EECS 388: Embedded Systems

3. CPU and Memory

Heechul Yun

Agenda

- Instruction-set architecture
- Computer organization

Computer

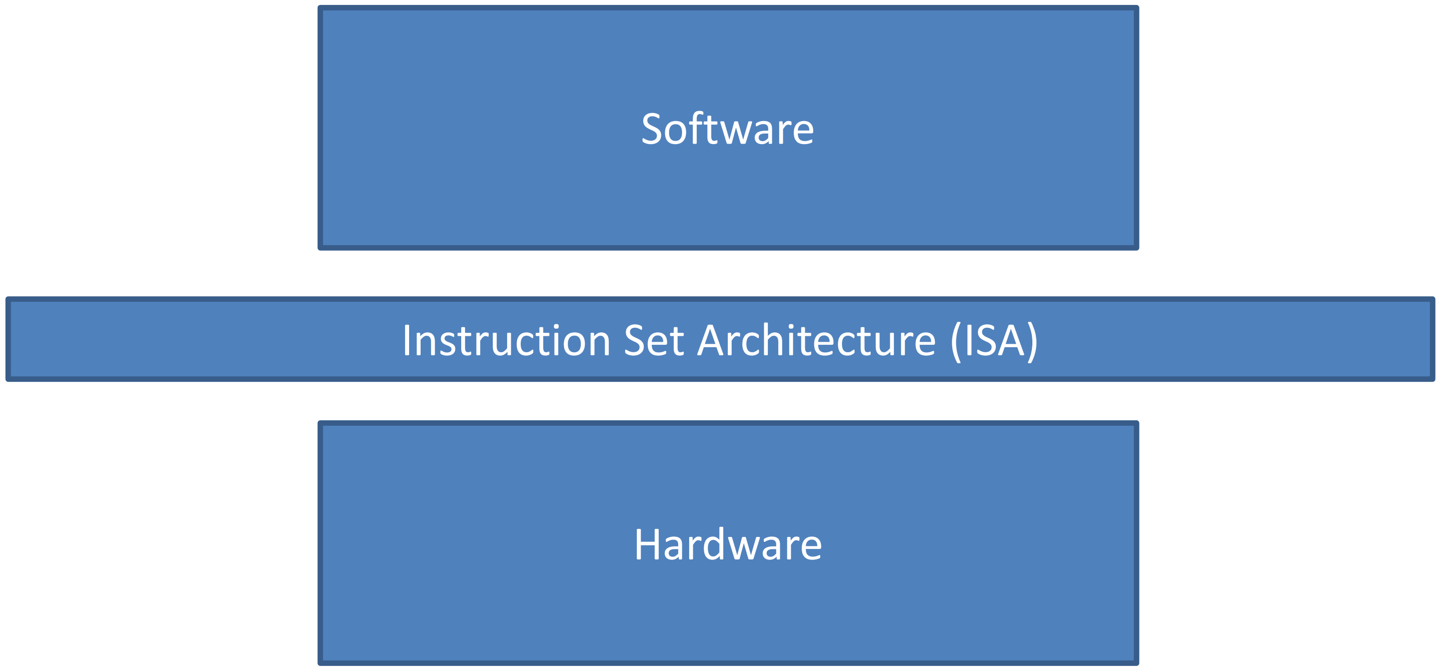
- Architecture
  - What it does
  - Instructions, program states, ..
- Microarchitecture
  - How it does what it does
  - Hardware structures and control logics
    - E.g., Cache, LRU cache replacement policy, …

Von Neumann Architecture

- Memory contains both program and data
- Program execution
  - Load instruction from memory
  - Update state (PC, registers, and memory)
  - Repeat
- Sequential process

KU
THE UNIVERSITY OF
KANSAS

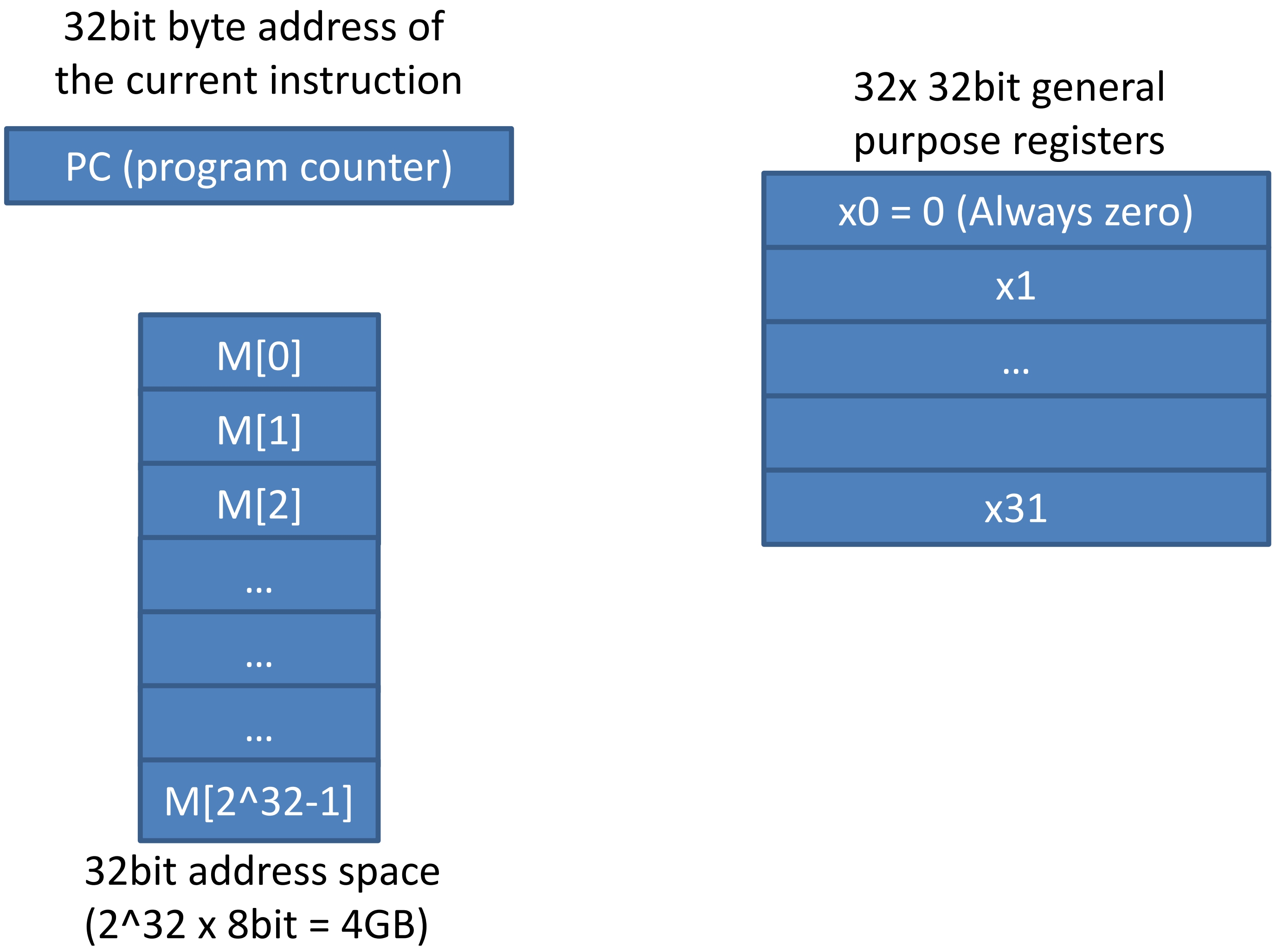
A Computing System



Instruction Set Architecture (ISA)

- A contract between software and hardware
- Defines an abstract hardware and its behaviors
  - Program visible states: memory, registers, PC, etc.
  - Instructions modify the states
- A single software binary can run on multiple different implementations of the same ISA
- Examples: x86 (x86-64), ARM (v7,v8), RISC-V (RV32I, RV32IMAC, RV64GC)

Program Visible State



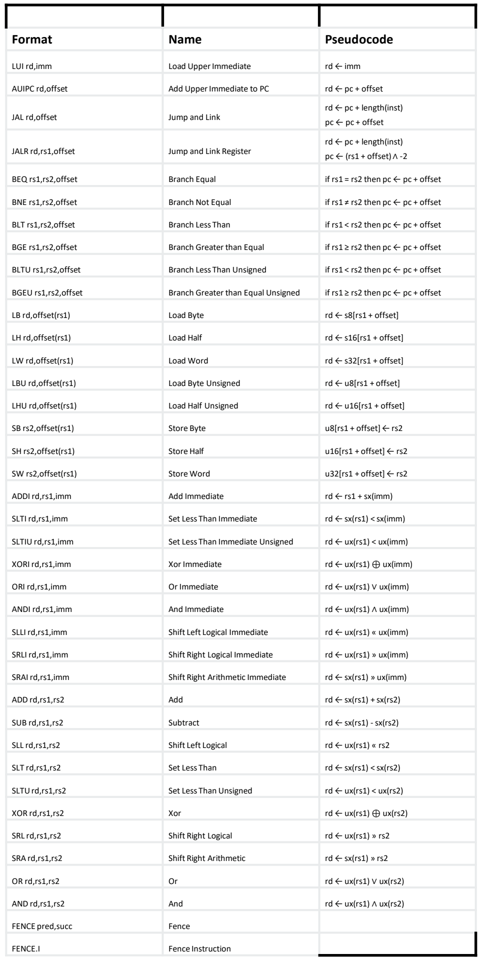
Program visible state of RV32I ISA

Instruction Classes

- Arithmetic and logical instructions
  - Fetch operands
  - Compute a result with the operands
  - Update the PC to the next instruction
- Data movement instructions
  - Load/store data from/to memory
  - Update the PC to the next instruction
- Control flow instructions
  - Fetch operands
  - Compute a branch condition and a target address
  - If branch condition is true, PC $\leftarrow$ the computed target; else PC $\leftarrow$ the next instruction

RISC-V RV32I ISA

39 instructions



Format	Name	Pseudocode
LUI rd,imm	Load Upper Immediate	rd ← imm
AUIPC rd,offset	Add Upper Immediate to PC	rd ← pc + offset
JAL rd,offset	Jump and Link	rd ← pc + length(inst) pc ← pc + offset
JALR rd,rs1,offset	Jump and Link Register	rd ← pc + length(inst) pc ← (rs1 + offset) ∧ -2
BEQ rs1,rs2,offset	Branch Equal	if rs1 = rs2 then pc ← pc + offset
BNE rs1,rs2,offset	Branch Not Equal	if rs1 ≠ rs2 then pc ← pc + offset
BLT rs1,rs2,offset	Branch Less Than	if rs1 < rs2 then pc ← pc + offset
BGE rs1,rs2,offset	Branch Greater than Equal	if rs1 ≥ rs2 then pc ← pc + offset
BLTU rs1,rs2,offset	Branch Less Than Unsigned	if rs1 < rs2 then pc ← pc + offset
BGEU rs1,rs2,offset	Branch Greater than Equal Unsigned	if rs1 ≥ rs2 then pc ← pc + offset
LB rd,offset(rs1)	Load Byte	rd ← s8[rs1 + offset]
LH rd,offset(rs1)	Load Half	rd ← s16[rs1 + offset]
LW rd,offset(rs1)	Load Word	rd ← s32[rs1 + offset]
LBU rd,offset(rs1)	Load Byte Unsigned	rd ← u8[rs1 + offset]
LHU rd,offset(rs1)	Load Half Unsigned	rd ← u16[rs1 + offset]
SB rs2,offset(rs1)	Store Byte	u8[rs1 + offset] ← rs2
SH rs2,offset(rs1)	Store Half	u16[rs1 + offset] ← rs2
SW rs2,offset(rs1)	Store Word	u32[rs1 + offset] ← rs2
ADDI rd,rs1,imm	Add Immediate	rd ← rs1 + sx(imm)
SLTI rd,rs1,imm	Set Less Than Immediate	rd ← sx(rs1) < sx(imm)
SLTIU rd,rs1,imm	Set Less Than Immediate Unsigned	rd ← ux(rs1) < ux(imm)
XORI rd,rs1,imm	Xor Immediate	rd ← ux(rs1) ⊕ ux(imm)
ORI rd,rs1,imm	Or Immediate	rd ← ux(rs1) ∨ ux(imm)
ANDI rd,rs1,imm	And Immediate	rd ← ux(rs1) ∧ ux(imm)
SLLI rd,rs1,imm	Shift Left Logical Immediate	rd ← ux(rs1) « ux(imm)
SRLI rd,rs1,imm	Shift Right Logical Immediate	rd ← ux(rs1) » ux(imm)
SRAI rd,rs1,imm	Shift Right Arithmetic Immediate	rd ← sx(rs1) » ux(imm)
ADD rd,rs1,rs2	Add	rd ← sx(rs1) + sx(rs2)
SUB rd,rs1,rs2	Subtract	rd ← sx(rs1) - sx(rs2)
SLL rd,rs1,rs2	Shift Left Logical	rd ← ux(rs1) « rs2
SLT rd,rs1,rs2	Set Less Than	rd ← sx(rs1) < sx(rs2)
SLTU rd,rs1,rs2	Set Less Than Unsigned	rd ← ux(rs1) < ux(rs2)
XOR rd,rs1,rs2	Xor	rd ← ux(rs1) ⊕ ux(rs2)
SRL rd,rs1,rs2	Shift Right Logical	rd ← ux(rs1) » rs2
SRA rd,rs1,rs2	Shift Right Arithmetic	rd ← sx(rs1) » rs2
OR rd,rs1,rs2	Or	rd ← ux(rs1) ∨ ux(rs2)
AND rd,rs1,rs2	And	rd ← ux(rs1) ∧ ux(rs2)
FENCE pred,succ	Fence	
FENCE.I	Fence Instruction	

https://rv8.io/isa.html

ADD Instruction

- Assembly
  - ADD rd, rs1, rs2
- Machine encoding

0000000	rs2	rs1	000	rd	0110011
7bit	5bit	5bit	3bit	5bit	7bit

- Semantics
  - GPR[rd] ← GPR[rs1] + GPR[rs2]
  - PC ← PC + 4

Assembly Programming Example

- C code

```
f = (g + h) + (i + j)
```

- Assembly code
  - Assume f, g, h, I, j are in r_f, r_g, r_h, r_i, r_j registers
  - Assume r_{temp} is a free register

ADD r_{temp},r_g,r_h
ADD r_f,r_i,r_j
ADD r_f,r_{temp},r_f

Load Instruction

- Assembly
  - LW rd, offset_{12}(base)

- Semantics
  - addr = offset + GPR[base]
  - GPR[rd] $\leftarrow$ MEM[addr]
  - PC $\leftarrow$ PC + 4

Store Instruction

- Assembly
  - SW rs2, offset_{12}(base)

- Semantics
  - addr = offset + GPR[base]
  - MEM[addr] = GPR[rs2]
  - PC $\leftarrow$ PC + 4

Assembly Programming Example 2

- C code

```
A[8] = h + A[0]
```

- Assembly code
  - Assume &A, h, are in r_A, r_h registers
  - Assume r_{temp} is a free register

```
LW r_temp,0(r_A)
ADD r_temp,r_h,r_temp
SW r_temp,32(r_A)
```

Conditional Branch Instruction

- Assembly
  - BEQ rs2, rs1, imm_{13}

- Semantics
  - target_addr = PC + imm
  - If GPR[rs1] == GPR[rs2] then PC $\leftarrow$ target_addr else PC $\leftarrow$ PC + 4

Assembly Programming Example 3

- C code

```
if (i == j) { e = g; } else { e = h;}
f = e
```

- Assembly code
  - Assume e, f, g, h, i, j are in r_e, r_f, r_g, r_h, r_i, r_j registers

```
        BEQ r_i,r_j,L1
        ADD r_e,r_h,x0
        BEQ x0,x0,L2
L1:     ADD r_e,r_g,x0
L2:     ADD r_f,r_e,x0
```

Function Call/Return

```
A:    BEQ x0,x0,F
      ...

B:    BEQ x0,x0,F
      ...

F:    ...
      BEQ x0,x0,???
```

- Need to jump back to the callers
- Where to jump?

Jump and Link Instruction

- Assembly
  - JAL rd, imm

- Semantics
  - target_addr = PC + imm
  - GPR[rd] ← PC + 4
  - PC ← target_addr

Jump and Link Register Instruction

- Assembly
  - JAL rd, rs1, imm

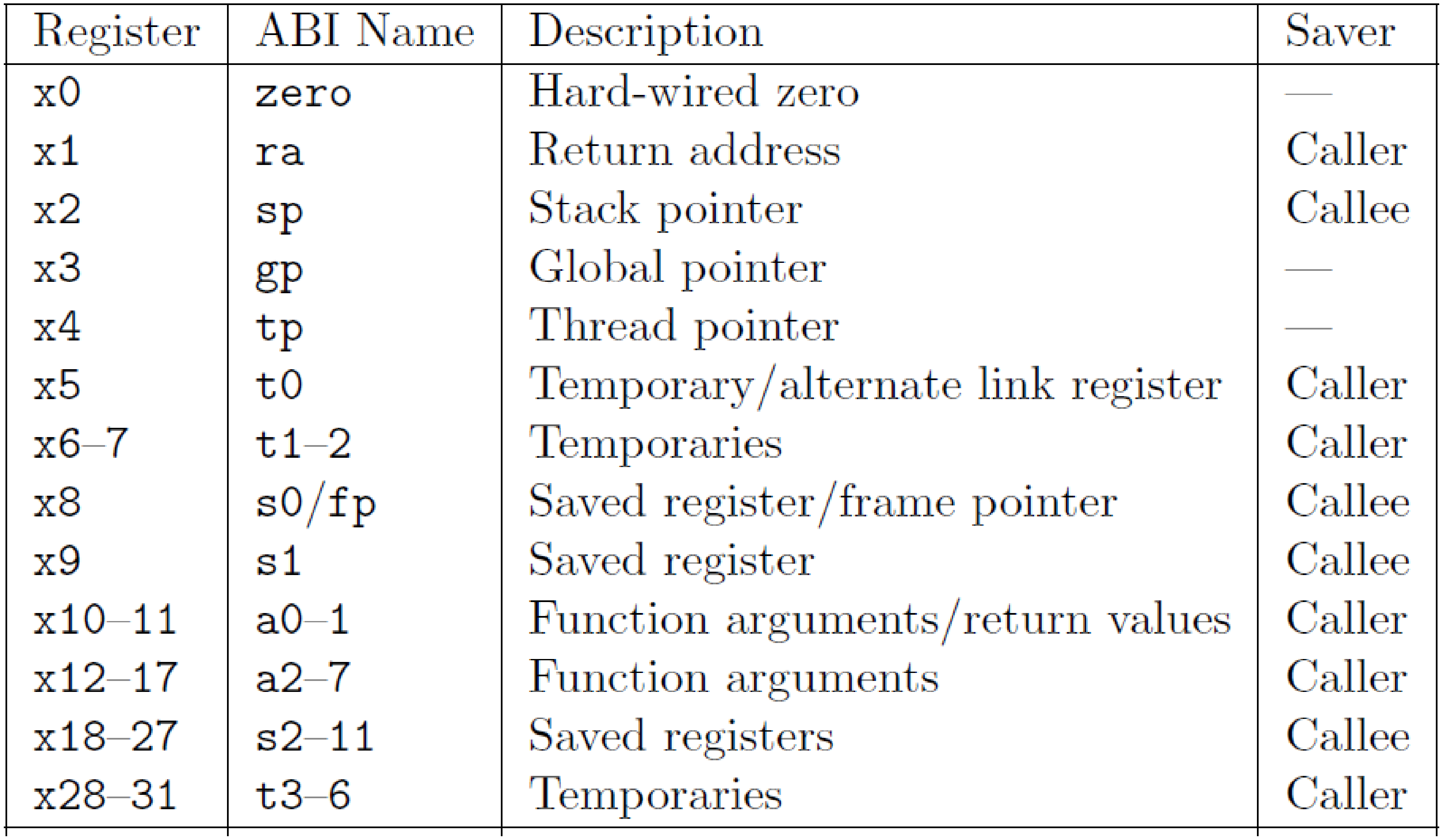
- Semantics
  - target_addr = (GPR[rs1] + imm) & 0xffff_fffe
  - GPR[rd] $\leftarrow$ PC + 4
  - PC $\leftarrow$ target_addr

Function Call/Return

```
A:    JAL x1,F
      ...

B:    JAL x1,F
      ...

F:    ...
      JALR x0,x1,0
```

- x1 register = return address
- Arguments? What registers to use?

Register Usage Convention



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Table 20.1: Assembler mnemonics for RISC-V integer and floating-point registers.
The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2

Caller/Callee Saved Registers

- Callee saved registers
  - Callee must restore the values of the caller
- Caller saved registers
  - Callee can update them freely

Calling Convention

- Caller saves caller-saved registers
- Caller loads arguments into x10-x17
- Caller jumps to the callee
- Callee allocates space on the stack
- Callee saves callee saved registers on the stack
- Do the work
- Callee loads the results into x10, x11
- Callee restore saved registers
- Return (JARL x0 x1 0)

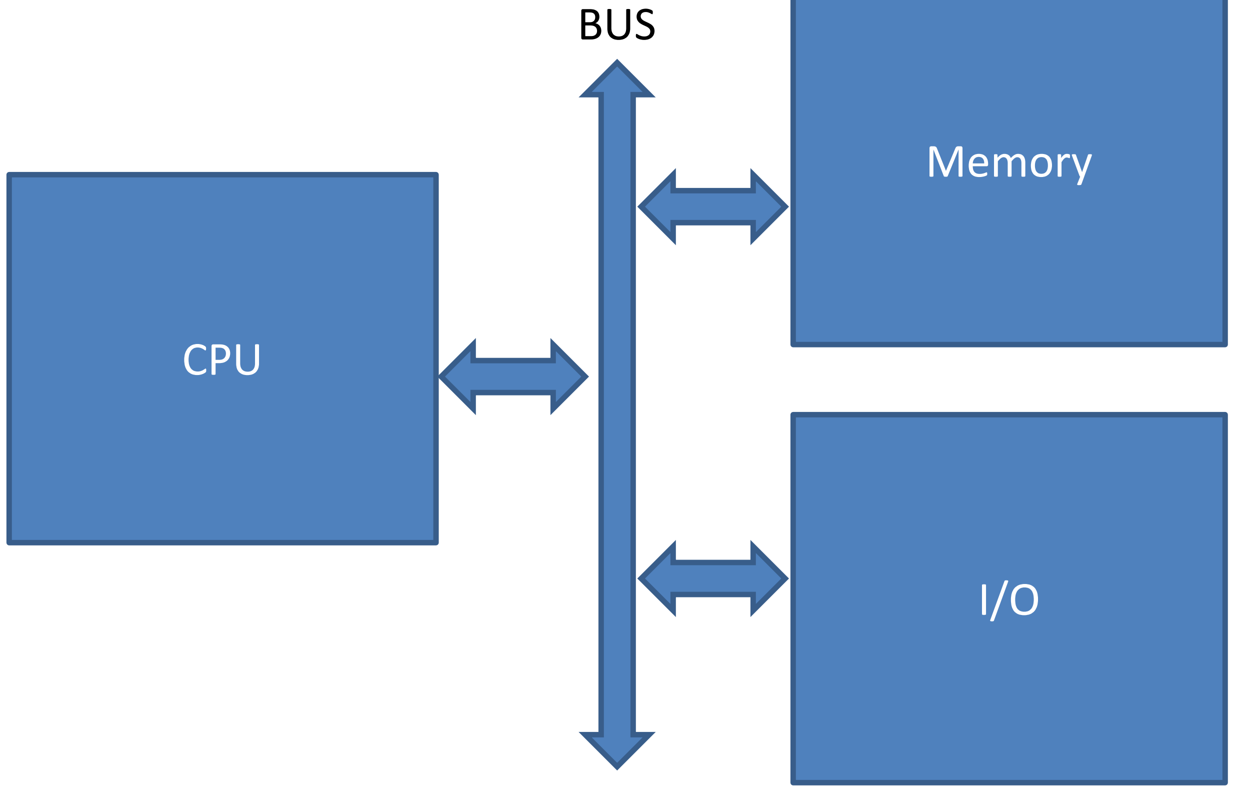
Computer

- Architecture
  - What it does
  - Instructions, program states, ..

- **Microarchitecture**
  - **How it does what it does**
  - **Hardware structures and control logics**
    - **E.g., Cache, LRU cache replacement policy, …**

KU THE UNIVERSITY OF KANSAS

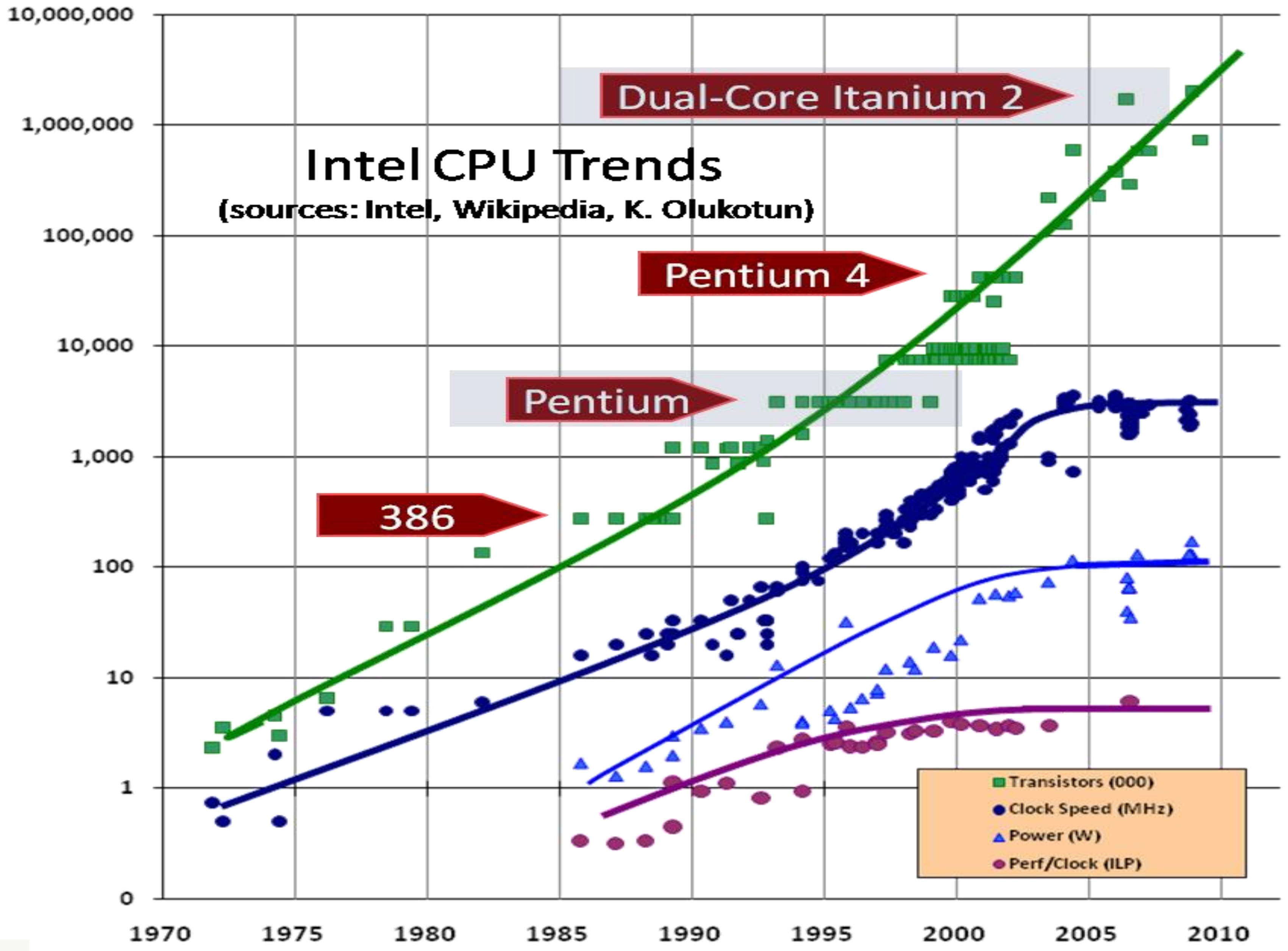
Computer Organization



Central Processing Unit (CPU)

- The brain of a computer
  - Fetch instruction from memory
  - Decode and execute
  - Store results on memory/registers

- Moore’s law
  - Transistors double every 1~2yr

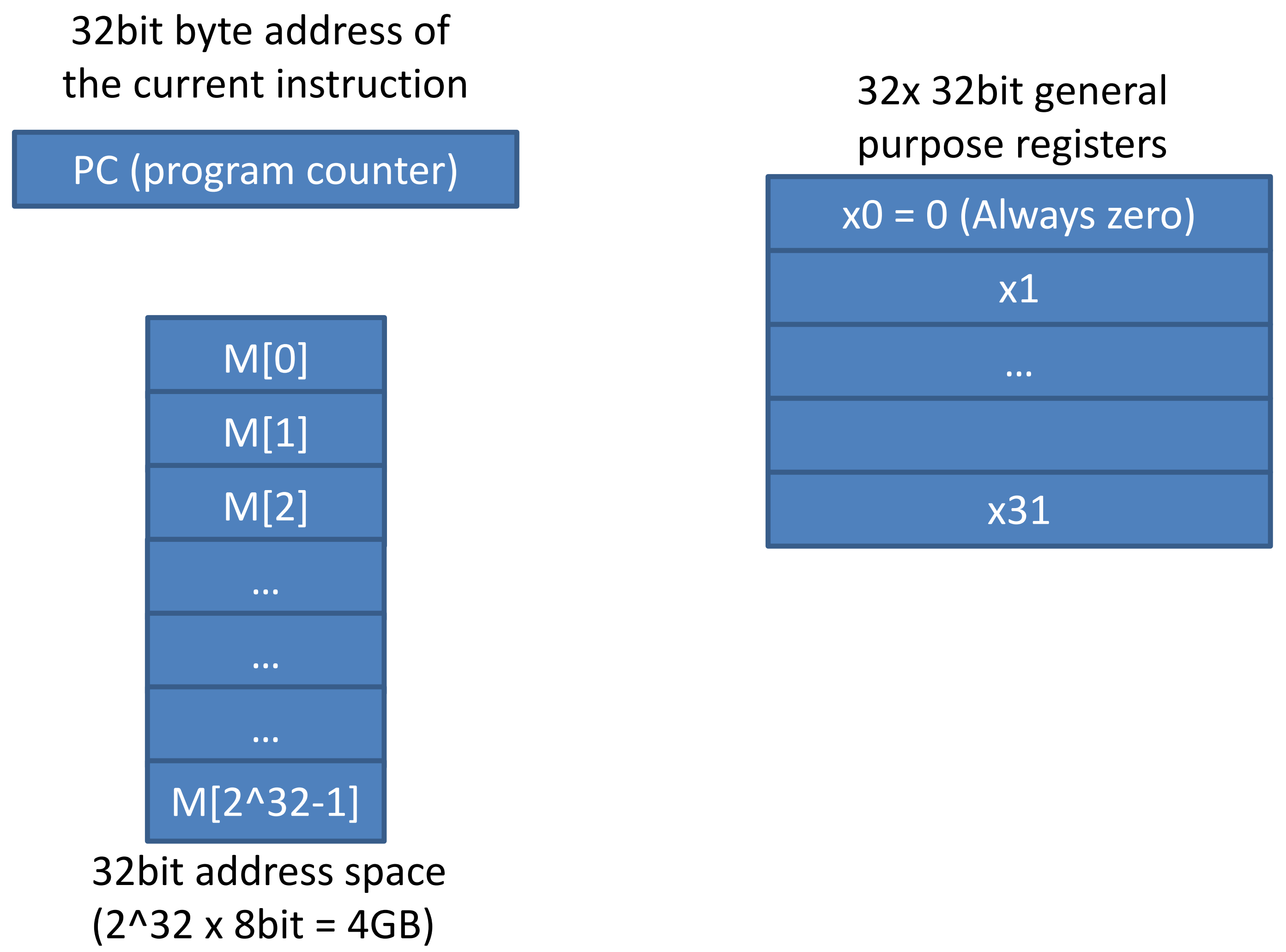


H Sutter, "The Free Lunch Is Over", Dr. Dobb's Journal, 2009

CPU

- Three basic components
  - Arithmetic Logical Unit: do the computing
  - Control Unit: do the flow control
  - Registers: temporary storage for the CPU.
- The complexity of a CPU can vary wildly
  - In-order vs. out-of-order. multi-level caches, prefetchers, branch predictors, etc.

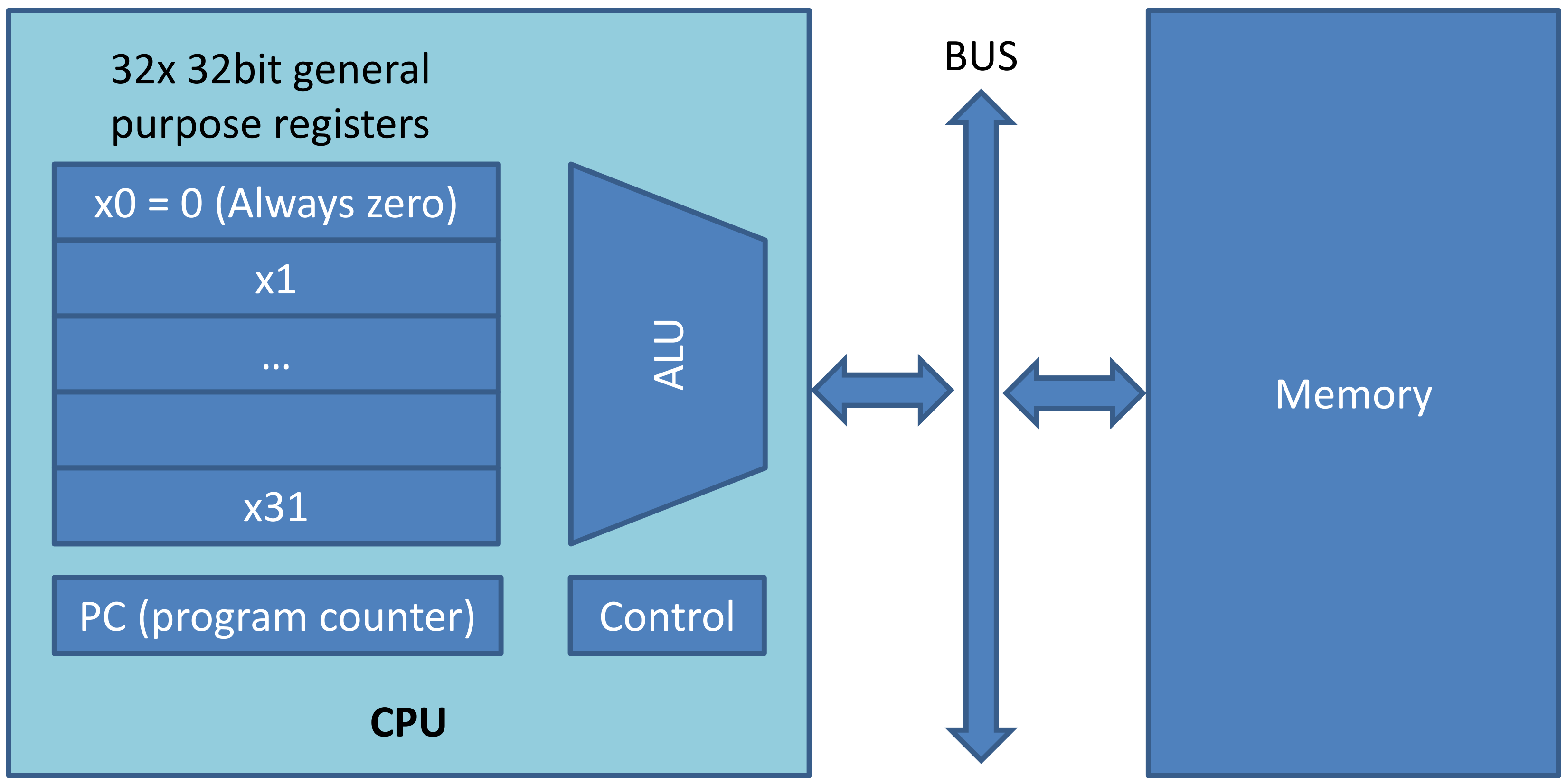
Recap: Program Visible State



32bit byte address of the current instruction
PC (program counter)
M[0]
M[1]
M[2]
…
…
…
M[2^32-1]
32bit address space (2^32 x 8bit = 4GB)
32x 32bit general purpose registers
x0 = 0 (Always zero)
x1
…
x31

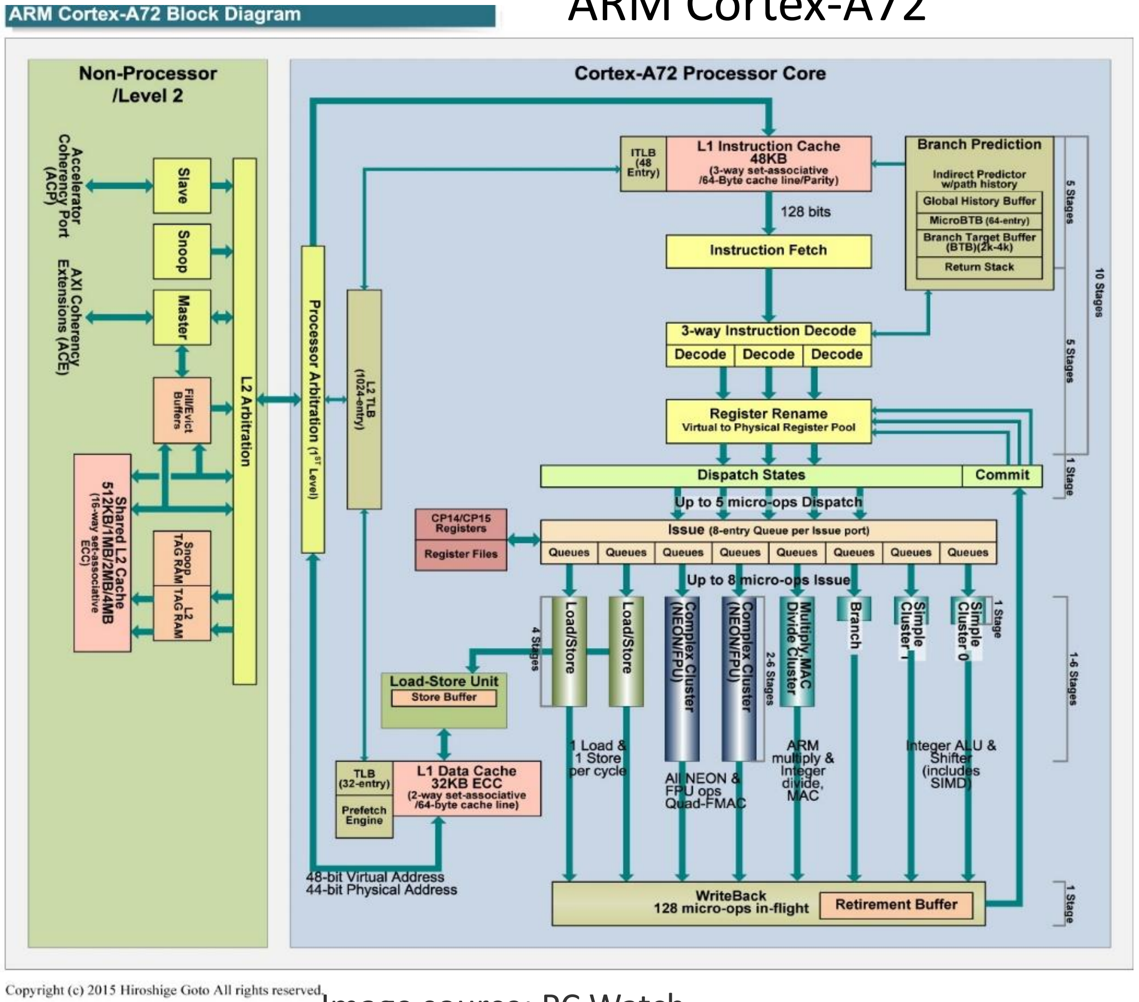
Program visible state of RV32I ISA

CPU: Architectural View



CPU: Micro-Architectural View

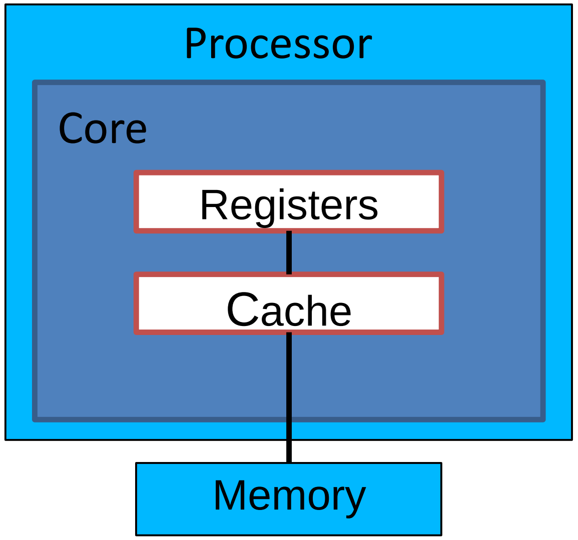
ARM Cortex-A72



ARM Cortex-A72 Block Diagram
Non-Processor /Level 2
Cortex-A72 Processor Core
Accelerator Coherency Port (ACP)
AXI Coherency Extensions (ACE)
Slave
Snoop
Master
Fill/Evict Buffers
L2 Arbitration
Shared L2 Cache 512KB/1MB/2MB/4MB (16-way set-associative ECC)
Snoop TAG RAM
L2 TAG RAM
Processor Arbitration (1ST Level)
L2 TLB (1024-entry)
ITLB (48 Entry)
L1 Instruction Cache 48KB (3-way set-associative /64-Byte cache line/Parity)
128 bits
Instruction Fetch
Branch Prediction
Indirect Predictor w/path history
Global History Buffer
MicroBTB (64-entry)
Branch Target Buffer (BTB)(2k-4k)
Return Stack
5 Stages
10 Stages
5 Stages
1 Stage
3-way Instruction Decode
Decode
Decode
Decode
Register Rename
Virtual to Physical Register Pool
Dispatch States
Commit
Up to 5 micro-ops Dispatch
CP14/CP15 Registers
Register Files
Issue (8-entry Queue per Issue port)
Queues
Up to 8 micro-ops Issue
Load/Store
Load/Store
Complex Cluster (NEON/FPU)
Complex Cluster (NEON/FPU)
Multiply MAC Divide Cluster
Branch
Simple Cluster 1
Simple Cluster 0
4 Stages
2-6 Stages
1 Stage
1-6 Stages
Load-Store Unit
Store Buffer
1 Load & 1 Store per cycle
All NEON & FPU ops Quad-FMAC
ARM multiply & Integer divide, MAC
Integer ALU & Shifter (includes SIMD)
TLB (32-entry)
Prefetch Engine
L1 Data Cache 32KB ECC (2-way set-associative /64-byte cache line)
48-bit Virtual Address
44-bit Physical Address
WriteBack 128 micro-ops in-flight
Retirement Buffer
1 Stage

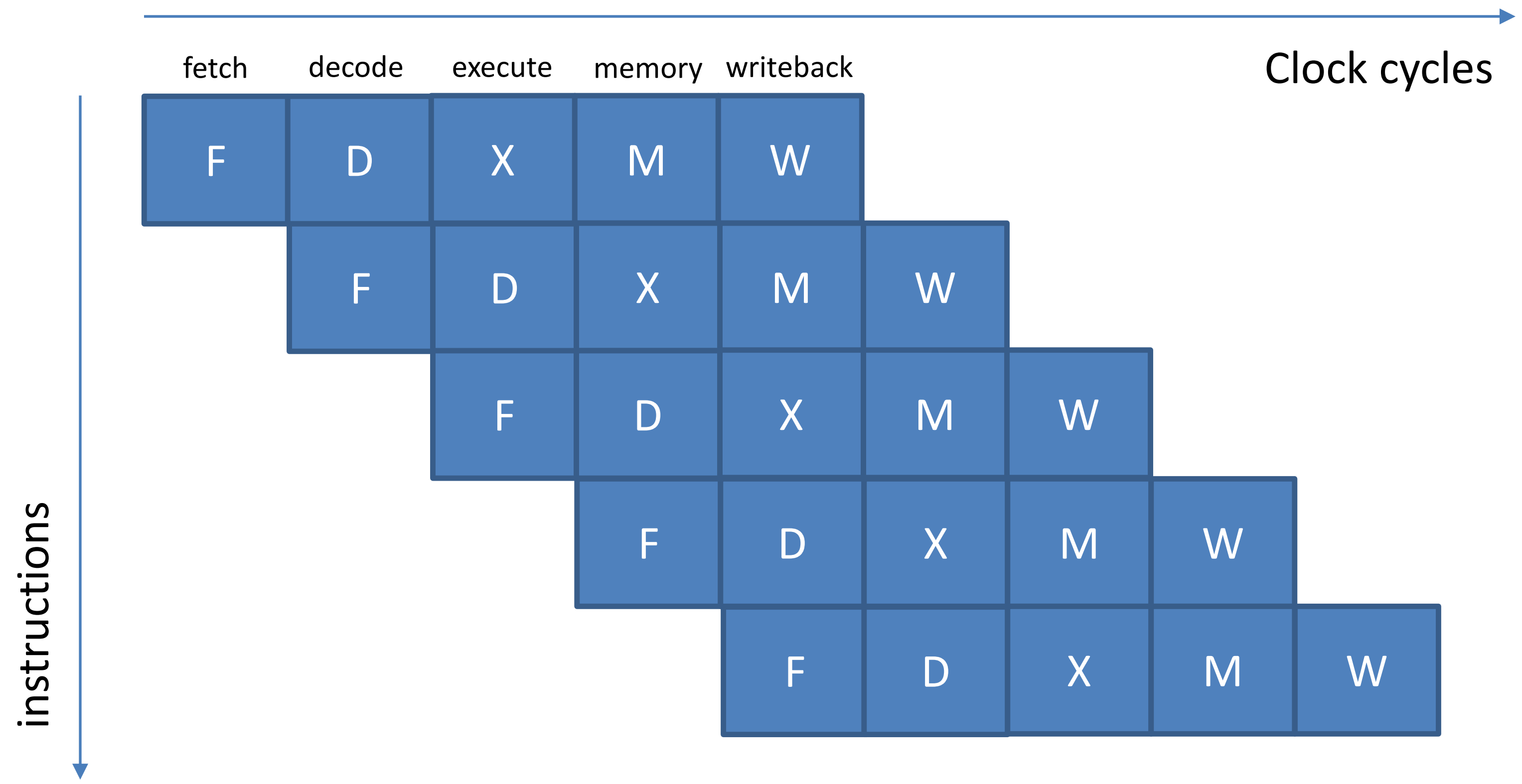
Image source: PC Watch.

Single-core CPU



- Sequential processing
  – One task at a time
  – (but multiple instructions can be executed at a time)

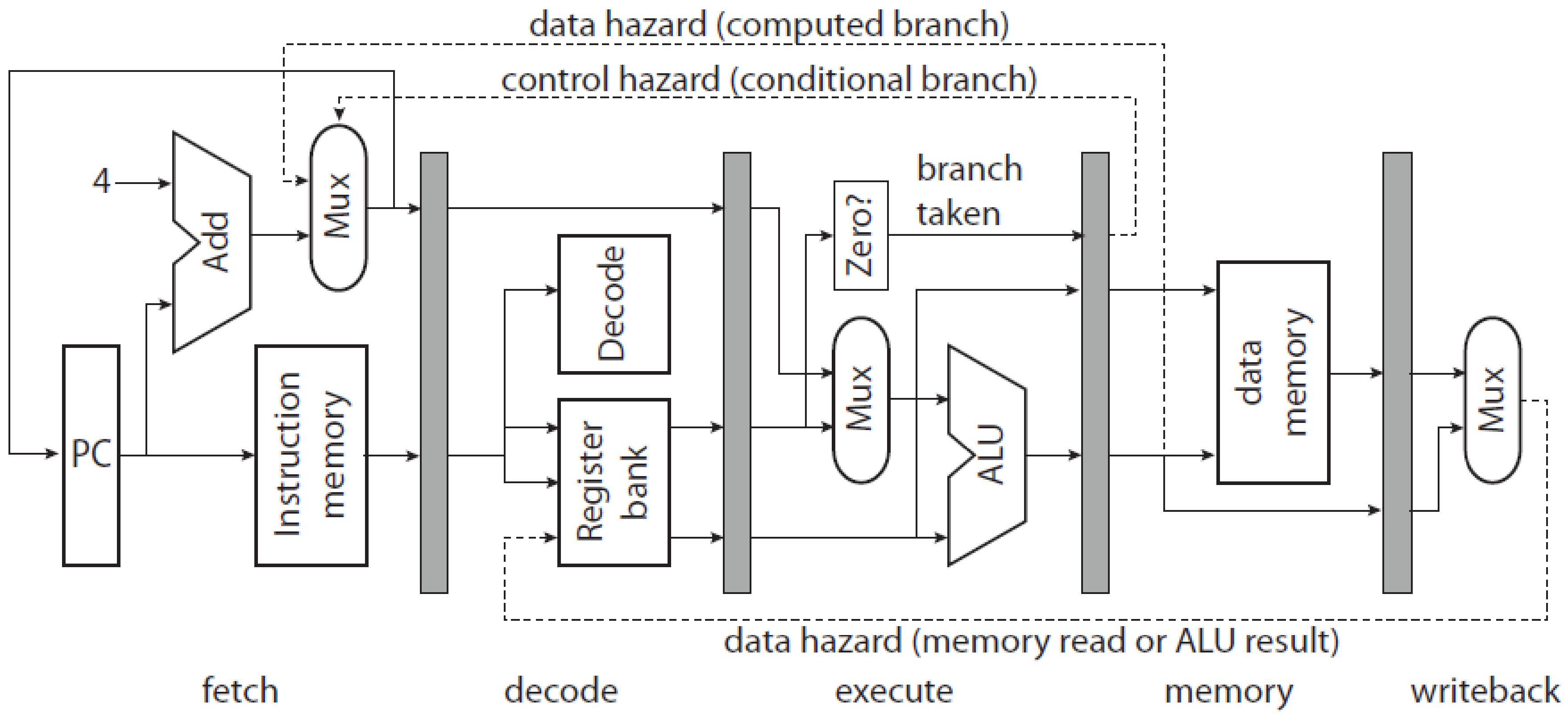
Pipelining



- Divide the work into multiple small pieces
- Easier to increase the clock speed

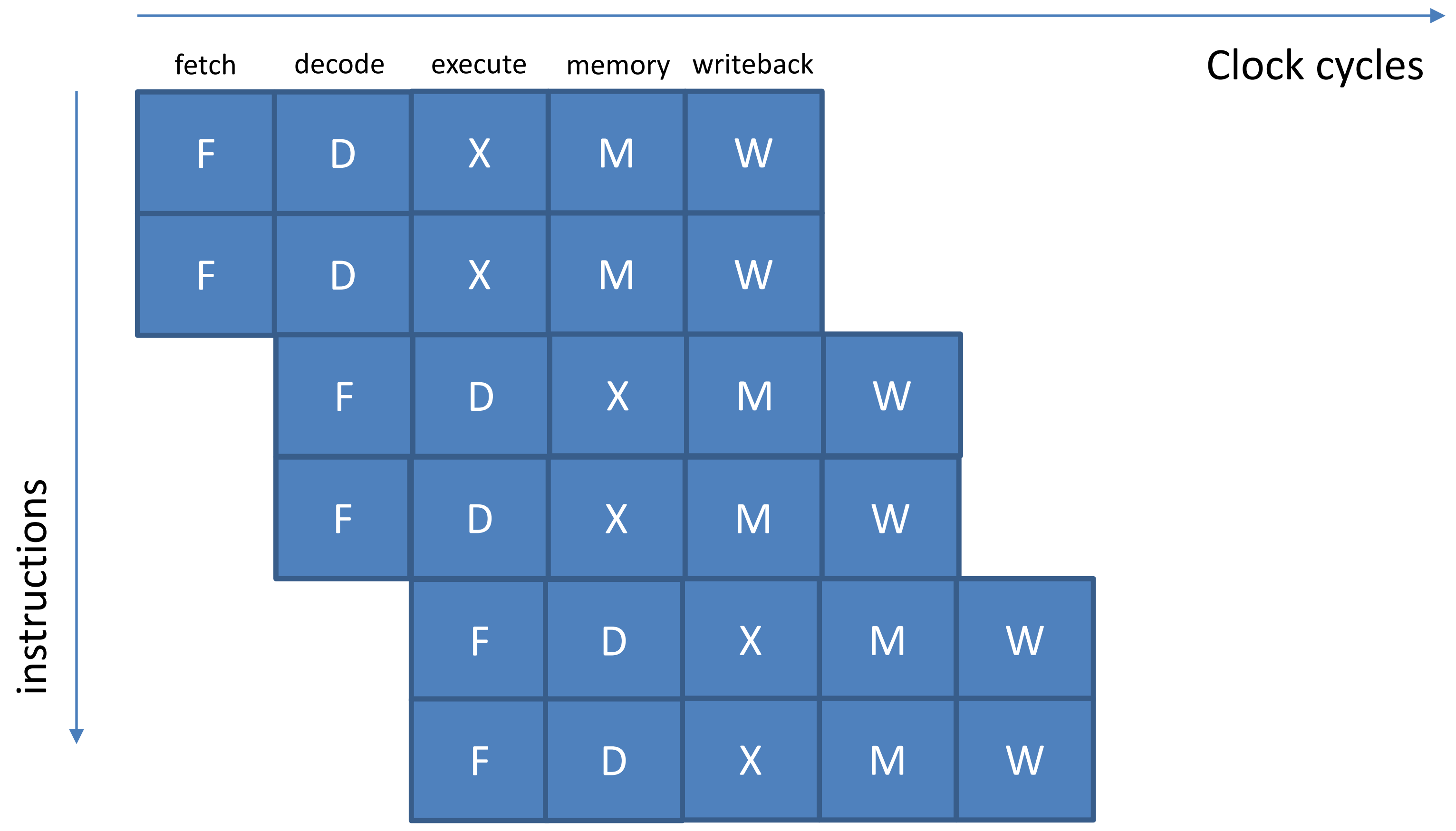
Pipelining

- A simple 5 stage pipeline



Source: Figure 8.2. of the textbook

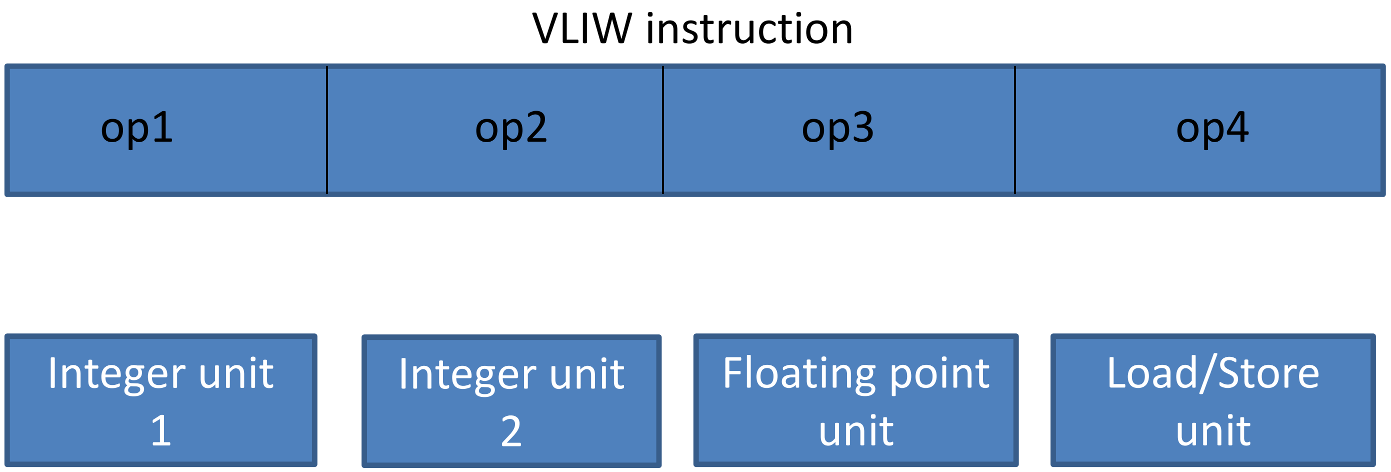
Superscalar (Multi-issuing)



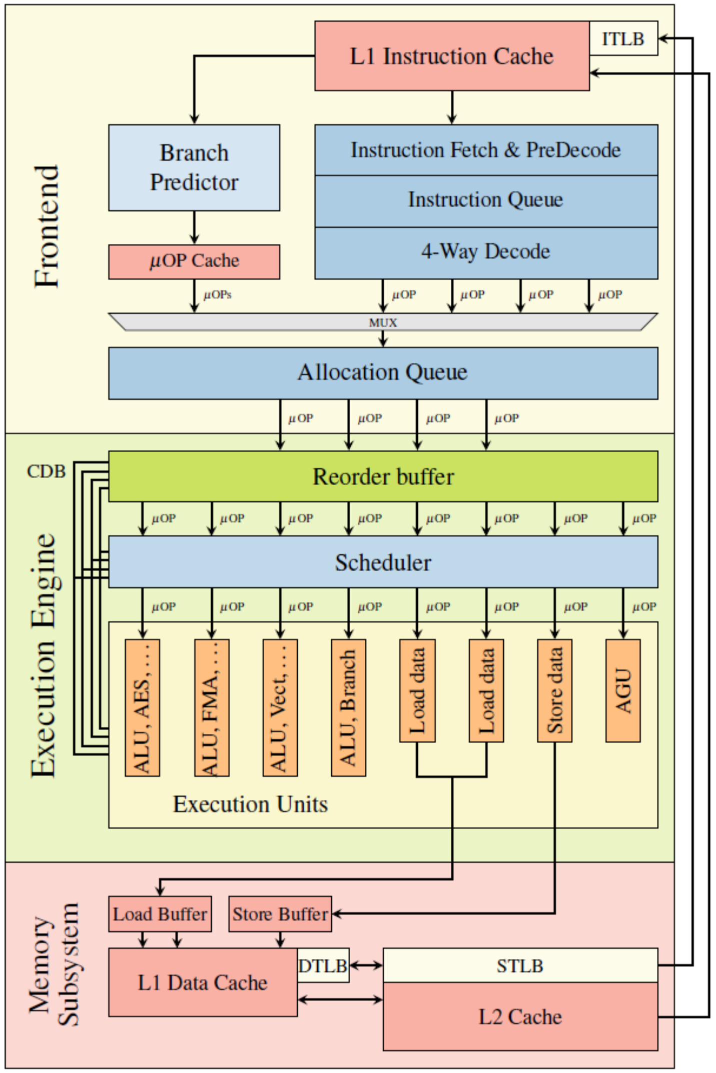
- Issue multiple instructions at a time

Very Long Instruction Word (VLIW)

- **Compiler** encodes multiple instructions into one long instruction that can be executed in parallel by the CPU’s multiple functional units



Out-of-Order Execution



- Instructions are fetched into a queue
- Any instructions whose data (operands) are ready are executed **out-of-order** (subject to data dependency)
- Results are "retired" in-order.

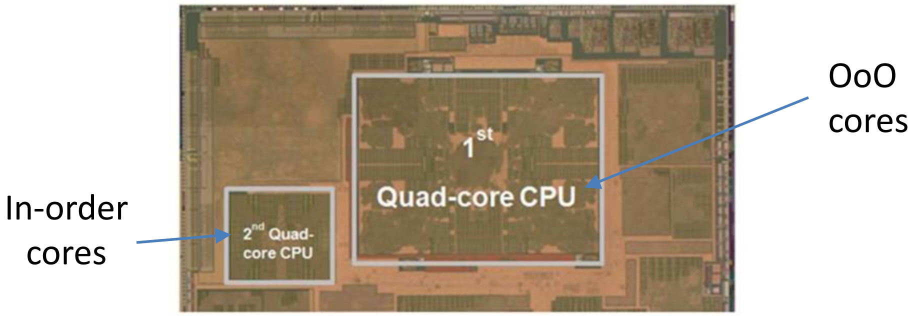
Data Dependency

- Line 1, 3 are independent instructions
- Line 2, 4 are dependent instructions

```
ADD r3, r1, r2          # r3 = r1 + r2

ADD r5, r3, r4          # r5 = r3 + r4

ADD r8, r6, r7          # r8 = r6 + r7

ADD r9, r5, r8          # r9 = r5 + r8
```

In-order vs. Out-of-order

- In-order execution
  - Instructions are fetched/executed/completed in the compiler generated order
  - One instruction stall, all the rest stall
  - Simple, power/energy efficient
- Out-of-order
  - Instructions are fetched in order, executed in out-of-order, and completed in-order
  - Instruction order is dynamically determined by the **hardware**
  - Reduce stall time, achieve higher performance, higher **ILP**
  - Complex, power-hungry



Samsung Exynos 5 Octa Dieshot

Instruction Level Parallelism (ILP)

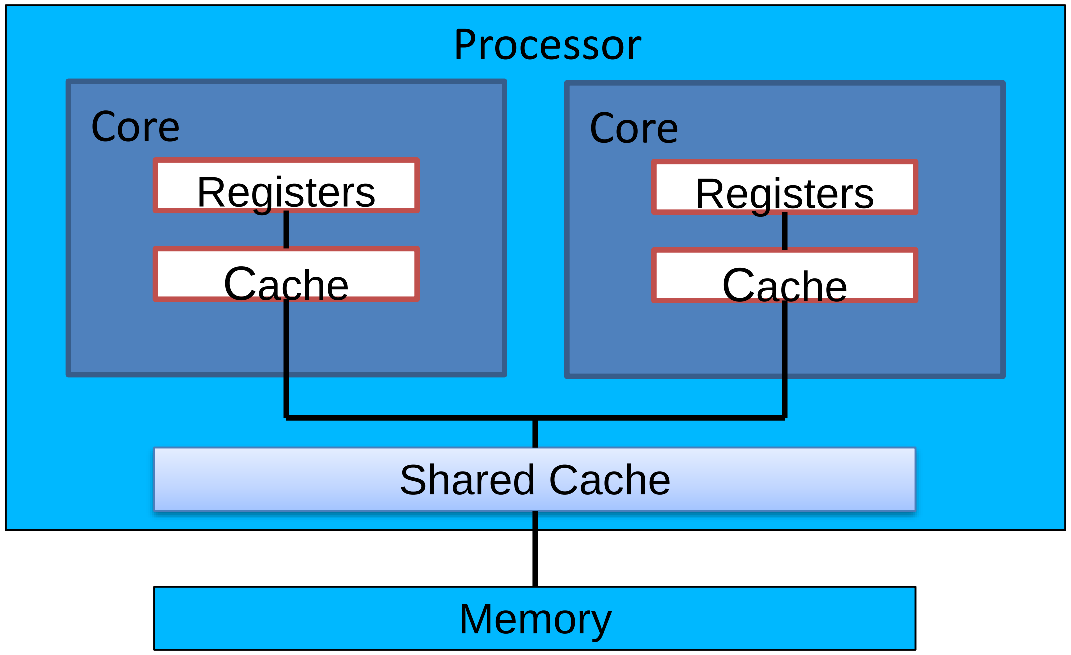
- A program executes a sequence of instructions
- Some of the instructions are independent
- Independent instructions can be executed simultaneously
- Both VLIW and OoO search independent instructions to exploit ILP
- VLIW and OoO differ in that in VLIW, the compiler does the finding, while in OoO hardware scheduler does it.
- Software developers don’t need to do anything

Speculative Execution

```
If (condition)
{
    Do something A1
    Do something A2
    Do something A3
} else {
    Do something B1
    Do something B2
    Do something B3
}
```

- Guess which branch to take.
- Speculatively execute instructions in the likely branch.
- If guessed wrong, squash the results
- Improve performance on average

Multicore CPU



- Parallel processing
  - Can run multiple tasks at a time

Thread Level Parallelism (TLP)

- Execute multiple instruction streams (threads) on different CPU cores
- Software must be explicitly written to take advantage of multiple cores
  - Using threading libraries (e.g., pthread)

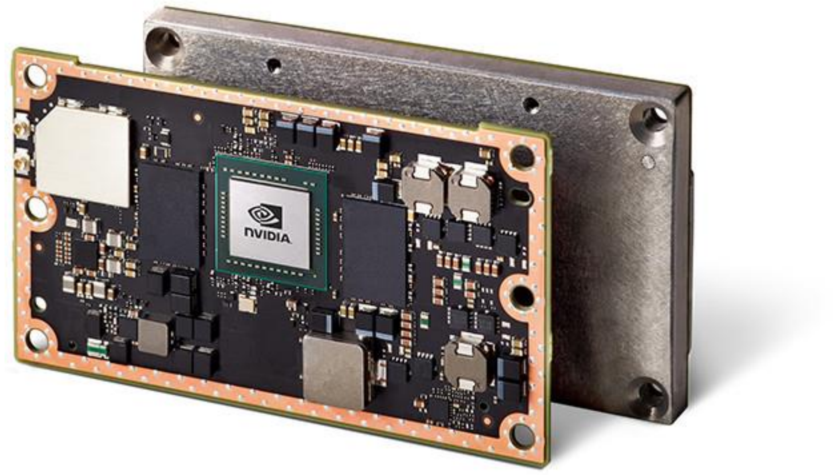
Graphic Processing Unit (GPU)

- GPU
  - Graphic is **embarrassingly parallel** by nature
  - GeForce 6800 (2003): 53GFLOPs (MUL)
  - Some **PhDs** tried to use GPU to do some general purpose computing, but difficult to program

- GPGPU
  - **Ian Buck** (Stanford PhD, 2004) joined Nvidia and created CUDA language and runtime.
  - General purpose: (relatively) easy to program, many scientific applications

Embedded GPU

- NVIDIA Jetson TX2
  - 256 CUDA GPU cores + 4 CPU cores



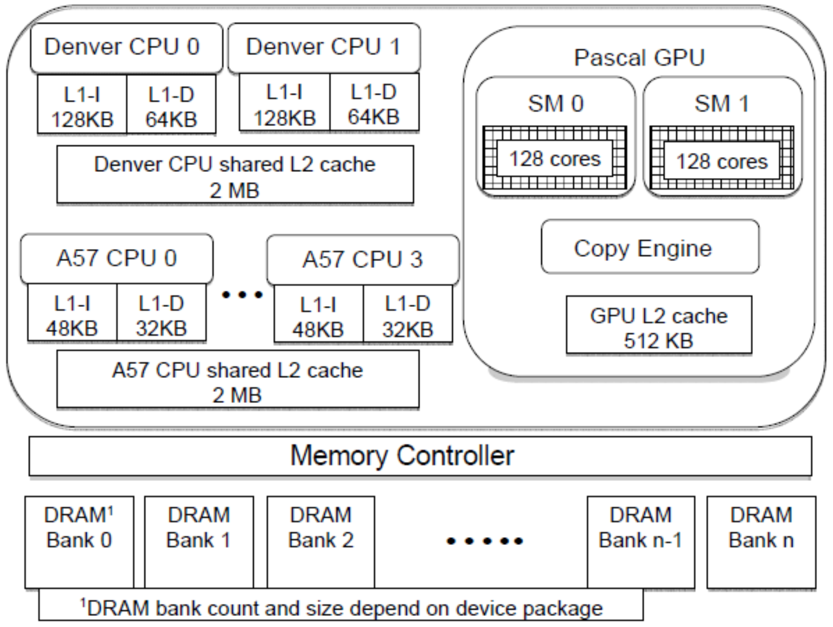
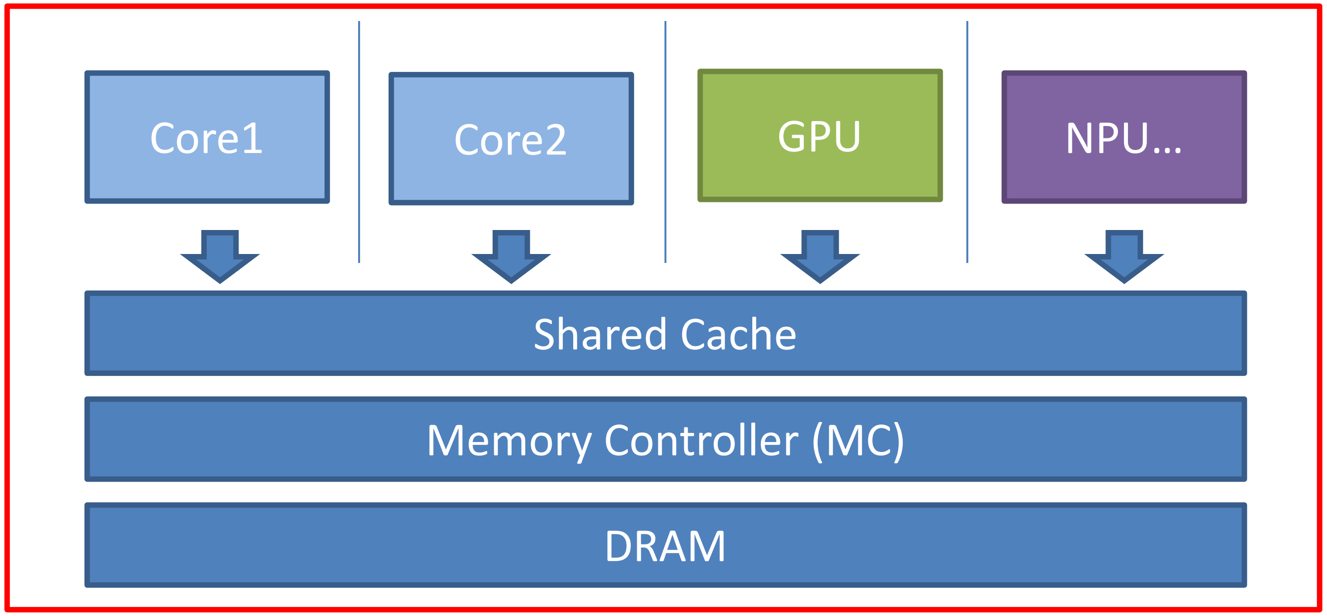


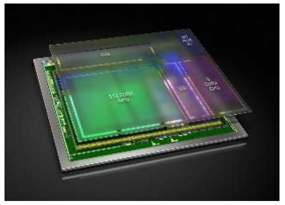
Image credit: T. Amert et al., "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," RTSS17

Data Level Parallelism

- Some data (e.g., vision) are a vector type where its elements can easily processed in parallel
- Single instruction multiple data (SIMD)
  - GPU
  - CPU’s SIMD instructions
    - Intel/AMD MMX, SSE, AVX, AVX2
    - ARM NEON

Heterogeneous Architecture

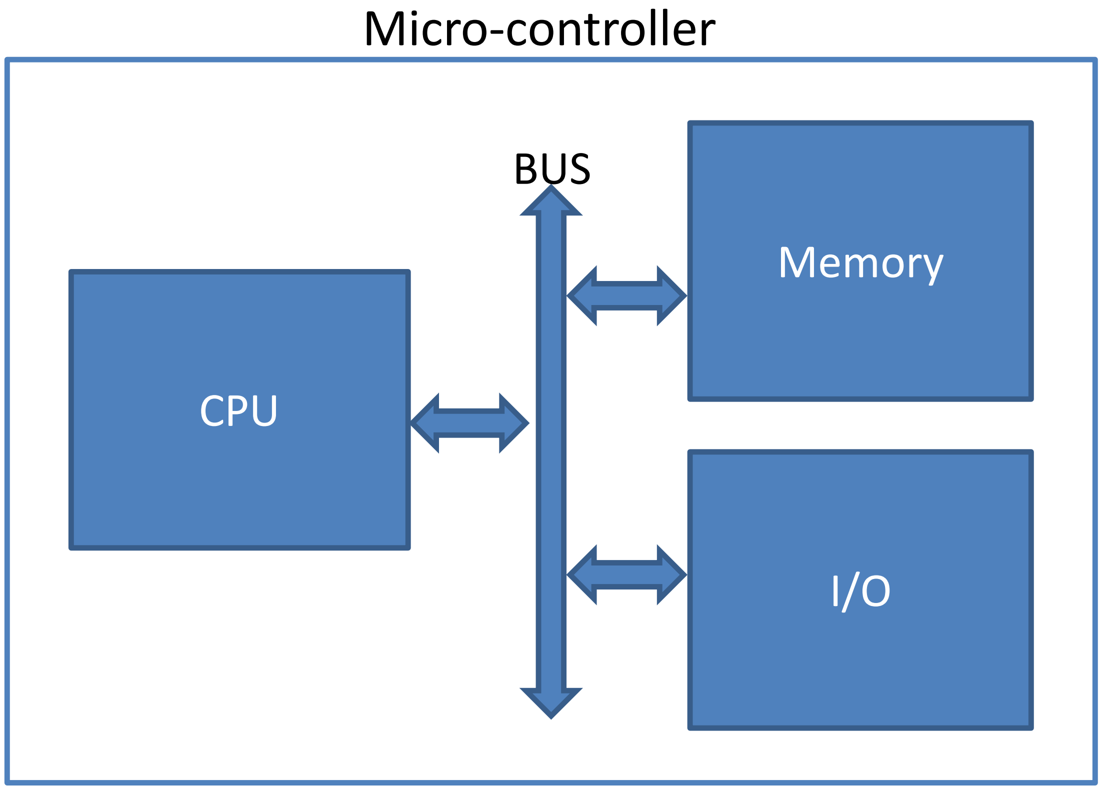




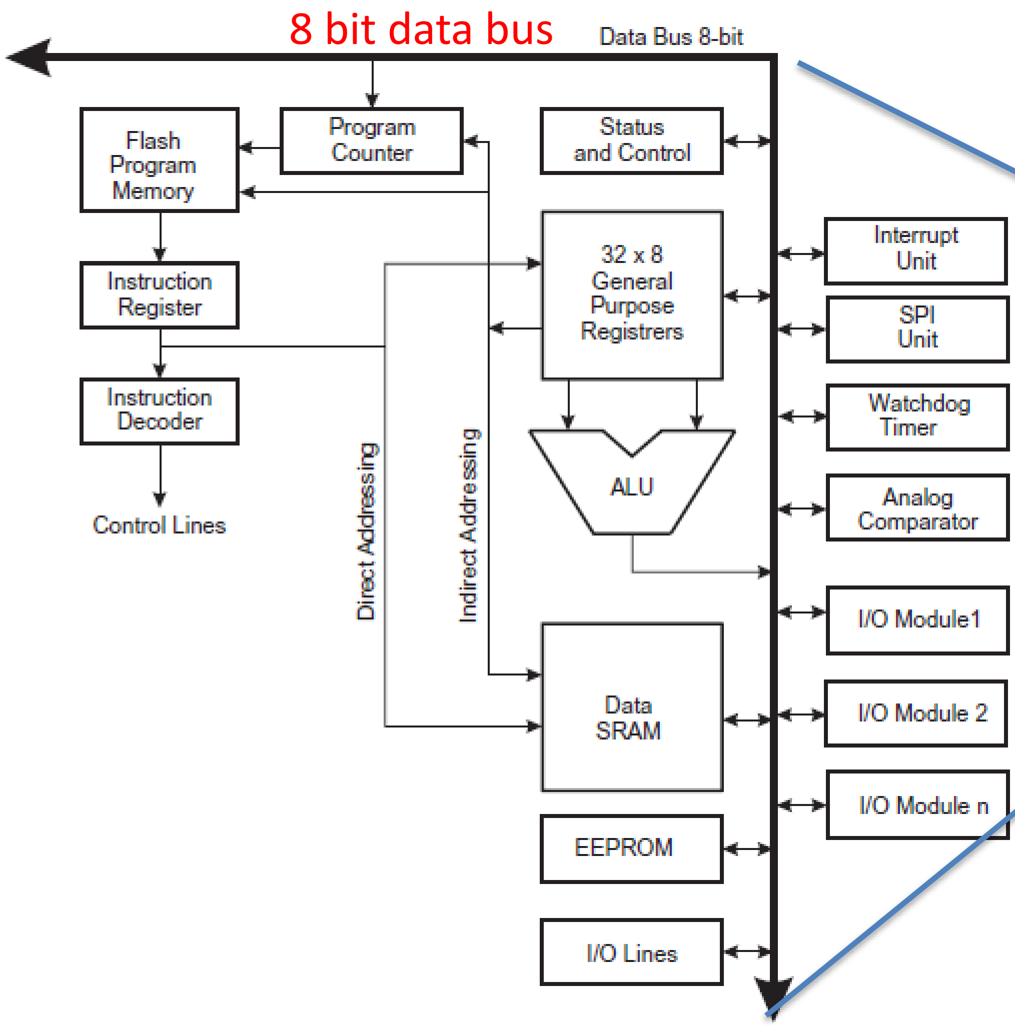
- Integrate multiple cores, GPU, accelerators into a single chip (system-on-a-chip)
- Good performance, size, weight, power

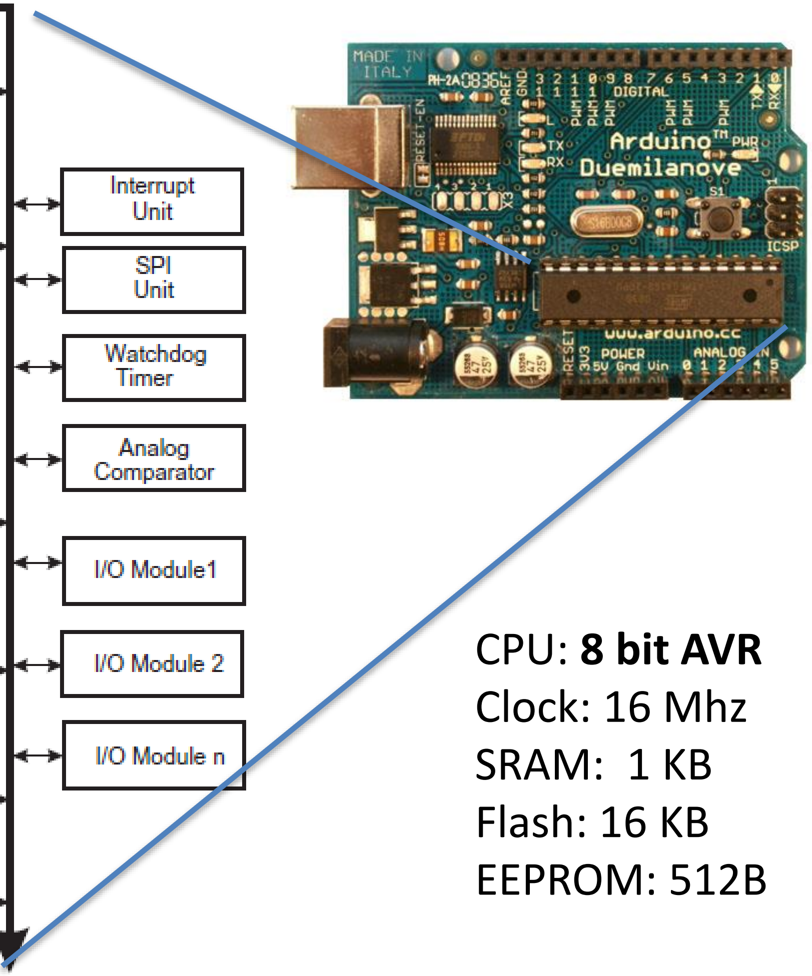
Microcontroller

- A small computer on a single integrated circuit
  - Combine (simple) CPU, RAM, ROM, I/O devices



Atmel ATmega168

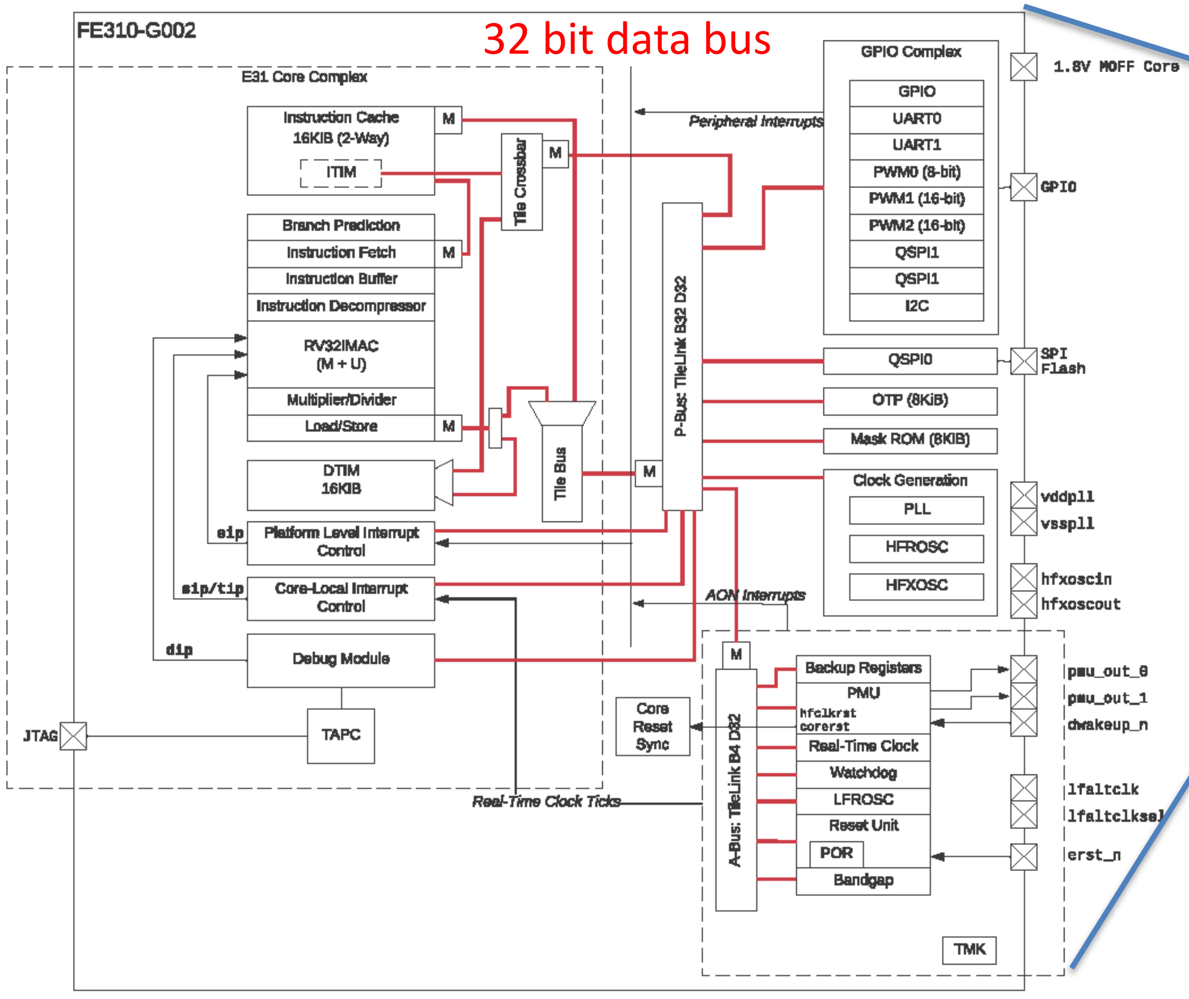


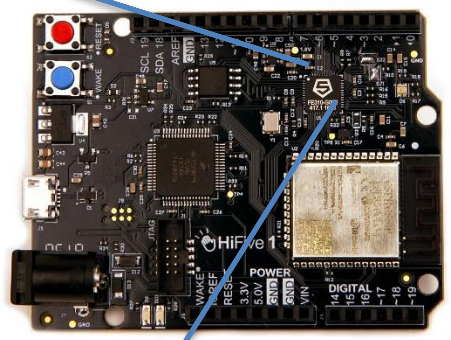


CPU: **8 bit AVR**
Clock: 16 Mhz
SRAM: 1 KB
Flash: 16 KB
EEPROM: 512B

Atmel ATmega168 datasheet

SiFive FE310





CPU: **32 bit RISC-V**
Clock: 320 MHz
SRAM: 16 (D) + 16 (I) KB
Flash: 4MB

Figure 1: FE310-G002 top-level block diagram.

Raspberry Pi 4: Broadcom BCM2711

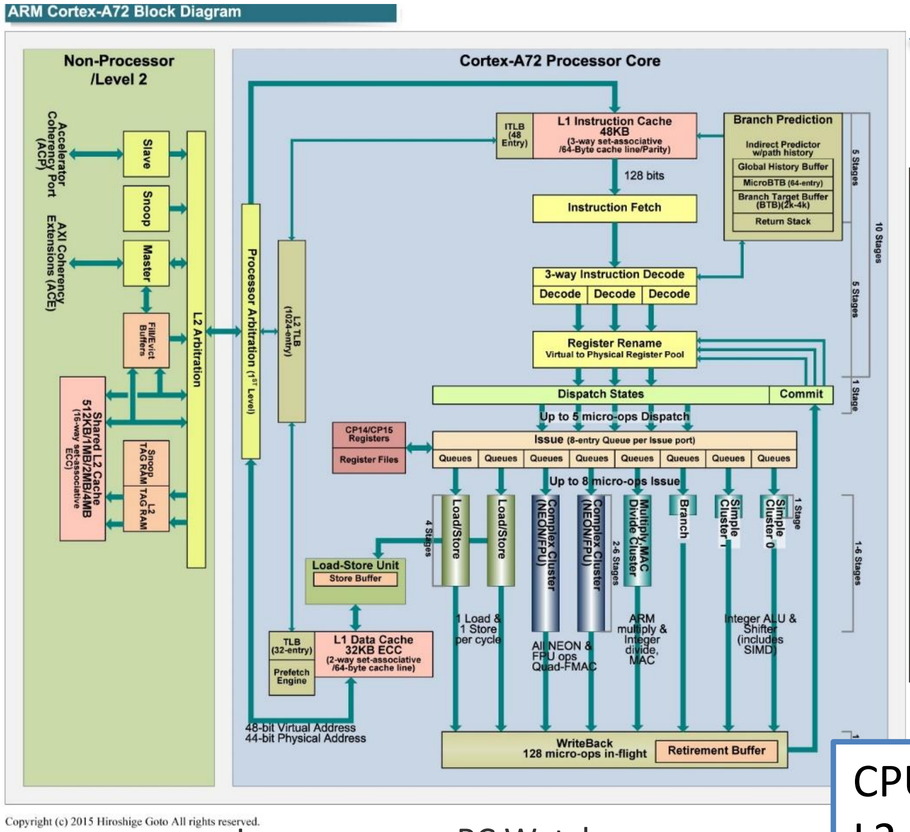
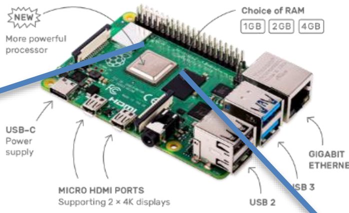
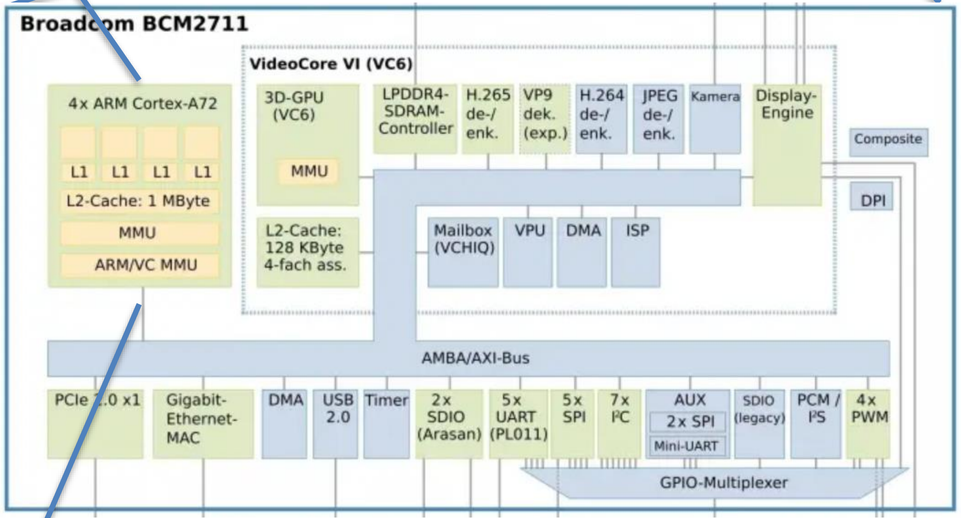


Image source: PC Watch.


NEW More powerful processor
Choice of RAM 1GB 2GB 4GB
USB-C Power supply
MICRO HDMI PORTS Supporting 2 × 4K displays
USB 2
USB 3
GIGABIT ETHERNET



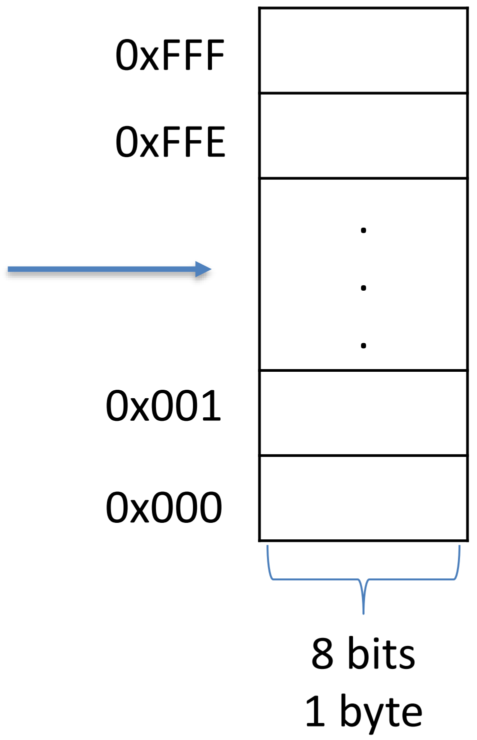
(Bild: ct.de/Maik Merten (CC BY SA 4.0))

CPU: 4x **Cortex-A72@**1.5GHz
L2 cache (shared): 1MB
GPU: VideoCore IV@500Mhz
DRAM: 1/2/4 GB LPDDR4-3200
Storage: micro-SD

Recall: Memory Address

- Byte addressed
  - Minimum unit = 1 byte (8 bits)

What's the size of this memory?

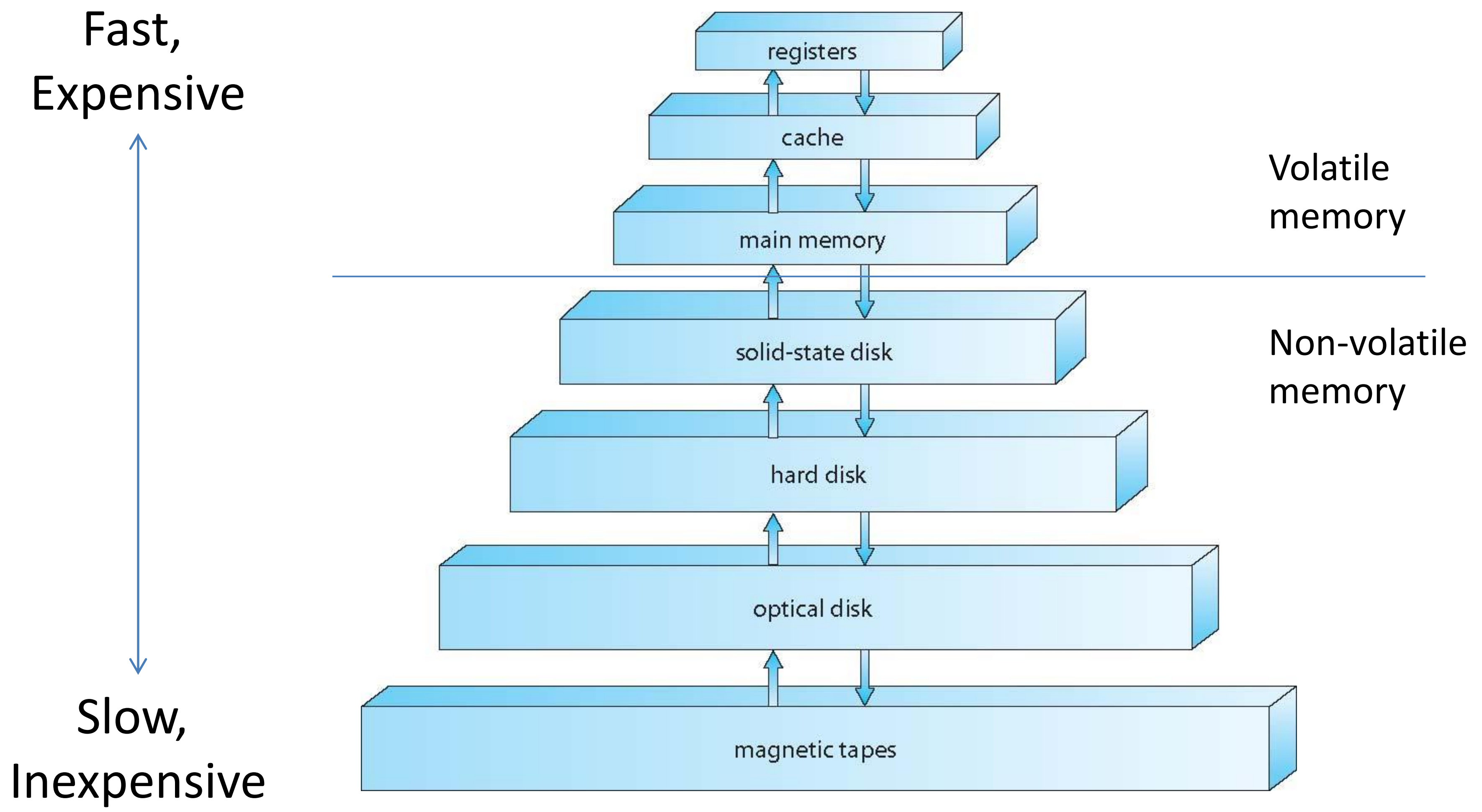


- Maximum addressable memory
  - Depends on the CPU architecture
    - 32 bit CPU: 2^{32} bytes
    - 16 bit CPU: ___ bytes
  - Depends on the platform
    - Some regions are mapped to ram, flash, or I/O devices
    - Some are unmapped.

Memory

- Volatile memory
  - Lose data when power is off
  - **SRAM**
  - **DRAM**
- Non-volatile memory
  - Retain data while power is off
  - Mechanical disk
  - **EPROM, EEPROM, Nor Flash, NAND Flash**
  - **eMMC, Solid state disk (SSD)**
  - PRAM, MRAM, Intel Optane memory

Memory Hierarchy



Memory Hierarchy

- Latency/bandwidth vary significantly

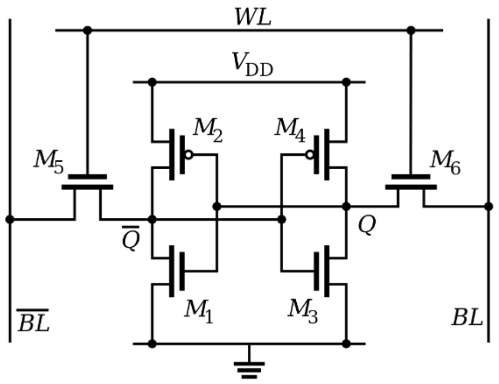
Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

SRAM

- Fast, small, expensive volatile memory
  - Need 6 transistors for one bit
    - Low density = bigger cost
  - C.f. DRAM use one TR for one bit
    - High density = lower cost
- Pros/Cons

  ++ Simplicity, performance, reliability, low idle power

  --- Price, density, high operational power
- Applications
  - On-chip memory, cache, scratchpad, buffers, …



A six-transistor CMOS SRAM cell

Cache vs. Scratchpad Memory

- Both are SRAM
- Differ in the ways they are used
- (CPU) Cache
  - Hardware managed SRAM (invisible to software)
  - Make avg. access speed to main memory faster
- Scratchpad
  - Software managed SRAM
  - Is the “main memory” in microcontrollers

Cache vs. Scratchpad Memory

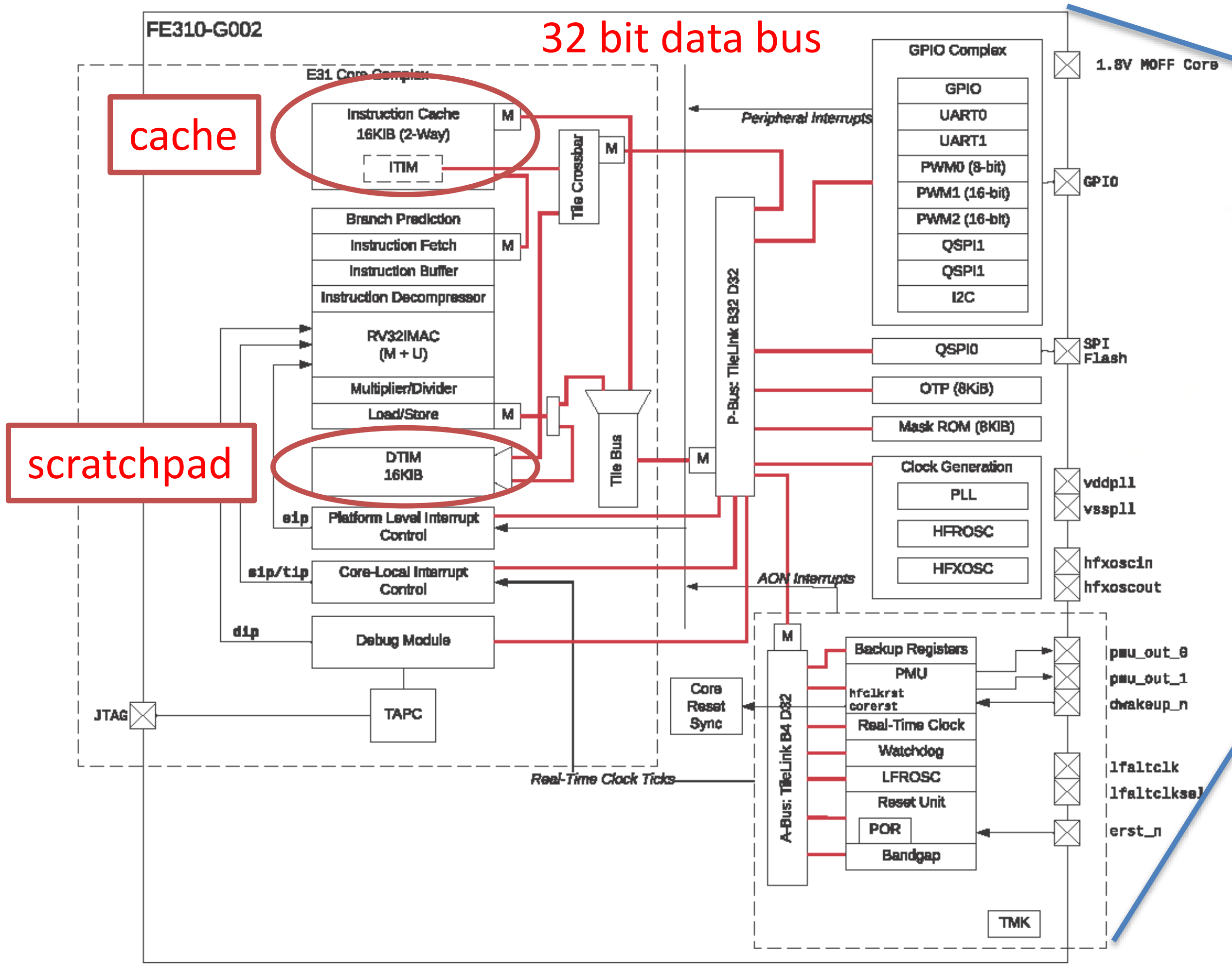
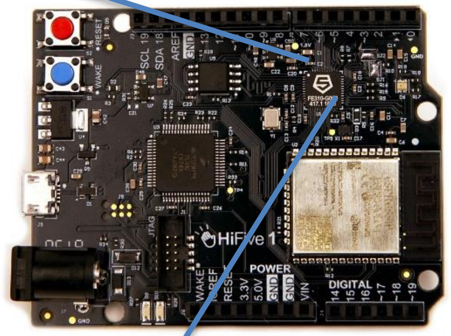


Figure 1: FE310-G002 top-level block diagram.

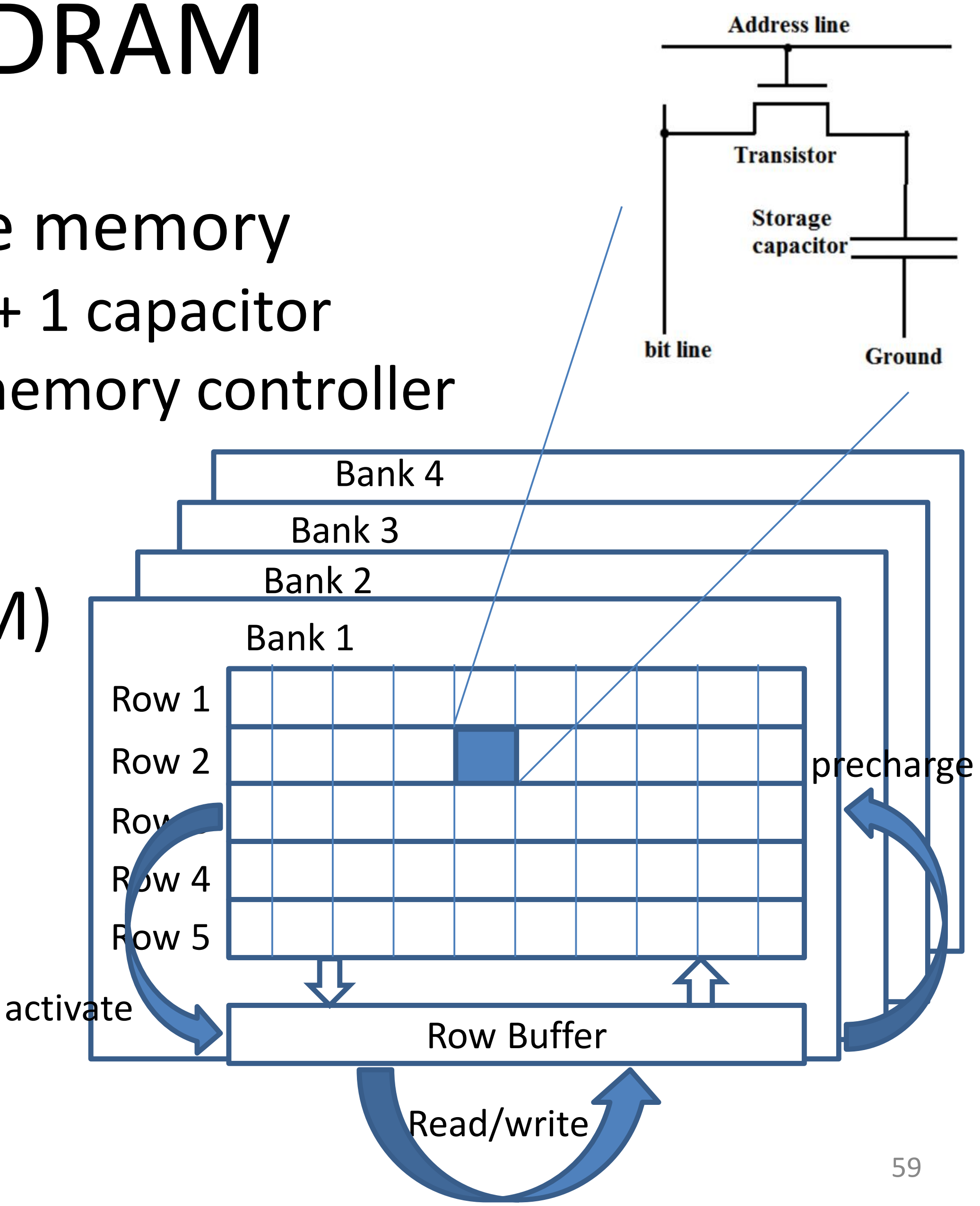
SiFive FE310



CPU: 32 bit RISC-V
Clock: 320 MHz
SRAM: **16 (D)** + **16 (I)** KB
Flash: 4MB

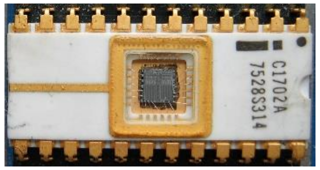
DRAM

- High density volatile memory
  - 1 cell = 1 transistor + 1 capacitor
  - Requires complex memory controller to access it
- Pros/Cons (vs. SRAM)
  - ++ cost, capacity
  - --- complexity, latency
- Applications
  - Main memory



MROM/EPROM/EEPROM

- Mask ROM (MROM)
  - Non-reprogrammable read only memory
- EPROM
  - Erasable programmable read-only memory
  - Erase by using ultraviolet light
- EEPROM
  - Electrically erasable programmable read-only memory
  - Can be reprogrammed many times eclectically.
- All are often used to store small read-only code
  - E.g., Boot loader

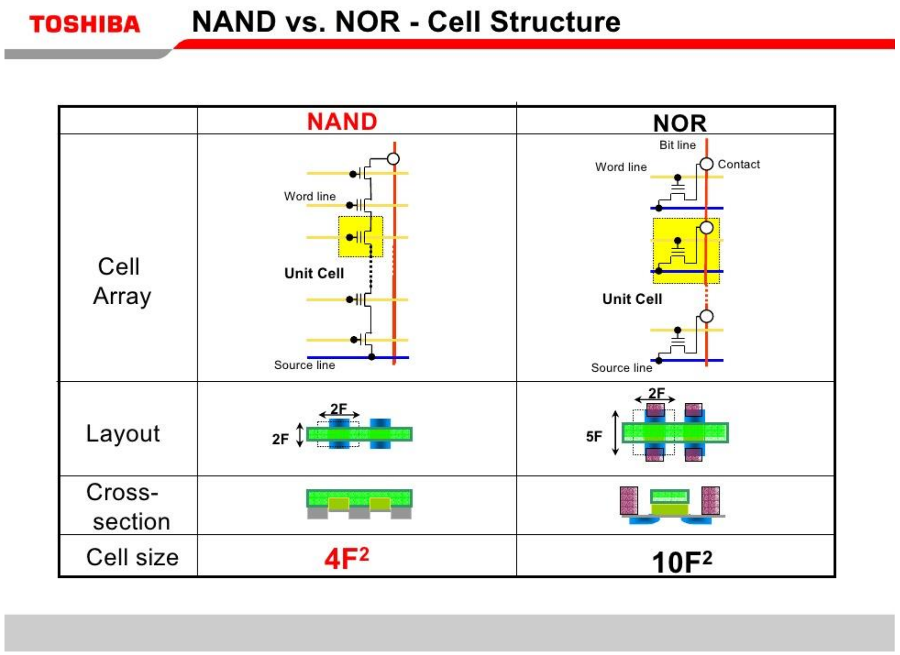


Flash Memory

- NOR Flash
  - Byte addressable
  - CPU can directly access it (e.g., to fetch instructions)
  - Can be re-programmed, but reprogramming is slow
  - Primarily used as code storage
- NAND Flash
  - Non byte-addressable (require controller circuit)
  - CPU cannot directly access it
  - Offer larger capacity at a lower cost
  - Primarily used as data storage (disk replacement)

KU THE UNIVERSITY OF KANSAS

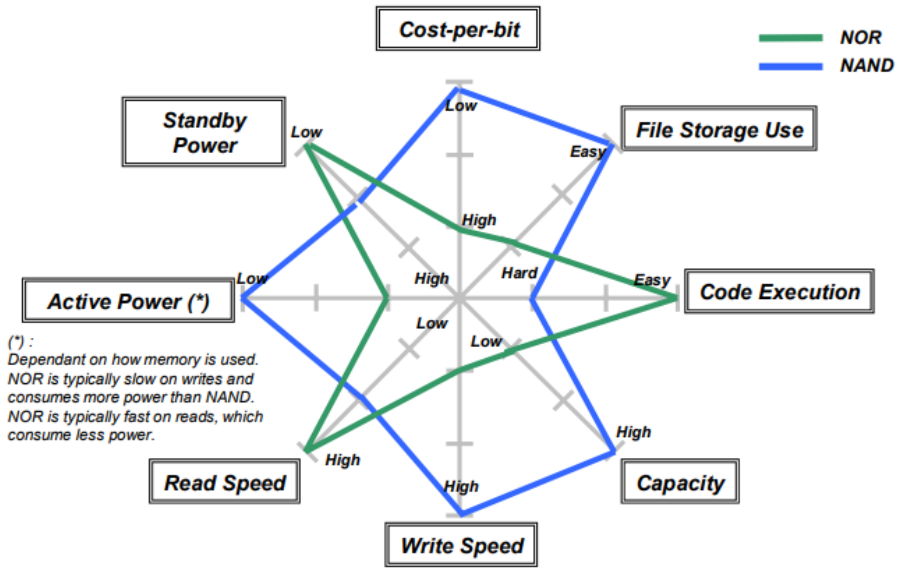
NAND vs. NOR Flash



Word line
Unit Cell
Source line
Bit line
Contact
2F
5F

TOSHIBA NAND vs. NOR - Cell Structure

	NAND	NOR
Cell Array		
Layout		
Cross-section		
Cell size	$4F^2$	$10F^2$



(*) :
Dependant on how memory is used.
NOR is typically slow on writes and consumes more power than NAND.
NOR is typically fast on reads, which consume less power.

Toshiba NAND vs. NOR Flash Memory Technology Overview

NAND Flash Memory Chip

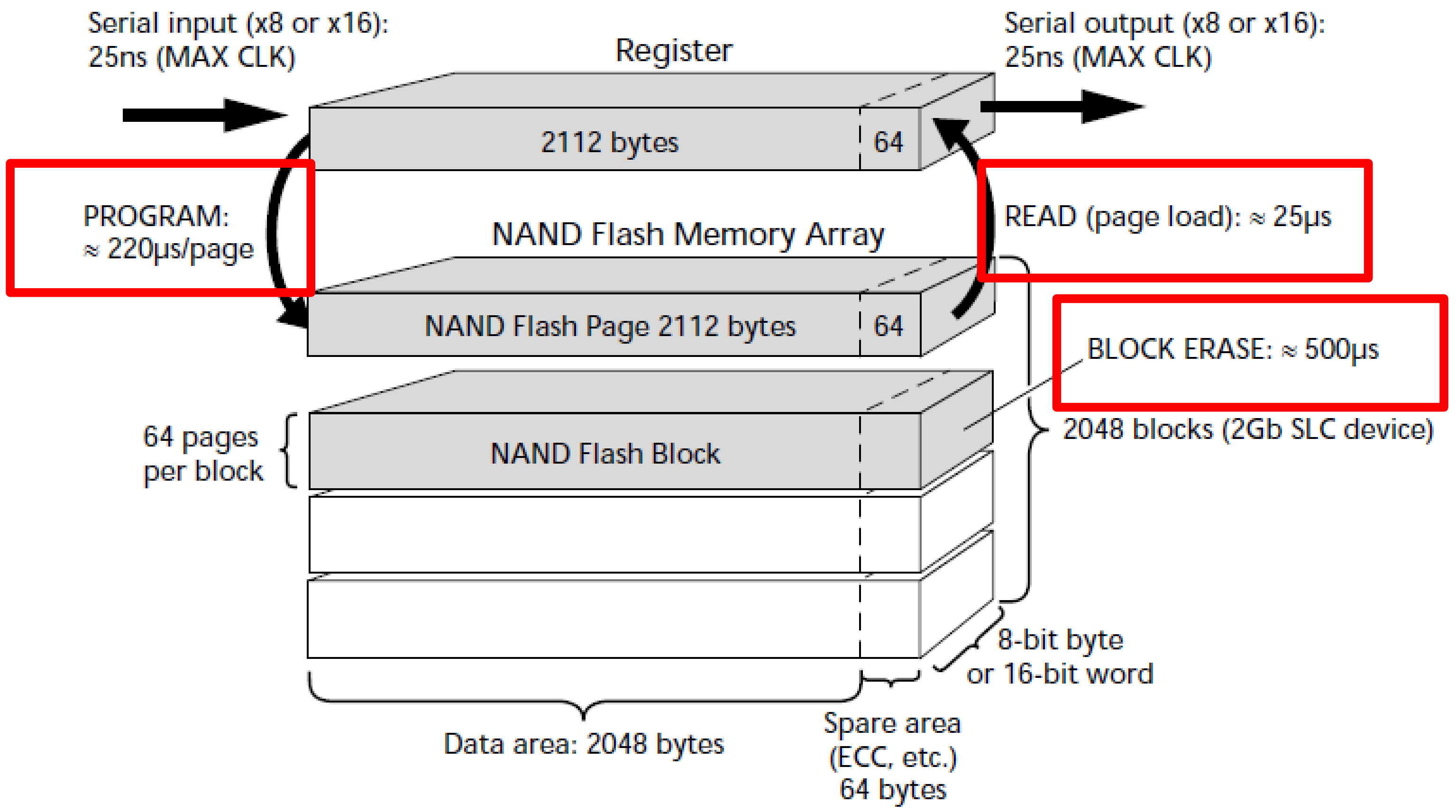
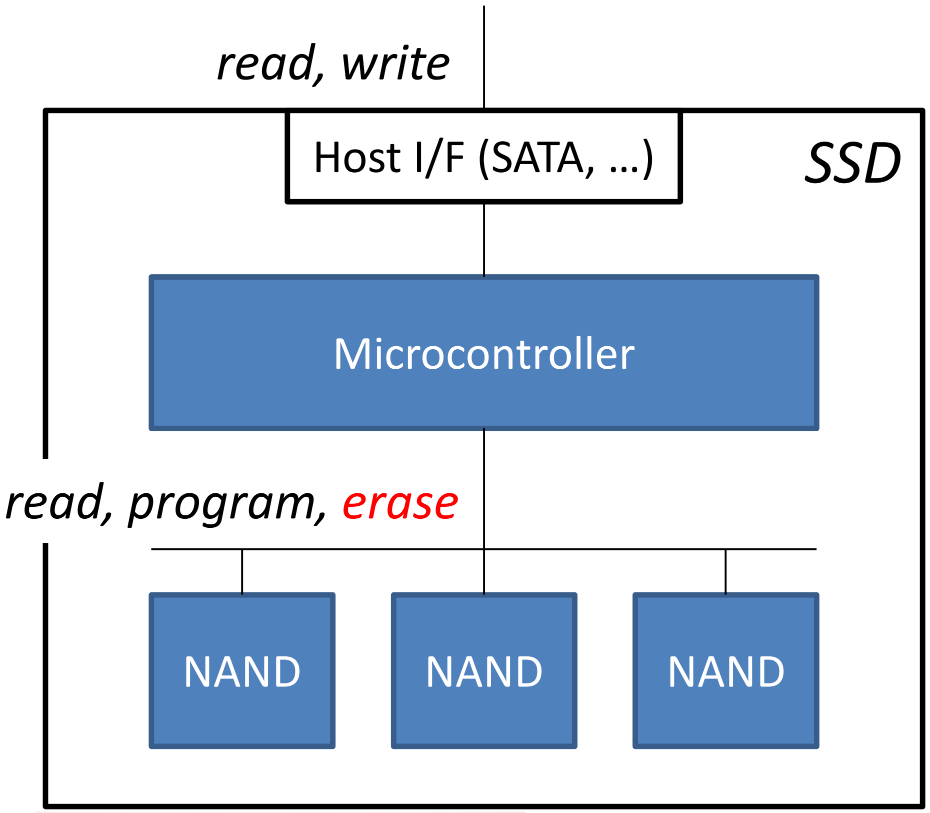
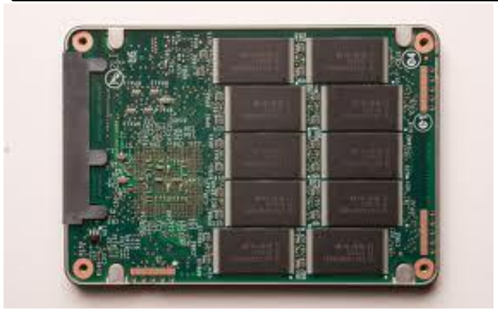


Figure source: Micron, “TN-29-19: NAND Flash 101”, 2010

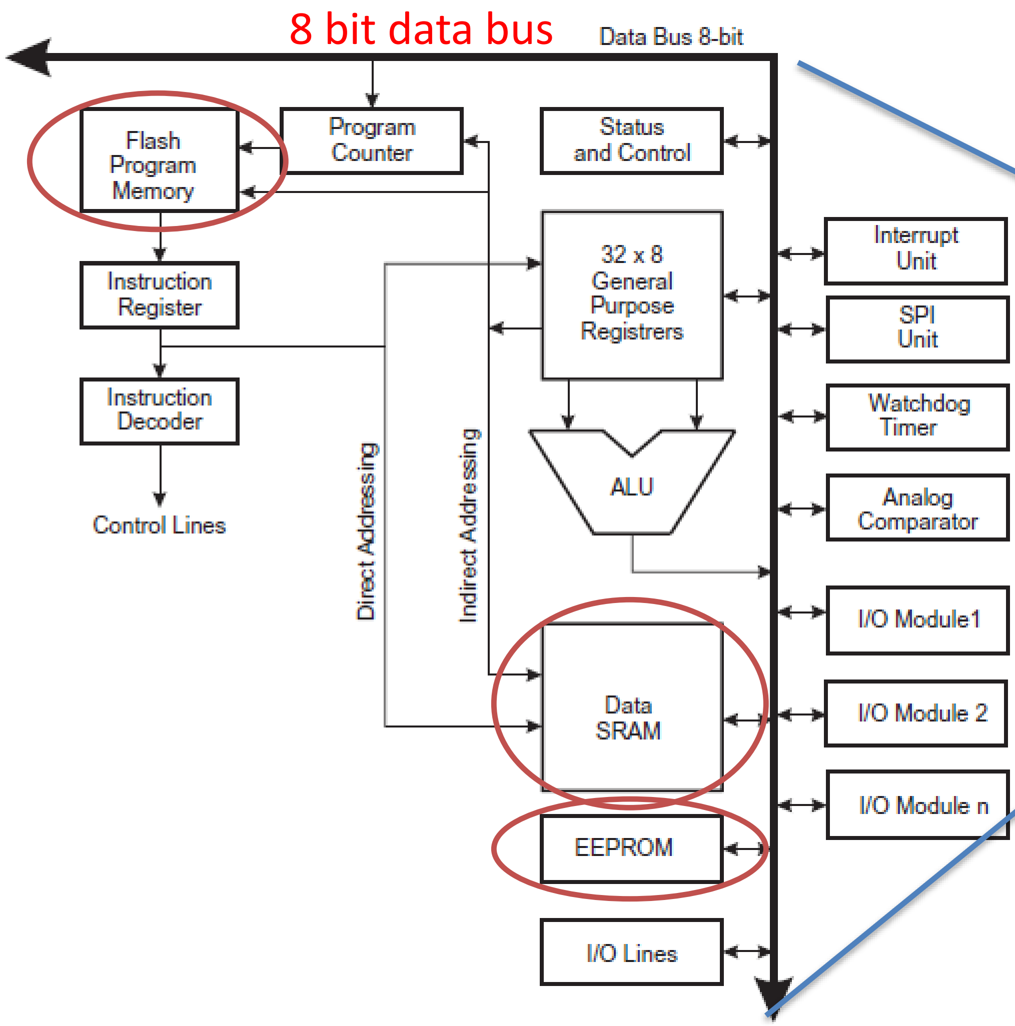
Solid-State Disk (SSD) / eMMC

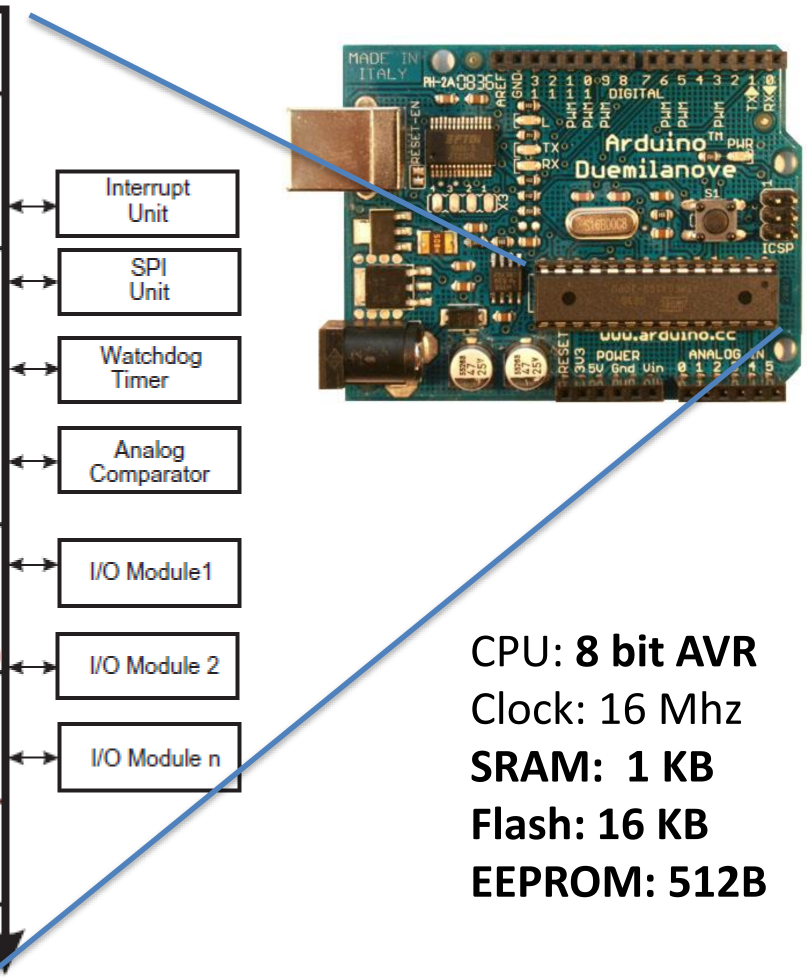


- Flash Translation Layer (FTL)
  - S/W running on the controller
  - Provides disk abstraction
- No seek time
  - No mechanical part
- High capacity
  - SSD: ~1000s GB
  - eMMC: ~10s GB
- High bandwidth
  - SSD: ~1000s MB/s
  - eMMC: ~100s MB/s



Atmel ATmega168





CPU: **8 bit AVR**
Clock: 16 Mhz
SRAM: 1 KB
Flash: 16 KB
EEPROM: 512B

Atmel ATmega168 datasheet

SiFive FE310

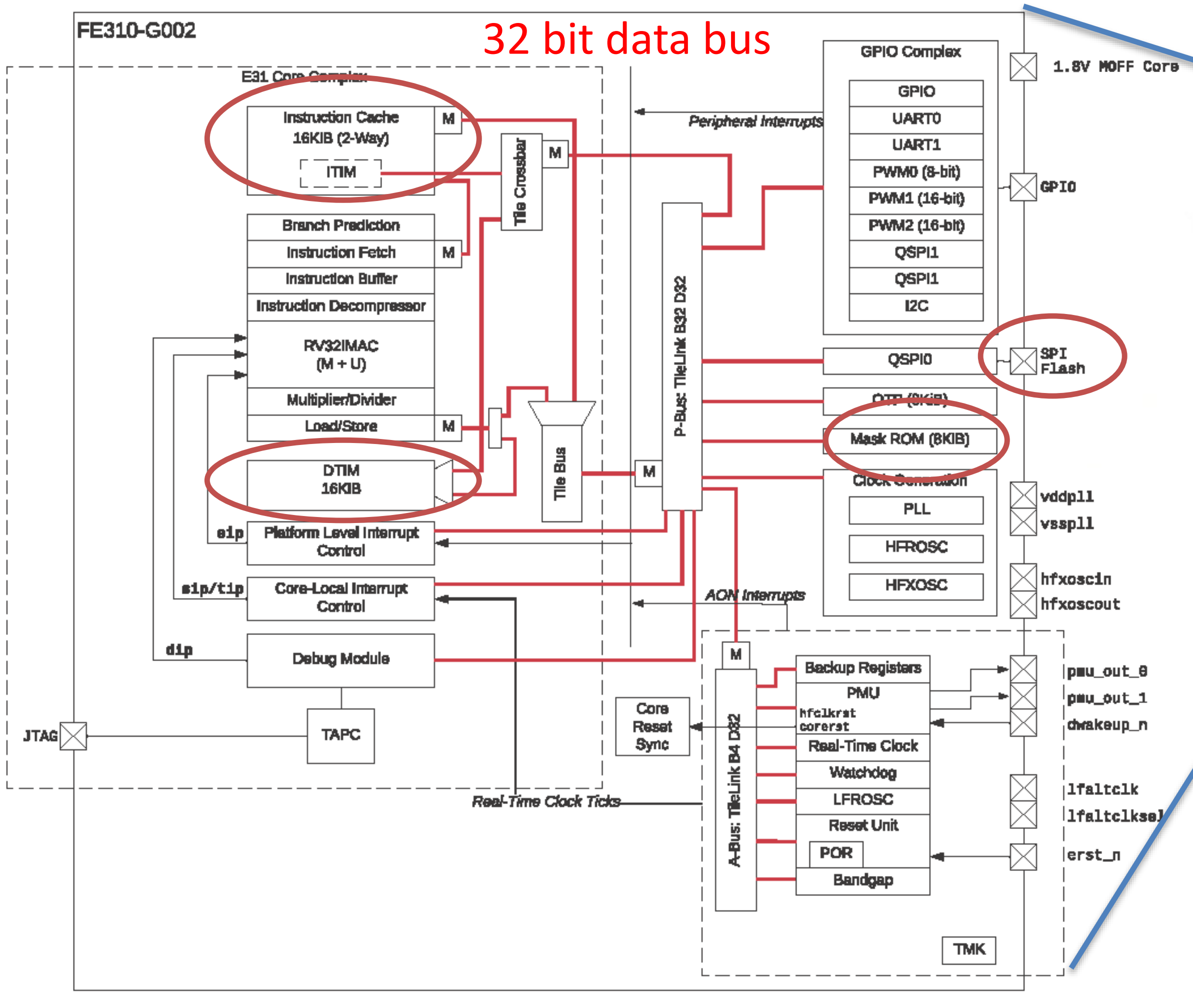
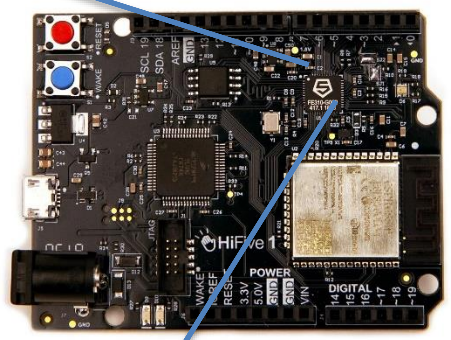


Figure 1: FE310-G002 top-level block diagram.



CPU: 32 bit RISC-V

Clock: 320 MHz

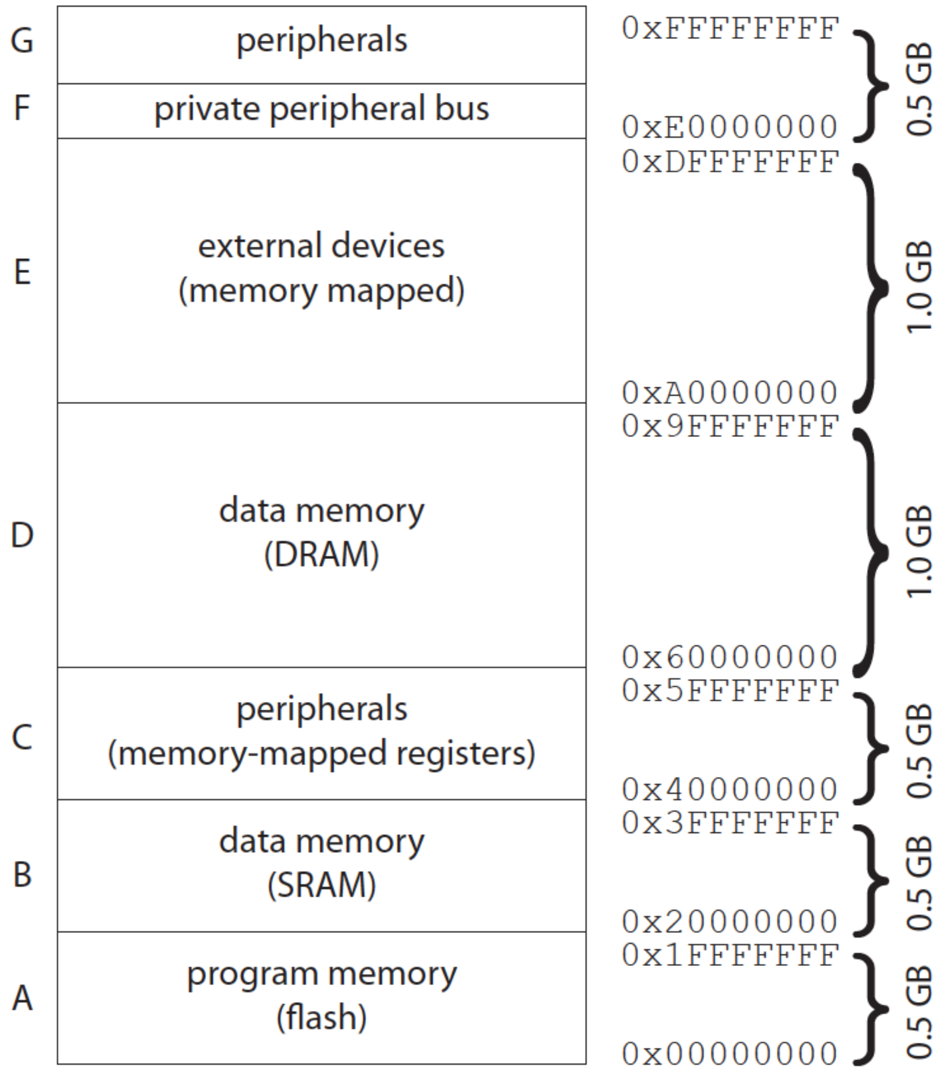
SRAM: 16 (D) + 16 (I) KB

Flash: 4MB

MaskROM: 8KB

Memory Map

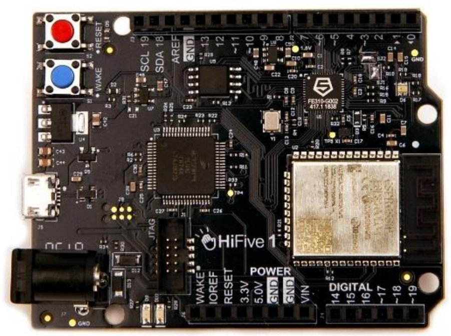
- CPU's view of the physical memory
- Segregated with multiple regions
- Each region can be connected to a different bus
- Not all regions are actually mapped.
- Each CPU may have different mappings



An ARM Cortex-M3's memory map
(from E. A Lee of UCB)

Memory Map of SiFive FE310

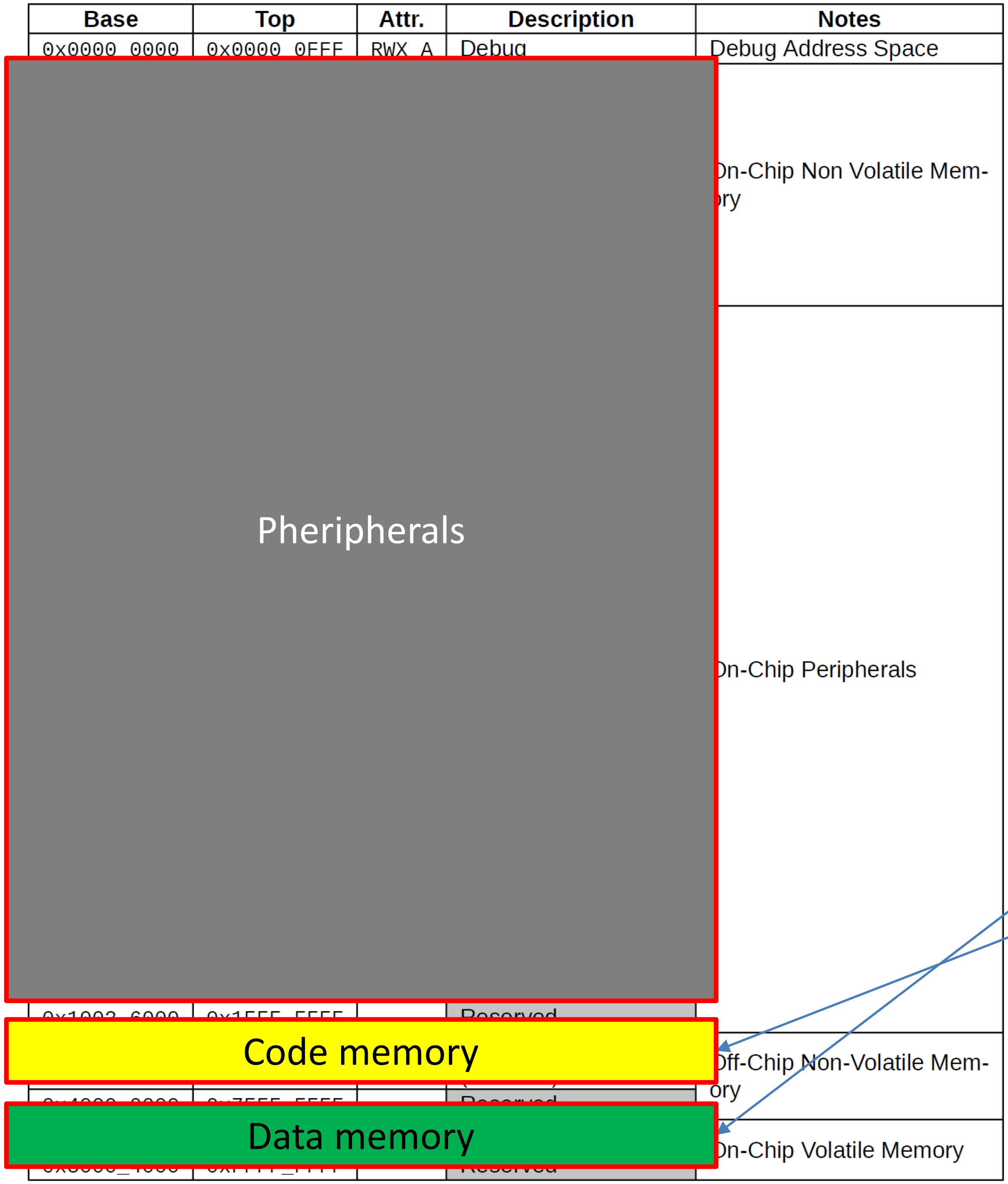
Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	



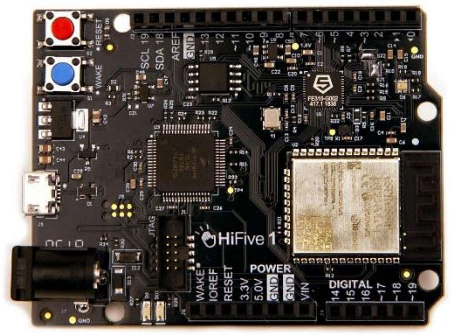
CPU: 32 bit RISC-V
Clock: 320 MHz
SRAM: 16 KB (D)
Flash: 4MB

What about the rest of the address space?

Memory Map of SiFive FE310



Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
				On-Chip Non Volatile Memory
				On-Chip Peripherals
0x1003_6000	0x1FFF_FFFF		Reserved	
				Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
				On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	



CPU: 32 bit RISC-V
Clock: 320 MHz
SRAM: 16 KB (D)
Flash: 4MB

Summary

- Instruction set architecture (ISA)
  - A contract between software and hardware
  - Defines an abstract machine
- CPU
  - Implements an ISA
  - Pipelining, superscalar, in-order, out-of-order
  - ILP, TLP, DLP
- Memory
  - Volatile memory
  - Non-volatile memory
  - Memory hierarchy

Acknowledgements

- Some slides are based on the material originally developed by
  - James C. Hoe (CMU) for 18-447 on RISC-V ISA