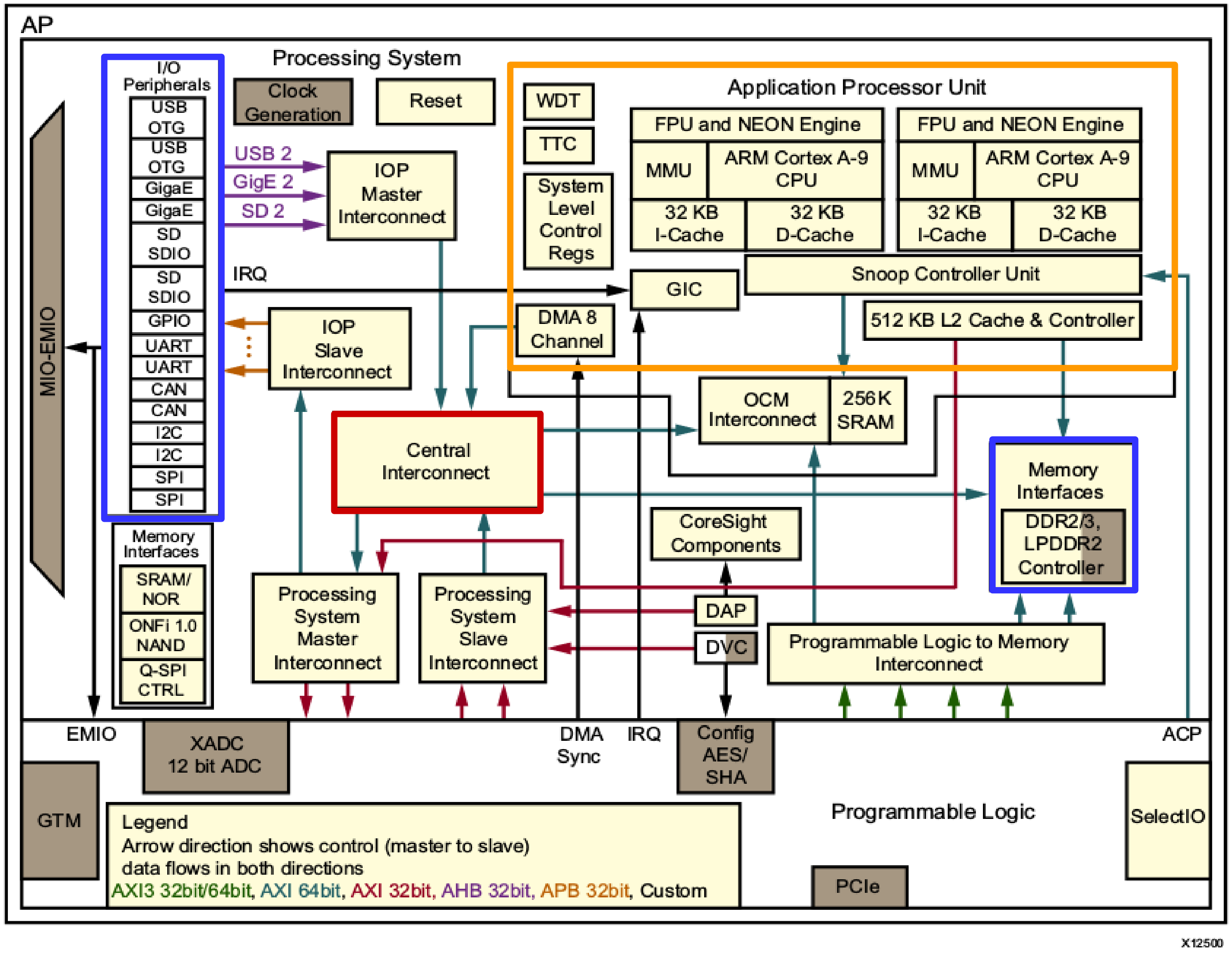


Figure 2-1: **Zynq-7000 AP SoC Processor System High-Level Diagram**

Your mind at work...

- Imagine your disk or network card
  - From the hardware side – hardware registers, layout, functions
  - From the software side – read and write a block
- Imagine your video buffer
  - From the hardware side – hardware registers, layout, functions
  - From the software side – send and receive Ethernet frame
- How would you code a simple game software?
  - Given that you have several devices
  - Given that you have animation on the screen

KEEP IT SIMPLE !

Remember
- **no IRQs**
- **no DMA**

Event-oriented Programming

- Event scheduler, with ready queue of events
  - Still polling devices...
- Event reactions run to completion
  - Long-running events are a BAD idea
  - No spinning
  - No blocking
- No need for synchronization
  - One event reaction at a time

```
struct event {
   void (*react)(struct event* evt, struct queue* rq);
}

void event_push(struct queue* rq,struct event* evt);
void event_pop(struct queue* rq,struct event* evt);

void scheduler(struct queue* rq) {
  ...
  for (;;) {
     struct event* evt = event_pop(rq);
     if (evt)
         evt→react(evt,rq);
     poll_devices();
  }
}
```

Event-oriented Programming

- Event-oriented game application
  - Periodic event to refresh the screen – flipping video buffers
  - Events to redraw the different items on the screen
  - Events to animate items
  - Etc.

Item example: an arrow in flight
XYZ coordinates
Direction of flight
Velocity

→ periodic reaction:
Update its position.
Compute if it hits something

```
struct event {
   void (*react)(struct event* evt, struct queue* rq);
}

void event_push(struct queue* rq,struct event* evt);
void event_pop(struct queue* rq,struct event* evt);

void scheduler(struct queue* rq) {
  ...
  for (;;) {
     struct event* evt = event_pop(rq);
     if (evt)
         evt→react(evt,rq);
     poll_devices();
  }
}
```

Your mind at work...

- Introduce interrupts to the previous design...
  - Keeping an event-oriented programming
  - Main concern – keep the programming model simple

Events & Interrupts

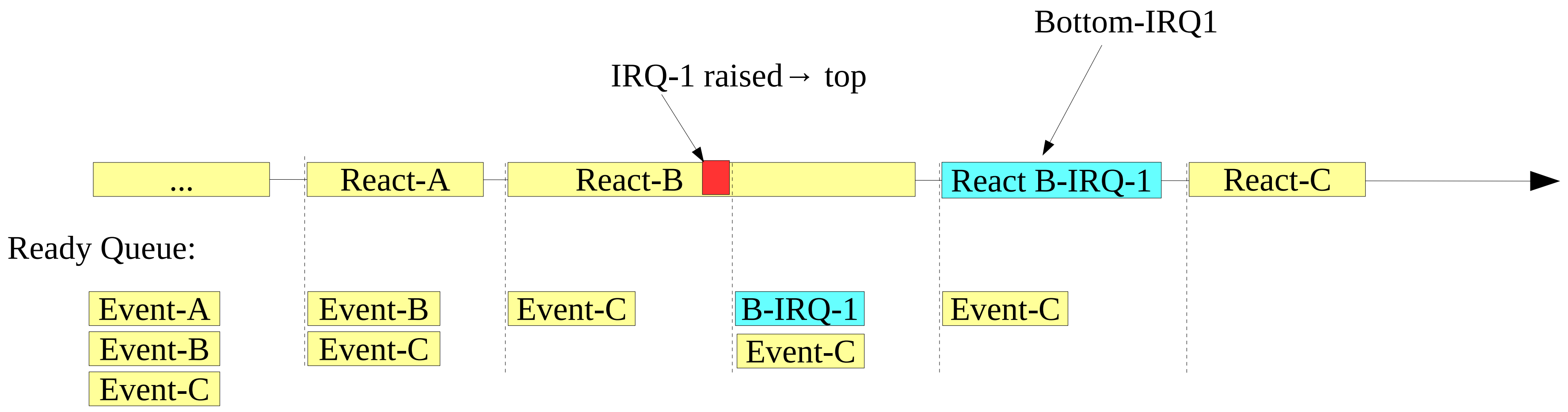
- Introducing interrupts
  - Interrupt vector in memory → Interrupt Service Routines (ISRs)
- Challenge – introduces a race condition
  - Between interrupt handlers and the event reactions
  - Simple solution: mutual exclusion
    - Enabling/disabling interrupts
  - Much better than polling
    - Especially from an energy consumption
  - But service latency is high
    - ISRs only execute between events

```
void scheduler(struct queue* rq) {
  ...
  for (;;) {
    disableInterrupts();
    struct event* evt = event_pop(rq);
    if (evt) {
       evt→react(evt,rq);
       enableInterrupts();
    } else {
       enableInterrupts();
        halt();
    }
  }
}
```

Tops & Bottoms

- Separating ISRs into a top and a bottom
  - Top handler:
    - Interrupt Service Routines (ISRs)
    - Happens when the interrupt is raised
  - Bottom event:
    - An event created to follow up top handlers
    - Will finish the processing

```
void **scheduler**(struct queue* rq) {
  ...
  for (;;) {
    struct event* evt =event_pop(rq);
    if (evt)
      evt→**react**(evt,rq);
    else
      halt();
  }
}
```



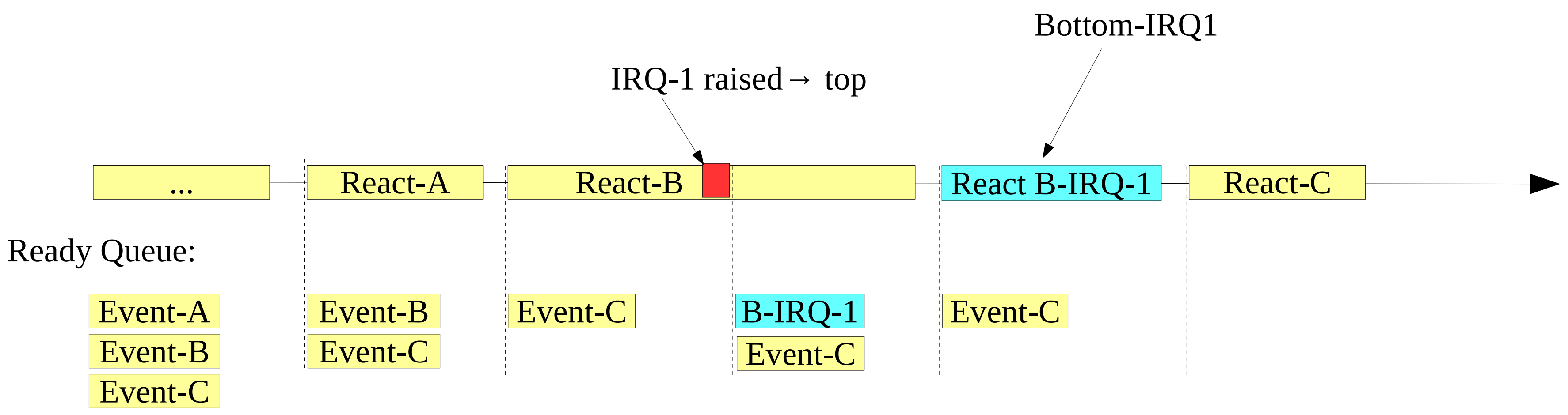
Your mind at work...

- Reflect on the programming constraint for top handlers..
  - Think in terms of when tops occur
  - Think in terms of race conditions
- Reflect on top versus bottom
  - One is a “handler”, the other is a regular event
  - Why?
  - Implications?
- Apply to an UART

Event-oriented Programming with Interrupts

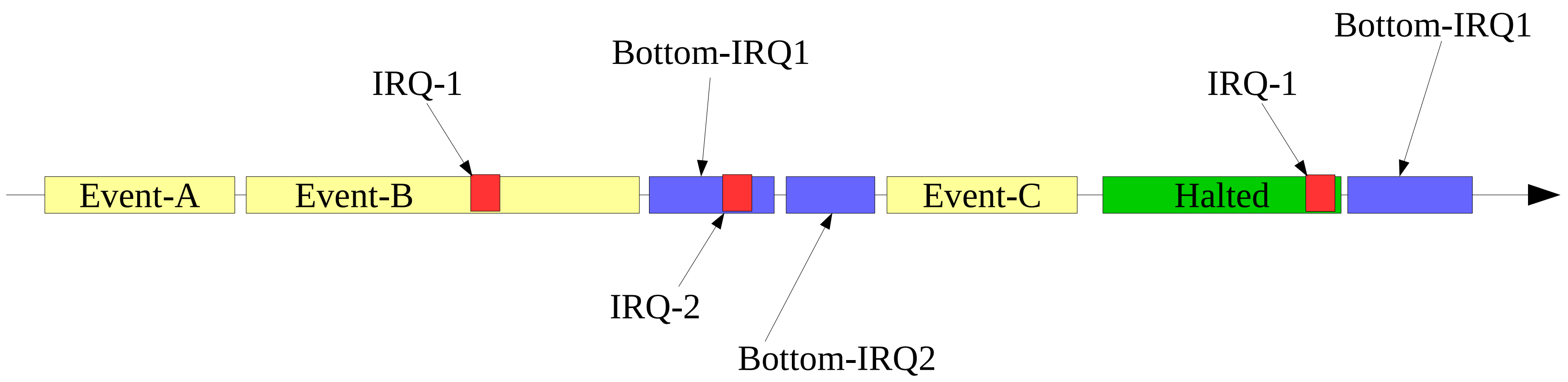
- Top programming model
  - Tops are time critical (must be short)
  - Tops happen any time, throughout events
  - Thus tops must be completely isolated
    - Must avoid any race condition
    - Cannot use synchronization
  - Usually rely on a lock-free circular buffer
    - To pass data to the bottom event
    - Lock-free with one producer and one consumer

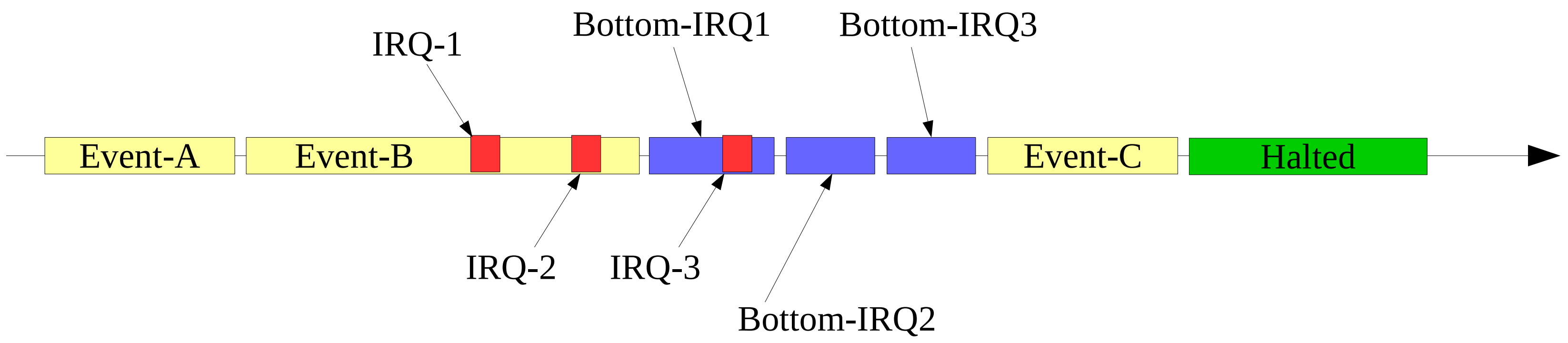
```
void scheduler(struct queue* rq) {
  ...
  for (;;) {
    struct event* evt =event_pop(rq);
    if (evt)
        evt→react(evt,rq);
    else
        halt();
  }
}
```



Event-oriented Programming with Interrupts

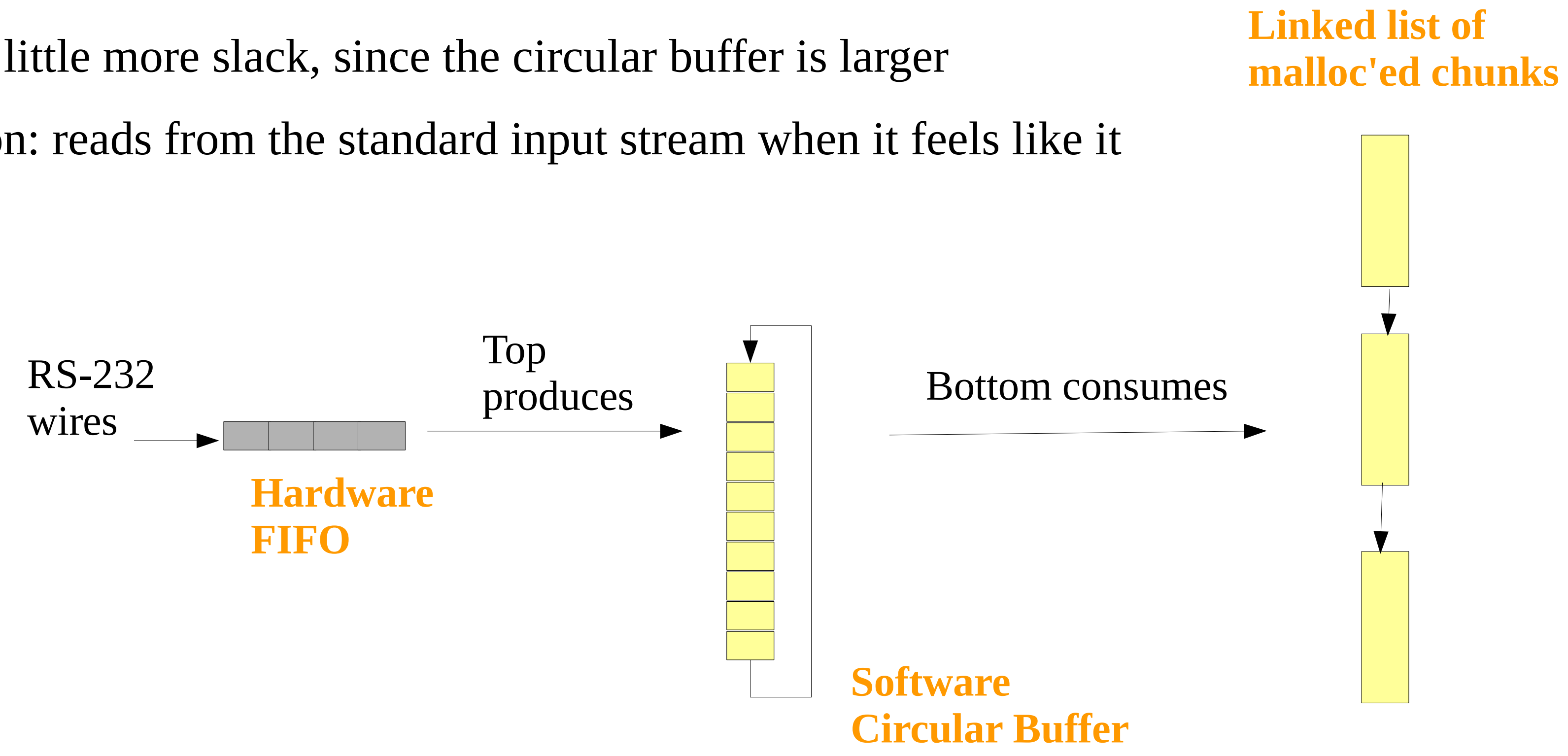
- Top/bottom sequencing examples





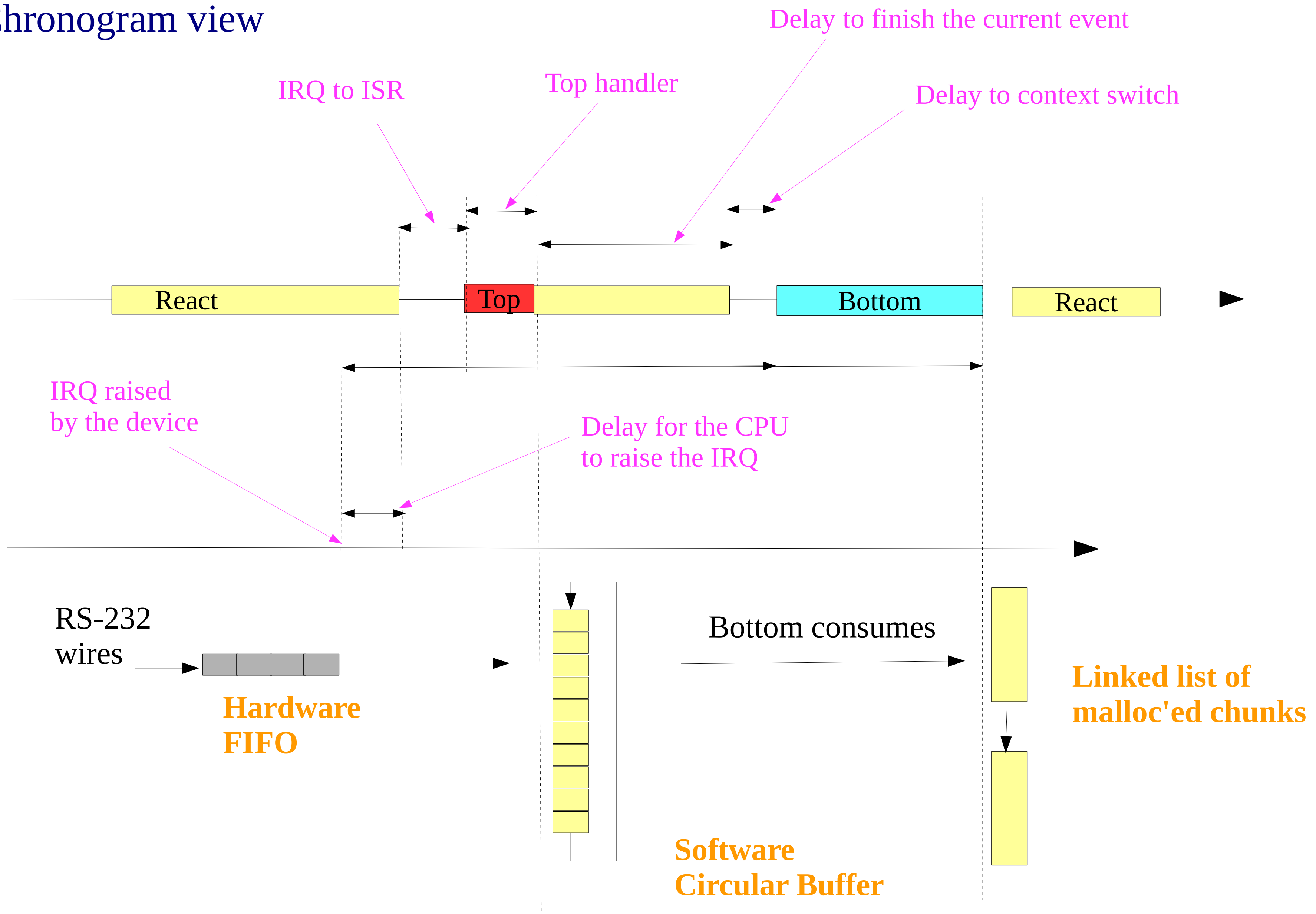
UART with Interrupts

- The goal is not loosing any character
  - Top: empty the "small" RX FIFO into a "larger" circular buffer in memory
  - Bottom: empty the fixed size circular buffer in an “variable-size buffer”
- A matter of delay requirements
  - Top: must be really quick to react
  - Bottom: a little more slack, since the circular buffer is larger
  - Application: reads from the standard input stream when it feels like it



UART with Interrupts

- Chronogram view

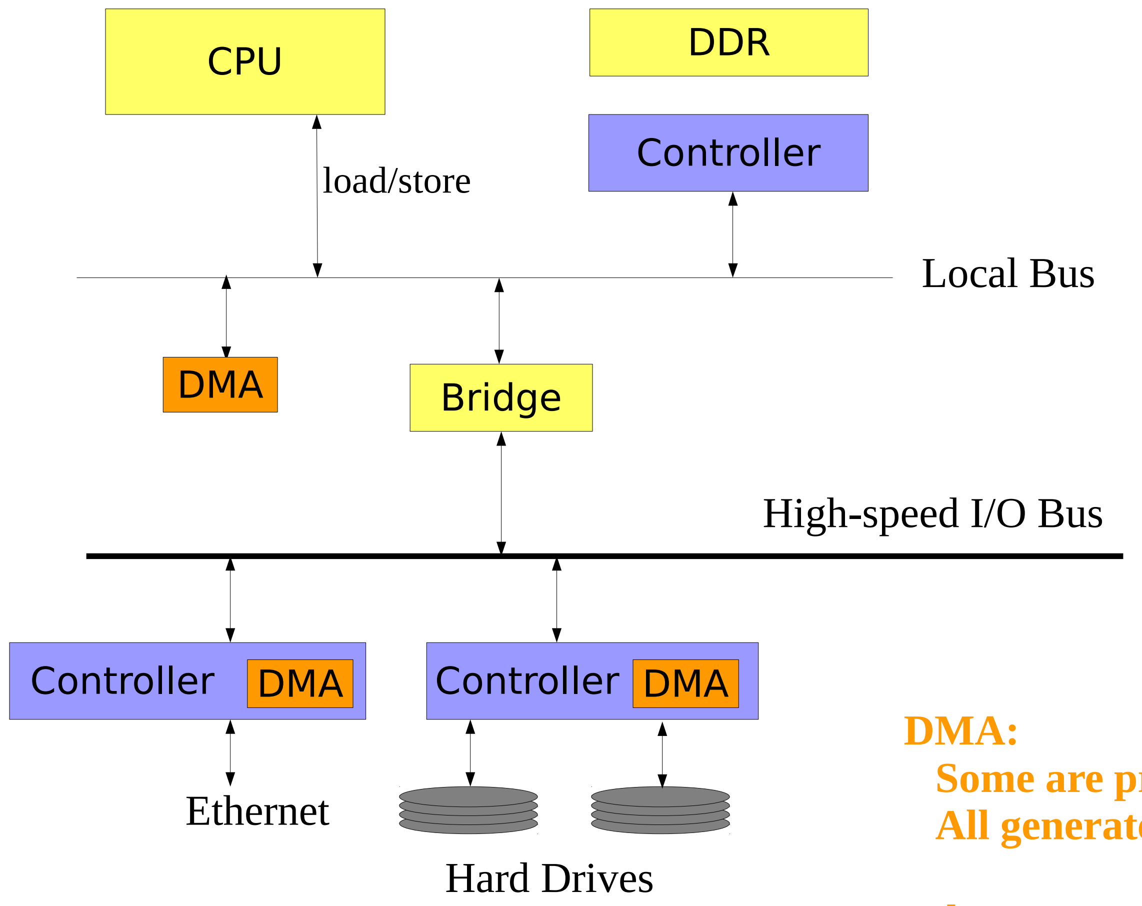


Computer Basics

- Hardware
  - Single-core CPU, with load/store interface
  - Memory controller and memory
  - Instructions: load/store, arithmetic, branches, etc.
  - CPU registers, both general purpose and special
  - Floating Point Unit (FPU), an arithmetic co-processor
  - **Interrupts with interrupt vector**
- Software
  - Event-oriented scheduler
  - **ISRs as top handler and bottom events**
  - Device drivers reading and writing mmio registers

Where is the potential performance bottleneck?

Direct Memory Access



DMA:

Some are programmable

All generate load/store operations on the bus

Why use DMA?

Free the CPU from byte-copying tasks

Can leverage burst on the bus, if they are available

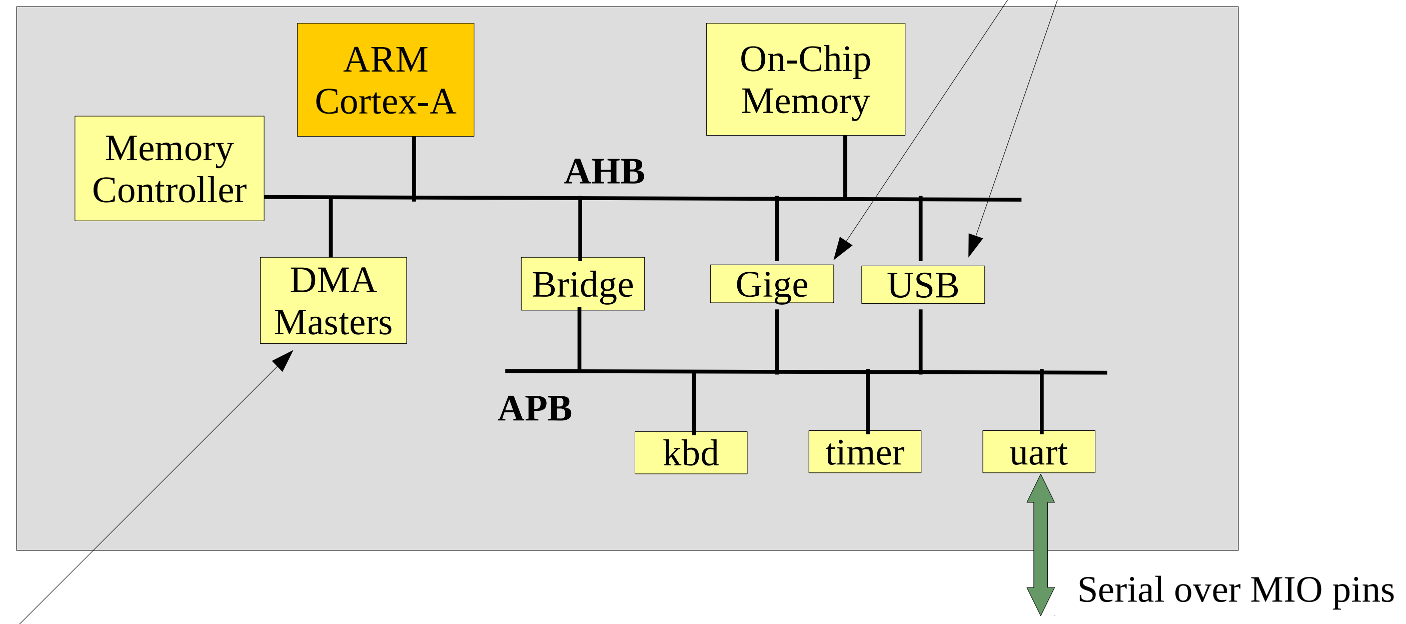
ARM – Advanced Microcontroller Bus Architecture

- Advanced Microcontroller Bus Architecture (AMBA)
  - A bus hierarchy
  - Advanced High-Performance Bus (AHB)
  - Advanced Peripheral Bus (APB)

Notice devices on both AHB and APB

Registers on APB

DMA on AHB



Eight programmable DMA on AHB

Zybo Giga Ethernet

- Giga Ethernet (Gige)
  - Scatter/gather DMA for send and receive rings
  - MMIO registers for commands and status
  - DMA is on the AHB and registers are on the APB

Serialization

Deserialization

Going out

of the processor

Across two buses...

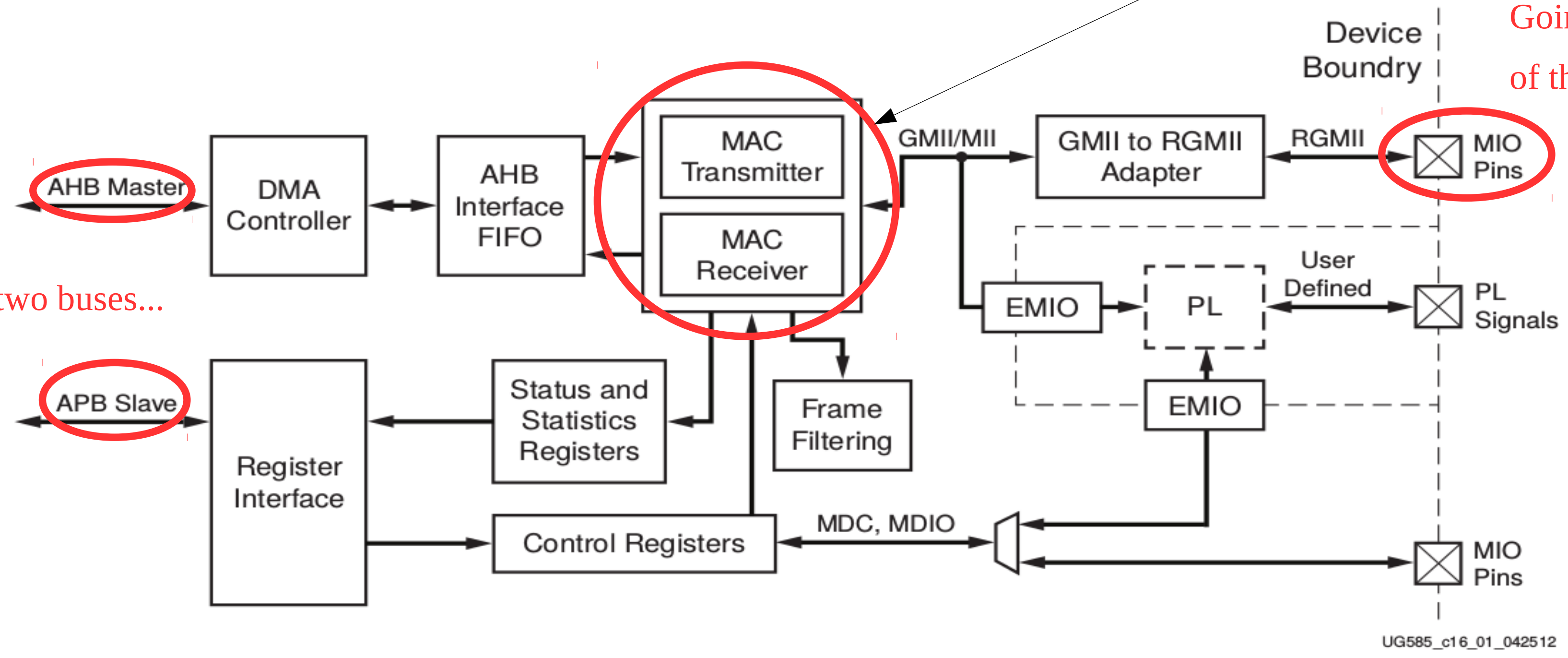
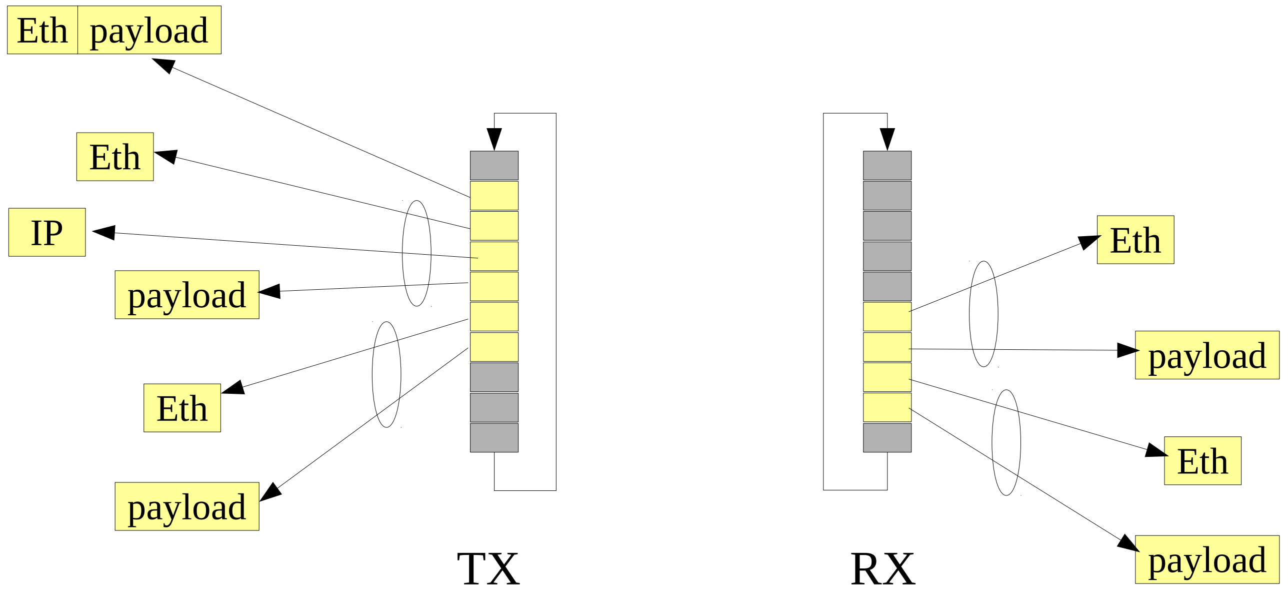


Figure 16-1: **Ethernet Controller**

MII: Media Independent Interface
GMII: Giga MMI
RGMII: Reduced GMII (fewer pins)

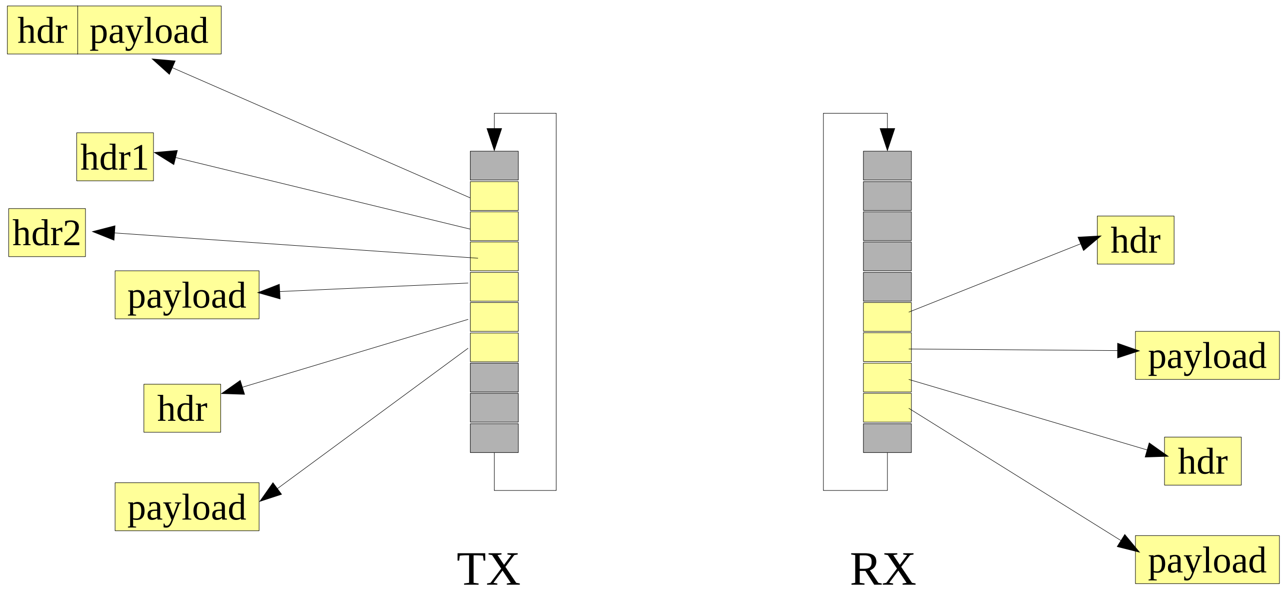
Zybo Giga Ethernet – Tx/Rx Rings

- Transfer ring (TX)
  - Gather DMA for sending frames
- Receive ring (RX)
  - Scatter DMA for received frames



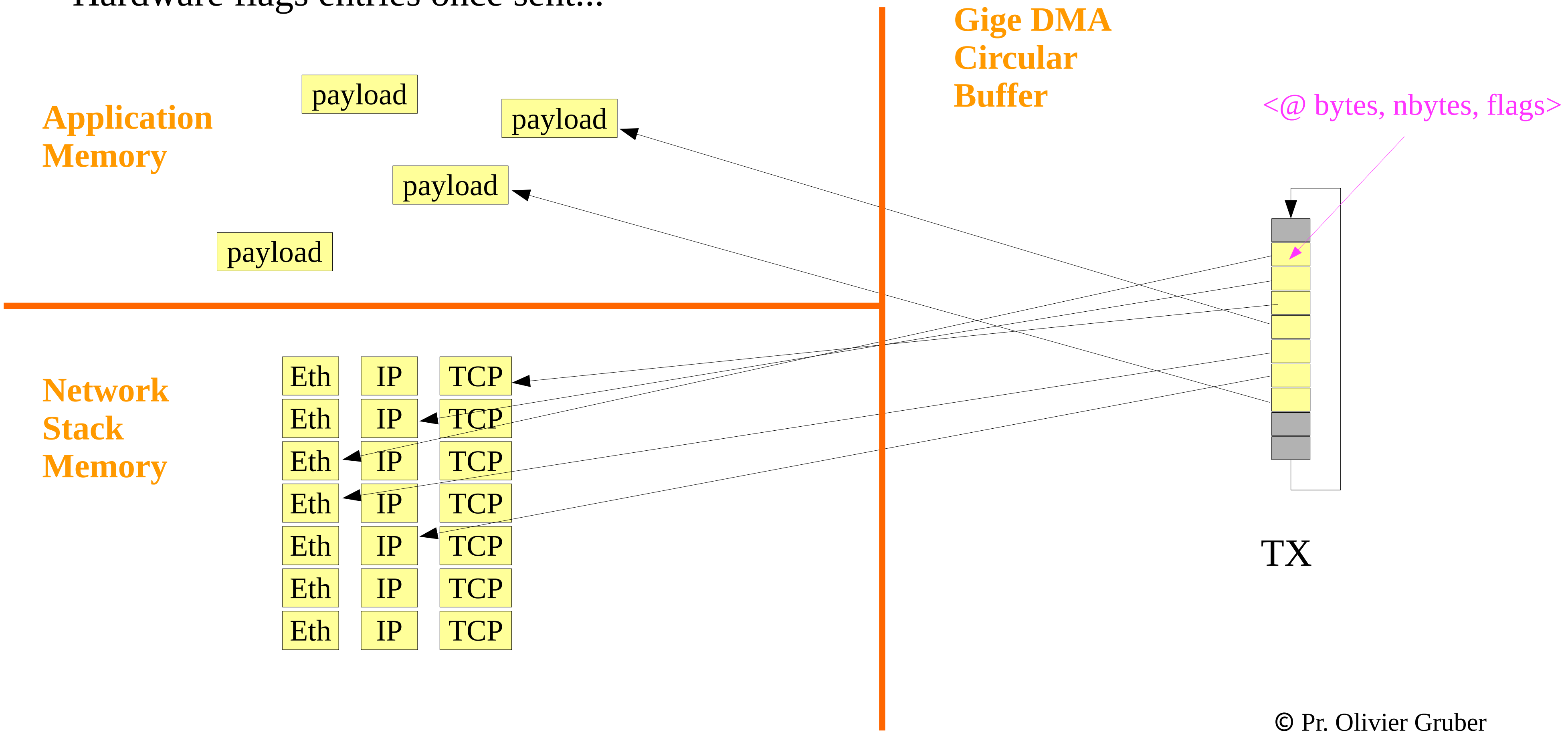
Your mind at work...

- Top/Bottom for Giga Ethernet
  - How do you manage your ring buffers?
  - What is the work done in the top?
  - What is the work done in the bottom?
  - Think about your network stack (IP/UDP/TCP)... which events to do what?



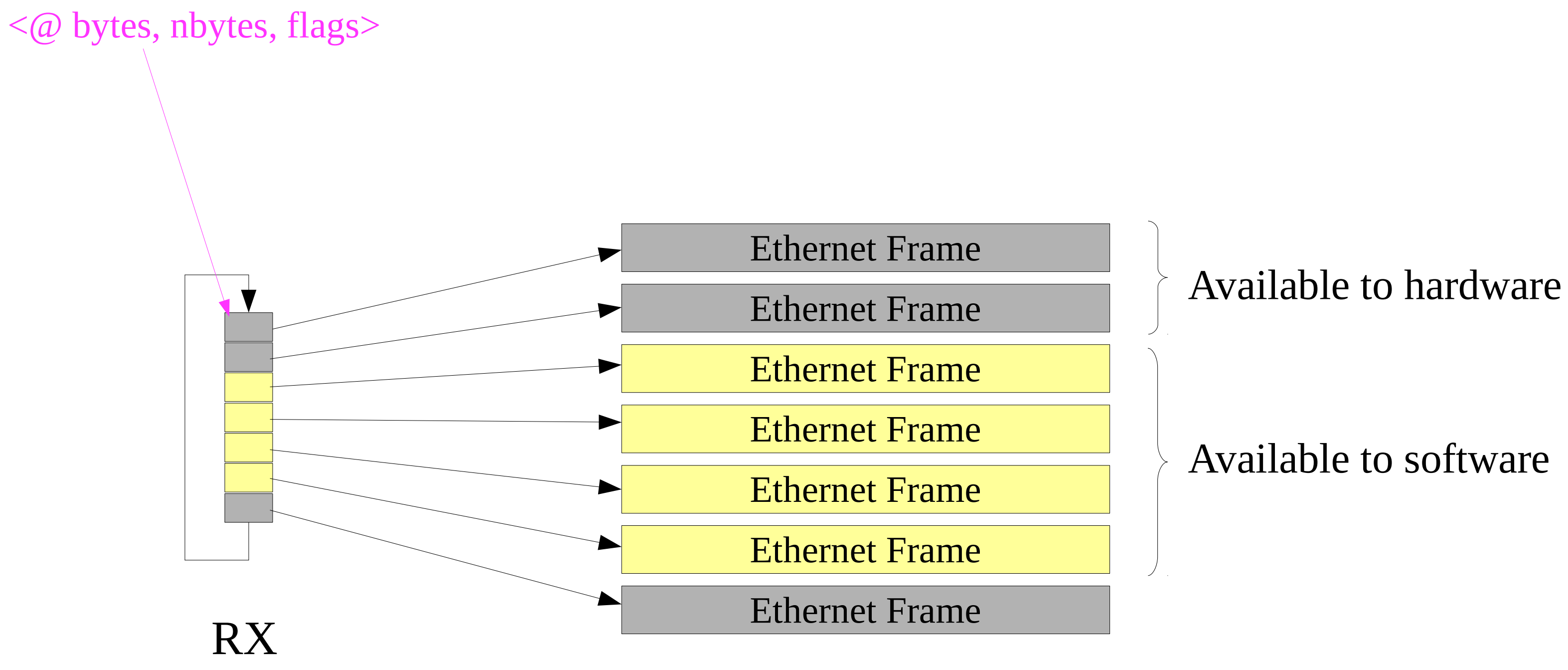
Zybo Giga Ethernet

- Transfer ring (TX)
  - Payloads to send from an application output stream
  - Grab headers (Ethernet, IP, TCP) and initialize them
  - Setup entries in the TX circular buffer
  - Hardware flags entries once sent...



Zybo Giga Ethernet

- Receive ring (RX)
  - Hardware Eth/IP/TCP checksums
  - Pre-allocated Ethernet frames (~1500 bytes)
  - One per entry in the circular buffer



Zybo Giga Ethernet

- Receive ring (RX)
  - Hardware Eth/IP/TCP checksums
  - Pre-allocated Ethernet frames (~1500 bytes)
  - One per entry in the circular buffer

Key design point:

- Aliasing received frames up the network stack?

- Copy them before passing them up?

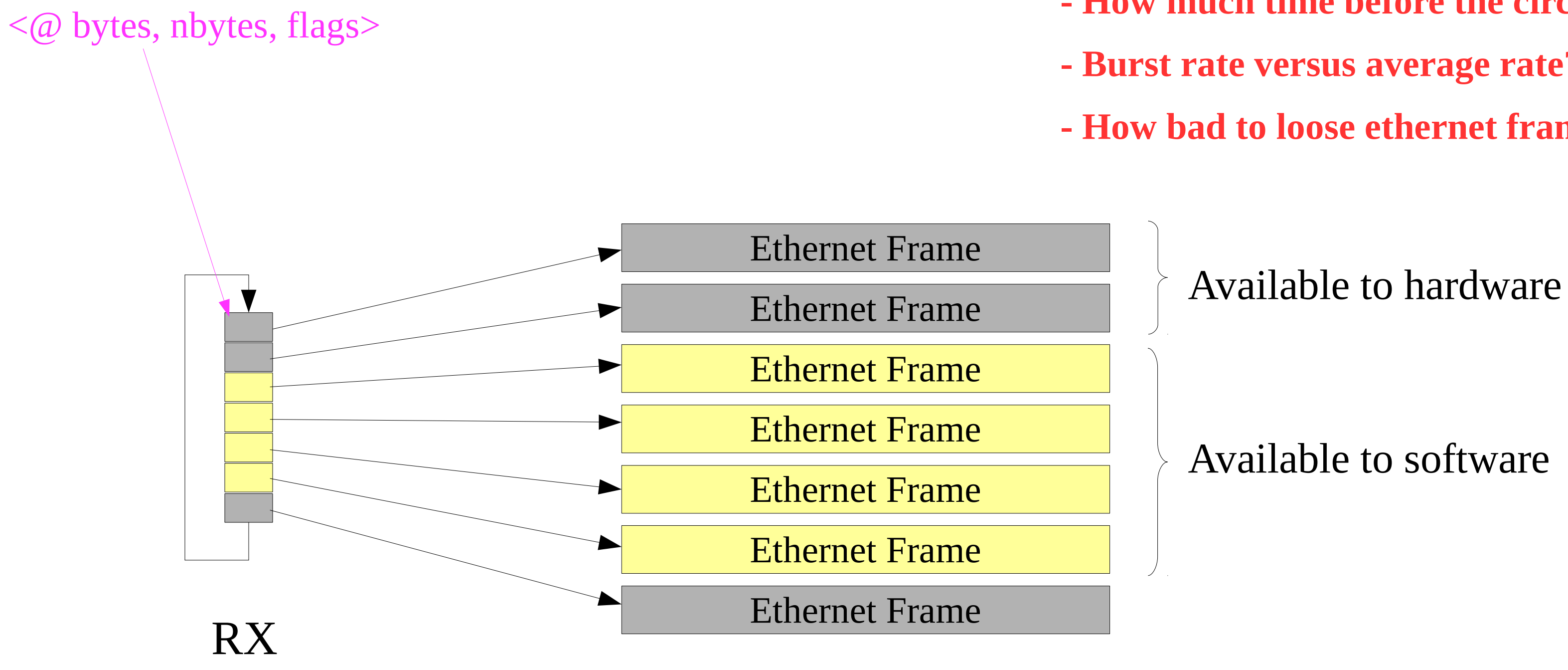
Questions:

- How fast are they processed?

- How much time before the circular buffer is full?

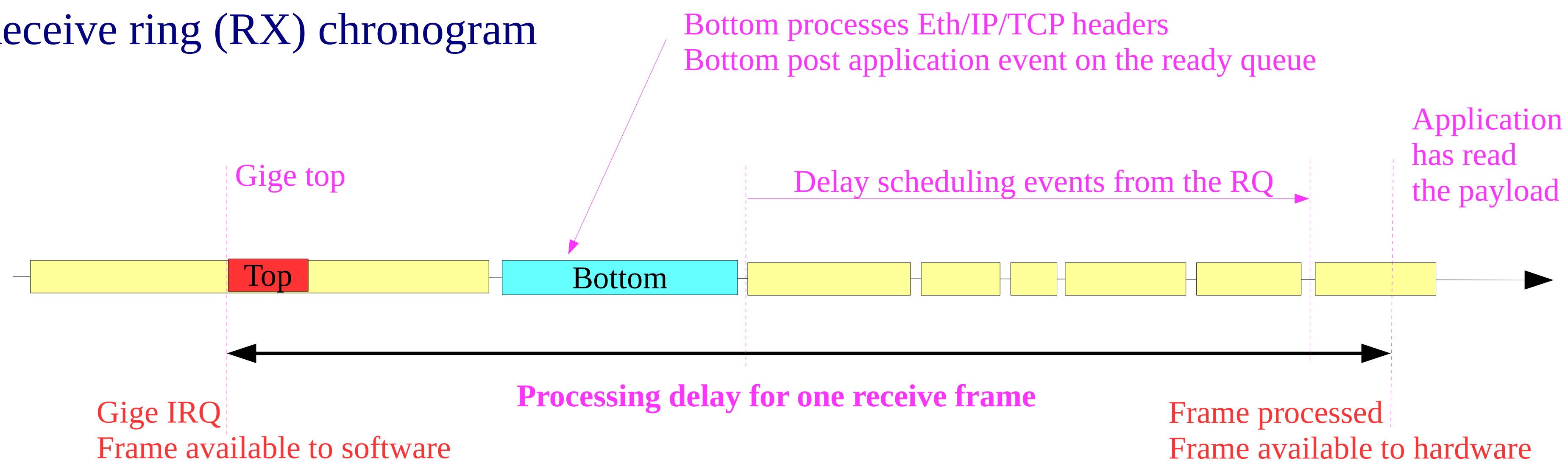
- Burst rate versus average rate?

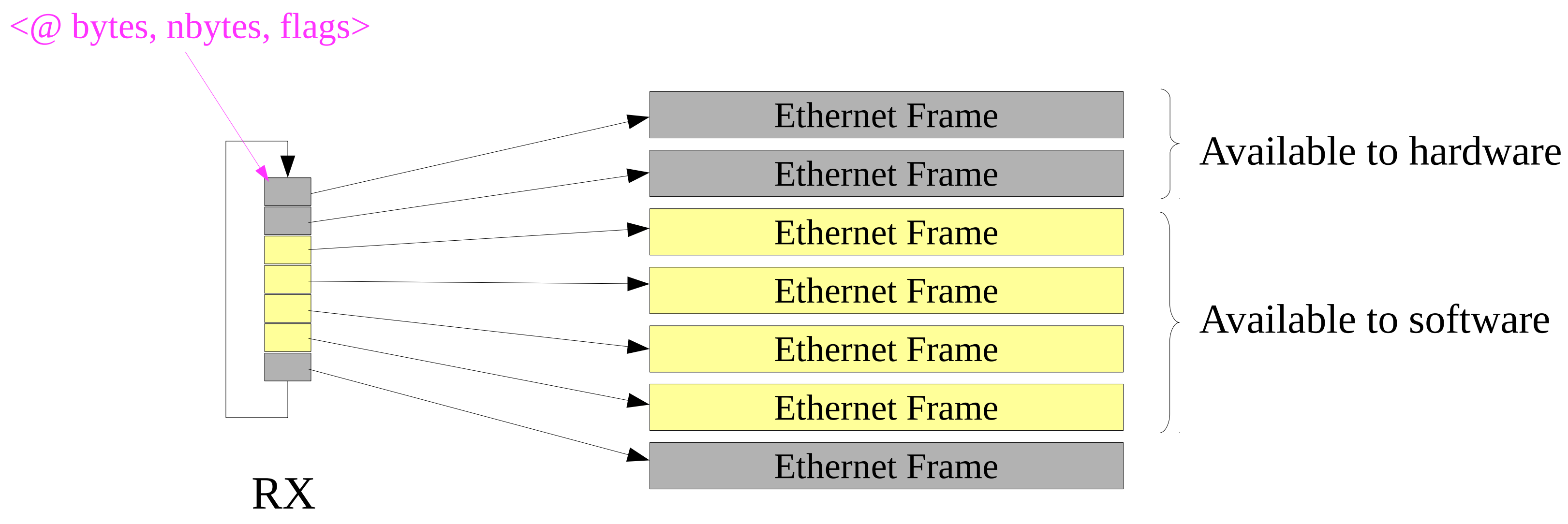
- How bad to loose ethernet frames?



Zybo Giga Ethernet

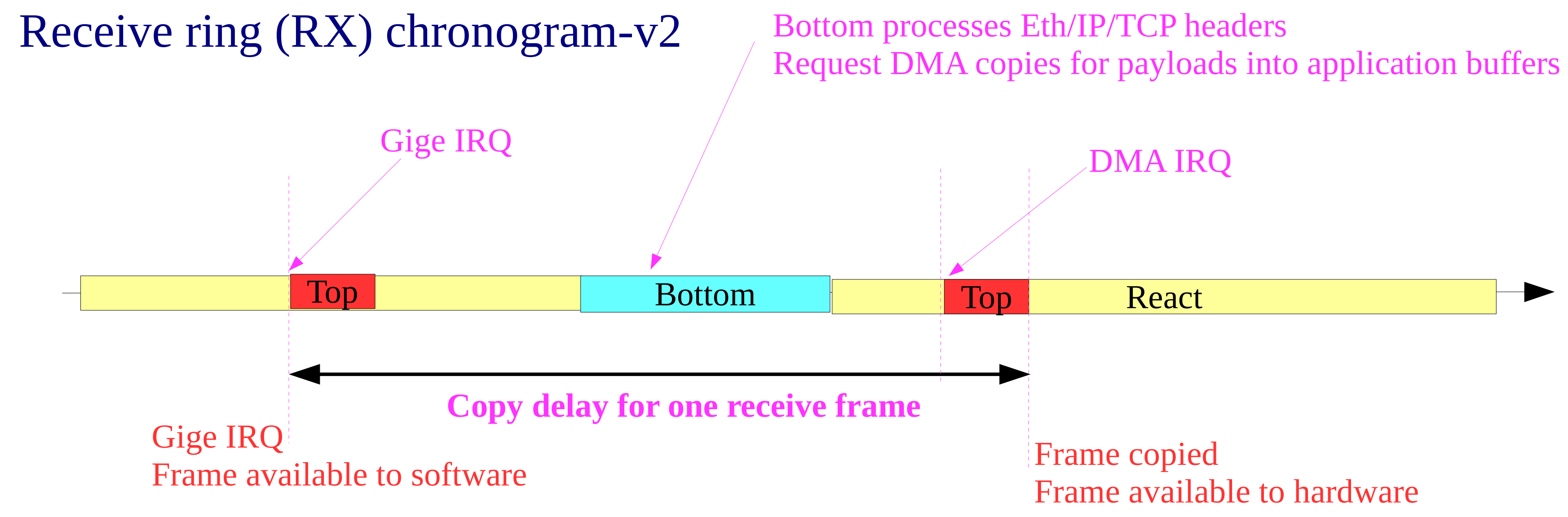
- Receive ring (RX) chronogram

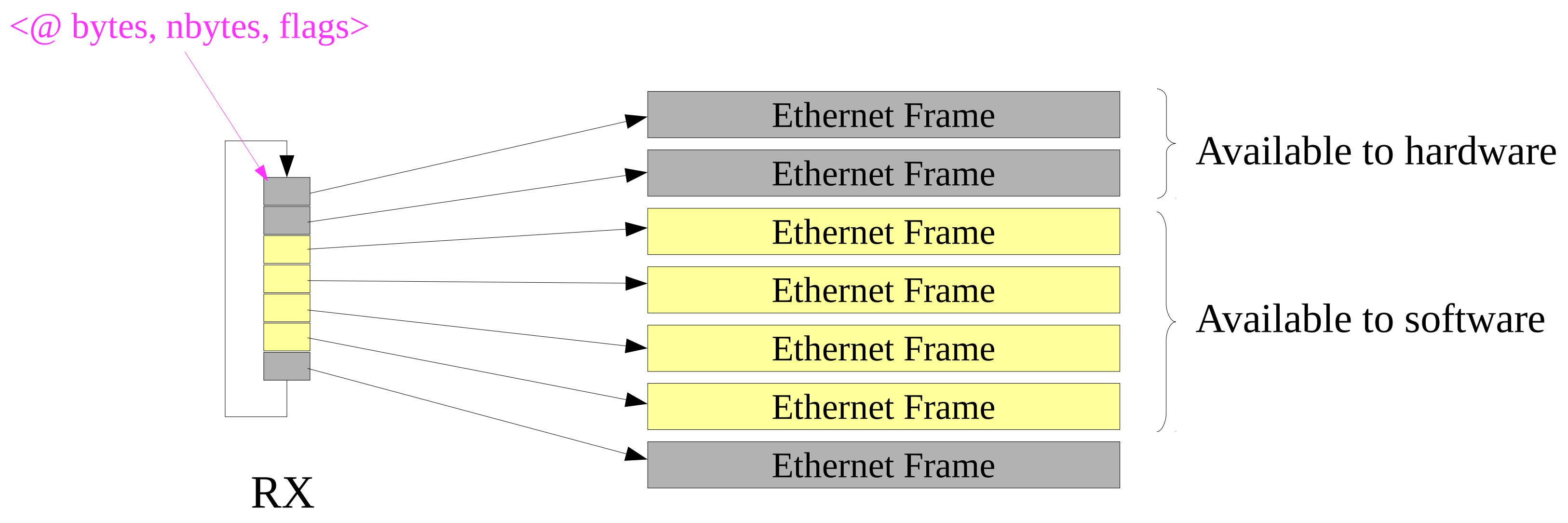

Bottom processes Eth/IP/TCP headers
Bottom post application event on the ready queue
Application has read the payload
Gige top
Delay scheduling events from the RQ
Top
Bottom
Processing delay for one receive frame
Gige IRQ
Frame available to software
Frame processed
Frame available to hardware



Zybo Giga Ethernet

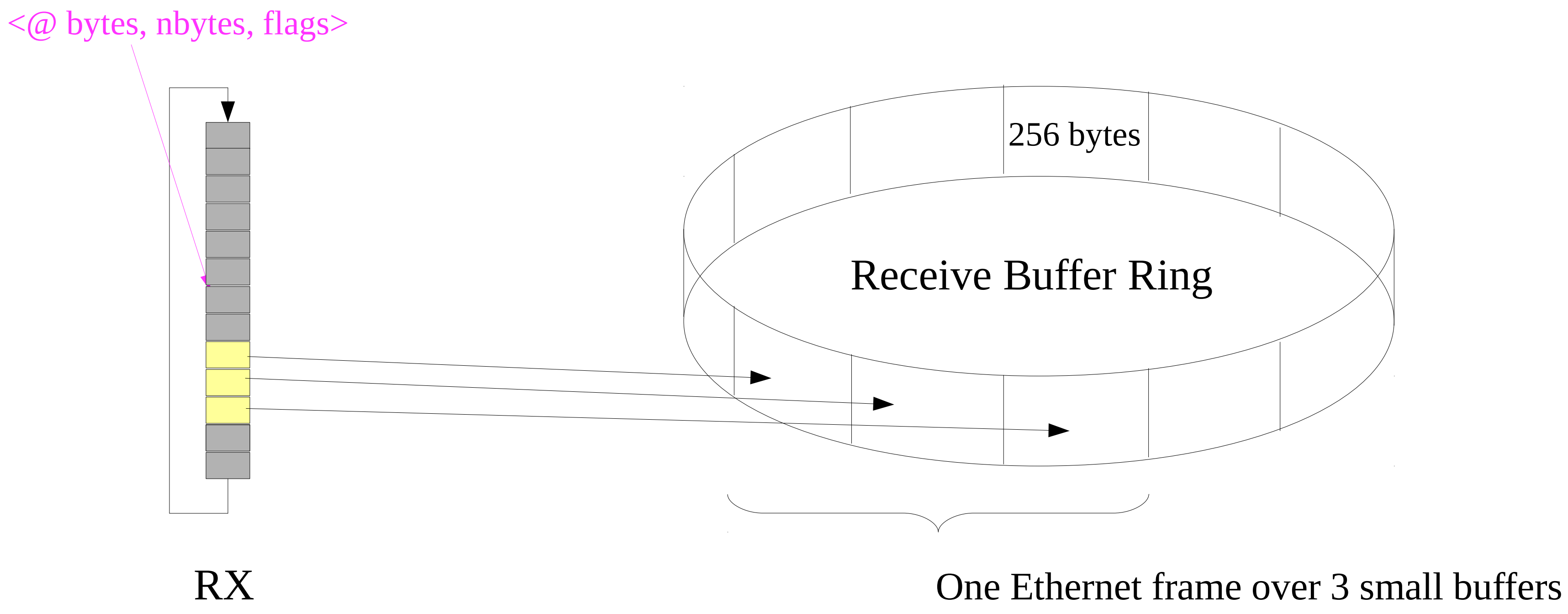
- Receive ring (RX) chronogram-v2


Bottom processes Eth/IP/TCP headers
Request DMA copies for payloads into application buffers
Gige IRQ
DMA IRQ
Top
Bottom
Top
React
Copy delay for one receive frame
Gige IRQ
Frame available to software
Frame copied
Frame available to hardware



Zybo Giga Ethernet

- Receive ring (RX) – A different organisation
  - Large RX ring of smaller buffers (256 bytes for example)
  - DMA copies one frame to one or several contiguous buffers
  - Exploits that many Ethernet frames are small
  - So can receive more frames potentially with less buffer memory



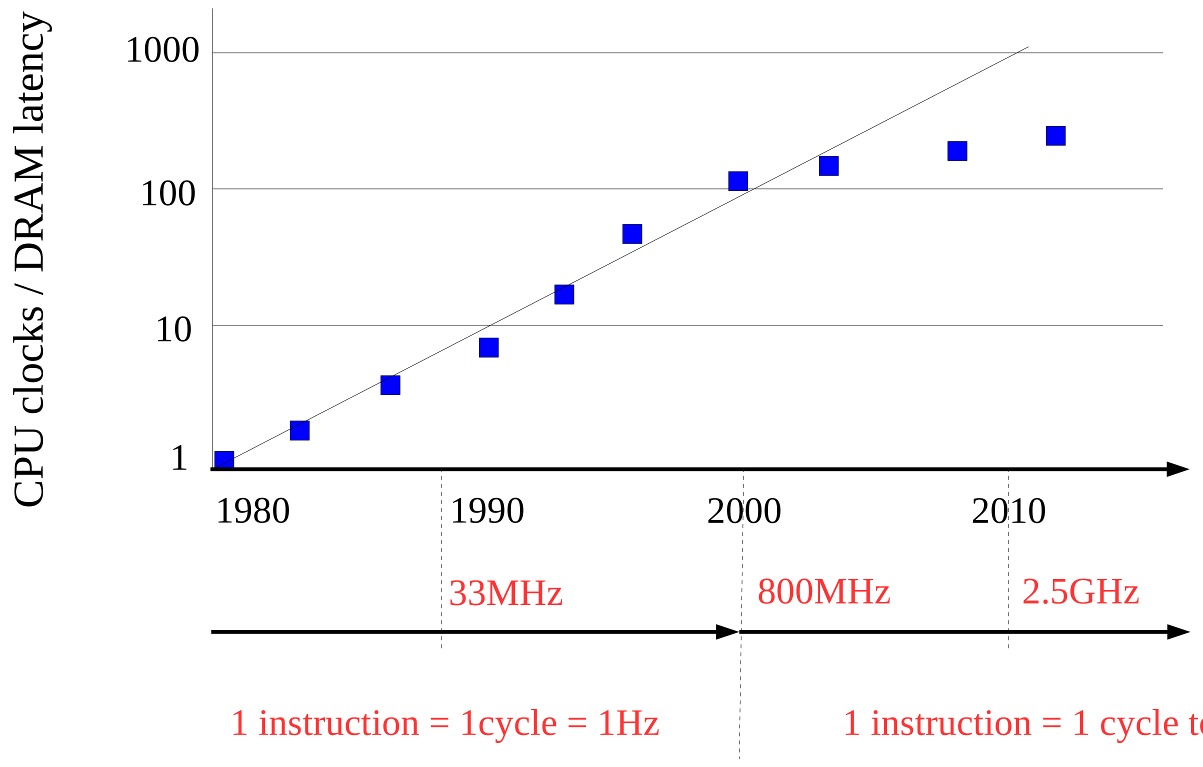
Computer Basics

- Hardware
  - Single-core CPU, with load/store interface
  - Memory controller and memory
  - Instructions: load/store, arithmetic, branches, etc.
  - CPU registers, both general purpose and special
  - Floating Point Unit (FPU), an arithmetic co-processor
  - Interrupts, **DMA**
- Software
  - Event-oriented scheduler
  - ISRs as top handler and bottom events
  - Device drivers:
    - Reading and writing mmio registers
    - **Leveraging DMA transfers**

Where is the potential performance bottleneck?

Memory Wall

- A major performance factor today
  - With latencies well over 100 cycles
  - In certain NUMA architecture, latency can be over a 1000 cycles



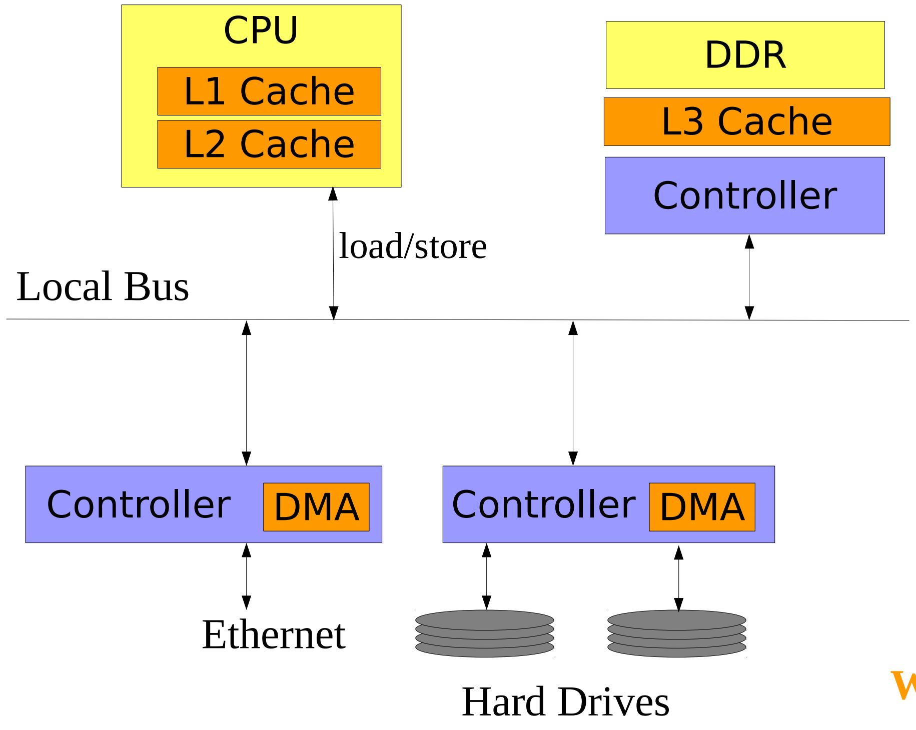
The flattening is due to the flattening of CPU clock frequency

1 instruction = 1cycle = 1Hz

1 instruction = 1 cycle to several cycles

No longer 1 instruction every cycle...

Introducing Cache Hierarchy

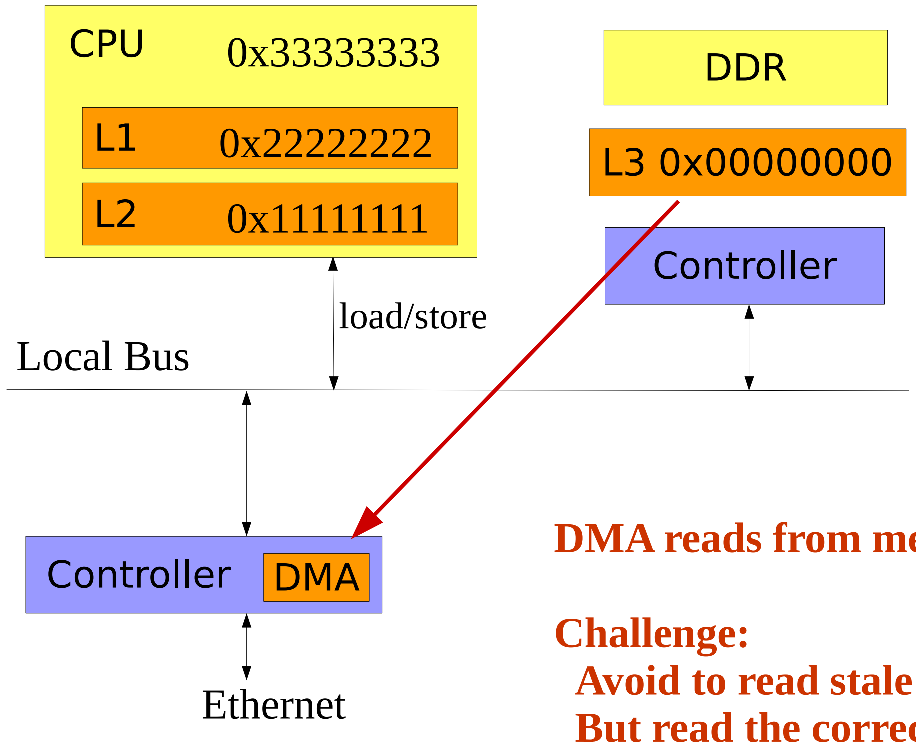


Introduce a cache,
with cache line transfers from/to memory.

- **better bus usage (burst mode)**
- **faster access time (local memory)**
  - **L1, latency around a few cycles**
  - **L2, latency around a dozen cycles**
  - **L3, latency around a hundred cycles**

WARNING:
Multi masters is delicate
Requires the ability to clean and invalidate caches

Cache Management – Clean Operation



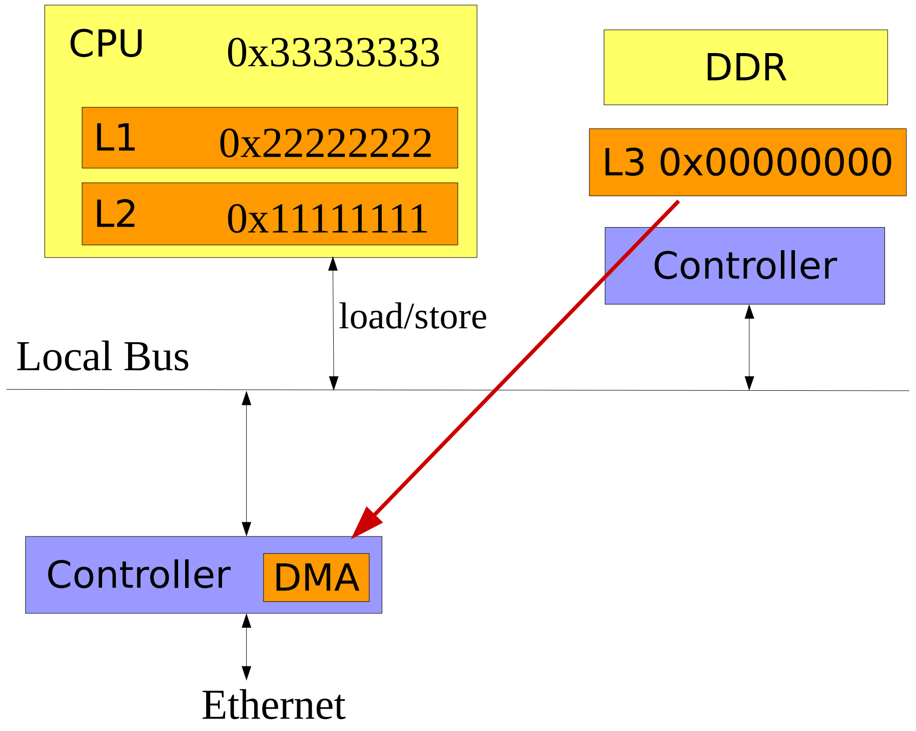
DMA reads from memory.

Challenge:
Avoid to read stale data: 0x00000000
But read the correct value: 0x33333333

Solution:

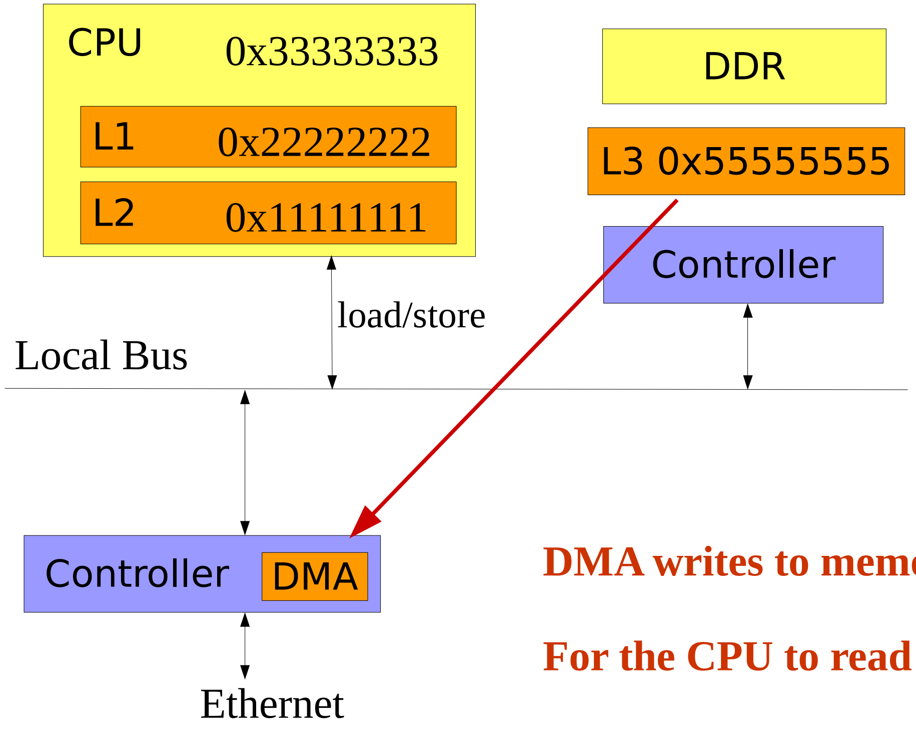
- Software must write CPU registers to memory → L1
- Software must clean L1 and L2 cache
- Software must wait until updates have reached the POC
  Point of Coherency is the DDR in this case
- Software must inform device that it can start its DMA
  Reading and writing mmio registers

Cache Management – Clean Operation



Point of Coherency is the L3 cache in this case

Cache Management – Clean Operation



DMA writes to memory: 0x55555555

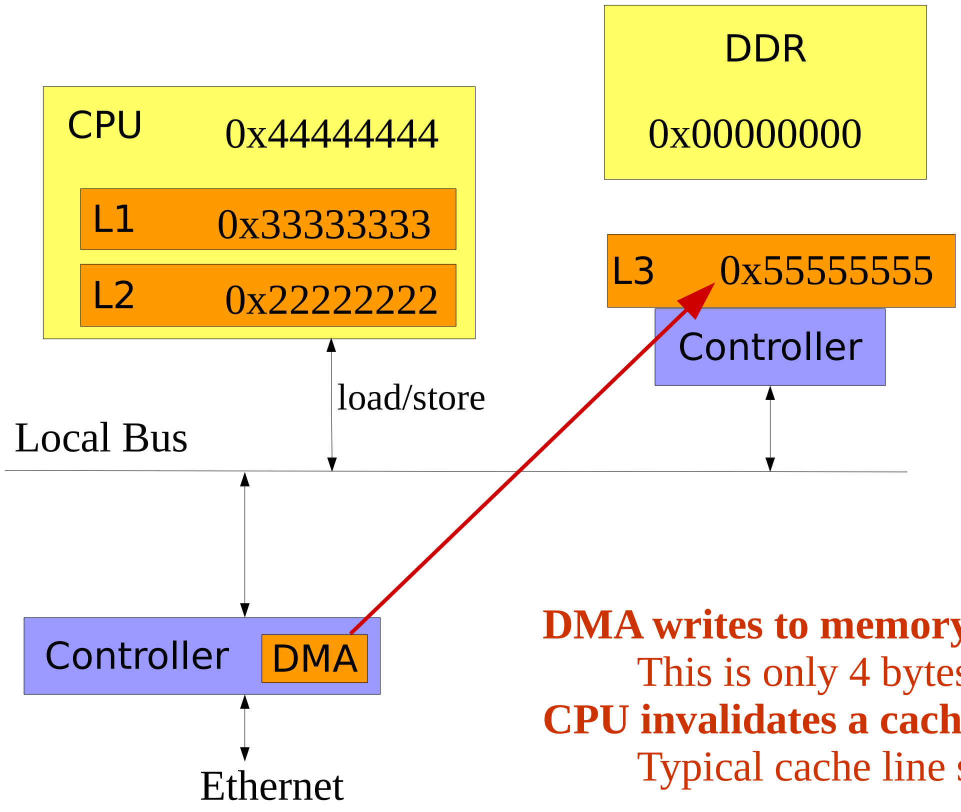
For the CPU to read the correct value:

- Software must invalidate the L1 & L2 cache
  Invalidate ! DO NOT CLEAN
- Software must make sure to read from memory (volatile variable)

Race condition:

- Follow a master/slave interaction
- Otherwise how do you know where and when a device issues a DMA

Cache Management – Invalidate Caveats



DMA writes to memory: 0x55555555

This is only 4 bytes

CPU invalidates a cache line

Typical cache line sizes are 16 or 32 bytes

Caveat:

So all updates to the entire cache line are lost!

Allocate your C structures carefully!

Computer Basics

- Hardware
  - Single-core CPU, with load/store interface
  - Memory controller and memory
  - Instructions: load/store, arithmetic, branches, etc.
  - CPU registers, both general purpose and special
  - Floating Point Unit (FPU), an arithmetic co-processor
  - Interrupts / DMA
  - Caches
- Software
  - Event-oriented scheduler
  - ISRs as top handler and bottom events
  - Device drivers

Question:

Do we have something useful?

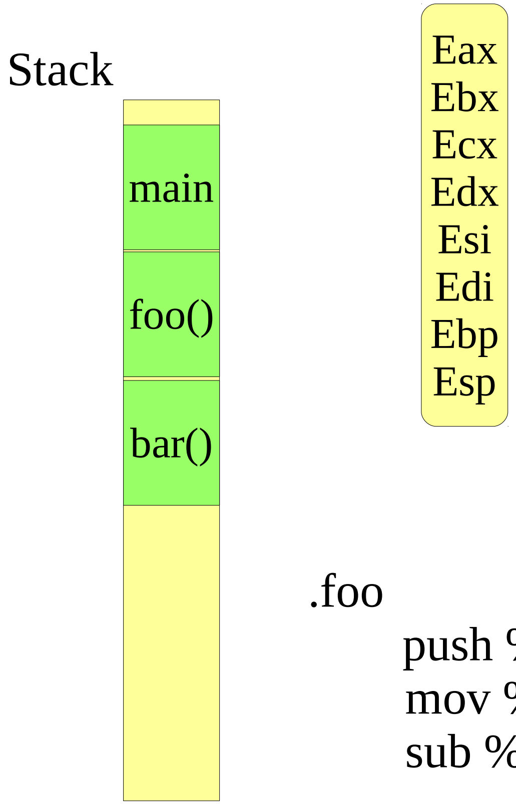
Of course we do!

For completeness:

Let's discuss threads and processes

Computer Basics – Threads

- Threads are virtualizing cores
  - A context in memory
    - A set of CPU registers,
    - A set of FPU co-processor registers
    - Maybe other things such as thread-local variables
  - A stack in memory
    - Used for stack frames
- Thread scheduling
  - Multiple threads can be scheduled on a single core
    - Notice that a process is not required
    - Threads share the same physical memory
  - It is mostly about a programming model choice
    - **Later, 30 years later,** it became a way to exploit multi-core...



```
.foo
      push %ebp
      mov %ebp,%esp
      sub %esp,#0x10

      <foo code>

      mov %esp,%ebp
      pop %ebp
      ret
```

Computer Basics – Processes

- Processes are virtualizing machines
  - One or more threads ← Virtualizing cores
  - Virtual memory ← Virtualizing physical memory
  - Signals ← Virtualizing traps and interrupts
- Processes for several reasons
  - Memory isolation (safety, separation of concerns)
  - Security (access rights)
  - Paging (virtual memory larger than physical)
  - Management concept (kill, resource accounting)

Computer Basics – Processes

- Processes require specific hardware support
  - Memory Management Unit (MMU)
    - Architected Translation Look-aside Buffer (TLB)
    - Architected page tables with TLB
  - Processor modes
    - Privileged instructions vs non-privileged
    - Traps
- Traps vs Interrupts
  - A trap occurs as a side effect of the execution of an instruction
    - Corresponds to *exception conditions*
    - *S*uch as arithmetic overflows, page faults, or violations of memory-access priviledges
  - Interrupts are caused by the occurrence of external events
    - Interrupts are not related to the execution of specific instructions
    - Interrupts are typically generated by controllers or subsystems like the timer

Computer Basics – Processes

- A process is a complete virtual machine
  - Privileged processes – Kernel mode with MMU turned on
  - Unprivileged processes – User mode with MMU turned off
- Privileged process
  - See the processor cores
    - Full set of registers
    - Full set of instructions
  - See virtual memory instead of physical memory
    - Full 32bit or 64bit address space
  - See devices
    - Full access to mmio registers
    - Full access to interrupt vector

Computer Basics – Processes

- Non-privileged process
  - See the processor cores
    - Full set of registers
    - Full set of instructions minus the privileged ones
    - The **syscall** instruction as a gate to the privileged world
  - See virtual memory instead of physical memory
    - Full 32bit or 64bit address space
    - Cannot change its own virtual memory mapping
  - Usually does not see device directly
    - mmio registers are not mapped in the address space
    - interrupt and trap vector are not mapped in the address space

Conclusion

- A slow co-design over 40 years
  - To invent the future, one must know the past
    - Concepts have been invented for something, in a given context
    - Then the context evolves, but concepts remain, resisting change
    - This is especially true of operating systems because they are the foundation
  - Avoid being religious
    - There is no point, software and hardware are about building tools
    - Like tools, there are many shapes that correspond to many usage
    - Avoid being a Linux bigot and bash Windows
    - Avoid promote C over everything else
  - Who knows what the next generation operating system will be like...
    - Microkernels were supposedly flawed – Mac-OS relies on one
    - Java was just good for Applet – Android is based on Java
    - So many more examples...