COL788: Advanced Topics in Embedded Computing

Lecture 1 – Introduction to Embedded Systems

Vireshwar Kumar
CSE@IITD

August 4, 2022

Semester I
2022-2023

Agenda

- Motivation

Some of the slide contents have been borrowed from those utilized by Prof. Kolin Paul.

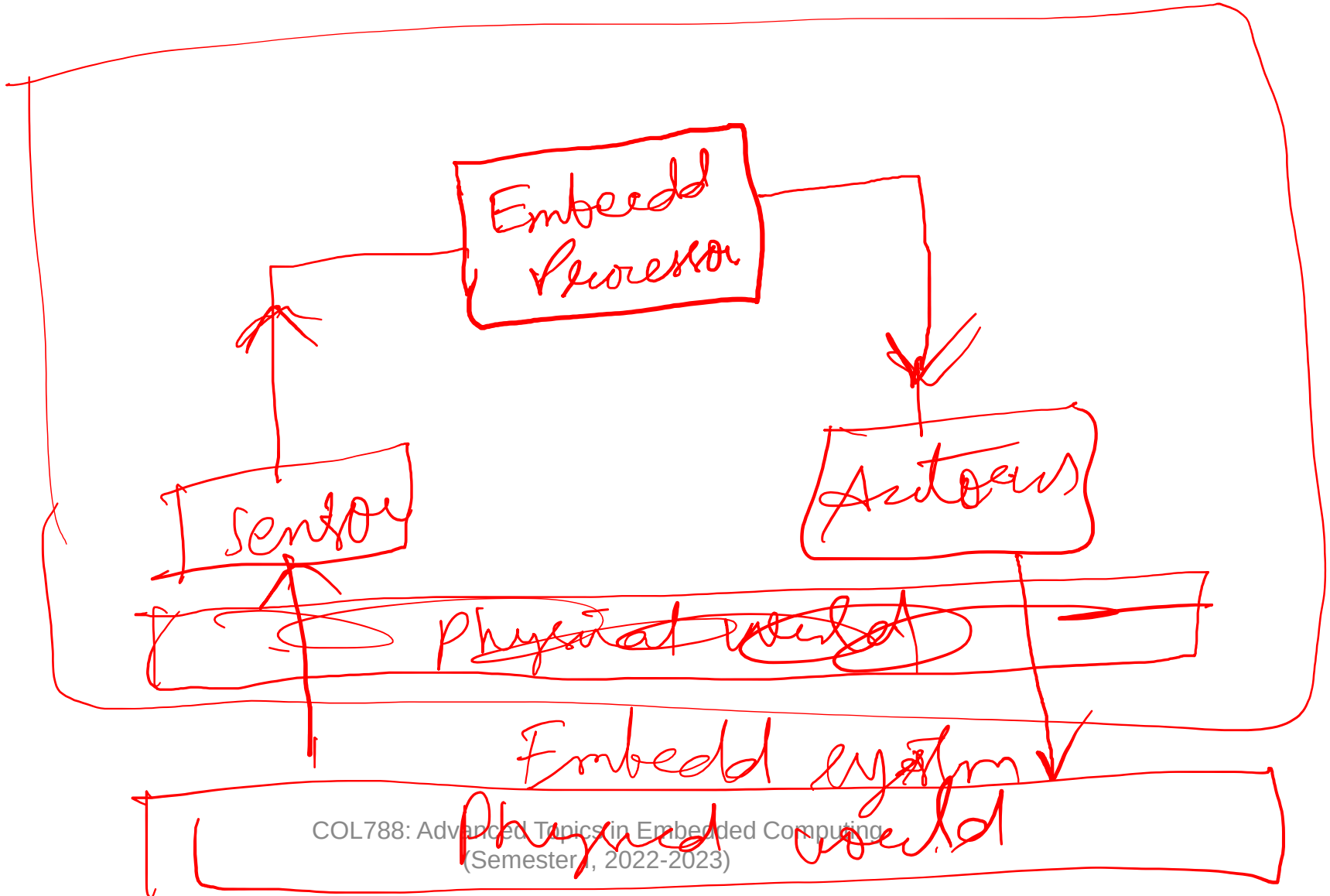
Embedded Systems

- Ubiquitous Invisible Computers
  - Automobiles
  - Drone
  - Printers

- Features
  - Interaction with physical systems
  - Limited resources
    - Storage
    - Computation
    - Communication

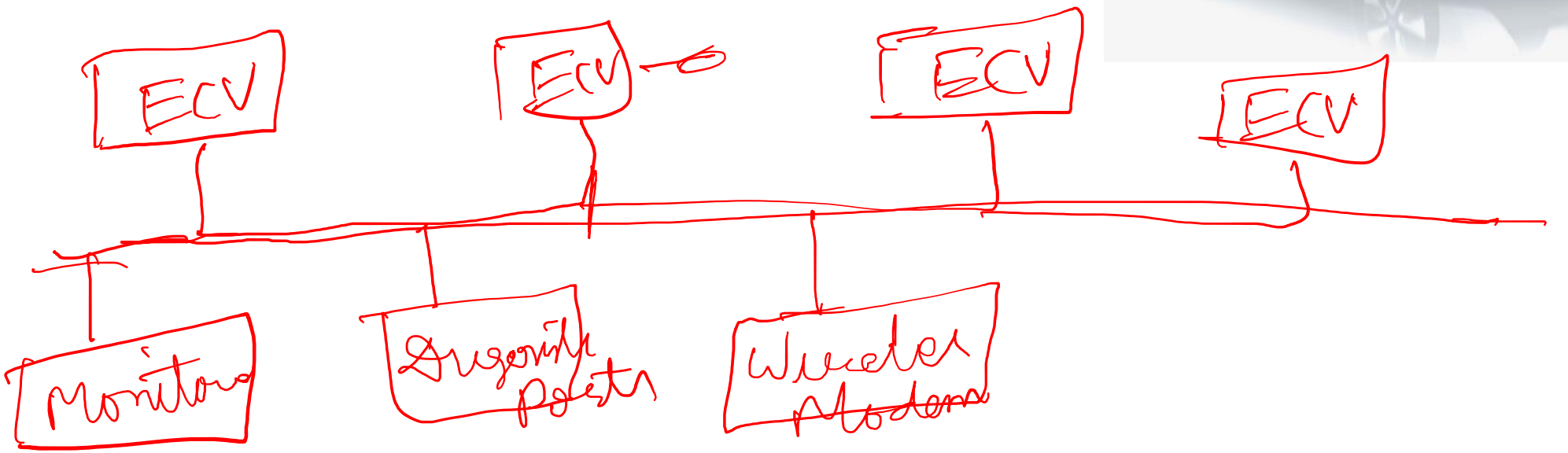
- Critical
  - Boeing 737 Max accidents due to sensor-related errors (The New York Times)

Connecting Cyber and Physical Worlds



Automotives

- Electronic control units (ECUs)
  - 100-200
- Communicate over wired protocols



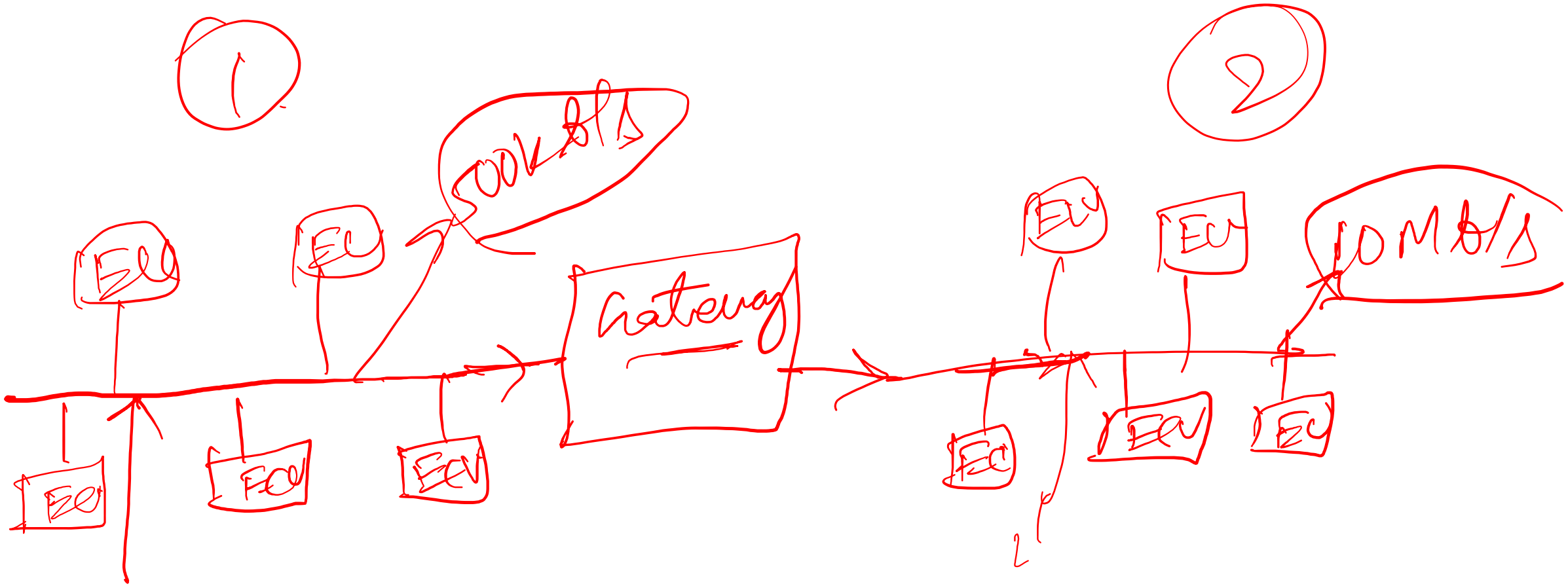
Automotive Functions

- Powertrain
  - Engine control, transmission and gear control
- Chassis
  - Antilock Braking System, Electronic Stability Program, Automatic Stability Control, Adaptive Cruise Control
- Body (comfort)
  - Air conditioning and climate control, dash board, wipers lights, doors, seats, windows, mirrors, cruise control, park distance control
- Telematics
  - Multimedia, infotainment, GPS and in-vehicle navigation systems, CD/DVD players, rear-seat entertainment
- Passive safety (emergency)
  - Rollover sensors, airbags, belt pretensioners

Typical Specifications

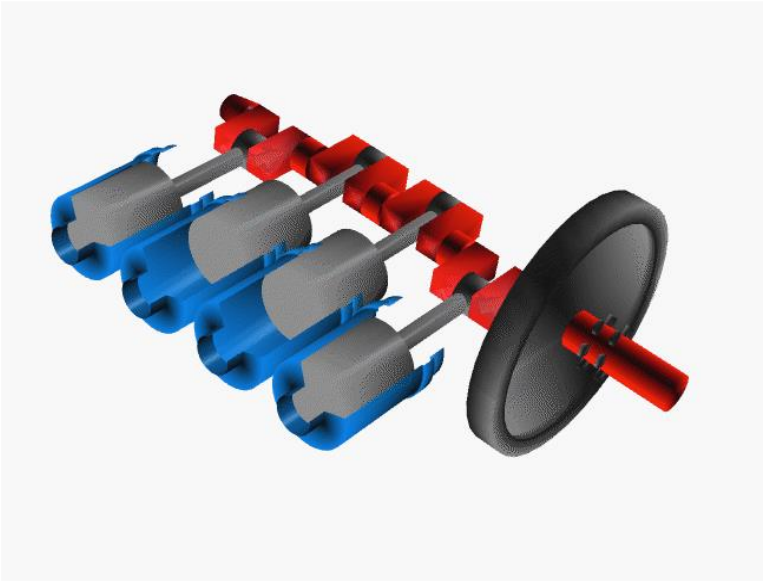
	Powertrain	Chassis	Body	Telematics	Passive safety
Program size	2 MB	4.5 MB	2.5 MB	100 MB	1.5 MB
Number of ECUs	3-6	6-10	14-30	4-12	11-12
Number of messages	36	180	300	660	20
Bus topology	Bus	Bus	Bus	Ring	star
Bandwidth	500 Kb/s	500 Kb/s	100 Kb/s	22 Mb/s	10 Mb/s
Cycle time	10 ms – 10 s	10 ms – 10 s	50 ms 2 s	20 ms 0 5 s	50 ms
Safety requirements	High	High	Low	Low	Very high

Gateways between Buses



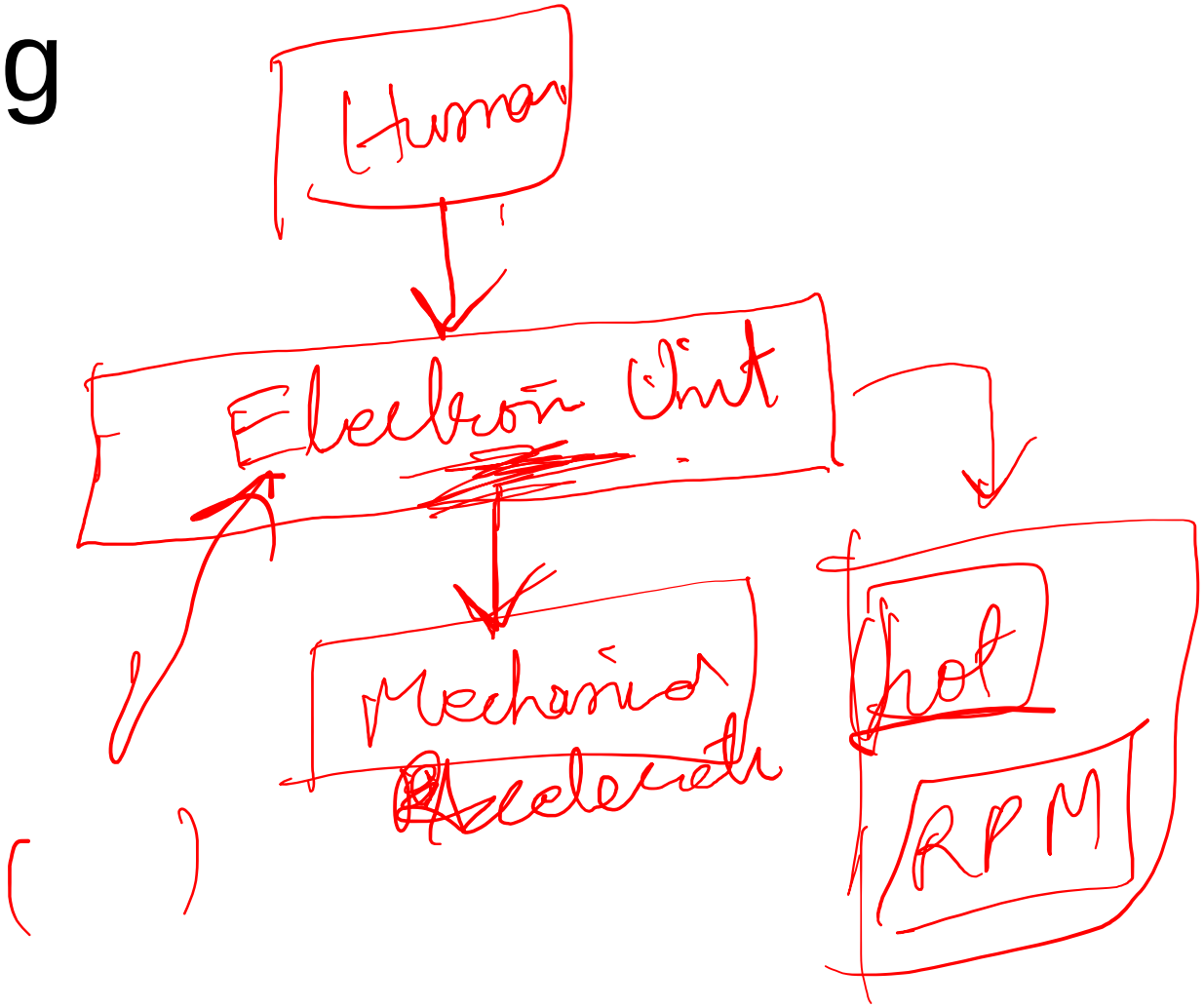
Engine Control

- Task of engine control
  - calculate amount of fuel
  - exact moment of injection
- Dependencies
  - pedal (driver)
  - load of the engine
  - temperature
- Sensors and actuators
  - position of crankshaft
  - valves
- Relevance
  - avoid mechanical damage
  - provide quality of control (e.g., fuel efficiency)



Computer Engineering

- Applications
  - Receive sensor data
  - Perform computation
  - Send actuator data
- Requirements
  - Real-time?



Capabilities of Automotive ECUs

- Computational Constraints
  - Less than 100 MHz
- Communication Constraints
  - Less than 100 bits
- Storage Constrains
  - Less than 100 MB

Experiments on Arduino Uno Board

- https://rweather.github.io/arduinolibs/crypto.html
- Example-1: AES-based Encryption
- Example-2: ECC-based Diffie-Hellman Key Exchange

What’s Next?

- Next Lecture (August 8, 11 am – 12 pm)
  - Lecture 2

COL788: Advanced Topics in Embedded Computing

Lecture 2 – System Architecture

Vireshwar Kumar
CSE@IITD

August 8, 2022

Semester I
2022-2023

Agenda

- System Architecture

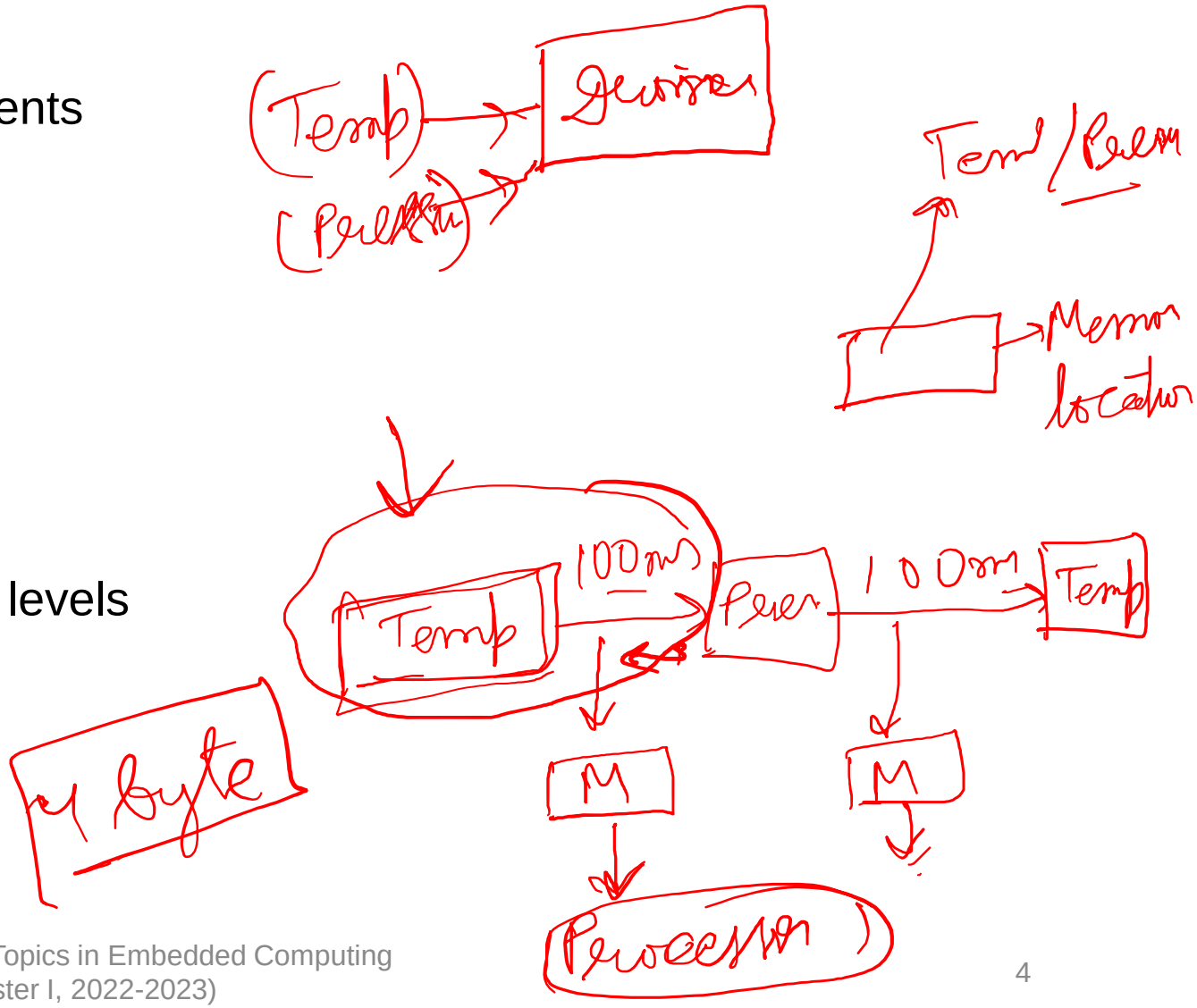
- Book
  - Peter Barry and Patrick Crowley, “Modern embedded computing: Designing connected, pervasive, media-rich systems,” Elsevier, 2nd edition, 2012.

Design Characteristics

- Interaction with the physical world
- Specific task
- Real-time (safety-critical)
- Large numbers
- Low cost
- Resource constraints

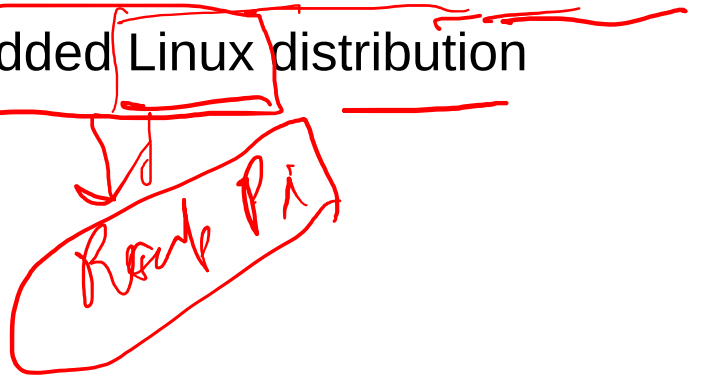
Design Process

- Highly optimized
  - Interactions among different components
  - Detailed implementation details
- Concurrency
  - Timing
- Correctness
  - Modeling at high and low abstraction levels



Operating System

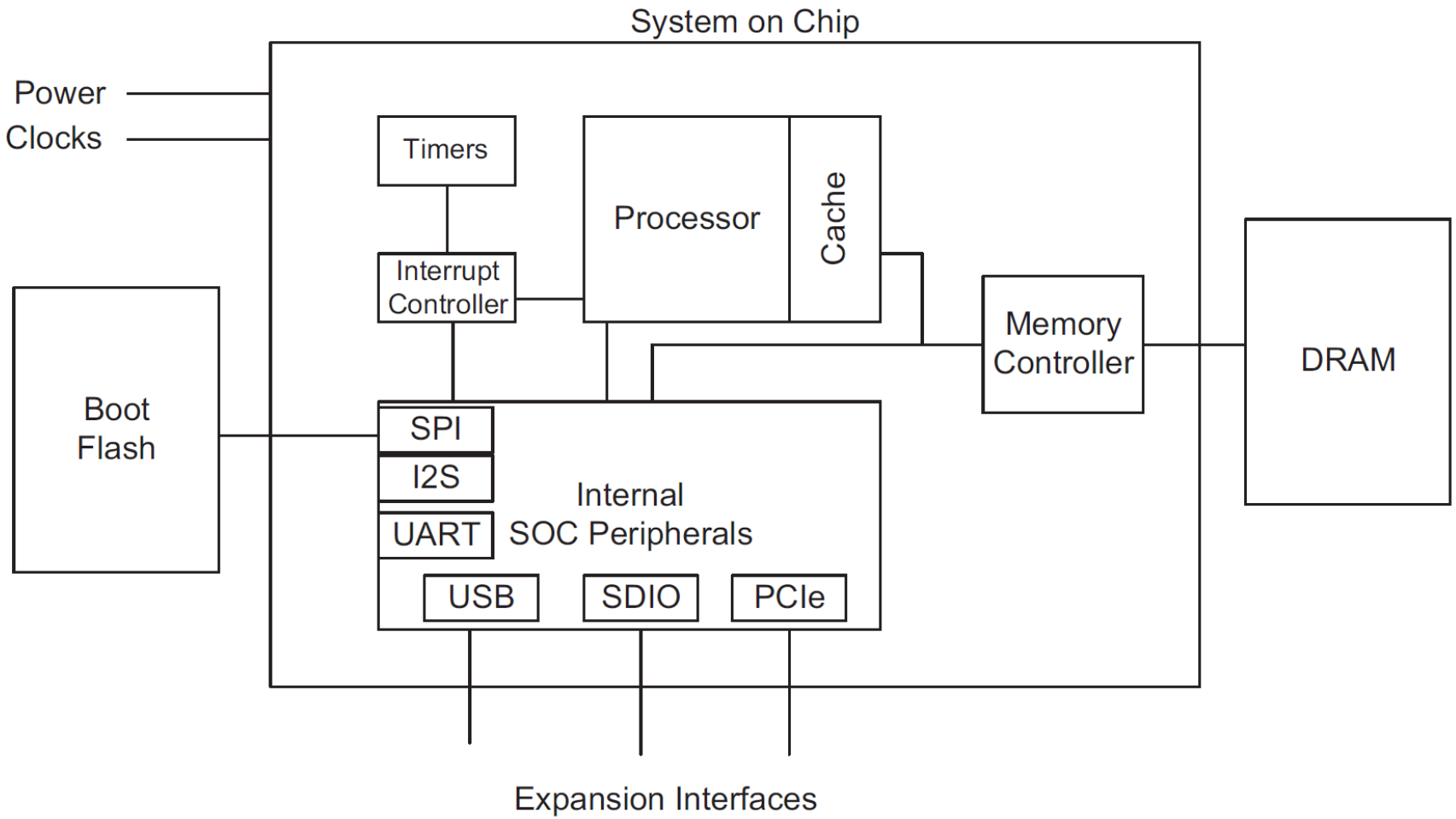
- No OS
- Real-time operating system (RTOS)
- Embedded Linux distribution



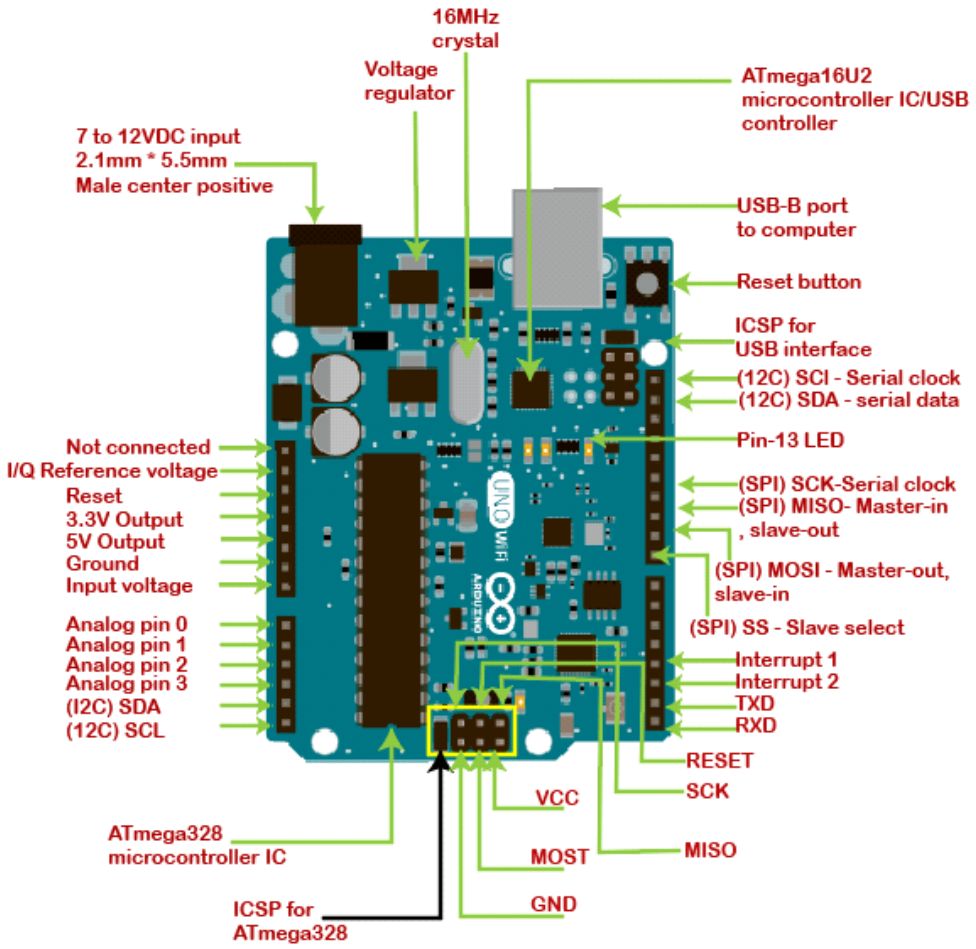
CPU

- Atmel AVR Microcontroller
  - 8-bit
  - Example: Arduino Uno
- MicroChip PIC microcontroller
  - 16-bit
- ARM Cortex-M microcontroller
  - 32-bit
- ARM Cortex-A microcontroller
  - 64-bit
  - Example: Raspberry Pi 4

System on Chip (SoC)



Arduino Uno Board



Parallelism

- Instruction-level
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution

- Data-level
  - Single instruction, multiple data (SIMD)

- Thread-level
  - Multithreading

Instruction Set Architecture (ISA)

- ARM
  - In-order cores
  - Low power and lesser area

- Intel Atom
  - In-order execution
  - Data-level parallelism

What’s Next?

- Next Lecture (August 10, Wednesday, 11 am – 12 pm)
  - Lecture 3

COL788: Advanced Topics in Embedded Computing

Lecture 3 – System Architecture (Cont.)

Vireshwar Kumar
CSE@IITD

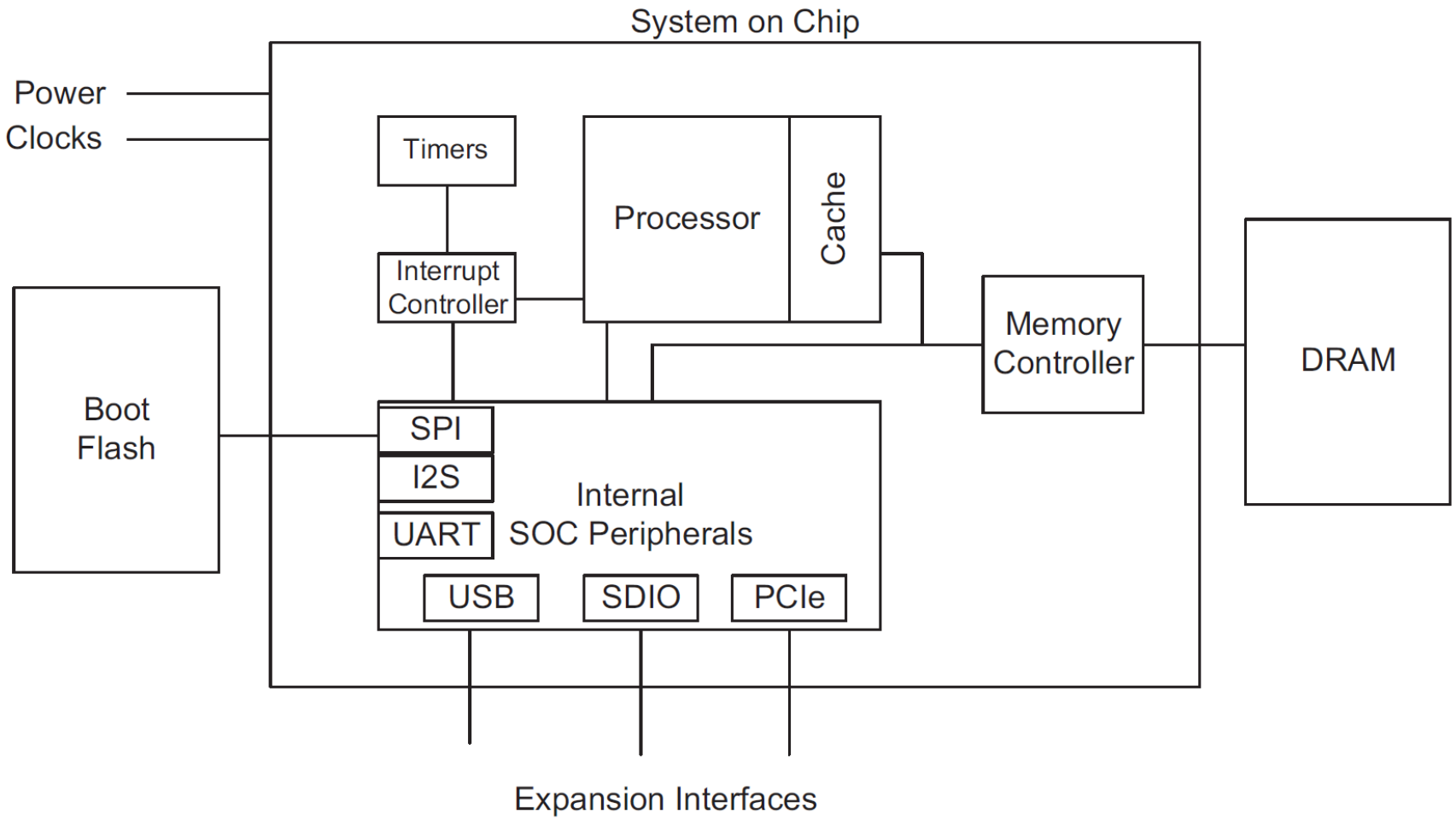
August 10, 2022

Semester I
2022-2023

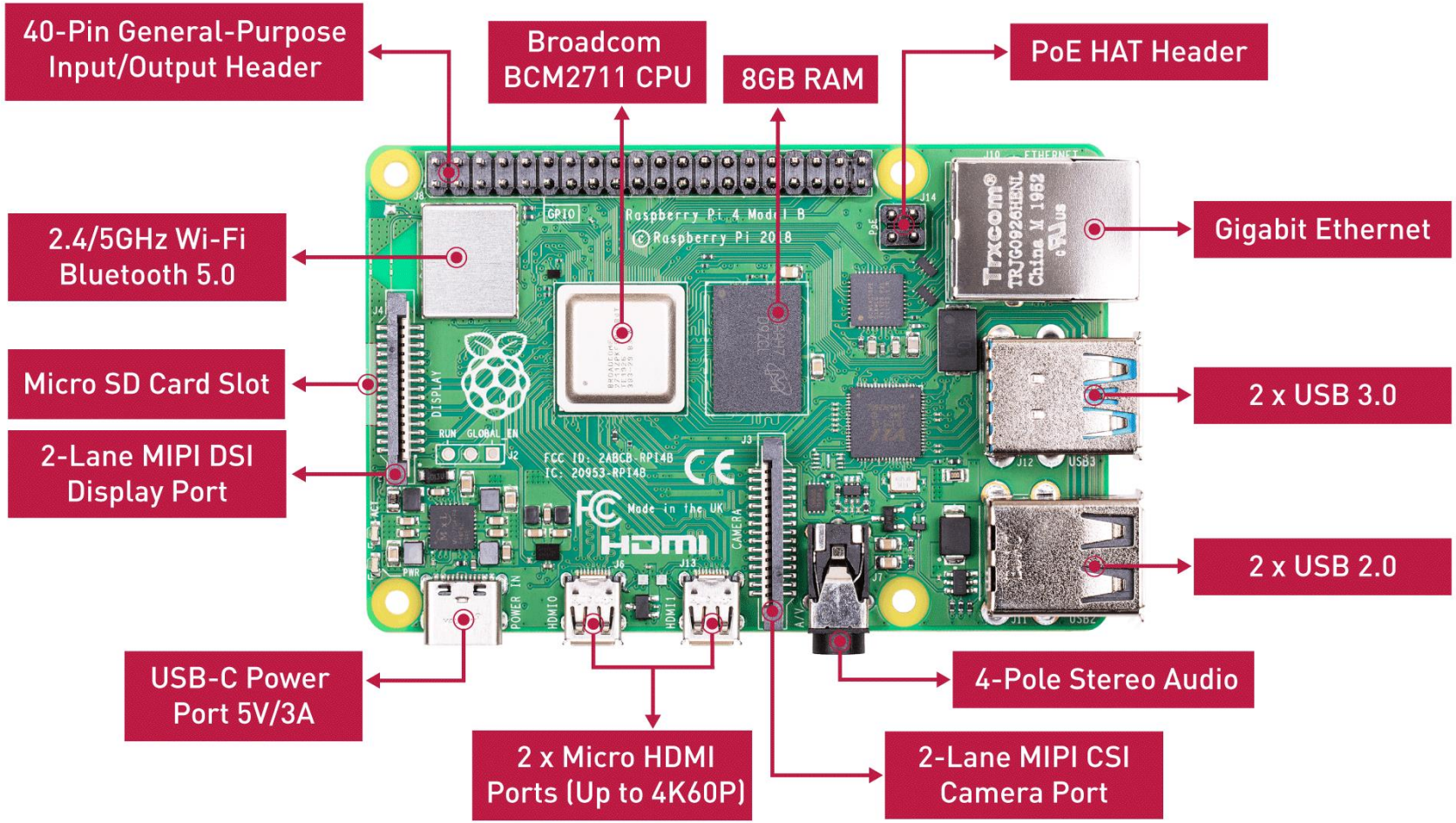
Agenda

- System Architecture

System on Chip (SoC)



Raspberry Pi 4B



Processor: Instruction Set

- Complex Instruction Set Computing (CISC)
  - e.g., Intel
  - Single instruction used for loading, evaluating and storing operations
  - Minimizes the number of instructions per program
  - Increases the number of cycles per instruction

- Reduced Instruction Set Computing (RISC)
  - e.g., ARM
  - Basic instructions for loading, evaluating and storing operations
  - Increases the number of instructions per program
  - Reduces the number of cycles per instruction

Processor: Scalar vs. Superscalar

- Scalar
  - Simple implementation
  - Slow

- Superscalar
  - Supports the parallel execution of instructions
  - Multiple copies of functional units, e.g., two arithmetic logic units
  - More than one instruction per clock cycle

Memory

- List of physical addresses of all the resources on the platform
  - DRAM
  - Interrupt controllers
  - I/O devices

- Usage
  - Processor generates a read or write request
  - Address is decoded by the system memory address decoders
  - Routed to the appropriate physical device to complete the transaction

- Address range
  - Main Memory Address Range: DRAM address accessed by CPU
  - Memory Mapped I/O (MMIO) Range: those decoded to select I/O devices

Memory Mapped I/O Address Range

- Fixed Address Memory Mapped Address
  - BIOS
  - timers
  - interrupt controllers

- Peripheral Component Interconnect Bus

What’s Next?

- Next Lecture (August 17, Wednesday, 11 am – 12 pm)
  - Lecture 4

COL788: Advanced Topics in Embedded Computing

Lecture 4 – System Architecture (Cont.)

Vireshwar Kumar
CSE@IITD

August 17, 2022

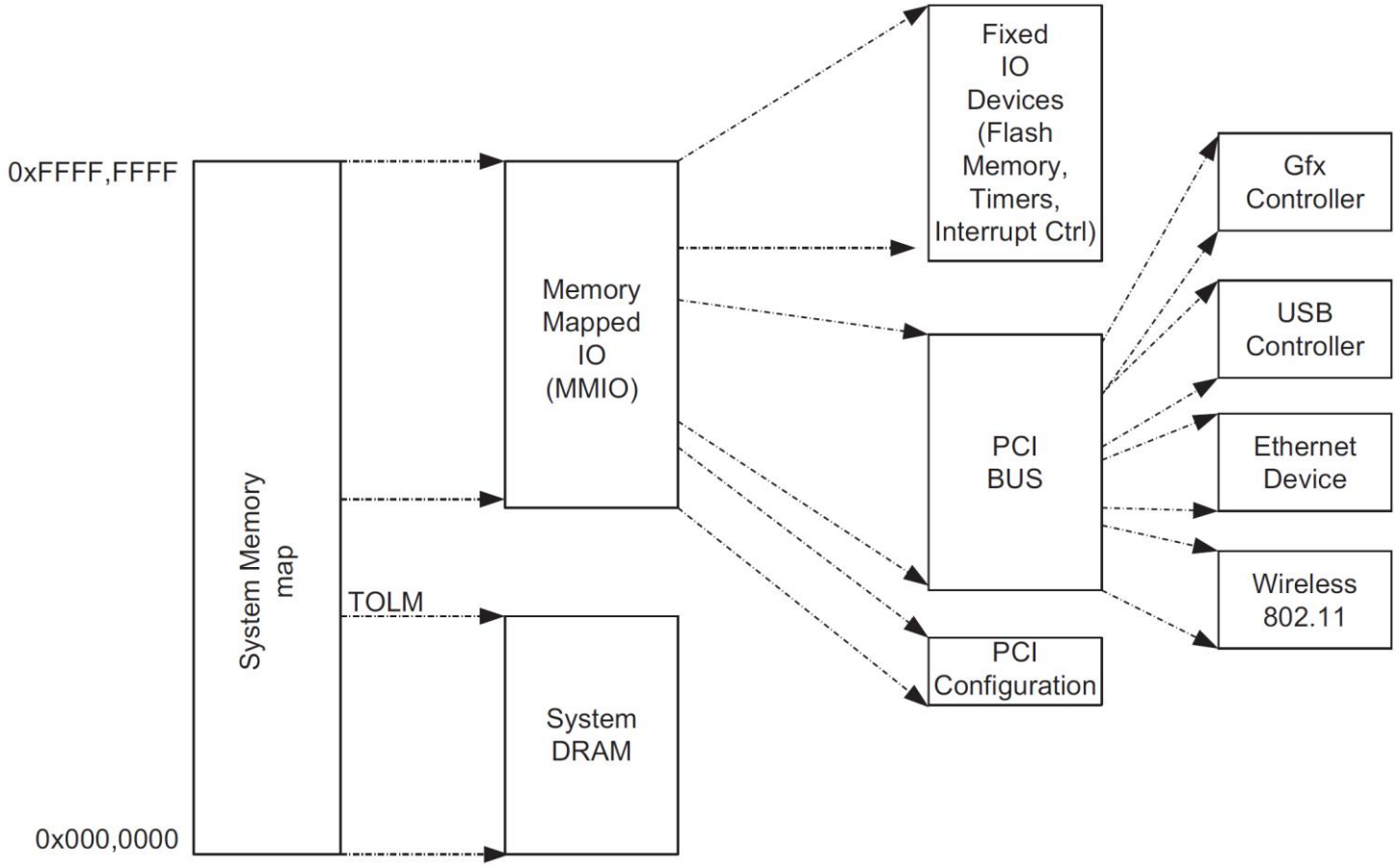
Semester I
2022-2023

Agenda

- Interrupt
- Timer

Last Lecture

- Processor
  - CISC vs RISC
- Memory
  - Main Memory
  - MMIO



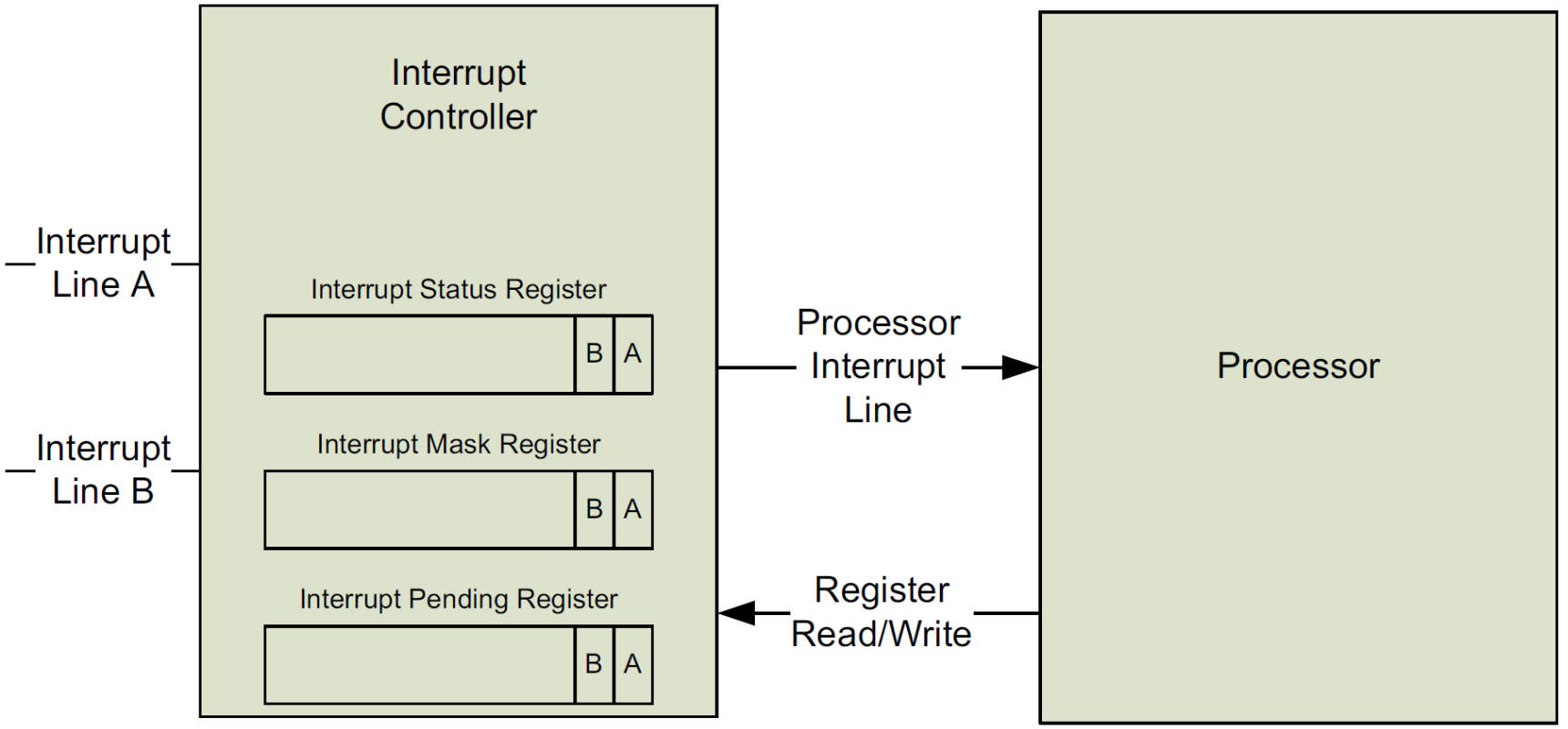
Interrupt

- Signal produced by a hardware
  - Temporarily halts the current process
  - Interrupt service routine (ISR)

- Mitigates the need for continuous monitoring
  - Keeps the processor free

- Interrupt controller
  - Collects the interrupt events
  - Labels the events
  - Routes to the processor

Interrupt Controller



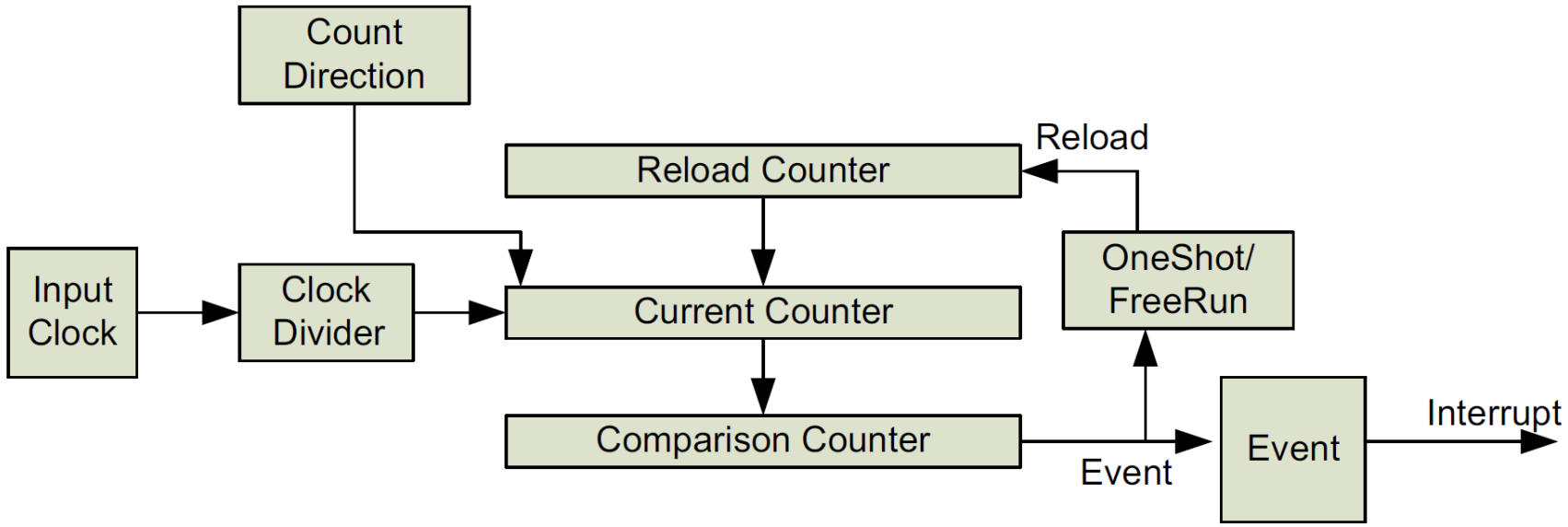
Timer

- Hardware-based counter driven by a clock

- Every embedded system has at least one timer
  - Clock divider

- Attributes
  - Accuracy (parts per million (PPM))
    - 25 PPM >>> around 2 seconds per day (25* 86400 seconds per day)
  - Count direction
  - Counter size
  - One-shot or free-run

Timer Configuration



What’s Next?

- Next Lecture (August 20, Saturday, 11 am – 12 pm)
  - Lecture 5
  - Venue: Teams

COL788: Advanced Topics in Embedded Computing

Lecture 5 – System Architecture (Cont.)

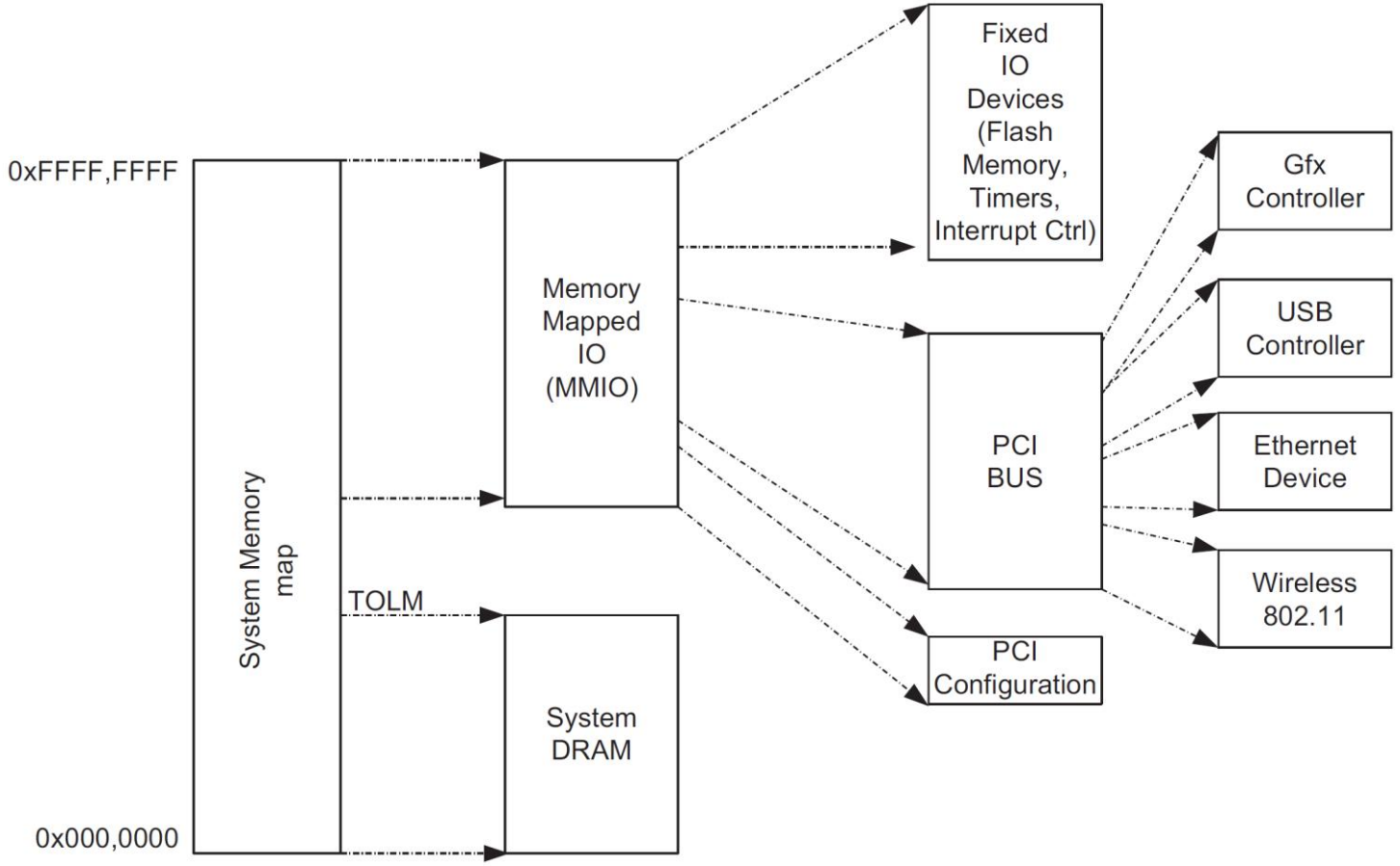
Vireshwar Kumar
CSE@IITD

August 20, 2022

Semester I
2022-2023

Last Lectures on System Architecture

- Processor
  - CISC vs RISC
- Memory Usage
  - Main Memory
  - MMIO
- Interrupt
- Timer

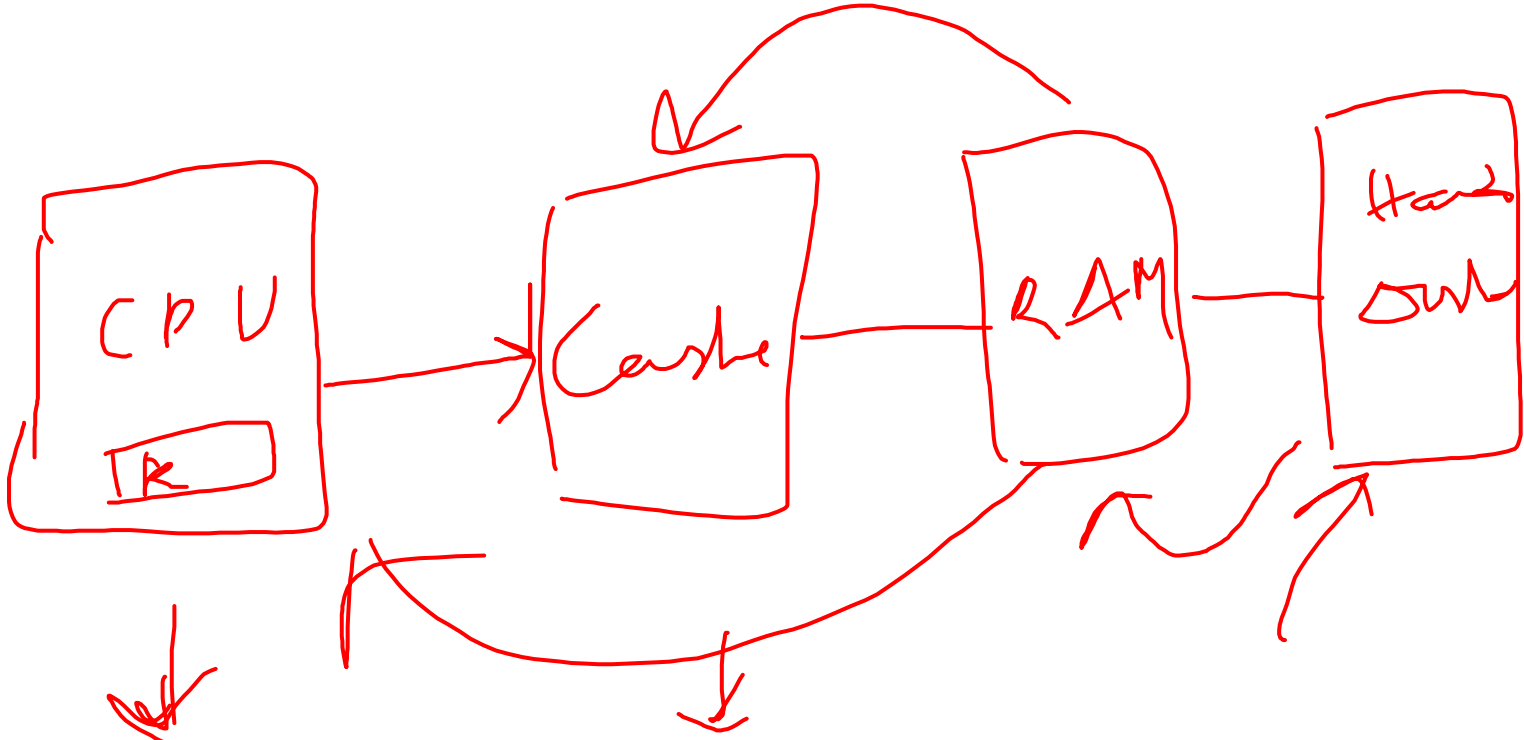


Agenda

- Memory types
- I/O types

Memory Types

- Register
- Cache
- Random Access Memory
- Disk Memory



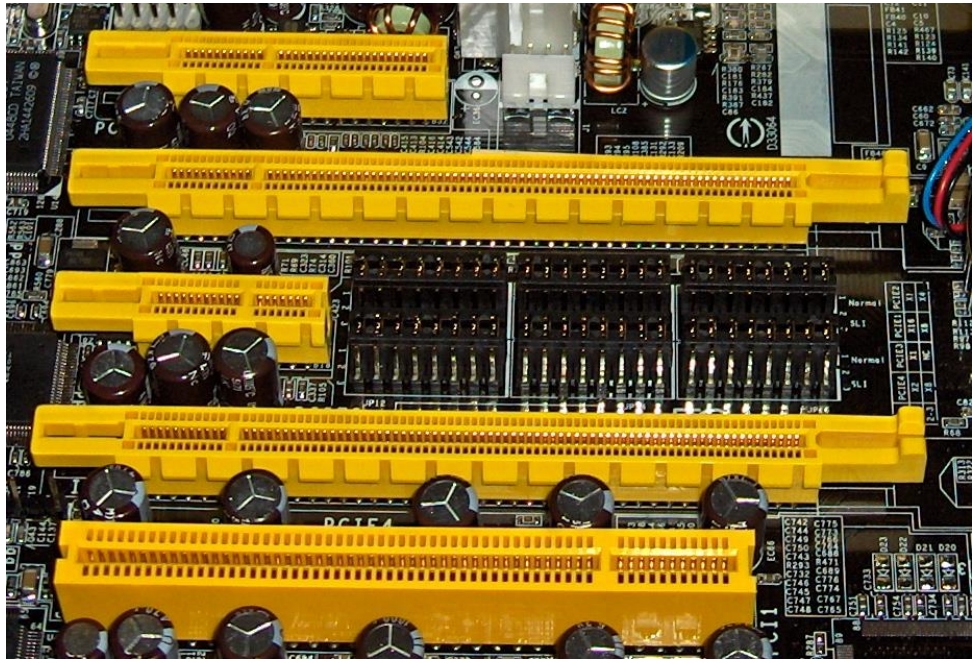
	Register	Cache	Random Access Memory	Disk Memory
Size	32 Bytes	64 KBs	128 MBs	32 GBs
Speed	3 ns	6 ns	60 ns	8 ms
Cost (Rupees)	Included with CPU	10,000/MB	100/MB	1/MB

Input/Output Types

- High Speed
  - Peripheral Component Interconnect (PCI)
  - Universal Serial Bus (USB)
- Low Speed
  - Inter-Integrated Circuit (I2C) bus
  - Serial Peripheral Interface (SPI) bus
  - Universal Asynchronous Receiver/Transmitter (UART)

PCI Express

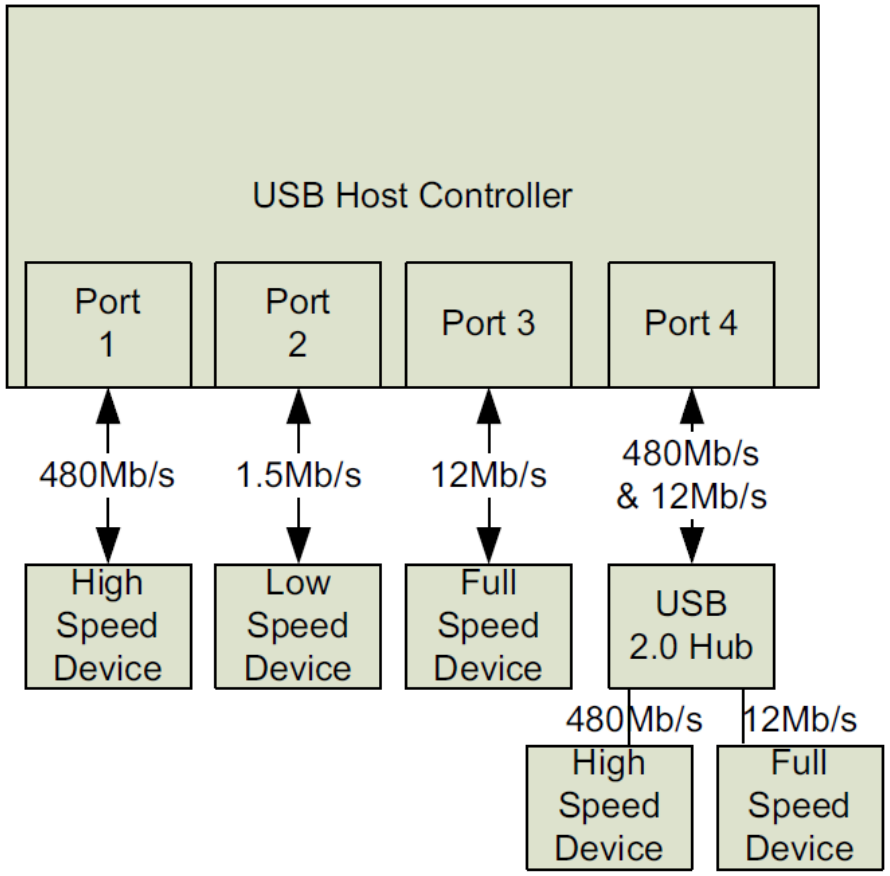
- Memory
- Graphics adapter
- Network interface



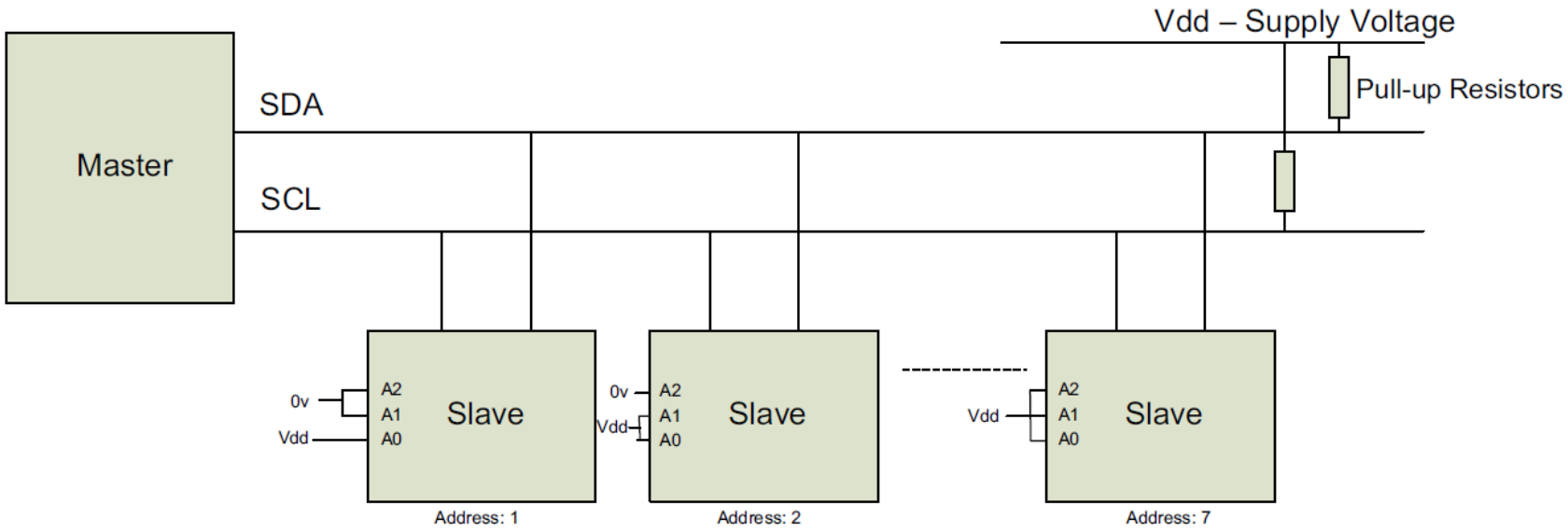
USB

- Printer
- Keyboard
- Mouse

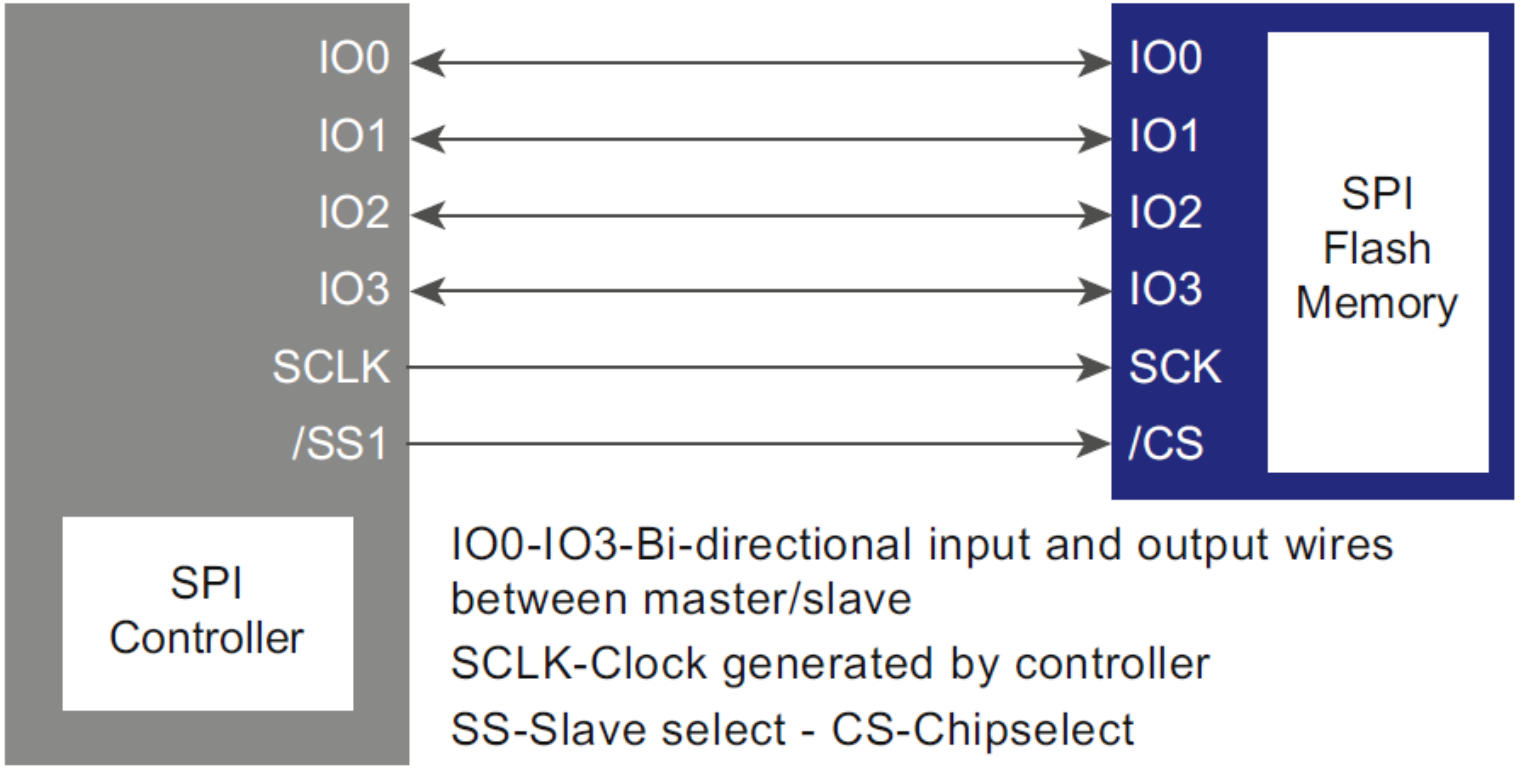
- Programming Interface



Inter-Integrated Circuit (I2C)



Serial Peripheral Interface (SPI)



IO0-IO3-Bi-directional input and output wires between master/slave

SCLK-Clock generated by controller

SS-Slave select - CS-Chipselect

What’s Next?

- Next Lecture (August 22, Monday, 11 am – 12 pm)
  - Lecture 6

COL788: Advanced Topics in Embedded Computing

Lecture 6 – Processor Architecture

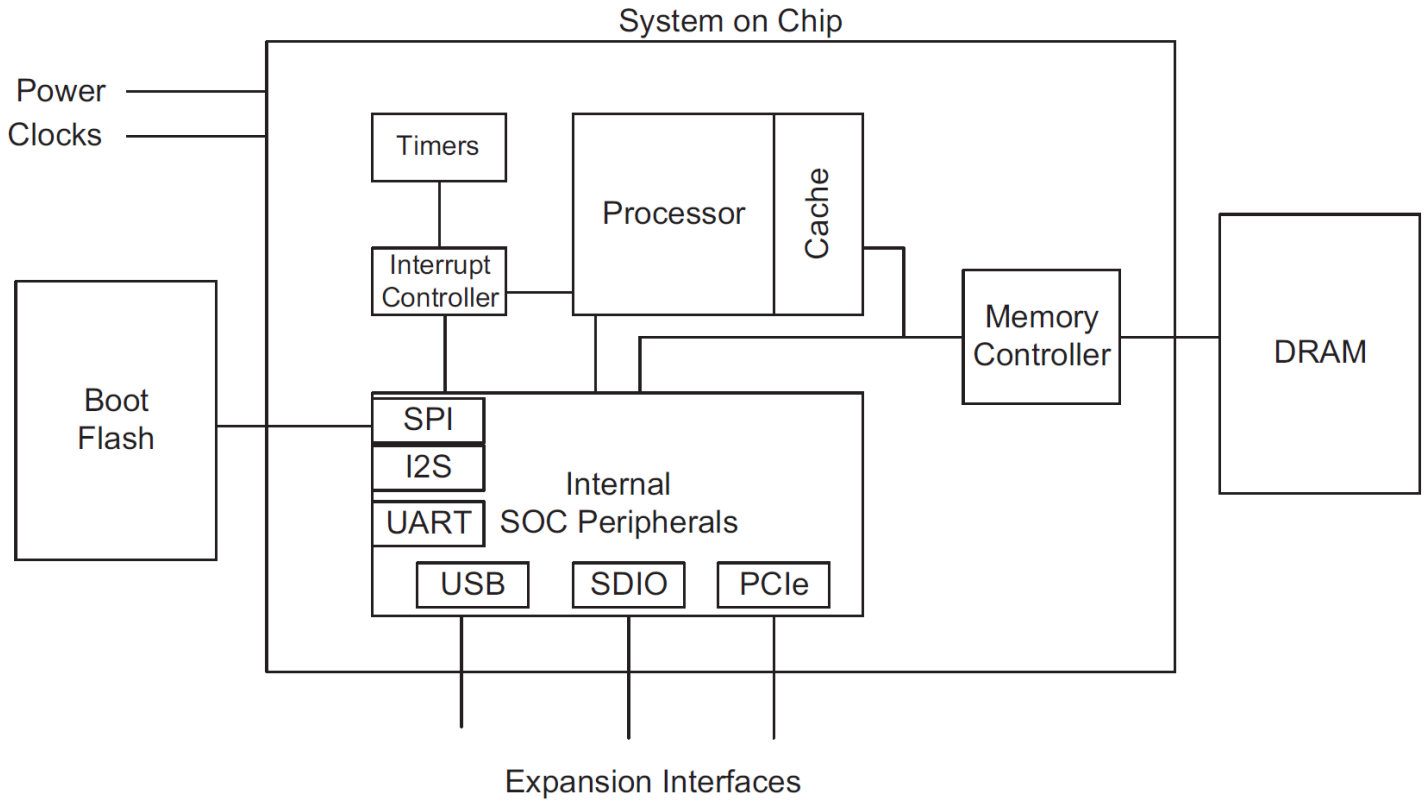
Vireshwar Kumar
CSE@IITD

August 22, 2022

Semester I
2022-2023

Last Lectures on System Architecture

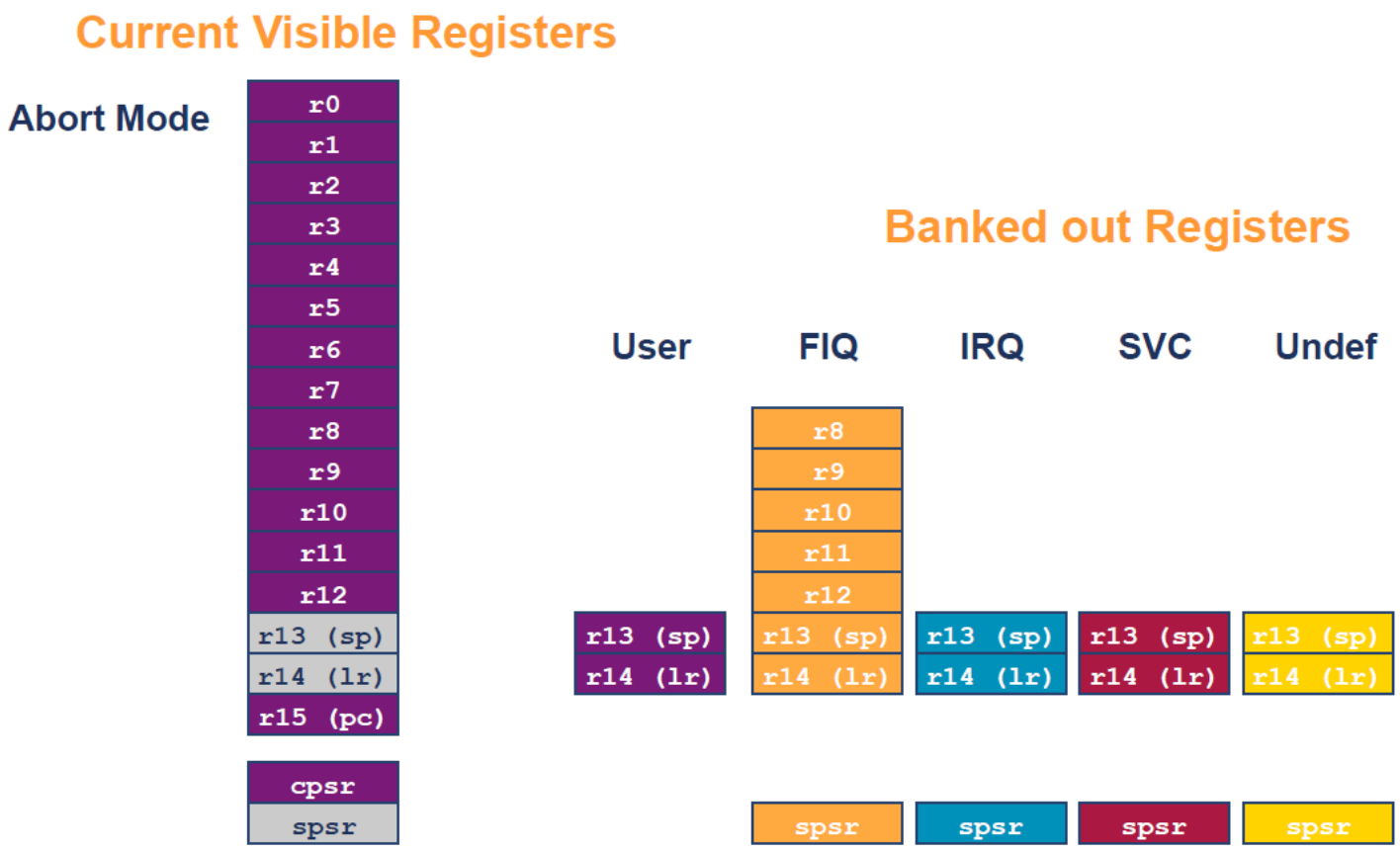
- Processor
- Memory
- Interrupt
- Timer
- I/O Interfaces



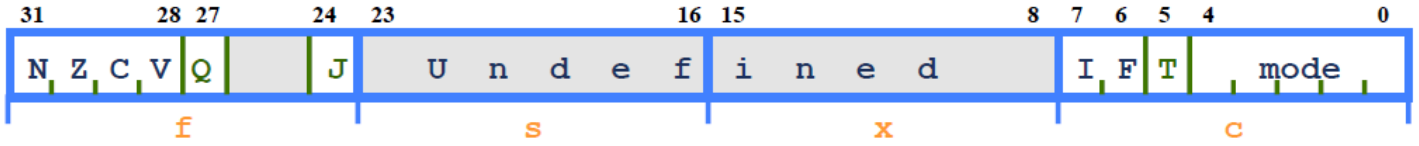
ARM Processor Architecture

- The ARM has seven basic operating modes:
  - User : unprivileged mode under which most tasks run
  - FIQ : entered when a high priority (fast) interrupt is raised
  - IRQ : entered when a low priority (normal) interrupt is raised
  - Supervisor : entered on reset and when a Software Interrupt instruction is executed
  - Abort : used to handle memory access violations
  - Undef : used to handle undefined instructions
  - System : privileged mode using the same registers as user mode

Register Set



Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Sticky Overflow flag - Q flag**
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- **J bit**
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- **Interrupt Disable bits.**
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- **T Bit**
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **Mode bits**
  - Specify the processor mode

Condition Codes

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Example: Conditional Codes

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

What’s Next?

- Next Lecture (August 24, Wednesday, 11 am – 12 pm)
  - Lecture 7

COL788: Advanced Topics in Embedded Computing

Lecture 7 – Processor Architecture (Cont.)

Vireshwar Kumar
CSE@IITD

August 24, 2022

Semester I
2022-2023

Last Lecture

- ARM Registers
  - General purpose registers (r0-r14)
  - Dedicated program counter (r15)
  - Dedicated current program status register (cpsr)
  - Dedicated saved program status registers (spsr)
- Program Status Registers (cpsr and spsr)
  - Flags
  - Interrupt
  - Mode
- Calling operations with conditions

Arithmetic Operations

- Operations are:
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparisons

- The only effect of the comparisons is to update the condition flags
  - Hence, there is no need to set S bit
- Operations are:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 EOR operand2, but result not written
- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:
  - CMP r0, r1
  - TSTEQ r2, #5

Logical Operations

- Operations are:
  - AND operand1 AND operand2
  - EOR operand1 EOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [ie bit clear]
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3,r0

Data Movement

- Operations are:
  - MOV operand2
  - MVN NOT operand2

  Note that these make no use of operand1.
- Syntax:
  - <Operation>{<cond>}{S} Rd, Operand2
- Examples:
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

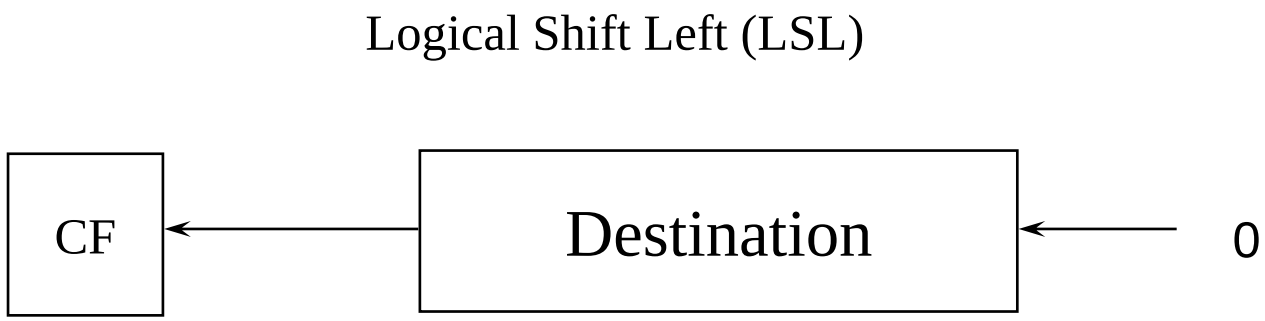
The Barrel Shifter

- The ARM doesn’t have actual shift instructions.

- Instead, it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.

Barrel Shifter - Left Shift

- Shifts left by the specified amount (multiplies by powers of two) e.g.
  LSL #5 = multiply by 32

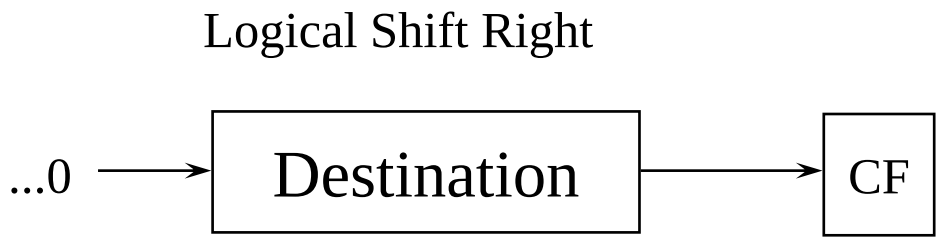


Barrel Shifter - Right Shifts

Logical Shift Right

- Shifts right by the specified amount (divides by powers of two) e.g.

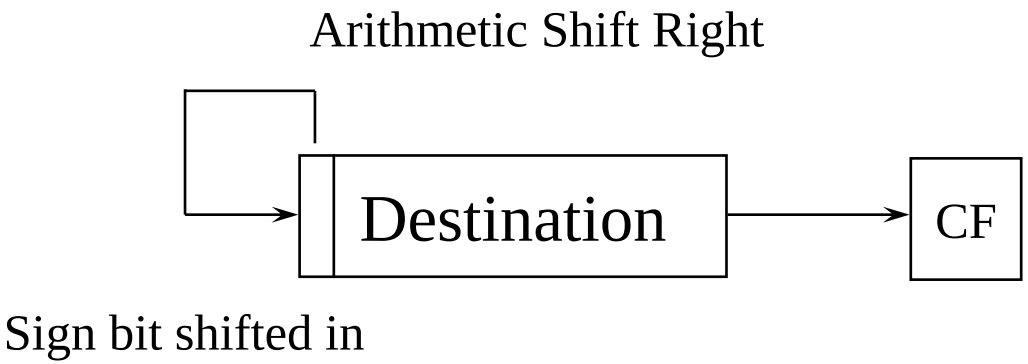
  LSR #5 = divide by 32



Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

  ASR #5 = divide by 32



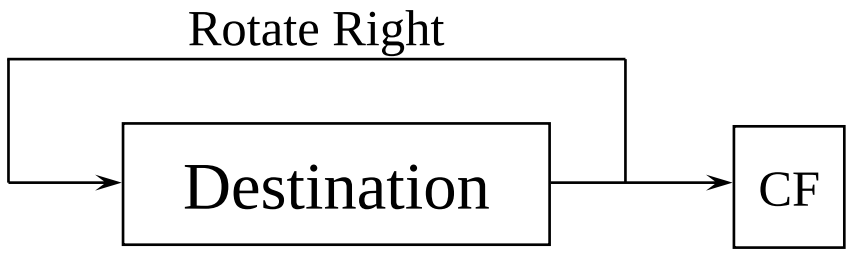
Barrel Shifter - Rotations

Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

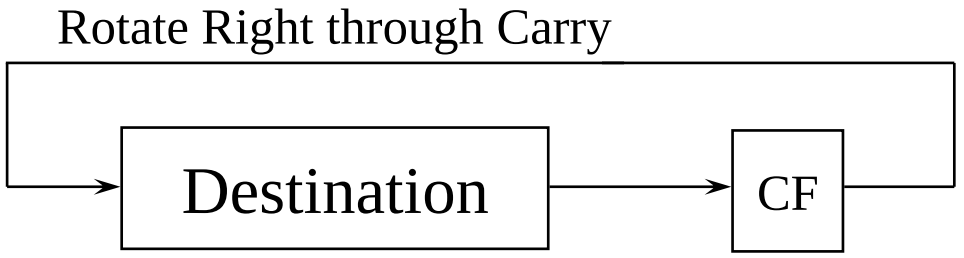
  e.g., ROR #5

- Note the last bit rotated is also used as the Carry Out.

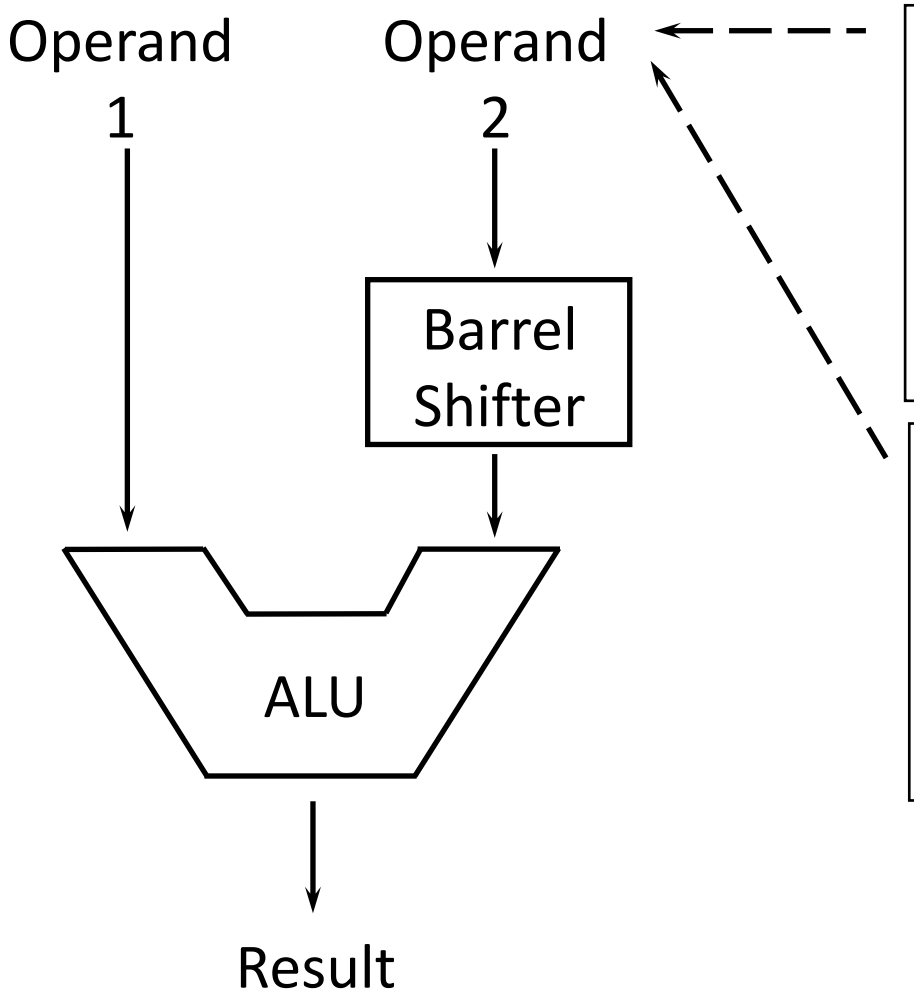


Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.



Using the Barrel Shifter: Second Operand



- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

* Immediate value

- 8 bit number
- Can be rotated right through an even number of positions.

Second Operand: Shifted Register

- The amount by which the register is to be shifted is contained in either:
  - the immediate 5-bit field in the instruction
    - No overhead as the Shift is done for free - executed in a single cycle.
  - the bottom byte of a register (not PC)
    - Then takes extra cycle to execute
    - ARM doesn’t have enough read ports to read 3 registers at once.
    - Then same as on other processors where shift is a separate instruction.
- If no shift is specified, then a default shift is applied: LSL #0
  - i.e., barrel shifter has no effect on value in register.

What’s Next?

- Next Lecture (August 25, Thursday, 12 pm – 1 pm)
  - Lecture 8

COL788: Advanced Topics in Embedded Computing

Lecture 8 – Processor Architecture (Cont.)

Vireshwar Kumar
CSE@IITD

August 25, 2022

Semester I
2022-2023

Last Lecture

- Arithmetic Operations
- Comparisons
- Logical Operations
- Data Movement
- Barrel Shifter

Immediate Value

- There is no single instruction which will load a 32-bit constant into a register without performing a data load from memory.
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data.
- The data processing instruction format has 12 bits available for operand2
  - If used directly this would only give a range of 4096.
- Instead it is used to store 8 bit constants, giving a range of 0 - 255.
- These 8 bits can then be rotated right through an even number of positions (ie RORs by 0, 2, 4,..30).
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

Range of Immediate Value

- This gives us:
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
- Example:
  - MOV r0, #0x40, 26 => MOV r0, #0x1000

Multiplication using a Shifted Register

- Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- Example:
  - r0 = r1 * 5 = r1 + (r1 * 4)
  - ADD r0, r1, r1, LSL #2
- Example:
  - r2 = r3 * 105 = r3 * 15 * 7 = r3 * (16 - 1) * (8 - 1)
  - RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
  - RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

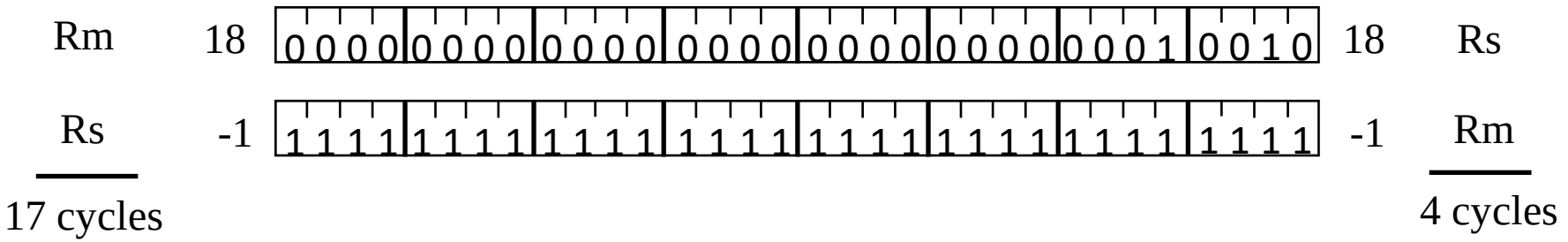
Multiplication Instructions

- The Basic ARM provides two multiplication instructions.
- Multiply
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
- Multiply Accumulate - does addition for free
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- Restrictions on use:
  - Rd and Rm cannot be the same register
    - Can be avoid by swapping Rm and Rs around. This works because multiplication is commutative.
  - Cannot use program counter

  These will be picked up by the assembler if overlooked.
- Operands can be considered signed or unsigned
  - Up to user to interpret correctly.

Multiplication Implementation

- The ARM makes use of Booth's Algorithm to perform integer multiplication.
- On some ARMs this operates on 2 bits of Rs at a time.
  - For each pair of bits this takes 1 cycle (plus 1 cycle to start with).
  - However when there are no more 1's left in Rs, the multiplication will early-terminate.
- Example: Multiply 18 and -1 : Rd = Rm * Rs



Loading 32-bit Constants

- Although the MOV/MVN mechanism will load a large range of constants into a register, sometimes this mechanism will not generate the required constant.
- Therefore, the assembler also provides a method which will load any 32-bit constant:
  - `LDR rd,=numeric constant`
- If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated.
- Otherwise, the assembler will produce an LDR instruction with a PC-relative address to read the constant from a literal pool.
  - `LDR r0,=0x42 ; generates MOV r0,#0x42`
  - `LDR r0,=0x55555555 ; generate LDR r0,[pc, offset to lit pool]`
- As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.

Load / Store Instructions

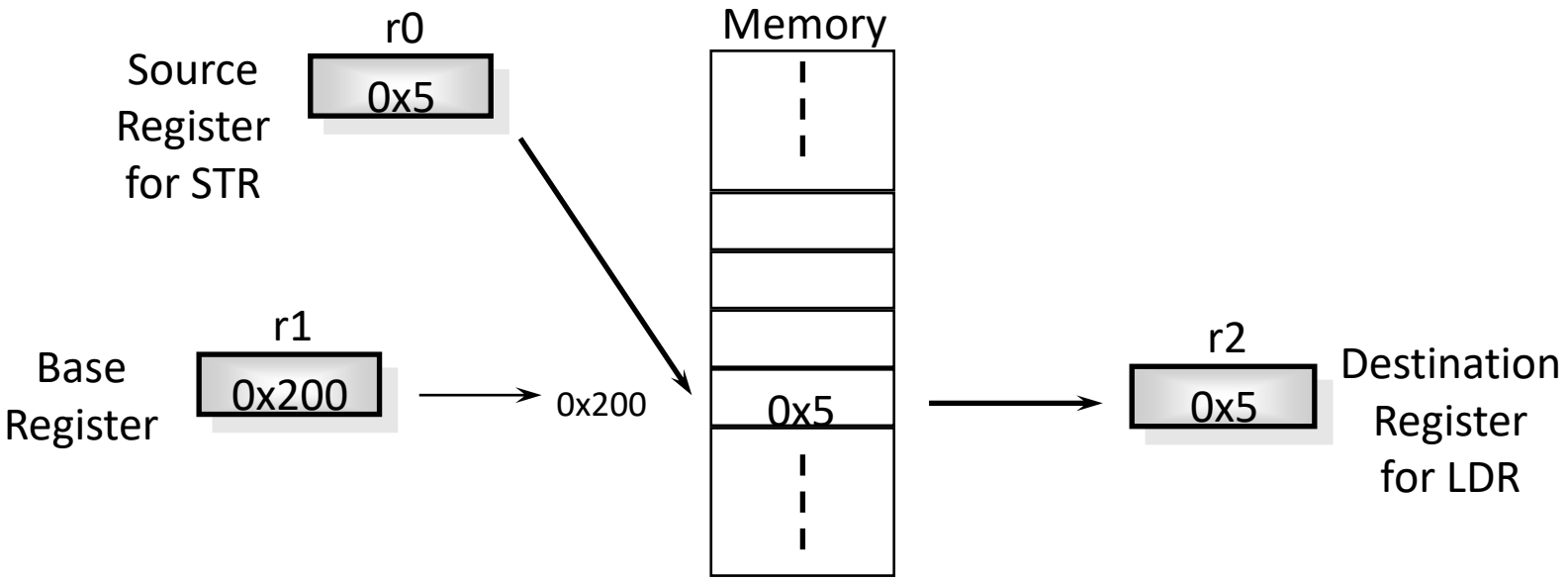
- The ARM is a Load / Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- This might sound inefficient, but in practice isn’t:
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

Single Register Data Transfer

- The basic load and store instructions are:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture Version 4 also adds support for halfwords and signed data.
  - Load and Store Halfword
    - LDRH / STRH
  - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.
  - e.g. LDREQB
- Syntax:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load and Store: Base Register

- The memory location to be accessed is held in a base register
  - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.

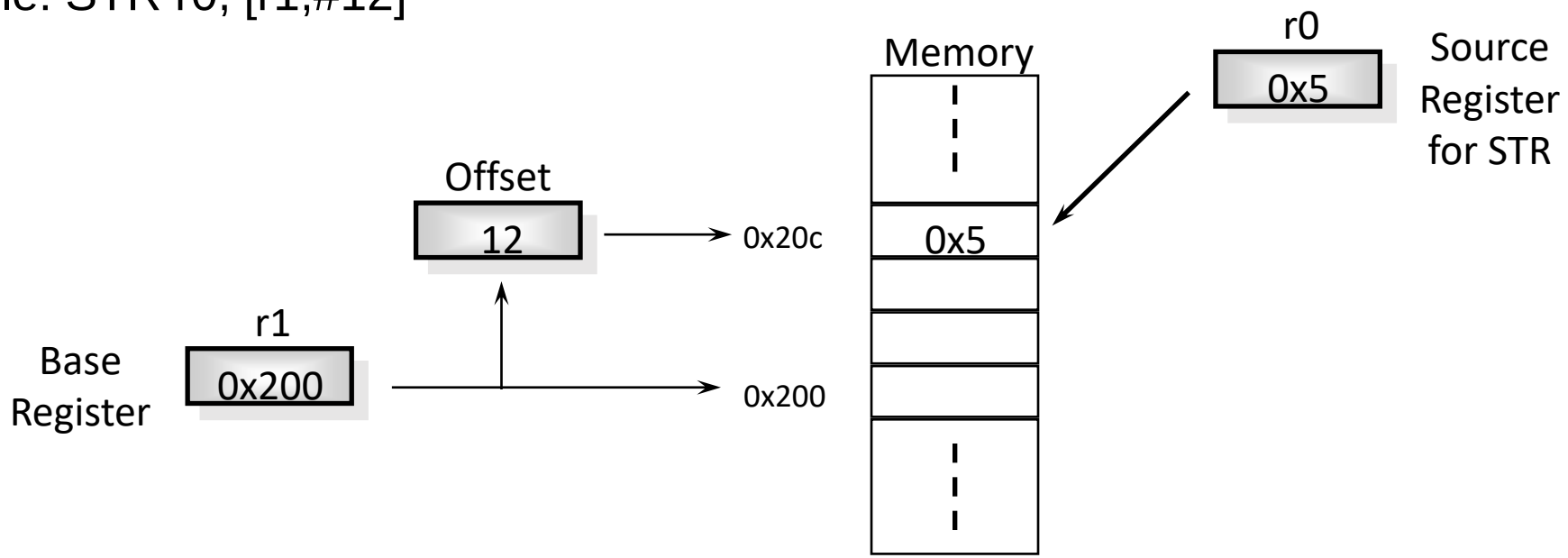


Load and Store: Offsets from Base Register

- As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.
- This offset can be
  - An unsigned 12bit immediate value (ie 0 - 4095 bytes).
  - A register, optionally shifted by an immediate value
- This can be either added or subtracted from the base register:
  - Prefix the offset value or register with '**+**' (default) or '**-**'.
- This offset can be applied:
  - before the transfer is made: ***Pre-indexed addressing***
    - optionally *auto-incrementing* the base register, by postfixing the instruction with an '**!**'.
  - after the transfer is made: ***Post-indexed addressing***
    - causing the base register to be *auto-incremented*.

Load and Store: Pre-indexed Addressing

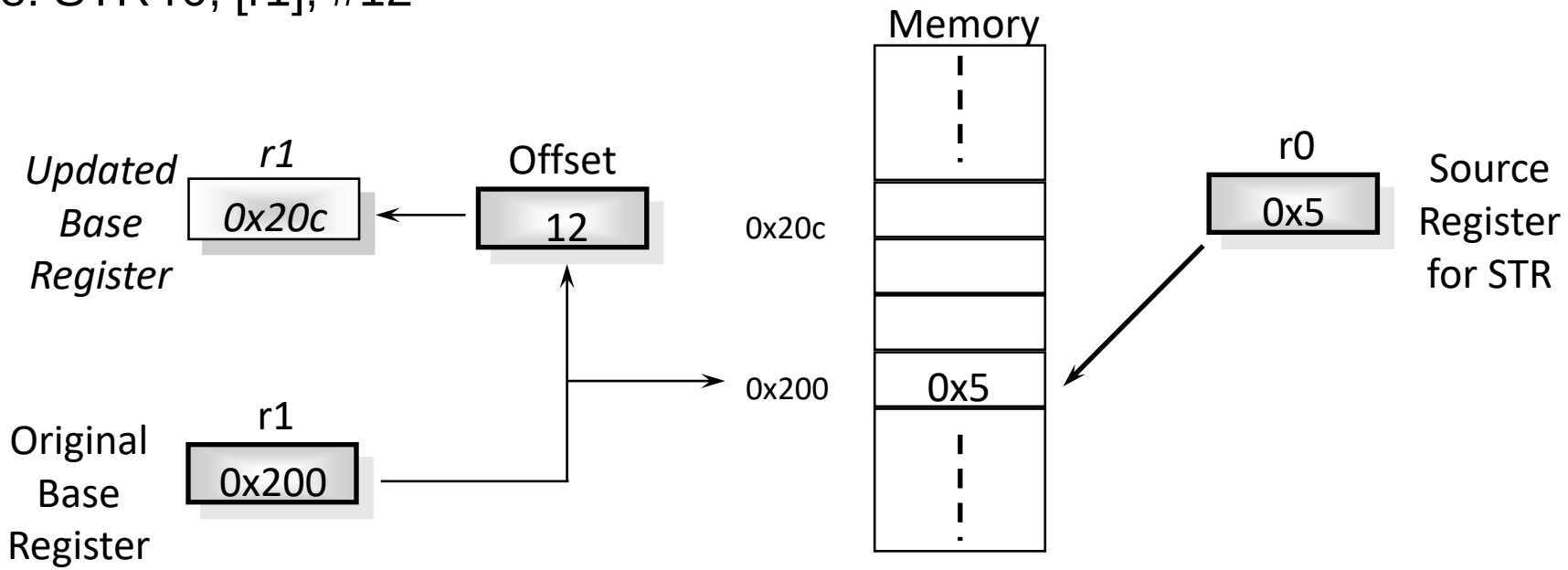
- Example: STR r0, [r1,#12]



- To store to location 0x1f4 instead use: STR r0, [r1,#-12]
- To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!
- If r2 contains 3, access 0x20c by multiplying this by 4:
  - STR r0, [r1, r2, LSL #2]

Load and Store: Post-indexed Addressing

- Example: STR r0, [r1], #12



- To auto-increment the base register to location 0x1f4 instead use:
  - STR r0, [r1], #-12
- If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:
  - STR r0, [r1], r2, LSL #2

What’s Next?

- Next Lecture (August 29, Monday, 11 am – 12 pm)
  - Lecture 9

COL788: Advanced Topics in Embedded Computing

Lecture 9 – Pipelining

Vireshwar Kumar
CSE@IITD

August 29, 2022

Semester I
2022-2023

Book and Reference Material

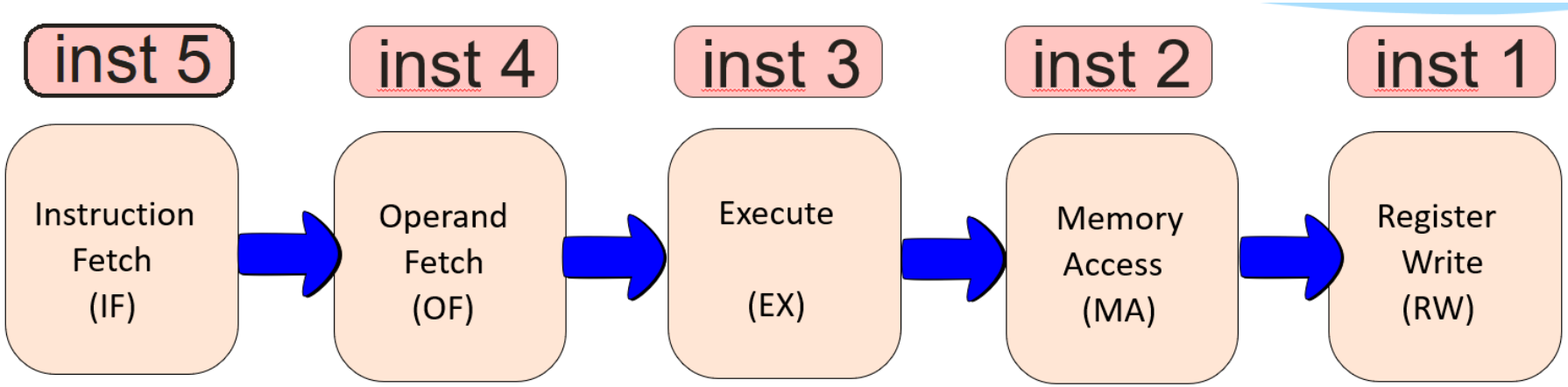
- Smruti R. Sarangi, “Basic Computer Architecture,” 1st edition, White Falcon Publishing, 2021.
  - https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

Last Lecture

- Immediate Value
- Multiplication
- Load/Store

Instruction Pipeline

- ARM uses a pipeline to increase the speed of the flow of instructions to the processor.
  - Allows several operations to be undertaken simultaneously, rather than serially.
  - The program counter (PC) points to the instruction being fetched.

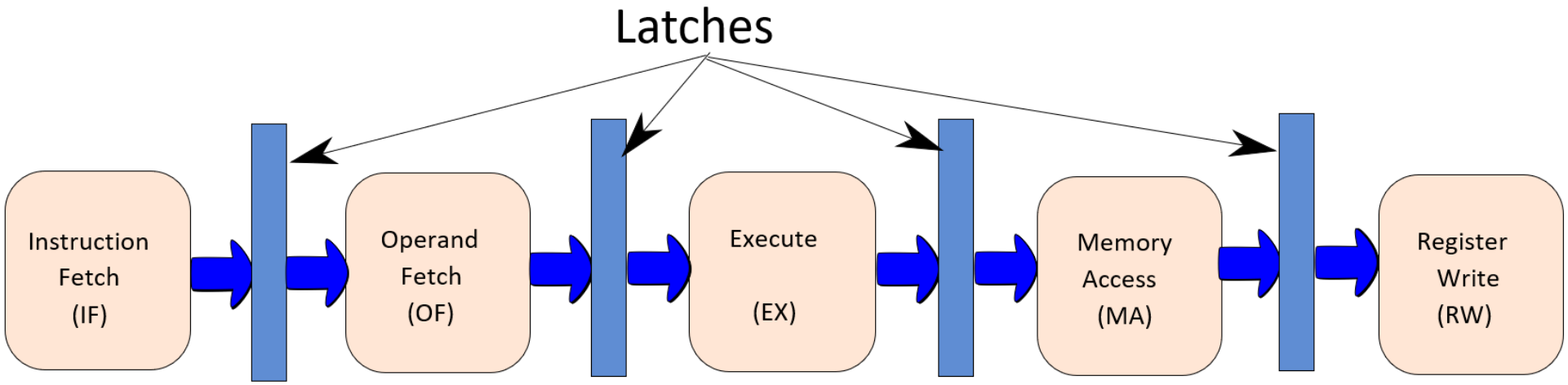


Advantage

- We keep all parts of the data path and the corresponding hardware, busy all the time

- Let us assume that all the 5 stages do the same amount of work
  - Without pipelining, every T seconds, an instruction completes its execution
  - With pipelining, every T/5 seconds, a new instruction completes its execution

Pipelined Data Path with Latches



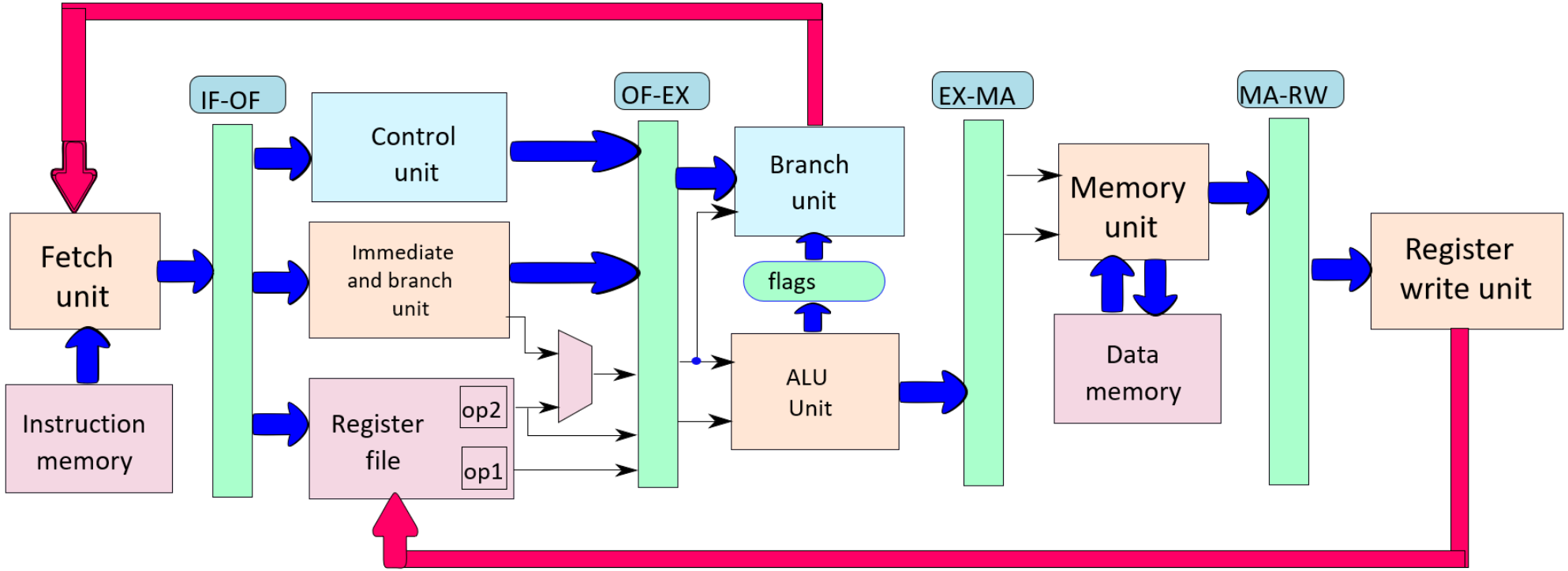
- Latches → IF-OF, OF-EX, EX-MA, and MA-RW
- At the negative edge of a clock, an instruction moves from one stage to the next

Instruction Packet

- Construction
  - Instruction contents
  - Program counter
  - All intermediate results
  - Control signals

- Implication
  - Every instruction moves with its entire state, no interference between instructions

Detailed Pipelined Datapath



What’s Next?

- Next Lecture (August 31, Wednesday, 11 am – 12 pm)
  - Lecture 10

COL788: Advanced Topics in Embedded Computing

Lecture 10 – Pipelining (Cont.)

Vireshwar Kumar
CSE@IITD

August 31, 2022

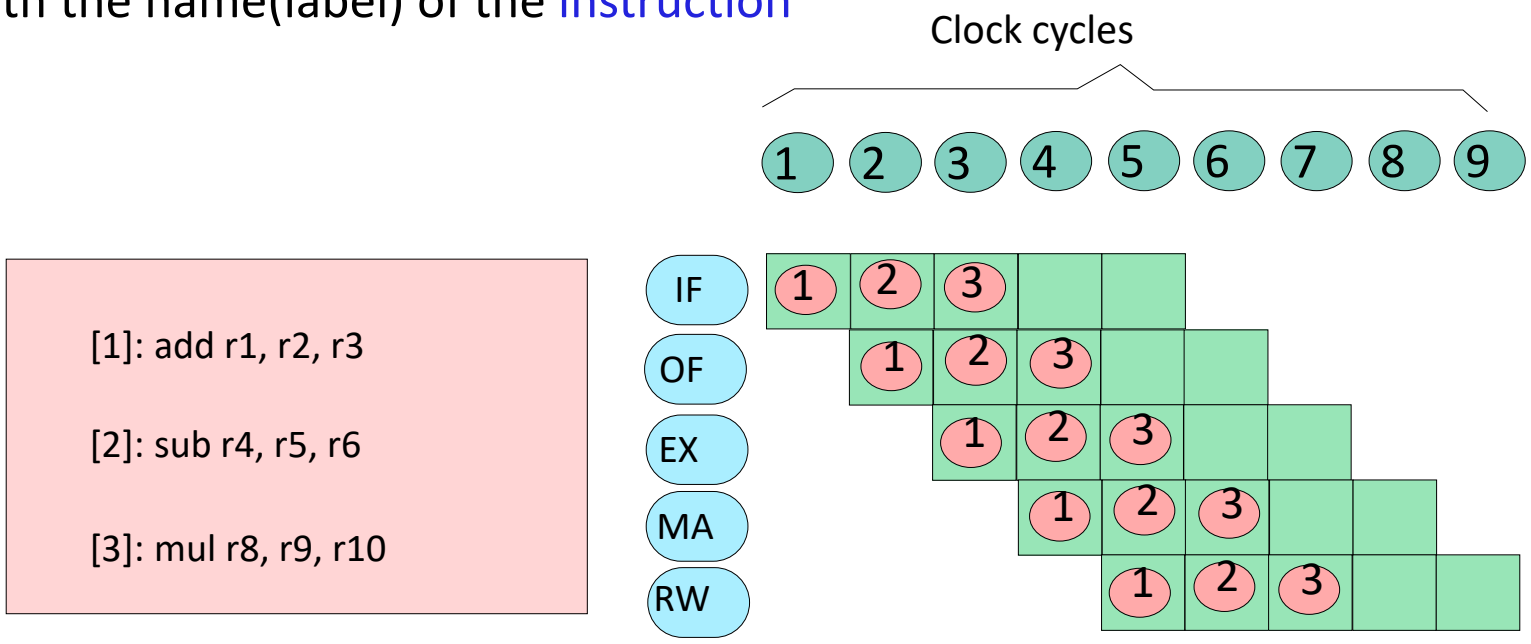
Semester I
2022-2023

Last Lecture

- Basics of Pipelining

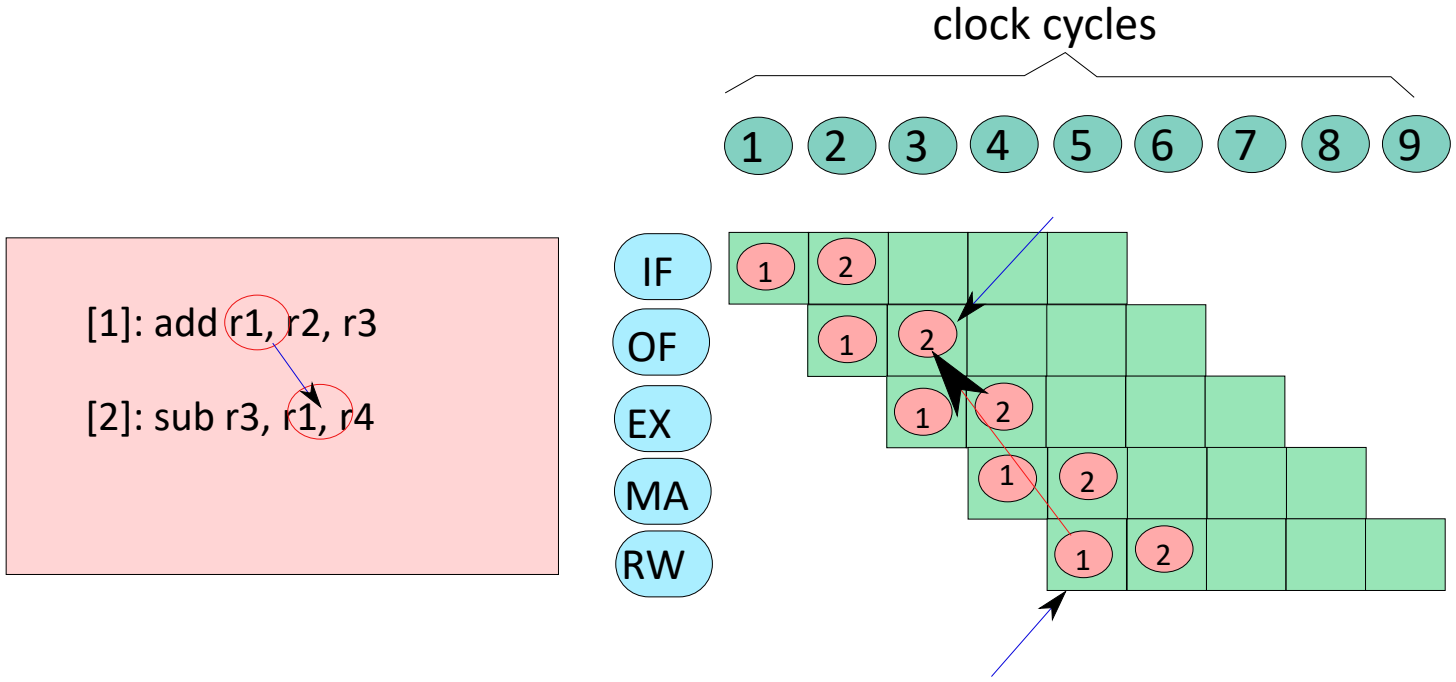
Pipeline Diagram

- Each column represents a clock cycle
- It has 5 rows: IF, OF, EX, MA, and RW
- Each cell represents the execution of an instruction in a stage
  - It is annotated with the name(label) of the instruction



Data Hazard

- RAW (read after write) hazard
- Instruction 2 will read incorrect values
  - The earliest we can dispatch instruction 2, is cycle 5



Control Hazard

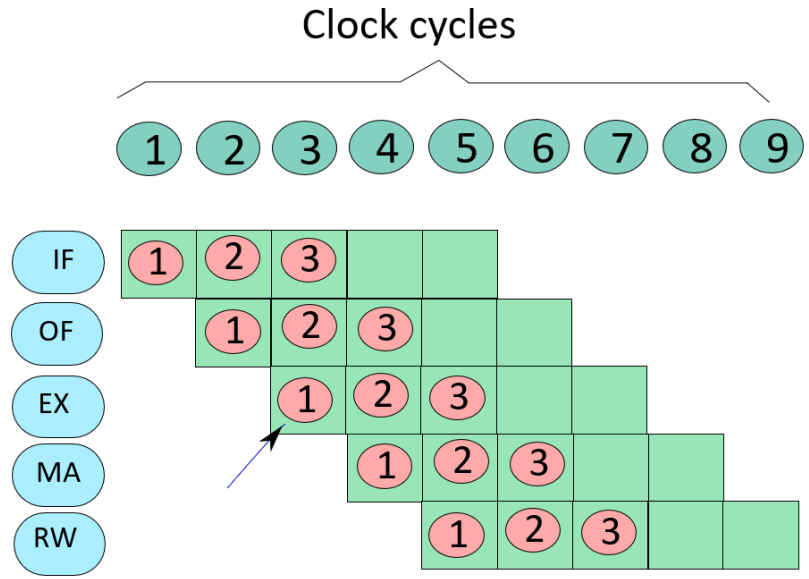
- If the branch is taken, instructions [2] and [3], might get fetched, incorrectly

```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
...
...
.foo:
[100]: add r4, r1, r2
```

```
[1]: beq .foo

[2]: mov r1, 4

[3]: add r2, r4, r3
```



What’s Next?

- Next Lecture (September 1, Thursday, 12 pm – 1 pm)
  - Lecture 11

COL788: Advanced Topics in Embedded Computing

Lecture 11 – Pipelining (Cont.)

Vireshwar Kumar
CSE@IITD

September 1, 2022

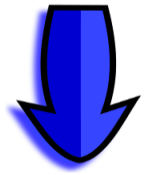
Semester I
2022-2023

Last Lectures

- Basics of Pipelining
- Data and Control Hazards

Software Solution to Data Hazard: *nop*

```
[1]: add r1, r2, r3
[2]: sub r3, r1, r4
```



```
[1]: add r1, r2, r3
[2]: nop
[3]: nop
[4]: nop
[5]: sub r3, r1, r4
```

Software Solution: Code Reordering

```
add r1, r2, r3
add r4, r1, 3
add r8, r5, r6
add r9, r8, r5
add r10, r11, r12
add r13, r10, 2
```

```
add r1, r2, r3
add r8, r5, r6
add r10, r11, r12
nop
add r4, r1, 3
add r9, r8, r5
add r13, r10, 2
```

Software Solution to Control Hazard

- Assume that the two instructions fetched after a branch are valid instructions
- These instructions are said to be in the delay slots as a delayed branch
- The compiler transfers instructions before the branchto the delay slots

```
add r1, r2, r3
add r4, r5, r6
b .foo
add r8, r9, r10
```

```
b .foo
add r1, r2, r3
add r4, r5, r6
add r8, r9, r10
```

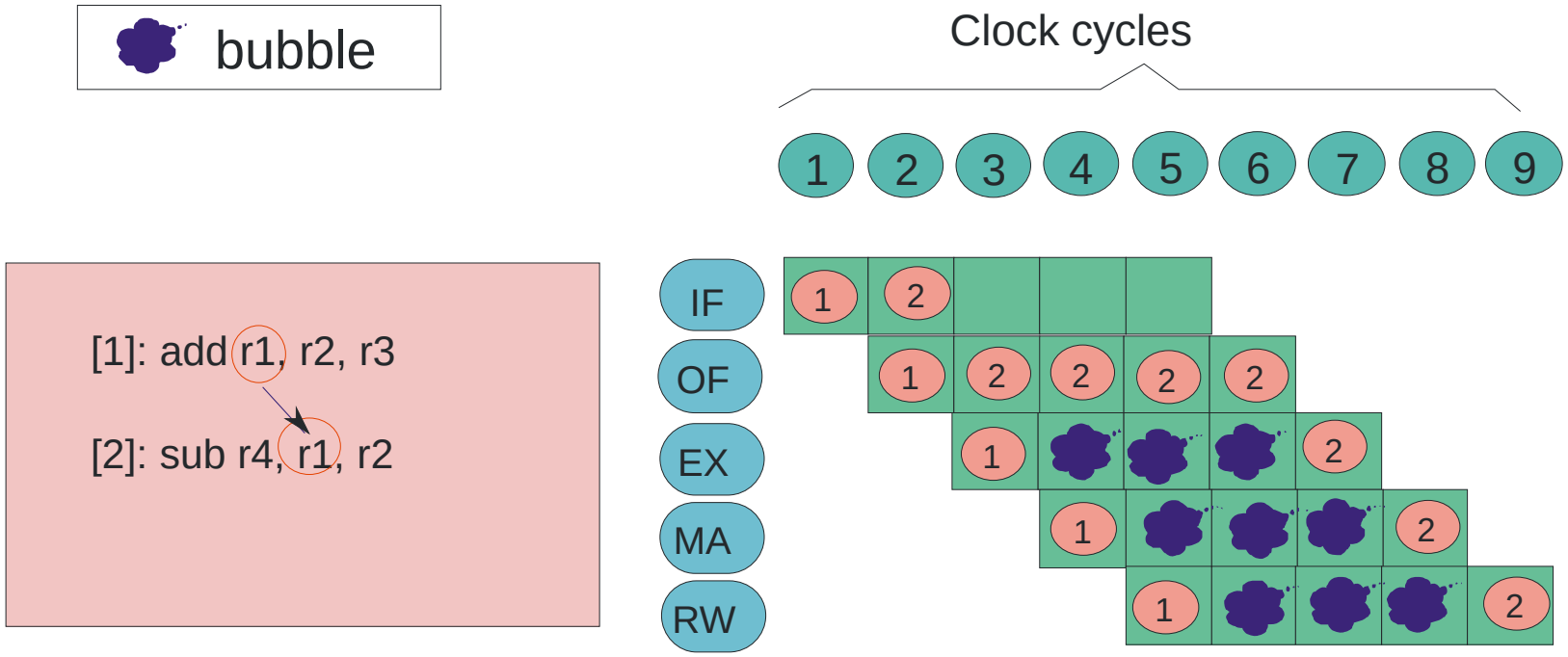
Hardware Solution: Interlock

- Hardware mechanism to enforce correctness → interlock

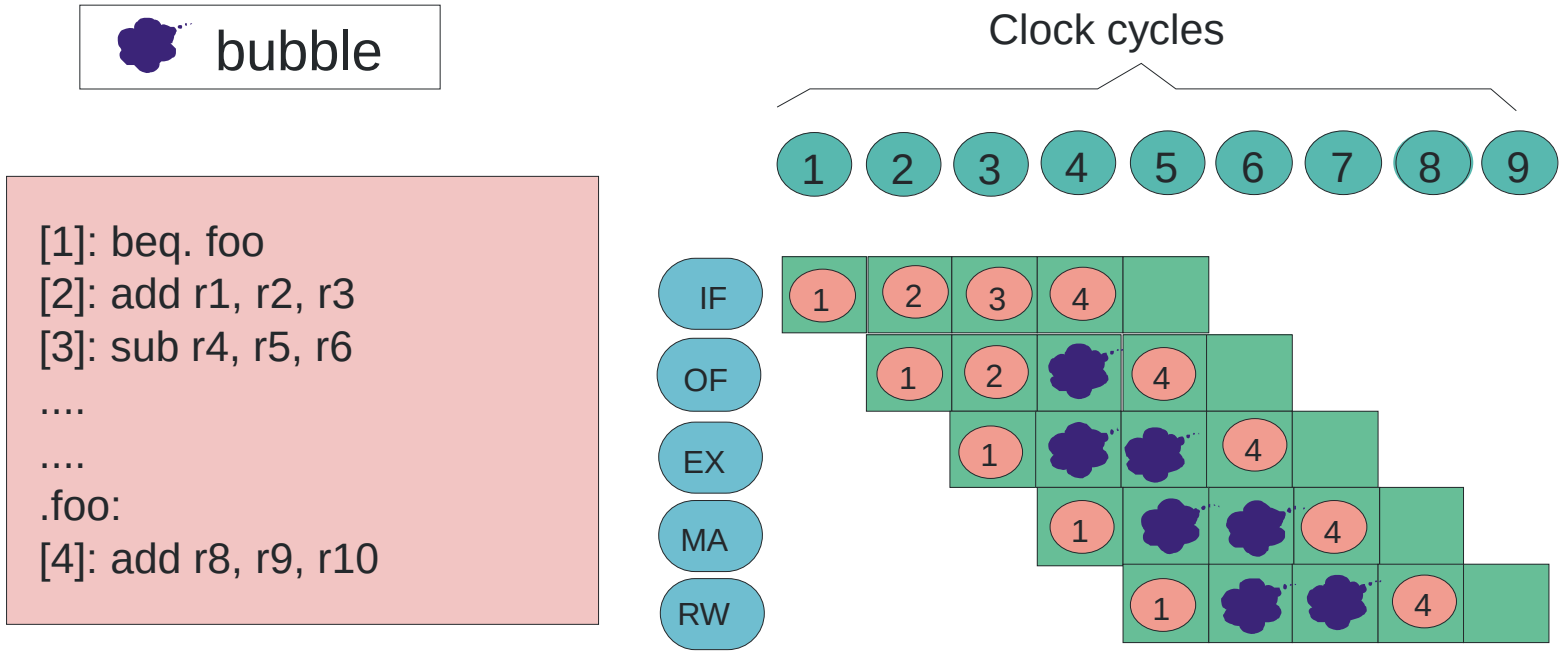
- Data-Lock
  - Do not allow a consumer instruction to move beyond the OF stage till it has read the correct values
  - Implication : Stall the IF and OF stages
- Branch-Lock
  - We never execute instructions in the wrong path

- A pipeline bubble is inserted into a stage, when the previous stage needs to be stalled

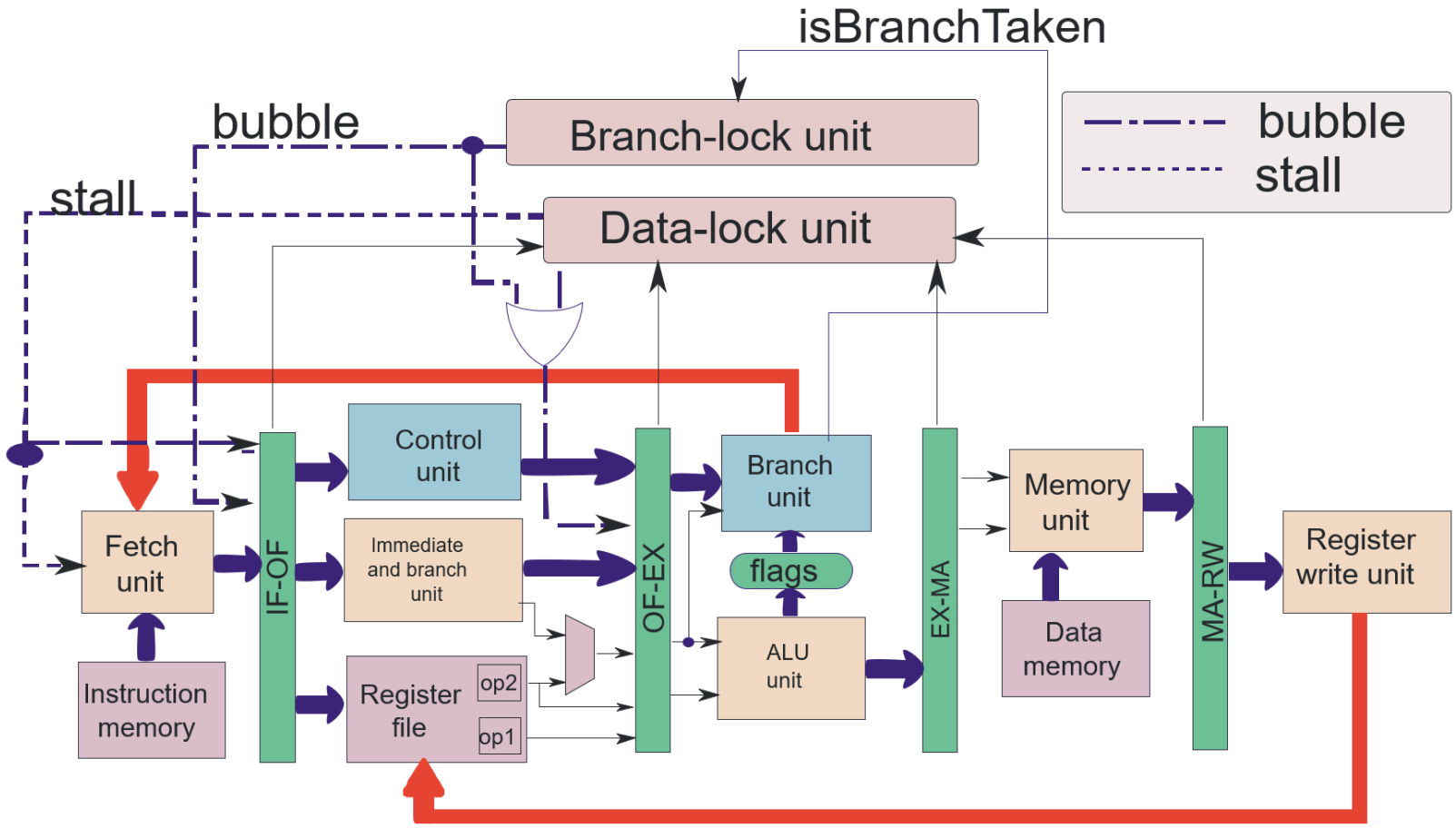
Data Hazard Mitigation



Control Hazard Mitigation



Data Path with Interlocks



Hardware vs Software Solutions

Attribute	Software	Hardware(withinterlocks)
Portability	Limited to a specific processor	Programs can be run on any processor irrespective of the nature of the pipeline
Branches	Possible to have no performance penalty, by using delay slots	Need to stall the pipeline for 2 cycles in our design
RAW hazards	Possible to eliminate them through code scheduling	Need to stall the pipeline
Performance	Highly dependent on the nature of the program	The basic version of a pipeline with interlocks is expected to be slower than the version that relies on software

What’s Next?

- Next Lecture (September 5, Monday, 11 am – 12 pm)
  - Lecture 12

COL788: Advanced Topics in Embedded Computing

Lecture 12 – Pipelining (Cont.)

Vireshwar Kumar
CSE@IITD

September 5, 2022

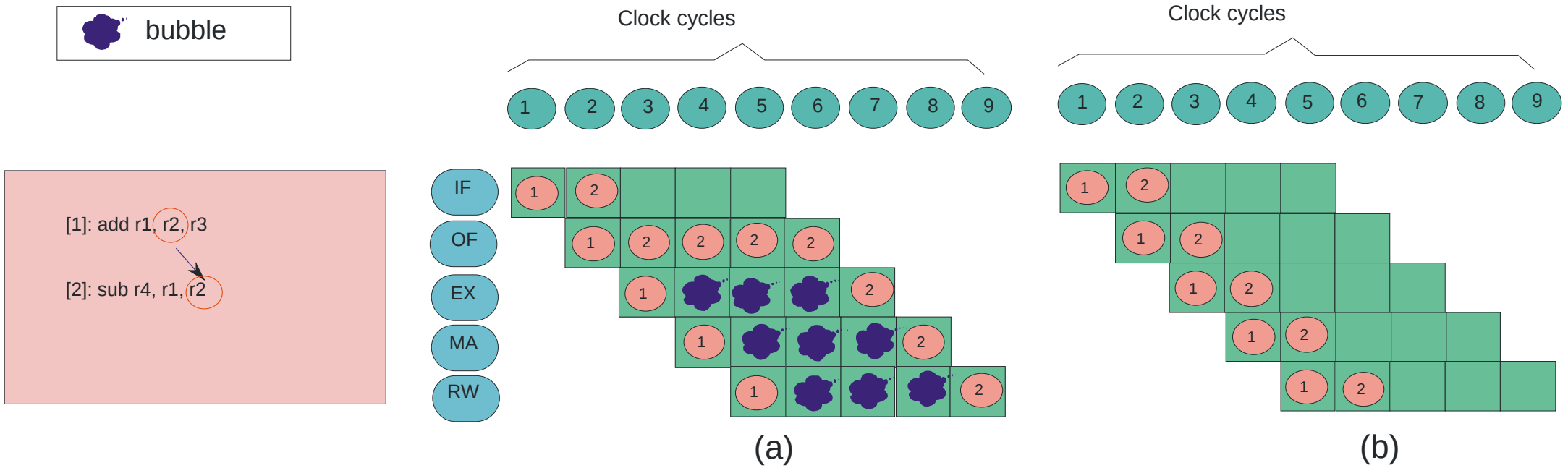
Semester I
2022-2023

Last Lectures

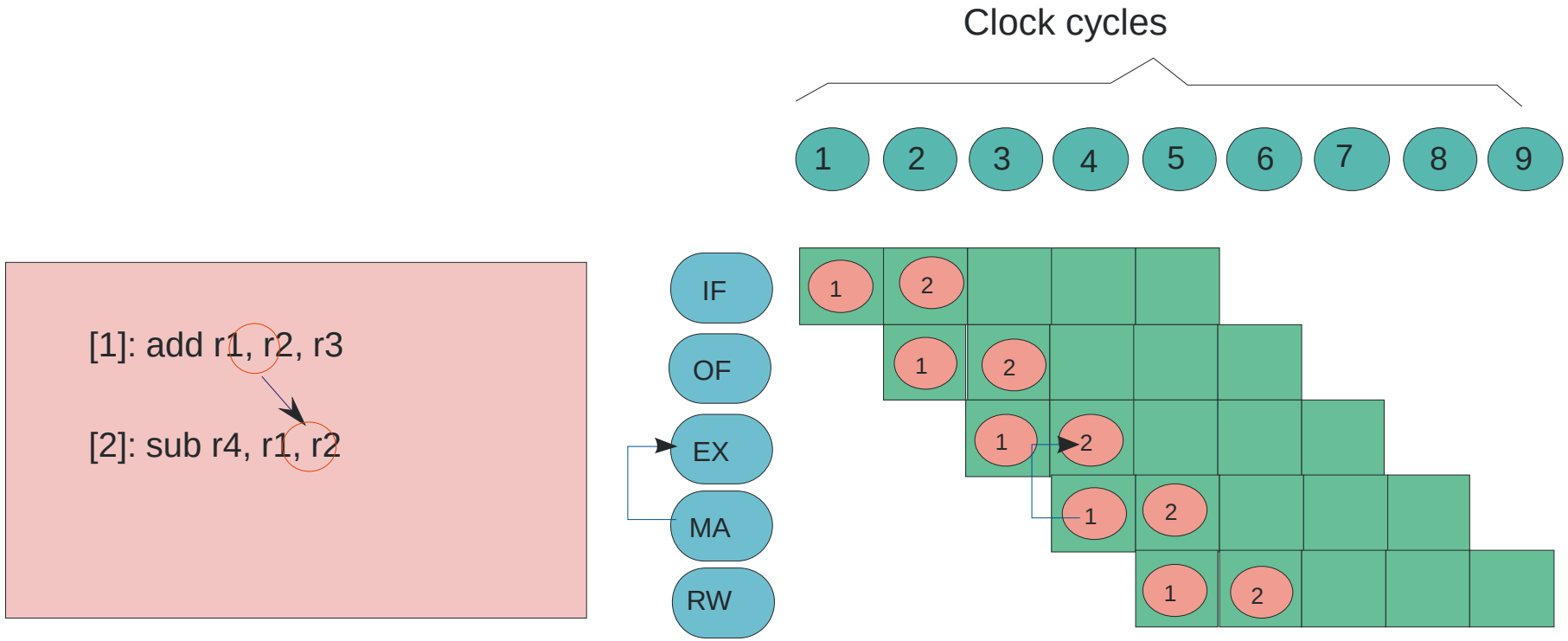
- Basics of Pipelining
- Data and Control Hazards
- Software and Hardware Solutions

Relook at the Pipeline Diagram

- When does instruction 2 need the value of r1: Cycle 4, EX Stage
- When does instruction 1 produce the value of r1: End of Cycle 3, EX Stage



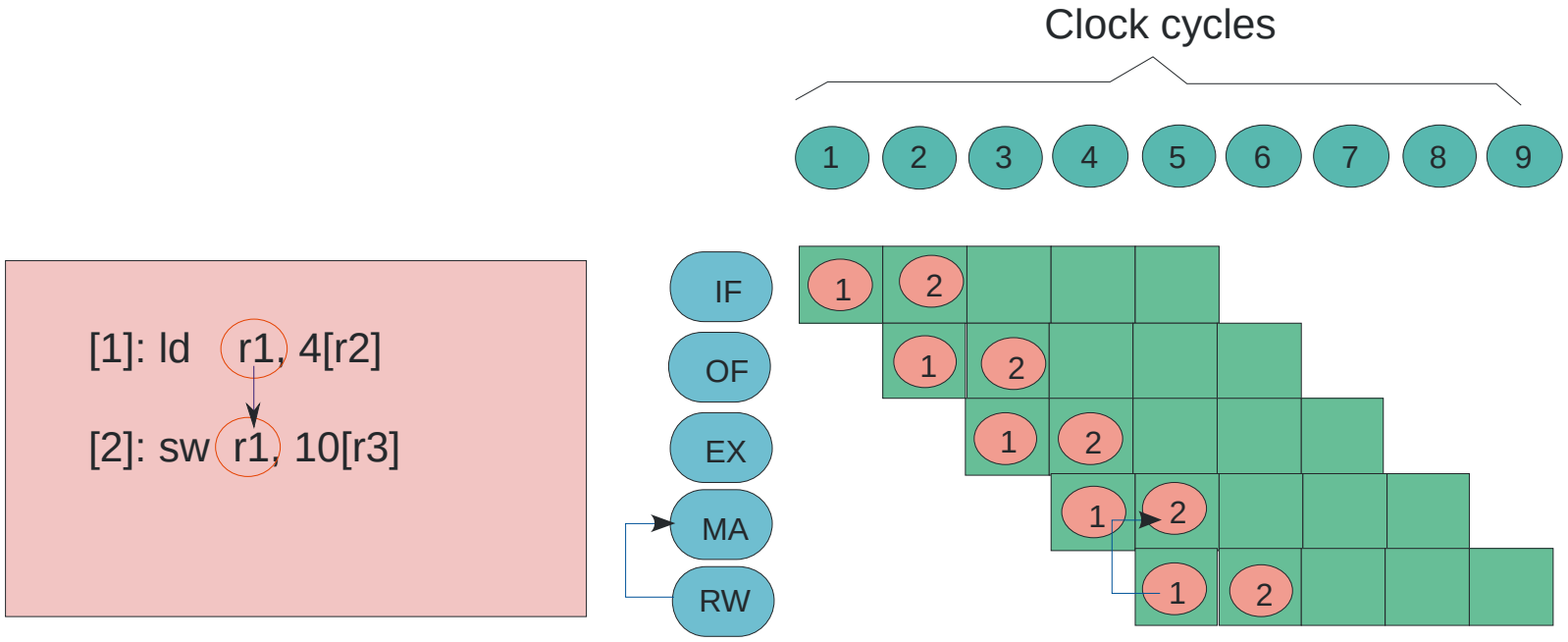
Forwarding


Clock cycles
1
2
3
4
5
6
7
8
9
[1]: add r1, r2, r3
[2]: sub r4, r1, r2
IF
OF
EX
MA
RW

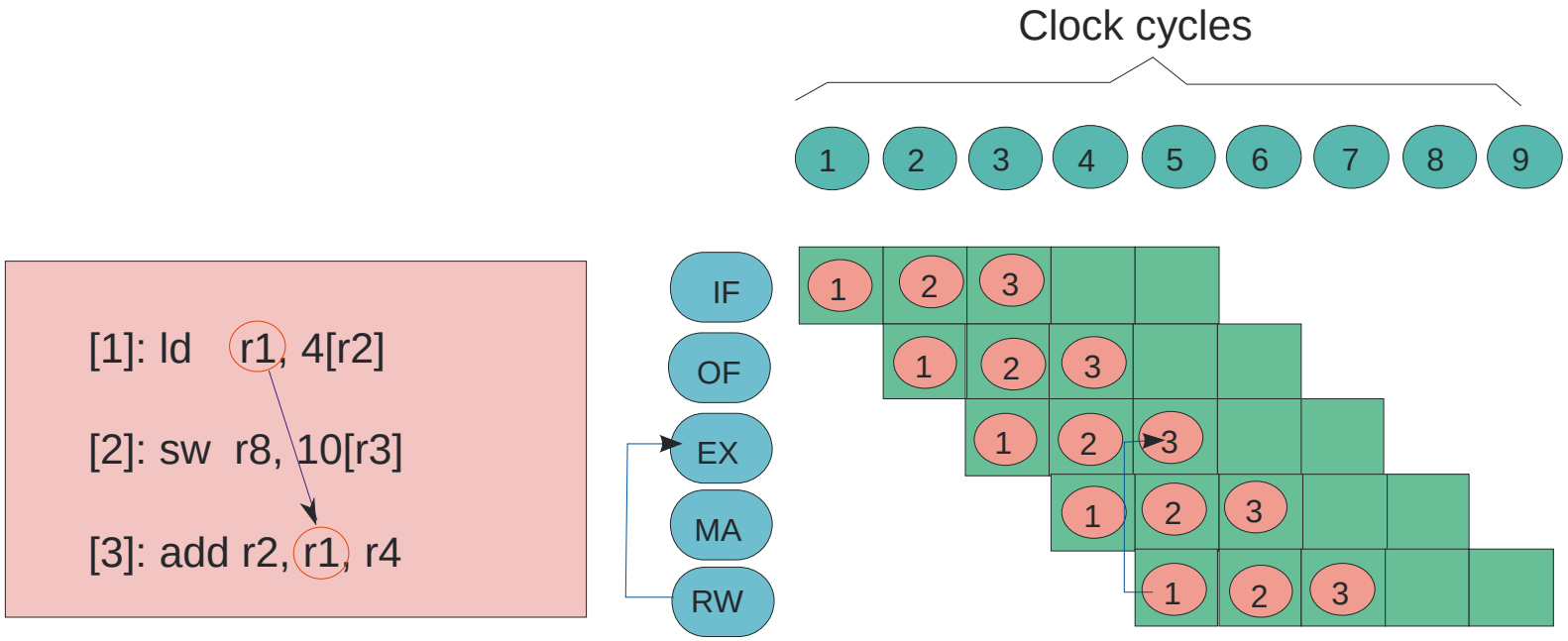
Forwarding Path

- 3 Stage Paths
  - RW → OF
- 2 Stage Paths
  - RW → EX
- 1 Stage Paths
  - RW → MA (load to store)
  - MA → EX (ALU Instructions, load, store)

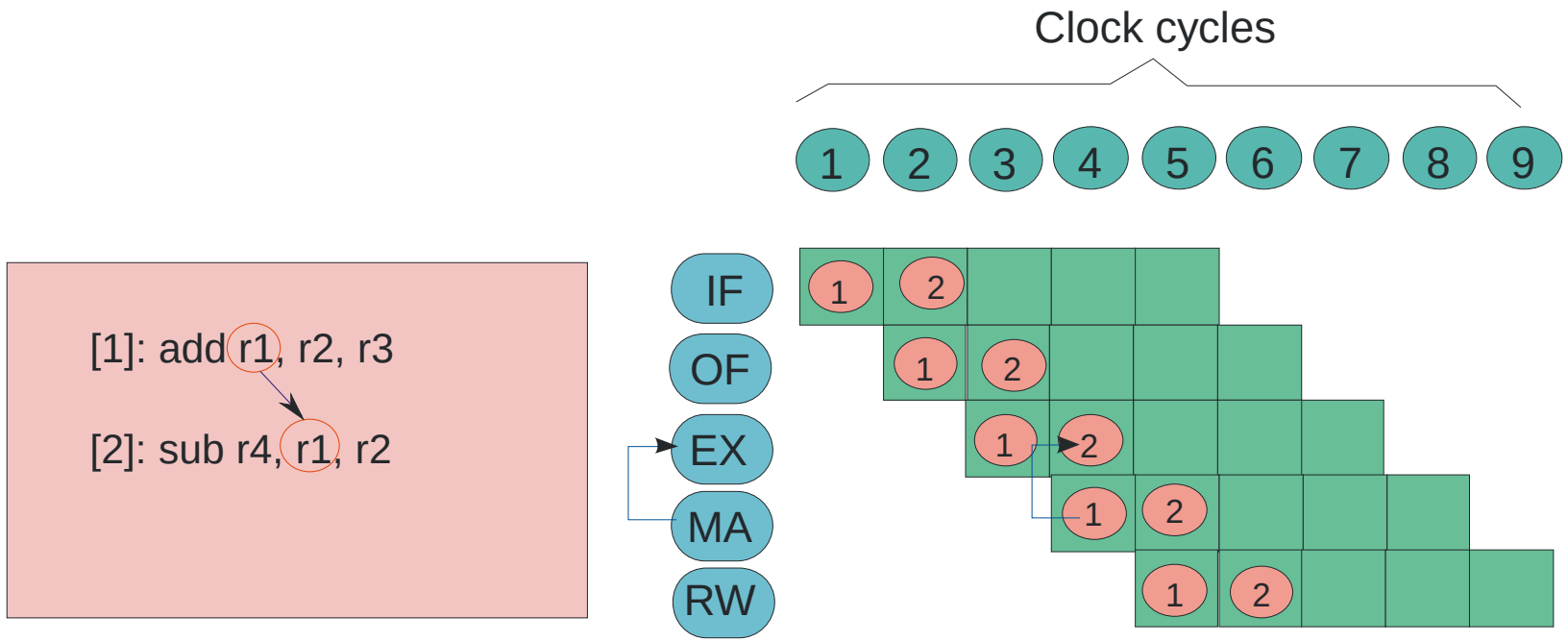
Forwarding Path: RW → MA



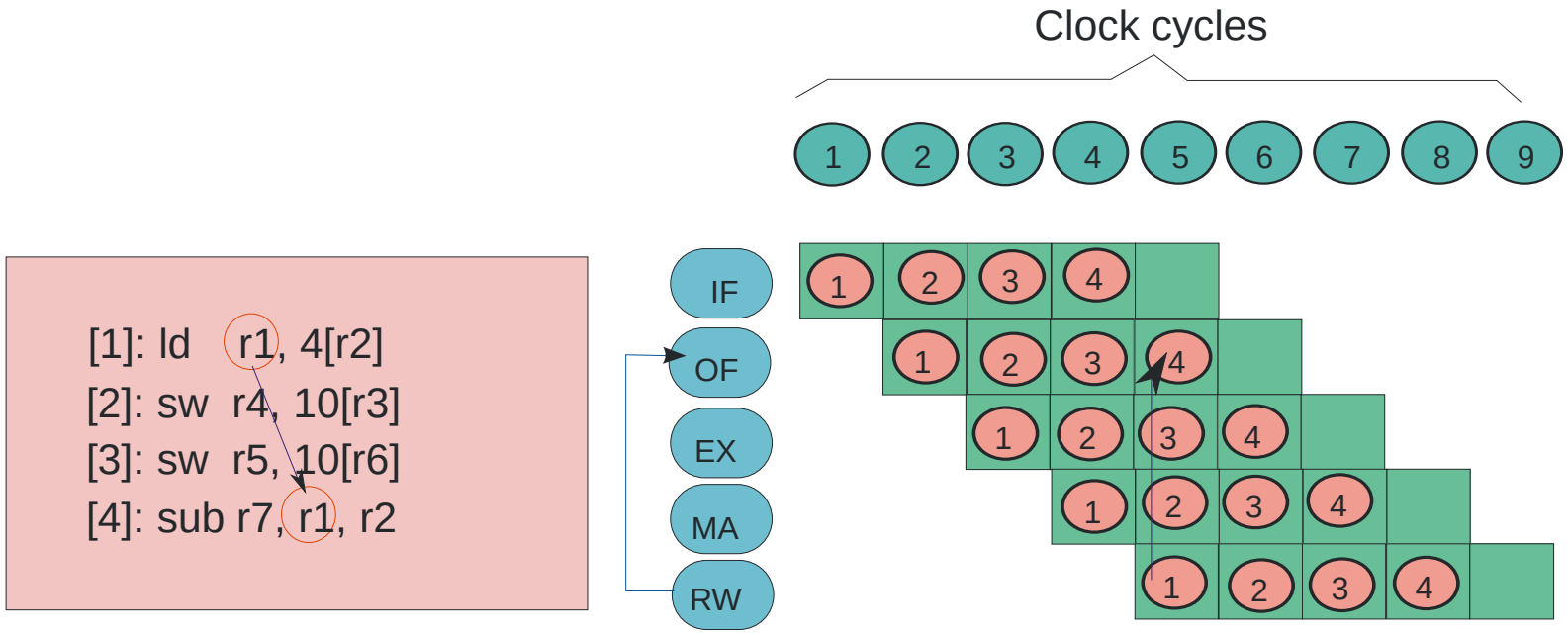
Forwarding Path: RW → EX



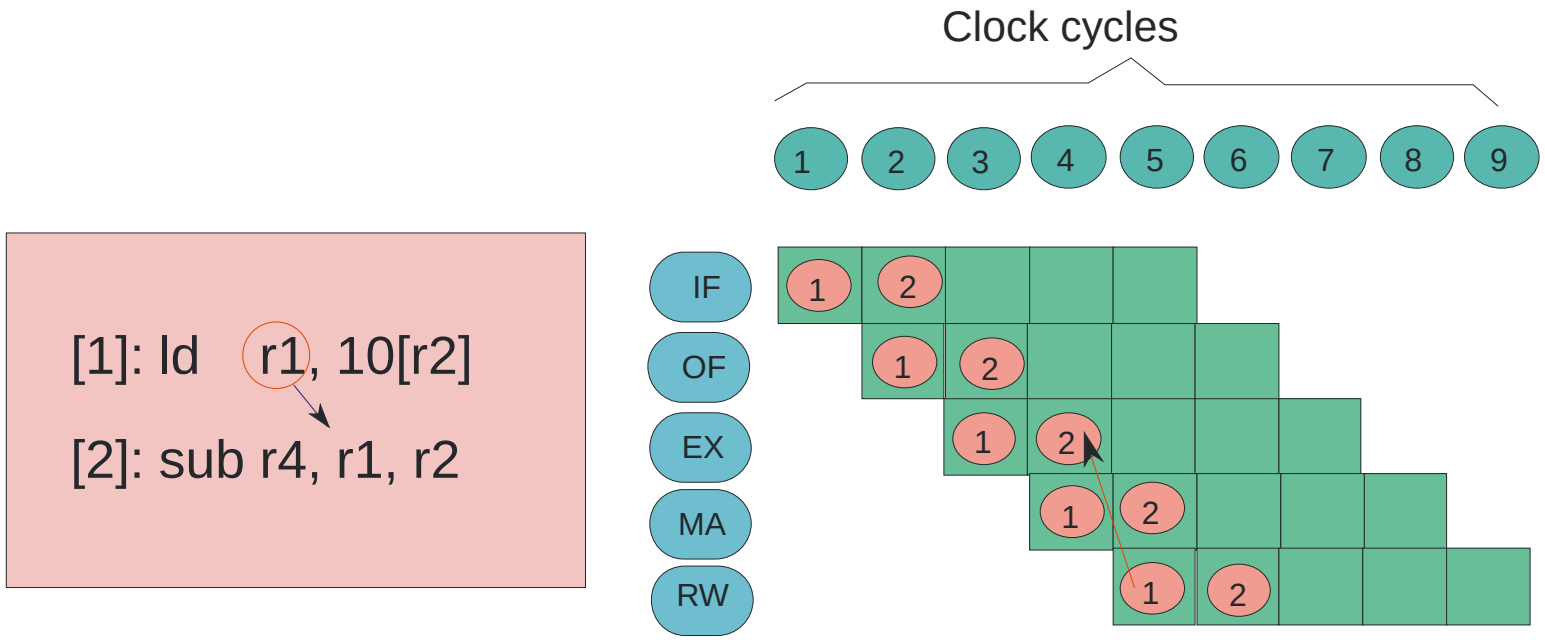
Forwarding Path : MA → EX



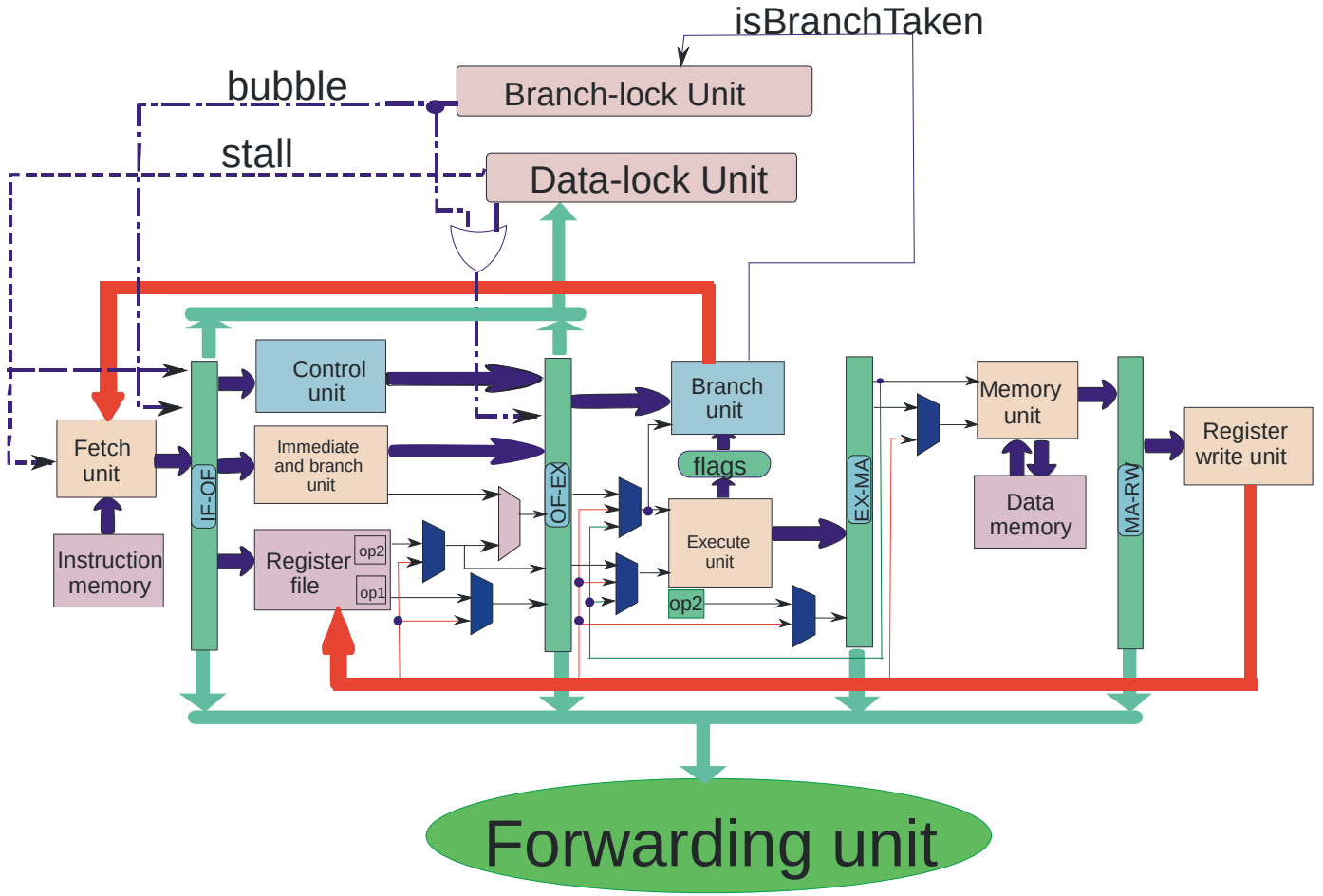
Forwarding Path : RW → OF



Load-Use Hazard



Complete Data Path



What’s Next?

- Next Lecture (September 7, Wednesday, 11 am – 12 pm)
  - Lecture 13

COL788: Advanced Topics in Embedded Computing

Lecture 13 – Memory System

Vireshwar Kumar
CSE@IITD

September 07, 2022

Semester I
2022-2023

Memory

- Many programs run at the same time
- The CPU of course runs one program at a time
  - Switches between programs periodically
  - Program states are stored in the memory

Memory Types

- Tradeoffs: Area, Power, and Latency
  - Increase Area → Reduce latency, increase power
  - Reduce latency → increase area, increase power
  - Reduce power → reduce area, increase latency

Cell Type	Area	Typical Latency
Master Slave D flip flop	0.8 μm^2	Fraction of a cycle
SRAM cell in an array	0.08 μm^2	1-5 cycles
DRAM cell in an array	0.005 μm^2	50-200 cycles

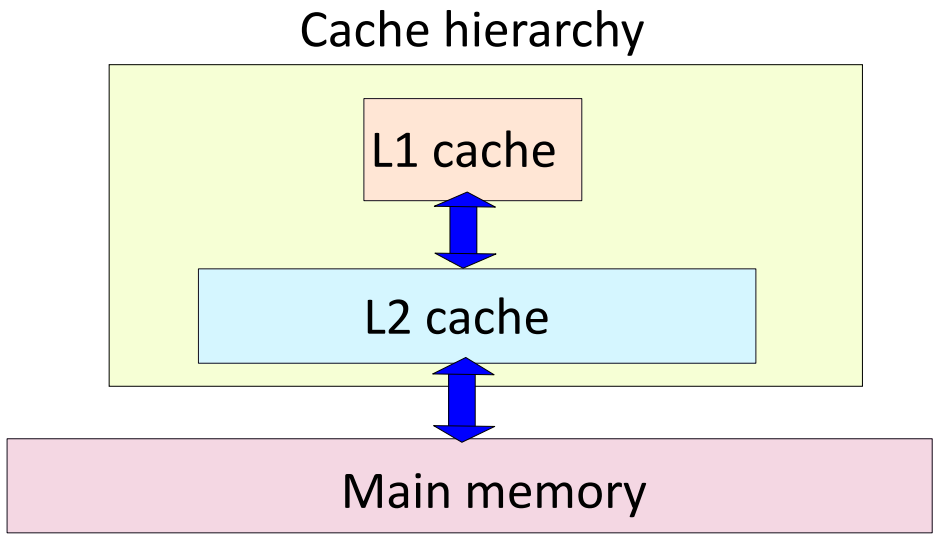
Temporal and Spatial Locality

- Temporal Locality
  - If a resource is accessed at some point of time, then most likely it will be accessed again in a short period of time.

- Spatial Locality
  - If a resource is accessed at some point of time, then most likely similar resources will be accessed again in the near future.

Exploiting Temporal Locality

- The L1 cache is a small memory (8-64 KB) composed of SRAM cells
- The L2 cache is larger and slower (128 KB – 4 MB) (SRAM cells)
- The main memory is even larger (1 – 64 GB) (DRAM cells)



Cache Hit and Miss

- Access Protocol
  - First access the L1 cache. If the memory location is present, we have a cache hit.
    - Perform the access (read/write)
  - Otherwise, we have a cache miss.
    - Fetch the value from the lower levels of the memory system, and populate the cache.
    - Follow this protocol recursively

- Typical Hit Rates, Latencies
  - L1 : 95 %, 1 cycle
  - L2 : 60 %, 10 cycles
  - Main Memory : 100 %, 300 cycles

Exploiting Spatial Locality

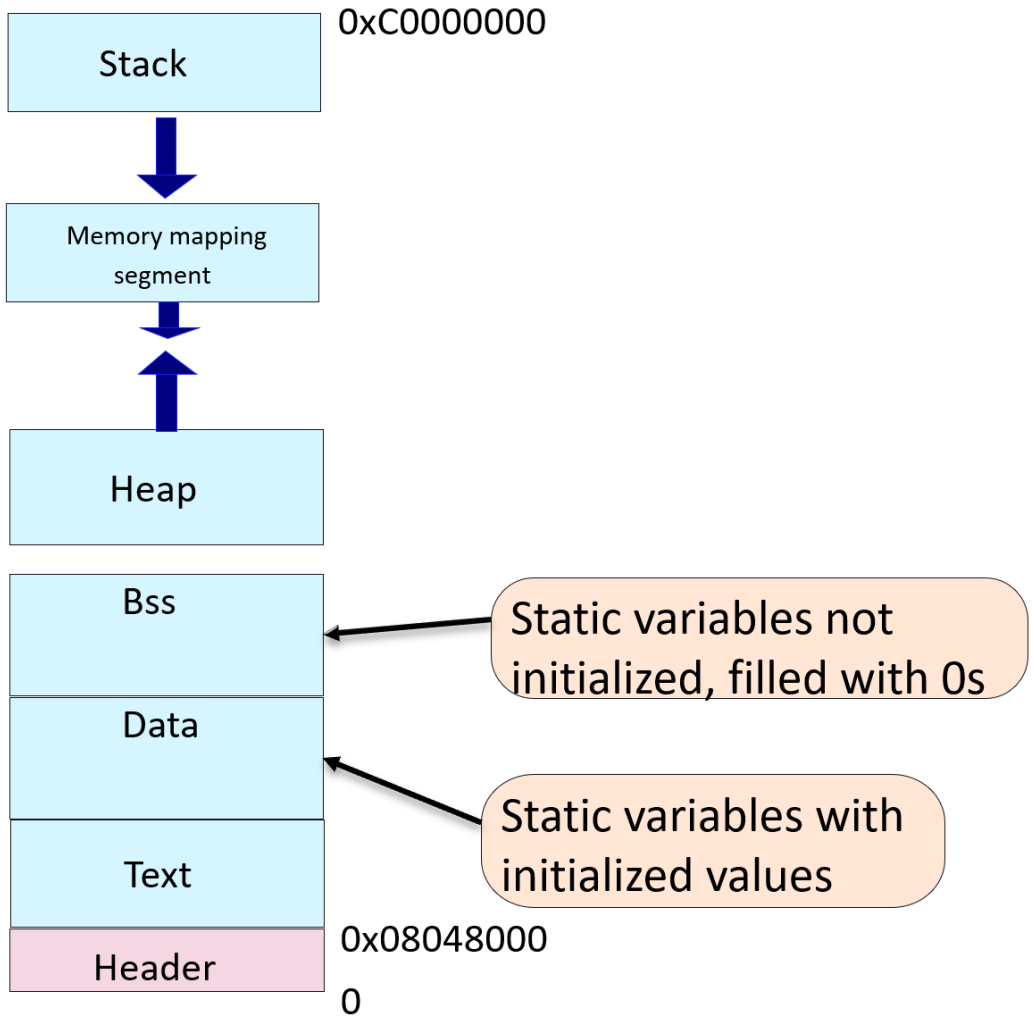
- Group memory addresses into sets of n bytes
- Each group is known as a cache line or cache block
- A cache block is typically 32, 64, or 128 bytes

Physical and Virtual Memory

- Physical Memory
  - Refers to the actual set of physical memory locations contained in the main memory, and the caches.

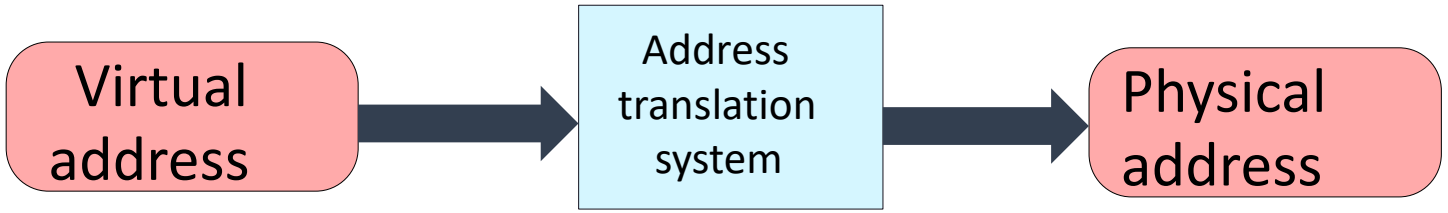
- Virtual Memory
  - The memory space assumed by a program.
  - Contiguous, without limits.

Virtual Memory Map Example

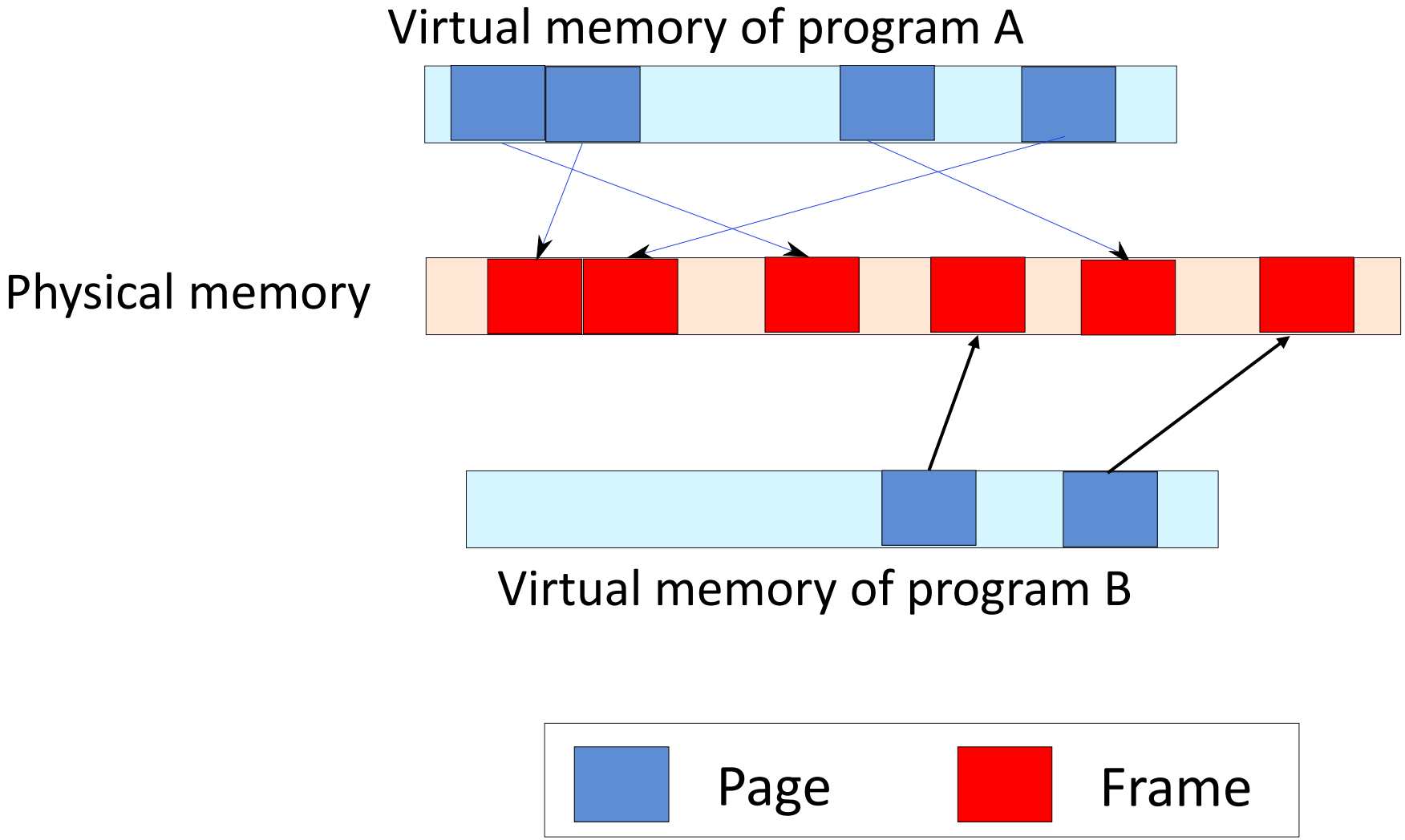


Address Translation

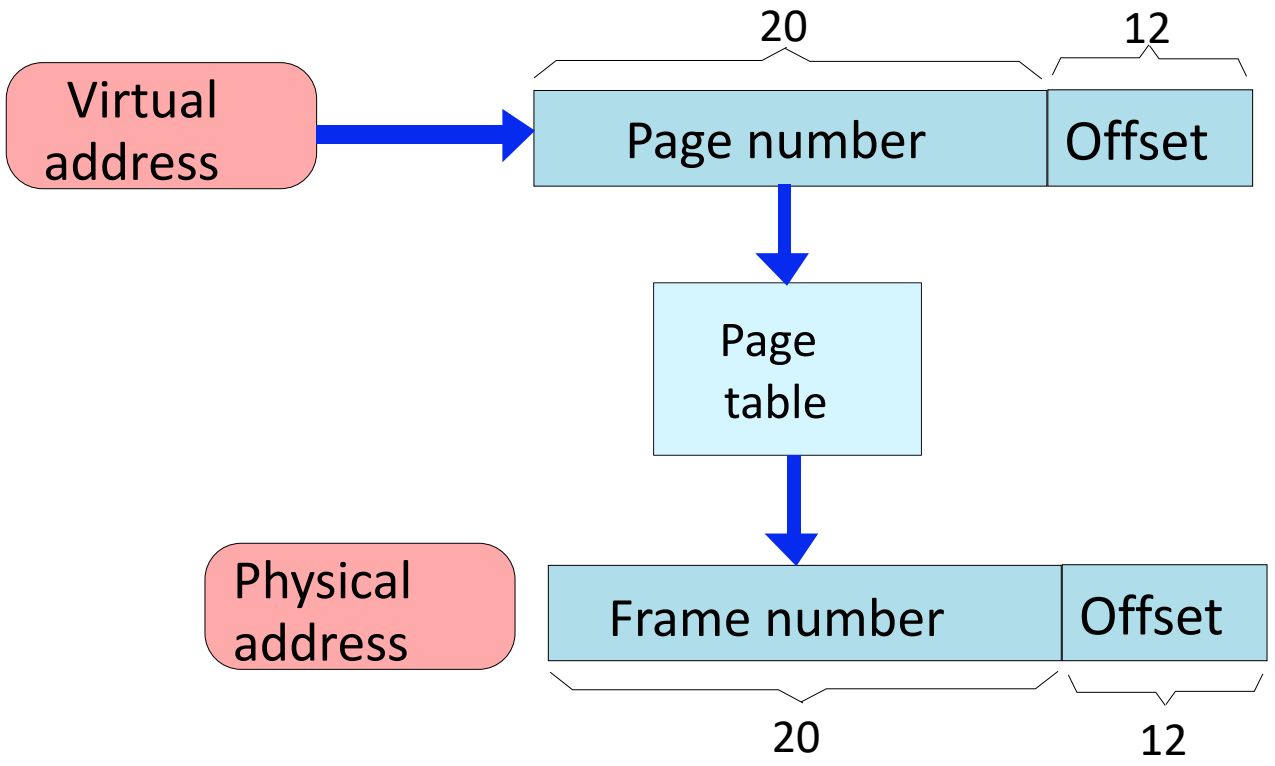
- Divide the virtual address space into chunks of 4 kB → page
- Divide the physical address space into chunks of 4 kB → frame
- Map pages to frames



Mapping Example

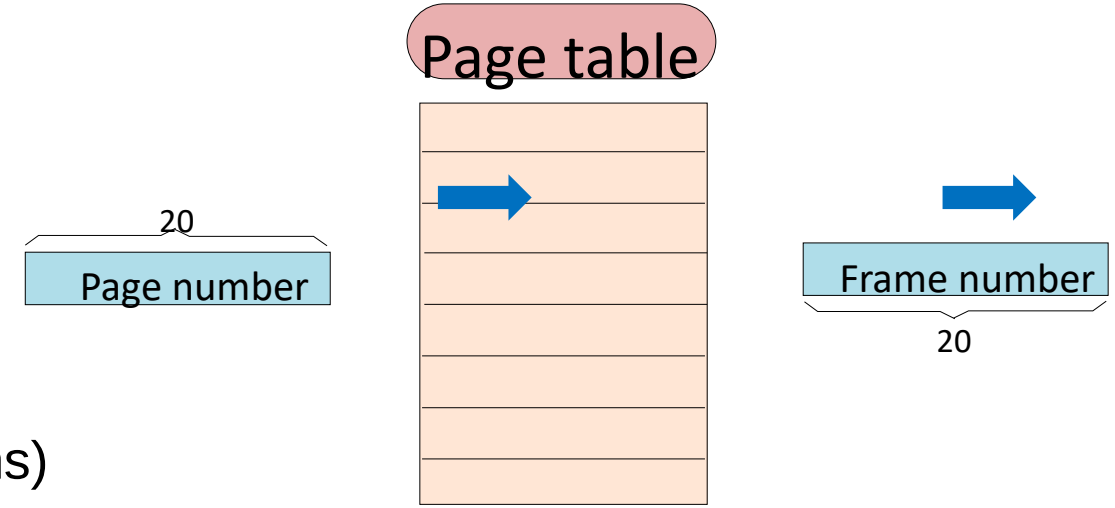


Page Translation Example



Single Level Page Table

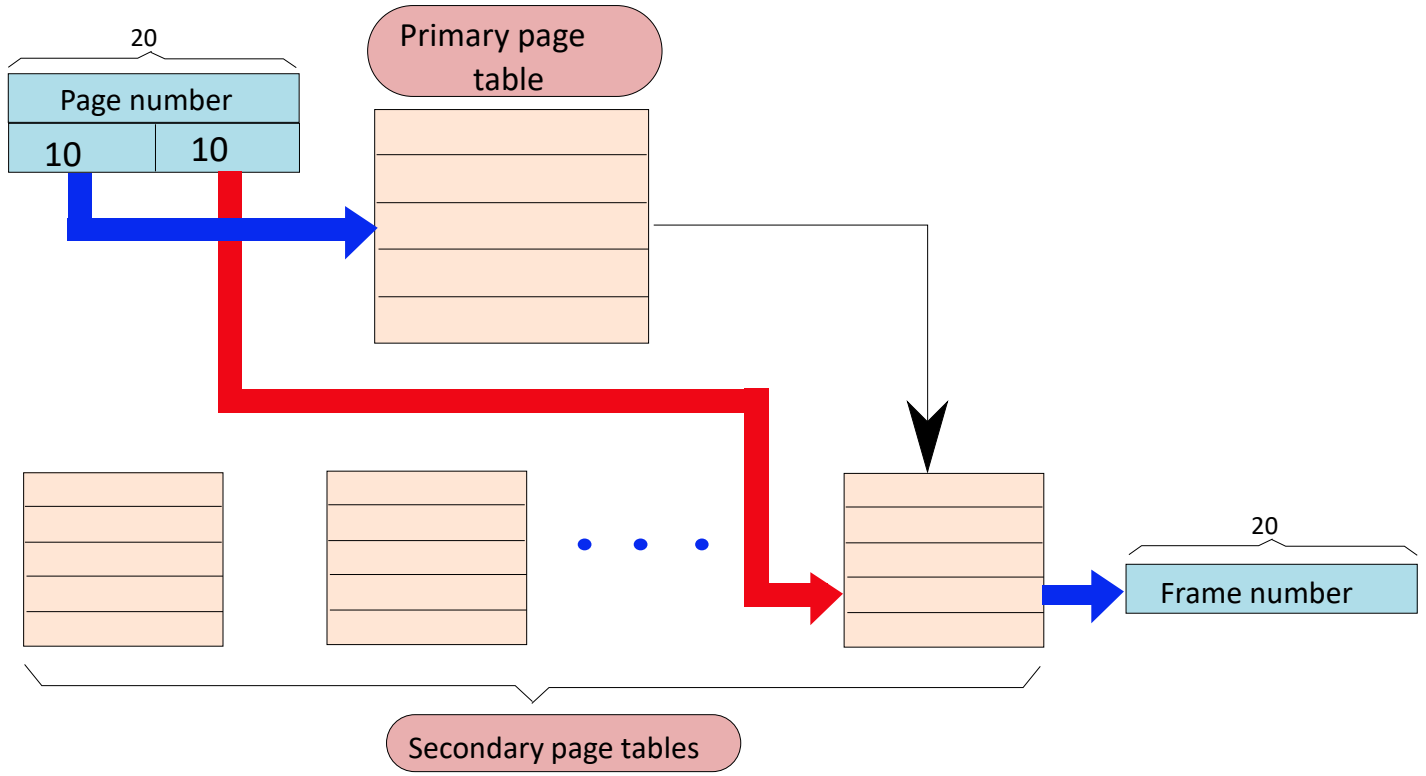
- Size of the single level page table
  - Size of an entry (20 bits = 2.5 bytes) *
  - Number of entries (220 = 1 million)
  - Total → 2.5 MB



- For 200 processes (running instances of programs)
  - We spend 500 MB in saving page tables (not acceptable)

- Implication
  - Most of the virtual address space is empty
  - Most programs do not require that much of memory

Two Level Page Table

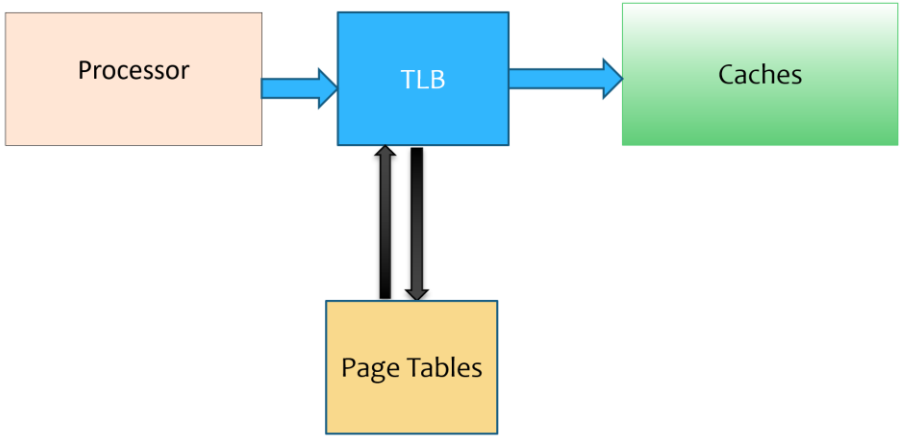


Advantages of Two Level Page Table

- We have a two level set of page tables
  - Primary and secondary page tables
- Not all the entries of the primary page table point to valid secondary page tables
- Each secondary page table → 1024 * 2.5 B = 2.5 KB
  - Maps 4MB of virtual memory
- Implications
  - Allocate only those many secondary page tables as required.
  - We do not need many secondary page tables due to spatial locality in programs
  - Example: If a program uses 100 MB of virtual memory and needs 25 secondary page tables, we need a total of 2.5KB * 25 = 62.5 KB of space for saving secondary page tables (minimal).

Translation Lookaside Buffer

- TLB (Translation Lookaside Buffer)
  - A fully associative cache
  - Each entry contains a page → frame (mapping)
  - Typically contains 64 entries
  - Very few accesses go to the page table



- Accesses that go to the page table
  - If there is no mapping, we have a page fault
  - On a page fault, create a mapping, and allocate an empty frame in memory. Update the list of empty frames.

What’s Next?

- Lecture 14
  - September 08, Thursday, 12 pm – 1 pm

COL788: Advanced Topics in Embedded Computing

Lecture 14 – Boot Sequence

Vireshwar Kumar
CSE@IITD

September 08, 2022

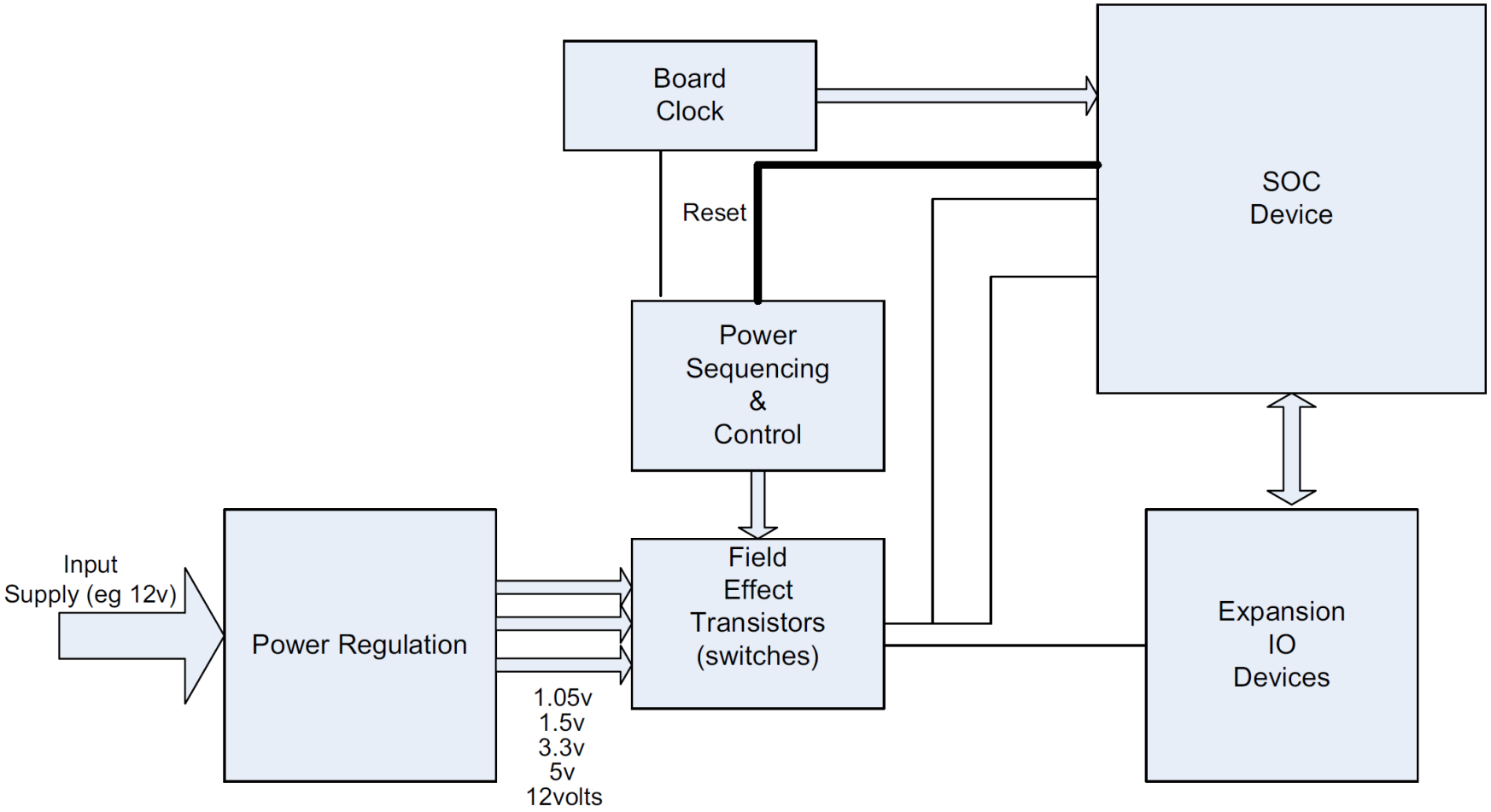
Semester I
2022-2023

Outline

- Previously
  - System Architecture
  - Processor Architecture
  - Memory

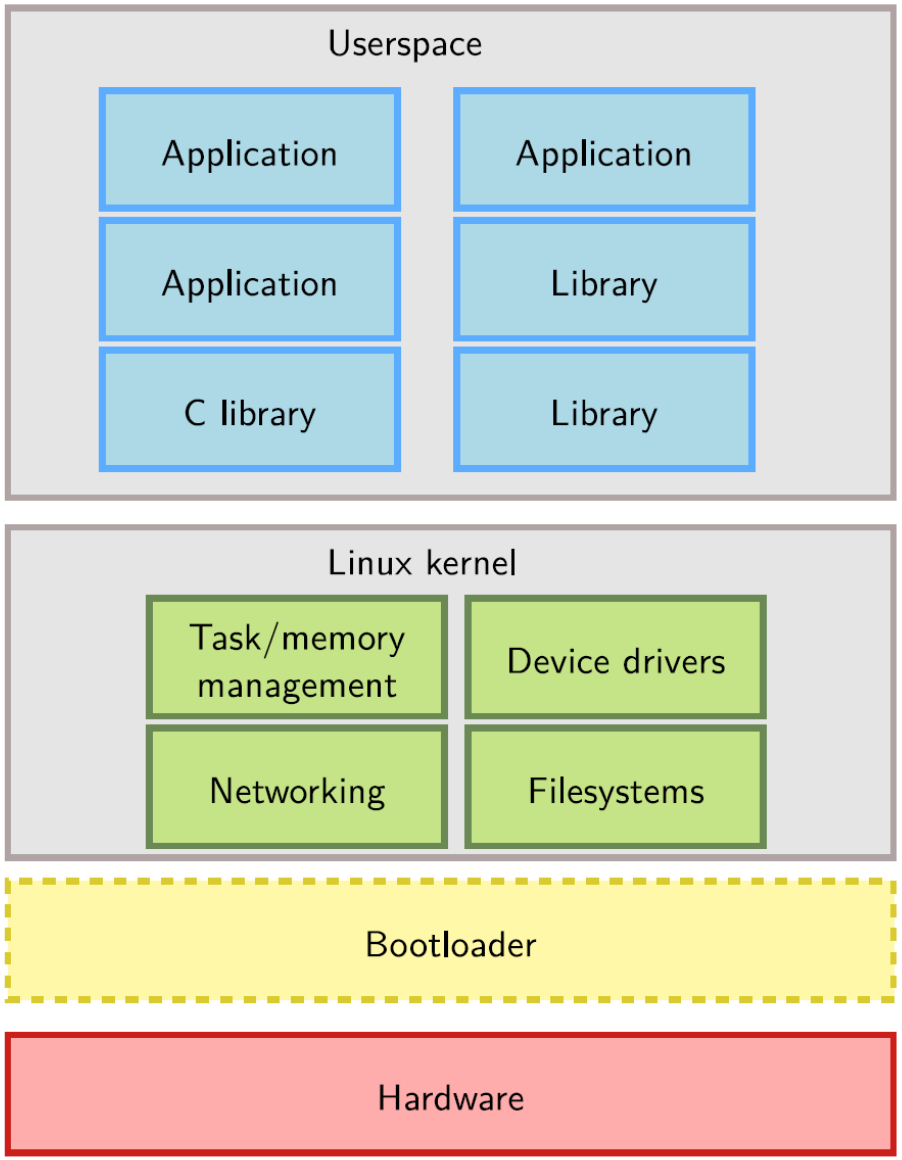
- Next
  - Boot Sequence
  - Operating System
  - Embedded Linux

Power Sequencing



Boot Sequence

- System Startup
  - Bootloader
  - Kernel
  - Init Process



Bootloader

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application

U (Universal)-Boot Bootloader

- U-Boot is a typical free software project
  - License: GPLv2 (same as Linux)
  - Freely available at https://www.denx.de/wiki/U-Boot

- Development and discussions happen around an open mailing-list

- Follows a regular release schedule. Every 2 or 3 months, a new version is released.

What’s Next?

- Lecture 15
  - September 12, Monday, 11 am – 12 pm

COL788: Advanced Topics in Embedded Computing

Lecture 15 – Embedded OS

Vireshwar Kumar
CSE@IITD

September 14, 2022

Semester I
2022-2023

Outline

- Previously
  - System Architecture
  - Processor Architecture
  - Memory

- Next
  - Boot Sequence
  - **Operating System**
  - Embedded Linux

Contiki OS

- Features
  - Memory Allocation
  - Full IP Networking
  - Power Awareness
  - 6Lowpan, RPL, CoAP
  - Dynamic Module Loading
  - Protothreads
  - File System
  - Contiki Shell

CooCox CoOS

- Features
  - Free and open real-time Operating System
  - Specially designed for Cortex-M series
  - Scalable, minimum system kernel is only 974Bytes
  - Adaptive Task Scheduling Algorithm.
  - Supports preemptive priority and round-robin
  - Interrupt latency is next to 0
  - Stack overflow detection option
  - Semaphore, Mutex, Flag, Mailbox and Queue for communication & synchronisation
  - Supports the platforms of ICCARM, ARMCC, GCC

TinyOS

- Overview
  - TinyOS is an "operating system" designed for low-power wireless embedded systems. Fundamentally, it is a work scheduler and a collection of drivers for microcontrollers and other ICs commonly used in wireless embedded platforms.

FreeRTOS

- What is FreeRTOS?
  - FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller
- What is an RTOS?
  - A part of the operating system called the scheduler is responsible for deciding which program to run when and provides the illusion of simultaneous execution by rapidly switching between each program.
  - The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as *deterministic*) execution pattern.

Embedded Operating System

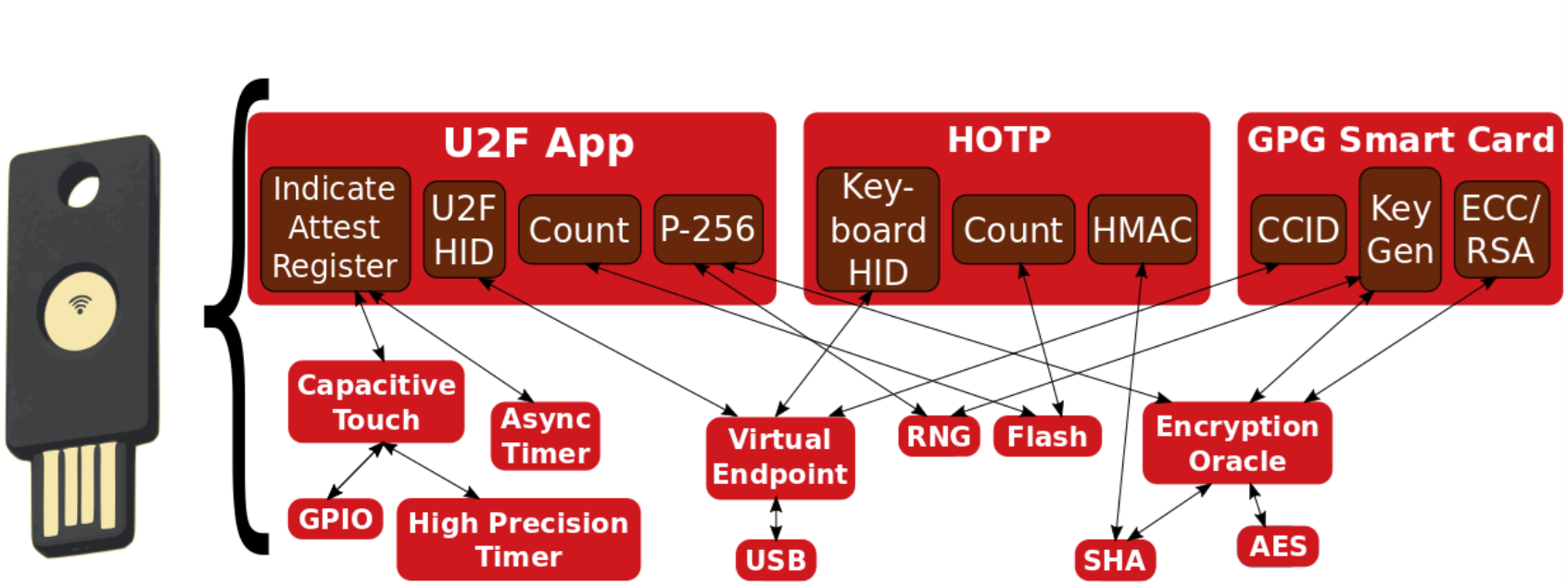
- Some subset of operating system principles, dictated by application
  - Hardware abstraction
    - Block storage versus file system?
    - But not architecture? [What’s a runtime?]
  - Resource management
    - Process isolation
    - Job control

Example: USB Authentication Key

Multiple independent applications

No programmability in favor of security

Result: Handful of programmers control software stack

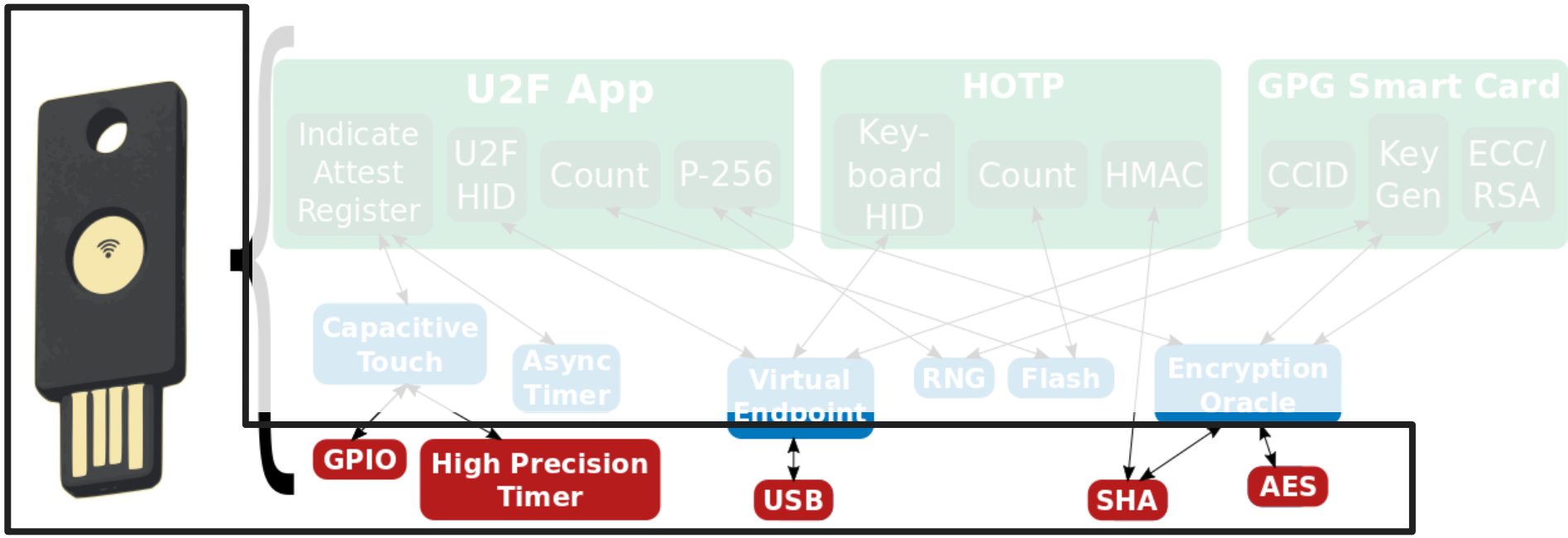


Platform

Build the hardware

Responsible for TCB: core kernel, MCU-specific code

Trusted: complete control over firmware & hardware



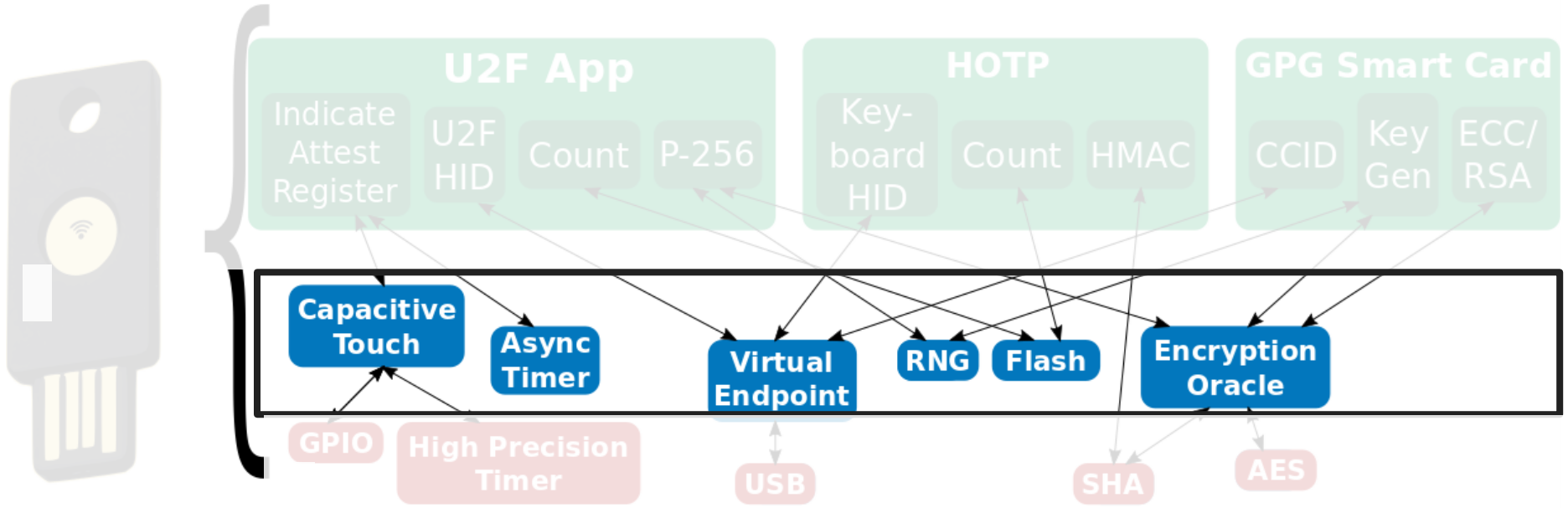
Goal: possible to correctly extend TCB

OS Services

Most OS services come from community

Device drivers, networking protocols, timers...

Platform provider can audit but won't catch all bugs



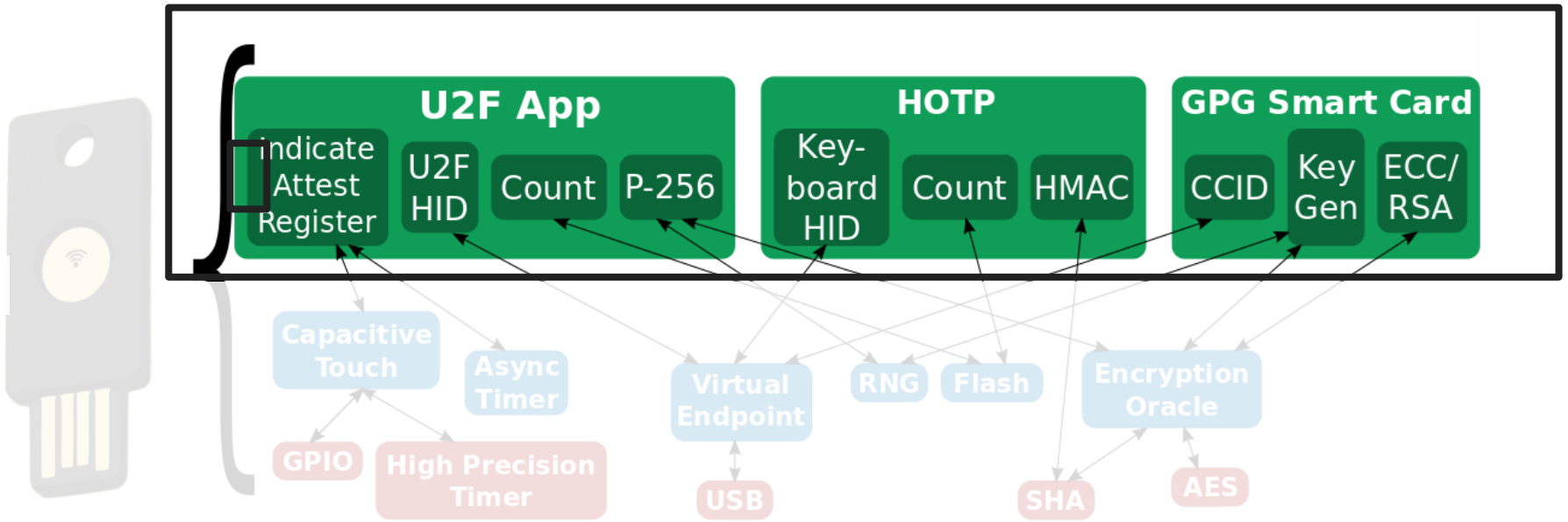
Goal: protect kernel from safety violations

Applications

Implement end-user functionality

“Third-party” developers: unknown to platform provider

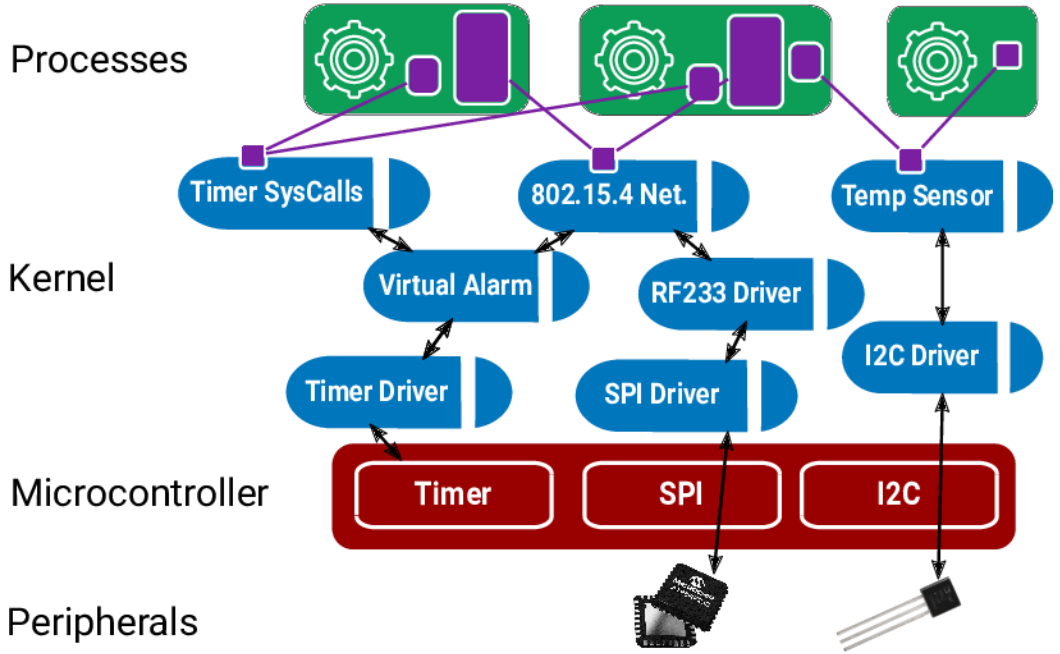
Potentially *malicious*



Goal: end-users can install 3rd-party apps

A Microcontroller OS

Processes: Use the Memory Protection Unit
Capsules: Type-safe Rust API for *safe* driver development
Grants: Bind dynamic kernel resources to process lifetime



What’s Next?

- Lecture 16
  - September 15, Thursday, 12 pm – 1 pm

COL788: Advanced Topics in Embedded Computing

Lecture 16 – Embedded OS (Cont.)

Vireshwar Kumar
CSE@IITD

September 15, 2022

Semester I
2022-2023

Tiny OS

- Low power wireless communication devices
  - Particularly wireless networked sensors
- Physical limitations
  - Computation ability
  - Memory
  - Power supply
  - High-level concurrency

Details

- An event-based operating system designed for wireless networked sensors.
- Designed to support concurrency-intensive operations required by networked sensors with minimal hardware requirements.
- C and Assembly languages
- Source code size: 500KB, 16KB commented lines

Different from Traditional OS

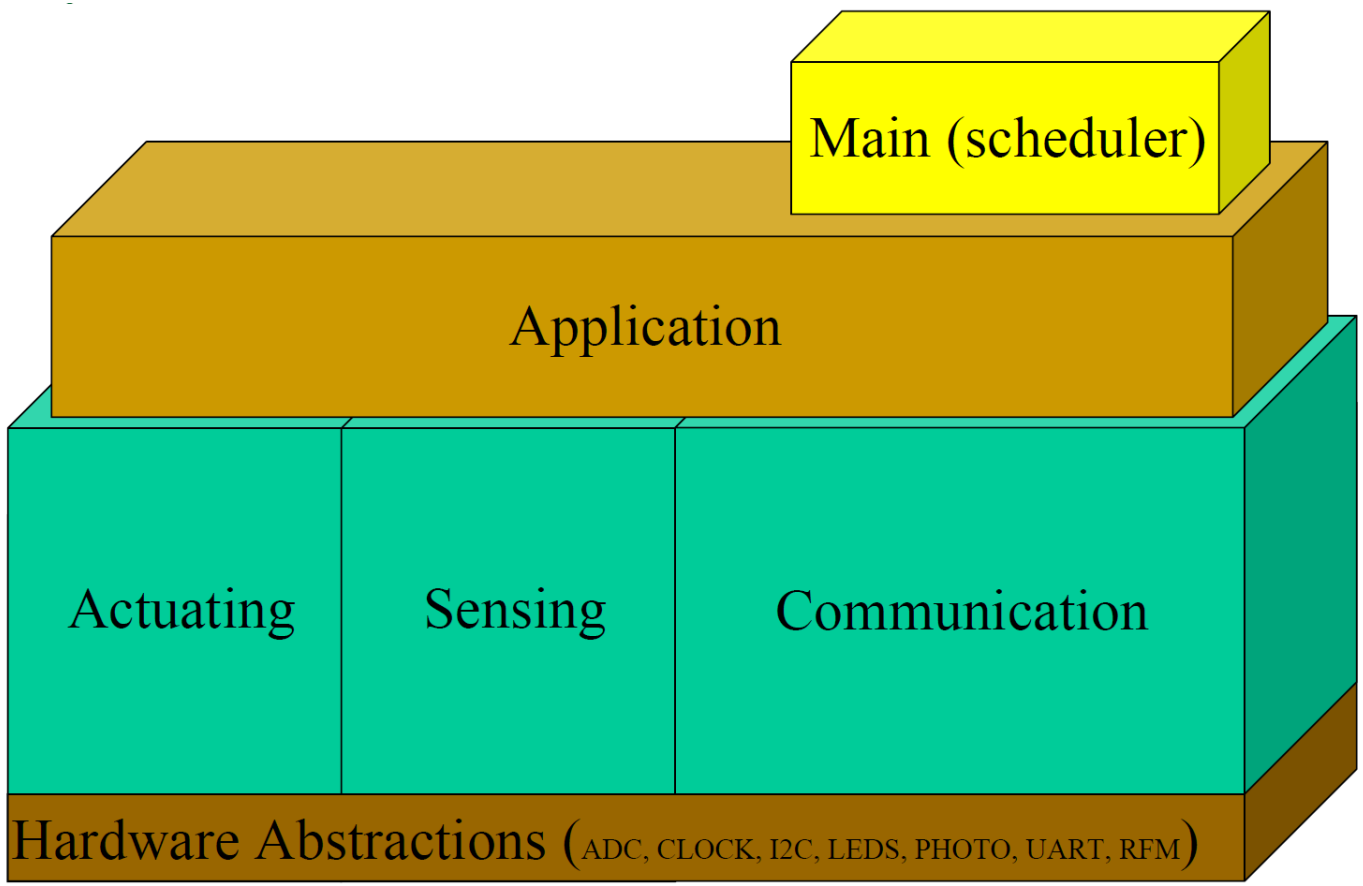
- Special purpose (not general purpose)
- Resource constraint
  - 4MHz ATMEL 8535 8bit MCU
  - 512 byte RAM and 8K Flash
- No dedicated I/O controller (missed deadline means loss data)
- One program at one time (no multi-programming)
- Thin-threads (tasks)

Model

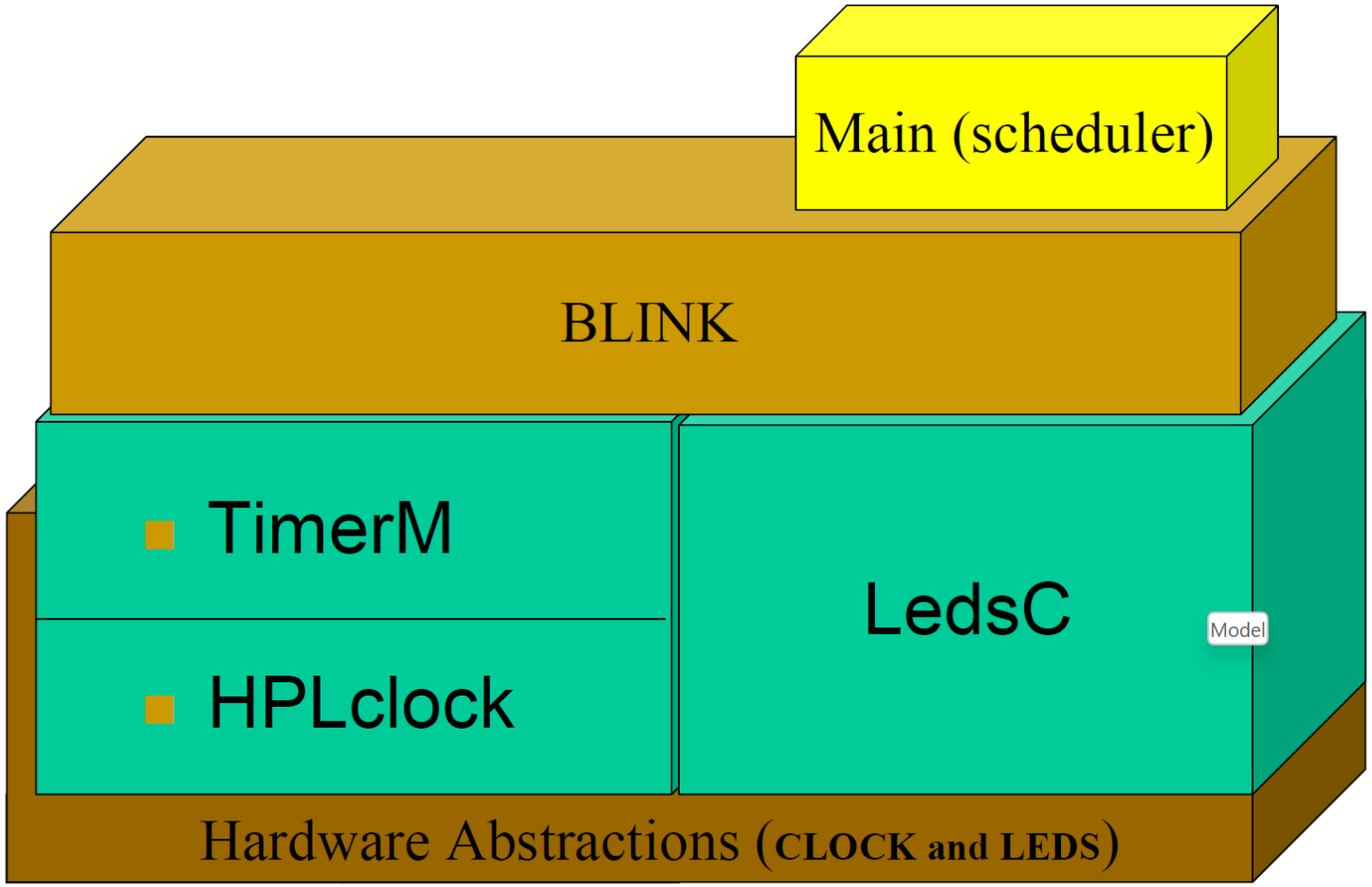
- Component-based model (modularity)
  - Simple functions are incorporated in components with clean interfaces
  - Complex functions can be implemented by composing components

- Event-based model
  - Interact with outside by events (no command shell)
  - There are two kinds of events for TinyOS:
    - External events: Clock events and message events;
    - Internal events triggered by external events

Structure


Main (scheduler)
Application
Actuating
Sensing
Communication
Hardware Abstractions (ADC, CLOCK, I2C, LEDS, PHOTO, UART, RFM)

Example


Main (scheduler)
BLINK
TimerM
HPLclock
LedsC
Model
Hardware Abstractions (CLOCK and LEDS)

What’s Next?

- Lecture 17
  - September 19, Monday, 11 am – 12 pm

COL788: Advanced Topics in Embedded Computing

Lecture 17 – Trusted Computing

Vireshwar Kumar
CSE@IITD

September 19, 2022

Semester I
2022-2023

Agenda

- Need for Trusted Computing
- Trusted Execution Architecture
- Example: Remote Attestation

Problem: Digital Right Management

Problem: Robust Security

Idea: Trusted Computing

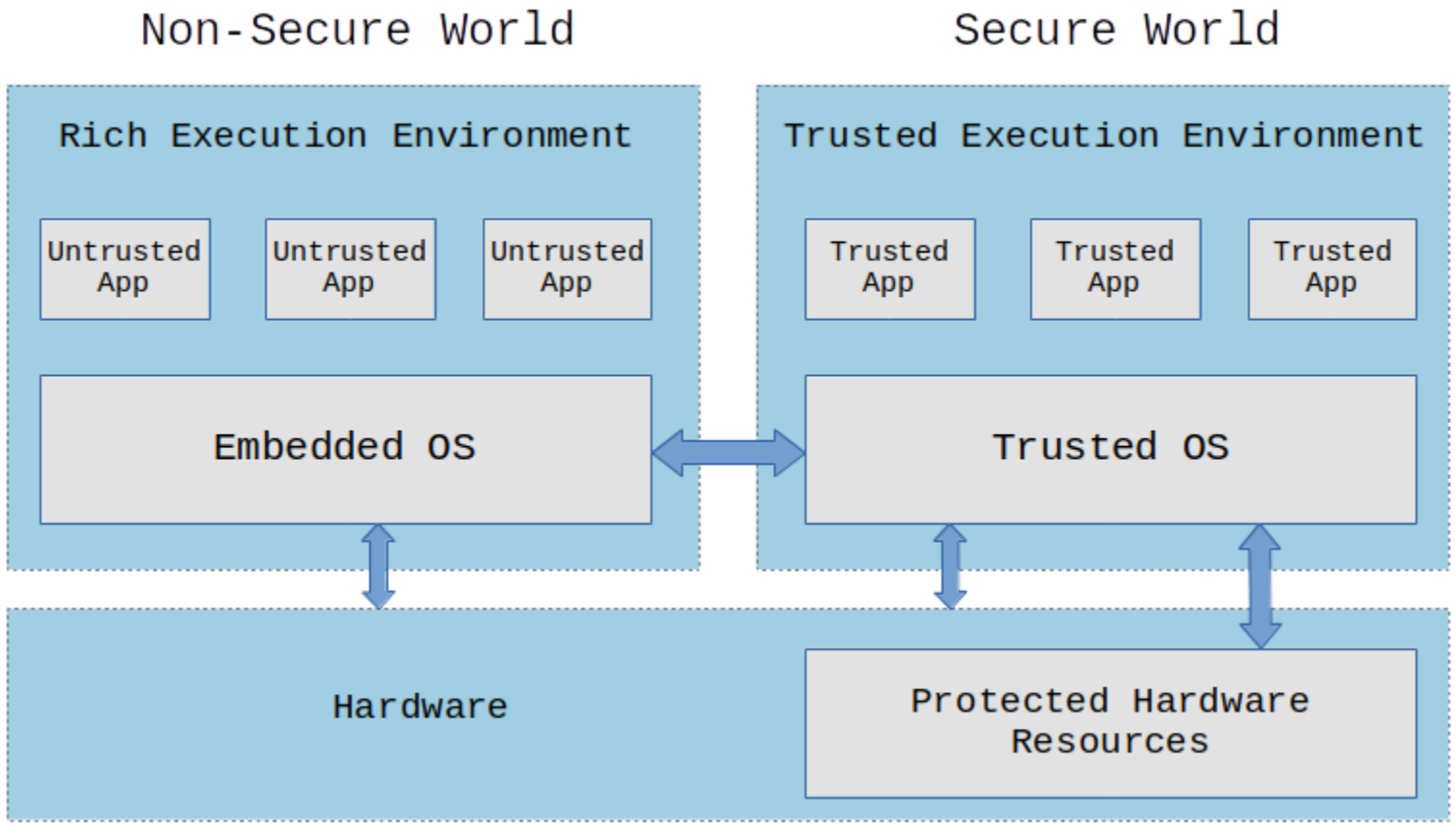
- Features
  - Isolation from the regular OS
  - Hardware-based security guarantees
  - Reconfigurability

- Implications
  - Enhanced confidence in the device security
  - Ensures that the device performs the way it is supposed to
  - Recovery after a potential compromise
  - Secure storage

Trusted Execution Environment OS

Company	Product	Hardware Used
Alibaba	Cloud Link TEE	
Apple	iOS Secure Enclave	Separate processor
BeanPod		Arm TrustZone
Huawei	iTrustee	Arm TrustZone
Google	Trusty	ARM / Intel
Linaro	OPTEE	Arm TrustZone
Qualcomm	QTEE	ARM TrustZone
Samsung	TEEgris	Arm TrustZone
TrustKernel	T6	Arm / Intel
Trustonic	Kinibi	Arm TrustZone
Trustonic	SW TEE	SW TEE on
Watchdata	WatchTrust	Arm TrustZone

ARM TEE Architecture



Example: Remote Attestation

Example: Remote Update

What’s Next?

- Lecture 18
  - September 21, Wednesday, 11 am – 12 pm

COL788: Advanced Topics in Embedded Computing

Lecture 18 – Trusted Computing (Cont.)

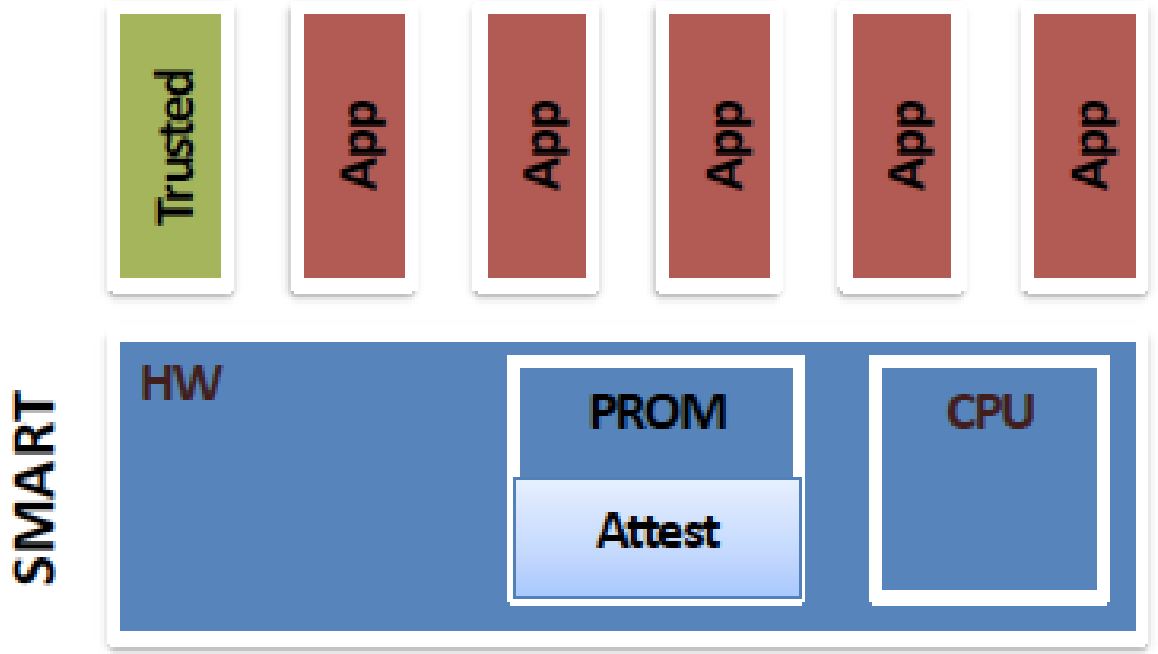
Vireshwar Kumar
CSE@IITD

September 21, 2022

Semester I
2022-2023

Basic Architecture

- SMART: Secure Minimal Architecture for (Establishing a Dynamic) Root of Trust



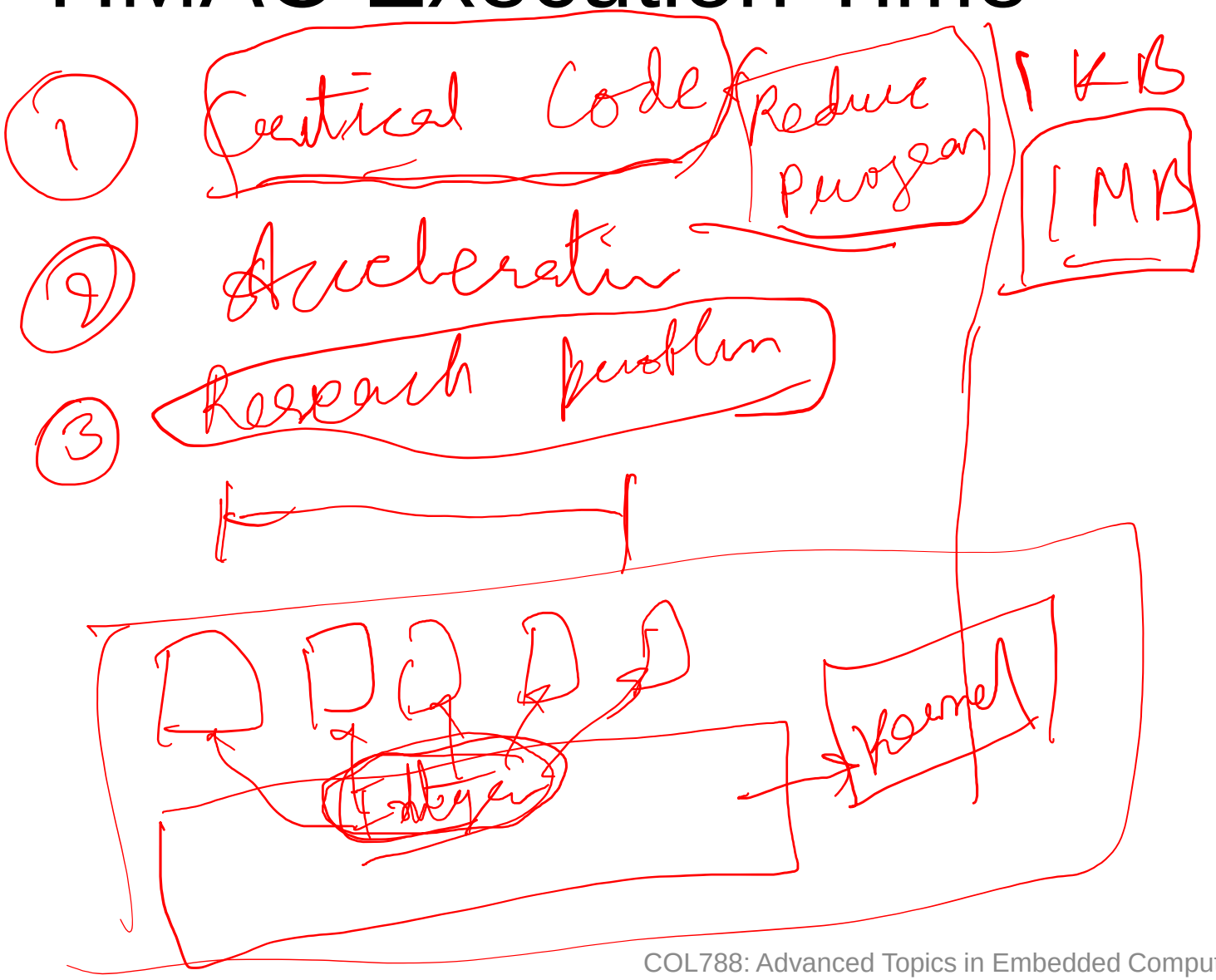
Threat Model

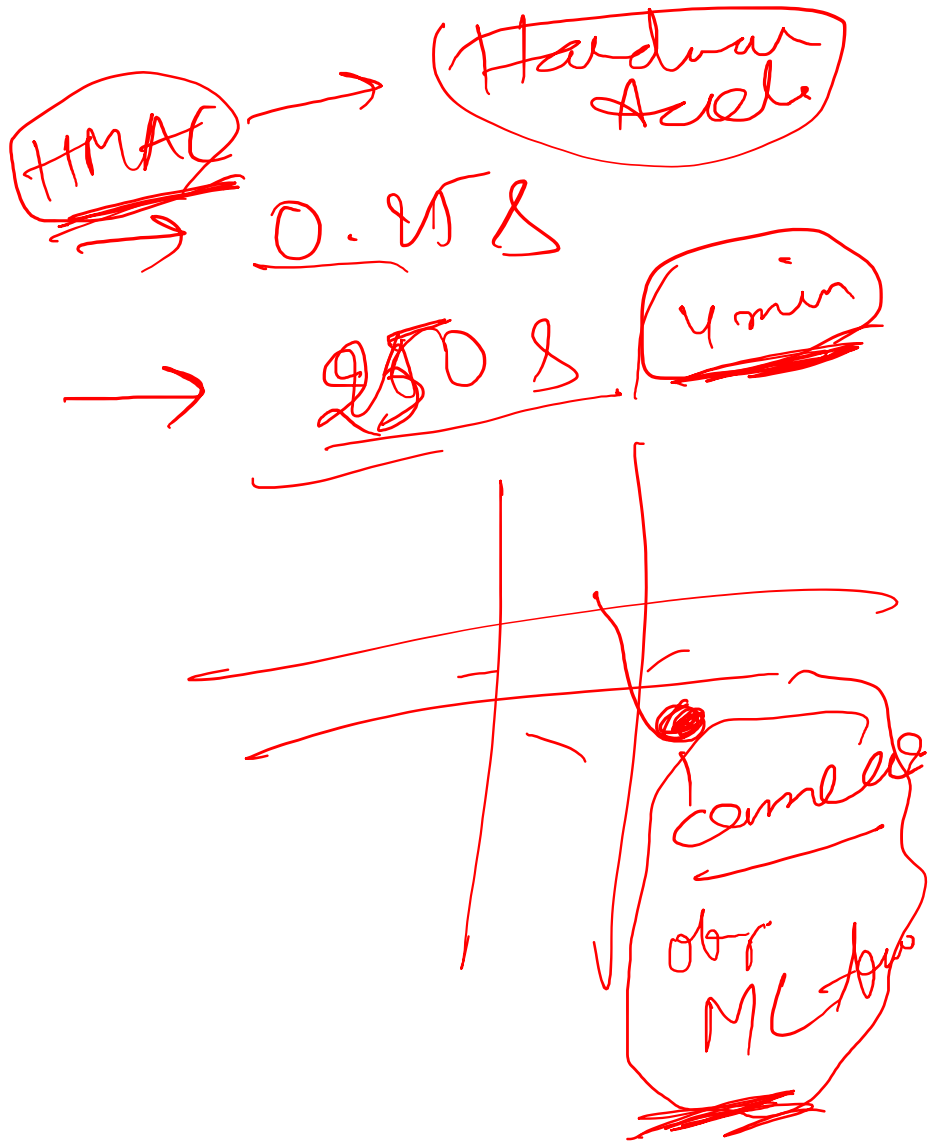
- System
  - Immutable ROM
  - RAM erased at reset

- Attacker
  - Full control over software
  - No invasive hardware attacks
  - No side-channel attacks

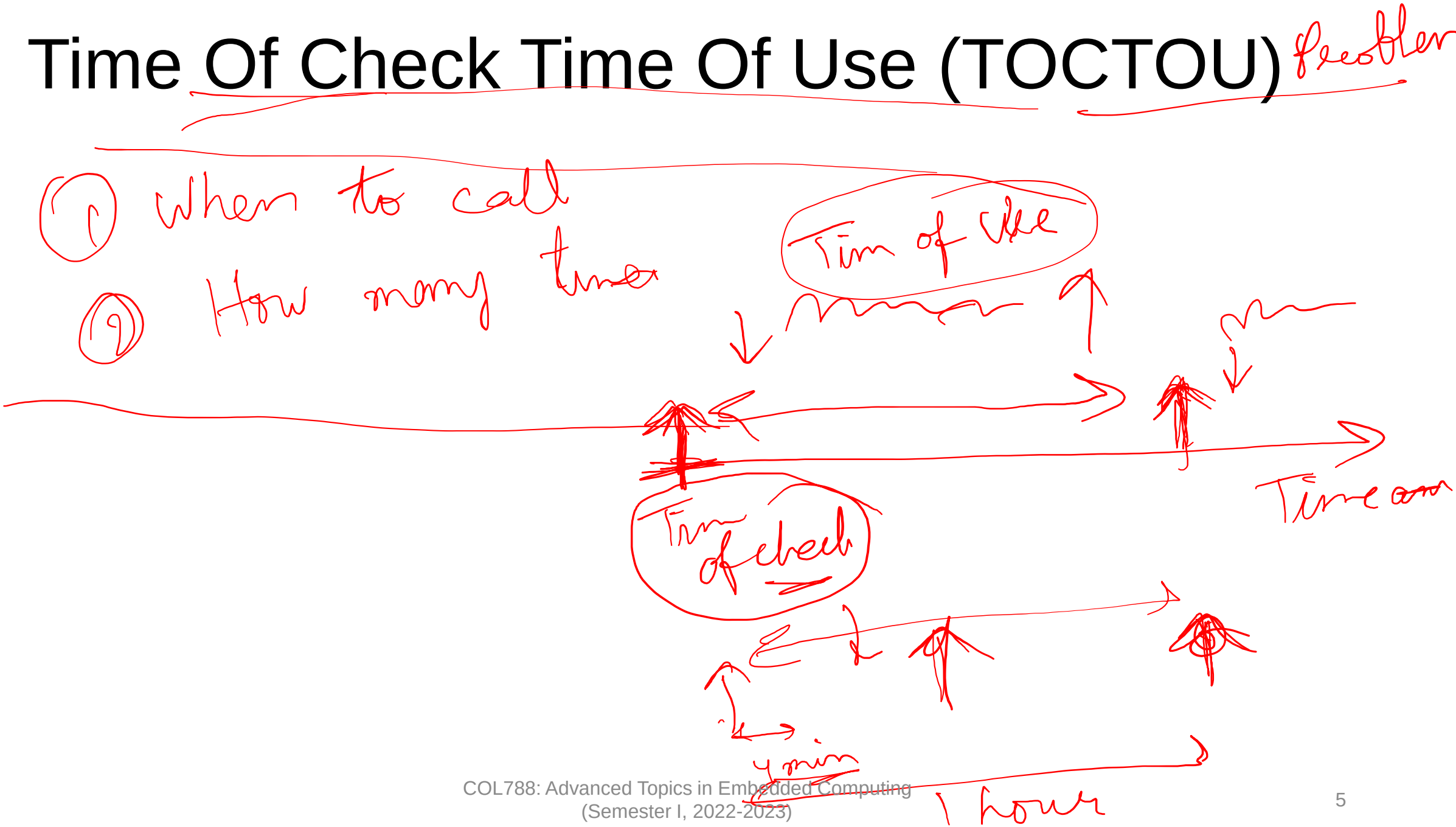
- Shared key between prover and verifier

HMAC Execution Time





Time Of Check Time Of Use (TOCTOU)



What Next?

- Lecture 19
  - September 22,Thursday, 12 pm – 1 pm