Hardware Design with Zynq® FPGA #2: How Can SW Access HW Component?

Jae W. Lee (jaewlee@snu.ac.kr)
Department of Computer Science and Engineering
Seoul National University

Slide credits: Prof. Sungjoo Yoo (Seoul National University)

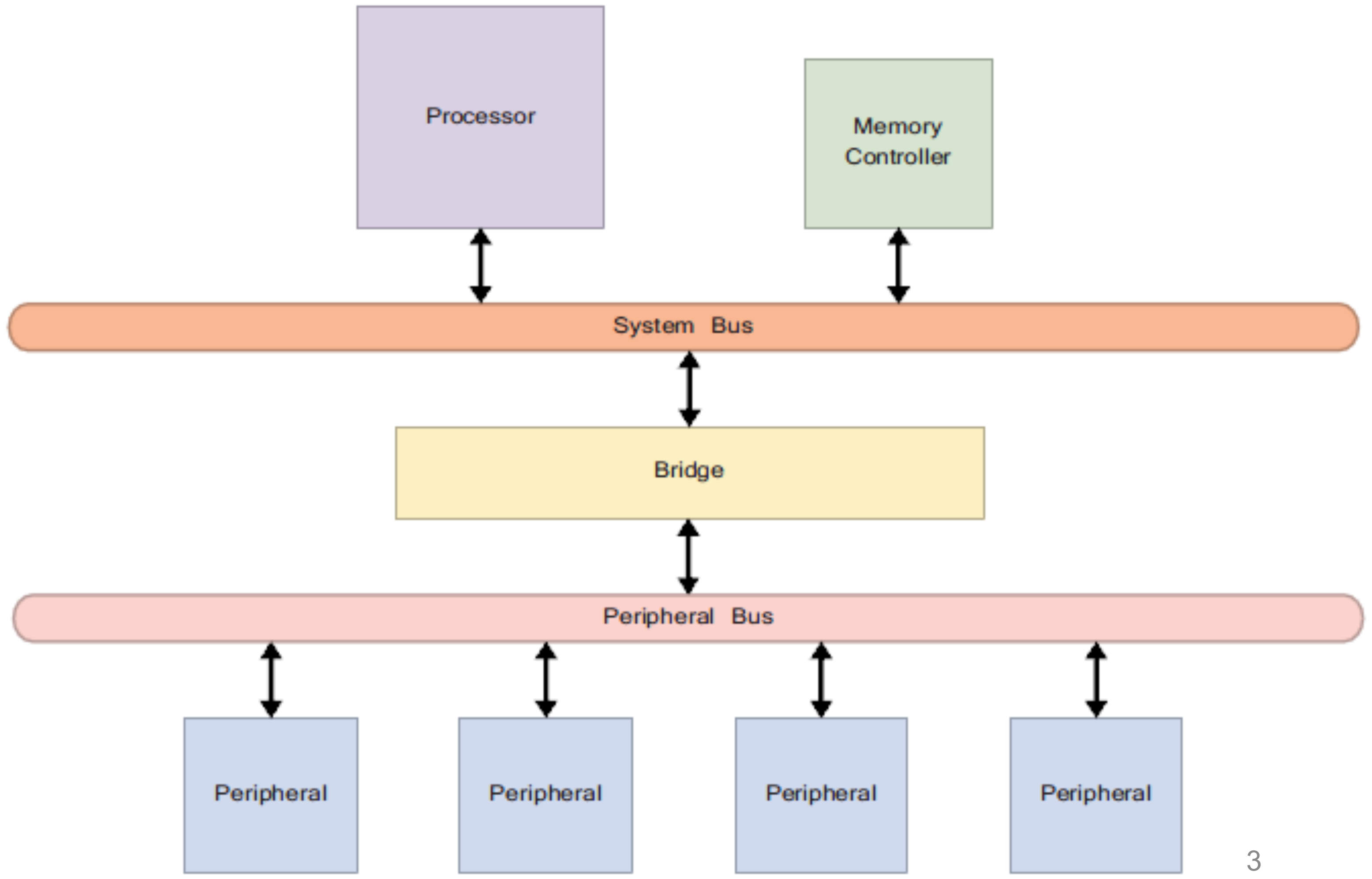
Outline

- How can software access a hardware component?
  - Memory-mapped I/O
  - What happens when executing a load instruction?
    - Virtual-to-physical memory mapping
    - Translation Lookaside Buffer (TLB)
    - Multi-level page table
- How can hardware components access the main memory?
- Overview of Lab 6 and 7

[The Zynq Book, 2014]

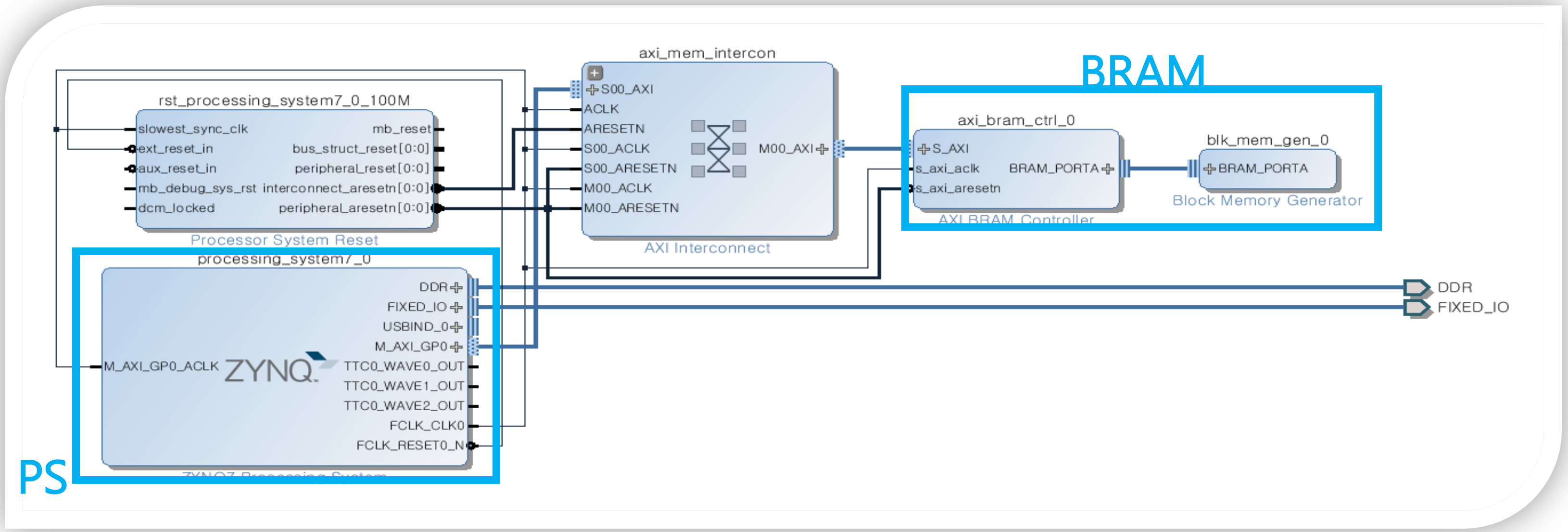
A Simplified View of Embedded System HW

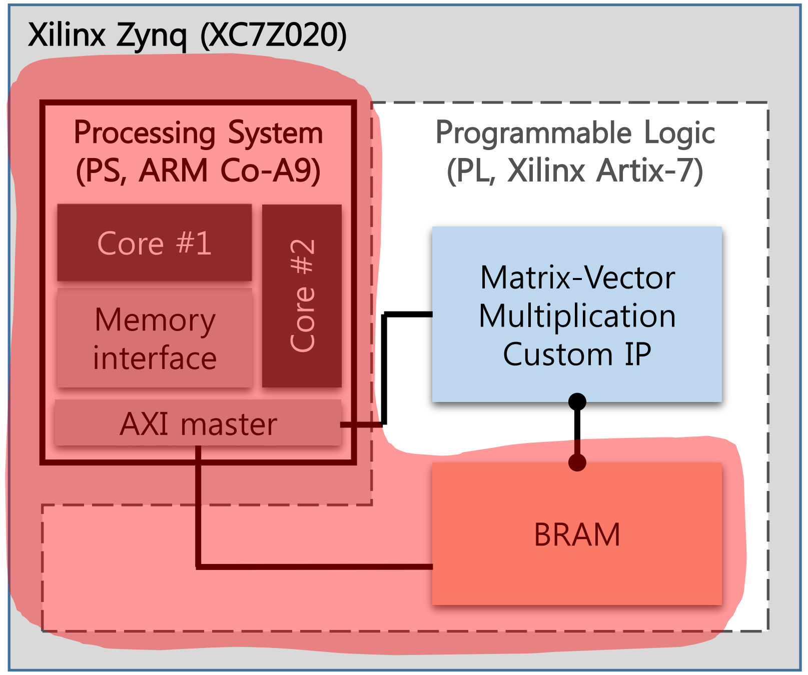
- Processor
- Memory Controller
- Peripherals
- System Bus
- Peripheral Bus



Memory-Mapped I/O: CPU and BRAM on PL

- Connecting CPU and BRAM (Block Random Access Memory)
  - CPU in PS & AXI interconnect + BRAM controller + BRAM in PL
- Our question
  - How our software accesses BRAM?





Memory-Mapped I/O: Communication in HW System

- How does my software communicate with a hardware component?
  - **Option 1: Consider the hardware component as a part of memory**
    - a.k.a. memory-mapped I/O, i.e., load/store instructions from/to a physical address in memory
  - Option 2: Consider it as a device
    - Device driver is used

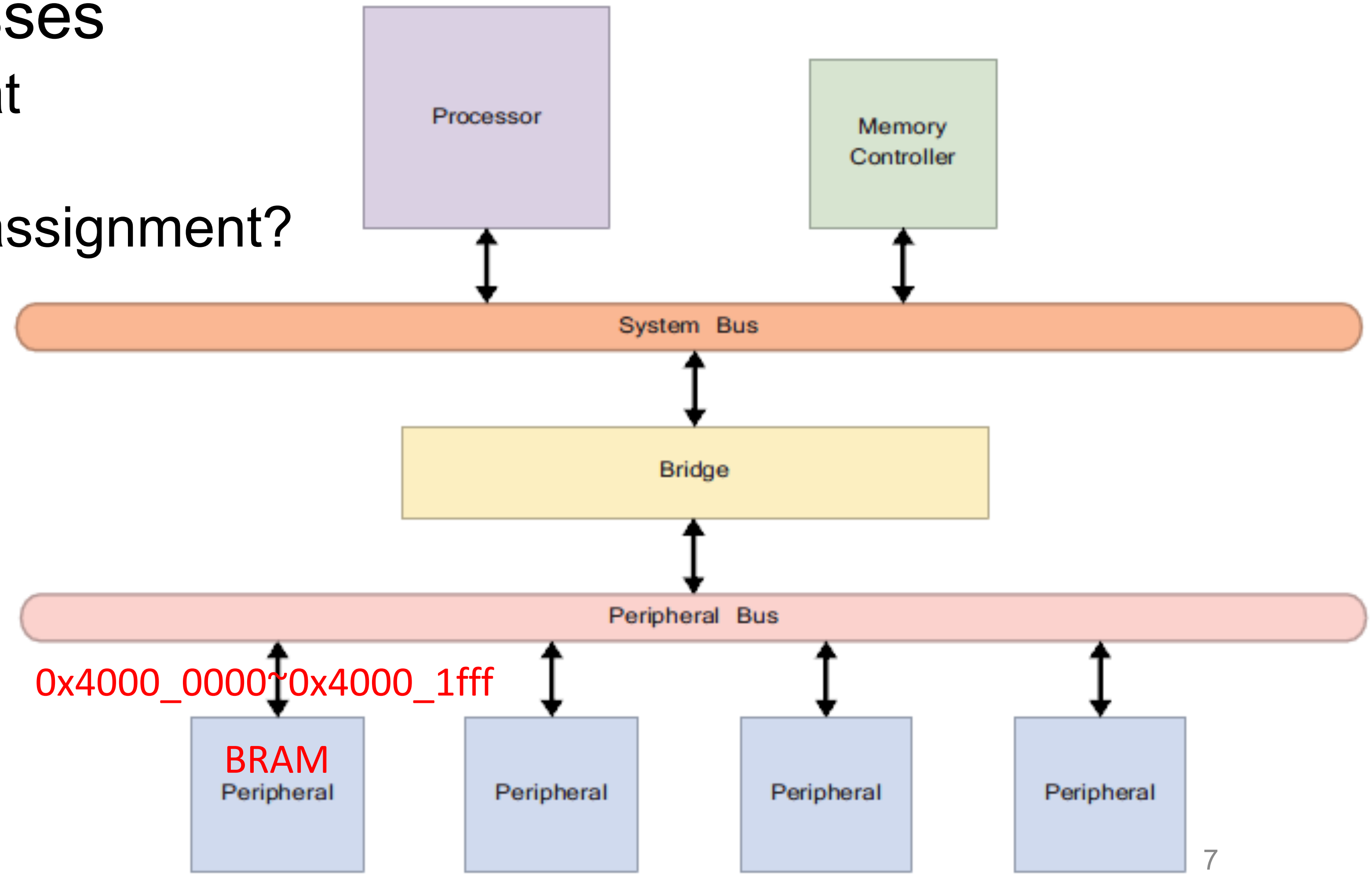
Memory-Mapped I/O: Communication in HW System

- Quick Answer: Physical memory of BRAM and `mmap()`
  - BRAM is mapped @ physical address 0x4000_0000 ~ 0x4000_1FFF (8KB)
  - Cf. Off-chip DRAM (BD.IC25/26) is @ address 0x0000_0000 ~ 0x3FFF_FFFF (1GB)

[The Zynq Book, 2014]

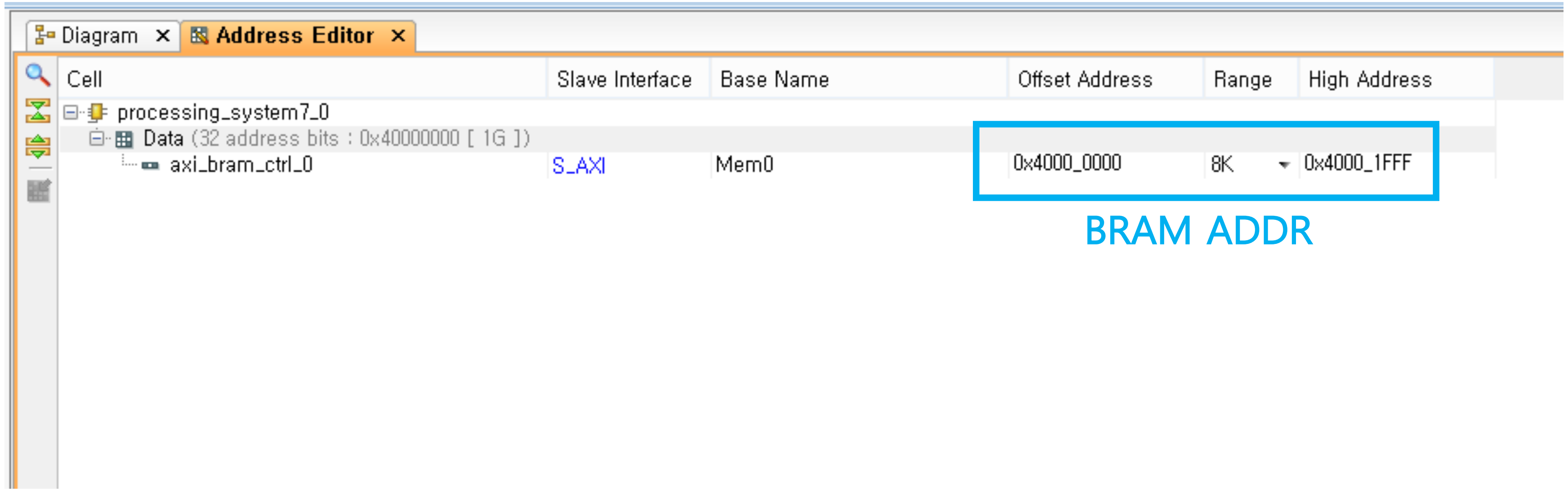
Memory-Mapped I/O: Communication in HW System

- Hardware components are assigned their own physical addresses
  - e.g., BRAM region starts at 0x4000_0000.
  - Question: Who does this assignment?
    - System designer!



Memory-Mapped I/O: Communication in HW System

- Quick answer: BRAM is @ address 0x4000_0000 ~ 0x4000_1FFF
  - Cf. DRAM (BD.IC25/26) is @ address 0x0000_0000 ~ 0x3FFF_FFFF
- System call `mmap()` can be used to create a virtual-to-physical address mapping
  - int foo = open("/dev/mem", O_RDWR);
  - int *ptr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF

Memory-Mapped I/O: Communication in HW System

- Software code using mmap() to access BRAM

```
int foo = open("/dev/mem", O_RDWR);
// Given a pathname for a file, open() returns a file descriptor
// 'dev/mem' refers to the system's physical memory
// O_RDWR means both readable and writable access mode
int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);
// mmap() creates a new mapping in the virtual address space of the calling process
// NULL means that the kernel chooses the address for mapping
// SIZE specifies the length of the mapping
// PROT_ arguments describe the memory protection (RD/WR)
// MAP_SHARED makes updates visible to other processes
// foo indicates the file descriptor to be mapped
// 0x4000_0000 refers to offset of the file descriptor → physical address for BRAM
```

```
for (i = 0; i < SIZE; i++)
  *(fpga_bram + i) = (i * 2);
// write arbitrary data on the BRAM area
printf("%-10s%-10s\n", "addr", "FPGA(hex)");
for (i = 0; i < SIZE; i++)
  printf("%-10d%-10X\n", i, *(fpga_bram + i));
// read and show the data to check if BRAM's working correctly
```

Decomposing Load Instruction Execution

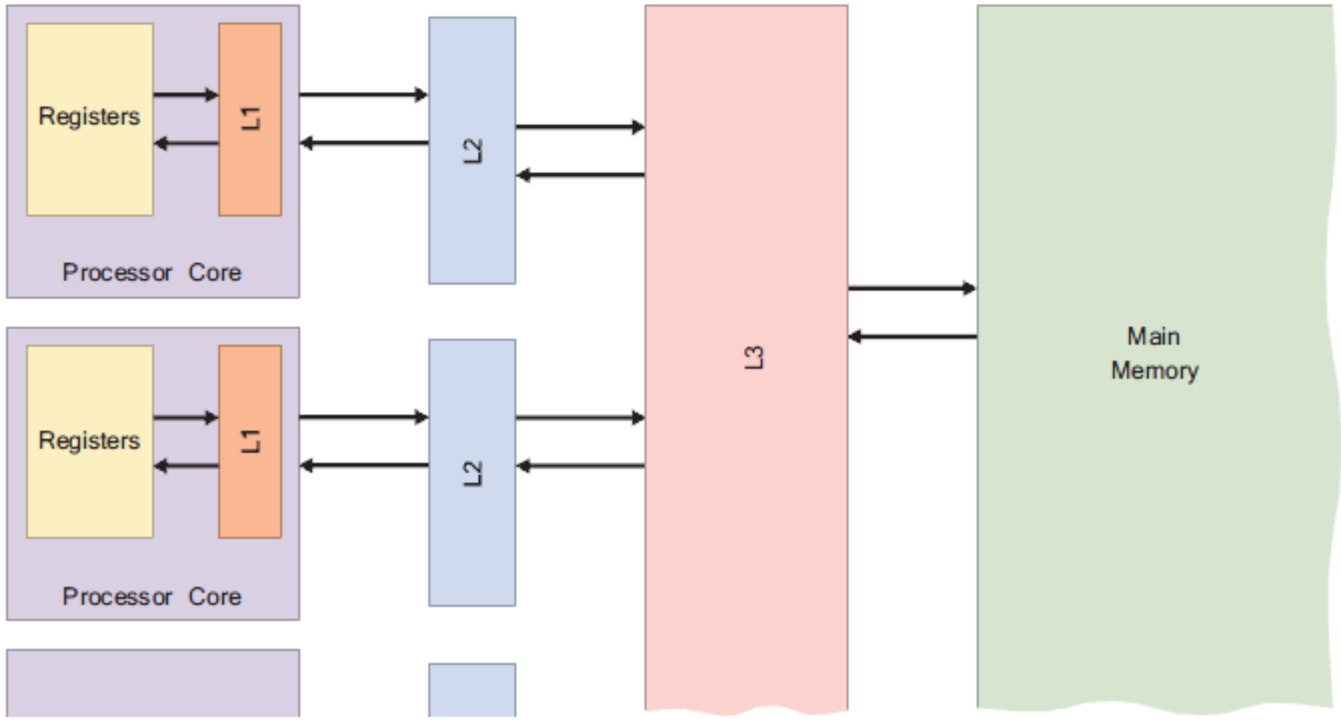
- Reading from a hardware component (e.g., BRAM)
  - Step 1: Load unit accesses L1 data cache
    - TLB access for virtual address to physical address translation
    - Non-cacheable access for hardware components other than main memory

[The Zynq Book, 2014]

Decomposing Load Instruction Execution

<u>Memory hierarchy</u>

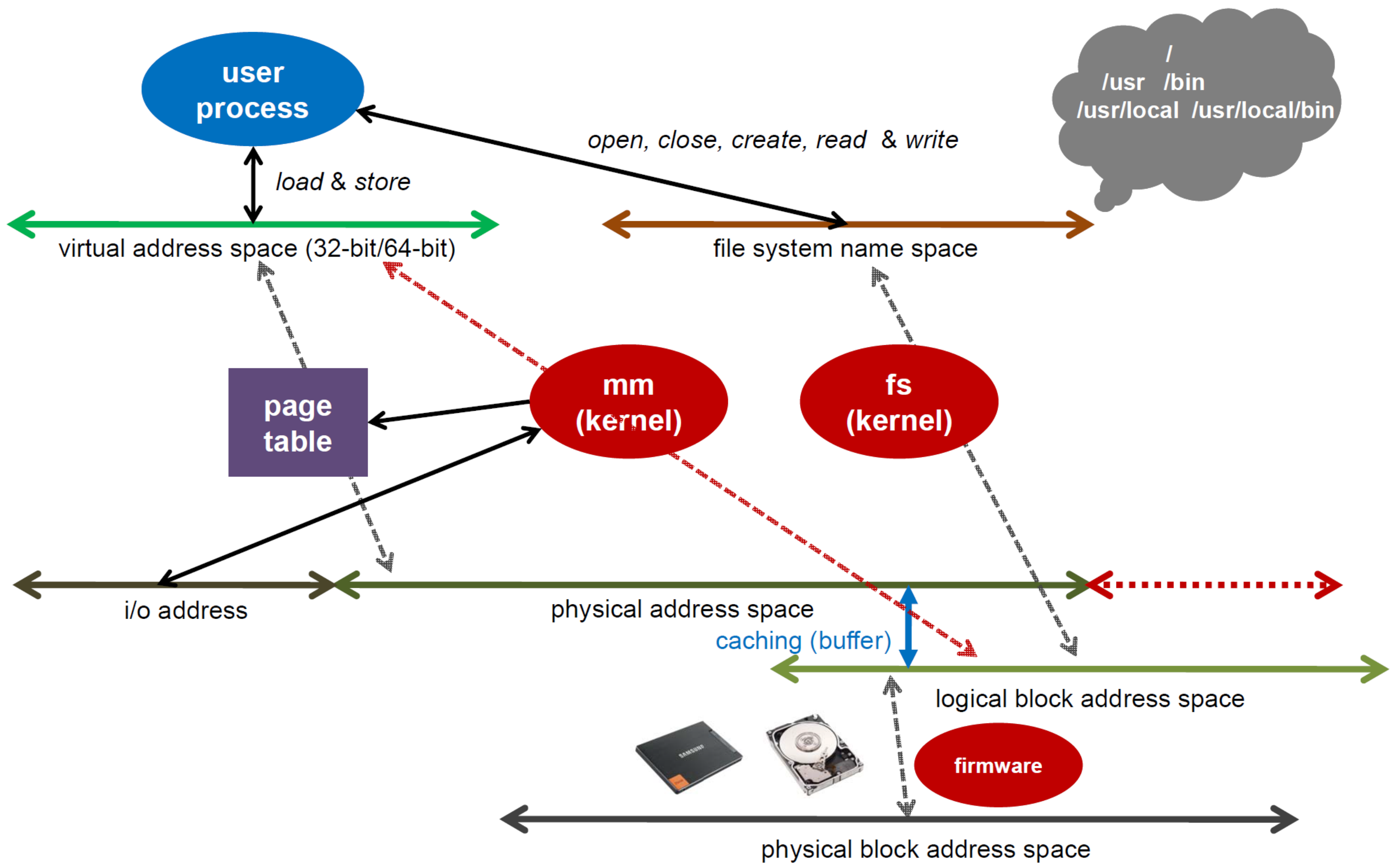
- Main memory
  - Dynamic RAM (DRAM)
- Cache
  - Static RAM (SRAM)
  - L1 cache
    - 1~2 clock cycles, ~32KB
    - Instruction (I) cache, data (D) cache
  - L2 cache
    - ~10 clock cycles, 100KB~1MB
    - Shared I+D
  - L3 cache
    - ~50 clock cycles, 1MB~10MB
    - eDRAM for better area efficiency in IBM PowerPC



[Cho, 2015]

Decomposing Load Instruction Execution

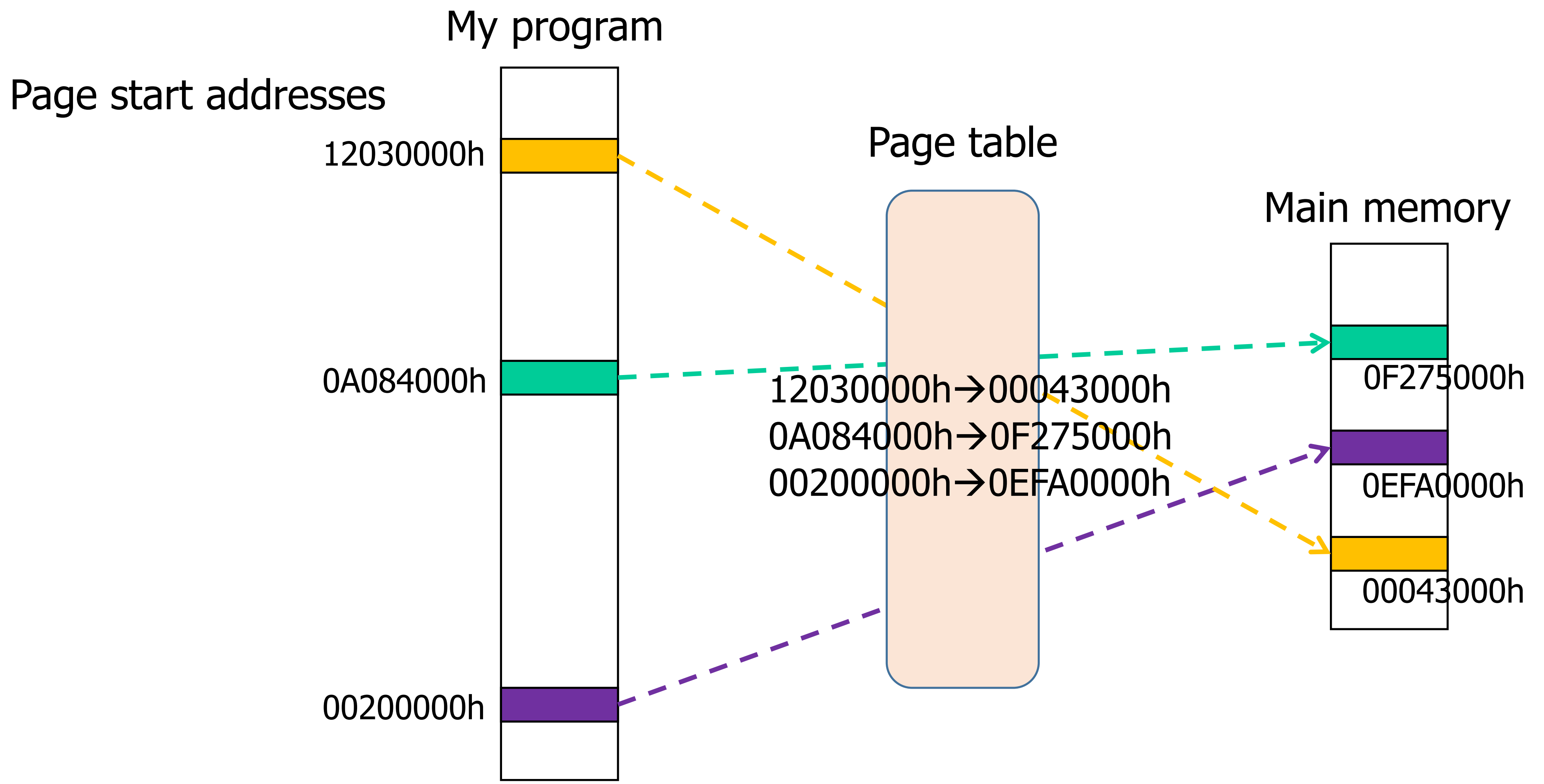
- Memory/storage address space



user process
load & store
open, close, create, read & write
/
/usr /bin
/usr/local /usr/local/bin
virtual address space (32-bit/64-bit)
file system name space
page table
mm (kernel)
fs (kernel)
i/o address
physical address space
caching (buffer)
logical block address space
firmware
physical block address space

Decomposing Load Instruction Execution

- Page table for VA-to-PA translation



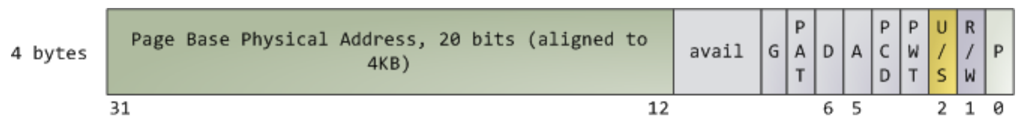
[Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 9]

Decomposing Load Instruction Execution

- Page table

<u>Hierarchical page table</u>
In reality, the page table is constructed in a hierarchical manner

A virtual page is the unit of memory protection because all of its bytes share the U/S and R/W flags



<u>PTE (page table entry)</u>
P: presence, R/W: read only or not
U/S: user/supervisor
A: accessed, D: dirty
G: granularity (4KB or 4MB)

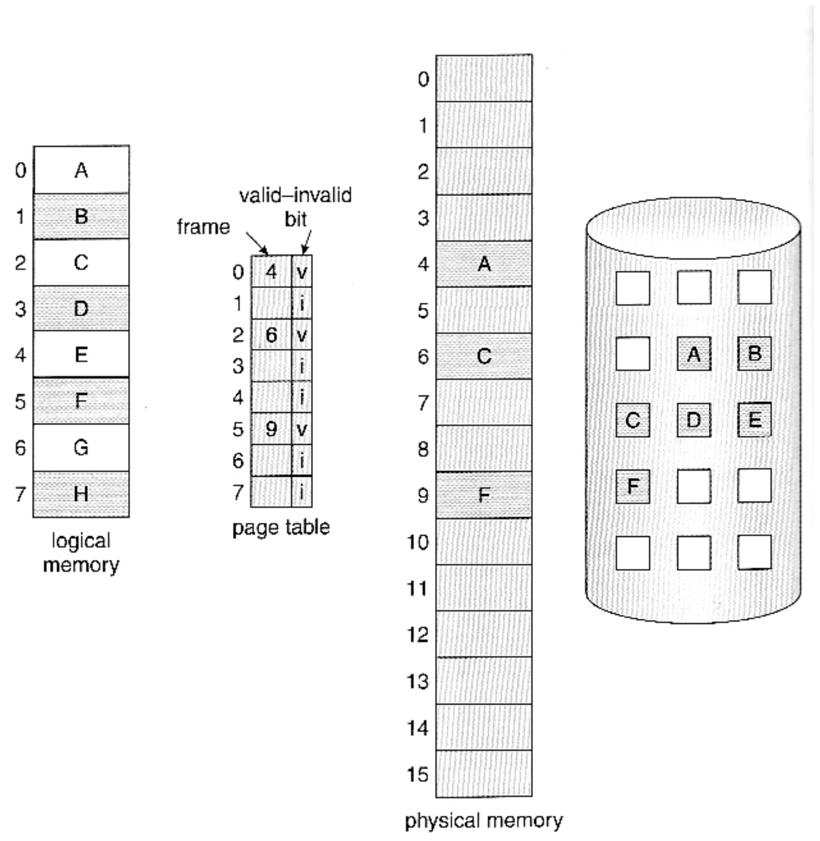


Figure 9.5 Page table when some pages are not in main memory.

Decomposing Load Instruction Execution

- Software code using mmap() to access BRAM

```
int foo = open("/dev/mem", O_RDWR);
// Given a pathname for a file, open() returns a file descriptor
// 'dev/mem' refers to the system's physical memory
// O_RDWR means both readable and writable access mode
int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);
// mmap() creates a new mapping in the virtual address space of the calling process
// NULL means that the kernel chooses the address for mapping
// SIZE specifies the length of the mapping
// PROT_ arguments describe the memory protection (RD/WR)
// MAP_SHARED makes updates visible to other processes
// foo indicates the file descriptor to be mapped
// 0x4000_0000 refers to offset of the file descriptor → physical address for BRAM
```

fpga_bram

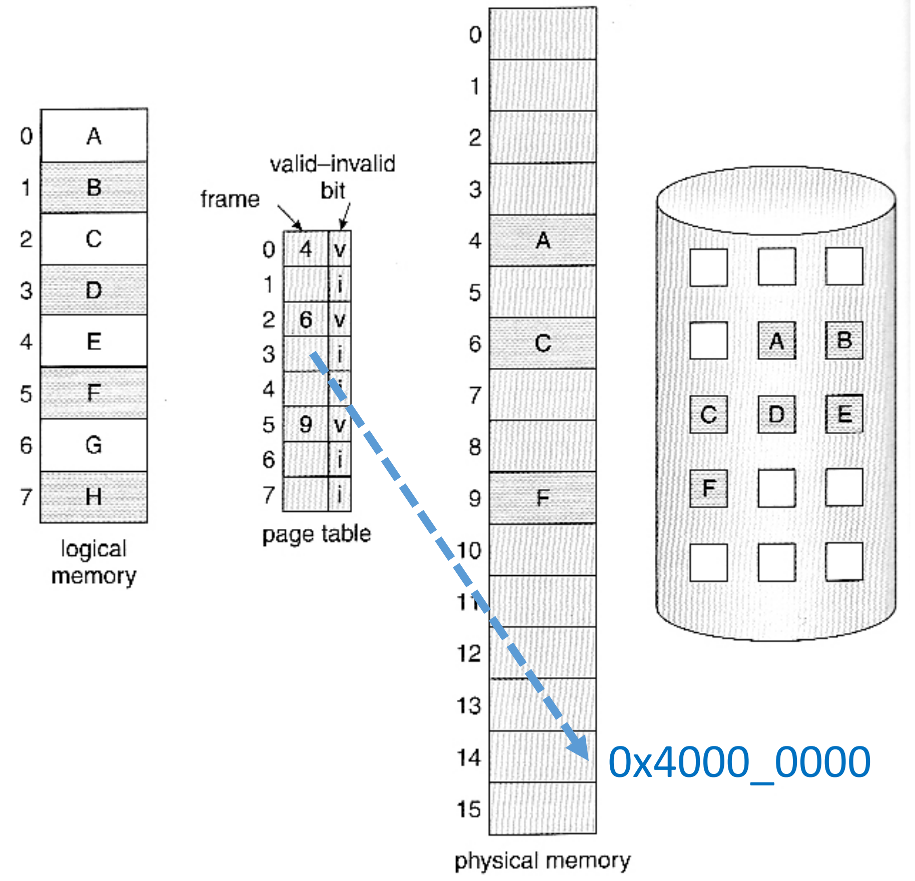


Figure 9.5 Page table when some pages are not in main memory.

Decomposing Load Instruction Execution

- What is the contents of PTE (Page Table Entry)?

```
int foo = open("/dev/mem", O_RDWR);
```

// Given a pathname for a file, open() returns a file descriptor

// 'dev/mem' refers to the system's physical memory

// O_RDWR means both readable and writable access mode

```
int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);
```

// mmap() creates a new mapping in the virtual address space of the calling process

// NULL means that the kernel chooses the address for mapping

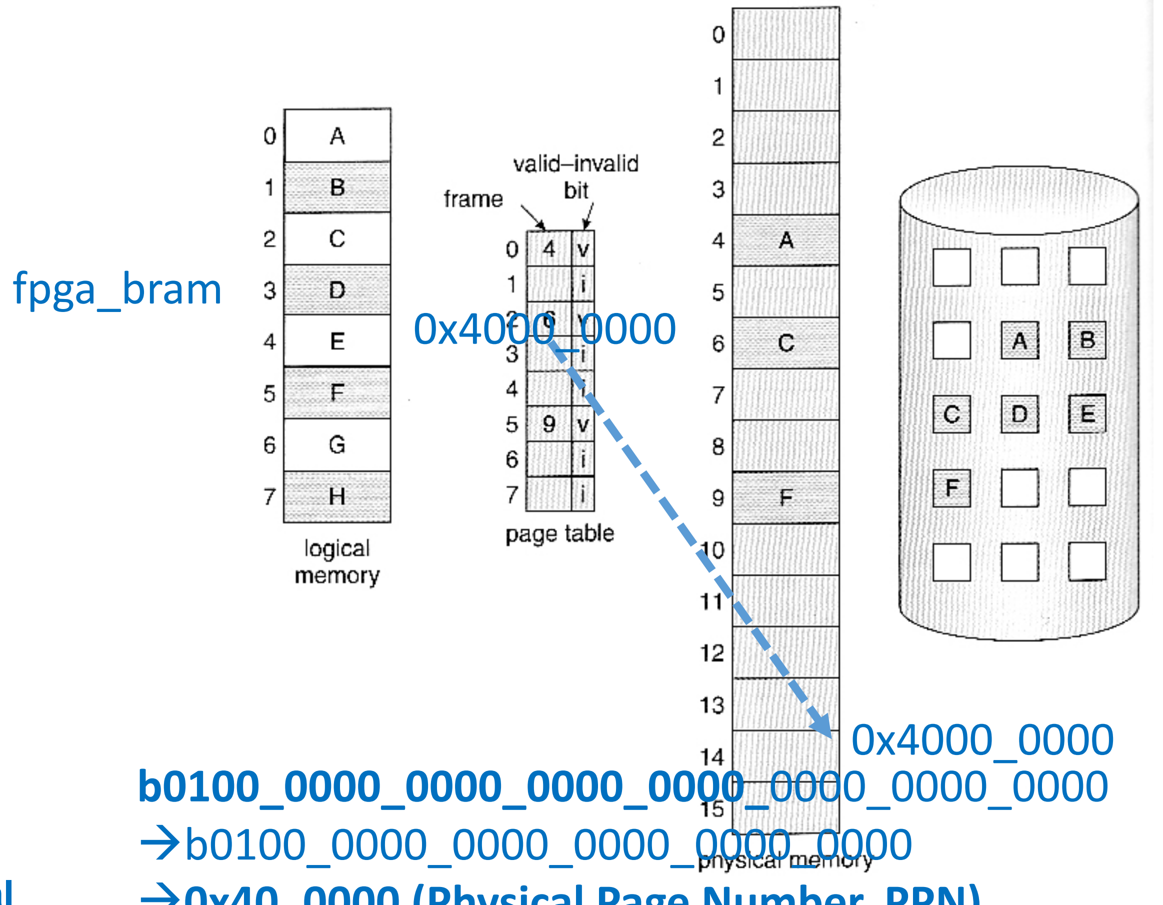
// SIZE specifies the length of the mapping

// PROT_ arguments describe the memory protection (RD/WR)

// MAP_SHARED makes updates visible to other processes

// foo indicates the file descriptor to be mapped

// **0x4000_0000 refers to offset of the file descriptor → physical address for BRAM**



b0100_0000_0000_0000_0000_0000_0000_0000

→b0100_0000_0000_0000_0000_0000

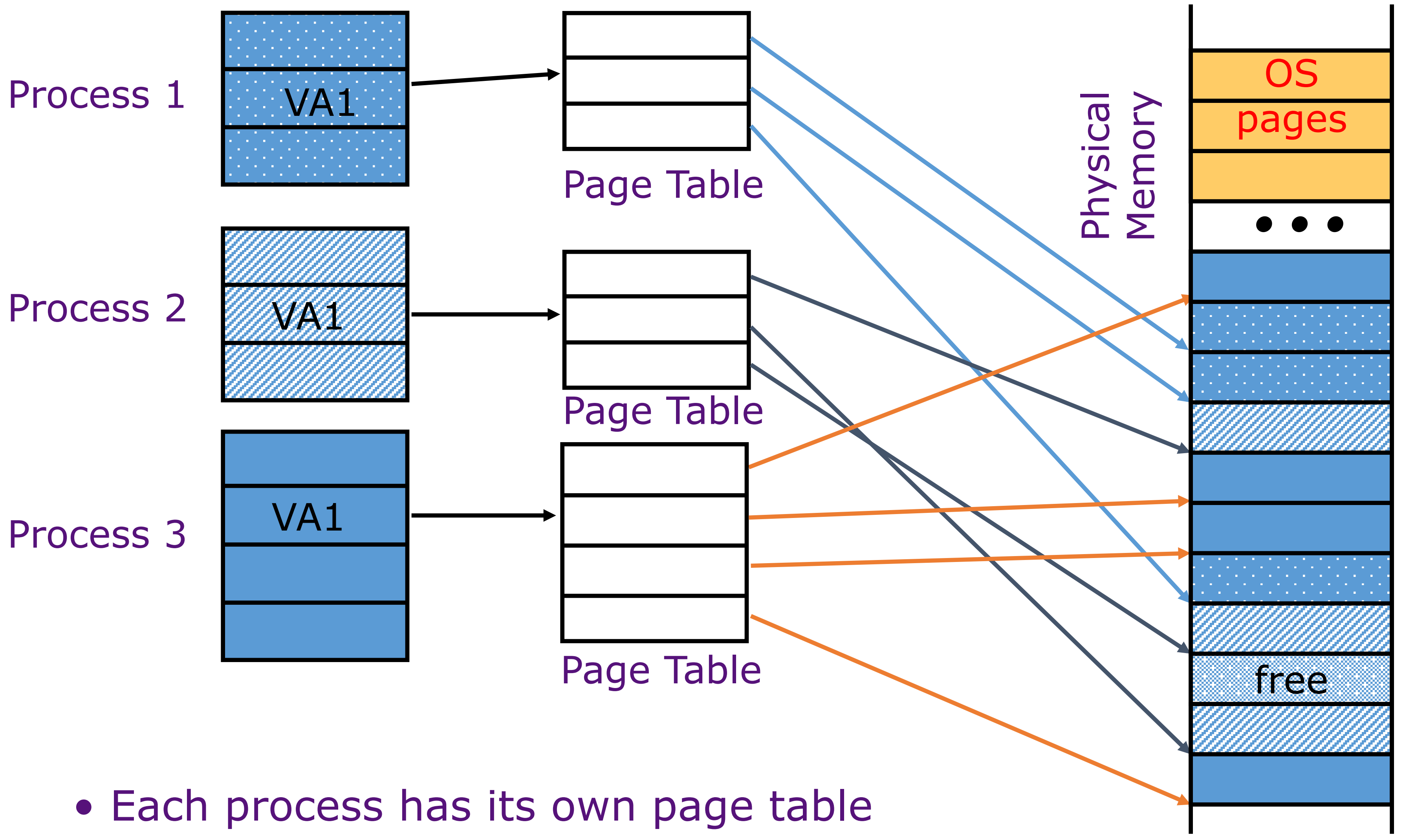
→0x40_0000 (Physical Page Number, PPN)

Figure 9.5 Page table when some pages are not in main memory.

[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

- Private virtual address space per process

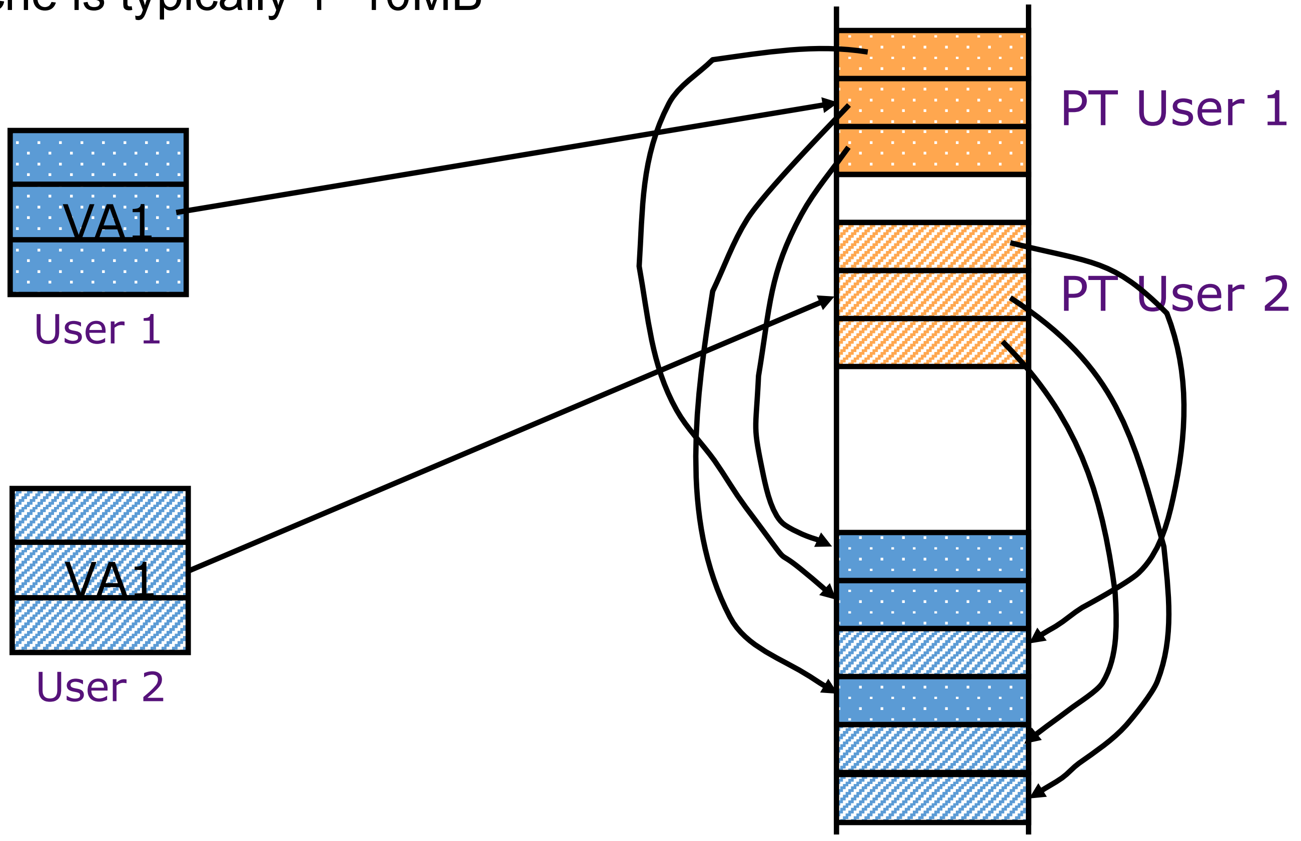


- Each process has its own page table

[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

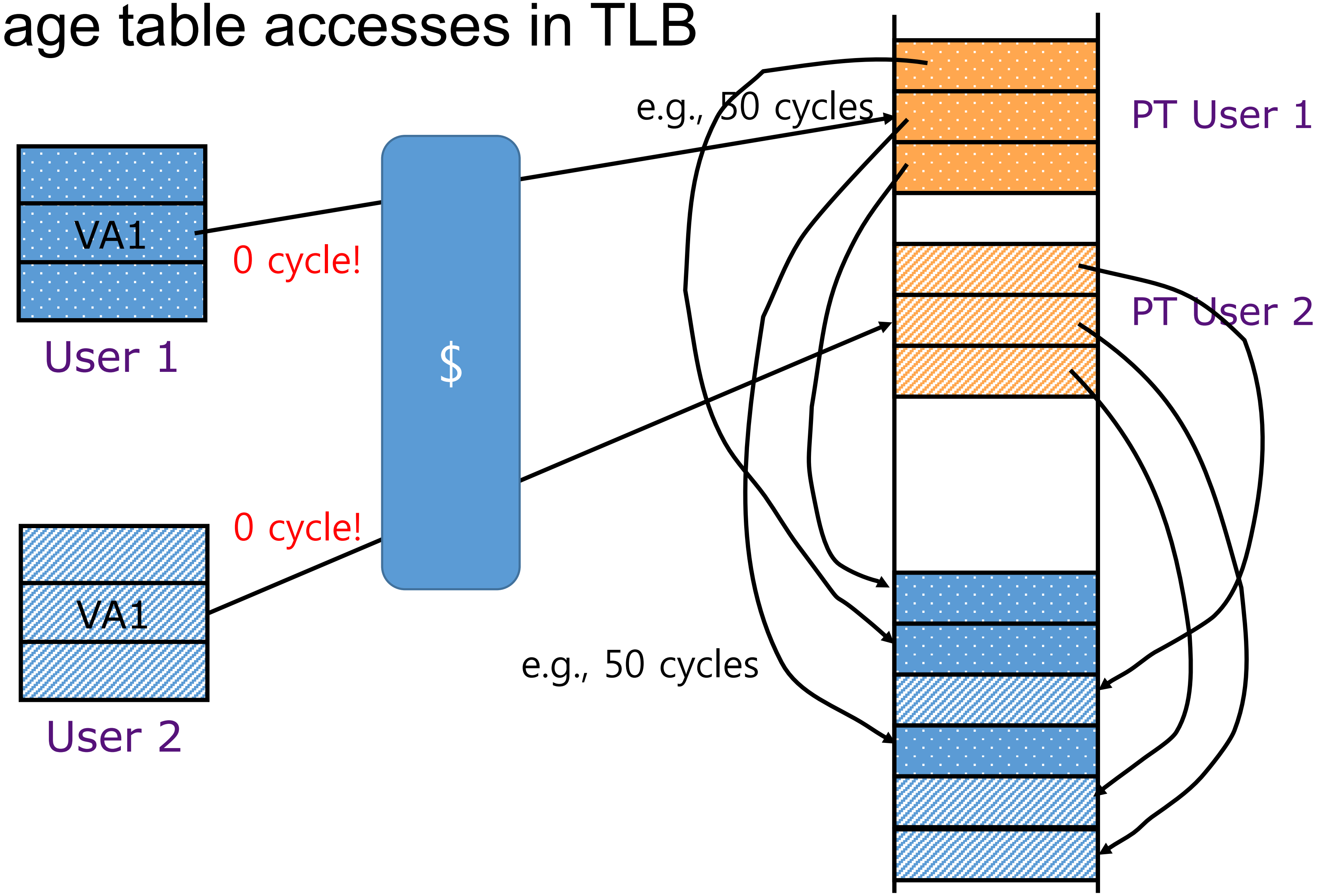
- Page tables in physical memory
  - Page table, requiring 4MB per process, is too big to be stored on CPU chip and thus stored in main memory
  - The size of CPU cache is typically 1~10MB



[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

- Caching page table accesses in TLB



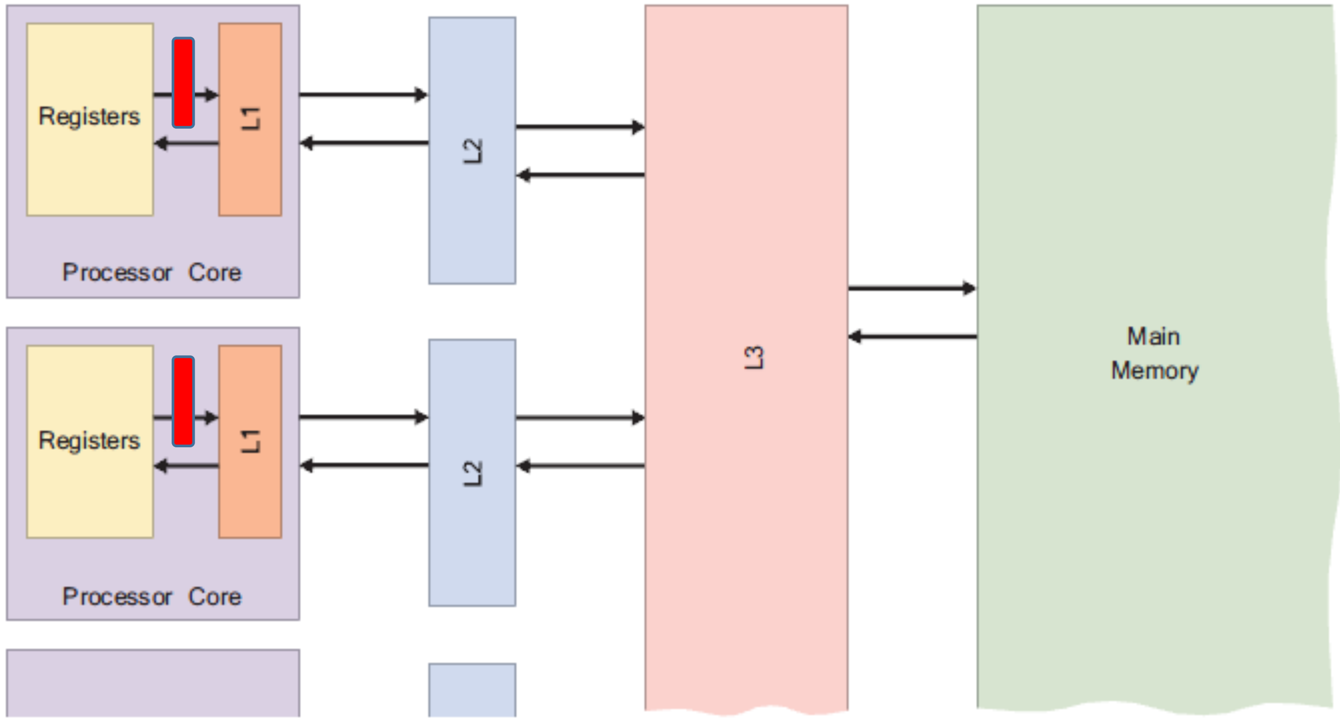
[The Zynq Book, 2014]

Decomposing Load Instruction Execution

Memory hierarchy

- Main memory
  - Dynamic RAM (DRAM)
- Cache
  - Static RAM (SRAM)
  - L1 cache
    - 1~2 clock cycles, ~32KB
    - Instruction (I) cache, data (D) cache
  - L2 cache
    - ~10 clock cycles, 100KB~1MB
    - Shared I+D
  - L3 cache
    - ~50 clock cycles, 1MB~10MB
    - eDRAM for better area efficiency in IBM PowerPC

TLB (Translation Lookaside Buffer) for virtual address (VA) to physical address (PA) translation



[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

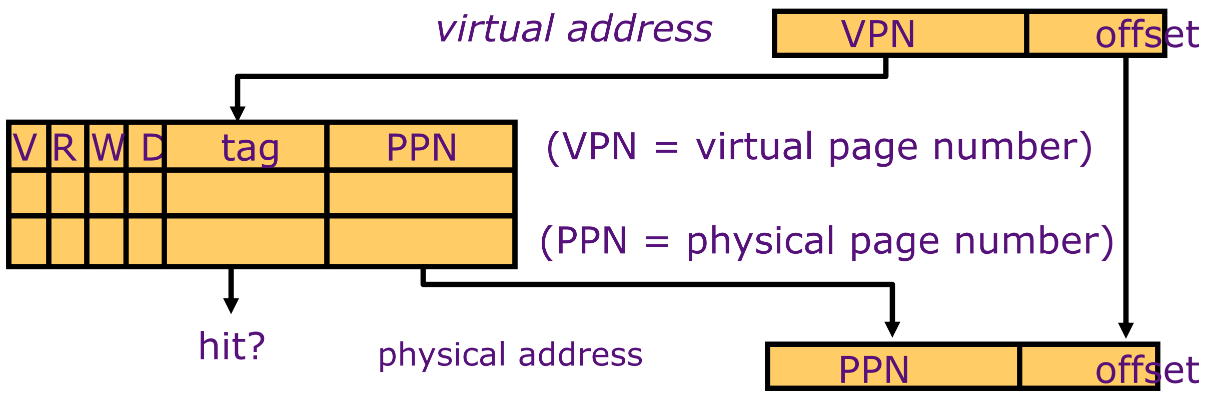
- Translation Lookaside Buffers (TLB)

Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

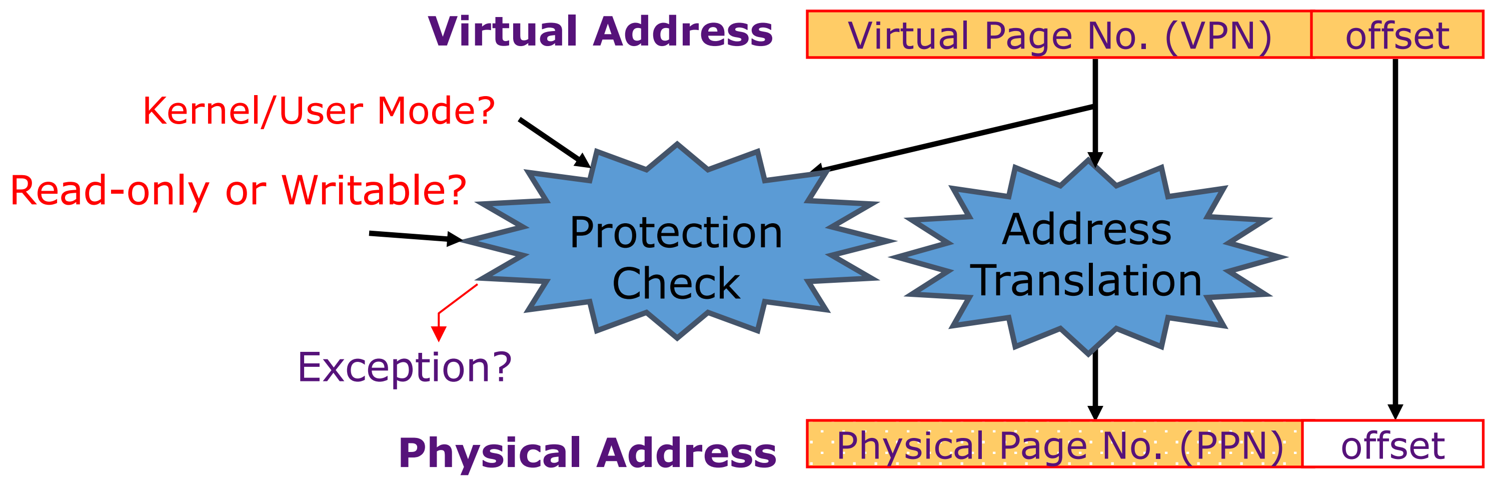
TLB hit	$\Rightarrow$ *Single Cycle Translation*
TLB miss	$\Rightarrow$ *Page Table Walk to refill*



[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

- Address translation & protection

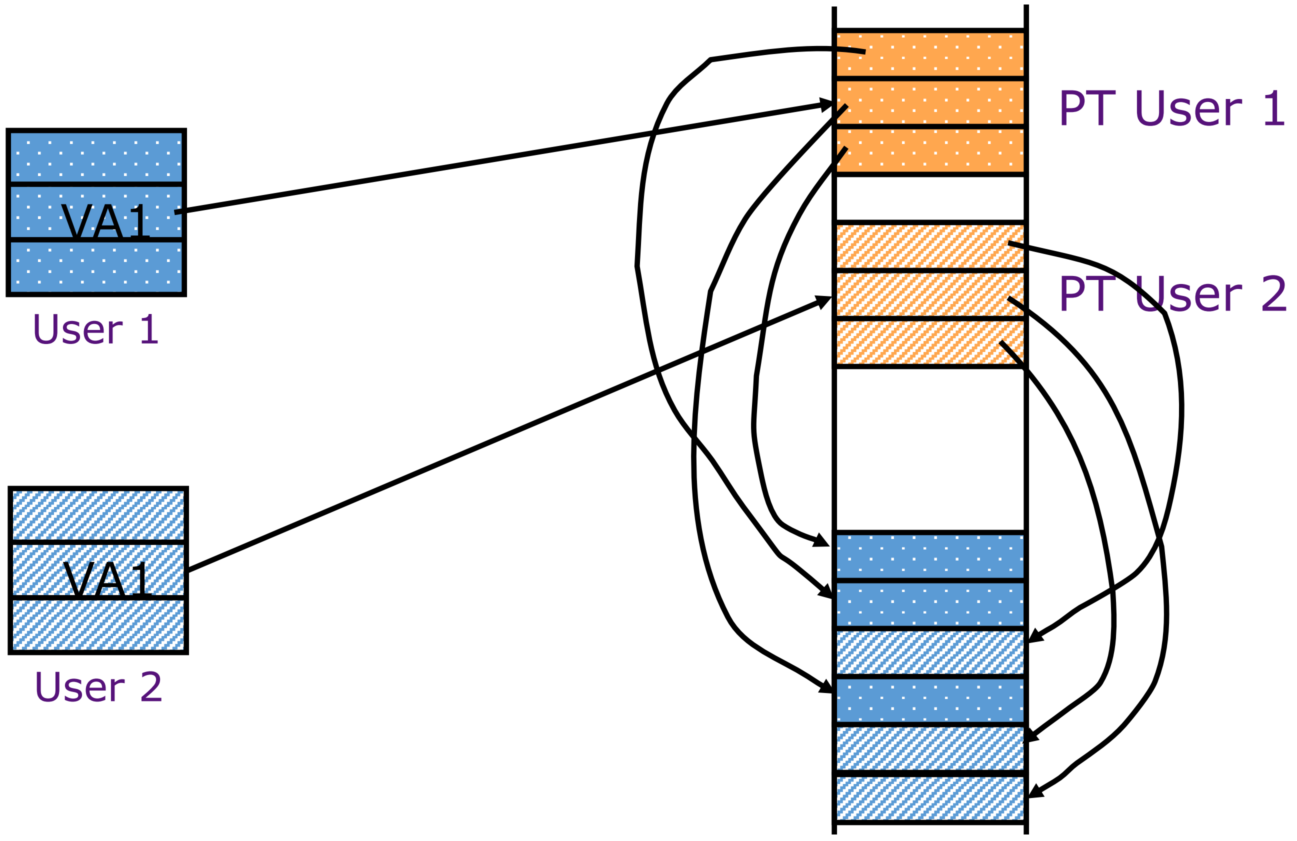


• Every instruction and data access (to cache) needs address translation and protection checks

[Source: K. Asanovic, 2008]

Decomposing Load Instruction Execution

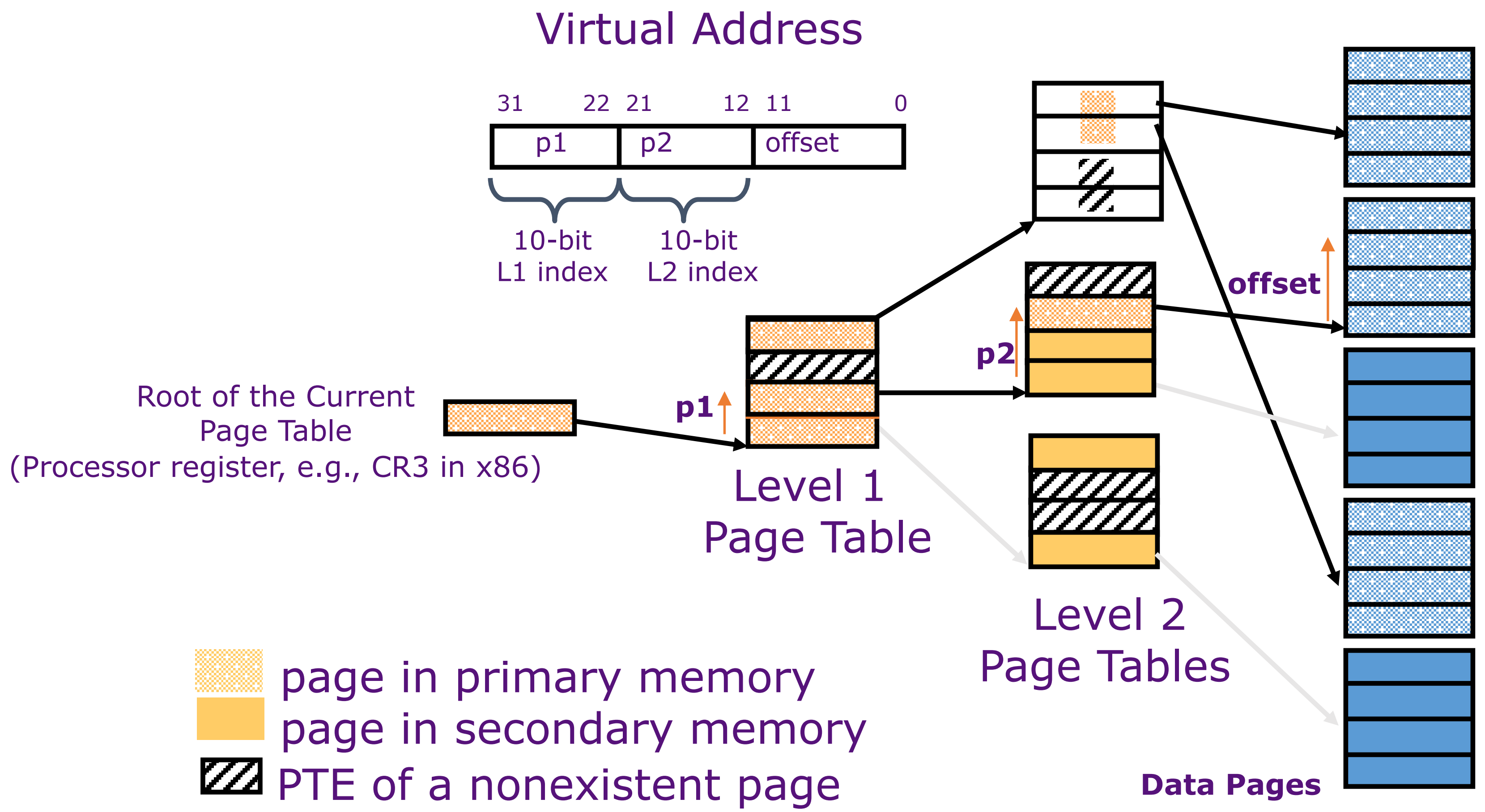
- Flat page tables are expensive
  - If you run 100 processes each requiring its own page table of 4MB, the total page table size, 400MB is too big



[Source: K. Asanovic, 2008]

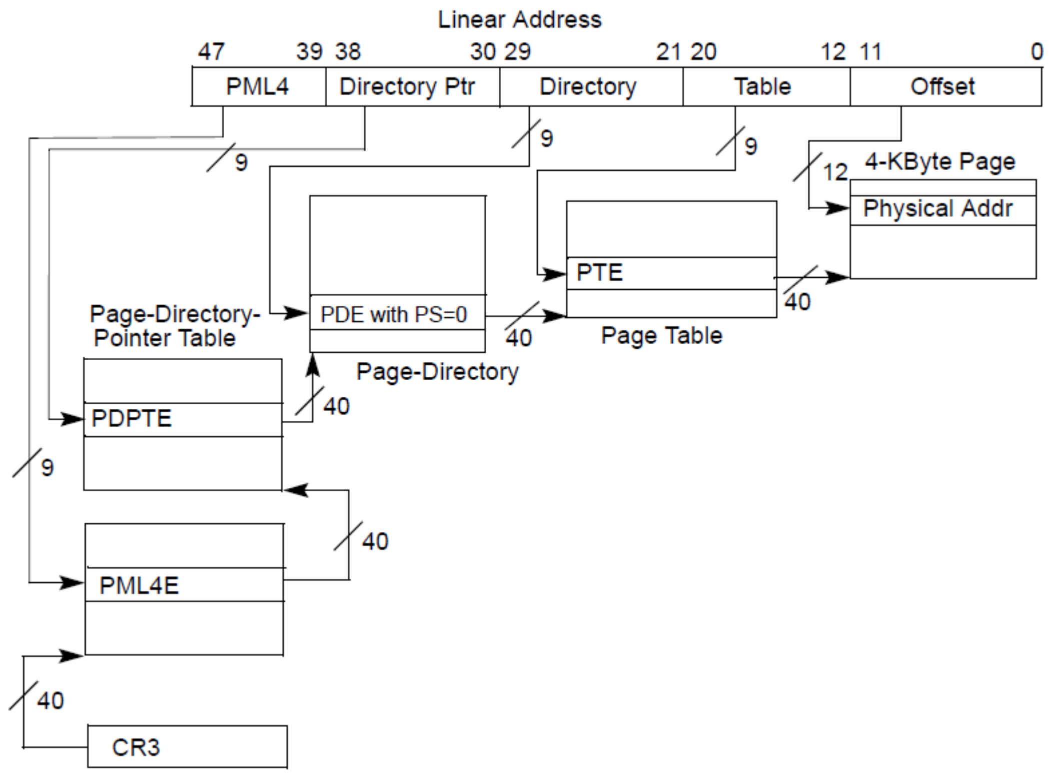
Decomposing Load Instruction Execution

- Hierarchical page table to reduce page table size



Decomposing Load Instruction Execution

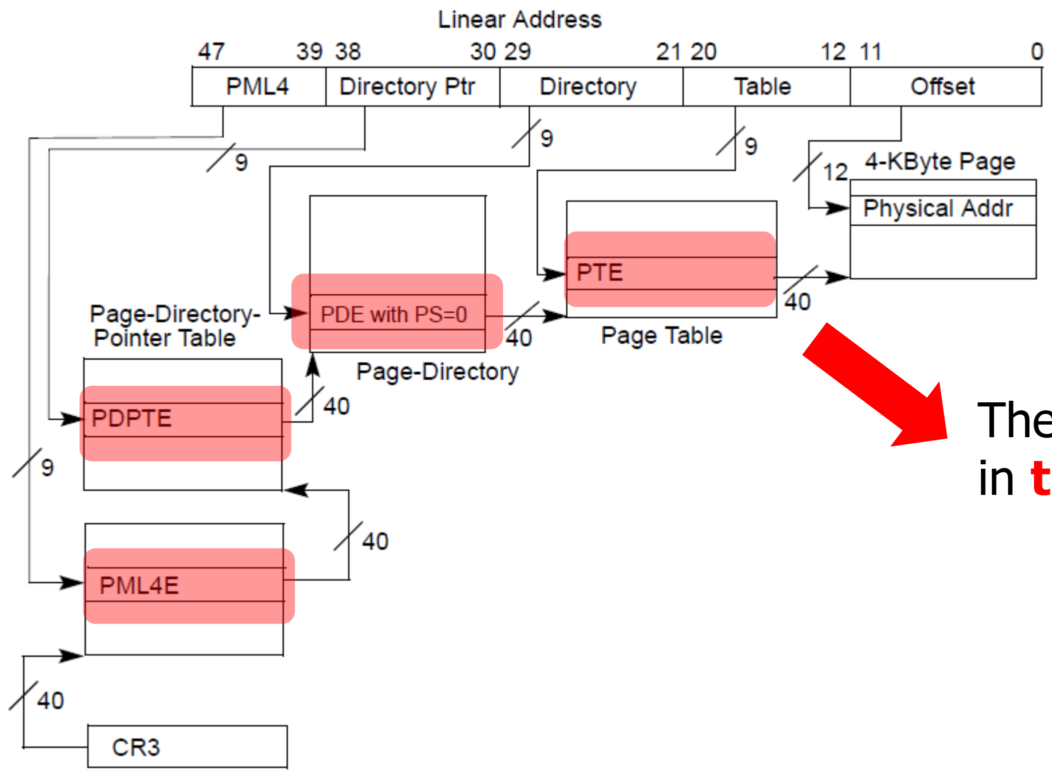
- Page table with 48b address space
  - All the tables are stored in main memory
  - How many memory accesses for address translation?



[Intel® 64 and IA-32 Architectures Software Developer's Manual]

Decomposing Load Instruction Execution

- TLB miss penalty in hierarchical page table
  - Since all the tables are stored in main memory, in the worst case, 4 memory accesses for address translation occur, ~50ns x 4=200ns!

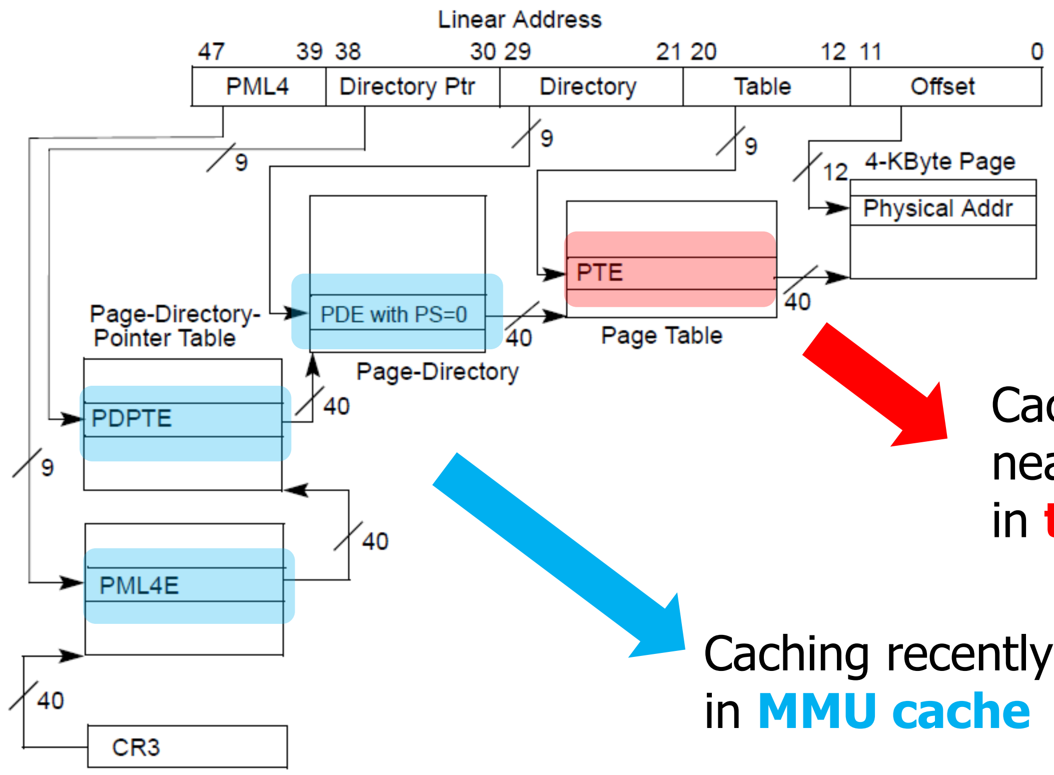


Then, the PTE is stored in **translation lookaside buffer (TLB)**

[Intel® 64 and IA-32 Architectures Software Developer's Manual]

Decomposing Load Instruction Execution

- Paging with 48b address TLB and MMU cache
  - TLB and MMU cache store recently accessed entries to reduce the average latency of address translation



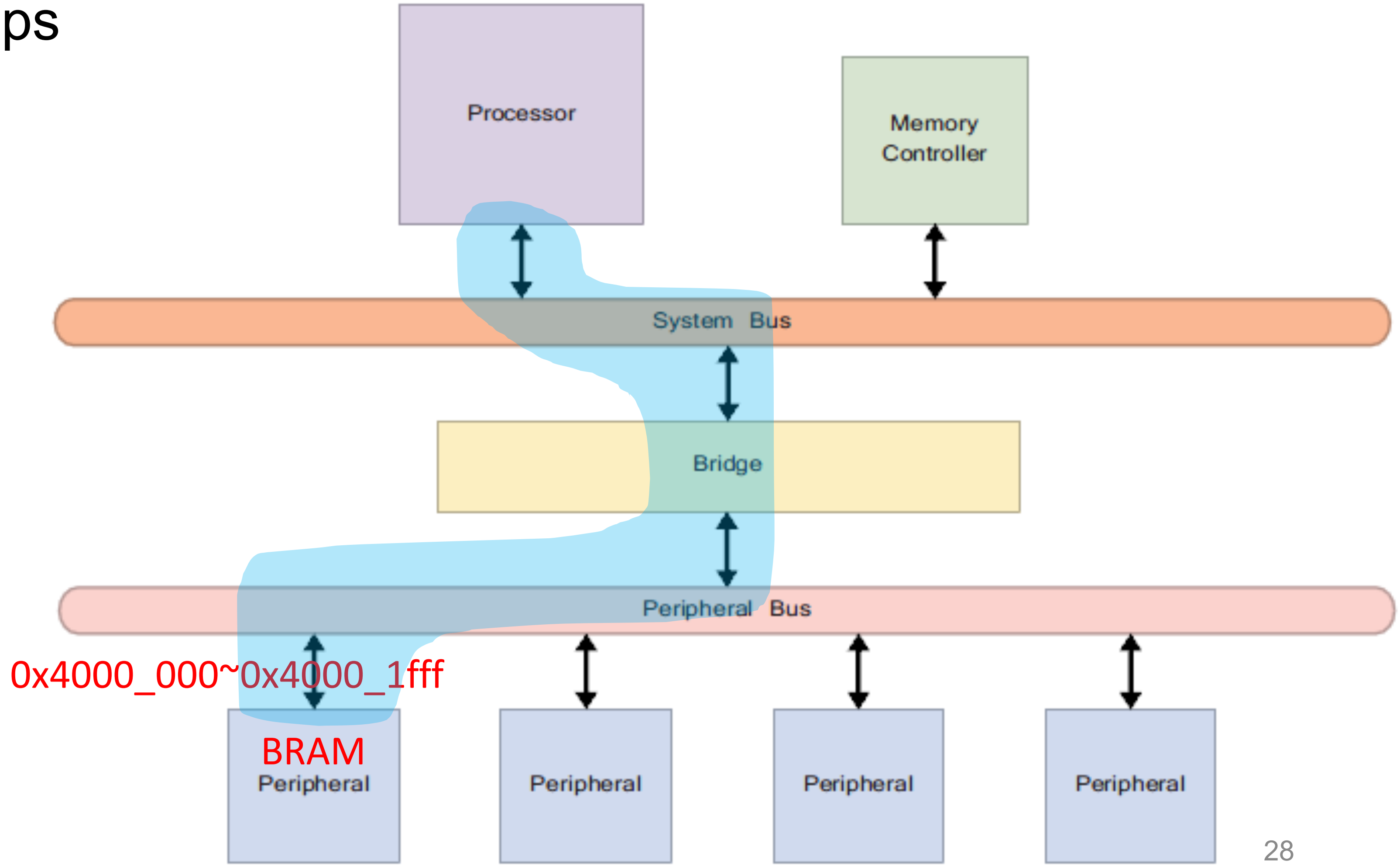
Caching recently accessed PTEs near the CPU in **translation lookaside buffer (TLB)**

Caching recently accessed entries of page directories in **MMU cache**

[The Zynq Book, 2014]

Decomposing Load Instruction Execution

- A tip of iceberg?
  - Triggers a series of steps
- CPU to L1 cache
  - VA to PA by TLB
- What happens next?



Decomposing Load Instruction Execution

<u>We have covered this step:</u>

- Step 1: Load unit accesses L1 data cache
  - TLB access
  - Non-cacheable access

<u>We will cover in the next lecture:</u>

- Step 2: CPU sends a read request to the bus
  - What does "read request" exactly mean?
- Step 3: Bus determines the destination and forwards the read request to the destination
  - How can the bus determine the destination?
  - What does 'forward' exactly mean?
- Step 4: The hardware component receives the read request and sends the required data to the bus
  - What does "receive the read request" exactly mean?
- Step 5: Bus forwards the data to CPU
- Step 6: CPU stores the data in its registers

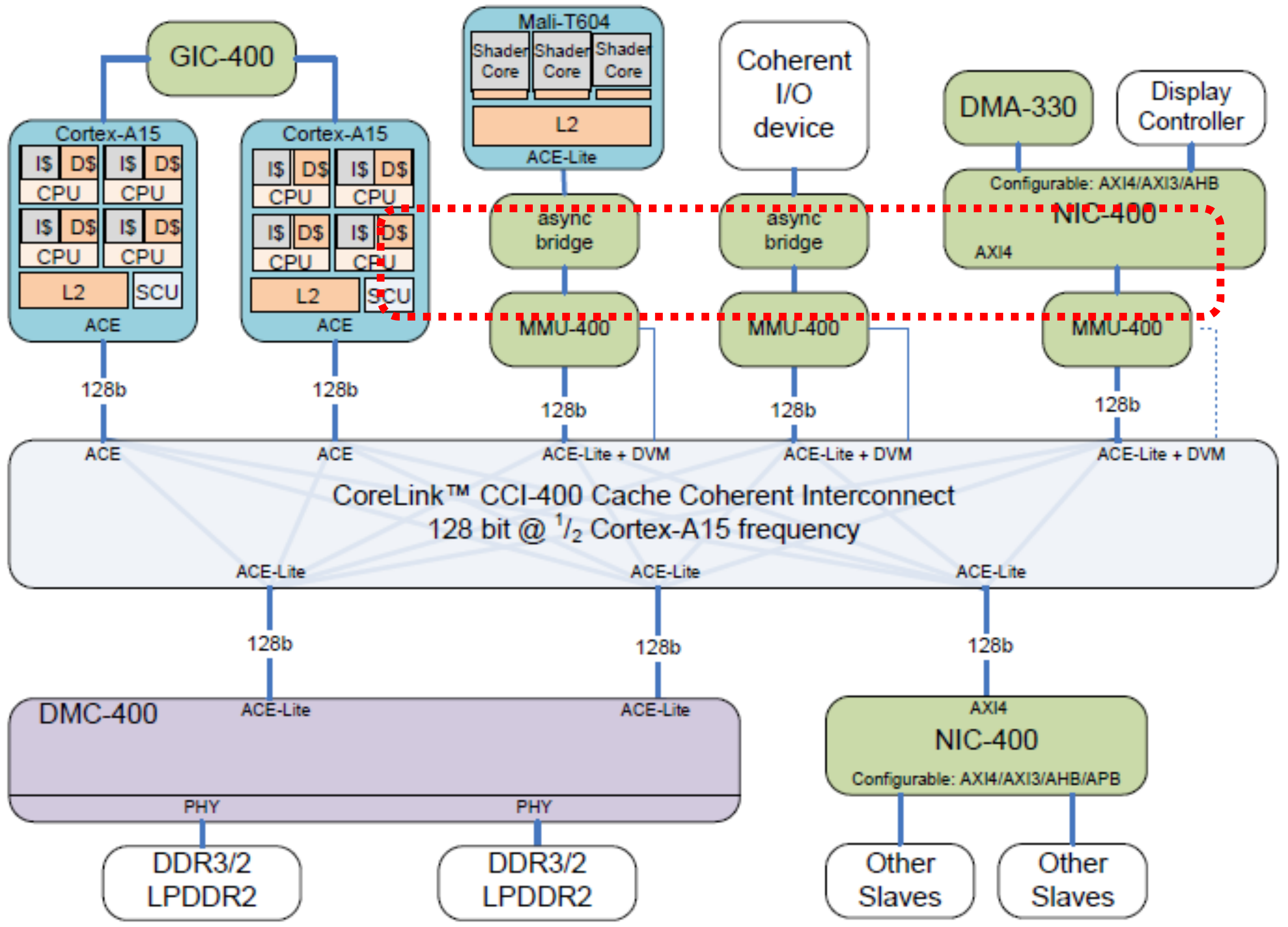
Outline

- How can software access a hardware component?
- How can hardware components access the main memory?
  - IOMMU
    - Hardware components use virtual address as well
    - VA to PA mapping is done by IOMMU
- Overview of Lab 6 and 7

[Stevens, 2011]

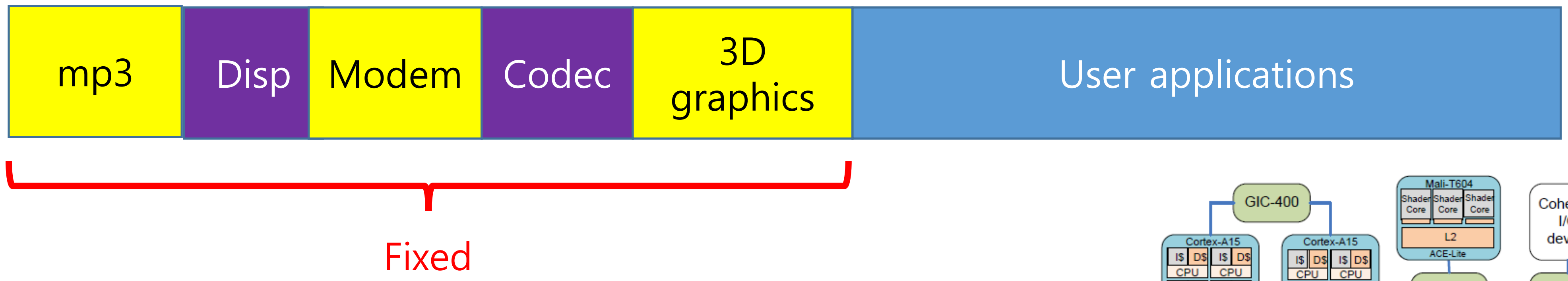
IOMMU: Why We Need It?

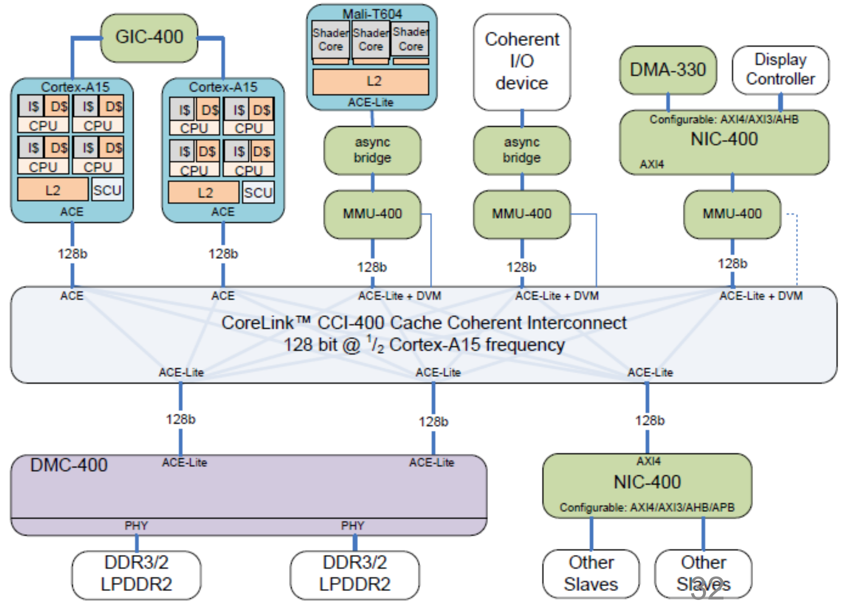
- Hardware devices require virtual address



IOMMU: Why We Need It?

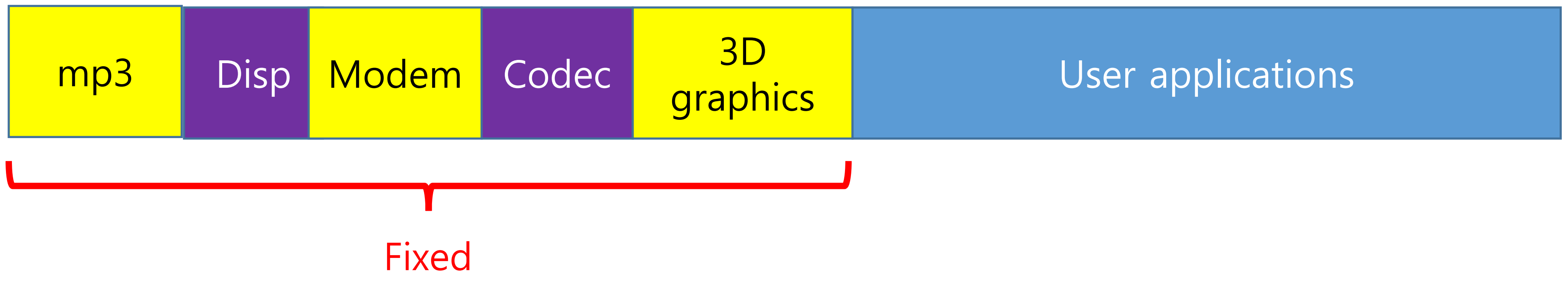
- What happens if hardware components use physical address?
  - Each hardware component needs a fixed region of physical address



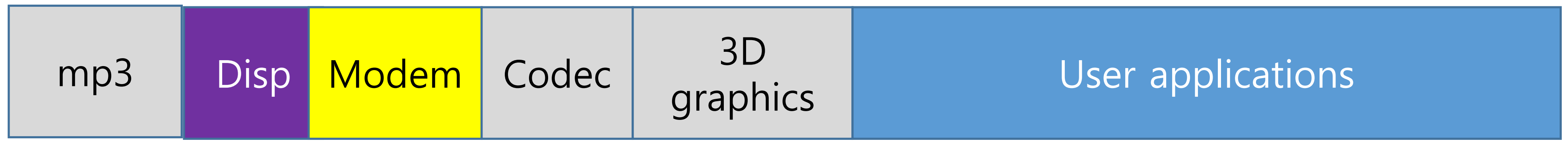


IOMMU: Why We Need It?

- What happens if hardware components use physical address?
  - Each hardware component needs a fixed region of physical address

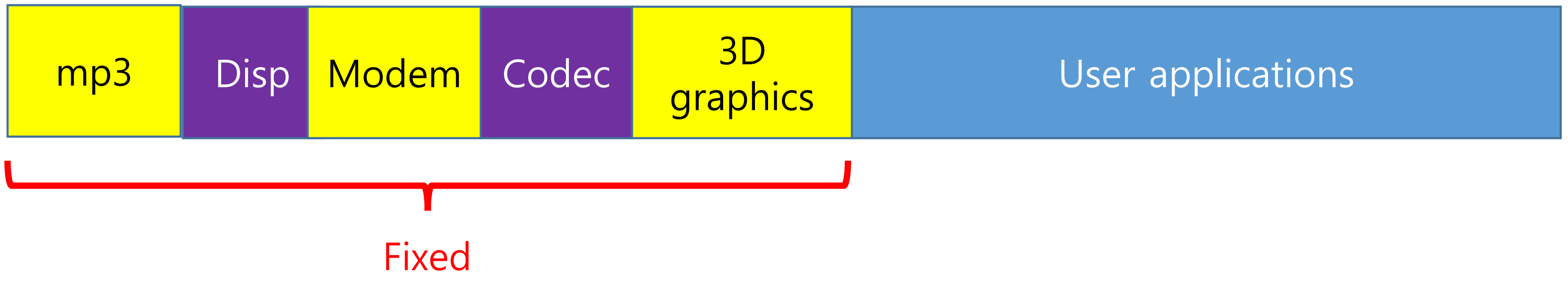


- What if you use only text messaging for now?
  - **Waste of main memory resource** due to fixed physical address allocation



IOMMU: Why We Need It?

- What happens if hardware components use physical address?
  - Each hardware component needs a fixed region of physical address

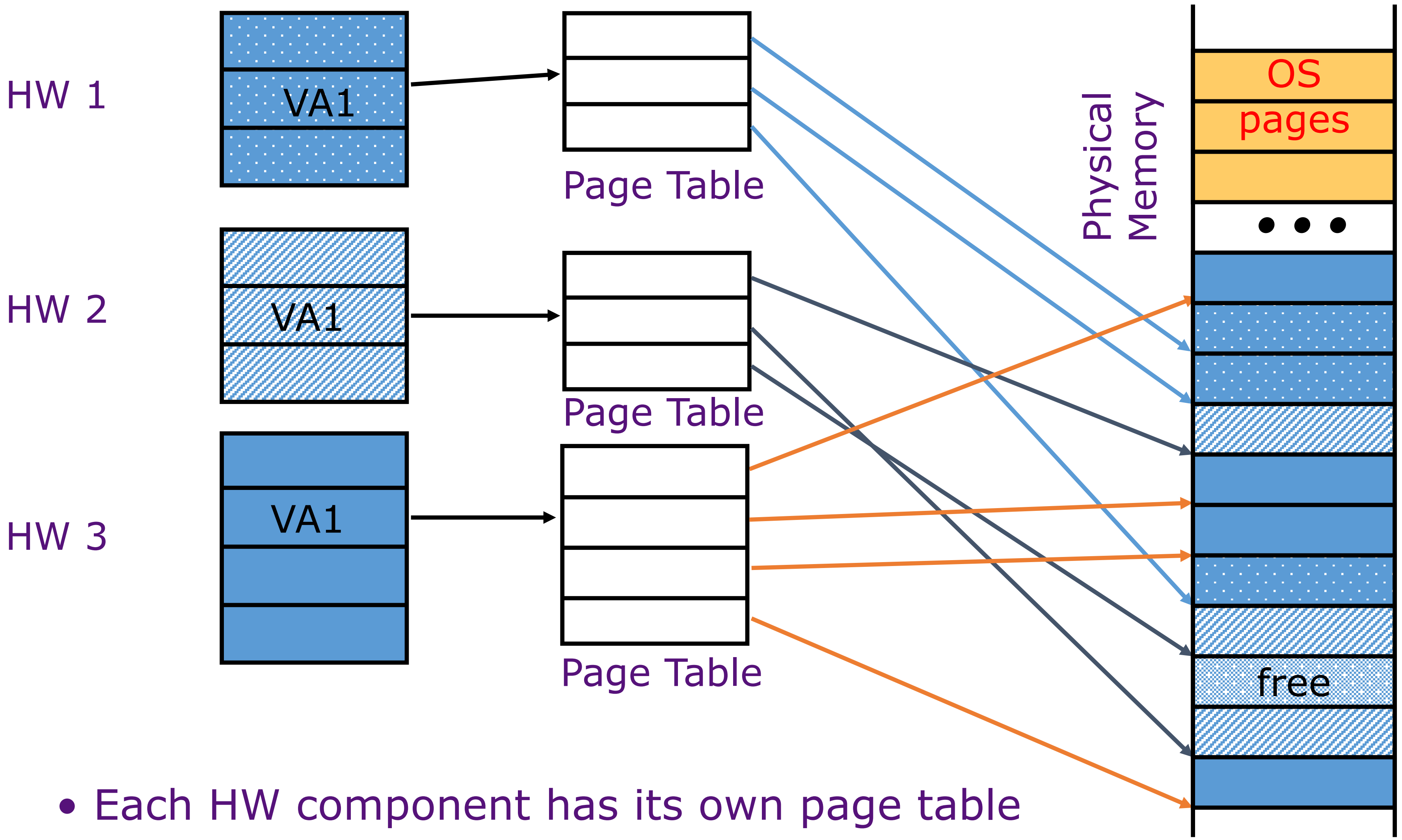


- What if you use only text messaging for now?
  - **Waste of main memory resource** due to fixed physical address allocation
- Solution? IOMMU!
  - Hardware components use virtual address
  - IOMMU does VA to PA translation for hardware components

[Source: K. Asanovic, 2008]

IOMMU for Better Utilization of Memory

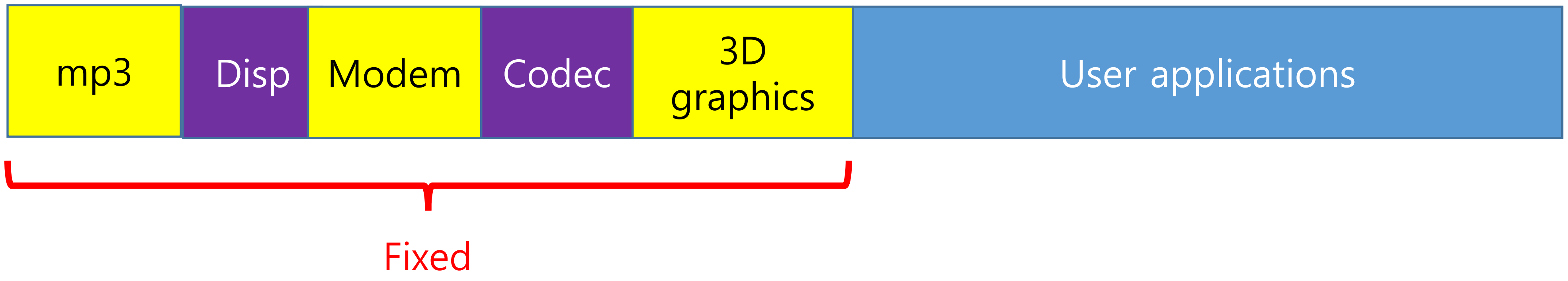
- Private virtual address space per HW component



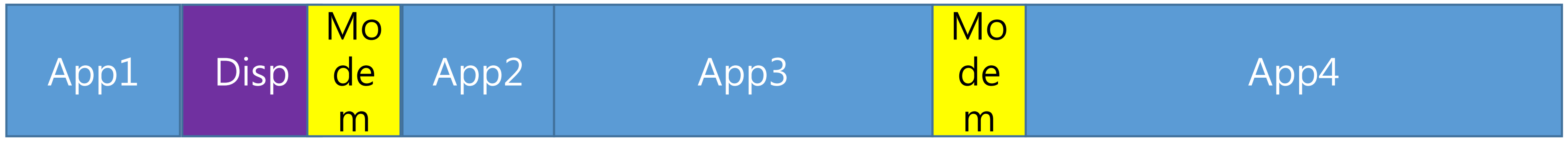
- Each HW component has its own page table

IOMMU for Better Utilization of Memory

- What happens if hardware components use physical address?
  - Each hardware component needs a fixed region of physical address

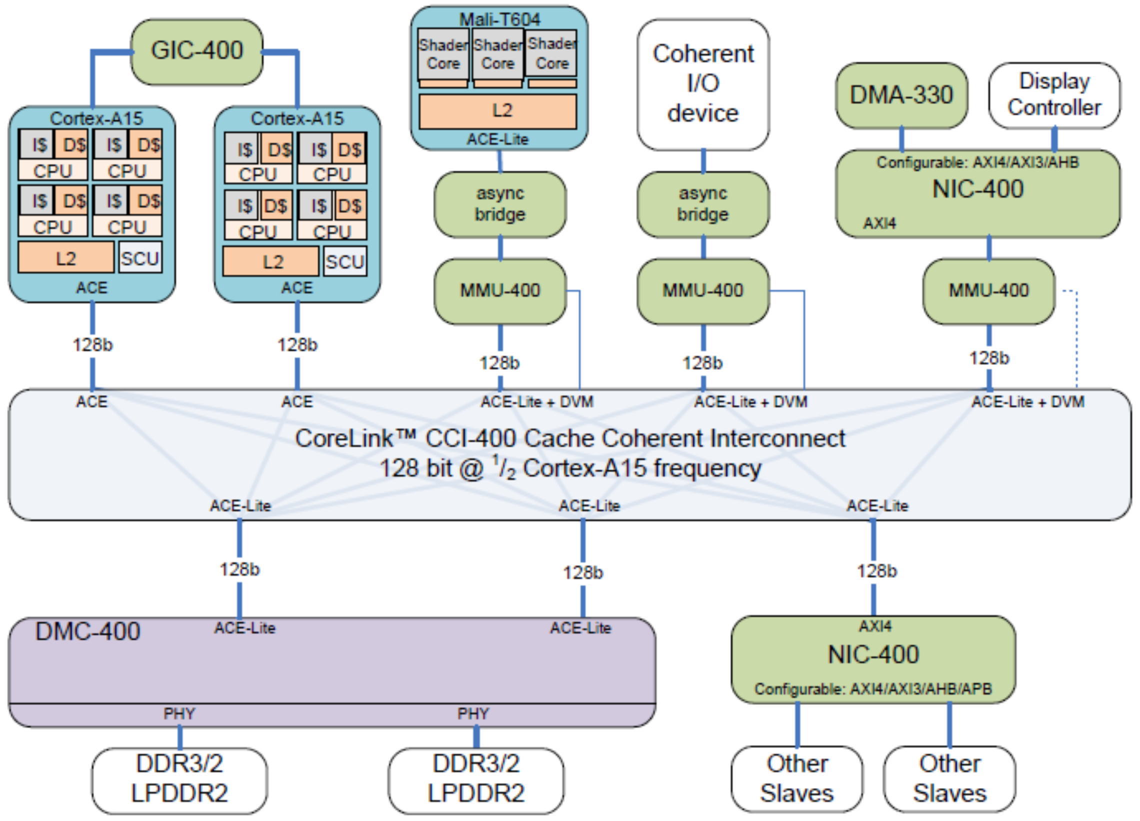


- What if you use only text messaging for now?
  - **Waste of main memory resource** due to fixed physical address allocation
- IOMMU enables better utilization of memory resource



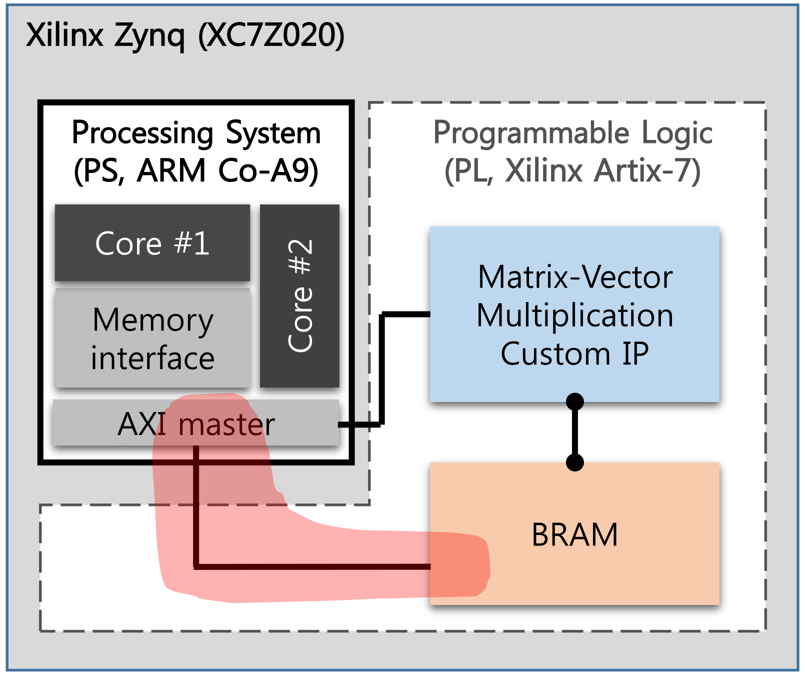
IOMMU for Better Utilization of Memory

- IOMMU (Input Output Memory Management Unit)
  - Each device needs its own TLB for VA to PA translation before accessing main memory

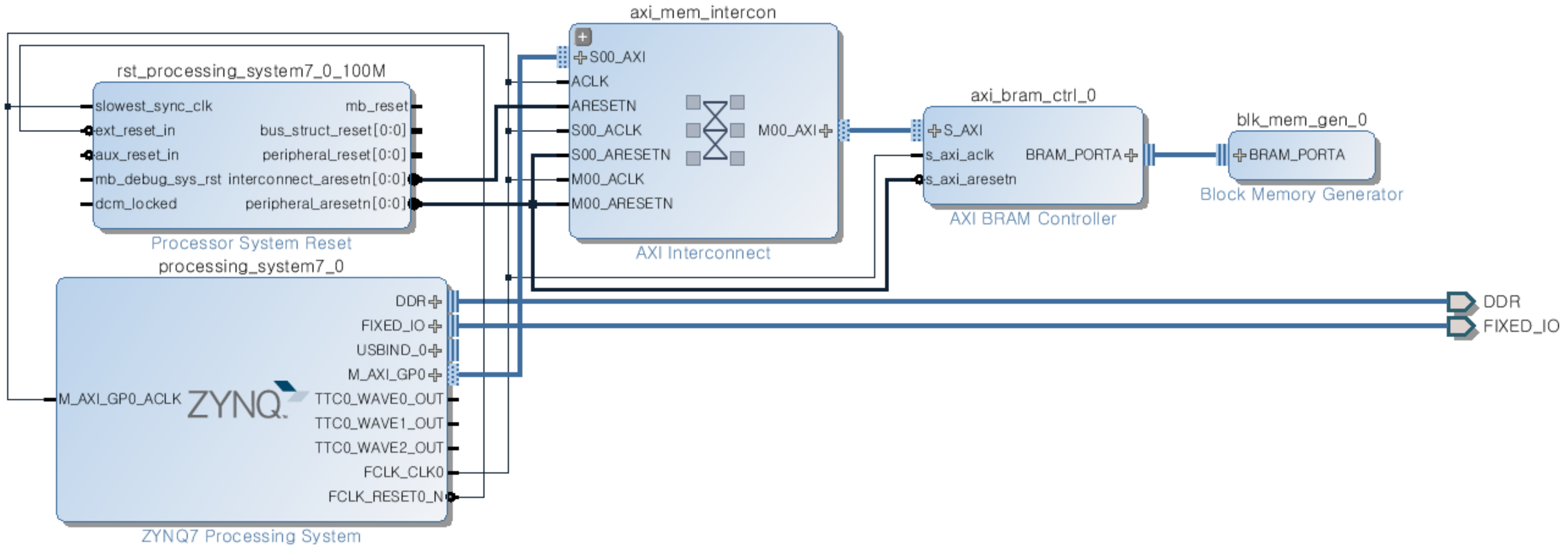


Outline

- How can software access a hardware component?
- How can hardware components access the main memory?
- Overview of Lab 6 and 7


Xilinx Zynq (XC7Z020)
Processing System (PS, ARM Co-A9)
Core #1
Memory interface
Core #2
AXI master
Programmable Logic (PL, Xilinx Artix-7)
Matrix-Vector Multiplication Custom IP
BRAM

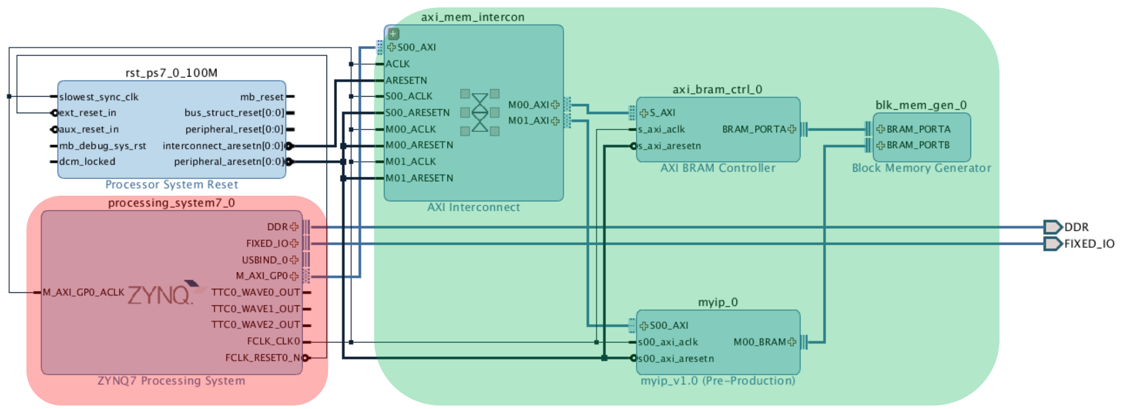
Lab 6: BRAM

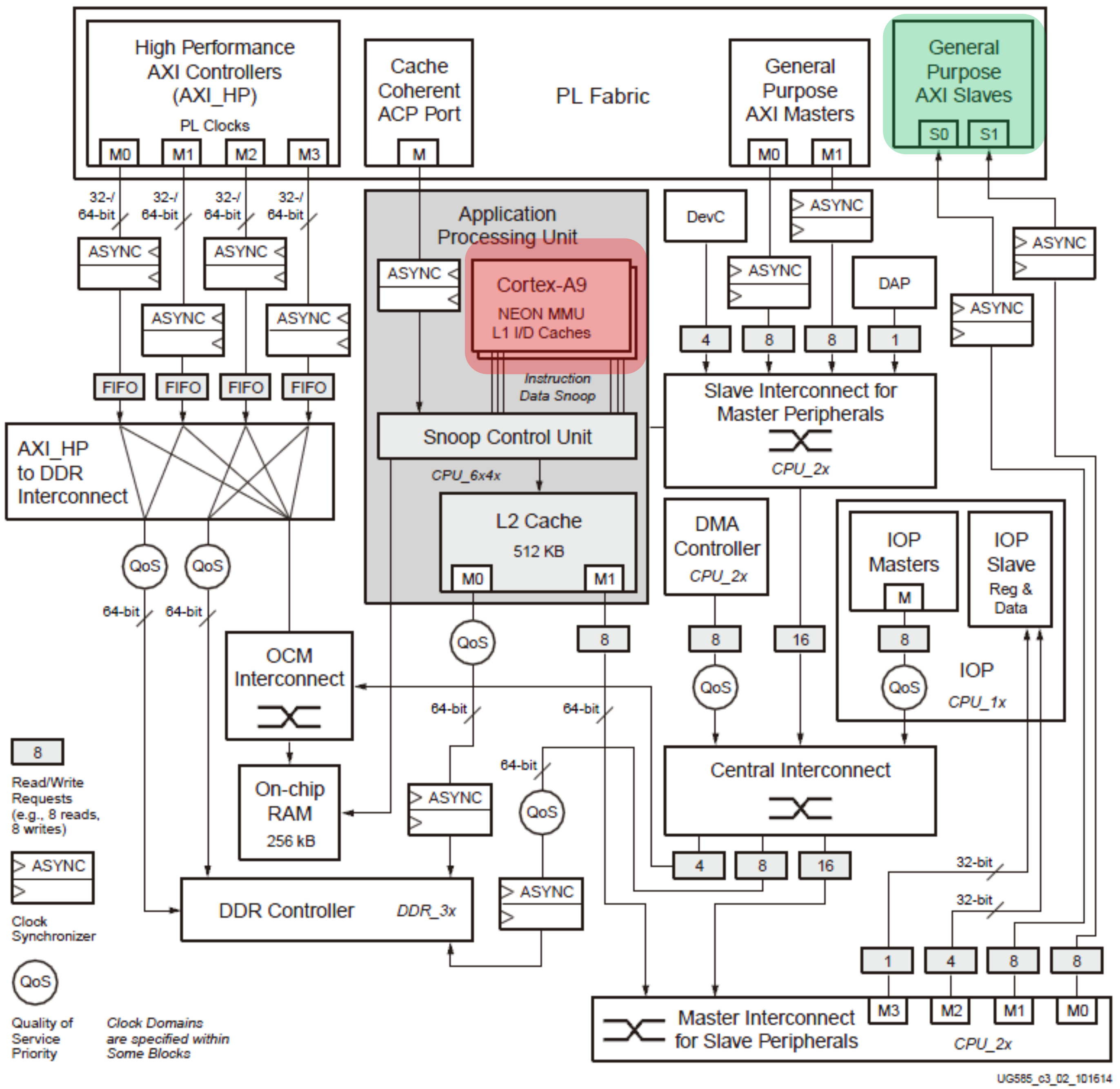

axi_mem_intercon
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M00_AXI
AXI Interconnect
rst_processing_system7_0_100M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
axi_bram_ctrl_0
S_AXI
s_axi_aclk
s_axi_aresetn
BRAM_PORTA
AXI BRAM Controller
blk_mem_gen_0
BRAM_PORTA
Block Memory Generator
processing_system7_0
M_AXI_GP0_ACLK
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
TTC0_WAVE0_OUT
TTC0_WAVE1_OUT
TTC0_WAVE2_OUT
FCLK_CLK0
FCLK_RESET0_N
ZYNQ
ZYNQ7 Processing System
DDR
FIXED_IO

Lab 6: BRAM

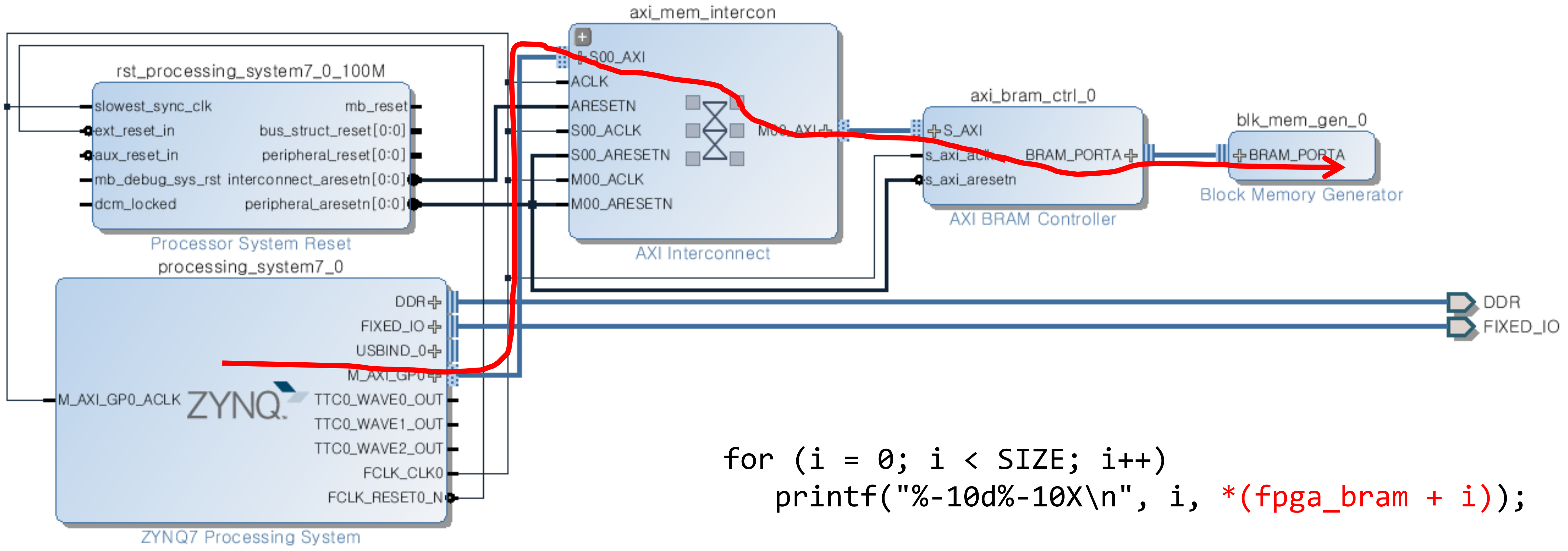
[UG585 Zynq TRM, 2016]

- Interconnect switches
  - Central Interconnect
  - Master Interconnect
  - Slave Interconnect
  - Memory Interconnect
  - OCM Interconnect





Lab 6: BRAM

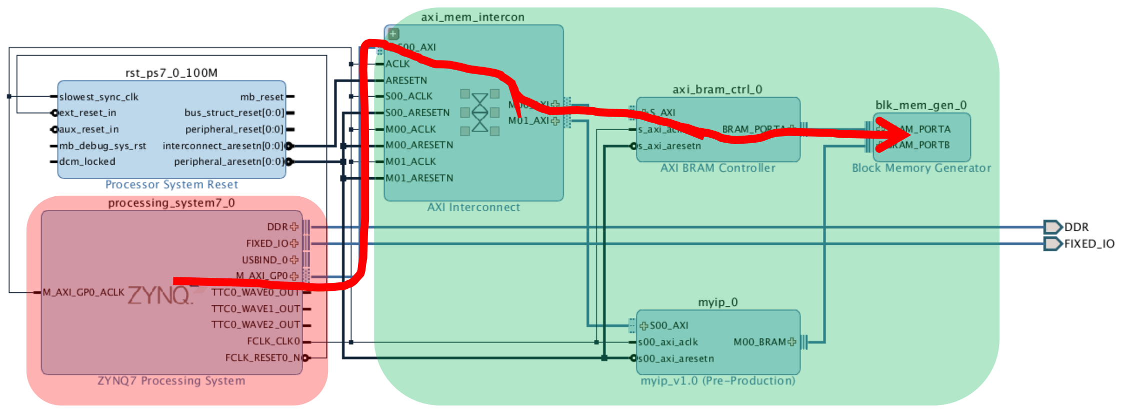


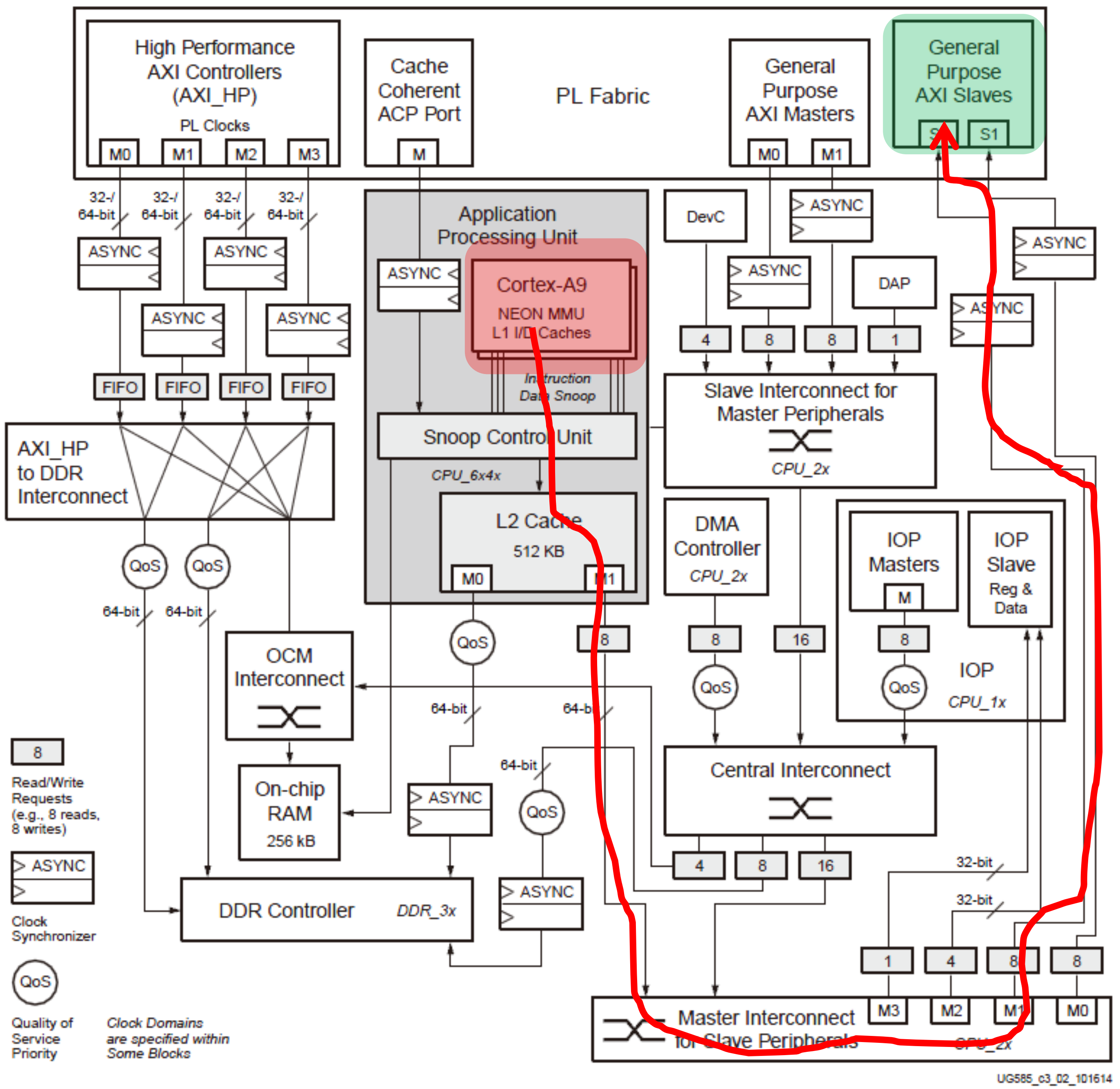
```
for (i = 0; i < SIZE; i++)
    printf("%-10d%-10X\n", i, *(fpga_bram + i));
```

[UG585 Zynq TRM, 2016]

Lab 6: BRAM

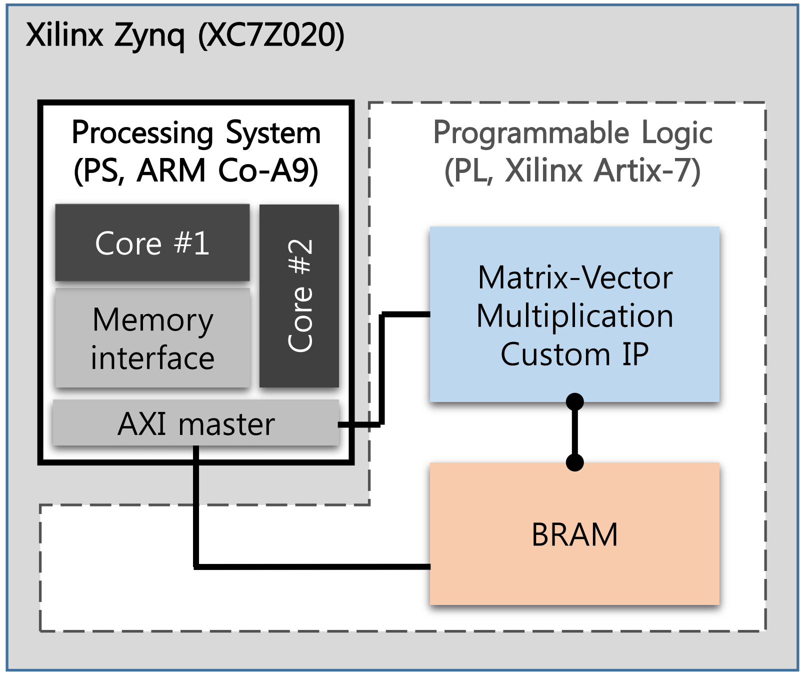
- Interconnect switches
  - Central Interconnect
  - Master Interconnect
  - Slave Interconnect
  - Memory Interconnect
  - OCM Interconnect



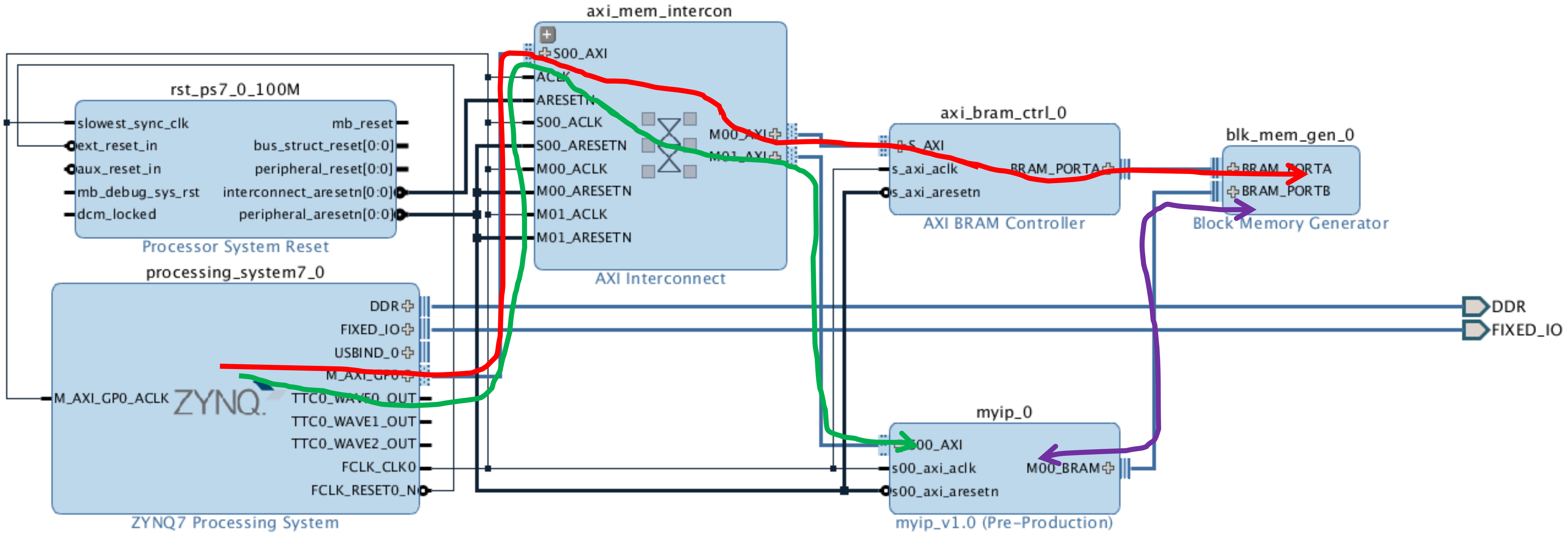


Next Week

- Communication between hardware components
  - CPU – bus (interconnect) – BRAM controller
  - BRAM controller – my hardware component


Xilinx Zynq (XC7Z020)
Processing System (PS, ARM Co-A9)
Core #1
Core #2
Memory interface
AXI master
Programmable Logic (PL, Xilinx Artix-7)
Matrix-Vector Multiplication Custom IP
BRAM

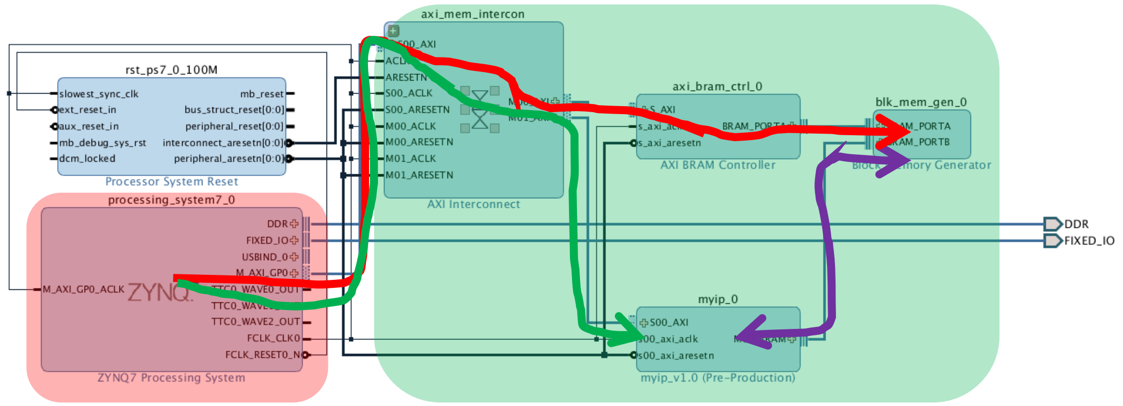
Lab 7: BRAM + My Hardware Component

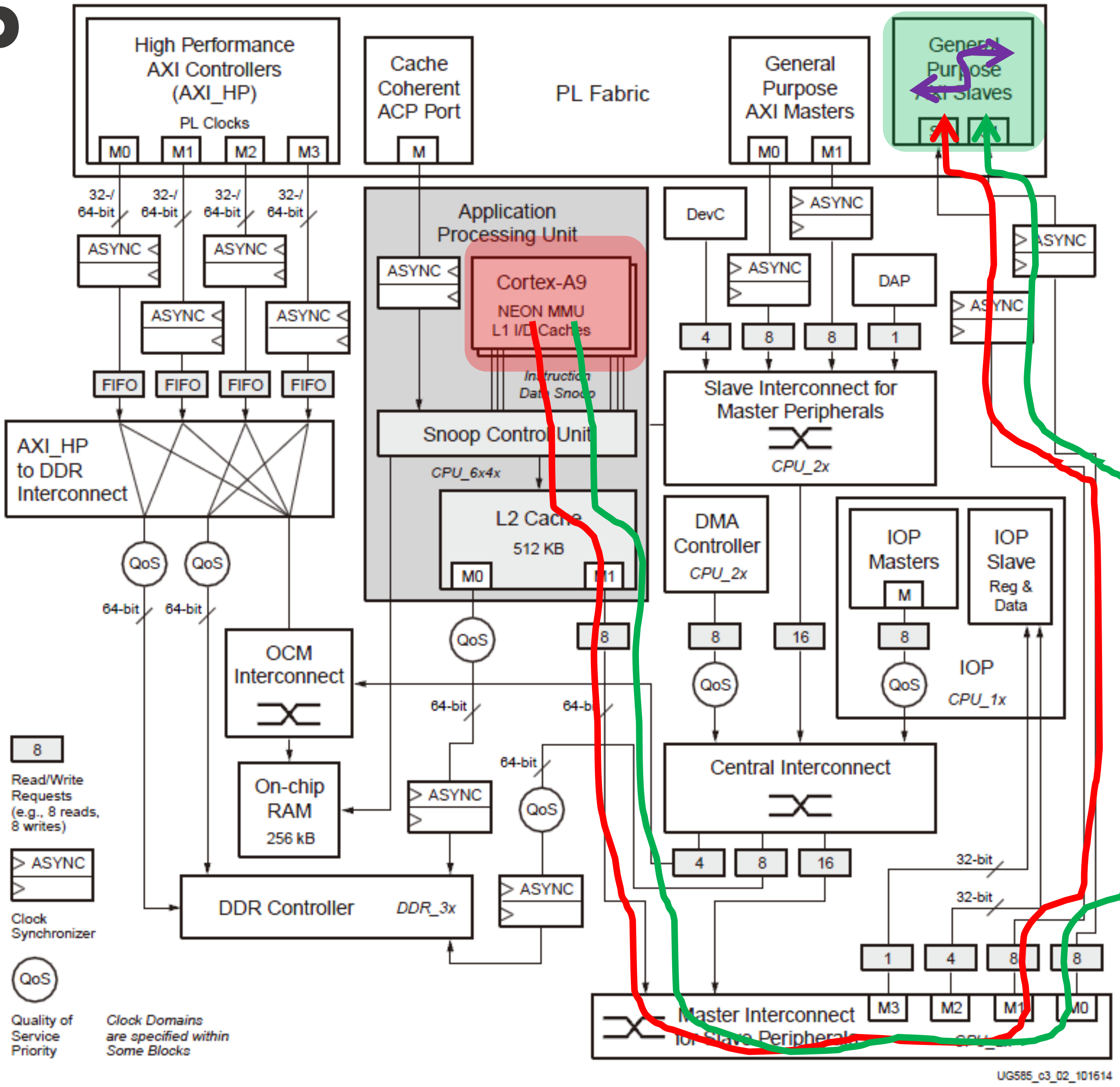


[UG585 Zynq TRM, 2016]

Lab 7: BRAM + My IP

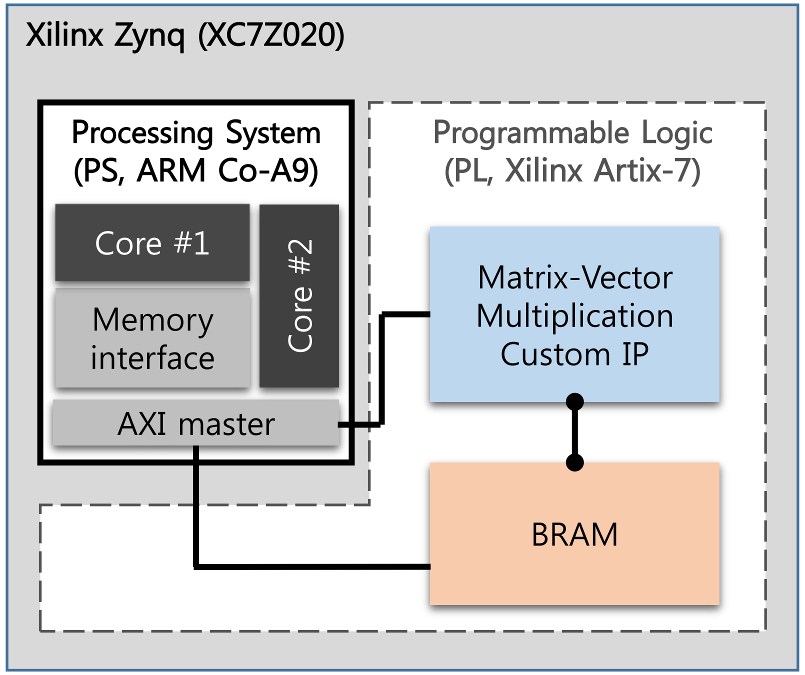
- Interconnect switches
  - Central Interconnect
  - Master Interconnect
  - Slave Interconnect
  - Memory Interconnect
  - OCM Interconnect





Next Week

- Communication between hardware components
  - CPU – bus (interconnect) – BRAM controller
  - BRAM controller – my hardware component
- Split transaction bus
  - ARM AMBA3 AXI protocol
  - Split transaction
    - Multiple read/write channels per port
    - Crossbar interconnect
  - Out of order transaction
    - Deadlock problem and solution



Xilinx Zynq (XC7Z020)
Processing System (PS, ARM Co-A9)
Core #1
Memory interface
Core #2
AXI master
Programmable Logic (PL, Xilinx Artix-7)
Matrix-Vector Multiplication Custom IP
BRAM